Linguagem Ruby integrada ao uso de LLM

Giulia Moura Ferreira, 20/00018795

[1]Dep. Ciência da Computação – Universidade de Brasília (UnB)
CIC0105 - Engenharia de Software

giulia.ferreira@aluno.unb.br

1. Introdução

O desenvolvimento em Ruby apresenta uma oportunidade única para explorar diversos paradigmas de programação, como o funcional e o orientado a objetos. Para a resolução dos exercícios de Ruby deste trabalho, utilizamos a IDE RubyMine, que facilitou a depuração e organização do código, essenciais para alcançar os resultados esperados. Além disso, o apoio dos Modelos de Linguagem de Grande Escala (LLMs), como o ChatGPT, revelou-se fundamental na redação e na análise crítica das atividades, oferecendo uma visão mais ampla sobre abordagens alternativas de implementação.

Os LLMs têm se mostrado ferramentas valiosas no aprendizado de Ruby ao auxiliarem tanto na solução de problemas quanto na otimização do código. Por meio de sugestões de melhoria e da exploração de conceitos fundamentais, como blocos e metaprogramação, eles permitem uma imersão mais direcionada nos recursos de Ruby. Dessa forma, é possível desenvolver soluções mais eficazes e concisas, aprimorando tanto o domínio da linguagem quanto a capacidade de adaptação a diferentes abordagens de programação.

O repositório das atividades pode ser encontrado a partir desse link.

2. Desenvolvimento

2.1. Parte 1: Diversão com Strings

2.1.1. Método palindrome?

O objetivo é implementar um método que verifica se uma palavra ou frase é um palíndromo.

Para isso, o método *palindrome?* normaliza a entrada (converte para minúsculas e remove caracteres não alfanuméricos) e então compara a string com sua versão invertida. Se ambas forem iguais, o método retorna *true*, indicando que a palavra ou frase é um palíndromo; caso contrário, retorna *false*.

```ruby
def palindrome?(string)
  normalized_string = string.downcase.gsub(/\W/, '')
  puts normalized_string == normalized_string.reverse
end
```

Figure 1. Implementação da Parte 1.1

Abaixo está uma explicação das estruturas utilizadas e do algoritmo implementado:

- **`string.downcase`**: Converte todos os caracteres para minúsculas, garantindo uma comparação sem distinção de maiúsculas e minúsculas (case-insensitive), de modo que uma palavra em maiúsculas seja considerada equivalente a uma em minúsculas.
- **`gsub(/\W/, '')`**: Remove todos os caracteres não alfanuméricos, substituindo-os por uma string vazia. Utiliza a expressão regular *W*, que seleciona caracteres que não são letras, números ou sublinhados (_).
- **`normalized_string.reverse`**: Inverte a string normalizada para realizar a comparação.
- **`puts normalized_string == normalized_string.reverse`**: Realiza a comparação e imprime o resultado.

Após um estudo mais aprofundado de Ruby, notei algumas melhorias possíveis na implementação, como:

Reescrever o método para adotar uma abordagem mais declarativa, retornando o valor ao invés de utilizar *puts*:

```ruby
def palindrome?(string)
  normalized_string = string.downcase.gsub(/\W/, '')
  normalized_string == normalized_string.reverse
end
```

Figure 2. Implementação Alternativa da Parte 1.1

A vantagem dessa abordagem é que ela segue o princípio funcional de retornar valores, minimizando efeitos colaterais. Além disso, métodos que retornam valores em vez de imprimi-los diretamente são geralmente mais flexíveis e reutilizáveis.

Para exibir o resultado ao usuário, a impressão deve ser feita no momento da chamada do método, como na figura 3. Dessa forma, o método palindrome? foca apenas em verificar e retornar o valor lógico, deixando a responsabilidade de exibição para o código que o invoca.

```ruby
# Casos de teste
puts palindrome?('A man, a plan, a canal -- Panama') # true
puts palindrome?("Madam, I'm Adam!") # true
puts palindrome?('Abracadabra') # false
```

Figure 3. Output da Implementação Alternativa da Parte 1.1

De modo geral, a lógica de palíndromos é simples de entender, especialmente por já ter implementado essa funcionalidade antes. O que me faltou foi um conhecimento

mais aprofundado da linguagem Ruby. No entanto, algumas pesquisas ajudaram a identificar as funções necessárias para alcançar o resultado desejado. Além disso, aproveitei minha experiência com outras linguagens, como Python e Java, para fazer uma análise comparativa e adaptar certos conceitos no Ruby — por exemplo, entender como usar o equivalente da função 'print' do Python em Ruby.

2.1.2. Método count_words

O objetivo deste exercício é criar um método que recebe uma string e retorna um hash onde as chaves são palavras da string e os valores representam a contagem de cada palavra.

Para isso, o método count_words primeiro normaliza a entrada (converte para minúsculas e extrai apenas palavras alfanuméricas) e, em seguida, percorre cada palavra para construir um hash de frequência. Cada vez que uma palavra é encontrada, o método incrementa o valor associado a essa palavra no hash. Ao final, o hash resultante contém todas as palavras como chaves e suas respectivas contagens como valores.

```ruby
def count_words(string)
  words = string.downcase.scan(/\w+/)
  frequency = Hash.new(0)
  words.each { |word| frequency[word] += 1 }
  puts frequency
end
```

Figure 4. Implementação da Parte 1.2

Abaixo está uma explicação das estruturas utilizadas e do algoritmo implementado, além do que já foi explicado na seção 2.1.1:

- **`scan(/\W+/)`**: Usa uma expressão regular *W+* para encontrar todas as palavras alfanuméricas na string e retorna um array com essas palavras.
- **`Hash.new(0)`**: Cria um hash onde o valor padrão é 0, permitindo incrementar a contagem de palavras sem verificar se a chave já existe
- **`words.each |word| frequency[word] += 1`**: É um loop que percorrecada palavra no array *words* e incrementa o valor correspondente no hash *frequency*, contando a frequência de cada palavra.

Após utilizar o modelo LLM 'ChatGPT', descobri duas novas maneiras de implementar este código.

A **primeira abordagem** adota um estilo mais funcional, removendo o uso de *puts*, e substituindo o escopo da função pelo método *each_with_object*. Esse método é semelhante ao *each*, mas passa um objeto extra em cada iteração, permitindo acumular resultados ao longo do loop [6]:

```ruby
def count_words(string)
  string.downcase.scan(/\w+/).each_with_object(Hash.new(0)) { |word, freq| freq[word] += 1 }
end
```

Figure 5. Implementação Alternativa da Parte 1.2

Essa versão é mais compacta e evita a necessidade de inicializar o hash fora do bloco de loop, mas pode ser menos legível para quem não está familiarizado com Ruby.

E, a **segunda abordagem**, utilizando o *inject* para acumular a contagem diretamente no hash. Esse método pode operar sobre um intervalo ou uma coleção e, ao ser chamado, passa cada elemento ao bloco e acumula o resultado sequencialmente [5]:

```ruby
def count_words(string)
  string.downcase.scan(/\w+/).inject(Hash.new(0)) { |freq, word| freq[word] += 1; freq }
end
```

Figure 6. Implementação Alternativa da Parte 1.2

Essa abordagem permite acumular o resultado diretamente, tornando o código mais conciso. No entanto, o uso de *inject* pode ser menos intuitivo para leitura e compreensão rápida, especialmente em comparação ao *each*.

A lógica de contagem de palavras é simples, mas é fundamental compreender o uso de *scan* em conjunto com a normalização de strings e a inicialização do hash com *Hash.new(0)*. A implementação do loop *each* é direta e, para desenvolvedores iniciantes em Ruby, pode ser mais fácil de entender em comparação com alternativas funcionais mencionadas. Ainda assim, é importante explorar essas alternativas para aproveitar a flexibilidade do Ruby na criação de soluções mais concisas e funcionais.

2.2. Parte 2: Pedra-Papel-Tesoura

Em um jogo de pedra-papel-tesoura, cada jogador escolhe jogar Pedra (R), Papel (P) ou Tesoura (S). As regras são: Pedra vence Tesoura, Tesoura vence Papel, e Papel vence Pedra.

Um jogo de pedra-papel-tesoura é codificado como uma lista, onde os elementos são listas de 2 elementos que codificam o nome de um jogador e a estratégia desse jogador.

2.2.1. Método rps_game_winner

O método *rps_game_winner* simula uma partida de Pedra-Papel-Tesoura entre dois jogadores e determina o vencedor com base nas seguintes regras:

1. **Número de jogadores**: Deve ser exatamente dois jogadores
2. **Estratégias válidas**: Apenas ‘R’, ‘P’ e ‘S’ são permitidas, sendo indiferente o uso de letras maiúsculas ou minúsculas (case-insensitive).
3. **Definição do vencedor**: Em caso de empate nas escolhas de estratégia, o primeiro jogador é declarado vencedor. Caso contrário, a vitória é determinada de acordo com as regras tradicionais do jogo.

Para isso, o método *rps_game_winner* começa com a validação do número de participantes e de suas estratégias, levantando as exceções, caso necessário. Após validar as entradas, o código compara as estratégias conforme as regras do jogo e determina o vencedor. Por fim, exibe o nome e a estratégia do vencedor.

```ruby
class WrongNumberOfPlayersError < StandardError ; end
class NoSuchStrategyError < StandardError ; end

def rps_game_winner(game)
  raise WrongNumberOfPlayersError unless game.length == 2

  valid_strategies = ["R", "P", "S"]

  player1, strategy1 = game[0][0], game[0][1].upcase
  player2, strategy2 = game[1][0], game[1][1].upcase

  raise NoSuchStrategyError unless valid_strategies.include?(strategy1) && valid_strategies.include?(strategy2)

  if strategy1 == strategy2
    winner = [player1, strategy1]
  elsif (strategy1 == "R" && strategy2 == "S") ||
    (strategy1 == "S" && strategy2 == "P") ||
    (strategy1 == "P" && strategy2 == "R")
    winner = [player1, strategy1]
  else
    winner = [player2, strategy2]
  end

  p winner
end
```

Figure 7. Implementação da Parte 2.1

Abaixo está uma explicação das estruturas utilizadas e do algoritmo implementado:

- **`WrongNumberOfPlayersError`**: Levantada se o número de jogadores não for igual a 2.
- **`NoSuchStrategyError`**: Levantada se uma das estratégias não for ‘R’, ‘P’ e ‘S’.
- **`if-else`**: É uma estrutura condicional para determinar o vencedor com base nas combinações de estratégias.
- **`p`**: Semelhante a *puts*, exibe o valor diretamente como ele é, incluindo aspas para strings e estruturas como arrays e hashes de forma mais detalhada. No caso deste método, que retorna um array contendo o nome do jogador vencedor e sua estratégia, *p* torna mais claro que a saída é uma estrutura de dados complexa.. [9]
- **`.upcase`**: Usado para converter todos os caracteres alfabéticos de uma string para letras maiúsculas.
- **`game[i][j]`**: Refere-se ao acesso de elementos em arrays multidimensionais. Neste contexto, game é um array contendo dois subarrays, cada um representando um jogador e sua estratégia.
- **`unless`**: É uma estrutura condicional em Ruby que funciona de forma oposta ao *if*. Ele executa um bloco de código somente se a condição for falsa. [8]

Após as validações, o método compara as estratégias. Se forem iguais, o primeiro jogador vence. Caso contrário, utiliza as regras do jogo para determinar o vencedor.

Ao analisar novamente o código, identifiquei duas melhorias potenciais que poderiam ser implementadas utilizando abordagens alternativas, além de uma terceira sugestão obtida com o auxílio do modelo LLM 'ChatGPT'.

A **primeira alternativa** consiste em simplificar o bloco de decisão removendo o *elsif* e reorganizando a lógica condicional. Isso também permite que o método retorne diretamente o vencedor, eliminando a necessidade de usar *p* dentro do escopo da função:

```
if (strategy1 == strategy2) ||
  (strategy1 == "R" && strategy2 == "S") ||
  (strategy1 == "S" && strategy2 == "P") ||
  (strategy1 == "P" && strategy2 == "R")
  winner = [player1, strategy1]
else
  winner = [player2, strategy2]
end

winner
```

Figure 8. Melhoria na Implementação da Parte 2.1

A **segunda alternativa** consiste em utilizar o método *case*, inspirado na estrutura de decisão *switch* da linguagem Java:

```
winner = case [strategy1, strategy2]
         when ["R", "S"], ["S", "P"], ["P", "R"], [strategy1, strategy1]
           [player1, strategy1]
         else
           [player2, strategy2]
         end

winner
```

Figure 9. Implementação Alternativa da Parte 2.1

Essa abordagem organiza a lógica de comparação de maneira mais clara e legível, ao alinhar as condições de vitória em um único bloco *case*. Ela também simplifica o fluxo, já que cada combinação de estratégias é avaliada em sequência, e o bloco *else* captura qualquer outra combinação.

Já a **terceira alternativa** utiliza o paradigma funcional com um *Hash* de resultados, mapeando diretamente as combinações vencedoras:

```ruby
def rps_game_winner(game)
  raise WrongNumberOfPlayersError unless game.length == 2

  valid_strategies = ["R", "P", "S"]
  outcomes = {
    "R" => { "R" => 0, "S" => 0, "P" => 1 },
    "S" => { "R" => 1, "S" => 0, "P" => 0 },
    "P" => { "R" => 0, "S" => 1, "P" => 0 }
  }

  player1, strategy1 = game[0][0], game[0][1].upcase
  player2, strategy2 = game[1][0], game[1][1].upcase

  raise NoSuchStrategyError unless valid_strategies.include?(strategy1) && valid_strategies.include?(strategy2)

  winner = game[outcomes[strategy1][strategy2]]

  p winner
end
```

Figure 10. Implementação Alternativa da Parte 2.1

Dessa forma, o hash 'outcomes' armazena todas as combinações possíveis de estratégias e o índice do jogador vencedor (0 para o primeiro jogador, 1 para o segundo). A lógica de comparação é substituída pela consulta ao hash, eliminando a necessidade de múltiplos if ou case. O índice do jogador vencedor é usado para acessar diretamente o jogador correspondente no array *game*.

Essa abordagem encapsula toda a lógica de decisão dentro do hash, o que torna o código mais compacto e facilita a manutenção, permitindo a inclusão de novos casos ou alterações sem modificar a estrutura principal do método. No entanto, essa solução apresenta algumas desvantagens: a lógica embutida no hash pode ser menos intuitiva para quem não conhece o contexto, e a criação e manutenção de um hash detalhado podem ser excessivas para um problema simples.

2.2.2. Método rps_tournament_winner

O objetivo deste exercício é simular um torneio de Pedra-Papel-Tesoura estruturado como uma árvore de jogos, onde cada nó representa um jogo, e os vencedores de cada rodada avançam até que o campeão seja determinado.

O método *rps_tournament_winner* funciona de forma recursiva para percorrer a árvore do torneio:

- Se o elemento atual é um jogo (ou seja, um array contendo dois jogadores), chama o método rps_game_winner (2.2.1) para determinar o vencedor.
- Se o elemento atual contém mais jogos (ou seja, é um array aninhado), divide o torneio em duas partes:
  - **winner1**: Determinado recursivamente para a primeira metade do torneio
  - **winner2**: Determinado recursivamente para a segunda metade do torneio.
- Os vencedores das duas metades disputam uma partida final, cujo vencedor é retornado.

```ruby
def rps_tournament_winner(tournament)
  if tournament[0][0].is_a?(String)
    return rps_game_winner(tournament)
  end

  winner1 = rps_tournament_winner(tournament[0])
  winner2 = rps_tournament_winner(tournament[1])

  rps_game_winner([winner1, winner2])
end
```

Figure 11. Implementação da Parte 2.2

Abaixo está uma explicação das estruturas utilizadas e do algoritmo implementado:

- **`.is_a?(String)`**: Em Ruby, o método *is_a?* é usado para verificar se um objeto pertence a uma determinada classe ou a uma instância de uma classe [11]. Nesse caso, o objetivo é verificar se o elemento atual da estrutura do torneio é uma string, ou seja, se representa o nome de um jogador.
- **`Recursividade`**: A lógica recursiva permite que o método processe torneios de qualquer tamanho ou profundidade sem a necessidade de loops complexos.

Nesta atividade, não identifiquei uma alternativa que pudesse melhorar significativamente o código em algum aspecto. Para mim, a implementação já abrange o essencial: **generalização**, pois funciona para torneios de qualquer tamanho, e **elegância**, uma vez que a recursividade simplifica a lógica e reduz a complexidade do código. Embora a profundidade da recursão possa gerar problemas de desempenho em torneios extremamente grandes, a solução apresentada é perfeitamente adequada para os casos analisados aqui.

2.3. Parte 3: Anagramas

O objetivo desta atividade é agrupar palavras que são anagramas em um array de grupos. Um anagrama é uma palavra ou frase formada ao reorganizar as letras de outra, ignorando diferenças de maiúsculas e minúsculas.

O método *combine_anagrams* utiliza um Hash para agrupar palavras que compartilham a mesma sequência de letras, independente da ordem ou capitalização:

```ruby
def combine_anagrams(words)
  anagrams = Hash.new
  words.each do |word|
    key = word.downcase.chars.sort.join
    if anagrams.has_key?(key)
      anagrams[key].push(word)
    else
      anagrams[key] = [word]
    end
  end
  p anagrams.values
end
```

Figure 12. Implementação da Parte 3

Abaixo está uma explicação das estruturas utilizadas e do algoritmo implementado:

- **`.word.downcase.chars.sort.join`**: É uma cadeia de métodos que transforma uma palavra em sua forma canônica para agrupar anagramas.
  - **`downcase`**: Como explicado no 2.1.1, converte todos os caracteres da string para letras minúsculas, evitando diferenças de capitalização.
  - **`chars`**: Divide a string em um array de caracteres individuais, permitindo manipular e ordenar cada caractere separadamente.
  - **`sort`**: Ordena os caracteres em ordem alfabética, garantindo que os anagramas gerem a mesma sequência de caracteres, independentemente da ordem original.
  - **`join`**: Junta os caracteres ordenados de volta em uma única string.
- **`.has_key?`**: É um método utilizado para verificar se um Hash contém uma determinada chave. Nesse caso, se a chave já existir, a palavra é adicionada ao array de valores correspondentes, se não, um novo array é criado com a palavra como seu primeiro elemento.

Utilizando o método *group_by*, é possível reduzir significativamente o código, tornando-o mais conciso e legível.

```ruby
def combine_anagrams(words)
  words.group_by { |word| word.downcase.chars.sort.join }.values
end
```

Figure 13. Implementação Alternativa da Parte 3

O método *group_by* é um método 'built_in' do Ruby, que agrupa os elementos de uma coleção com base no critério fornecido em um bloco. Ele retorna um Hash,

onde as chaves são os resultados do bloco e os valores são arrays contendo os elementos correspondentes. [3]

Para melhor compreensão, a figura 14 demonstra o caso de teste e a saída retornada pelo código apresentado.

```
p combine_anagrams( words ['cars', 'for', 'potatoes', 'racs', 'four', 'scar', 'creams', 'scream'])
# => [["cars", "racs", "scar"], ["for"], ["potatoes"], ["four"], ["creams", "scream"]]
```

Figure 14. Output da Implementação da Parte 3

2.4. Parte 4: Programação Orientada a Objetos Básica

2.4.1. Classe Dessert

O objetivo desta atividade é criar uma classe chamada *Dessert* com as seguintes características:

- **Atributos**:
  - **Nome**: Nome da sobremesa.
  - **Calories**: Número de calorias da sobremesa.
- **Métodos:**
  - **healthy?**: Retorna *true* se a sobremesa tiver menos de 200 calorias.
  - **delicious?**: Retorna *true* para todas as sobremesas.

```ruby
class Dessert
  def initialize(name, calories) → nil
    @name = name
    @calories = calories
  end

  def healthy?
    @calories < 200
  end

  def delicious?
    true
  end
end
```

Figure 15. Implementação da Parte 4.1

Abaixo está uma explicação das estruturas utilizadas e do algoritmo implementado:

- **`initialize`**: Método construtor que inicializa os atributos *@name* e *@calories* ao criar um objeto da classe *Dessert*.

- **`@name` e `@calories`**: São atributos de instância, identificados pelo uso do símbolo @. O @ indica que a variável tem escopo limitado à instância da classe, ou seja, pode ser acessada e manipulada apenas dentro dos métodos da própria classe em que foi definida. [7]

Nesta atividade, também não identifiquei uma alternativa que trouxesse melhorias significativas, visto que a implementação é simples, exigindo pouco esforço de compreensão e código. O método atende perfeitamente aos requisitos propostos.

O comportamento esperado pode ser verificado com os seguintes exemplos:

```
dessert = Dessert.new( name "Bolo",  calories 300)
puts dessert.healthy? # false
puts dessert.delicious? # true

dessert = Dessert.new( name "Salada de Frutas",  calories 150)
puts dessert.healthy? # true
puts dessert.delicious? # true
```

Figure 16. Output da Implementação da Parte 4.1

Esses resultados confirmam que o método *healthy?* avalia corretamente se a sobremesa é saudável, enquanto o método *delicious?* retorna sempre true, como especificado.

2.4.2. Classe JellyBean Extendendo Dessert

Nesta atividade, o objetivo é criar uma classe chamada *JellyBean* que herda da classe *Dessert* e implementa um comportamento específico para o atributo *flavor*. A principal modificação é no método *delicious?*, que deve retornar false se o sabor for "black licorice", mas continuar retornando true para outros sabores e sobremesas não relacionadas a *JellyBean*.

```ruby
class JellyBean < Dessert
  def initialize(name, calories, flavor) → nil
    super(name, calories)
    @flavor = flavor
  end

  def flavor
    @flavor
  end

  def flavor=(flavor)
    @flavor = flavor
  end

  def delicious?
    @flavor == "black licorice" ? false : super
  end
end
```

Figure 17. Implementação da Parte 4.2

Abaixo está uma explicação das estruturas utilizadas e do algoritmo implementado:

- **`Herança (< Dessert)`**: JellyBean herda os métodos e atributos da classe Dessert, reutilizando a lógica para métodos como *healthy?*.
- **`super`**: Utilizado no método *initialize* para chamar o construtor da class *Dessert*, inicializando os atributos @*name* e @*calories*.
- **`Getter e Setter (flavor e flavor=)`**: Implementados para acessar e modificar o atributo @*flavor*, garantindo encapsulamento e flexibilidade.
- **`Sobrescrita de delicious?`**: Modifica o comportamento apenas para instâncias de JellyBean com sabor "black licorice", retornando. Para outros sabores, repassa a lógica para o método *delicious?* da superclasse usando super.

Em relação à alternativas de implementação, o uso de *attr_accessor* pode simplificar a definição dos getters e setters para o atributo 'flavor'. A principal vantagem é a redução na quantidade de código, automatizando a criação desses métodos de forma eficiente:

```ruby
class JellyBean < Dessert
attr_accessor :flavor

  def initialize(name, calories, flavor) → nil
    super(name, calories)
    @flavor = flavor
    end

  def delicious?
    @flavor == "black licorice" ? false : super
  end
end
```

Figure 18. Implementação Alternativa da Parte 4.2

Em Ruby, *attr_accessor* é um método que facilita a criação de getters e setters para variáveis de instância. Ele é amplamente utilizado para definir propriedades de instância em classes de maneira concisa e legível.

Com *attr_accessor*, elimina-se a necessidade de implementar manualmente os métodos para acessar (getter) e modificar (setter) o valor de uma variável de instância, tornando o código mais enxuto e de fácil manutenção. [1]

2.5. Parte 5: Programação Orientada a Objetos Avançada – Metaprogramação, Classes Abertas e Duck Typing

Nesta atividade, o objetivo é implementar o método *attr_accessor_with_history*, que estende a funcionalidade do *attr_accessor* padrão do Ruby para acompanhar o histórico de valores atribuídos a um atributo específico. Esse método mantém um registro de todas as alterações feitas no atributo, incluindo seu valor inicial nil.

```ruby
class Class
  def attr_accessor_with_history(attr_name)
    attr_name = attr_name.to_s
    attr_reader attr_name
    attr_reader "#{attr_name}_history"
    class_eval %Q{
      def #{attr_name}=(value)
        @#{attr_name}_history ||= [nil]
        @#{attr_name}_history << value
        @#{attr_name} = value
      end
    }
  end
end
```

Figure 19. Implementaçãoda Parte 5

Abaixo está uma explicação das estruturas utilizadas e do algoritmo implementado:

- **`att_reader`**: É utilizado para criar os getters para o atributo *attr_name* e seu histórico *#attr_name_history*.
- **`class_eval`**: Gera dinamicamente o método setter #attr_name= dentro da classe que chamou *attr_accessor_with_history*, permitindo a inserção de código no contexto da classe durante a execução.
- **`#attr_name_history`**: O setter inicializa o histórico como um array contendo *nil* como valor inicial. Cada novo valor atribuído ao atributo é adicionado ao histórico antes de atualizar o valor real do atributo.
- **`Interpolação de Strings`**: O # é usado para interpolação de strings em Ruby, permitindo que o valor de uma variável ou expressão Ruby seja inserido dentro de uma string. [2]

Não identifiquei uma alternativa que simplificasse o código de maneira significativa. Embora soluções como o uso do método *define_method* em vez de *class_eval* (figura 20) tenham sido consideradas, elas não apresentaram vantagens claras em relação à implementação atual.

```ruby
class Class
  def attr_accessor_with_history(attr_name)
    attr_name = attr_name.to_s
    attr_reader attr_name
    attr_reader "#{attr_name}_history"

    define_method("#{attr_name}=") do |value| [self: instance]
      instance_variable_set("@#{attr_name}_history", (instance_variable_get("@#{attr_name}_history") || [nil]))
      instance_variable_get("@#{attr_name}_history") << value
      instance_variable_set("@#{attr_name}", value)
    end
  end
end
```

Figure 20. Implementação Alternativa da Parte 5

Nesta parte da atividade, a metaprogramação se mostrou desafiadora, mas o uso do *attr_accessor_with_history* ilustrou de forma prática como manipular dinamicamente o comportamento de classes. Embora a implementação com *class_eval* possa parecer mais complexa inicialmente, a lógica torna-se bastante clara uma vez compreendida, destacando a flexibilidade e o poder da metaprogramação em Ruby.

2.6. Parte 6: Programação Orientada a Objetos Avançada – Metaprogramação, Classes Abertas e Duck Typing, continuação

2.6.1. Extensão de Conversão de Moedas

O objetivo desta atividade foi expandir o exemplo de conversão de moedas para permitir operações entre diversas moedas, utilizando Ruby e metaprogramação. Essa funcionalidade permite converter valores com sintaxes como **'5.dollars.in(:euros)'**.

O autor fornece uma classe-base, podendo ser encontrado neste link.

O código implementa essa funcionalidade ao modificar a classe Numeric (figura 21) e introduzir uma classe Currency (figura 22):

```ruby
class Numeric
  @@currencies = {
    'yen' => 0.013, 'euro' => 1.292, 'rupee' => 0.019, 'dollar' => 1
  }

  def method_missing( Symbol method_id, * untyped args, & (*untyped, **untyped) -> untyped block) -> untyped
    singular_currency = method_id.to_s.gsub(/s$/, '')

    if @@currencies.has_key?(singular_currency)
      Currency.new(self * @@currencies[singular_currency])
    else
      super
    end
  end
end
```

Figure 21. Implementação da Parte 6.1 - Classe Numeric

A classe Numeric ganha um método *method_missing*, que captura qualquer

chamada de método inexistente (como dollars ou euros) e verifica se corresponde a uma moeda válida definida em @@*currencies*. Se a moeda for válida, cria um objeto Currency com o valor convertido para dólares.

```ruby
class Currency
  def initialize(amount_in_dollars) → nil
    @amount_in_dollars = amount_in_dollars
  end

  def in(target_currency)
    singular_currency = target_currency.to_s.gsub(/s$/, '')

    if Numeric.class_variable_get(:@@currencies).has_key?(singular_currency)
      @amount_in_dollars / Numeric.class_variable_get(:@@currencies)[singular_currency]
    else
      raise "Currency not supported"
    end
  end
end
```

Figure 22. Implementação da Parte 6.1 - Classe Currency

A classe Currency, por sua vez, oferece o método *in(target_currency)*, que converte o valor em dólares para a moeda de destino.

Abaixo está uma explicação das estruturas utilizadas e do algoritmo implementado:

1. **`@@currencies`**: Um hash que define as taxas de câmbio relativas ao dólar. Uma variável @@ significa que é uma variável de classe, ou seja, são variáveis que armazenam informações sobre uma classe. [4]
2. **`Classe Currency`**: Responsável por armazenar o valor em dólares e converter para outras moedas.
3. **`in(target_currency)`**: Método que converte o valor em dólares para a moeda especificada pelo usuário
4. **`.to_s.gsub(\s$/, '')`**: Converte o valor *target_currency* para uma string, em seguida, usa o método *gsub* para realizar substituições em strings. Nesse caso, vai buscar um ”s” apenas se ele estiver no final da string. Caso encontre, ele o substitui por uma string vazia (”), removendo-o. Dessa forma, é possível ignorar o plural ao referenciar o nome da moeda.

Essa é a abordagem Imperativa, mas também é possível utilizar a abordagem Funcional. Na abordagem Funcional, o uso de estado compartilhado é evitado, e os métodos retornam novos valores em vez de modificar o objeto *Currency* ou valores dentro da classe *Numeric*. Cada chamada gera uma nova instância de conversão ou uma representação temporária do valor convertido.

```ruby
class Currency
  def self.convert(amount, from_currency, to_currency)
    currencies = { 'yen' => 0.013, 'euro' => 1.292, 'rupee' => 0.019, 'dollar' => 1 }
    amount_in_dollars = amount * currencies[from_currency]
    amount_in_dollars / currencies[to_currency]
  end
end

puts Currency.convert( amount 5,  from_currency :dollars,  to_currency :euros)
```

Figure 23. Implementação Alternativa da Parte 6.1 - Classe Currency

Essa abordagem evita a criação de objetos desnecessários, simplificando a implementação e minimizando o uso de memória. Contudo, essa vantagem tem um custo: a fluidez e a legibilidade são reduzidas, pois a implementação não apresenta uma sintaxe como **'5.dollars.in(:euros)'**.

2.6.2. Adaptando a solução 2.1.1

A atividade solicita a adaptação da solução para que o método *palindrome?* apresentado na seção 2.1.1 possa ser chamado diretamente em um objeto do tipo String, como em *"foo".palindrome?*

```ruby
class String
  def palindrome?
    normalized_string = self.downcase.gsub(/\W/, '')
    normalized_string == normalized_string.reverse
  end
end

# Casos de teste
puts "A man, a plan, a canal -- Panama".palindrome?
puts "Madam, I'm Adam!".palindrome?
puts "Abracadabra".palindrome?
```

Figure 24. Implementação da Parte 6.2

Abaixo está uma explicação das estruturas utilizadas e do algoritmo implementado:

1. **`Definição da Classe String`**: Ao abrir a classe String, foi adicionado o método *palindrome?* diretamente, permitindo que todas as instâncias de String utilizem o método. Esse recurso de abrir classes para adicionar métodos é um recurso poderoso e natural em Ruby, permitindo que a linguagem seja mais expressiva e que métodos personalizados possam ser usados diretamente em objetos de tipos primitivos. [10]

2. **`Implementação do Método palindrome?`**: O método é o mesmo apresentado na seção 2.1.1.

Esta solução é direta e eficaz, pois utiliza a capacidade de metaprogramação do Ruby para adicionar métodos a classes existentes, permitindo uma sintaxe mais natural

2.6.3. Adaptando a solução 2.1.1 para Enumerável

Esta atividade adapta a solução do método *palindrome?* para que funcione com qualquer coleção que inclua o módulo *Enumerable*. Agora, é possível verificar se uma lista é um palíndromo ao chamar o método diretamente em objetos Enumerable, como arrays.

```ruby
module Enumerable [Elem]
  def palindrome?
    self.to_a == self.to_a.reverse
  end
end

puts [1,2,3,2,1].palindrome?
puts [1,2,3,4,5].palindrome?
puts [1,2,3,2,1,2].palindrome?
```

Figure 25. Implementação da Parte 6.3

Abaixo está uma explicação das estruturas utilizadas e do algoritmo implementado:

1. **`Definição do Módulo Enumerable`**: O código abre o módulo *Enumerable* e adiciona o método *palindrome?*. Dessa forma, todos os objetos que incluem Enumerable (como arrays e ranges) passam a ter acesso a esse método.
2. **`Implementação do Método palindrome?`**: converte a coleção para um array, independentemente do tipo original (como Range) e, em seguida, o código verifica se o array é igual à sua versão invertida *(self.to_a.reverse)*. Se forem iguais, a coleção é um palíndromo, e o método retorna true. Caso contrário, retorna false.

É importante observar que o método não foi projetado para funcionar com hashes, mesmo que eles incluam Enumerable. Ao converter um hash para array com *to_a*, o formato resultante pode ser inesperado e, portanto, não faz sentido verificar palíndromos nesse contexto. Apesar disso, a implementação é curta e eficiente, usando apenas três linhas de código, e aproveita a capacidade de metaprogramação do Ruby para estender funcionalidades de módulos e torná-los mais expressivos.

3. Conclusão e feedback

Compreender e desenvolver os exercícios exigiu uma imersão nas particularidades do Ruby, como o uso de blocos e o suporte à programação funcional. As atividades variaram

em complexidade, com as partes finais (a partir da Parte 5) apresentando maior desafio devido ao uso avançado de metaprogramação. Estimei cerca de 30 minutos para a compreensão e desenvolvimento de cada parte, enquanto a análise criteriosa e a comparação de alternativas demandaram aproximadamente 1 hora adicional por exercício, exigindo também pesquisa aprofundada sobre os métodos e funcionalidades oferecidos pelo Ruby.

A experiência de desenvolvimento foi enriquecedora, especialmente pela oportunidade de explorar múltiplos paradigmas, como o funcional e o orientado a objetos, ambos suportados pelo Ruby. A IDE RubyMine foi uma ferramenta essencial para agilizar o processo, proporcionando recursos robustos para depuração e execução do código. A análise e implementação de alternativas ajudaram a consolidar o entendimento das vantagens e limitações de diferentes abordagens, além de realçar a flexibilidade do Ruby para construir soluções concisas e eficazes.

Para melhorar a clareza e abrangência do conteúdo escrito, utilizei o modelo de linguagem (LLM) ChatGPT. Ele auxiliou na redação e adaptação de diferentes seções do relatório, oferecendo sugestões de abordagens para algumas atividades, além das que desenvolvi autonomamente. Antes de iniciar as interações, compartilhei com o modelo algumas informações prévias para que as respostas se alinhassem ao que já compreendia e esperava do conteúdo, tornando as discussões mais direcionadas e úteis.

References

[1] Dio. Attr_accessor - estudando ruby. https://www.dio.me/articles/attr_accessor-estudando-ruby. Accessed: 2024-11-06.

[2] Ruby for Beginners. String interpolation. https://ruby-for-beginners.rubymonstas.org/bonus/string_interpolation.html. Accessed: 2024-11-06.

[3] Geeks for Geeks. Ruby — enumerable group_by() function. https://www.geeksforgeeks.org/ruby-enumerable-group_by-function/. Accessed: 2024-11-05.

[4] Medium. Class variable, class methods, and self in ruby. https://medium.com/geekculture/class-variable-class-methods-and-self-in-ruby-388706a4c491. Accessed: 2024-11-07.

[5] Medium. Inject method: Explained. https://medium.com/@terrancekoar/inject-method-explained-ed531eff9af8. Accessed: 2024-11-05.

[6] Stack Overflow. .each_with_object ruby explanation? https://stackoverflow.com/questions/26634897/each-with-object-ruby-explanation. Accessed: 2024-11-05.

[7] Stack Overflow. O que significa o arroba (@) em variáveis no ruby? https://pt.stackoverflow.com/questions/321956/o-que-significa-o-arroba-em-variveis-no-ruby. Accessed: 2024-11-06.

[8] Stack Overflow. Para que serve o comando unless no ruby. https://pt.stackoverflow.com/questions/218143/

para-que-serve-o-comando-unless-no-ruby. Accessed: 2024-11-05.

[9] Stack Overflow. puts e p em ruby. https://pt.stackoverflow.com/questions/210705/puts-e-p-em-ruby. Accessed: 2024-11-05.

[10] Stack Overflow. Understanding ruby open classes. https://stackoverflow.com/questions/6745537/understanding-ruby-open-classes. Accessed: 2024-11-07.

[11] Scaler Topics. Ruby is_a? method. https://www.scaler.com/topics/ruby-is_a/. Accessed: 2024-11-05.