11/10/2025

ATIVIDADE - ANÁLISE DE PRINCÍPIOS DE DESIGN APLICADOS

Nome/RA:	*Bernardo Seijas Cavalcante/24026290*
Nome/RA:	*Eduardo Chen Zou/24025817*
Nome/RA:	*Fabiano Henrique Chou/24025991*
Nome/RA:	*Nicolas Morales/24025897*

Visao Geral do Data Analytics:

Ao longo do processo de desenvolvimento, buscou-se aplicar de forma consistente os princípios de design, garantindo que o sistema apresentasse robustez, escalabilidade e facilidade de manutenção.

Esta nova versão reformula o material da entrega original, adaptando-o à realidade do projeto PicMoney, que conta com arquitetura em camadas e backend implementado em JavaScript.

Utilizamos JavaScript e JSX para construir a aplicação e realizar a leitura e o processamento dos arquivos CSV, integrando as informações de forma dinâmica e eficiente.

Além de conservar a fundamentação teórica e os exemplos práticos, foram inseridos diagramas UML em alta resolução, oferecendo uma visão mais clara e completa da arquitetura proposta.

SOLID

Esta é uma análise de como os princípios SOLID e o padrão Observer (Observador) foram implementados na arquitetura do ExecutiveOverviewDashboard.

1. (S) - Princípio da Responsabilidade Única (SRP)

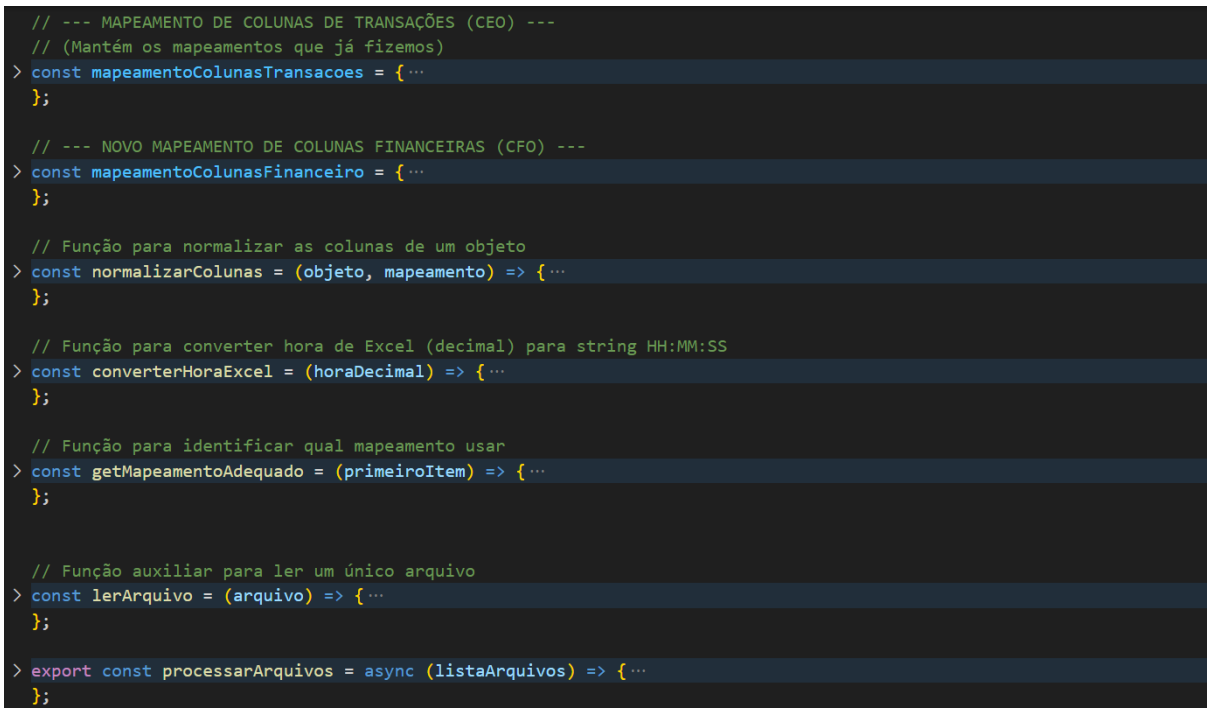
"Um componente ou módulo deve ter apenas uma, e somente uma, razão para mudar."

Este é o princípio que mais aplicamos. Nós quebramos o processo complexo do dashboard em múltiplos arquivos, cada um com uma única responsabilidade:

Src/api/processamentoDados.js

Responsabilidade Única: Ler arquivos brutos (.csv, .xlsx), "limpar" os dados (ex: converter hora, normalizar nomes de colunas) e entregar um array de objetos JavaScript padronizado.

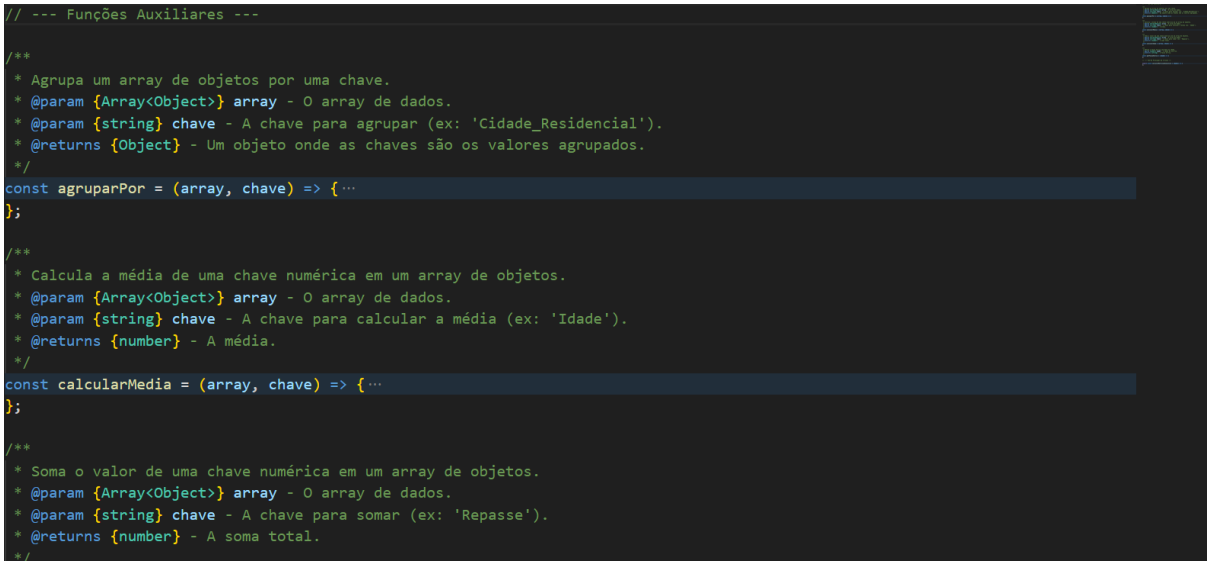
Razão para Mudar: Somente se o formato dos arquivos de entrada mudar (ex: uma nova coluna ou um novo formato de hora). Ele não sabe o que é um KPI.



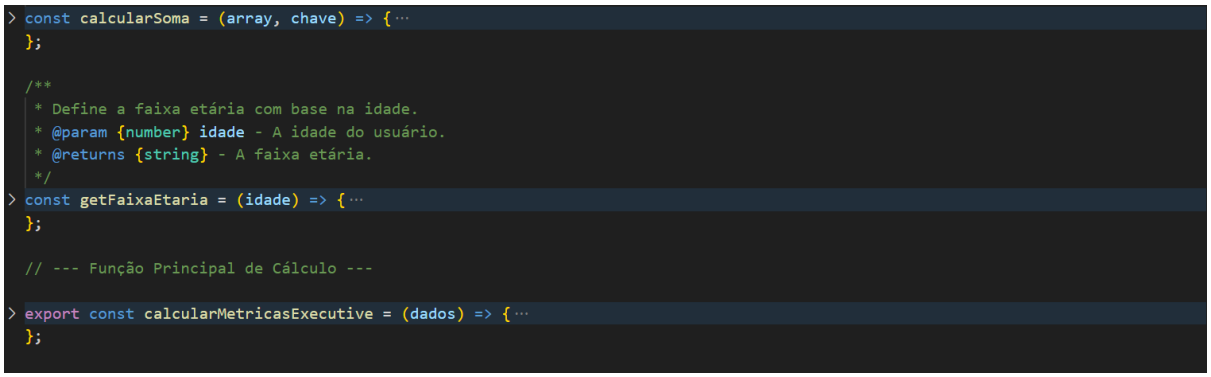
src/utils/calculosExecutive.js

Responsabilidade Única: Receber dados limpos e executar lógica de negócio (cálculos de KPIs e agregação de dados para os gráficos).

Razão para Mudar: Somente se uma regra de negócio mudar (ex: a fórmula do "Ticket Médio" for alterada). Ele não sabe ler arquivos nem desenhar gráficos.



```
// --- Funções Auxiliares ---

/**
 * Agrupa um array de objetos por uma chave.
 * @param {Array<Object>} array - O array de dados.
 * @param {string} chave - A chave para agrupar (ex: 'Cidade_Residencial').
 * @returns {Object} - Um objeto onde as chaves são os valores agrupados.
 */
const agruparPor = (array, chave) => {···
};

/**
 * Calcula a média de uma chave numérica em um array de objetos.
 * @param {Array<Object>} array - O array de dados.
 * @param {string} chave - A chave para calcular a média (ex: 'Idade').
 * @returns {number} - A média.
 */
const calcularMedia = (array, chave) => {···
};

/**
 * Soma o valor de uma chave numérica em um array de objetos.
 * @param {Array<Object>} array - O array de dados.
 * @param {string} chave - A chave para somar (ex: 'Repasse').
 * @returns {number} - A soma total.
 */
```



```
> const calcularSoma = (array, chave) => {···
  };

  /**
   * Define a faixa etária com base na idade.
   * @param {number} idade - A idade do usuário.
   * @returns {string} - A faixa etária.
   */
> const getFaixaEtaria = (idade) => {···
  };

  // --- Função Principal de Cálculo ---

> export const calcularMetricasExecutive = (dados) => {···
  };
```

src/pages/ExecutiveOverviewDashboard.jsx

Responsabilidade Única: Orquestrar a exibição dos dados. Ele consome os dados e os cálculos e os distribui para os componentes de UI.

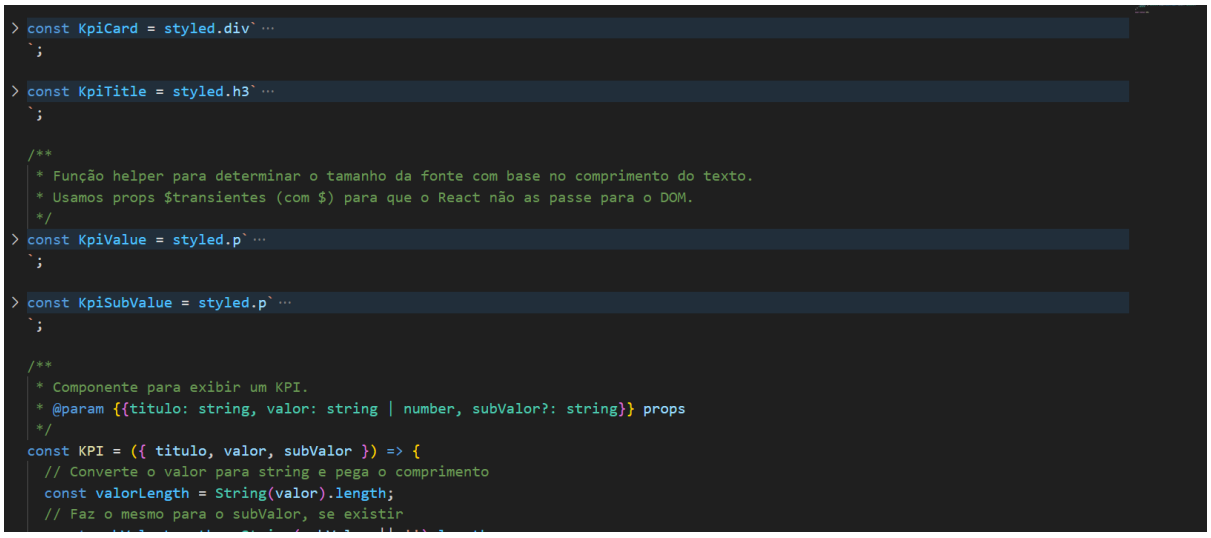
Razão para Mudar: Somente se o layout do dashboard mudar (ex: um gráfico mudar de lugar). Ele não sabe como os dados foram calculados.

src/components/dashboards/KPI.jsx

Responsabilidade Única: Exibir um título, um valor e um subvalor.

Razão para Mudar: Somente se a aparência visual de um cartão de KPI mudar.

Ao seguir o SRP, garantimos que uma mudança nos cálculos não quebre a leitura de arquivos, e uma mudança no layout não quebre os cálculos.



2. (O) - Princípio Aberto/Fechado (OCP)

"As entidades de software (componentes, módulos) devem estar abertas para extensão, mas fechadas para modificação."

Em React, alcançamos isso principalmente através da composição e do uso da prop children.

SecaoExpansivel.jsx e GraficoWrapper.jsx

Estes componentes estão fechados para modificação: a lógica de "expandir/recolher" ou de "criar um contêiner responsivo" está pronta e não precisa ser alterada.

Eles estão abertos para extensão: Usando a prop children, podemos colocar qualquer conteúdo dentro deles (uma grid de KPIs, um gráfico de barras, um gráfico de pizza) sem nunca ter que editar o código-fonte desses componentes.

```
> const SecaoContainer = styled.div` ···
  `;

> const SecaoHeader = styled.div` ···
  `;

> const SecaoTitulo = styled.h2` ···
  `;

> const BotaoToggle = styled.button` ···
  `;

> const SecaoConteudo = styled.div` ···
  `;

> const SecaoExpansivel = ({ titulo, children, comecarAberto = true }) => {···
  };

  export default SecaoExpansivel;
```

```
const ChartContainer = styled.div`
  background: ${({ theme }) => theme.cardBg};
  border: 1px solid ${({ theme }) => theme.borderColor};
  border-radius: 12px;
  padding: 1.5rem;
  box-shadow: ${({ theme }) => theme.shadow};
  height: 400px; /* Altura padrão para os gráficos */
`;

const ChartTitle = styled.h3`
  font-size: 1.1rem;
  font-weight: 600;
  margin-bottom: 1.5rem;
  text-align: center;
`;

const GraficoWrapper = ({ titulo, children }) => {
  return (
    <ChartContainer>
      <ChartTitle>{titulo}</ChartTitle>
      {/* ResponsiveContainer garante que o gráfico se ajuste ao container */}
      <ResponsiveContainer width="100%" height="85%">
        {children}
      </ResponsiveContainer>
    </ChartContainer>
  );
};
```

3. (D) - Princípio da Inversão de Dependência (DIP)

"Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações."

Este é um dos princípios mais importantes que usamos, e ele está diretamente ligado ao Padrão Observer.

O Problema (Sem DIP): O ExecutiveOverviewDashboard (módulo de alto nível) precisaria "saber" do SeletorArquivos (módulo de baixoível) e perguntar a ele: "Ei, você já carregou os dados?". Isso cria um acoplamento forte.

A Solução (Com DIP):

A Abstração: Nós criamos o DadosContext. O Contexto não sabe quem vai consumir os dados nem quem vai fornecê-los. Ele é apenas uma "caixa" central abstrata.

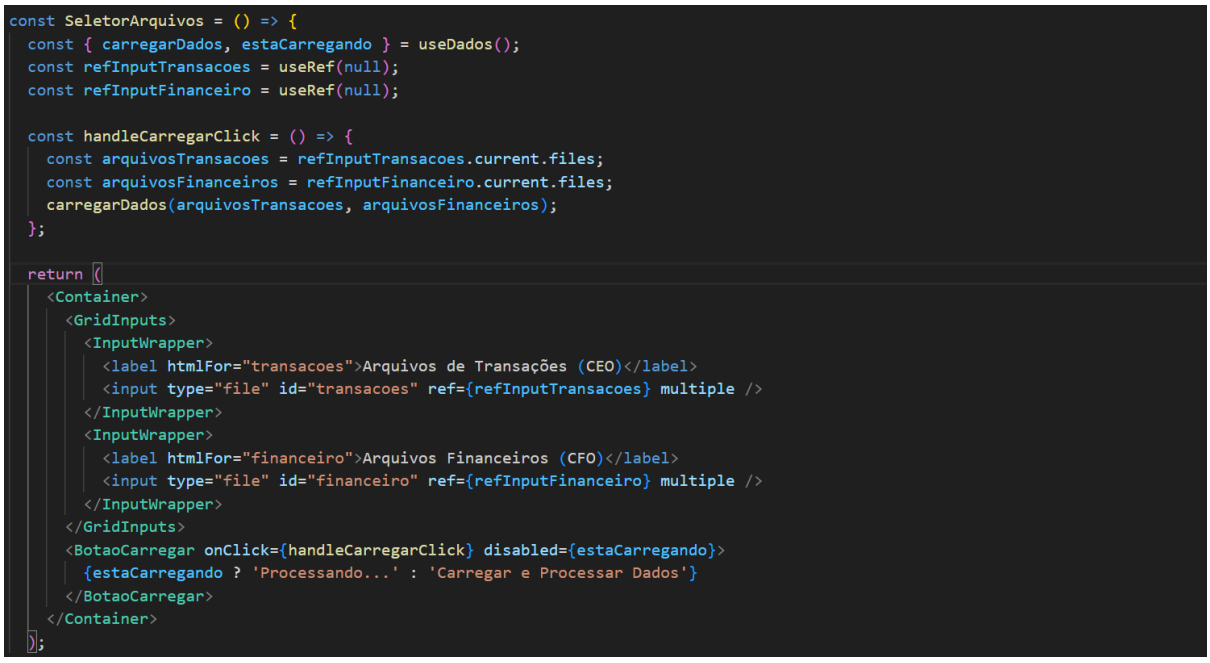
A Inversão:

O ExecutiveOverviewDashboard (alto nível) depende da abstração (useDados()).

O SeletorArquivos (baixo nível) também depende da abstração (ele chama carregarDados() do contexto).

O módulo de alto nível (dashboard) não depende mais do módulo de baixo nível (seletor). Ambos dependem do Contexto.

```
const ExecutiveOverviewDashboard = () => {
  const { dadosTransacoes, estaCarregando, erro } = useDados();

  // Estado para o filtro Top N do gráfico 15
  const [topNEstab, setTopNEstab] = useState(10);

  // 1. Hook de Filtros
  const { filtros, setFiltros, dadosFiltrados, opcoesFiltros } = useFiltrosExecutive(dadosTransacoes);

  // 2. Memoiza os cálculos
  const metricas = useMemo(() => {
    return calcularMetricasExecutive(dadosFiltrados);
  }, [dadosFiltrados]);

  // --- Renderização ---

  if (estaCarregando) {
    return <TelaCarregamento />;
  }

  if (dadosTransacoes.length === 0) {
    return (
      <DashboardContainer>
```



```
const SeletorArquivos = () => {
  const { carregarDados, estaCarregando } = useDados();
  const refInputTransacoes = useRef(null);
  const refInputFinanceiro = useRef(null);

  const handleCarregarClick = () => {
    const arquivosTransacoes = refInputTransacoes.current.files;
    const arquivosFinanceiros = refInputFinanceiro.current.files;
    carregarDados(arquivosTransacoes, arquivosFinanceiros);
  };

  return (
    <Container>
      <GridInputs>
        <InputWrapper>
          <label htmlFor="transacoes">Arquivos de Transações (CEO)</label>
          <input type="file" id="transacoes" ref={refInputTransacoes} multiple />
        </InputWrapper>
        <InputWrapper>
          <label htmlFor="financeiro">Arquivos Financeiros (CFO)</label>
          <input type="file" id="financeiro" ref={refInputFinanceiro} multiple />
        </InputWrapper>
      </GridInputs>
      <BotaoCarregar onClick={handleCarregarClick} disabled={estaCarregando}>
        {estaCarregando ? 'Processando...' : 'Carregar e Processar Dados'}
      </BotaoCarregar>
    </Container>
  );
```

4. Padrão Observer (Observador)

O seu PDF pedia especificamente por este padrão, e o DadosContext (junto com os hooks useState/useContext) o implementa perfeitamente:

O Subject (Sujeito): O DadosProvider é o sujeito que "possui" o estado (os dados).

Os Observers (Observadores): Qualquer componente que usa useDados() (como ExecutiveOverviewDashboard e FinancialPerformanceDashboard) torna-se um observador.

```jsx
import React, { createContext, useState, useContext } from 'react';
import { processarArquivos } from '../api/processamentoDados';

export const DadosContext = createContext();

export const DadosProvider = ({ children }) => {
  const [dadosTransacoes, setDadosTransacoes] = useState([]);
  const [dadosFinanceiros, setDadosFinanceiros] = useState([]);
  const [estaCarregando, setEstaCarregando] = useState(false);
  const [erro, setErro] = useState(null);

  // Função para carregar e processar os arquivos
  const carregarDados = async (arquivosTransacoes, arquivosFinanceiros) => {
    setEstaCarregando(true);
    setErro(null);
    try {
      const [transacoes, financeiros] = await Promise.all([
        processarArquivos(arquivosTransacoes),
        processarArquivos(arquivosFinanceiros),
      ]);
      setDadosTransacoes(transacoes);
      setDadosFinanceiros(financeiros);
    } catch (e) {
      setErro('Falha ao processar os arquivos. Verifique o formato e tente novamente.');
      console.error(e);
    } finally {
      setEstaCarregando(false);
    }
  };
```

A Notificação: Quando o SeletorArquivos chama carregarDados(), o Subject (Provider) atualiza seu estado. O React automaticamente "notifica" todos os Observers (componentes) de que o estado mudou, fazendo com que eles renderizem novamente com os novos dados.

Isso nos permite ter múltiplos dashboards observando os mesmos dados e reagindo em tempo real quando eles são carregados, sem que eles precisem "conversar" entre si

Modelo de Design de Arquitetura

A arquitetura do projeto PicMoney foi estruturada com base em uma abordagem modular em camadas, combinando organização lógica, reutilização de componentes e baixo acoplamento entre as partes do sistema. Essa estrutura facilita a evolução do código, a integração de novas funcionalidades e a manutenção a longo prazo.

1. Visão Geral da Arquitetura

A aplicação segue uma arquitetura em três camadas principais:

Camada de Apresentação (Frontend)
Desenvolvida em React-Vite, é responsável pela interface com o usuário.
Utiliza componentes reutilizáveis e responsivos, implementando os princípios SRP e OCP através de componentes modulares (como KPI.jsx, GraficoWrapper.jsx e SecaoExpansivel.jsx).

Camada de Lógica de Negócio (Business Logic)
Implementada em JavaScript, esta camada concentra o processamento e as regras de negócio.
Arquivos como calculosExecutive.js aplicam o DIP, garantindo que as funções de cálculo fiquem independentes da leitura de dados e da interface.

Camada de Dados (Data Layer)
Responsável por carregar e tratar os arquivos .csv e .xlsx (localizados em processamentoDados.js).
A comunicação entre as camadas ocorre via Context API, que atua como uma abstração centralizada para compartilhamento de dados e estados.

2. Fluxo de Comunicação

O fluxo segue o padrão unidirecional, alinhado com os conceitos do Flux/Observer Pattern:

O usuário carrega um arquivo via SeletorArquivos;

O arquivo é processado em processamentoDados.js e armazenado no DadosContext;

Os componentes observadores (ExecutiveOverviewDashboard, FinancialPerformanceDashboard) são automaticamente atualizados quando o estado do contexto muda;

Os dados tratados são exibidos visualmente por meio de componentes de UI dinâmicos.

Essa abordagem assegura baixo acoplamento e alta coesão, promovendo maior estabilidade e previsibilidade no comportamento da aplicação.

3. Benefícios da Arquitetura

Escalabilidade: novas páginas ou dashboards podem ser adicionados sem alterar os módulos existentes.

Manutenibilidade: alterações em cálculos, layout ou formato de dados impactam apenas seus módulos específicos.

Reusabilidade: componentes e hooks podem ser utilizados em diferentes partes da aplicação.

Desempenho: o React-Vite otimiza o carregamento e atualização de componentes, reduzindo o tempo de resposta.

Diagrama

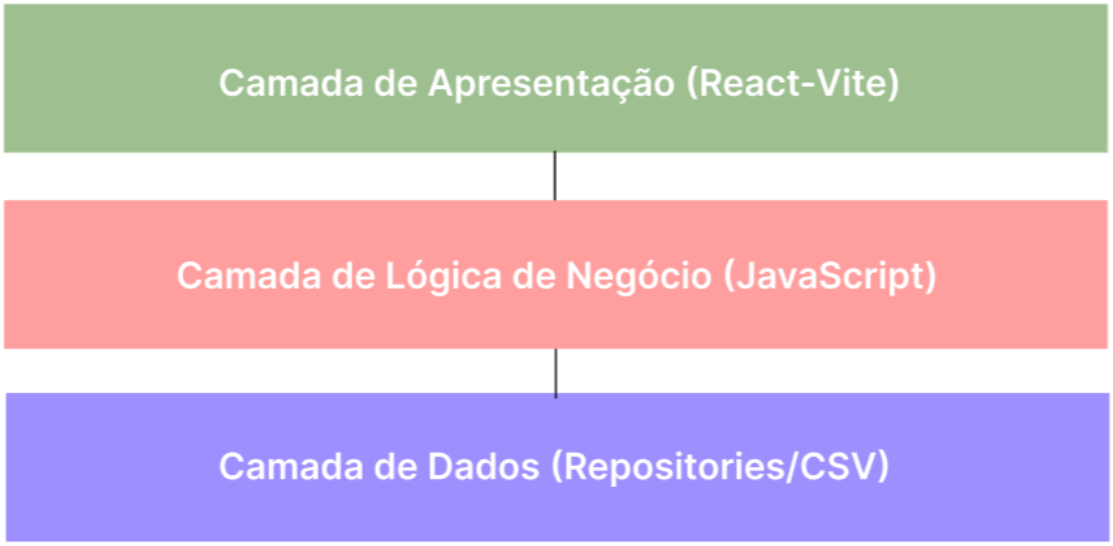


Figura 1 – Arquitetura em Camadas

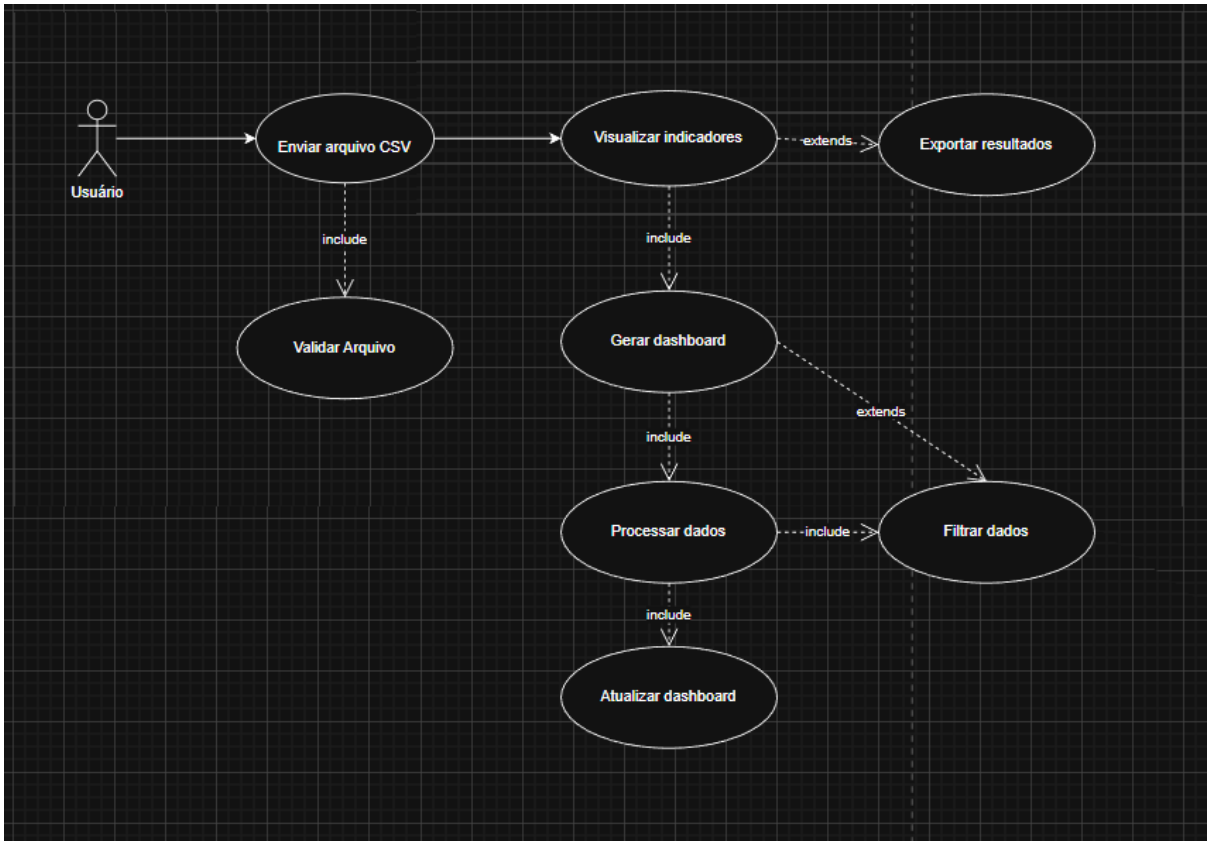


Figura 2 – Diagrama de caso de uso

Descrição Textual

O diagrama de caso de uso do projeto PicMoney representa as principais interações entre o usuário e o sistema, destacando as funcionalidades de carregamento, processamento e visualização de dados. Ele reflete uma arquitetura modular e escalável, de acordo com os princípios SOLID aplicados durante o desenvolvimento.