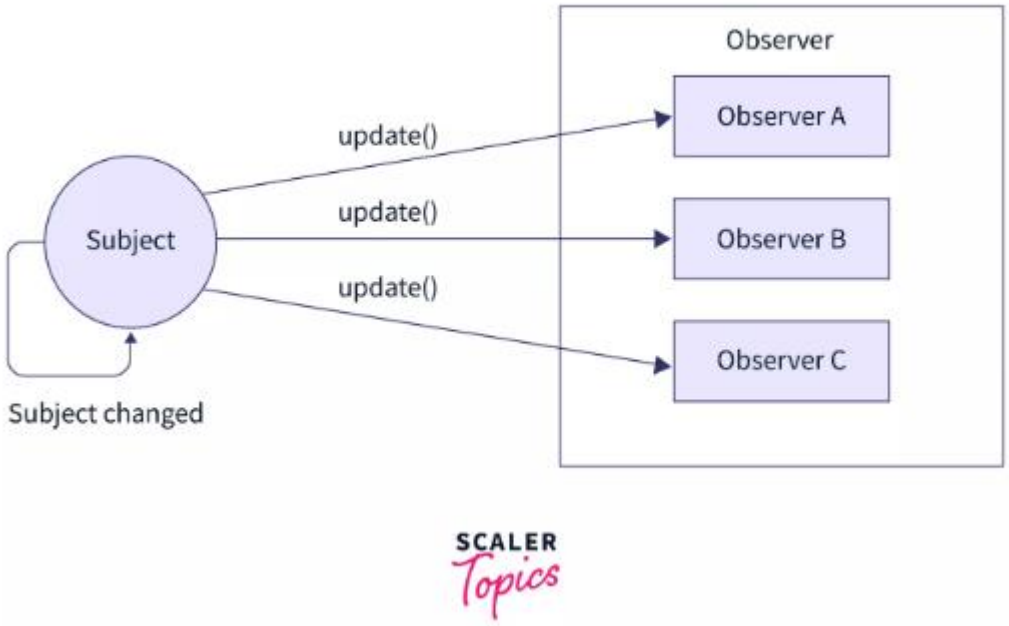
OBSERVER

Observer design pattern falls under the category of behavioral design patterns. The Observer Pattern maintains a one-to-many relationship among objects, ensuring that when the state of one object is changed, all of its dependent objects are simultaneously informed and updated. This design pattern is also referred to as Dependents.

A subject and observer (many) exist in a one-to-many relationship. Here the observers do not have any access to data, so they are dependent on the subject to feed them with information.

The observer design patterns when designing a system where several objects are interested in any possible modification to a specific object. In other words, the observer design pattern is employed when there is a one-to-many relationship between objects, such as when one object is updated, its dependent objects must be automatically notified.



SCALER Topics

Real-World Example: If a bus gets delayed, then all the passengers who were supposed to travel in it get notified about the delay, to minimize inconvenience. Here, the bus agency is the subject and all the passengers are the observers. All the passengers are dependent on the agency to provide them with information about

any changes regarding their travel journey. Hence, there is a one-to-many relationship between the travel agency and the passengers.

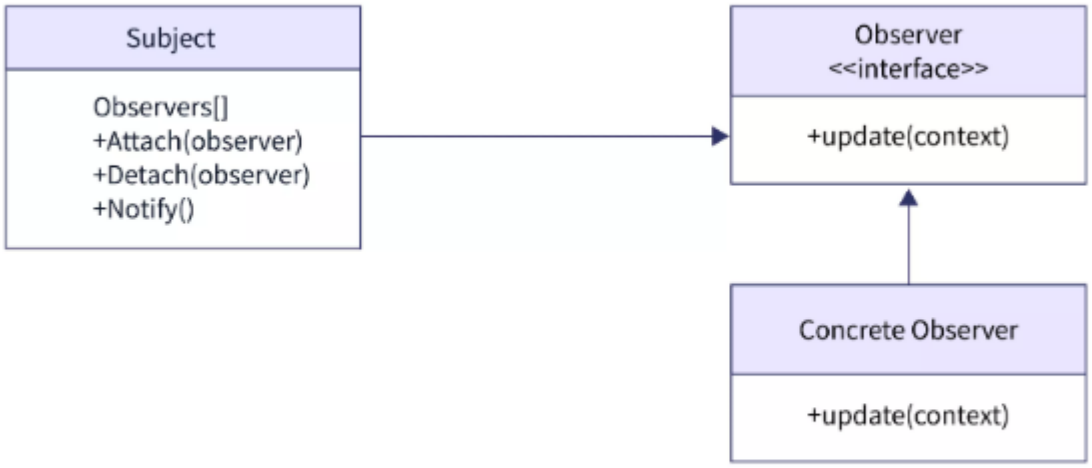
Other Examples of observer design patterns:

- **Social Media Platforms for example, many users can follow a particular person(subject) on a social media platform. All the followers will be updated if the subject updates his/her profile. The users can follow and unfollow the subject anytime they want.**
- **Newsletters or magazine subscription**
- **Journalists providing news to the media**
- **E-commerce websites notify customers of the availability of specific products**

Consider the following scenario: There is a new laughter club in town, with a grand opening, it caught the attention of a lot of people interested in being a member of the club. Thrilled with the overwhelming response, the club owner was a bit worried about the smooth management and involvement of all the members.

The observer design pattern is the best solution to the owner's problem. The owner here is the subject and all the members of the club are observers. The observers have no access to the club's information and the upcoming events unless the owner notifies them of it. Also, the members have the option to opt out of the club whenever they want to. This allows the owner to easily manage and engage all of the members.

How does Observer Design Patterns work?



SCALER Topics

Structure

- **The subject delivers events that are intriguing to the observers. These events occur due to changes in the state of the subject or the execution of certain behaviors. Subjects have a registration architecture that enables new observers to join and existing observers to withdraw from the list.**
- **Whenever a new event occurs, the subject iterates through the list of observers, calling the notify method provided in the 'observer' interface.**
- **The notification interface is declared by the Observer interface. It usually includes an updating method. The method may include numerous options that allow the subject to provide event information together with the update.**
- **Concrete Observers do certain activities in response to alerts sent by the Subject. All the concrete observer classes should implement the base observer interface, and the subject interface is coupled only with the base observer interface.**
- **Sometimes, observers want some additional context in order to properly process any update notified by the Subject. As a result, the Subject frequently supplies some contextual information as parameters to the notification function. The subject can pass itself as an argument, allowing the subject to immediately retrieve any needed data.**

Implementation

1. Declare an Observer interface with an update method at the least.
2. Declare the subject interface and a couple of methods for attaching and detaching observer objects from the list of observers. (Remember that publishers must only interact with subscribers through the subscriber interface.)
3. Create a concrete subject class if needed and add the subscribers' list to this class. Since this concrete class extends the Subject interface, it inherits all its properties including adding and removing observers. Every time something significant happens inside the subject class, it must inform to all the observers about it.
4. In concrete observer classes, add the update notification methods. Some observers might want basic background information about the occurrence. The notification method accepts it as an input.

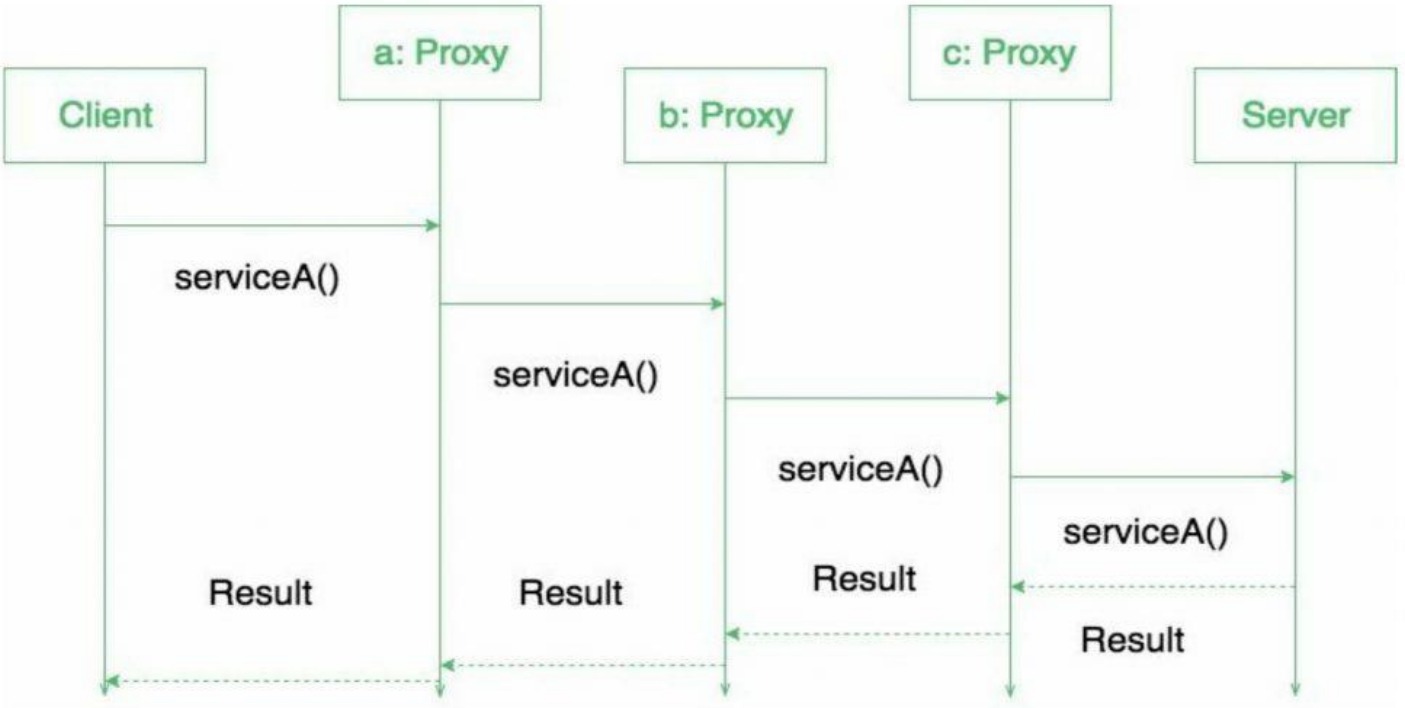
PROXY DESIGN PATTERN

Proxy means 'in place of', representing' or 'in place of' or 'on behalf of' are literal meanings of proxy and that directly explains Proxy Design Pattern. Proxies are also called surrogates, handles, and wrappers. They are closely related in structure, but not purpose, to Adapters and Decorators.

A real world example can be a cheque or credit card is a proxy for what is in our bank account. It can be used in place of cash, and provides a means of accessing that cash when required. And that's exactly what the Proxy pattern does – "Controls and manage access to the object they are protecting".

BEHAVIOR

As in the decorator pattern, proxies can be chained together. The client, and each proxy, believes it is delegating messages to the real server:



When to use this pattern?

Proxy pattern is used when we need to create a wrapper to cover the main object's complexity from the client.

TYPES OF PROXIES

Remoteproxy:

They are responsible for representing the object located remotely. Talking to the real object might involve marshalling and unmarshalling of data and talking to the remote object. All that logic is encapsulated in these proxies and the client application need not worry about them.

Virtual proxy:

These proxies will provide some default and instant results if the real object is supposed to take some time to produce results. These proxies initiate the operation on real objects and provide a default result to the application. Once the real object is done, these proxies push the actual data to the client where it has provided dummy data earlier.

Protection proxy:

If an application does not have access to some resource then such proxies will talk to the objects in applications that have access to that resource and then get the result back.

Smart Proxy:

A smart proxy provides additional layer of security by interposing specific actions when the object is accessed. An example can be to check if the real object is locked before it is accessed to ensure that no other object can change it.

Some Examples

A very simple real life scenario is our college internet, which restricts few site access. The proxy first checks the host you are connecting to, if it is not part of restricted site list, then it connects to the real internet. This example is based on Protection proxies.

Interface of Internet

```
package com.saket.demo.proxy;

public interface Internet

{

   public void connectTo(String serverhost) throws Exception;

}
```

Benefits:

- **One of the advantages of Proxy pattern is security.**
- **This pattern avoids duplication of objects which might be huge size and memory intensive. This in turn increases the performance of the application.**
- **The remote proxy also ensures about security by installing the local code proxy (stub) in the client machine and then accessing the server with help of the remote code.**

Drawbacks/Consequences:

This pattern introduces another layer of abstraction which sometimes may be an issue if the RealSubject code is accessed by some of the clients directly and some of them might access the Proxy classes. This might cause disparate behaviour.

FAÇADE:

The facade pattern (also spelled façade) is a software-design pattern commonly used in object-oriented programming. Analogous to a facade in architecture, a facade is an object that serves as a front-facing interface masking more complex underlying or structural code.

Facade is a part of the Gang of Four design patterns and it is categorized under Structural design patterns. Before we dig into the details of it, let us discuss some examples which will be solved by this particular Pattern. So, As the name suggests, it means the face of the building.

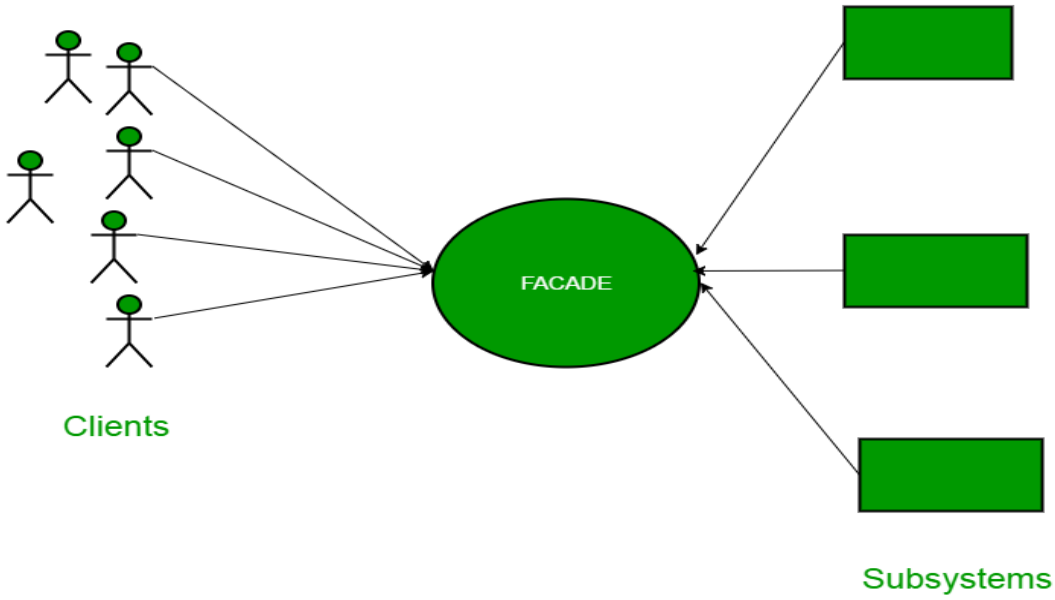
The people walking past the road can only see the glass face of the building. They do not know anything about it, the wiring, the pipes, and other complexities. It hides all the complexities of the building and displays a friendly face.

More examples

In Java, the interface JDBC can be called a facade because we as users or clients create connections using the "java.sql.Connection" interface, the implementation of which we are not concerned about.

The implementation is left to the vendor of the driver. Another good example can be the startup of a computer. When a computer starts up, it involves the work of CPU, memory, hard drive, etc.

To make it easy to use for users, we can add a facade that wraps the complexity of the task, and provide one simple interface instead. The same goes for the Facade Design Pattern. It hides the complexities of the system and provides an interface to the client from where the client can access the system.



FACADE DESIGN PATTERN DIAGRAM

Now Let's try and understand the facade pattern better using a simple example. Let's consider a hotel. This hotel has a hotel keeper.

There are a lot of restaurants inside the hotel e.g. Veg restaurants, Non-Veg restaurants, and Veg/Non Both restaurants. You, as a client want access to different menus of different restaurants. You do not know what are the different menus they have. You just have access to a hotel keeper who knows his hotel well. Whichever menu you want, you tell the hotel keeper and he takes it out of the respective restaurants and hands it over to you. Here, the hotel keeper acts as the facade, as he hides the complexities of the system hotel. Let's see how it works:

1. **Facade is a structural design pattern with the intent to provide a simplified (but limited) interface to a complex system of classes, library or framework.**

2. **A Facade class can often be transformed into a Singleton since a single facade object is sufficient in most cases.**

3. **Facade routinely wraps multiple objects.**

4. **The Facade class will not encapsulate subclasses but will provide a simple interface to their functions.**

5. **Classes in the subsystem will still be available to the client. However, Facade will decouple client and subsystems so that they can change independently.**

6. **We can combine different features by writing different Facade classes for different clients.**

7. **Facade can be recognized in a class that has a simple interface, but delegates most of the work to other classes.**

8. **Usually, facades manage the full life cycle of objects they use.**

9. **Client is shielded from the unwanted complexities of the subsystems and gets only to a fraction of a subsystem's capabilities.**

10. **Abstract Factory can serve as an alternative to Facade when you only want to hide the way the subsystem objects are created from the client code.**

Usage:

1. **Use the Facade pattern when you need to have a limited but straightforward interface to a complex subsystem.**

2. **Use the Facade when you want to structure a subsystem into layers.**

Related Patterns:

1. **Combined with Abstract Factory or used as an alternative to it.**

2. **Often confused with Adapter and mediator patterns.**

Some use cases in Enterprise Applications:

1. **Decoupling Application Code From the Library.**
2. **Reusing Legacy Code in New Application.**
3. **Fixing Interface Segregation Principle Violation.**