POO

Plataforma de Gestão Financeira para PMEs

Nomes: Gabriel Gonçalves Brito

Guilherme Ferreira Sanguinete dos Santos

Professor: Carlos Veríssimo

São Paulo - SP

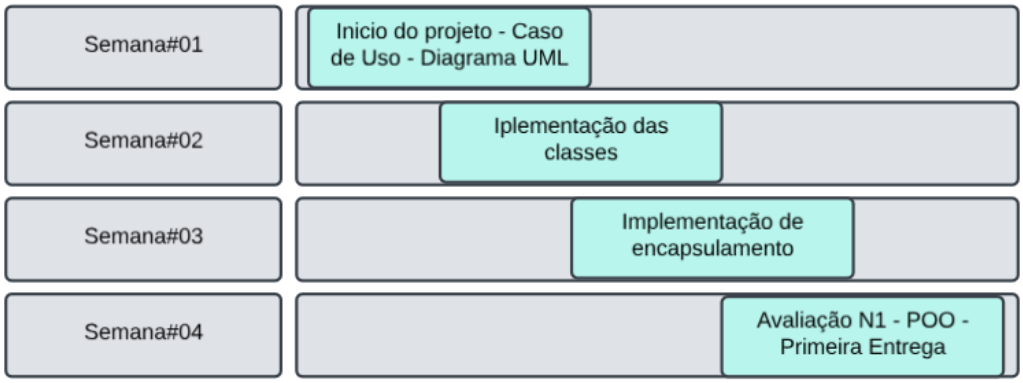
2023

Sumário

1. Cenário

À medida que a procura de soluções financeiras entre as pequenas e médias empresas (PME) continua a crescer, surgiram plataformas online inovadoras para satisfazer esta procura. A plataforma oferece recursos de gerenciamento financeiro, incluindo orçamento, fluxo de caixa, contabilidade simplificada, gerenciamento de clientes e fornecedores, consultoria financeira e educação.

2. Cronograma



3. 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.

4. Requisitos do Cliente

4.1. 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.

4.2. 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.

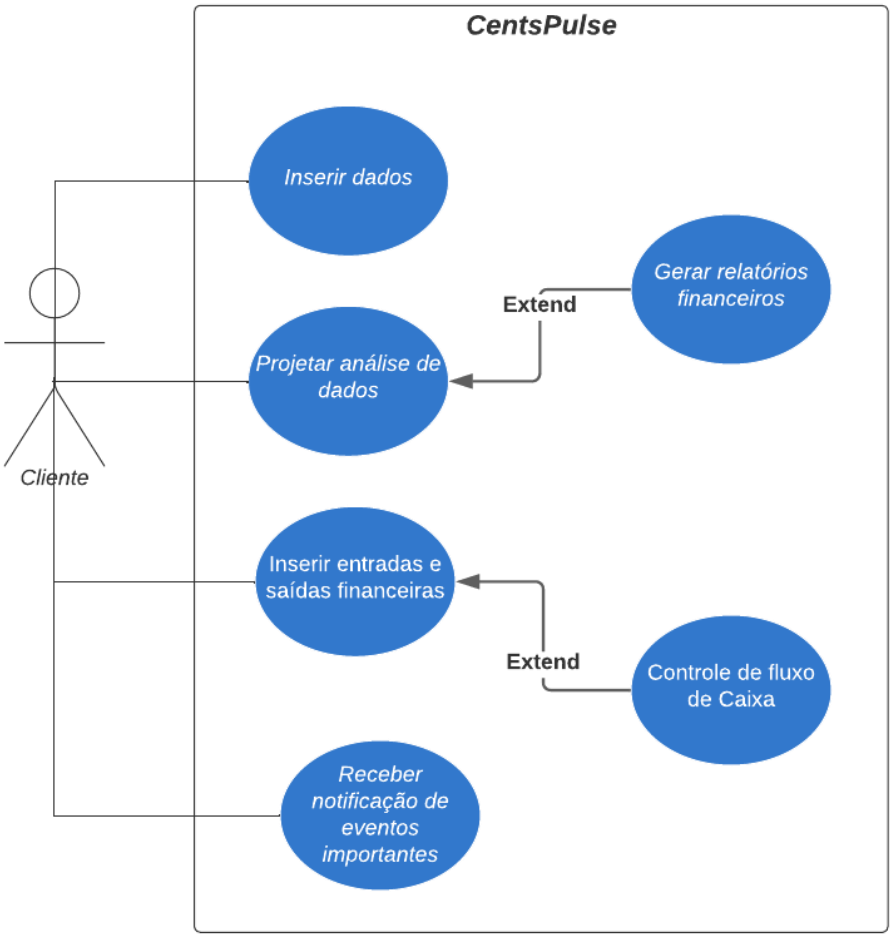
RNF7: O sistema deverá estar ativo 99,9% do tempo.

RNF8: Deverá fornecer formas de login;

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

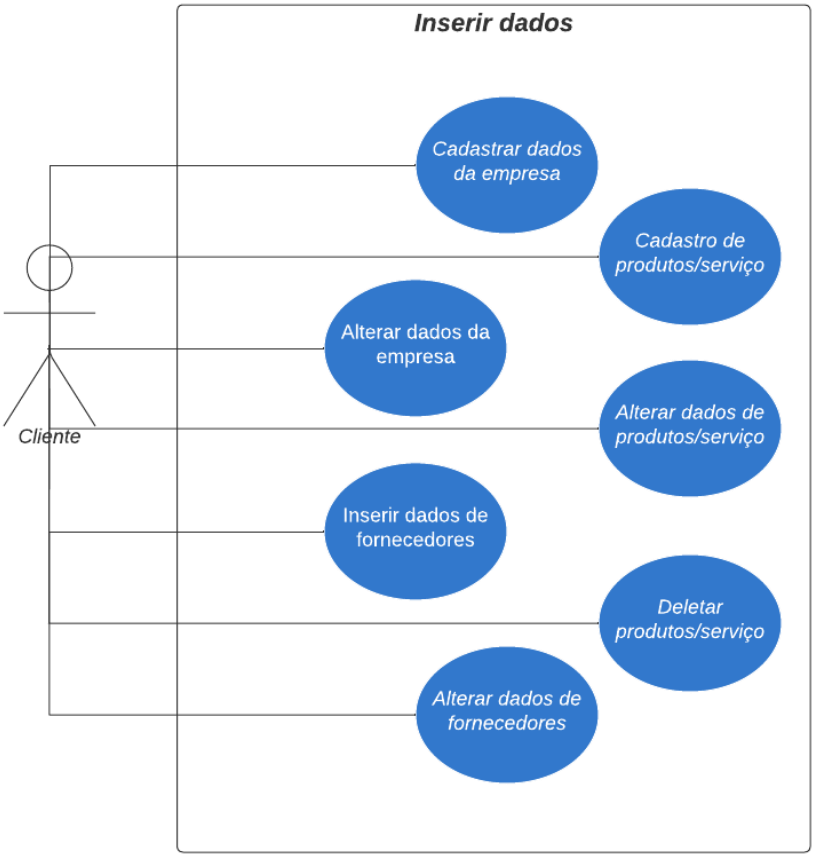
5. Casos de Uso

5.1 Diagrama de Caso de Uso - Visão Geral



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

5.2.1 Diagrama de Caso de Uso #1



5.2.2 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

5.2.3 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

5.2.4 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

5.2.5 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

5.2.6 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

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

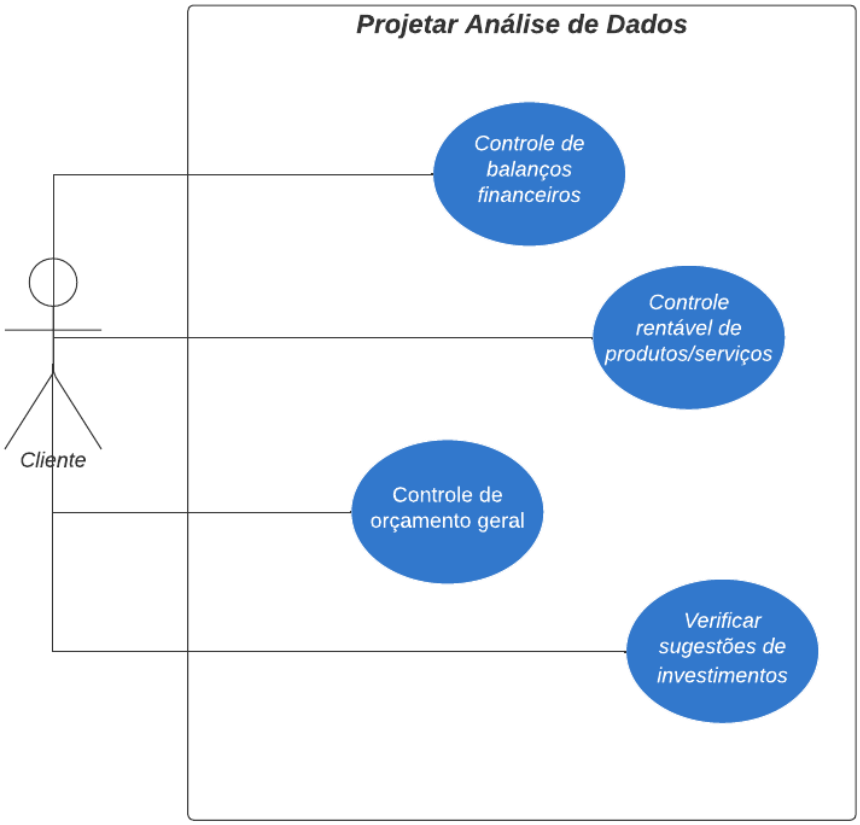
Nome do caso de uso:	1.6 Deletar produtos/serviços
Atores:	Cliente
Trigger:	Apagar produtos e serviços que não existem mais
Pré-requisito:	Tem que estar logado no sistema
Fluxo de eventos:	Cliente faz o login no sistema, clica no botão “Perfil”, e deleta os produtos e serviços que ele quiser

5.2.8 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

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

5.3.1 Diagrama de Caso de Uso #2



5.3.2 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”

5.3.3 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”

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

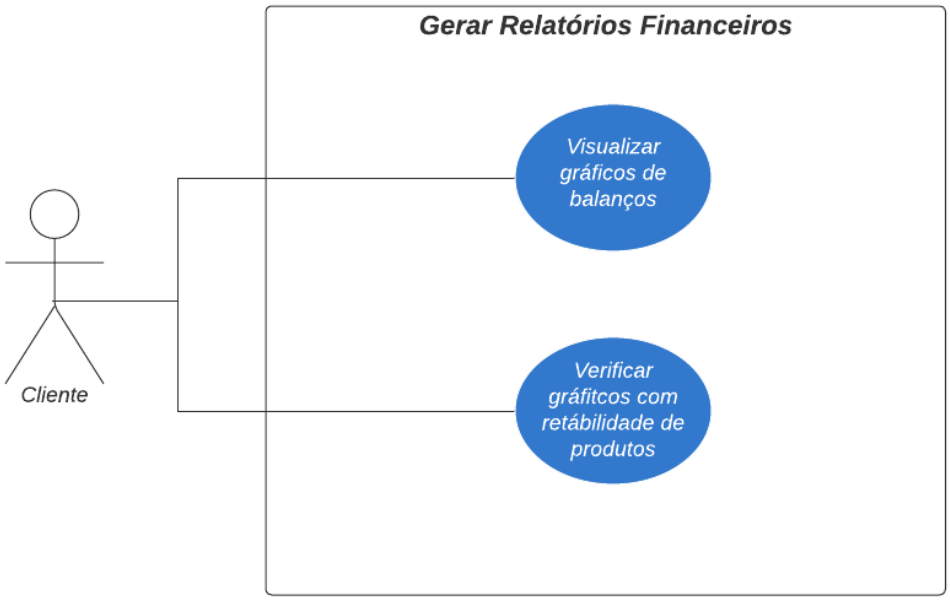
Nome do caso de uso:	2.3 Controle de orçamento geral
Atores:	Cliente
Trigger:	Ter o controle dos orçamentos da empresa
Pré-requisito:	Tem que estar logado no sistema
Fluxo de eventos:	Cliente faz o login no sistema, clica no botão “Análise”, e clica na opção “Orçamento geral”

5.3.5 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”

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

5.4.1 Diagrama de Caso de Uso #3



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

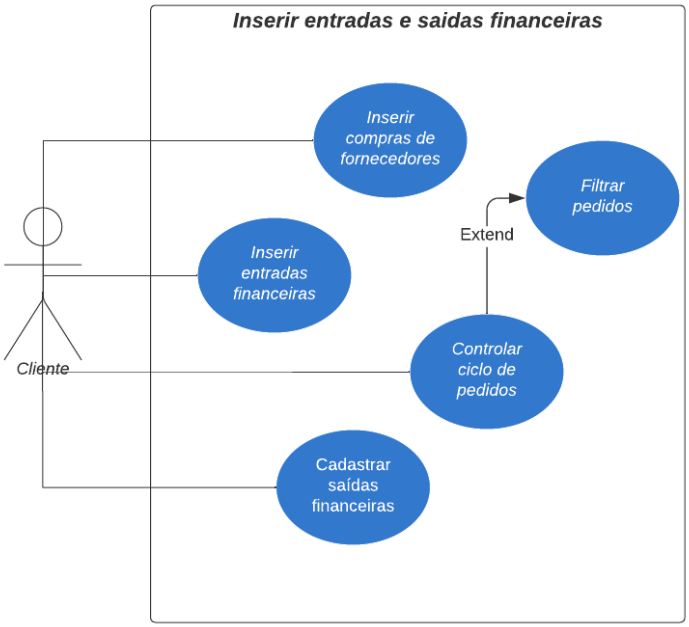
Nome do caso de uso:	3.1 Visualizar gráficos de balanços
Atores:	Cliente
Trigger:	Acompanhar balanços mensais, trimestrais e geral
Pré-requisito:	Tem que estar logado no sistema
Fluxo de eventos:	Cliente faz o login no sistema, clica no botão “Gerar relatórios financeiros”, e clica na opção “Gráficos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

5.4.3 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”

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

5.5.1 Diagrama de Caso de Uso #4



5.5.2 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”

5.5.3 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”

5.5.4 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”

5.5.5 Detalhamento do Caso de uso #4.4 – Filtar pedidos

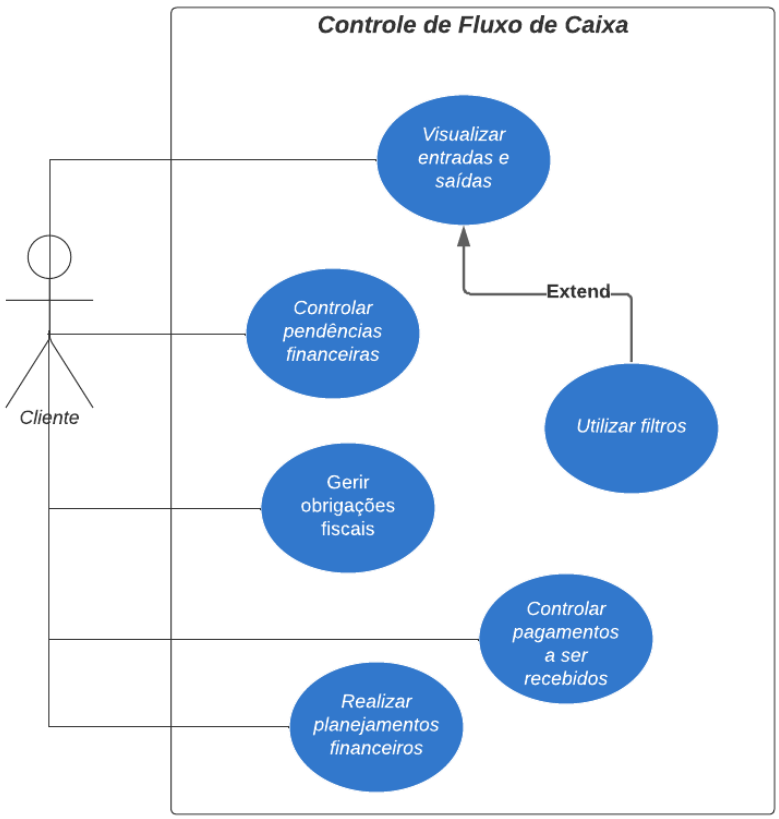
Nome do caso de uso:	4.4 Filtrar pedidos
Atores:	Cliente
Trigger:	Filtrar pedidos por estado e data
Pré-requisito:	Tem que estar logado no sistema
Fluxo de eventos:	Cliente faz o login no sistema, clica no botão “Inserir entradas e saídas financeiras”, clica na opção “Controle do ciclo de pedidos”, e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

5.5.6 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”

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

5.6.1 Diagrama de Caso de Uso #5



5.6.2 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”

5.6.3 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”

5.6.4 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”

5.6.5 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”

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

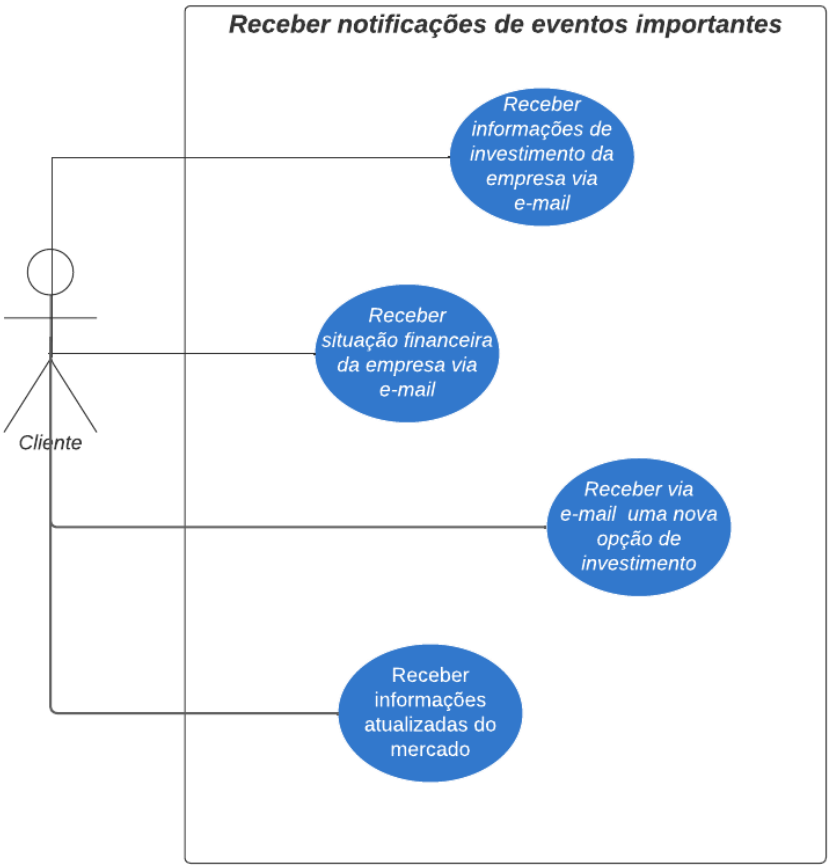
Nome do caso de uso:	5.3 Gerir obrigações fiscais
Atores:	Cliente
Trigger:	Fazer o planejamento financeiro para o futuro
Pré-requisito:	Tem que estar logado no sistema
Fluxo de eventos:	Cliente faz o login no sistema, clica no botão “Controle de fluxo de caixa”, e clica na opção “Planejamento financeiro”

5.6.7 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”

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

5.7.1 Diagrama de Caso de Uso #6



5.7.2 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”

5.7.3 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”

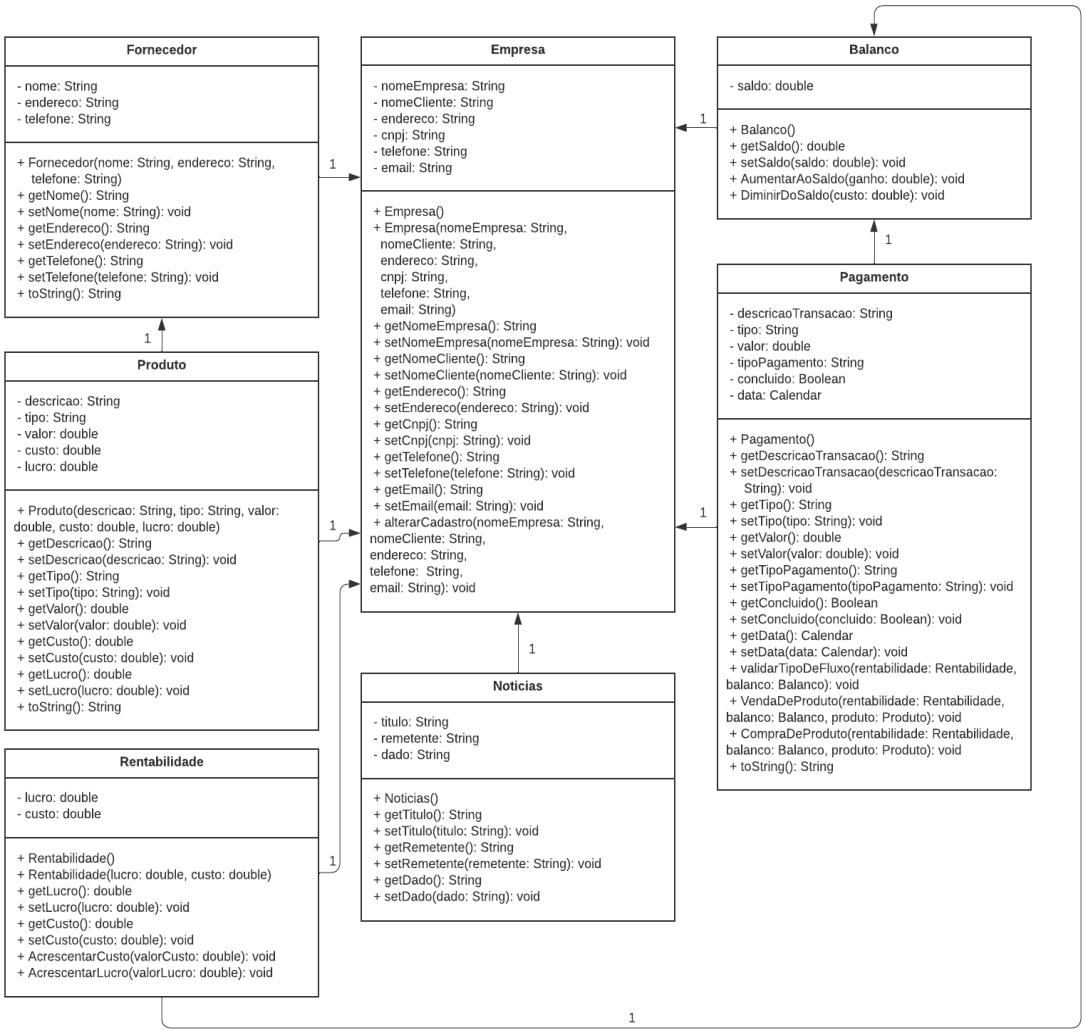
5.7.4 Detalhamento do Caso de uso #6.3 – Receber situação financeira da empresa

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”

5.7.5 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”

6. Diagrama de Classe



7. Encapsulamento

7.1 Encapsulamento da classe Balanco

```java
package app;

public class Balanco {
    private double saldo;

    public Balanco() {
        this.saldo = 0.0;

    }

    public double getSaldo() {
        return saldo;
    }

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

    public void AumentarAoSaldo(double ganho) {
        this.saldo =+ ganho;
    }

    public void DiminirDoSaldo(double custo) {
        this.saldo =- custo;
    }
}
```

7.2 Encapsulamento da classe Empresa

```
package app;

public class Empresa {
    private String nomeEmpresa;
    private String nomeCliente;
    private String endereco;
    private String cnpj;
    private String telefone;
    private String email;

    public Empresa() {
    }

    public Empresa(String nomeEmpresa, String nomeCliente, String
endereco, String cnpj, String telefone, String email) {
        this.nomeEmpresa = nomeEmpresa;
        this.nomeCliente = nomeCliente;
        this.endereco = endereco;
        this.cnpj = cnpj;
        this.telefone = telefone;
        this.email = email;
    }

    public String getNomeEmpresa() {
        return nomeEmpresa;
    }

    public void setNomeEmpresa(String nomeEmpresa) {
        this.nomeEmpresa = nomeEmpresa;
    }

    public String getNomeCliente() {
        return nomeCliente;
    }

    public void setNomeCliente(String nomeCliente) {
        this.nomeCliente = nomeCliente;
    }

    public String getEndereco() {
        return endereco;
    }

    public void setEndereco(String endereco) {
        this.endereco = endereco;
    }

    public String getCnpj() {
```

```
        return cnpj;
    }

    public void setCnpj(String cnpj) {
        this.cnpj = cnpj;
    }

    public String getTelefone() {
        return telefone;
    }

    public void setTelefone(String telefone) {
        this.telefone = telefone;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public void alterarCadastro(String nomeEmpresa, String nomeCliente,
String endereco, String telefone, String email) {
        this.nomeEmpresa = nomeEmpresa;
        this.nomeCliente = nomeCliente;
        this.endereco = endereco;
        this.telefone = telefone;
        this.email = email;
    }
}
```

7.3 Encapsulamento da classe Fornecedor

```
package app;

public class Fornecedor {
    private String nome;
    private String endereco;
    private String telefone;

    public Fornecedor(String nome, String endereco, String telefone) {
        this.nome = nome;
        this.endereco = endereco;
        this.telefone = telefone;
    }

    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;
    }

    @Override
    public String toString() {
        return "Fornecedor: " + this.nome + " | Telefone: " +
this.telefone + " | Endereço: " + this.endereco;
    }
}
```

7.4 Encapsulamento da classe Noticias

```java
package app;

public class Noticias {
    private String titulo;
    private String remetente;
    private String dado;

    public Noticias() {
    }
    public String getTitulo() {
        return titulo;
    }
    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }
    public String getRemetente() {
        return remetente;
    }
    public void setRemetente(String remetente) {
        this.remetente = remetente;
    }
    public String getDado() {
        return dado;
    }
    public void setDado(String dado) {
        this.dado = dado;
    }
}
```

7.5 Encapsulamento da classe Pagamento

```java
package app;

import java.util.Calendar;

public class Pagamento {
    private String descricaoTransacao;
    private String tipo;
    private double valor;
    private String tipoPagamento;
    private Boolean concluido;
    private Calendar data;

    public Pagamento() {
    }

    public String getDescricaoTransacao() {
        return descricaoTransacao;
    }
    public void setDescricaoTransacao(String descricaoTransacao) {
        this.descricaoTransacao = descricaoTransacao;
    }
    public String getTipo() {
        return tipo;
    }
    public void setTipo(String tipo) {
        this.tipo = tipo;
    }
    public double getValor() {
        return valor;
    }
    public void setValor(double valor) {
        this.valor = valor;
    }
    public String getTipoPagamento() {
        return tipoPagamento;
    }

    public void setTipoPagamento(String tipoPagamento) {
        this.tipoPagamento = tipoPagamento;
    }

    public Boolean getConcluido() {
        return concluido;
    }

    public void setConcluido(Boolean concluido) {
        this.concluido = concluido;
    }
```

```
    public Calendar getData() {
        return data;
    }

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

    public void validarTipoDeFluxo(Rentabilidade rentabilidade, Balanco
balanco) throws Exception{
        if(this.concluido == true) {
            if(tipo.equals("Entrada")) {
                balanco.AumentarAoSaldo(valor);
                rentabilidade.AcrecentarLucro(valor);
                }
            else if(tipo.equals("Saída")) {
                balanco.DiminirDoSaldo(valor);
                rentabilidade.AcrecentarCusto(valor);
                }
            else {
                throw new Exception("O tipo de entrada está inválido");
                }
        }
    }
    public void VendaDeProduto(Rentabilidade rentabilidade, Balanco
balanco, Produto produto) throws Exception{
        validarTipoDeFluxo(rentabilidade, balanco);
    }
    public void CompraDeProduto(Rentabilidade rentabilidade, Balanco
balanco, Produto produto) throws Exception{
        validarTipoDeFluxo(rentabilidade, balanco);
    }

    @Override
    public String toString() {
        String conclusao;
        if(concluido == true)
            conclusao = "Sim";
        else
            conclusao = "Não";

        return "Descrição: " + descricaoTransacao + " | Tipo: " + tipo +
" | Valor: " + valor
                + "Tipo =" + tipoPagamento + " Concluido? " + conclusao +
" Data : " + data + "]";
    }
}
```

7.6 Encapsulamento da classe Produto

```
package app;

public class Produto {
    private String descricao;
    private String tipo;
    private double valor;
    private double custo;
    private double lucro;


    public Produto(String descricao, String tipo, double valor, double
custo, double lucro) {
        this.descricao = descricao;
        this.tipo = tipo;
        this.valor = valor;
        this.custo = custo;
        this.lucro = lucro;
    }

    public String getDescricao() {
        return descricao;
    }

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

    public String getTipo() {
        return tipo;
    }

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

    public double getValor() {
        return valor;
    }

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

    public double getCusto() {
        return custo;
    }

    public void setCusto(double custo) {
```

```
        this.custo = custo;
    }

    public double getLucro() {
        return lucro;
    }

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

7.7 Encapsulamento da classe Rentabilidade

```java
package app;

public class Rentabilidade {
    private double lucro;
    private double custo;

    public Rentabilidade() {
    }

    public Rentabilidade(double lucro, double custo) {
        this.lucro = lucro;
        this.custo = custo;
    }

    public double getLucro() {
        return lucro;
    }

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

    public double getCusto() {
        return custo;
    }

    public void setCusto(double custo) {
        this.custo = custo;
    }

    public void AcrecentarCusto (double valorCusto) {
        this.custo =+ valorCusto;
    }

    public void AcrecentarLucro (double valorLucro) {
        this.custo =+ valorLucro;
    }
}
```

8. Baixo Acoplamento das classes

8.1 Acoplamento da classe Balanco

Neste código, a classe “Balanco” possui métodos simples para manipular o saldo (“AumentarAoSaldo” e “DiminirDoSaldo”) e para acessar o saldo atual (“getSaldo”). Ela não possui dependências externas complexas ou conexões diretas com outras classes, o que a torna isolada e fácil de entender e manter.

8.2 Acoplamento da classe Empresa

A classe “Empresa” não depende de outras classes para executar suas funções principais. Ela possui métodos simples para acessar e modificar os dados dos objetos (“get” e “set”), sem depender de funcionalidades complexas de outras classes ou módulos do sistema.

Além disso, a classe “Empresa” não contém referências diretas a outras classes ou detalhes de implementação de funcionalidades que poderiam aumentar seu acoplamento. A única operação que ela executa além de obter e modificar seus atributos é a operação “alterarCadastro”, que ainda é localizada na própria classe e não envolve interações complexas com outras partes do sistema.

8.3 Acoplamento da classe Fornecedor

A classe “Fornecedor” demonstra um baixo acoplamento, pois a classe é coesa e tem uma única responsabilidade: gerenciar os dados de um fornecedor. Ela contém apenas métodos básicos para acessar e modificar os atributos (“get” e “set”), sem depender de funcionalidades complexas de outras classes ou módulos do sistema.

Essa independência funcional permite que a classe “Fornecedor” seja facilmente modificada sem afetar outras partes do sistema. Se houver mudanças nos requisitos relacionados às informações do fornecedor, essas mudanças podem ser feitas na classe “Fornecedor” sem causar impacto em outras classes. Isso promove a flexibilidade e a manutenibilidade do código, tornando-o mais fácil de entender, modificar e expandir no futuro.

8.4 Acoplamento da classe Noticias

A classe “Noticias” demonstra um baixo acoplamento, pois a classe é altamente coesa e possui uma única responsabilidade: representar informações relacionadas a notícias. Ela contém apenas métodos básicos para acessar e modificar os atributos (“get” e “set”), sem depender de funcionalidades complexas de outras classes ou módulos do sistema.

Essa independência funcional permite que a classe “Noticias” seja facilmente modificada sem afetar outras partes do sistema. Se houver mudanças nos requisitos relacionados às informações da notícia, essas mudanças podem ser feitas na classe “Noticias” sem causar impacto em outras classes. Isso promove a flexibilidade e a manutenibilidade do código, tornando-o mais fácil de entender, modificar e expandir no futuro.

8.5 Acoplamento da classe Pagamento

A classe “Pagamento” ilustra um baixo acoplamento devido à sua independência e foco em uma única responsabilidade: gerenciar transações financeiras. A classe Pagamento não tem acoplamento forte com outras partes do sistema, pois não possui dependências complexas com outras classes. Ela simplesmente aceita instâncias de “Rentabilidade” e “Balanco” como parâmetros em alguns métodos, mas não tem conhecimento detalhado de sua implementação interna.

Os métodos “validarTipoDeFluxo()”, “VendaDeProduto()” e “CompraDeProduto()” são designados para validar e executar operações com base no tipo de transação, mas eles apenas chamam métodos correspondentes nas instâncias de “Rentabilidade” e “Balanco”. A classe “Pagamento” não está ciente da lógica específica desses métodos; ela apenas delega a responsabilidade para as instâncias de “Rentabilidade” e “Balanco”, mantendo assim um baixo acoplamento.

Além disso, o código não possui referências diretas a outras classes fora do seu escopo (“Rentabilidade”, “Balanco” e “Produto”). As variáveis e métodos são acessados através de parâmetros ou objetos locais, o que também contribui para a redução do acoplamento.

8.6 Acoplamento da classe Produto

Contém apenas métodos básicos para acessar e modificar seus atributos (“get” e “set”), sem depender de funcionalidades complexas de outras classes ou módulos do sistema.

A classe “Produto” não possui referências diretas a outras classes ou detalhes de implementação que poderiam aumentar seu acoplamento. Ela não contém lógica de negócios complexa ou interações intrincadas com outras partes do sistema. Em vez disso, ela encapsula os dados do produto e fornece métodos simples para acessar e modificar esses dados.

8.7 Acoplamento da classe Rentabilidade

A classe “Rentabilidade” demonstra um baixo acoplamento, pois a classe é altamente coesa e mantém uma única responsabilidade: gerenciar dados relacionados à rentabilidade, incluindo lucro e custos. A classe “Rentabilidade” não possui dependências complexas com outras classes ou módulos do sistema. Ela apenas contém métodos básicos para acessar e modificar seus atributos (“get” e “set”), bem como métodos para adicionar valores aos atributos lucro e custo.

Então, a classe “Rentabilidade” não tem referências diretas a outras classes ou detalhes de implementação que poderiam aumentar seu acoplamento. As operações dentro dos métodos “AcrecentarCusto” e “AcrecentarLucro” não envolvem interações complexas com outras partes do sistema. Ela aceita valores como parâmetros e os utiliza localmente para atualizar os atributos da classe.

9. Conclusão

Neste contexto, foi desenvolvida uma plataforma inovadora para responder à crescente procura de soluções financeiras dirigidas às pequenas e médias empresas (PME). A plataforma oferece uma ampla gama de recursos de gestão financeira, desde o registro de transações e controle do fluxo de caixa até a geração de relatórios e cumprimento de regulamentações fiscais.

Ao analisar as classes do sistema, encontramos acoplamento fraco, o que significa que cada classe mantém responsabilidades específicas e independência funcional. Isso simplifica a manutenção e a extensão do código porque as alterações em uma classe não afetam negativamente outras partes do sistema. Além disso, a plataforma foi projetada para ser altamente coesa, garantindo que cada classe esteja focada em uma única função, tornando o código mais legível e fácil de entender.

Resumindo, a plataforma atende às necessidades financeiras exclusivas das pequenas e médias empresas, fornecendo uma arquitetura de software flexível e eficiente. Esta abordagem garante que a empresa possa gerir as suas finanças de forma eficaz e adaptar-se às novas circunstâncias empresariais. Com funcionalidade confiável e design de código bem estruturado, a plataforma está bem posicionada para apoiar o sucesso financeiro de pequenas e médias empresas.