Design Patterns In Ruby and Rails (part 1)

By John Fitzpatrick

Enable Labs

Patterns

- Design Patterns: Elements of Reusable Object-Oriented Software
  - Erich Gamma
  - Richard Helm
  - Ralph Johnson
  - John Vlissides

Some Low-Hanging Fruit

- Factory Method – “Define and interface for creating an object, but let subclasses decide which classes to instantiate” ***Object.new*** is an example.
- Iterator – “Provide a method to access elements of an aggregate object without exposing its underlying representation” ***Object.each*** is an example.

Singleton

“Ensure a class only has one instance, and provide a global point of access to it.”

Singleton Examples

Constant

```
LOGGER = Logger.new
```

Global

```
$logger = Logger.new
```

Module

```
class Logger
      include Singleton
end
```

Class

```
class Logger
      self.log(msg)
            @@log ||= File.open(“log.txt”, “a”)
            @@log.puts(msg)
      end
end
```

Adaptor

“Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces”

Adapter Examples

Pure

```
class Thing
      def move
      end
      def stop
      end
end

class ThingAdapter
      def initialize(thing)
            @thing = thing
      end

      def go
            @thing.move
      end

      def method_missing(name, *args)
            @thing.__SEND__(name, *args)
      end
end
```

Reopen

```
Class Thing
      def move
      end
      def stop
      end
end

class Thing
      def go
            move
      end
end
```

Extend

```
Class MetricRuler
      def measure
            #length in meters
      end
end

class ImperialRuler < Metric Ruler
      alias_method :met_measure :measure

      def measure
            met_measure * 1.0936133
      end
end
```

- Pure – when you do not want to or cannot modify the existing class definition.
- Reopen – when the method you want to call doesn't exist in the class definition.
- Extend – when the method exists but the output is not suitable for the client.

Template Method

“Defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps on an algorithm without changing the algorithm’s structure.“

Template Method Example

Definitions

```
class Game
      def setup_board
      end

      def play
      end

      def put_away
      end

      def play_game
            setup_board
            play
            put_away
      end
end
```

```
class Chess < Game
      def setup_board
            # arrange chess pieces
      end
end

class Monopoly < Game
      def setup_board
            # shuffle cards
            # distribute money
      end
end
```

Usage

```
game = Monopoly.new
game.play_game
```

Abstract Factory

“Provide an interface for creating families of related or dependent objects without specifying their concrete classes.“

Abstract Factory Example

Definitions

```
module AbstractGameFactory
      def create(title)
            raise NotImplementedError, “Needs to be implemented”
      end
end

class RpgGameFactory
      include AbstractGameFactory

      def create(title)
            Rpg.new title
      end
end

class BoardGameFactory
      include AbstractGameFactory

      def create(title)
            Board.new title
      end
end
```

```
class Game
      def initialize(title)
            this.title = title
      end
end

class Rpg
end

class Board
end
```

Usage

```
games = []
game_list = [{type: “rpg”, name: “World of Warcraft”}, {type: “board”, name: “Monopoly”}]
game_list.each do |game|
      if game[:type] eq “rpg”
            games << RpgFactory.create game[:title]
      elsif game[:type] eq “board”
            games << BoardFactory.create game[:title]
      end
end
```

Rubyish Abstract Factory

Definitions

```
module AbstractGameFactory
      def create(type, title)
            raise NotImplementedError, “Needs to be implemented”
      end
end

class GameFactory
      include AbstractGameFactory

      def create(type, title)
            klass = Object.const_get(type.to_s.capitalize)
            klass.new title
      end
end
```

```
class Game
      def initialize(title)
            this.title = title
      end
end

class Rpg < Game
end

class Board < Game
end
```

Usage

```
games = []
game_list = [{type: “rpg”, name: “World of Warcraft”}, {type: “board”, name: “Monopoly”}]
game_list.each do |game|
      games << GameFactory.create game[:type], game[:title]
end
```

Chain of Responsibility

“Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.”

Ruby’s method dispatch is a form of Chain of Responsibility.

Chain of Responsibility Example

Definitions

```
class Link
      def initialize(next_in_line)
            @next_in_line = next_in_line;
      end

      def process(request)
            return if handle(request)
            @next_in_line.process(request)
      end

      def handle(request)
            raise NotImplementedError “error”
      end
end
```

```
class FirstLink < Link
      def handle(request)
            if request instanceof String
                  puts ”It’s a string”
                  return true
            end
            false
      end
end

class SecondLink < Link
      def handle(request)
            if request instanceof Double
                  puts “It’s a double”
                  return true
            end
            false
      end
end

class ThirdLink < Link
      def handle(request)
            puts “It’s something else”
            true
      end
end
```

Usage

```
irb> chain = FirstLink.new(SecondLink.new(ThirdLink.new(nil)))
irb> chain.process String.new(“Hi”)
  It’s a string
irb> chain.process Double.new(“1.00”)
  It’s a double
irb> chain.process nil
  It’s something else
```

Chain of Responsibility, con’t

Ruby allows us to do something interesting with Chain of Responsibility:

```
def method_missing(name, *args)
  @next_in_line.__SEND__(name, *args)
End
```

With this, “links” in the chain do not have to have the same interface.

Strategy Pattern

“Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it”

Strategy Pattern Example

Definitions

```
class CashPaymentStrategy
	def process(order)
		# process cash payment
	end
end

class CreditCardPaymentStrategy
	def process(order)
		# do processing via merchant account
	end
end

class PaypalPaymentStrategy
	def process(order)
		# do processing via paypal
	end
end

class PaymentStrategyFactory
	self.create(order)
		if order.payment_method eq :credit_card
			return CresitCardPaymentStrategy.new
		elsif order.payment_method eq :paypal
			return PaypalPaymentStrategy.new
		end
		CashPaymentStrategy.new
	end
end
```

Usage

```
order = Order.new
payment_strategy = PaymentStrategyFactory.create(order.payment_method)
payment_strategy.process(order)
```

Bridge (a.k.a Double Abstraction)

“Decouples an abstraction from its implementation so that the two can vary independently.“

Bridge Example

Definitions

```
class Vehicle
      def initialize(movement)
            @movement = movement
      end

      def forward(distance)
            @movement.forward distance
      end
end

class Car < Vehicle
      def forward(distance)
            super distance
      end
end

class Bicycle < Vehicle
      def forward(distance)
            super distance
      end
end
```

```
class Movement
      def forward(distance)
            # move forward
      end
end

class SlowedMovement < Movement
      def forward(distance)
            super (distance * .75)
      end
end

class FastMovement < Movement
      def forward(distance)
            super (distance * 1.5)
      end
end
```

Usage

```
car = Car.new(FastMovement.new)
car.forward

bicycle = Bicycle.new(SlowedMovement.new)
bicycle.forward
```

Rubyish Bridge Example

Definitions

```
require ‘rubygems’
require ‘active_support’

class Vehicle
	include Moveable, Turnable

	def initialize(move, turn)
		self.extend move.to_s.camelize.constantize
		self.extend turn.to_s.camelize.constantize
	end

	def location
		puts “X = #{x}, Y = #{y}”
	end
end

class Car < Vehicle
end

class Bicycle < Vehicle
end
```

Usage

```
irb> car = Car.new(:fast_mover, :wide_turner)
irb> car.go(50).turn(50,:left,45).go(50).location
   X = 196, Y = 84
irb> bicycle = Bicycle.new(:slow_mover, :tight_turner)
irb> bicycle.go(50).turn(50,:left,45).go(50).location
   X = 70, Y = 80
```

```
module Moveable
	def go(distance)
		# move forward
	end
end

module FastMover
	include Moveable
	alias_method :old_go, :go

	def go(distance)
		old_go (distance * 1.50)
	end
end

module SlowMover
	include Moveable
	alias_method :old_go, :go

	def go(distance)
		old_go (distance * 0.75)
	end
end

module Turnable
	def turn(direction, angle)
		# change direction
	end
end

module TightTurner
	def turn(distance, direction, angle)
		# turn quick
	end
end

module WideTurner
	def turn(distance, direction, angle)
		# turn slowly
	end
end
```

Credits

- Quotes about each pattern come straight out of the Design Patterns book.
- While I’ve tried to come up with unique examples, many are inspired by existing blog posts and wiki pages.

There's More

Repo: http://github.com/john-fitzpatrick/design_patterns

Design Patters Will Return!