Design Patterns in Ruby

Pisano

What's a design pattern ?

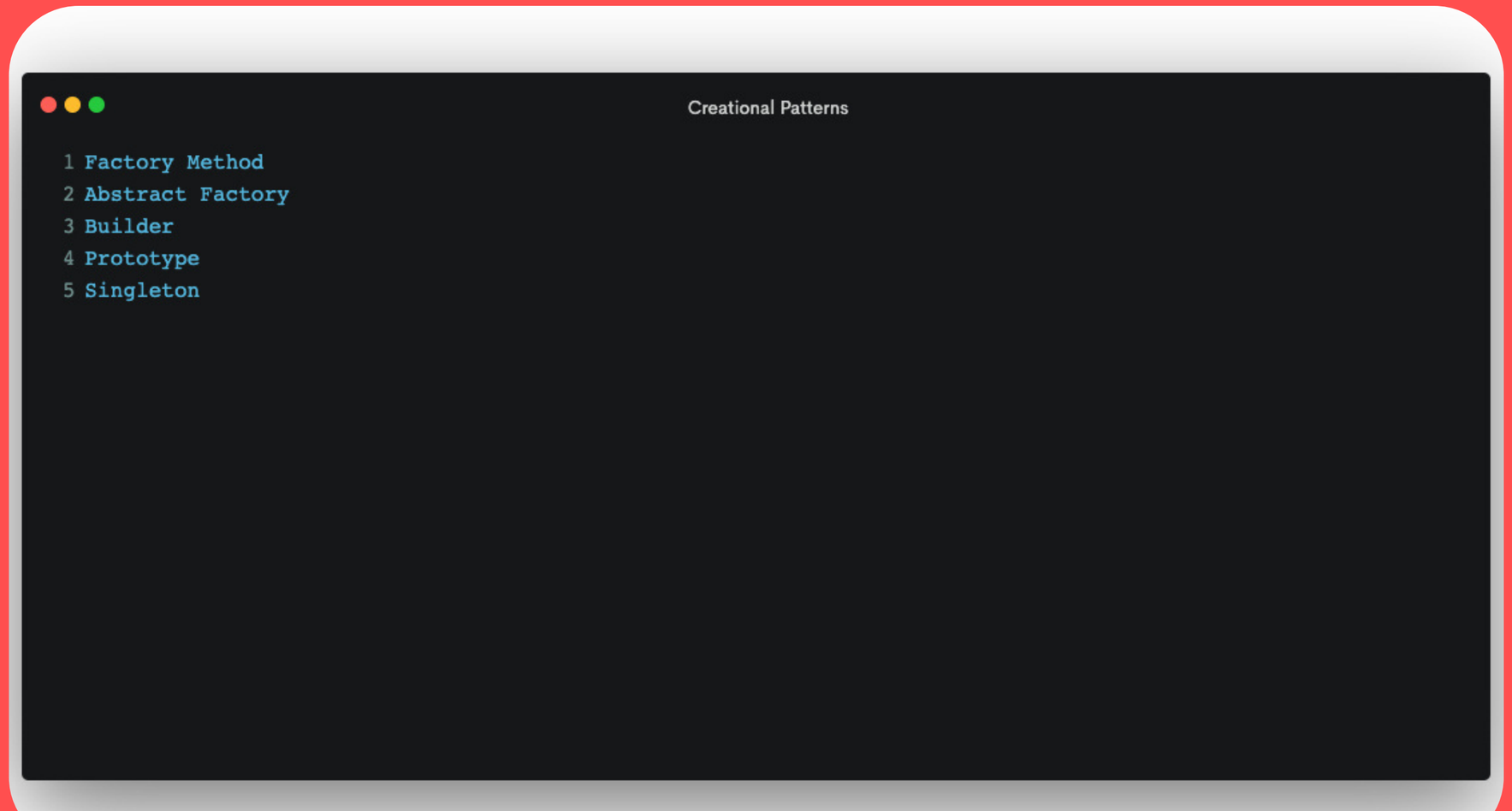
Who Invented Patterns ?

Why Should I Learn Patterns ?

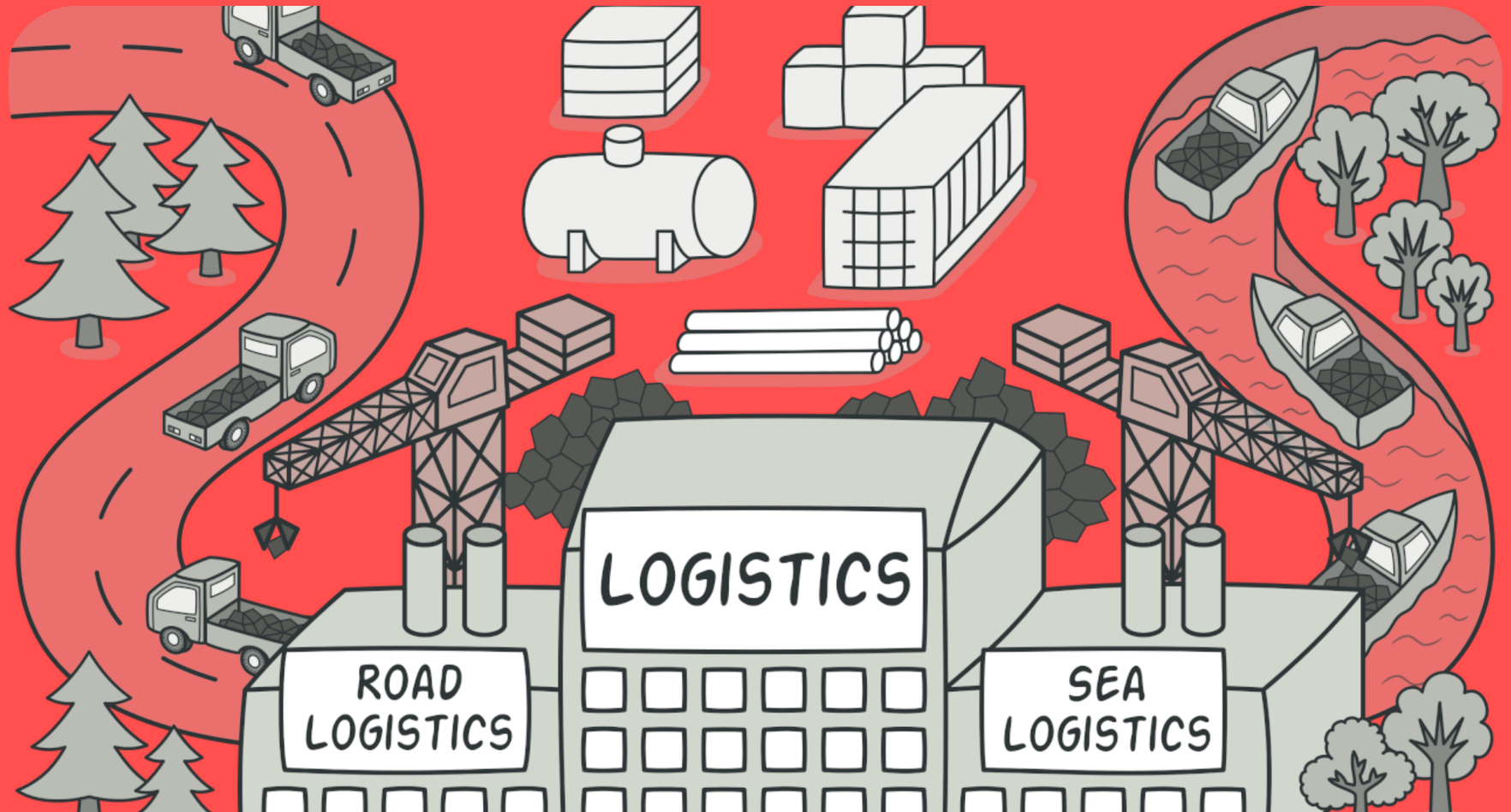
Criticism of patterns

Classification of patterns

Creational Patterns



Factory Method



Factory Method

```ruby
class Creator
  def factory_method
    raise NotImplementedError, "#{self.class} has not implemented method '#{__method__}'"
  end
  def some_operation
    product = factory_method
    result = "Creator: The same creator's code has just worked with #{product.operation}"
    result
  end
end

class ConcreteCreator1 < Creator
  def factory_method
    ConcreteProduct1.new
  end
  end

class ConcreteCreator2 < Creator
  # @return [ConcreteProduct2]
  def factory_method
    ConcreteProduct2.new
  end
end

class Product
  # return [String]
  def operation
    raise NotImplementedError, "#{self.class} has not implemented method '#{__method__}'"
  end
end

class ConcreteProduct1 < Product
  # @return [String]
  def operation
    '{Result of the ConcreteProduct1}'
  end
end

class ConcreteProduct2 < Product
  # @return [String]
  def operation
    '{Result of the ConcreteProduct2}'
  end
end

def client_code(creator)
  print "Client: I'm not aware of the creator's class, but it still works.\n"\
        "#{creator.some_operation}"
end

puts 'App: Launched with the ConcreteCreator1.'
client_code(ConcreteCreator1.new)
puts "\n\n"

puts 'App: Launched with the ConcreteCreator2.'
client_code(ConcreteCreator2.new)
```

Factory Method

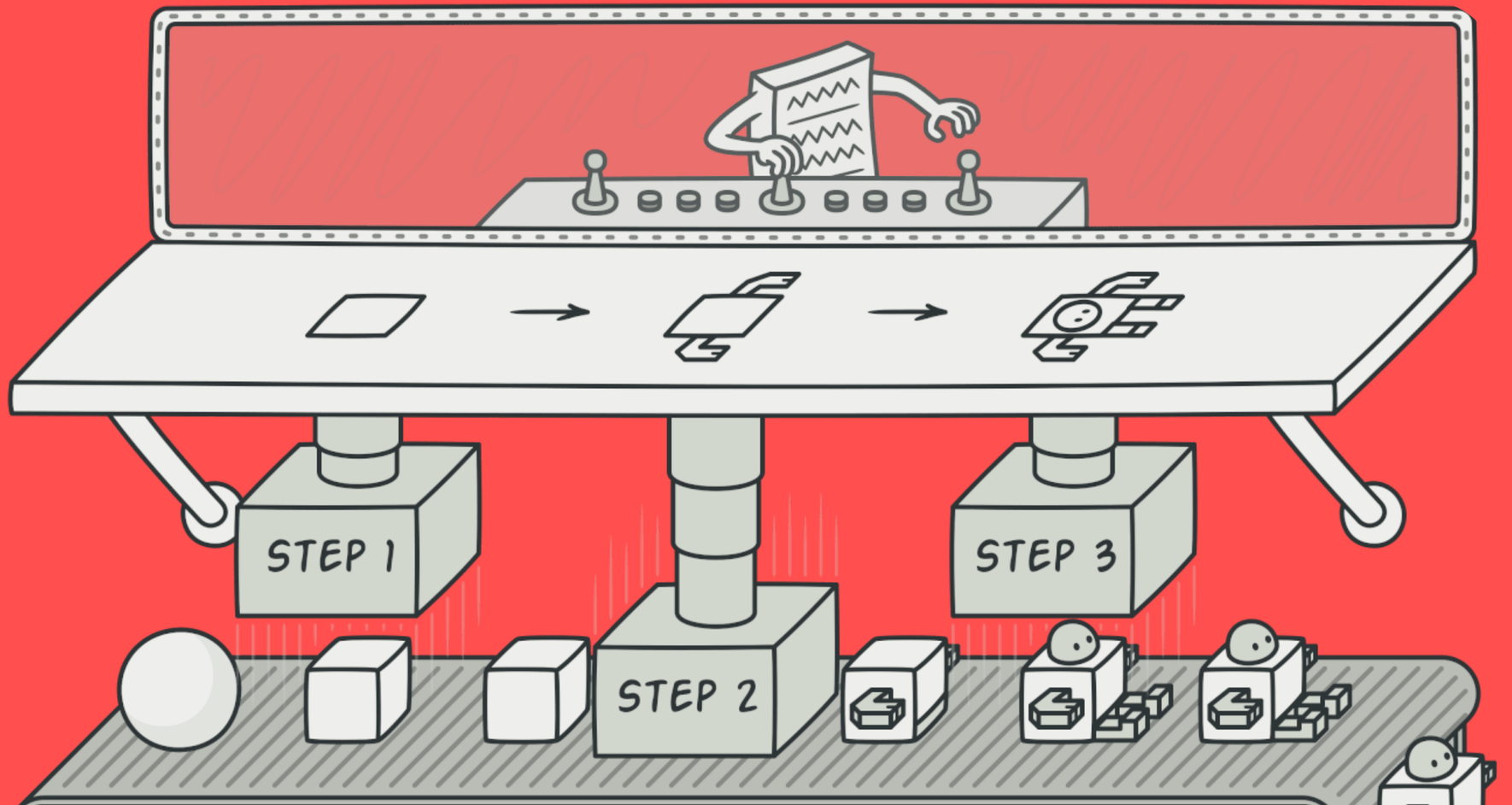
Abstract Factory

Abstract Factory Method

```ruby
class AbstractFactory
  def create_product_a
    raise NotImplementedError, "#{self.class} has not implemented method '#{__method__}'"
  end

  def create_product_b
    raise NotImplementedError, "#{self.class} has not implemented method '#{__method__}'"
  end
end

class ConcreteFactory1 < AbstractFactory
  def create_product_a
    ConcreteProductA1.new
  end

  def create_product_b
    ConcreteProductB1.new
  end
end

class ConcreteFactory2 < AbstractFactory
  def create_product_a
    ConcreteProductA2.new
  end

  def create_product_b
    ConcreteProductB2.new
  end
end

class AbstractProductA
  def useful_function_a
    raise NotImplementedError, "#{self.class} has not implemented method '#{__method__}'"
  end
end

class ConcreteProductA1 < AbstractProductA
  def useful_function_a
    'The result of the product A1.'
  end
end

class ConcreteProductA2 < AbstractProductA
  def useful_function_a
    'The result of the product A2.'
  end
end

class AbstractProductB
  def useful_function_b
    raise NotImplementedError, "#{self.class} has not implemented method '#{__method__}'"
  end
```

Abstract Factory Method

Builder

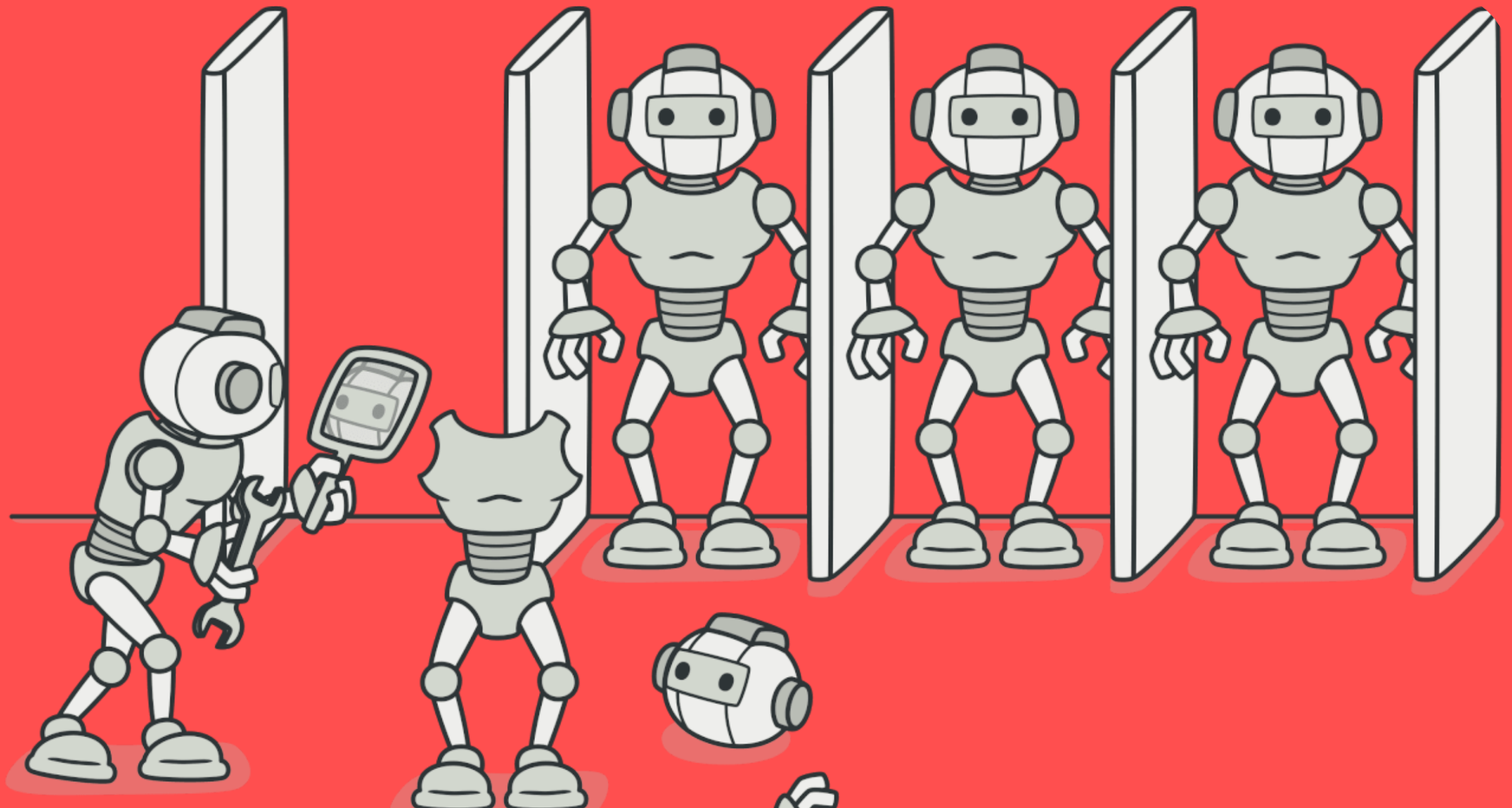


Builder

```ruby
class Builder
  # @abstract
  def produce_part_a
    raise NotImplementedError, "#{self.class} has not implemented method '#{__method__}'"
  end

  # @abstract
  def produce_part_b
    raise NotImplementedError, "#{self.class} has not implemented method '#{__method__}'"
  end

  # @abstract
  def produce_part_c
    raise NotImplementedError, "#{self.class} has not implemented method '#{__method__}'"
  end
end

class ConcreteBuilder1 < Builder
  def initialize
    reset
  end

  def reset
    @product = Product1.new
  end

  def product
    product = @product
    reset
    product
  end

  def produce_part_a
    @product.add('PartA1')
  end

  def produce_part_b
    @product.add('PartB1')
  end

  def produce_part_c
    @product.add('PartC1')
  end
end

class Product1
  def initialize
    @parts = []
  end

  # @param [String] part
  def add(part)
```

Builder in Ruby

Prototype

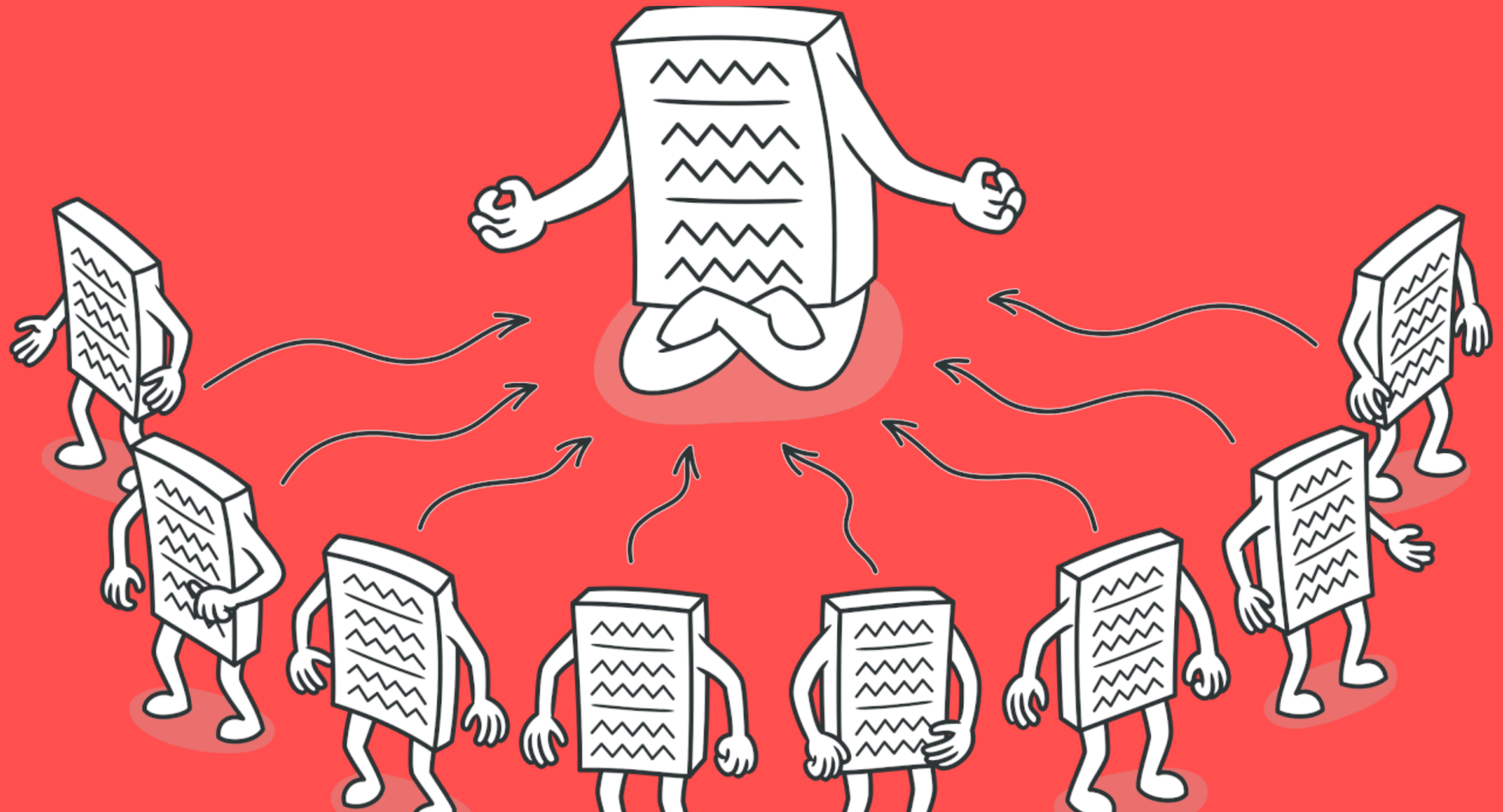


Prototype

```ruby
class Prototype
  attr_accessor :primitive, :component, :circular_reference

  def initialize
    @primitive = nil
    @component = nil
    @circular_reference = nil
  end

  # @return [Prototype]
  def clone
    @component = deep_copy(@component)
    @circular_reference = deep_copy(@circular_reference)
    @circular_reference.prototype = self
    deep_copy(self)
  end

  private def deep_copy(object)
    Marshal.load(Marshal.dump(object))
  end
end

class ComponentWithBackReference
  attr_accessor :prototype

  # @param [Prototype] prototype
  def initialize(prototype)
    @prototype = prototype
  end
end

# The client code.
p1 = Prototype.new
p1.primitive = 245
p1.component = Time.now
p1.circular_reference = ComponentWithBackReference.new(p1)

p2 = p1.clone

if p1.primitive == p2.primitive
  puts 'Primitive field values have been carried over to a clone. Yay!'
else
  puts 'Primitive field values have not been copied. Booo!'
end

if p1.component.equal?(p2.component)
  puts 'Simple component has not been cloned. Booo!'
else
  puts 'Simple component has been cloned. Yay!'
end

if p1.circular_reference.equal?(p2.circular_reference)
  puts 'Component with back reference has not been cloned. Booo!'
else
  puts 'Component with back reference has been cloned. Yay!'
end

if p1.circular_reference.prototype.equal?(p2.circular_reference.prototype)
  print 'Component with back reference is linked to original object. Booo!'
else
  print 'Component with back reference is linked to the clone. Yay!'
end
```

Prototype

Singleton



Singleton

```ruby
class Singleton
  attr_reader :value

  @instance_mutex = Mutex.new

  private_class_method :new

  def initialize(value)
    @value = value
  end

  def self.instance(value)
    return @instance if @instance

    @instance_mutex.synchronize do
      @instance ||= new(value)
    end

    @instance
  end

  def some_business_logic
    # ...
  end
end

def test_singleton(value)
  singleton = Singleton.instance(value)
  puts singleton.value
end

puts "If you see the same value, then singleton was reused (yay!)\n"\
     "If you see different values, then 2 singletons were created (booo!!)\n\n"\
     "RESULT:\n\n"

process1 = Thread.new { test_singleton('FOO') }
process2 = Thread.new { test_singleton('BAR') }
process1.join
process2.join
```

Singleton

Thanks

Pisano