DESCRIÇÃO

Apresentação de padrões GoF de projeto estruturais: Adapter, Bridge, Decorator, Composite, Facade, Flyweight, Proxy.

PROPÓSITO

Compreender os padrões de projeto GoF ligados à estruturação de objetos e identificar oportunidades para a sua aplicação na programação orientada a objetos são habilidades importantes para um projetista de software, pois, sem elas, as soluções geradas podem se tornar inflexíveis e dificultar a evolução dos sistemas em prazo e custo aceitáveis.

PREPARAÇÃO

Antes de iniciar o conteúdo, é recomendado instalar em seu computador um programa que permita elaborar modelos sob a forma de diagramas da UML (Linguagem Unificada de Modelagem). Nossa sugestão inicial é o Free Student License for Astah UML, usado nos exemplos deste estudo. Para isso, será necessário usar seu e-mail institucional para ativar a licença.

Preencha os dados do formulário no site do software, envie e aguarde a liberação de sua licença em seu e-mail institucional. Ao receber a licença, siga as instruções do e-mail e instale o produto em seu computador. Os arquivos Astah com diagramas UML utilizados neste conteúdo estão disponíveis para download.

Sugestões de links adicionais de ferramentas livres para modelagem de sistemas em UML (UML Tools) podem ser encontradas em buscas na Internet.

Além disso, recomendamos a instalação de um ambiente de programação em Java. O ambiente recomendado para iniciantes em Java é o Apache Netbeans, cujo instalador pode ser encontrado no site do ambiente, acessando o menu Download. Porém, antes de instalar o

Netbeans, é necessário ter instalado o JDK (Java Development Kit) referente à edição Java SE (Standard Edition), que pode ser encontrado no site da Oracle Technology Network: Java SE - Downloads | Oracle Technology Network | Oracle.

OBJETIVOS

MÓDULO 1

Reconhecer o propósito, a estrutura e as situações de aplicação do padrão de projeto Adapter

MÓDULO 2

Reconhecer o propósito, a estrutura e as situações de aplicação dos padrões de projeto Bridge e Decorator

MÓDULO 3

Reconhecer o propósito, a estrutura e as situações de aplicação dos padrões de projeto Composite e Facade

MÓDULO 4

Reconhecer o propósito, a estrutura e as situações de aplicação dos padrões de projeto Flyweight e Proxy

INTRODUÇÃO

Os padrões GoF, do inglês “Gang of Four”, são padrões de projeto orientados a objetos divididos em três categorias: de criação, estruturais e comportamentais. São assim denominados por terem sido introduzidos pelos quatro autores do livro *Design Patterns: Elements of Reusable Object-Oriented Software* (GAMMA *et al* ., 1994). Os padrões de projeto GoF estruturais descrevem estratégias para compor classes e objetos de modo a compor estruturas maiores, que podem ser formadas com soluções baseadas em herança ou em composição de objetos. Neste conteúdo, você aprenderá os sete padrões deste grupo: Adapter, Bridge, Decorator, Composite, Facade, Flyweight e Proxy.

O padrão Adapter permite que módulos de um sistema possam interagir com diferentes implementações de um mesmo serviço por meio de uma interface genérica. O padrão Bridge separa a abstração de um objeto da sua implementação, de modo que ambos possam variar de forma independente. O padrão Decorator descreve uma maneira de adicionar, dinamicamente, responsabilidades a objetos por meio de estruturas de composição, em vez de soluções baseadas em herança. O padrão Composite descreve como podemos estruturar uma hierarquia de objetos construída a partir de dois tipos de elementos: primitivos e compostos. O padrão Facade propõe a criação de uma interface de alto nível para um subsistema, de modo a isolar os módulos clientes do conhecimento da estrutura interna desse subsistema.

O padrão Flyweight apresenta uma solução para o problema de compartilhamento de um número elevado de pequenos objetos, com o objetivo de utilizar os recursos de memória de forma mais racional. Por fim, o padrão Proxy consiste na criação de um objeto substituto do real destino de uma requisição, sendo útil para abstrair a complexidade na comunicação entre objetos distribuídos ou para lidar com a criação sob demanda de um objeto com grande consumo de recursos, como imagens, por exemplo.

MÓDULO 1

⦿ **Reconhecer o propósito, a estrutura e as situações de aplicação do padrão de projeto Adapter**

INTENÇÃO DO PADRÃO ADAPTER

Adapter é um padrão cujo propósito é converter a interface de uma classe existente em outra interface esperada pelos módulos clientes.

O padrão Adapter é especialmente útil quando estamos desenvolvendo um sistema que precisa interagir com um serviço implementado por terceiros, que forneçam componentes e interfaces diferentes e incompatíveis entre si.

PROBLEMA RESOLVIDO PELO PADRÃO ADAPTER

Imagine um turista que esteja viajando do Brasil para os EUA e depois para alguns países da Europa.

Ele certamente vai precisar carregar o seu telefone celular durante a viagem, não é mesmo?

Entretanto, o padrão de tomada brasileiro é diferente do norte-americano e do europeu. O que ele precisa levar na bagagem para poder plugar o seu carregador em tomadas com diferentes padrões?

Isso mesmo! Ele vai precisar de um ou mais adaptadores de tomada.

Mas qual é a relação entre tomadas e o desenvolvimento de software?

Suponha que você esteja desenvolvendo um sistema para vendas on-line de diferentes varejistas, sendo que cada loja pode utilizar uma ou mais das soluções de pagamento disponíveis no mercado.

Cada fornecedor de solução de pagamento permite, por exemplo, a realização de uma transação em cartão de crédito, por meio de uma chamada à sua API de pagamento.

Portanto, o software de vendas deve ser capaz de ser plugado a diferentes APIs de pagamento que oferecem basicamente o mesmo serviço: intermediar as diversas formas de pagamento existentes no mercado.

O problema é que cada API tem uma interface proprietária definida pelo fornecedor.

Gostaríamos que nosso software de vendas pudesse trabalhar com novos fornecedores que apareçam no mercado, sem que seja necessário alterar módulos já existentes, mas apenas adicionando novos módulos.

O código a seguir apresenta um exemplo hipotético e simplificado de duas soluções de pagamento disponíveis no mercado.

A primeira oferece uma classe de nome **PagXPTO** com duas operações distintas para o pagamento, conforme a bandeira do cartão (Visa ou Mastercard).

A segunda fornece a classe **ABCPagamentos** com apenas uma operação de pagamento.

Perceba que, além de nomes distintos, os parâmetros dessas operações também são diferentes.

```
// Solução de pagamentos do fornecedor XPTO
public class PagXPTO {
public void pagarCartaoVisa (DadosCartao cartao, BigDecimal valor);
public void pagarCartaoMastercard (DadosCartao cartao, BigDecimal valor);
}
// Solução de pagamentos do fornecedor ABC
public class ABCPagamentos {
public void pagarEmCartaoCredito (String numeroCartao, String nome, String CVV, String validade, BigDecimal valor);
}
```

Sem a utilização do padrão Adapter, um módulo cliente que precisasse chamar uma API de pagamento poderia ser implementado da forma esquemática ilustrada pelo código a seguir.

```
public class ServicoPagamento {
// o parâmetro nomeBroker define a API a ser chamada: XPTO ou ABC
public void pagarCartaoCredito (String nomeBroker, CartaoCredito cartao, BigDecimal valor) {
if (“XPTO”.equals(nomeBroker)) {
pagarCartaoXPTO(cartao, valor);
} else if (“ABC”.equals(nomeBroker)) {
pagarCartaoABC(cartao, valor);
}
}

// chamada à API de pagamento do fornecedor XPTO
private void pagarCartaoXPTO(CartaoCredito cartao, BigDecimal valor) {
PagXPTO brokerPagamento = new PagXPTO();
// converte o parâmetro cartao para a estrutura requerida pela API
DadosCartao dadosCartao = converterCartao(cartao);
// com base no número do cartão, define a função da API a ser chamada
if (isCartaoVisa(cartao)) {
brokerPagamento.pagarCartaoVisa(dadosCartao, valor);
} else {
brokerPagamento.pagarCartaoMastercard(dadosCartao, valor);
}
```

```
}

// chamada à API de pagamento do fornecedor ABC
private void pagarCartaoABC(CartaoCredito cartao, BigDecimal valor) {
ABCPagamentos brokerPagamento = new ABCPagamentos();
brokerPagamento.pagarEmCartaoCredito(cartao.numero(), cartao.nome(),
cartao.cvv(), cartao.validade(), valor);
}
}
```

Note que, como cada API possui uma interface específica, o código para a realização do pagamento em cartão de crédito precisa ser específico para cada fornecedor, sendo implementado pelos métodos pagarCartaoXPTO e pagarCartaoABC.

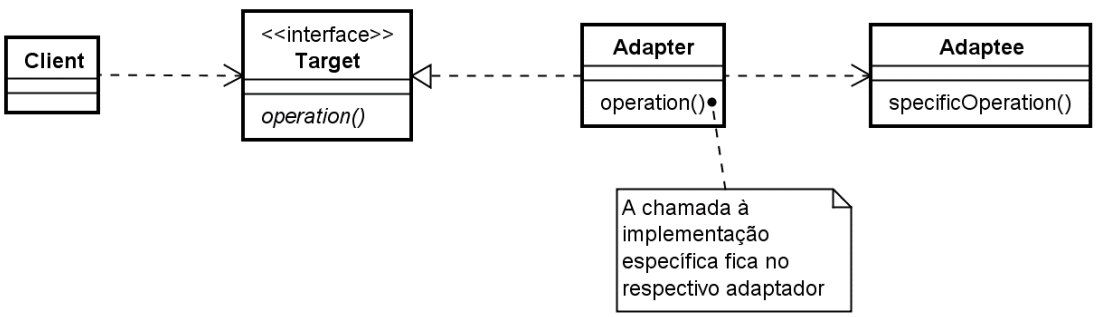
Você consegue visualizar como ficaria esse código com dezenas de fornecedores e de operações além do pagamento em cartão de crédito?

O resultado seria um código enorme, complexo e teria que ser modificado a cada nova operação ou fornecedor.

Portanto, na analogia com o carregador de telefone celular, o módulo cliente corresponde ao carregador, enquanto os fornecedores ABC e XPTO correspondem aos diferentes tipos de tomada. Precisamos de uma solução que nos permita conectar o módulo cliente com qualquer fornecedor de serviço de pagamento.

SOLUÇÃO DO PADRÃO ADAPTER

O Adapter é um padrão cuja estrutura é apresentada no diagrama UML a seguir.

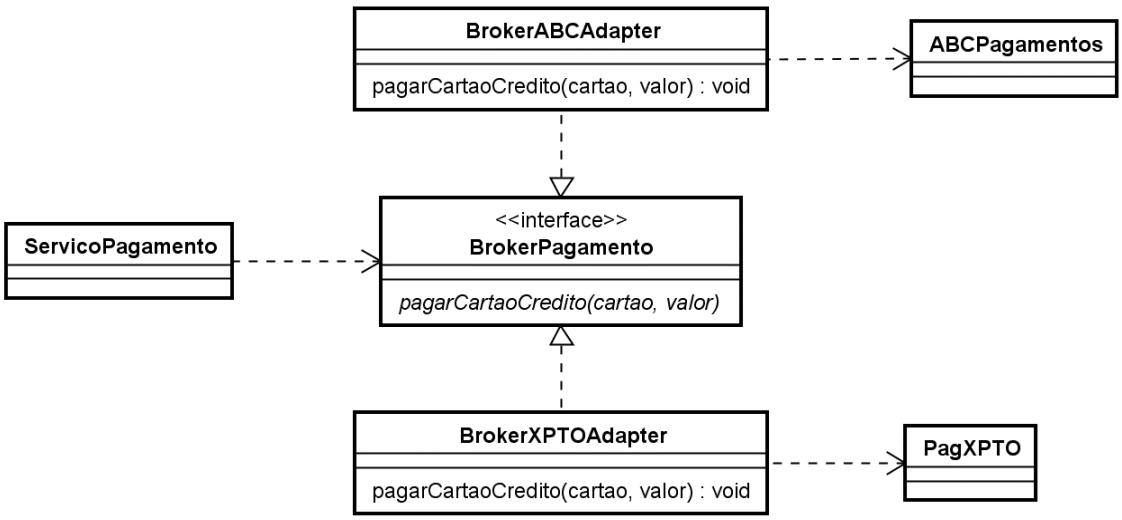


O participante Client representa um módulo cliente, enquanto Adaptee representa uma implementação específica do serviço. Em vez de o módulo Client utilizar diretamente essa implementação específica, o padrão sugere o uso de uma abstração, representada pelo participante Target, que define uma interface comum a todas as implementações.

💬 COMENTÁRIO

Para cada Adaptee específico, deve ser criado um adaptador, representado pelo participante Adapter, que traduz a chamada genérica definida na interface Target (operation) em uma ou mais chamadas da respectiva implementação específica (specificOperation).

A imagem a seguir ilustra a aplicação desse padrão no exemplo dos fornecedores de API de pagamentos.



Nesse exemplo, cada fornecedor (ABCPagamentos e PagXPTO) corresponde a uma ocorrência do participante Adaptee, definindo uma API específica que não pode ser alterada.

A interface BrokerPagamento representa a generalização dos serviços oferecidos pelos diferentes fornecedores e corresponde ao participante Target do padrão.

Os demais módulos do sistema utilizarão apenas essa interface, ficando isolados do fornecedor concreto da implementação desses serviços. BrokerABCAdapter e BrokerXPTOAdapter são classes que representam o papel Adapter do padrão e são responsáveis por traduzir a interface genérica definida em BrokerPagamento em chamadas específicas definidas pelos fornecedores ABCPagamentos e PagXPTO, respectivamente.

COMENTÁRIO

Cada fornecedor possui um adaptador específico, assim como existem adaptadores específicos para cada padrão de tomada elétrica.

A implementação a seguir mostra a estrutura da implementação com a utilização do padrão.

A classe ServicoPagamento, que representa um módulo cliente de uma solução de pagamento, recebe um objeto (broker) que implementa a interface genérica de pagamento (BrokerPagamento), ficando, portanto, isolada das implementações específicas. Novas soluções de pagamento podem ser acrescentadas ao produto sem haver necessidade do módulo cliente ser alterado.

Para cada fornecedor de API, foi criada uma classe Adapter que implementa o mapeamento da interface genérica nas chamadas específicas da respectiva API.

Observe como o código, que na solução anterior estava todo concentrado no módulo cliente, deu origem aos adaptadores para cada fornecedor de API de pagamento.

```
public class ServicoPagamento {
// o parâmetro BrokerPagamento é um objeto (adapter) que implementa a interface genérica
// BrokerPagamento
public void pagarCartaoCredito (BrokerPagamento broker, CartaoCredito cartao,
BigDecimal valor) {
broker.pagarCartaoCredito(cartao, valor);
}
}

public class BrokerXPTOAdapter implements BrokerPagamento (
public void pagarCartaoCredito(CartaoCredito cartao, BigDecimal valor) {
PagXPTO brokerPagamento = new PagXPTO();
DadosCartao dadosCartao = converterCartao(cartao);
if (isCartaoVisa(cartao)) {
brokerPagamento.pagarCartaoVisa(dadosCartao, valor);
} else {
brokerPagamento.pagarCartaoMastercard(dadosCartao, valor);
}
}
```

```
}

public class BrokerABCAdapter implements BrokerPagamento {
public void pagarCartaoCredito(CartaoCredito cartao, BigDecimal valor) {
ABCPagamentos brokerPagamento = new ABCPagamentos();
brokerPagamento.pagarEmCartaoCredito(cartao.numero(), cartao.nome(),
cartao.cvv(), cartao.validade(), valor);
}
}
```

CONSEQUÊNCIAS E PADRÕES RELACIONADOS AO PADRÃO ADAPTER

O padrão Adapter permite incorporar módulos previamente desenvolvidos, modificando sua interface original, sem que haja necessidade de alterá-los. Esse tipo de solução possibilita a utilização de diferentes implementações proprietárias compartilhando uma única interface, isolando, portanto, os módulos clientes das diferentes interfaces proprietárias.

O esforço envolvido na implementação de um Adapter depende do grau de semelhança que exista entre a interface genérica Target e a interface específica do componente para o qual o Adapter esteja sendo construído. Quanto maior a semelhança, menor o esforço e vice-versa.

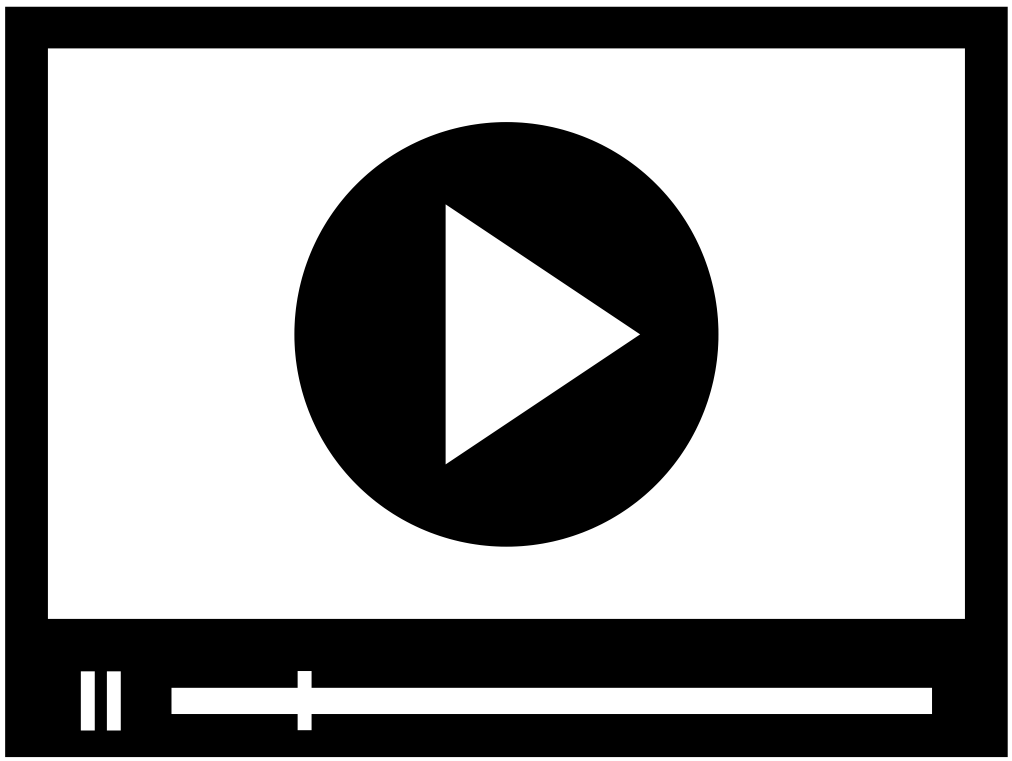
💬 COMENTÁRIO

O participante Target pode ser definido como uma interface puramente abstrata ou como uma classe abstrata, caso os diversos adaptadores possuam métodos em comum ou as operações compartilhem uma sequência comum de passos, em que cada passo tenha uma implementação específica em cada adaptador.

Neste último caso, portanto, as operações com uma sequência comum seriam implementadas na superclasse da qual os adaptadores específicos herdariam, aplicando-se conjuntamente o padrão Template Method.

COMENTÁRIO

O padrão Bridge tem uma estrutura similar ao padrão Adapter, porém, com um objetivo diferente, que é separar a interface da sua implementação, de modo que elas possam mudar de forma independente, sendo aplicado quando estamos implementando tanto os módulos clientes, como os fornecedores. O padrão Adapter, por sua vez, visa oferecer uma diferente interface para um ou mais componentes existentes, tipicamente fornecidos por terceiros.



PADRÃO DE PROJETO ADAPTER

No vídeo a seguir, apresentamos um exemplo de aplicação do padrão Adapter no desenvolvimento de software.

VERIFICANDO O APRENDIZADO

MÓDULO 2

⦿ **Reconhecer o propósito, a estrutura e as situações de aplicação dos padrões de projeto Bridge e Decorator**

INTENÇÃO DO PADRÃO BRIDGE

O objetivo principal do padrão Bridge é desacoplar uma abstração da sua implementação, permitindo que ambas possam variar de forma independente.

O padrão Bridge é, tipicamente, utilizado para evitar um problema de explosão combinatória de classes, resultante de soluções baseadas em herança, especialmente em hierarquias que precisam combinar aspectos do domínio (abstração) com outros específicos de implementação, tais como plataformas, tecnologias, estruturas de dados etc.

PROBLEMA RESOLVIDO PELO PADRÃO BRIDGE

Normalmente, quando uma abstração pode ter diferentes implementações com código em comum entre elas, a solução costuma ser a definição de uma hierarquia na qual o código comum é estabelecido na superclasse abstrata e cada implementação específica é definida em uma subclasse concreta.

Entretanto, esse tipo de solução não é flexível para comportar diferentes classificações da mesma abstração, pois cada classificação demanda uma hierarquia própria. O resultado é que

a combinação dessas diferentes classificações por meio de herança gera um número enorme de classes, tornando o projeto complexo e inflexível.

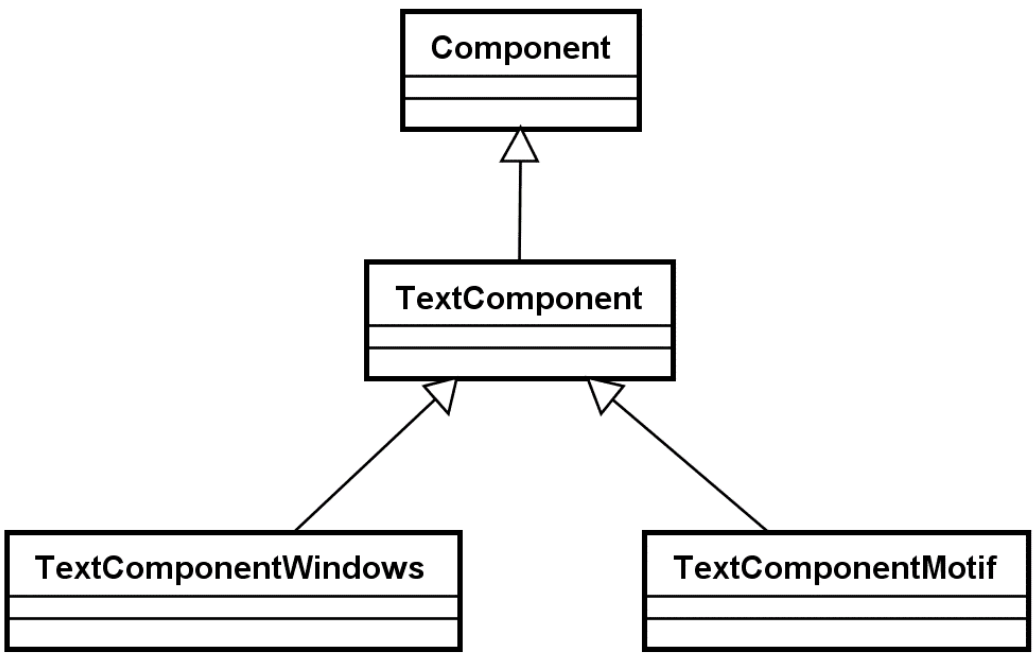
Esse problema costuma ocorrer quando combinamos diferentes perspectivas de classificação de uma abstração.

★ EXEMPLO

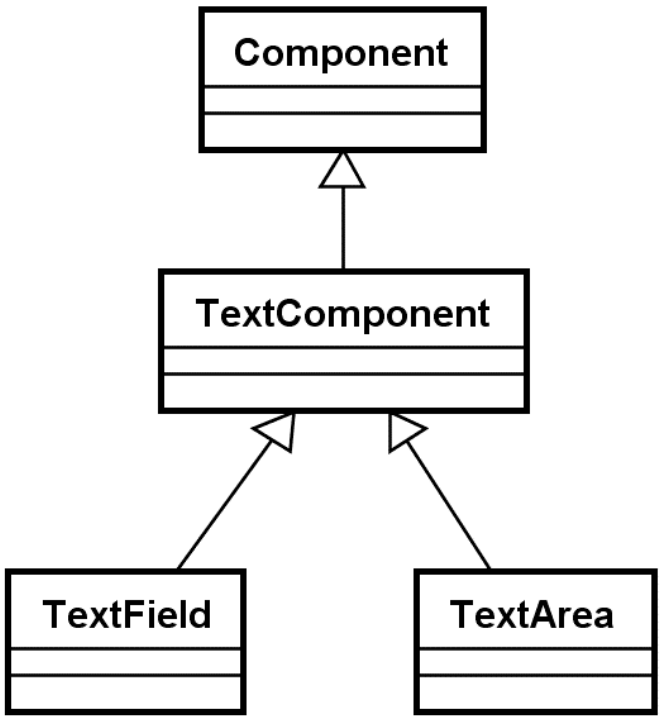
Podemos combinar a perspectiva do domínio com a de plataforma, como ocorre na implementação de frameworks de interface gráfica com o usuário, pois os elementos visuais, como painéis, botões e caixas de texto correspondem às abstrações que precisam ser implementadas em diferentes plataformas, tais como Windows, Motif etc.

A imagem a seguir apresenta um exemplo hipotético de um componente textual (TextComponent), que possui uma implementação específica para Windows e outra para Motif.

Você consegue perceber que o aspecto de variação entre as duas subclasses de TextComponent é a plataforma?



A solução seria aceitável se houvesse apenas essa perspectiva de variação. Porém, podem existir especializações de um componente textual inerentes ao domínio da abstração, tais como uma área de texto (TextArea) e um campo de texto (TextField), conforme ilustrado pelo diagrama a seguir.



Você consegue visualizar que existem especializações do componente texto próprias do domínio de interface gráfica, ao mesmo tempo em que existem especializações do componente texto decorrentes das diferentes plataformas de implementação da interface gráfica?

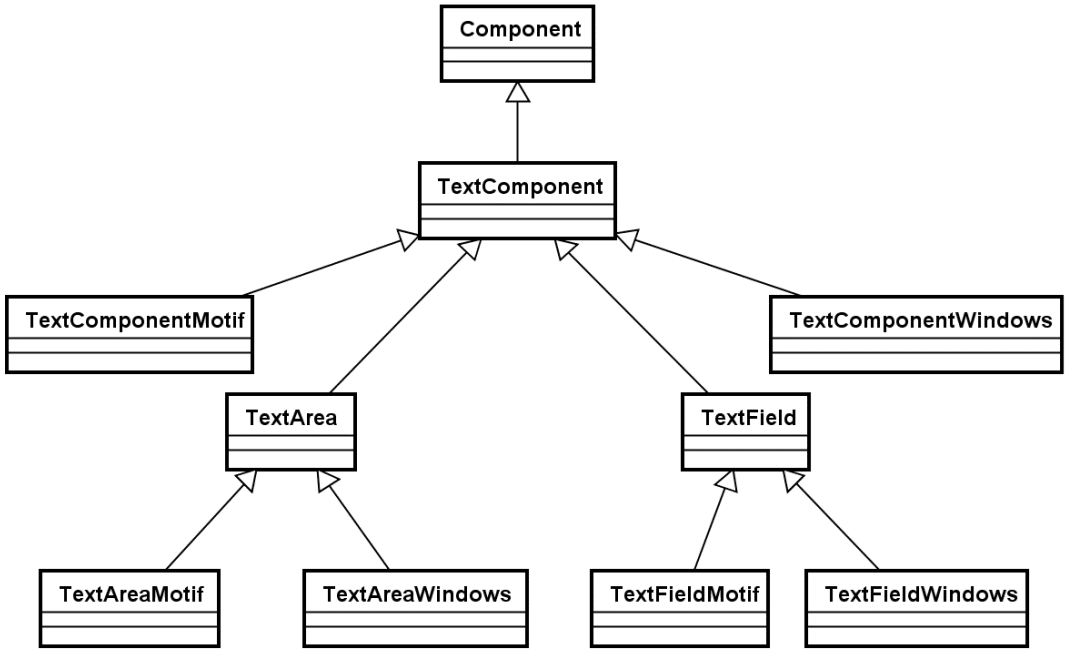
O problema agora é que tanto um TextField quanto um TextArea precisam ser implementados nas duas plataformas: Windows e Motif.

Qual seria a solução mais adequada?

Uma solução baseada na combinação dessas classificações é ilustrada no diagrama UML a seguir.

💬 COMENTÁRIO

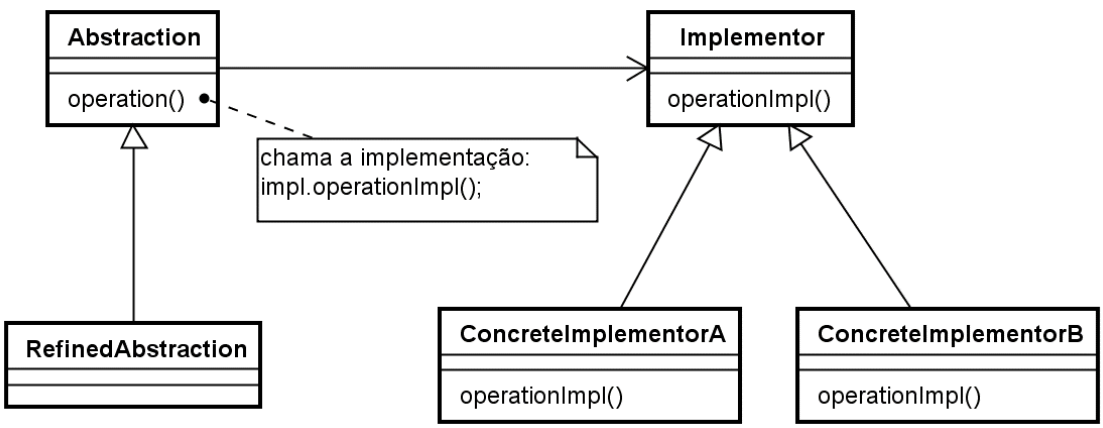
Perceba que esse caminho de solução gera uma enorme combinação de classes, misturando aspectos do domínio com aspectos de implementação e resultando em uma estrutura complexa e inflexível.



Portanto, o problema que o padrão Bridge soluciona é a separação da abstração (TextComponent e suas especializações do domínio, TextArea e TextField) dos aspectos de sua implementação (plataformas Windows e Motif), permitindo que ambas possam evoluir separadamente, isto é, podemos adicionar novos tipos de componentes e novas plataformas sem gerar um problema combinatório de difícil gestão.

SOLUÇÃO DO PADRÃO BRIDGE

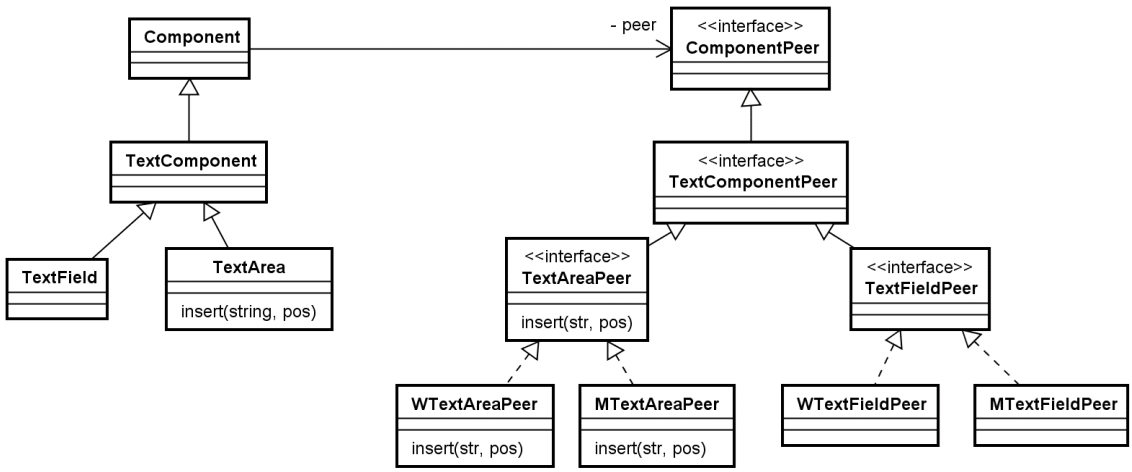
A estrutura da solução proposta pelo padrão Bridge está representada no diagrama de classes a seguir:



COMENTÁRIO

A ideia proposta pelo padrão é separar a hierarquia de abstrações da hierarquia de implementações, fazendo com que a implementação das operações na abstração (participante Abstraction) seja responsável por delegar a execução para a implementação específica que estiver conectada à abstração (participante ConcreteImplementor).

Veja na imagem a seguir como esse padrão é aplicado na implementação do Java AWT (Abstract Window Toolkit), framework básico de componentes de interface gráfica com o usuário da linguagem Java.



Perceba que a hierarquia à esquerda corresponde à abstração dos componentes do domínio de interface gráfica definida pelo participante Abstraction na estrutura do padrão Bridge, enquanto a hierarquia à direita corresponde aos aspectos de implementação dessa abstração nas diversas plataformas.

ATENÇÃO

Cada componente na abstração de elementos visuais do AWT possui uma ponte para o elemento peer correspondente, sendo que cada elemento folha da hierarquia peer possui implementação específica para uma plataforma.

Desse modo, um componente TextArea, por exemplo, possui uma operação insert, cuja implementação é responsável por chamar a operação insert do elemento peer associado. Neste caso, o componente TextAreaPeer possui duas implementações específicas: uma implementação Windows (WTextAreaPeer) e uma implementação Motif (MTextAreaPeer).

O código a seguir é um extrato da implementação desses componentes no AWT.

💬 COMENTÁRIO

Note que o método insertText, chamado pela operação insert da classe TextArea, acessa o elemento peer definido no ancestral Component e faz o downcasting desse elemento para TextAreaPeer, pois esse elemento é definido com o tipo genérico ComponentPeer. Na sequência, a operação insert do peer associado é chamada.

Desse modo, a implementação do componente TextArea não tem qualquer menção à plataforma específica. A classe WTextAreaPeer, por sua vez, implementa a operação insert chamando um código nativo para Windows, que atualizará o texto do componente na interface gráfica do Windows.

```
public class TextArea extends TextComponent {
public void insert (String str, int pos) {
insertText (str, pos);
}
public void insertText (String str, int pos) {
// insere o texto na posição indicada
String tmp1 = getText().substring(0, pos);
String tmp2 = getText().substring(pos, getText().length());
// chama o método setText da superclasse para atualizar o novo conteúdo
setText (tmp1 + str + tmp2);
// obtém o elemento peer associado específico de plataforma
TextAreaPeer peer = (TextAreaPeer) super.getPeer();
// chama a operação do componente peer para renderizar o texto na plataforma alvo
if (peer != null) {
peer.insert(str, pos);
}
}
}

// classe TextArea na plataforma alvo Windows
public class WTextAreaPeer implements TextAreaPeer {
```

```
public void insert(String text, int pos) {
replaceRange(text, pos, pos);
}

// código nativo específico da plataforma Windows
public native void replaceRange(String text, int start, int end);
}
```

CONSEQUÊNCIAS E PADRÕES RELACIONADOS AO PADRÃO BRIDGE

O padrão Bridge possibilita o desacoplamento de uma hierarquia de abstrações em relação a uma hierarquia de implementações associadas, o que promove a extensibilidade da solução, pois podemos acrescentar, de forma independente, tanto novos itens à abstração quanto novas implementações associadas.

Os objetos da hierarquia de implementação precisam ser criados e plugados nas respectivas abstrações.

Para promover o isolamento entre as abstrações e as suas implementações específicas, o padrão Abstract Factory pode ser utilizado para criar e configurar uma ponte entre as duas hierarquias.

💬 COMENTÁRIO

A estrutura de solução do padrão Bridge é similar à do padrão Adapter. A diferença fundamental é que o padrão Adapter é aplicado quando desejamos incorporar à solução elementos de terceiros, já prontos, que não podem ser modificados e que precisam ser acessados a partir de uma interface comum. Desse modo, no padrão Adapter, a interface comum é tipicamente definida a posteriori, ou seja, a partir de implementações já existentes que possuem interfaces distintas e incompatíveis entre si. O padrão Bridge, por outro lado, é uma solução definida a priori, ou seja, a interface comum de implementação é determinada de forma a permitir que diferentes implementações possam ser construídas a partir da sua definição.

INTENÇÃO DO PADRÃO DECORATOR

Decorator é um padrão **que permite adicionar responsabilidades a um objeto de forma dinâmica e mais flexível do que utilizando subclasses**.

PROBLEMA RESOLVIDO PELO PADRÃO DECORATOR

Uma forma comum de estender a funcionalidade de classes é por meio da criação de subclasses.

★ EXEMPLO

Imagine que você queira adicionar às classes de leitura e escrita de arquivos texto a capacidade de criar e ler arquivos texto criptografados e compactados.

Poderíamos resolver esse problema por meio de herança das classes do pacote Java I/O, criando uma estrutura como a ilustrada no código a seguir. O pacote Java I/O define a classe FileWriter, que oferece operações para a escrita de arquivos texto. Definimos, então, a classe EncryptedFileWriter como uma subclasse de FileWriter e sobrescrevemos o método write, adicionando a capacidade de criptografar o texto antes que seja escrito no arquivo.

```
public class EncryptedFileWriter extends FileWriter {
public EncryptedFileWriter(File file) throws IOException {
super(file);
}

public void write(String text) throws IOException {
String encryptedText = encrypt(text);
super.write(encryptedText); // comanda a gravação em disco via FileWriter
}
```

```
private String encrypt(String text) {
String result = text;
// aqui estaria o código para encriptar o texto
return result;
}
}
```

Para adicionar a capacidade de criar arquivos texto compactados, podemos criar uma outra extensão da classe FileWriter, denominada CompressedFileWriter, sobrescrevendo os métodos write e close. Nessa solução simplificada, supomos que a compactação ocorre somente no momento do fechamento do arquivo.

```
public class CompressedFileWriter extends FileWriter {
StringBuilder buffer = new StringBuilder();

public CompressedFileWriter(File file) throws IOException {
super(file);
}
public void write(String text) throws IOException {
buffer.append(text);
}
public void close() throws IOException {
// compacta o conteúdo no instante em que o arquivo vai ser fechado
char[] compressedBuffer = compress(buffer.toString());
super.write(compressedBuffer); // comanda a gravação em disco via FileWriter
super.close();
}
private char[] compress(String buffer) {
// algoritmo de compressão implementado aqui
// retorna resultado da compressão em um array
}
}
```

Soluções baseadas em herança para problemas dessa natureza são pouco flexíveis.

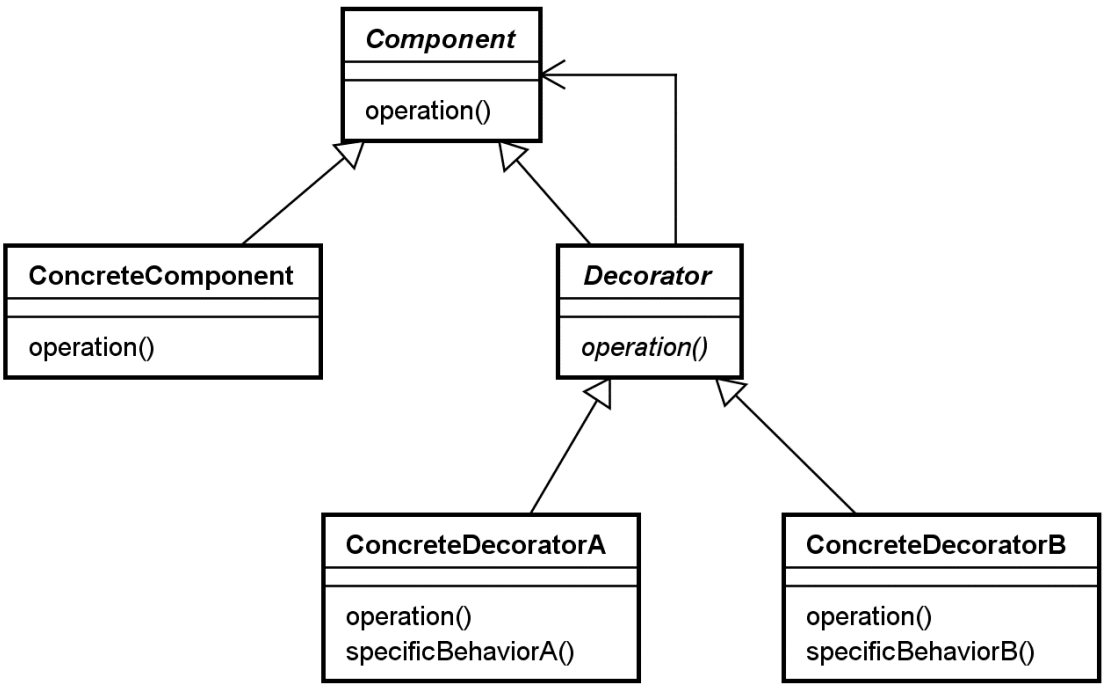
Você consegue identificar por que a solução dada para a criptografia e a compressão de arquivos texto nos exemplos anteriores é pouco flexível?

Agora, imagine que você queira gerar um arquivo texto criptografado e compactado. Poderíamos definir a classe CompressedFileWriter como uma subclasse da classe EncryptedFileWriter, em vez de FileWriter. Porém, cairíamos em outra situação problemática, pois não poderíamos gerar um arquivo apenas compactado e sem criptografia, a menos que adicionássemos complexidade ao projeto por meio de parâmetros de controle como setCryptoON, setCryptoOFF, por exemplo.

Portanto, o problema resolvido pelo padrão Decorator consiste em adicionar novos comportamentos às funcionalidades já existentes em uma determinada classe, sem que seja necessário alterar o código dessa classe ou criar subclasses, de modo que esses elementos possam ser combinados de maneiras diferentes, conferindo flexibilidade à solução.

SOLUÇÃO DO PADRÃO DECORATOR

A estrutura da solução proposta pelo padrão Decorator está representada no diagrama de classes a seguir:



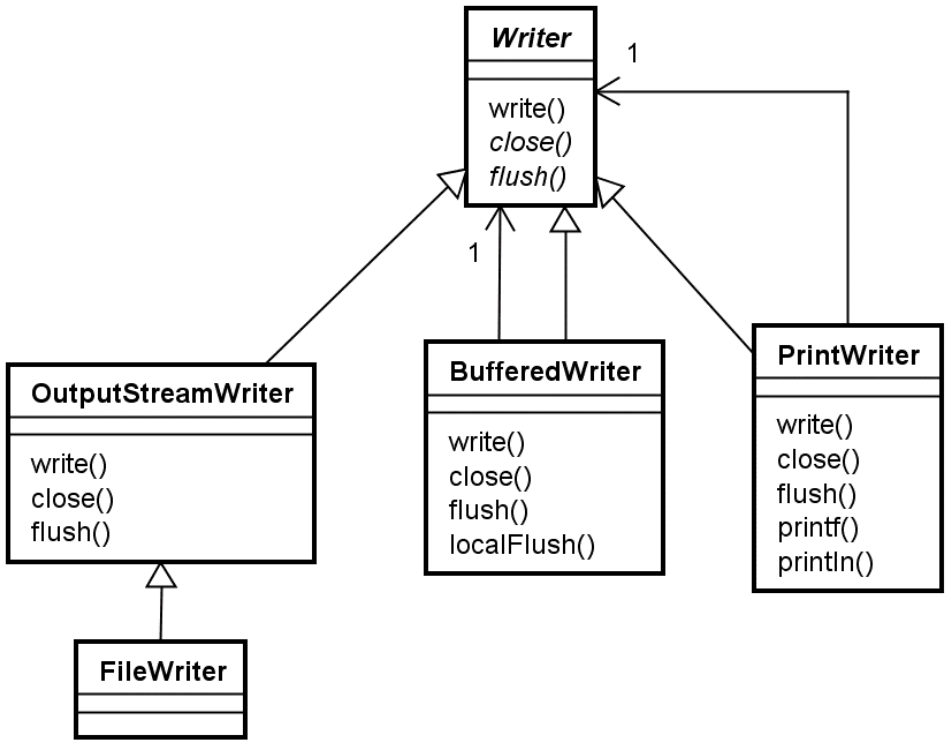
O participante Component define uma interface implementada pelas classes concretas, correspondentes ao participante ConcreteComponent, que podem ter comportamentos adicionados a elas dinamicamente. O participante Decorator também é uma especialização de Component e, portanto, todas as operações definidas em Component podem ser sobrescritas pelos elementos representados pelo participante Decorator. Além disso, comportamentos

adicionais são definidos nas implementações concretas do Decorator, representadas pelos participantes ConcreteDecoratorA e ConcreteDecoratorB.

📢 ATENÇÃO

Cada Decorator tem uma referência para o elemento Component, ao qual ele está adicionando funcionalidades, funcionando como uma espécie de envoltório ao componente original.

As classes do pacote Java I/O utilizam amplamente essa solução, conforme ilustrada por alguns elementos presentes no diagrama UML, a seguir:



Writer é uma classe abstrata que corresponde ao participante Component do padrão.

As classes OutputStreamWriter e FileWriter desempenham o papel de ConcreteComponent e implementam as operações de escrita em um arquivo texto.

As classes BufferedWriter e PrintWriter correspondem ao participante ConcreteDecorator, adicionando funcionalidades a algum objeto que implemente a definição da classe Writer.

Note que os desenvolvedores optaram por não implementar uma superclasse comum no papel de Decorator, fazendo com que ambas as classes tenham uma referência para o Writer que está sendo decorado.

A classe BufferedWriter adiciona a um Writer a capacidade de manter uma área de armazenamento interno em memória e somente comandar a escrita em disco quando essa área estiver cheia, reduzindo o número de acessos a disco. A classe PrintWriter adiciona funcionalidades de saída formatada por meio de operações como printf e println.

O código a seguir apresenta um trecho da implementação da classe BufferedWriter, utilizando o padrão Decorator.

Todo Decorator recebe, no seu construtor, um parâmetro correspondente ao participante Component do padrão, que, neste exemplo, corresponde à classe abstrata Writer.

A classe BufferedWriter implementa uma versão específica decorada da operação write que armazena os dados recebidos em um array e, somente quando a capacidade ultrapassa um limiar, ela chama a operação localFlush, que se responsabiliza por chamar a operação write do componente que está sendo decorado, fazendo efetivamente a escrita da área temporária para o arquivo.

```
public class BufferedWriter extends Writer
private Writer out; // elemento que está sendo decorado
char[] buffer; // área de escrita temporária em memória
int count; // número de caracteres ocupados na área de escrita temporária

public BufferedWriter(Writer out) throws IOException {
this.out = out;
buffer = new char[4096]; // cria uma área temporária de 4K
}
public void write(char[] buf, int offset, int len) throws IOException {
if (count + len > buffer.length) { // se a área a ser escrita não couber mais no buffer
localFlush(); // descarrega área para disco
} else {
System.arraycopy(buf, offset, buffer, count, len); // salva na área temporária
count += len; // incrementa espaço ocupado
if (count == buffer.length) // chegou no limite da área temporária?
localFlush(); // descarrega área para disco
}
}

public void flush() throws IOException {
```

```
localFlush();
out.flush();
}

protected void localFlush() throws IOException {
if (buffer.length > 0)
out.write(buffer, 0, count); // chama operação write do componente decorado
}
}
```

O código a seguir mostra como essas classes podem ser utilizadas em conjunto. Note como o objeto writer é decorado por um BufferedWriter e por um PrintWriter, que adicionam as funcionalidades de armazenamento em área temporária e saída formatada, respectivamente, ao objeto FileWriter base. Em vez de herança, um esquema de composição sucessiva é utilizado, com o objeto base sendo passado como parâmetro para o construtor do elemento que adiciona novas funcionalidades. Desse modo, um objeto FileWriter é passado ao construtor da classe BufferedWriter que, por sua vez, é passado ao construtor da classe PrintWriter.

```
public class Exemplo {
public static void main(String[] args) {
try (FileWriter writer = new FileWriter("app. log");
BufferedWriter bw = new BufferedWriter(writer); // writer é o objeto base
PrintWriter pw = new PrintWriter(bw)) { // bw é o objeto base

pw.println("Exemplo de uso do decorator");

} catch (IOException e) {
System.err.format("IOException: %s%n", e);
}
}
}
```

CONSEQUÊNCIAS E PADRÕES

RELACIONADOS AO PADRÃO DECORATOR

O padrão Decorator oferece uma forma mais flexível de adicionar funcionalidades a objetos do que as soluções baseadas em herança, por ser baseado em composição de objetos. Com a utilização desse padrão, responsabilidades podem ser adicionadas, em tempo de execução, pela adição de camadas de decoradores, formando uma estrutura similar a uma cebola.

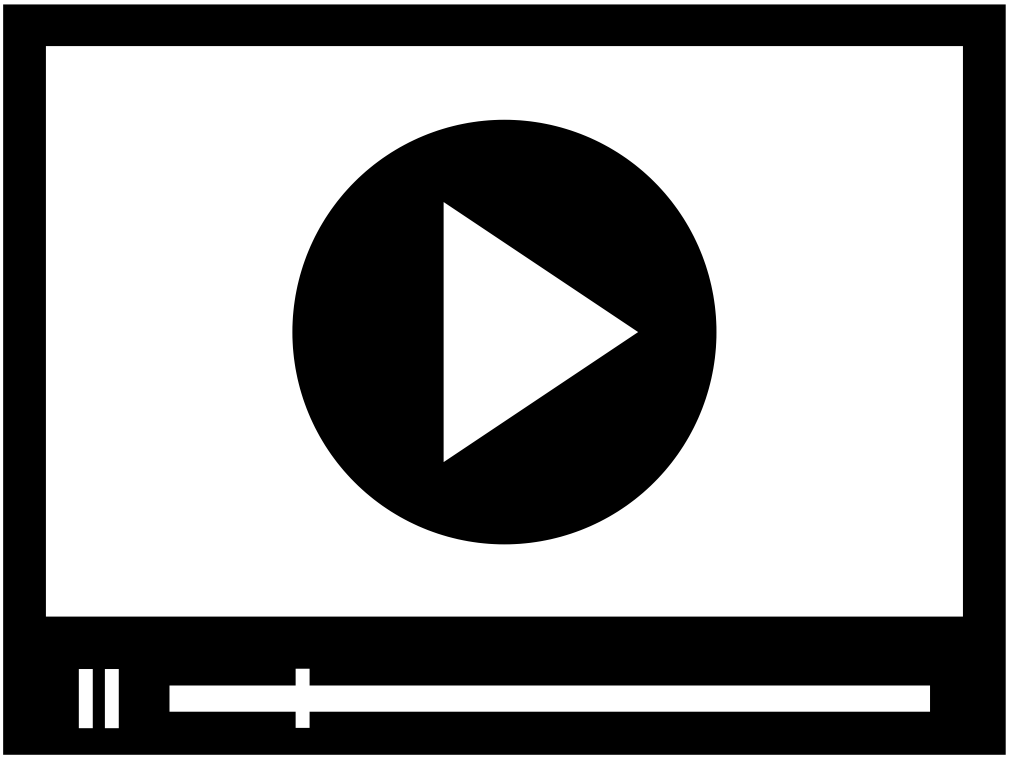
Além disso, a solução baseada em composição possibilita a definição de uma interface mais simples para o objeto base (o núcleo da cebola), ao contrário das soluções baseadas em herança, em que frequentemente a superclasse tem que ser estendida para comportar novas operações definidas nas subclasses.

Entretanto, as soluções que utilizam o padrão Decorator podem acabar utilizando muitos objetos pequenos, todos muito parecidos e conectados, o que pode complicar o seu entendimento e a correção de erros.

O padrão Decorator tem estrutura similar à utilizada no padrão Composite, mas um decorator tem o propósito de adicionar responsabilidades a outros objetos, enquanto o padrão Composite está mais ligado à agregação de objetos em uma estrutura hierárquica.

DESAFIO

Você consegue implementar as classes EncryptedFileWriter e CompressedFileWriter apresentadas anteriormente utilizando o padrão Decorator?



PADRÕES DE PROJETO BRIDGE E DECORATOR

Apresentamos exemplos de aplicação dos padrões Bridge e Decorator no desenvolvimento de software, no vídeo a seguir.

VERIFICANDO O APRENDIZADO

MÓDULO 3

⊙ **Reconhecer o propósito, a estrutura e as situações de aplicação dos padrões de projeto Composite e Facade**

INTENÇÃO DO PADRÃO COMPOSITE

O padrão Composite **permite representar hierarquias de composição de objetos em uma estrutura de árvore**, viabilizando que a manipulação dos objetos individuais e dos agregados possa ser feita com o mesmo conjunto de operações.

PROBLEMA RESOLVIDO PELO PADRÃO COMPOSITE

★ EXEMPLO

Suponha que você esteja desenvolvendo um programa de envio e recepção de mensagens texto que devem ser organizadas em pastas. Além de mensagens, pastas podem conter outras pastas.

Podemos realizar operações com as mensagens, tais como apagar uma mensagem, criptografar uma mensagem, entre outras. Essas mesmas operações se aplicam às pastas, sendo que apagar uma pasta implica em apagar todas as mensagens e as pastas descendentes dela.

O padrão Composite é especialmente interessante para problemas de representação de hierarquias de elementos nas quais observamos recursividade na execução das operações aplicadas aos agregados.

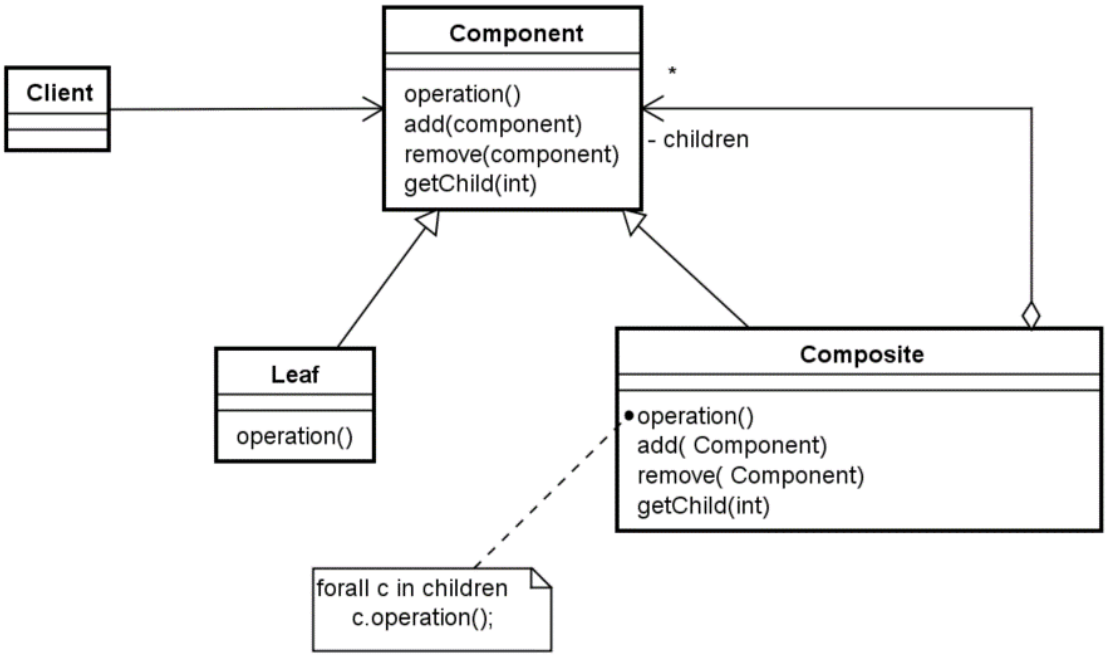
O código apresentado a seguir ilustra a estrutura de implementação para o problema sem utilizar o padrão Composite.

```
public class ServicoMensagem {
public void apagar(Object elemento) {
if (elemento instanceof Mensagem)
apagarMensagem((Mensagem) elemento);
else if (elemento instanceof Pasta)
apagarPasta((Pasta) elemento);
}
private void apagarMensagem(Mensagem mensagem) {
// lógica para apagar a mensagem
}
private void apagarPasta(Pasta pasta) {
// lógica para apagar a pasta
// que replicaria a lógica condicional da operação apagar
// pois uma pasta pode conter mensagens e pastas
}
}
```

Esse código apresenta problemas, pois, além das operações com os diferentes elementos estarem concentradas em uma única classe (ServicoMensagem), existe um código condicional baseado no tipo do objeto (mensagem ou pasta), que identifica a operação a ser executada (apagarMensagem ou apagarPasta). A adição de novas operações e novos tipos de objeto, certamente, aumentará a complexidade dessa implementação.

SOLUÇÃO DO PADRÃO COMPOSITE

A estrutura da solução proposta pelo padrão Composite está representada no diagrama de classes a seguir:



A ideia central do padrão é representar a hierarquia de modo que todos os elementos herdem de uma superclasse genérica representada pelo participante Component.

O participante Leaf representa um elemento primitivo da hierarquia (folha), isto é, um elemento que não possui descendentes, enquanto o participante Composite representa os agrupamentos de elementos (folhas ou outros agregados).

O agrupamento é definido pelo relacionamento entre o agregado Composite e seus filhos do tipo Component.

O participante Client representa um módulo cliente qualquer que utiliza os elementos da hierarquia.

Nesse modelo, operation representa cada operação aplicável tanto à folha como ao agregado.

Além disso, um Composite define a implementação para operações de gerenciamento do agregado, permitindo adicionar e remover elementos, além de obter um elemento específico.

A estrutura de código a seguir ilustra a aplicação do padrão no exemplo do programa de mensagens.

```
public abstract class Elemento {
public void adicionar(Elemento elem) { }
public void remover(Elemento elem) { }
public abstract void apagar();
public abstract void criptografar();
}
```

```
public class Mensagem extends Elemento {
public void apagar() {
// lógica para apagar a mensagem
}
public void criptografar() {
// lógica para criptografar a mensagem
}
}

public class Pasta extends Elemento {
private List filhos = new ArrayList<>();
public void adicionar(Elemento elem) {
filhos.add(elem);
}
public void remover(Elemento elem) {
filhos.remove(elem);
}
public void apagar() {
for (Elemento elemento : filhos) // apaga todos os filhos
elemento.apagar();
// lógica adicional para apagar a pasta
}
public void criptografar() {
for (Elemento elemento : filhos) // criptografa todos os filhos
elemento.criptografar();
// lógica adicional para criptografar a pasta
}
public class ServicoMensagem {
public void apagar(Elemento elemento) {
// código adicional de tratamento da requisição ficaria aqui
elemento.apagar(); // apaga o elemento: pode ser uma mensagem ou uma pasta
}
public void criptografar(Elemento elemento) {
// código adicional de tratamento da requisição ficaria aqui
elemento.criptografar(); // criptografa o elemento: pode ser uma mensagem ou uma pasta
```

}
}

A classe **Elemento** corresponde ao participante Component do padrão, definindo as operações para adicionar e remover elementos de um agregado, além das operações abstratas apagar e criptografar, que se aplicam tanto a uma mensagem como a uma pasta de mensagens.

A classe **Mensagem** corresponde ao participante Leaf do padrão, definindo a implementação específica das operações para apagar e criptografar uma mensagem.

A classe **Pasta** corresponde ao participante Composite do padrão, definindo a implementação das operações de adição e remoção de elementos à pasta, além do comportamento específico para as operações apagar e criptografar.

Como é comum na implementação do padrão Composite, as operações apagar e criptografar do agregado chamam as operações de mesmo nome dos filhos, podendo adicionar algum tratamento específico, além daquele implementado nas folhas.

ServicoMensagem representa um módulo cliente, que, simplesmente, chama a operação sem precisar saber se o elemento é uma Mensagem ou uma Pasta.

CONSEQUÊNCIAS E PADRÕES RELACIONADOS AO COMPOSITE

O padrão Composite representa uma estrutura hierárquica de objetos, permitindo que os módulos clientes possam manipular seus elementos de maneira uniforme por meio da mesma interface.

Em geral, os módulos clientes não precisam saber se estão lidando com um elemento primitivo (folha) ou com um agregado, o que simplifica o código, pois não é necessário utilizar estruturas condicionais para identificar a operação a ser chamada.

O padrão permite adicionar novos tipos de componentes sem que seja necessário alterar o código dos módulos clientes.

Por outro lado, a estrutura é muito genérica, o que dificulta a implementação de hierarquias que tenham regras de composição mais restritivas em relação aos tipos de elemento que um determinado agregado possa ter.

💬 COMENTÁRIO

O padrão Iterator é, muitas vezes, utilizado para fazer o percurso pelos elementos de uma hierarquia implementada com o padrão Composite.

O padrão Decorator também é utilizado com frequência junto com o padrão Composite. Nesse caso, eles, tipicamente, herdam da mesma superclasse, fazendo com que as classes que representam o participante Decorator tenham que implementar a interface definida pelo participante Component.

INTENÇÃO DO PADRÃO FACADE

O padrão Facade **tem o propósito de oferecer uma interface de alto nível para um componente ou subsistema**, de modo que os módulos clientes possam utilizá-lo mais facilmente, não precisando manipular uma estrutura complexa de navegação de objetos, nem ficando vulnerável a alterações nesta estrutura.

PROBLEMA RESOLVIDO PELO PADRÃO FACADE

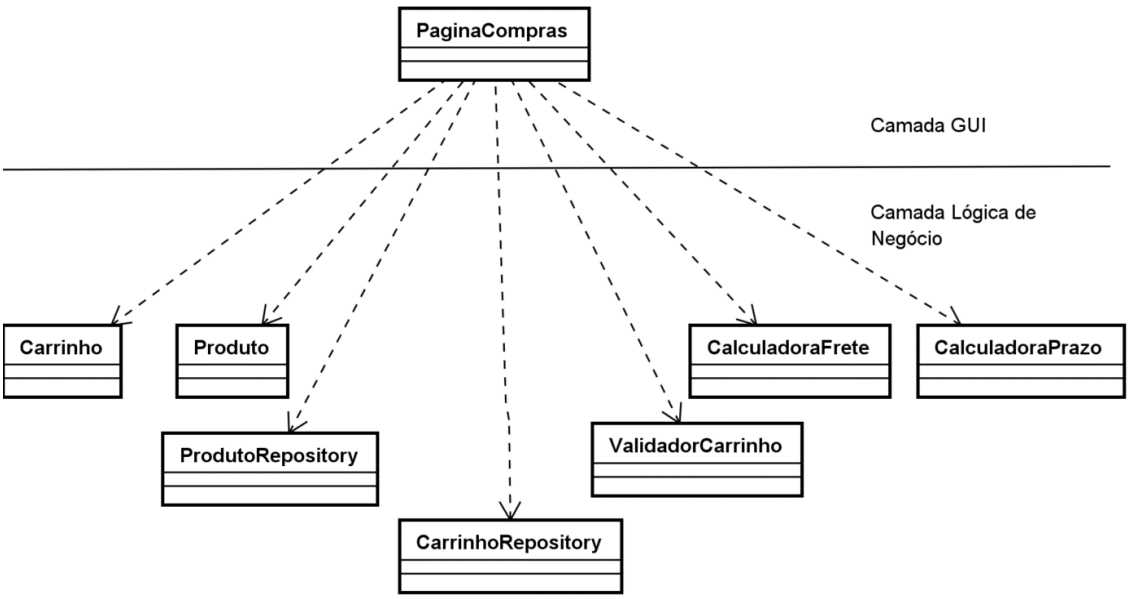
⭐ EXEMPLO

Imagine que você esteja implementando um sistema de comércio eletrônico que pode ser utilizado tanto em uma aplicação web, como em um aplicativo para dispositivo móvel.

Uma estrutura comum para esse tipo de sistema é estratificar os componentes em camadas, separando em camadas distintas os elementos que compõem a interface com o usuário, a lógica do negócio e os demais elementos dependentes de tecnologia, tais como armazenamento e recuperação de dados, integração com sistemas e dispositivos externos, entre outros.

Suponha que para implementar uma parte da página de compras, o código precise manipular um objeto CarrinhoRepository para recuperar o carrinho do cliente, um objeto Carrinho para adicionar produtos ao carrinho do cliente, um objeto ProdutoRepository para recuperar produtos a serem apresentados ao cliente, um objeto ValidadorCarrinho para verificar se o carrinho atende às regras do negócio, um objeto para calcular o valor do frete até o destino definido pelo cliente e um objeto para calcular o prazo de entrega.

A estrutura de dependências resultante dessa solução é apresentada no diagrama de classes a seguir.



O problema dessa estrutura é que o elemento PaginaCompras, que pertence à camada de interface com usuário, precisa conhecer e manipular diversos elementos da camada de Lógica de Negócio, o que torna seu código mais complexo e suscetível a modificações que ocorram na estrutura de objetos de negócio.

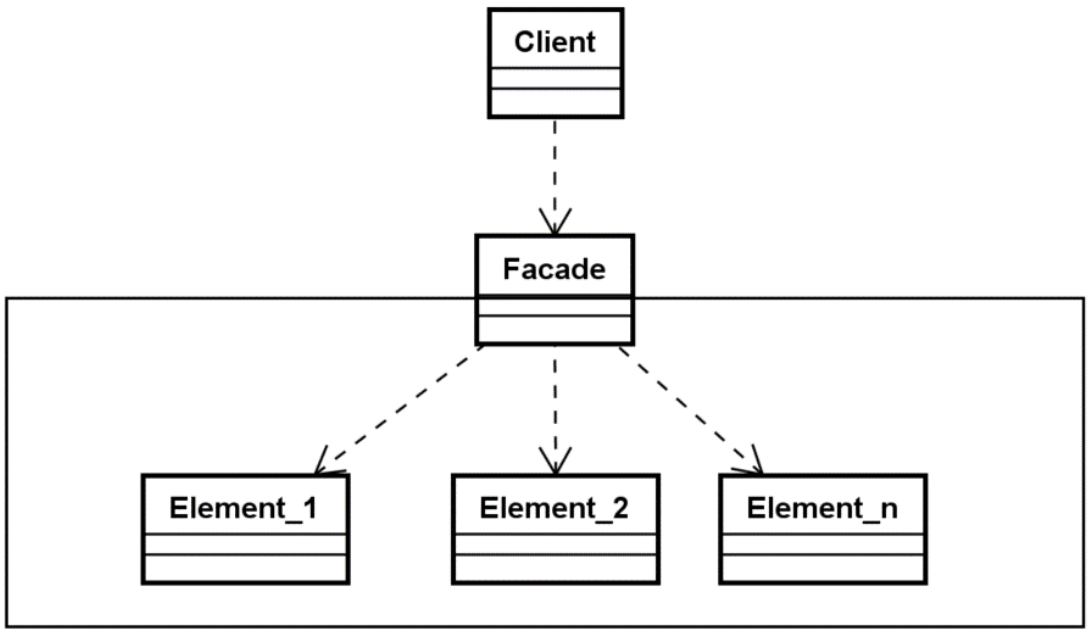
💬 COMENTÁRIO

De forma geral, o problema resolvido pelo padrão Facade é permitir que um módulo cliente utilize um subsistema ou uma camada complexa de um sistema sem precisar conhecer ou manipular diversos elementos e, com isso, reduzir o acoplamento entre o subsistema e os módulos clientes.

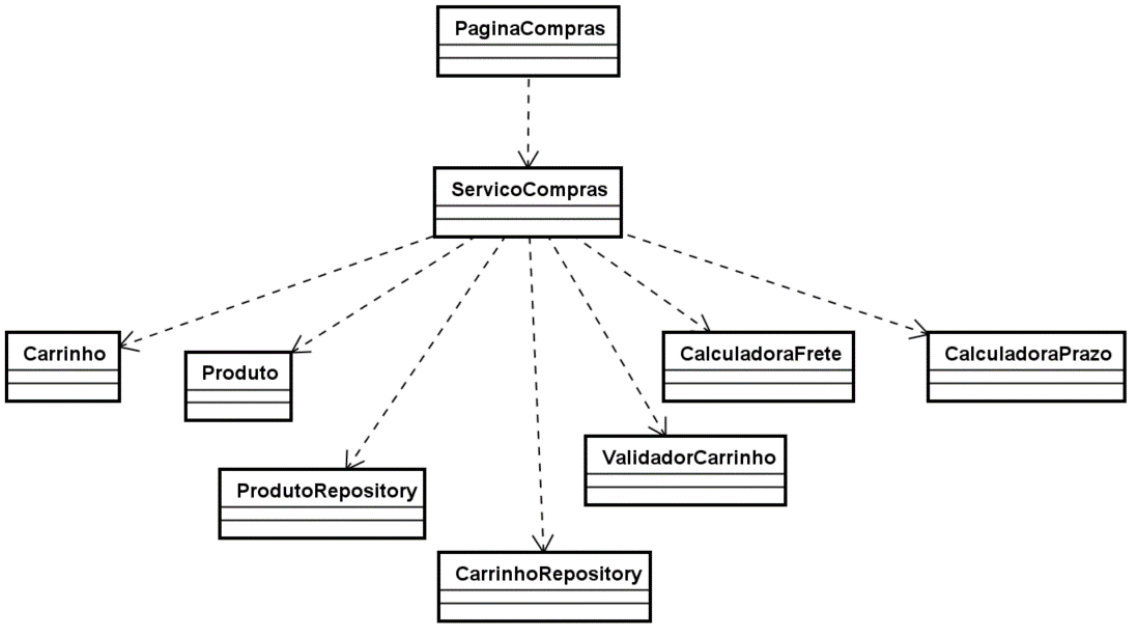
SOLUÇÃO DO PADRÃO FACADE

A estrutura da solução proposta pelo padrão Facade é apresentada no diagrama UML a seguir.

Consiste, basicamente, em definir uma classe correspondente ao participante Facade, que oferece uma interface de alto nível ao participante Client, concentrando toda a manipulação dos elementos pertencentes à camada ou subsistema. Portanto, o elemento Facade passa a ser o ponto focal de contato entre o cliente e o subsistema, ou camada com o qual ele precisa se comunicar.



A aplicação do padrão no problema mencionado anteriormente nos leva à estrutura definida no diagrama a seguir. O elemento PaginaCompras agora interage com uma interface de mais alto nível oferecida pelo elemento ServicoCompras. Desse modo, toda a manipulação de objetos que estava presente em PaginaCompras passa a ser implementada em ServicoCompras.



CONSEQUÊNCIAS E PADRÕES RELACIONADOS AO FACADE

O padrão Facade reduz a complexidade de implementação dos módulos clientes de um subsistema, promovendo menor acoplamento entre o subsistema e os seus clientes.

Isso permite que os componentes internos do subsistema possam ser modificados sem afetar os módulos clientes.

Entretanto, o padrão não impede que os módulos clientes possam acessar elementos do subsistema, se necessário. Normalmente, o acesso a esses elementos é promovido e controlado pelo módulo Facade.

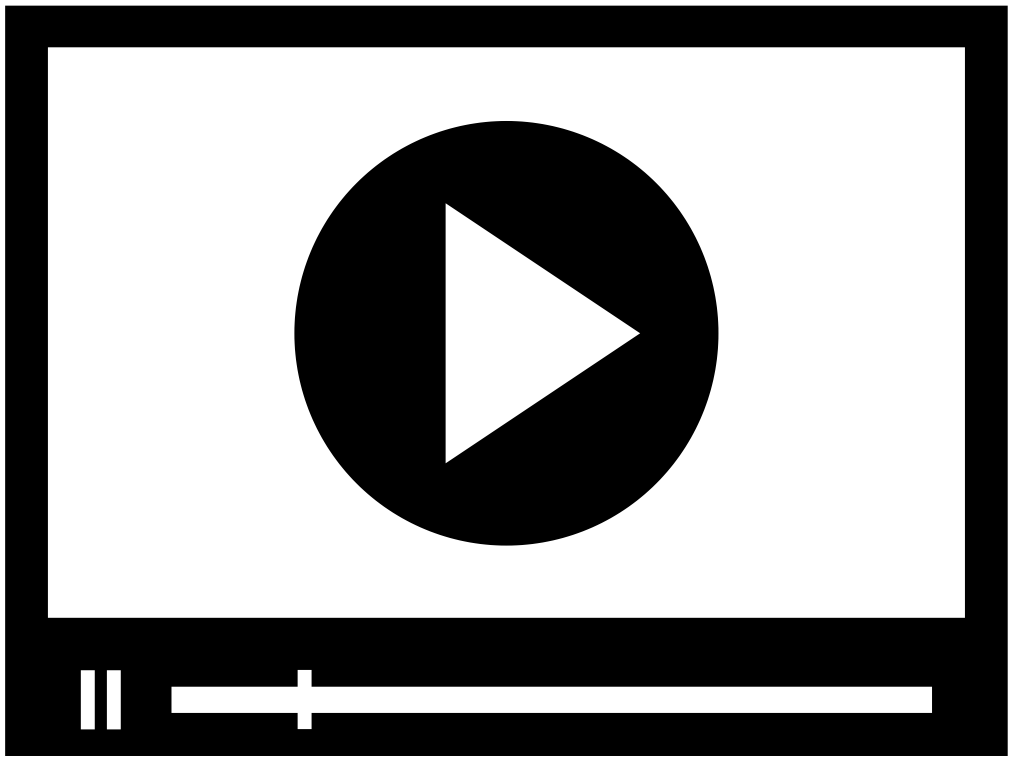
É possível reduzir ainda mais o acoplamento entre um subsistema e os seus clientes ao implementar o Facade por meio de uma interface abstrata que pode ser implementada por diferentes classes concretas.

💬 COMENTÁRIO

O padrão Abstract Factory ou um framework de injeção de dependências podem ser utilizados para criar a implementação concreta de um subsistema, caso o Facade seja uma interface

abstrata.

O padrão Mediator possui uma estrutura similar ao Facade, contudo, tem o propósito de abstrair a comunicação entre objetos da mesma camada, centralizando funcionalidades que não sejam específicas de nenhum desses objetos. O padrão Facade, por outro lado, tem o propósito de fornecer uma interface de alto nível para clientes de outros subsistemas.



PADRÕES DE PROJETO COMPOSITE E FACADE

Encerramos este módulo com o vídeo a seguir, no qual apresentamos exemplos de aplicação dos padrões Composite e Facade no desenvolvimento de software.

VERIFICANDO O APRENDIZADO

MÓDULO 4

⊙ **Reconhecer o propósito, a estrutura e as situações de aplicação dos padrões de projeto Flyweight e Proxy**

INTENÇÃO DO PADRÃO FLYWEIGHT

O propósito do padrão Flyweight **é permitir a utilização de diversos pequenos objetos de forma eficiente**, por meio de uma solução baseada em compartilhamento de objetos.

PROBLEMA RESOLVIDO PELO PADRÃO FLYWEIGHT

Suponha que você esteja implementando uma aplicação envolvendo substâncias químicas.

Uma substância química simples é formada por átomos de um único elemento químico, como, por exemplo, o hidrogênio (H_2) e o enxofre (S_8). Uma substância composta é formada por átomos de elementos químicos diferentes como, por exemplo, a água (H_2O) e o bicarbonato de sódio ($NaHCO_3$).

Uma maneira de representar esses elementos em Java é expressa no código a seguir:

```
public class ElementoQuimico {
private String simbolo;
private String nome;
```

```
public ElementoQuimico(String simbolo, String nome) {
this.simbolo = simbolo;
this.nome = nome;
}
// getters e setters omitidos
}

public abstract class Substancia {
private String nome;
public Substancia(String nome) {
this.nome = nome;
}
// getters e setters omitidos
}

public class SubstanciaSimples extends Substancia {
private int atomos;
private ElementoQuimico elemento;

public SubstanciaSimples(String nome, ElementoQuimico elemento, int atomos) {
super(nome);
this.atomos = atomos;
this.elemento = elemento;
}
// getters e setters omitidos
}

public class SubstanciaComposta extends Substancia {
// Conjunto de elementos químicos e respectivas quantidades de átomos
private Map<ElementoQuimico, Integer> composicao;

public SubstanciaComposta(String nome, Map<ElementoQuimico, Integer> composicao) {
super(nome);
this.composicao = composicao;
}
```

```
// getters e setters omitidos
}
```

Um elemento químico contém os atributos símbolo e nome.

A classe SubstanciaSimples tem o nome herdado da superclasse e define o número de átomos do respectivo elemento químico.

A classe SubstanciaComposta define a composição de elementos químicos e suas respectivas quantidades de átomos em uma estrutura do tipo chave-valor (Map).

O código a seguir mostra como objetos dessas classes poderiam ser instanciados. Note como o elemento químico oxigênio é instanciado repetidamente nas diferentes substâncias criadas.

```
public class Exemplo {
public static void main(String[] args) {
SubstanciaSimples s1 = new SubstanciaSimples("Oxigênio",
new ElementoQuimico("O", "Oxigênio"), 2);
SubstanciaSimples s2 = new SubstanciaSimples("Ozônio",
new ElementoQuimico("O", "Oxigênio"), 3);

Map<ElementoQuimico, Integer> composicaoAgua = new HashMap<>();
composicaoAgua.put(new ElementoQuimico("H", "Hidrogênio"), 2);
composicaoAgua.put(new ElementoQuimico("O", "Oxigênio"), 1);

SubstanciaComposta s3 = new SubstanciaComposta("Agua", composicaoAgua);
}
}
```

A definição de elemento químico é um exemplo característico do que denominamos objeto imutável, isto é, os valores de seus atributos não mudam.

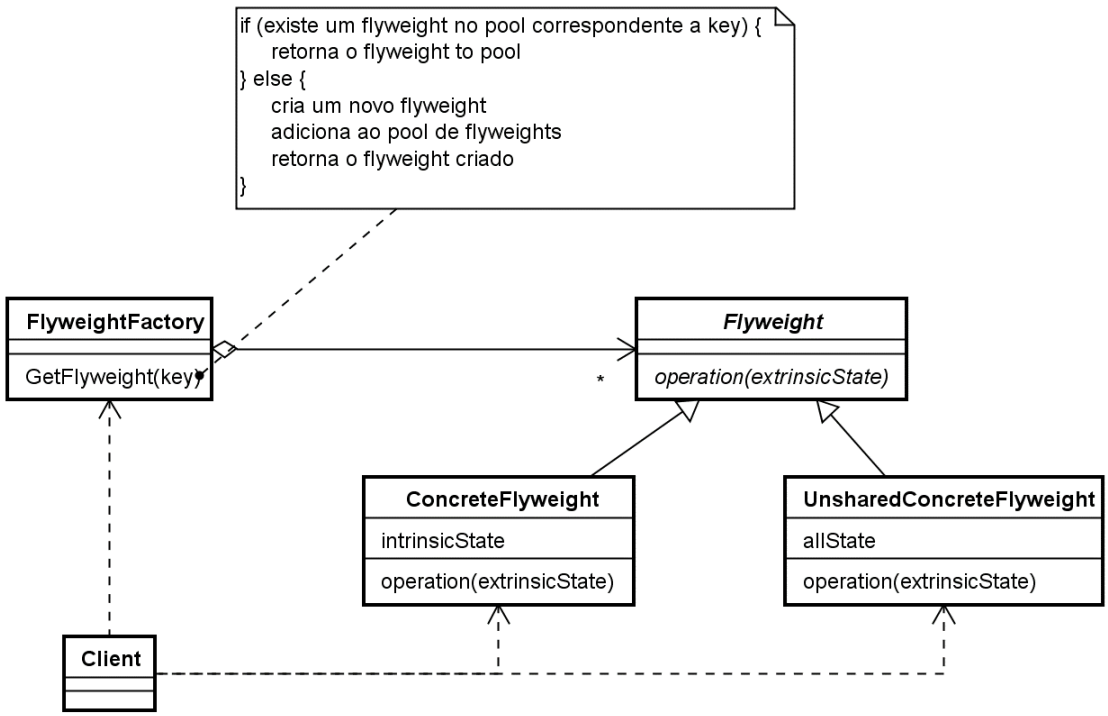
💬 COMENTÁRIO

Considerando que um elemento químico é imutável, e supondo que precisamos criar milhares de substâncias na nossa aplicação de química, a solução apresentada é altamente ineficiente no uso de memória, pois serão instanciados milhares de pequenos objetos correspondentes aos elementos químicos.

O problema que esse padrão tenta resolver, portanto, consiste em como podemos compartilhar objetos imutáveis, ou as partes imutáveis de objetos, de modo a utilizar os recursos de memória de forma mais eficiente.

SOLUÇÃO DO PADRÃO FLYWEIGHT

A estrutura da solução proposta pelo padrão Flyweight está representada no diagrama de classes a seguir.



A ideia central consiste na construção de um pool de objetos compartilhados, criado e atualizado por uma fábrica. Desse modo, os objetos clientes, em vez de instanciarem diretamente esses objetos, passam a fazer uso da fábrica.

O participante FlyweightFactory representa a fábrica que gerencia o pool de objetos compartilháveis. Sempre que um cliente pedir um objeto flyweight, a fábrica retorna uma já existente ou cria uma nova, se ainda não existir uma.

Os objetos flyweight estão organizados em uma hierarquia com duas subclasses:

ConcreteFlyweight, representando os objetos que são compartilhados.

UnsharedConcreteFlyweight, que representa os objetos que não são compartilhados.

Isso acontece porque o padrão não obriga que todos os objetos que implementam a interface abstrata Flyweight sejam compartilhados, mas permite que os clientes fiquem isolados da decisão de compartilhar ou não os objetos de uma determinada classe.

Portanto, podemos iniciar um projeto definindo que objetos de uma classe não serão compartilhados (assumindo o papel de UnsharedConcreteFlyweight) e depois passarmos para uma solução em que eles sejam compartilhados, assumindo o papel de ConcreteFlyweight, sem qualquer impacto para os módulos clientes.

COMENTÁRIO

Note que um ConcreteFlyweight somente compartilha o seu estado intrínseco, isto é, um conjunto de propriedades imutáveis. Se os objetos compartilhados de uma classe possuírem propriedades extrínsecas, ou seja, que possam mudar de valor, elas deverão ser mantidas pelos módulos clientes, que poderão passar esses dados como parâmetros de operações do flyweight.

O exemplo a seguir mostra como o código das substâncias químicas pode ser escrito com a utilização do padrão Flyweight. Os elementos químicos, em vez de serem instanciados diretamente pelos módulos clientes, passam a ser instanciados pela classe ElementoQuimicoFactory, que desempenha o papel do participante FlyweightFactory do padrão. Ela armazena todos os elementos já criados e se necessário, cria novos elementos. O programa exemplo cria as substâncias solicitando os elementos para a fábrica. Desse modo, apenas dois elementos são instanciados (H e O), em vez de quatro, como no programa original, dado que o objeto representando o elemento oxigênio é compartilhado por todos os objetos que representam as substâncias instanciadas.

Note que o construtor da classe ElementoQuimico não é mais público, para evitar sua instanciação direta. As classes ElementoQuimico e ElementoQuimicoFactory devem ficar no mesmo pacote para permitir que a fábrica tenha acesso ao construtor de ElementoQuimico.

```
public class ElementoQuimicoFactory {
private Map<String, ElementoQuimico> elementos;

public ElementoQuimico criarElemento(String simbolo, String nome) {
ElementoQuimico elemento = elementos.get(simbolo);
if (elemento == null) {
```

```
elemento = new ElementoQuimico(simbolo, nome);
elementos.put(simbolo, elemento);
}
return elemento;
}
}

public class ElementoQuimico {
private String simbolo;
private String nome;

ElementoQuimico(String simbolo, String nome) {
this.simbolo = simbolo;
this.nome = nome;
}
// getters e setters omitidos
}

public class Exemplo {
public static void main(String[] args) {
ElementoQuimicoFactory factory = new ElementoQuimicoFactory();
SubstanciaSimples s1 = new SubstanciaSimples("Oxigênio",
factory.criarElemento("O", "Oxigênio"), 2);
SubstanciaSimples s2 = new SubstanciaSimples("Ozônio",
factory.criarElemento("O", "Oxigênio"), 3);

Map<ElementoQuimico, Integer> composicaoAgua = new HashMap<>();
composicaoAgua.put(factory.criarElemento("H", "Hidrogenio"), 2);
composicaoAgua.put(factory.criarElemento("O", "Oxigênio"), 1);

SubstanciaComposta s3 = new SubstanciaComposta("Agua", composicaoAgua);
}
}
```

CONSEQUÊNCIAS E PADRÕES RELACIONADOS AO FLYWEIGHT

O padrão Flyweight permite a utilização mais racional da memória em casos nos quais exista a criação de um número elevado de pequenos objetos replicados. Ele é aplicável, especialmente, quando esses objetos possuírem um estado intrínseco imutável.

Um exemplo conhecido de objetos imutáveis em Java corresponde aos objetos da classe String.

As constantes literais do tipo String em Java são implementadas como objetos String imutáveis, compartilhados em um pool de constantes. Desse modo, apesar de uma constante literal String ser utilizada em vários módulos de um programa, apenas uma instância dessa constante é criada.

💬 COMENTÁRIO

O padrão Flyweight pode ser combinado com o padrão Composite na implementação de estruturas de árvores com folhas compartilhadas. Outro padrão frequentemente combinado com o Flyweight é o State.

No padrão State, cada estado de um objeto é representado como uma classe. Desse modo, se uma classe Pedido, por exemplo, tiver cinco estados possíveis e instanciarmos mil pedidos, teremos para cada pedido um objeto estado associado. Portanto, se essa estrutura for implementada, instanciando-se diretamente cada objeto, teremos dois mil objetos em memória. Entretanto, se o padrão Flyweight for utilizado, teremos apenas mil objetos da classe Pedido mais cinco objetos (um para cada estado possível) compartilhados por esses mil objetos.

Conseguiu perceber como o uso da memória fica mais racional nesse caso?

INTENÇÃO DO PADRÃO PROXY

O propósito do padrão Proxy **é fornecer aos clientes um objeto proxy, com a mesma interface do objeto destino real**, que delega as requisições dos clientes para o objeto real.

Esse padrão é muito utilizado quando o objeto real é remoto ou de instanciação muito custosa, como, por exemplo, o caso de imagens, ou quando se deseja interceptar a requisição antes que ela chegue ao objeto real para fins de autorização, monitoração, validação, entre outros.

PROBLEMA RESOLVIDO PELO PADRÃO PROXY

Quando estamos desenvolvendo um programa orientado a objetos, implementar a chamada de uma operação de outro objeto é uma tarefa simples caso eles estejam no mesmo processo. Imagine, agora, que esses objetos estejam em processos diversos, rodando em máquinas diferentes, como é comum em projetos distribuídos.

Para que um objeto consiga chamar uma operação de um objeto remoto, é necessário estabelecer uma conexão, empacotar os dados para transmissão e fazer a chamada, ou seja, não é simplesmente chamar uma operação como ocorre em um programa orientado a objetos simples.

💬 COMENTÁRIO

Para você entender melhor o problema, veja o código a seguir, que implementa a interação de um módulo cliente com um módulo EJB servidor que executa uma transferência entre contas. Imagine que o módulo cliente seja uma parte da implementação da camada de interface com o usuário que precisa solicitar esse serviço ao componente de negócio remoto. Se você não conhecer a tecnologia EJB, não tem problema.

O importante é você perceber a complexidade que foi inserida na implementação para que o módulo cliente conseguisse fazer a chamada da operação do componente EJB correspondente à lógica de negócio.

```
public class ModuloCliente {
private ServicoConta session; // referência para o objeto remoto
```

```
private static final Class homeClazz = ServicoContaHome.class;

public void executarTransferencia(Transferencia transf) throws ServicoContaException {
try {
ServicoContaHome home = (ServicoContaHome) getHome("ServicoConta", homeClazz);
session = home.create();
session.executarTransferencia(transf); // chamada delegada ao objeto real
} catch (Exception ex) {
throw new ServicoContaException(ex);
}
}
public EJBHome getHome(String name, Class clazz) throws NamingException {
Object objref = context.lookup(name);
EJBHome home = (EJBHome) PortableRemoteObject.narrow(objref, clazz);
return home;
}
}
```

Imagine, agora, que o sistema que você está desenvolvendo tenha dezenas de módulos clientes, que precisam interagir com os objetos EJB que realizam a lógica de negócio. Indo além, suponha que você precise mudar a tecnologia de invocação dos serviços de EJB para uma REST API.

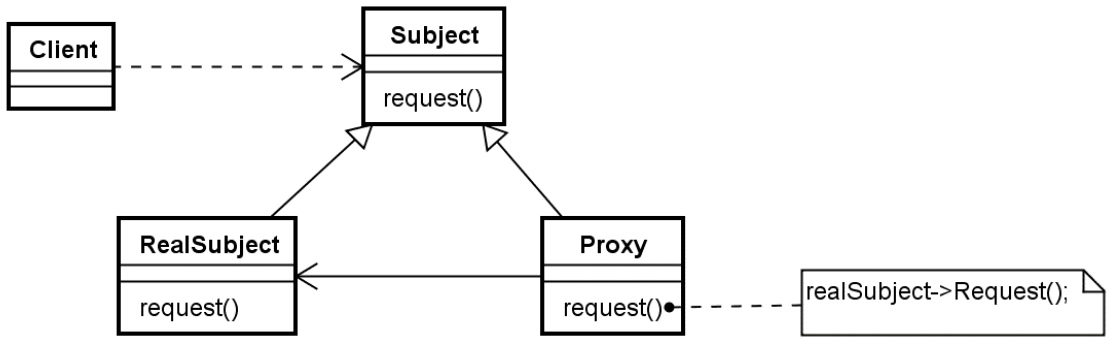
Qual seria o impacto nos módulos clientes?

Se pensou em um impacto gigantesco, você acertou!

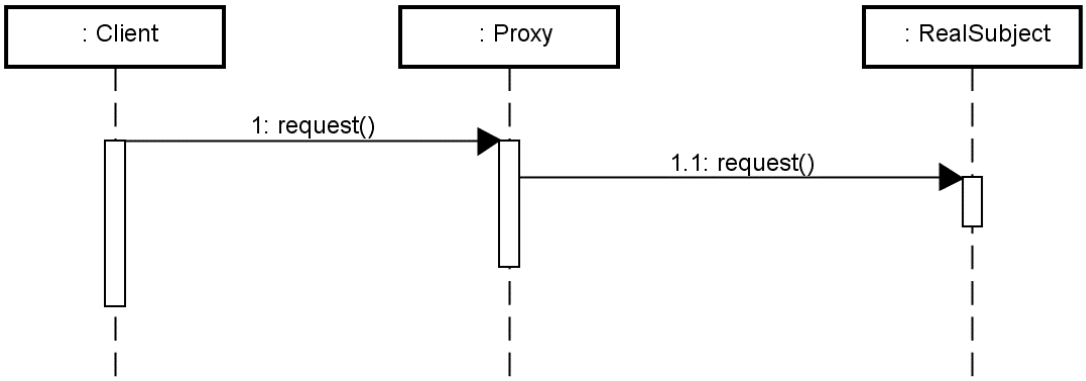
Embora o próprio EJB já contenha uma implementação do padrão Proxy, oferecendo um objeto proxy local que abstrai os detalhes de transporte e comunicação via rede com o objeto EJB remoto, o exemplo anterior mostra que os módulos clientes continuam tendo que lidar com detalhes da tecnologia envolvida na comunicação entre as camadas de uma aplicação.

SOLUÇÃO DO PADRÃO PROXY

A estrutura da solução proposta pelo padrão Proxy está representada no diagrama de classes a seguir.



A dinâmica entre os participantes é ilustrada no diagrama de sequência a seguir. Note que o papel do proxy é fundamentalmente repassar a chamada ao objeto destino.



O principal participante do padrão é o elemento Proxy que controla o acesso ao objeto destino. Existem diferentes tipos de proxy:

REMOTE PROXY

É responsável por oferecer uma interface local idêntica à implementada pelo componente remoto de destino (RealSubject), isolando o cliente das questões envolvendo estabelecimento de conexão, codificação do conteúdo da requisição e tratamento de erros. Esse proxy é utilizado na comunicação entre objetos distribuídos em diferentes máquinas.

VIRTUAL PROXY

É utilizado quando o objeto real tem uma carga muito custosa ou demorada. Uma interface com o usuário com imagens pesadas, por exemplo, pode fazer uso de virtual proxies para o conteúdo real das imagens, permitindo apresentar mensagens ou imagens mais leves e, somente sob o comando do usuário, trazer a imagem completa para a tela. Outro uso para esse tipo de proxy ocorre em frameworks de persistência de objetos, quando um objeto está relacionado a centenas de outros e desejamos acessar apenas as propriedades básicas do objeto, sem a necessidade de carregar todos os objetos relacionados, estratégia conhecida como lazy instantiation.

INTERCEPTOR PROXY

É utilizado quando a chamada para o objeto destino deve ser interceptada para realização de atividades como verificação de permissão de acesso, monitoração, redirecionamento da chamada, entre outras possibilidades, para, então, ser redirecionada ao objeto destino.

O código a seguir mostra a implementação de um proxy para o componente de negócio implementado em EJB. Esse é um padrão J2EE conhecido como BusinessDelegate, implementado utilizando o padrão de projeto proxy.

A classe ServicoContaProxyEJB implementa uma interface genérica (ServicoContaDelegate) que replica as operações oferecidas pelo objeto remoto destino. A implementação desse proxy encapsula o mecanismo de conexão com o componente de negócio, neste caso, EJB. Se implementarmos um mecanismo de acesso via REST API, bastaria criarmos outro proxy implementando a interface ServicoContaDelegate, delegando as chamadas para essa API. Perceba que a operação executarTransferencia delega a execução para o componente remoto, mas, para isso, o proxy precisa se conectar ao objeto EJB destino.

```
public interface ServicoContaDelegate {
void executarTransferencia(Transferencia tranf) throws ServicoContaException;
}

public class ServicoContaProxyEJB implements ServicoContaDelegate {
private ServicoConta session; // referência para o objeto remoto
private static final Class homeClazz = ServicoContaHome.class;

public void executarTransferencia(Transferencia transf) throws ServicoContaException {
try {
session = getSession(homeClazz); // conexão com o objeto remoto
session.executarTransferencia(transf); // chamada delegada ao objeto real
} catch (Exception ex) {
throw new ServicoContaException(ex);
}
}
public ServicoConta getSession(Class homeClazz) throws ServicoContaException {
try {
ServicoContaHome home = (ServicoContaHome) getHome("ServicoConta", homeClazz);
session = home.create();
```

```
} catch (Exception ex) {
throw new ServicoContaException(ex);
}
}

public EJBHome getHome(String name, Class clazz) throws NamingException {
Object objref = context.lookup(name);
EJBHome home = (EJBHome) PortableRemoteObject.narrow(objref, clazz);
return home;
}
}
```

O código a seguir mostra como ficariam os módulos clientes que precisassem requisitar um serviço de componente de negócio. Note que a implementação desses clientes está totalmente isolada dos detalhes de conexão e tratamento de exceções inerentes ao protocolo de comunicação com o objeto remoto.

```
public class ModuloCliente {
private ServicoContaDelegate servicoConta;
public ModuloCliente(ServicoContaDelegate servicoConta) {
this.servicoConta = servicoConta;
}

public void executarTransferencia(Transferencia transf) {
try {
servicoConta.executarTransferencia(transf);
} catch (ServicoContaException ex) {
// apresenta mensagem de erro
}
}
}
```

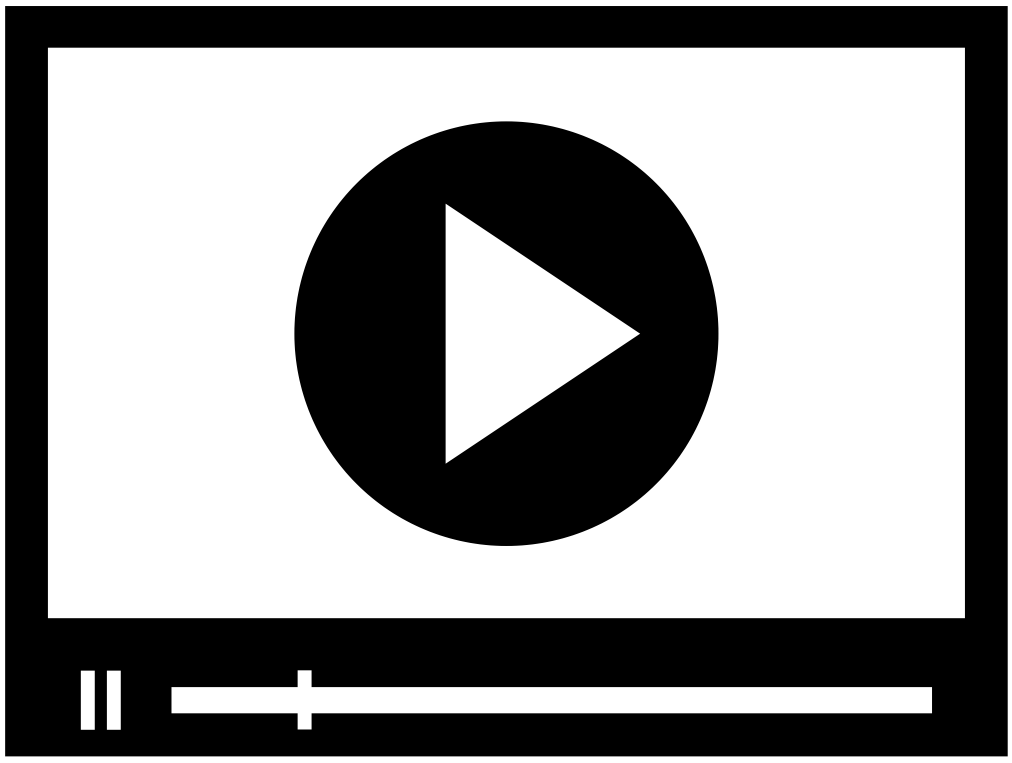
CONSEQUÊNCIAS E PADRÕES RELACIONADOS AO PROXY

O padrão Proxy introduz um nível de indireção para o acesso a um objeto. Esse nível adicional possibilita a transparência de localização do objeto destino para os objetos clientes, caso o objeto destino esteja em outra máquina. Além disso, é possível construir objetos grandes ou numerosos sob demanda sem mudar a interface oferecida ao módulo cliente.

COMENTÁRIO

O padrão Proxy possui algumas similaridades com o padrão Adapter. Ambos oferecem uma forma de acesso indireta ao objeto destino, porém, o padrão Adapter traduz uma interface padrão em uma interface específica oferecida, tipicamente, por um componente de terceiros, enquanto o padrão proxy preserva a mesma interface oferecida pelo objeto alvo.

O padrão Decorator possui uma implementação similar a um proxy, porém, tem um propósito diferente, por adicionar novas funcionalidades a um objeto, enquanto um proxy mantém as funcionalidades oferecidas pelo objeto destino.



PADRÕES DE PROJETO FLYWEIGHT E PROXY

No vídeo a seguir, apresentamos exemplos de aplicação dos padrões Flyweight e Proxy no desenvolvimento de software.

VERIFICANDO O APRENDIZADO

CONCLUSÃO

CONSIDERAÇÕES FINAIS

Neste conteúdo, vimos como os padrões de projeto GoF estruturais podem auxiliar na criação de soluções de projeto mais flexíveis e menos acopladas.

O padrão Adapter é muito utilizado para isolar módulos clientes das diferentes implementações de um mesmo serviço oferecidas por terceiros. O desafio fundamental do padrão Adapter reside na definição de uma interface comum às diversas soluções.

Os padrões Bridge e Adapter possuem uma estrutura de solução similar, porém, com objetivos distintos. Enquanto o padrão Adapter visa adaptar uma interface comum a diferentes implementações, tipicamente conhecidas a posteriori, o padrão Bridge promove uma separação a priori entre uma abstração e as possíveis implementações da sua representação interna, permitindo que ambas possam evoluir de forma independente.

O padrão Decorator é uma forma mais flexível de adicionar novos comportamentos a classes já existentes, se comparada às soluções baseadas em herança. A sua estrutura de solução é baseada na composição de objetos que compartilham uma interface genérica comum, gerando

estruturas similares a uma cebola ou a uma matrioska (você conhece aquela série de bonecas russas de tamanhos variados colocadas umas dentro das outras?).

O padrão Composite oferece uma estrutura de composição recursiva que permite o tratamento de hierarquias de objetos que possuam operações que se apliquem tanto às folhas como aos agregados. O padrão Facade é muito utilizado na estruturação da arquitetura de sistemas, oferecendo uma interface de alto nível para cada subsistema ou componente macro da solução. O padrão Flyweight oferece uma forma de utilização de memória mais eficiente em casos nos quais precisamos instanciar uma grande quantidade de objetos imutáveis ou que possuam uma parte significativa do seu estado imutável.

O padrão Proxy fornece um nível de indireção a um objeto, de forma parecida com o padrão Decorator, pois ambos possuem uma referência para um objeto para o qual as requisições são direcionadas. Entretanto, esses padrões possuem objetivos distintos. O Decorator se preocupa em adicionar funcionalidades ao objeto alvo, enquanto o Proxy oferece a mesma interface do objeto destino, porém, isolando o cliente de problemas como acesso remoto ou otimização de instanciação de objetos com utilização intensiva de memória.

PODCAST

Ouça o podcast. Com ele, encerramos o nosso estudo dos padrões GoF estruturais.

REFERÊNCIAS

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design patterns:** elements of reusable object-oriented software. 1. ed. Boston: Addison-Wesley, 1994.

METSKER, S. J.; WAKE, W. C. **Design patterns in Java.** 1.ed. Boston: Addison-Wesley, 2006.

EXPLORE+

Para saber mais sobre a programação orientada a objetos, acesse o site da DevMedia e leia o artigo *Utilização dos princípios SOLID na aplicação de padrões de projeto* .

O site **Padrões de Projeto/Design patterns – Refactoring.Guru** apresenta um conteúdo interativo e bastante completo de todos os padrões GoF, com exemplos de código em diversas linguagens de programação.

Além dos padrões GoF tradicionais, outros voltados para o desenvolvimento de aplicações corporativas em Java EE podem ser encontrados no livro *Java EE 8 Design Patterns and Best Practices* , escrito por Rhuan Rocha e João Purificação. A obra aborda padrões para interface com o usuário, lógica do negócio, integração de sistemas, orientação a aspectos, programação reativa e microsserviços.

CONTEUDISTA

Alexandre Luís Correa