

Computação Orientada a Objetos

Padrões de Projeto

Profa. Karina Valdivia Delgado
EACH-USP

1

Catálogo de soluções

- Um padrão encerra o conhecimento de uma pessoa muito experiente em um determinado assunto.
- Esse conhecimento pode ser transmitido para outras pessoas menos experientes.
 - Outras ciências (p.ex. química) e engenharias possuem catálogos de soluções.
- Desde 1995, o desenvolvimento de software passou a ter o seu primeiro catálogo de soluções para projeto de software: o livro GoF.

PADRÕES DE PROJETO

- Passamos a ter um **vocabulário comum** para conversar sobre projetos de software.
- Soluções que não tinham nome passam a ter nome.
- Ao invés de discutirmos um sistema em termos de pilhas, filas, árvores e listas ligadas, passamos a **falar de coisas de muito mais alto nível** como:
 - Fábricas
 - Fachadas
 - Observador
 - Estratégia, etc.

FORMATO DO PADRÃO GOF

1.Nome

- expressa a própria essência do padrão de forma sucinta

1.Objetivo / Intenção

2.Também conhecido como (AKA)

3.Motivação

- cenário mostrando o problema e a necessidade da solução

1.Aplicabilidade

- situações em que o padrão é aplicável

1.Estrutura

- representação gráfica da estrutura de classes do padrão

FORMATO DO PADRÃO GOF

7.Participantes

- classes e objetos que participam e suas responsabilidades

1.Colaborações

- como os participantes colaboram para exercer suas responsabilidades

7.Consequências

- custos e benefícios da utilização do padrão

7.Implementação

- quais são os detalhes importantes na hora de implementar o padrão
- aspectos específicos de cada linguagem

FORMATO DO PADRÃO GOF

10.Exemplo de Código

- no caso do GoF, em C++ (a maioria) ou Smalltalk

10.Usos Conhecidos

- exemplos de sistemas reais em que o padrão é utilizado

10.Padrões Relacionados

- quais outros padrões devem ser usados em conjunto com esse
- quais padrões são similares a este, quais são as diferenças

TIPOS DE PADRÕES DE PROJETO do GoF

- Padrões de Criação:
 - são utilizados para criar classes e objetos
 - ajudam a tornar um sistema independente de como seus objetos são criados
- Padrões Estruturais:
 - preocupam-se com a forma com que objetos e classes são compostos para formar estruturas mais complexas.
 - utilizam mecanismos de herança e composição.
- Padrões Comportamentais:
 - preocupam-se com algoritmos e a atribuição de responsabilidades entre objetos.

Os 23 Padrões do GoF

Criação

- ▶ Processo de criação de objetos
 - Abstract Factory
 - Builder
 - Factory Method
 - Prototype
 - Singleton

Os 23 Padrões do GoF

Estruturais

- ▶ Composição de classes ou objetos para formar estruturas complexas
 - Adapter
 - Bridge
 - Composite
 - Decorator
 - Façade
 - Flyweight
 - Proxy

Os 23 Padrões do GoF

Comportamentais

- Forma com que classes ou objetos interagem e distribuem responsabilidades.

- 
- ▶ Chain of Responsibility
 - ▶ Command
 - ▶ Interpreter
 - ▶ Iterator
 - ▶ Mediator
 - ▶ Memento

- ▶ Observer
- ▶ State
- ▶ Strategy
- ▶ Template Method
- ▶ Visitor



PADRÕES DE CRIAÇÃO



PADRÃO SINGLETON

PADRÃO SINGLETON

- Padrão de Criação.
- **Objetivo / Intenção**
 - Garantir que uma classe tenha apenas uma única instância, independente do número de requisições que receber para criá-la.
 - Fornecer um ponto global de acesso para essa instância.
 - A instância é criada somente no momento da sua primeira requisição (*lazy instantiation*).

PADRÃO SINGLETON

○ Motivação:

- Em alguns sistemas, é necessário ter apenas uma instância de uma determinada classe e um ponto de acesso global é necessário.
- Por exemplo:
 - Um único banco de dados
 - Um único acesso a um arquivo de log
 - Uma única configuração de jogo
 - Uma única central de controle das eleições.

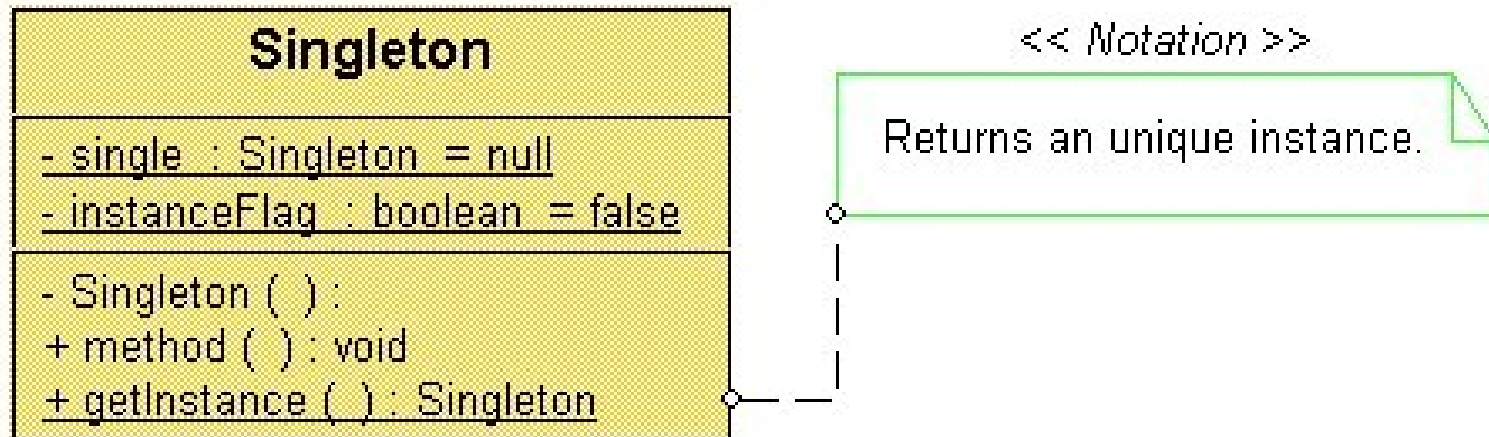
PADRÃO SINGLETON

○ Aplicabilidade

- Este padrão é utilizado quando é necessário haver apenas uma instância de uma classe e essa instância tiver que dar acesso aos clientes através de um ponto bem conhecido.
- A única instância pode ser extensível através de subclasses, possibilitando aos clientes usar uma instância estendida sem alterar o seu código.

PADRÃO SINGLETON

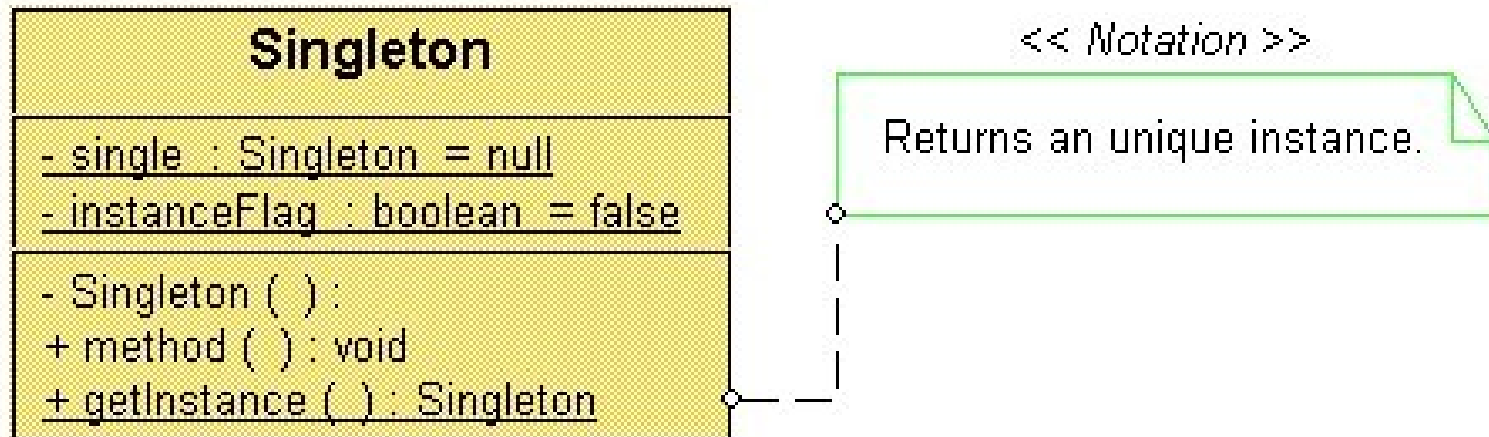
○ Estrutura



- O **construtor** do Singleton deve ser **privado**, para evitar que outras classes possam instanciar a classe diretamente.

PADRÃO SINGLETON

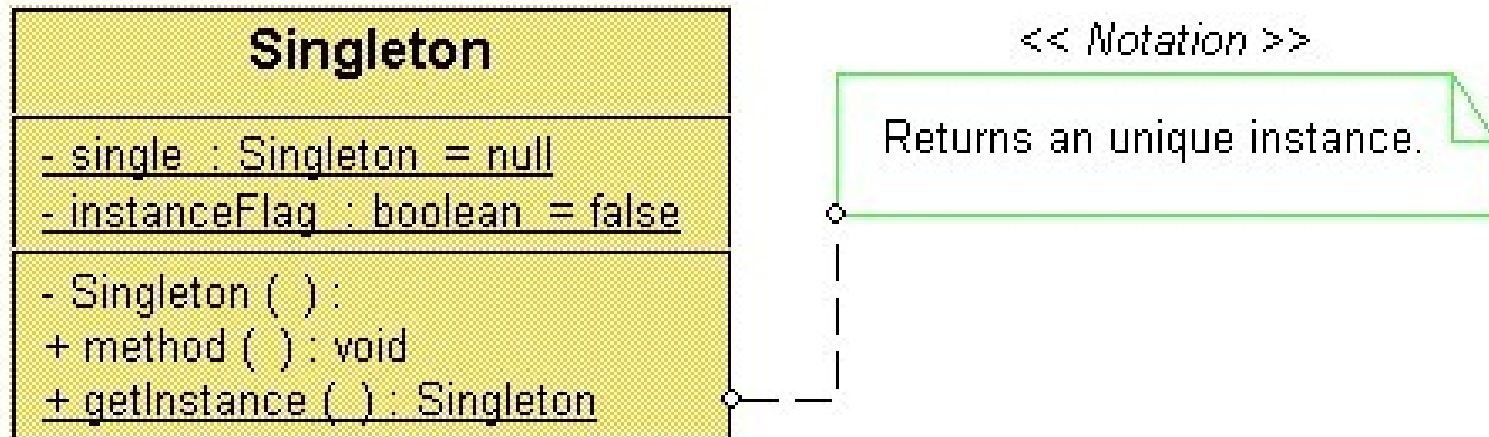
○ Estrutura



- single, instanceFlag e getInstance são estáticos.

PADRÃO SINGLETON

◦ Estrutura



- A única **instância** de Singleton é um **atributo privado**.
- O **método** de acesso à instância (*getInstance*) é um método **público**.

PADRÃO SINGLETON

Implementação (a mais simples)

```
public class Singleton {  
    private static Singleton instance;// instancia unica  
    private Singleton() {}  
    public static Singleton getInstance() {  
        // instanciação tardia  
        if (instance==null){  
            instance = new Singleton();  
        }  
        return instance;  
    }  
}
```

Lazy instantiation: o programa só inicia o recurso quando este for necessário

- controla o uso de recursos.
- economiza espaço de memória.



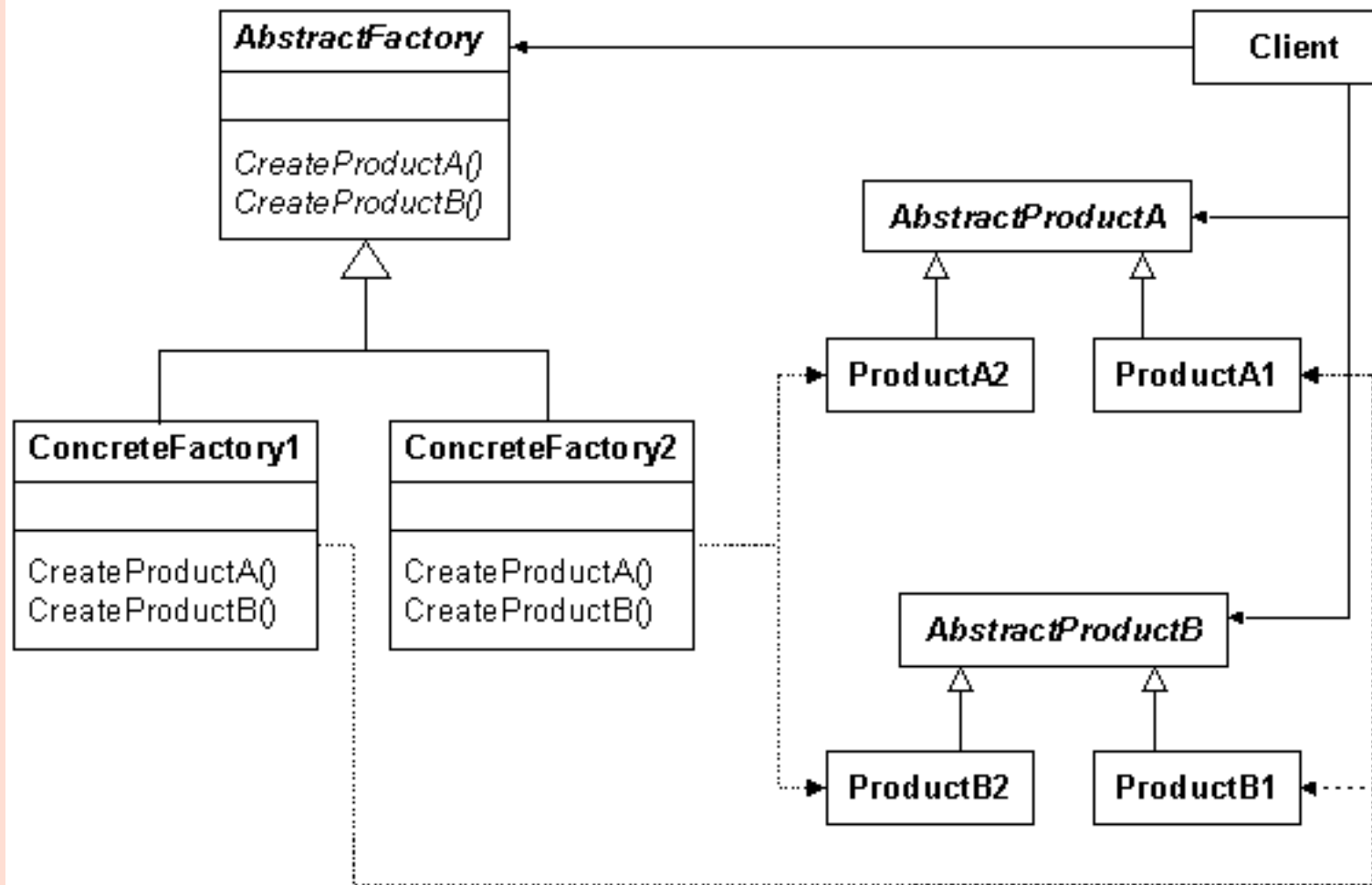
PADRÃO ABSTRACT FACTORY

PADRÃO ABSTRACT FACTORY

- Padrão de Criação
- **Objetivo:**
 - prover uma interface para criação de famílias de objetos sem especificar sua classe concreta.

PADRÃO ABSTRACT FACTORY

○ Estrutura



PADRÃO ABSTRACT FACTORY

○ Participantes:

- AbstractFactory:
 - Define quais produtos serão “produzidos” pela fábrica.
- ConcreteFactory
 - Implementa de fato a criação dos produtos.
- AbstractProduct
 - Define os contratos (métodos) que os produtos da fábrica deverão cumprir.
- ConcreteProduct
 - Implementa os contratos de cada produto.
- Client
 - Utiliza apenas as interfaces AbstractFactory e AbstractProduct.



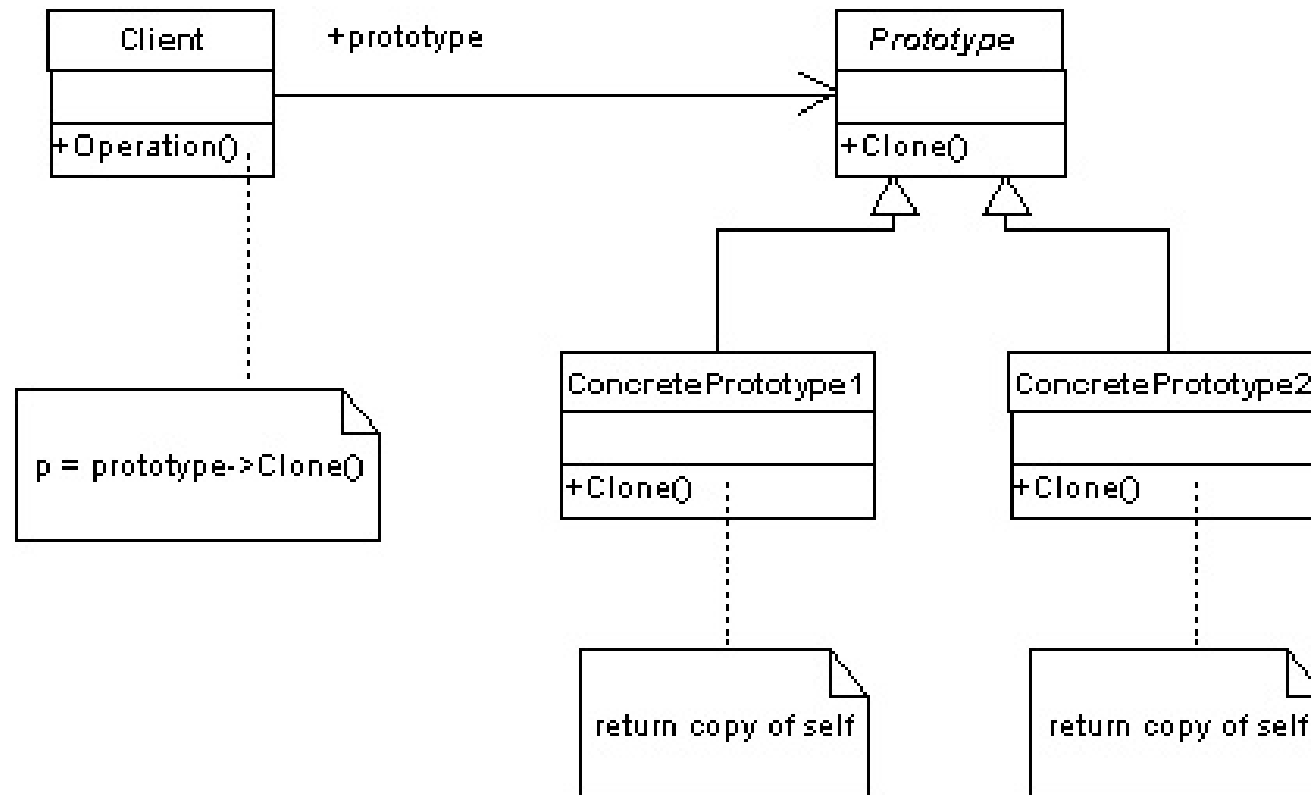
PADRÃO PROTOTYPE

PADRÃO PROTOTYPE

- Padrão de Criação.
- Objetivo:
 - Especificar os tipos de objetos a serem criados usando uma **instância como protótipo** e criar novos objetos ao copiar este protótipo.
- Motivação:
 - Criar um objeto novo aproveitando o estado previamente existente em outro objeto.

PADRÃO PROTOTYPE

○ Estrutura



PADRÃO PROTOTYPE

- Participantes

- Prototype:

- Só tem um método clone (Interface Cloneable).

- ConcretePrototype:

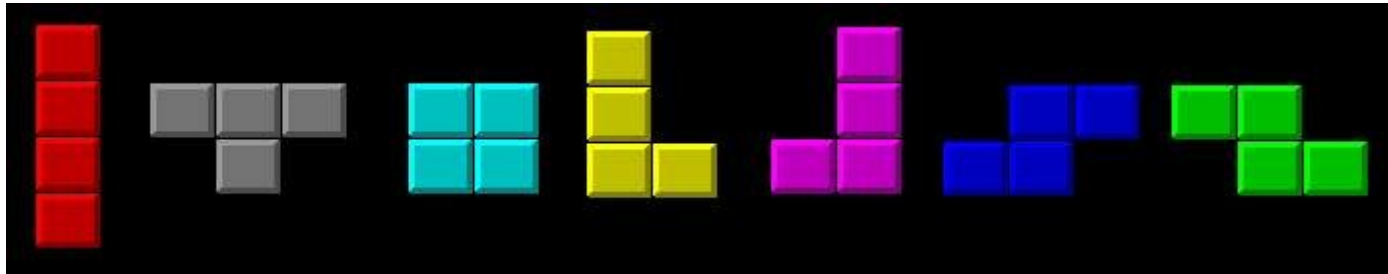
- Implementa o método que clona uma determinada instância, clonando internamente todos os seus componentes.

- Cliente:

- Cria um novo objeto solicitando ao protótipo que se clone.

PADRÃO PROTOTYPE

- Exemplo:
 - Em um jogo de tetris poderíamos criar instâncias protótipo para os sete tipos de peças:





PADRÕES ESTRUTURAIS

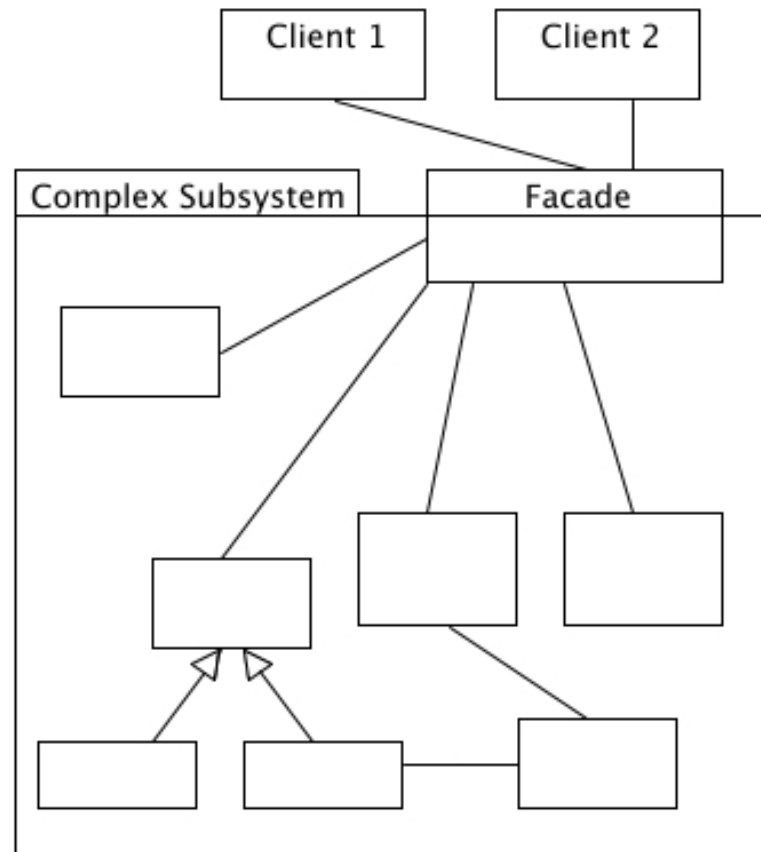
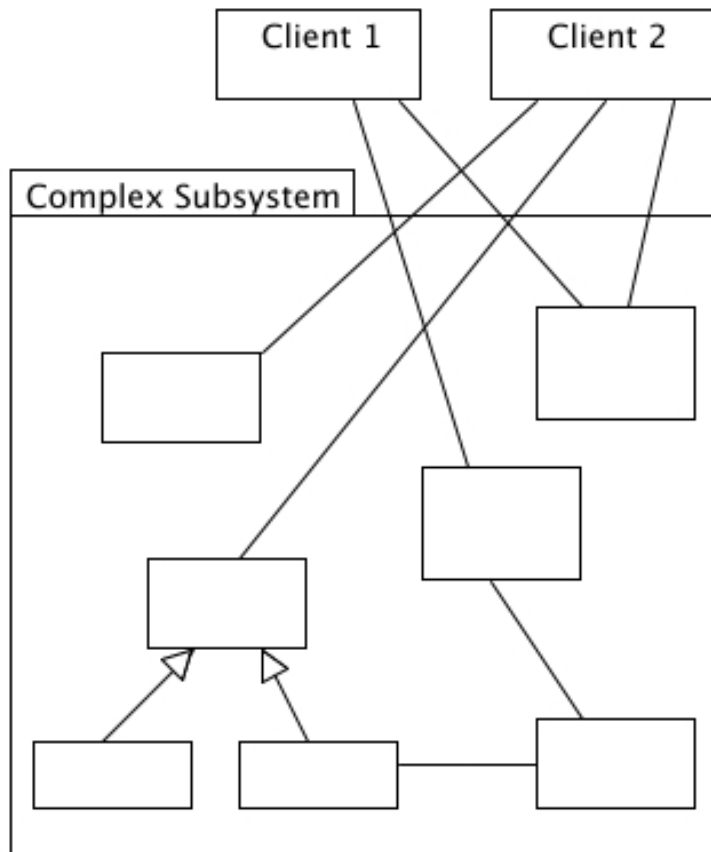
PADRÃO FAÇADE



PADRÃO FAÇADE

- Padrão Estrutural.
- **Objetivo:**
 - Oferecer uma **interface única** para um conjunto de interfaces de um subsistema.
 - Façade define uma interface de **mais alto nível** que torna mais fácil o uso do subsistema.
- **Motivação:**
 - Implementação com **baixo acoplamento** de interação entre subsistemas.

PADRÃO FAÇADE




PADRÃO FAÇADE

○ Participantes:

- Façade
 - Conhece quais as classes do subsistema são responsáveis pelo atendimento de uma solicitação.
 - Delega as solicitações do cliente a objetos apropriados do subsistema.
- Classes do Subsistema
 - Implementam a funcionalidade do subsistema.
 - Não são cientes da existência do Façade.
 - Encarregam-se do trabalho delegado pelo Façade.
- Cliente
 - O cliente não acessa as classes do subsistema.

Resumo Padrão Façade

- ▶ Quando você precisar **simplificar** e unificar um conjunto complexo de interfaces, use uma fachada.
 - ▶ Uma fachada **desconecta** um cliente de um subsistema complexo.
 - ▶ A implementação da fachada exige que **componhamos** a fachada com o seu subsistema e usemos **delegação** para executar o trabalho.
 - ▶ É possível implementar mais de uma fachada para o mesmo subsistema.
- 

CONSTRUINDO A FACHADA

```
public class HomeTheaterFacade  
{
```

```
    Amplifier amp;  
    Tuner tuner;  
    DvdPlayer dvd;  
    CdPlayer cd;  
    Projector projector;  
    TheaterLights lights;  
    Screen screen;  
    PopcornPopper popper;  
    // construtor ...
```

Todos estes são componentes
do subsistema

```
    public void watchMovie(String movie){  
        popper.on();  
        popper.pop();  
        lights.dim(10);  
        screen.down();  
        projector.on();  
  
        ...
```

delegação

```
    }  
}
```



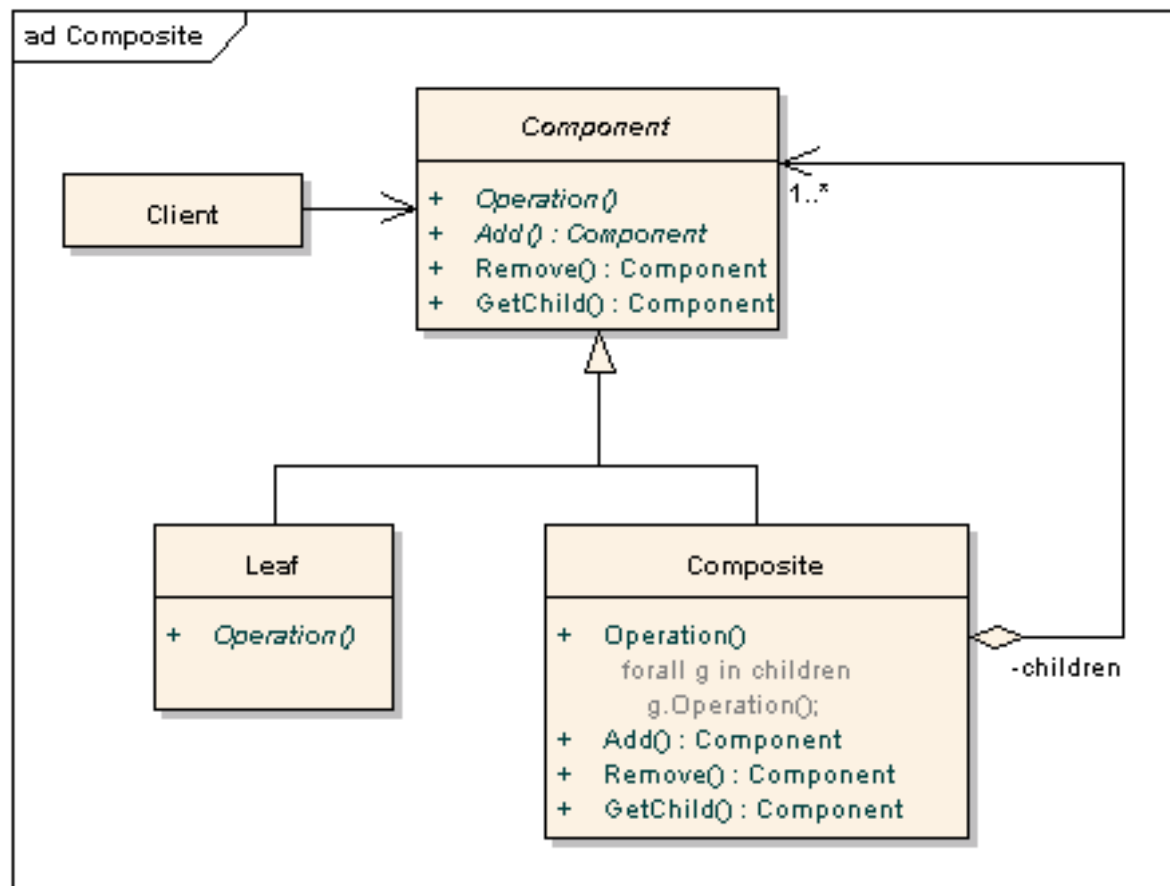
PADRÃO COMPOSITE

PADRÃO COMPOSITE

- Padrão Estrutural
- **Objetivo:**
 - Compor objetos em estruturas de árvore para representar hierarquias todo-parte.
 - Composite permite que clientes tratem objetos individuais e composições de objetos de maneira uniforme.

PADRÃO COMPOSITE

○ Estrutura:



PADRÃO COMPOSITE

○ Participantes:

● Component

- Declara a interface para os objetos na composição.
- Implementa, se necessário, o comportamento padrão.
- Declara uma interface para acessar e gerenciar os seus componentes filhos.

● Leaf

- Representa objetos folha na composição (sem filhos).
- Define comportamento para esses objetos.

● Composite

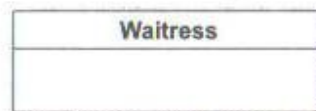
- Define comportamento para os componentes que têm filhos.
- Armazena referências a os componentes filho.

● Client

- Manipula os objetos na hierarquia utilizando instâncias Component.

A Garçonete usará a interface para acessar tanto os menus como os itens de menu

Usamos uma classe abstrata para fornecer uma implementação default para estes métodos



Esses métodos serverm para manipular os componentes

Tanto Menu como MenuItem substituem o método print()



Substitue os métodos que fazem sentido e usa a implementação padrão para aqueles que não lhe são relevantes, como add(), uma vez que não podemos acrescentar um componente a uma folha



Substitue os métodos que fazem sentido, como aqueles que lhe permitem acrescentar ou remover itens de menu (ou outros menus).

Métodos já existentes

PADRÃO DECORATOR

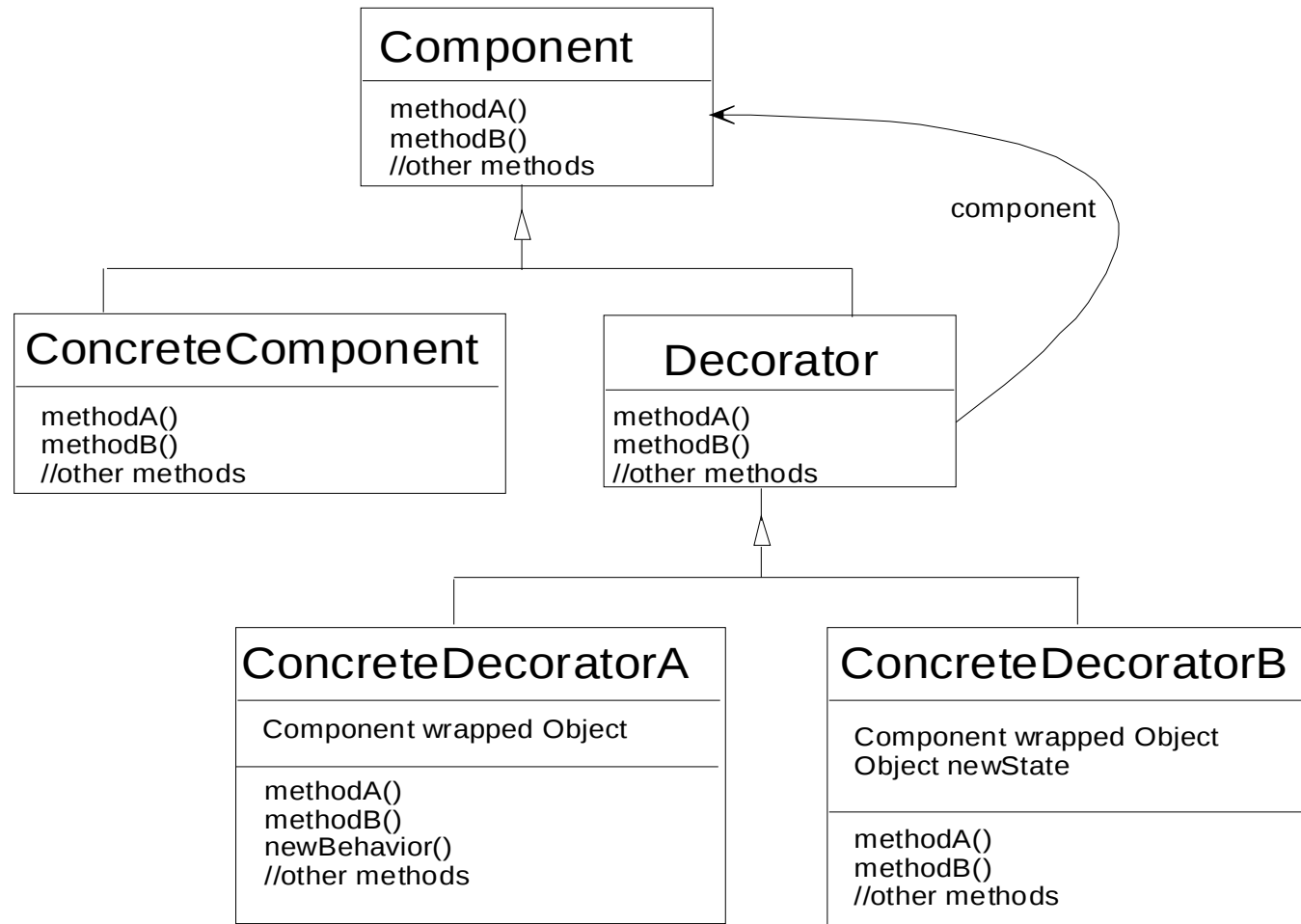


PADRÃO DECORATOR

- Padrão Estrutural.
- **Objetivo:**
 - Anexar **responsabilidades adicionais** a um objeto dinamicamente (“Enfeitar”).
 - Decorators oferecem uma alternativa para **estender a funcionalidade**.

PADRÃO DECORATOR

○ Estrutura



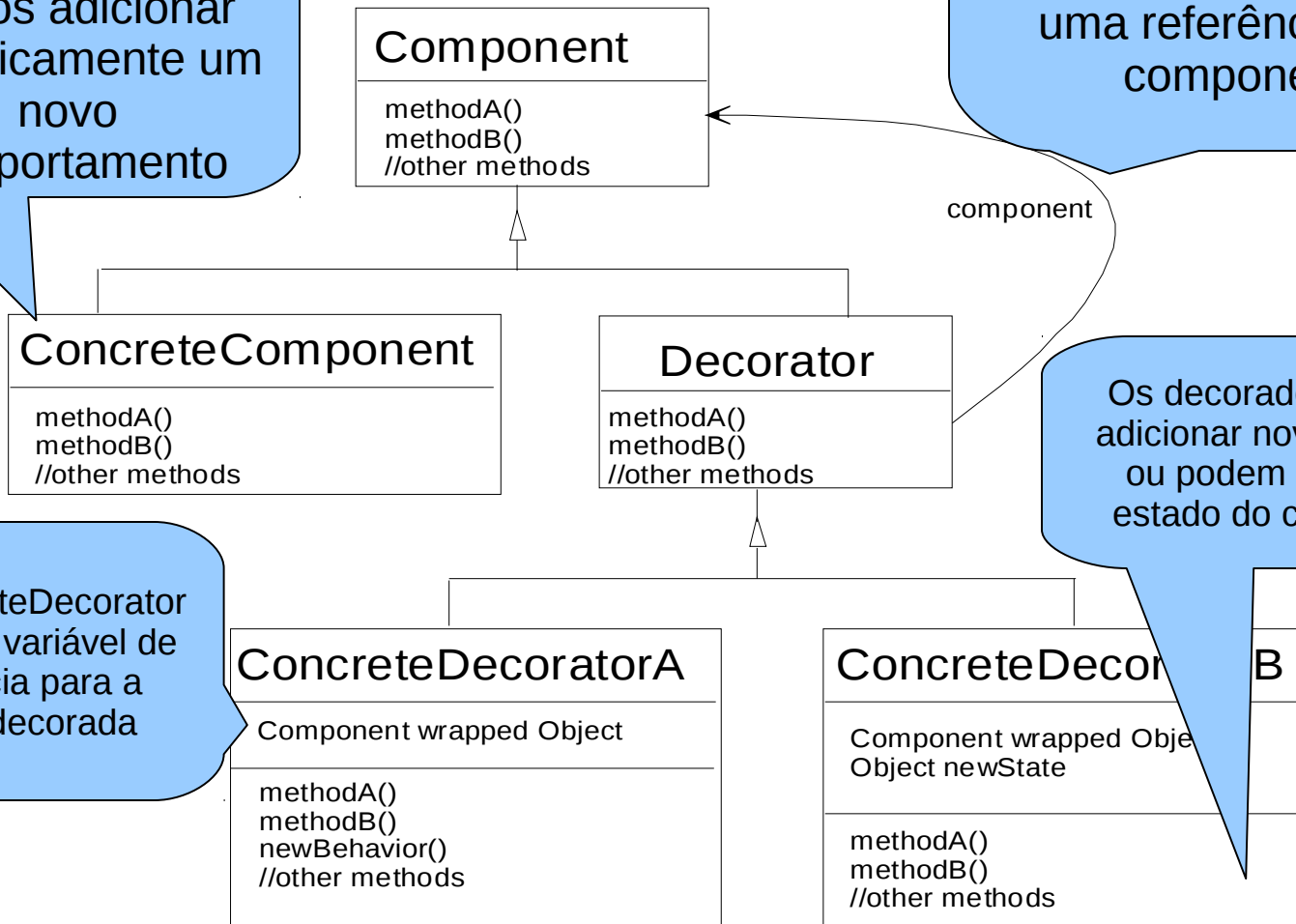
PADRÃO DE PROJETO

◦ Estrutura

Os decoradores implementam a mesma interface ou classe abstrata que o componente que irão decorar

Cada decorador TEM-UM componente, o que significa que o decorador tem uma variável de instância que contém uma referência a um componente

é o objeto ao qual vamos adicionar dinamicamente um novo comportamento



O ConcreteDecorator tem uma variável de instância para a coisa decorada

Os decoradores podem adicionar novos métodos ou podem estender o estado do componente



PADRÃO DECORATOR

○ Participantes

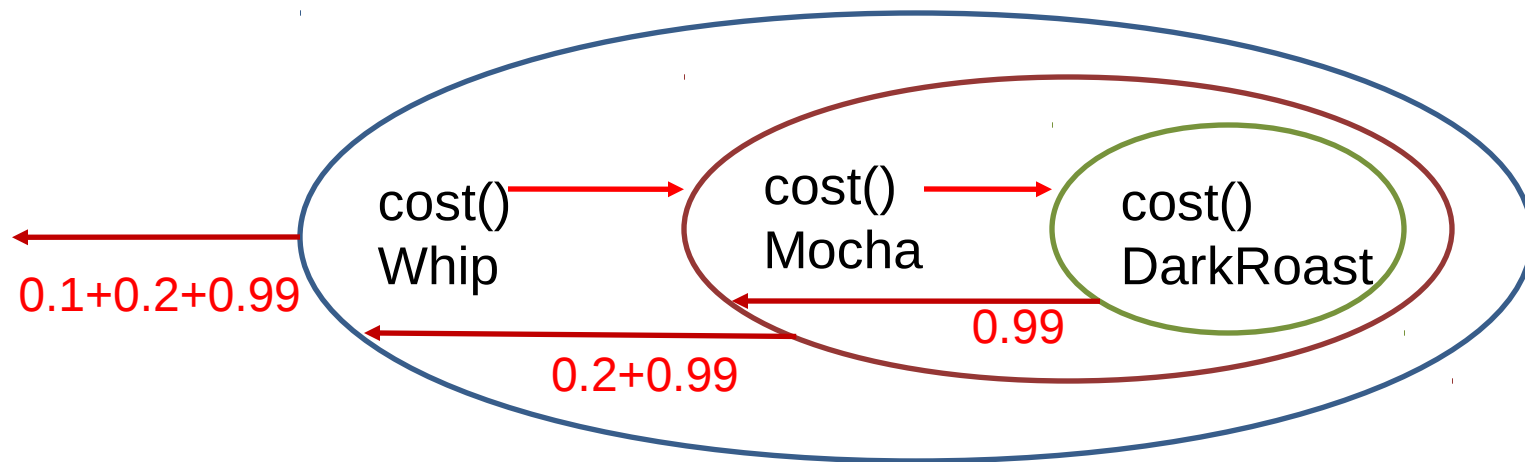
- Component:
 - A interface do objeto que será “enfeitado”.
- ConcreteComponent:
 - A implementação “básica” do componente.
- Decorator:
 - Mantém uma referência para um objeto Component.
 - Sobre-escreve o método (ou os métodos) do componente, chamando a funcionalidade básica e preparando o caminho para os decoradores concretos.
- ConcreteDecorator:
 - Enfeita o método adicionando algum comportamento ao método

PADRÃO DECORATOR

- Comentários adicionais
 - A herança é uma forma de extensão, mas não necessariamente a melhor maneira de obter flexibilidade em nossos projetos
 - Composição e delegação podem ser sempre usadas para adicionar novos comportamentos no tempo de execução.
 - Os decoradores mudam o comportamento de seus componentes adicionando novos recursos antes e/ou depois de chamadas de método para o componente.
 - Os decoradores podem resultar em muitos objetos pequenos em nosso design e o uso exagerado deve ser evitado

Aplicando o Padrão Decorator ao exemplo Starbuzz

O cliente quer Cafe (Dark Roast) com Mocha (Mocha) e Creme(Whip)



Criamos um Calculando o custo no envoltório `o` :



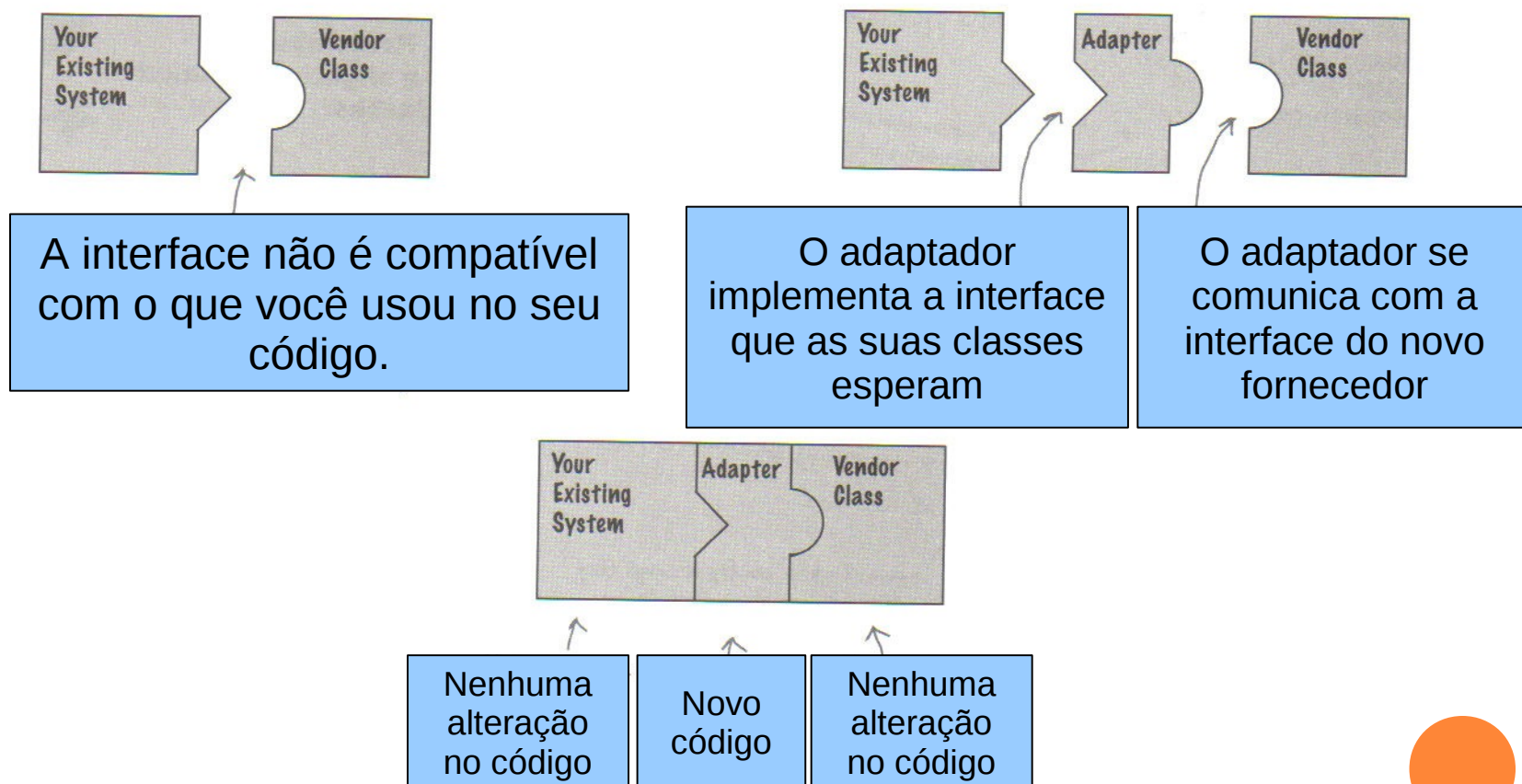
PADRÃO ADAPTER



PADRÃO ADAPTER

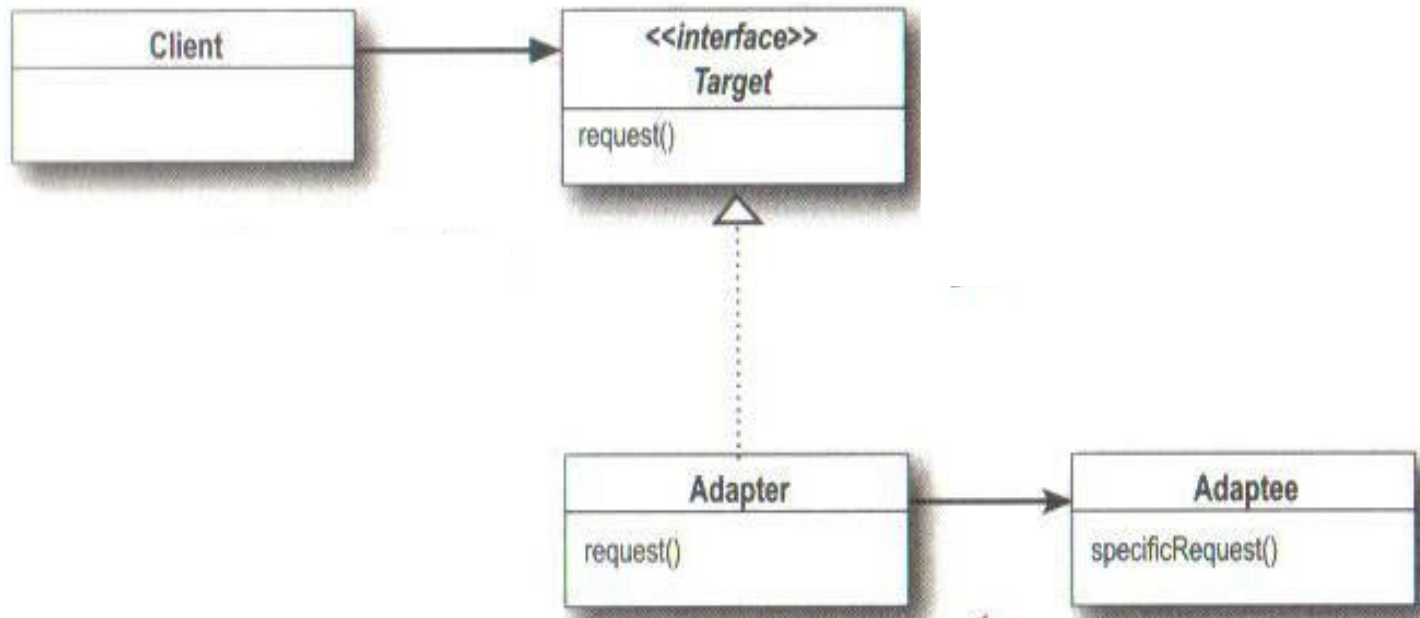
- Padrão Estrutural
- **Objetivo:**
 - Converter a interface de uma classe para outra interface que o cliente espera encontrar.
 - Permite que classes com interfaces incompatíveis trabalhem juntas

Adaptadores orientados a objetos



PADRÃO ADAPTER

○ Estrutura



PADRÃO ADAPTER

○ Participantes:

- Target
 - Define quais operações serão acessadas pelo cliente.
- Adapter
 - Adapta uma classe determinada ao Target, oferecendo as operações acessadas pelo cliente
- Adaptee
 - É o módulo pré-existente que oferece o serviço que pretendemos adaptar
- Client
 - Utiliza apenas a interface Target para acessar os serviços do provedor.
- Cadeia de chamadas: Client -> Target -> ObjectAdapter -> Adaptee.



PADRÕES COMPORTAMENTAIS



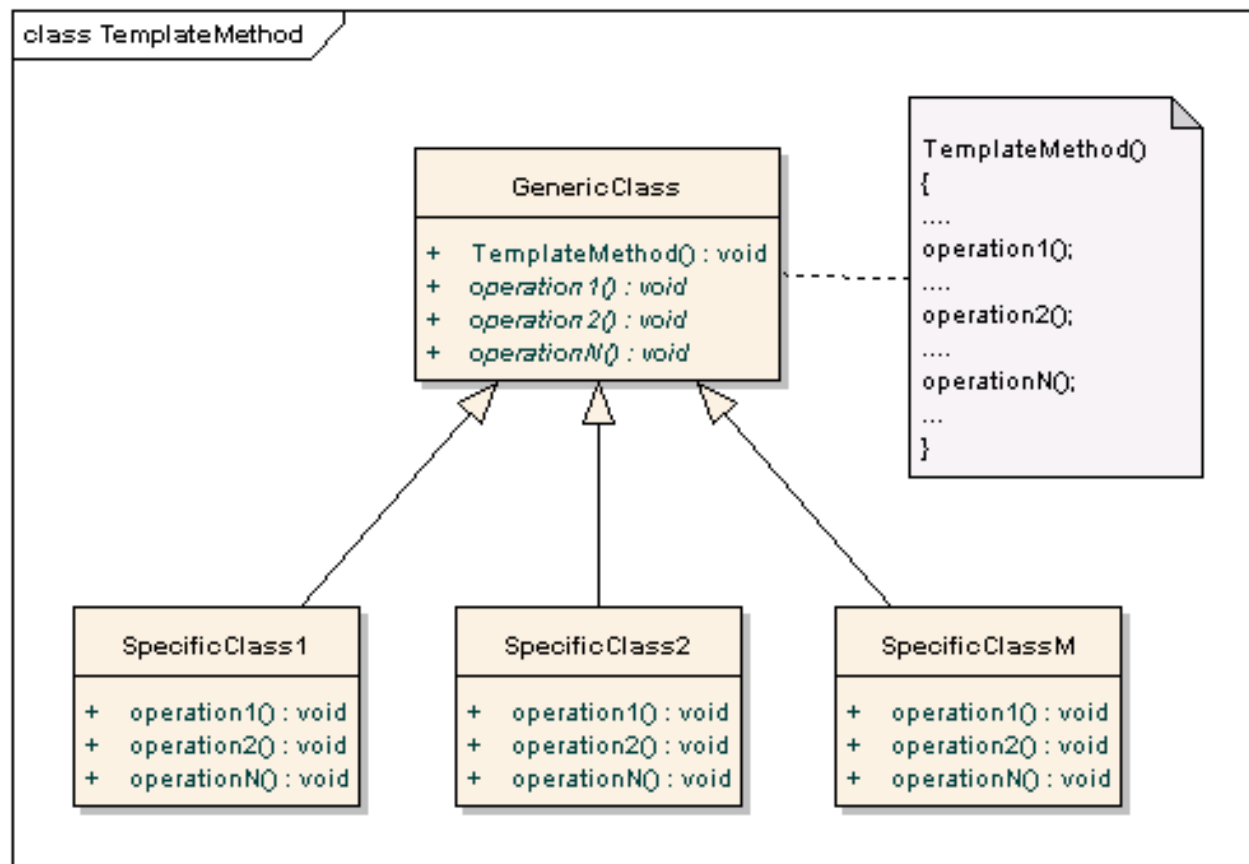
PADRÃO TEMPLATE METHOD

PADRÃO TEMPLATE METHOD

- Padrão Comportamental.
- **Objetivo:**
 - Definir o **esqueleto de um algoritmo** dentro de um método, **transferindo alguns de seus passos** para as subclasses. O Template Method permite que as subclasses redefinam certos passos de um algoritmo sem alterar a estrutura do próprio algoritmo.

PADRÃO TEMPLATE METHOD

○ Estrutura:



PADRÃO TEMPLATE METHOD

○ Participantes:

- GenericClass: **Classe abstrata**.
 - Define operações primitivas abstratas a serem concretizadas.
 - Implementa (pelo menos) um **método “template”** que define o esqueleto do algoritmo. Este método deve ser declarado como **final** (em Java) para **evitar que as subclasses o modifiquem**.
- SpecificClass
 - Implementa as operações primitivas para executarem os **passos específicos do algoritmo**.
 - Não sobre-escrevem o método template.

PADRÃO TEMPLATE METHOD: RESUMO

- Don't call us we'll call you
- O padrão Template define o conjunto de passos de um algoritmo
- Subclasses não podem mudar o algoritmo (final)
- Facilita o reuso de código

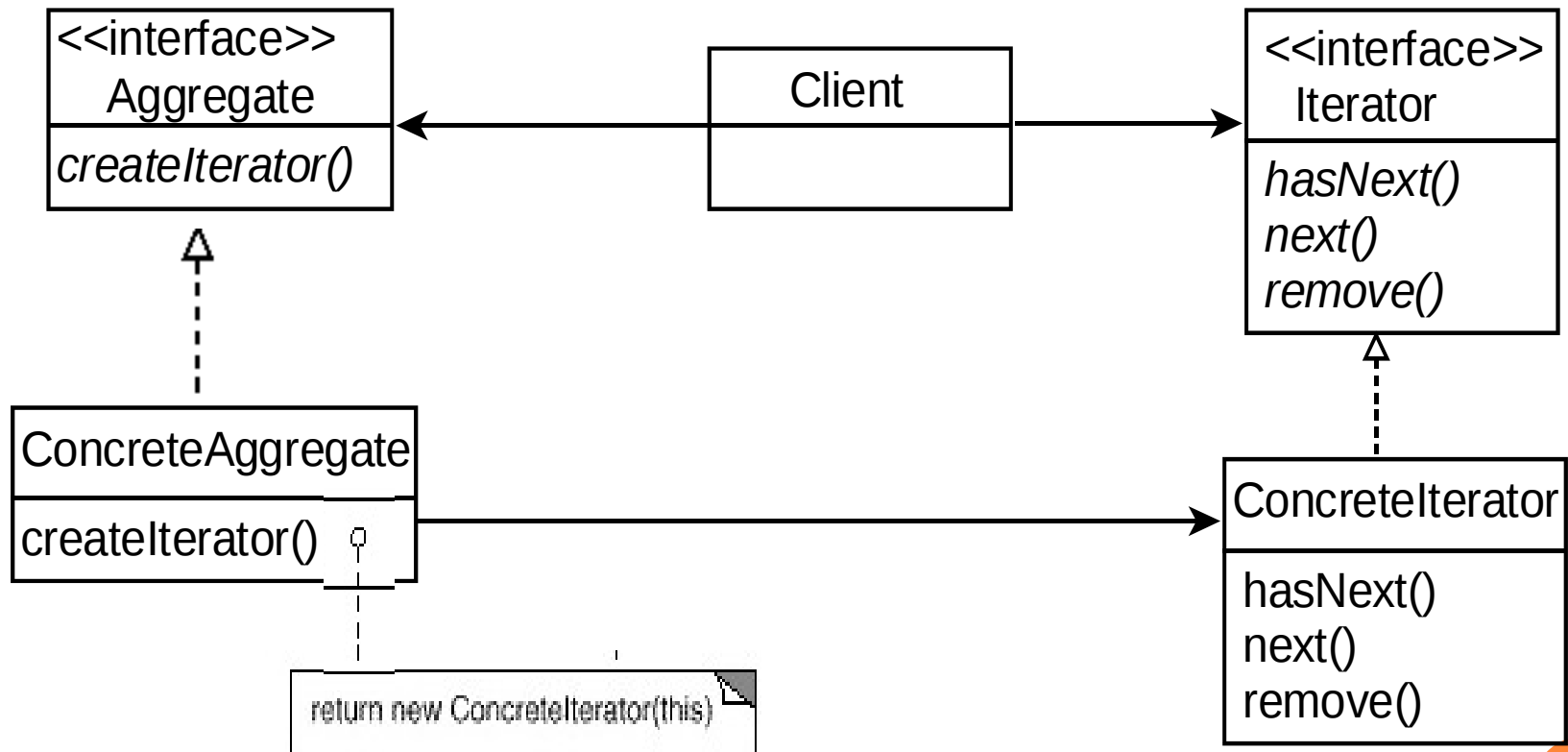


PADRÃO ITERATOR

PADRÃO ITERATOR – Objetivo

- Padrão Comportamental.
- Fornece um meio de **acessar sequencialmente um objeto agregado** (por ex, uma coleção) sem expor a sua representação.

Padrão Iterator - Estrutura



Padrão Iterator

○ Participantes:

- Iterator:
 - Define uma interface para acessar e percorrer elementos
- ConcreteIterator
 - Implementa a interface de Iterador
 - Mantém o controle da posição corrente no percurso do agregado.
- Aggregate
 - Define uma interface para a criação de um objeto Iterator
- ConcreteAggregate
 - Implementa a interface de criação do Iterator para retornar uma instância do ConcreteIterator apropriado

PADRÃO OBSERVER



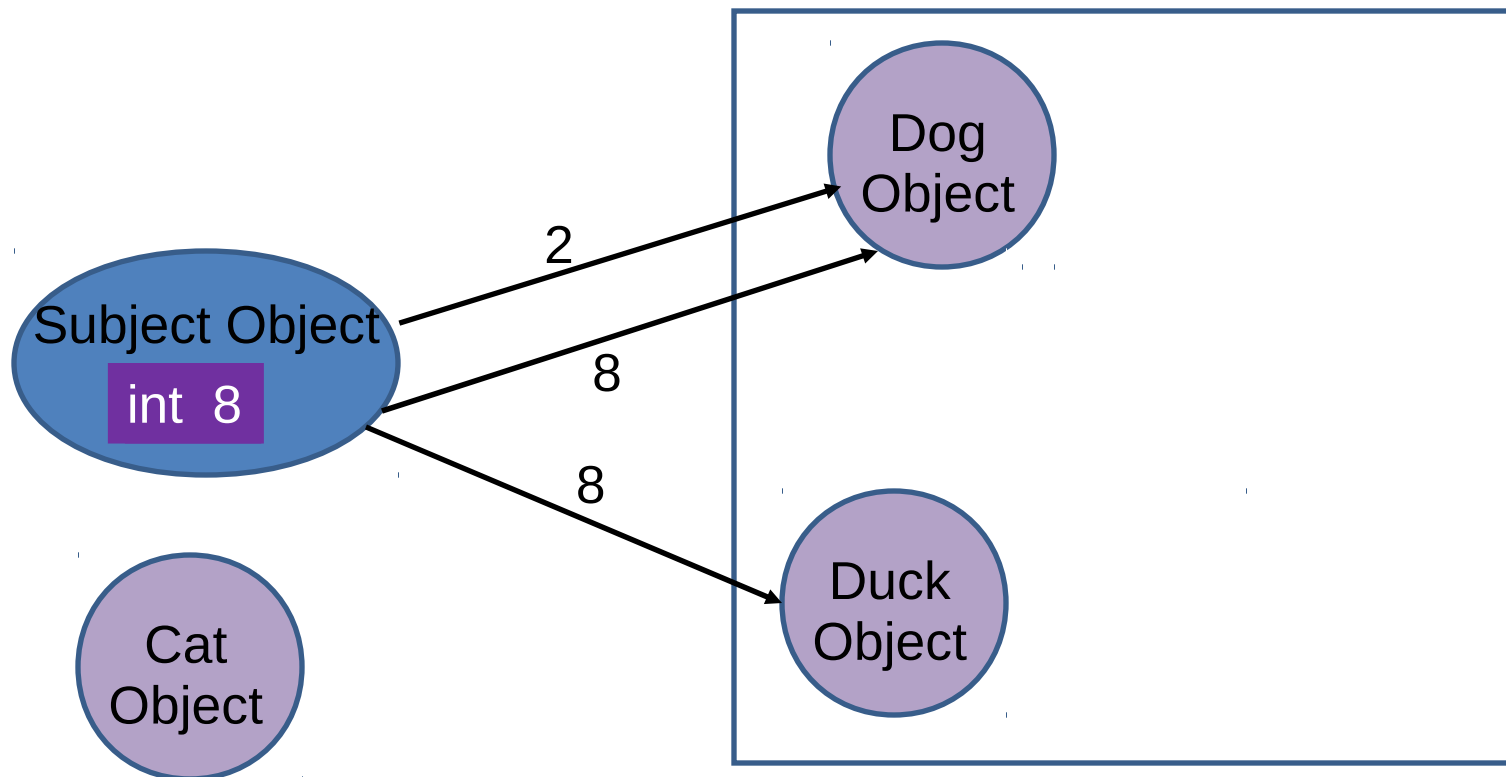
PADRÃO OBSERVER

- Padrão Comportamental.
- Objetivo:
 - Definir uma dependência um – para – muitos entre objetos, para que quando um objeto mudar de estado, todos os seus dependentes sejam notificados e atualizados automaticamente.
- Motivação:
 - Quando temos um objeto (ou conjunto de objetos) cujo estado ou comportamento depende do estado de outro objeto.
 - O objeto “observador” precisa ser informado das mudanças no objeto “observado”.
 - O objeto “observador” também pode alterar o “observado”.

Padrão Observer

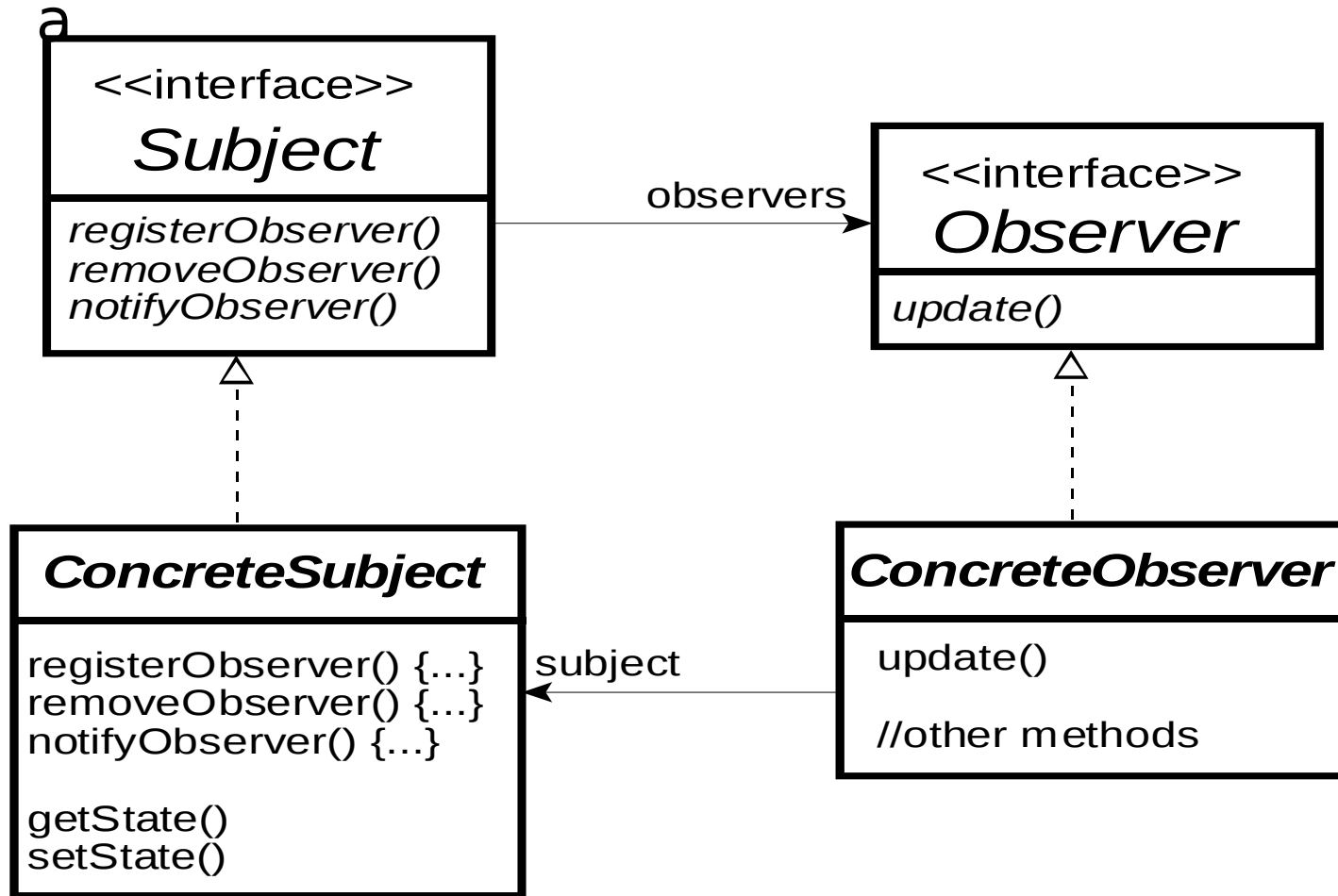
Subject Broadcasts

Observers



PADRÃO OBSERVER

○ Estrutur



PADRÃO OBSERVER

○ Participantes

● Subject:

- Pode possuir uma lista dos seus observadores, mas não conhece o tipo ou o objetivo de cada um deles.
- Possui interface para adicionar e remover “assinantes” ou observadores.

● ConcreteSubject:

- Notifica seus observadores quando o seu estado muda.

● Observer:

- Define uma interface para ser notificado de mudanças.

● ConcreteObserver:

- Mantém uma referência ao objeto observado.
- Guarda estado do observado para identificar as mudanças.
- Implementa a resposta às mudanças do observado.

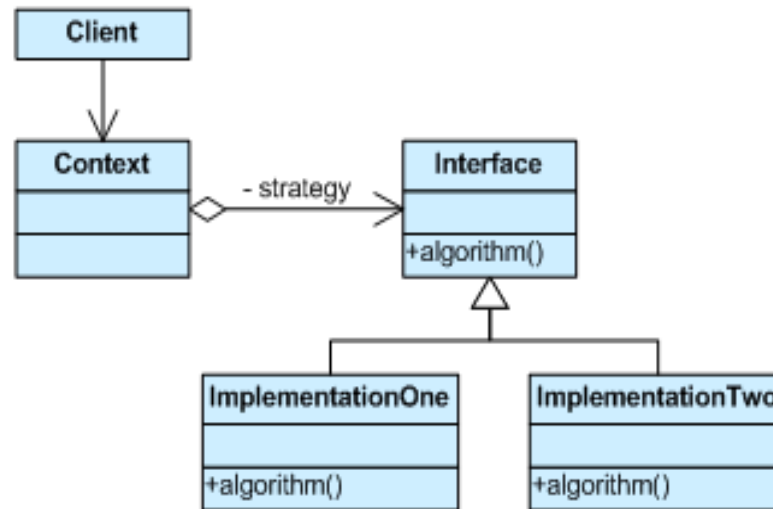
PADRÃO STRATEGY



Padrão Strategy – Objetivo

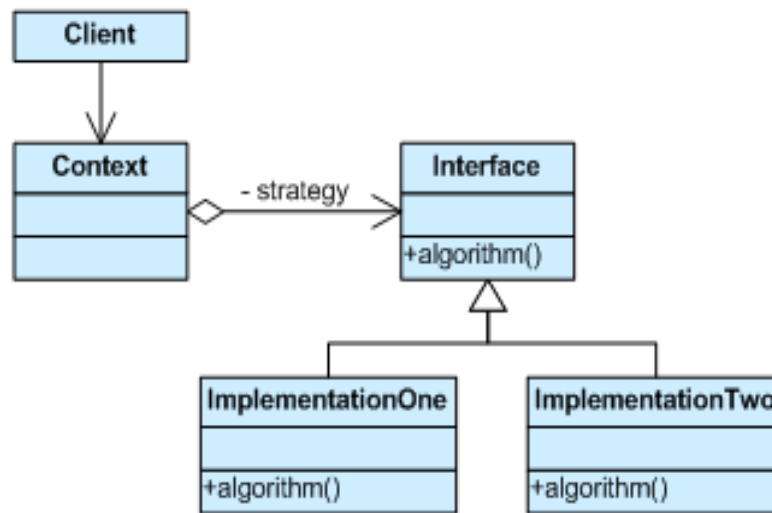
- Define e encapsula uma **família de algoritmos** e os torna intercambiáveis.
- Permite que o algoritmo varie independentemente das suas classes clientes.
- Captura a abstração em uma interface
 - detalhes de implementação são deixados para as classes derivadas.

Padrão Strategy - Estrutura



- A classe Context:
 - é composta de uma estratégia (Interface Strategy)
 - representa uma funcionalidade da classe Cliente que requer comportamentos variantes.
 - Por ex, uma funcionalidade de busca.

Padrão Strategy - Estrutura



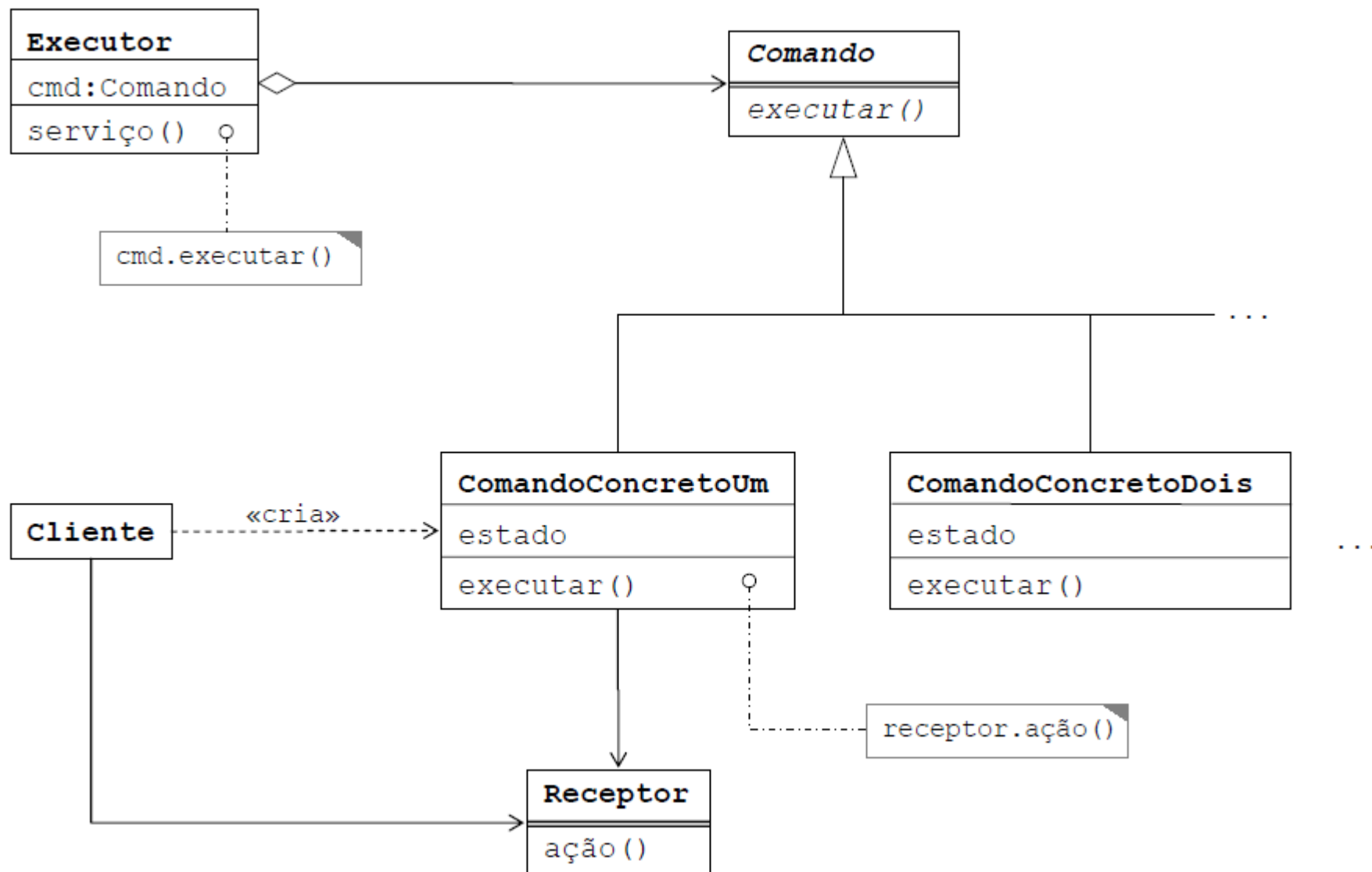
- Como a estratégia (Strategy) é implementada como uma interface, é possível trocar uma implementação concreta sem afetar a classe Context.

PADRÃO COMMAND

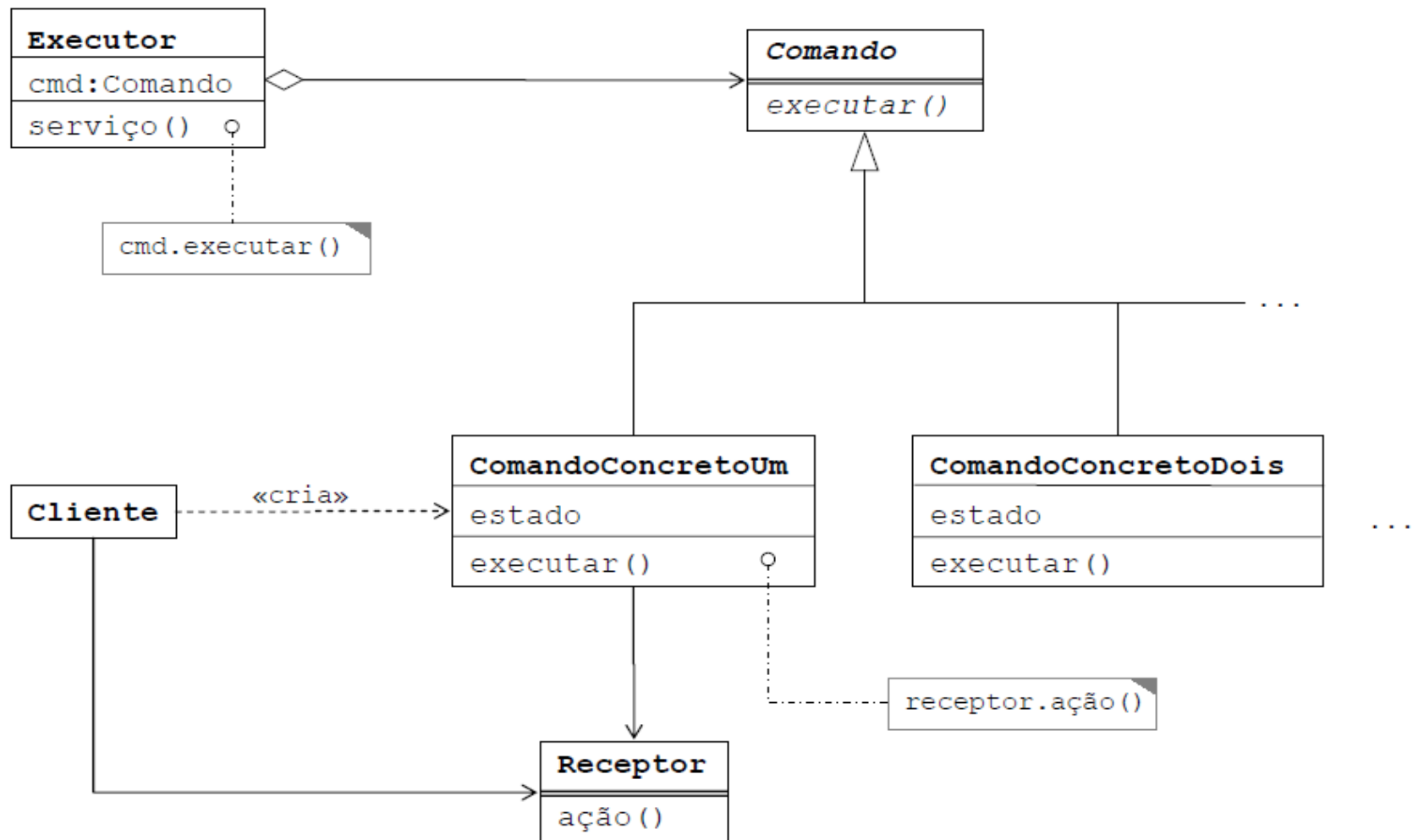


Padrão Command – Objetivo

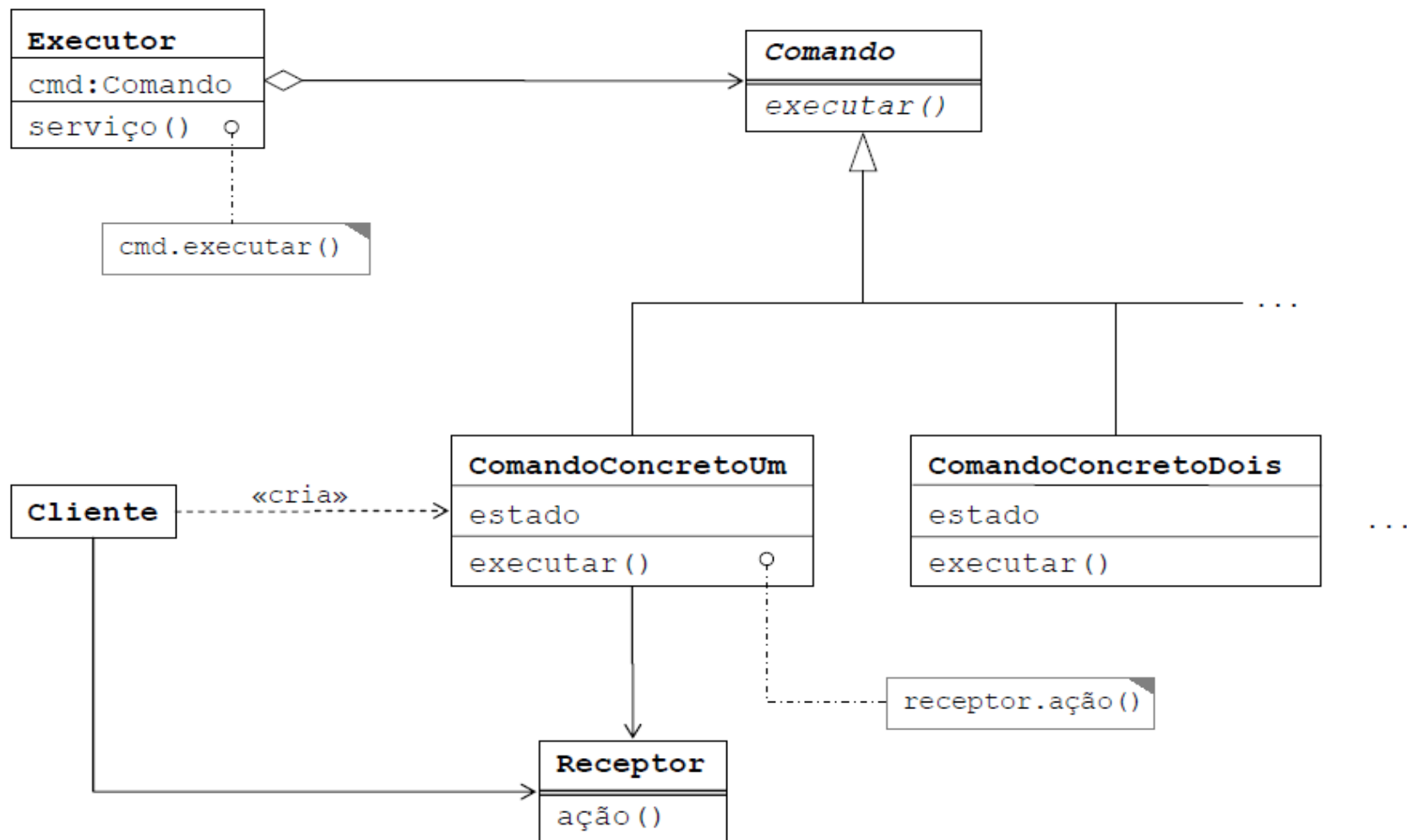
- Encapsular uma solicitação como um objeto, permitindo:
 - parametrizar clientes com diferentes solicitações;
 - enfileirar ou fazer registros (logs) de solicitações;
 - suportar operações que podem ser desfeitas.
- Criar uma abstração para invocar uma tarefa a ser executada.



- A interface Comando contém um método executar() que simplesmente chama ação definida no receptor.



- As subclasses concretas de Comando
 - armazena o receptor como uma variável de instância.
 - implementa o método `executar()` para realizar a solicitação.



- O Executor trata os seus objetos command como “caixas pretas”, simplesmente invocando o método `execute()` a cada solicitação de um “serviço”.