

Mecânica Nexus ERP
Projeto de Programação em Java
Com Integração ao Banco de Dados

¹Discente do curso de Ciência da Computação
Unoesc-Campus de São Miguel do Oeste
Rua Oiapoc, 2011. São Miguel do Oeste-SC
ivan.js23silva@gmail.com

²Discente do curso de Ciência da Computação
Unoesc-Campus de São Miguel do Oeste
Rua Oiapoc, 2011. São Miguel do Oeste-SC
demossigabi@gmail.com

³Discente do curso de Ciência da Computação
Unoesc-Campus de São Miguel do Oeste
Rua Oiapoc, 2011. São Miguel do Oeste-SC
depineguilherme@gmail.com

⁴Discente do curso de Ciência da Computação
Unoesc-Campus de São Miguel do Oeste
Rua Oiapoc, 2011. São Miguel do Oeste-SC
gabriel.mn@unoesc.edu.br

⁵Discente do Curso de Ciência da Computação
Unoesc-Campus de São Miguel do Oeste
Rua Oiapoc, 2011. São Miguel do Oeste-SC
kauan.ac@unoesc.edu.br

⁶Discente do Curso de Ciência da Computação
Unoesc-Campus de São Miguel do Oeste
Rua Oiapoc, 2011. São Miguel do Oeste-SC
vitor.konzen@unoesc.edu.br

⁷Mestre em Informática
Docente do Curso de Ciência da Computação
Unoesc-Campus de São Miguel do Oeste
Rua Oiapoc, 2011. São Miguel do Oeste-SC
Otilia.barbosa@unoesc.edu.br

⁸Mestre em Informática
Docente do Curso de Ciência da Computação
Unoesc-Campus de São Miguel do Oeste
Rua Oiapoc, 2011. São Miguel do Oeste-SC
Otilia.barbosa@unoesc.edu.br

RESUMO

O artigo aborda o desenvolvimento do Mecânica Nexus ERP, um sistema operacional voltado para a gestão de processos empresariais com integração a banco de dados. A motivação principal foi preencher lacunas deixadas por soluções ERP existentes, como falta de personalização, alto custo e complexidade de implementação. O projeto oferece uma plataforma intuitiva, versátil e acessível, destacando-se em funcionalidades como cadastro de dados, gestão de operações, geração de relatórios, backup e segurança, além de interface amigável. Diagramas e exemplos de códigos ilustram a estrutura e a funcionalidade do sistema, evidenciando sua aplicabilidade e potencial de impacto no mercado.

Palavras-chave: ERP, Gestão Empresarial, Banco de Dados, Automação, Sistema Operacional.

1. INTRODUÇÃO

No mercado competitivo e dinâmico atual, a eficiência na gestão de processos empresariais é essencial para o sucesso das organizações. Embora existam diversas soluções de software ERP (*Enterprise Resource Planning*) disponíveis, muitas delas apresentam limitações específicas, como falta de personalização, altos custos ou complexidade de implementação, deixando um espaço considerável no mercado para alternativas mais acessíveis e direcionadas.

Foi nesse contexto que a Mecânica Nexus ERP foi desenvolvida. Com o objetivo de preencher essa lacuna, a solução se destaca por oferecer uma plataforma versátil, intuitiva e altamente adaptável às necessidades de empresas que buscam otimizar sua gestão operacional sem comprometer o orçamento ou a funcionalidade.

Este artigo explora os principais diferenciais da Mecânica Nexus ERP, abordando seu desenvolvimento, suas características principais e o impacto potencial no mercado de ERPs. A introdução detalha a relevância dessa inovação no setor, enquanto as seções subsequentes apresentam a metodologia de análise, os resultados obtidos e as perspectivas futuras para a ferramenta.

2. FUNCIONALIDADES DA MECÂNICA NEXUS ERP

2.1 CADASTRO DE DADOS

- Cadastro de Clientes:
- Permitir que o sistema registre informações de clientes, como CPF ou CNPJ.
- Cadastro de Fornecedores:
- Possibilitar o cadastro de fornecedores, incluindo dados básicos de identificação.
- Cadastro de Produtos ou Serviços:
- Permitir que usuários registrem produtos ou serviços para controle de estoque e vendas.
- Cadastro de Agendamentos:
- Oferecer a funcionalidade de agendar serviços com código de identificação para controle.

2.2 GESTÃO DE OPERAÇÕES

- Realizar Vendas:
- Registrar vendas com base no número de identificação do cliente, integrando estoque e financeiro.
- Visualizar Dados:
 - Clientes: Possibilitar a visualização dos cadastros de clientes.
 - Fornecedores: Exibir informações dos fornecedores cadastrados.
 - Produtos: Permitir que os usuários consultem informações dos produtos cadastrados.
 - Agendamentos: Mostrar os agendamentos registrados no sistema.

2.3 RELATÓRIOS

- Relatório de Vendas: Gerar relatórios detalhados contendo nome, CNPJ e outras informações relevantes de vendas realizadas.

2.4 BACKUP E RECUPERAÇÃO

- Backup do Banco de Dados: Oferecer a opção de gerar backups completos dos dados do sistema, garantindo a segurança das informações.
- Restauração de *Backup*: Implementar uma funcionalidade para restaurar dados a partir de *backups*, quando necessário.

2.5 SEGURANÇA E ACESSOS

- Tela de Autenticação: Solicitar login do usuário para acessar o sistema, garantindo o controle de acessos.
- Sair do Sistema: Permitir que o sistema seja fechado de maneira segura, finalizando sessões ativas.

2.6 INTERFACE E DESEMPENHO

- Interface Simples e Funcional: Desenvolver uma interface intuitiva, fácil de usar e responsiva, focada na experiência do usuário.
- Armazenamento Seguro de Serviços: Garantir que todos os dados sejam armazenados de forma segura e acessível, mesmo após atualizações.

3 DIAGRAMAS

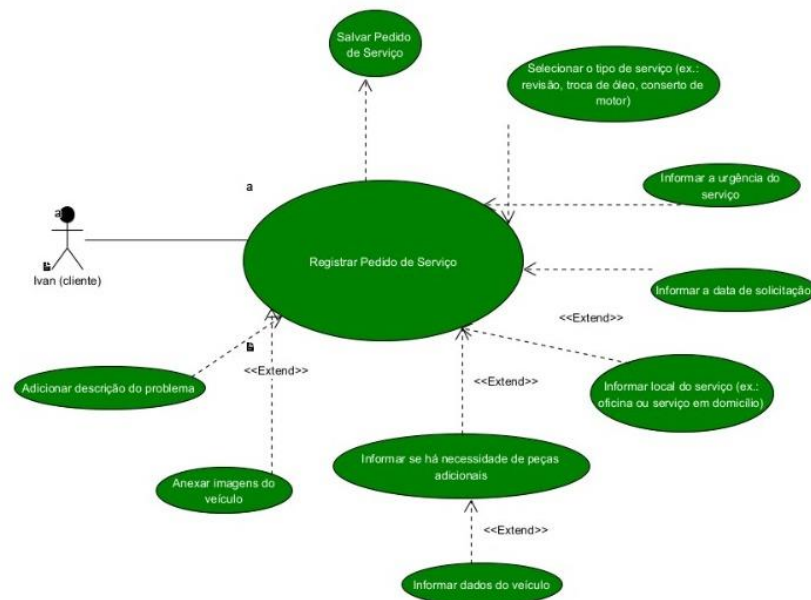
3.1 CASOS DE USO

Este diagrama representa o caso de uso principal "Registrar Pedido de Serviço", detalhando os possíveis cenários e interações envolvidos. Nele inclui:

1. Ator principal: o usuário "Ivan (cliente)", que interage com o sistema.
2. Extensões: elementos que representam ações adicionais ou opcionais ao fluxo principal. Assim como:
 - Adicionar descrição do problema.
 - Anexar imagens do veículo.
 - Informar necessidades de peças adicionais.
 - Informar local do serviço (se será na oficina ou em domicílio).
 - Informar dados do veículo.
3. Conexões: as linhas pontilhadas com a notação `<<extend>>` indicam que estas ações são extensões, ou seja, podem ser executadas dependendo da necessidade do cliente ou da situação.

Este diagrama reflete o processo de registro de um pedido de serviço e inclui várias opções que o cliente pode preencher, dependendo das circunstâncias.

Diagrama 1: Diagrama de caso de uso geral



Fonte: Os autores (2024).

Este diagrama tem um escopo mais amplo e abrange os seguintes atores e funcionalidades:

Atores:

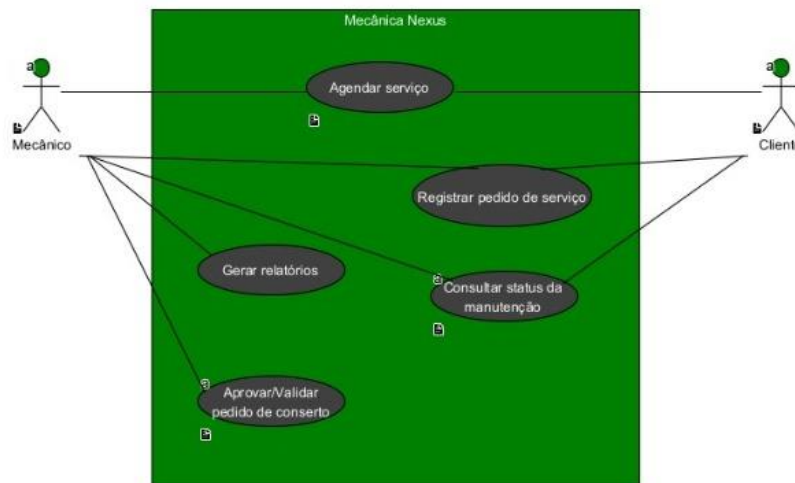
- Mecânico: Responsável por gerenciar serviços e validar documentos.
- Cliente: Pode registrar pedidos, consultar o status da manutenção, etc.

Casos de Uso:

- Agendar serviço.
- Registrar pedido de serviço.
- Consultar status da manutenção.
- Gerar relatórios.
- Aprovar/validar emissão de orçamento.

Este diagrama apresenta uma visão mais integrada do sistema, mostrando como o cliente e o mecânico interagem com funcionalidades distintas.

Diagrama 2: Diagrama de caso de uso.



Fonte: Os autores(2024).

3.2 SEQUENCIA

O Diagrama de Sequência apresentado representa a interação entre três elementos principais: Usuário, Sistema e Banco de Dados.

3.2.1 Usuário:

- O usuário inicia o processo acessando o sistema.
- Realiza o **Registro de Cadastro**, inserindo informações no sistema.

Sistema:

- Recebe as informações do usuário e as processa.
- Solicita a **Descrição do Serviço**.
- Solicita os **Dados Obrigatórios** e os **Defeitos do Veículo** para prosseguir com o registro.

3.2.2 Interação com o Banco de Dados:

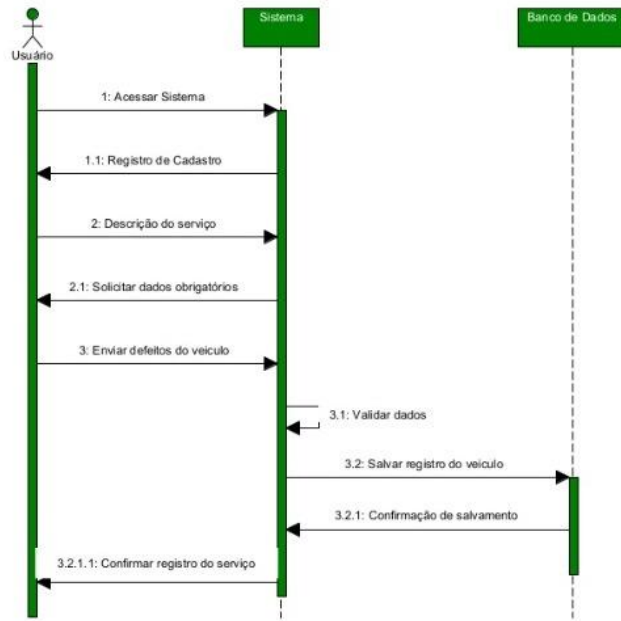
- O sistema valida os dados enviados pelo usuário (**3.1 Validar Dados**).
- Após a validação, salva o registro do veículo no banco de dados (**3.2 Salvar Registro do Veículo**).
- Confirma o salvamento das informações (**3.2.1 Confirmação do Salvamento**) e retorna a confirmação para o usuário.

3.2.3 Confirmação final:

- O sistema envia uma confirmação ao usuário sobre o registro do serviço solicitado.

Este diagrama mostra como as informações fluem entre o usuário, o sistema e o banco de dados. Ele evidencia a comunicação em tempo real para garantir que os dados sejam validados e registrados corretamente, concluindo o processo de forma segura e eficiente.

Diagrama 3: Diagrama de Sequência



Fonte: Os autores (2024).

3.3 ESTADO

Esse é um Diagrama de Estado, utilizado para representar os estados pelos quais uma entidade ou sistema pode passar durante seu ciclo de vida, com foco nas transições entre esses estados baseadas em eventos ou condições.

3.3.1 Estado Inicial:

- Representado pelo círculo preto, indica o ponto de partida do processo: "Iniciando o Registro".

3.3.2 Estados do Processo:

- "Descrevendo dados do veículo": O usuário insere informações sobre o veículo.
- "Dados informados": O sistema reconhece que os dados foram preenchidos.
- "Validando internamente os dados": O sistema realiza verificações automáticas para garantir a consistência e integridade das informações fornecidas.
- "Dados validados": Estado intermediário onde os dados podem ser aprovados ou não.

3.3.3 Decisão (Condicional):

- Após a validação, ocorre um desvio (representado pelo losango) com duas possibilidades:
 - Sim: Os dados estão corretos, e o processo avança para o estado "Registrado a Ordem de Serviço".
 - Não: Os dados apresentam erros, e o sistema entra no estado "Mostrando erro".

3.3.4 Estados Alternativos:

- "Revalidando dados": Se ocorrerem erros, o sistema volta ao estado de correção, permitindo ao usuário revisar as informações.

3.3.5 Estado Final:

- "Registrado a Ordem de Serviço": Representado pelo círculo preto com um contorno, indica o encerramento bem-sucedido do processo.

Este diagrama reflete o fluxo do registro de uma ordem de serviço, mostrando como o sistema gerencia o estado dos dados fornecidos. Ele é útil para:

- Identificar falhas: Mostra pontos críticos onde erros podem ocorrer (como a validação dos dados).
- Automação: Ilustra como o sistema valida e reage automaticamente aos eventos.
- Planejamento: Ajuda a projetar funcionalidades e identificar os estados necessários no desenvolvimento do sistema.

Diagrama 4: Diagrama de Estado



Fonte: Os autores (2024).

3.4 ATIVIDADE

Este é um Diagrama de Atividade, que descreve o fluxo de execução de um processo, destacando as ações, decisões e caminhos alternativos envolvidos. Ele é muito útil para representar processos de negócios ou fluxos dentro de sistemas.

3.4.1 Estado Inicial:

- Representado pelo círculo preto, indica o início do processo: "Acessar o sistema".

3.4.2 Etapas do Processo:

- "Selecionar uma das opções": O usuário escolhe o que deseja realizar no sistema.
- "Cadastro de Usuário / *Login*": O sistema verifica se o usuário está cadastrado e se pode prosseguir.
 - Se não estiver cadastrado, o processo é encerrado aqui (seta para fora do fluxo).
 - Se sim, o fluxo continua.
- "Validar se os dados informados estão corretos": O sistema verifica a validade das informações fornecidas.
 - Se não estiverem corretos, o fluxo retorna para o cadastro ou login.
 - Se sim, o processo avança.

3.4.3 Ação Principal:

- "Selecionar a opção 'Registro de Veículo'": O usuário escolhe registrar um veículo.
- "Preencher dados do veículo": O sistema permite ao usuário inserir as informações necessárias.

3.4.4 Decisão (Condicional):

- Após preencher os dados, ocorre uma validação (representada pelo losango):
 - Se os dados estão corretos, o processo avança para "Confirmação de Registro".
 - Se não estão corretos, o fluxo retorna para a etapa de preenchimento, permitindo ajustes

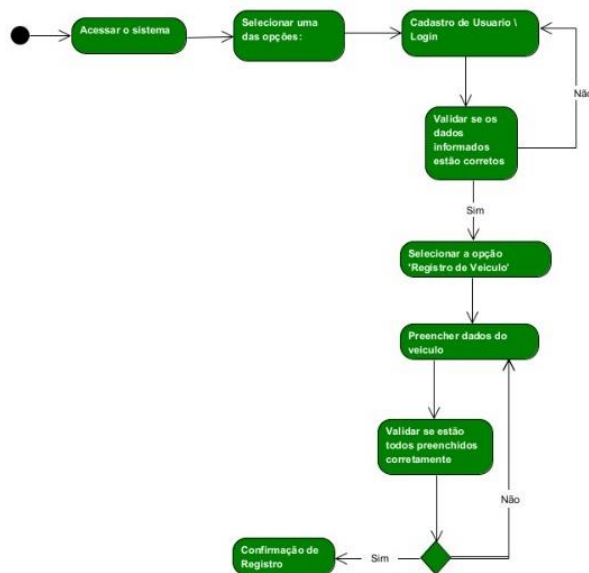
3.4.5 Estado Final:

- "Confirmação de Registro": Representado pelo círculo preto com contorno, indica o término bem-sucedido do processo.

Este diagrama é útil para:

- Descrever processos detalhados: Mostra todas as etapas que o usuário e o sistema executam.
- Identificar pontos críticos: Evidencia onde podem ocorrer falhas (validação de dados, *login*, etc.).
- Comunicar fluxo: Serve como uma representação visual clara para equipes de desenvolvimento, negócios ou outras partes interessadas.

Diagrama 5: Diagrama da Atividade



Fonte: Os autores(2024).

3.5 CLASSE

Este é um diagrama de classes representado no formato de um banco de dados relacional. Vou explicar cada uma das tabelas:

3.5.1 Usuário

- **Descrição:** Representa os usuários do sistema.
- **Campos:**
 - id (Chave primária): Identificador único do usuário.
 - usuario: Nome do usuário.
 - senha: Senha para autenticação.
 - perfil: Perfil ou tipo de usuário (ex.: administrador, cliente).

- estado: Estado atual do usuário no sistema (ex.: ativo, inativo).

3.5.2 Permissao_Usuario

- **Descrição:** Tabela associativa que relaciona permissões específicas a usuários.
- **Campos:**
 - PermissaoId (Chave estrangeira): Refere-se a uma permissão específica.
 - UsuarioId (Chave estrangeira): Refere-se a um usuário.

3.5.3 Permissao

- **Descrição:** Define os diferentes tipos de permissões que podem ser atribuídas.
- **Campos:**
 - id (Chave primária): Identificador único da permissão.
 - nome: Nome da permissão.
 - descricao: Detalhes sobre a permissão.

3.5.4 Agendamento

- **Descrição:** Representa agendamentos realizados no sistema.
- **Campos:**
 - id: Identificador único.
 - id_cliente: Relaciona-se a um cliente específico.
 - nome_veiculo: Nome ou modelo do veículo associado ao agendamento.
 - data_entrada: Data de início.
 - data_saida: Data prevista de saída.
 - id_serv: Serviço associado ao agendamento.
 - Outros campos específicos relacionados ao serviço.

3.5.5 Cliente

- **Descrição:** Contém informações dos clientes.
- **Campos:**
 - id: Identificador único.
 - cpf: CPF do cliente.
 - nome: Nome completo.
 - telefone: Contato do cliente.
 - email: Endereço de e-mail.
 - Outros campos, como endereço, cidade e detalhes adicionais.

3.5.6 Venda_rte

- **Descrição:** Representa informações de vendas realizadas.

- **Campos:**

- id: Identificador único.
- nome_cliente: Nome do cliente associado à venda.
- forma_pagamento: Método de pagamento utilizado.
- data_venda: Data em que a venda foi realizada.
- Outros campos relacionados ao produto, peça ou serviço vendido.

3.5.7 Fornecedor

- **Descrição:** Tabela que representa os fornecedores cadastrados.

- **Campos:**

- id: Identificador único.
- cnpj: Cadastro Nacional de Pessoa Jurídica do fornecedor.
- nome: Nome do fornecedor.
- Outros campos como endereço, telefone e cidade.

3.5.8 Estoque

- **Descrição:** Controla os produtos em estoque.

- **Campos:**

- id_prod (Chave primária): Identificador único do produto.
- codigo_prod: Código do produto.
- quantidade: Quantidade disponível no estoque.
- data_compra: Data de aquisição do produto.
- Outros campos relacionados ao estado ou status do produto.

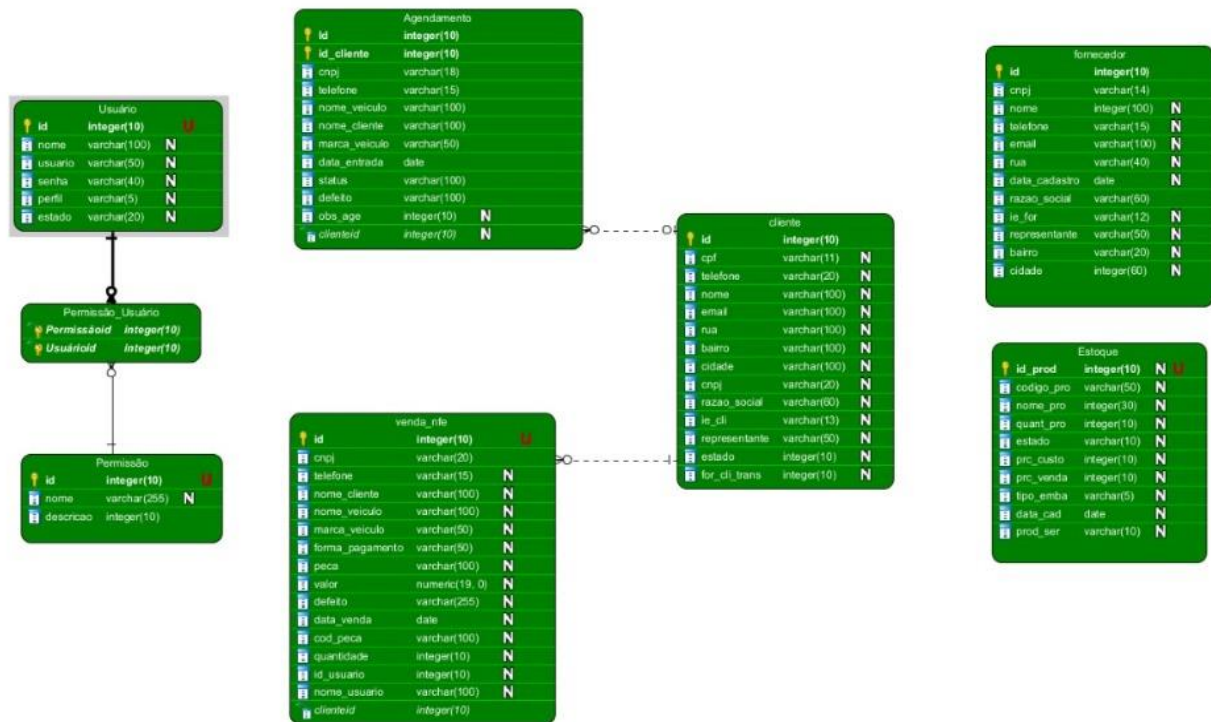
3.5.9 Relações Entre as Tabelas

- As tabelas estão conectadas através de **chaves primárias (PK)** e **chaves estrangeiras (FK)**.

Ao exemplo:

- id_cliente em **Agendamento** refere-se ao campo id na tabela **Cliente**.
- Permissao_usuario é uma tabela intermediária que conecta permissões com usuários.

Diagrama 6: Diagrama de Classes



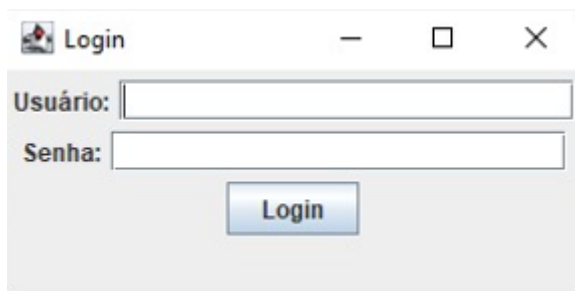
Fonte: Os autores(2024).

3.6 CODIGOS E INTERFACES

3.6.1 Interface de Login

O código cria uma interface gráfica em Java para login de usuários usando Swing. Ele possui campos para nome de usuário e senha, além de um botão "Login". Ao clicar no botão, as credenciais são validadas no banco de dados por meio da classe UsuarioDAO e da conexão ConexaoSQL.

Se as credenciais forem válidas, a tela de login é fechada, e um *listener* (definido por setLoginListener) pode abrir outra janela, como o painel principal do sistema. Caso contrário, é exibida uma mensagem de erro informando credenciais inválidas. O código segue uma estrutura modular, separando a interface gráfica, a lógica de controle e a interação com o banco de dados.



Código fonte:

```
package front;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JPasswordField;
import javax.swing.JTextField;

import modelo.dominio.dao.conexao.ConexaoSQL;
import modelo.dominio.dao.usuario.UsuarioDAO;

/**
 * Classe responsável por fornecer uma interface gráfica para login no sistema.
 * Valida as credenciais do usuário através de uma conexão com o banco de dados.
 */
```



```

public class LoginFrame extends JFrame {

    private static final long serialVersionUID = 1L;
    private JButton btnLogin;
    private JTextField txtUsername;
    private JPasswordField txtPassword;
    private Runnable loginListener; // Listener para notificar o sucesso no
login
    private UsuarioDAO usuarioDAO;

    /**
     * Construtor da classe `LoginFrame`.
     * Configura a interface gráfica para login e inicializa os componentes.
     */
    public LoginFrame() {
        setTitle("Login");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(300, 150);
        setLocationRelativeTo(null);

        // Criação dos componentes da tela de login
        txtUsername = new JTextField(20);
        txtPassword = new JPasswordField(20);
        btnLogin = new JButton("Login");

        // Conexão com o banco de dados
        ConexaoSQL conexao = new ConexaoSQL();
        usuarioDAO = new UsuarioDAO(conexao);

        // Adicionando os componentes na tela
        JPanel panel = new JPanel();
        panel.add(new JLabel("Usuário:"));
        panel.add(txtUsername);
        panel.add(new JLabel("Senha:"));
        panel.add(txtPassword);
        panel.add(btnLogin);

        add(panel);

        // Ação do botão de login
        btnLogin.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                String username = txtUsername.getText();
                String password = new String(txtPassword.getPassword());

                // Verifica as credenciais no banco de dados
                if (usuarioDAO.validarCredenciais(username, password)) {
                    if (loginListener != null) {

```

```

        loginListener.run(); // Chama o listener (abre o
MainFrame)
    }
    dispose(); // Fecha a tela de login
} else {
    JOptionPane.showMessageDialog(LoginFrame.this, "Usuário ou
senha inválidos.", "Erro de Login", JOptionPane.ERROR_MESSAGE);
}
}
});

setVisible(true);
}

/**
 * Define o listener que será chamado quando o login for bem-sucedido.
 *
 * @param listener Runnable que será executado após o sucesso no login.
 */
public void setLoginListener(Runnable listener) {
    this.loginListener = listener;
}
}

```

3.6.2 Interface De Cadastro De Cliente

As interfaces apresentadas são responsáveis pelo cadastro de clientes em um sistema. Elas fornecem um ambiente gráfico para que o usuário insira informações como CPF/CNPJ,

telefone, nome, e-mail, rua, bairro e cidade. Após preencher os campos, o usuário pode salvar ou cancelar o cadastro.

A interface utiliza a biblioteca Swing e possui botões de ação com as seguintes funcionalidades:

1. Salvar: Quando clicado, o sistema solicita ao usuário que escolha se está cadastrando um cliente com CPF ou CNPJ, e realiza a validação dos dados:

- Validação de campos: Verifica se todos os campos foram preenchidos. Caso contrário, exibe uma mensagem de erro pedindo o preenchimento completo.
- CPF ou CNPJ: O sistema valida a quantidade de dígitos (11 para CPF e 14 para CNPJ) para garantir dados consistentes.
- Telefone: Valida se o número de telefone tem 10 ou 11 dígitos, ignorando caracteres não numéricos.

Após validações bem-sucedidas, os dados são enviados para o banco de dados usando uma conexão SQL. Se houver sucesso no cadastro, uma mensagem de confirmação é exibida, e a janela é fechada. Em caso de erro no banco de dados, uma mensagem de erro é mostrada.

2. Cancelar: Encerra a janela sem salvar os dados.

Essas interfaces são úteis para facilitar a entrada e validação de dados, garantindo que clientes sejam registrados corretamente no banco de dados e reduzindo possíveis erros de digitação ou formato.

Cadastro de Cliente

CNPJ/CPF:

Telefone:

Nome:

Email:

Rua:

Bairro:

Cidade:

Salvar Cancelar

Cadastro de Cliente

CNPJ/CPF:

Telefone:

Nome:

Email:

Rua:

Bairro:

Cidade:

Escolha o tipo de cliente

Você está cadastrando um:

Cliente (CPF) Cliente (CNPJ) Cancelar

Salvar Cancelar

Cadastro de Cliente

CNPJ/CPF:

Telefone:

Nome:

Email:

Rua:

Bairro:

Cidade:

Erro

Preencha todos os campos!

OK

Salvar Cancelar

Código fonte:

```
package front;

import java.awt.*;
import java.sql.Connection;
import java.sql.SQLException;

import javax.swing.*;
```

```

import modelo.dominio.dao.conexao.Conexao;

/**
 * Classe responsável por fornecer uma interface gráfica para o cadastro de
 * clientes.
 * Permite que o usuário insira informações de cliente (CPF/CNPJ, nome, endereço,
 * telefone, etc.)
 * e as armazene no banco de dados.
 */
public class ClienteFrame extends JFrame {

    private static final long serialVersionUID = 7125715240622945208L;
    private JTextField txtCnpj, txtTelefone, txtNome, txtEmail, txtRua,
    txtBairro, txtCidade;
    private Conexao conexao;

    /**
     * Construtor da classe `ClienteFrame`.
     * Inicializa a interface gráfica e seus componentes.
     *
     * @param conexao Objeto de conexão com o banco de dados.
     */
    public ClienteFrame(Conexao conexao) {
        this.conexao = conexao;

        setTitle("Cadastro de Cliente");
        setSize(600, 400); // Dimensões ajustadas
        setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        setLocationRelativeTo(null);

        // Painei principal
        JPanel mainPanel = new JPanel();
        mainPanel.setLayout(new GridLayout(8, 2, 10, 10)); // Alinhar os campos
        mainPanel.setBackground(Color.decode("#C0C0C0"));

        // Labels e campos de entrada
        JLabel lblCnpj = new JLabel("CNPJ/CPF:");
        txtCnpj = new JTextField();
        JLabel lblTelefone = new JLabel("Telefone:");
        txtTelefone = new JTextField();
        JLabel lblNome = new JLabel("Nome:");
        txtNome = new JTextField();
        JLabel lblEmail = new JLabel("Email:");
        txtEmail = new JTextField();
        JLabel lblRua = new JLabel("Rua:");
        txtRua = new JTextField();
        JLabel lblBairro = new JLabel("Bairro:");
        txtBairro = new JTextField();
    }
}

```

```

JLabel lblCidade = new JLabel("Cidade:");
txtCidade = new JTextField();

// Adicionar componentes ao painel
mainPanel.add(lblCnpj);
mainPanel.add(txtCnpj);
mainPanel.add(lblTelefone);
mainPanel.add(txtTelefone);
mainPanel.add(lblNome);
mainPanel.add(txtNome);
mainPanel.add(lblEmail);
mainPanel.add(txtEmail);
mainPanel.add(lblRua);
mainPanel.add(txtRua);
mainPanel.add(lblBairro);
mainPanel.add(txtBairro);
mainPanel.add(lblCidade);
mainPanel.add(txtCidade);

// Painel para botões
JPanel buttonPanel = new JPanel();
buttonPanel.setLayout(new FlowLayout(FlowLayout.CENTER, 20, 10));
buttonPanel.setBackground(Color.decode("#C0C0C0"));

// Botão Salvar (Verde)
JButton btnSalvar = new JButton("Salvar");
btnSalvar.setBackground(Color.decode("#2E8B57"));
btnSalvar.setForeground(Color.WHITE);
btnSalvar.setFont(new Font("Arial", Font.BOLD, 14));
btnSalvar.setPreferredSize(new Dimension(120, 40)); // Botão maior
btnSalvar.setFocusPainted(false);

// Botão Cancelar (Vermelho)
JButton btnCancelar = new JButton("Cancelar");
btnCancelar.setBackground(Color.RED);
btnCancelar.setForeground(Color.WHITE);
btnCancelar.setFont(new Font("Arial", Font.BOLD, 14));
btnCancelar.setPreferredSize(new Dimension(120, 40)); // Botão maior
btnCancelar.setFocusPainted(false);
btnCancelar.addActionListener(e -> dispose()); // Fecha a janela

// Adicionar ação ao botão Salvar
btnSalvar.addActionListener(e -> {
    String[] opcoes = { "Cliente (CPF)", "Cliente (CNPJ)", "Cancelar"
};

    int escolha = JOptionPane.showOptionDialog(this, "Você está
cadastrando um:", "Escolha o tipo de cliente",
        JOptionPane.DEFAULT_OPTION, JOptionPane.INFORMATION_MESSAGE,
        null, opcoes, opcoes[0]);

```

```

        if (escolha == 0) {
            salvarCliente("CPF");
        } else if (escolha == 1) {
            salvarCliente("CNPJ");
        } else {
            JOptionPane.showMessageDialog(this, "Operação cancelada.",
"Cancelar", JOptionPane.INFORMATION_MESSAGE);
        }
    });

    // Adicionar botões ao painel
    buttonPanel.add(btnSalvar);
    buttonPanel.add(btnCancelar);

    // Adicionar painéis à janela
    add(mainPanel, BorderLayout.CENTER);
    add(buttonPanel, BorderLayout.SOUTH);

    setVisible(true);
}

/**
 * Salva as informações de um cliente no banco de dados.
 *
 * @param tipoCliente O tipo do cliente, podendo ser "CPF" ou "CNPJ".
 */
private void salvarCliente(String tipoCliente) {
    try {
        // Capturar dados dos campos
        String identificador = txtCnpj.getText().trim(); // Para CPF ou CNPJ
        String telefoneStr = txtTelefone.getText().trim();
        String nome = txtNome.getText().trim();
        String email = txtEmail.getText().trim();
        String rua = txtRua.getText().trim();
        String bairro = txtBairro.getText().trim();
        String cidade = txtCidade.getText().trim();

        // Validação básica
        if (identificador.isEmpty() || telefoneStr.isEmpty() ||
nome.isEmpty() || email.isEmpty() || rua.isEmpty()
|| bairro.isEmpty() || cidade.isEmpty()) {
            JOptionPane.showMessageDialog(this, "Preencha todos os campos!",
"Erro", JOptionPane.ERROR_MESSAGE);
            return;
        }

        // Validar CPF ou CNPJ
        if (tipoCliente.equals("CPF") && identificador.length() != 11) {

```

```

        JOptionPane.showMessageDialog(this, "CPF deve ter 11 dígitos!",
"Erro", JOptionPane.ERROR_MESSAGE);
        return;
    } else if (tipoCliente.equals("CNPJ") && identificador.length() !=
14) {
        JOptionPane.showMessageDialog(this, "CNPJ deve ter 14 dígitos!",
"Erro", JOptionPane.ERROR_MESSAGE);
        return;
    }

    // Validar e formatar o telefone (remover caracteres não numéricos)
    String telefoneFormatado = telefoneStr.replaceAll("[^0-9]", "");
    if (telefoneFormatado.length() < 10 || telefoneFormatado.length() >
11) {
        JOptionPane.showMessageDialog(this, "Telefone inválido! Deve
ter 10 ou 11 dígitos.", "Erro",
        JOptionPane.ERROR_MESSAGE);
        return;
    }

    try (Connection conn = conexao.obterConexao()) {
        String inserirSQL = "INSERT INTO cliente (cnpj, telefone, nome,
email, rua, bairro, cidade) VALUES (?, ?, ?, ?, ?, ?, ?)";
        var stmt = conn.prepareStatement(inserirSQL);
        stmt.setString(1, identificador);
        stmt.setString(2, telefoneFormatado);
        stmt.setString(3, nome);
        stmt.setString(4, email);
        stmt.setString(5, rua);
        stmt.setString(6, bairro);
        stmt.setString(7, cidade);
        stmt.executeUpdate();
        conn.commit();
        JOptionPane.showMessageDialog(this, "Cliente cadastrado com
sucesso!", "Sucesso",
        JOptionPane.INFORMATION_MESSAGE);
        dispose();
    } catch (SQLException ex) {
        JOptionPane.showMessageDialog(this, "Erro ao salvar no banco: "
+ ex.getMessage(), "Erro",
        JOptionPane.ERROR_MESSAGE);
    }
    } catch (NumberFormatException ex) {
        JOptionPane.showMessageDialog(this, "CNPJ ou Telefone inválidos!",
"Erro", JOptionPane.ERROR_MESSAGE);
    }
}
}

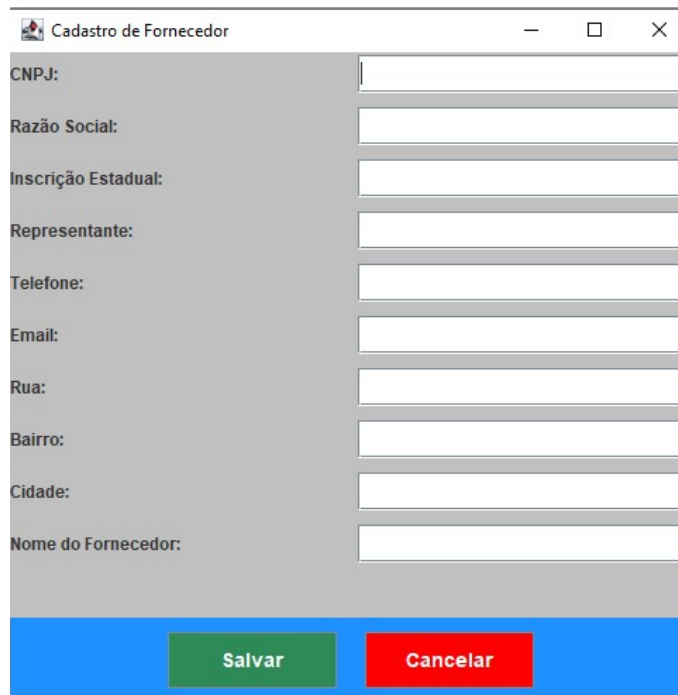
```


3.6.3 INTERFACE DE CADASTRO DE FORNECEDOR

O sistema de cadastro de fornecedores tem como objetivo permitir o registro organizado de informações de fornecedores em um banco de dados, utilizando uma interface gráfica e uma lógica implementada em Java. A interface apresenta campos para coletar dados essenciais, como CNPJ, razão social, inscrição estadual, representante, telefone, e-mail, endereço (composto por rua, bairro e cidade) e o nome do fornecedor. Há também botões para que o usuário possa salvar os dados no banco de dados ou cancelar o cadastro.

A lógica do sistema é implementada na classe `CadastroFornecedor`, que é responsável por gerenciar a interação com o usuário e a persistência dos dados no banco. O método principal, `coletarDadosFornecedor()`, guia o usuário na inserção das informações e realiza validações, como verificar se o CNPJ possui 14 dígitos ou se o telefone está no formato correto. Caso o usuário opte por confirmar o cadastro, os dados do fornecedor são encapsulados em um objeto e enviados ao banco de dados pelo método `inserirFornecedor()`, que utiliza uma conexão SQL para realizar a inserção. Além disso, o sistema oferece a opção de cancelar o cadastro a qualquer momento.

O código é projetado para lidar com erros comuns, como entradas inválidas (exemplo: CNPJ ou telefone fora do formato esperado) ou falhas na conexão com o banco de dados. Assim, ele assegura que apenas dados consistentes e válidos sejam armazenados. Em resumo, o sistema proporciona uma solução eficiente para gerenciar fornecedores, combinando validações, segurança e persistência de dados.



Cadastro de Fornecedor

CNPJ:

Razão Social:

Inscrição Estadual:

Representante:

Telefone:

Email:

Rua:

Bairro:

Cidade:

Nome do Fornecedor:

Salvar Cancelar

Código Fonte:

```
package modelo.dominio.dao.conexao;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.Scanner;

import gestao.Fornecedor;

/**
 * Classe responsável por gerenciar o cadastro de fornecedores no sistema.
 * Permite a coleta de informações de fornecedores e a inserção dessas
 * informações no banco de dados.
 */
public class CadastroFornecedor {

    /**
     * Conexão com o banco de dados.
     */
    private Conexao conexao;

    /**
     * Scanner para coleta de entradas do usuário.
     */
    private Scanner scanner;
```

```

/**
 * Construtor da classe que inicializa a conexão e o scanner.
 *
 * @param conexao a instância de conexão com o banco de dados
 */
public CadastroFornecedor(Conexao conexao) {
    this.conexao = conexao;
    this.scanner = new Scanner(System.in);
}

/**
 * Solicita uma entrada do usuário com base em uma mensagem exibida no
console.
 * Permite o retorno ao menu principal se o usuário digitar "voltar".
 *
 * @param mensagem a mensagem exibida ao usuário
 * @return a entrada do usuário ou {@code null} se o usuário optar por voltar
 */
private String solicitarEntrada(String mensagem) {
    System.out.print(mensagem + " (ou digite 'voltar' para retornar): ");
    String entrada = scanner.nextLine();
    if ("voltar".equalsIgnoreCase(entrada)) {
        System.out.println("Retornando ao menu principal...");
        return null;
    }
    return entrada;
}

/**
 * Coleta os dados do fornecedor a partir das entradas do usuário e realiza
a validação das informações.
 * Permite o cancelamento ou confirmação do cadastro.
 */
public void coletarDadosFornecedor() {
    while (true) {
        try {
            // Coleta e validação de CNPJ
            String cnpjInput = solicitarEntrada("Digite o CNPJ:");
            if (cnpjInput == null) return;
            cnpjInput = cnpjInput.replaceAll("[^0-9]", "");
            if (cnpjInput.length() != 14) {
                System.out.println("CNPJ inválido! Deve ter 14 dígitos.");
                continue;
            }
            long cnpj = Long.parseLong(cnpjInput);

            // Coleta de outros dados do fornecedor
            String razaoSocial = solicitarEntrada("Digite a Razão Social:");
            if (razaoSocial == null) return;

```

```

        String inscricaoEstadual = solicitarEntrada("Digite a Inscrição Estadual:");
        if (inscricaoEstadual == null) return;

        String representante = solicitarEntrada("Digite o Representante:");
        if (representante == null) return;

        String telefoneInput = solicitarEntrada("Digite o Telefone:");
        if (telefoneInput == null) return;
        telefoneInput = telefoneInput.replaceAll("[^0-9]", "");
        if (telefoneInput.length() < 10 || telefoneInput.length() > 11)
        {
            System.out.println("Número de telefone inválido! Deve ter entre 10 e 11 dígitos.");
            continue;
        }
        long telefone = Long.parseLong(telefoneInput);

        String email = solicitarEntrada("Digite o Email:");
        if (email == null) return;

        String rua = solicitarEntrada("Digite a Rua:");
        if (rua == null) return;

        String bairro = solicitarEntrada("Digite o Bairro:");
        if (bairro == null) return;

        String cidade = solicitarEntrada("Digite a Cidade:");
        if (cidade == null) return;

        String nome = solicitarEntrada("Digite o Nome do Fornecedor:");
        if (nome == null) return;

        // Criação do objeto Fornecedor
        Fornecedor fornecedor = new Fornecedor(cnpj, razaoSocial, inscricaoEstadual, representante, telefone, email, rua, bairro, cidade, nome);

        // Confirmação do cadastro
        System.out.println("Você deseja: ");
        System.out.println("1. Cadastrar");
        System.out.println("2. Cancelar");
        System.out.print("Escolha uma opção (1 ou 2): ");
        int opcao = Integer.parseInt(scanner.nextLine());

        switch (opcao) {
            case 1:

```

```

        try (Connection conn = conexao.obterConexao()) {
            inserirFornecedor(fornecedor, conn);
            System.out.println("Fornecedor cadastrado com
sucesso!");
        } catch (SQLException e) {
            System.out.println("Erro ao cadastrar o fornecedor:
" + e.getMessage());
        }
        return;

    case 2:
        System.out.println("Cadastro cancelado.");
        return;

    default:
        System.out.println("Opção inválida. Tente novamente.");
        break;
    }
} catch (NumberFormatException e) {
    System.out.println("Entrada inválida. Por favor, tente
novamente.");
}
}

/**
 * Insere os dados do fornecedor no banco de dados.
 *
 * @param fornecedor o objeto fornecedor a ser cadastrado
 * @param conn      a conexão com o banco de dados
 * @throws SQLException se ocorrer um erro ao executar a operação no banco
de dados
 */
public void inserirFornecedor(Fornecedor fornecedor, Connection conn) throws
SQLException {
    String inserirSQL = "INSERT INTO fornecedor (cnpj, razao_social, ie_for,
representante, telefone, email, rua, bairro, cidade, nome) VALUES (?, ?, ?, ?,
?, ?, ?, ?, ?, ?)";

    try (PreparedStatement stmt = conn.prepareStatement(inserirSQL)) {
        stmt.setLong(1, fornecedor.getCnpj());
        stmt.setString(2, fornecedor.getRazaoSocial());
        stmt.setString(3, fornecedor.getInscricaoEstadual());
        stmt.setString(4, fornecedor.getRepresentante());
        stmt.setLong(5, fornecedor.getTelefone());
        stmt.setString(6, fornecedor.getEmail());
        stmt.setString(7, fornecedor.getRua());
        stmt.setString(8, fornecedor.getBairro());
        stmt.setString(9, fornecedor.getCidade());
    }
}

```

```

        stmt.setString(10, fornecedor.getNome());

        int rowsAffected = stmt.executeUpdate();
        conn.commit();
        System.out.println("Linhas inseridas: " + rowsAffected);
    }
}
}

```

3.6.4 INTERFACE DE BACKUP DO BANCO DE DADOS

Esta interface tem como objetivo facilitar o processo de *backup* do banco de dados PostgreSQL, permitindo que o usuário escolha o diretório onde o arquivo de *backup* será salvo e execute o procedimento de forma prática. A aplicação é implementada utilizando o *framework* *Swing*, que fornece os componentes visuais da interface.

Ao abrir o programa, o usuário encontra uma janela com uma caixa de texto que exibe o caminho do diretório onde o *backup* será armazenado, dois botões principais e um *layout* visualmente organizado:

1. **Botão "Escolher Diretório":** Quando pressionado, exibe um diálogo para que o usuário selecione a pasta de destino para salvar o *backup*. Após a escolha, o caminho do diretório selecionado é exibido na caixa de texto.
2. **Botão "Realizar Backup":** Quando acionado, executa o processo de *backup* do banco de dados, utilizando a classe *BackupPostgres*. Essa classe contém a lógica para gerar o arquivo de *backup* e salvá-lo no local indicado.

O código está estruturado para garantir facilidade de uso e personalização. Por exemplo, o diretório padrão de *backup* é exibido na interface, e o processo de seleção do

diretório é guiado pelo diálogo gráfico, evitando a necessidade de inserção manual de caminhos. Após selecionar o diretório e iniciar o *backup*, o programa atualiza as configurações da classe responsável e executa o processo automaticamente.

Essa interface, portanto, simplifica a criação de *backups* do banco de dados, oferecendo uma solução amigável e eficiente para usuários que precisam realizar cópias de segurança sem lidar diretamente com comandos técnicos.



Código fonte:

```
package front;

import modelo.dominio.dao.conexao.BackupPostgres;

import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

/**
 * Classe responsável por fornecer uma interface gráfica para realizar o backup
 * do banco de dados PostgreSQL.
 * Permite que o usuário escolha um diretório de destino para o arquivo de backup
 * e execute o processo de backup.
 */
public class BackupPostgresUI extends JFrame {

    private static String backupDir = ""; // Diretório padrão para backup
    private JTextField directoryField;
    private JButton chooseDirectoryButton;
    private JButton backupButton;

    /**
     * Método principal para iniciar a aplicação de backup.
     */
}
```

```

    * @param args Argumentos da linha de comando (não utilizados).
    */
    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> {
            new BackupPostgresUI().setVisible(true);
        });
    }

    /**
     * Construtor da classe `BackupPostgresUI`.
     * Configura a interface gráfica e inicializa os componentes necessários
     para o processo de backup.
     */
    public BackupPostgresUI() {
        setTitle("Backup do Banco de Dados");
        setSize(400, 250);
        setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE); // Fecha apenas a
janela de backup
        setLocationRelativeTo(null);

        // Criar o painel de layout
        JPanel panel = new JPanel();
        panel.setLayout(new FlowLayout());
        panel.setBackground(Color.DARK_GRAY);

        // Caixa de texto para exibir o diretório de backup
        directoryField = new JTextField(30);
        directoryField.setEditable(false);
        directoryField.setBackground(Color.WHITE);
        directoryField.setText(backupDir);

        // Botão para escolher o diretório
        chooseDirectoryButton = new JButton("Escolher Diretório");
        chooseDirectoryButton.setBackground(new Color(76, 175, 80)); // Cor
verde
        chooseDirectoryButton.setForeground(Color.WHITE);

        // Botão para realizar o backup
        backupButton = new JButton("Realizar Backup");
        backupButton.setBackground(new Color(33, 150, 243)); // Cor azul
        backupButton.setForeground(Color.WHITE);

        // Adicionar os componentes ao painel
        panel.add(directoryField);
        panel.add(chooseDirectoryButton);
        panel.add(backupButton);

        // Adicionar o painel à janela
        add(panel);
    }

```



```

        // Configurar ação para escolher o diretório
        configureChooseDirectoryAction();

        // Configurar ação para realizar o backup
        configureBackupAction();
    }

    /**
     * Configura a ação para o botão "Escolher Diretório".
     * Abre um diálogo para que o usuário selecione o diretório onde o backup
     será salvo.
     */
    private void configureChooseDirectoryAction() {
        chooseDirectoryButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                JFileChooser directoryChooser = new JFileChooser();
                directoryChooser.setDialogTitle("Escolher diretório para
backup");
                directoryChooser.setFileSelectionMode(JFileChooser.DIRECTORIES
_ONLY);
                directoryChooser.setCurrentDirectory(new
java.io.File(backupDir));

                int result =
directoryChooser.showOpenDialog(BackupPostgresUI.this);
                if (result == JFileChooser.APPROVE_OPTION) {
                    backupDir =
directoryChooser.getSelectedFile().getAbsolutePath() + "\\";
                    directoryField.setText(backupDir);
                }
            }
        });
    }

    /**
     * Configura a ação para o botão "Realizar Backup".
     * Atualiza o diretório de backup e executa o método de backup no objeto
`BackupPostgres`.
     */
    private void configureBackupAction() {
        backupButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                // Atualizar diretório de backup e realizar o processo
                BackupPostgres.BACKUP_DIR = backupDir;
                BackupPostgres.realizarBackup();
            }
        });
    }

```

```

    });
}
}

```

3.6.5 Interface De Notas Fiscais

A interface apresentada é uma aplicação desenvolvida em Java com o framework Swing, voltada para a realização de vendas associadas a Notas Fiscais Eletrônicas (NF-e). Ela visa facilitar o registro de vendas, integração com o banco de dados e atualização de informações relacionadas ao estoque e agendamentos de serviços.

3.6.5.1 Objetivo da Interface

A interface permite que um operador registre os detalhes de uma venda, incluindo a venda de peças associadas a um agendamento de cliente. Além disso, a aplicação calcula o valor total da venda, atualiza o estoque e registra todas as informações no banco de dados.

3.6.5.2 Funcionalidades da Interface

3.6.5.2.1 Campos de Entrada:

- ID do Agendamento: Identifica o serviço ou atendimento associado à venda.
- Forma de Pagamento: Informa como o cliente efetuará o pagamento (por exemplo, cartão, dinheiro, etc.).
- Defeito Solucionado: Descreve o defeito resolvido no atendimento, caso aplicável.
- Venda de Peça: Indica se a venda inclui peças adicionais (*checkbox* para marcar ou desmarcar).
- Código da Peça: Insere o código da peça que será vendida.
- Quantidade: Define a quantidade de peças que o cliente deseja adquirir.

3.6.5.2.2 Botões de Ação:

- Adicionar Produto: Adiciona a peça selecionada à lista de itens da venda. Verifica o estoque e calcula o valor total com base no preço e na quantidade do produto selecionado.
- Confirmar Venda: Finaliza o processo de venda. Registra os detalhes no banco de dados, atualiza o status do agendamento, desconta as quantidades vendidas do estoque e exibe mensagens de sucesso ou erro.

3.6.5.2.3 Valor Total da Venda:

- Exibe o valor total acumulado dos itens incluídos na venda. Este valor é atualizado automaticamente toda vez que um novo item é adicionado.

3.6.5.3 Processo Interno

3.6.5.3.1 Busca de Dados no Banco de Dados:

- Verifica o agendamento associado à venda para obter informações do cliente.
- Consulta o estoque para confirmar a existência do produto e buscar seu preço.

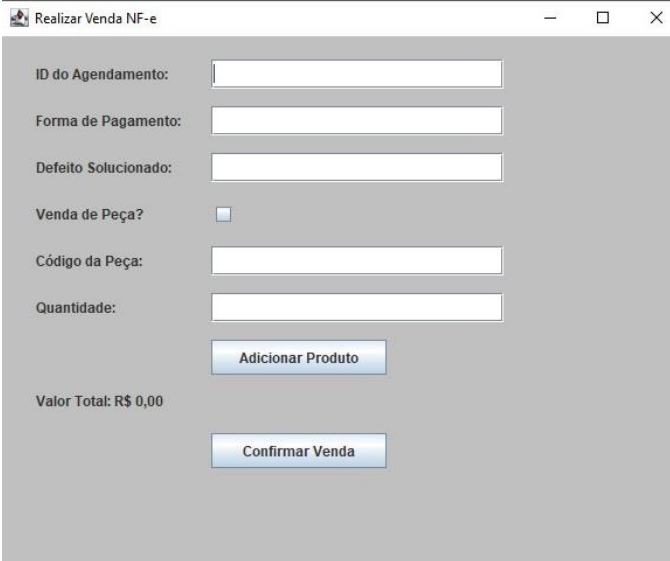
3.6.5.3.2 Registro da Venda:

- Insere os detalhes da venda na tabela correspondente no banco de dados (venda_nfe).
- Atualiza os detalhes dos produtos vendidos, como código da peça, quantidade e preço.
- Reduz a quantidade das peças vendidas do estoque e atualiza o status do agendamento como "finalizado".

3.6.5.4 Cenário de Uso

Este sistema é ideal para empresas que precisam integrar o processo de vendas com o gerenciamento de estoque e serviços de atendimento. Por exemplo, oficinas mecânicas que realizam serviços em veículos e frequentemente vendem peças como parte do reparo podem usar essa interface para registrar suas operações de forma eficiente e precisa.

Em resumo, a interface automatiza e organiza o processo de vendas, garantindo que todas as etapas sejam registradas corretamente, reduzindo a possibilidade de erros e aumentando a eficiência operacional.



A interface "Realizar Venda NF-e" apresenta os seguintes elementos:

- Campos de entrada: ID do Agendamento, Forma de Pagamento, Defeito Solucionado, Código da Peça, Quantidade.
- Botão: "Adicionar Produto".
- Botão: "Confirmar Venda".
- Indicador: "Venda de Peça?" com uma caixa de seleção.
- Exibição: "Valor Total: R\$ 0,00".

```

package front;

import java.awt.Color;
import java.awt.event.ActionEvent;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;

import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JTextField;

import gestao.venda.VendaNFE;
import modelo.dominio.dao.conexao.Conexao;

public class VendaNFEFrame extends JFrame {

    private static final long serialVersionUID = 1;
    private JTextField txtIdAgendamento, txtFormaPagamento, txtDefeito,
txtCodPeca, txtQtdPeca;
    private JCheckBox chkVendaPeca;
    private Conexao conexao;
    private ArrayList<String> codigosProdutos;
    private ArrayList<Integer> quantidadesProdutos;
    private double valorTotal;

    public VendaNFEFrame(Conexao conexao) {
        this.conexao = conexao;
        codigosProdutos = new ArrayList<>();
        quantidadesProdutos = new ArrayList<>();
        valorTotal = 0.0;

        // Configurações básicas da janela
        setTitle("Realizar Venda NF-e");
        setSize(600, 500);
        setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        setLocationRelativeTo(null);

        // Painel principal
        JPanel mainPanel = new JPanel();
        mainPanel.setBackground(Color.decode("#C0C0C0")); // Cor de fundo cinza
claro
        mainPanel.setLayout(null);

```

```

// Labels e campos de entrada
JLabel lblIdAgendamento = new JLabel("ID do Agendamento:");
lblIdAgendamento.setBounds(30, 20, 150, 25);
mainPanel.add(lblIdAgendamento);

txtIdAgendamento = new JTextField();
txtIdAgendamento.setBounds(180, 20, 250, 25);
mainPanel.add(txtIdAgendamento);

JLabel lblFormaPagamento = new JLabel("Forma de Pagamento:");
lblFormaPagamento.setBounds(30, 60, 150, 25);
mainPanel.add(lblFormaPagamento);

txtFormaPagamento = new JTextField();
txtFormaPagamento.setBounds(180, 60, 250, 25);
mainPanel.add(txtFormaPagamento);

JLabel lblDefeito = new JLabel("Defeito Solucionado:");
lblDefeito.setBounds(30, 100, 150, 25);
mainPanel.add(lblDefeito);

txtDefeito = new JTextField();
txtDefeito.setBounds(180, 100, 250, 25);
mainPanel.add(txtDefeito);

JLabel lblVendaPeca = new JLabel("Venda de Peça?");
lblVendaPeca.setBounds(30, 140, 150, 25);
mainPanel.add(lblVendaPeca);

chkVendaPeca = new JCheckBox();
chkVendaPeca.setBounds(180, 140, 20, 25);
chkVendaPeca.setBackground(Color.decode("#C0C0C0")); // Mesma cor do
painel principal
chkVendaPeca.setOpaque(false); // Deixa o fundo do checkbox transparente
mainPanel.add(chkVendaPeca);

JLabel lblCodPeca = new JLabel("Código da Peça:");
lblCodPeca.setBounds(30, 180, 150, 25);
mainPanel.add(lblCodPeca);

txtCodPeca = new JTextField();
txtCodPeca.setBounds(180, 180, 250, 25);
mainPanel.add(txtCodPeca);

JLabel lblQtdPeca = new JLabel("Quantidade:");
lblQtdPeca.setBounds(30, 220, 150, 25);
mainPanel.add(lblQtdPeca);

```

```

txtQtdPeca = new JTextField();
txtQtdPeca.setBounds(180, 220, 250, 25);
mainPanel.add(txtQtdPeca);

// Botão para adicionar produto à venda
JButton btnAdicionarProduto = new JButton("Adicionar Produto");
btnAdicionarProduto.setBounds(180, 260, 150, 30);
mainPanel.add(btnAdicionarProduto);

// Exibir valor total da venda
JLabel lblValorTotal = new JLabel("Valor Total: R$ 0,00");
lblValorTotal.setBounds(30, 300, 250, 25);
mainPanel.add(lblValorTotal);

// Ação do botão Adicionar Produto
btnAdicionarProduto.addActionListener((ActionEvent e) -> {
    try {
        String codPeca = txtCodPeca.getText();
        int qtdPeca = Integer.parseInt(txtQtdPeca.getText());

        if (codPeca.isEmpty() || qtdPeca <= 0) {
            JOptionPane.showMessageDialog(this, "Por favor, insira um
código de peça válido e quantidade maior que 0.");
            return;
        }

        // Adicionar produto à lista de produtos e calcular valor total
        double valorProduto = buscarValorProduto(codPeca);
        if (valorProduto > 0) {
            codigosProdutos.add(codPeca);
            quantidadesProdutos.add(qtdPeca);
            valorTotal += valorProduto * qtdPeca;

            // Atualizar o valor total na tela
            lblValorTotal.setText("Valor Total: R$ " +
String.format("%.2f", valorTotal));
            txtCodPeca.setText("");
            txtQtdPeca.setText("");
        } else {
            JOptionPane.showMessageDialog(this, "Produto não encontrado
no estoque.");
        }

    } catch (NumberFormatException ex) {
        JOptionPane.showMessageDialog(this, "Por favor, insira uma
quantidade válida.");
    }
});

```

```

// Botão de Confirmar
JButton btnConfirmar = new JButton("Confirmar Venda");
btnConfirmar.setBounds(180, 340, 150, 30);
mainPanel.add(btnConfirmar);

// Ação do botão Confirmar
btnConfirmar.addActionListener((ActionEvent e) -> {
    try {
        int idAgendamento = Integer.parseInt(txtIdAgendamento.getText());
        String formaPagamento = txtFormaPagamento.getText();
        String defeito = txtDefeito.getText();

        if (formaPagamento.isEmpty()) {
            JOptionPane.showMessageDialog(this, "Por favor, informe a
forma de pagamento.");
            return;
        }

        // Criar objeto VendaNFE e realizar a venda
        VendaNFE vendaNFE = new VendaNFE(conexao);
        boolean sucesso = vendaNFE.realizarVenda(idAgendamento,
formaPagamento, defeito, codigosProdutos, quantidadesProdutos);

        if (sucesso) {
            JOptionPane.showMessageDialog(this, "Venda NF-e realizada
com sucesso!");
            dispose(); // Fecha a janela
        } else {
            JOptionPane.showMessageDialog(this, "Erro ao realizar a
venda.");
        }
    } catch (NumberFormatException ex) {
        JOptionPane.showMessageDialog(this, "ID do Agendamento
inválido.");
    }
});

// Adicionar painel à janela
add(mainPanel);
}

// Método para buscar o valor de uma peça no estoque
private double buscarValorProduto(String codPeca) {
    double valorProduto = 0.0;
    try (Connection conn = conexao.obterConexao();
        PreparedStatement stmt = conn.prepareStatement("SELECT prc_venda
FROM estoque WHERE codigo_pro = ?")) {

        stmt.setString(1, codPeca);

```

```

        try (ResultSet rs = stmt.executeQuery()) {
            if (rs.next()) {
                valorProduto = rs.getDouble("prc_venda");
            }
        }
    } catch (SQLException e) {
        System.out.println("Erro ao buscar o valor do produto: " +
e.getMessage());
    }
    return valorProduto;
}
}

```

```

package gestao.venda;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;

import modelo.dominio.dao.conexao.Conexao;

/**
 * Classe responsável por realizar o processo de venda e registrar os detalhes
da Nota Fiscal Eletrônica (NF-e).
 */

public class VendaNFE {
    private Conexao conexao;

    /**
     * Construtor que inicializa a classe com a conexão ao banco de dados.
     *
     * @param conexao Instância da classe {@link Conexao} para gerenciar a
conexão com o banco de dados.
     */
    public VendaNFE(Conexao conexao) {
        this.conexao = conexao;
    }

    /**
     * Realiza o processo completo de venda, registrando os dados da venda,
atualizando o estoque
     * e o status do agendamento.
     *
     * @param idAgendamento ID do agendamento associado à venda.
     * @param formaPagamento Forma de pagamento utilizada na venda.
     * @param defeito Defeito solucionado durante o atendimento.
     */
}

```



```

    * @param codigosProdutos    Lista de códigos dos produtos vendidos.
    * @param quantidadesProdutos Lista de quantidades correspondentes aos
produtos vendidos.
    * @return {@code true} se a venda for realizada com sucesso, caso contrário,
{@code false}.
    */
    public boolean realizarVenda(int idAgendamento, String formaPagamento,
String defeito,
                                ArrayList<String> codigosProdutos, ArrayList<Integer>
quantidadesProdutos) {
        Connection conn = null;
        PreparedStatement stmtAgendamento = null;
        PreparedStatement stmtVendaNFE = null;
        PreparedStatement stmtDetalheVendaNFE = null;
        PreparedStatement stmtAtualizaEstoque = null;
        PreparedStatement stmtAtualizaStatus = null;
        ResultSet rsAgendamento = null;
        /**
         * Busca informações sobre uma peça específica no estoque com base no código
informado.
         *
         * @param conn    Conexão com o banco de dados.
         * @param codPeca Código da peça a ser buscada.
         * @return Um array de objetos contendo:
         *         - Nome da peça (índice 0)
         *         - Preço de venda (índice 1)
         *         Retorna {@code null} se a peça não for encontrada.
         */
        double valorTotal = 0.0;

        try {
            conn = conexao.obterConexao();
            conn.setAutoCommit(false);

            // Buscar os dados do agendamento
            String sqlAgendamento = "SELECT id_cliente, cnpj, telefone, nome_cliente,
nome_veiculo, marca_veiculo FROM agendamento WHERE id = ?";
            stmtAgendamento = conn.prepareStatement(sqlAgendamento);
            stmtAgendamento.setInt(1, idAgendamento);
            rsAgendamento = stmtAgendamento.executeQuery();

            if (!rsAgendamento.next()) {
                System.out.println("Agendamento não encontrado.");
                return false;
            }

            // Recupera as informações do agendamento
            int idCliente = rsAgendamento.getInt("id_cliente");
            String cnpj = rsAgendamento.getString("cnpj");

```

```

String telefone = rsAgendamento.getString("telefone");
String nomeCliente = rsAgendamento.getString("nome_cliente");
String nomeVeiculo = rsAgendamento.getString("nome_veiculo");
String marcaVeiculo = rsAgendamento.getString("marca_veiculo");

// Inserir venda na tabela venda_nfe
String sqlVendaNFE = "INSERT INTO venda_nfe (id_cliente, cnpj, telefone,
nome_cliente, nome_veiculo, marca_veiculo, forma_pagamento, defeito) "
    + "VALUES (?, ?, ?, ?, ?, ?, ?, ?) RETURNING id"; // Usando
RETURNING para pegar o ID gerado
stmtVendaNFE = conn.prepareStatement(sqlVendaNFE);
stmtVendaNFE.setInt(1, idCliente);
stmtVendaNFE.setString(2, cnpj);
stmtVendaNFE.setString(3, telefone);
stmtVendaNFE.setString(4, nomeCliente);
stmtVendaNFE.setString(5, nomeVeiculo);
stmtVendaNFE.setString(6, marcaVeiculo);
stmtVendaNFE.setString(7, formaPagamento);
stmtVendaNFE.setString(8, defeito);

// Executando a inserção na tabela venda_nfe e obtendo o ID gerado
ResultSet rsVendaNFE = stmtVendaNFE.executeQuery();
int idVenda = -1;
if (rsVendaNFE.next()) {
    idVenda = rsVendaNFE.getInt("id");
}

if (idVenda == -1) {
    System.out.println("Falha ao registrar venda na tabela venda_nfe.");
    conn.rollback(); // Reverte transação em caso de falha
    return false;
} else {
    System.out.println("Venda registrada com sucesso na tabela
venda_nfe.");
}

// Novo PreparedStatement para atualizar os detalhes da venda_nfe
String sqlDetalheVendaNFE = "UPDATE venda_nfe SET cod_peca = ?, valor =
?, quantidade = ?, peca = ? WHERE id = ?";
stmtDetalheVendaNFE = conn.prepareStatement(sqlDetalheVendaNFE);

// Inicializa stmtAtualizaEstoque antes do loop
String sqlAtualizaEstoque = "UPDATE estoque SET quant_pro = quant_pro -
? WHERE codigo_pro = ?";
stmtAtualizaEstoque = conn.prepareStatement(sqlAtualizaEstoque);

for (int i = 0; i < codigosProdutos.size(); i++) {
    String codProduto = codigosProdutos.get(i);
    int quantidade = quantidadesProdutos.get(i);

```

```

        // Buscar o preço e nome da peça
        Object[] infoProduto = buscarInformacoesPeca(conn, codProduto);
        if (infoProduto != null) {
            double precoProduto = (Double) infoProduto[1]; // Preço do
produto
            String nomePeca = (String) infoProduto[0]; // Nome da peça
            valorTotal += precoProduto * quantidade; // Calcular o valor
total

            // Atualizar os detalhes do produto na tabela venda_nfe
            stmtDetalheVendaNFE.setString(1, codProduto); // código da peça
            stmtDetalheVendaNFE.setDouble(2, precoProduto); // preço
            stmtDetalheVendaNFE.setInt(3, quantidade); // quantidade
            stmtDetalheVendaNFE.setString(4, nomePeca); // nome da peça
            stmtDetalheVendaNFE.setInt(5, idVenda); // ID da venda
            stmtDetalheVendaNFE.executeUpdate();

            // Atualizar o estoque
            stmtAtualizaEstoque.setInt(1, quantidade);
            stmtAtualizaEstoque.setString(2, codProduto);
            stmtAtualizaEstoque.executeUpdate();
        }
    }

    // Atualizar status do agendamento
    String sqlAtualizaStatus = "UPDATE agendamento SET status = 'F' WHERE
id = ?";
    stmtAtualizaStatus = conn.prepareStatement(sqlAtualizaStatus);
    stmtAtualizaStatus.setInt(1, idAgendamento);
    stmtAtualizaStatus.executeUpdate();

    // Commit da transação
    conn.commit();

    // Opcional: Se necessário, salvar o valor total em algum lugar ou exibir
    System.out.println("Valor total da venda: " + valorTotal);

    return true;
} catch (SQLException e) {
    System.out.println("Erro ao realizar venda: " + e.getMessage());
    try {
        if (conn != null) {
            conn.rollback(); // Reverte transação em caso de erro
        }
    } catch (SQLException rollbackEx) {
        System.out.println("Erro ao reverter transação: " +
rollbackEx.getMessage());
    }
}

```

```

        return false;
    } finally {
        try {
            if (rsAgendamento != null)
                rsAgendamento.close();
            if (stmtAgendamento != null)
                stmtAgendamento.close();
            if (stmtVendaNFE != null)
                stmtVendaNFE.close();
            if (stmtDetalheVendaNFE != null)
                stmtDetalheVendaNFE.close();
            if (stmtAtualizaEstoque != null)
                stmtAtualizaEstoque.close();
            if (stmtAtualizaStatus != null)
                stmtAtualizaStatus.close();
            if (conn != null)
                conn.close();
        } catch (SQLException e) {
            System.out.println("Erro ao fechar recursos: " + e.getMessage());
        }
    }
}

/**
 * Busca informações de uma peça no estoque.
 *
 * @param conn    Conexão com o banco de dados.
 * @param codPeca Código da peça a ser buscada.
 * @return Um array com o nome do produto e seu preço, ou {@code null} se
 * não encontrado.
 */
private Object[] buscarInformacoesPeca(Connection conn, String codPeca) {
    try (PreparedStatement stmtEstoque = conn
        .prepareStatement("SELECT nome_pro, prc_venda FROM estoque WHERE
codigo_pro = ?")) {
        stmtEstoque.setString(1, codPeca);
        try (ResultSet rsEstoque = stmtEstoque.executeQuery()) {
            if (rsEstoque.next()) {
                return new Object[] { rsEstoque.getString("nome_pro"),
rsEstoque.getDouble("prc_venda") };
            }
        }
    } catch (SQLException e) {
        System.out.println("Erro ao buscar informações da peça: " +
e.getMessage());
    }
    return null;
}
}

```

4 CONCLUSÃO

O Mecânica Nexus ERP demonstra ser uma solução promissora para empresas que buscam eficiência e otimização operacional sem comprometer o orçamento. Por meio de uma abordagem acessível e funcional, o sistema oferece ferramentas robustas para a gestão empresarial, preenchendo lacunas deixadas por ERPs convencionais. A implementação prática do projeto confirma sua relevância no contexto atual, com possibilidades de expansão e adaptação para atender demandas futuras. A pesquisa reforça a importância de soluções personalizadas e acessíveis no mercado dinâmico de tecnologia empresarial.

Mecânica Nexus ERP
Java Programming Project
With Database Integration

Abstract

This article discusses the development of Mecânica Nexus ERP, an operating system aimed at managing business processes with database integration. The primary motivation was to address gaps left by existing ERP solutions, such as lack of customization, high cost, and implementation complexity. The project provides an intuitive, versatile, and affordable platform, standing out with features like data registration, operations management, report generation, backup and security, and a user-friendly interface. Diagrams and code examples illustrate the system's structure and functionality, highlighting its applicability and market potential.

Keywords: ERP, Business Management, Database, Automation, Operating System.

5 REFERÊNCIAS

QUITUMBA FERREIRA. Criação de software de gestão de venda aula 06 Criação do projecto no netbeans, organização dos. YouTube, 2021. Disponível em: https://www.youtube.com/watch?v=iPOTsTMG_jo&list=PLqs4l_WmH-7AjY9OSKBHNzSFyTEsauVVK&index=8. Acesso em: 12 out. 2024.