Orientação a Objetos com Ruby

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

Aula 05 - Bibliotecas e Metaprogramação

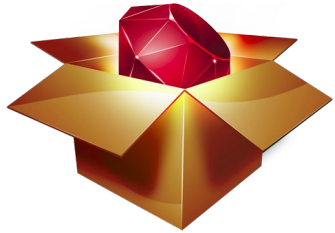
Arthur de Moura Del Esposte - esposte@ime.usp.br

Agenda

- Ruby Gems
- Metaprogramação
- Passos futuros

Ruby Gems

Gems

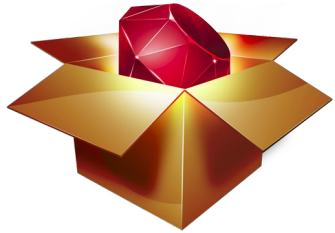


- **Gems são pacotes de software ruby**
- Uma Gem é uma biblioteca ou um conjunto de arquivos reutilizáveis, etiquetadas em um nome e uma versão
- **RubyGems** é um sistema de gerenciamento de pacotes Ruby que facilita a criação, compartilhamento e instalação de bibliotecas
- A instalação do Ruby já vem com o gerenciador de pacotes Ruby que pode ser acessado via linha de comando:

```
$ gem  -v

$ gem  -h
```

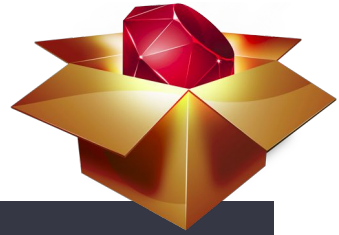
Mais detalhes na Documentação Oficial e nesse Blog Post

Uso de Gems



- **Existem MUITAS bibliotecas disponíveis em Ruby para os mais variados propósitos**
- Os passos básicos para usar uma **Gem** são:
  a. Encontrar bibliotecas
  b. Instalar bibliotecas localmente
  c. Importar as bibliotecas para o código-fonte
  d. Interagir com a biblioteca através de sua API
- As bibliotecas geralmente possuem código-fonte no Github

Encontrando bibliotecas



- Buscar por bibliotecas

```
$ gem search rails
```

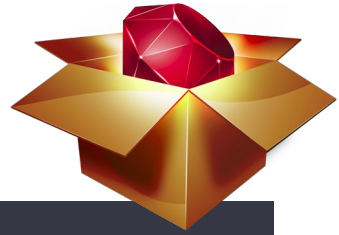
- Busca com mais informações

```
$ gem search remote-user -d
```

- Buscar por bibliotecas instaladas

```
$ gem search -l rails
```

Instalando bibliotecas



- Instalação comum de uma biblioteca

```
$ gem install colorize
```

- Instalação sem documentação

```
$ gem install colorize --no-doc
```

- Instalar uma versão específica

```
$ gem install rails -v 4.0
```

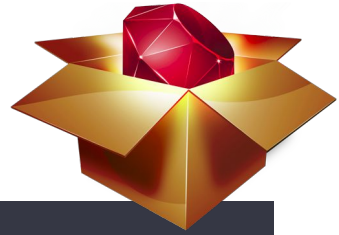
- Listar todas as gems instaladas

```
$ gem list
```

- Remoção de uma gem instalada

```
$ gem uninstall colorize
```

Usando bibliotecas

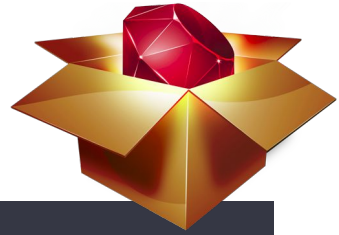


- Ler a documentação localmente

```
$ gem server
```

- Para carregamos a infraestrutura de RubyGems temos que usar:
  - `require 'rubygems'`
- Assim, podemos incluir Gems instaladas
  - `require 'colorize'`

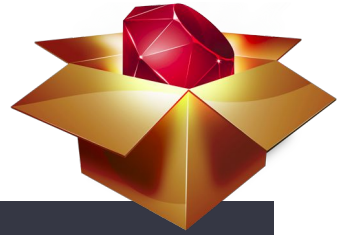
GEM - **Bundler**



```
$ gem install bundler
```

- Bundler proporciona um ambiente para gerenciamento de dependências de RubyGems para projetos em Ruby
- Mapeia e instala as dependências necessárias de um projeto
- Para usá-lo, temos que criar um arquivo na raiz do projeto chamado **Gemfile,** onde especificamos quais Gems e versões são necessárias para esse projeto

GEM - **Bundler**



- Instalando dependências

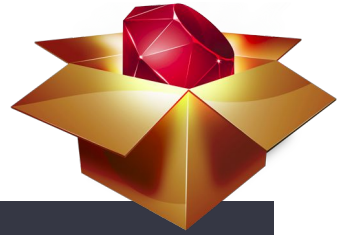
```
$ bundle install
```

- Atualizando dependências

```
$ bundle update
```

- Após a instalação, o **Bundler** vai gerar um arquivo chamado **Gemfile.lock** que contém exatamente quais versões foram instaladas de cada dependência
- Veja o exemplo em um projeto real: https://github.com/Kuniri/kuniri

GEM - **Colorize**

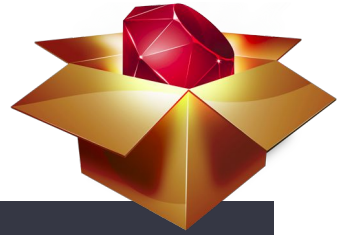


```
$ gem install colorize
```

- A Gem Colorize adiciona vários métodos que permitem a formatação de **cores** e **modos** em Strings
- Veja o exemplo **colorize.rb**

```
==================================
SOME EXAMPLES:
Blue text
Red underline text
Green bold text
```

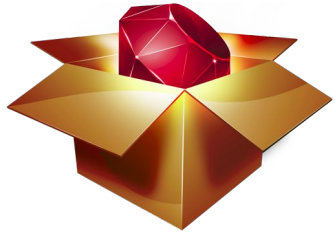
GEM - **PowerPack**



```
$ gem install powerpack
```

- A Gem PowerPack oferece extensões úteis para as classes nativas de Ruby
- Em sua página tem a lista de todos os métodos adicionados em cada classe
- Use o IRB para testar alguns dos métodos adicionados
- Alguma ideia de como esse biblioteca adiciona esses métodos?
- **Desafio**: estenda a classe **Array** e adicione o método **element_types** que retorne uma Hash com a relação dos tipos de elementos existentes e a quantidade de cada tipo
  - **Entrada**: `[1, 2, "oi", :boy].element_types`
  - **Saída**: `{Fixnum=>2, String=>1, Symbol=>1}`

GEM - **Outras Gems**



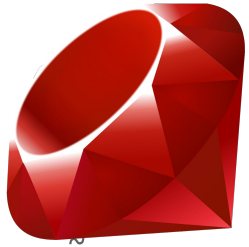
- Existem MUITAS Gems para serem exploradas, tais como:
  - Graticule
  - GLI
  - Mechanize
  - Sinatra
  - Rails
- Você também pode criar sua própria Gem utilizando o Bundler
- Usar bibliotecas é uma das melhoras formas de **reutilização de código**. Muitas vezes algo que você deseja, já está pronto e empacotado como um Gem que você pode reaproveitar.

Metaprogramação

Metaprogramação e Reflexão

- **Reflexão** permite que nós perguntemos aos objetos questões sobre suas propriedades, tais como os métodos :**class**, :**superclass**, :**respond_to**?, etc...
- **Metaprogramação** é o desenvolvimento de códigos que são capazes de modificar e adicionar outros códigos em tempo de execução
- De forma mais simples: **Metaprogramação** é escrever programas que escrevem programas enquanto são executados:
  - Reabrir e modificar classes
  - Criar métodos que não existem à medida que seja necessário
  - Chamar métodos de forma dinâmica
- **Essa é uma das melhores ferramentas do Ruby**
- Na prática, toda programação em Ruby é metaprogramação

Metaprogramação - Classes abertas



- Reabrir classes para modificar seus comportamentos é metaprogramação

```ruby
class Numeric
  def euros
    self * 3.40
  end
  def dollars
    self * 3.14
  end
end

puts "2 reais equals #{2.euros} euros"
puts "2 reais equals #{2.dollars} dollars"
```

Metaprogramação - Exemplo com Mix-ins

- Modificando um programa em tempo de execução com o que já vimos:

```ruby
module EnglishSpeaker
  def talk_in_english
    puts "Hello! I'm talking in english!"
  end
end

class EnglishSchool
  def teach_english(student)
    student.extend EnglishSpeaker
  end
end
```

```ruby
school = EnglishSchool.new
string_student = "Joao"
school.teach_english(string_student)
string_student.talk_in_english
```

Metaprogramação - Conversão para Constante

- Alguns métodos de conversão de strings e símbolos são muito úteis para metaprogramação
- Veja o exemplo do código do método **Learner#become_a** que recebe um parâmetro para definir qual módulo deve estender um determinado objeto

```ruby
module Learner
  def become_a(profession)
    if profession == :programmer
      self.extend Programmer
    elsif profession == :designer
      self.extend Designer
    else
      raise "Cannot become a #{profession}"
    end
  end
end
```

Metaprogramação - Conversão para Constante

- Alguns métodos de conversão de strings e símbolos são muito úteis para metaprogramação
- Veja o exemplo do código do método **Learner#become_a** que recebe um parâmetro para definir qual módulo deve estender um determinado objeto

```
module Learner
  def become_a(profession)
    begin
      module_constant = Kernel.const_get(profession.to_s.capitalize)
      self.extend module_constant
    rescue NameError => e
      raise NameError.new("Cannot become a #{profession}: " + e.message)
    end
  end
end
```

Metaprogramação - Send

- Uma chamada de método significa enviar uma **mensagem** ao objeto **receptor**, onde a mensagem é composta pelo nome do método e seus parâmetros
- Além das formas comuns de chamar métodos, nós podemos invocar qualquer método de um objeto através do método **send** definido em **Object**
- O método send recebe o nome do método que deve ser invocado (como símbolo ou string) e os parâmetros que devem ser passados

```
1 + 2 # => 3
1.+(2) # => 3
1.send(:+, 2) # => 3
```

```
"HELLO WORLD".downcase
"HELLO WORLD".send(:downcase)
```

- Observação: o **send** ignora a visibilidade do método, podendo invocar um método privado

Metaprogramação - Send

- Com o uso do **send**, nós não precisamos saber em tempo de programação qual método deverá ser chamado em um objeto:

```ruby
module EnglishSpeaker
  def say(method_name)
    self.send(method_name)
  end

  def greeting
    puts ['hello', 'good day', "what's up", 'yo', 'hi', 'hey'].sample
  end

  def farewell
    puts ['goodbye', 'see you later', 'bye for now'].sample
  end

  def something_nice
    puts ['you look nice', 'i like your shirt'].sample
  end
end
```

```ruby
class Person
end

person = Person.new
person.extend EnglishSpeaker
person.say(:greeting)
person.say(:something_nice)
person.say(:farewell)
```

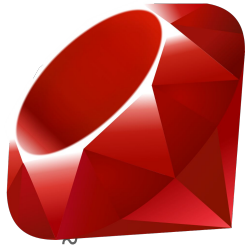
Metaprogramação - Send

- Com parâmetros variados:

```ruby
module DifferentSpeaker
  def say(method_name, method_parameters = [])
    self.send(method_name, *method_parameters)
  end

  def nothing
    puts "nothing"
  end

  def name(name)
    puts "I love this name: #{name}"
  end

  def name_and_age(name, age)
    puts "This is #{name}, he/she is #{age}"
  end
end
```

```ruby
man = Person.new
man.extend DifferentSpeaker
man.say(:nothing)
man.say(:name, ["Arthur"])
man.say(:name_and_age, ["Arthur", 24])
```

Metaprogramação - Chamada de método



- Quando você chama um método em um objeto, o Ruby realiza duas ações:
  a. **Method Lookup:** Ele procura e encontra a definição do método
  b. **Method Execution**: Ele executa o método
- O objeto cujo método foi chamado é definido como o **receptor**
- Se chamarmos o método **my_method** no objeto receptor **a (a.my_method)**, o Ruby vai até a classe do **receptor** procurando a definição do método chamada. Caso essa definição não esteja na classe de **A**, o Ruby procura em todos os módulos incluídos na classe **A** e estendido pelo objeto **a**. Caso não encontre, o Ruby percorre a cadeia de superclasses de **A** em busca dessa definição com o mesmo processo.

Metaprogramação - Chamada de método

```ruby
class A < String
  def duplicate
    self + self
  end
end

a = A.new("Hello")
a.duplicate
a.downcase
a.class
a.non_existent_method
```

- Caso encontre a definição dos métodos chamados em **a** na classe **A**, em um do seus módulos ou superclasses, o método chamado é executado
- Se não encontrar, a exceção **NoMethodError** é lançada

Method Missing

- Quando o Ruby não consegue encontrar um método chamado de um objeto em sua cadeia de classes ou módulos adicionados, ele chama o método **method_missing** no objeto receptor
- Esse **method_missing** está definido no módulo **Kernel** que é incluído na classe **Object**
- O **method_missing** recebe como argumento o nome do método não encontrado (símbolo), um array de argumentos que foram passados para o método original e um bloco de código
- Sua implementação lança a exceção **NoMethodError**
- Como as classes em Ruby são abertas, nós podemos sobrescrever o comportamento do **method_missing** para adicionar novos comportamentos

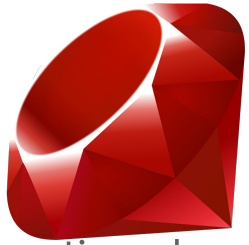
Method Missing - Exemplo 1

- Quando temos composição, é muito comum ficarmos delegando coisas para os objetos que compõem uma classe. Com method_missing isso é ainda mais fácil:

```ruby
class Agenda
  attr_reader :contacts
  def initialize
    @contacts = {}
  end

  def add_contact(name, number)
    @contacts[name] = number
  end
end
```

```ruby
class SmartPhone
  def initialize
    @agenda = Agenda.new
  end

  def method_missing(method, *args, &block)
    if @agenda.respond_to? method
      @agenda.send(method, *args)
    else
      super
    end
  end
end
```

Method Missing - Exemplo 2



```ruby
class Camera
  def initialize
    @flash = true
    @gray_mode = false
    @hdr = true
    @focus_control = true
  end

  def enable_flash
    @flash = true
  end
  def disable_flash
    @flash = false
  end
end
```

- Repare que para permitir o mesmo tipo de método para as outras variáveis eu teria que escrever um método para habilitar e desabilitar cada uma dessas propriedades
- Com method_missing, podemos fazer isso de forma mais **reutilizável**

Method Missing - Exemplo 2

```ruby
class Camera
  def method_missing(method, *args, &block)
    method_name = method.to_s
    if method_name.start_with? 'enable_'
      feature = method_name.sub('enable_', '')
      if self.instance_variables.include? "@#{feature}".to_sym
        return self.instance_variable_set("@#{feature}".to_sym, true)
      end
    elsif method_name.start_with? 'disable_'
      feature = method_name.sub('disable_', '')
      if self.instance_variables.include? "@#{feature}".to_sym
        return self.instance_variable_set("@#{feature}".to_sym, false)
      end
    end
    super
  end
end
```

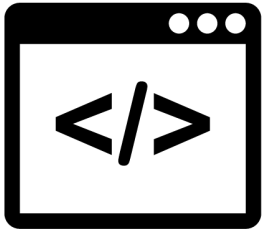
Esse método é suficiente caso a câmera tenha novas opções

Evaluation

- É possível avaliarmos código Ruby em tempo de execução sem esse código ser de fato um código Ruby, mas sim uma string
- O método **eval** recebe uma string para avaliar como um código ruby e o executa considerando o contexto atual

```ruby
eval "puts 2+2"

my_variable = "Hello"
eval "my_variable + ' everyone!'" # => "Hello everyone"
```

Exercício



Utilize o método **eval** para fazer um programa semelhante ao IRB, ou seja um interpretador interativo que recebe código do usuário e o executa imediatamente

- A cada linha, o programa deve imprimir um sinal de maior > para indicar ao usuário onde escrever
- Esse programa não aceitará blocos de código escrito em mais de uma linha
- Esse programa não tratará a criação de variáveis
- Teste coisas simples e nativas como o uso de puts, operações matemáticas e manipulação de strings
- Se o programa lançar exceção, mostre a mensagem da exceção mas não deixe interromper a execução do interpretador iterativo

Class Evaluation

- Podemos reabrir classes utilizando o método **class_eval**
- Assim, podemos escrever novos códigos para essa classe:

```ruby
class Person
end

Person.class_eval do
  def hello
    "I'm a person saying hello!"
  end
end

Person.new.hello # => hello
```

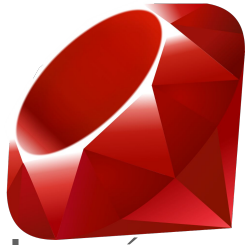
Class Evaluation

- Com **class_eval** ganhamos flexibilidade para fazermos coisas do tipo:

```ruby
module MethodCreator
  def self.create_method(klass)
    klass.class_eval do
      def goodbye
        puts "goodbye"
      end
    end
  end
end
```

```ruby
MethodCreator.create_method(Person)
Person.new.goodbye
MethodCreator.create_method(String)
"I want to say".goodbye
```

Instance Evaluation



- O método **instance_eval** funciona de forma semalhante ao **class_eval,** porém este permite adicionar comportamentos de classe (Ex. métodos estáticos)
- Assim, podemos escrever novos códigos para essa classe:

```ruby
module MethodCreator
  def self.create_method(klass)
    klass.class_eval do
      def goodbye
        puts "goodbye"
      end
    end
  end
end
```

```ruby
MethodCreator.create_class_method(Person)
Person.goodbye
MethodCreator.create_class_method(String)
String.goodbye
```

Removendo métodos

- É possível remover métodos de classes em tempo de execução também
- Isso pode ser útil em alguns casos onde se deseja que um objeto deixe de se comportar de algumas formas - **“Se não anda e nem nada mais como um pato, não deve ser mais um pato”**
- O método de classe **remove_method** remove um método que está definido em uma classe. Porém, não impede que os objetos dessa classe recebam o método removido, caso ele também esteja definido nas superclasses
- O método **undef_method** impede que a classe especificada a responder a chamada de um método mesmo que ele também esteja implementado em superclasses

Removendo métodos com **remove_method**

```ruby
class Animal
  def take_a_nap
    print "Sleeping..."
    sleep 3
  end
end

class Duck < Animal
  def take_a_nap
    super
    print "I'm a lazy duck!"
    sleep 2
  end
end
```

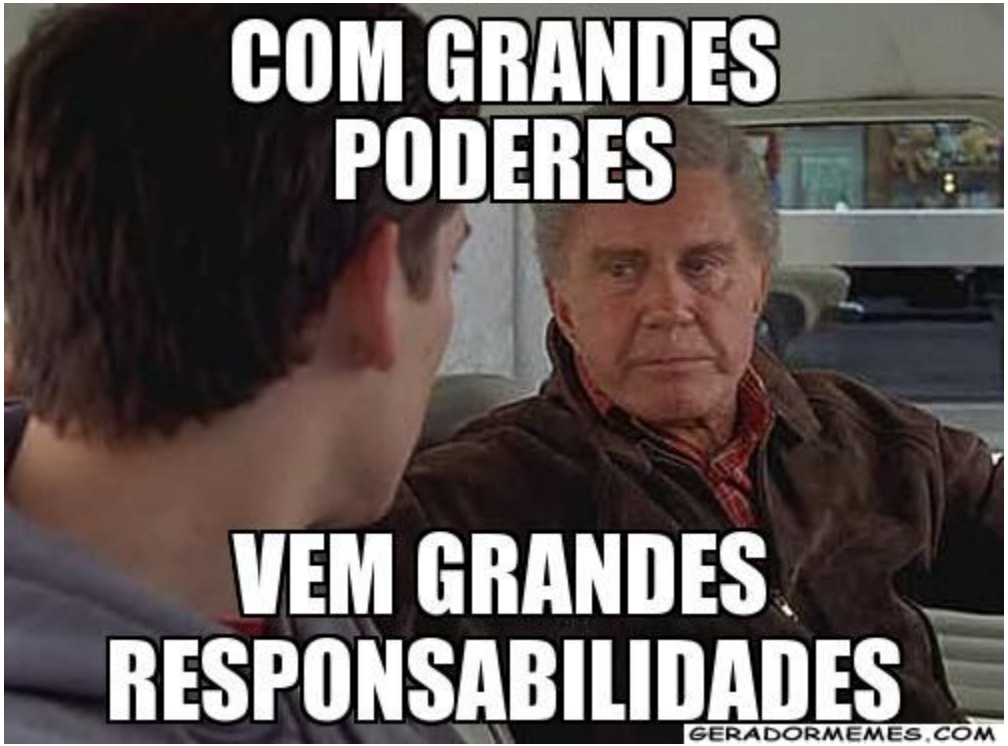
```ruby
Duck.new.take_a_nap # => "Sleeping...I'm a lazy duck!"
Duck.class_eval do
  remove_method :take_a_nap
end
Duck.new.take_a_nap # => "Sleeping..."
```

Removendo métodos com **undef_method**

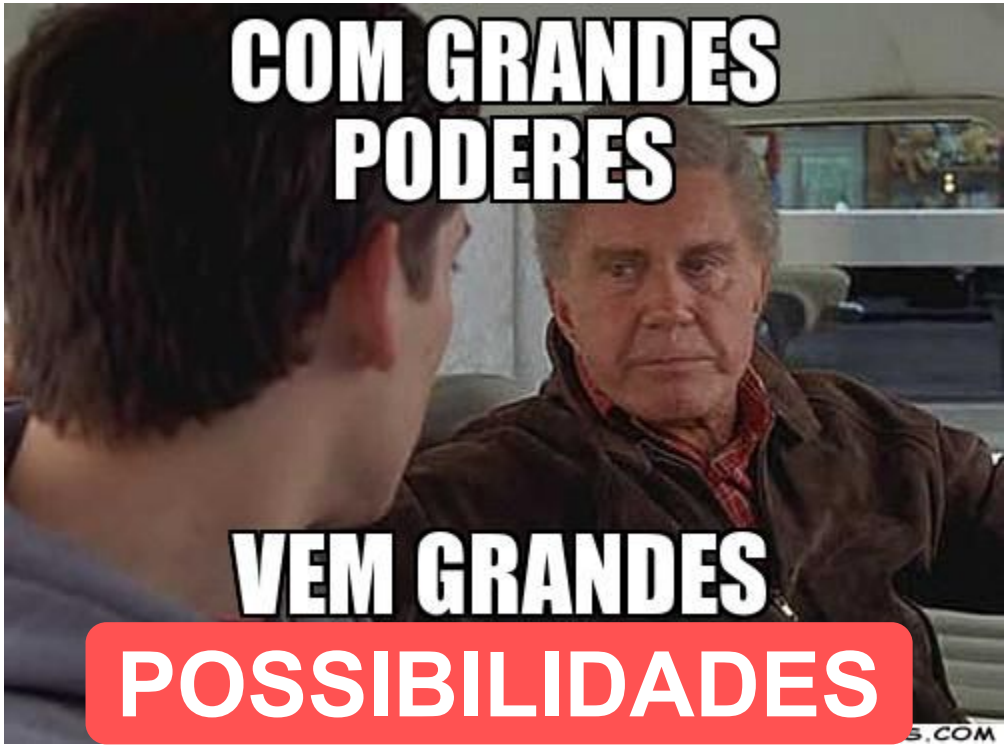
```ruby
class Animal
  def take_a_nap
    print "Sleeping..."
    sleep 3
  end
end

class Duck < Animal
  def take_a_nap
    super
    print "I'm a lazy duck!"
    sleep 2
  end
end
```

```ruby
Duck.class_eval do
  undef_method :take_a_nap
end
Duck.new.take_a_nap # => NoMethodError
```

Metaprogramação



Metaprogramação



Atividades Sugeridas!

Para o Futuro - Estudos Avançados

- Ruby Gems
- Orientação a Objetos, princípios e padrões
- Metaprogramação
- Blocos e fechamentos
- Programação funcional e yield em Ruby
- Manipulação de Arquivos
- Frameworks existentes em Ruby
- Testes automatizados
- Rails

Para o Futuro

- Continue a estudar Ruby:
  - Leia livros
  - Faça exercícios
  - Crie seu próprio programa
  - Contribua para algum software existente
- **A melhor maneira de aprender algo em programação é exercitando e repetindo cada vez mais!**

KEEP CALM

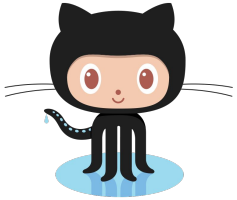
AND

CODE IN

RUBY

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!