**Builder**

* Builder pattern is a type of creational design pattern
* Any complex object can be built in a step-by-step approach
* Builder pattern can be used in cases where create an object involves many optional parameters and helps building the object n step-by-step approach
* Such complex objects cannot build by using Factory patterns because when there are lot Fields need to be set in an object, the order of all the fields need to remembered and making the object creation cumbersome
* Only downside to this pattern is the implementation involves more lines of code as Builder class need to be implemented on each type.

**Example Program**

public class Employee {

private String employeeId;

private String foodCard;

private String adminAccess;

private Employee(EmployeeBuilder employeeBuilder) {

this.employeeID = builder.employeeID;

this. foodCard = builder. foodCard;

this. adminAccess = builder. adminAccess;

}

public String getEmployeeId() {

return employeeId;

}

public String getFoodCard() {

return foodCard;

}

public String getAdminAccess() {

return adminAccess;

}

public static class EmployeeBuilder {

private String employeeId;

private String foodCard;

private String adminAccess;

public EmployeeBuilder (String employeeId) {

this.employeeID = builder.employeeID;

}

public EmployeeBuilder setFoodCard(String foodCard) {

this.foodCard = foodCard;

}

public EmployeeBuilder setAdminAccess(String adminAccess) {

this. adminAccess = adminAccess;

}

public Employee build() {

return new Employee(this);

}

}

}

**Composite**

* Composite design pattern is one of the structural design pattern
* Composite pattern is used where we need to treat a group of objects in similar way as a single object.
* This pattern creates a class that contains group of its own objects. This class provides ways to modify its group of same objects.
* Composite Pattern consists of Base Component, Leaf, Composite
* **Base Component** – Base component is the interface for all objects in the composition, client program uses base component to work with the objects in the composition.
* **Leaf** – Defines the behaviour for the elements in the composition. It is the building block for the composition and implements base component
* **Composite** – It consists of leaf elements and implements the operations in base component
* **Major advantage of this pattern is that the implementation becomes light and easy to understand**
* **Disadvantage is when there is change composite feature it is hard to maintain or add more features as code changes becomes cumbersome**

**Example Program**

public interface Shape {

public void draw(String fillColor);

}

public class Triangle implements Shape {

@Override

public void draw(String fillColor) {

System.out.println("Drawing Triangle with color "+fillColor);

}

}

public class Circle implements Shape {

@Override

public void draw(String fillColor) {

System.out.println("Drawing Circle with color "+fillColor);

}

}

public class Drawing implements Shape{

private List<Shape> shapes = new ArrayList<Shape>();

@Override

public void draw(String fillColor) {

for(Shape sh : shapes)

{

sh.draw(fillColor);

}

}

public void add(Shape s){

this.shapes.add(s);

}

public void remove(Shape s){

shapes.remove(s);

}

public void clear(){

System.out.println("Clearing all the shapes from drawing");

this.shapes.clear();

}

}

**State**

* State pattern is one of the Behavioral design patterns
* Used in cases where the behavior will change when the state of the object changes
* Provides systematic and loosely coupled way to achieve the behavior change based on state
* This pattern uses a state variable in the object. Using if-else condition different behavior can be performed depending on the state variable.
* Advantage of this pattern is it helps reducing the condition complexity.
* Disadvantage is large code needs to be written when there many states

**Example Program**

public interface State {

public void doAction();

}

public class TVStartState implements State {

@Override

public void doAction() {

System.out.println("TV is turned ON");

}

}

public class TVStopState implements State {

@Override

public void doAction() {

System.out.println("TV is turned OFF");

}

}

**Prototype**

* Prototype pattern is one of the creational design pattern
* Prototype pattern refers to creating duplicate object while keeping performance in mind
* Prototype pattern provides a mechanism to copy the original object to a new object and then modify it according to our needs.
* Prototype design pattern uses java cloning to copy the object.
* Advantage is it hides complexities of creating objects.
* Disadvantage is that it is overkill for a project that uses very few objects

**Example Program**

public class Employees implements Cloneable{

private List<String> empList;

public Employees(){

empList = new ArrayList<String>();

}

public Employees(List<String> list){

this.empList=list;

}

public void loadData(){

//read all employees from database and put into the list

empList.add("Pankaj");

empList.add("Raj");

empList.add("David");

empList.add("Lisa");

}

public List<String> getEmpList() {

return empList;

}

@Override

public Object clone() throws CloneNotSupportedException{

List<String> temp = new ArrayList<String>();

for(String s : this.getEmpList()){

temp.add(s);

}

return new Employees(temp);

}

}