UFF - UNIVERSIDADE FEDERAL FLUMINENSE
CIÊNCIA DA COMPUTAÇÃO
PROJETO DE SOFTWARE

IGOR QUINTES DOS SANTOS
JOÃO VICTOR LAGOS DE AGUIAR
MATHEUS LOPES CARAPINA DO NASCIMENTO
PEDRO RIBEIRO FILHO
VITOR VIEIRA GOIS

DOCUMENTO DE ARQUITETURA:

Gerenciador de Finanças Pessoais

Niterói
2024

Universidade Federal Fluminense - UFF
Ciência da Computação
Projeto de Software 2024.1

METODOLOGIA DE DESENVOLVIMENTO DE SOFTWARE

Documento de Arquitetura de Software

Objetivo deste Documento

Este documento tem como objetivo descrever as principais decisões de projeto tomadas pela equipe de desenvolvimento e os critérios considerados durante a tomada destas decisões.

Histórico de Produção

Autor	Atribuições
Igor Quintes dos Santos	• Produção do presente documento (.docx) • Definição dos Requisitos Não Funcionais (RNF1 ao RNF11) • Algumas visões arquiteturais: DSS (Manter Metas Financeiras), Comunicação (Adicionar Metas) • Produção da aplicação dos princípios SOLID e GRASP neste documento juntamente com Matheus Lopes Carapina • Finalização na definição das LPS e construção do gráfico
João Victor Lagos de Aguiar	• Produção do presente documento (.docx) • Introdução e Escopo do Sistema • Definição dos Requisitos Funcionais (RF1 ao RF9) • Alguma visões arquiteturais: Caso de Uso, DSS (Login, Receber Notificação e Visualizar Extrato), Comunicação (Inserir Credenciais e Adicionar Transação), Atividades (Login) e Estado (Manter Conta Bancária) • Criação do tópico sobre GoF • Produção da LPS
Matheus Lopes Carapina do Nascimento	• Produção do presente documento (.docx) • Definição e produção do Diagrama de Classe Detalhado • Produção da aplicação dos princípios SOLID e GRASP neste documento juntamente com Igor Quintes dos Santos
Pedro Ribeiro Filho	• Ajudou e auxiliou na validação do Caso de Uso do Sistema • Definiu a linguagem de programação e frameworks ideais para a criação do sistema, antecipadamente • Alguma visões arquiteturais: DSS (Adicionar transação, cadastrar, categorizar registros, gerenciar conta e integrar conta)
Vitor Vieira Gois	• Produção deste documento (.docx) • Objetivos Arquiteturais • Restrições Arquiteturais • Escolha e explanação dos Padrões Arquiteturais (Microsserviços e Arquitetura Orientada por Mensagens) • Algumas visões arquiteturais: Modelo Conceitual, DSS (Manter e Integrar Contas Bancárias e Manter Despesas/Receitas), Comunicação (Integrar/Manter Conta e Visualizar Extrato)

SUMÁRIO

1. INTRODUÇÃO.....5
1.2. CONTEXTUALIZAÇÃO.....5
2. DESCRIÇÃO DO ESCOPO DO SISTEMA.....6
3. INFORMAÇÕES ARQUITETURAIS.....8
3.1. REQUISITOS ARQUITETURAIS.....8
3.2. OBJETIVOS ARQUITETURAIS.....10
3.3. RESTRIÇÕES ARQUITETURAIS.....10
3.4. PADRÕES ARQUITETURAIS.....11
4. VISÕES ARQUITETURAIS.....13
4.1. CASO DE USO.....13
4.2. MODELO CONCEITUAL.....14
4.3. DIAGRAMAS DE SEQUÊNCIA DE SISTEMA.....15
4.3.1. DSS: LOGIN.....15
4.3.2. DSS: MANTER E INTEGRAR CONTAS BANCÁRIAS.....16
4.3.3. DSS: MANTER DESPESAS/RECEITAS.....16
4.3.4. DSS: RECEBER NOTIFICAÇÃO.....17
4.3.5. DSS: VISUALIZAR EXTRATO.....17
4.3.6. DSS: MANTER METAS FINANCEIRAS.....18
4.3.7. DSS: ADICIONAR TRANSAÇÃO.....18
4.3.8. DSS: CADASTRAR.....19
4.3.9. DSS: CATEGORIZAR REGISTROS.....19
4.3.10. DSS: GERENCIAR CONTA.....20
4.3.11. DSS: INTEGRAR CONTA.....20
4.4. DIAGRAMAS DE INTERAÇÃO (COMUNICAÇÃO).....21
4.4.1. INSERIR CREDENCIAIS.....21
4.4.2. ADICIONAR TRANSAÇÃO.....22
4.4.3. INTEGRAR/MANTER CONTA.....23
4.4.4. VISUALIZAR EXTRATO.....23
4.4.5. ADICIONAR METAS.....23
4.5. DIAGRAMAS DE ESTADO OU ATIVIDADES.....24
4.5.1. DIAGRAMA DE ATIVIDADES: LOGIN.....25
4.5.2. DIAGRAMA DE ESTADO: MANTER CONTA BANCÁRIA.....25
4.6. DIAGRAMA DE CLASSE DETALHADO.....26
5. APLICAÇÃO DOS PRINCÍPIOS SOLID.....27
5.1. RESPONSABILIDADE ÚNICA (SRP).....27
5.2. ABERTO-FECHADO (OCP).....27
5.3. SEGREGAÇÃO DE INTERFACE (ISP).....27
6. APLICAÇÃO DOS PADRÕES GRASP.....29

6.1. ESPECIALISTA..... 29
6.2. CRIADOR..... 29
6.3. ACOPLAMENTO FRACO..... 30
6.4. COESÃO ALTA..... 30
6.5. CONTROLADOR..... 31
7. PADRÕES GOF..... 32
7.1. SINGLETON..... 32
7.2. TEMPLATE METHOD..... 32
7.3. BUILDER..... 34
8. LINHA DE PRODUTO DE SOFTWARE (LPS)..... 36

1. INTRODUÇÃO

O presente documento visa apresentar uma visão abrangente do sistema Gerenciador de Finanças Pessoais desenvolvido pelo grupo *VIPJM Moedinhas Malucas*, antiga [EM DEFINIÇÃO…], da disciplina de Projeto de Software, cursada no semestre 2024.1 do curso de Ciência da Computação, na Universidade Federal Fluminense (UFF). O referido grupo é composto por cinco integrantes, são eles: Igor Quintes dos Santos, João Victor Lagos de Aguiar, Matheus Lopes Carapina do Nascimento, Pedro Ribeiro Filho e Vitor Vieira Gois.

Este documento também visa delinear principais funcionalidades, escopo do sistema, requisitos arquiteturais do sistema, padrões arquiteturais, objetivos e restrições da arquitetura, bem como diagramas de casos de usos, modelo conceitual, diagramas de sequência do sistema e entre outros, conferindo assim uma qualidade maior na produção do software.

Por meio deste documento, espera-se que os integrantes do grupo possam transmitir claramente o escopo e os objetivos do projeto. Ademais, este documento servirá como um guia de referência durante as etapas de desenvolvimento e criação do sistema em questão e dos diagramas.

O link referente a este trabalho (projeto), no repositório do GitHub, é: https://github.com/vaniacourses/trabalho-pr-tico-vipjm-moedinhas-malucas.
**Observação: Este repositório é privado, portanto pode haver peculiaridades no acesso.*

1.2. CONTEXTUALIZAÇÃO

No mundo contemporâneo, o gerenciamento eficiente das finanças pessoais é fundamental para garantir a estabilidade financeira e o alcance de objetivos financeiros de curto, médio e longo prazo. Com a crescente complexidade das transações financeiras e a diversidade de fontes de renda e despesas, torna-se imperativo contar com ferramentas tecnológicas capazes de oferecer suporte na organização e controle das finanças individuais. Nesse contexto, surge a necessidade de desenvolvimento de um sistema Gerenciador de Finanças Pessoais, uma aplicação que visa facilitar o acompanhamento e a gestão das receitas, despesas e investimentos de um indivíduo.

2. DESCRIÇÃO DO ESCOPO DO SISTEMA

O sistema Gerenciador de Finanças Pessoais é uma aplicação de software projetada para ajudar/auxiliar os usuários na gestão eficaz e correta de suas finanças pessoais. O sistema permitirá aos usuários registrar e monitorar suas receitas e despesas, categorizando cada transação de acordo com sua natureza (ex: alimentação, transporte, moradia, lazer, etc.). Além disso, possibilitará o acompanhamento do saldo disponível, a definição de metas financeiras e a análise do histórico de transações.

Com isso, suas principais funções incluem, em um rol exemplificativo:

- **Registro de Receitas e Despesas**: Os usuários terão a capacidade de registrar diversas transações recorrentes, abrangendo tanto receitas quanto despesas. Este recurso visa fornecer uma visão abrangente das finanças pessoais do usuário, permitindo um acompanhamento detalhado de todas as fontes de entrada e saída de recursos financeiros.
  Algumas das transações que podem ser registradas incluem:
  - **Receitas:**
    - Salário mensal
    - Renda extra (por exemplo, freelancers, vendas, investimentos)
    - Entre outras fontes de renda
  - **Despesas:**
    - Contas de serviços públicos (água, luz, gás, etc)
    - Aluguel ou prestação da casa
    - Alimentação (supermercado)
    - Transporte (combustível, transporte público)
    - Educação (mensalidade)
    - Plano de saúde
    - Entre outras despesas

- **Categorização dos Registros:** O usuário poderá categorizar os registros, tanto de despesas como de receitas com base em padrões previamente estabelecidos, facilitando o processo de organização financeira. Essas categorias podem abranger uma variedade de áreas, como visto acima. Com isso, o usuário poderá acompanhar e analisar os seus gastos e ganhos.

- **Análise de Desempenho Financeiro:** O sistema fornecerá gráficos e relatórios detalhados sobre fluxo de caixa, hábitos de consumo, receitas e despesas, a fim de que os usuários possam identificar áreas de melhoria e tomar decisões financeiras mais informadas.

- **Integração com Contas Bancárias:** O sistema poderá integrar as contas

bancárias dos usuários, permitindo a importação automática de informações financeiras e facilitando a reconciliação de dados. Um exemplo da utilidade dessa função é que caso o usuário tenha sua conta bancária vinculada ao aplicativo, ele não precisará colocar a informação do seu salário, rendimentos de aplicações financeiras, mensalidade de escola ou plano de saúde e entre outros, pois o sistema irá identificar esses dados e importará para o aplicativo.

- **Notificações Financeiras:** Os usuários poderão ser informados pelo sistema sobre transações importantes, vencimentos de contas e outras informações relevantes para sua saúde financeira.

3. INFORMAÇÕES ARQUITETURAIS

Nesta seção será detalhado todos os aspectos relacionados à arquitetura do sistema. Serão abordados os requisitos arquiteturais (requisitos funcionais e requisitos não funcionais), objetivo e restrições da arquitetura.

Ao identificar e documentar os requisitos arquiteturais do sistema, podemos entender melhor as necessidades específicas que influenciam a arquitetura. Os objetivos e restrições da arquitetura fornecem uma estrutura para avaliar e justificar as decisões arquiteturais, garantindo que o sistema atenda aos requisitos funcionais e não funcionais, enquanto cumpre limitações e considerações importantes.

3.1. REQUISITOS ARQUITETURAIS

Os requisitos arquiteturais representam as necessidades específicas que moldam a arquitetura do referido sistema. Estes requisitos abrangem os aspectos funcionais e os não funcionais e desempenham um papel importante na estrutura e no comportamento do software.

Os requisitos funcionais delineiam as capacidades que o sistema deve ter, descrevendo as funcionalidades específicas que devem ser implementadas para atender às necessidades dos usuários e às exigências do domínio do problema. Já os requisitos não funcionais referem-se a características ou qualidades do sistema. No contexto deste software, consideramos aspectos como segurança, eficiência, portabilidade e confiabilidade, entre outros, a serem considerados.

Requisitos Funcionais:

- **RF1:** O sistema deverá permitir que o cliente cadastre e faça o respectivo login da sua conta no aplicativo.
- **RF2:** O sistema deverá permitir ao usuário gerenciar/manter o seu perfil, podendo configurá-lo da maneira que desejar.
- **RF3:** O sistema deverá permitir que o cliente crie várias contas e gerencie/mantenha as respectivas contas (CRUD das contas).
- **RF4:** O sistema deverá permitir ao usuário cadastrar as despesas e receitas em sua conta.
- **RF5:** O sistema deverá permitir que o usuário categorize os respectivos registros (receitas e despesas) por especialização, como alimentação, educação, transporte e entre outros.
- **RF6:** O sistema deverá permitir ao usuário visualizar o seu extrato e, a partir dele, a possibilidade de gerar um relatório financeiro, que será disponibilizado pelo sistema, referente ao respectivo extrato.

- **RF7:** O sistema deverá permitir ao usuário a possibilidade de definir metas financeiras.
- **RF8:** O sistema deverá permitir que o usuário possa receber notificações financeiras a respeito das suas contas.
- **RF9:** O sistema deverá permitir a integração de contas bancárias por parte dos usuários, permitindo assim a importação financeira automática e facilitando a reconciliação de dados.

Requisitos Não Funcionais:

Segurança

- **RNF1:** O sistema deve assegurar que as contas dos usuários sejam completamente protegidas contra acessos não autorizados.
- **RNF2:** O sistema deve incluir protocolos de criptografía para proteger todas as comunicações entre o servidor e o usuário.
- **RNF3:** O sistema deve ser capaz de registrar e rastrear todas as ações realizadas pelos usuários dentro da plataforma.

Eficiência

- **RNF4:** O sistema deve otimizar consultas ao banco de dados para garantir tempos de resposta rápidos, especialmente durante períodos de alto tráfego.
- **RNF5:** O sistema deve minimizar o consumo de recursos, como uso de CPU e memória, para garantir que o aplicativo seja executado suavemente em dispositivos com diferentes especificações.
- **RNF6:** O sistema deve garantir que o tempo de resposta para as interações do usuário seja mantido dentro de um limite máximo de 2 segundos.

Portabilidade

- **RNF7:** O sistema deve ser programado na linguagem Ruby.
- **RNF8:** O sistema deve implementar o Framework Ruby on Rails, que possui o padrão MVC e uma vasta gama de bibliotecas para desenvolvimento ágil e seguro de aplicações web.
- **RNF9**: O sistema deve garantir que a interface do usuário seja responsiva e se adapte automaticamente a diferentes tamanhos de tela e orientações de dispositivo.
- **RNF10:** O sistema deve ser focado, inicialmente, para a versão web.

Confiabilidade

- **RNF11:** O sistema deve implementar backups regulares dos dados dos usuários para evitar perda de informações importantes.
- **RNF12:** O sistema deve estabelecer um sistema de monitoramento contínuo para detectar e corrigir rapidamente problemas de desempenho ou falhas.

Manutenibilidade

- **RNF13:** O sistema deve ser desenvolvido com código limpo e bem documentado para facilitar a manutenção e aprimoramentos futuros.

3.2. OBJETIVOS ARQUITETURAIS

Os objetivos da arquitetura representam as metas que são buscadas alcançar por meio da implementação da arquitetura do sistema. Com isso em mente, segue os objetivos desejados ao término da implementação.

O primeiro deles é um sistema que consiga atender às necessidades dos usuários de forma abrangente e que consiga atender pessoas de diferentes níveis financeiros ou de conhecimento técnico de forma que se consiga garantir um ambiente financeiro mais saudável em suas vidas pessoais.

Outro objetivo seria permitir de forma mais eficiente a visualização do estado financeiro do usuário, sem precisar acessar diversos programas e aplicativos bancários que muitas vezes demoram e não são eficientes devido à alta demanda de usuários acessando o sistema ou por questões de hardware do próprio usuário.

Além disso, o sistema deve conseguir unificar dados de diferentes contas financeiras de forma segura que evite invasões e, em caso de tentativas dessas invasões, tenha a capacidade de impedir o vazamento de dados desses usuários.

E para terminar, o sistema deve ser escalável de forma que se possa inserir mais funcionalidades, usuários e transações no sistema sem comprometer o desempenho e a confiabilidade.

3.3. RESTRIÇÕES ARQUITETURAIS

As restrições da arquitetura envolvem os limites impostos ao projeto e influenciam nas escolhas arquiteturais. Os mais fáceis a serem observados são o prazo para a implementação do sistema e os recursos limitados que se tem em hardware e equipe e que criam como consequência a busca por maneiras mais rápidas e fáceis de se adicionar as funcionalidades e relacioná-las umas às outras.

Outra restrição seria a capacidade de se obter as informações bancárias de um usuário sem acesso aos dados de forma direta aos dados dentro da agência financeira e de forma que o sistema não possa ser utilizado como um auxílio para um possível invasor consiga acessar e manipular o dinheiro do usuário guardado nas contas bancárias inseridas no sistema.

A maturidade do grupo também é um ponto a se pensar. Em discussões percebe-se que a experiência em se produzir um gerenciador de finanças não é

forte e que possivelmente novos conhecimentos técnicos de linguagens e formas de como se organizar o código vão ser aprendidos no momento da implementação, desacelerando o progresso de criação do sistema.

E por fim, como esse sistema poderia ser utilizado em diferentes dispositivos com desempenhos e sistemas operacionais diferentes. Há de se pensar então na implementação do sistema com formas de ser facilmente eficiente e portável caso um novo ambiente ou atualizações de ambientes apareçam.

3.4. PADRÕES ARQUITETURAIS

Padrões arquiteturais são organizações estruturais de sistemas com algum grau de semelhança e que podem ser usados para delimitar restrições, organizar módulos e definir suas relações em contextos parecidos. Ou seja, orientam a solução de uma família de problemas. Dentro desse sistema, o padrão escolhido foi o Model-View-Controller (MVC).

Model-View-Controller (MVC): Este padrão é amplamente utilizado em aplicações web e é nativamente suportado pelo framework que será utilizado para desenvolvimento da aplicação, o Ruby on Rails. O MVC separa a aplicação em três componentes principais:

1. **Model (Modelo):** Responsável pela lógica de negócios e pela comunicação com o banco de dados. No Ruby on Rails, isso é implementado utilizando Active Record, facilitando a manipulação de dados.

2. **View (Visualização):** Encapsula a lógica de apresentação e é responsável por gerar a interface do usuário. Em Rails, as Views são tipicamentes arquivos HTML com integração Ruby (ERB), que permite a inserção de código Ruby dentro do HTML.

3. **Controller (Controlador):** Atua como intermediário entre o model e o view. Ele recebe as requisições do usuário, processa os dados por meio dos modelos e seleciona a view apropriada para renderizar a resposta.

Essa separação de responsabilidade traz diversos benefícios para o desenvolvimento e manutenção do sistema, incluindo:

- **Facilidade de Manutenção e Evolução:** Com cada componente isolado em sua própria função, é mais fácil realizar alterações ou adicionar novas funcionalidades sem impactar outras partes do sistema.

- **Reutilização de Código:** A lógica de negócios encapsulada nos modelos pode ser reutilizada em diferentes partes da aplicação, promovendo a reutilização de código.

- **Testabilidade:** A separação clara entre modelo, visualização e controladores facilita a escrita de testes unitários e de integração, garantindo a qualidade do software.

- **Desempenho e Escalabilidade:** Embora o padrão MVC não separe a aplicação em serviços independentes como em uma arquitetura de microservices, ele permite a organização eficiente do código e a utilização de técnicas de caching e otimização para melhorar o desempenho. Além disso, o Rails oferece suporte a escalabilidade horizontal, permitindo que a aplicação cresça conforme a demanda.

4. VISÕES ARQUITETURAIS

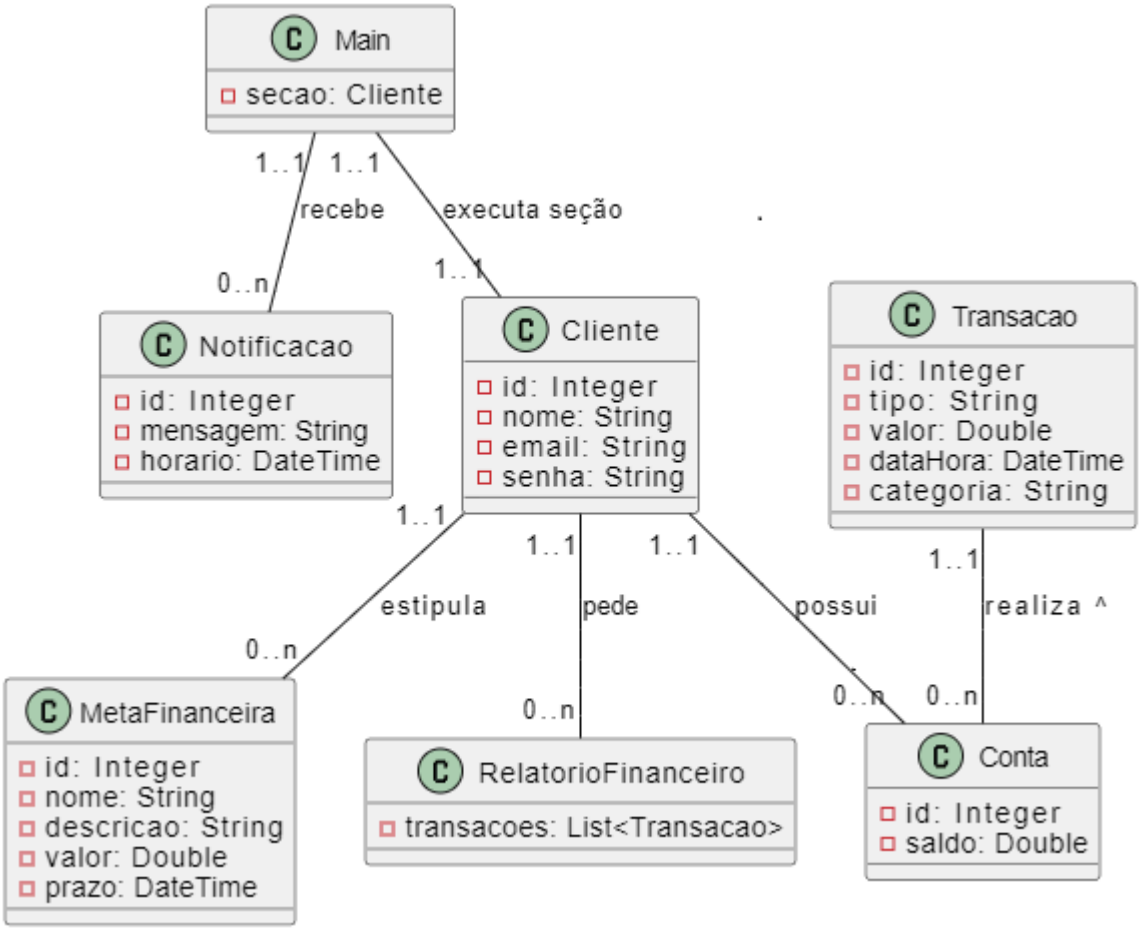
Em projeto de software, as visões arquiteturais consistem em representações abstratas e estruturadas de um sistema de software, que capturam diferentes perspectivas arquiteturais relevantes de forma que se observe a mesma coisa, ressaltando características e propriedades que sejam consideradas interessantes e omitindo as não relevantes. Uma visão arquitetural descreve como o sistema é organizado/esquematizado e como seus elementos interagem para atender aos requisitos funcionais e não funcionais observados.

As visões arquiteturais são de suma importância para o desenvolvimento do software, pois ajudam a decompor o complexo sistema em partes menores e gerenciáveis, além de comunicar, com clareza, as decisões para as partes interessadas. Cada visão arquitetural dá ênfase em aspectos específicos do sistema e fornece informações detalhadas sobre esses aspectos.

Existem diversas visões arquiteturais no mundo do desenvolvimento de software. Para o referente projeto, as visões arquiteturais disponíveis estão representadas abaixo:

4.1. CASO DE USO

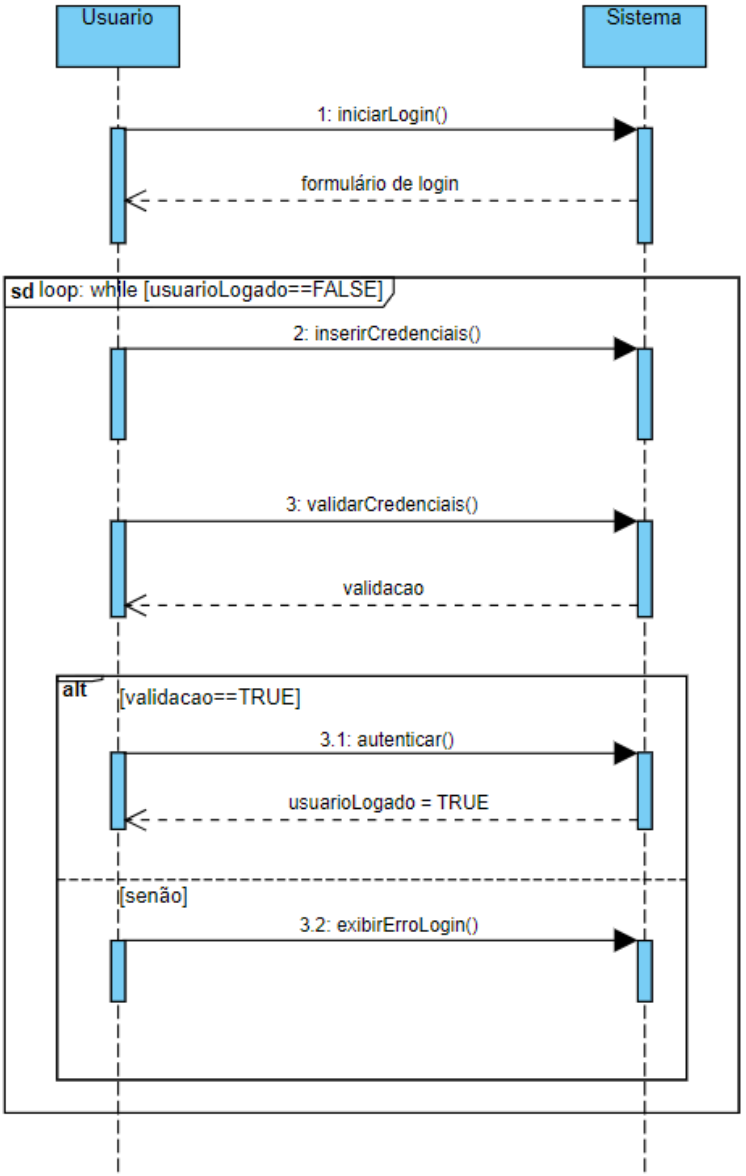
4.2. MODELO CONCEITUAL



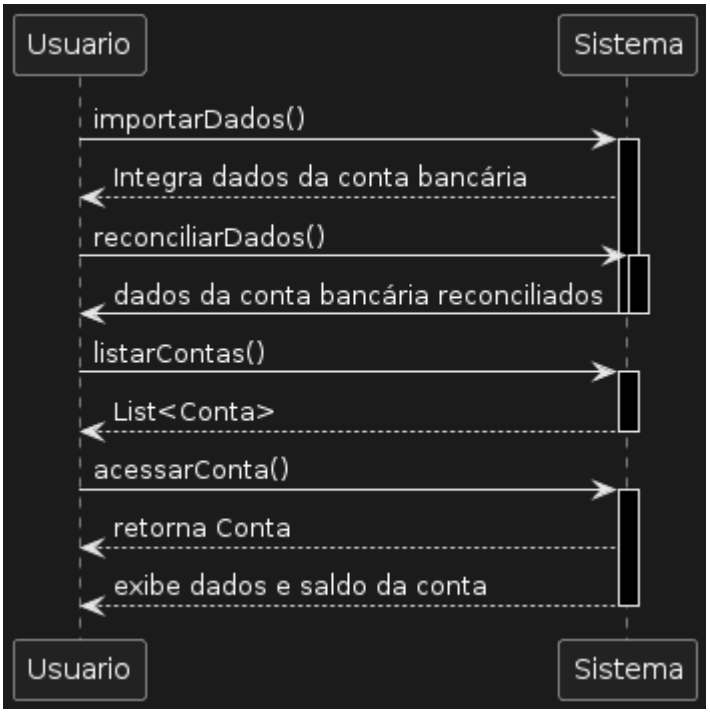
4.3. DIAGRAMAS DE SEQUÊNCIA DE SISTEMA

O Diagrama de Sequência é uma ferramenta de modelagem de sistemas. Ela representa a interação entre objetos em uma determinada sequência temporal, dando enfoque principalmente na troca de mensagens entre esses objetos ao longo do tempo. Para o software a ser desenvolvido, temos os seguintes diagramas de sequência de sistema representados abaixo:

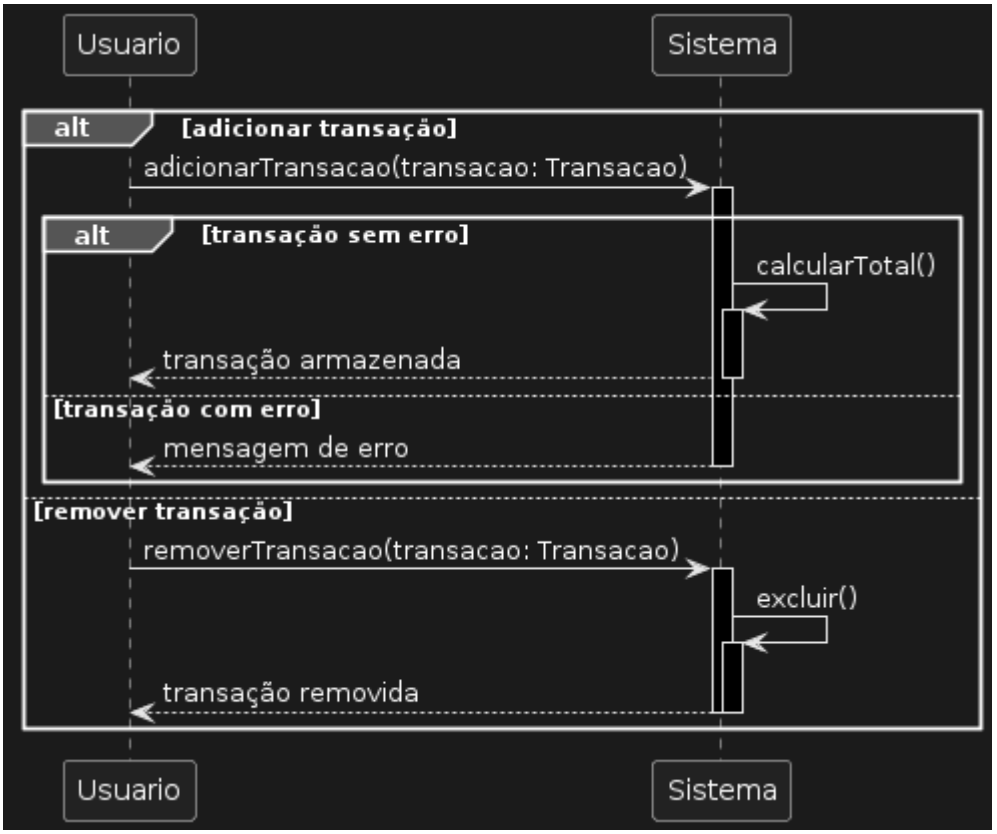
4.3.1. DSS: LOGIN



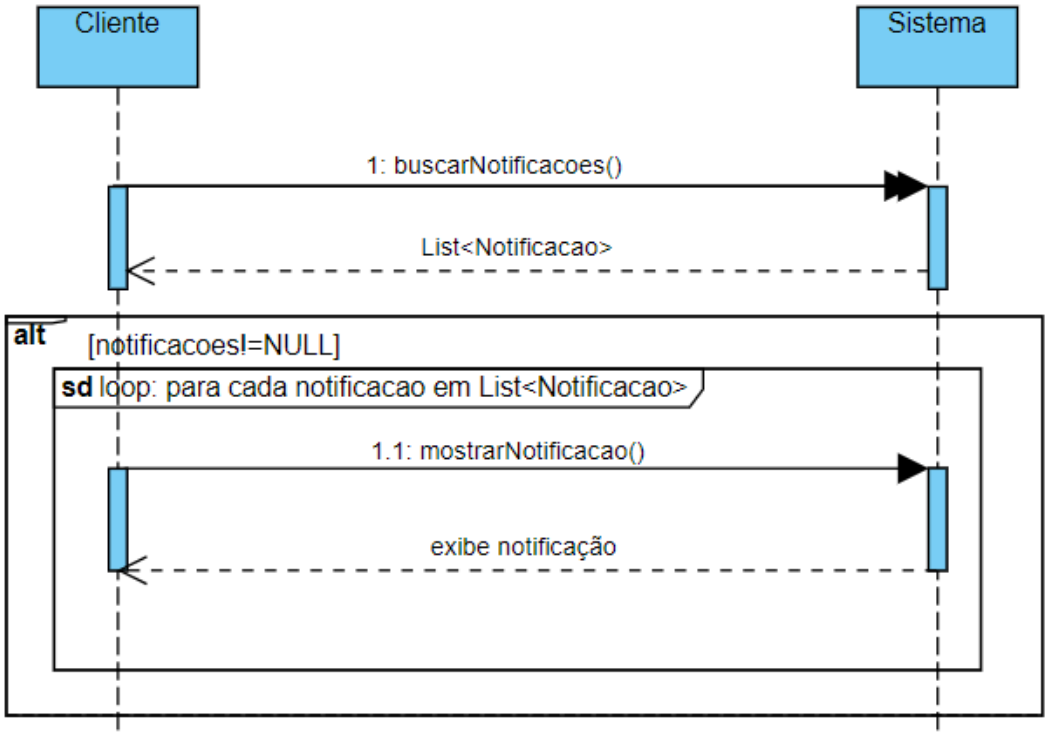
4.3.2. DSS: MANTER E INTEGRAR CONTAS BANCÁRIAS



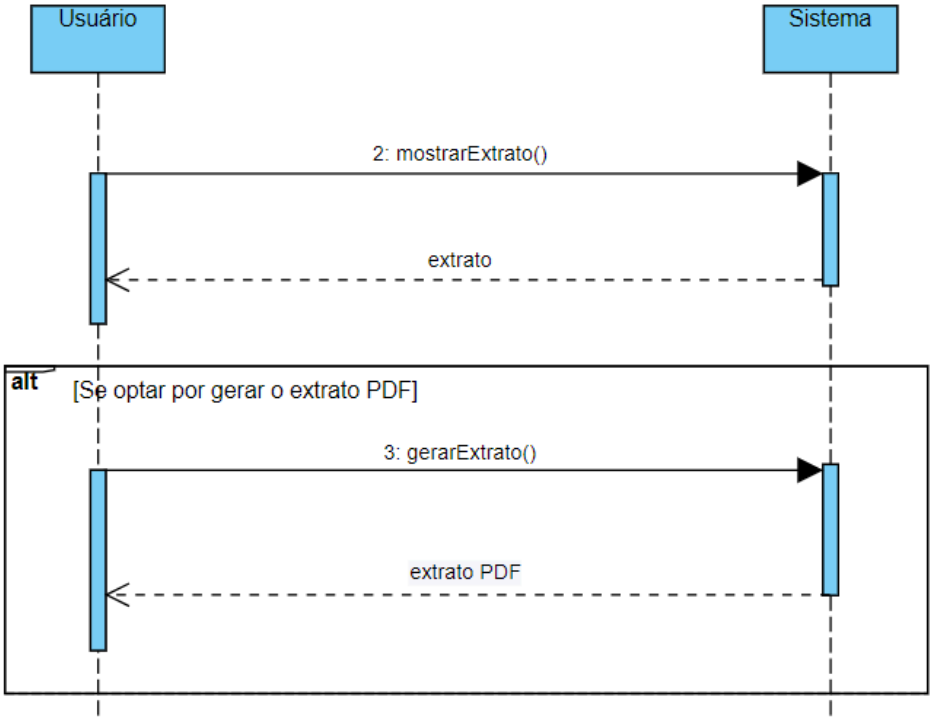
4.3.3. DSS: MANTER DESPESAS/RECEITAS



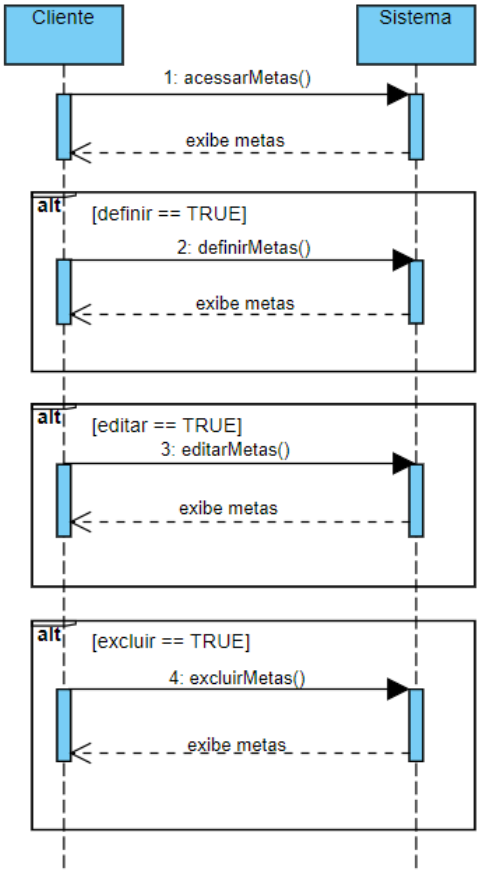
4.3.4. DSS: RECEBER NOTIFICAÇÃO



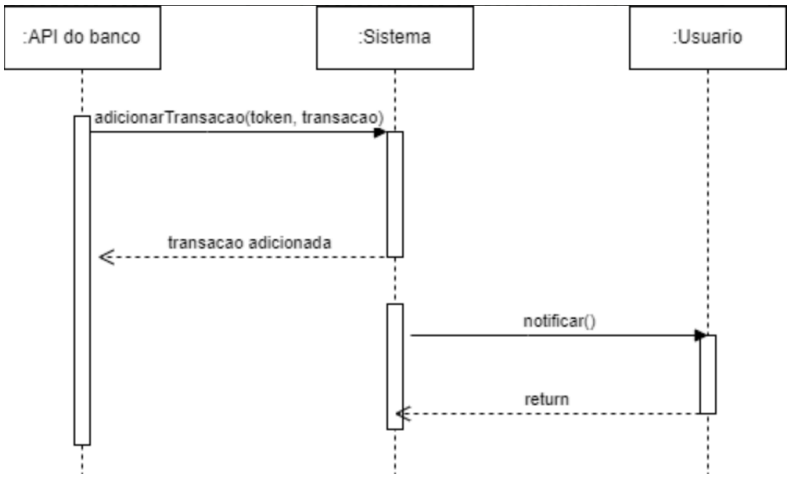
4.3.5. DSS: VISUALIZAR EXTRATO



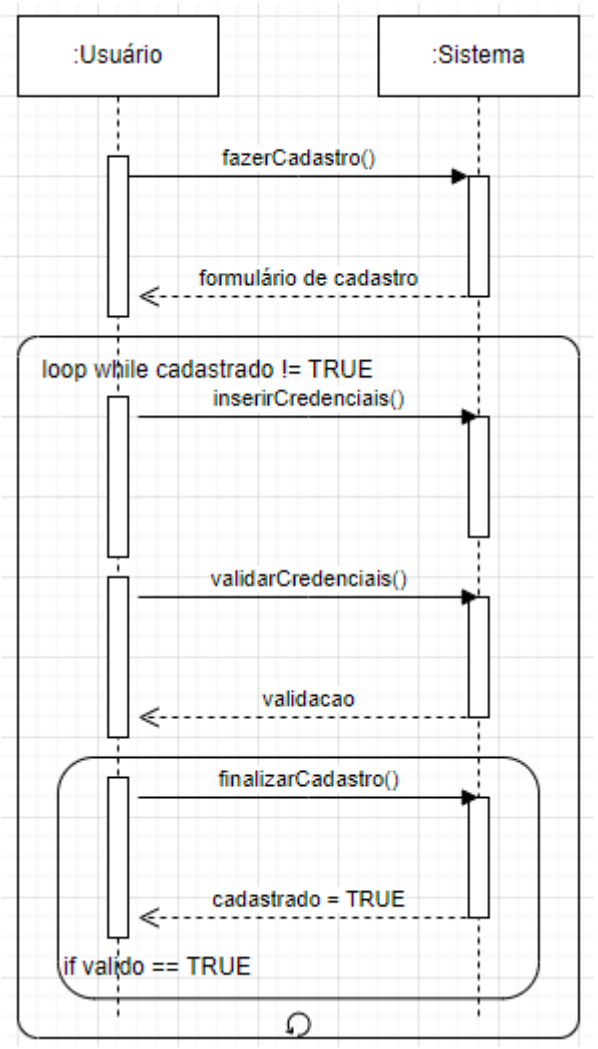
4.3.6. DSS: MANTER METAS FINANCEIRAS



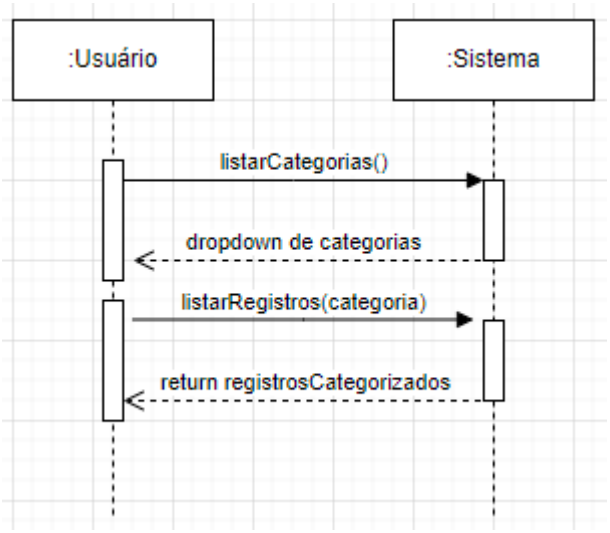
4.3.7. DSS: ADICIONAR TRANSAÇÃO



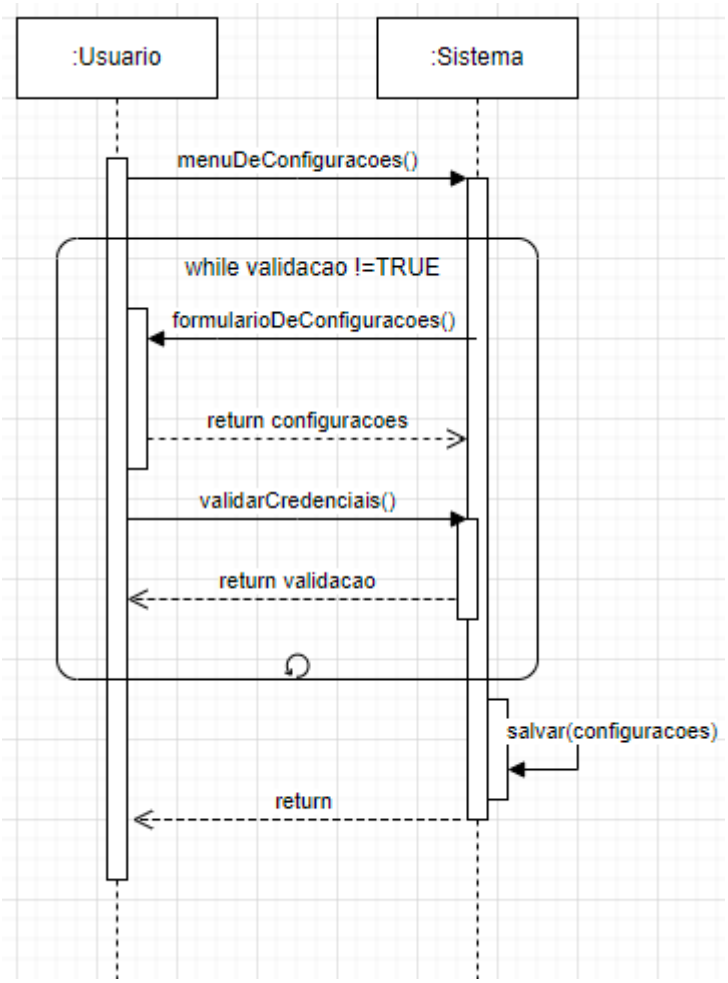
4.3.8. DSS: CADASTRAR



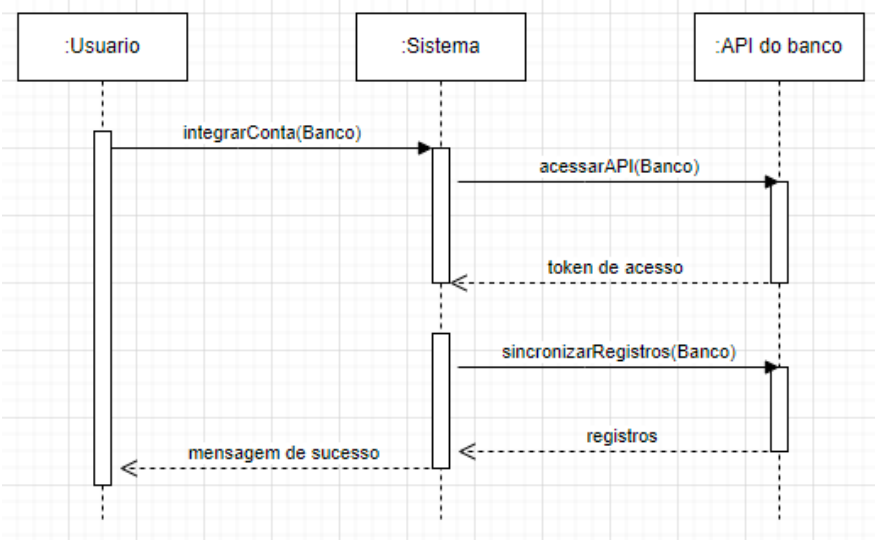
4.3.9. DSS: CATEGORIZAR REGISTROS



4.3.10. DSS: GERENCIAR CONTA



4.3.11. DSS: INTEGRAR CONTA



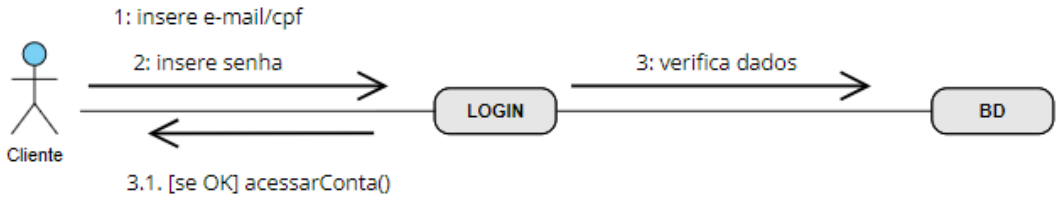
4.4. DIAGRAMAS DE INTERAÇÃO (COMUNICAÇÃO)

Os Diagramas de Interação são ferramentas utilizadas na modelagem de sistemas que proporcionam uma visão detalhada das interações dos objetos ao longo do tempo. Os principais diagramas de interação são os Diagramas de Sequência e os Diagramas de Comunicação.

Por opção, já que já temos o Diagrama de Sequência de Sistemas (DSS) acima, escolhemos representar essa modelagem através, então, do Diagrama de Comunicação.

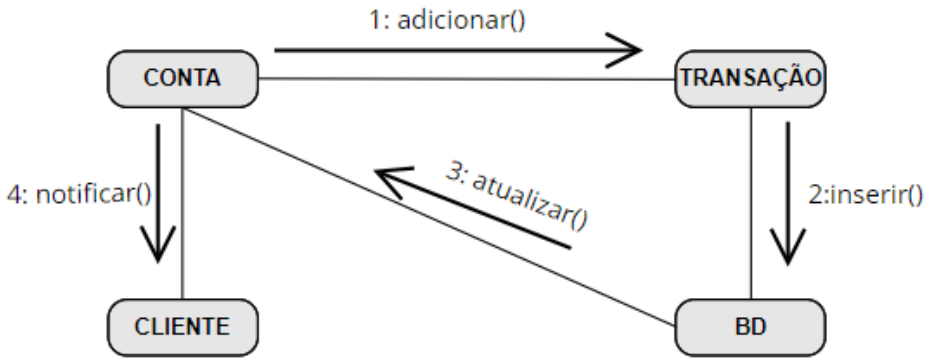
O diagrama de Comunicação oferece uma representação mais simplificada da interação entre objetos, enfatizando as relações estruturais e a troca de mensagens essenciais entre os objetos, sendo uma alternativa das mesmas informações dos diagramas de sequência. Nesse diagrama, a preocupação não está na ordem temporal em que as mensagens são trocadas, mas sim na organização estrutural dos objetos.

4.4.1. INSERIR CREDENCIAIS



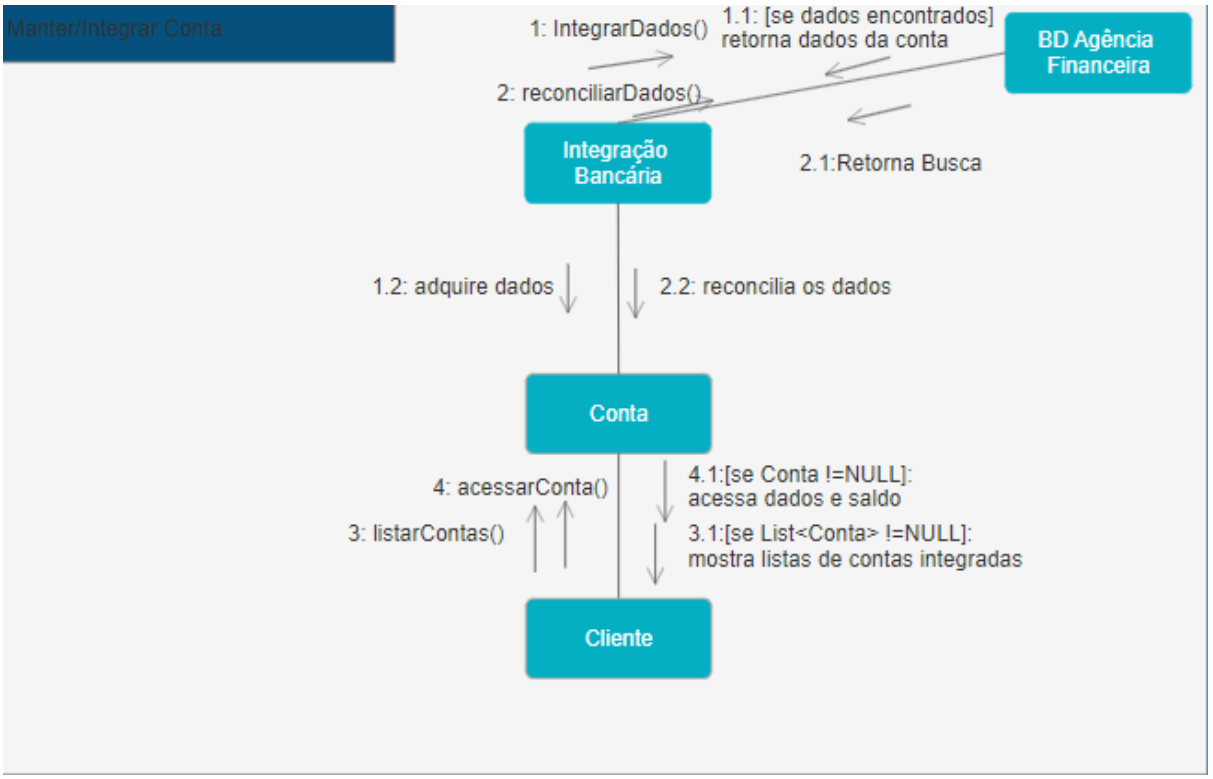
CONTRATO DA OPERAÇÃO	
OPERAÇÃO:	inserirCredenciais()
REFERÊNCIAS CRUZADAS:	Caso de Uso: “Login”
PRÉ-CONDIÇÕES:	• O cliente existe e sabe seu “ID”. • O cliente possui uma conta cadastrada no sistema.
PÓS-CONDIÇÕES:	• As credenciais foram inseridas pelo usuário. • O sistema autenticou o usuário e o associou a uma sessão.

4.4.2. ADICIONAR TRANSAÇÃO

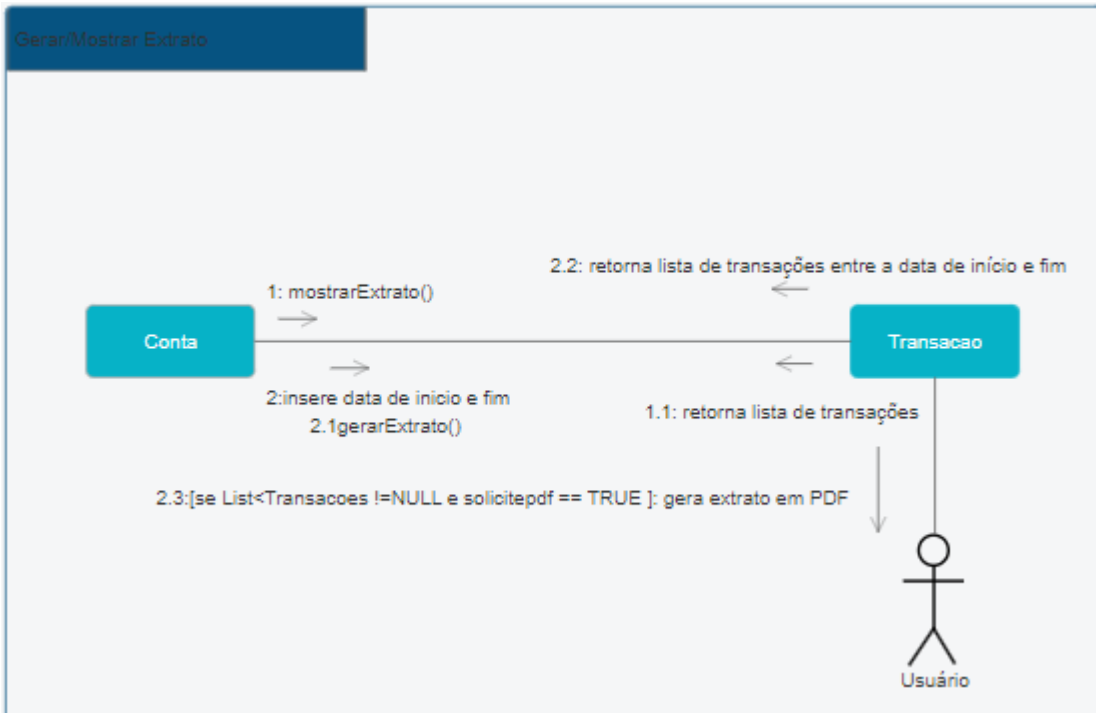


CONTRATO DA OPERAÇÃO	
OPERAÇÃO:	adicionarTransacao()
REFERÊNCIAS CRUZADAS:	Caso de Uso: “Manter Despesas/Receitas”
PRÉ-CONDIÇÕES:	• O cliente está logado no sistema. • O tipo de transação é especificada, receita ou despesa. • A descrição da transação é fornecida. • O valor da transação é fornecido.
PÓS-CONDIÇÕES:	• A transação é adicionada com sucesso ao sistema. • O saldo do usuário é atualizado no banco de dados do sistema.

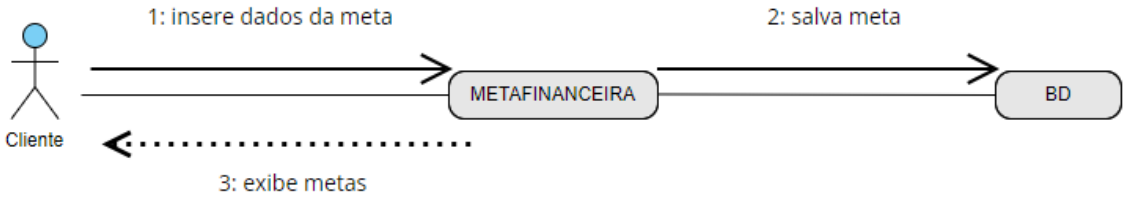
4.4.3. INTEGRAR/MANTER CONTA



4.4.4. VISUALIZAR EXTRATO



4.4.5. ADICIONAR METAS



CONTRATO DA OPERAÇÃO	
OPERAÇÃO:	adicionarMetas()
REFERÊNCIAS CRUZADAS:	Caso de Uso: “Manter Metas”
PRÉ-CONDIÇÕES:	• O cliente está logado no sistema. • Os dados da meta são válidos.
PÓS-CONDIÇÕES:	• A nova meta é adicionada à lista de metas.

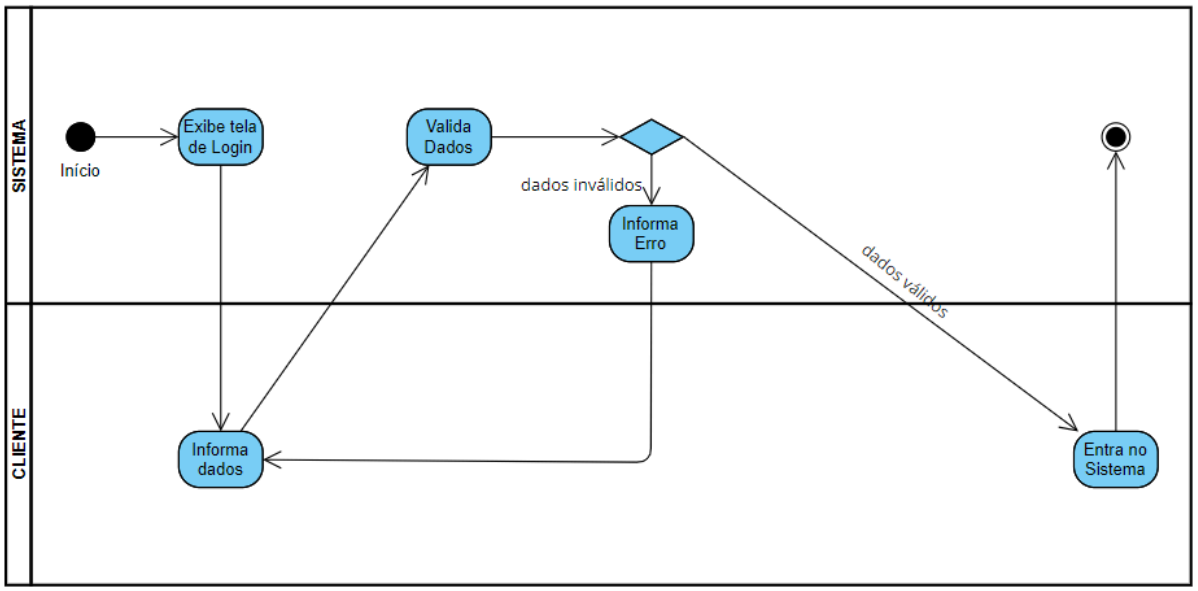
4.5. DIAGRAMAS DE ESTADO OU ATIVIDADES

O Diagrama de Estados, também conhecido como máquina de estados, se concentra em descrever o comportamento de um sistema em resposta a eventos que ocorrem, sejam eles internos ou externos. Com isso, esse diagrama serve para representar os diferentes estados que o sistema pode assumir ao longo do tempo e as transições desses estados.

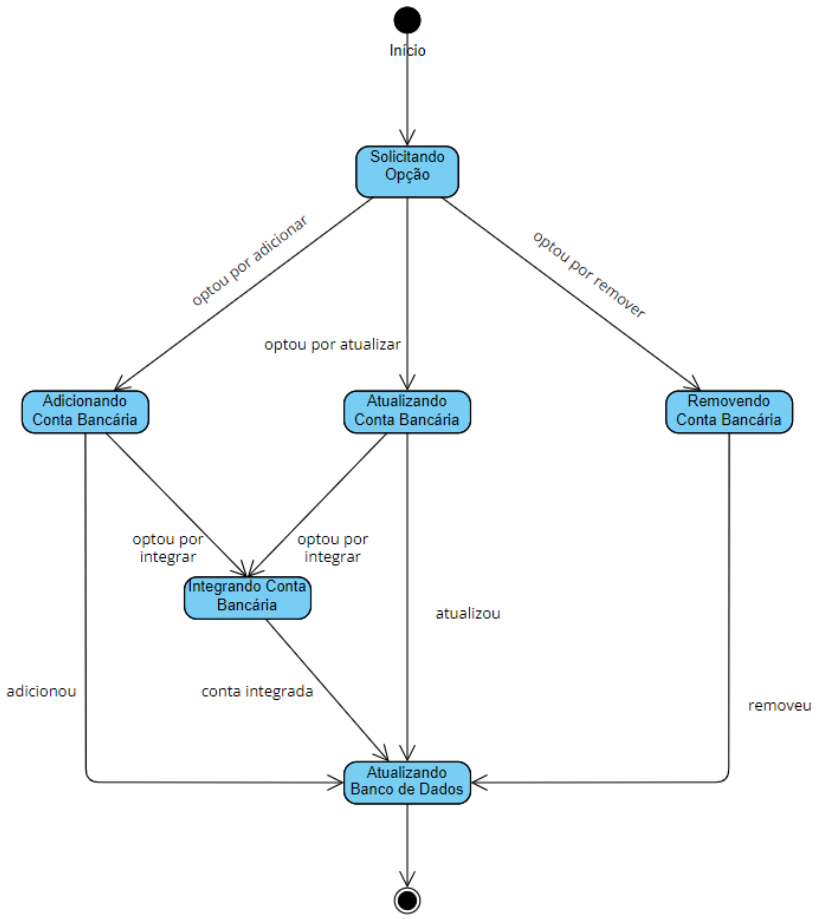
Já o Diagrama de Atividades se concentra em descrever o fluxo de trabalho do sistema ao mostrar atividades realizadas durante o processo, ou seja, fornece uma visualização do comportamento de um sistema descrevendo a sequência de ações em um processo.

Para o presente software em desenvolvimento, optamos por alguns diagramas de estado e atividades.

4.5.1. DIAGRAMA DE ATIVIDADES: LOGIN

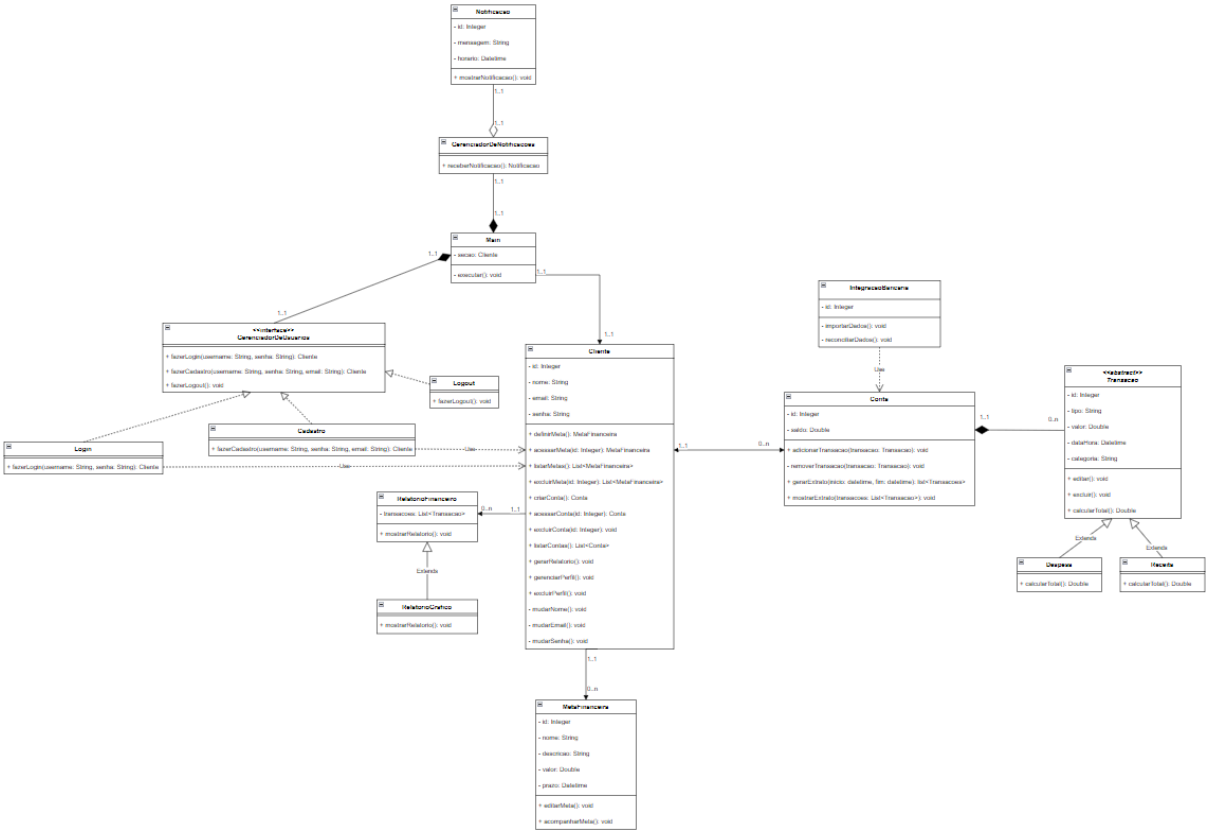


4.5.2. DIAGRAMA DE ESTADO: MANTER CONTA BANCÁRIA



4.6. DIAGRAMA DE CLASSE DETALHADO

O Diagrama de Classes Detalhado, é uma representação visual da estrutura do sistema em uma perspectiva orientada a objetos. Ele se concentra em descrever as classes do sistema, bem como seus atributos e métodos, além do relacionamento entre elas. Segue abaixo o referido diagrama deste projeto:



5. APLICAÇÃO DOS PRINCÍPIOS SOLID

Os Princípios SOLID são diretrizes cruciais para o desenvolvimento de software que visam promover código limpo, flexível e de fácil manutenção. Nesta seção, exploraremos como esses princípios são utilizados no contexto da aplicação.

5.1. RESPONSABILIDADE ÚNICA (SRP)

O Princípio da Responsabilidade Única afirma que uma classe deve ter uma, e apenas uma função. No caso usaremos o exemplo da classe AccountsController a qual, tem como responsabilidade gerenciar as operações CRUD (Create, Read, Update, Delete) relacionadas ao modelo Account. Isso está de acordo com o SRP, pois a classe como um todo está focada em uma única área funcional do sistema.

5.2. ABERTO-FECHADO (OCP)

As classes devem estar abertas para extensão, mas fechadas para modificação. Considerando o exemplo dos controladores AccountsController e TransacaosController. Se for preciso adicionar novas funcionalidades, como novos métodos de pagamento ou tipos de transações, é possível fazer isso criando subclasses ou módulos adicionais sem modificar os controladores existentes. Isso mantém o código existente intacto e promove a extensibilidade..

```
Unset
class AccountsController < ApplicationController
  # Código atual que lida com operações CRUD para contas
end

# Nova funcionalidade sem modificar AccountsController
class AdvancedAccountsController < AccountsController
  def export_to_csv
    # Implementação para exportar contas para CSV
  end
end
```

5.3. SEGREGAÇÃO DE INTERFACE (ISP)

O princípio da Segregação de Interfaces afirma que as classes não devem ser forçadas a implementar interfaces que não utilizam. Na aplicação, utilizamos a gem Devise para autenticação de usuários, que promovem implicitamente a Segregação de Interfaces. Por exemplo, Devise separa a lógica de autenticação em

controladores específicos, como Users::SessionsController, Users::RegistrationsController, e Users::PasswordsController, cada um lidando com aspectos específicos da autenticação.

6. APLICAÇÃO DOS PADRÕES GRASP

Os padrões GRASP são fundamentais para a construção de um sistema bem estruturado e de fácil manutenção. Nesta seção, discutimos como esses padrões foram aplicados em diferentes partes da nossa aplicação, garantindo uma arquitetura robusta e coesa.

6.1. ESPECIALISTA

O padrão Especialista é seguido no Users::RegistrationsController, que herda de Devise::RegistrationsController. Este controlador gerencia SOMENTE as ações relacionadas ao registro de novos usuários, como *new*, *create*, *edit*, *update* e entre outros, encapsulando a lógica necessária para o processo de criação de contas de forma coesa e especializada.

```
Unset
class Users::RegistrationsController < Devise::RegistrationsController
       before_action :configure_sign_up_params, only: [:create]
       protected
       defconfigure_sign_up_params
              devise_parameter_sanitizer.permit(:sign_up, keys: [:name])
       end
       protected
       def after_sign_up_path_for(resource)
              root_path
       end
       ...
```

6.2. CRIADOR

Os criadores são classes responsáveis por criar objetos de outras classes, ou seja, uma classe A deve ser responsável por criar instâncias da classe B. No referido projeto, pode-se observar que a classe *User* possui associações com contas, metas financeiras e transação, ou seja, ao criar um novo usuário, ele pode automaticamente criar e associar novas contas. Para isso, basta observar um trecho de código do model abaixo:

```
Unset
class User < ApplicationRecord
  # Include default devise modules. Others available are:
```

```
  # :confirmable, :lockable, :timeoutable, :trackable and :omniauthable
  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :validatable

  validates :name, presence: true

  has_many :accounts, dependent: :destroy
  has_many :meta_financeiras, dependent: :destroy
  has_many :transacaos, dependent: :destroy
end
```

O mesmo vale para a classe Conta, que é responsável por geradas instâncias da classe Transação.

6.3. ACOPLAMENTO FRACO

O princípio de Acoplamento Fraco é seguido através da utilização de associações do ActiveRecord, que gerenciam a ligação entre diferentes modelos sem um acoplamento rígido. Por exemplo, a classe Transacao está associada às classes User e Account usando belongs_to, o que permite que cada transação conheça sua conta e usuário associados sem depender fortemente de detalhes específicos da implementação de User e Account.

```
Unset
class Transacao < ApplicationRecord
       belongs_to :user
       belongs_to :account
end
```

6.4. COESÃO ALTA

As classes no sistema exibem alta coesão, significando que cada uma tem um conjunto bem definido de responsabilidades que estão intimamente relacionadas às suas respectivas áreas funcionais. A classe User gerencia os dados do usuário, autenticação e relacionamentos com contas, metas financeiras e transações. Todos os métodos e atributos são voltados para a gestão do usuário, tornando a classe altamente coesa.

```
Unset
class User < ApplicationRecord
       devise :database_authenticatable, :registerable,
              :recoverable, :rememberable, :validatable
       validates :name, presence: true
       has_many :accounts, dependent: :destroy
       has_many :meta_financeiras, dependent: :destroy
       has_many :transacaos, dependent: :destroy
end
```

6.5. CONTROLADOR

O padrão Controlador é implementado em todas as classes *controller* da aplicação. Por exemplo, *AccountsController* e *MetaFinanceirasController* recebem e manipulam requisições HTTP, delegando as tarefas apropriadas aos modelos.

As classes controladoras são um “meio de campo” entre o *Model* e o *View*, ou seja, são responsáveis por gerenciar a comunicação entre o usuário e o sistema, através das *views*, e delegar as ações para os *models* apropriados.

7. PADRÕES GOF

Os padrões de design GoF (Gang of Four) são um conjunto de 23 padrões de design amplamente utilizados no desenvolvimento de softwares para resolver problemas comuns de design de software. Em Ruby on Rails, esses padrões são aplicados de maneira implícita através das ferramentas fornecidas pelo framework.

A seguir, pode-se perceber como alguns desses padrões podem ser identificados e utilizados dentro do Rails.

7.1. SINGLETON

O padrão Singleton pode ser percebido na biblioteca Devise (gem Devise). O Devise é uma biblioteca de autenticação que manipula sessões de usuários e usa o Singleton para gerenciar suas configurações globais e garantir que haja apenas uma instância de usuários logada/autenticada na aplicação.

Toda a configuração do Devise é centralizada e compartilhada através de uma única instância de configuração. Isso garante que todas as partes do seu aplicativo que utilizam o Devise estejam usando as mesmas configurações.

Esse padrão é utilizado no referente projeto para ter apenas uma instância de usuário logado na sessão. Isso é fundamental para manter a consistência e a segurança da aplicação, garantindo que as informações de autenticação e sessão sejam gerenciadas de forma centralizada e segura. Além disso, a gestão da sessão do usuário é um aspecto crucial onde o Singleton se destaca. Quando um usuário faz login, as informações de sessão são armazenadas em uma instância única, evitando conflitos e garantindo que apenas um usuário esteja autenticado por sessão.

7.2. TEMPLATE METHOD

O padrão Template Method é visível nos controladores e modelos do Ruby on Rails, onde métodos base fornecem um esqueleto/template/modelo e permitem que subclasses definam comportamentos específicos. Isso promove a reutilização de código e a separação de preocupações, facilitando a manutenção e a extensão da aplicação.

No exemplo dos Controllers, pode-se perceber o padrão Template Method dentro de qualquer uma das classes *[nome]_controller.rb*, que herda de *ApplicationController*.

No *transacaos_controller.rb*, é visto esse fato. Para isso, basta observar o

trecho de código abaixo:

```
Unset
class TransacaosController < ApplicationController
  before_action :set_transacao, only: %i[ show edit update destroy ]
  before_action :authenticate_user!

  # GET /transacaos or /transacaos.json
  def index
    if params[:account_id].present?
      @transacaos = current_user.transacaos.where(account_id:
params[:account_id])
    else
      @transacaos = current_user.transacaos
    end
  end

  # GET /transacaos/1 or /transacaos/1.json
  def show
  end

  # GET /transacaos/new
  def new
    @transacao = Transacao.new
  end

  # GET /transacaos/1/edit
  def edit
  end

  # POST /transacaos or /transacaos.json
  def create
    @transacao = current_user.transacaos.build(transacao_params)

    respond_to do |format|
      if @transacao.save
        format.html { redirect_to @transacao, notice: 'Transacao criada com
sucesso.' }
        format.json { render :show, status: :created, location: @transacao }
      else
        format.html { render :new, status: :unprocessable_entity }
        format.json { render json: @transacao.errors, status:
:unprocessable_entity }
      end
    end
  end
```

Nesse controlador, pode-se identificar o padrão Template Method principalmente através do uso de callbacks como *before_action*. Esses callbacks definem um esqueleto de operações que devem ocorrer antes de certas ações, permitindo que o comportamento específico seja implementado conforme necessário. Também pode-se observar no uso de certos métodos que são herdados da classe pai, como o *index*, *show*, *new*, *edit*, *create* e entre outros.

7.3. BUILDER

O padrão Builder é visível na criação de objetos complexos no Ruby on Rails, onde uma série de etapas são necessárias para configurar corretamente um objeto. O Builder separa a construção de um objeto complexo da sua representação, permitindo a criação de diferentes representações para o mesmo tipo de objeto.

Em Ruby on Rails, o Builder é utilizado para a construção de formulários. O método form_with é um exemplo da aplicação desse padrão, onde o formulário é construído passo a passo e é passado como deve ser feita a construção desse formulário. Observe:

```
Unset
<%= form_with(model: account) do |form| %>
  <% if account.errors.any? %>
    <div style="color: red">
      <h2><%= pluralize(account.errors.count, "error") %> prohibited this
account from being saved:</h2>

      <ul>
        <% account.errors.each do |error| %>
          <li><%= error.full_message %></li>
        <% end %>
      </ul>
    </div>
  <% end %>

  <div>
    <%= form.label :nomeConta, style: "display: block" %>
    <%= form.text_field :nomeConta, class:"form-control" %>
  </div>

  <div>
    <%= form.label :saldo, style: "display: block" %>
    <%= form.text_field :saldo, class:"form-control" %>
  </div>
```

```
  <div class="form-group text-center mt-4">
    <%= form.submit  class: "btn btn-success btn-block
custom-nova-conta-btn", style:"width:15vw; background-color: green; color:
white; border: 2px solid transparent; transition: all 0.3s ease;"%>
  </div>
<% end %>
```

Neste exemplo, o *form_with* age como um construtor, permitindo que o desenvolvedor adicione campos de formulário de maneira incremental e organizada.

8. LINHA DE PRODUTO DE SOFTWARE (LPS)

A Linha de Produto de Software é uma estratégia de desenvolvimento de software que envolve a criação de uma família de produtos relacionados, que compartilham características comuns que satisfazem as necessidades de um segmento específico, mas também possuem variações que permitem a customização para diferentes áreas.

Ela possui um conjunto de características que são definidas em mandatórias, aquelas que todos os produtos da linha devem possuir; opcionais, que são características adicionais que podem ser incluídas ou não; e alternativas, que são características que podem ter diferentes implementações, mas todo produto deve contar com uma das alternativas.

Para esse Gerenciador de Finanças Pessoais, a LPS é a seguinte:

- **Mandatórias:**
  - Cadastro de Usuário
  - Cadastro de Contas
  - Cadastro de Transações e Saldos
  - Segurança dos Dados
  - Dashboard de Resumo Financeiro
- **Opcionais:**
  - Análise de Investimentos
  - Planejamento de Aposentadoria
  - Gestão de Impostos
  - Scanner de Recibos
- **Alternativas:**
  - Metodologia de Sincronização de Dados
    - Armazenamento em Nuvem
    - Sincronização Local
    - Integração com APIs de Terceiros
  - Modos de Visualização de Dados
    - Visualização de Calendário
    - Listagens Detalhadas
    - Visualizações Gráficas Avançadas

Dadas as características acima, tem-se o seguinte gráfico: