UNIVERSIDADE DO MINHO

Trabalho Prático 1

Mestrado Integrado em Engenharia Biomédica
Aplicações Distribuídas
(1ºSemestre/2014-2015)

A65078 Ana Sofia Quintas
A61777 Cármina Azevedo
A65037 Margarida Costa

Resumo

No âmbito da Curricular de Aplicações Distribuídas foi-nos proposto a realização de um trabalho prático que consistiu no desenvolvimento de um aplicação Cliente/Servidor de gestão e para uma Unidade de Saúde.

Este trabalho teve diversos objetivos dois quais se destacam a familiarização com a linguagem de programação JAVA, nomeadamente: estruturas de dados, cliente servidor utilizando JAVA RMI e de controlo de concorrência.

A problematização deste exercício prático, passou pelo desenvolvimento de um Centro Hospitalar UMINHO com mais de 100 mil utentes, 5 mil médicos 7000 enfermeiros e 3000 funcionários onde se pretendeu criar um sistema integrado de gestão de consultas, ficha clínica, atos médicos, atos de enfermagem, gestão de farmácia hospitalar. Assim, o presente relatório resulta do proposta de resolução para o exercício proposto e serve como suporte teórico para o mesmo.

Conteúdo

1 Introdução

De maneira a preencher todos os requisitos exigidos ao longo deste trabalho, recorreu-se ao Eclipse, uma plataforma de desenvolvimento de software livre extensível, baseada em Java, devido à familiarização do grupo com este IDE.
Tal, como foi proposto no enunciado deste trabalho prático, foi utilizado o Java RMI (*Remote Method Invocation*) e implementado o controlo de concorrência, explicado no capítulo 1.2.

1.1 Funcionamento do RMI

A versão utilizada do Java inclui uma API (*Application Programming Interface* conhecida como *Remote Method Invocation* (RMI) que facilita o desenvolvimento de aplicações distribuídas. O RMI permite a um objeto cliente que reside numa *Java Virtual Machine* (JVM) invocar métodos de um objeto servidor remoto que reside numa JVM distinta. O RMI está definido de uma forma que permite a sua integração na linguagem Java, no IDE (*integrated development environment*) e na JVM [1]
Numa aplicação RMI é a definição de uma interface que permite a invocação remota de métodos. Para que isto aconteça, o objeto cliente tem que possuir uma referência para o objeto remoto que inclui a localização do *host* onde o objeto remoto reside. Esta referência é obtida de registos. O objeto servidor deve disponibilizar os objetos remotos com um nome no sistema de registos ao qual o cliente acede. [1]
Uma interface é um tipo de objeto implementado por uma ou mais classes. Um objeto que se deseje que esteja acessível remotamente deve implementar pelo menos uma interface Java que, direta ou indiretamente, implementa a interface pré-definida *java.rmi.Remote*. A *Remote* não define nenhuma operação específica, contudo, define o nível de abstração de acesso remoto. [1]
Uma invocação remota de métodos é muito semelhante a uma invocação local, excepto no mecanismo utilizado para passar objetos. Por exemplo, para passar um objeto não remoto através de um método remoto, apenas uma cópia do objeto será passada, já que eles não são acessíveis remotamente. Como consequência, os objetos não remotos devem implementar a interface *Java.io.Serializable*. Deve-se ter atenção que qualquer alteração posterior efetuada nesses objetos não se irá refletir na cópia. Por outro lado, objetos que implementam a interface *Remote* não são copiados, mas sim referenciados, já que têm a capacidade de ser acedidos remotamente e, por esse motivo, qualquer alteração será visível do lado cliente. [1]
Contudo, como em qualquer aplicação, podem ocorrer falhas. Quando isto acontece a situação é comunicada ao cliente através de exceções. Qualquer operação declarada numa interface remota deve declarar que poderá lançar uma *RemoteException*. Instâncias desta classe podem ser utilizadas pelo cliente para obter a mensagem detalhada do erro. [1]
Numa aplicação Java RMI não é permitida múltipla herança de classes, mas é permitida múltipla herança para interfaces. Isto significa que uma interface pode extender várias interfaces (remotas ou não). Isto possibilita que haja polimorfismo entre objetos remotos e não remotos. [1]

1.1.1 Arquitetura RMI

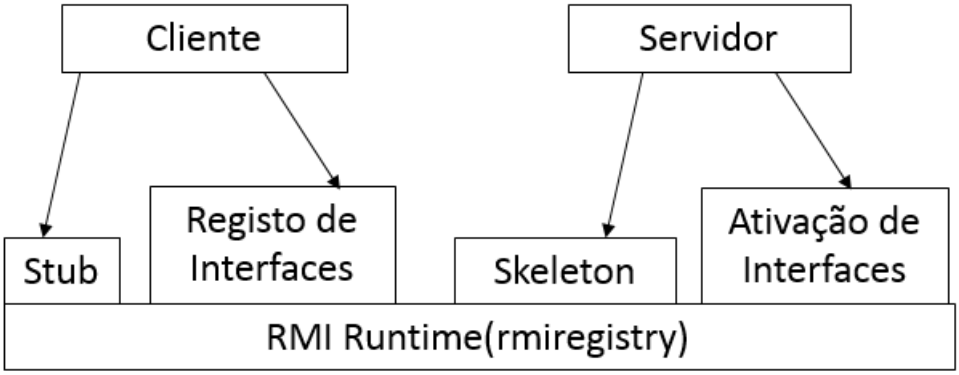

Figura 1: Representação esquemática da arquitura RMI

Na figura 1 apresenta-se um esquema da arquitetura do Java RMI. A invocação remota de métodos é iniciada pelo cliente através de um método local chamado *stub*. A camada *stub/skeleton* é responsável por receber as chamadas da aplicação cliente feitas à interface e por reencaminhá-las para o objeto remoto. O *stub* inclui todos os métodos remotos que estão disponíveis no objeto remoto, do lado do servidor. [1]

O cliente obtém referências desses objetos através do registo. O servidor regista referências dos objetos através desse registo, de forma a que o cliente os possa localizar. Objetos servidor são ativados quando é feita uma invocação remota para esse objeto ou, explicitamente por algum administrador. Em último caso, o objeto deverá ter a capacidade de ser ativado e de usar interfaces de ativação para se registar a si mesmo. Durante a ativação, a interface chama o construtor do objeto permitindo-lhe restaurar o seu estado. Uma vez terminada a ativação controlo é passsado para o *skeleton* que invoca o método remoto. [1]

No fundo, o papel dos *stubs* e *skeletons* consiste em mascarar e desmascarar a invocação de parâmetros e implementar a camada de apresentação. O mecanismo de geração de *stubs* e *skeletons* ocorre automaticamente a partir da definição da interface. [1]

Um outro conceito importante acerca da ativação de um objeto diz respeito à referência de falha (*faulting reference*) que se trata de uma referência inteligente que contém um ativador de identficação que fornece a informação necessária à ativação do objeto. Inclui também uma *live reference*, possivelmente do tipo *void*. Se a referência de falha for utilizada na invocação de um método, então ela irá confirma se a *live reference* corresponde efetivamente ao objeto ativo. Se for esse o caso, então a *live reference* é utilizada para se proceder com a invocação. Caso contrário é utilizado o identificador de ativação é utilizado para encontrar o *host* remoto onde o objeto deverá ser ativado. [1]

Estas operações são todas transparentes do lado do cliente. O cliente apenas necessita de obter uma referência para o objeto remoto através do registo e utiliza-a para invocar métodos remotos. A ligação entre as máquinas virturais é realizada na camada de transporte através dos protocolo TCP/IP. [1]

1.2 O controlo de concorrência

Por norma, os servidores têm mais que um objeto cliente. Por essa razão, pode acontecer, por exemplo, dois clientes requisitarem um pedido de acesso a um objeto remoto em paralelo, o que resulta em interferências, denominadas *lost updates*. Na realidade, isto acontece quando duas *threads* tentam modificar um objeto partilhado. A anomalia que ocorre é intitulada *inconsistent analysis* e esta é, claramente, uma situação indesejada. No sentido de resolver este problema recorre-se ao controlo de concorrência que tem como principal objeto efetuar a serialização.
A técnica mais popular de serialização é o *two-phase locking* e baseia-se na ideia de pedir um *lock* de prioridades para objetos partilhados. O controlo de concorrência apenas disponibiliza um *lock* para determinado recurso se isso não interferir com *locks* garantidos previamente. A partir do momento em que já não seja necessário acessar esse objeto, o *lock* é libertado. É importante que esta libertação ocorra o mais cedo possível para que um outro processo que necessite de aceder a esse objeto possa adquirir o seu *lock*. Uma abordagem simples desta logística é a de entender os *locks* como informação binária, na qual os objetos estão, ou não, relacionados com um *lock*. A implementação de métodos *synchronized* utiliza os locks binários. Se uma *thread* tenta efetuar o *lock* de um objeto que já está em *lock* , o controlo de concorrência irá forçar que essa *thread* aguarde até que o *lock* seja libertado. Contudo, esta opção pode, por vezes, ser desnecessária, por exemplo, quando um processo apenas pretende ler um objeto. O controlo de concorrência poderá permitir que múltiplas operações leiam o mesmo objeto, mas terá que prevenir que informações desse objeto sejam alteradas enquanto ele é lido. Para evitar, então, restrições de concorrência desnecessárias, o controlo de concorrência distingue os *locks* com diferentes *models*: para ler é utilizado um *read lock* e para modificar é utilizado um *write lock*. Para que isto possa ocorrer deve ser efetuado um registo de compatibilidade entre *locks*, onde se encontram registados todos os *locks* permitidos anteriormente . [1]

2 Descriçao do Trabalho e Resultados

Os processos participantes da aplicação hospital incluem um projeto Cliente, Servidor e um sistema de registo de objetos. De forma geral, o cliente invoca métodos de objetos remotos; o servidor implementa os métodos remotos; o registo é um servidor que associa aos objetos nomes com os quais estes são registados, efetuando o registo.

2.1 Servidor

O projeto Servidor inclui as classes que representam as entidades utente, ficha clínica, médico, enfermeiro, funcionário, medicamento, produto, ato médico, prescrição, ato de enfermagem, encomendas de medicamentos e encomendas de produtos. Cada uma destas classes contém os atributos próprios da respetiva entidade, o construtor da classe e métodos *getters* e *setters* para cada atributo. O diagrama de classes que representa as correspondências entre estas classes consta do anexo I.
Para além das classes enumeradas, este projeto contém, ainda duas classes especiais: o Servidor e a Database.

2.1.1 Classe Servidor

A classe servidor implementa um sistema de registos que corre na porta 1098.Antes de invocar um método remoto, o cliente deve contactar com o registo para obter acesso ao objeto remoto.

```
package servidor;
import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;



import servidor.DataBase;



public class Servidor {
        private static DataBase stubAdmin;


        public static void carregar() throws RemoteException, IOException {
```

```
            try {

                    FileInputStream fis = new
        FileInputStream("dados.dat");
                    BufferedInputStream buffer =
        new BufferedInputStream(fis);
                    ObjectInputStream ois =
        new ObjectInputStream(buffer);
                    stubAdmin = (DataBase)
        ois.readObject();
                    buffer.close();
                    ois.close();
                    System.out.println("Dados
        carregados de ficheiro!");
            } catch (Exception ex) {
                    System.err.println("O ficheiro
        nao existe");
                    stubAdmin = new DataBase();
            }
    }

    public static void main(String args[])
throws RemoteException,
    FileNotFoundException,
IOException, ClassNotFoundException
{
            if(System.getSecurityManager()
  ==null){
                    System.setSecurityManager
        (new SecurityManager());
            }
            System.err.println("A correr!");
            carregar();
            System.err.println("Servidor pronto!");
            try {
                    Registry registry = LocateRegistry.createRegistry(1098);

                    registry.rebind("Administradores",
        stubAdmin);
                    registry.rebind("Guardar",
        stubAdmin);
                    registry.rebind("Medicos",
```

```
                stubAdmin);
                        registry.rebind("Funcionarios",
                stubAdmin);
                        registry.rebind("Enfermeiros",
                stubAdmin);




                        (new Thread() {
                                @Override
                                public void run() {
                                        try {
                                                while(true) {
                                                        stubAdmin.guardar();
                                                        Thread.sleep(5 *
                            60 * 1000);
                            // guarda de 5 em 5 minutos
                                                }
                                        } catch (RemoteException e) {
                                                e.printStackTrace();
                                        } catch (InterruptedException e) {
                                                e.printStackTrace();
                                        }
                                }
                        }).start();


                } catch (RemoteException e) {
                        e.printStackTrace();
                }



        }
}
```

Inicialmente é declarado o *stub* Administrador. Do ponto de vista do cliente, um stub representa o objeto servidor. Do ponto de vista do servidor, o stub representa o objeto cliente. Os pedidos de acesso a objetos remotos utilizam os stubs para organizar dados num computador de forma patronizada para que estes possam ser lidos por diferentes aplicações. Para esta aplicação escolheu-se ter apenas o *stub* Administrador já que esta é a interface correspondente à classe

DataBase, onde estão implementados todos os métodos necessários para que se aceda aos objetos remotos. No anexo II encontra-se representado um diagrama UML que elucida esta relação.
Esta classe contém um método main que é chamado quando o ficheiro Servidor.java corre e inicia a execução da *thread* que permite a gravação de dados a cada 5 minutos, através de um ciclo infinito *while(true)....* . Para isto é invocado o método guardar() que se encontra localizado na classe DataBase (é explicado na subsecção seguinte).
É de notar que esta classe implementa a interface Runnable, que define o método run(), que contém o código executado na *Thread*. Implementando esta interface é providenciado um objeto Runnable. Este objeto é passado ao construtor da *Thread*.
Para além disso tem dois métodos especialmente importantes: o carregar() e o run(). O método carregar() carrega dados para o um ficheiro binário, gravando instâncias da classe Administrador, através de um *BufferedInputStream ObjectInputStream* imprimindo, após este processo uma mensagem que indica que os dados foram carregados. No caso de não ser possível é lançada uma excepção que comunica ao utilizador que o ficheiro não existe.
O método run() implementa um sistema de registos que corre na porta 1098 . Uma vez iniciado, o servidor regista uma instância da interface administrador com o nome no sistema "Administradores", através da geração de um *stub*, ocorrendo o mesmo para os registos "Guardar", "Medicos", "Funcionarios", "Enfermeiros". Em seguida espera-se um input do teclado, que caso se trate da string "quit"então chama o método guardar() para garantir que os dados não são perdidos quando o programa é interrompido. Seguidamente, o servidor é desligado.

2.1.2 Classe DataBase

A classe DataBase representa o hospital, pois é ela não é nada mais do que o conjunto de todos os objectos existentes num hospital (Entidades: Medico, Utente, Funcionario, Medicamento...), de todas as relações existentes e ações praticadas entre elas (Métodos: adiciona, remove e altera utente, agenda consuta, passa prescrição,...).

Na implementação do programa escolheu-se os HashMaps como estruturas de armazenamento de dados principais, já que permitem aceder a vários valores correspondentes a uma chave; são de fácil implementação e são extremamente eficientes. Os HashMaps utilizados encontram-se definidos no início da classe DataBase que se encontra abaixo:

```
package Servidor;
import java.rmi.RemoteException;


import java.rmi.server.UnicastRemoteObject;
import java.util.Collections;
import java.util.GregorianCalendar;
import java.util.HashMap;
import java.util.Map;
import java.io.Serializable;
import java.util.HashSet;
```

```
import InterfaceAD.Administrador;

public class DataBase extends UnicastRemoteObject implements
Administrador, Serializable {
    /**
         *
         */
        private static final long serialVersionUID = 1L;


//id,entidade
public HashMap<String, Utente> utentes;
public HashMap<String, Medicamento> medicamentos;
public HashMap<String, Medico> medicos ;
public HashMap<String, Enfermeiro> enfermeiros ;
public HashMap<String, Funcionario> funcionarios ;
public HashMap<String, Produto> produtos;
public HashMap<String, AtoMedico> atosmedicos_passados;
public HashMap<String, AtoMedico> atosmedicos_futuros;
public HashMap<String, AtoEnfermagem> atosenfermagem_passados;
public HashMap<String, AtoEnfermagem> atosenfermagem_futuros;
public HashMap<String, Prescricao> prescricoes ;
public HashMap<String, EncomendaMedicamento> encomendasmedicamentos ;
public HashMap<String, EncomendaProduto> encomendasprodutos ;
public HashMap<String, FichaClinica> fichasclinicas;


public DataBase () throws RemoteException {

        this.utentes=new HashMap<String,Utente>();
        this.medicamentos=new HashMap<String,Medicamento>();
        this.medicos=new HashMap<String,Medico>();
        this.enfermeiros=new HashMap<String,Enfermeiro>();
        this.funcionarios=new HashMap<String,Funcionario>();
        this.produtos=new HashMap<String,Produto>();
        this.atosmedicos_passados=new HashMap<String,AtoMedico>();
        this.atosmedicos_futuros=new HashMap<String,AtoMedico>();
        this.atosenfermagem_passados=new HashMap<String,AtoEnfermagem>();
        this.atosenfermagem_futuros=new HashMap<String,AtoEnfermagem>();
        this.prescricoes=new HashMap<String,Prescricao>();
        this.encomendasmedicamentos=new HashMap<String,EncomendaMedicamento>();
```

```
        this.encomendasprodutos=new HashMap<String,EncomendaProduto>();
        this.fichasclinicas=new HashMap<String,FichaClinica>();


}
```

Nesta classe está contido o método público guardar() que é invocado na classe Servidor. Neste método está contido um ciclo *try (...) catch* que cria um ficheiro de dados de extensão .dat a partir de um *FileOutputStream*. Seguidamente, permite que se possa escrever para esse ficheiro através de um *ObjectOutputStream* (oos) que recebe o *BufferedOutputStream* (buffer). Uma vez escritos os dados no ficheiro, é feito o *close* do oos e do buffer e é impressa uma mensagem que transmite ao utilizador que os dados foram gravados com sucesso. Caso ocorra algum erro, essa exceção é apanhada no *catch* e impressa. O código correspondente ao método *guardar()* segue abaixo.

```
public void guardar() throws RemoteException{


            try {
                    OutputStream fos = new FileOutputStream("dados.dat");
                    BufferedOutputStream buffer = new
        BufferedOutputStream(fos);
                    ObjectOutputStream oos = new
        ObjectOutputStream(buffer);
                    oos.writeObject(this);
                    oos.flush();
                    fos.close();
                    oos.close();
                    System.out.println("Dados binarios gravados!");
            } catch (IOException e) {
                    e.printStackTrace();
            }
    }
```

Como foi dito anteriormente, os métodos criados nesta classe permitiram recriar as ações executadas num hospital, assim como relacionar as todas as classes de entidades criadas. A figura 2 ilustra algumas destas relações entre entidades e métodos criados que serão de seguida, sucimente, descritos.

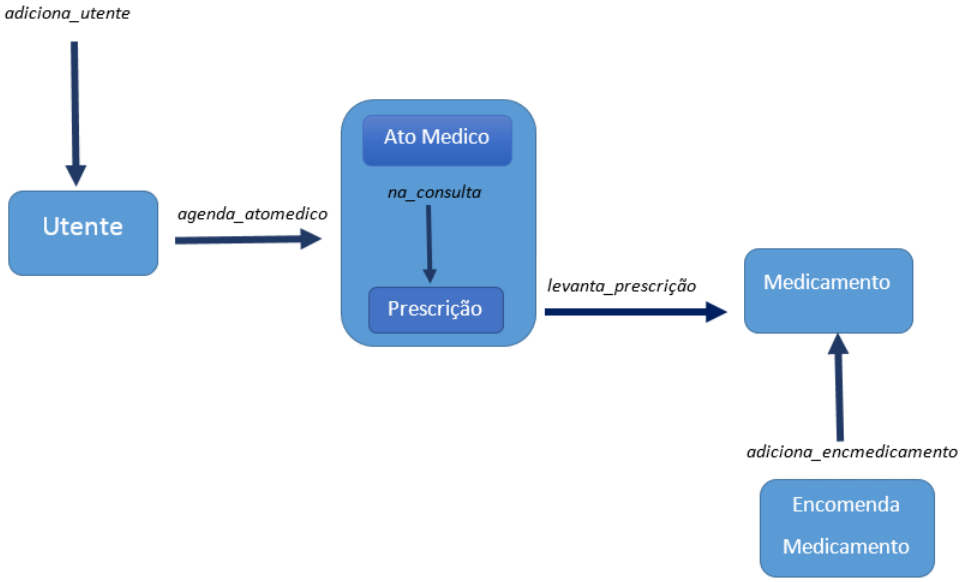

Figura 2: Representação de parte do hospital reltivo às entidades Utente, Ato Médico, Medicamento e Encomenda.

- ***Adiciona Utente***

  O objetivo principal deste método é criar objetos do tipo Utente e adicioná-los HashMap utentes da classe DataBase, por outras palavras adicionar utentes ao hospital. Deve-se notar que este método é *synchronized*. Isto significa que o cliente é bloqueado enquanto o servidor efetua uma operação pedida. O controlo apenas retorna ao cliente quando a execução do método termina ou quando ocorre um erro.

```
    public synchronized String adiciona_utente(String morada, int BI,
int contato, GregorianCalendar data_nascimento,
        String nome_utente) throws RemoteException {
            if (existe_bi(BI)) {
                    return "Utente ja existente";
            } else {
                    Utente u = new Utente(idUtente++ , morada,
        BI, contato, data_nascimento, nome_utente);
                    utentes.put(u.getCod_utente(), u);
                    return "Utente adicionado com sucesso - " +
        u.getCod_utente();
            }
    }
```

Este método recebe o construtor da classe Utente. Em primeiro lugar, é verificado se o utente já existe, através do método existe_bi. Este método retorna um valor boolean. Caso esse valor seja *true* então é lançada uma exceção remota e o utente não é adicio-

nado, comunicando-se ao utilizador uma *string* referindo que o utente já existe. No caso do método retornar *false* então pode-se proceder à criação do novo utente através da instrução: *Utente u = new Utente(idUtente++ , morada, BI, contato, data_nascimento, nome_utente);*. Em seguida é necessário adicionar esse utente ao HashMap *utentes* que guarda todos os utentes do hospital. Uma vez concluído o processo, o método adiciona_utente retorna uma *string* que indica ao utilizador que o utente foi adicionado com sucesso e o respetivo código de utente que o identifica inequivocamente. Em seguida apresenta-se o método existe_bi:

```
public boolean existe_bi(int bi_utente)throws RemoteException{
            for(Utente ep:this.utentes.values()){
                    if(ep.getBI()== bi_utente) {
                            return true;
                    }
            }
            return false;
    }
```

- ***Agenda consulta***
  O método agenda_atomedico é em tudo semelhante ao adiciona_utente. O seu principal objetivo é criar objetos do tipo AtoMedico e adicioná-los HashMap tosmedicos_futuros da classe DataBase, por outras palavras, o seu objetivo é agendar consultas médicas no hospital. Este método é igualmente *synchronized* para impedir que, caso dois clientes tentem aceder simultaneamente a mesma posição do HashMap, não ocorram problemas.

```
        public synchronized String agenda_atomedico (String cod_medico,
String cod_utente,
                        GregorianCalendar data_AtoMedico)
 throws
RemoteException {
                if(medicos.containsKey(cod_medico)==
false ){
                        return "Medico nao existente na
 base de dados!";
                }
                if(utentes.containsKey(cod_utente)
==false){
                        return "Utente nao existente na base de dados!";
                }
                if(comparadata(cod_medico, data_AtoMedico)){
                        return "Impossivel agendar
```

```
ato medico para esta hora!";
                }
 if(atosmedicos_futuros.values().contains(cod_utente)
&& atosmedicos_futuros.values().contains
(data_AtoMedico)){
return "Ja tem umaconsulta agendada para esta hora!";
                }
                else{
                        AtoMedico am = new AtoMedico(idAtoMedico++,
cod_medico, cod_utente, data_AtoMedico);
                        atosmedicos_futuros.put(am.getCod_AtoMedico(), am);
                        return "Ato medico "+ am.getCod_AtoMedico()+
" agendado com sucesso";
                }

        }
```

Um ato médico só será agendado caso não ocorra uma das exceções:

1. O médico para o qual se pretende marcar o ato médico não existir;
2. O utente para o qual se pretende marcar o ato médico não existir;
3. O médico para o qual se pretende marcar o ato médico já tenha uma consulta agendada para a data e hora desejadas;
4. O utente para o qual se pretende marcar o ato médico já tenha uma consulta agendada para a data e hora desejadas;

Para comprarar as datas ds consultas foi desenvolvidada a função:

```
        public boolean comparadata(String cod_med,
GregorianCalendar data_AtoMedico) throws RemoteException {
    for(AtoMedico a:this.atosmedicos_futuros.values()){
 if(a.getCod_medico().equals(cod_med) )
 if(a.getData_AtoMedico().equals(data_AtoMedico))
  return true;}
}       return false;
        }
```

- ***Na consulta***

  O método na_consulta tem o prepósito de simular as ações que decorrem durante uma consulta, neste caso passar uma prescrição, passar o ato médico de previsto para passado, e ainda, adicioná-lo ao historial de atos médicos da ficha do utente.

```
public String na_consulta (String cod_consulta, HashMap
<String,Integer>quant) throws RemoteException{
if (atosmedicos_futuros.containsKey(cod_consulta)){
AtoMedico consulta= atosmedicos_futuros.get(cod_consulta);
                for(String a:quant.keySet()){
                if(medicamentos.containsKey(a)==false){
                return "Chave do medicamento " + a + " invalida.";
                }
}
        consulta.prescricao.setQuantidades(quant);

        consulta.prescricao.setPodelevantar(true);
                atosmedicos_passados.put
(cod_consulta, consulta);
                        fichasclinicas.get(consulta.getCod_utente()).
adiciona_atomed(cod_consulta, consulta);
atosmedicos_futuros.remove(cod_consulta);
        return "Prescricao adicionada com sucesso!";
}else{
        return "Impossivel adicionar prescricao.
Agende uma consulta!";
                }
        }
```

A primeira coisa que a função faz é procurar o ato médico agendado, caso ele não exista, o utilizador será informado deste facto; caso exista acontecerá o seguinte:

1. **Passar uma prescrição:** no momento de agendar um ato médico, a prescrição não é, obviamente, preenchida. É entao neste ponto, que através do código do ato médico se vai buscar o ato pretendido (get) ao HashMap atosmedicos_futuros e a sua respetiva prescrição e se altera (set) o HashMap *quantidades* contido na classe Prescrição . Este processo ocorre caso os medicamentos que se pretendam prescrever existam na base de dados.Para isso é necessário percorrer o HashMap *quant* dado pelo utilizador e verificar se o código de cada medicamento existe no HashMap da DataBase *medicamentos*.
2. **Pode levantar:** A classe Prescrição contém uma variável *podelevantar* que é gerada a *false* por *default*. Após ser passada a receita altera-se, então, o valor boolean para *true*.
3. **Passagem de ato médico previsto a passdo:** Visto que se está no decorrer da consulta, esta será adicionada ao HashMap da DataBase atosmedicos_passados, assim como à ficha clinica do utente. Logicamente, será então removido do HashMap da DataBase atosmedicos_futuro.

- ***Levanta prescrição***

  O método levanta_prcrição tem por objetivo permtir que um utente levante na farmácia hospitalar a medicação que lhe foi prescrita em determinadas consultas, unicamentevatravés do código de utente.

```
public String levanta_prescricao
(String cod_utente) throws RemoteException {
HashSet<String> medicacao
levantada=new HashSet<String>();
 String text_result="";
        for (AtoMedico am: this.fichasclinicas.get(cod_utente).
getHistorial_AtoMedico().values()){
        if(am.prescricao.isPodelevantar()==true)
{
        boolean existe_quant=true;
        for (String cod_medicamento: am.prescricao.quantidades.keySet()){
                if(this.medicamentos.get
(cod_medicamento).
getQuantidade()<am.prescricao.
quantidades.get
(cod_medicamento)){
                existe_quant=false;
                        }
        }
        if (existe_quant){
        for (String cod_medicamento:
 am.prescricao.quantidades.
keySet()){
                this.medicamentos.get
(cod_medicamento).
dec_qt_medicamento
(am.
prescricao.quantidades.
get(cod_medicamento));
        medicacao_levantada.add
(cod_medicamento);
                }
                this.atosmedicos_passados.get
(am.getCod_AtoMedico()).
prescricao.setPodelevantar
(false);
        am.prescricao.setPodelevantar
```

```
(false);
        text_result+= " Prescricao da
consulta "+ am.getCod_AtoMedico()+" levantada com sucesso. \n";
        }else{
           text_result+="Nao existe stock
suficiente para
efetuar o levantamento da prescricao
referente "+
am.getCod_AtoMedico()+"."+"\n";
                }
        }
}
return text_result + " \n
Medicamentos levantados: "+
medicacao_levantada;}
```

A primeira coisa que a função faz é procurar os atos médicos existentes no historial da ficha clinica do utentes, para os quais a variável *podelevantar* esteja a true. Sendo assim, uma limitação desta função é que obriga o utente a levantar simultaneamente todas as prescrições por levantar registadas no seu historial.

O passo seguidamente efetuado é verificar se existe em stock, suficientes medicamentos, de cada um dos prescritos em cada ato médico, para vender. Caso não exista quantidade suficiente de um deles utiliza-se a variável boleana *existe_ quant* com o valor *false* e nenhum medicamento poderá ser levantado.

Caso contrário, para cada Medicamento no hashMap da DataBase medicamentos, decrementa-se a quantidade prescrita no HashMap quantidades pertencente a classe Prescrição.

```
Classe Medicamento:

        public void dec_qt_medicamento
(int receitado) throws RemoteException{
                int a= this.getQuantidade()-
receitado;
                this.setQuantidade(a);
        }
```

Por último, coloca-se a variável *podelevantar* a *false*, tanto na ficha clínica do utente como no registo de atos médicos passados do hospital.

- ***Encomenda medicamento***

  A classe medicamento contém o nome e código de um medicamento e a sua respectiva

quantidade existente no stock da farmácia hospitalar. Sendo assim, do mesmo modo que a função levanta_prescricao vai decrementando esta quantidade, criaram-se encomendas geridas por funcionários do hospital, que no atonda sua receção incrementam o stock.

```
public String recebe_encomendamedica
(String cod_encmedica, String
cod_func_rec) throws RemoteException {
        if(encomendasmedicamentos.
containsKey(cod_encmedica)){
        this.encomendasmedicamentos.
get(cod_encmedica).setRecebido(true);
        this.medicamentos.get(this.
\encomendasmedicamentos.get(cod_encmedica).getCod_medicamento()).
inc_qt_medicamento
(encomendasmedicamentos.get(cod_encmedica).
getQuantidade());
        this.encomendasmedicamentos.get
(cod_encmedica).setCod_funcionario_recebeu
(cod_func_rec);
        return "Encomenda recebida
com sucesso.";
}else{
        return "Esta encomenda nao foi
realizada.";
                }
}
```

O método recebe_encomendamedica começa por verificar se a encomenda foi realmente efetuada através do seu código. Para tal, no momento da receçao da encomenda o funcionário terá de saber o seu código, dado no momento em que a encomenda foi registada. Caso a encomenda exista a quantidade dos Medicamentos encomendados será incrementada, a variável *Recebido* alterada para o valor *true*, e o campo destinado à identificação do funcionário que recebeu a encomenda atualizado.

Classe Medicamento:

```
        public void inc_qt_medicamento(int rec)
throws RemoteException{
                int a= this.getQuantidade()+ rec;
                this.setQuantidade(a);
        }
```

Por último, coloca-se a variável *podelevantar* a *false*, tanto na ficha clínica do utente como no registo de atos médicos passados do hospital.

2.1.3 Entidades

Optou-se por explicar apenas uma entidade, uma vez que todas estão estruturadas de acordo com uma metodologia idêntica. Em seguida apresenta-se uma parte da classe utente:

```
package Servidor;
import java.io.Serializable;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.GregorianCalendar;

import InterfaceAD.IUtente;

public class Utente extends UnicastRemoteObject  implements IUtente,
Serializable  {

        /**
         *
         */
        private static final long serialVersionUID = 1L;

        static long id = 0;

        private String cod_utente;
        private String nome_utente;
        private int BI;
        private String morada;
        private int contato;
        private GregorianCalendar data_nascimento;

        public Utente(String morada, int BI, int contato,
    GregorianCalendar data_nascimento, String nome_utente)
        throws RemoteException {

        id++;

        this.cod_utente= "U" + String.valueOf(id);
        this.BI=BI;
        this.morada=morada;
        this.contato=contato;
```

```
        this.data_nascimento=data_nascimento;
        this.nome_utente=nome_utente;

    public Utente()  throws RemoteException{

        id++;
        this.cod_utente= "U" + String.valueOf(id);
        }
}
```

Todas as entidades (classes remotas) devem implementar a interface remota correspondente. No exemplo acima, verifica-se que é implementada a interface IUtente. Deve também ser implementada a classe *Serializable* que torna objetos *serializable*. Isto significa que o conteúdo desses objetos pode ser representado como um conjunto de *bytes* e depois reconstruído para os seus respetivos valores. Isto torna-se útil, por exemplo, para guarda o conteúdo de um objeto num ficheiro e, mais tarde, ler o conteúdo deve ficheiro. No caso específico do RMI, é de grande utilidade tornar um objeto *serializable* para o passar como parâmetro para um método remoto ou para o receber como resultado de um método remoto. Nesta situação, não é comunicado um conjunto de *bytes* mas sim o valor do objeto em si.
Uma vez implementada a interface *Serializable* é associado à classe um número de série denominado *serialVersionUID* que é utilizado para verificar que quem recebe e envia um objeto *serializable* carregou classes para esse objeto que sejam compatíveis com a serialização.
Para além disso, cada classe remota deve extender a classe *UnicastRemoteObject*.Os objetos desta classe existem no endereço da classe *servidor* e podem ser invocados, remotamente. Todas as entidades incluem um construtor que é utilizado pelo servidor para definir a mensagem que será enviada ao cliente. Neste caso, o construtor *Utente* recebe as *strings* morada, nome e código do utente; os inteiros contacto e número de BI e a data de nascimento que é do tipo *GregorianCalendar*. Utiliza-se, ainda um construtor especial que tem o objetivo de gerar códigos inicializados com a letra "U"seguida de um *long* id que é inicializado em 0 (quando ainda não existe nenhum utente) e incrementa o valor à medida que são criados objetos do tipo Utente. Para além disto, cada entidade contém métodos *get* e *set* que deve lançar exceções remotas, no caso de o procedimento remoto falhar:

```
public String getCod_utente() throws RemoteException{
                return cod_utente;
        }

        public void setCod_utente(String cod_utente)
        throws RemoteException{
                        this.cod_utente = cod_utente;
        }
```

```
    public String getNome_utente()
    throws RemoteException{
                    return nome_utente;
    }

    public void setNome_utente(String nome_utente)
throws RemoteException{
                    this.nome_utente = nome_utente;
    }

    public int getBI() throws RemoteException{
            return BI;
    }

    public void setBI(int bI) throws RemoteException{
            BI = bI;
    }

    public String getMorada() throws RemoteException{
            return morada;
    }

    public void setMorada(String morada) throws
RemoteException{
            this.morada = morada;
    }

    public int getContato() throws RemoteException{
            return contato;
    }

    public void setContato(int contato) throws
RemoteException{
            this.contato = contato;
    }

    public GregorianCalendar getData_nascimento() throws
RemoteException{
            return data_nascimento;
    }

    public void setData_nascimento(GregorianCalendar
```

```
data_nascimento) throws RemoteException{
        this.data_nascimento = data_nascimento;
    }
```

Para finalizar, implementou-se os métodos *hashCode* e *equals* que introduzem a particularidade de que, para dois objetos da classe Utente serem iguais, então deverão ter o mesmo valor inteiro de *hashCode()*. A implementação destes métodos segue abaixo:

```
@Override
public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result
                        + ((cod_utente == null) ? 0 :
        cod_utente.hashCode());
        return result;
}

@Override
public boolean equals(Object obj) {
        if (this == obj)
                return true;
        if (obj == null)
                return false;
        if (getClass() != obj.getClass())
                return false;
        Utente other = (Utente) obj;
        if (cod_utente == null) {
                if (other.cod_utente != null)
                        return false;
        } else if (!cod_utente.equals(other.cod_utente))
                return false;
        return true;
}

@Override
public String toString() {
        return "Utente [cod_utente=" + cod_utente + ",
        nome_utente=" + nome_utente + ", BI=" + BI +
    ", morada=" + morada + ", contato=" + contato + ",
    data_nascimento=" + data_nascimento + "]"; }
}
```

2.2 InterfaceAD

No sentido de construir uma aplicação RMI, todas as classes remotas devem ser definas pela própria classe (explicitadas na secção 2.1.3) e por uma interface. Assim, todas as entidades representadas no projeto Servidor têm uma definição de interface remota correspondente. No anexo II encontra-se um diagrama de relacionamento entre as classes DataBase e Interface e as classes de quais estas são donas.

2.2.1 Administrador

A classe administrador inclui todos os métodos que se encontram na classe DataBase. O cliente acede aos métodos desta classe a partir da interface.

2.2.2 Interfaces de entidades

No seguimento da apresentação da classe Utente, escolheu-se apenas apresentar a interface que lhe corresponde, a IUtente.

```
package InterfaceAD;
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.GregorianCalendar;

public interface IUtente extends Remote {

        public String getCod_utente() throws RemoteException;

        public void setCod_utente(String cod_utente) throws
    RemoteException;

        public String getNome_utente() throws RemoteException;

        public void setNome_utente(String nome_utente) throws
    RemoteException;

        public int getBI() throws RemoteException;

        public void setBI(int bI) throws RemoteException;

        public String getMorada() throws RemoteException;

        public void setMorada(String morada) throws RemoteException;

        public int getContato() throws RemoteException;
```

```
        public void setContato(int contato) throws RemoteException;

        public GregorianCalendar getData_nascimento() throws
    RemoteException;

        public void setData_nascimento(GregorianCalendar
    data_nascimento) throws RemoteException;

}
```

As interfaces devem ser públicas e devem estender *java.rmi.Remote*. Cada método declarado numa interface deve lançar uma *java.rmi.RemoteException* e deve corresponder àqueles que constam da classe remota correspondente.

2.3 Cliente

2.3.1 Menu

No sentido de iniciar um programa cliente que permita aceder a todas as operações disponibilizadas no projeto Cliente criou-se uma classe Menu que tem um método *main* que invoca todas os métodos *main* das outras classes do projeto Cliente, consoante a operação desejada pelo utilizador, utilizando, para tal, ciclos *do (...) while*. Inicialmente é feito o *lookup* ao sistema de registos relativo ao Administrador. O código incluído nesta classe segue abaixo.

```
package Cliente;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.util.Scanner;
import inter.Administrador;
public class Menu {
static Registry registry;
static Administrador ad;
@SuppressWarnings("deprecation")
public static void main(String args[]) throws RemoteException,
NotBoundException {
try {
 registry = LocateRegistry.getRegistry("127.0.0.1",
 1098);
```

```
ad = (Administrador)    registry.lookup("Administradores");
Scanner s = new Scanner(System.in);
int a = 0;
do {
        System.out.println("Indique qual opcao pretende
    consultar");
        System.out.println("1- Administrador");
        System.out.println("2- Funcionario");
        System.out.println("3- Medico");
        System.out.println("4- Enfermeiro");
        System.out.println("5- Sair");
        a = s.nextInt();
        switch (a) {
        case 1:
          int b;
                do{
                  System.out.println("Indique o servico
          que deseja consultar:");
                  System.out.println("1-Gerir dados pessoais do
pessoal qualificado");
                  System.out.println("2-Estatisticas
hospitalares");
                 System.out.println("3-Sair");
                 b = s.nextInt();

                 switch (b) {
                 case 1:
                   int c;
                         do{
                                 System.out.println("1-Medicos");
                                 System.out.println("2-Enfermeiros");
                                 System.out.println("3-Funcionarios");
                                 System.out.println("4-Retornar ao menu
                inicial");
                                 c = s.nextInt();
                                 if(c==1){
                          AdicionaRemoveProcuraAlteraMedico.main(args);
                         }
                                 if(c==2){
                          AdicionaRemoveProcuraAlteraEnfermeiro.main(args);
                         }
                                 if(c==3){
```

```
                    AdicionaRemoveProcuraAlteraFuncionario.main(args);
                }
            }while (c!=4);
            break;

            case 2:
            Estatisticas.main(args);
            default:
            break;
                }
     }while(b!=3);
     break;

     case 2:
     int d;
     do{
       System.out.println("Indique o servico que deseja
         consultar:");
            System.out.println("1-Gerir dados pessoais dos
     utentes");
            System.out.println("2-Gerir Consultas");
            System.out.println("3-Gerir Encomendas produtos
    de enfermagem e medicamentos");
            System.out.println("4-Gerir Stocks de produtos de
    enfermagem e medicamentos");
            System.out.println("5-Sair");
            d = s.nextInt();
            if(d==1){
                        AdicionaRemoveProcuraAlteraUtente.main(args);
            }
            if(d==2){
            GerirConsultas.main(args);
            }
            if(d==3){
            Encomendas.main(args);
            }
            if(d==4){
            Stock.main(args);
            }
    }while(d!=5);
    break;
    case 3:
```

```
            NoAtoMedico.main(args);
            break;

            case 4:
            NoAtoEnfermagem.main(args);
            break;
            default:
            break;
            }
      } while (a != 5);
} catch (IOException e) {
                        e.printStackTrace();
                  }
            }
}
```

2.3.2 Adicionar, Remover, Procurar e Alterar

Na sequência das apresentações de código anteriores, optou-se por representar, da parte do cliente, apenas a classe que permite a adição, remoção, procura e alteração de objetos da classe Utente. Em seguida encontra-se o código desta classe divido em partes e a sua respetiva explicação.

```
package Cliente;
import java.io.IOException;
import java.rmi.NotBoundException;
import java.rmi.RMISecurityManager;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.util.GregorianCalendar;
import java.util.Scanner;
import InterfaceAD.Administrador;
import InterfaceAD.IUtente;

public class AdicionaRemoveProcuraAlteraUtente {
        static Registry registry;
        static Administrador ad;

@SuppressWarnings("deprecation")

        public static void main(String args[]) throws RemoteException,
                        NotBoundException {
```

```
            try {
                    registry = LocateRegistry.getRegistry
        ("127.0.0.1", 1098);
                    System.setSecurityManager(new RMISecurityManager());
                    ad = (Administrador)
        registry.lookup("Administradores");
                    menuPrincipalUtente();
            } catch (IOException e) {
                    e.printStackTrace();
            }
    }
```

Esta classe tem um método main que consulta o sistema de registos, onde se encontra registado o objeto remoto Administrador. Caso não seja possível consultar os registos, então é lançada uma exceção com o erro que ocorreu. Seguidamente, a classe inclui um método void menuPrincipalUtente() que segue abaixo:

```
public static void menuPrincipalUtente() throws RemoteException {
            Scanner s = new Scanner(System.in);
            int a;
            do {
                    System.out.println("Escolha uma opcao");
                    System.out.println("1 - Criar utente");
                    System.out.println("2 - Remover utente");
                    System.out.println("3 - Alterar utente");
                    System.out.println("4 - Procurar utente");
                    System.out.println("5 - Listar utentes");
                    System.out.println("6 - Sair");
                    a = s.nextInt();
                    switch (a) {
                    case 1:
                            menuCriarUtente();
                            break;
                    case 2:
                            System.out.println("Insira o codigo do
            utente:");
                            String cod = s.next();
                            System.out.println(ad.remove_utente(cod));
                            System.out.println(
            ad.remove_fichaclinica(cod));
                            break;
                    case 3:
```

```
                        menuAlterarDadosUtente();
                        break;
                case 4:
                        System.out.println("Insira o codigo do
            utente");
                        cod = s.next();
                        System.out.println(ad.procura_utente(cod));
                        break;
                case 5:
                        System.out.println(ad.ListaUtentes());
                        break;
                default:
                        break;
                }
            } while (a != 6);
        }
```

Este menu fornece 6 opções ao utilizador, sendo que 5 dizem respeito a operações relacionadas com utentes e a sexta permite sair do menu. Espera-se, então, um input do teclado do tipo inteiro. É utilizado um ciclo *do (...)while* que efetua um conjunto de operações enquanto o input do teclado for diferente de 6.
Se esse input tiver o valor 1 então é chamado o método menuCriarUtente().
Se tiver o valor 2, então pede-se ao utilizador que insira o código do utente e em seguida é invocado o método remove_utente da interface Administrador. Paralelamente à remoção dessa instância da classe Utente, é também removida uma instância da classe FichaClinica que está associada a esse utente.
Se o valor introduzido for 3 então é invocado o método menuAlterarDadosUtente(). Se for o 4 é pedido ao utilizador que insira uma *string* com o código do utente que se pretende procurar. Utiliza-se, então, o método procura_utente da classe Administrador que recebe o código introduzindo, devolvendo todos os dados desse utente. Por último, se a opção escolhida for a 5, é chamado o método ListaUtentes() da classe administrador que devolve uma listagem de todos os utentes introduzidos no sistema até ao momento.
Abaixo encontra-se o menu de alteração e de criação de utentes referenciados nas opções 1 e 3, respetivamente, completando-se, assim a totalidade do código da classe AdicionaRemoveProcuraAlteraUtente.

```
public static void menuAlterarDadosUtente() throws RemoteException {
            Scanner s = new Scanner(System.in);
            int b;
            do {
                    System.out.println("Insira o codigo do utente:");
                    String cod = s.next();
```

```
                System.out.println("Escolha uma opcao:");
                System.out.println("1 - Alterar nome do utente");
                System.out.println("2 - Alterar BI do utente");
                System.out.println("3 - Alterar morada do utente");
                System.out.println("4 - Alterar data de
        nascimento do utente");
                System.out.println("5 - Alterar contacto do utente");
                System.out.println("6 - Sair");
                b = s.nextInt();
                if (b == 1) {
                        System.out.println("Insira o novo nome");
                        String nome = s.next();
                        String res = ad.alteranome_utente(cod, nome);
                        System.out.println(res);
                }
                if (b == 2) {
                        System.out.println("Insira o novo BI");
                        int bi = s.nextInt();
                        String res = ad.alteraBI_utente(cod, bi);
                        System.out.println(res);
                }
                if (b == 3) {
                        System.out.println("Insira a nova morada");
                        String mor = s.next();
                        String res = ad.alteramorada_utente(cod, mor);
                        System.out.println(res);
                }
                if (b == 4) {
                        System.out.println("Insira a nova data
            de nascimento:");
                        System.out.println("Insira o dia (xx):");
                        int dia = s.nextInt();
                        System.out.println("Insira o mes (xx):");
                        int mes = s.nextInt();
                        System.out.println("Insira o ano (xxxx):");
                        int ano = s.nextInt();
                        GregorianCalendar data = new
            GregorianCalendar(ano, mes, dia);
                        String res = ad.alteradata_utente(cod, data);
                        System.out.println(res);
                }
                if (b == 5) {
```

```
                                System.out.println("Insira o novo contacto");
                                int cont = s.nextInt();
                                String res =
                ad.alteracontato_utente(cod, cont);
                                System.out.println(res);
                        }
                } while (b != 6);
        }

        public static void menuCriarUtente() throws RemoteException {
                Scanner s = new Scanner(System.in);
                System.out.println("Insira o nome do utente");
                String nome = s.next();
                System.out.println("Insira o BI do utente");
                int bi = s.nextInt();
                System.out.println("Insira a morada do utente");
                String mor = s.next();
                System.out.println("Insira o contacto utente");
                int cont = s.nextInt();
                System.out.println("Insira a data de nascimento do
        utente:");
                System.out.println("Insira o dia (xx):");
                int dia = s.nextInt();
                System.out.println("Insira o mes (xx):");
                int mes = s.nextInt();
                System.out.println("Insira o ano (xxxx):");
                int ano = s.nextInt();
                GregorianCalendar data = new GregorianCalendar(ano,
        mes, dia);
                System.out.println(ad.adiciona_utente(mor, bi, cont,
        data, nome));

        }
```

O método void menuAlterarDadosUtente() solicita novamente ao utilizador que seja inserida uma opção do tipo inteiro. É novamente utilizado um ciclo *for (...) while* que executa um conjunto de ciclos *if* enquanto a opção escolhida não for a 6 (que permite retroceder para o menu principal). Dentro dos ciclos *if* recorre-se a métodos definidos na interface Administrador para alteração de dados da classe utente, dado um código de utente fornecido pelo utilizador. O método menuCriarUtente() pede ao utilizador que insira os vários dados do novo utente, utilizando o método adiciona_utente do administrador que recebe um construtor com esses mesmos dados.

2.3.3 Cliente Especial

No sentido de introduzir no sistema os 100000 utentes, 5000 médicos 7000 enfermeiros e 3000 funcionários pedidos no enunciado, criou-se uma classe denominada ClienteEspecial que insere dados aleatórios para as entidades referidas. Abaixo segue a parte inicial do código desta classe.

```
package Cliente;

import InterfaceAD.Administrador;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.util.ArrayList;
import java.util.GregorianCalendar;

public class ClienteEspecial {


public static void main(String args[]) throws RemoteException,
NotBoundException {

        Registry registry = LocateRegistry.getRegistry("127.0.0.1", 1098);

        Administrador ad= (Administrador)registry.lookup("Administradores");
        try {

                final File file_n = new File(
                "/Users/anasofiaquintas/Desktop/nomes.txt");
                final File file_m = new File(
                "/Users/anasofiaquintas/Desktop/morada.txt");
                final File file_a = new File(
                "/Users/anasofiaquintas/Desktop/apelidos.txt");
                final File file_e = new File(
                "/Users/anasofiaquintas/Desktop/especialidades.txt");
                final File file_p = new File(
                "/Users/anasofiaquintas/Desktop/produtos.txt");
                final File file_med = new File(
                /Users/anasofiaquintas/Desktop/medicamentos.txt");
```

```
            BufferedReader reader_nome =
        new BufferedReader(new FileReader(file_n));
            BufferedReader reader_morada =
        new BufferedReader(new FileReader(file_m));
            BufferedReader reader_apelido =
        new BufferedReader(new FileReader(file_a));
            BufferedReader reader_especialidade =
        new BufferedReader(new FileReader(file_e));
            BufferedReader reader_produto =
        new BufferedReader(new FileReader(file_p));
            BufferedReader reader_medicamento =
        new BufferedReader(new FileReader(file_med));

            ArrayList<String> temp_nome = new ArrayList<String>();
            ArrayList<String> temp_apelido = new ArrayList<String>();
            ArrayList<String> temp_morada = new ArrayList<String>();
            ArrayList<String> temp_especialidade = new ArrayList<String>();
            ArrayList<String> temp_produto = new ArrayList<String>();
            ArrayList<String> temp_med = new ArrayList<String>();
            System.out.print(" Criei Arrays");

            String line_n, line_m, line_a, line_e, line_p, line_med;
            int countn = 0, countm = 0, counta = 0, counte = 0,
        countp = 0, countmed = 0;
```

Esta classe tem um método *main* dentro do qual se procura no sistema de registos (criado na classe Servidor) a classe Administrador da InterfaceAD. De seguida inicia-se os blocos *try (...) catch (...)*. No bloco *try* são lidos os dados dos ficheiros de texto onde se encontram guardadas sequências de nomes próprios, moradas, apelidos, especialidades, produtos e medicamentos através de um *BufferedReader*. Essas informações são guardads em *ArrayLists* respetivas. Efetua-se a declaração das variáveis *String* que correspondem a cada linha de cada ficheiro e dos inteiros que contam o número de adicionados. Abaixo encontra-se os ciclos *while* utilizados para percorrer cada linha de cada ficheiro, até que se encontre uma linha vazia. Enquanto isto não acontecer, é adicionado a cada *ArrayList* criado previamente uma linha, incrementando-se o contador respetivo com +1. A partir do momento em que se encontre uma linha vazia, o ciclo é abandonado e a leitura termina, fechando-se o *textit* através do método *close()*.

```
while ((line_n = reader_nome.readLine()) != null) {
                    temp_nome.add(line_n);
                    countn += 1;
            }
```

```
            while ((line_m = reader_morada.readLine()) != null) {
                    temp_morada.add(line_m);
                    countm += 1;
            }
            while ((line_a = reader_apelido.readLine()) != null) {
                    temp_apelido.add(line_a);
                    counta += 1;
            }
            while ((line_e = reader_especialidade.readLine()) !=
        null) {
                    temp_especialidade.add(line_e);
                    counte += 1;
            }
            while ((line_p = reader_produto.readLine()) != null) {
                    temp_produto.add(line_p);
                    countp += 1;
            }
            while ((line_med = reader_medicamento.readLine()) !=
        null) {
                    temp_med.add(line_med);
                    countmed += 1;
            }

            reader_nome.close();
            reader_morada.close();
            reader_apelido.close();
            reader_especialidade.close();
            reader_produto.close();
            reader_medicamento.close();
```

Uma vez terminada a leitura dos ficheiros de texto, resta, então, gerar os dados aleatórios de contacto, número de BI, data de nascimento, através da utilização de funções *random*. Uma vez reunidos todos os dados necessário são utilizados os métodos de adicionar entidades contidos na classe Administrador, para efetivamente, criar as instâncias. Estes métodos recebem o construtor da classe, preenchido com os dados gerados. Abaixo segue a parte restante do código da classe ClienteEspecial.

```
int i, dia, mes, contato, bi, quant;

            int ano = 0;
            String nomep = null, nomea = null, morada = null,
        especialidade = null, produto = null, medicamento = null;
```

```
        quant = (int) (Math.random()*100) + 50;
        nomep = temp_nome.get((int) (Math.random() * countn));
        nomea = temp_apelido.get((int) (Math.random() * counta));
        morada = temp_morada.get((int) (Math.random() *
countm));
        especialidade = temp_especialidade.get((int)
(Math.random()* counte));


        //Adicionar 100,000 utentes
        for (i = 0; i < 100000; i++) {
                dia = (int) (Math.ceil(Math.random()*31));

                mes = (int) (Math.ceil(Math.random()*11));

                ano = (int) (Math.random()*1974) + 40;

                bi = (int) (Math.random()*90000000)+10000000;

                contato = (int) (Math.random()*100000000) + 900000000;



                String nomeu = (nomep + " " + nomea);

                GregorianCalendar date = new GregorianCalendar(ano,
    mes, dia);

                ad.adiciona_utente(morada, bi, contato, date, nomeu);

        }

        //Adicionar 5000 medicos
        for (i = 1; i <= 5000; i++) {

                nomep = temp_nome.get((int) (Math.random() * countn));
                nomea = temp_apelido.get((int) (Math.random() *
    counta));
                String nomem = (nomep + " " + nomea);
                dia = (int) (Math.ceil(Math.random()*31));
```

```
                mes = (int) (Math.ceil(Math.random()*12));

                ano = (int) (Math.random()*1974) + 40;

                bi = (int) (Math.random()*90000000)+10000000;

                contato = (int) (Math.random()*100000000) + 900000000;

                GregorianCalendar date = new GregorianCalendar(ano,
mes, dia);

                ad.adiciona_medico(bi, morada, contato, date,
nomem, especialidade);

        }

        //Adicionar 7000 enfermeiros
        for (i = 1; i <= 7000; i++) {
                nomep = temp_nome.get((int) (Math.random() * countn));
                nomea = temp_apelido.get((int) (Math.random() *
counta));
                String nomee = (nomep + " " + nomea);
                dia = (int) (Math.ceil(Math.random()*31));
                mes = (int) (Math.ceil(Math.random()*12));

                ano = (int) (Math.random()*1974) + 40;

                bi = (int) (Math.random()*90000000)+10000000;

                contato = (int) (Math.random()*100000000) + 900000000;

                GregorianCalendar date = new GregorianCalendar(ano,
mes, dia);

                ad.adiciona_enf(bi, morada, contato, date, nomee);

        }

        //Adicionar 3000 funcionarios
        for (i = 1; i <= 3000; i++) {
                nomep = temp_nome.get((int) (Math.random() * countn));
                nomea = temp_apelido.get((int) (Math.random() *
```

```
counta));
                String nomef = (nomep + " " + nomea);
                dia = (int) (Math.ceil(Math.random()*31));
                mes = (int) (Math.ceil(Math.random()*12));

                ano = (int) (Math.random()*1974) + 40;

                bi = (int) (Math.random()*90000000)+10000000;

                contato = (int) (Math.random()*100000000) + 900000000;


                GregorianCalendar date = new GregorianCalendar(ano,
mes, dia);

                ad.adiciona_func(bi, morada, contato, date, nomef);

        }

        //Adiciona 43 produtos

        for (i = 1; i <= 43; i++) {
                nomep = temp_produto.get((int) (Math.random() *
countp));

                quant = (int) (Math.random()*100) + 50;

                ad.adiciona_produto(produto, quant);

        }

        //Adiciona 43 medicamentos
        for (i = 1; i <= 43; i++) {
                quant = (int) (Math.random()*100) + 50;
                medicamento = temp_med.get((int) (Math.random() *
countmed));


                ad.adiciona_medica(medicamento, quant);

        }
```

```
                System.out.println("Dados gerados!");
} catch (Exception e) {
        System.err.println(e.getMessage());
                }
        }
}
```

2.4 Política de segurança e compilação

2.4.1 A política de segurança

Anexado ao projeto Servidor e Cliente está um ficheiro denominado *security.policy* que controla quais os clientes que têm acesso a quais privilégios. No caso da aplicação hospital que se construiu todos os clientes têm permissões globais.
O *security manager* implementado na classe Servidor efetua o controlo da *security.policy* na aplicação. Ele determina se determinadas operações deve ser de livre acesso ou se devem estar bloqueadas.

```
grant {
        permission java.security.AllPermission;
  };
```

3 Conclusão

A elaboração deste trabalho prático revelou-se um desafio interessante para o grupo, dadas as dificuldades que foram surgindo durante o seu progresso. Ainda assim, o grupo conclui ter sido uma experiência muito positiva, já que permitiu desenvolver capacidades ao nível da programação em Java e aplicações Cliente/Servidor, utilizando o RMI (que era antes um ferramenta por explorar).

Face aos requisitos especificados no enunciado do trabalho, conclui-se que estes foram cumpridos na sua totalidade: efetuou-se a definição das entidades e relações entre elas; definição das interfaces; implementação do JAVA RMI nos programas servidor e cliente; implementação de persistência de dados, isto é, os dados são guardados em disco durante a execução do servidor; são recuperados caso o programa seja reiniciado e existe um volume significativo de dados na aplicação. No entanto, dos 100,000 utentes pedidos no enunciado apenas foram gerados pelo *ClienteEspecial* cerca de 99,950 uma vez que a função que gera aleatoriamente os números de bilhete de identidade por vezes repete um número que já constava da base de dados. A função *verifica_bi* não permite que o método *cria_utente* gere um novo utente caso o número de bi já se encontre nos dados. Consequentemente são criados menos que 100,000 utentes.

Ainda assim, reconhece-se que algumas alterações poderiam ter sido implementadas no que diz respeito à estrutura da aplicação. Por exemplo, a entidade funcionários deveria especificar funções para duas categorias distintas: os funcionários rececionistas, que seriam responsáveis pela marcação de consultas e alteração de dados pessoais; os funcionários da farmácia que deveriam gerir as encomendas e os stocks de produtos de enfermagem e medicamentos bem como a receção dos mesmos.

Por outro lado, poder-se-ia ter criado um sistema de login que permitisse identificar o utilizador como médico, enfermeiro ou funcionário, disponibilizando-lhe, assim, apenas os menus que dissessem respeito a cada entidade.

Poderia-se, ainda, ter implementado mais restrições a nível de código que condicionassem a informação inserida pelo utilizador na aplicação, de modo a aumentar a coerência do programa desenvolvido e torná-lo mais adaptado à realidade.

Não se procedeu às alterações descritas dada a sua minuciosidade aliada à altura em que o grupo se apercebeu desta possibilidade, momento em que o programa já se encontrava completo e a data de entrega se encontrava próxima.

Referências

[1] W. Emmerich, *Engineering Distributed Objects.* Wiley.

4 Anexos

4.1 Anexo I

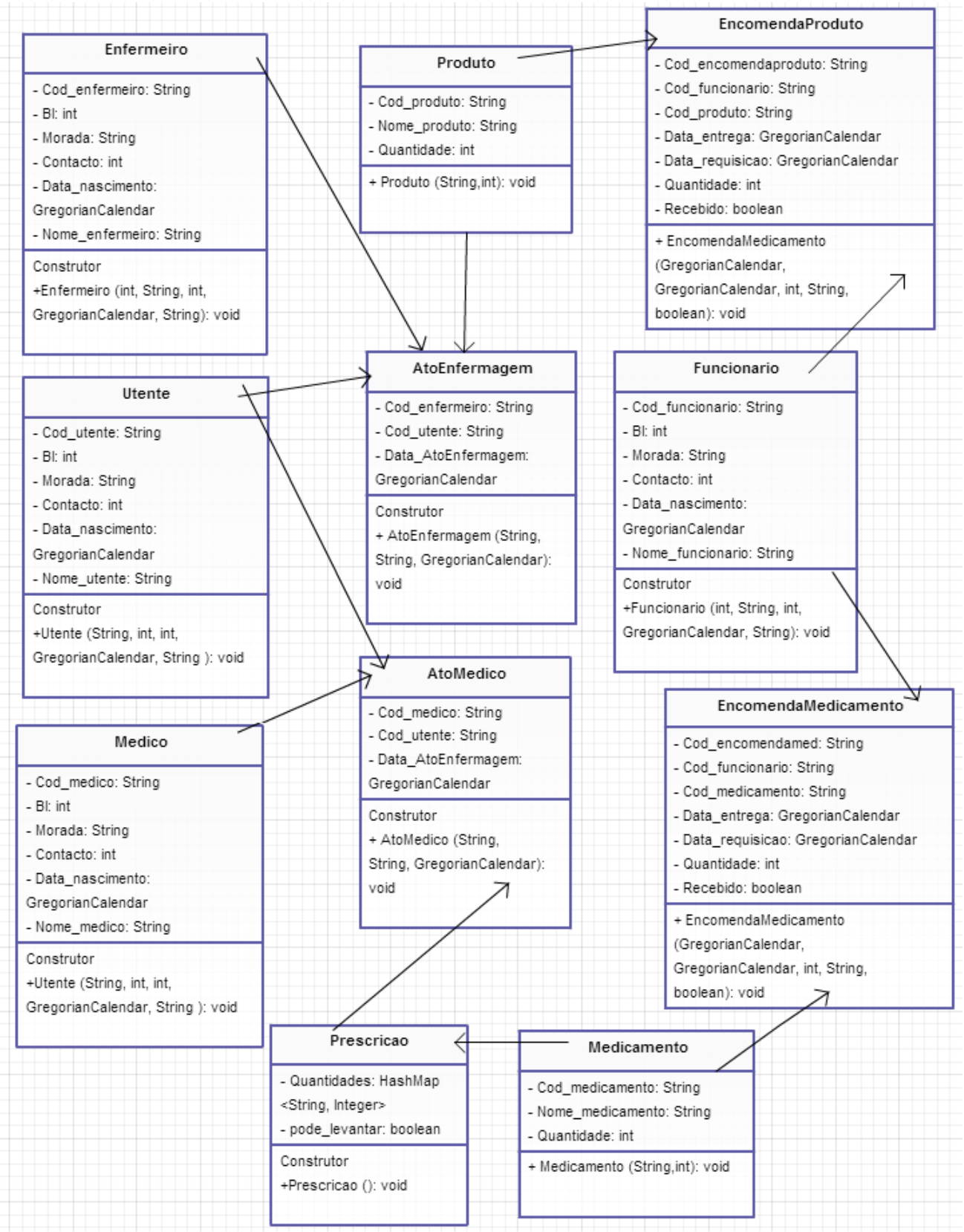

Figura 3: Diagrama de classes que relaciona as entidades do hospital

4.2 Anexo II

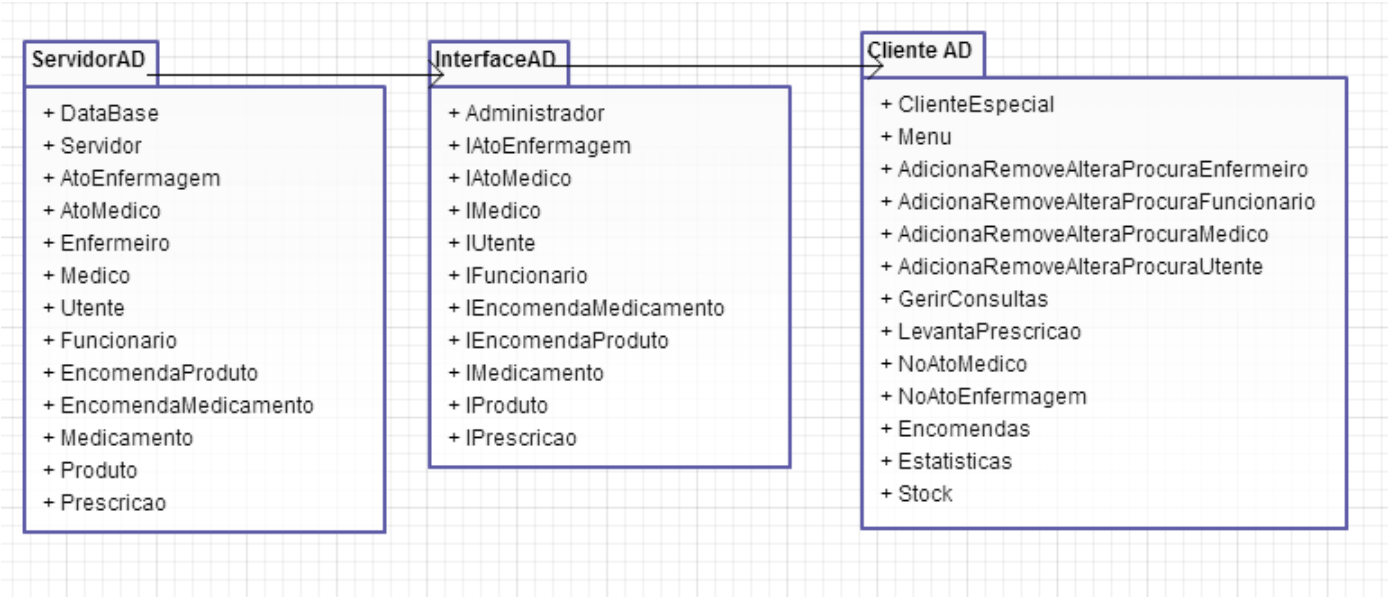

Figura 4: Diagrama de packages e classes que cada um contém

4.3 Anexo III

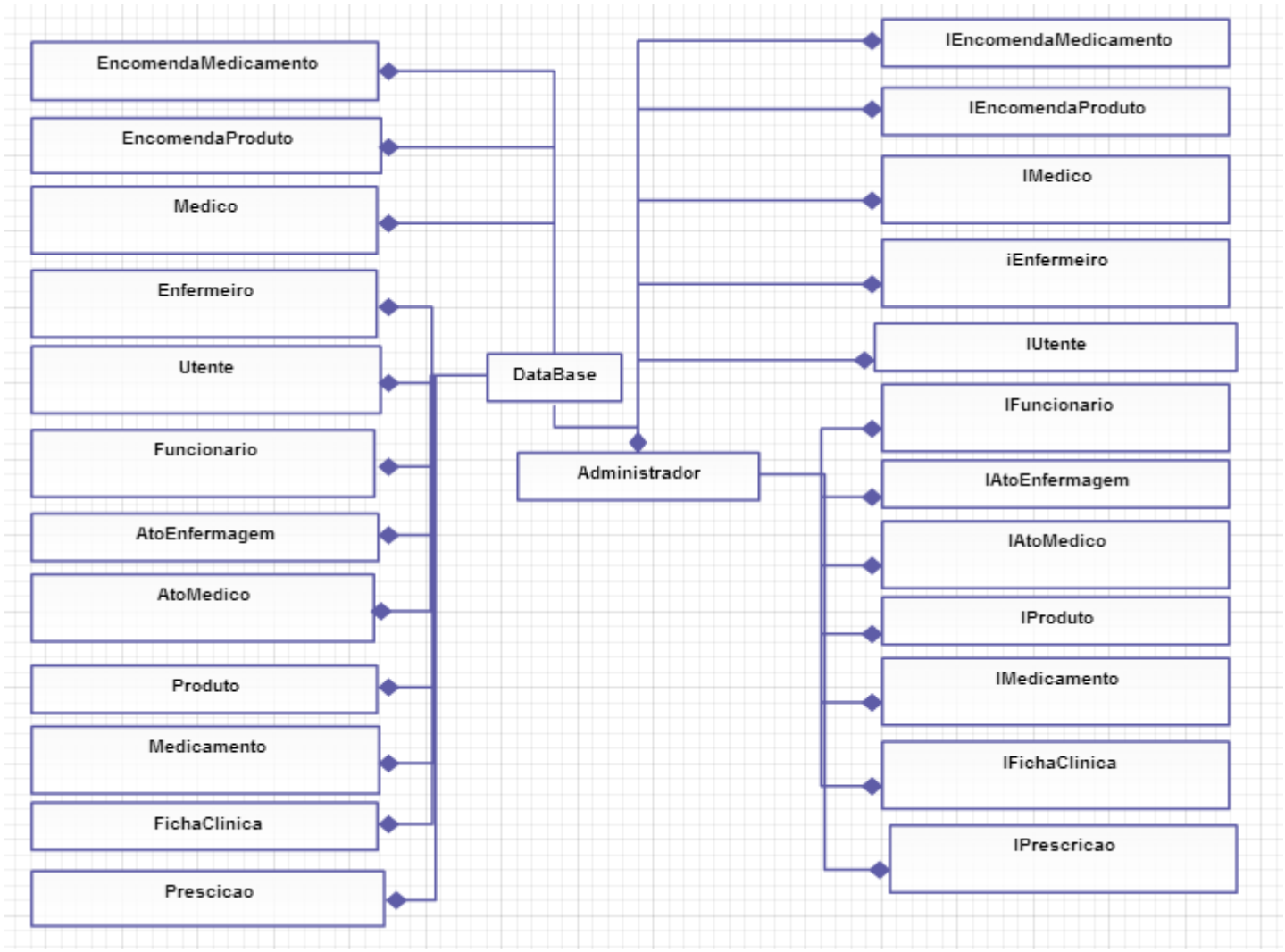

Figura 5: Diagrama de classes que relaciona as entidades com a DataBase e o admnistrador com as intefaces