

Engenharia de Software e Arquitetura de Sistemas

Entrega 2

Guilherme Barioni RA:24026140
Iury Xavier da Silva Mangueira RA:24026311
Lilian Mercedes Paye Conde RA:24026462
Marcus Miranda Duque RA:24026080
Murilo de Souza Vieira RA:24025726

Design de Software

Durante o desenvolvimento, a aplicação dos conceitos de design de software foi essencial para garantir que o nosso sistema fosse robusto, escalável e de fácil manutenção.

Princípio Solid

Single Responsibility Principle (SRP)

Todos os módulos no nosso projeto possuem somente uma responsabilidade e só podem ser modificados por apenas um motivo, por exemplo, o módulo PadronizacaoCSV.py é responsável somente pela padronização das planilhas que são enviadas ao sistema, enquanto o módulo Metricas.py é responsável somente pela extração dos dados para a análise financeira.

Ao separar essas funções em dois módulos distintos, garantimos mudanças afetaram somente aquele módulo específico, trazendo manutibilidade, testabilidade e flexibilidade.

Open Closed Principle (OSP)

Na nossa dashboard, cada usuário terá acesso a uma visualização diferente em relação aos dados e gráficos dependendo das suas preferências, para isso temos o módulo Detalhes.py, que traz uma padronização no layout que será utilizado para a demonstração dos dados e para exibir as informações distintas foi definido um método para cada categoria que expande em cima das informações do layout no Detalhes.py, mas sem modificá-lo diretamente.

Esse princípio garantiu a extensibilidade do código, permitindo a expansão sem modificar o código existente, a estabilidade, reduzindo a possibilidade de introduzir novos bugs ao realizar mudanças e a flexibilidade que torna mais fácil a adaptação e alterações de requisitos.

Liskov Substitution Principle (LSP)

As funções no arquivo Detalhes.py são independentes e fixas, todas as funcionalidades extras adicionadas ao código são feitas através de módulos e funções adicionais que acoplam com a função principal sem interferir

diretamente com as funções do arquivo. Estas medidas além de garantir esse polimorfismo, também aumentam a qualidade do software e a previsibilidade impedindo que novos erros sejam introduzidos.

Interface Segregation Principle (ISP)

O módulo MapaCalor.py permite a visualização de cada cupom capturado pelos usuários da PicMoney em um mapa de calor interativo que pode ser filtrado a partir de inúmeras categorias. Para isso, foi necessário a junção de inúmeras tabelas, utilizando a interface formatacao.py que contém somente os métodos específicos de formatação, com cada módulo de formatação implementando esses métodos.

A separação das funções utilizando o princípio ISP garantiu flexibilidade e desacoplamento, visto que cada módulo é independente e sendo possível a adaptação de cada função para situações específicas, evitando também dependência desnecessária.

Dependency Inversion Principle (DIP)

O módulo Relatorios.py é responsável pela geração de relatórios detalhados com base nos dados extraídos e processados ao longo do sistema. Em vez de depender diretamente de módulos específicos para gerar esses relatórios, ele abstrai suas dependências por meio de interfaces e injeção de dependência.

Por exemplo, o módulo Relatorios.py não depende diretamente da classe PadronizacaoCSV.py ou Metricas.py, mas sim de interfaces genéricas, como IFormatacao e IDados, que são implementadas por esses módulos. Isso significa que, se quisermos alterar a forma de gerar relatórios, como adicionar novos tipos de análise ou mudar a forma de exportação de dados (para PDF, Excel, etc.), podemos fazer isso sem modificar o código do Relatorios.py, mas apenas implementando novas classes que seguem essas interfaces.

Esse princípio assegura a flexibilidade e a manutenibilidade do sistema, pois mudanças em módulos de baixo nível não afetam diretamente os módulos de alto nível. Também melhora a testabilidade do código, pois podemos facilmente mockar ou substituir as dependências em testes, sem precisar alterar o código existente.

Modelo de Design

Os modelos de design foram fundamentais para guiar o desenvolvimento do software, ajudando a definir a estrutura e comportamento do Software.

Para definir e representar o nosso projeto escolhemos a arquitetura em camadas para ilustrar a lógica de negócio, a arquitetura em micro serviços para demonstrar como o nosso programa se relaciona com as APIs e o diagrama de casos que ajudará a guiar quais são os passos e em qual sequência o sistema realiza uma ação a partir da interação do usuário.

Arquitetura em camadas

A arquitetura em camadas contribui na visualização de das diferentes lógicas da operação, separando em diferentes camadas a lógica de apresentação, onde o usuário irá interagir com o sistema, a lógica de negócios, que rege sobre as funcionalidades e necessidades do sistema e a lógica de permanência de dados, que dita como os dados são tratados e armazenados:



No diagrama UML acima, o sistema foi dividido em três camadas:

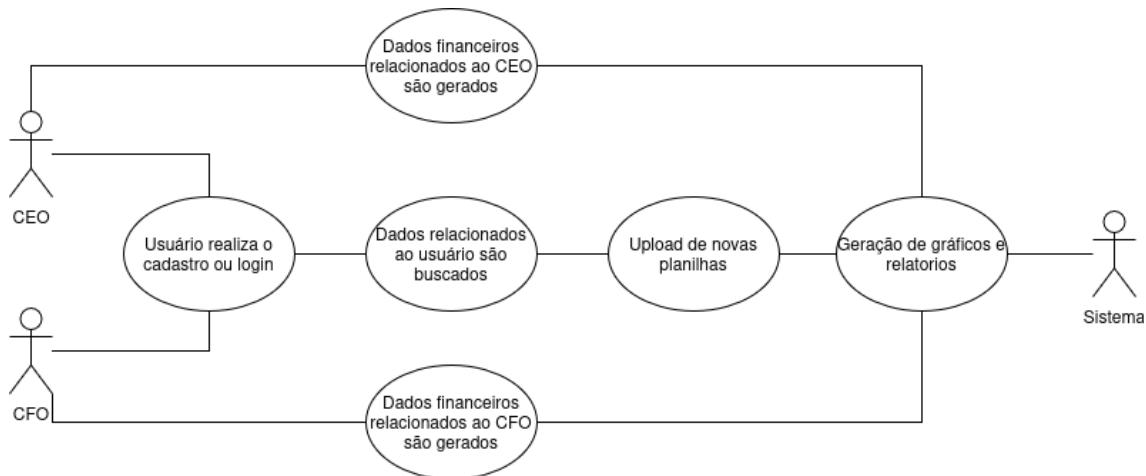
- **Camada de apresentação:** Com as informações sintetizadas em uma dashboard interativa, o usuário será capaz de acessar o sistema a partir de qualquer navegador acessando a url em computadores ou celulares.
- **Camada da lógica de negócio:** Com o cadastro das credenciais, os usuários podem fazer o envio das planilhas financeiras relacionadas ao período especificado, recebendo as métricas importantes extraídas dos

dados, com o envio de dados relevantes para o tratamento e armazenamento no backend.

- **Camada de persistência de dados:** Ao receber os dados, o backend faz o tratamento dos dados, validações, criptografia e envia para serem armazenados no banco de dados MySQL.

Diagrama de caso de uso

O diagrama de caso de uso ajuda o negócio e os desenvolvedores à entender como o usuário deve interagir com o sistema, permitindo um planejamento detalhado sobre as suas diversas funcionalidades.



Conclusão

A aplicação dos princípios de design de software e boas práticas de arquitetura foi essencial para garantir que o sistema desenvolvido fosse robusto, escalável e de fácil manutenção. A utilização dos princípios SOLID proporcionou um

código mais organizado, modular e flexível, permitindo a expansão de novas funcionalidades sem comprometer a estabilidade do sistema.

O Single Responsibility Principle (SRP) assegurou que cada módulo tivesse uma responsabilidade única, facilitando a manutenção e os testes. O Open/Closed Principle (OCP) promoveu a extensibilidade do código, reduzindo o risco de falhas ao realizar mudanças. O Liskov Substitution Principle (LSP) reforçou a previsibilidade e o polimorfismo entre as funções, enquanto o Interface Segregation Principle (ISP) garantiu o desacoplamento e a independência entre os módulos. Por fim, o Dependency Inversion Principle (DIP) trouxe maior flexibilidade e testabilidade, isolando os módulos de alto e baixo nível por meio de interfaces.

Além disso, a adoção de modelos de design como arquitetura em camadas, microserviços e MVC (Model-View-Controller) contribuiu para uma estrutura de software bem definida, separando as responsabilidades de apresentação, negócio e persistência de dados. O diagrama de caso de uso também foi essencial para mapear as interações do usuário e apoiar o planejamento das funcionalidades do sistema.

Em conjunto, essas práticas de design e arquitetura consolidaram um sistema modular, reutilizável e preparado para evoluções futuras, atendendo às necessidades do projeto com qualidade, eficiência e sustentabilidade técnica.