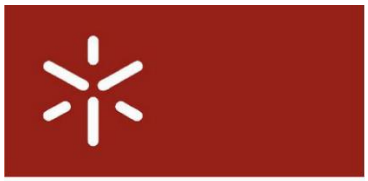


Universidade do Minho

Universidade Do Minho

Escola de Engenharia

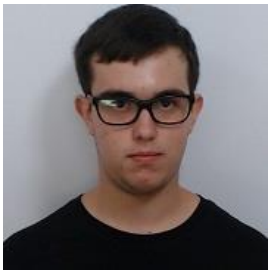
Relatório do Projeto Traz Aqui

Programação Orientada aos Objetos

LCC 2º Semestre

Junho 2020

Grupo 38



André Araújo

a87987

Henrique Ferreira

a87953

Daniel Ribeiro

a87994

Índice:

1 Introdução

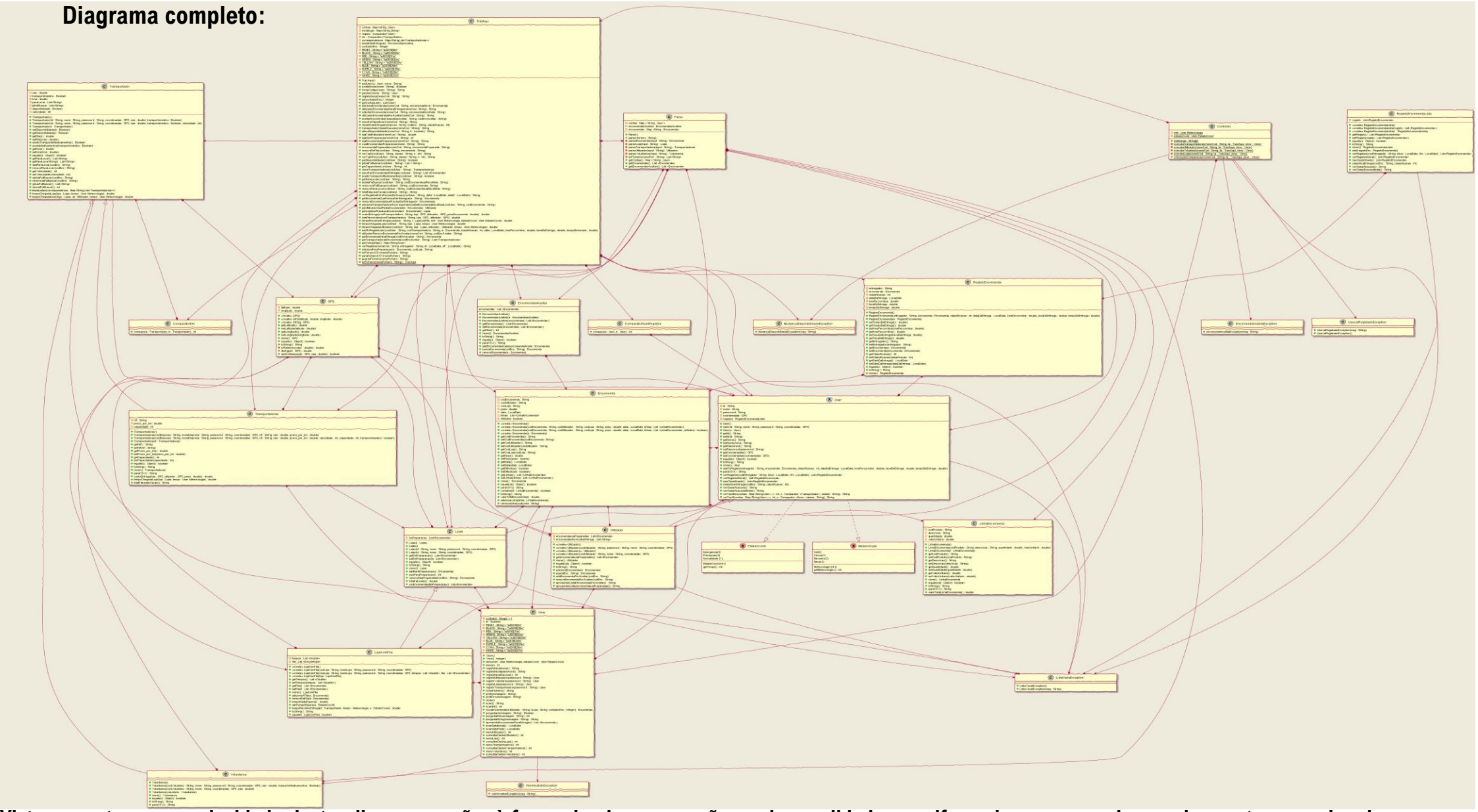
Este relatório aborda a implementação de uma aplicação capaz de gerir, encomendar e entregar encomendas. Sendo esta aplicação pensada na atuais necessidades de entregas de bens a pessoas confinadas nas suas habitações.

A aplicação tem vários tipos de Utilizadores que a podem usar, sendo implementado um sistema de registo, login e menu para cada um deles, os quatros tipos de usuários assim são: Utilizadores, Voluntários, Transportadoras e Lojas.

Estas quatro entidades funcionam entre si sendo que os Utilizadores são quem pede uma encomenda a uma determinada Loja. A Loja irá preparar essa tal encomenda, e quando estiver preparada o Utilizador pode solicitar a sua entrega e desta forma um Voluntário poderá levar a encomenda até ao Utilizador ou as Transportadoras poderão competir entre si para levar a tal encomenda, sendo o Utilizador quem têm a palavra final na aceitação de uma Transportadora, em relação ao custo, tempo de entrega e classificações anteriores dos seus serviços.

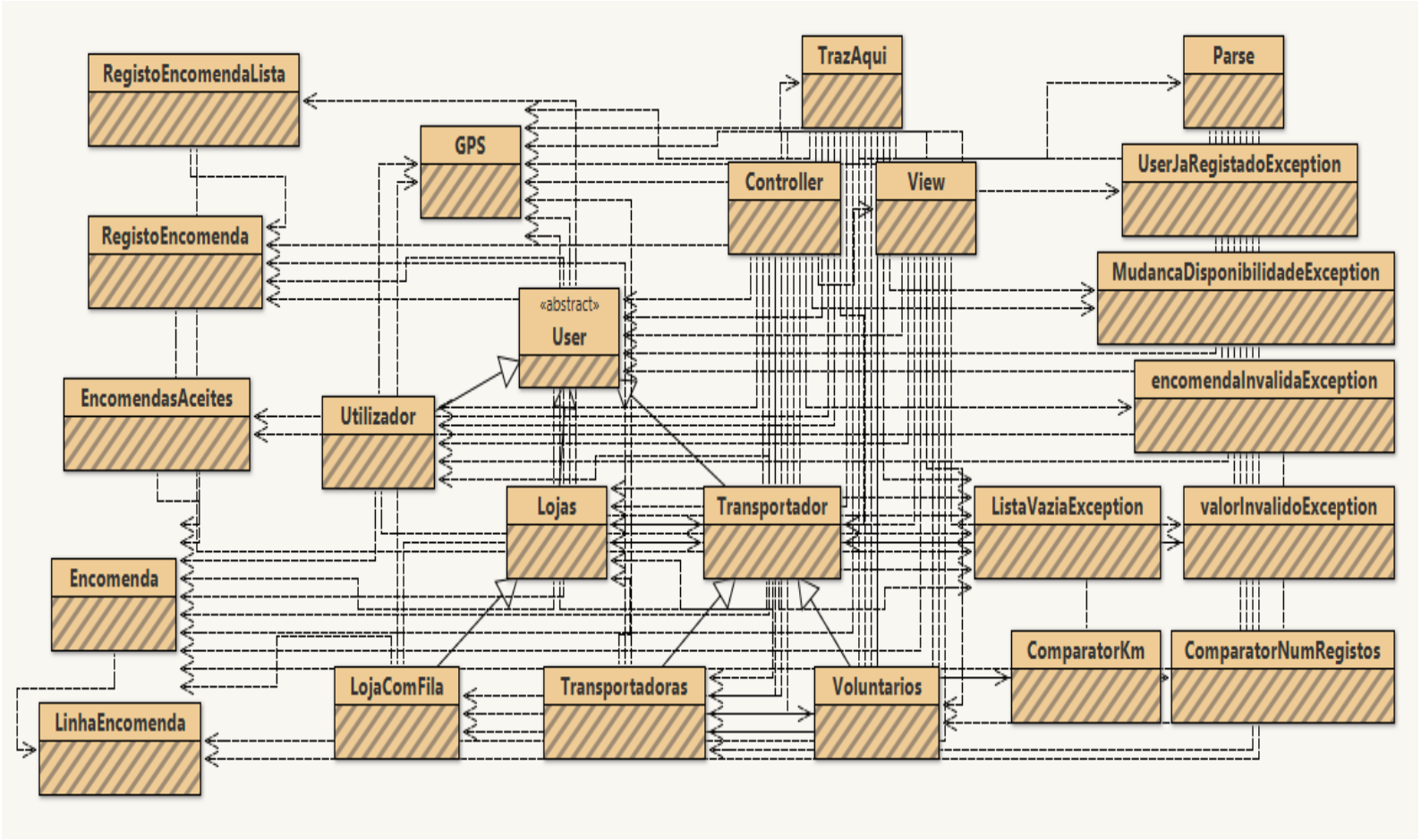
2 Arquitetura de Classes

Diagrama completo:

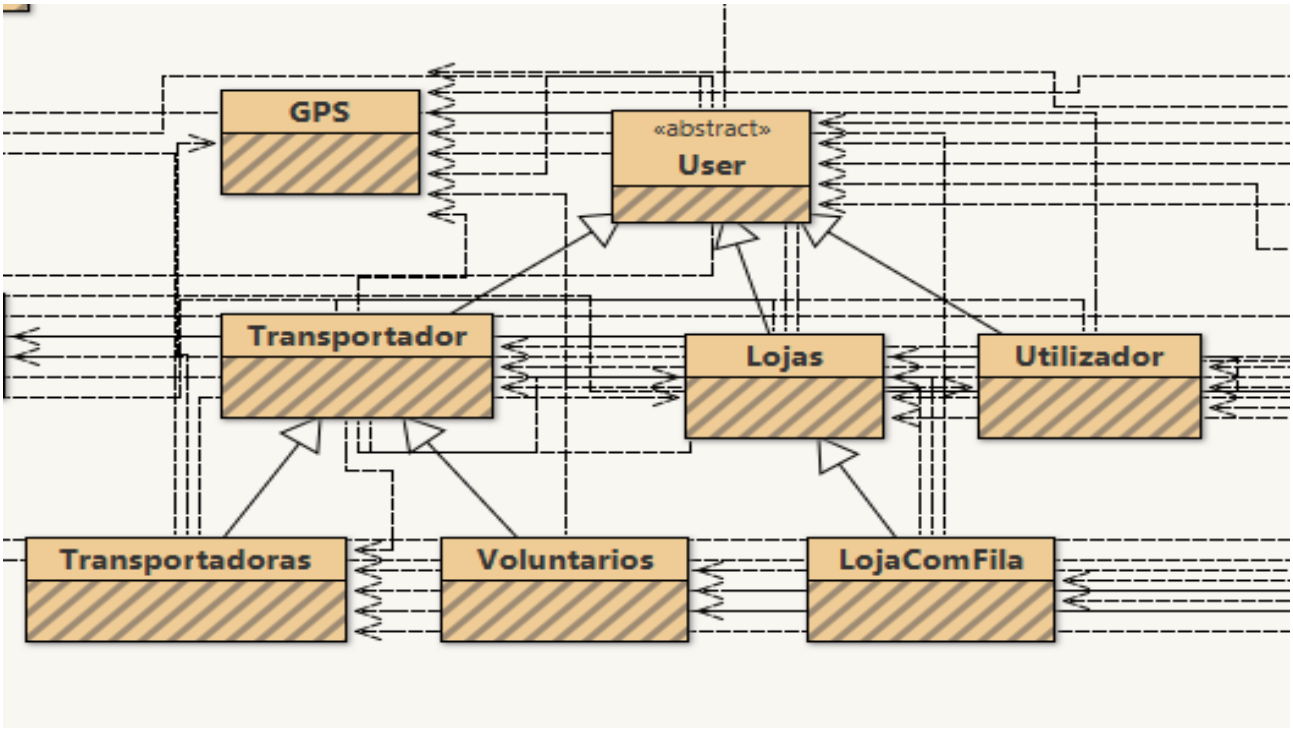


Visto a extrema complexidade deste diagrama , não à forma de ele mesmo não perder qualidade no pdf, por isso recomendamos vivamente que veja a imagem com melhor qualidade pelo seguinte link: https://cdn.discordapp.com/attachments/693157427520077905/720117494198829070/unknown.png

Em seguida para iniciar a apresentação deste trabalho decidimos incorporar um diagrama de classes simplificado (sem métodos ou variáveis) e descrever um pouco cada classe, as suas funções assim como as suas dependências de outras, sem mais demoras:



2.1 User, suas heranças e classes relacionadas



Começamos por criar a classe User, e também construtores, setters, getters, equals, clone e toString das já mencionadas anteriormente: Utilizador, Voluntários, Transportadoras e Lojas.

2.1.1 Utilizador

O Utilizador para além das variáveis de instância que herda da sua superclasse **User**, possui ainda duas listas adicionais. A primeira chama-se "**encomendasJaPreparadas**" e guarda as encomendas ,cujo as lojas ao qual o Utilizador solicitou encomendas, já sinalizaram que estão prontas a ser transportadas.

A outra lista denominada "**encomendasPorAceitarEntrega**" , guarda os códigos das encomendas cujo Utilizador já autorizou a sua entrega, esses códigos permanecerão na lista até que um **Voluntário aceite transportar a respetiva encomenda** , ou **o Utilizador aceite que uma Transportadora a transporte**.

Variáveis de instância:

```
-encomendasJaPreparadas : List<Encomenda>
-encomendasPorAceitarEntrega : List<String>
```

Métodos:

```
+Utilizador()
+Utilizador(codUtilizador : String, password : String, nome : String, coordenadas : GPS)
+Utilizador(u : Utilizador)
+Utilizador(codUtilizador : String, nome : String, coordenadas : GPS)
+getEncomendasParaSerEntregue() : List<Encomenda>
+clone() : Utilizador
+equals(obj : Object) : boolean
+toString() : String
+adicionaEncomenda(e : Encomenda) (*1)
+pop(idEnc : String) : Encomenda (*2)
+addEncomendaPorAceitar(codEnc : String)
+removeEncomendaPorAceitar(codEnc : String)
+apresentarListaEncomendasPorAceitar() : String
+apresentarListaEncomendasParaSerEntregue() : String
```

*1 -> adiciona a encomenda à lista **encomendasJaPreparadas** quando a Loja sinaliza que ta esta pronta;

*2 -> remove uma encomenda da lista **encomendasJaPreparadas** quando o Utilizador deseja que tal seja entregue, e retorna essa encomenda, que será adicionada à lista **encomendasPorAceitarEntrega;**

2.1.2 Voluntários

A seguir a definição dos Voluntários, bastante intuitiva

Variáveis de instância:

Todas as variáveis são herdadas da superclasse Transportador

Métodos:

+Voluntarios()

+Voluntarios(codVoluntario : String, nome : String, password : String, coordenadas : GPS, raio : double, tranportoMedicamentos : Boolean)

+Voluntarios(codVoluntario : String, nome : String, coordenadas : GPS, raio : double)

+Voluntarios(voluntario : Voluntarios)

+clone() : Voluntarios

+equals(o : Object) : boolean

+toString() : String

+paraCSV() : String

2.1.3 Transportadoras

Seguidamente a definição das Transportadoras, que possuem as mesmas variáveis de instância dos Voluntários (definidas elas na superclasse comum a ambos : **Transportador**) e mais algumas particulares das Transportadoras.

Variáveis de instância:

-nif : String

-preco_por_km : double

-capacidade : int

Métodos:

+Transportadoras()

+Transportadoras(codEmpresa : String, nomeEmpresa : String, password : String, coordenadas : GPS, nif : String, raio : double, preco_por_km : double)

+Transportadoras(codEmpresa : String, nomeEmpresa : String, password : String, coordenadas : GPS, nif : String, raio : double, preco_por_km : double, velocidade : int, capacidade : int, transportemedico : boolean)

+Transportadoras(t : Transportadoras)

+getNif() : String

+setNif(nif : String)

+getPreco_por_km() : double

+setPreco_por_km(preco_por_km : double)

+getCapacidade() : int

+setCapacidade(capacidade : int)

+equals(o : Object) : boolean

+toString() : String

+paraCSV() : String

+clone() : Transportadoras

+custoEntrega(loja : GPS, utilizador : GPS, peso : double) : double (*1)

+tempoChegadaLoja(loja : Lojas, tempo : User.Meteorologia) : double (*2)

+totalFaturadoTaxas() : String (*3)

*1 -> custo de entrega que será **apresentado ao Utilizador** quando este estiver a **escolher a qual Transportadora confiar a sua encomenda;**

*2 -> tempo de chegada da Transportadora à Loja;

*3 -> valor total faturado com as entregas efetuadas;

2.1.4 Lojas

As lojas possuem uma variável de instância chamada "**emPreparacao**" sendo ela uma lista que guarda as encomendas que lhe **foram pedidas por Utilizadores**.

Variáveis de instância:

```
-emPreparacao : List<Encomenda>
```

Métodos:

```
+Lojas(l : Lojas)
+Lojas()
+Lojas(id : String, nome : String, password : String, coordenadas : GPS)
+Lojas(id : String, nome : String, coordenadas : GPS)
+getCodLoja() : String
+setCodLoja(codLoja : String)
+getNomeLoja() : String
+setNomeLoja(nomeLoja : String)
+equals(o : Object) : boolean
+toString() : String
+clone() : Lojas
+addParaPreparacao(e : Encomenda)
+sizeParaPreparacao() : int
+verEncomendasEmPreparacao() : String
+removeDasPreparadas(codEnc : String) : Encomenda
+totalFaturado() : double
```

2.1.5 GPS

Rapidamente nos apercebemos que deveriamos criar uma Classe GPS comum a estas classes básicas, onde colocariamos as suas coordenadas (latitude e longitude), assim como seriam feitos todos os cálculos de distâncias e averiguação de posicionamento no raio de ação das Transportadoras / Voluntários.

Variáveis de instância:

```
-latitude : double
-longitude : double
```

Métodos:

```
+GPS()
+GPS(latitude : double, longitude : double)
+GPS(g : GPS)
+getLatitude() : double
+setLatitude(latitude : double)
+getLongitude() : double
+setLongitude(longitude : double)
+clone() : GPS
+equals(o : Object) : boolean
+toString() : String
-toRadianos(valor : double) : double
+dist(gps2 : GPS) : double
+dentroRaio(ponto : GPS, raio : double) : boolean
```

2.1.6 Transportador

Por falar em Transportadoras / Voluntários, achamos bem criar uma Superclasse chamada "Transportador" que ambos a vão herdar visto que tanto Transportadoras e Voluntários partilham variáveis de instância tal como o **número de kms percorridos**, **velocidade**, **disponibilidade**, **raio** e se **aceita transporte médico.** Sendo esta a responsável por muitos dos métodos comuns aos dois (como por exemplo os tempos de chegada) de forma a não repetir código.

Esta classe pode calcular o tempo de entrega da encomenda, assim como o tempo de chegada a uma loja, usando para isso as distâncias e velocidades, mas também tendo em conta fatores **meteorológicos** do dia presente, e o **estado pandémico** atual.

Esta classe possui duas listas , a primeira chamada "**paraLevar**" no caso de um **Voluntário** guarda no **máximo uma encomenda** - (que é a capacidade de um Voluntário) - **que este escolheu entregar**. No caso de uma **Transportadora** com **capacidade N**, guarda no **máximo N encomendas** , que **Utilizadores aprovaram que a Transportadora as entregasse**.

A segunda lista chamada **"jaFoiBuscar"** , no caso do Voluntário guarda a encomenda que este foi buscar à Loja , e no caso da Transportadora guarda as encomendas que esta foi buscar.

Variáveis de instância:

```
-raio : double
-transportemedico : Boolean
-kms : double
-paraLevar : List<String>
-jaFoiBuscar : List<String>
-disponibilidade : Boolean
-velocidade : int
```

Métodos:

```
+Transportador()
+Transportador(id : String, nome : String, password : String, coordenadas : GPS,
raio : double, transportemedico : Boolean)
+Transportador(id : String, nome : String, password : String, coordenadas : GPS,
raio : double, transportemedico : Boolean , velocidade : int)
+Transportador(t : Transportador)
+setDisponibilidade(b : Boolean)
+getDisponibilidade() : Boolean
```

+getRaio() : double

+setRaio(raio : double)

+isTransportemedico() : Boolean

+aceitoTransporteMedicamentos() : Boolean

+aceitaMedicamentos(transportemedico : Boolean)

+getKms() : double

+setKms(kms : double)

+equals(o : Object) : boolean

+getParaLevar() : List<String>

+addParaLevar(codEnc : String)

+removeParaLevar(codEnc : String)

+getVelocidade() : int

+setVelocidade(velocidade : int)

+toString() : String

+clone() : Transportador

+addJaFoiBuscar(codEnc : String)

+removeJaFoiBuscar(codEnc : String)

+getJaFoiBuscar() : List<String>

+sizeJaFoiBuscar() : int

+limpaListas(correspondencia : Map<String,List<Transportadoras>>) (*1)

+tempoChegadaLoja(loja : Lojas, tempo : User.Meteorologia) : double

+tempoChegadaUser(loja : Lojas, uti : Utilizador, tempo : User.Meteorologia) : double

*1 -> quando uma **Transportadora sinaliza que já não está disponível** a fazer entregas é necessário **"devolver" as encomendas** às Lojas no caso de ter encomendas guardadas nas listas acima faladas, de forma a permitir que o Utilizador possa escolher outra Transportadora, e é necessário **"remover do sistema" temporariamente esta Transportadora até que esta sinalize que está disponível**;

2.1.7 Lojas Com Fila

A nossa classe Lojas tem uma especilização as "LojasComFila", este tipo de Lojas implementa uma fila de espera.

As novidades nestas lojas com Fila é elas conseguirem calcular os diversos tempos de espera utilizando diversos fatores como o número de pessoas na fila, a metereologia e o estado da pandemia.

Esta classe tem duas lista, a primeira "**tempos**" guarda todos os tempos de entrega de forma a ser mais fácil a calcular a média, e a "**fila**" guarda todas as encomendas que ainda não foram levantadas da Loja.

Variáveis de instância:

```
-tempos : List <Double>

-fila : List <Encomenda>
```

Métodos:

```
+LojaComFila()

+LojaComFila(codLoja : String, nomeLoja : String, password : String, coordenadas
: GPS)

+LojaComFila(codLoja : String, nomeLoja : String, password : String, coordenadas
: GPS, tempos : List <Double>, fila : List <Encomenda>)

+LojaComFila(loja : LojaComFila)

+getTempos() : List <Double>

+setTempos(tempos : List <Double>)

+getFila() : List <Encomenda>

+setFila(f : List <Encomenda>)

+adicionaAFila(e : Encomenda)

+removeDaFila(e : Encomenda)

+addTempoEspera(e : EstadoCovid) (*1)

+tempoMedioEspera() : double

+tempoRecolherEntrega(t : Transportador, tempo : Meteorologia, e : EstadoCovid)
: double  (*2)
```

*1 -> adiciona um tempo à lista de tempos, a função recebe como argumento o estado covid do dia, e a **"fórmula"** de cálculo do tempo é (**tamFila * tempoMedio) + tempo de atraso** devido ao estado do covid;

*2 -> calcula a soma do tempo a chegar à Loja e de recolher a encomenda

2.1.7 User

Todas estas classes faladas anteriormente com exceção do GPS são subclasses da classe abstrata User, classe que foi criada com o intuito de guardar as credenciais de acesso, as coordenadas GPS e os registos de Encomendas que irá ser o foco do próximo tema.

Variáveis de instância:

```
-id : String
-nome : String
-password : String
-coordenadas : GPS
-registos : RegistoEncomendaLista
```

Métodos:

```
+User()
+User(id : String, nome : String, password : String, coordenadas : GPS)
+User(u : User)
+getId() : String
+setId(id : String)
+getNome() : String
+setNome(nome : String)
+getPassword() : String
+setPassword(password : String)
+getCoordenadas() : GPS
+setCoordenadas(coordenadas : GPS)
+equals(o : Object) : boolean
+toString() : String
+clone() : User
+addToRegisto(entregador : String, encomenda : Encomenda, classificacao : int,
dataDaEntrega : LocalDate, kmsPercorridos : double, taxaDeEntrega : double,
tempoDeEntrega : double) (*1)
+verRegistos(codEntregador : String, inicio : LocalDate, fim : LocalDate) :
List<RegistoEncomenda>
+verRegistosGeral() : List<RegistoEncomenda>
+naoClassificado() : List<RegistoEncomenda> (*2)
+classificarEntrega(codEnc : String, classificacao : int)
+verClassificacoes() : String
+verClassificacoesMedia() : String
+verTopN(contas : Map<String,User>, n : int, c : Comparator <User>, classe :
String) (*3)
```

+verTopNKm (contas: Map<String,User>, n : int, c : Comparator <Transportador>, classe : String) (*4)

*1 -> cria e adiciona um novo registo a lista de registos

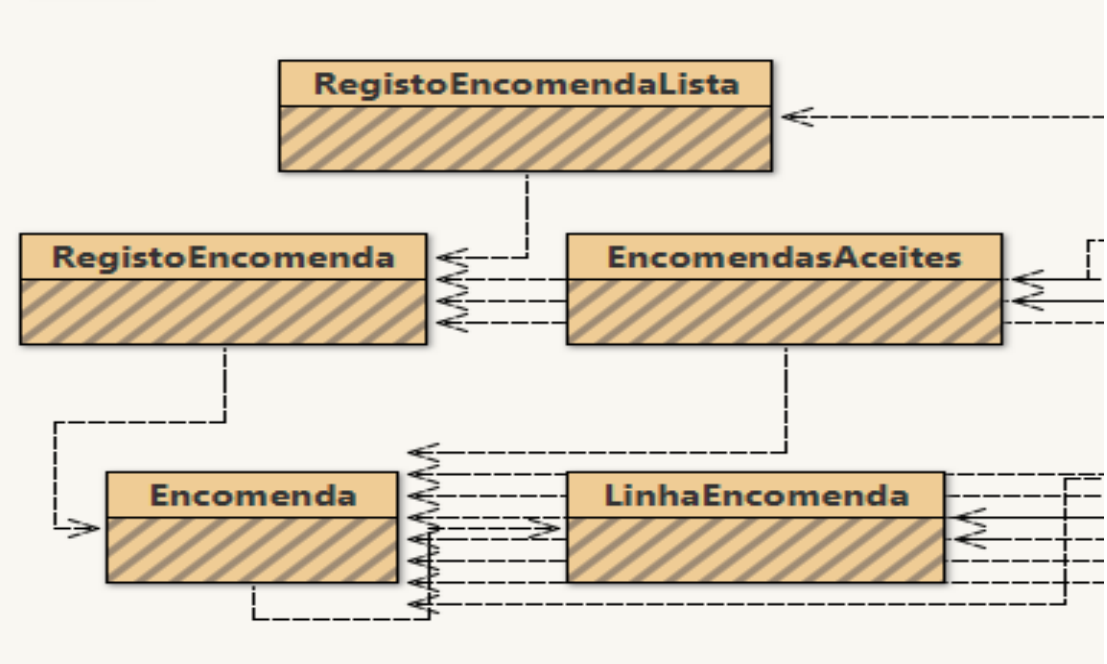
*2 -> devolve uma lista com os registos ainda não classificados

*3 e *4 -> print das classificações

2.2 Sistema de Registos de Encomendas

Como deu para reparar o User têm uma Lista muito importante nas variáveis de instância, chamada “registos” , onde guardamos todas as encomendas anteriores, feitas por um utilizador (caso seja um Utilizador), transportadas por um Transportador (Voluntário ou Transportadora) ou preparadas por uma loja (caso seja obviamente do tipo Loja).

Esta lista é do tipo “registoEncomendaLista” uma classe criada por nós que iremos abordar em mais detalhe nesta secção, seguidamente o Diagrama das classes que iremos abordar nesta secção:



Á primeira vista tantas classes com Encomenda no nome parece confuso, mas vamos passo a passo explicar como tudo isto funciona, começando por o mais básico, a Linha de Encomenda.

2.2.1 Linha de Encomenda

A Linha de Encomenda é onde se encontra os produtos que o Utilizador vai encomendar, está lá todas as características desses mesmos produtos a encomendar, desde os códigos, as suas descrições , preços e quantidades.

Variáveis de instância:

```
-codProduto : String
-descricao : String
-quantidade : double
-valorUnitario : double
```

Métodos:

```
+LinhaEncomenda()
+LinhaEncomenda(codProduto : String, descricao : String, quantidade : double,
valorUnitario : double)
+LinhaEncomenda(l : LinhaEncomenda)
+getCodProduto() : String
+setCodProduto(codProduto : String)
+getDescricao() : String
+setDescricao(descricao : String)
+getQuantidade() : double
+setQuantidade(quantidade : double)
+getValorUnitario() : double
+setValorUnitario(valorUnitario : double)
+clone() : LinhaEncomenda
+equals(obj : Object) : boolean
+toString() : String
+valorTotalLinhaEncomenda() : double
```

2.2.2 Encomenda

Uma encomenda será constituída por uma **lista de Linhas de Encomenda** sendo estes os artigos da Encomenda, e também por informações dela mesma, como **código de Utilizador** que encomendou, **código da Loja** que preparou, **peso, data** e se a encomenda é **médica**.

Variáveis de instância:

```
-codEncomenda : String
-codUtilizador : String
-codLoja : String
-peso : double
-data : LocalDate
-linhas : List <LinhaEncomenda>
-eMedica : Boolean
```

Métodos:

```
+Encomenda()
+Encomenda(codEncomenda : String, codUtilizador : String, codLoja : String,
peso : double, data : LocalDate, linhas : List <LinhaEncomenda>)
+Encomenda(e : Encomenda)
+getCodEncomenda() : String
+setCodEncomenda(codEncomenda : String)
+getCodUtilizador() : String
+setCodUtilizador(codUtilizador : String)
+getCodLoja() : String
+setCodLoja(codLoja : String)
+getPeso() : double
+setPeso(peso : double)
+getData() : LocalDate
+setData(data : LocalDate)
+getEMedica() : Boolean
+setEMedica(b : Boolean)
+getLinhas() : List <LinhaEncomenda>
+setLinhas(linhas : List <LinhaEncomenda>)
+clone() : Encomenda
+equals(obj : Object) : boolean
+contains(le : LinhaEncomenda) : boolean
+toString() : String
+valorTotalEncomenda() : double
```

```
+adicionaLinha(linha : LinhaEncomenda)
+removeLinha(codLinha : String)
```

2.2.3 Encomendas Aceites

As Encomendas Aceites é uma classe constituída por uma Lista de Encomendas, sendo estas as Encomendas já aceites (Encomendadas) por um utilizador e precisam de ser entregues aliás apenas o Model (TrazAqui.java) usa esta class como uma variável de instância com o nome de **precisaSerEntregue**, esta lista é nos útil para as Transportadoras e Voluntários saberem quais as encomendas que podem aceitar para entregar, sendo assim após um utilizador aceitar a entrega ou um Voluntário, a encomenda irá sair da lista.

Variáveis de instância:

```
-encomendas : List <Encomenda>
```

Métodos:

```
+EncomendasAceites()
+EncomendasAceites(e : EncomendasAceites)
+EncomendasAceites(encomendas : List<Encomenda>)
+getEncomendas() : List<Encomenda>
+setEncomendas(encomendas : List<Encomenda>)
+getSize() : int
+clone() : EncomendasAceites
+toString() : String
+equals(o : Object) : boolean
+addEncomendaAceite(encomendaAceite : Encomenda)
+buscaEncomenda(codEnc : String) : Encomenda (*1)
+removeEncomenda(e : Encomenda)

*1 -> encontra uma encomenda e devolve um clone da mesma
```

2.2.4 Registo Encomenda

O registo de Encomenda é uma classe onde fica registado tudo o que é útil de uma encomenda, sendo isso o objeto Encomenda em si, o entregador, a classificação obtida pelo entregador, a data de entrega, os kms percorridos, a taxa de entrega se aplicável e o tempo de entrega.

Variáveis de instância:

```
-entregador : String
-encomenda : Encomenda
-classificacao : int
-dataDaEntrega : LocalDate
-kmsPercorridos : double
-taxaDeEntrega : double
-tempoDeEntrega : double
```

Métodos:

```
+RegistoEncomenda()
+RegistoEncomenda(entregador : String, encomenda : Encomenda, classificacao : int, dataDaEntrega : LocalDate, kmsPercorridos : double, taxaDeEntrega : double, tempoDeEntrega : double)
+RegistoEncomenda(e : RegistoEncomenda)
+setTempoDeEntrega(x : double)
+getTempoDeEntrega() : double
+setKmsPercorridos(kmsPercorridos : double)
+getKmsPercorridos() : double
+setTaxaDeEntrega(taxaDeEntrega : double)
+getTaxaDeEntrega() : double
+getEntregador() : String
+setEntregador(entregador : String)
+getEncomenda() : Encomenda
+setEncomenda(encomenda : Encomenda)
+getClassificacao() : int
+setClassificacao(classificacao : int)
+getDataDaEntrega() : LocalDate
+setDataDaEntrega(dataDaEntrega : LocalDate)
+equals(o : Object) : boolean
+toString() : String
+clone() : RegistoEncomenda
```

2.2.5 Registo Encomenda Lista

A classe registoEncomendaLista como o nome indica é constituída por uma lista de Registos de Encomendas, todo o User tem uma onde guarda as encomendas efetuadas, transportadas ou preparadas, dependendo o tipo de User que ele é.

É a partir desta lista de registos que conseguimos calcular os totais faturados assim como calcular os top x melhores Users, com base em diversos critérios.

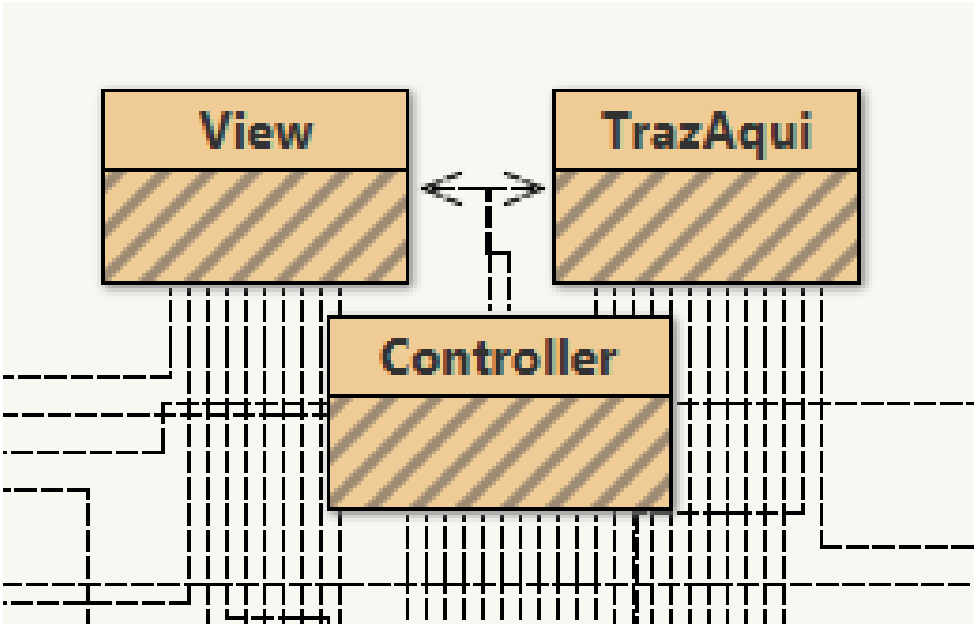
Variáveis de instância:

```
-registo : List<RegistoEncomenda>
```

Métodos:

```
+RegistoEncomendaLista()

+RegistoEncomendaLista(registo : List<RegistoEncomenda>)

+RegistoEncomendaLista(r : RegistoEncomendaLista)

+getRegisto() : List<RegistoEncomenda>

+setRegisto(registo : List<RegistoEncomenda>)

+equals(o : Object) : boolean

+toString() : String

+clone() : RegistoEncomendaLista

+add(registoEnc : RegistoEncomenda)

+verRegistos(codEntregador : String, inicio : LocalDate, fim : LocalDate) :
List<RegistoEncomenda>

+verRegistosGeral() : List<RegistoEncomenda>

+naoClassificado() : List<RegistoEncomenda>

+classificaEntrega(codEnc : String, classificacao : int)

+verClassificacoes() : String

+verClassificacaoMedia() : String
```

2.3 Model, View, Controller

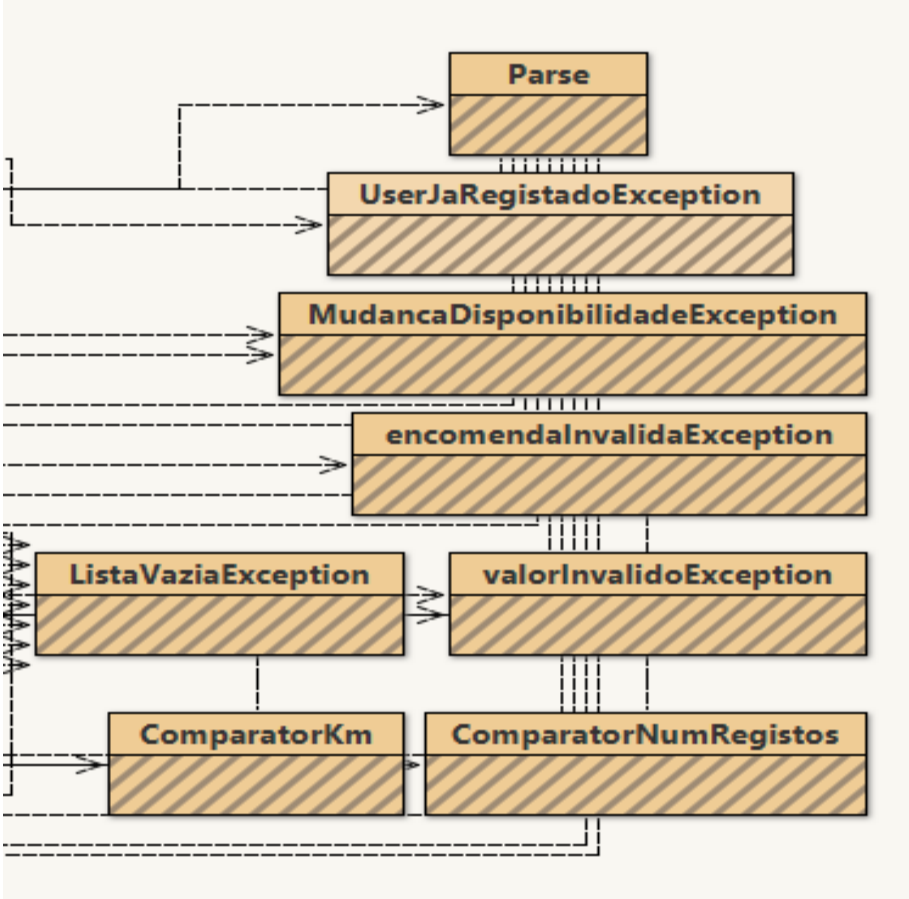


Neste parte iremos explicar de que forma aplicamos a arquitetura MVC no nosso programa.

- **Model**: O nosso Model é a class TrazAqui, são aqui que são guardadas as contas e as diversas listas de encomendas assim como muita da lógica do programa.
- **Controller**: A classe controller é aqui que fazemos a ligação entre o Model e a View.
- **View**: É a classe view onde temos todos os menus, inputs e outputs.

Desta forma podemos crescer e modificar o nosso programa de forma correta e segura sem comprometer as suas funcionalidades.

2.4 Classes Auxiliares



Neste subtópico iremos abordar as classes auxiliares, neste caso os comparadores, as exceptions e o Parse.

Começando pelos comparadores:

- **ComparatorNumRegistos**: Vai comparar o tamanho da lista das encomendas nos registos em cada Usuário, é um comparador muito geral pois pode ser usado em todas as subclasses do Usuário para vários fins, entre eles: saber quantas encomendas preparou (Loja), quantas encomendas transportou (Voluntários / Transportadoras) ou quantas encomendas fez (Utilizadores).
- **ComparadorKm** vai comparar o número de kilómetros percorridos pelos Voluntários ou Transportadoras.

Ambos os comparadores são úteis para o top 'x' abordado mais à frente.

Passando ás exceptions, temos 4 classes mas antes disso temos uma exception especial que pode ocurrer nos Registos / Logins, a NULLPOINTEREXCEPTION explicada abaixo:

- **NULLPOINTEREXCEPTION**: Quando inserimos uma palavra-passe tanto na parte do registo, tanto na parte do login, usamos uma função chamada console.readPassword(), mas algumas IDE (nomeadamente o Bluej), não suportam funções do tipo console, logo irá ter uma NULLPOINTEREXCEPTION, nós tratamos dessa exceção e adicionamos uma mensagem de erro quando isso acontecer.

Agora que vimos este caso especial, vamos ver as nossas 5 classes de exceptions:

- **MudancaDisponibilidadeException**: Acontece quando uma Transportadora ou Voluntário tenta mudar a sua disponibilidade para false enquanto estiver a transporta uma encomenda.
- **UserJaRegistadoException**: Acontece quando um Usuário tenta se registar com um nome já antes escolhido por outro.
- **ValorInvalidoException**: Acontece quando o Usuário insere valores negativos onde é impossivel isso acontecer (velocidade, peso, preço, etc..)
- **EncomendaInvalidaException**: Acontece quando algum Usuário tenta escolher uma encomenda para transportar, preparar ou aceitar inválida (que não existe ou que não é relevante a ele mesmo)
- **ListaVaziaException**: Acontece quando vamos tentar escolher uma encomenda, quer seja para aceitar, preparar ou transportar de uma lista vazia.

Além destas exceptions, também temos outras ao longo do programa como as FILENOTFOUNDEXCEPTION, IOEXCEPTION, CLASSNOTFOUNDEXCEPTION, entre outras..

2.4.1 Parse

Variáveis de instância:

-contas : Map < String , User >

-encomendasAceites : EncomendasAceites

-encomendas : Map <String , Encomenda>

Métodos:

+Parse()

+parse(ficheiro : String)

-parseEncomenda(input : String) : Encomenda

-parseLoja(input : String) : Lojas

-parseTransportadora(input : String) : Transportadoras

-parseUtilizador(input : String) : Utilizador

-parseVoluntarios(input : String) : Voluntarios

-lerFicheiro(nomeFich : String) : List<String>

+getContas() : Map < String , User>

+getEncomendas() : List <Encomenda>

+getEncomendasAceites() : List <Encomenda>

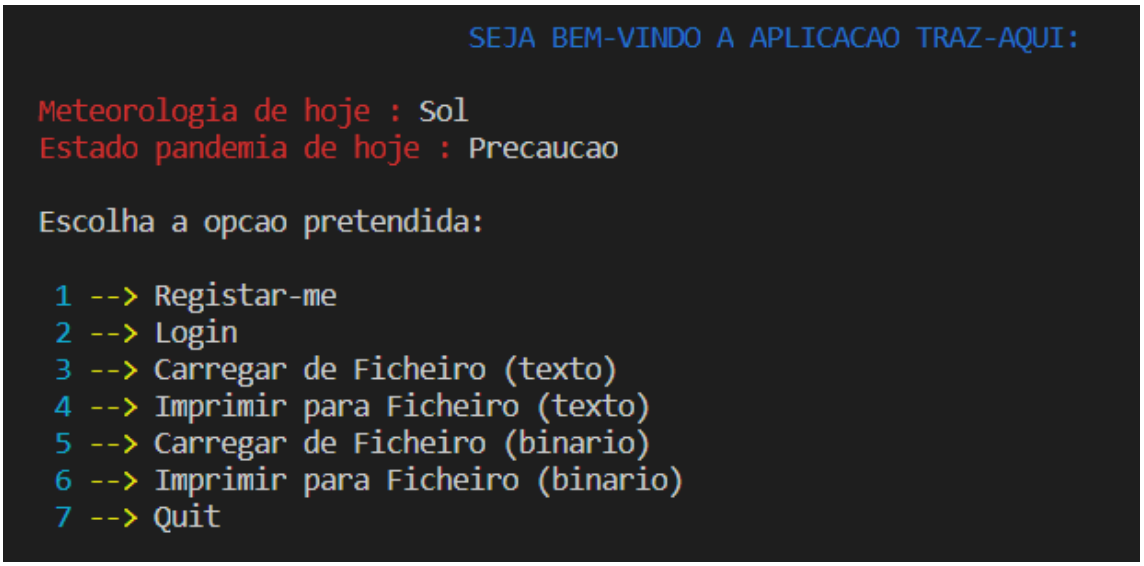
3 Descrição da Aplicação

Neste tópico iremos fazer uma breve descrição da aplicação, os seus menus e as suas funcionalidades.

3.1 Menu Inicial

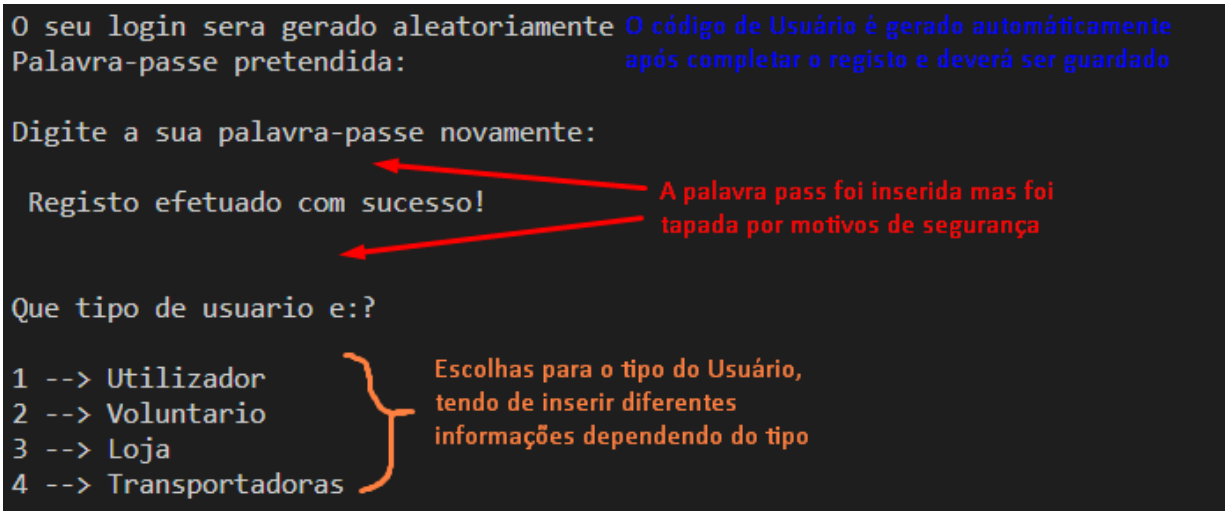
Ao abrir a aplicação, se existir o ficheiro "**save**" (em binário) será automaticamente carregado para a memória da aplicação, este ficheiro é também automaticamente gravado ao fechar-mos a aplicação com segurança na opção de **quit**.

Quando um utilizador abre a aplicação ele é deparado com o menu principal constituído por diversas opções que funcionam à base de números, sendo este:



Notamos logo no início que a aplicação dispõe da **metereologia** de hoje assim como o **estado da pandemia**, visto que não temos forma de sabermos as verdadeiras situações do dia, estes serão gerados aleatoriamente ao abrir a aplicação.

Opção 1: opção de registo, primeiro insere-se o nome de conta pretendido, depois será pedido para inserir a password* pretendida 2 vezes e deverá coincidir, após isso o utilizador escolhe o tipo de Usuário que é.



Após o seu registo, o Usuário será redirecionado para o menu principal onde já poderá fazer login com as suas credenciais acessando depois o seu respetivo menu com as opções para tudo o que necessitar.

Opção 2: Os Usuários da aplicação podem fazer Login na opção 2 inserindo o código de Usuário que é gerado ao registar ou ao inserir o nome que foi escolhido no registo, seguindo da password respetiva**

*É de notar que devido a usarmos o método console.readpassword() para ler-mos as passwords, algumas IDE's podem não funcionar com tal método e criar uma NULLPOINTEREXCEPTION nomeadamente o BlueJ(Na mesma temos um try catch para esses casos)

**Se for lido de um ficheiro CSV todos os Usuários ficam com a password pré definida "123"

Na secção 3.2 vamos apresentar os diferentes menus lado a lado e depois iremos explicar cada um em mais detalhe, agora iremos continuar por analisar este menu inicial.

Já analisamos as opções de registo e Login, as próximas a analizar serão as opções 3 e 4, sendo elas respetivamente carregar de um ficheiro em texto no formato CSV e ler de um ficheiro em texto, no formato CSV.

A **opção 3** irá carregar de um ficheiro CSV à escolha do Usuário, todos os Usuários carregados irão ficar com a palavra-passe pré-definida “123”, visto que os ficheiro CSV carregados têm de ser formatados aos log.txt de exemplo.

A **opção 4** imprime o estado atual em formato CSV, esta opção irá pedir o nome do ficheiro no qual será guardado.

A **opção 5** carrega todo o model de um ficheiro binário para o nosso programa.

A **opção 6** imprime todo o model para um ficheiro á escolha em binário.

A **opção 7** é para sair do programa, antes de sair iremos:

1 - Guardar todo o estado atual do programa em binário

2 - Fechar todos os scanners

E só depois saímos com segurança da nossa aplicação, é de notar que a próxima vez que executar o programa, toda essa informação em binário irá ser carregada para o nosso programa.

3.2 Menus dos Usuários

Começemos por colocar os quatro diferentes menus de Usuários lado a lado, depois iremos apresentar cada um individualmente.

Utilizador:

```
Selecione a opcao desejada:

1 - Criar encomenda
2 - Pedir entrega de uma encomenda
3 - Aceitar Transporte
4 - Classificar Voluntario / Empresa de Transporte
5 - consultas avancadas
6 - Voltar ao menu inicial
```

Voluntário:

```
Selecione a opcao desejada:

1 - Sinalizar disponibilidade de entrega
2 - Ir buscar certa encomenda a uma loja dentro do seu raio
3 - Transportar encomenda
4 - consultas avancadas
5 - Voltar ao menu inicial
```

Lojas:

```
Selecione a opcao desejada:

1 - Sinalizar que certa encomenda esta disponivel para entrega
2 - consultas avancadas
3 - Voltar ao menu inicial
```

Transportadoras:

```
Selecione a opcao desejada:

1 - Sinalizar disponibilidade de entrega
2 - Aceitar entrega de certa encomenda no seu raio
3 - Ir buscar certa encomenda a uma loja
4 - Transportar encomenda
5 - consultas avancadas
6 - Voltar ao menu inicial
```

3.2.1 Menu dos Utilizadores

1- Pedir Encomenda: Nesta opção o utilizador irá pedir uma encomenda a uma loja, o pedido é basico, o utilizador apenas escolhe umas das lojas disponíveis da aplicação, depois disso escolhe os produtos e preenche as suas características, quando tiver finalizado a Encomenda a Loja começa a preparar-la.

2- Solicitar Entrega: Após uma loja preparar uma encomenda o Utilizador pode solicitar a sua entrega para permitir aos Transportadores a aceitar.

3- Aceitar Transporte: Caso nenhum voluntário ainda tenha aceite transportar a encomenda, o utilizador poderá selecionar esta opção para escolher uma das transportadoras da lista de transportadoras que aceitaram entregar uma encomenda, será disponibilizada a lista dessas tais transportadoras que aceitaram assim como várias informações sobre as mesmas (tempos, custos, classificações..) para o Utilizador poder fazer a melhor escolha.

4- Classificar Voluntario / Empresa de Transporte: O Utilizador poderá classificar os Voluntários e/ou Empresas de Transporte após um transporte bem sucedido.

5- Consultas avançadas: Uma lista de opções para ver dados avançados pertinentes ao Utilizador, como ver o seu perfil, registo de encomendas (total ou em certas datas), encomendas pedidas ainda não entregues ou até mesmo ver o top ‘x’.

6- **Voltar ao menu inicial:** Voltar atrás.

3.2.3 Menu dos Voluntários

1-Sinalizar disponibilidade de entrega: Mudar a sua disponibilidade de poder entregar encomendas numa dada altura (Nota: não poderá mudar a disponibilidade para Falso se estiver a transportar uma encomenda nesse momento).

2-Ir buscar uma certa encomenda a uma Loja: Das encomendas que aceitou, poderá ir buscar (uma de cada vez visto a capacidade do Voluntário ser igual 1) uma delas para entregar.

3-Transportar encomenda: Transporta a encomenda que foi buscar até ao utilizador, no final os registos dos três (Utilizador, Voluntário e Loja) serão atualizados.

4- Consultas avançadas: Uma lista de opções para ver dados avançados pertinentes ao Voluntário, como ver o seu perfil, as suas classificações, registos de encomendas entregues (total ou em certas datas) e ver o top 'x'.

5- **Voltar ao menu inicial:** Voltar atrás.

3.2.4 Menu das Lojas

1-Sinalizar que certa encomenda está disponível para ser entregue: Após a loja preparar uma encomenda anteriormente pedida por um Utilizador, a Loja deve selecionar esta opcao e escolher qual a encomenda está prepara da lista das encomendas por preparar para depois o Utilizador poder solicitar a sua entrega.

2- Consultas avançadas: Uma lista de opções para ver dados avançados pertinentes à Loja, como ver o seu perfil, total faturado na aplicação, total de pessoas na fila, registos de encomendas preparadas (total ou entre certas datas) e o top ‘x’.

3- **Voltar ao menu inicial:** Voltar atrás.

3.2.5 Menu das Transportadoras

1-Sinalizar disponibilidade de entrega: Mudar a sua disponibilidade de poder entregar encomendas numa dada altura (Nota: não poderá mudar a disponibilidade para Falso se estiver a transportar uma encomenda nesse momento).

2- Aceitar a entrega de certa encomenda no seu raio: Como o nome indica, esta opção é para aceitar a entrega de uma determinada encomenda dentro do seu raio.

3- Ir buscar certa encomenda a uma loja: Para ir buscar uma encomenda á qual foi escolhida para entregar por um Utilizador.

4 – Transportar encomenda: Transporta a encomenda que foi buscar até ao utilizador, no final os registos dos três (Transportadora, Voluntário e Loja) serão atualizados.

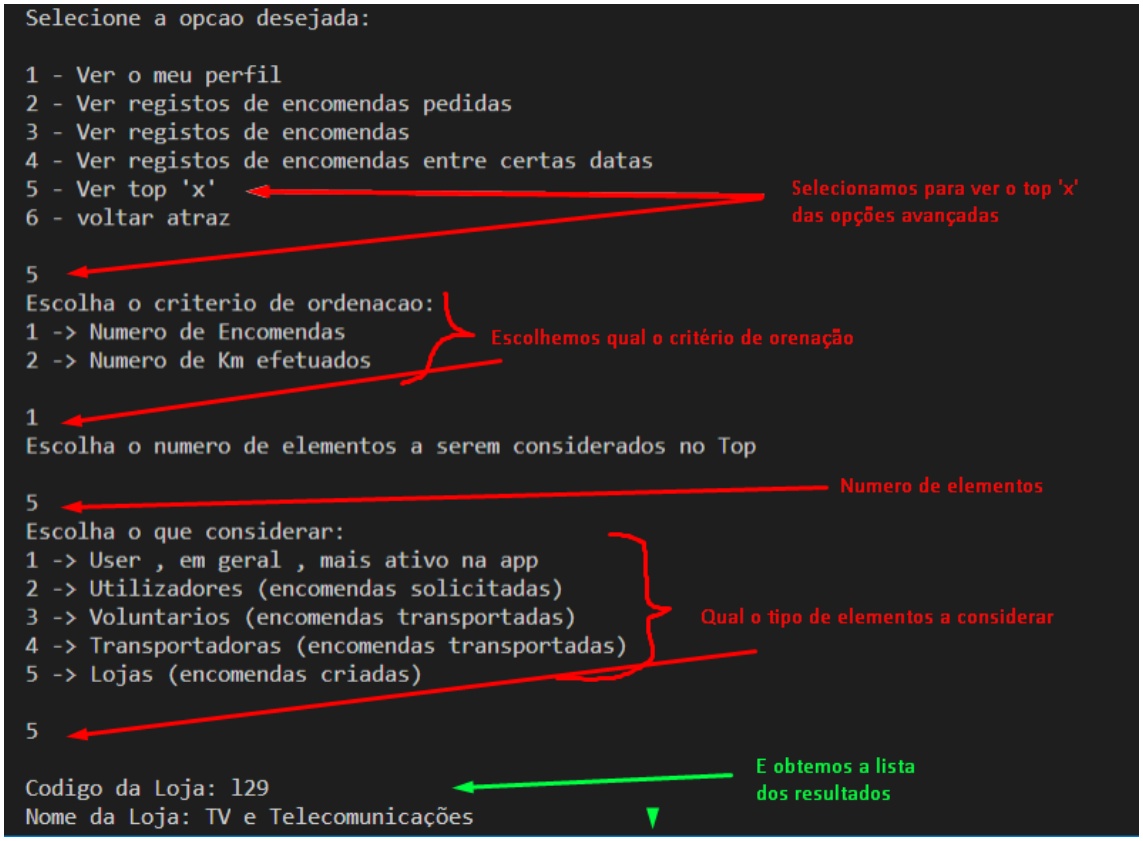
5- Consultas avançadas: Uma lista de opções para ver dados avançados pertinentes à Transportadora, como ver o seu perfil, as suas classificações, total faturado na aplicação num período de tempo, registos de encomendas entregues (total e num certo período de tempo) e ver o top ‘x’.

6 - Voltar ao menu inicial: Voltar atrás.

3.2.6 Top ‘x’

Como vimos todos os menus dos diferentes usuários possuem uma opção chamada Ver top ‘x’, esta é a opção que nos permite ver um ‘x’ número de melhores Usuários (Users, Lojas, Voluntários, Transportadoras ou Utilizadores) ordenados por diferentes critérios à escolha (User mais ativo, mais encomendas solicitadas, mais encomentas preparas, mais kms percorridos, etc..).

Vamos exemplificar o seu funcionamento:



Este top ‘x’ foi implementado a pensar na sua escalabilidade, usamos assim uma Function que pega no comparador escolhido, podemos assim escalar criando novos comparadores e passando para essa Function.

4 Escalabilidade da Aplicação

Consideramos que a aplicação construía seja facilmente escalável, adicionando novos tipos de Usuários nela, assim como novas funcionalidades, opções avançadas e até mesmo critérios novos de comparação do top 'x'

Devido ao modelo MVC a aplicação pode também suportar outros tipos de views sem comprometer o resto da aplicação, sendo desta forma bastante flexível no que toca à inclusão de novos conceitos e funcionalidades.

5 Aleatoriedade da Aplicação (Tempos de Espera) e tipos de encomendas

A nossa aplicação foi desenvolvida com uma certa aleatoriedade na meteorologia e estado da pandemia do presente dia, podendo esta ter diferentes estados:

- Sol
- Chuva
- Nevoeiro
- Neve

Cada tipo de meteorologia gerado aleatoriamente no inicio do programa vai abrandar uns minutos o tempo de chegada á loja e ao utilizador respetivamente 0,1,2,4 minutos.

O estado da pandemia também pode ter 3 estados, sendo eles:

- Normalidade
- Precaução
- Emergência

Os quais atrasam o tempo médio de espera entre 2 clientes e o tempo de um transportador (transportadora ou voluntário) levantar a encomenda em 1,3,5 minutos respetivamente.

Estas são as únicas aleatoriedades consideradas no nosso programa.

O nosso programa também tem tipos de encomendas diferentes, neste caso apenas 2 tipos, encomendas médicas e não médicas. Um transportador para poder levar uma encomenda médica tem de possuir um certificado de que a pode transportar, sendo isto perguntado no registo do mesmo.

6 Conclusão

Achamos este projeto de grande importância visto que abordou praticamente todos os conceitos abordados ao longo deste semestre onde tivemos de aplicar-los numa aplicação de maior escala.

À medida que fomos criando mais código e adicionando classes novas, fomos notando que o código estava a ficar muito confuso e complicado de descobrir onde estavam os erros, daí a importância de uma boa organização e estruturação do código. Devíamos ter planeado melhor antes de passar à programação, iremos assim ter isso em conta nos próximos projetos.

Os nossos objetivos foram alcançados visto que conseguimos cumprir todos os requesitos básicos e avançados da aplicação desta forma estamos contentes com o resultado final, alguns aspectos poderiam ser melhorados ou pensados de forma diferente nomeadamente a organização do código, a estruturação dos mecanismos de encomendas (pensamos em uma variável para os diversos estados da encomenda), um melhoramento da forma como o utilizador pede as encomendas, a adição de mais algumas opções avançadas relevantes assim com a adição de mais algumas exceptions que nos falhou.