I. O que é um padrão de projeto?

Padrões de projeto são soluções típicas para problemas comuns em projeto de software. Eles são como as plantas de uma obra pré-fabricadas que você pode customizar para resolver um problema de projeto recorrente em seu código.

Você não pode apenas encontrar um padrão e copiá-lo para dentro do seu programa, como você faz com funções e bibliotecas que encontra por aí. O padrão não é um pedaço de código específico, mas um conceito geral para resolver um problema em particular. Você pode seguir os detalhes do padrão e implementar uma solução que se adeque às realidades do seu próprio programa.

Os padrões são frequentemente confundidos com algoritmos, porque ambos os conceitos descrevem soluções típicas para alguns problemas conhecidos. Enquanto um algoritmo sempre define um conjunto claro de ações para atingir uma meta, um padrão é mais uma descrição de alto nível de uma solução. O código do mesmo padrão aplicado para dois programas distintos pode ser bem diferente.

Uma analogia a um algoritmo é que ele seria uma receita de comida: ambos têm etapas claras para chegar a um objetivo. Por outro lado, um padrão é mais como uma planta de obras: você pode ver o resultado e suas funcionalidades, mas a ordem exata de implementação depende de você.

Do que consiste em padrão?

A maioria dos padrões são descritos formalmente para que as pessoas possam reproduzi-los em diferentes contextos. Aqui estão algumas seções que são geralmente presentes em uma descrição de um padrão:

- O **Propósito** do padrão descreve brevemente o problema e a solução.
- A **Motivação** explica a fundo o problema e a solução que o padrão torna possível.
- As **Estruturas** de classes mostram cada parte do padrão e como se relacionam.
- **Exemplos de código** em uma das linguagens de programação populares tornam mais fácil compreender a ideia por trás do padrão.

Alguns catálogos de padrão listam outros detalhes úteis, tais como a aplicabilidade do padrão, etapas de implementação, e relações com outros padrões.

- Os padrões de projeto são um kit de ferramentas para **soluções tentadas e testadas** para problemas comuns em projeto de software. Mesmo que você nunca tenha encontrado esses problemas, saber sobre os padrões é ainda muito útil porque eles ensinam como resolver vários problemas usando princípios de projeto orientado a objetos.
- Os padrões de projeto definem uma linguagem comum que você e seus colegas podem usar para se comunicar mais eficientemente. Você pode dizer, “Oh, é só usar um *Singleton* para isso,” e todo mundo vai entender a ideia por trás da sua sugestão. Não é preciso explicar o que um *singleton* é se você conhece o padrão e seu nome.

- Os **padrões criacionais** fornecem mecanismos de criação de objetos que aumentam a flexibilidade e a reutilização de código.
- Os **padrões estruturais** explicam como montar objetos e classes em estruturas maiores, enquanto ainda mantém as estruturas flexíveis e eficientes.
- Os **padrões comportamentais** cuidam da comunicação eficiente e da assinalação de responsabilidades entre objetos.

inbhhhhhhhhhhhhhhhhhhhhhhh

Exemplo conceitual:
Este exemplo ilustra a estrutura do padrão de projeto ***Singleton***. Ele se concentra em responder a estas perguntas:

- De quais classes ele consiste?
- Quais papéis essas classes desempenham?
- De que maneira os elementos do padrão estão relacionados?

index.ts: Exemplo conceitual

```
/*The Singleton class defines an `instance` getter, that lets
 * clients access the unique singleton instance.*/
class Singleton {
    static #instance: Singleton;

    /*The Singleton's constructor should always be private to
     * prevent direct construction calls with the `new` operator.
     */
    private constructor() { }

    /*The static getter that controls access to the singleton
     * instance. This implementation allows you to extend the
     * Singleton class while keeping just one instance of each
     * subclass around.*/
    public static get instance(): Singleton {
        if (!Singleton.#instance) {
            Singleton.#instance = new Singleton();
        }
        return Singleton.#instance;
    }
    /*Finally, any singleton can define some business logic, which
     *can be executed on its instance. */
    public someBusinessLogic() {
        // ...
    }
}
/**
 * The client code.*/
function clientCode() {
    const s1 = Singleton.instance;
    const s2 = Singleton.instance;

    if (s1 === s2) {
        console.log(
            'Singleton works, both variables contain the same
                instance.'
        );
    } else {
        console.log('Singleton failed, variables contain
                different instances.');
```

```
        }
    }

    clientCode();
```

II. Factory Method

Também conhecido como**:** Método fábrica, Construtor virtual

Propósito:

O *Factory* 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.

Problema:

Imagine que você está criando uma aplicação de gerenciamento de logística. A primeira versão da sua aplicação pode lidar apenas com o transporte de caminhões, portanto a maior parte do seu código fica dentro da classe Caminhão.

Depois de um tempo, sua aplicação se torna bastante popular. Todos os dias você recebe dezenas de solicitações de empresas de transporte marítimo para incorporar a logística marítima na aplicação.

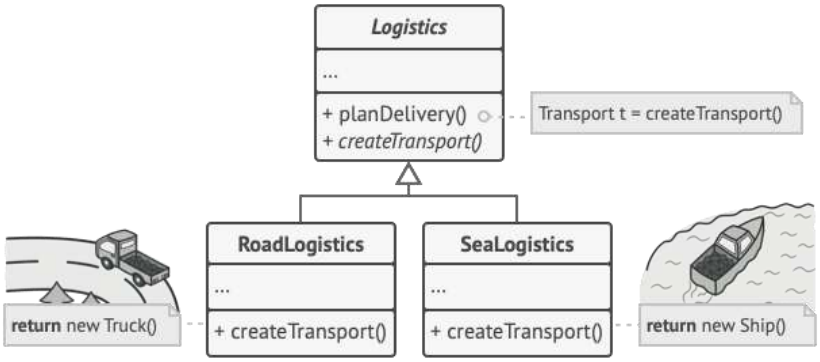
Adicionar uma nova classe ao programa não é tão simples se o restante do código já estiver acoplado às classes existentes.

Atualmente, a maior parte do seu código é acoplada à classe Caminhão. Adicionar Navio à aplicação exigiria alterações em toda a base de código. Além disso, se mais tarde você decidir adicionar outro tipo de transporte à aplicação, provavelmente precisará fazer todas essas alterações novamente.

Como resultado, você terá um código bastante sujo, repleto de condicionais que alteram o comportamento da aplicação, dependendo da classe de objetos de transporte.

Solução:

O padrão *Factory* sugere que você substitua chamadas diretas de construção de objetos (usando o operador new) por chamadas para um método *fábrica* especial. Não se preocupe: os objetos ainda são criados através do operador new, mas esse está sendo chamado de dentro do método fábrica. Objetos retornados por um método fábrica geralmente são chamados de *produtos*.

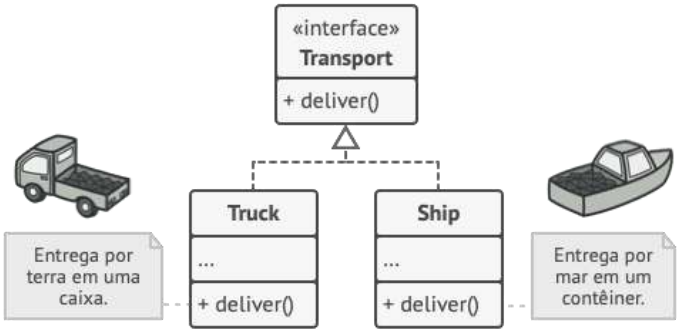


As subclasses podem alterar a classe de objetos retornados pelo método fábrica.

À primeira vista, essa mudança pode parecer sem sentido: apenas mudamos a chamada do construtor de uma parte do programa para outra. No entanto, considere o seguinte: agora

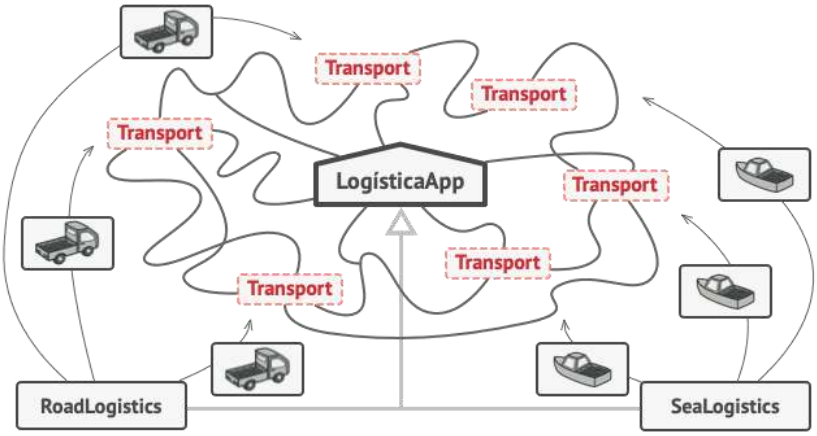
você pode sobrescrever o método fábrica em uma subclasse e alterar a classe de produtos que estão sendo criados pelo método.

Porém, há uma pequena limitação: as subclasses só podem retornar tipos diferentes de produtos se esses produtos tiverem uma classe ou interface base em comum. Além disso, o método fábrica na classe base deve ter seu tipo de retorno declarado como essa interface.



Todos os produtos devem seguir a mesma interface.

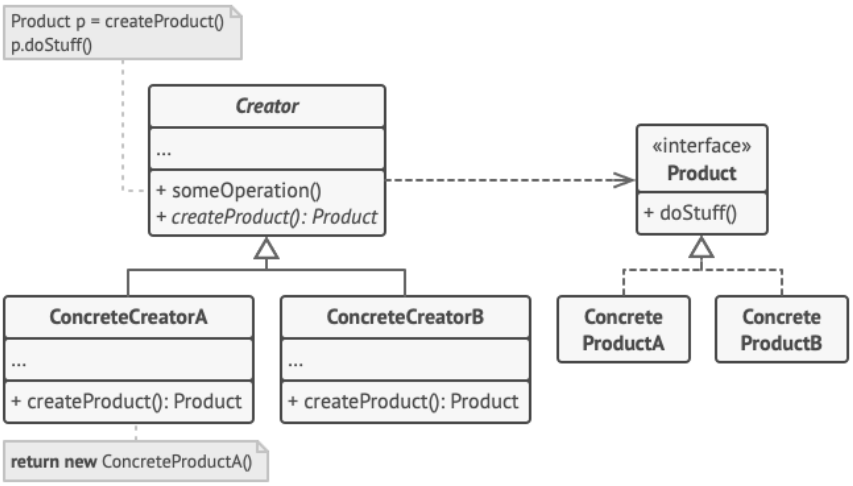
Por exemplo, ambas as classes Caminhão e Navio devem implementar a interface Transporte, que declara um método chamado entregar. Cada classe implementa esse método de maneira diferente: caminhões entregam carga por terra, navios entregam carga por mar. O método fábrica na classe LogísticaViária retorna objetos de caminhão, enquanto o método fábrica na classe LogísticaMarítima retorna navios.



Desde que todas as classes de produtos implementem uma interface comum, você pode passar seus objetos para o código cliente sem quebrá-lo.

O código que usa o método fábrica (geralmente chamado de código *cliente*) não vê diferença entre os produtos reais retornados por várias subclasses. O cliente trata todos os produtos como um Transporte abstrato. O cliente sabe que todos os objetos de transporte devem ter o método entregar, mas como exatamente ele funciona não é importante para o cliente.

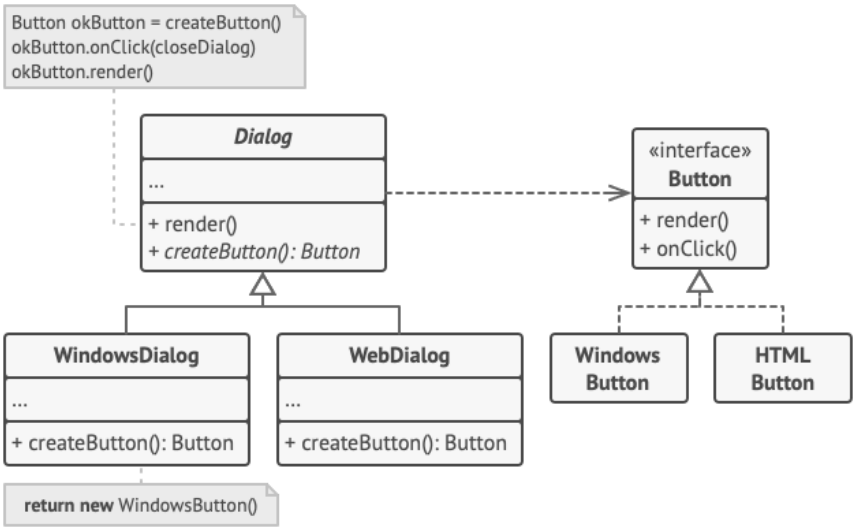
Estrutura:



1. O Produto declara a interface, que é comum a todos os objetos que podem ser produzidos pelo criador e suas subclasses.
2. Produtos Concretos são implementações diferentes da interface do produto.
3. A classe Criador declara o método fábrica que retorna novos objetos produto. É importante que o tipo de retorno desse método corresponda à interface do produto.
   Você pode declarar o método fábrica como abstrato para forçar todas as subclasses a implementar suas próprias versões do método. Como alternativa, o método fábrica base pode retornar algum tipo de produto padrão.
   Observe que, apesar do nome, a criação de produtos não é a principal responsabilidade do criador. Normalmente, a classe criadora já possui alguma lógica de negócio relacionada aos produtos. O método fábrica ajuda a dissociar essa lógica das classes concretas de produtos. Aqui está uma analogia: uma grande empresa de desenvolvimento de software pode ter um departamento de treinamento para programadores. No entanto, a principal função da empresa como um todo ainda é escrever código, não produzir programadores.
4. Criadores Concretos sobrescrevem o método fábrica base para retornar um tipo diferente de produto.
   Observe que o método fábrica não precisa criar instâncias o tempo todo. Ele também pode retornar objetos existentes de um cache, um conjunto de objetos, ou outra fonte.

Pseudocódigo:

Este exemplo ilustra como o *Factory* pode ser usado para criar elementos de interface do usuário multiplataforma sem acoplar o código do cliente às classes de UI concretas.



Exemplo de diálogo de plataforma cruzada.

A classe base diálogo usa diferentes elementos da UI do usuário para renderizar sua janela. Em diferentes sistemas operacionais, esses elementos podem parecer um pouco diferentes, mas ainda devem se comportar de forma consistente. Um botão no Windows ainda é um botão no Linux.

Quando o método fábrica entra em ação, você não precisa reescrever a lógica da caixa de diálogo para cada sistema operacional. Se declararmos um método fábrica que produz botões dentro da classe base da caixa de diálogo, mais tarde podemos criar uma subclasse de caixa de diálogo que retorna botões no estilo Windows do método fábrica. A subclasse herda a maior parte do código da caixa de diálogo da classe base, mas, graças ao método fábrica, pode renderizar botões estilo Windows na tela.

Para que esse padrão funcione, a classe base da caixa de diálogo deve funcionar com botões abstratos: uma classe base ou uma interface que todos os botões concretos seguem. Dessa forma, o código da caixa de diálogo permanece funcional, independentemente do tipo de botão com o qual ela trabalha.

Obviamente, você também pode aplicar essa abordagem a outros elementos da UI. No entanto, com cada novo método fábrica adicionado à caixa de diálogo, você se aproxima do padrão *Abstract Factory*.

```
// A classe criadora declara o método fábrica que deve retornar
// um objeto de uma classe produto. As subclasses da criadora
// geralmente fornecem a implementação desse método.
class Dialog is
    // A criadora também pode fornecer alguma implementação
    // padrão do Factory Method.
    abstract method createButton():Button
    // Observe que, apesar do seu nome, a principal
    // responsabilidade da criadora não é criar produtos. Ela
    // geralmente contém alguma lógica de negócio central que
    // depende dos objetos produto retornados pelo método
    // fábrica. As subclasses podem mudar indiretamente essa
    // lógica de negócio ao sobrescreverem o método fábrica e
    // retornarem um tipo diferente de produto dele.
    method render() is
        // Chame o método fábrica para criar um objeto produto.
        Button okButton = createButton()
        // Agora use o produto.
        okButton.onClick(closeDialog)
```

```
            okButton.render()
    // Criadores concretos sobrescrevem o método fábrica para mudar
    // o tipo de produto resultante.
    class WindowsDialog extends Dialog is
        method createButton():Button is
            return new WindowsButton()

    class WebDialog extends Dialog is
        method createButton():Button is
            return new HTMLButton()


    // A interface do produto declara as operações que todos os
    // produtos concretos devem implementar.
    interface Button is
        method render()
        method onClick(f)

    // Produtos concretos fornecem várias implementações da
    // interface do produto.
    class WindowsButton implements Button is
        method render(a, b) is
            // Renderiza um botão no estilo Windows.
        method onClick(f) is
            // Vincula um evento de clique do SO nativo.
    class HTMLButton implements Button is
        method render(a, b) is
            // Retorna uma representação HTML de um botão.
        method onClick(f) is
            // Vincula um evento de clique no navegador web.
    class Application is
        field dialog: Dialog
        // A aplicação seleciona um tipo de criador dependendo da
        // configuração atual ou definições de ambiente.
        method initialize() is
            config = readApplicationConfigFile()
            if (config.OS == "Windows") then
                dialog = new WindowsDialog()
            else if (config.OS == "Web") then
                dialog = new WebDialog()
            else
                throw new Exception("Error! Unknown operating
                                    system.")
        // O código cliente trabalha com uma instância de um criador
        // concreto, ainda que com sua interface base. Desde que o
        // cliente continue trabalhando com a criadora através da
        // interface base, você pode passar qualquer subclasse da
        // criadora.
        method main() is
            this.initialize()
            dialog.render()
```

Aplicabilidade:

Use o *Factory* quando não souber de antemão os tipos e dependências exatas dos objetos com os quais seu código deve funcionar.

O *Factory* separa o código de construção do produto do código que realmente usa o produto. Portanto, é mais fácil estender o código de construção do produto independentemente do restante do código.

Por exemplo, para adicionar um novo tipo de produto à aplicação, só será necessário criar uma subclasse criadora e substituir o método fábrica nela.

Use o *Factory* quando desejar fornecer aos usuários da sua biblioteca ou framework uma maneira de estender seus componentes internos.

Herança é provavelmente a maneira mais fácil de estender o comportamento padrão de uma biblioteca ou framework. Mas como o framework reconheceria que sua subclasse deve ser usada em vez de um componente padrão?

A solução é reduzir o código que constrói componentes no framework em um único método fábrica e permitir que qualquer pessoa sobrescreva esse método, além de estender o próprio componente.

Vamos ver como isso funcionaria. Imagine que você escreva uma aplicação usando um framework de UI de código aberto. Sua aplicação deve ter botões redondos, mas o *framework* fornece apenas botões quadrados. Você estende a classe padrão Botão com uma gloriosa subclasse BotãoRedondo. Mas agora você precisa informar à classe principal *UIFramework* para usar a nova subclasse no lugar do botão padrão.

Para conseguir isso, você cria uma subclasse UIComBotõesRedondos a partir de uma classe base do framework e sobrescreve seu método criarBotão. Enquanto este método retorna objetos Botão na classe base, você faz sua subclasse retornar objetos BotãoRedondo. Agora use a classe UIComBotõesRedondos no lugar de *UIFramework*.

Use o *Factory* quando deseja economizar recursos do sistema reutilizando objetos existentes em vez de recriá-los sempre.

Você irá enfrentar essa necessidade ao lidar com objetos grandes e pesados, como conexões com bancos de dados, sistemas de arquivos e recursos de rede.

Vamos pensar no que deve ser feito para reutilizar um objeto existente:

1. Primeiro, você precisa criar algum armazenamento para manter o controle de todos os objetos criados.
2. Quando alguém solicita um objeto, o programa deve procurar um objeto livre dentro desse conjunto.
3. ...e retorná-lo ao código cliente.
4. Se não houver objetos livres, o programa deve criar um (e adicioná-lo ao conjunto de objetos).

Isso é muito código! E tudo deve ser colocado em um único local para que você não polua o programa com código duplicado.

Provavelmente, o lugar mais óbvio e conveniente onde esse código deve ficar é no construtor da classe cujos objetos estamos tentando reutilizar. No entanto, um construtor deve sempre retornar novos objetos por definição. Não pode retornar instâncias existentes.

Portanto, você precisa ter um método regular capaz de criar objetos e reutilizar os existentes. Isso parece muito com um método fábrica.

Como implementar:

1. Faça todos os produtos implementarem a mesma interface. Essa interface deve declarar métodos que fazem sentido em todos os produtos.
2. Adicione um método fábrica vazio dentro da classe criadora. O tipo de retorno do método deve corresponder à interface comum do produto.
3. No código da classe criadora, encontre todas as referências aos construtores de produtos. Um por um, substitua-os por chamadas ao método fábrica, enquanto extrai o código de criação do produto para o método fábrica.

   Pode ser necessário adicionar um parâmetro temporário ao método fábrica para controlar o tipo de produto retornado.

   Neste ponto, o código do método fábrica pode parecer bastante feio. Pode ter um grande operador `switch` que escolhe qual classe de produto instanciar. Mas não se preocupe, resolveremos isso em breve.
4. Agora, crie um conjunto de subclasses criadoras para cada tipo de produto listado no método fábrica. Sobrescreva o método fábrica nas subclasses e extraia os pedaços apropriados do código de construção do método base.

5. Se houver muitos tipos de produtos e não fizer sentido criar subclasses para todos eles, você poderá reutilizar o parâmetro de controle da classe base nas subclasses.
   Por exemplo, imagine que você tenha a seguinte hierarquia de classes: a classe base Correio com algumas subclasses: CorreioAéreo e CorreioTerrestre; as classes Transporte são Avião, Caminhão e Trem. Enquanto a classe CorreioAéreo usa apenas objetos Avião, o CorreioTerrestre pode funcionar com os objetos Caminhão e Trem. Você pode criar uma subclasse (por exemplo, CorreioFerroviário) para lidar com os dois casos, mas há outra opção. O código do cliente pode passar um argumento para o método fábrica da classe CorreioTerrestre para controlar qual produto ele deseja receber.
6. Se, após todas as extrações, o método fábrica base ficar vazio, você poderá torná-lo abstrato. Se sobrar algo, você pode tornar isso em um comportamento padrão do método.

Prós e contras:

- Você evita acoplamentos firmes entre o criador e os produtos concretos.
- *Princípio de responsabilidade única*. Você pode mover o código de criação do produto para um único local do programa, facilitando a manutenção do código.
- *Princípio aberto/fechado*. Você pode introduzir novos tipos de produtos no programa sem quebrar o código cliente existente.
- O código pode se tornar mais complicado, pois você precisa introduzir muitas subclasses novas para implementar o padrão. O melhor cenário é quando você está introduzindo o padrão em uma hierarquia existente de classes criadoras.

Exemplos de uso: O padrão *Factory* é amplamente utilizado no código TypeScript. É muito útil quando você precisa fornecer um alto nível de flexibilidade para seu código.

Identificação: Os métodos *Factory* podem ser reconhecidos por métodos de criação, que criam objetos de classes concretas, mas os retornam como objetos de tipo ou interface abstrata.

Exemplo conceitual:
Este exemplo ilustra a estrutura do padrão de projeto **Factory Method**. Ele se concentra em responder a estas perguntas:

- De quais classes ele consiste?
- Quais papéis essas classes desempenham?
- De que maneira os elementos do padrão estão relacionados?

index.ts: Exemplo conceitual

```
/* The Creator class declares the factory method that is supposed
to return an object of a Product class. The Creator's subclasses
usually provide the implementation of this method.*/
abstract class Creator {
    /* Note that the Creator may also provide some default
     * implementation of the factory method.*/
    public abstract factoryMethod(): Product;
    /* Also note that, despite its name, the Creator's primary
     * responsibility is not creating products. Usually, it
     * contains some core business logic that relies on Product
     * objects, returned by the factory method. Subclasses can
     * indirectly change that business logic by overriding the
     * factory method and returning a different type of product
     * from it.
     */
    public someOperation(): string {
        // Call the factory method to create a Product object.
```

```
        const product = this.factoryMethod();
        // Now, use the product.
        return `Creator: The same creator's code has just worked
                with ${product.operation()}`;
    }
}
/*Concrete Creators override the factory method in order to change
the resulting product's type.*/
class ConcreteCreator1 extends Creator {
    /* Note that the signature of the method still uses the
     * abstract product type, even though the concrete product is
     * actually returned from the method. This way the Creator can
     * stay independent of concrete product classes.*/
    public factoryMethod(): Product {
        return new ConcreteProduct1();
    }
}
class ConcreteCreator2 extends Creator {
    public factoryMethod(): Product {
        return new ConcreteProduct2();
    }
}
/* The Product interface declares the operations that all concrete
 * products must implement.*/
interface Product {
    operation(): string;
}
/* Concrete Products provide various implementations of the
Product interface.*/
class ConcreteProduct1 implements Product {
    public operation(): string {
        return '{Result of the ConcreteProduct1}';
    }
}
class ConcreteProduct2 implements Product {
    public operation(): string {
        return '{Result of the ConcreteProduct2}';
    }
}
/* The client code works with an instance of a concrete creator,
 * albeit through its base interface. As long as the client keeps
 * working with the creator via the base interface, you can pass
 * it any creator's subclass*/
function clientCode(creator: Creator) {
    // ...
    console.log('Client: I\'m not aware of the creator\'s class,
                        but it still works.');
    console.log(creator.someOperation());
    // ...
}
/*The Application picks a creator's type depending on the
configuration or environment.*/
console.log('App: Launched with the ConcreteCreator1.');
clientCode(new ConcreteCreator1());
console.log('');

console.log('App: Launched with the ConcreteCreator2.');
clientCode(new ConcreteCreator2());
```

III. Adapter

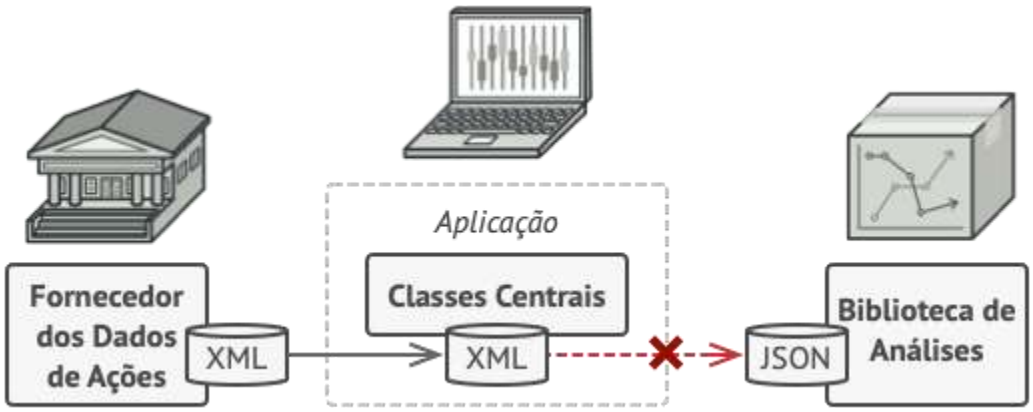
Também conhecido como: Adaptador, Wrapper.

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

Problema:

Imagine que você está criando uma aplicação de monitoramento do mercado de ações da bolsa. A aplicação baixa os dados as ações de múltiplas fontes em formato XML e então mostra gráficos e diagramas maneiros para o usuário.

Em algum ponto, você decide melhorar a aplicação ao integrar uma biblioteca de análise de terceiros. Mas aqui está a pegadinha: a biblioteca só trabalha com dados em formato JSON.



Você não pode usar a biblioteca "como ela está" porque ela espera os dados em um formato que é incompatível com sua aplicação.

Você poderia mudar a biblioteca para que ela funcione com XML. Contudo, isso pode quebrar algum código existente que depende da biblioteca. E pior, você pode não ter acesso ao código fonte da biblioteca para começo de conversa, fazendo dessa abordagem uma tarefa impossível.

Solução:

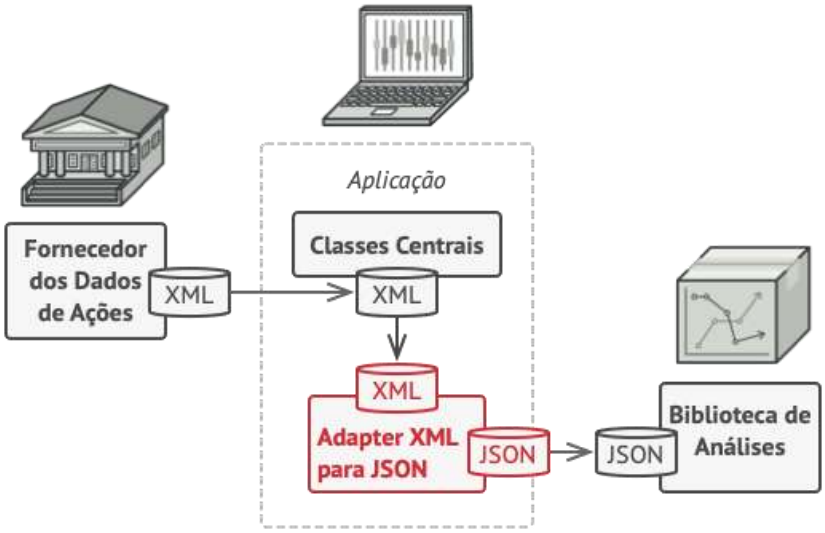
Você pode criar um *adaptador*. Ele é um objeto especial que converte a interface de um objeto para que outro objeto possa entendê-lo.

Um adaptador encobre um dos objetos para esconder a complexidade da conversão acontecendo nos bastidores. O objeto encobrido nem fica ciente do adaptador. Por exemplo, você pode encobrir um objeto que opera em metros e quilômetros com um adaptador que converte todos os dados para unidades imperiais tais como pés e milhas.

Adaptadores podem não só converter dados em vários formatos, mas também podem ajudar objetos com diferentes interfaces a colaborar. Veja aqui como funciona:

1. O adaptador obtém uma interface, compatível com um dos objetos existentes.
2. Usando essa interface, o objeto existente pode chamar os métodos do adaptador com segurança.
3. Ao receber a chamada, o adaptador passa o pedido para o segundo objeto, mas em um formato e ordem que o segundo objeto espera.

Algumas vezes é possível criar um adaptador de duas vias que pode converter as chamadas em ambas as direções.



Vamos voltar à nossa aplicação da bolsa de valores. Para resolver o dilema dos formatos incompatíveis, você pode criar adaptadores XML-para-JSON para cada classe da biblioteca de análise que seu código trabalha diretamente. Então você ajusta seu código para comunicar-se com a biblioteca através desses adaptadores. Quando um adaptador recebe uma chamada, ele traduz os dados entrantes XML em uma estrutura JSON e passa a chamada para os métodos apropriados de um objeto de análise encoberto.

Analogia com o mundo real:

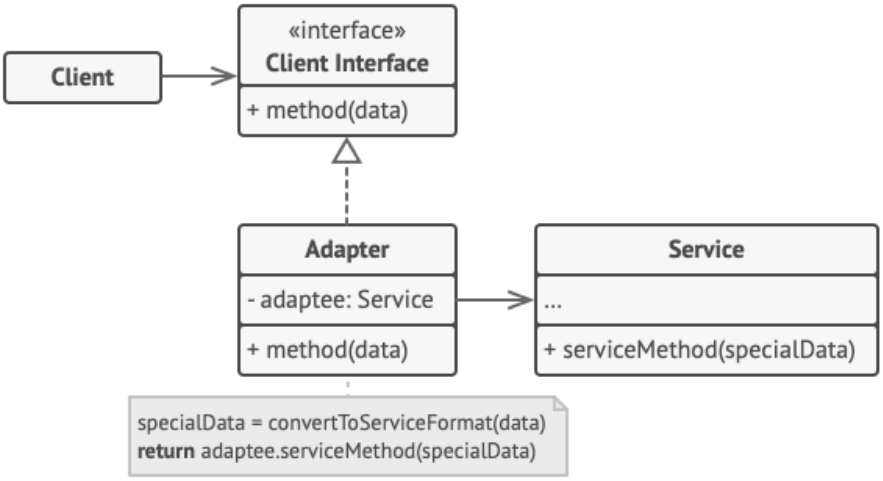
Uma mala antes e depois de uma viagem ao exterior.

Quando você viaja do Brasil para a Europa pela primeira vez, você pode ter uma pequena surpresa quando tenta carregar seu laptop. O plugue e os padrões de tomadas são diferentes em diferentes países. É por isso que seu plugue do Brasil não vai caber em uma tomada da Alemanha. O problema pode ser resolvido usando um adaptador de tomada que tenha o estilo de tomada Brasileira e o plugue no estilo Europeu.

Estrutura

Adaptador de objeto:

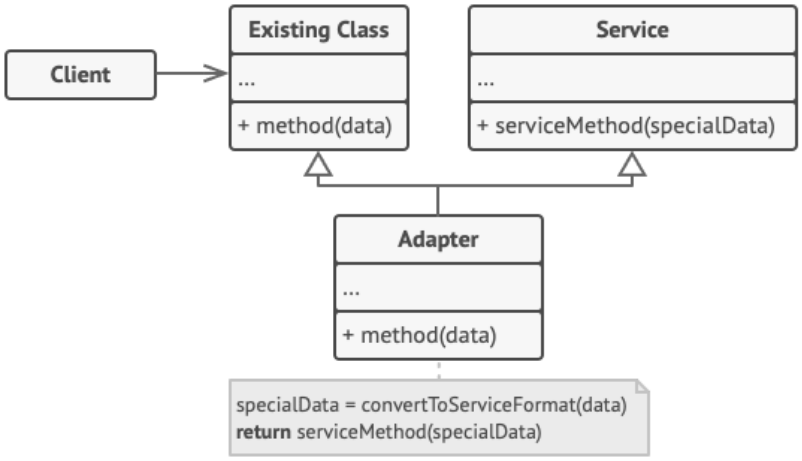
Essa implementação usa o princípio de composição do objeto: o adaptador implementa a interface de um objeto e encobre o outro. Ele pode ser implementado em todas as linguagens de programação populares.



1. O **Cliente** é uma classe que contém a lógica de negócio do programa existente.
2. A **Interface do Cliente** descreve um protocolo que outras classes devem seguir para ser capaz de colaborar com o código cliente.
3. O **Serviço** é alguma classe útil (geralmente de terceiros ou código legado). O cliente não pode usar essa classe diretamente porque ela tem uma interface incompatível.
4. O **Adaptador** é uma classe que é capaz de trabalhar tanto com o cliente quanto o serviço: ela implementa a interface do cliente enquanto encobre o objeto do serviço. O adaptador recebe chamadas do cliente através da interface do cliente e as traduz em chamadas para o objeto encobrido do serviço em um formato que ele possa entender.
5. O código cliente não é acoplado à classe concreta do adaptador desde que ele trabalhe com o adaptador através da interface do cliente. Graças a isso, você pode introduzir novos tipos de adaptadores no programa sem quebrar o código cliente existente. Isso pode ser útil quando a interface de uma classe de serviço é mudada ou substituída: você pode apenas criar uma nova classe adaptador sem mudar o código cliente.

Adaptador de classe:

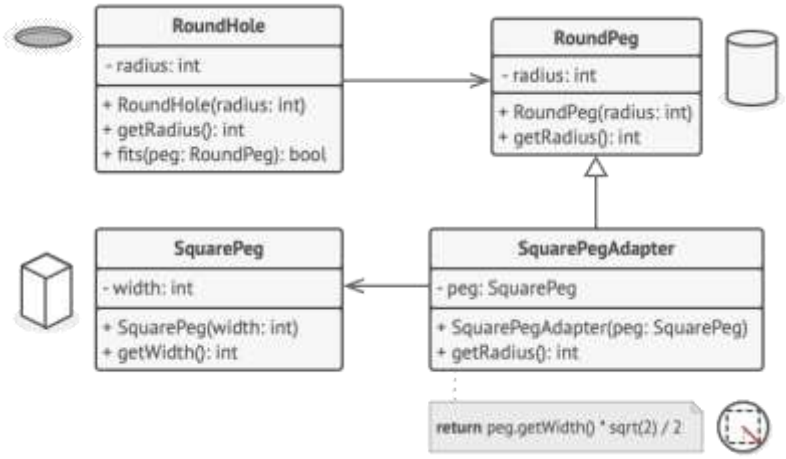
Essa implementação utiliza herança: o adaptador herda interfaces de ambos os objetos ao mesmo tempo. Observe que essa abordagem só pode ser implementada em linguagens de programação que suportam herança múltipla, tais como C++.



1. A **Classe Adaptador** não precisa encobrir quaisquer objetos porque ela herda os comportamentos tanto do cliente como do serviço. A adaptação acontece dentro dos métodos sobrescritos. O adaptador resultante pode ser usado em lugar de uma classe cliente existente.

Pseudocódigo:

Esse exemplo do padrão **Adapter** é baseado no conflito clássico entre pinos quadrados e buracos redondos.



Adaptando pinos quadrados para buracos redondos.

O adaptador finge ser um pino redondo, com um raio igual a metade do diâmetro do quadrado (em outras palavras, o raio do menor círculo que pode acomodar o pino quadrado).

```
// Digamos que você tenha duas classes com interfaces
// compatíveis: RoundHole (Buraco Redondo) e RoundPeg (Pino
// Redondo).
class RoundHole is
    constructor RoundHole(radius) { ... }

    method getRadius() is
        // Retorna o raio do buraco.

    method fits(peg: RoundPeg) is
        return this.getRadius() >= peg.getRadius()

class RoundPeg is
    constructor RoundPeg(radius) { ... }
```

```
        method getRadius() is
            // Retorna o raio do pino.


    // Mas tem uma classe incompatível: SquarePeg (Pino Quadrado).
    class SquarePeg is
        constructor SquarePeg(width) { ... }

        method getWidth() is
            // Retorna a largura do pino quadrado.


    // Uma classe adaptadora permite que você encaixe pinos
    // quadrados em buracos redondos. Ela estende a classe RoundPeg
    // para permitir que objetos do adaptador ajam como pinos
    // redondos.
    class SquarePegAdapter extends RoundPeg is
        // Na verdade, o adaptador contém uma instância da classe
        // SquarePeg.
        private field peg: SquarePeg

        constructor SquarePegAdapter(peg: SquarePeg) is
            this.peg = peg

        method getRadius() is
            // O adaptador finge que é um pino redondo com um raio
            // que encaixaria o pino quadrado que o adaptador está
            // envolvendo.
            return peg.getWidth() * Math.sqrt(2) / 2


    // Em algum lugar no código cliente.
    hole = new RoundHole(5)
    rpeg = new RoundPeg(5)
    hole.fits(rpeg) // true

    small_sqpeg = new SquarePeg(5)
    large_sqpeg = new SquarePeg(10)
    // Isso não vai compilar (tipos incompatíveis).
    hole.fits(small_sqpeg)

    small_sqpeg_adapter = new SquarePegAdapter(small_sqpeg)
    large_sqpeg_adapter = new SquarePegAdapter(large_sqpeg)
    hole.fits(small_sqpeg_adapter) // true
    hole.fits(large_sqpeg_adapter) // false
```

Aplicabilidade:

Utilize a classe Adaptador quando você quer usar uma classe existente, mas sua interface não for compatível com o resto do seu código.

O padrão *Adapter* permite que você crie uma classe de meio termo que serve como um tradutor entre seu código e a classe antiga, uma classe de terceiros, ou qualquer outra classe com uma interface estranha.

Utilize o padrão quando você quer reutilizar diversas subclasses existentes que não possuam alguma funcionalidade comum que não pode ser adicionada a superclasse.

Você pode estender cada subclasse e colocar a funcionalidade faltante nas novas classes filhas. Contudo, você terá que duplicar o código em todas as novas classes, o que não é muito indicado.

Uma solução muito mais elegante seria colocar a funcionalidade faltante dentro da classe adaptadora. Então você encobriria os objetos com as funcionalidades faltantes dentro do adaptador, ganhando tais funcionalidades de forma dinâmica. Para isso funcionar, as classes

alvo devem ter uma interface em comum, e o campo do adaptador deve seguir aquela interface. Essa abordagem se parece muito com o padrão ***Decorator***.

Como implementar:

1. Certifique-se que você tem ao menos duas classes com interfaces incompatíveis:
    - Uma classe *serviço* útil, que você não pode modificar (quase sempre de terceiros, antiga, ou com muitas dependências existentes).
    - Uma ou mais classes *cliente* que seriam beneficiadas com o uso da classe serviço.
2. Declare a interface cliente e descreva como o cliente se comunica com o serviço.
3. Cria a classe adaptadora e faça-a seguir a interface cliente. Deixe todos os métodos vazios por enquanto.
4. Adicione um campo para a classe do adaptador armazenar uma referência ao objeto do serviço. A prática comum é inicializar esse campo via o construtor, mas algumas vezes é mais conveniente passá-lo para o adaptador ao chamar seus métodos.
5. Um por um, implemente todos os métodos da interface cliente na classe adaptadora. O adaptador deve delegar a maioria do trabalho real para o objeto serviço, lidando apenas com a conversão da interface ou formato dos dados.
6. Os Clientes devem usar o adaptador através da interface cliente. Isso irá permitir que você mude ou estenda o adaptador sem afetar o código cliente.

Prós e contras:

- *Princípio de responsabilidade única*. Você pode separar a conversão de interface ou de dados da lógica primária do negócio do programa.
- *Princípio aberto/fechado*. Você pode introduzir novos tipos de adaptadores no programa sem quebrar o código cliente existente, desde que eles trabalhem com os adaptadores através da interface cliente.
- A complexidade geral do código aumenta porque você precisa introduzir um conjunto de novas interfaces e classes. Algumas vezes é mais simples mudar a classe serviço para que ela se adeque com o resto do seu código.

Adapter em TypeScript:

O **Adapter** é um padrão de projeto estrutural, que permite a colaboração de objetos incompatíveis.

O Adapter atua como um *wrapper* entre dois objetos. Ele captura chamadas para um objeto e as deixa reconhecíveis tanto em formato como interface para este segundo objeto.

Exemplos de uso: O padrão Adapter é bastante comum no código TypeScript. É frequentemente usado em sistemas baseados em algum código legado. Nesses casos, os adaptadores criam código legado com classes modernas.

Identificação: O adapter é reconhecível por um construtor que utiliza uma instância de tipo abstrato/interface diferente. Quando o adaptador recebe uma chamada para qualquer um de seus métodos, ele converte parâmetros para o formato apropriado e direciona a chamada para um ou vários métodos do objeto envolvido.

Exemplo conceitual:

Este exemplo ilustra a estrutura do padrão de projeto **Adapter**. Ele se concentra em responder a estas perguntas:

- De quais classes ele consiste?
- Quais papéis essas classes desempenham?

- De que maneira os elementos do padrão estão relacionados?

index.ts: Exemplo conceitual

```
/**
 * The Target defines the domain-specific interface used by the client
code.
 */
class Target {
    public request(): string {
        return 'Target: The default target\'s behavior.';
    }
}

/**
 * The Adaptee contains some useful behavior, but its interface is
incompatible
 * with the existing client code. The Adaptee needs some adaptation before
the
 * client code can use it.
 */
class Adaptee {
    public specificRequest(): string {
        return '.eetpadA eht fo roivaheb laicepS';
    }
}
/**
 * The Adapter makes the Adaptee's interface compatible with the Target's
 * interface.
 */
class Adapter extends Target {
    private adaptee: Adaptee;

    constructor(adaptee: Adaptee) {
        super();
        this.adaptee = adaptee;
    }

    public request(): string {
        const                                result                                =
this.adaptee.specificRequest().split('').reverse().join('');
        return `Adapter: (TRANSLATED) ${result}`;
    }
}
/**
 * The client code supports all classes that follow the Target interface.
 */
function clientCode(target: Target) {
    console.log(target.request());
}
console.log('Client: I can work just fine with the Target objects:');
const target = new Target();
clientCode(target);

console.log('');

const adaptee = new Adaptee();
console.log('Client: The Adaptee class has a weird interface. See, I
don\'t understand it:');
console.log(`Adaptee: ${adaptee.specificRequest()}`);

console.log('');

console.log('Client: But I can work with it via the Adapter:');
const adapter = new Adapter(adaptee);
clientCode(adapter);
```

Output.txt: Resultados da execução

```
Client: I can work just fine with the Target objects:
Target: The default target's behavior.

Client: The Adaptee class has a weird interface. See, I don't understand it:
Adaptee: .eetpadA eht fo roivaheb laicepS

Client: But I can work with it via the Adapter:
Adapter: (TRANSLATED) Special behavior of the Adaptee.
```

IV. Strategy

Também conhecido como: Estratégia

Propósito:

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.

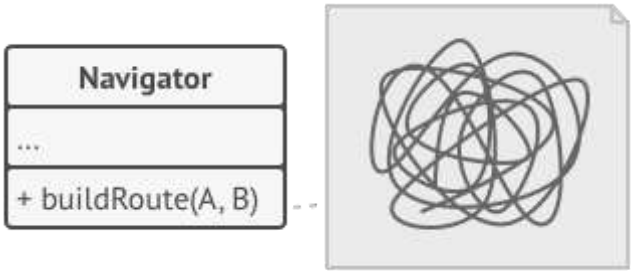
Problema

Um dia você decide criar uma aplicação de navegação para viajantes casuais. A aplicação estava centrada em um mapa bonito que ajudava os usuários a se orientarem rapidamente em uma cidade.

Uma das funcionalidades mais pedidas para a aplicação era o planejamento automático de rotas. Um usuário deveria ser capaz de entrar com um endereço e ver a rota mais rápida no mapa.

A primeira versão da aplicação podia apenas construir rotas sobre rodovias, e isso agradou muito quem viaja de carro. Porém aparentemente, nem todo mundo dirige em suas férias. Então com a próxima atualização você adicionou uma opção de construir rotas de caminhada. Logo após isso você adicionou outra opção para permitir que as pessoas usem o transporte público.

Contudo, isso foi apenas o começo. Mais tarde você planejou adicionar um construtor de rotas para ciclistas. E mais tarde, outra opção para construir rotas até todas as atrações turísticas de uma cidade.



O código do navegador ficou muito inchado.

Embora da perspectiva de negócio a aplicação tenha sido um sucesso, a parte técnica causou a você muitas dores de cabeça. Cada vez que você adicionava um novo algoritmo de roteamento, a classe principal do navegador dobrava de tamanho. Em determinado momento, o monstro se tornou algo muito difícil de se manter.

Qualquer mudança a um dos algoritmos, seja uma simples correção de bug ou um pequeno ajuste no valor das ruas, afetava toda a classe, aumentando a chance de criar um erro no código já existente.

Além disso, o trabalho em equipe se tornou ineficiente. Seus companheiros de equipe, que foram contratados após ao bem-sucedido lançamento do produto, se queixavam que gastavam muito tempo resolvendo conflitos de fusão. Implementar novas funcionalidades necessitava mudanças na classe gigantesca, conflitando com os códigos criados por outras pessoas.

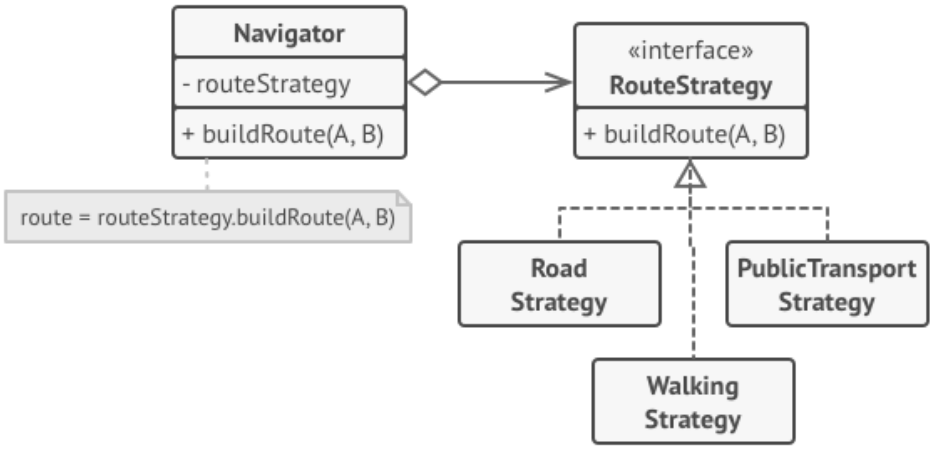
Solução:

O padrão Strategy sugere que você pegue uma classe que faz algo específico em diversas maneiras diferentes e extraia todos esses algoritmos para classes separadas chamadas *estratégias*.

A classe original, chamada *contexto*, deve ter um campo para armazenar uma referência para um dessas estratégias. O contexto delega o trabalho para um objeto estratégia ao invés de executá-lo por conta própria.

O contexto não é responsável por selecionar um algoritmo apropriado para o trabalho. Ao invés disso, o cliente passa a estratégia desejada para o contexto. Na verdade, o contexto não sabe muito sobre as estratégias. Ele trabalha com todas elas através de uma interface genérica, que somente expõe um único método para acionar o algoritmo encapsulado dentro da estratégia selecionada.

Desta forma o contexto se torna independente das estratégias concretas, então você pode adicionar novos algoritmos ou modificar os existentes sem modificar o código do contexto ou outras estratégias.

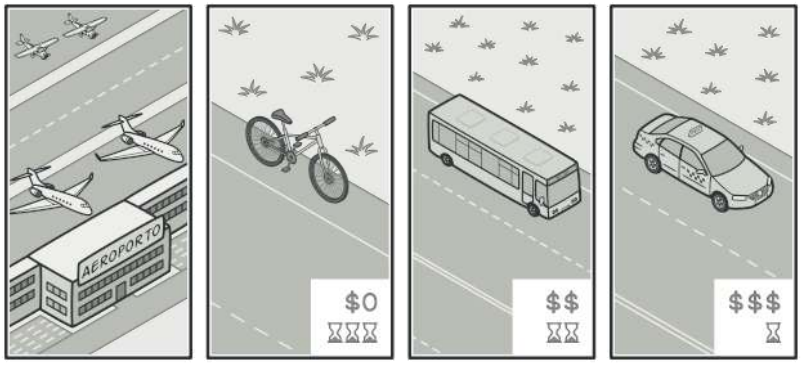


Estratégias de planejamento de rotas.

Em nossa aplicação de navegação, cada algoritmo de roteamento pode ser extraído para sua própria classe com um único método `construirRota`. O método aceita uma origem e um destino e retorna uma coleção de pontos da rota.

Mesmo dando os mesmos argumentos, cada classe de roteamento pode construir uma rota diferente, a classe navegadora principal não se importa qual algoritmo está selecionado uma vez que seu trabalho primário é renderizar um conjunto de pontos num mapa. A classe tem um método para trocar a estratégia ativa de rotas, então seus clientes, bem como os botões na interface de usuário, podem substituir o comportamento de rotas selecionado por um outro.

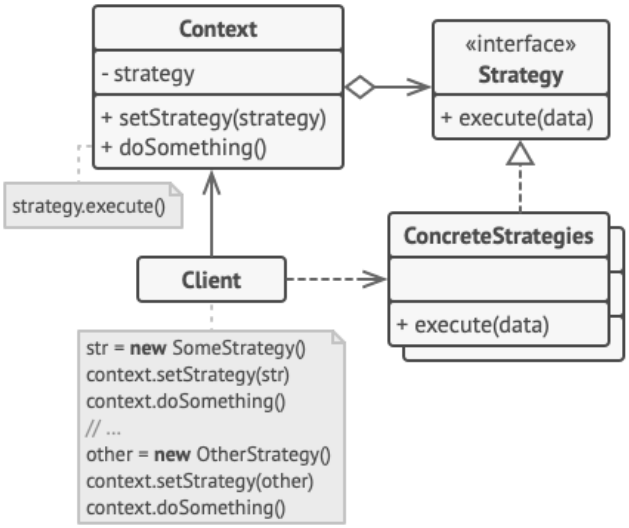
Analogia com o mundo real:



Várias estratégias para se chegar ao aeroporto.

Imagine que você tem que chegar ao aeroporto. Você pode pegar um ônibus, pedir um táxi, ou subir em sua bicicleta. Essas são suas estratégias de transporte. Você pode escolher uma das estratégias dependendo de fatores como orçamento ou restrições de tempo.

Estrutura:



1. O **Contexto** mantém uma referência para uma das estratégias concretas e se comunica com esse objeto através da interface da estratégia.
2. A interface **Estratégia** é comum à todas as estratégias concretas. Ela declara um método que o contexto usa para executar uma estratégia.
3. **Estratégias Concretas** implementam diferentes variações de um algoritmo que o contexto usa.
4. O contexto chama o método de execução no objeto estratégia ligado cada vez que ele precisa rodar um algoritmo. O contexto não sabe qual tipo de estratégia ele está trabalhando ou como o algoritmo é executado.
5. O **Cliente** cria um objeto estratégia específico e passa ele para o contexto. O contexto expõe um setter que permite o cliente mudar a estratégia associada com contexto durante a execução.

Pseudocódigo:

Neste exemplo, o contexto usa múltiplas **estratégias** para executar várias operações aritméticas.

```
// A interface estratégia declara operações comuns a todas as
// versões suportadas de algum algoritmo. O contexto usa essa
// interface para chamar o algoritmo definido pelas estratégias
// concretas.
interface Strategy is
    method execute(a, b)
// Estratégias concretas implementam o algoritmo enquanto seguem
// a interface estratégia base. A interface faz delas
// intercomunicáveis no contexto.
class ConcreteStrategyAdd implements Strategy is
    method execute(a, b) is
        return a + b

class ConcreteStrategySubtract implements Strategy is
    method execute(a, b) is
        return a - b

class ConcreteStrategyMultiply implements Strategy is
    method execute(a, b) is
        return a * b
// O contexto define a interface de interesse para clientes.
class Context is
    // O contexto mantém uma referência para um dos objetos
    // estratégia. O contexto não sabe a classe concreta de uma
    // estratégia. Ele deve trabalhar com todas as estratégias
    // através da interface estratégia.
    private strategy: Strategy
    // Geralmente o contexto aceita uma estratégia através do
    // construtor, e também fornece um setter para que a
    // estratégia possa ser trocado durante o tempo de execução.
    method setStrategy(Strategy strategy) is
        this.strategy = strategy
    // O contexto delega algum trabalho para o objeto estratégia
    // ao invés de implementar múltiplas versões do algoritmo
    // por conta própria.
    method executeStrategy(int a, int b) is
        return strategy.execute(a, b)
// O código cliente escolhe uma estratégia concreta e passa ela
// para o contexto. O cliente deve estar ciente das diferenças
// entre as estratégia para que faça a escolha certa.
class ExampleApplication is
    method main() is
        Cria um objeto contexto.

        Lê o primeiro número.
        Lê o último número.
        Lê a ação desejada da entrada do usuário

        if (action == addition) then
            context.setStrategy(new ConcreteStrategyAdd())

        if (action == subtraction) then
            context.setStrategy(new ConcreteStrategySubtract())

        if (action == multiplication) then
            context.setStrategy(new ConcreteStrategyMultiply())

        result = context.executeStrategy(First number, Second number)

        Imprimir resultado.
```

Aplicabilidade:

Utilize o padrão Strategy quando você quer usar diferentes variantes de um algoritmo dentro de um objeto e ser capaz de trocar de um algoritmo para outro durante a execução.

O padrão Strategy permite que você altere indiretamente o comportamento de um objeto durante a execução ao associá-lo com diferentes sub-objetos que pode fazer sub-tarefas específicas em diferentes formas.

Utilize o Strategy quando você tem muitas classes parecidas que somente diferem na forma que elas executam algum comportamento.

O padrão Strategy permite que você extraia o comportamento variante para uma hierarquia de classe separada e combine as classes originais em uma, portando reduzindo código duplicado.

Utilize o padrão para isolar a lógica do negócio de uma classe dos detalhes de implementação de algoritmos que podem não ser tão importantes no contexto da lógica.

O padrão Strategy permite que você isole o código, dados internos, e dependências de vários algoritmos do restante do código. Vários clientes podem obter uma simples interface para executar os algoritmos e trocá-los durante a execução do programa.

Utilize o padrão quando sua classe tem um operador condicional muito grande que troca entre diferentes variantes do mesmo algoritmo.

O padrão Strategy permite que você se livre dessa condicional ao extrair todos os algoritmos para classes separadas, todos eles implementando a mesma interface. O objeto original delega a execução de um desses objetos, ao invés de implementar todas as variantes do algoritmo.

Como implementar:

1. Na classe contexto, identifique um algoritmo que é sujeito a frequentes mudanças. Pode ser também uma condicional enorme que seleciona e executa uma variante do mesmo algoritmo durante a execução do programa.
2. Declare a interface da estratégia comum para todas as variantes do algoritmo.
3. Um por um, extraia todos os algoritmos para suas próprias classes. Elas devem todas implementar a interface estratégia.
4. Na classe contexto, adicione um campo para armazenar uma referência a um objeto estratégia. Forneça um setter para substituir valores daquele campo. O contexto deve trabalhar com o objeto estratégia somente através da interface estratégia. O contexto pode definir uma interface que deixa a estratégia acessar seus dados.
5. Os Clientes do contexto devem associá-lo com uma estratégia apropriada que coincide com a maneira que esperam que o contexto atue em seu trabalho primário.

Prós e contras:

- Você pode trocar algoritmos usados dentro de um objeto durante a execução.
- Você pode isolar os detalhes de implementação de um algoritmo do código que usa ele.
- Você pode substituir a herança por composição.
- *Princípio aberto/fechado*. Você pode introduzir novas estratégias sem mudar o contexto.
- Se você só tem um par de algoritmos e eles raramente mudam, não há motivo real para deixar o programa mais complicado com novas classes e interfaces que vêm junto com o padrão.
- Os Clientes devem estar cientes das diferenças entre as estratégias para serem capazes de selecionar a adequada.
- Muitas linguagens de programação modernas têm suporte do tipo funcional que permite que você implemente diferentes versões de um algoritmo dentro de um conjunto de funções anônimas. Então você poderia usar essas funções exatamente como se estivesse usando objetos estratégia, mas sem inchar seu código com classes e interfaces adicionais.

Strategy em TypeScript:

O **Strategy** é um padrão de projeto comportamental que transforma um conjunto de comportamentos em objetos e os torna intercambiáveis dentro do objeto de contexto original.

O objeto original, chamado contexto, mantém uma referência a um objeto strategy e o delega a execução do comportamento. Para alterar a maneira como o contexto executa seu trabalho, outros objetos podem substituir o objeto strategy atualmente vinculado por outro.

Exemplos de uso: O padrão Strategy é muito comum no código TypeScript. É frequentemente usado em várias estruturas para fornecer aos usuários uma maneira de alterar o comportamento de uma classe sem estendê-la.

Identificação: O padrão Strategy pode ser reconhecido por um método que permite que o objeto aninhado faça o trabalho real, bem como pelo setter que permite substituir esse objeto por outro diferente.

Exemplo conceitual

Este exemplo ilustra a estrutura do padrão de projeto **Strategy**. Ele se concentra em responder a estas perguntas:

- De quais classes ele consiste?
- Quais papéis essas classes desempenham?
- De que maneira os elementos do padrão estão relacionados?
-

index.ts: Exemplo conceitual

```
/** The Context defines the interface of interest to clients.*/
class Context {
    /** @type {Strategy} The Context maintains a reference to one of the
    Strategy objects. The Context does not know the concrete class of a
    strategy. It should work with all strategies via the Strategy
    interface.*/
    private strategy: Strategy;
    /* Usually, the Context accepts a strategy through the constructor,
    but also provides a setter to change it at runtime.*/
    constructor(strategy: Strategy) {
        this.strategy = strategy;
    }
    /* Usually, the Context allows replacing a Strategy object at runtime.
    public setStrategy(strategy: Strategy) {
        this.strategy = strategy;
    }
    /* The Context delegates some work to the Strategy object instead of
     * implementing multiple versions of the algorithm on its own.*/
    public doSomeBusinessLogic(): void {
        // ...

       console.log('Context: Sorting data using the strategy (not sure how
                   it\'ll do it)');
        const result = this.strategy.doAlgorithm(['a', 'b', 'c', 'd',
                   'e']);
        console.log(result.join(','));
        // ...
    }
}
/*The Strategy interface declares operations common to all supported
versions of some algorithm.
* The Context uses this interface to call the algorithm defined by Concrete
* Strategies.*/
interface Strategy {
    doAlgorithm(data: string[]): string[];
}
/* Concrete Strategies implement the algorithm while following the base
```

```
Strategy interface. The interface makes them interchangeable in the
Context.*/
class ConcreteStrategyA implements Strategy {
    public doAlgorithm(data: string[]): string[] {
        return data.sort();
    }
}

class ConcreteStrategyB implements Strategy {
    public doAlgorithm(data: string[]): string[] {
        return data.reverse();
    }
}

/*The client code picks a concrete strategy and passes it to the context.
The client should be aware of the differences between strategies in order
to make the right choice.*/

const context = new Context(new ConcreteStrategyA());
console.log('Client: Strategy is set to normal sorting.');
context.doSomeBusinessLogic();

console.log('');

console.log('Client: Strategy is set to reverse sorting.');
context.setStrategy(new ConcreteStrategyB());
context.doSomeBusinessLogic();
```

Output.txt: Resultados da execução

Client: Strategy is set to normal sorting.
Context: Sorting data using the strategy (not sure how it'll do it)
a,b,c,d,e

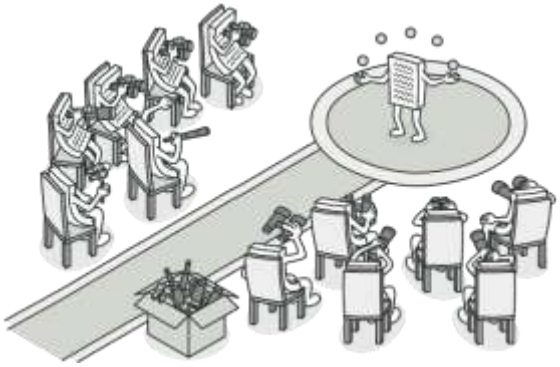
Client: Strategy is set to reverse sorting.
Context: Sorting data using the strategy (not sure how it'll do it)
e,d,c,b,a

V. Observer

Também conhecido como: Observador, Assinante do evento, EventSubscriber, Escutador, Listener

Propósito:

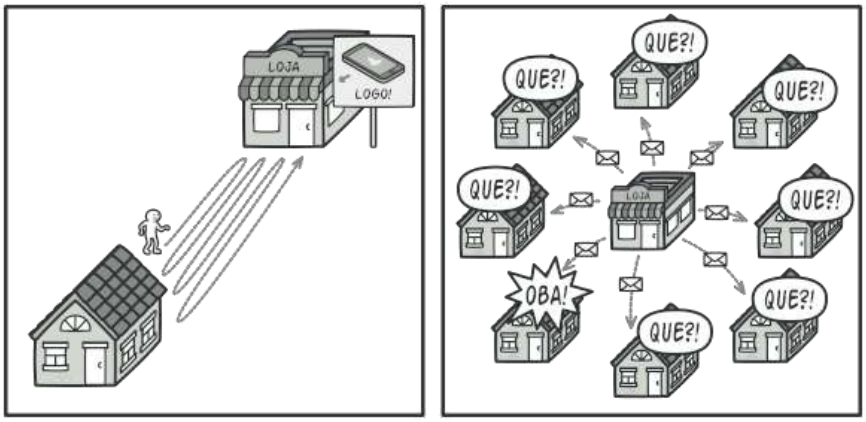
O **Observer** é um padrão de projeto comportamental que permite que você defina um mecanismo de assinatura para notificar múltiplos objetos sobre quaisquer eventos que aconteçam com o objeto que eles estão observando.



Problema:

Imagine que você tem dois tipos de objetos: um Cliente e uma Loja. O cliente está muito interessado em uma marca particular de um produto (digamos que seja um novo modelo de iPhone) que logo deverá estar disponível na loja.

O cliente pode visitar a loja todos os dias e checar a disponibilidade do produto. Mas enquanto o produto ainda está a caminho, a maioria dessas visitas serão em vão.



Visitando a loja vs. enviando spam

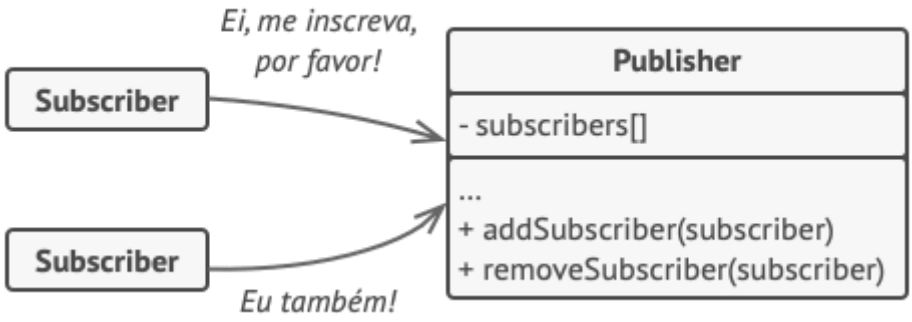
Por outro lado, a loja poderia mandar milhares de e-mails (que poderiam ser considerados como spam) para todos os clientes cada vez que um novo produto se torna disponível. Isso salvaria alguns clientes de incontáveis viagens até a loja. Porém, ao mesmo tempo, irritaria outros clientes que não estão interessados em novos produtos.

Parece que temos um conflito. Ou o cliente gasta tempo verificando a disponibilidade do produto ou a loja gasta recursos notificando os clientes errados.

Solução:

O objeto que tem um estado interessante é quase sempre chamado de *sujeito*, mas já que ele também vai notificar outros objetos sobre as mudanças em seu estado, nós vamos chamá-lo de *publicador*. Todos os outros objetos que querem saber das mudanças do estado do publicador são chamados de *assinantes*.

O padrão Observer sugere que você adicione um mecanismo de assinatura para a classe editora para que objetos individuais possam assinar ou desassinar uma corrente de eventos vindo daquela editora. Nada tema! Nada é complicado como parece. Na verdade, esse mecanismo consiste em 1) um vetor para armazenar uma lista de referências aos objetos do assinante e 2) alguns métodos públicos que permitem adicionar assinantes e removê-los da lista.

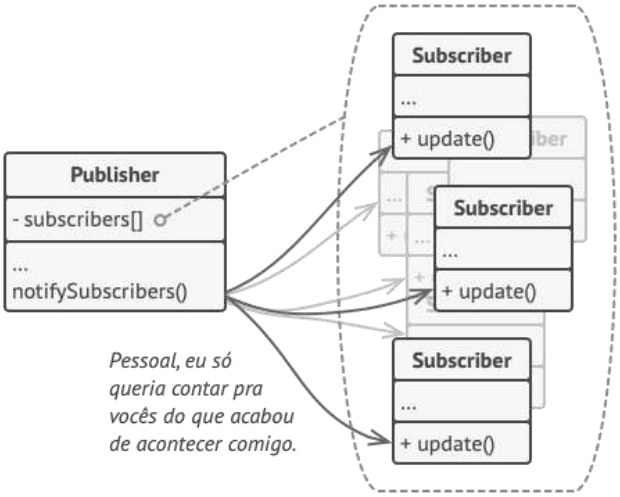


Um mecanismo de assinatura permite que objetos individuais se inscrevam a notificações de eventos.

Agora, sempre que um evento importante acontece com a editora, ele passa para seus assinantes e chama um método específico de notificação em seus objetos.

Aplicações reais podem ter dúzias de diferentes classes assinantes que estão interessadas em acompanhar eventos da mesma classe editora. Você não iria querer acoplar a editora a todas essas classes. Além disso, você pode nem estar ciente de algumas delas de antemão se a sua classe editora deve ser usada por outras pessoas.

É por isso que é crucial que todos os assinantes implementem a mesma interface e que a editora se comunique com eles apenas através daquela interface. Essa interface deve declarar o método de notificação junto com um conjunto de parâmetros que a editora pode usar para passar alguns dados contextuais junto com a notificação.



A editora notifica os assinantes chamando um método específico de notificação em seus objetos.

Se a sua aplicação tem diferentes tipos de editoras e você quer garantir que seus assinantes são compatíveis com todas elas, você pode ir além e fazer todas as editoras seguirem a mesma interface. Essa interface precisa apenas descrever alguns métodos de inscrição. A interface permitirá assinantes observar o estado das editoras sem se acoplar a suas classes concretas.

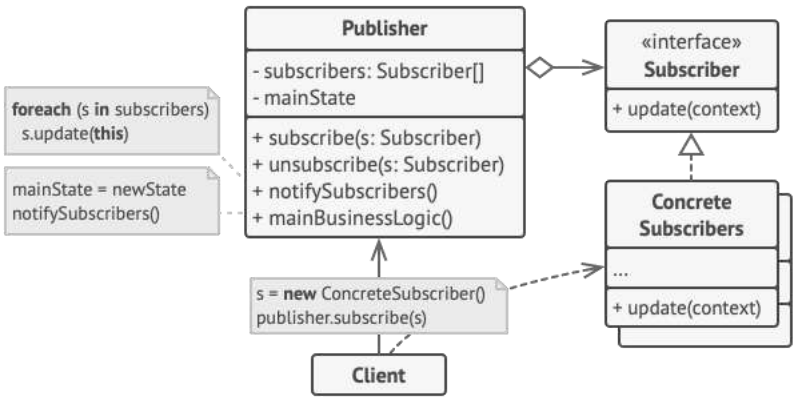
Analogia com o mundo real:

Assinaturas de revistas e jornais.

Se você assinar um jornal ou uma revista, você não vai mais precisar ir até a banca e ver se a próxima edição está disponível. Ao invés disso a editora manda novas edições diretamente para sua caixa de correio após a publicação ou até mesmo com antecedência.

A editora mantém uma lista de assinantes e sabe em quais revistas eles estão interessados. Os assinantes podem deixar essa lista a qualquer momento quando desejarem que a editora pare de enviar novas revistas para eles.

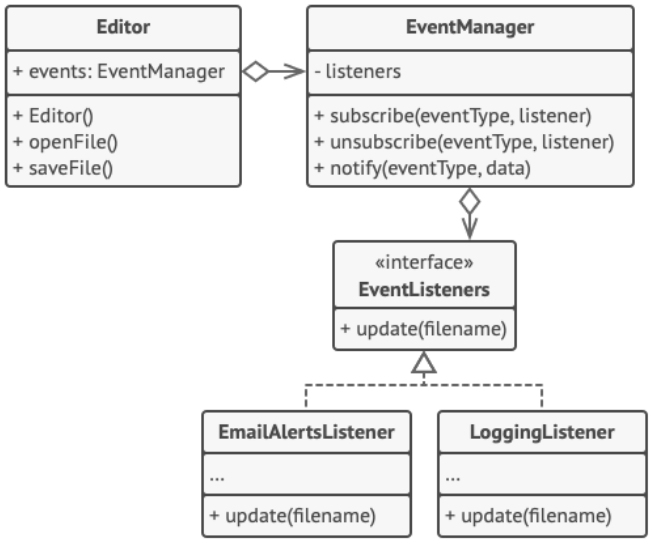
Estrutura:



1. A **Editora** manda eventos de interesse para outros objetos. Esses eventos ocorrem quando a editora muda seu estado ou executa algum comportamento. As editoras contêm uma infraestrutura de inscrição que permite novos assinantes se juntar aos atuais assinantes ou deixar a lista.
2. Quando um novo evento acontece, a editora percorre a lista de assinantes e chama o método de notificação declarado na interface do assinante em cada objeto assinante.
3. A interface do **Assinante** declara a interface de notificação. Na maioria dos casos ela consiste de um único método atualizar. O método pode ter vários parâmetros que permite que a editora passe alguns detalhes do evento junto com a atualização.
4. **Assinantes Concretos** realizam algumas ações em resposta às notificações enviadas pela editora. Todas essas classes devem implementar a mesma interface para que a editora não fique acoplada à classes concretas.
5. Geralmente, assinantes precisam de alguma informação contextual para lidar com a atualização corretamente. Por esse motivo, as editoras quase sempre passam algum dado de contexto como argumentos do método de notificação. A editora pode passar a si mesmo como um argumento, permitindo que o assinante recupere quaisquer dados diretamente.
6. O **Cliente** cria a editora e os objetos assinantes separadamente e então registra os assinantes para as atualizações da editora.

Pseudocódigo:

Neste exemplo o padrão **Observer** permite que um objeto editor de texto notifique outros objetos de serviço sobre mudanças em seu estado.



Notificando objetos sobre eventos que aconteceram com outros objetos.

A lista de assinantes é compilada dinamicamente: objetos podem começar ou parar de ouvir às notificações durante a execução do programa, dependendo do comportamento desejado pela sua aplicação.

Nesta implementação, a classe do editor não mantém a lista de assinatura por si mesmo. Ele delega este trabalho para um objeto ajudante especial devotado a fazer apenas isso. Você pode melhorar aquele objeto para servir como um enviador de eventos centralizado, permitindo que qualquer objeto aja como uma editora.

Adicionar novos assinantes ao programa não exige mudança nas classes editoras existentes, desde que elas trabalhem com todos os assinantes através da mesma interface.

```
// A classe editora base inclui o código de gerenciamento de
// inscrições e os métodos de notificação.
class EventManager is
    private field listeners: hash map of event types and listeners

    method subscribe(eventType, listener) is
        listeners.add(eventType, listener)

    method unsubscribe(eventType, listener) is
        listeners.remove(eventType, listener)

    method notify(eventType, data) is
        foreach (listener in listeners.of(eventType)) do
            listener.update(data)
// O publicador concreto contém a verdadeira lógica de negócio
// que é de interesse para alguns assinantes. Nós podemos
// derivar essa classe a partir do publicador base, mas isso nem
// sempre é possível na vida real devido a possibilidade do
// publicador concreto já ser uma subclasse. Neste caso, você
// pode remendar a lógica de inscrição com a composição, como
// fizemos aqui.
class Editor is
    public field events: EventManager
    private field file: File

    constructor Editor() is
        events = new EventManager()
    // Métodos da lógica de negócio podem notificar assinantes
    // acerca de mudanças.
    method openFile(path) is
```

```
            this.file = new File(path)
            events.notify("open", file.name)

        method saveFile() is
            file.write()
            events.notify("save", file.name)
// Aqui é a interface do assinante. Se sua linguagem de
// programação suporta tipos funcionais, você pode substituir
// toda a hierarquia do assinante por um conjunto de funções.
interface EventListener is
    method update(filename)
// Assinantes concretos reagem a atualizações emitidas pelo
// publicador a qual elas estão conectadas.
class LoggingListener implements EventListener is
    private field log: File
    private field message: string

    constructor LoggingListener(log_filename, message) is
        this.log = new File(log_filename)
        this.message = message

    method update(filename) is
        log.write(replace('%s',filename,message))

class EmailAlertsListener implements EventListener is
    private field email: string
    private field message: string

    constructor EmailAlertsListener(email, message) is
        this.email = email
        this.message = message

    method update(filename) is
        system.email(email, replace('%s',filename,message))
// Uma aplicação pode configurar publicadores e assinantes
// durante o tempo de execução.
class Application is
    method config() is
        editor = new Editor()

        logger = new LoggingListener(
            "/path/to/log.txt",
            "Someone has opened the file: %s")
        editor.events.subscribe("open", logger)

        emailAlerts = new EmailAlertsListener(
            "admin@example.com",
            "Someone has changed the file: %s")
        editor.events.subscribe("save", emailAlerts
```

Aplicabilidade:

Utilize o padrão Observer quando mudanças no estado de um objeto podem precisar mudar outros objetos, e o atual conjunto de objetos é desconhecido de antemão ou muda dinamicamente.

Você pode vivenciar esse problema quando trabalhando com classes de interface gráfica do usuário. Por exemplo, você criou classes de botões customizados, e você quer deixar os clientes colocar algum código customizado para seus botões para que ele ative sempre que usuário aperta um botão.

O padrão Observer permite que qualquer objeto que implemente a interface do assinante possa se inscrever para notificações de eventos em objetos da editora. Você pode

adicionar o mecanismo de inscrição em seus botões, permitindo que o cliente coloque seu próprio código através de classes assinantes customizadas.

Utilize o padrão quando alguns objetos em sua aplicação devem observar outros, mas apenas por um tempo limitado ou em casos específicos.

A lista de inscrição é dinâmica, então assinantes podem entrar e sair da lista sempre que quiserem.

Como implementar:

1. Olhe para sua lógica do negócio e tente quebrá-la em duas partes: a funcionalidade principal, independente de outros códigos, irá agir como editora; o resto será transformado em um conjunto de classes assinantes.
2. Declare a interface do assinante. No mínimo, ela deve declarar um único método atualizar.
3. Declare a interface da editora e descreva um par de métodos para adicionar um objeto assinante e removê-lo da lista. Lembre-se que editoras somente devem trabalhar com assinantes através da interface do assinante.
4. Decida onde colocar a lista atual de assinantes e a implementação dos métodos de inscrição. Geralmente este código se parece o mesmo para todos os tipos de editoras, então o lugar óbvio para colocá-lo é dentro de uma classe abstrata derivada diretamente da interface da editora. Editoras concretas estendem aquela classe, herdando o comportamento de inscrição.
   Contudo, se você está aplicando o padrão para uma hierarquia de classe já existente, considere uma abordagem baseada na composição: coloque a lógica da inscrição dentro de um objeto separado, e faça todos as editoras reais usá-la.
5. Crie as classes editoras concretas. A cada vez que algo importante acontece dentro de uma editora, ela deve notificar seus assinantes.
6. Implemente os métodos de notificação de atualização nas classes assinantes concretas. A maioria dos assinantes precisarão de dados contextuais sobre o evento. Eles podem ser passados como argumentos do método de notificação.
   Mas há outra opção. Ao receber uma notificação, o assinante pode recuperar os dados diretamente da notificação. Neste caso, a editora deve passar a si mesma através do método de atualização. A opção menos flexível é ligar uma editora ao assinante permanentemente através do construtor.
7. O cliente deve criar todas os assinantes necessários e registrá-los com suas editoras apropriadas.

Prós e contras:

- *Princípio aberto/fechado*. Você pode introduzir novas classes assinantes sem ter que mudar o código da editora (e vice-versa se existe uma interface editora).
- Você pode estabelecer relações entre objetos durante a execução.
- Assinantes são notificados em ordem aleatória

Relações com outros padrões

- O **Chain of Responsibility**, **Command**, **Mediator** e **Observer** abrangem várias maneiras de se conectar remetentes e destinatários de pedidos:
  - O *Chain of Responsibility* passa um pedido sequencialmente ao longo de um corrente dinâmica de potenciais destinatários até que um deles atua no pedido.
  - O *Command* estabelece conexões unidirecionais entre remetentes e destinatários.
  - O *Mediator* elimina as conexões diretas entre remetentes e destinatários, forçando-os a se comunicar indiretamente através de um objeto mediador.

    - O *Observer* permite que destinatários inscrevam-se ou cancelem sua inscrição dinamicamente para receber pedidos.
- A diferença entre o **Mediator** e o **Observer** é bem obscura. Na maioria dos casos, você pode implementar qualquer um desses padrões; mas às vezes você pode aplicar ambos simultaneamente. Vamos ver como podemos fazer isso.
  O objetivo primário do *Mediator* é eliminar dependências múltiplas entre um conjunto de componentes do sistema. Ao invés disso, esses componentes se tornam dependentes de um único objeto mediador. O objetivo do *Observer* é estabelecer comunicações de uma via dinâmicas entre objetos, onde alguns deles agem como subordinados de outros.
  Existe uma implementação popular do padrão Mediator que depende do *Observer*. O objeto mediador faz o papel de um publicador, e os componentes agem como assinantes que inscrevem-se ou removem a inscrição aos eventos do mediador. Quando o *Mediator* é implementado dessa forma, ele pode parecer muito similar ao *Observer*.
  Quando você está confuso, lembre-se que você pode implementar o padrão Mediator de outras maneiras. Por exemplo, você pode ligar permanentemente todos os componentes ao mesmo objeto mediador. Essa implementação não se parece com o *Observer* mas ainda irá ser uma instância do padrão Mediator.
  Agora imagine um programa onde todos os componentes se tornaram publicadores permitindo conexões dinâmicas entre si. Não haverá um objeto mediador centralizado, somente um conjunto distribuído de observadores.

Observer em TypeScript

O **Observer** é um padrão de projeto comportamental que permite que um objeto notifique outros objetos sobre alterações em seu estado.

O padrão Observer fornece uma maneira de assinar e cancelar a assinatura desses eventos para qualquer objeto que implemente uma interface de assinante.

Exemplos de uso: O padrão Observer é bastante comum no código TypeScript, especialmente nos componentes da interface de usuário. Ele fornece uma maneira de reagir a eventos que acontecem em outros objetos sem acoplamento às suas classes.

Identificação: O padrão pode ser reconhecido por métodos de assinatura, que armazenam objetos em uma lista e por chamadas para o método de atualização emitido para objetos nessa lista.

Exemplo conceitual:

Este exemplo ilustra a estrutura do padrão de projeto **Observer**. Ele se concentra em responder a estas perguntas:

- De quais classes ele consiste?
- Quais papéis essas classes desempenham?
- De que maneira os elementos do padrão estão relacionados?

index.ts: Exemplo conceitual

```
/* The Subject interface declares a set of methods for managing
subscribers.*/
interface Subject {
    // Attach an observer to the subject.
    attach(observer: Observer): void;
    // Detach an observer from the subject.
    detach(observer: Observer): void;
    // Notify all observers about an event.
    notify(): void;
```

```
}
/*The Subject owns some important state and notifies observers when the
state changes.*/
class ConcreteSubject implements Subject {
    /*@type {number} For the sake of simplicity, the Subject's state,
     * essential to all subscribers, is stored in this variable.*/
    public state: number;
    /* @type {Observer[]} List of subscribers. In real life, the list of
     * subscribers can be stored more comprehensively (categorized by
     * event type, etc.).*/
    private observers: Observer[] = [];
    /* The subscription management methods.*/
    public attach(observer: Observer): void {
        const isExist = this.observers.includes(observer);
        if (isExist) {
            return console.log('Subject: Observer has been attached
                   already.');
        }
        console.log('Subject: Attached an observer.');
        this.observers.push(observer);
    }

    public detach(observer: Observer): void {
        const observerIndex = this.observers.indexOf(observer);
        if (observerIndex === -1) {
            return console.log('Subject: Nonexistent observer.');
        }
        this.observers.splice(observerIndex, 1);
        console.log('Subject: Detached an observer.');
    }
    /* Trigger an update in each subscriber.*/
    public notify(): void {
        console.log('Subject: Notifying observers...');
        for (const observer of this.observers) {
            observer.update(this);
        }
    }
    /*Usually, the subscription logic is only a fraction of what a Subject
     * can really do. Subjects commonly hold some important business
     * logic, that triggers a notification method whenever something
     * important is about to happen (or after it).
     */
    public someBusinessLogic(): void {
        console.log('\nSubject: I\'m doing something important.');
        this.state = Math.floor(Math.random() * (10 + 1));

        console.log(`Subject:   My   state   has   just   changed   to:
${this.state}`);
        this.notify();
    }
}
/*The Observer interface declares the update method, used by subjects.*/
interface Observer {
    // Receive update from subject.
    update(subject: Subject): void;
}
/* Concrete Observers react to the updates issued by the Subject they had
 * been attached to.*/
class ConcreteObserverA implements Observer {
    public update(subject: Subject): void {
        if (subject instanceof ConcreteSubject && subject.state < 3) {
            console.log('ConcreteObserverA: Reacted to the event.');
        }
    }
}
class ConcreteObserverB implements Observer {
    public update(subject: Subject): void {
```

```
        if (subject instanceof ConcreteSubject && (subject.state === 0 ||
subject.state >= 2)) {
            console.log('ConcreteObserverB: Reacted to the event.');
        }
    }
}
/* The client code.*/

const subject = new ConcreteSubject();

const observer1 = new ConcreteObserverA();
subject.attach(observer1);

const observer2 = new ConcreteObserverB();
subject.attach(observer2);

subject.someBusinessLogic();
subject.someBusinessLogic();

subject.detach(observer2);

subject.someBusinessLogic();
```

Output.txt: Resultados da execução

```
Subject: Attached an observer.
Subject: Attached an observer.

Subject: I'm doing something important.
Subject: My state has just changed to: 6
Subject: Notifying observers...
ConcreteObserverB: Reacted to the event.

Subject: I'm doing something important.
Subject: My state has just changed to: 1
Subject: Notifying observers...
ConcreteObserverA: Reacted to the event.
Subject: Detached an observer.

Subject: I'm doing something important.
Subject: My state has just changed to: 5
Subject: Notifying observers...
```

VI. Referências:

Refactoring Guru, Padrões de Projetos, em https://refactoring.guru/pt-br/design-patterns, acesso em 03/06/2024.