



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

Relatório POO

Uma abordagem orientada a objetos

Ciência da Computação

Gabriel Almeida Matrícula: 20211100143

Professora: Annabell Tamariz

13 de setembro de 2023

1 Introdução

Programação orientada a objetos (POO) é um paradigma de linguagem de programação na qual consistem em criar moldes do mundo real ou virtual em classes. Segundo (Pecinovsky, 2013) O mundo é criado de objetos e POO os generaliza, pegando suas características, eventos, e estados. Ainda, (Weisfeld, Matt 2019) complementando, o descreve que um objeto é uma entidade que contem dados e comportamentos, ou seja, praticamente tudo ao nosso redor.

O (Holmevik, J.R 1994) relata que POO surgiu em 1970 com a linguagem SIMULA, criada por Ole-Johan Dahl e Kristen Nygaard na faculdade Norwegian Compute Centre (NCC) em Oslo durante 1962 e 1967, foi desenvolvida para ser utilizado em eventos de simulação, mas posteriormente expandida e reinventada para linguagem de propósito geral. Foi com SIMULA que foram introduzidos conceitos que são os pilares da programação orientada a objetos como: classes, objetos e herança.

O objetivo desse relatório é descrever, minuciosamente, como foram utilizados os conceitos de POO. Identificando classes para moldar os objetos essenciais no código fonte, determinar e descrever os seus atributos e métodos, as relações entre as classes e por fim, os benefícios da abordagem da POO no código fonte.

2 Procedimento Experimental

O trabalho foi realizado nas seguintes condições:

- Linguagem: Ruby
- IDE: Visual Studio Code

3 Contextualização

São 4 exercícios propostos, sendo os 3 primeiros da programação procedural transformada no paradigma orientado a objetos.

- 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 multiplicação 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.
- EXERCÍCIO 4: Criar uma classe **jogador de futebol** com atributos: **Primeironome**, **ultimoNome** e **camisa**, contendo os métodos **nome**, **posicao**, sendo a posição determinada pelo seu numero da camisa
 - 1-5: Zagueiro
 - 6-10: Meio de campo
 - Numero maior que 10: Atacante
 - Não pode existir número menor ou igual a 0

O código deverá realizar, no mínimo, tais operações:

- Chamar classe jogador
- Estabelecer nome para cada time

- Lista de jogadores de cada time
- Adicionar mais jogadores à lista
- Mostrar todos os jogadores de cada time

4 Desenvolvimento

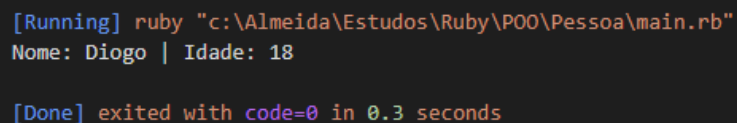
4.1 Exercício 1

O primeiro exercício é bem simples de ser feito. O código a seguir demonstra sua implementação.

```
1  class Pessoa
2  def initialize(nome, idade)
3    @nome = nome
4    @idade = idade
5  end
6
7  def exibir_dados
8    puts "Nome: #{@nome} | Idade: #{@idade}"
9  end
10 end
11
12
13 def main()
14   diogo = Pessoa.new("Diogo", 18)
15   diogo.exibir_dados
16 end
17
18 main()
```

Primeiramente, criamos a classe `Pessoa` que possui seu inicializador, em Ruby é chamado de **initialize** a qual recebe dois parâmetros e adiciona-os como atributos do objeto pelo símbolo `@`. Podemos interpretar o `@` como sendo o `self`, que em inglês é um pronome reflexivo, utilizado para referir a si mesmo, ou seja, nesse contexto ao objeto.

Na linha 7, criamos um método `exibir_dados` que exibe a mensagem com o nome e a idade da pessoa. Na linha 13, fazemos a função `main` e instanciamos uma pessoa `diogo` passando como parâmetro para classe seu nome e sua idade, por fim, na linha 18 chamamos o `main` e o código será executado. Teremos essa exibição na tela:



```
[Running] ruby "c:\Almeida\Estudos\Ruby\P00\Pessoa\main.rb"
Nome: Diogo | Idade: 18
[Done] exited with code=0 in 0.3 seconds
```

Figura 1 – Output Pessoa

4.2 Exercício II

O segundo exercício pede uma calculadora com operações básicas e no final exibi-las.

```
1  class Calculadora
```

```

2      attr_accessor :num1, :num2
3      def initialize(num1, num2)
4          @num1 = num1;
5          @num2 = num2
6      end
7
8      def somar
9          @num1 + num2
10     end
11
12     def subtrair
13         @num1 - @num2
14     end
15
16     def multiplicar
17         @num1 * num2
18     end
19
20     def dividir
21         if @num2 != 0
22             return @num1.to_f / @num2.to_f
23         else
24             puts "Nao pode dividir por zero!!!"
25         end
26     end
27 end
28
29 def main()
30
31     print("n1: ")
32     n1 = gets.to_i
33     print("n2: ")
34     values = Calculadora.new(n1, n2)
35
36     puts "#{n1} + #{n2} é #{values.somar}"
37     puts "#{n1} - #{n2} é: #{values.subtrair}"
38     puts "#{n1} * #{n2} é: #{values.multiplicar}"
39     puts "#{n1} / #{n2} é: #{values.dividir}"
40 end
41
42 main()

```

Primeiro iniciamos criando a classe `Calculadora`, em seguida damos `attr_accessor` para as variáveis `num1` e `num2`, ou seja, podemos acessar e rescreve-las. Ao inicializar a classe, passamos dois parâmetros como atributos `@num1` e `@num2`.

Na linha 8 criamos o método **somar** que retorna a soma dos dois números definidos do objeto. O mesmo esquema se repete nos métodos seguintes, com uma ênfase na linha 20 onde contém o método **dividir**. Nele, nós

temos um caso especial, primeiro verificamos se `@num2` não é zero, pois, nenhum número pode dividir por zero; caso essa afirmação for tautológica, ambos valores obterão a tipagem `float` para permitir divisão corretamente. Agora, se a afirmação for falsa, exibirá uma mensagem para o usuário na tela, e seu retorno será `nil`.

Por fim, na linha 29 teremos a criação do `main`. Entre 31 a 33 receberemos os valores do usuário. (Foi utilizado `print` para não ter quebra de linha; e o `gets.to_i` irá armazenar o valor digitado como inteiro). Posteriormente, temos uma sequência de `puts` que exibe as operações. A seguir, o output do código.

```
=> :main
main.rb(main):042:0> main()
n1: 4
n2: 2
4 + 2 é 6
4 - 2 é: 2
4 * 2 é: 8
4 / 2 é: 2.0
=> nil
```

Figura 2 – Output calculadora

4.3 Exercício III

O princípio é o mesmo do anterior, mas agora teremos um loop para rodar o programa até quando o usuário quiser.

```
1  class Calculadora
2    attr_accessor :num1, :num2
3    def initialize(num1, num2)
4      @num1 = num1;
5      @num2 = num2
6    end
7
8    def somar
9      @num1 + num2
10   end
11
12   def dividir
13     if @num2 != 0
14       @num1 / @num2
15     else
16       puts "Não pode dividir por zero"
17     end
18   end
19
20   def subtrair
21     @num1 - @num2
22   end
```

```
23
24     def multiplicar
25       @num1 * num2
26     end
27
28     def potencia
29       @num1 ** num2
30     end
31
32   end
33
34   class Menu
35     def exbir
36       puts "-----"
37       puts "[1] Adição"
38       puts "[2] Subtração"
39       puts "[3] Divisão"
40       puts "[4] Multiplicação"
41       puts "[5] potência"
42       puts "[0] Mudar valores"
43       puts "[-1] para sair"
44       puts "-----"
45     end
46   end
47
48   # Function made to get values from user
49
50   def get_values
51     puts "Valor de n1"
52     num1 = gets.to_i
53     puts "Valor de n2"
54     num2 = gets.to_i
55
56     calc = Calculadora.new(num1, num2)
57     return calc
58   end
59
60
61   def main()
62     menu = Menu.new()
63
64     calculadora = get_values()
65
66     while true
67       menu.exbir()
68
69       resp = gets.to_i
```

```

70
71     case resp
72     when 1
73     puts "A soma é: #{calculadora.somar}"
74     when 2
75     puts "A subtração é: #{calculadora.subtrair}"
76     when 3
77     puts "A divisão é: #{calculadora.dividir}"
78     when 4
79     puts "A multiplicação é: #{calculadora.multiplicar}"
80     when 5
81     puts "A potência é: #{calculadora.potencia}"
82     when -1
83     break
84     when 0
85     calculadora = get_values()
86     else
87     puts "Digite um comando valido!!!"
88     end
89
90     sleep(1)
91
92     end
93     end
94
95     main()

```

Como o código é o mesmo, iremos apenas comentar das alterações feitas. Na linha 28 foi criado um novo método **potencia** para calcular potência. Na linha 34 criamos uma classe **menu** que sua função é apenas exibir um menu de opções na tela. Já na linha 50, foi feita uma função **get_values** responsável por perguntar ao usuário dois valores inteiros, e criar uma instância da classe chamada de **calc**, no final, retornamos a **calc**.

Na linha 61, criamos o **main** que instância a classe **Menu** e na variável **calculadora** recebe o valor da função **get_values**. Em 66, é criado um loop infinito que sempre exibirá o menu a cada iteração, assim como, armazenar a escolha do usuário de qual opção ele deseja realizar do menu. Em 71, criamos a estrutura **case** que irá realizar as operações desejadas conforme a escolha do menu, note que, caso seja digitado um valor diferente das opções disponíveis, uma mensagem informando que o comando é inválido aparecerá. Na linha 90, utilizamos o **sleep** para fazer o terminal demorar 1 segundo para exibir, utilizamos isso apenas para fins estéticos no terminal, fazendo o menu demora 1 segundo para aparecer.

Segue um exemplo com

```

-----
[1] Adição
[2] Subtração
[3] Divisão
[4] Multiplicação
[5] potência
[0] Mudar valores
[-1] para sair
-----
1
A soma é: 12
-----
[1] Adição
[2] Subtração
[3] Divisão
[4] Multiplicação
[5] potência
[0] Mudar valores
[-1] para sair
-----
2
A subtração é: 8

```

Figura 3 –

```

3
A divisão é: 5
-----
[1] Adição
[2] Subtração
[3] Divisão
[4] Multiplicação
[5] potência
[0] Mudar valores
[-1] para sair
-----
4
A multiplicação é: 20
-----
[1] Adição
[2] Subtração
[3] Divisão
[4] Multiplicação
[5] potência
[0] Mudar valores
[-1] para sair
-----
5
A potência é: 100
-----

```

Figura 4 –

4.4 Exercício IV

Dado o enunciado apresentado anteriormente do problema, foi criado a classe Jogador com os métodos `primeiro_nome`, `ultimo_nome` e `numero_camisa`.

```

1  class Jogador
2      # Criando get metodos
3      attr_reader :primeiro_nome, :ultimo_nome, :numero_camisa, :posicao
4      def initialize(primeiro_nome, ultimo_nome, numero_camisa)
5          @primeiro_nome = primeiro_nome
6          @ultimo_nome = ultimo_nome
7          @numero_camisa = numero_camisa
8          @posicao = posicao()
9      end
10     # Metodo que imprime o nome
11     def nome

```



```

12     print "#{@ultimo_nome} #{@primeiro_nome}".capitalize!
13 end
14
15 # Metodo para determinar posicao
16 def posicao
17     case @numero_camisa
18     when 1..5
19         pos = "Zagueiro"
20     when 6..10
21         pos = "Meio de campo"
22     else
23         pos = "Atacante"
24     end
25
26     @posicao = pos
27 end
28
29 #Metodo para exibir dados
30 def exibir_dados
31     puts "-----"
32     print "#{nome()} | Camisa: #{@numero_camisa} | #{@posicao} \n"
33     puts "-----"
34 end
35 end

```

Dado o enunciado apresentado anteriormente do problema, foi criado a classe Jogador com os métodos **primeiro_nome**, **ultimo_nome** e **numero_camisa**.

Na linha 3, nós temos o **attr_reader** que permite ler os atributos fora da classe. Na linha 4, foi criada o **inicializador** que recebe os parâmetros passados quando o objeto for instanciado e coloca-os nos atributos da instância.

Na linha 11, temos a criação do método **nome**, este que sua função é apenas exibir o nome do **ultimo_nome** do jogador na frente do seu primeiro nome, sendo estes com a inicial maiúscula por causa do método **capitalize!**. Na linha 16, temos o método **posicao** que vai determinar a função do jogador com base no número da sua camisa. Foi utilizado o **case** para evitar uma sequência de **ifs**; ao final, é armazenado na variável de instância **@posicao** a posição do jogador. Por fim, na linha 30, é criado o método **exibir_dados** para mostrar na tela os dados definidos do jogador.

Essa é a nossa primeira classe do código, a seguir, foi criada a classe **Time** para organizar jogador de cada time.

```

1     class Time
2         attr_reader :nome_time, :jogadores_time
3         def initialize(nome_time)
4             @nome_time = nome_time
5             @jogadores_time = []
6         end
7
8         def adicionar_jogador(jogador)
9             @jogadores_time << jogador

```

```

10     end
11
12     def exibir_informacoes
13     puts "-----"
14     puts "#{@nome_time}!".upcase!.red
15     puts "-----"
16     @jogadores_time.each_with_index do |jogador, index|
17     puts "Jogador #{index + 1}".yellow
18     jogador.exibir_dados()
19     end
20     end
21     end

```

Novamente, é criada a classe **Time** com permissão de leitura para as variáveis `nome_time` e `jogadores_time`. A inicialização da classe recebe o nome de um time, o qual será armazenado em `@nome_time`, além de criarmos uma lista `jogadores_time` que irá conter todos jogadores do time.

Na linha 8, criamos o método **adicionar_jogador** que recebe como parâmetro um jogador que irá ser adicionado a lista de jogador. Note que aqui temos uma interação diferente, é passado como parâmetro um objeto instanciado da classe **jogador**, ou seja, aqui temos uma relação de associação.

N linha 12, temos o método **exibir_informações** que irá inicialmente informar o nome do time, após isso, na linha 16, iremos fazer um `each_with_index` na lista de jogadores `@jogadores_time`. O que esse método faz é simples, ele vai percorrer cada elemento da minha lista e armazenar na variável `jogador` e um índice de numeração em `index`. Assim vamos exibir cada jogador e seu respectivo dado, lembre-se, `jogador` é um objeto! Estamos acessando `jogador` e utilizando o método definido na sua classe **exibir_dados**.

Saindo das classes, vamos criar agora uma função para criar os jogadores.

```

1     # Funcao para criar jogadores
2     def criar_jogador(nome=nil, ultimo_nome=nil, camisa=nil)
3     # Verificacao para criar jogadores caso os dados nao forem passados na
4     # chamada da funcao
5     if nome.nil? && ultimo_nome.nil? && camisa.nil?
6
7     #Pegar dados com o usuario
8     puts 'Digite o nome'
9     nome = gets.chomp.strip # Chomp para eliminar \n e strip para remover
10    # espaços em branco no final
11    puts 'Ultimo nome'
12    ultimo_nome = gets.chomp.strip
13    # Loop para obter camisa apenas maior que 0
14
15    while true
16    puts 'Informe a camisa: '
17    camisa = gets.to_i
18    if camisa > 0
19    break
20    else
21    puts "ERRO, precisa ser maior que 0"
22    end

```

```

21     end
22     end
23
24     # Cria jogador e retorna
25     jogador = Jogador.new(nome, ultimo_nome, camisa)
26     return jogador
27 end

```

Definimos a função como **criar_jogadores** a qual recebe três parâmetros **nome**, **ultimo_nome** e **camisa**, os parâmetros são opcionais, caso não seja passado nenhum na chamada da função, eles receberão **nil** como valor. Isso foi feito para permitir que o código crie jogadores sem precisar passar pelo usuário, ou seja, na própria chamada da função já passar os valores.

Na linha 4, verificamos se já foi passado os valores para função, caso seja falso, pedirá ao usuário para informar os valores. Na linha 13, é feito um loop infinito para caso o usuário digite uma camisa \leq a 0, o qual foi definido como regra não ter camisa nesse intervalo, e será exibido uma mensagem para situar o usuário. No final, criamos a instância da classe **Jogador** e passamos suas informações, e a função retornará essa instância.

Agora só falta criar o **main**

```

1     def main
2         # Lista de jogadores
3         fluminense = Time.new("Fluminense")
4         botafogo = Time.new("Botafogo")
5
6         # Cria jogadores do fluminense e adiciona no time
7         for i in 1..2
8             if i == 1
9                 puts "Jogador do FLUMINENSE"
10            end
11            jogador = criar_jogador()
12            fluminense.adicionar_jogador(jogador)
13        end
14
15        # Cria jogadores do botafogo e adiciona no time
16        for i in 1..2
17            if i == 1
18                puts "Jogador do BOTAFOGO"
19            end
20            jogador = criar_jogador()
21            botafogo.adicionar_jogador(jogador)
22        end
23
24        # Cria um jogador automatico sem perguntar ao usuario os dados
25        cano = criar_jogador("Cano", "German", "87")
26
27        fluminense.adicionar_jogador(cano)
28
29        #Exibe dados de cada time
30

```

```

31     fluminense.exibir_informacoes()
32     botafogo.exibir_informacoes()
33     end
34
35     main()

```

No **main** já nas primeiras linhas criamos dois times diferentes. Em seguida fazemos um **for** para criar cada jogador do time FLUMINENSE e na linha 12, é feito de fato o que comentamos anteriormente, a relação de associação entre as classes. O time **fluminense** invoca o método **adicionar_jogador** que recebe o jogador criado na função **criar_jogador**. O mesmo é feito para o time do BOTAFOGO. Em 25, criamos um jogador automaticamente sem ter que pedir ao usuário as informações, como foi explicado dentro do código da função **criar_jogador**. No final, exibimos informações de cada time.

O código exibido será assim:

```

-----
FLUMINENSE
-----
Jogador 1
-----
Ganso paulo | Camisa: 10 | Meio de campo
-----
Jogador 2
-----
Arias john | Camisa: 11 | Atacante
-----
Jogador 3
-----
German cano | Camisa: 87 | Atacante
-----
-----
BOTAFOGO
-----
Jogador 1
-----
Suares tiquinho | Camisa: 9 | Meio de campo
-----
Jogador 2
-----
Perri lucas | Camisa: 1 | Zagueiro
-----

```

Figura 5 – output jogadores

5 Conclusão

O relatório descreveu a aplicação POO em 4 exercícios, demonstrando como o paradigma é utilizado para modelar objetos existentes. A abordagem orientada a objetos permite vantagens como: Encapsulamento, reutilização de código, facilidade para manutenção e legibilidade.

Além de notar a diferença entre o POO e o procedural, sendo este último, o primeiro paradigma ensinado na maioria das universidades, onde seu foco está na ação, diferente de POO o qual seu foco está na modelagem

do objeto. Portanto, esse relatório buscou demonstrar na prática como são aplicadas os conceitos de POO.

6 Referências

PECINOVSKÝ, Rudolf. OOP - Aprenda o Pensamento e a Programação Orientados a Objetos. Walnut Creek, CA: Tomas Bruckner, 2013. 528 p

WEISFELD, Matt. Processo de Pensamento Orientado a Objetos. 5. ed. Addison-Wesley Professional, 2019. 240 p. ISBN 978-0135181966.

HOLMEVIK, J.R. "Compilação do SIMULA: um estudo histórico da gênese tecnológica."IEEE Annals of the History of Computing, v. 16, n. 4, 1994, p. 25-37. DOI: 10.1109/85.329756.