



Universidade Estadual do Norte Fluminense Darcy Ribeiro
Centro de Ciência e Tecnologia
Laboratório de Ciências Matemáticas

Trabalho 1

Desafios usando Ruby OO

Ciência da Computação

Mariana Cossetti Dalfior

Matrícula: 20211100064

Professor: Annabell Tamariz

14 de setembro de 2023

1 Introdução

A proposta da Programação Orientação a Objetos é representar o mais fielmente possível as situações do mundo real nos sistemas computacionais. Nós entendemos o mundo como um todo composto por vários objetos que interagem uns com os outros. Da mesma maneira, a Orientação a Objetos consiste em considerar os sistemas computacionais não como uma coleção estruturada de processos, mas sim como uma coleção de objetos que interagem entre si. (FARINELLI, 2007)

(TROQUETE, 2019) Muitos dão como o início da programação orientada a objetos por volta de 1970 com a chegada do Smalltalk. Não se pode negar que essa linguagem tenha sido responsável por popularizar o paradigma, porém o SIMULA foi o grande responsável pela concepção da programação orientada a objetos.

2 Procedimento Experimental

- Linguagem: Ruby
- IDE: Visual Studio Code

Para realizar os exercícios propostos será utilizado o IDE Visual Studio Code e a linguagem de programação Ruby. Dessa forma, será utilizado o paradigma orientado a objetos para a realização de todos os exercícios.

3 Contextualização

Esses são os seguintes exercícios propostos:

- Exercício 1: Crie um programa que receba o nome e idade de uma pessoa e no final exiba estes dois dados em uma única frase.
- Exercício 2: Crie um programa que receba dois números inteiros e no final exiba a soma, a subtração, a adição e a divisão entre eles.
- Exercício 3: Utilizando as estruturas de iteração e condição, crie uma calculadora que ofereça ao usuário a opção de multiplicar, dividir, adicionar ou subtrair dois número. Não esqueça da opção de sair do programa.
- Exercício 4: Vamos criar um sistema que tenha:
 - o jogador com camisa de 1 até 5 terá sua função na zaga;
 - o jogador com camisa de 6 à 10 estará no meio de campo;
 - o jogador com número maior será atacante;
 - 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.
- Exercício 5: Criar um programa quem contenha:
 - a classe Esportista com o método competir
 - a classe JogadorDeFutebol com o método correr
 - a classe Maratonista com o método correr
 - as classes JogadorDeFutebol e Maratonista devem herdar comportamento da classe Esportista
- Exercício 6: Simular a compra de um produto em um mercado.

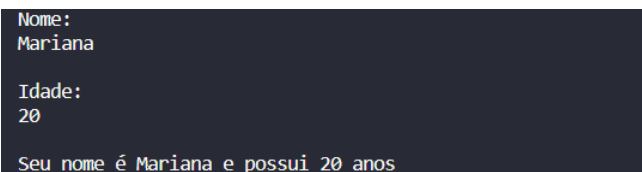
4 Desenvolvimento

4.1 Exercício 1

```
1 class Cadastro
2     def initialize(nome, idade)
3         @nome = nome
4         @idade = idade
5     end
6
7     def mostrar
8         puts "Seu nome é #{@nome} e possui #{@idade} anos"
9     end
10 end
11
12 puts "Nome: "
13 nome = gets.chomp.to_str
14
15 puts "\n"
16 puts "Idade: "
17 idade = gets.chomp.to_i
18
19 cad = Cadastro.new(nome, idade)
20
21 puts "\n"
22 cad.mostrar
```

Primeiro criamos a classe “Cadastro” que possui um construtor chamado “initialize” em Ruby. O construtor pega dois parâmetros e os adiciona como propriedades do objeto usando o símbolo “@”. Podemos entender “@” como uma referência ao próprio objeto, semelhante ao uso do pronome reflexivo “self” em inglês para se referir ao objeto em questão.

Na linha 7, definimos um método chamado “mostrar” que exibe uma mensagem contendo o nome e idade da pessoa. Na linha 19, instanciamos um objeto “Cadastro” o qual será passado informações fornecidas pelo usuário. Por fim, na linha 22, chamamos o “cad.mostrar” para mostrar na tela. O código será executado para exibir na tela o seguinte:



```
Nome:
Mariana

Idade:
20

Seu nome é Mariana e possui 20 anos
```

4.2 Exercício 2

```
1 class Calcular
2     def initialize(n1, n2)
3         @n1 = n1
4         @n2 = n2
```

```

5     end
6
7     def somar
8         puts "A soma desses números resulta em #{@n1+@n2}"
9     end
10
11    def subtracao
12        puts "A subtração desses números resulta em #{@n1-@n2}"
13    end
14
15    def multiplicacao
16        puts "A multiplicação desses números resulta em #{@n1*@n2}"
17    end
18
19    def divisao
20        if @n2 != 0
21            puts "A divisão desses números resulta em #{@n1/@n2}"
22        else
23            puts " Nao pode dividir por zero !!! "
24        end
25    end
26 end
27
28
29 puts 'Digite um número inteiro: '
30 n1 = gets.chomp.to_f
31
32 puts "\n"
33
34 puts 'Digite outro número inteiro: '
35 n2 = gets.chomp.to_f
36
37 calculo = Calculo.new(n1, n2)
38
39 puts "\n"
40 calculo.somar
41 calculo.subtracao
42 calculo.multiplicacao
43 calculo.divisao
44 puts "\n"

```

Primeiro criamos a classe "Calculo" que possui um construtor chamado "initialize" e o inicializar a classe passamos dois parâmetros como propriedades @n1 e @n2. Na linha 7, criamos o método somar que retorna um puts com a soma dos dois números do objeto. Da mesma forma é feitos nos próximos três métodos, porém existe um caso especial na divisao, em que primeiro verificamos se @n2 não é zero, porque nenhum número pode ser dividido por zero; Se este quadro for uma tautologia, ambos os valores serão do tipo ponto flutuante para permitir a divisão correta. Agora, se for falso, ele exibirá uma mensagem para o usuário na tela e seu valor de

retorno será nulo. Finalmente, na linha 30 e 35 receberemos o valor do usuário. Depois disso, é chamados todas as métodos da instancia e assim mostrados os resultados na tela. Como é possível observar a seguir:

```
Digite um número inteiro:
2

Digite outro número inteiro:
3

A soma desses números resulta em 5.0
A subtração desses números resulta em -1.0
A multiplicação desses números resulta em 6.0
A divisão desses números resulta em 0.6666666666666666
```

4.3 Exercício 3

```
1 class Calculadora
2   def initialize (n1, n2, operacao)
3     @n1 = n1
4     @n2 = n2
5     @ope = operacao
6   end
7
8   def calcular
9     case @ope
10    when 1
11      resul = @n1 + @n2
12      puts "O resultado dessa soma é: #{resul}"
13    when 2
14      resul = @n1 - @n2
15      puts "O resultado dessa subtração é: #{resul}"
16    when 3
17      resul = @n1 * @n2
18      puts "O resultado dessa multiplicação é: #{resul}"
19    when 4
20      resul = @n1 / @n2
21      puts "O resultado dessa divisão é: #{resul}"
22    when 5
23      resul = @n1 ** @n2
24      puts "O resultado dessa exponenciação é: #{resul}"
25    when 6
26      exit
27    else
28      puts 'Operação inválida'
29    end
30  end
31 end
32
33 puts 'Digite um número inteiro: '
34 n1 = gets.chomp.to_i
```

```

35
36 puts "\n"
37
38 puts 'Digite outro número inteiro: '
39 n2 = gets.chomp.to_i
40
41 puts "\n"
42
43 puts "Digite a operação que deseja realizar\n1 - soma\n2 - subtração\n3 -
    multiplicação\n4 - divisão\n5 - potenciação\n6 - sair"
44 operacao = gets.chomp.to_i
45
46 calculando = Calculadora.new(n1, n2, operacao)
47
48 puts "\n"
49 calculando.calcular

```

O código em Ruby exibe a classe "Calculadora" como a estrutura principal do programa. Nesse caso, a calculadora é representada por um objeto dessa classe. Os objetos de cálculo e outros objetos que podem ser criados no futuro representam instâncias individuais dessa calculadora. As propriedades "n1", "n2" e "ope" são encapsuladas usando o símbolo "@" antes de seus nomes para garantir que suas referências sejam acessíveis somente dentro do escopo da classe. Isso evita alterações acidentais e fornece controle sobre seu valor.

A classe "Calculadora" contém um método básico chamado "calcular". Esse método é responsável por executar cálculos com base nos valores do atributo "ope" que representam as operações matemáticas necessárias. Os resultados são exibidos na tela por meio da função puts. O código usa uma estrutura condicional "Case" para selecionar uma ação a ser executada com base no valor de "ope". Isso pode ser considerado um exemplo de polimorfismo moderado, pois o método de cálculo é capaz de executar diferentes operações e se adaptar ao contexto, dependendo da operação selecionada.

A seguir é possível observar a execução desse código:

```

Digite um número inteiro:
2

Digite outro número inteiro:
3

Digite a operação que deseja realizar
1 - soma
2 - subtração
3 - multiplicação
4 - divisão
5 - potenciação
6 - sair
1

O resultado dessa soma é: 5

```

4.4 Exercício 4

```

1 class Jogador
2     attr_reader :primeiro_nome, :ultimo_nome, :numero_camisa, :posicao
3     def initialize (primeiro_nome, ultimo_nome, numero_camisa)
4         @primeiro_nome = primeiro_nome

```

```

5      @ultimo_nome = ultimo_nome
6      @camisa = numero_camisa
7      @posicao = posicao()
8  end
9
10 def nome
11     print "#{@ultimo_nome}, #{@primeiro_nome}".capitalize!
12 end
13
14 def posicao
15     case @camisa
16     when 1..5
17         pos = " Zagueiro "
18     when 6..10
19         pos = " Meio de campo "
20     else
21         pos = " Atacante "
22     end
23     @posicao = pos
24 end
25
26 def exibir_dados
27     puts "====="
28     print "#{nome()} | Camisa : #{@camisa} | Posição:#{@posicao} \n"
29     puts "====="
30 end
31 end
32
33 class Time
34     attr_reader :time, :jogadores
35     def initialize (nome_time)
36         @time = nome_time
37         @jogadores = []
38     end
39
40     def adicionar_jogador(jogador)
41         @jogadores << jogador
42     end
43
44     def exibir_informacoes
45         puts "====="
46         puts "#{@time}".upcase!
47         puts "====="
48         @jogadores.each_with_index do |jogador , index|
49             puts " Jogador #{index + 1}"
50             jogador.exibir_dados()
51         end

```

```
52     end
53 end
54
55 def criar_jogador ( nome=nil, ultimo_nome=nil, camisa=nil)
56     if nome.nil? && ultimo_nome.nil? && camisa.nil?
57         puts 'Digite o nome '
58         nome = gets.chomp.strip
59         puts 'Ultimo nome '
60         ultimo_nome = gets.chomp.strip
61
62         while true
63             puts 'Informe o número da camisa: '
64             camisa = gets.to_i
65             if camisa > 0
66                 break
67             else
68                 puts "ERRO, tem que ser maior que 0"
69             end
70         end
71     end
72
73     jogador = Jogador.new(nome, ultimo_nome, camisa )
74     return jogador
75 end
76
77 vasco = Time.new (" Vasco ")
78 botafogo = Time.new (" Botafogo ")
79
80 for i in 1..2
81     if i == 1
82         puts "Jogador do Vasco"
83     end
84     jogador = criar_jogador()
85     vasco.adicionar_jogador(jogador)
86 end
87
88 for i in 1..2
89     if i == 1
90         puts " Jogador do Botafogo "
91     end
92     jogador = criar_jogador()
93     botafogo.adicionar_jogador(jogador)
94 end
95
96 vegetti = criar_jogador("Vegetti ", "Pablo ", "99")
97 vasco.adicionar_jogador(vegetti)
98
```



```

99 vasco.exibir_informacoes()
100 botafogo.exibir_informacoes()

```

Para resolver o problema descrito anteriormente, criamos uma classe intitulada "Jogador" que abrange uma série de funções, incluindo "primeiro_nome", "ultimo_nome" e "numero_camisa". Dentro da estrutura de classes, implementamos "attr_reader" para garantir que os atributos possam ser lidos externamente. Na linha subsequente, construímos um construtor que aceita os parâmetros fornecidos durante a instanciação do objeto e os atribui aos atributos da instância.

Incorporado na décima linha está o método "nome", que serve para exibir o nome do jogador em um formato específico. Este formato envolve colocar o sobrenome do jogador antes do primeiro nome.

Na décima quarta linha do código, estabelecemos o método "posicao". Este método avalia a posição de um jogador com base no número da camisa atribuído. Para evitar uma enorme quantidade de instruções "if" aninhadas, implementamos uma estrutura de controle "case". Por fim, na linha 26, introduzimos o método "exibir_dados", que mostra as informações do jogador conforme definidas na tela.

Após definir a classe "Jogador", criamos a classe "Time" para organizar os jogadores de cada equipe. Novamente, usamos "attr_reader" para permitir a leitura das variáveis "@time" e "@jogadores". O construtor da classe "Time" recebe o nome do time armazenado em "@time" e cria uma lista chamada "@jogadores" contendo todos os jogadores do time.

Na linha 40 implementamos o método "adicionar_jogador", que recebe o jogador como parâmetro e o adiciona à lista de jogadores. Observe que estabelecemos uma associação aqui porque estamos passando como parâmetro um objeto instanciado da classe "Jogador".

Na linha 44 definimos o método "exibir_informacoes" que primeiro exibe o nome do time. Então, na linha 48, usamos o método "each_with_index" para iterar na lista de jogadores "@jogadores". Este método percorre cada elemento da lista, armazenando o player e seu índice em "index". Desta forma, podemos visualizar as informações de cada jogador, lembre-se que "jogador" é um objeto e acessamos suas informações através do método "exibir_dados" definido em sua classe.

Agora, saindo da classe, criamos uma função chamada "criar_jogador" para criar jogadores. Esta função recebe três parâmetros: "primeiro_nome", "ultimo_nome" e "camisa". Os parâmetros são opcionais; se não forem fornecidos, receberão "nil" como valor padrão. Essa flexibilidade nos permite criar players sem interação do usuário, já que podemos fornecer valores diretamente nas chamadas de função.

Na linha 56, verificamos se o valor foi fornecido à função. Caso contrário, pediremos ao usuário que o forneça. Na linha 62, implementamos um loop infinito para garantir que o usuário não insira um número de camisa menor ou igual a 0, pois isso foi definido como uma regra inválida. Por fim, criamos uma instância da classe "Jogador", passamos a ela as informações apropriadas e retornamos essa instância como resultado da função.

Finalmente, criamos duas equipes diferentes. A seguir, usamos um loop "for" para criar cada jogador do time do Vasco. Na linha 85 estabelecemos uma associação entre classes, na qual o time do Vasco chama o método "adicionar_jogador", que recebe o jogador criado pela função "criar_jogador". Repita o mesmo processo com a equipe Botafogo. Na linha 96, criamos automaticamente um player sem interação do usuário, conforme descrito na função "criar_jogador". Por fim, exibimos informações sobre cada equipe.

A execução desse código resulta em:

```

=====
VASCO
=====
Jogador 1
=====
Payet, dimitri | Camisa : 99 | Posição: Atacante
=====
Jogador 2
=====
Medel, gary | Camisa : 17 | Posição: Atacante
=====
Jogador 3
=====
Pablo , vegetti | Camisa : 99 | Posição: Atacante
=====
BOTAFOGO
=====
Jogador 1
=====
Costa, diego | Camisa : 19 | Posição: Atacante
=====
Jogador 2
=====
Soares, tiquinho | Camisa : 9 | Posição: Meio de campo
=====

```

4.5 Exercício 5

```

1 class Esportista
2   def competir
3     puts "Participando de uma competição"
4   end
5
6   def correr
7     end
8 end
9
10 class JogadordeFutebol < Esportista
11   def correr
12     puts "Correndo atrás da bola"
13   end
14 end
15
16 class Maratonista < Esportista
17   def correr
18     puts "Percorrendo o circuito"
19   end
20 end
21
22 joana = JogadordeFutebol.new
23 zehlu = Maratonista.new
24
25 puts "Joana está: "
26 joana.correr
27 joana.competir

```

```

28
29 puts "\nJosé está: "
30 zehlu.correr
31 zehlu.competir

```

O código reconhece três classes: Esportista, JogadordeFutebol e Maratonista. A classe Esportista é a classe base da qual as outras duas classes são derivadas. Isso ilustra o conceito de herança, em que uma classe filha herda propriedades e métodos da classe pai. Nesse caso, a subclasse herda os métodos de competir e correr da classe Esportista.

Cada subclasse de JogadordeFutebol e Maratonista implementa o método correr de forma diferente. Esse é um exemplo de polimorfismo, em que métodos com o mesmo nome podem se comportar de forma diferente em classes diferentes. No código, o método correr de cada classe fornece uma descrição específica das ações de corrida dos JogadordeFutebol e dos Maratonista.

São criadas duas instâncias de objeto: joana e zehlu. Ambos são objetos que são subclasses de Esportista e demonstram como criar novos objetos com base em classes existentes. Isso ilustra a flexibilidade da POO para criar e gerenciar objetos em tempo de execução. Os objetos joana e zehlu respondem polimorficamente ao método correr e executam a implementação concreta de cada classe. Isso permite que o código comum seja aplicado a diferentes classes de objetos, tornando o código mais flexível e reutilizável.

O código acima executado:

```

Joana está:
Correndo atrás da bola
Participando de uma competição

José está:
Percorrendo o circuito
Participando de uma competição

```

4.6 Exercício 6

```

1 require_relative 'produto'
2 require_relative 'mercado'
3
4 produto = Produto.new("Chocolate", 5)
5 produto2 = Produto.new("Amaciante", 20)
6
7 compras = Mercado.new(produto)
8 compras.comprar
9 compras = Mercado.new(produto2)
10 compras.comprar

1 class Mercado
2   def initialize(produto)
3     @produto = produto
4   end
5
6   def comprar
7     puts "Você comprou o produto #{@produto.nome} no valor de
          #{@produto.preco} reais"
8   end

```

```

9 end

1  class Produto
2      attr_accessor :nome, :preco
3
4      def initialize(nome, preco)
5          @nome = nome
6          @preco = preco
7      end
8  end

```

O código mostra duas classes principais: "Produto" e "Mercado". A classe é a estrutura básica em POO e é usada para definir objetos e suas propriedades e métodos. As instâncias das classes são criadas no código, por exemplo, "produto" e "produto2" representam produtos específicos e "compras" representa compras no mercado.

Ao definir os atributos da classe, siga o princípio do encapsulamento. Na classe Produto, os atributos nome e preco são encapsulados e protegidos para que esses dados só possam ser acessados por meio de métodos get e set, garantindo assim um controle mais seguro desses atributos.

A classe "Mercado" recebe um objeto da classe "Produto" como parâmetro em seu construtor, estabelecendo assim uma associação entre as duas classes. Isso reflete a capacidade de objetos de diferentes classes trabalharem juntos para executar tarefas específicas.

Embora não seja um exemplo complexo, esse código ilustra uma forma leve de polimorfismo. O método "comprar" da classe "Mercado" é capaz de exibir informações sobre diferentes produtos com base no objeto "Produto" passado como parâmetro. Isso ilustra a capacidade de lidar com objetos de forma polimórfica e de se adaptar às propriedades de cada objeto.

A execução do código é mostrada a seguir:

```

Você comprou o produto Chocolate no valor de 5 reais
Você comprou o produto Amaciante no valor de 20 reais

```

5 Conclusão

No relatório, seis exercícios foram usados para demonstrar a utilização da aplicação de programação orientada a objetos (POO). Esses exercícios ilustraram como o paradigma é empregado, evidenciando a abordagem de modelagem de objetos pré-existentes, que tornou-se um método predominante na programação. Essa metodologia orientada a objetos oferece vários benefícios, incluindo o encapsulamento, a capacidade de reutilizar o código, a manutenção aprimorada e a legibilidade melhorada.

A facilidade com que algo pode ser reutilizado, mantido e lido é um fator importante a ser considerado. A capacidade de reutilizar um item ou material reduz o desperdício e conserva os recursos. A facilidade de manutenção garante que o item possa ser mantido em boas condições com o mínimo de esforço ou despesa. A legibilidade refere-se ao grau em que as informações são claras, compreensíveis e acessíveis ao público-alvo.

O objetivo deste relatório foi mostrar a aplicação prática dos conceitos de programação orientada a objetos (POO). O relatório visava exemplificar como esses conceitos de programação orientada a objetos são utilizados no mundo real para criar e manipular objetos.

Referências

FARINELLI, F. Conceitos básicos de programação orientada a objetos. **Instituto Federal Sudeste de Minas Gerais**, 2007.

TROQUETE, R. Programação orientada a objetos: uma visão conceitual dos elementos de modelagem. Pontifícia Universidade Católica de São Paulo, 2019.