



Instituto de Ciências Matemáticas e de Computação  
Universidade de São Paulo - USP

# Sistema para gestão financeira e controle de carteira de investimentos.

**Disciplina:** Análise e Projeto Orientados a Objetos - SSC0124

**Docente:** Dra. Lina Garcés.

**Alunos [nome(NUSP)]:**

- Maria Rita Gomides Cruz Hott (14758717)
- José Carlos Andrade do Nascimento (12549450)
- Shogo Shima (12675145)

# Sumário

<b>Sumário.....</b>	<b>2</b>
<b>1. Introdução.....</b>	<b>4</b>
<b>2. Especificação de Requisitos.....</b>	<b>4</b>
2.1. História de Usuário.....	4
2.2. Personas.....	5
2.3. Diagrama de Casos de Uso.....	6
2.4. Especificação de Casos de Uso Textuais Abstratos.....	6
2.5. Especificação de Casos de Uso Textuais Estendidos.....	9
2.5.1. Criar metas financeiras.....	9
2.5.2. Adicionar gastos fixos/variáveis.....	9
2.5.3. Visualizar histórico de transações.....	10
<b>3. Arquitetura do Sistema:.....</b>	<b>11</b>
3.1. Diagrama de componentes:.....	11
3.2. Descrição dos Componentes do sistema:.....	11
3.2.1. Componentes Explícitos.....	11
3.2.2. Componentes Implícitos.....	12
<b>4. Modelo de Classes.....</b>	<b>12</b>
4.1. Descrição dos Componentes do sistema.....	12
4.1.1. Componente de Usuário:.....	12
4.1.2. Componente de Transação:.....	13
4.1.3. Componente de Autenticação:.....	13
4.2. Identificação das propriedades ou atributos das Classes.....	13
4.2.1. Componente de Usuário.....	13
4.2.2. Componente de Transação.....	14
4.2.3. Componente de Autenticação.....	15
4.3. Visibilidade de atributos.....	16
4.4. Relacionamentos e Multiplicidade.....	16
4.5. Diagrama de Classes Atualizado:.....	18
<b>5. Modelo de comportamento.....</b>	<b>18</b>
5.1. Diagrama de atividades do sistema com partições.....	18
5.1.1. Diagrama de Atividade para autenticação do usuário:.....	19
5.1.2. Diagrama de Atividade para adicionar dados de transações:.....	19
5.1.3. Diagrama de Atividades para registrar objetivos:.....	20
5.2. Diagrama de atividades do sistema.....	21
5.2.1. Diagrama de Atividade para autenticação do usuário:.....	21
5.2.2. Diagrama de Atividade para adicionar dados de transações:.....	22
5.2.3. Diagrama de Atividades para registrar objetivos:.....	23
<b>6. Modelos de Interações.....</b>	<b>24</b>
6.1. Diagramas de Comunicação.....	24
6.1.1. Diagrama de Sequência para Criação de metas ou objetivos:.....	24
6.1.2. Diagrama de Atividades para adicionar dados de transações:.....	25
6.1.3. Diagrama de Atividades para registrar controle de gastos:.....	25
6.2. Diagramas de Sequência.....	26
6.2.1. Diagrama de Sequência para Criação de metas ou objetivos:.....	26
6.2.2. Diagrama de Atividades para adicionar dados de transações:.....	27
6.2.3. Diagrama de Atividades para registrar controle de gastos:.....	28

<b>7. Código do Sistema</b>	<b>28</b>
7.1. Descrição da Tecnologia Seleccionada	29
7.1.1. Next.js:	29
7.1.2. Prisma:	29
7.1.3. PostgreSQL:	29
7.2. Print da estrutura do projeto	29
7.3. Link para o GitHub	29
7.4. Código correspondente aos casos de uso desenvolvidos	29
<b>8. Conclusão e Discussões</b>	<b>30</b>
8.1. Desafios de planeamento de correlação entre os componentes	30
8.2. Arquitetura limpa do sistema	30
8.3. Segurança e proteção dos dados	30
8.4. Concepção e desenvolvimento das interfaces	30

# 1. Introdução

O projeto central da disciplina de Análise e Projeto Orientado a Objetos, sob a orientação da professora Dra. Lina Garcés, propõe o desenvolvimento de um sistema para controle de finanças. Este aplicativo tem como objetivo auxiliar os usuários a gerenciar suas finanças de maneira prática e eficiente, abordando problemas comuns como a falta de controle sobre despesas, a dificuldade em estabelecer e manter metas financeiras, e a necessidade de um acompanhamento contínuo das transações.

Os principais atores do sistema são os usuários finais, que buscam melhorar o gerenciamento de suas finanças pessoais. Esse público abrange uma ampla gama de stakeholders, desde universitários iniciantes em estágios até investidores mais experientes. O sistema foi projetado para atender às necessidades específicas desse público, proporcionando uma interface acessível e intuitiva.

As principais funcionalidades do sistema incluem: o cadastro de renda fixa e variável, o registro detalhado de despesas, um histórico de transações mensais, o estabelecimento de metas financeiras e uma funcionalidade que orienta os usuários na direção de investimentos. Essas funcionalidades foram selecionadas para garantir que o aplicativo não apenas organize as finanças dos usuários, mas também os capacite a tomar decisões informadas.

Como resultados esperados, espera-se que os usuários desenvolvam um maior controle sobre suas finanças, promovendo uma melhor organização e gestão do fluxo de caixa. O sistema proporcionará uma experiência didática e intuitiva, permitindo que os usuários alcancem maior estabilidade financeira e, conseqüentemente, melhorem sua qualidade de vida.

Com esse projeto, buscamos não apenas oferecer uma ferramenta, mas também criar um suporte valioso para que os usuários possam alcançar suas metas financeiras e enfrentar os desafios do dia a dia com confiança e clareza.

## 2. Especificação de Requisitos

### 2.1. História de Usuário

Dentre as histórias de usuário entregues nas atividades semanais da disciplina, vamos destacar as narrativas mais relevantes para o sistema a seguir:

- **Recém-formado:**

Como um **recém-formado** que acabou de conseguir seu primeiro emprego, eu quero uma ferramenta que me ajude a criar um orçamento inicial e controlar meus gastos diários, garantir que eu não gaste mais do que ganho e possa começar a juntar dinheiro para o futuro.

- **Investidor Experiente :**

Como um **investidor experiente** com múltiplos ativos e investimentos em diferentes plataformas, eu quero um aplicativo que centralize todas as minhas

movimentações financeiras em um único lugar, que eu possa controlar de forma eficiente meu portfólio sem perder tempo alternando entre diferentes sistemas

- **Usuário médio:**

Como um **usuário médio** que tem algumas economias e deseja investir em fundos de investimento, eu quero uma ferramenta que me ajude a entender melhor os diferentes tipos de fundos disponíveis, que eu possa tomar decisões mais informadas e com base em dados reais sobre onde alocar meu dinheiro.

## 2.2. Personas

Com o auxílio das histórias de usuário apresentadas no seguimento anterior, foi possível definir 3 personas, as quais são representações fictícias baseadas em dados reais, para o sistema. Essas podem ser detalhadas como:

- **Tipo 1: Investidor experiente**

João Silva tem 45 anos e é um empresário e investidor em tempo integral. Ele começou a investir aos 25 anos, inicialmente no mercado imobiliário, e com o tempo diversificou sua carteira para incluir ações, fundos de investimento e criptomoedas. Ele utiliza várias plataformas para gerenciar seus investimentos e realiza operações de compra e venda com frequência. Para João, ter controle rigoroso e detalhado sobre suas finanças é essencial.

- **Frustração:** Perder tempo com várias plataformas separadas e não ter uma visão consolidada de todos os seus ativos.
- **Motivação:** Maximizar o retorno de seus investimentos, controlando detalhadamente todas as transações.

**Histórias de Usuário:**

1. Gostaria de centralizar todas as minhas movimentações financeiras em uma única plataforma.
2. Gostaria de visualizar em tempo real o desempenho do meu portfólio.
3. Gostaria de gerar relatórios detalhados sobre meus ganhos e perdas em múltiplos ativos.

- **Tipo 2: Iniciantes de carreira**

Luísa Almeida tem 24 anos e é estagiária de Marketing Digital. Ela recentemente começou sua carreira e agora tem uma renda estável pela primeira vez. Antes disso, ela usava planilhas para tentar controlar suas finanças, mas as achava confusas e ineficazes. Ela deseja começar a economizar e investir, mas ainda não tem muito conhecimento sobre o mercado financeiro.

- **Frustração:** Dificuldade em manter controle financeiro e entender onde economizar para começar a investir.
- **Motivação:** Organizar suas finanças de maneira simples e começar a investir para o futuro.

**Histórias de Usuário:**

1. Gostaria de criar um orçamento mensal fácil de seguir.
2. Gostaria de visualizar meus gastos de forma simples para saber onde estou gastando demais.
3. Gostaria de definir metas financeiras para curto e médio prazo.

- **Tipo 3: Usuário médio**

Ricardo Oliveira tem 35 anos e é analista de Sistemas. Ele tem uma renda estável e já fez alguns investimentos pequenos em fundos de renda fixa e poupança. Ele não acompanha seus investimentos com tanta frequência, mas sabe que precisa de mais organização para evitar gastos desnecessários e manter sua saúde financeira em dia.

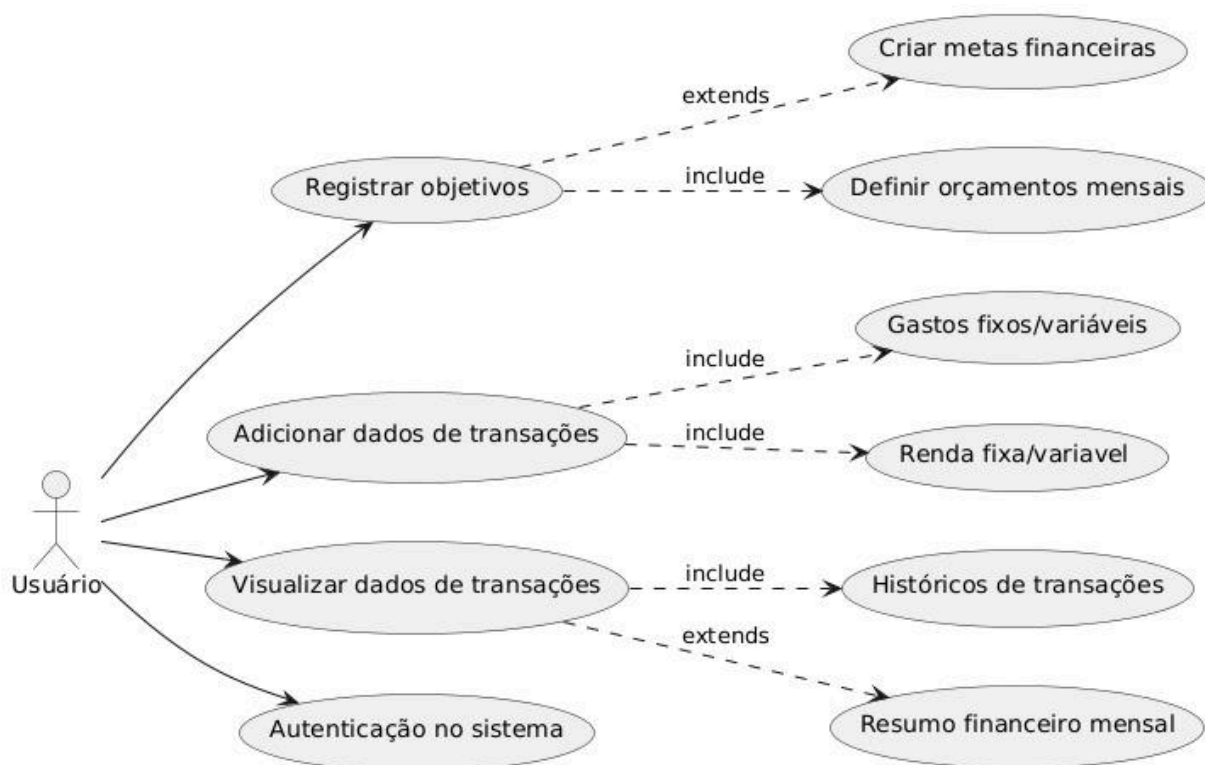
- **Frustração:** Falta de controle sobre pequenos gastos e pouca visibilidade de suas economias ao longo do tempo.
- **Motivação:** Evitar desperdício de dinheiro e ter mais clareza sobre o andamento de seus investimentos.

#### Histórias de Usuário:

1. Gostaria de visualizar de forma simples todos os meus gastos e rendimentos mensais.
2. Gostaria de controlar o quanto estou economizando a cada mês.
3. Gostaria de ter um relatório anual das minhas despesas e economias.

## 2.3. Diagrama de Casos de Uso

O diagrama de casos de uso foi elaborado e refinado com base nos objetivos gerais do sistema e nos conhecimentos obtidos durante o curso, assim, o diagrama final se definiu como:



## 2.4. Especificação de Casos de Uso Textuais Abstratos

1. **Cadastrar renda fixa/variável:** O usuário adiciona uma nova entrada de renda e classifica como fixa (ex.: salário) ou variável (ex.: bônus, renda extra). O fluxo principal segue abaixo:
  - 1.1. O usuário acessa a funcionalidade de cadastro de renda.

- 1.2. O sistema solicita informações sobre a renda, como valor, data e tipo.
  - 1.3. O usuário insere os detalhes e confirma o cadastro.
  - 1.4. O sistema salva a entrada de renda e a associa ao histórico do usuário.
2. **Cadastrar gastos fixos/variáveis:** O usuário registra uma nova despesa e classifica como fixa (ex.: aluguel, contas) ou variável (ex.: lazer, compras). O fluxo principal segue abaixo:
  - 2.1. O usuário acessa a funcionalidade de cadastro de despesas.
  - 2.2. O sistema solicita informações sobre o gasto, como valor, data e tipo.
  - 2.3. O usuário insere os detalhes e confirma o registro.
  - 2.4. O sistema salva a despesa e a associa ao histórico do usuário.
3. **Visualizar resumo financeiro mensal:** O sistema gera um resumo das entradas e saídas de dinheiro do mês, exibindo o saldo final e a distribuição dos gastos. O fluxo principal segue abaixo:
  - 3.1. O usuário acessa a opção de resumo financeiro.
  - 3.2. O sistema recupera as entradas e saídas do mês atual.
  - 3.3. O sistema calcula o saldo final e distribui os gastos por categoria.
  - 3.4. O resumo é exibido para o usuário.
4. **Criar metas financeiras:** O usuário define metas de economia ou investimento, especificando valores e prazos para serem alcançados. O fluxo principal segue abaixo:
  - 4.1. O usuário acessa a funcionalidade de criação de metas.
  - 4.2. O sistema solicita o valor e o prazo para a meta.
  - 4.3. O usuário define os detalhes e confirma.
  - 4.4. O sistema salva a meta e associa ao perfil do usuário.
5. **Gerar relatórios financeiros:** O usuário solicita relatórios detalhados de suas transações financeiras em períodos específicos ou categorias. O fluxo principal segue abaixo:
  - 5.1. O usuário acessa a funcionalidade de geração de relatórios.
  - 5.2. O sistema solicita o período ou as categorias a serem incluídas.
  - 5.3. O usuário define os filtros desejados e confirma.
  - 5.4. O sistema gera e exibe o relatório para o usuário.
6. **Inserir e visualizar histórico de transações:** Todas as transações são salvas automaticamente no histórico, e o usuário pode consultar transações anteriores. O fluxo principal segue abaixo:

- 6.1. O usuário acessa a funcionalidade de histórico de transações.
  - 6.2. O sistema recupera as transações do período desejado.
  - 6.3. O usuário visualiza o histórico solicitado.
7. **Conectar a plataformas de investimentos:** O usuário integra suas contas de investimento para visualizar o desempenho dos ativos. O fluxo principal segue abaixo:
  - 7.1. O usuário acessa a opção de conexão com plataformas de investimento.
  - 7.2. O sistema orienta o usuário para autorizar o acesso à plataforma de terceiros.
  - 7.3. O sistema recupera os dados de investimentos e exibe o desempenho em tempo real.
8. **Definir orçamentos mensais:** O usuário define um orçamento mensal, incluindo alertas para evitar o excesso de gastos. O fluxo principal segue abaixo:
  - 8.1. O usuário acessa a funcionalidade de definição de orçamento.
  - 8.2. O sistema solicita o limite de gastos.
  - 8.3. O usuário define o limite e confirma.
  - 8.4. O sistema registra o orçamento e ativa alertas para controle dos gastos.
9. **Alertas de economia e investimento:** O sistema sugere ao usuário que direcione parte de seu saldo para economias ou investimentos, com base em suas metas financeiras. O fluxo principal segue abaixo:
  - 9.1. O sistema monitora o saldo e as metas do usuário.
  - 9.2. Quando uma meta é próxima de ser alcançada, o sistema gera um alerta.
  - 9.3. O sistema sugere ao usuário direcionar parte do saldo para a meta ou investimento.
10. **Autenticação e cadastro de usuário:** O sistema permite que o usuário faça cadastro, login e tenha acesso às suas informações pessoais e financeiras. O fluxo principal segue abaixo:
  - 10.1. O usuário acessa a funcionalidade de login/cadastro.
  - 10.2. O sistema solicita informações de registro ou autenticação.
  - 10.3. O usuário insere as informações e confirma.
  - 10.4. O sistema valida as informações e concede acesso ao usuário.



## 2.5. Especificação de Casos de Uso Textuais Estendidos

A Especificação de Casos de Uso Textuais Estendidos serve para descrever detalhadamente o comportamento de um caso de uso em UML, indo além do diagrama de casos de uso e fornecendo uma descrição textual completa de cada interação e cenário. Assim, os textos estendidos para o trabalho foram:

### 2.5.1. Criar metas financeiras

**Ator principal:** Usuário

**Interessados e interesses:**

- Usuário: Quer estabelecer metas de economia ou investimento para acompanhar o progresso financeiro.

**Pré-condições:**

- O usuário está logado no sistema.
- O usuário possui uma fonte de renda ou saldo inicial no sistema para definir metas.

**Pós-condições:**

- Uma nova meta financeira é criada e salva no sistema.
- O usuário pode visualizar a meta nas seções apropriadas.

**Cenário de sucesso principal:**

1. O usuário acessa a seção "Metas Financeiras" no aplicativo.
2. O sistema exibe uma opção para criar uma nova meta.
3. O usuário clica na opção de criar uma nova meta.
4. O sistema solicita informações como o tipo de meta (economia, investimento), valor alvo, data limite, e prioridades.
5. O usuário preenche as informações e confirma a criação da meta.
6. O sistema salva a nova meta e a exibe na lista de metas.
7. O usuário é redirecionado para a visualização das metas, onde pode acompanhar o progresso.

**Fluxos alternativos:**

(6). Falha na validação dos dados:

Se o usuário inserir um valor inválido (ex.: valor negativo ou data anterior à atual), o sistema exibe uma mensagem de erro e solicita correções antes de prosseguir.

### 2.5.2. Adicionar gastos fixos/variáveis

**Ator principal:** Usuário

**Interessados e interesses:**

- Usuário: Quer registrar despesas para acompanhar melhor o orçamento mensal.

**Pré-condições:**

- O usuário está logado no sistema.
- O sistema possui categorias de gastos fixos e variáveis definidas.

**Pós-condições:**

- O gasto é adicionado à lista de despesas e é contabilizado no orçamento mensal.

**Cenário de sucesso principal:**

1. O usuário acessa a seção "Gastos" no aplicativo.
2. O sistema oferece a opção de adicionar um novo gasto.

3. O usuário escolhe entre "gasto fixo" ou "gasto variável".
4. O sistema solicita informações como nome do gasto, valor, data, e recorrência (para gastos fixos).
5. O usuário preenche as informações e confirma a adição.
6. O sistema registra o gasto e o atualiza no orçamento mensal.
7. O usuário é redirecionado para a tela de visualização dos gastos.

**Fluxos alternativos:**

(5). O usuário não define recorrência para gastos fixos:

O sistema pergunta se o gasto fixo deve ser recorrente e, se o usuário não definir, o sistema assume que é um gasto único e procede com o cadastro.

(5). O usuário cancela o processo:

Se o usuário optar por cancelar antes de confirmar, o sistema descarta as informações e retorna à página anterior.

### 2.5.3. Visualizar histórico de transações

**Ator principal:** Usuário

**Interessados e interesses:**

- Usuário: Quer ter uma visão geral de suas transações financeiras, tanto de entradas quanto de saídas.

**Pré-condições:**

- O usuário está logado no sistema.
- O sistema possui transações financeiras registradas para o usuário.

**Pós-condições:**

- O histórico de transações é exibido ao usuário.
- O usuário pode filtrar e visualizar transações específicas.

**Cenário de sucesso principal:**

1. O usuário acessa a seção "Histórico de Transações" no aplicativo.
2. O sistema exibe uma lista das transações recentes.
3. O usuário pode navegar pela lista ou aplicar filtros (ex.: por data, categoria, valor).
4. O sistema filtra as transações de acordo com os critérios selecionados.
5. O usuário visualiza as transações desejadas com detalhes como data, valor e categoria.
6. O usuário retorna para outras áreas do aplicativo ou fecha a visualização.

**Fluxos alternativos:**

(2). Não há transações registradas:

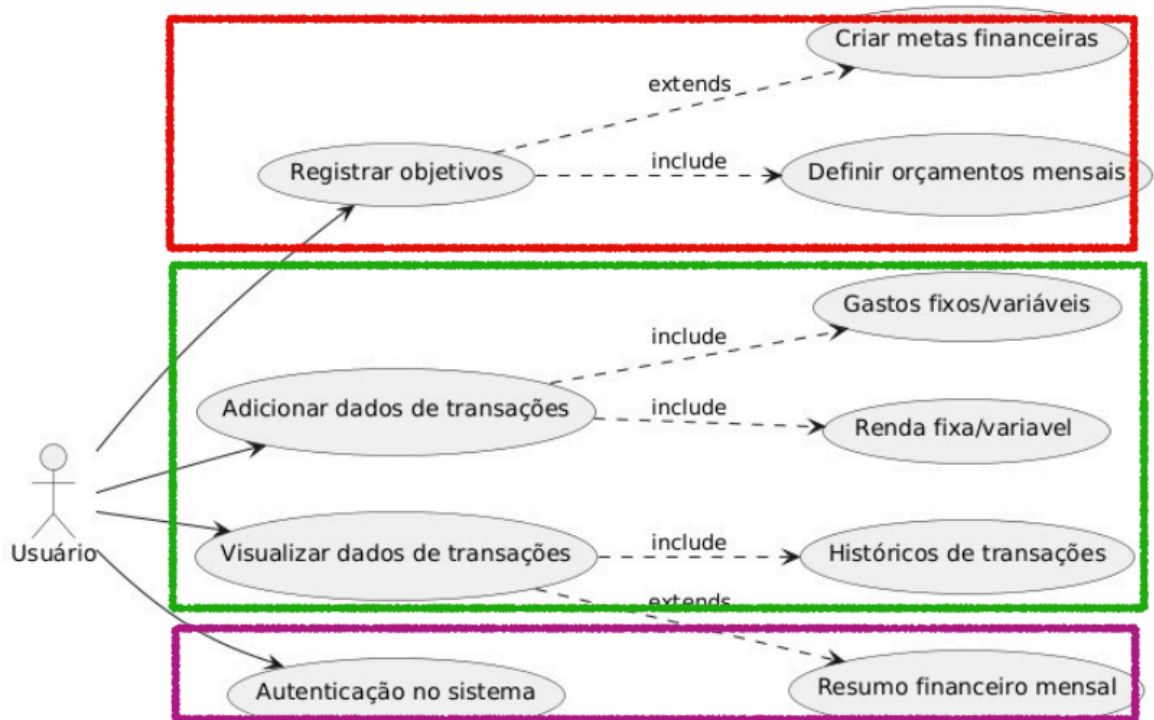
Se o usuário não tiver nenhuma transação registrada, o sistema exibe uma mensagem informando que não há transações e oferece atalhos para adicionar gastos ou registrar entradas financeiras.

(4). Erro ao aplicar filtros:

Se o usuário aplicar filtros que não resultam em nenhuma transação, o sistema exibe uma mensagem indicando que não há resultados e permite redefinir os filtros.

### 3. Arquitetura do Sistema:

#### 3.1. Diagrama de componentes:



- **Vermelho:** Componentes de usuário
- **Verde:** Componentes de transação
- **Roxo:** Componentes de autenticação

#### 3.2. Descrição dos Componentes do sistema:

##### 3.2.1. Componentes Explícitos

- **Componente de Gestão Autenticação**
  - **Grupo de Casos de Uso:** Grupo 1 (Informações de segurança)
  - **Casos de Uso Sob sua Responsabilidade**
    - Cadastro de usuário
    - Login de usuário
    - Edição de dados de usuário
    - Controle do banco de dados
    - Autorização para acesso seguro aos dados do usuário
- **Componente de Gestão de Usuário**
  - **Grupo de Casos de Uso:** Grupo 2 (Informações de Usuário)
  - **Casos de Uso Relacionados:**
    - Cadastro de metas
    - Atualização do banco de dados
    - Cálculo da proximidade das metas estabelecidas
    - Sugestão de aumentar/diminuir reserva
- **Componente de Gestão de Transações**
  - **Grupo de Casos de Uso:** Grupo 3 (Informações de Transação)
  - **Casos de Uso Sob sua Responsabilidade:**
    - Cadastro de transação
    - Atualização do banco de dados
    - Inserção de investimentos

- Consulta de histórico de transações
- Trabalho em conjunto com o grupo 2

### 3.2.2. Componentes Implícitos

- **Componente de Backup e Recuperação**
  - **Descrição:** Sistema de backup de dados e recuperação em caso de falhas
  - **Justificativa:** Necessário para garantir a integridade e a segurança dos dados do sistema
- **Interface de Usuário (UI/UX)**
  - **Descrição:** Interface gráfica para preenchimento de formulários, visualização de gráficos de gastos, metas e históricos.
  - **Justificativa:** Necessário para o usuário interagir com o sistema.
- **Middleware de Integração com APIs**
  - **Descrição:** Plataformas de investimento e serviços de autenticação de terceiros (ex: APIs de investimento).
  - **Justificativa:** Necessário para comunicação com sistemas externos
- **Mecanismo de Notificações**
  - **Descrição:** Envio de notificações automáticas.
  - **Justificativa:** Necessário para alertar sobre metas atingidas, transações relevantes ou avisos de segurança.
- **Segurança de Dados e Criptografia**
  - **Descrição:** Criptografia de dados sensíveis, como transações financeiras e dados de login.
  - **Justificativa:** Necessário para garantir a privacidade e segurança.

## 4. Modelo de Classes

### 4.1. Descrição dos Componentes do sistema

Após algumas iterações ao implementar o sistema gerado pelo modelo unificado, as classes finais geradas, fazendo parte de cada componente, é descrita abaixo.

#### 4.1.1. Componente de Usuário:

- **Classe User:** Representa o usuário do sistema, com atributos como nome completo, CPF, senha (armazenada de forma segura), avatar, metas financeiras e datas de criação/atualização do perfil.
- **Classe Goal:** Representa as metas financeiras associadas ao usuário, incluindo nome, valor-alvo, valor acumulado, prazo, tipo de meta, status e data de criação.
- **Classe Preferences:** Armazena as preferências do usuário, como notificações, idioma, tema (dark mode ou light mode) e moeda padrão.

### 4.1.2. Componente de Transação:

- **Classe Transaction:** Representa uma transação financeira com atributos como valor, data, descrição, tipo (renda, despesa ou investimento), categoria e referência a um histórico de transações.
- **Classe History:** Centraliza o histórico de transações do usuário, contendo detalhes como período (data de início e término), soma de rendas, despesas e economia líquida.
- **Classe Budget:** Representa o orçamento do usuário, com um limite de gastos definido e associado às transações.

### 4.1.3. Componente de Autenticação:

- **Classe Auth:** Gerencia a autenticação do usuário, incluindo credenciais, verificação de e-mail, hash de senha, token JWT e histórico de login.
- **Classe Recovery:** Controla o processo de recuperação de senha, com atributos como token de recuperação, validade e referência ao usuário que realizou o pedido.

## 4.2. Identificação das propriedades ou atributos das Classes

Descritas as classes finais acima, os atributos associados às classes são mostrados abaixo.

### 4.2.1. Componente de Usuário

- **User (Usuário):**
  - **id** (UUID): Identificador único do usuário.
  - **name** (string): Nome completo do usuário.
  - **cpf** (string): CPF do usuário.
  - **password** (string): Senha (armazenada de forma segura, ex. hash).
  - **goals** (array of **Goal**): Metas financeiras associadas ao usuário.
  - **avatar** (string): URL para a imagem de perfil.
  - **createdAt** (timestamp): Data de criação do perfil.
  - **updatedAt** (timestamp): Última atualização dos dados.
- **Goal (Meta Financeira):**
  - **id** (UUID): Identificador único da meta.
  - **userId** (UUID): Identificador único do usuário autenticado.

- **name** (string): Nome da meta (ex: "Economizar para viagem").
  - **targetAmount** (float): Valor que o usuário deseja atingir.
  - **currentAmount** (float): Valor atual acumulado.
  - **deadline** (date): Prazo limite para atingir a meta.
  - **type** (enum: curto prazo, médio prazo, longo prazo): Tipo de meta com base no prazo.
  - **status** (enum: ativa, concluída, cancelada): Estado atual da meta.
  - **createdAt** (timestamp): Data de criação da meta.
- **Preferences**(**Preferências do usuário**):
    - **userId** (UUID): Identificador único do usuário autenticado.
    - **notifications**(boolean): se o usuário quer receber notificações.
    - **language** (string): adapta a linguagem para o usuário.
    - **Theme** (string): Dá a opção de dark mode ou light mode.
    - **Currency** (string): Dá a opção de mudar a moeda.

#### 4.2.2. Componente de Transação

- **Transaction** (**Transação**):
  - **id** (UUID): Identificador único da transação.
  - **type** (enum: renda, despesa, investimento): Tipo de transação.
  - **amount** (float): Valor da transação.
  - **date** (date): Data da transação.
  - **description** (string): Descrição da transação (ex: "Salário", "Compra de ações").
  - **category** (string): Categoria da transação (ex: "Alimentação", "Investimento").
  - **historyId** (UUID): Identificador do history em que a transação está registrada.
  - **createdAt** (timestamp): Data de criação da transação.
- **History** (**Histórico de Transações**):
  - **id** (UUID): Identificador único do histórico (caso seja necessário manter várias versões).
  - **userId** (UUID): Identificador único do usuário autenticado.

- **transactions** (array of **Transaction**): Lista de transações associadas ao usuário.
- **startDate** (date): Data de início do período.
- **endDate** (date): Data de término do período.
- **totalIncome** (float): Soma total de todas as rendas do período.
- **totalExpenses** (float): Soma total de todas as despesas do período.
- **netSavings** (float): Diferença entre renda e despesas no período.
- **Budget (Orçamento):**
  - **id** (UUID): Identificador único do orçamento.
  - **limit** (float): Limite de gastos definido pelo usuário.
  - **userId** (UUID): Identificador do usuário.

#### 4.2.3. Componente de Autenticação

- **Auth (Autenticação):**
  - **id** (UUID): Identificador único da autenticação.
  - **userId** (UUID): Identificador único do usuário autenticado.
  - **email** (string): E-mail usado para autenticação.
  - **emailVerified** (boolean): Boolean para saber se o email está verificado.
  - **passwordHash** (string): Hash da senha para validação de credenciais.
  - **jwtToken** (string): Token de sessão gerado no login.
  - **lastLogin** (timestamp): Última vez que o usuário fez login.
  - **loginAttempts** (integer): Contagem de tentativas de login.
- **Recovery (Recuperação de Senha):**
  - **id** (UUID): Identificador único do pedido de recuperação.
  - **userId** (UUID): Identificador único do usuário que solicitou a recuperação.
  - **recoveryToken** (string): Token de recuperação enviado ao usuário.
  - **expiresAt** (timestamp): Data e hora de expiração do token de recuperação.

### 4.3. Visibilidade de atributos

A maioria dos atributos das nossas classes são privadas.

No componente de autenticação, todos os atributos das classes Auth e Recovery serão privados, para evitar acesso direto e promover segurança. Outros atributos financeiros como `totalIncome` e `totalExpenses` na classe Budget também serão mantidos privados para evitar manipulação direta.

Apenas métodos ou atributos específicos que precisam ser acessíveis externamente terão visibilidade pública, como métodos que retornam informações gerais do usuário ou métodos de leitura para atributos que não apresentam riscos de segurança. Os atributos públicos serão listados a seguir:

- **User (Usuário):**
  - `id` (UUID): Identificador único do usuário.
- **Goal (Meta Financeira):**
  - `id` (UUID): Identificador único da meta.
  - `name` (string): Nome da meta, podendo ser exibido publicamente para facilitar a identificação do objetivo financeiro.
- **Preferences (Preferências do usuário):**
  - `notifications` (boolean): Preferência por notificações, permitindo ativar/desativar notificações publicamente.
  - `language` (string): Idioma preferido do usuário, para personalização da interface.
- **Transaction (Transação):**
  - `id` (UUID): Identificador único da transação.
- **History (Histórico de Transações):**
  - `id` (UUID): Identificador único do histórico (caso seja necessário manter várias versões).
  - `startDate` (date): Data de início do histórico, que pode ser exibida publicamente para indicar o período registrado.
  - `endDate` (date): Data de término do histórico, exibida publicamente para marcar o período de um relatório.
- **Budget (Orçamento):**
  - `id` (UUID): Identificador único do orçamento.
  - `limit` (float): Limite do orçamento, que pode ser exibido publicamente para indicar restrições de gastos.

### 4.4. Relacionamentos e Multiplicidade

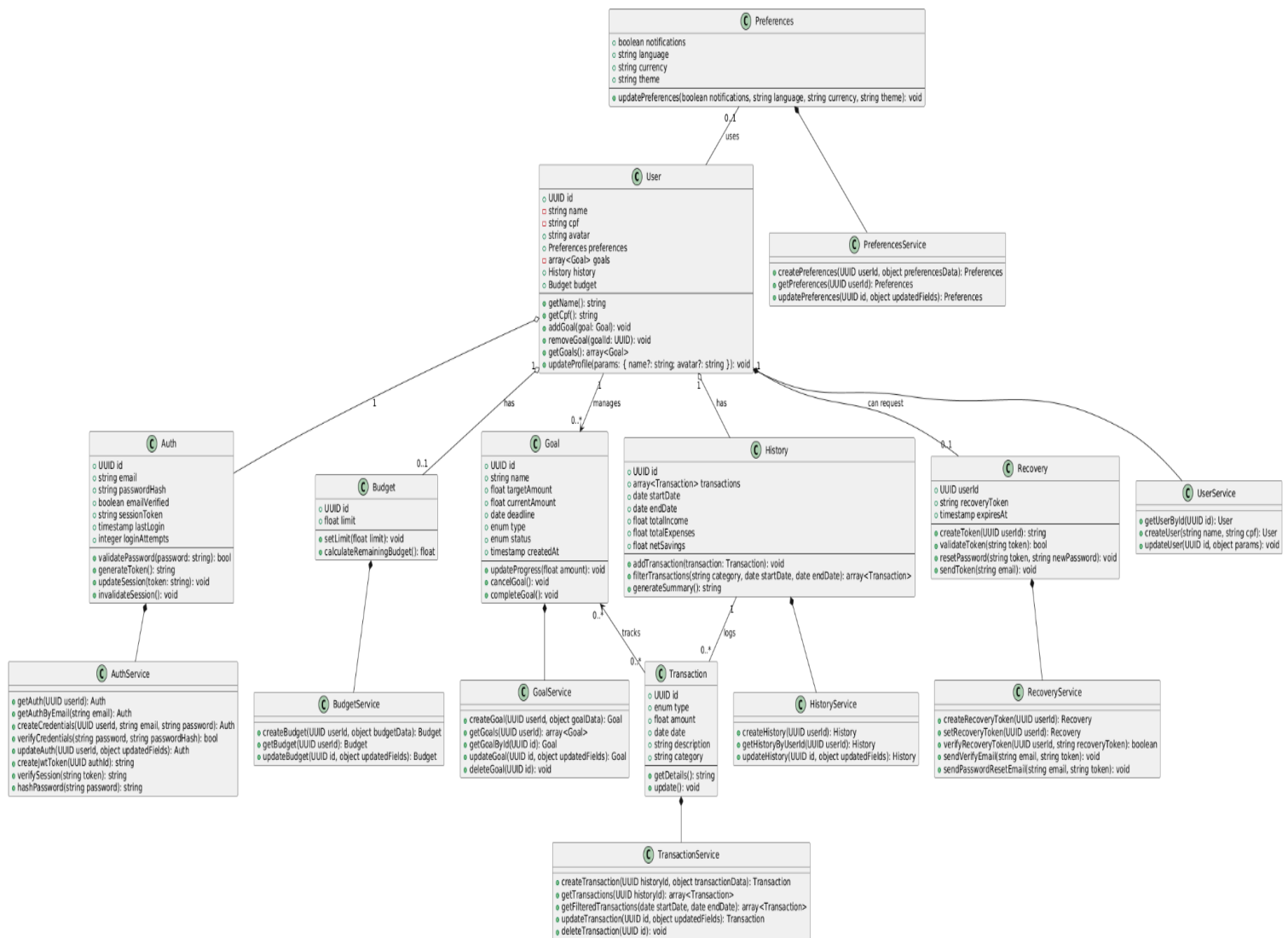
A tabela com os relacionamentos e suas multiplicidades é mostrada a seguir. Vale destacar que foram criadas classes auxiliares (services/controllers) para ser a ponte com o banco de dados.

Classe Primária	Classe	Tipo de	Multiplicidade
-----------------	--------	---------	----------------



	Associada	Relacionamento	
User	Goal	Associação	1 – 0..*
User	Transaction	Associação	1 – 0..*
User	Auth	Associação	1 – 1
User	Recovery	Associação	1 – 0..1
History	Transaction	Agregação	1 – 0..*
User	Preferences	Composição	1 – 1
User	Budget	Composição	1 – 0..1
AuthService	Auth	Composição	1 – 1
BudgetService	Budget	Composição	1 – 1
GoalService	Goal	Composição	1 – 1
HistoryService	History	Composição	1 – 1
PreferencesService	Preferences	Composição	1 – 1
RecoveryService	Recovery	Composição	1 – 1
TransactionService	Transaction	Composição	1 – 1
UserService	User	Composição	1 – 1

#### 4.5. Diagrama de Classes Atualizado:



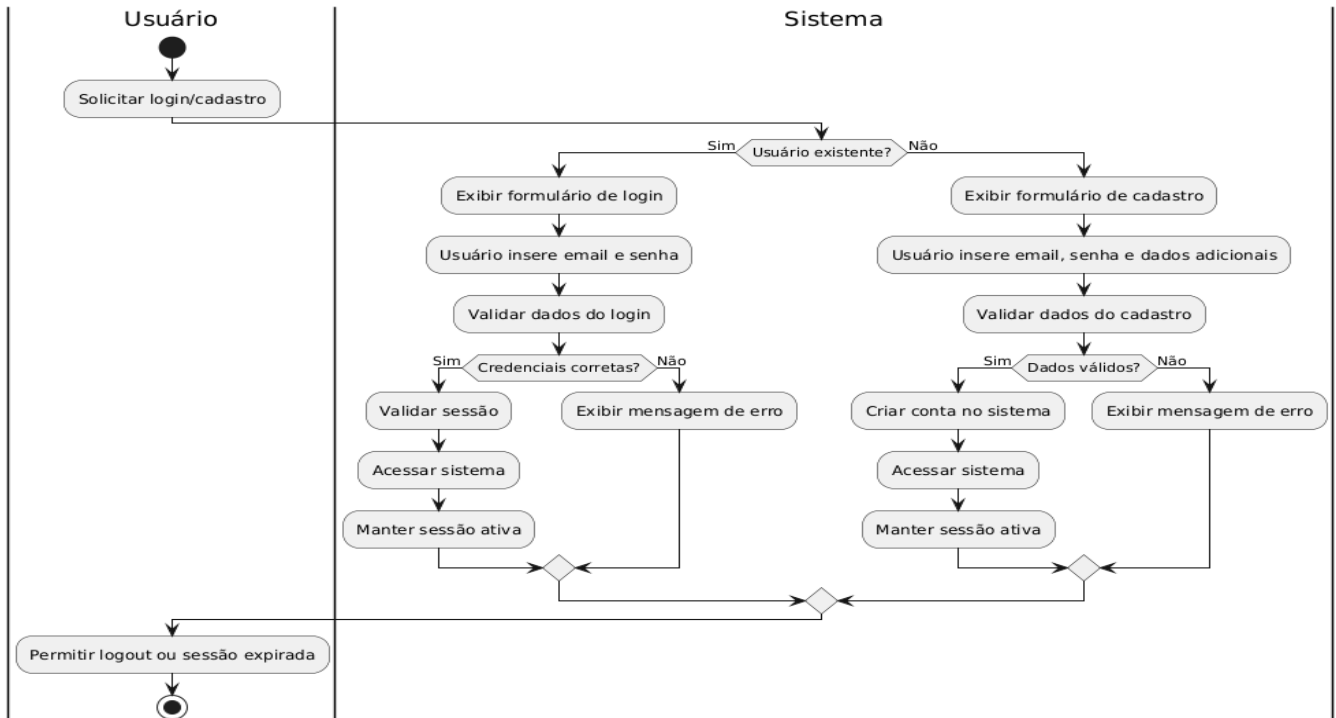
## 5. Modelo de comportamento

Os diagramas de atividades representam aspectos dinâmicos das estruturas do sistema, sendo utilizados na fase de levantamento de requisitos para modelar processos e ajudar a refinar casos de uso e na fase de design de software para identificar fluxos de processos dentro do sistema, possuindo, assim, um maior detalhe de comportamento. A seguir, será detalhado os diagramas de atividades elaborados para o projeto:

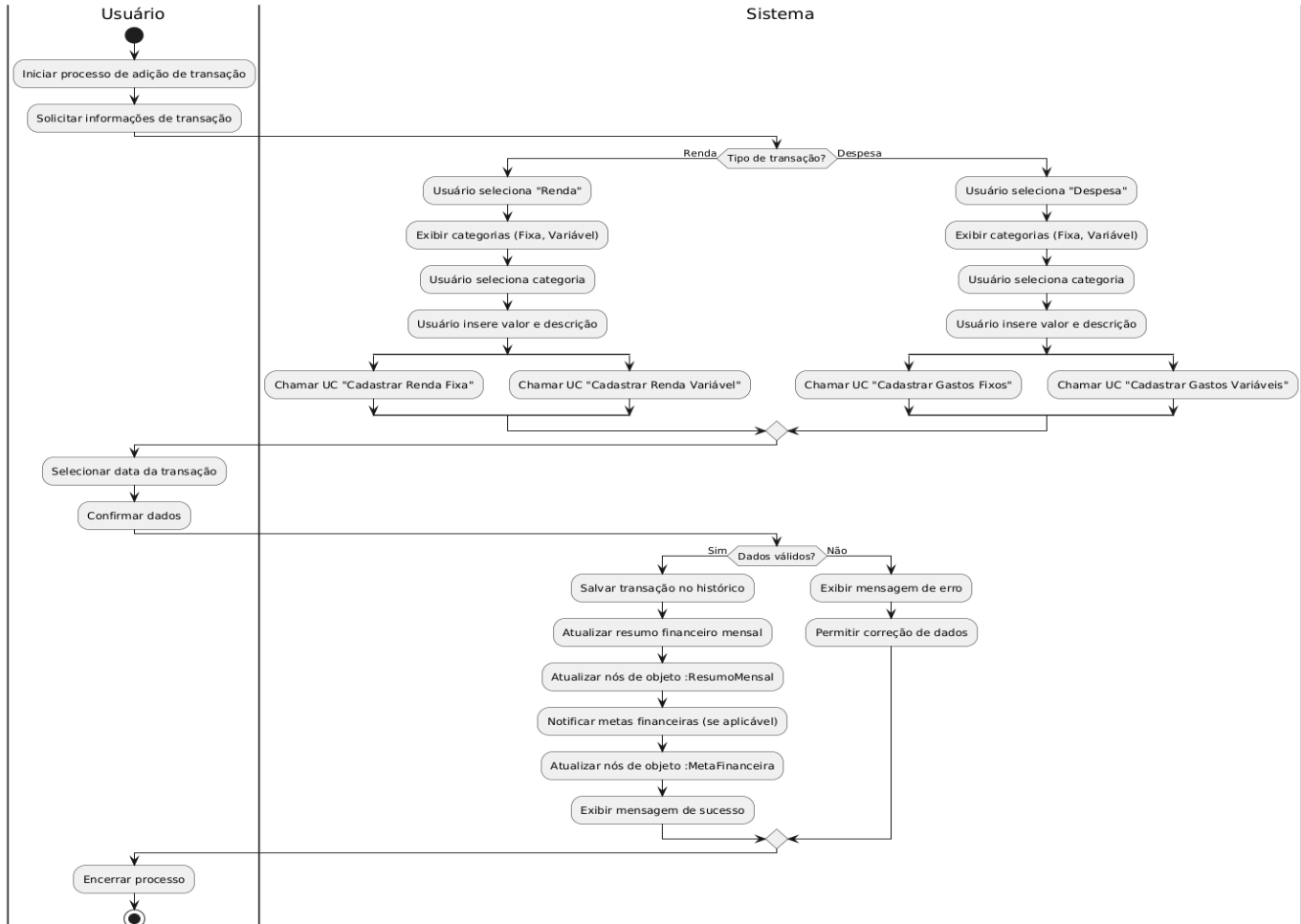
### 5.1. Diagrama de atividades do sistema com partições

Apesar de possuir apenas um ator para o diagrama de casos de uso, apresentamos possíveis maneiras de visualizar os diagramas de atividades com partições a partir da separação entre Usuário e Sistemas, de modo que podem ser visualizados em sequência:

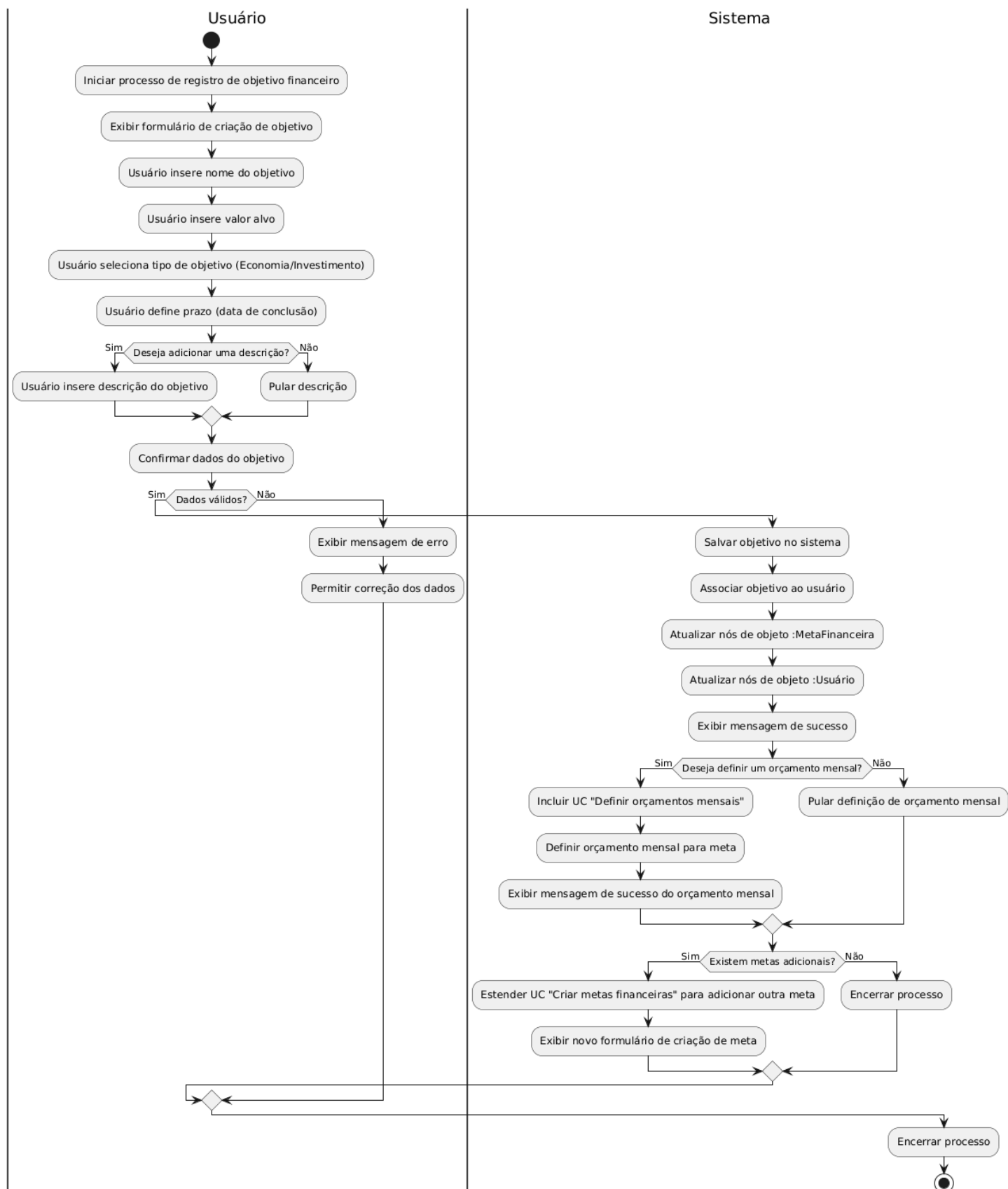
### 5.1.1. Diagrama de Atividade para autenticação do usuário:



### 5.1.2. Diagrama de Atividade para adicionar dados de transações:



### 5.1.3. Diagrama de Atividades para registrar objetivos:

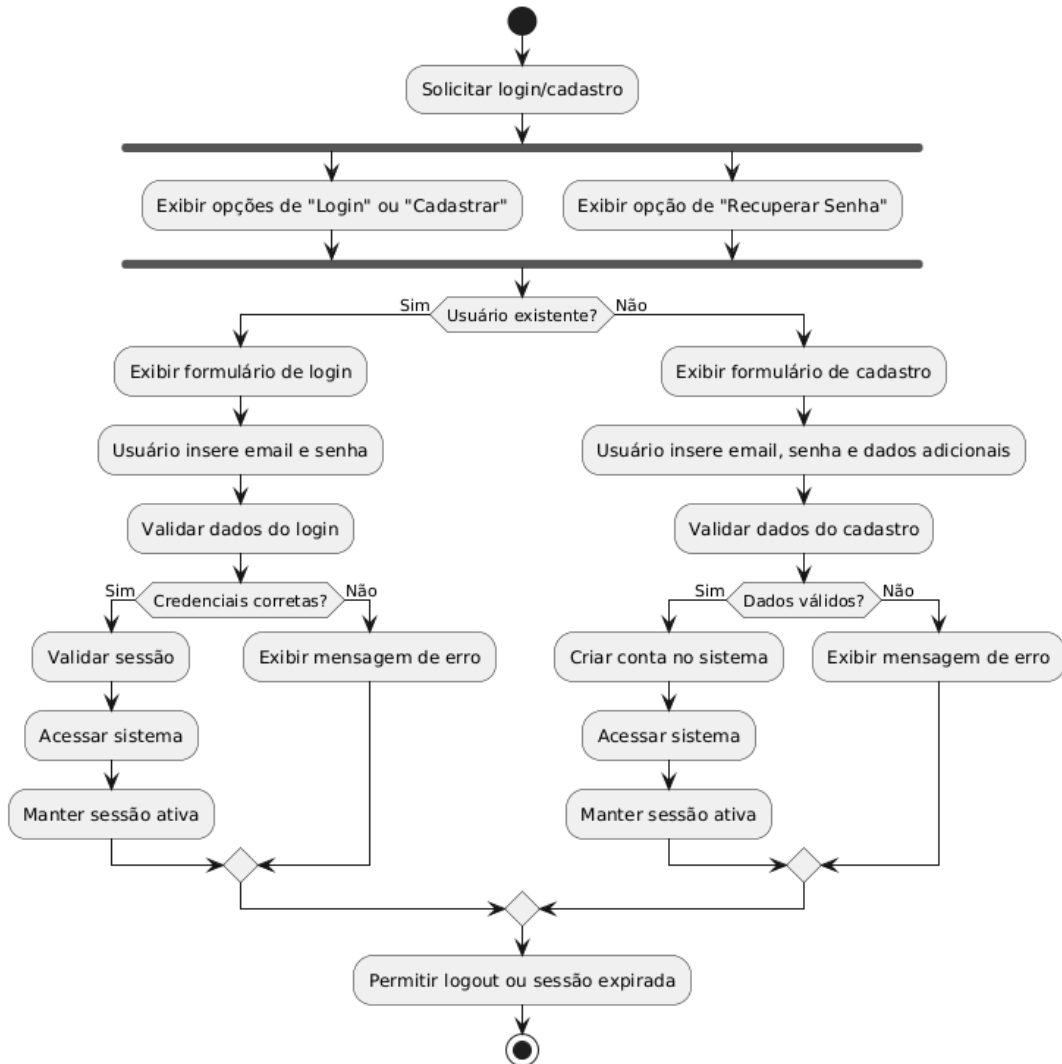


## 5.2. Diagrama de atividades do sistema

A seguir serão apresentados três diagramas relacionados aos casos de uso principais detalhadas no tópico de casos de uso:

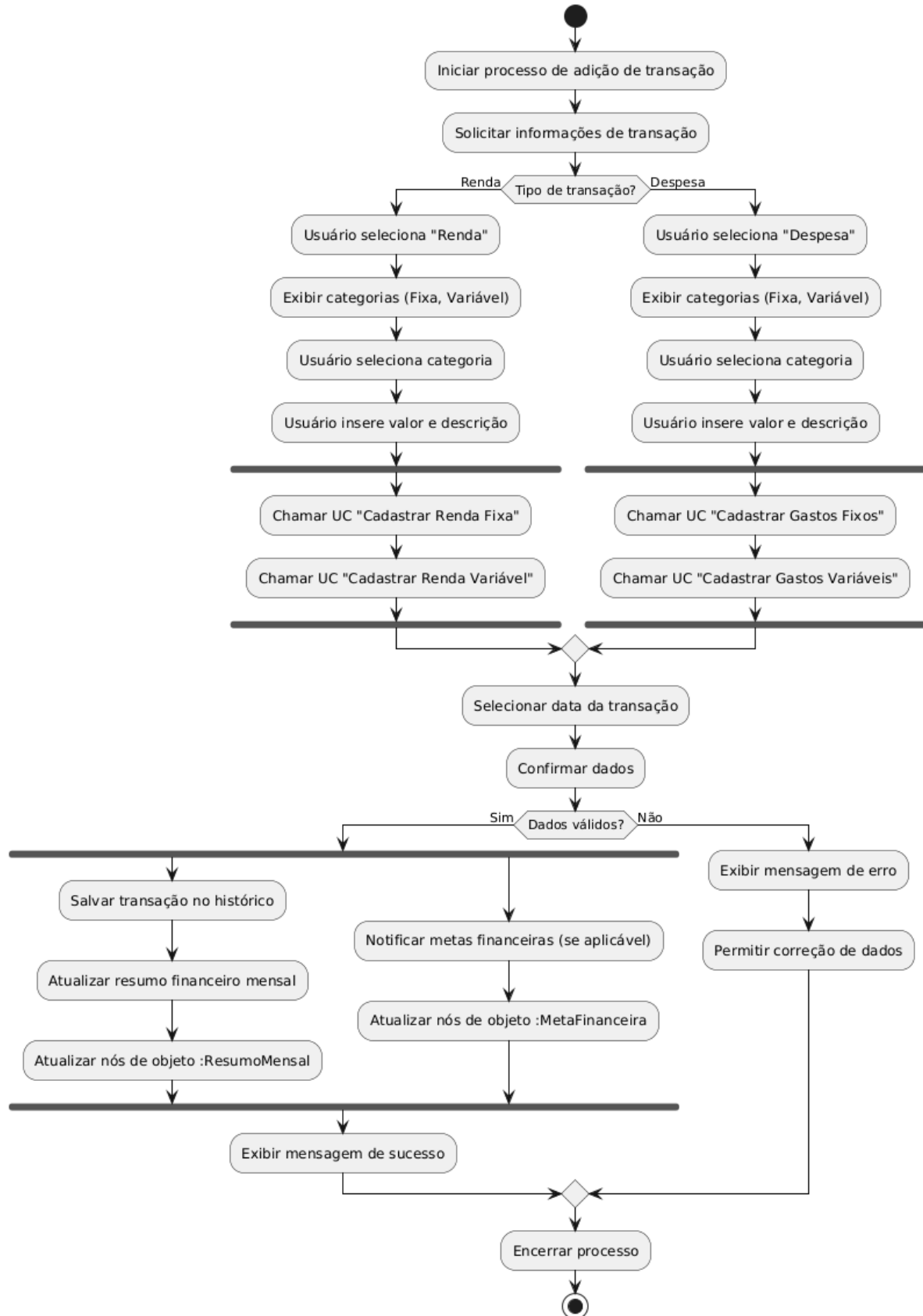
### 5.2.1. Diagrama de Atividade para autenticação do usuário:

- Para nossa aplicação, a autenticação do sistema não possui Caso de Uso que seja estendido.



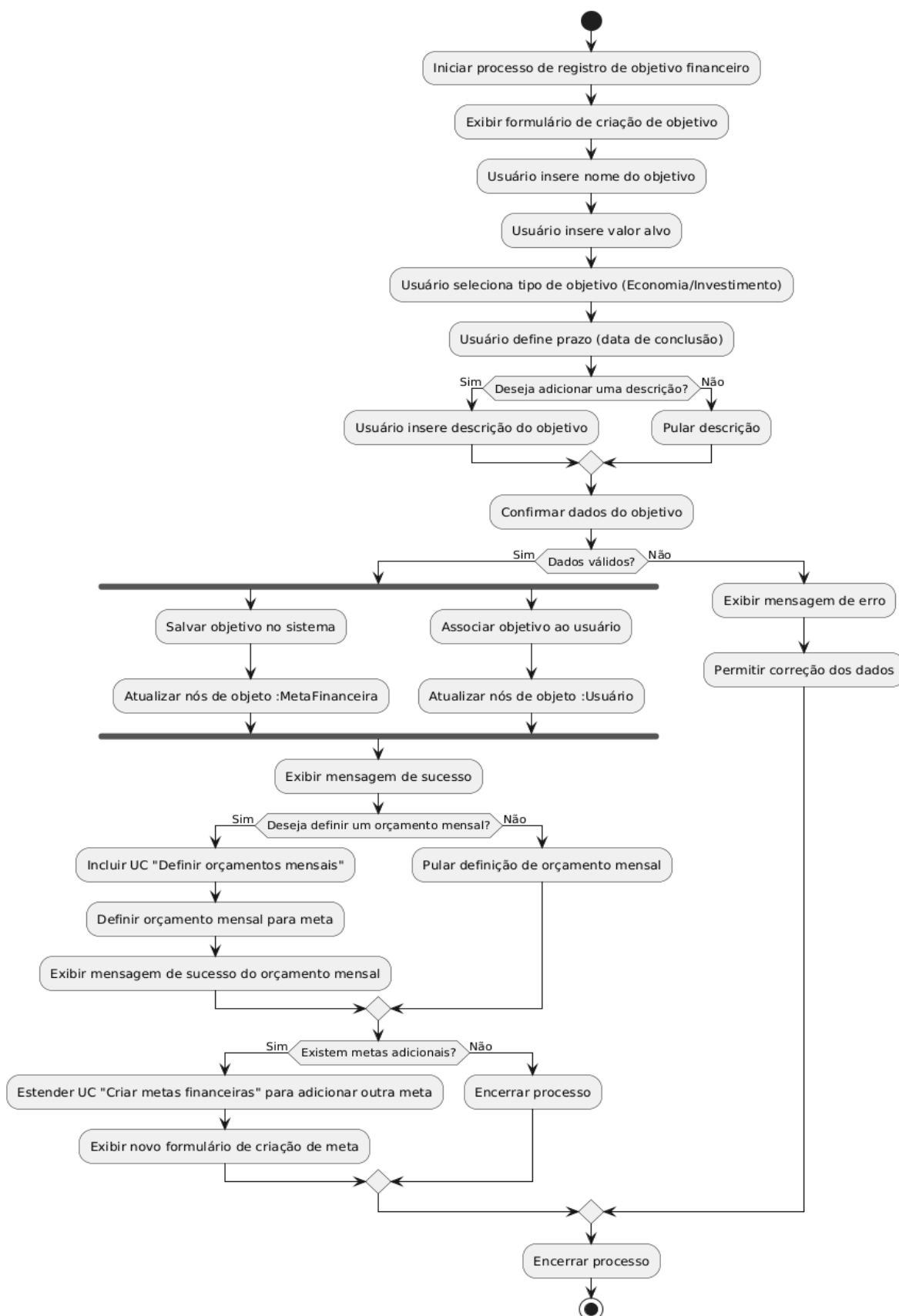
### 5.2.2. Diagrama de Atividade para adicionar dados de transações:

- Os casos de uso são para o cadastro de rendas e gastos fixos/variáveis.



### 5.2.3. Diagrama de Atividades para registrar objetivos:

- O caso de uso definido como <include> é o “Definir orçamentos mensais” e o caso de uso <extend> é dado pelo “Criar metas financeiras”.



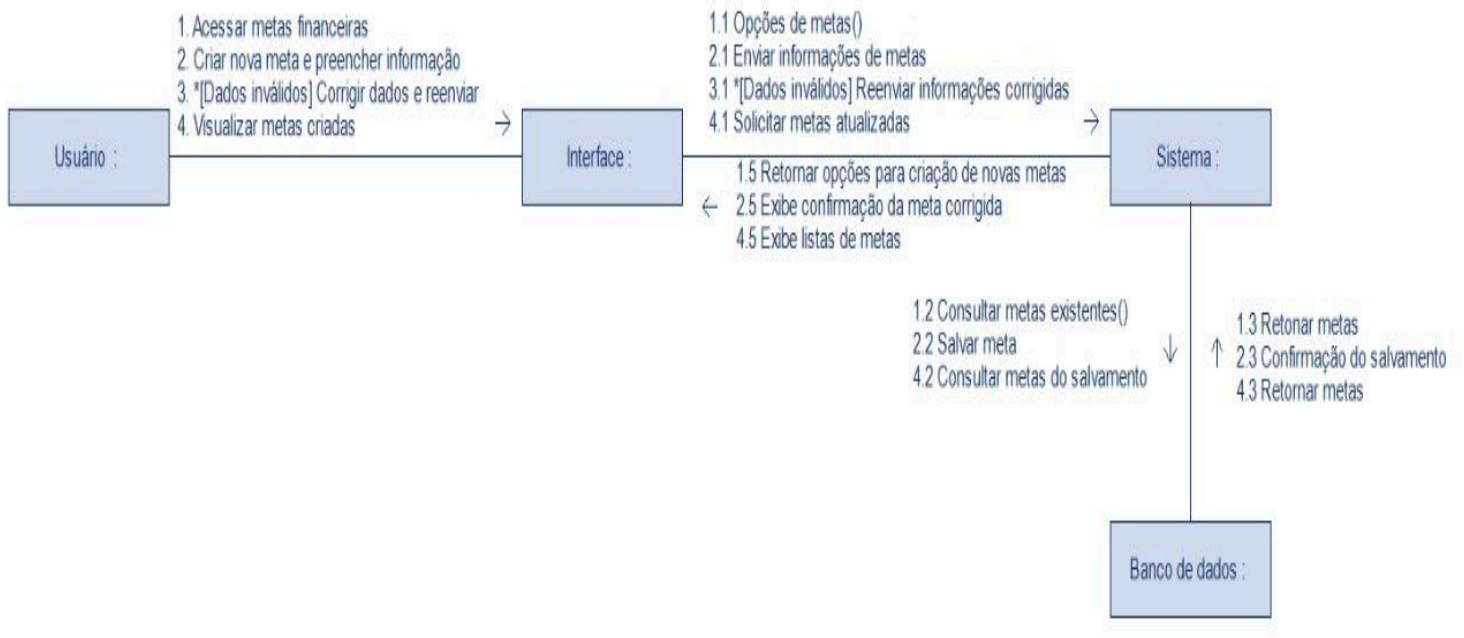
## 6. Modelos de Interações

Os diagramas de interação possuem o objetivo de especificar como os elementos de software nos diferentes níveis de abstração, como arquitetura e projeto interno de componente, precisam trocar informações entre eles para desempenhar as funcionalidades e comportamentos esperados no sistema.

A seguir, será exibido os diagramas de comunicação e os diagramas de sequências, os quais focam em representar as interações entre os objetos participando de um processo, modelados para o projeto de acordo com os diagramas de atividades apresentados no tópico anterior.

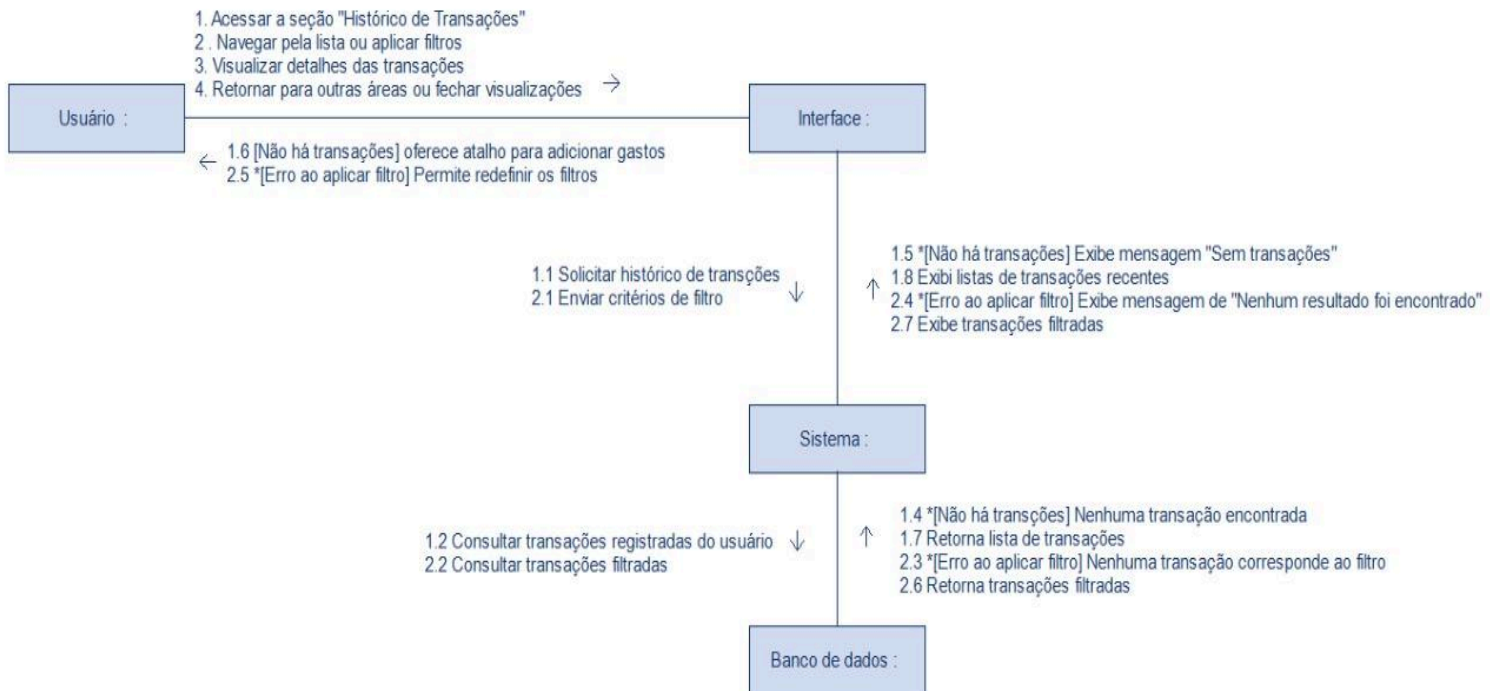
### 6.1. Diagramas de Comunicação

#### 6.1.1. Diagrama de Sequência para Criação de metas ou objetivos:

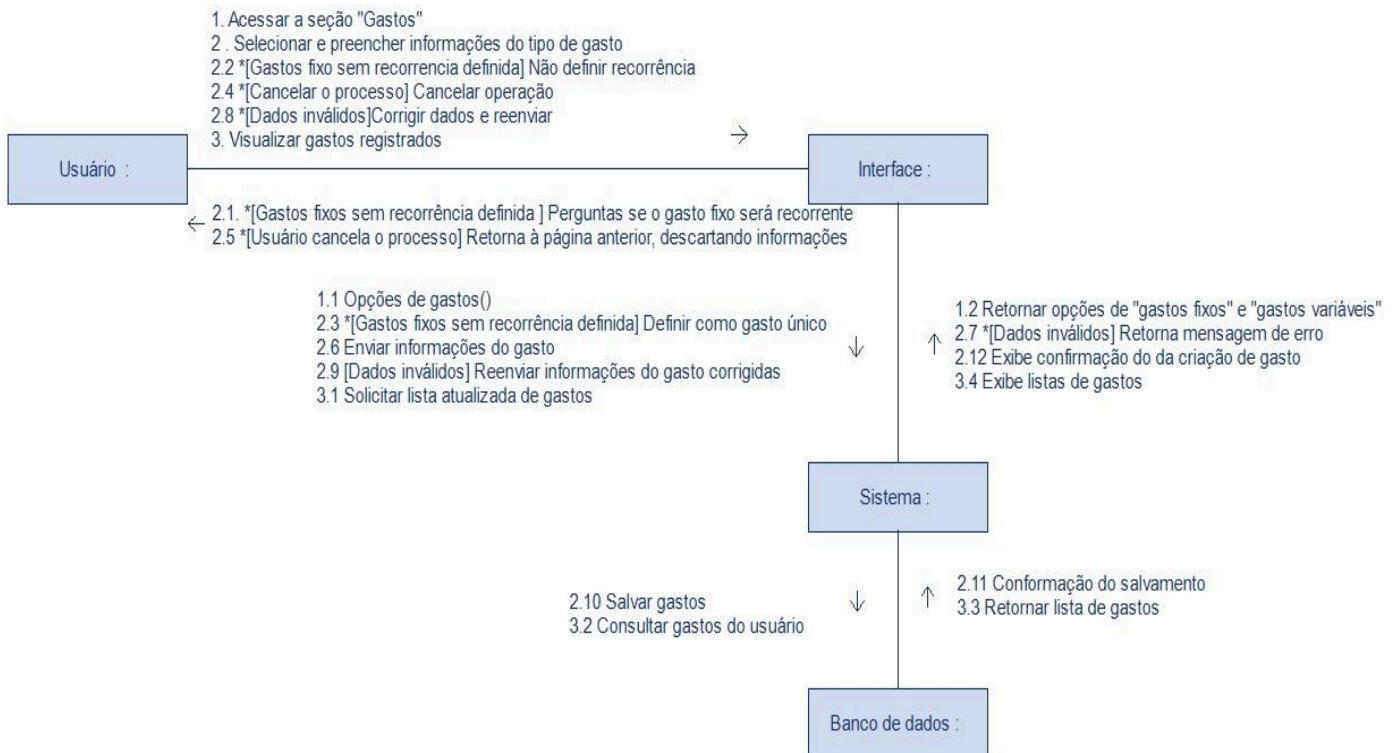




### 6.1.2. Diagrama de Atividades para adicionar dados de transações:

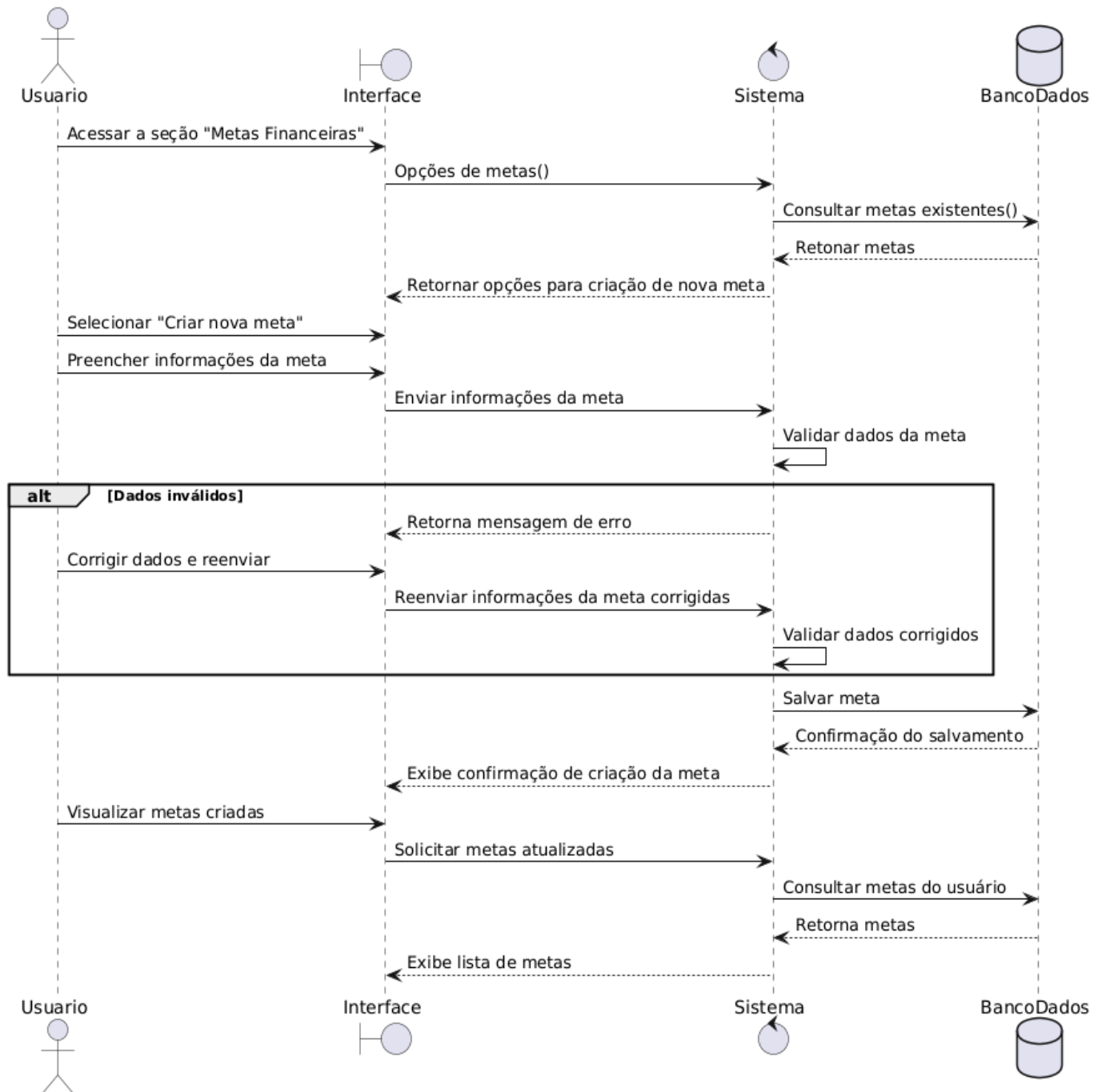


### 6.1.3. Diagrama de Atividades para registrar controle de gastos:

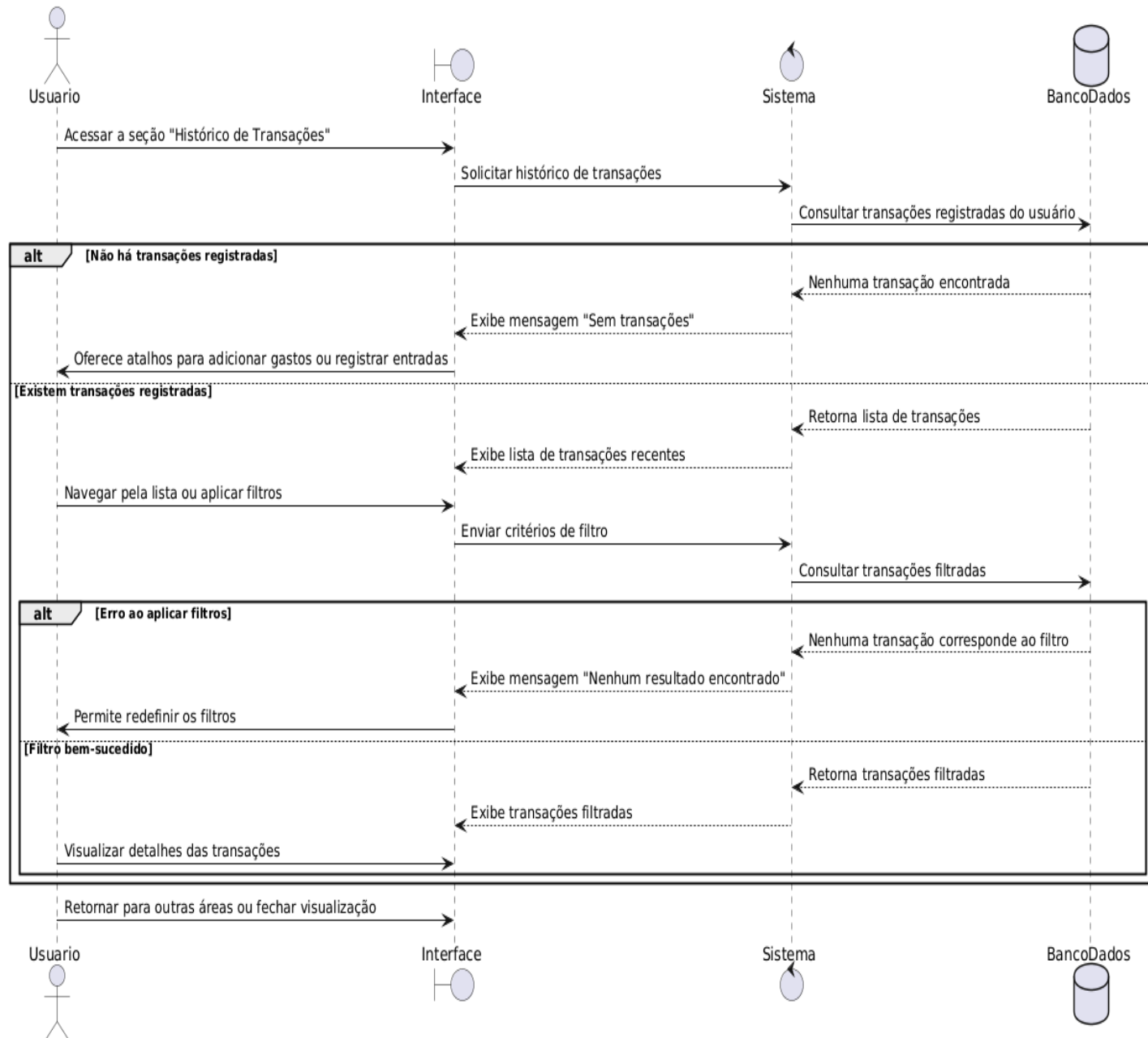


## 6.2. Diagramas de Sequência

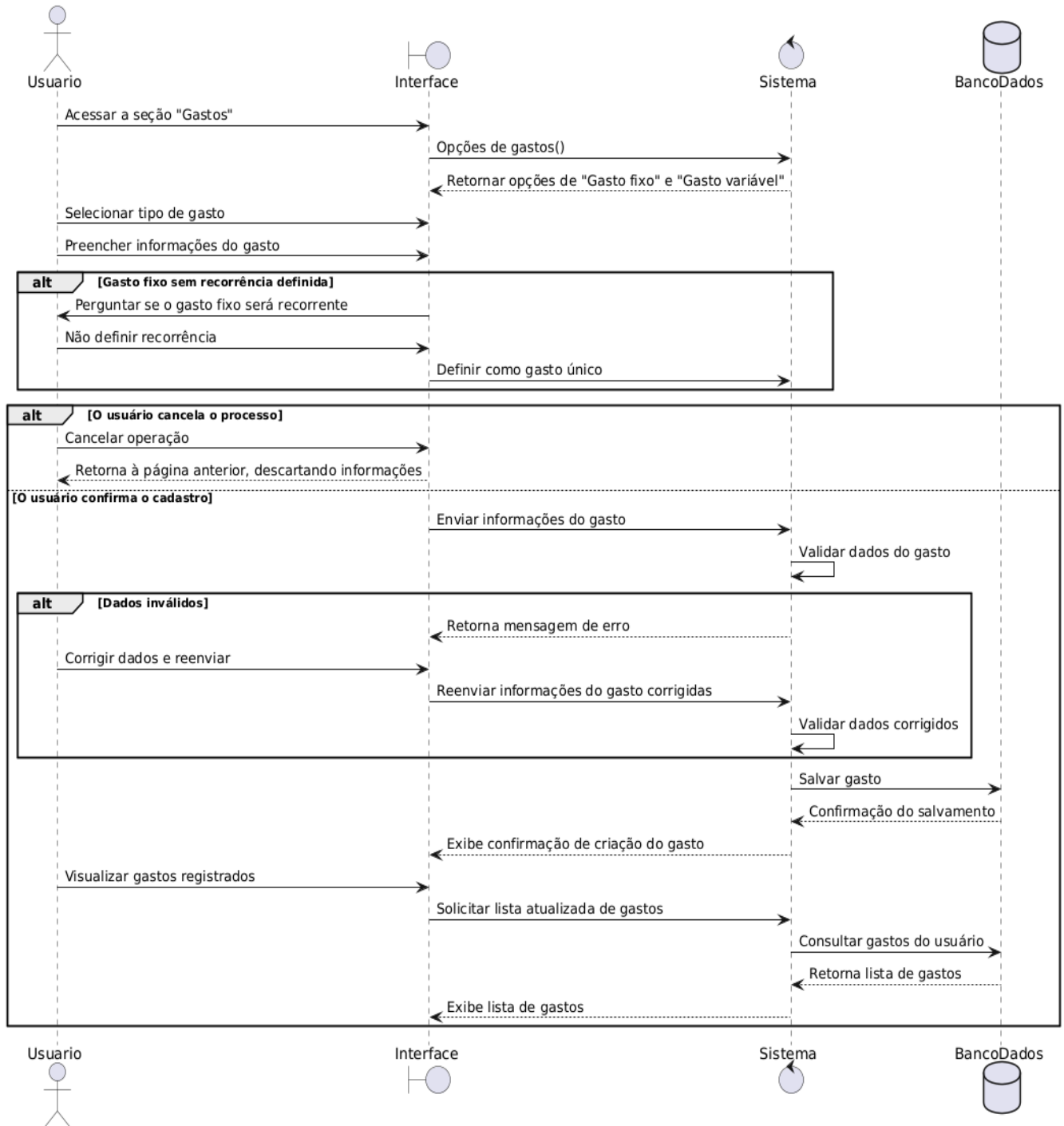
### 6.2.1. Diagrama de Sequência para Criação de metas ou objetivos:



### 6.2.2. Diagrama de Atividades para adicionar dados de transações:



### 6.2.3. Diagrama de Atividades para registrar controle de gastos:



## 7. Código do Sistema

Para a implementação do sistema, conforme pedido pela professora, o principal caso de uso implementado corresponde às transações (adição, visualização, etc).

Porém alguns outros casos de uso foram implementados também, como a autenticação.

## 7.1. Descrição da Tecnologia Selecionada

Nesse projeto, foram utilizadas as tecnologias de Next.js, Prisma e PostgreSQL.

Para executar o projeto, usamos docker com docker compose. Abaixo estão as descrições resumidas de cada tecnologia:

### 7.1.1. Next.js:

Um framework poderoso baseado em React, projetado para criar aplicações web escaláveis e de alto desempenho. Ele oferece recursos como renderização híbrida (SSR e SSG), geração de páginas estáticas, roteamento dinâmico, otimização automática e suporte integrado a APIs, tornando-o ideal para desenvolvimento full-stack.

### 7.1.2. Prisma:

Um ORM (Object-Relational Mapping) de última geração que simplifica o trabalho com bancos de dados. Ele fornece um cliente de consulta intuitivo e baseado em tipos, permitindo interagir com os dados de forma segura, eficiente e alinhada ao TypeScript. Ideal para integrações com frameworks modernos, como Next.js.

### 7.1.3. PostgreSQL:

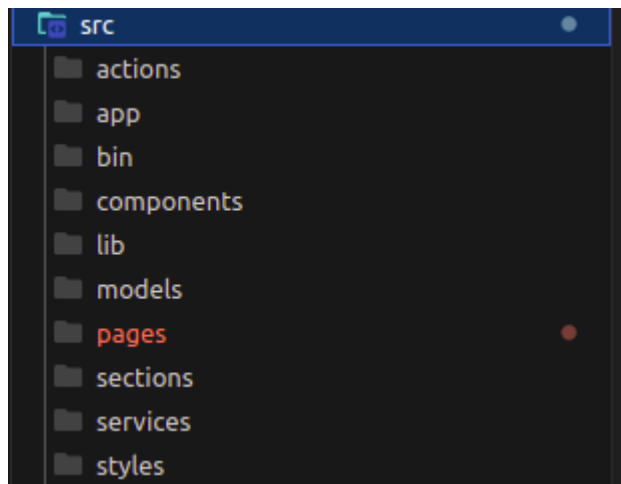
Um dos sistemas de banco de dados relacional mais avançados e amplamente utilizados no mundo. Conhecido por sua robustez, extensibilidade e suporte a funcionalidades avançadas, como dados geoespaciais, JSON, índices personalizados e transações ACID, é uma escolha confiável para aplicações que exigem desempenho e escalabilidade.

### 7.1.4. Docker e Compose

Usamos Docker juntamente com docker-compose para subir a aplicação sem se preocupar com a plataforma e com dependências. Assim, para testar a plataforma basta ter docker e docker compose instalado, e o sistema já sobe uma instância do PostgreSQL e também já inicializa o cliente do Prisma, assim como suas *migrations*. Em seguida, a aplicação Next é iniciada de fato.

## 7.2. Print da estrutura do projeto

7.2.1. Primeiro, vamos mostrar a estrutura do projeto em pastas:



O projeto tem algumas camadas:

1. Pages
  - a. Páginas de fato (todos os diretórios e arquivos exceto o **api/**)
    - i. Implementação do front end em si, em React com Typescript e Tailwind.
  - b. API
    - i. Tudo o que está dentro da pasta **api/**. O sistema de roteamento do *nextjs* nos permite *hostear* rotas desta forma. Serve para estabelecer a comunicação entre o front e o back.
2. Actions
  - a. Actions são funções que são forçadas a ser executadas no lado do servidor, para impedir que o banco de dados seja acessado do lado do cliente (navegador)
3. Models
  - a. Camada responsável por desacoplar a injeção de dependências da plataforma, ela abstrai a comunicação entre a aplicação e o banco de dados usando a modelagem de classes das informações que a plataforma usa de forma abstrata.
4. Services
  - a. Camada responsável por implementar a injeção de dependências. No caso do projeto, a única injeção é a do banco de dados, o qual implementamos com o Prisma. Ou seja, a camada de *services* pode ser vista como responsável por alterar o banco de dados (da perspectiva das outras camadas, pois o Prisma consegue abstrair ainda mais).

### 7.3. Link para o GitHub

<https://github.com/shogoshima/financial-flow-app.git>

### 7.4. Código correspondente aos casos de uso desenvolvidos

Para desenvolver os códigos correspondentes ao caso de uso, trabalhou-se em algumas etapas, todas dentro da pasta **src/**:

1. Desenvolvimento das classes principais: diretório **models/**
2. Desenvolvimento das classes auxiliares (services/controllers): diretório **services/**
3. Desenvolvimento do frontend e rotas de api: diretório **pages/**
4. Vale mencionar a camada de *actions*, que, embora não estejam presentes nos diagramas, viraram uma dependência do ecossistema Nextjs, justamente para executar as funções no lado do servidor. Isso se dá porque o Prisma não pode ser executado no Browser (do lado do cliente). Esta camada está no diretório **actions/**

O caso de uso escolhido para ser implementado foi a consulta e **adição de transações**. Portanto, embora o banco de dados esteja modelado com todas as informações citadas nos diagramas e a camada de **service** implemente as regras de negócios para estas informações, apenas as **Transactions** foram integradas ao front-end.

Vale ressaltar que, como a disciplina não é sobre desenvolvimento web diretamente, não implementamos fatores de segurança que seriam essenciais em um sistema real. Por exemplo, não há validação de email, senha, cpf, etc. Além disso, devido a conflitos de dependência do Nextjs, optamos por simular o JWT, salvando o id do usuario logado ao invés do token de fato.

Para explicar melhor as regras de negócio da nossa estrutura, vamos mostrar como o fluxo de chamadas acontece com as Transações.

Ao entrar na página /dashboard (após o devido login) o usuário se depara com esta tela:

**Welcome, José Carlos Andrade do Nascimento**

[Profile](#) [Goals](#) [Budget](#) [Transactions](#) [Log out](#)

### Add Transaction

Type:

Amount:

Date:

Description:

Category:

Add Transaction

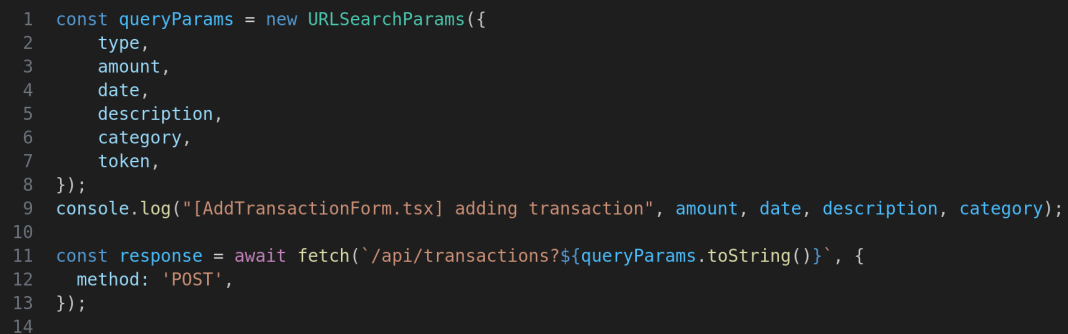
### Transactions

2024-12-16	ultima - bolsa	R\$ 700
2024-12-16		R\$ 10000

Caso não esteja logado, o usuário será forçado a fazer login ou cadastro na plataforma.

**Prints relacionados as camadas usadas. As seguintes capturas de telas foram feitas usando a extensão SnapCode do VSCode:**

**[Front end]:** Dessa forma, o usuário pode consultar as transações registradas na plataforma, assim como adicionar uma nova com as informações mostradas na tela. Após clicar em “Add Transaction”, o front end faz uma **requisição POST** para o backend (**nextjs**) na rota “**/transactions**” com os devidos parâmetros de URL. -> **[API]** :

A screenshot of a code editor window with a dark background and light-colored text. The code is written in TypeScript and is part of a file named AddTransactionForm.tsx. It shows the creation of a URLSearchParams object with properties type, amount, date, description, category, and token. A console.log statement is used to log the transaction details. Finally, a fetch call is made to the /api/transactions endpoint with the method set to POST and the query parameters converted to a string.

```
1  const queryParams = new URLSearchParams({
2    type,
3    amount,
4    date,
5    description,
6    category,
7    token,
8  });
9  console.log("[AddTransactionForm.tsx] adding transaction", amount, date, description, category);
10
11  const response = await fetch(`/api/transactions?${queryParams.toString()}`, {
12    method: 'POST',
13  });
14
```

FrontEnd - Trecho do código src/components/AddTransactionForm.tsx - linhas 19 a 31



— — —

**[API]:** Quando esta API recebe essa requisição, ela chama a camada **actions** (que, mais uma vez, não foi mostrada no diagrama por ser uma dependência de nextjs e não de projeto). Em resumo, forçamos que a requisição seja processada do lado do servidor, além de desacoplar o código melhorando a injeção de dependências. Essa API, após processar o request (chamando a camada actions), retorna uma resposta para o *front ent* dependendo do sucesso da requisição. -> **[Actions]**

```
1 export default async function handler(req: NextApiRequest, res: NextApiResponse) {
2   if (req.method === 'POST') {
3
4     try {
5       const { type, amount, date, description, category, token } = req.query;
6
7
8       const formData = new FormData();
9       formData.append('type', type as string);
10      formData.append('amount', amount as string);
11      formData.append('date', date as string);
12      formData.append('description', description as string);
13      formData.append('category', category as string);
14      formData.append('token', token as string);
15
16      console.log("[transaction/index.ts] adding transaction", formData);
17
18      await addTransaction(formData);
19      res.status(200).json({ message: 'Transaction added successfully' });
20    } catch (error) {
21      console.error('Error adding transaction:', error);
22      res.status(500).json({ error: 'Internal server error' });
23    }
24  } else if (req.method === 'GET') {
25    try {
26      console.log("[transaction/index.ts] fetching transactions");
27      const { token } = req.query;
28      console.log("[transaction/index.ts] token", token);
29      const transactions = await getTransactions(token as string);
30      res.status(200).json(transactions);
31    } catch (error) {
32      console.error('Error fetching transactions:', error);
33      res.status(500).json({ error: 'Internal server error' });
34    }
35  } else {
36    res.status(405).json({ error: 'Method not allowed' });
37  }
38 }
```

API - Trecho do código src/pages/api/transactions/index.ts - Linhas 4 a 41

— — —

**[Actions]:** Retornando para a camada de **actions**, quando a requisição chega na API, ela chama a função **addTransaction()**, situada em **actions/**. Essa função é responsável por consultar a relação de **History** do usuário, encontrá-la e posteriormente criar uma nova **Transaction**. Lembrando que **History** e **Transaction** são classes modeladas nos diagramas situadas na camada **models/**. Assim, ao chamar **Transaction.create()**, a camada **models** é invocada. -> **[Models]**

```
1 export async function addTransaction(formData: FormData): Promise<void> {
2
3   console.log("[data.ts] adding transaction", formData);
4
5   const token = formData.get("token") as string;
6
7   console.log("[data.ts] found user token")
8
9   const history = await History.getByUserId(token);
10  if (!history) return;
11
12  console.log("[data.ts] found history")
13  let transactionType = ""
14  switch (formData.get("type")) {
15    case "income":
16      transactionType = TransactionType.INCOME
17      break;
18    case "expense":
19      transactionType = TransactionType.EXPENSE
20      break;
21    default:
22      transactionType = TransactionType.EXPENSE
23  }
24
25  await Transaction.create(history.id, {
26    type: transactionType as TransactionType,
27    amount: Number(formData.get("amount")),
28    date: new Date(formData.get("date") as string),
29    description: formData.get("description") as string,
30    category: formData.get("category") as string,
31  });
32
33  console.log("[data.ts] created transaction")
34
35  // history totals
36  await history.updateTotals();
37
38  console.log("[data.ts] updated totals")
39 }
```

— — —

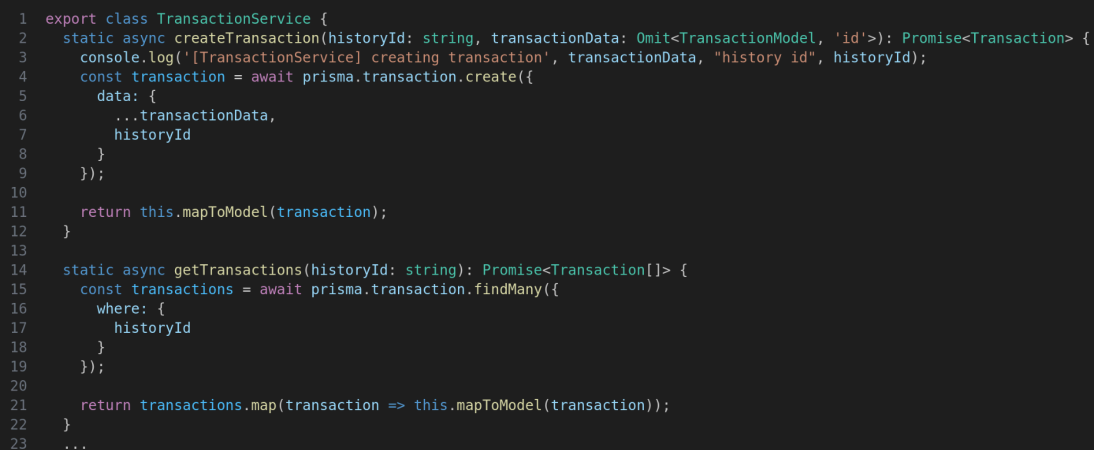
**[Models]:** Esta camada é responsável por realizar a injeção de dependências do projeto. No caso, a única dependência externa fortemente ligada é a camada do banco de dados. Para um melhor desacoplamento das camadas, o Prisma (ORM) é implementado na camada **services**. Portanto **models.Transaction.create()** invoca uma chamada assíncrona de **TransactionService.createTransaction()** na camada de serviço. -> **[Services]**

```
1 export class Transaction {
2   public id: string;
3   public type: TransactionType;
4   public amount: number;
5   public date: Date;
6   public description: string;
7   public category: string;
8
9   constructor({
10    id, type, amount, date, description, category
11  }: TransactionModel
12  ) {
13    this.id = id;
14    this.type = type;
15    this.amount = amount;
16    this.date = date;
17    this.description = description;
18    this.category = category;
19  }
20
21  static async create(historyId: string, data: Omit<TransactionModel, "id" | "historyId">): Promise<Transaction> {
22    const transaction = await TransactionService.createTransaction(historyId, data);
23
24    return new Transaction(transaction);
25  }
26
27  getDetails() {
28    return {
29      id: this.id,
30      type: this.type,
31      amount: this.amount,
32      date: this.date,
33      description: this.description,
34      category: this.category
35    }
36  }
37
38  async update(updatedFields: Partial<TransactionModel>): Promise<void> {
39    this.type = updatedFields.type ?? this.type;
40    this.amount = updatedFields.amount ?? this.amount;
41    this.date = updatedFields.date ?? this.date;
42    this.description = updatedFields.description ?? this.description;
43    this.category = updatedFields.category ?? this.category;
44
45    await TransactionService.updateTransaction(this.id, updatedFields);
46  }
47 }
```

Models - Trecho do código src/models/transactions.ts - linhas 20 a 66

— — —

**[Services]:** Por fim, esta camada implementa a dependência do **Prisma**, ou seja, ela realiza as chamadas e as alterações no banco de dados “diretamente”. Poderíamos, para exemplificar a injeção de dependências completa, citar a abstração fornecida pelo ORM Prisma, o qual é um ótimo exemplo para a disciplina. No entanto, como foge do escopo do projeto em si e chega a ser semelhante à nossa implementação, vamos pular. -> **[Prisma] (BD)**



```
1 export class TransactionService {
2   static async createTransaction(historyId: string, transactionData: Omit<TransactionModel, 'id'>): Promise<Transaction> {
3     console.log('[TransactionService] creating transaction', transactionData, "history id", historyId);
4     const transaction = await prisma.transaction.create({
5       data: {
6         ...transactionData,
7         historyId
8       }
9     });
10
11     return this.mapToModel(transaction);
12   }
13
14   static async getTransactions(historyId: string): Promise<Transaction[]> {
15     const transactions = await prisma.transaction.findMany({
16       where: {
17         historyId
18       }
19     });
20
21     return transactions.map(transaction => this.mapToModel(transaction));
22   }
23   ...
24 }
```

Trecho do código src/services/transaction.service.ts - Linhas 4 a 25

Concluindo, todas as funções do front end seguem esse fluxo de interação, mostrado nos diagramas das outras seções.

## 8. Conclusão e Discussões

### 8.1. Desafios de planejamento de correlação entre os componentes

A maior dificuldade encontrada pela equipe no planejamento do sistema foi encontrar uma relação adequada entre as classes e os componentes projetados. Embora simples, o sistema requer uma relação coesa e bem definida, garantindo que todos os elementos possam complementar o sistema como um todo, sem falhar em seus comportamentos individuais. Assim, seguimos os padrões passados em sala de aula para a modelagem das correspondências dos elementos, quebrando a aplicação em pequenos pedaços relacionando cada parte com outra.

### 8.2. Arquitetura limpa do sistema

Além disso, arquitetura do sistema também foi um desafio, pois a plataforma deve ser desenvolvida de forma modular e seguindo os princípios de evitar o acoplamento de suas partes, ou o famoso código “macarrão”, o qual é extremamente complexo de realizar qualquer alteração e atualização, assim como adição de novas funcionalidades.

Com isso em mente, o sistema foi projetado visando a separação das partes que compõem um grupo de funcionalidade, como componentes de **usuário**, **transação** e **autenticação**. Dessa forma, cada parte “se preocupa” em executar sua função e só. Ao modelar a estrutura dessa maneira, dividimos a aplicação em módulos que são facilmente alterados e melhorados conforme o processo de desenvolvimento acontece, assim como sugerido por *Robert C. Martin*, autor do livro “*Clean Architecture: A Craftsman’s Guide to Software Structure and Design*”, o qual defende que o processo de desenvolvimento e planejamento de uma arquitetura e modelagem não é um processo linear com início, meio e fim, e sim um processo em constante transição e ocorrência.

### 8.3. Segurança e proteção dos dados

Outro ponto discutido pelo grupo foi a proteção dos dados sensíveis do usuário. Como se trata de informações sensíveis, como quantias de dinheiro, investimentos, gastos, etc, a plataforma tem que garantir a proteção e recuperação de todos os usuários de forma robusta. Com isso em mente, a implementação de uma *pipeline* de segurança mais fundamentada e moderna ainda segue em discussão, porém começará com alguns testes de encriptação de dados de forma mais objetiva para um primeiro começo

### 8.4. Concepção e desenvolvimento das interfaces

Por último, um desafio muito grande também será o desenvolvimento das interfaces de usuários para a visualização de todos os dados. Como se trata de dados voláteis e dinâmicos, o ideal seria construir uma visualização intuitiva, separando as informações mais importantes primeiro e mostrando os dados analíticos de forma gráfica e matemática. Transformar todo esse planejamento em uma interface responsiva e leve não será simples, mas será um aprendizado muito valioso para o grupo. Soluções mais comumente utilizadas para mostrar gráficos dinâmicos pelo front end estão sendo consideradas, como *Chart.js*.