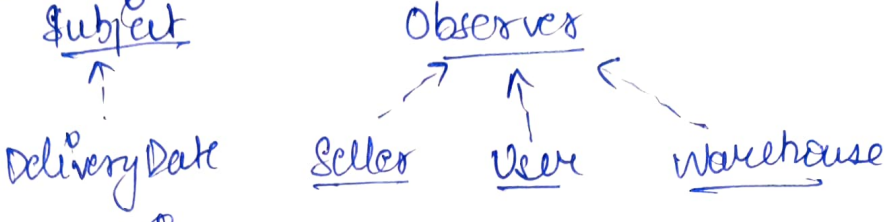
Observer Design Pattern

It is used when there is one-to many relationship b/w objects such as if one object is modified, its dependent objects are to be notified automatically.

Ex- Delivery Notification System



Example Implementation →

```
Subject
<<interface>>
{
 void register(observer obj);
 void unregister(observer obj);
 void notify();
}

Observer
<<interface>>
{
 void update(String location);
}

Seller implements Observer
{
 Pvt. String location;
 void update(String location)
 { this.location = location;
   showLocation();
 }
 void showLocation()
{ sop("Notification loc:" + location);
}
}

DeliveryDate implements Subject
{
 Pvt. List<Observer> observers;
 Pvt. String location;
 DeliveryDate(){
  this.observers = new ArrayList<>();
 }
 void register(Observer obj)
 { observers.add(obj); }
 void unregister(Observer obj)
 { observers.remove(obj); }
 void notify(){
  for(Observer obj : observers)
  { obj.update(location); }
 }
 void locationChanged()
 { this.location = getLocation();
   notify();
 }
 String getLocation()
 { return "Kolkata"; }
}
```

Similar classes for User and warehouse class →

User implements Observer
- String : location
+ update (String location) : void
+ showLocation ()
{ sop("Non" at User" + location);
}

warehouse implements Observer
- location : String
+ update (String) : void
+ showLocation () : void
{ sop("Non" at warehouse" + location);
}

```
class ObserverPatternTest
{  p s v main ()
   {  DeliveryData.topic = new DeliveryData ();
      Observer obj1 = new Seller ();
      Observer obj2 = new User ();
      Observer obj3 = new WareHouse ();
      topic.register (obj1);
      ( , obj2);
      ( , obj3);
      topic.locationChanged ();
      topic.unregister (obj3);
      topic.locationChanged ();
   }
}
```

State Design Pattern →

A class behaviour changes based on its state.
We create objects which represents various states and a context object whose behaviour varies as its state object changes.

Interface State
```
{ void doAction (context con);
}
```

StopState implements State
```
{ void doAction (context con)
  { sop ("stop State");
    con.setState (this);
  }
}
```

StartState implements State {
```
{ void doAction (context con)
  { sop ("Start State");
    con.setState (this);
  }
  public String toString () {
    return "Start State";
  }
}
public String toString () {
  return "Stop State";
}
```

Context

```
{ pvt State state;
  public Context()
  { state = null; }
  void setState(State st)
  { state = st; }
  State getState(){
    return state;
  }
}
```

class StatePatternDemo

```
{ p s v main()
  { Context con = new Context();
    StartState start = new StartState();
    start.doAction(con);
    sop(con.getState().toString());
    StopState stop = new StopState();
    stop.doAction(con);
    sop(con.getState().toString());
  }
}
```

Null Object Pattern

Instead of putting if check for null value, Null Object reflects a do Nothing relationship.

AbstractCustomer

```
protected String Name;
abstract boolean isNil();
abstract String getName();
```

RealCustomer imp. Abst

```
RealCustomer(String name)
{ this.name = name; }
p String getName(){
    return Name; }
p boolean isNil(){
    return false; }
```

NullCustomer imp. Abst

```
p String getName()
{ return "NA"; }
p boolean isNil()
{ return true; }
```

CustomerFactory {

```
p s f String[] names = {"..."};
p s AbstractCus getCust(name)
{ for(int i=0; i<names.length; i++)
  { if(names[i].equals(name))
    { return new RealCustomer(name); }
  }
  return NullCustomer();
}
```

NullPatternDemo {

```
p s v main()
{ AbstractCus c1 = new Abs("Bob");
  ——— c2 = ——— ("Robby");
  ——— c3 = ——— ("Laura");
  sop(c1.getName());
  ——(c2.———);
  ——(c3.———);
}
```

Template Design Pattern →

Just define the skeleton of a function in an operation, deferring some steps to its subclasses.

Example

abstract class Game

```
abstract void Initialize();
abstract void start();
abstract void end();
// template method.
final void play()
{  Initialize();
   start();
   end();
}
```

Cricket extends Game

```
void Initialize()
{ sop("Cricket Initialized"); }
void start()
{ sop("Cricket started"); }
void end()
{ sop("Cricket ended"); }
```

Football extends Game

similar to above

class TemplateDemo

```
{ P S V main()
  { Game game = new Cricket();
    game.play();
    game = new Football();
    game.play();
  }
}
```

Visitor Design Pattern

we use a visitor class which changes the executing algorithm of an element class.

ComputerPart

```
<<interface>>
void accept(Visitor visitor)
```

Keyboard imp. ComputerPart

```
void accept(Visitor visitor)
{ visitor.visit(this); }
```

Monitor implements ComputerPart()

```
void accept(Visitor visitor)
{ visitor.visit(this); }
```

Mouse implements ComputerPart{

```
void accept(Visitor visitor)
{ visitor.visit(this); }
```

Computer implements ComputerPart

```
ComputerPart[] parts;
p. Computer()
{ parts = new CompPart[] { new Mouse(), new Keyboard(), new Monitor
3.
p v accept(Visitor visitor){
    for(int i=0; i< parts.length; i++) {
        parts[i].accept(visitor);
    }
    visitor.visit(this);
}
}
```

~~ComputerPartVisitor~~

Visitor.

<<interface>>

```
void visit(Computer c);
void visit(Mouse m);
void visit(Keyboard k);
void visit(Monitor M);
```

DisplayVisitor implements Visitor

```
{ void visit(Computer c)
  { sop("Computer"); }
  void visit(Mouse m)
  { sop("Mouse"); }
  void visit(Keyboard k)
  { sop("keyboard"); }
  void visit(Monitor M)
  { sop("Monitor");
}
```

public class VisitorPatternDemo

```
{ p s v main()
  {
    ComputerPart computer = new Computer();
    computer.accept(new DisplayVisitor());
  }
}
```

Output

```
Displaying Mouse
_____ Keyboard
_____ Monitor
_____ Computer
```