06

Movimentações do fantasmas e do herói e posições.

Botando os fantasmas para correr: arrays associativos, duck typing e outros

Array associativo: case e +1, -1

Já falamos anteriormente que sequências de `ifs`, e o uso de `cases` são possíveis ::code smells::. Eles indicam que pode estar presente algo sujo, que pode atrapalhar a manutenção do código ou facilitar a criação de novos bugs.

Nossa função `calcula_nova_posicao` demonstra exatamente este cenário:

```
def calcula_nova_posicao(heroi, direcao)
    heroi = heroi.dup
    case direcao
        when "W"
            heroi[0] -= 1
        when "S"
            heroi[0] += 1
        when "A"
            heroi[1] -= 1
        when "D"
            heroi[1] += 1
    end
    heroi
end
```

Como mudar nosso código para evitar esse `case`? Repare que o que fazemos aqui é basicamente mapear uma mudança de posição para cada tecla, e a mudança é feita tanto na linha (posição `0` da array) quanto na coluna (posição `1`).

```
def calcula_nova_posicao(heroi, direcao)
    heroi = heroi.dup
    case direcao
        when "W"
            heroi[0] -= 1
            heroi[1] += 0
        when "S"
            heroi[0] += 1
            heroi[1] += 0
        when "A"
            heroi[0] += 0
            heroi[1] -= 1
        when "D"
            heroi[0] += 0
            heroi[1] += 1
    end
    heroi
end
```

Podemos padronizar mais ainda, utilizando somente somas, seja de um número positivo ou negativo:

```
def calcula_nova_posicao(heroi, direcao)
    heroi = heroi.dup
    case direcao
        when "W"
            heroi[0] += -1
            heroi[1] += 0
        when "S"
            heroi[0] += 1
            heroi[1] += 0
        when "A"
            heroi[0] += 0
            heroi[1] += -1
        when "D"
            heroi[0] += 0
            heroi[1] += 1
    end
    heroi
end
```

Podemos também padronizar criando duas variáveis, `anda_linha` e `anda_coluna` :

```
def calcula_nova_posicao(heroi, direcao)
    heroi = heroi.dup
    case direcao
        when "W"
            anda_linha += -1
            anda_coluna += 0
        when "S"
            anda_linha += +1
            anda_coluna += 0
        when "A"
            anda_linha += 0
            anda_coluna += -1
        when "D"
            anda_linha += 0
            anda_coluna += +1
    end
    heroi[0] += anda_linha
    heroi[1] += anda_coluna
    heroi
end
```

Mas Guilherme, seu código está ainda maior que o que possuíamos antes. É verdade, ele está maior, mas mais claro. Ele diz que se você me passa determinado caractere, eu faço você andar um número determinado de linhas e colunas, isto é, eu ::mapeio:: uma letra (uma chave), para um número (um valor).

W anda -1 e 0. S anda +1 e 0. A anda 0 e -1. D anda 0 e +1. Eu crio uma conexão entre uma chave (nesse caso, o caractere) e um valor (o quanto ele anda em cada dimensão, linha e coluna). Adivinha? Ruby já tem essa estrutura, alguém que

mapeia algo para outra coisa, assim como um dicionário mapeia palavras para suas definições, aqui um dicionário mapeará, conectará um caractere com o quanto devemos andar:

```
heroi = heroi.dup
movimentos = {
    "W" => [-1, 0],
    "S" => [+1, 0],
    "A" => [0, -1],
    "D" => [0, +1]
    }
```

E usamos o dicionário como se fosse um array, afinal ele é um array, mas um array que associa chave e valor, um ::array associativo:::

```
heroi = heroi.dup
movimentos = {
    "W" => [-1, 0],
    "S" => [+1, 0],
    "A" => [0, -1],
    "D" => [0, +1]
    }
movimento = movimentos[direcao]
heroi[0] += movimento[0]
heroi[1] += movimento[1]
```

Isto é, trocamos nosso `case` ou sequência de `ifs` por um array associativo:

```
def calcula_nova_posicao(heroi, direcao)
    heroi = heroi.dup
    movimentos = {
        "W" => [-1, 0],
        "S" => [+1, 0],
        "A" => [0, -1],
        "D" => [0, +1]
        }
    movimento = movimentos[direcao]
    heroi[0] += movimento[0]
    heroi[1] += movimento[1]
    heroi
end
```

Existe uma boa e válida discussão aqui, se essa refatoração em si é justificável. O código final é mais explícito, mas utilizamos um artifício para chegar onde queríamos. Nesse código, por exemplo, o que acontece se utilizarmos uma tecla inválida? E no código anterior o que aconteceria?

Claro, tratamento de tecla inválida deve ser feito de qualquer maneira, mas a utilização de dicionários (arrays associativos, mapas etc) como instrumento para execução de lógica de negócios tem que ser tratada com cuidado. Pense sempre se o código final justificou sua utilização.

Acredito que o código no nosso exemplo atual é mais explícito no que está fazendo e qual o resultado final da função. Mas não acredito que arrays associativos possam ser abusados como recurso lógico. Como regra geral, verifique a quantidade extra de `ifs` que irá adicionar ao seu código somente por usar um array associativo. No final das contas, valeu a pena?

Movimento dos fantasmas: o desafio no duck typing

Chegou a hora dos nossos inimigos moverem. Começaremos alterando nosso código para que o mapa carregado seja o segundo:

```
def joga(nome)
    mapa = le_mapa(2)
    # ...
end
```

E no fim de nossa jogada, movemos os fantasmas:

```
def joga(nome)
    mapa = le_mapa(2)
    while true
        # ...

        move_fantasmas mapa
    end
end
```

Para o primeiro passo de "inteligência" de nosso inimigo, faremos com que os fantasmas andem sempre para a direita. Como fazer isso? Queremos pegar todos os fantasmas e andar uma casa para a direita. Quebremos essa afirmação, como crianças, em passos básicos para cada uma das frases:

1. Queremos pegar todos os fantasmas
2. e andar uma casa para a direita

Ótimo, primeiro queremos todos os fantasmas. Como encontrá-los? Podemos varrer nosso array de strings, procurando o caractere `F`. Para isso primeiro passamos por todas as linhas:

```
def move_fantasmas(mapa)
    caracter_do_fantasma = "F"
    mapa.each_with_index do |linha_atual, linha|
    end
end
```

Agora passamos por cada caractere de uma linha:

```
def move_fantasmas(mapa)
```

```
    caracter_do_fantasma = "F"
    mapa.each_with_index do |linha_atual, linha|
        linha_atual.each_with_index do |caractere_atual, coluna|
        end
    end
end
```

Verificamos se ele é um fantasma e, se sim, movemos:

```
def move_fantasmas(mapa)
    caracter_do_fantasma = "F"
    mapa.each_with_index do |linha_atual, linha|
        linha_atual.each_with_index do |caractere_atual, coluna|
            eh_fantasma = caractere_atual == caracter_do_fantasma
            if eh_fantasma
                move_fantasma
            end
        end
    end
end
```

Agora que encontramos todos eles, precisamos implementar o `move_fantasma` onde fazemos ele andar. Primeiro tiramos ele da posição atual:

```
def move_fantasma(mapa, linha, coluna)
    mapa[linha][coluna] = " "
end
```

Movemos ele para direita:

```
def move_fantasma(mapa, linha, coluna)
    mapa[linha][coluna] = " "
    linha += 0
    coluna += 1
end
```

E colocamos ele na nova posição:

```
def move_fantasma(mapa, linha, coluna)
    mapa[linha][coluna] = " "
    linha += 0
    coluna += 1
    mapa[linha][coluna] = "F"
end
```

E ao invocar passamos os parâmetros necessários:

```
move_fantasma mapa, linha, coluna
```

Pronto. Nossos fantasmas podem andar... testamos uma rodada:

```
fogefoge.rb:31:in `block in move_fantasmas':
    undefined method `each_with_index' for
        "XXXXXXXXX":String (NoMethodError)
```

O que acontece? `String` não possui o método `each_with_index`? Nem tudo que parece um pato é um pato. Antes eu queria alguém com `each`, `size` e `[]`. Agora eu quero com `each_with_index`. Eu enganei a mim mesmo, menti para mim quando disse que queria alguém que se parecesse com um array, que uma `String` bastava. Eu já estava atrelado a existência e significado de três métodos de um array, agora estou a quatro. Não parece ser pouca coisa o suficiente para dizer que eu queria somente alguem que se comportasse como um array. Eu queria um array, estou triste.

Deveríamos ter trocado antes? Agora? Nunca deveríamos ter trabalhado com `String`? A discussão é longa e com ótimos argumentos para os dois lados. Para nosso aprendizado esse é o momento ideal para efetuarmos a troca, e visualizarmos a vantagem e uma primeira desvantagem do uso de duck typing. A segunda desvantagem é quando o código roda, e o efeito é inesperado. A grande vantagem era até agora não ter precisado se preocupar com isso.

O que fazer para corrigir nosso problema? Uma solução seria ao ler o mapa, transformar as linhas de `String`s em `Array`s de caracter, utilizando o método `chars` em cada linha. Mas a impressão de um array de caracteres ficaria horrível, teríamos que reconcatenar os caracteres para formar um `String`. Note que realmente quem parece um pato não é um pato. ::Todo:: momento que usamos nosso mapa, usamos ele sabendo que ele era um array de String. Mudar o tipo é mudar tudo, não é mudar pouco.

Nossa solução será, portanto, simples. Ao precisarmos do método `each_with_index` invocaremos o método `chars`:

```
def move_fantasmas(mapa)
    caracter_do_fantasma = "F"
    mapa.each_with_index do |linha_atual, linha|
        linha_atual.chars.each_with_index do |caractere_atual, coluna|
            eh_fantasma = caractere_atual == caracter_do_fantasma
            if eh_fantasma
                move_fantasma mapa, linha, coluna
            end
        end
    end
end
```

Agora sim, rodamos o jogo, movemos uma para a esquerda e temos os fantasmas andando uma para a direita:

```
...
XXXXXXXXX
X H     X
X X XXX X
X X X   X
```

```
X   X X X
  X     X
 XXX XX X
  X     X
X   X X X
XXXF  F X
XXX XXX X
XXX XXX X
XXX     XPara onde deseja ir?
A
XXXXXXXXX
XH      X
X X XXX X
X X X   X
X   X X X
  X     X
 XXX XX X
  X     X
X   X X X
XXX F  FX
XXX XXX X
XXX XXX X
XXX     X
Para onde deseja ir?
```

A sacada do duck typing é que ele é genial ao permitir que o interpretador não se preocupe com a tipagem nem em tempo de compilação (na implemetação padrão do ruby compilação é quando ele valida a sintaxe do seu arquivo), nem durante a execução do código. O interpretador somente está preocupado se o valor referenciado pela nossa variável responde a função, ao método, em questão. Se sim, ótimo.

Por outro lado, nós como desenvolvedores estamos preocupados com a tipagem ao escrevermos nosso código. Não importa ter um valor que responda ao método `size` se daqui a pouco preciso do método `each_with_index`, o que eu quero agora é alguém que se comporte quase como um `array`, alguém que tenha os mais e mais métodos de `array`. É arriscado afirmar que estamos totalmente atrelados ao tipo em si, mas uma vez que estamos atrelados ao significado do método que invocamos, estamos atrelados a algo além do que somente a existência dele, porém o interpretador não está preocupado com o tipo. Assim podemos definir a dinamica de tipos no Ruby... o interpretador não se preocupa com eles, nós nos preocupamos sempre que precisamos escrever ou alterar nosso código.

Movimento dos fantasmas: reutilização de função

Mas se andarmos novamente, agora para baixo, os fantasmas movem mais um para a direita, entrando em contato com o muro.

```
...
XXXXXXXXX
X       X
XHX XXX X
X X X   X
X   X X X
  X     X
 XXX XX X
  X     X
```

```
X   X X X
XXX F  FX
XXX XXX X
XXX XXX X
XXX     X
Para onde deseja ir?
S
XXXXXXXXX
X       X
X X XXX X
XHX X   X
X   X X X
  X     X
 XXX XX X
  X     X
X   X X X
XXX  F  F
XXX XXX X
XXX XXX X
XXX     X
```

Faltou validarmos a posição de nossos fantasmas. Já temos a função `posicao_valida?` , basta invocá-la verificando se o resultado é válido antes de mover o fantasma. Começamos criando a nova posição em um array:

```
def move_fantasma(mapa, linha, coluna)
    posicao = [linha, coluna + 1]
end
```

Só alteramos os valores de nosso mapa se a posição for válida:

```
def move_fantasma(mapa, linha, coluna)
    posicao = [linha, coluna + 1]
    if posicao_valida? mapa, posicao
        mapa[linha][coluna] = " "
        mapa[posicao[0]][posicao[1]] = "F"
    end
end
```

Caso a posição não seja válida, o fantasma fica parado. Testamos nos movimentar duas vezes e agora o fantasma fica encostado na parede, não passa por ela:

```
XXXXXXXXX
X  H    X
X X XXX X
X X X   X
X   X X X
  X     X
 XXX XX X
  X     X
X   X X X
```

```
XXX F  FX
XXX XXX X
XXX XXX X
XXX     X
Para onde deseja ir?
D
XXXXXXXXX
X   H   X
X X XXX X
X X X   X
X   X X X
  X     X
 XXX XX X
  X     X
X   X X X
XXX  F FX
XXX XXX X
XXX XXX X
XXX     X
```

Fantasma contra fantasma?

Se andarmos um quarto passo, o mapa se torna:

```
...
XXXXXXXXX
X       X
X X XXX X
X X X   X
XH  X X X
  X     X
 XXX XX X
  X     X
X   X X X
XXX    FX
XXX XXX X
XXX XXX X
XXX     X
```

O que aconteceu? Um dos fantasmas sumiu. Acontece que enquanto um dos fantasmas ficou parado, o outro tomou seu lugar. Com isso, perdemos um deles. Devemos então proibir um fantasma de andar para a posição de outro fantasma. A solução inicial é não permitir que ele ande para uma casa onde já existe um `F` marcado. Bloquearemos a possibilidade de um fantasma andar em cima de outro. Aliás, não faz sentido o jogador ir em direção ao fantasma. Portanto alteramos nossa função de `posicao_valida?` para conferir se ela possui muro (`X`) ou fantasma (`F`):

```
    if mapa[posicao[0]][posicao[1]] == "X"
        return false
    end
    if mapa[posicao[0]][posicao[1]] == "F"
        return false
    end
```

Opa, se estamos usando essa posição duas vezes, ::extract variable:: fica mais claro:

```
valor_local = mapa[posicao[0]][posicao[1]]
if valor_local == "X" || valor_local == "F"
    return false
end
```

Agora sim, após andar diversos passos para a direita os fantasmas não brigam mais, ficam encostados um ao outro:

```
XXXXXXXXX
X      HX
X X XXX X
X X X   X
X   X X X
  X     X
 XXX XX X
  X     X
X   X X X
XXX   FFX
XXX XXX X
XXX XXX X
XXX     X
```

Resumindo

Vimos como funciona o dicionário, um array associativo, e tivemos que tomar cuidado com mudanças de +1 e -1, algo extremamente perigoso em programação. Entendemos melhor o que significa o ::duck typing:: e sua vantagem além de corrigir um bug que surgiu no jogo, onde dois fantasmas ocupavam a mesma posição. Vimos que o duck typing permite que o interpretador fique despreocupado em relação a tipagem, mas que nós ainda nos preocupamos com o significado e existência dos métodos, portanto com quem definiu eles, seus tipos. Vimos também que tanto bugs quanto funcionalidades são descritos através de estruturas de lógica, com controle de fluxo e laços.