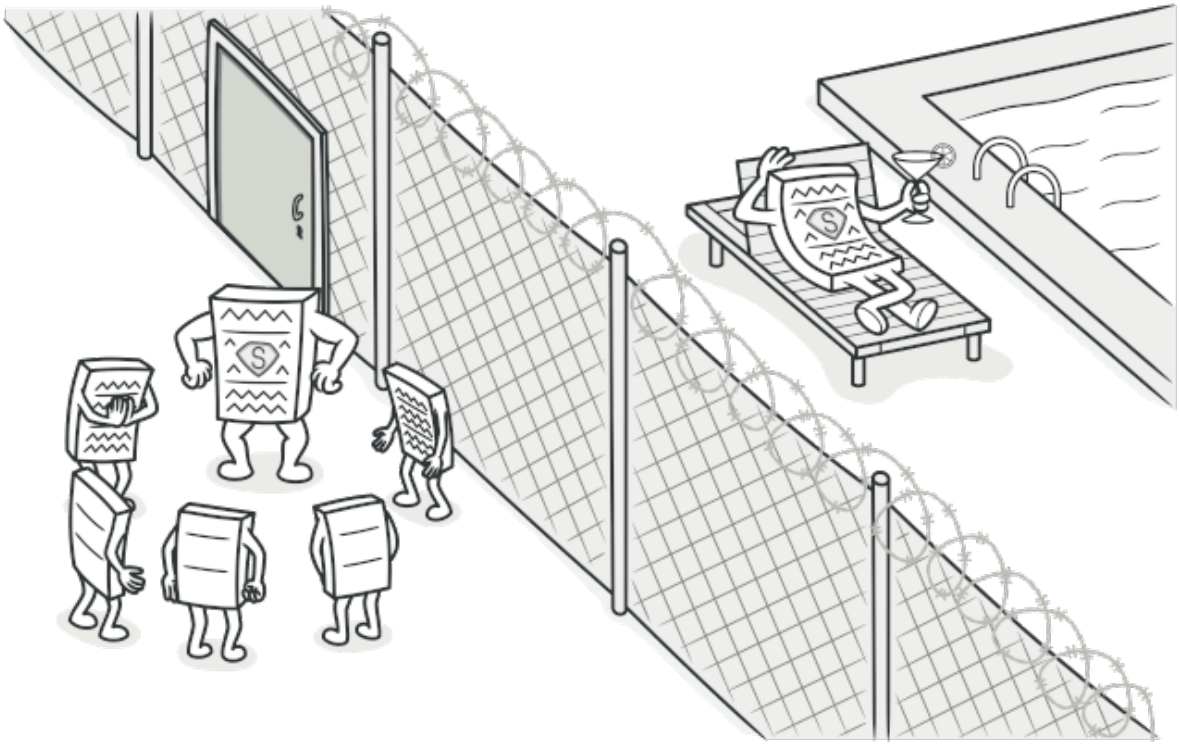
refactoring.guru

Proxy

12–16 minutes

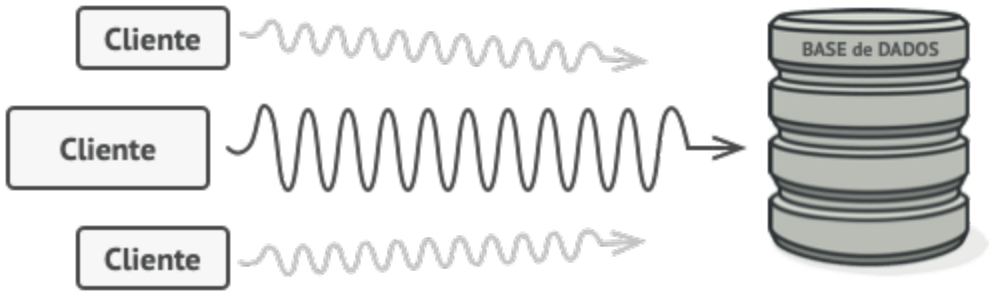
Propósito

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 ou antes ou depois do pedido chegar ao objeto original.



Problema

Por que eu iria querer controlar o acesso a um objeto? Aqui está um exemplo: você tem um objeto grande que consome muitos recursos do sistema. Você precisa dele de tempos em tempos, mas não sempre.



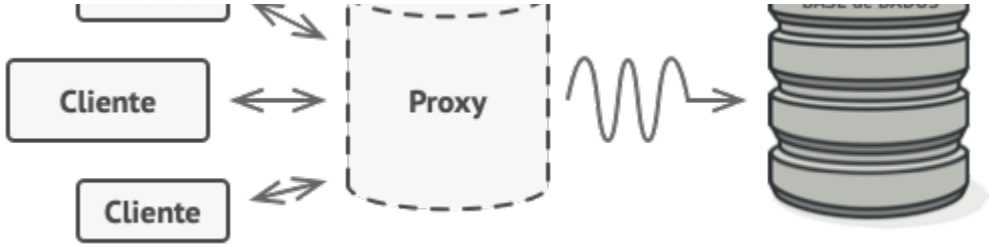
Solicitações para bases de dados podem ser bem lentas.

Você poderia implementar uma inicialização preguiçosa: criar esse objeto apenas quando ele é realmente necessário. Todos os clientes do objeto teriam que executar algum código adiado de inicialização. Infelizmente, isso provavelmente resultaria em muito código duplicado.

Em um mundo ideal, gostaríamos que você colocasse esse código diretamente dentro da classe do nosso objeto, mas isso nem sempre é possível. Por exemplo, a classe pode fazer parte de uma biblioteca fechada de terceiros.

Solução

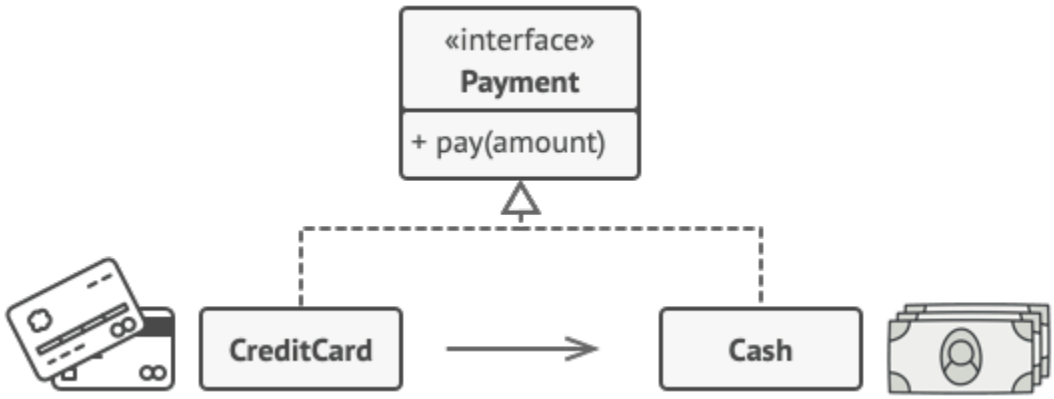
O padrão Proxy sugere que você crie uma nova classe proxy com a mesma interface do objeto do serviço original. Então você atualiza sua aplicação para que ela passe o objeto proxy para todos os clientes do objeto original. Ao receber uma solicitação de um cliente, o proxy cria um objeto do serviço real e delega a ele todo o trabalho.



O proxy se disfarça de objeto de base de dados. Ele pode lidar com inicializações preguiçosas e caches de resultados sem que o cliente ou a base de dados fiquem sabendo.

Mas qual é o benefício? 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.

Analogia com o mundo real



Cartões de crédito podem ser usados como pagamentos assim como o dinheiro.

Um cartão de crédito é um proxy para uma conta bancária, que é um proxy para uma porção de dinheiro. Ambos implementam a mesma interface porque não há necessidade de carregar uma porção de dinheiro por aí. Um cliente se sente bem porque não

precisa ficar carregando montanhas de dinheiro por aí. Um dono de loja também fica feliz uma vez que a renda da transação é adicionada eletronicamente para sua conta sem o risco de perdê-la no depósito ou de ser roubado quando estiver indo ao banco.

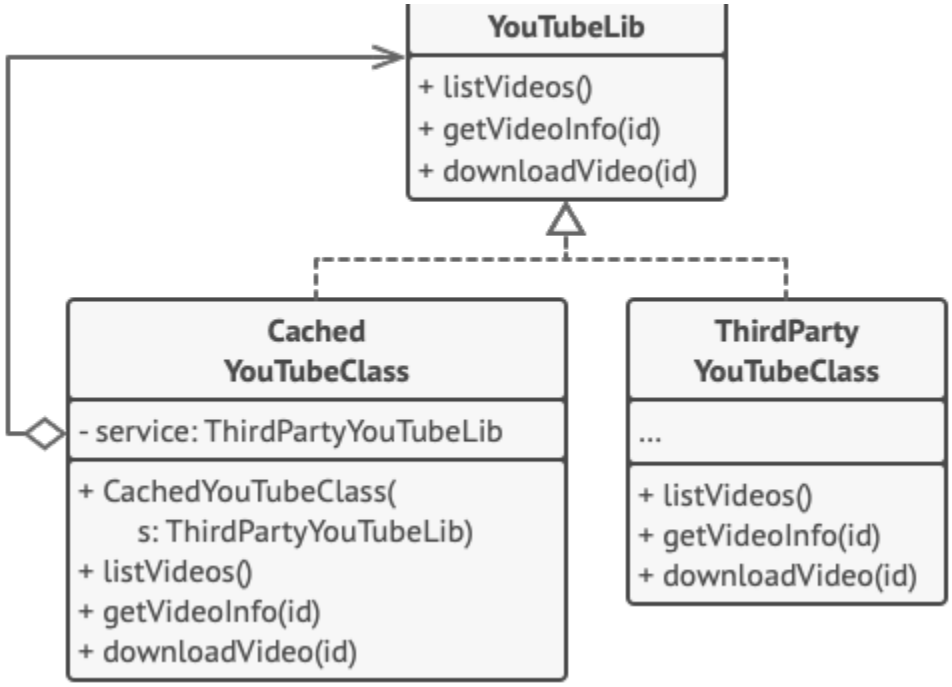
Estrutura

1. A **Interface do Serviço** declara a interface do Serviço. O proxy deve seguir essa interface para ser capaz de se disfarçar como um objeto do serviço.
2. O **Serviço** é uma classe que fornece alguma lógica de negócio útil.
3. A classe **Proxy** tem um campo de referência que aponta para um objeto do serviço. Após o proxy finalizar seu processamento (por exemplo: inicialização preguiçosa, acesso, acessar controle, colocar em cache, etc.), ele passa o pedido para o objeto do serviço.

   Geralmente os proxies gerenciam todo o ciclo de vida dos seus objetos de serviço.
4. O **Cliente** deve trabalhar tanto com os serviços e proxies através da mesma interface. Dessa forma você pode passar uma proxy para qualquer código que espera um objeto do serviço.

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()
```

Aplicabilidade

Há dúzias de maneiras de utilizar o padrão Proxy. Vamos ver os usos mais populares.

Inicialização preguiçosa (proxy virtual). Este é quando você tem

um objeto do serviço peso-pesado que gasta recursos do sistema por estar sempre rodando, mesmo quando você precisa dele de tempos em tempos.

Ao invés de criar um objeto quando a aplicação inicializa, você pode atrasar a inicialização do objeto para um momento que ele é realmente necessário.

Controle de acesso (proxy de proteção). Este é quando você quer que apenas clientes específicos usem o objeto do serviço; por exemplo, quando seus objetos são partes cruciais de um sistema operacional e os clientes são várias aplicações iniciadas (incluindo algumas maliciosas).

O proxy pode passar o pedido para o objeto de serviço somente se as credenciais do cliente coincidem com certos critérios.

Execução local de um serviço remoto (proxy remoto). Este é quando o objeto do serviço está localizado em um servidor remoto.

Neste caso, o proxy passa o pedido do cliente pela rede, lidando com todos os detalhes sujos pertinentes a se trabalhar com a rede.

Registros de pedidos (proxy de registro). Este é quando você quer manter um histórico de pedidos ao objeto do serviço.

O proxy pode fazer o registro de cada pedido antes de passar ao serviço.

Cache de resultados de pedidos (proxy de cache). Este é quando você precisa colocar em cache os resultados de pedidos do cliente e gerenciar o ciclo de vida deste cache, especialmente se os resultados são muito grandes.

O proxy pode implementar o armazenamento em cache para pedidos recorrentes que sempre acabam nos mesmos resultados. O proxy pode usar como parâmetros dos pedidos as chaves de cache.

Referência inteligente. Este é para quando você precisa ser capaz de se livrar de um objeto peso-pesado assim que não há mais clientes que o usam.

O proxy pode manter um registro de clientes que obtiveram uma referência ao objeto serviço ou seus resultados. De tempos em tempos, o proxy pode verificar com os clientes se eles ainda estão ativos. Se a lista cliente ficar vazia, o proxy pode remover o objeto serviço e liberar os recursos de sistema que ficaram empatados.

O proxy pode também fiscalizar se o cliente modificou o objeto do serviço. Então os objetos sem mudança podem ser reutilizados por outros clientes.

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.

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.

Relações com outros padrões

- Com Adapter, você acessa um objeto existente por meio de uma interface diferente. Com Proxy, a interface permanece a mesma. Com Decorator, você acessa o objeto por meio de uma interface

aprimorada.

- O Facade é parecido como o Proxy no quesito que ambos colocam em buffer uma entidade complexa e inicializam ela sozinhos. Ao contrário do *Facade*, o *Proxy* tem a mesma interface que seu objeto de serviço, o que os torna intermutáveis.
- O Decorator e o Proxy têm estruturas semelhantes, mas propósitos muito diferentes. Alguns padrões são construídos no princípio de composição, onde um objeto deve delegar parte do trabalho para outro. A diferença é que o *Proxy* geralmente gerencia o ciclo de vida de seu objeto serviço por conta própria, enquanto que a composição do *decoradores* é sempre controlada pelo cliente.