Metaprogramming Ruby

Domain-Specific Languages for Programmers

Glenn Vanderburg
glv@vanderburg.org

O'Reilly Open Source Convention
August 1 - 5, 2005

Metaprogramming

- My definition: "programming your programming."
- Change the way you program in your programming language.
- Transform your general-purpose language …
- Make it a domain-specific language.
- Program in a language designed for the problem you're solving.

Lisp

- Metaprogramming seems to have originated in Lisp.

Lisp is a programmable programming language.

—John Foderaro

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 isn't the only programmable language.

Ruby

- Rubyists have been rediscovering metaprogramming.
- Ruby style and idioms are still changing and adapting.
- Rails leverages metaprogramming heavily.
  - To great effect!
- Ruby is a natural for metaprogramming.

What Makes Ruby Good for Metaprogramming

- Dynamic and reflective
- Everything is open to change
- Blocks allow writing new control structures
  - Add continuations to really get fancy
- Most declarations are executable statements
- Only slightly less malleable than Lisp (no macros)
- A fantastic syntax!
  - Neutral and unobtrusive
  - Enough to distinguish different *kinds* of constructs
  - Not enough to complicate straightforward statements

Built-In Examples

- Declaring object properties:
  attr_reader :id, :age
  attr_writer :name
  attr_accessor :color
- Not syntax, just methods (defined in Module)
- Let's go see how they're written!
- Oh. They're written in C.

But Here's How It Would Be Done

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

And attr_writer …

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

Mathieu Bouchard's X11 Library

- From the X11 Protocol Specification:

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**

- From X11/XID.rb

```
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

- From the Java Debug Wire Protocol specification:

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.

- From rubyjdwp (jdwp_spec.rb):

```
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
```

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 int,            :reg_affiliate, references(AffiliateTable,
                                                     :aff_id)
    field varchar(100), :reg_name
  end
end
```

How To Think About Metaprogramming

- Defining new constructs for your programming language.
- OK, but … constructs to do what?
- Whatever your domain-specific language (DSL) needs to do.

Another Way To Think About Metaprogramming

- A new set of conceptual tools for eliminating duplication (and other smells) from your code.

Conventional Constructs

- DSLs need all of the general-purpose things:
  - Types
  - Literals
  - Declarations
  - Expressions
  - Operators
  - Statements
  - Control Structures

Other 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

Expressions

- Damian Conway's state transition diagram:

```
fsm_string = FSM.new("STRING") do
   states :in_string, :ignore_next
   events '"' => :quote, '\\' => :backslash

   start  >-------------- quote      >---- in_string
   in_string >---------- backslash  >---- ignore_next
   ignore_next  >-----  other      >---- in_string
   in_string >---------- quote      >---- accept
end
```

- On second thought … no, that's just wrong.

Contexts

- Establishing a context for a set of statements
  - Constrain those statements to the context
  - Multiple, concise operations on the context
- What's a context? A *scope*.

```
Struct.new("Interval", :start, :end) do  # from Ruby 1.9
   def length
      @start - @end
   end
end
```

- From rake:

```
task :test do
   ruby %{-Ilib -e 'Dir["test/test*.rb"].each {|fn| load fn}'}
end
```

Implementing Contexts

- Struct is written in C, but here's how to do it in Ruby:

```
class Struct
   def initialize (*args, &block)
      struct_class = # define struct using args
      struct_class.class_eval(&block) if block_given?
   end
end
```

- Modified code from Rake:

```
def task (*args, &block)
   t = Task.new
   # process and store dependencies …
   t.actions = block if block_given?
   # block is called later, when task is fired.
end
```

Implementing Contexts

- Here's another example, from the Systir system testing tool:

```
add_user {
    name "Charles"
    password "hello123"
    privileges normal
}
```

- Implementing that:

```
def add_user (&block)
    u = User.new
    # User class has name, password, privileges methods
    u.instance_eval(&block) if block_given?
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.

Implementing Commands and Sentences

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

Implementing Commands and Sentences

```
field(autoinc,  :reg_id,          pk)
```

- Overall, it's just a method call.
  - Optional parentheses are very handy for sentences.

Implementing Commands and Sentences

```
field(autoinc,  :reg_id,          pk)
```

- Overall, it's just a method call.
  - Optional parentheses are very handy for sentences.
- The first parameter—the type—is a method call.
  - def autoinc; return FieldType::AutoInc.instance; end
  - Restricts (mostly) arguments to predefined alternatives.

Implementing Commands and Sentences

```
field(autoinc,  :reg_id,          pk)
```

- Overall, it's just a method call.
  - Optional parentheses are very handy for sentences.
- The first parameter—the type—is a method call.
  - def autoinc; return FieldType::AutoInc.instance; end
  - Restricts (mostly) arguments to predefined alternatives.
- The second parameter is a symbol.
  - Infinite number of possible field names.
  - Ruby symbols restrict syntax to good name-like strings.

Implementing Commands and Sentences

```
field(autoinc,  :reg_id,          pk)
```

- Overall, it's just a method call.
  - Optional parentheses are very handy for sentences.
- The first parameter—the type—is a method call.
  - def autoinc; return FieldType::AutoInc.instance; end
  - Restricts (mostly) arguments to predefined alternatives.
- The second parameter is a symbol.
  - Infinite number of possible field names.
  - Ruby symbols restrict syntax to good name-like strings.
- Additional parameters are method calls.
  - Options, constraints
  - Methods return objects that encode constraints

Units

- General-purpose languages deal with scalars
  - Programs must maintain knowledge about units.
- 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.
  - Don't forget to support math on quantities.
  - Use operator overloading if it makes sense!
  - May require mixed-base arithmetic. (Time certainly does.)

- 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
```

- Jim Weirich's XmlMarkup class (used in Rails):

```
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:

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

Implementing Hierarchy

- Called from method_missing:

```
def element (elem_name) {
    if block_given? then
        puts “<#{elem_name}>”
        yield
        puts “</#{elem_name}>”
    else
        puts “<#{elem_name}/>”
    end
end
```

- You could use instance_eval to avoid typing “xml.” before every call. But don’t.

Resources

- These slides:
  http://www.vanderburg.org/Speaking/Stuff/oscon05.pdf
- *The Ruby Way*, by Hal Fulton
  http://hypermetrics.com/rubyhacker/coralbook/
- *Programming Ruby*, by Dave Thomas and Andy Hunt
  http://www.pragmaticprogrammer.com/titles/ruby/
- Use the source!
  - Most of the examples shown are open-source systems.