

T7 – Empresa de Táxis

Algoritmos e Estrutura de Dados

Nuno Manuel Ferreira Corte-Real – 201405158 – up201405158@fe.up.pt

20 de Novembro de 2016

Turma – 2MIEIC03 – Grupo - G

Faculdade de Engenharia da Universidade do Porto

Rua Roberto Frias, sn, 42^o-465 Porto, Portugal

Índice

Empresa de Táxis (especificação do tema do projecto)	3
Solução Implementada	4
- Empresa.h / Empresa.cpp	4
- Serviço.h / Serviço.cpp	7
- Cliente.h / Cliente.cpp	10
-- Clientes Registados	12
-- Clientes Empresariais	15
-- clientes Empresariais	16
- Interface.h / Interface.cpp	17
- Utilidades.h	20
- main.cpp	21
Diagrama de Classes (UML)	22
Casos de Utilização	23
Dificuldades Encontradas	24
Distribuição do Trabalho pelos Elementos de Grupo	24

Empresa de Táxis

Este trabalho tem como tema uma empresa de táxis e a sua manutenção. Foca-se especialmente na relação entre clientes, serviços prestados pela empresa e variáveis incluídas.

No programa, existem vários tipos de clientes. Existem os clientes não registados na empresa, que somente solicitam serviços rápidos (ficando os serviços solicitados guardados na empresa, mas não dados relativamente a clientes não registados);

Clientes Registados, que por sua vez podem assumir três formatos: Cliente Registado normal, Cliente Registado Empresarial, Cliente Registado Particular. Este tipo de clientes guarda em si um histórico de Serviços solicitados à empresa, que são usados mais tarde para o cálculo de possíveis promoções na cobrança dos serviços.

Não podem existir dois clientes idênticos, ou seja, com o mesmo id (que neste caso, é representado pelo seu número NIF).

Os pagamentos dos serviços podem também ser efectuados de vários tipos: aos clientes não registados estão à sua disposição os pagamentos por numerário e por multibanco. Aos clientes registados estão disponíveis todas e quaisquer formas de pagamento, variando estas entre numerário, multibanco, pagamento ao fim do mês e pagamento por cartão de crédito. De salientar que os pagamentos por cartão de crédito envolvem um imposto de 5% do valor total do serviço e os pagamentos ao fim do mês envolvem um imposto de 2% ao valor total de todos os serviços pagos no fim dest e período.

A empresa também poderá oferecer aos seus clientes registados certos descontos confirme estes tenham acumulado vários serviços solicitados à empresa no seu histórico de serviços.

Dois serviços são idênticos se possuírem a mesma origem, o mesmo destino e forem efectuados à mesma hora e data. Tal não pode suceder.

O valor de cada serviço é influenciado pela hora a que é feito (devido ao período de maior trânsito) e pela distância desse serviço.

Solução Implementada

Empresa.h/Empresa.cpp

A Empresa de Táxis em si mesma, componente principal do programa foi implementada usando uma classe Empresa que permite a gestão de seus clientes e serviços.

Cada empresa possui um nome e uma morada (ambos definidos por objectos do tipo `std::string`), não podendo duas empresas possuir um nome ou morada idênticos.

A classe Empresa possui em si os seguintes métodos públicos:

`Empresa() {}` -> constructor de um objecto do tipo Empresa com os seus membros inicializados a valor nulos

`Empresa(std::string nome)` -> constructor de um objecto Empresa com um nome específico

`Empresa(std::string nome, std::string morada)` -> constructor de um objecto Empresa com um nome e uma morada específicos

`std::string getNome() const` -> retorna o nome da Empresa

`std::string getMorada() const` -> retorna a morada da Empresa

`std::vector<Registado> * getRegistados() const` -> retorna um vector que possui todos os elementos Registados guardados pela Empresa

`Registado* getRegistadoByNIF(unsigned int NIF) const` -> retorna um cliente Registado com um NIF específico, passado como argumento

`Servico* getServicoByID(unsigned int servID) const` -> retorna um Serviço com um Id específico, passado como argumento

```

bool adicionaCliente(Registado *cli) -> tenta adicionar um cliente Registado à
    Empresa, retornando true se conseguiu ou
    false se não conseguiu. Caso o Cliente seja
    repetido, lança uma exceção do tipo
    ClienteRepetido.

bool removeCliente(Registado *cli) -> tenta remover um cliente Registado da
    Empresa, retornando true se conseguiu ou
    false se não conseguiu. Caso o cliente
    solicitado no argumento da função não exista,
    lança uma exceção do tipo
    ClienteInexistente.

bool removeCliente(unsigned int NIF) -> tenta remover um cliente Registado com um NIF
    específico. Caso consiga, retorna true; caso
    não consiga, retorna false; Caso o cliente de NIF
    igual ao mandado como argumento da função não
    exista, lança uma exceção do tipo
    ClienteInexistente.

std::vector<Servico *> getHistorico() const -> retorna o vector de serviços guardados
    da empresa (histórico de serviços)

std::string getInformacao() const -> retorna toda a informação relativa à Empresa, já
    formatada, numa std::string. A informação
    consiste em toda a informação relativa a cada
    cliente e os seus respectivos históricos de
    serviços acumulados. Também indica o estado de cada
    serviço.

std::string getInformacaoServicos() const -> retorna toda a informação relativa a cada
    serviço acumulado pela Empresa.

std::string getNIFsClientes() const -> retorna uma std::string onde são escritos todos
    os NIF's de todos os clients registados na
    Empresa, formatados de uma forma amigável para o
    utilizador.

std::string getIDsServicos() const -> retorna uma std::string onde são escritos todos
    os Id's de todos os servicos de todos os
    clients guardados pela Empresa, formatados de uma
    forma amigável para o utilizador.

std::string getServicosByCliente(unsigned int NIF) const -> retorna uma std::string
    onde são escritos todos
    os Id's de um cliente registado
    com um NIF específico,

```

formatados de uma forma
amigável para o utilizador.

```
bool adicionaServicoCliente(Servico* serv, unsigned int NIF) -> adiciona um Serviço a  
um cliente com um NIF  
específico, retornando  
true caso consiga, false  
caso não consiga, ou  
lançando uma excepção do  
tipo ClienteInexistente  
caso o cliente com o NIF  
passado como argumento  
da função não exista.
```

```
bool servicoRapido(Servico* serv, Cliente* cli, std::string metodo_pagamento) ->  
  
efectua um serviço rápido a um cliente não registado,  
retornando true caso consiga e false caso não consiga.
```

```
bool pagamentoServicoCliente(unsigned int NIF) -> efectua o pagamento de um service a  
um cliente com um NIF  
  
específico, retornando true caso  
consiga ou false caso não consiga.
```

As excepções `ClienteRepetido` e `ClienteInexistente` são definidas no ficheiro `Empresa.h`.
ambas possuem os membros privados `nome` e `morada` (definidos por uma `std::string`) e o
membro `NIF` (definido por um `unsigned int`).

Possuem em si também os métodos `ClienteInexistente()` / `ClienteRepetido()`,
`ClienteInexistente(std::string nome, std::string morada)` / `ClienteRepetido(std::string
nome, std::string morada)` e `ClienteInexistente(std::string nome, std::string morada,
unsigned int NIF)` / `ClienteRepetido(std::string nome, std::string morada, unsigned int
NIF)` e ainda um método `void printInfo()` cada uma (imprime as informações relativas ao
cliente defeituoso, de uma forma amigável ao utilizador).

Servico.h / Servico.cpp

Cada Serviço foi implementado recorrendo a uma classe serviço. Esta classe os seguintes objectos privados:

```
unsigned int id;
```

id do serviço, actualizado a cada criação de um novo serviço. Deste modo, o primeiro serviço criado possui id = 1, o segundo serviço criado possui id = 2, e assim sucessivamente. Esta funcionalidade foi conseguida recorrendo a variáveis estáticas.

```
std::string origem;  
std::string destino;
```

Origem e Destino da viagem/service, respectivamente.

```
unsigned int distancia;
```

Distância da viagem/serviço, por agora não possui qualquer tipo de uso.

```
float valor;
```

Valor a pagar pelo serviço/preço do serviço. Depende da distância e hora a que é efectuado.

```
data_struct data_realizacao;  
tempo horas;
```

Hora e data de realização de serviço, respectivamente. A hora é definida por uma struct (tempo) constituída pelos objectos horas, minutos, segundos; A data é definida Também por uma struct (data_realização) e é constituída pelos objectos dia, mês e ano. Todos os objectos das duas structs são definidos através de `unsigned int`

```
bool estado;
```

Estado do objectido, podendo variar entre "true" (pago) ou "false" (por pagar).

```
std::string tipo_pagamento;
```

Tipo de pagamento utilizado para pagar o serviço, podendo variar entre as std::string's "numerario", "cartao de credito", "multibanco", "fim do mes" para os clientes registados e pelas std::string's "numerario" e "multibanco" para os clientes não Registados.

```
unsigned int NIF;
```

NIF do cliente que pagou o Serviço

Possui também os seguintes métodos públicos:

`Servico()` -> constructor de um Serviço com todos seus membros inicializados a valores Nulos

`Servico(std::string origem, std::string destino)` -> constructor de um Serviço inicializado com valores específicos de origem, destino

`Servico(std::string origem, std::string destino, data_struct dt, tempo horas)` -> constructor de um Serviço inicializado com valores específicos de origem, destino, horas e data

`~Servico()` -> destructor de um objecto Serviço

`bool operator== (const Servico &serv) const` -> operador== para dois serviços. Dois serviços são iguais se possuírem a mesma origem, o mesmo destino, a mesma hora e a mesma data.

`std::string getOrigem() const` -> retorna a origem de um serviço

`std::string getDestino() const` -> retorna o destino de um serviço

`tempo getHoras() const` -> retorna a hora de um serviço

`float getValor() const` -> retorna o valor/preço de um service

`data_struct getData() const` -> retorna a data de realização de um serviço

`unsigned int getId() const` -> retorna o Id de um destino

`std::string getInformacao() const` -> retorna toda a informação relativa a um serviço numa std::string, formatada de uma forma amigável ao utilizador

`bool getEstado() const` -> retorna o estado de pagamento de um serviço

`unsigned int getDefaultId() const` -> retorna o default_id, usado para a implementação de todos os restantes Id's de todos os serviços criados

`float getDefaultValor() const` -> retorna o valor genérico de um serviço, sem qualquer tipo de imposto acrescido

`tempo getInicioPeriodoTransito() const` -> retorna o inicio do periodo de maior trânsito

`tempo` getFimPeriodoTransito() `const` -> retorna o fim do período de maior trânsito
`float` getImpostoTransito() `const` -> retorna o imposto calculado ao preço de um
serviço

`unsigned int` getNIF() `const` -> retorna o NIF do cliente que pagou o serviço

`void` setOrigem(`std::string` origem) -> define a origem de um serviço

`void` setDestino(`std::string` destino) -> define o destino de um serviço

`void` setData(`data_struct` dt) -> define a data de realização de um serviço

`void` setHoras(`tempo` horas) -> define a hora de realização de um serviço

`void` setPago() -> define um serviço como pago

`void` setTipoPagamento(`std::string` tipo) -> define o tipo de pagamento do serviço

`void` setValor(`float` valor) -> define o valor de um serviço

`void` setNIF(`unsigned int` NIF) -> define o NIF do cliente que pagou o serviço

`void` setPeriodoTransito() -> define o period de maior hora de trânsito

`void` setImpostoTransito(`float` imposto) -> define o imposto acrescido ao valor de um
serviço

Cliente.h / Cliente.cpp

Os clientes foram implementados recorrendo a técnicas de polimorfismo. Deste modo, existe uma class mãe Cliente de onde a qual são herdados os diferentes tipos de clientes.

Da classe mãe Cliente é herdada a classe Registrado (cliente Registrado);

Da subclasse mãe Registrado são herdados os clientes Empresariais (Empresarial) e os clientes particulares (Particular).

A classe mãe Cliente possui os seguintes membros privados (herdados aos outros tipos de clientes):

```
std::string nome;
```

nome do cliente

```
std::string morada;
```

morada do cliente

```
unsigned int NIF;
```

NIF do cliente (serve como identificaro de cada cliente, já que cada cliente possui um NIF único)

A classe mãe Cliente possui os seguintes métodos públicos:

Cliente() -> constructor do default Cliente (inicializado com todos os seus membros privados a valores nulos)

Cliente(const std::string nome , const std::string morada, const unsigned int NIF) ->

Constructor de um Cliente com um nome, morada e NIF específicos

Cliente(const std::string nome, const std::string morada) -> constructor de um
Cliente com nome e
morada específicos

Cliente(std::string nome) -> constructor de um Cliente com nome específico

~Cliente() -> destructor de um objecto do tipo Cliente

bool operator== (const Cliente &cli) const -> operador == para dois clients. Dois
clients só são idênticos se possuírem
o mesmo id (neste caso, o número NIF)

bool operator< (const Cliente &cli) const -> operador < para dois clients. Um
cliente é “menor” que outro cliente se
possuir um nome de maior grandeza,
considerando-se a ordem alfabética de
organização dos clientes, pelo seu nome

void operator= (const Cliente &cli) -> operador = para dois clients. Coloca os
valores de nome, morada e NIF de um cliente
nos respectivos valores correspondentes de outro
cliente.

//METODOS GET

std::string getNome() const -> retorna o nome de um cliente

std::string getMorada() const -> retorna a morada de um cliente

unsigned int getNIF() const -> retorna o NIF de um cliente

virtual std::string getInformacao() const -> função virtual, retorna toda a informação
relativa a um cliente não registado,
formatada numa string de uma forma amigável
ao utilizado

//METODOS SET

void setNome(std::string nome) -> define um nome para um cliente

void setMorada(std::string morada) -> define uma morada para um cliente

void setNIF(unsigned int NIF) -> define um NIF para um cliente

//METODOS RELATIVOS A SERVICOS

`virtual bool pagamentoNumerario(Servico &serv)` -> função virtual que efectua o pagamento de um serviço pelo método de pagamento numerário

`virtual bool pagamentoMultibanco(Servico &serv)` -> função virtual que efectua o pagamento de um serviço pelo método de pagamento por multibanco

Cientes Registrados

A classe Regsitados, herdada da classe mãe Cliente, possui os seguintes membros privados adicionais:

```
std::vector<Servico *> historico;
```

Vector que guarda todos os (apontadores para) serviços que um cliente Registrado solicitou à Empresa.

Possui os seguintes métodos públicos:

`Registrado()` -> constructor de uma class default Registrado com todos os membros privados inicializados a valores nulos

`Registrado(std::string nome, std::string morada, const unsigned int NIF)` -> constructor de um Registrado com nome, morada e NIF específicos

`Registrado(std::string nome, std::string morada)` -> constructor de um Registrado com nome e morada específicos

`Registrado(std::string nome)` -> constructor de um Registrado com nome específico

`std::vector<Servico *> getHistorico() const` -> retorna o vector de apontadores ara
serviços de um cliente Registrado

`virtual std::string getInformacao() const` -> retorna toda a informação relativa a um
Registrado numa `std::string`, formatada
de forma amigável ao utilizador

`//METODOS RELATIVOS A SERVICOS`

`void operator=(const Registrado ®)` -> operador = para classes do tipo Registrado;
dois Registrados são iguais se possum o mesmo
valor de NIF

`virtual std::string getInformacaoServicos() const` -> retorna toda a informação
relativa aos serviços guardados
no vector “histórico” de um Registrado
numa `std::string`, formatada de uma
forma amigável para o utilizador

`virtual std::string getTipo()const` -> retorna o tipo de cliente (neste caso,
“registrado”)

`virtual bool pagamentoNumerario(Servico &serv)` -> função virtual derivada da função
de igual nome da classe mãe
Cliente. Efectua o pagamento de um
serviço através do método de pagamento
por numerario.

`virtual bool pagamentoMultibanco(Servico &serv)` -> função virtual derivada da função
da classe mãe Cliente. Efectua
o pagamento de um serviço através do
método de pagamento por multibanco.

`virtual bool pagamentoCartaoCredito(Servico &serv)` -> função virtual que efectua o
pagamento de um serviço pelo
método de “cartao de credito”,
acrescendo ao valor total do serviço
um imposto de 5%

`virtual bool pagamentoFimDoMes(std::vector<Servico *> servs)` -> função virtual que efectua o pagamento dos vários serviços passados como argumento da função num vector de apontadores para serviços pelo método de pagamento “fim do mes”, acrescentando ao valor total de todos os serviços um imposto de 2%. O pagamento só é efectuado se as datas de pagamento dos vários serviços se encontrarem entre o primeiro dia do mês passado e o último dia do mês actual.

`virtual bool pagamentoNumerario(unsigned int id)` -> função virtual derivada da função da classe mãe de mesmo nome que efectua o pagamento de um serviço com Id específico pelo método de pagamento por “numerario”

`virtual bool pagamentoMultibanco(unsigned int id)` -> função derivada da função da classe mãe de mesmo nome que efectua o pagamento de um serviço com um Id específico pelo método de pagamento por “multibanco”

`virtual bool pagamentoCartaoCredito(unsigned int id)` -> função virtual que efectua o pagamento de um serviço com um Id específico pelo método de pagamento por “cartao de credito”, acrescentando ao valor total do serviço um imposto de 5%

`virtual bool pagamentoFimDoMes(std::vector<unsigned int> ids)` -> função virtual que efectua o pagamento de vários serviços, cada um com um Id específico passados como argumento num vector de Id's pelo método de pagamento “fim do mês”. O pagamento só é efectuado se as datas de pagamento dos vários serviços se encontrarem entre o primeiro dia do mês passado e o último dia do mês actual.

NOTA: para todas as funções de pagamento, é retornado o valor true caso o pagamento seja feito com sucesso ou false caso o pagamento tenha falhado.

`bool` adicionaServico(`Servico*` serv) -> adiciona um Serviço a um cliente Registrado. Retorna true caso consiga e false caso não consiga.

Cientes Empresariais

As classe Empresarial é derivada da sub-classe mãe Registrado.

Não apresenta nenhuns valores privados adicionais.

Possui os seguintes métodos públicos:

Empresarial() -> constructor de um objecto defaulte Empresarial, inicializao com valores nulos.

Empresarial(`std::string` nome, `std::string` morada, `const unsigned int` NIF) ->
constructor de um objecto Empresarial, inicializado com valores específicos de nome, morada e NIF.

Empresarial(`std::string` nome, `std::string` morada) -> constructor de um objecto Empresarial, inicializado com valores específicos de nome e morada.

Empresarial(`std::string` nome) -> constructor de um objecto do tipo Empresarial com um nome específico

`virtual std::string` getInformacao() `const` -> função virtual derivada da função de mesmo nome da sub-classe mãe Registrado que retorna toda a informação relativa ao Empresarial numa `std::String`, formatada de forma amigável ao utilizador.

`virtual std::string` getTipo() `const` -> função virtual que retorna o tipo de Registrado que o objecto é (neste caso, retorna “empresarial”)

Cientes Particulares

Tal como o objecto Empresarial, a classe Particular é derivada da sub-classe mãe Registrado, e também não possui métodos privados adicionais.

Possui os seguintes métodos públicos:

`Particular()` -> constructor de um objecto default Particular, inicializado com todos os valores a nulo

`Particular(std::string nome, std::string morada, const unsigned int NIF)` ->
constructor de um objecto Particular com nome, morada e NIF específicos

`Particular(std::string nome, std::string morada)` -> constructor de um objecto Particular com nome e morada específicos

`Particular(std::string nome)` -> constructor de um objecto Particular com nome específico

`virtual std::string getInformacao() const` -> função virtual derivada da função de mesmo nome da sub-classe mãe Registrado que retorna toda a informação relativa ao objecto Particular, num `std::string` formatada de forma amigável ao utilizador.

`virtual std::string getTipo() const` -> função virtual que retorna o tipo de Registrado que o objecto é (neste caso, retorna "particular")

Interface.h / Interface.cpp

A interface é o meio de comunicação entre o utilizador e o programa. É implementada através de um classe Interface.

Não possui nenhuns membros privados, só métodos públicos, já que somente tem a função de manusear o programa.

Deste modo, a classe Interface possui os seguintes métodos declarados na classe:

`int` GetInput() -> solicita ao utilizador um integral e retorna esse integral. Usado na escolha das opções, em cada menu.

`void` DisplayMenuInicial() -> imprime o Menu Inicial do programa

`void` MenuInicial(`Empresa*` emp) -> chama o Menu Inicial do programa. Neste menu, são assinaladas quaisquer exceções vindas do ficheiro main.cpp e o programa comprimenta o utilizador. Imprime uma mensagem de terminação de programa quando este retorna desta função.

`void` DisplayMenuPrincipal() -> imprime o Menu Principal do programa

`void` MenuPrincipal(`Empresa*` emp) -> chama o Menu Principal do programa, tomando como argumento a Empresa do utilizador em questão. Nest menu, o utilizador tem as seguintes opções:

`0 - Informacao geral` -> imprime toda a informação relativa à Empresa em questão, de uma forma amigável ao utilizador

`1 - Informacao relativa aos Servicos` -> imprime toda a informação relativa a todos os serviços de todos os clientes da Empresa em questão, de uma forma amigável ao utilizador

2 - Efectuar servico rapido (para clientes nao registados) -> invoca uma função serviçoRápido() que permite efectuar um serviço a um cliente não registado e guardá-lo na Empresa, sem guardar dados sobre o cliente que o efectuou

3 - Adicionar Registrado (cliente registrado) -> adiciona um Registrado à Empresa

5 - Adicionar Servico a Cliente -> adiciona um service a um cliente

6 - Pagamento de Servicos -> invoca o menu de pagamento de serviços

7 - Carregar ficheiro -> permite carregar um programa a partir de ficheiros

8 - Guardar sessao actual -> permite guardar em ficheiros o programa actual

9 - Sair -> retorna ao Menu Inicial

void NovoFicheiro(Empresa* emp) -> cria um novo ficheiro

void CarregaFicheiro(Empresa* emp) -> carrega um programa novo a partir de ficheiros. Os Clientes são carregados de um ficheiro [nome_d_ficheiro]_clientes.txt; os Serviços são carregados de um [nome_do_ficheiro]_servicos.txt; A Empresa carregada de um ficheiro [nome_do_ficheiro]_empresa.txt.

void GuardaFicheiro(Empresa* emp) -> guarda o ficheiro actual em ficheiros. Os Clientes são guardados num ficheiro [nome_d_ficheiro]_clientes.txt; os Serviços são guardados num ficheiro [nome_do_ficheiro]_servicos.txt; A Empresa é guardada num ficheiro [nome_do_ficheiro]_empresa.txt.

Cliente* CriarCliente() -> invoca o menu de criação de um Cliente, retornando-o.

Registrado* CriarRegistrado() -> invoca o menu de criação de um Registrado, retornando-o.

Servico* CriarServico() -> invoca o menu de criação de um Serviço, retornando-o.

Empresa* CriarEmpresa() -> invoca o menu de criação de uma Empresa, retornando-a.

`void DisplayMenuPagamento()` -> imprime o Menu de Pagamento de uma forma amigável ao utilizador.

`void MenuPagamento(Empresa* emp)` -> chama o Menu de Pagamento. Neste, menu, o utilizador escolhe de entre os quatro tipos de pagamentos, escolhe um cliente, e escolhe um dos serviços desse cliente para pagar.

NOTA: no ficheiro Interface.cpp existem também os seguintes métodos, declarados fora da class Interface:

`void Pausa()` -> para o programa até o utilizador carregar numa tecla e chama a função `clearBuffer()`. Semelhante ao `system("pause")`.

`void clearBuffer()` -> limpa o input buffer.

Utilidades.h

Neste ficheiro estão declarados métodos e objectos auxiliares que são usados pelo resto do programa.

Estão neste ficheiro declarados os seguintes objectos:

```
struct tempo{
    unsigned int horas;
    unsigned int minutos;
    unsigned int segundos;
};
```

Struct para definição da hora a que um Serviço é realizado.

```
struct data_struct{
    unsigned int dia;
    unsigned int mes;
    unsigned int ano;
};
```

Struct para a definição da data a que um Serviço é realizado.

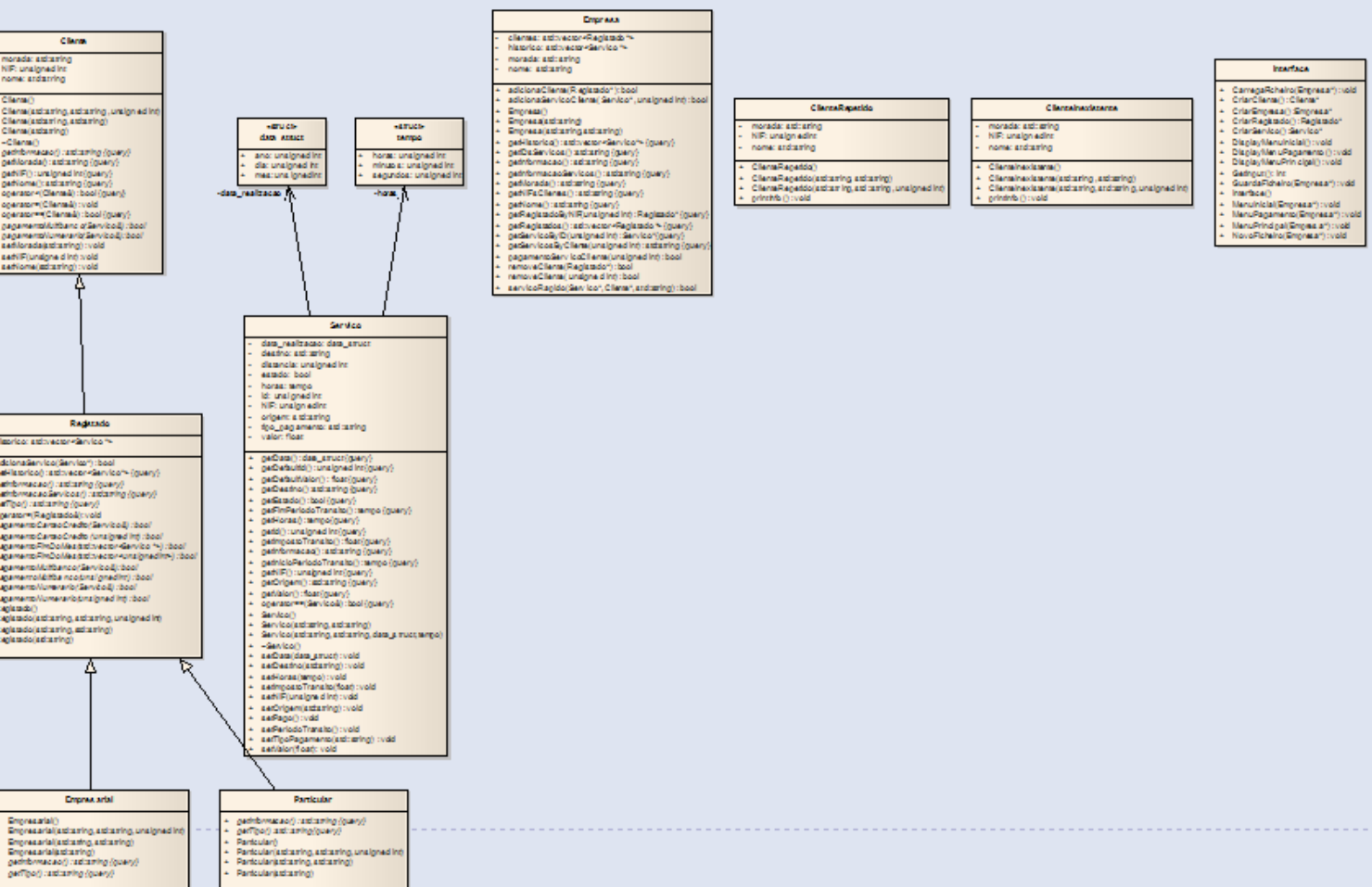
Main.cpp

O programa é executado pelo ficheiro main.cpp. Neste ficheiro foram feitos os diversos testes do programa, incluindo tratamento de excepções e declaração de objectos.

É neste ficheiro que é criada e invocada a class Interface.

O programa inicializa ao solicitar ao utilizador a hora de maior trânsito.

Diagrama de Classes (UML)



Casos de Utilização

Criação de Empresas de Táxis, ou qualquer tipo de Empresa relacionada com transportes

Alterar nome da Empresa

Adicionar Clientes à Empresa

Adicionar Serviços aos Clientes da Empresa

Remover Clientes da Empresa

Criação de Serviços

Criação de Clientes

Impressão de informação relacionada com a Empresa

Impressão de informação relacionada com os Clientes

Impressão de informação relacionada com os Serviços

Entre outros

Dificuldades Encontradas

A dificuldade principal foi criar uma estrutura segura para o programa. Devido à falta de tempo, o programa não está perfeitamente estável e a estrutura geral podia ser melhorada. Também devido a esse factor não foi possível implementar todas as funcionalidades pretendidas do programa.

Foram encontradas dificuldades ao nível do IDE Eclipse, com o qual o projecto foi primeiramente realizado. Devido a problemas com esse IDE que não foram possíveis de ser solucionados, foi usado o IDE Visual Studio Enterprise 2015, fornecido pelo Dremaspark.

O resultado final não foi o esperado, já que pode-se considerar o trabalho “incompleto”, devido à falta de tempo.

Distribuição de Trabalho pelos Elementos do Grupo

Antes da inicialização de distribuição de trabalho pelo grupo, os dois restantes membros desistiram da cadeira. Devido a este precalce, todo o trabalho foi efectuado pelo aluno Nuno Manuel Ferreira Corte-Real.