Practical Metaprogramming: Modeling Thought

Steven G. Harms

2011-08-12

Contact Me!

Steven G. Harms

Physically: San Francisco, CA
Email: `lsrcv@sgharms.oib.com`
Twitter / GitHub: `sgharms`
G+

Austin

Austin has many charms. This is from Torchy's

What We'll Cover

Practical Metaprogramming: "Modeling Thought"

Goals

1. Reflect upon how we learn MP in the Ruby community

Goals

1. Reflect upon how we learn MP in the Ruby community
2. Demonstrate MP's ubiquity: you can't *not* learn this

Goals

1. Reflect upon how we learn MP in the Ruby community
2. Demonstrate MP's ubiquity: you can't *not* learn this
3. Advise when you should reach for the MP "hammer"

Goals

1. Reflect upon how we learn MP in the Ruby community
2. Demonstrate MP's ubiquity: you can't *not* learn this
3. Advise when you should reach for the MP "hammer"
4. Provide a real-world example of thinking in terms of MP

Goals

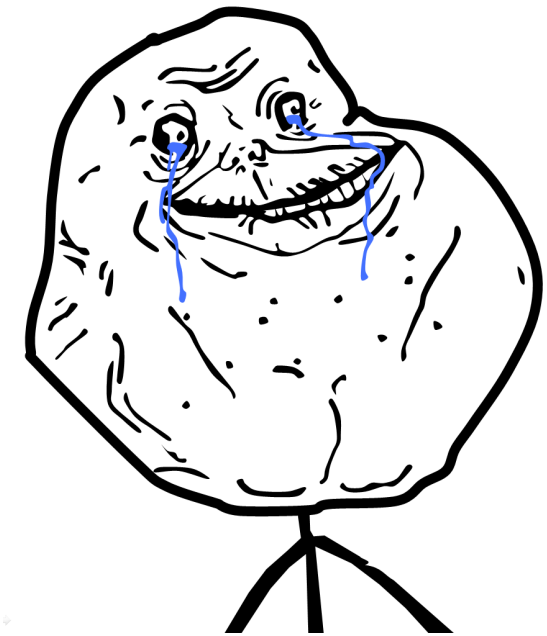
1. Reflect upon how we learn MP in the Ruby community
2. Demonstrate MP's ubiquity: you can't *not* learn this
3. Advise when you should reach for the MP "hammer"
4. Provide a real-world example of thinking in terms of MP
5. Give you the confidence to use MP **boldly**

Intermission

INTERMISSION

Socially Awkward Penguin

End Slide if Everyone Leaves



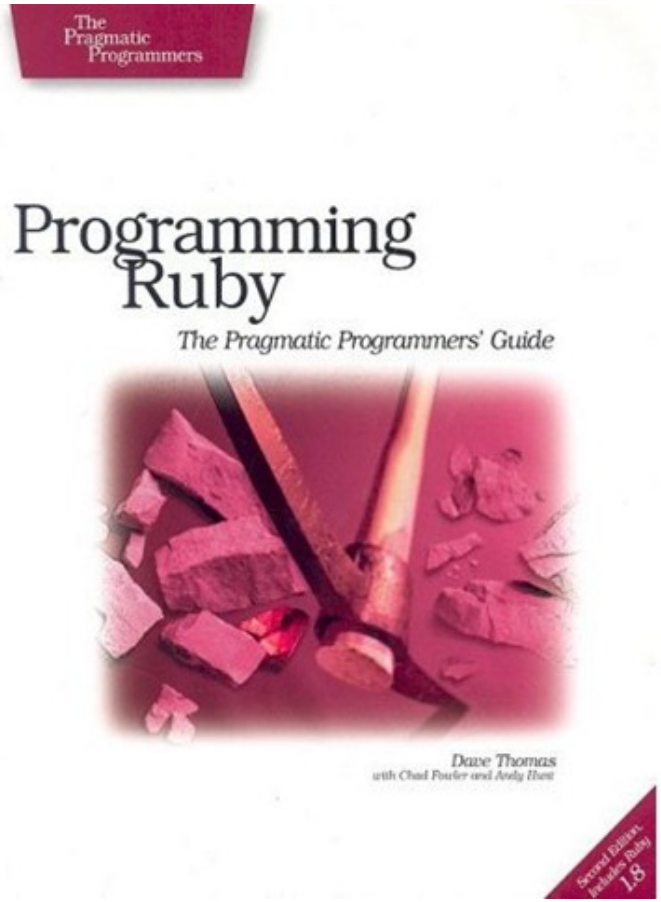
Forever Alone. . .

“Practical Metaprogramming: First Contact”

. . . and is it to be called an “eigenclass” or a “singleton class,” ma’am?

To Metaprogramming via Ruby

`attr_*` *Page 30*



Dynamic Getter / Setter Generation


```
sgharms@stharms-mac:~$ irb
ruby-1.9.2-p180 > class Demo
ruby-1.9.2-p180 ?>  attr_accessor :secret_word
ruby-1.9.2-p180 ?>  end
 => nil
ruby-1.9.2-p180 > Demo.instance_methods false
 => [:secret_word, :secret_word=]
ruby-1.9.2-p180 > d = Demo.new
 => #<Demo:0x000001010820d0>
ruby-1.9.2-p180 > d.secret_word="King Hobo!"
 => "King Hobo!"
ruby-1.9.2-p180 > d.secret_word
 => "King Hobo!"
ruby-1.9.2-p180 > d.respond_to? :secret_word
 => true
ruby-1.9.2-p180 >
```

To Metaprogramming via Rails

Rails (`order.discount=0.5`): *Page 28*

```
require 'active_record'

class Order < ActiveRecord::Base
end

order = Order.find(1)
order.discount = 0.5
order.save
```

Practical Metaprogramming

- `attr_*`: *Page 30*
- Rails (ORM): *Page 28*

Practical Metaprogramming

- `attr_*`: *Page 30*
- Rails (ORM): *Page 28*

But then what?

Momentary Aside: Terminology

"Spells" and their names derive from Metaprogramming Ruby by Paolo "Nusco" Perrotta:

http://ducktypo.blogspot.com/2010/08/metaprogramming-spell-book.html

"Slightly Impractical Metaprogramming:" Open Classes

- Open Classes

```
sgharms@stharms-mac:~$ irb
ruby-1.9.2-p180 > class String
ruby-1.9.2-p180 ?>  def hop_like_a_frog
ruby-1.9.2-p180 ?>    '*sproing*'
ruby-1.9.2-p180 ?>    end; end
 => nil
ruby-1.9.2-p180 > "Iggy Pop Ref.".hop_like_a_frog
 => "*sproing*"
ruby-1.9.2-p180 >
```

"Slightly Impractical Metaprogramming:" Kernel Method

- Open Classes
- Kernel Method

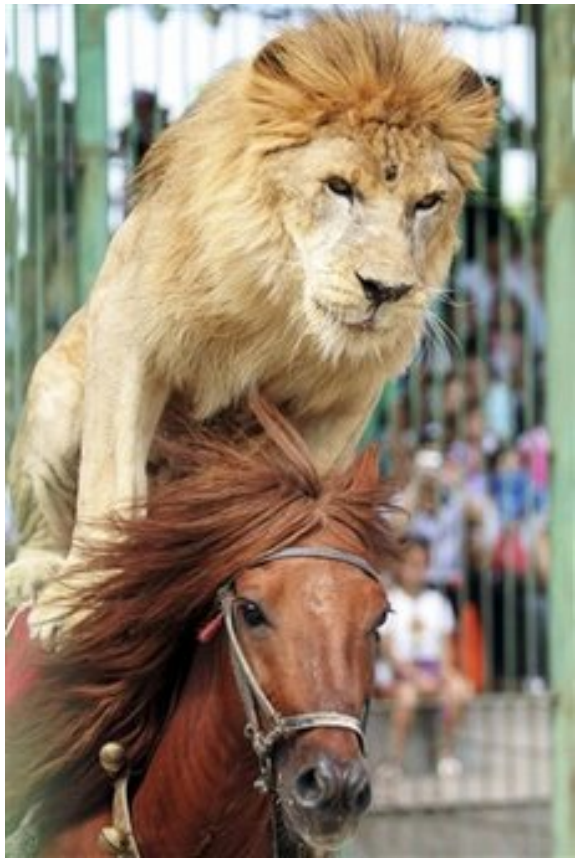
```
sgharms@stharms-mac:~$ irb
ruby-1.9.2-p180 > module Kernel
ruby-1.9.2-p180 ?>   def mc_hobo
ruby-1.9.2-p180 ?>     'yeehaw!'
ruby-1.9.2-p180 ?>     end
ruby-1.9.2-p180 ?>   end
 => nil
ruby-1.9.2-p180 > mc_hobo
 => "yeehaw!"
ruby-1.9.2-p180 >
```

"Slightly Impractical Metaprogramming:" Singleton Method

- Open Classes
- Kernel Method
- Singleton Method

```
ruby-1.9.2-p180 > class Cake; end
 => nil
ruby-1.9.2-p180 > birfday=Cake.new
 => #<Cake:0x000001009da840>
ruby-1.9.2-p180 > class << birfday
ruby-1.9.2-p180 ?>  def decorate
ruby-1.9.2-p180 ?>    "decorated a #{self}"
ruby-1.9.2-p180 ?>    end
ruby-1.9.2-p180 ?>  end
 => nil
ruby-1.9.2-p180 > puts birfday.decorate
decorated a #<Cake:0x000001009da840>
 => nil
ruby-1.9.2-p180 > zabu=Cake.new
 => #<Cake:0x000001009ad7f0>
ruby-1.9.2-p180 > zabu.decorate
NoMethodError: undefined method `decorate'
```

AWESOMENESS



– *Credit Unknown*

. . . Or Madness?

Incautiously used, these lead to the dangers of MP:

- Opaqueness
- Unpredictability
- Unsupportability

Thesis: F.U.D.

The first rule of ruby metaprograming is you don't talk about ruby metaprogramming (because you'll lead people new to it to over-use it). ;)

The second rule is "don't do it", and that might be part of why all the materials on it don't give the OP a good next step, because the next step should be "don't". If you can possible get away with it, then use something simpler and clearer. Unfortunately, you cannot simple be told not to MP, you must see it yourself. ;)

–Tim Connor: SF Ruby Mailing List

Antithesis: anti-F.U.D.

> Does metaprogramming make your code easier to manage if you have (for example) incomplete specifications? Maybe, as long as you know the techniques and know when to apply them. Again, knowledge is key, and I want to learn these techniques even if I end up choosing not to use them. That self-conscious choice can only come with knowledge, so I disagree with people warning Ruby programmers to steer clear of metaprogramming entirely.

–Paolo Perrotta, author of Metaprogramming Ruby, in e-mail to Steven Harms

Synthesis: You Need To Learn This: Precedent

1. Virtually all core libraries make use of MP

Synthesis: You Need To Learn This: Precedent

1. Virtually all core libraries make use of MP

```
50 module Exception2MessageMapper
51   @RCS_ID='-$Id: e2mmap.rb,v 1.10 1999/02/17 12:33:17 keiju Exp keiju $-'
52
53   E2MM = Exception2MessageMapper
54
55   def E2MM.extend_object(cl)
56     super
57     cl.bind(self) unless cl < E2MM
58   end
59
```

Synthesis: You Need To Learn This: Precedent

1. Virtually all core libraries make use of MP

```
50 module Exception2MessageMapper
51   @RCS_ID='-$Id: e2mmap.rb,v 1.10 1999/02/17 12:33:17 keiju Exp keiju $-'
52
53   E2MM = Exception2MessageMapper
54
55   def E2MM.extend_object(cl)
56     super
57     cl.bind(self) unless cl < E2MM
58   end
59
```

2. Rails uses MP all over the place

Synthesis: You Need To Learn This: Your Future

1. Save yourself a lot of typing

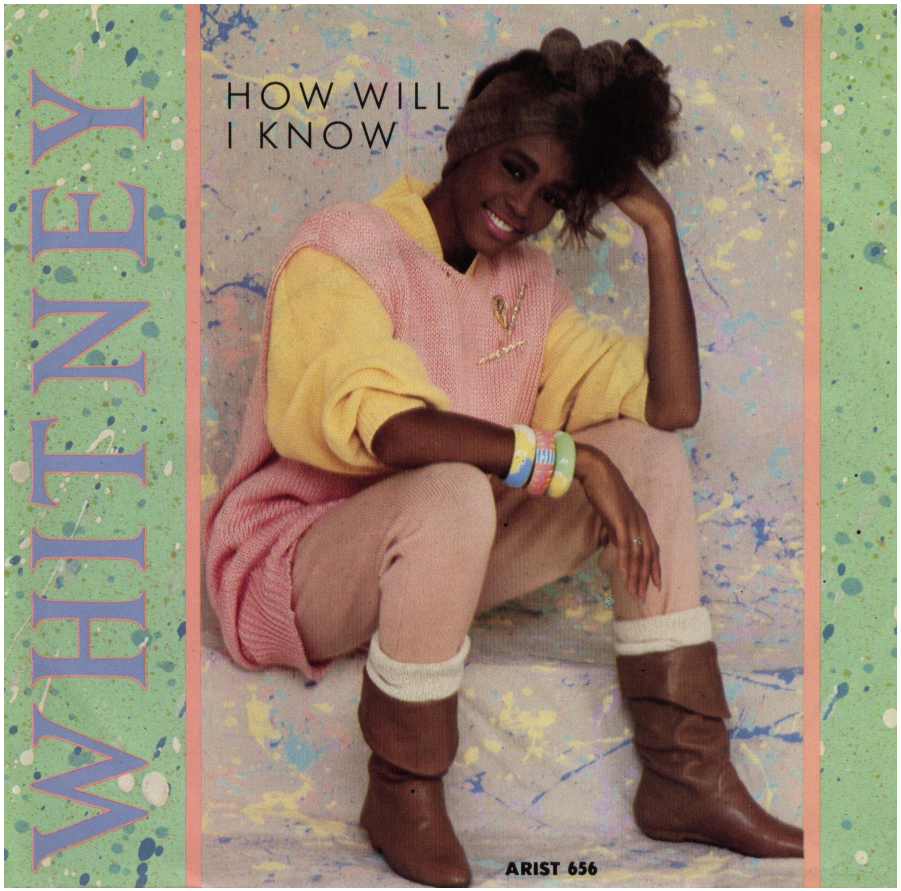
Synthesis: You Need To Learn This: Your Future

1. Save yourself a lot of typing
2. Reflect the interior world of your problem domain in your application code

Synthesis: You Need To Learn This: Your Future

1. Save yourself a lot of typing
2. Reflect the interior world of your problem domain in your application code
3. Pleasant surprises

How Will I Know?



Let This Be Your Guide

"Modeling Thought"

"Modeling Thought"

First Law of MP

$+$

Second Law of MP

$=$

"Modeling Thought"

First Law of Metaprogramming

A metaprogrammatic solution is suitable when you need to provide unambiguous answers (return values) to ambiguous questions (flexible / incomplete method calls)

First Law of Metaprogramming

A metaprogrammatic solution is suitable when you need to provide unambiguous answers (return values) to ambiguous questions (flexible / incomplete method calls)

e.g. Rails' ORM Calculation

First Law of Metaprogramming

A metaprogrammatic solution is suitable when you need to provide unambiguous answers (return values) to ambiguous questions (flexible / incomplete method calls)

e.g. Rails' ORM Calculation

AKA: "Pursuit of Insight"

Second Law of Metaprogramming

A metaprogrammatic solution is suitable when the mechanical recording of the the values is time-inefficient when compared to learning the generating heuristic.

Second Law of Metaprogramming

A metaprogrammatic solution is suitable when the mechanical recording of the the values is time-inefficient when compared to learning the generating heuristic.

e.g. attr_* methods

Second Law of Metaprogramming

A metaprogrammatic solution is suitable when the mechanical recording of the the values is time-inefficient when compared to learning the generating heuristic.

e.g. attr_* methods

AKA: "Avoidance of Typing"

First Corollary:

Any significant metaprogramming work undertaken to meet either of the laws will eventually look like it was undertaken for a reason in service to the opposite law

First Corollary:

Any significant metaprogramming work undertaken to meet either of the laws will eventually look like it was undertaken for a reason in service to the opposite law

Avoidance of typing reveals insight; pursuit of insight reduces typing

Fascinating Symmetries: Metaprogramming and Thinking

Law I: **Thinking** provides unambiguous answers to ambiguously asked things
Law II: **Thinking** is the more efficient learning of generating heuristics opposed to rote recording of data

This symmetry is the basis of "Modeling Thought."

“Modeling Thought:” Born That Way

Stepwise Support in Ruby for "Modeling Thought"

- Matz' Design

Stepwise Support in Ruby for "Modeling Thought"

- Matz' Design
- Perrotta's "Spells"

Stepwise Support in Ruby for "Modeling Thought"

- Matz' Design
- Perrotta's "Spells"
- A rule of thumb: "Modeling Thought"

Stepwise Support in Ruby for "Modeling Thought"

- Matz' Design
- Perrotta's "Spells"
- A rule of thumb: "Modeling Thought"
- Your **bold** usage

Demo

`LatinVerb`

Just Enough Latin

"Captain! My Captain!"

This Shouldn't Hurt. . . Much

Martin, the Wizard of Latin

Conjugation...

Given the four principle parts: "amō, amāre, amāvī, amatum"

The Specific Vector: "Active Voice / indicative mood / present tense/ first person / singular number" uniquely identifies:

amō

This process is called conjugation.

A Conjugation is a Unique Specification: Stargate

6 Points in Space

Unique Specification: Latin

5 Aspects

1. voice

Unique Specification: Latin

5 Aspects

1. voice
2. mood

Unique Specification: Latin

5 Aspects

1. voice
2. mood
3. tense

Unique Specification: Latin

5 Aspects

1. voice
2. mood
3. tense
4. person

Unique Specification: Latin

5 Aspects

1. voice
2. mood
3. tense
4. person
5. number

Unique Specification: Latin

5 Aspects

1. voice
2. mood
3. tense
4. person
5. number

A unique coordinate is a *vector* or a *conjugation*

Aggregation

Occasionally we want to cluster unique vectors *or* leave out an aspect to get a less-granular result:

Aggregation

Occasionally we want to cluster unique vectors *or* leave out an aspect to get a less-granular result:

- "active voice indicative mood present tense" (3 aspects, 6 results)

Aggregation

Occasionally we want to cluster unique vectors *or* leave out an aspect to get a less-granular result:

- "active voice indicative mood present tense" (3 aspects, 6 results)
- "active voice indicative mood present tense first person" (4 apsects, 3 results)

Aggregation

Occasionally we want to cluster unique vectors *or* leave out an aspect to get a less-granular result:

- "active voice indicative mood present tense" (3 aspects, 6 results)
- "active voice indicative mood present tense first person" (4 apsects, 3 results)
- "active voice indicative mood present tense first person singular number (Fully-Qualified)" (5 aspects, 1 result)

	Singular Number	Plural Number
First Person	laudō	laudāmus
Second Person	laudās	laudatis
Thrid Person	laudat	laudant

Vector Generation

- Vector generation is a well-known, well-established, heuristically-modeled domain.

Vector Generation

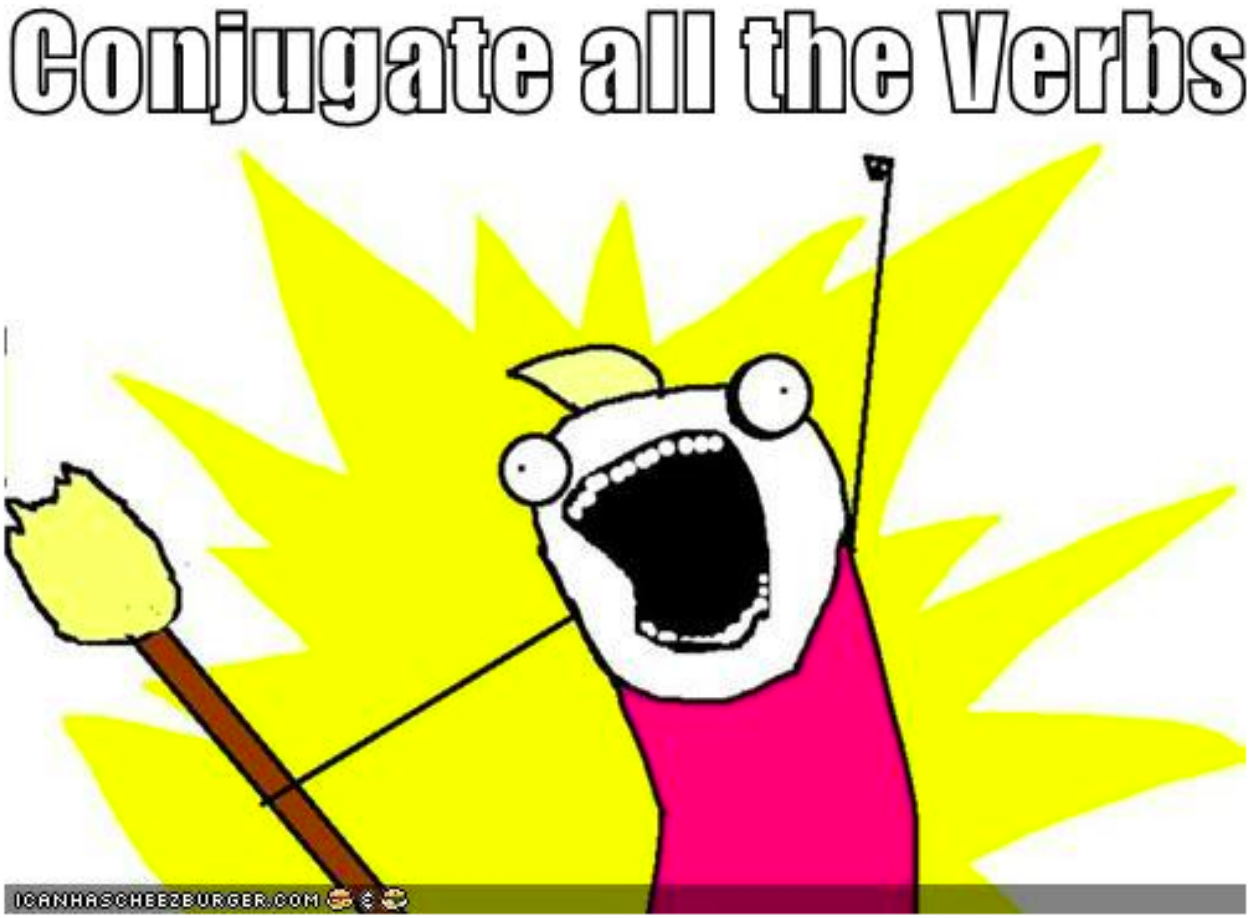
- Vector generation is a well-known, well-established, heuristically-modeled domain.
- $\approx$ 2,500 years of documentation

Vector Generation

- Vector generation is a well-known, well-established, heuristically-modeled domain.
- $\approx$ 2,500 years of documentation

"Erasmus" by Dürer

Model it in Ruby!



Credit: Allie Brosh

Or as Aaron Patterson once said:

"Do something worthless of questionable value."
– *Aaron Patterson*

LatinVerb's Purpose

LatinVerb should be a library for conjugating Latin verbs provided their "four principle parts (amō, amāre, amāvī, amatum)."

LatinVerb's Purpose

LatinVerb should be a library for conjugating Latin verbs provided their "four principle parts (amō, amāre, amāvī, amatum)."

Vectors should be accessed by pretty method calls like:

```
active_voice_indicative_mood_present_tense...
_first_person_singular_number
```

Painful Combinations

- 6 results means 6 methods to be defined *per tense*

Painful Combinations

- 6 results means 6 methods to be defined *per tense*
- . . . $\times$ 6 tenses (present/imperfect/future/perfect/past-perfect/future-perfect)

Painful Combinations

- 6 results means 6 methods to be defined *per tense*
- . . . $\times$ 6 tenses (present/imperfect/future/perfect/past-perfect/future-perfect)
- . . . $\times$ 2 voices (active/present)

Painful Combinations

- 6 results means 6 methods to be defined *per tense*
- . . . $\times$ 6 tenses (present/imperfect/future/perfect/past-perfect/future-perfect)
- . . . $\times$ 2 voices (active/present)
- . . . *and then there's another mood with 4 tenses of its own!*

Painful Combinations

- 6 results means 6 methods to be defined *per tense*
- . . . $\times$ 6 tenses (present/imperfect/future/perfect/past-perfect/future-perfect)
- . . . $\times$ 2 voices (active/present)
- . . . *and then there's another mood with 4 tenses of its own!*
- Each regular Latin verb has $\approx$ 160 unique vectors

Painful Combinations

- 6 results means 6 methods to be defined *per tense*
- . . . $\times$ 6 tenses (present/imperfect/future/perfect/past-perfect/future-perfect)
- . . . $\times$ 2 voices (active/present)
- . . . *and then there's another mood with 4 tenses of its own!*
- Each regular Latin verb has $\approx$ 160 unique vectors
- There are 5 standard paradigms

Painful Combinations

- 6 results means 6 methods to be defined *per tense*
- . . . $\times$ 6 tenses (present/imperfect/future/perfect/past-perfect/future-perfect)
- . . . $\times$ 2 voices (active/present)
- . . . *and then there's another mood with 4 tenses of its own!*
- Each regular Latin verb has $\approx$ 160 unique vectors
- There are 5 standard paradigms
- . . . and at least $1,000$ verbs

Pain

- Typing all of these several hundred(thousand?)-odd values for *fully-specified* vectors would definitely hurt

Pain

- Typing all of these several hundred(thousand?)-odd values for *fully-specified* vectors would definitely hurt
- What if you had *incomplete* data per our Aggregation section? 4/5 of the vector specification? 3/5? Write *yet* even more methods

	Singular Number	Plural Number
First Person	laudō	laudāmus
Second Person	laudās	laudatis
Thrid Person	laudat	laudant

Metaprogramming Makes a Saving Throw!

Law II says to MP our way out of the pain

Credit: Marco26 on DeviantArt

Specification

Take an array of elements and tack them onto a base, derived from the given, second part, with the results returned as an array. Sub-specify by person (1, 2, 3) and/or number or cluster by either.

Specification

Take an array of elements and tack them onto a base, derived from the given, second part, with the results returned as an array. Sub-specify by person (1, 2, 3) and/or number or cluster by either.

Why does that sound familiar?

Translating a Domain Problem to Ruby (1/3)

Take an array of elements and tack them onto a base, derived from the given, with the results returned as an array.

```
LatinIRB > p PERSONAL_ENDINGS
["s", "t", "mus", "tis", "nt"]
LatinIRB > p infinitive
"laudāre"
LatinIRB > stem=infinitive.sub /(.*)re$/, "\\1"
LatinIRB > p stem
"laudā"
LatinIRB > p PERSONAL_ENDINGS.map{|e| stem + e}
["laudās", "laudāt", "laudāmus", "laudātis", "laudānt"]
```

Translating a Domain Problem to Ruby (2/3)


```
LatinIRB > p aLatinVerb.original_string
"laudō   laudāre  laudāvī  laudatum"
LatinIRB > k=long_call.join(%q/_/).to_sym
LatinIRB > p k
:active_voice_indicative_mood_present_tense
LatinIRB > aTB=aLatinVerb.send(k.to_sym)
LatinIRB > aTB.to_a.each do |x|
LatinIRB >     puts x
LatinIRB ?>  end
laudō
laudās
laudat
laudāmus
laudātis
laudant
```

Translating a Domain Problem to Ruby (3/3)

Sub-specify by person (1, 2, 3) or number or cluster by either.

Translating a Domain Problem to Ruby (3/3)

Sub-specify by person (1, 2, 3) or number or cluster by either. . . . and allow terms in method call to be reordered

```
LatinIRB > p aTB.second_person
["laudātis", "laudās", "laudātis"]
LatinIRB > p aTB.plural_number
["laudātis", "laudāmus", "laudātis", "laudant"]
LatinIRB > p aTB.second_person_plural_number
"laudātis"
LatinIRB > p aTB.plural_number_second_person
"laudātis"
LatinIRB >
```

MP Provides: Massive Laziness Win

- $\approx$ 48 methods covered; 6 written

MP Provides: Massive Laziness Win

- $\approx$ 48 methods covered; 6 written
- 2 aspects in play

MP Provides: Massive Laziness Win

- $\approx$ 48 methods covered; 6 written
- 2 aspects in play
- One reponse class (`TenseBlock`)

Scale it Up!: Dynamic Dispatch in `method_missing`

Be flexible on all 5 aspects

```
41        def method_missing(symbol, *args )  # :nodoc:
42          super if @tense_list.nil?
43          @tense_list.find do |e|
44            if symbol.to_s.match /^(#{e})_(.*)/
45              tense_method, vector_specifier = $1, $2
46              # This is added to prevent stack-level too deep errors
47              begin
48                # Handle the base case
49                if self.respond_to?(tense_method.to_sym)
50                  return send(tense_method.to_sym).send(vector_specifier.to_sym)
51                end
52              rescue SystemStackError => e
53                STDERR.puts "We encountered a SystemStackError when calling #{tens
54                STDERR.puts "WARNING:  Failed to resolve [#{tense_method}] with [#
55                super
56              end
57            end
58          end
59          super
60        end
```

Result: Super-Massive Laziness Win

- Covered the thousands of methods predicted

Result: Super-Massive Laziness Win

- Covered the thousands of methods predicted
- . . . and provided the clustering methods as well as a surprising bonus

Result: Super-Massive Laziness Win

- Covered the thousands of methods predicted
- . . . and provided the clustering methods as well as a surprising bonus

I only wrote 24 methods

Benefits via Law II

- Saved many keystrokes by writing using a Ghost Method

Benefits via Law II

- Saved many keystrokes by writing using a Ghost Method
- Saved creating **$A_LOT_OF** additional methods – *many of which I didn't even think of!*

The Corollary Emerges

- Pursued "less typing" but wound up with "respond to ambiguous calls with unambiguous data"

The Corollary Emerges

- Pursued "less typing" but wound up with "respond to ambiguous calls with unambiguous data"
- *Insights Emerged!*: 'Modeling Thought" yields many of these surprises

Law I Emerges. . . With Surprises

- 5 additional "aggregate methods" per tense emerged
  - aTenseBlock.singular_number (`3 results`)
  - aTenseBlock.first_person (`2 results`)

Law I Emerges. . . With Surprises

- 5 additional "aggregate methods" per tense emerged
  - aTenseBlock.singular_number (`3 results`)
  - aTenseBlock.first_person (`2 results`)
- Flexible word order *emerged* that did the right thing
  - first_person_singular_number
  - singular_number_first_person

Law I Emerges. . . With Surprises

- 5 additional "aggregate methods" per tense emerged
  - aTenseBlock.singular_number (`3 results`)
  - aTenseBlock.first_person (`2 results`)
- Flexible word order *emerged* that did the right thing
  - first_person_singular_number
  - singular_number_first_person
- Avoided Java-ish paramteterized brain damage

Java-ish Brain Damage: Parameterization


```
String calculate_vector(VerbyType aV, String v,
String m, String t, String p, String n)
```

OR

```
Object[] calculated_values = {aV, v, m, t, p, n};
String calculate_vector(calculated_values);
```

anti-Parameterization: Not How We Think, Not Modeled Thought

- We *do not* think like we are ordering from Subway.

anti-Parameterization: Not How We Think, Not Modeled Thought

- We *do not* think like we are ordering from Subway.
- NO: *"I'll have a sandwich: bread is rustic Italian, meat is salami, cheese is provelone, veggies are an array of lettuce, tomato . . ."*

anti-Parameterization: Not How We Think, Not Modeled Thought

- We *do not* think like we are ordering from Subway.
- NO: *"I'll have a sandwich: bread is rustic Italian, meat is salami, cheese is provelone, veggies are an array of lettuce, tomato . . ."*
- YES: *"I'll have an Italian on rustic Italian with salami and provelone and from the veggie bin, lessee, lettuce and tomato. . ."*

Pretty Parameterization

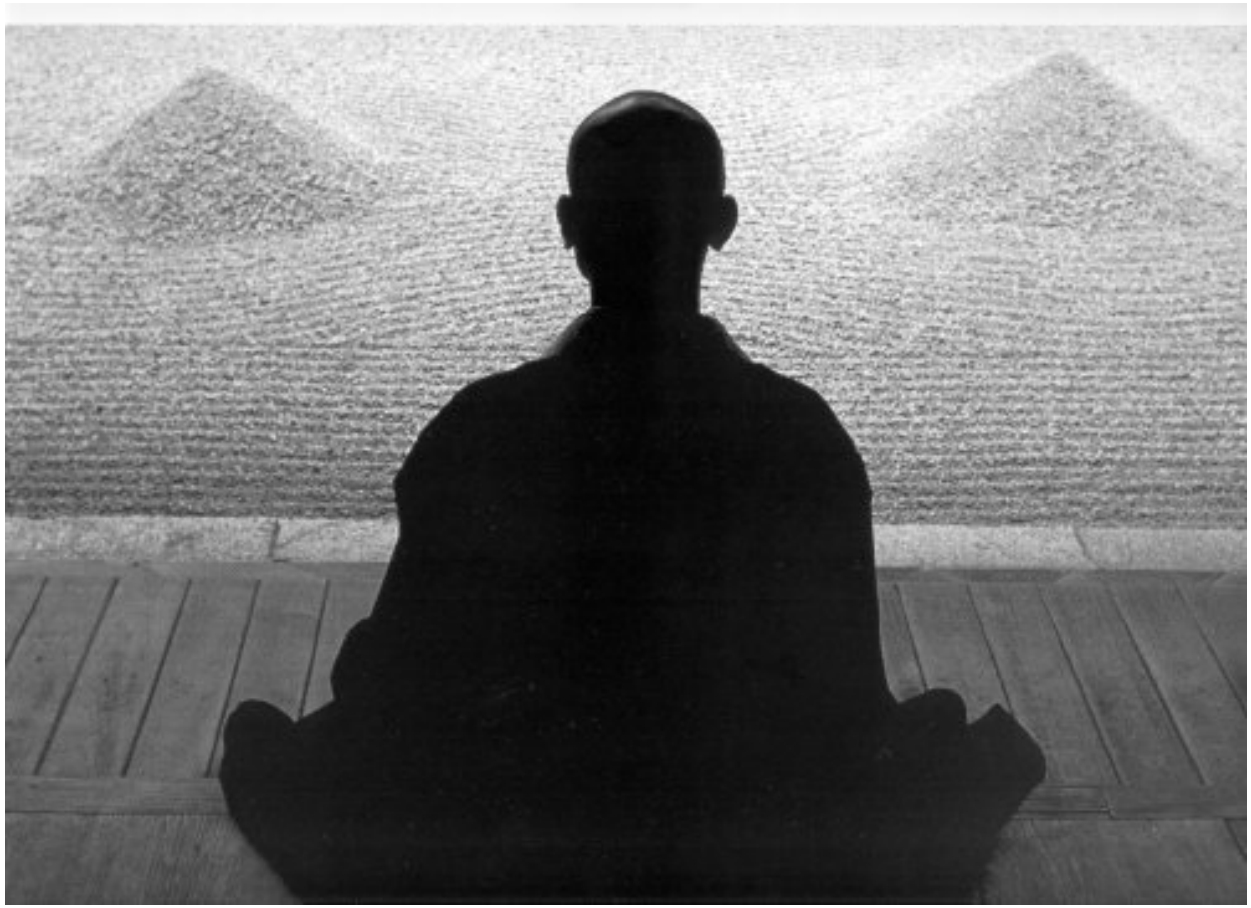

```
instance.active_voice_indicative_mood_present_tense
```

NOT

```
instance.calculate_vector('active','indicative')
...
```

Greatest Benefit: Clarity & Communication

"Code that Communicates"



Pause for “applause”

`LatinVerb: Demo End`

Pause for “applause”

`LatinVerb: Demo End`

Applause is derived from our example word “laudāre” meaning “to praise” literally meaning “to praise toward”

I'm not that desperate for approval. . .

I'm not that desperate for approval. . .

I lied. . .

What MP Techniques Make This Possible?

HOW?

Ghost Method in TenseBlock


```
##
#
# Provide a method_missing so that ambiguous cases can be resolved
#
##
def method_missing(symbol, *args)
  begin
    returnArray = []
    methods.grep(/#{symbol.to_s}/) do |s|
     returnArray.push(send s)
    end
    return returnArray unless returnArray.empty?
  rescue Exception
  end
  super
end
```

MP Techniques Used: Second-Degree Spells

1. Blank Slate: A TenseBlock is a Blank Slate, effectively
2. Ghost Method: Dont define a method (in `TenseBlock`) so that its `method_missing` acts as `method_called` for dispatching
3. Dynamic Dispatch: `self.send` in `TenseBlock`

Third-Degree Spells

- Dynamic Method
- Around Alias
- DSL: *See Evan's talk!*
- Class Extension a.k.a. Mixin

Conclusion: Goals Into Action!

1. **Let's provide a clearer path to learning** MP in the Ruby community . . .

Conclusion: Goals Into Action!

1. **Let's provide a clearer path to learning** MP in the Ruby community . . .
2. **since we know we** can't *not* learn this.

Conclusion: Goals Into Action!

1. **Let's provide a clearer path to learning** MP in the Ruby community . . .
2. **since we know we** can't *not* learn this.
3. **Since we know when we** should reach for the MP "hammer" **thanks to the "modeling thought" guideline**

Conclusion: Goals Into Action!

1. **Let's provide a clearer path to learning** MP in the Ruby community . . .
2. **since we know we** can't *not* learn this.
3. **Since we know when we** should reach for the MP "hammer" **thanks to the "modeling thought" guideline**
4. **and have a demonstrated** example of thinking in terms of MP

Conclusion: Goals Into Action!

1. **Let's provide a clearer path to learning** MP in the Ruby community . . .
2. **since we know we** can't *not* learn this.
3. **Since we know when we** should reach for the MP "hammer" **thanks to the "modeling thought" guideline**
4. **and have a demonstrated** example of thinking in terms of MP
5. **we will** use MP **boldly**

Supplementary Information

Book

Metaprogramming Ruby by Paolo Perrotta

List of Spells

http://ducktypo.blogspot.com/2010/08/metaprogramming-spell-book.html

(Meta)programming Politely

http://confreaks.net/videos/374-rubyconf2010-the-polite-programmer-s-guide-to-ruby-etiquette

Photo Credits

1. "Zen" pic: `http://www.insidesocal.com/tomhoffarth/archives/2011/06/shawn-greens-ze.html`

Colophon

LaTeXand the Beamer Slide Toolkit

SpeakerRate

Help Me Get Better!

http://speakerrate.com/talks/7831-practical-metaprogramming-modeling-thought

Contact Me! (Again)

Steven G. Harms

Physically: San Francisco, CA
Email: `lsrcv@sgharms.oib.com`
Twitter / GitHub: `sgharms`
G+