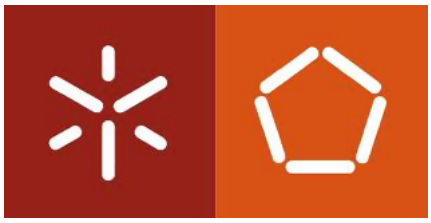
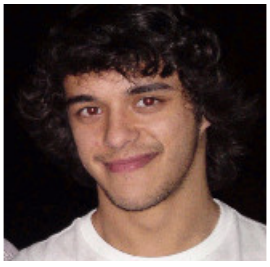


Escola de Engenharia
Departamento de Informática

Licenciatura em Engenharia Informática

Projecto Java - FitnessUM

Programação Orientada aos Objectos



69303
Bruno Pereira

66822
Miguel Guimarães

69854
João Mano

Braga, Junho de 2014

Conteúdo

1 Estrutura da aplicação

1.1 Actividades

Foram definidas as seguindes actividades desportivas para a nossa aplicação: zz

- Yoga
- Aerobics
- Swimming
- IndoorCycling
- Handball
- Basketball
- TableTennis
- Boxing
- Badminton
- VolleyBallIndoor
- Football
- VolleyBallBeach
- Running
- Skating
- Saling
- Walking
- Tennis
- Skiing
- Cycling
- MountainBiking
- Orienteering
- Snowboarding
- Polo

Para a implementação destas actividades foi usada a seguinte estrutura:

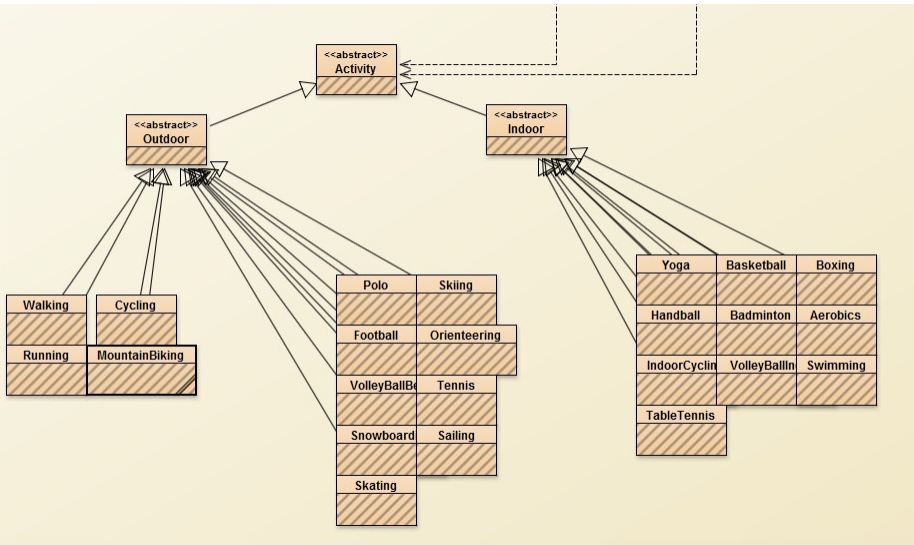


Figura 1: Estrutura das actividades

1.1.1 Classe abstracta Activity

Esta é a classe mais abstracta que contem o conceito de actividade. Contém variáveis comuns a todas as actividades:

- *String name*, nome da actividade criada.

- *GregorianCalendar date*, data de quando se realizou a actividade.
- *double timeSpent*, tempo gasto na actividade.
- *double calories*, campo preenchido pela aplicaçao de uma fórmula.

tal como os construtores, *getters* e *setters*.

1.1.2 Indoor,Outdoor e actividades desportivas

Todas as actividades desportivas tem um aspecto importante,o clima caso sejam praticadas ao ar livre. Devido a este aspecto foram criadas duas classes abstractas,subclasses de *Activity*,para essa distinção.

- Outdoor,contém a variável: *String weather*
- Indoor

Todas as actividades desportivas são subclasses de *Indoor* ou *Outdoor* como exemplicado na figura 1.

1.1.3 Comparadores e Interfaces

Para organizar as actividades criaram-se dois tipo de comparadores:

- CompareActivity- Compara a actividade pela data da realização da mesma.
- CompareActivityByTime- Compara a actividade pelo tempo gasto na realização desta.

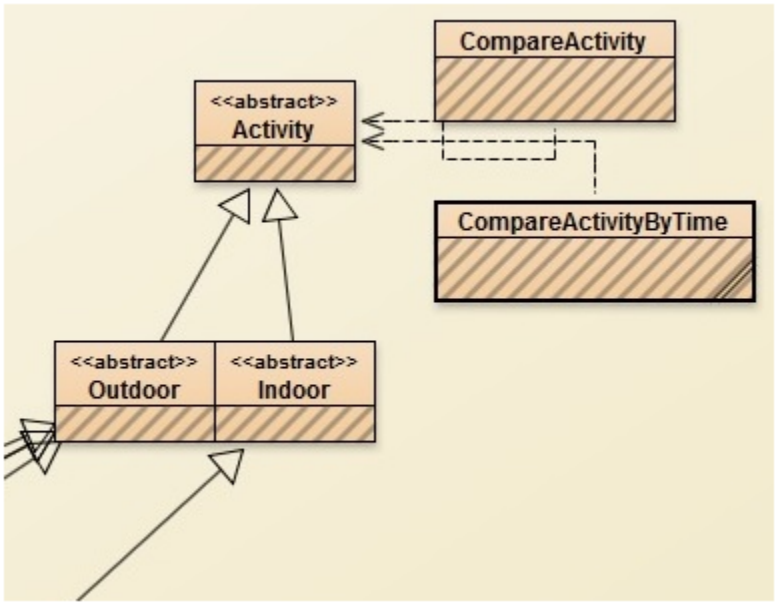


Figura 2: Comparador Activity

Depois de uma análise às actividades desportivas, ficou claro que para certas actividades se deviam registar distancias e para outras registar pontuações,neste seguimento foram criadas as seguintes interfaces:

- UserVs-Interface de métodos relacionados com pontos(pontos próprios e pontos do adversário)
- Distance -Interface de métodos relacionados com actividades de distancia.
- VerticalDistance- Interface de métodos relacionados com actividades de distancia vertical

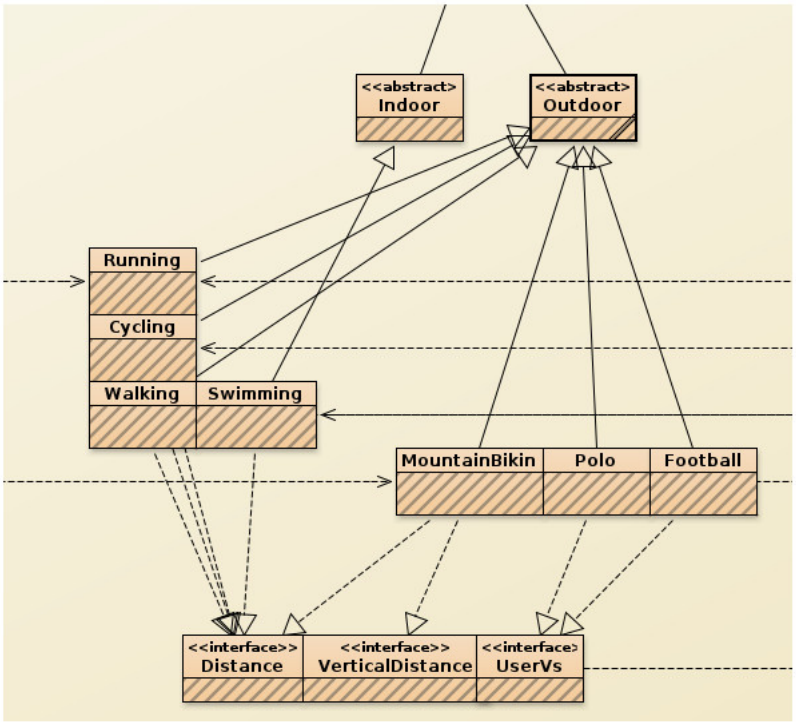


Figura 3: Exemplo de algumas actividades que implementam as interfaces

1.2 Utilizadores

Para distinguir utilizadores regulares de administradores criou-se a seguinte estrutura:

1.2.1 Classe abstracta Person

Classe geral para todo tipo de utilizador. As suas variáveis são:

- *String email*;
- *String password*;
- *String name*;
- *char gender*;
- *GregorianCalendar dateOfBirth*;

1.2.2 Classes User e Admin

As subclasses de Person referem-se a dois possíveis tipos de utilizador, utilizador normal ou utilizador com privilégios de administrador.
A classe Admin não tem métodos ou variáveis adicionais, visto que este tipo de utilizador apenas opera sobre a base de dados da aplicação.
A classe User adiciona as seguintes variáveis:

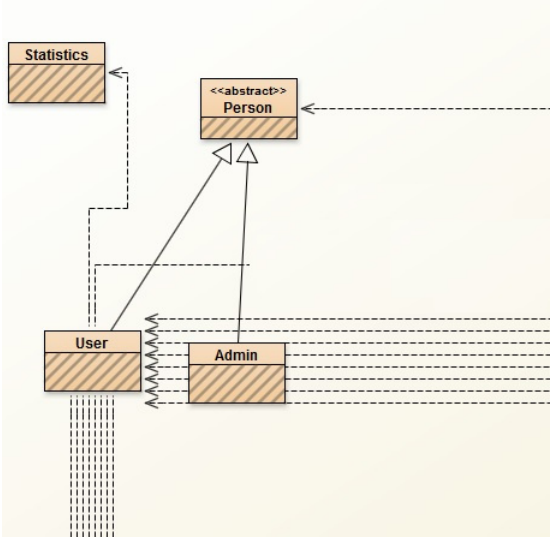


Figura 4: Estrutura das classes User e Admin

- *int height*;
- *double weight*;
- *String favoriteActivity*;
- *TreeSet<Activity> userActivities* - Actividades realizadas pelo utilizador;
- *TreeSet<String> friendsList* - Lista dos amigos do utilizador;
- *TreeMap<String, ListRecords> records* - Lista dos seus recordes pessoais;
- *TreeSet<String messageFriend* - Lista de pedidos de amizade;

Respectivos métodos *getters* e *setters*, construtores e métodos auxiliares para a gestão de amigos/pedidos de amizade, recordes pessoais, das suas actividades e estatísticas relevantes. Ainda contém funções auxiliares para a simulação de eventos.

1.2.3 Comparators

O tipo Person tem apenas um comparator:

- ComparePersonByName - que ordena por ordem alfabética do seu nome.

1.2.4 Statistics

A classe Statistics é usada para mostrar ao utilizador dados relevantes das suas actividades, estes podem ser descriminados por um dado mês ou por um ano. As suas variáveis são:

- *double timeSpend*;
- *double calories*;
- *double distance*;

contém os respectivos métodos *getters* e *setters* e construtores.

1.3 Recordes Pessoais

Para registar os recordes chegou-se a estrutura da fig **??**:

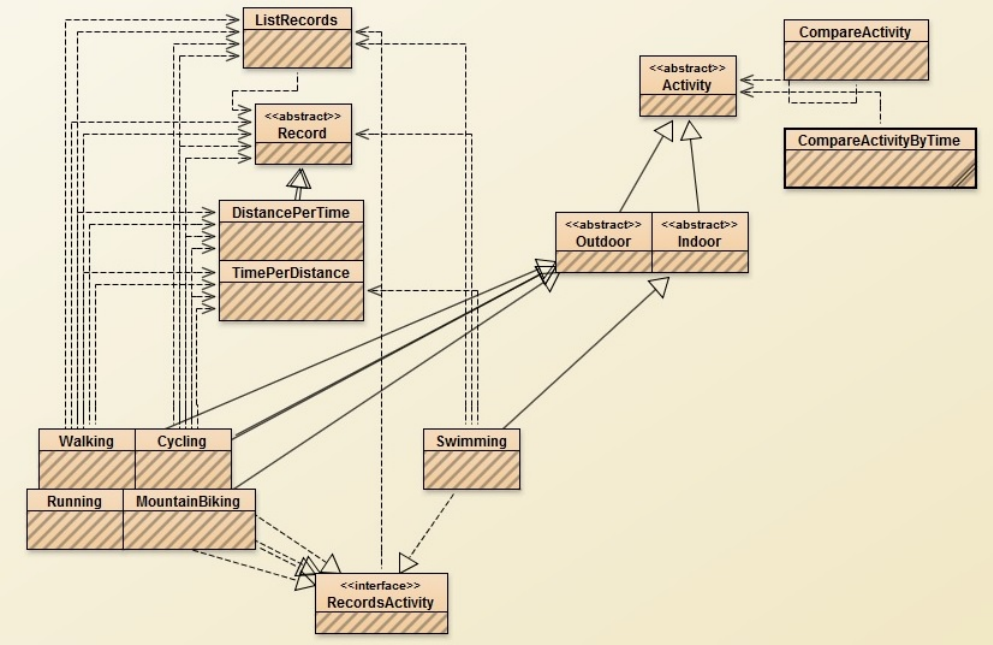


Figura 5: Estrutura dos recordes

Como se pode verificar na figura **??**, apenas as seguintes actividades contêm recordes:

- Running
- Cycling
- Walking
- MountainBiking
- Swimming

1.3.1 Classe abstracta Record

Esta classe representa todos os recordes que o utilizador pode bater. Contém apenas uma variável:

- *String name*-Nome do tipo de recorde a bater(ex: 1km,10 miles,Cooper...)

métodos construtores, *getName()* e *isEmpty()* que verifica se esse recorde existe ou não.

1.3.2 DistancePerTime e TimePerDistance

Estas classes simbolizam os dois diferentes tipos de recordes.

DistancePerTime é um recorde em que o objectivo é fazer a maior distância para um dado tempo. As suas variáveis são:

- *double recordTime* - Tempo do recorde;

- *double distance* - Distância registada;

Enquanto que TimePerDistance representa um recorde de menor tempo para uma certa distância. As suas variaveis sao:

- *double recordDistance* - Distância do recorde;
- *double time* - Tempo registado;

Estas duas classes têm os mesmos métodos, no entanto os métodos *update* e *setStatistics*, estão implementados de maneiras diferentes, tendo em conta que em DistancePerTime, quanto maior a distância melhor é o recorde, e no caso do TimePerDistance, o melhor recorde é o de menor tempo.

1.3.3 ListRecords

Classe que agrupa todos os recordes de uma actividade. Tem como variáveis:

- *String name* - Aqui o nome simboliza o tipo de actividade (Ex: Running, Walking...);
- *ArrayList<Record> recs* - Lista dos recordes;

Tem implementado métodos construtores, *getters*, *setters* e ainda um método *updateList()* que aplica a função *update()* a todos os objectos *Record* da lista. (Subsitui na lista original caso recorde da segunda lista seja melhor).

1.3.4 Interfaces

Nesta fase,visto que nem todas as actividades desportivas implementarem recordes, chegou-se então à conclusão que estas actividades precisam sempre de devolver a lista de recordes registados,então implementou-se a seguinte interface:

- *RecordsActivity*;

Que contém o seguinte método:

- *getListRecords*;

1.4 Fórmulas

Em certos momentos do trabalho surgiu a necessidade de codificar fórmulas.
Tal aconteceu para calcular as calorias gastas em cada actividade e para a simulação dos eventos.

1.4.1 Fórmula das Calorias

MET(Metabolic Equivalent of Task)- É uma medida fisiológica que expressa o custo energético de cada actividade física.

Sabendo o que MET's representa, e retirando essa medida, para cada actividade ,pelo seguinte quadro: `http://www.cdof.com.br/MET_compendium.pdf`
Criou-se a seguinte fórmula das calorias para cada actividade:

$$\textbf{Calorias} = mets * pesoDoUtilizador * (tempoGasto(min)/60)$$

1.4.2 Fórmula para a Simulação

Para inferir um valor médio de minutos/km foi seguido o seguinte raciocínio:

1. Calculou-se um tempo médio em função do evento.
2. Contou o número de actividades praticadas do tipo do evento(ex: evento-Marathon,tipo do evento-Running).
3. Aplicou-se a fórmula idealizada.

Para calcular o tempo médio em função do evento,fez-se uma distinção entre MarathonBTT e os outros eventos.

Para o evento **MarathonBTT** percorreu-se todas as actividades do tipo "MountainBiking"e para cada uma delas:

- Calculou-se um factor, de 0 a 1, em função do parâmetro VerticalDistance(quanto maior,maior é o factor).
- Somou-se a distancia com o acumulado da mesma.
- Somou-se o acumulado do tempo com o tempo gasto/ distancia feita.

No fim calcula-se o tempo médio total em função do tempo médio calculado com o factor médio e distancia média realizadas nas actividades.

Para os restantes eventos, a ideia foi a mesma,apenas não se usou o factor.

Tendo o tempo médio calculado(tm) a fórmula para o calculo do tempo médio final é a seguinte:

$$\boxed{\textbf{tempo} = tm + (1 * tabWeather(weather)) + (1 * tabTemp(temperatura)) - (n^o/100) + (age/100)}$$

tabWeather− > método que devolve um factor(de 0 a 1) em função dos seguintes climas:

- Sol
- Sol intenso
- Sol intenso com ventos fortes
- Chuva
- Chuva com ventos fortes
- Chuva intensa
- Chuva intensa com ventos fortes
- Trovoada
- Trovoada com ventos fortes
- Nublado

tabTemp− >método que devolve um factor (de 0 a 1) em função da temperatura, em graus celsius.
n^o − > número de actividades praticadas do tipo do evento.

Em **cada km** da simulação o usa-se este tempo calculado e multiplica-se por $(Math.random() + 0.5)$, factor aleatório que aumenta ou diminui o tempo no km do participante.

Para a possibilidade de um participante do evento **desistir num certo Km** foi usada a seguinte estratégia:

1. Calculou-se a idade do participante.
2. Calculou-se uma probabilidade.
3. Calculou-se o possível km em que desiste.

A probabilidade foi calculada em função da idade:

- age < 15- factor = Math.random() + 0.1;
- age < 20- factor = Math.random() + 0.5;
- age < 25- factor = Math.random() + 0.7;
- age < 30- factor = Math.random() + 0.85;
- age < 35- factor = Math.random() + 0.9;
- age < 40- factor = Math.random() + 0.6;
- age < 45- factor = Math.random() + 0.5;
- age < 50- factor = Math.random() + 0.4;
- age < 55- factor = Math.random() + 0.3;
- age < 60- factor = Math.random() + 0.2;
- age < 65- factor = Math.random() + 0.1;
- age >= 65- factor = Math.random() + 0.05;

Finalmente multiplica-se este factor pela totalidade de km do evento, e se este km for menor que a distancia total do evento,então será o km de desistência do participante.