Design Patterns

STRATEGY – ADAPTER – BRIDGE – FACTORY - ABSTRACT FACTORY – SINGLETON - FACADE - COMMAND

1. **STRATEGY:**
   - **Conceito:** O **padrão Strategy** define uma família de algoritmos, encapsula cada um deles e os torna intercambiáveis. Permite que o cliente escolha o algoritmo a ser utilizado em tempo de execução.
   - **Aplicação:** Útil quando há diferentes algoritmos disponíveis para realizar uma tarefa e é desejável permitir que o cliente escolha dinamicamente qual algoritmo utilizar.
   - **Cenário:** Um sistema de precificação dinâmica para uma loja online. A loja oferece descontos diferentes com base em estratégias variáveis, como descontos sazonais, descontos por lealdade do cliente ou descontos especiais para produtos específicos. Cada estratégia de precificação é encapsulada como uma classe separada, e o sistema permite que a loja escolha dinamicamente a estratégia de precificação a ser aplicada.

```
// Interface para as estratégias de precificação
interface PricingStrategy {
    double calculatePrice(double price);
}

// Estratégia de precificação sazonal
class SeasonalPricing implements PricingStrategy {
    @Override
    public double calculatePrice(double price) {
        // Lógica de precificação sazonal
        return price * 0.9; // 10% de desconto sazonal
    }
}
```

```
// Estratégia de precificação de lealdade do cliente
class LoyaltyPricing implements PricingStrategy {
    @Override
    public double calculatePrice(double price) {
        // Lógica de precificação de lealdade
        return price * 0.8; // 20% de desconto para clientes leais
    }
}

// Contexto que utiliza a estratégia
class PriceContext {
    private PricingStrategy pricingStrategy;

    public PriceContext(PricingStrategy pricingStrategy) {
        this.pricingStrategy = pricingStrategy;
    }

    public double calculateFinalPrice(double initialPrice) {
        return pricingStrategy.calculatePrice(initialPrice);
    }
}
```

2. **ADAPTER (Adaptador):**
    - **Conceito:** O padrão Adapter permite que a interface de uma classe existente seja usada como outra interface. Ele atua como um intermediário que permite que objetos com interfaces incompatíveis colaborem.
    - **Aplicação:** Útil quando você precisa integrar sistemas ou componentes que têm interfaces diferentes.
    - **Cenário:** Uma empresa adota um novo sistema de processamento de pedidos que tem uma interface diferente do sistema antigo. Para garantir a continuidade dos negócios, um adaptador é criado para traduzir as chamadas do sistema antigo para o formato esperado pelo novo sistema, permitindo uma transição suave entre os dois sem alterar o código existente.

```
// Interface do sistema antigo
interface SistemaAntigo {
    void processarPedidoAntigo();
}

// Implementação do sistema antigo
class SistemaAntigoImpl implements SistemaAntigo {
@Override
    public void processarPedidoAntigo() {
        System.out.println("Processando pedido no sistema antigo.");
    }
}

// Nova interface do sistema
interface NovoSistema {
    void processarPedidoNovo();
}

// Adaptador para o sistema antigo se adequar à nova interface
class AdaptadorSistemaAntigo implements NovoSistema {
```

```
    private SistemaAntigo sistemaAntigo;

    public AdaptadorSistemaAntigo(SistemaAntigo) {
        this.sistemaAntigo = sistemaAntigo;
    }

    @Override
    public void processarPedidoNovo() {
        sistemaAntigo.processarPedidoAntigo();
    }
}
```

3. **BRIDGE (Ponte):**
    - **Conceito:** O padrão Bridge separa uma abstração da sua implementação, de modo que ambas possam evoluir independentemente uma da outra.
    - **Aplicação:** Útil quando há uma necessidade de evitar um vínculo permanente entre uma abstração e sua implementação, permitindo que ambas possam ser estendidas de forma independente.
    - **Cenário:** Uma empresa de e-commerce possui diferentes plataformas de pagamento (PayPal, cartão de crédito direto, criptomoedas). O padrão Bridge é aplicado para separar a lógica de pagamento da lógica de processamento de pedidos. Dessa forma, a empresa pode introduzir novos métodos de pagamento sem modificar a lógica principal de processamento de pedidos.

```
// Interface de implementação
interface PlataformaPagamento {
    void processarPagamento();
}

// Implementação específica para PayPal
class PayPal implements PlataformaPagamento {
    @Override
    public void processarPagamento() {
        System.out.println("Processando pagamento via PayPal.");
    }
}

// Implementação específica para cartão de crédito
class CartaoCredito implements PlataformaPagamento {
    @Override
    public void processarPagamento() {
        System.out.println("Processando pagamento via cartão de crédito.");
    }
}
```

```
// Abstração que utiliza a implementação
abstract class Pedido {
    protected PlataformaPagamento plataforma;

    public Pedido (PlataformaPagamento plataforma) {
        this.plataforma = plataforma;
    }

    public abstract void processarPedido();
}

// Implementação específica para processar pedidos
class ProcessadorPedidos extends Pedido {
    public ProcessadorPedidos(PlataformaPagamento plataforma) {
        super(plataforma);
    }

    @Override
    public void processarPedido() {
        System.out.println("Processando pedido...");
        plataforma.processarPagamento();
    }
}
```

4. **FACTORY (Fábrica):**

- **Conceito:** O padrão Factory fornece uma interface para criar objetos em uma superclasse, mas permite que as subclasses alterem o tipo de objetos que serão criados.
- **Aplicação:** Útil quando a criação de um objeto envolve uma lógica complexa, e essa lógica precisa ser encapsulada em uma classe separada.
- **Cenário:** Uma fábrica de automóveis deseja criar diferentes modelos de carros. Cada modelo tem suas próprias complexidades de fabricação. O padrão Factory é utilizado para criar uma fábrica abstrata que define uma interface para a criação de diferentes partes de um carro, enquanto as fábricas concretas produzem peças específicas para cada modelo.

```
// Interface para a criação de carros
interface CarFactory {
    Car createCar();
}

// Interface para os carros
interface Car {
    void assemble();
}

// Implementação concreta da fábrica e do carro
class SedanFactory implements CarFactory {
    @Override
    public Car createCar() {
        return new Sedan();
    }
}

class Sedan implements Car {
    @Override
    public void assemble() {
        System.out.println("Assembling Sedan Car");
    }
}
```

```
class SUVFactory implements CarFactory {
    @Override
    public Car createCar() {
        return new SUV();
    }
}

class SUV implements Car {
    @Override
    public void assemble() {
        System.out.println("Assembling SUV Car");
    }
}

// Cliente que utiliza a fábrica
public class CarClient {

    public static void main(String[] args) {

        CarFactory sedanFactory = new SedanFactory();
        Car sedan = sedanFactory.createCar();
        sedan.assemble(); // Saída: Assembling Sedan Car

        CarFactory suvFactory = new SUVFactory();
        Car suv = suvFactory.createCar();
        suv.assemble(); // Saída: Assembling SUV Car

    }
}
```

5. **ABSTRACT FACTORY (Fábrica Abstrata):**
    - **Conceito:** O padrão Abstract Factory fornece uma interface para criar famílias de objetos relacionados ou dependentes sem especificar suas classes concretas.
    - **Aplicação:** Útil quando um sistema precisa ser independente de como seus objetos são criados, compostos e representados, e os sistemas são configurados para trabalhar com várias famílias de objetos.
    - **Cenário:** Um sistema de gerenciamento de restaurantes oferece opções para restaurantes de fast food e restaurantes gourmet. Cada tipo de restaurante possui diferentes categorias de itens de menu, como entradas, pratos principais e sobremesas. O padrão Abstract Factory é aplicado para criar famílias de objetos relacionados, como FastFoodFactory e GourmetFactory, que produzem itens de menu compatíveis com cada tipo de restaurante.

```
// Interface para os produtos relacionados a restaurantes (Entrada, Prato Principal, Sobremesa)
interface Entrada {
    void exibirInfo();
}

interface PratoPrincipal {
    void exibirInfo();
}

interface Sobremesa {
    void exibirInfo();
}

// Fábrica abstrata para criar famílias de produtos relacionados a restaurantes
interface RestauranteFactory {
    Entrada criarEntrada();
    PratoPrincipal criarPratoPrincipal();
```

```
    Sobremesa criarSobremesa();
}

// Implementações concretas para produtos de restaurante de fast food
class FastFoodEntrada implements Entrada {
    @Override
    public void exibirInfo() {
        System.out.println("Fast Food: Entrada");
    }
}

class FastFoodPratoPrincipal implements PratoPrincipal {
    @Override
    public void exibirInfo() {
        System.out.println("Fast Food: Prato Principal");
    }
}

class FastFoodSobremesa implements Sobremesa {
    @Override
    public void exibirInfo() {
        System.out.println("Fast Food: Sobremesa");
    }
}

// Implementações concretas para produtos de restaurante gourmet
class GourmetEntrada implements Entrada {
    @Override
    public void exibirInfo() {
        System.out.println("Gourmet: Entrada");
```

```
    }
}

class GourmetPratoPrincipal implements PratoPrincipal {
    @Override
    public void exibirInfo() {
        System.out.println("Gourmet: Prato Principal");
    }
}

class GourmetSobremesa implements Sobremesa {
    @Override
    public void exibirInfo() {
        System.out.println("Gourmet: Sobremesa");
    }
}


// Fábrica concreta para criar produtos de restaurante de fast food
class FastFoodFactory implements RestauranteFactory {
    @Override
    public Entrada criarEntrada() {
        return new FastFoodEntrada();
    }

    @Override
    public PratoPrincipal criarPratoPrincipal() {
        return new FastFoodPratoPrincipal();
    }
```

```
    @Override
    public Sobremesa criarSobremesa() {
        return new FastFoodSobremesa();
    }
}

// Fábrica concreta para criar produtos de restaurante gourmet
class GourmetFactory implements RestauranteFactory {
    @Override
    public Entrada criarEntrada() {
        return new GourmetEntrada();
    }

    @Override
    public PratoPrincipal criarPratoPrincipal() {
        return new GourmetPratoPrincipal();
    }

    @Override
    public Sobremesa criarSobremesa() {
        return new GourmetSobremesa();
    }
}

// Cliente que utiliza as fábricas abstratas
public class RestauranteClient {

    public static void main(String[] args) {

        // Criando um restaurante de fast food
```

```
        RestauranteFactory fastFoodFactory = new FastFoodFactory();
        Entrada fastFoodEntrada = fastFoodFactory.criarEntrada();
        PratoPrincipal fastFoodPratoPrincipal = fastFoodFactory.criarPratoPrincipal();
        Sobremesa fastFoodSobremesa = fastFoodFactory.criarSobremesa();

        fastFoodEntrada.exibirInfo();             // Saída: Fast Food: Entrada
        fastFoodPratoPrincipal.exibirInfo();      // Saída: Fast Food: Prato Principal
        fastFoodSobremesa.exibirInfo();           // Saída: Fast Food: Sobremesa

        // Criando um restaurante gourmet
        RestauranteFactory gourmetFactory = new GourmetFactory();
        Entrada gourmetEntrada = gourmetFactory.criarEntrada();
        PratoPrincipal gourmetPratoPrincipal = gourmetFactory.criarPratoPrincipal();
        Sobremesa gourmetSobremesa = gourmetFactory.criarSobremesa();

        gourmetEntrada.exibirInfo();              // Saída: Gourmet: Entrada
        gourmetPratoPrincipal.exibirInfo();       // Saída: Gourmet: Prato Principal
        gourmetSobremesa.exibirInfo();            // Saída: Gourmet: Sobremesa

    }
}
```

OUTROS EXEMPLOS

ABSTRACT FACTORY

```
// Interfaces para as fábricas abstratas
interface AbstractCarFactory {
    Sedan createSedan();
    SUV createSUV();
}

// Interfaces para os carros abstratos
interface Sedan {
    void assembleSedan();
}

interface SUV {
    void assembleSUV();
}

// Implementações concretas das fábricas e carros
class LuxuryCarFactory implements AbstractCarFactory {
    @Override
    public Sedan createSedan() {
        return new LuxurySedan();
    }

    @Override
    public SUV createSUV() {
        return new LuxurySUV();
    }
}
```

```
class LuxurySedan implements Sedan {
    @Override
    public void assembleSedan() {
        System.out.println("Assembling Luxury Sedan Car");
    }
}

class LuxurySUV implements SUV {
    @Override
    public void assembleSUV() {
        System.out.println("Assembling Luxury SUV Car");
    }
}

class EconomyCarFactory implements AbstractCarFactory {
    @Override
    public Sedan createSedan() {
        return new EconomySedan();
    }

    @Override
    public SUV createSUV() {
        return new EconomySUV();
    }
}

class EconomySedan implements Sedan {
    @Override
    public void assembleSedan() {
```

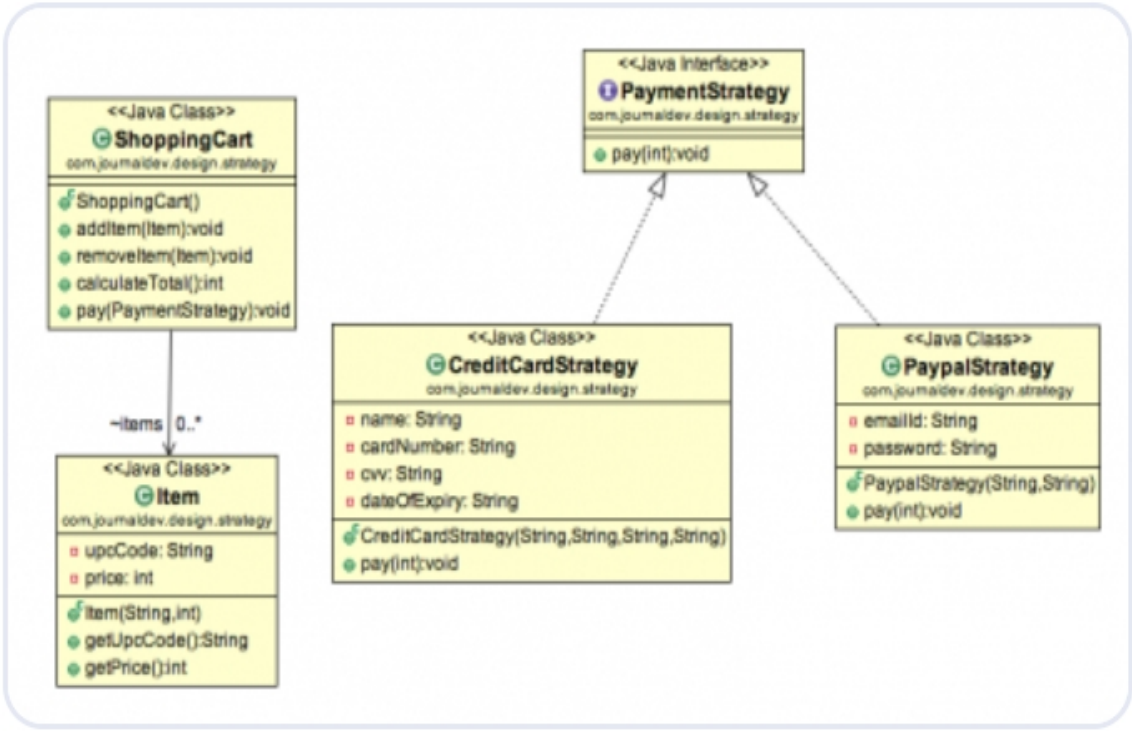
```
        System.out.println("Assembling Economy Sedan Car");
    }
}

class EconomySUV implements SUV {
    @Override
    public void assembleSUV() {
        System.out.println("Assembling Economy SUV Car");
    }
}


// Cliente que utiliza a fábrica abstrata
public class AbstractCarClient {

    public static void main(String[] args) {

        AbstractCarFactory luxuryFactory = new LuxuryCarFactory();
        Sedan luxurySedan = luxuryFactory.createSedan();
        luxurySedan.assembleSedan(); // Saída: Assembling Luxury Sedan Car
        SUV luxurySUV = luxuryFactory.createSUV();
        luxurySUV.assembleSUV(); // Saída: Assembling Luxury SUV Car

        AbstractCarFactory economyFactory = new EconomyCarFactory();
        Sedan economySedan = economyFactory.createSedan();
        economySedan.assembleSedan(); // Saída: Assembling Economy Sedan Car
        SUV economySUV = economyFactory.createSUV();
        economySUV.assembleSUV(); // Saída: Assembling Economy SUV Car
    }
```

```
}
```

A escolha entre esses padrões depende do contexto específico do problema que você está resolvendo. Se você conseguiu atender a uma tarefa usando vários padrões, é possível que você tenha escolhido abordagens diferentes para alcançar o mesmo resultado. A escolha do padrão dependerá das necessidades específicas do seu sistema, da flexibilidade desejada e da manutenibilidade do código. Não há uma resposta única sobre qual é "mais aplicável" sem um contexto mais detalhado sobre o problema que você está tentando resolver.

USANDO PADRÃO STRATEGY

Strategy Design Pattern Class Diagram



Para lidar com catracas de fornecedores diferentes que possuem configurações de acesso distintas, você pode considerar o uso do padrão de projeto Strategy.

O padrão Strategy permite que você defina uma família de algoritmos, encapsule cada um deles e os torne intercambiáveis.
Isso é útil quando você tem várias maneiras de realizar uma tarefa e deseja poder escolher a abordagem a ser usada dinamicamente.

Aqui está um exemplo simples em Java utilizando o padrão Strategy para lidar com configurações de acesso diferentes em catracas de fornecedores diferentes:

```
// Interface para os algoritmos de configuração de acesso
interface AccessConfiguration {
    void configureAccess();
}

// Implementações específicas para DIMEP e HIKEVISION
class DimepAccessConfiguration implements AccessConfiguration {
    @Override
    public void configureAccess() {
        // Lógica de configuração de acesso para DIMEP
        System.out.println("Configurando acesso para DIMEP");
    }
}

class HikevisionAccessConfiguration implements AccessConfiguration {
    @Override
    public void configureAccess() {
        // Lógica de configuração de acesso para HIKEVISION
        System.out.println("Configurando acesso para HIKEVISION");
    }
}
```

// Classe que utiliza a estratégia de configuração de acesso

```
class AccessConfigurationContext {

	private AccessConfiguration accessConfiguration;

	public AccessConfigurationContext(AccessConfiguration accessConfiguration) {
		this.accessConfiguration = accessConfiguration;
	}

	public void setAccessConfiguration(AccessConfiguration accessConfiguration) {
		this.accessConfiguration = accessConfiguration;
	}

	public void configureAccess() {
		accessConfiguration.configureAccess();
	}
}
```

// Exemplo de uso

```
public class Main {

	public static void main(String[] args) {

		// Criando instâncias das configurações de acesso específicas
		AccessConfiguration dimepConfig = new DimepAccessConfiguration();
		AccessConfiguration hikevisionConfig = new HikevisionAccessConfiguration();
```

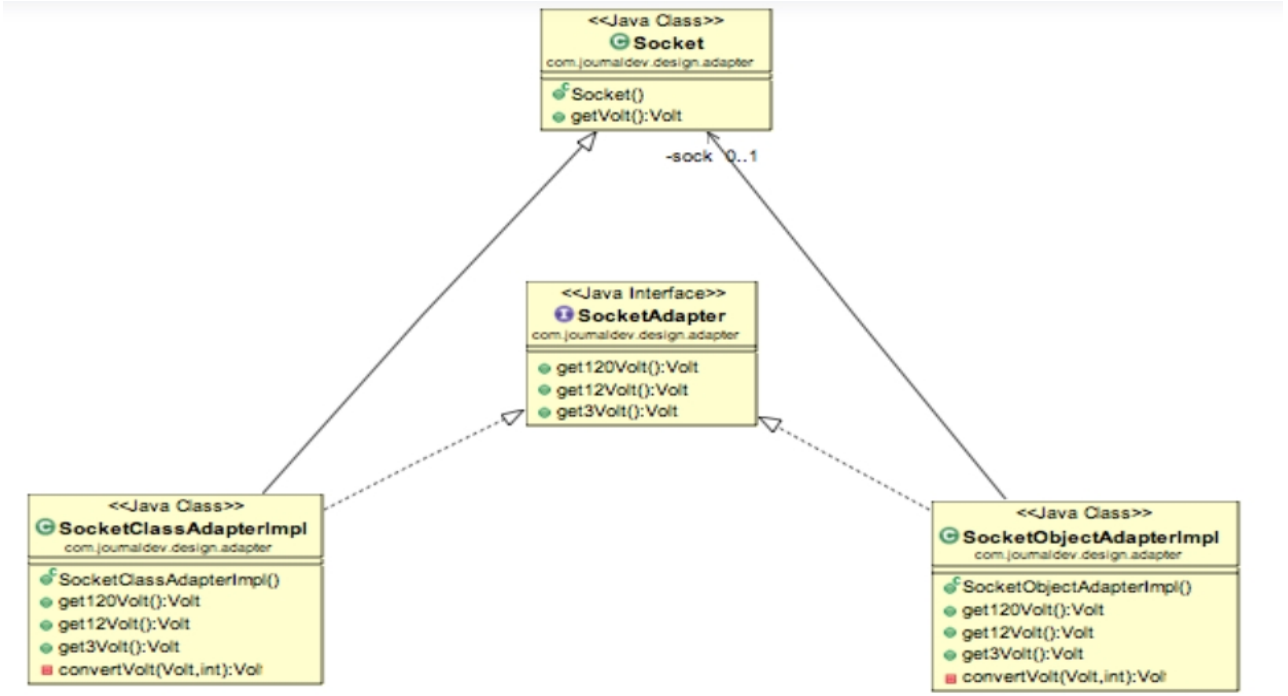
```
        // Criando o contexto com a configuração de acesso específica desejada
        AccessConfigurationContext context = new AccessConfigurationContext(dimepConfig);

        // Configurando o acesso usando a estratégia escolhida
        context.configureAccess();

        // Mudando dinamicamente para outra configuração de acesso
        context.setAccessConfiguration(hikevisionConfig);
        context.configureAccess();
    }
}
```

Neste exemplo, AccessConfiguration é a interface que define o contrato para as estratégias específicas de configuração de acesso.

As classes DimepAccessConfiguration e HikevisionAccessConfiguration são as implementações específicas para os fornecedores DIMEP e HIKEVISION, respectivamente.
A classe AccessConfigurationContext é responsável por manter a estratégia atual e delegar a configuração de acesso para a estratégia correta.

Isso permite que você altere dinamicamente a estratégia de configuração de acesso conforme necessário.

USANDO PADRÃO ADAPTER



Você pode usar o padrão de projeto Adapter para adaptar as interfaces específicas das catracas

(DIMEP e HIKEVISION) para uma interface comum. Isso permite que você trate ambas as catracas da mesma

maneira, independentemente das diferenças em suas interfaces originais. Aqui está um exemplo em Java usando o padrão Adapter:

// Interface comum para a configuração de acesso

interface AccessConfiguration {

```
        void configureAccess();
}

// Implementação específica para DIMEP
class DimepAccessConfiguration {
        void configureDimepAccess() {
                System.out.println("Configurando acesso para DIMEP");
        }
}
// Adapter para DIMEP
class DimepAdapter implements AccessConfiguration {
        private DimepAccessConfiguration dimepConfig;

        public DimepAdapter(DimepAccessConfiguration dimepConfig) {
                this.dimepConfig = dimepConfig;
        }

        @Override
        public void configureAccess() {
                dimepConfig.configureDimepAccess();
        }
}

// Implementação específica para HIKEVISION
class HikevisionAccessConfiguration {
```

```
    void configureHikevisionAccess() {
        System.out.println("Configurando acesso para HIKEVISION");
    }
}

// Adapter para HIKEVISION
class HikevisionAdapter implements AccessConfiguration {
    private HikevisionAccessConfiguration hikevisionConfig;

    public HikevisionAdapter(HikevisionAccessConfiguration hikevisionConfig) {
        this.hikevisionConfig = hikevisionConfig;
    }

    @Override
    public void configureAccess() {
        hikevisionConfig.configureHikevisionAccess();
    }
}

// Exemplo de uso

public class Main {

    public static void main(String[] args) {

        // Criando instâncias das configurações específicas
        DimepAccessConfiguration dimepConfig = new DimepAccessConfiguration();
```

```
        HikevisionAccessConfiguration hikevisionConfig = new HikevisionAccessConfiguration();

        // Criando adaptadores para as configurações específicas
        AccessConfiguration dimepAdapter = new DimepAdapter(dimepConfig);
        AccessConfiguration hikevisionAdapter = new HikevisionAdapter(hikevisionConfig);

        // Configurando o acesso usando os adaptadores
        dimepAdapter.configureAccess();
        hikevisionAdapter.configureAccess();
    }
}
```

Neste exemplo, as classes DimepAdapter e HikevisionAdapter atuam como adaptadores, implementando a interface comum AccessConfiguration e delegando chamadas aos métodos específicos de cada catraca. Isso permite que você use as classes adaptadas de maneira uniforme, independentemente das diferenças nas interfaces originais das catracas DIMEP e HIKEVISION.

Outro Exemplo do ADAPTER

The Adapter design pattern is a structural design pattern that allows two incompatible interfaces to work together. It acts as a bridge between two interfaces, making them compatible and enabling them to work seamlessly. This pattern is useful when you need to integrate existing code or libraries that have different interfaces with your codebase without making significant modifications.

Understanding the Adapter Pattern

The Adapter Pattern involves three key components:

1. **Target Interface:** This is the interface that the client code expects and understands.
2. **Adaptee:** This is the class or interface that you want to adapt to the Target Interface. It is the existing code or component with an incompatible interface.
3. **Adapter:** The Adapter is a class that implements the Target Interface and wraps an instance of the Adaptee, translating calls between them.

Example Scenario: Payment Gateway Integration

Imagine you're building an e-commerce application that supports payments from different payment gateway providers, such as PayPal and Stripe. Each payment gateway has its own unique interface and methods for processing payments. However, you want to create a unified payment processing system in your application that can work with various payment gateway providers without changing your existing code.

Implementing the Adapter Pattern

Step 1: Define a Common Interface

The first step in implementing the Adapter Pattern is to define a common interface that your application understands. In our case, we'll create a `PaymentGateway` interface:

```
public interface PaymentGateway {
    void processPayment(double amount);
}
```

This interface defines a single method, `processPayment`, which is responsible for processing payments.

Step 2: Create Payment Gateway Adapters

Next, we'll create adapter classes for each specific payment gateway provider (PayPal and Stripe) to make them compatible with our `PaymentGateway` interface. Let's start with the PayPal adapter:

PayPal Adapter

```
public class PayPalAdapter implements PaymentGateway {
    private PayPal paymentGateway;

    public PayPalAdapter(PayPal paymentGateway) {
        this.paymentGateway = paymentGateway;
    }

    @Override
    public void processPayment(double amount) {
        // Convert our application's method to PayPal's method
        paymentGateway.makePayment(amount);
    }
}
```

Stripe Adapter

Similarly, we create an adapter for the Stripe payment gateway:

```
public class StripeAdapter implements PaymentGateway {

    private StripePaymentGateway paymentGateway;

    public StripeAdapter(StripePaymentGateway paymentGateway) {
        this.paymentGateway = paymentGateway;
    }

    @Override
    public void processPayment(double amount) {
        // Convert our application's method to Stripe's method
        paymentGateway.charge(amount);
    }
}
```

These adapter classes take the specific payment gateway instances as constructor parameters and implement the `processPayment` method by converting the method calls to the corresponding methods of the payment gateway providers.

Step 3: Implement Concrete Payment Gateway Providers

In our example, we'll create hypothetical implementations of the PayPal and Stripe payment gateways.

PayPal Implementation

```
public class PayPal {
    public void makePayment(double amount) {
        // PayPal-specific payment processing logic
        System.out.println("Paid $" + amount + " via PayPal.");
    }
}
```

Stripe Implementation

```
public class StripePaymentGateway {
    public void charge(double amount) {
        // Stripe-specific payment processing logic
        System.out.println("Charged $" + amount + " using Stripe.");
    }
}
```

These classes represent the specific payment gateway providers and define their unique payment processing logic.

Step 4: Client Code

Now that we have defined the common interface (`PaymentGateway`), created adapter classes for payment gateways, and implemented concrete payment gateway providers, we can use these components in our client code:

```
public class PaymentApp {
    public static void main(String[] args) {
        PaymentGateway paypalGateway = new PayPalAdapter(new PayPal());
        PaymentGateway stripeGateway = new StripeAdapter(new StripePaymentGateway());

        double amount = 100.0;

        // Process payments using different payment gateways
        paypalGateway.processPayment(amount);
        stripeGateway.processPayment(amount);
    }
}
```

In the client code, we create instances of the PayPal and Stripe adapters, passing the corresponding payment gateway instances as parameters. Then, we can use the `processPayment` method to initiate payments through the unified `PaymentGateway` interface.

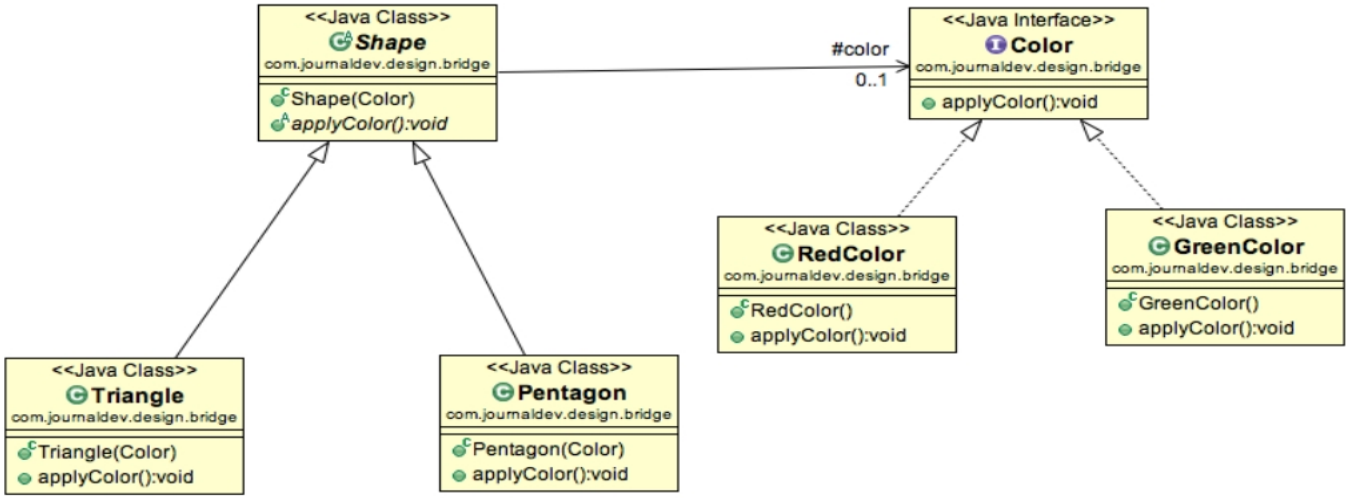
Conclusion

The Adapter Design Pattern is a powerful tool for integrating different systems or components with incompatible interfaces into your application. In our example, we seamlessly integrated payment gateway providers (PayPal and Stripe) into our e-commerce application

without modifying our existing codebase. By creating adapter classes that bridge the gap between the common `PaymentGateway` interface and the specific payment gateway provider interfaces, we achieved a unified and flexible payment processing system.

Whether you're working with third-party services, legacy code, or external libraries, the Adapter Design Pattern can simplify the integration process, promote code reusability, and maintain the integrity of your application's architecture.

USANDO O PADRÃO BRIDGE



Além do padrão de projeto Adapter, você também pode considerar o padrão de projeto Bridge para resolver esse cenário.
O padrão Bridge é projetado para separar uma abstração de sua implementação, permitindo que ambas variem independentemente.

Aqui está um exemplo em Java usando o padrão Bridge para lidar com configurações de acesso de catracas de fornecedores diferentes:

```
// Implementação da configuração de acesso para DIMEP
interface DimepAccessConfiguration {
	void configureDimepAccess();
}

class DimepAccessConfigurationImpl implements DimepAccessConfiguration {
	@Override
	public void configureDimepAccess() {
		System.out.println("Configurando acesso para DIMEP");
```

```
        }
}

// Implementação da configuração de acesso para HIKEVISION
interface HikevisionAccessConfiguration {
        void configureHikevisionAccess();
}

class HikevisionAccessConfigurationImpl implements HikevisionAccessConfiguration {
        @Override
        public void configureHikevisionAccess() {
                System.out.println("Configurando acesso para HIKEVISION");
        }
}

// Abstração para a configuração de acesso
interface AccessConfiguration {
        void configureAccess();
}

// Implementação da abstração usando Bridge
class AccessConfigurationBridge implements AccessConfiguration {
        private DimepAccessConfiguration dimepConfig;
        private HikevisionAccessConfiguration hikevisionConfig;

        public AccessConfigurationBridge(DimepAccessConfiguration dimepConfig) {
                this.dimepConfig = dimepConfig;
        }
```

```
    public AccessConfigurationBridge(HikevisionAccessConfiguration hikevisionConfig) {
        this.hikevisionConfig = hikevisionConfig;
    }

    @Override
    public void configureAccess() {
        if (dimepConfig != null) {
            dimepConfig.configureDimepAccess();
        } else if (hikevisionConfig != null) {
            hikevisionConfig.configureHikevisionAccess();
        }
    }
}

// Exemplo de uso
public class Main {
    public static void main(String[] args) {

        // Criando instâncias das configurações específicas
        DimepAccessConfiguration dimepConfig = new DimepAccessConfigurationImpl();
        HikevisionAccessConfiguration hikevisionConfig = new HikevisionAccessConfigurationImpl();

        // Criando instâncias da abstração usando Bridge
        AccessConfiguration dimepBridge = new AccessConfigurationBridge(dimepConfig);
        AccessConfiguration hikevisionBridge = new AccessConfigurationBridge(hikevisionConfig);

        // Configurando o acesso usando as instâncias da abstração
        dimepBridge.configureAccess();
        hikevisionBridge.configureAccess();
```

```
        }
}
```

Neste exemplo, DimepAccessConfiguration e HikevisionAccessConfiguration são interfaces que representam as implementações específicas para DIMEP e HIKEVISION, respectivamente.

AccessConfigurationBridge é uma classe de abstração que usa a implementação Bridge para delegar a chamada ao método apropriado com base na implementação específica fornecida.

Ambos os padrões Adapter e Bridge podem ser aplicados para resolver esse problema, e a escolha entre eles dependerá das características específicas do seu sistema e dos requisitos do projeto.

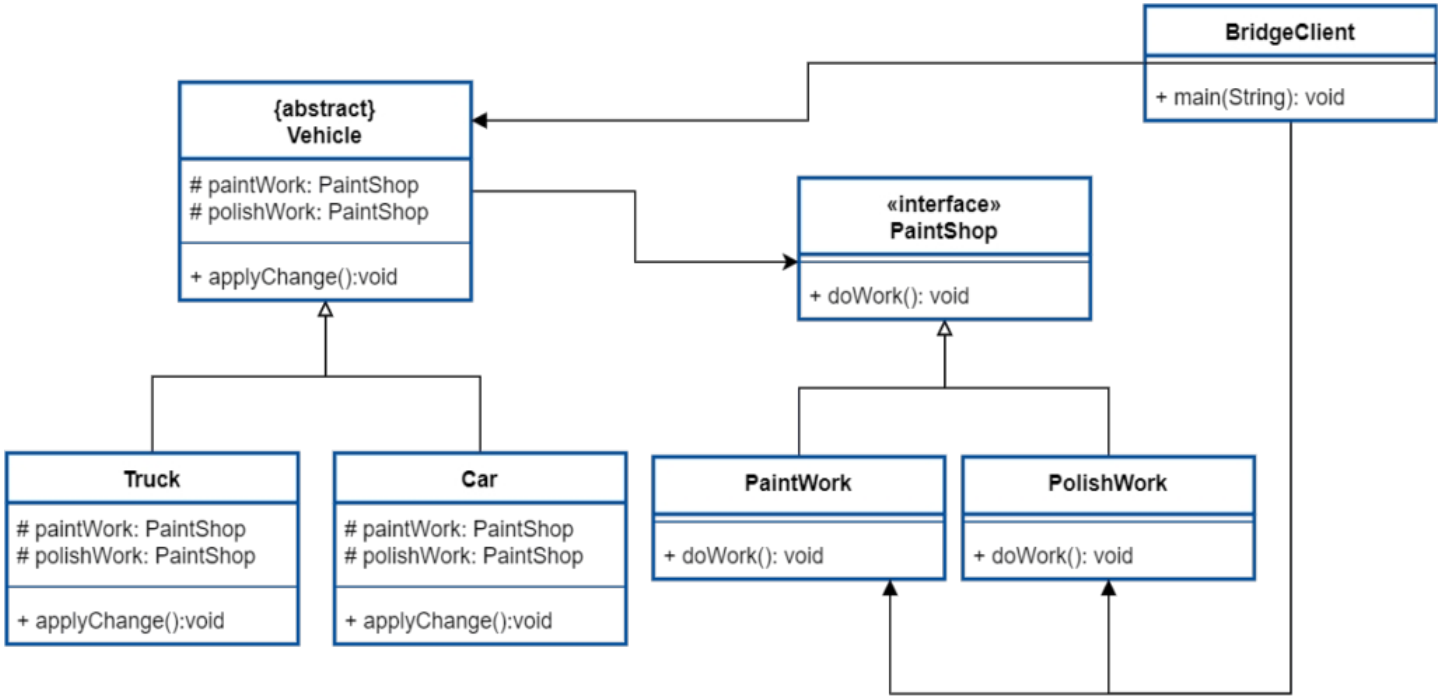
OUTRO EXEMPLO COM BRIDGE

Bridge Design Pattern is one of the main ***Design Patterns*** focusing on decoupling abstraction from implementation. The concept helps to grow implementation and abstraction separately without coupling each other. Also, Bridge Design Pattern hides the implementation from clients.

Bridge Design Pattern comes under structural design patterns. It increases the loose couple between abstraction and implementation. Just use the bridge to do the connection between them. Let's take an example now.

Bridge Design Pattern Real World Example

Consider Vehicle Paint Shop which provides services such as Painting and Polishing. Different types of vehicles come to the shop, but shops do not depend on the vehicle types just doing the paintings and polishing. In this case, has to decouple the vehicles and shop services. Let’s move on to the class diagram of the paint shop as a real-world problem.



Bridge Design Pattern for Vehicle PaintShop

According to the diagram, the paint shop could have many services(Paintwork, Polish work), and vehicles come to the shop to get some services. According to the example services are independent and the abstract vehicle creates the connection between Vehicle and Paint Shop.

Bridge Design Pattern In Java

https://www.ennicode.com/bridge-design-pattern/

Bridge Design patterns can divide into the abstraction side and the implementation side. But According to the diagram, there are four main participants in the design pattern. Let's see what are the participants in this pattern

- Abstraction: Define the functionality (PaintShop)
- Bridge: Connection between abstraction side and implementation side (Vehicle)
- Implementation: Implement the functionality defined by the abstraction. (PaintWork, PolishWork)
- Concrete Implementations: Use the Bridge and Complete the work using abstract implementation (Truck, Car)

In this example, There are Trucks and Cars inherited from the Vehicle class. But there can be many vehicle types. On another side, there is an interface as PaintShop and there are PaintWork and PolishWork inherited from PaintShop. New services are able to add to the paint shop without affecting the system.

```
// Define the work of Paint Shop
interface PaintShop{
   void doWork();
}

//Implements the services that Paint shop Provide
class PaintWork implements PaintShop{
   @Override
   public void doWork(){
      System.out.println("Paint Shop doing paint works");
   }
}

class PolishWork implements PaintShop{
   @Override
   public void doWork(){
      System.out.println("Paint Shop doing polish works");
   }
}

// Bridge the PaintShop and Vehicles
abstract class Vehicle{

   protected  PaintShop paintWork;
   protected  PaintShop polishWork;
   public Vehicle(PaintShop paintWork,PaintShop polishWork) {
```

```
      this.paintWork=paintWork;
      this.polishWork=polishWork;
   }

   public abstract void applyChange();

}

// Get the required services from PaintShop
class Truck extends Vehicle{
   public Truck(PaintShop paintWork,PaintShop polishWork) {
     super(paintWork,polishWork);
   }
   @Override
   public void applyChange(){
      System.out.println("Truck work going to start");
      paintWork.doWork();
      polishWork.doWork();
   }
}

class Car extends Vehicle{
   public Car(PaintShop paintWork,PaintShop polishWork) {
     super(paintWork,polishWork);
   }

   @Override
   public void applyChange(){
      System.out.println("Car work going to start");
      paintWork.doWork();
```

```
      polishWork.doWork();
   }
}

public class BridgeClient {

   public static void main(String[] args) {
      Vehicle vehicle1= new Truck(new PaintWork(), new PolishWork());
      vehicle1.applyChange();
      Vehicle vehicle2= new Car(new PaintWork(), new PolishWork());
      vehicle2.applyChange();
   }
}
```

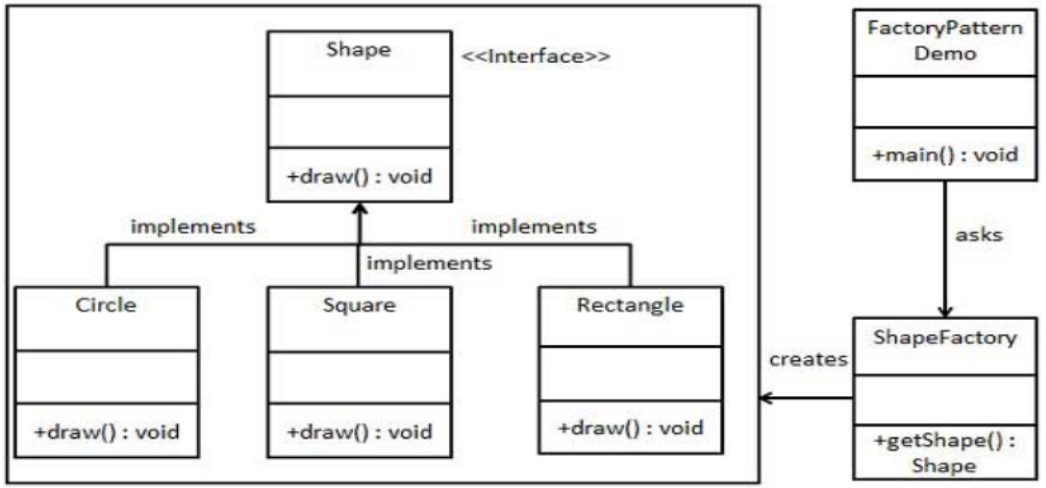
Output

```
Truck work going to start
Paint Shop doing paint works
Paint Shop doing polish works
Car work going to start
Paint Shop doing paint works
Paint Shop doing polish works
```

Advantages Of Bridge Design Pattern

- Bridge Design Pattern use to decouple an abstraction from implementation.
- Hide the implementation from clients.
- Possible to individually improve the interface side and implementation.
- Possible to implement as runtime binding objects.

USANDO O PADRÃO FACTORY



Este padrão é útil quando você tem uma interface comum, mas as subclasses precisam ser instanciadas de maneira diferente. No seu caso, você poderia ter uma fábrica para criar objetos de configuração de acesso com base no fornecedor da catraca.

```
// Interface para a configuração de acesso
interface AccessConfiguration {
        void configureAccess();
}

// Implementação específica para DIMEP
class DimepAccessConfiguration implements AccessConfiguration {
        @Override
```

```
        public void configureAccess() {
                System.out.println("Configurando acesso para DIMEP");
        }
}

// Implementação específica para HIKEVISION
class HikevisionAccessConfiguration implements AccessConfiguration {
        @Override
        public void configureAccess() {
                System.out.println("Configurando acesso para HIKEVISION");
        }
}

// Interface da fábrica
interface AccessConfigurationFactory {
        AccessConfiguration createAccessConfiguration();
}

// Fábrica para DIMEP
class DimepAccessConfigurationFactory implements AccessConfigurationFactory {
        @Override
        public AccessConfiguration createAccessConfiguration() {
                return new DimepAccessConfiguration();
        }
}
```

```
// Fábrica para HIKEVISION
class HikevisionAccessConfigurationFactory implements AccessConfigurationFactory {
	@Override
	public AccessConfiguration createAccessConfiguration() {
		return new HikevisionAccessConfiguration();
	}
}

// Exemplo de uso
public class Main {

	public static void main(String[] args) {

		// Criando instâncias das fábricas específicas
		AccessConfigurationFactory dimepFactory = new DimepAccessConfigurationFactory();
		AccessConfigurationFactory hikevisionFactory = new HikevisionAccessConfigurationFactory();

		// Criando configurações de acesso usando as fábricas
		AccessConfiguration dimepConfig = dimepFactory.createAccessConfiguration();
		AccessConfiguration hikevisionConfig = hikevisionFactory.createAccessConfiguration();

		// Configurando o acesso
		dimepConfig.configureAccess();
		hikevisionConfig.configureAccess();
	}
}
```

Neste exemplo, AccessConfigurationFactory é a interface da fábrica que declara o método createAccessConfiguration(). As classes DimepAccessConfigurationFactory e HikevisionAccessConfigurationFactory são as implementações concretas

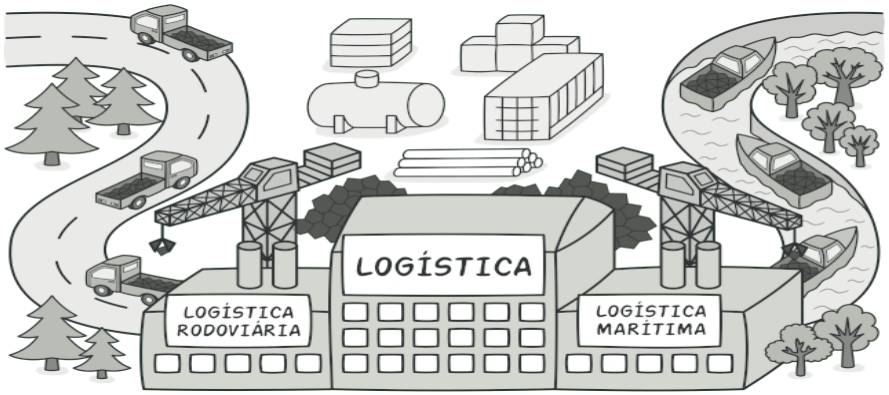
dessa interface que criam instâncias das configurações específicas para DIMEP e HIKEVISION.

O padrão Factory Method permite que você adicione novas implementações de configuração de acesso no futuro sem modificar o código existente. Cada nova implementação de configuração teria sua própria fábrica correspondente.

Outro Exemplo Factory Method

Propósito

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.



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**.

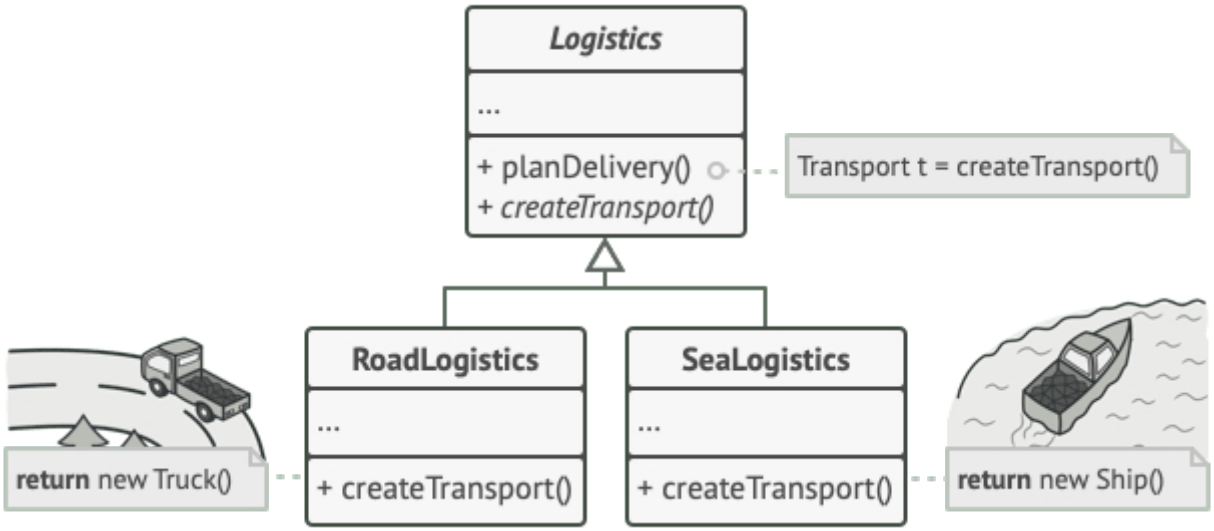
Depois de um tempo, sua aplicação se torna bastante popular. Todos os dias você recebe dezenas de solicitações de empresas de transporte marítimo para incorporar a logística marítima na aplicação.

Adicionar uma nova classe ao programa não é tão simples se o restante do código já estiver acoplado às classes existentes. Boa notícia, certo? Mas e o código? 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

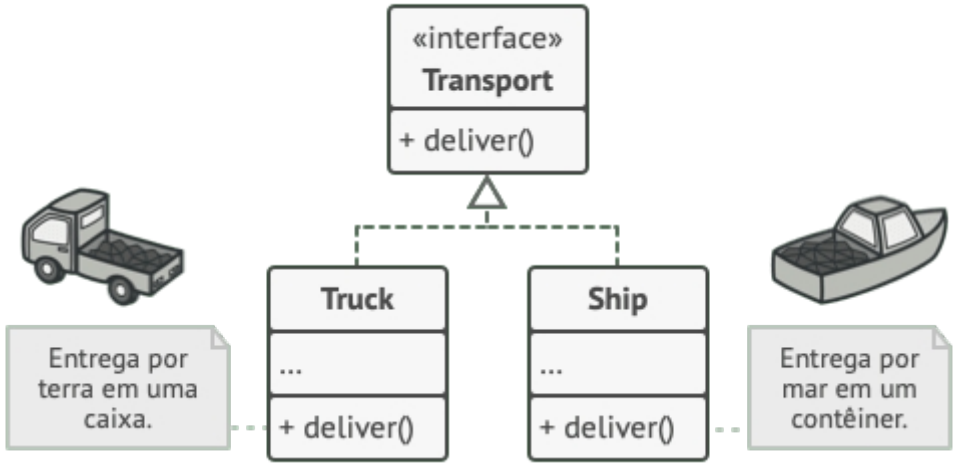
O padrão Factory Method sugere que você substitua chamadas diretas de construção de objetos (usando o operador new) por chamadas para um método *fábrica* especial. Não se preocupe: os objetos ainda são criados através do operador new, mas esse está sendo chamado de dentro do método fábrica. Objetos retornados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



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

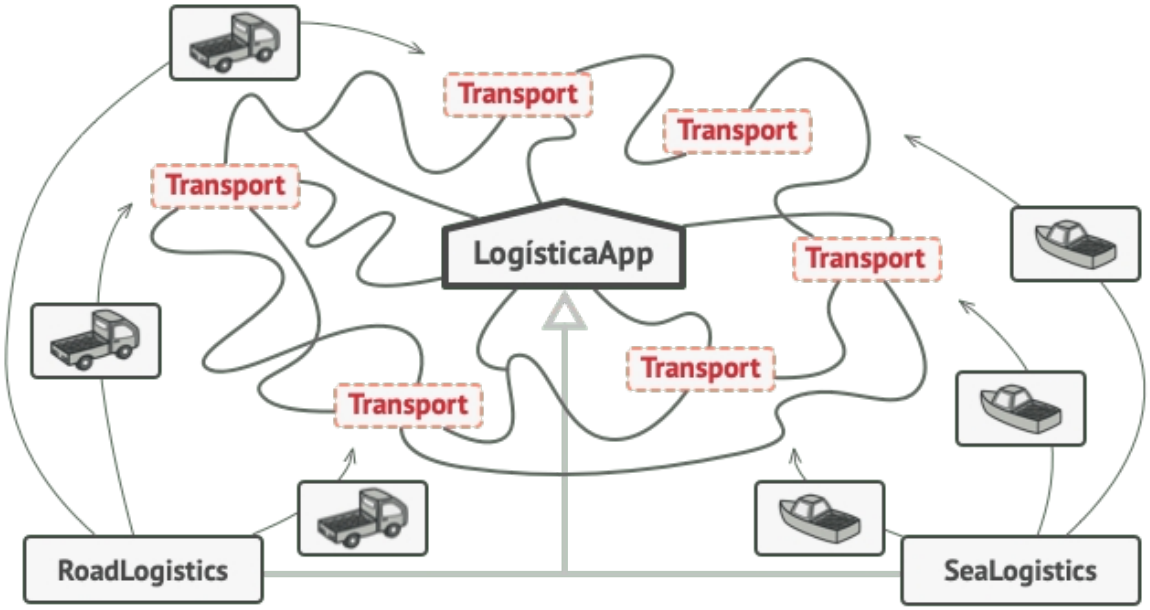
À primeira vista, essa mudança pode parecer sem sentido: apenas mudamos a chamada do construtor de uma parte do programa para outra. No entanto, considere o seguinte: agora você pode sobrescrever o método fábrica em uma subclasse e alterar a classe de produtos que estão sendo criados pelo método.

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.

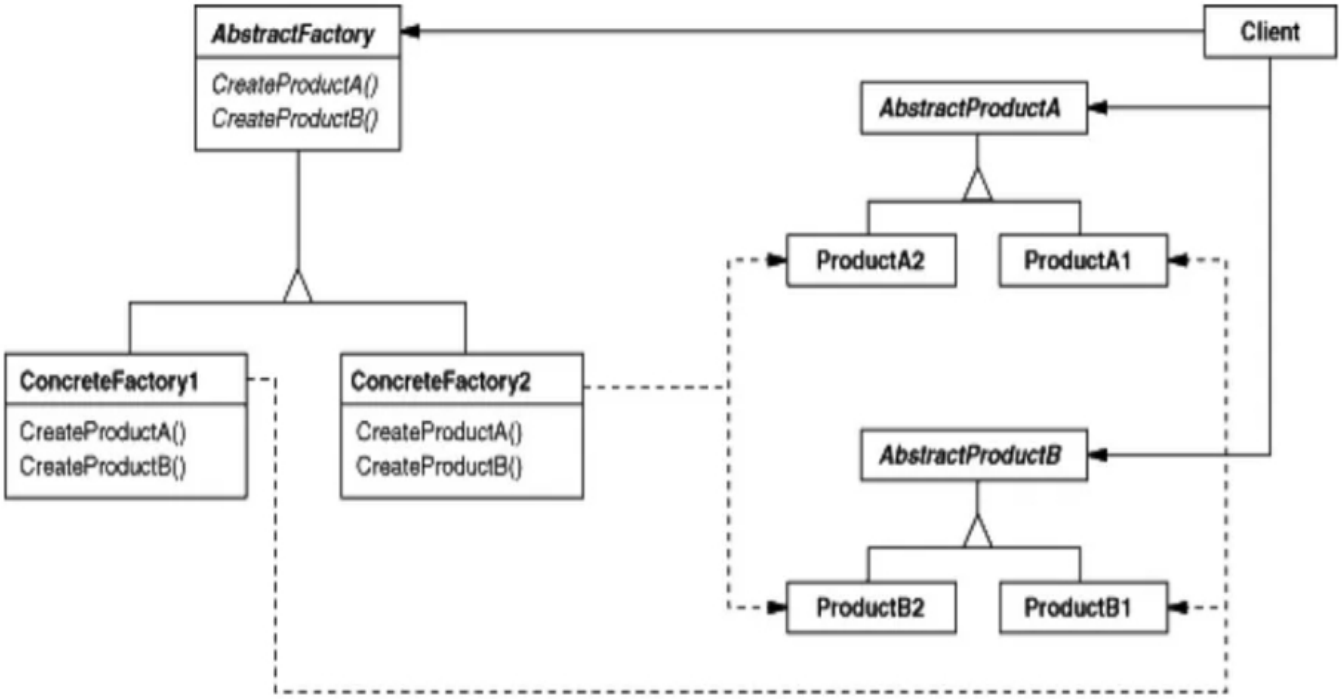
Por exemplo, ambas as classes **Caminhão** e **Navio** devem implementar a interface **Transporte**, que declara um método chamado **entregar**. Cada classe implementa esse método de maneira diferente: caminhões entregam carga por terra, navios entregam carga por mar. O método fábrica na classe **LogísticaViária**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

USANDO ABSTRACT FACTORY



O padrão Abstract Factory fornece uma interface para criar famílias de objetos relacionados ou dependentes sem especificar suas classes concretas.

```
// Interface para a configuração de acesso
interface AccessConfiguration {
	void configureAccess();
}

// Implementação específica para DIMEP
class DimepAccessConfiguration implements AccessConfiguration {
	@Override
	public void configureAccess() {
		System.out.println("Configurando acesso para DIMEP");
	}
}

// Implementação específica para HIKEVISION
class HikevisionAccessConfiguration implements AccessConfiguration {
	@Override
	public void configureAccess() {
		System.out.println("Configurando acesso para HIKEVISION");
	}
}

// Interface da fábrica abstrata
interface AccessConfigurationAbstractFactory {
	AccessConfiguration createAccessConfiguration();
}
```

```
// Fábrica abstrata para DIMEP
class DimepAccessConfigurationFactory implements AccessConfigurationAbstractFactory {
	@Override
	public AccessConfiguration createAccessConfiguration() {
		return new DimepAccessConfiguration();
	}
}

// Fábrica abstrata para HIKEVISION
class HikevisionAccessConfigurationFactory implements AccessConfigurationAbstractFactory {
	@Override
	public AccessConfiguration createAccessConfiguration() {
		return new HikevisionAccessConfiguration();
	}
}

// Cliente que usa a fábrica abstrata
class AccessConfigurationClient {
	private AccessConfigurationAbstractFactory factory;

	public AccessConfigurationClient(AccessConfigurationAbstractFactory factory) {
		this.factory = factory;
	}

	public void configureAccess() {
		AccessConfiguration config = factory.createAccessConfiguration();
		config.configureAccess();
	}
}
```

```
// Exemplo de uso
public class Main {

	public static void main(String[] args) {

		// Criando instâncias das fábricas específicas
		AccessConfigurationAbstractFactory dimepFactory = new DimepAccessConfigurationFactory();
		AccessConfigurationAbstractFactory hikevisionFactory = new HikevisionAccessConfigurationFactory();

		// Criando clientes que usam as fábricas
		AccessConfigurationClient dimepClient = new AccessConfigurationClient(dimepFactory);
		AccessConfigurationClient hikevisionClient = new AccessConfigurationClient(hikevisionFactory);

		// Configurando o acesso usando os clientes
		dimepClient.configureAccess();
		hikevisionClient.configureAccess();
	}
}
```

Neste exemplo, AccessConfigurationAbstractFactory é a interface abstrata da fábrica que declara o método createAccessConfiguration(). As classes DimepAccessConfigurationFactory e HikevisionAccessConfigurationFactory são as implementações concretas dessa interface que criam instâncias das configurações específicas para DIMEP e HIKEVISION.

O cliente (AccessConfigurationClient) utiliza a fábrica abstrata para criar uma instância da configuração de acesso e, em seguida, configura o acesso.
O padrão Abstract Factory permite que você crie famílias de objetos relacionados sem especificar suas classes concretas, tornando o código mais flexível para extensão.

USANDO SINGLETON PATTERN

Definição de Singleton em Java: Um Singleton é um padrão de design que garante que uma classe tenha apenas uma instância e fornece um ponto global para acessar essa instância. Ele é útil quando precisamos de exatamente uma instância de uma classe para coordenar ações em todo o sistema.

Benefícios do Singleton:

1. **Controle de Acesso:** Fornece um ponto de acesso global para a única instância, permitindo um controle mais fácil sobre a inicialização e acesso à instância.
2. **Conservação de Recursos:** Evita a criação de múltiplas instâncias, economizando recursos do sistema.
3. **Coordenação Centralizada:** Facilita a coordenação de operações em toda a aplicação, pois há apenas uma instância central.
4. **Lazy Initialization:** Pode ser configurado para inicializar a instância apenas quando necessária, melhorando o desempenho.

```
public class Singleton {

    // A instância única da classe
    private static Singleton instance;

    // Construtor privado para evitar a criação de instâncias fora da classe
    private Singleton() {
        // Inicializações, se necessário
    }
```

```
    // Método para obter a instância única (criando-a, se necessário)
    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }

    // Métodos da classe, se houver
}

package org.example.singleton;

public class SingletonRunner {

        public static void main(String[] args) {
            // Obtendo a instância única do Singleton
            Singleton singletonInstance = Singleton.getInstance();

            // Agora você pode usar a instância conforme necessário
            // ...
```

```
        //  Testando se será criada uma nova instância ou será
        //  usada a instância já existente
        Singleton singletonInstance1 = Singleton.getInstance();

        // Exemplo: Imprimir alguma informação da instância
        System.out.println("Hashcode da instância: " + singletonInstance.hashCode());
    }
}
```

- O construtor é privado para impedir a criação de instâncias fora da classe.
- **`getInstance()`** é o método estático que fornece a única instância da classe, criando-a se ainda não existir (técnica chamada de "Lazy Initialization").
- O uso do **`synchronized`** na verificação dupla dentro de **`getInstance()`** ajuda a evitar problemas de concorrência quando várias threads tentam acessar a instância simultaneamente.

É importante observar que, com o avanço das versões do Java, existem outras formas mais modernas de implementar Singletons, como o uso de Enums ou a utilização de inicialização estática garantida pela JVM. Essas abordagens podem ser mais simples e seguras em ambientes multithread.

USANDO FACHADA PATTERN

Definição de Fachada em Java: A Fachada (Facade) é um padrão de design estrutural que fornece uma interface unificada para um conjunto de interfaces em um subsistema. Ele define uma interface de nível mais alto que torna o subsistema mais fácil de ser usado, proporcionando uma única entrada simplificada para a funcionalidade mais complexa do subsistema.

Benefícios da Fachada:

1. **Simplicidade de Uso:** A Fachada fornece uma interface simplificada, ocultando a complexidade do subsistema e tornando mais fácil para os clientes interagirem com ele.
2. **Desacoplamento:** Os clientes interagem com a Fachada, não com os componentes internos do subsistema. Isso permite que o subsistema evolua sem afetar os clientes.
3. **Abstração:** A Fachada fornece uma camada de abstração sobre o subsistema, isolando os detalhes internos e proporcionando uma visão mais clara da funcionalidade.
4. **Manutenção Facilitada:** Alterações no subsistema podem ser feitas na Fachada, mantendo a interface externa inalterada. Isso simplifica a manutenção e evolução do sistema.

Exemplo de Fachada em Java: Suponha que você tenha um sistema com vários subsistemas complexos, como um subsistema de gerenciamento de pedidos, um subsistema de estoque e um subsistema de faturamento. **A Fachada pode ser usada para simplificar a interação com esses subsistemas.**

Aqui está um exemplo simplificado em Java:

```
// Subsistema 1: Gerenciamento de Pedidos
class OrderSystem {
    public void placeOrder() {
        // Lógica para colocar um pedido
        System.out.println("Pedido colocado com sucesso.");
    }
}

// Subsistema 2: Controle de Estoque
class StockSystem {
    public void checkStock() {
        // Lógica para verificar o estoque
        System.out.println("Estoque verificado.");
    }
}

// Subsistema 3: Faturamento
class BillingSystem {
    public void generateInvoice() {
        // Lógica para gerar fatura
        System.out.println("Fatura gerada com sucesso.");
    }
}

// Fachada
class OnlineShoppingFacade {
    private OrderSystem orderSystem;
    private StockSystem stockSystem;
    private BillingSystem billingSystem;
```

```
    public OnlineShoppingFacade() {
        this.orderSystem = new OrderSystem();
        this.stockSystem = new StockSystem();
        this.billingSystem = new BillingSystem();
    }

    // Métodos simplificados para o cliente interagir com o sistema
    public void buyProduct() {
        orderSystem.placeOrder();
        stockSystem.checkStock();
        billingSystem.generateInvoice();
    }
}

// Cliente
public class Main {
    public static void main(String[] args) {
        // Usando a Fachada para simplificar a interação com o sistema
        OnlineShoppingFacade facade = new OnlineShoppingFacade();
        facade.buyProduct();
    }
}
```

USANDO COMMAND PATTERN

O padrão Command é um padrão de design comportamental que converte uma solicitação ou operação em um objeto independente. Isso permite parametrizar clientes com diferentes solicitações, atrasar a execução de uma solicitação e suportar operações que podem ser desfeitas.

Vamos criar um exemplo simples para o padrão Command:

```
package org.example.command;

public interface CommandInterface {

void executar();

}
```

```
package org.example.command.actions;

// Receiver public class Luz {

public void ligar() {
   System.out.println("Luz ligada");
}
```

```
public void desligar() {
    System.out.println("Luz desligada");
}

}
```

```
package org.example.command.commands;

import org.example.command.RemoteControl; import org.example.command.actions.Luz;

package org.example.command;

// Invoker public class RemoteControl {

private CommandInterface comando;

public void setComando(CommandInterface comando) {
    this.comando = comando;
}

public void pressionarBotao() {
    comando.executar();
}

}

// Cliente public class RemoteControlRunner {
```

```
public static void main(String[] args) {

    Luz minhaLuz = new Luz();
    CommandLigarLuz ligarLuz = new CommandLigarLuz(minhaLuz);
    CommandDesligarLuz desligarLuz = new CommandDesligarLuz(minhaLuz);

     RemoteControl controle = new RemoteControl();

    controle.setComando(ligarLuz);
    controle.pressionarBotao();

    controle.setComando(desligarLuz);
    controle.pressionarBotao();
}

}
```

```
package org.example.command.commands;

import org.example.command.CommandInterface; import org.example.command.actions.Luz;

public class CommandDesligarLuz implements CommandInterface {

private Luz luz;

CommandDesligarLuz(Luz luz) {
    this.luz = luz;
```

```
}

@Override
public void executar() {
    luz.desligar();
}

}
```

```
package org.example.command.commands;

import org.example.command.CommandInterface; import org.example.command.actions.Luz;

public class CommandLigarLuz implements CommandInterface {

private Luz luz;

CommandLigarLuz(Luz luz) {
    this.luz = luz;
}

@Override
public void executar() {
    luz.ligar();
}

}
```

Estudos João Caboclo S. Filho