

Design Patterns Estruturais

Facade, Adapter, Decorator

2022

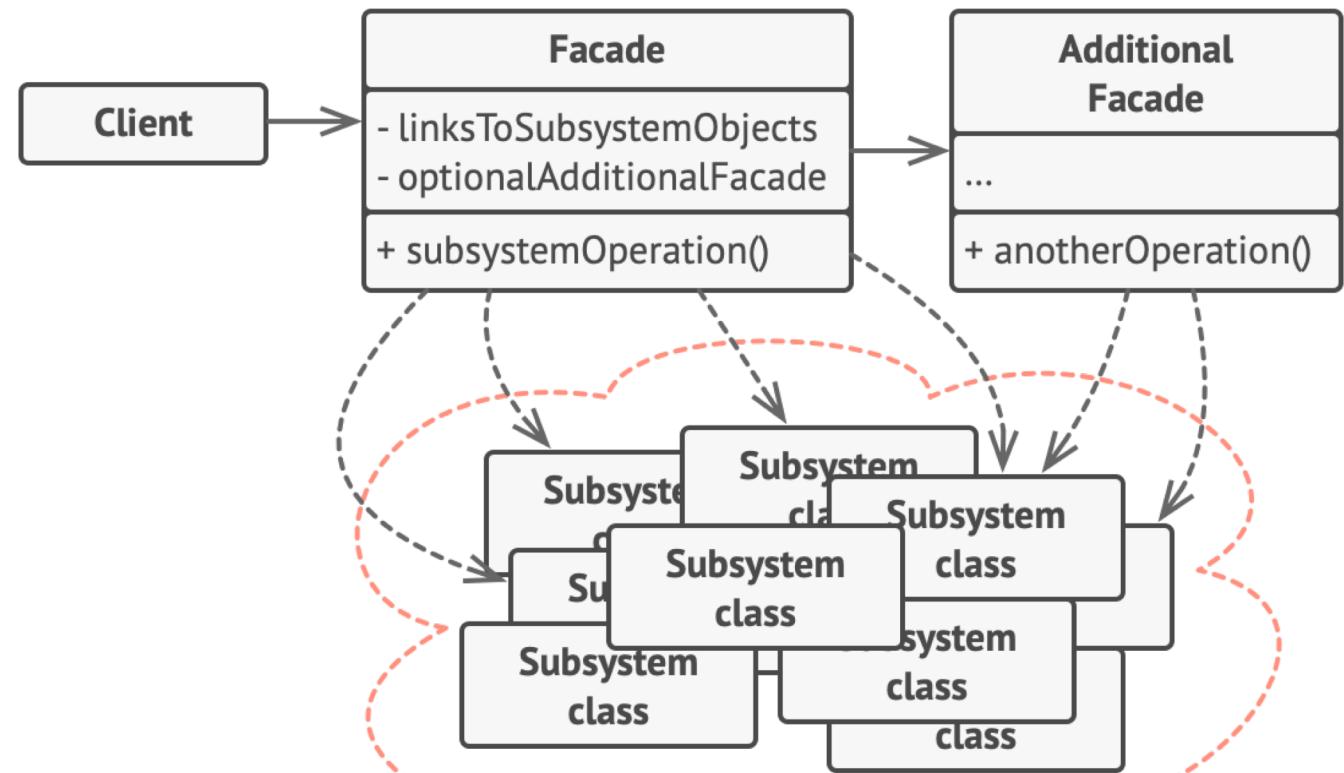
Prof. Sergio Bonato

Propósito e Estrutura

Material adaptado do livro: Mergulho nos Padrões de Projetos, de Alexander Shvets. A maior parte do conteúdo do livro está disponível em <https://refactoring.guru/pt-br/design-patterns>, onde também há o link para compra.

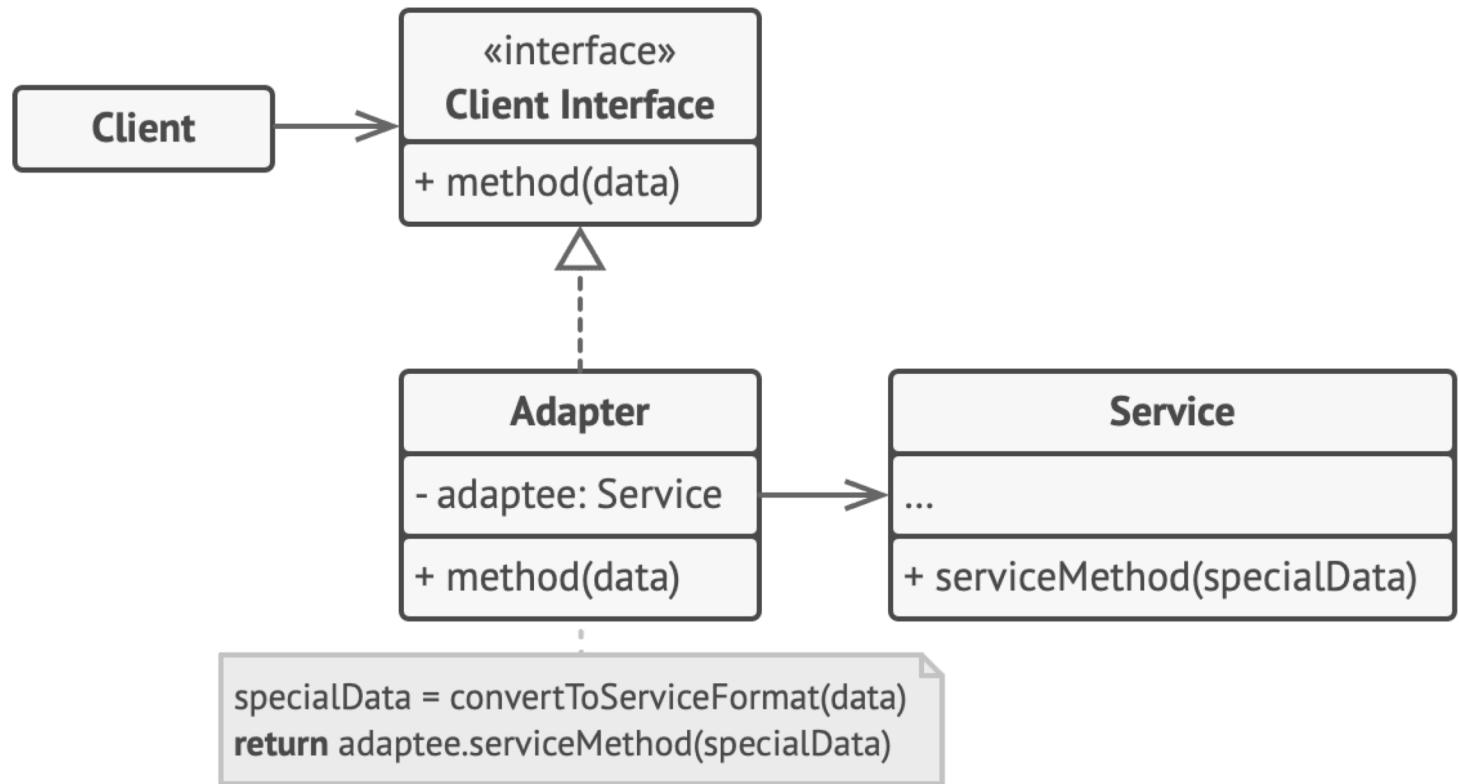
Facade

O Facade é um padrão de projeto estrutural que fornece uma interface simplificada para uma biblioteca, um framework, ou qualquer conjunto complexo de classes.



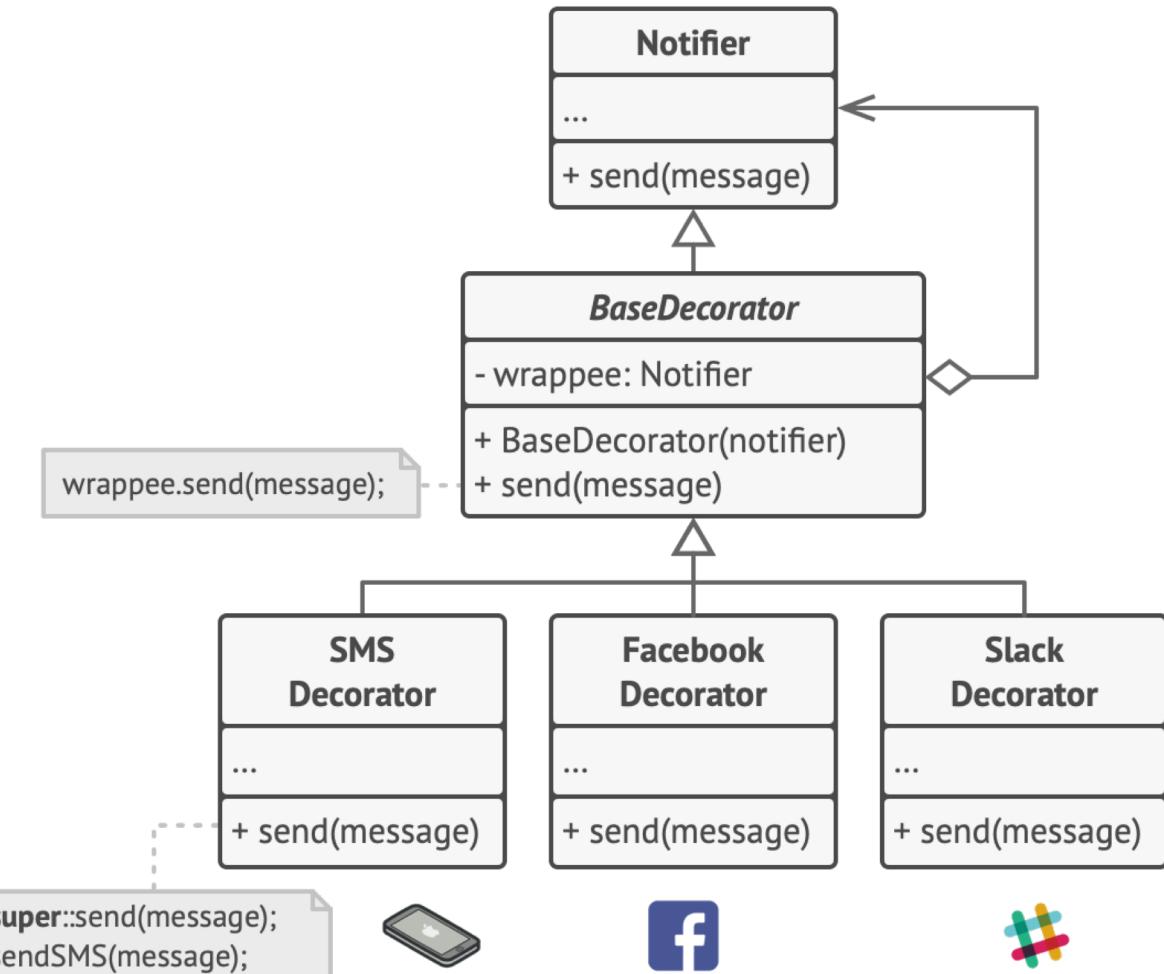
Adapter

O Adapter é um padrão de projeto estrutural que permite objetos com interfaces incompatíveis colaborarem entre si.



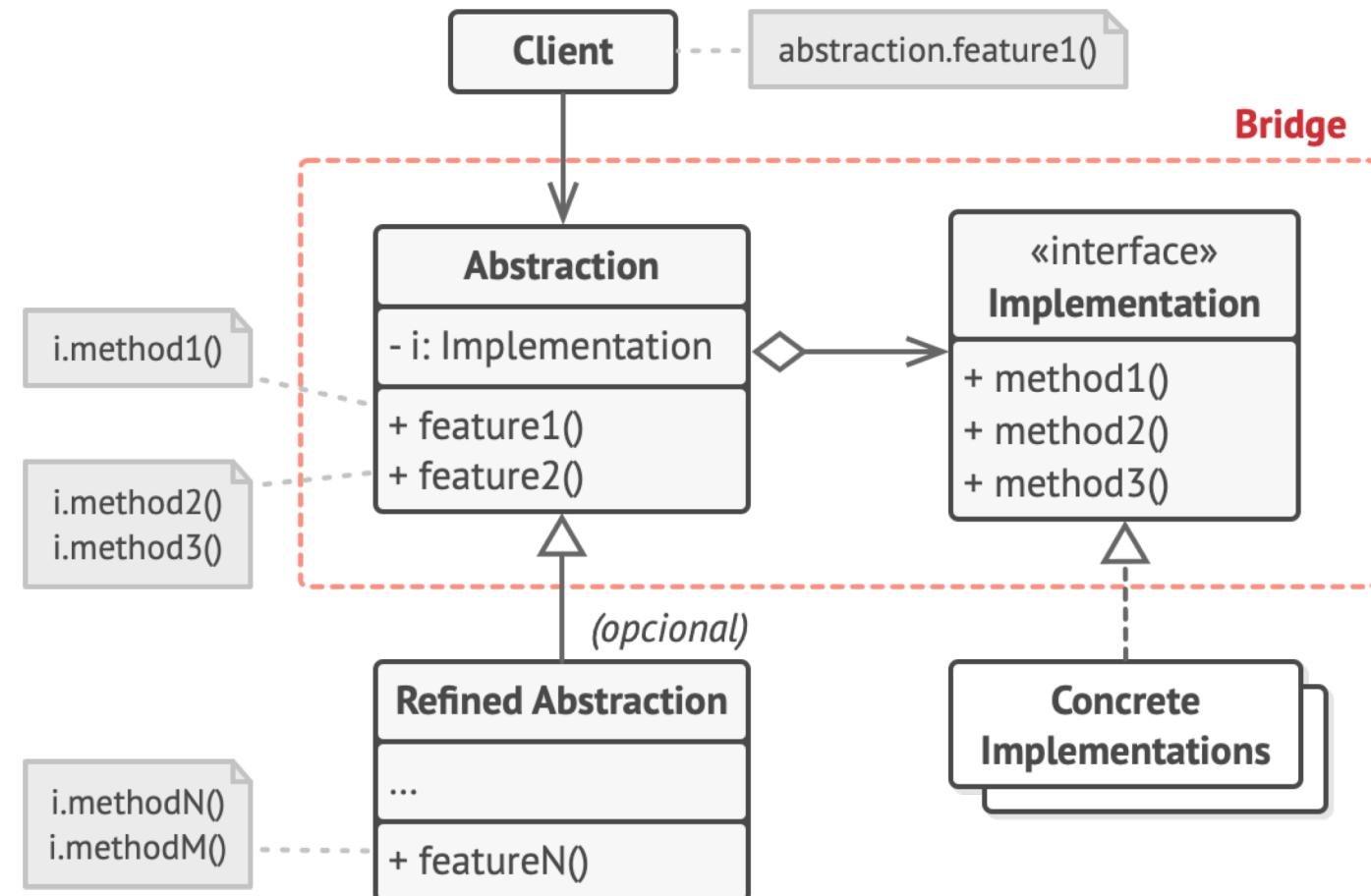
Decorator

O Decorator é um padrão de projeto estrutural que permite que você acople novos comportamentos para objetos ao colocá-los dentro de invólucros de objetos que contém os comportamentos.



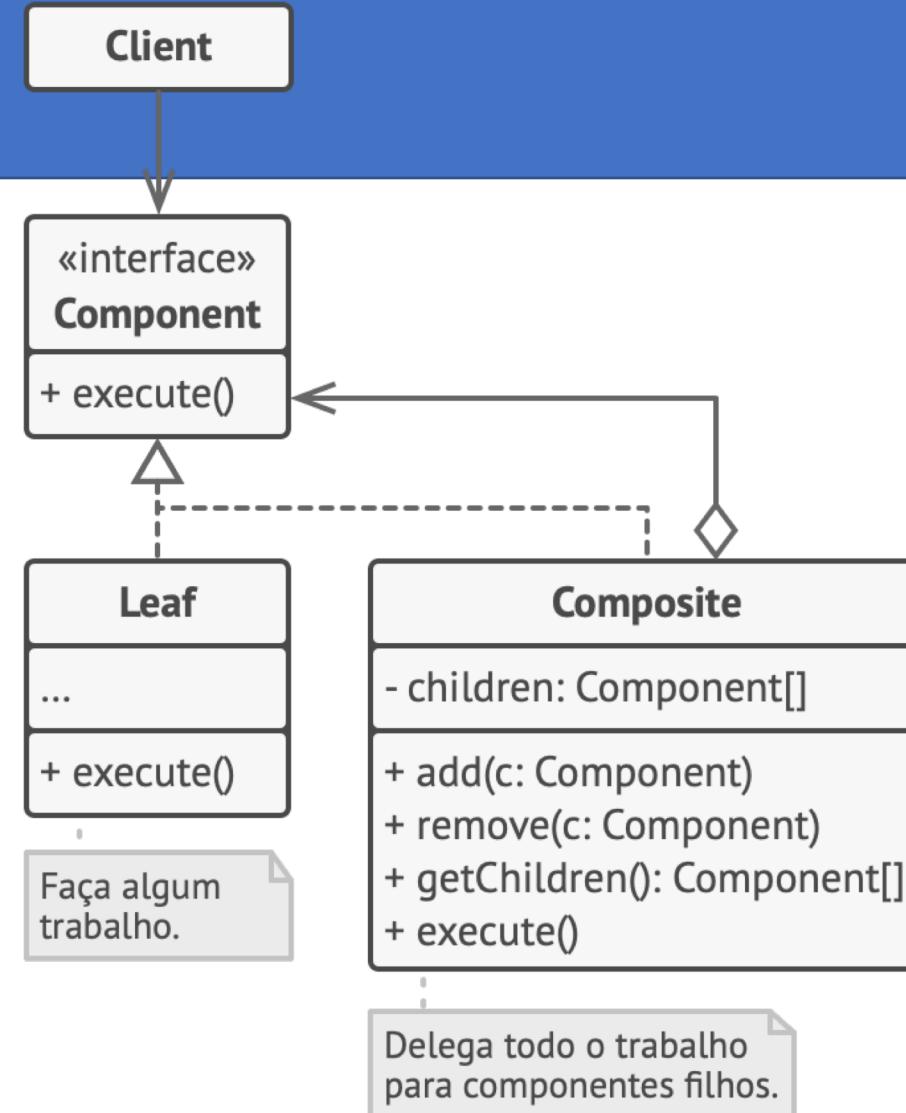
Bridge

O Bridge é um padrão de projeto estrutural que permite que você divida uma classe grande ou um conjunto de classes intimamente ligadas em duas hierarquias separadas—abstração e implementação—que podem ser desenvolvidas independentemente umas das outras.



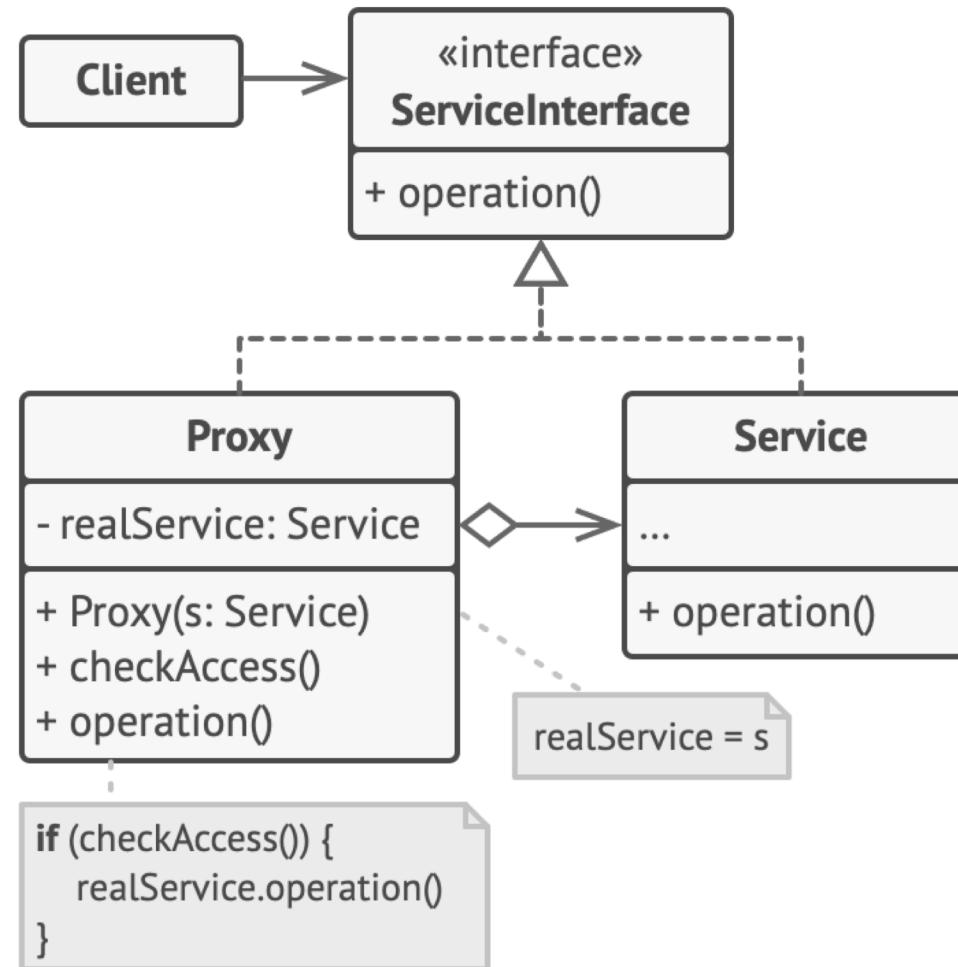
Composite

O Composite é um padrão de projeto estrutural que permite que você componha objetos em estruturas de árvores e então trabalhe com essas estruturas como se elas fossem objetos individuais.



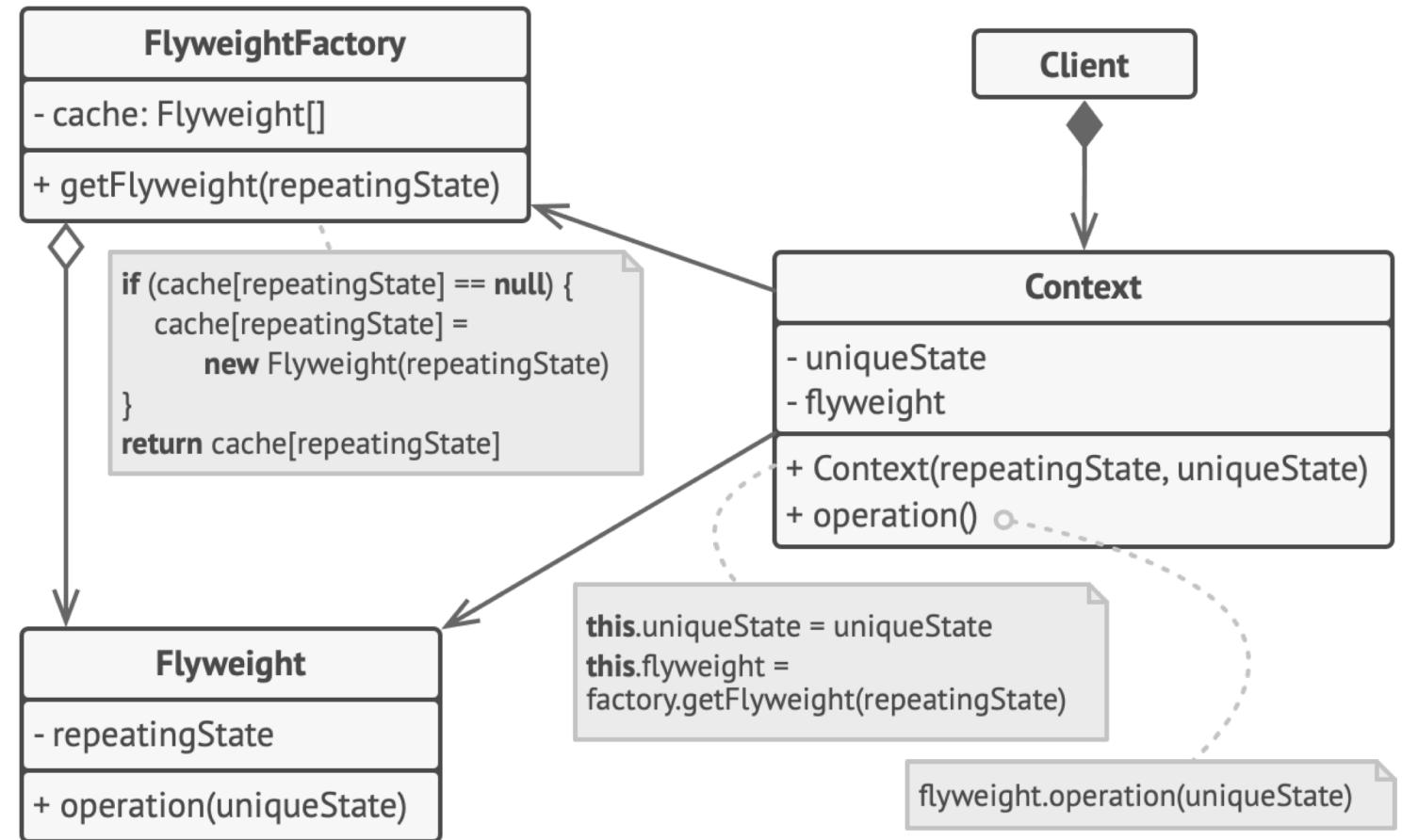
Proxy

O Proxy é um padrão de projeto estrutural que permite que você forneça um substituto ou um espaço reservado para outro objeto. Um proxy controla o acesso ao objeto original, permitindo que você faça algo ou antes ou depois do pedido chegar ao objeto original.



Flyweight

O Flyweight é um padrão de projeto estrutural que permite a você colocar mais objetos na quantidade de RAM disponível ao compartilhar partes comuns de estado entre os múltiplos objetos ao invés de manter todos os dados em cada objeto.



Detalhes de Implementação

- Facade
- Adapter
- Decorator

Os exemplos desta seção foram retirados do livro Java Design Patterns, de Rohit Josh, mas o código foi portado de Java para TypeScript. O livro do Josh pode ser obtido em

<https://www.javacodegeeks.com/2015/09/java-design-patterns.html>

Facade

Facade: Motivação

- Sua empresa é uma empresa baseada em produtos e lançou um produto no mercado, chamado Schedule Server. É um tipo de servidor em si e é usado para gerenciar tarefas. Os trabalhos poderiam ser qualquer tipo de trabalho, como enviar uma lista de e-mails, sms, ler ou gravar arquivos de um destino, ou simplesmente transferir arquivos de uma fonte para o destino. O produto é usado pelos desenvolvedores para gerenciar esse tipo de trabalho e serem capazes de se concentrar mais em seu objetivo de negócios. O servidor executa cada trabalho no horário especificado e também gerencia todos os problemas de subjacentes, como o problema de concorrência e o de segurança. Como desenvolvedor, basta codificar apenas os requisitos de negócios relevantes e uma boa quantidade de chamadas de API é fornecida para agendar um trabalho de acordo com as necessidades deles.
- Tudo estava indo bem, até que os clientes começaram a reclamar sobre iniciar e parar o processo do servidor. Eles disseram que, embora o servidor esteja funcionando muito bem, os processos de inicialização e encerramento são muito complexos e eles querem uma maneira fácil de fazer isso. O servidor expôs uma interface complexa para os clientes que parece um pouco complicado para eles.
- Precisamos fornecer uma maneira fácil de iniciar e parar o servidor. Sem ter que refazer a codificação a partir do zero. Precisamos de uma maneira de resolver esse problema e tornar a interface fácil de acessar.

Facade: Motivação

Código para iniciar o servidor antes do Facade

start_server.ts

```
import { ScheduleServer } from "../ScheduleServer";

let scheduleServer: ScheduleServer = new ScheduleServer();

scheduleServer.startBooting();
scheduleServer.readSystemConfigFile();
scheduleServer.init();
scheduleServer.initializeContext();
scheduleServer.initializeListeners();
scheduleServer.createSystemObjects();

console.log('Start working...');
console.log('After work done...');
```

Facade: Motivação

Código para parar o servidor antes do Facade

stop_server.ts

```
import { ScheduleServer } from "../ScheduleServer";

let scheduleServer: ScheduleServer = new ScheduleServer();

scheduleServer.releaseProcesses();
scheduleServer.destroy();
scheduleServer.destroySystemObjects();
scheduleServer.destroyListeners();
scheduleServer.destroyContext();
scheduleServer.shutdown();
```

Facade: Intenção

- O Facade Pattern torna uma interface complexa mais fácil de usar, usando uma classe Facade. O Facade Pattern fornece uma interface unificada para um conjunto de interface em um subsistema. O Facade define uma interface de nível superior que facilita o uso do subsistema.
- O Facade unifica as complexas interfaces de baixo nível de um subsistema em ordem para fornecer uma maneira simples de acessar essa interface. Ele apenas fornece uma camada para as interfaces complexas do subsistema, o que facilita o uso.
- A Facade não encapsula as classes ou interfaces do subsistema; Ele apenas fornece uma interface simplificada para sua funcionalidade. Um cliente pode acessar essas classes diretamente. Ele ainda expõe a funcionalidade completa do sistema para os clientes que podem precisar dele.
- Uma Facade não é apenas capaz de simplificar uma interface, mas também separa um cliente de um subsistema. Ele adere ao Princípio do Mínimo Conhecimento, que evita o acoplamento entre o cliente e o subsistema. Isso proporciona flexibilidade: suponha que, no problema relatado, a empresa deseje adicionar mais algumas etapas para iniciar ou parar o Schedule Server, que possuem suas próprias interfaces diferentes. Se você codificou seu código de cliente para a fachada em vez do subsistema, seu código de cliente não precisa ser alterado, apenas a fachada necessária para ser alterada, que seria entregue com uma nova versão para o cliente.

Facade: Código Exemplo

```
export class ScheduleServer{
    public startBootning(): void {
        console.log('Start booting...');
    }

    public readSystemConfigFile(): void {
        console.log('Reading system configuration file...');
    }

    public init(): void {
        console.log('Initializing...');
    }

    public initializeContext(): void {
        console.log('Initializing context...');
    }

    public initializeListeners(): void {
        console.log('Initializing listeners...');
    }

    public createSystemObjects(): void {
        console.log('Creating system objects...')
    }
}
```

Classe servidor

ScheduleServer.ts

Facade: Código Exemplo

```
public releaseProcesses(): void {
    console.log('Releasing processes...');
}

public destroy(): void {
    console.log('Destroying...');
}

public destroySystemObjects(): void {
    console.log('Destroying system objects...');
}

public destroyListeners(): void {
    console.log('Destroying listeners...');
}

public destroyContext(): void {
    console.log('Destroying context...');
}

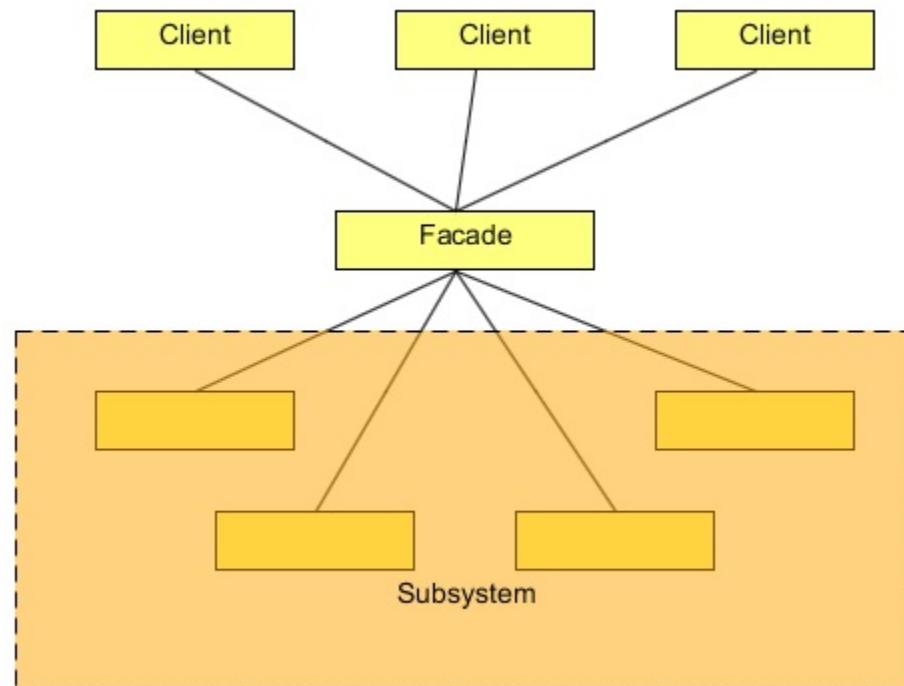
public shutdown(): void {
    console.log('Shutting down');
}
```

Classe servidor

ScheduleServer.ts

Facade: Estrutura

- Os clientes se comunicam com o subsistema enviando solicitações ao Facade, que os encaminha ao(s) objeto(s) do subsistema apropriado(s).
- Embora os objetos do subsistema executem o trabalho real, a fachada pode ter que fazer o trabalho próprio para converter sua interface em interfaces do subsistema.
- Os clientes que usam a fachada não precisam acessar diretamente os objetos do subsistema.



Facade: Código Exemplo

```
import { ScheduleServer } from "./ScheduleServer";

export class ScheduleServerFacade {
    private readonly scheduleServer: ScheduleServer;

    constructor(scheduleServer: ScheduleServer) {
        this.scheduleServer = scheduleServer;
    }

    public startServer(): void {
        this.scheduleServer.startBooting();
        this.scheduleServer.readSystemConfigFile();
        this.scheduleServer.init();
        this.scheduleServer.initializeContext();
        this.scheduleServer.initializeListeners();
        this.scheduleServer.createSystemObjects();
    }

    public stopServer(): void {
        this.scheduleServer.releaseProcesses();
        this.scheduleServer.destroy();
        this.scheduleServer.destroySystemObjects();
        this.scheduleServer.destroyListeners();
        this.scheduleServer.destroyContext();
        this.scheduleServer.shutdown();
    }
}
```

Fachada para iniciar e parar o servidor

ScheduleServerFacade.ts

Facade: Código Exemplo

Testando: iniciando e parando o servidor a partir da Facade

index.ts

```
import { ScheduleServer } from "./ScheduleServer";
import { ScheduleServerFacade } from "./ScheduleServerFacade";

let scheduleServer: ScheduleServer = new ScheduleServer();
let facadeServer: ScheduleServerFacade = new ScheduleServerFacade(scheduleServer);

facadeServer.startServer();
console.log('.....Start working.....');
console.log('.....After work done.....');
facadeServer.stopServer();
```

Facade: Código Exemplo

Resultado

```
C:\patterns\exemplos\estruturais\facade> ts-node index.ts
Start booting...
Reading system configuration file...
Initializing...
Initializing context...
Initializing listeners...
Creating system objects...
.....Start working.....
.....After work done.....
Releasing processes...
Destroying...
Destroying system objects...
Destroying listeners...
Destroying context...
Shutting down
```

Adapter

Adapter: Motivação

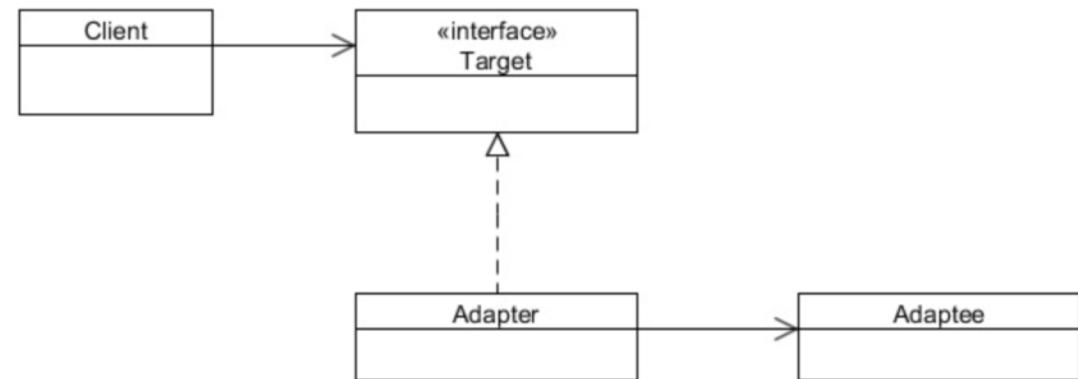
- Um desenvolvedor de software, Max, trabalhou em um site de comércio eletrônico. O site permite que os usuários façam compras e paguem on-line. O site é integrado a um portal de pagamento de terceiros, através do qual os usuários podem pagar suas contas usando seu cartão de crédito. Tudo estava indo bem, até que seu gerente o chamou para uma mudança no projeto.
- O gerente disse a ele que está planejando alterar o fornecedor do gateway de pagamento e precisa implementá-lo no código.
- O problema que surge aqui é que o site é anexado ao gateway de pagamento Xpay, que recebe um tipo de objeto Xpay. O novo fornecedor, PayD, permite apenas que o tipo de objeto PayD acesse o processo. Max não deseja alterar o conjunto de 100 classes que fazem referência a um objeto do tipo XPay. Isso também aumenta o risco do projeto, que já está sendo executado na produção. Nem ele pode alterar a ferramenta de terceiros do gateway de pagamento. O problema ocorreu devido às interfaces incompatíveis entre as duas partes diferentes do código. Para que o processo funcione, Max precisa encontrar uma maneira de tornar o código compatível com a API fornecida pelo fornecedor.

Adapter: Intenção

- O que o Max precisa aqui é um adaptador que pode se encaixar entre o código e a API do fornecedor e permitir que o processo flua. Mas antes da solução, vamos primeiro ver o que é um adaptador e como ele funciona.
- Às vezes, pode haver um cenário em que dois objetos não se encaixam, como devem ser feitos para que o trabalho seja feito. Essa situação pode surgir quando estamos tentando integrar um código legado a um novo código ou ao alterar uma API de terceiros no código. Isto é devido a interfaces incompatíveis dos dois objetos que não se encaixam.
- O padrão do adaptador permite adaptar o que um objeto ou uma classe expõe ao que outro objeto ou classe espera. Converte a interface de uma classe em outra interface que o cliente espera. Ele permite que as classes trabalhem juntas, o que não poderia ocorrer devido a interfaces incompatíveis. Permite fixar a interface entre os objetos e as classes sem modificar os objetos e as classes diretamente.
- Você pode pensar em um adaptador como um adaptador do mundo real que é usado para conectar duas peças diferentes de equipamento que não podem ser conectadas diretamente. Um adaptador fica entre esses equipamentos, obtém o fluxo do equipamento e o fornece para o outro equipamento na forma desejada, o que, de outra forma, é impossível devido às suas interfaces incompatíveis.

Adapter: Estrutura

- Um adaptador usa composição para armazenar o objeto que deve se adaptar e, quando os métodos do adaptador são chamados, ele converte essas chamadas em algo que o objeto adaptado pode entender e passa as chamadas para o objeto adaptado. O código que chama o adaptador nunca precisa saber que ele não está lidando com o tipo de objeto que ele considera, mas sim com um objeto adaptado.



Adapter: Código Exemplo

Adaptee: interface da classe concreta XPay

Xpay.ts

```
export interface Xpay{
    getCreditCardNo(): string;
    getCustomerName(): string;
    getCardExpMonth(): string;
    getCardExpYear(): string;
    getCardCVVNo(): number;
    getAmount(): number;

    setCreditCardNo(creditCardNo: string): void;
    setCustomerName(customerName: string): void;
    setCardExpMonth(cardExpMonth: string): void;
    setCardExpYear(cardExpYear: string): void;
    setCardCVVNo(cardCVVNo: number): void;
    setAmount(amount: number): void;
}
```

Adapter: Código Exemplo

Adaptee: Código da Classe Concreta XPay

XpayImpl.ts

```
import { Xpay } from "./Xpay";

export class XpayImpl implements Xpay{
    private creditCardNo!: string;
    private customerName!: string;
    private cardExpMonth!: string;
    private cardExpYear!: string;
    private cardCVVNo!: number;
    private amount!: number;
```

continua no próximo slide...

Adaptee: continuação do código da classe concreta XPay

```
/** gets */
getCreditCardNo(): string {
    return this.creditCardNo;
}
getCustomerName(): string {
    return this.customerName;
}
getCardExpMonth(): string {
    return this.cardExpMonth;
}
getCardExpYear(): string {
    return this.cardExpYear;
}
getCardCVVNo(): number {
    return this.cardCVVNo;
}
getAmount(): number {
    return this.amount;
}
```

Adaptee: continuação do código da classe concreta XPay

```
/** sets */
setCreditCardNo(creditCardNo: string): void {
    this.creditCardNo = creditCardNo;
}
setCustomerName(customerName: string): void {
    this.customerName = customerName;
}
setCardExpMonth(cardExpMonth: string): void {
    this.cardExpMonth = cardExpMonth;
}
setCardExpYear(cardExpYear: string): void {
    this.cardExpYear = cardExpYear;
}
setCardCVVNo(cardCVVNo: number): void {
    this.cardCVVNo = cardCVVNo;
}
setAmount(amount: number): void {
    this.amount = amount;
}
```

Adapter: Código Exemplo

Target Interface: interface do novo gateway de pagamentos PayD

PayD.ts

```
export interface PayD{
    getCustCardNo(): string;
    getCardOwnerName(): string;
    getCardExpMonthDate(): string;
    getCardCVVNo(): string;
    getAmount(): number;

    setCustCardNo(custCardNo: string): void;
    setCardOwnerName(cardOwnerName: string): void;
    setCardExpMonthDate(cardExpMonthDate: string): void;
    setCardCVVNo(cardCVVNo: number): void;
    setAmount(amount: number): void;
}
```

Note que as diferenças estão na data de expiração, que usar mês e ano juntos: na interface XPay, são métodos separados; nos nomes dos métodos e também no tipo dos métodos getCardCVVNo e setCardCVVNo, que agora retornam e recebem strings, respectivamente.

Adapter: Código Exemplo

Adapter: Código da Classe Concreta XPayToPayDAdapter, que usa o código existente do Adaptee XPay para implementar os métodos de PayD

```
import { PayD } from "./PayD";
import { Xpay } from "./Xpay";

export class XpayToPayDAdapter implements PayD {
    private custCardNo!: string;
    private cardOwnerName!: string;
    private cardExpMonthDate!: string;
    private cVVNo!: string;
    private totalAmount!: number;

    private readonly xpay: Xpay;
```

XpayToPayDAdapter.ts

```
constructor(xpay: Xpay){  
    this.xpay = xpay;  
    this.setProp();  
}  
getCustCardNo(): string {  
    return this.custCardNo;  
}  
getCardOwnerName(): string {  
    return this.cardOwnerName;  
}  
getCardExpMonthDate(): string {  
    return this.cardExpMonthDate;  
}  
getCVVNo(): string {  
    return this.cVVNo;  
}  
getTotalAmount(): number {  
    return this.totalAmount;  
}  
setCustCardNo(custCardNo: string): void {  
    this.custCardNo = custCardNo;  
}
```

continuação do código da Classe Concreta
XPayToPayDAdapter

```
setCardOwnerName(cardOwnerName: string): void {
    this.cardOwnerName = cardOwnerName;
}
setCardExpMonthDate(cardExpMonthDate: string): void {
    this.cardExpMonthDate = cardExpMonthDate;
}
setCVVNo(cVVNo: string): void {
    this.cVVNo = cVVNo;
}
setTotalAmount(totalAmount: number): void {
    this.totalAmount = totalAmount;
}
setProp(): void{
    this.setCardOwnerName(this.xpay.getCustomerName());
    this.setCustCardNo(this.xpay.getCreditCardNo());
    this.setCardExpMonthDate(this.xpay.getCardExpMonth() +
        "/" + this.xpay.getCardExpYear());
    this.setCVVNo(this.xpay.getCardCVVNo().toString());
    this.setTotalAmount(this.xpay.getAmount());
}
```

**continuação do código
da Classe Concreta
XPayToPayDAdapter**

Adapter: Código Exemplo

Client: roda o exemplo de adapter

index.ts

```
import { PayD } from "./PayD";
import { Xpay } from "./Xpay";
import { XpayImpl } from "./XpayImpl";
import { XpayToPayDAdapter } from "./XpayToPayDAdapter";

let xpay: Xpay = new XpayImpl();
xpay.setCreditCardNo('4789565874102365');
xpay.setCustomerName('Max Warner');
xpay.setCardExpMonth('09');
xpay.setCardExpYear('25');
xpay.setCardCVVNo(235);
xpay.setAmount(2565.23);
```

continuação do código cliente

```
let payD: PayD = new XpayToPayDAdapter(xpay);
console.log(payD.getCardOwnerName());
console.log(payD.getCustCardNo());
console.log(payD.getCardExpMonthDate());
console.log(payD.getCVVNo());
console.log(payD.getTotalAmount());
```

Adapter: Código Exemplo

Resultado

```
C:\patterns\exemplos\estruturais\adapter> ts-node index.ts
Max Warner
4789565874102365
09/25
235
2565.23
```

Decorator

Decorator : Motivação

- Uma empresa de pizzas quer fazer uma calculadora de coberturas extras.
- Um cliente pode pedir para adicionar cobertura extra a uma pizza e nosso trabalho é adicionar coberturas e aumentar seu preço usando o sistema.
- Isso é algo como adicionar uma responsabilidade extra ao nosso objeto pizza em tempo de execução e o Decorator Design Pattern é adequado para esse tipo de requisito.

Decorator: Intenção

- A intenção do Decorator Design Pattern é anexar responsabilidades adicionais a um objeto dinamicamente. Os decoradores fornecem uma alternativa flexível às subclasses para estender sua funcionalidade.
- O Padrão Decorator é usado para estender a funcionalidade de um objeto dinamicamente sem precisar alterar a fonte da classe original ou usar herança. Isso é feito criando um *wrapper* de objeto chamado Decorator em torno do objeto real.
- O objeto Decorator foi projetado para ter a mesma interface que o objeto subjacente. Isso permite que um objeto cliente interaja com o objeto Decorator exatamente da mesma maneira que faria com o objeto real subjacente. O objeto Decorator contém uma referência ao objeto real.

Decorator: Intenção

- O objeto Decorator recebe todas as solicitações (chamadas) de um cliente. Por sua vez, ele encaminha essas chamadas para o objeto subjacente. O objeto Decorator adiciona algumas funcionalidades adicionais antes ou depois de encaminhar solicitações para o objeto subjacente. Isso garante que a funcionalidade adicional possa ser adicionada a um determinado objeto externamente em tempo de execução sem modificar sua estrutura.
- Decorator evita a proliferação de subclasses levando a menos complexidade e confusão. É fácil adicionar qualquer combinação de recursos. A mesma capacidade pode até ser adicionada duas vezes. Torna-se possível ter objetos decoradores diferentes para um determinado objeto simultaneamente. Um cliente pode escolher quais recursos deseja enviando mensagens para um decorador apropriado.

Decorator: Estrutura

Componente

- Define a interface para objetos que podem ter responsabilidades adicionadas a eles dinamicamente.

Componente Concreto

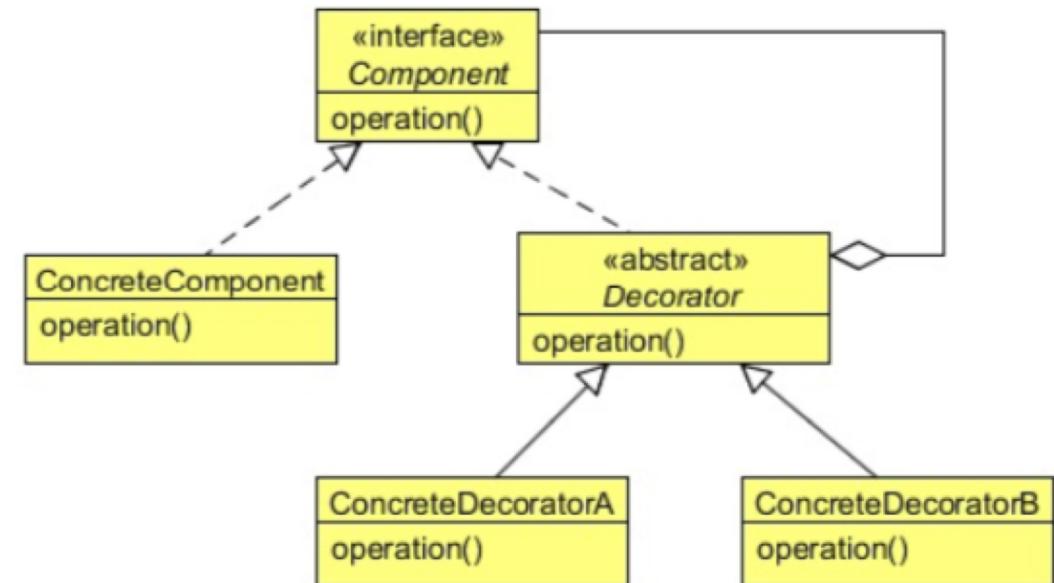
- Define um objeto ao qual podem ser atribuídas responsabilidades adicionais.

Decorador

- Mantém uma referência a um objeto Componente e define uma interface que está em conformidade com a interface do Componente.

Decorador Concreto

- Adiciona responsabilidades ao componente.



Decorator: Código Exemplo

Component: interface da classe que terá responsabilidades adicionadas de maneira dinâmica

Pizza.ts

```
export interface Pizza {  
    getDesc(): string;  
    getPrice(): number;  
}
```

Decorator: Código Exemplo

ConcreteComponent: classe que terá responsabilidades adicionadas de maneira dinâmica

SimplyVegPizza.ts

```
import { Pizza } from "./Pizza";

export class SimplyVegPizza implements Pizza{
    getDesc(): string {
        return 'SimplyVegPizza (230)';
    }
    getPrice(): number {
        return 230;
    }
}
```

Decorator: Código Exemplo

ConcreteComponent: classe que terá responsabilidades adicionadas de maneira dinâmica

SimplyNonVegPizza.ts

```
import { Pizza } from "./Pizza";

export class SimplyNonVegPizza implements Pizza{
    getDesc(): string {
        return 'SimplyNonVegPizza (350)';
    }
    getPrice(): number {
        return 350;
    }
}
```

Decorator: Código Exemplo

Decorator: classe abstrata que adicionará responsabilidades ao Component de maneira dinâmica

PizzaDecorator.ts

```
import { Pizza } from "./Pizza";

export abstract class PizzaDecorator implements Pizza{
    public getDesc(): string {
        return "Toppings";
    }

    public abstract getPrice(): number;
}
```

Decorator: Código Exemplo

ConcreteDecorator: classe concreta que adicionará responsabilidades ao Component

Broccoli.ts

```
import { Pizza } from "./Pizza";
import { PizzaDecorator } from "./PizzaDecorator";

export class Broccoli extends PizzaDecorator{
    private readonly pizza: Pizza;

    constructor(pizza: Pizza){
        super();
        this.pizza = pizza;
    }

    public getDesc(): string {
        return this.pizza.getDesc() + ', Broccoli (9.25)';
    }

    public getPrice(): number {
        return this.pizza.getPrice() + 9.25;
    }
}
```

Decorator: Código Exemplo

ConcreteDecorator: classe concreta que adicionará responsabilidades ao Component

Cheese.ts

```
import { Pizza } from "./Pizza";
import { PizzaDecorator } from "./PizzaDecorator";

export class Cheese extends PizzaDecorator{
    private readonly pizza: Pizza;

    constructor(pizza: Pizza){
        super();
        this.pizza = pizza;
    }

    public getDesc(): string {
        return this.pizza.getDesc() + ', Cheese (20.72)';
    }

    public getPrice(): number {
        return this.pizza.getPrice() + 20.72;
    }
}
```

Decorator: Código Exemplo

ConcreteDecorator: classe concreta que adicionará responsabilidades ao Component

FetaCheese.ts

```
import { Pizza } from "./Pizza";
import { PizzaDecorator } from "./PizzaDecorator";

export class FetaCheese extends PizzaDecorator{
    private readonly pizza: Pizza;

    constructor(pizza: Pizza){
        super();
        this.pizza = pizza;
    }

    public getDesc(): string {
        return this.pizza.getDesc() + ', FetaCheese (25.88)';
    }

    public getPrice(): number {
        return this.pizza.getPrice() + 25.88;
    }
}
```

Decorator: Código Exemplo

ConcreteDecorator: classe concreta que adicionará responsabilidades ao Component

Spinach.ts

```
import { Pizza } from "./Pizza";
import { PizzaDecorator } from "./PizzaDecorator";

export class Spinach extends PizzaDecorator{
    private readonly pizza: Pizza;

    constructor(pizza: Pizza){
        super();
        this.pizza = pizza;
    }

    public getDesc(): string {
        return this.pizza.getDesc() + ', Spinach (7.92)';
    }

    public getPrice(): number {
        return this.pizza.getPrice() + 7.92;
    }
}
```

Decorator: Código Exemplo

ConcreteDecorator: classe concreta que adicionará responsabilidades ao Component

RomaTomatoes.ts

```
import { Pizza } from "./Pizza";
import { PizzaDecorator } from "./PizzaDecorator";

export class RomaTomatoes extends PizzaDecorator{
    private readonly pizza: Pizza;

    constructor(pizza: Pizza){
        super();
        this.pizza = pizza;
    }

    public getDesc(): string {
        return this.pizza.getDesc() + ', RomaTomatoes (5.20)';
    }

    public getPrice(): number {
        return this.pizza.getPrice() + 5.20;
    }
}
```

Decorator: Código Exemplo

ConcreteDecorator: classe concreta que adicionará responsabilidades ao Component

RedOnions.ts

```
import { Pizza } from "./Pizza";
import { PizzaDecorator } from "./PizzaDecorator";

export class RedOnions extends PizzaDecorator{
    private readonly pizza: Pizza;

    constructor(pizza: Pizza){
        super();
        this.pizza = pizza;
    }

    public getDesc(): string {
        return this.pizza.getDesc() + ', RedOnions (3.75)';
    }

    public getPrice(): number {
        return this.pizza.getPrice() + 3.75;
    }
}
```

Decorator: Código Exemplo

ConcreteDecorator: classe concreta que adicionará responsabilidades ao Component

Meat.ts

```
import { Pizza } from "./Pizza";
import { PizzaDecorator } from "./PizzaDecorator";

export class Meat extends PizzaDecorator{
    private readonly pizza: Pizza;

    constructor(pizza: Pizza){
        super();
        this.pizza = pizza;
    }

    public getDesc(): string {
        return this.pizza.getDesc() + ', Meat (14.25)';
    }

    public getPrice(): number {
        return this.pizza.getPrice() + 14.25;
    }
}
```

Decorator: Código Exemplo

ConcreteDecorator: classe concreta que adicionará responsabilidades ao Component

Ham.ts

```
import { Pizza } from "./Pizza";
import { PizzaDecorator } from "./PizzaDecorator";

export class Ham extends PizzaDecorator{
    private readonly pizza: Pizza;

    constructor(pizza: Pizza){
        super();
        this.pizza = pizza;
    }

    public getDesc(): string {
        return this.pizza.getDesc() + ', Ham (18.12)';
    }

    public getPrice(): number {
        return this.pizza.getPrice() + 18.12;
    }
}
```

Decorator: Código Exemplo

ConcreteDecorator: classe concreta que adicionará responsabilidades ao Component

GreenOlives.ts

```
import { Pizza } from "./Pizza";
import { PizzaDecorator } from "./PizzaDecorator";

export class GreenOlives extends PizzaDecorator{
    private readonly pizza: Pizza;

    constructor(pizza: Pizza){
        super();
        this.pizza = pizza;
    }

    public getDesc(): string {
        return this.pizza.getDesc() + ', GreenOlives (5.47)';
    }

    public getPrice(): number {
        return this.pizza.getPrice() + 5.47;
    }
}
```

Decorator: Código Exemplo

ConcreteDecorator: classe concreta que adicionará responsabilidades ao Component

Chicken.ts

```
import { Pizza } from "./Pizza";
import { PizzaDecorator } from "./PizzaDecorator";

export class Chicken extends PizzaDecorator{
    private readonly pizza: Pizza;

    constructor(pizza: Pizza){
        super();
        this.pizza = pizza;
    }

    public getDesc(): string {
        return this.pizza.getDesc() + ', Chicken (12.75)';
    }

    public getPrice(): number {
        return this.pizza.getPrice() + 12.75;
    }
}
```

Decorator: Código Exemplo

Teste

index.ts

```
import { Cheese } from "./Cheese";
import { GreenOlives } from "./GreenOlives";
import { Ham } from "./Ham";
import { Meat } from "./Meat";
import { Pizza } from "./Pizza";
import { RomaTomatoes } from "./RomaTomatoes";
import { SimplyNonVegPizza } from "./SimplyNonVegPizza";
import { SimplyVegPizza } from "./SimplyVegPizza";
import { Spinach } from "./Spinach";

let pizza: Pizza = new SimplyVegPizza()
pizza = new RomaTomatoes(pizza);
pizza = new GreenOlives(pizza);
pizza = new Spinach(pizza);
console.log("Desc:", pizza.getDesc())
console.log("Price:", pizza.getPrice().toFixed(2));
```

Decorator: Código Exemplo

Teste - continuação

```
 pizza = new SimplyNonVegPizza()
pizza = new Meat(pizza);
pizza = new Cheese(pizza);
pizza = new Cheese(pizza);
pizza = new Ham(pizza);
console.log("Desc:", pizza.getDesc())
console.log("Price:", pizza.getPrice().toFixed(2));
```

Decorator: Código Exemplo

Resultado

```
C:\patterns\exemplos\estruturais\decorator>ts-node index.ts
Desc: SimplyVegPizza (230), RomaTomatoes (5.20), GreenOlives (5.47),
Spinach (7.92)
Price: 248.59
Desc: SimplyNonVegPizza (350), Meat (14.25), Cheese (20.72), Cheese
(20.72), Ham (18.12)
Price: 423.81
```