# Especialização em Desenvolvimento Java

## Padrões de Projeto

Prof. Vinícius de Paula

https://github.com/viniciusdepaula/aulas-uml-padroes

Unitri
CENTRO UNIVERSITÁRIO DO TRIÂNGULO

# Padrões Estruturais

# Padrões Estruturais

- As interações entre os objetos de um sistema podem gerar fortes dependências entre esses elementos.
- Essas dependências aumentam a complexidade das eventuais alterações no funcionamento do sistema.
- Consequentemente, o custo de manutenção aumenta.
- Os padões estruturais diminuem o acoplamento entre os objetos de um sistema orientado a objetos.

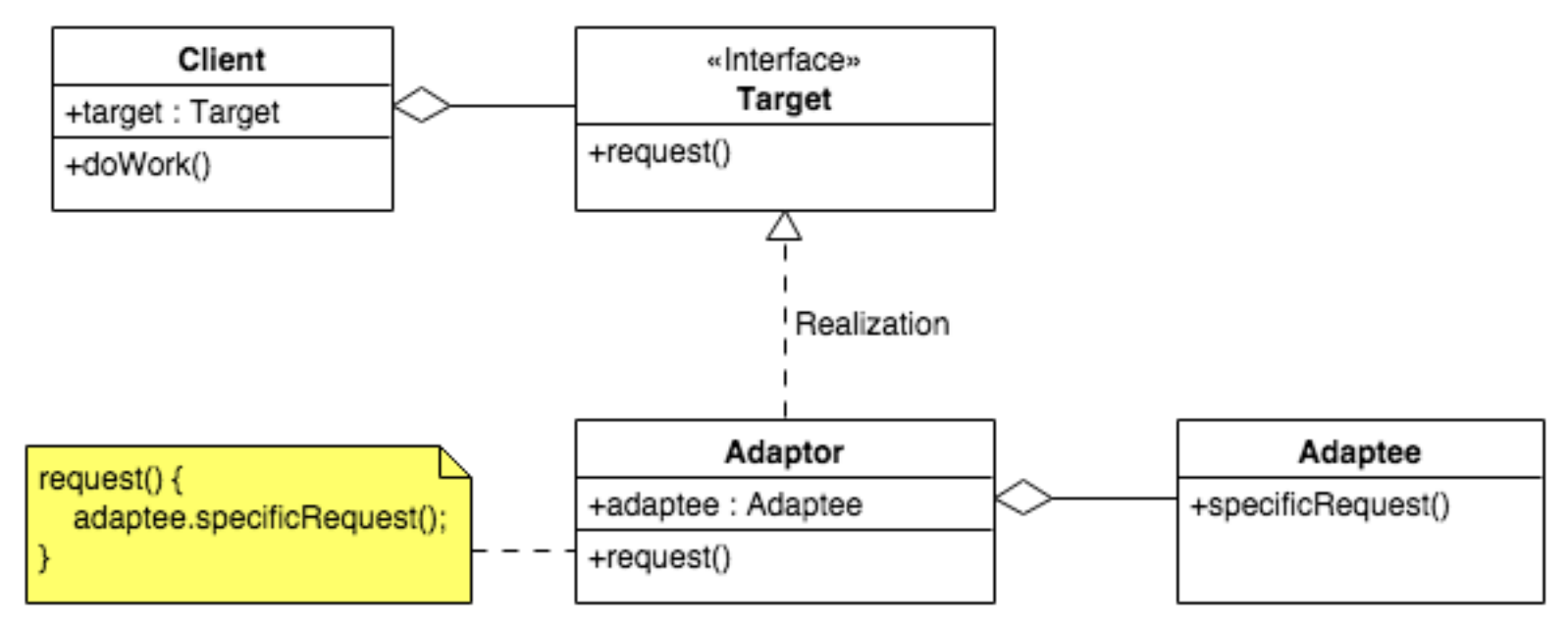
# Adapter

# Adapter

**Tem como objetivo:**

- Permitir que um objeto seja substituído por outro que, apesar de realizar a mesma tarefa, possui uma interface diferente.

# Adapter


Client
+target : Target
+doWork()
«Interface»
Target
+request()
Realization
Adaptor
+adaptee : Adaptee
+request()
Adaptee
+specificRequest()
request() {
adaptee.specificRequest();
}

# Adapter

**Exemplo prático:**

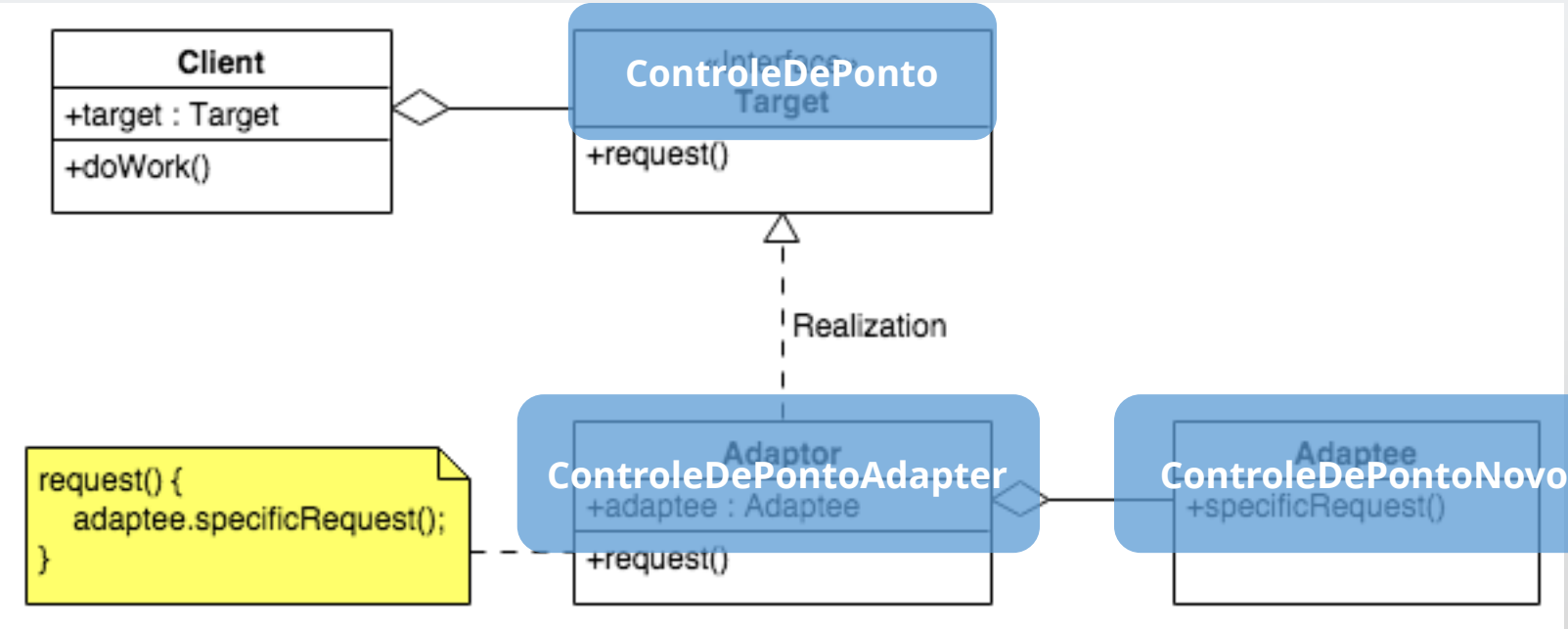
- Estamos realizando manutenção no sistema de gerenciamento de uma determinada empresa. O controle de ponto desse sistema possui diversas limitações. Essas limitações causam muitos prejuízos. Principalmente, prejuízos financeiros.
- Uma empresa parceira implementou uma biblioteca Java para controlar a entrada e saída dos funcionários. Essa biblioteca não possui as limitações que existem hoje no sistema que estamos realizando manutenção. Os diretores decidiram que a melhor estratégia seria adquirir essa biblioteca e implantá-la no sistema.

# Adapter

**Exemplo prático:**

- Para implantar essa biblioteca, teremos que substituir as classes que atualmente cuidam do controle de ponto pelas classes dessa biblioteca. A complexidade dessa substituição é alta pois os métodos das classes antigas não são compatíveis com os métodos das classes novas. Em outras palavras, as interfaces são diferentes.

# Adapter


Client
+target : Target
+doWork()
ControleDePonto
Target
+request()
Realization
request() {
adaptee.specificRequest();
}
ControleDePontoAdapter
Adaptor
+adaptee : Adaptee
+request()
ControleDePontoNovo
Adaptee
+specificRequest()

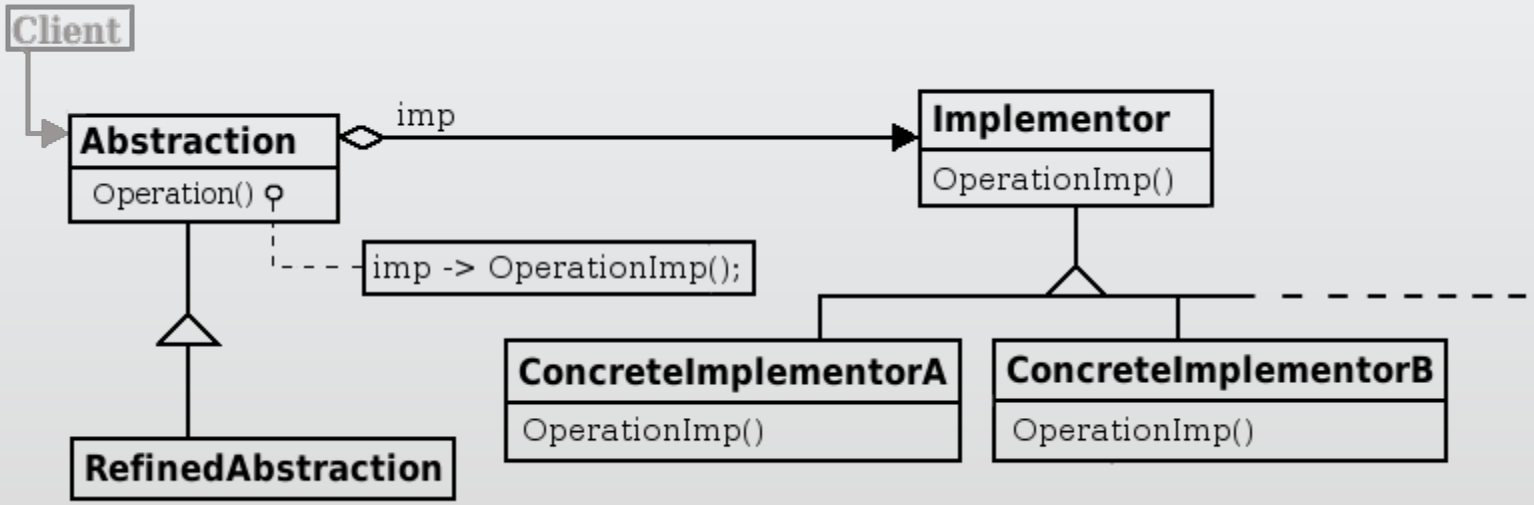

Bridge

# Bridge

**Tem como objetivo:**

- Separar uma abstração de sua representação, de forma que ambos possam variar e produzir tipos de objetos diferentes.

# Bridge

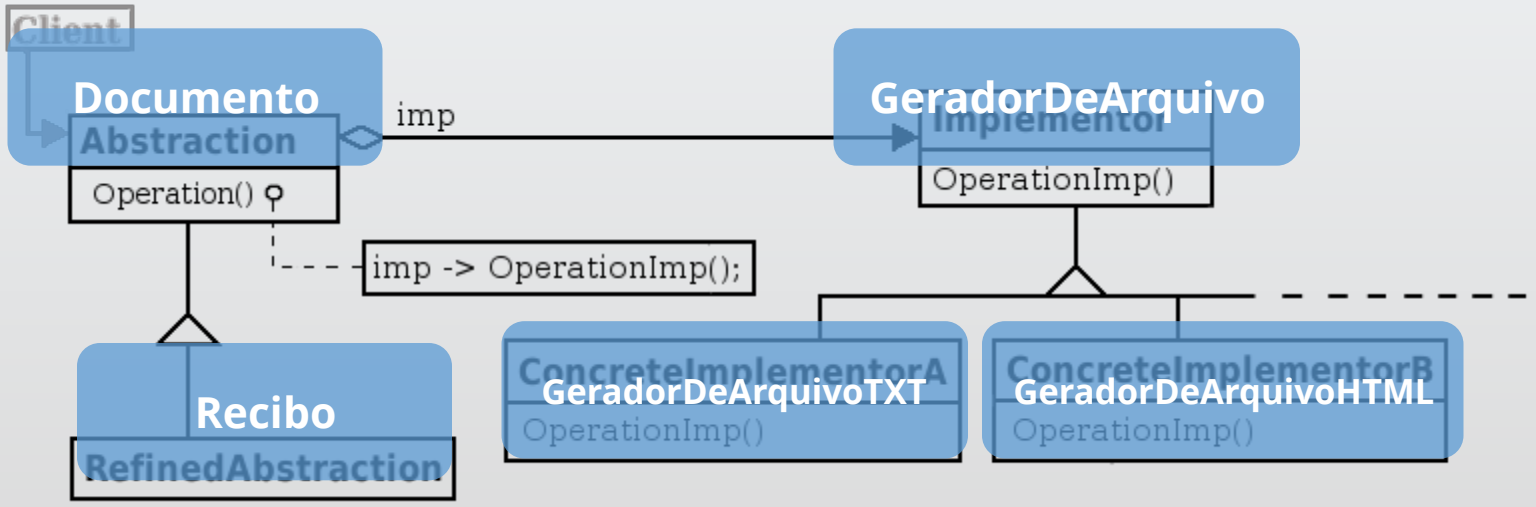

Client
Abstraction
Operation()
imp
Implementor
OperationImp()
imp -> OperationImp();
ConcreteImplementorA
OperationImp()
ConcreteImplementorB
OperationImp()
RefinedAbstraction

# Bridge

**Exemplo prático:**

- Estamos desenvolvendo um sistema que deve gerar diversos tipos de documentos (recibos, atestados, comunicados, etc) em diversos formatos de arquivos (txt, html, pdf, etc).

# Bridge


Client
Documento
Abstraction
Operation()
imp
GeradorDeArquivo
Implementor
OperationImp()
imp -> OperationImp();
Recibo
RefinedAbstraction
ConcreteImplementorA
GeradorDeArquivoTXT
OperationImp()
ConcreteImplementorB
GeradorDeArquivoHTML
OperationImp()

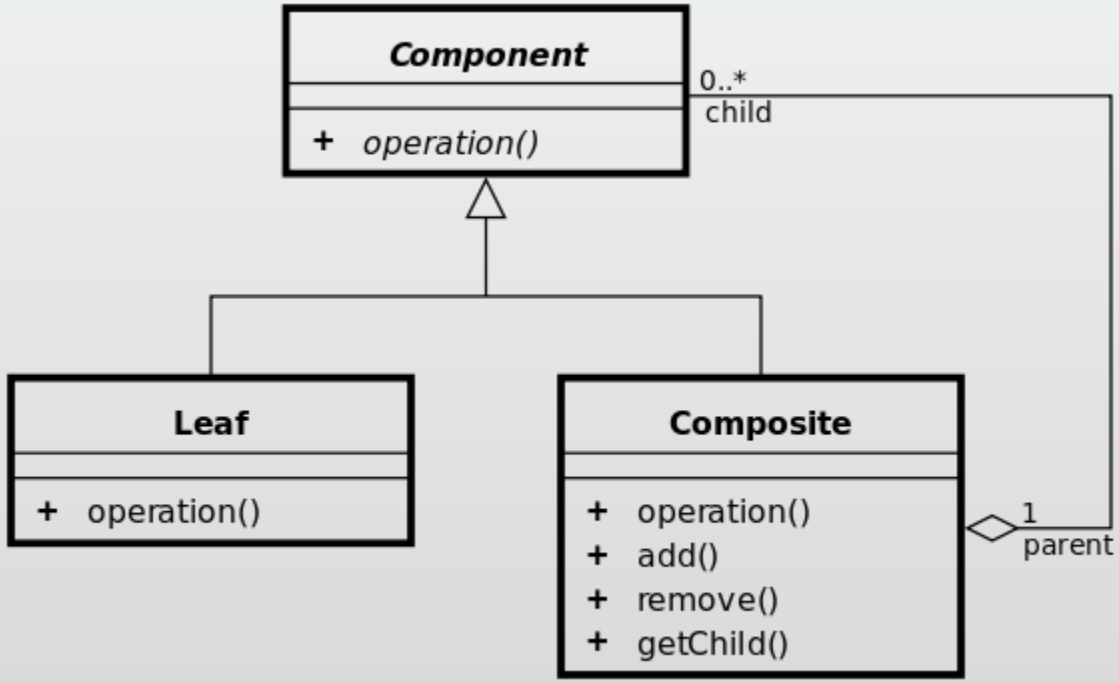

Composite

# Composite

**Tem como objetivo:**

- Agrupar objetos que fazem parte de uma relação parte-todo de forma a tratá-los sem distinção.

# Composite

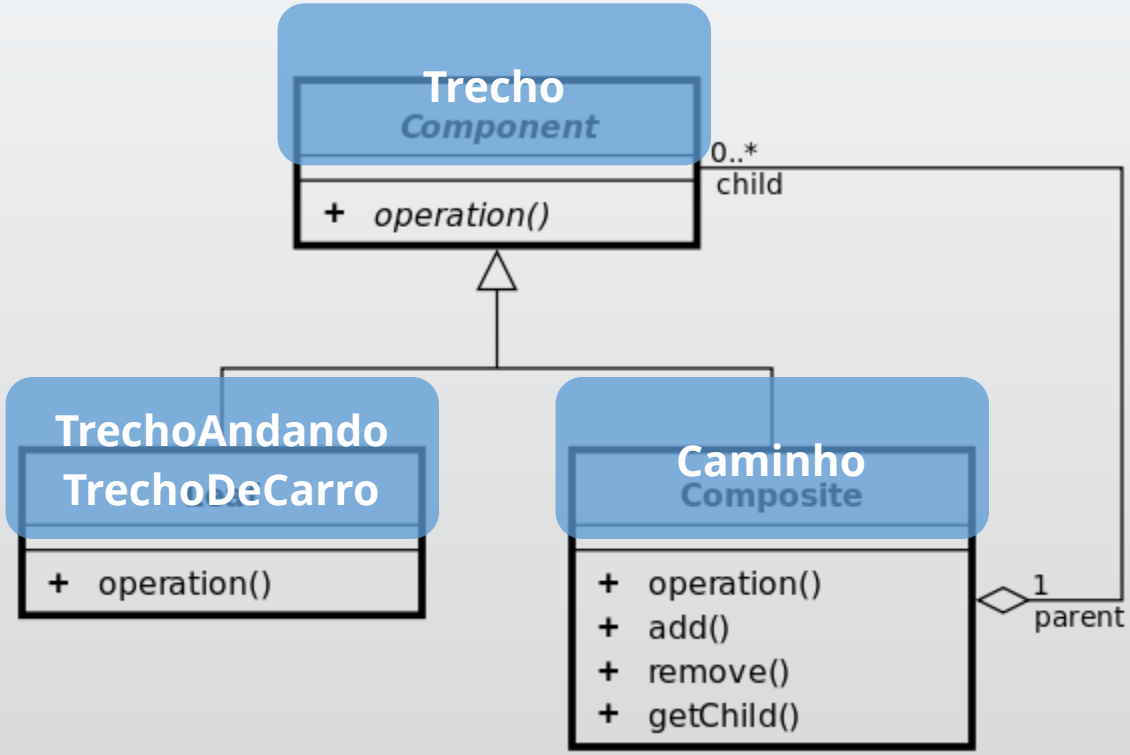

Component
+ operation()
0..*
child
Leaf
+ operation()
Composite
+ operation()
+ add()
+ remove()
+ getChild()
1
parent

# Composite

**Exemplo prático:**

- Suponha que estamos desenvolvendo um sistema para calcular um caminho entre quaisquer dois pontos do mundo. Um caminho pode ser percorrido de diversas maneiras: à pé, de carro, de ônibus, de trem, de avião, de navio, etc.
- O sistema deve apresentar graficamente para os usuários as rotas que forem calculadas. Cada tipo de trecho deve ser apresentado de uma maneira específica. Por exemplo, se o trecho for de caminhada então deve aparecer na impressão da rota a ilustração de uma pessoa andando.
- Cada tipo de trecho pode ser implementado por uma classe e seria interessante definir uma interface para padronizá-las.

# Composite


Trecho
Component
+ operation()
0..*
child
TrechoAndando
TrechoDeCarro
+ operation()
Caminho
Composite
+ operation()
+ add()
+ remove()
+ getChild()
1
parent

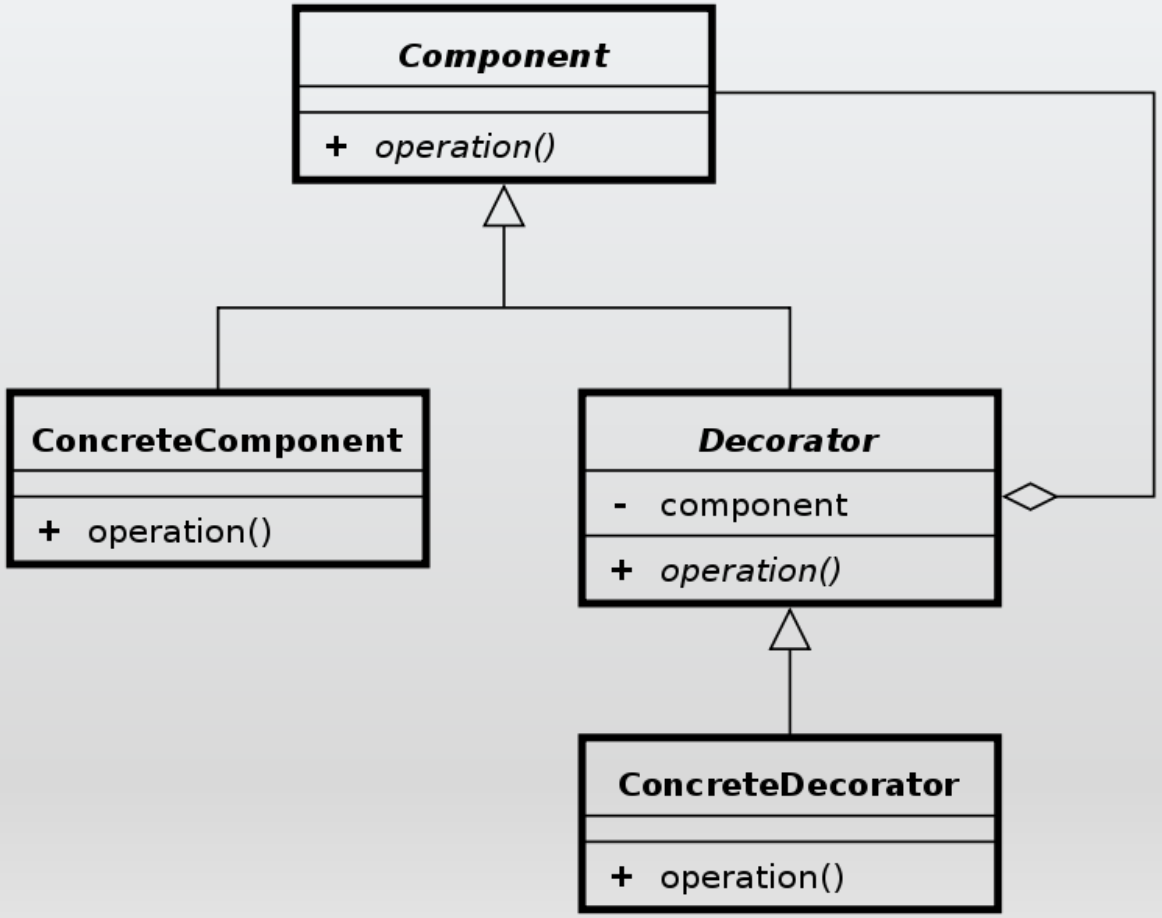

Decorator

# Decorator

**Tem como objetivo:**

- Adicionar funcionalidade a um objeto dinamicamente.

# Decorator


Component
+ operation()
ConcreteComponent
+ operation()
Decorator
- component
+ operation()
ConcreteDecorator
+ operation()

# Decorator

**Exemplo prático:**

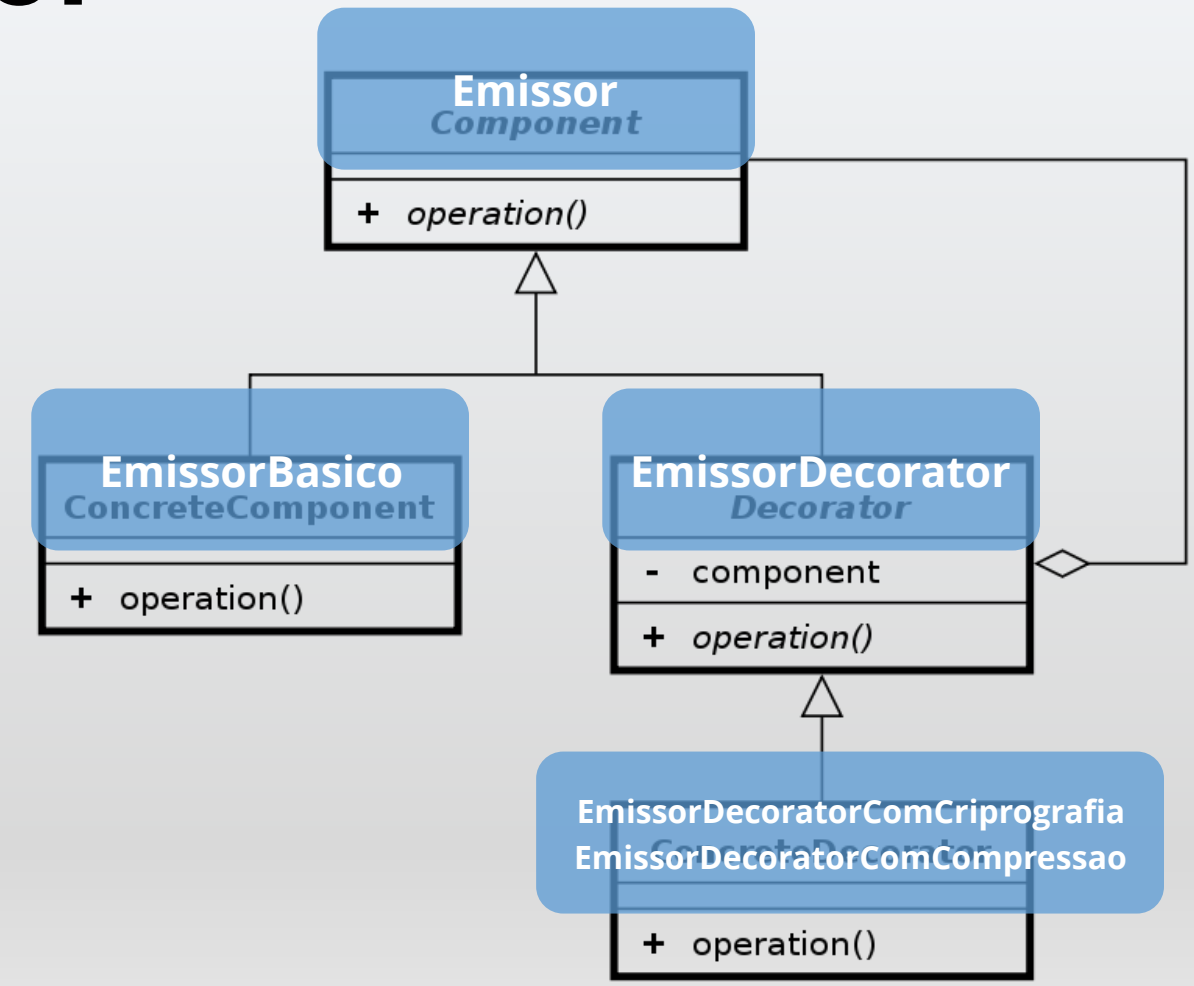
- Como exemplo prático do padrão Factory Method, consideramos um sistema de envio de mensagens. Nesse exemplo, definimos uma interface para padronizar os emissores.
- Agora, suponha que estejamos interessados em adicionar algumas funcionalidades no processo de envio de mensagem. Tais funcionalidades incluem criptografia e compressão das mensagens.
- Para não alterar as classes que definem os emissores, cada funcionalidade adicional (decoração) será implementada por um novo objeto (decorador).

# Decorator

**Exemplo prático:**

- Quando queremos enviar uma mensagem, não podemos chamar diretamente os emissores, pois as funcionalidades adicionais não serão executadas.
- Portanto, devemos entregar a mensagem a um decorador, que executará a tarefa para a qual foi concebido.
- O decorador, por sua vez, terá também a responsabilidade de repassar a mensagem a um emissor para que ela seja enviada. Dessa forma, todo decorador deve possuir um emissor.

# Decorator


Emissor
Component
+ operation()
EmissorBasico
ConcreteComponent
+ operation()
EmissorDecorator
Decorator
- component
+ operation()
EmissorDecoratorComCriprografia
EmissorDecoratorComCompressao
+ operation()

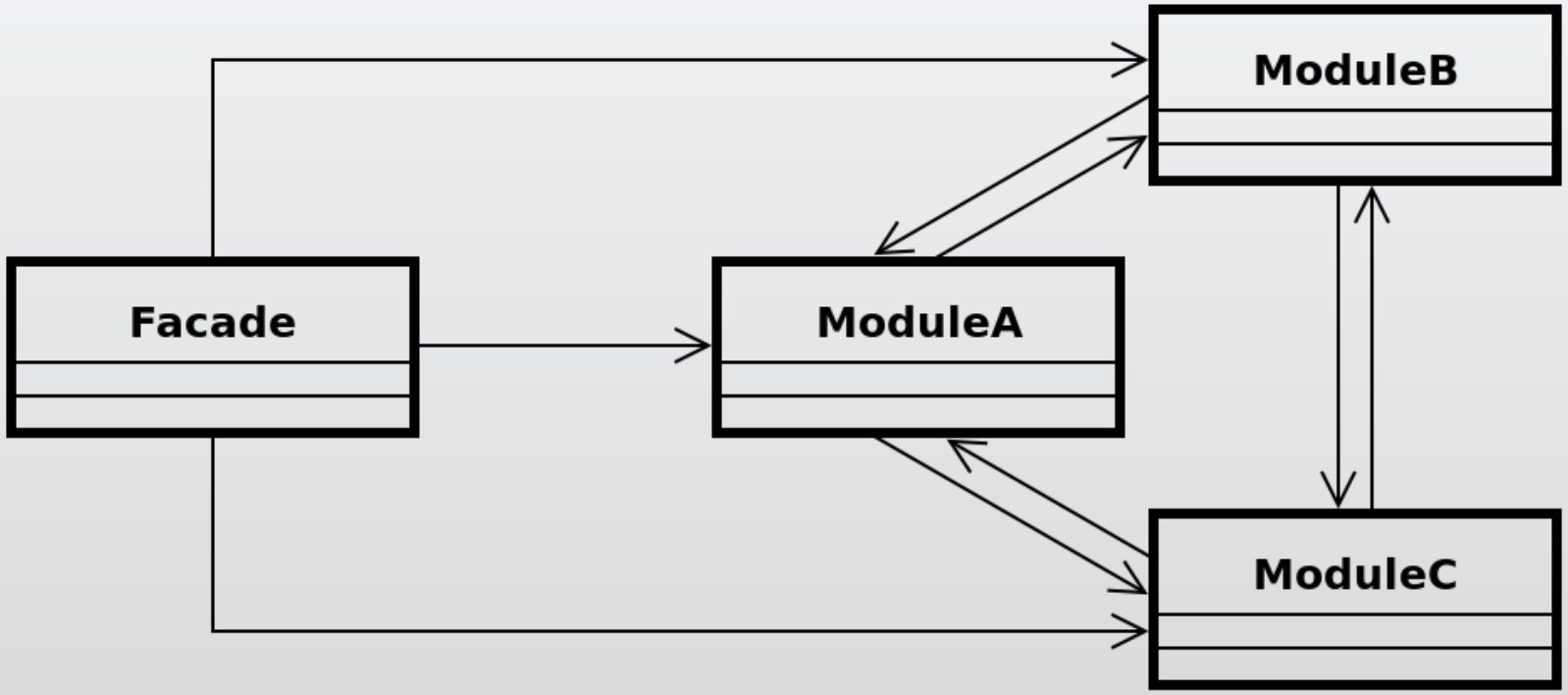

Facade

# Facade

**Tem como objetivo:**

- Prover uma interface simplificada para a utilização de várias interfaces de um subsistema.

# Facade



# Facade

**Exemplo prático:**

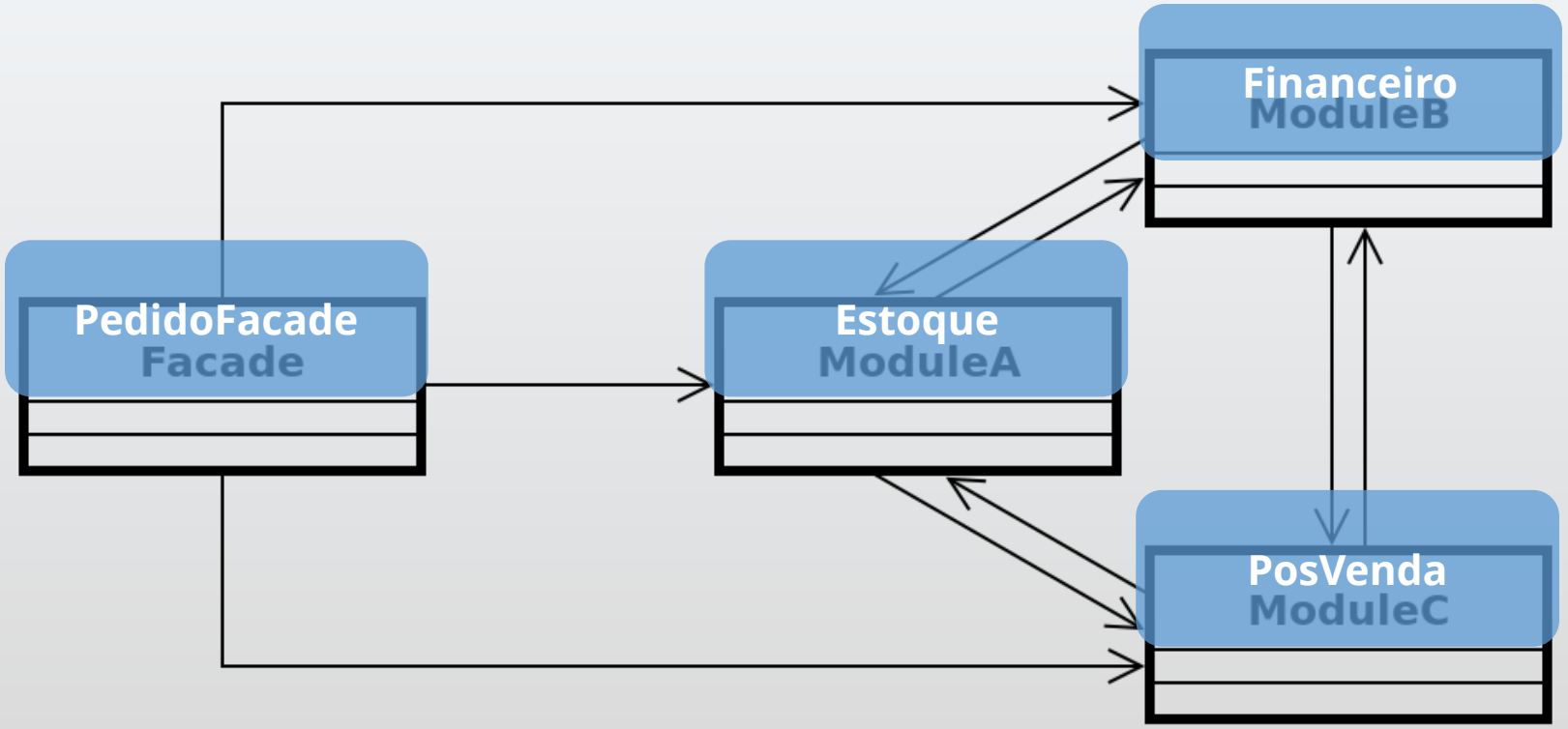
- Estamos melhorando um sistema que realiza todos os procedimentos que devem ser realizados após o registro de um pedido. Quando um pedido é realizado, o módulo que gerencia o estoque deve ser avisado para que o produto seja encaminhado ao endereço de entrega.
- O módulo financeiro deve ser avisado para que o processo de faturamento seja realizado.
- O módulo de pós venda também deve ser avisado para que contatos futuros sejam realizados com o cliente com o intuito de verificar a satisfação do mesmo com o produto obtido.

# Facade

**Exemplo prático:**

- O sistema já está funcionando e realiza todos os processos decorrentes da realização de um novo pedido. Mas, queremos simplificar essa lógica encapsulando as chamadas aos módulos de estoque, financeiro e de pós venda.
- A ideia é criar uma classe que encapsula todos os processos que envolvem o acesso aos módulos de estoque, financeiro e de pós venda.

# Facade


Financeiro
ModuleB
PedidoFacade
Facade
Estoque
ModuleA
PosVenda
ModuleC

# Bibliografia


- Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson e Vlissides, Addison-Wesley, 1995.(Padrões de Projeto - Soluções Reutilizáveis de Software Orientado a Objeto - Gamma, Helm, Johnson e Vlissides, Bookman, 2000.
- Patterns of Enterprise Application Architecture, Fowler, Addison Wesley, 2003.