**DESIGN PATTERN**

[**https://www.youtube.com/watch?v=YDobmucohqkhttps://www.youtube.com/watch?v=YDobmucohqk**](https://www.youtube.com/watch?v=YDobmucohqkhttps://www.youtube.com/watch?v=YDobmucohqk)

**Clearly explained theory:** [**http://www.oodesign.com**](http://www.oodesign.com)

**Easy to understand videos with example for behavioural:** [**https://www.youtube.com/watch?v=9qA5kw8dcSU**](https://www.youtube.com/watch?v=9qA5kw8dcSU)

**Design patterns all easy :** [**https://ramj2ee.blogspot.in/#**](https://ramj2ee.blogspot.in/#) **Take snapshots from here**

Design patterns are time-tested solutions to OOP problems. All design patterns are centered around the concept of polymorphism. Polymorphism is a way to achieve decoupling (not hard-coding the implementation too early. Hard coding is deferred till run time.). Why decoupling? Through decoupling any evolution/extension in requirement can be dealt with code change at only one place rather than at many places. So the key idea is to avoid changes at too many places. For eg. : creation of concrete objects should be decoupled from the client code. A client is a piece of code that uses other piece of code.

In OOP, there are three phases:

1. Class definition ( Structural Pattern)
2. Object creation (Creational Pattern)
3. Interaction/communication between objects at runtime (Behavioral Pattern)

Structural Pattern:

Relationship b/w classes can be

1. IS A : Parent-child relationship. Inheritance.
2. HAS A : Composition.

How good a soft architecture is can be seen by how efficiently it adapts to a change/evolution in requirement. If it requires modifications at too many places, it’s a bad architecture.

RIP pattern : Replace If With Polymorphism

Lazy/eager loading

Factory Pattern : centralized object creation

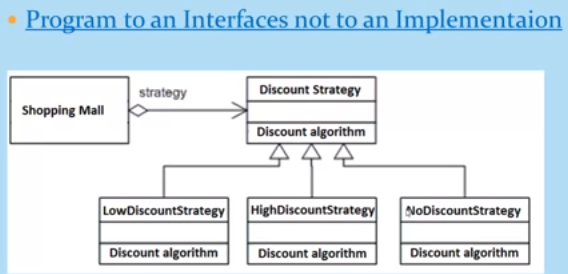
Strategy Pattern :

1. Define a family of algorithms, encapsulate each one and make them interchangeable. Strategy lets algorithm vary independently from clients that use it. It solves two problems :

Enforces Open-close principle: Software entities should be open for extension but closed for modification.

Makes different algorithms re-usable.

Program to an interface not to an implementation.



Shopping mall class has a reference to the discount strategy which is an interface(polymorphism !!). The concrete strategy is passed to this class by the client code at run time. This way the client code is not tied up to a particular strategy but can choose from all available strategies without changes at any other place.



1. Command Pattern :

Command decouples the object that invokes the operation from the one that knows how to perform it. The Command design pattern encapsulates commands (method calls) in objects allowing us to issue requests without knowing the requested operation or the requesting object. Command design pattern provides the options to queue commands, undo/redo actions and other manipulations. Allows us to save request/action into a class or an object so that we can use it at later time and do operations on it. This is extensively used in editors and games where we have to revert back to previous state based on last action performed.

Intent

- encapsulate a request in an object

- allows the parameterization of clients with different requests

- allows saving the requests in a queue

Client uses an invoker object. Invoker contains command interface. Actual command(concrete implementation) is chosen by the client and passed to invoker based on action taken at run time.

Implement undo-redo using command pattern : <http://gernotklingler.com/blog/implementing-undoredo-with-the-command-pattern/>

Example : in game design where undoing/redoing a move is normal, In TV switch on/off, channel change, volume up/down,

Observer Design Pattern:

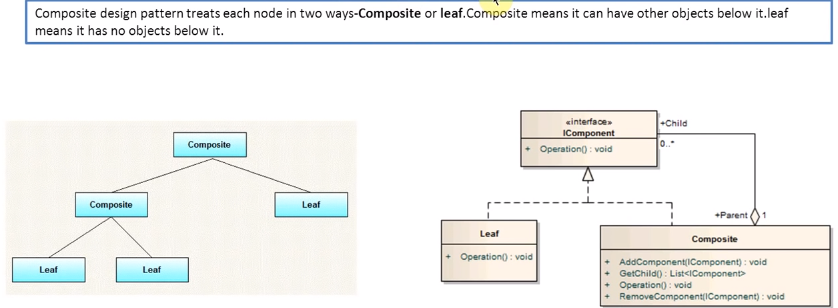
Also called publisher – subscriber pattern.

Example : we subscribe for notifications for news updates, new product releases from a company, software update, when we go to purchase an item online and its not available then we subscribe for notification when its availability.

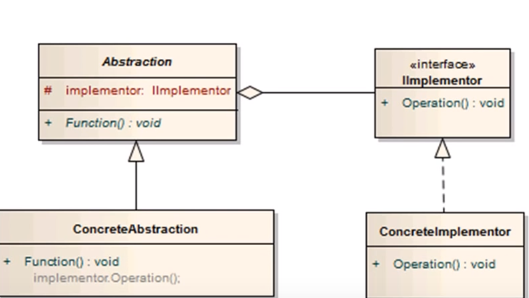
<https://www.youtube.com/watch?v=Ocqj4vyzeQQ>

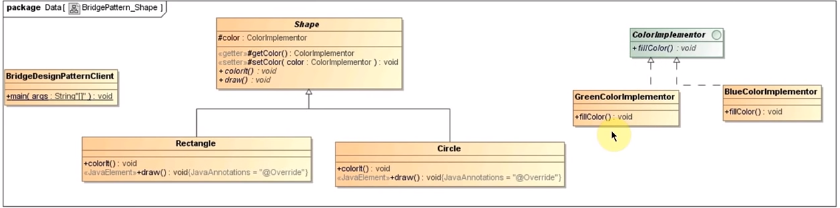
**Structural Patterns :**

1. **Composite Pattern :** Used where we need to create a group of objects in a similar way as a single object. This pattern composes objects in terms of a tree structure to represent *part-whole hierarch*y. Each node in the tree performs a task. Each node can either be – *Composite or leaf.* A composite has other nodes below it while a leaf has none.

****

1. **Builder pattern :**
2. **Adapter pattern :** An adapter makes two incompatible interfaces compatible or talk to each other. There are 2 kinds of adapters – object adapter(uses composition) and class adaptor(uses inheritance).
3. **Bridge pattern :** allows abstraction layer and implementation layer to be developed independently without affecting each other’s functionality.

****

****