

Design Patterns Criacionais

Factory Method, Abstract Factory e Singleton

2022

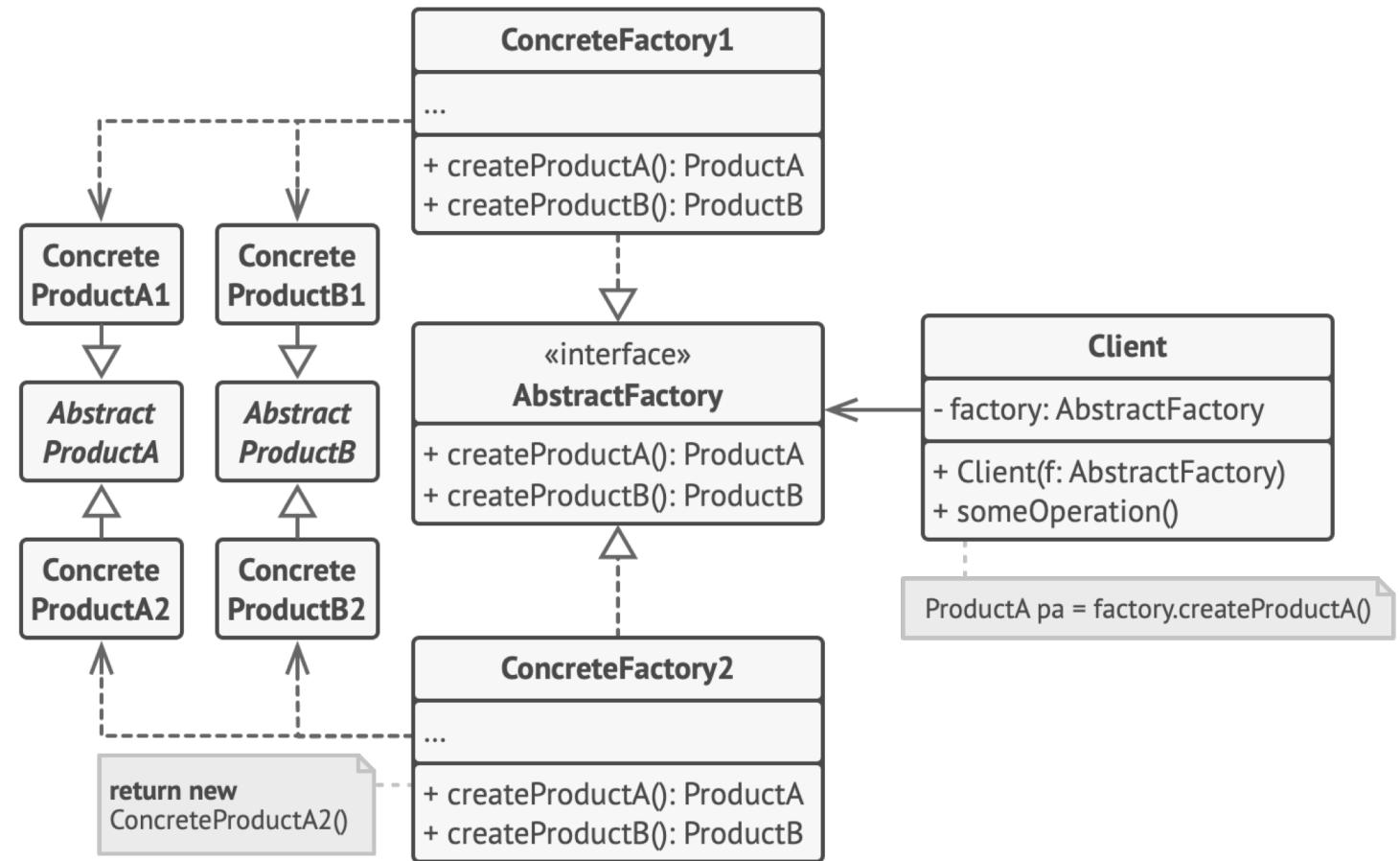
Prof. Sergio Bonato

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.

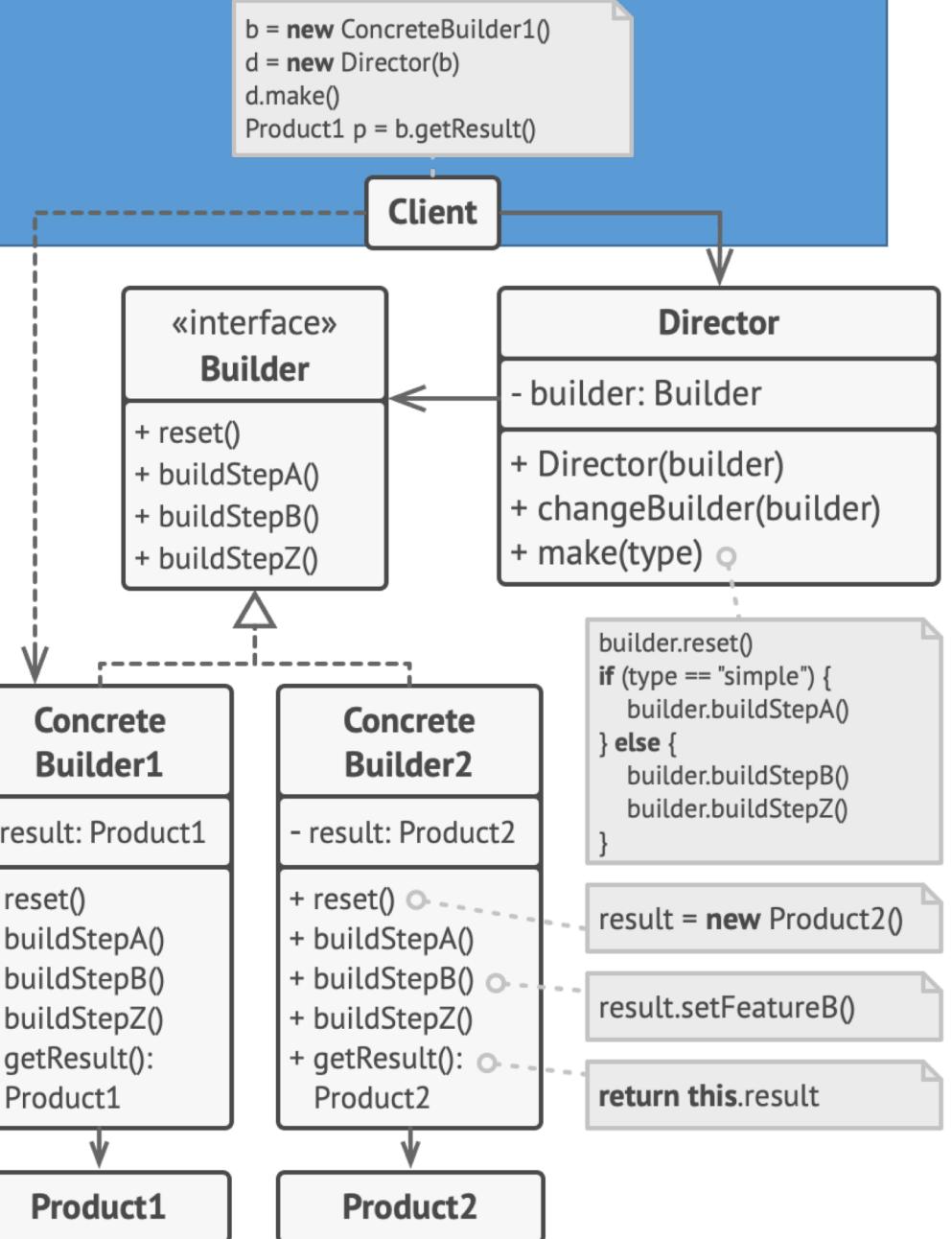
Abstract Factory

O Abstract Factory é um padrão de projeto criacional que permite que você produza famílias de objetos relacionados sem ter que especificar suas classes concretas.



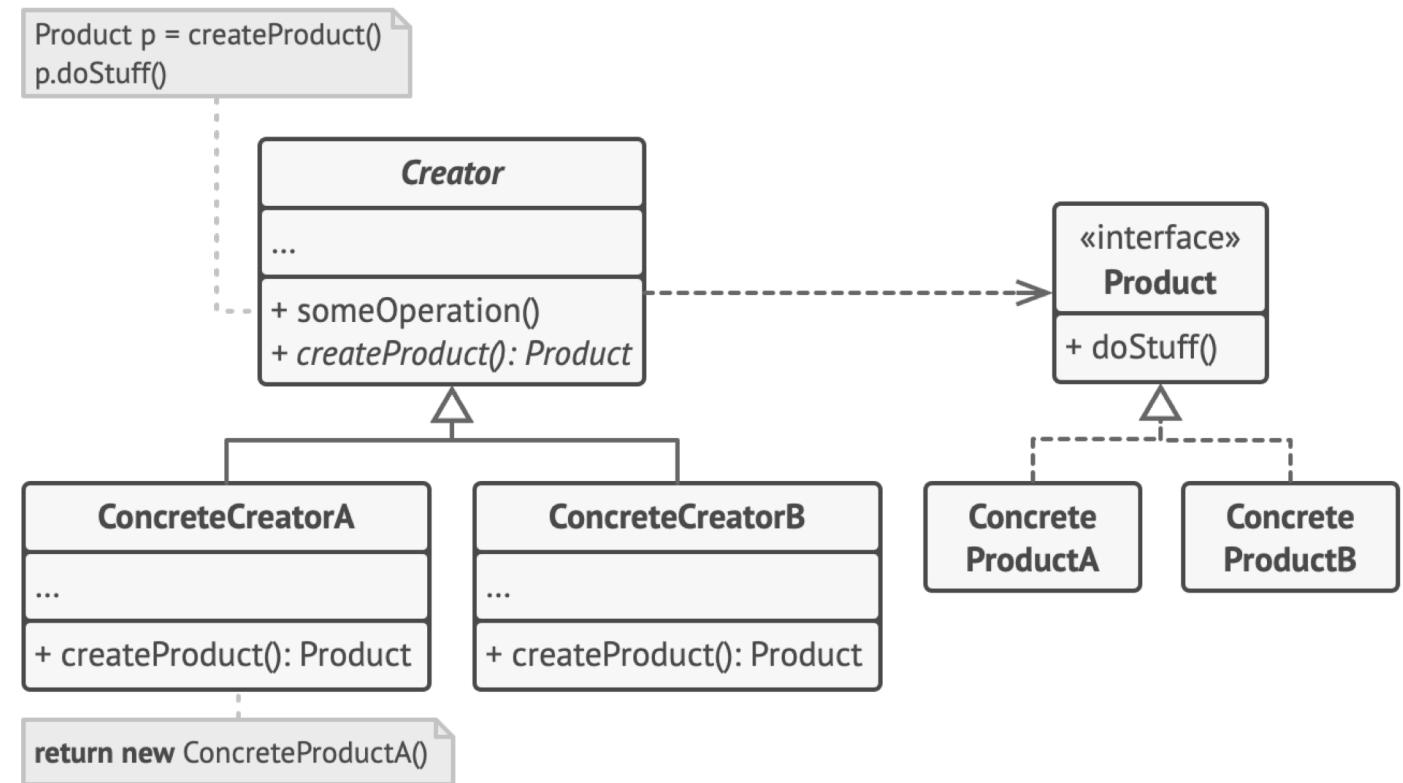
Builder

O Builder é um padrão de projeto criacional que permite a você construir objetos complexos passo a passo. O padrão permite que você produza diferentes tipos e representações de um objeto usando o mesmo código de construção.



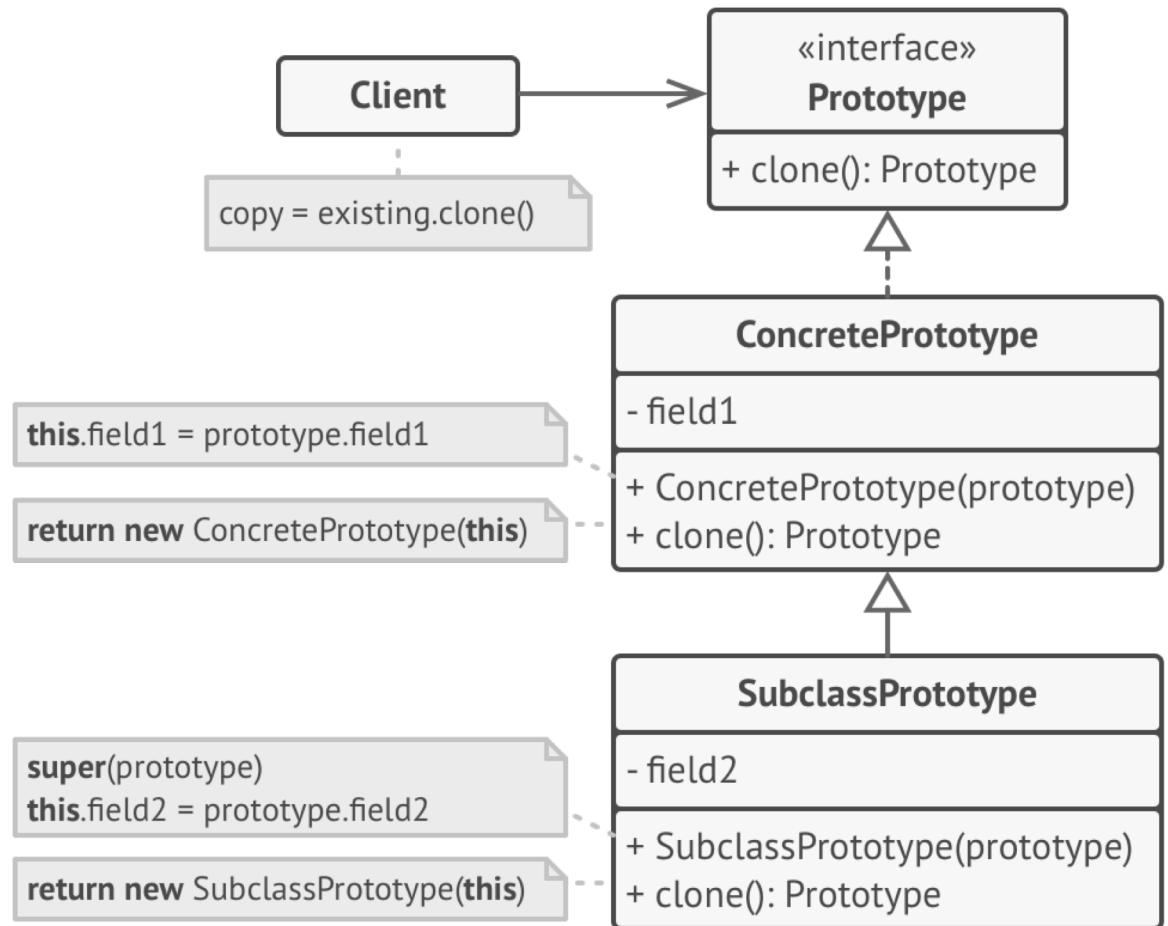
Factory Method

O Factory Method é um padrão criacional de projeto que fornece uma interface para criar objetos em uma superclasse, mas permite que as subclasses alterem o tipo de objetos que serão criados.



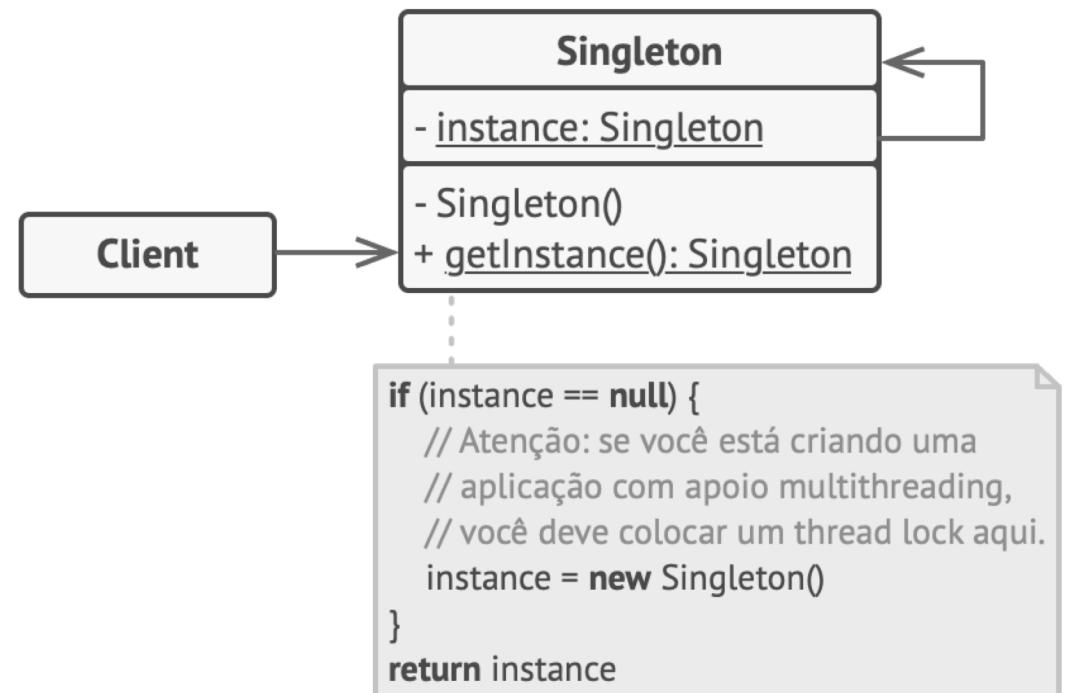
Prototype

O Prototype é um padrão de projeto criacional que permite copiar objetos existentes sem fazer seu código ficar dependente de suas classes.



Singleton

O Singleton é um padrão de projeto criacional que permite a você garantir que uma classe tenha apenas uma instância, enquanto provê um ponto de acesso global para essa instância.



Detalhes de Implementação

- Factory Method
- Abstract Factory
- Singleton

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>

Factory Method

Factory Method: Motivação

- No mundo moderno de hoje, todos estão usando software para facilitar seus trabalhos. Recentemente, uma empresa de produtos mudou a maneira como costumava receber pedidos de seus clientes. A empresa está agora procurando usar um aplicativo para receber pedidos deles. Eles recebem pedidos, erros em pedidos, feedback para o pedido anterior e respostas ao pedido em um formato XML. A empresa pediu que você desenvolvesse um aplicativo para analisar o XML e exibir o resultado para eles.
- O principal desafio para você é analisar um XML e exibir seu conteúdo para o usuário. Existem diferentes formatos XML, dependendo dos diferentes tipos de mensagens que a empresa recebe de seus clientes. Como, por exemplo, um XML de tipo de pedido tem conjuntos diferentes de tags xml em comparação com a resposta ou erro XML. Mas o trabalho principal é o mesmo; isto é, para exibir ao usuário a mensagem sendo transportada nesses XMLs.

Factory Method: Motivação

- Embora o trabalho principal seja o mesmo, o objeto que seria usado varia de acordo com o tipo de XML que o aplicativo obtém do usuário. Portanto, um objeto de aplicativo pode saber apenas que precisa acessar uma classe de dentro da hierarquia de classes (hierarquia de analisadores diferentes), mas não sabe exatamente qual classe entre o conjunto de subclasses da classe pai deve ser selecionada.
- Nesse caso, é melhor fornecer uma fábrica, ou seja, uma fábrica para criar analisadores e, em tempo de execução, um analisador é instanciado para executar a tarefa, de acordo com o tipo de XML que o aplicativo recebe do usuário.

Factory Method: Motivação

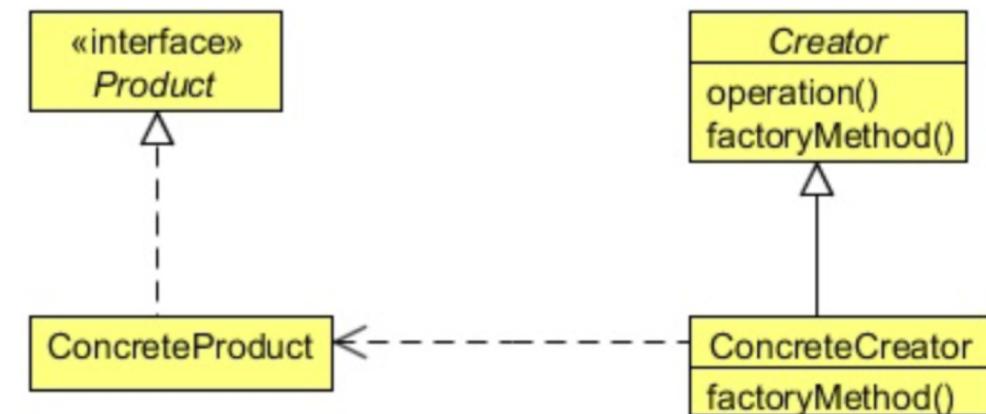
- O Factory Method Pattern, adequado para essa situação, define uma interface para criar um objeto, mas permite que as subclasses decidam qual classe instanciar. O Factory Method permite que uma classe adie a instanciação para subclasses.
- Vamos ver mais detalhes sobre o Factory Method Pattern e, em seguida, vamos usá-lo para implementar o analisador XML para o aplicativo.

Factory Method: Intenção

- O Factory Method Pattern nos dá uma maneira de encapsular as instanciações dos tipos concretos. O padrão Factory Method encapsula a funcionalidade necessária para selecionar e instanciar uma classe apropriada, dentro de um método designado, chamado de método de fábrica. O Factory Method seleciona uma classe apropriada de uma hierarquia de classes com base no contexto da aplicação e outros fatores de influência. Em seguida, instancia a classe selecionada e a retorna como uma instância do tipo de classe pai.
- A vantagem dessa abordagem é que os objetos do aplicativo podem usar o método factory para obter acesso à instância de classe apropriada. Isso elimina a necessidade de um objeto de aplicativo para lidar com os diversos critérios de seleção de classe.

Factory Method: Estrutura

- Product
 - Define a interface de objetos criada pelo método de fábrica.
- ConcreteProduct
 - Implementa a interface do produto.
- O Criador
 - Declara o método factory, que retorna um objeto do tipo Product. O criador também pode definir uma implementação padrão do método de fábrica que retorna um objeto ConcreteProduct padrão.
 - Pode chamar o método factory para criar um objeto Product.
- ConcreteCreator
 - Substitui o método de fábrica para retornar uma instância de um ConcreteProduct.



Factory Method: Código Exemplo

Product: a interface de um parser XML

`XMLParser.ts`

```
export interface XMLParser {  
  parse(): string;  
}
```

Factory Method: Código Exemplo

ConcreteProduct: 4 diferentes parsers XML

Parser de Erros

ErrorXMLParser.ts

```
import { XMLParser } from './XMLParser'

export class ErrorXMLParser implements XMLParser {
    parse():string {
        console.log("Parsing error XML...");
        return "Error XML Message";
    }
}
```

Factory Method: Código Exemplo

ConcreteProduct: 4 diferentes parsers XML

Parser de Feedback

FeedbackXMLParser.ts

```
import { XMLParser } from './XMLParser'

export class FeedbackXMLParser implements XMLParser {
    parse():string {
        console.log("Parsing feedback XML...");
        return "Feedback XML Message";
    }
}
```

Factory Method: Código Exemplo

ConcreteProduct: 4 diferentes parsers XML

Parser de Pedido

OrderXMLParser.ts

```
import { XMLParser } from './XMLParser'

export class OrderXMLParser implements XMLParser {
    parse():string {
        console.log("Parsing order XML...");
        return "Order XML Message";
    }
}
```

Factory Method: Código Exemplo

ConcreteProduct: 4 diferentes parsers XML

Parser de Resposta

ResponseXMLParser.ts

```
import { XMLParser } from './XMLParser'

export class ResponseXMLParser implements XMLParser {
    parse():string {
        console.log("Parsing response XML...");
        return "Response XML Message";
    }
}
```

Factory Method: Código Exemplo

Creator: classe que mostra a mensagem parseada

DisplayService.ts

```
import { XMLParser } from './XMLParser'

export abstract class DisplayService{
    public display(): void {
        const parser: XMLParser = this.getParser();
        const msg: string = parser.parse();
        console.log(msg);
    }

    public abstract getParser(): XMLParser;
}
```

Factory Method: Código Exemplo

ConcreteCreator: 4 diferentes mostradores de mensagem XML

Mostrador de Erro

ErrorXMLDisplayService.ts

```
import { DisplayService } from './DisplayService';
import { ErrorXMLParser } from './ErrorXMLParser'
import { XMLParser } from './XMLParser'

export class ErrorXMLDisplayService extends DisplayService {
    public getParser(): XMLParser{
        return new ErrorXMLParser();
    }
}
```

Factory Method: Código Exemplo

ConcreteCreator: 4 diferentes mostradores de mensagem XML

Mostrador de Feedback

FeedbackXMLDisplayService.ts

```
import { DisplayService } from './DisplayService';
import { FeedbackXMLParser } from './FeedbackXMLParser'
import { XMLParser } from './XMLParser'

export class FeedbackXMLDisplayService extends DisplayService {
    public getParser(): XMLParser{
        return new FeedbackXMLParser();
    }
}
```

Factory Method: Código Exemplo

ConcreteCreator: 4 diferentes mostradores de mensagem XML

Mostrador de Pedido

OrderXMLDisplayService.ts

```
import { DisplayService } from './DisplayService';
import { OrderXMLParser } from './OrderXMLParser'
import { XMLParser } from './XMLParser'

export class OrderXMLDisplayService extends DisplayService {
    public getParser(): XMLParser{
        return new OrderXMLParser();
    }
}
```

Factory Method: Código Exemplo

ConcreteCreator: 4 diferentes mostradores de mensagem XML

Mostrador de Resposta

ResponseXMLDisplayService.ts

```
import { DisplayService } from './DisplayService';
import { ResponseXMLParser } from './ResponseXMLParser'
import { XMLParser } from './XMLParser'

export class ResponseXMLDisplayService extends DisplayService {
    public getParser(): XMLParser {
        return new ResponseXMLParser();
    }
}
```

Factory Method: Código Exemplo

Testando

index.ts

```
import { DisplayService } from "./DisplayService";
import { ErrorXMLDisplayService } from "./ErrorXMLDisplayService";
import { FeedbackXMLDisplayService } from "./FeedbackXMLDisplayService";
import { OrderXMLDisplayService } from "./OrderXMLDisplayService";
import { ResponseXMLDisplayService } from "./ResponseXMLDisplayService";

let service: DisplayService = new FeedbackXMLDisplayService();
service.display();
service = new ErrorXMLDisplayService();
service.display();
service = new OrderXMLDisplayService();
service.display();
service = new ResponseXMLDisplayService();
service.display();
```

Factory Method: Código Exemplo

Resultado

```
C:\patterns\exemplos\criacionais\factory_method> ts-node index.ts
Parsing feedback XML...
Feedback XML Message
Parsing error XML...
Error XML Message
Parsing order XML...
Order XML Message
Parsing response XML...
Response XML Message
```

Abstract Factory

Abstract Factory: Motivação

- No pattern anterior, desenvolvemos um aplicativo para uma empresa de produtos para analisar XMLs e exibir resultados para eles. Fizemos isso criando diferentes analisadores para os diferentes tipos de comunicação entre a empresa e seus clientes. Usamos o pattern Factory Method para resolver seu problema.
- O aplicativo está funcionando bem para eles. Mas agora os clientes não querem seguir as regras XML específicas da empresa. Os clientes desejam usar suas próprias regras XML para se comunicar com a empresa do produto. Isso significa que, para cada cliente, a empresa deve ter analisadores XML específicos do cliente. Por exemplo, para o cliente NY, deve haver quatro tipos específicos de analisadores XML, ou seja, NYErrorXMLParser, NYFeedbackXML, NYOrderXMLParser, NYResponseXMLParser e quatro analisadores diferentes para o cliente TW.

Abstract Factory: Motivação

- A empresa pediu que você altere o aplicativo de acordo com o novo requisito. Para desenvolver o aplicativo analisador, usamos o pattern Factory Method, no qual o objeto exato a ser usado é decidido pelas subclasses de acordo com o tipo de analisador. Agora, para implementar este novo requisito, usaremos uma fábrica de fábricas, ou seja, um Abstract Factory.
- Desta vez, precisamos de analisadores de acordo com XMLs específicos do cliente, portanto, criaremos diferentes fábricas para diferentes clientes, o que fornecerá o XML específico do cliente para análise. Faremos isso criando uma Fábrica Abstrata e, em seguida, implementando a fábrica para fornecer fábrica XML específica do cliente. Em seguida, usaremos essa fábrica para obter o objeto do analisador XML específico do cliente desejado.
- Abstract Factory é o padrão de design de nossa escolha e antes de implementá-lo para resolver nosso problema, nos permite saber mais sobre isso.

Abstract Factory: Intenção

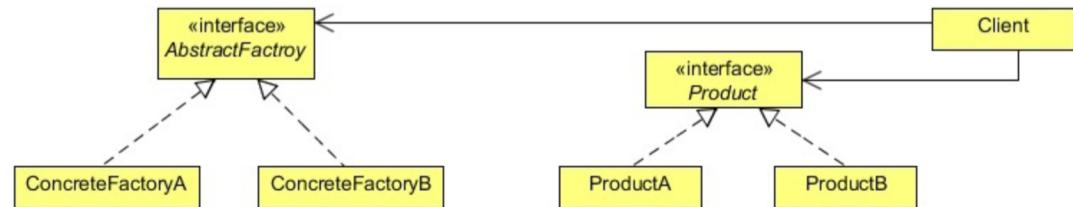
- O Abstract Factory (também chamado de Kit) é um padrão de projeto que fornece uma interface para criar famílias de objetos relacionados ou dependentes sem especificar suas classes concretas. O padrão Abstract Factory leva o conceito do Factory Method Pattern para o próximo nível. Uma fábrica abstrata é uma classe que fornece uma interface para produzir uma família de objetos. Em Java, ele pode ser implementado usando uma interface ou uma classe abstrata.

Abstract Factory: Intenção

- O padrão Fábrica Abstrata é útil quando um objeto cliente deseja criar uma instância de um conjunto de classes dependentes relacionadas sem precisar saber qual classe concreta específica deve ser instanciada. Diferentes fábricas de concreto implementam a interface abstrata de fábrica. Os objetos clientes fazem uso dessas fábricas concretas para criar objetos e, portanto, não precisam saber qual classe concreta é realmente instanciada.
- A fábrica abstrata é útil para conectar um grupo diferente de objetos para alterar o comportamento do sistema. Para cada grupo ou família, é implementada uma fábrica de concreto que gerencia a criação dos objetos e as interdependências e requisitos de consistência entre eles. Cada fábrica de concreto implementa a interface da fábrica abstrata

Abstract Factory: Estrutura

- **AbstractFactory**
 - Declara uma interface para operações que criam objetos abstratos do produto.
- **ConcreteFactory**
 - Implementa as operações para criar objetos de produtos concretos.
- **AbstractProduct**
 - Declara uma interface para um tipo de objeto de produto.
- **ConcreteProduct**
 - Define um objeto de produto a ser criado pela fábrica de concreto correspondente.
 - Implementa a interface do AbstractProduct.
- **Cliente**
 - Usa apenas interfaces declaradas pelas classes AbstractFactory e AbstractProduct



Abstract Factory: Código Exemplo

Product

XMLParser.ts

```
export interface XMLParser {
    parse(): string;
}
```

Abstract Factory : Código Exemplo

ConcreteProduct: 4 diferentes parsers XML

Parser de Error

NYErrorXMLParser.ts

Fábrica Concreta NY

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

export class NYErrorXMLParser implements XMLParser{
    public parse(): string{
        console.log("NY Parsing error XML...");
        return "NY Error XML Message"
    }
}
```

Abstract Factory : Código Exemplo

ConcreteProduct: 4 diferentes parsers XML

Parser de Feedback

NYFeedbackXMLParser.ts

Fábrica Concreta NY

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

export class NYFeedbackXMLParser implements XMLParser{
    public parse(): string{
        console.log("NY Parsing Feedback XML...");
        return "NY Feedback XML Message"
    }
}
```

Abstract Factory : Código Exemplo

ConcreteProduct: 4 diferentes parsers XML

Parser de Pedido

NYOrderXMLParser.ts

Fábrica Concreta NY

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

export class NYOrderXMLParser implements XMLParser{
    public parse(): string{
        console.log("NY Parsing Order XML...");
        return "NY Order XML Message"
    }
}
```

Abstract Factory : Código Exemplo

ConcreteProduct: 4 diferentes parsers XML

Parser de Resposta

NYResponseXMLParser.ts

Fábrica Concreta NY

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

export class NYResponseXMLParser implements XMLParser{
    public parse(): string{
        console.log("NY Parsing Response XML..."); 
        return "NY Response XML Message"
    }
}
```

Abstract Factory : Código Exemplo

ConcreteProduct: 4 diferentes parsers XML

Parser de Error

TWErrorXMLParser.ts

Fábrica Concreta TW

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

export class TWErrorXMLParser implements XMLParser{
    public parse(): string{
        console.log("TW Parsing error XML...");
        return "Tw Error XML Message"
    }
}
```

Abstract Factory : Código Exemplo

ConcreteProduct: 4 diferentes parsers XML

Parser de Feedback

TWFeedbackXMLParser.ts

Fábrica Concreta TW

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

export class TWFeedbackXMLParser implements XMLParser{
    public parse(): string{
        console.log("TW Parsing Feedback XML..."); 
        return "TW Feedback XML Message"
    }
}
```

Abstract Factory : Código Exemplo

ConcreteProduct: 4 diferentes parsers XML

Parser de Pedido

TWOrderXMLParser.ts

Fábrica Concreta TW

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

export class TWOrderXMLParser implements XMLParser {
    public parse(): string {
        console.log("TW Parsing Order XML...");
        return "Tw Order XML Message"
    }
}
```

Abstract Factory : Código Exemplo

ConcreteProduct: 4 diferentes parsers XML

Parser de Resposta

TWResponseXMLParser.ts

Fábrica Concreta TW

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

export class TWResponseXMLParser implements XMLParser{
    public parse(): string{
        console.log("TW Parsing Response XML...");
        return "Tw Response XML Message"
    }
}
```

Abstract Factory : Código Exemplo

AbstractFactory: a interface de um parser XML

AbstractParserFactory.ts

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

export interface AbstractParserFactory {
    getParserInstance(parserType: string): XMLParser;
}
```

Abstract Factory : Código Exemplo

NYParserFactory.ts

ConcreteFactory: 2 diferentes fábricas de parsers XML

Parser NY

```
import { AbstractParserFactory } from "./AbstractParserFactory";
import { NYErrorXMLParser } from "./NYErrorXMLParser";
import { NYFeedbackXMLParser } from "./NYFeedbackXMLParser";
import { NYOrderXMLParser } from "./NYOrderXMLParser";
import { NYResponseXMLParser } from "./NYResponseXMLParser";
import { XMLParser } from "./XMLParser";

export class NYParserFactory implements AbstractParserFactory{
    public getParserInstance(parserType: string): XMLParser | null {
        switch(parserType){
            case 'NYERROR': return new NYErrorXMLParser();
            case 'NYFEEDBACK': return new NYFeedbackXMLParser();
            case 'NYORDER': return new NYOrderXMLParser();
            case 'NYRESPONSE': return new NYResponseXMLParser();
        }
        return null;
    }
}
```

Abstract Factory : Código Exemplo

TWParserFactory.ts

ConcreteFactory: 2 diferentes fábricas de parsers XML

Parser TW

```
import { AbstractParserFactory } from "./AbstractParserFactory";
import { TWErrorXMLParser } from "./TWErrorXMLParser";
import { TWFeedbackXMLParser } from "./TWFeedbackXMLParser";
import { TWOrderXMLParser } from "./TWOrderXMLParser";
import { TWResponseXMLParser } from "./TWResponseXMLParser";
import { XMLParser } from "./XMLParser";

export class TWParserFactory implements AbstractParserFactory{
    public getParserInstance(parserType: string): XMLParser | null {
        switch(parserType){
            case 'TWERROR': return new TWErrorXMLParser();
            case 'TWFEEDBACK': return new TWFeedbackXMLParser();
            case 'TWORDER': return new TWOrderXMLParser();
            case 'TWRESPONSE': return new TWResponseXMLParser();
        }
        return null;
    }
}
```

Abstract Factory : Código Exemplo

FactoryProducer: opcional – evita o acoplamento entre o código cliente e as fábricas

ParserFactoryProducer.ts

```
import { AbstractParserFactory } from "./AbstractParserFactory";
import { NYParserFactory } from "./NYParserFactory";
import { TWParserFactory } from "./TWParserFactory";

export class ParserFactoryProducer {
    private constructor(){
        throw new Error('A classe não pode ser instanciada');
    }
    public static getFactory(factoryType: string): AbstractParserFactory | null{
        switch(factoryType){
            case "NYFactory": return new NYParserFactory();
            case "TWFactory": return new TWParserFactory();
        }
        return null;
    }
}
```

Abstract Factory : Código Exemplo

Testando

index.ts

```
import { AbstractParserFactory } from "./AbstractParserFactory";
import { ParserFactoryProducer } from "./ParserFactoryProducer";
import { XMLParser } from "./XMLParser";

let parserFactory: AbstractParserFactory | null =
ParserFactoryProducer.getFactory("NYFactory");
let parser: XMLParser | null = parserFactory ?
parserFactory.getParserInstance("NYORDER") : null;
let msg: string = parser ? parser.parse() : "";
console.log(msg);

console.log('*****\n');

parserFactory = ParserFactoryProducer.getFactory("TWFactory");
parser = parserFactory ? parserFactory.getParserInstance("TWFEEDBACK") : null;
msg = parser ? parser.parse() : "";
console.log(msg);
```

Abstract Factory : Código Exemplo

Resultado

```
C:\patterns\exemplos\criacionais\abstract_factory> ts-node index.ts
NY Parsing Order XML...
NY Order XML Message
```

```
*****
```

```
TW Parsing Feedback XML...
TW Feedback XML Message
```

Singleton

Singleton: Motivação

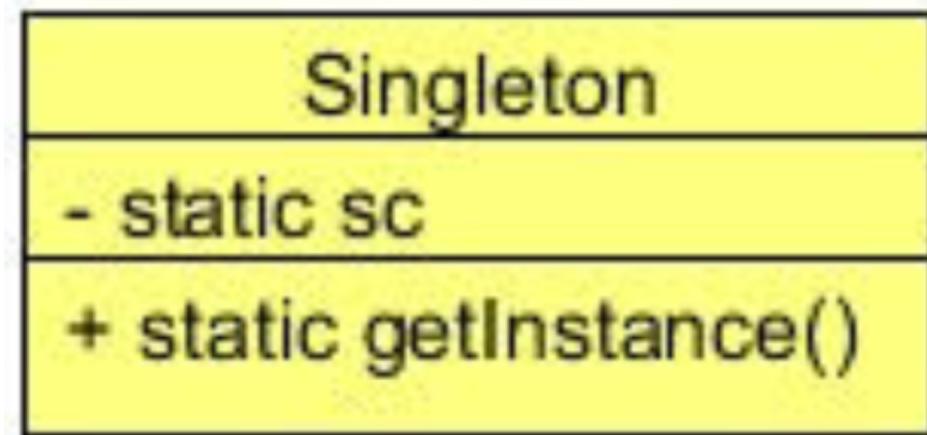
- Às vezes, é importante que algumas classes tenham exatamente uma instância. Há muitos objetos que só precisamos de uma instância deles e, se instanciamos mais de um, encontraremos todos os tipos de problemas, como comportamento incorreto do programa, uso excessivo de recursos ou resultados inconsistentes.
- Você pode exigir apenas um objeto de uma classe, por exemplo, quando você cria o contexto de um aplicativo, ou um pool gerenciável de thread, configurações de registro, um driver para conectar ao console de entrada ou saída, etc. Mais de um objeto de esse tipo claramente causará inconsistência no seu programa.

Singleton: Intenção

- O Padrão Singleton garante que uma classe tenha apenas uma instância e forneça um ponto global de acesso a ela.

Singleton: Estrutura

- Embora o Singleton seja o mais simples em termos de diagrama de classes, porque há apenas uma única classe, sua implementação é um pouco mais complicada.



SingletonEager: Código Exemplo

Construtor e variável públicos

SingletonEagerSimples.ts

Este é o jeito mais fácil de se construir um singleton: usando uma variável de classe (static) pública e instanciando-a logo de início (por isso é eager, isto é, ansioso). Mas este código tem um problema. Um objeto pode, em vez de usar a variável pública sc, instanciar sua própria versão da classe SingletonEager e aí o pattern perdeu o sentido. Por isso usamos construtor private, como na próxima versão.

```
export class SingletonEagerSimples {  
  public static sc: SingletonEagerSimples = new SingletonEagerSimples();  
}
```

SingletonEager: Código Exemplo

Construtor private e variável pública

SingletonEagerMelhor.ts

Com o construtor private outros objetos não podem instanciar seu próprio singleton. Entretanto, não é uma boa prática dar acesso direto às variáveis da classe. Veja a versão final do singleton eager no próximo slide.

```
export class SingletonEagerMelhor {  
  public static sc: SingletonEagerMelhor = new SingletonEagerMelhor();  
  private constructor(){}
}
```

SingletonEager: Código Exemplo

Construtor e variável private

SingletonEager.ts

Versão Final

Além do construtor private, agora a variável estática sc também é private. Foi criado um método getInstance que retorna a instância do singleton.

[SingletonEagerMelhor](#)

```
export class SingletonEager {  
    private static sc: SingletonEager = new SingletonEager();  
  
    private constructor(){}
  
  
    public static getInstance(): SingletonEager{  
        return SingletonEager.sc;
    }
}
```

SingletonLazy: Código Exemplo

Similar a versão final do SingletonEager, mas não instancia o objeto logo no início, mas espera que alguém peça uma instância. Aí ele instancia. Depois, para quem pedir, retorna o objeto que já está instanciado.

SingletonLazy.ts

```
export class SingletonLazy {
    private static sc: SingletonLazy;

    private constructor(){}

    public static getInstance(): SingletonLazy{
        if (!SingletonLazy.sc){
            SingletonLazy.sc = new SingletonLazy();
        }
        return SingletonLazy.sc;
    }
}
```

Bibliografia

- [JOSHI 16] JOSHI, R.; Java Design Patterns: reusable solutions to common problems. Java Code Geeks. 2016.
- [GAMMA 00] GAMMA et al.; Padrões de Projeto: soluções reutilizáveis de software orientado a objetos. 1ª Edição. Bookman. 2000.
- [FREEMAN 04] FREEMAN et al.; Head First Design Patterns. 1ª Edição. O'Reilly. 2004.
- [SHVETS 22] SHVETS, A.; Mergulho nos Padrões de Projeto; v2022-1.22, Refactorin Guru, 2022.