17__04-09-2022

Sunday, 4 September 2022 8:05 PM

Builder design pattern

IOrder

↳ interface

↓

buildOrder()

↳ with this you will be creating your order

Order

- Mandatory attributes
  - Product Name
  - Price
- Optional attributes
  - wrapping

OrderBuilder → implements IOrder

```
public interface IOrder {
   public Order buildOrder();
}
```

```
public class Order {
   //Mandatory attributes
   private float price;
   private String name;

   //Optional attributes
   private String wrapping;

   public Order(OrderBuilder orderBuilder)
   {
      this.name = orderBuilder.name;
      this.price = orderBuilder.price;
      this.wrapping = orderBuilder.wrapping;
   }

   @Override
   public String toString() {
      return "Order{" +
            "price=" + price +
            ", name='" + name + '\'' +
            ", wrapping='" + wrapping + '\'' +
            '}';
   }

   public static class OrderBuilder implements IOrder {
      //Mandatory attributes
      private float price;
      private String name;

      //Optional attributes
      private String wrapping;

      public OrderBuilder(float price, String name)
      {
         this.name = name;
         this.price = price;
      }

      //for wrapping - optional attribute
      public OrderBuilder setWrapping(String wrapping)
```

```
        {
            this.wrapping = wrapping;
            return this;
        }

        @Override
        public Order buildOrder() {
            return new Order(this);
        }
    }
}
```

```
public class Program {
    public static void main(String[] args) {
        Order order1 = new Order.OrderBuilder(1000f, "Plant").buildOrder();
        System.out.println(order1);

        Order order2 = new Order.OrderBuilder(1500f, "Painting").
                setWrapping("Gift wrap costs Rs 30.").buildOrder();
        System.out.println(order2);
    }
}
```

```
Output:
Order{price=1000.0, name='Plant', wrapping='null'}
Order{price=1500.0, name='Painting', wrapping='Gift wrap costs Rs 30.'}
```

public class Product implements Iproduct

{

Provide setter for price and name.

}

Indoor Gift

public class Indoor Gift

{

→ Indoor Plant

→ Painting - Small

}

public class Outdoor Gift

{

→ outdoor plant

→ Big painting

}

Having mandatory attribute

public class Order

Have include

{

optional attributes

public class OrderBuilder

{

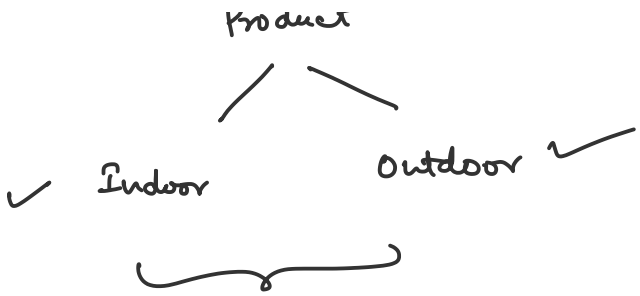
}

}

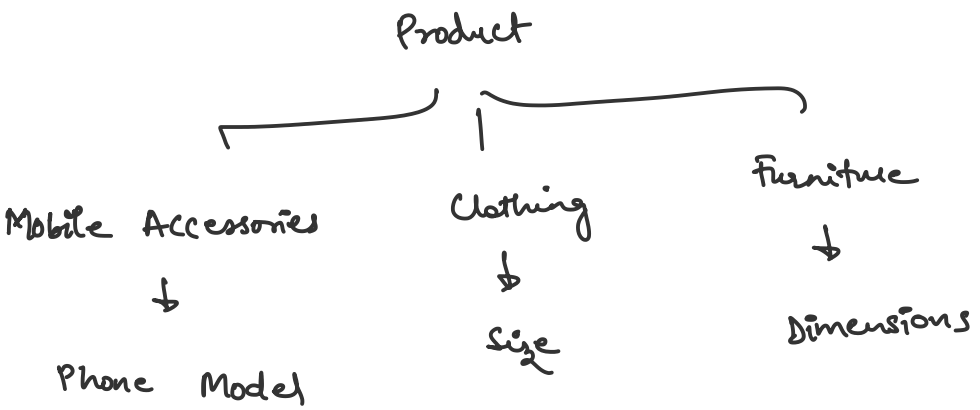
Client class

```
Product product = new Product();
product.setProductName("Painting");
product.setProductPrice(1000f);
```

```
Order order = new Order.
                OrderBuilder(product).
                buildGift();
```

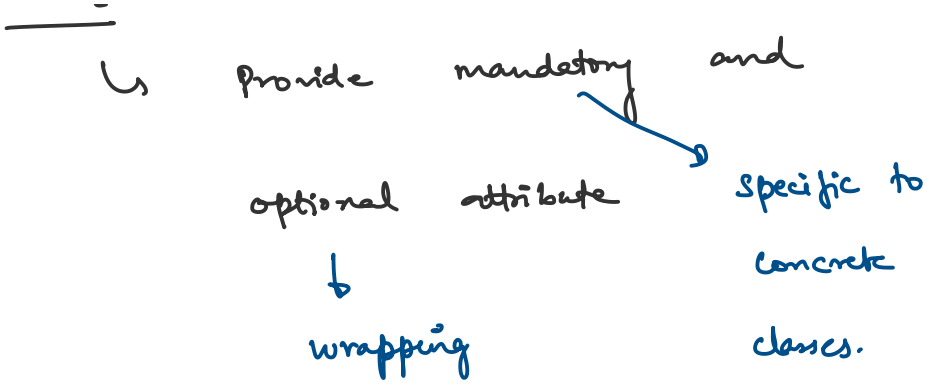


Amazon



Mandatory attributes are now going to be specific to concrete classes.

Order



Order

```
private List<IProduct> productList =
        new ArrayList<>();

public void addItem(IProduct product)
{
    productList.add(product);
}

public float getTotalCost()
{
    float cost = 0.0f;
```

```
        for (IProduct product : productList)
        {
            cost += product.getProductPrice();
        }
        return cost;
    }

public void showProducts()
{
    for (IProduct product : productList)
    {
        SOP (product.getProductName())
        SOP (product.getProductPrice())
    }
}
```

Order Builder

```
public Order PrepareIndoorCombo()
{
    Order order = new Order();
    order.addItem("Painting");
    Order.addItem("Indoor Plant");
}
```

If you have classes for these products then pass object for the same

↓

new IndoorPlant()

Builder vs factory and abstract factory

→ Builder design pattern is used when you have a complex object (having

mandatory and optional attributes) into a single method.

→ Builder design pattern focuses on construction of objects in a step by step manner whereas in factory and abstract factory we create it in single go.

→ Factory focuses on creating objects of same type and abstract factory focuses on creating objects of same family or category.

→ Builder design pattern returns the object after step by step construction

of the complex object whereas in abstract factory and factory pattern we return the created object immediately.

→ Appⁿ design complexity order —

factory < Abstract factory < Builder

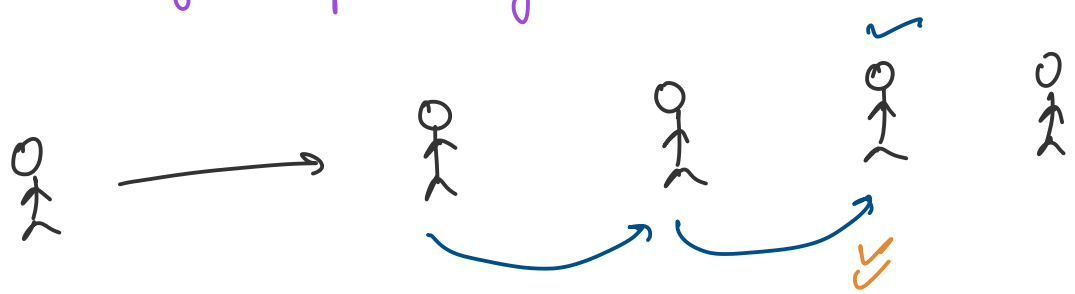
↓ ↓ ↓

less complex More complex

↓

flexibility

Chain of responsibility.



Observer design pattern

↳ used for asynchronous communication.

↓

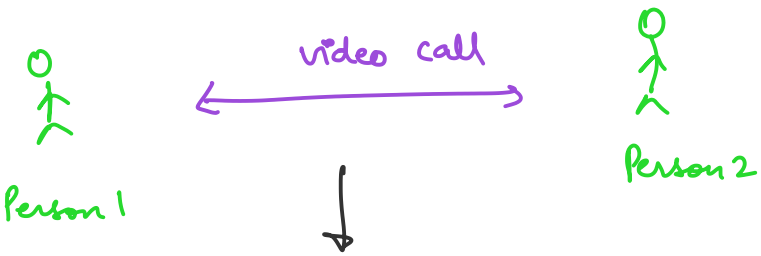
Messaging systems

Kafka, Rabbit MQ, AMQP

↓

Advanced message Queuing Protocol

Synchronous Comm^n



↓

You are just talking and not doing anything else.

↓

Synchronous comm^n

Blocking comm^n

Asynchronous comm^n

or

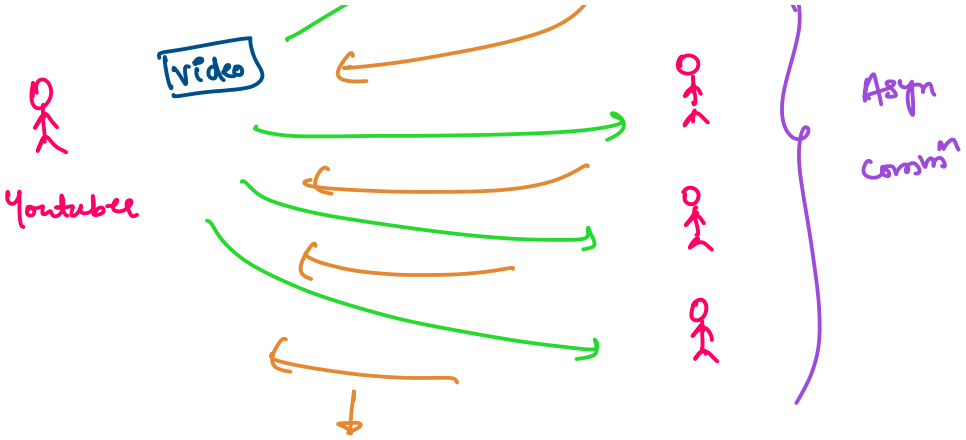
sending message

↓

Asyn comm^n or non-blocking comm^n because we are doing other things in parallel.

Another eg

YT sending notification

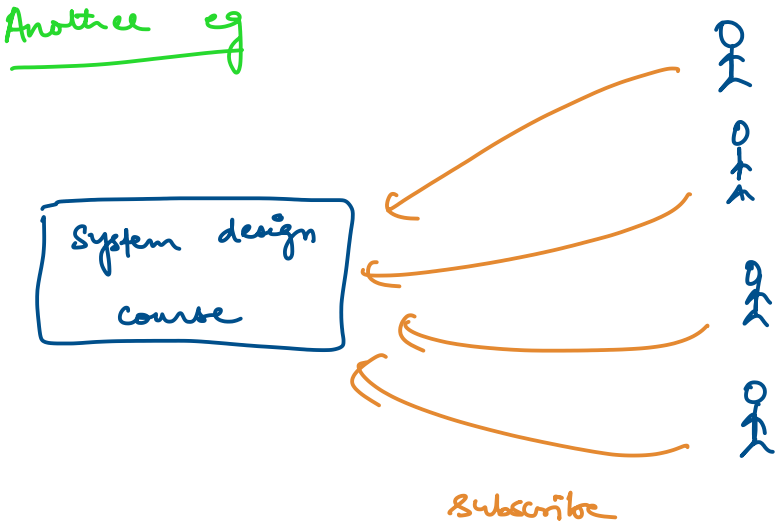


checking profile
if video has been
uploaded or not

Calling everyone to check the newly
uploaded video

↓

Synch comm^n | Blocking comm^n



2 models — Pull model, Push Model

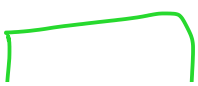
Pull

↳ All students will come and check if there is some class scheduled or not, so everyone is pulling the data every morning.

Push model

↳ This will be pushing notifications to all the subscribed students.

Amazon



↳ Availability

↳ only 2 left

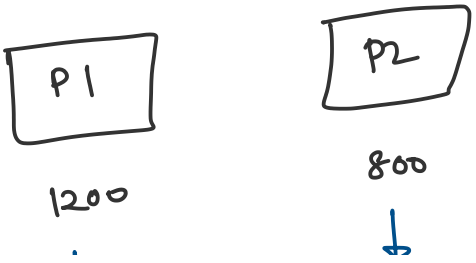
↳ out of stock

If out of stock then

Enter to email to get notified once it is back in the stock

Amazon

↳ Items in the cart



1300 → 500 ↓

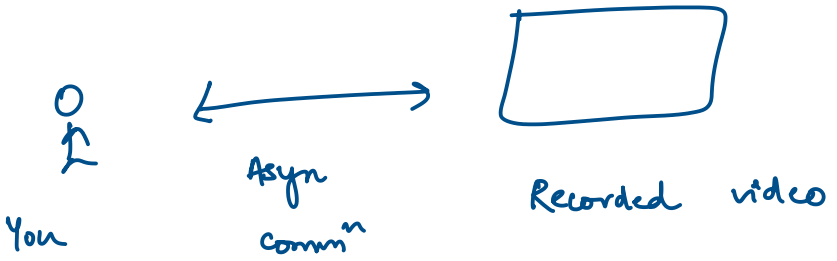
notified when you will open the cart.

Another eg -

on ticket booking apps you can set alerts for price drop.

Sync comm^n

Asyn comm^n



Definition

In this we maintain a list of its dependent (called as observers) and notifies them automatically whenever any state changes by calling one of their methods.

In this design patteen we have one-to-one dependency so that when one object changes state then all its dependents are notified and updated

automatically.

The observer design pattern is also known as dependents or publish-subscribe.