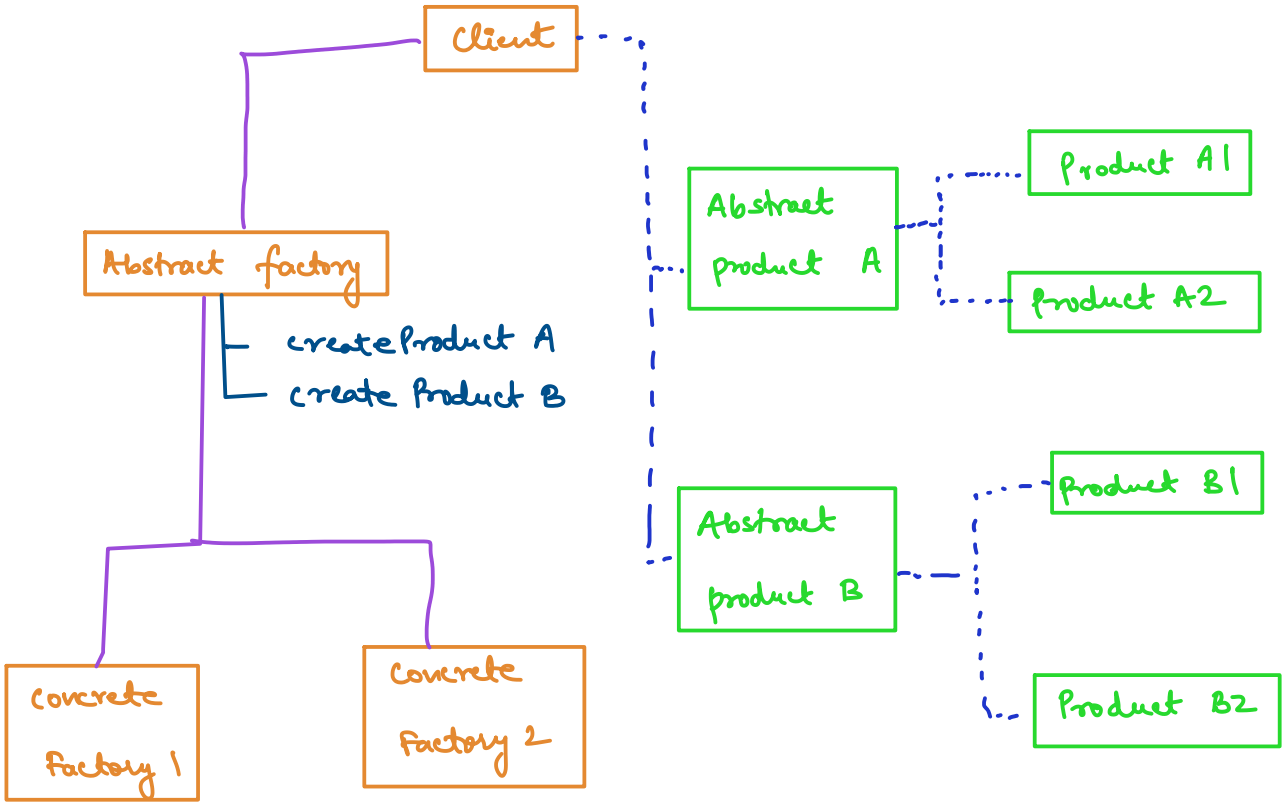
# 14__27-08-2022

Saturday, 27 August 2022 8:02 PM

## Abstract factory design pattern

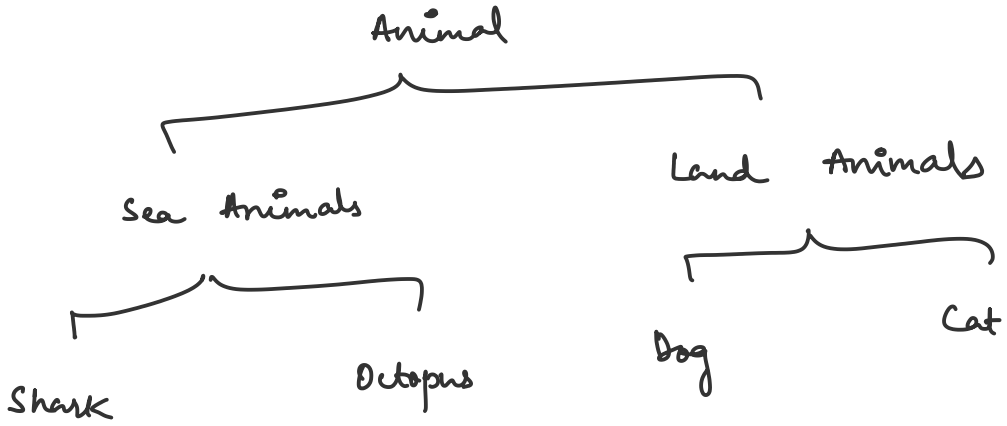



## Advantages of abstract factory design pattern

1. Abstract factory pattern isolates the client code from the concrete class.

↓

implementation classes.




2. It eases the exchanging of object families.

3. It promotes consistency among objects.

4. It enables loose coupling.

5. It provides another level of abstraction.

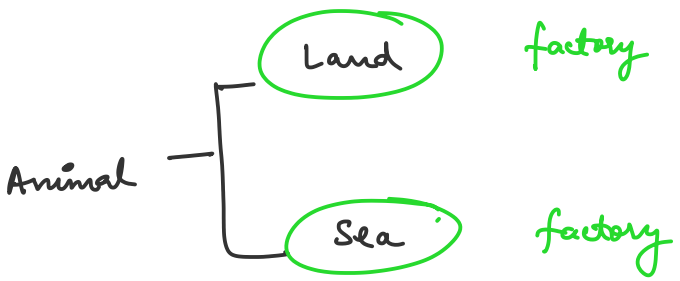
## Usages of abstract factory design pattern

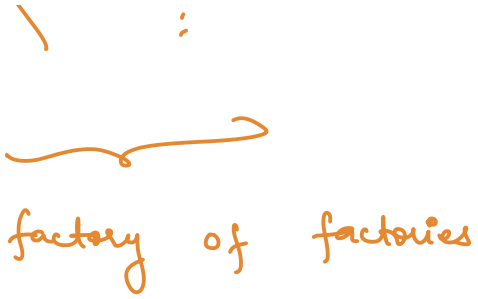
1. When the system needs to be configured with one of a multiple family of objects.

2. When the system needs to be independent of how its objects are created, composed and represented.

3. When the family of related objects has to be used together then this design pattern should be used.

4. When you want to provide a library of objects that does not show implementations and we want to achieve abstraction.

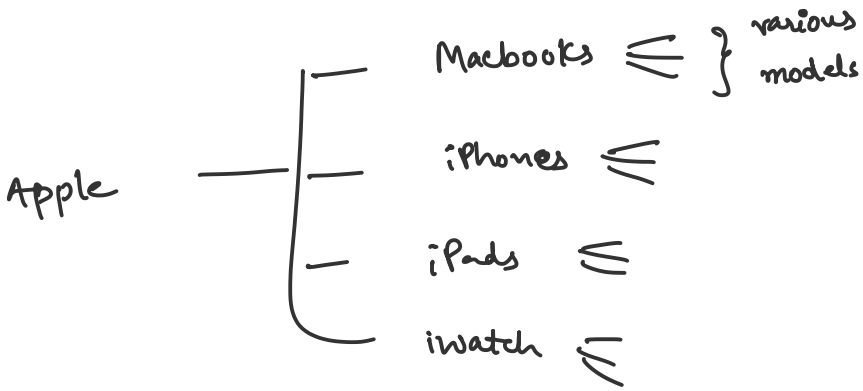



## Abstract factory vs factory method

1. Abstract factory patten adds a layer of abstraction to the factory method pattern.

2. Abstract factory patten implementation can have multiple factory methods.

3. 


similar products of a factory implementation

are grouped in abstract factory.

Creational design pattern

↓

creating objects

↳ singleton

↳ 1 copy of object

↳ factory

↳ multiple objects of same type

↳ Abstract factory

↳ use when we have sub-categories available and it is used when we want multiple objects of the same type.

Builder design pattern.

- what it is ?
- Implementation Guidelines
- Builder vs factory & abstract factory

Builder design pattern

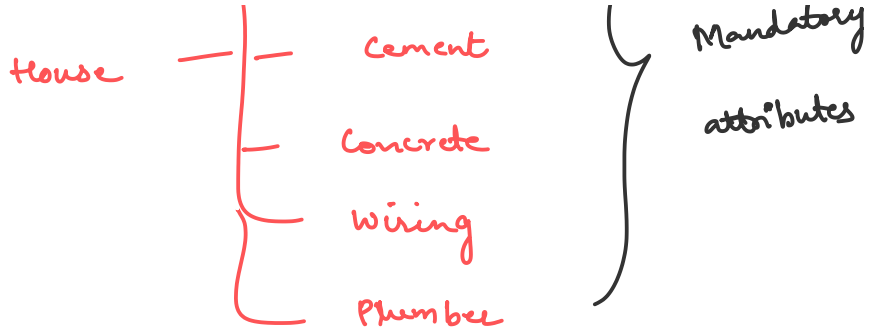
↳ use it when we have complex object.

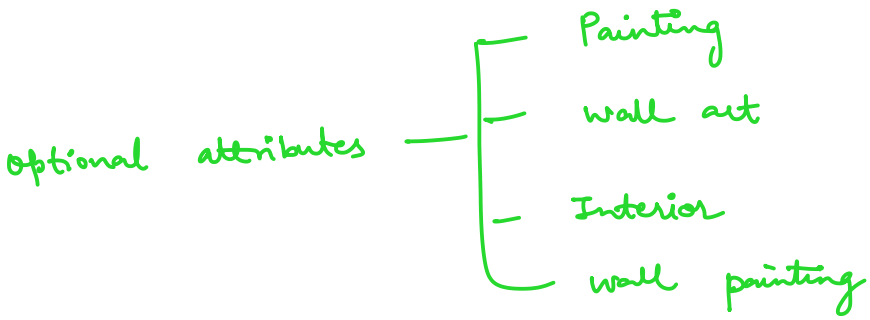
complex object — having many attributes

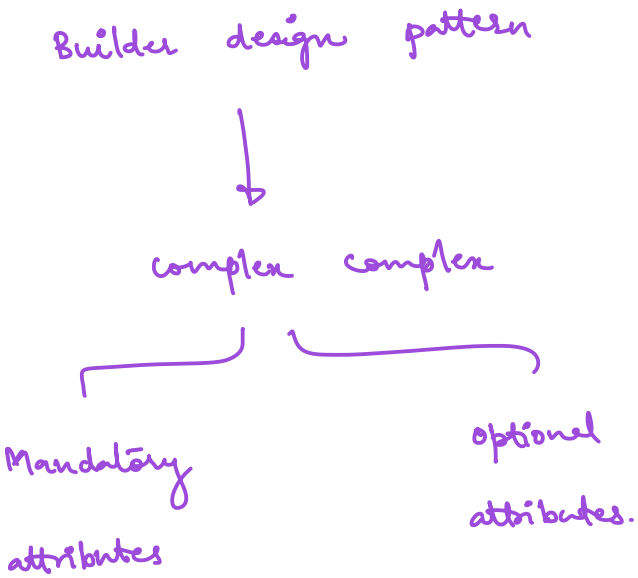
- Mandatory attributes
- optional attributes

Builder — who builds buildings | apartments

- Bricks




According to gang of four -

Separate the construction of a complex

object from its represe...

same construction process can create

different representations.

Builder patteen solves the situation of

increasing constructor parameters and

constructors of a given class by providing

a step by step initialization of parameters.

After step by step initialization, it returns

the resulting constructed object at once.

Implementation guidelines -

When should we use builder design

patteen -

... need to break the construction of

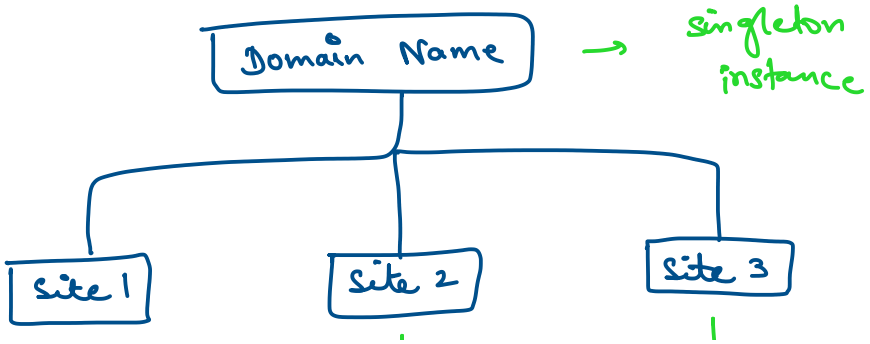
1. we ...

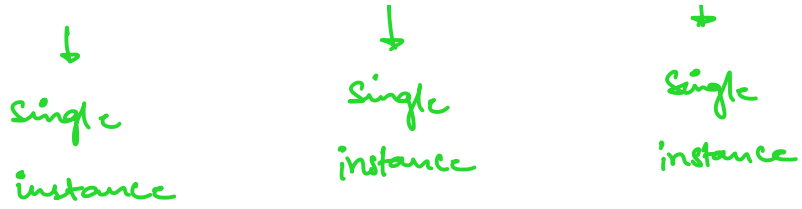
a complex object.

2. we need to create a complex object and it should be independent of the parts that make up the object.

3. The construction process must allow multiple representations of the same class.

Multi- tenant based appⁿ



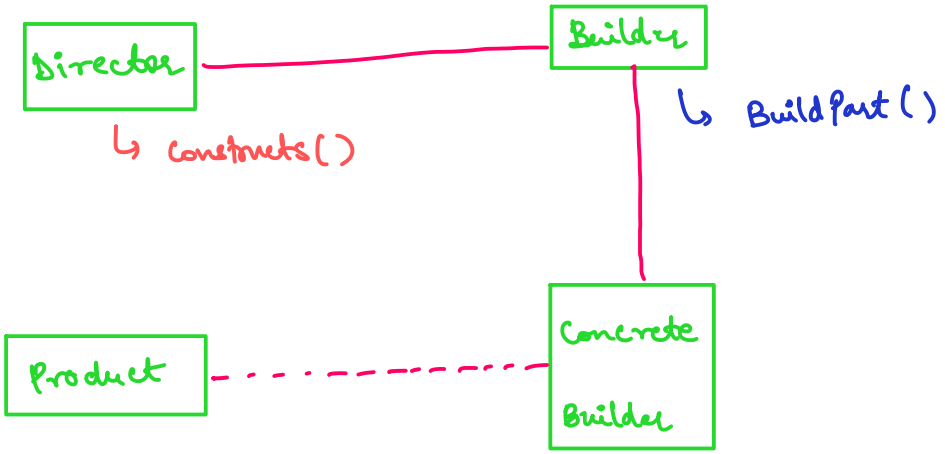



Singleton pattern will have one object, but in multi-tenant it will be common for all.

We need a single instance of the concrete class common to a virtual site whereas singleton instance is common to all the virtual sites.

This design is different from the actual meaning of singleton. Using it, we cannot create a singleton instance.

## Builder representation




### Director

↳ Responsible to construct the object using builder interface.

### Builder

↳ Defines a template (skeleton) for the steps to construct the product.

Builder specifies abstract interface

product

for creating parts of

object.

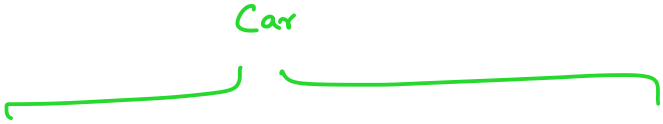
Concrete Builder

↳ Implements the builder interface and provides an interface for getting the product.

Product

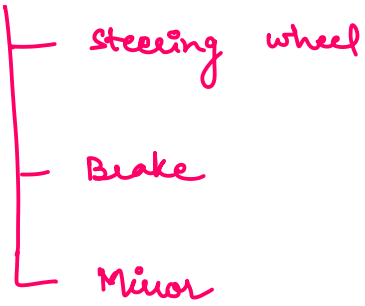
↳ Product is the main object that is constructed and represented.

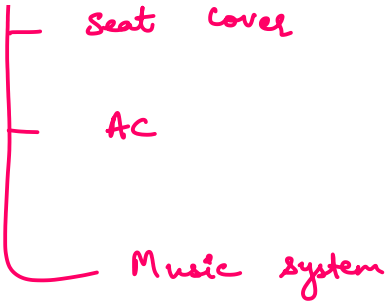
This is the complex object.




Mandatory Attributes

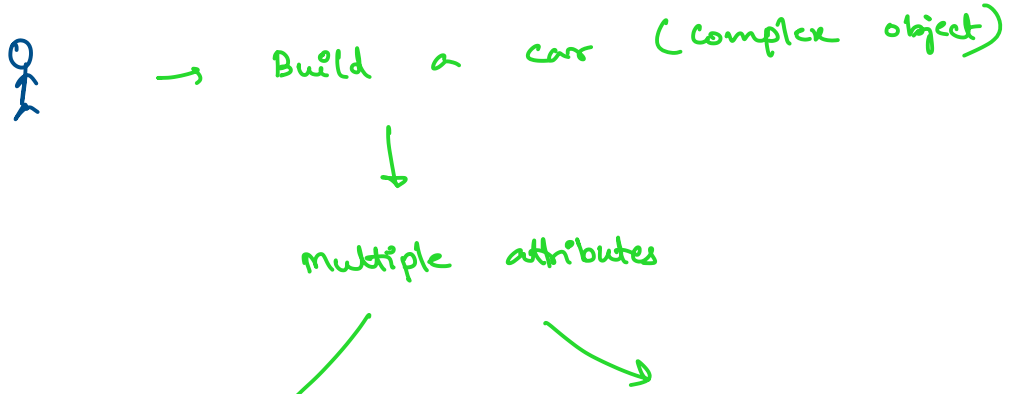
Optional Attributes




```
public class Car
{
    ===

    public class carBuilder  → Static class
    {
        ===
    }
}
```



Mandatory optional.

1. Create car class.

2. Specify the mandatory and optional attributes.

3. Define inner static car builder class. Here we will be using same optional and mandatory attributes.

4. For mandatory attributes
   ↓
   use a constructor to set these attribute values.

5. For optional attributes
   ↓

Define setter methods.

6. Create build method.

Is to build a car object

```
public class Car {
   //Mandatory attributes
   public String steeringWheel;
   public String mirror;
   public String brake;

   //Optional attributes
   public String seatCover;
   public String musicSystem;
   public String ac;

   //mention both mandatory and optional attributes
   public Car(CarBuilder carBuilder)
   {
      this.steeringWheel = carBuilder.steeringWheel;
      this.mirror = carBuilder.mirror;
      this.ac = carBuilder.ac;
      this.musicSystem = carBuilder.musicSystem;
      this.seatCover = carBuilder.seatCover;
      this.brake = carBuilder.brake;
   }

   @Override
   public String toString() {
      String str = "Car details : ";
      str += "steering wheel : " + steeringWheel;
      str += ", mirror : " + mirror;
      str += ", ac : " + ac;
      str += ", music system : " + musicSystem;
      str += ", seat cover : " + seatCover;
      str += ", brake : " + brake;
      return str;
   }

   public static class CarBuilder
   {
```

```
        //Mandatory attributes
        public String steeringWheel;
        public String mirror;
        public String brake;

        //Optional attributes
        public String seatCover;
        public String musicSystem;
        public String ac;

        //to set mandatory attributes
        public CarBuilder(String steeringWheel, String mirror, String brake)
        {
           this.steeringWheel = steeringWheel;
           this.mirror = mirror;
           this.brake = brake;
        }

        //create setters for optional attributes
        //for ac
        public CarBuilder setAc(String ac) {
           this.ac = ac;
           return this;
        }

        //for seat cover
        public CarBuilder setSeatCover(String seatCover)
        {
           this.seatCover = seatCover;
           return this;
        }

        //for music system
        public CarBuilder setMusicSystem(String musicSystem)
        {
           this.musicSystem = musicSystem;
           return this;
        }

        //provide build method
        //this method will be used for creating the object
        public Car build()
        {
           return new Car(this);
        }
     }
  }
```

```
public class Program {
   public static void main(String[] args) {
      Car car1 = new Car.CarBuilder("SW", "M", "B").build();
      System.out.println(car1);

      Car car2 = new Car.CarBuilder("SW", "M", "B").setAc("AC").build();
      System.out.println(car2);

      Car car3 = new Car.CarBuilder("SW", "M", "B").
            setAc("AC").setSeatCover("SC").setMusicSystem("MS").build();
      System.out.println(car3);
   }
}
```

Output:

```
Car details : steering wheel : SW, mirror : M, ac : null, music system : null, seat cover : null, bra
Car details : steering wheel : SW, mirror : M, ac : AC, music system : null, seat cover : null, bral
Car details : steering wheel : SW, mirror : M, ac : AC, music system : MS, seat cover : SC, brake
```