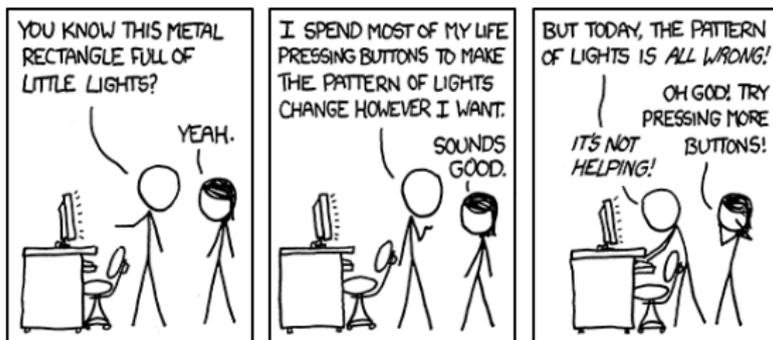


Sobre Programação Estruturada

Pedro Ribeiro

DCC/FCUP

2022/2023



Sobre Programação Estruturada

- Este capítulo serve essencialmente para vos dar alguns (bons) **exemplos** de como criar um programa.
- Algumas boas "**regras metodológicas**":
 - ▶ **Ler o enunciado com atenção** (ou definir bem o problema). Perceber bem quais as tarefas e as restrições (ex: quais os limites do *input*?)
 - ▶ **Não começar logo a escrever código**. Primeiro **pensar** como vou organizar: como posso dividir as tarefas em blocos mais pequenos? Que entidades existem? Que classes vou precisar? Onde vou guardar os dados? Quais os métodos essenciais?
 - ▶ **Escrevam "código para humanos"**. A melhor documentação é um código estruturado e bem escrito. Usem nomes adequados de variáveis e métodos e adicionem pitadas de comentários nas partes não triviais. Procurem implementar de forma a que se vocês próprios forem ver o código daqui a um mês ainda percebam o que o código faz.

Sobre Programação Estruturada

- **É expressamente proibido fazer tudo no main() :**) Procurem dividir o programa em blocos/métodos pequenos, especializados, que podem ser implementados e testados de forma independente.
- **Generalizem.** Se estão a "repetir" código e usar muito *copy+paste*, é indício que poderiam ter generalizado e ter um único pedaço de código para fazer essa tarefa. Evitem repetições e reduzam os sítios onde podem cometer erros.
- **Escrever um bloco/método de cada vez e ir testando sempre cada pedaço de código. É importantíssimo ir testando à medida que se implementa.** Fizeram a leitura? Testem! **E não apaguem o código que usaram para testar** - basta por exemplo comentar.

Sobre Programação Estruturada

- **Testem o vosso programa.** Usem no mínimo os exemplos de input dados, mas façam outros testes. Não dependam do Mooshak para perceber se o programa está correcto.

Para testar os exemplos de input copiem a partir do enunciado e colem num ficheiro (introduzir manualmente dá origem a erros!). Se o ficheiro se chamar `input.txt` e o vosso programa estiver numa classe `ED`, então executem o seguinte comando na vossa *shell*:

```
$ javac ED.java && java ED < input.txt
```

Isto compila o programa, e caso a compilação tenha sucesso, executa o programa redireccionando o input a partir do ficheiro que contém o exemplo que querem testar. *Com um único comando conseguem logo ver o comportamento do vosso programa!*

- **Tenham orgulho nos vossos programas.** Programar é uma "arte". Não se contentem com o Accepted. Se vêem que podem ter algo melhor, procurem perceber como fazer. Perguntem. Se eu for ver o vosso código, estariam contentes com o que me estão a mostrar?

Um exemplo: Jogo do Galo e Generalização

- Problema [ED004] - Jogo do Galo: <http://www.dcc.fc.up.pt/~pribeiro/aulas/edados2223/problemas/prob004.html>
- Problema pede para, dado um tabuleiro de $n \times n$, saber o estado do jogo (alguém ganhou? houve um empate? o jogo está por terminar?)
- Quais as classes a criar?
 - ▶ Uma classe para o programa e o método main: ED004
(*usar o código 004 permite depois rapidamente identificar o programa, pois todos os problemas de EDados têm um número associado*)
 - ▶ Uma classe Game para definir um jogo do Galo
- Quais os métodos a criar na classe Game?
 - ▶ Game(int n) - construtor para criar o tabuleiro $n \times n$
 - ▶ read(Scanner in) - método para ler o conteúdo do tabuleiro
 - ▶ check() - um método para verificar o estado do tabuleiro
(ou seja, para resolver o problema)

Um exemplo: Jogo do Galo e Generalização

- A **classe principal** fica muito simples e consegue "ler-se" o que faz:
 - 1 ler o tamanho do tabuleiro
 - 2 criar um jogo com esse tamanho
 - 3 ler o conteúdo do tabuleiro
 - 4 verificar o estado do jogo (resolver o problema pedido)

```
public class ED004 {  
    public static void main(String[] args) {  
        Scanner in = new Scanner(System.in);  
        int n = in.nextInt();  
        Game g = new Game(n);  
        g.read(in);  
        g.check();  
    }  
}
```

Um exemplo: Jogo do Galo e Generalização

- Adicionemos a classe `Game`.
- Que atributos deve ter?
 - ▶ um inteiro n para o tamanho
 - ▶ uma matriz de caracteres m para o tabuleiro
- *By default*, Colocamos atributos como privados (só tornamos públicos se realmente for necessário)
- Precisamos também do construtor para criar um jogo de $n \times n$

```
class Game {
    private int n;           // tamanho do tabuleiro
    private char m[][];     // matriz que representa o tabuleiro

    // Construtor que recebe como argumento o tamanho n
    Game(int size) {
        n = size;
        m = new char[n][n];
    }

    // (...)
}
```

Um exemplo: Jogo do Galo e Generalização

- Adicionemos mais métodos à classe `Game`
- A leitura do *input* é fácil
- Temos de receber o `Scanner` usado no `main` como argumento (nunca criar dois `Scanners` do `System.in` no mesmo programa!)
- O método `next()` lê um *token*: um conjunto de caracteres delimitado por *whitespace* (espaços, tabs, mudanças de linhas, etc)
- Depois de adicionar este método devemos testá-lo (ex: criar método para escrever conteúdo da matriz e chamar para ver se ficou bem lido)

```
void read(Scanner in) {
    for (int i=0; i<n; i++) {
        String buf = in.next();           // next() lê um "token" (uma string s)
        for (int j=0; j<n; j++)
            m[i][j] = buf.charAt(j);
    }
}
```

Um exemplo: Jogo do Galo e Generalização

- Verificar se o jogo terminou é mais complicado
- Uma implementação *naive* pensaria nos casos todos como separados:
 - ▶ Será que alguma linha está toda preenchida?
 - ▶ Será que alguma coluna está toda preenchida?
 - ▶ Será que alguma das duas diagonais principais está toda preenchida?
- Isto teria de ser verificado para 'x' e para 'o'.
- Claramente existe maneira melhor do que fazer tanta coisa "parecida" muitas vezes, com copy+paste de código... Como generalizar?

Um exemplo: Jogo do Galo e Generalização

- No fundo estamos sempre a verificar se uma linha tem os mesmos caracteres:
 - ▶ O que muda é a orientação (horizontal, vertical ou diagonal)
- Seja (x, y) a posição inicial da linha. Verificar uma linha é começar aí e ir modificando essas coordenadas para "percorrer" a linha.
- Seja $incx$ e $incy$ os incrementos que se fazem a x e y para ver o elemento seguinte da linha:
 - ▶ $incx = 1$ e $incy = 0$ incrementa apenas o x (linha horizontal)
 - ▶ $incx = 0$ e $incy = 1$ incrementa apenas o y (linha vertical)
 - ▶ $incx = 1$ e $incy = 1$ incrementa ambos (diagonal)
 - ▶ $incx = 1$ e $incy = -1$ incrementa ambos (a outra diagonal)
- Verificar agora uma linha é só começar no sítio certo, e fazer um ciclo de tamanho n , chamando os incrementos correctos, e verificar se o caracter é sempre o mesmo (seja ele 'x' ou 'o')

Um exemplo: Jogo do Galo e Generalização

- A função `verify(int y, int x, int incy, int incx)` da classe `Game` verifica se a linha que começa em (x, y) e usando os incrementos `incx` e `incy` é uma linha completamente preenchida.
- Notem alguns pormenores como:
 - ▶ Verificamos se todos os caracteres da linha são iguais ao primeiro (generalizando para 'X' ou 'O').
 - ▶ No for podemos ter várias instruções separadas por vírgula:
 - ★ `int i=0, yy=y, xx=x` é executado no início do ciclo
 - ★ `i++, yy+=incy, xx+=incx` é executado no final de cada iteração

```
void verify(int y, int x, int incy, int incx) {
    if (m[y][x] == '.') return; // Posição está por preencher
    for (int i=0, yy=y, xx=x; i<n; i++, yy+=incy, xx+=incx)
        if (m[y][x] != m[yy][xx]) return;
    win(m[y][x]);
}

// Método auxiliar para escrever o resultado (seja qual for o jogador)
void win(char player) {
    System.out.println("Ganhou o " + player);
    System.exit(0); // Sai do programa
}
```

Um exemplo: Jogo do Galo e Generalização

- Estamos agora prontos para criar o método `check()` da classe `Game` para resolver o problema
- Notem como a subtarefa de verificar se um tabuleiro está completo está separada num outro método (*dividir código em blocos pequenos*)
- Notem como o código ficou legível e sem repetição.

```
void check() {
    for (int i=0; i<n; i++) verify(i, 0, 0, 1); // Linhas
    for (int j=0; j<n; j++) verify(0, j, 1, 0); // Colunas
    verify(0, 0, 1, 1); // Diagonal 1
    verify(0, n-1, 1, -1); // Diagonal 2

    if (!finished()) System.out.println("Jogo incompleto");
    else System.out.println("Empate");
}

// Devolve true se o jogo já terminou ou false caso contrário
boolean finished() {
    for (int i=0; i<n; i++)
        for (int j=0; j<n; j++)
            if (m[i][j] == '.') return false;
    return true;
}
```