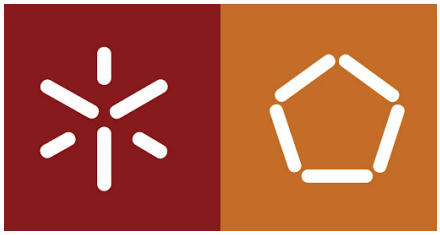


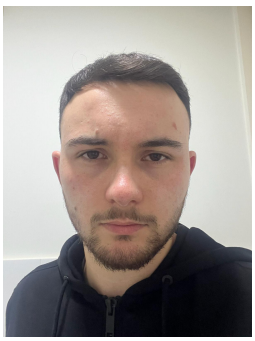
Universidade do Minho
Escola de Engenharia
Departamento de Informática

Programação Orientada a Objetos

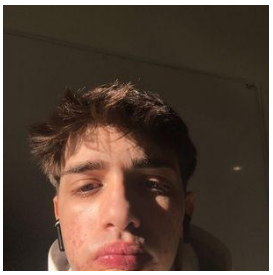
Relatório do Trabalho Prático

1 de Maio de 2024

Desenvolvido por:



Gonçalo Alves - a104079



João Cunha - a104611

João Sá - a104612

Conteúdo

1 Introdução

Este relatório tem como intuito abordar o projeto da UC Programação Orientada a Objetos do ano letivo 2023/2024.

Ao longo deste relatório iremos abordar as decisões tomadas e funcionalidades implementadas, entre outros. O objetivo deste trabalho foi desenvolver uma aplicação que faça a gestão das actividades e planos de treino, de praticantes de actividades físicas, à qual atribuímos o nome de "FITDOWN".

2 Interface

A nossa interface segue um estilo de interação com a aplicação através da escolha de opções. Tal como é possível observar na imagem seguinte, existem várias opções para o utilizador escolher, sendo que existe uma função responsável por desenhar cada tipo de interface (e.g. Home Page, Register/Login Page, User Page).

Figura 1: Interface Menu Inicial

Para escolher obter a escolha do utilizador, utilizamos um *scanner* que estará à espera de um valor do tipo inteiro. Caso o valor não seja um inteiro, o sistema irá avisar o utilizador e não irá prosseguir até que o valor esteja correto.

```java
if (scanner.hasNextInt()) {
  return scanner.nextInt();
} else {
  System.out.println("Por favor, escolha uma opção válida!");
  scanner.nextLine();
  System.out.println("Pressione Enter para continuar...");
  scanner.nextLine();
  return MenuInicio(dataSistema);
}
```

Figura 2: Verificação do input na função *MenuInicio*

Esta verificação, neste caso, é um tanto quanto simples, porém, em casos mais específicos tais como no registro de um novo utilizador. Neste caso, o utilizador deve seguir um formato específico ao introduzir os dados. O formato requisitado deve ser cumprido para que o parser, posteriormente, funcione sem nenhum problema. Para isso, utilizamos o conceito de RegEx (Regular Expression). Antes do input, definimos o formato desejado, que vai ser comparado com os dados introduzidos do utilizador, utilizando o método *Pattern.matches*.

```java
String input = scanner.nextLine();

// Expressão regular para verificar o formato da entrada
String formato = "\\d+;[^;]+;[^;]+;[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,};\\d+;\\d+";

if (Pattern.matches(formato, input)) {
  return input;
} else {
  System.out.println("Formato incorreto. Por favor, insira novamente.");
  System.out.println("Pressione Enter para continuar...");
  scanner.nextLine();
  return register();
}
```

Figura 3: Verificação do input na função *Register*

De modo a tornar a aplicação funcional, decidimos criar uma classe *Interactive*. Esta classe é responsável pelo funcionamento do programa em relação aos menus. Através de *switches*, os métodos chamam o menu correspondente e o mesmo se volta a aplicar para os menus chamados

3 Modelos

A aplicação tem presente vários modelos, cada um com características únicas. Todos os modelos foram implementados de forma a se manter uma expansão de funcionalidades sustentável do programa

3.1 Atividades

Propusemos as seguintes atividades: *Burpees, Deadlift, Mountain Bike, Canoagem.* Todas estas atividades estendem uma classe abstrata *Atividade* que contém informações comuns de todas as atividades (data de início, data de fim, duração, frequência cardíaca associada ao utilizador, tipo de atividade, gasto calórico, dificuldade). É possível construir o seguinte diagrama de classes (com omissões):

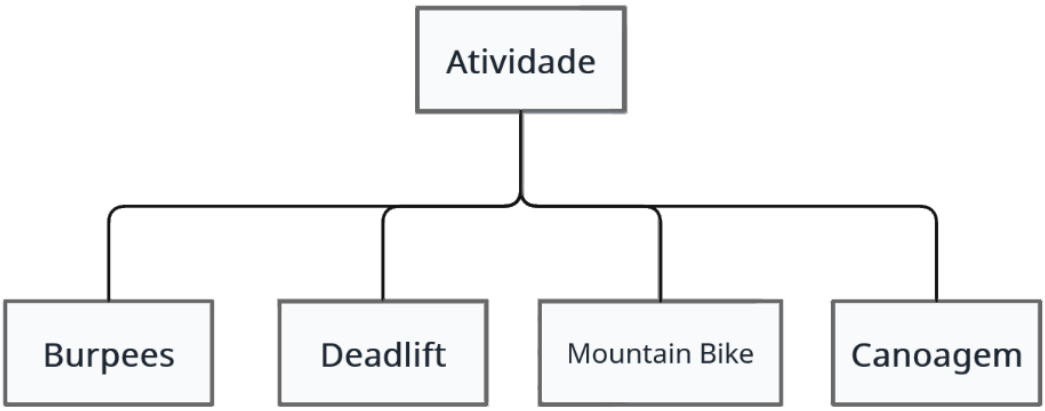


Figura 4: Diagrama de classes das Atividades

Para tornar o programa mais realista possível, decidimos criar um método para cada tipo de atividade. Este método, no caso da atividade *Deadlift*, recebe um utilizador (do qual retiramos o seu peso) e determinadas características da atividade (número de repetições, número de sets, descanso e o peso total)

```
public boolean isDeadliftHard(Utilizador user) {
  int reps = this.getNumReps();
  int sets = this.getNumSets();
  int descanso = this.getDescanso();
  int pesoTotal = this.getPesoTotal();
  int pesoUser = user.getPeso();

  return (
    (reps >= 8 && sets >= 3 && descanso <= 3) || pesoTotal >= 1.5 * pesoUser
  );
}
```

Figura 5: Implementação da lógica da dificuldade da atividade Deadlift

Para além destes métodos especifícos de cada classe, cada uma contém informações adicionais de acordo com o tipo de atividade (ex: Deadlift tem peso barra/total).

3.2 Utilizadores

O nosso programa permite a existência de 3 tipos de utilizadores: *Amadores, Ocasionais, Profissionais*. Todos estes tipos de utlizador são subclasses que extendem a superclasse abstrata *Utilizador*. Todas as diferentes subclasses tem as mesmas características (Codigo, nome, morada, email, frqueência cardíaca média, peso, listas com as atividades agendadas e realizadas e listas com os planos agendados e realizados)

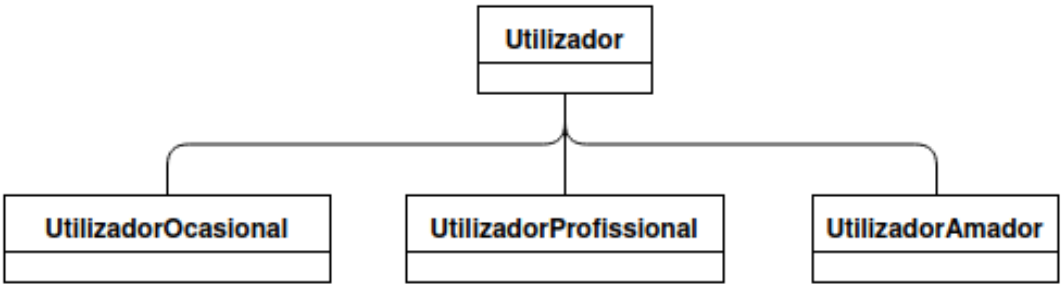


Figura 6: Diagrama de classes dos Utilizadores

Implementamos um metodo abstrato na superclasse para obter o fator calorico de cada utilizador, que vai diferir com as suas características.

```
abstract double calculaFatorCalorico();
```

Figura 7: Método abstrato

Na imagem abaixo temos um exemplo da implementação deste método para utilizadores ocasionais:

```
public double calculaFatorCalorico() {
  return (0.8 * (this.getfcm() / 60.00));
}
```

Figura 8: Exemplo da implementação

Além deste método abstrato, não existe nenhuma diferença entre utilizadores como dito acima , exceto o seu tipo/classe.(ex: Amador, Ocasional)

3.3 Planos de Treino

Desenvolvemos uma classe chamada de *PlanoTreino*, que define um plano treino no nosso programa , e tem como características uma lista de atividades que constroem o plano , e uma data de realização do mesmo. São associados os planos aos utilzidores através da lista de planos que existe na classe *Utilizador* mencionada acima.

Esta classe não tem subclasses associadas, nem é abstrata , o que a torna mais simples.

Os planos são diários.

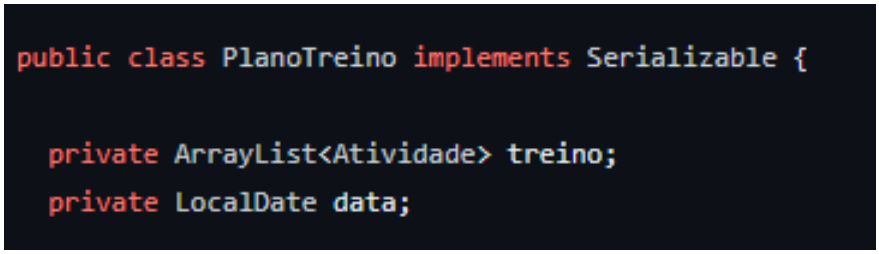


Figura 9: Exemplo da implementação

4 Funcionalidades

4.1 Adicionar Atividades , Utilizadores e Planos

Como requisitado, tornamos possível adicionar Atividades , Utilizadores e Planos de treino no nosso programa.

4.1.1 Adicionar Utilizadores

Para adicionar utilizadores, temos de especificar o tipo de utilizador primeiro :

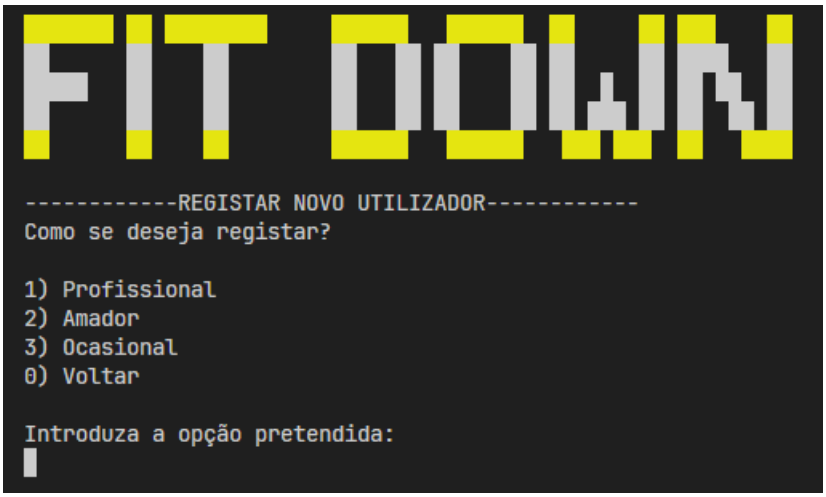


Figura 10: Registo de utilizador

De seguida, terão de ser dados como input ao programa algumas características específicas , como no exemplo abaixo:

Figura 11: Registo de utilizador 2

Estas características serão lidas pelo nosso Parser de Utilizadores presente na classe *Parsing*:

```
public class Parsing {

  public static Utilizador parserUtilizador(String chave, int tipo) {
    String tipoUser = null;

    String[] parts = chave.split(regex:";");
    String codigoUtilizador = parts[0];
    String nome = parts[1].trim();
    String morada = parts[2].trim();
    String email = parts[3].trim();
    Double fcm = Double.parseDouble(parts[4]);
    int peso = Integer.parseInt(parts[5]);
```

Figura 12: Parser Utilizadores

Este mesmo parser será chamado na nossa classe *Controlador*, que vai adicionar o utilizador registado á lista de utilizadores do programa com a função *insereUtilizador*.

```
public static void insereUtilizador(
  int menu,
  String register,
  Utilizadores utilizadores
) {
  utilizadores.setUtilizador(Parsing.parserUtilizador(register, menu));
}
```

Figura 13: Adiciona Utilizadores

4.1.2 Adicionar Atividades

Tal como ao adicionar utilizadores, primeiro especifica-se o tipo de atividade:

Figura 14: Registo de Atividade

De seguida , preenchemos o input com os detalhes da atividade específica , que vão mudar dependendo do tipo. Na imagem seguinte temos o exemplo de MountainBike:

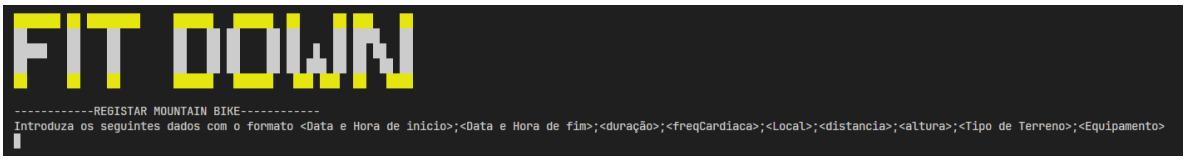


Figura 15: Registo de Atividade 2

Os mesmos dados serão lidos pelo nosso Parser da atividade específica presente na classe *Parsing*. De seguida temos o exemplo de um dos parsers de atividades:

```
public static Deadlift parserDeadlift(String chave, Utilizador utilizador) {
  String[] parts = chave.split(regex:";");

  DateTimeFormatter formatter = DateTimeFormatter.ofPattern(
    pattern:"dd/MM/yyyy HH:mm"
  );
  try {
    Deadlift atividade = new Deadlift(
      LocalDateTime.parse(parts[0], formatter),
      LocalDateTime.parse(parts[1], formatter),
      Integer.parseInt(parts[2]),
      Double.parseDouble(parts[3]),
      Integer.parseInt(parts[4]),
      Integer.parseInt(parts[5]),
      Integer.parseInt(parts[6]),
      Integer.parseInt(parts[7]),
      Integer.parseInt(parts[8])
    );
    return atividade;
```

Figura 16: Registo de Atividade 2

E tal como ao adicionar os utilizadores, a função *insereAtividade* da classe *Controlador* trata de adicionar a atividade registada, associá-la ao utilizador que a registou, e calcular as calorias dispendidas durante a mesma:

```
if (menu == 1) {
  atividade_aux = Parsing.parserDeadlift(register, utilizador);
} else if (menu == 2) {
  atividade_aux = Parsing.parserMountainBike(register, utilizador);
} else if (menu == 3) {
  atividade_aux = Parsing.parserBurpees(register, utilizador);
} else if (menu == 4) {
  atividade_aux = Parsing.parserCanoagem(register, utilizador);
}
if (atividade_aux == null) {
  return;
}
atividade_aux.calculaCalorias(utilizador);
if (atividade_aux instanceof Deadlift) {
  boolean dif = ((Deadlift) atividade_aux).isDeadliftHard(utilizador);
  atividade_aux.setDificuldade(dif);
}
if (!atividades.getAtividades().contains(atividade_aux)) {
  atividades.setAtividade(atividade_aux);
}

utilizador.addAtividade(atividade_aux);
```

Figura 17: Registo de Atividade 3

4.1.3 Adicionar Planos de Treino

Neste caso utilizamos um método um bocado diferente. Ao registar um Plano de Treino , um utilizador escolherá ao ínicio quantas atividades o plano terá:

Figura 18: Registo de Plano De Treino

De seguida , é chamada a função *insereAtividade* (esta função é semelhante á insereAtividade, apenas atribuímos a data das atividades ao plano) do *Controlador* dentro de um loop, até o plano ser preenchido pelo número de atividades escolhido pelo utilizador, e só no final do loop associámos o plano ao utilizador caso o mesmo não esteja vazio:

```
if (n_atividades == 0 && !(plano.getTreino().isEmpty())) {
  utilizadorLogado.addPlano(plano);
  System.out.println(
    x:"\n\nPlano adicionado com sucesso! Pressione ENTER para prosseguir...\n"
  );
  System.out.println(x:"Pressione ENTER para prosseguir...\n");
  scanner.nextLine();
```

Figura 19: Registo de Plano De Treino

4.2 Time Skip

O sistema guarda, num estado, a data do mesmo. Se uma data tiver sido guardada, ao efetuar "Salvar", essa data será carregada. Caso seja a primeira vez a executar o programa (ou seja, não existe uma data guardada), o sistema tem a data predefinida 11/04/2024 00:00.

Após efetuado o login, o utilizador tem a opção de avançar no tempo. Para tal, deve selecionar a opção "Avançar Data" no Menu. Após selecionada a opção, o utilizador tem duas opções relativamento ao avanço.

4.2.1 Avançar um dia

Para avançar um dia, optámos por aplicar a função predefinida ***plusDays***, com o valor "1", à data atual do sistema.

Figura 20: Interface para avançar a data

4.2.2 Introduzir uma data

Para a esta opção, o utilizador deve introduzir uma data por extenso. Para tal, deve ter em conta as seguintes restrições:

- A data introduzida deve ser uma data válida.
- A data introduzida deve seguir o formato requisitado.
- A data introduzida não deve ser anterior à data atual do sistema

Tal como podemos observar na figura abaixo, caso um dos requisitos acima referidos não for cumprido, a data não será atualizada.

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern(
            "dd/MM/yyyy HH:mm"
          );
          try {
            LocalDateTime dataNova = LocalDateTime.parse(
              dataIntroduzida,
              formatter
            );
            if (dataNova.isBefore(estado.getDataEstado())) {
              System.out.println(
                "\nA data introduzida é anterior à data do sistema, por favor introduza uma data posterior!"
              );
            } else {
              estado.setDataEstado(dataNova);
              System.out.println(estado.getDataEstado());
            }
          } catch (DateTimeParseException e) {
            System.out.println(
              "\nA data introduzida é inválida, por favor tente novamente!"
            );
          }
```

Figura 21: Lógica para avançar a data por extenso

4.3 Persistência do estado de um programa

Para guardar o estado do programa em disco, recorremos à funcionalidade de serialização de classes do Java. Para tal, criámos um **Estado**, que é responsável por criar (caso ainda não exista) um ficheiro chamado *estado.obj* e guardar as informações relativas ao programa (Atividades, Utilizadores, Data Sistema).

Na classe *Estado*, também está definido um método para ler o estado do ficheiro, e carregá-lo no sistema.

Com esta funcionalidade, o utilizador pode reter os seus dados e atividades e também é possível popular a aplicação com utilizadores, atividades ,etc, dando assim mais "vida"à mesma.

4.4 Estatísticas

Foram implementadas 7 estatísticas. Cada uma foi implementada na classe mais oportuna. As estatísticas implementadas são as seguintes:

4.4.1 Estatística I - Qual o utilizador que mais calorias gastou.

Nesta estatística, percorremos os utilizadores, calculando as calorias gastas de cada um, a partir da soma das calorias gastas por cada atividade realizada pelos utilizadores e comparamos-las para obter o utilizador que mais calorias gastou. Para o caso de ser num intervalo, fazemos o mesmo processo, mas verificamos se a data da atividade se encontra entre as datas de início e fim dadas pelo utilizador.

4.4.2 Estatística II - Qual o utilizador que mais atividades realizou

Aqui percorremos os utilizadores, e vemos o tamanho da ArrayList das atividades realizadas e comparamos-las para obter a maior. Para o caso de ser num intervalo, fazemos o mesmo processo, mas verificamos se a data da atividade se encontra entre as datas de início e fim dadas pelo utilizador.

4.4.3 Estatística III - Qual é o tipo de atividade mais realizada

Neste caso, temos 4 variáveis int, onde cada uma representa um tipo de atividades e que à medida que vamos percorrendo a List de atividades realizadas, somamos uma unidade à variável que representa o tipo dessa atividade. Finalmente, comparamos as variáveis e vemos qual o tipo ou os tipos de atividade mais realizada.

4.4.4 Estatística IV - Quantos Quilômetros percorreu o utilizador

Para este caso, percorremos a lista de atividades realizadas pelo utilizador e verificamos a classe de cada uma e caso estas sejam "Canoagem"ou "MountainBike", somamos os quilômetros precorridos pelo utilizador em cada iteração. Para o caso de ser num intervalo, fazemos o mesmo processo, mas verificamos se a data da atividade se encontra entre as datas de início e fim dadas pelo utilizador.

4.4.5 Estatística V - Quantos metros de altimetria foram totalizados pelo utilizador

Para este caso, percorremos a lista de atividades realizadas pelo utilizador e verificamos a classe de cada uma e caso esta seja "MountainBike", somamos a altura da atividade. Para o caso de ser num intervalo, fazemos o mesmo processo, mas verificamos se a data da atividade se encontra entre as datas de início e fim dadas pelo utilizador.

4.4.6 Estatística VI - Qual o plano de treino mais exigente em função do dispêndio de calorias proposto

Nesta estatística, percorremos os utilizadores e percorremos os planos de treino de cada utilizador, somando o dispêndio de calorias de cada plano de treino, comparando ao dos outros.

4.4.7 Estatística VII - Listar as atividades de um utilizador

Nesta estatísticas, criamos uma string na qual juntamos as atividades do ArrayList atividadesRealizadas e as atividades do ArrayList atividadesAgendadas do utilizador.

4.5 Gerador de Planos de Treino de acordo com Objetivos

Com a implementação desta funcionalidade, o sistema é capaz de gerar um Plano de Treino de acordo com algumas preferências do utilizador em questão, tendo também em conta o seu tipo.

Num primeiro momento, é questionado ao utilizador se quer atividades *hard* no seu plano, e de seguida o mesmo tem de introduzir uma série de preferências sobre o plano:

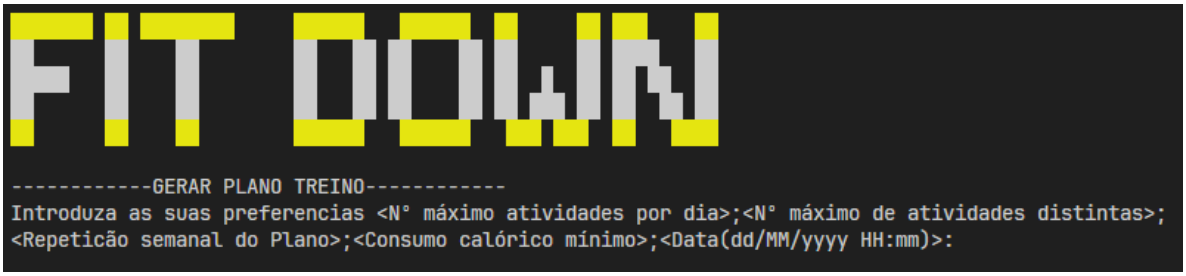


Figura 22: Geração inteligente de Planos

O nosso programa lê as preferências do utilizador e chama a função *geraPlanoTreino*. Primeiramente, a função itera sobre a lista de todas as atividades existentes no sistema e verifica se o utilizador quer ou não atividades *hard*. Em caso afirmativo, verifica se não existe nenhuma atividade *hard* nos dias vizinhos.
Em caso de não existir, adicionámos a atividade presente no iterador ao plano, e junta-mos o seu gasto calórico ao gasto calórico total do plano , para comparar com o objetivo do utilizador.

```
do {
    for (Atividade atividade : atividades.getAtividades()) {
      if (isHard) {
        if (
          atividades_aux.size() < n_atividadesDiarias &&
          !(Atividades.checkHardActivityDayBefore(utilizador, data)) &&
          !(Atividades.checkHardActivityDayFollowed(utilizador, data))
        ) {
          atividades_aux.add(atividade);
          atividade.calculaCalorias(utilizador);
          calorias += atividade.getGastoCalorico();
          counter = 1;
        }
```

Figura 23: Geração inteligente de Planos 2

Se após o preenchimento do plano com o número de atividades diárias pretendidas pelo utilizador, o mesmo ainda não tiver atingido o gasto calórico mínimo necessário, retiramos uma atividade random do plano para voltar a preenche-lo com outra.

```
if (
  !Atividades.checkHardActivitiesPlan(atividades_aux) ||
  (calorias < min_calorias && atividades_aux.size() == n_atividadesDiarias)
) {
  Random random = new Random();
  int index = random.nextInt(atividades_aux.size() - 1);
  calorias -= atividade.getGastoCalorico();
  atividades_aux.remove(index);
}
if (calorias > min_calorias) break;
```

Figura 24: Geração inteligente de Planos 3

Quando atingimos o gasto calórico mínimo desejado, acaba o ciclo e são colocadas todas as datas necessárias antes de retornar o plano para ser adicionado pelo *Controlador*.

```
    } while (calorias < min_calorias);
    this.setDataAtividades(data, atividades_aux);
    this.setData(data.toLocalDate());
    this.setTreino(atividades_aux);
    return this;
  }
```

Figura 25: Geração inteligente de Planos 4

A implementação desta funcionalidade foi sem dúvida a mais desafiante, foram necessárias várias funções auxiliares , e mesmo funcionando, o nosso grupo concorda que podia exister mais aleatoriedade na mesma.

5 Outros

5.1 Documentação

Ao longo do projeto, não conseguimos dedicar tanto tempo a esta vertente como pretendido. Mesmo assim, consideramos que o código está bastante legível com métodos nomeados apropriadamente.

5.2 Diagramas de classes

Ao longo do relatório foram apresentados vários diagramas de classes simplificados. O diagrama de classes, devidamente, completo está presente na mesma pasta que este relatório.

5.3 Gestão de Erros

Este programa é bastante sujeito a erros, especialmente de inputs graças à natureza dos scanners em *Java*.

Tendo isto em conta, decidimos gerir estes erros da maneira que achamos mais aproriada, de seguida temo um exemplo:

```
} catch (
  DateTimeParseException
  | NumberFormatException
  | ArrayIndexOutOfBoundsException e
) {
  System.err.println(
    "Erro ao fazer o parsing da atividade: " +
    e.getMessage() +
    "\n" +
    "Pressione uma tecla para prosseguir"
  );
  return null;
}
```

Figura 26: Parsing Exceptions

Aqui tentamos gerir as exceções no parsing do programa através da utilização de *try catch*;

6 Conclusão

Com o desenvolvimento deste projeto, foi possível consolidar os conceitos expostos nas aulas teóricas e praticados nas aulas práticas. Conseguimos, também, consolidar o nosso conhecimento neste novo paradigma e na linguagem Java. Foi possível colocar em prática conceitos como hierarquia, encapsulamento e modularidade, tendo estes sido muito úteis no desenvolvimento deste programa.

De uma forma geral, estamos bastante satisfeitos com o trabalho. Sabemos que existem aspetos a melhorar mas achamos que o trabalho foi bem conseguido, ajudando-nos a aprofundar o nosso conhecimento na àrea.