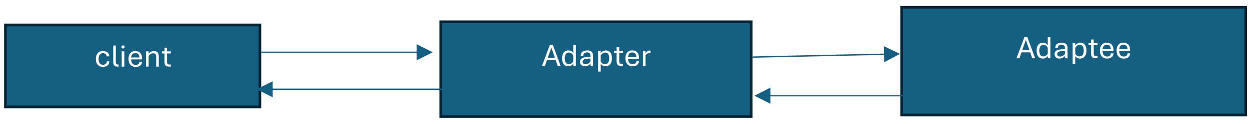
Structural Design Pattern

Is a way to combine or arrange different classes and objects to form a complex or bigger structure to solve a particular requirement.

Adapter Design pattern

This pattern act as a bridge or intermediate between 2 incompatible interfaces.



//WeightMachine Interface

```
package LowLevelDesign.DesignPatterns.AdapterDesignPattern.Adaptee;


public interface WeightMachine {
  // Return the weight in pounds
  public double getWei-*/ghtPound();
}
```

//WeightMachineForBabies Class

```
package LowLevelDesign.DesignPatterns.AdapterDesignPattern.Adaptee
public class WeightMachineForBabies implements WeightMachine {
  @Override
  public double getWeightPound() {
    return 28;
  }
}
```

//WeightMachineAdapter Interface

```
package LowLevelDesign.DesignPatterns.AdapterDesignPattern.Adapter;


public interface WeightMachineAdapter {
  public double getWeightInKg();
}
```

//WeightMachineAdapterImpl Class

```
package LowLevelDesign.DesignPatterns.AdapterDesignPattern.Adapter;


import
LowLevelDesign.DesignPatterns.AdapterDesignPattern.Adaptee.WeightMachine;


public class WeightMachineAdapterImpl implements WeightMachineAdapter {
  WeightMachine weightMachine;


  // constructor
  public WeightMachineAdapterImpl(WeightMachine weightMachine) {
    this.weightMachine = weightMachine;
  }



  public double getWeightInKg() {
    double weightInPound = weightMachine.getWeightPound();
    // Convert pounds to kilograms
    return weightInPound * 0.45;
  }
}
package LowLevelDesign.DesignPatterns.AdapterDesignPattern.Client;
import
LowLevelDesign.DesignPatterns.AdapterDesignPattern.Adapter.WeightMachineAdapte
r;
```

```
import
LowLevelDesign.DesignPatterns.AdapterDesignPattern.Adapter.WeightMachineAdapte
rImpl;

import
LowLevelDesign.DesignPatterns.AdapterDesignPattern.Adaptee.WeightMachineForBab
ies;
```

//Main class

```
public class Main {

   public static void main(String[] args) {

  WeightMachineAdapter weightMachineAdapter = new
WeightMachineAdapterImpl(new WeightMachineForBabies());

      System.out.println("Weight in KG: " + weightMachineAdapter.getWeightInKg());

   }

}
```

BRIDGE DESIGN PATTERN

This pattern helps to decouple an abstraction from its implementation ,so that two can vary independently.

```
public abstract class LivingThings {

        abstract public void breatheProcess();

}

public class Dog extends LivingThings {

        public void breatheProcess() {

                //Breath through NOSE

                //Inhale oxygen  from Air

                //Exhale carbondioxide

        }

}

public class Fish extends LivinfThings {

        public void breatheProcess() {
                //breathe through GILLS

                //Absorb oxgen from water

                //Release carbon dioxide

        }

}

public class Tree extends LivingThings {

        public void breatheProcess() {

                //Breathe through LEAVES

                //Inhale Carbon dioxide

                //Exhale oxygen through photosynthesis

        }

}
```

To add new breathe process ,We should add a new class like bird

```
public class Bird extends LivingThings {

        public void breatheProcess() {

        //Inhale through NOSEL;

        //Exhale through mouth;

        ...

        }

}
```

But how to add new Breathing Process without adding any class of LivingThings?

There is no child class currently using such breathe process which I want to include in my application.as they tightly coupled

Implementor Interface

```
public interface BreatheImplementor {

  void breatheProcess();

}
```

Concrete Implementors

```
public class LandBreatheImplementation implements BreatheImplementor {
  public void breatheProcess() {
    // Breathe through nose
    // Inhale oxygen from air
    // Exhale carbon dioxide
    System.out.println("Land breathing using lungs");
  }
}
public class WaterBreatheImplementation implements BreatheImplementor {
  public void breatheProcess() {
    // Breathe through gills
    // Absorb oxygen from water
    // Release carbon dioxide
    System.out.println("Water breathing using gills");
  }
}
public class TreeBreatheImplementation implements BreatheImplementor {
  public void breatheProcess() {
    // Breathe through leaves
    // Inhale carbon dioxide
    // Exhale oxygen
    System.out.println("Tree breathing using leaves");
  }
}
```

Abstraction

```
public abstract class LivingThings {

  protected BreatheImplementor breatheImplementor;

  public LivingThings(BreatheImplementor breatheImplementor) {
    this.breatheImplementor = breatheImplementor;
  }

  public abstract void breathe();
}
```

Refined Abstractions

```
public class Dog extends LivingThings {

  public Dog(BreatheImplementor breatheImplementor) {
    super(breatheImplementor);
  }

  public void breathe() {
    breatheImplementor.breatheProcess();
  }
}
public class Fish extends LivingThings {

  public Fish(BreatheImplementor breatheImplementor) {
    super(breatheImplementor);
  }
  public void breathe() {
    breatheImplementor.breatheProcess();
  }
}
public class Tree extends LivingThings {

  public Tree(BreatheImplementor breatheImplementor) {
    super(breatheImplementor);
  }

  public void breathe() {
    breatheImplementor.breatheProcess();
  }
}
```

Client Code

```
public class BridgeDemo {
  public static void main(String[] args) {

    LivingThings dog = new Dog(new LandBreatheImplementation());
    dog.breathe();

    LivingThings fish = new Fish(new WaterBreatheImplementation());
    fish.breathe();

    LivingThings tree = new Tree(new TreeBreatheImplementation());
    tree.breathe();
  }
}
```

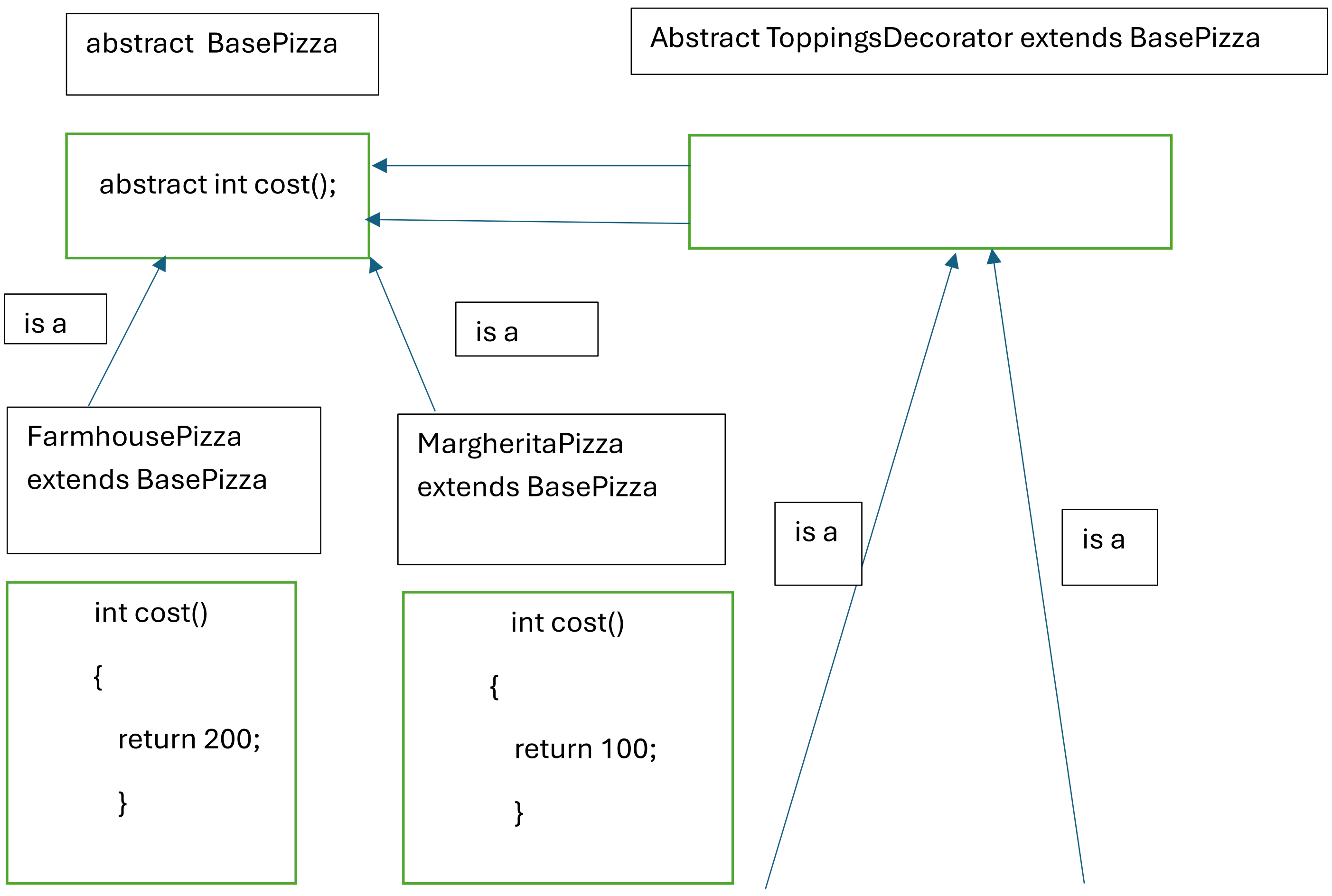
Decorator Design Pattern

Which to Choose?

- **Builder Pattern: Use this when you want to construct pizzas in a step-by-step manner with a fixed set of options, ensuring immutability after creation. This approach is straightforward and works well when the combinations are limited and known beforehand.**
- **Decorator Pattern: Opt for this when you need the flexibility to add or remove toppings dynamically at runtime, allowing for a wide variety of combinations without creating a subclass for each possible pizza variant. This pattern is beneficial when new toppings are introduced frequently, as it promotes scalability and maintainability.**

Decorator Design Pattern

This pattern helps to add more functionality to existing object , without changing its structure.


abstract BasePizza
abstract int cost();
Abstract ToppingsDecorator extends BasePizza
is a
is a
FarmhousePizza extends BasePizza
int cost() { return 200; }
MargheritaPizza extends BasePizza
int cost() { return 100; }
is a
is a

ExtraCheese extends ToppingDecorator

```
BasePizza basePizza;

public EXtraCheese(BasePizza pizza)

    {

          this.basePizza=pizza;

    }

 int cost()

 {

          return basePizza.cost+10;

}
```

Mushroom extends ToppingDecorator

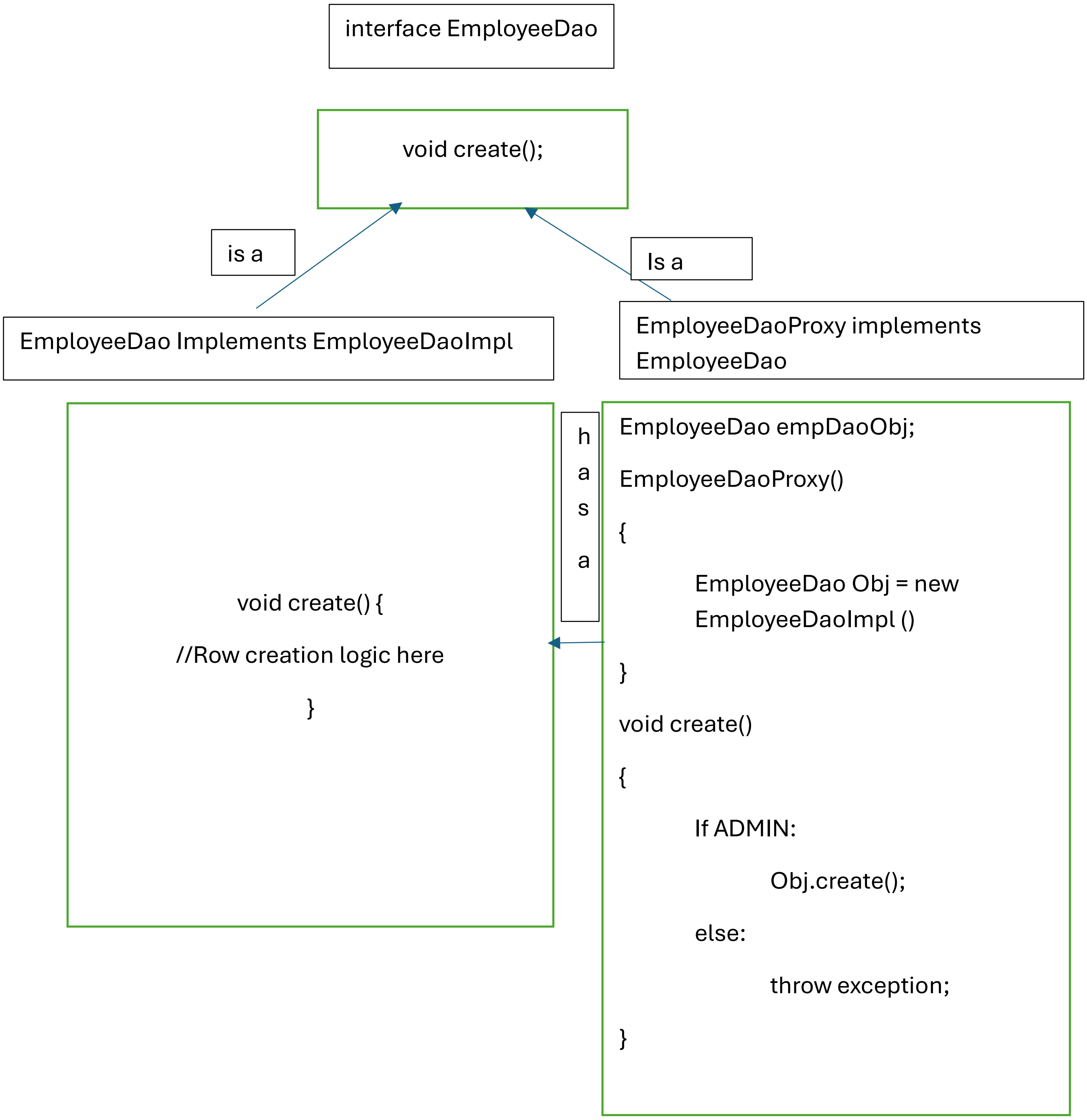
```
BasePizza basePizza;

public Mushroom(BasePizza pizza)

    {

          this.basePizza=pizza;

    }

 int cost()

 {

          return basePizza.cost+15;

}
```

```
BasePizza pizza = new Mushroom(new ExtraCheese(new Farmhouse()));
```

2.proxy patterns

The pattern helps to provide control access to original object.

Proxy will act like a middle ware between client and resource, before accessing certain resource.



```
interface EmployeeDao

void create();
```

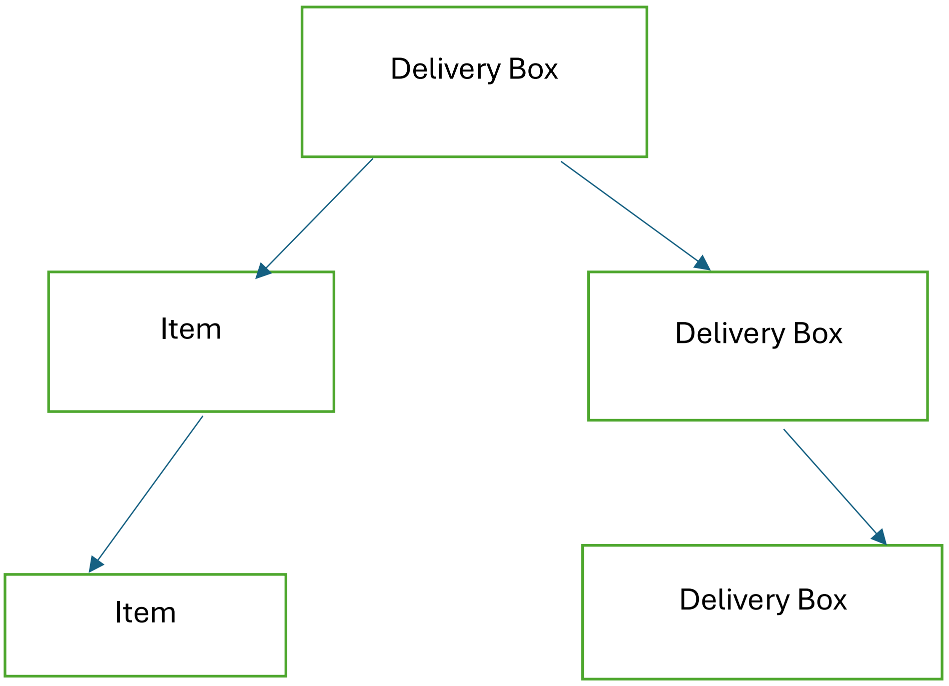
```
EmployeeDao Implements EmployeeDaoImpl

void create() {
//Row creation logic here
}
```

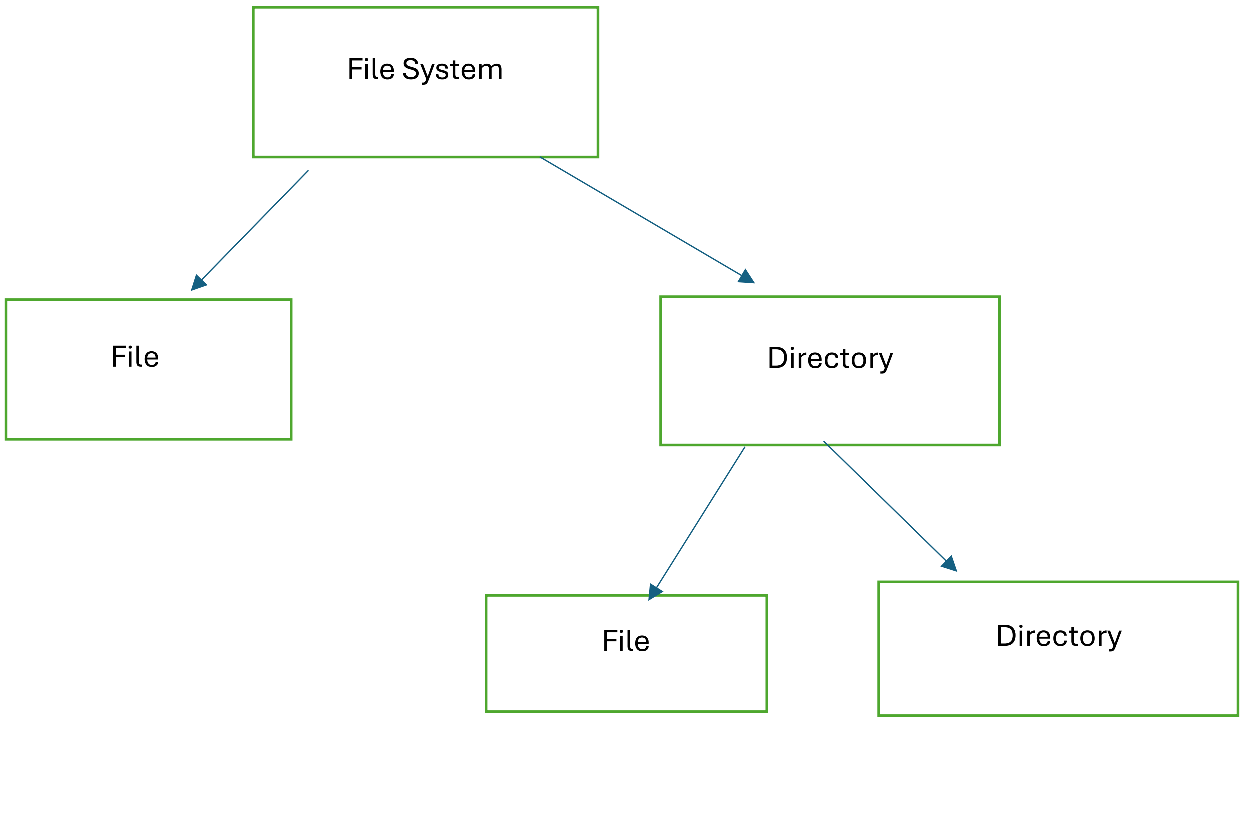
```
EmployeeDaoProxy implements
EmployeeDao

EmployeeDao empDaoObj;
EmployeeDaoProxy()
{
        EmployeeDao Obj = new
        EmployeeDaoImpl ()
}
void create()
{
        If ADMIN:
                Obj.create();
        else:
                throw exception;
}
```

```
EmployeeDao empProxyObj = new EmployeeDaoProxy();
empProxyObj.create();
```

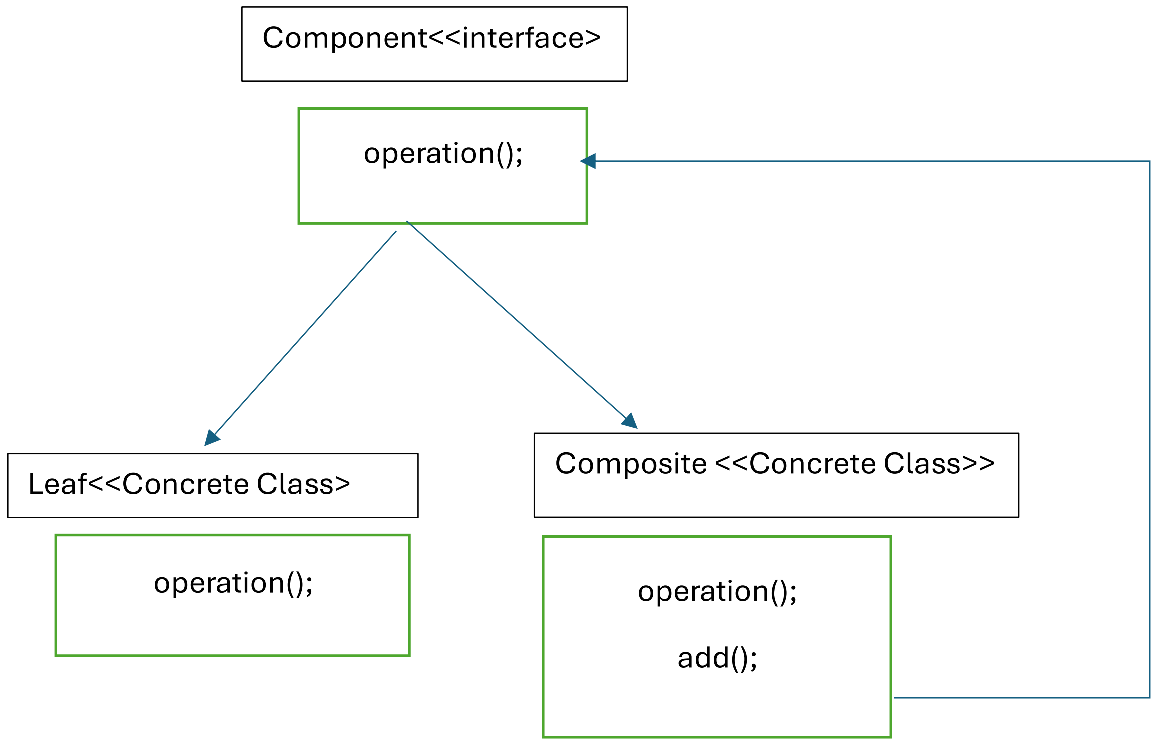
3.COMPOSITE PATTERN

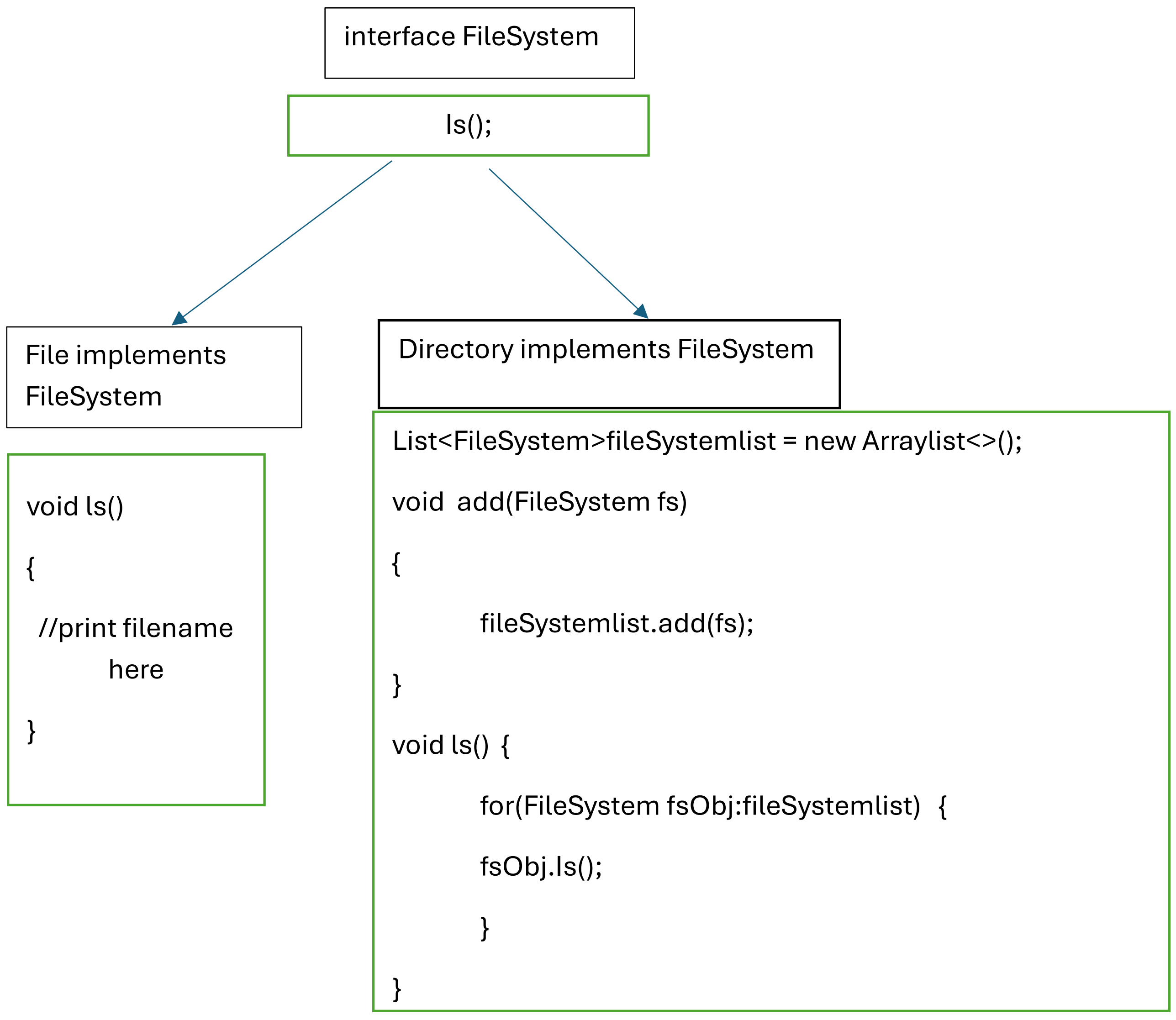
This pattern helps in scenarios where we have OBJECT inside OBJECT(tree like structure)





Uml diagram





```
void ls()
{
  //print filename
        here
}
```

```
List<FileSystem>fileSystemlist = new Arraylist<>();
void  add(FileSystem fs)
{
        fileSystemlist.add(fs);
}
void ls() {
        for(FileSystem fsObj:fileSystemlist)  {
        fsObj.ls();
        }
}
```

```
Directory parentDir =new Directory();

FileSystem fileObj1 = new File();

parentDir.add(fileObj1);

Directory childDir = new Directory();

FileSystem fileObj2 = new File();

childDir.add(fileObj2);

parentDir.add(childDir);

parentDir.ls();
```

Façade design pattern

When to use and why to use?

When we have do hide the system complexity from client,Expose only the necessary details to the client.

Eg: Car client do not know the complexity of accelerate or break

DAO-data access object

```
public class EmployeeDAO {

        public void insert() {

                //insert into Employee Table

}

public void updateEmployeeDetails(String empname) {

        //updating employee Name

}.

public Employee getEmployeeDetails(String EmpID) {

        //get employee details based on EmpID

        return new Employee();

 }

public Employee getEmployeeDetails(int EmpID){

        //get employee details based on EmpID

        Return new Employee();
}

}

public class EmployeeFacade {

EmployeeDAO employeeDAO;

public EmployeeFacade(){

        employeeDAO= new EmployeeDAO();

}

public void insert () {

        employeeDAO.insert();

}

public Employee getEmployeeDetails(int empID){

        return employeeDAO.getEmployeeDetails(empID);

}

}

//client

public class EmployeeClient {

        public void getEmployeDetails() {

                EmployeeFacade employeeFacade= new EmployeeFacade();

                Employee employeeDetails=
employeeFacade.getEmployeeDetails(empID: 121222);

}

}
```

Façade responsible for creation of object for required class and expose only methods need for client to reduce complexity

7.Flyweight Design Pattern

This pattern helps to reduce memory usage by sharing data among multiple objects.

Issue: lets say memory is 21GB

```
Robot
```

```
int coordinateX;  //4bytes

int coordinateY;  //4bytes

String type;     //50bytes (1 byte @ 50 char length)

Sprites body;  //2d bitmap,31KB

Robot(int x,int y,String type,Sprites body)

{

        this.coordinateX = x;

        this.coordinateY=y;

        this.type=type;

        this.body=body;

}
```

=~31KB

10lakh*~31KB=31GB

ISSUE AS MEMORY IS 21GB ONLY

```
int x=0;

int  y=0;

for(int i=1;i<500000;i++) {

        Sprites humanoidSprite = new Sprites();

        Robot humanoidBotObj = new Robot(x+I;y+I,"HUMANOID",humanoidSprite);

        }

for(int i=1;i<500000;i++) {

        Sprites rpboticDogSprite= new Sprites();

        Robot roboticDobObj= new Robot(x+I,y+i."ROBOTICDOB",robotDogSprite);

        }

}
```

Intrinsic data: shared among objects and remain same once defined one value.

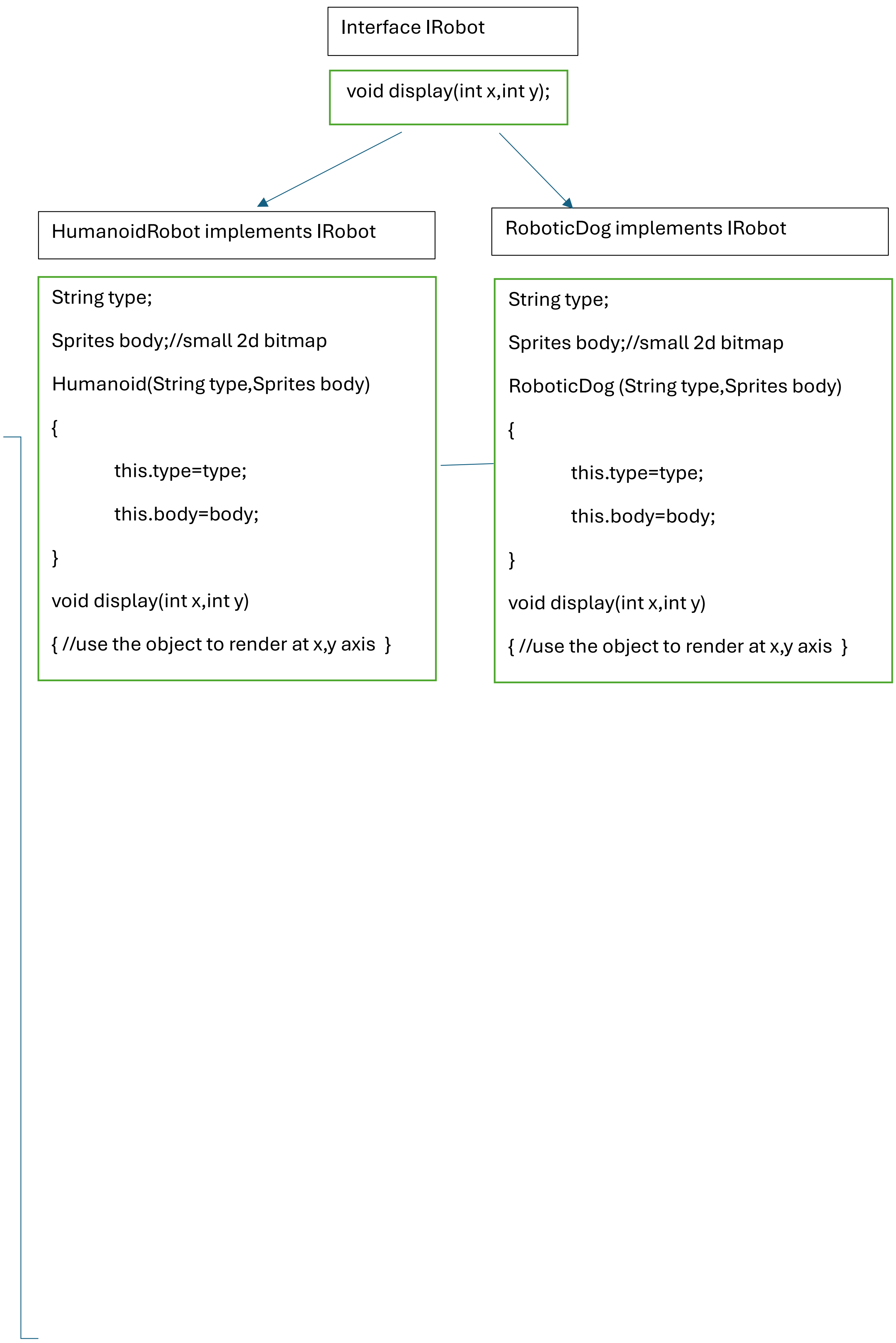
Like in above example:Type and Body is Instrinsic data.

Extrinsic data: change based on client input and differs from one object to another .Like in above example :X and Y axis are Extrinsic data

From Object ,remove all the Extrinsic data and keep only Intrinsic data(this object is called Flyweight Object)

Extrinsic data can be passed in the parameter to the Flyweight class.

Caching can be used for the Flyweight object and used when ever required.



```
Interface IRobot

void display(int x,int y);
```

```
HumanoidRobot implements IRobot

String type;

Sprites body;//small 2d bitmap

Humanoid(String type,Sprites body)

{

        this.type=type;

        this.body=body;

}

void display(int x,int y)

{ //use the object to render at x,y axis  }
```

```
RoboticDog implements IRobot

String type;

Sprites body;//small 2d bitmap

RoboticDog (String type,Sprites body)

{

        this.type=type;

        this.body=body;

}

void display(int x,int y)

{ //use the object to render at x,y axis  }
```

Robotic Factory

```
                    static Map<String,IRobot>roboticObjectCache= new HasMap<>();
static IRobot createRobot(String robotType))
{
        if(roboticObjectCache.containsKey(robotType))
        {
                return roboticObjectCache.get(robotType);
        }
        if(robotType.equals("HUMANOID")
        {
                Sprites humanoidSprite = new Sprite();
                IRobot humanRobotObj = new HumanoidRobot(robotType,humanoidSprite);
                roboticObjectCache.put(robotType,humanRobotObj);
                return humanRobotObj;
        }
        Else if(robotType.equals("ROBOTIcDOG")
        {
                Sprites roboticDogSprite = new Sprite();
                IRobot roboticDogObj = new RoboticDog(robotType,roboticDogSprite);
                roboticObjectCache.put(robotType,roboticDogObj);
                return roboticDogObj;
        }
        return null;
}
```

```
IRobot humanoidRobot1 = RoboticFactory.createRobot("HUMANOID");
humanoidRobot2.display(1,2);

IRobot humanoidRobo21= RoboticFactory.createRobot("HUMANOID");
humanoidRobot2.display(1,2);
```