Universidade Estadual do Norte Fluminense Darcy Ribeiro
Centro de Ciência e Tecnologia
Programação Orientado a Obejtos (POO)

Relatório N° 1

Relatório Exercícios

Ciência da Computação

Gabriel Costa Fassarella Matrícula: 20211100046

Professora: Annabell Tamariz

12 de setembro de 2023

1 Introdução

Programação orientada a objetos (POO) é um dos principais paradigmas de programação da atualidade, sendo um dos mais utilizados tanto no mundo acadêmico quanto no mercado de trabalho. Sua principal utilidade é a programação buscando uma representação o mais próxima possível da realidade. Visto que, o mundo é formado por objetos, e a orientação a objetos é responsável por abstrair as características e ações do objeto para a utilização durante a programação.

O objetivo desse relatório é relatar como os conceitos aprendidos de orientação a objetos, durante as aulas, foram de extrema importância para o desenvolvimento dos códigos e programas dos exercícios, além do desenvolvimento teórico da criação e abstração de classes e objetos.

2 Procedimento Experimental

- Linguagem: Ruby - IDE: Visual Studio Code (Vscode)

Para a realização do experimento foram utilizados os conhecimentos obtidos em aula para a resolução dos exercícios, e desafios, além disso, também foi utilizado para aprendizado vídeo aulas de ruby do canal one bit code, com a resolução também de outros exercícios dessas mesmas vídeo aulas.

Os exercícios que foram feitos durante os estudos foram:

- EXERCÍCIO 1: Defina classes e objetos:1. Crie um programa que leia 3 inteiros a partir do teclado e determina: (a) o maior (b) o menor (c) o produto (d) a media
- EXERCÍCIO 2: Crie um programa que leia dois inteiros e determine se o primeiro e um multiplo do segundo.
- EXERCÍCIO 3: Crie um programa que leia o raio de um circulo e imprima seu diametro, area e circunferencia.
- EXERCÍCIO 4: Crie uma classe chamada Conta, que represente contas bancarias. A classe deve conter como atributo o saldo da conta (um numero real). A classe deve possuir um construtor que recebe o saldo inicial para inicializar o atributo, validando se o valor e maior ou igual a zero. Caso o valor seja menor que zero, o atributo deve ser inicializado com zero, e uma mensagem de erro deve ser apresentada. Crie um programa que contenha dois objetos desta classe e utilize cada um dos tres metodos: (a) credito: adiciona um valor ao saldo atual (b) debito: subtrai um valor do saldo atual, garantindo que o saldo nao ficara negativo. Se o debito for maior que o saldo, a operacao nao deve ser realizada e uma mensagem apresentada (c) getSaldo: retorna o saldo.
- EXERCÍCIO 5 (One Bit Code): Criar um programa que tenha as classes maratonista e jogador de futebol que herdem da classe esportista.
- EXERCÍCIO 6 (One Bit Code): Criar um programa que simule uma o ato de comprar algum produto de algum programa.

3 Discussão

3.1 Exercício 1:

O primeiro exercício foi feito de maneira bem simples

```
1   class Numeros
2   attr_reader :num1, :num2, :num3
```

```ruby

    def initialize(num1, num2, num3)
      @num = [num1, num2, num3]
    end

    def maior
      maior = @num[0]
      for i in 0..2 do
        if maior <= @num[i]
          maior = @num[i]
        end
      end
      puts "Maior: #{maior}"
    end

    def menor
      menor = @num[0]
      for i in 0..2 do
        if menor >= @num[i]
          menor = @num[i]
        end
      end
      puts "Menor: #{menor}"
    end

    def produto
      prod = @num[0] * @num[1] * @num[2]
      puts "Produto: #{prod}"
    end

    def media
      med = (@num[0] + @num[1] + @num[2]) / 3
      puts "Media: #{med}"
    end

end

def main
    puts "Escreva o valor do primeiro número: "
    num1 = gets.chomp.to_i
    puts "Escreva o valor do segundo número: "
    num2 = gets.chomp.to_i
    puts "Escreva o valor do terceiro número: "
    num3 = gets.chomp.to_i

    nums = Numeros.new(num1, num2, num3)
    nums.maior
```

```
50      nums.menor
51      nums.produto
52      nums.media
53    end
54
55  main
```

Para a criação desse código foi necessário criar uma classe que represente esse conjunto de números, sendo que são passados os valores e esses valores são instanciados dentro de uma lista no interior da classe. Com isso são criados os métodos que executam as operações desejadas, desde de identificar o maior e o menor valor, até as operações exigidas como o produto dos números e a média dos 3 valores. No main é realizada a instanciação do objeto e logo em seguida a chamada dos métodos.

3.2 Exercício 2:

O segundo exercício foi também feito de maneira simples

```
1     class Nums
2     attr_reader :num1, :num2
3
4     def initialize(num1, num2)
5       @num1 = num1
6       @num2 = num2
7     end
8
9     def mult
10      if (num1 % num2).zero?
11        puts "   múltiplo"
12      else
13        puts "Não é múltiplo"
14      end
15    end
16  end
17
18  def main
19    puts "Primeiro número: "
20    num1 = gets.chomp.to_i
21
22    puts "Segundo número: "
23    num2 = gets.chomp.to_i
24
25    multiplo = Nums.new(num1, num2)
26    multiplo.mult
27  end
28
29  main
```

Para a criação desse exercício foi necessário apenas criar uma classe que representasse os dois números, com isso após instanciar os atributos, foi criado o metodo de verificação se um é múltiplo do outro, verificando se o

resto da divisão entre os números é 0 ou não. Com isso no código principal é instanciado o objeto e logo em seguida é realizada a chamada do método de verificação se um é múltiplo do outro.

3.3 Exercício 3:

O exercício 3 funcionou de forma parecida com o 1 e 2, bastou abstrair os atributos da forma geométrica círculo e criar os métodos que calculassem o que foi desejado.

```
    class Circ
    attr_reader :raio

    def initialize(raio)
        @raio = raio
    end

    def diam
        diam = 2*@raio
        puts "Diametro: #{diam}"
    end

    def area
        area = 3.14*@raio**2
        puts "Area: #{area}"
    end

    def compr
        compr = 2*3.14*@raio
        puts "Comprimento: #{compr}"
    end
end

def main
    puts "Raio: "
    raio = gets.chomp.to_i

    circ = Circ.new(raio)

    circ.diam
    circ.area
    circ.compr
end

main
```

Para a realização desse exercício foi necessário efetuar a abstração da forma geométrica, assim criando uma classe que represente o círculo que tenha como atributo o raio, que será instanciado. Após a instanciação do raio do círculo são criados os métodos responsáveis por efetuar os calculos matemáticos desejados, como o cálculo do diametro, comprimento e área, utilizando as suas respectivas fórmulas matemáticas necessárias. Para isso é instanciado um objeto e passado os valores, e longo em seguida é realizada a chamada dos métodos.

3.4 Exercício 4:

O exercício 4 foi o mais próximo do real, foi necessário abstrair uma classe que representasse uma conta bancária, com atributos que representassem valores da conta e metodos que representem a movimentação da conta bancária.

```
1     class Conta_Bancaria
2     attr_reader :saldo
3
4     def initialize(sal)
5         if sal < 0
6             @saldo = 0
7         else
8             @saldo = sal
9         end
10    end
11
12    def credito(cred)
13        @saldo += cred
14    end
15
16    def debito(deb)
17        if deb > @saldo
18            puts "Não é possível realizar a operação"
19        else
20            @saldo = @saldo - deb
21        end
22    end
23
24    def getSaldo
25        puts "Saldo: R$#{saldo}"
26    end
27 end
28
29 def main
30    puts "Saldo: "
31    saldo1 = gets.chomp.to_f
32    conta1 = Conta_Bancaria.new(saldo1)
33
34    while true
35        puts "======================="
36        puts "1) Crédito"
37        puts "2) Débito"
38        puts "3) Mostrar Saldo"
39        puts "4) Sair"
40        op = gets.chomp.to_i
41
42        if op == 1
```

```
43              puts "Valor: "
44              cred = gets.chomp.to_f
45              conta1.credito(cred)
46          elsif op == 2
47              puts "Valor: "
48              deb = gets.chomp.to_f
49              conta1.debito(deb)
50          elsif op == 3
51              conta1.getSaldo
52          else
53              break
54          end
55      end
56
57  end
58
59  main
```

Para a criação desse programa, inicialmente foi necessário criar a classe conta que recebe como atributo o valor que representa o saldo da conta do usuário, quem em seguida realiza uma verificação exigida antes de instânciar o atributo, visto que o o saldo da conta não deve ser inferior a 0. Após isso são criados os métodos responsáveis por efetuar as operações bancárias da conta do usuário, como creditar e debitar um valor.

Já no main, tudo isso é gerenciado por um sistema básico e simples de menu facilitando o manuseio do usuário para creditar e debitar valores, assim como mostrar o saldo atual da conta do usuário.

3.5 Exercício 5:

A realização desse exercício foi simples, ele apresentou conceitos básicos de herança de classes e polimorfismo.

```
1     class Esportista
2     def correr
3       puts "Correndo..."
4     end
5
6     def competir
7       puts "Competindo..."
8     end
9   end
10
11  class JogadorFutebol < Esportista
12    def competir
13      puts "Competindo bola..."
14    end
15  end
16
17  class Maratonista < Esportista
18    def competir
19      puts "Competindo corrida..."
20    end
```

```
21  end
22
23  jogador = JogadorFutebol.new
24  maratonista = Maratonista.new
25
26  jogador.correr
27  maratonista.correr
28  jogador.competir
29  maratonista.competir
```

Para a criação desse programa foi necessário criar inicialmente uma classe extremamente simples chamada esportista, essa classe por sua vez não possui nenhum atributo, mas é responsável por realizar 2 métodos: "correr"e "competir". Após isso foram criadas 2 subclasses que herdam da superclasse esportista: "maratonista"e "jogadorFutebol", classes essas que por meio do polimorfismo, ambas realizam o método "competir"de formas diferentes.

3.6 Exercício 6:

O exercício 6 ainda que simples, apresentou um grau maior de dificuldade, visto que esse exercício utilizou conceitos de modularização.

```
1     #produto.rb:
2  class Produto
3     attr_reader :nome, :preco
4
5     def initialize(nome, preco)
6         @nome = nome
7         @preco = preco
8     end
9  end
10
11    #mercado.rb:
12 require_relative 'produto'
13
14 class Mercado
15    attr_reader :produto
16
17    def initialize(produto)
18        @produto = produto
19    end
20
21    def comprar
22        puts "Você comprou o produto #{produto.nome} no valor
              R$#{produto.preco}"
23    end
24 end
25
26   #app.rb
27 require_relative 'mercado'
```

```
28 require_relative 'produto'
29
30 produto = Produto.new("Biscoito Nikito", 3.99)
31
32 mercado = Mercado.new(produto)
33
34 mercado.comprar
```

Para a realização desse exercício foi necessário realizar a abstração dos atributos de um produto de um mercado e a criação da classe "produtos"em um arquivo separado, com o objetivo de gerar o uso da modularização, ou seja, será necessário organizar as classes em arquivos separados. Para que isso seja feito, será utilizado o comando require-relative que procura o arquivo com o nome requisitado na mesma pasta que o arquivo atual.

Após isso foi criada uma nova classe chamada "mercado"em outro arquivo que recebreu como parâmetro o objeto "produto"já instanciado, servindo assim como um atributo de mercado, para que dessa forma fosse possível executar o método da classe "mercado", que será responsável por imprimir os atributos de "produto".

3.7 Desafio:

O desafio proposto, exigia que fosse criado um programa que representasse um time de futebol. A classe deveria receber o nome, sobrenome, número e posição do jogador como atributos, o nome, número e sobrenome seriam instanciados com valores passados pelo usuário e a posição seria decidida com base no número da camisa do jogador.

```
1     class JogadorDeFutebol
2     attr_reader :nome, :sobrenome, :num, :posicao
3
4     def initialize(nome, sobrenome, num)
5         @nome = nome
6         @sobrenome = sobrenome
7         @num = num
8         @posicao = posicao()
9     end
10
11    def name
12        puts "#{sobrenome} #{nome}".capitalize()
13    end
14
15    def posicao
16        if num >= 1 && num <= 5
17            @posicao = "Zagueiro"
18
19        elsif num >= 6 && num <= 10
20            @posicao = "Meia"
21
22        else
23            @posicao = "Atacante"
24        end
25    end
26
```

```
27      def jogadores
28          puts "#{nome} #{num} #{posicao}"
29      end
30  end
31
32  def cria_jogador()
33      puts "Nome do jogador: "
34      nome = gets.chomp
35
36      puts "Sobrenome: "
37      sobrenome = gets.chomp
38
39      while true
40          puts "Número da camisa: "
41          num = gets.chomp.to_i
42          if num > 0
43              break
44          end
45      end
46
47      jogador = JogadorDeFutebol.new(nome, sobrenome, num)
48      return jogador
49
50  end
51
52  def main
53      time = []
54
55      for i in 0 ..2
56          time[i] = cria_jogador
57      end
58
59      puts "Time do Flamengo (maior do mundo):"
60      time.each do |jogador|
61          puts jogador.jogadores
62      end
63  end
64
65  main
```

Para a criação desse programa foi inicialmente criada a classe JogadorDeFutebol, que receberia os atributos nome, sobrenome e número que são instanciados, e também o atributo posição que decide a posição do jogador por meio de um método que se basea em seu número. Para isso deve seguir a seguinte ideia:

1. o jogador com camisa de 1 até 5 terá sua função na zaga;
2. o jogador com camisa de 6 à 10 estará no meio de campo;
3. o jogador com número maior será atacante;
4. no time não é possível ter uma camisa com numeração igual ou menor que 0, sendo assim, enquanto esse

valor for colocado, o código rodará novamente pedindo um número possível.

A classe também possui um método chamado name que printa o sobrenome e o nome do jogador com a letra inicial maiuscula, por meio do método capitalize. Após isso, foi criada uma função responsável por ler os dados dos jogadores do elenco e instânciar os objetos, os salvando dentro de uma lista que irá formar o elenco do time.

4 Conclusão

Nesse relatório foram realizados inúmeros exercícios referentes ao paradigma de programação orientada a objetos (POO). Com isso, foi possível aprofundar o conhecimento em conceitos de abstração, classes, objetos, polimorfismo e herança, além de aprimorar os conhecimentos de programação em ruby.

Ao longo dos exercícios, foi necessário a criação de programas que explorassem a utilização de diversos conceitos de POO, como principalmente conceitos de abstração, encapsulamento, classes e objetos, e em alguns casos conceitos de herança e polimorfismo. Com isso, foi possível consolidar os conhecimentos e utilização da programação em ruby, e principalmente aprofundar o conhecimento em orientação a objetos.