Universidade do Minho
Licenciatura em Engenharia Informática

Unidade Curricular de Programação Orientada aos Objetos

Ano Letivo de 2023/2024

ActFit!



Fernando Pires – A77399

Rafael Seara – A104094

Sara Silva – A104608

Maio, 2024

POO

Resumo

Este projeto foi desenvolvido no contexto da Unidade Curricular de Programação Orientada aos Objetos e visa criar uma aplicação para gerir atividades e planos de treino para praticantes de atividades físicas. A aplicação foi projetada para registar e monitorizar as atividades físicas realizadas pelos utilizadores, cada uma com características específicas, como distância percorrida, repetições, e uso de pesos, além de atividades classificadas como *Hard*.

O sistema permite a criação de utilizadores com diferentes perfis, desde profissionais a ocasionais, e regista detalhadamente os exercícios realizados, incluindo a distância percorrida, as séries e repetições realizadas bem como as calorias gastas. Essas métricas são calculadas usando fórmulas especificamente desenvolvidas para cada tipo de atividade e tipo de usuário, considerando também a dificuldade das atividades.

Adotamos o padrão arquitetónico *Model-View-Controller* (MVC) para estruturar a aplicação, o que facilitou a manutenção do código e a separação clara entre a interface do utilizador, a lógica de negócios e a representação dos dados. Esta escolha provou ser essencial para a gestão eficiente do projeto, permitindo uma implementação modular e a colaboração eficaz entre membros do grupo com diferentes responsabilidades.

Além de registar atividades, a aplicação suporta a geração de estatísticas detalhadas sobre o desempenho dos utilizadores, permitindo análises como o utilizador que mais calorias gastou, o que realizou mais atividades, entre outras métricas importantes para quem pratica exercícios físicos. Também foi implementada uma funcionalidade para simular o avanço do tempo, facilitando a simulação de um uso real da aplicação ao longo de diferentes períodos.

Por fim, a aplicação também está equipada para gerar planos de treino personalizados com base nos objetivos do utilizador, considerando o tipo de atividade desejada, a frequência das sessões e o consumo calórico alvo. Esses planos são adaptáveis e consideram restrições como a intensidade das atividades e a compatibilidade entre elas para garantir a eficácia do treino.

Índice

Índice de Figuras

1. Introdução

O presente relatório é referente ao projeto prático da Unidade Curricular de Programação Orientada a Objetos da Licenciatura em Engenharia Informática do Departamento da Informática da Escola de Engenharia da Universidade do Minho, para o ano de 2024.

Este projeto, desenvolvido em Java, consiste na implementação de uma aplicação de gestão de planos de treino, composta essencialmente por Utilizadores, Atividades e Planos de treino.

Entre os principais requisitos encontram-se **requisitos base de gestão das entidades**, como ter a capacidade de criar utilizadores, atividades e planos de treino, **efetuar estatísticas sobre o estado do programa**, como por exemplo o utilizador com mais calorias despendidas ou atividades feitas e a **criação de um plano de treino** de acordo com os objetivos de cada utilizador.

O projeto possui também uma funcionalidade para simular o avanço do tempo, facilitando a simulação de um uso real da aplicação ao longo de diferentes períodos.

Serão enunciadas as principais decisões tomadas pelo grupo na elaboração do projeto, analisando as suas consequências e eventuais vantagens e desvantagens, assim como será explicada em detalhe a arquitetura da aplicação desenvolvida.

2. Arquitetura

2.1. *Model-View-Controller*

O modelo *Model-View-Controller* (MVC) é um padrão de arquitetura de software amplamente utilizado que separa a aplicação em três componentes principais: o modelo (*Model*), a vista (*View*) e o controlador (*Controller*).

O Modelo representa a lógica de negócios e os dados, a Vista exibe os dados (o modelo) ao utilizador e pode enviar comandos ao controlador, e o Controlador atua como um intermediário entre o modelo e a vista, controlando o fluxo de dados entre eles e gerindo as interações do utilizador, transformando-as em ações a serem realizadas pelo modelo.

Optamos por utilizar o MVC neste projeto devido aos seus múltiplos benefícios. Primeiramente, o MVC promove uma organização clara e modular, facilitando a manutenção e a escalabilidade do código. Por separar a interface do utilizador da lógica de negócios, permite que os programadores (neste caso, os elementos do grupo) trabalhem de forma mais independente e simultânea. Além disso, esta separação torna a aplicação menos propensa a *bugs*, pois modificações numa parte do sistema geralmente não afetam as outras.

Com o MVC também é possível reutilizar componentes com maior facilidade, o que pode acelerar significativamente o desenvolvimento de novas funcionalidades ou alterações. Tudo isto contribui para um desenvolvimento mais ágil e eficiente, tornando o MVC uma escolha ideal para a estruturação deste projeto.

2.2. *Gradle*

Para facilitar o desenvolvimento do projeto, optamos por utilizar a ferramenta *Gradle* para auxiliar na compilação e gestão das dependências da aplicação.

Para compilar e rodar o projeto utilizam-se os comandos (a partir da diretoria base do projeto):

```
cd ActivityManager
./gradlew clean build
java -cp build/classes/java/main activity_manager.Main
```

3. Aplicação

3.1. Interface

A aplicação desenvolvida tem uma interface pelo terminal. A interação com esta é feita através de comandos.

Figura 1 – Interface Login

Os dados ou resultados são mostrados em formato de tabela quando mais do que um objeto é apresentado (Figura 1), ou uma lista de atributos para um objeto singular.

3.2. Funcionalidades

Os requisitos propostos no enunciado foram cumpridos. Assim, para os listar, a aplicação desenvolvida suporta:

- Capacidade de criar utilizadores, atividades e planos de treino bem como registar a execução de uma atividade por parte de um utilizador;
- Consulta de utilizadores, atividades e planos de treino registados;
- Avançar e recuar no tempo;
- Efetuar estatísticas relativamente ao estado da aplicação;
- Guardar e carregar uma simulação a partir de ficheiros;

- Atividades classificadas como *Hard*, bem como criação das mesmas;
- Gerar um plano de treino de acordo com os objetivos.

No que às estatísticas da aplicação diz respeito, estas podem ser feitas utilizando os seguintes comandos:

Figura 2 – Menu de Estatísticas

1. *User with most calories spent* – devolve o utilizador que mais calorias despendeu desde o início da simulação bem como o número de calorias;
2. *User with most activities done* – devolve o utilizador que mais atividades realizou desde o início da simulação bem como o número de atividades;
3. *Type of Activity performed the most* – devolve o tipo de atividade que mais vezes foi realizada desde o início da simulação;
4. *Most Kms done by User* – ao passar o email de um utilizador devolve o número de quilómetros que este realizou desde o início da simulação;
5. *Most Meters climbed by User* – ao passar o email de um utilizador devolve o número de metros que este subiu desde o início da simulação;
6. *Most demanding Workout by calories spent* – devolve o plano de treino em registo que é mais exigente em função das calorias gastas.

4. Modelo

4.1. Utilizadores

A aplicação possui três diferentes tipos de utilizadores: **profissionais**, **amadores** e **ocasionais**. Todos são instâncias de uma classe abstrata chamada **User** e todos possuem os mesmos atributos:

```
public abstract class User implements Serializable {  3 inheritors
    private final UUID id;  3 usages
    private String name;  4 usages
    private String email;  5 usages
    private Address residence;  4 usages
    private int avgHeartRate;  4 usages
    private double calories;  4 usages
    private int numberOfActivities;  4 usages
    private WorkoutPlan workoutPlan;  14 usages
    private List<Activity> activities;  10 usages
    protected double caloriesFactor;  5 usages
```

Figura 3 – Atributos da classe User

A principal característica que diferencia estes tipos de utilizadores é o atributo *caloriesFactor*. Optamos por atribuir valores fixos a cada tipo de utilizador para depois os utilizar na fórmula de cálculo de calorias despendidas. Assim temos:

- **Profissionais:** 1
- **Amadores:** 0.75
- **Ocasionais:** 0.5

Deixamos como exemplo o método construtor de um dos utilizadores:

```
public class UserPro extends User{  2 usages  Fernando Pires +1
    public UserPro(String name, String email, Address residence, int avgHeartRate) {  1 usage
        super(name, email, residence, avgHeartRate,  caloriesFactor: 1);
    }
```

Figura 4 – Método construtor utilizador profissional

4.2. Atividades

A nossa aplicação possui uma classe "mãe" abstrata chamada **Activity**, que possui todos os atributos requeridos pelo enunciado do projeto:

```
public abstract class Activity implements Serializable {  8 inheritors
    private final UUID id;  3 usages
    private String name;  4 usages
    private boolean isHard;  4 usages
    protected double calorieFactor;  4 usages
```

Figura 5 – Atributos da classe Activity

Esta classe tem como instâncias quatro tipos principais de atividades: **distância e altimetria**, **distância**, **repetições** e **repetições com pesos**.

O atributo *isHard* indica se a atividade é considerada uma atividade difícil ou não e o atributo *calorieFactor* define um fator calórico de cada tipo de atividade, usado para o cálculo de calorias despendidas em cada atividade.

Cada uma destas classes possui também alguns atributos próprios. Entraremos em detalhe sobre eles mais à frente.

Sentimos que existe a necessidade de elaborar cada uma destas classes com mais pormenor. Posto isto:

- **Distância e altimetria**

```
public class DistanceAndAltimeter extends Activity {  16 usages  Fernando
    private int distanceInMeters;  6 usages
    private int altimeterInMeters;  6 usages
    private double paceInKmH;  6 usages


    public DistanceAndAltimeter(String name, boolean isHard) {  1 usage
        super(name, isHard,  calorieFactor: 3);
        this.distanceInMeters = 0;
        this.altimeterInMeters = 0;
        this.paceInKmH = 0;
    }
```

Figura 6 – Atributos da classe DistanceAndAltimeter

Este tipo de atividade é considerado um dos mais difíceis devido à carga de distância e de altimetria tendo um *calorieFactor* de 3.

Possui três atributos próprios essenciais que a identificam: a distância (em metros) que percorreu, a altura (em metros) que subiu e o ritmo a que foi realizada a atividade. Todos estes atributos são utilizados para o cálculo de calorias gastas durante a atividade.

A fórmula utilizada para o cálculo é a seguinte:

```
@Override  3 usages  Fernando Pires *
public double getCaloriesSpent() {
    return ((double) distanceInMeters /1000) *  altimeterInMeters * paceInKmH * getCalorieFactor() * 0.025;
}
```

Figura 7 – Fórmula de cálculo de calorias da atividade distância e altimetria

- **Distância**

```
public class Distance extends Activity{  10 usages  Fernando
    private int distanceInMeters;  6 usages
    private double paceInKmH;  6 usages

    public Distance(String name, boolean isHard) {  1 usage
        super(name, isHard,  calorieFactor: 2);
        this.distanceInMeters = 0;
        this.paceInKmH = 0;
    }
```

Figura 8 – Atributos da classe Distance

Este tipo de atividade é considerado de dificuldade moderada pois apenas percorre uma distância tendo um *calorieFactor* de 2.

Possui dois atributos próprios essenciais que a identificam: a distância (em metros) que percorreu e o ritmo a que foi realizada a atividade. Todos estes atributos são utilizados para o cálculo de calorias gastas durante a atividade.

A fórmula utilizada para o cálculo é a seguinte:

```java
@Override  3 usages  Fernando Pires
public double getCaloriesSpent() {
    return ((double) distanceInMeters /1000) * paceInKmH * getCalorieFactor();
}
```

Figura 9 - Fórmula de cálculo de calorias da atividade distância

- **Repetições**

```java
public class Repetitions extends Activity {  3 usages  Fernando
    private int sets;  6 usages
    private int repetitions;  6 usages

    public Repetitions(String name, boolean isHard) {  1 usage
        super(name, isHard,  calorieFactor: 1);
        this.sets = 0;
        this.repetitions = 0;
    }
```

Figura 10 – Atributos da classe Repetitions

Este tipo de atividade é considerado de dificuldade reduzida pois apenas faz séries de repetições utilizando o peso corporal tendo um *calorieFactor* de 1.

Possui dois atributos próprios essenciais que a identificam: as séries que fez e as repetições efetuadas durante a atividade. Todos estes atributos são utilizados para o cálculo de calorias gastas durante a atividade.

A fórmula utilizada para o cálculo é a seguinte:

```java
@Override  3 usages  Fernando Pires
public double getCaloriesSpent() {
    return sets * repetitions * 0.05;
}
```

Figura 11 - Fórmula de cálculo de calorias da atividade repetitions

- **Repetições com pesos**

```java
public abstract class RepetitionsWithWeights extends Activity {  5 usages  4 inheritors
    public RepetitionsWithWeights(String name, boolean isHard) {  8 usages  Fernando
        super(name, isHard,  calorieFactor: 3);
    }
```

Figura 12 – Declaração da classe RepetitionsWithWeights

Este tipo de atividade é considerado de dificuldade elevada pois faz séries de repetições utilizando peso externo tendo um *calorieFactor* de 3.

Possui quatro tipos de atividade “filhos”: **peito**, **costas**, **braços**, **pernas**. Estes tipos de exercícios implementam uma interface ***WeightedActivity***.

Figura 13 – Interface WeightedActivity

Deixando, a título de exemplo, uma das classes "filho" vemos que possuem dois atributos próprios essenciais que as identificam: as séries que fez e as repetições efetuadas durante a atividade bem como o peso externo usado para o exercício. Todos estes atributos são utilizados para o cálculo de calorias gastas durante a atividade.

Figura 14 – Atributos da classe Chest

A fórmula utilizada para o cálculo é a seguinte:

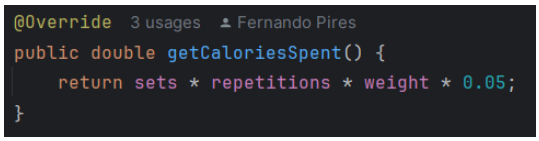

Figura 15 – Fórmula de cálculo de calorias da atividade chest

4.3. Planos de treino

O modelo de planos de treino é composto por duas classes principais: ***WorkoutPlan*** e ***DailyPlan***.

- ***WorkoutPlan***

Esta classe representa um plano de treino específico. Cada plano de treino possui um identificador único (UUID), um nome, uma data de início e uma data de fim. Além disso, um plano de treino contém uma lista de *DailyPlan* (Plano Diário), que representa as atividades planeadas para cada dia do treino.

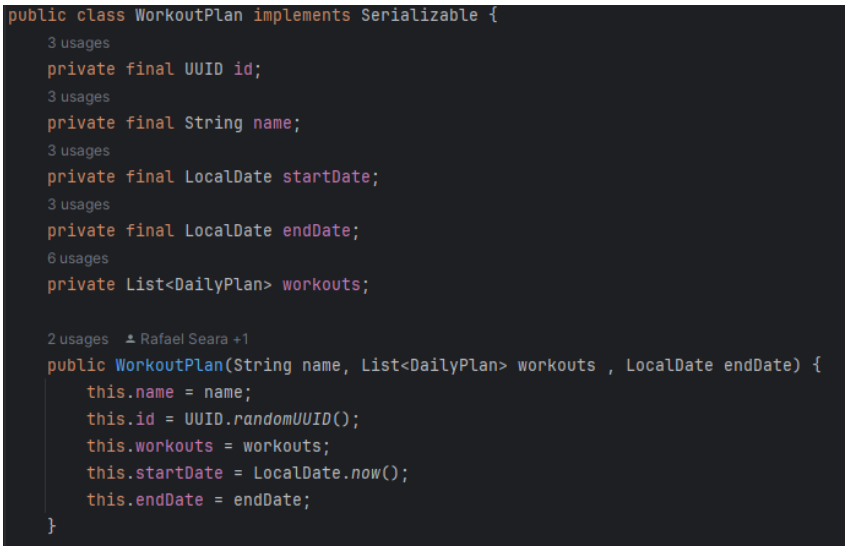

Figura 16 – Atributos da classe WorkoutPlan

- ***DailyPlan***

Esta classe representa o plano de atividades para um dia específico dentro de um plano de treino. Cada plano diário possui um identificador único, um dia da semana (*DayOfWeek*) e uma lista de atividades planeadas para esse dia.

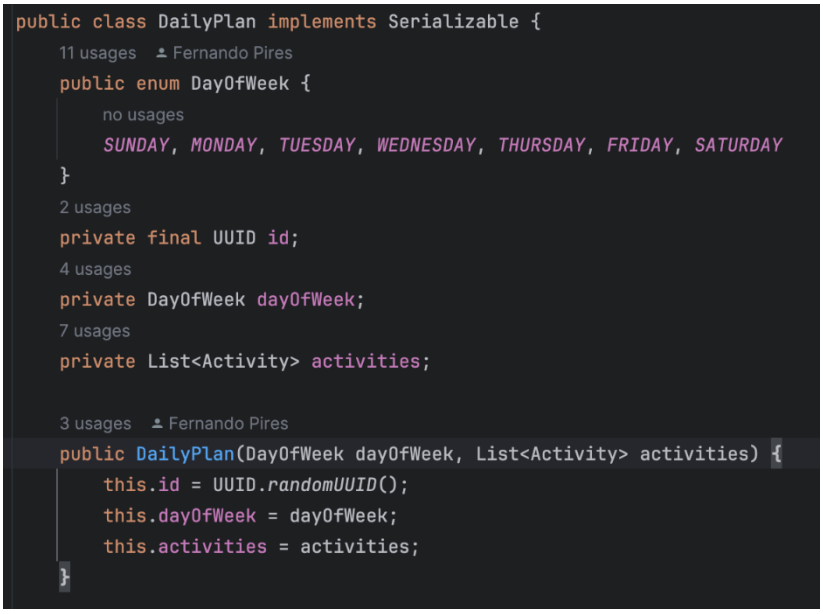

Figura 17 – Atributos da classe DailyPlan

4.4. Ficheiros de estado da aplicação

No desenvolvimento de sistemas que requerem armazenamento e recuperação de estado, como a nossa aplicação, a persistência de dados é um aspeto crítico. A interface *Serializable* do Java fornece a capacidade necessária para serializar objetos - convertê-los numa sequência de bytes - o que permite que sejam facilmente gravados em ficheiros. Isto é particularmente útil para garantir a integridade dos dados em casos de falhas do sistema ou para permitir a interoperabilidade entre diferentes componentes que operam os dados.

A interface *Serializable* indica que a classe pode ser serializada. Para que isso aconteça, é necessário que a classe implemente a interface *Serializable* e seus atributos também sejam serializáveis. Todos os utilizadores, atividades e planos de treino implementam esta interface pois é essencial podermos gravar em ficheiro os seus dados.

A serialização de todas estas entidades permite que o sistema funcione de maneira robusta, confiável e segura, protegendo contra perdas de dados e facilitando a manutenção e escalabilidade da aplicação.

5. Vistas

As Vistas desempenham um papel crucial na arquitetura de um programa, especialmente quando seguimos o padrão MVC (*Model-View-Controller*).

As Vistas são responsáveis por apresentar os dados ao utilizador de uma forma compreensível e interativa. Elas desempenham várias funções essenciais, incluindo:

- **Apresentação dos Dados**

As Vistas têm a responsabilidade de apresentar os dados fornecidos pelo Modelo ao utilizador final. Por exemplo, na nossa aplicação, uma vista pode exibir as atividades disponíveis para utilização.

```
public static void listActivities(Map<String, List<String>> activitiesByType) {
    int pageNumber = 1, option = 1;
    while (option > 0) {
        Utils.clearScreen();
        Utils.printHeader();
        Utils.println("\nList of Activities:\n");

        // Display activities for each type
        for (Map.Entry<String, List<String>> entry : activitiesByType.entrySet()) {
            String type = entry.getKey();
            List<String> activities = entry.getValue();

            Utils.println("==================================");
            Utils.println(type);

            for (String activity : activities) {
                Utils.println(activity);
            }
        }

        Utils.println("\n[1] <-");
        Utils.println("[2] ->");
        Utils.println("[3] Back");

        option = Utils.inputOption( options: 3);

        if (option == 1 && pageNumber > 1) {
            pageNumber--;
        } else if (option == 2) {
            pageNumber++;
        } else if (option == 3) {
            break;
        }
    }
}
```

Figura 18 – Método para listar atividades

- **Interatividade**

Também foram utilizadas para a interação do utilizador com o sistema, recebendo entradas e respondendo a elas de forma adequada.

- **Contribuição para a Modularidade e Flexibilidade**

Uma das principais vantagens desta abordagem é a contribuição para a modularidade e flexibilidade do sistema. Ao separar a lógica de apresentação dos dados do resto do sistema, as Vistas permitem que diferentes interfaces de utilizador sejam facilmente desenvolvidas e modificadas sem afetar o resto do código. Por exemplo, a vista activityAddSuccessfuly é usada modelarmente por mais do que um controlador.

```
public static void activityAddSuccessfully() {
    Utils.println("\nActivity add Successfully.");
    Utils.input("Press enter to go back to the Admin menu");
}
```

Figura 19– Método para indicar que uma atividade foi adicionada

As Vistas têm um papel crucial na arquitetura da nossa aplicação, permitindo apresentar os dados de forma eficiente e interativa ao utilizador final. A sua clara separação de responsabilidades, conforme preconizado pelo padrão MVC, promove a modularidade, flexibilidade e manutenção do sistema. Ao compreender a importância das Vistas e das suas responsabilidades, conseguimos desenvolver interfaces robustas e intuitivas, melhorando a experiência do utilizador e facilitando a evolução do sistema ao longo do tempo.

6. Controladores

Os controladores desenvolvidos serviram para ligar ambas as interfaces desenvolvidas ao modelo lógico e aplicacional. Foram desenvolvidos vários controladores distintos, um para cada entidade relevante. Assim sendo, os controladores expõem a funcionalidade da aplicação às vistas.

A aplicação inicia com o controlador de menu que, após receber a opção do utilizador, define para que controlador irá ser levada a aplicação.

```
public class ControllerMenu {  7 usages  Fernando Pires
    public static void menu() {  Fernando Pires
        int option = ViewMenu.menu();

        switch (option) {
            case 1:
                ControllerAuthentication.login();
                break;
            case 2:
                ControllerAuthentication.signUp();
                break;
            case 3:
                ControllerAdmin.menu();
                break;
            case 4:
                System.exit( status: 0);
        }
    }
}
```

Figura 20 – Controlador Menu

Após esta interação inicial, conforme as opções do utilizador, outros controladores são chamados a executar a lógica que se lhes pede.

7. Testes Unitários

Com o propósito de assegurar o correto funcionamento da aplicação, com ênfase no seu modelo, testes unitários em paralelo com o desenvolvimento do código do projeto são de extrema importância. O principal objetivo destes testes é garantir que a aplicação corre e executa de maneira correta e expectável.

Infelizmente não fomos capazes de os aplicar na nossa aplicação, essencialmente por motivos de tempo.

Tal como abordado nas aulas práticas, iria ser utilizada a biblioteca *JUnit* para o desenvolvimento dos testes unitários. Estes encontrar-se-iam na diretoria `test`.

8. Conclusão

O desenvolvimento deste projeto de gestão de atividades e planos de treino em Java representou uma experiência enriquecedora e desafiadora para todos os elementos do grupo. Ao longo deste percurso, a contribuição de cada membro do grupo foi fundamental, com todos demonstrando comprometimento, colaboração e uma excelente atitude. Tudo isto permitiu-nos superar os desafios encontrados e contribuir de maneira significativa para o sucesso do projeto.

Um dos desafios mais significativos foi projetar e implementar uma aplicação que não apenas atendesse às necessidades imediatas de funcionalidade, mas que também fosse bem estruturada para permitir fácil manutenção e escalabilidade futura. A adoção do modelo *Model-View-Controller* (MVC) foi uma decisão estratégica que se provou acertada, permitindo-nos separar as responsabilidades de forma clara, facilitando tanto a construção quanto a expansão do sistema. Isto assegurou que a aplicação possa crescer e evoluir sem comprometer a base de código existente, facilitando a adição de novas funcionalidades conforme a necessidade.

Trabalhar com Java e o paradigma de programação orientada a objetos ofereceu uma plataforma robusta para a implementação do nosso sistema. Através deste paradigma, aprendemos a pensar em termos de objetos e as suas interações, o que nos ajudou a modelar o sistema de forma mais intuitiva e eficiente. A orientação a objetos não apenas facilitou a representação de entidades complexas, como utilizadores e atividades, mas também nos ensinou importantes princípios de design de software, como encapsulamento, herança e polimorfismo.

A implementação da serialização para persistência de dados foi outra área de grande aprendizagem. Compreendemos a importância de garantir que os dados dos utilizadores não se perdessem entre sessões e como o Java facilita este processo com sua interface *Serializable*, permitindo-nos guardar e recuperar estados complexos da aplicação com facilidade.

Em resumo, este projeto não só cumpriu com os seus objetivos técnicos, proporcionando uma ferramenta útil e eficaz para a gestão de treinos e atividades físicas, como também foi uma jornada valiosa de crescimento pessoal e profissional para todos os envolvidos. As habilidades e conhecimentos adquiridos durante este projeto serão certamente ativos valiosos nas nossas futuras iniciativas académicas e profissionais.

9. Anexos

9.1. Diagrama de Classes