Geocaching POO

Junho 2015

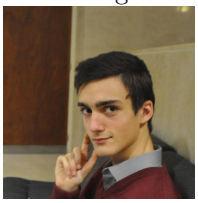
Grupo 42

Realizado por:

Jéssica Pereira a71164

Adelino Costa a70563

Martinho Aragão a72205



Conteúdo

1 Introdução

Este trabalho sobre o conceito de Geocaching conhecido nas redes sociais: pretendemos simular e registar atividades e descobrimentos de caches. Para isso foi necessário modularizar e fazer as devidas abstrações na preparação para o trabalho, pois quanto mais abstrações forem criadas mais independentes os módulos serão, podendo depois usar a composição entre estas classes e fornecendo uma melhor compreensão do código e tratamento da informação.

É necessário o estudo e concepção das classes necessárias, estruturas de dados, métodos respetivos, variáveis de instância e de classe necessárias, imagina a dependência entre as classes (composição) e definir uma hierarquia (nomeadamente criando uma super class para as Caches).

Para além de criar classes abstratas, também é necessário guardar todos os dados relativos às Caches e aos Utilizadores para poder criar métodos eficientes relativos ao tratamento destes.

De entre os requesitos básicos também serão implementados os Eventos, com a simulação da metereologia e cálculo de distâncias entre caches dados duas coordenadas (coordenadas iniciais e as coordenadas da cache). Estes dois últimos pontos serão implementados também em toda a descoberta de Caches/Actividades e não somente no decorrer de Eventos pois serão úteis nomeadamente para o cálculo de pontuações.

2 Modularização - Classes criadas

Decidimos apenas criar 4 tipos de Caches visto a Cache-Evento nao fazer sentido quando temos uma classe que simula eventos, e a Cache Virtual ...

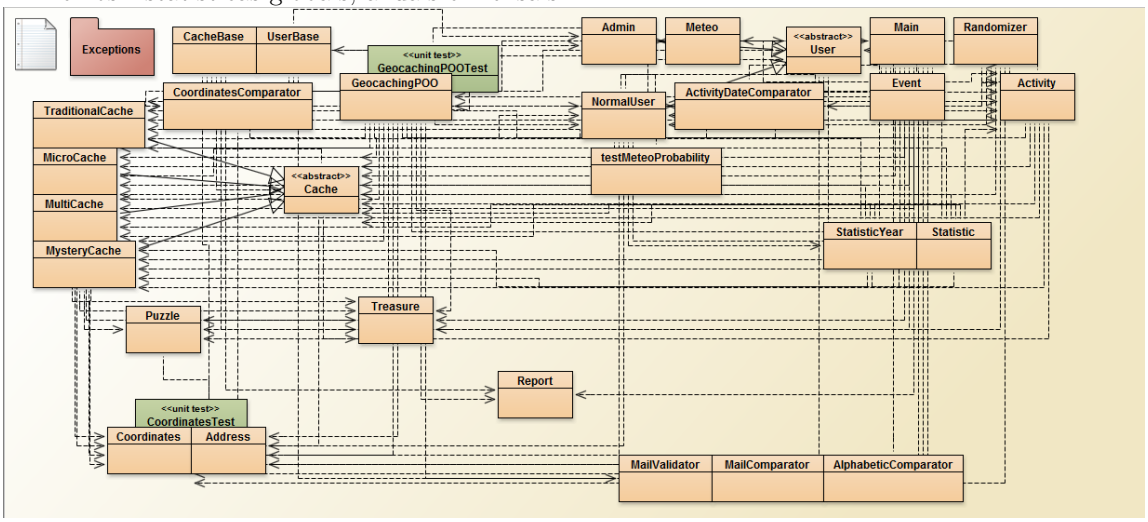
Guardamos todos os users e todas as caches criadas numa base de dados.

Para classes que usam estruturas do tipo set criamos vários comparadores úteis para diferentes tipos de ordenação a diferentes classes.

Inicialmente tinhamos uma classe chamada 'Server' que acabamos por eliminar pois esta baseia-se na classe GeocachingPOO. A ideia era na classe Server ter acesso a toda a informação, sendo esta apenas acessível aos Admins, mas no próprio GeocachingPOO temos a opção de fazer login como Admin, simplificando e colocando tudo numa classe.

Temos hierarquia quanto às caches e quanto aos users.

Temos Estatísticas globais, anuais e mensais.



3 Bases de dados

3.1 CacheBase

Antes de implementarmos a classe **CacheBase** reflectimos que características únicas teria cada cache para a diferenciar de todas as outras.

Cada cache tem coordenadas únicas, não sendo possível criar uma cache em coordenadas onde já existe uma cache, independentemente do tipo de cache, isto levou-nos a implementar um **TreeMap** para mapear coordenadas a IDs de cache.

Para guardar as caches utilizamos um **ArrayList** visto que sabendo o ID de uma cache é bastante fácil encontrá-lo nesta estrutura pois para um dado valor de ID sabemos que a Cache, caso exista, estará no indíce de valor igual a (ID - 1).

Portanto a implementação das variáveis de instância de **CacheBase** é a seguinte:

```
/* ArrayList com as caches */
private ArrayList<Cache> caches;
/* Mapeamento entre e-mails e IDs */
private TreeMap<Coordinates, Double> coords;
```

Como é necessário um dado Utilizador poder ver as caches que criou utilizamos um **TreeMap** para mapear IDs de Utilizadores a um **ArrayList** que contém os IDs das caches que o Utilizador criou, caso tenha criado alguma.

```
/* Mapeamento IDs de Utilizadores e IDs de Caches */
```

```
private TreeMap<Double, ArrayList<Double>> owners;
```

Finalmente para implementar o 'report' de Caches criámos outra variável de instância que mapea-se IDs de Caches a **ArrayList** de Reports dessa Cache.

```
/* Mapeamento entre IDs de Caches e Reports dessa cache */
private TreeMap<Double, ArrayList<Report>> reported_caches;
```

3.2 UserBase

Para guardar tanto Utilizadores como Administratores primeiros pensamos nas várias maneiras de referenciar um Utilizador, assumimos que as maneiras de referenciar um Utilizador seria através do seu ID ou através do seu e-mail.

Para os Utilizadores criámos utilizamos um **TreeMap** que faz o mapeamento de um e-mail para um ID, e utilizamos um **ArrayList** para guardar os Utilizadores pois torna-se bastante rápido encontrar um utilizador dado o seu ID, visto que para um dado ID o Utilizador, caso exista, estará no indice de valor igual a (ID - 1) no **ArrayList**.

Ficaram então definidas desta forma as variáveis de instância de **UserBase**:

```
/* ArrayList com os utilizadores */
private ArrayList<NormalUser> users;
/* Mapeamento entre e-mails e IDs */
private TreeMap<String, Double> userMails;
```

Para guardar as várias instâncias de **Admin** utilizamos o mesmo método que no caso das instâncias de **NormalUser**. Segue-se a definição das variáveis de instância:

```
/* ArrayList com os administradores */
private ArrayList<Admin> admins;
/* Mapeamento entre e-mails e IDs */
private TreeMap<String, Double> adminMails;
```

TODO

4 Caches

Como criamos uma Cache? Como tratamos da Invalidação de uma Cache? Como permitirmos ao user para reportar uma cache e quando é que ela é efetivamente reportada? pelo admin...

Mencionar que Cache é superclasse e os tipos de cache são sub-classes Estrutura de cada uma;

Ao fazer report de uma cache o admin tem acesso a essas caches e pode remover essa cache aceitando o report ou eliminar esse report. Se o utilizador remover a cache (invalidando-a) depois de ter feito report a essa cache, quando voltamos para o contexto do admin, essa cache já nao existe nos reports porque foi eliminada.

Para fazer login como admin existe uma conta: grupoajm@gmail.com cuja pass é AdminAdmin.

4.1 Cache Tradicional

4.2 Micro Cache

4.3 Multi Cache

4.4 Cache Mistério

5 Registo e Login

Uma das funções mais básicas da aplicação baseia-se no registo e login de Utilizadores. No Menu inicial o utilizador tem a opção de se registar ou então fazer login caso já possua uma conta.

Caso o utilizador deseje registar uma conta na aplicação terá de fornecer o seu e-mail, nome, password, data de nascimento, a cidade e país de residência e o seu género (Masculino ou Feminino). A class **Main** invocará os métodos necessários da classe **GeocachingPOO** que por sua vez invocará métodos da classe **UserBase** para registar o utilizador.

Caso o utilizador especifique um e-mail já em uso será negado o registo da conta. Caso se verifique que o utilizador se possa registar uma mensagem de sucesso aparecerá no ecrã, caso contrário uma mensagem de erro, informando o utilizador o porquê da falha do registo.

Aproveitamos para implementar um método que encripta a password do utilizador para que deste modo, se um utilizador executar o método da classe **User** que permite obter a password apenas verá o resultado da encriptação. Sempre que é necessário confirmar se a password fornecida coincide com a guardada no utilizador o programar encripta a password fornecida e verifica se é igual ao resultado da encriptação já armazenado.

Depois de um utilizador criar uma conta pode então executar o login, a classe **Main**, como trata de I/O, recebe qual o e-mail e a password do utilizador, a

Figura 1: Registo de um utilizador

classe **GeocachingPOO** trata do resto das invocações dos métodos necessários para executar o login.

Caso o e-mail esteja associado a uma conta existente e a password fornecida confira com a password guardada então será apresentado um menu ao utilizador, onde no topo lhe é apresentado o seu nome e o total de pontos que consegui juntar até ao momento.

A partir deste menu de utilizador o utilizador consegue executar todas as funções básicas propostas, como, por exemplo, mudar informações pessoais (nome, e-mail, género, password,..), criar uma cache de enter os 4 tipos que a aplicação conhece, registar uma activadade, consultar as suas actividades, enviar pedidos de amizade, etc.

Decidimos também implementar uma opção no menu relativo a caches que

permita a um utilizador descobrir quais as caches que se encontram a uma determinada distância (raio) de uma certa localização, indicada pelo utilizador.

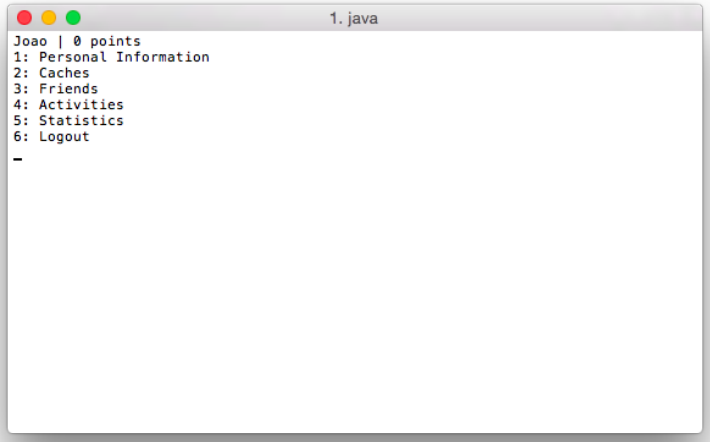


Figura 2: Menu após o login

6 Atividades e Estatísticas

6.1 Atividades

Quando o user encontra uma cache, regista uma nova Atividade que é automaticamente adicionada às Estatísticas.

Esta atividade têm como informações a data em que foi adicinada, a Cache que foi encontrada, os kilometros que foram percorridos para encontrar esta Cache, a pontuação ganha pelo feito e a Meteorologia desse dia.

O utilizador apenas tem de fornecer os dados quanto à Cache que encontrou e em que dia foi. O resto é tratado pelo programa, havendo simulações e cálculos.

6.1.1 Kilómetros

Quando adiciona a primeira atividade, são criadas as coordenadas iniciais do local onde o User iniciou a procura, geradas aleatóriamente fazendo cópia das coordenadas da Cache e um incremento da latitude e longitude de valores entre 0,001 e 0,4 por exemplo. No momento em que, ao adicionar esta primeira atividade, o programa calcula as distâncias entre estas coordenadas geradas aleatóriamente com base neste range, e as coordenadas da Cache, os kilometros calculados dão entre valores de 0,1km até 40 kms, em média. Este incremento de latitude e longitude são métodos que usam o Math.Random, presentes na classe "Coordinates".

Caso já exista uma atividade anterior a esta que está prestes a ser adicionada, a distância calculada será entre as coordenadas desta cache e as coordenadas

da cache da ultima atividade, tornando isto o mais realista possivel. A última atividade é a última posição conhecida do utilizador.

6.1.2 Meteorologia

Na classe Meteo são geradas as meteorologias de uma forma aleatória também. Existem dois campos que dizem respeito à Temperatura e ao Estado de Tempo.

int Low = -10;

int High = 40;

São definidos os valores mínimos e máximos para a Temperatura. Para a Weather existem 7 tipos possíveis que são os seguintes:

Rainy 0

Stormy 1

Sunny 2

Cloudy 3

Windy 4

Foggy 5

Hail 6

A cada estado de tempo está associado um número que vai ser gerado aleatóriamente com o Math.Random entre os valores de inteiros de 0 até 7 (exclusive).

Tudo isto é muito importante para as Estatísticas pois são calculados os pon-

tos em cada Atividade, conforme o estado de tempo, os kilometros percorridos e o tipo de cache encontrada.

Para cada atividade existe um limite de pontos que decidimos atribuir: 100 pontos. Assim se quisermos diminuir ou aumentar este limite é só tratar da escala deste limite que será mais fácil. (Por exemplo, para cada atividade apenas querer 10 pontos, ou querer 1000, etc. ...).

Este limite de pontos total rege-se também por um limite às 3 atribuiçoes de pontos que criamos, kilometros, meteorologia e cache, que são apresentados de seguida:

```
private static int limit_points = 100;
private static int limit_points_cache = 50;
private static int limit_points_kms = 30;
private static int limit_points_meteo = 20;
```

* Decidimos atribuir mais pontos ao fator do tipo da cache. Se for uma Micro Cache atribuimos o mínimo de pontos (10) mas se for uma Mystery Cache o user consegue obter os 50 pontos máximos, dependendo também da dificuldade do Puzzle que foi atribuida. Como a estrela de dificuldade vai de 1 a 10, decidimos atribuir dificuldade * 5 pontos pela Mystery Cache.

Da mesma forma, tomamos decisões para o cálculo de pontos da Meteorologia e dos kilometros. Se percorrer mais kilómetros, ganha mais pontos e se o tempo estiver mau (Stormy, Hail ...) atribuimos o máximo de pontos para a Weather que são 10 pontos. Não esquecer que também temos a Temperatura, e para o caso de temperaturas extremas (muito baixas e muito altas) atribuimos também

pontuações máximas. Depois, para cada situação à uma atribuição mediana conforme a meteorologia e temperatura.

TODO CODIGO A SER FEITO

O user tem possibilidade de visualizar as 10 últimas Atividades tanto dele como dos amigos, e de, se assim o quiser, as remover. Quando remove uma atividade, todas as informações e pontos adquiridos por esta são automaticamente retirados das estatisticas.

6.2 Estatísticas

Quando às Estatísticas temos duas classes: StatisticYear e Statistic. A razão pela qual temos duas classes é porque decidimos guardar também as Estatísticas Globais do user, ou seja, as estatísticas de todos os anos. Para melhor entender o funcionamento e a estrutura destas classes, começaremos por explicar a Statistic.

Na Statistic temos as estatísticas de um dado ano. A estrutura é a seguinte:

`ArrayList < TreeSet<Activity>>.`

Tentamos criar um array para ter em cada indice o mês ligado ao conjunto de atividades desse mês mas é impossivel criar array com valores que não sejam primitivos, logo tivemos de mudar a estrutura para um ArrayList. Este ArrayList terá como indices o mês da estatistica. O seu conteúdo será o conjunto de Atividades realizadas nesse mês. Para adicionar uma atividade garantimos que não é possivel adicionar uma atividade se o ano não for o mesmo. Por isso é importante buscar o ano desta estatística, fazer set do ano desta estatística... O ano assumido como default é o current year que é 2015. Também são disponibi-

lizadas funções como o de contar quantos tipos de cache num dado mês existem, e devolver num array, soma de pontos, de kms totais, numero de todas as caches encontradas, tanto para este ano como para um dado mês. Estas função são usadas como auxiliares na classe StatisticYear, e o porquê será entendido em seguida, quando for explicada a estrutuda da mesma.

StatisticYear é basicamente uma estrutura que mapeia um dado ano para uma Statistic. É esta a estrutura usada no programa pois facilmente se adiciona uma atividade nesta estrutuda, tendo o ano em que queremos inserir na data da atividade e usando os métodos existentes no Statistic como auxiliares. Todo o resto sobre o numero total de pontos, kilometros percorridos, numero de caches, etc... também são fornecidos por métodos de assinaturas iguais (tirando partido do Overloading), tanto para um dado ano como para todos os anos (global).

Permitimos ao user ver as suas estatiticas globais, anuais (de um dado ano) e mensais. Para as estatisticas mensais assumimos que ele quer ver um dado mês do current year (2015). Mas tal pode ser alterado facilmente, pedindo apenas mais uma informação extra ao user (que ano quer) antes de mostrar informações destas estatísticas.

Existe ainda uma opção para ver um gráfico de quantas caches dos tipos existentes o usuário encontrou, num dado mês.

7 Amigos

Implementamos também uma rede de amigos, cada instância de **NormalUser** contém variáveis para guardar os IDs dos utilizadores que são seus amigos e os IDs dos utilizadores que enviaram pedidos de amizade. Essas variáveis ficaram então definidas da seguinte forma:

```
/* IDs dos amigos do utilizador */
ArrayList<Double> friends;
/* IDs dos utlizadores que enviaram pedidos de amizade */
ArrayList<Double> friend_requests;
```

A aplicação permite que utilizadores enviem pedidos de amizade a outros utilizadores inserindo o e-mail a quem desejam enviar o pedido de amizade. O utilizador que recebe o pedido de amizade pode aceitar o pedido de amizade e ambos os utilizadores guardam o ID um do outros na sua lista de amigos. A lista de amigos pode depois ser visualizada dentro da própria aplicação.

Ao adicionar um amigo um utilizador poderá então, se desejar, visualizar a lista das 10 actividades mais recentes de qualquer um dos seus amigos.

8 Eventos

TODO

9 Classes Main e GeocahingPOO

Para poder utilizar as classes desenvolvidas para o projecto criamos uma class à parte, a classe **Main**, que contém o método *main. O único propósito desta classe é então lidar com a parte de I/O, relativamente apresentação de menus e tratamento de input por parte do utilizador, esta classe também é responsável por invocar os diferentes métodos existentes na classe* ***Geocaching*** *quando o utilizador necessitar de se registar, criar uma actividade, etc. Também é neste método* main que se lida com as Excepções criadas.

A classe **GeocachingPOO** contém variáveis das classes **CacheBase** e **UserBase** para poder guardar os utilizadores e as caches criadas. Contêm também variáveis para guardar o utilizador ou administrador atualmente 'loggado' e variáveis para controlo de IDs actuais a utilizar.

A classe contêm métodos para executar os vários requisitos do projecto, por exemplo, registar um utilizador novo, a classe faz isto invocando métodos existentes em **UserBase**, já no caso de criação de caches, por exemplo, a classe invoca métodos definidos em **CacheBase**. Basicamente a classe **GeocahingPOO** agrupa os vários métodos que sejam precisos para utilizar a aplicação, definindo também quais os métodos necessários invocar para os diferentes objectos e ações executadas.

Separar a parte de I/O da classe **GeocachingPOO** permite que a classe seja reutilizada noutros ambientes, por exemplo com uma Interface Gráfica.

10 Salvaguarda do Estado

Para guardar o estado da aplicação utilizamos classes já definidas em Java, nomeadamente a class **ObjectOutputStream**. Utilizando esta classe é possível criar um ficheiro objecto com as informações de todos os objectos da aplicação no estado actual. Para isso incluimos uma opção no menu principal que guardar o estado actual da aplicação num ficheiro com o nome 'geocaching'.

Para carregar um estado anterior da aplicação utilizamos a class **ObjectInputStream** que permite ler objectos de um ficherio objecto. Adicionamos uma opção no menu principal para carregar um estado anterior da aplicação, a aplicação tenta automaticamente ler o ficheiro 'geocahing' criado quando se guardar o estado da aplicação, caso o ficheiro não esteja na mesma diretoria onde se executa a aplicação o estado anterior não será carregado.

11 Tratamento de Excepções

Decidimos colocar todas as excepções num package chamado Exceptions para uma melhor visualização das classes e clareza no diagrama. Para todas as classes que implementam Excepções é feito import deste Package. Apenas a função *main faz o tratamento de Excepções.*

12 Conclusão