Ministério da Educação
Secretaria de Educação Profissional e Tecnológica
Instituto Federal do Norte de Minas Gerais
Campus Montes Claros- Bacharel em Ciência da computação

Padrões de Projeto
Proxy e Facade

Graciely Duarte Dias

Trabalho apresentado como pré-requisito para aprovação na disciplina Programação Orientada a Objetos.

Montes Claros - 2023

1. Introdução

Hodiernamente não se conjectura um processo de desenvolvimento de software sério sem a utilização da orientação a objetos, pois esta permite agregar qualidades importantes aos sistemas desenvolvidos sob seus paradigmas, como a extensibilidade e a reusabilidade. No entanto, Gamma, Helm, Johnson e Vlissides (a Gang of Four , GoF) no livro Padrões de Projeto: soluções reutilizáveis de software orientado a objetos afirmam que projetar software orientado a objetos é difícil, mas projetar software reutilizável orientado a objetos é ainda mais complicado. Isso ocorre devido a necessidade de identificar objetos pertinentes, fatorá-los em classes no nível correto de granularidade, definir as interfaces das classes, as hierarquias de herança e estabelecer as relações-chave entre eles. O projeto precisa ser específico para o problema a resolver, mas também genérico o suficiente para atender problemas e requisitos futuros, também deseja evitar o reprojeto, ou pelo menos minimizá-lo.
Levando em consideração essas informações, projetistas experientes sabem que não devem resolver cada problema a partir de princípios elementares ou do zero. Em vez disso, eles utilizam soluções que funcionaram no passado. Desse modo, consequentemente, são encontrados padrões, de classes e de comunicação entre objetos, que aparecem frequentemente em muitos sistemas orientados a objetos. Esses padrões resolvem problemas específicos de projetos e tornam os projetos orientados a objetos mais flexíveis e, em última instância, reutilizáveis.Desta forma, eles auxiliam os projetistas a reutilizar projetos bem-sucedidos ao basear os novos projetos na experiência anterior, pois, uma vez que se conhece o padrão , uma grande quantidade de decisões de projeto decorre automaticamente.

2. Introdução aos padrões de projetos

2.1. Quando surgiu a ideia de padrão na computação?

A primeira aplicação da ideia de padrão de projetos na computação ocorreu em 1994 quando quatro autores, Erich Gamma, John Vlissides, Ralph Johnson, e Richard Helm, publicaram o livro “Padrões de Projeto — Soluções Reutilizáveis de Software Orientado a Objetos”. O livro mostrava 23 padrões que resolviam vários problemas de projeto orientado a objetos e se tornou um best-seller rapidamente. Devido a seu longo título, as pessoas começaram a chamá-lo simplesmente de “o livro da Gangue dos Quatro (Gang of Four)” que logo foi simplificado para o “livro GoF”. Desde então, dúzias de outros padrões orientados a objetos foram descobertos. A “abordagem por padrões” se tornou muito popular em outros campos de programação, então muitos desses padrões agora existem fora do projeto orientado a objetos também.

2.2. O que é padrões de projeto?

O primeiro a apresentar uma definição do que seria um padrão, foi o arquiteto e professor Christopher Alexandre, no seu livro “A Times Way of Building” (Oxford University Press, 1979), que é: “Cada padrão é uma regra de três partes, que expressa uma relação entre um certo contexto, um problema e uma solução”. Muito embora Alexander estivesse falando acerca de padrões, em construções e cidades, o que ele diz é verdadeiro em relação aos padrões de projeto orientados a objeto. Nossas soluções são expressas em termos de objetos e interfaces em vez de paredes e portas, mas no cerne de ambos os tipos de padrões está a solução para um problema num contexto. Em geral, um padrão tem quatro elementos essenciais:

- O nome do padrão de projeto é essencial para descrever de forma concisa e coesa um problema de projeto, suas soluções e consequências. Esses nomes ampliam imediatamente nosso vocabulário de projeto, permitindo-nos trabalhar em um nível mais abstrato. Com um vocabulário de padrões, podemos discuti-los com colegas, documentá-los e até mesmo nos comunicar melhor conosco mesmos. Os nomes facilitam a reflexão sobre os projetos, bem como a comunicação de seus custos, benefícios e outros aspectos para outras pessoas. No entanto, encontrar nomes adequados é uma das partes mais desafiadoras do desenvolvimento de um catálogo de padrões.
- O problema define o momento adequado para aplicar um padrão, descrevendo o contexto e o problema em si. Pode se referir a problemas específicos de projeto, como a representação de algoritmos como objetos, ou a estruturas de classes ou objetos que indicam inflexibilidade no projeto. Em algumas situações, o problema pode incluir uma lista de condições que devem ser atendidas para que a aplicação do padrão faça sentido.
- A solução descreve os componentes do projeto, seus relacionamentos, responsabilidades e colaborações. Não se trata de um projeto específico ou de uma implementação particular, pois um padrão é como um modelo que pode ser aplicado em várias situações diferentes. Em vez disso, o padrão fornece uma descrição abstrata de um problema de projeto e como um conjunto geral de elementos (classes e objetos, neste caso) resolve esse problema.
- As consequências referem-se aos resultados e análises das vantagens e desvantagens da aplicação do padrão. Embora muitas vezes não sejam mencionadas ao descrever decisões de projeto, são essenciais para avaliar alternativas de projeto e compreender os custos e benefícios da aplicação do padrão. No contexto do software, as consequências frequentemente envolvem compromissos de espaço e tempo, além de aspectos relacionados a linguagens e implementação. Como a reutilização é um fator importante no projeto orientado a objetos, as consequências de um padrão incluem seu impacto na flexibilidade, extensibilidade ou portabilidade de um sistema. Explorar explicitamente essas consequências ajuda a compreendê-las e avaliá-las.

Em linhas gerais, os padrões de projeto não são projetos tais como listas encadeadas e tabelas de acesso aleatório que podem ser codificadas em classes e ser reutilizadas tais como estão. Nem projetos complexos, de domínio específico, para uma aplicação inteira ou subsistema. Os padrões de projeto são descrições de objetos e classes comunicantes que são customizadas para resolver um problema geral de projeto num contexto particular. Um padrão de projeto nomeia, abstrai e identifica os aspectos chave de uma estrutura de projeto comum para torná-la útil para a criação de um projeto orientado a objetos reutilizáveis. O padrão de projeto identifica as classes e instâncias participantes, seus papéis, colaborações e a distribuição de responsabilidades. Cada padrão de projeto focaliza um problema ou tópico particular de projeto orientado a objetos. Ele descreve quando pode ser aplicado, se ele pode ser aplicado em função de outras restrições de projeto e as consequências, custos e benefícios de sua utilização.

2.3. Vantagens da utilização dos padrões de projeto

O mais importante sobre os padrões é que eles são soluções aprovadas. Cada catálogo inclui apenas padrões que foram considerados úteis por diversos desenvolvedores em vários projetos. Os padrões catalogados também são bem definidos; os autores (principalmente GoF) descrevem cada padrão com muito cuidado e em seu próprio contexto, portanto será fácil aplicar o padrão em suas próprias

circunstâncias. Eles também formam um vocabulário comum entre os desenvolvedores. Desse modo, os padrões de projeto oferecem as seguintes vantagens:

- Solução comprovada: Os padrões de projeto são baseados em experiências anteriores e melhores práticas da indústria. Eles são soluções testadas e comprovadas para problemas específicos, fornecendo um caminho confiável para a resolução de desafios de desenvolvimento.
- Reutilização de soluções: Ao utilizar padrões de projeto, você pode reutilizar soluções existentes em vez de começar do zero. Isso economiza tempo e esforço, pois você não precisa reinventar a roda a cada projeto.
- Comunicação eficaz: Os padrões de projeto estabelecem uma terminologia comum que permite uma comunicação eficaz entre membros da equipe. Ao falar sobre padrões de projeto, todos entendem os conceitos e as soluções propostas, facilitando a discussão e a colaboração.
- Melhoria da qualidade do código: Ao seguir os princípios e práticas dos padrões de projeto, você cria um código mais estruturado, modular e fácil de entender. Isso resulta em um software de melhor qualidade, com menor complexidade e maior facilidade de manutenção.
- Adaptabilidade e flexibilidade: Os padrões de projeto fornecem uma base sólida para o desenvolvimento de sistemas flexíveis e adaptáveis a mudanças. Eles permitem que você introduza novas funcionalidades ou faça alterações sem afetar o restante do sistema, pois os padrões estabelecem uma estrutura clara e modular.

3. Categorias dos padrões de projeto

Padrões de projeto diferem por sua complexidade, nível de detalhes, e escala de aplicabilidade ao sistema inteiro sendo desenvolvido. Os padrões mais básicos e de baixo nível são comumente chamados idiomáticos. Eles geralmente se aplicam apenas a uma única linguagem de programação. Os padrões mais universais e de alto nível são os padrões arquitetônicos; desenvolvedores podem implementar esses padrões em praticamente qualquer linguagem. Ao contrário de outros padrões, eles podem ser usados para fazer o projeto da arquitetura de toda uma aplicação.

Além disso, todos os padrões podem ser categorizados por seu propósito, ou intenção. Sendo os três grupos principais de padrões:

- Os padrões criacionais abstraem o processo de instanciação. Eles ajudam a tornar um sistema independente de como seus objetos são criados, compostos e representados. Um padrão de criação de classe usa a herança para variar a classe que é instanciada, enquanto que um padrão de criação de objeto delega a instância para outro objeto. Os padrões de criação se tornam importantes à medida que os sistemas evoluem no sentido de depender mais da composição de objetos do que da herança de classes. Quando isso acontece, a ênfase se desloca da codificação rígida de um conjunto fixo de comportamentos para a definição de um conjunto menor de comportamentos fundamentais, os quais podem ser compostos em qualquer número para definir comportamentos mais complexos. Assim, criar objetos com comportamentos particulares exige mais do que simplesmente instanciar uma classe. Há dois temas recorrentes nesses padrões. Primeiro, todos encapsulam conhecimento sobre quais classes concretas são usadas pelo sistema. Segundo, ocultam o modo como as instâncias destas classes são criadas e compostas. Tudo o que o sistema sabe no geral sobre os objetos é que suas classes são definidas por classes abstratas. Consequentemente, os padrões de criação dão muita flexibilidade ao que, como e quando é criado e a quem cria. Eles permitem configurar um sistema com "objetos-produtos" que variam amplamente em estrutura e funcionalidade. A configuração pode ser estática ou dinâmica.

- Os padrões estruturais se preocupam com a forma como classes e objetos são compostos para formar estruturas maiores. Os padrões estruturais de classes utilizam a herança para compor interfaces ou implementações. Dando um exemplo simples, considere como a herança múltipla mistura duas ou mais classes em uma outra. O resultado é uma classe que combina as propriedades das suas classes ancestrais. Esse padrão é particularmente útil para fazer bibliotecas de classes desenvolvidas independentemente trabalharem juntas. Em lugar de compor interfaces ou implementações, os padrões estruturais de objetos descrevem maneiras de compor objetos para obter novas funcionalidades, A flexibilidade obtida pela composição de objetos provém da capacidade de mudar a composição em tempo de execução, o que é impossível com a composição estática de classes.
- Os padrões comportamentais se preocupam com algoritmos e a atribuição de responsabilidades entre objetos. Os padrões comportamentais não descrevem apenas padrões de objetos ou classes, mas também os padrões de comunicação entre eles. Esses padrões caracterizam fluxos de controle para permitir que você se concentre somente na maneira como os objetos são interconectados. Os padrões comportamentais de classe utilizam a herança para distribuir o comportamento entre classes. Os padrões comportamentais de objetos utilizam a composição de objetos em vez da herança.

Os catálogo de padrões de projeto sendo:

Escopo	Criacionais	Estruturais	Comportamentais
Classe	Factory Method	Adapter (classe)	Interpreter
			Template Method
Objeto	Abstract Factory	Adapter (objeto)	Chain of Responsibility
	Builder	Bridge	Command
	Prototype	Composite	Iterator
	Singleton	Decorator	Mediator
		Facade	Memento
		Flyweight	Observer
		Proxy	State
			Strategy
			Visitor

3.1. Padrões Criacionais

- Factory Method: fornece uma interface para criar objetos em uma superclasse, mas permite que as subclasses alterem o tipo de objetos que serão criados.
- Abstract Factory: permite que você produza famílias de objetos relacionados sem ter que especificar suas classes concretas.
- Builder: 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.

- Prototype: permite copiar objetos existentes sem fazer seu código ficar dependente de suas classes.
- Singleton: permite a você garantir que uma classe tenha apenas uma instância, enquanto provê um ponto de acesso global para essa instância.

3.2. Padrões Estruturais

- Adapter: permite objetos com interfaces incompatíveis colaborarem entre si.
- Bridge: permite que você divida uma classe grande ou um conjunto de classes intimamente ligadas em duas hierarquias separadas—abstração e implementação—que podem ser desenvolvidas independentemente umas das outras.
- Composite: permite que você componha objetos em estruturas de árvores e então trabalhe com essas estruturas como se elas fossem objetos individuais.
- Decorator: permite que você acople novos comportamentos para objetos ao colocá-los dentro de invólucros de objetos que contém os comportamentos.
- Facade: fornece uma interface simplificada para uma biblioteca, um framework, ou qualquer conjunto complexo de classes.
- Flyweight: 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.
- Proxy: 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 ou antes ou depois do pedido chegar ao objeto original.

3.3. Padrões Comportamentais

- Interpreter: usado para definição de linguagens. Define representações para gramáticas e abstrações para análise sintática.
- Template Method: define o esqueleto de um algoritmo em uma operação. O Template Method permite que subclasses componham o algoritmo e tenham a possibilidade de redefinir certos passos a serem tomados no processo, sem contudo mudá-lo.
- Chain of Responsibility: encadeia os objetos receptores e transporta a mensagem através da corrente até que um dos objetos responda. Assim, separa objetos transmissores dos receptores, dando a chance de mais de um objeto poder tratar a mensagem.
- Command: encapsula uma mensagem como um objeto, de modo que se possa parametrizar clientes com diferentes mensagens. Separa, então, o criador da mensagem do executor da mesma.
- Iterator: provê um modo de acesso a elementos de um agregado de objetos, sequencialmente, sem exposição de estruturas internas.
- Mediator: desacopla e gerencia as colaborações entre um grupo de objetos. Define um objeto que encapsula as interações dentre desse grupo.
- Memento: captura e externaliza o estado interno de um objeto (captura um "snapshot"). O Memento não viola o encapsulamento.
- Observer: provê sincronização, coordenação e consistência entre objetos relacionados.
- State: deixa um objeto mudar seu comportamento quando seu estado interno muda, mudando, efetivamente, a classe do objeto.

- Strategy: define uma família de algoritmos, encapsula cada um deles, e torna a escolha de qual usar flexível. O Strategy desacopla os algoritmos dos clientes que os usa.
- Visitor: representa uma operação a ser realizada sobre elementos da estrutura de um objeto. O Visitor permite que se crie uma nova operação sem que se mude a classe dos elementos sobre os quais ela opera.

4. Padrão de projeto Proxy

O padrão de projeto Proxy também conhecido como surrogate fornece um substituto ou marcador da localização de outro objeto para controlar o acesso a esse objeto.

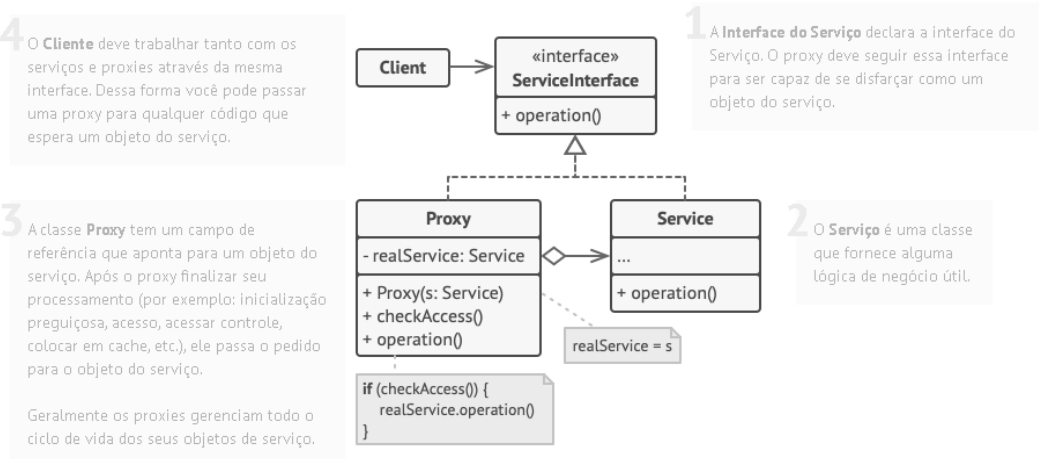
4.1. Objetivo

O objetivo do padrão de projeto Proxy é fornecer um substituto ou representante de um objeto real. Ele atua como uma camada intermediária entre o cliente e o objeto real, permitindo controlar o acesso a esse objeto e adicionar funcionalidades extras sem expor diretamente o objeto original.

4.2. Aplicabilidade

O padrão Proxy é aplicável sempre que há necessidade de uma referência mais versátil, ou sofisticada, do que um simples apontador para um objeto. Pois, o padrão Proxy funciona interceptando as solicitações do cliente e processando-as ou redirecionando-as para o objeto real, conforme necessário. Ele pode realizar ações adicionais, como realizar cache de resultados, controlar o acesso concorrente ou executar operações de segurança. O padrão Proxy é amplamente aplicado em situações em que é necessário adicionar lógica de controle de acesso, melhorar o desempenho ou lidar com tarefas auxiliares. Desse modo, o padrão Proxy resolve problemas relacionados ao controle de acesso, desempenho e tarefas auxiliares. Ele permite restringir o acesso direto ao objeto real, fornecer permissões específicas de acesso e adicionar funcionalidades adicionais, como o cache de resultados, tratamento de chamadas remotas ou operações de segurança.

4.3. Estrutura



4.4. Conceitos envolvidos

O padrão Proxy envolve conceitos como encapsulamento, interface e delegação. O Proxy encapsula a complexidade da interação com o objeto real, fornecendo uma interface simplificada para o cliente. Ele também pode delegar tarefas específicas para o objeto real, garantindo que sua funcionalidade principal seja preservada.

4.5. Consequências

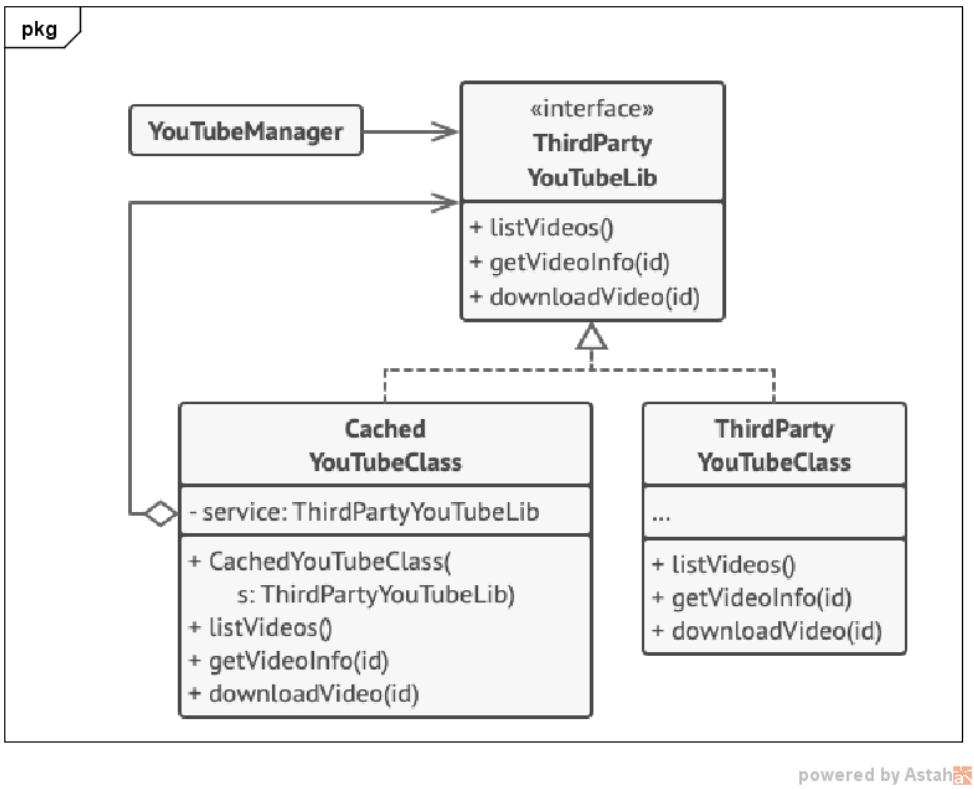
O padrão Proxy pode ocultar do cliente a otimização denominada de copy-on-write, que está relacionada à criação de um objeto sob demanda. Copiar um objeto grande e complicado pode ser uma operação cara do ponto de vista computacional. Se a cópia nunca é modificada, então não há a necessidade de incorrer neste custo. Pela utilização de um proxy para postergar o processo de cópia, garantimos que pagamos o preço da cópia do objeto somente se ele for modificado. Para copy-on-write funcionar, o objeto deve ter suas referências contadas. Copiar a proxy não fará nada mais do que incrementar esta contagem de referências. Somente quando o cliente solicita uma operação que modifica o objeto, o proxy realmente o copia. Nesse caso, o proxy também deve diminuir a contagem de referências do objeto. Quando a contagem de referências se torna zero, o objeto é deletado.A abordagem copy-on-write pode reduzir significativamente o custo computacional da cópia de objetos muito pesados.

4.6. Como Implementar

1. Se não há uma interface do serviço pré existente, crie uma para fazer os objetos proxy e serviço intercomunicáveis. Extrair a interface da classe serviço nem sempre é possível, porque você precisaria mudar todos os clientes do serviço para usar aquela interface. O plano B é fazer do proxy uma subclasse da classe serviço e, dessa forma, ele herdará a interface do serviço.
2. Crie a classe proxy. Ela deve ter um campo para armazenar uma referência ao serviço. Geralmente proxies criam e gerenciam todo o ciclo de vida de seus serviços. Em raras ocasiões, um serviço é passado ao proxy através do construtor pelo cliente.
3. Implemente os métodos proxy de acordo com o propósito deles. Na maioria dos casos, após realizar algum trabalho, o proxy deve delegar o trabalho para o objeto do serviço.
4. Considere introduzir um método de criação que decide se o cliente obtém um proxy ou serviço real. Isso pode ser um simples método estático na classe do proxy ou um método factory todo implementado.
5. Considere implementar uma inicialização preguiçosa para o objeto do serviço.

4.7. Pseudocódigo

Este exemplo ilustra como o padrão Proxy pode ajudar a introduzir uma inicialização preguiçosa e cache para um biblioteca de integração terceirizada do YouTube.



Colocando em cache os resultados de um serviço com um proxy.

A biblioteca fornece a nós com uma classe de download de vídeo. Contudo, ela é muito ineficiente. Se a aplicação cliente pedir o mesmo vídeo múltiplas vezes, a biblioteca apenas baixa de novo e de novo, ao invés de colocar ele em cache e reutilizar o primeiro arquivo de download.

A classe proxy implementa a mesma interface que a classe baixadora original e delega-a todo o trabalho. Contudo, ela mantém um registro dos arquivos baixados e retorna o resultado em cache quando a aplicação pede o mesmo vídeo múltiplas vezes.

```
// A interface de um serviço remoto.
interface ThirdPartyYouTubeLib is
    method listVideos()
    method getVideoInfo(id)
    method downloadVideo(id)

// A implementação concreta de um serviço conector. Métodos
// dessa classe podem pedir informações do YouTube. A
velocidade
// do pedido depende da conexão do usuário com a internet, bem
// como do YouTube. A aplicação irá ficar lenta se muitos
// pedidos forem feitos ao mesmo tempo, mesmo que todos peçam
a
// mesma informação.
```

```
class ThirdPartyYouTubeClass implements ThirdPartyYouTubeLib
is
    method listVideos() is
        // Envia um pedido API para o YouTube.

    method getVideoInfo(id) is
        // Obtém metadados sobre algum vídeo.

    method downloadVideo(id) is
        // Baixa um arquivo de vídeo do YouTube.

// Para salvar largura de banda, nós podemos colocar os
// resultados do pedido em cache e mantê-los por determinado
// tempo. Mas pode ser impossível colocar tal código
diretamente
// na classe de serviço. Por exemplo, ele pode ter sido
// fornecido como parte de uma biblioteca de terceiros e/ou
// definida como `final`. É por isso que nós colocamos o
código
// do cache em uma nova classe proxy que implementa a mesma
// interface que a classe de serviço. Ela delega ao objeto do
// serviço somente quando os pedidos reais foram enviados.
class CachedYouTubeClass implements ThirdPartyYouTubeLib is
    private field service: ThirdPartyYouTubeLib
    private field listCache, videoCache
    field needReset

    constructor CachedYouTubeClass(service:
ThirdPartyYouTubeLib) is
        this.service = service

    method listVideos() is
        if (listCache == null || needReset)
            listCache = service.listVideos()
        return listCache

    method getVideoInfo(id) is
        if (videoCache == null || needReset)
            videoCache = service.getVideoInfo(id)
        return videoCache

    method downloadVideo(id) is
```

```
        if (!downloadExists(id) || needReset)
            service.downloadVideo(id)

// A classe GUI, que é usada para trabalhar diretamente com um
// objeto de serviço, permanece imutável desde que trabalhe
com
// o objeto de serviço através de uma interface. Nós podemos
// passar um objeto proxy com segurança ao invés de um objeto
// real de serviço uma vez que ambos implementam a mesma
// interface.
class YouTubeManager is
    protected field service: ThirdPartyYouTubeLib

    constructor YouTubeManager(service: ThirdPartyYouTubeLib)
is
        this.service = service

    method renderVideoPage(id) is
        info = service.getVideoInfo(id)
        // Renderiza a página do vídeo.

    method renderListPanel() is
        list = service.listVideos()
        // Renderiza a lista de miniaturas do vídeo.

    method reactOnUserInput() is
        renderVideoPage()
        renderListPanel()

// A aplicação pode configurar proxies de forma fácil e
rápida.
class Application is
    method init() is
        aYouTubeService = new ThirdPartyYouTubeClass()
        aYouTubeProxy = new
CachedYouTubeClass(aYouTubeService)
        manager = new YouTubeManager(aYouTubeProxy)
        manager.reactOnUserInput()
```

4.8. Prós e Contras

✔️ Você pode controlar o objeto do serviço sem os clientes ficarem sabendo.
✔️ Você pode gerenciar o ciclo de vida de um objeto do serviço quando os clientes não se importam mais com ele.
✔️ O proxy trabalha até mesmo se o objeto do serviço ainda não está pronto ou disponível.
✔️ *Princípio aberto/fechado*. Você pode introduzir novos proxies sem mudar o serviço ou clientes.
❌O código pode ficar mais complicado uma vez que você precisa introduzir uma série de novas classes.
❌ A resposta de um serviço pode ter atrasos.

5. Padrão de projeto Facade

O padrão Facade fornece uma interface unificada para um conjunto de interfaces em um subsistema. Facade define uma interface de nível mais alto que torna o subsistema mais fácil de ser usado.

5.1. Objetivo

O objetivo do padrão Facade é fornecer uma interface unificada e simplificada para um subsistema complexo. Ele encapsula a complexidade do subsistema e fornece uma interface de nível mais alto que facilita a utilização e o entendimento por parte dos clientes.

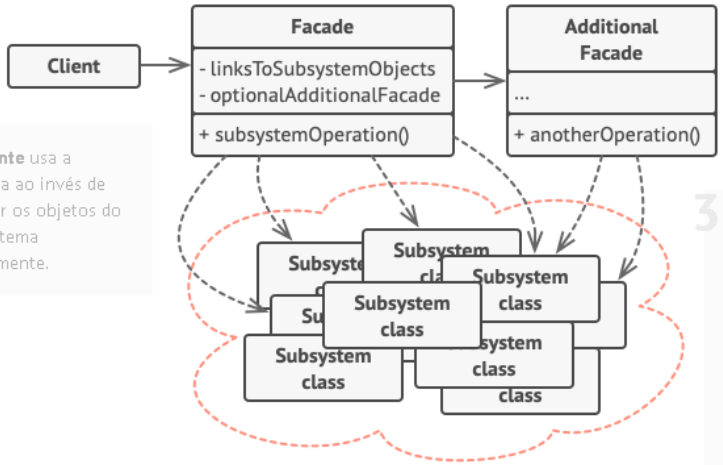
5.2. Aplicabilidade

O padrão Facade funciona recebendo solicitações dos clientes e realizando as operações necessárias no subsistema correspondente. Ele pode coordenar várias chamadas e interações internas com o subsistema, ocultando essa complexidade dos clientes. O Facade simplifica o uso do subsistema, fornecendo métodos específicos que encapsulam a lógica interna e apresentam uma interface mais amigável. O padrão Facade é frequentemente aplicado em situações em que há um subsistema complexo com múltiplas classes e interações intrincadas. Ele ajuda a melhorar a modularidade, o desacoplamento e a legibilidade do código, facilitando a manutenção e evolução do sistema como um todo. O padrão Facade resolve problemas relacionados à complexidade de interação com um subsistema. Ele simplifica a interface do subsistema, ocultando sua estrutura interna e fornecendo uma interface mais intuitiva e direta para os clientes. Isso facilita o uso do subsistema e reduz a dependência dos clientes em relação à sua implementação detalhada.

5.3. Estrutura

1 A **Fachada** fornece um acesso conveniente para uma parte particular da funcionalidade do subsistema. Ela sabe onde direcionar o pedido do cliente e como operar todas as partes móveis.

2 Uma classe **Fachada Adicional** pode ser criada para prevenir a poluição de uma única fachada com funcionalidades não relevantes que podem torná-lo mais uma estrutura complexa. Fachadas adicionais podem ser usadas tanto por clientes como por outras fachadas.



4 O **Cliente** usa a fachada ao invés de chamar os objetos do subsistema diretamente.

3 O **Subsistema Complexo** consiste em dúzias de objetos variados. Para tornar todos eles em algo que signifique alguma coisa, você tem que mergulhar fundo nos detalhes de implementação do subsistema, tais como objetos de inicialização na ordem correta e suprí-los com dados no formato correto.

As classes do subsistema não estão cientes da existência da fachada. Elas operam dentro do sistema e trabalham entre si diretamente.

5.4. Conceitos envolvidos

O padrão Facade envolve conceitos como encapsulamento, abstração e composição. O Facade atua como uma fachada para o subsistema, isolando os clientes da complexidade interna e fornecendo uma interface única e simplificada. Ele usa a composição para agrupar as diferentes partes do subsistema e expõe métodos mais simples e de alto nível para sua utilização.

5.5. Consequências

O padrão Facade oferece os seguintes benefícios:

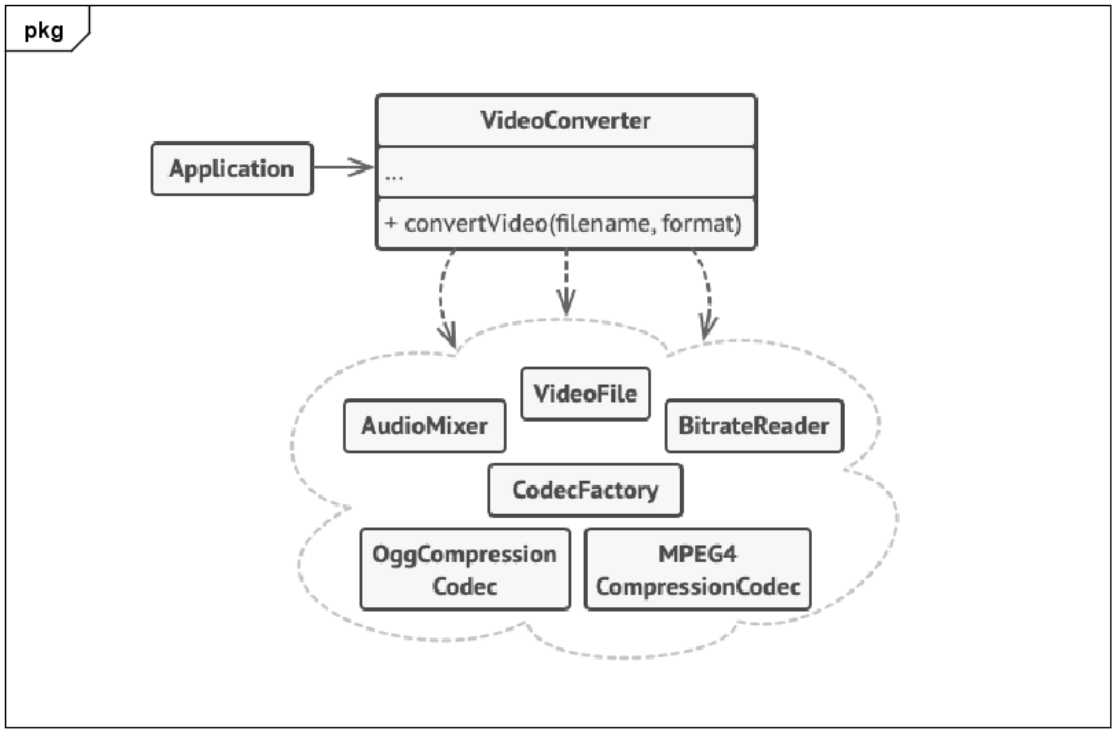
- Isola os clientes dos componentes do subsistema, dessa forma reduzindo o número de objetos com que os clientes têm que lidar e tornando o subsistema mais fácil de usar;
- Promove um acoplamento fraco entre o subsistema e seus clientes. Frequentemente, os componentes num subsistema são fortemente acoplados. O acoplamento fraco permite variar os componentes do subsistema sem afetar os seus clientes. As Facades ajudam a estratificar um sistema e as dependências entre objetos. Elas podem eliminar dependências complexas ou circulares. Isso pode ser uma consequência importante quando o cliente e o subsistema são implementados independentemente.
- Não impede as aplicações de utilizarem as classes do subsistema caso necessite fazê-lo. Assim, pode-se escolher entre a facilidade de uso e a generalização.

5.6. Como Implementar

1. Verifique se é possível providenciar uma interface mais simples que a que o subsistema já fornece. Você está no caminho certo se essa interface faz o código cliente independente de muitas classes do subsistema.
2. Declare e implemente essa interface em uma nova classe fachada. A fachada deve redirecionar as chamadas do código cliente para os objetos apropriados do subsistema. A fachada deve ser responsável por inicializar o subsistema e gerenciar seu ciclo de vida a menos que o código cliente já faça isso.
3. Para obter o benefício pleno do padrão, faça todo o código cliente se comunicar com o subsistema apenas através da fachada. Agora o código cliente fica protegido de qualquer mudança no código do subsistema. Por exemplo, quando um subsistema recebe um upgrade para uma nova versão, você só precisa modificar o código na fachada.
4. Se a fachada ficar **grande demais**, considere extrair parte de seu comportamento para uma nova e refinada classe fachada.

5.7. Pseudocódigo

Neste exemplo, o padrão Facade simplifica a interação com um framework complexo de conversão de vídeo.



Um exemplo de isolamento de múltiplas dependências dentro de uma única classe facade.

Ao invés de fazer seu código funcionar com dúzias de classes framework diretamente, você cria a classe fachada que encapsula aquela funcionalidade e a esconde do resto do código. Essa estrutura

também ajuda você a minimizar o esforço usando para atualizar para futuras versões do framework ou substituí-lo por outro. A única coisa que você precisaria mudar em sua aplicação seria a implementação dos métodos da fachada.

```
// Essas são algumas das classes de um framework complexo de
um
// conversor de vídeo de terceiros. Nós não controlamos aquele
// código, portanto não podemos simplificá-lo.

class VideoFile
// ...

class OggCompressionCodec
// ...

class MPEG4CompressionCodec
// ...

class CodecFactory
// ...

class BitrateReader
// ...

class AudioMixer
// ...


// Nós criamos uma classe fachada para esconder a complexidade
// do framework atrás de uma interface simples. É uma troca
// entre funcionalidade e simplicidade.
class VideoConverter is
    method convert(filename, format):File is
        file = new VideoFile(filename)
        sourceCodec = (new CodecFactory).extract(file)
        if (format == "mp4")
            destinationCodec = new MPEG4CompressionCodec()
        else
            destinationCodec = new OggCompressionCodec()
        buffer = BitrateReader.read(filename, sourceCodec)
        result = BitrateReader.convert(buffer,
destinationCodec)
```

```
        result = (new AudioMixer()).fix(result)
        return new File(result)

// As classes da aplicação não dependem de um bilhão de
classes
// fornecidas por um framework complexo. Também, se você
decidir
// trocar de frameworks, você só precisa reescrever a classe
// fachada.
class Application is
    method main() is
        convertor = new VideoConverter()
        mp4 = convertor.convert("funny-cats-video.ogg", "mp4")
        mp4.save()
```

5.8. Prós e Contras

✔️ Você pode isolar seu código da complexidade de um subsistema.
❌ Uma fachada pode se tornar **um objeto deus** acoplado a todas as classes de uma aplicação.

6. Conclusão

O uso de padrões de projeto proporciona flexibilidade na construção de aplicações e estruturas de código, além de documentar soluções reutilizáveis. Identificar os pontos comuns entre soluções diferentes para um mesmo problema permite o desenvolvimento de soluções mais eficientes, impulsionando o avanço do conhecimento. Os padrões facilitam a transferência de conhecimento entre projetistas experientes e novatos, por meio de uma linguagem clara e concisa, simplificando o desenvolvimento e o reaproveitamento de código.
Foram explorados os padrões de projeto Proxy e Facade. Esses padrões são amplamente utilizados na área de desenvolvimento de software para resolver problemas comuns e melhorar a estrutura e a modularidade dos sistemas. Esses padrões de projeto fornecem abstrações úteis e estratégias eficazes para lidar com problemas recorrentes no desenvolvimento de software. Eles promovem a modularidade, a reusabilidade e a flexibilidade do código, tornando os sistemas mais robustos e fáceis de evoluir.

7. Referências

7.1.Livros

- GAMMA, Erich. **Padrões de projetos: soluções reutilizáveis**. Bookman editora, Reimpressão 2008.
- SHVETS, Alexander. **Mergulho nos Padrões de Projeto.** Refactoring.Guru, 2021.

7.2. Links

- Conheça os Padrões de Projeto
- DESIGN PATTERNS - PADRÕES DE PROJETO
- Padrões de Projeto / Design patterns
- Proxy
- Facade