Mestrado em Informática Médica

Aplicações Distribuídas

CuidaUM - Registo de dados e cuidados de saúde

Realizado por:
Beatriz Rodrigues, PG53696
Bruno Machado PG53709
Rúben Ganança, PG54203

Docente: António Luís Pinto Ferreira Sousa

21 de novembro de 2023

Índice

1 Introdução **3**

2 Preliminares **4**

2.1 Classes 4

2.2 Interfaces 4

2.3 Exceções 4

2.4 Java RMI 5

2.5 Arquitetura Cliente-Servidor 5

3 Desenvolvimento do Sistema de Registo de Dados **6**

3.1 Classes das Entidades 7

3.1.1 Classe *Utilizador* 7

3.1.2 Classe *Utente* 8

3.1.3 Classe *ProfSaude* 9

3.1.4 Classe *Familiar* 10

3.1.5 Classe *ParamMedico* 10

3.1.6 Classe *Consulta* 11

3.1.7 Classe *Prescricao* 11

3.1.8 Classe *Exame* 12

3.1.9 Classe *Gestor* 13

3.2 Métodos do Gestor 14

3.2.1 Métodos *criarUtente*, *criarProfSaude*, *criarFamiliar* 14

3.2.2 Métodos *criarConsulta*, *criarPrescricao* e *criarExame* . . . 15

3.2.3 Método *criarParametro* 16

3.2.4 Métodos *listarExames*, *listarConsultas*, *listarParametros* e *listarPrescricoes* 16

3.2.5 Métodos *carregarFicheiros* e *guardarFicheiros* 17

3.2.6 Método *gerarCSV* 18

3.2.7 Métodos Auxiliares 19

3.3 Interface - *GestorInterface* 20
3.4 Exceções 21
3.5 *GestorServidor* 21
3.6 *GestorCliente* 22
3.7 Funcionalidades Extra - Estatística, Palavra-Passe, Notificações . 23
3.7.1 Estatística - *numeroConsultas* 23
3.7.2 Palavra-Passe 24
3.7.3 Notificações - Caixa de mensagens 25

4 Conclusão **27**

Referências **28**

1. Introdução

No âmbito da Unidade Curricular de Aplicações Distribuídas realizou-se o seguinte trabalho cujo principal objetivo consiste na implementação de um sistema de registo de dados e cuidados de saúde (CuidaUM). Na sua realização utilizou-se uma arquitetura Cliente - Servidor e *Java RMI*.

Neste projeto pretendeu-se desenvolver uma aplicação ou conjunto de aplicações que permitissem a gestão e registo dos cuidados de saúde e dados fisiológicos de uma unidade de cuidados de saúde.

Assim, os principais requisitos consistiram em:

- Definir as Interfaces a desenvolver e as aplicações associadas;
- Definir e implementar as entidades envolvidas;
- Implementar o processo de registo de utilizadores;
- Implementar o do processo de registo de atualizações de dados;
- Implementar o processo de consulta de dados.

De modo a cumprir estes requisitos, foram estabelecidas as **Entidades** e **Interfaces** fundamentais para o projeto, tendo em consideração a lógica de gestão concebida para garantir o funcionamento adequado da aplicação.

2. Preliminares

2.1 Classes

De forma a indicar os atributos identificadores de cada entidade e efetuar o registo dos dados utilizados, é necessário implementar diversas classes. Cada classe corresponde a uma entidade pertencente ao sistema e contém todas as informações inerentes a ela, permitindo assim, que estas informações sejam manipuladas através dos métodos criados.

2.2 Interfaces

Todos os objetos remotos possuem uma Interface que especifica quais dos métodos podem ser invocados remotamente. Sendo assim, uma Interface é uma classe abstrata que é usada para agrupar métodos relacionados com corpos vazios [1].

Para este projeto foi criada a interface *GestorInterface* que dá base ao *Gestor*. Nesta interface estão definidos métodos cujos corpos estão definidos na classe referida anteriormente.

2.3 Exceções

Durante a experimentação, implementação e execução do código, podem ocorrer diversos erros, nomeadamente, erros de codificação pelo programador, erros devido a *input* incorreto, ou mesmo erros derivados da distribuição e/ou execução do método. Quando ocorre um erro, por norma, o programa termina e gera uma mensagem de erro, isto é, uma exceção [2]. Assim, foram criadas as seguintes exceções: *FamiliarJaExiste, ProfSaudeJaExiste, UtenteJaExiste*.

2.4 Java RMI

Para a implementação deste projeto utilizou-se a tecnologia *RMI* (*Remote Method Invocation*), suportada pelo JAVA, que facilita o desenvolvimento de aplicações distribuídas baseadas em objetos. Neste sentido, o *RMI* tem como principal objetivo permitir a invocação de métodos de objetos remotos, ou seja, permite simplificar a comunicação entre dois objetos em máquinas virtuais distintas, indo ao encontro do conceito de aplicações distribuídas. Isto possibilita a troca de classes e objetos com diferentes complexidades entre clientes e servidores. [3] [4]

Desta forma, o *RMI* permite que uma determinada aplicação **Cliente** adquira uma **Interface** que irá definir o seu comportamento tendo como referência uma determinada Classe. Esta Classe contém o processo de implementação, e pode existir numa máquina virtual distinta. Assim, uma aplicação *RMI* é frequentemente composta por dois programas diferentes, um **servidor**, que cria objetos remotos e faz referência a esses objetos disponíveis, e um **cliente**, que invoca os métodos sobre os objetos. [3] [4]

2.5 Arquitetura Cliente-Servidor

A arquitetura cliente-servidor é uma arquitetura de aplicação distribuída, ou seja, na rede existem os fornecedores de recursos ou serviços, que são chamados de servidores, e existem os requerentes dos recursos ou serviços, denominados clientes. Então, o cliente é responsável por iniciar a comunicação e o servidor por aguardar restricões de entrada. [5]

3. Desenvolvimento do Sistema de Registo de Dados

Deste modo e de maneira a cumprir os objetivos supramencionados foram definidas as entidades bem como a interface envolvida no projeto, tendo em conta a lógica de gestão idealizada para o correto funcionamento do programa. Assim sendo, projetou-se uma arquitetura Servidor-Cliente composta pelos seguintes elementos: um servidor, um cliente que pode aceder ao sistema (Gestor) e consequentemente uma Interface que permite a ligação entre o cliente e o servidor.

Na elaboração deste projeto, utilizou-se o *UML - Unified Modeling Language* para representar a arquitetura e a estrutura do sistema proposto. Tanto as Classes como as suas relações são expressas através de diagramas de Classes, que permitem uma visão mais abrangente das entidades bem como, das suas interações no sistema. Ao adotar o *UML*, permitiu-se obter uma compreensão visual clara da arquitura, da estruturação e da lógica subjacentes. Na figura 3.1, abaixo representada, encontra o *UML* representativo da aplicação.

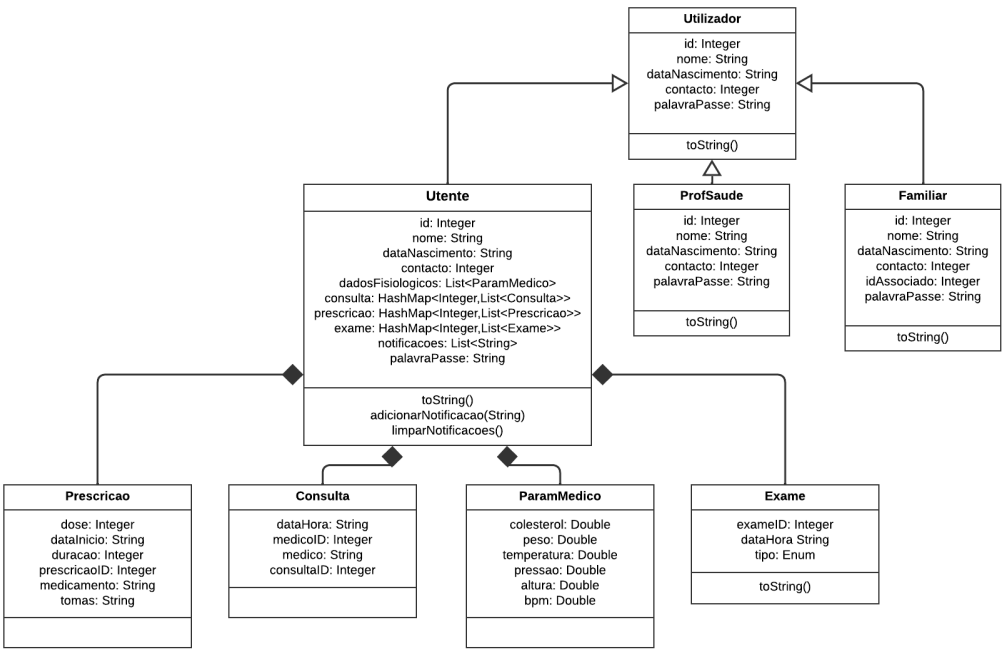

Figura 3.1: Diagrama *UML*

3.1 Classes das Entidades

Relativamente ao projeto desenvolvido foi necessária a elaboração de um conjunto de entidades para o registo de uma panóplia de informações referentes aos vários envolvidos do sistema.

O desenvolvimento destas entidades mostrou-se crucial para que fosse possível manipular e caracterizar as informações inerentes a cada uma delas nos diferentes métodos presentes no sistema. De seguida, todas as entidades serão apresentadas em detalhe.

3.1.1 Classe *Utilizador*

A entidade *Utilizador*, representada na Figura 3.2, é estruturada numa superclasse, cujos atributo são posteriormente herdados pelas entidades filho, nomeadamente, as entidades *Utente, ProfSaude* e *Familiar*. A criação desta superclasse teve como principais objetivos obter uma maior simplificação e eficiência no código, visto que, caso esta não existisse, seria necessário descrever todos os seus atributos nas três entidades que têm esses atributos transversalmente. Além disso, definiram-se todos os *sets* e *gets* para cada atributo, bem como o construtor da classe.

Assim, os atributos para esta entidade são:

- `id`: identificador único do utilizado que, de forma a simplificar, se trata do número de saúde do mesmo;
- `nome`: nome do utilizador;
- `dataNascimento`: data de nascimento do utilizador;
- `contacto`: contacto telefónico do utilizador;
- `palavraPasse`: palavra passe do utilizador, assegurando a privacidade dos dados inseridos.

```java
package Entidades;

import java.io.Serializable;

7 inheritors
public class Utilizador implements Serializable {

    4 usages
    private int id;

    4 usages
    private String nome;

    4 usages
    private String dataNascimento;

    4 usages
    private int contacto;

    3 usages
    private String palavraPasse;

    public Utilizador(int id, String nome, String dataNascimento, int contacto, String palavraPasse) {
        this.id = id;
        this.nome = nome;
        this.dataNascimento = dataNascimento;
        this.contacto = contacto;
        this.palavraPasse = palavraPasse;
    }
```

Figura 3.2: Superclasse *Utilizador*

3.1.2 Classe *Utente*

A classe *Utente*, presente na Figura 3.3 é uma subclasse da superclasse *Utilizador*. Desta forma, herda os mesmos atributos desta superclasse através do *extends* da mesma, ou seja, tem como atributos os referidos no tópicos anterior.

Além disso, é composta por outros atributos, nomeadamente:

- `dadosFisiológicos`: parâmetros médicos do utente medidos nas consultas;
- `notificacao`: guarda as mensagens para o utente de marcações de consultas, exames e prescrições;
- `consulta`: agendamento das consultas do utente;
- `prescricao`: descrição das prescrições dos medicamentos tomados pelo utente;
- `exames`: informações acerca dos exames realizados ao utente.

```java
public class Utente extends Utilizador implements Serializable {

    1 usage
    private ArrayList<ParamMedico> dadosFisiologicos;

    4 usages
    private ArrayList<String> notificacao;

    1 usage
    private Map<Integer, List<Consulta>> consulta;

    1 usage
    private Map<Integer, List<Prescricao>> prescricao;

    1 usage
    private Map<Integer, List<Exame>> exames;

    5 usages
    public Utente (int id, String nome, String dataNascimento, int contacto, String palavraPasse) {
        super( id , nome , dataNascimento , contacto, palavraPasse);
        this.dadosFisiologicos = new ArrayList<>();
        this.notificacao = new ArrayList<>();
        this.consulta = new HashMap<>();
        this.prescricao = new HashMap<>();
        this.exames = new HashMap<>();
    }
```

Figura 3.3: Classe *Utente*

3.1.3 Classe *ProfSaude*

A classe *ProfSaude* é também uma subclasse da superclasse *Utilizador* herdando, portanto, os seus atributos. Os atributos desta classe são apenas os herdados pela superclasse. Esta classe encontra-se na figura 3.4.

```java
package Entidades;

import java.io.Serializable;

6 usages
public class ProfSaude extends Utilizador implements Serializable {

    2 usages
    public ProfSaude (int id, String nome, String dataNascimento, int contacto, String palavraPasse) { super( id , nome , dataNascimento , contacto, palavraPasse ); }

    public String toString() {

        return "Profissional de saúde { "
                + super.getId() + ", "
                + super.getNome() +", "
                + super.getDataNascimento() + ", "
                + super.getContacto() + " }";
    }

    public String getPalavraPasse(){return super.getPalavraPasse();}

}
```

Figura 3.4: Classe *ProfSaude*

3.1.4 Classe *Familiar*

Tal como as classes anteriores, a classe *Familiar* também herda os atributos da superclasse, tendo apenas mais um atributo que se trata do `idAssociado` referente ao id do utente associado a cada familiar.

```
package Entidades;

import java.io.Serializable;

6 usages
public class Familiar extends Utilizador implements Serializable {

    3 usages
    private int idAssociado;
    1 usage
    public Familiar(int id, String nome, String dataNascimento, int contacto, int idAssociado,String palavraPasse) {
        super(id, nome, dataNascimento, contacto, palavraPasse);
        this.idAssociado = idAssociado;
    }

    public String toString() {
```

Figura 3.5: Classe *Familiar*

3.1.5 Classe *ParamMedico*

Esta classe herda os atributos da classe *Utente* e tem ainda os seus próprios atributos de maneira a guardar os dados físicos do utente. Desta maneira os novos atributos foram o `peso`, `colesterol`, `bpm`, `pressao`, `altura` e `temperatura`.

```
public class ParamMedico extends Utente implements Serializable {

    3 usages
    private double peso;

    3 usages
    private double colesterol;

    3 usages
    private double bpm;

    3 usages
    private double pressao;

    3 usages
    private double altura;

    3 usages
    private double temperatura;

    2 usages
    public ParamMedico(int id, String nome, String dataNascimento, int contacto,String palavraPasse){
        super(id,nome,dataNascimento,contacto, palavraPasse);
        this.peso= 0;
        this.colesterol = 0;
        this.bpm = 0;
        this.pressao = 0;
        this.altura = 0;
        this.temperatura = 0;
    }
```

Figura 3.6: Classe *ParamMedico*

3.1.6 Classe *Consulta*

Tal como a classe anterior esta herda os atributos da classe *Utente* e tem ainda os seus próprios atributos como a `consultaID`, `medicoID`, `medico` e `dataHora`. Salienta-se que o `consultaID` é um número único para as consultas de todos os pacientes do hospital, isto é, a unicidade deste atributo, não é apenas para o contexto individual de cada utente, mas sim de forma global. Esta classe tem como objetivo guardar as informações acerca de uma consulta.

```
package Entidades.UtenteInfo;
import ...

16 usages
public class Consulta extends Utente implements Serializable {

    3 usages
    private int consultaID;

    3 usages
    private int medicoID;

    3 usages
    private String medico;

    3 usages
    private String dataHora;

    2 usages
    public Consulta(int id, String nome, String dataNascimento, int contacto, String palavraPasse){
        super(id,nome,dataNascimento,contacto,palavraPasse);
        this.consultaID = 0;
        this.medicoID = 0;
        this.medico = null;
        this.dataHora = null;

    }
```

Figura 3.7: Classe *Consulta*

3.1.7 Classe *Prescricao*

Esta classe tem como fim guardar os dados de uma prescrição médica. Esta classe herda, à semelhança da anterior, os atributos da *Utente* e tem outros atributos para a definir tal como `prescricaoID`, `medicamento`, `dose`, `duracao`, `dataInicio`, `tomas`. Analogamente à classe anterior, a variável `prescricaoID` é também única em todo o universo de registos hospitalar.

```java
public class Prescricao extends Utente implements Serializable {

    3 usages
    private int prescricaoID;

    3 usages
    private String medicamento;

    3 usages
    private int dose;

    3 usages
    private int duracao;

    3 usages
    private String dataInicio;

    3 usages
    private String tomas;

    1 usage
    public Prescricao(int id, String nome, String dataNascimento, int contacto, String palavraPasse){
        super(id,nome,dataNascimento,contacto, palavraPasse);
        this.prescricaoID = 0;
        this.medicamento = null;
        this.dose = 0;
        this.duracao = 0;
        this.tomas = null;
        this.dataInicio = null;
    }
```

Figura 3.8: Classe *Prescricao*

3.1.8 Classe *Exame*

A classe *Exame* tem como função guardar os dados obtidos após a realização de um determinado exame. Deste modo, herdará também as informações da classe *Utente*, tendo novos atributos como o `exameID` (segue o mesmo raciocinio do `id` da classe anterior), `dataHora` da realização do exame e o `tipoExame` que é um *enum* para que os tipos de exame sejam apenas os enumerados.

```java
package Entidades.UtenteInfo;

import ...

20 usages
public class Exame extends Utente implements Serializable {

    4 usages
    private int exameID;

    3 usages
    private String dataHora;

    10 usages
    public enum tipoExame {
        no usages
        ECG, RaioX, Hemograma, Tomografia, Ressonância, Ecografia, Outros
    }

    4 usages
    private tipoExame tipo;

    // Construtor da classe
    4 usages
    public Exame(int id, String nome, String dataNascimento, int contacto, String palavraPasse) {
        super(id,nome,dataNascimento,contacto, palavraPasse);
        this.exameID = 0;
        this.tipo = null;
        this.dataHora = null;
    }
```

Figura 3.9: Classe *Exame*

3.1.9 Classe *Gestor*

Nesta classe, visível na figura 3.10, foram definidos todos os métodos necessários ao desenvolvimento da aplicação cliente-servidor, ou seja, é nesta classe que estão definidos os métodos que permitem a construção e manipulação das entidades supramente descritas.

De forma a que fosse mais fácil o manuseamento das classes, foram criados *HashMap* e *ArrayList* com todas as informações necessárias ao funcionamento da aplicação.

Assim sendo, esta classe tem os atributos:

- `utente`: é um *HashMap* que tem como chave o nº de saúde do utente e como valor as suas informações;
- `medico`: é um *HashMap* que tem como chave o nº de saúde do médico e como valor as suas informações;
- `familiar`: é um *HashMap* que tem como chave o nº de saúde do familiar e como valor as suas informações;
- `consulta`: é um *HashMap* que tem como chave o nº de saúde do utente e como valor a lista de consultas desse utente;
- `prescricao`: é um *HashMap* que tem como chave o nº de saúde do utente e como valor a lista de prescrições desse utente;
- `exame`: é um *HashMap* que tem como chave o nº de saúde do utente e como valor a lista de exames desse utente;
- `parametro`: é um *HashMap* que tem como chave o nº de saúde do utente e como valor os últimos parâmetros médicos associados a esse utente.

```
public class Gestor implements Serializable, GestorInterface {

    19 usages
    private Map<Integer, Utente> utente;
    10 usages
    private Map<Integer, ProfSaude> medico;
    10 usages
    private Map<Integer, Familiar> familiar;
    12 usages
    private Map<Integer, List<Consulta>> consulta;
    10 usages
    private Map<Integer, List<Prescricao>> prescricao;
    10 usages
    private Map<Integer, List<Exame>> exame;
    9 usages
    private Map<Integer, ParamMedico> parametro;
```

Figura 3.10: Classe *Gestor*

3.2 Métodos do Gestor

3.2.1 Métodos *criarUtente*, *criarProfSaude*, *criarFamiliar*

Estes métodos têm como objetivo a criação de novos utentes, profissionais de saúde e de familiares, que vão estar associados aos utentes. Assim sendo, estes métodos recebem como parâmetro os dados para a criação destas entidades e guardam-nas nos respetivos *HashMaps*. A verificação para utentes, profissionais e familiares repetidos é feita posteriormente.

Por fim, ao terminar cada um dos métodos é atualizado um *csv* com as informações gerais de todos os utilizadores. Estes métodos encontram-se representados na figura 3.11.

```
public void criarUtente(int id, String nome, String dataNascimento, int contacto, String palavraPasse){
    Utente utentes = new Utente(id,nome,dataNascimento,contacto, palavraPasse);
    this.utente.put(id,utentes);
    gerarCSV();
}

1 usage
public void criarProfSaude(int id, String nome, String dataNascimento, int contacto, String palavraPasse){
    ProfSaude medicos = new ProfSaude(id,nome,dataNascimento,contacto, palavraPasse);
    this.medico.put(id,medicos);
    gerarCSV();
}

1 usage
public void criarFamiliar(int id, String nome, String dataNascimento, int contacto, int idAssociado, String palavraPasse){
    Familiar familiares = new Familiar(id,nome,dataNascimento,contacto,idAssociado, palavraPasse);
    this.familiar.put(id,familiares);
    gerarCSV();
}
```

Figura 3.11: Métodos para a criação de novos utentes, médicos e familiares.

3.2.2 Métodos *criarConsulta*, *criarPrescricao* e *criarExame*

Estes métodos têm a finalidade de criar novas consultas, prescrições e exames acrescentando estas novas entradas à lista que se encontra nos valores dos respetivos *HashMaps*. Desta forma é garantido que nenhum destes valores é perdido ou substituido, tendo o utente acesso a todo o histórico de ações que teve. Na figura 3.12 encontra-se um destes métodos para servir de exemplo.

```
public void criarConsulta(int utenteID,int medicoID, String medico, String dataHora){

    if (!consulta.containsKey(utenteID)) {
        consulta.put(utenteID, new ArrayList<>());
    }

    int consultaID = gerarIdConsulta()+1;
    Utente utenteAtual = utente.get(utenteID);
    String nomeUtente = utenteAtual.getNome();
    String dataNascimentoUtente = utenteAtual.getDataNascimento();
    int contactoUtente = utenteAtual.getContacto();
    String passeUtente = utenteAtual.getPalavraPasse();

    Consulta consultas = new Consulta(utenteID,nomeUtente,dataNascimentoUtente,contactoUtente,passeUtente);
    consultas.setConsultaID(consultaID);
    consultas.setMedicoID(medicoID);
    consultas.setMedico(medico);
    consultas.setDataHora(dataHora);

    this.consulta.get(utenteID).add(consultas);
}
```

Figura 3.12: Método para a criar uma consulta.

3.2.3 Método *criarParametro*

Neste método, ao contrário do anterior, não se pretende criar uma lista para armazenar todos os parâmetros como histórico. Assim, na primeira medição, os parâmetros são criados pela primeira vez, e, caso contrário, os dados são atualizados com os últimos parâmetros medidos.

Assim sendo, foi necessário recorrer aos métodos *sets* definidos na classe *ParamMedico*. Este método encontra-se na figura 3.13.

```
public void criarParametro(int utenteID,double peso, double colesterol, double bpm, double pressao, double altura, double temperatura){

    // Utente já existe e vai substituir pelos novos valores
    if (parametro.containsKey(utenteID)) {
        ParamMedico parametros = parametro.get(utenteID);
        parametros.setPeso(peso);
        parametros.setColesterol(colesterol);
        parametros.setBpm(bpm);
        parametros.setPressao(pressao);
        parametros.setAltura(altura);
        parametros.setTemperatura(temperatura);
    }

    Utente utenteAtual = utente.get(utenteID);
    String nomeUtente = utenteAtual.getNome();
    String dataNascimentoUtente = utenteAtual.getDataNascimento();
    int contactoUtente = utenteAtual.getContacto();
    String passeUtente = utenteAtual.getPalavraPasse();

    ParamMedico parametros = new ParamMedico(utenteID,nomeUtente,dataNascimentoUtente,contactoUtente,passeUtente);
    parametros.setPeso(peso);
    parametros.setColesterol(colesterol);
    parametros.setBpm(bpm);
    parametros.setPressao(pressao);
    parametros.setAltura(altura);
    parametros.setTemperatura(temperatura);
    this.parametro.put(utenteID,parametros);
}
```

Figura 3.13: Método para a criar/atualizar os parâmetros médicos.

3.2.4 Métodos *listarExames*, *listarConsultas*, *listarParametros* e *listarPrescricoes*

Os métodos `listar` foram criados com o objetivo de otimizar a consulta de dados pelos diferentes tipos de utilizador. Estes utilizam o `UtenteID` para encontrar as informações pretendidas nas diferentes estruturas de dados, isto é, nos *HashMaps* e *Arraylist*. Assim, nas diferentes estruturas a *key* correspondente ao *UtenteID* é procurada, sendo retornado todo o histórico do objeto em questão. Salienta-se que o `listarParametros`, não contêm histórico, sendo estes atributos atualizados consoante uma nova consulta.

Dado que os métodos referidos comportam-se de forma semelhante, de seguida é apresentado na Figura 3.14 apenas um exemplo dos métodos abordados.

```java
public void listarExames(int utenteID) {
    StringBuilder informacoes = new StringBuilder();
    informacoes.append("\nInformações dos exames do utente: \n");

    if (!exame.containsKey(utenteID) || exame.get(utenteID) == null) {
        System.out.println("\nAinda não tem exames marcados.\n");
        return; // Sair do método, pois não há exames para listar
    }

    List<Exame> listaExames = exame.get(utenteID);

    for (Exame exame : listaExames) {
        informacoes.append("ID do Exame: ").append(exame.getExameID()).append("\n");
        informacoes.append("Tipo de Exame: ").append(exame.getTipo()).append("\n");
        informacoes.append("Data e hora do Exame: ").append(exame.getDataHora()).append("\n");
    }

    System.out.println(informacoes.toString());
}
```

Figura 3.14: Método para a listar o histórico de exames.

3.2.5 Métodos *carregarFicheiros* e *guardarFicheiros*

De forma a ter uma base de dados sempre funcional e carregada foram criados métodos para guardar as informações de todo o tipo de dados, carregando estes dados quando a aplicação é iniciada e guardando-os quando ela encerra.

Assim sendo, foram usados ficheiros do tipo *.ser* de forma melhor armazenar objetos serializados, estando estes ficheiros guardados como fluxos de bytes [6] .

Na figura 3.15 encontra-se um exemplo de métodos usados para carregar e guardar estes ficheiros, sendo que este raciocínio também foi aplicado para o armazenamento de consultas, exames, familiares, médicos, parametros, prescrições, representado na figura 3.16.

```java
public void guardarUtentes() {
    try (ObjectOutputStream outputStream = new ObjectOutputStream(new FileOutputStream( name: "utentes.ser"))) {
        outputStream.writeObject(utente);
    } catch (IOException e) {
        e.printStackTrace();
    }
}

1 usage
public void carregarUtentes() {
    try (ObjectInputStream inputStream = new ObjectInputStream(new FileInputStream( name: "utentes.ser"))) {
        utente = (Map<Integer, Utente>) inputStream.readObject();
    } catch (IOException | ClassNotFoundException e) {}
}
```

Figura 3.15: Método para a carregar e guardar os utentes.

```
public void carregarFicheiros(){
    carregarUtentes();
    carregarProfSaude();
    carregarFamiliar();
    carregarExames();
    carregarConsultas();
    carregarParametros();
    carregarPrescricoes();
}

1 usage
public void guardarFicheiros(){
    guardarUtentes();
    guardarProfSaude();
    guardarFamiliar();
    guardarExames();
    guardarConsultas();
    guardarParametros();
    guardarPrescricoes();
}
```

Figura 3.16: Método para a carregar e guardar todas as estruturas de dados.

3.2.6 Método *gerarCSV*

Este método tem a finalidade de gerar um ficheiro *csv* com as informações dos utentes, profissionais de saúde e familiares à exceção das palavras-passe destes utilizadores.

Este ficheiro apenas é acessível ao gestor da aplicação e serve apenas para visualizar os utilizadores e algumas das suas informações.

```
public void gerarCSV() {
    try (FileWriter writer = new FileWriter( fileName: "utilizadores.csv")) {

        // Escrever cabeçalho
        writer.append("ID, Nome, Data de Nascimento, Contacto, Tipo\n");

        // Escrever informações dos utentes
        escreverCSV(writer, utente, TipoUtilizador.UTENTE);

        // Escrever informações dos médicos
        escreverCSV(writer, medico, TipoUtilizador.MEDICO);

        // Escrever informações dos familiares
        escreverCSV(writer, familiar, TipoUtilizador.FAMILIAR);

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

3 usages
private void escreverCSV(FileWriter writer, Map<Integer, ? extends Utilizador> map, TipoUtilizador tipo) throws IOException {
    for (Utilizador utilizador : map.values()) {
        writer.append(String.format("%d, %s, %s, %d, %s\n",
                utilizador.getId(), utilizador.getNome(), utilizador.getDataNascimento(), utilizador.getContacto(), tipo.name()));
    }
}
```

Figura 3.17: Método para guardar o ficheiro *csv*.

3.2.7 Métodos Auxiliares

Métodos geradores de id´s:

Tal como referido anteriormente, os valores atribuídos aos id´s das `consultas`, `exames` e `prescricoes` são únicos em todo o universo de registo de dados hospitalar. Para que fosse possível esta unicidade, formam criados métodos que gerassem o próximo id disponível.

O funcionamento destes métodos passa por percorrer a estrutura de dados referente ao objeto procurado em questão, identificando todos os id´s já utilizados que, posteriormente são adicionados a um Array, `listadeIds`.

Assim, com o tamanho dessa estrutura é possível obter o próximo id disponível facilmente. Na figura 3.18 é apresentado o método `gerarIdConsulta`. Importante referir que, devido à extrema semelhança, a apresentação dos outros métodos é omitida, sendo eles o `gerarIdExame` e `gerarIdPrescricao`.

```
public int gerarIdConsulta(){

    List<Integer> listadeIds = new ArrayList<>();

    for (List<Consulta> consultas : consulta.values()){
        if(consultas != null){
            for (Consulta consulta : consultas){
                listadeIds.add(consulta.getConsultaID());
            }
        }
    }

    return listadeIds.size();
}
```

Figura 3.18: Método parar gerar o próximo id consulta.

Métodos *existe*:

Estes métodos foram criados com o intuito de verificar se determinada informação a ser inserida na base de dados não se encontra já na mesma. Isto é, serve para verificar no momento de registar um novo utilizador, se o id inserido já existe em alguma lista de id´s dos vários utilizadores.

Assim, foi criado o método `existeID`. Para permitir uma procura mais específica em cada tipo de utilizador, foram criados os métodos `existeUtente`, `existeProfSaude` e `existeFamiliar`. Estes métodos retornam um boleano consoante o sucesso na procura do id.

Para além do registo, estes métodos são tambémm chamados no tanto no momento de *login* bem como no momento de marcar algum tipo de marcação, por exemplo, um médico só pode agendar uma consulta para um utente que se encontre registado na base de dados do hospital. De seguida, apresenta-se na figura 3.19 apenas um dos métodos supramencionados como forma de simplificação.

```
public boolean existeUtente(int utenteID) { return utente.containsKey(utenteID); }
```

Figura 3.19: Método parar verificar a existência do utenteID.

3.3 Interface - *GestorInterface*

Para esta aplicação foi implementada uma interface, com a finalidade de servir como meio de comunicação entre o cliente e o servidor. É nesta interface que os métodos criados no *Gestor* são utilizados.

Nesta interface estão estabelecidos todos os métodos, sendo eles os responsáveis por criar novas entidades, listar, carregar e guardar ficheiros. Estão estabelecidas também, todas estas operações pois são as necessárias ao gestor da aplicação.

```
import java.io.IOException;
import java.rmi.Remote;

7 usages  1 implementation
public interface GestorInterface extends Remote {
    1 usage  1 implementation
    public void criarUtente(int id, String nome, String dataNascimento, int contacto, String palavraPasse) throws IOException;

    1 usage  1 implementation
    public void criarProfSaude(int id, String nome, String dataNascimento, int contacto, String palavraPasse) throws IOException;

    1 usage  1 implementation
    public void criarFamiliar(int id, String nome, String dataNascimento, int contacto, int idAssociado, String palavraPasse) throws IOException;

    1 usage  1 implementation
    public void criarConsulta(int utenteID,int medicoID, String medico, String dataHora) throws IOException;

    1 usage  1 implementation
    public void criarPrescricao(int utenteID, String medicamento, int dose, int duracao, String tomas, String dataInicio) throws IOException;
```

Figura 3.20: Excerto da interface *GestorInterface*

3.4 Exceções

As exceções criadas são aplicadas na classe *Gestor* de forma a indicar determinados erros que podem acontecer. Por exemplo, se for requisitado o registo de um novo utilizador que já existe na base de dados é esperado que o programa não permita o registo. Desta forma, foram criadas as exceções `UtenteJaExiste`, `ProfSaudeJaExiste` e `FamiliarJaExiste` para abranger os diferentes tipos de utilizadores no momento do registo. A seguir, apresenta-se na Figura 3.21 o exemplo relativo à exceções, dado que, todas mantêm a mesma estrutura.

```
package Entidades.Excecoes;

import java.rmi.RemoteException;

5 usages
public class UtenteJaExiste extends RemoteException {
    1 usage
    public UtenteJaExiste(String mensagem) { super(mensagem); }
}
```

Figura 3.21: Método parar verificar a existência do utenteID.

3.5 *GestorServidor*

Esta classe define o servidor da aplicação. Este *servidor RMI* responde a métodos remotos que estão na interface *GestorInterface*. Este servidor permanece sempre em execução à espera de algum cliente.

A classe *GestorServidor* possui um *main* onde são criadas as instâncias para a implementação do objeto remoto e vai servir como ponto de entrada para o programa. Ainda neste *main* é criado um bloco de *try-catch* de modo a correr o servidor e caso aconteça algum erro inesperado ele indica que erro foi ao emitir uma mensagem. Esta classe está visível na figura 3.22.

```java
import java.net.MalformedURLException;
import java.rmi.Naming;
import java.rmi.RemoteException;

public class GestorServidor {
    public static void main(String[] args) {
        try {
            GestorInterface server = new Gestor();
            Naming.rebind( name: "rmi://localhost:50000/GM", server);
            System.out.println("Running");

            System.out.println("Pressione Enter para encerrar o servidor.");
            System.in.read();

        } catch (RemoteException e) {
            e.printStackTrace();
            System.out.println("Erro de RemoteException: " + e.getMessage());
            throw new RuntimeException(e);
        } catch (MalformedURLException e) {
            e.printStackTrace();
            System.out.println("Erro de MalformedURLException: " + e.getMessage());
            throw new RuntimeException(e);
        } catch (Exception e) {
            e.printStackTrace();
            System.out.println("Erro inesperado: " + e.getMessage());
            throw new RuntimeException(e);
        }
    }
}
```

Figura 3.22: Classe *GestorServidor*

3.6 *GestorCliente*

A classe *GestorCliente* funciona como o cliente da aplicação, onde os métodos definidos na classe *Gestor* estão a ser executados remotamente. Consequentemente, após invocar a interface correspondente à classe *Gestor*, a *GestorInterface*, torna-se possível executar os métodos requeridos e manipular a informação.

O cliente, representado pela classe *GestorCliente*, interage com o servidor invocando esses métodos para realizar diversas operações, estabelecendo assim a comunicação entre as partes. É também nesta classe que se encontra a parte interativa da aplicação.

```java
import java.rmi.RemoteException;


public class GestorCliente {

    no usages
    protected GestorCliente() throws RemoteException{
        super();
    }

    public static void main(String[] args) throws IOException {

        GestorInterface gm = null;
        try {
            gm = (GestorInterface) Naming.lookup( name: "rmi://localhost:50000/GM");
        } catch (NotBoundException e) {
            throw new RuntimeException(e);
        } catch (MalformedURLException e) {
            throw new RuntimeException(e);
        } catch (RemoteException e) {
            System.err.println("Erro ao conectar ao servidor RMI: " + e.getMessage());
            e.printStackTrace();

        }

        gm.carregarFicheiros();
        gm.inicializar();
        gm.processarEscolhaUtilizador();
        gm.guardarFicheiros();

    }
}
```

Figura 3.23: Classe *GestorCliente*

3.7 Funcionalidades Extra - Estatística, Palavra-Passe, Notificações

3.7.1 Estatística - *numeroConsultas*

A fim de se providenciar ao profissional de saúde uma melhor forma de analisar a quantidade de consultas dadas foi criado um método com esta finalidade.

O método *numeroConsultas* recebe como parâmetro um id relativo a um médico e retorna um inteiro que é o número de consultas total que este já deu. Este método encontra-se representado na figura 3.24:

```
1 usage
public int numeroConsultas(int medicoID) {
    // Inicializa a variável que armazenará o total de consultas
    int totalConsultas = 0;

    // Itera sobre os valores do mapa 'consulta' (cada valor é uma lista de consultas)
    for (List<Consulta> consultasMedico : consulta.values()) {
        // Verifica se a lista de consultas não é nula
        if (consultasMedico != null) {
            // Itera sobre as consultas na lista
            for (Consulta consulta : consultasMedico) {
                // Verifica se o ID do médico associado à consulta é igual ao 'medicoID' fornecido
                if (consulta.getMedicoID() == medicoID) {
                    // Incrementa o contador de consultas
                    totalConsultas++;
                }
            }
        }
    }

    // Retorna o total de consultas associadas ao médico com o 'medicoID'
    return totalConsultas;
}
```

Figura 3.24: Método relativo à estatística do médico

3.7.2 Palavra-Passe

Com a finalidade de aumentar a segurança das contas dos utilizadores e de manter os seus dados privados, foi criado um atributo extra a cada um deles denomidado por `palavraPasse`.

Esta palavra-passe é preenchida no momento do registo da conta e é sempre pedida e verificada quando ocorre o *login*. De forma a verificar corretamente esta palavra-passe foi usado o seguinte método:

```
public boolean verificarPasse(int id, TipoUtilizador tipo, String passeTentada){

    if(tipo == TipoUtilizador.UTENTE){
        Utente utenteAtual = utente.get(id);
        String palavraPasse = utenteAtual.getPalavraPasse();
        return(palavraPasse.equals(passeTentada));

    }else if(tipo == TipoUtilizador.MEDICO){
        ProfSaude profissionalAtual = medico.get(id);
        String palavraPasse = profissionalAtual.getPalavraPasse();
        return(palavraPasse.equals(passeTentada));

    }else if(tipo == TipoUtilizador.FAMILIAR){
        Familiar familiarAtual = familiar.get(id);
        String palavraPasse = familiarAtual.getPalavraPasse();
        return(palavraPasse.equals(passeTentada));
    }
    return false;
}
```

Figura 3.25: Método relativo à verificação da palavra-passe

Este método retorna portanto um valor *booleano* em que retornando *true* a autenticação da conta é permitida e caso contrário não permite o acesso à conta em questão.

3.7.3 Notificações - Caixa de mensagens

De modo a que um utente e também o seu respetivo familiar, tenham conhecimento das marcações que lhe são feitas foi criado uma **caixa de mensagens**. Ela guarda todas as novas operações que são feitas como a marcação de consultas, prescrições e exames.

Os métodos referentes a estas operações encontram-se na figura 3.26.

```
public void criarNotificacao(int utenteID, String mensagem) {
    if (utente.containsKey(utenteID)) {
        Utente utenteAtual = utente.get(utenteID);
        utenteAtual.adicionarNotificacao(mensagem);
    }
}
2 usages
public void listarNotificacoes(int utenteID) {
    Utente utenteAtual = utente.get(utenteID);
    List<String> notificacoes = utenteAtual.getNotificacoes();

    if(notificacoes.isEmpty()){
        System.out.println("\nNão tem nenhuma notificação.\n");
    }else{
        for (String notificacao : notificacoes) {
            System.out.println(notificacao);
        }
    }
}
```

Figura 3.26: Métodos relativos à criação de notificações.

Por último foi ainda acrescentado um método em que sempre que o utente faz *login* na sua conta é verficado quando é que foi a sua última consulta. Este método serve para verificar se a última consulta do utente foi à mais de 6 meses e, se o for, manda uma notificação a alertá-lo desse feito e a sugerir para marcar um consulta de rotina. Este método encontra-se na figura 3.27

```java
public void verificarUltimaConsulta(int utenteID) {
    List<Consulta> consultas = consulta.get(utenteID);

    if (consultas != null && !consultas.isEmpty()) {
        Consulta ultimaConsulta = consultas.get(consultas.size() - 1);
        String dataConsultaStr = ultimaConsulta.getDataHora();

        // Converter a string da data para o formato LocalDate
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:ss");
        LocalDate dataConsulta = LocalDate.parse(dataConsultaStr, formatter);

        // Verificar se a última consulta foi há mais de 6 meses
        LocalDate dataAtual = LocalDate.now();
        LocalDate dataLimite = dataAtual.minusMonths( monthsToSubtract: 6);

        if (dataConsulta.isBefore(dataLimite)) {
            criarNotificacao(utenteID, mensagem: "Você deve marcar uma nova consulta, pois a última ocorreu há mais de 6 meses e foi a "+dataConsulta +".");
        }
    }
}
```

Figura 3.27: Métodos relativos à notificação de uma consulta de rotina.

4. Conclusão

Os sistemas distribuídos estão a tornar-se cada vez mais relevantes, uma vez que permitem um acesso generalizado sem restrições de localização, partilha de recursos distribuídos por utilizadores distintos, distribuição de carga, o que resulta numa melhoria de desempenho e na tolerância a falhas, proporcionando assim uma melhoria de disponibilidade. [7] A crescente importância dos sistemas distribuídos destaca a relevância da tecnologia JAVA RMI, uma vez que facilita o desenvolvimento de aplicações distribuídas. [4]

Desta forma, a elaboração deste trabalho permitiu um maior aprofundamento e familiarização com a linguagem de programação JAVA e JAVA RMI. De uma forma geral, considera-se o objetivo deste trabalho prático atingido, sendo o sistema de registos de dados e cuidados de saúde CuidaUM criado e implementado com sucesso.

Tendo em conta o que foi mencionado anteriormente, e no que refere a melhorias futuras, poder-se-ia implementar uma funcionalidade de multi-clientes, isto é, que vários clientes pudessem estar conectados ao servidor ao mesmo tempo, dado que, com apenas uma relação cliente-servidor, não se tira o máximo partido desta tecnologia. Além disso, também tornava a aplicação mais próxima do contexto hospitalar real.

Por fim, do ponto de vista de gestão hospitalar, poderiam ser criadas mais estatística que fornecessem mais informação sobre a eficácia do trabalho realizado na entidade de saúde. A título de exemplo, poderia ser analisada a frequência dos vários tipos de exames, bem como a regularidade em que cada utente recorre ao hospital.

Referências

[1] w3Schools, "Interface Java.", https://www.w3schools.com/java/java-interface.asp

[2] "Exceções Java." [Online], https://www.w3schools.com/java/java-try-catch.asp

[3] "Breve Introdução ao RMI - Remote Method Invocation.", https://web.fe.up.pt/~eol/AIAD/aulas/JINIdocs/rmi1.html

[4] J. Gabriel, "Concorrência e Java RMI.", https://web.fe.up.pt/~eol/AIAD/aulas/JINIdocs/rmi1.html.

[5] "Arquitetura Cliente-Servidor," 2012, https://arqserv.wordpress.com/2012/03/17/como-funciona-a-arquitetura-cliente-servidor/

[6] FILExt, "Abrir SER ficheiros", https://filext.com/pt/extensao-do-arquivo/SER

[7] N. Preguiça, "Sistemas Distribuídos.", https://asc.di.fct.unl.pt/~smd/isctem/teoricas/sd-1-introducao.pdf