Entrega 2 PI - ES e AS

Aplicação de Design de Software no Projeto

No nosso projeto buscamos aplicar as melhores práticas de design de software, com o foco em ter um código limpo e de fácil manutenção, entre os conceitos aplicados estão: modularidade, legibilidade, encapsulamento, coesão, reuso, simplicidade e o projeto também buscou seguir os princípios SOLID de programação.

Alguns exemplos práticos destas aplicações:

Encapsulamento

Separamos as interfaces específicas do CEO e CFO, e há também a separação dos arquivos por página, ou seja, existe separação de responsabilidades e nenhum componente tem acesso a métodos ou informações que não precisa, os dados estão encapsulados.

- Importações organizadas entre módulos
- Separação clara entre dados, visualizações e interface

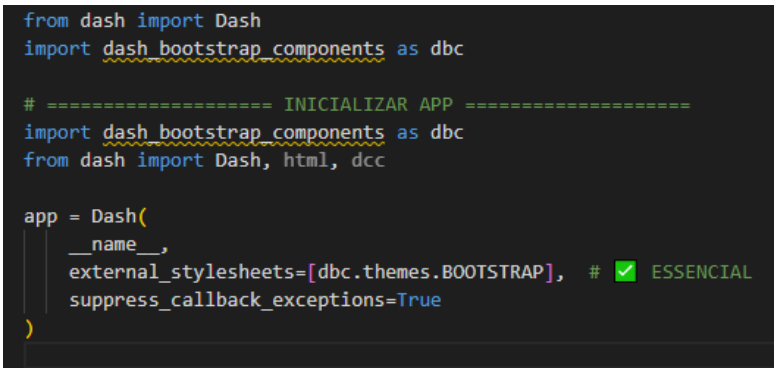
Exemplo prático:

```
def norm_cel(s):
    if pd.isna(s): return s
    return ''.join(ch for ch
in str(s) if ch.isdigit())
```

Oculta a complexidade: A lógica de limpeza do número de celular (verificar se é nulo, converter para string, filtrar apenas dígitos) fica escondida dentro da função.

Coesão e Legibilidade

Durante o projeto criamos um código coeso, simples e focado onde cada classe tinha uma única responsabilidade e bem definida. Também tivemos como foco criar um código fácil de se lido. O trecho a seguir é limpo, autoexplicativo e possui uma formatação consistente com comentários que auxiliam a legibilidade:



- Responsabilidade única: configurar a aplicação Dash
- Todas as linhas contribuem para o mesmo objetivo

Modularidade

A modularidade é um conceito que se baseia em dividir o sistema em partes menores, independentes e bem definidas, chamadas módulos. No nosso projeto a modularidade é crucial por várias razões:

1. Facilidade de Manutenção e Evolução

Módulos bem definidos isolam o impacto das mudanças.

- Manutenção Localizada: Se houver uma alteração na fonte de dados (por exemplo, mudar de CSV para um banco de dados), você só precisa modificar os módulos na Camada de Dados (CSVLoader → SQLLoader, por exemplo). As camadas superiores (Apresentação e Negócio) não precisam ser alteradas, desde que a interface do módulo de Dados permaneça a mesma.
- Evolução: Se você precisar adicionar uma nova visualização (ex: SalesPage), você pode criar um novo módulo de Apresentação, um novo serviço de Negócio (SalesChartService) e um novo loader de Dados (se necessário), sem reescrever o que já funciona para CEO e CFO.

Aqui está uma parte da estrutura de diretórios exemplificando a modularidade:

src/

|— utils/ # Módulo de utilitários

|— pages/ # Módulo de interface

|— app.py # Módulo de configuração

|— main.py # Módulo de controle

2. Testabilidade Aprimorada

A modularidade facilita os Testes Unitários.

- Testes Independentes: Você pode testar a lógica de negócio do CEOChartService sem precisar rodar a aplicação Dash inteira, e pode testar a lógica de processamento do DataProcessor sem depender da conexão com arquivos. Basta simular (mockar) as dependências de outras camadas.

3. Reutilização de Componentes

Componentes modulares com alta coesão (como discutido anteriormente) são mais fáceis de reutilizar.

- Exemplo: O ChartFactory é um módulo coeso que pode ser reutilizado por qualquer *ChartService que precise criar um gráfico, sem que o serviço precise saber *como* o gráfico é desenhado, apenas que ele pode *solicitá-lo*.

4. Gerenciamento de Complexidade

Um sistema grande como um dashboard pode se tornar rapidamente incontrolável se não for dividido. A modularidade gerencia essa complexidade:

- Visão Estruturada: As camadas definem uma hierarquia clara de dependência: a Camada de Apresentação depende da Camada de Negócio, que depende da Camada de Dados. Isso fornece um mapa mental claro de como o sistema funciona.

Princípios de Programação SOLID

Na nossa aplicação nós nos baseamos nos Princípios de Programação SOLID que são tão importantes para o Design de Software. O princípio que mais se destaca no nosso projeto é o **SRP (Single Responsibility Principle).** A modularidade reforça o princípio de que cada módulo ou camada deve ter uma responsabilidade primária clara.

Exemplo: A Camada de Dados (DataRepository, DataProcessor) é um módulo focado em acesso e limpeza de dados. A Camada de Apresentação (CEOPage, CFOPage) é um módulo focado em layout e visualização. Essa separação impede que a lógica de negócios se misture com a lógica de acesso a arquivos, por exemplo.

O princípio **OCP (Open Close Principle)** também aparece no nosso projeto pois novas páginas podem ser inseridas sem alterar as páginas existentes.

O princípio **ISP (Interface Segregation Principle)** pode ser encontrado nas seguintes funções que possuem responsabilidades específicas ao invés de herdarem de uma interface gigante

com todos os métodos, assim evitando dependências desnecessárias:

```python
def criar_grafico_receita_segmento(df):
    df_seg = (df
              .dropna(subset=["categoria_estabelecimento","valor_cupom"])
              .groupby("categoria_estabelecimento", as_index=False)["valor_cupom"].sum()
              .sort_values("valor_cupom", ascending=True))

    fig = px.bar(df_seg, x="valor_cupom", y="categoria_estabelecimento", orientation="h",
                 title="Receita total por segmento",
                 labels={"valor_cupom":"Receita (R$)","categoria_estabelecimento":"Segmento"})
    fig.update_layout(margin=dict(l=150), template='plotly_white')
    return fig

def criar_grafico_scatter(df_massa):
    df_scatter = df_massa.dropna(subset=["valor_cupom","valor_compra"])
    fig = px.scatter(df_scatter, x="valor_cupom", y="valor_compra", trendline="ols",
                     title="Relacionamento: valor do cupom x valor final da compra",
                     labels={"valor_cupom":"Valor do Cupom (R$)","valor_compra":"Valor da Compra (R$)"})
    fig.update_layout(template='plotly_white')
    return fig

def criar_grafico_ticket_medio(df_massa):
    df_mean = (df_massa.dropna(subset=["nome_loja","valor_compra"])
               .groupby("nome_loja", as_index=False)["valor_compra"].mean()
               .sort_values("valor_compra", ascending=False))

    fig = px.bar(df_mean, x="nome_loja", y="valor_compra",
                 title="Valor médio de venda por loja")
    fig.update_layout(xaxis_tickangle=-45, template='plotly_white')
    return fig
```

Diagramas UML do projeto

Utilizamos diagramas de UML para criar representações mais visuais do nosso projeto, com estes diagramas nós conseguimos explicar nosso projeto para os Stakeholders sem precisar nos aprofundar nas questões técnicas da programação. Os diagramas também foram úteis para direcionar nosso planejamento.

Diagrama de Casos de Uso

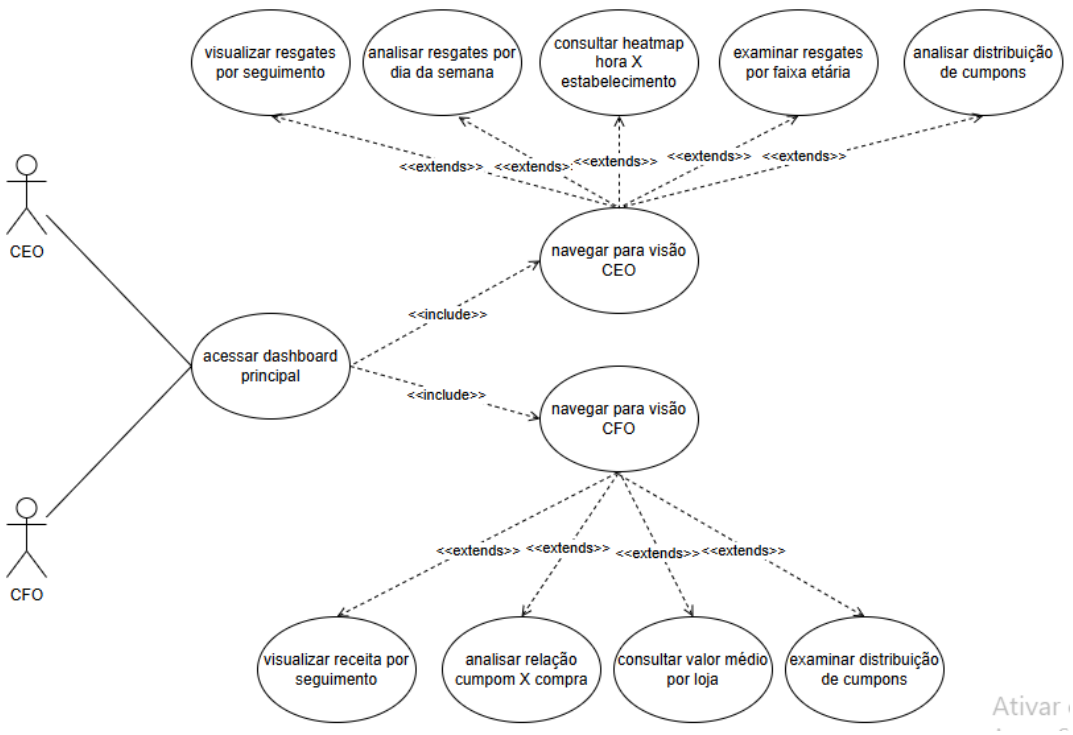
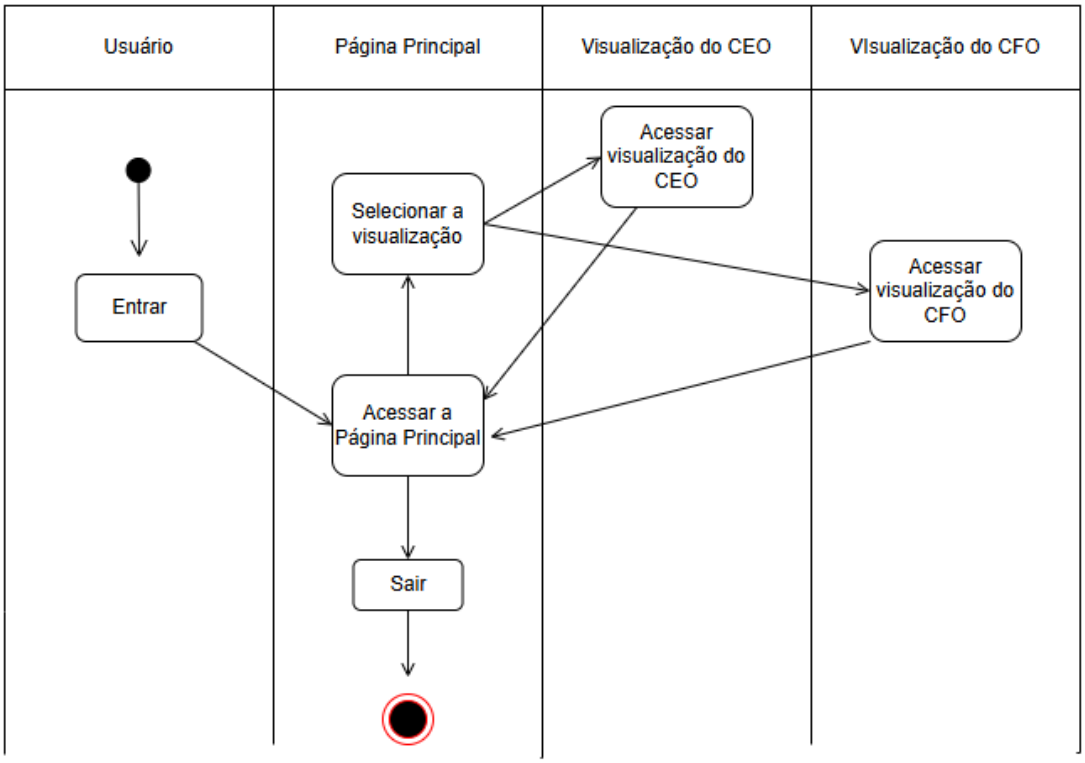


Diagrama de atividades



Conclusão

A utilização do Design de Software e dos diagramas foi essencial para a organização e bom andamento do projeto, nos ajudando a ter uma direção clara a seguir do ponto de vista de planejamento do projeto e das boas práticas de programação assim guiando nossa codificação. No final o resultado foi um projeto robusto e escalável.