Design patterns represent the best practices used by experienced object-oriented software developers. Design patterns are solutions to general problems that software developers faced during software development. These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.

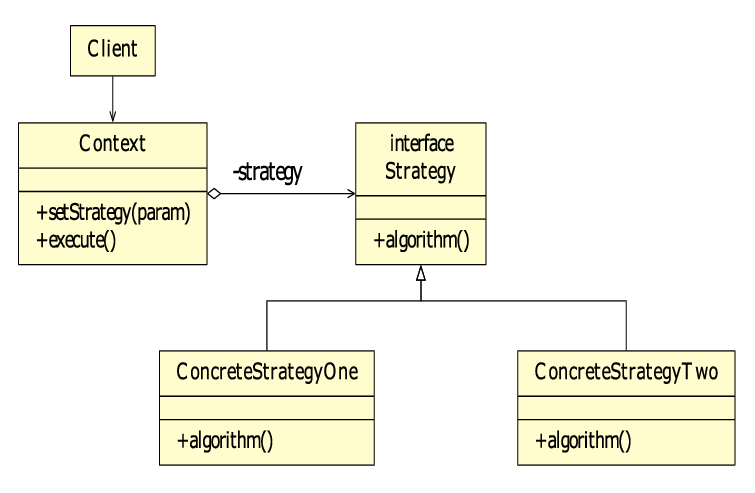
Design Patterns have two main usages in software development.

-Common platform for developers

Design patterns provide a standard terminology and are specific to particular scenario. For example, a singleton design pattern signifies use of single object so all developers familiar with single design pattern will make use of single object and they can tell each other that program is following a singleton pattern.

-Best Practices

Design patterns have been evolved over a long period of time and they provide best solutions to certain problems faced during software development. Learning these patterns helps unexperienced developers to learn software design in an easy and faster way.



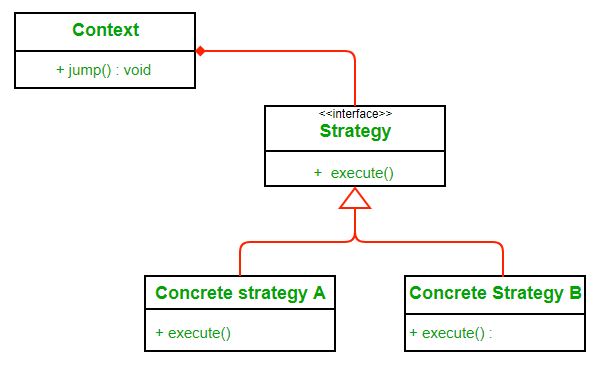
**Strategy Pattern**

this pattern by defining a problem and using strategy pattern to solve it. In Strategy pattern, a class behavior or its algorithm can be changed at run time. This type of design pattern comes under behavior pattern.

In Strategy pattern, 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.

Implementation:

We are going to create a *Strategy* interface defining an action and concrete strategy classes implementing the *Strategy* interface. *Context* is a class which uses a Strategy.



Here we rely on composition instead of inheritance for reuse. Context is composed of a Strategy. Instead of implementing a behavior the Context delegates it to Strategy. The context would be the class that would require changing behaviors. We can change behavior dynamically. Strategy is implemented as interface so that we can change behavior without affecting our context.

**Adapter Pattern**

Adapter pattern works as a bridge between two incompatible interfaces. This type of design pattern comes under structural pattern as this pattern combines the capability of two independent interfaces.

This pattern involves a single class which is responsible to join functionalities of independent or incompatible interfaces. A real life example could be a case of card reader which acts as an adapter between memory card and a laptop. You plugin the memory card into card reader and card reader into the laptop so that memory card can be read via laptop.

To use an adapter:

1. The client makes a request to the adapter by calling a method on it using the target interface.
2. The adapter translates that request on the adaptee using the adoptee interface.
3. Client receive the results of the call and is unaware of adapter’s presence.

 The client sees only the target interface and not the adapter. The adapter implements the target interface. Adapter delegates all requests to Adoptee.

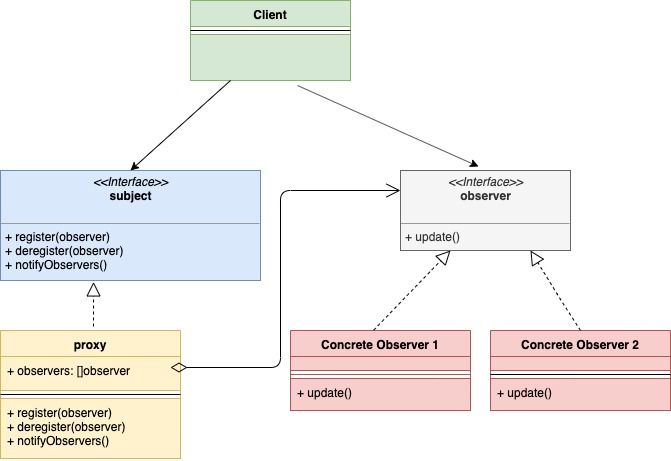
**Observer Pattern**

Observer pattern is used when there is one-to-many relationship between objects such as if one object is modified, its depenedent objects are to be notified automatically. Observer pattern falls under behavioral pattern category.

To understand observer pattern, first you need to understand the subject and observer objects.

The relation between subject and observer can easily be understood as an analogy to magazine subscription.

* A magazine publisher(subject) is in the business and publishes magazines (data).
* If you (user of data/observer) are interested in the magazine you subscribe(register), and if a new edition is published it gets delivered to you.
* If you unsubscribe(unregister) you stop getting new editions.
* Publisher doesn’t know who you are and how you use the magazine, it just delivers it to you because you are a subscriber (loose coupling).



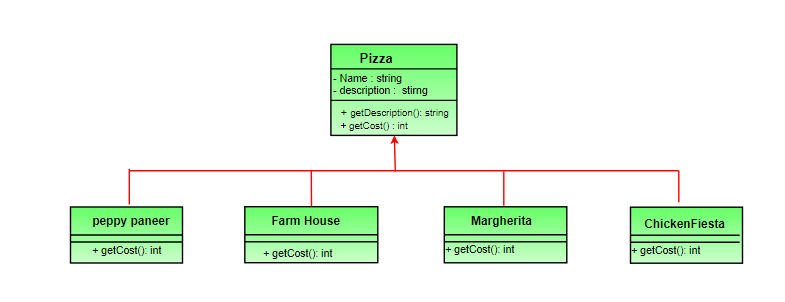
**Decorator Pattern**

Decorator pattern allows a user to add new functionality to an existing object without altering its structure. This type of design pattern comes under structural pattern as this pattern acts as a wrapper to existing class.

This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.

We are demonstrating the use of decorator pattern via following example in which we will decorate a shape with some color without alter shape class.

To understand decorator pattern let us consider a scenario inspired from the book “Head First Design Pattern”.  Suppose we are building an application for a pizza store and we need to model their pizza classes. Assume they offer four types of pizzas namely Peppy Paneer, Farmhouse, Margherita and Chicken Fiesta. Initially we just use inheritance and abstract out the common functionality in a Pizza class.



**Factory Pattern**

Factory pattern is one of the most used design patterns in Java. This type of design pattern comes under creational pattern as this 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.

Implementation

We're going to create a *Shape* interface and concrete classes implementing the *Shape* interface. A factory class *ShapeFactory* is defined as a next step.

*FactoryPatternDemo*, our demo class will use *ShapeFactory* to get a *Shape* object. It will pass information (*CIRCLE / RECTANGLE / SQUARE*) to *ShapeFactory* to get the type of object it needs.

