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

//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.WeightMachineAdapter;
import LowLevelDesign.DesignPatterns.AdapterDesignPattern.Adapter.WeightMachineAdapterImpl;
import LowLevelDesign.DesignPatterns.AdapterDesignPattern.Adaptee.WeightMachineForBabies;
```

//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

```
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;
	...
	}
```

```
}
```

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

```
public interface BreatheImplementor {
        public void breather();
}
public class LandBreatheImplementation implements BreatheImplementor {
public void breathe() {
                //Breath through NOSE
                //Inhale oxygen  from Air
                //Exhale carbondioxide
        }
}
public class WaterBreatheImplementation implements BreatheImplementor {
public void breathe() {
                //breathe through GILLS
                //Absorb oxgen from water
                //Release carbon dioxide
        }
}
public class TreeBreatheImplementation implements BreatheImplementor {
public void breathe() {
public void breatheProcess() {
                //Breathe through LEAVES
                //Inhale Carbon dioxide
                //Exhale oxygen through photosynthesis
        }
}
```

```
public abstract class LivingThings {
	BreatheImplementor   breatheImplementor;
	public LivingThings(BreatheImplementor breatheImplementor) {
		this.breatheImplementor=breatheImplementor;
	}
	abstract public void breatheProcess();
	}
public class Dog extends LivingThings {
	public Dog(BreatheImplementor breatheImplementor) {
		super(breatheImplementor);
	}
	public void breatheProcess() {
		breatheImplementor.breathe();
	}
}
public class Fish extends LivingThings {
	public Fish(BreatheImplementor breatheImplementor) {
		super(breatheImplementor);
	}
	public void breatheProcess() {
		breatheImplementor.breathe();
	}
}
public class Tree extends LivingThings {
	public Tree(BreatheImplementor breatheImplementor) {
		super(breatheImplementor);
	}
	public void breatheProcess() {
```

```
            breatheImplementor.breathe();
        }
}
```

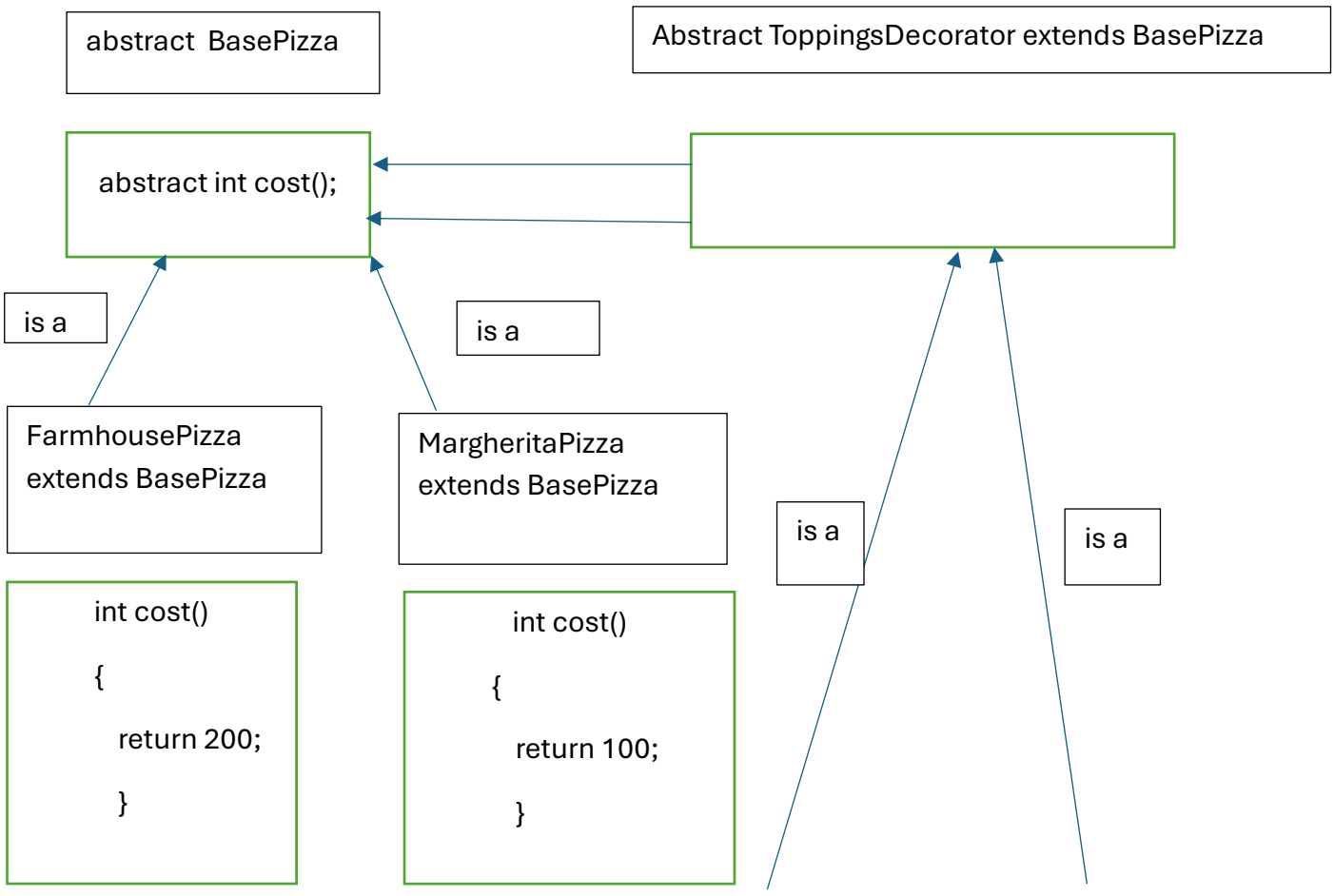
Decorator Design Pattern

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.



ExtraCheese extends ToppingDecorator

```
BasePizza basePizza;
public EXtraCheese(BasePizza pizza)
    {
        this.basePizza=pizza;
    }
 int cost()
 {
        return basePizza.cost+10;
}
```

Mushroom extends ToppingDecorator

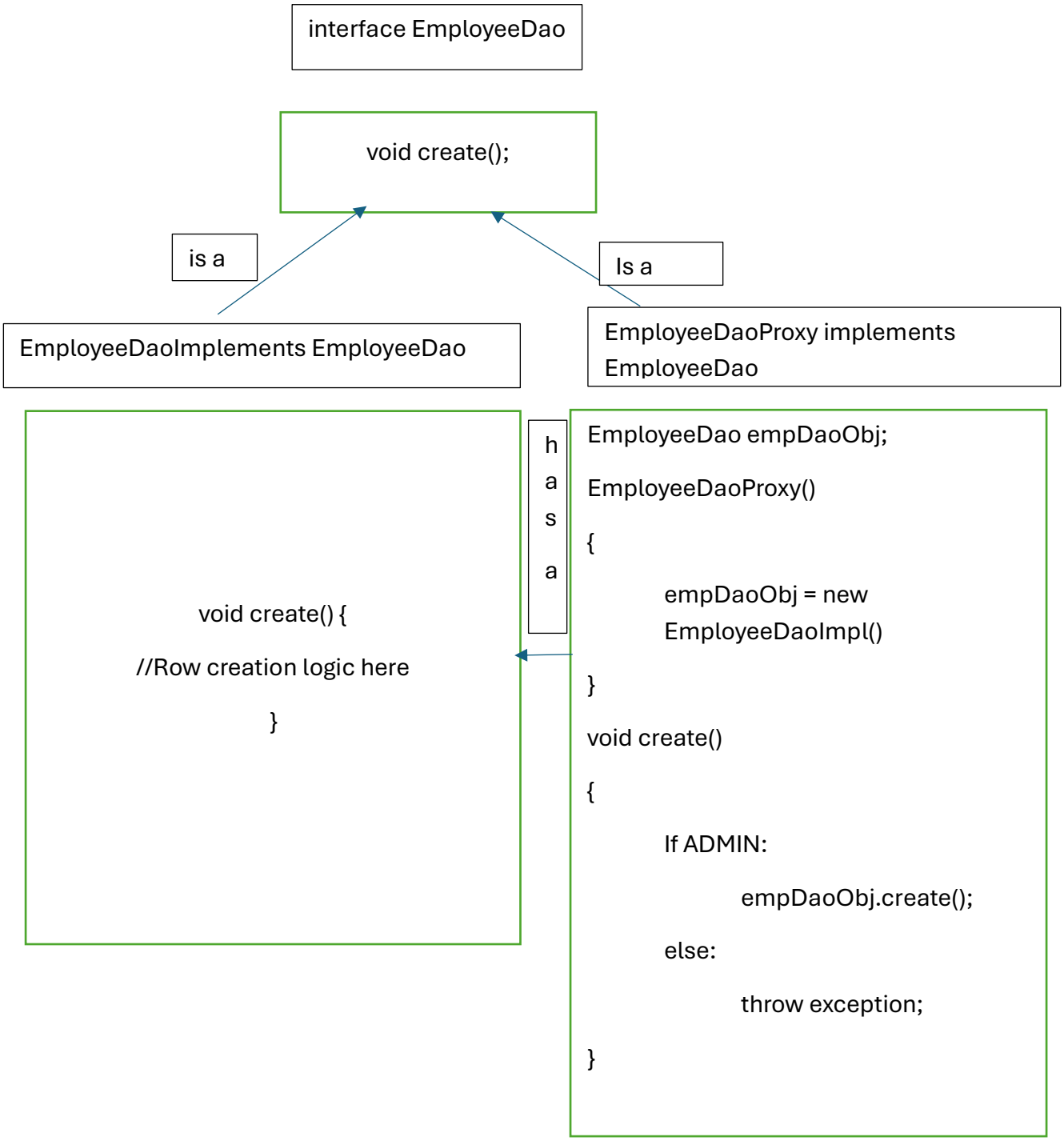
```
BasePizza basePizza;
public Mushroom(BasePizza pizza)
    {
        this.basePizza=pizza;
    }
 int cost()
 {
        return basePizza.cost+15;
}
```

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

2.proxy patterns

The pattern helps to provide control access to original object.

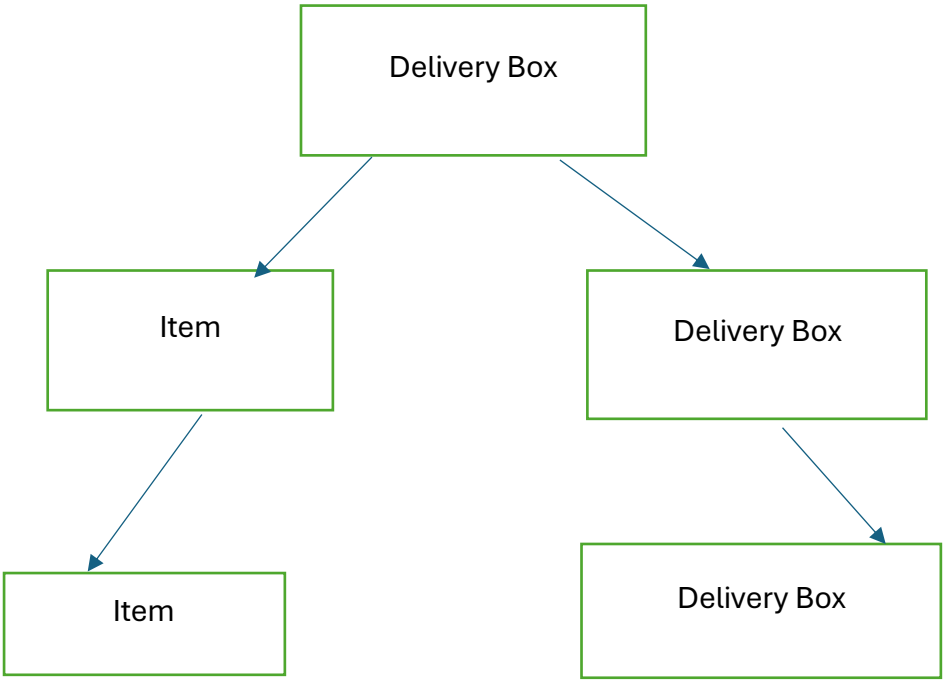
Proxy will act like a middle ware between client and resource, before accessing certain validation

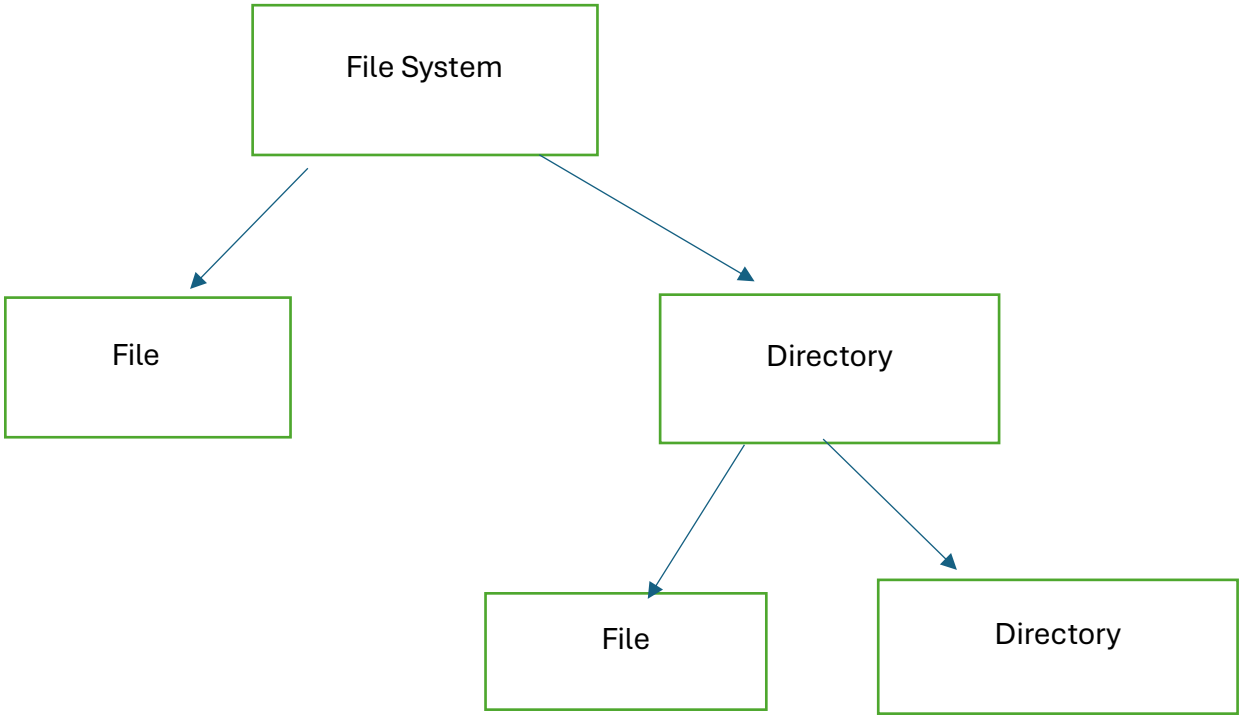



```
EmployeeDao empProxyObj = new EmployeeDaoProxy();
empProxyObj.create();
```

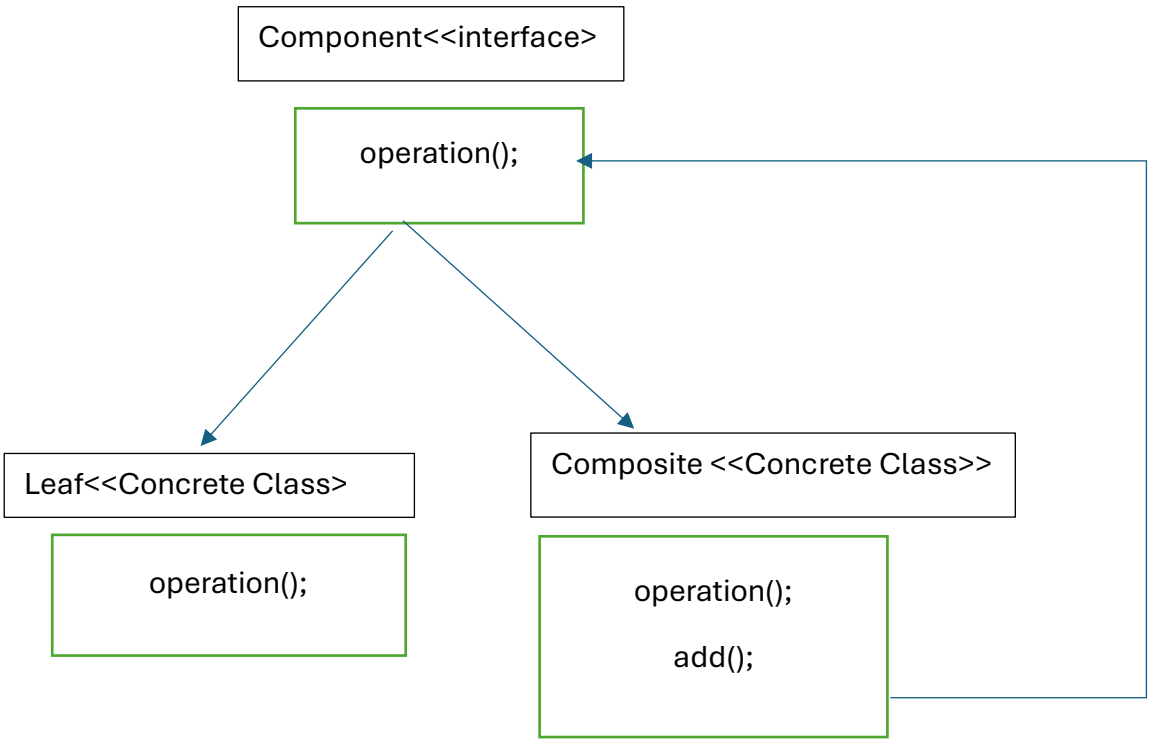
3.COMPOSITE PATTERN

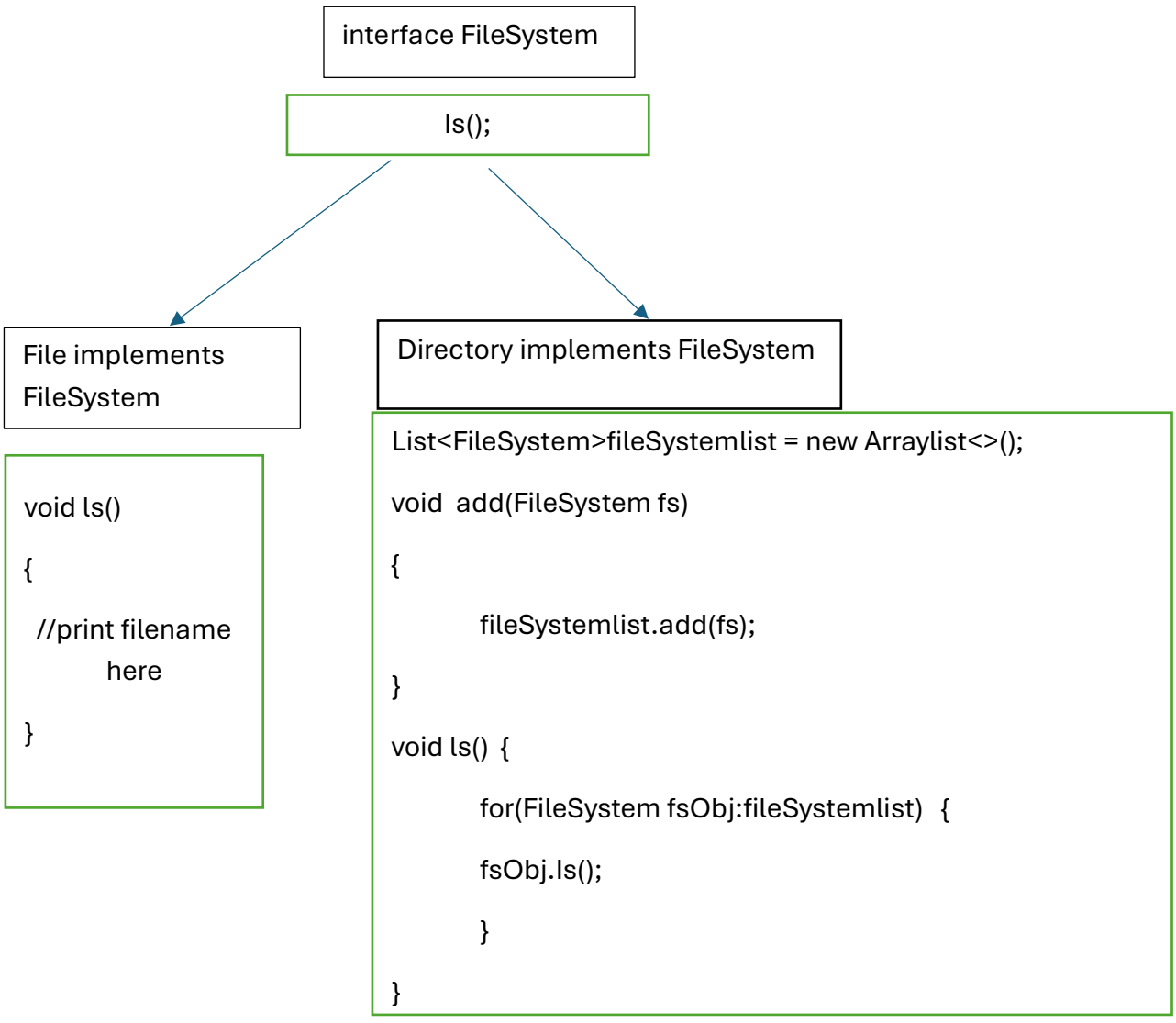
This pattern helps in scenarios where we have OBJECT inside OBJECT(tree like structure)





Uml diagram


Component<<interface>
operation();
Leaf<<Concrete Class>
operation();
Composite <<Concrete Class>>
operation();
add();


interface FileSystem
ls();
File implements FileSystem
void ls()
{
//print filename here
}
Directory implements FileSystem
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

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 emailID) {
        //updating employee Name
}
public Employee getEmployeeDetails(String emailID) {
        //get employee details based on EmpID
        return new Employee();
 }
public Employee getEmployeeDetails(int email){
        //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 clas and expose only methods need for client to reduce complexity

Senerio-2

```
public class ProductDAO {
        public Product getProduct(int product ID) {
                //get product based on product id and return
                Return new Product();
        }
```

```
}
public class Payment {
        public Boolean makePayment(){
        //initiate payment and return true if success
        Return true;
}
public class invoice {
        public void generateInVoice () {
        //generate invoice
}
}
public class SendNotification {
        public void sendNotification(){
                //this will send notification to customer on mobile
         }
}
Public class OrderFacade {
ProductDAO productDAO;
Invoice invoice;
Payment payment;
SendNotification notification;
public OrderFacade() {
        produceDAO = new ProduceDAO();
        invoice =new Invoice();
        payment= new Payment();
        notification =new SendNotification();
//order creation successful
}
```

```
}
public class OrderClient {
public static void main(String argd[]){
        OrderFacade orderFacade = new OrderFacade();
        orderFacade.createOrder();
}
}
```

PROXY DESIGN PATTERN

```
public interface Internet {
        void connectTo(String host);
}
public class RealInternet implements Internet {
        public void connectTo(String host) {
        }
}
public static void main(String [] args) {
        Internet internet = new RealInternet();
        Internet.connectTo("google.com");
}
public class ProxyInternet implements Internet {
        private static final  List<String> bannedSites;
        private final Internet internet =  new RealInternet();
        static {
                bannedSites = new ArrayList<>();
                bannedSites.add("banned.com");
        }
```

```
        public void connectTo(String host) {
                if(bannedSites.contains(host) {
                        System.out.println("Access Denied!");
                        return;
                }
                Internet.connectIn(host);
}
```

Provides a substitute for another object and controls access to that object,allowing you to perform something before or after the request reaches the original object

Senerio-2

```
public class RealVideoDownloader implements VideoDownloader {
        public Video getVideo(String videoName) {
                System.out.println("Connecting to https://www.youtube.com/");
                System.out.println("Downloading Video");
                System.out,println("Retrieving video Metadata");
                return new Video(VideoName);
                }
                }
                public static void main(final Static arguments) {
                        VideoD0wnloader videoDownloader= new RealDownloader();
                        videoDownloader.getVideo("geekific");
                        videoDownloader.getVideo("geekific");
                        videoDownloader.getVideo("likeNsub"(;
                        videoDownloader.getVideo("likeNsub");
                        videoDownloader.getVideo("geekific");
                }
                We need to cache the info of downloaded videos!
                 public class ProxyVideoDownloader implements VideoDownloader {
```

```
                    private final Map<String, Video> videoCache = new HashMap<>();
                    private final VideoDownloader downloader = new
        RealVideoDownLoader();
            public Video getVideo(String videoName) {
                    if(!videoCache.containsKey(videoName) {
                            vidoeCache.put(videoName,
        downloader.getVideo(videoName));
            }
            Return videoCache.get(videoName);
            }
}
public static void main(String[] args) {
        Internet internet = new ProxyInternet();
        internet.connectTo(“google.com”);
        internet.connectTo(“banned.com”);
}
```

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
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.

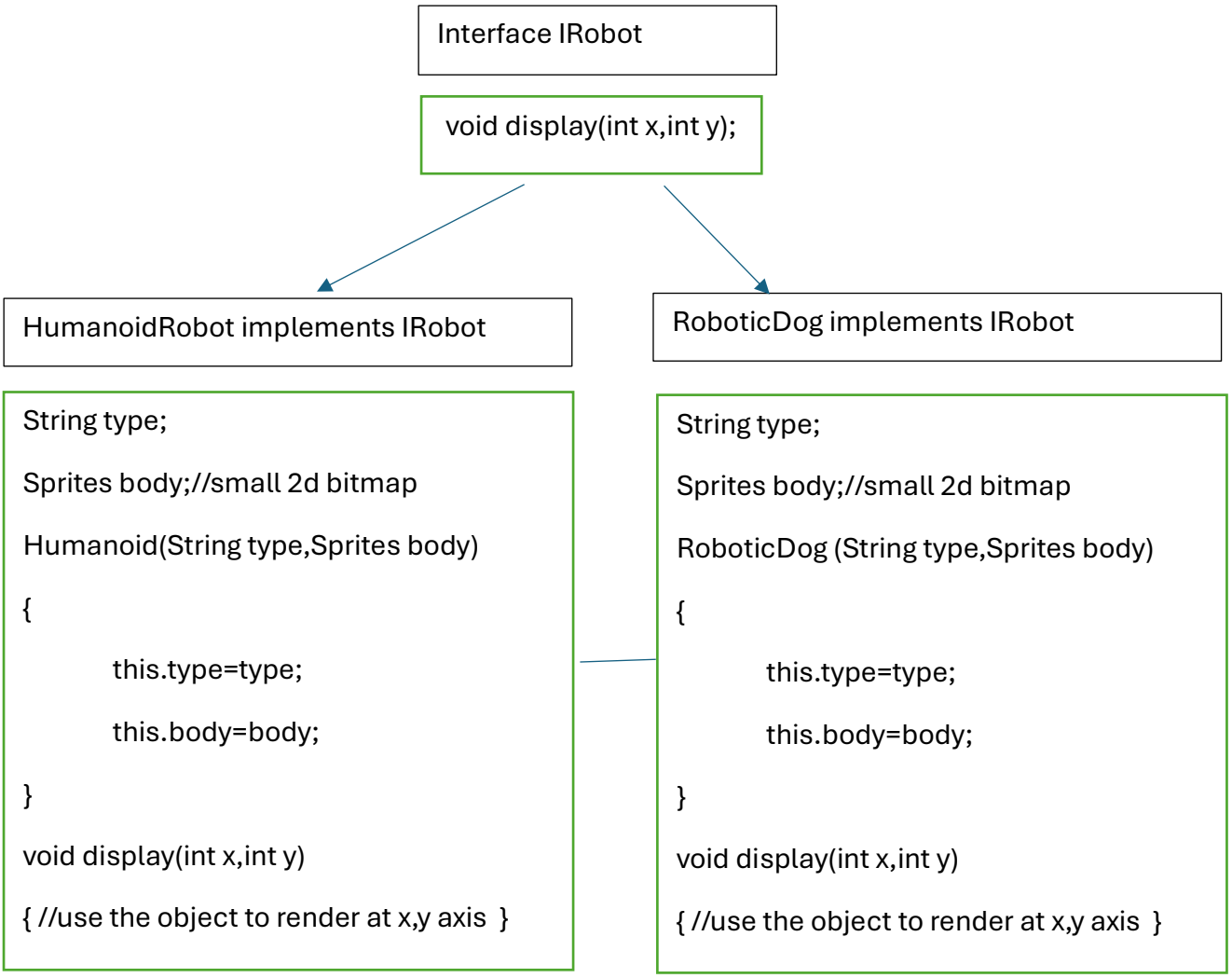
Like in above example:Type and Body is Instrinsic data.

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
RoboticDog 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 }
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);