Universidade do Minho

LEI 2ºAno 2ºSemestre

Programação Orientada aos Objectos - Linguagem JAVA

FitnessUM

Uma aplicação de registo e simulação de atividades fitness

Grupo 29

António Anjo a67660, J. Daniel Caldas a67691, José Francisco a67724

7 de Junho de 2014

Conteúdo

1 Introdução

Foi proposto ao grupo de trabalho que fosse desenvolvida uma **API Fitness** que consiste numa rede de utilizadores interligada (rede de amigos) que *registam/simulam actividades físicas/desportivas*, partilham resultados entre si, consultam as suas actividades, recordes pessoais e ainda mais funcionalidades como veremos ao longo deste relatório. Tentaremos sempre que possível ligar a **interface do utilizador** à explicação. Será implementada uma solução na linguagem **JAVA** seguindo-se os princípios da **Programação Orientada aos Objectos (POO)**. Neste projeto são explorados mecanismos muito poderosos do JAVA como *Hierarquia e Herança e composição de classes*, quanto a *POO* preocupamo-nos essencialmente com o **encapsulamento dos dados**. A nossa aplicação divide-se em essencialmente em 4 partes como demonstramos na figura em baixo.

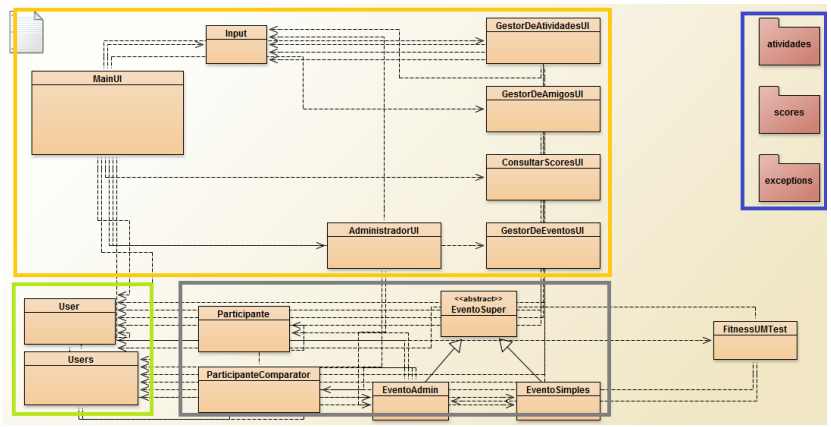


Figura 1: Na imagem podemos observar as diferentes classes que constituem a API estando agrupadas por funcionalidades no caso das actividades, scores e excepções constituem até um package, o que permite uma *organização do código e gestão do projeto muito eficientes.*

Na caixa a amarelo temos as classes que juntamente com a main formam a **Interface do Utilizador**. Na caixa a verde podemos observar a **"base de dados"** de todo o sistema, a classe **Users** *que basicamente é o coração da API*, esta é formada na sua essêcia por composião simples da classe User. Na caixa a cinzento encontram-se as classes que permitem que o administrador **crie eventos, convide utilizadores e execute simulações**... A azul temos os **packages** os quais exploraremos detalhadamente mais à frente. Resta fazer referêcia à classe *FitnessUMTest* que contém dados de teste consistentes, e à classe **Input** uma classe muito útil pois auxililia o tratamento de erros, esta é a única classe externa ao trabalho, ou seja que não é desenvolvida pelo grupo. Temos de demonstrar que é muito importante **codificar** assim como é muito importante **reutilizar**.

2 Classe User - Um utilizador

Na classe user para além dos campos básicos de informação como nome, email, credenciais de acesso, etc. ... o grupo adicionou à classe as variáveis de instância da figura, sendo estas as mais merecedoras da atenção pela maior complexidade.

```
private HashSet<String> friends;
private HashSet<String> requests;
private HashSet<String> sent;
private Atividades atividades;
private Score scores;
private HashMap<String,EventoSimples> convites;
```

2.0.1 Variáveis e métodos de instância

No que toca a variáveis de instância:

- **friends** contém todos os emails dos utilizadores que são amigos do user.
- **requests** contém todos os emails dos utilizadores que enviaram pedidos de amizade.
- **sent** contém todos os emails dos utilizadores a quem foram enviados pedidos de amizade.
- **atividades** contém todas as atividades do utilizador.
- **scores** contém todos os scores do utilizador.
- **convites** contém todos os convites para eventos enviados pelo admin ao utilizador.

De seguida descrevemos os métodos mais críticos desta classe:

```
public HashSet<String> getFriends()
-Método que devolve um set com os emails dos amigos

public HashSet<String> getRequests()
-Método que devolve um set com os os emails dos
utilizadores que enviaram pedidos de amizade

public HashSet<String> getSent()
-Método que devolve um set com os emails dos utilizadores a quem foram
enviados pedidos de amizade.
```

```
public boolean addFriend(String friend)
-Método que devolve um boleano e que adiciona um email ao conjunto de amigos.
Retorna false caso o email a adicionar já se encontre no conjunto
e true se não existir.

public boolean addRequest(String request)
-Método que devolve um boleano e que adiciona um email ao conjunto de
pedidos recebidos. Retorna false caso o email a adicionar
já se encontre no conjunto e true se não existir.

public boolean addSent(String sent)
-Método que devolve um boleano e que adiciona um email ao
conjunto de pedidos enviados. Retorna false caso o email a
adicionar já se encontre no conjunto e true se não existir.

public boolean removeFriend(String friend)
-Método que devolve um boleano e que remove um email do
conjunto de amigos. Retorna true caso o email a remover se
encontre no conjunto e false se não existir.

public boolean removeRequest(String request)
-Método que devolve um boleano e que remove um email do
conjunto de pedidos recebidos. Retorna true caso
o email a remover se encontre no conjunto e false se não existir.

public boolean removeSent(String sent)
-Método que devolve um boleano e que remove um email do
conjunto de pedidos enviados. Retorna true caso
o email a remover se encontre no conjunto e false se não existir.

public int getNrFriends()
-Método que devolve o número de amigos.

public int getNrRequests()
-Método que devolve o número de pedidos recebidos.

public boolean isFriend(String friend)
-Método que dado o email de um user permite
saber se este está na lista de amigos.

public boolean isRequested(String friend)
-Método que dado o email de um user permite saber se este
está na lista de pedidos recebidos.

public String friendsList()
-Método que devolve uma lista com os emails dos amigos.

public String requestsList()
-Método que devolve uma lista com os emails de utilizadores
```

que enviaram pedidos de amizade.

3 Classe Users - Os utilizadores

A classe possui duas variáveis de instância, **users** e **eventos**, em que a primeira consiste num **HashMap<String,User>**, que irá guardar todos os utilizadores da aplicação, e a segunda consiste num **HashMap<String,EventosAdmin>** que armazena todos os eventos criados pelo administrador.

3.0.2 Variáveis e métodos de instância

```
public Users()
public Users(HashMap<String,User> users, HashMap<String,EventoAdmin> ev)

Para além dos habituais métodos equals, clone e toString, a classe ainda possui
alguns métodos muito úteis como:

public void registerUser(String email, String password,
String nome,String genero, int altura,int peso,
GregorianCalendar datanasc,String fav)
- Método que permite registar utilizador passados os parâmetros
do seu registo nome, idade, altura etc. ...

public User login(String email, String password)
- Método que dado um email e uma password retorna o
utilizador associado no caso de os campos estarem corretos
(caso contrário retorna null)

public boolean containsUser(String email)
- Método que dado um email verifica se existe um utilizador associado

public void addFriend(User a, String email) throws UserNaoExisteException
AmigoExisteException, ProprioUserException, ConviteEnviadoException
- Método que envia um pedido do utilizador 'a' para o utilizador cujo email é
igual ao email passado ao método

public void rejectRequest(User a, String email) throws UserNaoExisteException,
NaoEnviouPedidoException, ProprioUserException
- Método que rejeita um pedido que o utilizador 'a' recebeu do utilizador
cujo email é igual ao email passado ao método

public void acceptFriend(User a, String email) throws UserNaoExisteException,
NaoEnviouPedidoException, ProprioUserException
- Método que aceita um pedido que o utilizador 'a' recebeu do utilizador
cujo email é igual ao email passado ao método

public void removeFriend(User a, String email) throws UserNaoExisteException,
```

```
NaoAmigoException, ProprioUserException
- Método que remove da lista de amigos do utilizador 'a' o utilizador
cujo email é igual ao email passado ao método

public User getFriend(User user, String friend) throws UserNaoExisteException,
NaoAmigoException, ProprioUserException
- Método que vai retorna um user cujo o email seja igual
à string 'friend' caso 'user' seja amigo dele

public String findUser(String nome)
- Método que dado um nome de um utilizador retorna o seu email
```

Todos os métodos relacionados com atividades, scores e eventos serão abordados numa parte mais avançada do relatório.

3.0.3 Simulação de envio e pedido de amizade

Como foi anteriormente mencionado a classe users é o **coração** desta API pois, é por essa mesma classe users que as diferentes classes comunicam com a Interface do Utilizador que por sua vez comunica com o utilizador, isto tudo **sempre garantindo o encapsulamento dos dados**. Na seguinte imagem podemos ver como por exemplo funciona um pedido de amizade.

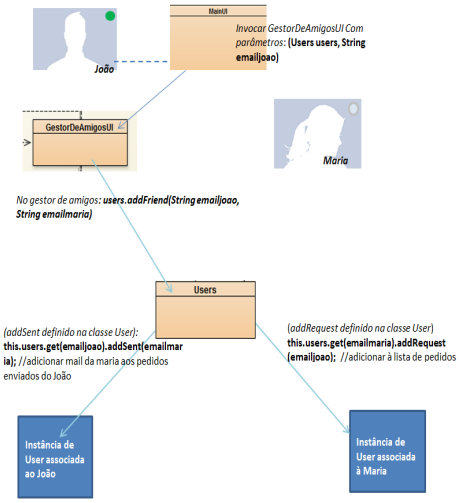


Figura 2: Cenário: João e Maria são instâncias da classe *User*, e vamos simular um cenário em que o João envia um pedido de amizade à Maria. A bolinha verde significa que o João está a utilizar a Interface do Utilizador, equanto que intuitivamente vemos que a Maria não está. Podemos melhor através do esquema do que olhando para o código, que **somente cada instância de utilizador atualiza a sua própria informação** em conformidade. **NOTA**:Na simulação considerarmos que ambos os utilizadores existem e que o pedido é *enviado* com sucesso (**excepções a ver mais à frente**).

No seguimento do exemplo anterior, vamos mostrar como funciona a **aceitação de um pedido de amizade** por parte de um utilizador, neste cenário em específico será *Maria* quem aceitará o pedido de amizade.

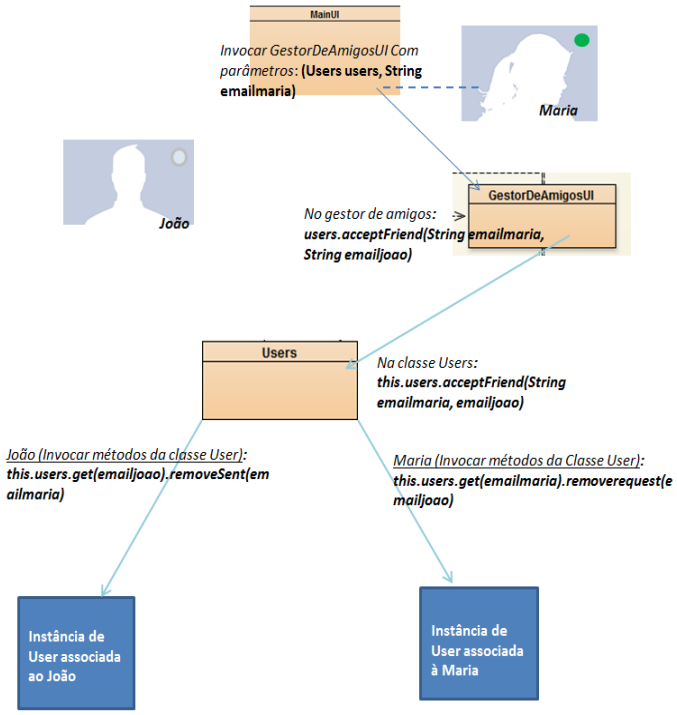


Figura 3: Cenário: Maria agora comunica com a interface do utilizador e aceita o pedido de amizade de João, podemos através da imagem ver todos os métodos invocados, e o contexto em que são invocados. **Nota**: Na simulação considerarmos que ambos os utilizadores existem e que o pedido é aceite com sucesso excepções a ver mais à frente).

4 Atividades

4.1 A Hierarquia de Classes das Atividades

Sendo esta uma aplicação *fitness* a mesma terá como suporte um conjunto de atividades pré-definidas pelo grupo de trabalho a que, cada utilizador terá acesso e poderá **fazer registo de uma *instância dessa atividade***. Antes de partirmos para as atividades em concreto, temos de explicar as mesmas num **contexto abstracto** i.e, as suas raízes numa hierarquia de classes extremamente versátil

da qual *podem eventualmente "ser estendidas"* ***outras atividades*** que não as pré-definidas pelo grupo.

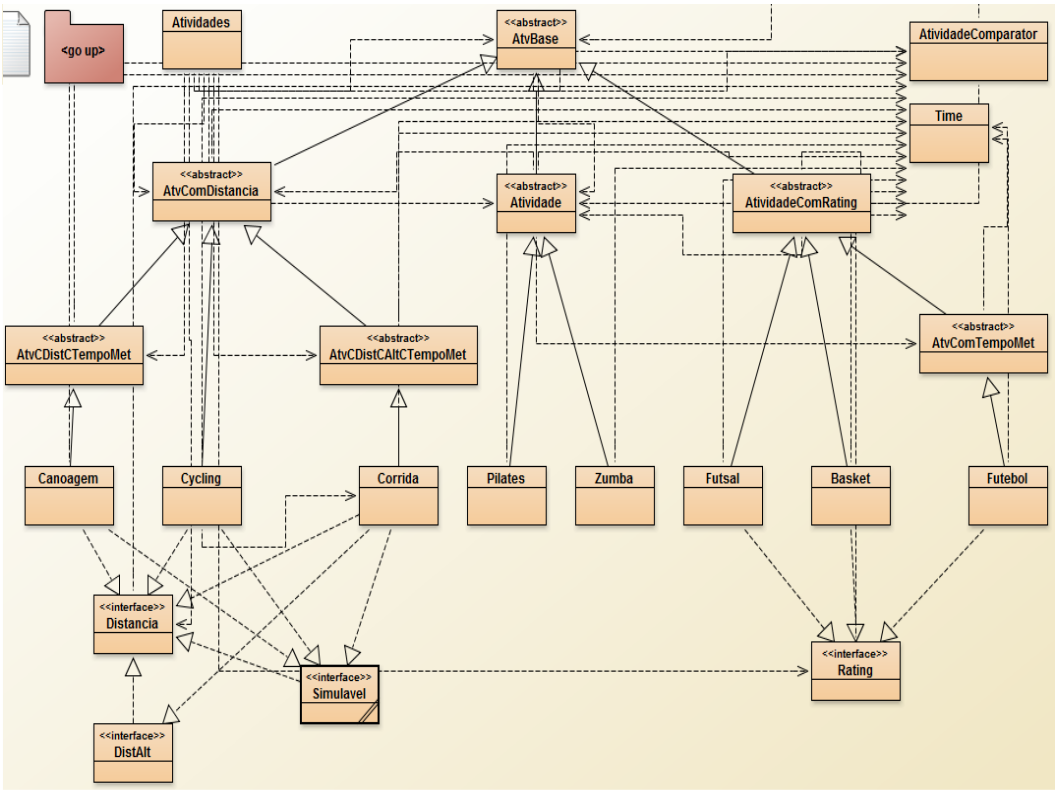


Figura 4: Hierarquia de Classe das atividades onde no topo da imagem se encontra o nível **mais abstracto** da hierarquia, à medida que *fazemos estender* essa superclasse **especializamos** as nossas atividades por fatores eliminatórios.

Como podemos ver na fig. 4 **AtvBase é a superclasse** onde se encontram as variáveis comuns a qualquer tipo de atividade, ora são elas: **Time** *duracao*, **int** *calorias*, **double** *hidratacao*, **GregorianCalendar** *data*. Inicialmente o grupo incluía também a variável **String** *nomeatividade*, o que é totalmente desnecessário uma vez que as Classes que instanciam atividades têm por nome o próprio nome da atividade, ficou então definido o método:

```
public abstract String getNome();
```

que cada classe de uma atividade implementa do seguinte modo:

```
public String getNome(){return this.getClass().getSimpleName();}
```

Para definirmos a hierarquia que demonstramos consideramos em primeiro plano os seguintes fatores: a presente atividade *comtemplar ou não* **distância**, **altimetria** e **tempo metereológico** (*Indoors/Outdoors*), ou nenhuma das anteriores - **Class Atividade** estendida por **Pilates** e **Zumba**.
Mas olhando bem para a hierarquia não parece que o referido seja o que a descreve na perfeição pois não? De facto **não**, **pois este package foi desenhado primeiro ao nível das *Classes* e segundo ao nível das *Interfaces***.

4.1.1 Uma Hierarquia extensível...

A partir da nossa hierarquia podemos agora muito facilmente introduzir mais atividades na nossa API, pois temos uma base de classes abstractas especializadas. Imaginemos que queríamos introduzir as Atividades **Hockey em Patins**, **Musculação** e **Triatlo**... Na fig.5 podemos observar o novo (velho) aspeto da hierarquia.

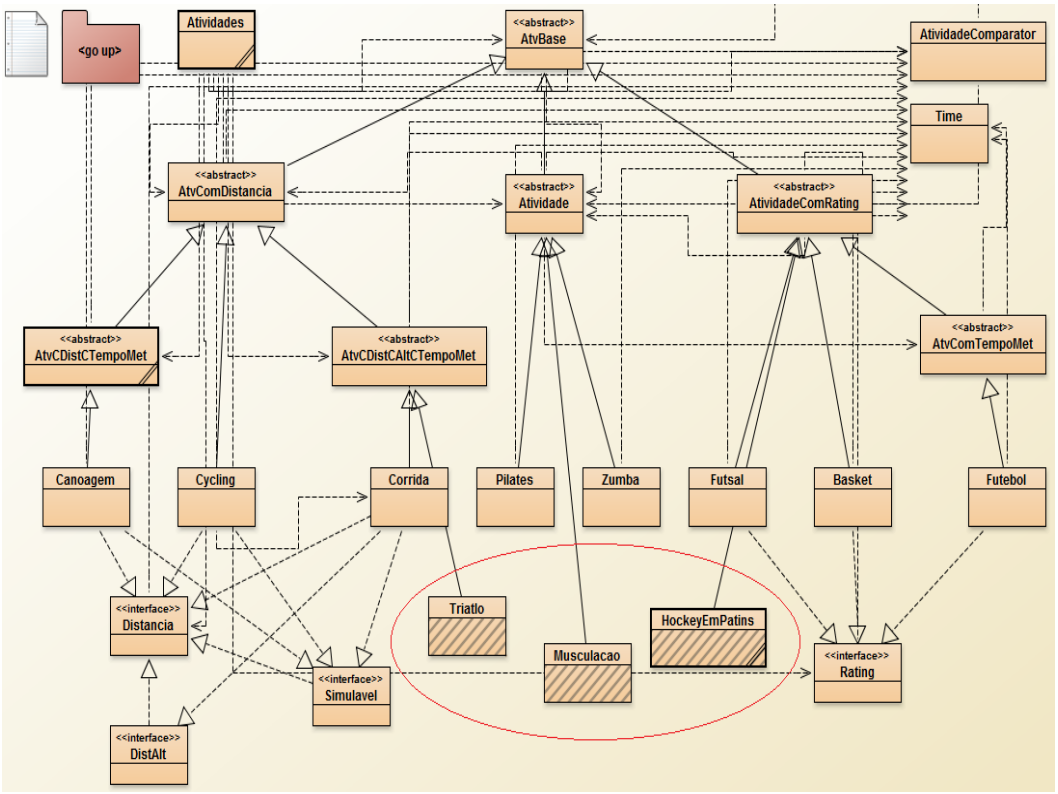


Figura 5: Hierarquia de Classe após acrescentadas as atividades HockeyEmPatins, Musculação e Triatlo.

Por certo que desenhar as "caixinhas"no IDE BlueJ não basta, teríamos de codificar cada classe implementando os habituais construtores e três métodos que a hierarquia força a implementar um deles já mencionado é o método

```
public String getNome()
```

outro é:

```
public NomeDaClasseAtivivdade clone();
```

e por último:

```
public int calcularCalorias(int idade, int altura, String genero,Time duracao, int peso)
```

Este último calcula as calorias gastas em função da atividade, mencionamos na bibliografia deste relatório alguns sites que o grupo consultou para implementar

este cálculo, limitamo-nos na maioria dos casos a fazer um cálculo baseado no **gasto de calorias a cada meia hora da atividade**, isto também em função do **género da pessoa**, da sua **idade** e do seu **peso**.

4.2 Estrutura de Dados e Polimorfismo

Construída toda uma hierarquia como a apresentada é possível agora usufruír do **polimorfismo**, i.e **podemos tratar todas as atividades da mesma maneira** pois todas **respodem a um conjunto de métodos comuns**, o mesmo é dizer que **os objetos têm agora praticamente o mesmo comportamento**! Criamos então a classe Atividades que agrupa os diferentes tipos de atividades unificando-os num só tipo.

```
public class Atividades implements Serializable{
//Variáveis de instância
private TreeSet<AtvBase> atividades;   //Set de atividades ordenadas
                                      por ordem cronológica (pelas datas de realização)
}
```

A estrutura de dados das atividades consiste num simples set de atividades contendo instâncias de várias atividades diferentes, estas são ordenadas por ordem cronológica, cujo algoritmo de ordenação está definido na Class **AtividadeComparator** (no caso de as datas serem iguais compara-as pelo nome da atividade ficando a ordem de 2º plano a ser a ordem alfabética).

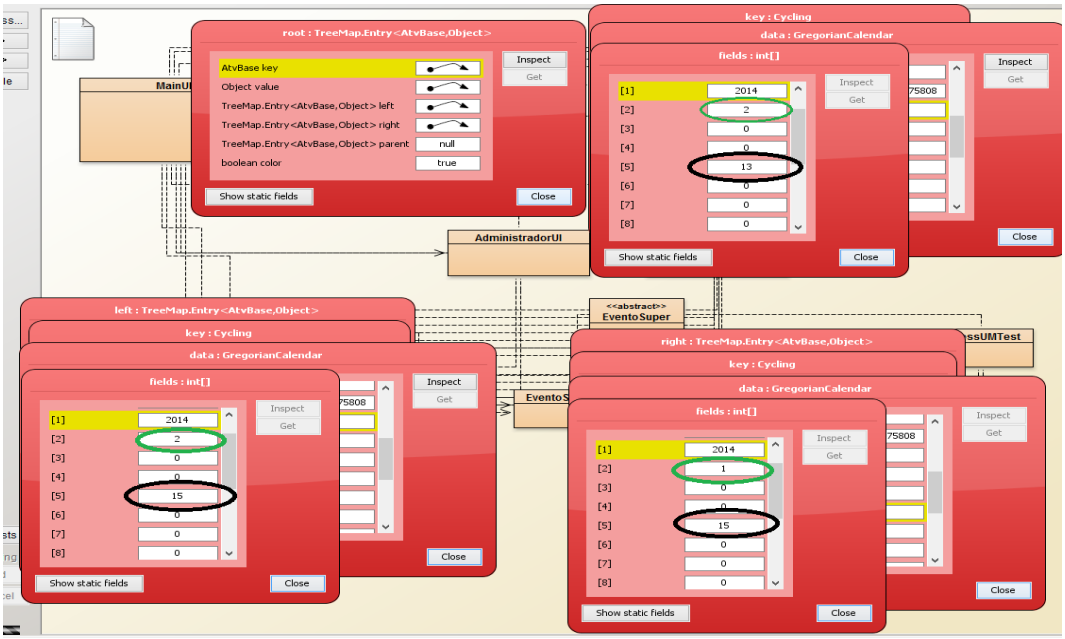


Figura 6: Aspeto interior da ráiz de uma árvore de atividades e do seu nodo esquerdo e direito. Nos círculos **a verde estão os valores dos meses** que para o *GregorianCalendar* contam a partir do 0 portanto no caso da raíz o mês é Março, a preto está o valor do do mês (o ano é 2014).

```
#####ATIVIDADES DE J. Caldas######
 1 - Desporto/Atividade: Corrida  Data: 15/3/2014
 2 - Desporto/Atividade: Cycling  Data: 15/3/2014
 3 - Desporto/Atividade: Corrida  Data: 13/3/2014
 4 - Desporto/Atividade: Cycling  Data: 13/3/2014
 5 - Desporto/Atividade: Corrida  Data: 19/2/2014
 6 - Desporto/Atividade: Cycling  Data: 19/2/2014
 7 - Desporto/Atividade: Corrida  Data: 15/2/2014
 8 - Desporto/Atividade: Cycling  Data: 15/2/2014
 9 - Desporto/Atividade: Corrida  Data: 12/2/2014
 10 - Desporto/Atividade: Cycling  Data: 12/2/2014


Para consultar o detalhe de uma atividade insira o seu número para saír insira 0:
```

Figura 7: Na figura podemos ver as 10 atividades mais recentes de um utilizador (apenas o cabeçalho pois temos de escolher a atividade para visualizar o seu detalhe) ordenadas **por ordem cronológica**. Os amigos do utilizador podem usufruir desta consulta.

```
### O MEU CALENDÁRIO DE ATIVIDADES ###

Fevereiro 2014
Dom   Seg   Ter   Qua   Qui   Sex   Sab
                                    1
  2     3     4     5     6     7     8
  9*   10    11    12*   13    14    15*
 16    17    18    19*   20    21    22
 23    24    25    26    27    28

Acções: ant (Mês anterior)  seg (Mês seguinte)    Dia do mes (Consultar atividade(s) do dia)    sair (Para saír):
```

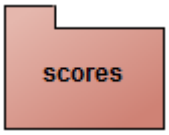
Figura 8: Nesta figura podemos observar o calendário de atividades, este é privado ao utilizador, apenas o próprio tem direito de navegar no seu calendário de atividades e consultar as atividades por *ordem cronológica*.

5 Interfaces

Antes de avançarmos mais no relatório é imperativo que se explicitem as interfaces presentes na hierarquia de classes de atividades na fig.4.

- **Distancia** - Marca todo o tipo de atividades que comtemplam distância, impondo às classes que se implementem métodos que permitem obter valores de distancias/velocidades.
- **DistAlt** - É subclasse da interface **Distância**, mas marca atividades que em simultâneo comtemplam distância.
- **Simulavel** - Obriga as classes a implementrem o método *public double getIncerteza()* que nos diz para uma dada atividade o valor de incerteza a que cada participante de um evento que associa essa atividade fica sujeito no decorrer da prova.
- **Rating** - Marca um conjunto de atividades que no seu score são comparáveis por rating.

6 Scores - Recordes Pessoais



Após o desenvolvimento das funcionalidades básicas da API é pedido ao grupo que desenvolva uma solução para que **cada utilizador posso aceder aos seus recordes psessoais em função de cada atividade**. Ora, o primeiro desafio será desenhar uma solução que possa lidar tanto com scores de atividades que contemplam distância, ou com scores de gastos de calorias ou duração etc. ... Ora este é o primeiro passo, pois sabemos à partida intuitivamente que teremos de usar uma das seguintes coleções, TreeMap ou TreeSet. O primeiro passo consiste em saber em que grupos de scores vamos dividir o nosso problema. Ora o primeiro é óbvio, será a distância/tempo, comparamos atividades que contemplam distância pela velocidade média no final da atividade para determinar o

melhor tempo do utilizador. O problema surge agora em arranjar critérios para as restantes classes. Foi pensado e discutido até com o docente da UC numa aula teórica que nas atividades com futebol, basket, surf, andebol, etc. ... Onde o score de cada utilizador é muito relativo, pois vejamos no futebol podemos marcar muitos golos mas termos estado estáticos no decorrer do jogo (o mesmo em muitos outros)...

6.0.1 Estrutura de dados

Surgiu então a ideia de avaliar scores destas atividades com base num **rating qualitativo** (0 a 10 pontos (*double*)) para estas atividades. As restantes atividades como Pilates e Zumba irão gerar scores com base no tempo de duração da atividade. Nas classes ScoreDistancia, ScoreDuracao e Score Rating temos a seguinte estrutura de dados:

```
public class Score****{
TreeMap<String,TreeSet<E>>  //Map: nome da atividade, TreeSet de atividades associado
...
}
```

Na fig.7 podemos muito bem observer a respresentação do TreeMap que declaramos em cima

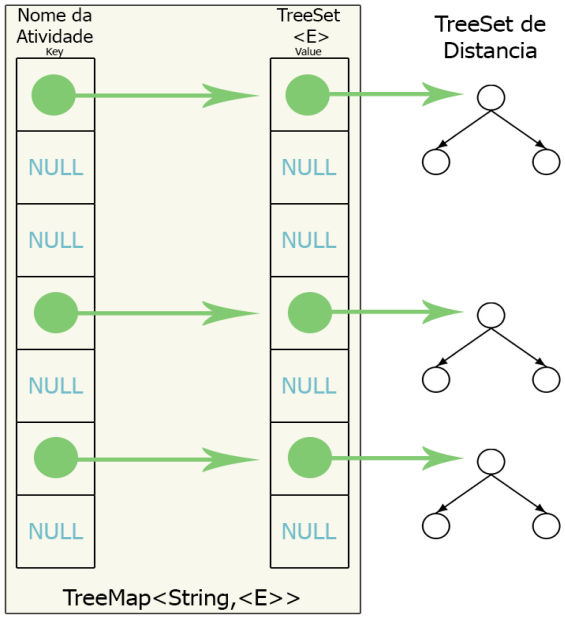


Figura 9: Estrutura de dados comum às classes de score, apenas varia o tipo de dados do TreeSet em conformidade.

A estrutura de dados é igual para as três classes de Scores apenas é alterado o tipo do TreeSet<E> para associar a uma atividade, no caso de **ScoreDistancia** *E = Distancia* e o algoritmo de comparação é definido em **ScoreDistComparator**, para **ScoreDuracao ScoreTimeComparator** *E=Atividade* compara

os tempos de duração das atividades e finalmente para **ScoreRating ScoreRatComparator** *E=Rating* ordena as atividades associadas com base numa avaliação qualitativa da prestação do utilizador.

Mas como agrupamos os três diferentes tipos de scores? Bem ao contrário das atividades o grupo decidiu aqui criar a classe **Score** que é criada através do mecanismo de composição. A cada utilizador é agora fornecedio **um novo tipo de dados**, a classe **Score**.

```
public class Score{

private ScoreDistancia scoredist; //para registar scores de atividades de distância/tempo
private ScoreDuracao scoreduracao; //para registar scores de duração
private ScoreRating scorerating; //para registar scores de rating

...
}
```

6.1 Uma nova funcionalidade

A partir do momento em que definimos a nossa estrutura de scores, a quando do registo de uma atividade essa atividade passa também a ser registada na instância de Score do utilizador, **agora temos acesso a um histórico de atividade ordenado por score**, e temos métodos que nos permitem aceder a esse certos valores em função da atividade, *tudo isto irá para além de simples "mandar para o ecrã scores"*, quando neste relatório abordarmos os **Eventos**. Na fig. podemos ver como funciona o registo de uma atividade.

7 Eventos

Os eventos desportivo, para o grupo o mais intrigante desafio do projeto, pensou-se numa solução que implicou que fossem executadas as seguintes tarefas:

- Criação de uma classe UI (*User interface*) **AdministradorUI** com credenciais de acesso especiais, i.e num contexto realista, a que mais ninguém possua acesso a não ser a administração - **user:** admin **pass:** admin;

- Criação de uma superclasse **EventoSuper** que abrangisse campos básicos de um evento

- Implementação de uma classe **EventoAdmin** (*subclasse de EventoSuper*) que armazenasse a informação do evento como o nome a data, o tipo de atividade associada etc. ... Mas que sobretudo contivesse um ou mais métodos que simulassem um evento dado um conjunto de utilizadores inscrito e que enviasse para a classe AdministradoUI através da Classe Users as tabelas que resultaram da simulação dos resultados. Esta classe possuí também métodos para inscrever utilizadores e enviar convites,v eremos que aqui o que se sucede não é muito diferente do que se sucede na rede de amigos.

- Criar uma classe **EventoSimples** que fosse íncluído no grupo de variáveis de instâncias de um utilizador, através do qual o mesmo pudesse receber/aceitar/rejeitar convites consultar detalhes e resultados (este último caso tivesse participado no evento).
- Codificar uma classe **Participante** que representasse um participante de um evento no decorrer do mesmo, que permitisse calcular o seu ritmo e atualizá-lo calcular a probabilidade de o utilizador desistir em função da idade e género e também atualizar este valor ao longo da prova. Também foi necessário criar um comparador para comparar participantes e atualizar a tabela classificativa ao longo da prova.

Para mais detalhes acerca de métodos podemos sempre consultar a documentação do código, ainda assim fica aqui a descrição de dois métodos cruciais neste desafio dos Eventos...

```
Na Classe User:
public String inscreveUserEmEvento(String mail, String nomeevento)

-Método que a quando a aceitação do convite por parte do utilizador o
"inscreve oficialmente", passos de execução:
    1 - Verifica se evento se encontra na data limite de inscrição
 (se ainda é válida a inscrição);
    2 - Verifica se evento ainda tem vagas disponíveis;
    3 - Caso tudo se verifique adicionamos o utilizador caso contrário devolvemos String
  com informação do porquê da operação ter "falhado";
    4 - No final removemos a atividade da lista de convites do utilizador;

Na Classe EventoAdmin:
public String SimulaEvento(Users users, String tempomet)

-Método que permite simular um Evento dados os utilizadores inscritos e a
descrição do tempo metereológico, algoritmo:
     1 - Recolha de dados: Criação de instância da atividade
   associada, criar ficha do evento;
     2 - Determinar fator de influência do tempo metereológico;
     3 - Inicializar tabela classificativa;
     4 - Simulação km a km com participantes dinâmicos guardando
   tabela classificativa km a km.
```

7.0.1 Evento, da Criação à Simulação

Vamos acompanhar um evento desde que este é criado até que o simulamos.

Vamos criar um evento, fazendo log in na conta do admnistrador.

Vamos então criar um evento...

Quando criamos um evento são automativamente enviados os convites ao utilizadores que praticam a atividade associada da seguinte forma, no final do método **CriarUmEvento** na classe *AdministradorUI*: **users.enviaConvitesDeEvento(evento);**, vemos em baixo o respetivo código.

```
########## PAINEL DA ADMINISTRAÇÃO ##########
1 - Criar evento
2 - Detalhes de um evento
3 - Simular evento
4 - Apagar um utilizador
5 - Apagar todos os utilizadores
0 - Sair
>
```

Figura 10: O menu de opções da administração.

```
###### CRIAR EVENTO #####


Nome do evento: M
Data do evento (dia mes ano)
Dia: 9
Mês: 9
Ano: 2014
Data de limite de inscrições (dia mes ano)
Dia: 8
Mês: 9
Ano: 2014
Limite máximo de inscrições: 10
Atividade associada: Corrida
Distância da prova: 20
```

Figura 11: A criar um evento, preenchendo os respetivos campos.

```
/**
 * Método que a partir de um eventoadmin recém-criado cria um evento simples
 * e efetua o convite a todos os users que praticam a(s) atividade(s) relacionada(s)
 */
public void enviaConvitesDeEvento(EventoAdmin ead) {


    //EventoSimples do qual criamos cÃ³pias para enviar ao utilizadores potenciais participantes
    EventoSimples es = new EventoSimples(ead.getNome(),
            (GregorianCalendar) ead.getData(),
            (GregorianCalendar) ead.getDataLimiteInscricoes(),
            ead.getLimiteInsc(), ead.getDist());

    boolean found;
    //Convidamos todos os users mesmo sabendo do limite de utilizadores,
    //pois temos de pensar que nem todos podem aceitar
    for (User user : this.users.values()) {
        found = false;
        if (user.getNomesAtividades().contains(ead.getAtividadeAssoc())) {
                found = true;
        }
        if (found == true) {
            user.addConvite(es.clone());
        }
    }
}
```

Figura 12: Enivamos os convites a todos os utilizadore que têm registo da atividade associada ao evento.

Este **Evento de nome "M"** já se encontra na classe de teste daí já termos os participantes inscritos como vamos ver a seguir, no entanto deixamos um pedido pendente para que possamos observar como é que um utilizador se inscreve num evento. Façamos então log in no utilizador **jorgecaldas**... **CriarUmEvento** na classe *AdministradorUI*: **users.enviaConvitesDeEvento(evento);**, vemos em baixo o respetivo código.

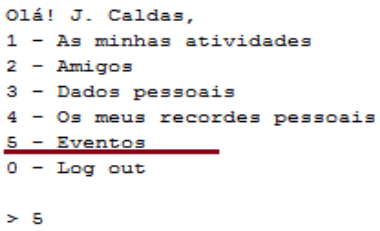


Figura 13: Acedemos à opção *5 - Eventos* de J. Caldas

De seguida o utilizador inscreve-se no evento (caso queira), mas antes tem acesso aos detalhes (públicos) do evento. **CriarUmEvento** na classe *AdministradorUI*: **users.enviaConvitesDeEvento(evento);**, vemos em baixo o respetivo código.

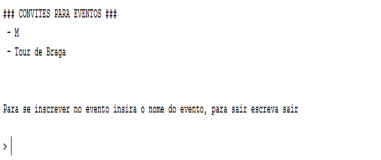


Figura 14: Inserimos o nome do evento se queremos proceder à inscrição, depois de vizualizados os detalhes.

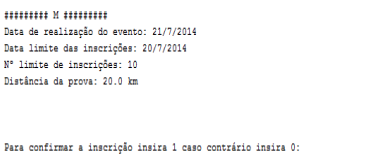


Figura 15: Detalhes do evento.

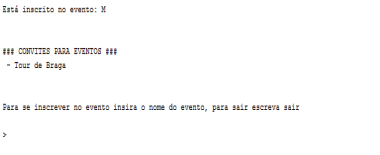


Figura 16: Confirmamos a inscrição!.

Só resta agora fazer log in na conta do administrador para começar a simulação do evento.

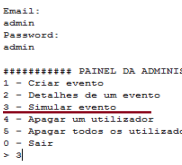


Figura 17: Escolhemos a opção *3 - Simular evento*.

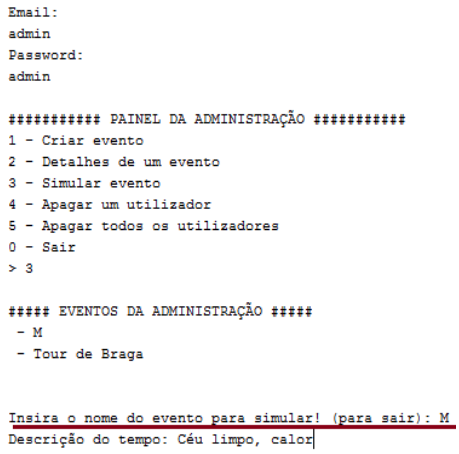


Figura 18: De seguida para dar ordem de simulação do evento.

Para dar a ordem de simulação do evento basta escrever o nome do evento e de seguida indicar as **condições metereológicas**, estas irão entrar também para o **cálculo dos ritmos dos participantes havendo fatores de afetação de tempo em pequenas percentagens, que alimentam a simulação com realismo**.

Neste momento todo o trabalho será executado pelo método **SimulaEvento** da classe *EventoAdmin*, este irá calculando os ritmos de cada particapante e depois vai gerar tempos aleatórios em intervalos fechados cuja variação é dado pelo **fator de incerteza** a variável de instância que todas as classes de Atividades que **implementam a interface Simulável** possuem. Exemplo:

1.249 *ritmoatualiza = (p.getRitmo()-incerteza) + (((p.getRitmo()+incerteza)-(p.getRitmo()-incerteza))) * refresh.nextDouble();*

A expressão em cima permite calcular o ritmo num determinado km da prova para um dado utiliziador.

De seguida é calculado e incrementado o tempo de prova do utilizador, através de uma regra três simples facilmente obtemos **o tempo que o utilizador X com o ritmo dado pela variável ritmoatualiza demora a percorrer um kilómetro**.

```
//Tempo que vai levar para o participante percorrer este kilómetro mediante o novo ritmo calculado
//CALCULO DO TEMPO POR UMA REGRA TRES SIMPLES
//  ritmoatualiza ------> 60 minutos
//       1 km     ------> ? quantos minutos => É este tempo que temos de adicionar ao tempo de prova do utilizador

double tempokm = (double) (60/ritmoatualiza);
p.addTempoDoKm(tempokm); //o tempo que demora a percorrer o kilómetro
```

Figura 19: Pedaço de código que calcular o tempo.

A cada volta é renovado uma variável que guarda a tabela classificativa de uma volta, *lap = new TreeSet<>(new ParticipanteComparator())* no final da volta com auxílio do fantástico **StringBuilder** construímos uma a tabela classificativa da volta e armazenamos essa String num Map *TreeMap<Integer,String> tabelas* que guarda para cada kilómetro (a chave por ordem natural) a string com a tabela classificativa correspondente...

Vejamos dois *screen shoots* de uma simulação um ao kilómetro 5 outro ao kilómetro 16.

```
- M - Tabela Classificativa ao Kilómetro: 5
1º Lugar: Nome: Bino Silva Ritmo: 13,80 km/h    Tempo: 0h:20m:20s
2º Lugar: Nome: Elina Castigo Ritmo: 10,66 km/h    Tempo: 0h:25m:37s
3º Lugar: Nome: Gina Portela Ritmo: 10,60 km/h    Tempo: 0h:25m:39s
4º Lugar: Nome: J. Caldas Ritmo: 11,37 km/h    Tempo: 0h:25m:16s
5º Lugar: Nome: Mario Silva Ritmo: 11,63 km/h    Tempo: 0h:25m:9s
6º Lugar: Nome: Joao Silva Ritmo: 10,20 km/h    Tempo: 0h:28m:52s
7º Lugar: Nome: Marta Costa Ritmo: 9,14 km/h    Tempo: 0h:30m:34s
8º Lugar: Nome: Arlindo Reis Ritmo: 8,24 km/h    Tempo: 0h:35m:16s

1 - Avançar   0 - Sair   >
```

Figura 20: Tabela classificativa ao kilómetro 5.

Ao kilómetro 16 podemos observar que dois dos participantes já desitiram, e as posições foram sofrendo alterações...

```
- M - Tabela Classificativa ao Kilómetro: 16
1º Lugar: Nome: Bino Silva Ritmo: 15,03 km/h    Tempo: 0h:56m:59s
2º Lugar: Nome: Gina Portela Ritmo: 12,22 km/h    Tempo: 1h:12m:54s
3º Lugar: Nome: Joao Silva Ritmo: 12,24 km/h    Tempo: 1h:17m:54s
4º Lugar: Nome: Mario Silva Ritmo: 10,90 km/h    Tempo: 1h:18m:30s
5º Lugar: Nome: J. Caldas Ritmo: 10,58 km/h    Tempo: 1h:20m:40s
6º Lugar: Nome: Arlindo Reis Ritmo: 8,33 km/h    Tempo: 2h:3m:12s
Nome: Elina Castigo(DESISTIU)
Nome: Marta Costa(DESISTIU)

1 - Avançar   0 - Sair   >
```

Figura 21: Tabela classificativa ao kilómetro 16.

Quando é dada por terminada a simulação o utilizador **J. Caldas** terá **acesso aos detalhes do Evento assim como à tabela classificativa final com os tempos e os ritmos médios dos utilizadores a cada kilómetro**!

Ao kilómetro 16 podemos observar que dois dos participantes já desitiram, e as posições foram sofrendo alterações...

```
######### M #########
Data de realização do evento: 21/7/2014
Data limite das inscrições: 20/7/2014
Nº limite de inscrições: 10
Distância da prova: 20.0 km


Resultados do Evento!
1º Lugar:  Bino Silva  Tempo: 1h:8m:56s    Ritmo médio: 14,97 km/h
2º Lugar:  Gina Portela  Tempo: 1h:29m:50s    Ritmo médio: 11,97 km/h
3º Lugar:  Joao Silva  Tempo: 1h:34m:50s    Ritmo médio: 11,46 km/h
4º Lugar:  Mario Silva  Tempo: 1h:38m:3s    Ritmo médio: 11,54 km/h
5º Lugar:  J. Caldas  Tempo: 1h:40m:13s    Ritmo médio: 10,94 km/h
6º Lugar:  Arlindo Reis  Tempo: 2h:30m:49s    Ritmo médio: 7,60 km/h
Elina Castigo (DESISTIU)
Marta Costa (DESISTIU)


##### EVENTOS EM QUE ESTOU INSCRITO ####
 - M

Insira o nome do evento para consultar detalhes ou sair>
```

Figura 22: Tabela classificativa final a que o utilizador tem acesso.

Apenas o administrador terá o poder de criar a simulação e comandá-la, daí só o próprio ter acesso aos tempos e ritmos por volta.

8 Excepções

O grupo de trabalho implementou também um conjunto de classes excepções para um efeciente tratamento dos diveros erros.

- **AmigoExisteException** - Excepção que indica que a pessoa que está a adicionar já é seu amigo.
- **ConviteEnviadoException** - Excepção que indica que a pessoa que está a adicionar já está na sua lista de pedidos enviados.
- **ProprioUserException** - Excepção que indica que está a tentar adicionar-se como amigo.
- **UserNaoExisteException** - Excepção que indica que o user que quer encontrar,adicionar como amigo,etc não existe.
- **NaoEnviouPedidoException** - Excepção que indica que o user ao qual quer rejeitar o pedido de amizade não lhe enviou nenhum pedido.
- **NaoAmigoException** - Excepçao que indica que não pode aceder as informações de um user porque não é amigo dele.

9 Bibliografia

- - www.endomondo.com
- - www.self.com
- - www.sparkpeople.com
- - www.nutristrategy.com
- - calorielab.com
- - www.everydayhealth.com
- - www.meiamaratonadelisboa.com
- - pt.wikipedia.org.meiamaratona

10 Conclusão

Após a conclusão deste projeto podemos afirmar que adquirimos bastantes conhecimentos no que toca à programação orientada a objectos. Fizemos uso das diversas capacidades da linguagem JAVA, incluíndo o polimorfismo que se alcança com abstração de classes que , a poderosa API disponibilizada pelo JDK e as funcionalidades geral do JAVA como as interfaces e as excepções. Posto isto, achamos o JAVA uma linguagem bastante poderosa e de uso relativamente fácil, muito à custa do polimorfismo e abstração de classes.