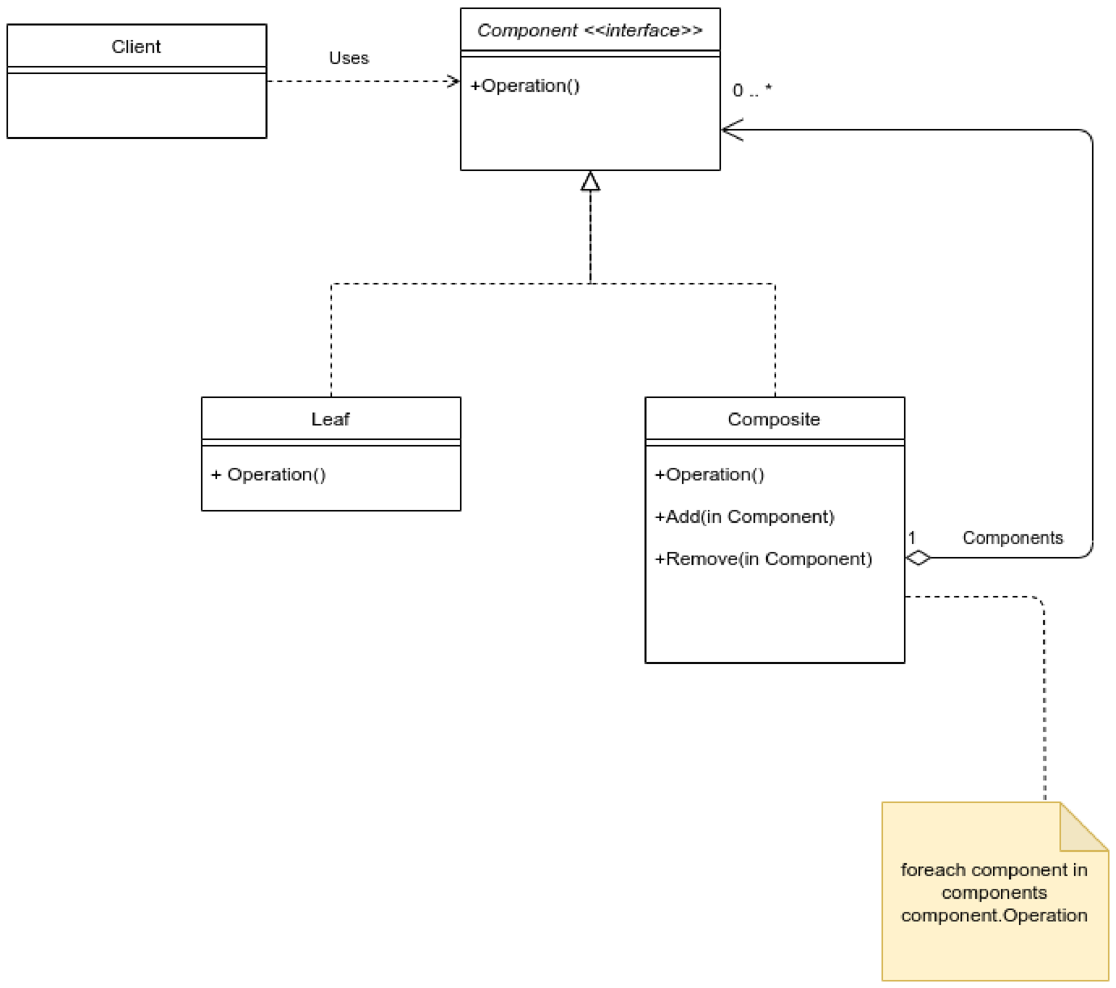
Structural:

1) Composite:

We use this pattern whenever we want to represent hierarchy and we want treat the objects in this hierarchy the same way, whether they're containers or parts, we want to treat them the same way

The Composite Pattern allows you to compose objects into tree structures to represent part whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

The Composite class is a container. This is where the name comes from. Because this composite class can be composed of zero or more components and each component can be a leaf , or a composite object.

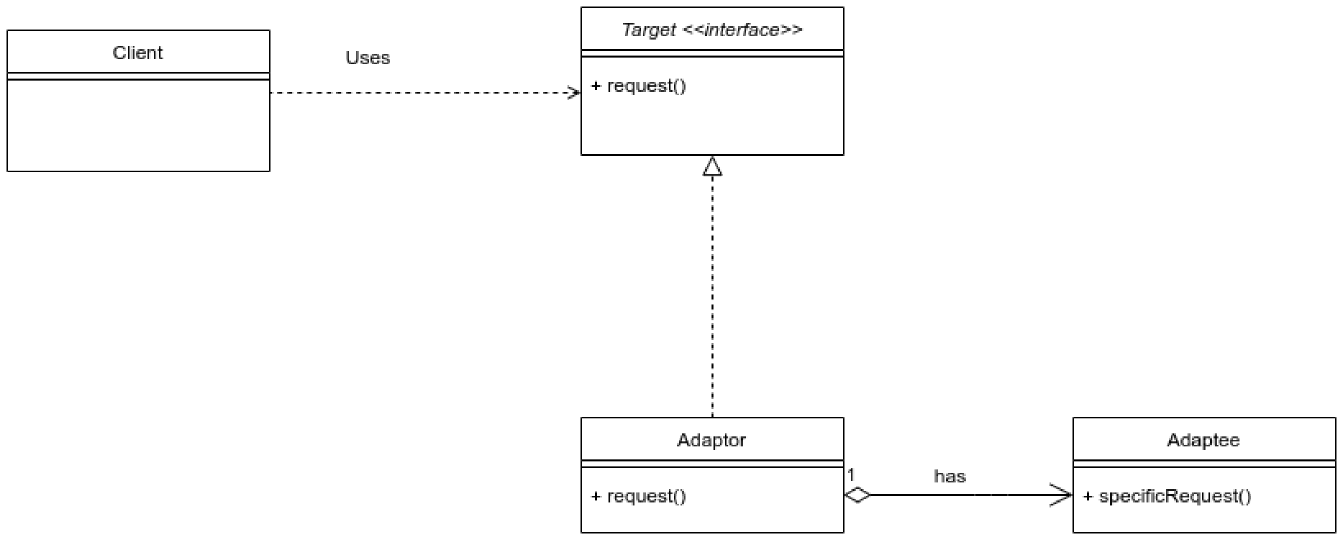


2) Adapter:

We use this pattern in situations where we have an existing class, and we to use it somewhere. But the interface of this class does not match the form of that we expect, so we use the adapter pattern to convert the interface of this class to different form.

The Adapter Pattern converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

The Adapter uses to convert an interface of a class to a different form, just how we use adapters in everyday life. With these adapters, we could convert the interface of an object to a different form.



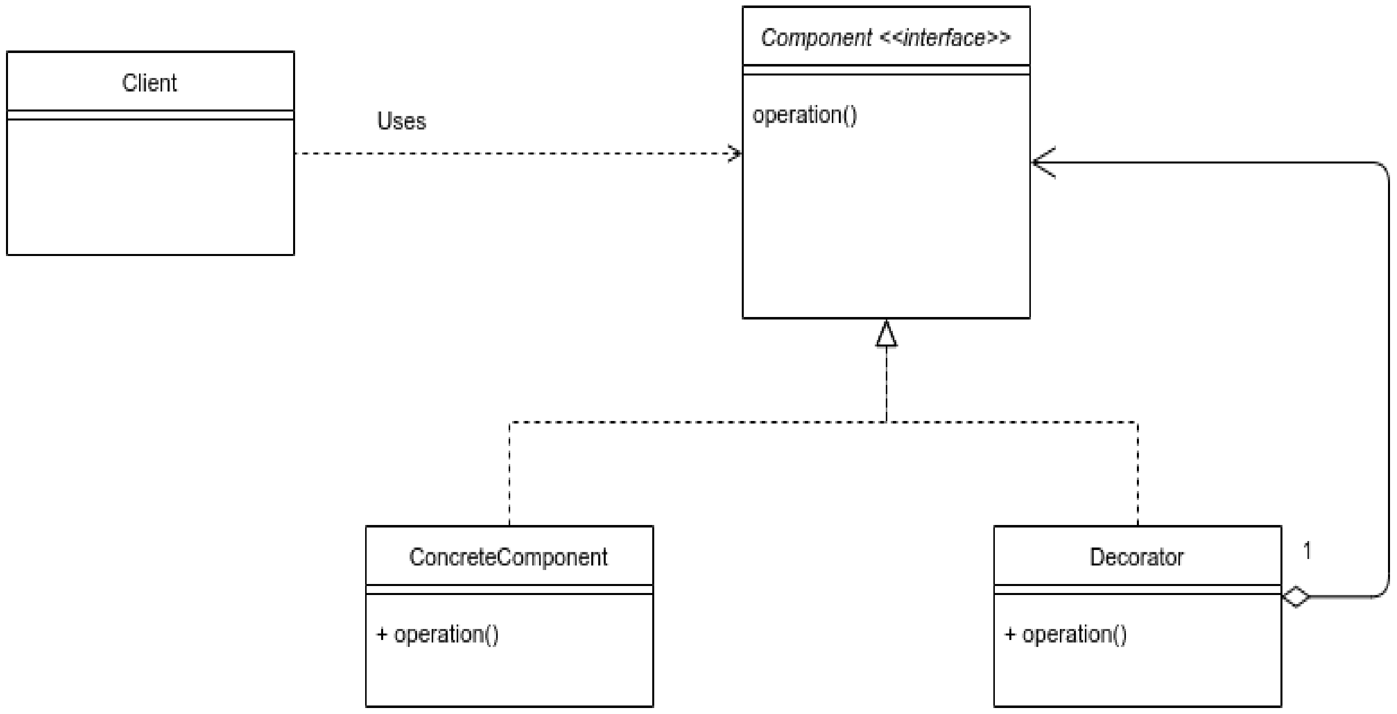
3) Decorator:

We use this pattern in situations where we want to add additional behavior to an existing object.

Difference between Adapter and Decorator:
With the adapter pattern, we change the interface of a class to a different form but with the decorator pattern we add additional behavior to an object.

The Decorator Pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

The Decorator class can decorate an existing object with additional behavior. This is where the name comes from.

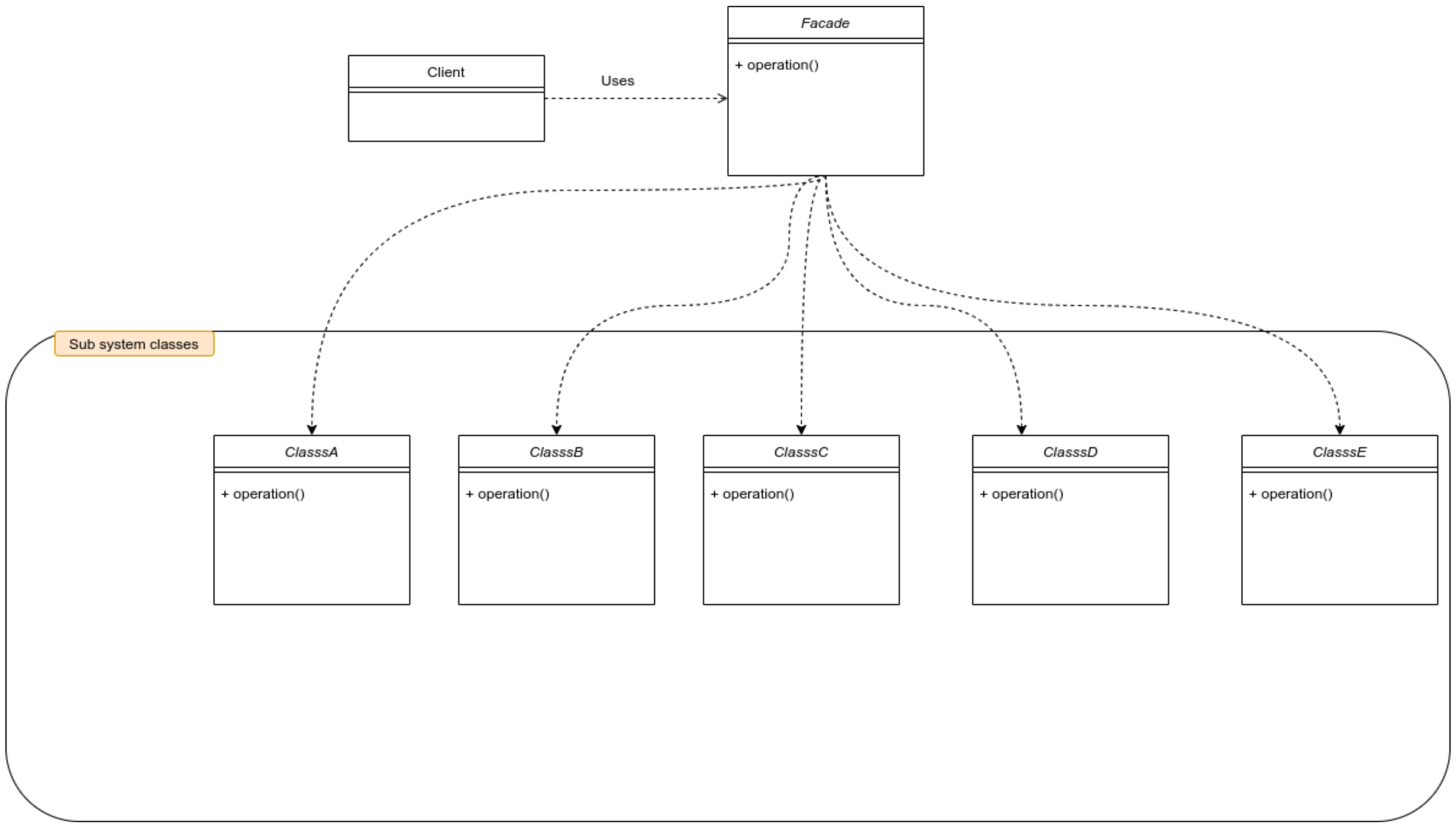


4) Facade:

We use this pattern to provide a simple interface to a complex system.

The Facade Pattern provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Facade means front or face and the Facade class is acting as a facade, or front of our application. This is where the name comes from.

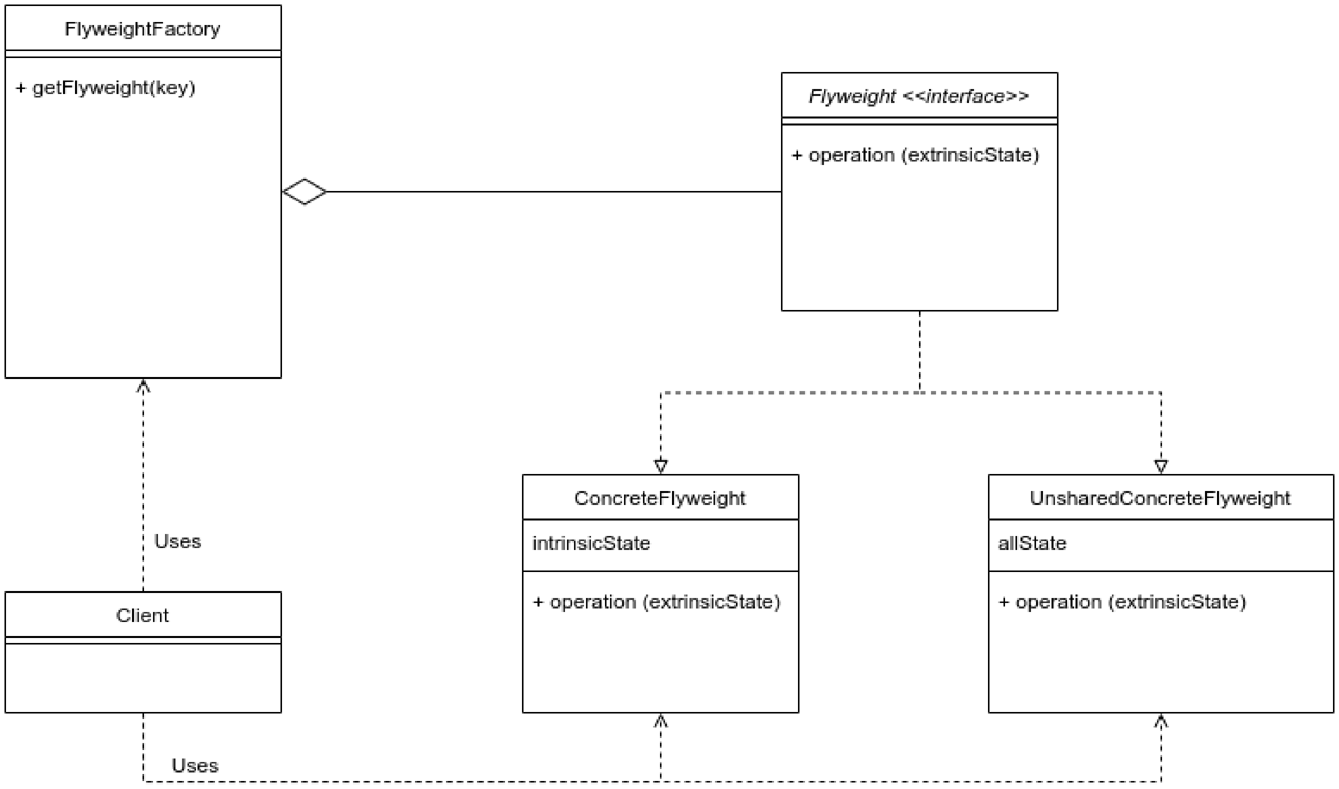


5) Flyweight:

We use this pattern in applications where we have a large number of objects , and these objects take a significant amount of memory with the flyweight pattern , we can reduce the amount of memory consumed by these objects.

With the Flyweight Pattern, we need to separate the data that we need to share store it somewhere else in a Flyweight class, and then implement a factory for caching these objects.

Flyweight is an object that we can share. This is where the name comes from.

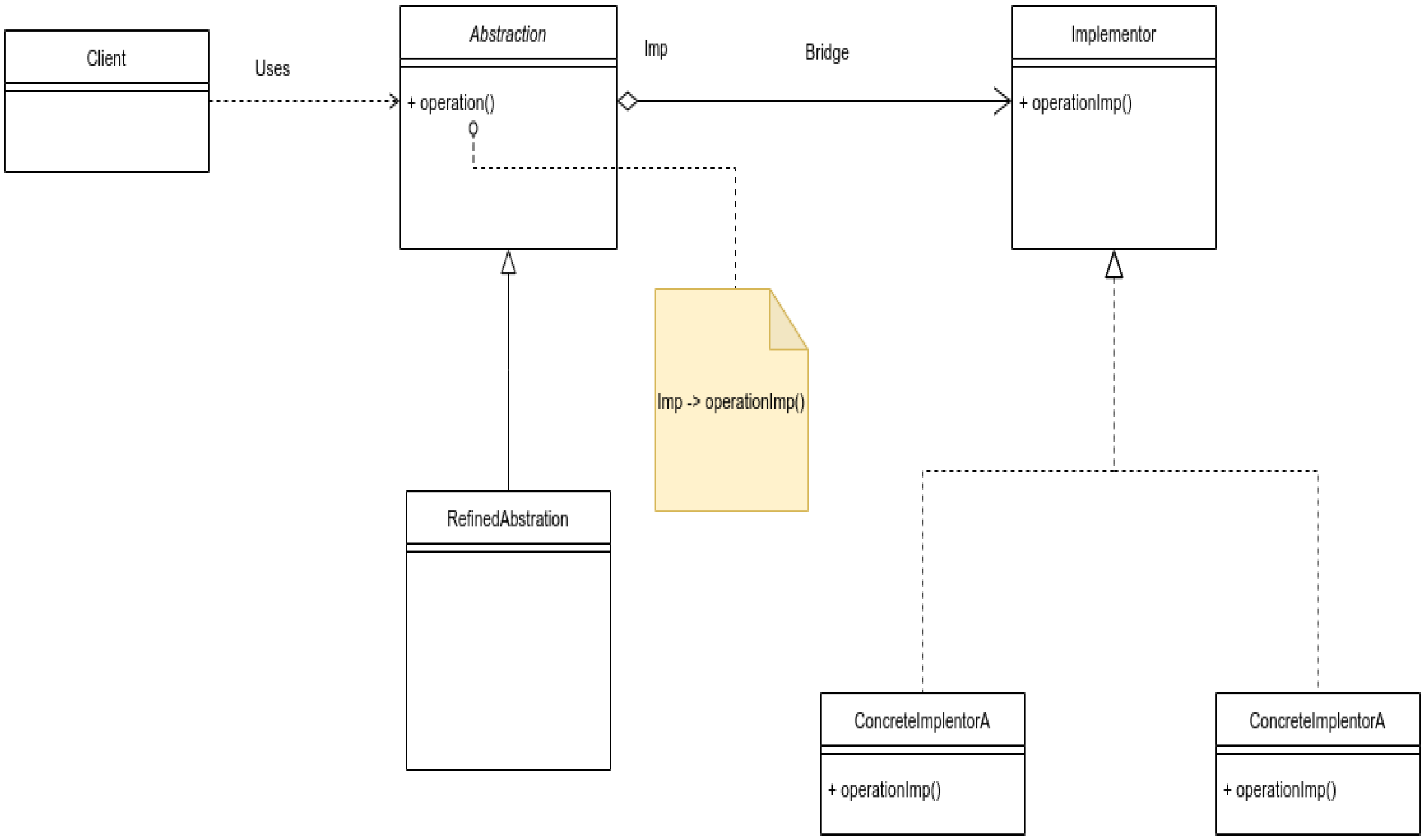


6) Bridge:

With the bridge pattern we can build flexible hierarchies that can grow independent of each order.

Use the Bridge Pattern to vary not only your implementations, but also your abstractions.

The reason we this pattern is called bridge pattern is because of this relationship between these two hierarchies. So this relationship is acting as a bridge and it connects two completely independent hierarchies

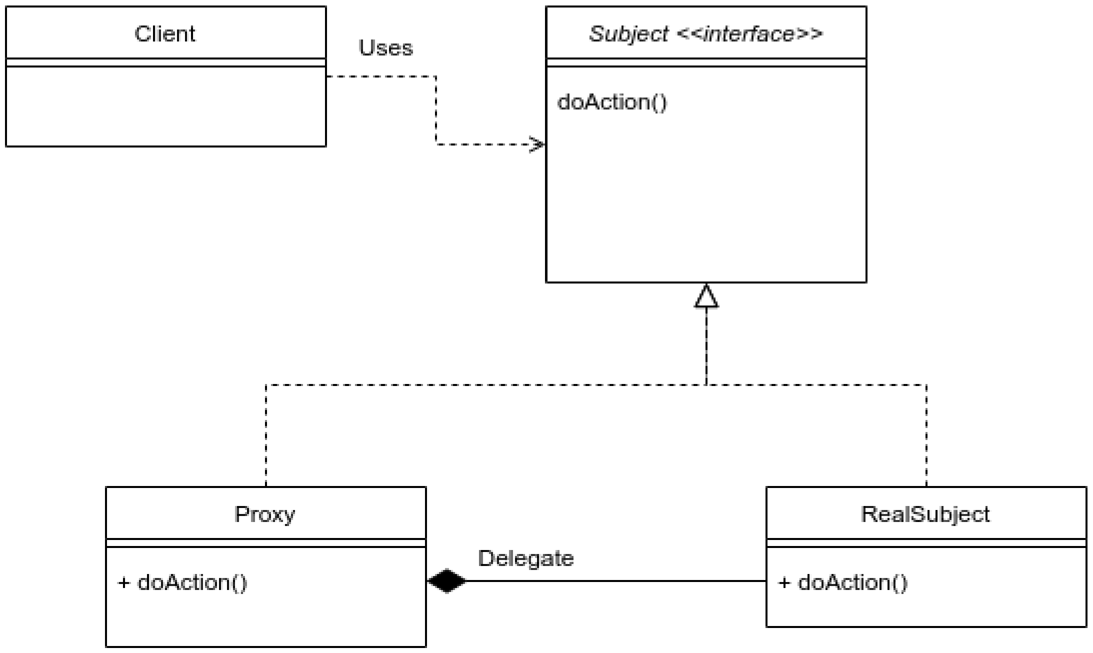


7) Proxy:

The Proxy pattern as the name implies, allows us to create a proxy or an agent for a real object. So if you want to talk to an object ,will talk to it through its proxy or agent, this agent takes our message and forwards it to the target object.

The Proxy Pattern provides a surrogate or placeholder for another object to control access to it.

The Proxy pattern as the name implies, allows us to create a proxy or an agent for a real object.


Client
Uses
Subject <<interface>>
doAction()
Proxy
+ doAction()
Delegate
RealSubject
+ doAction()