POO – MA

Caso de Uso

Plataforma de Gestão Financeira para PMEs

Nomes: Gabriel Gonçalves Brito

Guilherme Ferreira Sanguinete dos Santos

Professor: Carlos Veríssimo

São Paulo - SP

2023

Sumário

1. Descrição do Domínio do problema 3
2. Requisitos do Cliente 3
2.1. Requisitos Funcionais 3
2.2. Requisitos Não Funcionais 4
3. Casos de Uso 5
3.1 Diagrama de Caso de Uso - Visão Geral 5
3.2 Caso de Uso - Caso de Uso #1 – Inserir dados 6
3.3 Caso de Uso - Caso de Uso #2 – Projetar análise de dados 8
3.4 Caso de Uso - Caso de Uso #3 – Gerar relatórios financeiros 10
3.5 Caso de Uso - Caso de Uso #4 – Inserir entradas e saídas financeiras 11
3.6 Caso de Uso - Caso de Uso #5 – Controle de fluxo de caixa 13
3.7 Caso de Uso - Caso de Uso #6 – Receber notificações de eventos importantes 15
4. Diagrama de Classe 17
4.1 Diagrama de Classe Principal 17
4.2 Diagrama de Classe – Empresa 18
4.3 Diagrama de Classe – Fluxo de Caixa 19
4.4 Diagrama de Classe – Analise Financeira 20
4.5 Diagrama de Classe – Relatórios Financeiros 21
5. Encapsulamento 22
5.1 Encapsulamento classe BalancoFinanceiro 22
5.2 Encapsulamento classe ControlePagamentosReceber 23
5.3 Encapsulamento classe DadoInvestimento 24
5.4 Encapsulamento classe EntradaFinanceira 25
5.5 Encapsulamento classe Fornecedor 26
5.6 Encapsulamento classe GraficoBalanco 27
5.7 Encapsulamento classe GraficoRentabilidadeProdutos 28
5.8 Encapsulamento classe InformacaoMercado 29
5.9 Encapsulamento classe ObrigacaoFiscal 30
5.10 Encapsulamento classe Orcamento 31
5.11 Encapsulamento classe PlanejamentoFinanceiro 32
5.12 Encapsulamento classe PendenciaFinanceira 33
5.13 Encapsulamento classe Rentabilidade 34
5.14 Encapsulamento classe SituacaoFinanceira 35
5.15 Encapsulamento classe SituacaoFinanceira 36
5.15 Encapsulamento classe SugestaoInvestimento 37
6. Baixo Acoplamento das classes 38

1. Descrição do Domínio do problema

Criar uma plataforma de Gestão Financeira para PMEs, projetada especificamente para atender às necessidades financeiras únicas de Pequenas e Médias Empresas. Essa plataforma visa capacitar e auxiliar os empreendedores e gestores a administrarem de forma eficiente e eficaz os principais aspectos financeiros de seus negócios.

2. Requisitos do Cliente

2.1. Requisitos Funcionais

RF1: Registro de transações financeiras, incluindo receitas, despesas, vendas, compras e pagamentos.

RF2: Controle de Fluxo de Caixa, para monitorar entradas e saídas de dinheiro ao longo do tempo.

RF3: Capacitar os usuários a criarem, personalizar e acompanhar orçamentos, auxiliando no planejamento financeiro.

RF4: Possibilitar o cadastro de fornecedores, o acompanhamento de ordens de compra e a gestão do ciclo de compras.

RF5: Permitir o registro e o acompanhamento de contas a pagar (dívidas e obrigações financeiras) e a receber (receitas pendentes).

RF6: Permitir o registro de vendas realizadas e a manutenção de informações sobre clientes.

RF7: Gerar relatórios financeiros, como balanços patrimoniais, demonstrativos de resultados e relatórios de fluxo de caixa.

RF8: Fornecer dados para analisar a rentabilidade de produtos, serviços e projetos.

RF9: Oferecer recursos para calcular e acompanhar obrigações fiscais, auxiliando no cumprimento das regulamentações tributárias.

RF10: Enviar notificações para pagamentos a vencer, prazos fiscais e outras datas importantes.

2.2. Requisitos Não Funcionais

RNF1: A interface da plataforma deve ser intuitiva e fácil de usar, mesmo para usuários com pouca experiência em finanças.

RNF2: A plataforma deve ter uma resposta rápida, mesmo quando lidando com grande volume de dados, para garantir a eficiência do uso.

RFN3: Garantir a segurança das informações de cada usuário.

RFN4: A plataforma deve estar em conformidade com as leis e regulamentações fiscais e financeiras relevantes.

RFN5: O sistema será desenvolvido em Java e suas bibliotecas.

RFN6: Deverá atender a plataforma Desktop.

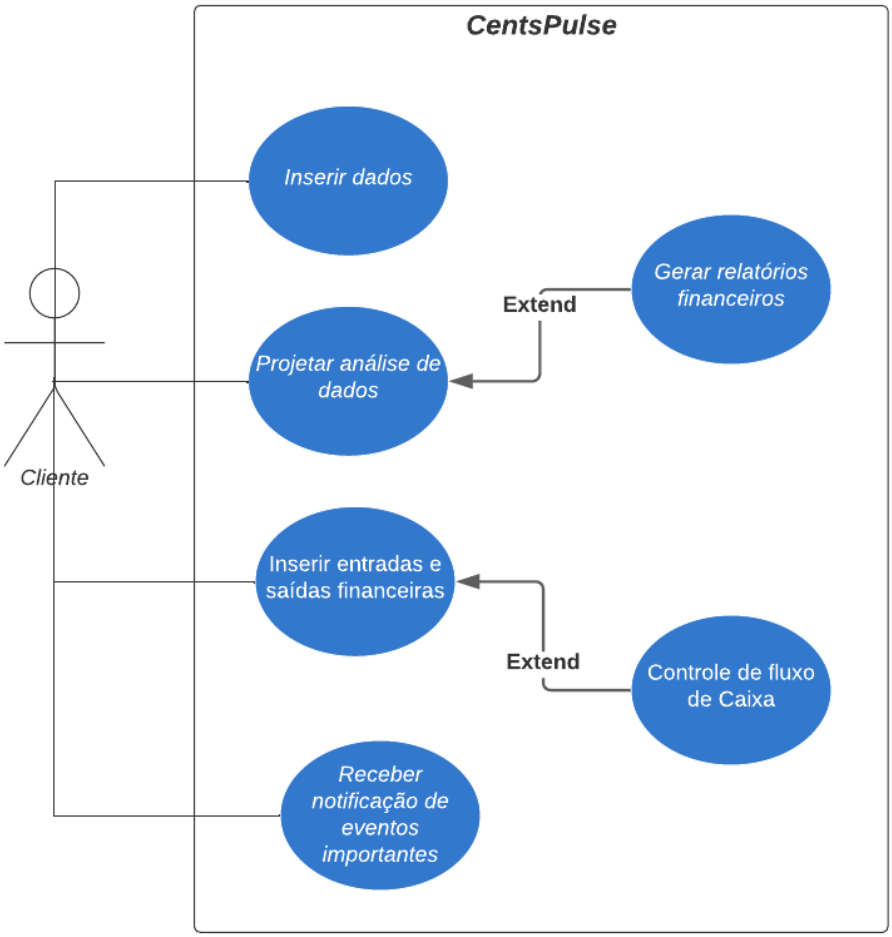
RNF7: O sistema deverá estar ativo 99,9% do tempo.

RNF8: Deverá fornecer formas de login;

RNF9: Design de ícones de fácil identificação e familiaridade com seu uso, para reconhecimento imediato;

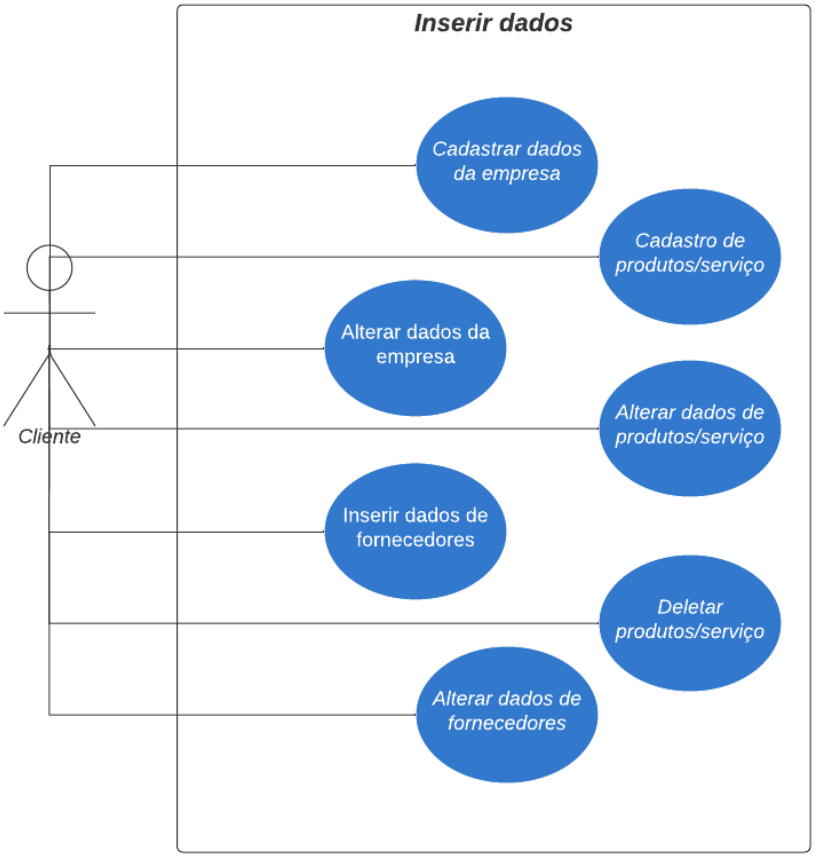
3. Casos de Uso

3.1 Diagrama de Caso de Uso - Visão Geral



3.2 Caso de Uso - Caso de Uso #1 – Inserir dados

3.2.1 Diagrama de Caso de Uso #1



3.2.2 Detalhamento do Caso de uso #1.1 – Cadastrar dados da empresa

Nome do caso de uso:	1.1 Cadastrar dados da empresa
Atores:	Cliente
Trigger:	Necessidade de informação da empresa
Pré-requisito:	Tem que estar logado no sistema
Fluxo de eventos:	Cliente faz o login no sistema, clica no botão “Perfil”, e insere os dados da empresa

3.2.3 Detalhamento do Caso de uso #1.2 – Alterar dados da empresa

Nome do caso de uso:	1.2 Alterar dados da empresa
Atores:	Cliente
Trigger:	Alterar os dados incorretos
Pré-requisito:	Tem que estar logado no sistema
Fluxo de eventos:	Cliente faz o login no sistema, clica no botão “Perfil”, e altera os dados da empresa

3.2.4 Detalhamento do Caso de uso #1.3 – Cadastro de produtos/serviços

Nome do caso de uso:	1.3 Cadastro de produtos/serviços
Atores:	Cliente
Trigger:	Cadastramento de serviços e produtos da empresa
Pré-requisito:	Tem que estar logado no sistema
Fluxo de eventos:	Cliente faz o login no sistema, clica no botão “Perfil”, e insere os serviços e produtos da empresa

3.2.5 Detalhamento do Caso de uso #1.4 – Alterar dados de produtos/serviços

Nome do caso de uso:	1.4 Alterar dados de produtos/serviços
Atores:	Cliente
Trigger:	Alterar os produtos e serviços incorretos
Pré-requisito:	Tem que estar logado no sistema
Fluxo de eventos:	Cliente faz o login no sistema, clica no botão “Perfil”, e altera os serviços e produtos da empresa

3.2.6 Detalhamento do Caso de uso #1.5 – Inserir dados de fornecedores

Nome do caso de uso:	1.5 Inserir dados de fornecedores
Atores:	Cliente
Trigger:	Inserir dados de fornecedores
Pré-requisito:	Tem que estar logado no sistema
Fluxo de eventos:	Cliente faz o login no sistema, clica no botão “Perfil”, e insere os dados dos fornecedores

3.2.7 Detalhamento do Caso de uso #1.6 – Deletar produtos/serviços

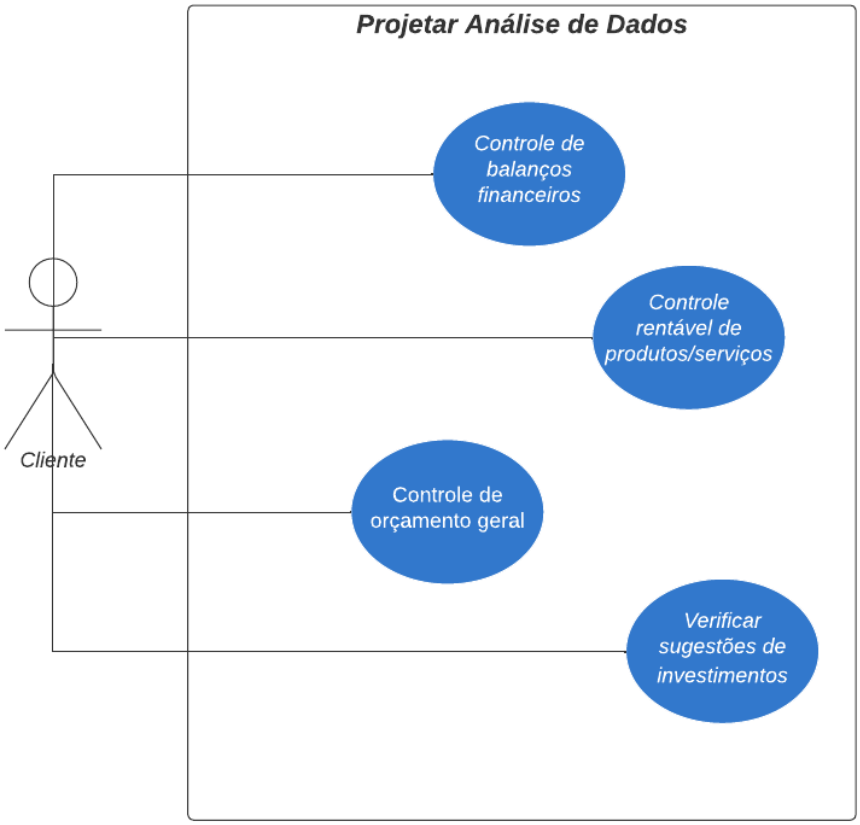
Nome do caso de uso:	1.6 Deletar produtos/serviços
Atores:	Cliente
Trigger:	Apagar produtos e serviços que não existem mais
Pré-requisito:	Tem que estar logado no sistema
Fluxo de eventos:	Cliente faz o login no sistema, clica no botão “Perfil”, e deleta os produtos e serviços que ele quiser

3.2.8 Detalhamento do Caso de uso #1.7 – Alterar dados de fornecedores

Nome do caso de uso:	1.7 Alterar dados de fornecedores
Atores:	Cliente
Trigger:	Alteração dos dados dos fornecedores
Pré-requisito:	Tem que estar logado no sistema
Fluxo de eventos:	Cliente faz o login no sistema, clica no botão “Perfil”, e altera os dados dos fornecedores

3.3 Caso de Uso - Caso de Uso #2 – Projetar análise de dados

3.3.1 Diagrama de Caso de Uso #2



3.3.2 Detalhamento do Caso de uso #2.1 – Controle de balanços financeiros

Nome do caso de uso:	2.1 Controle de balanços financeiros
Atores:	Cliente
Trigger:	Ter o controle da situação financeira da empresa
Pré-requisito:	Tem que estar logado no sistema
Fluxo de eventos:	Cliente faz o login no sistema, clica no botão “Análise”, e clica na opção “Balanço”

3.3.3 Detalhamento do Caso de uso #2.2 – Controle rentável de produtos e serviços

Nome do caso de uso:	2.2 Controle rentável de produtos e serviços
Atores:	Cliente
Trigger:	Ter o controle rentável dos produtos e serviços
Pré-requisito:	Tem que estar logado no sistema
Fluxo de eventos:	Cliente faz o login no sistema, clica no botão “Análise”, e clica na opção “Produtos/Serviços”

3.3.4 Detalhamento do Caso de uso #2.3 – Controle de orçamento geral

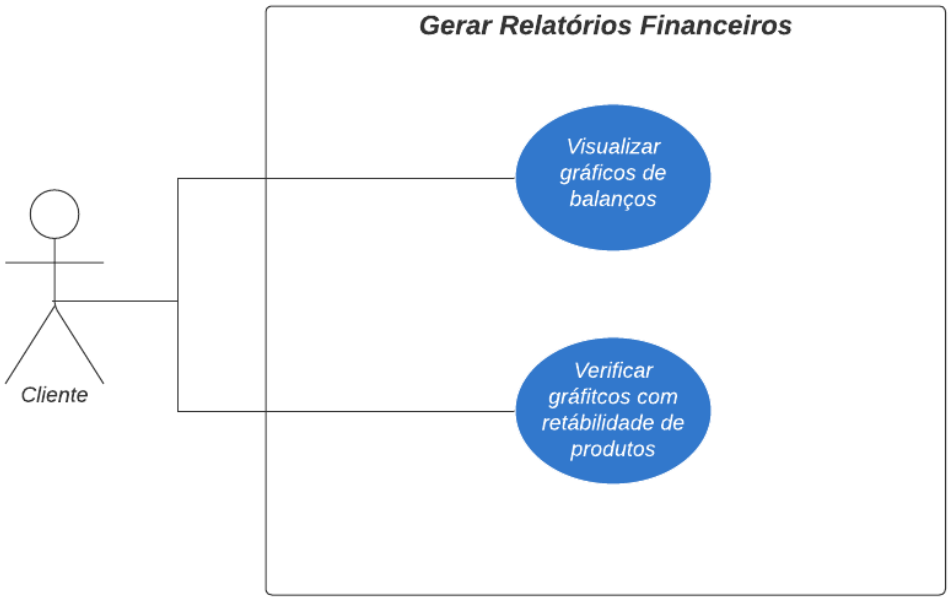
Nome do caso de uso:	2.3 Controle de orçamento geral
Atores:	Cliente
Trigger:	Ter o controle dos orçamentos da empresa
Pré-requisito:	Tem que estar logado no sistema
Fluxo de eventos:	Cliente faz o login no sistema, clica no botão “Análise”, e clica na opção “Orçamento geral”

3.3.5 Detalhamento do Caso de uso #2.4 – Verificar sugestão de investimento

Nome do caso de uso:	2.4 Verificar sugestão de investimento
Atores:	Cliente
Trigger:	Ver novas sugestões de investimentos
Pré-requisito:	Tem que estar logado no sistema
Fluxo de eventos:	Cliente faz o login no sistema, clica no botão “Análise”, e clica na opção “Sugestão para investimento”

3.4 Caso de Uso - Caso de Uso #3 – Gerar relatórios financeiros

3.4.1 Diagrama de Caso de Uso #3



3.4.2 Detalhamento do Caso de uso #3.1 – Visualizar gráficos de balanços

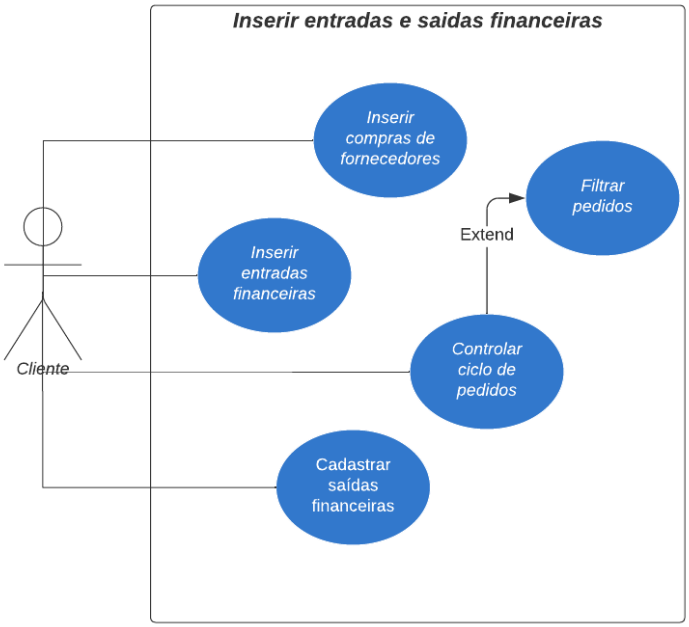
Nome do caso de uso:	3.1 Visualizar gráficos de balanços
Atores:	Cliente
Trigger:	Acompanhar balanços mensais, trimestrais e geral
Pré-requisito:	Tem que estar logado no sistema
Fluxo de eventos:	Cliente faz o login no sistema, clica no botão “Gerar relatórios financeiros”, e clica na opção “Gráficos de balanços”

3.4.3 Detalhamento do Caso de uso #3.2 – Verificar gráficos com rentabilidade de produtos

Nome do caso de uso:	3.2 Verificar gráficos com rentabilidade de produtos
Atores:	Cliente
Trigger:	Ver a rentabilidade dos produtos em forma de gráfico
Pré-requisito:	Tem que estar logado no sistema
Fluxo de eventos:	Cliente faz o login no sistema, clica no botão “Gerar relatórios financeiros”, e clica na opção “Gráfico da rentabilidade dos produtos”

3.5 Caso de Uso - Caso de Uso #4 – Inserir entradas e saídas financeiras

3.5.1 Diagrama de Caso de Uso #4



3.5.2 Detalhamento do Caso de uso #4.1 – Inserir compras de fornecedores

Nome do caso de uso:	4.1 Inserir compras de fornecedores
Atores:	Cliente
Trigger:	Inserir as compras feitas dos fornecedores
Pré-requisito:	Tem que estar logado no sistema
Fluxo de eventos:	Cliente faz o login no sistema, clica no botão “Inserir entradas e saídas financeiras”, e clica na opção “Inserir compras de fornecedores”

3.5.3 Detalhamento do Caso de uso #4.2 – Inserir entradas financeiras

Nome do caso de uso:	4.2 Inserir entradas financeiras
Atores:	Cliente
Trigger:	Inserir a quantidade de dinheiro que entrou na compra dos fornecedores **
Pré-requisito:	Tem que estar logado no sistema
Fluxo de eventos:	Cliente faz o login no sistema, clica no botão “Inserir entradas e saídas financeiras”, e clica na opção “Inserir entradas financeiras”

3.5.4 Detalhamento do Caso de uso #4.3 – Controlar ciclo de pedidos

Nome do caso de uso:	4.3 Controlar ciclo de pedidos
Atores:	Cliente
Trigger:	Ter o controle do ciclo de pedidos
Pré-requisito:	Tem que estar logado no sistema
Fluxo de eventos:	Cliente faz o login no sistema, clica no botão “Inserir entradas e saídas financeiras”, e clica na opção “Controle do ciclo de pedidos”

3.5.5 Detalhamento do Caso de uso #4.4 – Filtar pedidos

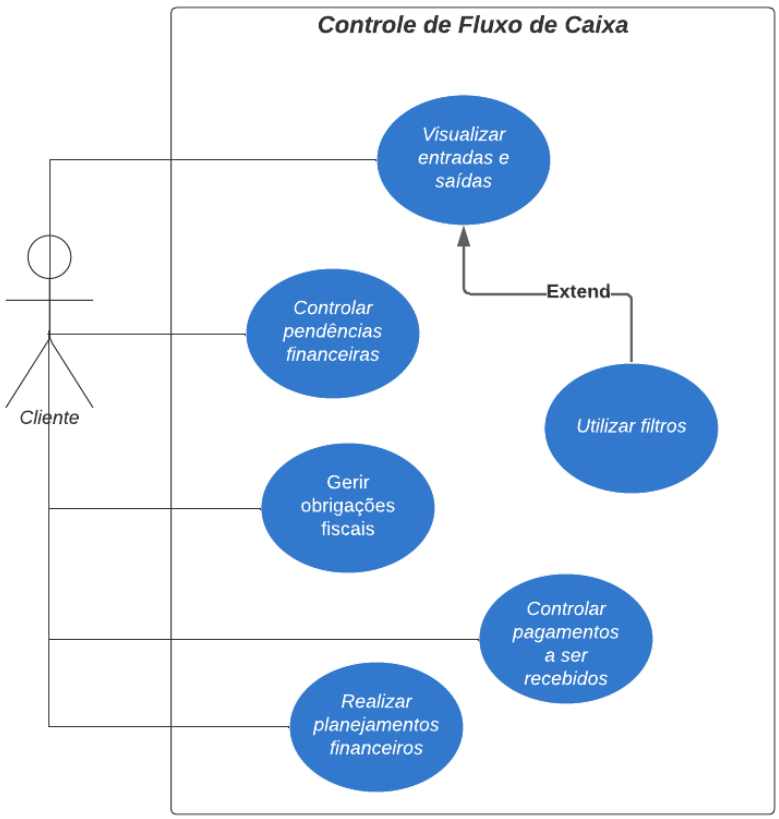
Nome do caso de uso:	4.4 Filtrar pedidos
Atores:	Cliente
Trigger:	Filtrar pedidos por estado e data
Pré-requisito:	Tem que estar logado no sistema
Fluxo de eventos:	Cliente faz o login no sistema, clica no botão “Inserir entradas e saídas financeiras”, clica na opção “Controle do ciclo de pedidos”, e escolher filtro

3.5.6 Detalhamento do Caso de uso #4.5 – Cadastrar saídas financeiras

Nome do caso de uso:	4.5 Cadastrar saídas financeiras
Atores:	Cliente
Trigger:	Fazer cadastro das saídas financeiras
Pré-requisito:	Tem que estar logado no sistema
Fluxo de eventos:	Cliente faz o login no sistema, clica no botão “Inserir entradas e saídas financeiras”, e clica na opção “Cadastrar saídas financeiras”

3.6 Caso de Uso - Caso de Uso #5 – Controle de fluxo de caixa

3.6.1 Diagrama de Caso de Uso #5



3.6.2 Detalhamento do Caso de uso #5.1 – Visualizar entradas e saídas

Nome do caso de uso:	5.1 Visualizar entradas e saídas
Atores:	Cliente
Trigger:	Visualização da entrada e saída da conta da empresa
Pré-requisito:	Tem que estar logado no sistema
Fluxo de eventos:	Cliente faz o login no sistema, clica no botão “Controle de fluxo de caixa”, e clica na opção “Vizualizar entrada e saída”

3.6.3 Detalhamento do Caso de uso #5.2 – Utilizar filtos

Nome do caso de uso:	5.2 Utilizar filtros
Atores:	Cliente
Trigger:	Filtar se quer ver “entrada” ou “saída”
Pré-requisito:	Tem que estar logado no sistema
Fluxo de eventos:	Cliente faz o login no sistema, clica no botão “Controle de fluxo de caixa”, clica na opção “Vizualizar entrada e saída”, e clicar na opção “Filtros”

3.6.4 Detalhamento do Caso de uso #5.3 – Gerar obrigações fiscais

Nome do caso de uso:	5.3 Gerir obrigações fiscais
Atores:	Cliente
Trigger:	Gerar todas obrigações fiscais da empresa
Pré-requisito:	Tem que estar logado no sistema
Fluxo de eventos:	Cliente faz o login no sistema, clica no botão “Controle de fluxo de caixa”, e clica na opção “Gerar obrigações fiscais”

3.6.5 Detalhamento do Caso de uso #5.4 – Controlar pagamentos a ser recebidos

Nome do caso de uso:	5.4 Controlar pagamentos a ser recebidos
Atores:	Cliente
Trigger:	Ter o controle dos pagamentos a serem recebidos
Pré-requisito:	Tem que estar logado no sistema
Fluxo de eventos:	Cliente faz o login no sistema, clica no botão “Controle de fluxo de caixa”, e clica na opção “Pagamentos a ser recebidos”

3.6.6 Detalhamento do Caso de uso #5.5 – Realizar planejamentos financeiros

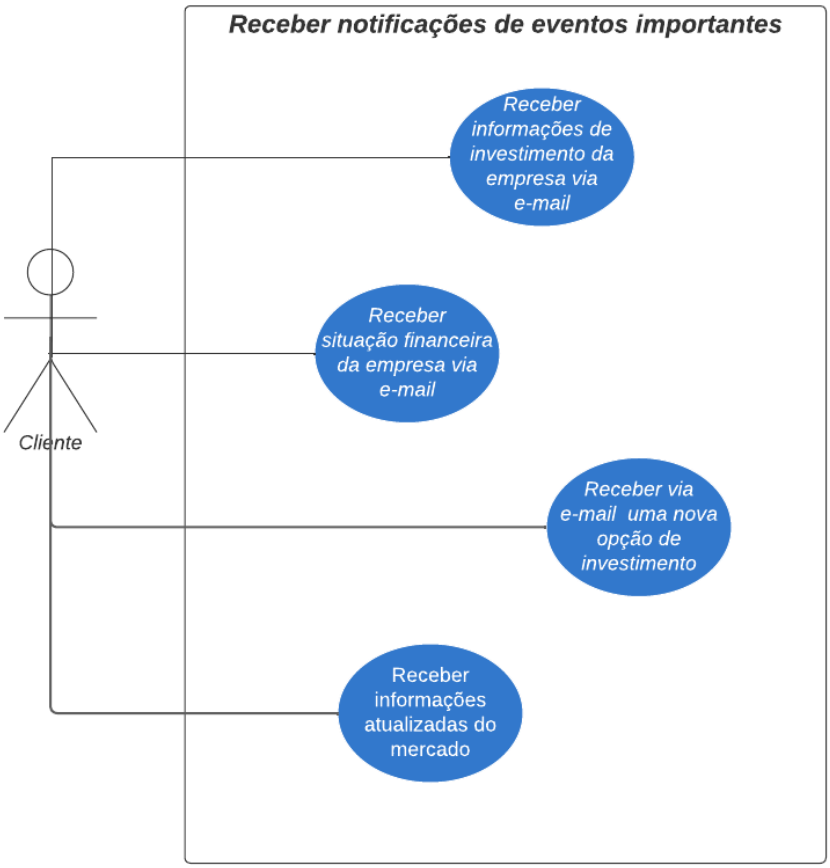
Nome do caso de uso:	5.3 Gerir obrigações fiscais
Atores:	Cliente
Trigger:	Fazer o planejamento financeiro para o futuro
Pré-requisito:	Tem que estar logado no sistema
Fluxo de eventos:	Cliente faz o login no sistema, clica no botão “Controle de fluxo de caixa”, e clica na opção “Planejamento financeiro”

3.6.7 Detalhamento do Caso de uso #5.6 – Controlar pendências financeiras

Nome do caso de uso:	5.6 Controlar pendências financeiras
Atores:	Cliente
Trigger:	Ter o controle das pendências financeiras da empresa
Pré-requisito:	Tem que estar logado no sistema
Fluxo de eventos:	Cliente faz o login no sistema, clica no botão “Controle de fluxo de caixa”, e clica na opção “Pendências financeiras”

3.7 Caso de Uso - Caso de Uso #6 – Receber notificações de eventos importantes

3.7.1 Diagrama de Caso de Uso #6



3.7.2 Detalhamento do Caso de uso #6.1 – Receber informações de investimento da empresa via e-mail

Nome do caso de uso:	6.1 Receber informações de investimento da empresa via e-mail
Atores:	Cliente
Trigger:	Ter as informações do investimento da sua empresa pelo e-mail
Pré-requisito:	Tem que estar logado no sistema
Fluxo de eventos:	Cliente faz o login no sistema, clica no botão “Receber notificações via e-mail”, e clica na opção “Receber informações de investimento feito”

3.7.3 Detalhamento do Caso de uso #6.2 – Receber via e-mail uma nova opção de investimento

Nome do caso de uso:	6.2 Receber via e-mail uma nova opção de investimento
Atores:	Cliente
Trigger:	Receber novas opções para investir
Pré-requisito:	Tem que estar logado no sistema
Fluxo de eventos:	Cliente faz o login no sistema, clica no botão “Receber notificações via e-mail”, e clica na opção “Receber novas opções para investir”

3.7.4 Detalhamento do Caso de uso #6.3 – Receber situação financeira da empresa

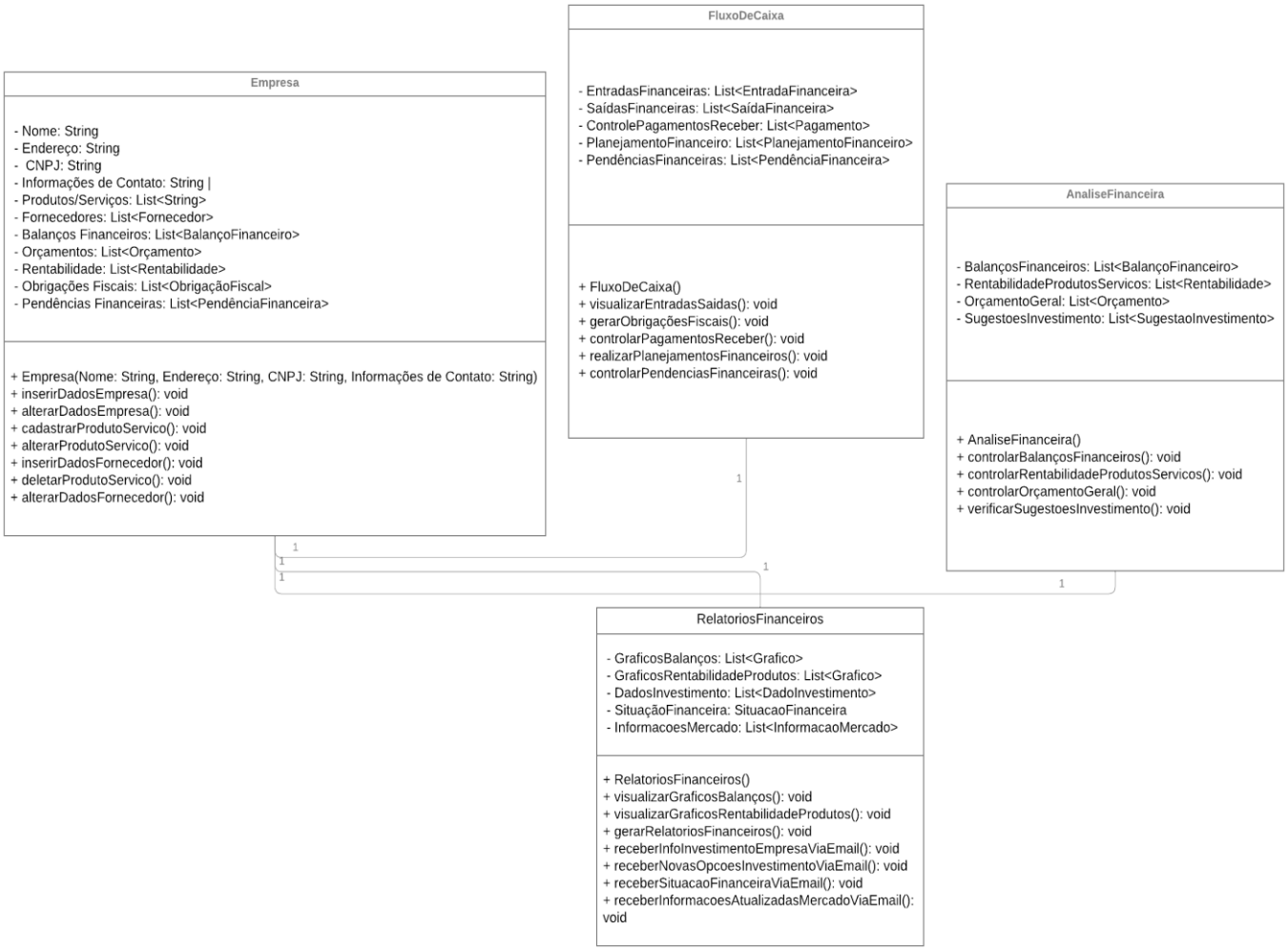
Nome do caso de uso:	6.3 Receber situação financeira da empresa
Atores:	Cliente
Trigger:	Ter acesso a situação financeira da empresa
Pré-requisito:	Tem que estar logado no sistema
Fluxo de eventos:	Cliente faz o login no sistema, clica no botão “Receber notificações via e-mail”, e clica na opção “Receber situação financeira”

3.7.5 Detalhamento do Caso de uso #6.4 – Receber informações atualizadas do mercado

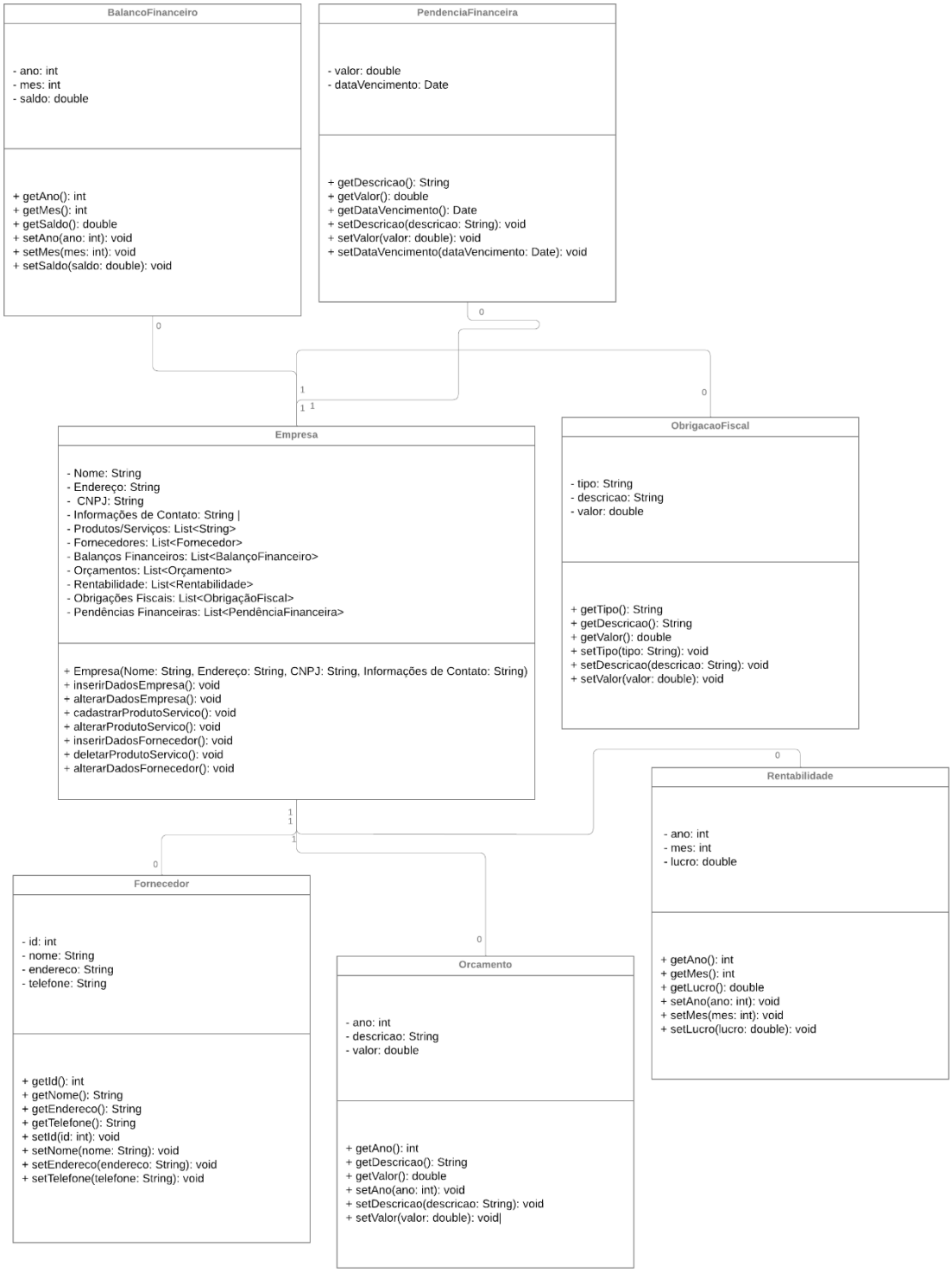
Nome do caso de uso:	6.4 Receber informações atualizadas do mercado
Atores:	Cliente
Trigger:	Ter informações sobre novos serviços e investimentos do mercado
Pré-requisito:	Tem que estar logado no sistema
Fluxo de eventos:	Cliente faz o login no sistema, clica no botão “Receber notificações via e-mail”, e clica na opção “Receber atualizações de mercado”

4. Diagrama de Classe

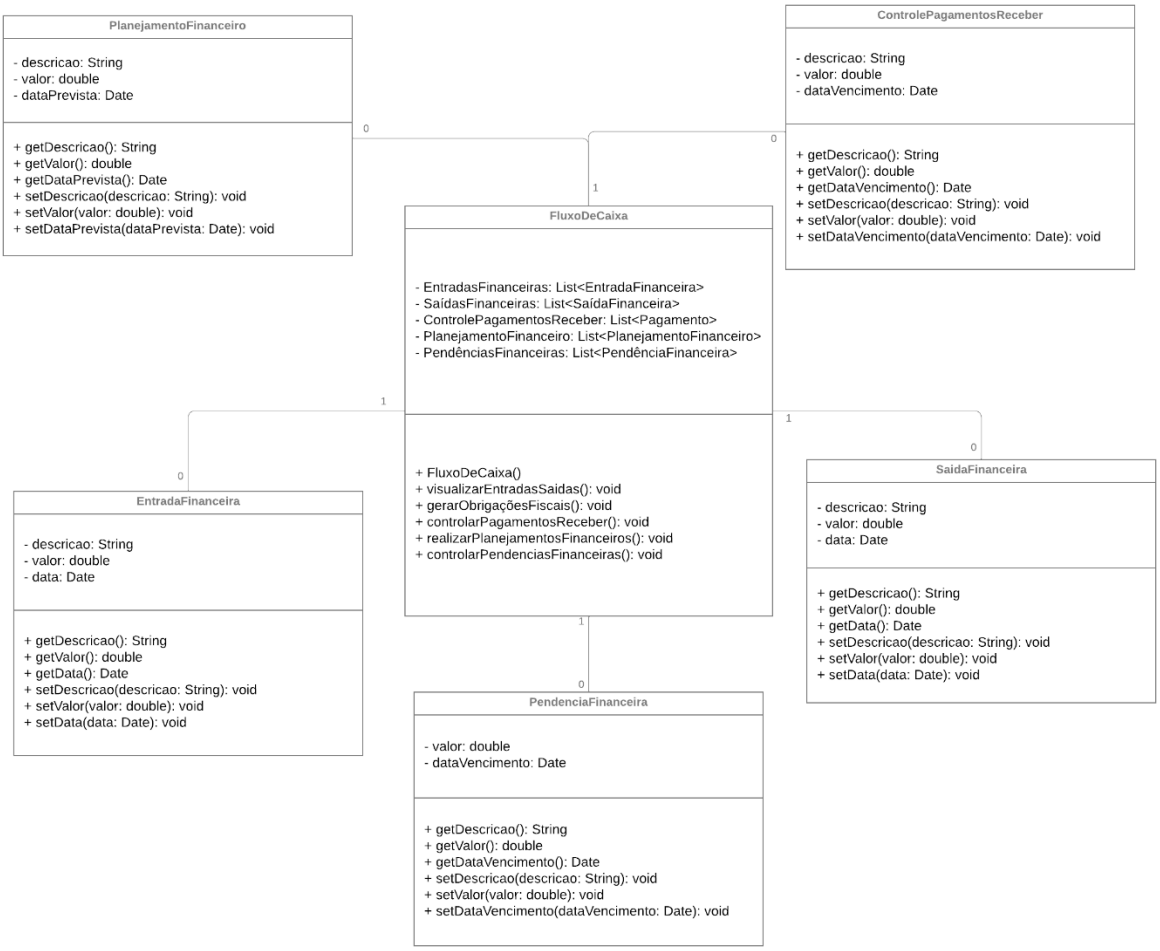
4.1 Diagrama de Classe Principal



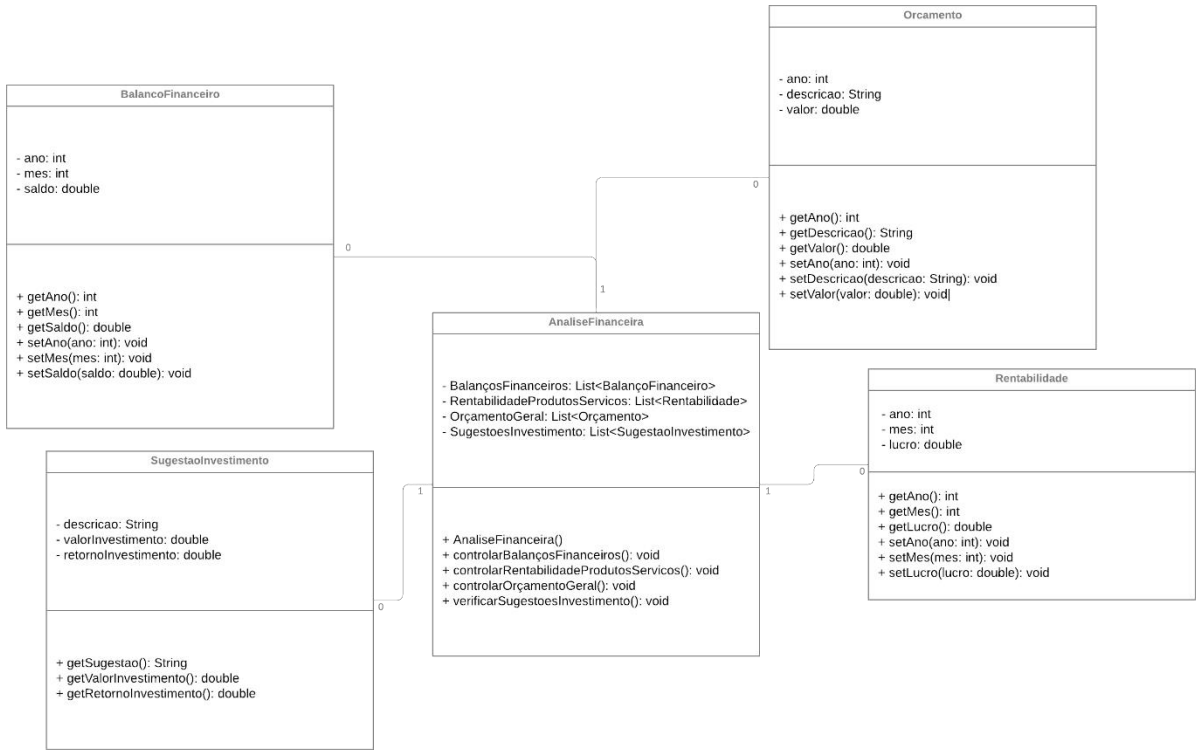
4.2 Diagrama de Classe – Empresa



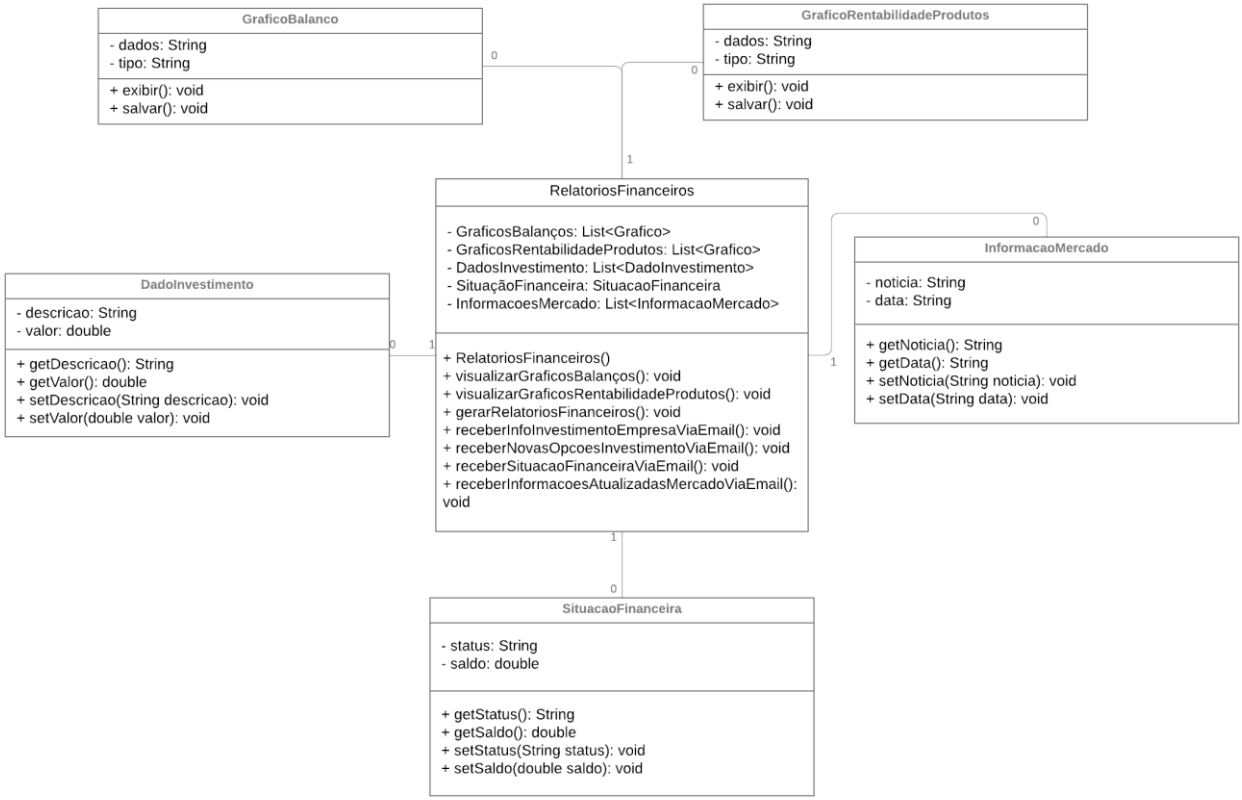
4.3 Diagrama de Classe – Fluxo de Caixa



4.4 Diagrama de Classe – Analise Financeira



4.5 Diagrama de Classe – Relatórios Financeiros



5. Encapsulamento

5.1 Encapsulamento classe BalancoFinanceiro

```java
public class BalancoFinanceiro {
    private int ano;
    private int mes;
    private double saldo;

    public BalancoFinanceiro(int ano, int mes, double saldo) {
        this.ano = ano;
        this.mes = mes;
        this.saldo = saldo;
    }

    public int getAno() {
        return ano;
    }

    public void setAno(int ano) {
        this.ano = ano;
    }

    public int getMes() {
        return mes;
    }

    public void setMes(int mes) {
        this.mes = mes;
    }

    public double getSaldo() {
        return saldo;
    }

    public void setSaldo(double saldo) {
        this.saldo = saldo;
    }
}
```

5.2 Encapsulamento classe ControlePagamentosReceber

```java
package centsPulse;
import java.util.Date;

public class ControlePagamentosReceber {
    private String descricao;
    private double valor;
    private Date dataVencimento;

    public ControlePagamentosReceber(String descricao, double valor, Date
dataVencimento) {
        this.descricao = descricao;
        this.valor = valor;
        this.dataVencimento = dataVencimento;
    }

    public String getDescricao() {
        return descricao;
    }

    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }

    public double getValor() {
        return valor;
    }

    public void setValor(double valor) {
        this.valor = valor;
    }

    public Date getDataVencimento() {
        return dataVencimento;
    }

    public void setDataVencimento(Date dataVencimento) {
        this.dataVencimento = dataVencimento;
    }
}
```

5.3 Encapsulamento classe DadoInvestimento

```java
package centsPulse;

public class DadoInvestimento {
    private String descricao;
    private double valor;

    public DadoInvestimento(String descricao, double valor) {
        this.descricao = descricao;
        this.valor = valor;
    }

    public String getDescricao() {
        return descricao;
    }

    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }

    public double getValor() {
        return valor;
    }

    public void setValor(double valor) {
        this.valor = valor;
    }
}
```

5.4 Encapsulamento classe EntradaFinanceira

```java
import java.util.Date;

public class EntradaFinanceira {
    private String descricao;
    private double valor;
    private Date data;

    public EntradaFinanceira(String descricao, double valor, Date data) {
        this.descricao = descricao;
        this.valor = valor;
        this.data = data;
    }

    public String getDescricao() {
        return descricao;
    }

    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }

    public double getValor() {
        return valor;
    }

    public void setValor(double valor) {
        this.valor = valor;
    }

    public Date getData() {
        return data;
    }

    public void setData(Date data) {
        this.data = data;
    }
}
```

5.5 Encapsulamento classe Fornecedor

```
public class Fornecedor {
    private int id;
    private String nome;
    private String endereco;
    private String telefone;

    public Fornecedor(int id, String nome, String endereco, String
telefone) {
        this.id = id;
        this.nome = nome;
        this.endereco = endereco;
        this.telefone = telefone;
    }
    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 getEndereco() {
        return endereco;
    }
    public void setEndereco(String endereco) {
        this.endereco = endereco;
    }
    public String getTelefone() {
        return telefone;
    }
    public void setTelefone(String telefone) {
        this.telefone = telefone;
    }
}
```

5.6 Encapsulamento classe GraficoBalanco

```java
public class GraficoBalanco {
    private String dados;
    private String tipo;

    public GraficoBalanco(String dados, String tipo) {
        this.dados = dados;
        this.tipo = tipo;
    }

    public void exibir() {
        System.out.println("Exibindo gráfico de balanço:");
        System.out.println("Dados: " + dados);
        System.out.println("Tipo: " + tipo);
    }

    public void salvar() {
        System.out.println("Salvando gráfico de balanço:");
        System.out.println("Dados: " + dados);
        System.out.println("Tipo: " + tipo);
    }

    public String getDados() {
        return dados;
    }

    public void setDados(String dados) {
        this.dados = dados;
    }

    public String getTipo() {
        return tipo;
    }

    public void setTipo(String tipo) {
        this.tipo = tipo;
    }
```

5.7 Encapsulamento classe GraficoRentabilidadeProdutos

```
public class GraficoRentabilidadeProdutos {
    private String dados;
    private String tipo;

    public GraficoRentabilidadeProdutos(String dados, String tipo) {
        this.dados = dados;
        this.tipo = tipo;
    }

    public void exibir() {
        System.out.println("Exibindo gráfico de rentabilidade de
produtos:");
        System.out.println("Dados: " + dados);
        System.out.println("Tipo: " + tipo);
    }

    public void salvar() {
        System.out.println("Salvando gráfico de rentabilidade de
produtos:");
        System.out.println("Dados: " + dados);
        System.out.println("Tipo: " + tipo);
    }

    public String getDados() {
        return dados;
    }

    public void setDados(String dados) {
        this.dados = dados;
    }

    public String getTipo() {
        return tipo;
    }

    public void setTipo(String tipo) {
        this.tipo = tipo;
    }
}
```

5.8 Encapsulamento classe InformacaoMercado

```java
public class InformacaoMercado {
    private String noticia;
    private String data;

    public InformacaoMercado(String noticia, String data) {
        this.noticia = noticia;
        this.data = data;
    }

    public String getNoticia() {
        return noticia;
    }

    public void setNoticia(String noticia) {
        this.noticia = noticia;
    }

    public String getData() {
        return data;
    }

    public void setData(String data) {
        this.data = data;
    }
}
```

5.9 Encapsulamento classe ObrigacaoFiscal

```
public class ObrigacaoFiscal {
    private String tipo;
    private String descricao;
    private double valor;

    public ObrigacaoFiscal(String tipo, String descricao, double valor) {
        this.tipo = tipo;
        this.descricao = descricao;
        this.valor = valor;
    }

    public String getTipo() {
        return tipo;
    }

    public void setTipo(String tipo) {
        this.tipo = tipo;
    }

    public String getDescricao() {
        return descricao;
    }

    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }

    public double getValor() {
        return valor;
    }

    public void setValor(double valor) {
        this.valor = valor;
    }

    @Override
    public String toString() {
        return "ObrigacaoFiscal [tipo=" + tipo + ", descricao=" +
descricao + ", valor=" + valor + "]";
    }
}
```

5.10 Encapsulamento classe Orcamento

```java
public class Orcamento {
    private int ano;
    private String descricao;
    private double valor;

    public Orcamento(int ano, String descricao, double valor) {
        this.ano = ano;
        this.descricao = descricao;
        this.valor = valor;
    }

    public int getAno() {
        return ano;
    }

    public void setAno(int ano) {
        this.ano = ano;
    }

    public String getDescricao() {
        return descricao;
    }

    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }

    public double getValor() {
        return valor;
    }

    public void setValor(double valor) {
        this.valor = valor;
    }
}
```

5.11 Encapsulamento classe PlanejamentoFinanceiro

```
import java.util.Date;

public class PlanejamentoFinanceiro {
    private String descricao;
    private double valor;
    private Date dataPrevista;

    public PlanejamentoFinanceiro(String descricao, double valor, Date
dataPrevista) {
        this.descricao = descricao;
        this.valor = valor;
        this.dataPrevista = dataPrevista;
    }

    public String getDescricao() {
        return descricao;
    }

    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }

    public double getValor() {
        return valor;
    }

    public void setValor(double valor) {
        this.valor = valor;
    }

    public Date getDataPrevista() {
        return dataPrevista;
    }

    public void setDataPrevista(Date dataPrevista) {
        this.dataPrevista = dataPrevista;
    }
}
```

5.12 Encapsulamento classe PendenciaFinanceira

```java
import java.util.Date;

public class PendenciaFinanceira {
    private String descricao;
    private double valor;
    private Date dataVencimento;

    public PendenciaFinanceira(String descricao, double valor, Date
dataVencimento) {
        this.descricao = descricao;
        this.valor = valor;
        this.dataVencimento = dataVencimento;
    }

    public String getDescricao() {
        return descricao;
    }

    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }

    public double getValor() {
        return valor;
    }

    public void setValor(double valor) {
        this.valor = valor;
    }

    public Date getDataVencimento() {
        return dataVencimento;
    }

    public void setDataVencimento(Date dataVencimento) {
        this.dataVencimento = dataVencimento;
    }
}
```

5.13 Encapsulamento classe Rentabilidade

```java
public class Rentabilidade {
    private int ano;
    private int mes;
    private double lucro;

    public Rentabilidade(int ano, int mes, double lucro) {
        this.ano = ano;
        this.mes = mes;
        this.lucro = lucro;
    }

    public int getAno() {
        return ano;
    }

    public void setAno(int ano) {
        this.ano = ano;
    }

    public int getMes() {
        return mes;
    }

    public void setMes(int mes) {
        this.mes = mes;
    }

    public double getLucro() {
        return lucro;
    }

    public void setLucro(double lucro) {
        this.lucro = lucro;
    }
}
```

5.14 Encapsulamento classe SituacaoFinanceira

```
import java.util.Date;

public class SaidaFinanceira {
    private String descricao;
    private double valor;
    private Date data;

    public SaidaFinanceira(String descricao, double valor, Date data) {
        this.descricao = descricao;
        this.valor = valor;
        this.data = data;
    }

    public String getDescricao() {
        return descricao;
    }

    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }

    public double getValor() {
        return valor;
    }

    public void setValor(double valor) {
        this.valor = valor;
    }

    public Date getData() {
        return data;
    }

    public void setData(Date data) {
        this.data = data;
    }

    @Override
    public String toString() {
        return "SaidaFinanceira [descricao=" + descricao + ", valor=" +
valor + ", data=" + data + "]";
    }
}
```

5.15 Encapsulamento classe SituacaoFinanceira

```java
package centsPulse;

public class SituacaoFinanceira {
    private String status;
    private double saldo;

    public SituacaoFinanceira(String status, double saldo) {
        this.status = status;
        this.saldo = saldo;
    }

    public String getStatus() {
        return status;
    }

    public void setStatus(String status) {
        this.status = status;
    }

    public double getSaldo() {
        return saldo;
    }

    public void setSaldo(double saldo) {
        this.saldo = saldo;
    }
}
```

5.15 Encapsulamento classe SugestaoInvestimento

```
package centsPulse;

public class SugestaoInvestimento {
    private String descricao;
    private double valorInvestimento;
    private double retornoInvestimento;

    public SugestaoInvestimento(String descricao, double
valorInvestimento, double retornoInvestimento) {
        this.descricao = descricao;
        this.valorInvestimento = valorInvestimento;
        this.retornoInvestimento = retornoInvestimento;
    }

    public String getDescricao() {
        return descricao;
    }

    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }

    public double getValorInvestimento() {
        return valorInvestimento;
    }

    public void setValorInvestimento(double valorInvestimento) {
        this.valorInvestimento = valorInvestimento;
    }

    public double getRetornoInvestimento() {
        return retornoInvestimento;
    }

    public void setRetornoInvestimento(double retornoInvestimento) {
        this.retornoInvestimento = retornoInvestimento;
    }

    public String getSugestao() {
        return "Descrição: " + descricao + ", Valor do Investimento: " +
valorInvestimento + ", Retorno Esperado: " + retornoInvestimento;
    }
}
```

6. Baixo Acoplamento das classes

6.1 Acoplamento da classe BalancoFinanceiro

BalancoFinanceiro: O código fornecido exemplifica o baixo acoplamento de classes por meio do encapsulamento, métodos de acesso e abstração. As variáveis de instância são privadas, acessíveis apenas por métodos públicos, promovendo controle sobre os dados. A presença de um construtor parametrizado permite a inicialização dos objetos sem depender de métodos adicionais, garantindo independência de inicialização. Assim, alterações na classe podem ocorrer sem afetar outras partes do sistema, mantendo uma baixa dependência e facilitando a manutenção e modificação do código.

6.2 Acoplamento da classe ControlePagamentosReceber

ControlePagamentosReceber: O código da classe “ControlePagamentosReceber” também demonstra baixo acoplamento, seguindo os princípios de encapsulamento e abstração. As variáveis de instância (“descricao”, “valor” e “dataVencimento”) são privadas, acessíveis somente por meio de métodos públicos, garantindo controle sobre os dados e ocultando a implementação interna. Além disso, a classe fornece métodos “get” e “set” para acessar e modificar os valores, permitindo que outras partes do sistema interajam com esses dados de forma controlada, sem depender diretamente da estrutura interna da classe. O uso de tipos de dados abstratos, como “Date”, também contribui para o baixo acoplamento, pois encapsula a complexidade do gerenciamento de datas. Assim, a classe “ControlePagamentosReceber” oferece uma interface clara e independente, facilitando futuras modificações internas sem afetar outras partes do sistema, promovendo assim um baixo acoplamento entre as classes.

6.3 Acoplamento da classe DadoInvestimento

DadoInvestimento: O código da classe “DadoInvestimento” exemplifica o baixo acoplamento ao seguir princípios essenciais de orientação a objetos. As variáveis de instância (“descricao” e “valor”) são privadas, acessíveis apenas por meio de métodos públicos (“getDescricao()”, setDescricao()”, “getValor()”, “setValor()”). Ao oferecer métodos de acesso e modificação, a classe permite interações controladas com seus dados, garantindo que outras classes possam acessar e modificar esses valores sem precisar conhecer os detalhes internos da implementação. Portanto, a classe “DadoInvestimento” segue o princípio de baixo acoplamento ao fornecer uma interface clara e independente para interações externas, facilitando a manutenção e a expansão do sistema sem introduzir dependências desnecessárias.

6.4 Acoplamento da classe EntradaFinanceira

EntradaFinanceira: O código da classe “EntradaFinanceira” adere ao princípio de baixo acoplamento por meio do encapsulamento e da abstração dos dados. As variáveis de instância (“descricao”, “valor” e “data”) são privadas, sendo acessíveis somente através de métodos públicos (“getDescricao()”, “setDescricao()”, “getValor()”, “setValor()”, “getData()”, setData()”). Esse encapsulamento garante que os detalhes internos da implementação da classe sejam ocultos de outras partes do sistema, reduzindo assim as dependências externas. Além disso, ao utilizar a classe “Date” para representar a data, a implementação dos detalhes complexos relacionados ao gerenciamento de datas é encapsulada, proporcionando uma interface simplificada para os usuários da classe. Dessa forma, qualquer alteração interna na forma como a data é manipulada não afetará o restante do sistema, mantendo um baixo acoplamento entre as classes.

6.5 Acoplamento da classe Fornecedor

Fornecedor: O código da classe “Fornecedor” demonstra baixo acoplamento através do encapsulamento e da clara definição de interfaces. As variáveis de instância (id, nome, endereco e telefone) são privadas, acessíveis apenas por meio de métodos públicos (“getId()”, “setId()”, “getNome()”, “setNome()”, “getEndereco()”, “setEndereco()”, “getTelefone()”, “setTelefone()”). Ao esconder os detalhes da implementação interna da classe, outras partes do sistema podem interagir com um objeto Fornecedor sem necessidade de conhecer ou depender dos detalhes de como os dados são armazenados ou manipulados. Isso cria um alto nível de independência entre a classe “Fornecedor” e outras classes que a utilizam, facilitando futuras modificações internas na classe “Fornecedor” sem afetar outras partes do sistema.

6.6 Acoplamento da classe GraficoBalanco

GraficoBalanco: A classe “GraficoBalanco” demonstra baixo acoplamento principalmente através da sua clara separação de responsabilidades e da encapsulação dos dados. Ao manter os atributos “dados” e “tipo” privados e fornecer métodos públicos (“getDados()”, “setDados()”, “getTipo()”, “setTipo()”) para acessá-los, a classe oculta os detalhes da sua implementação interna. Isso significa que outras partes do sistema podem interagir com um objeto “GraficoBalanco” sem precisar saber como os dados são armazenados ou manipulados, promovendo um baixo nível de dependência.

Além disso, a classe oferece métodos “exibir()” e “salvar()”, que representam diferentes comportamentos relacionados à exibição e ao armazenamento do gráfico de balanço. Ao separar essas funcionalidades em métodos distintos, a classe adere ao princípio da separação de interesses, tornando cada método responsável por uma única tarefa específica. Isso não apenas facilita a manutenção do código, mas também permite que esses comportamentos sejam modificados independentemente um do outro, garantindo um baixo acoplamento entre as funcionalidades da classe.

6.7 Acoplamento da classe GraficoRentabilidadeProdutos

GraficoRentabilidadeProdutos: O código da classe “GraficoRentabilidadeProdutos” mantém um baixo acoplamento devido à encapsulação efetiva e à separação clara de responsabilidades. Os atributos “dados” e “tipo” são privados e só podem ser acessados através de métodos públicos (“getDados()”, “setDados()”, “getTipo()”, “setTipo()”), garantindo que outras partes do sistema não dependam da estrutura interna da classe. Isso promove a independência entre a classe “GraficoRentabilidadeProdutos” e outras partes do sistema, pois essas partes podem interagir com a classe sem precisar entender ou depender dos detalhes de implementação, reduzindo assim a complexidade e o acoplamento do sistema como um todo.

Além disso, a classe fornece métodos “exibir()” e “salvar()”, que separam claramente a lógica de exibição da lógica de salvamento do gráfico de rentabilidade. Essa separação de comportamentos em métodos distintos promove o princípio da responsabilidade única, facilitando a manutenção e modificação futura da classe. Por exemplo, se a forma de exibição ou o método de salvamento precisar ser alterado, isso pode ser feito independentemente um do outro, sem afetar outras partes do sistema.

6.8 Acoplamento da classe InformacaoMercado

InformacaoMercado: O código da classe “InformacaoMercado” demonstra baixo acoplamento devido à encapsulação eficaz dos dados. As variáveis de instância “noticia” e “data” são privadas e só podem ser acessadas através de métodos públicos (“getNoticia()”, “setNoticia()”, “getData()”, “setData()”).

Ao fornecer métodos de acesso (“getNoticia()” e “getData()”) e métodos de modificação (“setNoticia()” e “setData()”), a classe oferece uma interface clara e controlada para interações externas. Isso significa que outras classes podem obter e modificar as informações de mercado de forma segura e eficaz, sem precisar entender como essas informações são armazenadas ou manipuladas internamente.

Essa independência entre a implementação interna da classe “InformacaoMercado” e outras partes do sistema resulta em um baixo acoplamento. Modificações futuras na forma como as informações de mercado são gerenciadas podem ser feitas na classe sem afetar outras partes do sistema, desde que a interface pública (métodos públicos) seja mantida. Assim, a classe “InformacaoMercado” promove a flexibilidade e a manutenção simplificada no sistema, atendendo ao princípio de baixo acoplamento.

6.9 Acoplamento da classe ObrigacaoFiscal

ObrigacaoFiscal: O código da classe "ObrigacaoFiscal" demonstra baixo acoplamento usando encapsulamento e definições claras de interface. Variáveis de instância ("tipo", "descrição" e "valor") são privadas e só podem ser acessadas através de métodos públicos ("getTipo()", "setTipo()", "getDescricao()", "setDescricao()" .), "getValor()", "setValor()"). Protege as informações internas de uma classe, permitindo que outras partes do sistema acessem e modifiquem essas propriedades através de uma interface controlada. As outras classes podem interagir com o objeto “ObrigacaoFiscal” sem conhecer ou depender das informações internas da classe. Isto reduz as dependências e aumenta a flexibilidade porque a implementação interna de uma classe pode ser alterada sem afetar outras partes do sistema, desde que a interface pública permaneça consistente.

6.10 Acoplamento da classe Orcamento

Orcamento: O código da classe “Orcamento” demonstra baixo acoplamento por meio do encapsulamento dos dados e da exposição controlada dos métodos de acesso. As variáveis de instância (“ano”, “descricao” e “valor”) são privadas, inacessíveis diretamente de fora da classe. Em vez disso, métodos públicos (“getAno()”, “setAno()”, “getDescricao()”, “setDescricao()”, “getValor()”, “setValor()”) são fornecidos para acessar e modificar esses dados.

Ao encapsular os dados e fornecer métodos públicos para acessá-los, a classe “Orcamento” cria uma interface clara e independente para outras partes do sistema. Isso significa que outras classes podem interagir com objetos “Orcamento” sem precisar entender ou depender dos detalhes internos de como os dados são armazenados ou manipulados. Esta independência é crucial para o baixo acoplamento, pois permite que a implementação interna da classe seja modificada sem afetar outras partes do sistema, desde que a interface pública (assinatura dos métodos) seja mantida consistente.

Além disso, a classe “Orcamento” segue o princípio da responsabilidade única, mantendo-se focada apenas na gestão dos dados relacionados ao orçamento. Em resumo, ao encapsular seus dados e fornecer uma interface controlada, a classe “Orcamento” promove um baixo acoplamento ao facilitar a interação com outras partes do sistema de maneira independente e flexível.

6.11 Acoplamento da classe PendenciaFinanceira

PendenciaFinanceira: O código da classe " PendenciaFinanceira" demonstra acoplamento fraco encapsulando dados e fornecendo métodos públicos controlados para acessar e modificar esses dados. Variáveis de instância ("descrição", "valor" e "expira") são privadas e não podem ser acessadas externamente. Em vez disso, a classe possui métodos públicos ("getDescricao()", "setDescricao()", "getValor()", "setValor()", "getDéscricao()", "setDataDescricao()" que podem acessar e modificar essas variáveis .) é fornecido. Ou seja, ao encapsular os dados e fornecer uma interface clara e controlada, a classe “PendenciaFinancial” promove baixo acoplamento, possibilitando flexibilidade e manutenção eficiente do sistema.

6.12 Acoplamento da classe PlanejamnetoFinanceiro

PlanejamnetoFinanceiro: O código da classe “PlanejamentoFinanceiro” demonstra baixo acoplamento através do encapsulamento e da clara definição de interfaces. As variáveis de instância (“descricao”, “valor” e “dataPrevista”) são privadas, inacessíveis diretamente de fora da classe. Em vez disso, a classe oferece métodos públicos (“getDescricao()”, “setDescricao()”, “getValor()”, “setValor()”, “getDataPrevista()”, “setDataPrevista()”) para acessar e modificar esses dados. Em resumo, ao encapsular os dados e fornecer uma interface clara e controlada, a classe “PlanejamentoFinanceiro” promove um baixo acoplamento, permitindo a flexibilidade e a manutenção eficiente do sistema.

6.13 Acoplamento da classe Rentabilidade

Rentabilidade: O código da classe “Rentabilidade” reflete baixo acoplamento por meio do encapsulamento dos dados e da clara definição de interfaces. As variáveis de instância (“ano”, “mês” e “lucro”) são privadas, inacessíveis diretamente de fora da classe. Para acessar ou modificar esses dados, a classe oferece métodos públicos (“getAno()”, “setAno()”, “getMes()”, “setMes()”, “getLucro()”, “setLucro()”) que garantem controle sobre como esses dados são manipulados. A utilização de tipos de dados primitivos e simples (“int” e “double”) para representar ano, mês e lucro também contribui para o baixo acoplamento, pois elimina a necessidade de entender estruturas de dados complexas para interagir com a classe.

6.14 Acoplamento da classe SituacaoFinanceira

SituacaoFinanceira: O código da classe "SituacaoFinanceira" segue o princípio do acoplamento fraco utilizando encapsulamento de dados e controle de divulgação do método de acesso. As variáveis de instância (“status” e “balance”) são privadas e não podem ser acessadas diretamente de fora. Em vez disso, a classe fornece métodos públicos ("getStatus()", "setStatus()", "getSaldo()", " getSaldo ()") para acessar e modificar esses dados. O uso de tipos de dados simples ("string" para "estado", "duplo" para "equilíbrio") promove um acoplamento fraco, eliminando a necessidade de compreender estruturas de dados complexas para interagir com as classes.

6.15 Acoplamento da classe SugestaoInvestimento

SugestaoInvestimento: O código da classe “SugestaoInvestimento” apresenta baixo acoplamento por utilizar encapsulamento de dados e exposição de método de acesso controlado. As variáveis de instância ("descrição", "valorInvestimento" e "retornoInvestimento") são privadas. E assim, a classe fornece métodos públicos para acessar e modificar esses dados: "getDescricao()", "setDescricao()", "getValorInvestimento()", "setValorInvestimento()", "getRetornoVestimento()", "setRetornoVestimento()") fornece. . . O uso de tipos de dados simples “String” para “descricao” e “double” para “valorInvestimento” e “retornoInvestimento”) promove um acoplamento fraco porque você não precisa entender estruturas de dados complexas para interagir com as classes.