**Builder Design Pattern**

Builder Design Pattern is object creational design pattern, it used when too many arguments to send in constructor have & it’s hard to maintain the order. We don’t need to send all parameters in object initialisation. we send optional parameters as null and build() method that will return the final object.  
  
  
**Example**  
  
public User (String firstName, String lastName, int age, String phone, String address){  
this.firstName = firstName;  
this.lastName = lastName;  
this.age = age;  
this.phone = phone;  
this.address = address;  
}  
  
  
**Create the user1 object**  
  
User user1 = new User.UserBuilder("Lokesh", "Gupta")  
.age(30)  
.phone("1234567")  
.address("Fake address 1234")  
.build();  
  
System.out.println(user1);

**Create the user2 object**  
  
User user2 = new User.UserBuilder("Jack", "Reacher")  
.age(40)  
.phone("5655")  
//no address  
.build();

System.out.println(user2);

**OOPS Concepts in project**

1. **Encapsulation**

Hiding data behind method is central concept of encaptulation.

Ex :

public class SiteVisitDetailsDTO implements Serializable {

private static final long serialVersionUID = 1L;

private Long siteVisitPk;

private String applNo;

private LocalDateTime dateSiteVisit;

private String address1;

private String address2;

private String address3;

public Long getSiteVisitPk() {

return siteVisitPk;

}

public void setSiteVisitPk(Long siteVisitPk) {

this.siteVisitPk = siteVisitPk;

}

public String getApplNo() {

return applNo;

}

public void setApplNo(String applNo) {

this.applNo = applNo;

}

public String getAddress1() {

return address1;

}

public void setAddress1(String address1) {

this.address1 = address1;

}

}

1. **Polymorphism**

**Over Loading :**

we can use the same method name with different arguments.

Ex :

**public void updateBranchBuisnessDate(S1Date branchBuisnessDate) throws RemoteException {**

**}**

**public void updateBranchBuisnessDate(S1Date branchBuisnessDate,String drawerid) throws RemoteException {**

**}**

**Overriding :**

Whatever the parent has by default available to the child. If the child is not satisfied with parent class implementation, then child is allowed to override parent class implementation in its own way.

**public class MBBAccountInqValidator extends Validator {**

@Override

**public** **void** validate(Object formObj, HashMap map) **throws** ValidationException {

**}**

**}**

1. **Abstraction**

Abstraction is a process of hiding the implementation details from the user, and only providing the functionality will be providing to the user.

**Ex :**

public abstract class AbstractMBBJMSHandlerService implements MBBJMSHandlerService {

Abstract void jmsconnection();

}

1. **Inheritance**

Interface is a collection of abstract methods.

Interface cannot be instantiated.

**Ex :**

public interface CashInsightAPIService extends javax.xml.rpc.Service {

public java.lang.String getCashInsightAPIServiceHttpSoap11Endpoint();

}