Orientação a Objetos com Ruby

Arthur de Moura Del Esposte - esposte@ime.usp.br

Aula 02 - Estruturas de Controle, Classes e Objetos

Arthur de Moura Del Esposte - esposte@ime.usp.br

Martin Fowler

“Qualquer um pode escrever código para o **computador** entender. **Bons programadores** escrevem código que **humanos** podem entender!” - *Refactoring: Improving the Design of Existing Code*

Agenda

- Estruturas de Controle
- Laços de Repetição
- Classes , Objetos e Métodos
- Herança e Composição
- Polimorfismo e *Duck Typing*

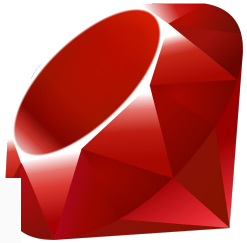
Condições

Estrutura Condicional

- **A estrutura básica é semelhante a outras linguagens**

```
3  if value == 'A'
4    puts "A"
5  elsif value == 'B'
6    puts "B"
7  else
8    puts "C"
9  end
```

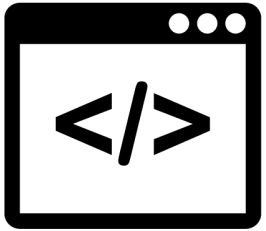
Estrutura Condicional Composta



- Exemplo com definição da nota final em uma disciplina que considera:
  - Presença
  - Pontuação

```
7  if presence >= 0.75
8    if score >= 9.0
9      grade = "A"
10   elsif score >= 7.0
11     grade = "B"
12   elsif score >= 5.0
13     grade = "C"
14   else
15     grade = "SR"
16   end
17 else
18   grade = "SR"
19 end
```

Exercício



Escreva um script Ruby que receba 5 números inteiros e imprima o maior deles

- **Entrada**: 5, 2, 23, 13, 18
- **Saída:** *“O maior número é o 23”*

- Dica:
  - *Pense em como usar as estruturas de dados e condicional já conhecidas*

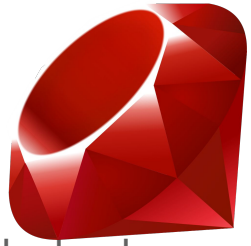
Operador ternário

- O Ruby também possui o operador ternário de condição:
  - <condição> **?** <retorno quando verdadeiro> **:** <retorno caso falso>
- O Ruby também possui o operador ternário de condição:

```ruby
a = true  ? 'a' : 'b' #=> "a"
b = false ? 'a' : 'b' #=> "b"
```

```ruby
if true
  a = 'a'
else
  a = 'b'
end
```

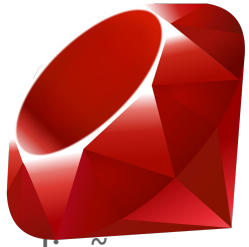
Unless



- A estrutura condicional **unless** é muito importante para a expressividade do código
- **unless** equivale ao **if negado**
  - **IF:** Se algo for verdadeiro, então execute esse código
  - **UNLESS:** A não ser que isso seja verdadeiro, execute esse código

```
23  my_array = []
24  unless my_array.empty?
25    puts "My array has at least one element"
26  end
```

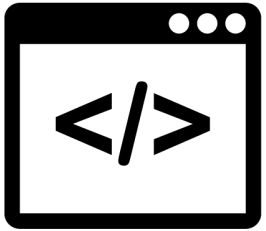
Condições com maior expressividade



Com o propósito de melhorar a leitura do código, o Ruby permite que condições sejam colocadas na mesma linha do comando que deverá ser executado!

```
29  exit unless "restaurant".include? "aura"
30
31  exit if my_array.count('a') == 5
```

Exercício



Escreva um script Ruby que receba 5 números inteiros e imprima somente os números pares

- **Entrada**: 5, 2, 23, 13, 18
- **Saída:** *“Os números pares são: 2, 18”*

- Obs:
  - Tente usar o **unless**

Laços de repetição

Relembrando iteradores em coleções...

- **Arrays** & **Hashes** já possuem métodos adequados para acessar seus elementos iterativamente

Array:

```ruby
["first", "middle", "last"].each { |element| puts element.capitalize }
```

Hash:

```ruby
my_hash = { "a" => 100, "b" => 200 }

my_hash.each {|key, value| puts "#{key} is #{value}" }
```

While

Possui a estrutura semelhante a de outras linguagens

```
10  i = 0
11  numbers = []
12
13  while i < 6
14    numbers.push(i)
15    i += 1
16  end
```

Until

- Executa um bloco de código até que a condição seja verdadeira
- Segue a mesma ideia de expressividade do **unless**

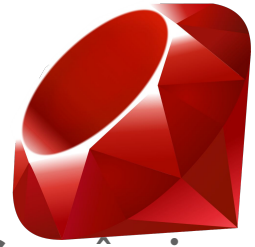
```
18  numbers = [1, 2, 3, 4, 5, 6]
19
20  until numbers.empty?
21    numbers.pop
22  end
23  puts numbers.size
```

For

- Executa um código uma vez para cada elemento na expressão

```
25  letters = ['a', 'b', 'c']
26  for letter in letters
27    puts letter
28  end
```

Ranges



- **Ranges** são estruturas de dados nativas do Ruby para representar **Sequências e Intervalos.** Podem ser usados em loops

```
3  elements = []
4
5  (0..5).each do |i|
6    puts "adding #{i} to the list."
7    elements.push(i)
8  end
```

Testes no IRB

Ranges

```
> puts (1..10).class

> (1..10).to_a

> ('a'..'z').to_a

# Range as Interval

> (1..10) === 5      # =>true

> (1..10) === 15     # => false

> ('a'..'j') === 'z'     # => false
```

Next & Break

- São utilizados para modificar a execução normal de um loop
- O **next** faz com que a execução atual seja interrompida, indo imediatamente para a próxima iteração do loop
- O **break** interrompe todas as execuções (atuais e futuras) de um laço, indo imediatamente para a primeira linha após o bloco de código do laço.

Next & Break

- Programa para calcular o valor de números ímpares ao quadrado:

```ruby
while true
  puts "Insert an odd number or 'q' to quit:"
  input = gets.chomp
  break if input == 'q'

  number = input.to_i
  if number.even?
    puts "Invalid input!"
    next
  end

  puts "The value of #{number}² is #{number**2}"
end

puts "Quiting..."
```

Quais dos códigos a seguir produzem a seguinte saída:

0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20,

1. Loop com Range

```ruby
(0..20).each do |i|
  next if i % 2 == 1
  print "#{i}, "
end
```

2. Loop com For

```ruby
for i in (0..20)
  puts "#{i}, " if i.even?
end
```

3. Loop com Until

```ruby
i = 0
until i > 20
  print "#{i}, " unless i.odd?
  i += 1
end
```

4. Loop com While

```ruby
i = 0
while true
  break if i >= 20
  print "#{i}, " if !i.odd?
  i += 1
end
```

Quais dos códigos a seguir produzem a seguinte saída:

0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20,

1. Loop com Range

```ruby
(0..20).each do |i|
  next if i % 2 == 1
  print "#{i}, "
end
```

2. Loop com For

```ruby
for i in (0..20)
  puts "#{i}, " if i.even?
end
```

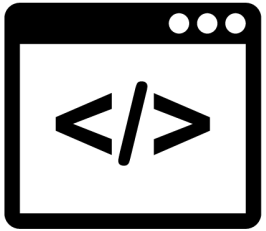
3. Loop com Until

```ruby
i = 0
until i > 20
  print "#{i}, " unless i.odd?
  i += 1
end
```

4. Loop com While

```ruby
i = 0
while true
  break if i >= 20
  print "#{i}, " if !i.odd?
  i += 1
end
```

Exercício

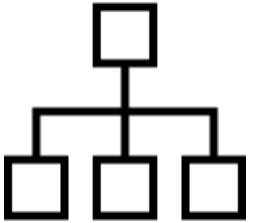


Escreva um script Ruby que receba o nome e a idade de várias pessoas. Ao receber as informações de uma pessoa, o programa deve perguntar se o usuário deseja inserir outra pessoa. Após o usuário ter informado todas as pessoas, imprima o nome da(s) pessoa(s) mais velha(s) e sua idade:

- **Entrada**:
  - Chico
  - 15
  - Joao
  - 56
  - Maria
  - 12
  - Larissa
  - 56
- **Saída:** *“Joao - Larissa - 56 anos”*

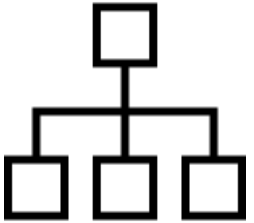
Orientação a Objetos: Classes, objetos e métodos

Linguagens OO modernas



- Objetos
- Atributos (propriedades), *getters* e *setters*
- Métodos
- Sobrecarga de operadores
- Interfaces
- Tipos primitivos e *Wrappers*
- ...

Em Ruby...



- **Tudo** é um objeto
- **Quase tudo** é uma chamada de método
- **a.b** significa: chame o método **b** no objeto **a**
  - **a** é o receptor para o qual você envia a chamada de método, assumindo que **a** irá responder à esta mensagem
  - Não significa que **b** é uma variável de instância de **a**
  - Não significa que **a** é uma estrutura de dados que tem **b** como um membro (como structs em C)

Exemplo: toda operação é uma chamada de método

```ruby
y = 3 + 5 # => 8
y = [1,2] + ["foo", :bar] # => [1,2,"foo",:bar]
y = "hello" + "world" # => "hello world"
```

- Não há **conversão** ou **promoção** de tipos de dados
- Todos são **métodos de instância** das classes **Numeric**, **Array**, **String** - e não operadores da linguagem!
- **Numeric#+**, **Array#+**, **String#+**, na notação utilizada em Ruby

Classes - Estrutura Básica

- Todo método retorna a última linha
- Logo, todo método retorna algum objeto, mesmo que seja o objeto **nil**
- A palavra return só precisa ser utilizada caso o retorno do método tenha que ser feito antes da última linha

```ruby
class MyClass
  def calculate(p1 = 0, p2 = 0, p3)
    return nil if p3 == 0

    (p1 + p2)/p3
  end
end
```

Classes - Construtor

- Uma classe básica com o método construtor.
- Atributos de instância possuem um @ no início do nome
- Os atributos passam a existir no momento em que são inicializados

```ruby
class Song
  def initialize
    @name = "One"
    @duration = 4.35
    @artist = "u2"
    @lyrics = "..."
  end
end
```

Crie um arquivo chamado **song.rb**, e inclua o código da classe **Song** no arquivo. Vamos fazer testes com o IRB.

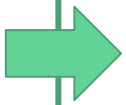
Testes no IRB

Classes - Parte 1

```
> require_relative 'song'

> song = Song.new

> puts song

> song.name # => NoMethodError
```

Classes - Acesso aos atributos

```ruby
def name
  @name
end

def name= name
  @name = name
end
```



```ruby
class Song
  attr_accessor :name, :duration, :artist, :lyrics

  def initialize
    @name = "One"
    @duration = 4.35
    @artist = "u2"
    @lyrics = "..."
  end
end
```

Testes no IRB

Classes - Parte 2

```
> require_relative 'song'

> song = Song.new

> puts song

> song.name

> song.duration = 3.0

> song
```

Ruby é Flexível

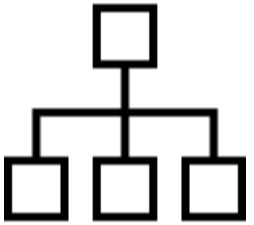
Testes no IRB

Classes - Parte 3 - Estendendo a Classe

```
> class Song
>   def to_s
>     puts "Name: #{@name}"
>   end
> end
> song.to_s
```

Orientação a Objetos: Herança e Composição

Herança



- Herança é um dos conceitos **mais importantes** em POO
- Uma classe em Ruby só pode herdar de uma única classe
- Tão simples quanto:

```
class KaraokeSong < Song
end
```

Icon made by Oliver from www.flaticon.com

Considerando o código ao lado, quais são as **duas** expressões que retornam *false*?

```ruby
class KaraokeSong < Song
end

karaoke_song = KaraokeSong.new
song = Song.new
```

1. `karaoke_song.is_a? KaraokeSong`
2. `karaoke_song.class == KaraokeSong`
3. `song.is_a? KaraokeSong`
4. `karaoke_song.is_a? Song`
5. `karaoke_song.class == Song`

Considerando o código ao lado, quais são as **duas** expressões que retornam *false*?

```ruby
class KaraokeSong < Song
end

karaoke_song = KaraokeSong.new
song = Song.new
```

1. `karaoke_song.is_a? KaraokeSong`
2. `karaoke_song.class == KaraokeSong`
3. ✓ `song.is_a? KaraokeSong`
4. `karaoke_song.is_a? Song`
5. ✓ `karaoke_song.class == Song`

Sobrescrita e Sobrecarga - Parte 1

- Em Ruby não há possibilidade de se ter dois métodos com mesmo nome, mas com parâmetros diferentes
- Portanto não existe **Sobrecarga** de método
- A última definição do método é a que prevalece, caracterizando uma **Sobrescrita** (*Overriding*)

Prevalece

```
3  class MyClass
4    def do_something(a, b, c)
5    end
6
7    def do_something(d)
8    end
9  end
```

Sobrescrita e Sobrecarga - Parte 2

- **Sobrescrita** é utilizada para **redefinição de métodos** da classe **Mãe** na classe **Filha**
- O código definido na classe **Mãe** pode ser invocado pela **Filha** através do método **super**

```ruby
class Animal
  def initialize(name, gender)
    @name = name
    @gender = gender
  end
end
```

```ruby
class Dog < Animal
  def initialize(name, gender, breed)
    super(name, gender)
    @breed = breed
  end
end
```

Visibilidade

- A visibilidade é definida para o objeto
- Um método **público** é acessível em qualquer lugar
- Um método **protegido** da classe A é acessível explicitamente pelos objetos de A e por seus filhos
- Um método **privado** da classe A só pode ser chamado pelo próprio objeto de forma não explícita (**self**)

```ruby
class Song
  def public_method
    puts "public method"
  end

  protected
  def protected_method
    puts "protected method"
  end

  private
  def private_method
    puts "private method"
  end
end
```

Mais informações sobre visibilidade

Visibilidade - Ruby vs Java

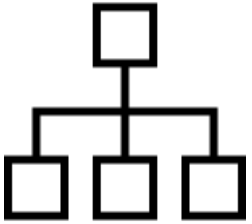
```ruby
class A
  def public_method(a)
    a.private_method
  end

  private
  def private_method(d)
    puts "Privte method"
  end
end

xpto1 = A.new
xpto2 = A.new
xpto1.public_method(xpto2)
```

```java
class A {
  public void public_method(A x){
    x.x_private(x);
  }

  private void private_method(A x) {
    System.out.println("Private method");
  }
}

class Teste{
  public static void main(String[] args){
    A xpto1 = new A();
    A xpto2 = new A();
    xpto1.private_method(xpto1);
  }
}
```

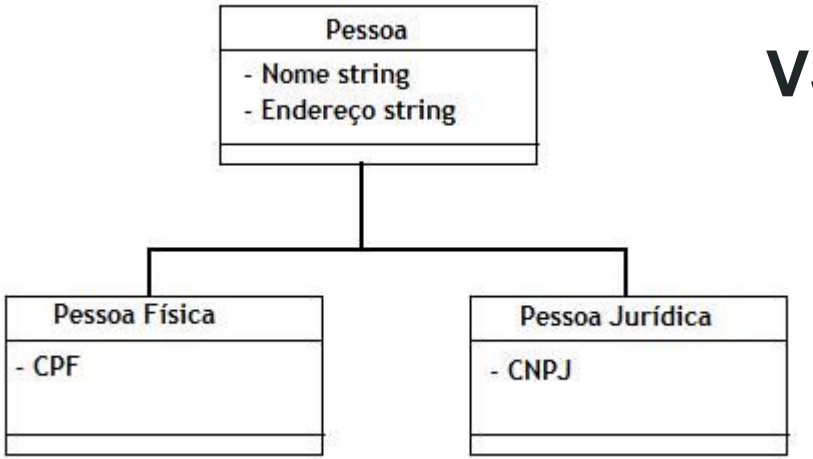
Atributos

- Atributos são **sempre privados**, somente acessíveis pelos respectivos métodos
- É importante entender que os atributos não são herdados
- Atributos são criados quando valores são atribuídos a eles
- Porém, como muitas vezes deixamos de sobrescrever alguns métodos ou realizamos chamadas **super**, acaba-se tendo atribuições dos atributos que são definidos na classe **Mãe**

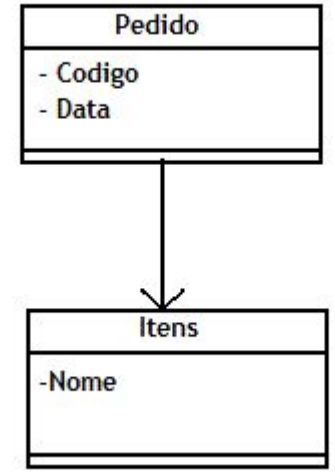
Composição



- Heranças devem ser usadas com cuidado!
- Composição é uma forma de estender um classe e delegar o trabalho para o objeto desta classe
- Enquanto a herança define uma relação **“É um”**, a composição define uma relação **“Tem um”**

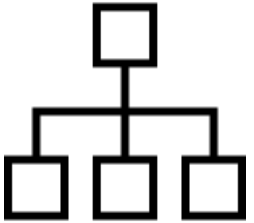


VS



Fonte original dos diagramas

Composição



- Composição apoia o **Baixo Acoplamento** e **Alta Coesão**

E se tivermos um carro com Alarme e Ar Condicionado?

Alguma ideia de como fazer uma classe Car com componentes em Ruby, em vez de usar herança?

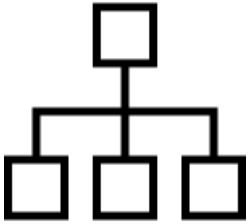
Alguma ideia de como fazer uma classe Car com componentes em Ruby, em vez de usar herança?

```ruby
class Car
  def initialize
    @price = 1.00
    @components = []
  end

  def add_component(component)
    @components << component
  end

  def price
    @price + (0.2 * @components.count)
  end
end
```

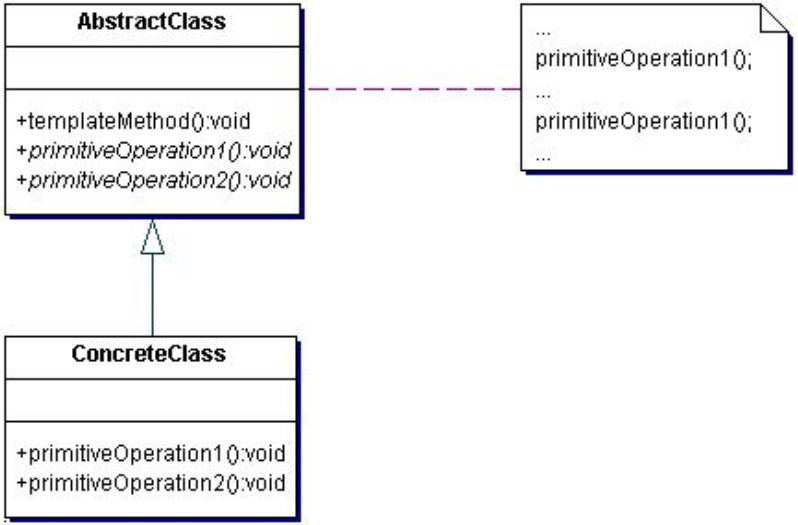
```ruby
class Alarm
  def price
    0.2
  end
end

class AirConditioning
  def price
    0.3
  end
end
```

Orientação a Objetos: Polimorfismo

Classes Abstratas - Parte 1



- Não há o conceito de **Classes Abstratas** em Ruby
- Porém, podemos alcançar os mesmos objetivos de design (como **Polimorfismo**) de outras formas
- Veja o exemplo com o padrão de projeto **Template Method**



Classes Abstratas - Parte 2

```ruby
class Worker
  def work
    puts 'Start working'
    design_product
    prepare_material
    build
    puts "The #{product} is ready"
  end

  def design_product
    puts "Designing a #{product}"
  end
end
```

```ruby
class CivilEngineer < Worker
  def prepare_material
    puts "Define materials for the floor, walls and ceiling"
  end

  def build
    puts "Putting the bricks..."
  end

  def product
    "House"
  end
end
```

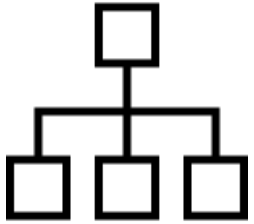
Classes Abstratas - Parte 3

- Outra estratégia é lançar uma **Exceção** no método que deve ser **obrigatoriamente sobrescrito**

```ruby
class Worker
  def work
    raise NotImplementedError.new("#{self.class.name}#work is an abstract method.")
  end
end
```

```ruby
class SoftwareEngineer < Worker
end
```

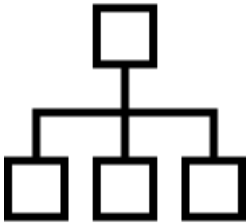
Polimorfismo e Duck Typing - Parte 1



- Polimorfismo é um conceito fundamental em POO para obter comportamentos diferentes entre objetos de tipos variados usando-se a mesma interface
- Em Ruby, duas classes diferentes não precisam compartilhar tipos para obtermos polimorfismo sobre seus métodos
- **Duck Typing** é uma forma de determinar a semântica válida de um objeto baseado no que ele pode fazer (seus métodos e propriedades), em vez de seu tipo (sua herança ou implementação de interface)

“Se anda como um pato e faz barulho como um pato, então deve ser um pato”

Polimorfismo e Duck Typing - Parte 2

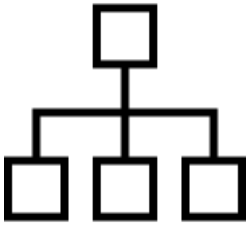


```ruby
class XmlParser
  def parse
    puts 'An instance of the XmlParser class received the parse message'
  end
end
```

```ruby
class JsonParser
  def parse
    puts 'An instance of the JsonParser class received the parse message'
  end
end
```

```ruby
class GenericParser
  def parse(parser)
    parser.parse
  end
end
```

Polimorfismo e Duck Typing - Parte 2



```ruby
class XmlParser
  def parse
    puts 'An instance of the XmlParser class received the parse message'
  end
end
```

```ruby
class JsonParser
  def parse
    puts 'An instance of the JsonParser class received the parse message'
  end
end
```

```ruby
class GenericParser
  def parse(parser)
    parser.parse
  end
end
```

Duck Typing =D

Revisão!

O que já vimos!

- Estruturas de Controle
- Laços de Repetição
- Classes , Objetos e Métodos
- Herança e Composição
- Polimorfismo e *Duck Typing*

Atividades Sugeridas!

Resolver os seguintes desafios - Estruturas de Controle

1. Faça um programa que mostre os números entre 1.000 e 2.000 que, quando divididos por 11, produzam resto 5
2. Faça um programa que receba a idade de 10 pessoas e que calcule e mostre a quantidade de pessoas com idade maior ou igual a 18
3. Faça um programa que receba um número e mostre a tabuada dele. O programa deve continuar recebendo um número até que o usuário escreva ‘sair’ ou ‘quit’
4. Faça um programa que receba o peso e idade de 7 pessoas e calcule:
    - A quantidade de pessoas com idade superior a 65 anos
    - A média das idades das sete pessoas

Resolver os seguintes desafios - Orientação a Objetos - Parte 1

- Crie uma classe chamada **Fraction** para representar uma fração matemática. Ela deve ter os seguintes métodos:
  - **Fraction#initalize** - Construtor para receber o numerador e denominador
  - **Fraction#to_f** - Método para conversão para Float
  - **Fraction#to_s** - Método que retorna um string com a fração
  - **Fraction#* -** Método que recebe outro objeto de Fração ou número inteiro e retorna uma novo objeto do tipo Fraction com o resultado da multiplicação
  - Métodos de acesso às variáveis de numerador e denominador

Resolver os seguintes desafios - Orientação a Objetos - Parte 2

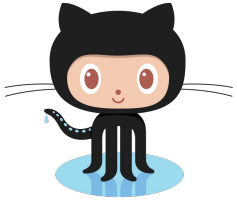
- Desenvolva um **Jogo da Forca** onde o usuário possui 7 chances para acertar qual as letras que compõem uma palavra em segredos. Caso o jogador erre mais de 7 vezes, ele perde. Caso complete a palavra, o jogador ganha! Confira as regras em: https://pt.wikipedia.org/wiki/Jogo_da_forca
- Melhore o **Jogo da Forca** para perguntar se o usuário deseja jogar novamente ao fim de cada partida. Desse modo, registre as vitórias e derrotas do usuário e apresente um resumo no final quando o usuário não desejar mais jogar.
- Mantenha uma lista de palavras em um arquivo words.txt onde seja fácil adicionar e remover novas palavras para o jogo. Leia esse arquivo para usar essas palavras dentro do jogo.

Estudar

- Estudar e dominar as estruturas de controle do Ruby
- Revise os conceitos de Orientação a Objetos
- Revise os princípios de design de Orientação a Objetos
- Revise os diagramas de classe UML

Contato

https://gitlab.com/arthurmde



https://github.com/arthurmde

http://bit.ly/2jvND12

http://bit.ly/2j0llo9

Centro de Competência em Software Livre - CCSL

esposte@ime.usp.br

Obrigado!