Design Pattern

- Design patterns represent the best practices used by experienced object-oriented software developers.
- Design patterns are solutions to general problems that software developers faced during software development.
- These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.

What is Gang of Four (GOF)?

In 1994, four authors Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides published a book titled Design Patterns - Elements of Reusable Object-Oriented Software which initiated the concept of Design Pattern in Software development.

These authors are collectively known as Gang of Four (GOF). According to these authors design patterns are primarily based on the following principles of object orientated design.

- Program to an interface not an implementation
- Favor object composition over inheritance

Usage of Design Pattern

Design Patterns have two main usages in software development.

Common platform for developers

Design patterns provide a standard terminology and are specific to particular scenario. For example, a singleton design pattern signifies use of single object so all developers familiar with single design pattern will make use of single object and they can tell each other that program is following a singleton pattern.

Best Practices

Design patterns have been evolved over a long period of time and they provide best solutions to certain problems faced during software development. Learning these patterns helps inexperienced developers to learn software design in an easy and faster way.

Types of Design Patterns

As per the design pattern reference book Design Patterns -

Elements of Reusable Object-Oriented Software ,

there are 23 design patterns which can be classified in three categories: Creational, Structural and Behavioral patterns.

We'll also discuss another category of design pattern: J2EE design patterns.

Creational

- Talks about object creations.
- To make design more readable and less complexity.
- Talks about efficiently object creation.

Structural

- Talks about how classes and objects can be composed.
- Parents-child relationship like inheritance, extend.
- Deals with simplicity in identifying relationships.

Behavioral

- Deals with interactions and responsibility of objects.
- Deals with static, private, protected.

Types of Patterns

Pattern & Description
Creational Patterns These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator. This gives program more flexibility in deciding which objects need to be created for a given use case.
Structural Patterns These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.

Behavioral Patterns

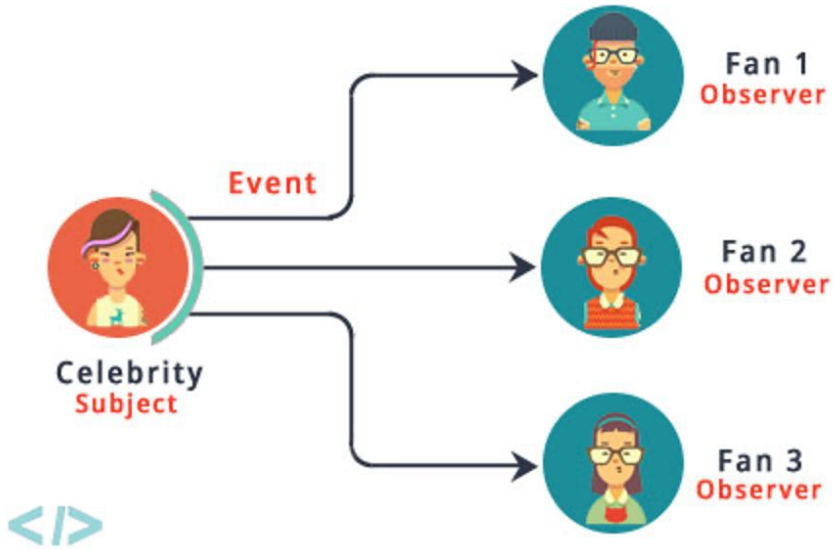
These design patterns are specifically concerned with communication between objects.

J2EE Patterns

These design patterns are specifically concerned with the presentation tier. These patterns are identified by Sun Java Center.

Observer Pattern

Intent: Define a one-to-many dependency between objects so that when oneobjectchange state, all its dependents are notified and updated automatically.



Student-Teacher Example

```
Class Teacher:
    def __init__(self):
        self.students=[]
    def attach_student(self, student):
        self.students.append(student)
        print(f"{student.name} has been attached to {self.name}")
Class Student:
    def__init__(self,name):
        self.teachers=[]
        self.name=name
    def add_teachers(self,teacher):
        self.teachers.append(teacher)
        teacher.attach_student(self)
```

```
teacher_fzn =Teacher()
teacher_fzn.name="fzn"

student_abc =Student(name="abc")
student_abc.add_teacher(teacher_fzn)
```

Fan-Celebrity Example

```
# Subject (Celebrity)
class Celebrity:
    def __init__(self):
        self._fans = []
        self._state = None
    def attach(self, fan):
        self._fans.append(fan)
    def detach(self, fan):
        self._fans.remove(fan)
    def _notify(self):
        for fan in self._fans:
            fan.update(self)
    def set_state(self, state):
        self._state = state
        self._notify()
    def get_state(self):
        return self._state
```

```
# Observer (Fan)
class Fan:
    def __init__(self):
        self._celebrities = []
    def update(self, celebrity):
        state = celebrity.get_state()
    def add_celebrity(self, celebrity):
        self._celebrities.append(celebrity)
        celebrity.attach(self)
    def remove_celebrity(self, celebrity):
        self._celebrities.remove(celebrity)
        celebrity.detach(self)
```

```
celebrity = Celebrity()
fan = Fan()
#fan starts following…
fan.add_celebrity(celebrity)
#celeb changes status and
#notification is received by fan.
celebrity.set_state('New state')
#fan can stop following…
fan.remove_celebrity(celebrity)
```

Observer Pattern


```
public class Celebrity{
   private List<Fan> fans =  new        ArrayList<Fan>();
private int state;
   void attach(Fan f){
      fans.add(f);
   }
    void remove(Fan f){
      fans.remove(f);
   }
   void notify(){
       foreach( Fan f: fans)
          f.update(this);
   }
void setState(int newState){
          state = newState;
          notify();
      }
      int getState(){
           return state;
     }

}
```

Celebrity class may look like this.

Observer Pattern


```
public class Celebrity{
   private List<Fan> fans =  new        ArrayList<Fan>();
private int state;
   void attach(Fan f){
      fans.add(f);
   }
    void remove(Fan f){
      fans.remove(f);
   }
   void notify(){
       foreach( Fan f: fans)
          f.update(this);
   }
void setState(int newState){
          state = newState;
          notify();
      }
      int getState(){
           return state;
     }

}
```

Celebrity class may look like this.

Now add the Fan class

Observer Pattern


```
public class Celebrity{
   private List<Fan> fans =  new ArrayList<Fan>();
private int state;
   void attach(Fan f){
      fans.add(f);
   }
    void remove(Fan f){
      fans.remove(f);
   }
   void notify(){
       foreach( Fan f: fans)
          f.update(this);
   }
void setState(int newState){
          state = newState;
          notify();
      }
      int getState(){
           return state;
     }

}
```

```
public class Fan{
    private List<Celebrity> celebrities=  new ArrayList<Celebrity>();
    void update(Celebrity c){
        c.getState();
    }
    void addCelebrity(Celebrity c){
        celebrities.add(c);
        c.attach(this);
    }
    void removeCelebrity(Celebrity c){
        celebrities.remove(c);
        c.remove(this);
    }

}
```

Observer Pattern

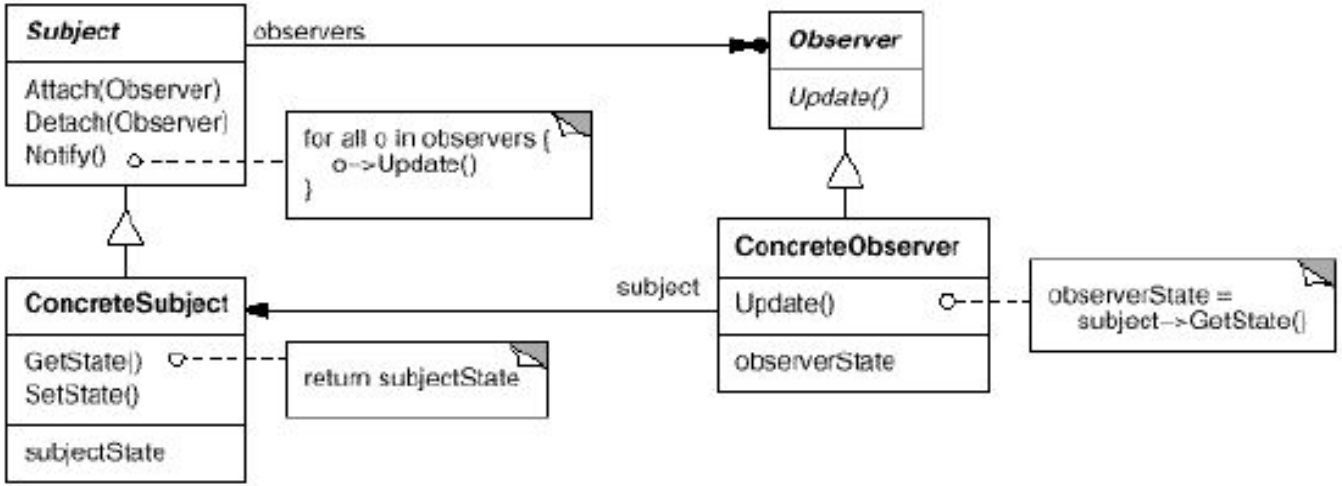
From main method –

```
Fan f1 = new Fan();
Fan f2 = new Fan();

Celebrity c1 = new Celebrity ();
Celebrity c2 = new Celebrity ();

f1.addCelebrity(c1);
f1.addCelebrity(c2);
f2.addCelebrity (c1);
```

Participants

- **Subject:** knows its observers. Any number of Observer objects may observe a subject. It sends a notification to its observers when its state changes. (ex: Celebrity)
- **Observer:** defines an updating interface for objects that should be notified changes in a subject. (ex: Fans)
- **ConcreteSubject:** **(ex: FilmCelebrity, FashionCelebrity)**
- **ConcreteObserver** **(ex: FilmFan, FashsionFan)**

Structure


Subject
Attach(Observer)
Detach(Observer)
Notify()
observers
Observer
Update()
for all o in observers {
o->Update()
}
ConcreteSubject
GetState()
SetState()
subjectState
return subjectState
subject
ConcreteObserver
Update()
observerState
observerState =
subject->GetState()

Advanced Observer with Concrete subjects and observers


```
Public FilmCelebrity extends
Celebrity{
        private int state;
        void setState(int newState){
            state = newState;
            notify();
        }
        int getState(){
             return state;
        }
}
```

Advanced Observer with Concrete subjects and observers


```
Public FilmCelebrity extends
Celebrity{
        private int state;
        void setState(int newState){
            state = newState;
            notify();
        }
        int getState(){
             return state;
        }
}
```

```
public class FilmFan extends Fan{
    private List<FilmCelebrity> filmCelebrities=  new
ArrayList<FilmCelebrity>();
    void update(FilmCelebrity fc){
        if(filmCelebrities.contain(fc){
            fc.getState();
        }
}
    void addCelebrity(FilmCelebrity fc){
        celebrities.add(fc);
        fc.attach(this);
    }
    void removeCelebrity(FilmCelebrity fc){
        celebrities.remove(fc);
        fc.remove(this);
    }

}
```