relevance

The State of the DSL Art in Ruby

Glenn Vanderburg
Relevance, Inc.
glenn@thinkrelevance.com

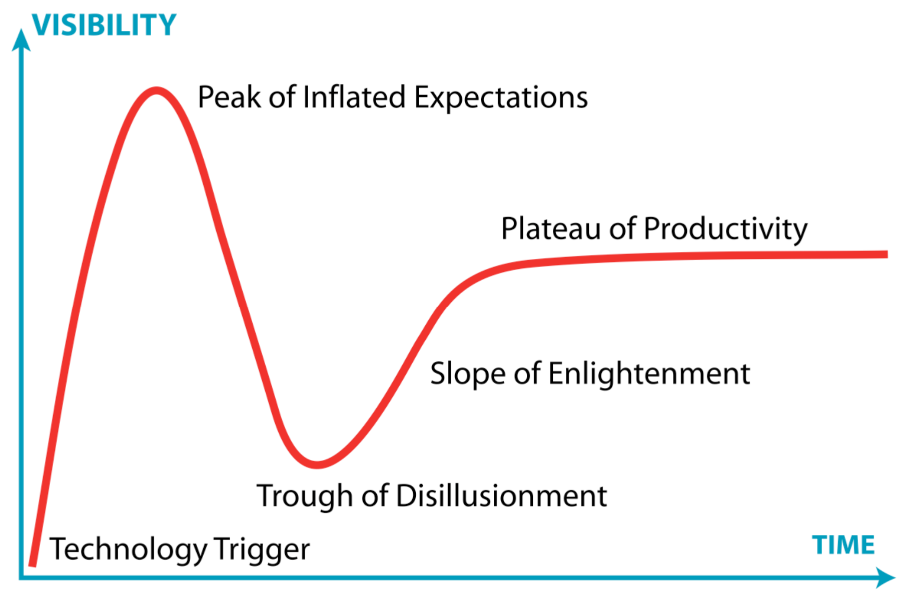
relevance

run›code›run

State of the Art

- Focus on internal DSLs
- Rubyists pushing the leading edge
- Ruby's features make it a good match

DSLs Are Overhyped



Evolution

Origins

- The whole idea of internal DSLs apparently originated in Lisp.

In Lisp, you don't just write your program down toward the language, you also build the language up toward your program.

—Paul Graham

Lisp

```
(task "warn if website is not alive"
      every 3 seconds
      starting now
      when (not (website-alive? "http://example.org"))
      then (notify "admin@example.org" "server down!"))
```

Functional Languages

- Many functional languages lend themselves to internal DSLs.
- Internal DSLs were a design goal of Haskell.

```
keepleft (p :>: ps)
    | keepleft p = case partitionFL keepleft ps of
                   a :> b -> p :>: a :> b
    | otherwise  = case commuteWhatWeCanFL (p :> ps) of
                   a :> p' :> b -> case partitionFL keepleft a of
                                   a' :> b' -> a' :> b' +>+ p' :>: b
```

Ruby, Out of the Box

- **Declaring object properties:**
```ruby
attr_reader   :id, :age
attr_writer   :name
attr_accessor :color
```
- **Not syntax, just methods (defined in Module)**

Here’s How To Do It

```ruby
class Module
  def attr_reader (*syms)
    syms.each do |sym|
      class_eval %{def #{sym}
                     @#{sym}
                   end}
    end
  end
end
```

relevance

And attr_writer ...

```
class Module
  def attr_writer (*syms)
    syms.each do |sym|
      class_eval %{def #{sym}= (val)
                     @#{sym} = val
                   end}
    end
  end
end
```

relevance

Mathieu Bouchard’s X11 Library

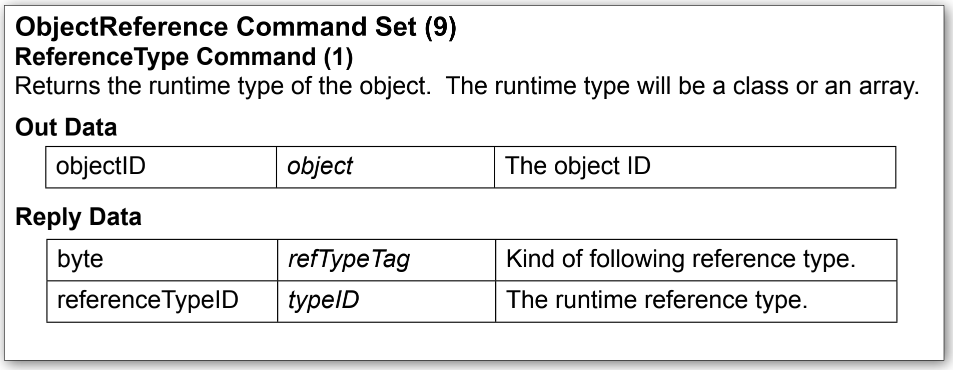
DestroySubwindows
window: WINDOW
Errors: **Window**

ChangeSaveSet
window: WINDOW
mode: {**Insert**, **Delete**}
Errors: **Match**, **Value**, **Window**

ReparentWindow
window, *parent*: WINDOW
x, *y*: INT16
Errors: **Match**, **Window**

```
def_remote :close_subwindows, 5, [Self]
def_remote :change_save_set, 6, [Self,
      [:change_type, ChangeMode, :in_header]]
def_remote :reparent, 7, [Self,
      [:parent, Window],
      [:point, Point]]
```

Styles Have Changed



ObjectReference Command Set (9)
ReferenceType Command (1)
Returns the runtime type of the object. The runtime type will be a class or an array.

Out Data

objectID	*object*	The object ID

Reply Data

byte	*refTypeTag*	Kind of following reference type.
referenceTypeID	*typeID*	The runtime reference type.

```
JDWP.add_command_set :ObjectReference, 9 do |set|
  set.add_command :ReferenceType do |cmd|
    cmd.description = "Returns the runtime type of the object.  The ..."
    cmd.out_data :objectID, :object, "The object ID"
    cmd.reply_data :byte, :refTypeTag, "Kind of following reference type."
    cmd.reply_data :referenceTypeID, :typeID, "The runtime reference ..."
  end
end
```

Dave’s Summer Project

- **Dave Thomas, RubyConf 2002**
  “How I Spent My Summer Vacation”

```
class RegionTable < Table
  table "region" do
    field autoinc,      :reg_id,        pk
    field varchar(100), :reg_name
    field int,          :reg_affiliate, references(AffiliateTable,
                                                   :aff_id)
  end
end
```

Rails

```
class CreateRegions < ActiveRecord::Migration
  def self.up
    create_table :regions do |t|
      t.string     :name
      t.belongs_to :affiliate
    end
  end

  def self.down
    drop_table :regions
  end
end
```

```
class Region < ActiveRecord::Base
  belongs_to :affiliate
end
```

What Makes Internal DSLs Special?

General-Purpose Constructs

- Types
- Literals
- Declarations
- Expressions
- Operators
- Statements
- Control Structures

Specialized Constructs

- Most DSLs also deal with things you don’t usually find in general-purpose languages:
  - Context-dependence
  - Commands and sentences
  - Units
  - Large vocabularies
  - Hierarchy

Contexts

```
Interval = new_struct(:start, :end) do
  def length
    self.start - self.end
  end
end
```

```
create_table :regions do |t|
  t.string     :name
  t.belongs_to :affiliate
end
```

Implementing Contexts

```
def new_struct (*args, &block)
  struct_class = Class.new
  struct_class.class_eval { attr_accessor *args }
  # define initialize method
  struct_class.class_eval(&block) if block_given?
  struct_class
end

def create_table(table_name, options = {})
  table_definition = TableDefinition.new(options)

  yield table_definition

  if options[:force] && table_exists?(table_name)
    drop_table(table_name, options)
  end

  execute table_definition.to_sql
end
```

Commands and Sentences

- Multipart, complex statements or declarations.
- Example: Dave’s database library

```
field autoinc,:reg_id,        pk
field int,    :reg_affiliate, references(AffiliateTable,
                                         :aff_id)
```

- Let’s take that apart.

Commands and Sentences

```
field autoinc,   :reg_id,   pk
```

- Overall, it’s just a method call.
- The first parameter—the type—is a method call.
- The second parameter is a symbol.
- Additional parameters are method calls.

Modern Sentences

- **From a Rails project:**

```
has_many :favorites, :order => :position,
         :conditions => {:state => 'public'}

has_many :roles, :through => :projects, :uniq => true

validates_length_of :login, :within => 3..40,
                    :on => :create

validates_presence_of :authority, :if => :in_leadership_role
              :message => "must be authorized for leadership."
```

Implementing Sentences

```
has_many :roles, :through => :projects,
                 :uniq => true
```

- **Once again, it's just a method call.**
- **First parameter is a symbol (exploits naming conventions).**
- **Second parameter is an implicit hash.**
- **Implementation pattern:**

```
def declaration(thing, options={})
  # validate and process options
  # create and store metadata
  # define custom methods
end
```

Units

- General-purpose languages deal with scalars
- Most domain-specific languages deal with quantities expressed using units.
- From Rails:

```
# A time interval
3.years + 13.days + 2.hours
# Four months from now, on a Monday
4.months.from_now.next_week.monday
```

Implementing Units

- The easy part: classes representing quantities.
  - Use operator overloading if it makes sense!
  - May require mixed-base arithmetic.
- Next: natural expression

```
# Augment the built-in classes
class Numeric
  def minutes; self * 60; end
  def hours; self * 60.minutes; end
  # etc.
end
```

Large Vocabularies

- Sometimes you need a command structure that's essentially open-ended.
- Roman numerals:

```
Roman.CCXX
Roman.XLII
```

- XmlMarkup class:

```
xm.em("emphasized")
xm.a("A Link", :href => "http://example.com/")
xm.target(:name => "compile", "option" => "fast")
```

Large Vocabularies

- Override `method_missing`.
- Here's the Roman numeral method:

```
class Roman
  def self.method_missing (method_id)
    str = method_id.id2name
    roman_to_int(str)
  end
end
```

- Be careful! Difficult bugs lurk here.

Hierarchy

- **XmlMarkup again:**

```
xml.html {
  xml.head {
    xml.title("History")
  }
  xml.body {
    xml.h1("Header")
    xml.p("paragraph")
  }
}
```

Implementing Hierarchy

- **Called from `method_missing`:**

```
def element (elem_name, opts={})
  write "<#{elem_name}#{encode_opts(opts)}"
  if block_given?
    puts ">#{yield}</#{elem_name}>"
  else
    puts "/>"
  end
end
```

- **You could use `instance_eval` to avoid typing "`xml.`" before every call. But don't.**

Perspective

Ruby's DSL Strengths

- Dynamic and reflective
- Blocks allow writing new control structures
- Declarations are executable
- Built-in contexts
- Only slightly less malleable than Lisp (no macros)

Syntax Matters

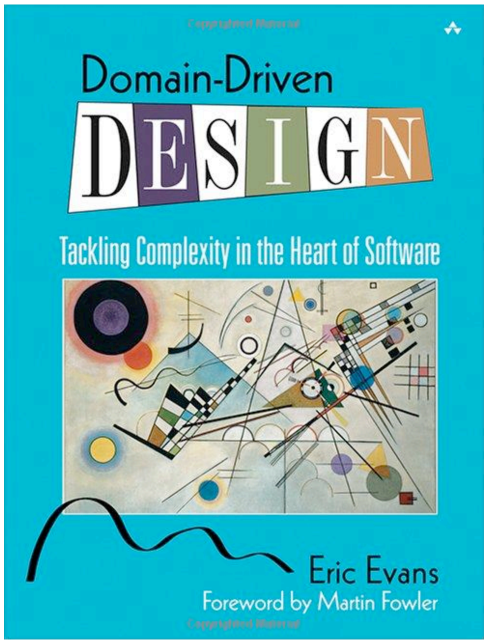
- Ruby’s syntax is great for DSLs
  - Neutral and unobtrusive
  - Enough to distinguish different kinds of constructs
  - Not enough to complicate straightforward statements
  - Most punctuation is optional

DSLs != Magic Pixie Dust

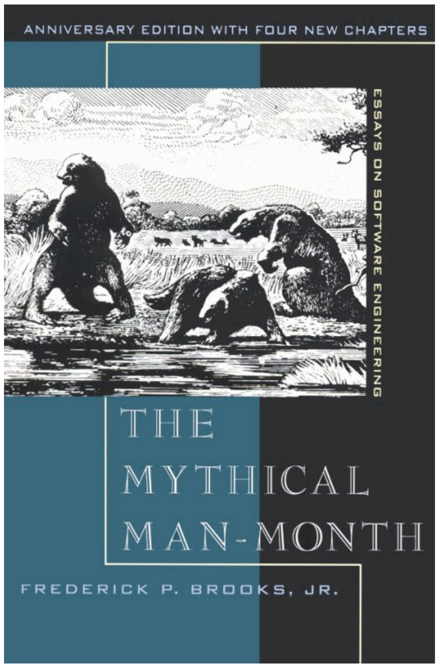
- DSLs don’t magically make your software better.
- They can be overused.
- They don’t always make code clearer.

Photo credit: Tracey Parker

Domain Language


Domain-Driven
DESIGN
Tackling Complexity in the Heart of Software
Eric Evans
Foreword by Martin Fowler

Essence and Accident


ANNIVERSARY EDITION WITH FOUR NEW CHAPTERS
ESSAYS ON SOFTWARE ENGINEERING
THE
MYTHICAL
MAN-MONTH
FREDERICK P. BROOKS, JR.

Good Software Design

- **Eliminate as much of the accidental complexity as possible.**
- ***Separate* the rest.**

Language and program evolve together. Like the border between two warring states, the boundary between language and program is drawn and redrawn, until eventually it comes to rest along the mountains and rivers, the natural frontiers of your problem. In the end your program will look as if the language had been designed for it. And when language and program fit one another well, you end up with code which is clear, small, and efficient.

—Paul Graham

```ruby
def create
  @post = Post.new(params[:post])

  respond_to do |format|
    if @post.save
      flash[:notice] = 'Post was successfully created.'
      format.html { redirect_to @post }
      format.xml  { render :xml => @post,
                           :status => :created,
                           :location => @post }
    else
      format.html { render :action => "new" }
      format.xml  { render :xml => @post.errors,
                           :status => :unprocessable_entity }
    end
  end
end
```

DSLs != Polyjuice Potion

- Create DSLs with “normal” constructs:
  - Objects and methods
  - Reflection
  - Openness
- Test them normally as well.
- Culture is limiting extreme uses.

Photo credit: Jo Naylor

Barrier to Understanding?

- The language is *for people who understand the domain.*
- The things that are implicit are *accidental complexity.*
- Learning the language *aids in understanding the domain.*

Good API Design

- Creating DSLs with everyday constructs is *powerful.*
- You can refactor to them as you find duplication, complexity.
- Internal DSLs are just a part of good API design in Ruby.

Library design is language design.

—Bell Labs Proverb

DSLs Are Cool!

- But what are they *really* good for?
- Solid domain modeling
- More and better options when refactoring
- Customer communication
- Clean separation of essence and accident