Design Patterns

Design patterns are reusable solutions for common system design problems that help developers build cleaner and maintainable systems.

Types include:
Creational, Structural, Behavioral

Creational Design Patterns:

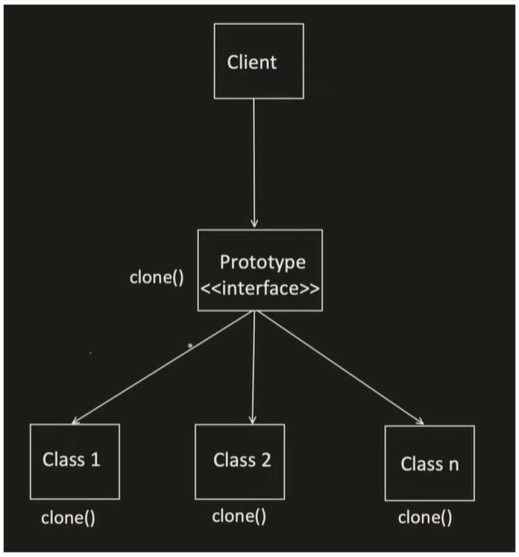
Creational Design Pattern responsibility is to create objects / control the creation of objects.

Types include:

Prototype, Singleton, Factory, Abstract Factory, Builder.

Prototype:

It is used when we have to make copy / clone from existing object when the object creation is more expensive than copying it.



Example:

```
class Shape {
public:
    virtual void draw() = 0;
    virtual unique_ptr<Shape> clone() const = 0;
    virtual ~Shape() {}
};
```

```
class Circle : public Shape {
    int radius;
public:
    Circle(int r) : radius(r) {}
    void draw() override {
        cout << "Drawing Circle with radius " << radius << "\n";
    }
```

```
    unique_ptr<Shape> clone() const override {
        return make_unique<Circle>(*this);   // deep copy using copy
constructor
    }
};
```

Singleton:

It is used when we have to create only 1 instance of the class.

Implementations – Eager, Lazy, Synchronized, Double Locking

Eager – Example: Early initialization

```
class Singleton {
private:
    static Singleton instance;   // created eagerly
    Singleton() {}
    ~Singleton() {}

public:
    static Singleton& getInstance() {
        return instance;
    }
};

Singleton Singleton::instance;   // definition
```

Lazy – Example: Lazy initialization (not Thread safe)

```
class Singleton {
private:
    static Singleton* instance;
    Singleton() {}
    ~Singleton() {}

public:
    static Singleton* getInstance() {
        if (instance == nullptr)
            instance = new Singleton();
        return instance;
    }
};

Singleton* Singleton::instance = nullptr;
```

Synchronized – Example:

```
#include <mutex>

class Singleton {
private:
    static Singleton* instance;
    static std::mutex mtx;

    Singleton() {}
    ~Singleton() {}
```

```
public:
    static Singleton* getInstance() {
        std::lock_guard<std::mutex> lock(mtx);   // synchronized
        if (instance == nullptr)
            instance = new Singleton();
        return instance;
    }
};

Singleton* Singleton::instance = nullptr;
std::mutex Singleton::mtx;
```

Double Locking – Example:

```
#include <mutex>

class Singleton {
private:
    static Singleton* instance;
    static std::mutex mtx;

    Singleton() {}
    ~Singleton() {}

public:
    static Singleton* getInstance() {
        if (instance == nullptr) {                  // 1st check (no lock)
            std::lock_guard<std::mutex> lock(mtx);
            if (instance == nullptr)                // 2nd check (safe)
                instance = new Singleton();
        }
        return instance;
    }
};

Singleton* Singleton::instance = nullptr;
std::mutex Singleton::mtx;
```

Best – Example:

```
class Singleton {
private:
    Singleton() {}
    ~Singleton() {}

public:
    static Singleton& getInstance() {
        static Singleton instance;   // thread-safe in C++11+
        return instance;
    }
};
```

Java Implementations:
Eager:

```java
public class DBConnection {

    private static DBConnection conObject = new DBConnection();

    private DBConnection(){
    }

    public static DBConnection getInstance(){
        return conObject;
    }
}
```

```java
public class Main {

    public static void main(String args[]) {

      DBConnection connObject = DBConnection.getInstance();
    }

}
```

Lazy: (no Thread safe)

```java
public class DBConnection {

    private static DBConnection conObject;

    private DBConnection(){
    }

    public static DBConnection getInstance(){

        if(conObject == null){
            conObject = new DBConnection();
        }
        return conObject;
    }
}
```

Synchronized:

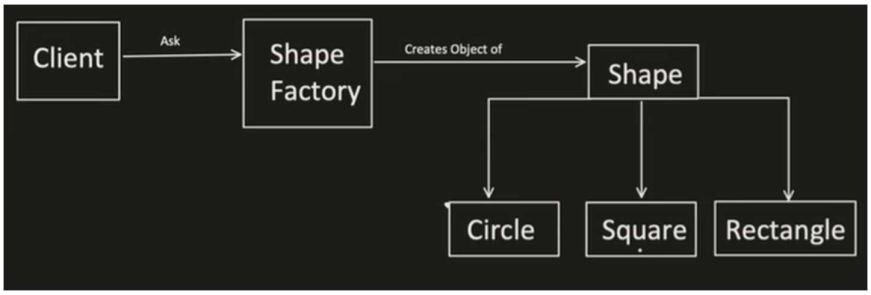
```java
public class DBConnection {

    private static DBConnection conObject;

    private DBConnection(){
    }

    synchronized public static DBConnection getInstance(){

        if(conObject == null){
            conObject = new DBConnection();
        }
        return conObject;
    }
}
```

Double Locking:

```java
public class DBConnection {

    private static DBConnection conObject;

    private DBConnection(){
    }

     public static DBConnection getInstance(){

        if(conObject == null){
            synchronized (DBConnection.class){
                if(conObject == null){
                    conObject = new DBConnection();
                }
            }
        }
        return conObject;
    }
}
```

Factory Pattern:

It is used when the object creation and its business logic are needed to be at one place.



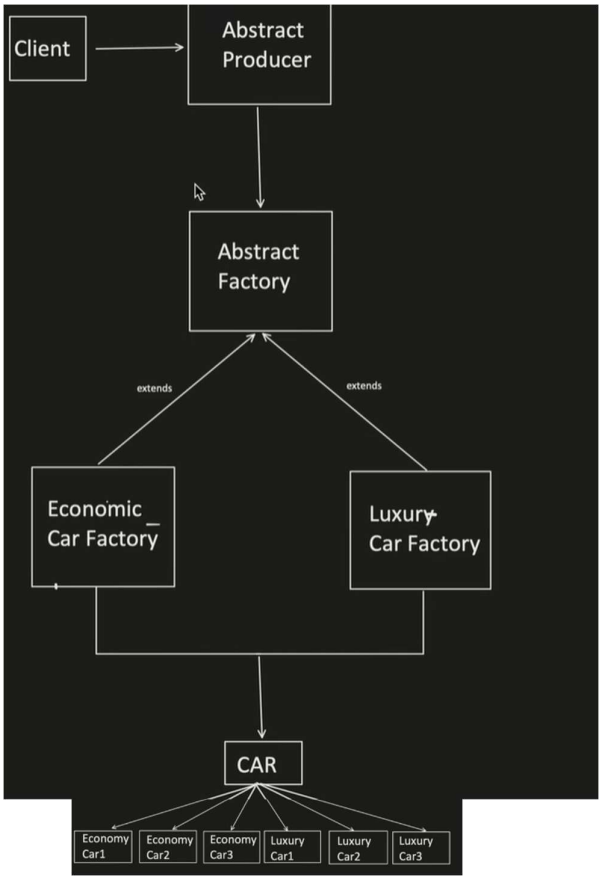
In the above example shape is an interface and shape factory is a class that can return objects of shape based on given input.

Example:

```java
public class ShapeInstanceFactory {

    public Shape getShapeInstance(String value){

        if(value.equals("Circle")){
            return new Circle();
        }
        else if(value.equals("Square")){
            return new Square();
        }
        else if (value.equals("Rectangle")){
            return new Rectangle();
        }
        return null;
    }
}
```

Abstract Factory:

It’s a Factory of Factory’s

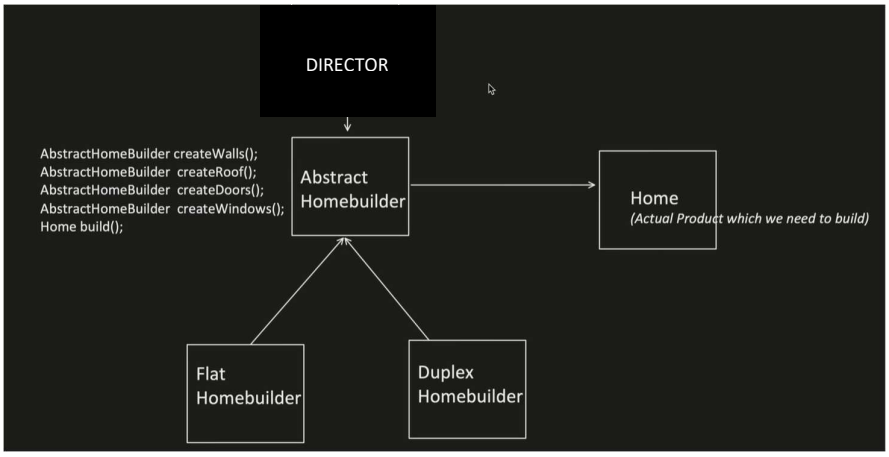


In the above Abstract Factory is an interface with a method public car getInstance()

Economic car Factory, and Luxury car Factory are Factory’s which return objects of class Car based on input given, Abstract Producer is the Parent Factory which returns objects of class Abstract Factory based on input given.

Builder Pattern:

It is used when you want to create objects step by step.

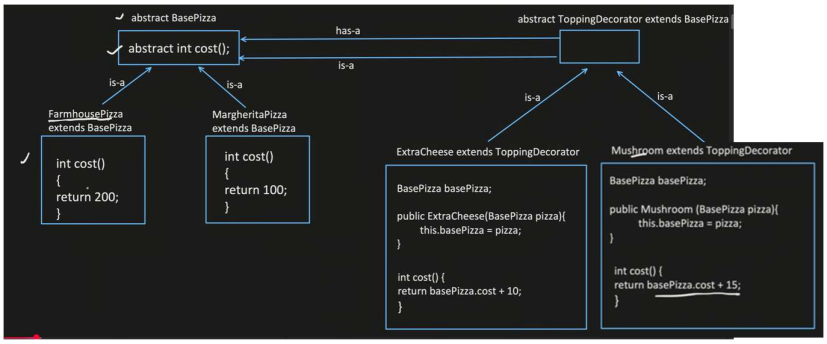


Structural Design Patterns:

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

Types Include – Decorator, Proxy, Composite, Adapter, Bridge, Façade, Flyweight

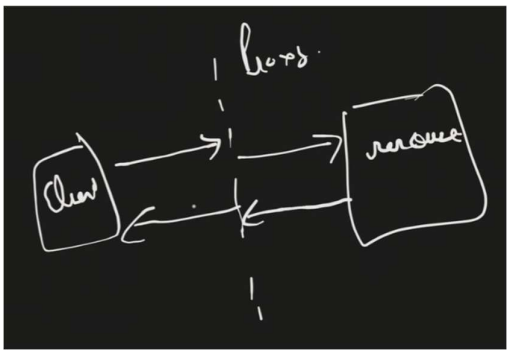
Decorator:
This pattern helps to add more functionality to existing object, without changing its structure.

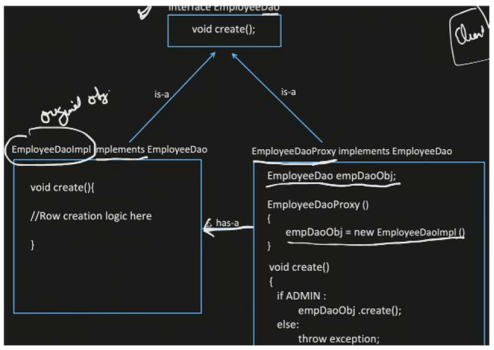


BasePizza pizza = new Mushroom(new ExtarCheese(new Farmhouse()));

Proxy:

This pattern helps to provide control access to original object. DAO(Direct Access Object)





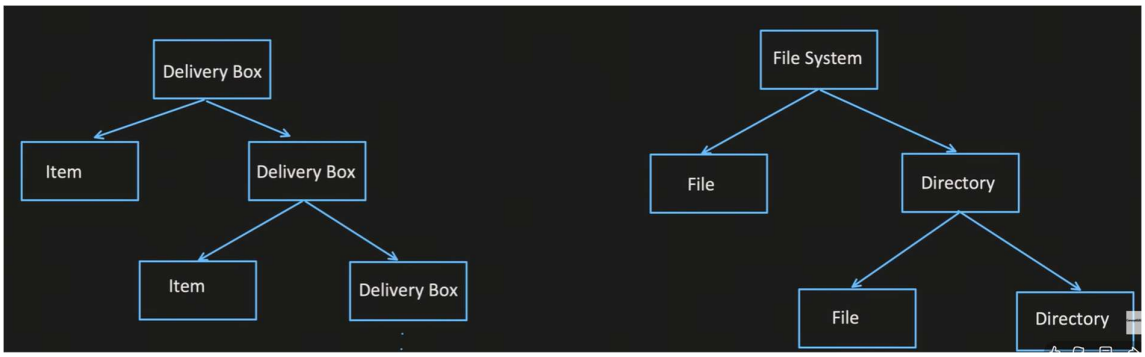
EmployeeDao empProxyObj = new EmployeeDaoProxy();

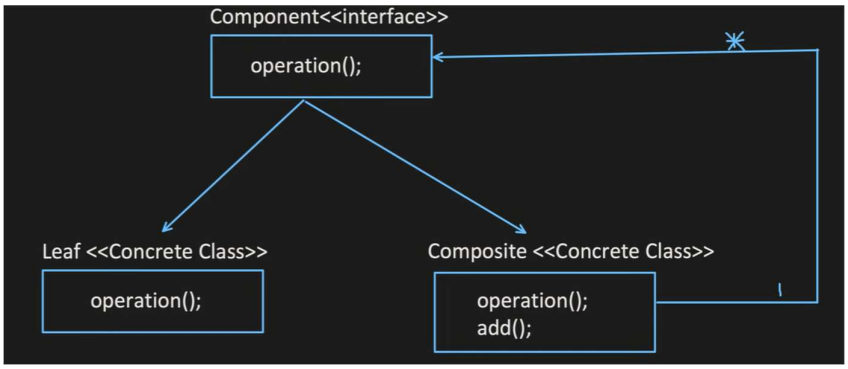
empProxyObj.create();

Proxy is like a intermediate b/w the resource and client so every request and response must pass through it.

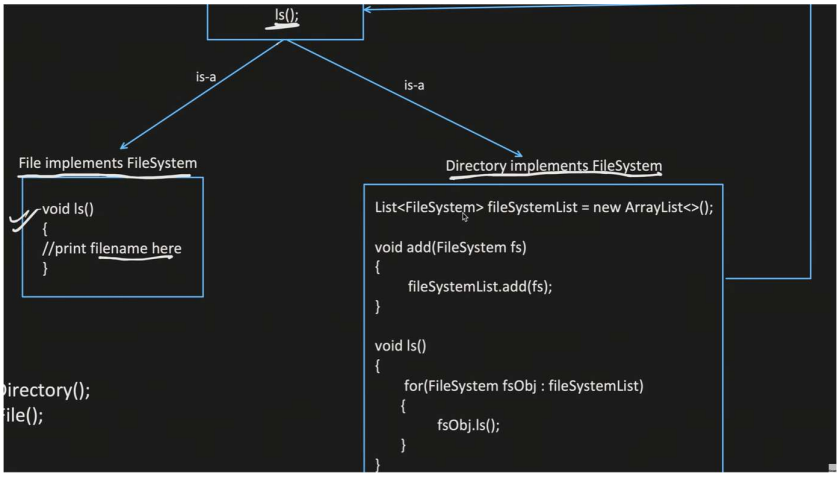
Composite:

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





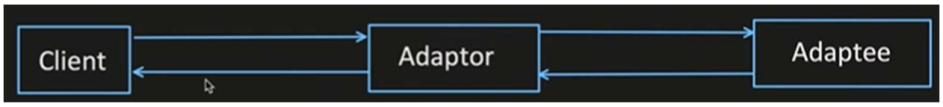
Abstract FileSystem

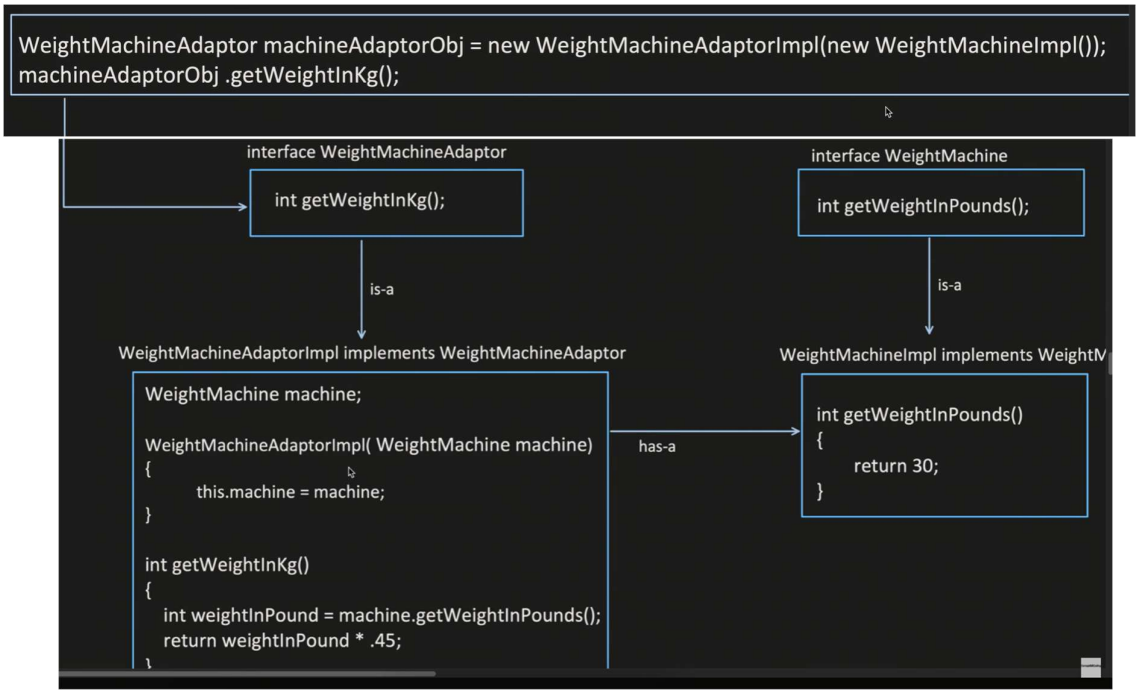



```
Directory parentDir= new Directory();
FileSystem fileObj1 = new File();

parentDir.add(fileObj1);

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

parentDir.add(childDir);

parentDir.ls();
```

Adapter:

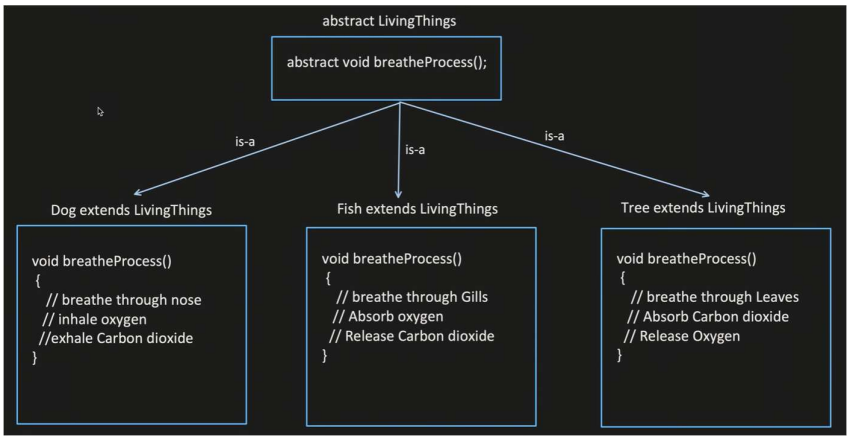
This pattern acts as a bridge or intermediate between 2 incompatible interfaces.



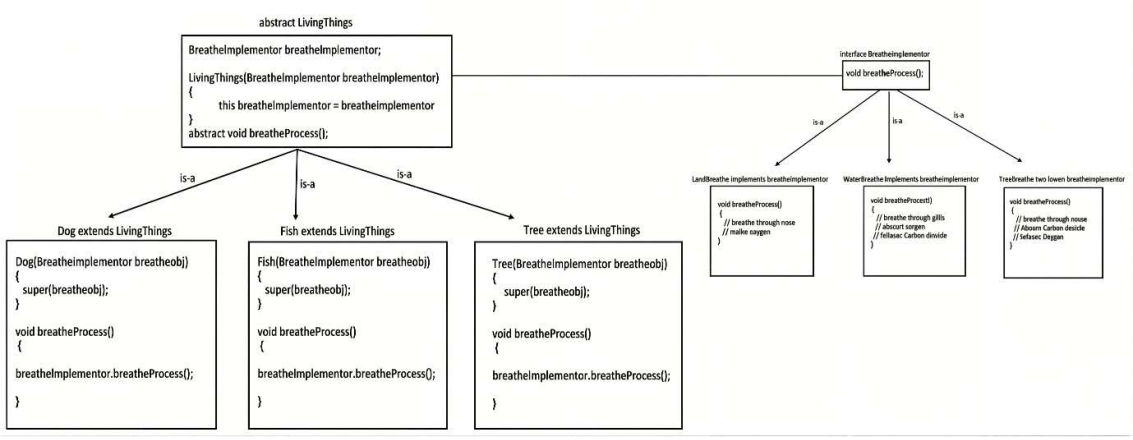


Bridge:

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?


abstract LivingThings
BreatheImplementor breatheImplementor;
LivingThings(BreatheImplementor breatheImplementor)
{
this breatheImplementor = breatheImplementor
}
abstract void breatheProcess();
interface BreatheinpIementor
void breatheProcess();
is-a
is-a
is-a
is-a
is-a
is-a
Dog extends LivingThings
Dog(Breatheimplementor breatheobj)
{
super(breatheobj);
}
void breatheProcess()
{
breatheImplementor.breatheProcess();
}
Fish extends LivingThings
Fish(BreatheImplementor breatheobj)
{
super(breatheobj);
}
void breatheProcess()
{
breatheImplementor.breatheProcess();
}
Tree extends LivingThings
Tree(BreatheImplementor breatheobj)
{
super(breatheobj);
}
void breatheProcess()
{
breatheImplementor.breatheProcess();
}
LandBreathe implements breatheImplementor
void breatheProcess()
{
// breathe through nose
// maike oaygen
}
WaterBreathe Implements breatheimplementor
void breatheProcestt)
{
// breathe through gills
// abscurt sorgen
// fellseac Carbon dinxide
}
TreeBreathe two lowen breatheimplementor
void breatheProcess()
{
// breathe through nouse
// Absorn Carbon desicle
// fellseac Deggen
}