**Design Patterns**

Design patterns are solutions to general problems that software developers faced during software development.

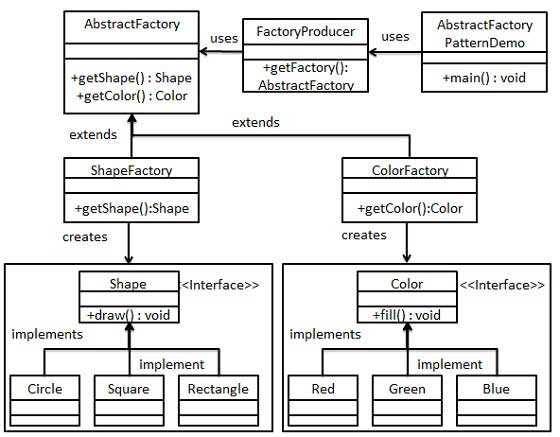
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| # | Creational | Structural | Behavioral | Others |
| 1 | Factory | Adapter | Chain Of Responsibility | Data Access Object |
| 2 | Abstract Factory | Bridge | Command | Front Controller |
| 3 | Builder | Composite | Interpreter |  |
| 4 | Prototype | Decorator | Iterator |  |
| 5 | Singleton | Façade | Mediator |  |
| 6 | Object Pool | Flyweight | Observer |  |
| 7 |  | Proxy | State |  |
| 8 |  |  | Strategy |  |
| 9 |  |  | Template |  |
| 10 |  |  | Visitor |  |

**Factory**: In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

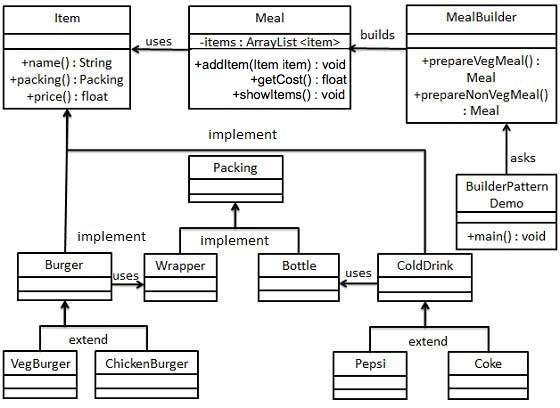


**Abstract Factory:** Factory of factory

Eg: Multiple Suites of related Objects



**Builder**: Separate the construction of a complex object from its representation so that the same construction process can create different representations **Problem** - An application needs to create the elements of a complex aggregate.

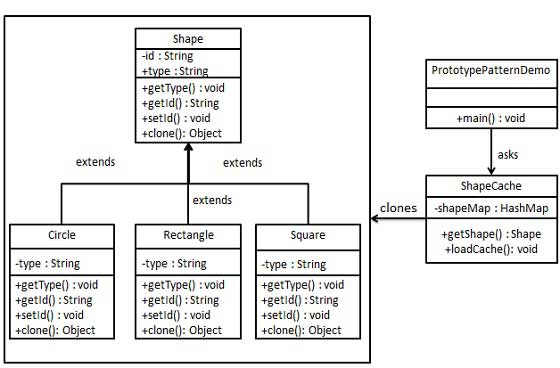


**Factory**: We create object without exposing the creation logic to the client and refer to newly created object using a common interface **Problem** - A framework needs to standardize the architectural model for a range of applications, but allow for individual applications to define their own domain objects and provide for their instantiation

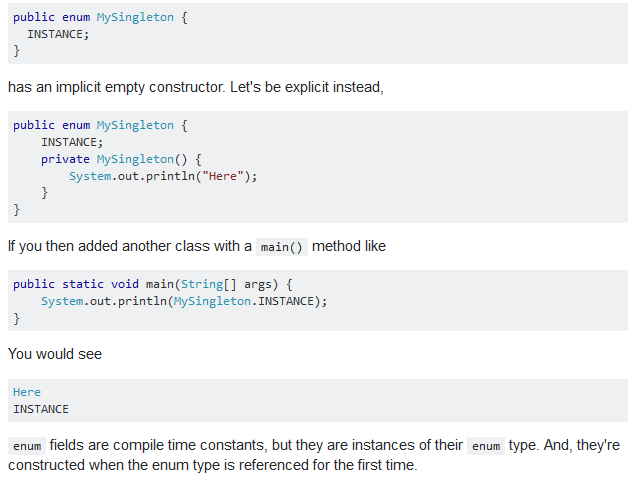


**Object Pool**: Object pooling can offer a significant performance boost; it is most effective in situations where the cost of initializing a class instance is high, the rate of instantiation of a class is high, and the number of instantiations in use at any one time is low.

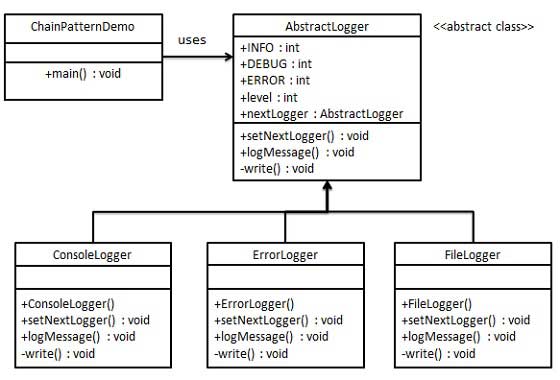
**Prototype**: Prototype pattern refers to creating duplicate object while keeping performance in mind. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.



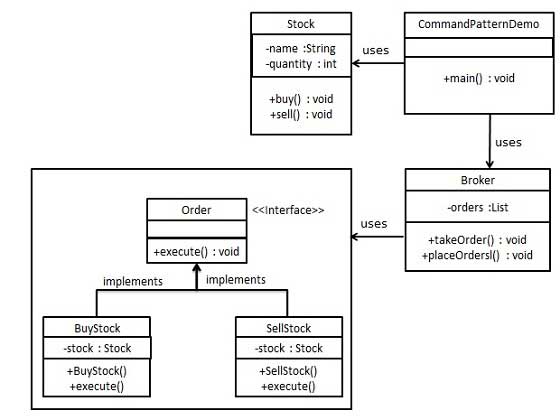
**Singleton**: Ensure a class has only one instance, and provide a global point of access to it.



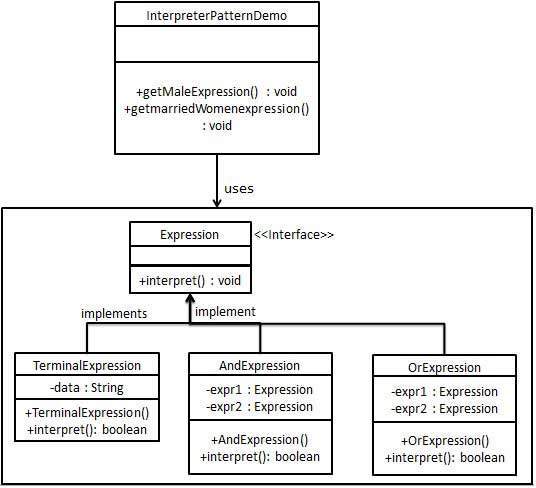
**Chain of Responsibility:** Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.



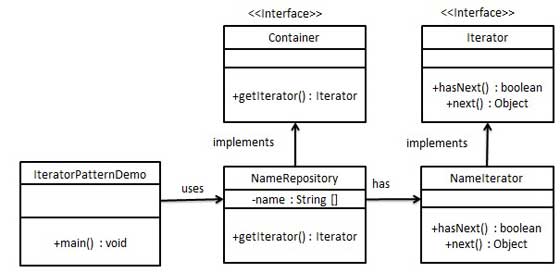
**Command:** Solves Need to issue requests to objects without knowing anything about the operation being requested or the receiver of the request || A request is wrapped under an object as command and passed to invoker object. Invoker object looks for the appropriate object which can handle this command and passes the command to the corresponding object which executes the command.



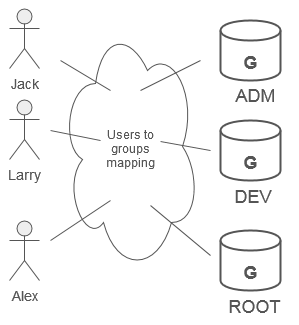
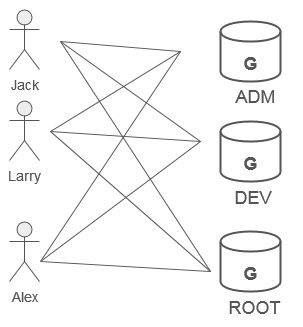
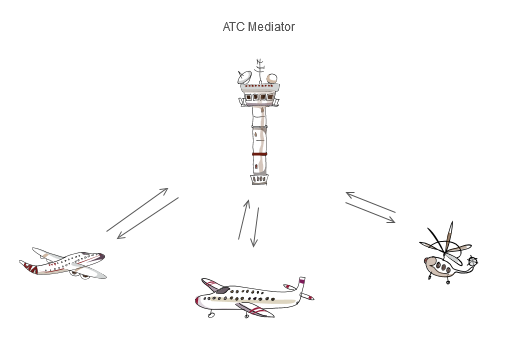
**Interpreter**: Interpreter pattern provides a way to evaluate language grammar or expression. This pattern is used in SQL parsing, symbol processing engine etc. (Reverse Polish = [+ - 10 2 3] is [10-2 + 3] DFS in tree solving left)



**Iterator:** This pattern is used to get a way to access the elements of a collection object in sequential manner without any need to know its underlying representation.

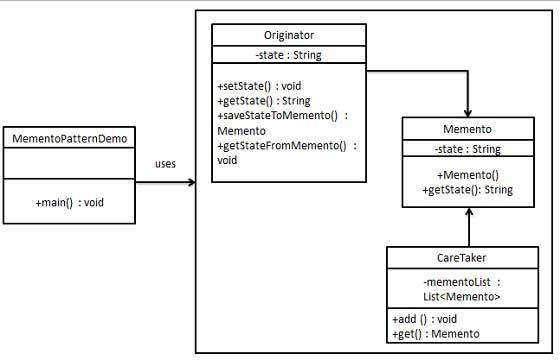


**Mediator**: Mediator pattern is used to reduce communication complexity between multiple objects or classes. This pattern provides a mediator class which normally handles all the communications between different classes and supports easy maintenance of the code by loose coupling. Solves - We want to design reusable components, but dependencies between the potentially reusable pieces demonstrates the "spaghetti code" phenomenon (trying to scoop a single serving results in an "all or nothing clump"). Basically avoid many-to-many relationships.



**Memento**: Solves - Need to restore an object back to its previous state. Without violating encapsulation, Capture and externalize an object's internal state so that the object can be returned to this state later.

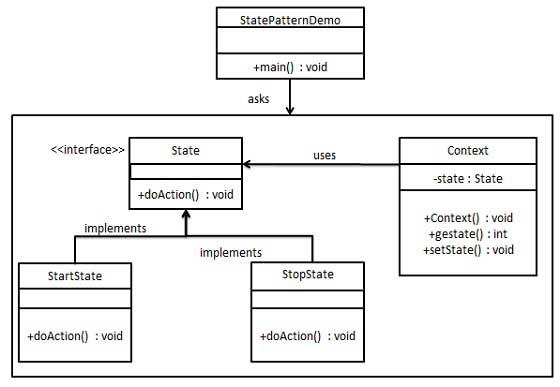
Memento pattern uses three actor classes. Memento contains state of an object to be restored. Originator creates and stores states in Memento objects and Caretaker object is responsible to restore object state from Memento.



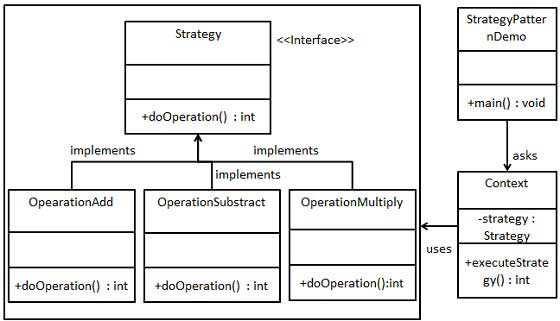
**Observer**: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. Subject(Observable) is an object having methods to attach and detach observers to a client object. We have created an abstract class *Observer* and a concrete class *Subject* that is extending class *Observer*.



**State**: A class behavior changes based on its state. *Problem* - 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.



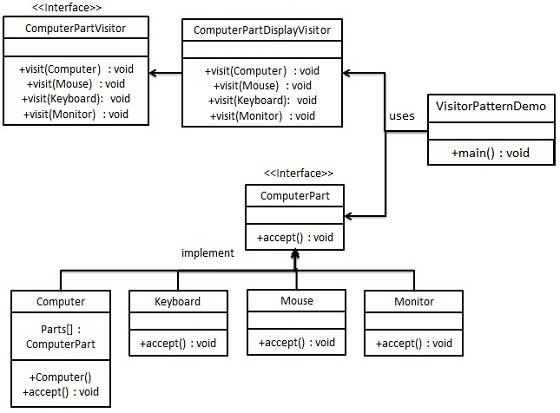
**Strategy:** Define a family of algorithms, encapsulate each one, and make them interchangeable(Open-Close principle).



**Template Method:** Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure



**Visitor:** Execution algorithm of element can vary as and when visitor varies || **Problem -** Many distinct and unrelated operations need to be performed on node objects in a heterogeneous aggregate structure. You want to avoid "polluting" the node classes with these operations. And, you don't want to have to query the type of each node and cast the pointer to the correct type before performing the desired operation.

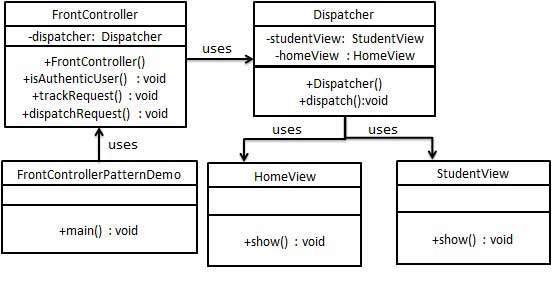


**Data Access Object Pattern**: used to separate low level data accessing API or operations from high level business services. Components: DAO Interface, DAO Concrete class, Model, VO or Entity to transfer data



**Front Controller**: The front controller design pattern is used to provide a centralized request handling mechanism so that all requests will be handled by a single handler. This handler can do the authentication/ authorization/ logging or tracking of request and then pass the requests to corresponding handlers

Components: Front Controller, Dispatcher, View



**Null Object**: The intent of a Null Object is to encapsulate the absence of an object by providing a substitutable alternative that offers suitable default do nothing behavior. In short, a design where "nothing will come of nothing"