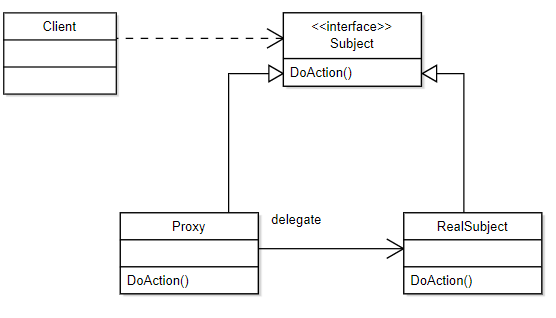
Composite UI :

Gmail ui, Visual Studio ui , Basically putting all the functionality in one page.

Proxy Pattern :

A *proxy*, in its most general form, is a class functioning as an interface to something else. The proxy could interface to anything: a network connection, a large object in memory, a file, or some other resource that is expensive or impossible to duplicate. In short, a proxy is a wrapper or agent object that is being called by the client to access the real serving object behind the scenes. Use of the proxy can simply be [forwarding](https://en.wikipedia.org/wiki/Forwarding_(object-oriented_programming)) to the real object, or can provide additional logic. In the proxy, extra functionality can be provided, for example caching when operations on the real object are resource intensive, or checking preconditions before operations on the real object are invoked. For the client, usage of a proxy object is similar to using the real object, because both implement the same interface.



interface Image {

public void displayImage();

}

*// On System A*

class RealImage implements Image {

private final String filename;

*/\*\**

*\* Constructor*

*\* @param filename*

*\*/*

public RealImage(String filename) {

this.filename = filename;

loadImageFromDisk();

}

*/\*\**

*\* Loads the image from the disk*

*\*/*

private void loadImageFromDisk() {

System.out.println("Loading " + filename);

}

*/\*\**

*\* Displays the image*

*\*/*

public void displayImage() {

System.out.println("Displaying " + filename);

}

}

*// On System B*

class ProxyImage implements Image {

private final String filename;

private RealImage image;

*/\*\**

*\* Constructor*

*\* @param filename*

*\*/*

public ProxyImage(String filename) {

this.filename = filename;

}

*/\*\**

*\* Displays the image*

*\*/*

public void displayImage() {

if (image == null) {

image = new RealImage(filename);

}

image.displayImage();

}

}

class ProxyExample {

*/\*\**

*\* Test method*

*\*/*

public static void main(final String[] arguments) {

Image image1 = new ProxyImage("HiRes\_10MB\_Photo1");

Image image2 = new ProxyImage("HiRes\_10MB\_Photo2");

image1.displayImage(); *// loading necessary*

image1.displayImage(); *// loading unnecessary*

image2.displayImage(); *// loading necessary*

image2.displayImage(); *// loading unnecessary*

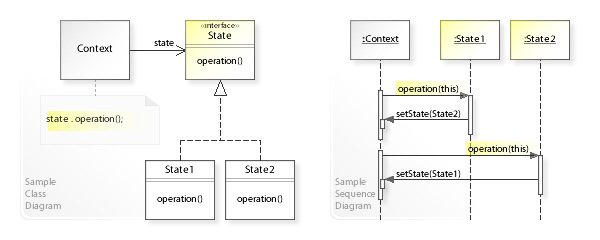
image1.displayImage(); *// loading unnecessary*

}

}

State Pattern

The state pattern is a [behavioral](https://en.wikipedia.org/wiki/Behavioral_pattern) [software design pattern](https://en.wikipedia.org/wiki/Software_design_pattern) that allows an object to alter its behavior when its internal state changes. This pattern is close to the concept of [finite-state machines](https://en.wikipedia.org/wiki/Finite-state_machine). The state pattern can be interpreted as a [strategy pattern](https://en.wikipedia.org/wiki/Strategy_pattern), which is able to switch a strategy through invocations of methods defined in the pattern's interface.

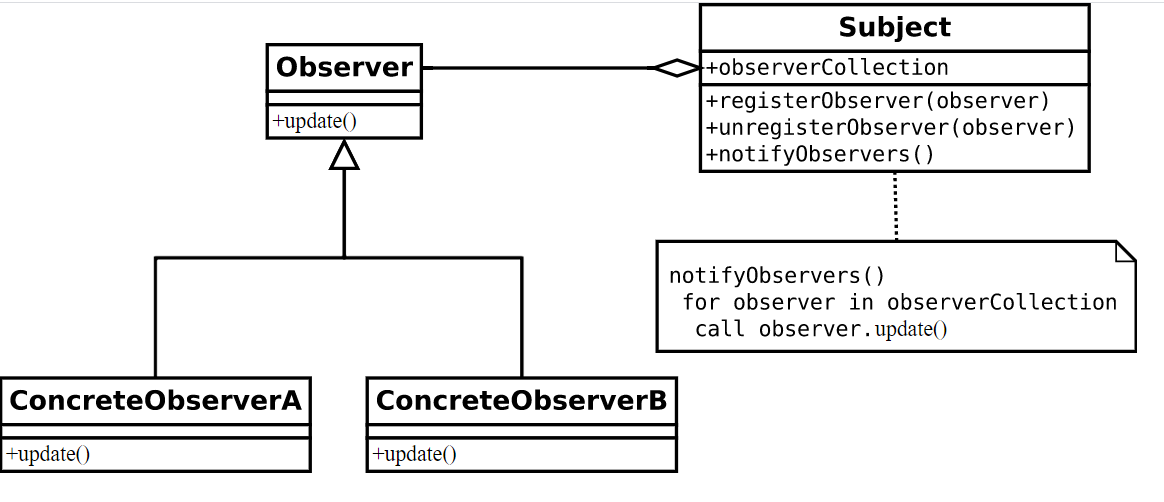


Chain of Responsibility

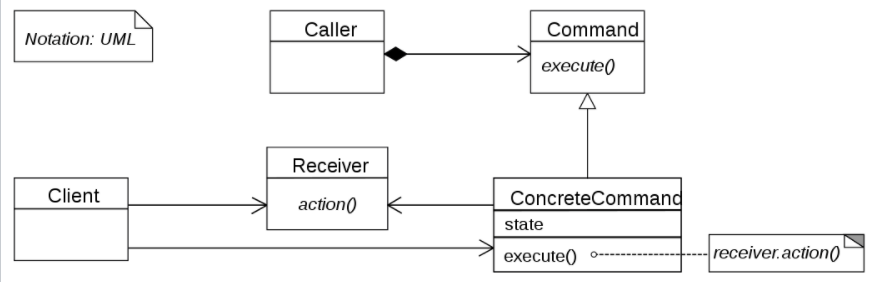
A source of [command objects](https://en.wikipedia.org/wiki/Command_pattern) and a series of processing objects.[[1]](https://en.wikipedia.org/wiki/Chain-of-responsibility_pattern#cite_note-1) Each processing object contains logic that defines the types of command objects that it can handle; the rest are passed to the next processing object in the chain. A mechanism also exists for adding new processing objects to the end of this chain.

Observer Pattern

It is mainly used for implementing distributed [event handling](https://en.wikipedia.org/wiki/Event_handling) systems, in "event driven" software. In those systems, the subject is usually named a "stream of events" or "stream source of events", while the observers are called "sinks of events".



Command Pattern : mobile code , progress bar



In [object-oriented programming](https://en.wikipedia.org/wiki/Object-oriented_programming), the command pattern is a [behavioral](https://en.wikipedia.org/wiki/Behavioral_pattern) [design pattern](https://en.wikipedia.org/wiki/Design_pattern_(computer_science)) in which an object is used to [encapsulate](https://en.wikipedia.org/wiki/Information_hiding) all information needed to perform an action or trigger an event at a later time. This information includes the method name, the object that owns the method and values for the method parameters.