Proxy pattern

While writing code, you may have been using something that is called a service object.

A service object is a part of code that works in response to requests from other objects.

Service objects aren't supposed to be used constantly.
Because it can create major issues for your application both in terms of performance and security.

So, you may find yourself having to check requests to this service object.
You will have to activate it when necessary, and sometimes make operations before requests reach it. For this purpose, you can use a specific design pattern called proxy.

Proxy pattern

Proxy is a structural design pattern that allows you to provide some service with an intermediate object that controls access to that service. This intermediate object is known as a proxy; the same as the pattern.

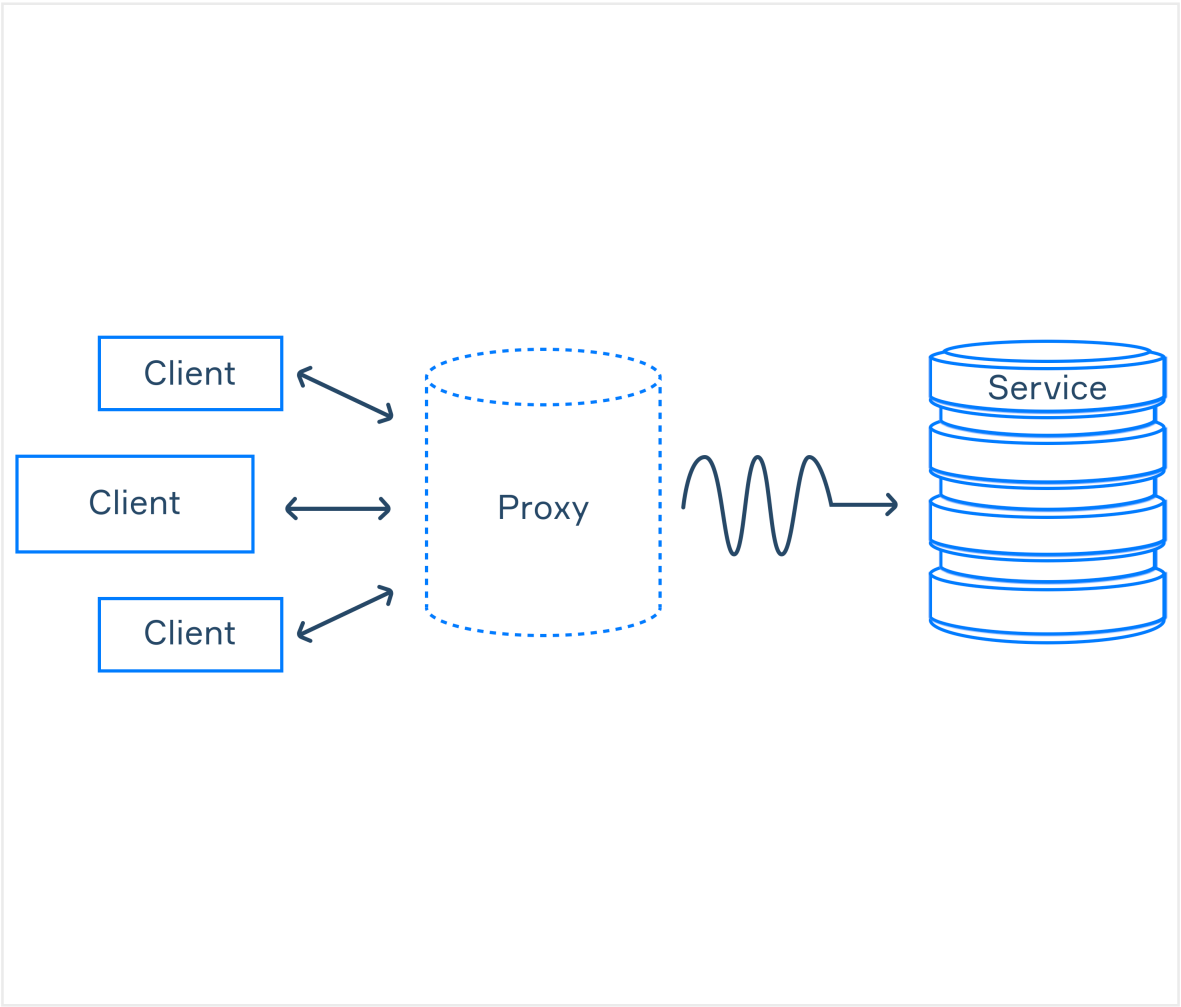
You can describe a proxy as a gate that performs some actions before a request from a client can access your service.

It is unlike the decorator pattern in which objects are provided with an enhanced interface or an adapter, where you change the interface altogether.

A proxy shares the same interface with the original object. So, other objects can access the proxy the same way.

The main point of proxy is that you can easily initialize service objects only when necessary. This is called **lazy initialization**.

Structure and types of proxy



Proxy is useful when you cannot alter the client so that it would include lazy initialization.

Instead, you include it with the proxy itself. This way you can save precious system resources that would otherwise be wasted by constantly maintaining the service.

Implementing a proxy can also help you avoid performance impact caused by wasted resources.

A proxy that provides you with lazy initialization is called a **virtual proxy**. But it isn't the only type of proxy there is.
There are a few popular proxy types that perform specific operations:

- **Protection proxy** – when you need to control which clients can access your service;
- **Cache proxy** – allows you to store results of service work to pass it on to future requests;
- **Logging proxy** – keeps the log of client requests;
- **Remote proxy** – passes object via a network connection to a remote service;
- **Smart proxy** – performs additional actions when accessing service.

As an example, counting the number of references to a service object, so that the service can be freed when there are no more references.

Example of proxy

Let's look at limited internet access inside an office.
In such cases, not every device should be given access to the local network. This needs to be checked by proxy. If a device is allowed, it will be passed to an internet access service.

Let's use an interface called InternetAccess and implement it in the service (RealInternetAccess) and the proxy (ProxyInternetAccess).

Our client object sends requests that will be processed through the proxy. In proxy, we will check client object clearance and if the object has it. Only then, real internet access is provided. Pretty simple, right? Now, let's try to depict this process in pseudocode.

Pseudocode for proxy

First things first, you need to define the InternetAccess interface where you will have methods for other classes.

```
interface InternetAccess is
  method grantAccess(id: Int)
```

Now, let's define internet access classes. Let's start with the service class:

```
class RealInternetAccess implements InternetAccess is
  constructor of RealInternetAccess() is
    ...

  method grantAccess(id: Int) is
    ...
```

In RealInternetAccess there is a method that does most of the work which is providing clients with access to the internet. Now, it's proxy time:

```
class ProxyInternetAccess implements InternetAccess is
  constructor of ProxyInternetAccess ()
    ...

  method getClearance(id: Int) is
    ...
    return clearance

  method grantAccess(id: Int) is
    Boolean clearance = getClearance(id)
    if (clearance == true) then
      InternetAccess service = new RealInternetAccess()
      service.grantAccess(id)
```

```
    else
      print("Internet access denied!")
```

In proxy, you check the clearance of a client program. If client clearance equals true, then you initialize your service class. Else, there will just be an error message.

Now that we are done with the proxy class, let's see how the client side is handled.

From the client side

In the client code, you will call the proxy class first. There you will check clearance and, if needed, initialize your service:

```
class Client is
  ...

  method accessInternet(id: Int) is
    InternetAccess service = new ProxyInternetAccess()
    service.grantAccess(id)
```

This approach implements lazy initialization in the code. So, if the InternetAccess class is really heavy on resources, you can initialize it only when needed.

But using proxy implementation can increase the response time from the service object. This is because the client needs to be given access by the proxy first. This change in response time might be undesirable in certain cases.

Conclusion

The proxy pattern is a great tool that you can use to maintain access to important service objects and perform operations with the requests before they reach those objects. It is useful when you cannot implement needed functions directly into your service. But, it is not a perfect solution to every problem either. Keep in mind that service response times could increase when using the proxy pattern.