Padrões de Projeto

Idéia proposta pelo arquiteto Christopher Alexander:
- A Pattern Language: Towns, Buildings, Construction, 1977.

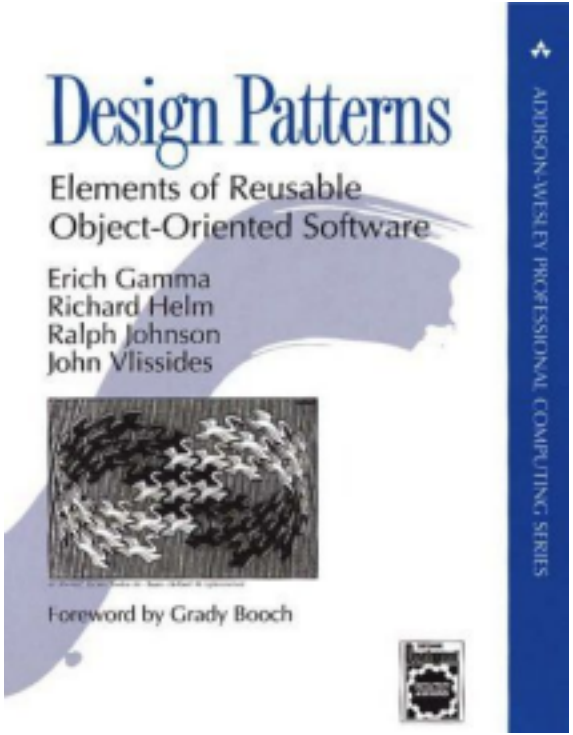
- "Cada padrão descreve um problema que ocorre repetidamente em nosso ambiente e, em seguida, descreve o núcleo da solução para esse problema, de tal forma que você

possa usar essa solução um milhão de vezes, sem nunca fazê-la da mesma maneira duas vezes.".

Padrões de Projeto

Na área de desenvolvimento de software, popularizados pelo livro:

- Design Patterns: Elements of Reusable Object-Oriented Software, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley, 1995

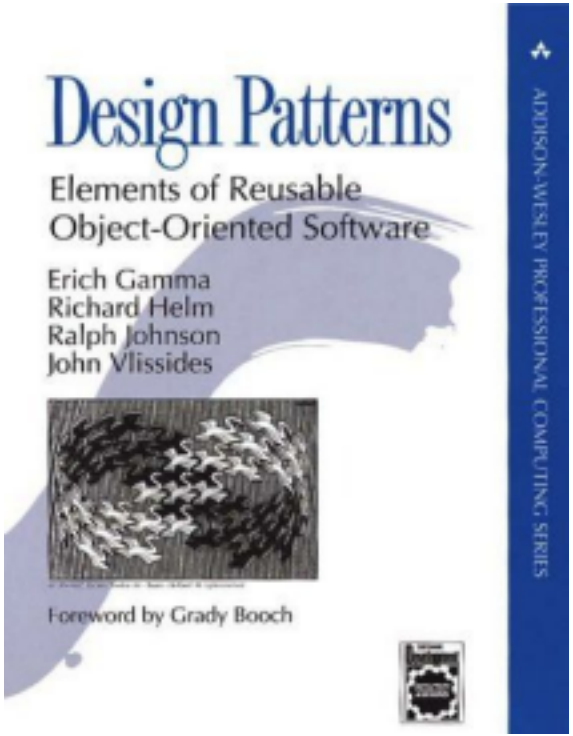


- Conhecido como “GoF (Gang of Four) book”

Padrões de Projeto

Definição segundo GoF: "descrições de objetos e classes de comunicação que são personalizados para resolver um problema geral de design em um contexto específico.."

Objetivo de padrões na comunidade de projetistas de sistemas: Criar um “catálogo de soluções” que possam ser reutilizadas em problemas que são recorrentes no desenvolvimento de sistemas .



Padrões de Projeto

Segundo o seu propósito, padrões podem ser:

Criacionais: tratam do processo de criação de objetos

Estruturais: tratam da composição de classes e objetos

Comportamentais: tratam das interações e distribuição de responsabilidades entre classes e objetos.

O Formato de um padrão

Nome

um bom nome é essencial para que o padrão caia na boca do povo

Propósito

uma breve descrição do objetivo do padrão

Problema/Solução

um cenário mostrando o problema e a necessidade da solução

Aplicabilidade

como reconhecer as situações nas quais o padrão é aplicável

Criacionais	Estruturais	Comportamentais
Abstract factory	Adapter	Chain of responsibility
Factory method	Bridge	Command
Prototype	Composite	Interpreter
Singleton	Decorator	Iterator
Builder	Façade	Mediator
	Flyweight	Memento
	Proxy	Observer
		State
		Strategy
		Template method
		Visitor

Importante

- Padrões de projeto ajudam em **design for change**
- Facilitam mudanças futuras no código
- Se o código dificilmente vai precisar de mudar, então uso de padrões é um exemplo de “overengineering” (uma solução que recebe mais funcionalidades, funções e recursos do que realmente precisa, sendo desenvolvida de maneira mais complexa do que o necessário.)

PADRÕES CRIACIONAIS

[redacted]

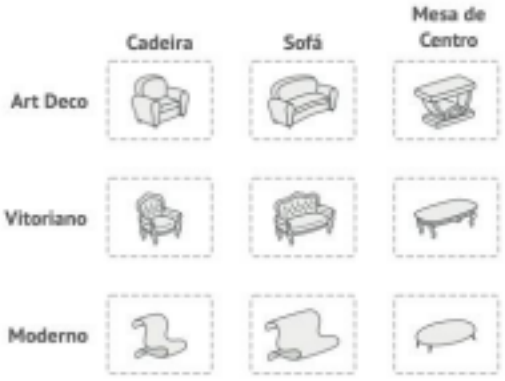
Propósito

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

Problema

Criação de um simulador de loja de móveis, onde há famílias de produtos relacionados (cadeira, sofá, mesa de centro) e suas variantes (moderno, vitoriano, art déco).

O desafio é criar objetos de móveis individuais que



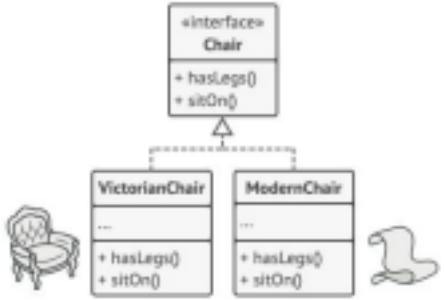
Famílias de produtos e suas variantes.

combinem com outros da mesma família, sem alterar o código ao adicionar novos produtos ou famílias.

Solução

A solução proposta pelo padrão Abstract Factory começa com a declaração de interfaces para cada produto distinto da família, como cadeira, sofá e mesa de centro.

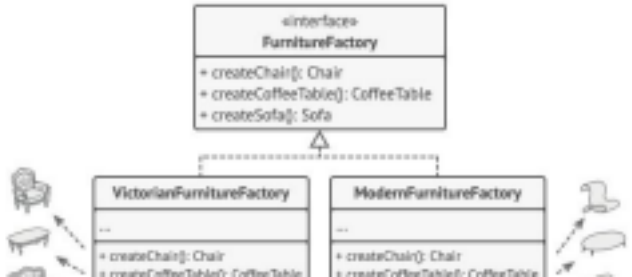
Variantes dos produtos implementam essas interfaces.



Todas as variantes do mesmo objeto podem ser movidas para uma mesma hierarquia de classe.

Em seguida, é criada uma fábrica abstrata, uma interface com métodos de criação para todos os produtos da família.

Cada variante de produto tem sua própria classe de fábrica baseada nessa interface.

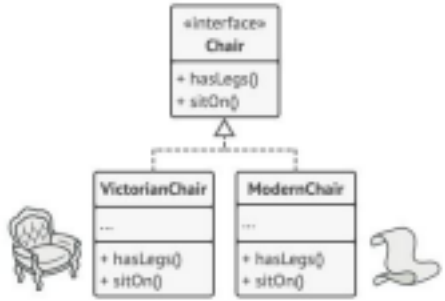


Solução

O cliente trabalha com fábricas e produtos por meio de interfaces abstratas, permitindo a troca de tipo de fábrica e variante de produto sem afetar o código existente.

O cliente não precisa conhecer a classe concreta da fábrica ou a variante específica do produto, tratando todos os produtos da mesma família de maneira uniforme.

A criação de objetos de fábrica geralmente ocorre durante a inicialização do programa, com a seleção do tipo de fábrica dependendo da configuração ou definições de ambiente.



Todas as variantes do mesmo objeto podem ser movidas para uma mesma hierarquia de classe.

Exemplo 1: Interface de Usuário com Temas Diferentes

Imagine que você está desenvolvendo uma aplicação que suporta diferentes temas de interface de usuário, como tema claro e tema escuro.
Cada tema possui seus próprios componentes, como botões, janelas e menus, que precisam ser criados de forma consistente.
Usando o padrão Abstract Factory, você pode criar uma fábrica abstrata UIFactory com métodos para criar esses componentes (createButton(), createWindow(), createMenu()).
Subclasses como LightThemeFactory e DarkThemeFactory implementam esses métodos para criar os componentes específicos para cada tema.

Exemplo 2: Sistemas de Banco de Dados com Diferentes SGBDs

Em uma aplicação que deve ser compatível com diferentes sistemas de gerenciamento de banco de dados (SGBDs), como MySQL, PostgreSQL e Oracle, você pode usar o padrão Abstract Factory para centralizar a criação de objetos relacionados ao banco de dados. Você pode criar uma fábrica abstrata DBFactory com métodos para criar conexões (createConnection()), comandos (createCommand()) e leitores de dados (createDataReader()). Subclasses como MySQLFactory, PostgreSQLFactory e OracleFactory implementam esses métodos para criar objetos específicos para cada SGBD.

Exemplo 3: Aplicações Multiplataforma

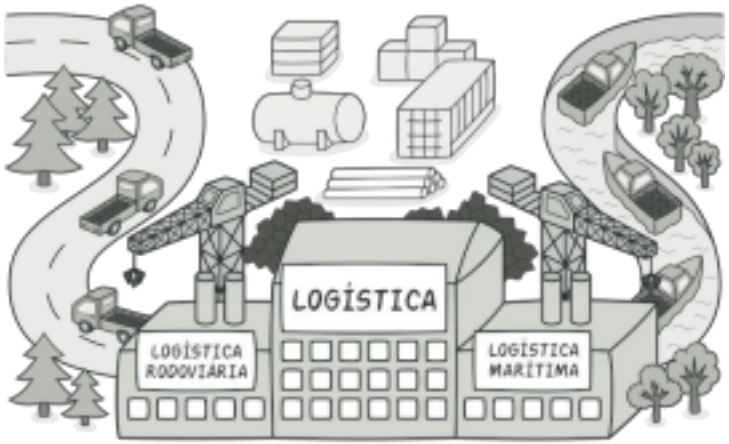
Para aplicações que precisam rodar em diferentes plataformas, como Windows, macOS e Linux, o padrão Abstract Factory pode ser usado para criar objetos específicos da

plataforma.

Você pode ter uma fábrica abstrata PlatformFactory com métodos para criar janelas (createWindow()), menus (createMenu()) e caixas de diálogo (createDialog()). Subclasses como WindowsFactory, MacOSFactory e LinuxFactory implementam esses métodos para criar componentes específicos para cada plataforma.

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.



[redacted]

[redacted]

[redacted]

[redacted]

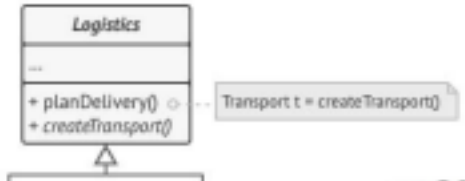
[redacted]

E se depois de um tempo surgir outros tipos de transporte? Marítimo, ferroviário, aéreo etc.

[redacted]

[redacted]

O padrão **Factory Method** sugere que você substitua chamadas

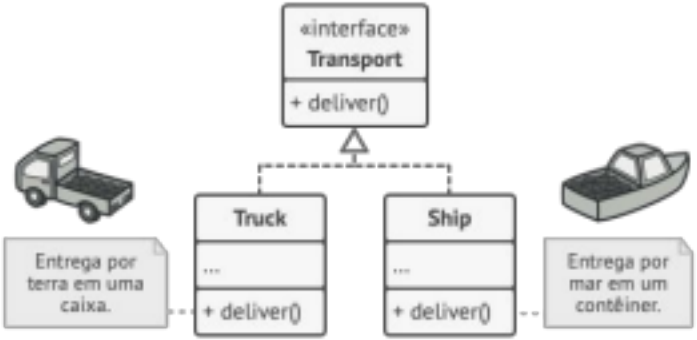


diretas de construção de objetos (usando o operador `new`) por chamadas para um método *fábrica* especial.

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

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.**



Exemplo 1: Aplicações Gráficas com Diferentes Formatos de Documento

Imagine que você está desenvolvendo uma aplicação gráfica que suporta a criação e leitura de diferentes formatos de documento, como PDF, DOCX e TXT.
Em vez de ter um único método de criação de documentos que contenha lógica condicional para cada formato, você pode usar o padrão Factory Method.
Cada formato de documento pode ter sua própria classe específica, e você pode ter uma classe abstrata Document com um método createDocument().
Subclasses como PDFDocument, DOCXDocument e TXTDocument podem implementar esse método para criar instâncias específicas de documentos.

Exemplo 2: Aplicações Bancárias com Diferentes Tipos de Contas

Em uma aplicação bancária, você pode ter diferentes tipos de contas, como conta corrente, conta poupança e conta salário.

Em vez de ter um método de criação de contas que usa lógica condicional para instanciar diferentes tipos de contas, você pode usar o padrão Factory Method.

Você pode ter uma classe base Account com um método createAccount(), e subclasses como CheckingAccount, SavingsAccount e SalaryAccount podem implementar esse método para criar instâncias específicas de contas.

Exemplo 3: Sistemas de Notificação com Diferentes Métodos de Notificação

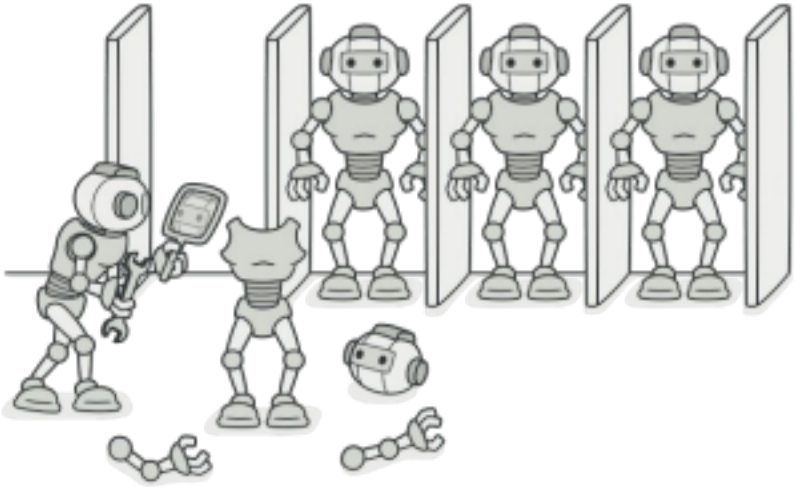
Em um sistema de notificação, você pode ter diferentes métodos de notificação, como e-mail, SMS e push notifications.

Em vez de ter um método de criação de notificações que usa lógica condicional para instanciar diferentes tipos de notificações, você pode usar o padrão Factory Method.

Você pode ter uma classe base Notification com um método createNotification(), e subclasses como EmailNotification, SMSNotification e PushNotification podem implementar esse método para criar instâncias específicas de notificações.

Prototype

Propósito



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

Prototype

Problema

Digamos que você tenha um objeto, e você quer criar uma cópia exata dele.

Como você faria? Primeiro, você tem que criar um novo objeto da mesma classe. Então você terá que ir por todos os campos do objeto original e copiar seus valores para o novo objeto.

Porém, nem todos os objetos podem ser copiados dessa forma porque alguns campos de objeto podem ser privados e não serão visíveis fora do próprio objeto.

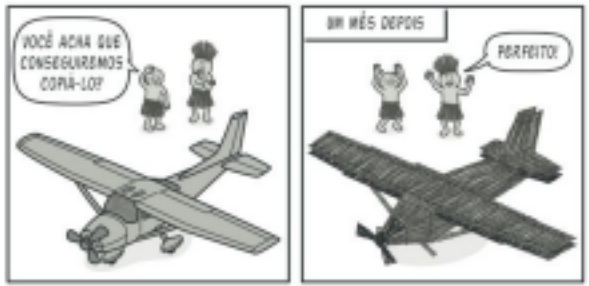
25

Prototype

Uma vez que você precisa saber a classe do objeto para criar uma cópia, seu código se torna dependente daquela classe.

Algumas vezes você só sabe a interface que o objeto segue, mas não sua classe concreta, quando, por exemplo, um parâmetro em um método aceita quaisquer objetos que seguem uma interface.

26



Copiando um objeto "de fora" nem sempre é possível.

Prototype

Solução

O padrão Prototype delega o processo de clonagem para o próprio objeto que está sendo clonado.

O padrão declara um interface comum para todos os objetos que suportam clonagem. Essa interface permite que você clone um objeto sem acoplar seu código à classe daquele objeto.

Geralmente, tal interface contém apenas um único método `clonar`.

A implementação do método `clonar` é muito parecida em todas as classes. O método cria um objeto da classe atual e carrega todos os valores de campo para do antigo objeto para o novo. Você pode até mesmo copiar campos privados porque a maioria das linguagens de programação permite objetos acessar campos privados de outros objetos que pertençam a mesma classe.

Um objeto que suporta clonagem é chamado de um *protótipo*. Quando seus objetos têm dúzias de campos e centenas de possíveis configurações, cloná-los pode servir como uma alternativa à subclasses.

Exemplo 1: Processamento de Documentos

Imagine que você está desenvolvendo um sistema de gerenciamento de documentos para uma empresa. Cada documento pode ter várias configurações, formatos e conteúdos. Criar um novo documento do zero pode ser trabalhoso e demorado, especialmente se muitas das configurações e formatos forem repetitivos. Em vez disso, você pode criar protótipos de documentos para diferentes tipos (relatório, fatura, contrato) e cloná-los quando precisar de novos documentos. Isso economiza tempo e garante que os novos documentos sejam consistentes com os padrões da empresa.

Prototype

Exemplo 2: Jogos de Vídeo

Em um jogo de vídeo, você pode ter diferentes tipos de personagens, inimigos ou itens que são criados repetidamente. Por exemplo, se você tiver um inimigo básico com várias configurações

(vida, força, habilidades), criar novos inimigos do zero pode ser ineficiente. Usando o padrão Prototype, você pode criar um protótipo de inimigo básico e cloná-lo para criar novos inimigos. Isso permite que você adicione pequenas variações aos inimigos clonados sem precisar redefinir todas as propriedades.

Prototype

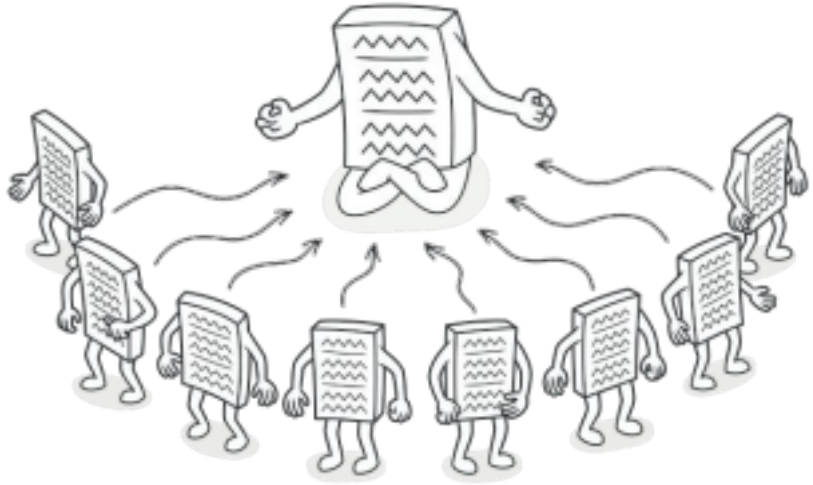
Exemplo 3: Prototipagem de Interfaces de Usuário

Ao desenvolver interfaces de usuário (UI) para aplicações, você pode ter vários componentes repetitivos, como botões, caixas de texto, menus, etc. Cada componente pode ter várias configurações e estilos. Em vez de criar novos componentes do zero toda vez, você pode criar protótipos de cada componente com as configurações padrão e cloná-los quando necessário. Isso facilita a manutenção de uma aparência e comportamento consistentes em toda a aplicação.

Singleton

Propósito

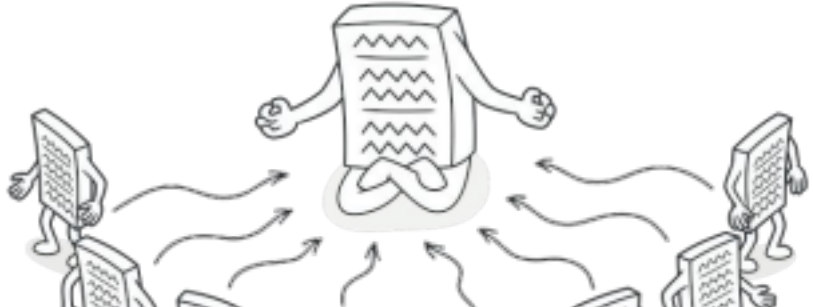
O Singleton é um padrão de projeto criacional que permite a você garantir que uma classe tenha apenas uma instância, enquanto provê um ponto de acesso global para essa instância.



Singleton

Problema

O padrão Singleton resolve dois problemas de uma só vez, violando o *princípio de responsabilidade única*:



Garantir que uma classe tenha apenas uma única instância.

A razão mais comum para isso é para controlar o acesso a algum recurso compartilhado—por exemplo, uma base de dados ou um arquivo.

Imagine que você criou um objeto, mas depois de um tempo você decidiu criar um novo. Ao invés de receber um objeto fresco, você obterá um que já foi criado.

Singleton

Assim como uma variável global, o padrão Singleton permite que você acesse algum objeto de qualquer lugar no programa.

Contudo, ele também protege aquela instância de ser sobrescrita por outro código.

Há outro lado para esse problema: você não quer que o código que resolve o problema #1 fique espalhado por todo seu programa. É muito melhor tê-lo dentro de uma classe, especialmente se o resto do seu código já depende dela.

Singleton

Solução

Todas as implementações do Singleton tem esses dois passos em comum:

- Fazer o construtor padrão privado, para prevenir que outros objetos usem o operador `new` com a classe singleton.
- Criar um método estático de criação que age como um construtor. Esse método chama o construtor privado por debaixo dos panos para criar um objeto e o salva em um campo estático. Todas as chamadas seguintes para esse método retornam o objeto em cache.

Se o seu código tem acesso à classe singleton, então ele será capaz de chamar o método estático da singleton. Então sempre que aquele método é chamado, o mesmo objeto é retornado.

Singleton

Exemplo 1: Logger

Imagine que você está desenvolvendo uma aplicação que precisa registrar logs. Ter múltiplas instâncias de uma classe de logger pode causar confusão e inconsistência nos logs. Usando o padrão Singleton, você pode garantir que apenas uma instância do logger exista, centralizando todas as operações de logging e garantindo consistência.

Singleton

Exemplo 2: Conexão com Banco de Dados

Em uma aplicação que interage com um banco de dados, ter múltiplas conexões simultâneas pode ser ineficiente e difícil de gerenciar. Usando o padrão Singleton, você pode garantir que apenas uma conexão com o banco de dados seja criada e reutilizada em toda a aplicação, melhorando a eficiência e facilitando a gestão de recursos.

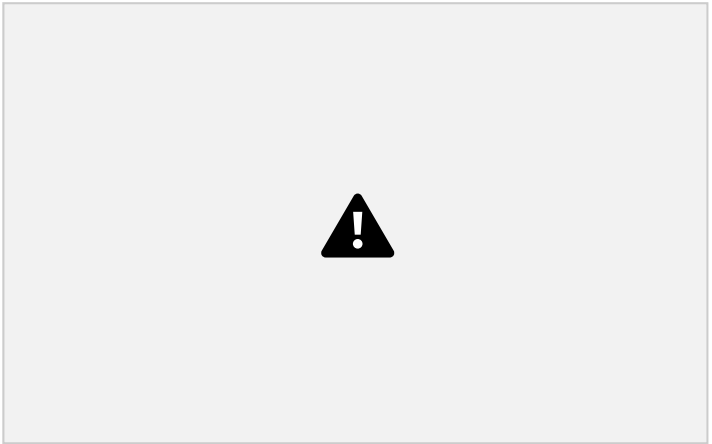
Exemplo 3: Gerenciador de Configurações

Para uma aplicação que lê configurações de um arquivo ou de um serviço de

configuração centralizado, você pode usar o padrão Singleton para garantir que as configurações sejam lidas e mantidas em uma única instância. Isso evita leituras repetidas e garante que todas as partes da aplicação usem a mesma configuração.

Propósito

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

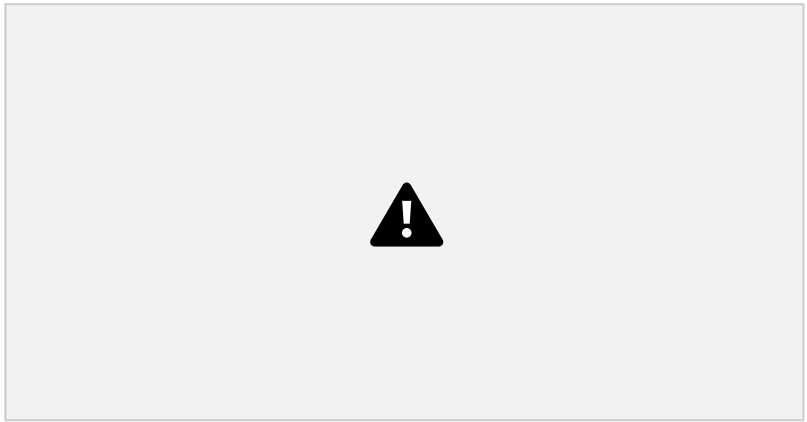


Problema

Criar objetos complexos que requerem uma inicialização passo a passo com muitos campos e objetos agrupados.

O código de inicialização muitas vezes se torna complicado, embutido em um construtor com vários parâmetros ou espalhado pelo código cliente.

A criação de muitas subclasses para diferentes configurações do objeto também pode tornar o programa complexo.



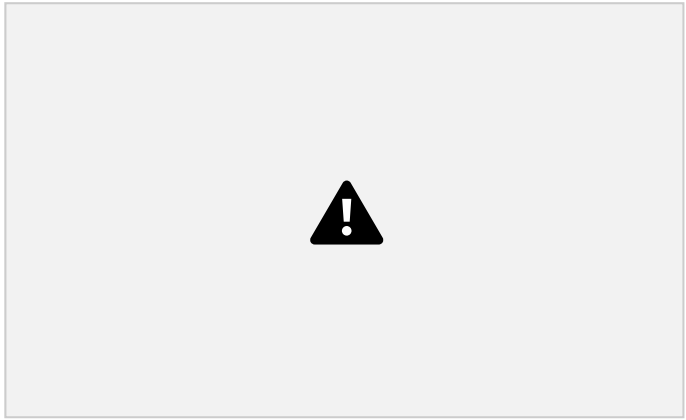
Solução

A solução proposta pelo padrão Builder envolve a extração do código de construção para objetos separados chamados builders. O padrão organiza a construção de objetos em uma série de etapas, permitindo a construção passo a passo de objetos complexos.

O Builder não permite que outros objetos acessem o produto enquanto ele está sendo construído.

Os builders executam etapas de construção necessárias para criar configurações específicas do objeto, eliminando a necessidade de chamar todas as etapas.

Diferentes builders podem implementar as mesmas etapas de construção de maneiras diferentes, permitindo a construção de várias representações do produto.



Exemplo 1: Construção de Objetos Imutáveis

Imagine que você está desenvolvendo um sistema que lida com objetos imutáveis, como uma classe Pessoa que possui muitos atributos opcionais, como nome, idade, endereço, telefone, etc. Usar um construtor tradicional com todos esses parâmetros pode ser confuso e propenso a erros.

Exemplo 2: Configuração de Conexão de Rede

Em uma aplicação que precisa configurar conexões de rede com várias opções, como tempo de espera, tamanho do buffer, e protocolo, o padrão Builder pode tornar o código mais claro e gerenciável. Em vez de usar um construtor com muitos parâmetros, você

pode usar um builder para configurar a conexão de forma mais legível.

Exemplo 3: Construção de Objetos de Configuração

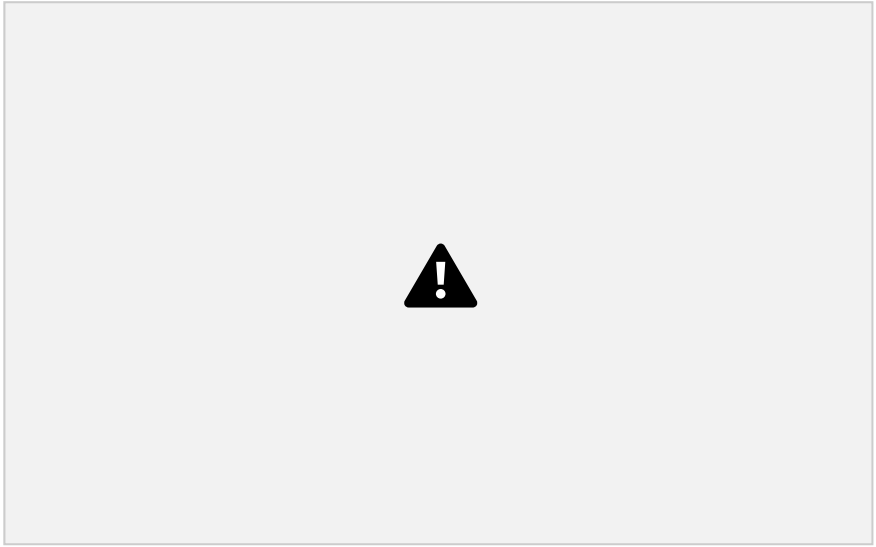
Para uma aplicação que precisa configurar objetos com muitas opções diferentes, como um servidor web, o padrão Builder pode ajudar a criar configurações de forma legível e organizada.

PADRÕES ESTRUTURAIS

Adapter

Propósito

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



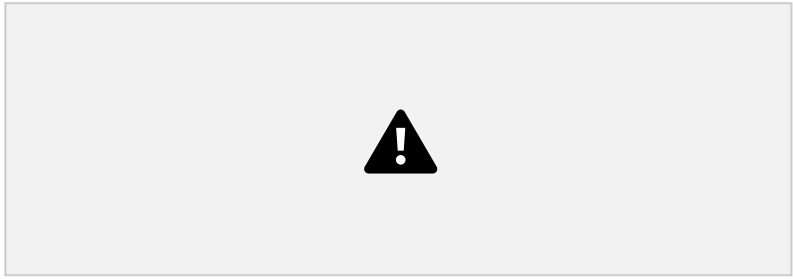
Adapter

Problema

Imagine que você está criando uma aplicação de monitoramento do mercado de ações da bolsa.

A aplicação baixa os dados as ações de múltiplas fontes em formato XML e então mostra gráficos e diagramas maneiros para o usuário.

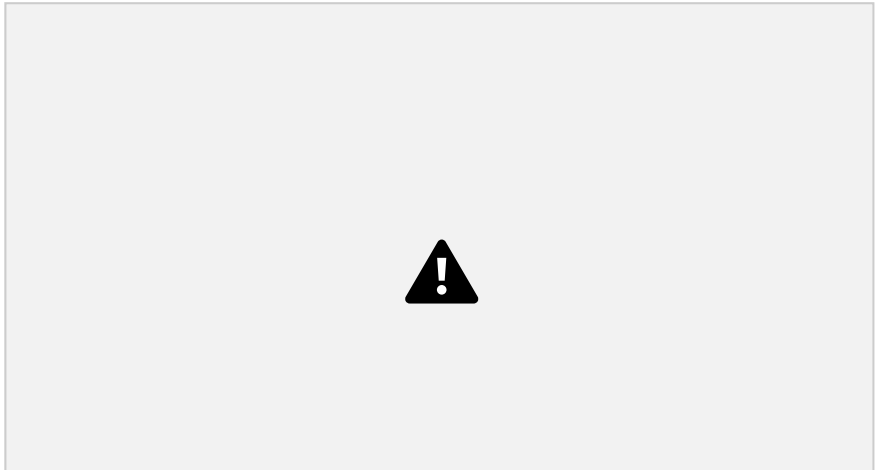
Imagine que você precisa usar outra biblioteca que só trabalha com dados em formato JSON.



Adapter

Solução

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



Um adaptador encobre um dos objetos para esconder a complexidade da conversão acontecendo nos bastidores.

O objeto encobrido nem fica ciente do adaptador.

Por exemplo, você pode encobrir um objeto que opera em metros e quilômetros com um adaptador que converte todos os dados para unidades imperiais tais como pés e milhas.

Adapter

Exemplo 1: Adaptador de API de Terceiros

Contexto: Imagine um aplicativo que precisa usar uma API de terceiros que possui uma interface diferente da sua aplicação.

Problema: A API externa usa métodos e estruturas de dados diferentes das que você utiliza no seu código.

Solução: Crie um adaptador que converta as chamadas da sua aplicação para o formato esperado pela API de terceiros.

Exemplo 2: Adaptador de Biblioteca de Logging
Contexto: Você tem uma aplicação que usa uma biblioteca de logging específica, mas decide mudar para outra biblioteca de logging.
Problema: A nova biblioteca de logging possui uma interface diferente da anterior. Solução: Crie um adaptador que implemente a interface da antiga biblioteca, mas use a nova biblioteca para realizar as operações de logging.

Bridge

Propósito

O Bridge é um padrão de projeto estrutural que permite que você divida uma classe grande ou um conjunto de classes intimamente ligadas em duas hierarquias separadas que são

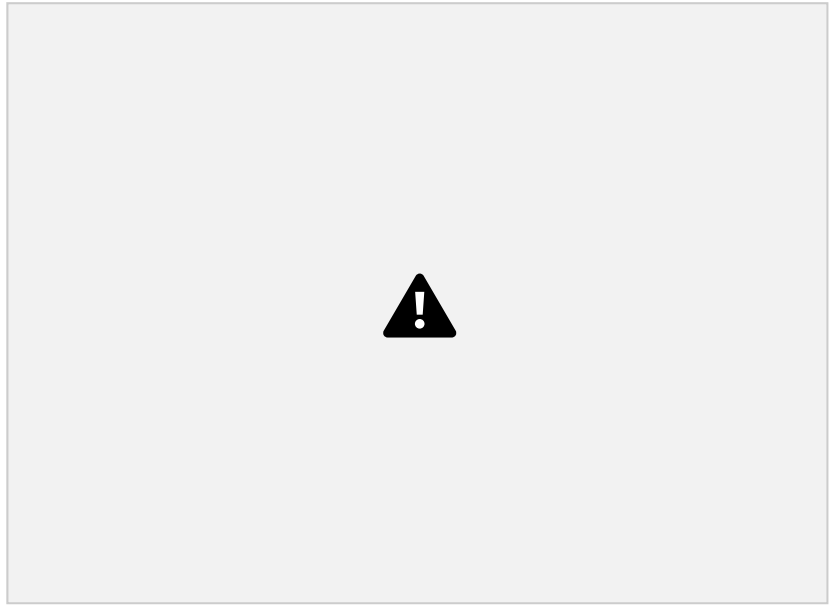
Que podem ser desenvolvidas independentemente umas das outras.

Bridge

Problema

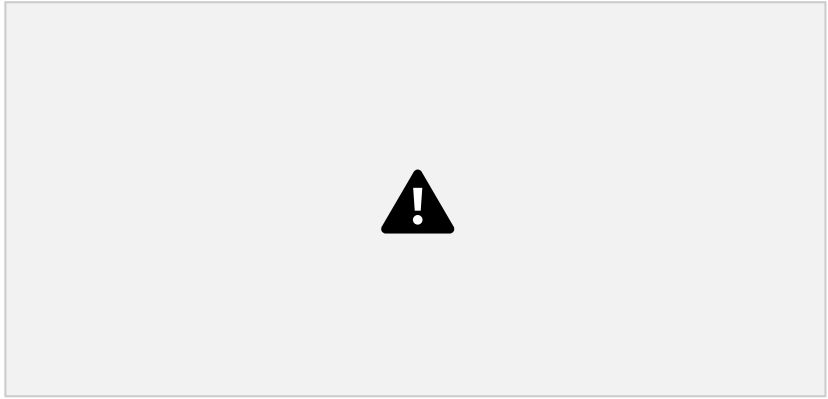
Digamos que você tem uma classe `Forma` geométrica com um par de subclasses: `Círculo` e `Quadrado`.

Você quer estender essa hierarquia de classe para incorporar cores, então você planeja criar as subclasses de forma `Vermelho` e `Azul`.



Contudo, já que você já tem duas subclasses, você precisa criar quatro combinações de classe tais como `CírculoAzul` e `QuadradoVermelho`.

Bridge



Solução

Esse problema ocorre porque estamos tentando estender as classes de forma em duas dimensões diferentes: por forma e por cor. Isso é um problema muito comum com herança de classe.

O padrão Bridge tenta resolver esse problema ao trocar de herança para composição do objeto.

Isso significa que você extrai uma das dimensões em uma hierarquia de classe separada, para que as classes originais

referenciem um objeto da nova hierarquia, ao invés de ter todos os seus estados e comportamentos dentro de uma classe.

Bridge

Exemplo 1: Aplicações de Desenho Gráfico
Contexto: Uma aplicação de desenho gráfico suporta diferentes formas geométricas (como Círculo e Quadrado) e diferentes tipos de renderização (como Raster e Vetor).
Problema: Queremos ser capazes de adicionar novas formas e novos métodos de renderização facilmente, sem modificar muito o código existente.
Solução: Usar o padrão Bridge para separar a hierarquia das formas da hierarquia dos métodos de renderização.

Exemplo 2: Dispositivos de Áudio
Contexto: Um sistema de controle de áudio que suporta diferentes dispositivos de áudio (como Rádio e MP3 Player) e diferentes tipos de controle (como Controle Básico e Controle Avançado).
Problema: Queremos ser capazes de adicionar novos dispositivos de áudio e novos tipos de controle sem modificar o código existente.

Solução: Usar o padrão Bridge para separar a hierarquia dos dispositivos de áudio da hierarquia dos controles.

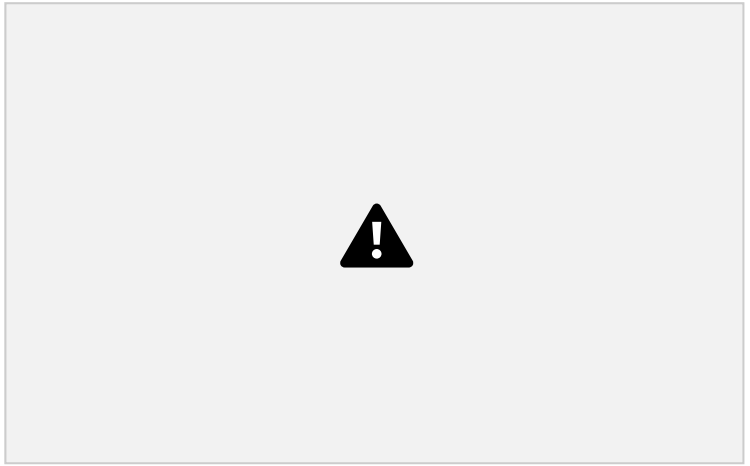
51

Composite

Propósito

O Composite é um padrão de projeto estrutural que permite que você componha objetos em estruturas de árvores e então trabalhe com essas estruturas como se elas fossem objetos individuais.

52



Composite

Problema

Usar o padrão Composite faz sentido apenas quando o modelo central de sua aplicação pode ser representada como uma árvore.

Por exemplo, imagine que você tem dois tipos de objetos: `Produtos` e `Caixas`.

Uma `Caixa` pode conter diversos `Produtos` bem como um número de `Caixas` menores. Essas `Caixas` menores também podem ter alguns `Produtos` ou até mesmo `Caixas` menores que elas, e assim em diante.

Como você faria para determinar o preço total de um pedido?

Composite

Solução

O padrão Composite sugere que você trabalhe com `Produtos` e `Caixas` através de uma interface comum que declara um método para a contagem do preço total.

Como esse método funcionaria? Para um produto, ele simplesmente retornaria o preço dele.

Para uma caixa, ele teria que ver cada item que ela contém, perguntar seu preço e então retornar o total para essa caixa.

Se um desses itens for uma caixa menor, aquela caixa também deve verificar seu conteúdo e assim em diante, até que o preço de todos os componentes internos sejam calculados. Uma caixa pode até adicionar um custo extra para o preço final, como um preço de embalagem.

Exemplo 1: Sistema de Arquivos

Pense em como os sistemas de arquivos funcionam no seu computador. Um sistema de arquivos é uma estrutura hierárquica onde diretórios (pastas) podem conter arquivos e outras pastas.

- **Arquivo**: Representa um arquivo simples no sistema.
- **Diretório**: Representa uma pasta que pode conter múltiplos arquivos ou outras pastas (subdiretórios).

No padrão **Composite**, um diretório pode ser tratado de maneira semelhante a um arquivo, mesmo que contenha outros arquivos e diretórios dentro dele. Por exemplo, se você calcular o tamanho total de um diretório, você somaria o tamanho de todos os arquivos e subdiretórios contidos nele.

Esse é um exemplo clássico do padrão Composite, onde a árvore de diretórios pode ser navegada e manipulada de maneira uniforme, sem se preocupar se o item atual é um arquivo individual ou um diretório contendo outros itens.

Composite

Exemplo 2: Organização de uma Empresa

Uma empresa geralmente tem uma estrutura organizacional hierárquica, onde há diferentes níveis de empregados, desde os executivos até os funcionários.

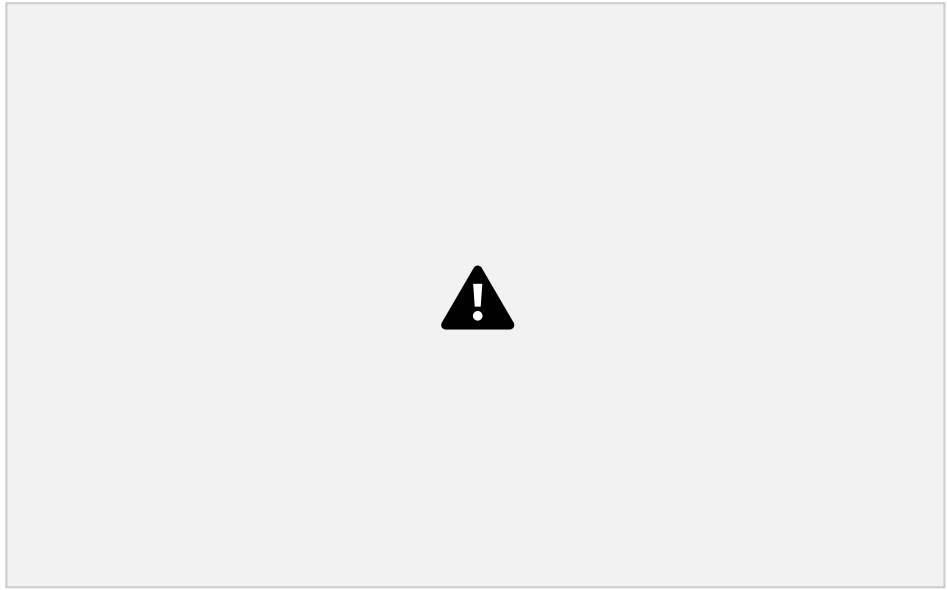
- **Funcionário**: Representa um funcionário individual que não gerencia outros funcionários. ● **Gestor**: Representa um gerente que supervisiona outros funcionários (que podem ser outros gestores ou funcionários individuais).

Na estrutura **Composite**, tanto os funcionários individuais quanto os gestores podem ser tratados de maneira uniforme. Por exemplo, se a empresa deseja enviar uma mensagem para todos os seus empregados, ela pode começar com o CEO (um gestor) e enviar a mensagem para todos os subordinados do CEO, cada um dos quais pode ser um gestor (que enviará para seus subordinados) ou um funcionário individual (que simplesmente receberá a mensagem).

Decorator

Propósito

O Decorator é um padrão de projeto estrutural que permite que você acople novos comportamentos para objetos ao colocá-los dentro de invólucros de objetos que contém os comportamentos.



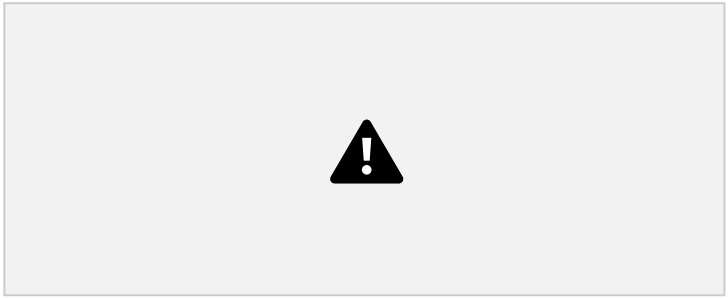
Decorator

Problema

Criação de uma biblioteca de notificação, inicialmente baseada em uma classe Notificador.

Os usuários da biblioteca desejam diferentes tipos de notificações, como SMS, Facebook, e Slack.

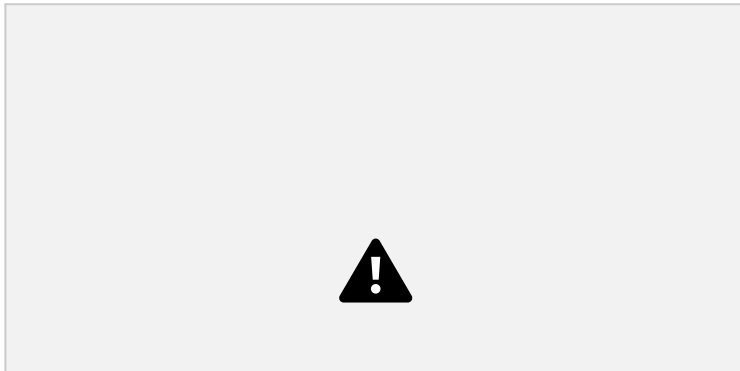
A solução inicial envolve a criação de subclasses para cada tipo de notificação, mas isso resulta em código inflado e complexo.



Decorator

Solução

Utilizar o padrão Decorator, evitando as limitações da herança.



A herança é estática, e as subclasses só podem ter uma classe pai, o que pode se tornar impraticável quando se deseja combinar diferentes tipos de notificações.

O Decorator utiliza a agregação ou composição para criar envoltórios (wrappers) que contêm referências aos objetos alvo e delegam funcionalidades a eles.

Decorator

Solução

O padrão Decorator permite que objetos sejam envolvidos em múltiplos decoradores, adicionando comportamento combinado.

A implementação prática envolve envolver um objeto Notificador básico em um conjunto de decoradores que correspondem às preferências do cliente.

Os objetos resultantes formam uma pilha, e o cliente interage com o último decorador na pilha. Isso permite a configuração flexível de pilhas complexas de notificações decorators, atendendo às necessidades específicas dos usuários.

O Decorator é apresentado como uma alternativa flexível e dinâmica à herança para adicionar funcionalidades a objetos existentes.

Decorator

Exemplo 1: Personalização de Bebidas em uma Cafeteria

Imagine que você está em uma cafeteria, e você pede um café. O café básico custa um determinado valor, mas você pode adicionar extras, como leite, chantilly, ou xarope de sabor. Cada um desses extras aumenta o custo da bebida e modifica sua descrição.

- **Café Simples**: Um café básico sem extras.
- **Decorator de Leite**: Adiciona leite ao café.
- **Decorator de Chantilly**: Adiciona chantilly ao café.

- **Decorator de Xarope**: Adiciona xarope ao café.

Se você pedir um "Café com Leite e Chantilly", a bebida básica é um café simples, mas você pode decorá-la com leite e chantilly. Cada um desses decoradores adiciona algo à bebida sem alterar a classe original de café.

Decorator

Exemplo 2: Personalização de um Carro

Pense em como você pode personalizar um carro ao comprar um veículo novo. O modelo básico de um carro vem com funcionalidades padrão, mas você pode adicionar vários opcionais, como sistema de som premium, teto solar, ou aquecimento nos bancos. Cada um desses opcionais adiciona um custo extra e funcionalidades ao carro.

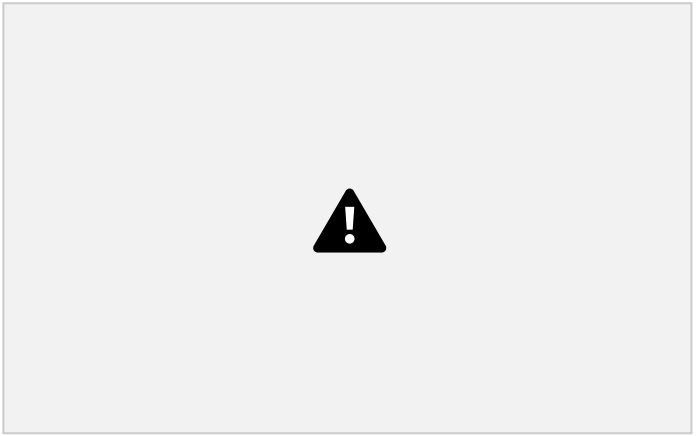
- **Carro Básico**: Um carro com funcionalidades padrão.
- **Decorator de Sistema de Som Premium**: Adiciona um sistema de som avançado.
- **Decorator de Teto Solar**: Adiciona um teto solar.
- **Decorator de Aquecimento nos Bancos**: Adiciona aquecimento nos bancos.

Se você optar por um "Carro com Sistema de Som Premium e Teto Solar", você começa com um carro básico e o decora com cada um dos opcionais desejados. Cada decorador adiciona uma nova funcionalidade ao carro original, sem alterar a classe

do carro básico.

Propósito

O Facade é um padrão de projeto estrutural que fornece uma interface simplificada para uma biblioteca, um framework, ou qualquer conjunto complexo de classes.

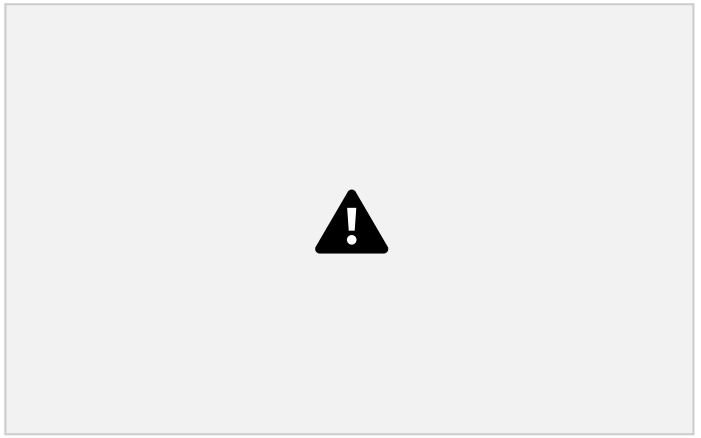


Problema

Imagine que você precisa fazer seu código funcionar com um amplo conjunto de objetos que pertencem a uma biblioteca ou framework.

Normalmente, você precisaria inicializar os objetos, rastrear as dependências, executar métodos na ordem correta, e assim por diante.

Problema: seu código pode ficar firmemente acoplada aos detalhes de implementação das classes de terceiros, tornando difícil a compreensão e manutenção.



Solução

Uma fachada é uma classe que fornece uma interface **simples** para um subsistema **complexo**

que contém muitas partes que se movem.

Uma fachada pode fornecer funcionalidades limitadas em comparação com trabalhar com os subsistemas diretamente. Contudo, ela inclui apenas aquelas funcionalidades que o cliente se importa.

Ter uma fachada é útil quando você precisa integrar sua aplicação com uma biblioteca sofisticada que tem dúzias de funcionalidades, mas você precisa de apenas um pouquinho delas.

Exemplo 1: Sistema de Home Theater

Imagine que você tem um sistema de home theater em casa. Para assistir a um filme, você precisa realizar várias operações: ligar a TV, ligar o sistema de som, ligar o leitor de Blu-ray,

ajustar o volume, escurecer as luzes, etc. Sem uma interface simplificada, isso exigiria interagir com vários controles remotos ou interfaces.

Façade neste contexto seria um controle remoto universal ou um sistema automatizado que, ao pressionar um único botão "Assistir Filme", executa todas essas operações complexas em segundo plano. O usuário interage apenas com uma interface simples, enquanto a lógica complexa de controle de todos os dispositivos é ocultada por trás do Façade.

Exemplo 2: Serviço de Reserva de Viagens

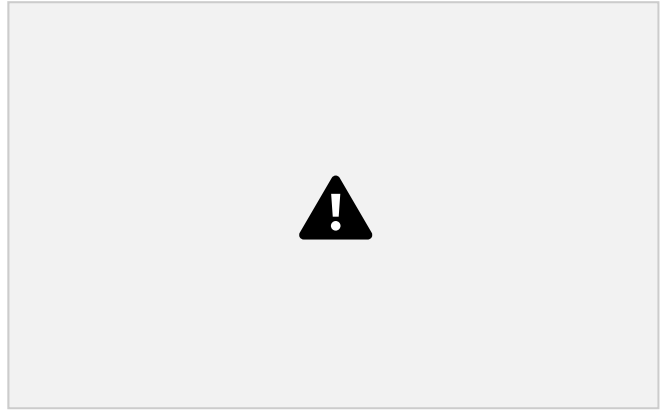
Quando você faz uma reserva de viagem através de uma agência de viagens online, você geralmente interage com uma interface simples, onde pode selecionar datas, destinos, hotéis, e voos.

Por trás dessa interface simples, o sistema pode estar comunicando-se com múltiplos serviços externos: um para verificar a disponibilidade de voos, outro para hotéis, outro para aluguel de carros, etc.

Façade aqui é a interface da agência de viagens online que fornece uma experiência unificada e simples ao usuário, enquanto todos os detalhes complexos da comunicação com múltiplos serviços e fornecedores são gerenciados em segundo plano. O usuário só vê a interface simplificada e não precisa se preocupar com a complexidade do sistema por trás dela.

Propósito

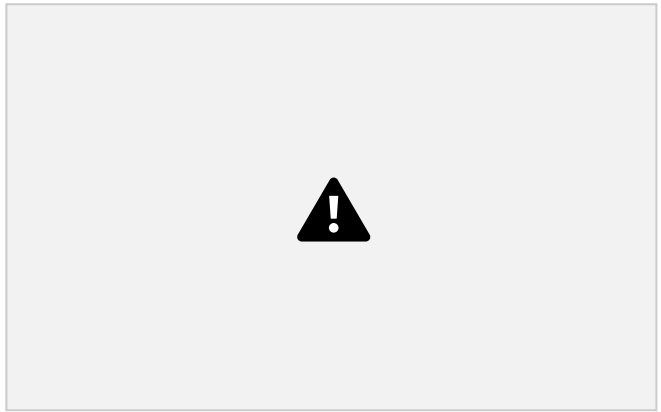
O Flyweight é um padrão de projeto estrutural que permite a você colocar mais objetos na quantidade de RAM disponível ao compartilhar partes comuns de estado entre os múltiplos objetos ao invés de manter todos os dados em cada objeto.



Problema

Você enfrenta um problema ao criar um jogo que tem gŕaficos ultra realistas.

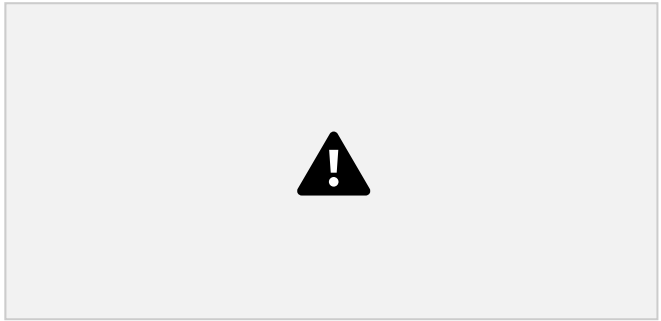
Após testar o jogo na sua máquina, descobre que, em computadores menos poderosos, o jogo quebra devido à quantidade insuficiente de RAM, causada pelo armazenamento excessivo de informações que a qualidade do jogo exige.



Solução

Utilizar o padrão Flyweight para otimizar o uso de memória.
Campos como cor e sprite consomem muita memória e são quase idênticos para todas as partes do jogo.

O padrão Flyweight sugere separar o estado intrínseco (constante para cada partícula) do estado extrínseco (variável).
Isso resulta em objetos Flyweight que armazenam o estado intrínseco e são reutilizados em diferentes contextos.



O estado extrínseco é movido para o objeto contêiner, neste caso, a classe Jogo. O contexto, que inclui o estado extrínseco e a referência ao Flyweight, é armazenado em uma classe separada. Isso reduz significativamente a quantidade de objetos contextuais, já que eles compartilham um único objeto Flyweight, economizando memória.

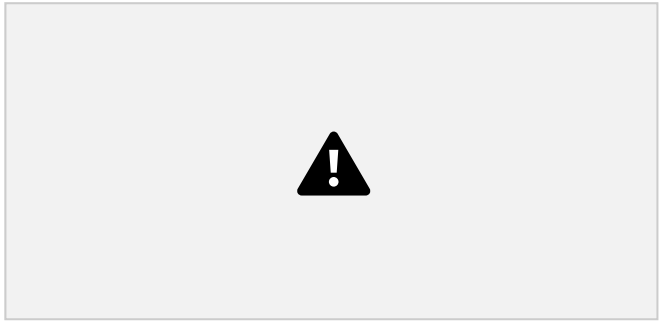
Solução

Imutabilidade e Fábrica Flyweight:

É importante garantir a imutabilidade nos objetos Flyweight, para que seu estado não seja modificado.

Uma fábrica Flyweight é sugerida para gerenciar os objetos existentes, fornecendo acesso conveniente e reutilizando objetos Flyweight semelhantes.

O padrão Flyweight é apresentado como uma solução elegante para reduzir a carga de memória em situações em que objetos compartilham características semelhantes, resultando em eficiência na gestão de recursos.



Exemplo 1: Aplicativos de Processamento de Texto

Em aplicativos de processamento de texto, como o Microsoft Word, cada caractere que você digita pode ser representado como um objeto na memória. Se cada caractere fosse armazenado separadamente com todas as suas propriedades (fonte, estilo, tamanho, cor, etc.), o consumo de memória seria muito alto.

Flyweight neste caso seria a reutilização de objetos de caracteres. Em vez de criar um novo objeto para cada caractere, o aplicativo compartilha objetos para caracteres iguais. Por exemplo, todas as letras "A" que aparecem no documento com a mesma formatação (mesma fonte, tamanho, cor) podem ser representadas por um único objeto na memória, economizando recursos.

Exemplo 2: Jogos de Vídeo com Muitos Objetos Similares

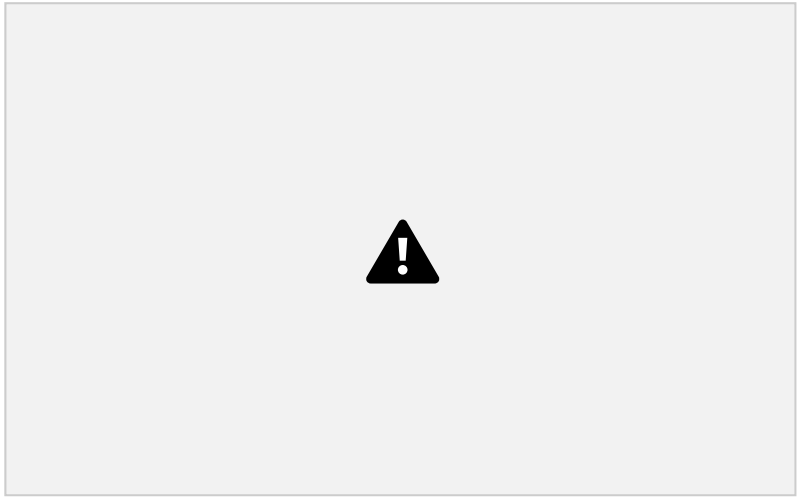
Em jogos de vídeo, especialmente aqueles que têm muitos elementos gráficos repetidos, como árvores em um cenário de floresta ou soldados em um exército, o padrão Flyweight pode ser usado para reduzir o consumo de memória.

Flyweight aqui seria a reutilização de objetos para representar elementos gráficos semelhantes. Por exemplo, em um jogo, todas as árvores que parecem iguais podem ser representadas por uma única instância de um objeto árvore, que é reutilizada em diferentes posições do cenário. As informações que variam, como a posição da árvore, são mantidas separadamente, mas a representação gráfica (como a textura e a forma) é compartilhada. Isso permite que o jogo gerencie grandes quantidades de elementos sem sobrecarregar a memória.

Propósito

O Proxy é um padrão de projeto estrutural que permite que você forneça um substituto ou um espaço reservado para outro objeto.

Um proxy controla o acesso ao objeto original, permitindo que você faça algo antes ou depois do pedido chegar ao objeto original.



Problema

Você tem um objeto grande que consome muitos recursos do sistema e precisa dele de tempos em tempos, mas não sempre.

Solução

O padrão Proxy sugere que você crie uma nova classe proxy com a mesma interface do objeto do serviço original.

Então você atualiza sua aplicação para que ela passe o objeto proxy para todos os clientes do objeto original.

Ao receber uma solicitação de um cliente, o proxy cria um objeto

do
serviço real e delega a ele todo o trabalho.

Se você precisa executar alguma coisa tanto antes como depois da lógica primária da classe, o proxy permite que você faça isso sem mudar aquela classe. Uma vez que o proxy implementa a mesma interface que a classe original, ele pode ser passado para qualquer cliente que espera um objeto do serviço real.

Exemplo 1: Banco e Cartão de Crédito

Imagine um banco oferecendo cartões de crédito aos clientes. Para cada transação realizada com o cartão, o banco deve verificar a autenticidade e a validade da transação. Se o banco precisasse realizar todas essas verificações diretamente sempre que um cliente usasse seu cartão, isso poderia ser lento e ineficiente.

Proxy neste contexto é um serviço de autorização de transações que atua como intermediário. Quando um cliente realiza uma compra, o proxy realiza as verificações iniciais para validar a transação (por exemplo, checar o saldo, verificar se o cartão está ativo) antes de encaminhar a solicitação para o sistema principal do banco para uma análise mais detalhada. O proxy melhora o desempenho e controla o acesso ao sistema principal, permitindo uma verificação preliminar mais rápida.

Exemplo 2: Sistema de Controle de Acesso a Documentos

Considere um sistema de gerenciamento de documentos em uma empresa onde apenas alguns funcionários têm permissão para acessar documentos confidenciais. O sistema pode utilizar um proxy para controlar o acesso a esses documentos.

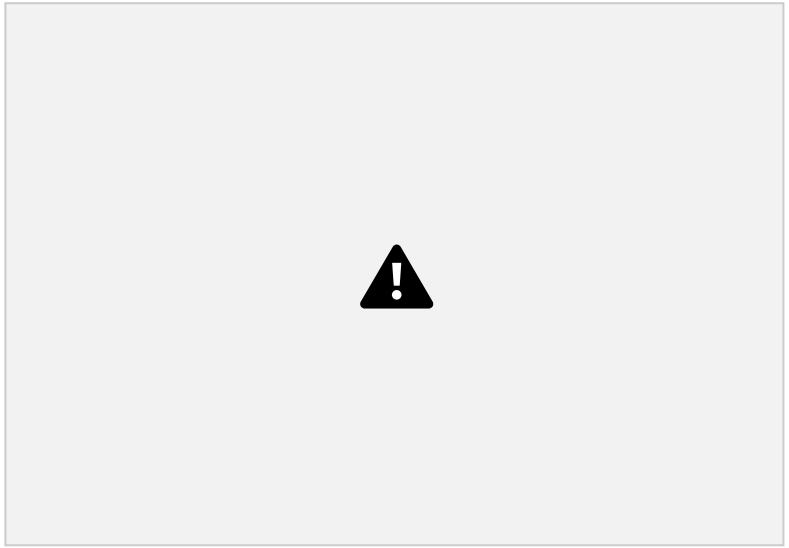
Proxy aqui é um mecanismo que verifica as permissões do usuário antes de permitir o acesso ao documento real. Quando um funcionário tenta abrir um documento confidencial, o proxy verifica se o funcionário tem as permissões necessárias. Se o funcionário tem permissão, o proxy fornece acesso ao documento real; caso contrário, ele nega o acesso. Esse proxy atua como um controle de acesso, garantindo que apenas os usuários autorizados possam interagir com documentos sensíveis.

PADRÕES COMPORTAMENTAIS

Chain of Responsability

Propósito

O Chain of Responsibility é um padrão de projeto comportamental que permite que você passe pedidos por uma corrente de handlers. Ao receber um pedido, cada handler decide se processa o pedido ou o passa adiante para o próximo handler na corrente.



Chain of Responsability

Problema

Ao desenvolver um sistema de encomendas online, a implementação de várias verificações sequenciais torna o código volumoso e desorganizado.

Adicionar novas funcionalidades resulta em código difícil de manter e reutilizar, levando a complexidades e duplicação.