RELATÓRIO DE PROJETO

JAVA Porjects

REALIZAÇÃO:

Todo o código demonstrada e aplicado tem inteira isenção de fontes externas ou até mesmo de outros projetos como este. A aplicação única de cada código é o fator que diferencia cada projeto. Com isto dizer, todo o código foi criado pelos respetivos alunos.

André Loras

Engenharia Informática – UC2015250489

João Rodrigues

Engenharia Informática -UC2016225773

Índice

Nota Introdutória

A feitura do presente trabalho vem de encontrar a complementar e aglomerar diversos conceitos lecionados durante todo o semestre, com objetivo de estabelecer um entendimento perante a linguagem de programação, neste caso, JAVA.

JAVA é uma linguagem orientada a objetos e uma plataforma computacional. Existem imensas aplicações, as quais utilizamos no quotidiano, que são criadas e implementadas a partir de JAVA. Esta linguagem de programação visa simplificar algumas restrições em acessos de memória, tornando-a uma linguagem rápida, segura e confiável.

Para criar esta aplicação, foram dedicados esforços ambos para compreender conceitos e aplica-los da melhor maneira. No entanto, como não existem códigos perfeitos, o que apresentamos nas seguintes páginas é a nossa melhor tentativa com a experiência que temos e com o tempo que nos foi dado para enraizar todos os conceitos lecionados, que passam de simples expressões de contro, inicialização de variáveis, tipos de variáveis, estruturas de dados, até listas, arrays, ficheiros, classes, as diversas nuances das classes, interfaces gráficas e outros.

Acreditamos ainda que dada a realização deste projeto, tendo em conta todos os requisitos, conseguimos adicionar conceitos nossos, originais, que achamos que iria completar de várias formas o projeto. Por exemplo, acrescentamos a funcionalidade de desinscrever um sujeito, habilitamos o login múltiplo, ou seja, o utilizador podes fazer quantos logins pretendes e manipular as suas informações, adicionamos também a funcionalidade de retirar um local da lista de locais instantaneamente, possibilitando a que, se o utilizador já não quiser ir àquele local que o posso retirar sem quaisquer problemas, sem ter de se desinscrever. Ainda adicionamos funcionalidade de ver inscritos por locais. Todas estas funcionalidades, e outras que não mencionamos, foram adicionadas de acordo a tornar a aplicação mais “User Friendly”, com um nível de realidade mais acentuado que um mero projeto. Basicamente o nosso objetivo foi criar algo que pudesse ser utilizado para fins concretos.

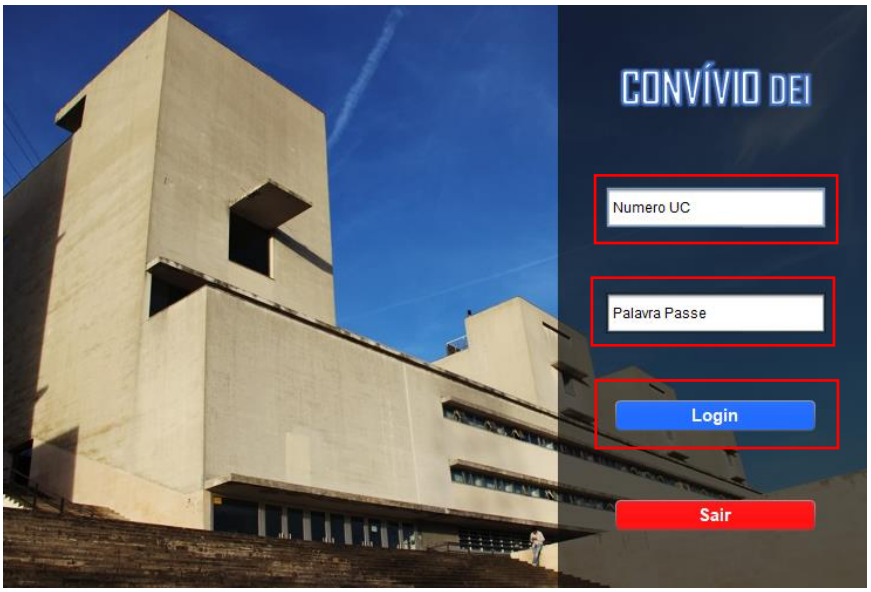
Por último, não achamos que esta realização seria o suficiente para ficarmos entendidos no que concerne a JAVA e as suas múltiplas funcionalidades. Pelo contrário. Haverá certamente mais a aperfeiçoar e mais que aprofundar no que diz respeito a esta linguagem de programação. Porque, a nosso ver, esta aplicação e a maneira que foi realizada ainda está a um nível muito amador. E é necessário melhorar.

HOW TO RUN IT?

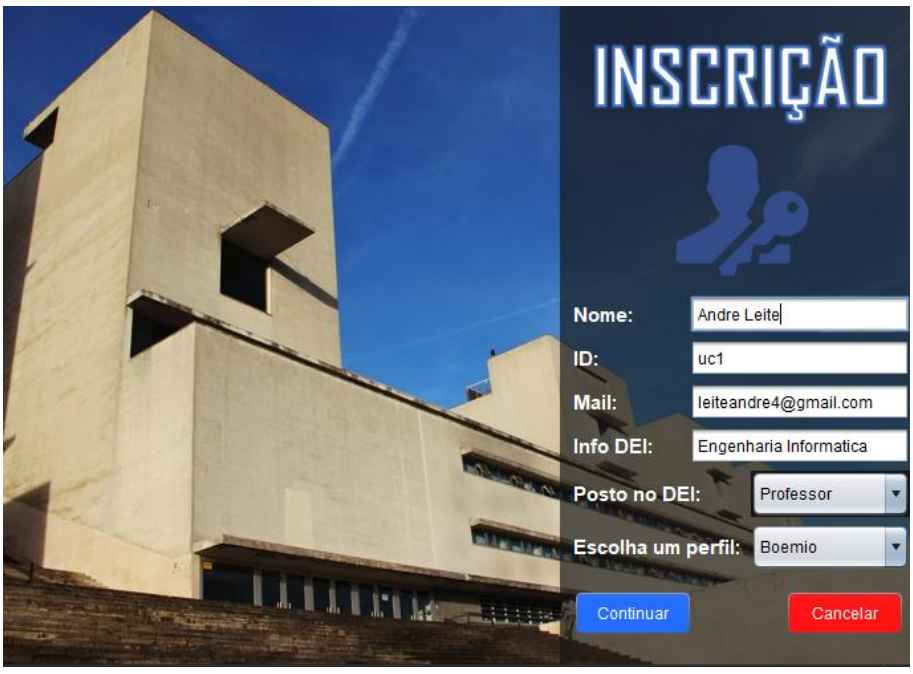
Como correr a aplicação?

Esta aplicação mesmo sendo o máximo "user friendly" que conseguimos alcançar, convém ser explicada de forma detalhada de forma a que não haja réstia de dúvidas quanto ao seu funcionamento.

Comecemos pelo menu de login:



Neste menu podemos verificar que existem duas entradas para texto, na **primeira está reservada a entrada ao número da instituição**, quanto ao **segundo está reservada a palavra** passe aplicada pelo utilizador. O **botão a azul**, irá levá-lo ao menu de inscrição se ainda não existir na base de dados de inscritos ou levá-lo-á ao menu principal no caso de já estar inscrito e estar mesmo a fazer login e não o primeiro login.



Neste menu, podemos verificar que existem mais componentes gráficas. Este menu é alcançado após o primeiro login. É responsável pela verificação e implementação da informação do utilizador no convívio. É também o primeiro frame em que o utilizador tem contacto com a sua informação, podendo confirmar se é mesmo ele.

É necessário nesta frame que o utilizador apenas escolha o Posto no DEI e o perfil, dado que as informações que são anteriores são informações base do utilizador, tal como no cartão de cidadão, não poderão ser modificadas de qualquer forma.

Aqui o utilizador terá de escolher de entre três postos possíveis, aluno, professor e auxiliar. também terá de escolher de entre quatro possíveis perfis, boémio, poupadinho, desportivo e cultural, sendo que o único a interferir diretamente com o ambiente é o perfil boémio.

Clicar em continuar quanto já tiver selecionado as suas informações irá increvê--lo no convívio e seguidamente levá-lo à frame de visualização e escolha de locais.

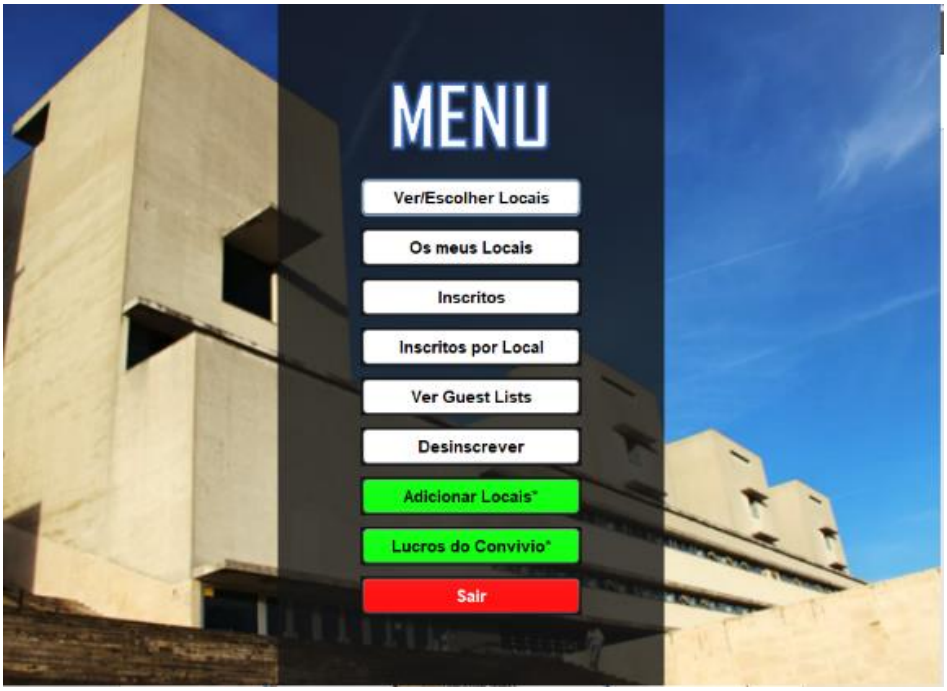
Tal como referido a cima, esta é a frame que aparecerá quando a inscrição for feita. É a frame onde escolherá o local, retirar o mesmo e ver todos os locais existentes no Convívio.

Tal como é possível verificar, o locais são disposto com o seu nome e com o numero de pessoas que lá estão inscritas de forma a facilitar a escolha.

Poderá também selecionar o tipo de local que pretende visualizar utilizando a barra mais à esquerda. Esta barra cobre todos os tipos de locais e facilita uma visualização no caso haver imensos locais e a apresentação em janela ser demasiado. Esta seleção reduz a informação em lista facilitando o uso.

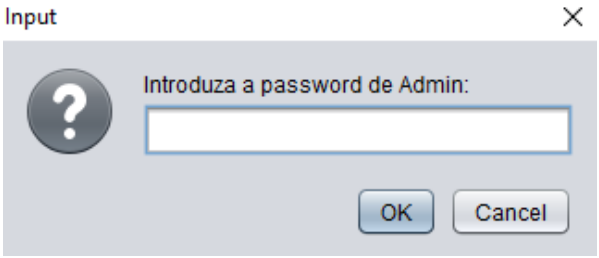
Para visualizar a informação de cada local basta clicar no mesmo e uma janela popup irá aparecer com as informações relativas a esse mesmo local. Informações essas que são possíveis exemplificar pela imagem que a cima se apresenta.

Selecionando o local na lista e depois clicando no selecionar será o método de inscrição em cada local. Para retirar este local basta clicar no mesmo e clicar no botão Retirar de forma a retirar o local da sua lista de locais e retirar o utilizador da lista de pessoas, do local. Este método funciona em paridade, ou seja, retirar o local da pessoa e retirar a pessoa do local e toda a informação referente, pois no caso de ser um bar, também terá de retirá-lo da Guest List caso faça parte.



Esta frame que se visualiza a cima, é o menu principal da aplicação. É aqui que se podem navegar por diante de todas as funcionalidades. É a frame mais complexa e a que mais ligações tem. Todas as opções que se visualizam estão operacionais e quando selecionadas permitem **SEMPRE** o retorno a este menu principal. **Somente em dois casos** particulares é que quando selecionados o executável irá encerrar e o programa também. Estes casos são o **Sair** e o **Desinscrever**. São métodos destrutivos e finais.

A **verde** estão representadas funcionalidade de Administrador, que poderão ser selecionadas, mas só serão executadas se no login que é disposto inserir a palavra passe de administrador, que neste caso foi implementada “admin” por convenção.



Todos as outras opções são abertas as todas as personalidades inscritas no Convívio.

Breve nota sobre as Opções e as suas Funcionalidade:

1. **"Ver/Escolher Locais"** é funcionalidade base desta aplicação visto que é aquela onde se pode visualizar e selecionar/retirar locais. Tal como já tinha sido explanado anteriormente neste relatório;
2. **"Os meus Locais"** é a opção que permite a visualização dos locais selecionados pelo Inscrito. Caso tenha retirado locais, esta lista também será atualizada;
3. **"Inscritos"** é a opção que permite a visualização de todos os inscritos no Convívio;
4. **(Extra)** "**Inscritos por local"** é a opção regente pela disposição de todos os inscritos espalhados pelos locais que selecionaram;
5. **"Ver Guest Lists"** é a segunda mais importante funcionalidade segundo o enunciado deste projeto. É a funcionalidade capaz de dispor as Guests Lists de cada bar e os seus inseridos. Todas as mudanças feitas nos locais e nas suas Guest Lists serão atualizadas e dispostas da mesma maneira na Frame. Por isso esta informação não é completamente estática pois está sempre sujeita a mudanças;
6. **(Extra) "Desinscrever"** funcionalidade capaz de libertar toda a informação do inscrito que a selecionar. Todos os locais onde este utilizador estiver inserido serão modificados, eliminando-o dos seus registos e retomando a informação como se ele nunca tivesse entrado. O mesmo se aplicada ao utilizador.
7. **(Extra) "Adicionar Locais"** é capaz de inserir locais no convívio a qualquer alturar que o desejar. No entanto, está indisponível devido a erros de Frame;
8. **"Lucros do Convívio"** é a opção que gere todos os fundos gerados pelas inscrições nos locais. Todos os lucros e lucro total, são gerados aqui e dispostos consequentemente. Esta funcionalidade também não é completamente estática na medida que está sujeita a alterações;
9. **"Sair"**, esta opção para além de cancelar todas as operações e fechar o programa também é a que os atualiza em ficheiros antes de fechar. Mantendo assim a informação o mais atualizada possível;

Métodos e Classes

Nesta instância do relatório vamos focar as atenções nas funções aplicadas e as classes que as possam conter.

Começando pela classe abstrata **convivioDe**i:

```
package com.AndreL;

import java.io.Serializable;
import java.util.LinkedList;
import java.util.List;

public class convivioDei implements Serializable {

    protected String perfil;
    protected String password;
    protected String id;
    protected String nome;
    protected Ficheiros fich;
    protected String mail;
    protected List<Locais> going;

    public convivioDei() {...}

    public void inscreve(){
        fich = new Ficheiros();
        fich.writeFile( fileName: "C:\\Users\\killz\\Desktop\\FCTUC\\2º ANO\\1º_Semestre\\POO\\ProjetoPoo\\Files\\baseDadosInscritos", pessoa: convivioDei.this);
    }

    public void addLocal(Locais x) { going.add(x); }

}

class aluno extends convivioDei{...}

class professor extends convivioDei{...}

class auxiliar extends convivioDei{...}
```

Esta função é responsável pela criação dos inscritos no sistema e pela sua escrita em ficheiro. Como podemos verificar a classe abstrata **convivioDei** é capaz de garantir a criação das informações do inscrito.

Começamos por inicializar os diversos atributos.

Consequentemente, iremos estabelecer os seus valores no construtor da própria classe abstrata, garantido que mal a criemos, estes valores sejam imediatamente criados em memória e guardados.

Seguindo pela classe, são criadas duas funções sem qualquer tipo de return, de maneira que implementamos com **void**. Estas funções são responsáveis pela escrita em ficheiro de texto (inscreve) e por adicionar locais à lista que cada pessoa tem direito.

No construtor todas as varáveis são inicializadas a **null** ou ao valor nulo, de modo a que só utilizarão informação concreta quando criarmos um inscrito, que as receber por herança e as poderá modificar no seu próprio construtor.

```
    public convivioDei() {
        this.nome = "None";
        this.id = "None";
        this.perfil = "None";
        this.mail = "mail@mail.com";
        this.password = "default";
        this.going = new LinkedList<>();
    }
```

```java
class aluno extends convivioDei{

    private String curso;
    private double promo = 0.10;

    public aluno(String curso,String password, String nome, String id, String perfil, String mail) {
        this.curso = curso;
        super.nome = nome;
        super.id = id;
        super.perfil = perfil;
        super.mail = mail;
        super.password = password;
        super.going = new LinkedList<>();
        inscreve();
    }

    public String getCurso() { return curso; }
    public double getPromo() { return promo; }

    @Override
    public void inscreve(){...}

    @Override
    public void addLocal(Locais x) { going.add(x); }

}
```

Esta subclasse é a responsável pela criação do objeto do tipo “aluno”, que terá em si informações específicas, daí serem inicializadas variáveis diferentes da classe abstrata e das subclasses seguintes.

No construtor desta subclasse iremos herdar as informações da classes abstrata e modificar esses mesmo valores com os parâmetros que são inseridos quando se inicializa um objeto deste tipo.

Por polimorfismo, podemos inicializar este objeto como:

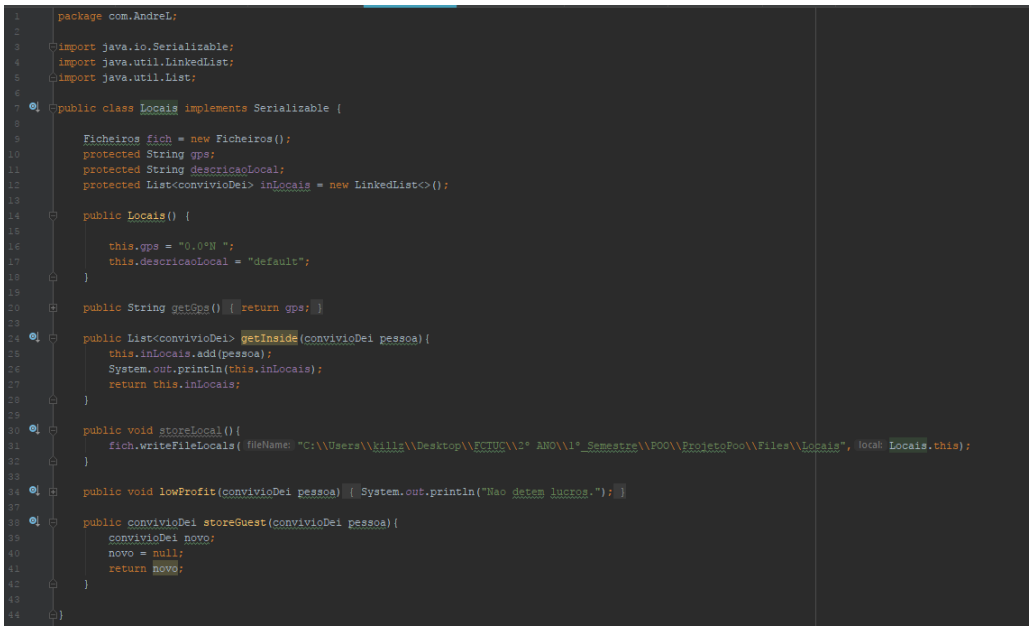
```java
convivioDei novo = new aluno("EI","123","Andre","2","x","@");
```

Como partilham funções, podemos dizer que é possível criar o objeto aluno por parte de do objeto convivioDei.

As funcionalidades das funções nesta subclasse são as mesmas do que na classe abstrata.

Para as subclasses seguintes os casos são semelhantes, com a diferença que cada subclasse tem os seus atributos, devido a uma questão lógica de personalidades.

Utilizamos nas classes, variáveis como **protected** para haver cedência de informação para as subclasses (herança). No entanto, as subclasses terão variáveis **privadas** pois não nos interessa partilhar as informações destes locais por toda a memória.



```java
package com.AndreL;

import java.io.Serializable;
import java.util.LinkedList;
import java.util.List;

public class Locais implements Serializable {

    Ficheiros fich = new Ficheiros();
    protected String gps;
    protected String descricaoLocal;
    protected List<convivioDei> inLocais = new LinkedList<>();

    public Locais() {

        this.gps = "0.0°N ";
        this.descricaoLocal = "default";
    }

    public String getGps() { return gps; }

    public List<convivioDei> getInside(convivioDei pessoa){
        this.inLocais.add(pessoa);
        System.out.println(this.inLocais);
        return this.inLocais;
    }

    public void storeLocal(){
        fich.writeFileLocals( fileName: "C:\\Users\\kills\\Desktop\\FCTUC\\2° ANO\\1° Semestre\\POO\\ProjetoPoo\\Files\\Locais", local: Locais.this);
    }

    public void lowProfit(convivioDei pessoa) { System.out.println("Nao detem lucros."); }

    public convivioDei storeGuest(convivioDei pessoa){
        convivioDei novo;
        novo = null;
        return novo;
    }

}
```

Passemos à segunda e ultima classe abstrata, Locais.

Esta classe é responsável pela manutenção e gestão dos objetos deste tipo. É esta classe que é responsável pela criação de locais e tudo que elas incluem.

Como é possível verificar nesta classe, na semelhança da anterior, os locais também terão cada um as suas características próprias, por isso é imperativo que tenhamos objetos do tipo local, e depois nesses locais, os vários tipos com as suas características próprias.

Novamente, tal como classe anterior, o construtor desta classe irá inicializar as variáveis a **null**, para depois, em cada objeto serem modificados respetivamente.

Utilizamos nas classes, variáveis como **protected** para haver cedência de informação para as subclasses (herança). No entanto, as subclasses terão variáveis **privadas** pois não nos interessa partilhar as informações destes locais por toda a memória.

Quanto às funções implementadas nestas classes, é imperativo ter funções que insiram o local em ficheiro (**storeLocal**), que insiram o inscrito na lista de inscritos referente ao local (**getInside**). Em casos que as classes gerem lucros também será necessária uma função que caso o inscrito sai da base de dados, que retorne as informações anteriores à sua inscrição e para isto criamos a função **lowProfit**, que diminui o lucro quando uma pessoa sai do mesmo. Nesta função fazemos referência ao desconto do objeto do tipo aluno criando uma condição que a verifique.

```java
class exposicao extends Locais{

    private String arte;
    private double ingresso;
    private double lucro;

    public exposicao(String arte, double ingresso, String nome, String gps) {
        this.arte = arte;
        this.ingresso = ingresso;
        super.gps = gps;
        super.descricaoLocal = nome;
        storeLocal();
    }
```

Neste exemplo iremos dispor a subclasse exposição que herda os atributos e métodos da classes abstrata e utiliza-os / modifica-os e acordo com as necessidades de código para cada caso.

Nesta classe só diferenciam as novas variáveis e os seus respetivos **getters**. Contudo, todos os restantes métodos permanecem iguais.

É importante referir também a subclasse bares. Visto que está presente nesta a **guestlist**.

Começando por referir que a variável **t é o tamanho máximo inicial (é decrementado)** da guest list, a variável **t_F é o tamanho máximo da guest list (estático**), a variável **lotação é a lotação do local (é decrementada)** e a varável **lotação_f é a lotação máxima do local (estática)**.

```java
public convivioDei storeGuest(convivioDei pessoa){

    boolean state =  false;
    convivioDei fora = null;
    if((this.lotacao > 0 && this.lotacao <= this.lotacaof && t == 0)||(this.t == 0 && this.lotacao == 0 && pessoa.perfil.equals("Boemio"))){
        for(int i = guest.size()-1 ; i >= 0 ; i--){
            if((!(guest.get(i).perfil.equals("Boemio")) && pessoa.perfil.equals("Boemio")) && this.lotacao > 0){
                guest.add(i,pessoa);
                guest.remove( index: i+1);
                state = true;
                break;
            }else if((!(guest.get(i).perfil.equals("Boemio")) && pessoa.perfil.equals("Boemio")) && this.lotacao == 0){
                guest.add(i,pessoa);
                lowProfit(guest.get(i+1));
                System.out.println(guest.get(i+1).going);
                int comp = guest.get(i+1).going.size();
                System.out.println(comp);
                for(int p = 0; p < comp; p++) {
                    if(guest.get(i+1).going.get(p).descricaoLocal.equals(this.descricaoLocal)){
                        guest.get(i+1).going.remove(guest.get(i+1).going.get(p));
                        fora = guest.get(i+1);
                        System.out.println(guest.get(i+1));
                        System.out.println(guest.get(i+1).going);
                        break;
                    }
                }
                guest.remove( index: i+1);
                state = true;
                this.inLocais.remove( index: this.inLocais.size()-1); //Retira o ultimo a estar inscrito no local
                getInside(pessoa);
                return fora;
            }
        }
        if(this.lotacao > 0) {
            getInside(pessoa);
            this.lotacao--;
        }
    }


    else if( guest.size() < tf){
        guest.add(pessoa);
        state = true;
        this.lotacao--;
        this.t--;
        getInside(pessoa);
    }

    System.out.println("Entrada de "+pessoa+" na guest -> "+state);
    System.out.println("guestlist: "+this.guest+"left:"+this.t);
    System.out.println("inscritos: "+this.inLocais+"left:"+this.lotacao);
```

Este método receber, como previsto, um objeto do tipo convivioDei, para inscrição na Guest list.

Começamos por verificar se a lotação da lista é zero, se a lotação é menor que a lotação_F e se o tamanho da guest list é máximo. Ou então, verificamos, se ambos atingiram os comprimentos máximos e se o objeto tem perfil boémio. Isto para cobrir o caso de haver uma pessoa não boemia a preencher o lugar na guest list e dar aso à substituição somente para pessoas com perfis boémios. Só assim poderão passar por esta condicional.

Se este caso não se verificar é porque ainda existem espaços vagos, portanto podemos adicionar pessoas sem restrições e decrementar os valores normalmente.

Caso contrário, vamos verificar do último para o primeiro, a existência de não boémios na guest list para haver a mudança. Se se verificar este caso, adicionamos o boémio na posição do não boémio e tratando-se de listas ligadas, como este node vai apontar para o próximo que é o que pretendemos retirar, fazemos o remove(i+1). No entanto este caso só funcionará numa condição de ainda existir lugar na lista geral, não tendo que retirar quaisquer pessoas deste local.

No caso da guest list estar cheia e a lista geral também o estar, verificamos na mesma se o utilizador é boémio, se o for prosseguimos. Neste caso, se encontrar um não boémio procede à eliminação da mesma forma, contudo, também terá de haver a entrada deste novo boémio no local com lista geral cheia. Para isso, introduzimos o novo boémio que teve entrada na guest list, na ultima posição possível da lista geral (o seu comprimento, lotação_f) e retiramos novamente o índice i+1 que foi o ultimo inscrever-se neste local e que não teve entrada na guest List.

Se o ultimo for boémio então é sempre afirmativo que a guest List está cheia assim com o local em si.

Dizemos o objeto convivioDei que representa aquele que saiu em ultimo é o que acabamos de remover do local cheio. E procedemos à eliminação dos dados referentes ao local no utilizador que saiu. Ficando como se nunca tivesse entrado.

As restantes subclasses funcionam todas com base no que já foi explicado.

Passemos à classe baseDadosDei:

Esta classe é meramente responsável pela gestão da informação em ficheiros, tanto que trabalha por composição, utilizando a classe Ficheiros, para aceder aos seus métodos.

Vizualizemos.

```
package com.AndreL;

import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;

public class baseDadosDei {

    protected List<String[]> dei = new ArrayList<>();
    protected List<String[]> nomes = new LinkedList<>();
    protected List<Locais> locais = new LinkedList<>();
    protected List<convivioDei> insc = new LinkedList<>();
    Ficheiros fich = new Ficheiros();

    public baseDadosDei(){...}

//    Pessoas

    public void readFrom() {...}

    public void storeIn(convivioDei inscrito){...}

    public List<convivioDei> getInscritos() { return insc; }

    public List<convivioDei> readFromOb(){...}

    public void replaceInsc(convivioDei pessoa){...}

//    Locais

    public void storeLoc(Locais loc){...}

    public List<Locais> readFromObLocals(){...}

    public void sortList(){...}

}
```

Esta classe lê e escreve em ficheiros, de acordo com o que guardou nas listas que guardam os diferentes objetos. Por exemplo, para inserir um novo local o ficheiro de objetos que o representa, a base de dados lê o ficheiro de objetos, faz o upload da informação para a lista, insere lá o objeto e volta a escrever, de raiz, no ficheiro de objetos, isto aplicado pela função **storeLoc**.

O mesmo é aplicado para os objetos do tipo convivioDei, aplicado pela função **storeInsc**.

Antes de aplicar estas funções terá de ser aplicada sempre a função **read** de cada tipo.

A função representada **sortList**() faz parte do método **atualizaData** presente na classe **mainMenu**, para que sempre que hajam alterações no numero de inscritos em cada local, esta função possa fazer o **sort** devido a cada lista.

A classe Ficheiros por sua vez, só existe para escrever e ler ficheiros, obtendo assim os métodos capazes do fazer para cada tipo de objeto deste projeto.

```
package com.AndreL;

import java.io.*;
import java.lang.reflect.Array;
import java.util.*;


public class Ficheiros implements Serializable{

//   ---------------------------------------PESSOAS---------------------------------

//    ler ficheiro texto inscritos

    public List<String[]> readFile(String fileName, List<String[]> lista){...}

//    escrever ficheiro texto inscritos

    public void writeFile(String fileName, convivioDei pessoa ){...}

//    ler ficheiro de objetos inscritos

    public void writeFileOb(String fileName, List<convivioDei> ob) {...}

//      ler ficheiro de objetos inscritos

    public LinkedList<convivioDei> readFileOb(String fileName) {...}

//    Elimina inscrito de ficheiro de texto

    public void removeInscrito(String fileName, String id){...}

//    ---------------------------------------LOCAIS---------------------------------

//  escrever ficheiro texto Locais

    public void writeFileLocals(String fileName, Locais local ){...}

//    ler ficheiro de objetos Locais

    public void writeFileObLocals(String fileName, List<Locais> ob) {...}

//      ler ficheiro de objetos Locais

    public LinkedList<Locais> readFileObLocals(String fileName) {...}
}
```

Estes métodos são utilizados independentemente ou em conjunto, por todo o programa, visando estabelecer a melhor atualização de dados possível e singular, com isto querendo dizer que separamos métodos para que fosse tudo feito de forma mais simples e clara, em vez que unir tudo num grande método que poderia gerar confusão e tornar a implementação menos flexível.

```java
package com.AndreL;

import java.util.*;

public class mainMenu {

    private Scanner scan = new Scanner(System.in);
    private convivioDei novo;
    private convivioDei repl;
    private Locais local;
    private Ficheiros fich = new Ficheiros();
    private baseDadosDei base;

    List<String> inDei;

    public mainMenu() {...}

    // Gerimento de inscritos

    public void getIn(String nome, String numero, String mail, String car, String pass,String tipo,String perfil){.

    public List<String> checkExistance(String id){...}

    public int checkin(String id_n, String pass){...}

    public int loginfo(String numero, String pass){...}

    public void atualizaData(){...}

    public int selectPlaces(int x) {...}

    public void desinscreve(){...}

    public int checkup(int x){...}

    public int removeLocal(int x){...}

    // Gerimento de Locais

    public void addLocal(){...}

    // Gestão do Programa

    public Scanner getScan() { return scan; }

    public convivioDei getNovo() { return novo; }

    public convivioDei getRepl() { return repl; }
```

A classe a cima é responsável pela parte que se concentra em alterar e atualizar os dados. É esta função que chama todas as outras, ou seja, por composição, esta classe está interligada com todas até com a interface gráfica. Começamos por, no construtor desta classe, inicializar as bases de dados, para que o programa fique pronto a trabalhar já com as listas de locais e pessoas. Chamamos também o método atualizaData, que basicamente sincroniza o objeto com qual estamos a fazer login com um hipotético que já tenha sido criado, não havendo objetos com o mesmo conteúdo e com id's diferente a circular. Assim garantimos a unicidade do objeto que estamos a trabalhar. Este método é o que chama o método sortList() anteriormente explicado.

```java
public mainMenu() {
    base = new baseDadosDei();
    base.readFromOb();
    base.readFromObLocals();
    System.out.println(base.locais);

    try{
        atualizaData();
    }catch(NullPointerException e){
        e.fillInStackTrace();
    }

}
```

Para proceder ao login/inscrição são recebidos o id e a palavra passe e disto prosseguimos à verificação da existência da pessoa na base dados do dei por verificação de id de cada pessoa com aquele que nos foi fornecido no login. Se houver algum id igual na base de dados então a existência é verificada e retorna um valor que servirá para outra verificação mais à frente.

```
public List<String> checkExistance(String id){

    List<String> personal = new ArrayList<>();
    for(String[] w: base.dei) {
        for (int i = 0; i < w.length; i++) {
            if (w[i].equals(id)) {
                System.out.println("O seu id é existente na comunidade do DEI");
                for (int p = 0 ; p < w.length ; p++){
                    personal.add(w[p]);
                }
                return personal;
            }
        }
    }
    System.out.println("O seu id é inexistente na comunidade do DEI");
    return personal;
}
```

Em seguida verificamos se já está inscrito ou não, basicamente por confirmando se o id e a pass já existência na base de dados. Caso não existam retorna um valor que quando devolvido a outro método obriga a inscrição. Se existir a password e o id em simultâneo atribuímos ao utilizador o objeto que lhe compete. Os erros estão cobertos nesta instância por isso, desconhecidos do DEI não entram.

```
public int checkin(String id_n, String pass){

    int true_info = 0;

    for(convivioDei w:base.insc){
        if((w.id.equals(id_n) && w.password.equals(pass))){
            true_info = 2;
            this.novo = w;
        }else if((w.id.equals(id_n) && w.password != (pass))){
            true_info = 1;
            System.out.println(" a");

        }else if(w.nome.equals(id_n) && w.password != (pass)){
            System.out.println("Este utilizador já está inscrito!");
            return 3;
        }
    }

    if(true_info == 2) {
        System.out.println("Login efetuado com sucesso.");
        System.out.println(novo.id);
        return 1;
    }else if(true_info == 1){
        System.out.println("A palavra-passe ou id estão incorretos");
        return -1;
    }else
        System.out.println(" incorretos");

    return 0;
}
```

Consequentemente de existir o id mas não uma palavra passe que o envolva passa diretamente à inscrição que é meramente uma função que recebe as informações do novo inscrito e cria o objeto referente ao mesmo. Vejamos.

```java
public void getIn(String nome, String numero, String mail, String car, String pass,String tipo,String perfil){

    convivioDei novo;
    switch (tipo){
        case "Professor":
            novo = new professor(car,pass,nome,numero,perfil,mail);
            base.storeIn(novo);
            System.out.println("Foi inscrito no convivio!");
            checkin(numero,pass);
            break;
        case "Aluno":
            novo = new aluno(car,pass,nome,numero,perfil,mail);
            base.storeIn(novo);
            System.out.println("Foi inscrito no convivio!");
            checkin(numero,pass);
            break;
        case "Auxiliar":
            novo = new auxiliar(car,pass,nome,numero,perfil,mail);
            base.storeIn(novo);
            System.out.println("Foi inscrito no convivio!");
            checkin(numero,pass);
            break;
        default:
            System.out.println("Nada foi inserido");
            break;
    }

}
```

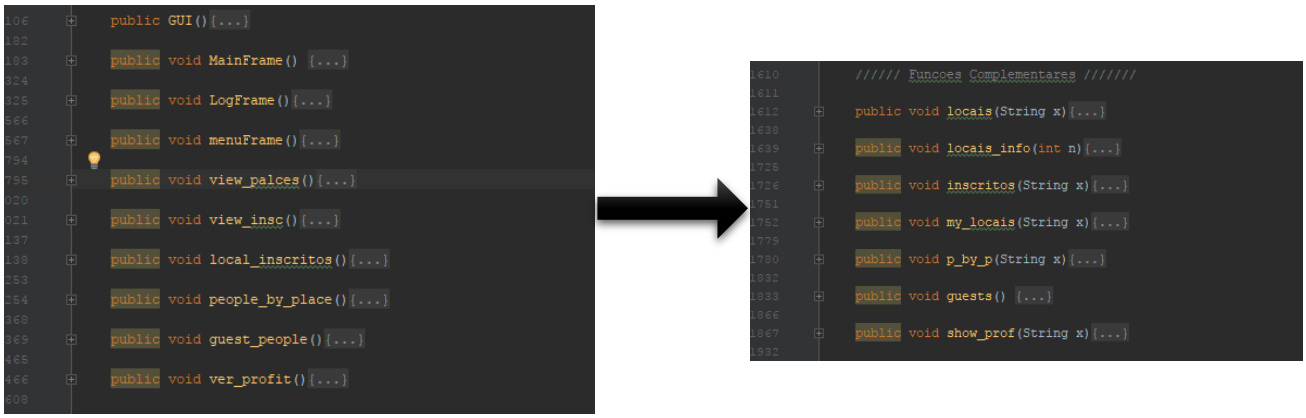
Todos estes métodos são controlados e chamados pelo método **logInfo** que trata de aplicar as condicionais devidas ao retorno de valores de cada método referente a esta instância.

```java
// Manipulação de dados

public void atualizaData(){...}

public int selectPlaces(int x) {...}

public void desinscreve(){...}

public int checkup(int x){...}

public int removeLocal(int x){...}
```

Estes métodos são os métodos que tratam de modificar os dados e atualizá-los a todo o tempo.

As funções destes métodos são especificas a cada um.

Em último e não menos importante, existe a classe **GUI**, que tanto gere a interface gráfica como gere toda a informação que lá aparece.



No que toca a estas duas instâncias da classe, a da esquerda é completamente dependente da direita, uma vez que as funções á esquerda são as frames que são responsáveis por dispor a informação que as funções á direita geram. Estas funções complementares servem para acederem à base de dados e retirarem as informações necessárias e restritas a partir de condicionais que as aprovam. Por exemplo, a função **view_insc** que serve somente para dispor os inscritos numa **frame**, vai ter que chamar a função **inscritos** para dispor qualquer tipo de informação. Ou seja há sempre uma relação de dependência entre os métodos das **frames** e os métodos que geram informação e guardam-na. Passemos a um exemplo de implementação destas funções.

```
public void inscritos(String x){

    int comp = gestor.getBase().insc.size();
    DefaultListModel listModel = new DefaultListModel();

    if(x.equals("Alunos")){
        x = "aluno";
    }else if(x.equals("Professores")){
        x = "professor";
    }else if(x.equals("Auxiliares")){
        x = "auxiliar";
    }else{
        x = "todos";
    }

    for(int i = 0 ; i < comp ; i++) {
        if(gestor.getBase().insc.get(i).getClass().toString().contains(x)){
            listModel.addElement(gestor.getBase().insc.get(i).nome+": "+gestor.getBase().insc.get(i).perfil);
        }else if(x.equals("todos")){
            listModel.addElement(gestor.getBase().insc.get(i).nome+": "+gestor.getBase().insc.get(i).perfil);
        }
    }
    this.lista_insc.setModel(listModel);

}
```

Este método (**inscritos**) recebe uma **string** que é fornecida pela frame e um dos seus componentes, e processa a informação da string de modo a saber o que há de retornar à própria frame. A aplicabilidade deste método é somente para ir à base de dados , que já foi inicializada no construtor desta classe, e obter as informações convenientes ao processo.

Gestor
#List Comunidade : ArrayList<Comunidade_Dei>
+Cria_Menu() : void
+Cria_Convivio() : boolean
+Cria_Inscricao() : boolean
+LogIn() : boolean
+Desincreve() : boolean
+Show_Rev() : void
+Show_Locais() : void
+Show_Inscritos() : void
+File_Read() : void
+File_Format() : void
+Insere_Local() : void
+Elimina_local() : void
Alunos
-curso : String
+get_info() : String
Professor
-categoria_profissional : String
+get_info() : String
Funcionario
-regime : String
+get_info() : String
Comunidade_DEI
#Nome : String
#Perfil : String
#E_mail : String
#Password : String
#List visitas : LinkedList<Locais>
+get_info() : String
+Seleciona_local()
+Elimina_local()
BaseDadosConvivio
#List Inscrito : LinkedList<Comunidade_dei>
#List Locais : LinkedList<Locais>
#List inscricao_locais : LinkedList<Comunidade_Dei>
#List Guest : LinkedList<Bares>
#List Receita_Local : HashMap<Locais>
#receita_total : float
+Get_GuestList() : List
+Get_Locais() : void
+Get_Inscritos() : void
+Calculate_Receita() : float
Guest_List
#List Guest : LinkedList<Comunidade_Dei>
#n_convidados : int
+Put_Guest() : void
+Switch_Guest() : void
+Check_Perfil() : void
Bares
-Consumo_min : float
+criaguest() : void
+get_info_d() : float
Locais
#Coordenadas : double
#Lotacao : double
#Horario : String
+Show_places()
+Create_new()
+get_info_s()
+get_info_d()
A.Desportiva
-Tipo_desporto : String
+get_info_s() : String
Jardins
-Area : String
+get_info_s() : String
Parques
+get_info_s() : String
Expo
-Arte : String
-Entranda : float
+get_info_s() : String
+get_info_d() : float
1
1..*
0
1..n

aluno
-curso : String
-promo : Double = 0.1
+getCurso() : String
+getPromo() : Double
+aluno()
+inscreve() : void
+addLocal(Locais x) : void
profesor
-catProfesional : String
+getCatProfesional() : String
+profesor()
+addLocal(Locais x) : void
+inscreve() : void
auxiliar
-regime : String
+getRegime() : String
+addLocal(Locais x) : void
+inscreve() : void
GUI
-attribute
-MainFrame() : void
-LogFrame()
-menuFrame() : void
-viewPieces() : void
-view_insc() : void
-local_inscritos() : void
-people_by_place() : void
-guest_people() : void
-ver_profil() : void
-locais(String) : void
-locais_info(int) : void
-inscritos(String x) : void
-my_locais(String) : void
-p_by_pl() : void
-guests() : void
-show_prof() : void
conviteDei
#perfil : String
#password : String
#id : String
#nome : String
#mail : String
#fich : Ficheros
+conviteDei()
+inscreve()
+addLocal()
mainMenu
-base : baseDadosDei
-fich : Ficheros
-repi : conviteDei
-novo : conviteDei
-local : Locais
#mainMenu()
#getIn(String nome, String numero, String mail, String car, String pass, String tipo, String perfil) : void
#checkExistence(String id) : List<String>
#checkIn(String id_n, String pass) : int
#loginIn(String numero, String pass) : int
#actualizaDate() : void
#selectPieces(int x) : int
#desinscreve() : void
#checkup(int x) : int
#addLocal() : void
#removeLocal(int) : int
baseDadosDei
#dei : ArrayList<String[]>
#nomes : LinkedList<String[]>
#locais : LinkedList<Locais>
#insc : LinkedList<conviteDei>
#fich : Ficheros
+baseDadosDei()
+readFront() : void
+storeIn(conviteDei inscrito) : void
+getInscritos() : List<conviteDei>
+readFromDb() : List<conviteDei>
+replaceInsc(conviteDei pessoa) : void
+storeLoc(Locais loc) : void
+readFromDbLocais() : List<Locais>
+sortList() : void
Ficheiros
+writeFile(String, conviteDei) : void
+readFile(String, List) : List<String[]>
+writeFileObj(String, List) : void
+readFileObj(String) : LinkedList
+removeInscrito(String, parameter) : void
+writeFileLocais(String, Locais) : void
+writeFileLocais(String, List) : void
+readFileDbLocais(String) : LinkedList
exposicao
#arte : String
#ingresso : double
#lucro : double
+getArte() : String
+getIngresso() : double
+getLucro() : double
+exposicao()
+getInside(conviteDei pessoa) : List
+howProfil(conviteDei pessoa) : void
+storeLocal() : void
jardins
-area : double
+getArea() : double
+jardins()
+getInside(conviteDei pessoa) : List
+storeLocal() : void
areaDesportiva
-desporto : String
+getDesporto() : String
+areaDesportiva()
+getInside(conviteDei pessoa) : List
+storeLocal() : void
Locais
#fich : Ficheros
#gps : String
#descricaoLocal : String
+Locais()
+getGps() : String
+getInside(conviteDei pessoa) : List
+storeLocal() : void
+howProfil(conviteDei pessoa) : void
+storeGuest(conviteDei pessoa) : conviteDei
bares
-guest : LinkedList
-consumoMin : Double
-t : int
-tf : int
-tobaco : int
-tobacoF : int
-lucro : Double
+getGuest() : LinkedList
+getConsumoMin() : Double
+getT() : int
+getTf() : int
+getLtobaco() : int
+getLtobacoF() : int
+getLucro() : Double
+bares()
+getInside(conviteDei pessoa) : List
+storeLocal() : void
+howProfil(conviteDei pessoa) : void
+storeGuest(conviteDei pessoa) : conviteDei
+restbre_t() : int
+restbre_tobaco() : int
+operation()

BREVE SUMA

Dado este relatório para fazer uma suma de conceitos, pode referir vários aspetos necessários para completar um projeto desta dimensão. Há que haver uma abstração para conseguir criar classes e métodos que sejam utilizáveis de forma momentânea, quando são necessários, é necessário um conhecimento geralmente de programação, regras de programação e conceitos específicos.

No decorrer de realização deste projeto concluímos que de maneira que prosseguimos mais próximo do objetivo há sempre algo mais a ser acrescentado e realizar um objetivo um pouco diferente que engloba o anterior e melhora-o em todos os aspetos. É necessária uma visão crítica e uma ambição que sai um pouco do papel que nos é dado.

Conseguimos absolver todos os casos que queiramos implementar e alguns extras que esperamos que façam a diferença na maneira como são vistos estes projetos.

A melhor forma de aprender é sem duvida aplicar os conceitos e aplicar sem nunca parar, contudo, estes projetos não testam só a viabilidade do aluno como programador, mas também como crítico e gostamos de pensar que neste projeto em especifico conseguimos estabelecer novos padrões.

Quanto á parte técnica do projeto, podemos verificar as possibilidades desta linguagem de programação, uma vez que já lidamos com muitas outras, tais como C ou Python ou C++, reparamos que JAVA, para além de ser uma linguagem de alto nível indicada a objetos, tenta estabelecer uma simplicidade a obter e criar estes programas. São sem dúvidas ilimitadas as possibilidades de projetos para esta linguagem e é sempre excelente concluir isto a partir da sua aplicação.