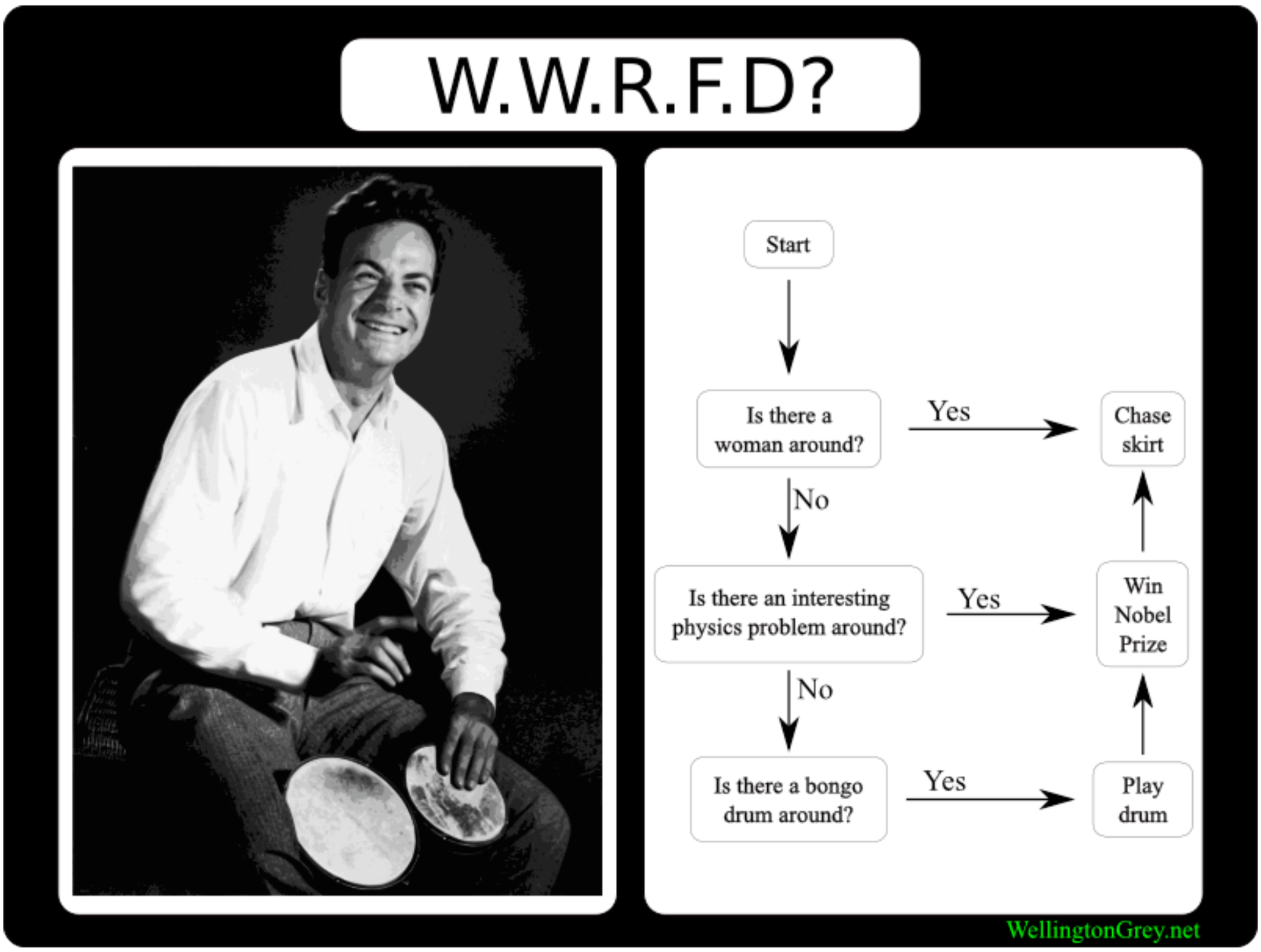
Six Easy Pieces (Twice Over)

Adam Keys
Dallas TechFest
http://therealadam.com

Six Easy Pieces



Richard Feynman. Awesome nerd, or awesomest nerd?

I'm going to try and walk in the footsteps of a giant here. I want to tell you why Ruby is special. Further, I want to convince you should use Ruby **even if you can't use Ruby.**

I know, its preposterous.

Luckily, I'm armed with two presentations, of six easy pieces each. Six Easy Pieces is The Feynman Lectures on Physics, distilled for the non-physicist. The presentations I have are for Rubyists, but I want to distill them down to something programmers of all sorts can appreciate.

The Culture And Aesthetic Of Ruby And Rails REVISITED

Act the first. I first gave this presentation in April of 2006. Scant months after I had started writing Ruby full time. It was sort of an intro to the non-syntactic aspects of Ruby and Rails.

I later realized that, hey, this presentation isn't really specific to Ruby. In many ways, it outlines what I think are important ideas for **all** software developers.

For today, I've extracted the most important bits.

Programming Shouldn't Suck

Very important! Dare I say, most important. We're stuck with these contraptions for many hours a day, so they might as well not suck.

Oddly enough, I encourage you to constantly seek ways to convince yourself that the grass is greener somewhere else. Learn other languages, try and use their idioms in your current language of choice. See where that works out and where its not quite satisfying.

This is how you **really** get to the point where programming doesn't suck. You've got a tool that supports lots of idioms, and most of them feel right.

Clever, Pretty And POWERFUL

This is how a language should make you feel. Some languages seem to taunt me. “You can do that, but then I couldn’t protect you.” I don’t want your protection! Other languages let me do clever things, but only grudgingly. “Fine, you can do that. But I’m not going to like it, and I’m sure not going to make it easy for you.”

Great languages flatter me. And you too. They make me feel smarter than I probably am. “Hey look what I just did! How cool is that?” They allow me to create something that seems relatively beautiful. “I like the way that code looks.”

Finally, they give me that “muahahahahaha” moment. Not the kind of “muahahaha” where you’re taunting the system; the kind where you worked with the system to do something you didn’t even think was possible when you started off.

That’s a good language to me. Right now Ruby is that language, but tomorrow it could be JavaScript, Erlang or something I’ve never even heard of before.

Build Up, Build Down

Allow me to weave a few ideas together.

I think the best software is built with equal parts building up and building down. You want to build the programming language **up** to meet a domain langauge that you build **down.** In Ruby-land, we tend to build both with Ruby. But in C# or Java, some XML or JSON might intervene.

While there's a lot to say about building everything in the same language, the really crucial part is the thought process. There's this whole Sapir-Whorf hypothesis that suggests that we can only think about that which we have language for. Thus, in learning the language our customers in, for example, international shipping use, we can "cluster" ideas and translate their jargon into classes, methods, components and processes in our software.

This inevitably leads to software with the "quality without a name" - the idea at the root of Pattern Languages. That's a term invented by Christopher Alexander, an architect who was trying to describe the things he saw in much of his work. That work in turn inspired the Gang of Four book, Design Patterns, which in turn gave us just "patterns."

Its unfortunate that we just call it "patterns" these days, because the really, really important part is that its a language. Which is what building your programming **langauge** up and your domain **language** down is all about.

More Is Less

Your brain. Its a fragile thing. Its ensconced in a hard shell, placed a top your body in a pretty easy to maneuver and defend spot. And yet, so many of us try to overburden it. Enterprise Beans this, Sharepoint that. Apache configurations over there, XML bindings making dust bunnies in the closet.

I’m picking on these things because they grew out of what Joel Spolsky would characterize as “architecture astronaut” takeovers. Software architects (the ivory tower sort) get paid for expanding the feature list at a minimum of bottom-line expense. Frequently they succeed at the former with a net failure at the latter.

I Do Solemnly Swear:

- To only write the code that is needed today and avoid building for hypothetical futures
- To delete code whenever it is possible and even when it is not entirely advisable
- To consider the next person to grok this application, since it will probably turn out that person is me

So, I'll stop picking on people if we will all agree to the following.

Further, I'd appreciate it if we could all keep our crazy experiments with new techniques and technologies isolated to side projects and blog posts. I think we can realize major gains by following these easy steps!

Don’t Repeat Yourself

I apologize if you’ve heard me say this a few times. It bares repeating.

Nothing’s worse in developing an application than the tedium that can take over when you changing one “thing” means several edits in several files spread across many folders. If you’re working on software like this, you’ve got a “smell” and its holding you back. That means you’re not having fun, and the whole point of writing software is having fun. In my book, at least.

OK, PSA over. Let us move on.

Testing As Feedback Loop

Test driven development is a skill unto itself. It seems it should prove as easy as another API. After all, its usually expressed as an API. To the contrary, getting **really** good at TDD is as hard as learning a language whose conventions and idioms are completely opposite of the one you use on a daily basis.

And its worth it. Because, if you can get good at it, **your code will talk to you.**

Not in the creepy, hearing voices in your head sort of way. Your code will tell you when you break it and when you fix it. It will tell you which parts are brittle and which parts are solid. It will tell you when code is well factored and when it could use some love.

TDD is the best feedback mechanism you could possibly imagine. If you're not using, find someone here who is and make a new best friend. You'll thank yourself later.

Borrowing From Ruby

Up to this point, I've given you some rousing ideas, but not much to execute on. Now I'm going to put some Ruby in front of you. But really, you could write most of this in any language you choose. I say most because as I get towards the end, your language choices are severely limited. But, there is a corresponding reward in what you can achieve.

Writing for humans

```
def sensor_for(sensor)
  71.5 if sensor == :heat
end

heat = sensor_for(:heat)

puts "Its #{heat} degrees outside. Beautiful!"
```


Step one. You can do this in any language know to man. Write your program so it reads for humans, not for machines. Don’t mention buffers and devices unless you’re a kernel programmer. Don’t mention Factories or Singletons unless you’re building a framework.

The power of the word ‘for’ is under-appreciated in programming, methinks. If we were to simply append that to some of our method names, things magically become more legible.

Closures

```
def sensor_for(sensor, &block)
  yield 71.5 if sensor == :heat
  yield 30.3 if sensor == :humidity
end

sensor_for(:heat) do |temp|
  puts "Its #{temp} degrees outside.
Beautiful!"
end
```

The Functional Style I

```
instruments = [:thermometer, :barometer]

instruments.map do |sensor|
  sensor_for(sensor) do |value|
    puts "The old #{sensor} reports #{value}!"
  end
end

pp instruments.map { |sensor|
sensor_for(sensor) * 2 }
```

The Functional Style II

```
def sensor_for(sensor)
  value = case sensor
  when :thermometer
    71.5
  when :barometer
    30.3
  else
    raise 'Invalid instrument'
  end

  if block_given?
    yield value
  else
    value
  end
end
```

Metaprogramming I

```
def sensor_for(name)
  # Sensors.send(name)
  Sensors.new.send(name)
end

Sensors.create_sensor(:thermometer, 71.5)
Sensors.create_sensor(:humidity, 31.4)

puts "Hey, its #{sensor_for(:thermometer)}
degrees. Go outside!"
puts "The humidity is #{sensor_for(:humidity)}
percent."
```

Metaprogramming II

```
class Sensors
  def self.create_sensor(name, value)
    # instance_eval <<-EOC
    #       def #{name}
    #         #{value}
    #       end
    #     EOC
    class_eval do
      define_method(name) do
        value
      end
    end
  end
end
```

Declarative Programming I

```
sensor do
  name :thermometer
  value 71.5
end

sensor do
  name :audience_enjoyment
  value 'unknown!'
end

puts "Its #{sensor_for(:thermometer).value} degrees outside"
puts "The audience's enjoyment of this talk is
#{sensor_for(:audience_enjoyment).value}"
```

Declarative Programming II

```
$sensors = []

def sensor(&block)
  $sensors << Sensor.create(&block)
end

def sensor_for(name)
  $sensors.each do |sensor|
    return sensor if sensor.name == name
  end
end
```

Declarative Programming III

```
class Sensor
  def self.create(&block)
    returning new do |s|
      s.instance_eval(&block)
    end
  end

  def name(name=nil)
    name.nil? ? @name : (@name = name)
  end

  def value(val=nil)
    val.nil? ? @value : (@value = val)
  end
end
```

Tasteful DSLs I

```
name 'operating system'
value {
  sh '/usr/bin/uname'
}
```

Tasteful DSLs II

```
s = SensorFu.new
s.parse(program)
s.run_sensors
```

Tasteful DSLs III

```
class SensorFu
  attr_reader :sensors

  def initialize
    @sensors = []
  end

  def parse(program)
    sensor = Sensor.new
    sensor.instance_eval(program)
    sensors << sensor
  end

  def run_sensors
    sensors.each do |s|
      "(#{s.display_name}) #{s.run}"
    end
  end

  class Sensor
  end
end
```

Tasteful DSLs IV

```
class Sensor
  attr_reader :display_name

  def name(name)
    @display_name = name
  end

  def value(&block)
    @value = block
  end

  def run
    @value.call
  end
end
```

EOF