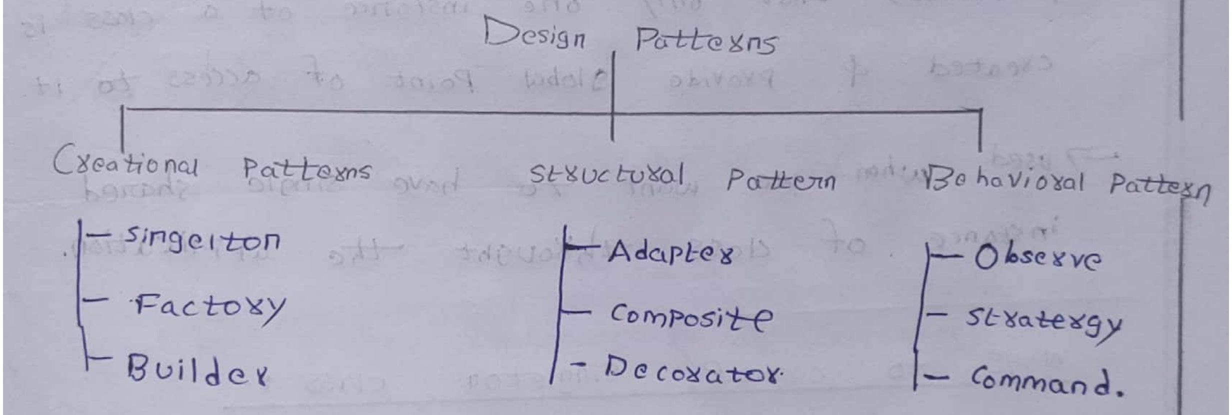
17-6-25.

DESIGN PATTERNS :-

→ A design pattern in programming is a reusable Solution to a common problem that occur during software development or design

→ These patterns provide structured approach to solve design & development issues.

→ used to create more maintainable, flexible & Scalable code.



Design Patterns

- Creational Patterns
  - Singelton
  - Factory
  - Builder
- Structural Pattern
  - Adapter
  - Composite
  - Decorator.
- Behavioral Pattern
  - Observe
  - Strategy
  - Command.

[1] Creational pattern :-

→ These pattern focus on object create mechanism.

→ Provide ways to create object in a manner that is flexible and maintanable.

→ Based on application specific requirements we choose a object creation mechanism.

[2] Structural pattern:-

→ These patterns focus on class & objects to create large structures while keeping them flexible and efficient.

→ Helps us to ensure that classes and objects can work togather efficiently to acheive a goal.

[3] Behavioral Pattern :-

→ Used for intraction & communication between object and classes.

→ Provide solution for efficiently manage flow of Control, behaviour between object.

* Creational Pattern :-

[i] Singelton Pattern :-

→ Ensures that only one instance of a class is created & provide global point of access to it.

→ used when we want to have single shared instance of class throught the application.

Ways to create singleton class :-

[i] Eager intialisation [4] Double check

[i] Lazy intialisation [5] Bill Pugh Solution.

[3] Synchronized block. [6] Enum

Main class :-

```
Public class main
{
    Public static void main (String args[])
    {
        DBConnection obj = DB Connection.get Instance();
    }
}
```

Eager Intialisation :-

```
Public class DBConnection
{
    Private Static DB Connection canobject = new DB Connection();
     Private DBConnection () { }
     Private Static DBConnection getInstance()
     {
          return canobject;
     }
}
```

Lazy intialization:-

```
Public class DBConnection
{
     private Static DBConnection conObject;
     private DBConnection () {}.
     Public static DB Connection getInstance()
     {
            if (conObject == null)
                   conObject = new DB Connection();
            return conObject
     }
}
```

Synchronization BLOCK :-

```
Public class DBConnection
{
     Private static DB Connection conObject
     private DBConnection () {};
     Synchronized Public static DBconnection getInstance()
     {
            if (conObject == null)
                   ConObject = new DB Connection()
            return conObject
     }
}
```

Double check locking system:-

```
Public class DB Connection
{
    Private static Volatile DB Connection con=new DB Connectio
    private DBConnection (){}
    Public static DB Connection get Instance()
    {
        Synchronized (DB Connection.class)
        {
            if (con ==null)
            {
                con = new DB Connection()·
            }
        }
        return con.
    }
}
```

Big Pugh singleton:- (uses eager intialisation)

```
Public class DBC
{
    private DBC () {}
    Private static class DBHelper
    {
        private static final DB = INSTANCE_OBJECT=new DBC
    }
    Public static DB get Instance()
    {
        return DBHelper.INSTANCE_OBJECT
    }
}
```

ENUM:-

```
Public enum DBC
{
    INSTANCE
    Public static doSomething (){
    }
}
```

Builder Pattern :-

→ Builder Pattern is a creational design Pattern that allows you to Construct Complex objects step-by-step separating the Construction logic from the representation.

→ useful when objects have many optional fields

→ Avoid telescoping Constructors.

Without Builder pattern

Person. Java :-

```
Public class Person
{
   private String name;
  private int age;

  Public Person (String name, int age)
  {
      this. name = name;
      this. age = age;
  }

  Public void display ()
  {
      S.O.P ("Name" + name + ...);
  }
}
```

Test. Java :

```
Public class Test
{
   Public static void main (String args[])
   {
      Person p = new person ("virat", 35).
      P. display ();
   }
}
```

with builder Pattern

Person. Java.

```
Public class Person
{
   private String name;
   private int age;
   private Person (Builder builder)
   {
      this.name = buider.name.
      this. age = builder.age.
   }
   Public void display () {
      S.o.f (name+ ...);
   }
   Public static class Builder {
      Private String name.
      Private int age.
      Public Builder setName (String n)
      {
         this.name = name;
         return this.
      }
      Public Buider setAge (int age)
      {
         this.age = age.
         return this.
      }
      Public Person build () {
         return new Person (this)
      }
   }
}
```

Inside main

```
Person P=new person.Builder()
                 .setName ("kohli").
                 .setAge (35)
                 .build ().
P. display
```

Factory Pattern :-

→ A creational design pattern that provides an interface to create objects in a superclass but allows subclass to alter the type of objects that will be created.

→ Avoid using new directly in the client.

→ We will use factory pattern when we need to create object based on input or condition.

Code :-

```
Public interface shape {
      void draw();
}

Public class circle implements shape {
        Public void draw () {
              system.out.println ("Drawing a circle");
        }
}
Public class Square implements shape {
        Public void draw () {
              system.out.println ("Drawing a square");
        }
}


Public class shape Factory
{
     Public shape get Shape (String type)
     {
          if (type.equalsIgnore Case ("circle")) {
                return new Circle();
          } else if (type.equalsIgnore Case ("square")) {
                return new Square();
          }
     } return null;
}
```

Main.Java

```
Public   class   Main
{
    Public  static  void  main (String[] args)
    {
        ShapeFactory  factory = new  ShapeFactory();
        Shape  s1 = factory.getShape("circle");
        s1.draw();
        Shape  s2 = factory.getShape("square");
        s2.draw();
    }
}
```

Structral Pattern :-

Adapter:

→ Adapter Pattern is a structural pattern that allows object with incompatible interfaces to work together by converting one interface into another.

→ Bridge the gap between two incompatible interfaces.

Client → Target (expected interface)

↑

Adapter

↓

Adaptee (incompatible class).

Code:

```
class   Old Printer {
        Public  void  printOld () {
            System.out.println("old printer");
}       }
```

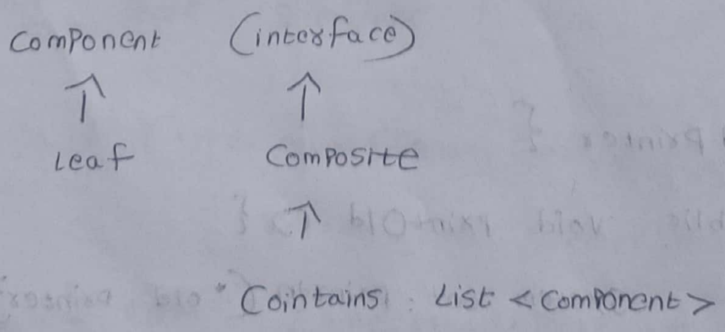
```
interface Printer {
        void print();
}

class PrinterAdapter implements Printer {
        Private OldPrinter oldPrinter;
        Public PrinterAdapter (OldPrinter oldPrinter){
                this.old Printer = oldPrinter;
        }
        Public Void print () {
                old Printer.printOld();
        }
}

Public class Test {
        Public static void main (String [] args) {
                oldprinter old = new old Printer();
                Printer adapter = new PrinterAdapter (old);
                adapter.print();
        }
}
```

[2] Composite Pattern :-

→ used to treate individual objects and groups of objects in a uniform way.



```
public interface Employee {
        void showDetails();
}
public class Developer implements Employee {
        private String name;
        Public Developer (String name) {
                this.name = name;
        }
        Public void showDetails () {
              System.out.println ("Developer :" +name);
        }
}

Public class Manager implements Employee {
        private String name;
        private List <Employee> team =new ArrayList<>();
        Public Manager (String name) {
              this.name = name;
        }
        Public void add (Employee emp) {
              team.add (emp);
        }
        Public void showDetails () {
              System.out.println ("Manager : ", +name);
              for (Employee e : team) {
                      e.showDetails ();
              }
        }
}
Public class Test {
        public Static void main (String [] args) {
              Developer dev1 =new Developer ("Rahul");
              Developer dev2 = new Developer ("kiran");
manager   manager   = new   Manager ("virat");
manager . add (dev1);
manager . add (dev2);
manager . show Details();
}
```

Behav

Behavioral Pattern :-

observe Pattern:

→ where an object maintains a list of dependents and notifies them automatically when its state changes.

```
Public interface observer {
        void update (String message);
}

Public   class   Follower  implements   observer {
      Private   String name;
      Public  Follower (String name) {
            this.name =name;
      }
      Public  void  update (String message) {
            System.out.println (name);
      }
}

Public  interface  Subject {
      Void  addObserver (Observer o);
      Void  removeObserver (Observer o);
      Void  notifyObserver (String msg);
}

import   java.util.*;

Public  class  Channel  implements  Subject{
      Private  List <observer> observers =new  ArrayList<>();
      Public  void  addObserver (observer o) {
            observs.add (o);
      }
}
```

```
Public   class   Test {
    Public  Static  void  main (String [] args){
        channel  chanel =new  channel();
         Follower  f1 =new  Follower ("vivat);
         channel . addObserver (f1);
         channel .upload ("Highlights").
    }
}
```