

Design Patterns Comportamentais

Strategy, Template, Observer, Chain of Responsibility

2022

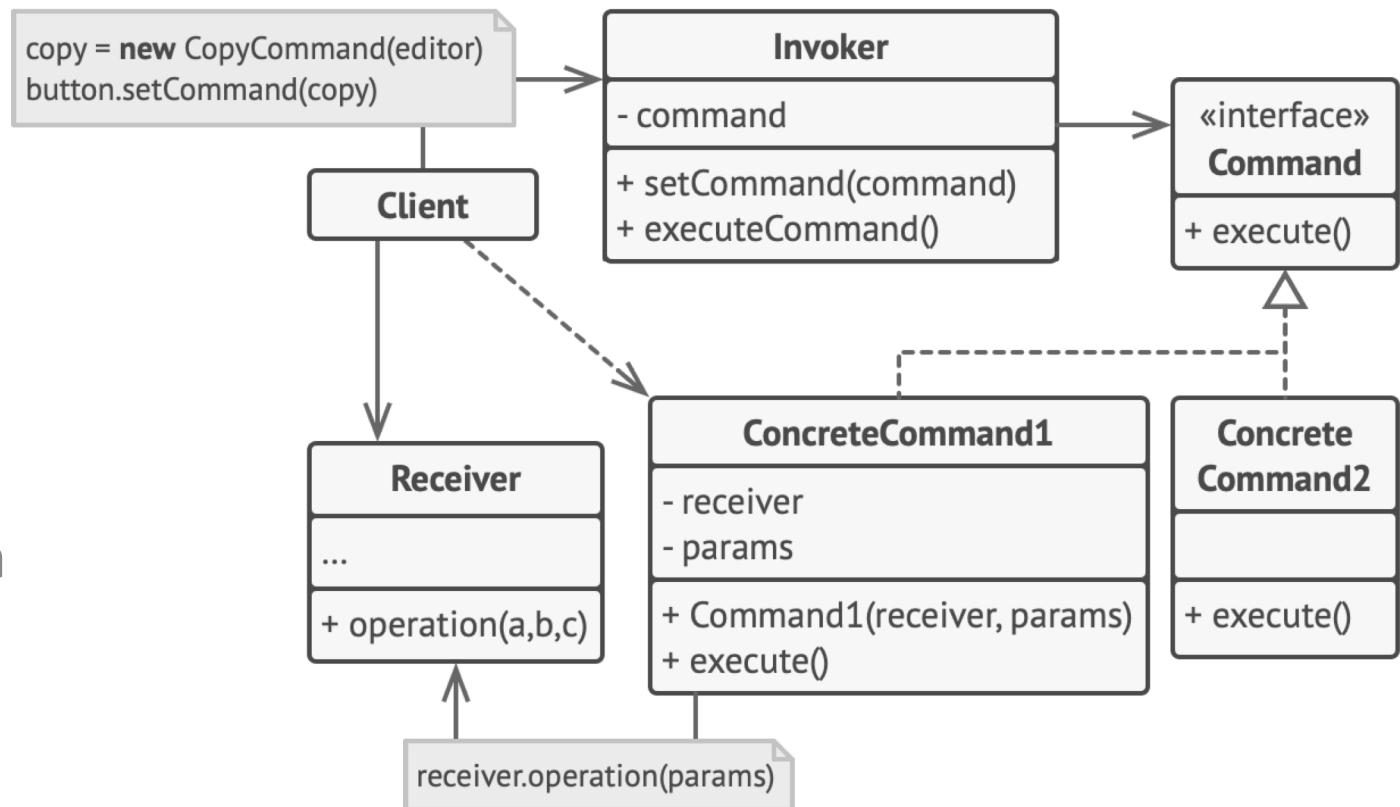
Prof. Sergio Bonato

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.

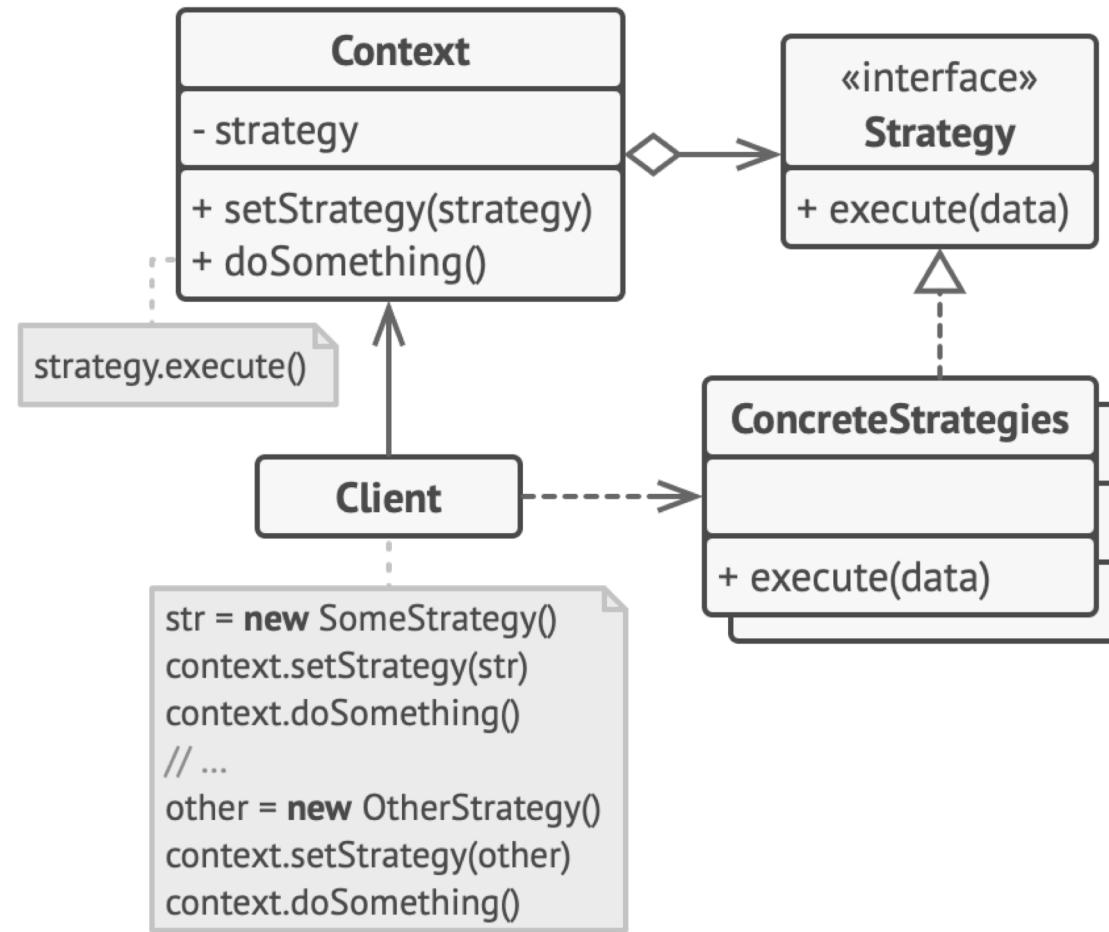
Command

O Command é um padrão de projeto comportamental que transforma um pedido em um objeto independente que contém toda a informação sobre o pedido. Essa transformação permite que você parametrize métodos com diferentes pedidos.



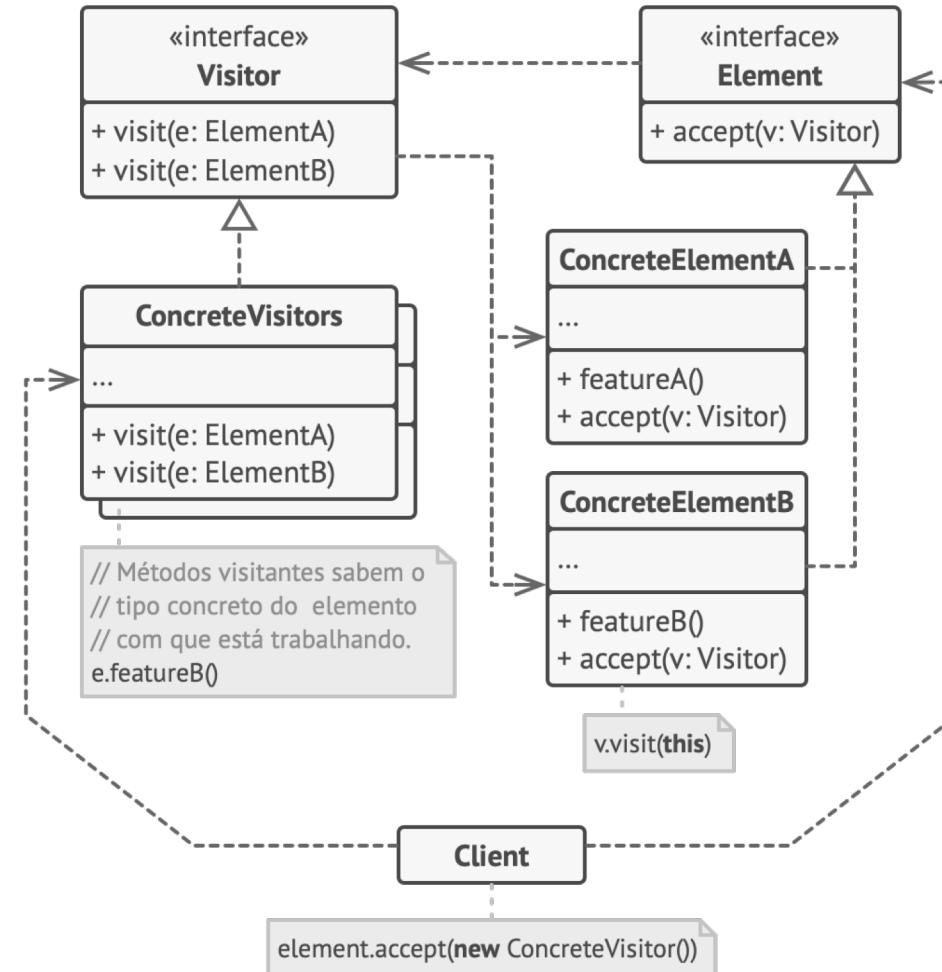
Strategy

O Strategy é um padrão de projeto comportamental que permite que você defina uma família de algoritmos, coloque-os em classes separadas, e faça os objetos deles intercambiáveis.



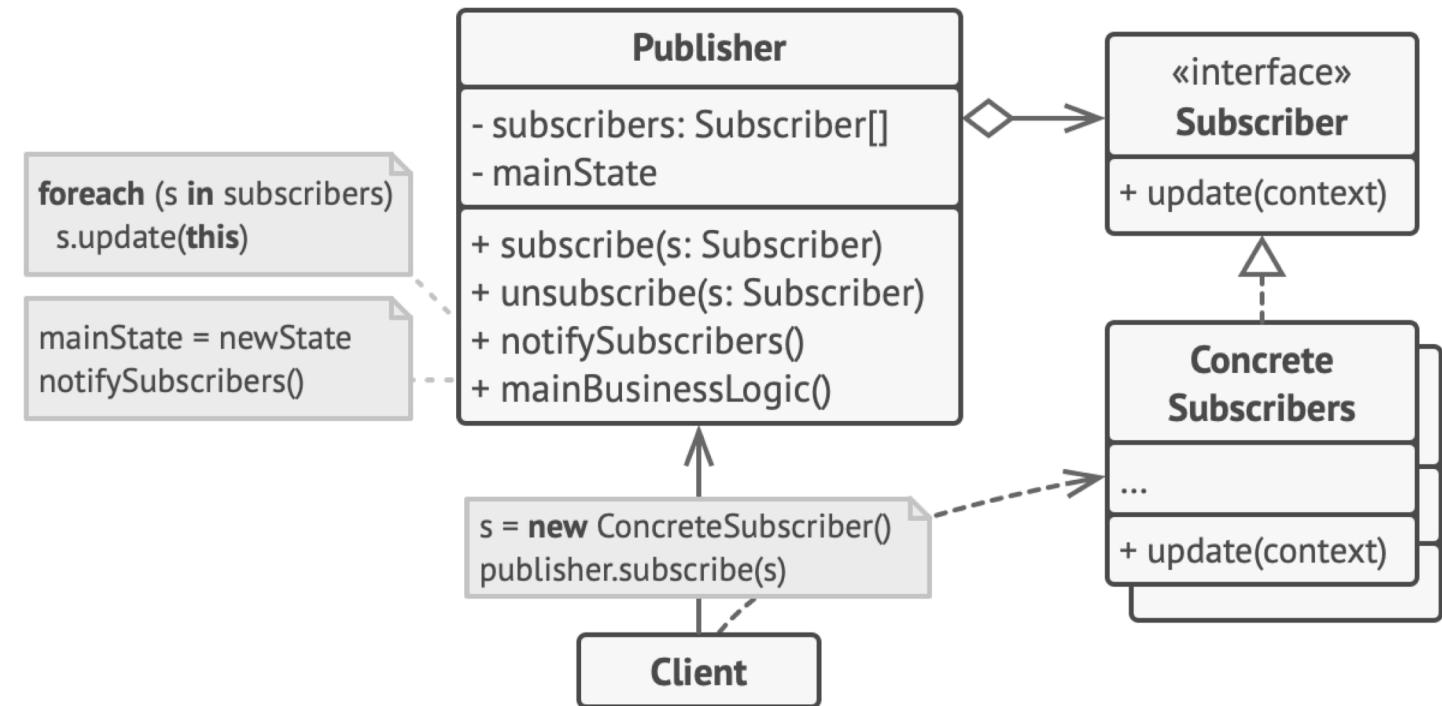
Visitor

O Visitor é um padrão de projeto comportamental que permite que você separe algoritmos dos objetos nos quais eles operam.



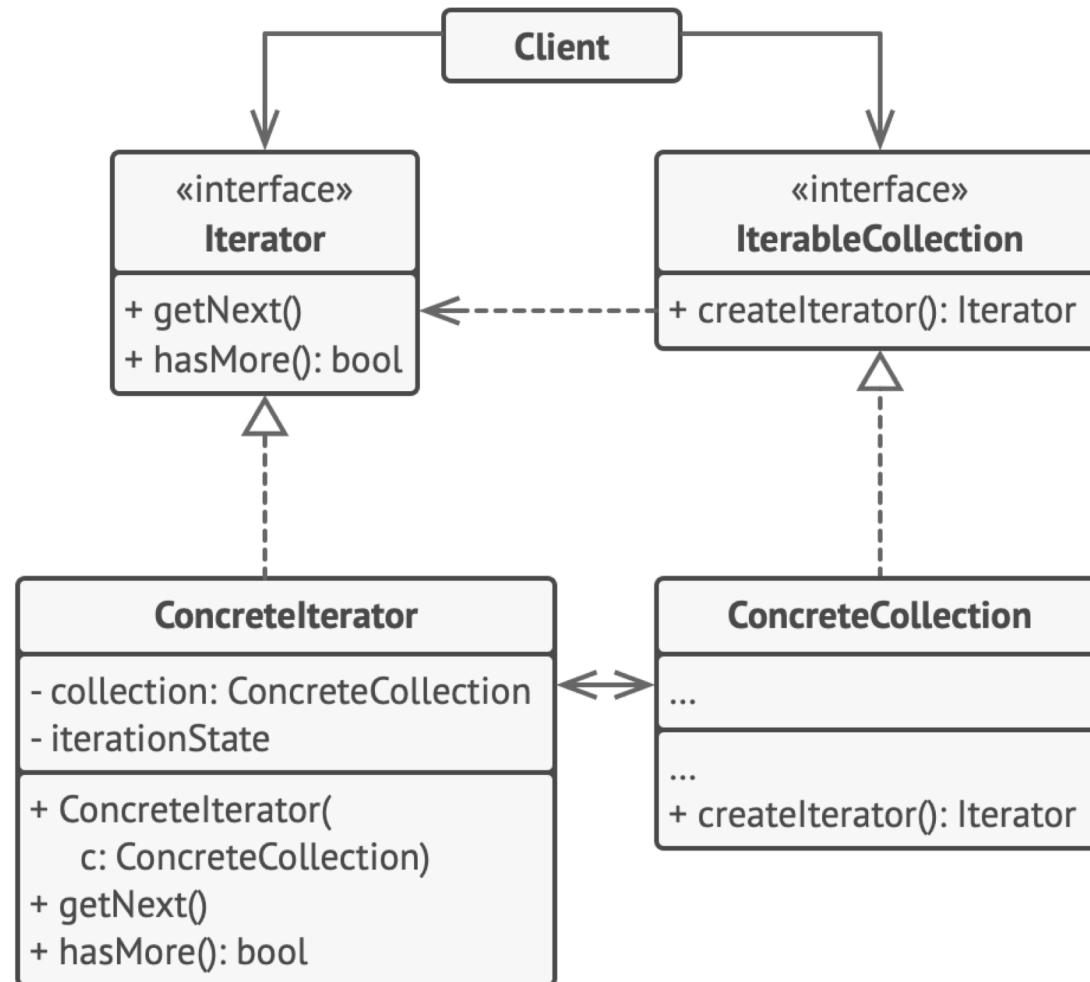
Observer

O Observer é um padrão de projeto comportamental que permite que você defina um mecanismo de inscrição para notificar múltiplos objetos sobre quaisquer eventos que aconteçam com o objeto que eles estão observando.



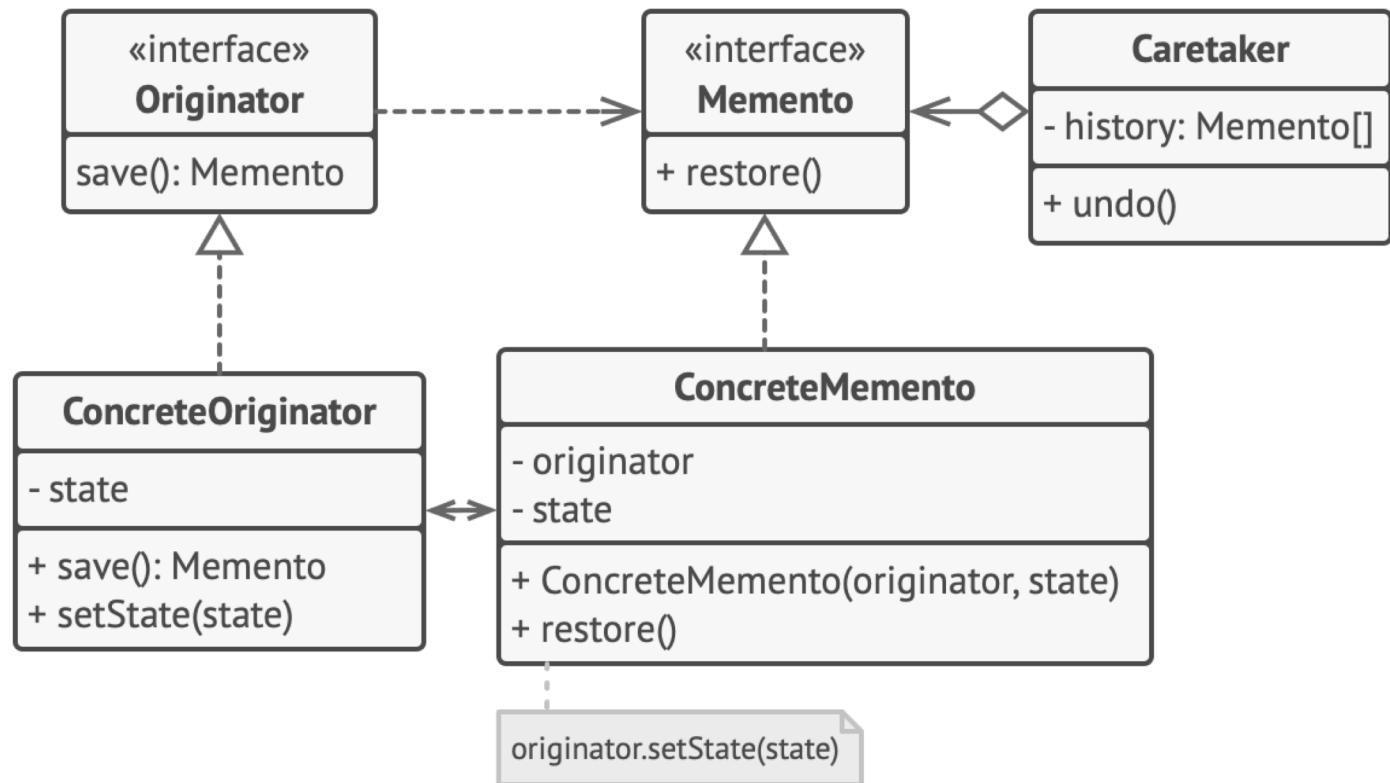
Iterator

O Iterator é um padrão de projeto comportamental que permite a você percorrer elementos de uma coleção sem expor as representações dele (lista, pilha, árvore, etc.)



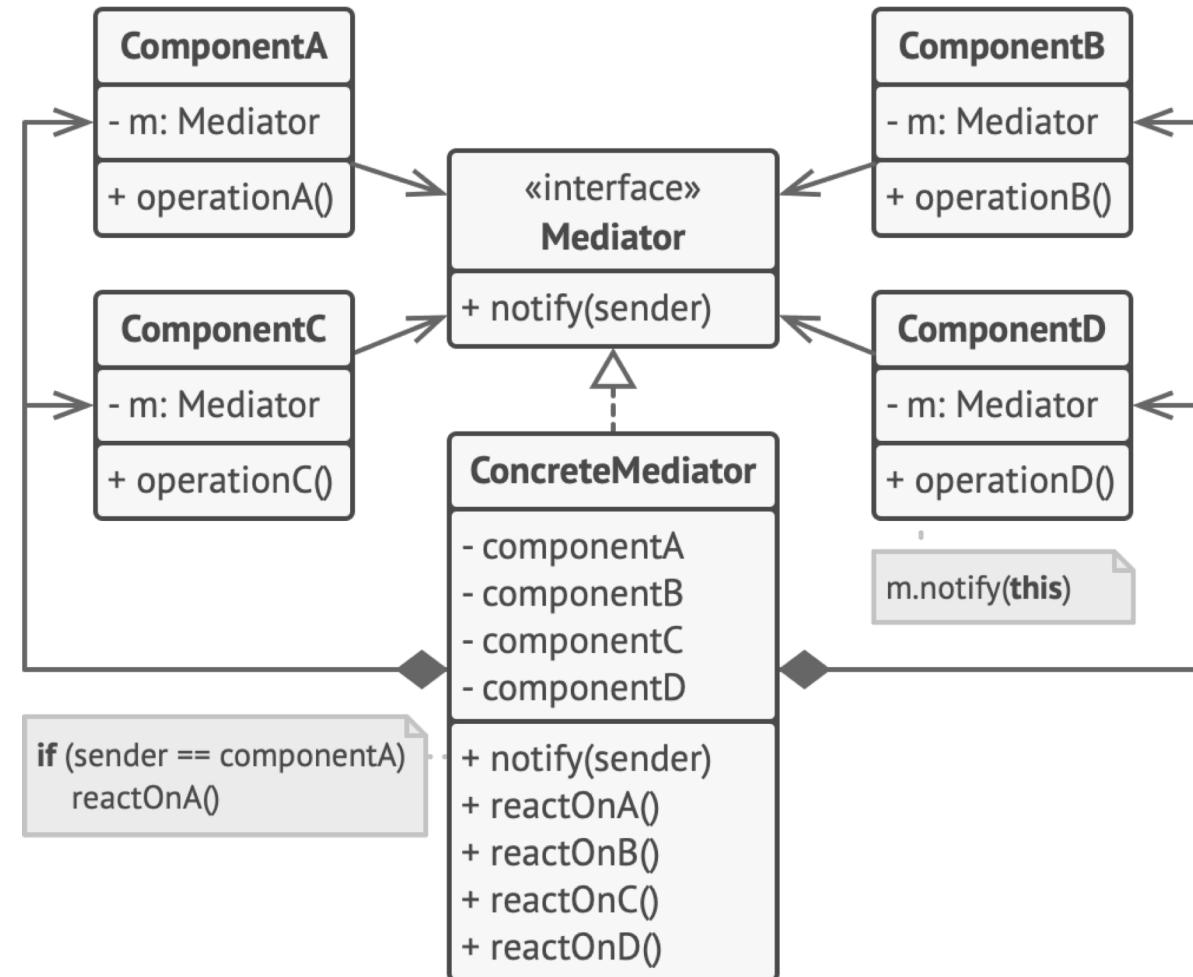
Memento

O Memento é um padrão de projeto comportamental que permite que você salve e restaure o estado anterior de um objeto sem revelar os detalhes de sua implementação.



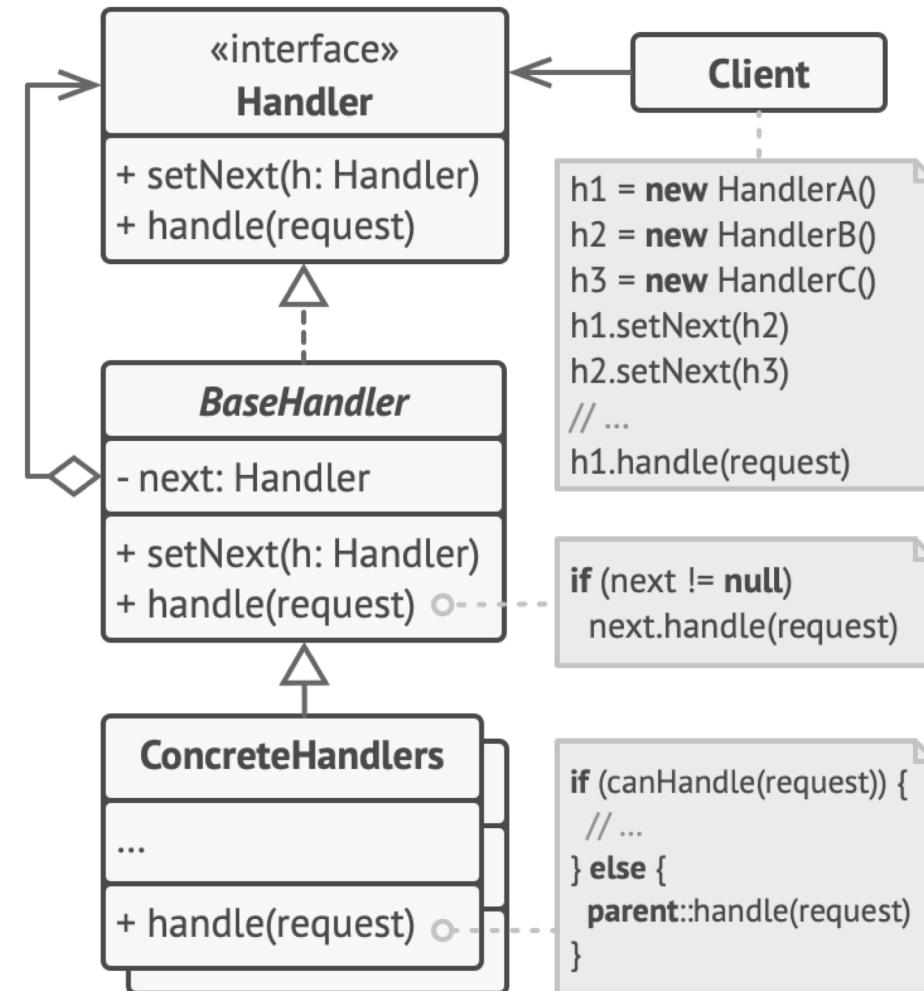
Mediator

O Mediator é um padrão de projeto comportamental que permite que você reduza as dependências caóticas entre objetos. O padrão restringe comunicações diretas entre objetos e os força a colaborar apenas através do objeto mediador.



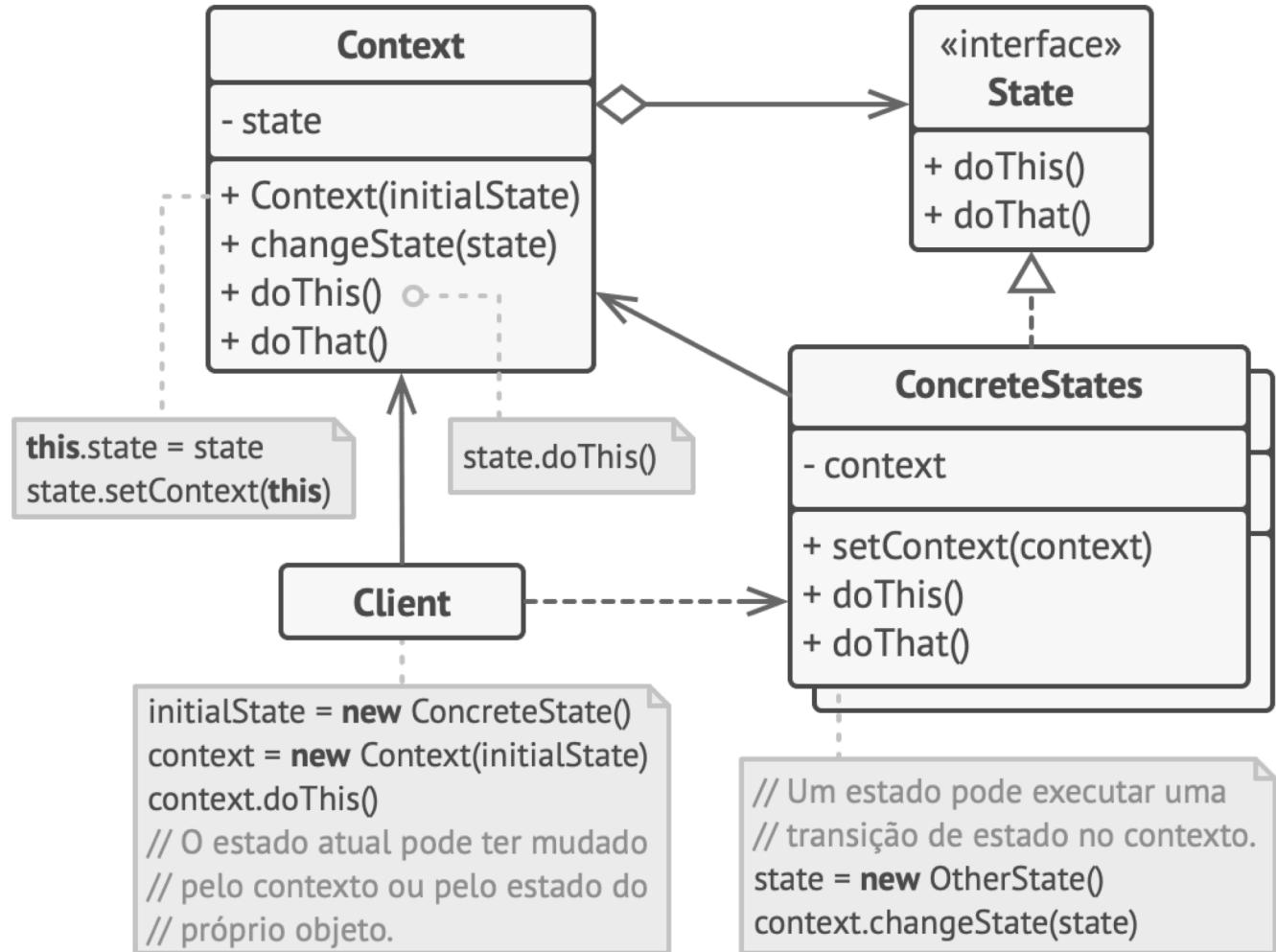
Chain of responsibility

O Chain of Responsibility é um padrão de projeto comportamental que permite que você passe pedidos por uma corrente de handlers. Ao receber um pedido, cada handler decide se processa o pedido ou o passa adiante para o próximo handler na corrente.



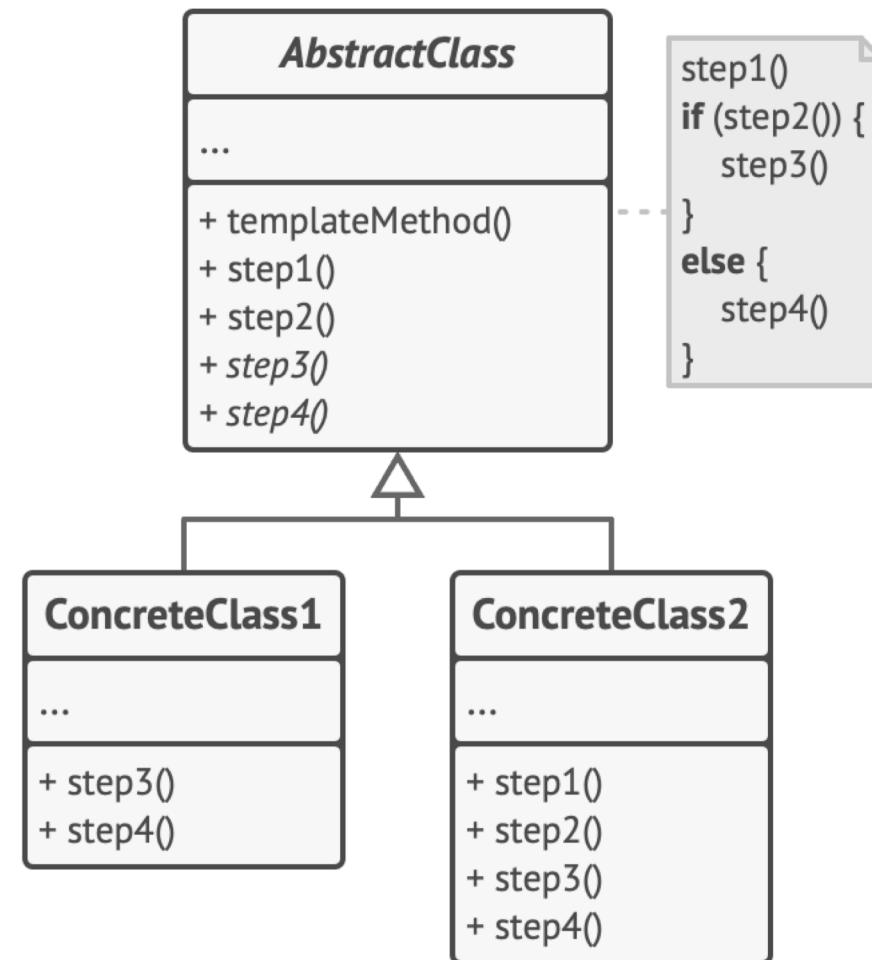
State

O State é um padrão de projeto comportamental que permite que um objeto altere seu comportamento quando seu estado interno muda. Parece como se o objeto mudasse de classe.



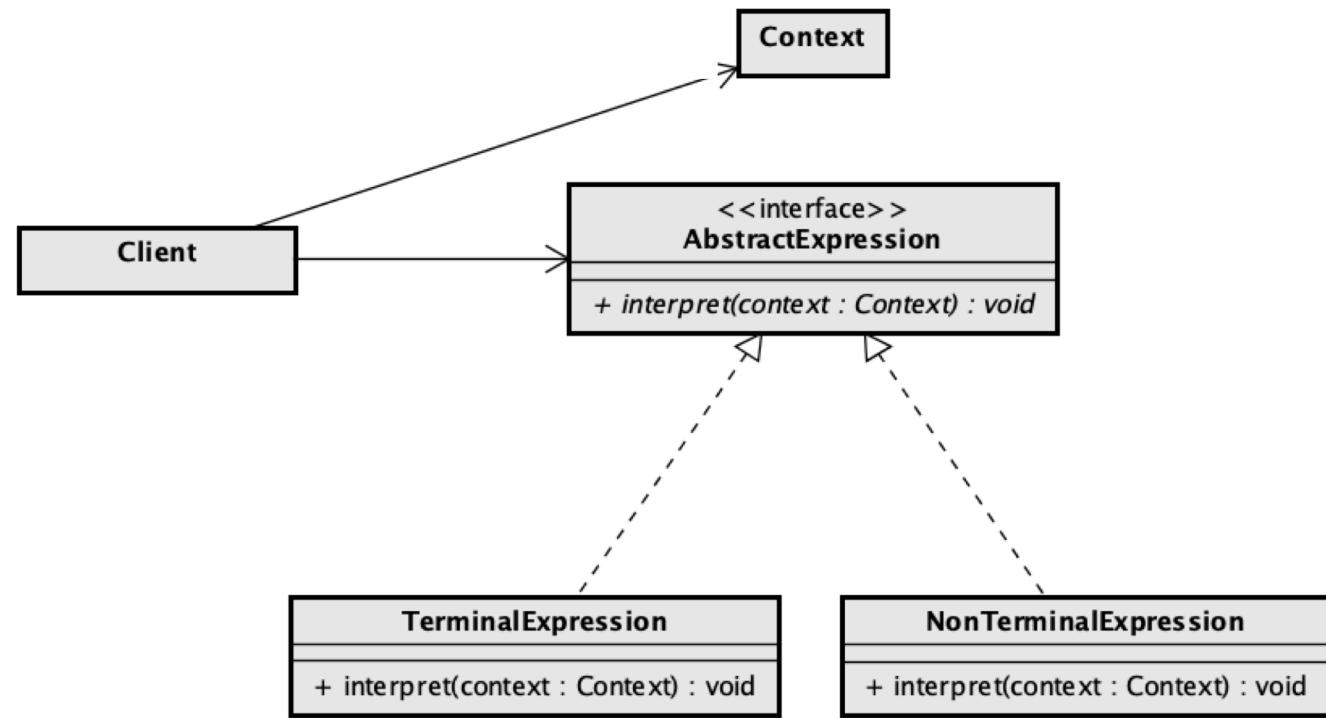
Template method

O Template Method é um padrão de projeto comportamental que define o esqueleto de um algoritmo na superclasse mas deixa as subclasses sobreescriverem etapas específicas do algoritmo sem modificar sua estrutura.



Interpreter

Dada uma determinada linguagem, o padrão Interpreter define uma representação para sua gramática e também um interpretador que usa a representação para interpretar sentenças da linguagem.



Detalhes de Implementação

- Strategy
- Template Method
- Observer
- Chain of Responsibility

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>

Strategy

Strategy: Motivação

- O Strategy Design Pattern parece ser o mais simples de todos os padrões de projeto, mas oferece grande flexibilidade ao seu código. Esse padrão é usado em quase todos os lugares, mesmo em conjunto com outros padrões de projeto.
- Para entender o padrão de design strategy, vamos criar um formatador de texto para um editor de texto. Um editor de texto pode ter diferentes formatadores de texto para formatar o texto. Podemos criar diferentes formatadores de texto e depois passar o necessário para o editor de texto, para que o editor possa formatar o texto conforme o exigido.
- O editor de texto manterá uma referência a uma interface comum para o formatador de texto e o trabalho do editor será passar o texto ao formatador para formatar o texto.

Strategy: Intenção

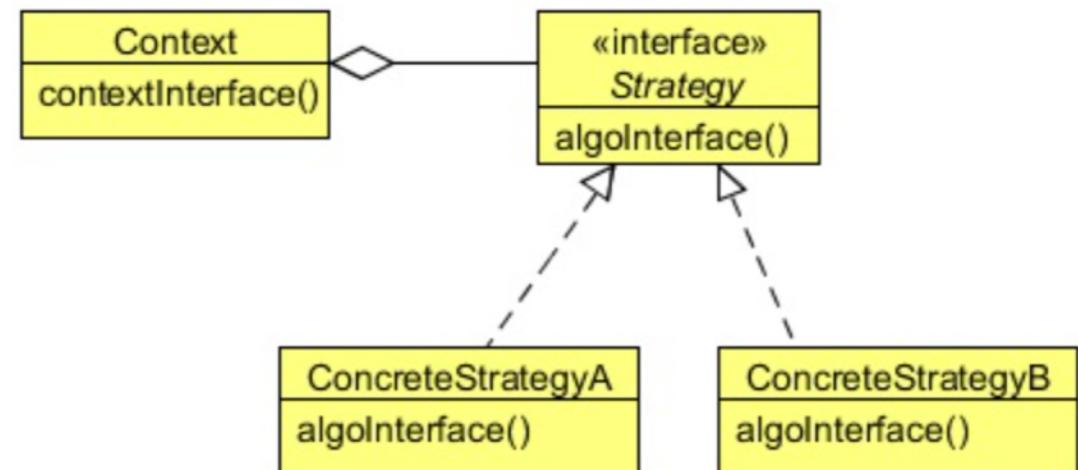
- O Strategy Design Pattern define uma família de algoritmos, encapsulando cada um deles, e tornando-os intercambiáveis. A estratégia permite que o algoritmo varie independentemente dos clientes que o usam.
- O padrão Strategy é útil quando há um conjunto de algoritmos relacionados e um objeto cliente precisa ser capaz de escolher e escolher dinamicamente um algoritmo desse conjunto que atenda à sua necessidade atual. O padrão Strategy sugere manter a implementação de cada um dos algoritmos em uma classe separada. Cada um desses algoritmos encapsulados em uma classe separada é chamado de estratégia. Um objeto que usa um objeto Strategy é geralmente chamado de objeto de contexto.

Strategy: Intenção

- Com diferentes objetos de Estratégia, alterar o comportamento de um objeto Contexto é simplesmente uma questão de mudar seu objeto Estratégia para aquele que implementa o algoritmo requerido. Para permitir que um objeto Contexto acesse diferentes objetos de Estratégia de maneira transparente, todos os objetos de Estratégia devem ser projetados para oferecer a mesma interface. Na linguagem de programação Java, isso pode ser feito projetando cada objeto de Estratégia como um implementador de uma interface comum ou como uma subclasse de uma classe abstrata comum que declara a interface comum requerida.
- Depois que o grupo de algoritmos relacionados é encapsulado em um conjunto de classes de estratégia em uma hierarquia de classes, um cliente pode escolher entre esses algoritmos selecionando e instanciando uma classe de estratégia apropriada. Para alterar o comportamento do contexto, um objeto cliente precisa configurar o contexto com a instância de estratégia selecionada. Esse tipo de arranjo separa completamente a implementação de um algoritmo do contexto que o utiliza. Como resultado, quando uma implementação de algoritmo existente é alterada ou um novo algoritmo é adicionado ao grupo, tanto o contexto quanto o objeto cliente (que usa o contexto) permanecem inalterados.

Strategy: Estrutura

- **Strategy**
 - Declara uma interface comum a todos os algoritmos suportados. O Context usa essa interface para chamar o algoritmo definido por um ConcreteStrategy.
- **ConcreteStrategy**
 - Implementa o algoritmo usando a interface Strategy.
- **Context**
 - Está configurado com um objeto ConcreteStrategy.
 - Mantém uma referência a um objeto Strategy.
 - Pode definir uma interface que permita à Strategy acessar seus dados.



Strategy: Código Exemplo

Strategy: a interface de um formatador de textos

TextFormatter.ts

```
export interface TextFormatter {  
    format(text: string): void;  
}
```

Strategy: Código Exemplo

ConcreteStrategy: 2 diferentes formatações de texto

Todas Maiúsculas

CapTextFormatter.ts

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

export class CapTextFormatter implements TextFormatter{
    public format(text: string): void {
        console.log(`[CapTextFormatter]: ${text.toUpperCase()}`);
    }
}
```

Strategy: Código Exemplo

ConcreteStrategy: 2 diferentes formatações de texto

Todas Minúsculas

LowerTextFormatter.ts

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

export class LowerTextFormatter implements TextFormatter{
    public format(text: string): void {
        console.log(`[LowerTextFormatter]: ${text.toLowerCase()}`);
    }
}
```

Strategy: Código Exemplo

Context: guarda a referência para a Estratégia – o formatador

TextEditor.ts

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

export class TextEditor {
    private readonly textFormatter: TextFormatter;

    constructor(textFormatter: TextFormatter){
        this.textFormatter = textFormatter;
    }

    public publishText(text: string): void {
        this.textFormatter.format(text);
    }
}
```

Strategy: Código Exemplo

Testando

index.ts

```
import { CapTextFormatter } from "./CapTextFormatter";
import { LowerTextFormatter } from "./LowerTextFormatter";
import { TextEditor } from "./TextEditor";
import { TextFormatter } from "./TextFormatter";

let formatter: TextFormatter = new CapTextFormatter();
let editor: TextEditor = new TextEditor(formatter);
editor.publishText('Testing text in caps formatter');

formatter = new LowerTextFormatter();
editor = new TextEditor(formatter);
editor.publishText('Testing text in lower formatter');
```

Strategy: Código Exemplo

Resultado

```
C:\patterns\exemplos\comportamentais\strategy>ts-node index.ts
[CapTextFormatter]: TESTING TEXT IN CAPS FORMATTER
[LowerTextFormatter]: testing text in lower formatter
```

Template

Template: Motivação

- Você já se conectou a um banco de dados de relacionamentos usando seu aplicativo Java? Vamos lembrar de algumas etapas importantes necessárias para conectar e inserir dados no banco de dados. Primeiro, precisamos de um driver de acordo com o banco de dados com o qual queremos nos conectar. Em seguida, passamos algumas credenciais para o banco de dados e, em seguida, preparamos uma instrução, definimos os dados na instrução de inserção e os inserimos usando o comando insert. Mais tarde, fechamos todas as conexões e, opcionalmente, destruímos todos os objetos de conexão.
- Você precisa escrever todas essas etapas, independentemente do banco de dados relacional de qualquer fornecedor. Considere um problema em que você precisa inserir alguns dados nos diferentes bancos de dados. Você precisa buscar alguns dados de um arquivo CSV e inseri-los em um banco de dados MySQL. Alguns dados vêm de um arquivo de texto e devem ser inseridos em um banco de dados Oracle. A única diferença é o driver e os dados, o restante das etapas deve ser o mesmo, já que o JDBC fornece um conjunto comum de interfaces para se comunicar com o banco de dados de relação específico de qualquer fornecedor.
- Podemos criar um modelo, que irá realizar algumas etapas para o cliente, e deixaremos algumas etapas para permitir que o cliente as implemente de maneira específica. Opcionalmente, um cliente pode substituir o comportamento padrão de algumas etapas já definidas.

Template: Intenção

- O Template Design Pattern é um padrão de comportamento e, como o nome sugere, ele fornece um modelo ou uma estrutura de um algoritmo que é usado pelos usuários. Um usuário fornece sua própria implementação sem alterar a estrutura do algoritmo.
- O Template Pattern define o esqueleto de um algoritmo em uma operação, adiando algumas etapas para subclasses. O Template Method permite que as subclasses redefinam certas etapas de um algoritmo sem alterar a estrutura do algoritmo.

Template: Intenção

- O padrão Template Method pode ser usado em situações em que há um algoritmo, e algumas etapas podem ser implementadas de várias maneiras diferentes. Em tais cenários, o padrão Template Method sugere manter o contorno do algoritmo em um método separado chamado de template em uma classe, que pode ser referida como uma classe template, deixando de fora as implementações específicas das partes variantes (passos que podem ser implementados de várias maneiras diferentes) do algoritmo para diferentes subclasses dessa classe.
- A classe Template não precisa necessariamente deixar a implementação para subclasses em sua totalidade. Em vez disso, como parte do fornecimento do esboço do algoritmo, a classe Template também pode fornecer uma certa quantidade de implementação que pode ser considerada invariante em diferentes implementações. Ele pode até fornecer implementação padrão para as partes variantes, se apropriado. Apenas detalhes específicos serão implementados dentro de diferentes subclasses. Esse tipo de implementação elimina a necessidade de código duplicado, o que significa uma quantidade mínima de código a ser gravada.

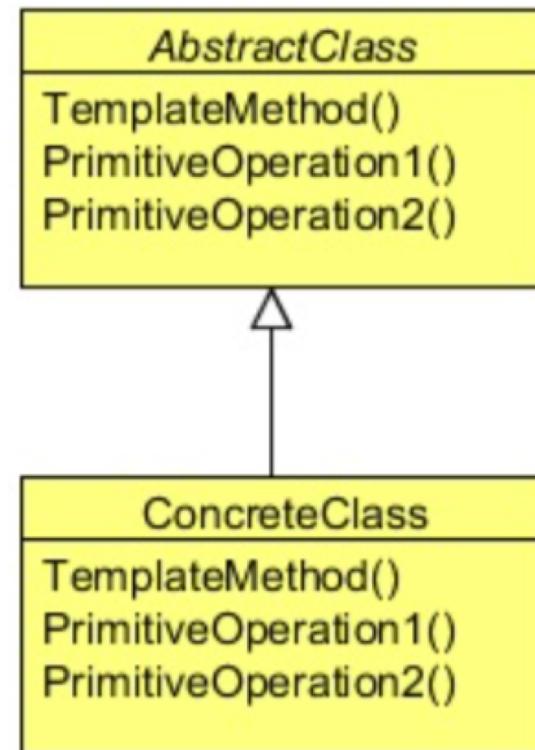
Template: Estrutura

- **AbstractClass**

- Define operações primitivas abstratas que subclasses concretas definem para implementar etapas de um algoritmo.
- Implementa um método de modelo que define o esqueleto de um algoritmo. O método de modelo também chama operações primitivas como operações definidas no AbstractClass ou de outros objetos.

- **ConcreteClass**

- Implementa as operações primitivas para transportar.



Template: Código Exemplo

AbstractClass: operações genéricas de JDBC

ConnectionTemplate.ts

```
export abstract class ConnectionTemplate {
    public run(): void {
        this.setDBDriver();
        this.setCredentials();
        this.connect();
        this.preparedStatement();
        this.setData();
        this.insert();
        this.close();
        this.destroy();
    }

    public abstract setDBDriver(): void;
    public abstract setCredentials(): void;
```

continuação da AbstractClass: operações genéricas de JDBC

```
public connect(): void {
    console.log('Setting connection...');

}

public preparedStatement(): void {
    console.log('Preparing insert statement...');

}

public abstract setData(): void;

public insert(): void {
    console.log('Insert data...');

}

public close(): void {
    console.log('Closing connections...');

}

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

}
```

Template: Código Exemplo

ConcreteClass: implementação para MySQL

MySQL

MySqlCSVCon.ts

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

export class MySqlCSVCon extends ConnectionTemplate {
    public setDBDriver(): void {
        console.log('Setting MySQL DB drivers...');
    }
    public setCredentials(): void {
        console.log('Setting credentials for MySQL DB...');
    }
    public setData(): void {
        console.log('Setting up data from csv file...');
    }
}
```

Template: Código Exemplo

ConcreteClass: implementação para Oracle

Oracle

OracleTxtCon.ts

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

export class OracleTxtCon extends ConnectionTemplate {
    public setDBDriver(): void {
        console.log('Setting Oracle DB drivers...');
    }
    public setCredentials(): void {
        console.log('Setting credentials for Oracle DB...');
    }
    public setData(): void {
        console.log('Setting up data from txt file...');
    }
}
```

Template: Código Exemplo

Testando

index.ts

```
import { ConnectionTemplate } from "./ConnectionTemplate";
import { MySqlCSVCon } from "./MySqlCSVCon";
import { OracleTxtCon } from "./OracleTxtCon";

console.log('For MySQL...');
let template: ConnectionTemplate = new MySqlCSVCon();
template.run();

console.log('For Oracle...');
template = new OracleTxtCon();
template.run();
```

Template: Código Exemplo

Resultado

```
C:\patterns\exemplos\comportamentais\template_method>ts-node index.ts
For MySQL...
Setting MySQL DB drivers...
Setting credentials for MySQL DB...
Setting connection...
Preparing insert statement...
Setting up data from csv file...
Insert data...
Closing connections...
Destroying connection objects...
For Oracle...
Setting Oracle DB drivers...
Setting credentials for Oracle DB...
Setting connection...
Preparing insert statement...
Setting up data from txt file...
Insert data...
Closing connections...
Destroying connection objects...
```

Observer

Observer: Motivação

- Sports Lobby é um site esportivo fantástico para os amantes do esporte. Eles cobrem quase todos os tipos de esportes e fornecem as últimas notícias, informações, datas agendadas, informações sobre um jogador ou time em particular. Agora, eles planejam fornecer comentários ao vivo ou resultados de partidas como um serviço SMS, mas apenas para seus usuários premium. Seu objetivo é enviar por SMS o placar ao vivo, a situação da partida e eventos importantes após curtos intervalos. Como usuário, você precisa se inscrever no pacote e, quando houver uma partida ao vivo, você receberá um SMS com os comentários ao vivo. O site também oferece uma opção para cancelar a assinatura do pacote sempre que desejar.

Observer: Motivação

- Como desenvolvedor, o Sport Lobby solicitou que você forneça esse novo recurso para eles. Os repórteres do Sport Lobby se sentam na frente da caixa de comentários da partida e atualizam os comentários ao vivo para um objeto de comentário. Como desenvolvedor, seu trabalho é fornecer o comentário aos usuários registrados, buscando-o no objeto de comentário quando estiver disponível. Quando houver uma atualização, o sistema deverá atualizar os usuários inscritos enviando-lhes o SMS.

Observer: Motivação

- Essa situação mostra claramente o mapeamento um-para-muitos entre as mensagens e os usuários, pois pode haver muitos usuários para assinar uma única mensagem. O Design Pattern Observer é mais adequado a essa situação, vamos ver esse padrão e criar o recurso para o Sport Lobby.

Observer : Intenção

- O Padrão do Observador é um tipo de padrão de comportamento que se preocupa com a atribuição de responsabilidades entre objetos. Os padrões de comportamento caracterizam fluxos de controle complexos que são difíceis de seguir no tempo de execução. Eles desviam seu foco do fluxo de controle para permitir que você se concentre apenas na maneira como os objetos são interconectados.

Observer : Intenção

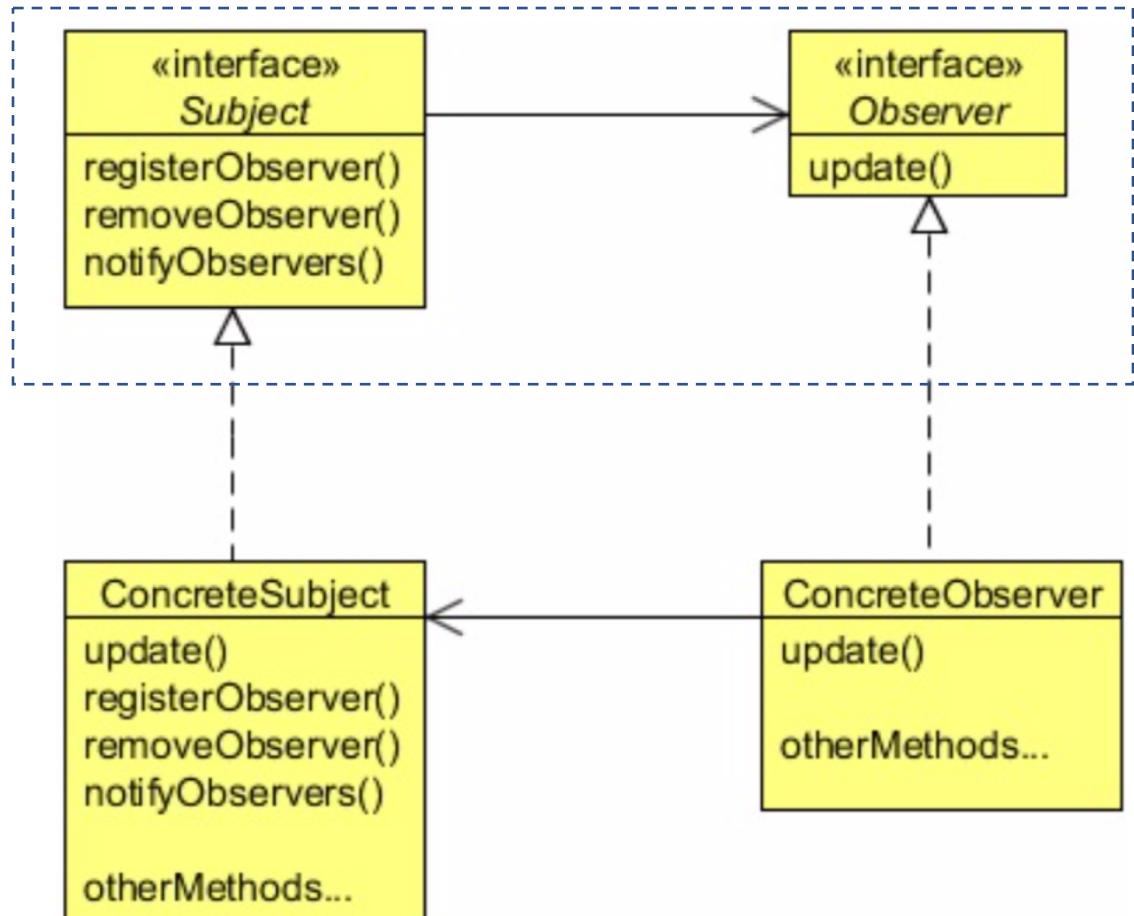
- O Padrão do Observador define uma dependência de um para muitos entre objetos, de modo que, quando um objeto muda de estado, todos os seus dependentes são notificados e atualizados automaticamente. O padrão Observer descreve essas dependências. Os principais objetos desse padrão são sujeito e observador. Um sujeito pode ter qualquer número de observadores dependentes. Todos os observadores são notificados sempre que o sujeito passa por uma mudança de estado. Em resposta, cada observador consultará o assunto para sincronizar seu estado com o estado do assunto.

Observer : Intenção

- A outra maneira de entender o Padrão do Observador é a maneira como o relacionamento entre Publicador e Assinante funciona. Vamos supor, por exemplo, que você se inscreve em sua revista de esportes ou revista de moda favorita. Sempre que uma nova edição é publicada, ela é entregue a você. Se você cancelar sua inscrição quando não desejar mais a revista, ela não será entregue a você. Mas a editora continua trabalhando como antes, pois há outras pessoas que também se inscreveram nessa revista em particular.

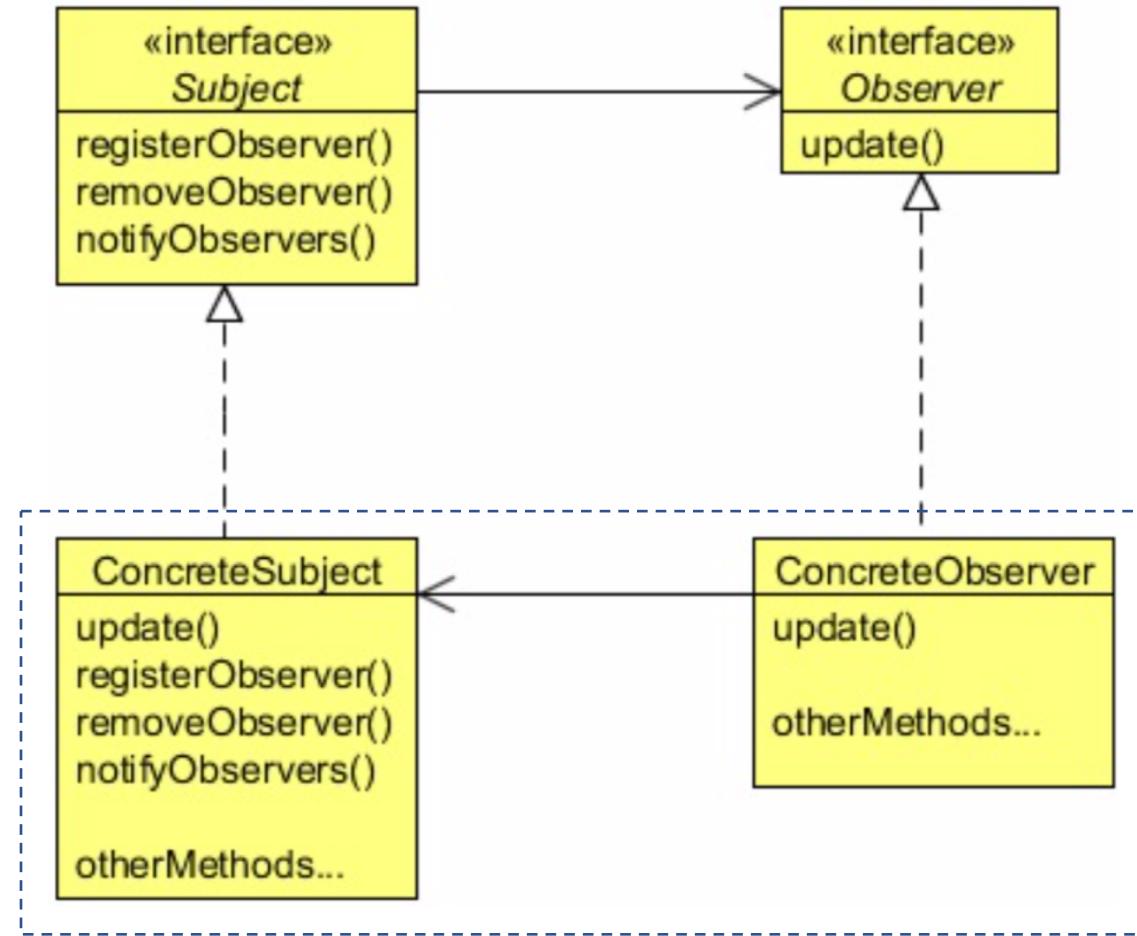
Observer : Estrutura

- Existem quatro participantes:
- **Subject** (Assunto), que é usado para registrar observadores. Os objetos usam essa interface para se registrar como observadores e também para evitar que sejam observadores.
- **Observer**, define uma interface de atualização para objetos que devem ser notificados sobre alterações em um assunto. Todos os observadores precisam implementar a interface Observer. Essa interface possui uma atualização de método (), que é chamada quando o estado do sujeito é alterado.

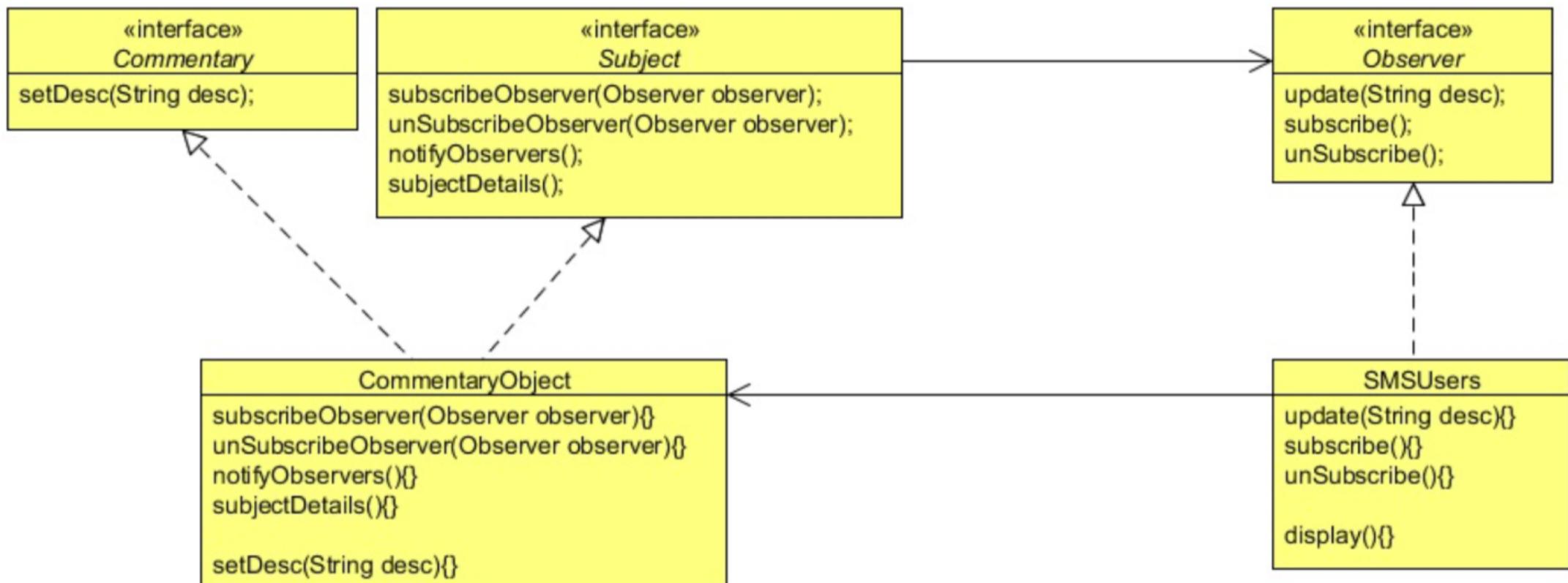


Observer : Estrutura

- **ConcreteSubject**, armazena o estado de interesse dos objetos **ConcreteObserver**. Ele envia uma notificação aos seus observadores quando seu estado muda. Um assunto concreto sempre implementa a interface Assunto. O método `notifyObservers()` é usado para atualizar todos os observadores atuais sempre que o estado muda.
- **ConcreateObserver**, mantém uma referência a um objeto **ConcreteSubject** e implementa a interface Observer. Cada observador registra um assunto concreto para receber atualizações.



Observer : Implementação



Observer : Código Exemplo

Interface Observer

Observer.ts

```
export interface Observer {  
    update(desc: string): void;  
    subscribe(): void;  
    unSubscribe(): void;  
}
```

- `update(String desc)`, o método é chamado pelo sujeito no observador para notificá-lo, quando houver uma alteração no estado do assunto.
- `subscribe()`, método é usado para se inscrever com o assunto.
- `unsubscribe()`, o método é usado para se desinscrever com o assunto.

Observer : Código Exemplo

Interface Subject

Subject.ts

```
import { Observer } from "./Observer";
export interface Subject {
    subscribeObserver(observer: Observer): void;
    unSubscribeObserver(observer: Observer): void;
    notifyObservers(): void;
    subjectDetails(): string;
}
```

- `subscribeObserver`, que é usado para inscrever observadores ou podemos dizer registrar os observadores para que, se houver uma alteração no estado do assunto, todos esses observadores devem ser notificados.
- `unSubscribeObserver`, que é usado para cancelar a inscrição de observadores, para que, se houver uma mudança no estado do assunto, esse observador não inscrito não seja notificado.
- `notifyObservers`, esse método notifica os observadores registrados quando há uma alteração no estado do assunto.

Observer : Código Exemplo

Interface Comentary

```
export interface Commentary{
    setDesc(desc: string): void;
}
```

- A interface acima é usada pelos repórteres para atualizar o comentário ao vivo no objeto de comentário. É uma interface opcional apenas para seguir o princípio do código para interface, não relacionado ao padrão Observador. Você deve aplicar os princípios de poo, juntamente com os padrões de design, sempre que aplicável. A interface contém apenas um método usado para alterar o estado do objeto concreto.

Observer : Código Exemplo

Classe **CommentaryObject** implementa as interfaces **Subject** e **Commentary**

CommentaryObject.ts

```
import { Commentary } from "./Commentary";
import { Observer } from "./Observer";
import { Subject } from "./Subject";

export class CommentaryObject implements Subject, Commentary {
    private observers: Observer[];
    private desc!: string;
    private details: string;

    constructor(observers: Observer[], subjectDetails: string){
        this.observers = observers;
        this.details = subjectDetails;
    }
}
```

Observer : Código Exemplo

continuação da Classe ComentaryObject

```
subscribeObserver(observer: Observer): void {
    this.observers = [...this.observers, observer];
}

unSubscribeObserver(observer: Observer): void {
    this.observers.splice(this.observers.indexOf(observer));
}

notifyObservers(): void {
    console.log('');
    this.observers.forEach(observer => {
        observer.update(this.desc);
    });
}
```

Observer : Código Exemplo

continuação da Classe ComentaryObject

```
subjectDetails(): string {
    return this.details;
}

setDesc(desc: string): void {
    this.desc = desc;
    this.notifyObservers();
}
```

Observer : Código Exemplo

Classe SMSUsers implementa a interface Observer

SMSUsers.ts

```
import { Observer } from "./Observer";
import { Subject } from "./Subject";

export class SMSUsers implements Observer {
    private readonly subject: Subject;
    private desc!: string;
    private userInfo: string;

    constructor(subject: Subject, userInfo: string){
        if (!subject){
            throw new Error("No Publisher found.");
        }
        this.subject = subject;
        this.userInfo = userInfo;
    }

    update(): void {
        console.log(`User ${this.userInfo} received message: ${this.desc}`);
    }
}
```

Observer : Código Exemplo

continuação da Classe SMSUsers

```
update(desc: string): void {
    this.desc = desc;
    this.display();
}

subscribe(): void {
    console.log('Subscribing '+this.userInfo+ ' to '+
    this.subject.subjectDetails()+...');
    this.subject.subscribeObserver(this);
    console.log('Subscribe successfully.');
}
```

Observer : Código Exemplo

continuação da Classe SMSUsers

```
unSubscribe(): void {
    console.log('Unsubscribing '+this.userInfo+ ' to '+
    this.subject.subjectDetails()+...' );
    this.subject.unSubscribe0bserver(this);
    console.log('Unsubscribe successfully.');
}

display(): void {
    console.log('['+this.userInfo+']: '+this.desc);
}
}
```

Observer : Código Exemplo

Script de Teste

index.ts

```
import { CommentaryObject } from "./CommentaryObject";
import { Subject } from "./Subject";
import { Observer } from "./Observer";
import { SMSUsers } from "./SMSUsers";
import { Commentary } from "./Commentary";

let subject: Subject = new CommentaryObject([], 'Soccer Match [2014AUG24]');
let observer: Observer = new SMSUsers(subject, "Adam Warner [New York]");
observer.subscribe();
console.log('');
```

Observer : Código Exemplo

continuação do script de Teste

```
let observer2: Observer = new SMSUsers(subject, "Tim Ronney [London]");  
observer2.subscribe();  
  
let c0bject: Commentary = <Commentary><unknown>subject;  
c0bject.setDesc('Welcome to live Soccer match');  
c0bject.setDesc('Current score 0-0');  
console.log('');  
  
observer2.unSubscribe();  
console.log('');
```

Observer : Código Exemplo

continuação do script de Teste

```
c0bject.setDesc("It's goal!!!!");
c0bject.setDesc('Current score 1-0');
console.log('');

let observer3: Observer = new SMSUsers(subject, "Marrie [Paris]");
observer3.subscribe();
console.log('');

c0bject.setDesc("It's another goal!!!!");
c0bject.setDesc('Half-time score 2-0');
```

Observer : Código Exemplo

Resultados

```
C:\patterns\exemplos\comportamentais\observer>ts-node index.ts
```

```
Subscribing Adam Warner [New York] to Soccer Match [2014AUG24]...
Subscribe successfully.
```

```
Subscribing Tim Ronney [London] to Soccer Match [2014AUG24]...
Subscribe successfully.
```

```
[Adam Warner [New York]]: Welcome to live Soccer match
[Tim Ronney [London]]: Welcome to live Soccer match
```

```
[Adam Warner [New York]]: Current score 0-0
[Tim Ronney [London]]: Current score 0-0
```

Observer : Código Exemplo

Resultados

```
Unsubscribing Tim Ronney [London] to Soccer Match [2014AUG24]...
Unsubscribe successfully.
```

```
[Adam Warner [New York]]: It's goal!!!
```

```
[Adam Warner [New York]]: Current score 1-0
```

```
Subscribing Marrie [Paris] to Soccer Match [2014AUG24]...
Subscribe successfully.
```

```
[Adam Warner [New York]]: It's another goal!!!
```

```
[Marrie [Paris]]: It's another goal!!!
```

```
[Adam Warner [New York]]: Half-time score 2-0
```

```
[Marrie [Paris]]: Half-time score 2-0
```

Chain of Responsibility

Chain of Responsibility: Motivação

- O padrão da Cadeia de Responsabilidade é um padrão de comportamento no qual um grupo de objetos é encadeado em uma sequência e uma responsabilidade (uma solicitação) é fornecida para ser tratada pelo grupo. Se um objeto no grupo puder processar uma solicitação específica, ele fará isso e retornará a resposta correspondente. Caso contrário, ele encaminha a solicitação para o objeto subsequente no grupo.

Chain of Responsibility: Motivação

- Para um cenário da vida real, para entender esse padrão, suponha que você tenha um problema a resolver. Se você é capaz de lidar com isso sozinho, você o fará, caso contrário, dirá ao seu amigo para resolvê-lo. Se ele conseguir resolver, ele fará isso ou também o encaminhará a outro amigo. O problema será encaminhado até que seja resolvido por um de seus amigos ou por todos os seus amigos que tenham visto o problema, mas ninguém seja capaz de resolvê-lo. Nesse caso, o problema permanece sem solução.

Chain of Responsibility: Motivação

- Vamos abordar um cenário da vida real. Sua empresa possui um contrato para fornecer um aplicativo analítico a uma empresa de saúde. O aplicativo informa o usuário sobre o problema de saúde específico, sua história, seu tratamento, medicamentos, entrevista da pessoa que sofre dele etc., tudo o que é necessário saber sobre ele. Para isso, sua empresa recebe uma enorme quantidade de dados. Os dados podem estar em qualquer formato, podem incluir arquivos de texto, arquivos doc, excel, áudio, imagens, vídeos, qualquer coisa que você possa imaginar estaria lá.

Chain of Responsibility: Motivação

- Agora, seu trabalho é salvar esses dados no banco de dados da empresa. Os usuários fornecerão os dados em qualquer formato e você deverá fornecer a eles uma única interface para carregar os dados no banco de dados. O usuário não está interessado, nem mesmo ciente, em saber como está salvando os diferentes dados não estruturados.
- O problema aqui é que você precisa desenvolver diferentes manipuladores para salvar os vários formatos de dados. Por exemplo, um manipulador de salvamento de arquivo de texto não sabe como salvar um arquivo mp3.

Chain of Responsibility: Motivação

- Para resolver esse problema, você pode usar o padrão de design da Cadeia de Responsabilidade. Você pode criar objetos diferentes que processam diferentes formatos de dados e os encadearam. Quando uma solicitação chega a um único objeto, ele verifica se pode processar e manipular o formato de arquivo específico. Se puder, irá processá-lo; caso contrário, ele o encaminhará para o próximo objeto encadeado a ele. Esse padrão de design também desacopla o usuário do objeto que está atendendo a solicitação; o usuário não está ciente de qual objeto está realmente atendendo sua solicitação.

Chain of Responsibility : Intenção

- A intenção desse padrão é evitar o acoplamento do remetente de uma solicitação ao seu receptor, dando a mais de um objeto a chance de lidar com a solicitação. Encadeamos os objetos de recebimento e passamos a solicitação ao longo da cadeia até que um objeto lide com isso.

Chain of Responsibility : Intenção

- Esse padrão tem tudo a ver com conectar objetos em uma cadeia de notificação; quando uma notificação viaja pela cadeia, ela é tratada pelo primeiro objeto configurado para lidar com a notificação específica.

Chain of Responsibility : Intenção

- Quando houver mais de um objeto que pode manipular ou atender a uma solicitação do cliente, o padrão recomenda dar a cada um desses objetos a chance de processar a solicitação em alguma ordem sequencial. Aplicando o padrão nesse caso, cada um desses manipuladores em potencial pode ser organizado na forma de uma cadeia, com cada objeto tendo uma referência ao próximo objeto na cadeia. O primeiro objeto da cadeia recebe a solicitação e decide manipular a solicitação ou passá-la para o próximo objeto da cadeia. A solicitação flui através de todos os objetos da cadeia, um após o outro, até que a solicitação seja manipulada por um dos manipuladores da cadeia ou a solicitação chegue ao fim da cadeia sem ser processada.

Chain of Responsibility : Estrutura

Handler

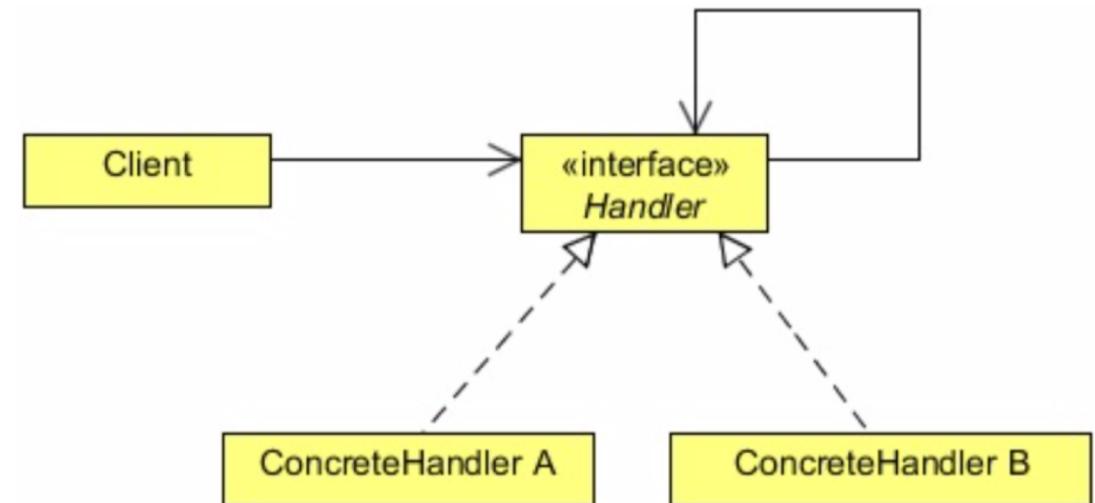
- Define uma interface para lidar com solicitações.
- (Opcionalmente) Implementa o link sucessor.

ConcreteHandler

- Lida com solicitações pelas quais é responsável.
- Pode acessar seu sucessor.
- Se o ConcreteHandler pode lidar com a solicitação, ele o faz; caso contrário, encaminha a solicitação ao seu sucessor.

Client

- Inicia a solicitação para um objeto ConcreteHandler na cadeia.
- Quando um cliente emite uma solicitação, a solicitação se propaga ao longo da cadeia até que um objeto ConcreteHandler assuma a responsabilidade de manipulá-la.



Chain of Responsibility : Código Exemplo

Classe File: é o arquivo que será carregado no sistema (upload)

SMSUsers.ts

```
export class File {  
    private readonly fileName: string;  
    private readonly fileType: string;  
    private readonly filePath: string;  
  
    constructor(fileName: string, fileType: string, filePath: string){  
        this.fileName = fileName;  
        this.fileType = fileType;  
        this.filePath = filePath;  
    }  
}
```

Chain of Responsibility : Código Exemplo

Classe File: é o arquivo que será carregado no sistema (upload)

```
public getFileNome(): string{
    return this.fileName;
}
public getFileType(): string{
    return this.fileType;
}
public getPath(): string{
    return this.filePath;
}
```

Chain of Responsibility : Código Exemplo

Handler:

Handler.ts

```
export interface Handler {  
    setHandler(handler: Handler): void;  
    process(file: File): void;  
    getHanlerName(): string;  
}
```

Chain of Responsibility : Código Exemplo

Manipulador Concreto:
arquivo de texto

TextFileHandler.ts

```
import { Handler } from "./Handler";
import { File } from "./File";

export class TextFileHandler implements Handler {
    private handler!: Handler;
    private handlerName: string;

    constructor(handlerName: string){
        this.handlerName = handlerName;
    }

    public setHandler(handler: Handler): void {
        this.handler = handler;
    }
    public process(file: File): void {
        if(file.getFileType() === 'text'){
            console.log('Process and saving text file... by '+
                this.handlerName);
        } else if (this.handler) {
            console.log(this.handlerName+' forwards request to '+
                this.handler.getHandlerName());
            this.handler.process(file);
        } else {
            console.log('File not supported');
        }
    }
    public getHandlerName(): string {
        return this.handlerName;
    }
}
```

Chain of Responsibility : Código Exemplo

Manipulador Concreto:
arquivo word (doc)

DocFileHandler.ts

```
import { Handler } from "./Handler";
import { File } from "./File";

export class DocFileHandler implements Handler {
    private handler!: Handler;
    private handlerName: string;

    constructor(handlerName: string){
        this.handlerName = handlerName;
    }

    public setHandler(handler: Handler): void {
        this.handler = handler;
    }
    public process(file: File): void {
        if(file.getFileType() === 'doc'){
            console.log('Process and saving doc file... by '+
                this.handlerName);
        } else if (this.handler) {
            console.log(this.handlerName+' forwards request to '+
                this.handler.getHandlerName());
            this.handler.process(file);
        } else {
            console.log('File not supported');
        }
    }
    public getHandlerName(): string {
        return this.handlerName;
    }
}
```

Chain of Responsibility : Código Exemplo

Manipulador Concreto:
arquivo de áudio

```
import { Handler } from "./Handler";
import { File } from "./File";

export class AudioFileHandler implements Handler {
    private handler!: Handler;
    private handlerName: string;

    constructor(handlerName: string){
        this.handlerName = handlerName;
    }

    public setHandler(handler: Handler): void {
        this.handler = handler;
    }
    public process(file: File): void {
        if(file.getType() === 'audio'){
            console.log('Process and saving audio file... by '+
                this.handlerName);
        } else if (this.handler) {
            console.log(this.handlerName+' forwards request to '+
                this.handler.getHandlerName());
            this.handler.process(file);
        } else {
            console.log('File not supported');
        }
    }
    public getHandlerName(): string {
        return this.handlerName;
    }
}
```

Chain of Responsibility : Código Exemplo

Manipulador Concreto:
arquivo Excel

ExcelFileHandler.ts

```
import { Handler } from "./Handler";
import { File } from "./File";

export class ExcelFileHandler implements Handler {
    private handler!: Handler;
    private handlerName: string;

    constructor(handlerName: string){
        this.handlerName = handlerName;
    }

    public setHandler(handler: Handler): void {
        this.handler = handler;
    }

    public process(file: File): void {
        if(file.getFileType() === 'excel'){
            console.log('Process and saving excel file... by '+
                this.handlerName);
        } else if (this.handler) {
            console.log(this.handlerName+' forwards request to '+
                this.handler.getHandlerName());
            this.handler.process(file);
        } else {
            console.log('File not supported');
        }
    }

    public getHandlerName(): string {
        return this.handlerName;
    }
}
```

Chain of Responsibility : Código Exemplo

Manipulador Concreto:
arquivo de imagem

ImageFileHandler.ts

```
import { Handler } from "./Handler";
import { File } from "./File";

export class ImageFileHandler implements Handler {
    private handler!: Handler;
    private handlerName: string;

    constructor(handlerName: string){
        this.handlerName = handlerName;
    }

    public setHandler(handler: Handler): void {
        this.handler = handler;
    }

    public process(file: File): void {
        if(file.getFileType() === 'image'){
            console.log('Process and saving image file... by '+
                this.handlerName);
        } else if (this.handler) {
            console.log(this.handlerName+' forwards request to '+
                this.handler.getHandlerName());
            this.handler.process(file);
        } else {
            console.log('File not supported');
        }
    }

    public getHandlerName(): string {
        return this.handlerName;
    }
}
```

Chain of Responsibility : Código Exemplo

Manipulador Concreto:
arquivo de vídeo

VideoFileHandler.ts

```
import { Handler } from "./Handler";
import { File } from "./File";

export class VideoFileHandler implements Handler {
    private handler!: Handler;
    private handlerName: string;

    constructor(handlerName: string){
        this.handlerName = handlerName;
    }

    public setHandler(handler: Handler): void {
        this.handler = handler;
    }
    public process(file: File): void {
        if(file.getType() === 'video'){
            console.log('Process and saving video file... by '+
                this.handlerName);
        } else if (this.handler) {
            console.log(this.handlerName+' forwards request to '+
                this.handler.getHandlerName());
            this.handler.process(file);
        } else {
            console.log('File not supported');
        }
    }
    public getHandlerName(): string {
        return this.handlerName;
    }
}
```

Chain of Responsibility : Código Exemplo

Classe de Teste

index.ts

```
import { AudioFileHandler } from "./AudioFileHandler";
import { DocFileHandler } from "./DocFileHandler";
import { ExcelFileHandler } from "./ExcelFileHandler";
import { File } from "./File"
import { Handler } from "./Handler";
import { ImageFileHandler } from "./ImageFileHandler";
import { TextFileHandler } from "./TextFileHandler";
import { VideoFileHandler } from "./VideoFileHandler";

let file: File;
let textHandler: Handler = new TextFileHandler("Text Handler");
let docHandler: Handler = new DocFileHandler("Doc Handler");
let audioHandler: Handler = new AudioFileHandler("Audio Handler");
let excelHandler: Handler = new ExcelFileHandler("Excel Handler");
let videoHandler: Handler = new VideoFileHandler("Video Handler");
let imageHandler: Handler = new ImageFileHandler("Image Handler");

textHandler.setHandler(docHandler);
docHandler.setHandler(excelHandler);
excelHandler.setHandler(audioHandler);
audioHandler.setHandler(videoHandler);
videoHandler.setHandler(imageHandler);

file = new File('Abc.mp3', 'audio', 'C:');
textHandler.process(file);
console.log('');
```

Chain of Responsibility : Código Exemplo

```
file = new File('Abc.jpg', 'video', 'C:');  
textHandler.process(file);  
console.log('');  
  
file = new File('Abc.doc', 'doc', 'C:');  
textHandler.process(file);  
console.log('');  
  
file = new File('Abc.bat', 'bat', 'C:');  
textHandler.process(file);
```

Classe de Teste

Chain of Responsibility : Código Exemplo

Resultados

```
C:\patterns\exemplos\comportamentais\chain_of_responsibility> node index.ts
```

```
Text Handler forwards request to Doc Handler  
Doc Handler forwards request to Excel Handler  
Excel Handler forwards request to Audio Handler  
Process and saving audio file... by Audio Handler
```

```
Text Handler forwards request to Doc Handler  
Doc Handler forwards request to Excel Handler  
Excel Handler forwards request to Audio Handler  
Audio Handler forwards request to Video Handler  
Process and saving video file... by Video Handler
```

```
Text Handler forwards request to Doc Handler  
Process and saving doc file... by Doc Handler
```

```
Text Handler forwards request to Doc Handler  
Doc Handler forwards request to Excel Handler  
Excel Handler forwards request to Audio Handler  
Audio Handler forwards request to Video Handler  
Video Handler forwards request to Image Handler  
File not supported
```