**DESIGN PATTERNS**

**CREATIONAL DESIGN PATTERNS**

**SINGLETON**

Singleton pattern is a design pattern which restricts a class to instantiate its multiple objects. Class is defined in such a way that only one instance of the class is created in the complete execution of a program or project. It is used where only a single instance of a class is required to control the action throughout the execution. A singleton class shouldn’t have multiple instances in any case and at any cost.

Singleton classes are used for logging, driver objects, caching and thread pool, database connections.

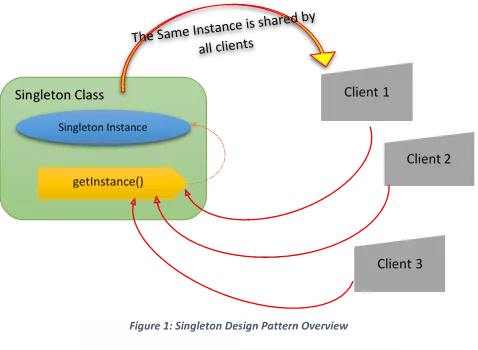


Fig: Singleton design pattern

**FACTORY**

This type of design pattern provides one of the best ways to create an object.In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

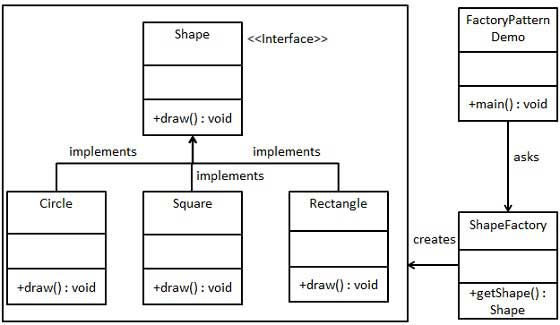


Fig: Factory design pattern

**ABSTRACT FACTORY**

Abstract Factory patterns work around a super-factory which creates other factories. This factory is also called as factory of factories.

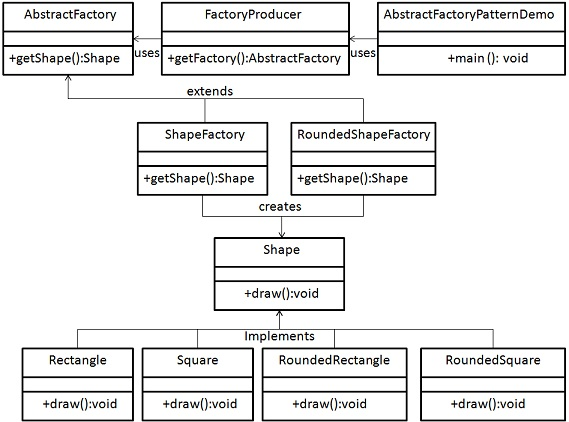


Fig: Abstract Factory design pattern

Steps

1. Create an interface Shape to draw.

2.Create concrete classes implementing the same interface.

3.Create an Abstract class to get factories for Normal and Rounded Shape Objects. I.e AbstractFactory.

4.Create Factory classes extending AbstractFactory to generate object of concrete class based on given information.They are ShapeFactory, RoundedShapeFactory.

5.Create a FactoryProducer class to get factories by passing an information such as Shape i.e rounded.

6.Create AbstractFactoryPatternDemo class and use the FactoryProducer to get AbstractFactory in order to get factories of concrete classes by passing an information such as type.

**BUILDER**

Builder pattern aims to “Separate the construction of a complex object from its representation so that the same construction process can create different representations.” It is used to construct a complex object step by step and the final step will return the object. The process of constructing an object should be generic so that it can be used to create different representations of the same object.

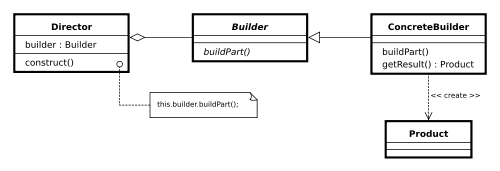
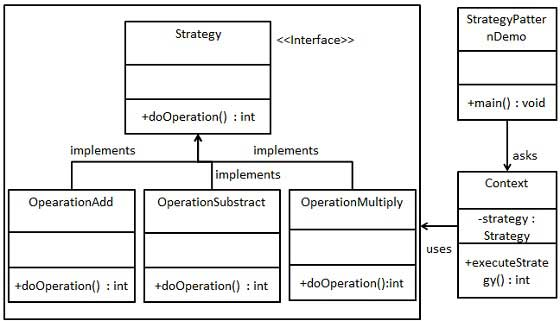


Fig:Builder design pattern

**Behavioural DESIGN PATTERN**

**STRATEGY**

In Strategy pattern, a class behavior or its algorithm can be changed at run time.In this, we create objects which represent various strategies and a context object whose behavior varies as per its strategy object. The strategy object changes the executing algorithm of the context object.



Steps:

1.Create an interface.

2.Create concrete classes implementing the same interface.

3.Create Context Class

4.Use the Context to see change in behaviour when it changes its Strategy.

**COMMAND**

A Command Pattern says that "encapsulates a request under an object as a command and pass it to invoker object. Invoker object looks for the appropriate object which can handle this command and pass the command to the corresponding object and that object executes the command". It is also known as Action or Transaction. It is used when you need to parameterize objects according to an action performed or when you need to create and execute requests at different times.

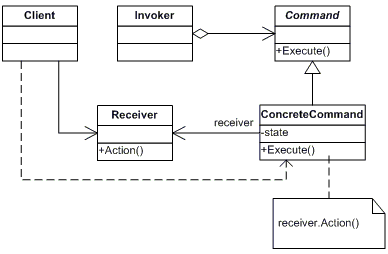


Fig: Command design pattern

**CHAIN OF RESPONSIBILITY**

The Chain of Responsibility design pattern allows an object to send a command without knowing what object will receive and handle it. The request is sent from one object to another making them parts of a chain and each object in this chain can handle the command, pass it on or do both. The most common example of a machine using the Chain of Responsibility is the vending machine coin slot: rather than having a slot for each type of coin, the machine has only one slot for all of them. The dropped coin is routed to the appropriate storage place that is determined by the receiver of the command.

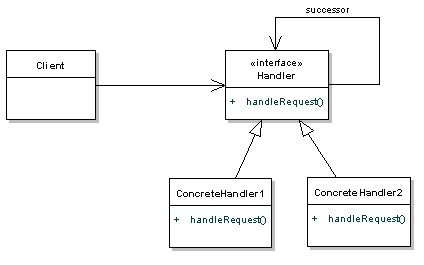


Fig: Chain of responsibility design pattern

**OBSERVER**

The Observer Pattern defines a one to many dependency between objects so that one object changes state, all its dependents are notified and updated automatically.

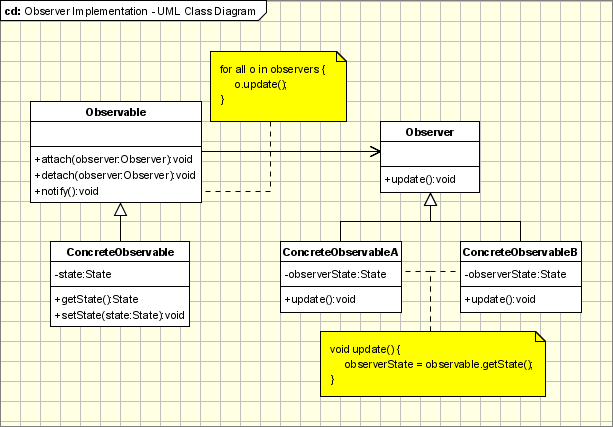


Fig: Observer design pattern

**STRUCTURAL DESIGN PATTERN**

**ADAPTER**

The adapter pattern is adapting between classes and objects. Like any adapter in the real world it is used to be an interface, a bridge between two objects. In the real world we have adapters for power supplies.It convert the interface of a class into another interface clients expect. Adapter lets classes work together, that could not otherwise because of incompatible interfaces.

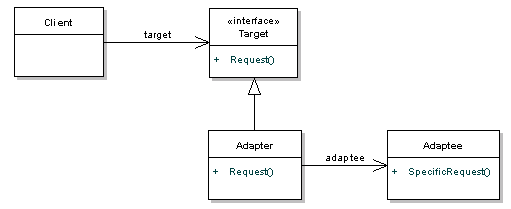


Fig: Adapter design pattern

**DECORATOR**

Decorator is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors. It provides greater flexibility than static inheritance. It enhances the extensibility of the object, because changes are made by coding new classes. It simplifies the coding by allowing you to develop a series of functionality from targeted classes instead of coding all of the behavior into the object. It is used when you want to transparently and dynamically add responsibilities to objects without affecting other objects.

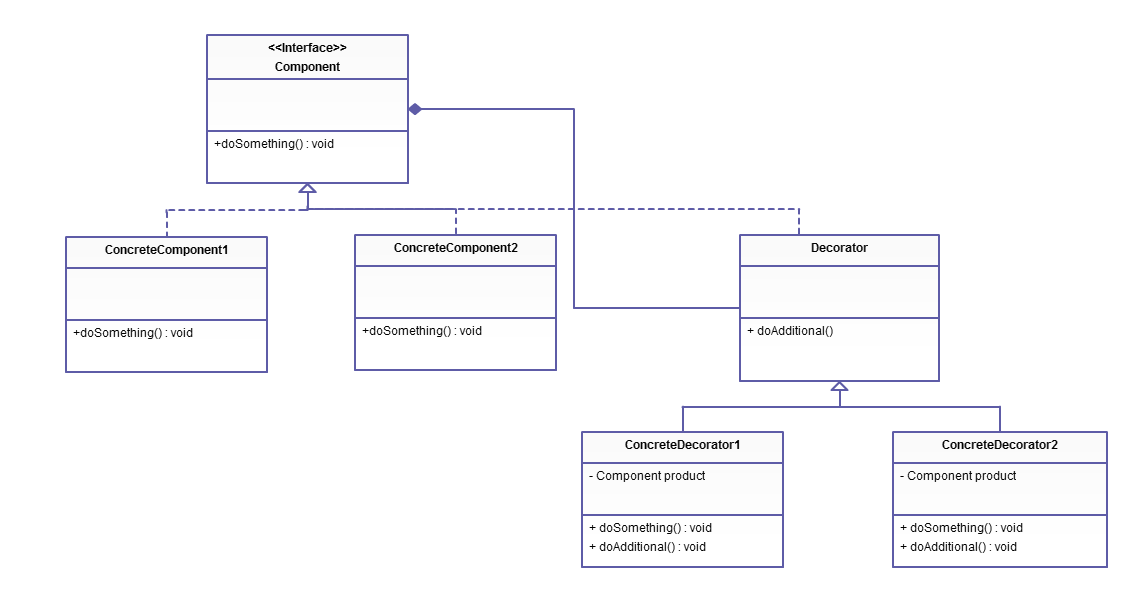


Fig: Decorator design pattern

**FLYWEIGHT**

Flyweight is a structural design pattern that lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object. A Flyweight Pattern allows us to reuse already existing similar kind of objects by storing them and create new object when no matching object is found. Its major advantage is that it reduces the number of objects and it reduces the amount of memory and storage devices required if the objects are persisted.

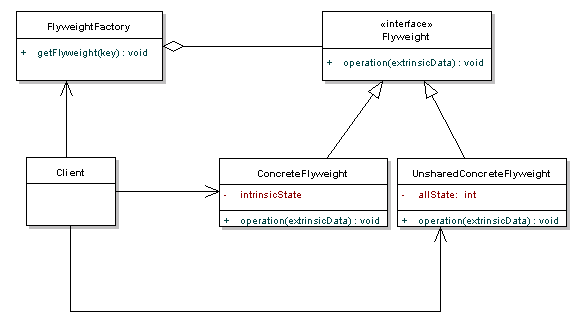
****

Fig: Flyweight design pattern