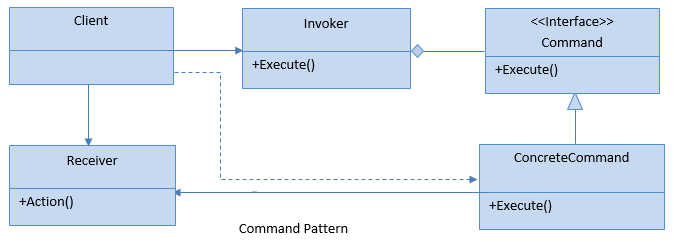
# Software Design Patterns

## Command Pattern

The Command pattern is a Behavioral Pattern. It is commonly used in the menu systems of many applications such as Editor, IDE etc. In this pattern, a request is wrapped under an object as a command and passed to invoker object. Invoker object passes the command to the appropriate object which can handle it and that object executes the command. This handles the request in traditional ways like as queuing and callbacks.

*Command Pattern - UML Diagram & Implementation*



The classes, interfaces and objects in the above UML class diagram are as follows:

**Client**: This is the class that creates and executes the command object.

**Invoker**: Asks the command to carry out the action.

**Command**: This is an interface which specifies the Execute operation.

**ConcreteCommand**: This is a class that implements the Execute operation by invoking operation(s) on the Receiver.

**Receiver**: This is a class that performs the Action associated with the request.

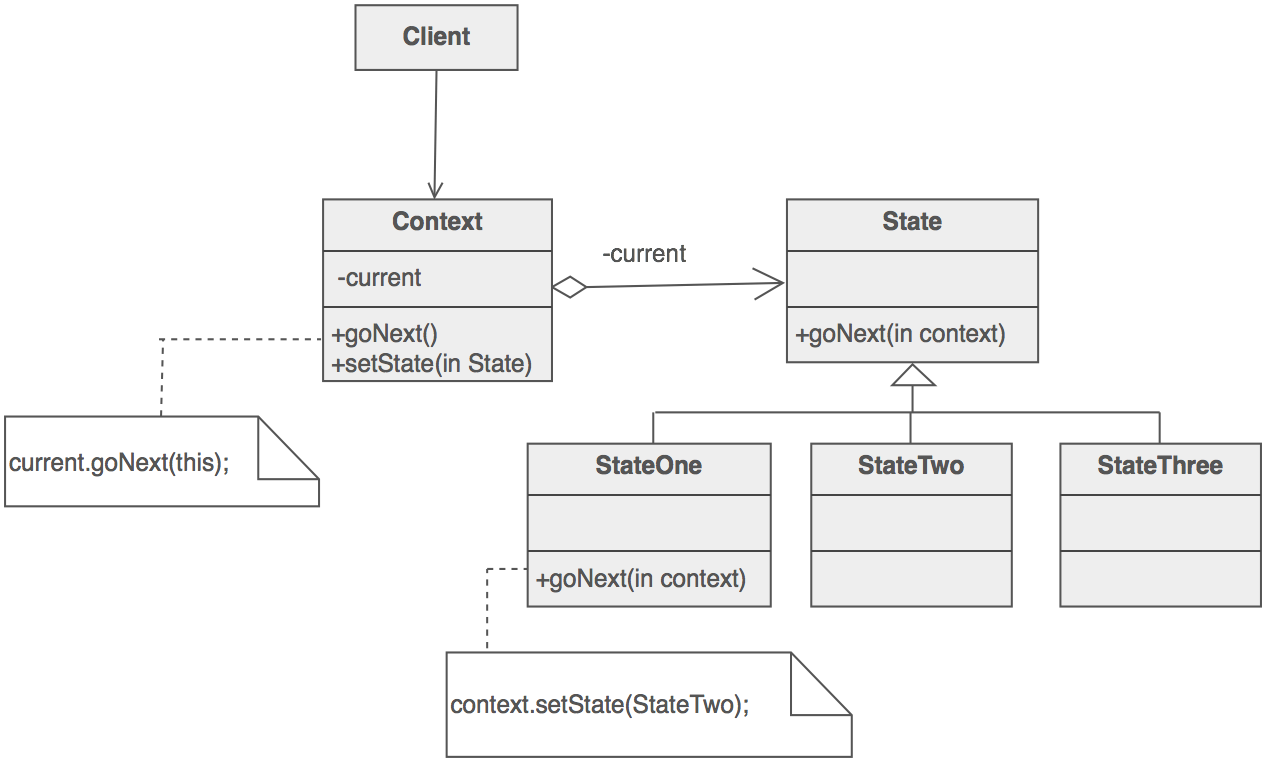
### When to use it?

* Need to implement callback functionalities.
* Need to support Redo and Undo functionality for commands.
* Sending requests to different receivers which can handle it in different ways.
* Need for auditing and logging of all changes via commands.

## State Design Pattern

The State pattern is a Behavioral Pattern, which is a solution to the problem of how to make behavior depend on state. For example, a monolithic object's behavior is a function of its state, and it must change its behavior at run-time depending on that state. Or, an application is characterized by large and numerous case statements that vector flow of control based on the state of the application.

*State Pattern - UML Diagram & Implementation*



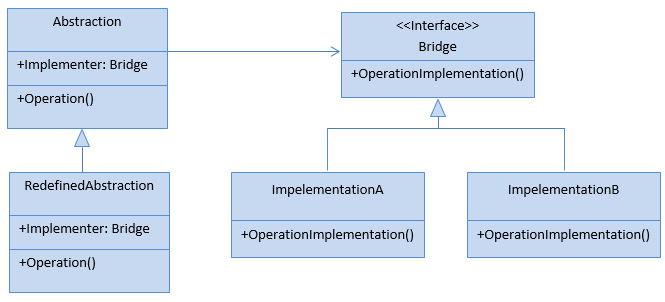
* Define a **Context** class to present a single interface to the outside world.
* Define a **State** abstract base class.
* Represent the different "states" of the state machine as **derived classes of the State** base class.
* Define state-specific behavior in the appropriate State derived classes.
* Maintain a pointer to the current "state" in the "context" class.
* To change the state of the state machine, change the current "state" pointer.

The State pattern does not specify where the state transitions will be defined. The choices are two: the "context" object, or each individual State derived class. The advantage of the latter option is ease of adding new State derived classes. The disadvantage is each State derived class has knowledge of (coupling to) its siblings, which introduces dependencies between subclasses.

## Bridge Design Pattern

The Bridge pattern is a Structural Pattern. All we know, Inheritance is a way to specify different implementations of an abstraction. But in this way, implementations are tightly bound to the abstraction and cannot be modified independently. The Bridge pattern provides an alternative to inheritance when there are more than one version of an abstraction. This pattern involves an interface which acts as a bridge between the abstraction class and implementer classes and also makes the functionality of implementer class independent from the abstraction class. Both types of classes can be modified without affecting to each other.

*Bridge Pattern - UML Diagram & Implementation*



The classes, interfaces and objects in the above UML class diagram are as follows:

**Abstraction**: This is an abstract class and containing members that define an abstract business object and its functionality. It contains a reference to an object of type Bridge. It can also acts as the base class for other abstractions.

**Redefined Abstraction**: This is a class which inherits from the Abstraction class. It extends the interface defined by Abstraction class.

**Bridge**: This is an interface which acts as a bridge between the abstraction class and implementer classes and also makes the functionality of implementer class independent from the abstraction class.

**ImplementationA & ImplementationB**: These are classes which implement the Bridge interface and also provide the implementation details for the associated Abstraction class.

### When to use it?

* Abstractions and implementations should be modified independently.
* Changes in the implementation of an abstraction should have no impact on clients.