[1- Abstract Factory: 2](#_Toc534281188)

[2- Builder: 2](#_Toc534281189)

[3- Decorator: 3](#_Toc534281190)

[4- Factory: 3](#_Toc534281191)

[5- Observer: 3](#_Toc534281192)

[6- Prototype: 3](#_Toc534281193)

[7- Singleton: 3](#_Toc534281194)

[8- Strategy: 4](#_Toc534281195)

[9- Adapter: 4](#_Toc534281196)

[10- Bridge: 5](#_Toc534281197)

[11- Composite: 5](#_Toc534281198)

[12- Decorator: 5](#_Toc534281199)

[13- Façade: 5](#_Toc534281200)

[14- Flyweight: 6](#_Toc534281201)

[15- Proxy: 6](#_Toc534281202)

[16- Chain of responsibility: 6](#_Toc534281203)

[17- Command: 7](#_Toc534281204)

[18- Interpreter: 7](#_Toc534281205)

[19- Iterator: 7](#_Toc534281206)

[20- Mediator: 7](#_Toc534281207)

[21- Memento: 8](#_Toc534281208)

[22- State: 8](#_Toc534281209)

[23- Template: 8](#_Toc534281210)

[24- Visitor: 8](#_Toc534281211)

# Abstract Factory:

Define an abstract class for creating families of related objects but without specifying their concrete sub-class;

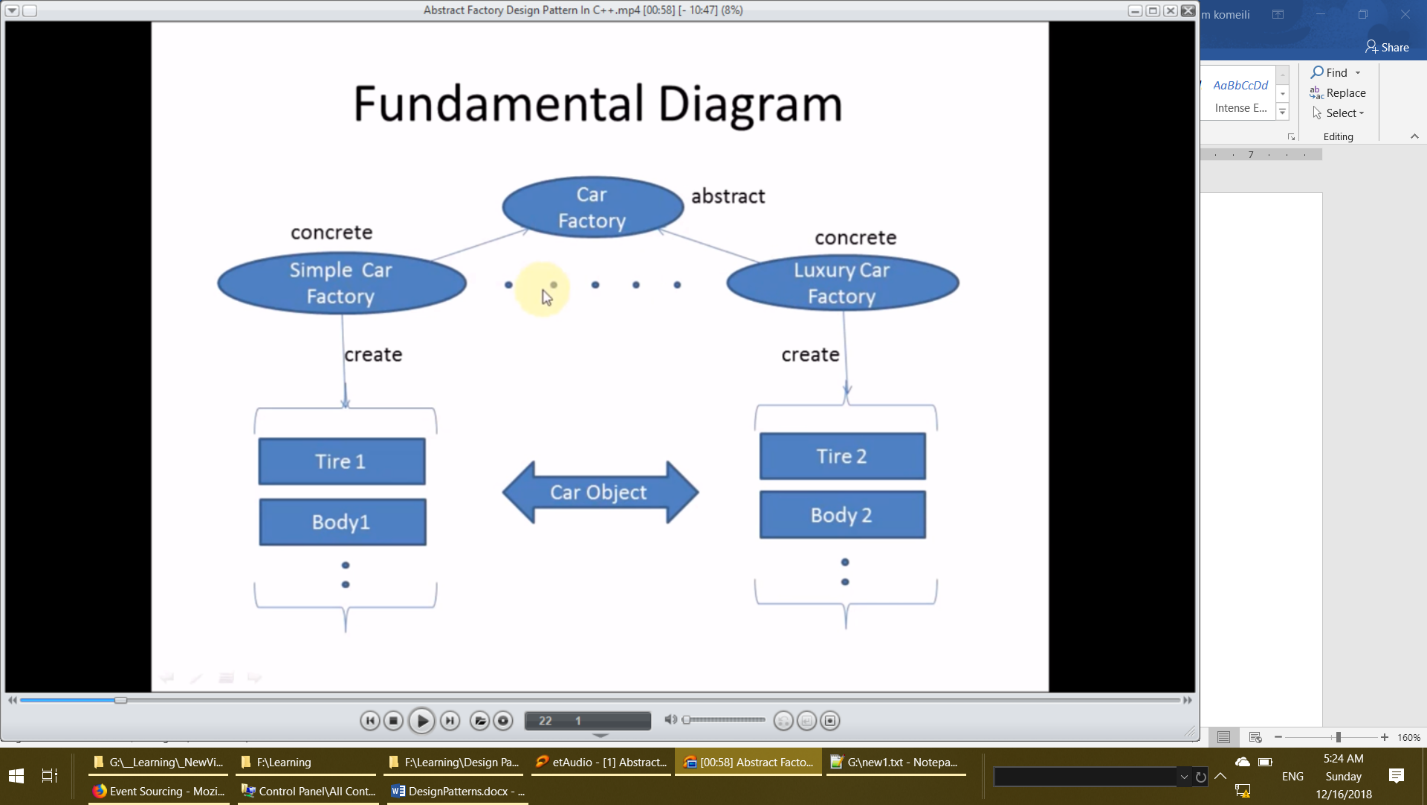
Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

## Problem:

Instantiation of families of related objects.

## Solution:

* Come up with a way to create the family of objects.
* Separate the instantiation of objects from use.



## Why/when use:

* You need system to be independent of how objects are created. Compose and represented.
* Show interface not implementation
* System need to be configured with one of the multiple family of objects.

# Builder:

Separate the construction of a complex object from its representation so that the same construction process can create different representation

## Problem:

Construction of complex object which requires different representation

## Solution:

* Come up with a way to create complex objects in steps.
* Construction process should be separated from representation of object
* Provide a way for different representation of object.

# Decorator:

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub classing for extending functionality.

## Problem:

Need to add additional functionality to object. Sub classing makes it inflexible and complex.

## Solution:

Give the responsibility of adding additional functionality to another object without using sub classing.

# Factory:

Define an interface for creating an object, but let sub classes decide which class to instantiate.

Factory method let’s a class defer instantiation to subclass.

## Problem:

A class has to instantiate subclass of another class but does not work which one.

## Solution:

Redefine a method in derived class which will decide which subclass to instantiate.

# Observer:

Define a one – to – many dependencies between objects so that when one object changes state, all its dependents are notified and updated automatically.

## Problem:

Number of objects are dependent another object and need to be notified for any changes to update themselves.

## Solution:

Give the responsibility of the monitoring the changes to changes to separate object which should notify to all dependent objects.

# Prototype:

* Creating an object is expensive than copy an object.
* All objects initial state in common and takes time to build.
* Composition, creation and representation of objects should be decoupled from the system.
* Hide the complexity of creating new instance from the user.
* Which classes to create are specified at runtime.

# Singleton:

Define a class that has only one instance and provide a global point of access that instance.

## Problem:

Application requires only one instance of the class and that instance of class should be accessible to multiple clients.

## Requirements:

* One and only one instance
* Global access
* No ownership
* Lazy initialization

## Advantage:

* Save memory
  + Only one is requires so why to create so many.
* Single access point
  + Logger, database connection
* Flexibility
  + Change any time you want to.

## Usage:

* Multithread
  + Thread pool
* Database application
  + logging
* Configuration setting
  + Game setting, application setting, etc.

# Strategy:

Define a family of algorithms, encapsulate each one, and make theme interchangeable. Strategy lets the algorithm vary independently from clients that use it.

## Problem:

Clients need to select different algorithm based on request or data. Also, it is difficult to add new algorithm if implementation is part of client.

## Solution:

Make the selection of algorithm separate from implementation and encapsulate the implementation of different algorithms in separate objects.

# Adapter:

Convert the interface of class to another interface clients expect. Adapter let’s classes work together that couldn’t otherwise because of incompatible interface.

## Problem:

Want to use existing system/component but the current system used by client doesn’t have interface that are compatible with existing one.

## Solution:

Come up with a class which adapts the interface of existing system/component to the new one which client expects.

# Bridge:

Decouple an abstraction from its implementation so that the two can vary independent.

## Problem:

Implementation classes driver from abstract class makes difficult to modify, extend and adding of new implementation.

## Solution:

Come up with interfaces for implementation

Derived the implementation classes from abstract implementation class

Use the interface of implementation in concrete class of abstract class

# Composite:

Compose object into tree structures to represent part-whole hierarchies. Composite let’s clients treat individual objects and composition of objects uniformly.

## Problem:

The primitive and composite objects have to be handled differently. To distinguish these objects for manipulation increases the complexity.

## Solution:

Come up with a way to manipulate the primitive and composite objects uniformly.

# Façade:

Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the sub system easier to use.

## Problem:

It’s difficult for client to use the sub system with existing interfaces.

## Solution:

Come up with new simplified interface for client to use the system easily.

# Flyweight:

Use sharing to support large numbers of fine-grained objects efficiently.

## Problem:

Having large number of objects to the granularity of system makes the system complex and inefficient.

## Solution:

Come up with an approach to share the objects at fine granularity to make the system simple and efficient.

# Proxy:

Provide a surrogate or placeholder for another object to control access to it.

## Problem:

Need to defer the instantiation of object until it is required.

## Solution:

Come up with another proxy class which acts for real class and instantiate the object of real class when required.

# 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.

## Problem:

Sender object does not know which object is going to handle the request.

## Solution:

Have multiple objects to handle the request of sender object .

# Command:

Encapsulate a request as an object thereby letting you parametrized clients with different requests, queue or log requests and support undoable an operation.

## Problem:

It is required to issue the requests to objects. The requested operation and the receiver are not known.

## Solution:

Come up with an object for requested operation, so that it can be passed as any other object.

# Interpreter:

Given a language, define a representation for its Grammar along with an interpreter that uses the representation to interpret sentences in the language.

## Problem:

A set of defined problem is coming again and again.

## Solution:

Define the problem in sentences of language and interpret these sentences to solve the problem.

# Iterator:

Need to access the elements of an aggregate object without exposing the internal details of the object.

## Problem:

Need to access the element of an aggregate object without exposing the internal details of the object.

## Solution:

Provide a way to access and traverse an aggregate object by giving responsibility of access and traversal to another object.

# Mediator:

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping object from referring to each other explicitly. And it let you vary their interaction independently.

## Problem:

There is too much communication between objects which is increasing the complexity and making it difficult to reuse the objects.

## Solution:

Encapsulate the behavior of interaction and coordination in a separate object.

# Memento:

Without violating encapsulation, capture and externalize an object’s internal state so that the object can be restored to this state later.

## Problem:

Need to save state an object to restore it later. Saving it externally is difficult as it violates encapsulation.

## Solution:

Store the state of the object in another object and make it accessible from the object whose state is saved.

# State:

Allow An object to alter its behavior when its internal state Changes. The object will appear to change its class.

## Problem:

Object has different states and has to respond differently based its one current state.

## Solution:

Have a separate abstract class for state, drive concrete class for each state and do the operation as required in overridden methods.

# Template:

Define the skeleton of an algorithm in an operation, deferring some steps to sub classes. Template method let’s subclasses redefine certain steps of an algorithm without changing the algorithm structure.

## Problem:

The algorithm has to be generic but some steps are required to be defined by the user.

## Solution:

* Identify the invariant and variant part of the algorithm.
* Have the implementation of invariant part in abstract class and variant part in derived class.

# Visitor:

Represent an operation to be performed on the elements of an object structure. Visitor let’s you define a new operation without changing the classes of the elements on which it operates.

## Problem:

An object structure contains many objects. There is need to perform different operations on these objects. Doing these operations in their classes will add complexity.

## Solution:

Have a separate object for operations of an element of object structure. Call the appropriate concrete object operation of that element when you traverse the elements.