29 All structural Design Patterns

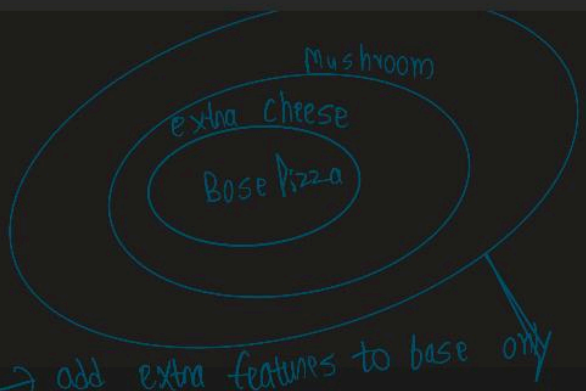
Structural Design Pattern is a way to combine or arrange different classes and objects to form a complex or bigger structure to solve a particular requirement.

25%

Types:

---

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

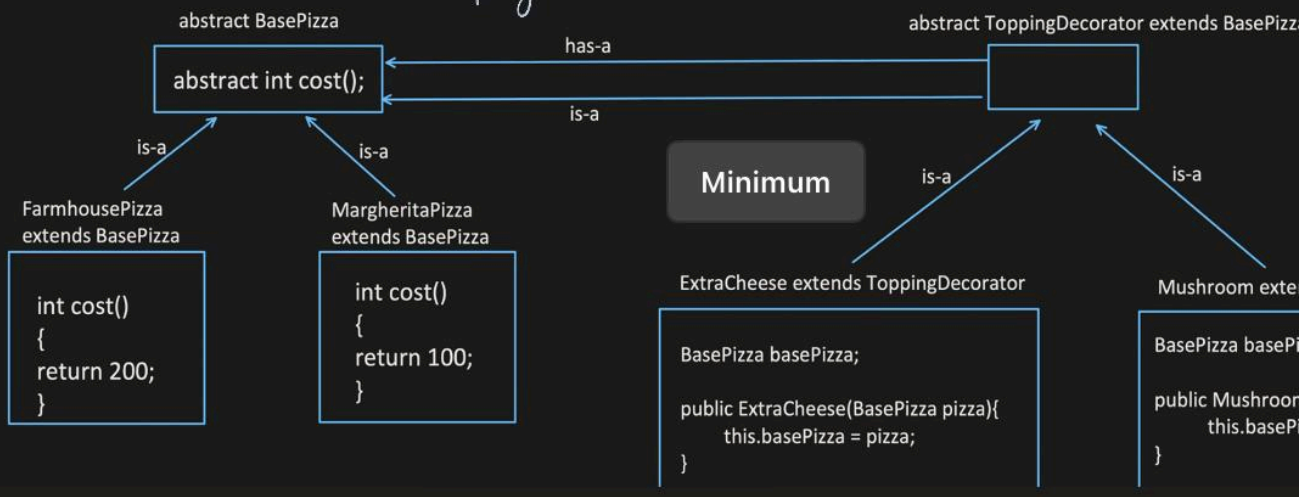


1. Decorator Pattern:

→ add extra features to base only

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

→ seperates base Pizza & topping decorator



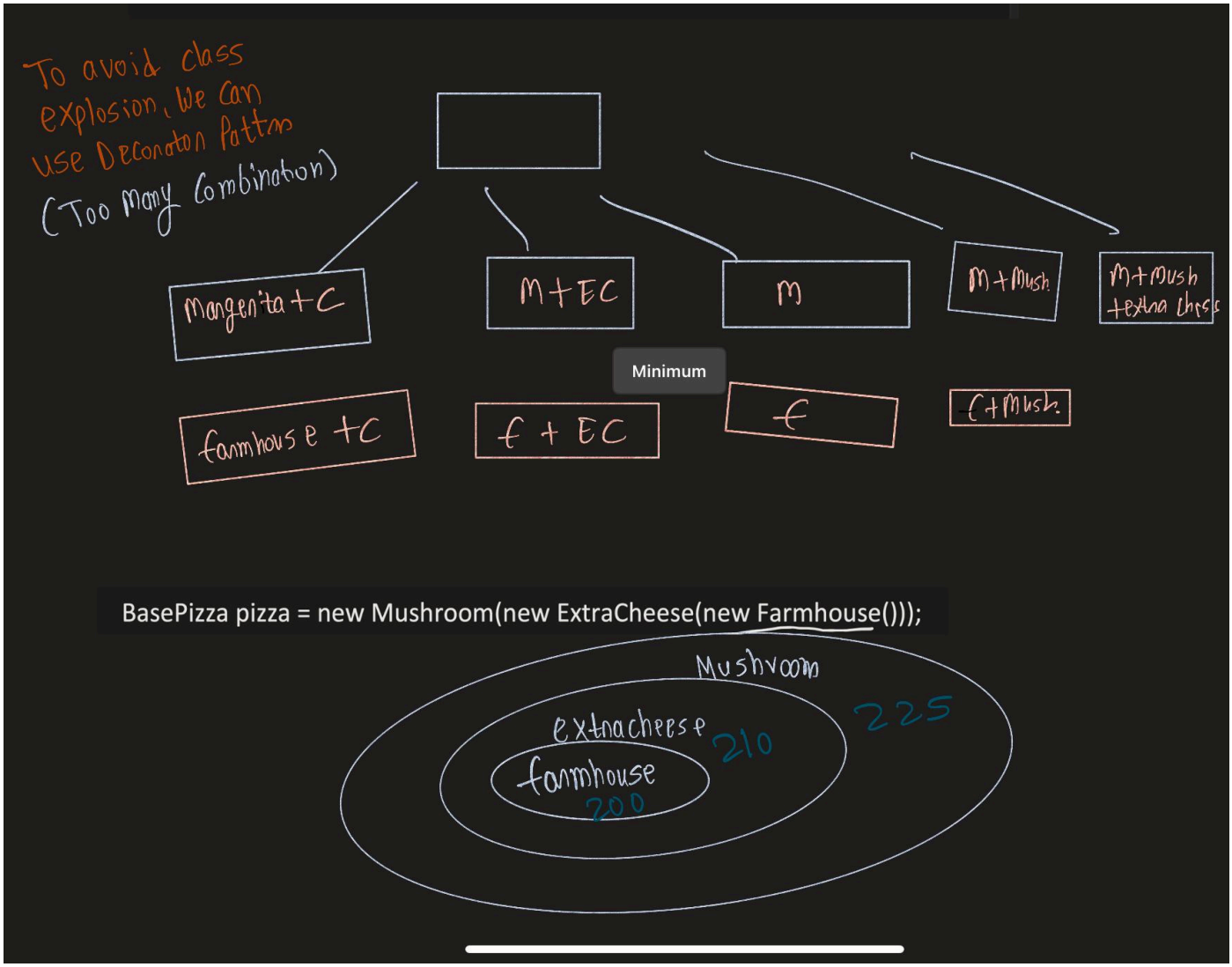
ExtraCheese extends ToppingDecorator

```
BasePizza basePizza;

public ExtraCheese(BasePizza pizza){
      this.basePizza = pizza;
}

int cost() {
return basePizza.cost + 10;
}
```

→ farmhouse or margarita

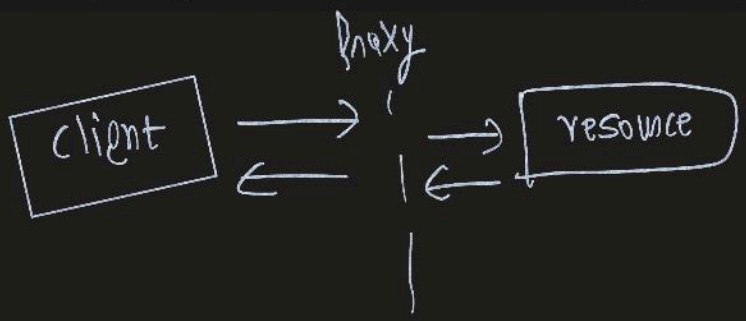
Mushroom extends ToppingDecorator

```
BasePizza basePizza;

public Mushroom (BasePizza pizza){
      this.basePizza = pizza;
}

 int cost() {
 return basePizza.cost + 15;
 }
```


To avoid class explosion, we can use Decorator Pattern
(Too many Combination)
Margerita+C
M+EC
M
M+Mush
M+Mush +extra cheese
Minimum
farmhouse +C
f+EC
f
f+Mush
BasePizza pizza = new Mushroom(new ExtraCheese(new Farmhouse()));
Mushroom
225
extracheese
210
farmhouse
200

2. Proxy Pattern:

This pattern helps to provide control access to original object.



interface EmployeeDao

```
void create();
```

Minimum

is-a

is-a

Original object

EmployeeDaoImpl implements EmployeeDao

```
void create(){

//Row creation logic here

}
```

Proxy has-a Original Obj

EmployeeDaoProxy implements EmployeeDao

```
EmployeeDao empDaoObj;

EmployeeDaoProxy ()
{
    empDaoObj = new EmployeeDaoImpl ()
}

void create()
{
  if ADMIN :
    empDaoObj .create();
  else:
      throw exception;
}
```

Controlled exception

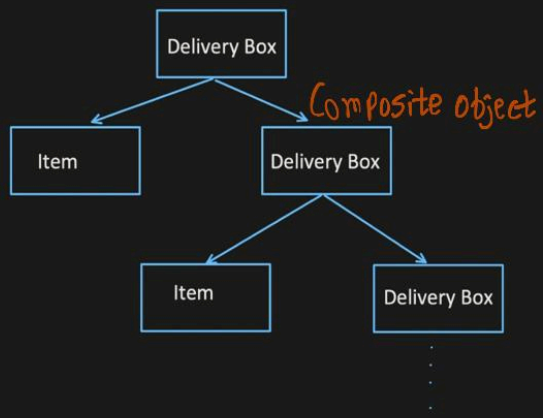
if admin, then invoke the original object else throw exception

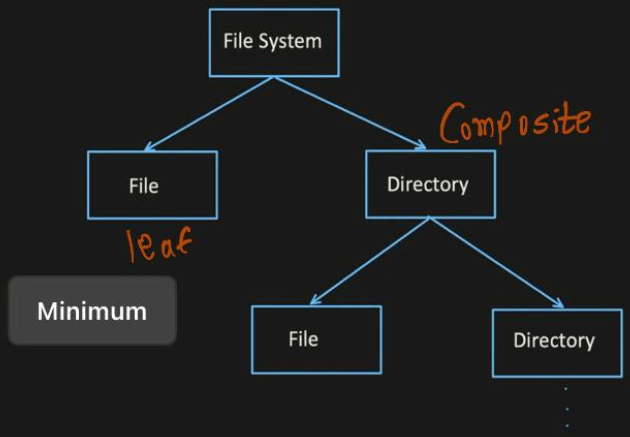
```
EmployeeDao empProxyObj = new EmployeeDaoProxy();
empProxyObj .create();
```

I am creating an object of Proxy & calling Create method

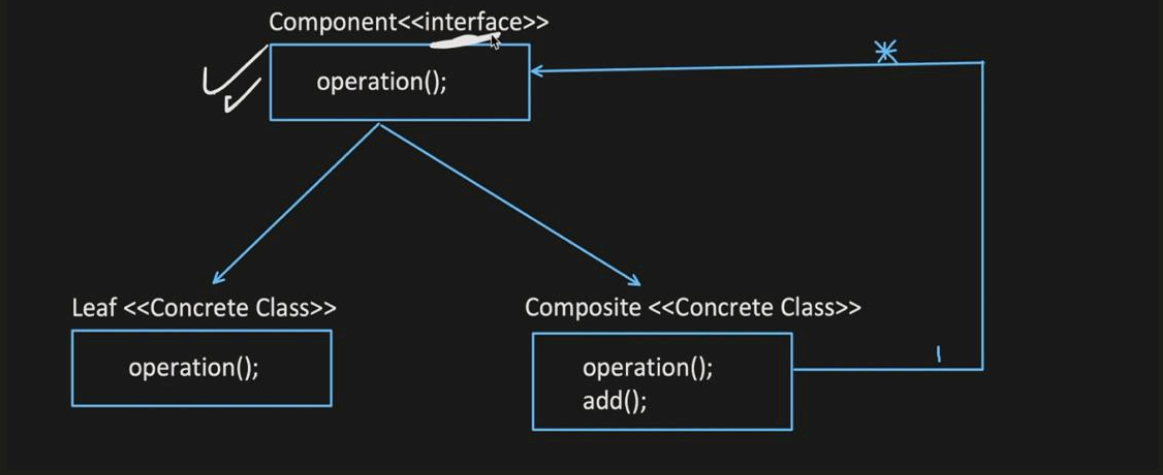
3. Composite Pattern:

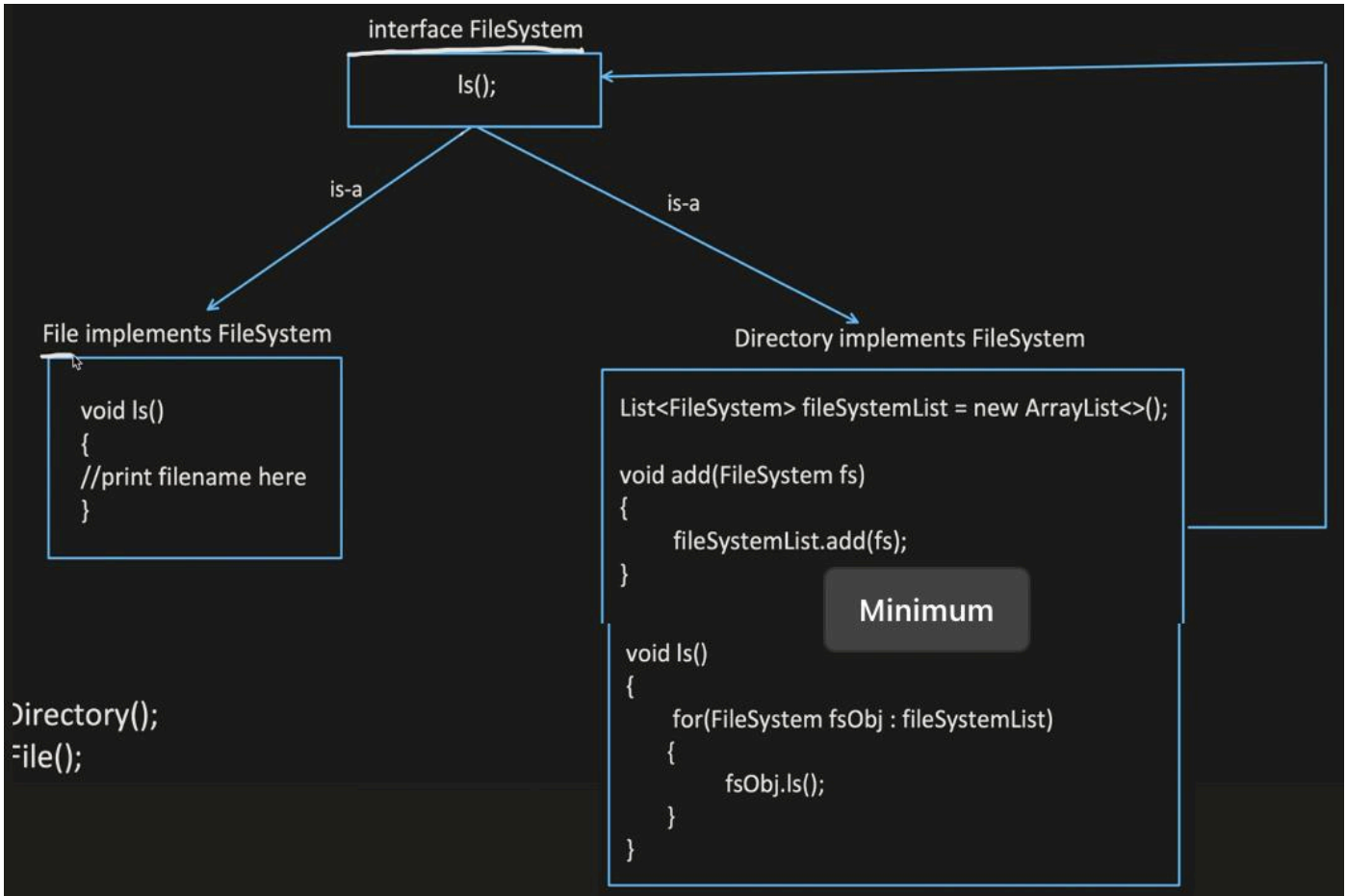
This pattern helps in scenarios where we have OBJECT inside OBJECT (tree like structure)





UML Diagram



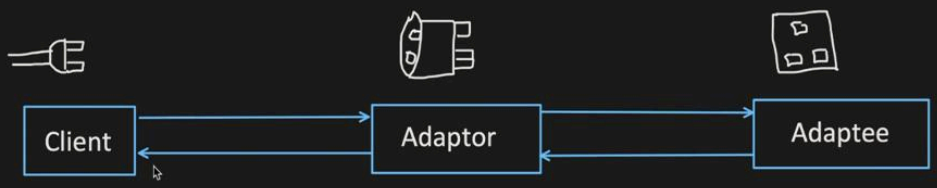


)irectory();
Tile();

```
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();
```

4. Adapter Pattern:

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




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

Minimum

(client)

interface WeightMachineAdaptor

```
int getWeightInKg();
```

is-a

interface WeightMachine (adaptee)

```
int getWeightInPounds();
```

is-a

WeightMachineAdaptorImpl implements WeightMachineAdaptor

```
WeightMachine machine;

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

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

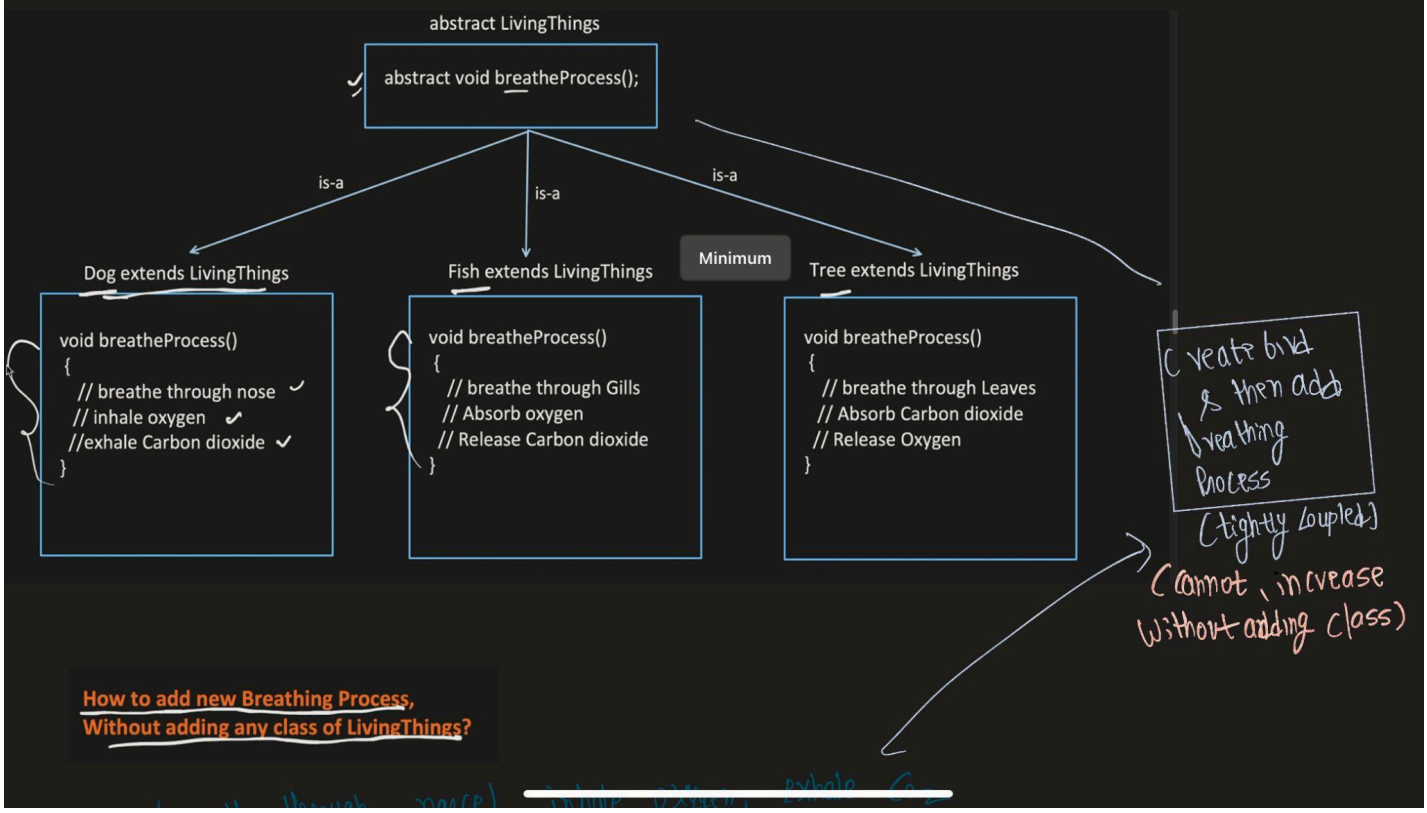
has-a

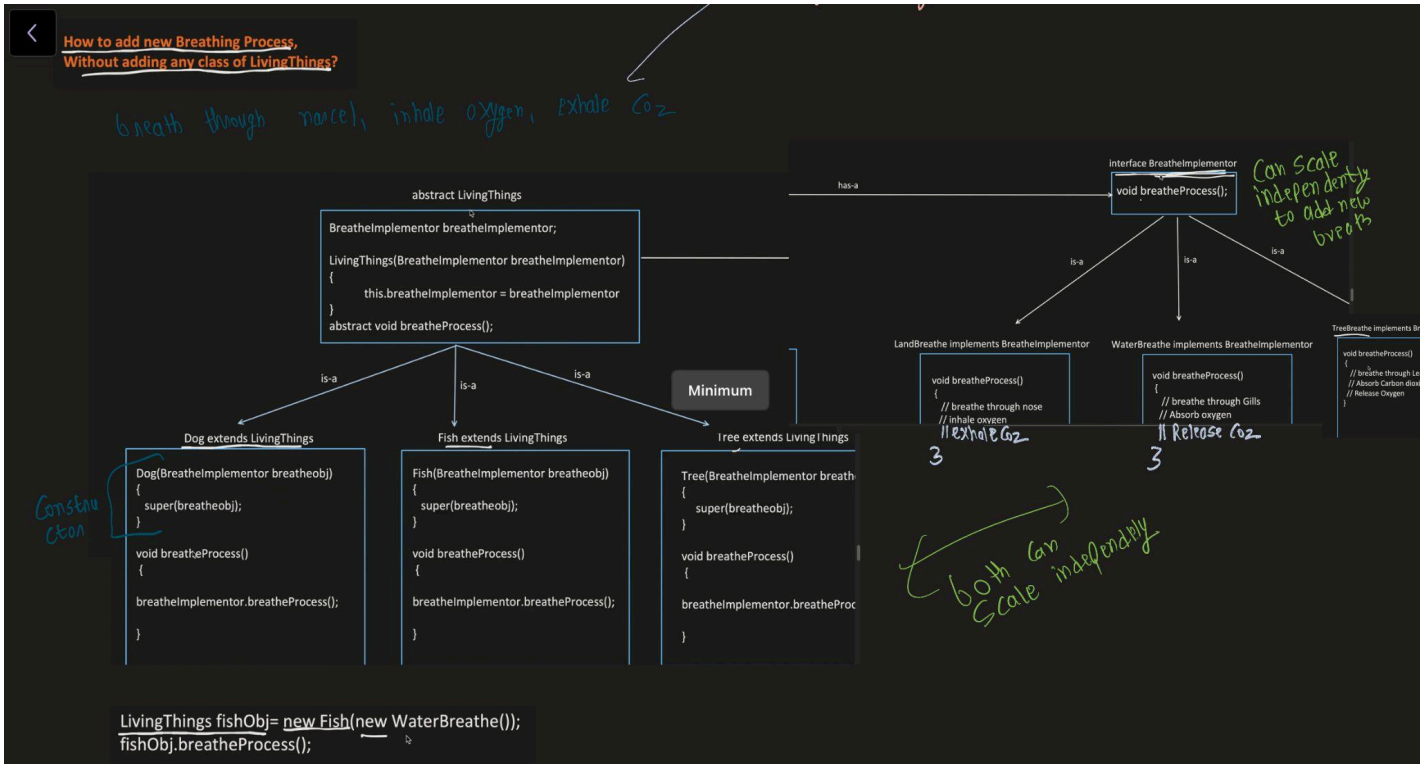
WeightMachineImpl implements V

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

5. Bridge Pattern:

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

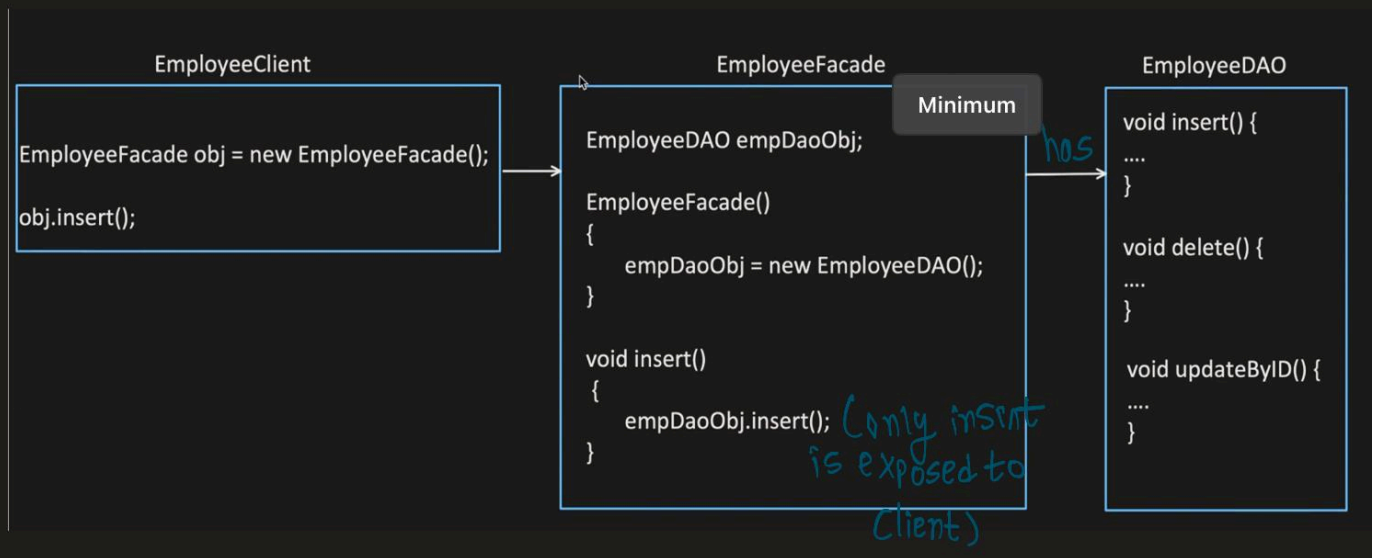



How to add new Breathing Process,
Without adding any class of LivingThings?
breath through nacel, inhale oxygen, Exhale Co2
abstract LivingThings
BreatheImplementor breatheImplementor;
LivingThings(BreatheImplementor breatheImplementor)
{
this.breatheImplementor = breatheImplementor
}
abstract void breatheProcess();
has-a
interface BreatheImplementor
void breatheProcess();
Can Scale independently to add new breaths
is-a
is-a
is-a
is-a
is-a
is-a
Minimum
LandBreathe implements BreatheImplementor
void breatheProcess()
{
// breathe through nose
// inhale oxygen
// exhale Co2
}
3
WaterBreathe implements BreatheImplementor
void breatheProcess()
{
// breathe through Gills
// Absorb oxygen
// Release Co2
}
3
Dog extends LivingThings
Dog(BreatheImplementor breatheobj)
{
super(breatheobj);
}
void breatheProcess()
{
breatheImplementor.breatheProcess();
}
Constructon
Fish extends LivingThings
Fish(BreatheImplementor breatheobj)
{
super(breatheobj);
}
void breatheProcess()
{
breatheImplementor.breatheProcess();
}
Tree extends LivingThings
Tree(BreatheImplementor breath
{
super(breatheobj);
}
void breatheProcess()
{
breatheImplementor.breatheProc
}
both can Scale independtly
LivingThings fishObj= new Fish(new WaterBreathe());
fishObj.breatheProcess();

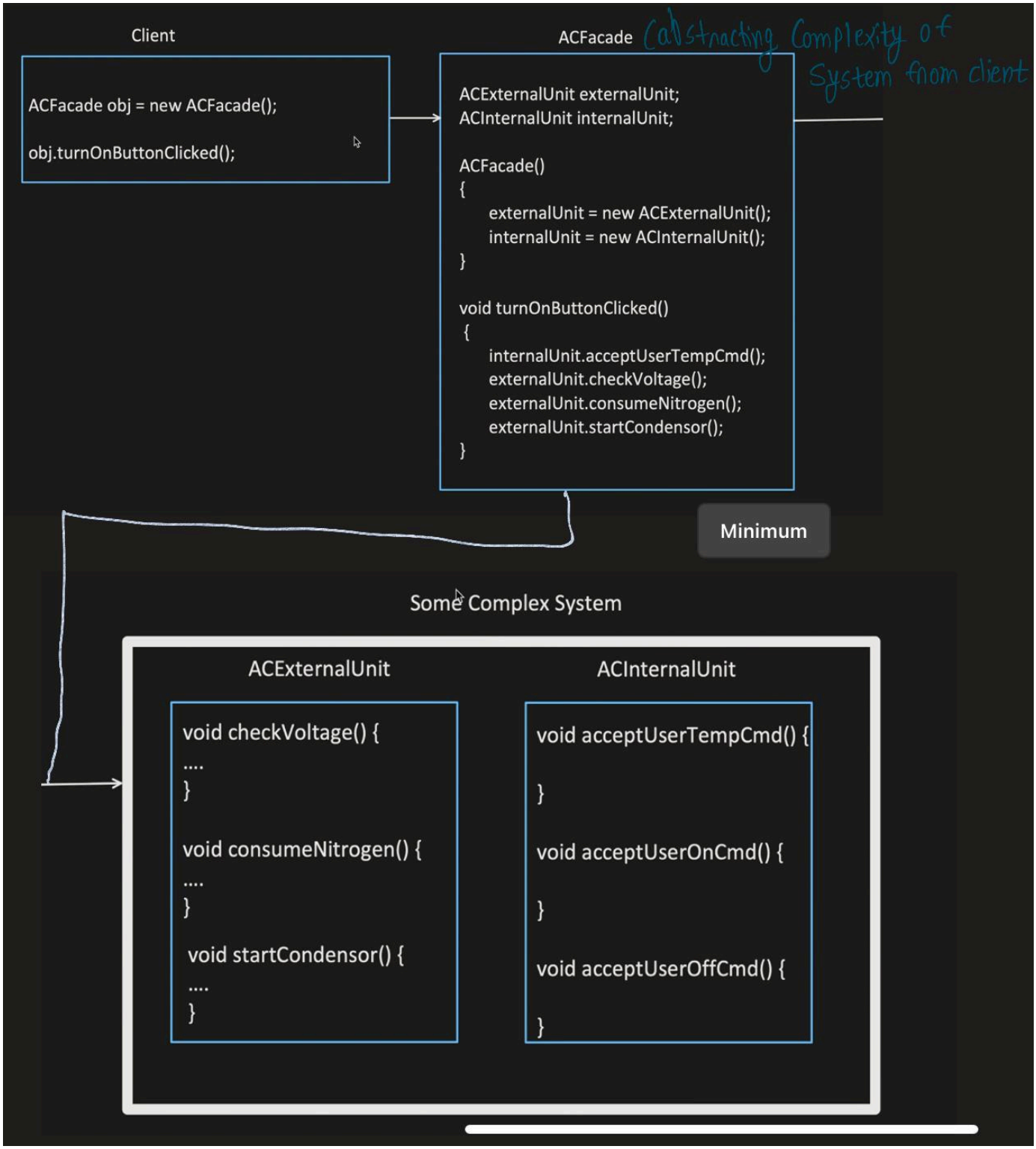
6. Facade Pattern:

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


Client
ACFacade obj = new ACFacade();
obj.turnOnButtonClicked();
ACFacade
Abstracting Complexity of System from client
ACExternalUnit externalUnit;
ACInternalUnit internalUnit;
ACFacade()
{
externalUnit = new ACExternalUnit();
internalUnit = new ACInternalUnit();
}
void turnOnButtonClicked()
{
internalUnit.acceptUserTempCmd();
externalUnit.checkVoltage();
externalUnit.consumeNitrogen();
externalUnit.startCondensor();
}
Minimum
Some Complex System
ACExternalUnit
void checkVoltage() {
....
}
void consumeNitrogen() {
....
}
void startCondensor() {
....
}
ACInternalUnit
void acceptUserTempCmd() {
}
void acceptUserOnCmd() {
}
void acceptUserOffCmd() {
}

7. Flyweight Pattern:

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

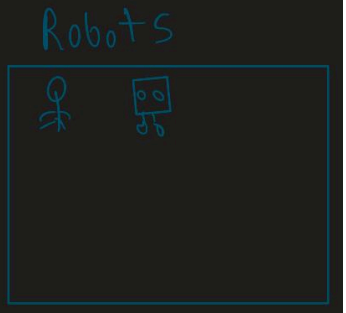
Issue: lets say memory is 21GB

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;
}
```

Minimum

= ~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);

}
```

→ loop for 5 lakhs time

= 10Lakh * ~31 KB

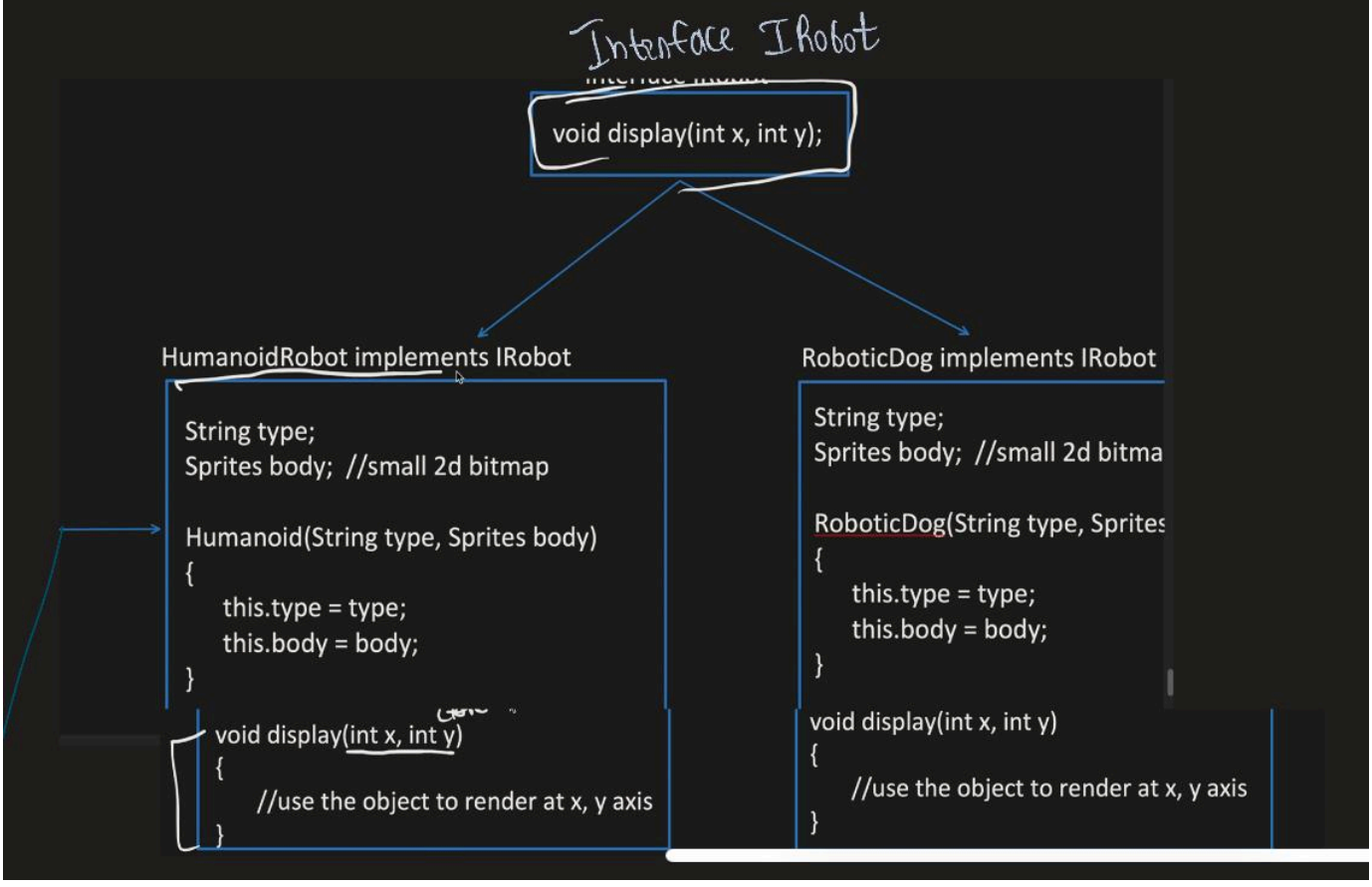
≈ 31 GB
(We have only 21GB)

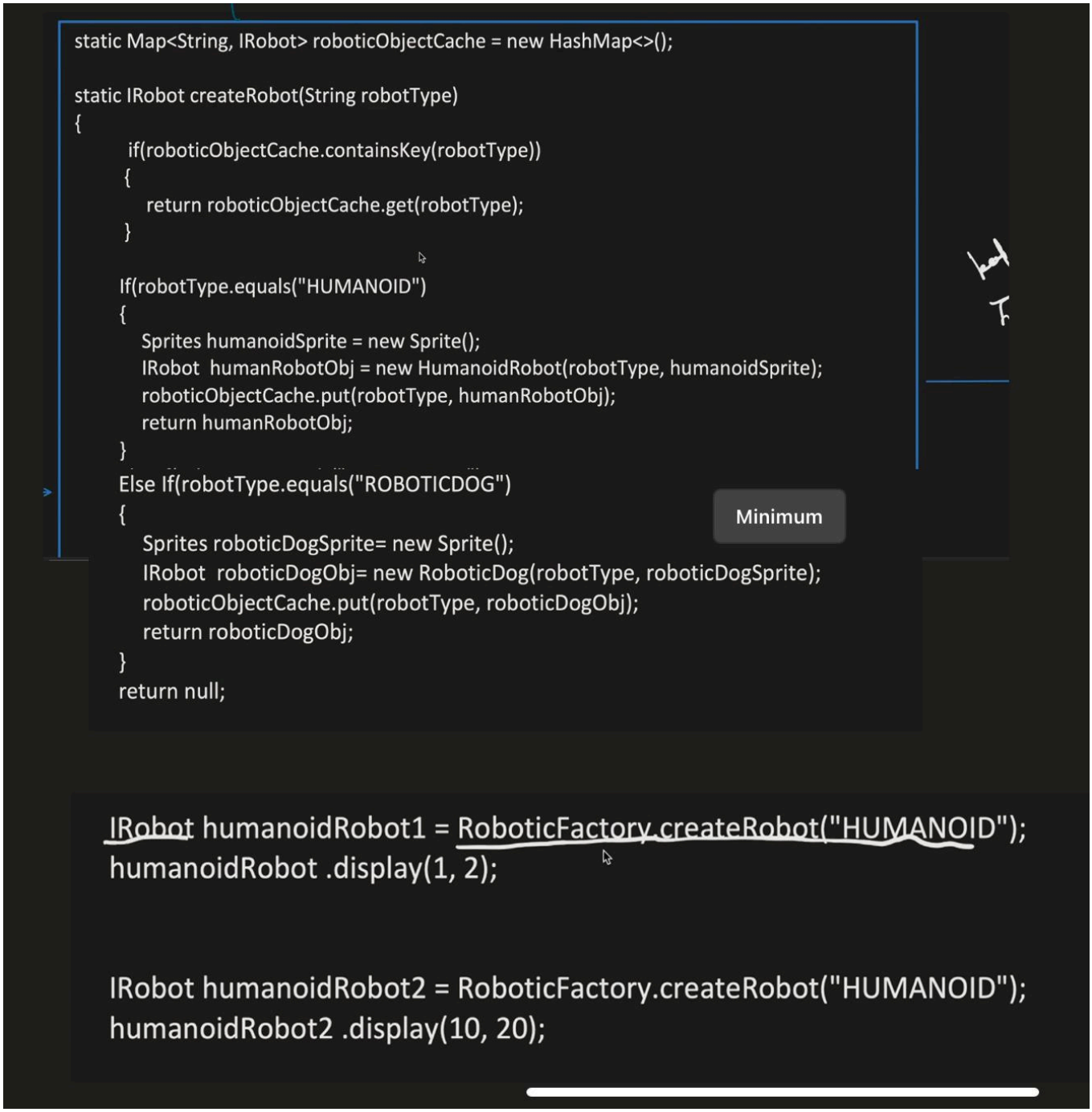
Intrinsic data: shared among objects and remain same once defined one value.
Like in above example : Type and Body is ***Instrinsic*** data.

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.

Minimum





```
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;
```

Minimum

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


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