Design Pattern in Ruby

1st Design Pattern:Singelton

Singleton is a design pattern that restricts instantiation of a class to only one instance that is globally available. It is useful when you need that instance to be accessible in different parts of the application, usually for logging functionality, communication with external systems, database access, etc.

**Example:**

class Logger

def initialize

@log = File.open("log.txt", "a")

end

@@instance = Logger.new

def self.instance

return @@instance

end

def log(msg)

@log.puts(msg)

end

private\_class\_method :new

end

Logger.instance.log('message 1')

inside class Logger we create instance of the very same class Logger and we can access that instance with class method (Logger.instance) whenever we need to write something to the log file using the instance method (log) In the (initialize) method we just opened a log file for appending, and at the end of Logger class, we made method (new) private so that we cannot create new instances of class Logger.

**The same example with singleton module:**

require 'singleton'

class Logger

include Singleton

def initialize

@log = File.open("log.txt", "a")

end

def log(msg)

@log.puts(msg)

End

end

Logger.instance.log('message 2')

Here we require and include (Singleton) module inside (Logger) class, define (Initialize) method which opens the log file for appending and instance method (log) for writing to that log file. Ruby Singleton module does lazy instantiation ((creates instance from Logger class at the moment when we call (Logger.instance) method)) and not during load time (like in the previous example). Also, Ruby Singleton module makes (new) method private, so we don't have to call (private\\_class\\_method).

2nd Design Pattern:Decorator

*Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.It use instead of inheritance.*

The problems with inheritance include:

* Choices are made statically.
* Clients can’t control how and when to decorate a component.
* Tight coupling.
* Changing the internals of the superclass means all subclasses must change.

**Example of inheritance:**

class Coffee

def cost

0.2

end

end

class CoffeeWithSugar < Coffee

def cost

super + 0.2

end

end

class CoffeeWithMilkAndSugar < Coffee

def cost

super + 0.4 + 0.2

end

end

\*\*\*In Ruby, **including a module is also inheritance.**

**Example of Module:**

module Milk

def cost\_of\_milk

0.4 if milk?

end

end

module Sugar

def cost\_of\_sugar

0.2 if sugar?

end

end

class Coffee

include Milk

include Sugar

def cost

2 + cost\_of\_milk + cost\_of\_sugar

end

end

## [How a decorator works](https://robots.thoughtbot.com/evaluating-alternative-decorator-implementations-in#how-a-decorator-works)?

Using Gang of Four terms, a **decorator** is an object that encloses a **component** object. It also:

* conforms to interface of component so its presence is transparent to clients.
* forwards (delegates) requests to the component.
* performs additional actions before or after forwarding.

This approach is more flexible than inheritance because you can mix and match responsibilities in more combinations and because the transparency lets you nest decorators recursively, it allows for an **unlimited number** of responsibilities.

## The same example with decorator ([Module + extend + super decorator](https://robots.thoughtbot.com/evaluating-alternative-decorator-implementations-in#module--extend--super-decorator)):

**class Coffee**

**def cost**

**2**

**end**

**end**

**module Milk**

**def cost**

**super + 0.4**

**end**

**end**

**module Sugar**

**def cost**

**super + 0.2**

**end**

**end**

**coffee = Coffee.new**

**coffee.extend(Milk)**

**coffee.extend(Sugar)**

**coffee.cost**

**------------------------------------------------------------------------------------------------------------------------------------------------**

3rd Design Pattern:Strategy

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

**Example:**

class Hair

attr\_reader :color, :texture, :length

def initialize(&block) # accepts a code block

@color = "brown"

@texture = "wavy"

@length = "short"

@block = block

end

def apply

@block.call(self) # calls code block with reference to itself

end

end

shampoo = Hair.new { |context| puts "washing #{context.color} hair..." }

shampoo.apply #=> washing brown hair...

conditioner = Hair.new do |context|

puts "since this hair is #{context.length},"

puts "we'll have to use"

puts "#{context.length == "short" ? "Brand A" : "Brand B"}"

end

conditioner.apply #=> since this hair is short, \n we'll have to use \n Brand A

By allowing our context to accept code blocks, we can quickly whip up new strategies at runtime without needing to define any additional class objects.

The Strategy pattern maintains a nice separation of concern between the **context** and **strategy** object(s). Since the **context** has no knowledge of how a **strategy** is implemented, the pattern imposes nearly no restrictions.

When implementing the Strategy pattern it is important to keep in mind how interfaces play a role. In our example, it was the apply method that was critical in making the **context** and **strategy** object(s) communicate. This delegation tradeoff is often a small price to pay compared to a tightly coupled inheritance-based implementation, however.

Beianca P. Yousif