**CENTRO ESTADUAL DE EDUCAÇÃO TECNOLÓGICA PAULA SOUZA**

**FATEC DA ZONA LESTE**

**Desenvolvimento de Software Multiplataforma**

**AMANDA SILVA BRITO**

**ANAHI NARIELI GUARACHI MAMANI**

**DEBORA VIEIRA CARVALHO**

**ISABELLA DA SILVA GONCALVES**

**JULIA SANTOS BONGIOVANI DE OLIVEIRA**

**DOCUMENTAÇÃO – P2**

**SEREN – SISTEMA GERENCIADOR PARA PSICOLÓGOS**

**São Paulo – SP**

**2025**

**AMANDA SILVA BRITO**

**ANAHI NARIELI GUARACHI MAMANI**

**DEBORA VIEIRA CARVALHO**

**ISABELLA DA SILVA GONCALVES**

**JULIA SANTOS BONGIOVANI DE OLIVEIRA**

**(3º semestre de DSM)**

**SEREN – SISTEMA GERENCIADOR PARA PSICOLÓGOS**

Atividade apresentada ao Curso de Tecnologia em Desenvolvimento de Software Multiplataforma da Faculdade de Tecnologia da Zona Leste, a ser utilizado como requisito parcial para aprovação na disciplina de Técnicas de Programação II.

Docente: Prof. Sirley Ambrosia Vitorio Addão.

**São Paulo – SP**

**2025**

[**1 INTRODUÇÃO** 5](#_Toc200074960)

[**1.1 Propósito** 5](#_Toc200074961)

[**1.2 Contexto geral** 5](#_Toc200074962)

[**2. Código das classes** 5](#_Toc200074963)

[**2.2 Classes Model** 5](#_Toc200074964)

[**2.2.1 Classe Usuário** 5](#_Toc200074965)

[**2.2.2 Classe Paciente** 8](#_Toc200074966)

[**2.3 Classes DAO** 12](#_Toc200074967)

[**2.3.1 Classe UsuarioDAO** 13](#_Toc200074968)

[**2.3.2 Classe PacienteDAO** 15](#_Toc200074969)

[**2.4 Classes controller** 20](#_Toc200074970)

[**2.4.1 Classe CadastroController** 20](#_Toc200074971)

[**2.4.2 Classe LoginController** 21](#_Toc200074972)

[**2.4.3 Classe CriarPacienteController** 22](#_Toc200074973)

[**2.4.4 Classe BuscarPacientesController** 23](#_Toc200074974)

[**2.4.5 Classe BuscarPacienteControlle** 24](#_Toc200074975)

[**2.4.6 Classe AlterarPacienteController** 25](#_Toc200074976)

[**2.4.7 Classe ExcluirPacienteController** 26](#_Toc200074977)

[**2.5 Classes utilitárias** 27](#_Toc200074978)

[**2.5.1 Classe ConfigUtil** 27](#_Toc200074979)

[**2.5.2 Classe MongoConnection** 29](#_Toc200074980)

[**2.5.3 Interface Validador** 30](#_Toc200074981)

[**2.5.4 Classe ValidadorDadosUsuario** 31](#_Toc200074982)

[**2.5.5 Classe ValidadorDadosUsuarioLogin** 33](#_Toc200074983)

[**2.5.6 Classe ValidadorDadosPaciente** 34](#_Toc200074984)

[**2.5.7 Classe DataUtil** 36](#_Toc200074985)

[**2.5.8 Classe LembrarEmailUtil** 37](#_Toc200074986)

[**2.6 View** 40](#_Toc200074987)

[**2.6.1 Classe Login** 40](#_Toc200074988)

[**2.6.2 Classe Cadastro** 44](#_Toc200074989)

[**2.6.3 Classe Tela Inicial** 47](#_Toc200074990)

[**2.6.4 Classe Tela Pacientes** 50](#_Toc200074991)

[**2.6.5 Classe Tela Cadastrar Pacientes** 57](#_Toc200074992)

[**2.6.6 Classe Tela Editar pacientes** 60](#_Toc200074993)

[**2.7 Components** 64](#_Toc200074994)

[**2.7.1 Classe CustomJCalendar** 64](#_Toc200074995)

[**2.7.2 Classe RoundedButton** 68](#_Toc200074996)

[**2.7.3 Classe RoundedJDateChooser** 70](#_Toc200074997)

[**2.7.4 Classe RoundedJFormattedTextField** 72](#_Toc200074998)

[**2.7.5 Classe RoundedJPasswordField** 74](#_Toc200074999)

[**2.7.6 Classe RoundedJTextField** 76](#_Toc200075000)

[**2.7.7 Classe RoundedPanel** 78](#_Toc200075001)

[**2.8 Conclusão** 79](#_Toc200075002)

# **1 INTRODUÇÃO**

## **1.1 Propósito**

Este documento apresenta a estrutura e as classes implementadas para a execução do sistema Seren, utilizando a biblioteca Java Swing. A arquitetura segue o padrão MVC, com explicações sobre os princípios de POO e padrão de projeto utilizado.

## **1.2 Contexto geral**

O projeto tem como objetivo desenvolver um Sistema Gerenciador com funções de login, cadastro de usuários, além de operações de adição, busca, edição e exclusão de pacientes. O sistema é baseado no projeto integrador atual, denominado Seren, cujo propósito é o gerenciamento de pacientes para psicólogos.

# **2. Código das classes**

Abaixo, apresentamos o código das classes desenvolvidas, com divisão entre classes model, controller, DAO (data access object), utilitárias, views e components.

## **2.2 Classes Model**

As classes models tem o propósito de representar as entidades armazenadas no banco de dados utilizado, MongoDB, nesta aplicação é introduzida duas classes, Usuario e Paciente.

### **2.2.1 Classe Usuário**

A classe Usuario representa um usuário do sistema, com dados como nome, email e senha. Utiliza encapsulamento ao manter seus atributos privados e expor apenas os getters e setters. A existência de dois construtores mostra sobrecarga, permitindo criar objetos de forma flexível, dependendo do contexto, sendo um novo cadastro ou dados vindos do banco.

Segue código abaixo:

package com.seren.model;

import org.bson.codecs.pojo.annotations.BsonId;

import org.bson.types.ObjectId;

// Classe Usuario que representa a entidade Usuário do Seren armazenada como documento no MongoDB.

// Demonstrando encapsulamento através de atributos privados com getters e setters.

public class Usuario {

// Atributos essenciais para a identificação e login do usuário.

// Marcação que indica que este id se trata do ObjectId localizado no banco de dados.

@BsonId

private ObjectId id;

private String nome;

private String email;

private long telefone;

private String senha;

// Construtor para criar um novo usuário, com a criação de um novo id e formatação do telefone.

public Usuario(String nome, String email, String telefone, String senha) {

this.id = new ObjectId();

this.nome = nome;

this.telefone = Long.parseLong(telefone);

this.email = email;

this.senha = senha;

}

// Construtor para usuário vindo do banco.

public Usuario(ObjectId id, String nome, String email, long telefone, String senha) {

this.id = id;

this.nome = nome;

this.telefone = telefone;

this.email = email;

this.senha = senha;

}

// Getters e Setters

public ObjectId getId() {

return id;

}

public String getNome() {

return nome;

}

public void setNome(String nome) {

this.nome = nome;

}

public String getEmail() {

return email;

}

public void setEmail(String email) {

this.email = email;

}

public long getTelefone() {

return telefone;

}

public void setTelefone(int telefone) {

this.telefone = telefone;

}

public String getSenha() {

return senha;

}

public void setSenha(String senha) {

this.senha = senha;

}

}

### **2.2.2 Classe Paciente**

A classe Paciente encapsula os dados de uma pessoa cadastrada no sistema, como nome, email e data de nascimento. Possui múltiplos construtores, demonstrando sobrecarga, para diferentes contextos: criação de novo paciente, leitura do banco e conversão de documento MongoDB. O método getIdade() representa um comportamento do objeto baseado em seus dados, reforçando a ideia de orientação a objetos.

Segue código abaixo:  
  
package com.seren.model;

import com.seren.util.DataUtil;

import java.util.Date;

import org.bson.Document;

import org.bson.codecs.pojo.annotations.BsonId;

import org.bson.types.ObjectId;

// Classe Paciente que representa a entidade Paciente do Seren armazenada como documento no MongoDB.

// Demonstrando encapsulamento através de atributos privados com getters e setters.

public class Paciente {

// Atributos essenciais para o cadastro de um paciente.

// Marcação que indica que este id se trata do ObjectId localizado no banco de dados.

@BsonId

private ObjectId id;

private String nome;

private String profissao;

private String genero;

private String estadoCivil;

private long telefone;

private String email;

private String preferenciaContato;

private Date dataNascimento;

private ObjectId id\_usuario;

// Construtor para novo paciente, com a criação de um novo id e formatação do telefone. .

public Paciente(String nome, String profissao, String genero, String estadoCivil, String telefone,

String email, String preferenciaContato, Date dataNascimento, ObjectId id\_usuario) {

this.id = new ObjectId();

this.nome = nome;

this.profissao = profissao;

this.genero = genero;

this.estadoCivil = estadoCivil;

this.telefone = Long.parseLong(telefone);

this.email = email;

this.preferenciaContato = preferenciaContato;

this.dataNascimento = dataNascimento;

this.id\_usuario = id\_usuario;

}

// Construtor para paciente vindo do banco.

public Paciente(ObjectId id, String nome, String profissao, String genero, String estadoCivil, Long telefone,

String email, String preferenciaContato, Date dataNascimento, ObjectId id\_usuario) {

this.id = id;

this.nome = nome;

this.profissao = profissao;

this.genero = genero;

this.estadoCivil = estadoCivil;

this.telefone = telefone;

this.email = email;

this.preferenciaContato = preferenciaContato;

this.dataNascimento = dataNascimento;

this.id\_usuario = id\_usuario;

}

// Construtor para paciente que foi recebido como documento pelo banco de dados.

public static Paciente pacienteDocumento(Document doc) {

return new Paciente(

doc.getObjectId("\_id"),

doc.getString("nome"),

doc.getString("profissao"),

doc.getString("genero"),

doc.getString("estadoCivil"),

doc.getLong("telefone"),

doc.getString("email"),

doc.getString("preferenciaContato"),

doc.getDate("dataNascimento"),

doc.getObjectId("id\_usuario")

);

}

// Getters e Setters

public ObjectId getId() {

return id;

}

public String getNome() {

return nome;

}

public void setNome(String nome) {

this.nome = nome;

}

public String getProfissao() {

return profissao;

}

public void setProfissao(String profissao) {

this.profissao = profissao;

}

public String getGenero() {

return genero;

}

public void setGenero(String genero) {

this.genero = genero;

}

public String getEstadoCivil() {

return estadoCivil;

}

public void setEstadoCivil(String estadoCivil) {

this.estadoCivil = estadoCivil;

}

public long getTelefone() {

return telefone;

}

public void setTelefone(long telefone) {

this.telefone = telefone;

}

public String getEmail() {

return email;

}

public void setEmail(String email) {

this.email = email;

}

public String getPreferenciaContato() {

return preferenciaContato;

}

public void setPreferenciaContato(String preferenciaContato) {

this.preferenciaContato = preferenciaContato;

}

public Date getDataNascimento() {

return dataNascimento;

}

public void setDataNascimento(Date dataNascimento) {

this.dataNascimento = dataNascimento;

}

public ObjectId getId\_usuario() {

return id\_usuario;

}

public void setId\_usuario(ObjectId id\_usuario) {

this.id\_usuario = id\_usuario;

}

// Método para retornar a idade com cálculo baseado na data de nascimento.

public int getIdade() {

return DataUtil.calcularIdade(dataNascimento);

}

}

## **2.3 Classes DAO**

As classes DAO realizam operações no banco de dados, manipulando diretamente os dados por meio de objetos das classes model. No sistema, temos duas, UsuarioDAO e PacienteDAO, cada uma responsável por lidar com os dados de sua respectiva model.

### **2.3.1 Classe UsuarioDAO**

A classe UsuarioDAO gerencia o acesso ao banco de dados para operações com usuários. Utiliza encapsulamento ao isolar o acesso com a coleção, mantendo os métodos de cadastro e login públicos, protegendo o acesso direto à base de dados.

Segue código abaixo:

package com.seren.dao;

import com.mongodb.client.MongoCollection;

import com.mongodb.client.MongoDatabase;

import com.seren.model.Usuario;

import org.bson.Document;

import com.seren.util.MongoConnection;

import org.mindrot.jbcrypt.BCrypt;

// Classe DAO(Data Access Object) responsável por gerenciar as operações de acesso ao banco de dados relacionada com a classe model Usuario.

public class UsuarioDAO {

// Atributo privado que serve como acesso a coleção Usuario do MongoDB.

private final MongoCollection<Document> collection;

// Construtor da classe UsuarioDAO, inicializa a conexão com a database e coleção "Usuario".

public UsuarioDAO() {

MongoDatabase db = MongoConnection.getInstancia().getDatabase();

this.collection = db.getCollection("Usuario");

}

// Método que realiza o cadastro de um novo usuário no banco de dados, recebe como parâmetro um objeto do tipo Usuario.

public boolean cadastrar(Usuario usuario) {

// Criação de um filtro de busca no formato BSON(usado pelo MongoDB), para encontrar um documento

// com o campo "email" igual ao email do usuário que entra como parâmetro.

Document filtro = new Document("email", usuario.getEmail());

// Realiza a busca com o filtro.

Document resultado = collection.find(filtro).first();

// Caso a busca não retorne um usuário com o email passado, utiliza o bcrypt para retornar um hash da senha criptografada.

if (resultado == null) {

String senhaCriptografada = BCrypt.hashpw(usuario.getSenha(), BCrypt.gensalt());

// Criação de um documento com os dados do usuário.

Document u = new Document(

"\_id", usuario.getId())

.append("email", usuario.getEmail())

.append("username", usuario.getNome())

.append("senha", senhaCriptografada)

.append("telefone", usuario.getTelefone());

// Inserção do documento na coleção "Usuario".

collection.insertOne(u);

return true;

}

return false;

}

// Método que autentifica um usuário com o email e senha fornecido.

public Usuario logar(String email, String senha) {

// Criação de um filtro de busca no formato BSON(usado pelo MongoDB), para encontrar um documento

// com o campo "email" igual ao email passado como parâmetro.

Document filtro = new Document("email", email);

// Realiza a busca com o filtro.

Document resultado = collection.find(filtro).first();

// Se for retornando um usuário, é instanciado um Usuario com os dados retornados.

if (resultado != null) {

// Compara a senha fornecida com a senha criptografada armazenada

String senhaArmazenada = resultado.getString("senha");

if (BCrypt.checkpw(senha, senhaArmazenada)) {

return new Usuario(

resultado.getObjectId("\_id"),

resultado.getString("username"),

resultado.getString("email"),

resultado.getLong("telefone"),

null

);

}

}

return null;

}

}

### **2.3.2 Classe PacienteDAO**

A classe PacienteDAO gerencia o acesso ao banco de dados para operações com pacientes. Usa encapsulamento para proteger a coleção MongoDB, centralizando métodos de cadastro, busca, alteração e exclusão de pacientes.

Segue código abaixo:

package com.seren.dao;

import com.mongodb.client.MongoCollection;

import com.mongodb.client.MongoDatabase;

import com.seren.model.Paciente;

import com.seren.util.MongoConnection;

import java.util.ArrayList;

import org.bson.Document;

import org.bson.types.ObjectId;

// Classe DAO(Data Access Object) responsável por gerenciar as operações de acesso ao banco de dados relacionada com a classe model Paciente.

public class PacienteDAO {

// Atributo privado que representa a coleção Paciente no MongoDB.

private final MongoCollection<Document> collection;

// Construtor da classe PacienteDAO. Inicializa a conexão com a base de dados e acessa a coleção "Paciente".

public PacienteDAO() {

MongoDatabase db = MongoConnection.getInstancia().getDatabase();

this.collection = db.getCollection("Paciente");

}

// Método que realiza o cadastro de um novo paciente no banco de dados.

public boolean cadastrarPaciente(Paciente paciente) {

// Criação de um filtro de busca no formato BSON(usado pelo MongoDB), para encontrar um documento

// com o campo "email" igual ao email do paciente que entra como parâmetro.

Document filtro = new Document("email", paciente.getEmail());

// Realiza a busca com o filtro.

Document resultado = collection.find(filtro).first();

// Caso a busca não retorne um paciente com o email passado, cria um novo documento

// com os dados de paciente e o insere na coleção "Pacientes".

if (resultado == null) {

Document p = new Document("nome", paciente.getNome())

.append("profissao", paciente.getProfissao())

.append("genero", paciente.getGenero())

.append("estadoCivil", paciente.getEstadoCivil())

.append("telefone", paciente.getTelefone())

.append("email", paciente.getEmail())

.append("preferenciaContato", paciente.getPreferenciaContato())

.append("dataNascimento", paciente.getDataNascimento())

.append("id\_usuario", paciente.getId\_usuario());

collection.insertOne(p);

return true;

}

return false;

}

// Método que busca todos os pacientes associados a um determinado usuário (por ID).

public ArrayList<Document> buscarPaciente(ObjectId id) {

// Criação do filtro com base no campo "id\_usuario" igual o id passado como parâmetro.

Document filtro = new Document("id\_usuario", id);

// Executa a consulta e retorna os documentos/pacientes encontrados como uma ArrayList.

ArrayList<Document> listaPacientes = new ArrayList<>();

return listaPacientes = collection.find(filtro).into(listaPacientes);

}

// Sobrecarga do método buscarPaciente, agora busca também por nome além do ID do usuário.

public ArrayList<Document> buscarPaciente(ObjectId id, String nome) {

// Filtro com ID de usuário e nome do paciente.

Document filtro = new Document("id\_usuario", id)

.append("nome", nome);

ArrayList<Document> listaPacientes = new ArrayList<>();

return listaPacientes = collection.find(filtro).into(listaPacientes);

}

// Método que atualiza os dados de um paciente, caso algum dado tenha sido alterado.

public boolean alterarPaciente(Paciente paciente) {

// Cria o filtro com base no ID do paciente registrado no banco e o id do paciente passado como parâmetro.

Document filtro = new Document("\_id", paciente.getId());

// Realiza a busca com o filtro.

Document resultado = collection.find(filtro).first();

// Se não existe este paciente, nada é alterado.

if (resultado == null) {

return false;

}

// Verifica se os dados do paciente são diferentes dos armazenados.

boolean dadosIguais

= paciente.getNome().equals(resultado.getString("nome"))

&& paciente.getProfissao().equals(resultado.getString("profissao"))

&& paciente.getGenero().equals(resultado.getString("genero"))

&& paciente.getEstadoCivil().equals(resultado.getString("estadoCivil"))

&& paciente.getTelefone() == resultado.getLong("telefone")

&& paciente.getEmail().equals(resultado.getString("email"))

&& paciente.getPreferenciaContato().equals(resultado.getString("preferenciaContato"))

&& paciente.getDataNascimento().equals(resultado.getDate("dataNascimento"))

&& paciente.getId\_usuario().equals(resultado.getObjectId("id\_usuario"));

// Nenhum dado foi alterado, então não há necessidade de atualização.

if (dadosIguais) {

return false;

}

// Criação do documento com os dados atualizados.

Document dadosAtualizados = new Document()

.append("nome", paciente.getNome())

.append("profissao", paciente.getProfissao())

.append("genero", paciente.getGenero())

.append("estadoCivil", paciente.getEstadoCivil())

.append("telefone", paciente.getTelefone())

.append("email", paciente.getEmail())

.append("preferenciaContato", paciente.getPreferenciaContato())

.append("dataNascimento", paciente.getDataNascimento())

.append("id\_usuario", paciente.getId\_usuario());

// Atualiza os dados do paciente usando o operador $set(usado pelo MongoDB).

Document atualizacao = new Document("$set", dadosAtualizados);

// Atualiza na coleção "Pacientes" o paciente encontrado com os dados alterados.

collection.updateOne(filtro, atualizacao);

return true;

}

// Método que remove um paciente da coleção com base no seu ID.

public boolean excluirPaciente(ObjectId id) {

// Cria um filtro com base no ID do paciente registrado no banco de dados e o id passado como parâmetro.

Document filtro = new Document("\_id", id);

// Realiza a busca com o filtro.

Document resultado = collection.find(filtro).first();

// Se encontrado, realiza a exclusão e verifica se exatamente um documento foi deletado com base na contagem de exclusão.

if (resultado != null) {

return collection.deleteOne(resultado).getDeletedCount() == 1;

}

return false;

}

}

## **2.4 Classes controller**

As classes controllers atuam como intermediárias entre as views e as DAO, reunindo e repassando informações para esta comunicação. Temos ao total, sete controllers.

### **2.4.1 Classe CadastroController**

A classe CadastroController gerencia a lógica de cadastro de usuários, atuando como intermediária entre a interface do usuário e a classe UsuarioDAO. Usa encapsulamento, escondendo o acesso com a DAO , e centralizando a criação do objeto Usuario e o envio dos dados para a DAO como método público.

Segue código abaixo:

package com.seren.controller;

import com.seren.dao.UsuarioDAO;

import com.seren.model.Usuario;

// Classe controller responsável por gerenciar a lógica de cadastro de usuários.

// Atua como intermediária entre a view Cadastro e a UsuarioDAO.

public class CadastroController {

// Instância da classe UsuarioDAO, responsável pelas operações no banco de dados para a entidade Usuario.

private final UsuarioDAO usuarioDAO;

// Construtor da classe controller, inicializa o objeto DAO para uso no método cadastrarUsuario.

public CadastroController() {

this.usuarioDAO = new UsuarioDAO();

}

// Método que executa a operação de cadastro de um novo usuário.

// Recebe os dados para cadastro de um usuário como parâmetros, cria um objeto Usuario, e repassa para a DAO.

// Retorna o "true"(cadastro sucedido)/"false"(usuário já existente) recebido da DAO.

public boolean cadastrarUsuario(String nome, String email, String telefone, String senha) {

Usuario usuario = new Usuario(nome, email, telefone, senha);

boolean sucesso = usuarioDAO.cadastrar(usuario);

return sucesso;

}

}

### **2.4.2 Classe LoginController**

A classe LoginController gerencia a lógica de login de usuários, atuando como intermediária entre a interface de login e a classe UsuarioDAO. Demonstra encapsulamento com o acesso privado com a DAO e a verificação de credenciais, retorno do objeto Usuario ou null, como método público.

Segue código abaixo:

package com.seren.controller;

import com.seren.dao.UsuarioDAO;

import com.seren.model.Usuario;

// Classe controller responsável por gerenciar a lógica de login de usuários.

// Atua como intermediária entre a view Login e a UsuarioDAO.

public class LoginController {

// Instância da classe UsuarioDAO, responsável pelas operações no banco de dados para a entidade Usuario.

private final UsuarioDAO usuarioDAO;

// Construtor da classe controller, inicializa o objeto DAO para uso no método loginUsuario.

public LoginController() {

this.usuarioDAO = new UsuarioDAO();

}

// Método que executa a operação de login de um usuário.

// Recebe como parâmetros o email e a senha digitados pelo usuário.

// Repassa os dados para a DAO, que realiza a verificação

// e retorna o objeto do tipo Usuario(caso dados estejam concretos) ou null(dados errados).

public Usuario loginUsuario(String email, String senha){

Usuario sucesso = usuarioDAO.logar(email, senha);

return sucesso;

}

}

### **2.4.3 Classe CriarPacienteController**

A classe CriarPacienteController gerencia a criação de pacientes, funcionando como intermediária entre a interface de cadastro e a PacienteDAO. Encapsulamento é demonstrado através do acesso privado com a DAO, centralizando o método de criação do objeto Paciente e o envio dele como público.

Segue código abaixo.

package com.seren.controller;

import com.seren.dao.PacienteDAO;

import com.seren.model.Paciente;

import java.util.Date;

import org.bson.types.ObjectId;

// Classe controller responsável por gerenciar a lógica de criação de um novo paciente.

// Atua como intermediária entre a view TelaCadastrarPaciente e a PacienteDAO.

public class CriarPacienteController {

// Instância da classe PacienteDAO responsável pelas operações no banco de dados para a entidade Paciente.

private final PacienteDAO pacienteDAO;

// Construtor da classe controller, inicializa o objeto DAO para uso no método criarPaciente.

public CriarPacienteController() {

this.pacienteDAO = new PacienteDAO();

}

// Método que executa a criação de um novo paciente.

// Recebe como parâmetros os dados do paciente, cria um objeto do tipo Paciente e repassa para a DAO realizar o cadastro.

// Retorna o "true"(paciente foi cadastrado)/"false"(paciente já registrado) recebido da DAO.

public boolean criarPaciente(String nome, String profissao, String genero, String estadoCivil,

String telefone, String email, String preferenciaContato, Date dataNascimento, ObjectId id\_usuario) {

Paciente paciente = new Paciente(nome, profissao, genero, estadoCivil, telefone, email, preferenciaContato, dataNascimento, id\_usuario);

boolean sucesso = pacienteDAO.cadastrarPaciente(paciente);

return sucesso;

}

}

### **2.4.4 Classe BuscarPacientesController**

A classe BuscarPacientesController gerencia a busca de pacientes vinculados a um usuário, atuando como intermediária entre a interface e a PacienteDAO. Demonstra o encapsulamento ao privar o acesso com a DAO e a chamada para obter a lista de pacientes do banco como método público.

Segue código abaixo.

package com.seren.controller;

import com.seren.dao.PacienteDAO;

import java.util.ArrayList;

import org.bson.Document;

import org.bson.types.ObjectId;

// Classe controller responsável por realizar a busca de todos os pacientes vinculados a um usuário.

// Atua como intermediária entre a view TelaPacientes e a PacienteDAO.

public class BuscarPacientesController {

// Instância protegida de PacienteDAO, permitindo o acesso pela classe filha BuscarPacienteController.

protected final PacienteDAO pacienteDAO;

// Construtor da classe controller, inicializa o objeto DAO para ser usado nos métodos de busca.

public BuscarPacientesController() {

this.pacienteDAO = new PacienteDAO();

}

// Método que executa a operação de busca de pacientes, recebendo como parâmetro o id do usuário.

// Repassa o id para a DAO e retorna uma ArrayList com todos os pacientes recebidos.

public ArrayList<Document> buscarPacientes(ObjectId id) {

ArrayList<Document> sucesso = pacienteDAO.buscarPaciente(id);

return sucesso;

}

}

### **2.4.5 Classe BuscarPacienteControlle**

A classe BuscarPacienteController herda de BuscarPacientesController, demonstrando o conceito de herança para especializar a busca de pacientes por nome. Atua como intermediária entre a interface e a PacienteDAO, reutilizando métodos da classe pai.

Segue código abaixo:

package com.seren.controller;

import java.util.ArrayList;

import org.bson.Document;

import org.bson.types.ObjectId;

// Classe controller que estende/herda BuscarPacientesController para especializar a lógica de busca de apenas um paciente.

// Atua como intermediária entre a view TelaBuscarPaciente e a PacienteDAO.

public class BuscarPacienteController extends BuscarPacientesController {

// Método que executa a operação de busca de um paciente, recebe, como parâmetro, um id(do usuário) e o nome de um paciente.

// Repassa os dados para a DAO e retorna a ArrayList com o paciente recebido.

public ArrayList<Document> buscarPacientes(ObjectId id, String nome) {

ArrayList<Document> sucesso = pacienteDAO.buscarPaciente(id, nome);

return sucesso;

}

}

### **2.4.6 Classe AlterarPacienteController**

A classe AlterarPacienteController gerencia a lógica de alteração de dados de pacientes, atuando como intermediária entre a interface e a PacienteDAO. Usa encapsulamento para manter a instância do DAO privada e centraliza a atualização dos dados do paciente como método público.

Segue código abaixo:

package com.seren.controller;

import com.seren.dao.PacienteDAO;

import com.seren.model.Paciente;

import java.util.Date;

import org.bson.types.ObjectId;

// Classe controller responsável por gerenciar a lógica de alteração de dados de um paciente.

// Atua como intermediária entre a view TelaEditarPaciente e a PacienteDAO.

public class AlterarPacienteController {

// Instância da classe PacienteDAO responsável pelas operações no banco de dados para a entidade Paciente.

private final PacienteDAO pacienteDAO;

// Construtor da classe controller, inicializa o objeto DAO para uso no método alterarPaciente.

public AlterarPacienteController() {

this.pacienteDAO = new PacienteDAO();

}

// Método que executa a operação de alteração de dados de um paciente, recebe, como parâmetro, um objeto Paciente com os dados atualizados.

// Repassa os dados para a DAO e retorna o "true"(paciente alterado)/"false"(paciente não existe ou com mesmo dados) recebido da DAO.

public boolean alterarPaciente(Paciente paciente) {

boolean sucesso = pacienteDAO.alterarPaciente(paciente);

return sucesso;

}

}

### **2.4.7 Classe ExcluirPacienteController**

A classe ExcluirPacienteController gerencia a exclusão de pacientes, atuando como intermediária entre a interface e a PacienteDAO. Usa encapsulamento para manter a instância do DAO privada e centraliza a operação de remoção.

Segue código abaixo:

package com.seren.controller;

import com.seren.dao.PacienteDAO;

import org.bson.types.ObjectId;

// Classe controller responsável por gerenciar a lógica de exclusão de pacientes.

// Atua como intermediária entre a view TelaPacientes e a PacienteDAO.

public class ExcluirPacienteController {

// Instância da classe PacienteDAO responsável pelas operações no banco de dados para a entidade Paciente.

private final PacienteDAO pacienteDAO;

// Construtor da classe controller, inicializa o objeto DAO para uso no método excluirPaciente.

public ExcluirPacienteController() {

this.pacienteDAO = new PacienteDAO();

}

// Método que executa a exclusão de um paciente.

// Recebe como parâmetro o id do paciente a ser excluído e repassa a chamada para a DAO.

// Retorna o "true"(paciente excluído)/"false"(paciente não encontrado) recebido da DAO.

public boolean excluirPaciente(ObjectId id) {

boolean sucesso = pacienteDAO.excluirPaciente(id);

return sucesso;

}

}

## **2.5 Classes utilitárias**

Classes utilitárias são importantes para a reutilização no código, durante o desenvolvimento, elas foram essencias para o impedimento de redudância com a conexão com o banco MongoDB, validação de dados, ambos de pacientes e usuários, cálculo da data de nascimento e leitura de arquivo .properties, como lembrar email e a configuração do banco de dados.

### **2.5.1 Classe ConfigUtil**

As classes ConfigUtil e MongoConnection são utilitárias que facilitam a configuração e conexão com o banco de dados MongoDB. ConfigUtil encapsula o carregamento das propriedades do arquivo config.properties, garantindo fácil acesso. MongoConnection usa encapsulamento para proteger os detalhes da conexão, mantendo privados os atributos de URI, cliente e banco, e expõe métodos públicos para acesso controlado. Além disso, implementa o padrão Singleton para gerenciar uma única conexão reutilizável com o MongoDB, promovendo eficiência no acesso ao banco.

Segue o código de ambas abaixo:

package com.seren.util;

import java.io.InputStream;

import java.util.Properties;

// Classe utilitária responsável por carregar propriedades de configuração recebidas da config.properties.

public class ConfigUtil {

// Objeto Properties utilizado para armazenar as configurações carregadas.

private static final Properties properties = new Properties();

// Bloco estático responsável por carregar o arquivo "config.properties" assim que a classe for carregada.

static {

// Por estarmos trabalhando com um arquivo, é necessário a utilização de try e catch.

try (InputStream input = ConfigUtil.class.getClassLoader().getResourceAsStream("config.properties")) {

// Verifica se o arquivo foi encontrado no classpath.

if (input == null) {

throw new RuntimeException("Arquivo config.properties não encontrado no classpath.");

}

// Carregamento das propriedades a partir do arquivo.

properties.load(input);

} catch (Exception e) {

// Em caso de erro no carregamento, é lançado uma exceção.

throw new RuntimeException("Erro ao carregar config.properties", e);

}

}

// Getter que retorna o valor da propriedade pelo seu identificador/key.

public static String get(String key) {

return properties.getProperty(key);

}

}

### **2.5.2 Classe MongoConnection**

Segue o código de ambas abaixo:

package com.seren.util;

import com.mongodb.client.MongoClient;

import com.mongodb.client.MongoClients;

import com.mongodb.client.MongoDatabase;

// Classe utilitária MongoConnection onde é possível a conexão com o banco de dados.

public class MongoConnection {

// Conseguindo o nome da nossa database e a URI guardados na config.properties

// com o auxílio da nossa classe utilitária ConfigUtil.

private static final String URI = ConfigUtil.get("mongodb.uri");

private static final String DATABASE\_NOME = ConfigUtil.get("mongodb.database");

// Instância estática usada para implementar o padrão de projeto Singleton.

private static MongoConnection instancia;

// Cliente Mongo utilizado para se conectar ao MongoDB usando a URI.

private final MongoClient mongoClient;

// Construtor da nossa conexão, que inicializa o cliente Mongo com a URI.

private MongoConnection() {

mongoClient = MongoClients.create(URI);

}

// Getter que retorna instância única da conexão, garantindo reutilização

// e evitando múltiplas conexões desnecessárias com o banco de dados.

public static synchronized MongoConnection getInstancia() {

if (instancia == null) {

instancia = new MongoConnection();

}

return instancia;

}

// Getter que retorna a nossa database já conectada e acessível.

public MongoDatabase getDatabase() {

return mongoClient.getDatabase(DATABASE\_NOME);

}

// Método que encerra a conexão com o banco e limpa a instância.

public void close() {

mongoClient.close();

instancia = null;

}

}

As classes utilitárias Validador, ValidadorDadosUsuario, ValidadorDadosUsuarioLogin e ValidadorDadosPacientes são responsáveis por validar dados antes de realizar operações como cadastro ou login.

### **2.5.3 Interface Validador**

A interface Validador define um contrato com o método validar(), que é implementado por todas as classes filhas.

Segue código abaixo:

package com.seren.util;

// Interface que define um contrato para as classes utilitárias que realizam a validação de dados,

// sendo de extrema importância a implementação do método validar, de modo que retorne uma String de mensagem de erro.

public interface Validador {

String validar();

}

### **2.5.4 Classe ValidadorDadosUsuario**

ValidadorDadosUsuario valida os dados do usuário para cadastro, ela se utiliza do polimorfismo, sobreescrevendo e implementando o método validar() da classe mãe, realiza encapsulamento para proteger os atributos.

Segue código abaixo:

package com.seren.util;

// Classe utilitária responsável por validar os dados antes da criação de um objeto do tipo Usuário.

// Implementa a interface Validador.

public class ValidadorDadosUsuario implements Validador {

// Atributos essenciais para o construtor de Usuário.

private String username;

private String email;

private String telefone;

private String senha;

private String senhaConfirmar;

// Construtor para inicializar os dados a serem validados.

public ValidadorDadosUsuario(String username, String email, String telefone, String senha, String senhaConfirmar) {

this.username = username;

this.email = email;

this.telefone = telefone;

this.senha = senha;

this.senhaConfirmar = senhaConfirmar;

}

// Implementação do método validar da interface, utilizando sobrescrita, realizando polimorfismo.

@Override

public String validar() {

// Verificação se os dados estão vazios, retornando uma String que será utilizada como mensagem de erro.

if (email == null || email.isEmpty() || senha == null || senha.isEmpty()

|| telefone == null || telefone.isEmpty() || username == null || username.isEmpty()) {

return "Preencha todos os campos.";

}

// Verificação se o email possui "@".

if (!email.contains("@")) {

return "Insira um email válido!";

}

// Verifica se as senhas coincidem

if (!senha.equals(senhaConfirmar)) {

return "Senhas não coincidem!";

}

// Verificando se a senha é segura atendendo os critérios de letra maiúscula, caractere especial e números.

if (senha.length() < 8

|| !senha.matches(".\*[A-Z].\*")

|| !senha.matches(".\*[!@#$%^&\*()\_+=|<>?{}\\[\\]~-].\*")

|| !senha.matches(".\*\\d.\*")) {

return "Insira uma senha com pelo menos 8 dígitos, letra maiúscula, caractere especial e números!";

}

// Remoção de qualquer caractere que não seja dígito do número de telefone.

telefone = telefone.replaceAll("[^\\d]", "");

// Valida se o telefone tem 11 dígitos, contando DDD + números.

if (telefone.length() != 11) {

return "Insira um número de telefone válido";

}

// Mensagem de erro não será retornada se os dados forem válidos.

return null;

}

}

### **2.5.5 Classe ValidadorDadosUsuarioLogin**

ValidadorDadosUsuarioLogin é responsável pela validação dos dados para autenticação de login, se utiliza do polimorfismo, sobreescrevendo e implementando o método validar() da classe mãe, realiza encapsulamento para proteger os atributos.

Segue código abaixo.

package com.seren.util;

// Classe utilitária responsável por validar os dados antes da autenticação para a controller de Login.

// Implementa a interface Validador.

public class ValidadorDadosUsuarioLogin implements Validador {

// Atributos essenciais para a autenticação.

private String email;

private String senha;

// Construtor para inicializar os dados a serem validados.

public ValidadorDadosUsuarioLogin(String email, String senha) {

this.email = email;

this.senha = senha;

}

// Implementação do método validar da interface, utilizando sobrescrita, realizando polimorfismo.

@Override

public String validar() {

// Verificação se os dados estão vazios, retornando uma String que será utilizada como mensagem de erro.

if (email == null || email.isEmpty() || senha == null || senha.isEmpty()) {

return "Preencha todos os campos.";

}

// Verificação se o email possui "@".

if (!email.contains("@")) {

return "Insira um email válido!";

}

// Mensagem de erro não será retornada se os dados forem válidos.

return null;

}

}

### **2.5.6 Classe ValidadorDadosPaciente**

ValidadorDadosPaciente valida campos essenciais do paciente, se utiliza do polimorfismo, sobreescrevendo e implementando o método validar() da classe mãe, realiza encapsulamento para proteger os atributos.

Segue código abaixo.

package com.seren.util;

import java.util.Date;

// Classe utilitária responsável por validar os dados antes da criação de um objeto do tipo Paciente.

// Implementa a interface Validador.

public class ValidadorDadosPaciente implements Validador {

// Atributos essenciais para o construtor de Paciente.

private String nome;

private String profissao;

private String genero;

private String estadoCivil;

private String telefone;

private String email;

private String preferenciaContato;

private Date dataNascimento;

// Construtor para inicializar os dados a serem validados.

public ValidadorDadosPaciente(String nome, String profissao, String genero, String estadoCivil, String telefone, String email, String preferenciaContato, Date dataNascimento) {

this.nome = nome;

this.profissao = profissao;

this.genero = genero;

this.estadoCivil = estadoCivil;

this.telefone = telefone;

this.email = email;

this.preferenciaContato = preferenciaContato;

this.dataNascimento = dataNascimento;

}

// Implementação do método validar da interface, utilizando sobrescrita, realizando polimorfismo.

@Override

public String validar() {

// Verificação se os dados estão vazios, retornando uma String que será utilizada como mensagem de erro.

if (nome.isEmpty() || profissao.isEmpty() || genero.isEmpty() || estadoCivil.isEmpty()

|| telefone.isEmpty() || email.isEmpty() || preferenciaContato.isEmpty() || dataNascimento == null) {

return "Preencha todos os campos.";

}

// Verificação se o email possui "@".

if (!email.contains("@")) {

return "Insira um email válido!";

}

// Verificação se o nome contém números.

if (nome.matches(".\*\\d.\*")) {

return "Insira um nome que não contenha dígitos.";

}

// Remoção de qualquer caractere que não seja dígito do número de telefone.

telefone = telefone.replaceAll("[^\\d]", "");

// Valida se o telefone tem 11 dígitos, contando DDD + números.

if (telefone.length() != 11) {

return "O número de telefone deve conter 11 dígitos.";

}

// Idade retornada com o suporte da classe DataUtil e o seu método calcularIdade.

int idade = DataUtil.calcularIdade(dataNascimento);

// Se a idade for menor de 18 retornará a devida String de mensagem de erro.

if (idade < 18) {

return "Paciente deve ter pelo menos 18 anos.";

}

// Mensagem de erro não será retornada se os dados forem válidos.

return null;

}

// Retorna o número em formato numérico (long), removendo qualquer formatação.

public long getTelefone() {

String telefoneNumero = telefone.replaceAll("[^\\d]", "");

return Long.parseLong(telefoneNumero);

}

}

### **2.5.7 Classe DataUtil**

A classe DataUtil é uma utilitária focada no cálculo da idade a partir da data de nascimento fornecida. Ela segue o princípio do encapsulamento, mantendo seu método estático para uso global sem necessidade de instanciar objetos, facilitando a reutilização.

Segue código abaixo.

package com.seren.util;

import java.time.LocalDate;

import java.time.Period;

import java.time.ZoneId;

import java.util.Date;

// Classe utilitária para cálculo de idade através da data de nascimento.

public class DataUtil {

// Método que retorna a idade calculada ondê o parâmetro é a data de nascimento.

public static int calcularIdade(Date dataNascimento) {

// Verificação para caso a data venha vazia, retornando 0.

if (dataNascimento == null) {

return 0;

}

// Converte a data para data local considerando o fuso horário do sistema.

LocalDate nascimento = dataNascimento.toInstant()

.atZone(ZoneId.systemDefault())

.toLocalDate();

// Obtém a data atual/hoje e armazena em um objeto do tipo data local.

LocalDate hoje = LocalDate.now();

// Retorna o período entre a data de nascimento e hoje, de modo que seja em ano.

return Period.between(nascimento, hoje).getYears();

}

}

### **2.5.8 Classe LembrarEmailUtil**

A classe LembrarEmailUtil, é uma utilitária que gerencia o armazenamento e recuperação do email do usuário para a funcionalidade "lembrar de mim". Utiliza métodos estáticos, aplica encapsulamento ao esconder a manipulação direta do arquivo .properties e expor apenas métodos públicos claros para salvar, obter, excluir e verificar o email salvo.

Segue código abaixo.

package com.seren.util;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

import java.util.Properties;

// Classe utilitária responsável por guardar e recuperar o email do usuário.

// Isso é usado para que a funcionalidade "lembrar de mim" seja concretizada.

public class LembrarEmailUtil {

// Caminho do arquivo onde as propriedades vão ser salvas.

private static final String localArquivo = "lembrarEmail.properties";

// Objeto Properties utilizado para armazenar as propriedades carregadas.

private static final Properties props = new Properties();

// Método que carrega o conteúdo do arquivo para a memória.

// Antes de carregar, ele limpa qualquer informação anterior que possa estar guardada.

private static void carregarArquivo() {

props.clear(); // Limpa qualquer dado anterior que esteja na memória.

// Cria um objeto "arquivo" apontando para o caminho definido acima.

File arquivo = new File(localArquivo);

// Verifica se o arquivo existe. Se existir, tenta abrir e carregar seu conteúdo.

// Uso do try e catch por conta de estarmos trabalhando com arquivo.

if (arquivo.exists()) {

try (FileInputStream input = new FileInputStream(arquivo)) {

// Carrega os dados do arquivo para o objeto props.

props.load(input);

} catch (IOException e) {

e.printStackTrace();

}

}

}

// Método que salva o email informado no arquivo .properties.

// Também marca que o email deve ser lembrado, salvando "lembrarEmail" como "true".

public static void salvarEmail(String email) {

props.setProperty("lembrarEmail", "true");

props.setProperty("email", email);

// Grava as informações no arquivo juntamente com um comentário.

try (FileOutputStream emailGuardado = new FileOutputStream(localArquivo)) {

props.store(emailGuardado, "EmailGuardado");

} catch (IOException e) {

e.printStackTrace();

}

}

// Método que retorna o email guardado, mas só se a opção "lembrarEmail" estiver marcada como "true".

public static String getEmailGuardado() {

// Garante que o arquivo foi carregado na memória antes de buscar.

carregarArquivo();

if ("true".equals(props.getProperty("lembrarEmail"))) {

return props.getProperty("email", "");

}

return "";

}

// Método que apaga o arquivo de propriedades.

// Também limpa a memória (o objeto props).

public static void excluirEmail() {

File arquivo = new File(localArquivo);

if (arquivo.exists()) {

arquivo.delete();

}

props.clear();

}

// Verifica se existe um email salvo e se a opção de lembrar está ativada.

public static boolean temEmailGuardado() {

// Garante que os dados estejam carregados.

carregarArquivo();

String lembrar = props.getProperty("lembrarEmail");

String email = props.getProperty("email");

// Retorna true se o "lembrarEmail" for true e o email não estiver vazio.

return "true".equalsIgnoreCase(lembrar) && email != null && !email.isEmpty();

}

}

## **2.6 View**

Views são todas as interfaces gráficas do sistema, essencial para interação com o usuário, seja visualizando o calendário ou inserindo dados para cadastro de um paciente. Neste projeto, foi utilizada a biblioteca Java Swing para construir essas telas, tendo uso principalmente com os componentes JFrame e JinternalFrame, criando janelas principais(Login, Cadastro, TelaInicial) e internas(TelaCadastrarPaciente, TelaEditarPaciente, TelaPacientes).

### **2.6.1 Classe Login**

A classe Login herda de JFrame e representa a interface gráfica de autenticação do sistema. Ela segue o padrão arquitetural MVC (Model-View-Controller), delegando a lógica de login ao LoginController, enquanto se responsabiliza apenas pela interface com o usuário. Utiliza composição para integrar componentes personalizados, como RoundedJTextField e RoundedJPasswordField, que tornam a experiência visual mais amigável. Também faz uso de validação de entrada por meio da classe ValidadorDadosUsuarioLogin e permite lembrar o e-mail do usuário utilizando LembrarEmailUtil. Este exemplo ilustra os princípios de separação de responsabilidades, reutilização de código e controle de eventos com Java Swing.

Segue o código da classe com comentários explicativos::

package com.seren.view;

import com.seren.controller.LoginController;

import com.seren.model.Usuario;

import com.seren.util.ValidadorDadosUsuarioLogin;

import javax.swing.ImageIcon;

import javax.swing.JOptionPane;

import com.seren.util.LembrarEmailUtil;

/\*\*

\* Classe login que contempla ao usuário o acesso a sua conta

\*/

public class Login extends javax.swing.JFrame {

// Composição: instância do controller responsável pela lógica de autenticação

private final LoginController controller = new LoginController();

// Construtor da interface Login

public Login() {

initComponents(); // Inicializa os componentes da interface gráfica

// Define o ícone da janela usando recurso da pasta "images"

setIconImage(new ImageIcon(getClass().getClassLoader().getResource("images/iconLogo.png")).getImage());

setResizable(false); // Impede redimensionamento da janela

// Verifica se há e-mail salvo e preenche automaticamente o campo

if (LembrarEmailUtil.temEmailGuardado()) {

emailInputLogin.setText(LembrarEmailUtil.getEmailGuardado());

jCheckBox1.setSelected(true);

} else {

emailInputLogin.setText("");

jCheckBox1.setSelected(false);

}

}

// Método chamado ao clicar no botão de cadastro

// Cria e exibe a tela de cadastro, ocultando a tela de login

private void cadastroButton1ActionPerformed(java.awt.event.ActionEvent evt) {

Cadastro cadastro = new Cadastro();

cadastro.setVisible(true);

setVisible(false);

}

// Método chamado ao clicar no botão de login

private void loginButtonActionPerformed(java.awt.event.ActionEvent evt) {

// Captura os dados dos campos de entrada

String email = emailInputLogin.getText();

String senha = new String(senhaInputLogin.getPassword());

// Validação básica usando classe utilitária

ValidadorDadosUsuarioLogin validador = new ValidadorDadosUsuarioLogin(email, senha);

String erro = validador.validar();

if (erro != null) {

JOptionPane.showMessageDialog(null, erro);

return;

}

// Tenta autenticar o usuário

Usuario usuarioLogado = controller.loginUsuario(email, senha);

if (usuarioLogado != null) {

// Se a opção "lembrar e-mail" estiver marcada, salva o e-mail

if (jCheckBox1.isSelected()) {

LembrarEmailUtil.salvarEmail(email);

} else {

LembrarEmailUtil.excluirEmail();

}

JOptionPane.showMessageDialog(null, "Login com sucesso!");

// Abre a tela inicial e fecha a de login

boolean lembrar = jCheckBox1.isSelected();

TelaInicial paginaInicial = new TelaInicial(usuarioLogado, lembrar);

paginaInicial.setVisible(true);

setVisible(false);

} else {

JOptionPane.showMessageDialog(null, "Credenciais inválidas.");

}

// Limpa os campos após tentativa de login

emailInputLogin.setText("");

senhaInputLogin.setText("");

}

// Método chamado ao interagir com a checkbox de "lembrar e-mail"

private void jCheckBox1ActionPerformed(java.awt.event.ActionEvent evt) {

// Se desmarcado, remove o e-mail salvo

if (!jCheckBox1.isSelected()) {

LembrarEmailUtil.excluirEmail();

}

}

// @SuppressWarnings("unchecked")

// A construção da interface com todos os componentes visuais (como botões, labels, inputs etc.)

// foi omitida por se tratar de código gerado automaticamente pelo editor visual do NetBeans (GUI Builder).

// O método main foi omitido nesta documentação por conter apenas a inicialização

}

### **2.6.2 Classe Cadastro**

A classe Cadastro estende JFrame e representa a interface gráfica destinada ao registro de novos usuários no sistema. Seguindo o padrão MVC, ela utiliza o CadastroController para processar a lógica de criação de usuários, mantendo a camada visual isolada. A validação de dados é feita por classes utilitárias que garantem consistência nas entradas fornecidas. A classe também demonstra o uso de eventos da interface gráfica para coordenar ações como cliques e submissões de formulário. Este exemplo reforça boas práticas de modularização, separação de responsabilidades e encapsulamento.

Segue o código da classe com comentários explicativos:

package com.seren.view;

import com.seren.controller.CadastroController;

import com.seren.util.ValidadorDadosUsuario;

import javax.swing.ImageIcon;

import javax.swing.JOptionPane;

/\*\*

\* Classe Cadastro que compõe a tela na qual o usuário poderá se cadastrar

\*/

public class Cadastro extends javax.swing.JFrame {

// Composição: Controller responsável pela lógica de cadastro

private final CadastroController controller = new CadastroController();

// Construtor da tela de cadastro

public Cadastro() {

initComponents(); // Inicializa os componentes visuais

// Define ícone da aplicação a partir de um recurso da pasta "images"

setIconImage(new ImageIcon(getClass().getClassLoader().getResource("images/iconLogo.png")).getImage());

setResizable(false); // Impede que a janela seja redimensionada

}

// Evento disparado ao clicar no botão de "Cadastrar"

private void cadastroButtonActionPerformed(java.awt.event.ActionEvent evt) {

// Captura os dados dos campos de entrada

String username = usernameInputCadastro.getText();

String email = emailInputCadastro.getText();

String telefoneFormatado = telefoneInputCadastro.getText();

String senha = new String(senhaInputCadastro.getPassword());

String senhaConfirmar = new String(senhaConfirmarInputCadastro.getPassword());

// Remove caracteres não numéricos do telefone

String telefone = telefoneFormatado.replaceAll("[^\\d]", "");

// Valida os dados utilizando classe utilitária

ValidadorDadosUsuario validador = new ValidadorDadosUsuario(username, email, telefone, senha, senhaConfirmar);

String erro = validador.validar();

if (erro != null) {

// Exibe erro de validação, se houver

JOptionPane.showMessageDialog(null, erro);

return;

}

// Tenta cadastrar o usuário via controller

boolean sucesso = controller.cadastrarUsuario(username, email, telefone, senha);

if (sucesso) {

// Se o cadastro for bem-sucedido, mostra mensagem e abre tela de login

JOptionPane.showMessageDialog(null, "Usuário cadastrado com sucesso!");

Login login = new Login();

login.setVisible(true);

setVisible(false);

} else {

// Caso o usuário já esteja cadastrado

JOptionPane.showMessageDialog(null, "Usuário já foi cadastrado anteriormente.");

}

// Limpa os campos após a tentativa

usernameInputCadastro.setText("");

emailInputCadastro.setText("");

telefoneInputCadastro.setText("");

senhaInputCadastro.setText("");

senhaConfirmarInputCadastro.setText("");

}

// Evento disparado ao clicar no botão "Já tenho login"

private void LoginButton1ActionPerformed(java.awt.event.ActionEvent evt) {

// Abre a tela de login e fecha a tela de cadastro

Login login = new Login();

login.setVisible(true);

setVisible(false);

}

// @SuppressWarnings("unchecked")

// A construção da interface com todos os componentes visuais (como botões, labels, inputs etc.)

// foi omitida por se tratar de código gerado automaticamente pelo editor visual do NetBeans (GUI Builder).

// O método main foi omitido nesta documentação por conter apenas a inicialização

}

### **2.6.3 Classe Tela Inicial**

A classe TelaInicial representa a primeira interface exibida ao usuário após a autenticação bem-sucedida. Sua principal função é exibir uma saudação dinâmica conforme o horário do dia e apresentar informações gerais, como o número total de pacientes vinculados ao usuário logado. Utiliza controladores como BuscarPacientesController e classes de utilidade como LembrarEmailUtil para organizar os dados exibidos. Este exemplo evidencia a separação entre lógica de negócio e interface visual, além de demonstrar o uso de listeners para eventos e interação entre camadas no padrão MVC.

Segue o código da classe com comentários explicativos:

package com.seren.view;

import com.seren.model.Usuario; // Representa o usuário logado

import javax.swing.ImageIcon;

import com.seren.components.RoundedPanel; // Provavelmente um painel com bordas arredondadas

import com.seren.controller.BuscarPacientesController; // Controller que interage com os dados de pacientes

import javax.swing.JOptionPane;

import com.seren.util.LembrarEmailUtil; // Utilitário para lembrar ou esquecer o e-mail do usuário

import java.util.ArrayList;

import org.bson.Document; // Utilizado para representar documentos do MongoDB

public class TelaInicial extends javax.swing.JFrame {

private TelaPacientes telaPacientes; // Referência à tela de pacientes

private Usuario usuario; // Armazena o usuário logado

private boolean lembrarEmail; // Define se o e-mail deve ser lembrado

// Construtor padrão (sem argumentos)

public TelaInicial() {

initComponents(); // Inicializa os componentes visuais

setIconImage(new ImageIcon(getClass().getClassLoader().getResource("images/iconLogo.png")).getImage()); // Define ícone

setResizable(false); // Impede redimensionamento da janela

}

// Construtor com argumentos (usuário logado e preferência de lembrar e-mail)

public TelaInicial(Usuario usuarioLogado, boolean lembrarEmail) {

initComponents();

this.usuario = usuarioLogado;

this.lembrarEmail = lembrarEmail;

exibirSaudacao(); // Mostra saudação personalizada

totalPacientesCadastrados(); // Exibe total de pacientes cadastrados

// Evento de clique para recarregar os dados da tela

labelSerengotoInitialScreen.addMouseListener(new java.awt.event.MouseAdapter() {

public void mouseClicked(java.awt.event.MouseEvent evt) {

if (telaPacientes != null && telaPacientes.isVisible()) {

try {

telaPacientes.setClosed(true); // Fecha a tela se estiver aberta

telaPacientes = null;

totalPacientesCadastrados(); // Atualiza o total

} catch (java.beans.PropertyVetoException e) {

e.printStackTrace();

}

}

}

});

labelSerengotoInitialScreen.setCursor(new java.awt.Cursor(java.awt.Cursor.HAND\_CURSOR)); // Altera o cursor

// Evento de clique para sair e retornar à tela de login

labelSairgotoLogin.addMouseListener(new java.awt.event.MouseAdapter() {

public void mouseClicked(java.awt.event.MouseEvent evt) {

if (!lembrarEmail) {

LembrarEmailUtil.excluirEmail(); // Remove o e-mail salvo se necessário

}

Login login = new Login(); // Abre a tela de login novamente

JOptionPane.showMessageDialog(null, "Sessão encerrada!");

login.setVisible(true);

setVisible(false); // Oculta a tela atual

}

});

labelSairgotoLogin.setCursor(new java.awt.Cursor(java.awt.Cursor.HAND\_CURSOR));

// Evento para abrir a tela de pacientes

pacientesButton.addActionListener(new java.awt.event.ActionListener() {

public void actionPerformed(java.awt.event.ActionEvent evt) {

pacientesButtonActionPerformed(evt);

}

});

}

// Exibe uma saudação conforme o horário do dia

private void exibirSaudacao() {

String saudacao;

int hora = java.time.LocalTime.now().getHour();

if (hora >= 5 && hora < 12) {

saudacao = "Bom dia";

} else if (hora >= 12 && hora < 18) {

saudacao = "Boa tarde";

} else {

saudacao = "Boa noite";

}

labelApresentarUsername.setText(saudacao + ", " + usuario.getNome()); // Saudação com o nome do usuário

}

// Método chamado ao clicar no botão "Pacientes"

private void pacientesButtonActionPerformed(java.awt.event.ActionEvent evt) {

if (telaPacientes == null || telaPacientes.isClosed()) {

telaPacientes = new TelaPacientes(this.usuario); // Cria nova tela de pacientes

jDesktopPane1.add(telaPacientes); // Adiciona ao desktop pane

telaPacientes.setVisible(true);

setIconImage(new ImageIcon(getClass().getClassLoader().getResource("images/iconLogo.png")).getImage());

setResizable(false);

}

}

// Atualiza a label com o total de pacientes cadastrados

private void totalPacientesCadastrados() {

BuscarPacientesController controller = new BuscarPacientesController(); // Controller que busca os pacientes

ArrayList<Document> pacientes = controller.buscarPacientes(usuario.getId()); // Busca usando ID do usuário

labelTotalPacientes.setText(pacientes.size() + " pacientes cadastrados"); // Atualiza o texto na tela

}

}

// @SuppressWarnings("unchecked")

// A construção da interface com todos os componentes visuais (como botões, labels, inputs etc.)

// foi omitida por se tratar de código gerado automaticamente pelo editor visual do NetBeans (GUI Builder).

// O método main foi omitido nesta documentação por conter apenas a inicialização

### **2.6.4 Classe Tela Pacientes**

A classe TelaPacientes é uma JInternalFrame que permite ao usuário visualizar, buscar, editar e excluir pacientes cadastrados no sistema. Ela interage com diferentes controladores, como BuscarPacienteController e ExcluirPacienteController, para executar operações de CRUD com os dados armazenados no banco MongoDB. A interface exibe os dados em uma JTable, atualizada dinamicamente conforme as ações do usuário. Esta implementação demonstra conceitos como coesão, reaproveitamento de lógica com métodos utilitários (como carregarPacientesNaTabela()) e separação de responsabilidades conforme o padrão MVC.

Segue o código da classe com comentários explicativos:

package com.seren.view;

import com.seren.controller.BuscarPacienteController;

import com.seren.controller.BuscarPacientesController;

import com.seren.controller.ExcluirPacienteController;

import com.seren.model.Paciente;

import com.seren.model.Usuario;

import com.seren.util.DataUtil;

import java.awt.Font;

import java.awt.Image;

import java.awt.event.MouseAdapter;

import java.awt.event.MouseEvent;

import java.util.ArrayList;

import javax.swing.plaf.basic.BasicInternalFrameUI;

import java.util.Date;

import javax.swing.ImageIcon;

import javax.swing.JMenuItem;

import javax.swing.JOptionPane;

import javax.swing.JPopupMenu;

import javax.swing.table.DefaultTableModel;

import javax.swing.table.JTableHeader;

import org.bson.Document;

import org.bson.types.ObjectId;

public class TelaPacientes extends javax.swing.JInternalFrame {

public TelaPacientes() {

initComponents();

setResizable(false);

}

private Usuario usuario;

// Instâncias dos controladores responsáveis por operações com os pacientes

BuscarPacientesController controller = new BuscarPacientesController();

BuscarPacientesController buscarPacientesController = new BuscarPacientesController();

BuscarPacienteController buscarPacienteController = new BuscarPacienteController();

ExcluirPacienteController excluirPacienteController = new ExcluirPacienteController();

// Lista para armazenar os pacientes retornados do banco

private ArrayList<Document> listaPacientes = new ArrayList<>();

// Menu de contexto com opções de editar/excluir

private JPopupMenu menuPopup;

private JMenuItem menuEditar;

private JMenuItem menuExcluir;

public TelaPacientes(Usuario usuarioLogado) {

initComponents();

this.usuario = usuarioLogado;

exibirInfoUsuario();

// Remove a borda superior do JInternalFrame

this.setBorder(null);

BasicInternalFrameUI ui = (BasicInternalFrameUI) this.getUI();

ui.setNorthPane(null);

criarMenuPopup(); // Cria o menu de clique direito com opções

adicionarListenerTabela(); // Adiciona escuta para clique na tabela

carregarPacientesNaTabela(); // Carrega os pacientes automaticamente

// Escuta de eventos para campo de busca: limpa e recarrega tudo ao apagar

buscarInputBuscarPaciente.addKeyListener(new java.awt.event.KeyAdapter() {

@Override

public void keyReleased(java.awt.event.KeyEvent e) {

String textoBusca = buscarInputBuscarPaciente.getText().trim();

if (textoBusca.isEmpty()) {

carregarPacientesNaTabela(); // Recarrega todos

}

}

});

}

// Método sobrecarregado para buscar todos os pacientes ou filtrar por nome

public void carregarPacientesNaTabela() {

carregarPacientesNaTabela(null);

}

public void carregarPacientesNaTabela(String filtroNome) {

DefaultTableModel modelo = new DefaultTableModel();

modelo.addColumn("Nome");

modelo.addColumn("Idade");

modelo.addColumn("Email");

// Decide se filtra ou busca todos

if (filtroNome == null || filtroNome.isEmpty()) {

listaPacientes = buscarPacientesController.buscarPacientes(usuario.getId());

} else {

listaPacientes = buscarPacienteController.buscarPacientes(usuario.getId(), filtroNome);

}

// Preenche a tabela com os dados

if (!listaPacientes.isEmpty()) {

for (Document doc : listaPacientes) {

String nomePaciente = doc.getString("nome");

Date dataNascimento = doc.getDate("dataNascimento");

int idade = DataUtil.calcularIdade(dataNascimento);

String email = doc.getString("email");

modelo.addRow(new Object[]{nomePaciente, idade, email});

}

} else {

modelo.addRow(new Object[]{"Nenhum paciente encontrado", "", ""});

}

tabelaPacientes.setModel(modelo);

// Ajusta o cabeçalho da tabela

JTableHeader cabecalho = tabelaPacientes.getTableHeader();

cabecalho.setFont(new Font("Segoe UI Ligth", Font.BOLD, 18));

}

// Exibe o nome e e-mail do usuário logado

private void exibirInfoUsuario() {

labelNomeUsuario.setText(usuario.getNome());

labelEmailUsuario.setText(usuario.getEmail());

}

// Cria o menu popup com as opções "Editar" e "Excluir"

private void criarMenuPopup() {

menuPopup = new JPopupMenu();

menuEditar = new JMenuItem("Editar");

menuExcluir = new JMenuItem("Excluir");

menuPopup.add(menuEditar);

menuPopup.add(menuExcluir);

menuEditar.addActionListener(e -> editarPacienteSelecionado());

menuExcluir.addActionListener(e -> excluirPacienteSelecionado());

}

// Adiciona os listeners de mouse para o menu popup

private void adicionarListenerTabela() {

tabelaPacientes.addMouseListener(new MouseAdapter() {

@Override

public void mousePressed(MouseEvent e) {

if (e.isPopupTrigger()) {

mostrarMenuPopup(e);

}

}

@Override

public void mouseReleased(MouseEvent e) {

if (e.isPopupTrigger()) {

mostrarMenuPopup(e);

}

}

});

}

// Exibe o menu popup na linha selecionada

private void mostrarMenuPopup(MouseEvent e) {

int row = tabelaPacientes.rowAtPoint(e.getPoint());

if (row >= 0 && row < listaPacientes.size()) {

tabelaPacientes.setRowSelectionInterval(row, row);

menuPopup.show(e.getComponent(), e.getX(), e.getY());

}

}

// Abre a tela de edição com os dados do paciente selecionado

private void editarPacienteSelecionado() {

int row = tabelaPacientes.getSelectedRow();

if (row >= 0 && row < listaPacientes.size()) {

Document docSelecionado = listaPacientes.get(row);

Paciente paciente = Paciente.pacienteDocumento(docSelecionado);

TelaEditarPaciente telaEditar = new TelaEditarPaciente(this.usuario, paciente, this);

jDesktopPane2.add(telaEditar);

telaEditar.setVisible(true);

telaEditar.centralizar();

}

}

// Solicita confirmação e exclui o paciente selecionado

private void excluirPacienteSelecionado() {

int row = tabelaPacientes.getSelectedRow();

if (row >= 0 && row < listaPacientes.size()) {

Document docSelecionado = listaPacientes.get(row);

Paciente paciente = Paciente.pacienteDocumento(docSelecionado);

int resposta = JOptionPane.showConfirmDialog(

null,

"Deseja excluir o paciente " + paciente.getNome() + " ?",

"Confirmar exclusão",

JOptionPane.YES\_NO\_OPTION

);

if (resposta == JOptionPane.YES\_OPTION) {

boolean sucesso = excluirPacienteController.excluirPaciente(paciente.getId());

if (sucesso) {

JOptionPane.showMessageDialog(null, "Paciente excluído com sucesso.");

carregarPacientesNaTabela(); // Atualiza a lista

} else {

JOptionPane.showMessageDialog(null, "Erro ao excluir o paciente.");

}

}

}

}

// Ação do botão "Cadastrar novo paciente"

private void btnCadastrarPacienteActionPerformed(java.awt.event.ActionEvent evt) {

TelaCadastrarPaciente cadastrarPaciente = new TelaCadastrarPaciente(this.usuario, this);

jDesktopPane2.add(cadastrarPaciente);

cadastrarPaciente.setVisible(true);

cadastrarPaciente.centralizar();

}

// Ação do botão de buscar paciente pelo nome

private void btnBuscarPacienteNomeActionPerformed(java.awt.event.ActionEvent evt) {

String nome = buscarInputBuscarPaciente.getText();

if (nome.isEmpty()) {

JOptionPane.showMessageDialog(null, "Campo vazio, insira o nome do paciente.");

carregarPacientesNaTabela();

return;

}

carregarPacientesNaTabela(nome);

}

}

// @SuppressWarnings("unchecked")

// A construção da interface com todos os componentes visuais (como botões, labels, inputs etc.)

// foi omitida por se tratar de código gerado automaticamente pelo editor visual do NetBeans (GUI Builder).

// O método main foi omitido nesta documentação por conter apenas a inicialização.

### **2.6.5 Classe Tela Cadastrar Pacientes**

A classe TelaCadastrarPaciente é uma JInternalFrame que permite o cadastro de novos pacientes no sistema. Ela segue o padrão MVC ao isolar a lógica de negócio no CriarPacienteController e delegar a validação dos dados ao ValidadorDadosPaciente. A classe lida apenas com a coleta de dados via interface gráfica e manipulação de eventos como cliques em botões. Este exemplo ilustra a aplicação do princípio de responsabilidade única e a separação entre interface e regras de negócio, facilitando a manutenção e evolução do código.

Segue o código da classe com comentários explicativos:

package com.seren.view;

import com.seren.controller.CriarPacienteController;

import com.seren.model.Usuario;

import com.seren.util.ValidadorDadosPaciente;

import java.util.Date;

import javax.swing.JOptionPane;

import javax.swing.plaf.basic.BasicInternalFrameUI;

import org.bson.types.ObjectId;

public class TelaCadastrarPaciente extends javax.swing.JInternalFrame {

// Instancia o controller responsável por cadastrar pacientes

private CriarPacienteController controller = new CriarPacienteController();

private Usuario usuario;

private TelaPacientes telaPacientes;

// Construtor padrão da tela

public TelaCadastrarPaciente() {

initComponents();

setResizable(false);

}

// Construtor com referência ao usuário logado e à tela de pacientes

public TelaCadastrarPaciente(Usuario usuarioLogado, TelaPacientes telaPacientes) {

initComponents();

this.usuario = usuarioLogado;

this.telaPacientes = telaPacientes;

// Remove a barra superior padrão da janela interna

BasicInternalFrameUI ui = (BasicInternalFrameUI) this.getUI();

ui.setNorthPane(null);

}

// Ação executada ao clicar no botão de cadastro

private void cadastrarPacientesButtonActionPerformed(java.awt.event.ActionEvent evt) {

// Coleta os dados preenchidos na interface

String nome = nomeInputCadastrarPaciente.getText();

String profissao = profissaoInputCadastrarPaciente.getText();

String genero = generoInputCadastrarPaciente.getText();

String estadoCivil = estadoCivilInputCadastrarPaciente.getText();

String telefoneFormatado = telefoneInputCadastrarPaciente.getText();

String email = emailInputCadastrarPaciente.getText();

String preferenciaContato = preferenciaContatoInputCadastrarPaciente.getText();

Date dataNascimento = dataNascimentoInputCadastrarPaciente.getDate();

ObjectId id\_usuario = usuario.getId();

// Remove formatações do telefone

String telefone = telefoneFormatado.replaceAll("[^\\d]", "");

// Valida os dados usando uma classe utilitária

ValidadorDadosPaciente validador = new ValidadorDadosPaciente(

nome, profissao, genero, estadoCivil, telefone, email, preferenciaContato, dataNascimento

);

String erro = validador.validar();

// Mostra erro, caso exista

if (erro != null) {

JOptionPane.showMessageDialog(null, erro);

return;

}

// Tenta cadastrar o paciente

boolean sucesso = controller.criarPaciente(

nome, profissao, genero, estadoCivil, telefone, email, preferenciaContato, dataNascimento, id\_usuario

);

// Mensagem de sucesso ou falha

if (sucesso) {

JOptionPane.showMessageDialog(null, "Paciente cadastrado com sucesso!");

setVisible(false);

telaPacientes.carregarPacientesNaTabela();

} else {

JOptionPane.showMessageDialog(null, "Paciente já cadastrado anteriormente");

}

}

// Botão de cancelar

private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {

setVisible(false);

}

// Método para centralizar a janela na tela principal

public void centralizar() {

if (getParent() != null) {

this.setLocation(

(getParent().getWidth() - this.getWidth()) / 2,

(getParent().getHeight() - this.getHeight()) / 2

);

}

}

}

// @SuppressWarnings("unchecked")

// A construção da interface com todos os componentes visuais (como botões, labels, inputs etc.)

// foi omitida por se tratar de código gerado automaticamente pelo editor visual do NetBeans (GUI Builder).

// O método main foi omitido nesta documentação por conter apenas a inicialização

### **2.6.6 Classe Tela Editar pacientes**

A classe TelaEditarPaciente permite a edição dos dados de um paciente previamente cadastrado. Ela reutiliza dados existentes ao preencher os campos da interface com informações já salvas e utiliza o AlterarPacienteController para aplicar as alterações. A validação é feita por meio da classe ValidadorDadosPaciente, assegurando que apenas dados válidos sejam persistidos. A lógica de apresentação permanece separada da lógica de negócio, seguindo os princípios do padrão MVC e promovendo modularização e reutilização de componentes.

Segue o código da classe com comentários explicativos:

package com.seren.view;

import com.seren.controller.AlterarPacienteController;

import com.seren.model.Paciente;

import com.seren.model.Usuario;

import com.seren.util.ValidadorDadosPaciente;

import java.util.Date;

import javax.swing.JOptionPane;

import javax.swing.plaf.basic.BasicInternalFrameUI;

public class TelaEditarPaciente extends javax.swing.JInternalFrame {

private AlterarPacienteController controller = new AlterarPacienteController();

private Usuario usuario;

private Paciente paciente;

private TelaPacientes telaPacientes;

// Construtor padrão

public TelaEditarPaciente() {

initComponents();

setResizable(false);

}

// Construtor com paciente selecionado, usuário logado e tela de pacientes

public TelaEditarPaciente(Usuario usuarioLogado, Paciente paciente, TelaPacientes telaPacientes) {

initComponents();

this.usuario = usuarioLogado;

this.paciente = paciente;

this.telaPacientes = telaPacientes;

// Remove a barra superior da janela

BasicInternalFrameUI ui = (BasicInternalFrameUI) this.getUI();

ui.setNorthPane(null);

// Preenche os campos com os dados atuais do paciente

nomeInputEditarPaciente.setText(paciente.getNome());

profissaoInputEditarPaciente.setText(paciente.getProfissao());

generoInputEditarPaciente.setText(paciente.getGenero());

estadoCivilInputEditarPaciente.setText(paciente.getEstadoCivil());

telefoneInputEditarPaciente.setText(String.valueOf(paciente.getTelefone()));

emailInputEditarPaciente.setText(paciente.getEmail());

preferenciaContatoInputEditarPaciente.setText(paciente.getPreferenciaContato());

dataNascimentoInputEditarPaciente.setDate(paciente.getDataNascimento());

}

// Ação de editar paciente

private void btnEditarPacientesActionPerformed(java.awt.event.ActionEvent evt) {

// Coleta os dados atualizados da interface

String nome = nomeInputEditarPaciente.getText();

String profissao = profissaoInputEditarPaciente.getText();

String genero = generoInputEditarPaciente.getText();

String estadoCivil = estadoCivilInputEditarPaciente.getText();

String telefone = telefoneInputEditarPaciente.getText();

String email = emailInputEditarPaciente.getText();

String preferenciaContato = preferenciaContatoInputEditarPaciente.getText();

Date dataNascimento = dataNascimentoInputEditarPaciente.getDate();

// Valida os dados

ValidadorDadosPaciente validador = new ValidadorDadosPaciente(

nome, profissao, genero, estadoCivil, telefone, email, preferenciaContato, dataNascimento

);

String erro = validador.validar();

// Em caso de erro, interrompe

if (erro != null) {

JOptionPane.showMessageDialog(null, erro);

return;

}

long telefoneNumero = validador.getTelefone();

// Atualiza os dados no objeto paciente

paciente.setNome(nome);

paciente.setProfissao(profissao);

paciente.setGenero(genero);

paciente.setEstadoCivil(estadoCivil);

paciente.setTelefone(telefoneNumero);

paciente.setEmail(email);

paciente.setPreferenciaContato(preferenciaContato);

paciente.setDataNascimento(dataNascimento);

// Tenta salvar as alterações

boolean sucesso = controller.alterarPaciente(paciente);

if (sucesso) {

JOptionPane.showMessageDialog(null, "Paciente alterado com sucesso!");

telaPacientes.carregarPacientesNaTabela();

setVisible(false);

} else {

JOptionPane.showMessageDialog(null, "Insira dados novos.");

}

}

// Botão cancelar edição

private void btnCancelarEdicaoActionPerformed(java.awt.event.ActionEvent evt) {

setVisible(false);

}

// Centraliza a janela

public void centralizar() {

if (getParent() != null) {

this.setLocation(

(getParent().getWidth() - this.getWidth()) / 2,

(getParent().getHeight() - this.getHeight()) / 2

);

}

}

}

// @SuppressWarnings("unchecked")

// A construção da interface com todos os componentes visuais (como botões, labels, inputs etc.)

// foi omitida por se tratar de código gerado automaticamente pelo editor visual do NetBeans (GUI Builder).

// O método main foi omitido nesta documentação por conter apenas a inicialização

## **2.7 Components**

Components são elementos reutilizáveis da interface gráfica, desenvolvidos para manter a padronização visual e funcional do sistema. Neste projeto, foram criados componentes personalizados estendendo classes do Java Swing, como CustomJCalendar RoudedButton, RoudedPanel, RoudedJFormattedTextField, RoudedJDataChooser, RoudedJPasswordField e RoudedJTextField com ajustes visuais (cores, fontes, bordas arredondadas) e comportamentais.

### **2.7.1 Classe CustomJCalendar**

A classe CustomJCalendar é um componente gráfico personalizado que estende a classe JCalendar da biblioteca *Toedter Calendar*, com o objetivo de adaptar visualmente o calendário utilizado nas interfaces do sistema. Esta customização aplica uma identidade visual própria, com cores, fontes e estilos que seguem a padronização definida para o sistema.

A classe demonstra uso de encapsulamento ao ocultar a lógica de estilização dentro de um método privado (estilizarComponentes) e promove a reutilização ao oferecer um calendário visualmente consistente sempre que for instanciado. Além disso, respeita o princípio da separação de responsabilidades ao isolar a configuração de interface sem afetar a lógica de funcionamento do calendário base.

Segue o código da classe com comentários explicativos:

package com.seren.components;

import com.toedter.calendar.JCalendar;

import com.toedter.calendar.JDayChooser;

import com.toedter.calendar.JMonthChooser;

import com.toedter.calendar.JYearChooser;

import java.awt.\*;

import java.util.Calendar;

import javax.swing.\*;

import javax.swing.border.EmptyBorder;

/\*\*

\* Componente personalizado que estende o JCalendar, aplicando estilo visual

\* consistente com o sistema.

\*/

public class CustomJCalendar extends JCalendar {

public CustomJCalendar() {

super(); // Chama o construtor da superclasse JCalendar

setOpaque(true); // Garante que o fundo do componente seja opaco

estilizarComponentes(); // Encapsulamento: lógica de estilo em método privado

}

/\*\*

\* Método responsável por aplicar estilos visuais personalizados ao calendário.

\* Demonstra separação de responsabilidades e reutilização.

\*/

private void estilizarComponentes() {

// Acesso aos subcomponentes do calendário

JDayChooser dayChooser = getDayChooser();

JMonthChooser monthChooser = getMonthChooser();

JYearChooser yearChooser = getYearChooser();

// Definição de fontes e cores customizadas (identidade visual)

Font font = new Font("Segoe UI", Font.PLAIN, 13);

Color fundo = Color.WHITE;

Color texto = Color.BLUE;

Color hojeFundo = new Color(0x0078D7); // Azul padrão do Windows

Color hojeTexto = Color.WHITE;

// Obtém a data atual

int diaHoje = Calendar.getInstance().get(Calendar.DAY\_OF\_MONTH);

int mesHoje = Calendar.getInstance().get(Calendar.MONTH);

int anoHoje = Calendar.getInstance().get(Calendar.YEAR);

// Aplica o estilo a cada botão que representa um dia no calendário

Component[] dias = dayChooser.getDayPanel().getComponents();

for (Component comp : dias) {

if (comp instanceof JButton btn) {

// Remoção de efeitos padrão do botão

btn.setContentAreaFilled(false);

btn.setFocusPainted(false);

btn.setBorderPainted(false);

btn.setOpaque(true);

// Aplicação de estilo visual personalizado

btn.setBackground(fundo);

btn.setForeground(texto);

btn.setFont(font);

btn.setUI(new javax.swing.plaf.basic.BasicButtonUI());

// Verificação para destacar o dia atual

try {

int dia = Integer.parseInt(btn.getText());

Calendar cal = getCalendar();

int mes = cal.get(Calendar.MONTH);

int ano = cal.get(Calendar.YEAR);

if (dia == diaHoje && mes == mesHoje && ano == anoHoje) {

btn.setBackground(hojeFundo);

btn.setForeground(hojeTexto);

}

} catch (NumberFormatException ignored) {

// Caso o texto do botão não represente um número (ex: célula vazia)

}

}

}

// Personalização dos seletores de mês e ano

monthChooser.setBackground(fundo);

monthChooser.setFont(font);

monthChooser.getComboBox().setBackground(fundo);

monthChooser.getComboBox().setFont(font);

yearChooser.setBackground(fundo);

yearChooser.setFont(font);

yearChooser.getSpinner().setFont(font);

}

/\*\*

\* Sobrescrita do método de pintura do componente.

\* Pode ser utilizado para aplicar renderizações adicionais no futuro.

\*/

@Override

protected void paintComponent(Graphics g) {

super.paintComponent(g);

}

}

### **2.7.2 Classe RoundedButton**

A classe RoundedButton é um componente gráfico personalizado que estende JButton, criado para oferecer uma aparência moderna com cantos arredondados e efeitos visuais ao passar o mouse (hover). O objetivo é substituir o botão padrão do Swing por um visual mais amigável, consistente com o restante da identidade visual do sistema.

Utiliza herança para estender o comportamento de JButton e encapsula a lógica de renderização e alteração de cor em métodos específicos. A classe também implementa eventos de mouse para proporcionar feedback visual ao usuário, aumentando a responsividade da interface. Esses ajustes demonstram o uso de boas práticas como separação de responsabilidades e reaproveitamento de código visual.

Segue código abaixo com comentários explicativos:

package com.seren.components;

import java.awt.\*;

import java.awt.event.\*;

import javax.swing.\*;

/\*\*

\* Componente personalizado que estende JButton e aplica cantos arredondados

\* e efeito visual ao passar o mouse (hover).

\*/

public class RoundedButton extends JButton {

// Encapsulamento: atributos de configuração visual do botão

private int cornerRadius;

private Color originalBackground;

// Construtor que recebe o raio dos cantos arredondados

public RoundedButton(int radius) {

this.cornerRadius = radius;

// Desativa o comportamento visual padrão do JButton

setOpaque(false); // Permite desenhar o fundo manualmente

setContentAreaFilled(false); // Remove preenchimento padrão

setFocusPainted(false); // Remove o foco pontilhado ao clicar/tabular

setBorderPainted(false); // Remove a borda padrão do botão

// Armazena a cor original do botão (útil para hover)

originalBackground = getBackground();

// Encapsulamento: manipulação do estado hover com MouseAdapter

addMouseListener(new MouseAdapter() {

@Override

public void mouseEntered(MouseEvent e) {

// Salva a cor atual e escurece para o efeito de hover

originalBackground = getBackground();

setBackground(darkenColor(originalBackground, 0.8f));

repaint(); // Redesenha o botão com a nova cor

}

@Override

public void mouseExited(MouseEvent e) {

// Restaura a cor original ao retirar o cursor do botão

setBackground(originalBackground);

repaint();

}

});

}

/\*\*

\* Sobrescrita do método paintComponent para desenhar o botão com cantos arredondados.

\*/

@Override

protected void paintComponent(Graphics g) {

Graphics2D g2 = (Graphics2D) g.create();

// Aplicação de anti-aliasing para suavizar as bordas arredondadas

g2.setRenderingHint(RenderingHints.KEY\_ANTIALIASING, RenderingHints.VALUE\_ANTIALIAS\_ON);

// Preenchimento com cor de fundo atual (personalizada ou de hover)

g2.setColor(getBackground());

g2.fillRoundRect(0, 0, getWidth(), getHeight(), cornerRadius, cornerRadius);

g2.dispose(); // Libera os recursos gráficos

// Desenha o conteúdo padrão do botão (texto, ícones)

super.paintComponent(g);

}

/\*\*

\* Método utilitário para escurecer uma cor com base em um fator multiplicador.

\* Demonstrando encapsulamento e reutilização.

\*/

private Color darkenColor(Color color, float factor) {

int r = Math.max((int) (color.getRed() \* factor), 0);

int g = Math.max((int) (color.getGreen() \* factor), 0);

int b = Math.max((int) (color.getBlue() \* factor), 0);

return new Color(r, g, b, color.getAlpha()); // Preserva transparência

}

}

### **2.7.3 Classe RoundedJDateChooser**

A classe RoundedJDateChooser estende JDateChooser (herança) para aplicar cantos arredondados e uma aparência visual mais moderna ao seletor de datas. Utiliza encapsulamento para esconder a lógica de estilização e sobrescreve métodos do Swing para garantir compatibilidade visual mesmo com mudanças de tema.

Segue código abaixo com comentários explicativos:

package com.seren.components;

import com.toedter.calendar.JDateChooser;

import java.awt.\*;

import javax.swing.\*;

import javax.swing.border.EmptyBorder;

/\*\*

\* Componente personalizado que aplica cantos arredondados ao JDateChooser.

\*/

public class RoundedJDateChooser extends JDateChooser {

// Encapsulamento: raio dos cantos como propriedade privada

private int cornerRadius;

// Herança: construtor chama super() e aplica personalização

public RoundedJDateChooser(int cornerRadius) {

super();

this.cornerRadius = cornerRadius;

// Configurações visuais básicas

setOpaque(false);

setBorder(null);

setBackground(Color.WHITE);

// Customiza o editor de texto interno

customizeDateEditor();

}

// Encapsulamento: método privado para ajustar aparência do editor de data

private void customizeDateEditor() {

if (getDateEditor() instanceof JComponent editor) {

editor.setOpaque(false);

editor.setBorder(new EmptyBorder(5, 10, 5, 10)); // Padding interno

}

}

// Sobrescrita: desenha fundo e borda arredondados

@Override

protected void paintComponent(Graphics g) {

super.paintComponent(g); // Mantém ícone de calendário

Graphics2D g2 = (Graphics2D) g.create();

g2.setRenderingHint(RenderingHints.KEY\_ANTIALIASING, RenderingHints.VALUE\_ANTIALIAS\_ON); // Suaviza bordas

g2.setColor(getBackground());

g2.fillRoundRect(0, 0, getWidth(), getHeight(), cornerRadius, cornerRadius); // Fundo arredondado

g2.setColor(getForeground());

g2.drawRoundRect(0, 0, getWidth() - 1, getHeight() - 1, cornerRadius, cornerRadius); // Borda

g2.dispose(); // Libera recursos gráficos

}

// Sobrescrita: mantém personalização mesmo ao trocar o LookAndFeel

@Override

public void updateUI() {

super.updateUI();

customizeDateEditor();

}

}

### **2.7.4 Classe RoundedJFormattedTextField**

A classe RoundedJFormattedTextField herda de JFormattedTextField para aplicar cantos arredondados na interface. Utiliza encapsulamento para controlar o raio dos cantos e sobrescreve métodos gráficos para personalizar a renderização do componente e da borda, contribuindo para a consistência visual da interface.

Segue código abaixo com comentários explicativos:

package com.seren.components;

import java.awt.\*;

import javax.swing.\*;

import javax.swing.JFormattedTextField;

/\*\*

\* Campo de texto formatado com cantos arredondados.

\*/

public class RoundedJFormattedTextField extends JFormattedTextField {

// Encapsulamento: raio dos cantos armazenado como atributo privado

private int cornerRadius;

// Herança: construtor define raio e estilo básico

public RoundedJFormattedTextField(int radius) {

this.cornerRadius = radius;

setOpaque(false); // Impede fundo padrão (permite customização)

}

// Sobrescrita: renderiza fundo arredondado

@Override

protected void paintComponent(Graphics g) {

Graphics2D g2 = (Graphics2D) g.create();

// Suaviza as bordas

g2.setRenderingHint(RenderingHints.KEY\_ANTIALIASING, RenderingHints.VALUE\_ANTIALIAS\_ON);

// Desenha fundo com cantos arredondados

g2.setColor(getBackground());

g2.fillRoundRect(0, 0, getWidth(), getHeight(), cornerRadius, cornerRadius);

g2.dispose(); // Libera recursos gráficos

super.paintComponent(g); // Desenha texto e conteúdo interno

}

// Sobrescrita: renderiza borda arredondada

@Override

protected void paintBorder(Graphics g) {

Graphics2D g2 = (Graphics2D) g.create();

g2.setRenderingHint(RenderingHints.KEY\_ANTIALIASING, RenderingHints.VALUE\_ANTIALIAS\_ON);

// Desenha borda com cantos arredondados

g2.setColor(getForeground());

g2.drawRoundRect(0, 0, getWidth() - 1, getHeight() - 1, cornerRadius, cornerRadius);

g2.dispose();

}

}

### **2.7.5 Classe RoundedJPasswordField**

A classe RoundedJPasswordField estende JPasswordField para exibir campos de senha com cantos arredondados. Utiliza encapsulamento para controlar o raio dos cantos e sobrescreve métodos de pintura para customizar o fundo e a borda, melhorando a estética da interface gráfica.

Segue código abaixo com comentários explicativos:

package com.seren.components;

import java.awt.\*;

import javax.swing.\*;

/\*\*

\* Campo de senha com cantos arredondados.

\*/

public class RoundedJPasswordField extends JPasswordField {

// Encapsulamento: controla o raio dos cantos

private int cornerRadius;

// Herança: construtor define configurações iniciais

public RoundedJPasswordField(int radius) {

this.cornerRadius = radius;

setOpaque(false); // Desativa fundo padrão para permitir customização

}

// Sobrescrita: desenha fundo com cantos arredondados

@Override

protected void paintComponent(Graphics g){

Graphics2D g2 = (Graphics2D) g.create();

// Antialiasing para suavizar bordas

g2.setRenderingHint(RenderingHints.KEY\_ANTIALIASING, RenderingHints.VALUE\_ANTIALIAS\_ON);

g2.setColor(getBackground());

g2.fillRoundRect(0, 0, getWidth(), getHeight(), cornerRadius, cornerRadius);

g2.dispose(); // Libera recurso gráfico

super.paintComponent(g); // Desenha o conteúdo (texto oculto)

}

// Sobrescrita: desenha borda arredondada

@Override

protected void paintBorder(Graphics g){

Graphics2D g2 = (Graphics2D) g.create();

g2.setRenderingHint(RenderingHints.KEY\_ANTIALIASING, RenderingHints.VALUE\_ANTIALIAS\_ON);

g2.setColor(getForeground());

g2.drawRoundRect(0, 0, getWidth()-1, getHeight()-1, cornerRadius, cornerRadius);

g2.dispose();

}

}

### **2.7.6 Classe RoundedJTextField**

A classe RoundedJTextField estende a classe JTextField da biblioteca Swing, personalizando a aparência de campos de texto com cantos arredondados. Utiliza o encapsulamento para controlar o raio dos cantos por meio de um atributo privado, e aplica herança para reutilizar a funcionalidade base do componente de texto. Para realizar a customização visual, sobrescreve os métodos paintComponent e paintBorder, utilizando técnicas de renderização com Graphics2D e antialiasing. Essa abordagem visa aprimorar a estética da interface gráfica, mantendo a funcionalidade do componente original.

Segue código abaixo com comentários explicativos:

package com.seren.components;

/\*\*

\*

\* Campo de texto com cantos arredondados.

\*/

import java.awt.\*;

import javax.swing.\*;

public class RoundedJTextField extends JTextField {

// Encapsulamento: controla o raio dos cantos do componente

private int cornerRadius;

// Construtor recebe o raio e inicializa o campo, além de desativar a opacidade padrão

public RoundedJTextField(int radius){

this.cornerRadius = radius;

setOpaque(false); // Permite personalizar a pintura do fundo

}

// Sobrescrita: personaliza a pintura do fundo do componente com cantos arredondados

@Override

protected void paintComponent(Graphics g){

Graphics2D g2 = (Graphics2D) g.create();

// Habilita antialiasing para suavizar os cantos

g2.setRenderingHint(RenderingHints.KEY\_ANTIALIASING, RenderingHints.VALUE\_ANTIALIAS\_ON);

g2.setColor(getBackground()); // Usa a cor de fundo configurada

// Desenha o fundo com cantos arredondados

g2.fillRoundRect(0, 0, getWidth(), getHeight(), cornerRadius, cornerRadius);

g2.dispose();

super.paintComponent(g); // Chama a pintura padrão do JTextField

}

// Sobrescrita: personaliza a borda do componente com cantos arredondados

@Override

protected void paintBorder(Graphics g){

Graphics2D g2 = (Graphics2D) g.create();

// Habilita antialiasing para suavizar a borda

g2.setRenderingHint(RenderingHints.KEY\_ANTIALIASING, RenderingHints.VALUE\_ANTIALIAS\_ON);

g2.setColor(getForeground()); // Usa a cor de texto como cor da borda

// Desenha a borda com cantos arredondados

g2.drawRoundRect(0, 0, getWidth()-1, getHeight()-1, cornerRadius, cornerRadius);

g2.dispose();

}

}

### **2.7.7 Classe RoundedPanel**

A classe RoundedPanel estende JPanel para criar painéis com cantos arredondados, melhorando a estética de interfaces gráficas em Java Swing. Essa customização é feita através da herança, reutilizando a estrutura funcional da classe base. Utiliza encapsulamento ao manter o raio dos cantos como um atributo privado, acessível apenas internamente à classe. A aparência do componente é personalizada por meio da sobrescrita do método paintComponent, que usa Graphics2D com antialiasing para suavizar o desenho dos cantos.

Segue código abaixo com comentários explicativos:

package com.seren.components;

/\*\*

\* Painéis com cantos arredondados.

\*/

import java.awt.\*;

import javax.swing.\*;

public class RoundedPanel extends JPanel {

// Encapsulamento: controla o raio dos cantos do painel

private int cornerRadius;

// Construtor define o raio e torna o painel não opaco para permitir pintura personalizada

public RoundedPanel(int radius) {

this.cornerRadius = radius;

setOpaque(false); // Permite que o fundo seja desenhado manualmente

}

// Sobrescrita: personaliza a pintura do fundo com cantos arredondados

@Override

protected void paintComponent(Graphics g) {

Graphics2D g2 = (Graphics2D) g.create();

// Ativa o antialiasing para suavizar o contorno arredondado

g2.setRenderingHint(RenderingHints.KEY\_ANTIALIASING, RenderingHints.VALUE\_ANTIALIAS\_ON);

g2.setColor(getBackground()); // Define a cor de fundo atual

// Desenha um retângulo com cantos arredondados que preenche todo o painel

g2.fillRoundRect(0, 0, getWidth(), getHeight(), cornerRadius, cornerRadius);

g2.dispose();

super.paintComponent(g); // Garante que o restante da pintura padrão ainda ocorra

}

}

## **2.8 Conclusão**

O projeto foi estruturado com base nos princípios da programação orientada a objetos e no padrão MVC, promovendo uma separação clara entre as responsabilidades das camadas Model, DAO, Controller, Utilitárias, View e Components.

As entidades do sistema são bem definidas, a lógica de negócio é centralizada nos controllers e a interface gráfica utiliza componentes personalizados para melhorar a experiência do usuário. O uso de utilitários e validações reforça a modularidade e a reutilização de código.

De forma geral, o sistema atende aos requisitos propostos e estabelece uma base sólida e organizada para possíveis melhorias futuras.