## linha horizontal

Sistema de gerenciamento e academia

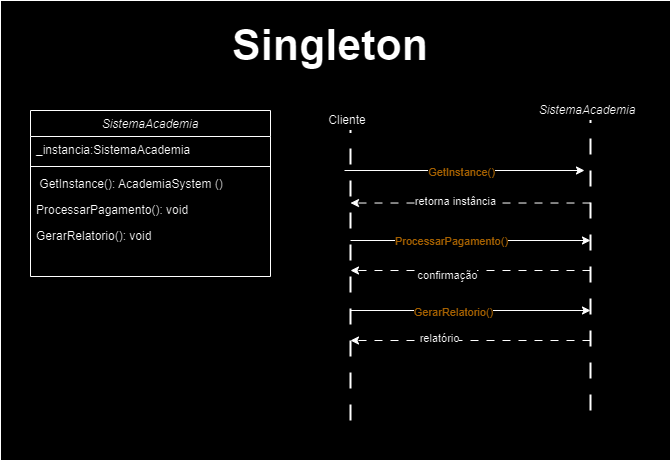
**─**

Kayky Matos Santana, RGM: 33345945

**1. Singleton**

No **Diagrama de Classes**, o padrão Singleton é representado de forma simples:

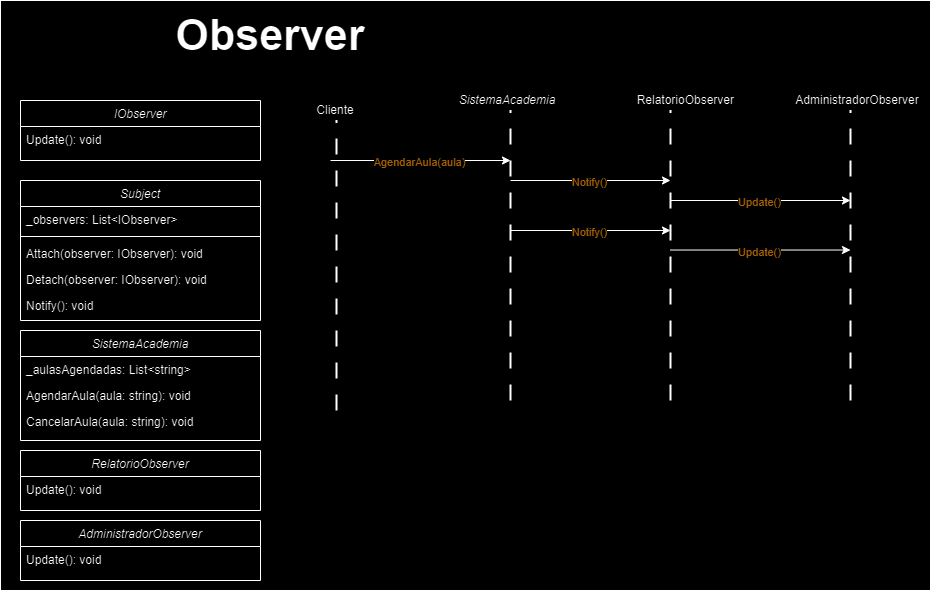
* A classe AcademiaSystem é responsável por gerenciar a única instância.
* Ela tem um método estático GetInstance() que retorna a instância única da classe.
* O construtor da classe é privado, para garantir que não possam ser criadas múltiplas instâncias.



**2.Observer**

O padrão **Observer** é um design pattern comportamental que define uma relação de dependência entre objetos, onde um ou mais objetos (chamados de "observadores") são notificados automaticamente sempre que o estado de um outro objeto (o "sujeito" ou "observable") muda. Isso é útil em sistemas onde várias partes precisam reagir a eventos ou mudanças de estado em outro componente.

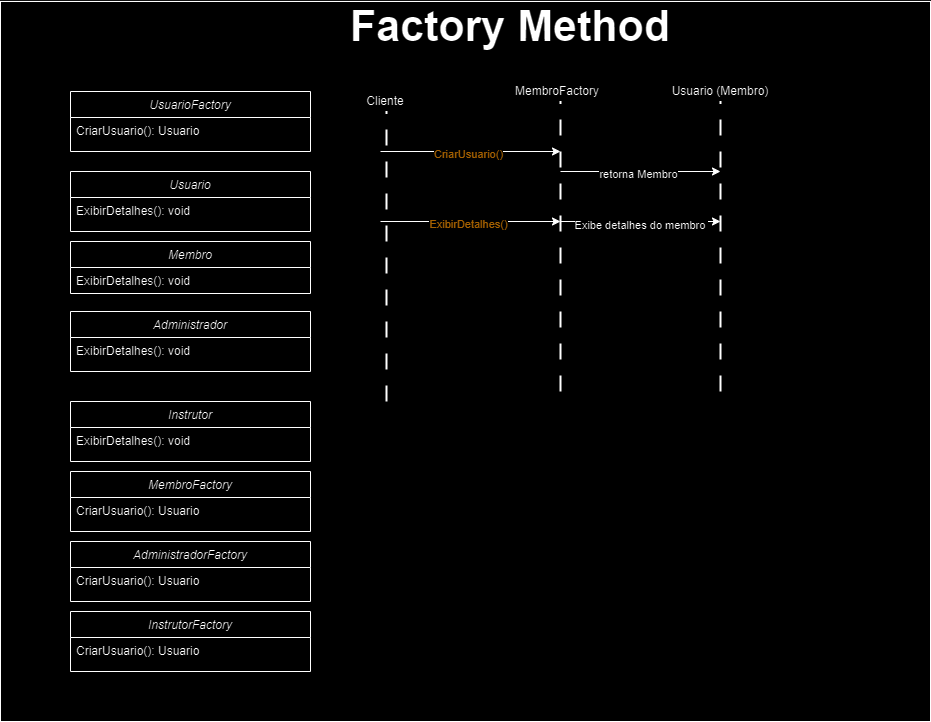
No contexto da "Academia XYZ", o padrão Observer poderia ser aplicado para notificar automaticamente quando um membro agenda ou cancela uma aula, ou quando o pagamento de um membro é atualizado, permitindo que os interessados (administradores, sistema de relatórios, etc.) sejam informados dessas mudanças.



**3.Factory Method**

O padrão **Factory Method** é um design pattern criacional que define uma interface para criar objetos, mas deixa que as subclasses decidam quais classes concretas serão instanciadas. Ele é útil quando o código precisa criar objetos, mas não quer se acoplar diretamente às classes concretas.

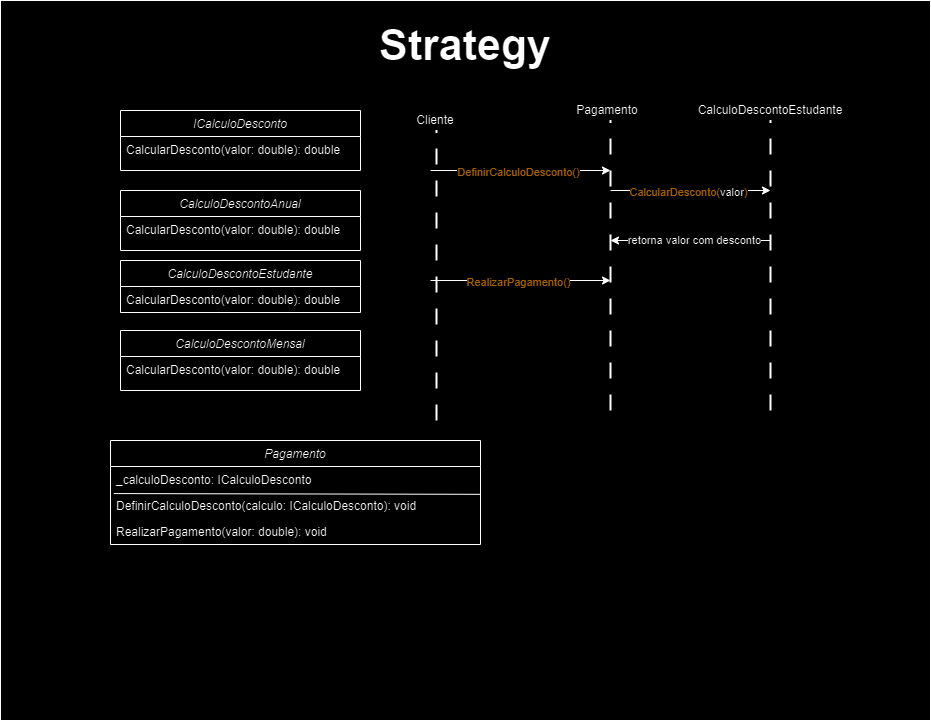
No sistema da "Academia XYZ", o **Factory Method** poderia ser usado para criar diferentes tipos de usuários (como membros, administradores, ou instrutores) ou para instanciar diferentes tipos de relatórios (financeiro, de frequência, etc.), dependendo da necessidade.



**4.Strategy**

O padrão **Strategy** define uma família de algoritmos, encapsula cada um deles e os torna intercambiáveis. O padrão permite que o algoritmo varie independentemente dos clientes que o utilizam. É útil quando você tem múltiplas maneiras de executar uma operação e quer escolher dinamicamente qual método usar.

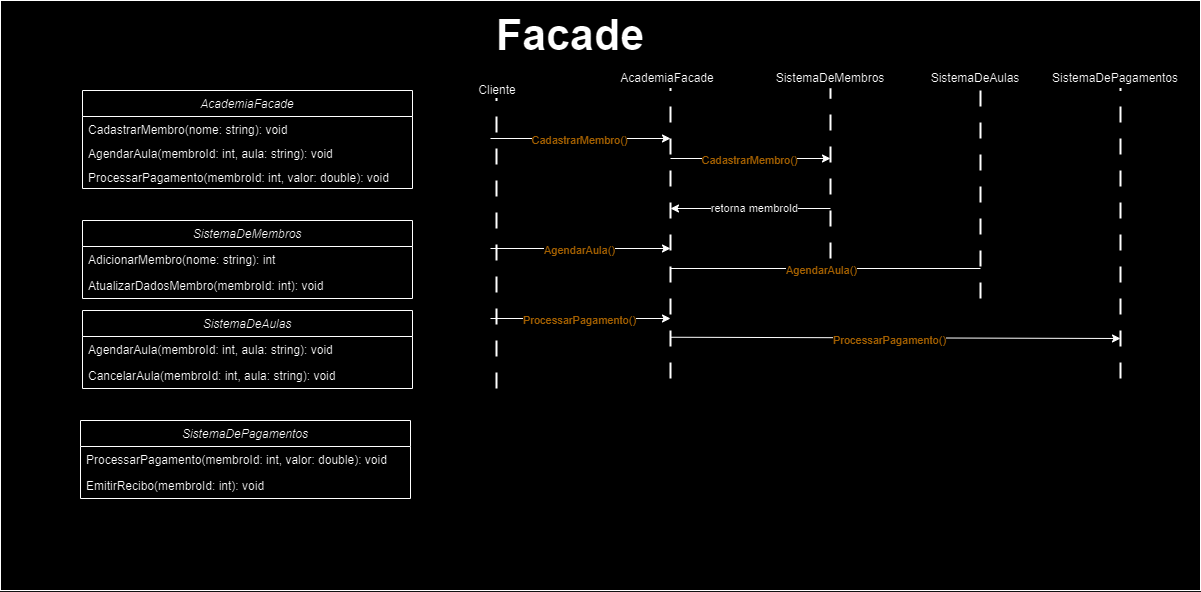
No sistema da "Academia XYZ", o padrão **Strategy** poderia ser aplicado para diferentes estratégias de cálculo de descontos para os membros da academia. Por exemplo, diferentes estratégias poderiam ser aplicadas para estudantes, membros que pagam anualmente ou membros que pagam mensalmente.



**5.Facade**

O padrão **Facade** fornece uma interface simplificada para um subsistema complexo. Em vez de o cliente interagir diretamente com várias classes de um sistema, ele pode usar uma classe **Facade** que abstrai as complexidades internas e oferece métodos simples para as operações mais comuns.

No sistema da "Academia XYZ", o padrão **Facade** poderia ser usado para simplificar as operações de **gerenciamento de membros**, **agendamento de aulas** e **processamento de pagamentos**. Em vez de o cliente interagir diretamente com várias classes que gerenciam essas funcionalidades, a classe Facade poderia fornecer métodos simplificados para realizar essas operações.



6.Decorator

O padrão **Decorator** é um design pattern estrutural que permite adicionar funcionalidades a objetos de forma dinâmica e flexível, sem alterar a estrutura das classes originais. Esse padrão é útil quando você deseja estender a funcionalidade de uma classe sem usar a herança, permitindo a combinação de diferentes comportamentos de forma flexível.

No contexto da "Academia XYZ", o padrão **Decorator** poderia ser usado para adicionar diferentes funcionalidades a um membro, como pacotes de aulas, personal trainer, ou acesso a áreas especiais, sem a necessidade de criar subclasses específicas para cada combinação.

