With the Decorator Pattern we create an Object that encloses a Component object. Also delegates to the component and perform additional actions before or after. It is more flexible than inheritance because you can mix the responsibilities in more combinations and for an unlimited number of responsibilities.

First thing I did was to create the Decorator which I can use it to decorate any component. I used Forwardable which allows me to delegate methods out of the 'blue' very handy. I need it to delegate two methods only – cost and description.

```
require 'forwardable'

class CoffeeDecorator
  extend Forwardable

  def_delegators :@component, :description, :cost

  def initialize(component)
    @component = component
  end

end
```

Next I created my four types of coffee Objects – ***Espresso, Decaffeinato, DarkRoastCoffee*** and ***HouseBlendCoffee*** , they were with a description and cost methods returning the relevant information. At this point my decorator came in play. All five Condiments – Sugar, Milk, Soy, Sweetener and Syrup extended the CoffeeDecorator created. The description and cost methods were implemented by appending the relevant description and price to the component. There is an example how it was done:

```
require_relative 'coffee_decorator.rb'

class Soy < CoffeeDecorator
  def description
    @component.description << " soy "
  end
  def cost
    @component.cost + 0.40
  end
end

require_relative 'coffee_decorator.rb'

class Sugar < CoffeeDecorator
  def description
    @component.description << " sugar "
  end
  def cost
    @component.cost + 0.20
  end
end
```

So far so good. Now I needed a DecoratedCoffee object which I can use to return me any type of coffee created with any condiments.

```ruby
require_relative 'coffee_decorator.rb'

class DecoratedCoffee < CoffeeDecorator
  def description
    @component.description
  end
  def cost
    @component.cost
  end
end
```

All what left now was a bit of imagination to make the perfect coffee. There are some test I ran and the outputs for them:

```ruby
decaff = Decaffeinato.new
espresso = Espresso.new
dark_coffee = DarkRoastCoffee.new
house_blend_coffee = HouseBlendCoffee.new

decaff_with_milk = Milk.new(decaff)
decaff_with_milk_sugar = Sugar.new(decaff_with_milk)
my_coffee = DecoratedCoffee.new(decaff_with_milk_sugar)
puts 'My coffee is:'
puts my_coffee.description
puts "and costs: €#{my_coffee.cost}"


decaff_with_syrup = Syrup.new(decaff)
decaff_with_syrup_sugar = Sugar.new(decaff_with_syrup)
my_coffee = DecoratedCoffee.new(decaff_with_syrup_sugar)
puts 'My coffee is:'
puts my_coffee.description
puts "and costs: €#{my_coffee.cost}"


espresso_with_milk = Milk.new(espresso)
espresso_with_milk_sugar = Sugar.new(espresso_with_milk)
my_coffee = DecoratedCoffee.new(espresso_with_milk_sugar)
puts 'My coffee is:'
puts my_coffee.description
puts "and costs: €#{my_coffee.cost}"

my_coffee = DecoratedCoffee.new(espresso)
puts 'My coffee is:'
puts my_coffee.description
puts "and costs: €#{my_coffee.cost}"

my_coffee = DecoratedCoffee.new(decaff)
puts 'My coffee is:'
puts my_coffee.description
puts "and costs: €#{my_coffee.cost}"

dark_coffee_with_milk = Milk.new(dark_coffee)
my_coffee = DecoratedCoffee.new(dark_coffee_with_milk)
puts 'My coffee is:'
puts my_coffee.description
puts "and costs: €#{my_coffee.cost}"

house_blend_coffee_with_soy = Soy.new(house_blend_coffee)
my_coffee = DecoratedCoffee.new(house_blend_coffee_with_soy)
puts 'My coffee is:'
puts my_coffee.description
puts "and costs: €#{my_coffee.cost}"
```

There are the outputs in the correct order:

```
My coffee is:
decaffeinated coffee milk  sugar
and costs: €2.7
My coffee is:
decaffeinated coffee syrup  sugar
and costs: €2.5
My coffee is:
espresso coffee milk  sugar
and costs: €2.2
My coffee is:
espresso coffee
and costs: €1.5
My coffee is:
decaffeinated coffee
and costs: €2.0
My coffee is:
dark roast coffee milk
and costs: €3.4
My coffee is:
house blend coffee soy
and costs: €3.1
```