Universidade do Minho
Escola de Engenharia
Mestrado Integrado em Engenharia Informática

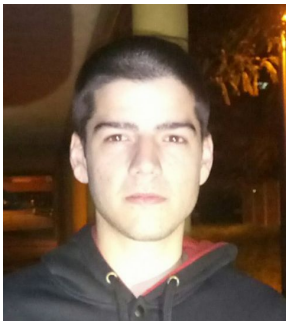
Programação Orientada aos Objectos

Ano Letivo de 2016/2017

UMer - Serviço de Transportes

Grupo 9

Manuel Maciel
A68410



Rui Santos
A67656

Índice

Resumo

Neste relatório iremos falar sobre o desenvolvimento de um serviço de transporte de passageiros. Esta aplicação tem como intuito criar uma rede onde os usuários possam escolher um determinado condutor para os ir buscar e levar ao local pretendido. Os tópicos abordados serão, a maneira como o grupo abordou certos problemas impostos pelo enunciado, a arquitectura de classes que resultou do planeamento do projeto entre outras. Por fim iremos fazer um breve tutorial de como utilizar a aplicação.

Introdução

No âmbito da cadeira de programação orientada aos objectos foi nos pedido o desenvolvimento de um serviço de táxis de nome UMer. Foi pedido também que desenvolvêssemos o serviço utilizando a linguagem JAVA através do IDE BlueJ. Este serviço terá como objectivo facilitar aos utilizadores arranjar transporte de uma sítio para o outro.

Análise do problema

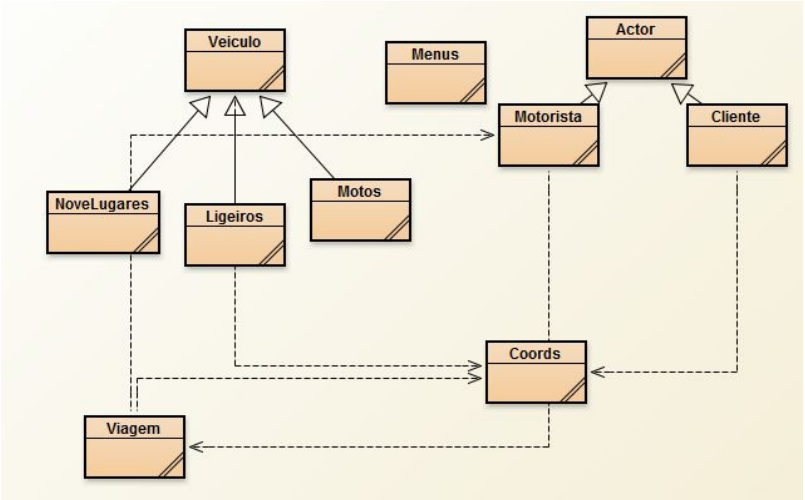
Neste tópico iremos falar sobre a análise inicial que o grupo teve sobre o problema, ou seja quais os problemas que apareceram durante a leitura do enunciado e quais as maneiras pensadas para resolver os mesmos.

Breve Descrição do enunciado

O objetivo da aplicação a desenvolver é a criação de uma rede onde existem dois tipos de utilizadores, um são os clientes, que representam os utilizadores que utilizam o sistema para solicitar e realizar viagens, e o outro são os motoristas que utilizam o sistema para serem solicitados e realizarem as viagens que o clientes pedirem. O sistema tem também como objectivo armazenar todas as viagens realizadas com o auxílio da aplicação, associado a essa viagem são guardados os intervenientes (motorista, cliente), os preços da mesma, as localizações (início e fim), e por fim a viatura que realizou essa viagem. Por último também deve ser possível guardar no sistema as viaturas que são usadas pelos utilizadores do tipo motorista.

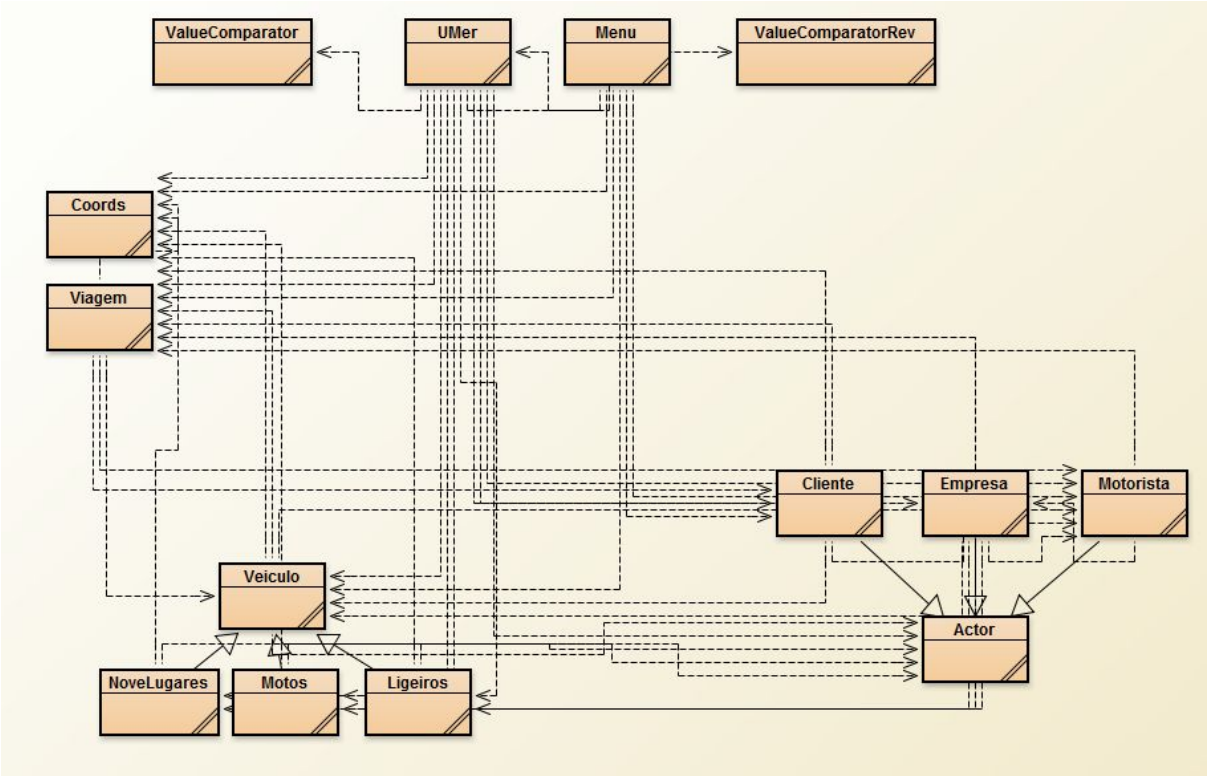
Primeira análise

Depois de lido o enunciado o grupo chegou à conclusão que eram preciso duas superclasses, uma dela é *Veiculo* visto que podem existir diferentes tipos de veículos no sistema (*Ligeiros*, *Motos*, *Novelugares*) e a outra é “Actor” que representa os diferentes tipos de utilizadores do sistema. Foi pensada uma classe “Menu” onde serão gerados todos os prints do sistema para o utilizador e por fim uma classe que represente as viagens efectuadas com o auxílio do sistema. Com isso em mente o grupo chegou ao resultado que pode ser visto na imagem
baixo.



Arquitectura de Classes

Depois de uma primeira análise ao trabalho começou-se a desenvolver o projeto, um primeiro passo foi analisar a arquitectura de classes resultante da primeira análise do enunciado e refazer pensando agora ainda mais profundamente nas classes que irão ser necessárias para o trabalho. Depois de desenvolvido o trabalho chegou-se ao seguinte resultado.



SuperClasses

O trabalho final apresenta apenas duas superclasses, “Veiculo” e “Actor”. Isto porque apenas estas podem ser divididas em subclasses, contrariamente ao que acontece no resto que não terá hierarquias.

Actor

Nesta superclasse são guardadas as variáveis comuns às três subclasses (Motorista, Cliente, Empresa)

```
public class Actor
{
    private String email;
    private String nome;
    private String password;
    private String morada;
    private LocalDate nascimento;
    public ArrayList<Viagem> listaViagens;
```

Veiculo

Nesta superclasse são guardadas as variáveis comuns às três subclasses (Ligeiro, Motos, Novelugares)

```
public class Veiculo {

    private String matricula;
    private int velMediaKM;
    private int precoPorKM;
    private int fiabilidade;
    private Coords posicao;
    private int totalFaturado;
    private boolean estado; // true o condutor está a usar o veiculo
    public ArrayList<Viagem> listaViagens = new ArrayList<Viagem>();
    public Motorista motorista = new Motorista();
```

Classes

Motorista

Na classe motorista é guardada toda a informação relacionada com os utilizadores do tipo motorista necessária para o funcionamento do serviço. Como se pode ver na imagem abaixo os atributos da classe Motorista são:

nrViagens - é um inteiro que representa o número de viagens feitas pelo motorista;
numKms - é um inteiro que representa o número de kms que o motorista já conduziu em serviço usando a aplicação;
empresa - caso o motorista trabalhe numa empresa, aqui é guardada a informação dessa empresa;
estado - é um boolean que se apresenta como verdadeiro caso o motorista esteja livre e possa ser chamado por um cliente, ou falso caso esteja já num serviço;
desvioAcumulado - é um double que representa o acumulado das diferenças do tempo estimado de viagem com o tempo final da mesma;

```
public class Motorista extends Actor
{
    private int nrViagens;
    private int numKms;
    private Empresa empresa;
    private boolean estado; //true livre, false ocupado, mais facil
    private Double desvioAcumulado;
```

Cliente

Na classe cliente é apenas guardada uma variável que é *totalGasto* esta variável representa o total que o cliente já gastou em viagens usando a aplicação.

```
public class Cliente extends Actor
{
    private Double totalGasto = 0.0;
```

Empresa

A classe empresa representa as empresas de taxis que usam os nossos serviços. Para além das variáveis de actor são também associadas a cada empresa duas listas. A primeira *motoristas* é um ArrayList de actores, ou seja, é uma lista de todos os motoristas que trabalham para a empresa. A segunda é um ArrayLyst de Veiculos (outra classe do projeto) chamado *viaturas* e é onde são guardados todos os veiculos que operam a serviço da empresa.

```
public class Empresa extends Actor {

    public ArrayList<Actor> motoristas;
    public ArrayList<Veiculo> viaturas;
```

Viagem

Nesta classe estão guardados todas as informações necessárias para armazenar todos os dados de uma viagem. As variáveis usadas são:
cliente - Representa o cliente que pediu e participou na viagem;
inicio - Representa as coordenadas onde a viagem se iniciou, mais precisamente onde o cliente entrou no veículo;
fim - Representa as coordenadas do destino da viagem;
distancia - Representa a distância percorrida na viagem desde o início até ao destino;
precoAcordado - É o valor que ficou acordado entre o motorista e o cliente para o preço da viagem;
precoFinal - É o valor que o cliente teve de pagar no final da viagem;
tempoEstimado - É o tempo que o motorista disse ao cliente que a viagem ia demorar;
tempoFinal - É o tempo que demorou a viagem;

veiculo - Representa o veiculo que foi usado para realizar a viagem em questão;
data - Ano-mês-dia em que foi realizada a viagem;
nota - Nota que o cliente deu à viagem no fim da mesma;

```
public class Viagem implements Serializable
{
    private Cliente cliente;
    private Coords inicio;
    private Coords fim;
    private Double distancia;
    private Double precoAcordado;
    private Double precoFinal;
    private Double tempoEstimado;
    private Double tempoFinal;
    private Motorista condutor;
    private Veiculo veiculo;
    private LocalDate data;
    private int nota;
```

Coords

Esta classe criou-se para evitar andar a trabalhar sempre com duas variáveis, pois usando esta classe sempre que era necessário usar as coordenadas de alguma coisa apenas era preciso uma variável que representa a coordenada X e a coordenada Y.

```
public class Coords implements Serializable
{
    private int x;
    private int y;
```

Menu

Esta classe funciona como interface entre os diversos utilizadores (empresas, motoristas e clientes) e o back-end do programa. Como podemos ver em baixo, todos os menus da aplicação são feitos nesta classe.

```
public class Menu {
    // variáveis de instância
    private List<String> opcoes;
    private String[] menuPrinc = {"LogIn", "Registar", "Estatistica", "Povoar", "Gravar"};
    private String[] menuCliente = {"Realizar Viagem", "Ver Viagens Efectuadas"};
    private String[] menuViagem = {"Viatura mais próxima","Escolher viatura"};
    private String[] menuEstatistica = {"Top 10 clientes gastadores", "Piores 5 motoristas",
        "Total facturado por uma empresa", "Total facturado por um veiculo"};
    private String[] menuMotoristaComEmpresa = {"Associar-se a uma viatura","Ver Viagens Efectuadas",
        "Mudar o estado", "Libertar Carro"};
    private String[] menuMotoristaPrivado = {"Registar Nova Viatura", "Ver Viagens Efectuadas",
        "Associar-se a uma empresa", "Mudar o estado"};
    private String[] menuEmpresa = {"Registar Nova Viatura","Ver Frota", "Ver Viagens Efectuadas"};
    private static final String OBJECT_FILE = "umerTaxis.obj";

    private int op, esc;
    public UMer umer;
    public Scanner escolha;
```

UMer

Esta classe é a principal da nossa aplicação, pois é na UMer que todos os processos para responder às queries do enunciado são feitos. Nesta classe são guardadas 3 estruturas, um inteiro, e uma variável do tipo *Actor*. A primeira estrutura é uma HashMap chamada *listaCliente*, que tem como chave o email do Actor e como valores o próprio actor, desta maneira podemos saber sempre quais os actores no sistema. A segunda é outro HashMap, *listaVeiculo*, neste é guardado todos os veiculos que já passaram pelo sistema, e usa como chave para os valores a matricula do veiculo. A ultima estrutura é também uma Hashmap de nome *listaViagens* e é onde estão guardadas todas as viagens realizadas atrávez do uso desta aplicação, a chave desta HashMap é um inteiro que corresponde ao ID da viagem em questão. Por último e de modo a facilitar a resolução de algumas queries, temos uma variável do tipo *Actor* de nome *currentUser* que corresponde ao utilizador que está neste momento em utilizar a aplicação.

```
public class UMer implements Serializable{

    private HashMap<String, Actor> listaCliente = new HashMap<String, Actor>();
    private HashMap<String, Veiculo> listaVeiculo = new HashMap<String, Veiculo>();
    private Map<Integer,Viagem> listaViagens = new HashMap<Integer,Viagem>();
    private int idViagem = 0;
    private Actor currentUser;
```

Estruturas de dados utilizadas

Para a realização deste trabalho foram usadas sete estruturas de dados, três na classe *UMer*, duas na classe *empresa*, uma em *Veiculo* e outra em *Actor*.
Na classe *UMer* as três estruturas usadas são maps, isto porque o grupo tinha como objectivo mapear o valor do que queriamos armazenar com uma chave identificadora do objecto em questão.
Na classe *Empresa* usamos duas estrutura ambas do tipo List, o grupo tomou a decisão de utilizar em ambos os casos List, pois aquilo que era necessário representar era uma lista tanto de *Veiculos* como de *Motoristas.*
Na classe *Veiculo* a estrutura usada é do tipo List, pois tal como em *Empresa* o objectivo era apenas fazer uma lista de viagens que o veiculo tenha feito.
Por fim em *Actor* usamos uma List exatamente com o mesmo intuito da List usada em *Veiculo,* aquilo que era necessário guardar em List era apenas uma lista de viagens realizadas pelos utilizadores da aplicação.

Resolução das Queries

Todas os métodos responsáveis por resolver as queries estão escritos na classe *UMer*, pode por vezes serem utilizados outros métodos de outras classes, mas apenas como papel mais secundário.

Registo de um utilizador:

Para se registar o utilizador tem de dizer qual é o tipo de utilizador que deseja ser (*Empresa, Cliente, Motorista)*, esse tipo é passado ao método “register” através da variável *tipoReg*, depois usando o switch o métodos regista o utilizador consoante o tipo escolhido

```
public void register(String email, String nome,String password, String morada, LocalDate dataRecebida, int tipoReg) {
    Cliente tempC;
    Motorista tempM;
    Empresa tempE;

    switch(tipoReg){
        case 1:
            tempC = new Cliente(email,nome, password, morada, dataRecebida);
            listaCliente.put(email, tempC);
            break;
        case 2:
            tempM = new Motorista(email, nome, password, morada, dataRecebida);
            listaCliente.put(email, tempM);
            break;
        case 3:
            tempE = new Empresa(email, nome, password, morada, dataRecebida);
            listaCliente.put(email, tempE);
            System.out.println(tempE.toString());
            break;
    }
}
```

Login de um utilizador:

Para um utilizador entrar no serviço tem de fornecer o email e password que usou durante o registo. Depois o método “login” vê a classe dos dados que o utilizador forneceu, e depois retorna um inteiro para o método de login na classe *menu* saiba qual o menu a fornecer. Caso o utilizador seja um motorista é necessário saber se é privado ou trabalha para uma empresa, para isso recorre-se ao método “getTipoMotorista”.

```java
public int login(String email, String pass) {
    Actor temp;
    temp = listaCliente.get(email);
    if (temp != null) {
        if (temp.logIn(pass)) {
            this.currentUser = temp;
            if(temp.getClass().getSimpleName() == "Cliente") return 1;
            if(temp.getClass().getSimpleName() == "Motorista") return 2;
            if(temp.getClass().getSimpleName() == "Empresa") return 3;
            return 1; //ok
        } else {
            return 0; //pass mal
        }
    } else {
        return -1;//nao existe user
    }
}
```

```java
public int getTipoMotorista(String user){
    Motorista m = (Motorista) this.currentUser;
    if( m.getEmpresa() == null) return 0;
    return 1;
}
```

Criar e inserir viatura:

Apenas os motoristas privados e as empresas têm acesso a esta funcionalidade. Para isso o utilizador tem de fornecer os dados pedidos pelo programa, sendo um deles o tipo de veiculo, visto que há três tipos na aplicação (*Ligeiros, Motos, NoveLugares)* depois o registo funciona de certa forma como o "register" do utilizador. Tivemos atenção no tipo de utilizador que registava o veiculo, pois se fosse uma empresa o *motorista* é "null" para que alguém da empresa possa usar a viatura, e se for um condutor privador, é necessário fazer com que a sua viatura antiga fique com o motorista "null".

```java
public int registarNovaViatura(String matricula, int tipoVeiculo, int velMediaKm, int precoPorKM, Coords c){
    NoveLugares vNL;
    Ligeiros vL;
    Motos vM;
    Empresa aux;
    Motorista m;

    switch(tipoVeiculo){
        case 1:
            if(this.currentUser.getClass().getSimpleName() == "Empresa"){
                vL = new Ligeiros(matricula, velMediaKm, precoPorKM,c);
                this.listaVeiculo.put(matricula,vL);
                aux = (Empresa) this.currentUser;
                aux.addVeiculoLigeiros(vL);
                return 1;
            }
            else{
                m = (Motorista) this.currentUser;
                vL = new Ligeiros(matricula, velMediaKm, precoPorKM, c,this.currentUser);
                for(Veiculo v : this.listaVeiculo.values()){
                    if(v.getMotorista()!=null)
                        if(v.getMotorista().getEmail().equals(m.getEmail())) v.removeMotorista();
                }
                this.listaVeiculo.put(matricula, vL);
                return 2;
            }
```

Associar motoristas a veículos

Na aplicação apenas permitimos utilizadores que trabalhem numa empresa a registar-se a uma veículo dessa empresa. Mesmo que o veículo tenha um motorista um outro motorista podesse associar a esse veiculo, a única condição é que o estado do veículo seja “true” ou seja, apesar de ter um motorista o veículo está parado na central da empresa.

```
public int motoristaAssociarAVeiculoDaEmpresa(String matricula){
    int res;
    Empresa e;
    Veiculo v;
    Motorista aux = (Motorista) this.currentUser;
    if(!this.listaVeiculo.containsKey(matricula)) return -1;

    e = aux.getEmpresa();

    v = e.getVeiculo(matricula);

    if(v == null) return 0;

    if(v.getEstado() == true    ) return -1;

    for(Veiculo vei : this.listaVeiculo.values()){
        if(vei.getMotorista()!=null)
            if(vei.getMotorista().getEmail().equals(aux.getEmail())) vei.removeMotorista();
    }

    v.setMotorista(aux);


    return 1;
}
```

Associar o motorista a uma empresa

Um motorista sem empresa pode associar-se a uma empresa apenas fornecendo o nome da empresa, no entanto fica impedido de utilizar o carro privado durante viagens usando a aplicação.

```
public void assciarCondutor(String nome){ //associa a empresa
    Motorista m = (Motorista) this.currentUser;
    Empresa empresa = (Empresa)listaCliente.get(nome);
    empresa.adicionarMotorista(m);
    m.setEmpresa(empresa);
    System.out.println(empresa);
    System.out.println(m);
    libertarCarro();
}
```

O utilizador pode ver as viagens efectuadas entre datas

Apenas os clientes e os motoristas podem utilizar esta funcionalidade. A única coisa que é necessário fazer é fornecer as datas limites e o método envia todas as viagens feitas nessa altura.

```
public ArrayList<Viagem> procuraEntreDatas(LocalDate dataInicial, LocalDate dataFinal){
    ArrayList<Viagem> tempV = new ArrayList<>();
    ArrayList<Viagem> tempVf = new ArrayList<>();

    tempV.addAll(this.currentUser.getListaViagens());
    for(Viagem v : tempV){
        if (v.getData().isAfter(dataInicial) && v.getData().isBefore(dataFinal)){
            tempVf.add(v);
        }
    }
    return tempVf;

}
```

Indicar o total facturado

Apenas os “admins” da aplicação têm acesso a esta funcionalidade, para isso é necessário aceder ao menu “estatisticas” utilizando uma password e depois fornecendo o nome de quem queremos procurar a aplicação corre as viagens todas desse utilizador e vai somando os valores finais.

```
public Double totalFaturadoVeiculo(String veiculo, LocalDate after, LocalDate before){
    Veiculo v = listaVeiculo.get(veiculo);
    Double total = 0.0;
    ArrayList<Viagem> lista =  v.getListaViagens();

    for (Viagem aux : lista){
            if (aux.getData().isBefore(before) && aux.getData().isAfter(after))
                total+=aux.getPrecoFinal();
    }
    return total;
}
```

```
public Double totalFaturado(String empresa,LocalDate after, LocalDate before){
    Empresa e = (Empresa) listaCliente.get(empresa);
    Double total = 0.0;
    ArrayList<Veiculo> lista = e.getViaturas();
    for(Veiculo v : lista)
        total+=totalFaturadoVeiculo(v.getMatricula(),after,before);

    return total;
}
```

10 clientes mais gastadores

Tal como a funcionalidade anterior apenas os “admins” podem aceder a esta funcionalidade. Utilizando uma estutura map auxiliar (SortedMap) os Clientes são copiados do map *listaCliente* para o tal map auxiliar, tendo como chave o próprio cliente e como valor o total gasto. Sempre que algo é inserido neste SortedMap este organiza-se usando um comparator que comparando dos valores do total gasto vai metendo nas posições iniciais os utilizadores com o maior valor.

```
public Map<Cliente,Double> top10clientes () {
    Cliente aux = new Cliente();
    Map<Cliente, Double> temp = new HashMap <Cliente, Double>();

    for (Actor a : this.listaCliente.values()){
        if (a.getClass().getSimpleName() == "Cliente"){
            aux = (Cliente) a;
            temp.put((aux), (Double)aux.getTotalGasto());//mapa com cliente e valor gasto total
        }
    }

    Map<Cliente,Double> sortedMap = sortByValueCliente(temp);

    return sortedMap;
}
```

```
private Map<Cliente,Double> sortByValueCliente(Map<Cliente,Double> unsortedMap) {
    Map<Cliente,Double> sortedMap = new TreeMap<Cliente,Double>(new ValueComparator(unsortedMap));

    sortedMap.putAll(unsortedMap);
    return sortedMap;
}
```

```
public class ValueComparator implements Comparator {
    Map map;

    public ValueComparator(Map map) {
        this.map = map;
    }

    public int compare(Object keyA, Object keyB) {
        Comparable valueA = (Comparable) map.get(keyA);
        Comparable valueB = (Comparable) map.get(keyB);
        if (valueB.compareTo(valueA)==0) return 1;
        return valueB.compareTo(valueA);
    }
}
```

Esta ultima class é genérica, por isso é reutilizada mais à frente na funcionalidade de piores motoristas, e para ajudar a calcular o motorista mais perto do cliente que está a solicitar uma viagem.

Maior desvio entre valor estimado e valor facturado

Esta é mais uma funcionalidade para os administradores apenas. O funcionamento desta funcionalidade é praticamente o mesmo da funcionalidade descrita em cima. Há um Map auxiliar que vai guardando como chave o motorista e como valor o desvioTotal, depois vai ordenando metendo nas posições iniciais os condutores com o desvio maior.

```
public Map<Motorista,Double> piores5condutores () {
    Motorista aux = new Motorista();
    Map<Motorista, Double> temp = new HashMap <Motorista, Double>();

    for (Actor a : this.listaCliente.values()){
        if (a.getClass().getSimpleName() == "Motorista"){
            aux = (Motorista) a;
            temp.put((aux), (Double)aux.getDesvioAcumulado());//mapa com cliente e valor gasto total
        }
    }

    Map<Motorista,Double> sortedMap = sortByValueMotorista(temp);

    return sortedMap;
}
```

Realizar uma viagem

Este é o método que realiza a viagem em si. Começa por calcular a distância a viajar, bem como o preço e tempo estimado. De seguida, através um de um factor de randomização, calculamos o tempo real que demora a viagem, e fazemos os ajustes de preço de acordo com
o desvio. Por fim fazemos a inserção da viagem na lista e chamamos os métodos de inserção auxiliares, que vão atualizar os campos necessários para as nossas estatísticas.

```
public int realizarViagem(String matricula, Coords inicio, Coords fim){
    Veiculo v;
    Motorista m;
    Cliente aux = (Cliente) this.currentUser;
    double precoAcordado;
    double dist;

    dist = inicio.distancia(fim);


    v = listaVeiculo.get(matricula);
    precoAcordado = dist * v.getPrecoPorKM();
    m = v.getMotorista();

    inserirViagem(aux, inicio, fim, precoAcordado, m, v);
    return idViagem;
}
```

```
public void inserirViagem(Cliente cliente,Coords inicio, Coords fim, double precoAcordado, Motorista condutor, Veiculo veiculo){


    Random rando = new Random();
    Double  rng = ((rando.nextInt(51) + (Double)85.0)/100.0); //de 0.85 a 1.35
    Double distancia = inicio.distancia(fim); //Double//discantica por calcular funçao manu
    LocalDate data = LocalDate.now(); //current date
    Double desvio;

    Double precoFinal = precoAcordado;
    Double tempoEstimado = distancia*veiculo.getPrecoPorKM();
    Double tempoFinal = tempoEstimado * rng;

    desvio = Math.abs(tempoEstimado-tempoFinal)/tempoEstimado;

    if (desvio<=0.25){
        precoFinal = precoAcordado * desvio;
    }


    Viagem nova = new Viagem(cliente, inicio, fim, distancia, precoAcordado, precoFinal,tempoEstimado, tempoFinal, condutor, veiculo, data, -1);
    idViagem++;
    listaViagens.put(idViagem,nova);

    cliente.addViagem(nova);
    condutor.addViagem(nova);
    veiculo.addViagem(nova);
}
```

```
public Map<Veiculo,Double> viaturasProx (Coords posicao) {
    Map<Veiculo, Double> temp = new HashMap <Veiculo, Double>();
    Double i;

    for (Veiculo aux : this.listaVeiculo.values()){

        if (aux.getMotorista() != null){           //ver se tem motorista
            Motorista motorista = aux.getMotorista();
            if (motorista.getEstado()){             //ver se motorista esta livre
                i = aux.getPosicao().distancia(posicao);
                temp.put((aux), i);//mapa com cliente e distancia
            }
        }

    }

    Map<Veiculo,Double> sortedMap = sortByValueVeiculo(temp);

    return sortedMap;
}
```

Manual de utilização

Menu principal

No menu principal são apresentadas as opções iniciais do programa. Poderá ser feito o log in na sua conta, ou registar caso não possua uma conta. Conta ainda com as opções de povoar o sistema, ver um conjunto de estatísticas sobre o mesmo, e por fim, uma opção para gravar o estado actual da aplicação para utilização futura que será automaticamente carregado.

Menu Cliente

No menu de cliente poderá efetuar uma nova viagem, especificando ou não o condutor pretendido, e também ver o registo de todas as viagens anteriores, com todos os detalhes da mesma.

Menu Motorista

Neste menu encontram-se todas as opções a ser usadas pelos condutores da nossa empresa. O condutor poderá associar a sua viatura previamente registada (ou regista-la caso seja um condutor privado), bem como elimina-la. Poderá ainda mudar o seu estado entre livre e ocupado.

Menu Empresa

Por fim temos o menu de empresa, que é uma espécie de menu de administração da empresa, o utilizador poderá registar novas viaturas na frota da empresa, ver uma lista com todas aquelas registadas na empresa, e por último ver todas as viagens efectuadas pela empresa entre duas datas.

Estatísticas

Este menu é dedicado a estatísticas gerais, entre as quais podemos ver quais sao os 10 clientes que mais gastaram na nossa aplicação, quais são são 5 condutores que apresentam maior desvio entre os valores estimados e os valores reais de espera/viagem e por fim o total faturado, quer por um carro, quer por uma empresa, num intervalo de tempo a especificar pelo utilizador.

Realizar uma viagem

Dentro do seu menu, o cliente poderá efetuar uma nova viagem. Para tal o cliente terá de dizer qual é a sua posição actual, e qual será o destino. De seguida será dada a escolha entre usar o táxi mais próximo, ou escolher dentro de uma lista dos 5 mais próximos, na qual terá acesso a mais informação sobre o veículo, custo e duração da mesma.

Conclusão e trabalho futuro

No fim do desenvolvimento do trabalho podemos verificar que ficamos aquém de alguns dos pontos inicialmente pedidos. O grupo teve dificuldades em imaginar o programa em tempo real, o que tornou a implementação de filas de espera bastante difícil, ou seja, não conseguimos implementar as mesmas. Também devido a não haver um elevado conhecimento de java o grupo pode dizer com certezas que nem todas os métodos estão otimizados da melhor maneira. No entanto há também alguns pontos positivos, o grupo conseguiu implementar com sucesso as viagens, conseguindo criar um ambiente aleatório na realização das mesmas, e estruturou o trabalho de modo a que seja fácil acrescentar novos tipos de utilizadores ou veículos.