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.

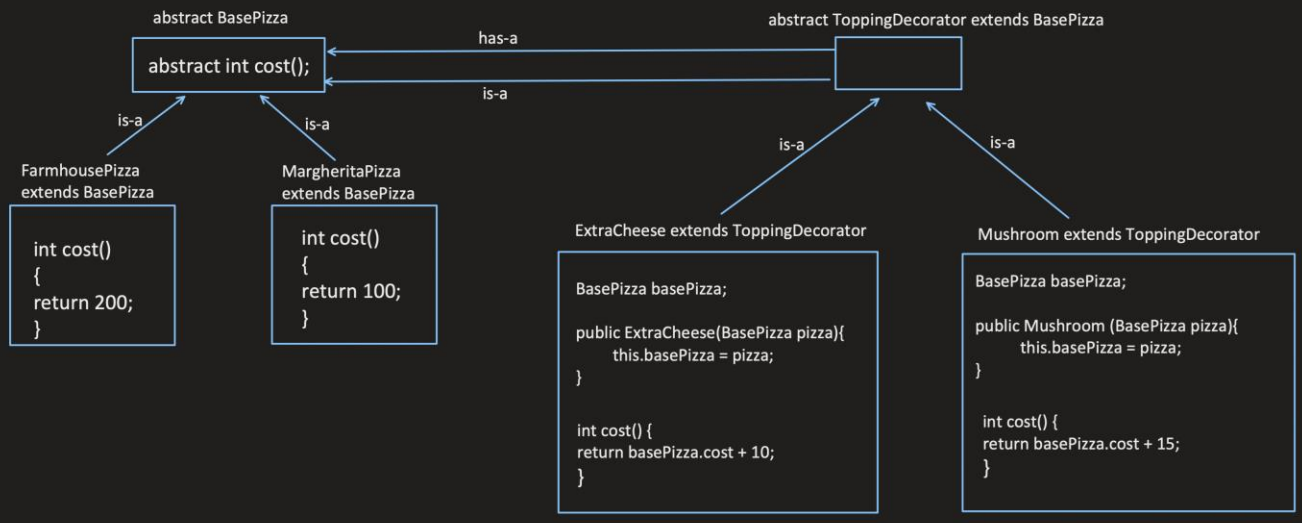
Types:

---

1. Decorator Pattern
2. Proxy Pattern
3. Composite Pattern
4. Adapter Pattern
5. Bridge Pattern
6. Facade
7. Flyweight

1. Decorator Pattern:

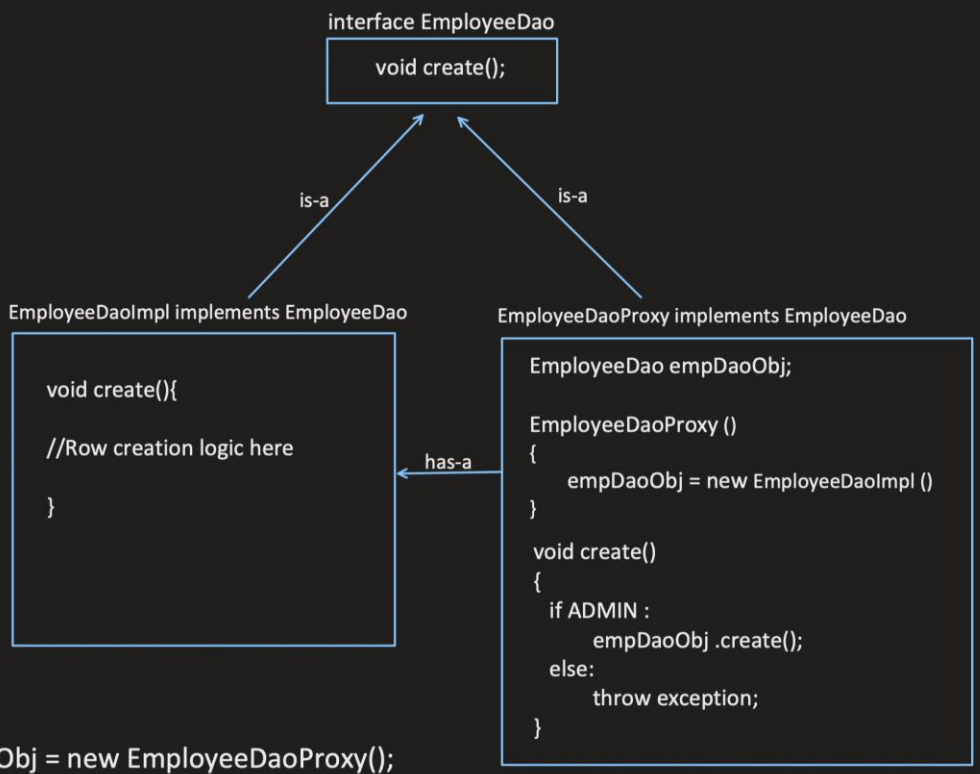
This pattern helps to add more functionality to existing object, without changing its structure.




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

2. Proxy Pattern:

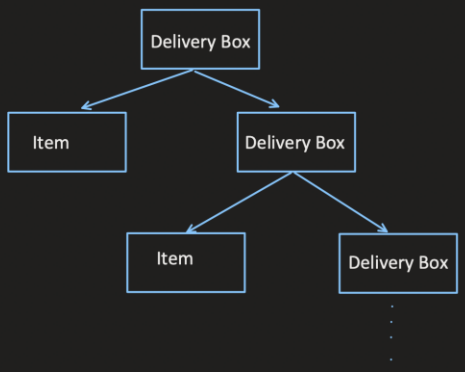
This pattern helps to provide control access to original object.

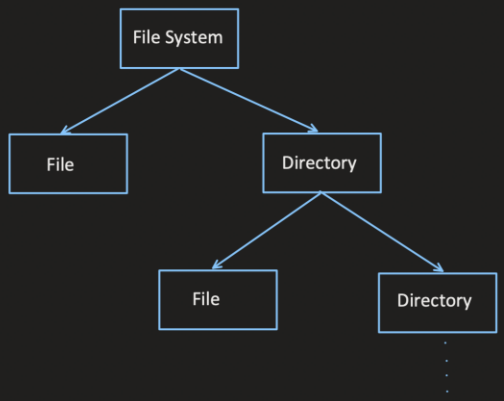


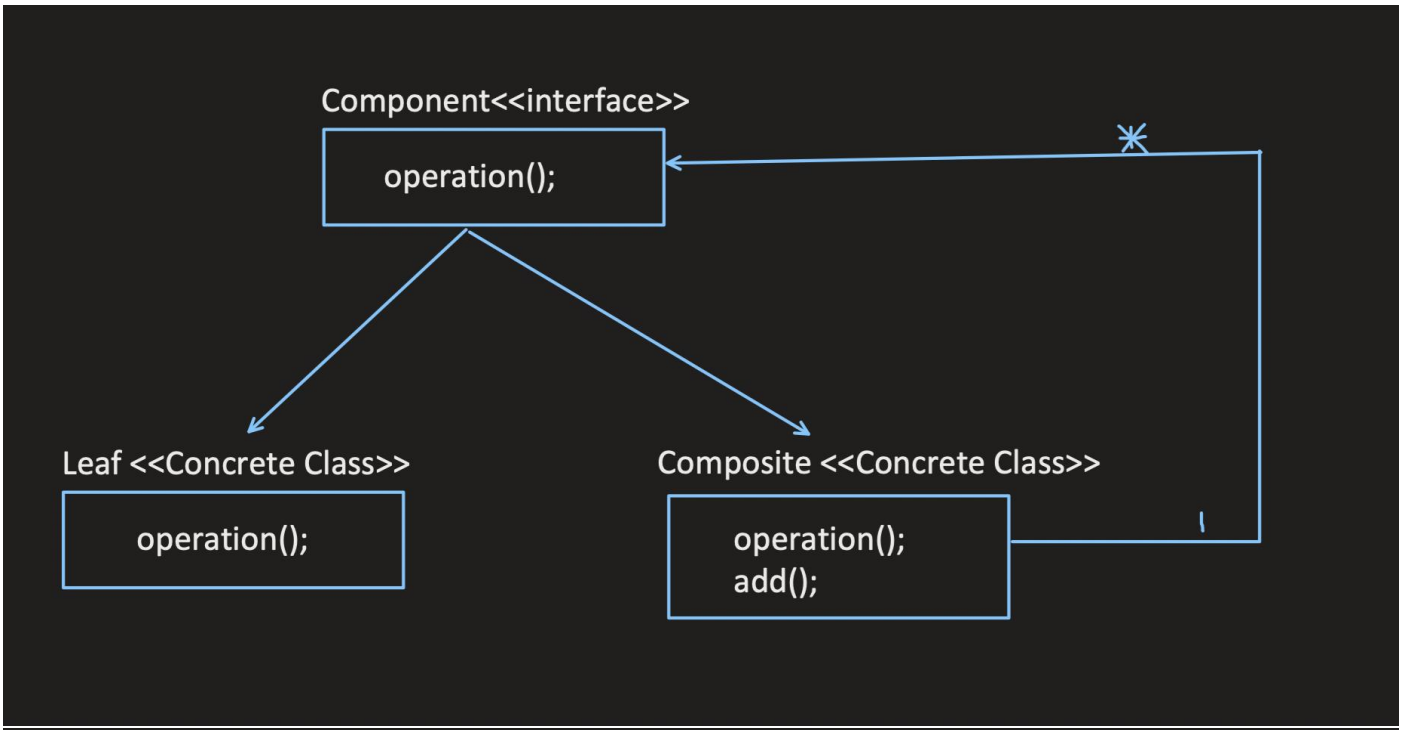

```
EmployeeDao empProxyObj = new EmployeeDaoProxy();
empProxyObj .create();
```

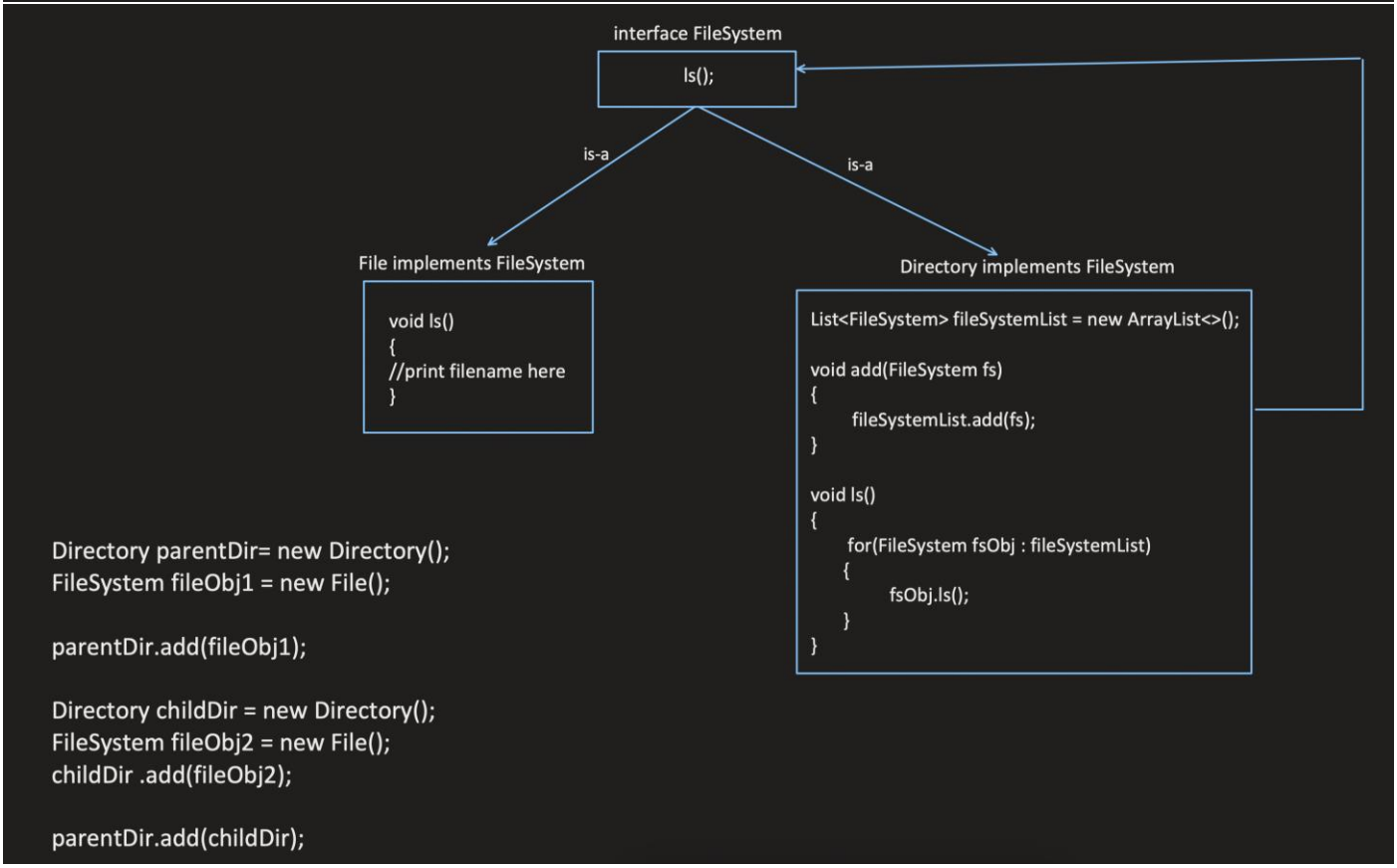
3. Composite Pattern:

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





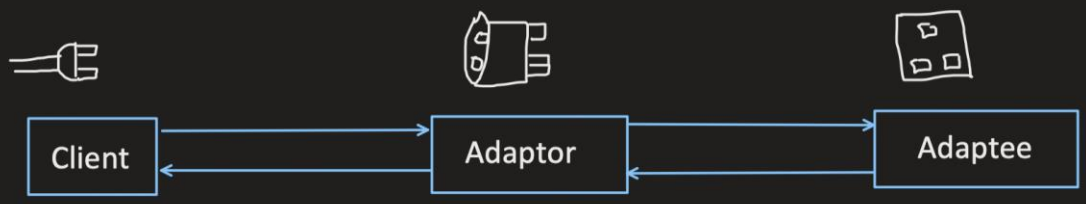





```
Directory parentDir= new Directory();
FileSystem fileObj1 = new File();

parentDir.add(fileObj1);

Directory childDir = new Directory();
FileSystem fileObj2 = new File();
childDir .add(fileObj2);

parentDir.add(childDir);
```

4. Adapter Pattern:

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




```
WeightMachineAdaptor machineAdaptorObj = new WeightMachineAdaptorImpl(new WeightMachineImpl());
machineAdaptorObj .getWeightInKg();
```

interface WeightMachineAdaptor

```
int getWeightInKg();
```

is-a

WeightMachineAdaptorImpl implements WeightMachineAdaptor

```
WeightMachine machine;

WeightMachineAdaptorImpl( WeightMachine machine)
{
        this.machine = machine;
}

int getWeightInKg()
{
   int weightInPound = machine.getWeightInPounds();
   return weightInPound * .45;
}
```

has-a

interface WeightMachine

```
int getWeightInPounds();
```

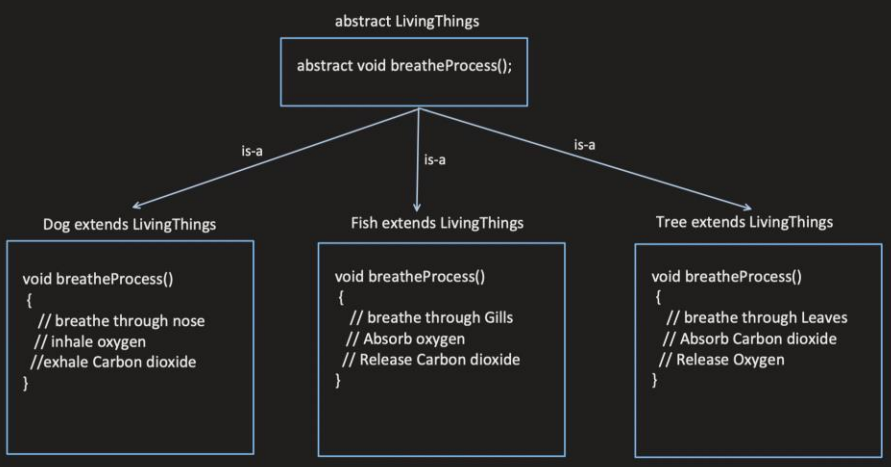
is-a

WeightMachineImpl implements WeightMachine

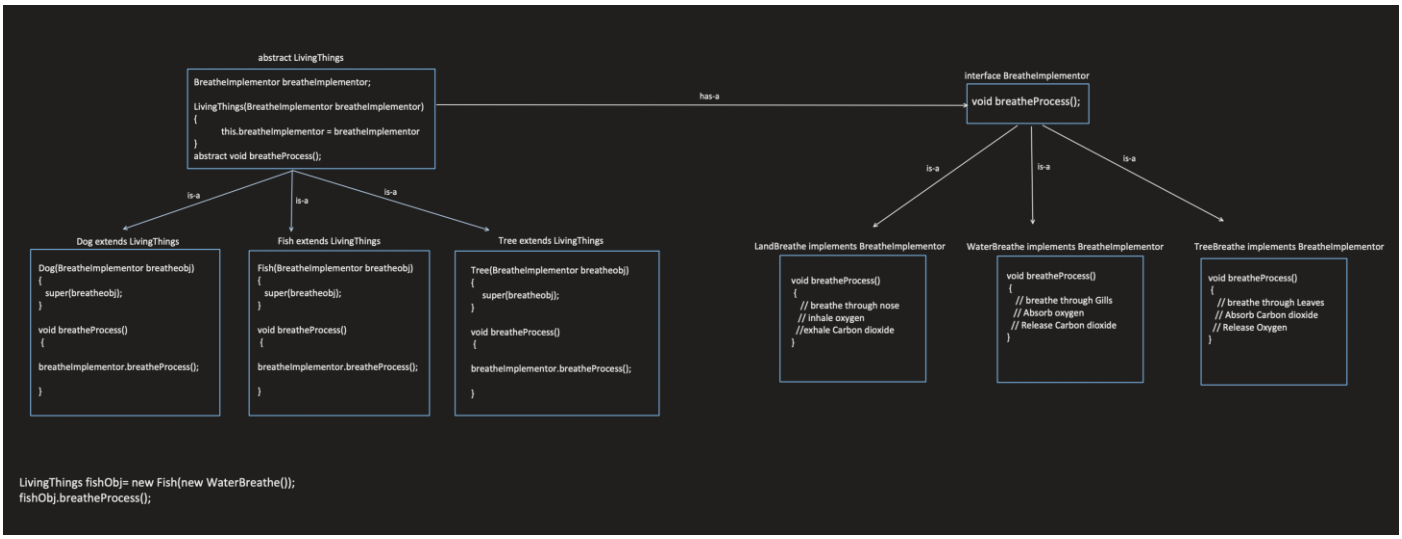
```
int getWeightInPounds()
{
        return 30;
}
```

5. Bridge Pattern:

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



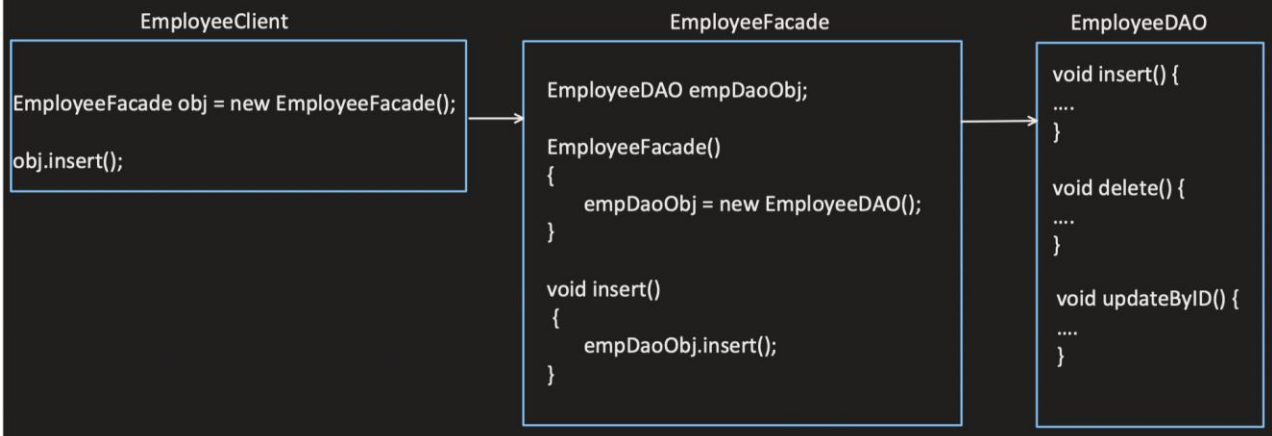
How to add new Breathing Process, Without adding any class of LivingThings?



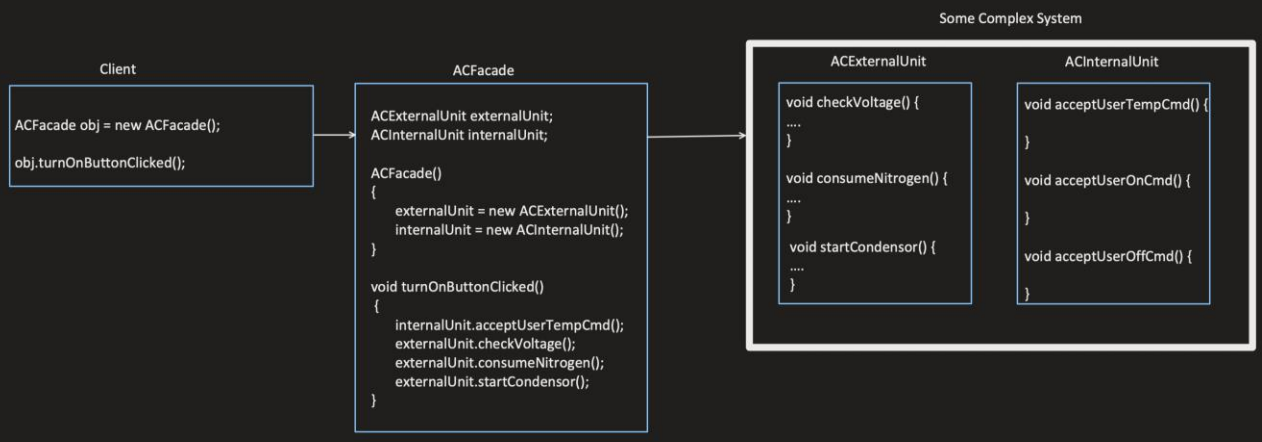
6. Facade Pattern:

This pattern helps to hide the system complexity from the client.

Example1 : expose only the necessary details to the client



Example2 : hide the system complexity from the client



7. Flyweight 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 corrdinateY;        //4bytes
String type;            //50bytes (1bytes * 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

```
int x=0;
int y=0;
for(int i=1; i<5000000; i++)
{
    Sprites humanoidSprite = new Sprites();
    Robot humanoidBotObj = new Robot(x+i; y+i, "HUMANOID", humanoidSprite);

}

for(int i=1; i<5000000; i++)
{
    Sprites roboticDogSprite = new Sprites();
    Robot roboticDobObj = new Robot(x+i; y+i, "ROBOTICDOB", roboticDogSprite);

}
```

= 10Lakh * ~31KB = 31GB

ISSUE as memory is 21GB only

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 whenever required.

RoboticFactory

```
static Map<String, IRobot> roboticObjectCache = new HashMap<>();

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;
}
```

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
}
```

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


IRobot humanoidRobot2 = RoboticFactory.createRobot("HUMANOID");
humanoidRobot2 .display(10, 20);
```