Nome: Mateus Barros Almeida

Matrícula: 2201130053

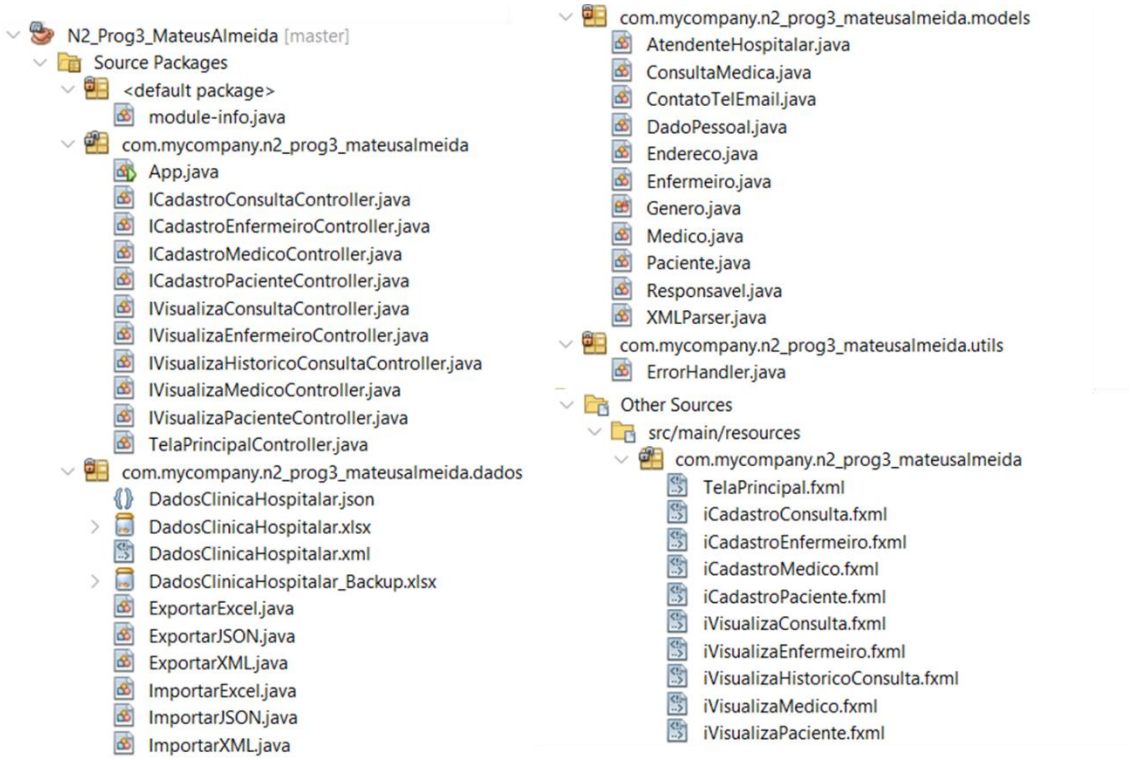
Link Youtube: https://www.youtube.com/watch?v=jn-eE-8NYuM

Documentação Clinica Hospitalar – N2

Projeto foi construído utilizando Apache NetBeans 21 e Java 17. Este consiste em um sistema de gerenciamento de informações de uma clinica médica, permitindo as operações de cadastro, visualização, edição e deleção de pacientes, médicos, enfermeiros e consultas, além da importação/exportação de dados em planilhas de excel, arquivo JSON e arquivo XML, os quais funcionam de forma análoga a um banco de dados.

Para a construção das telas foi utilizado o software SceneBuilder, o qual permite a criação de interface de aplicações JavaFX. Para importação/exportação do arquivo excel foi utilizada a API POI Apache, para importação/exportação do arquivo XML foi utilizada a API JAXB e para importação de arquivo JSON foi utilizada a API JSON In Java.

O projeto foi divido em 3 camadas, a camada de dados, em que se concentram as classes com os métodos para importação e exportação para a planilha de excel, arquivo XML e JSON, a camada de controllers, a qual se encontra na pasta raiz do projeto, junto com a classe de inicialização App.java, nela estão as classes que se comunicam com as interfaces e fazem todas as operações durante a execução do programa, a última camada é a de models, onde estão todas as classes bases, com seus respectivos atributos e métodos que serão instanciadas e utilizadas ao longo da execução do programa.



Foi criado um pacote “utils” para a classe auxiliar de tratamento de erro, a qual possui métodos para geração de caixas de diálogo. Um outro diretório importante é o de recursos, onde ficam os arquivos de extensão .fxml que são gerado pelo SceneBuilder, os quais são utilizados para renderizar as interfaces, além de identificar os elementos na tela e atribuir funções aos eventos executados.

Para as classes do pacote model foi utilizado como padrão a criação de atributos privados, com seus respectivos métodos de get e set público, além de construtores padrões e especializados, pensando em todas as possibilidades de instanciação de objetos da classe durante a execução.

```
public class AtendenteHospitalar extends DadoPessoal{
    private String setor;
    private int chSemanal;

    public AtendenteHospitalar() {
        super();
    }

    public AtendenteHospitalar(String setor, int chSemanal,String nomeCompleto, Date dataNascimento, Endereco endereco,
        super(nomeCompleto, dataNascimento, endereco, contato,genero);
        this.setor = setor;
        this.chSemanal = chSemanal;
    }

    public String getSetor() {
        return setor;
    }

    public void setSetor(String setor) {
        this.setor = setor;
    }

    public int getChSemanal() {
        return chSemanal;
    }

    public void setChSemanal(int chSemanal) {
        this.chSemanal = chSemanal;
    }

    @Override
    public String toString() {
        return "AtendenteHospitalar{" + "setor=" + setor + ", chSemanal=" + chSemanal + '}';
```

Além dos métodos supracitados, também foram utilizados métodos específicos para operações específicas. Como, por exemplo, a classe Paciente que apresenta os seguintes métodos.

```
    public static Paciente findById(ArrayList<Paciente> array,long id){
        for(Paciente paciente : array){
            if(paciente.getIdPaciente() == id)
                return paciente;
        }
        return null;
    }

    public void removerConsultaHistorico(ConsultaMedica consulta){
        this.historicoConsultasMedicas.remove(consulta);
    }

    //metodo para remover consulta do historico de consultas
    public static void removerConsultaHistorico(ArrayList<Paciente> array,ConsultaMedica consulta){
        for(Paciente paciente : array){
            Iterator<ConsultaMedica> consultas = paciente.getHistoricoConsultasMedicas().iterator();
            while(consultas.hasNext()){
                ConsultaMedica consultaAux = consultas.next();
                if(consultaAux.equals(consulta))
                    consultas.remove();
            }
        }
    }
```

O método findById() foi criado para localizar e retornar um paciente, em um vetor de pacientes, caso este seja encontrado. Já o método removerConsultaHistorico() foi desenvolvido para varrer um vetor de pacientes, procurando no histórico de consultas, se existia a consulta em questão. Caso, fosse encontrada, seria removida.

Todas as operações associadas a objetos específicos foram armazenadas em suas respectivas classes.

A tela principal do aplicativo foi construída da seguinte forma.

Nela, se apresentam os botões (Cadastrar) para acessar as telas de cadastro de médicos, enfermeiros, pacientes e consultas. Além dos botões “Editar Cadastro” que acessam as telas para visualização, edição e deleção dos registros.

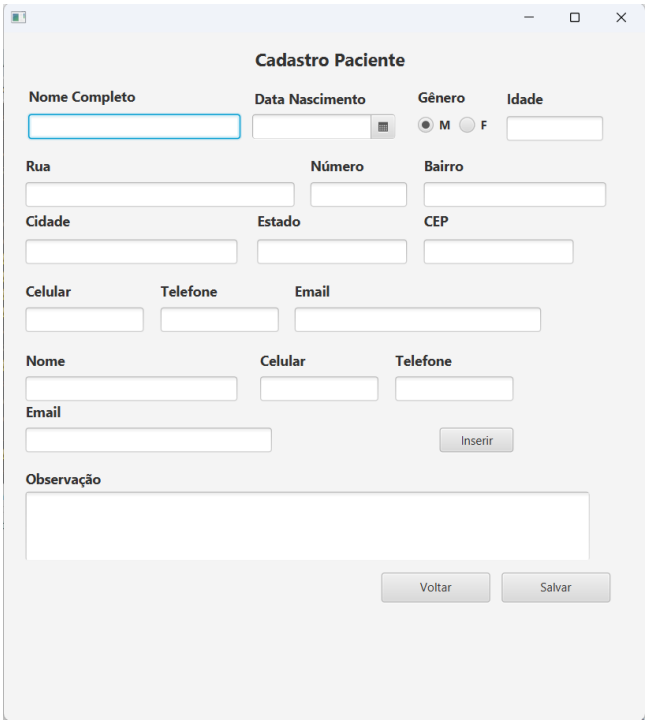
Cada tela foi construída de forma individual através do SceneBuilder e era invocada a partir da tela principal, associando o respectivo .fxml ao objeto App, utilizando o método .setRoot(), a mesma estratégia era utilizadas para navegar para qualquer página do aplicativo.

```
@FXML
private void switchToNovoPaciente() throws IOException {
    App.setRoot("iCadastroPaciente");
}
```

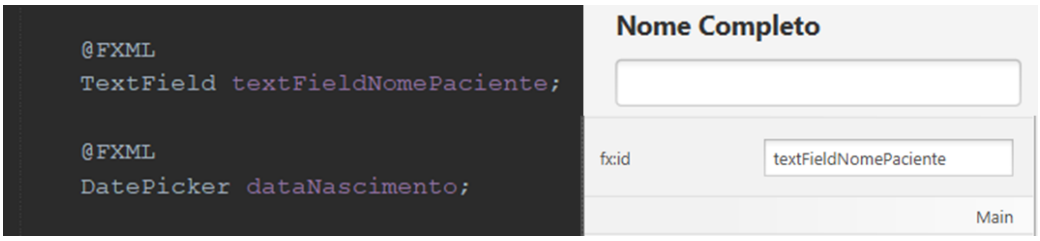
Para armazenamento das informações de médicos, pacientes, consultas médicas e enfermeiros, foram criados atributos estáticos do tipo arraylist na tela principal, cada um para cada tipo de objeto, os quais eram iniciados ao iniciar a tela principal e poderiam ser utilizados em todas as classes do sistema.

```java
public class TelaPrincipalController {

    //arraylists estaticos que serao utilizados como fonte de dados para o aplicativo, poderao
    public static ArrayList<Medico> arrayMedicos = new ArrayList<Medico>();
    public static ArrayList<Enfermeiro> arrayEnfermeiros = new ArrayList<Enfermeiro>();
    public static ArrayList<Paciente> arrayPacientes = new ArrayList<Paciente>();
    public static ArrayList<ConsultaMedica> arrayConsultas = new ArrayList<ConsultaMedica>();
```

As telas de cadastro seguiram um padrão semelhantes, as funcionalidades serão exibidas pelo cadastro de paciente, a qual possui maior complexidade. A interface de cadastro de paciente é mostrada a seguir.



Todos os campos foram criados a partir do SceneBuilder, onde foram atribuídos id's para cada um dos elementos, de forma que pudessem ser invocados e interagidos pelo programa.



Para seleção de elementos contidos em jRadioButton's de duas opções, foi utilizada uma lógica ao clique, quando um era selecionado, o outro automaticamente era desselecionado, não permitindo a seleção simultânea de ambas as opções, como pode ser visto a seguir.

```java
@FXML
private void selecionarGeneroF() throws IOException{
    if(radioBtnF.isSelected())
        radioBtnM.setSelected(false);
    else
        radioBtnM.setSelected(true);
}
```

Os métodos definidos no controller de cada das telas possuíam a anotação @FXML, dessa forma eram mapeados para o SceneBuilder e podia ser atribuídos a eventos do tipo “On Click” e “On Change” referentes aos objetos que iam acioná-lo.

Para cada paciente, havia um vetor de objetos do tipo Responsavel. Por conta disso, foi criado um atributo do tipo ArrayList de Responsavel na interface de cadastro de pacientes. Ela serviu como um vetor auxiliar para armazenar os pacientes de cada cadastro.

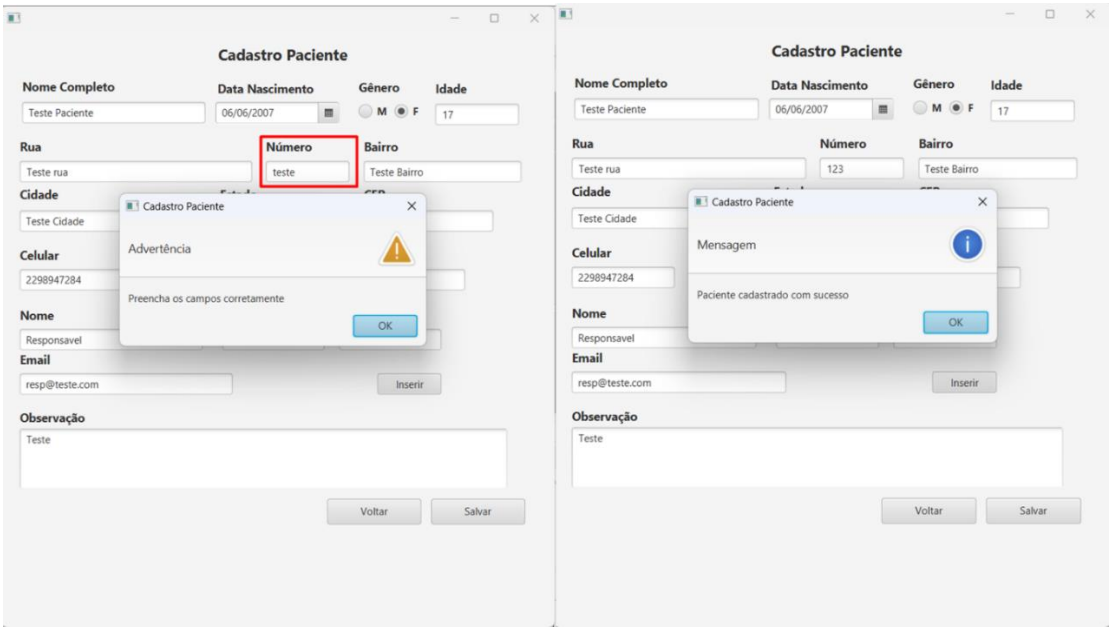
```
public class ICadastroPacienteController implements Initializable{

    ArrayList<Responsavel> contatosResp = new ArrayList<>();
```

Ao preencher as informações de Nome, Celular, Telefone e Email do responsável e clicar no botão “Inserir”, um objeto do tipo Responsavel era instanciado e associado ao vetor contatosResp, como pode ser visto no código de ação do botão.

```
@FXML
private void inserirResponsavel() throws IOException{
    String nomeResp = textFieldRespNome.getText();
    String celularResp = textFieldRespCelular.getText();
    String telefoneResp = textFieldRespTelefone.getText();
    String emailResp = textFieldRespEmail.getText();
    contatosResp.add(new Responsavel(nomeResp,celularResp,telefoneResp,emailResp));
    textFieldRespCelular.setText("");
    textFieldRespTelefone.setText("");
    textFieldRespEmail.setText("");
    textFieldRespNome.setText("");
}
```

Após o botão ser clicado, os campos eram limpos e poderiam ser inseridas as informações de um novo responsável. Dessa forma foi possível associar múltiplos responsáveis a um paciente.

Preenchendo todas as informações e clicando em um botão Salvar, haviam dois comportamentos possíveis. Caso as informações estivessem corretas, respeitando o tipo de informação de cada campo, haveria uma mensagem de sucesso. Caso houvesse alguma inconsistência, como por exemplo, uma String no campo “Número”, uma mensagem de erro era exibida na tela, os campos eram mantidos e o objeto não era criado.



Para a criação do objeto do tipo Paciente ao clicar em salvar, foram lidos os dados de todos os campos, criando os objetos Endereco, ContatoTelEmail e ArrayList<Responsavel> com as informações, permitindo a utilização do construtor especializado da classe. Toda essa execução foi envolvida em um bloco try catch permitindo a continuidade da aplicação mediante a ocorrência de uma exceção.

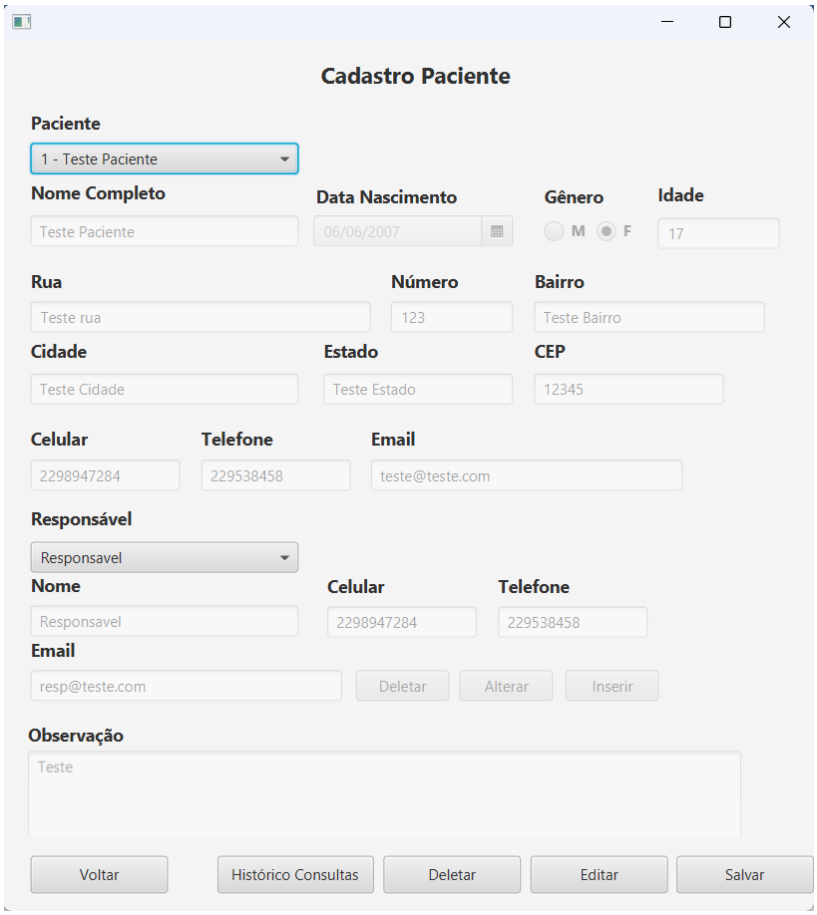
```
        String cidade = textFieldCidade.getText();
        String estado = textFieldEstado.getText();
        int cep = Integer.parseInt(textFieldCEP.getText());
        String telefone = textFieldTelefone.getText();
        String celular = textFieldCelular.getText();
        String email = textFieldEmail.getText();
        String nomeResponsavel = textFieldRespNome.getText();
        String telefoneResponsavel = textFieldRespTelefone.getText();
        String celularResponsavel = textFieldRespCelular.getText();
        String emailResponsavel = textFieldRespEmail.getText();
        String obsGeral = textAreaObs.getText();
        Endereco endereco = new Endereco(rua, numero, bairro, cidade, estado, cep);
        ContatoTelEmail contato = new ContatoTelEmail(telefone, celular, email);
        //Cria um objeto do tipo Responsavel com os valores dos campos na tela e adiciona no array de responsaveis
        contatosResp.add(new Responsavel(nomeResponsavel, telefoneResponsavel, celularResponsavel, emailResponsavel));
        Paciente paciente = new Paciente(obsGeral, new ArrayList<Responsavel>(contatosResp), nome, dataNascimento, endereco
        TelaPrincipal.arrayPacientes.add(paciente);
        System.out.println(TelaPrincipal.arrayPacientes.get(0).getNomeCompleto());
        contatosResp = new ArrayList<Responsavel>();
        ErrorHandler.exibirMsgInfo("Paciente cadastrado com sucesso", "Cadastro Paciente");
        limparCampos();
    }catch(NumberFormatException e){
        ErrorHandler.exibirMsgAlerta("Preencha os campos corretamente", "Cadastro Paciente");
    }catch(Exception e){
        ErrorHandler.exibirMsgErro("Tente Novamente", "Cadastro Paciente");
    }
}
```

Os dados de responsável vigentes na tela foram utilizados para a criação de um objeto do tipo Responsavel e esse foi adicionado ao vetor “contatosResp”, o qual foi utilizado na criação do objeto do tipo Paciente. Após a criação, o vetor “constatosResp” foi reiniciado, para não manter nenhuma informação de responsável ao criar um novo paciente.

Foi chamada a função privada limparCampos() para apagar todas as informações preenchidas e permitir uma nova execução, um método auxiliar privado associado à tela de cadastro de pacientes.

```
private void limparCampos()throws IOException{
    textFieldNomePaciente.setText("");
    dataNascimento.setValue(null);
    radioBtnM.setSelected(true);
    radioBtnF.setSelected(false);
    textFieldRua.setText("");
    textFieldNumero.setText("");
    textFieldBairro.setText("");
    textFieldCidade.setText("");
    textFieldEstado.setText("");
    textFieldCEP.setText("");
    textFieldTelefone.setText("");
    textFieldCelular.setText("");
    textFieldEmail.setText("");
    textFieldRespNome.setText("");
    textFieldRespCelular.setText("");
    textFieldRespEmail.setText("");
    textFieldRespTelefone.setText("");
    textAreaObs.setText("");
}
```

A tela de exibição, edição e visualização de consulta foi construída da seguinte forma.



Nas telas de edição havia um comboBox de seleção dos itens, utilizando como referência os itens contidos no vetor estático de pacientes, para isso, os elementos foram atribuídos ao comboBox na inicialização da classe controller, através da sobrescrita do método initialize().

```
@Override
public void initialize(URL location, ResourceBundle resources) {
    cbConsulta.getItems().addAll(TelaPrincipal.arrayConsultas);
    cbMedico.getItems().addAll(TelaPrincipal.arrayMedicos);
    cbPaciente.getItems().addAll(TelaPrincipal.arrayPacientes);
    if(TelaPrincipal.arrayConsultas.size()>0){
        cbConsulta.setValue(TelaPrincipal.arrayConsultas.get(0));
        preencherInformacoes(TelaPrincipal.arrayConsultas.get(0));
        Paciente paciente = TelaPrincipal.arrayPacientes.get(cbPaciente.getSelectionModel().getSelectedIndex());
        textFieldNomePaciente.setText(paciente.getNomeCompleto());
        textFieldIdade.setText(String.format("%d", paciente.getIdade()));
        Medico medico = TelaPrincipal.arrayMedicos.get(cbMedico.getSelectionModel().getSelectedIndex());
        textFieldNomeMedico.setText(medico.getNomeCompleto());
        textFieldCRM.setText(String.format("%d", medico.getNumeroCRM()));
    }else{
        btnDeletar.setDisable(true);
    }
    bloquearBotoes();
}
```

Dessa forma, caso existissem objetos de paciente no sistema, eles apareceriam no combobox para seleção com o texto definido pelo método toString() da classe paciente, nesse caso uma concatenação entre o id do objeto, um hífen e o nome do paciente.

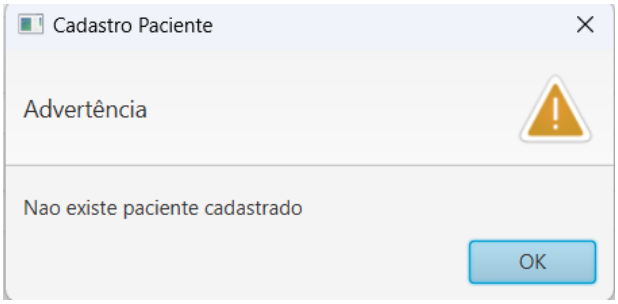
```
@Override
public String toString() {
    return idPaciente + " - " + this.getNomeCompleto();
}
```

Também foi usada a função bloquearBotoes() no construtor, a qual atribuía o valor true para o atributo disable de todos os campos que podiam ser preenchidos. Em conjunto com o métodos desbloquearBotoes() era possível fazer o controle de preenchimento dos campos.

O botão “Editar” era responsável por liberar o preenchimento dos campos, além de habilitar a opção de Salvar e desabilitar a deleção.

```
@FXML
private void editarConsulta() throws IOException {
    //Ao clicar em Editar, habilitar todos os campos e o botao de salvar, desabilitar deletar.
    if(TelaPrincipal.arrayConsultas.size() > 0){
        desbloquearBotoes();
        btnSalvar.setDisable(false);
        btnDeletar.setDisable(true);
        cbConsulta.setDisable(true);
        btnEditar.setDisable(true);
    }else{
        ErrorHandler.exibirMsgAlerta("Nao existe consulta cadastrado", "Cadastro Paciente");
    }
}
```

A fim de garantir a integridade das informações, era sempre verificado se haviam pacientes no sistema para liberar a edição, caso não houvesse, uma mensagem era mostrada ao usuário.

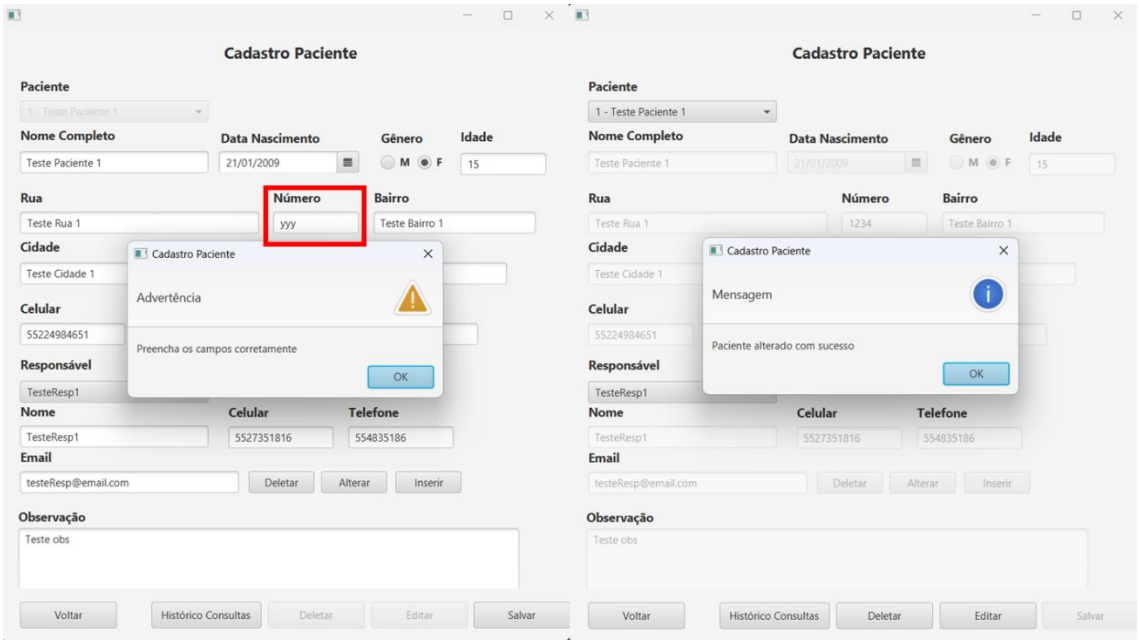


Ao clicar em “Salvar” após realizar as alterações no registro do paciente, o seguinte código é executado.

```
@FXML
private void salvarInformacoes() throws IOException{
   //editar o paciente no array estaticos com os valores preenchidos nos campos
    int id = cbPaciente.getSelectionModel().getSelectedIndex();
    Paciente paciente = TelaPrincipal.arrayPacientes.get(id);
    try{
        paciente.setNomeCompleto((String)textFieldNomePaciente.getText());
        paciente.setDataNascimento(Date.from(this.dataNascimento.getValue().atStartOfDay().toInstant(ZoneOffset.UTC)));
        paciente.setIdade((int)Integer.parseInt(textFieldIdade.getText()));
        paciente.getEndereco().setRua((String)textFieldRua.getText());
        paciente.getEndereco().setNumero((int)Integer.parseInt(textFieldNumero.getText()));
        paciente.getEndereco().setBairro((String)textFieldBairro.getText());
        paciente.getEndereco().setCidade((String)textFieldCidade.getText());
        paciente.getEndereco().setEstado((String)textFieldEstado.getText());
        paciente.getEndereco().setCep((int)Integer.parseInt(textFieldCEP.getText()));
        paciente.getContato().setCelular((String) textFieldCelular.getText());
        paciente.getContato().setTelefone((String) textFieldTelefone.getText());
        paciente.getContato().setEmail((String) textFieldEmail.getText());
        int idResponsavel = cbResponsavel.getSelectionModel().getSelectedIndex();
        paciente.getContatoResponsavel().get(idResponsavel).setCelular(textFieldRespCelular.getText());
        paciente.getContatoResponsavel().get(idResponsavel).setTelefone(textFieldRespTelefone.getText());
        paciente.getContatoResponsavel().get(idResponsavel).setEmail(textFieldRespEmail.getText());
        paciente.getContatoResponsavel().get(idResponsavel).setNomeResponsavel(textFieldRespNome.getText());
        btnEditar.setDisable(false);
        btnDeletar.setDisable(false);
        btnSalvar.setDisable(true);
        cbPaciente.setDisable(false);
        bloquearBotoes();
        ErrorHandler.exibirMsgInfo("Paciente alterado com sucesso", "Cadastro Paciente");
    }catch(NumberFormatException e){
        ErrorHandler.exibirMsgAlerta("Preencha os campos corretamente", "Cadastro Paciente");
    }catch(Exception e){
```

O paciente no vetor estático de pacientes é localizado a partir do índice do item no combobox, o qual corresponde ao índice do paciente no vetor de pacientes, dessa forma o objeto paciente é criado e seus atributos são alterados com as informações vigentes na tela através de métodos set.

Caso queira-se editar os outros responsáveis, ele deve ser selecionado pelo combobox de responsável e atualizado individualmente com o botão “Alterar”. Apenas o responsável selecionado ao clicar no botão no botão “Salvar” será alterado pelo método “salvarInformações”. Ao clicar em salvar, existem dois comportamentos possíveis, um para sucesso e outro para falha, como demonstrado a seguir



Através do botão “Deletar” é possível eliminar um paciente da base de dados, o código dessa ação é exibido a seguir.

```
@FXML
private void deletarInformacoes() throws IOException{
    //deletar o paciente selecionado da base e preencher o combobox com os remanescentes
    if(TelaPrincipal.arrayPacientes.size() > 0){
        int id = cbPaciente.getSelectionModel().getSelectedIndex();
        long idPaciente = TelaPrincipal.arrayPacientes.get(id).getIdPaciente();
        //apagar consultas do array estatico de consulta que tiverem o paciente deletado
        ConsultaMedica.removerConsultaPorIdPaciente(TelaPrincipal.arrayConsultas, idPaciente);
        TelaPrincipal.arrayPacientes.remove(id);
        ErrorHandler.exibirMsgInfo("Paciente deletado com sucesso", "Cadastro Paciente");
    }
    if(TelaPrincipal.arrayPacientes.size() != 0){
        ObservableList<Paciente> pacientes = FXCollections.observableArrayList(TelaPrincipal.arrayPacientes);
        cbPaciente.setItems(pacientes);
        cbPaciente.setValue(TelaPrincipal.arrayPacientes.get(0));
    }else{
        limparCampos();
        cbPaciente.setItems(null);
    }
}
```

Além de remover o paciente pelo método .remove(id) no vetor estático de pacientes, também é chamado o método estático removerConsultaPorIdPaciente(ArrayList<Consulta>, long), o qual remover todas as consultas do vetor que armazena as consultas médica, cujo paciente era o objeto deletado. Dessa forma, elimina a possibilidade de inconsistências na base de dados, armazenando consultas cujo id do paciente é nulo.

```java
//Metodo para excluir consulta da base de consultas ao excluir o paciente
public static void removerConsultaPorIdPaciente(ArrayList<ConsultaMedica> array, long idPaciente){
    Iterator<ConsultaMedica> consultas = array.iterator();
    while(consultas.hasNext()){
        ConsultaMedica consulta = consultas.next();
        if(consulta.getIdPaciente()== idPaciente)
            consultas.remove();
    }
}
```

Para exclusão das consultas através do método foi utilizado um objeto Iterator<ConsultaMedica>, pois ao tentar excluir diretamente do vetor estático, era exibida a exceção “ConcurrentModificationException”. O objeto iterador foi utilizado para contornar esse problema.

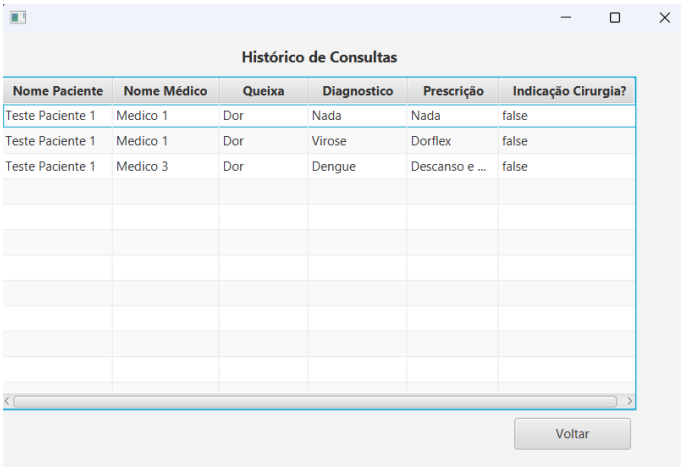
Um método semelhante também foi criado para a deleção de um médico, o qual pode ser visto no método a seguir.

```java
//Metodo para excluir consulta da base de consultas e do historico de consultas ao excluir medico
public static void removerConsultaPorIdMedico(ArrayList<ConsultaMedica> array, long idMedico){
    Iterator<ConsultaMedica> consultas = array.iterator();
    while(consultas.hasNext()){
        ConsultaMedica consulta = consultas.next();
        if(consulta.getIdMedico()== idMedico){
            Paciente.removerConsultaHistorico(TelaPrincipal.arrayPacientes, consulta);
            consultas.remove();
        }
    }
}
```

```java
//metodo para remover consulta do historico de consultas
public static void removerConsultaHistorico(ArrayList<Paciente> array,ConsultaMedica consulta){
    for(Paciente paciente : array){
        Iterator<ConsultaMedica> consultas = paciente.getHistoricoConsultasMedicas().iterator();
        while(consultas.hasNext()){
            ConsultaMedica consultaAux = consultas.next();
            if(consultaAux.equals(consulta))
                consultas.remove();
        }
    }
}
```

No entanto, além de remover a consulta do vetor estático de consultas, também era removida a consulta do histórico dos pacientes.

A última funcionalidade da tela de edição de pacientes era a exibição do histórico de consultas, a qual instanciava uma nova tela que possuía uma TableView com os dados de consultas do paciente.



Para associar os valores referentes ao paciente, o id do paciente no vetor estático de pacientes era atribuído a uma varável estática no controller IVisualizaPaciente sempre que um valor era selecionado no combobox.

```
public class IVisualizaPacienteController implements Initializable{

    public static int idPaciente;
```

Dessa forma, era possível buscar os dados do paciente. Foi necessário criar uma classe estática MyDataModel que possuía 6 propriedades, o mesmo número de colunas da tabela, associando cada informação do objeto de consulta que seria exibida na tabela com uma das propriedades.

```
public static class MyDataModel {
    private final SimpleStringProperty property1;
    private final SimpleStringProperty property2;
    private final SimpleStringProperty property3;
    private final SimpleStringProperty property4;
    private final SimpleStringProperty property5;
    private final SimpleStringProperty property6;

    public MyDataModel(String property1, String property2, String property3,
        this.property1 = new SimpleStringProperty(property1);
        this.property2 = new SimpleStringProperty(property2);
        this.property3 = new SimpleStringProperty(property3);
        this.property4 = new SimpleStringProperty(property4);
        this.property5 = new SimpleStringProperty(property5);
        this.property6 = new SimpleStringProperty(property6);
    }
```

Assim, era possível extrair todas as informações do histórico de consultas do paciente desejado e preencher os dados na tabela através do método de inicialização da classe, criando um objeto do tipo MyDataModel, utilizando os dados do objeto ConsultaMedica.

```
@Override
public void initialize(URL url, ResourceBundle rb) {
    colNomePaciente.setCellValueFactory(new PropertyValueFactory<>("property1"));
    colNomeMedico.setCellValueFactory(new PropertyValueFactory<>("property2"));
    colQueixa.setCellValueFactory(new PropertyValueFactory<>("property3"));
    colDiagnostico.setCellValueFactory(new PropertyValueFactory<>("property4"));
    colPrescricao.setCellValueFactory(new PropertyValueFactory<>("property5"));
    colCirurgia.setCellValueFactory(new PropertyValueFactory<>("property6"));

    ArrayList<MyDataModel> dataArray = new ArrayList<>();

    if(TelaPrincipal.arrayConsultas.size() > 0 && IVisualizaPaciente.idPaciente >= 0){
        for(ConsultaMedica consulta : TelaPrincipal.arrayPacientes.get(IVisualizaPaciente.idPaciente).getHistoricoConsulta
            String nomePaciente = Paciente.findById(TelaPrincipal.arrayPacientes, consulta.getIdPaciente()).getNomeComplet
            String nomeMedico = Medico.findById(TelaPrincipal.arrayMedicos, consulta.getIdMedico()).getNomeCompleto();
            dataArray.add(new MyDataModel(nomePaciente, nomeMedico, consulta.getExameQueixa(), consulta.getDiagnostico(),
        }
    ObservableList<MyDataModel> dataList = FXCollections.observableArrayList(dataArray);
    tabelaConsultas.setItems(dataList);
    }
```

O desenvolvimento das telas de Médico, Enfermeiro e Consulta seguiram quase os mesmos passos, possuindo métodos muitos semelhantes aos demonstrados para paciente. Por conta disso, não farão parte da documentação.

Para as funcionalidades de Importar/Exportar dados do excel foram criadas as classes ExportarExcel e ImportarExcel no pacote de dados, utilizando a API excel Apache POI, a qual foi adicionada como dependência através do arquivo de configuração pom.xml

```xml
<dependencies>
    <dependency>
        <groupId>org.apache.poi</groupId>
        <artifactId>poi</artifactId>
        <version>5.0.0</version>
    </dependency>
    <dependency>
        <groupId>org.apache.poi</groupId>
        <artifactId>poi-ooxml</artifactId>
        <version>5.0.0</version>
    </dependency>
    <dependency>
        <groupId>com.toedter</groupId>
        <artifactId>jcalendar</artifactId>
        <version>1.4</version>
    </dependency>
</dependencies>
```

Para identificação da API no código e liberação de acesso às classes do projeto, também era necessário adicionar o módulo no arquivo modules-info no pacote padrão, esse procedimento foi feito para todas as dependências importadas.

```
module com.mycompany.n2_prog3_mateusalmeida {
    requires javafx.controls;
    requires javafx.fxml;
    opens com.mycompany.n2_prog3_mateusalmeida to javafx.fxml;
    opens com.mycompany.n2_prog3_mateusalmeida.models to java.xml.bind;
    exports com.mycompany.n2_prog3_mateusalmeida;
    requires org.apache.poi.poi;
    requires org.apache.poi.ooxml;
    requires org.json;
    requires java.xml.bind;
    requires java.activation;
}
```

Através dessa biblioteca foi possível criar objetos do tipo worksheet, sheet e row, permitindo a criação e interação com planilhas Excel.

A planilha foi estruturada em cinco abas, Paciente, Médico, Enfermeiro, Consultas Médicas e Responsavel.

Começando pela função de ExportarExcel. Inicialmente foram construídas as Strings com o cabeçalho de cada uma das abas e adicionados no workbook.

```
//criar cabecalho da aba Paciente
String[] pacienteHeader = {"id Paciente","Nome Completo","Data de Nascimento", "Idade",
    "Genero","Data de Cadastro","Rua","Numero","Bairro","Cidade","Estado","CEP",
    "Celular","Telefone","Email","obsGeral","Historico Consulta","contatoResponsavel"};

XSSFSheet abaPaciente = workbook.createSheet( "Paciente");
Row headerRow = abaPaciente.createRow(0);

for (int i = 0; i < pacienteHeader.length; i++) {
    Cell cell = headerRow.createCell(i);
    cell.setCellValue(pacienteHeader[i]);
}
```

Definido o cabeçalho, varre-se o vetor estático com as informações para preenchimento da planilha

```
//Contador utilizado para criar o id Responsavel que sera usado como PK para correlacionar com paciente
int contador = 0;
//Preencher informacoes de paciente na aba paciente
for(int i = 0; i < TelaPrincipal.arrayPacientes.size(); i++) {
    Row dataRow = abaPaciente.createRow(i + 1);
    dataRow.createCell(0).setCellValue(TelaPrincipal.arrayPacientes.get(i).getIdPaciente());
    dataRow.createCell(1).setCellValue(TelaPrincipal.arrayPacientes.get(i).getNomeCompleto());
    dataRow.createCell(2).setCellValue(formatoData.format(TelaPrincipal.arrayPacientes.get(i).getDataNascimento()));
    dataRow.createCell(3).setCellValue(TelaPrincipal.arrayPacientes.get(i).getIdade());
    dataRow.createCell(4).setCellValue(TelaPrincipal.arrayPacientes.get(i).getGenero().toString());
    dataRow.createCell(5).setCellValue(formatoData.format(TelaPrincipal.arrayPacientes.get(i).getDataCadastro()));
```

Como não seria possível a criação de um objeto de informação de consultas e responsáveis no excel, foi criada uma String com os id's de consulta e responsável associados ao paciente, separados por vírgula.

Os id's de consultas eram extraídos dos objetos que compunham o arraylist historicoConsultas, que estava em cada paciente.

```
//Array criado com os ids de todas as consultas, separados por ","
for(ConsultaMedica consulta : TelaPrincipal.arrayPacientes.get(i).getHistoricoConsultasMedicas()){
    consultasAux += consulta.getIdConsulta() + ",";
}
if(consultasAux.length()>0)
//Metodo para remover a ultima ","
dataRow.createCell(16).setCellValue(consultasAux.substring(0, consultasAux.length()-1));
```

Para determinação do id de responsável, como não havia um vetor com todos os responsáveis cadastrados, foi criada uma aba no excel onde eram registrados os responsáveis e o id designado ao registrar era associado ao paciente.

```
//Array criado com os ids de todas os responsaveis, separados por ","
String responsavelAux = "";
for(Responsavel responsavel : TelaPrincipal.arrayPacientes.get(i).getContatoResponsavel()){
    //Criando registro de Responsavel a partir do arraylist de contato Responsavel de cada paciente
    //Foi feito dessa forma para linkar o id gravado na tabela Paciente com o id Responsavel da tabela Responsavel
    Row dataRowResponsavel = abaResponsavel.createRow(contador + 1);
    dataRowResponsavel.createCell(0).setCellValue(contador+1);
    dataRowResponsavel.createCell(1).setCellValue(responsavel.getNomeResponsavel());
    dataRowResponsavel.createCell(2).setCellValue(responsavel.getCelular());
    dataRowResponsavel.createCell(3).setCellValue(responsavel.getTelefone());
    dataRowResponsavel.createCell(4).setCellValue(responsavel.getEmail());
    responsavelAux += (contador+1) + ",";
    contador++;
}
//Metodo para remover a ultima ","
if(responsavelAux.length()>0)
    dataRow.createCell(17).setCellValue(responsavelAux.substring(0, responsavelAux.length()-1));
```

A inserção dos registros nas outras abas foi feita de modo semelhantes.

Já para a importação dos dados executa-se o fluxo inverso, incialmente são zerados os vetores estáticos com as informações, pois optou-se ao importar, deixar apenas os dados importados na base de dados. Em seguida, são lidas as abas e contabilizados os números de registros.

```
TelaPrincipal.arrayConsultas = new ArrayList<>();
TelaPrincipal.arrayMedicos = new ArrayList<>();
TelaPrincipal.arrayPacientes = new ArrayList<>();
TelaPrincipal.arrayEnfermeiros = new ArrayList<>();

SimpleDateFormat formato = new SimpleDateFormat("dd/MM/yyyy");

//Atribuir abas aos objetos do tipo Sheet
Sheet sheetPaciente = workbook.getSheet("Paciente");
Sheet sheetMedico = workbook.getSheet("Médico");
Sheet sheetEnfermeiro = workbook.getSheet("Enfermeiro");
Sheet sheetConsulta = workbook.getSheet("Consulta Médica");
Sheet sheetResponsavel = workbook.getSheet("Responsavel");
//Atribuir o numero de registros em cada aba a uma variavel
int rowCountPaciente = sheetPaciente.getLastRowNum()-sheetPaciente.getFirstRowNum();
int rowCountMedico = sheetMedico.getLastRowNum()-sheetMedico.getFirstRowNum();
int rowCountEnfermeiro = sheetEnfermeiro.getLastRowNum()-sheetEnfermeiro.getFirstRowNum();
int rowCountConsulta = sheetConsulta.getLastRowNum()-sheetConsulta.getFirstRowNum();
int rowCountResponsavel = sheetResponsavel.getLastRowNum()-sheetResponsavel.getFirstRowNum();
```

Os registros são lidos, armazenados nas respectivas variáveis. Os objetos necessários são criados, utilizando-se os respectivos construtores especializados.

```
//Preencher o array estatico de consultas com as informacoes da aba Consulta Medica do excel
for (int i=1; i<=rowCountConsulta; i++){
    Row rowConsulta = sheetConsulta.getRow(i);
    long idConsulta = (long)rowConsulta.getCell(0).getNumericCellValue();
    long idPaciente = (long)rowConsulta.getCell(1).getNumericCellValue();
    long idMedico = (long)rowConsulta.getCell(2).getNumericCellValue();
    String queixa = rowConsulta.getCell(3).getStringCellValue();
    String diagnostico = rowConsulta.getCell(4).getStringCellValue();
    String prescricao = rowConsulta.getCell(5).getStringCellValue();
    boolean indicacaoCirurgia = rowConsulta.getCell(6).getBooleanCellValue();
    ConsultaMedica consulta = new ConsultaMedica(idConsulta, idPaciente, idMedico, queixa, diagnostico,
    TelaPrincipal.arrayConsultas.add(consulta);
```

Após a criação do objeto ele era salvo no vetor estático correspondente à aba da planilha que estava sendo varrida.

A importação das outras abas ocorreu de forma semelhantes, com exceção da tabela de pacientes, pois paciente possui dois atributos que são arraylists.

Para a criação do vetor histórico de consulta, varreu-se o vetor de String criado a partir da String de id's de consulta, utilizando o método split(), buscando o objeto consulta no vetor estático de consultas através do método findById().

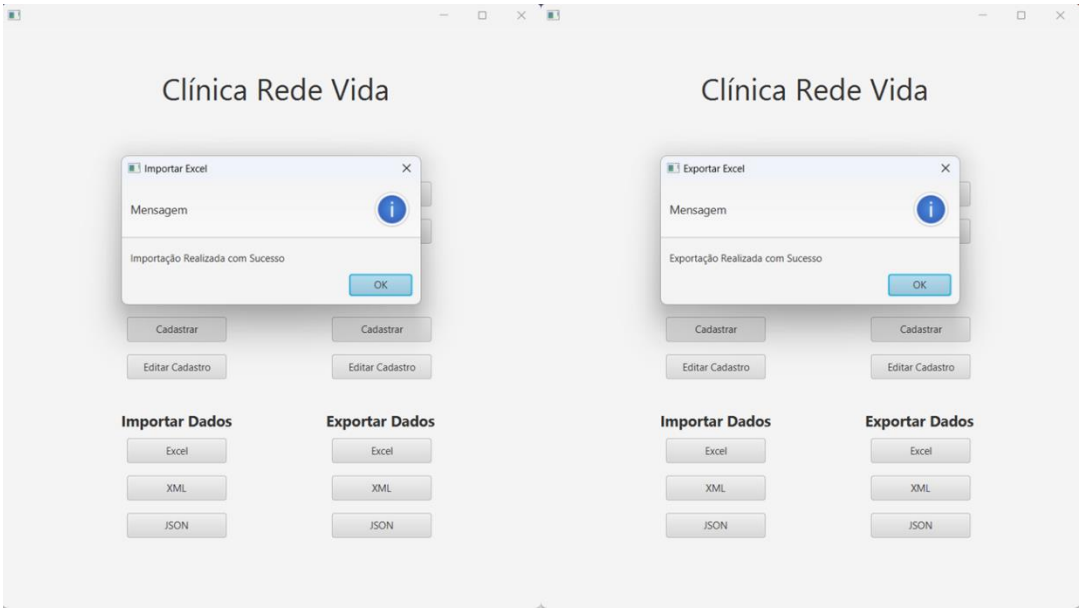
```
ArrayList<ConsultaMedica> historicoConsultas =  new ArrayList<>();
ArrayList<Responsavel> contatoResponsavel = new ArrayList<>();
//Varrer a string com as ids de consulta
for(String id : idConsultas){
    //Acessar a consulta no vetor estatico de consultas utilizando o idd
    ConsultaMedica consulta = ConsultaMedica.findById(TelaPrincipal.arrayConsultas, Long.parseLong(id));
    if(consulta != null)
        //se existir consulta adicionar ao historico
        historicoConsultas.add(consulta);
}
```

Já para a criação do vetor de responsáveis, utilizou-se uma estratégia parecida, ao invés de buscar em um vetor estático, o id do responsável era buscado na tabela Responsável do próprio excel, varrendo todos os seus registros.

```java
//Varrer a string com as ids dos responsaveis
for(String id : idResponsaveis){
    for (int j=1; j<=rowCountResponsavel; j++){
        Row rowResponsavel = sheetResponsavel.getRow(j);
        long idResponsavel = (long)rowResponsavel.getCell(0).getNumericCellValue();
        //acessar os registros da tabela Responsavel e verificar o registro que possui o id
        if(Long.parseLong(id) == idResponsavel){
            String nomeResponsavel = rowResponsavel.getCell(1).getStringCellValue();
            String celularResponsavel = rowResponsavel.getCell(2).getStringCellValue();
            String telefoneResponsavel = rowResponsavel.getCell(3).getStringCellValue();
            String emailResponsavel = rowResponsavel.getCell(4).getStringCellValue();
            Responsavel responsavel = new Responsavel(nomeResponsavel, telefoneResponsavel, celularResponsavel,
            //criar um objeto responsavel e adicionar no arraylist de contato responsavel
            contatoResponsavel.add(responsavel);
        }
    }
}
```

Dessa forma, foi possível criar os ArrayLists de Consulta Medica e Responsavel, possibilitando a instanciação de um objeto Paciente e sua posterior adição ao vetor estático de pacientes.

Ao selecionar a importação ou exportação de dados, avisos na tela com a conclusão da operação são exibidos.



Para as funcionalidades de Importar/Exportar dados do XML foram criadas as classes ExportarXML e ImportarXML no pacote de dados, utilizando a API JAXB, a qual foi adicionada como dependência através do arquivo de configuração pom.xml

```xml
<!-- https://mvnrepository.com/artifact/javax.xml.bind/jaxb-api -->
    <dependency>
        <groupId>javax.xml.bind</groupId>
        <artifactId>jaxb-api</artifactId>
        <version>2.3.0</version>
    </dependency>

<dependency>
        <groupId>org.glassfish.jaxb</groupId>
        <artifactId>jaxb-runtime</artifactId>
        <version>2.3.1</version>
    </dependency>
    <dependency>
        <groupId>javax.activation</groupId>
        <artifactId>javax.activation-api</artifactId>
        <version>1.2.0</version>
    </dependency>
```

Para fazer a conversão das informações no arquivo XML foi criada a classe XMLParser no pacote de models. Essa classe era constituída de quatro vetores, um referente a cada tipo de objeto que seria guardado no XML (Paciente, Medico, Enfermeiro, ConsultaMedica) e seus respectivos métodos get e set.

```java
@XmlRootElement(name = "dados")
public class XMLParser {

    private Paciente[] pacientes;
    private Medico[] medicos;
    private ConsultaMedica[] consultas;
    private Enfermeiro[] enfermeiros;

    public XMLParser() {
    }

    public XMLParser(Paciente[] pacientes, Medico[] medicos, ConsultaMedica[] consultas, Enfermeiro[] enfermeiros) {
        this.pacientes = pacientes;
        this.medicos = medicos;
        this.consultas = consultas;
        this.enfermeiros = enfermeiros;
    }

    public Paciente[] getPacientes() {
        return pacientes;
    }
```

Em cima da classe é utiliza a annotation @XmlRootElement, indicando a tag raiz do arquivo XML gerado, caso queria atribuir um nome específico, pode-se colocar adicionar como parâmetro da annotation. Para os atributos da classe é possível usar a annotation @XMLElement e definir o nome da tag, mas nesse projeto não foi utilizada e, por padrão, foi selecionado o nome dos atributos.

A função utilizada para a exportação dos dados em XML foi a seguinte.

```java
public class ExportarXML {

    public static void exportarXML() throws IOException{

        try{

            JAXBContext jaxbContextDados = JAXBContext.newInstance(XMLParser.class);
            Marshaller marshallerConsulta = jaxbContextDados.createMarshaller();
            marshallerConsulta.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
            FileWriter fDados = new FileWriter("C:/Users/mateu/Desktop/Estudos/Femass/Prog 3/N2_Prog3_MateusAlmeida/
            ConsultaMedica[] consultas = new ConsultaMedica[TelaPrincipalController.arrayConsultas.size()];
            for(int i = 0; i < TelaPrincipalController.arrayConsultas.size(); i++){
                consultas[i] = TelaPrincipalController.arrayConsultas.get(i);
            }
            Enfermeiro[] enfermeiros = new Enfermeiro[TelaPrincipalController.arrayEnfermeiros.size()];
            for(int i = 0; i < TelaPrincipalController.arrayEnfermeiros.size(); i++)
                enfermeiros[i] = TelaPrincipalController.arrayEnfermeiros.get(i);
            Medico[] medicos = new Medico[TelaPrincipalController.arrayMedicos.size()];
            for(int i = 0; i < TelaPrincipalController.arrayMedicos.size(); i++)
                medicos[i] = TelaPrincipalController.arrayMedicos.get(i);
            Paciente[] pacientes = new Paciente[TelaPrincipalController.arrayPacientes.size()];
            for(int i = 0; i < TelaPrincipalController.arrayPacientes.size(); i++)
                pacientes[i] = TelaPrincipalController.arrayPacientes.get(i);
            XMLParser dadosXML = new XMLParser(pacientes, medicos, consultas, enfermeiros);
            marshallerConsulta.marshal(dadosXML, fDados);
            ErrorHandler.exibirMsgInfo("Exportação Realizada com Sucesso", "Exportar XML");

        }catch(Exception ex){
            ErrorHandler.exibirMsgErro("Tente Novamente", "Exportar XML");
        }
    }
```

Essa função necessita de dois objetos, um da classe JAXBContext, a qual é criada utilizando a classe de converão (XMLParser) como referência, e um da classe Marshaller que é responsável pela conversão do objeto do tipo XMLParser em um arquivo XML através do método .marshal(). Essa conversão é realizada em um arquivo cujo diretório é especificado em código e o resultado pode ser observado a seguir.

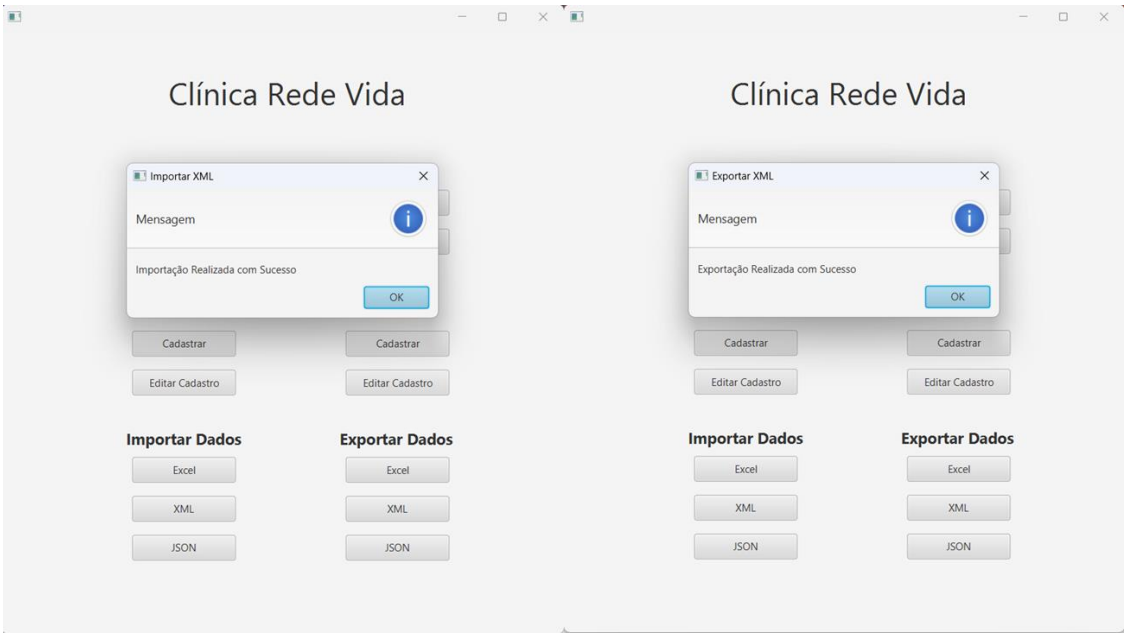
```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<dados>
    <consultas>
        <diagnostico>Nada</diagnostico>
        <exameQueixa>Dor</exameQueixa>
        <idConsulta>1</idConsulta>
        <idMedico>1</idMedico>
        <idPaciente>1</idPaciente>
        <indicacaoCirurgica>false</indicacaoCirurgica>
        <prescricao>Nada</prescricao>
    </consultas>
    <consultas>
        <diagnostico>Virose</diagnostico>
        <exameQueixa>Dor</exameQueixa>
        <idConsulta>2</idConsulta>
        <idMedico>1</idMedico>
        <idPaciente>1</idPaciente>
        <indicacaoCirurgica>false</indicacaoCirurgica>
        <prescricao>Dorflex</prescricao>
    </consultas>
```

Para importar as informações de um arquivo de extensão XML foi utilizada o método estático importarXML() da classe ImportarXML, o processo ocorre de forma análoga à exportação, no entanto, é utilizado um objeto do tipo Unmarshaller, o qual realiza a conversão do arquivo XML em objetos das classes mapeadas pela classe de conversão.

```java
public static void importarXML() throws IOException{

    TelaPrincipalController.arrayConsultas = new ArrayList<>();
    TelaPrincipalController.arrayMedicos = new ArrayList<>();
    TelaPrincipalController.arrayPacientes = new ArrayList<>();
    TelaPrincipalController.arrayEnfermeiros = new ArrayList<>();

    try{
    JAXBContext jaxbContextDados = JAXBContext.newInstance(XMLParser.class);
    Unmarshaller unMarshallerPaciente = jaxbContextDados.createUnmarshaller();
    XMLParser dados = (XMLParser)unMarshallerPaciente.unmarshal(new FileReader("C:/Users/
    for(Paciente paciente : dados.getPacientes()){
        TelaPrincipalController.arrayPacientes.add(paciente);
    }
    for(Medico medico : dados.getMedicos()){
        TelaPrincipalController.arrayMedicos.add(medico);
    }
    for(Enfermeiro enfermeiro : dados.getEnfermeiros()){
        TelaPrincipalController.arrayEnfermeiros.add(enfermeiro);
    }
    for(ConsultaMedica consulta : dados.getConsultas()){
        TelaPrincipalController.arrayConsultas.add(consulta);
    }
    ErrorHandler.exibirMsgInfo("Importação Realizada com Sucesso", "Importar XML");
    }catch(Exception ex){
        ErrorHandler.exibirMsgErro("Tente Novamente", "Importar XML");
    }
}
```

Ao executar o método .unmarshal(File) um objeto do tipo XMLParser é criado e o conteúdo é atribuído para seus atributos, os quais posteriormente são atribuídos para os vetores estáticos utilizados na aplicação, ficando disponível para acesso.

Ao selecionar a importação ou exportação de XML, avisos na tela com a conclusão da operação são exibidos.



Para as funcionalidades de Importar/Exportar dados do XML foram criadas as classes ExportarJSON e ImportarJSON no pacote de dados, utilizando a API JSON In Java, a qual foi adicionada como dependência através do arquivo de configuração pom.xml

```
<!-- https://mvnrepository.com/artifact/org.apache.poi/poi-ooxml -->
<dependency>
    <groupId>org.apache.poi</groupId>
    <artifactId>poi-ooxml</artifactId>
    <version>5.2.5</version>
</dependency>

<!-- https://mvnrepository.com/artifact/org.json/json -->
<dependency>
    <groupId>org.json</groupId>
    <artifactId>json</artifactId>
    <version>20240303</version>
</dependency>
```

Para realizar a exportação dos dados da aplicação em um arquivo no formato .JSON foi utilizado o método estático exportarJSON() da classe ExportarJSON. Ele se comporta de maneira similar ao método de exportação de excel, no entanto, ao invés de estruturar abas, colunas e linhas, trabalhamos com objetos JSON (JSONObject) e arrays JSON (JSONArray).

Tomando como exemplo a criação de um JSON para paciente. Inicialmente um objeto JSON é criado, cada atributo será uma propriedade e seu conteúdo será um valor, no entanto, atributos agregados, como Endereço, o valor será um outro objeto do tipo JSONObject. Dessa forma é possível montar um objeto JSON para um paciente, esse processo é feito para cada paciente contido no array estático da aplicação e posteriormente eles são adicionados em um array destinado a sua classe de origem, nesse caso um array de pacientes. Após realizar o mesmo processo para as consultas, médicos e enfermeiros, esses array são adicionados em um array geral, o qual é exportado para o arquivo final.

```java
public static void exportarXML() throws IOException{
    JSONObject dados = new JSONObject();
    SimpleDateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy");

    JSONArray pacientesArray = new JSONArray();
    JSONArray medicosArray = new JSONArray();
    JSONArray enfermeirosArray = new JSONArray();
    JSONArray consultasArray = new JSONArray();

    //Loop through the pacientes ArrayList and convert each Paciente to JSONObject
    for (Paciente paciente : TelaPrincipalController.arrayPacientes){
        JSONObject pacienteObject = new JSONObject();
        pacienteObject.put("nomeCompleto", paciente.getNomeCompleto());
        pacienteObject.put("dataNascimento", dateFormat.format(paciente.getDataNascimento()));
        JSONObject enderecoObject = new JSONObject();
        enderecoObject.put("rua", paciente.getEndereco().getRua());
        enderecoObject.put("numero", String.format("%d", paciente.getEndereco().getNumero()));
        enderecoObject.put("bairro", paciente.getEndereco().getBairro());
        enderecoObject.put("cidade", paciente.getEndereco().getCidade());
        enderecoObject.put("estado", paciente.getEndereco().getEstado());
        enderecoObject.put("CEP", String.format("%d", paciente.getEndereco().getCep()));
        pacienteObject.put("endereco", enderecoObject);
        JSONObject contatoObject = new JSONObject();
        contatoObject.put("celular", paciente.getContato().getCelular());
        contatoObject.put("telefone", paciente.getContato().getTelefone());
        contatoObject.put("email", paciente.getContato().getEmail());
        pacienteObject.put("contato", contatoObject);
        pacienteObject.put("genero", paciente.getGenero());
        pacienteObject.put("idPaciente", paciente.getIdPaciente());
        pacienteObject.put("idade", paciente.getIdade());
```

Após a adição de todas as informações no array geral, o objeto é escrito no arquivo estipulado, utilizando o método .toString.

```java
dados.put("consultas",consultasArray);

// Write the JSONArray to a file;
try{
FileWriter writer = new FileWriter("C:/Users/mateu/Desktop/Estudos/Femass/Prog 3
writer.write(dados.toString());
ErrorHandler.exibirMsgInfo("Exportação Realizada com Sucesso", "Exportar JSON");
writer.close();
}catch(Exception ex){
    ErrorHandler.exibirMsgErro("Tente Novamente", "Exportar JSON");
}
```

Exemplo do arquivo gerado.

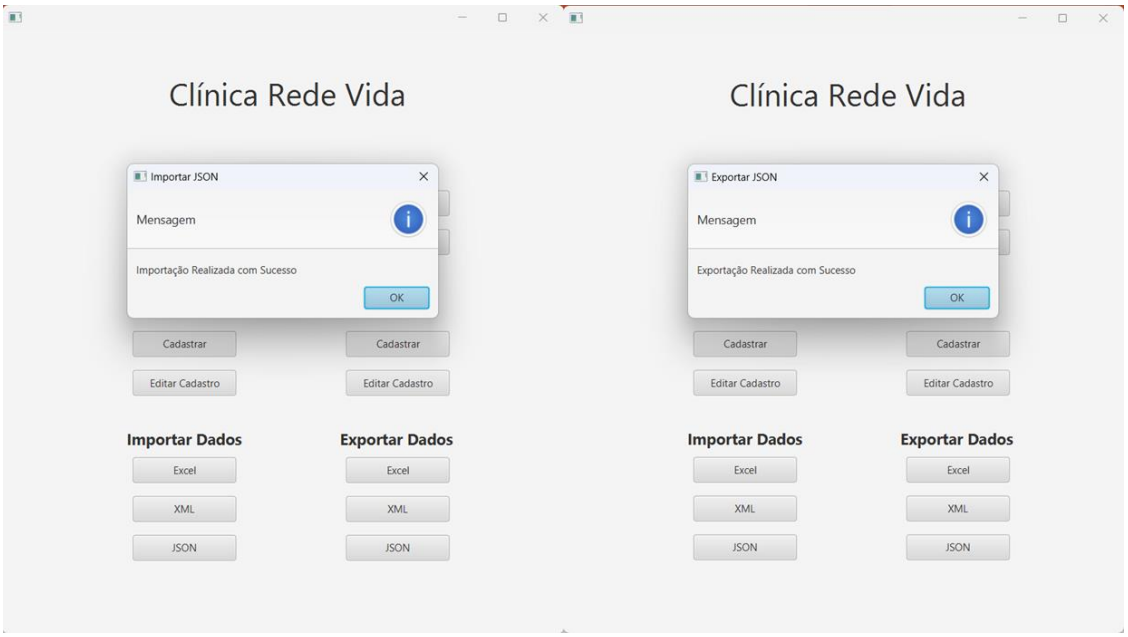

```json
{
    "enfermeiros": [
        {
            "setor": "teste",
            "idEnfermeiro": 1,
            "endereco": {
                "cidade": "teste",
                "estado": "teste",
                "numero": "555",
                "bairro": "teste",
                "rua": "teste",
                "CEP": "68546"
            },
```

Para realizar a importação foi utilizado o método estático importarJSON() da classe ImportarJSON. O processo ocorre de forma análoga à exportação, no entanto, a informação é recuperada dos objetos/arrays JSON e objetos de Paciente, Medico, Enfermeiro e ConsultaMedica são criados e adicionados no array estático referente à respectiva classe. Uma parte do código pode ser observada abaixo.

```
public static void importarJSON() throws IOException{
    TelaPrincipalController.arrayConsultas = new ArrayList<>();
    TelaPrincipalController.arrayMedicos = new ArrayList<>();
    TelaPrincipalController.arrayPacientes = new ArrayList<>();
    TelaPrincipalController.arrayEnfermeiros = new ArrayList<>();

    SimpleDateFormat formato = new SimpleDateFormat("dd/MM/YYYY");

     try {

        // Specify path to your JSON file
        String content = new String(Files.readAllBytes(Paths.get("C:/Users/mateu/

        // Read JSON from file and map to object
        JSONObject dados = new JSONObject(content);
        JSONArray arrayPacientes = dados.getJSONArray("pacientes");
        JSONArray arrayMedicos = dados.getJSONArray("medicos");
        JSONArray arrayEnfermeiros = dados.getJSONArray("enfermeiros");
        JSONArray arrayConsultas = dados.getJSONArray("consultas");

        for(int i = 0; i < arrayConsultas.length(); i++){
            JSONObject jsonObject = arrayConsultas.getJSONObject(i);
            long idConsulta = jsonObject.getLong("idConsulta");
            long idPaciente = jsonObject.getLong("idPaciente");
            long idMedico = jsonObject.getLong("idMedico");
            String queixa = jsonObject.getString("queixa");
            String diagnostico = jsonObject.getString("diagnostico");
            String prescricao = jsonObject.getString("prescricao");
            boolean indicacaoCirurgia = jsonObject.getBoolean("cirurgia");
            ConsultaMedica consulta = new ConsultaMedica(idConsulta, idPaciente,
            TelaPrincipalController.arrayConsultas.add(consulta);
```

Esse processo é realizado em todos os objetos do arquivo JSON e todas as informações são recuperadas.

Ao selecionar a importação ou exportação de JSON, avisos na tela com a conclusão da operação são exibidos.



Para finalizar, as maiores dificuldades na elaboração do projeto se concentraram na descoberta e utilização das API's para a conversão das informações em JSON e XML, já que se tratavam de bibliotecas externas que demandaram uma grande carga de estudo para adequação ao código. O desenvolvimento do backend teve menor impacto nesse projeto, pois grande parte já estava desenvolvida para a aplicação anterior, podendo ser reaproveitada em sua grande maioria.