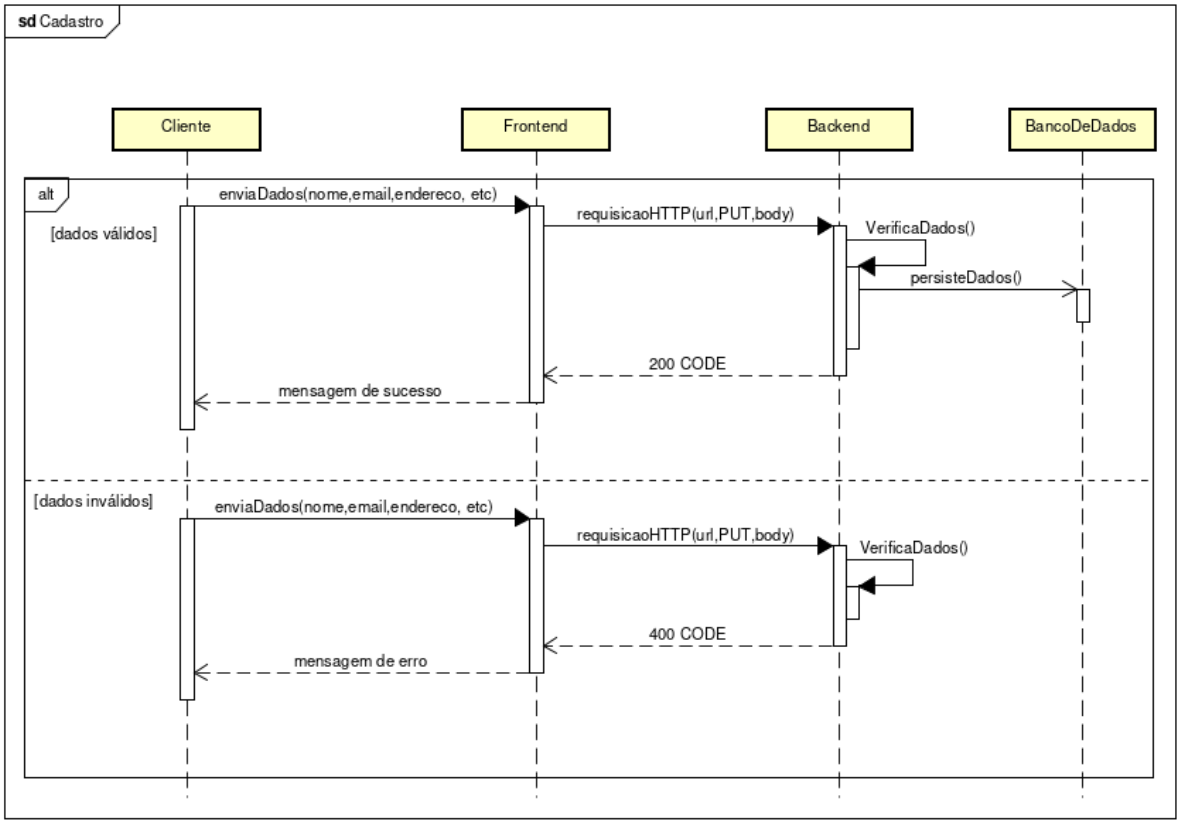
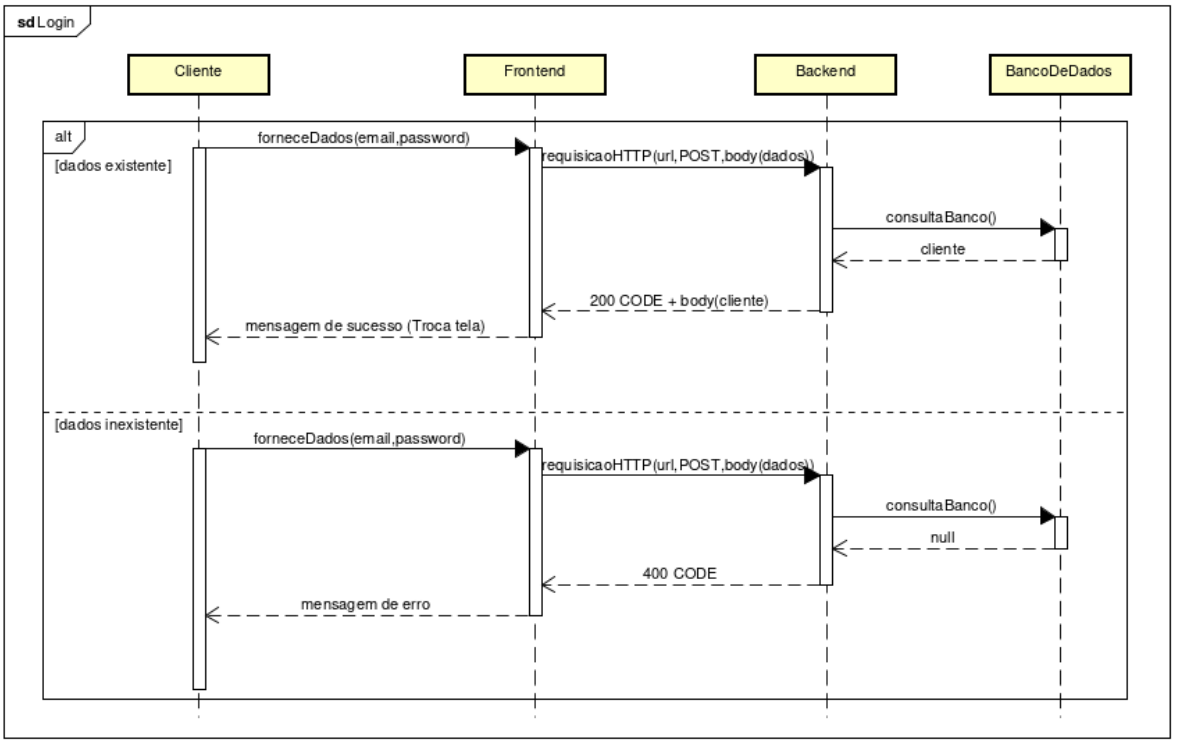
Etapa 5
Diagramas de sequência

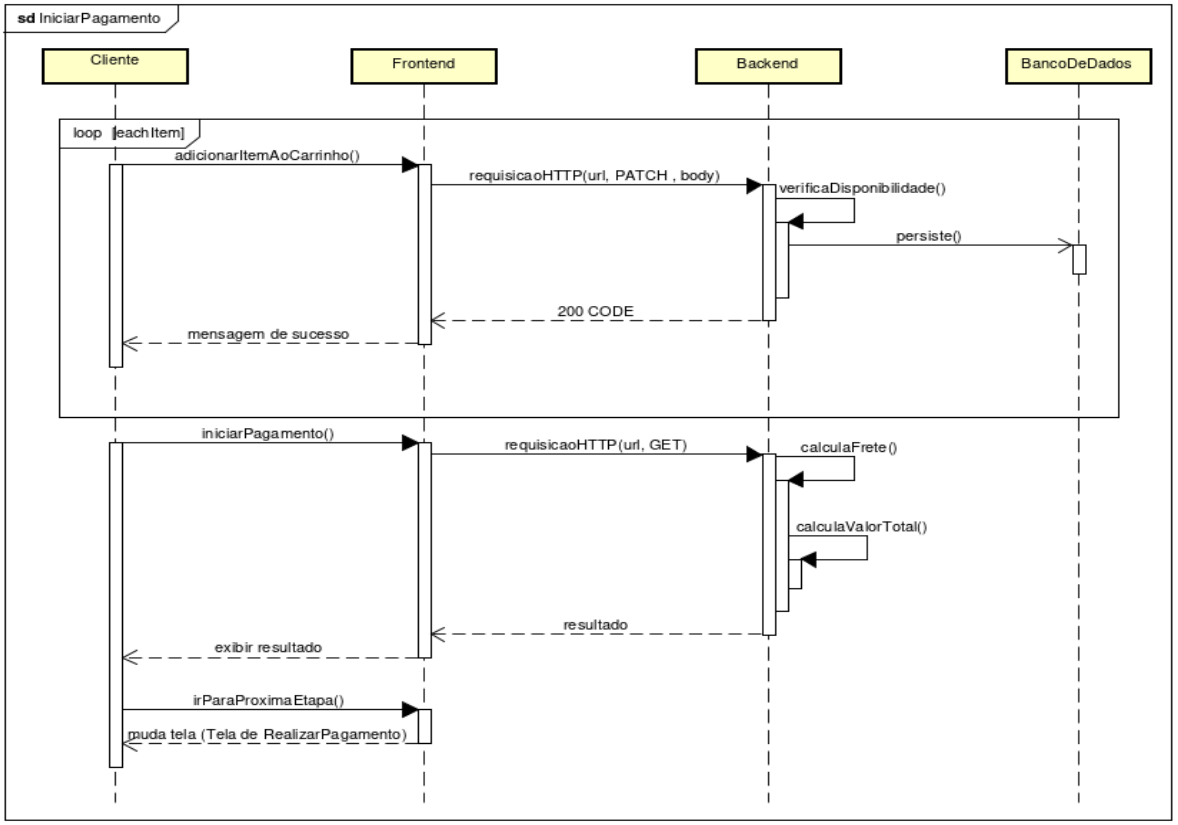
Cadastro:



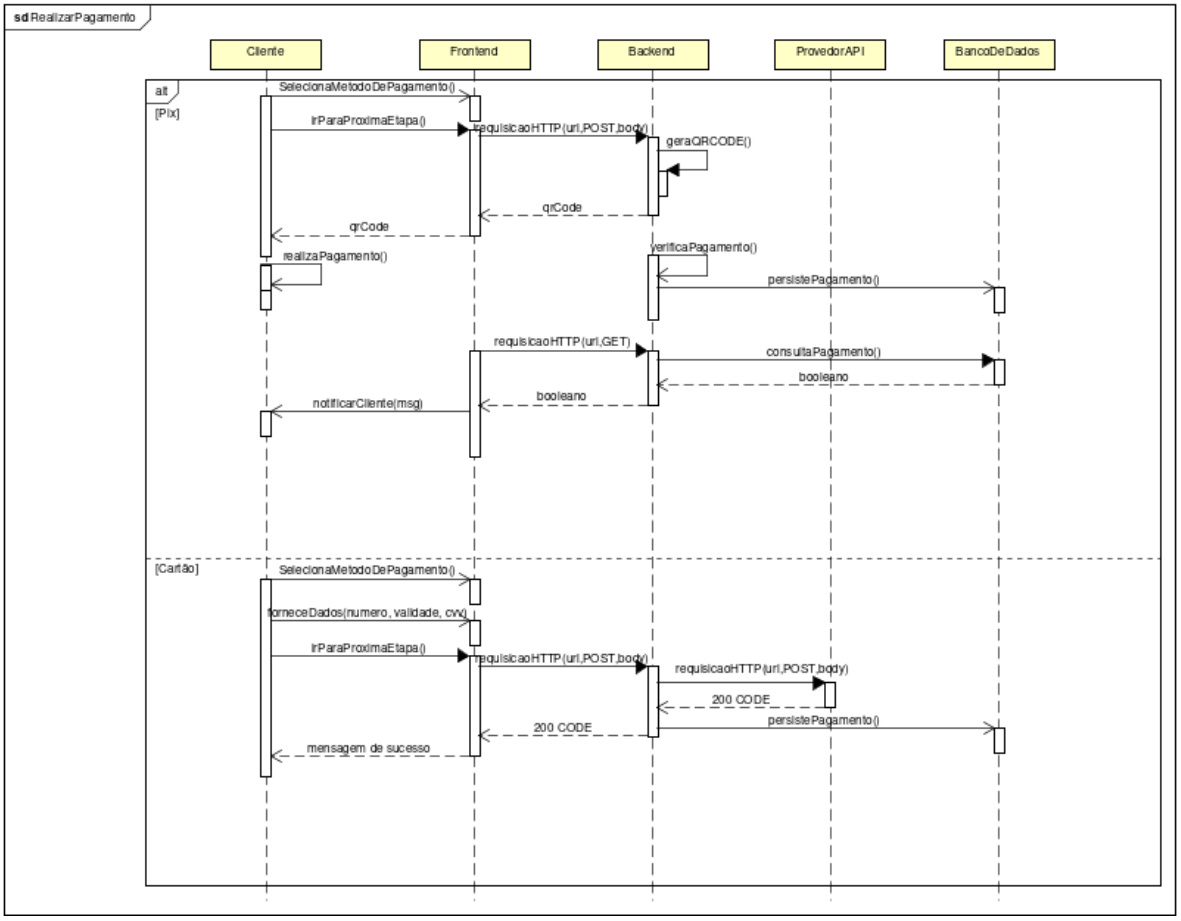
Login:


sd Login
Cliente
Frontend
Backend
BancoDeDados
alt
[dados existente]
forneceDados(email,password)
requisicaoHTTP(url,POST,body(dados))
consultaBanco()
cliente
200 CODE + body(cliente)
mensagem de sucesso (Troca tela)
[dados inexistente]
forneceDados(email,password)
requisicaoHTTP(url,POST,body(dados))
consultaBanco()
null
400 CODE
mensagem de erro

IniciarPagamento:



RealizarPagamento:



Etapa 6

Padrão criacional

Nome do padrão: Factory

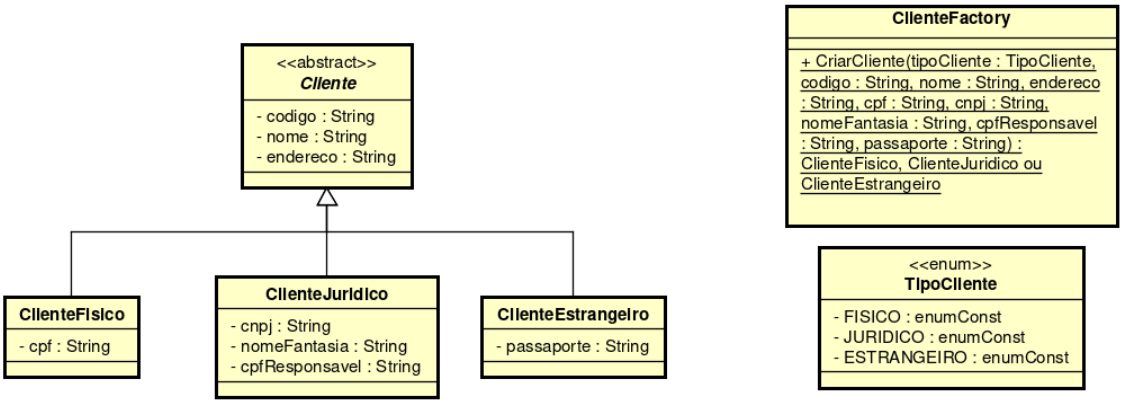
Problema que resolve no contexto escolhido:
O sistema precisa lidar com 3 tipos de clientes, físico, jurídico e estrangeiro, com cada um desses possuindo atributos distintos uns dos outros. Sem uma abordagem centralizada, cada instância desses clientes teriam condicionais espalhadas pelo código, podendo gerar repetições e dificuldade para manutenção ou extensão do código. Exemplo: Por conta das instâncias estarem espalhadas, se ocorrer alterações na forma de instanciar um cliente, como um novo atributo, seria necessário a modificação em vários locais, com o Factory, como toda a criação de clientes está centralizada nele, só seria necessário a alteração do mesmo.

Aplicabilidade do padrão:
Descrição textual: Teremos 5 classes: Cliente, ClienteFisico, ClienteJuridico, ClienteEstrangeiro e Cliente Factory, e 1 enum: TipoCliente.

A classe Cliente é abstrata e será a classe mãe para as classes ClienteFisico, ClienteJuridico e ClienteEstrangeiro, ela conterá todos os atributos que são em comuns nessas 3 classes:
código, nome, endereco, cidade, estado, cep, telefone, email. Assim evitando repetição de código nas classes filhos, estas terão apenas os atributos distintos.
ClienteFisico: cpf.
ClienteJuridico: cnpj, nomeFantasia e cpfResponsavel.
ClienteEstrangeiro: passaporte.

O enum TipoCliente conterá 3 constantes: FISICO, JURIDICO e ESTRANGEIRO, será importante para especificar ao ClienteFactory qual instanciação de Cliente é desejada.

ClienteFactory: Possuirá um método CriarCliente que é static para que possa ser utilizado sem precisar de uma instância de ClienteFactory, esse método recebe como argumento: uma constante enum do TipoCliente e todos os atributos que sejam necessário para instanciar um cliente, seja ele fisico juridico ou estrangeiro, por meio da constante enum TipoCliente ele retornará a instanciação desejada do Cliente. Como a criação de clientes está centralizada, imagine o cenário onde é adicionado um novo tipo de cliente, só precisariamos colocar uma nova constante enum em TipoCliente e realizar uma pequena modificação em ClienteFactory para que ele possa instanciar esse novo tipo de cliente.



obs: Houve abstração de alguns atributos

Padrão comportamental

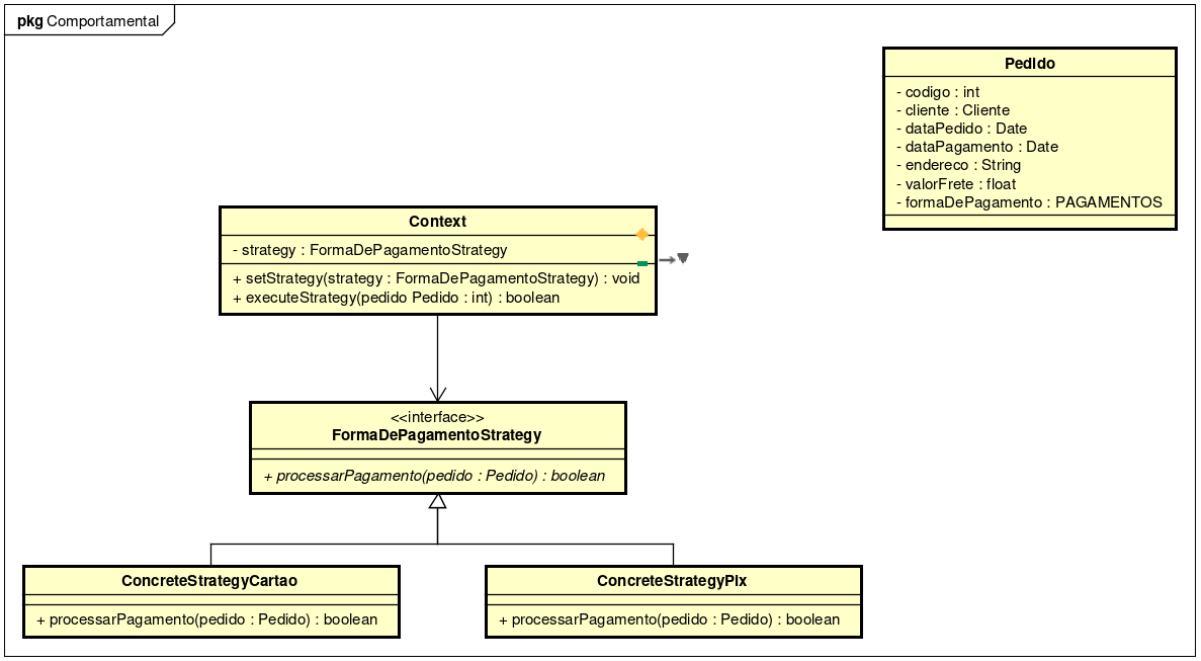
Nome do padrão: Strategy

Problema que resolve no contexto escolhido:
O sistema precisa lidar com diferentes tipos de pagamentos (cartão, PIX, boleto etc.), cada um com lógicas específicas. Se todas essas formas fossem implementadas diretamente na classe principal, a cada nova adição de método de pagamento, o código se tornaria mais extenso e complexo. Além disso, correções de bugs ou ajustes em uma forma específica poderiam impactar inadvertidamente outras partes da classe principal, gerando riscos de novos erros. O Strategy resolve esse problema ao isolar as variações em componentes independentes.

Aplicabilidade do padrão:
Como cada forma de pagamento possui regras únicas, mas compartilha a mesma finalidade (processar um pagamento), o Strategy propõe a criação de uma interface que define um método comum (por exemplo, processarPagamento()). Cada forma de pagamento é então implementada em uma classe específica, que encapsula sua própria lógica.

Para garantir flexibilidade, o padrão utiliza uma classe Contexto, responsável por:
1. Armazenar a estratégia selecionada
2. Delegar a execução para o método correspondente da estratégia (por exemplo, contexto.executarPagamento()).

Assim, o cliente define qual forma de pagamento será usada, o contexto armazena essa informação, e chama o método executarPagamento() da respectiva classe.

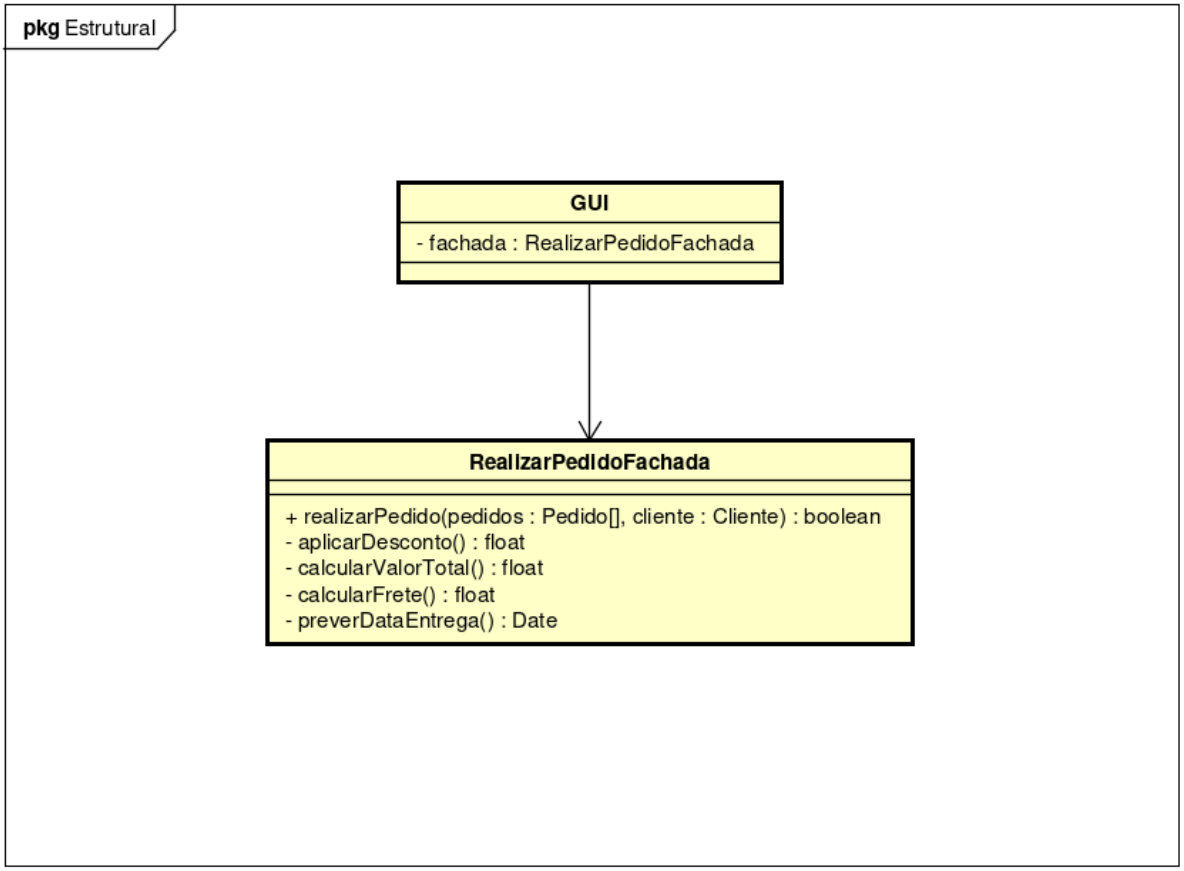


Padrão estrutural

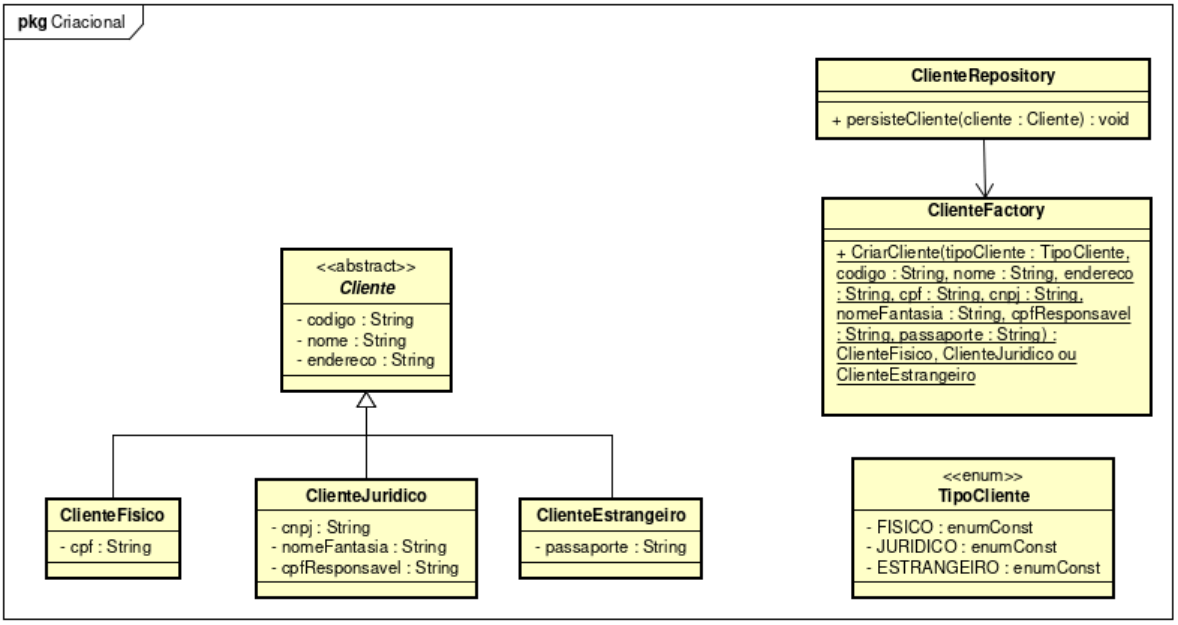
Nome do padrão: Facade

Problema que resolve no contexto escolhido:
Imagine um cenário em que um cliente seleciona vários itens em seu carrinho e, ao clicar no botão "Realizar Pedido" na interface, diversas verificações e cálculos são disparados: cálculo de frete, soma do preço total, aplicação de descontos, previsão da data de entrega, entre outros. Sem o uso de um padrão, essa ação simples pode gerar um código extenso e confuso na classe principal, dificultando a manutenção. O padrão Facade, como o próprio nome sugere, concentra toda essa lógica complexa em uma classe específica, permitindo que a classe principal apenas invoque um método da fachada e receba o resultado, abstraindo toda a complexidade envolvida.

Aplicabilidade do padrão:
Para aplicar o Facade, basta criar uma classe denominada "Fachada" e definir um método público – que será chamado pela interface do usuário – responsável por orquestrar todo o processamento necessário. Esse método pode instanciar novos objetos, utilizar métodos privados, consultar APIs e realizar todas as operações exigidas pela regra de negócio, mantendo o código organizado e de fácil manutenção.



Etapa 7:
Responsabilidade Única: Imagine o cenário onde uma classe persiste um cliente no banco de dados, **e** além disso ela é responsável por instanciar a Classe correta para o tipo de usuário (física, jurídica, estrangeira) para facilitar o JSON do cliente na requisição HTTP. Perceba que essa classe faz duas coisas, o que quebra o 1° princípio SOLID, o que podemos fazer é criar uma nova classe que seja responsável apenas para a criação do tipo de cliente correto, e deixar a nossa classe antiga apenas com a persistência no banco de dados.



Aberto / Fechado: Esse princípio relata que uma classe deve ser fechada para modificações, porém aberta para extensões. Imagine o cenário de formas de pagamento, temos várias delas, em vez de ter uma classe centralizando tudo, onde ela processa pagamentos dependendo de qual forma de pagamento foi escolhida, assim tendo um único método, porém para cada forma de pagamento terá que ser adicionado um if/else novo, o que quebra o princípio de ser fechada para modificações. O que podemos fazer é: Transformar essa classe que contém o método de processar pagamento como uma interface, onde criaremos novas classes (aberta para extensão), essas classes implementam o método de processar pagamento conforme a sua própria lógica de negócio.