Universidad Autónoma de Chihuahua

Facultad de Ingeniería

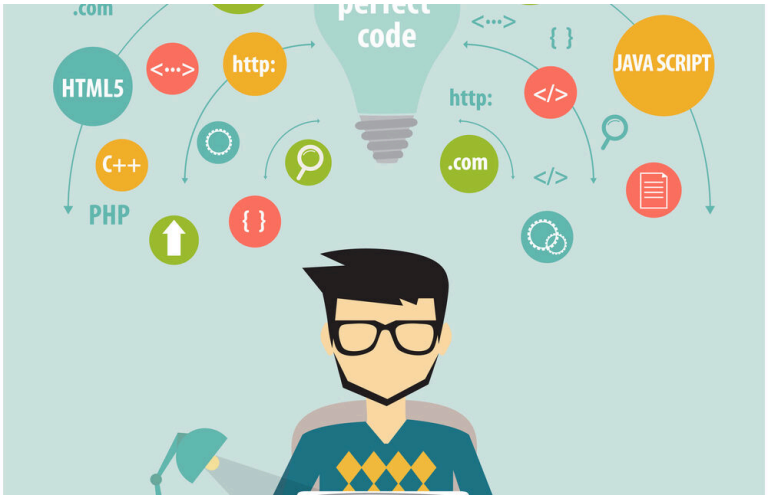
PARADIGMAS DE PROGRAMACIÓN

Docente: Majalca Martínez Ricardo

Grupo: 6CC2

Parcial 3

PROYECTO FINAL DE SEMESTRE

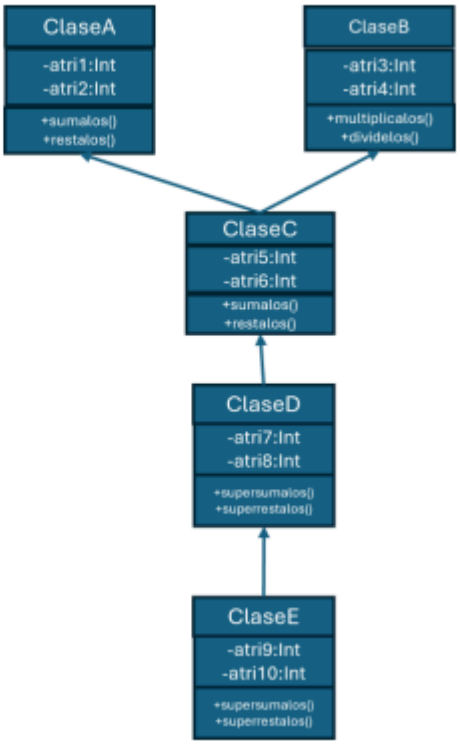


Luz Mariam García Castillo 348409
Emiliano Herrera Dominguez 353259
Gerardo Esteban Jurado Carrera 273880
José Ángel Ortiz Meraz 353195

Ingeniería en Ciencias de la Computación

31/05/2024

(a).- Tenemos la situación ilustrada en el siguiente diagrama de clases UMl.



En esta jerarquía de clases tenemos herencia múltiple, los métodos sumales(), restalos(), multiplicalos() y dividelos() en las clases ClaseA y ClaseB solo suman y restan los respectivos atributos privados en cada clase. Los método sumalos() y restalos() en ClaseC suma y resta los atributos en esa misma clase, tanto los heredados como los allí definidos, naturalmente, estos dos métodos deberán ser rescritos de manera que no son propiamente los métodos que heredaría de sus clases base. De la misma manera, los métodos supersumalos() y superrestalos() en ClaseD suman restan respectivamente todos los atributos, los heredados y los propios en esta clase. Por su parte, los métodos supersumalos() y superrestalos() en ClaseE también suman y restan todos sus atributos, tanto los heredados como los propios, y como era de esperar, estos dos métodos son resrcitos en ClaseE, en lugar de ser los que heredaría de ClaseD. Debemos implementar esto en nuestro lenguaje de programación orientada a objetos favoritos.

Lenguaje Utilizado: Ruby. Por Luz Mariam García Castillo y Gerardo Esteban Jurado Carrera.

```ruby
# El módulo es el que contiene métodos para operaciones en ClaseA
module OperacionesA
  # Método para sumar los atributos @atri1 y @atri2
  def sumalos
    @atri1 + @atri2
  end
```

```ruby
  # Método para restar los atributos @atri1 y @atri2
  def restalos
    @atri1 - @atri2
  end
end

# El módulo que contiene métodos para operaciones en ClaseB
module OperacionesB
  # Método para multiplicar los atributos @atri3 y @atri4
  def multiplicalos
    @atri3 * @atri4
  end

  # Método para dividir los atributos @atri3 y @atri4
  def dividelos
    @atri3 / @atri4
  end
end

# ClaseA que incluye el módulo OperacionesA, que son los métodos propios de la ClaseA
class ClaseA
  include OperacionesA

  # El método initialize define los atributos privados @atri1 y @atri2
  def initialize(atri1, atri2)
    @atri1 = atri1
    @atri2 = atri2
  end
```

```ruby
  private

  # Métodos de acceso privados para los atributos @atri1 y @atri2
  attr_accessor :atri1, :atri2
end

# ClaseB que incluye el módulo OperacionesB, que son los métodos propios de la ClaseB, se utilizará para poder realizar laa herencia múltiplee
class ClaseB
  include OperacionesB

  # El método initialize define los atributos privados @atri3 y @atri4
  def initialize(atri3, atri4)
    @atri3 = atri3
    @atri4 = atri4
  end

  private

  # Métodos de acceso privados para los atributos @atri3 y @atri4
  attr_accessor :atri3, :atri4
end

# ClaseC que hereda de ClaseA y también incluye los métodos de ClaseB (simulando herencia múltiple)
class ClaseC < ClaseA
  include OperacionesB

   # El método initialize define los atributos de ClaseA y nuevos
```

```
atributos @atri5 y @atri6

  def initialize(atri1, atri2, atri3, atri4, atri5, atri6)

    super(atri1, atri2)  # Llama al constructor de ClaseA

    @atri3 = atri3

    @atri4 = atri4

    @atri5 = atri5

    @atri6 = atri6

  end



   # Redefine el método sumalos para incluir todos los atributos de
ClaseC

  def sumalos

    @atri1 + @atri2 + @atri5 + @atri6

  end



   # Redefine el método restalos para incluir todos los atributos de
ClaseC

  def restalos

    @atri1 - @atri2 - @atri5 - @atri6

  end



  private



   # Métodos de acceso privados para los nuevos atributos @atri5 y
@atri6

  attr_accessor :atri5, :atri6

end



# ClaseD que hereda de ClaseC

class ClaseD < ClaseC

   # El método initialize define los atributos de ClaseC y nuevos
```

```ruby
atributos @atri7 y @atri8

    def initialize(atri1, atri2, atri3, atri4, atri5, atri6, atri7,
atri8)

    super(atri1, atri2, atri3, atri4, atri5, atri6)

    @atri7 = atri7

    @atri8 = atri8

  end


   # Método para sumar todos los atributos heredados y los propios de
ClaseD

  def supersumalos

    @atri1 + @atri2 + @atri5 + @atri6 + @atri7 + @atri8

  end


   # Método para restar todos los atributos heredados y los propios de
ClaseD

  def superrestalos

    @atri1 - @atri2 - @atri5 - @atri6 - @atri7 - @atri8

  end


  private


    # Métodos de acceso privados para los nuevos atributos @atri7 y
@atri8

  attr_accessor :atri7, :atri8

end


# ClaseE que hereda de ClaseD

class ClaseE < ClaseD

    # El método initialize define los atributos de ClaseD y nuevos
atributos @atri9 y @atri10

    def initialize(atri1, atri2, atri3, atri4, atri5, atri6, atri7,
```

```
atri8, atri9, atri10)

    super(atri1, atri2, atri3, atri4, atri5, atri6, atri7, atri8)

    @atri9 = atri9

    @atri10 = atri10

  end



  # Redefine el método supersumalos para incluir todos los atributos de
ClaseE

  def supersumalos

     @atri1 + @atri2 + @atri5 + @atri6 + @atri7 + @atri8 + @atri9 +
@atri10

  end



  # Redefine el método superrestalos para incluir todos los atributos
de ClaseE

  def superrestalos

    - @atri1 - @atri2 - @atri5 - @atri6 - @atri7 - @atri8 - @atri9 -
@atri10

  end



  private



   # Métodos de acceso privados para los nuevos atributos @atri9 y
@atri10

  attr_accessor :atri9, :atri10

end



# Método main para encapsular la lógica del programa y ejemplificar

def main

  # Crear una instancia de ClaseE con valores iniciales para todos los
atributos

  e = ClaseE.new(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

```ruby
  # Llamar al método supersumalos y mostrar el resultado
  puts "Resultado de supersumalos: #{e.supersumalos}"  # Output: 48

  # Llamar al método superrestalos y mostrar el resultado
  puts "Resultado de superrestalos: #{e.superrestalos}"  # Output: -48
end

# Llamar al método main para ejecutar el programa
main
```

(b).- Tenemos la necesidad de aplicar las funciones f(x) = x^2 + 25x y g(x) = x^2 + 25x a todos los elementos de una lista de datos enteros, sea cual sea esta lista. Y al hacerlo generamos una segunda lista y una tercera lista. Hagamos esto desde un punto de vista funcional en nuestro lenguaje de programación funcional favorito.

Lenguaje Utilizado: Python. Por José Ángel Ortiz Meraz y Emiliano Herrera Dominguez.

```python
# Definir las funciones lambda
f = lambda x: x**2 + 25 * x
g = lambda x: x**2 + 25 * x

# Aplicar las funciones a una lista usando map
def apply_functions(data_list):
    list_f = list(map(f, data_list))
    list_g = list(map(g, data_list))
    return list_f, list_g

# Uso de ejemplo
data_list = [1, 2, 3, 4, 5]
list_f, list_g = apply_functions(data_list)
```

```
print("Lista después de aplicar f(x):", list_f)

print("Lista después de aplicar g(x):", list_g)
```