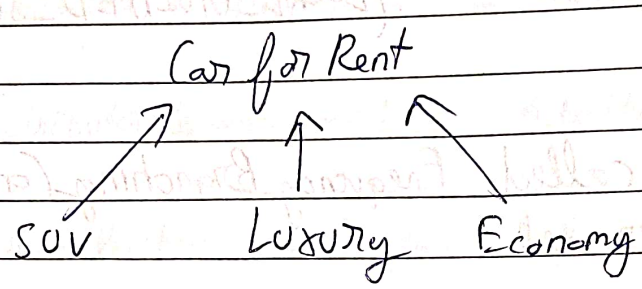
Lecture 3: Simple Factory, Factory Method & Abstract Factory Design

Factory Design Pattern:

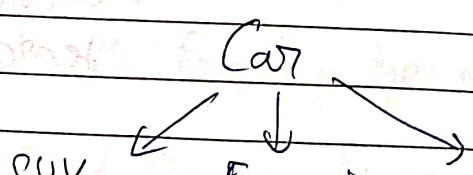



```
Factory{
  if (carType == SUV)
      createSUV();
  if (carType == Luxury)
      createLuxury();
  if (carType == EV)
      createEVC();
}
```

How is Factory Design Pattern following SOLID PRINCIPLES

* Single Responsibility Principle → We have assigned Factory class to only do the job of generating classes

Open Closed Principle → So any new carType comes, we need to just extend it and just add in Factory.



(Abstraction)

Client is unaware of Factory.

Suppose a new CarType is added to the Factory.

Client will not be knowing of the new classtype being Added.

(Liskov Substitution Principle) (Dependency Inversion)

In the client class.

```
class Client {
    CarType car;   → (this is an interface)
    Factory carfactory;

    car = carfactory.getCar(InputData);

}
```

relying on Interface and not on Concrete Classes.

Tip

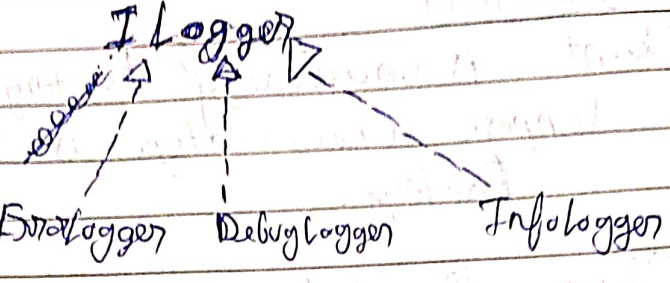
Inside the Factory class keep the method as (static)

So we don't have to initialize the Factory.

Just Do.

ClassName.getCar(input);

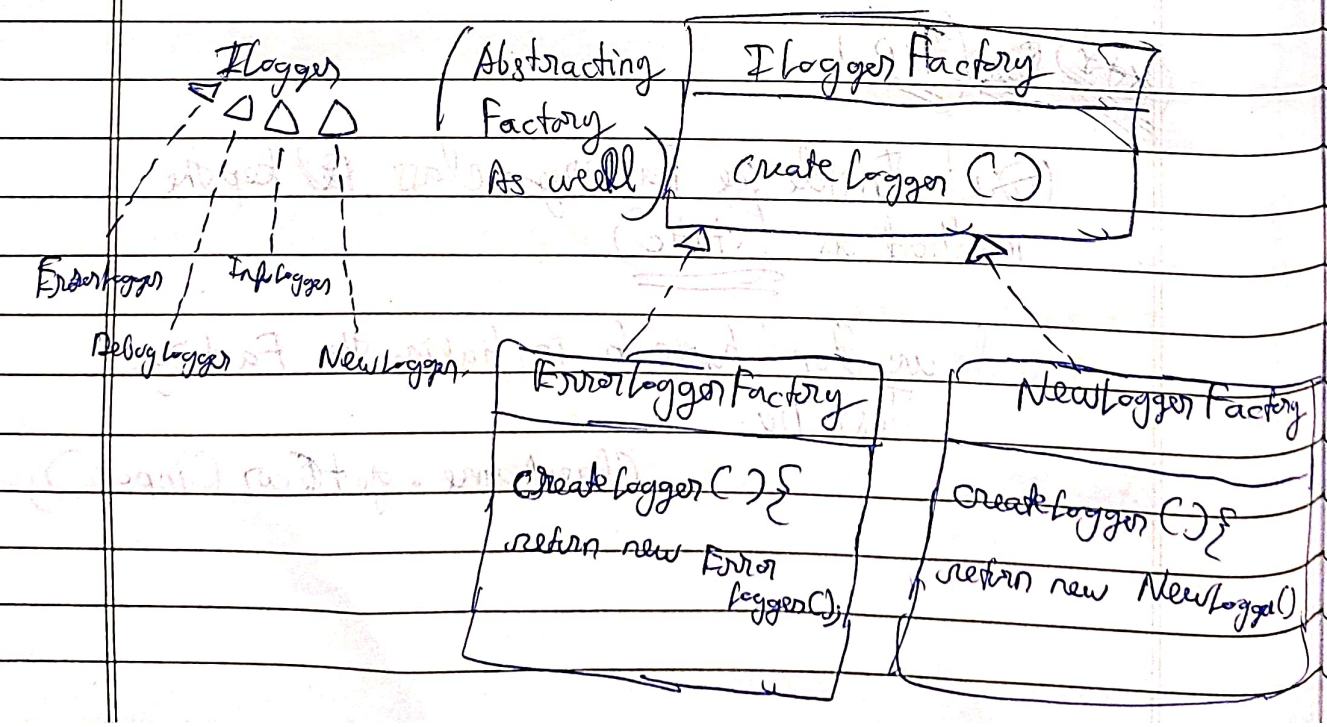
Example)




```
class LoggerFactory {

    public ILogger createLogger (Input){

        switch (input)
            Case "Error" : return new ErrorLogger()
            Case "DebugLogger": return new DebugLogger()
            Case "InfoLogger": return new InfoLogger()

    }

}
```

Now if we want to avoid making changes in the LoggerFactory to follow Open/Closed Principle.

We add Abstraction to LoggerFactory As well



The Implementation, which you just saw is Factory Method.

(**) We added an abstraction for Factory As well,

* Honestly I feel this is redundant. Because anyway we have to specify somewhere that which type of Logger we want and use its LoggerFactory.

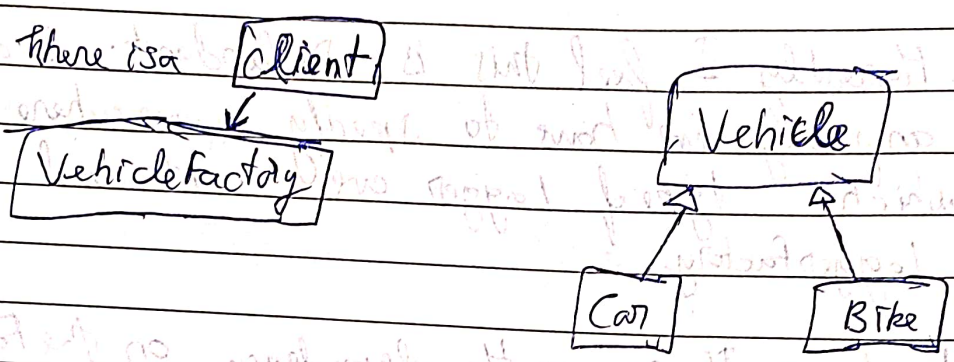
Advantage: It removes the dependency on the Factory class alone, we dont have to modify that code.

Argument can be given: If we wanted to remove this we could have had no Factory at all and just use new InfoLogger or rather new ErrorLogger as per our need.

Conclusion: During Interview go for Simple Factory and not Factory method. If interviewer asks, you can explain it, whatever you have understood.

Abstract Factory Design Pattern

Example

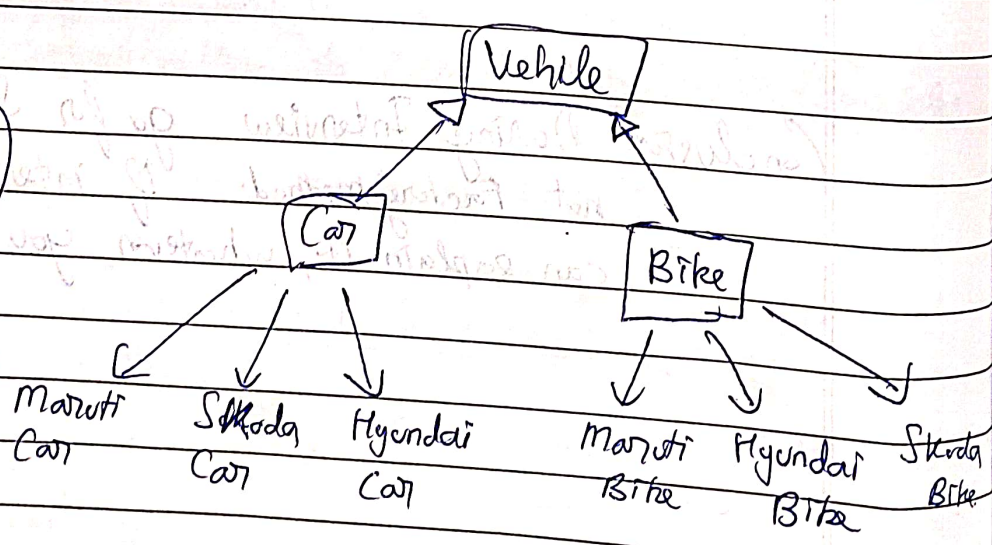


This is simple like client will ask you to either return a Car or a Bike and you implement it in the VehicleFactory and you are done.

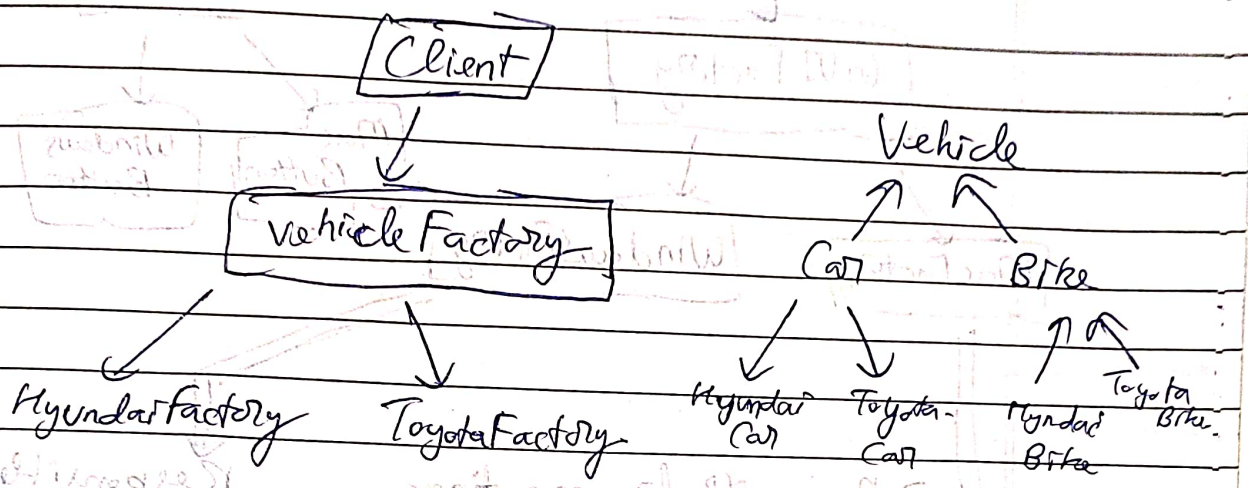
But Lets say we introduce Brand today.

Brand

Maruti
Skoda
Hyundai



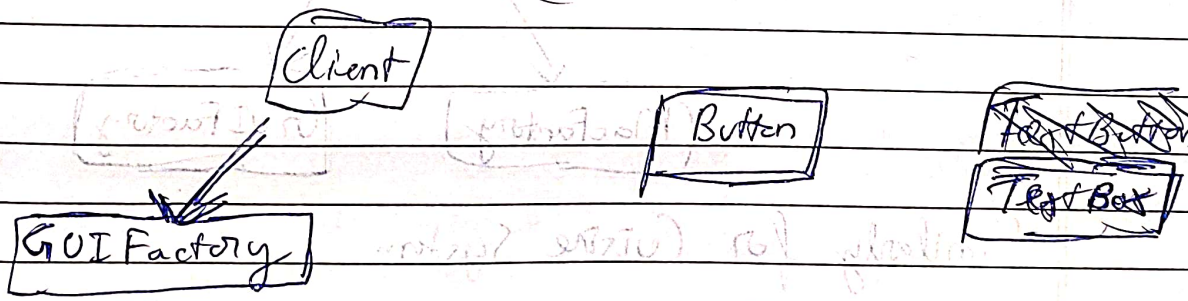
For this we will add extention of factories in vehicle Factory.



Hyundai Factory is responsible for creating → Hyundai Car
→ ~~Toyota Car~~ Hyundai Bike

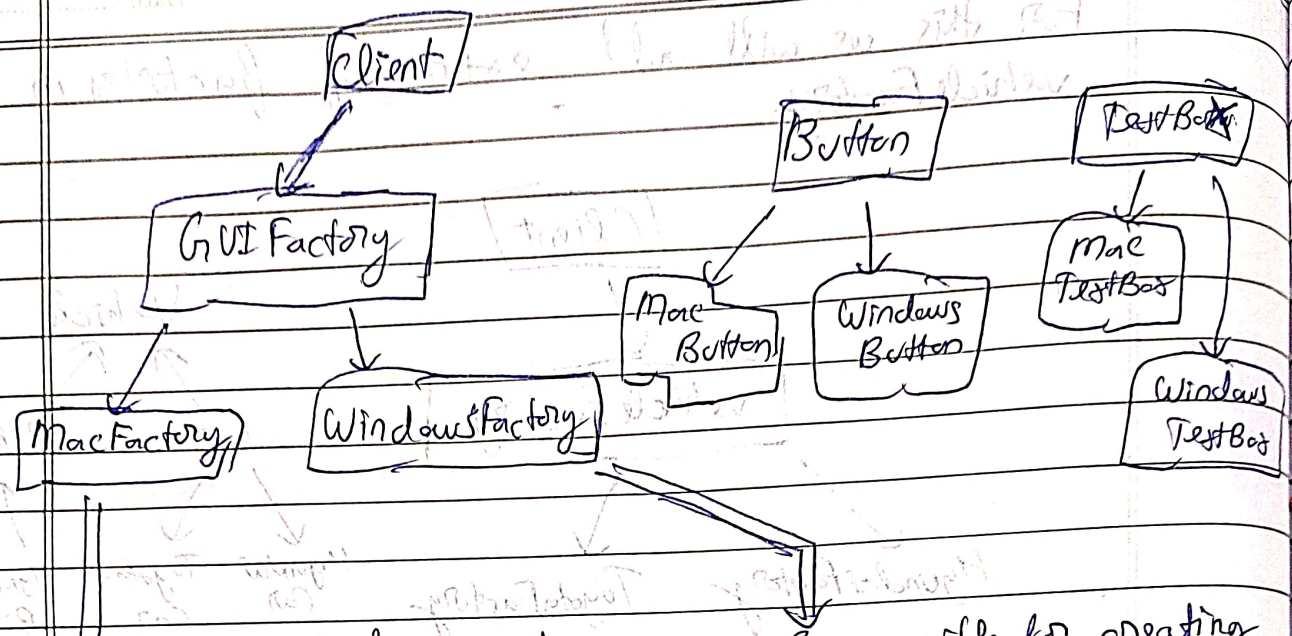
Toyota Factory is responsible for creating → ~~Hyundai~~
→ Toyota Car
→ Toyota Bike

Lets ~~take~~ an example. (UI Operation)



Client as per need asks for Button / Text Box and gets the required feature

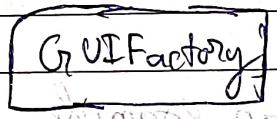
But Now demand may come for adding Window / Mac Button



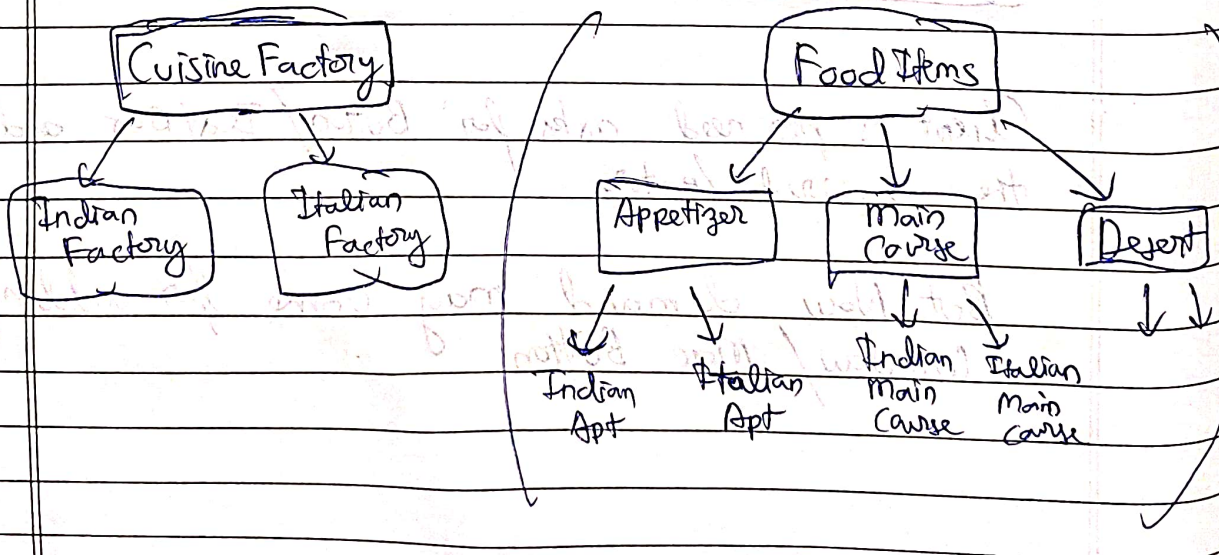
Responsible for creating Mac Button / Mac TestBox

Responsible for creating Windows Button / Windows TestBox

(Notice) We are actually creating Factories of ~~Factory~~ Factories here.

We created  who is responsible for creating

Similarly for Cuisine System

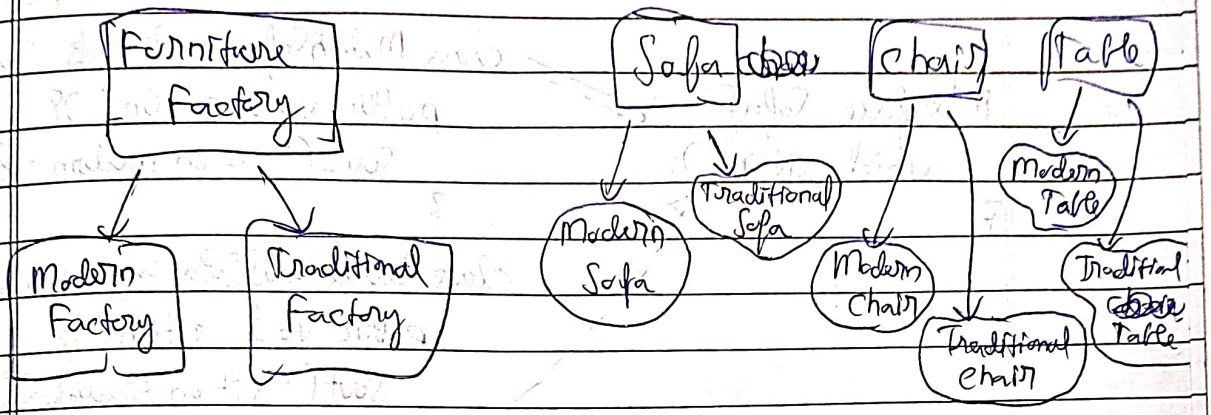


PAGE EDGE

Date :.....

(Code reuse)

(Furniture Example)



(UI example Code)

```
interface IFactory {
    IButton createButton();
    ITestBox createTestBox();
}

① class WinFactory implements IFactory {
    @Override
    public IButton createButton() {
        return new WindowsButton();
    }
    @Override
    public ITestBox createTestBox() {
        return new WindowsTestBox();
    }
}

② class MacFactory implements IFactory {
    @Override
    public IButton createButton() {
        return new MacButton();
    }
    @Override
    public ITestBox createTestBox() {
        return MacTestBox();
    }
}
```

```
interface IButton {
    void press();
}
interface ITestBox {
    void setText();
}
class MacButton implements IButton {
    @Override
    public void press() { }
}
class WindowsButton implements IButton {
    @Override
    public void press() { }
}
class WindowsTestBox implements ITestbox {
    @Override
    public void setText();
}
class MacTestbox implements ITestBox {
    @Override
    public void setText();
}
```

(Furniture Example & Code)

Homework: Think of more use-case of Factory / Abstract Factory

```
interface Sofa{
    void sitOn();
}

class ModernSofa implements Sofa{
    public void sitOn(){
        sout("sit on modern sofa");
    }
}

class TraditionalSofa implements Sofa{
    public void sitOn(){
        sout("sit on traditional Sofa");
    }
}
```

```
interface Chair{
    void sitOn();
}

class ModernChair implements Chair{
    public void sitOn(){
        sout("sit on Modern chair");
    }
}

class TraditionalChair implements Chair{
    public void sitOn(){
        sout("sit on traditional chair");
    }
}
```

```
interface Table{
    void place();
}

class ModernTable implements Table{
    public void place(){
        return ModernTable();
    }
}

class TraditionalTable implements Table{
    public void place(){
        sout("Place on traditional table");
    }
}
```

PAGE EDGE

Date :.....

The actual Factory which creates other Factories, the abstract Factory:

```
interface FurnitureFactory {

    Sofa  createSofa();
    Chair createChair();
    Table createTable();

    public static FurnitureFactory createFurnitureFactory(input) {

        ① → return new ModernFurnitureFactory();
        ② → return new TraditionalFurnitureFactory();

    }
}
```

Note: Interface can have concrete methods only if they are static

```
class ModernFurnitureFactory implements FurnitureFactory {

    public Sofa createSofa() {
        return new ModernSofa();
    }
    public Chair createChair() {
        return new ModernChair();
    }
    public Table createTable() {
        return new ModernTable();
    }
}
```

```
class TraditionalFurnitureFactory implements FurnitureFactory {

    public Sofa createSofa() {
        return new TraditionalSofa();
    }
    public Chair createChair() {
        return new TraditionalChair();
    }
    public Table createTable() {
        return new TraditionalTable();
    }
}
```

(Logger Example) → (Real Life Problem of Logger)

Lets say we want Console and File and Dashboard