07

Refatorando e organizando o jogo

Responsabilidades

Mostrando parte da palavra secreta

Nosso jogo ainda é difícil de ganhar: não mostramos as posições onde as letras foram encontradas. Por exemplo, ao chutar a letra `o` deveríamos ter a resposta do programa:

```
__o______o_
```

Depois de `o`, ao chutar `a`:

```
__o__a_a_o_
```

Como podemos fazer isso? Primeiro pensamos onde desejamos implementar esse código. Vamos dar essa informação como resposta ao jogador, como ::feedback::, a cada nova rodada. Portanto vamos imprimí-la durante o processo de pedir um novo chute, dentro do `pede_um_chute`:

```
def pede_um_chute(chutes, erros)
    puts "\n\n\n\n"
    puts "Erros até agora: #{erros}"
    puts "Chutes até agora: #{chutes}"
    puts "Entre com a letra ou palavra"
    chute = gets.strip
    puts "Será que acertou? Você chutou #{chute}"
    chute
end
```

Repare que a função de `pedir_um_chute` é uma função de interface com o jogador, de interface com o usuário (::user interface:: ou ::UI::). Diferente de um código como a função `joga` que é de lógica do nosso jogo, lógica do nosso negócio (::business logic::).

Tudo em um arquivo, 80 linhas, tudo como se fosse um escopo global, qualquer um pode fazer qualquer coisa, de onde venho chamamos isso de "mistura", tem um pouco de tudo, mistura tudo num único recipiente. Na comida fica gostoso, mas aqui fica ruim de entender.

Chegou a hora de começarmos a separar quem é quem, o que é o que. E todo tipo de separação é feita em níveis, já fizemos uma primeira, quando extraímos e isolamos variáveis, segunda quando extraímos e isolamos funções, agora vamos extrair e isolar partes de nosso jogo de acordo com suas responsabilidades. Queremos uma única responsabilidade por arquivo, para ficar mais fácil de encontrar o que está onde.

Separando a interface com o usuário da lógica de negócios

As funções que temos em nosso programa são:

```
da_boas_vindas
sorteia_palavra_secreta
pede_um_chute
nao_quer_jogar?
joga
```

Lembrando que nosso jogo possui uma interface de texto com o usuário, tanto para a entrada quanto para a saída, das funções acima, `da_boas_vindas` , `pede_um_chute` e `nao_quer_jogar?` são funções claramente de interface com o usuário, elas mostram e pedem informação, sem executar nenhum código de lógica, são somente sequências de invocações a funções do tipo `gets` e `puts` , no máximo com uma conversão de um dado de entrada que é `String` para um `booleano` (verdadeiro ou falso).

Já a função `sorteia_palavra_secreta` por enquanto só efetua `puts` , não possui nenhuma lógica, portanto colocaremos também no grupo de interface com o usuário.

Criamos então um arquivo chamado `ui.rb` e colocamos nele nossas funções:

```
def da_boas_vindas
    puts "Bem vindo ao jogo da forca"
    puts "Qual é o seu nome?"
    nome = gets.strip
    puts "\n\n\n\n\n\n"
    puts "Começaremos o jogo para você, #{nome}"
    nome
end

def pede_um_chute(chutes, erros)
    puts "\n\n\n\n"
    puts "Erros até agora: #{erros}"
    puts "Chutes até agora: #{chutes}"
    puts "Entre com a letra ou palavra"
    chute = gets.strip
    puts "Será que acertou? Você chutou #{chute}"
    chute
end

def nao_quer_jogar?
    puts "Deseja jogar novamente? (S/N)"
    quero_jogar = gets.strip
    nao_quero_jogar = quero_jogar.upcase == "N"
end

def sorteia_palavra_secreta
    puts "Escolhendo uma palavra..."
    palavra_secreta = "programador"
    puts "Escolhida uma palavra com #{palavra_secreta.size} letras... boa sorte!"
    palavra_secreta
end
```

É fácil notar que a última função, ::joga::, é na verdade uma mistura de lógica (`ifs` , `laços` etc) e interface com o usuário (`puts`). O que fazer então com ela? Vamos analisar caso a caso cada um dos `puts` de nossa função, extraindo a parte de `UI` da de lógica, desembaralhando o spaguetti que temos.

Primeiro encontramos o código que verifica se já chutamos esse mesmo valor anteriormente:

```
        chute = pede_um_chute chutes, erros
        if chutes.include? chute
            puts "Você já chutou #{chute}"
            next
        end
        chutes << chute
```

O que podemos fazer aqui? Uma solução é criar uma função chamada `avisa_chute_repetido` que somente invoca o `puts` , no nosso arquivo `ui.rb` :

```
def avisa_chute_repetido(chute)
    puts "Você já chutou #{chute}"
end
```

E alterar nosso código de lógica de valor inválido para chamar essa função:

```
        chute = pede_um_chute chutes, erros
        if chutes.include? chute
            avisa_chute_repetido chute
            next
        end
        chutes << chute
```

Depois temos a situação onde procuramos a letra, com dois possíveis resultados impressos:

```
            if total_encontrado == 0
                puts "Letra não encontrada!"
                erros += 1
            else
                puts "Letra encontrada #{total_encontrado} vezes!"
            end
```

Trocamos os mesmos por novos métodos, de responsabilidade bem clara em nosso `ui.rb` :

```
def avisa_letra_nao_encontrada
    puts "Letra não encontrada!"
end
def avisa_letra_encontrada(total_encontrado)
```

```
    puts "Letra encontrada #{total_encontrado} vezes!"
end
```

E os dois casos de chute de palavra completa também podem ser extraídos em funções para o nosso `ui.rb`:

```
def avisa_acertou_palavra
    puts "Parabéns! Acertou!"
end
def avisa_errou_palavra
    puts "Que pena... errou!"
end
```

Nosso último `puts` da função `joga` é aquele que mostra os pontos:

```
def avisa_pontos(pontos_ate_agora)
    puts "Você ganhou #{pontos_ate_agora} pontos."
end
```

Ficando com a função `joga` limpa de qualquer invocação direta a `puts` ou `gets`. Tudo que é feito com a interface do usuário está feito no arquivo `ui.rb`:

```
def joga(nome)
    palavra_secreta = sorteia_palavra_secreta

    erros = 0
    chutes = []
    pontos_ate_agora = 0

    while erros < 5
        chute = pede_um_chute chutes, erros
        if chutes.include? chute
            avisa_chute_repetido chute
            next
        end
        chutes << chute

        chutou_uma_unica_letra = chute.size == 1
        if chutou_uma_unica_letra
            total_encontrado = palavra_secreta.count(chute[0])
            if total_encontrado == 0
                avisa_letra_nao_encontrada
                erros += 1
            else
                avisa_letra_encontrada total_encontrado
            end
        else
            acertou = chute == palavra_secreta
            if acertou
                avisa_acertou_palavra
                pontos_ate_agora += 100
```

```
                break
            else
                avisa_errou_palavra
                pontos_ate_agora -= 30
                erros += 1
            end
        end

    end

    avisa_pontos pontos_ate_agora
end
```

Portanto nosso arquivo `forca.rb` possui a definição da função `joga` e a invocação do jogo:

```
nome = da_boas_vindas

loop do
    joga nome
    break if nao_quer_jogar?
end
```

Testamos rodar nosso `ruby forca.rb` e temos um erro:

```
forca.rb:43:in `<main>': undefined local variable or method
    `da_boas_vindas' for main:Object (NameError)
```

Ele não encontra a função `da_boas_vindas` do nosso outro arquivo, `ui.rb` . Claro, um arquivo não vê outro arquivo por padrão. Seria uma loucura que o interpretador abrisse todos os arquivos de nosso computador automaticamente, por padrão, para buscar tudo o que fosse necessário. Por isso é necessário indicar no arquivo `forca.rb` que precisamos do arquivo de interface com o usuário, requeremos o carregamento do arquivo `ui.rb` relativo ao diretório atual, indicando logo no começo de nosso `forca.rb` :

```
require_relative 'ui'

def joga(nome)
    # ...
end

# ...
```

Agora sim, rodamos nosso jogo e tudo volta a funcionar normalmente!

require

Assim como existe a instrução `require_relative` existem também a `require` . Ela busca em uma lista de diretórios específicos pelo arquivo mencionado. Por exemplo, podemos usar ela para carregar a biblioteca padrão de acesso

HTTP:

```
require "net/http"
require "uri"

uri = URI.parse("http://www.casadocodigo.com.br")
Net::HTTP.get_print(uri)
```

Extraindo a lógica de negócios

Nossa função `joga` é a nossa lógica de negócios principal - e claramente grande demais. Mantemos ela no arquivo chamado `forca.rb`. Mas também pensando assim o início de nosso jogo é um código de lógica:

```
nome = da_boas_vindas

loop do
    joga nome
    break if nao_quer_jogar?
end
```

Nada mais natural nessa visão do que isolá-lo em uma função em nosso `forca.rb` que se chame `jogo_da_forca`:

```
require_relative 'ui'

def joga(nome)
    # ...
end

def jogo_da_forca
    nome = da_boas_vindas

    loop do
        joga nome
        break if nao_quer_jogar?
    end
end
```

Terminamos então nosso jogo nesse instante com dois arquivos. O `forca.rb` com as duas funções de lógica, o `ui.rb` com nossas função de interface com o usuário. Mas e a chamada para o `jogo_da_forca`? Devemos colocar ela em algum lugar. Criamos então o arquivo `main.rb` com a invocação ao início do jogo:

```
require_relative 'forca'

jogo_da_forca
```

Note que todo programa costuma ter uma função de entrada. Nosso código não está mais voando, como código global. Isso é tão comum que em diversas linguagens é obrigatório a definição dessa função principal (::main::).

Extraindo a lógica de um chute válido

Desejamos fazer uma pequena extração de função para ver como ficou mais fácil entender onde está nosso código e onde ele deve ficar. Pense na lógica de não permitir chutes repetidos. Encontrou?

```
        chute = pede_um_chute chutes, erros
        if chutes.include? chute
            avisa_chute_repetido chute
            next
        end
```

Que código feio, note quanta informação em cinco linhas. Isso no meio de uma função de cerca de 40 linhas. Vamos isolá-lo numa lógica clara, que pede um chute ::válido:::

```
chute = pede_um_chute_valido chutes, erros
```

Agora basta definir a função de `pede_um_chute_valido` . Mas calma, onde deve ficar essa função de lógica, que chama nossa UI diversas vezes (até um chute válido)? No arquivo de lógica, claro. Nossa função faz um loop e pede um chute até que o mesmo seja válido:

```
def pede_um_chute_valido(chutes, erros)
    loop do
        chute = pede_um_chute chutes, erros
        if chutes.include? chute
            avisa_chute_repetido chute
        else
            return chute
        end
    end
end
```

Repare que ao pedir novamente um chute, imprimimos novas quatro linhas e todo o cabeçalho, meio repetitivo. Vamos mudar então nossas duas funções que pedem um chute para simplificá-las. Ao pedir o chute mostramos somente uma vez as linhas em branco, os erros e os chutes:

```
# no arquivo ui.rb
def cabecalho_de_tentativa(chutes, erros)
    puts "\n\n\n\n"
    puts "Erros até agora: #{erros}"
    puts "Chutes até agora: #{chutes}"
end

# no arquivo forca.rb
```

```
def pede_um_chute_valido(chutes, erros)
    cabecalho_de_tentativa chutes, erros
    loop do
        chute = pede_um_chute chutes, erros
        if chutes.include? chute
            avisa_chute_repetido chute
        else
            return chute
        end
    end
end
```

Pedir um chute fica mais simples:

```
def pede_um_chute(chutes, erros)
    puts "Entre com a letra ou palavra"
    chute = gets.strip
    puts "Será que acertou? Você chutou #{chute}"
    chute
end
```

Podemos inclusive remover os dois parâmetros ao pedir um chute:

```
def pede_um_chute
    puts "Entre com a letra ou palavra"
    chute = gets.strip
    puts "Será que acertou? Você chutou #{chute}"
    chute
end
```

Claro, ao invocar a função não precisamos mais dos parâmetros:

```
        chute = pede_um_chute
```

Ficando então com as duas funções de `ui`:

```
def pede_um_chute
    puts "Entre com a letra ou palavra"
    chute = gets.strip
    puts "Será que acertou? Você chutou #{chute}"
    chute
end

def cabecalho_de_tentativa(chutes, erros)
    puts "\n\n\n\n"
    puts "Erros até agora: #{erros}"
```

```
    puts "Chutes até agora: #{chutes}"
end
```

E uma de lógica:

```
def pede_um_chute_valido(chutes, erros)
    cabecalho_de_tentativa chutes, erros
    loop do
        chute = pede_um_chute
        if chutes.include? chute
            avisa_chute_repetido chute
        else
            return chute
        end
    end
end
```

Implementação: mostrando parte da palavra secreta

Agora que isolamos duas camadas - lógica de negócio e interface com o usuário - estamos preparados para refatorar e adicionar novas funcionalidades com mais facilidade e organização.

Nosso próximo passo é implementar o mostrador de palavra parcial como citamos anteriormente. Por exemplo, ao chutar as letras `a` e `o` teríamos:

```
__o__a_a_o_
```

Qual o algoritmo para fazer isso? Queremos imprimir cada letra de nossa palavra secreta, então temos que ver se cada uma das letras da palavra secreta já foi chutada. Se sim, usa a letra, se não, usa o `_` (::underline::).

Temos nosso algoritmo! Repare que foram utilizados somente as palavras ::para cada:: (::laço::) e ::se::/::senão:: (::if/else::):

```
para cada letra in palavra_secreta
    if letra ja foi chutada
        usa letra
    else
        usa "_"
    end
end
```

Claro, vamos traduzir para Ruby:

```
for letra in palavra_secreta.chars
    if chutes.include? letra
        puts letra
```

```
    else
        puts "_"
    end
end
```

Esse código é uma função de lógica misturado com interface, não parece ser bom. Além disso cada `puts` vai imprimir uma quebra de linha, teremos uma letra por linha, que não é o que queremos. Para com isso. Ao invés disso, podemos criar uma única função de lógica, `palavra_mascarada` , que, dado os chutes e a palavra secreta, devolve a palavra mascarada:

```
def palavra_mascarada(chutes, palavra_secreta)
    mascara = ""
    for letra in palavra_secreta.chars
        if chutes.include? letra
            mascara += letra
        else
            mascara += "_"
        end
    end
    mascara
end
```

Essa palavra pode ser calculada logo antes de pedir um chute, no começo do laço:

```
# ...
    while erros < 5
        mascara = palavra_mascarada chutes, palavra_secreta
        chute = pede_um_chute_valido chutes, erros
        chutes << chute
# ...
```

E ao pedir o chute, passaremos a máscara também como parâmetro para a interface com o usuário:

```
# ...
    while erros < 5
        mascara = palavra_mascarada chutes, palavra_secreta
        chute = pede_um_chute_valido chutes, erros, mascara
        chutes << chute
# ...
```

Vamos agora na interface com o usuário, fazer com que ao pedir um chute válido, mostremos a mensagem com a máscara:

```
# ui.rb
def cabecalho_de_tentativa(chutes, erros, mascara)
    puts "\n\n\n\n"
    puts "Palavra secreta: #{mascara}"
```

```
    puts "Erros até agora: #{erros}"
    puts "Chutes até agora: #{chutes}"
end

# forca.rb
def pede_um_chute_valido(chutes, erros, mascara)
    cabecalho_de_tentativa chutes, erros, mascara
    loop do
        chute = pede_um_chute
        if chutes.include? chute
            avisa_chute_repetido chute
        else
            return chute
        end
    end
end
```

Agora sim, jogamos uma rodada, escolhendo a letra `a`:

```
Palavra secreta: ___________
...
Será que acertou? Você chutou a
Letra encontrada 2 vezes!
...
Palavra secreta: _____a_a___
Erros até agora: 0
Chutes até agora: ["a"]
...
```

Note que ao adicionar uma nova funcionalidade alteramos o arquivo de lógica, nosso `forca.rb`. Ao alterar a maneira que o programa interage o usuário, alteramos o `ui.rb`. Se adicionamos um nova funcionalidade que também muda a interação com o usuário, alteramos ambos os arquivos.

Resumindo

Aprendemos a separar o código de lógica de negócios do código de interface com o usuário (::UI::) e para deixar tal separação ainda mais óbvia, criamos arquivos distintos, gerando a dependência (o requerimento) entre eles através do `require_relative`.

Ao finalmente adicionar a lógica de mostrar a palavra secreta com os chutes nas posições adequadas fomos capazes de perceber que a estrutura criada nos ajuda a definir onde vai cada modificação: uma nova lógica de negócios e uma nova interação através da interface com o usuário vão em arquivos distintos.

Em sistemas maiores ou utilizando práticas de controle de escopo (::namespace::), de orientação a objetos (::OO::) ou linguagens funcionais, somos capazes de organizar ainda mais nosso código. A medida que evoluir no aprendizado de linguagens você terá a oportunidade de aprender cada um desses conceitos, tudo no momento adequado e dependendo do caminho que escolher para seu próximo passo.