refactoring.guru

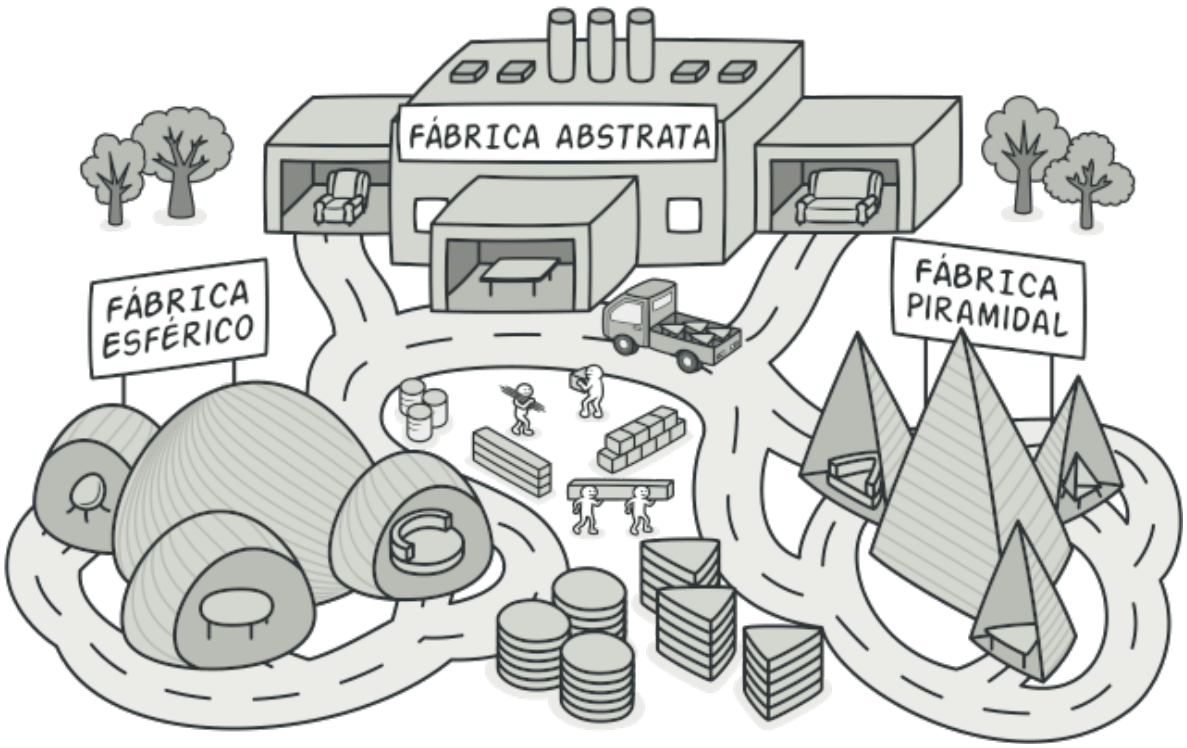
Abstract Factory

14–19 minutes

Também conhecido como: Fábrica abstrata

Propósito

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.



Problema

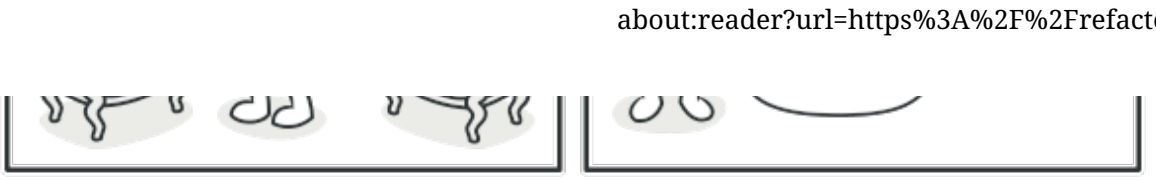
Imagine que você está criando um simulador de loja de mobílias.

Seu código consiste de classes que representam:

1. Uma família de produtos relacionados, como: `Cadeira` + `Sofá` + `MesaDeCentro`.
2. Várias variantes dessa família. Por exemplo, produtos `Cadeira` + `Sofá` + `MesaDeCentro` estão disponíveis nessas variantes: `Moderno`, `Vitoriano`, `ArtDeco`.

Famílias de produtos e suas variantes.

Você precisa de um jeito de criar objetos de mobília individuais para que eles combinem com outros objetos da mesma família. Os clientes ficam muito bravos quando recebem mobília que não combina.

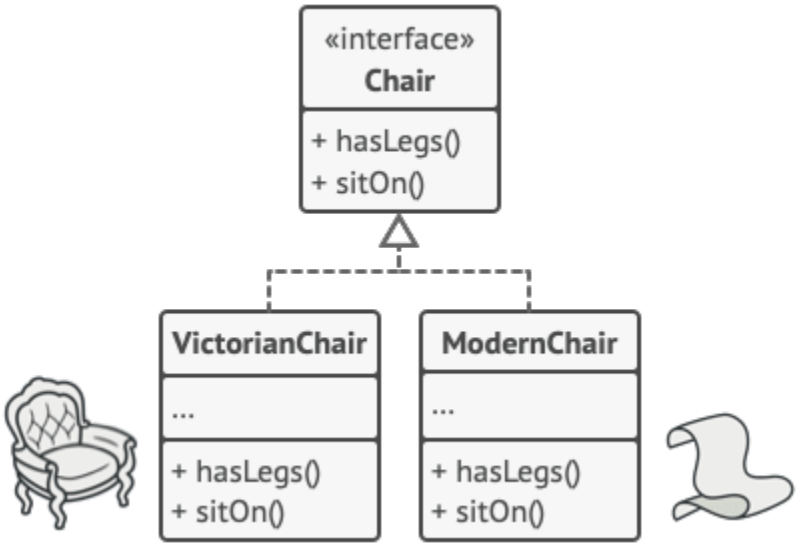


Um sofá no estilo Moderno não combina com cadeiras de estilo Vitoriano

E ainda, você não quer mudar o código existente quando adiciona novos produtos ou famílias de produtos ao programa. Os vendedores de mobílias atualizam seus catálogos com frequência e você não vai querer mudar o código base cada vez que isso acontece.

Solução

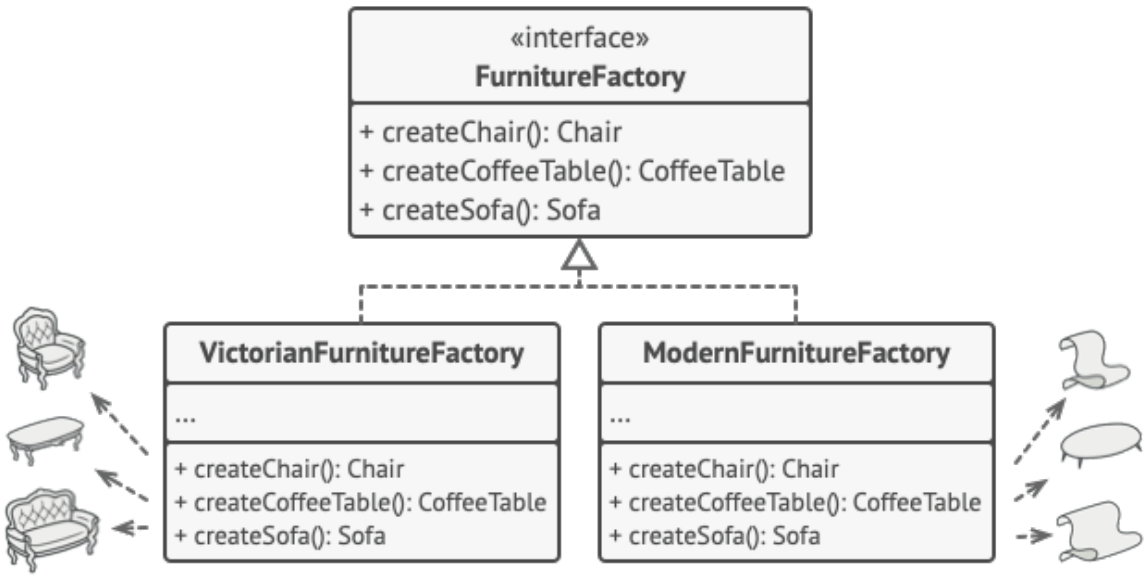
A primeira coisa que o padrão Abstract Factory sugere é declarar explicitamente interfaces para cada produto distinto da família de produtos (ex: cadeira, sofá ou mesa de centro). Então você pode fazer todas as variantes dos produtos seguirem essas interfaces. Por exemplo, todas as variantes de cadeira podem implementar a interface `Cadeira`; todas as variantes de mesa de centro podem implementar a interface `MesaDeCentro`, e assim por diante.



Todas as variantes do mesmo objeto podem ser movidas para

uma mesma hierarquia de classe.

O próximo passo é declarar a *fábrica abstrata*—uma interface com uma lista de métodos de criação para todos os produtos que fazem parte da família de produtos (por exemplo, `criarCadeira`, `criarSofá` e `criarMesaDeCentro`). Esses métodos devem retornar tipos **abstratos** de produtos representados pelas interfaces que extraímos previamente: `Cadeira`, `Sofá`, `MesaDeCentro` e assim por diante.



Cada fábrica concreta corresponde a uma variante de produto específica.

Agora, e o que fazer sobre as variantes de produtos? Para cada variante de uma família de produtos nós criamos uma classe fábrica separada baseada na interface `FábricaAbstrata`. Uma fábrica é uma classe que retorna produtos de um tipo em particular. Por exemplo, a classe `FábricaMobíliaModerna` só pode criar objetos `CadeiraModerna`, `SofáModerno`, e `MesaDeCentroModerna`.

O código cliente tem que funcionar com ambos as fábricas e produtos via suas respectivas interfaces abstratas. Isso permite

que você mude o tipo de uma fábrica que passou para o código cliente, bem como a variante do produto que o código cliente recebeu, sem quebrar o código cliente atual.

O cliente não deveria se importar com a classe concreta da fábrica com a qual está trabalhando.

Digamos que o cliente quer que uma fábrica produza uma cadeira. O cliente não precisa estar ciente da classe fábrica, e nem se importa que tipo de cadeira ele receberá. Seja ela um modelo Moderno ou no estilo Vitoriano, o cliente precisa tratar todas as cadeiras da mesma maneira, usando a interface abstrata `Cadeira`. Com essa abordagem, a única coisa que o cliente sabe sobre a cadeira é que ela implementa o método `sentar` de alguma maneira. E também, seja qual for a variante da cadeira retornada, ela sempre irá combinar com o tipo de sofá ou mesa de centro produzido pelo mesmo objeto fábrica.

Há mais uma coisa a se clarificar: se o cliente está exposto apenas às interfaces abstratas, o que realmente cria os objetos fábrica então? Geralmente, o programa cria um objeto fábrica concreto no estágio de inicialização. Antes disso, o programa

deve selecionar o tipo de fábrica dependendo da configuração ou definições de ambiente.

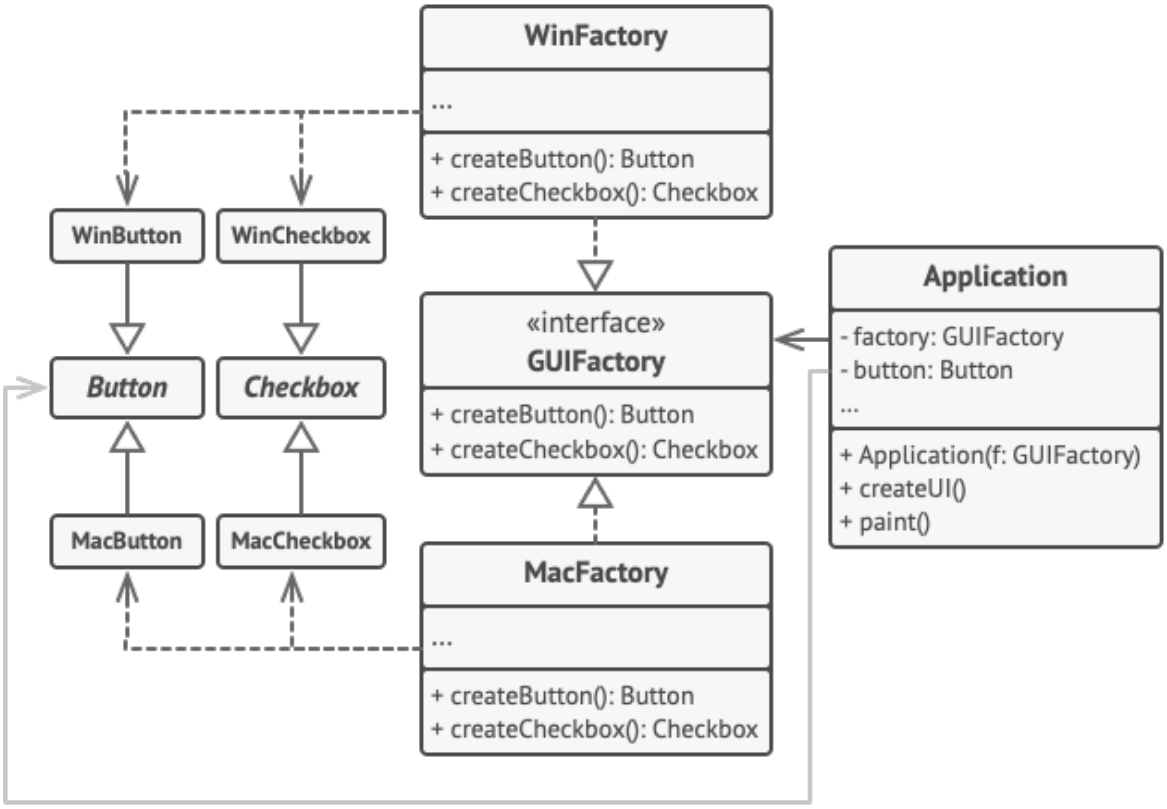
Estrutura

1. **Produtos Abstratos** declaram interfaces para um conjunto de produtos distintos mas relacionados que fazem parte de uma família de produtos.
2. **Produtos Concretos** são várias implementações de produtos abstratos, agrupados por variantes. Cada produto abstrato (cadeira/sofá) deve ser implementado em todas as variantes dadas (Vitoriano/Moderno).
3. A interface **Fábrica Abstrata** declara um conjunto de métodos para criação de cada um dos produtos abstratos.
4. **Fábricas Concretas** implementam métodos de criação fábrica abstratos. Cada fábrica concreta corresponde a uma variante específica de produtos e cria apenas aquelas variantes de produto.
5. Embora fábricas concretas instanciam produtos concretos, assinaturas dos seus métodos de criação devem retornar produtos *abstratos* correspondentes. Dessa forma o código cliente que usa uma fábrica não fica ligada a variante específica do produto que ele pegou de uma fábrica. O **Cliente** pode trabalhar com qualquer variante de produto/fábrica concreto, desde que ele se comunique com seus objetos via interfaces abstratas.

Pseudocódigo

Este exemplo ilustra como o padrão **Abstract Factory** pode ser

usado para criar elementos UI multiplataforma sem ter que ligar o código do cliente às classes UI concretas, enquanto mantém todos os elementos criados consistentes com um sistema operacional escolhido.



Exemplo das classes UI multiplataforma.

É esperado que os mesmos elementos UI de um aplicativo multiplataforma se comportem de forma semelhante, mas que se pareçam um pouco diferentes nos diferentes sistemas operacionais. Além disso, é seu trabalho garantir que os elementos UI coincidam com o estilo do sistema operacional atual. Você não vai querer que seu programa renderize controles macOS quando é executado no Windows.

A interface da fábrica abstrata declara um conjunto de métodos de criação que o código cliente pode usar para produzir diferentes tipos de elementos de UI que coincidam com o SO particular.

Fábricas concretas correspondem a sistemas operacionais específicos e criam os elementos de UI que corresponde com aquele SO em particular.

Funciona assim: quando a aplicação inicia, ela checa o tipo de sistema operacional que está sendo utilizado. A aplicação usa essa informação para criar um objeto fábrica de uma classe que corresponde com o sistema operacional. O resto do código usa essa fábrica para criar elementos UI. Isso previne que elementos errados sejam criados.

Com essa abordagem, o código cliente não depende de classes concretas de fábricas e elementos UI desde que ele trabalhe com esses objetos através de suas interfaces abstratas. Isso também permite que o código do cliente suporte outras fábricas ou elementos UI que você possa adicionar no futuro.

Como resultado, você não precisa modificar o código do cliente cada vez que adicionar uma variação de elementos de UI em sua aplicação. Você só precisa criar uma nova classe fábrica que produza esses elementos e modificar de forma sutil o código de inicialização da aplicação de forma que ele selecione aquela classe quando apropriado.

```
// A interface fábrica abstrata declara um conjunto de métodos
// que retorna diferentes produtos abstratos. Estes produtos são
// chamados uma família e estão relacionados por um tema ou
// conceito de alto nível. Produtos de uma família são
// geralmente capazes de colaborar entre si. Uma família de
// produtos pode ter várias variantes, mas os produtos de uma
// variante são incompatíveis com os produtos de outro variante.
interface GUIFactory is
```

```
    method createButton():Button
    method createCheckbox():Checkbox


// As fábricas concretas produzem uma família de produtos que
// pertencem a uma única variante. A fábrica garante que os
// produtos resultantes sejam compatíveis. Assinaturas dos
// métodos fabrica retornam um produto abstrato, enquanto que
// dentro do método um produto concreto é instanciado.
class WinFactory implements GUIFactory is
    method createButton():Button is
        return new WinButton()
    method createCheckbox():Checkbox is
        return new WinCheckbox()

// Cada fábrica concreta tem uma variante de produto
// correspondente.
class MacFactory implements GUIFactory is
    method createButton():Button is
        return new MacButton()
    method createCheckbox():Checkbox is
        return new MacCheckbox()


// Cada produto distinto de uma família de produtos deve ter uma
// interface base. Todas as variantes do produto devem
// implementar essa interface.
interface Button is
    method paint()
```

```
// Produtos concretos são criados por fábricas concretas
// correspondentes.
class WinButton implements Button is
    method paint() is
        // Renderiza um botão no estilo Windows.

class MacButton implements Button is
    method paint() is
        // Renderiza um botão no estilo macOS.

// Aqui está a interface base de outro produto. Todos os
// produtos podem interagir entre si, mas a interação apropriada
// só é possível entre produtos da mesma variante concreta.
interface Checkbox is
    method paint()

class WinCheckbox implements Checkbox is
    method paint() is
        // Renderiza uma caixa de seleção estilo Windows.

class MacCheckbox implements Checkbox is
    method paint() is
        // Renderiza uma caixa de seleção no estilo macOS.


// O código cliente trabalha com fábricas e produtos apenas
// através de tipos abstratos: GUIFactory, Button e Checkbox.
// Isso permite que você passe qualquer subclasse fábrica ou de
// produto para o código cliente sem quebrá-lo.
class Application is
```

```
    private field factory: GUIFactory
    private field button: Button
    constructor Application(factory: GUIFactory) is
        this.factory = factory
    method createUI() is
        this.button = factory.createButton()
    method paint() is
        button.paint()


// A aplicação seleciona o tipo de fábrica dependendo da atual
// configuração do ambiente e cria o widget no tempo de execução
// (geralmente no estágio de inicialização).
class ApplicationConfigurator is
    method main() is
        config = readApplicationConfigFile()

        if (config.OS == "Windows") then
            factory = new WinFactory()
        else if (config.OS == "Mac") then
            factory = new MacFactory()
        else
            throw new Exception("Error! Unknown operating system.")

        Application app = new Application(factory)
```

Aplicabilidade

Use o Abstract Factory quando seu código precisa trabalhar com diversas famílias de produtos relacionados, mas que você não

quer depender de classes concretas daqueles produtos-eles podem ser desconhecidos de antemão ou você simplesmente quer permitir uma futura escalabilidade.

O Abstract Factory fornece a você uma interface para a criação de objetos de cada classe das famílias de produtos. Desde que seu código crie objetos a partir dessa interface, você não precisará se preocupar em criar uma variante errada de um produto que não coincida com produtos já criados por sua aplicação.

Considere implementar o Abstract Factory quando você tem uma classe com um conjunto de métodos fábrica que desfoquem sua responsabilidade principal.

Em um programa bem desenvolvido *cada classe é responsável por apenas uma coisa*. Quando uma classe lida com múltiplos tipos de produto, pode valer a pena extrair seus métodos fábrica em uma classe fábrica solitária ou uma implementação plena do Abstract Factory.

Como implementar

1. Mapeie uma matriz de tipos de produtos distintos versus as variantes desses produtos.
2. Declare interfaces de produto abstratas para todos os tipos de produto. Então, faça todas as classes concretas de produtos implementar essas interfaces.
3. Declare a interface da fábrica abstrata com um conjuntos de métodos de criação para todos os produtos abstratos.
4. Implemente um conjunto de classes fábricas concretas, uma para cada variante de produto.

5. Crie um código de inicialização da fábrica em algum lugar da aplicação. Ele deve instanciar uma das classes fábrica concretas, dependendo da configuração da aplicação ou do ambiente atual. Passe esse objeto fábrica para todas as classes que constroem produtos.
6. Escaneie o código e encontre todas as chamadas diretas para construtores de produtos. Substitua-as por chamadas para o método de criação apropriado no objeto fábrica.

Prós e contras

- Você pode ter certeza que os produtos que você obtém de uma fábrica são compatíveis entre si.
- Você evita um vínculo forte entre produtos concretos e o código cliente.
- *Princípio de responsabilidade única*. Você pode extrair o código de criação do produto para um lugar, fazendo o código ser de fácil manutenção.
- *Princípio aberto/fechado*. Você pode introduzir novas variantes de produtos sem quebrar o código cliente existente.
- O código pode tornar-se mais complicado do que deveria ser, uma vez que muitas novas interfaces e classes são introduzidas junto com o padrão.

Relações com outros padrões

- Muitos projetos começam usando o Factory Method (menos complicado e mais customizável através de subclasses) e evoluem para o Abstract Factory, Prototype, ou Builder (mais

flexíveis, mas mais complicados).

- O Builder foca em construir objetos complexos passo a passo. O Abstract Factory se especializa em criar famílias de objetos relacionados. O *Abstract Factory* retorna o produto imediatamente, enquanto que o *Builder* permite que você execute algumas etapas de construção antes de buscar o produto.
- Classes Abstract Factory são quase sempre baseadas em um conjunto de métodos fábrica, mas você também pode usar o Prototype para compor métodos dessas classes.
- O Abstract Factory pode servir como uma alternativa para o Facade quando você precisa apenas esconder do código cliente a forma com que são criados os objetos do subsistema.
- Você pode usar o Abstract Factory junto com o Bridge. Esse pareamento é útil quando algumas abstrações definidas pelo *Bridge* só podem trabalhar com implementações específicas. Neste caso, o *Abstract Factory* pode encapsular essas relações e esconder a complexidade do código cliente.
- As Fábricas Abstratas, Construtores, e Protótipos podem todos ser implementados como Singletons.