Design patterns in Ruby

Aleksander Dąbrowski

3 Mar 2009

www.wrug.eu

What are design patterns?

Why should I know them?

For Money ;)

They sound like something advance and professional

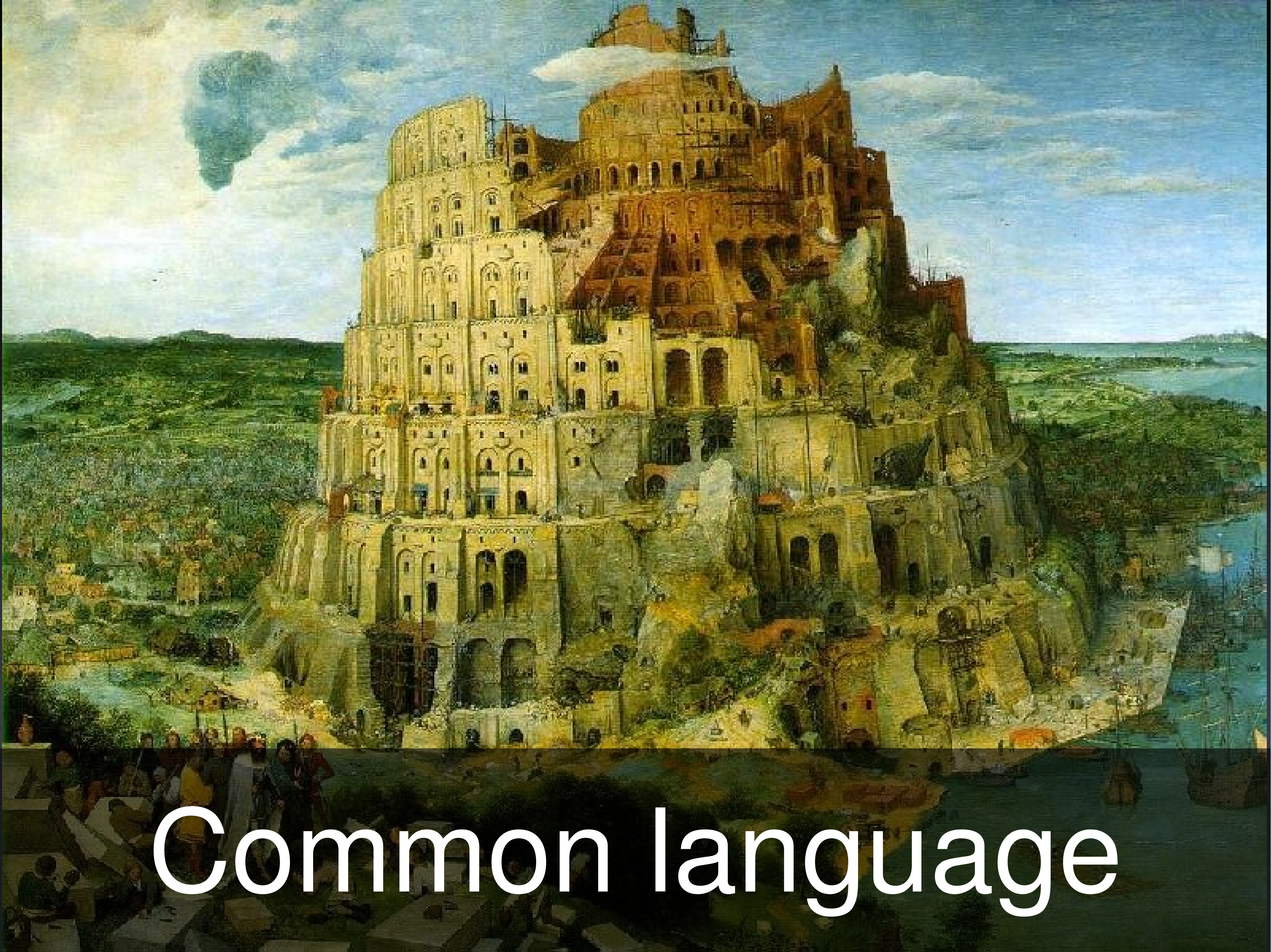
Popular problems are already solved

Don't invent the wheel

FAIL

GTFO n00b

Design pattern is description of popular solution



Common language

Plan:

1. Observer
2. Template Method
3. Strategy

Let's go to the point

1. Observer

We want to be notified when something change status

Example:

Cat is observing a mouse hole.

When mouse leaves the hole, cat starts to hunt.

```
class Hole

  def enter( mouse )
         puts '#{ mouse.name } is safe'
    end

    def exit( mouse )
         puts '#{ mouse.name } is not safe'
  end
end
```

How to make the cat observer?

```
module Observable
  def initialize
    @observers=[]
  end

  def add_observer(observer)
    @observers << observer
  end

  def delete_observer(observer)
    @observers.delete(observer)
  end

  def notify_observers
    @observers.each do |observer|
      observer.update(self)
    end
  end
end
```

```
class Cat

  def update
    hunt_the_mouse
  end

  private:

  def hunt_the_mouse
    jump
    kill
    ...
  end

end
```

```
class Hole
  include Observable

  def observe( cat )
     add_observer( cat )
  end

  def enter( mouse )
    puts '#{ mouse.name } is safe'
  end

  def exit( mouse )
    puts '#{ mouse.name } is not safe'
    notify_observers
  end
end
```

i can waits
ICANHASCHEEZBURGER.COM

How to make the cat observer?

In Ruby simply use Observable mixin

require 'observer'

```
require 'observer'

class Hole
  include Observable

  def observe( cat )
     add_observer( cat )
  end

  def enter( mouse )
    puts '#{ mouse.name } is safe'
  end

  def exit( mouse )
    puts '#{ mouse.name } is not safe'
    changed
    notify_observers( self )
  end
end
```

Observable:

```
add_observer( observer )
changed( state = true )
changed?
count_observers
delete_observer( observer )
delete_observers
notify_observers( *arg )
```

Are you already using observer?

```
class EmployeeObserver < ActiveRecord::Observer
  def after_create(employee)
    # New employee record created.
  end

  def after_update(employee)
    # Employee record updated
  end

  def after_destroy(employee)
    # Employee record deleted.
  end
end
```

2. Template Method

Use it when:

part of code has to cope with different tasks

probably more changes will be made

Generating HTML report

```
class Report
  def initialize
    @title = 'Monthly Report'
    @text =  ['Things are going', 'really, really well.']
  end

  def output_report
    puts('<html>')
    puts('  <head>')
    puts("    <title>#{@title}</title>")
    puts('  </head>')
    puts('  <body>')
    @text.each do |line|
      puts("    <p>#{line}</p>" )
    end
    puts('  </body>')
    puts('</html>')
  end
end
```

```
report = Report.new
report.output_report
```

Output

```
<html>
  <head>
    <title>Monthly Report</title>
  </head>
  <body>
    <p>Things are going</p>
    <p>really, really well.</p>
  </body>
</html>
```

How to add generating plain text report?

```
def output_report(format)
  if format == :plain
    puts("*** #{@title} ***")
  elsif format == :html
    puts('<html>')
    puts('  <head>')
    puts("    <title>#{@title}</title>")
    puts('  </head>')
    puts('  <body>')
  else
    raise "Unknown format: #{format}"
  end

  @text.each do |line|
    if format == :plain
      puts(line)
    else
      puts("    <p>#{line}</p>" )
    end
  end
```

It's little complicated. What will happen when we add PDF?

Isolation of elements

```
class Report
  def initialize
    @title = 'Monthly Report'
    @text =  ['Things are going', 'really, really well.']
  end

  def output_report
    output_start
    output_head
    output_body_start
    output_body
    output_body_end
    output_end
  end

  def output_body
    @text.each do |line|
      output_line(line)
    end
  end
```

Use abstract classes

Ruby doesn't has abstract classes

Solution:

Use 'raise'

```
def output_body
    @text.each do |line|
      output_line(line)
    end
  end

  def output_start
    raise 'Called abstract method: output_start'
  end

  def output_head
    raise 'Called abstract method: output_head'
  end

  def output_body_start
    raise 'Called abstract method: output_body_start'
  end
```

Two subclasses:
html & plain
(pdf, rdf, doc... in
future)

```
class HTMLReport < Report
  def output_start
    puts('<html>')
  end

  def output_head
    puts('  <head>')
    puts("    <title>#{@title}</title>")
    puts('  </head>')
  end

  def output_body_start
    puts('<body>')
  end

  def output_line(line)
    puts("  <p>#{line}</p>")
  end
```

```
class PlainTextReport < Report
  def output_start
  end

  def output_head
    puts("**** #{@title} ****")
    puts
  end

  def output_body_start
  end

  def output_line(line)
    puts line
  end
```

How to use it?

```
report = HTMLReport.new
report.output_report
```

```
report = PlainTextReport.new
report.output_report
```

Subclasses covers abstract methods

They do not cover output report

Hook methods

Non abstract methods,
which can be covered.

```
class Report

(...)

  def output_start
  end

  def output_head
    output_line( @title )
  end

  def output_body_start
  end

  def output_line( line )
     raise 'Called abstract method: output_line'
  end
```

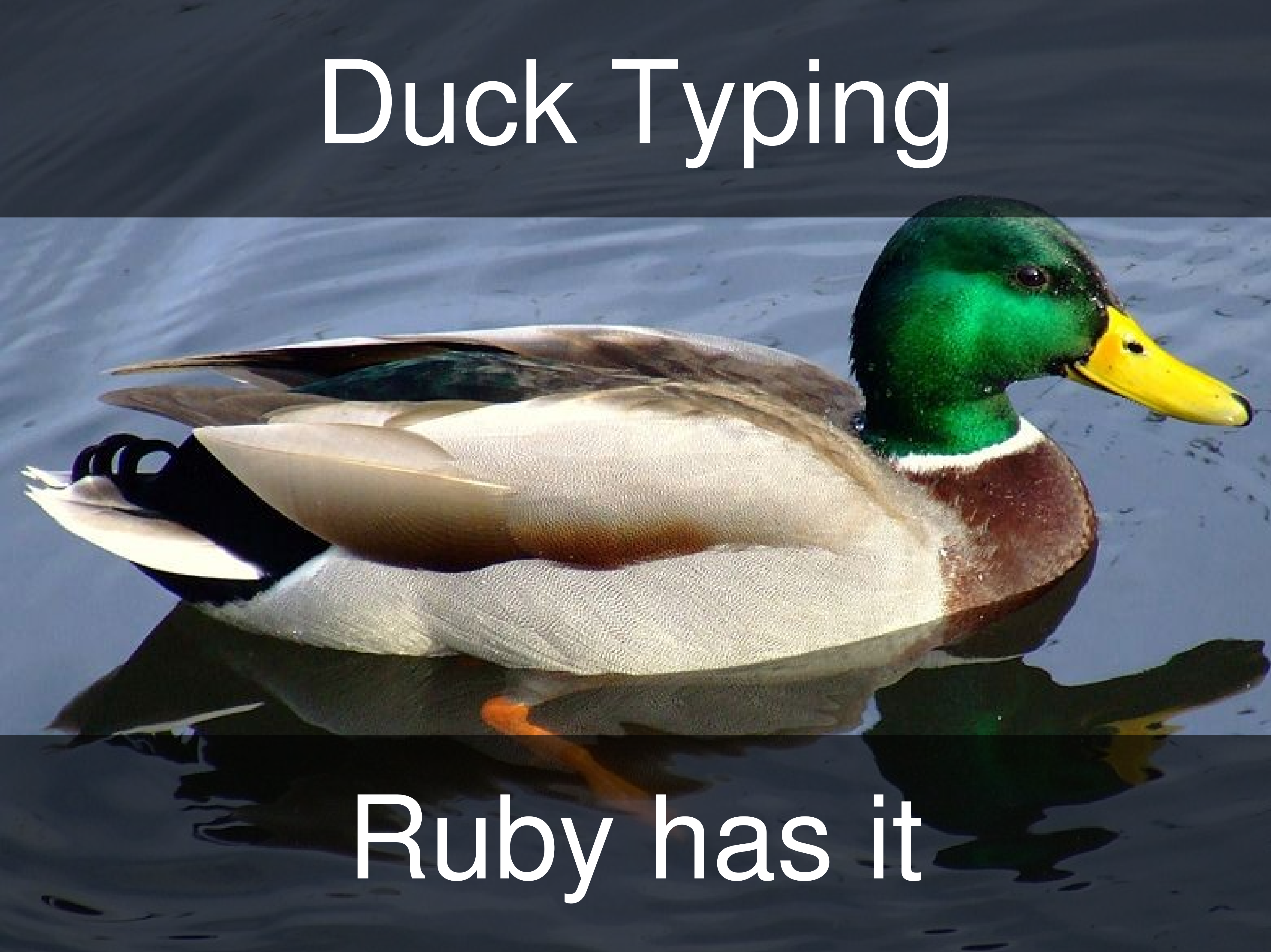
```
report = HTMLReport.new
```

Duck Typing

“If it looks like a duck, swims like a duck and quacks like a duck, then it probably is a duck.”

Ronald Reagen

Duck Typing



Ruby has it

3. Strategy

```
class Formatter
  def output_report(title, text)
    raise 'Abstract method called'
  end
end

class HTMLFormatter < Formatter
  def output_report( title, text )
    puts('<html>')
    puts('  <head>')
    puts("    <title>#{title}</title>")
    puts('  </head>')
    puts('  <body>')
    text.each do |line|
      puts("    <p>#{line}</p>" )
    end
    puts('  </body>')
    puts('</html>')
  end
end
```

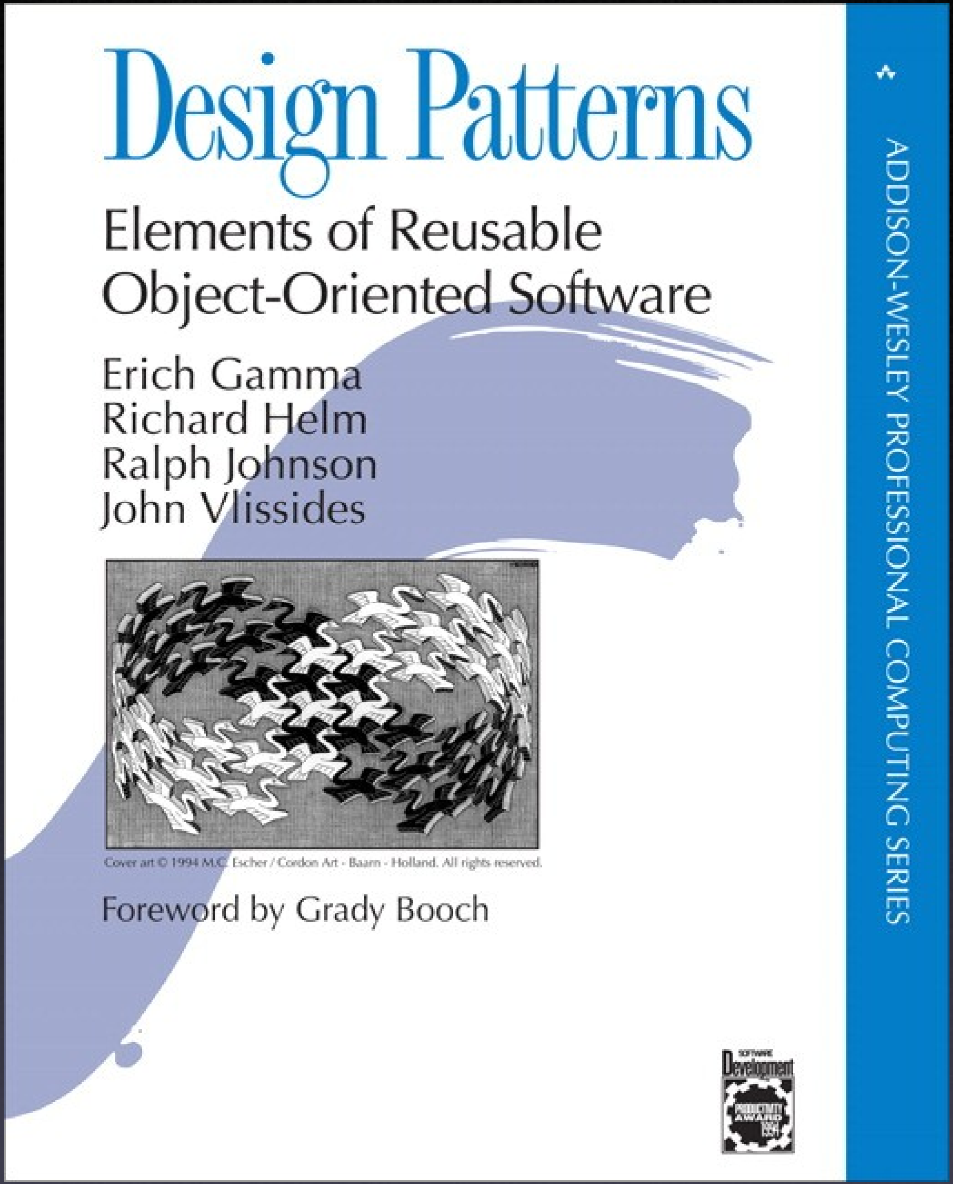
```
class PlainTextFormatter < Formatter
  def output_report(title, text)
    puts("***** #{title} *****")
    text.each do |line|
      puts(line)
    end
  end
end
```

```
class Report
  attr_reader :title, :text
  attr_accessor :formatter

  def initialize(formatter)
    @title = 'Monthly Report'
    @text =  ['Things are going', 'really, really well.']
    @formatter = formatter
  end

  def output_report
    @formatter.output_report( @title, @text)
  end
end
```

We can change strategy during program execution

```
report = Report.new(HTMLFormatter.new)
report.output_report

report.formatter = PlainTextFormatter.new
report.output_report
```

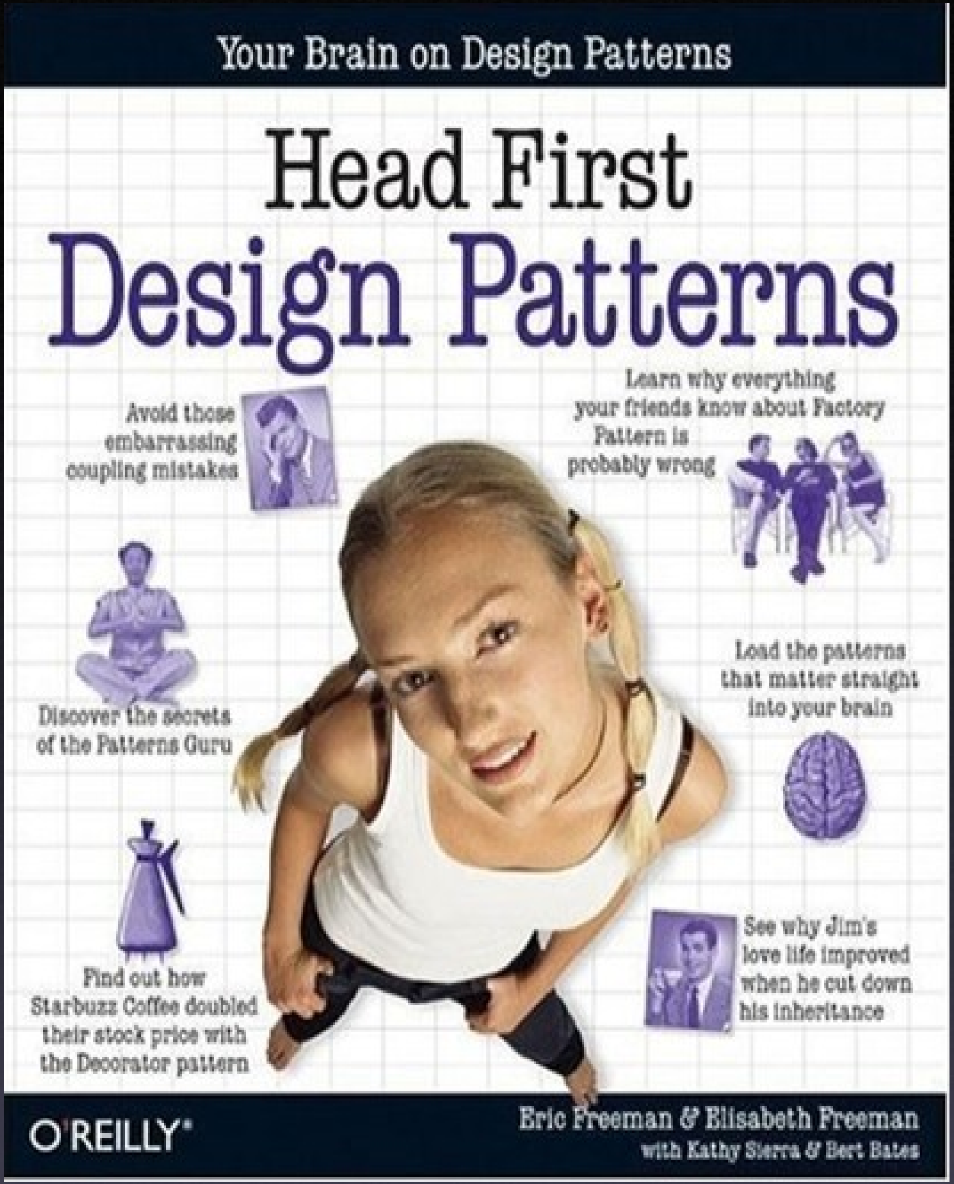
Sources


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Bible

Addison-Wesley Professional Ruby Series
DESIGN PATTERNS IN RUBY
Foreword by Obie Fernandez
Professional Ruby Series Editor
RUSS OLSEN

Ruby only


Your Brain on Design Patterns

Head First Design Patterns

Avoid those embarrassing coupling mistakes

Learn why everything your friends know about Factory Pattern is probably wrong

Discover the secrets of the Patterns Guru

Load the patterns that matter straight into your brain

Find out how Starbuzz Coffee doubled their stock price with the Decorator pattern

See why Jim's love life improved when he cut down his inheritance

O'REILLY

Eric Freeman & Elisabeth Freeman with Kathy Sierra & Bert Bates

Simple

in Java