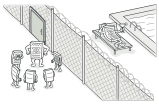
Hey, Dave! Just released the great Rust's ownership, safety and pure-loop programming skills, the third part of RUST! Check it out!

Proxy

Intent

Proxy is a structural design pattern that lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.



Problem

Why would you want to control access to an object? Here is an example: you have a massive object that consumes a vast amount of system resources. You need it from time to time, but not always.

Database queries can be really slow.

You could implement lazy initialization: create this object only when it's actually needed. All of the object's clients would need to execute some deferred initialization code. Unfortunately, this would probably cause a lot of code duplication.

In an ideal world, we'd want to put this code directly into our object's class, but that isn't always possible. For instance, the class may be part of a closed-format library.

Solution

The Proxy pattern suggests that you create a new proxy class with the same interface as an original service object. Then you update your app so that it passes the proxy object to all of the original object's clients. Upon receiving a request from a client, the proxy creates a real service object and delegates all the work to it.

The proxy disguises itself as a database object, or can cache lazy initialization and result caching without the client or the real database object even knowing.

But what's the benefit? If you need to execute something either before or after the primary logic of the class, the proxy lets you do this without changing that class. Since the proxy implements the same interface as the original class, it can be passed to any client that expects a real service object.

Real-World Analogy

Credit cards can be used for payments just like cash, and vice versa.

A credit card is a proxy for a bank account, which is a proxy for a bundle of cash. Both implement the same interface: they can be used for making a payment. A consumer feels great because there's no need to carry loads of cash around. A shop owner is also happy since the income from a transaction gets added electronically to the shop's bank account without the risk of losing the deposit or getting robbed on the way to the bank.

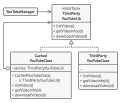
Structure

1. The **Service Interface** declares the interface of the Service. The proxy must follow this interface to be able to disguise itself as a service object.
2. The **Service** is a class that provides some useful business logic.
3. The **Proxy** class has a reference field that points to a service object. After the proxy finishes its processing (e.g., lazy initialization, logging, access control, caching, etc.), it passes the request to the service object.

   Usually, proxies manage the full lifecycle of their service objects.
4. The **Client** should work with both services and proxies via the same interface. This way you can pass a proxy into any code that expects a service object.

Pseudocode

This example illustrates how the **Proxy** pattern can help to introduce lazy initialization and caching in a 3rd party YouTube integration library.



Caching results of a service with a proxy.

The library provides us with the video downloading class. However, it's very inefficient. If the client application requests the same video multiple times, the library just downloads it over and over, instead of caching and reusing the files downloaded file.

The proxy class implements the same interface as the original downloader and delegates it all the work. However, it keeps track of the downloaded files and returns the cached result when the app requests the same video multiple times.

```
// The interface of a remote service.
interface ThirdPartyYouTubeLib {
    method listVideos();
    method getVideoInfo(id);
    method downloadVideo(id);

// The concrete implementation of a service connector. Methods
// of this class can request information from YouTube over the Internet
// or take requests as a user's browser connection as
// well as trackers. The application will view some of a lot of
// different data, but in the same time, some of them will request
// the same information.
class ThirdPartyYouTubeClass implements ThirdPartyYouTubeLib {
    method listVideos() {
        // Send an API request to YouTube.
    }

    method getVideoInfo(id) {
        // Get metadata about some video.
    }

    method downloadVideo(id) {
        // Download a video file from YouTube.
    }

// To save some bandwidth, we can cache request results and keep
// track for some time. We'll may be implemented in our code some
// directly into the service class, but example, to avoid keep
// been included as part of a third-party library, another method
// is to create a new class and to put all caching stuff into it.
// Proxy class which implements the same interface as the
// service class. It delegates to the service object only when
// the real requests have to be sent.
class CachedYouTubeClass implements ThirdPartyYouTubeLib {
    private field service: ThirdPartyYouTubeLib
    private field listCache, videoCache
    field cachePeriod

    constructor CachedYouTubeClass(service: ThirdPartyYouTubeLib, 
        this.service = service

    method listVideos() {
        if (listCache == null || expired)
            listCache = service.listVideos()
        return listCache
    }

    method getVideoInfo(id) {
        if (videoCache == null || expired)
            videoCache[id] = service.getVideoInfo(id)
        return videoCache[id]
    }

    method downloadVideo(id) {
        if (!localCache.exists(id) || expired)
            service.downloadVideo(id)
    }

// The GUI class, which used to work directly with a service
// object, stays unchanged as long as it works with the service
// object through an interface. We can safely pass a proxy
// instead of the real service object since they both
// implement the same interface.
class YouTubeManager {
    protected field service: ThirdPartyYouTubeLib

    constructor YouTubeManager(service: ThirdPartyYouTubeLib) {
        this.service = service
    }

    method renderVideoPage(id) {
        info = service.getVideoInfo(id)
        // Render the video page.
    }

    method renderSmallPreview(id) {
        info = service.getVideoInfo(id)
        // Render the list of video thumbnails.
    }

    method launchHomePage() {
        service.listVideos()
        // Render the home page.
    }

// The application can configure proxies on the fly.
class Application {
    method init() {
        aThirdPartyYouTubeClass = new ThirdPartyYouTubeClass()
        aCachedProxy = new CachedYouTubeClass(aThirdPartyYouTubeClass
        manager = new YouTubeManager(aCachedProxy)
        manager.launchHomePage()
```

Applicability

There are dozens of ways to utilize the Proxy pattern. Let's go over the most popular uses.

Lazy initialization (virtual proxy). This is when you have a heavyweight service object that wastes system resources by being always up, even though you only need it from time to time.

Instead of creating the object when the app launches, you can delay the object's initialization to a time when it's really needed.

Access control (protection proxy). This is when you want only specific clients to be able to use the service object; for instance, when your objects are crucial parts of an operating system and clients are various (another) applications (including malicious ones).

The proxy can pass the request to the service object only if the client's credentials match some criteria.

Local execution of a remote service (remote proxy). This is when the service object is located on a remote server.

In this case, the proxy passes the client request over the network, handling all of the nasty details of working with the network.

Logging requests (logging proxy). This is when you want to keep a history of requests to the service object.

The proxy can log each request before passing it to the service.

Caching request results (caching proxy). This is when you need to cache results of client requests and manage the life cycle of this cache, especially if results are quite large.

The proxy can implement caching for recurring requests that always yield the same results. The proxy may use the parameters of requests as the cache keys.

Smart reference. This is when you need to be able to access a heavyweight object even there are no clients that use it.

The proxy can keep track of clients that obtained a reference to the service object or its results. From time to time, the proxy may go over the clients and check whether they are still active. If the client list gets empty, the proxy might dismiss the service object and free the underlying system resources.

The proxy can also track whether the client had modified the service object. Then the unchanged objects may be reused by other clients.

How to Implement

1. If there's no pre-existing service interface, create one to make proxy and service objects interchangeable. Extracting the interface from the service class isn't always possible, because you'd need to change all of the service's clients to use that interface. Plan B is to make the proxy a subclass of the service class, and this way it'll inherit the interface of the service.
2. Create the proxy class. It should have a field for storing a reference to the service. Usually, proxies create and manage the whole life cycle of their services. On rare occasions, a service is passed to the proxy via a constructor by the client.
3. Implement the proxy methods according to their purposes. In most cases, after doing some work, the proxy should delegate the rest to the service object.
4. Consider introducing a creation method that decides whether the client gets a proxy or a real service. This can be a simple static method in the proxy class or a full-blown factory method.
5. Consider implementing lazy initialization for the service object.

Pros and Cons

- ✔ You can control the service object without clients knowing about it.
- ✔ You can manage the lifecycle of the service object when clients don't care about it.
- ✔ The proxy works even if the service object isn't ready or is not available.
- ✔ *Open/Closed Principle*. You can introduce new proxies without changing the service or clients.
- ✖ The code may become more complicated since you need to introduce a lot of new classes.
- ✖ The response from the service might get delayed.

Relations with Other Patterns

- **Adapter** provides a different interface to the wrapped object, **Proxy** provides it with the same interface, and **Decorator** provides it with an enhanced interface.
- **Facade** is similar to **Proxy** in that both buffer a complex entity and initialize it on its own. Unlike *Facade*, *Proxy* has the same interface as its service object, which makes them interchangeable.
- **Decorator** and **Proxy** have similar structures, but very different intents. Both patterns are built on the composition principle, where one object is supposed to delegate some of the work to another. The difference is that a *Proxy* usually manages the life cycle of its service object on its own, whereas the composition of *Decorators* is always controlled by the client.

Code Examples

Support our free website and own the eBook!
- 22 design patterns and 9 principles explained in detail.
- 405 well-structured, easy-to-read, jargon-free pages
- 325 clear and helpful illustrations and diagrams.
- Lifetime updates. Current book covers Java 17.
- All devices supported: PDF/EPUB/MOBI/AZW3 formats.
Learn more

RETURN: ← Flyweight
READ NEXT: Behavioral Patterns →

© 2014-2023 Refactoring.Guru. All rights reserved.
Kreditkartenstraße 171, 77, Karlsruhe / Hardtberg, Ukraine, 12345
Email: support@refactoring.guru
Illustrations by Dmitry Zhart
Terms & Conditions
Privacy Policy
Content Usage Policy