Logo da Instituição
UNISUL
DAVI SAMUEL DA SILVA
PEDRO HENRIQUE LIMA DE ARAÚJO

SISTEMA DE GERENCIAMENTO DE ESTOQUE

Florianópolis
2025

DAVI SAMUEL DA SILVA
PEDRO HENRIQUE LIMA DE ARAÚJ

SISTEMA DE GERENCIAMENTO DE ESTOQUE

Trabalho A3 apresentado ao Curso de Programação de Soluções Computacionais da Unisul como como parte das atividades acadêmicas do semestre **[1° de 2025]**.

Orientador: Prof. Nome do Professor, Dr./Ms./Bel./Lic.

Florianópolis
2025

RESUMO

Este trabalho desenvolve um **Sistema de Gerenciamento de Estoque** em Java para otimizar o controle de produtos em pequenos negócios. A necessidade de substituir planilhas manuais por uma solução automatizada que evita erros e perdas. O objetivo central foi criar uma ferramenta intuitiva para cadastrar produtos/categorias, controlar níveis de estoque (mínimo/máximo) e gerar relatórios estratégicos. O método adotou a arquitetura **Model-View-DAO**, com interface gráfica em Swing e banco de dados MySQL, validada através de testes manuais e simulações de uso. Alertas automáticos para estoque baixo/excesso e relatórios financeiros precisos. Com potencial para expansão com leitores de código de barras e módulo de compras.

Palavras-chave: Controle de Estoque. Sistema Java. Gestão de Produtos. Relatórios Gerenciais. Automação Comercial.

LISTA DE TABELAS

LISTA DE FIGURAS

1 INTRODUÇÃO

A gestão eficiente de estoques representa um desafio crítico para micro e pequenas empresas, onde processos manuais frequentemente resultam em erros operacionais, perdas financeiras e rupturas de suprimentos. Este trabalho desenvolve e implementa um Sistema de Controle de Estoque utilizando Java e padrões de arquitetura fundamentado no padrão Model-View-DAO (MVD), o sistema organiza-se em três camadas interconectadas: a camada Model (entidades como Produto e Categoria com validações de negócio, exemplificada pelo método setQuantidadeMinima(int min) que assegura valores positivos), a camada DAO (operações de persistência atômica via JDBC, como deletar(int id) em ProdutoDAO.java que previne SQL Injection com PreparedStatement), e a camada View (interface gráfica em Swing com feedback imediato, como a validação em tempo real em atualizarQuantidade(int operacao) de ProdutoView.java). Os objetivos específicos incluíram: (a) automatização de operações CRUD para produtos/categorias, (b) controle preventivo de níveis mínimo/máximo de estoque, e (c) geração dinâmica de relatórios estratégicos (implementados em RelatorioView.java com abas para balanço financeiro e lista de preços dentre outras). A metodologia adotou desenvolvimento iterativo em três etapas: modelagem conceitual, implementação progressiva das camadas, e refinamento baseado em testes manais. Fornece alertas preventivos para estoques críticos.

[1] Uma transação atômica é aquela que ocorre de forma indivisível: ou todas as suas mudanças são aplicadas com sucesso (COMMIT), ou são completamente revertidas (ROLLBACK), garantindo a integridade dos dados.

2 ANÁLISE DO SITEMA

2.1 REQUISITOS FUNCIONAIS

1. **Gerenciamento de Produtos** - Cadastrar, editar, consultar e excluir produtos com os campos: Nome, Preço unitário, Unidade de medida, Quantidade em estoque, Quantidade mínima, Quantidade máxima e Categoria.
2. **Gerenciamento de Categorias** - Cadastrar, editar, consultar e excluir categorias com os campos: Nome, Tamanho (Pequeno/Médio/Grande) e Embalagem (Lata/Vidro/Plástico).
3. **Controle de Movimentações** - Registrar entrada/saída de produtos com alertas automáticos para estoque abaixo do mínimo ou acima do máximo.
4. **Ajuste de Preços** - Aplicar reajuste percentual em todos os produtos simultaneamente.
5. **Relatórios** - Gerar: Lista de Preços, Balanço Físico/Financeiro, Produtos abaixo/sobre estoque e Quantidade por categoria, todos ordenados alfabeticamente.

2.2 REQUISITOS NÃO FUNCIONAIS

1. **Banco de Dados** - Armazenar produtos, categorias e movimentações.
2. **Interface** - Telas para CRUD, movimentações e relatórios.
3. **Desempenho** - Tempo de resposta adequado para operações diárias.
4. **Validações** - Impedir cadastros incompletos e quantidades mínimas / máximas.

2.3 REGRAS DE NEGÓCIO

1. Atualizar estoque automaticamente (entradas aumentam, saídas diminuem).
2. Emitir alertas quando estoque estiver abaixo do mínimo ou acima do máximo.
3. Todo produto deve ter categoria vinculada.
4. Bloquear saídas sem estoque suficiente.
5. Relatórios sempre em ordem alfabética por nome.

2.4 TABELA DE ESPECIFICAÇOES DE REQUISITOS.

Tabela 1 (Especificações de requisitos)

Componente	Detalhes
FUNCIONALIDADES	- CRUD completo de produtos e categorias - Movimentações com alertas (mínimo/máximo) - Relatórios: lista de preços, balanço, níveis de estoque
DEPENDÊNCIAS	- Vinculação obrigatória entre produto e categoria
VALIDAÇÕES	- Frontend: Combobox de categorias - Regras: Verificações no DAO - Banco de Dados: Chave estrangeira (FK)
REGRAS DE NEGÓCIO	- Bloqueio de exclusão de categorias com produtos - Alertas automáticos de estoque
ARQUITETURA	- Base para modelagem do BD e classes de domínio - Define contratos de interface

2.5 CASO DE USO

O diagrama abaixo, após a tabela de especificação do caso de uso, destaca as dependências críticas do sistema, particularmente a relação obrigatória entre produtos e categorias, onde todo produto deve estar vinculado a uma categoria previamente cadastrada. Essa dependência é implementada através de três mecanismos de validação: uma combobox na interface do usuário (ProdutoView) que limita a seleção a categorias existentes, verificações no ProdutoDAO para garantir a integridade dos dados, e a chave estrangeira no banco de dados que mantém o relacionamento.

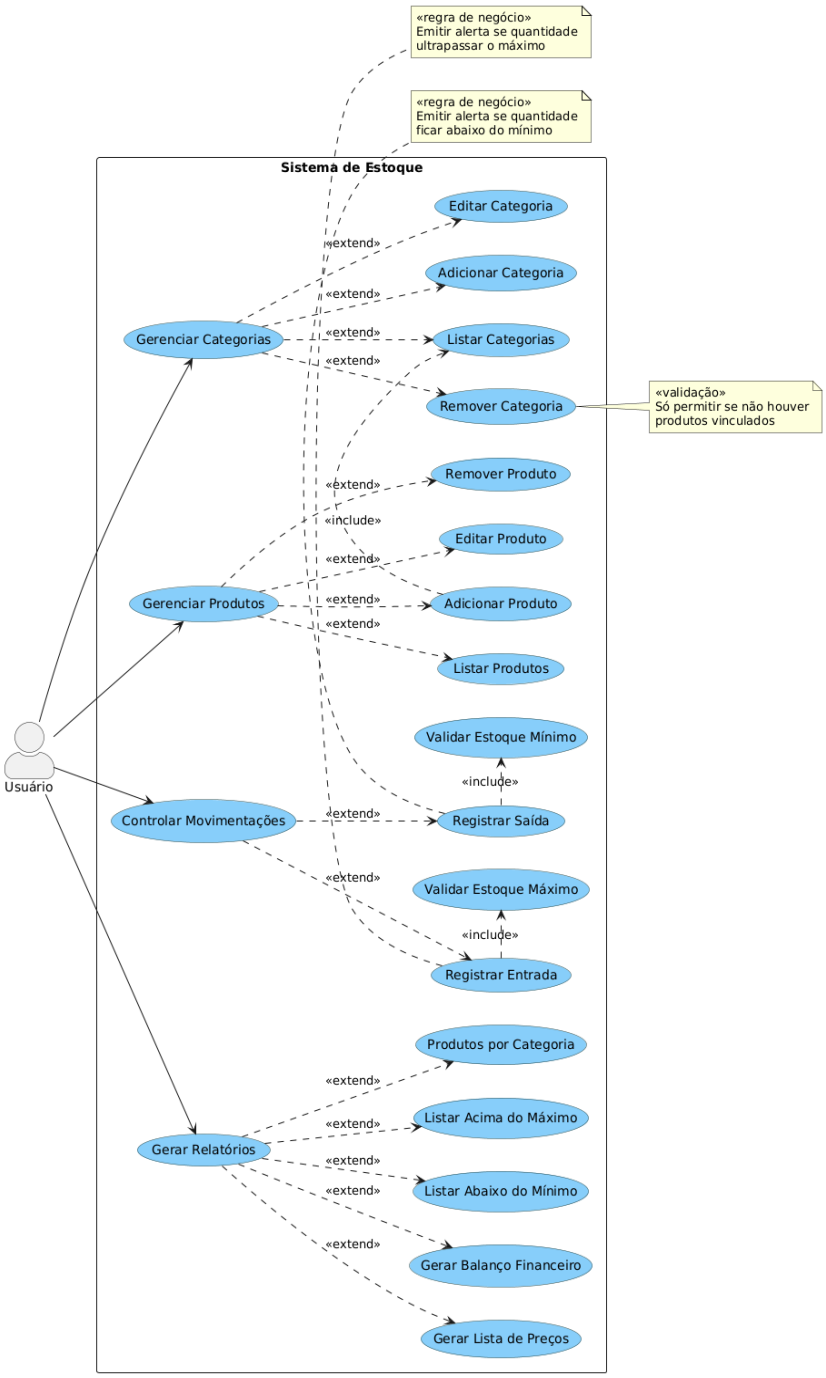
Além de definir o escopo, o diagrama documenta importantes regras de negócio, como os alertas automáticos para estoque abaixo do mínimo ou acima do máximo durante as movimentações, e a restrição que impede a exclusão de categorias que possuam produtos vinculados.

Essas validaçõessão representadas tanto visualmente no diagrama quanto em notas explicativas complementares.Como artefato central da análise, este diagrama de casos de uso fornece os requisitos necessários para a modelagem subsequente do sistema, incluindo a definição das classes de domínio, o design da camada de persistência e os contratos de interface. Ele estabelece claramente os limites do sistema e serve como referência para toda a equipe de desenvolvimento, garantindo que todas as funcionalidades essenciais sejam adequadamente implementadas e integradas.

Tabela 2 (especificação do diagrama do caso de uso)

Caso de uso	Descrição	Elemento no diagrama
Gerenciar categorias	Adicionar/Editar/Listar/Removr categorias (validação: não remover com produtos ativos)	Editor Categoria. Remover Categoria (nota).
Gerencia produtos	CRUD de produtos com vinculação obrigatória a categorias (combobox + FK)	Adicionar/Editar Produto. Exige categorias (seta).
Registrar movimentações	Entrada e saída com validações de estoque mínimo e máxix'mo	Registrar Entrada/Saída. Abaixo do Mínimo (nota)
Gerar relatórios	Listas de preços, balanço e níveis de estoque (sempre ordenados alfabeticamente)	Gerar Relatórios. Lista de Preços.
Validação do sistema	Garantir integridade: 1- FrontEnd: Combobox 2- BackEnd: Verificação no DAO 3- Banco de Dados: Chave estrangeira	Validações de Produto. Combobox (nota) . Verificação no ProdutoDAO. Chave estrangeira (BD.

Imagem 1 (Diagrama de caso de uso)



2.6 CLASSES

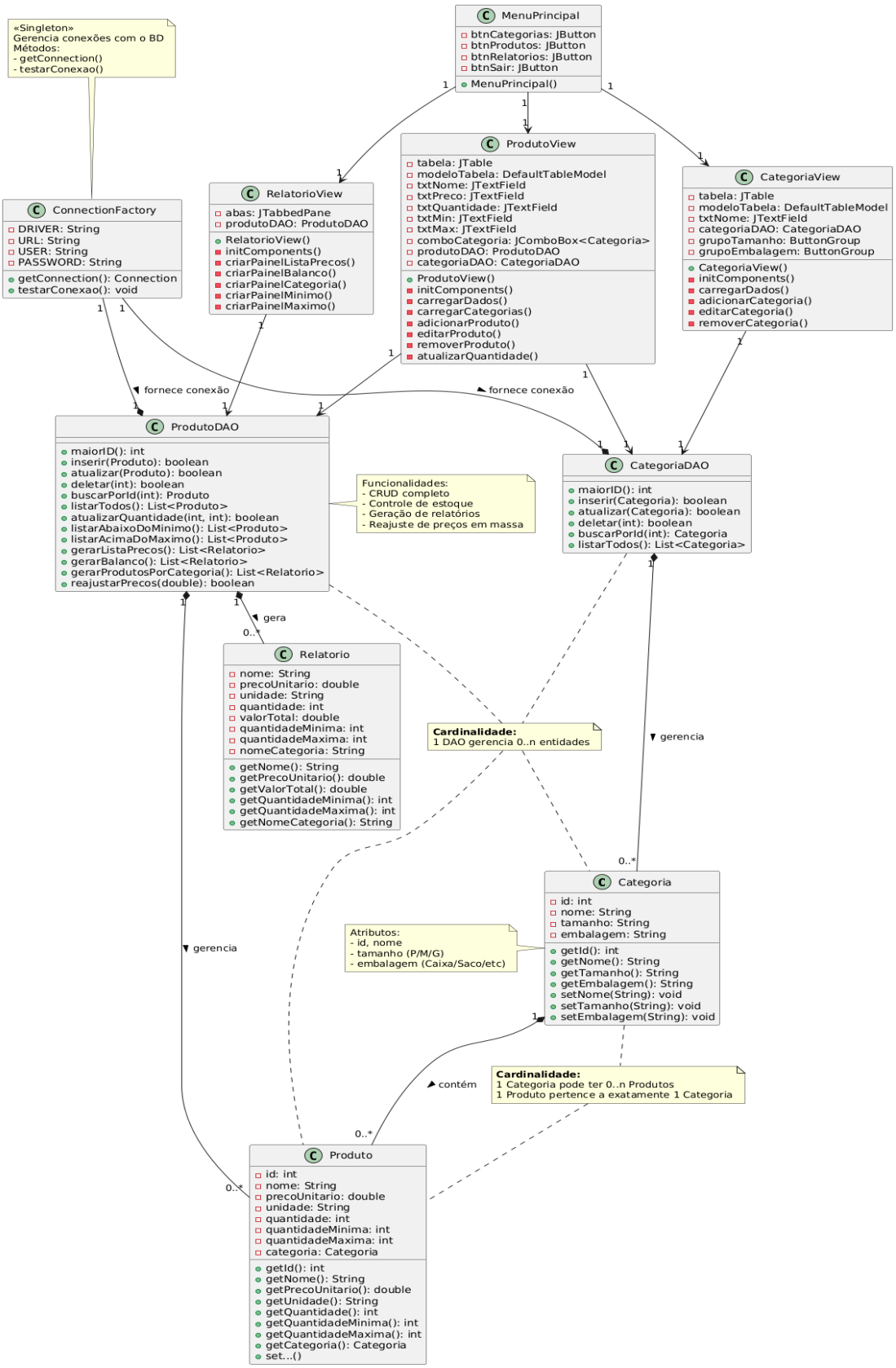
O diagrama de classes representa o sistema de gestão, desenvolvido com base em princípios de modularidade, separação de responsabilidades e reutilização de código. A

arquitetura adotada segue um padrão Model-View-DAO (MVD), implícito, organizando o sistema em três camadas principais: interface gráfica (View), lógica de negócios (Model) e acesso a dados (DAO). Essa divisão garante maior clareza no desenvolvimento, facilitando a manutenção e a escalabilidade do software.

As Views (CategoriaView, ProdutoView e RelatorioView) são responsáveis pela interação com o usuário, exibindo dados e capturando entradas. Elas se comunicam com os DAOs (CategoriaDAO e ProdutoDAO), que, por sua vez, gerenciam todas as operações de persistência no banco de dados. A ConnectionFactory atua como um ponto centralizado para conexões com o banco, seguindo o padrão Singleton para evitar múltiplas instâncias desnecessárias e otimizar o uso de recursos.

[2] O Singleton garante que uma classe tenha apenas uma instância globalmente acessível, melhorando o desempenho e evitando conflitos na execução.

Imagem 2 (diagrama de classe)


«Singleton»
Gerencia conexões com o BD
Métodos:
- getConnection()
- testarConexao()
MenuPrincipal
- btnCategorias: JButton
- btnProdutos: JButton
- btnRelatorios: JButton
- btnSair: JButton
+ MenuPrincipal()
ProdutoView
- tabela: JTable
- modeloTabela: DefaultTableModel
- txtNome: JTextField
- txtPreco: JTextField
- txtQuantidade: JTextField
- txtMin: JTextField
- txtMax: JTextField
- comboCategoria: JComboBox<Categoria>
- produtoDAO: ProdutoDAO
- categoriaDAO: CategoriaDAO
+ ProdutoView()
- initComponents()
- carregarDados()
- carregarCategorias()
- adicionarProduto()
- editarProduto()
- removerProduto()
- atualizarQuantidade()
CategoriaView
- tabela: JTable
- modeloTabela: DefaultTableModel
- txtNome: JTextField
- categoriaDAO: CategoriaDAO
- grupoTamanho: ButtonGroup
- grupoEmbalagem: ButtonGroup
+ CategoriaView()
- initComponents()
- carregarDados()
- adicionarCategoria()
- editarCategoria()
- removerCategoria()
RelatorioView
- abas: JTabbedPane
- produtoDAO: ProdutoDAO
+ RelatorioView()
- initComponents()
- criarPainelListaPrecos()
- criarPainelBalanco()
- criarPainelCategoria()
- criarPainelMinimo()
- criarPainelMaximo()
ConnectionFactory
- DRIVER: String
- URL: String
- USER: String
- PASSWORD: String
+ getConnection(): Connection
+ testarConexao(): void
fornece conexão
fornece conexão
ProdutoDAO
+ maiorID(): int
+ inserir(Produto): boolean
+ atualizar(Produto): boolean
+ deletar(int): boolean
+ buscarPorId(int): Produto
+ listarTodos(): List<Produto>
+ atualizarQuantidade(int, int): boolean
+ listarAbaixoDoMinimo(): List<Produto>
+ listarAcimaDoMaximo(): List<Produto>
+ gerarListaPrecos(): List<Relatorio>
+ gerarBalanco(): List<Relatorio>
+ gerarProdutosPorCategoria(): List<Relatorio>
+ reajustarPrecos(double): boolean
Funcionalidades:
- CRUD completo
- Controle de estoque
- Geração de relatórios
- Reajuste de preços em massa
CategoriaDAO
+ maiorID(): int
+ inserir(Categoria): boolean
+ atualizar(Categoria): boolean
+ deletar(int): boolean
+ buscarPorId(int): Categoria
+ listarTodos(): List<Categoria>
gera
Relatorio
- nome: String
- precoUnitario: double
- unidade: String
- quantidade: int
- valorTotal: double
- quantidadeMinima: int
- quantidadeMaxima: int
- nomeCategoria: String
+ getNome(): String
+ getPrecoUnitario(): double
+ getValorTotal(): double
+ getQuantidadeMinima(): int
+ getQuantidadeMaxima(): int
+ getNomeCategoria(): String
Cardinalidade:
1 DAO gerencia 0..n entidades
gerencia
gerencia
Categoria
- id: int
- nome: String
- tamanho: String
- embalagem: String
+ getId(): int
+ getNome(): String
+ getTamanho(): String
+ getEmbalagem(): String
+ setNome(String): void
+ setTamanho(String): void
+ setEmbalagem(String): void
Atributos:
- id, nome
- tamanho (P/M/G)
- embalagem (Caixa/Saco/etc)
contém
Cardinalidade:
1 Categoria pode ter 0..n Produtos
1 Produto pertence a exatamente 1 Categoria
Produto
- id: int
- nome: String
- precoUnitario: double
- unidade: String
- quantidade: int
- quantidadeMinima: int
- quantidadeMaxima: int
- categoria: Categoria
+ getId(): int
+ getNome(): String
+ getPrecoUnitario(): double
+ getUnidade(): String
+ getQuantidade(): int
+ getQuantidadeMinima(): int
+ getQuantidadeMaxima(): int
+ getCategoria(): Categoria
+ set...()

2.7 DIAGRAMA DE FLUXO

2.7.1 Início do Sistema

O usuário inicia o sistema pelo Menu Principal, que oferece opções de gestão. Ao selecionar "Gerenciar Categorias", o sistema carrega a tela específica e consulta o banco de dados para exibir as categorias existentes através do CategoriaDAO. Castra produtos preenchendo o formulário. Executa o INSERT no banco de dados.

2.7.2 Cadastro de Produtos

Ao acessar "Gerenciar Produtos", o sistema primeiro carrega as categorias disponíveis (pré-requisito essencial) antes de exibir os produtos existentes. Para cadastrar um novo produto, o usuário preenche todos os campos do formulário, incluindo a seleção de uma categoria válida campos validos. Executa o INSERT no banco de dados. Alertas, erros e sucessos são gerados.

2.7.3 Controle de Estoque

Na tela de produtos, movimentações de estoque são realizadas através dos botões "Entrada" (aumento) e "Saída" (redução). O sistema solicita a quantidade, valida contra o estoque atual e limites configurados, e executa o UPDATE no banco via ProdutoDAO. Alertas, erros e sucessos são gerados.

2.7.4 Relatórios

A opção "Relatórios" aciona consultas específicas no banco de dados para cada tipo de análise (lista de preços, balanço, produtos por categoria, e alertas de estoque). O ProdutoDAO transforma os ResultSets em objetos Relatorio, que são formatados e exibidos em abas separadas.

2.7.5 Persistência e Conexões

A ConnectionFactory centraliza o gerenciamento de conexões com o banco, seguindo o padrão Singleton. Todos os DAOs utilizam esta fábrica para obter conexões, que são automaticamente fechadas após cada operação.

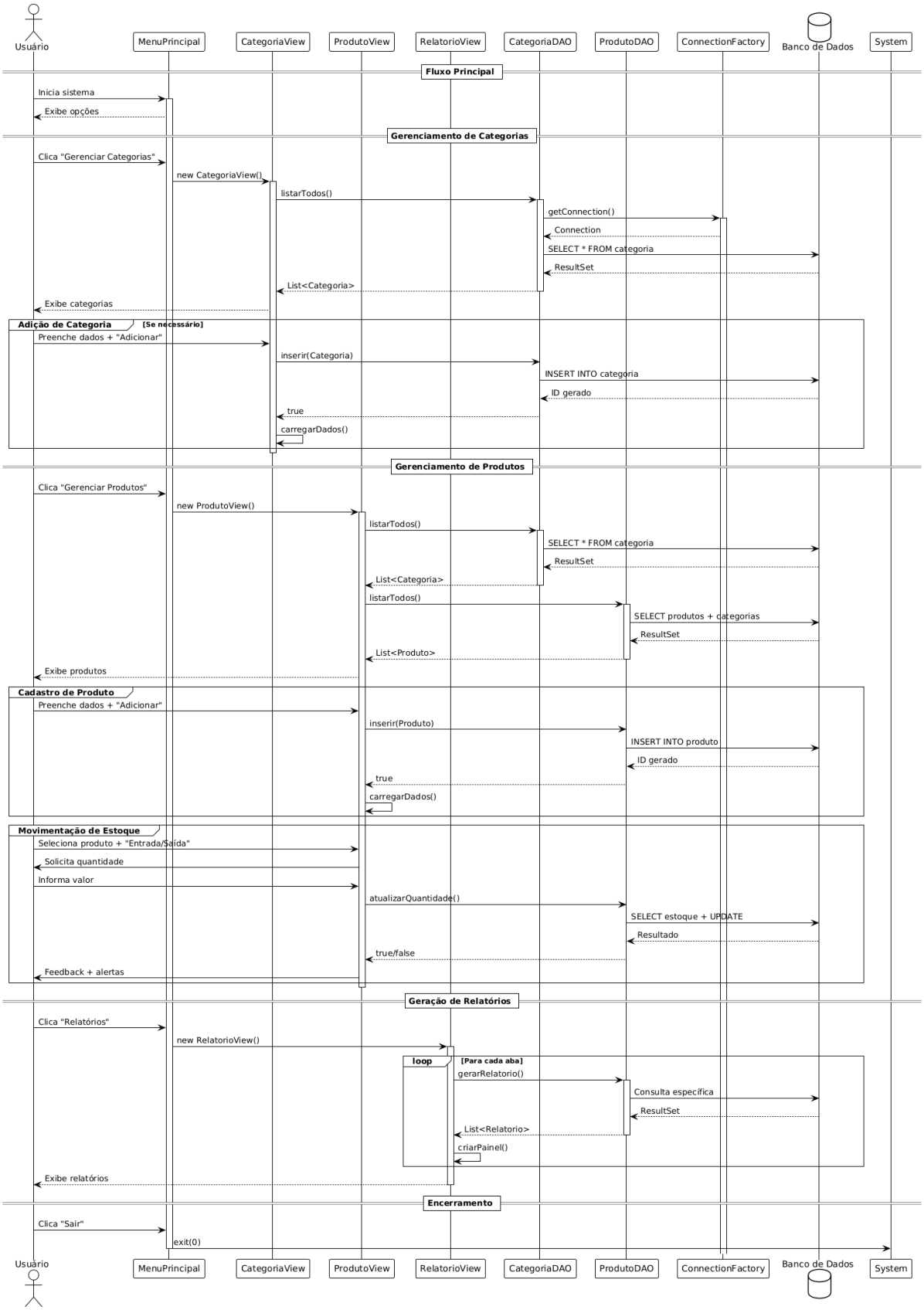
2.7.6 Validações e Feedback

Em cada etapa, o sistema realiza validações de dados (campos obrigatórios, formatos numéricos, consistência de estoque) e fornece feedback imediato ao usuário através de mensagens dialógicas. A interface é atualizada automaticamente após operações bem-sucedidas.

2.7.7 Encerramento

A saída do sistema encerra completamente a aplicação, garantindo que todas as conexões com o banco sejam finalizadas e recursos liberados.

Imagem 3 (diagrama de floxo)


Usuário
MenuPrincipal
CategoriaView
ProdutoView
RelatorioView
CategoriaDAO
ProdutoDAO
ConnectionFactory
Banco de Dados
System
Fluxo Principal
Inicia sistema
Exibe opções
Gerenciamento de Categorias
Clica "Gerenciar Categorias"
new CategoriaView()
listarTodos()
getConnection()
Connection
SELECT * FROM categoria
ResultSet
List<Categoria>
Exibe categorias
Adição de Categoria
[Se necessário]
Preenche dados + "Adicionar"
inserir(Categoria)
INSERT INTO categoria
ID gerado
true
carregarDados()
Gerenciamento de Produtos
Clica "Gerenciar Produtos"
new ProdutoView()
listarTodos()
SELECT * FROM categoria
ResultSet
List<Categoria>
listarTodos()
SELECT produtos + categorias
ResultSet
List<Produto>
Exibe produtos
Cadastro de Produto
Preenche dados + "Adicionar"
inserir(Produto)
INSERT INTO produto
ID gerado
true
carregarDados()
Movimentação de Estoque
Seleciona produto + "Entrada/Saída"
Solicita quantidade
Informa valor
atualizarQuantidade()
SELECT estoque + UPDATE
Resultado
true/false
Feedback + alertas
Geração de Relatórios
Clica "Relatórios"
new RelatorioView()
loop
[Para cada aba]
gerarRelatorio()
Consulta específica
ResultSet
List<Relatorio>
criarPainel()
Exibe relatórios
Encerramento
Clica "Sair"
exit(0)

2.8 MODELAGEM DE DADOS

O código SQL inicia com a criação do banco de dados db_estoque e sua seleção imediata para uso. Em seguida, define-se a tabela produto com os campos essenciais para o armazenamento de informações sobre os itens em estoque. Cada produto possui um identificador único gerado automaticamente (id), nome, preço unitário, unidade de medida, quantidade atual em estoque, níveis mínimo e máximo de estoque.

Posteriormente, cria-se a tabela categoria para classificar os produtos. Esta tabela inclui identificador único autoincrementável, nome (com restrição de unicidade para evitar duplicatas), tamanho (com valores pré-definidos "Pequeno", "Médio" ou "Grande" e padrão "Médio"), e tipo de embalagem (obrigatoriamente "Lata", "Plástico" ou "Vidro").

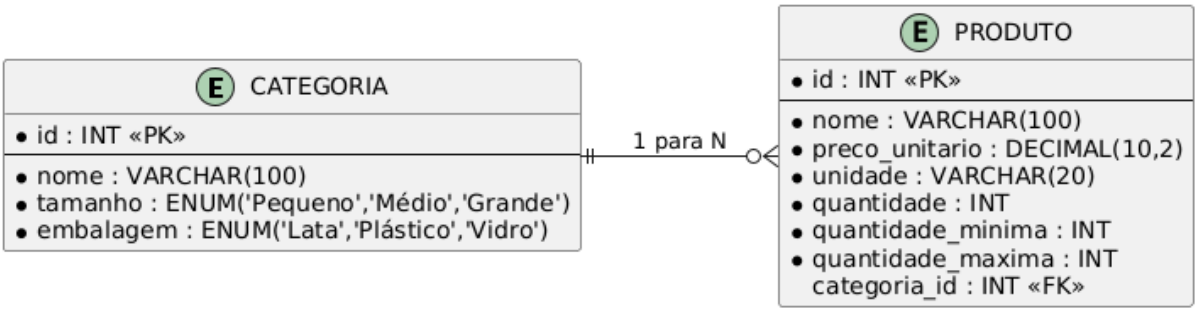
Por fim, estabelece a relação entre as tabelas através de uma chave estrangeira. O campo categoria_id na tabela produto é vinculado ao id da tabela categoria, criando uma associação onde cada produto pode pertencer a uma categoria específica. Esta estrutura relacional organiza os dados de forma coesa, permitindo consultas eficientes e mantendo a consistência entre produtos e suas categorias correspondentes, base para todo o sistema de gestão de estoque.

Imagem 4 (script SQL)

```sql
CREATE DATABASE IF NOT EXISTS db_estoque;

USE db_estoque;

CREATE TABLE IF NOT EXISTS produto (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    preco_unitario DECIMAL(10, 2) NOT NULL,
    unidade VARCHAR(20) NOT NULL,
    quantidade INT NOT NULL,
    quantidade_minima INT NOT NULL,
    quantidade_maxima INT NOT NULL,
    categoria_id INT
)ENGINE=InnoDB;

CREATE TABLE  IF NOT EXISTS categoria (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    tamanho ENUM('Pequeno', 'Médio', 'Grande') NOT NULL DEFAULT 'Médio',
    embalagem ENUM('Lata', 'Plástico', 'Vidro') NOT NULL
) ENGINE=InnoDB;

ALTER TABLE produto
ADD CONSTRAINT fk_produto_categoria
FOREIGN KEY (categoria_id)
REFERENCES categoria(id);
```

Modelo Entidade-Relacionamento:

Imagem 5 (diagrama ER)



3 DESENVOLVENDO O SISTEMA.

3.1 BACK-END

3.1.1 Models (Modelos)

O primeiro passo para o desenvolvimento do sistema em questão é a criação das classes que servirão de modelo. As classes Produto e Categoria constituem a espinha dorsal do modelo de dados do sistema.

A classe **Categoria.java** define grupos lógicos para organização do estoque, encapsulando propriedades essenciais como identificador único, nome descritivo, classificação dimensional e tipo de embalagem. Sua estrutura simples e autônoma permite a categorização eficiente de itens sem complexidades operacionais.

A classe **Produto.java** representa os itens físicos gerenciados pelo sistema, incorporando atributos como código identificador, denominação, valor unitário, unidade de medida e controles de estoque (quantidades atual, mínima e máxima). A conexão entre ambas as classes se materializa através de uma associação direta, onde cada produto se vincula a uma categoria específica, estabelecendo hierarquia e organização no inventário.

A classe **Relatorio.java** desempenha um papel fundamental como modelo especializado para transferência de dados consolidados, se diferenciando das demais entidades por sua natureza híbrida e orientada a análise. Ao contrário de Produto e Categoria, que representam entidades de negócio puras, esta classe atua como um contêiner de dados agregados, projetado exclusivamente para suportar a geração de relatórios estratégicos. Foco em DTO (Data Transfer Object): Cada construtor corresponde a uma consulta específica do sistema, permitindo que o ProdutoDAO retorne dados já estruturados para a camada de visualização RelatorioView.

Note que os construtores se diferenciam por suas assinaturas de métodos.

3.1.1.1 Categoria.java

O primeiro construtor é essencial para a aplicação porque, em primeiro momento, não há ID's. Isso permite que o objeto seja instanciado sem nem um dado armazenado no banco:

Imagem 6(construtor Categori.java)

```
// para iniciar categorias com o banco vazio
public Categoria(String nome, String tamanho, String embalagem) {
    this.nome = nome;
    this.tamanho = tamanho;
    this.embalagem = embalagem;
}

public Categoria(int id, String nome, String tamanho, String embalagem) {
    this.id = id;
    this.nome = nome;
    this.tamanho = tamanho;
    this.embalagem = embalagem;
}

// Getters e Setters
```

Seguido de seus Getters e Setters:

Imagem 7(Getters e Setters Categoria.java)

```
// Getters e Setters
public int getId() {return id;}
public void setId(int id) {this.id = id;}
public String getNome() {return nome;}
public void setNome(String nome) {this.nome = nome;}
public String getTamanho() {return tamanho;}
public void setTamanho(String tamanho) {this.tamanho = tamanho;}
public String getEmbalagem() {return embalagem;}
public void setEmbalagem(String embalagem) {this.embalagem = embalagem;}
```

3.1.1.2 Produto.java

Da mesma forma que o construtor sem ID na Categoria.java, no Produto.java também serve um construtor sem ID para instanciar o objeto sem dados prévios:

Imagem 8 (Construtores Produto.java)

```java
// para produtos com  banco de dados vasio
public Produto(String nome, double precoUnitario, String unidade, int quantidade,
        int quantidadeMinima, int quantidadeMaxima, Categoria categoria) {
    this.nome = nome;
    this.precoUnitario = precoUnitario;
    this.unidade = unidade;
    this.quantidade = quantidade;
    this.quantidadeMinima = quantidadeMinima;
    this.quantidadeMaxima = quantidadeMaxima;
    this.categoria = categoria;
}

public Produto(int id, String nome, double precoUnitario, String unidade, int quantidade,
        int quantidadeMinima, int quantidadeMaxima, Categoria categoria) {
    this.id = id;
    this.nome = nome;
    this.precoUnitario = precoUnitario;
    this.unidade = unidade;
    this.quantidade = quantidade;
    this.quantidadeMinima = quantidadeMinima;
    this.quantidadeMaxima = quantidadeMaxima;
    this.categoria = categoria;
}
```

E seus Getters e Setters:

Imagem 9(Getters e Setters Produto.java)

```java
public int getId() {return id;}
public void setId(int id) {this.id = id;}
public String getNome() {return nome;}
public void setNome(String nome) {this.nome = nome;}
public double getPrecoUnitario() {return precoUnitario;}
public void setPrecoUnitario(double precoUnitario) {this.precoUnitario = precoUnitario;}
public String getUnidade() {return unidade;}
public void setUnidade(String unidade) {this.unidade = unidade;}
public int getQuantidade() {return quantidade;}
public void setQuantidade(int quantidade) {this.quantidade = quantidade;}
public int getQuantidadeMinima() {return quantidadeMinima;}
public void setQuantidadeMinima(int quantidadeMinima) {this.quantidadeMinima = quantidadeMinima;}
public int getQuantidadeMaxima() {return quantidadeMaxima;}
public void setQuantidadeMaxima(int quantidadeMaxima) {this.quantidadeMaxima = quantidadeMaxima;}
public Categoria getCategoria() {return categoria;}
public void setCategoria(Categoria categoria) {this.categoria = categoria;}
```

3.1.1.3 Relatorio.java

Para atender aos diferentes tipos de relatórios, foram implementados construtores específicos na classe Relatório, cada um inicializando apenas os campos necessários para cada cenário.

Assinatura do construtor [0]: **nome do produto, preço unitário, unidade de medida e categoria:**

Imagem 10(construtor [0] Relatori.java)

```java
// Para Preços
public Relatorio(String nome, double precoUnitario, String unidade, String categoria) {
    this.nome = nome;
    this.precoUnitario = precoUnitario;
    this.unidade = unidade;
    this.categoria = categoria;
}
```

Assinatura do construtor [1]: **nome do produto, quantidade em estoque, preço unitário e valor total do estoque:**

Imagem 11 (construtor [1] Relatori.java)

```java
// Para Balanço
public Relatorio(String nome, int quantidade, double precoUnitario, double valorTotal) {
    this.nome = nome;
    this.quantidade = quantidade;
    this.precoUnitario = precoUnitario;
    this.valorTotal = valorTotal;
}
```

Assinatura do construtor [2]: **nome do produto, quantidade em estoque, quantidade máxima e quantidade mínima:**

Imagem 12 (construtor [2] Relatori.java)

```java
// Para Produtos abaixo do mínimo/acima do máximo
public Relatorio(String nome, int quantidade, int quantidadeMinima, int quantidadeMaxima) {
    this.nome = nome;
    this.quantidade = quantidade;
    this.quantidadeMinima = quantidadeMinima;
    this.quantidadeMaxima = quantidadeMaxima;
}

// Para Produtos por Categoria
```

Assinatura do construtor [3]: **categoria e quantidade de produto:**

Imagem 13 (construtor [3] Relatori.java)

```java
// Para Produtos por Categoria
public Relatorio(String nomeCategoria, int quantidadeProdutos) {
    this.nomeCategoria = nomeCategoria;
    this.quantidadeProdutos = quantidadeProdutos;
}
```

Aqui há uma diferença entre os modelos de Produtos e Categorias. Nos Relatórios é feita apenas as consultas no banco de dados(**Getters**). Não necessitando de **Setters**:

Imagem 14 (Getters Relatori.java)

```
// Getters
public String getNome() {return nome;}
public double getPrecoUnitario() {return precoUnitario;}
public String getUnidade() {return unidade;}
public String getCategoria() {return categoria;}
public int getQuantidade() {return quantidade;}
public double getValorTotal() {return valorTotal;}
public int getQuantidadeMinima() {  return quantidadeMinima;}
public int getQuantidadeMaxima() {return quantidadeMaxima;}
public String getNomeCategoria() {return nomeCategoria;}
public int getQuantidadeProdutos() {return quantidadeProdutos;}
```

3.1.2 DAO (Data Access Object)

As classes **CategoriaDAO**, **ProdutoDAO** e **ConnectionFactory** constituem a infraestrutura de persistência do sistema, atuando como mediadoras entre o modelo de domínio e o banco de dados. Sua arquitetura segue princípios de coesão e encapsulamento, onde cada DAO assume responsabilidades específicas de manipulação de dados para sua respectiva entidade.

A **CategoriaDAO** concentra-se na gestão de grupos lógicos do estoque, implementando operações essenciais com validação de integridade. Seu diferencial está na prevenção de exclusões que comprometam a consistência dos dados, garantindo que categorias vinculadas a produtos não sejam removidas inadvertidamente. Além das operações básicas de CRUD, oferece funcionalidades estratégicas como a identificação automática de novos ID's e listagem ordenada para otimização de interfaces.

O **ProdutoDAO** destaca-se pela complexidade operacional, gerenciando não apenas o ciclo de vida dos produtos, mas também regras de negócio críticas. Implementa:

Controle dinâmico de estoque com validação de saldo negativo

Geração de relatórios estratégicos (balanço financeiro, níveis de estoque)

Operações em massa como reajuste percentual de preços

Recuperação de dados associados via JOINs com categorias

Sua estrutura reflete a natureza central dos produtos no sistema, integrando métricas operacionais e financeiras.

A **ConnectionFactory** atua como alicerce técnico, abstraindo a complexidade de conexões JDBC. Seu design *Factory Method* simplifica a manutenção de credenciais e promove

reuso em toda a camada de persistência. A capacidade de testar conexões agiliza a detecção de problemas de infraestrutura.

> A classe ConnectionFactory segue o padrão de criação Factory Method, encapsulando a lógica de conexão e permitindo maior reutilização e manutenção centralizada. Isso evita a repetição de código em cada DAO e melhora a escalabilidade do sistema." (MERC, 2025)

3.1.2.1 Categoria.DAO.

Dentre os 8 métodos que há na classe se destaca o de inserção, usa a própria conexão para retornar os ID's atualizados:

Imagem 15 (Método inserir Relatori.java)

```
public boolean inserir(Categoria categoria) {
    String sql = "INSERT INTO categoria (nome, tamanho, embalagem)"
            + " VALUES (?, ?, ?)";

    try (Connection conexao = ConnectionFactory.getConnection();
         PreparedStatement stmt = conexao.prepareStatement(sql,
                 Statement.RETURN_GENERATED_KEYS)) {
        stmt.setString(1, categoria.getNome());
        stmt.setString(2, categoria.getTamanho());
        stmt.setString(3, categoria.getEmbalagem());
        if (stmt.executeUpdate() > 0) {
            try (ResultSet rs = stmt.getGeneratedKeys()) {
                if (rs.next()) {
                    categoria.setId(rs.getInt(1));
                }
            }
            return true;
        }
    } catch (SQLException e) {
        System.err.println("Erro ao inserir categoria: " + e.getMessage());
    }
    return false;
}
```

Tabela de comparação entre os métodos na Categori.DAO:

Tabela 3 (comparação entre os métodos na Categoria.DAO)

Método	**Ação CRUD**	**Descrição Resumida**	**Parâmetros**	**Retorno**
Inserir	CREATE	Adiciona novo registro	Objeto Categoria	boolean (sucesso)
Atualizar	UPDATE	Modifica registro existente	Objeto Categoria	boolean (sucesso)

Deletar	DELETE	Remove registro (com verificação)	ID (int)	boolean (sucesso)
BuscarPorId	READ	Recupera um registro por ID	ID (int)	Categoria ou null
listarTodos	READ	Recupera todos os registros	—	List<Categoria>
GetMinhaLista	READ	Lista todos (formato alternativo)	—	ArrayList<Categoria>
maiorID	Auxiliar	Retorna o maior ID usado	—	int
ExistemProdutosAssociados	Auxiliar	Verifica produtos vinculados	ID (int)	boolean

3.1.2.2 Produto.DAO

No Produto.DAO que se destaca por implementar uma regra de negócio crítica (controle de estoque) com validação de integridade e operação atômica, atendendo aos requisitos funcionais RF3 e regras de negócio RN2/RN4. É o atualizarQuantidade().

Imagem 16 (método que muda o estado da quantidade na tabela produto)

```java
// Atualiza a quantidade em estoque de um produto
public boolean atualizarQuantidade(int id, int quantidadeVariacao) {
    if (quantidadeVariacao < 0) {
        String sqlVerificaEstoque = "SELECT quantidade FROM produto WHERE id = ?";
        try (Connection conexao = ConnectionFactory.getConnection();
                PreparedStatement stmtVerifica = conexao.prepareStatement(sqlVerificaEstoque)) {
            stmtVerifica.setInt(1, id);
            try (ResultSet rs = stmtVerifica.executeQuery()) {
                if (rs.next()) {
                    int estoqueAtual = rs.getInt("quantidade");
                    if (estoqueAtual + quantidadeVariacao < 0) {
                        throw new IllegalArgumentException("Quantidade insuficiente em estoque");
                    }}}
        } catch (SQLException e) {
            System.err.println("Erro ao verificar estoque: " + e.getMessage());
            return false;
        }}
    String sqlAtualiza = "UPDATE produto SET quantidade = quantidade + ? WHERE id = ?";
    try (Connection conexao = ConnectionFactory.getConnection();
            PreparedStatement stmtAtualiza = conexao.prepareStatement(sqlAtualiza)) {
        stmtAtualiza.setInt(1, quantidadeVariacao);
        stmtAtualiza.setInt(2, id);
        return stmtAtualiza.executeUpdate() > 0;
    } catch (SQLException e) {
        System.err.println("Erro ao atualizar quantidade: " + e.getMessage());
    }
    return false;
}
```

Tabela de comparação entre os métodos na Categori.DAO:

Tabela 34(comparação entre os métodos na Produto.DAO)

Método	**Ação CRUD**	**Descrição Resumida**	**Parâmetros**	**Retorno**
inserir	CREATE	Adiciona um novo produto ao banco de dados	Objeto Produto	boolean (sucesso)

atualizar	UPDATE	Modifica um produto existente	Objeto Produto	boolean (sucesso)
deletar	DELETE	Remove um produto pelo ID	ID (int)	boolean (sucesso)
buscarPorId	READ	Recupera um produto específico pelo ID (com dados da categoria associada)	ID (int)	Produto ou null
listarTodos	READ	Recupera todos os produtos (com informações de categoria)	—	List<Produto>
getMinhaLista	READ	Lista todos os produtos (formato alternativo: ArrayList)	—	ArrayList<Produto>
maiorID	Auxiliar	Retorna o maior ID de produto cadastrado	—	int
atualizarQuantidade	UPDATE	Ajusta a quantidade em estoque (com verificação para valores negativos)	ID (int), quantidadeVariacao (int)	boolean (sucesso)
listarAbaixoDoMinimo	READ	Lista produtos com quantidade abaixo do mínimo definido	—	List<Produto>
listarAcimaDoMaximo	READ	Lista produtos com quantidade acima do máximo definido	—	List<Produto>
gerarListaPrecos	READ	Gera relatório de preços (nome, preço, unidade e categoria)	—	List<Relatorio>
gerarBalanco	READ	Calcula valor total em estoque (quantidade × preço)	—	List<Relatorio>
gerarProdutosPorCategoria	READ	Conta produtos por categoria	—	List<Relatorio>

reajustarPrecos	UPDATE	Aplica reajuste percentual em todos os preços		

3.1.2.3 ConnectionFactory

Configurações para o banco de dados de forma global no sistema. Passando somente uma vez as credenciais:

Imagem 16 (Configuração do banco de dados ConnectionFactory)

```
private static final String DRIVER = "com.mysql.cj.jdbc.Driver";
private static final String URL = "jdbc:mysql://localhost:3306/"
        + "db_estoque?useTimezone=true&serverTimezone=UTC";
private static final String USER = "root";
private static final String PASSWORD = "aaaa";
```

Método para gerar uma conceção com as credenciais passadas nos atributos:

Imagem 17 (Método para obter uma conexão)

```
// Método para obter uma conexão
public static Connection getConnection() {
    try {
        Class.forName(DRIVER);
        return DriverManager.getConnection(URL, USER, PASSWORD);
    } catch (ClassNotFoundException | SQLException e) {
        throw new RuntimeException("Erro ao conectar com o banco de"
                + " dados", e);
    }
}
```

E por fim, o método par disparar exceções em relação à conceção com o BD:

Imagem 18 (Método para teste de conexão)

```
// Método para testar a conexão
public static void testarConexao() {
    try (Connection conn = getConnection()) {
        System.out.println("Conexão com o banco de dados"
                + " estabelecida com sucesso!");
    } catch (SQLException e) {
        System.err.println("Erro ao testar conexão " + e.getMessage());
    }
}
```

4 REFERÊNCIAS

Livros PRESSMAN, Roger S. *Engenharia de software: uma abordagem profissional*. 8. ed. Porto Alegre: McGraw-Hill, 2019.

Documentos Institucionais

ORACLE. *JDBC basics*. Disponível em: <https://docs.oracle.com/javase/tutorial/jdbc/basics/>. Acesso em: 14 maio 2025.

MySQL. *MySQL 8.0 Reference Manual*. Disponível em: <https://dev.mysql.com/doc/refman/8.0/en/>. Acesso em: 14 maio 2025.

Artigos Online

GEEKSFORGEEKS. *Java Swing | Formulário de Registro de Usuário Simples*. Disponível em: <https://www.geeksforgeeks.org/java-swing-user-registration-form/>. Acesso em: 15 maio 2025.

GUJ. *ConnectionFactory JDBC - Boas ideias para melhorar a classe*. Disponível em: <https://www.guj.com.br/t/connectionfactory-jdbc-boas-ideias-para-melhorar-a-classe/294712>. Acesso em: 14 maio 2025.

Repositórios GitHub MERC, Tiago. *Java e JDBC: trabalhando com um banco de dados - ALURA*. GitHub, 2025. Disponível em: <https://github.com/TiagoMerc/Java-e-JDBC-trabalhando-com-um-banco-de-dados-ALURA>. Acesso em: 14 jun. 2025.

PEIXOTO, Maria. *JFrame Projects*. GitHub, 2025. Disponível em: <https://github.com/mariacpeixoto/JFrame-projects>. Acesso em: 18 maio 2025.

Outras Referências JAVAMEX. *InvokeLater em Java*. Disponível em: <https://www.javamex.com/tutorials/threads/invokelater.shtml>.

Vídeos CRYSWERTON SILVA. *Curso de Java Básico*. YouTube, 2025. Disponível em: <https://www.youtube.com/watch?v=r8Vhn_yWTMo>.

CARLOS HENRIQUE JAVA. *Introdução ao JDBC em Java*. YouTube, 2025. Disponível em: <https://www.youtube.com/watch?v=FES8Cwl3LFs>.