INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE SÃO PAULO
CAMPUS GUARULHOS

GUILHERME DIONIZIO LUDGERO (GU3054918)
LUCAS SILVA DE OLIVEIRA (GU3054314)

PROJETO FINAL - APO1
ERP - LG SYSTEM

Guarulhos - SP

2025

Objetivo do Projeto

Este projeto visa o desenvolvimento do **ERP LG System**, uma solução de software feita para otimizar a gestão de processos em assistências técnicas de aparelhos eletrônicos. A ideia do sistema vem do fato de que a eficiência operacional nesses estabelecimentos é frequentemente comprometida pela fragmentação de informações e pela dependência de controles manuais, resultando em lentidão no atendimento e potencial perda de dados.

Por isso, o ERP LG System atua como uma plataforma unificada que visa aliviar essas ineficiências. As funcionalidades atualmente implementadas ajudam diretamente este fim:

- **Gerenciamento de Clientes e Funcionários:** A capacidade de cadastrar, consultar e atualizar registros de forma centralizada elimina a necessidade de planilhas ou arquivos físicos. Isso garante que toda a equipe tenha acesso rápido e consistente às informações.
- **Controle de Acesso por Níveis de Privilégio:** A distinção entre usuários administradores e normais introduz uma camada de segurança e organização. Ao restringir operações críticas, como a gestão de funcionários e perfis autorizados, o sistema assegura a integridade dos dados e estabelece um fluxo de trabalho claro e seguro.
- **Funcionalidade de Inativação de Clientes (Soft Delete):** Em vez da exclusão permanente, o sistema adota a prática de marcar registros como inativos. Essa abordagem preserva o histórico completo do cliente, uma funcionalidade importante para futuras consultas, reativações de cadastro ou análises, sem poluir as listas de consulta diárias.

Para a consolidação do LG System como uma ferramenta completa, o futuro do projeto inclui a adição de outros módulos que serão entregues no próximo semestre, apesar de já fazerem parte dos diagramas e planejamento. Pretende-se implementar um sistema para cadastro de aparelho**s** vinculados aos clientes, um módulo para a criação e o acompanhamento de Ordens de Serviço e um sistema de gerenciamento de estoque. A

integração dessas funcionalidades permitirá o rastreamento completo do ciclo de vida de um reparo, desde a recepção do aparelho até a sua devolução, transformando o LG System em uma solução de gestão end-to-end.

Documento de Requisitos do Sistema ERP

1. Requisitos Gerais do Sistema (Não Funcionais)

Estes requisitos aplicam-se a toda a aplicação para garantir consistência, qualidade e manutenibilidade.

- **RNF-GER-01 (Plataforma e Portabilidade):** O sistema deve ser desenvolvido em Java com a biblioteca Swing para a interface gráfica, garantindo sua portabilidade e execução em qualquer sistema operacional que possua uma Java Virtual Machine (JVM) compatível.
- **RNF-GER-02 (Arquitetura e Reutilização):** O sistema deve empregar uma arquitetura modular, utilizando:
    - **Padrão DAO (Data Access Object):** Para separar a lógica de acesso a dados da lógica de negócios e da interface.
    - **Padrão Template Method:** Através de uma classe base abstrata (TelaGerenciamentoAbstrata / TelaCadastroAbstrata) para padronizar a estrutura e o comportamento de todas as telas de gerenciamento e cadastro, maximizando o reuso de código.
    - **Componentes Customizados:** Utilização de componentes de UI especializados (GerenciamentoTable, GerenciamentoForm, etc.) para manter a consistência visual e encapsular funcionalidades.
- **RNF-GER-03 (Manutenibilidade):** O código deve ser organizado em pacotes e classes com responsabilidades claras. O uso de nomes de métodos explícitos, herança e delegação de eventos via expressões lambda deve facilitar a depuração e futuras modificações.
- **RNF-GER-04 (Tratamento de Exceções):** O sistema deve capturar exceções de forma robusta, tanto as esperadas (ex: NumberFormatException) quanto as inesperadas (Exception), prevenindo o encerramento abrupto da aplicação e exibindo mensagens de erro amigáveis ao usuário.

- **RNF-GER-05 (Desempenho):** As operações de banco de dados, especialmente as de listagem em telas de gerenciamento, devem ser executadas sob demanda (na inicialização, ao filtrar ou após uma alteração de dados) para evitar consultas desnecessárias e otimizar a performance.
- **RNF-GER-06 (Usabilidade Geral):** A interface deve ser clara e intuitiva, com fontes legíveis e layouts organizados para proporcionar uma boa experiência ao usuário.

2. Modelo de Dados e Regras de Negócio

- **RF-MDL-01 (Cadastro de Funcionário e Usuário):** O sistema deve permitir o cadastro de funcionários, e cada funcionário deve estar associado a uma única conta de usuário do sistema (relação 1:1). O cadastro deve incluir:
    - **Dados do Funcionário:** Nome, CPF, Função, Telefone, Email.
    - **Dados do Usuário:** Nome de Usuário (único), Senha e Nível de Privilégio (Administrador ou Comum).
- **RF-MDL-02 (Consistência de Email):** O campo "email" da conta de usuário deve ser sempre idêntico ao campo "email" do funcionário vinculado. O sistema deve garantir essa consistência, seja via lógica de aplicação ou gatilhos no banco de dados.
- **RF-MDL-03 (Status de Entidades):** As entidades principais (Cliente, Funcionário, Usuário) devem possuir um status de "ativo" ou "inativo", permitindo que sejam desativadas sem a necessidade de exclusão.
- **RNF-MDL-01 (Integridade Referencial):** A relação entre funcionários e usuários no banco de dados deve ser garantida por chaves estrangeiras (foreign key).

3. Requisitos da Interface do Usuário (UI)

A. Tela de Login

- **RF-LOG-01 (Autenticação):** O sistema deve permitir que o usuário faça login usando seu "Nome de Usuário" ou "Email" e "Senha".
- **RF-LOG-02 (Validação de Campos):** O sistema deve verificar se os campos de login e senha estão preenchidos antes de tentar a autenticação, exibindo um aviso caso contrário.

- **RF-LOG-03 (Visibilidade da Senha):** Uma caixa de seleção deve permitir ao usuário mostrar ou ocultar os caracteres digitados no campo de senha.
- **RF-LOG-04 (Navegação Pós-Login):** Após um login bem-sucedido, a tela de login deve ser fechada e a tela principal deve ser aberta, carregando o contexto do usuário autenticado.
- **RF-LOG-05 (Feedback de Autenticação):** O sistema deve exibir mensagens claras de sucesso, erro de credenciais inválidas ou falhas inesperadas no processo.
- **RF-LOG-06 (Acessibilidade):** Pressionar a tecla "Enter" no campo de senha deve acionar a tentativa de login.
- **RF-LOG-07 (Saída):** Um botão "Sair" deve fechar a janela de login e encerrar a aplicação.

- **RNF-LOG-01 (Segurança da Interface):** O campo de senha deve ser, por padrão, um JPasswordField para mascarar a digitação.

B. Tela Principal (Shell da Aplicação)

- **RF-PRI-01 (Apresentação Personalizada):** A janela deve exibir um título de boas-vindas com o nome do usuário logado.
- **RF-PRI-02 (Navegação por Abas):** O conteúdo principal deve ser organizado em abas, cada uma contendo uma tela de gerenciamento.
- **RF-PRI-03 (Controle de Acesso por Privilégio):** A visibilidade de menus e abas deve ser controlada pelo privilégio do usuário:
    - **Padrão:** Acesso à gestão e cadastro de "Clientes".
    - **Administrador:** Acesso total, incluindo gestão e cadastro de "Funcionários" e "Usuários".
- **RF-PRI-04 (Logout):** Um menu "Sistema" -> "Sair" deve fechar a tela principal e retornar à tela de login.
- **RF-PRI-05 (Encerramento):** Fechar a janela principal deve encerrar toda a aplicação.
- **RF-PRI-06 (Acesso a "Sobre"):** Um menu "Sistema" -> "Sobre" deve exibir informações da aplicação.

- **RNF-PRI-01 (Passagem de Contexto):** A tela principal é responsável por passar a instância do usuarioLogado para as telas filhas, garantindo que elas operem com as permissões corretas.

C. Telas de Gerenciamento

i. Padrões Funcionais Comuns

- **RF-GER-01 (Listagem de Dados):** Cada tela deve exibir os dados da sua respectiva entidade em uma tabela.
- **RF-GER-02 (Seleção e Preenchimento):** Clicar em uma linha da tabela deve preencher automaticamente os campos do formulário com os dados do registro selecionado.
- **RF-GER-03 (Limpeza de Formulário):** Um botão "Limpar" deve esvaziar todos os campos do formulário e remover a seleção da tabela.
- **RF-GER-04 (Ativação e Desativação):** Devem existir botões para ativar/desativar um registro. O sistema deve impedir ações redundantes (ex: ativar um item já ativo) e pedir confirmação ao usuário.
- **RF-GER-05 (Exclusão):** Um botão "Excluir" deve permitir a remoção permanente de um registro, somente após um aviso claro e confirmação do usuário.
- **RF-GER-06 (Filtragem por Status):** Uma caixa de seleção "Mostrar Inativos", visível apenas para Administradores, deve permitir alternar a visualização da tabela entre registros ativos e todos os registros.
- **RF-GER-07 (Feedback de Operações):** O sistema deve usar diálogos para informar o sucesso ou falha das operações (CRUD) e para validar ações (ex: exigir a seleção de um item antes de atualizar).
- **RF-GER-08 (Acesso ao Cadastro):** Um botão "Cadastrar" deve abrir a tela de cadastro da respectiva entidade. Esta tela deve ser modal, bloqueando a interação com a janela principal até que seja fechada, para garantir que o usuário complete ou cancele a operação de cadastro antes de retornar à tela de gerenciamento.

ii. Especificidades por Tela

- **Clientes:**
    - A tabela deve exibir colunas de dados pessoais e de endereço.
    - O formulário é dividido em dois painéis: "Dados do Cliente" e "Endereço Residencial".
- **Funcionários:**
    - A tabela deve exibir colunas de dados profissionais (ID, Nome, CPF, Função, etc.).
    - O formulário é único e centralizado.
- **Usuários:**
    - A tabela deve exibir dados da conta (ID, Nome de Usuário, Email, Privilégio). A senha não deve ser exibida.
    - **Segurança do Formulário:** Ao selecionar um usuário, o campo de senha é limpo. Se a atualização for salva com a senha em branco, a senha original é mantida.
    - **Ausência de Cadastro Direto:** Esta tela não possui um botão "Cadastrar", pois o cadastro de usuário está vinculado ao de funcionário.

Códigos do Projeto

01 - Alterações no arquivo DBQuery

O arquivo DBQuery, fornecido pelo professor em sala de aula, foi utilizado para esse projeto. Nele foi realizado algumas adições, para se adequar ao contexto do projeto, que são:

Método: *public int insert(String[] values, String editableFieldsName)*

Foi criada uma sobrecarga do método insert para que ele também receba todos os campos da tabela, exceto o campo da chave primária. Essa nova versão do método retorna o valor da chave primária (ID) gerada no momento da inserção do dado.

```
public int insert(String[] values, String editableFieldsNames) {
    for (String value : values) {
        System.out.println(value);
    }
    String[] campos = editableFieldsNames.split(",");

    if (values.length == campos.length) {
        String sql = "INSERT INTO " + this.tableName + " ( " + this.joinElements(campos, ", ");
        sql += ") VALUES ('" + joinElements(values, "','") + "')";
        System.out.print(sql);
        return this.execute(sql, true);
    } else {
        System.out.print("O número de valores informados não é equivalente aos campos da tabela!");
    }
    return 0;
}
```

Método: *public int execute(String sql, boolean retornarID)*

Para viabilizar isso, também foi feita uma sobrecarga do método execute, permitindo que ele retorne a chave primária do último registro inserido. Essa sobrecarga é utilizada internamente pelo novo método insert.

```
public int execute(String sql, boolean retornarId) {
    if (!retornarId) {
        return execute(sql); // Chama o método original se o boolean for false
    }

    try {

        int rs = statement.executeUpdate(sql, Statement.RETURN_GENERATED_KEYS);
        if (rs > 0) {
            ResultSet lastId = statement.getGeneratedKeys();
            if (lastId.next()) {
                return lastId.getInt(1); // Retorna o ID gerado
            } else {
                System.out.println("Aviso: Nenhuma chave gerada foi retornada.");
            }
        }
    } catch (SQLException e) {
        System.out.println("\nErro ao executar comando SQL: " + e.getMessage());
    }

    return 0;
}
```

02 - Alterações do arquivo DBConnection

No arquivo DBConnection, outro código fornecido pelo professor durante as aulas, não foram realizadas modificações do código. Apenas foram atribuídos os valores utilizados para a conexão do banco de dados do projeto.

```
public DBConnection() {
    this.setHost    ("localhost");
    this.setPort    ("3306");
    this.setSchema  ("projeto_apo");
    this.setUser    ("root");
    this.setPassword("");
    this.doConnection();
}
```

03 - Arquivos Enums

Os códigos dos arquivos enums, localizados dentro do pacote model.enums, compartilham a mesma estrutura. Optamos por um enum em vez de Strings comuns, pois torna o código mais seguro, pois previne erros de digitação e garante que apenas os valores definidos (EM_MANUTENCAO, EM_ESTOQUE, etc.) possam ser utilizados em todo o sistema. Para a explicação dos blocos, utilizaremos o enum EstadoAparelho como exemplo.

1a - Definição das Constantes e Associação de Valores

```
EM_MANUTENCAO("em_manutencao"),
EM_ESTOQUE("em_estoque"),
EM_ENTREGA("em_entrega");
```

Aqui são declarados os únicos valores possíveis para o EstadoAparelho. Cada um deles (EM_MANUTENCAO, EM_ESTOQUE, EM_ENTREGA) é uma constante e um objeto imutável. O texto entre parênteses, como "em_manutencao", é um argumento passado para o construtor do enum no momento em que ele é inicializado, representando o valor exato que será armazenado na coluna correspondente do banco de dados.

1b - Mecanismo Interno de Armazenamento

```
private final String valorDb;

EstadoAparelho(String valorDb) {
    this.valorDb = valorDb;
}
```

Este bloco é o mecanismo interno que permite a associação do valor do banco de dados a cada constante. O atributo private final String valorDb serve para guardar a string bruta (ex: "em_estoque"). O construtor EstadoAparelho(String valorDb) é chamado automaticamente para cada constante declarada acima, recebendo o valor passado entre parênteses e atribuindo-o ao atributo valorDb.

1c - Métodos de Representação (Saída de Dados)

```
public String getValorDb() {
    return valorDb;
}

@Override
public String toString() {
    switch (this) {
        case EM_MANUTENCAO: return "Em manutenção";
        case EM_ESTOQUE: return "Em estoque";
        case EM_ENTREGA: return "Em entrega";
        default: return valorDb;
    }
}
```

Para interagir com o resto do sistema, o enum fornece estes dois métodos de "saída". O método getValorDb() é utilizado pela camada de acesso a dados (DAO) para obter a string exata que deve ser salva ou consultada no banco. Por outro lado, o método toString() é sobrescrito para oferecer uma representação textual amigável ao usuário. É este formato ("Em estoque") que será exibido em componentes da interface, como tabelas e listas de seleção, melhorando a usabilidade.

1d - Métodos de Conversão (Entrada de Dados)

```
public static EstadoAparelho fromDb(String valor) {
    for (EstadoAparelho e : values()) {
        if (e.valorDb.equals(valor)) {
            return e;
        }
    }
    throw new IllegalArgumentException(
        "Valor inválido para EstadoAparelho: " + valor);
}

public static EstadoAparelho fromString(String texto) {
    for (EstadoAparelho e : values()) {
        if (e.toString().equalsIgnoreCase(texto)) {
            return e;
        }
    }
    throw new IllegalArgumentException(
        "Texto inválido para EstadoAparelho: "+ texto);
}
```

Fazendo o caminho inverso, estes outros dois métodos de fábrica estáticos servem para converter uma string de volta para um objeto EstadoAparelho. O método fromDb é usado para pegar o valor lido do banco de dados (ex: "em_manutencao") e encontrar a constante enum correspondente. De forma análoga, fromString faz o mesmo, mas a partir do texto formatado exibido na interface (ex: "Em manutenção"), sendo útil para processar a seleção de um usuário. Ambos garantem a robustez do código ao lançar uma exceção (IllegalArgumentException) se o texto de entrada não corresponder a nenhum estado válido.

04 - Arquivos de classe modelo

Sua principal função é servir como um “molde” para os dados das entidades do banco de dados. Esses arquivos se encontram no pacote model. Para a explicação do código utilizaremos a classe **Usuario.**

4a - Declaração da Classe e Atributos

```
package model;

import model.enums.Privilegios;

public class Usuario {
    private int id;
    private int idFuncionario;
    private String usuario;
    private String email;
    private String senha;
    private Privilegios privilegios;
    private boolean ativo;
```

Este bloco inicial declara a classe Usuario dentro do pacote model. Em seguida, são definidos os atributos (ou campos) da classe. Cada atributo é declarado como private, o que significa que eles só podem ser acessados diretamente de dentro da própria classe Usuario. Essa é uma prática fundamental da programação orientada a objetos chamada encapsulamento. Note que os atributos (id, usuario, senha, etc.) correspondem diretamente às colunas da sua tabela tb_usuario no banco de dados, e o tipo de privilegios é o enum que criamos, garantindo que ele só possa receber valores válidos.

4b - Construtores da classe

```java
public Usuario() {
    this.ativo = true;
}

public Usuario(
        int id, int idFuncionario, String usuario, String email,
        String senha, Privilegios privilegios,boolean ativo) {
    this.setId(id);
    this.setIdFuncionario(idFuncionario);
    this.setUsuario(usuario);
    this.setEmail(email);
    this.setSenha(senha);
    this.setPrivilegios(privilegios);
    this.setAtivo(ativo);
}
```

Aqui temos os construtores, que são métodos especiais responsáveis por criar e inicializar um objeto Usuario.

O primeiro, public Usuario(), é o construtor padrão (ou sem argumentos). Ele é útil para criar um objeto Usuario vazio que será preenchido depois, como acontece nos métodos listar do DAO. É importante notar que ele define this.ativo = true, estabelecendo que todo novo usuário criado desta forma começa como ativo por padrão.

O segundo construtor é parametrizado. Ele é um atalho para criar um objeto já com todos os seus atributos preenchidos de uma vez.

4c - Métodos de Acesso (Getters e Setters)

```java
public void setSenha(String senha) {
    this.senha = senha;
}

public Privilegios getPrivilegios() {
    return privilegios;
}

public void setPrivilegios(Privilegios privilegios) {
    this.privilegios = privilegios;
}
```

E o bloco restante do código desse arquivo, são os métodos de acessos públicos. Como os atributos da classe são privados, esses métodos são a única maneira de interagir com eles a partir de fora da classe. Com isso, abre a possibilidade de futuramente incluirmos tratamentos para a entrada e saída desses dados da classe, como por exemplo, verificação de valores válidos para CPF, senha e etc. Os "Getters" (como getId()) leem os valores, enquanto os "Setters" (como setId()) os modificam. É importante notar também a convenção isAtivo() para o getter do atributo booleano ativo, que torna o código em condicionais mais natural e legível (ex: if (usuario.isAtivo())).

05 - Arquivos de classe DAO

Sua principal função é servir como uma ponte entre a aplicação e o banco de dados, encapsulando toda a lógica de acesso e manipulação de uma entidade específica. Conhecidas como DAO (Data Access Object), essas classes são responsáveis por executar os comandos SQL de inserção, atualização, exclusão e consulta. Esses arquivos se encontram no pacote control. Para a explicação do código, utilizaremos a classe UsuarioDAO.

5a - Estrutura e Configuração Inicial

```
import database.DBQuery;
import model.Usuario;
import model.enums.Privilegios;

public class UsuarioDAO {
    private DBQuery dbQuery;
    private String editableFieldsName =
        "id_funcionario, usuario, senha, email, privilegios";

    public UsuarioDAO() {
        String tableName = "tb_usuario";
        String fieldNames =
            "id_usuario, id_funcionario, usuario, senha, "
            + "email, privilegios, ativo";
        String fieldKey = "id_usuario";

        dbQuery = new DBQuery(tableName, fieldNames, fieldKey);
    }
```

Este bloco inicial define a estrutura da classe UsuarioDAO. Ela declara um atributo dbQuery, que é um objeto auxiliar para executar as operações no banco de dados, e uma

string editableFieldsName, que lista os campos que podem ser preenchidos em uma inserção (excluindo a chave primária auto-incrementável). No construtor UsuarioDAO(), o objeto dbQuery é configurado especificamente para trabalhar com a tabela tb_usuario, informando a ele o nome da tabela, a lista completa de colunas e qual delas é a chave primária (id_usuario). Essa configuração centralizada evita a repetição de nomes de tabelas e campos nos outros métodos da classe.

5b - Método de Inserção de Dados (Salvar)

```
28⊕     public int salvar(Usuario usuario) {
45      }
```

O método salvar é responsável por inserir um novo registro de usuário no banco de dados. Ele recebe um objeto Usuario completo, extrai seus valores e os organiza em um array de strings na ordem correta dos editableFieldsName. Note como ele utiliza usuario.getPrivilegios().getValorDb() para obter a string correta do enum, demonstrando a integração entre as camadas. Em seguida, ele delega a operação de INSERT para o objeto dbQuery, que executa o comando SQL e retorna o ID gerado pelo banco para o novo registro, valor útil para operações subsequentes.

5c. Métodos de Modificação (Atualizar, Ativar, Desativar)

```
47⊕     public boolean atualizar(Usuario usuario) {
59      }
60
61⊕     public boolean ativar(Usuario usuario) {
71      }
72
73⊕     public boolean desativar(Usuario usuario) {
83      }
```

Este conjunto de métodos lida com a atualização de registros existentes. O método atualizar é genérico, pegando todos os valores de um objeto Usuario e passando para o dbQuery.update(), que monta um comando UPDATE completo. Já os métodos ativar e desativar são mais específicos. Em vez de atualizar todos os campos, eles executam um comando SQL UPDATE direcionado, que altera apenas a coluna ativo para 1 ou 0. Eles também incluem validações para garantir que um objeto Usuario válido foi fornecido antes de tentar a operação no banco.

5d - Método de Exclusão

```
85⊕     public boolean excluir(Usuario usuario) {
97      }
```

O método excluir tem a responsabilidade de remover um registro do banco de dados. Ele recebe um objeto Usuario e, de forma similar ao atualizar, extrai seus valores. Em seguida, chama dbQuery.delete(), que usa a string dos valores para executar o comando DELETE no registro correspondente. Ele retorna true se a operação foi bem-sucedida e false caso contrário.

5e - Métodos de Listagem e Busca

```
 99⊕     public List<Usuario> listarTodos() {
124      }
125
126⊕     public List<Usuario> listarApenasAtivos() {
151      }
152
153⊕     public Usuario buscar(int id) {
176      }
```

Esses métodos são responsáveis por recuperar dados do banco (SELECT). O método listarTodos busca todos os registros da tabela, enquanto listarApenasAtivos aplica um filtro para trazer apenas os usuários com ativo = 1. Já o método buscar recupera um único usuário pelo seu ID. Todos eles seguem um padrão: executam uma query através do dbQuery.select(), recebem um ResultSet, e então iteram sobre os resultados. Para cada linha encontrada, eles criam um novo objeto Usuario e o populam com os dados das colunas, convertendo os valores do banco (como a string de privilégios) de volta para os tipos corretos do Java (usando Privilegios.fromDb()).

5f - Método de Autenticação

```
178⊕     public Usuario autenticar(String login, String senha) {
202      }
```

Este é um método de busca para uma lógica de negócio específica: a autenticação de login. Diferente das buscas genéricas, ele constrói um filtro SQL personalizado para encontrar um registro que atenda a todas as condições de um login válido: o login fornecido deve corresponder ao campo usuario ou email, a senha deve ser exata e o usuário deve estar

ativo. Se um registro correspondente for encontrado, ele cria e retorna o objeto Usuario completo, caso contrário, retorna null, indicando que a autenticação falhou.

6 - Arquivo FormInputH

```
17  public class FormInputH extends JPanel {
18      private static final long serialVersionUID = 1L; // Default
19
20      private JLabel label;
21      private JComponent campo;
22
23      private Font labelFont = new Font("Arial", Font.BOLD, 12);
24      private Font inputFont = new Font("Arial", Font.PLAIN, 12);
25
26      private Dimension inputSize = new Dimension(200, 25);
27
28      public FormInputH(String textoLabel, JComponent c) {
```

O componente “FormInputH” é uma parte reutilizável da View, projetada para padronizar campos de formulário das Telas de Cadastro, e garantir a consistência visual em todo o sistema. Ele consiste em um JPanel que encapsula um JLabel e um JComponent genérico, como JTextField, JComboBox ou JPassword, que é injetado através de seu construtor, tornando-o agnóstico ao tipo de entrada. Para simplificar seu uso, FormInputH oferece uma interface unificada com métodos como getText() e getSelectedItem() que redirecionam a chamada ao componente interno, ocultando a complexidade de interagir com diferentes classes.

7 - Arquivo FormInputV

```
17  public class FormInputV extends JPanel {
18      private static final long serialVersionUID = 1L;
19
20      private JLabel label;
21      private JComponent campo;
22
23      private Font labelFont = new Font("Arial", Font.BOLD, 12);
24      private Font inputFont = new Font("Arial", Font.PLAIN, 12);
25      private int inputHeight = 25;
26
27      public FormInputV(String textoLabel, JComponent c) {
53      }
```

O componente FormInputV também é um elemento de interface reutilizável, feito para padronizar a criação de campos de formulário para as Telas de Gerenciamento, na view e garantir a consistência visual do sistema. Estruturado como um JPanel com BoxLayout vertical, ele encapsula um JLabel acima de um JComponent genérico, que é injetado via construtor para uma maior flexibilidade.

8 - Arquivo GerenciamentoButton

```
12  public class GerenciamentoButton extends JPanel {
13      private static final long serialVersionUID = 1L;
14
15      private JButton btnNovo = new JButton("Novo");
16      private JButton btnAtualizar = new JButton("Atualizar");
17      private JButton btnLimpar = new JButton("Limpar");
18      private JButton btnAtivar = new JButton("Ativar");
19      private JButton btnDesativar = new JButton("Desativar");
20      private JButton btnExcluir = new JButton("Excluir");
21
22      public GerenciamentoButton(
35      }
36
37      public GerenciamentoButton(
54      }
55
56      private void inicializar(
93      }
```

O componente GerenciamentoButton é um JPanel que centraliza a criação e a lógica do painel de botões, garantindo um comportamento e uma interface consistente em todas as telas de gerenciamento do sistema.

Adicionalmente, ele encapsula a lógica de autorização da camada de visão, pois recebe o objeto Usuário logado e o utiliza para renderizar condicionalmente os botões de acesso restrito, como "Ativar" e "Excluir", apenas para perfis de administrador.

9 - Arquivo GerenciamentoForm

```
10  public class GerenciamentoForm extends JPanel{
11      private static final long serialVersionUID = 1L; // Default serialVersi
12
13      private String titulo;
14
15      public GerenciamentoForm(String tituloForm, FormInputV... entradas) {
35      }
36
37      public GerenciamentoForm() { // Cria um painel de preenchimento
40      }
41
42      public String getTitulo() {
43          return this.titulo;
44      }
```

O componente GerenciamentoForm atua como um contêiner na view, cuja finalidade é agrupar campos de entrada relacionados em um JPanel com borda e título, organizando visualmente os formulários.

Seu design flexível é evidenciado pelo construtor principal, que aceita uma quantidade variável de componentes “FormInputV”. A classe automatiza o layout vertical e a inserção de espaçamento, enquanto um construtor sobrecarregado permite a criação de um painel vazio, que serve para o alinhamento de layouts mais complexos.

10 - Arquivo GerenciamentoTable

```
13  public class GerenciamentoTable extends JTable {
14      private static final long serialVersionUID = 1L; // Default
15
16      // Componentes da tabela
17      private DefaultTableModel tableModel;
18      private String[] colunas;
19
20      // Personalização
21      private Font headerFont = new Font("Arial", Font.BOLD, 12);
22      private Font cellsFont = new Font("Arial", Font.PLAIN, 12);
23      private int alturaTabelaScroll = 180;
24
25      public GerenciamentoTable (String[] colunasTabela){
40      }
41
42      private void configurarTabela() {
52      }
```

A classe GerenciamentoTable é um componente especializado que herda diretamente de JTable, projetado para padronizar a exibição de dados tabulares em todas as telas do sistema.

Sua principal característica é a implementação de uma tabela padrão interna que, torna as células não editáveis, garantindo a integridade da visualização. Adicionalmente, ela encapsula a lógica de estilização e a criação de um JScrollPane através do método utilitário comScrollPanel(), simplificando drasticamente sua implementação.

11 - Arquivo Tela CadastroAbstrata

Esta classe é a fundação de todas as telas de cadastro do sistema. Sua principal função é definir uma estrutura e um comportamento padrão, forçando todas as telas de cadastro a terem a mesma aparência e funcionalidades básicas. Ela utiliza o padrão de projeto Template Method.

11a - Estrutura da Classe

```
public abstract class TelaCadastroAbstrata extends JDialog {
    private static final long serialVersionUID = 1L; // Default serialVersion

    protected JPanel painelPrincipal = new JPanel();
    protected JPanel painelFormulario1 = new JPanel();
    protected JPanel painelFormulario2 = new JPanel();
    protected JPanel painelBotoes = new JPanel();

    protected JButton btnSalvar = new JButton("Salvar");
    protected JButton btnCancelar = new JButton("Cancelar");

    // Personalização
    protected Font panelFont = new Font("Arial", Font.BOLD, 14);
    protected Dimension buttonSize = new Dimension(100, 30);
```

Este bloco define a TelaCadastroAbstrata como um JDialog, o que a torna uma janela de diálogo que pode ser exibida sobre a janela principal. Ela já pré-define os painéis que irão compor a tela (painelPrincipal, painelFormulario1, etc.) e os botões comuns a todo cadastro (btnSalvar, btnCancelar). Declarar esses componentes aqui garante que todas as telas que herdarem dela terão esses elementos à disposição.

11b - Construtor e Configurações Padrão

```
public TelaCadastroAbstrata(String tituloJanela) {
    // -- CONFIGURAÇÕES DA JANELA --
    setTitle(tituloJanela);
    setDefaultCloseOperation(DISPOSE_ON_CLOSE);
    setResizable(false);
    setModal(true); // Bloqueia outras janelas enquanto aberta

    // -- CONFIGURAÇÕES DOS PAINEIS --
    this.painelPrincipal.setLayout(new BoxLayout(this.painelPrincipal, BoxLayout.Y_AXIS));
    this.painelPrincipal.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));

    this.painelBotoes.setLayout(new FlowLayout(FlowLayout.RIGHT));

    // -- AÇÕES E MONTAGEM DOS BOTÕES --
    btnCancelar.addActionListener(e -> dispose());
    this.painelBotoes.add(this.btnCancelar);
    this.painelBotoes.add(this.btnSalvar);
}
```

O construtor da classe abstrata é responsável por aplicar configurações que serão idênticas em todas as telas de cadastro. Ele define o título da janela, a torna não redimensionável e, mais importante, a define como modal (setModal(true)), o que bloqueia a

interação com o resto da aplicação enquanto ela estiver aberta. Ele também já define a ação padrão do botão "Cancelar", que é simplesmente fechar a janela (dispose()), um comportamento que não precisa ser reescrito nas classes filhas.

11c - Métodos Auxiliares de Construção

```
protected void finalizarTela() {
}

// Cria um painel de formulário
protected JPanel criarPainelFormulario(String titulo) {
}

// Adiciona um painel personalizado ao painel principal
protected void adicionarPainelFormulario(JPanel painel) {
}

// Adicionar componentes ao painel de formulário
protected void adicionarEntrada(JPanel painel, FormInputH input) {
}

protected void aplicarEstiloPainel() {
}
```

Estes são métodos protected projetados para serem "ferramentas" para as classes filhas. Eles simplificam a construção da interface, oferecendo uma forma padronizada de criar um painel com título (criarPainelFormulario), de adicionar esse painel à janela principal com um espaçamento (adicionarPainelFormulario), e de adicionar cada campo de formulário (FormInputH) de maneira alinhada e organizada dentro de um painel (adicionarEntrada).

11d - Métodos Abstratos (O "Contrato")

```
100     // Subclasse deve definir o layout e ação de salvar
101     protected abstract void construirFormulario();
102     protected abstract void aoSalvar();
```

Aqui está a essência do padrão Template Method. Estes dois métodos são declarados como abstract, o que significa que eles não têm implementação nesta classe. Em vez disso, eles funcionam como um "contrato", obrigando qualquer classe que herde de TelaCadastroAbstrata (como a TelaCadastroFuncionario) a fornecer uma implementação concreta para eles. A subclasse deve dizer como seu formulário específico será construído e o que deve acontecer quando o botão "Salvar" for clicado.

12 - Arquivos de TelaCadastro

Esta são as classes com uma implementação concreta da TelaCadastroAbstrata, fornecendo a interface e a lógica específicas para cadastros. Para esse exemplo, utilizaremos o TelaCadastroFuncionario, que é a tela para cadastrar um funcionário e sua conta de usuário associada.

12a - Estrutura e Inicialização

```java
public class TelaCadastroFuncionario extends TelaCadastroAbstrata {
    private static final long serialVersionUID = 1L; // Default serialVersion

    private TelaGerenciamentoFuncionarios painelGerenciamento;

    private static final String TITULO_JANELA = "Cadastro de Funcionário";

    // -- COMPONENTES DE ENTRADA --
    private FormInputH tfNome;
    private FormInputH tfCpf;
    private FormInputH cbFuncao;
    private FormInputH tfTelefone;
    private FormInputH tfEmail;

    private FormInputH tfUsuario;
    private FormInputH pfSenha;
    private FormInputH cbPrivilegios;

    public TelaCadastroFuncionario() {
        this((TelaGerenciamentoFuncionarios) null);
    }

    public TelaCadastroFuncionario(TelaGerenciamentoFuncionarios painelGerenciamento) {
        super(TITULO_JANELA);

        this.painelGerenciamento = painelGerenciamento;
        inicializarCampos();
        construirFormulario();
        finalizarTela();
    }
```

A classe herda de TelaCadastroAbstrata e declara seus próprios campos de formulário (FormInputH) para os dados do funcionário e do usuário. O construtor segue a sequência definida pelo "template": primeiro ele chama o construtor da classe pai (super(...)) para configurar a janela, depois inicializa seus campos específicos (inicializarCampos()), constrói o formulário (construirFormulario()) e, por fim, chama finalizarTela() (método da classe pai) para montar tudo e exibir a janela.

12b - Construção do Formulário Específico

```
62      @Override
63      protected void construirFormulario() {
81      }
```

Aqui, o método abstrato construirFormulario é implementado. Ele utiliza os métodos auxiliares herdados (criarPainelFormulario, adicionarEntrada) para montar sua interface, dividida em dois painéis: "Dados do Funcionário" e "Conta". Ao final, ele adiciona o ActionListener ao btnSalvar, conectando o clique do botão à lógica de salvamento que será definida no método aoSalvar.

12c - Lógica de Negócio (aoSalvar)

```
 83      @Override
 84      protected void aoSalvar() {
128      }
```

Esta é a implementação do segundo método abstrato e contém a lógica de negócio mais importante da tela. O processo é executado em uma sequência transacional, começando pela coleta dos dados do formulário para popular um objeto Funcionario. Este objeto é então salvo no banco de dados para que seu ID gerado possa ser recuperado. Com esse ID em mãos, os dados da conta são coletados para um objeto Usuario, e o ID do funcionário recém-criado é utilizado para vincular as duas entidades. Logo após, o objeto Usuario também é salvo no banco, completando o cadastro. Se tudo ocorrer bem, o sistema exibe uma mensagem de sucesso, atualiza a tabela na tela de gerenciamento e, por fim, fecha a janela de cadastro. Todo esse fluxo de trabalho é envolto em um bloco try-catch para garantir que qualquer erro durante o processo seja capturado e comunicado ao usuário.

13 - Arquivo TelaGerenciamentoAbstrata

Esta classe abstrata serve como um molde padronizado para todas as telas que gerenciam dados (CRUD - Criar, Ler, Atualizar, Excluir) no sistema. Ela garante que todas essas telas tenham uma aparência e comportamento consistentes.

13a - Estrutura da classe

```
20  public abstract class TelaGerenciamentoAbstrata extends JPanel {
21      private static final long serialVersionUID = 1L; // Default serialVersion
22
23      protected Usuario usuarioLogado;
24
25      protected JPanel painelTabela = new JPanel();
26      protected JPanel painelFormularios = new JPanel();
27      protected JPanel painelBotoes = new JPanel();
28
29      // Componentes do painel tabela
30      protected GerenciamentoTable tabela;
31      protected GerenciamentoForm formularioEsquerdo;
32      protected GerenciamentoForm formularioDireito;
33
34      protected JCheckBox ckbMostrarInativos = new JCheckBox("Mostrar Inativos");
35      protected String[] colunasTabela;
36
37      // Personalização
38      protected Font panelFont = new Font("Arial", Font.BOLD, 14);
39      protected Dimension buttonSize = new Dimension(100, 30);
40
41      public  TelaGerenciamentoAbstrata(Usuario usuarioInstancia) {
42          this.usuarioLogado = usuarioInstancia;
43
44          this.setLayout(new GridBagLayout());
45      }
```

A classe é declarada como abstract e herda de JPanel, o que significa que ela é um painel que pode ser adicionado a uma aba ou a uma janela. Sua função é definir os três principais blocos visuais que compõem uma tela de gerenciamento: um painel para a tabela de dados (painelTabela), um para os formulários de edição (painelFormularios) e um para os botões de ação (painelBotoes). Ela também já declara os componentes reutilizáveis que serão usados, como a GerenciamentoTable e os GerenciamentoForm.

13b - Montagem da Tabela de Dados

```
 68      protected void criarPainelTabela(String nomeTabela, String[] colunas) {
113      }
```

O método auxiliar criarPainelTabela padroniza a criação da seção superior da tela, que contém a tabela de dados. Suas responsabilidades incluem configurar o título e o layout do painel, além de adicionar condicionalmente a caixa de seleção "Mostrar Inativos" apenas se o usuário logado tiver privilégios de Administrador. Após isso, ele cria a GerenciamentoTable com as colunas especificadas e, adiciona um MouseListener à tabela. Esse listener é utilizado

para detectar o clique do usuário em uma linha e delega a ação para o método abstrato preencherFormulario, que será implementado pela classe filha.

13c - Montagem dos Formulários

```
117⊕    protected void criarPainelFormulario() {
175     }
```

Este método, criarPainelFormulario, é responsável por montar a seção intermediária da tela, onde os formulários de entrada de dados são exibidos. Sua principal característica é a flexibilidade do layout, que se adapta dependendo de quantos formulários a tela de gerenciamento específica precisa. Se um formularioDireito com título for definido pela classe filha, o método organiza os dois formulários lado a lado, dividindo o espaço horizontalmente. No entanto, se apenas o formularioEsquerdo for relevante, o layout se ajusta de forma inteligente para centralizar este único formulário, utilizando painéis vazios como espaçadores nas laterais para uma apresentação visualmente mais agradável. Essa abordagem permite que as classes filhas decidam se usarão um ou dois painéis de formulário sem a necessidade de reescrever a lógica de layout.

13d - Métodos Abstratos (Os “contrato” da classe)

```
115     protected abstract void carregarDadosTabela();
116
```

```
177     protected abstract void preencherFormulario(int linha);
178
179     protected abstract void limparFormulario();
```

Os métodos carregarDadosTabela, preencherFormulario e limparFormulario são declarados como abstract. Isso significa que eles funcionam como um "contrato", obrigando qualquer classe filha a fornecer sua própria implementação para essas três ações essenciais: buscar os dados específicos para popular a tabela, pegar os dados de uma linha selecionada para preencher os formulários e, por fim, limpar os campos do formulário.

14 - Arquivo TelaGerenciamento

Esta classe é uma implementação concreta da TelaGerenciamentoAbstrata. Ela "preenche as lacunas" do contrato para criar as telas de gerenciamentos, com suas próprias regras e componentes. Para este exemplo, utilizaremos a TelaGerenciamentoFuncionario.

14a - Estrutura e Construção da Tela

```
23  public class TelaGerenciamentoFuncionarios extends TelaGerenciamentoAbstrata {
24      private static final long serialVersionUID = 1L; // Default serialVersion
25
26      // Tabela
27      private String nomeTabela = "Lista de Funcionários";
28      private String[] colunas = {
29          "ID", "Nome", "CPF", "Função", "Telefone", "Email"
30      };
31
32      // Campos do formulário
33      private String nomeFormulario1 = "Dados do Funcionário";
34
35      private FormInputV tfId        = new FormInputV("ID:", new JTextField());
36      private FormInputV tfNome      = new FormInputV("Nome:", new JTextField());
37      private FormInputV tfCpf       = new FormInputV("CPF:", new JTextField());
38      private FormInputV cbFuncao    = new FormInputV("Função:", new JComboBox<>(FuncaoFuncionario.values()));
39      private FormInputV tfTelefone  = new FormInputV("Telefone:", new JTextField());
40      private FormInputV tfEmail     = new FormInputV("Email:", new JTextField());
41
42      public TelaGerenciamentoFuncionarios(Usuario usuarioInstancia) {
68      }
```

A classe herda de TelaGerenciamentoAbstrata e, em seu construtor, executa a sequência lógica para montar a tela. Ela chama o construtor da classe pai, usa os métodos auxiliares herdados como criarPainelTabela e criarPainelFormulario para montar a interface, define seus formulários específicos (neste caso, apenas um à esquerda) e configura o painel de botões com as ações que disparam os métodos da própria classe. Ao final, ela chama finalizarTela() para juntar todas as partes e exibir o painel completo.

14b - Implementação dos Métodos Abstratos

```
70      protected void carregarDadosTabela() {
92      }
93
94      protected void preencherFormulario(int linha) {
116     }
117
118     protected void limparFormulario() {
127     }
```

Aqui a classe cumpre seu "contrato" ao implementar os métodos abstratos. O método carregarDadosTabela contém a lógica específica para funcionários, criando um FuncionarioDAO, verificando o estado da caixa "Mostrar Inativos" para decidir qual método do DAO chamar e, por fim, iterando sobre a lista de funcionários para popular a tabela. Da

mesma forma, preencherFormulario sabe exatamente como ler os dados de uma linha da tabela de funcionários e atribuí-los aos JTextField e JComboBox corretos no formulário.

E por fim, temos o método limparFormulario, que define o contrato para a ação de limpar o formulário. A implementação na classe filha deve garantir que todos os campos de entrada de dados sejam redefinidos para seu estado inicial (campos de texto vazios, caixas de seleção desmarcadas, etc.). Também foi incluído na implementação, remover qualquer seleção da tabela (tabela.clearSelection()) para que a interface permaneça em um estado consistente.

14c - Métodos de Ação (CRUD)

```
129⊕    private void atualizarFuncionario() {
183     }
184
185⊕    private void ativarFuncionario() {
242     }
243
244⊕    private void desativarFuncionario() {
302     }
303
304⊕    private void excluirFuncionario() {
359     }
```

Finalmente, este bloco contém os métodos privados que implementam a lógica de negócio para cada botão de ação, como Atualizar ou Ativar. Cada um desses métodos segue um fluxo claro e consistente: primeiro, ele verifica se um item foi selecionado na tabela para então pegar os dados necessários do formulário. Em seguida, ele cria uma instância do FuncionarioDAO e chama o método apropriado para interagir com o banco de dados. Ao final da operação, ele fornece um feedback ao usuário por um JOptionPane e, por fim, atualiza a tabela e limpa o formulário para refletir a mudança no estado dos dados.

15 - Tela Sobre

A classe TelaSobre é uma janela de diálogo simples e informativa.

```
 6  public class TelaSobre extends JDialog {
 7      private static final long serialVersionUID = 1L;
 8
 9      private Font boldFont = new Font("Arial", Font.BOLD, 12);
10      private Font normalFont = new Font("Arial", Font.PLAIN, 12);
11
12      public TelaSobre() {
56      }
57  }
```

Este bloco inicial define a classe TelaSobre como uma janela de diálogo, herdando de JDialog. No construtor, são definidas suas características essenciais: o título da janela será "Sobre", a operação de fechamento padrão (DISPOSE_ON_CLOSE) garantirá que apenas esta janela seja fechada sem encerrar a aplicação inteira, e a propriedade setModal(true) fará com que ela bloqueie a interação com a janela principal do sistema enquanto estiver aberta, forçando o usuário a focar nela.

Para a estrutura visual da janela, um JPanel é criado para servir como o contêiner principal de todo o conteúdo. A ele é aplicado um BoxLayout com a orientação Y_AXIS, o que força todos os componentes adicionados a se empilharem verticalmente, um abaixo do outro. Para um acabamento visual mais agradável, uma margem interna de 10 pixels é adicionada com createEmptyBorder. Adicionalmente, duas fontes, uma normal e uma em negrito, são definidas para definir o visual no texto que será exibido.

Cada linha de informação da tela é criada como um objeto JLabel com o alinhamento a esquerda definido pelo uso de setAlignmentX(Component.LEFT_ALIGNMENT). Em seguida, as fontes definidas anteriormente são aplicadas a cada JLabel para diferenciar os títulos dos nomes dos colaboradores.

E para finalizar, temos montagem da interface. Os objetos JLabel são adicionados um a um ao painel principal. Entre eles, o Box.createVerticalStrut() é utilizado para inserir espaçamentos verticais de tamanho fixo, controlando a distância entre as linhas de texto de forma precisa. Depois que o painel está completo, ele é adicionado ao JDialog. O método pack() é chamado para redimensionar a janela de forma inteligente, ajustando-a ao tamanho exato de seu conteúdo. Por fim, setLocationRelativeTo(null) a centraliza na tela e setVisible(true) a torna visível para o usuário.

16 - Tela de Login

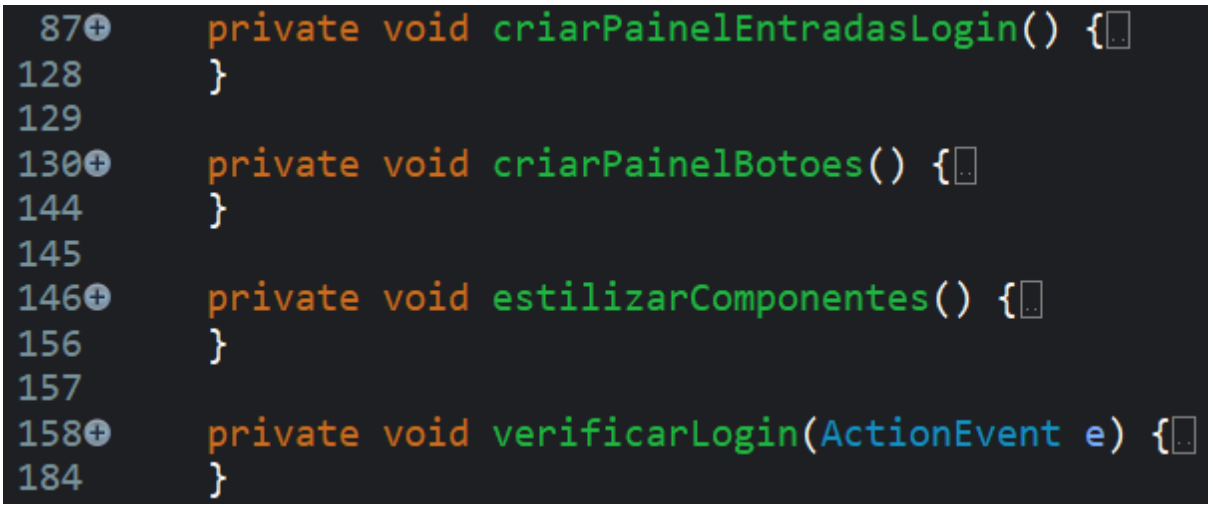
A TelaLogin é o ponto de entrada da aplicação. Ela é responsável por montar a interface com os campos de usuário e senha, configurar os botões "Entrar" e "Sair" e executar a lógica para autenticar as credenciais do usuário no banco de dados.

16a - Estrutura e Configuração da Janela

```
28  public class TelaLogin extends JFrame {
29      private static final long serialVersionUID = 1L; // Default serialV
30
31      // -- PAINEIS --
32      private String tituloJanela = "Login";
33      private JPanel painelPrincipal = new JPanel();
34      private JPanel painelEntradasLogin = new JPanel();
35      private JPanel painelBotoes = new JPanel();
36
37      // -- COMPONENTES DE ENTRADA --
38      private JLabel lbLogin = new JLabel("Usuário ou Email:");
39      private JTextField tfLogin = new JTextField();
40
41      private JLabel lbSenha = new JLabel("Senha:");
42      private JPasswordField pfSenha = new JPasswordField();
43
44      private JCheckBox ckbMostrarSenha = new JCheckBox("Mostrar senha");
45
46      private JButton btnEntrar = new JButton("Entrar");
47      private JButton btnSair = new JButton("Sair");
48
49      // -- FONTE E TAMANHO DAS ENTRADAS --
50      private Font panelFont = new Font("Arial", Font.BOLD, 14);
51      private Font labelFont = new Font("Arial", Font.BOLD, 12);
52      private Font inputFont = new Font("Arial", Font.PLAIN, 12);
53
54      private Dimension inputSize = new Dimension(200, 25);
55      private Dimension buttonSize = new Dimension(100, 30);
56
57      public TelaLogin() {
85      }
```

O bloco inicial define a TelaLogin como a janela principal da aplicação, herdando de JFrame. Dentro do construtor, são aplicadas configurações essenciais: o título da janela é definido como "Login", a operação de fechamento padrão (EXIT_ON_CLOSE) faz com que a aplicação inteira se encerre ao fechar esta janela, e a propriedade setResizable(false) impede que o usuário altere seu tamanho. A estrutura visual é baseada em um painel principal com BoxLayout vertical, que serve para empilhar o painel de entradas e o de botões.

16b - Os métodos



O método criarPainelEntradasLogin é responsável por construir a seção onde o usuário insere suas credenciais. Ele utiliza um GridBagLayout, um gerenciador de layout que permite alinhar os componentes de forma precisa em uma grade. Os rótulos e os campos de entrada são adicionados a este painel. Uma funcionalidade importante implementada aqui é a da caixa "Mostrar senha", cujo ActionListener alterna o caractere de eco do campo de senha entre o padrão ('●') e um caractere nulo, tornando o texto visível.

Já o método criarPainelBotoes monta a parte inferior da tela, que contém os botões de ação. Ele usa um FlowLayout alinhado à direita para posicionar os botões "Sair" e "Entrar". As ações são conectadas aqui: o botão "Sair" simplesmente fecha a janela (dispose), enquanto o botão "Entrar" é vinculado ao método verificarLogin. Uma melhoria de usabilidade crucial é a linha getRootPane().setDefaultButton(this.btnEntrar), que faz com que o botão "Entrar" seja acionado automaticamente quando o usuário pressiona a tecla Enter.

E por fim, o método verificarLogin é onde se encontra a lógica de autenticação. Quando acionado, ele primeiro obtém os dados dos campos de texto e senha. Em seguida, faz uma validação simples para garantir que ambos foram preenchidos. O núcleo da operação ocorre dentro de um bloco try-catch, onde uma instância de UsuarioDAO é criada para chamar o método autenticar. Se o método retornar um objeto Usuario (indicando sucesso), a TelaPrincipal é criada, o contexto do usuário é passado para ela, e a tela de login é fechada. Caso contrário, uma mensagem de erro apropriada é exibida ao usuário.

17 - Tela Principal

É a classe que cria a janela principal do sistema, que funciona como um "contêiner" após o login. Ela é responsável por montar a barra de menus superior e o painel de abas, controlando o que é exibido com base nos privilégios do usuário logado.

17a - Configuração da Janela

```
16  public class TelaPrincipal extends JFrame {
17      private static final long serialVersionUID = 1L;
18
19      private Usuario usuarioLogado;
20
21      public TelaPrincipal(Usuario usuario) {
87      }
```

O bloco inicialmente define a TelaPrincipal como a janela principal da aplicação, herdando de JFrame. Seu construtor recebe o objeto Usuario que foi autenticado na tela de login, o que é fundamental para o restante da lógica.

A primeira ação é personalizar o título da janela com uma mensagem de boas-vindas, exibindo o nome do usuário. Em seguida, são aplicadas configurações padrão, como definir um tamanho mínimo e garantir que a aplicação seja encerrada (EXIT_ON_CLOSE) quando esta janela for fechada.

Em seguida, a barra de menus superior da aplicação é construída. Ela é composta por dois menus principais (JMenu): "Cadastrar" e "Sistema". Cada menu contém itens clicáveis (JMenuItem) que disparam ações específicas através de ActionListeners. Por exemplo, o item "Sair" tem a lógica de fechar a janela principal (dispose) e abrir uma nova TelaLogin, efetivamente realizando um logout. Ao final, a barra de menus completa é adicionada à janela.

Após a construção do menu superior, o bloco seguinte do código é responsável por criar o painel central da aplicação, que utiliza um JTabbedPane para organizar as diferentes telas de gerenciamento em abas. A aba "Cliente" é adicionada para todos os usuários, assim como também outras abas que tem uso liberado para todos os privilégios de usuários, serão adicionadas aqui posteriormente com a expansão do sistema. Um detalhe importante, é que cada tela de gerenciamento (que é um JPanel) é envolvida por um JScrollPane. Isso garante que, se a janela for redimensionada para um tamanho menor, o conteúdo da aba se torne rolável, mantendo a usabilidade.

E por fim, o código verifica o nível de privilégio do usuarioLogado. É nesse bloco onde é definido as abas que serão visíveis apenas se o usuário for um ADMINISTRADOR. Assim, apenas usuários desse privilégio terão os itens de menu e as abas para gerenciamento de "Funcionários" e "Usuários" adicionados à interface. Isso implementa o Controle de Acesso Baseado em Papéis diretamente na camada de visualização, garantindo que usuários com permissões padrão não tenham acesso visual ou funcional a áreas restritas do sistema.