**Command Pattern**

**Problem Description / Context**

* Want to encapsulate requests so that they can be executed, undone, queue, etc. independently of the request.
  + This would be making commands behave like objects -- can store additional info with them or put them into collections

**Where and When Chain of Responsibility pattern is applicable :**

* When you want to decouple a request’s sender and receiver
* Multiple objects, determined at runtime, are candidates to handle a request
* When you don’t want to specify handlers explicitly in your code
* When you want to issue a request to one of several objects without specifying the receiver explicitly.

**Composite Pattern**

**Problem Description / Context**

* Primitive objects can be combined into composite objects
* Clients treat composite object as a primitive object -- i.e. we'd like to be able to use some of the same basic interface for both composite and the primitives

# **Iterator Pattern**

lets it iterate through its objects. However while doing so it should make sure that it does not expose its implementation.

# **Mediator design pattern**

Mediator enables decoupling of objects by introducing a layer in between so that the interaction between objects happen via the laye

**Observer Pattern**

**Problem Description / Context**

* One object, the *subject* (our text calls the *publisher*), maintains some state and/or is the source of specific events
* Other objects, the *observers* (our text calls them *subscribers*), wish to know when something happens to the subject (like a state change or an event), so that they can respond and maintain consistency

# **State Design Pattern**

State design pattern is used when an Object changes its behavior based on its internal state.

Façade

The facade pattern is appropriate when you have a **complex system**that you want to expose to clients in a simplified way, or you want to make an external communication layer over an existing system which is incompatible with the system. Facade deals with interfaces, not implementation. Its purpose is to hide internal complexity behind a single interface that appears simple on the outside.

# **Bridge Design Pattern**

The Bridge design pattern allows you to separate the abstraction from the implementation.It is a structural design pattern.