

# **Engenharia De Software e Arquitetura de Sistemas**

## **Entrega 2**

### **Integrantes:**

Carlos Roberto Santos Latorre

Felipe Lin

Felipe Wakasa Klabunde

Stephany Aliyah Guimarães Eurípedes de Paula

## **1. Introdução**

O presente documento apresenta como os conceitos de Design de Software, princípios SOLID, padrões de arquitetura e diagramas UML foram aplicados no desenvolvimento do Dashboard.

Este projeto foi estruturado para implementar diretamente os fundamentos de design e arquitetura na prática de desenvolvimento. O foco é evidenciar como essa abordagem resultou em um sistema mais organizado, adaptável e sustentável para desenvolvimentos futuros.

## **2. Design de Software**

O Design de Software é a etapa que transforma as necessidades de um projeto em um plano técnico. Na realidade, refere-se ao momento em que optamos pela estruturação do sistema de modo que o código se torne simples para manutenção, reutilização e atualizações futuras.

Nessa fase, definimos as “peças” do sistema como módulos, classes e funções e como elas vão se conectar. O foco é criar componentes que tenham responsabilidades claras (alta coesão) e que não dependam demais uns dos outros (baixo acoplamento).

Para que um projeto de design seja eficaz, é fundamental que ele seja descomplicado, coerente e promova a reutilização. Isso torna o sistema mais forte, com menos chances de falhas, e ajuda outros desenvolvedores a entenderem o código.

### **2.1 Princípio SOLID**

Ao longo do projeto, usamos os princípios SOLID como referência para manter o código mais organizado e fácil de evoluir. A ideia foi construir um sistema modular, com partes bem definidas, o que facilitou bastante tanto o desenvolvimento quanto a manutenção. Esses princípios orientaram principalmente a forma como estruturamos a autenticação, o tratamento de dados e a forma como os componentes se relacionam entre si.

O sigla SOLID reúne cinco princípios importantes da programação orientada a objetos:

#### **S – Single Responsibility Principle (SRP)**

Esse princípio diz que cada classe, módulo ou função deve ter apenas uma responsabilidade clara, ou seja, uma única razão para mudar.

No projeto, aplicamos isso ao dividir a lógica de autenticação em duas partes:

- Uma classe apenas para verificar e criar o usuário administrador;
- Outra focada exclusivamente em autenticar quem está tentando acessar o sistema.

Separar essas funções tornou o código mais limpo e simples de manter, além de facilitar testes e ajustes futuros.

### **O – Open/Closed Principle (OCP)**

Pelo OCP, o sistema deve permitir a criação de novas funcionalidades sem que seja preciso alterar o código já pronto.

Seguindo essa ideia, organizamos os componentes para que novas funções possam ser incluídas por extensão, sem mexer no núcleo do código. Isso ajudou a manter o sistema estável conforme ele crescia.

### **L – Liskov Substitution Principle (LSP)**

O LSP diz que as classes-filhas devem conseguir substituir suas classes-pai sem alterar o funcionamento esperado.

Na prática, isso orientou a criação de classes compatíveis entre si, permitindo que diferentes versões ou especializações pudessem ser usadas sem quebrar o comportamento do sistema. Assim, conseguimos flexibilidade sem perder estabilidade.

### **I – Interface Segregation Principle (ISP)**

Esse princípio defende que é melhor criar interfaces específicas do que uma interface única e muito geral.

Sob esse viés, dividimos as funcionalidades em módulos menores e mais focados. Cada parte do sistema implementa apenas o que realmente precisa, deixando o código mais claro e evitando sobrecarga desnecessária.

### **D – Dependency Inversion Principle (DIP)**

O DIP orienta que módulos de alto nível devem depender de abstrações, não de detalhes.

Por isso, organizamos a comunicação entre as camadas do sistema por meio de serviços e contratos mais genéricos em vez de depender diretamente de implementações específicas. Isso tornou o sistema mais flexível e menos sujeito a impactos quando alguma tecnologia precisar ser trocada ou atualizada.

Portanto, a aplicação dos princípios SOLID ajudou a manter o projeto mais estável e fácil de trabalhar.

Com menos acoplamento e mais clareza na função de cada parte, o sistema ficou mais preparado para crescer, receber melhorias e passar por manutenção sem grandes dificuldades.

### 3. Design de Software e Arquitetura Adotada

Nesse projeto foi desenvolvido com a arquitetura MVC (Model-View-Controller) para este projeto. Essa foi a decisão mais lógica para um sistema que precisa de uma interface interativa, manipula muitos dados e se comunica com um banco de dados.

O padrão MVC estabelece e separa as responsabilidades:

- O **Model** cuida dos dados e das regras de negócio.
- O **View** é a interface gráfica que o usuário vê e com a qual interage.
- O **Controller** age como o "intermediário" entre eles, processando as ações do usuário e devolvendo as respostas.

Na prática, fizemos o frontend (View) com React.js, que ficou responsável pelas telas e pelos gráficos interativos. O backend (Model e Controller) foi construído em Node.js, usando Express para gerenciar as rotas e o MySQL para guardar os dados. Para os gráficos dinâmicos, usamos bibliotecas como Plotly e Chart.js. Essa estrutura modular foi o que nos permitiu separar claramente as funções de cada parte do sistema. O resultado é um código muito mais limpo, organizado e fácil de manter.

Além disso, fizemos questão de usar outras boas práticas de arquitetura, como o Repository Pattern e a injeção de dependências. Isso torna o sistema mais flexível e pronto para crescer. Deixamos a aplicação preparada para futuras integrações com APIs e até para possíveis extensões com Machine Learning.

### 4. Diagramas UML Aplicados

A UML (Linguagem de Modelagem Unificada) é, basicamente, a forma padronizada que usamos para “desenhar” o software. Em vez de cada um representar o sistema de um jeito diferente, usamos a UML para que todos possam olhar para um diagrama e entender a mesma informação.

Ela nos ajuda a pensar na estrutura e no funcionamento do sistema antes de começar a programar, o que é ideal para planejar e alinhar expectativas entre a equipe e também com o cliente.

No nosso projeto, utilizamos principalmente três tipos de diagrama:

- **Diagrama de Casos de Uso:** para mostrar o que cada ator (**Usuário, CEO, CFO, CTO e Sistema**) pode fazer, como acessar o sistema, visualizar KPIs, aplicar filtros, gerar dados financeiros e receber alertas;
- **Diagrama de Classes:** apresenta a estrutura do sistema, como as classes (por exemplo, **Usuario, Dashboard, KPIService, FinanceiroServiceCFO, Transacao**) se organizam, seus atributos, métodos e suas relações;

- **Diagrama de Sequência:** mostra passo a passo como ocorre a troca de mensagens entre os componentes (como interface, controladores, serviços e banco de dados) ao longo do fluxo: cadastro/login → KPIs → dados financeiros (CFO) → gráficos em tela.

Com esses diagramas, conseguimos “ver” como os componentes se organizam e se comunicam. Isso é essencial para explicar ideias complexas, confirmar se uma funcionalidade faz sentido antes de implementá-la e garantir que todos estão seguindo o mesmo plano.

No fim, esses diagramas serviram como guias na fase de planejamento. Eles nos deram uma visão clara do todo, permitiram ajustar o desenho do sistema logo no início e ajudaram a manter uma documentação alinhada com o que foi construído.

#### 4.1 Diagrama de Casos de Uso

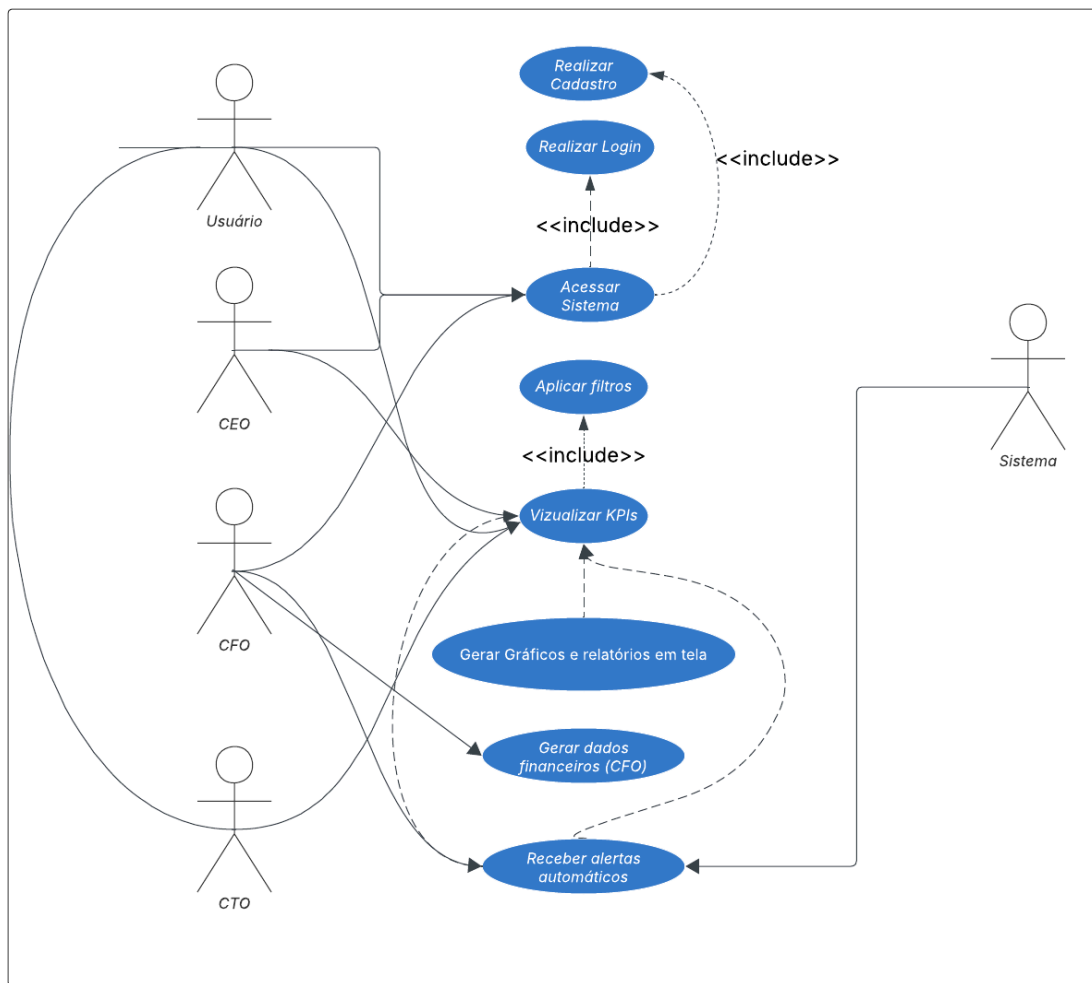
O diagrama de casos de uso mostra, de forma visual, o que cada tipo de usuário pode fazer no sistema.

No projeto, definimos quatro "atores": Usuário, CEO, CFO, CTO e o próprio Sistema (que dispara ações automáticas). As principais ações (casos de uso) que mapeamos são:

- Acessar o sistema (seja fazendo login ou um novo cadastro);
- Visualizar KPIs (o que exige o passo de aplicar filtros);
- Gerar gráficos e relatórios na tela, com base nos indicadores;
- Gerar dados financeiros (uma ação específica para o perfil CFO);
- Receber alertas automáticos (uma ação do próprio sistema).

O diagrama também esclarece a ordem das coisas. Fica claro que algumas ações dependem de outras: para "Visualizar KPIs", o usuário precisa primeiro "Acessar o sistema". Da mesma forma, para "Gerar gráficos", ele precisa antes ter visualizado os indicadores.

Ele também mostra que "Gerar dados financeiros" é uma função específica do CFO, enquanto os "Alertas automáticos" são um processo à parte, que não depende de uma ação do usuário. Esse diagrama foi essencial para entendermos o que cada ator pode fazer e como as funcionalidades se conectam.



## 4.2 Diagrama de Classes

O Diagrama de Classes funciona como a "planta baixa" do sistema. Ele define as classes principais, o que elas contêm (atributos), o que sabem fazer (métodos) e como se conectam.

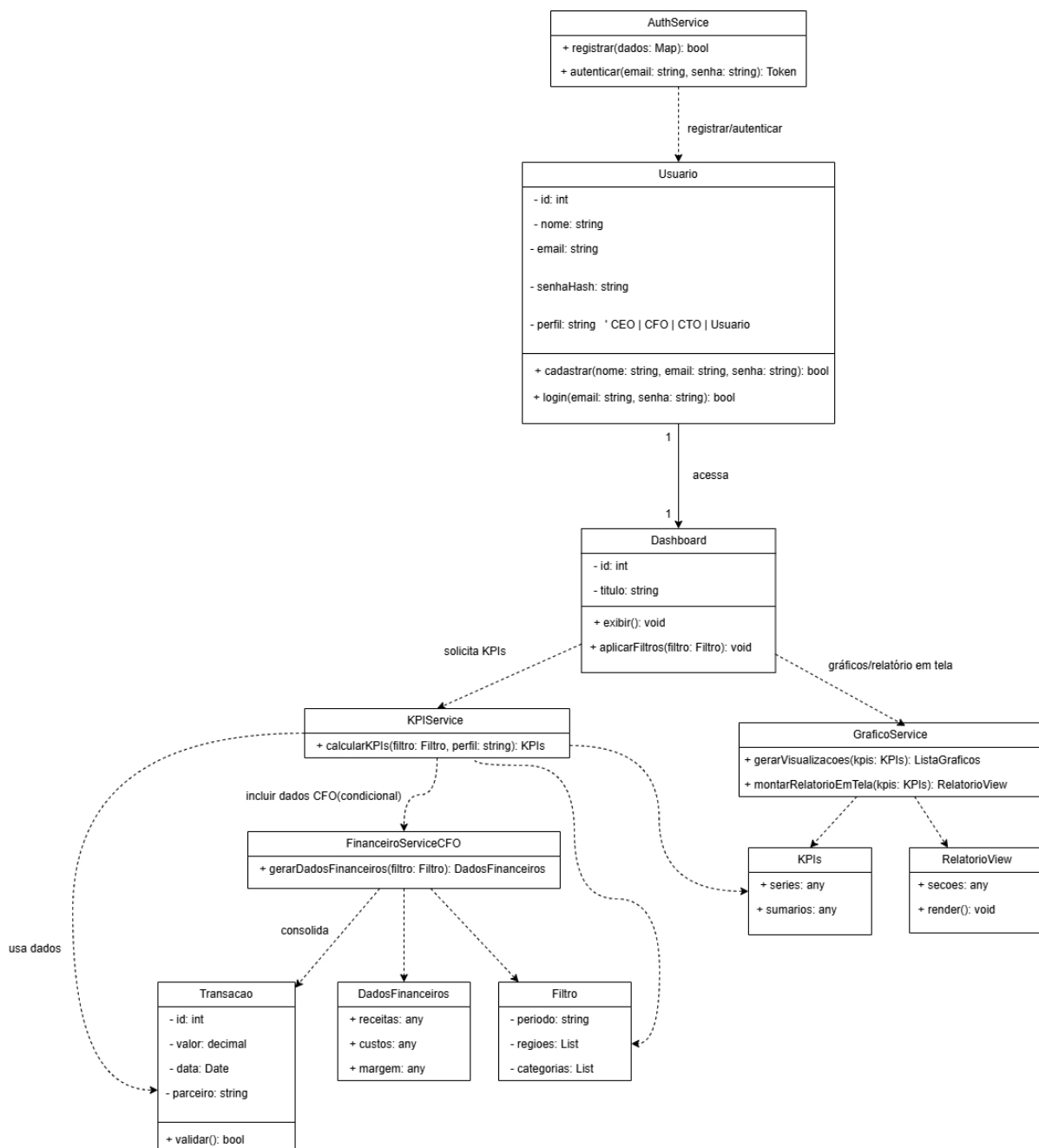
No diagrama do projeto, definimos as classes centrais: **Usuario**, **Dashboard**, **KPIService**, **FinanceiroServiceCFO**, **Transacao** e outras, cada uma com um papel bem específico.

- A classe **Usuario** cuida das informações de perfil (nome, e-mail, tipo) e dos processos de login e cadastro.
- A **Dashboard** é a interface principal onde o usuário vê as informações e aplica filtros.
- O **KPIService** é quem processa os indicadores, calculando os KPIs com base nos filtros e nos dados da **Transacao**.

- Quando o usuário é um CFO, o **FinanceiroServiceCFO** entra em ação para consolidar os dados financeiros (receitas, custos, margem).

Para organizar tudo isso, classes como **KPIs**, **DadosFinanceiros** e **Filtro** estruturam as informações. Por fim, a **GraficoService** pega os KPIs calculados e gera os gráficos e relatórios na tela.

Esse diagrama foi fundamental para visualizar o fluxo de informações e garantir que cada classe tivesse sua responsabilidade bem definida. Isso nos ajudou a seguir os princípios SOLID e o MVC, deixando o código mais organizado, flexível e fácil de manter.

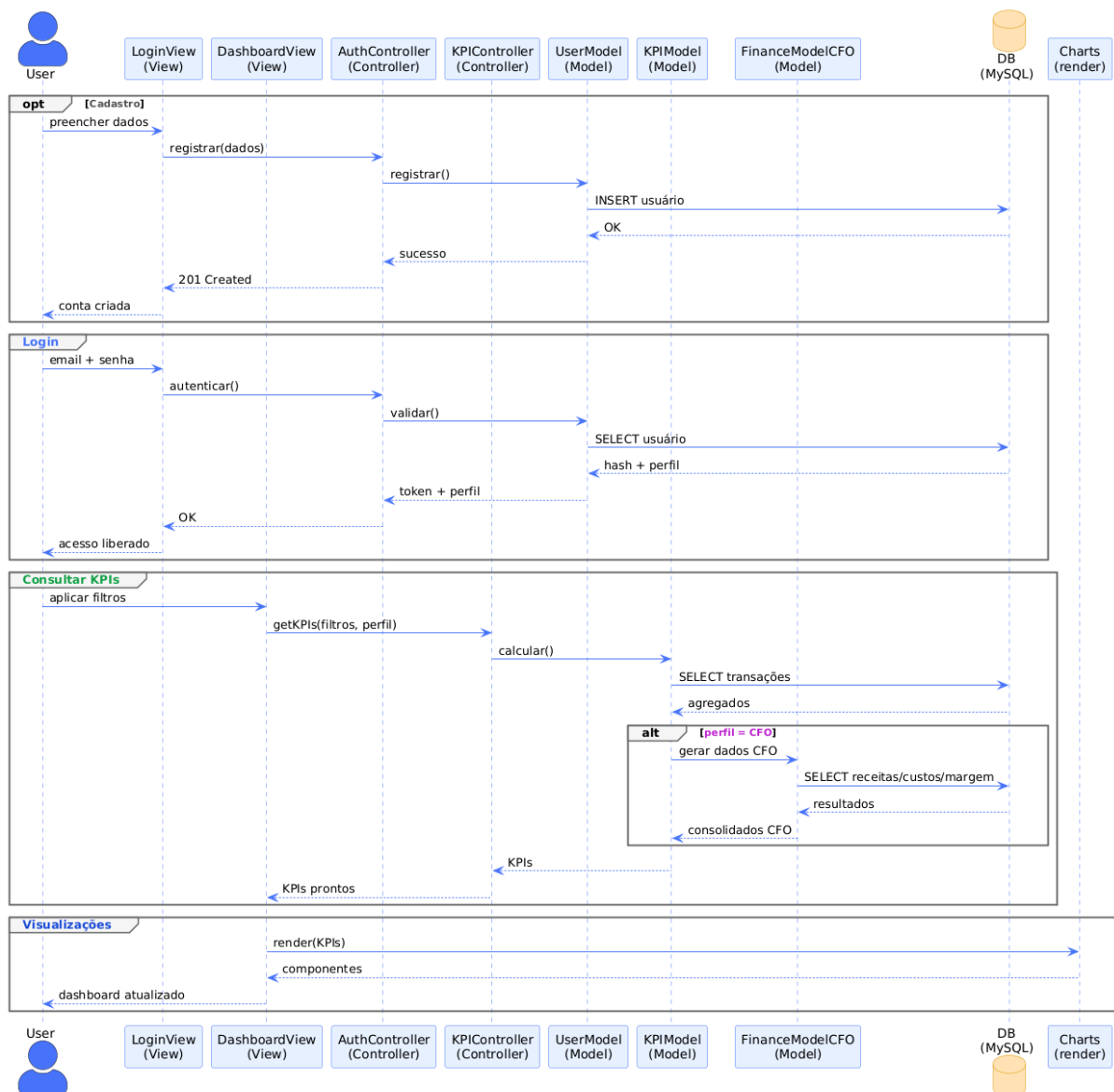


### **4.3 Diagrama de Sequência**

O Diagrama de Sequência mostra, passo a passo, como ocorre a comunicação entre os principais elementos do sistema durante sua execução. Ele descreve a troca de mensagens entre a interface, os controladores, os serviços e o banco de dados, permitindo visualizar claramente como as funcionalidades são realizadas.

No projeto, utilizamos esse diagrama para representar o fluxo de cadastro e login, seguido pela consulta de KPIs, que pode envolver aplicação de filtros e, quando o usuário é do perfil CFO, a geração de dados financeiros específicos. Em seguida, o sistema prepara a exibição de gráficos e relatórios em tela. Além disso, o diagrama também reflete que o Sistema pode acionar automaticamente o envio de alertas, sem depender da ação direta do usuário.

Esse recurso foi essencial para detalhar a forma como a interface se comunica com os controladores e serviços, garantindo que o fluxo de informações entre as camadas aconteça de forma coerente e organizada. Isso nos ajudou a validar a lógica do processo e a manter a aplicação estruturada e alinhada com o modelo arquitetural definido.



## 5. Conclusão

Este projeto consolidou a aplicação prática de conceitos fundamentais de design e arquitetura de software. A adesão aos princípios SOLID, com destaque para a Responsabilidade Única ou Single Responsibility Principle (SRP), produziu um código mais coeso e de manutenção simplificada. A arquitetura MVC, complementada pelos diagramas UML, forneceu uma visão clara da estrutura e operação do sistema, conectando efetivamente a teoria às demandas reais do projeto.

Isso nos permitiu entender que um bom design de software não é só sobre organizar o código. Ele é a base para que o sistema possa crescer (escalabilidade), ser seguro e continuar evoluindo, o que garante um produto final mais sólido e confiável.