Structural Design Pattern

Hung Tran

Fpt software

January 6, 2022

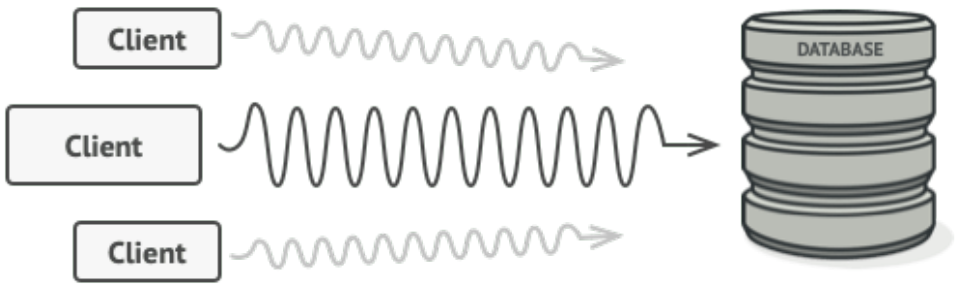
Outline

Structural Pattern Overview

How classes and objects are composed to form larger structure.

- **Adapter**: Convert the interface of a class into another interface.
- **Bridge**: Decouple an abstraction from its implementation.
- **Composite**: Compose objects into tree structure.
- **Decorator**: Attach additional responsibilities to an object dynamically.
- **Façade**: Provide a unified interface to a set of interfaces.
- **Flyweight**: Use sharing to support large numbers of fine-grained objects efficiently.
- **Proxy**: Provide a surrogate or placeholder for another object to control access to it.

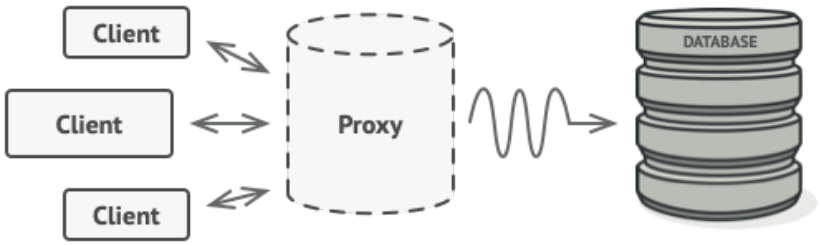
Problem Statement



- 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 3rd-party library.

- You have a massive object that consumes a vast amount of system resources. You need it from time to time, but not always.
- 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.

Solution: Facade Pattern

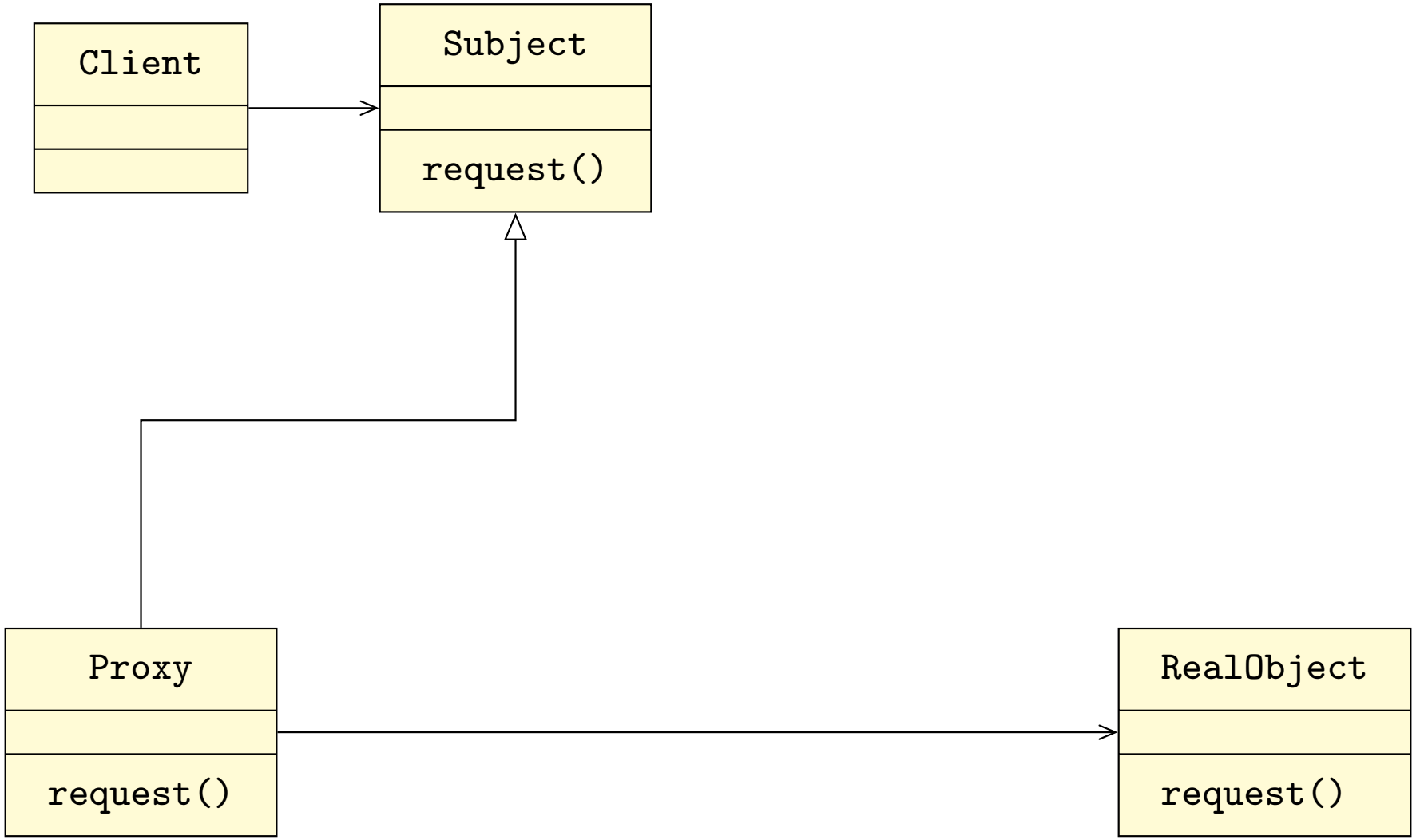


- A new proxy class with the same interface as an original service object.
- Then you update your app so that it passes the proxy object to all of the original object's clients.
- Upon receiving a request from a client, the proxy creates a real service object and delegates all the work to it.

The Intent of Proxy Design Pattern

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.

Structure of proxy Pattern



Structure of proxy Pattern

- 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.
- The Service is a class that provides some useful business logic.
- 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.
- 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.

Applicability

- 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.
- 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 launched applications (including malicious ones).
- Local execution of a remote service (remote proxy). This is when the service object is located on a remote server.
- Logging requests (logging proxy). This is when you want to keep a history of requests to the service object.

Applicability

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

How to Implement

- 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.

- 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.

- Implement the proxy methods according to their purposes. In most cases, after doing some work, the proxy should delegate the work to the service object.

- Consider introducing a creation method that decides whether the

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.

Thank You!