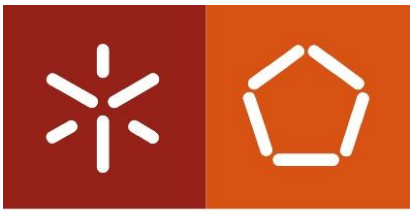


Universidade do Minho

Mestrado Integrado em Engenharia Informática

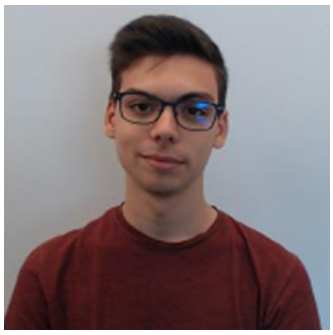
POO - Trabalho Prático

Grupo 05

João Barbosa (a93270) Simão Cunha (a93262)
Tiago Silva (a93277)

Ano Letivo 2020/2021





Conteúdo

Capítulo 1

Introdução e principais desafios

Este projeto consistiu em criar um sistema de gestão e simulação de equipas de um determinado desporto (neste caso futebol) muito semelhante ao jogo *Football Manager*, de forma a aplicarmos os vários conhecimentos adquiridos nas aulas ao longo do semestre.

Tivemos alguns obstáculos para a concessão deste projeto, como o planeamento da arquitetura do programa (principalmente a hierarquia dos `atributos`) e a gestão de tempo que condicionaram o resultado final.

Capítulo 2

Classes

2.1 Main

Classe responsável pelo arranque do programa. Para isso, o método `main` invoca o método `run` da classe `Controlo`.

2.2 Controlo

```
private TratamentoDados cd;
private final Scanner scan;
private final String[] menuPrincipal;
private final String[] menuDados;
private final String[] gravarDados;
private final String[] lerDados;
private final String[] menuSimulacao;
private final String[] menuSimulacoes;
private final String[] menuSimulacaoRapida;
private final String[] menuEditarEquipa;
private final String[] menuEditarJogador;
private final String[] menuCriarDados;
private final String[] menuCriarJogador;
private final String[] menuPosicao;
private final String[] menuEscolheTatica;
private final String[] menuAtributos;
private final String[] menuatributosGR;
private final String[] menuatributosDF;
private final String[] menuatributosMD;
private final String[] menuatributosLT;
private final String[] menuatributosAV;
private final String[] simulacao;
```

`Controlo` é a classe responsável pelo controlo da aplicação. Associa um menu a uma funcionalidade do programa. Por exemplo, mostra no ecrã a simulação de um jogo de futebol, recorrendo a métodos da classe `ExecutaPartida` e com mensagens pré-concebidas para vários momentos do

jogo.

Neste menu temos as seguintes opções:

1. *Simular uma partida de futebol*: é possível escolher equipas, definir o seu onze inicial, esquema tático, efetuar um jogo, fazer simulações *rápidas* e *normais*,etc

2. *Manipular dados*: é possível carregar, criar, ver/editar, gravar e fazer reset a dados

2.3 TratamentoDados

```
private Map<String,EquipaFutebol>equipas;
private Map<String,List<PartidaFutebol>>partidas;
```

`Controlo de dados` é a classe que guarda todos os dados da aplicação, como as equipas com os seus jogadores (num *HashMap* onde cada chave é o nome da equipa) e as partidas de futebol executadas por cada equipa (informação que é guardada num *ArrayList* que está numa *HashMap* onde a chave é o nome da equipa).

Obviamente, ao adicionar cada partida, vai ter 2 entradas na *HashMap* de partidas, pois uma partida é realizada por 2 equipas.

2.4 ViewJogo

```
private static Scanner is;
private boolean continuacao;
private List<String> opcoes;
private List<PreCondition> disponivel;
private List<Handler> handlers;
```

A classe `ViewJogo`(criada a partir do exemplo `DriveIt` das aulas práticas), tal como o nome indica, oferece toda a visualização à nossa aplicação. Gere tanto o fluxo do programa através da gestão de menus controlada pela classe `Controlo` como oferece um suporte visual à `ExecutaPartida`, como, por exemplo, o método `comentarioJogo`, que mostra vários comentários durante a execução de uma partida.

2.5 Jogador

```
private int numero;
private String nome;
private String posicao;
private List <String> historial;
private Atributos atributos;
```

Esta é uma das classes essenciais da aplicação. A classe `jogador` representa um jogador de futebol, onde são armazenados o número da camisola, o seu nome, a sua posição em campo, o historial dos clubes por onde passou e os seus atributos (como a capacidade de remate, velocidade, passe,...). Existem **5** tipos de jogador: guarda-redes, defesa, lateral, médio e avançado.

2.6 CompPosicao

A classe `CompPosicao` implementa um *Comparator* para a classe `Jogador`. Este `Comparator` compara dois jogadores através das suas posições ocupadas em campo. Se forem iguais, o critério passa a ser o número da camisola.

NB: Guarda-Redes $\rightarrow$ $Defesa \rightarrow Lateral \rightarrow Médio \rightarrow Avançado$

2.7 Atributos

```
private int velocidade;
private int resistencia;
private int destreza;
private int impulsao;
private int jogoDeCabeca;
private int remate;
private int controloDePasse;
```

A classe `Atributos` é abstrata, pois os valores atribuídos para cada parâmetro variam conforme a posição do jogador. Como será visível mais à frente, decidimos adicionar mais atributos a jogadores com posições diferentes, mais própriamente em classe hereditárias. Esta classe obriga a criação de 2 métodos abstratos: *overall* e *desgaste*.

2.7.1 AtributosGR

```
private int elasticidade;
private int reflexos;
```

Esta classe refere-se aos atributos de um guarda-redes. Herda os atributos da classe `Atributos` e acrescenta mais 2 atributos extra: a elasticidade e os reflexos. O *overall* faz uma média pesada dos atributos, tendo maior peso os atributos *reflexos*, *elasticidade* e *impulsão*. O desgaste de um guarda-redes é menor do que para as outras posições e descresce um pouco todos os atributos consoante a sua resistência.

2.7.2 AtributosLateral

```
private int precisaoCruzamentos;
private int drible;
```

Esta classe herda atributos da classe `Atributos` e decidimos acrescentar mais 2 atributos: a precisão em efetuar cruzamentos e a capacidade de driblar um adversário. O *overall* faz uma média pesada dos atributos, tendo maior peso os atributos *precisão de cruzamentos* e *resistência*. O desgaste de um lateral descresce um pouco todos os atributos consoante a sua resistência.

2.7.3 AtributosDefesa

```
private int posicionamentoDefensivo;
private int cortes;
```

Esta classe herda atributos da classe `Atributos` e decidimos acrescentar mais 2 atributos: o posicionamento defensivo e a capacidade de cortar a bola. O *overall* faz uma média pesada dos atributos, tendo maior peso os atributos *posicionamento defensivo* e *cortes*. O desgaste de um defesa descresce um pouco todos os atributos consoante a sua resistência.

2.7.4 AtributosMedio

```
private int recuperacaoDeBolas;
private int visaoDeJogo;
```

Esta classe herda atributos da classe `Atributos` e decidimos acrescentar mais 2 atributos: a capacidade de recuperação da bola e a visão de jogo. O *overall* faz uma média pesada dos atributos, tendo maior peso os atributos *visão de jogo* e *controlo de passe*. O desgaste de um médio descresce um pouco todos os atributos consoante a sua resistência.

2.7.5 AtributosAvancado

```
private int penaltis;
private int desmarcacao;
```

Esta classe herda atributos da classe `Atributos` e decidimos acrescentar mais 2 atributos: a capacidade de marcação da marca dos onze metros e a desmarcação para se isolar dos adversários. O *overall* faz uma média pesada dos atributos, tendo maior peso os atributos *remate*, *desmarcação* e *jogo de cabeça*. O desgaste de um avançado descresce um pouco todos os atributos consoante a sua resistência.

2.8 Plantel

```
private Map<Integer,Jogador> titulares;
private Map<Integer,Jogador> suplentes;
private int nJogadoresNoPlantel;
private Tatica tatica;
```

A classe `Plantel` representa um plantel de futebol, com um *map* de jogadores no onze inicial, um *map* de jogadores no banco, o número total de jogadores num plantel e o esquema tático adotado pela equipa.

NB: o número total de jogadores de um plantel não excede 22.

2.9 Tatica

```
    private int nGR;
    private int nDF;
    private int nLT;
    private int nMD;
    private int nPL;
```

A classe `Tatica` define o esquema tático de uma equipa. Para tal, será necessário saber o número de guarda-redes (que será sempre 1), o número de defesas, o número de laterais, o número de médios e o número de avançados

2.10 EquipaFutebol

```
    private String nome;
    private Plantel plantel;
```

O conceito da classe `EquipaFutebol` pode ser confundido com o conceito da classe `Plantel`. Uma `Plantel` é dividido em vários parâmetros para serem mais facilmente acedidos pelo utilizado. Já uma `EquipaFutebol` engloba todos estes parâmetros num objeto `Plantel`, incluindo o nome da equipa.

2.11 PartidaFutebol

```
    private double tempo;
    private int golosVisitante;
    private int golosVisitado;
    private int substituicoesVisitante, substituicoesVisitados;
    private int[][] subsVisitante, subsVisitada;
    private EquipaFutebol equipaVisitante, equipaVisitada;
    private LocalDate data;
};
```

A classe `PartidaFutebol` representa uma partida de futebol. São incluídos o tempo de jogo, o número de golos da equipa visitante, o número de golos da equipa da casa, o número de substituições que a equipa visitante pode efetuar, o número de substituições da equipa da casa que ainda pode fazer, um *array* de substituições para cada equipa (visitante ou visitada) onde o jogador é representado pelo seu número de camisola, os equipas que vão jogar e a data da realização do jogo.

2.12 ExecutaPartida

```
    private PartidaFutebol partida;
    private Jogador jogadorAtual;
    private Boolean casa;
    private Boolean comecou;
    private Random random;
    private final ViewJogo v;
    private boolean comentariosON;
    private static final double acaoRapida = 0.5;
    private static final double acaoMedia = 1.0;
```

A classe `ExecutaPartida` é responsável por, tal como diz o nome, executar uma partida de futebol. É constituído por uma partida de futebol, pelo jogador que possui atualmente a bola, um booleano que verifica se o jogador é da equipa caseira, um booleano que verifica que equipa começa o jogo com a posse de bola (já que ao intervalo a posse é dada à equipa contrária), um objeto `Random` para gerar imprevisibilidade ao jogo e duas constantes que simbolizam uma *ação rápida* (por exemplo, tentar passar a bola) e uma *ação média* (por exemplo, marcar golo). Tem ainda uma variável da classe *ViewJogo* que oferece suporte visual à partida e um booleano *comentariosON* que liga/desliga os comentários durante o jogo.

NB: Ao longo do projeto foi usada a interface *ANSIIColour* que possibilita colorir *strings*.

Capítulo 3

Estrutura do projeto

O nosso projeto segue a estrutura *Model View Controller* (MVC), estando por isso organizado em três camadas:

- A camada de dados (o **"Model"**) é composta pelas classes Jogador, Atributos, AtributosGR, AtributosDefesa, AributosLateral, AtributosMedio, AtributosAvancado, EquipaFutebol, Plantel, Tatica, PartidaFutebol, ExecutaPartida e TratamentoDados e pela interface CompPosicao.
- A camada de interação com o utilizador (a **"View"**, ou apresentação) é composta unicamente pela classe ViewJogo.
- A camada de controlo do fluxo do programa (o **"Controller**) é composta unicamente pela classe Controlo.

Achamos que o nosso projeto cumpriu o princípio da programação orientada aos objetos: o *encapsulamento*. São exemplos notórios deste cumprimento os métodos *getters* e *setters* nas classes com objetos privados mutáveis, pois estes invocam sempre o método `clone`.

Capítulo 4

Conclusão

A nível geral, e tendo em conta o que foi explicado nos capítulos anteriores, podemos afirmar que temos um projeto bem conseguido, apesar de não estar exatamente igual ao que idealizávamos. Acreditamos que respondemos de forma correta à simulação de uma partida de futebol, que foi um dos pontos mais trabalhosos do nosso projeto. Além disso, é possível um crescimento controlado da nossa aplicação, uma vez que é possível, por exemplo, adicionar mais desportos no diretório *Desporto*, adicionar posições mais específicas aos jogadores, assim como aumentar a hierarquia de atributos.

Apêndice A

Diagrama de Classes

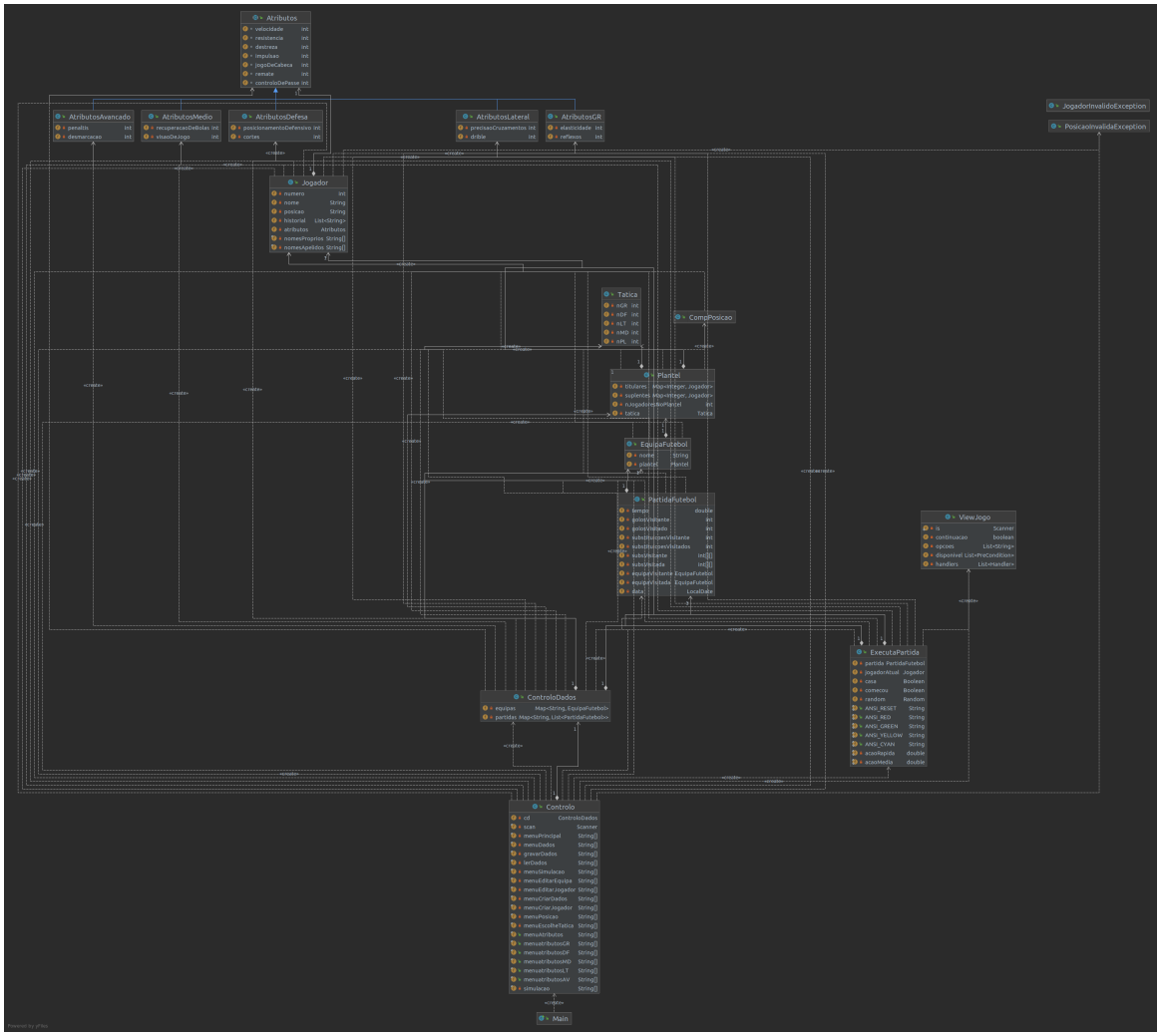


Figura A.1: Diagrama de classes do programa, gerado pelo *IntelliJ*