Luiz Felipe Neves dos Santos Siqueira

Programação orientada a objetos POO

Professor Carlos Veríssimo

Senac

TADS - Analise Desenvolvimento de Sistemas

AVALIAÇAO POO 1- GESTAO DE ESTOQUE-

Grupo 18

13/10/2023

CRONOGRAMA


INICIO DO PROJETO
CASO DE USO
DIAGRAMAS UML
DIAGRAMA DE CLASSES
MANIPULANDO OBJETOS
ENCAPSULAMENTO E HERANÇA
PROVA 1

Sumário

introduçao

Este projeto apresenta um sistema de gerenciamento de estoque desenvolvido com base nos princípios da Programação Orientada a Objetos (POO). A POO é uma abordagem de programação que utiliza a ideia de “objetos” para representar dados e métodos. Neste projeto, utilizamos essa abordagem para modelar um sistema de gerenciamento de estoque.

O sistema de gerenciamento de estoque é uma parte crucial de qualquer negócio que lida com a venda de produtos físicos. Ele permite que as empresas acompanhem seus produtos, gerenciem o reabastecimento e evitem a falta ou o excesso de estoque. No entanto, gerenciar um estoque pode ser uma tarefa complexa e demorada sem o software adequado.

Portanto, desenvolvi este sistema para simplificar o processo. Utilizamos conceitos-chave da POO, como classes, objetos, encapsulamento e herança, para criar um sistema robusto e flexível. Com este sistema, as empresas podem facilmente adicionar ou remover itens, verificar a quantidade de itens disponíveis e muito mais.

Espero que este projeto não apenas demonstre nossa compreensão dos conceitos da POO, mas também forneça uma solução útil para empresas que buscam melhorar seu gerenciamento de estoque. Agradeço a oportunidade de aplicar o que aprendemos em nossa matéria de Programação Orientada a Objetos.

OBJETIVOS

O objetivo deste projeto é desenvolver um sistema de gerenciamento de estoque eficiente e fácil de usar, utilizando os princípios da Programação Orientada a Objetos. Pretendemos criar um sistema que permita às empresas gerenciar seu estoque de forma eficaz, proporcionando uma visão clara dos produtos disponíveis, necessidades de reabastecimento e movimentação de produtos.

Além disso, buscamos demonstrar a aplicação prática dos conceitos de POO, como encapsulamento, herança e polimorfismo, na resolução de problemas do mundo real. Através deste projeto, pretendemos não apenas aprimorar nossas habilidades em POO, mas também fornecer uma solução que possa ser utilizada por empresas para melhorar suas operações de gerenciamento de estoque.

Em última análise, o objetivo é criar um software que seja robusto, flexível e útil para as necessidades de gerenciamento de estoque das empresas. Acredito que este projeto será uma valiosa contribuição para nossa aprendizagem em Programação Orientada a Objetos e para o campo da gestão de estoque.

TECNOLOGIAS UTILIZADAS

JAVA

Java é a linguagem de programação orientada a objetos, desenvolvida pela empresa Sun Microsystems, capaz de criar tanto aplicativos para Desktop como para Web, aplicações comerciais, softwares robustos, completos e independentes. A linguagem Java foi desenvolvida na primeira metade da década de 90 nos laboratórios da Sun Microsystems com o objetivo de ser simples e eficiente. Esta é uma linguagem multiplataforma ela teve seu auge em 1995, devido ao sucesso mundial da World Wide Web. Nessa época a tecnologia Java teve uma enorme utilização e logo grandes empresas como a IBM, anunciaram que estariam dando suporte ao Java e a partir de então os seus aplicativos iriam rodar em Java. Em 2003 o Java já tinha mais de 4 milhões de desenvolvedores. Em 2006 a linguagem Java estava sendo disponível gratuitamente para o público por Software Livre. A linguagem revolucionou a área de desenvolvimento e sua utilização aumenta a cada dia. A escolha desta linguagem, visa ser uma das linguagens na atualidade com uma ampla utilização no mercado de Tecnologia de Informação (TI), e de fácil compreende mento no desenvolvimento dos códigos.

MYSQL

O MySQL foi desenvolvido por uma empresa de consultoria na Suécia chamada inicialmente de TcX, depois, com a popularidade do MySQL, passou a se chamar MySQL AB. Seu desenvolvimento ocorreu quando estavam precisando de um sistema de banco de dados que fosse extremamente rápido e flexível. Foi, assim então, que eles criaram o MySQL. (Gonçalves, 2006) E é um sistema de relacionamento de bancos de dados SGBD, que utiliza a linguagem SQL (Linguagem de Consulta Estruturada, do inglês Structured Query Language) como interface. E atualmente um dos bancos de dados mais populares, devido a sua qualidade e disponibilização Livre sem custos, facilitando sua acessibilidade para muitas pessoas. (Oficina Da Net, 2014) No Projeto ele será utilizado para a criação do banco de dados do sistema.

Diagramas da UML

Para fazer a modelagem do sistema, se utilizara a metodologia de Análise Orientada a Objeto, UML (Unified Modeling Language), é uma tentativa de padronizar a modelagem Orientada a Objetos, de forma que qualquer sistema possa ser modelado corretamente, a UML é constituída por elementos gráficos, utilizados na modelagem que permitem representar os conceitos do paradigma da Orientação a Objetos, através destes elementos gráficos podemos construir vários diagramas. Um diagrama é a representação gráfica de um conjunto de elementos, geralmente representados como gráficos de vértices (itens) e arcos (relacionamentos). São desenhados para permitir a visualização de um sistema sob diferentes perspectivas; nesse sentido, um diagrama constitui uma projeção de um determinado sistema. Em todos os sistemas, com exceção dos mais triviais, um diagrama representa uma visão parcial dos elementos que compõem o sistema. Abaixo estão listados alguns diagramas da UML.

Diagrama de caso de uso

Diagrama de sequência

Diagrama de classes

Mapa mental.

Diagrama de Caso de Uso

Um diagrama de caso de uso exibe um conjunto de caso de uso e atores Os atores representam os papéis desempenhados pelos diversos usuários que poderão utilizar, os serviços e funções do sistema. Já o caso de uso referem-se aos serviços, tarefas ou funcionalidades que podem ser utilizados de alguma maneira pelos atores 20 que interagem com o sistema, sendo utilizado para expressar e documentar os comportamentos pretendidos para as funções deste atores e caso de uso. Diagramas de caso de uso abrangem a visão estática de casos de uso do sistema. Esses diagramas são importantes principalmente para a organização e a modelagem de comportamentos do sistema.

Diagrama de Sequência

Um diagrama de sequência é um diagrama de interação que dá ênfase à ordenação temporal de mensagens. Um diagrama de sequência mostra conjunto de objetos e as mensagens enviadas e recebidas por esses objetos. Tipicamente os objetos são instâncias nomeadas ou anônimas de classes, mas também podem representar instâncias de outros itens, como colaborações, componentes e nós. Use os diagramas de sequência para ilustrar a visão dinâmica de um sistema.

Diagrama de Classe

Exibe um conjunto de classes, interfaces e colaborações, bem como seus relacionamentos. Esses diagramas são encontrados com maior frequência em sistemas de modelagem orientados a objetos e abrangem uma visão estática da estrutura do sistema. Os diagramas de classes que incluem classes ativas direcionam a perspectiva do processo estático do sistema.

Mapa Mental

Um mapa mental é caracterizado como um diagrama hierarquizado de informações, nele é possível perceber facilmente as relações e os vínculos entre elas. O mapa facilita a interpretação das palavras, imagens, números e conceitos lógicos, de maneira clara, concisa e consistente.

1- **DESCRIÇAO DO DOMINIO DO PROBLEMA**

Esse sistema se refere ao processo de planejar, organizar, controlar e supervisionar todos os aspectos relacionados ao armazenamento, movimentação e disponibilidade de produtos em uma empresa.

Requisitos Funcionais:

RF1: Cadastro de Produtos

RF2: Controle de Estoque

RF3: Recebimento de Mercadorias

RF4: Gestão de Fornecedores

RF5: Previsão de Demanda

RF6: Gerar nota

RF7: Gestão de Pedidos

RF8: Acesso e Controle de Usuário

RF9: Segurança de Dados

RF10: Relatórios e Análises

RF11: Acesso e Controle de Usuário

RF12: Cadastrar, excluir, editar e visualizar clientes

RF13: Recuperar senha

Requisitos Não Funcionais

RNF1: Criptografia na senha dos usuários (a definir)

RNF2: Disponibilidade do servidor

RNF3: Eficiência na estrutura do software

RNF4: Desempenho

RNF5: Segurança

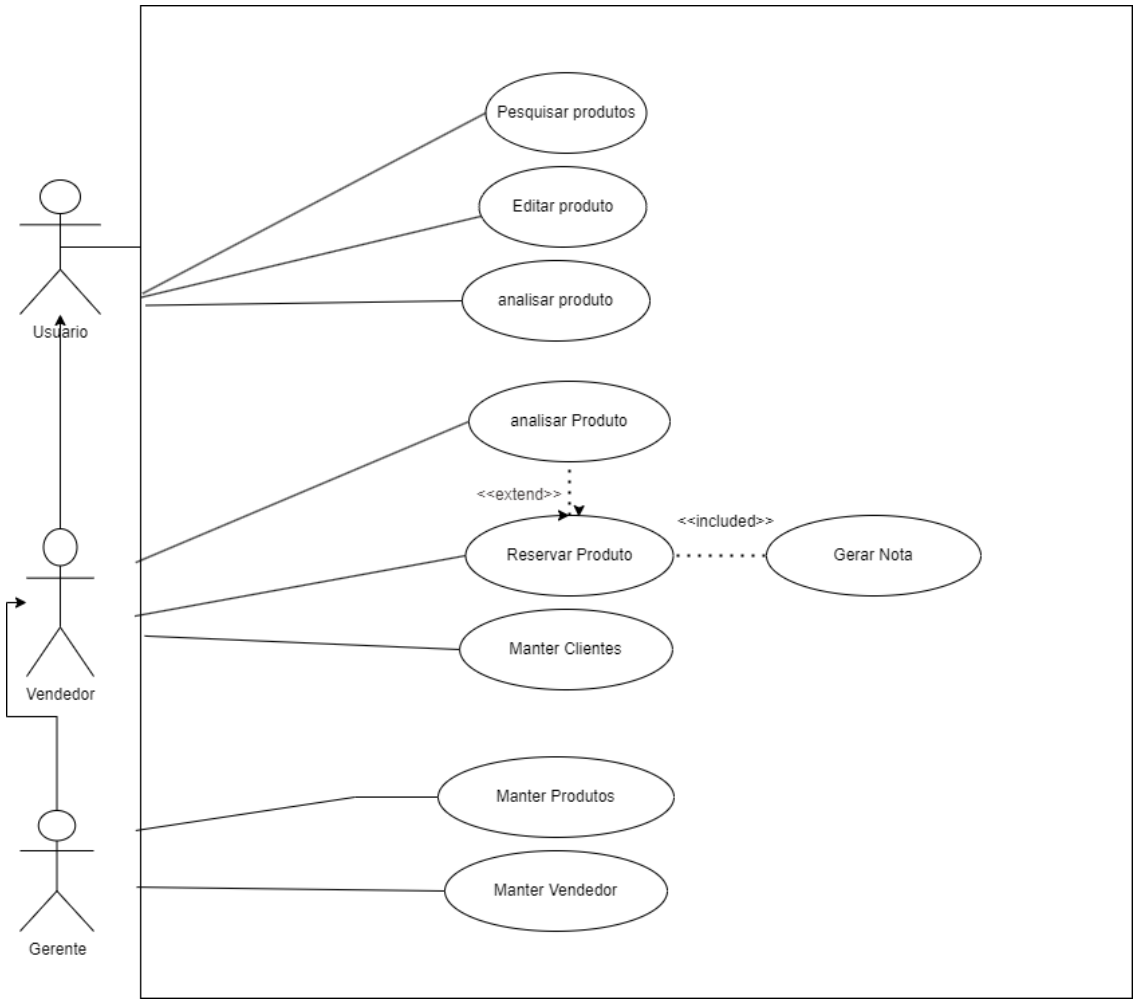
RNF6: Manutenção e Atualizações

RNF7: Custos Operacionais

RNF8: Compatibilidade

RNF9: Privacidade e ética

Diagrama de Caso de Uso - Visão Geral



Detalhamento do Caso de uso #1.1 – Pesquisar produtos

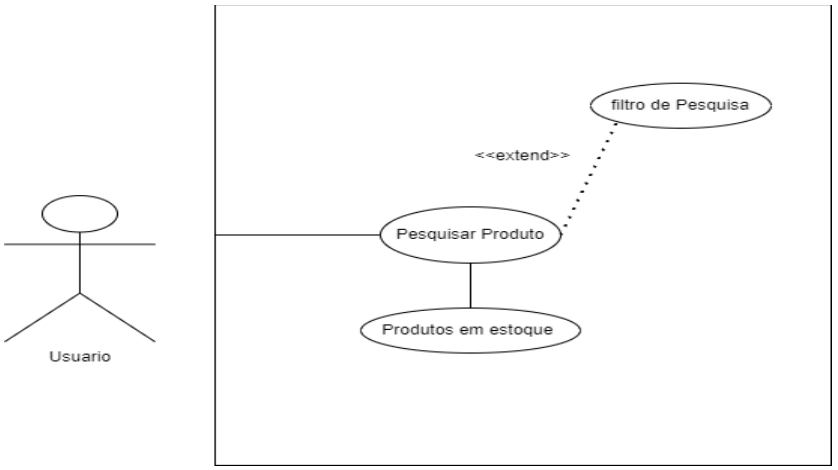


Diagrama do Caso de uso #1

Nome do caso de uso	Pesquisar produtos
atores	Usuários
trigger	Controle de produto
Pré-Requisito	Estar afiliado ao Sistema
Fluxo de eventos	Usuário efetua o login no sistema, logo após pesquisa o produto que tenha interesse e que esteja em estoque, por fim aperte a tecla entrar e aparecera os produtos que estejam disponíveis para tal.

Detalhamento do Caso de uso #1.2 –filtro de Pesquisa

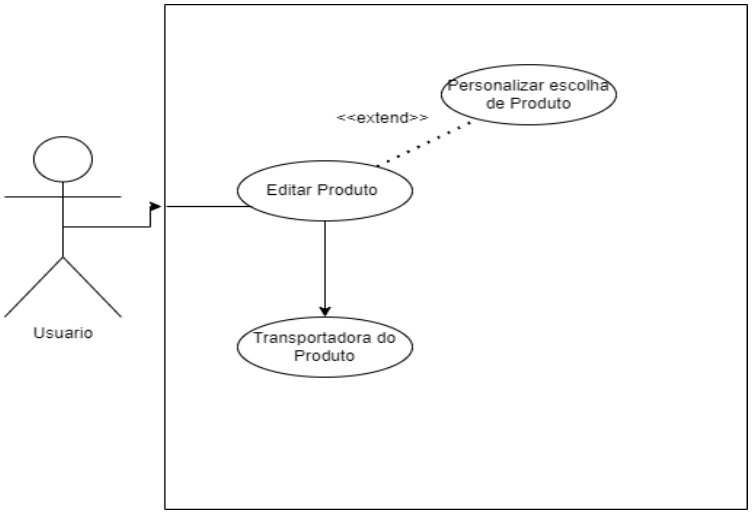
Nome do caso de uso	Filtro de Pesquisa
atores	Usuário
trigger	Controle de produto
Pré-requisito	Escolher um dos filtros
Fluxo de eventos	Loga no sistema, em seguida na própria página de pesquisa, escolhe o filtro.

Detalhamento do Caso de uso #1.3 – Produtos em Estoque

Nome do caso de uso	Produtos em estoque
atores	Usuário
trigger	Controle de produto
Pré-requisito	Estar logado no Sistema
Fluxo de eventos	Usuário loga no sistema, entra na barra de pesquisa, pesquisa o produto de interesse e observa a quantidade do produto em interesse no estoque.

3 Caso de Uso - Caso de Uso #2 Editar Produto em Estoque

A

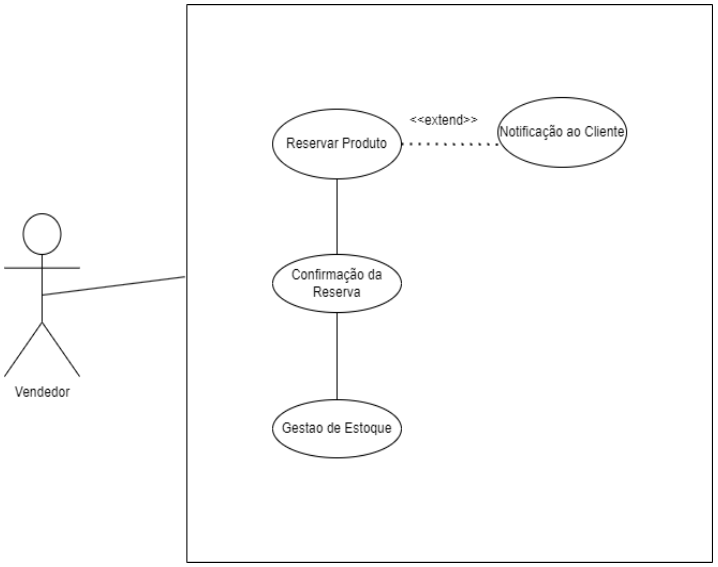


Nome do caso de uso	Editar Produtos em Estoque
Atores	Usuário
Trigger	Controle de produto
Pré-requisito	Estar logado no Sistema
Fluxo de eventos	Usuário entra no sistema, na mesma aba de escolha de produto, toque e escolha o tipo de produto, optando tanto como quantidade ou meio de transporte.

3 Caso de Uso - Caso de Uso #3 Transporte do Produto

Nome do caso de uso	Transporte do Produto
Atores	Usuário
Trigger	Controle de produto
Pré-requisito	Estar na Aba de finalização do Produto e envio
Fluxo de eventos	Estar na aba de métodos de envios e escolhas de meio de Transporte e distribuidora.

Caso de Uso - Caso de Uso 4 Reserva Produto



Nome do caso de uso	Reservar Produtos
Atores	Vendedor
Trigger	Atualização de estoque
Pré-requisito	Vender Produto
Fluxo de Eventos	Usuário escolhe o produto e o vendedor o auxilia.

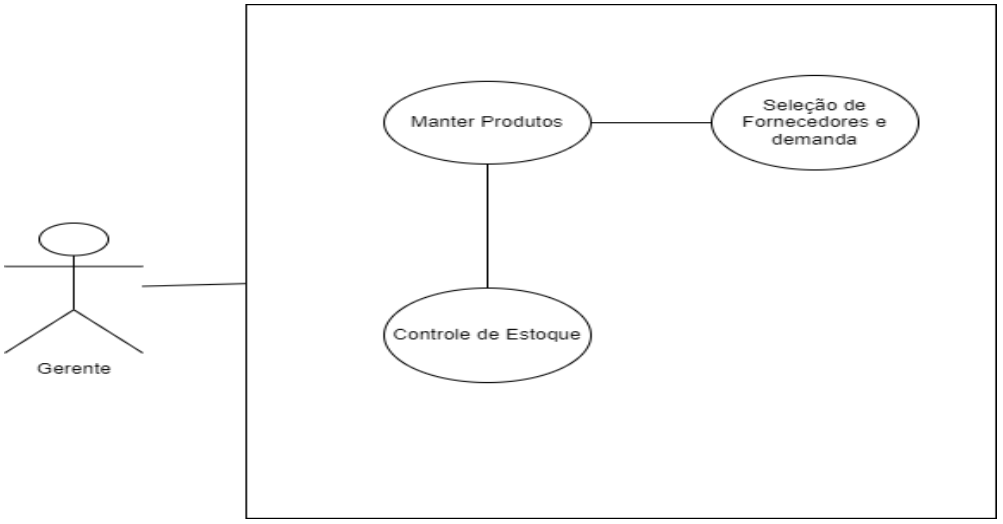
Caso de Uso - Caso de Uso 5 Confirmação de Reserva

Nome do caso de uso	Confirmação de Reserva
Atores	Vendedor
Trigger	Atualização de estoque
Pré-requisito	Notificar o êxito da compra para o usuário
Fluxo de Eventos	O sistema exibe uma confirmação da reserva para o vendedor, incluindo informações sobre o produto reservado, a quantidade e a data de retirada ou entrega prevista.

Caso de Uso - Caso de Uso 6 Gestão de Estoque

Nome do caso de uso	Gestão de estoque
Atores	Vendedor
Trigger	Atualização de Estoque
Pré-requisito	Monitoramento de Estoque
Fluxo de Eventos	Garantir que o produto reservado seja separado do estoque principal e mantido reservado até a conclusão da transação.

Caso de Uso - Caso de Uso 8 Manter Produtos



Nome do caso de uso	Manter Produto
Atores	Gerente
Trigger	Monitora Controle do estoque
Pré-requisito	Supervisionar o controle de estoque de produtos
Fluxo de Eventos	Definir metas de desempenho, como níveis de estoque ideais e redução de custos, e monitorar o progresso em direção a essas metas.

Caso de Uso - Caso de Uso 9 Seleção de fornecedores e demanda

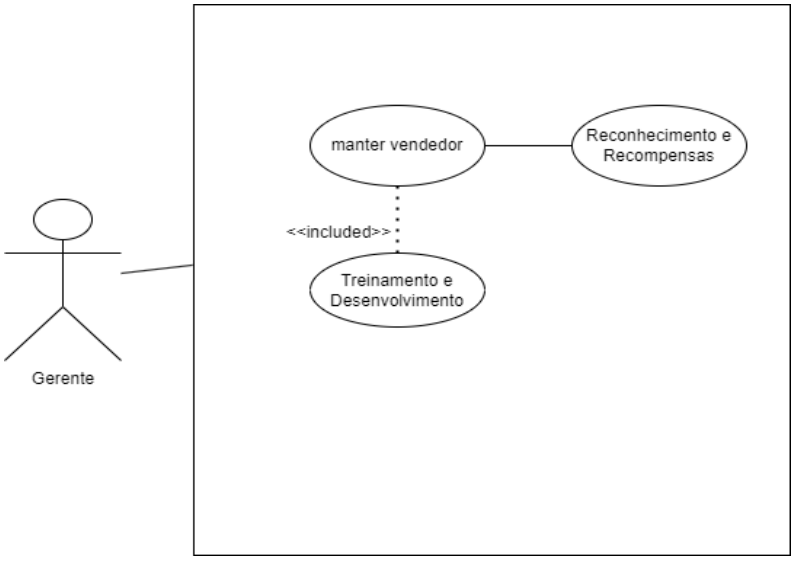
Nome do caso de uso	Seleção de fornecedores e demanda
Atores	Gerente
Trigger	Controle de estoque
Pré-requisito	Identificar e selecionar fornecedores confiáveis e fornecedores estratégicos que atendam às necessidades da empresa em termos de qualidade, prazos de entrega e custos.

Fluxo de Eventos	Desenvolver métodos de previsão de demanda para garantir que o estoque esteja alinhado com as necessidades dos clientes. Usar dados históricos e tendências de mercado para melhorar a precisão das previsões.

Caso de Uso - Caso de Uso 10 controle de estoque

Nome do caso de uso	Controle de Estoque
Atores	Gerente
Trigger	Monitoramento e controle do estoque
Pré-requisito	Desenvolver estratégias de gestão de estoque alinhadas com os objetivos gerais da empresa
Fluxo de Eventos	Supervisionar o controle de estoque, incluindo o registro e monitoramento das entradas e saídas de produtos. Garantir a precisão dos registros de estoque para evitar discrepâncias.

Caso de Uso - Caso de Uso Manter Vendedor



Caso de Uso - Caso de Uso Manter vendedor

Nome do caso de uso	Manter vendedor
Atores	Gerente
Trigger	Notificação ao Funcionário
Pré-requisito	Mostrar aos funcionários as oportunidades de crescimento dentro da empresa
Fluxo de Eventos	Incentivar o funcionário a buscar oportunidades de aprendizado contínuo e a se manter atualizado com as melhores práticas do setor.

Caso de Uso - Caso de Uso Reconhecimento e Recompensas

Nome do caso de uso	Reconhecimento e recompensas
Atores	Gerente
Trigger	Notificação ao Funcionário
Pré-requisito	Certificasse de que o funcionário tenha as ferramentas e recursos necessários para executar suas tarefas de maneira eficaz.
Fluxo de Eventos	Desde o início, estabeleça expectativas claras em relação às responsabilidades e metas do funcionário de gestão de estoque. Reconheça e recompense o bom desempenho do funcionário.

Caso de Uso - Caso de Uso treinamento e desenvolvimento

Nome do caso de uso	Treinamento e desenvolvimento
Atores	Gerente
Trigger	Notificação ao Funcionário
Pré-requisito	Investir em treinamento e desenvolvimento contínuos para o funcionário de gestão de estoque, garantindo que ele esteja atualizado com as melhores práticas e as últimas tecnologias.
Fluxo de Eventos	funcionário em decisões relacionadas à gestão de estoque sempre que possível. Isso pode incluir planejamento de

	compras, seleção de fornecedores e definição de políticas de estoque.

CHAPTER 2

DAREMOS CONTINUIDADE A GESTAO DE ESTOQUE ATRAVES DO DIAGRAMA UML DE TODAS AS CLASSES

Caso de Uso 1 – Manter Funcionário

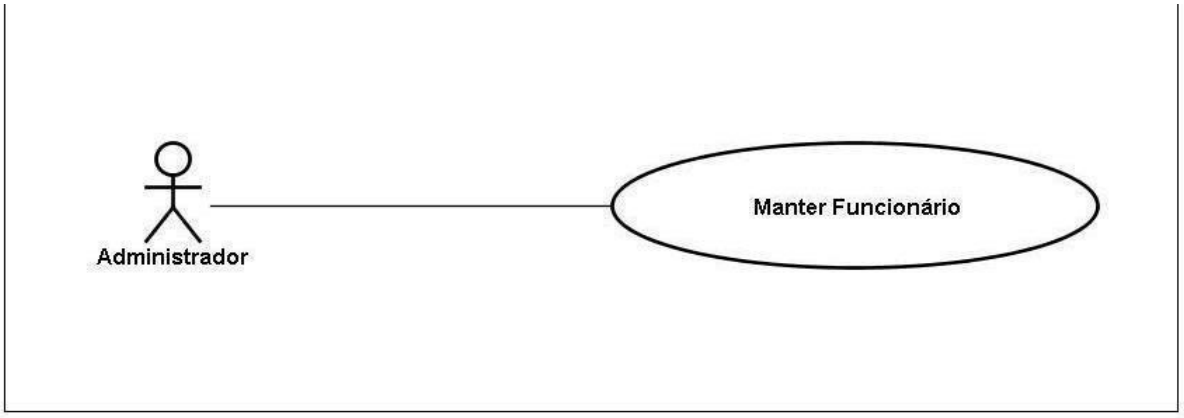


Diagrama de Caso de Uso Manter Funcionário

Funcionalidade/Objetivo	Inserir, alterar, excluir e pesquisar funcionário.
Ator	Administrador
Pré-Condição	O administrador deverá estar autenticado no sistema.

Cenário Principal	1 – O sistema solicita os dados necessários para o cadastro do funcionário. 2 – O administrador informa os dados de acordo com os campos a serem preenchidos. 3 – O sistema solicita os dados para o cadastro da função. 4 – O administrador informa os dados necessário. 5 – O administrador seleciona a opção “Cadastrar”. 6 – o sistema emite a mensagem “Funcionário Cadastrado com Sucesso”. 7 – O sistema cadastra o funcionário.
Cenário Alternativo	A1 - O administrador não informar os dados para o cadastro da função, o sistema informa que o funcionário não está cadastrado. A2 - O administrador poderá cancelar o processo durante o cadastro.
Casos de Teste	4.1- O sistema verifica se os campos foram preenchidos corretamente. 4.2- O sistema não confirma o cadastro e emite uma mensagem de erro. 4.3- O sistema cancela a operação.

Quadro 1 – Manter Funcionário

Caso de Uso 2 – Manter Cliente

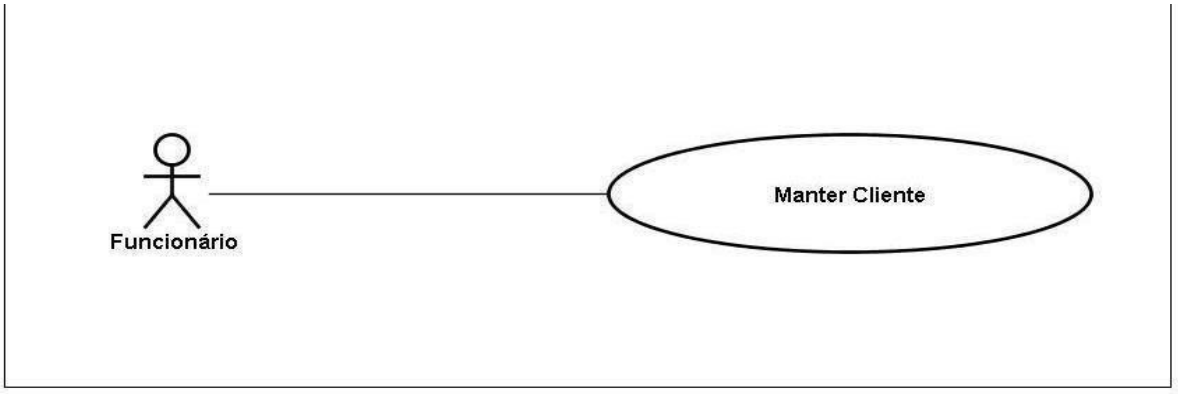
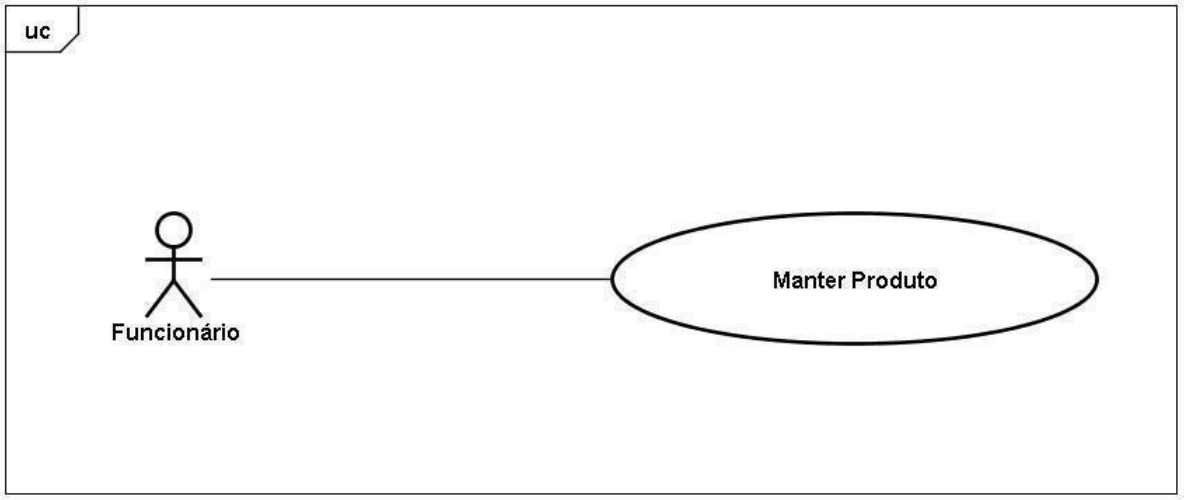


Diagrama de Caso de Uso Manter Cliente

Funcionalidade/Objetivo	Inserir, alterar, excluir e pesquisar cliente
Ator	Funcionário
Pré-Condição	O funcionário deverá estar autenticado no sistema.
Cenário Principal	1 – O sistema solicita os dados necessários para o cadastro do cliente. 2 – O funcionário informa os dados de acordo com os campos a serem preenchidos. 3 – O sistema solicita os dados para o cadastro da função. 4 – O funcionário informa os dados necessário. 5 – O Funcionário seleciona a opção “Cadastrar”. 6 – O sistema emite a mensagem “Cliente Cadastrado com Sucesso”. 7 – O sistema cadastra o cliente.
Cenário Alternativo	A1 – O funcionário não informar os dados para o cadastro da função, o sistema informa que o cliente não está cadastrado. A2 – O funcionário poderá cancelar o processo durante o cadastro.
Casos Teste	4.1 – O sistema verifica se os campos foram preenchidos corretamente. 4.2 – O sistema não confirma o cadastro e emitir uma mensagem de erro. 4.3 – O sistema cancela a operação.

Quadro 2 – Manter Cliente

Caso de Uso 3 Manter Produto



3 Diagrama de Caso de Uso Manter Produto

Funcionalidade/Objetivo	Inserir, alterar, excluir e pesquisar produtos
Ator	Funcionário
Pré-Condição	O funcionário deverá estar autenticado no sistema
Cenário Principal	1 – O sistema solicita os dados necessários para o cadastro do produto. 2 – O funcionário informa os dados de acordo com os campos a serem preenchidos. 3 – O sistema solicita os dados para o cadastro da função. 4 – O funcionário informa os dados necessário. [A2] 5 – O funcionário seleciona a opção “Cadastrar”. 6 – O sistema emite a mensagem “Produto Cadastrado com Sucesso”. 7 – O sistema cadastra o produto.
Cenário Alternativo	A1 – Se o funcionário não informar os dados para o cadastro da função, o sistema informa que o produto não está cadastrado. A2 – O funcionário poderá cancelar o processo durante o cadastro.
Casos Teste	4.1- O sistema verifica se os campos foram preenchidos corretamente. 4.2 – O sistema não confirma o cadastro e emite uma mensagem de erro. 4.3 – O sistema cancela a operação.

Quadro 3 – Manter Produto

4 Movimentar Vendas

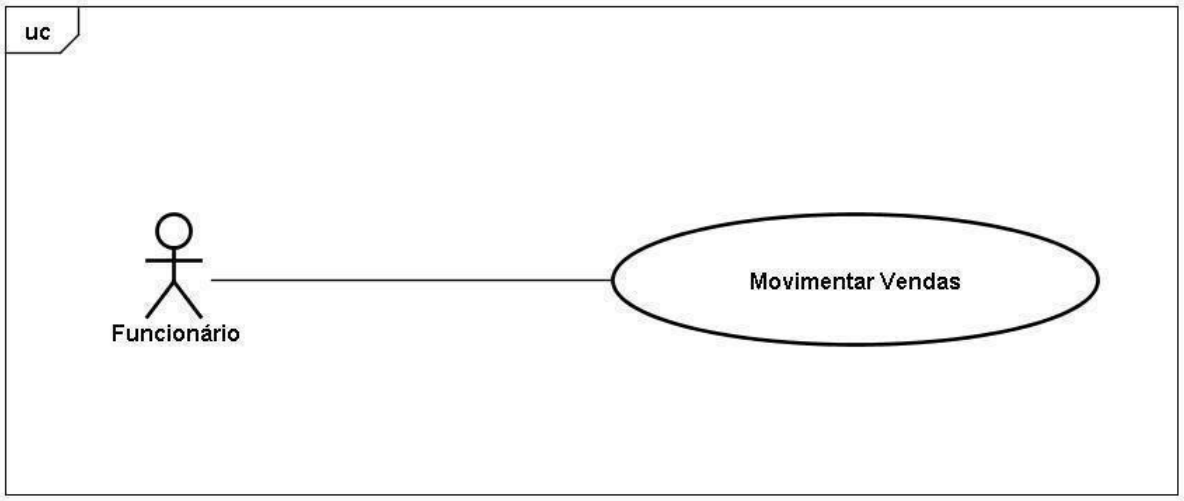
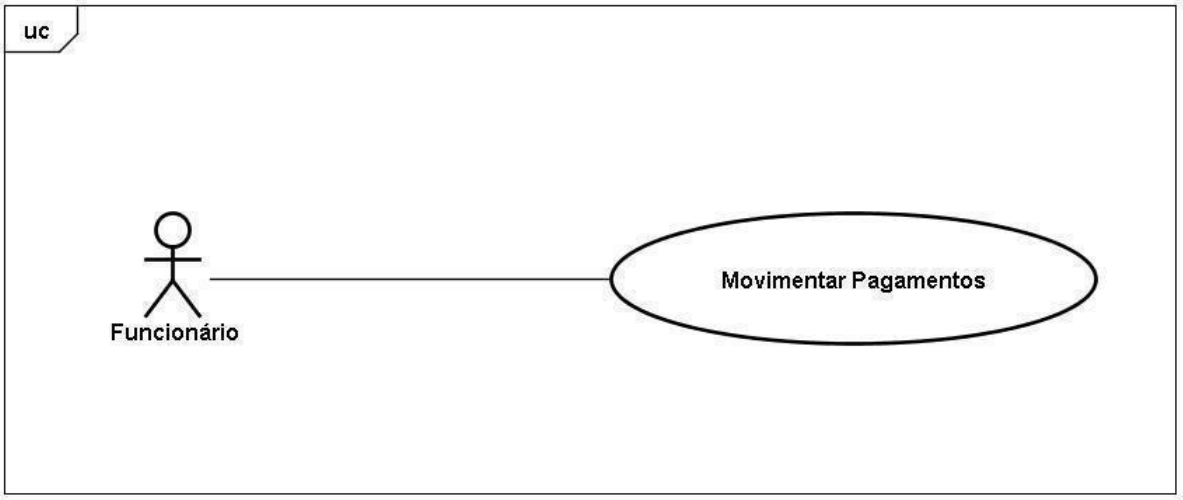


Diagrama de Caso de Uso Movimentar Vendas

Funcionalidade/Objetivo	Permite ao funcionário fornecer informações para a movimentação de vendas.
Ator	Funcionário
Pré-Condição	O funcionário deverá estar autenticado no sistema
Cenário Principal	1 – O sistema solicita os dados necessários para movimentar vendas. 2 – O funcionário informa os dados de acordo com os campos a serem preenchidos. 3 – O sistema solicita os dados para o cadastro da função. 4 – O funcionário informa os dados necessários. 5 – O funcionário seleciona a opção “Salvar”. 6 – O sistema emite a mensagem “Operação Realizada com Sucesso”.
Cenário Alternativo	A1 – O funcionário poderá cancelar o processo durante a movimentação.
Casos Teste	5.1 – O sistema verifica se os campos foram preenchidos corretamente. 5.2 – O sistema não confirma o cadastro e emite uma mensagem de erro. 5.3 – O sistema cancela a operação.

5 Caso de Uso 5 Movimentar Pagamentos



UC5 Diagrama de Caso de Uso Movimentar Pagamentos

Funcionalidade/Objetivo	Permite ao funcionário fornecer informações para a movimentação de pagamentos.
Ator	Funcionário
Pré-Condição	O funcionário deverá estar autenticado no sistema
Cenário Principal	1 – O sistema solicita os dados necessários para movimentar pagamento. 2 – O funcionário informa os dados de acordo com os campos a serem preenchidos. 3 – O sistema solicita os dados para o cadastro da função. 4 – O funcionário informa os dados necessários. 5 – O funcionário seleciona a opção “Salvar”. 6 – O sistema emite a mensagem “Operação Realizada com Sucesso”.
Cenário Alternativo	A1 – O funcionário poderá cancelar o processo durante a movimentação.
Casos Teste	5.1 – O sistema verifica se os campos foram preenchidos corretamente. 5.2 – O sistema não confirma o cadastro e emite uma mensagem de erro. 5.3 – O sistema cancela a operação.

Quadro 5 – Movimentar Pagamentos

6 Caso de Uso 6 Movimentar Recebimentos

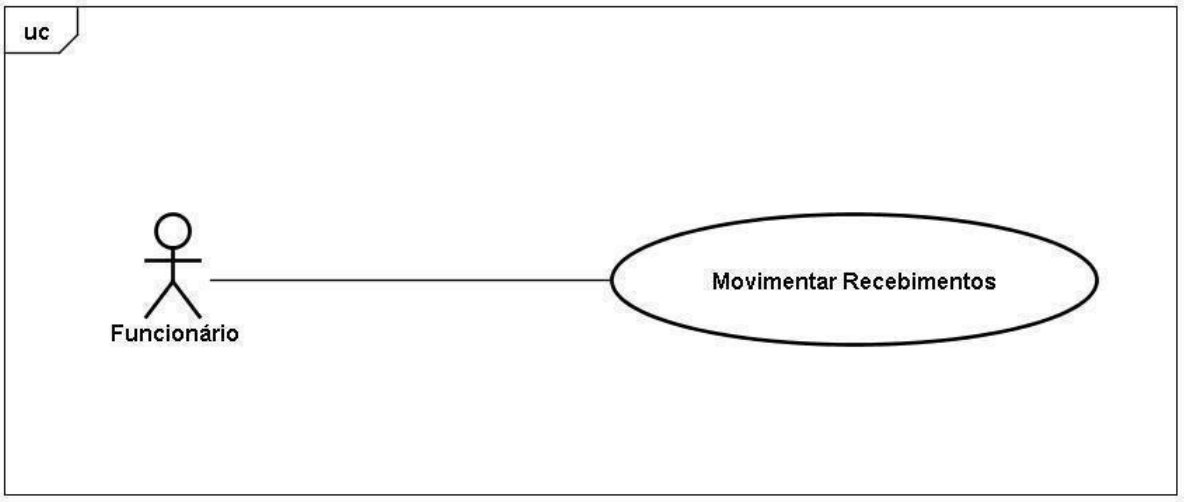
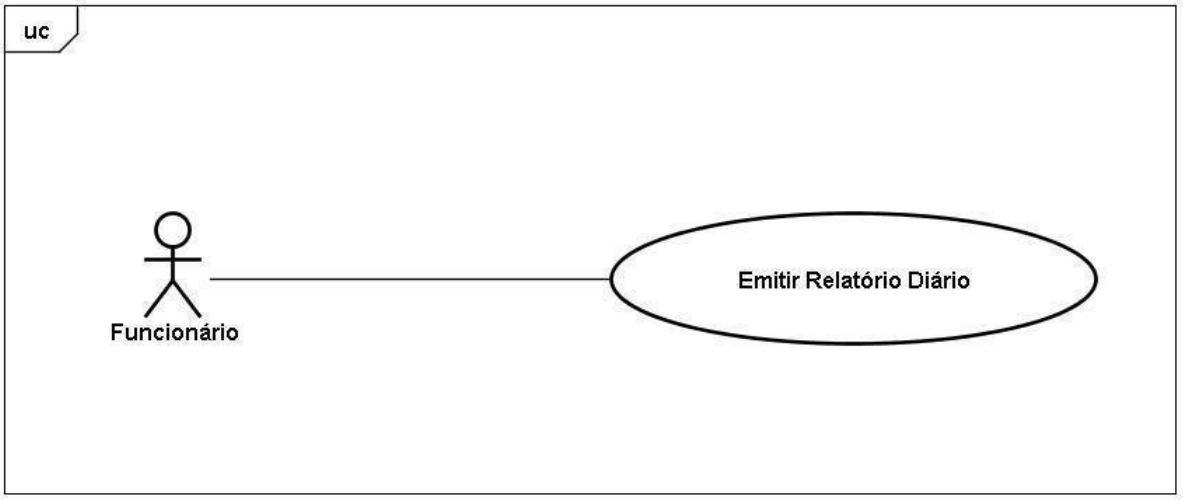


Diagrama de Caso de Uso Movimentar Recebimentos

Funcionalidade/Objetivo	Permite ao funcionário fornecer informações para a movimentação de recebimentos.
Ator	Funcionário
Pré-Condição	O funcionário deverá estar autenticado no sistema
Cenário Principal	1 – O sistema solicita os dados necessários para movimentar recebimento. 2 – O funcionário informa os dados de acordo com os campos a serem preenchidos. 3 – O sistema solicita os dados para o cadastro da função. 4 – O funcionário informa os dados necessários. 5 – O funcionário seleciona a opção “Salvar”. 6 – O sistema emite a mensagem “Operação Realizada com Sucesso”.
Cenário Alternativo	A1 – O funcionário poderá cancelar o processo durante a movimentação.

Casos Teste	5.1 – O sistema verifica se os campos foram preenchidos corretamente. 5.2 – O sistema não confirma o cadastro e emite uma mensagem de erro. 5.3 – O sistema cancela a operação.

7 Caso de Uso 7 Emitir Relatório Diário



Funcionalidade/Objetivo	Permite ao funcionário emitir relatório diário
Ator	Funcionário
Pré-Condição	O funcionário deverá estar autenticado no sistema
Cenário Principal	1 – O sistema disponibiliza os dados necessários para o relatório. 2 – O funcionário seleciona o botão “Relatório”. [A1] 3 – O funcionário seleciona o botão “Imprimir”. 4 – O sistema imprime o relatório com sucesso.
Cenário Alternativo	A1 – O funcionário poderá visualizar o relatório e não imprimir.
Casos Teste	4.1 – O funcionário cancela a operação.

Quadro 7 – Emitir Relatório Diário

Caso de Uso 8 Emitir Relatório Mensal

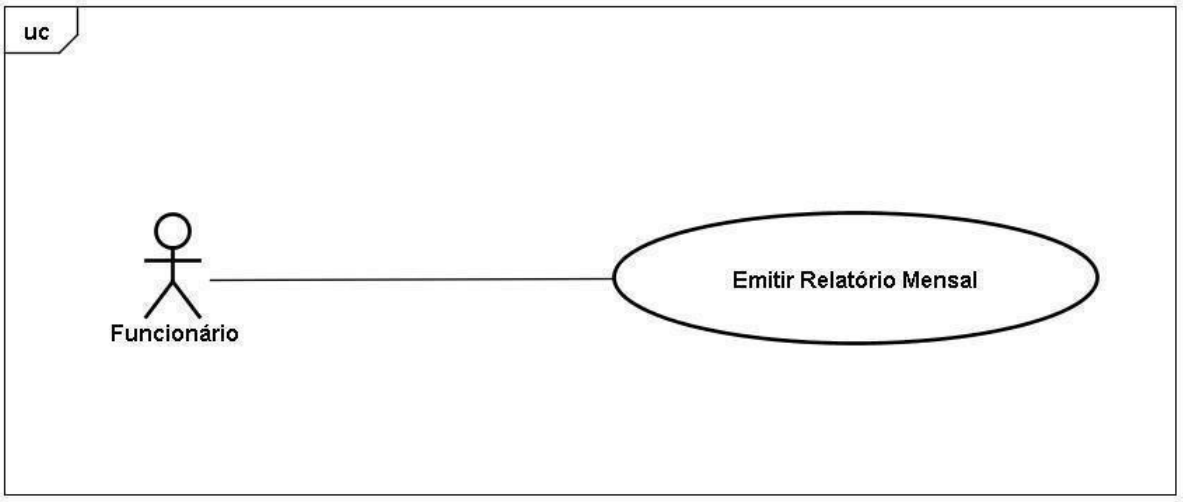


Figura 9 – UC8 Diagrama de Caso de Uso Emitir Relatório Mensal

Funcionalidade/Objetivo	Permite ao funcionário emitir relatório mensal
Ator	Funcionário
Pré-Condição	O funcionário deverá estar autenticado no sistema
Cenário Principal	1 – O sistema disponibiliza os dados necessários para o relatório. 2 – O funcionário seleciona o botão “Relatório”. [A1] 3 – O funcionário seleciona o botão “Imprimir”. 4 – O sistema imprime o relatório com sucesso.
Cenário Alternativo	A1 – O funcionário poderá visualizar o relatório e não imprimir.
Casos Teste	4.1 – O funcionário cancela a operação.

Quadro 8 – Emitir Relatório Mensal

Diagrama elaborado foi usado apenas um ator, o administrador, uma vez que o sistema será utilizado apenas pelo Adm, possuindo apenas o responsável por gerenciá-lo.

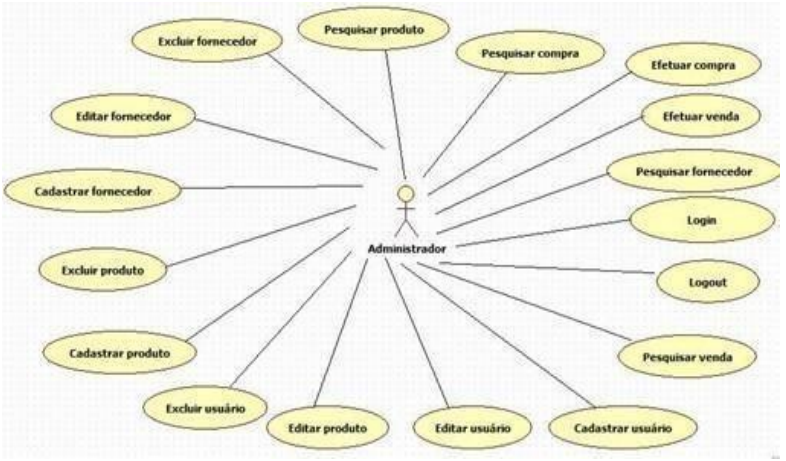
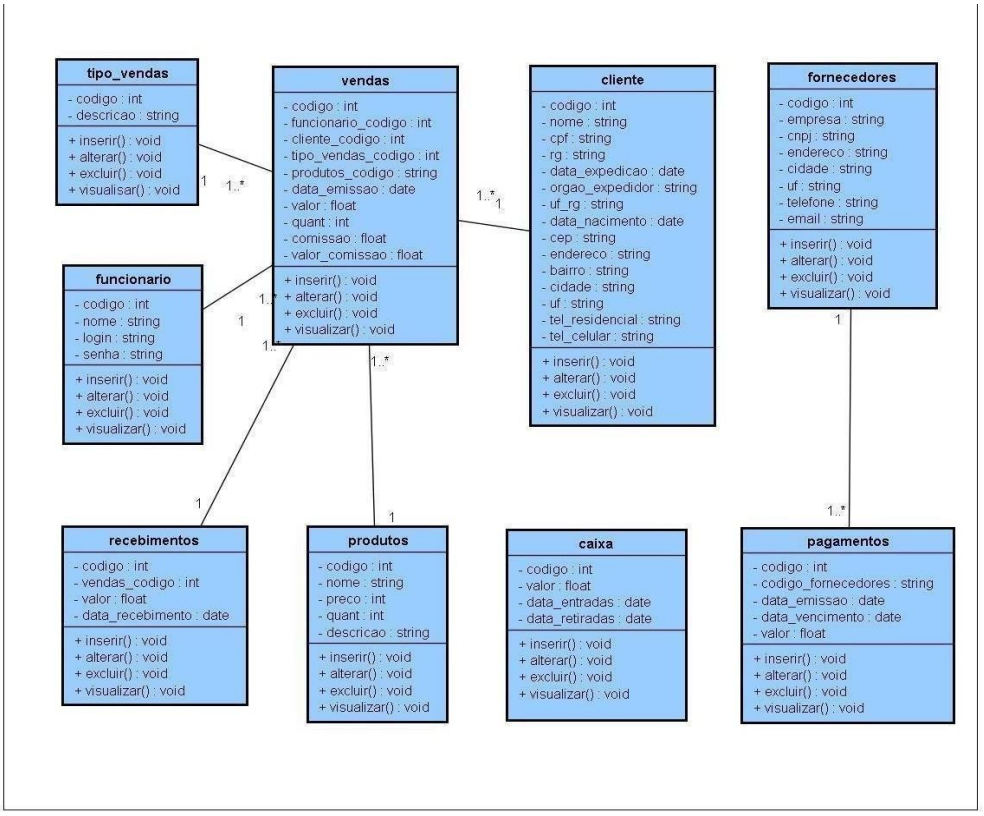


Diagrama de classe Mysql



(CODIGO INICIAL) JAVA ATRAVES DO DIAGRAMA DE CLASSE

```
import java.sql.*;

public class GestaoEstoque {
    private Connection conexao;

    public GestaoEstoque(String url, String usuario, String senha) throws
SQLException {
        conexao = DriverManager.getConnection(url, usuario, senha);
    }

    public void adicionarProduto(String nome, int quantidade) throws SQLException
{
        String sql = "INSERT INTO estoque (nome, quantidade) VALUES (?, ?)";
        PreparedStatement stmt = conexao.prepareStatement(sql);
        stmt.setString(1, nome);
        stmt.setInt(2, quantidade);
        stmt.executeUpdate();
    }

    public void atualizarQuantidade(String nome, int quantidade) throws
SQLException {
        String sql = "UPDATE estoque SET quantidade = ? WHERE nome = ?";
        PreparedStatement stmt = conexao.prepareStatement(sql);
        stmt.setInt(1, quantidade);
        stmt.setString(2, nome);
        stmt.executeUpdate();
    }

    public int obterQuantidade(String nome) throws SQLException {
        String sql = "SELECT quantidade FROM estoque WHERE nome = ?";
        PreparedStatement stmt = conexao.prepareStatement(sql);
        stmt.setString(1, nome);
        ResultSet rs = stmt.executeQuery();

        if (rs.next()) {
            return rs.getInt("quantidade");
        } else {
            return 0;
        }
    }
```

```
    public static void main(String[] args) {
        try {
            GestaoEstoque gestaoEstoque = new
GestaoEstoque("url_do_banco_de_dados", "usuario", "senha");
            gestaoEstoque.adicionarProduto("produto_teste", 100);
            gestaoEstoque.atualizarQuantidade("produto_teste", 200);
            int quantidade = gestaoEstoque.obterQuantidade("produto_teste");
            System.out.println("Quantidade do produto_teste: " + quantidade);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Tabela para armazenar informações doproduto

```
CREATE TABLE produtos (
id INT AUTO_INCREMENT PRIMARY KEY, nome VARCHAR(255) NOT NULL,
descricao TEXT,
preco DECIMAL(10, 2) NOT NULL,
quantidade INT NOT NULL
);

-- Tabela para armazenar informações de vendas CREATE TABLE vendas (
id INT AUTO_INCREMENT PRIMARY KEY,
data_venda DATE NOT NULL, cliente_id INT,
funcionario_id INT, caixa_id INT,
FOREIGN KEY (cliente_id) REFERENCES clientes(id), FOREIGN KEY (funcionario_id)
REFERENCES funcionarios(id), FOREIGN KEY (caixa_id) REFERENCES caixa(id)
);

-- Tabela para armazenar itens de venda CREATE TABLE itens_venda (
id INT AUTO_INCREMENT PRIMARY KEY,
venda_id INT, produto_id INT,
quantidade INT NOT  NULL,
preco_unitario DECIMAL(10, 2) NOT NULL,

14


FOREIGN KEY (venda_id) REFERENCES vendas(id), FOREIGN KEY (produto_id) REFERENCES
produtos(id)
);

-- Tabela para armazenar informações de clientes CREATE TABLE clientes (
id INT AUTO_INCREMENT PRIMARY KEY, nome VARCHAR(255) NOT NULL,
endereco VARCHAR(255), telefone VARCHAR(20)
);

-- Tabela para armazenar informações de funcionários CREATE TABLE funcionarios (
```

```
id INT AUTO_INCREMENT PRIMARY KEY, nome VARCHAR(255) NOT NULL,
cargo VARCHAR(255),
salario DECIMAL(10, 2)
);

-- Tabela para armazenar informações de caixa CREATE TABLE caixa (
id INT AUTO_INCREMENT PRIMARY KEY, saldo DECIMAL(10, 2) NOT NULL
);
```

Neste momento estamos aplicando o encapsulamento em uma gestao de estoque

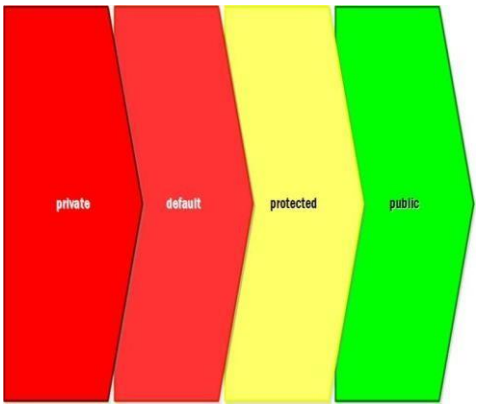
```java
public class Estoque {
    // Variáveis de instância privadas para encapsulamento
    private String produto;
    private int quantidade;

    // Construtor
    public Estoque(String produto, int quantidade) {
        this.produto = produto;
        this.quantidade = quantidade;
    }

    // Getters (Acessadores)
    public String getProduto() {
        return this.produto;
    }

    public int getQuantidade() {
        return this.quantidade;
    }

    // Setters (Modificadores)
    public void setProduto(String novoProduto) {
        this.produto = novoProduto;
    }

    public void setQuantidade(int novaQuantidade) {
        this.quantidade = novaQuantidade;
    }

    // Método para adicionar ao estoque
    public void adicionarEstoque(int quantidade) {
        if (quantidade > 0) {
            this.quantidade += quantidade;
        }
    }

    // Método para remover do estoque
    public void removerEstoque(int quantidade) {
        if (quantidade > 0 && this.quantidade >= quantidade) {
            this.quantidade -= quantidade;
        }
    }

    // Método protegido
    protected void metodoProtegido() {
        // Código aqui
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        // Criando um novo objeto Estoque
        Estoque estoque = new Estoque("Produto1", 100);

        // Exibindo o produto e a quantidade inicial
        System.out.println("Produto: " + estoque.getProduto());
        System.out.println("Quantidade inicial: " + estoque.getQuantidade());

        // Adicionando ao estoque
        estoque.adicionarEstoque(50);
        System.out.println("Quantidade após adicionar: " +
estoque.getQuantidade());

        // Removendo do estoque
        estoque.removerEstoque(30);
        System.out.println("Quantidade após remover: " +
estoque.getQuantidade());
    }
}
```

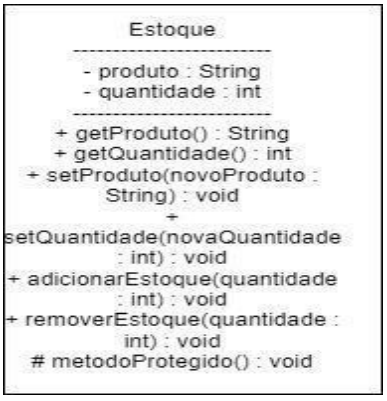
Neste exemplo, os conceitos de Programação Orientada a Objetos (POO) são aplicados da seguinte maneira:

- **Encapsulamento:** As variáveis produto e quantidade são privadas e só podem ser acessadas através dos métodos getters e setters.
- **Getters e Setters**: Os métodos getProduto, getQuantidade, setProduto e setQuantidade permitem que você acesse e modifique as variáveis de instância de maneira controlada.
- **Modificadores de acesso**: private é usado para restringir o acesso direto às variáveis de instância.
- **Objeto this**: É usado para se referir à instância atual da classe.
- **Acoplamento (Coupling):** Este código tem baixo acoplamento porque a classe Estoque não depende de outras classes.
- **Encapsulamento**: Os atributos das classes Produto e Estoque são privados. Eles só podem ser acessados ou modificados através dos métodos públicos dessas classes. Isso é encapsulamento.
- **Baixo acoplamento**: A classe Estoque depende da classe Produto, mas a classe Produto não depende de nenhuma outra classe. Além disso, a maneira como implementamos a classe Estoque permite que ela funcione com qualquer objeto que seja do tipo Produto, sem precisar saber detalhes específicos sobre a classe Produto. Isso é baixo acoplamento.

UML dos modificadoíes de acesso:



Public: É o modificador de acesso mais permissivo que existe. Atributos, métodos e classes declarados como public são acessíveis por qualquer classe. Em UML, é definido pelo símbolo.
Private: É o modificador de acesso mais restritivo. Atributos e métodos declarados como private são acessíveis somente pela classe que os declara. Em UML, é definido pelo símbolo.
Protected: Atributos e métodos declarados como protected são acessíveis pela classe que os declara, suas subclasses em outros pacotes e outras classes dentro do mesmo pacote. Em UML, é definido pelo símbolo .
Default: Modificador de acesso padrão, usado quando nenhum for definido. Neste caso os atributos, métodos e classes são visíveis por todas as classes dentro do mesmo pacote. Em UML, é definido pelo símbolo “~” (package visibility).



Utilizando HERANÇA

A herança é um dos pilares da Programação Orientada a Objetos (POO) que permite a reutilização de software. Na gestão de estoque, a herança pode ser usada para criar uma estrutura de classes que reflete a natureza hierárquica dos diferentes tipos de produtos ou entidades envolvidas. Por exemplo, você pode ter uma classe base chamada Produto que contém atributos e métodos comuns a todos os produtos, como nome, preço e quantidadeEmEstoque. A partir dessa classe base, você pode criar subclasses para diferentes tipos de produtos, como ProdutoPerecível, ProdutoEletrônico, etc. Cada uma dessas subclasses herda os atributos e métodos da classe base e pode adicionar seus próprios atributos e métodos específicos.

Da mesma forma, você pode ter classes para diferentes tipos de entidades envolvidas na gestão de estoque, como Cliente, Fornecedor e Funcionário, todas herdando de uma classe base Pessoa.

A principal vantagem da herança é que ela permite a reutilização de código, reduzindo a redundância e tornando o código mais fácil de manter. Além disso, a herança assegura que programas orientados a objetos cresçam de forma linear e não geometricamente em complexidade. Cada nova classe derivada não possui interações imprevisíveis em relação ao restante do código do sistema.

```
      import java.util.ArrayList;
import java.util.List;

class Pessoa {
    private String nome;

    public Pessoa(String nome) {
        this.nome = nome;
    }

    // Getter e Setter
}

class Cliente extends Pessoa {
    public Cliente(String nome) {
        super(nome);
    }
}

class Fornecedor extends Pessoa {
    public Fornecedor(String nome) {
        super(nome);
    }
}

class Funcionario extends Pessoa {
    public Funcionario(String nome) {
        super(nome);
    }
}

class Estoque {
    private String produto;
```

```
    private int quantidade;
    private double preco;

    public Estoque(String produto, int quantidade, double preco) {
        this.produto = produto;
        this.quantidade = quantidade;
        this.preco = preco;
    }

    // Getters e Setters
    public String getProduto() {
        return produto;
    }

    public int getQuantidade() {
        return quantidade;
    }

    public double getPreco() {
        return preco;
    }

    public void adicionarEstoque(int quantidade) {
        this.quantidade += quantidade;
    }

    public void removerEstoque(int quantidade) {
        if (this.quantidade >= quantidade) {
            this.quantidade -= quantidade;
        } else {
            System.out.println("Quantidade insuficiente no estoque.");
        }
    }
}

public class EstoqueAvancado extends Estoque {
    private String tipoVendas;
    private int vendas;
    private List<Cliente> clientes;
    private List<Fornecedor> fornecedores;
    private List<Funcionario> funcionarios;
    private double recebimentos;
    private double caixa;
    private double pagamentos;

    public EstoqueAvancado(String produto, int quantidade, double preco, String
tipoVendas, int vendas, double recebimentos, double caixa, double pagamentos) {
        super(produto, quantidade, preco);
        this.tipoVendas = tipoVendas;
        this.vendas = vendas;
        this.clientes = new ArrayList<>();
        this.fornecedores = new ArrayList<>();
        this.funcionarios = new ArrayList<>();
        this.recebimentos = recebimentos;
```

```
        this.caixa = caixa;
        this.pagamentos = pagamentos;
    }

    // Getters e Setters para os novos atributos

    // Métodos para adicionar e remover clientes, fornecedores e funcionários
    public void adicionarCliente(Cliente cliente) {
        this.clientes.add(cliente);
    }

    public void removerCliente(Cliente cliente) {
        this.clientes.remove(cliente);
    }

    public void adicionarFornecedor(Fornecedor fornecedor) {
        this.fornecedores.add(fornecedor);
    }

    public void removerFornecedor(Fornecedor fornecedor) {
        this.fornecedores.remove(fornecedor);
    }

    public void adicionarFuncionario(Funcionario funcionario) {
        this.funcionarios.add(funcionario);
    }

    public void removerFuncionario(Funcionario funcionario) {
        this.funcionarios.remove(funcionario);
    }

    // Método para calcular o valor do estoque
    public double calcularValor() {
        return getQuantidade() * getPreco();
    }
}
```

```
// Exemplo de uso
public class Main {
  public static void main(String[] args) {
     EstoqueAvancado estoqueAvancado = new EstoqueAvancado("Teclado", 50, 100.0,
"Venda Direta", 200, 5000.0, 10000.0, 2000.0);

     Cliente cliente1 = new Cliente("Cliente X");
     estoqueAvancado.adicionarCliente(cliente1);

     Fornecedor fornecedor1 = new Fornecedor("Fornecedor Y");
     estoqueAvancado.adicionarFornecedor(fornecedor1);
```

```
    Funcionario funcionario1 = new Funcionario("Funcionario Z");
    estoqueAvancado.adicionarFuncionario(funcionario1);

    estoqueAvancado.adicionarEstoque(20);
    estoqueAvancado.removerEstoque(15);

    System.out.println(estoqueAvancado.getQuantidade());  // Deve imprimir 55
  }
}
```

POLIMORFISMO

Agora ao utilizar o polimorfismo basta acrescentar o seguinte :

```
@Override
   public void imprimirDetalhes() {
      System.out.println("Nome do Funcionário: " + nome);
```

adicionamos um método imprimirDetalhes na classe Pessoa que é sobrescrito nas subclasses Cliente, Fornecedor e Funcionario. Isso é um exemplo de polimorfismo, onde o mesmo método tem comportamentos diferentes dependendo da classe do objeto. Quando chamamos o método imprimirDetalhes em um objeto de qualquer uma dessas classes, o método correspondente dessa classe específica será chamado. Isso é conhecido como vinculação dinâmica ou ligação tardia.

CONCLUSAO

Em conclusão, a gestão de estoque é um componente crucial para o sucesso de qualquer negócio, seja ele pequeno, médio ou grande. Através do estudo e implementação de um sistema de gestão de estoque utilizando a Programação Orientada a Objetos (POO), pudemos observar como a abstração, encapsulamento, herança e polimorfismo podem ser aplicados para criar um sistema eficiente e robusto.

A POO permitiu a criação de objetos representando entidades do mundo real, como produtos e fornecedores, facilitando a compreensão e manutenção do código. Além disso, a capacidade de reutilizar código através da herança e polimorfismo resultou em um sistema mais enxuto e eficiente.

No entanto, é importante ressaltar que a escolha da linguagem de programação e do paradigma deve sempre levar em consideração as necessidades específicas do projeto. Embora a POO tenha se mostrado eficaz para este projeto, outras abordagens podem ser mais adequadas dependendo do contexto.

Por fim, este trabalho reforçou a importância da gestão de estoque eficiente para a saúde financeira e operacional de uma empresa. Através da implementação prática, pudemos ver em primeira mão como um sistema projetado pode levar a melhorias significativas na gestão de estoque.