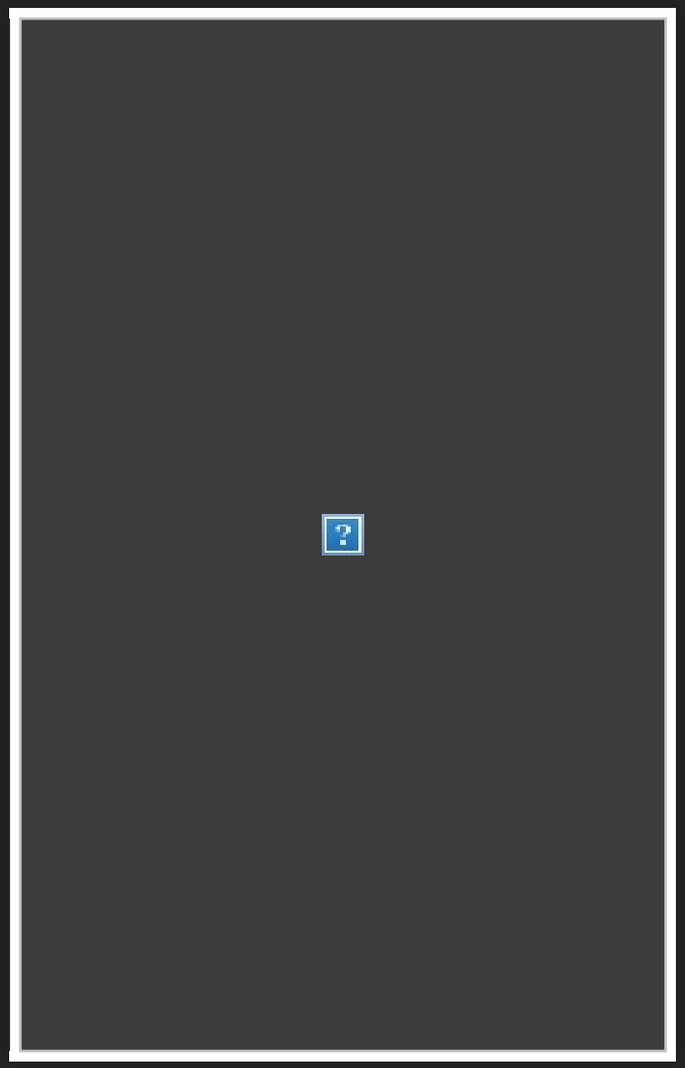
THE SOLID SERIES ™

OPEN CLOSED PRINCIPLE



github: @mracos

twitter: @mractos

- o que é?
- por que foi criado?
- exemplinhos

essa talk é um beta, ela vem sem garantias e não me responsabilizo se as coisas não funcionarem, sua casa pegar fogo ou o maluco do U2 vier cantar na sua casa

- o que é?

Segundo a wikipedia ™:

> *"software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification"*

Significa que tais entidades devem permitir que seu comportamento seja extendido **sem alterar o código fonte**

- por que foi criado? (e por quem?)
- exemplinhos

- qual a forma mais fácil de extender um software?
- Modificando a implementação atual, certo?
- Adicionando um if num método, ou algo assim
- Mas ai depois isso pode ir crescendo e indo ficando complicado de manter

```ruby
class Square; attr_acessor :color, :height, :width; end
class Circle; attr_acessor :color, :radius; end

class Drawer
  def draw(shapes)
    shapes.each do |shape|
      case shape.class
        when Square then draw_square(square)
        when Circle then draw_circle(circle)
      end
    end
  end

def draw_square(square)
  "#{square.height} x #{square.width}: #{square.color}"
```

- e se a gente adicionar mais uma forma? sei lá, `Triangle`
  - teriamos que adicionar um método no `Drawer` e modificar o método `draw`
- e se color retornar algo que não uma string?
- agora *Just imagine* ~~all the people~~ se esse é um sistema complexo com centenas de entidades que dependem de Square
- o custo de mudança acaba sendo enorme

- Bertrand Meyer foi quem coinou o termo, no livro Object Oriented Software Construction (1988!!!)
- O maluco que criou a linguagem *Eiffel* e o conceito de *design by contract*

- originalmente a solução pensada pra resolver esse problema foi usando `herança por implementação`
- uma entidade poderia herdar de um pai um comportamento padrão, mas poderia adicionar sua própria lógica naquela base
- porém depender de classes concretas dificulta a mudança

- Dai veio o Robert C. Martin (titio Bob) com o The Open Closed Principle em 1996 com basicamente dois axiomas

A module will be said to be open if it is still available for extension. For example, it should be possible to add fields to the data structures it contains, or new elements to the set of functions it performs.

A module will be said to be closed if [it] is available for use by other modules. This assumes that the module has been given a well-defined, stable description (the interface in the sense of information hiding)

- depois nos anos 90 foi pensado no uso de classes abstratas e interfaces como forma de atingir isso
- várias classes que herdam de uma classe abstrata e que implementam uma interface portanto são intercambiáveis (pois são polimórficas)

```ruby
class Shape
  attr_accessor :color

  def draw; "drawing: #{@color}"; end
end

class Circe < Shape
  attr_accessor :radius

  def draw; "#{@radius}: #{@color}"; end
end

class Square < Shape
  attr_accessor :height, :width
```

- aqui poderiamos adicionar uma classe `Triangle` facilmente, contanto que mantivesse a mesma interface

- fazer um módulo ser open closed é custoso, abstrações são custosas
- abstrações (as vezes) são mais dificeis de manter
- abstrações incorretas são problemáticas
- sempre vai ter uma mudança que fere o princípio

Questions? Coisas pra compartilhar? Fala ai

Thanks! 

EXTRAS

REFS:

- https://en.wikipedia.org/wiki/Open%E2%80%93close
- https://web.archive.org/web/20060822033314if_/http
- http://stg-tud.github.io/sedc/Lecture/ws16-17/3.3-OC