Table of Contents

LLD: Object Pool Design Pattern

Category: Creational Design Pattern

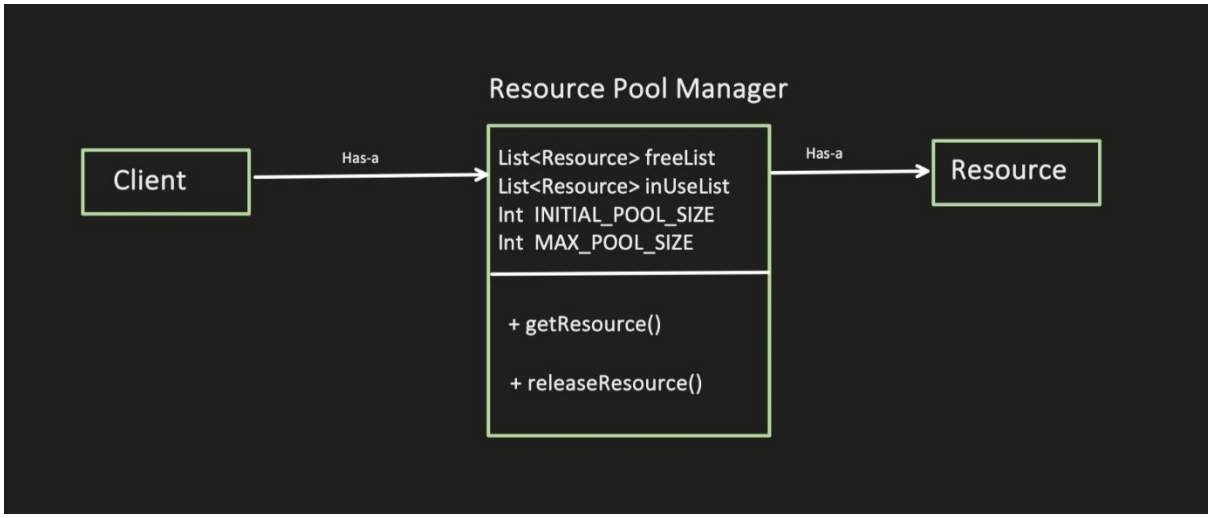
- Manages the pool of reusable objects like DBConnection object.
- Borrow from the pool -> use it -> then return it back to the pool.

Advantages:

- Reduce the overhead of creating and destroying the frequently required object *(generally resource intensive objects)*
- Reduce the latency, as it uses the pre initialized object.
- Prevent Resource exhaustion by managing the number of resource intensive object creation.

Disadvantages:

- Resource Leakage can happen, if object is not handled properly and not being returned to the pool.
- Required more memory because of managing the pool.
- Pool management required thread safety, which is additional overhead.
- Adds application complexity because of managing the pool.



Many engineers makes 1 mistake while coding for this design pattern?

```java
public class Client {

    public static void main(String args[]){

        DBConnectionPoolManager poolManager = new DBConnectionPoolManager();

        DBConnection dbConnection1 = poolManager.getDBConnection();
        DBConnection dbConnection2 = poolManager.getDBConnection();
        DBConnection dbConnection3 = poolManager.getDBConnection();
        DBConnection dbConnection4 = poolManager.getDBConnection();
        DBConnection dbConnection5 = poolManager.getDBConnection();
        DBConnection dbConnection6 = poolManager.getDBConnection();
        poolManager.getDBConnection();
        poolManager.releaseDBConnection(dbConnection6);

    }
}
```

```java
public class DBConnectionPoolManager {

    List<DBConnection> freeConnectionsInPool = new ArrayList<>();
    List<DBConnection> connectionsCurrentlyInUse = new ArrayList<>();
    int INITIAL_POOL_SIZE = 3;
    int MAX_POOL_SIZE = 6;

    public DBConnectionPoolManager() {
        for (int i = 0; i < INITIAL_POOL_SIZE; i++) {
            freeConnectionsInPool.add(new DBConnection());
        }
    }

    public DBConnection getDBConnection() {
        if (freeConnectionsInPool.isEmpty() && connectionsCurrentlyInUse.size() < MAX_POOL_SIZE) {
            freeConnectionsInPool.add(new DBConnection());
            System.out.println("creating new connection and putting into the pool, free pool size: " + freeConnectionsInPool.size());
        } else if (freeConnectionsInPool.isEmpty() && connectionsCurrentlyInUse.size() >= MAX_POOL_SIZE) {
            System.out.println("can not create new DBConnection, as max limit reached");
            return null;
        }
        DBConnection dbConnection = freeConnectionsInPool.remove( index: freeConnectionsInPool.size() - 1);
        connectionsCurrentlyInUse.add(dbConnection);
        System.out.println("Adding db connection into Use pool, size: " + connectionsCurrentlyInUse.size());
        return dbConnection;
    }

    public void releaseDBConnection(DBConnection dbConnection) {
        if (dbConnection != null) {
            connectionsCurrentlyInUse.remove(dbConnection);
            System.out.println("Removing db connection from Use pool, size: " + connectionsCurrentlyInUse.size());
            freeConnectionsInPool.add(dbConnection);
            System.out.println("Adding db connection into free pool, size: " + freeConnectionsInPool.size());
        }
    }
}
```

```java
public class DBConnection {

    Connection mysqlConnection;

    DBConnection() {
        try {
            mysqlConnection = DriverManager.getConnection( url: "url",  user: "username",  password: "password");
        } catch (Exception e) {
            //handle exception here
        }
    }
}
```

What's wrong with the above code?

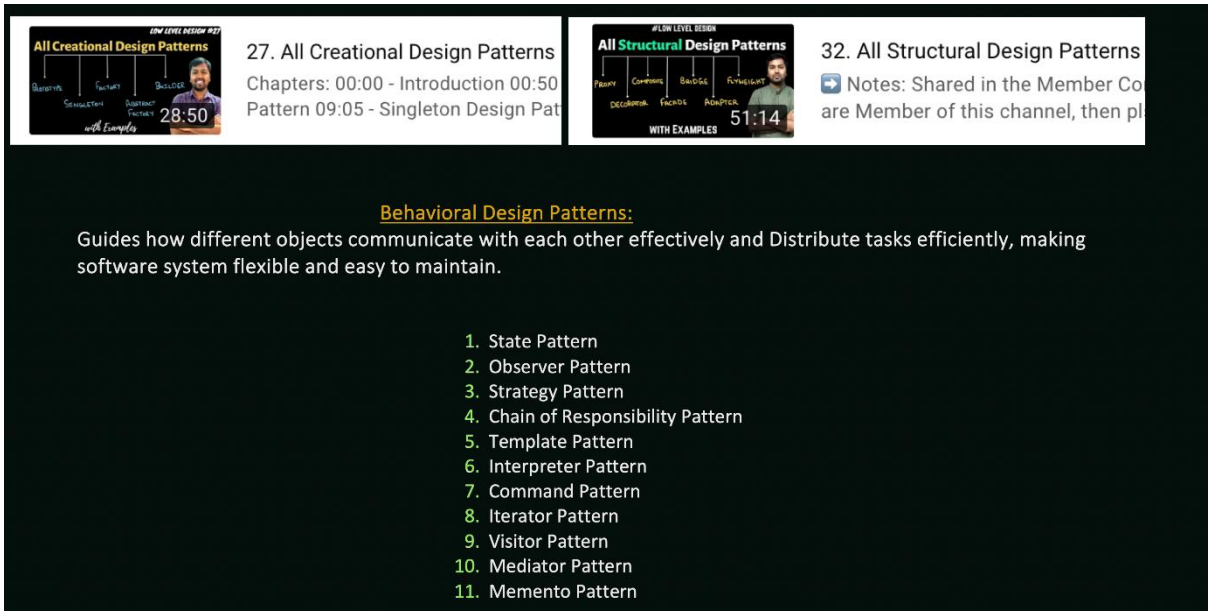
This Object Pool Design pattern is used with Singleton design pattern and required thread safety while acquiring and releasing the resource.

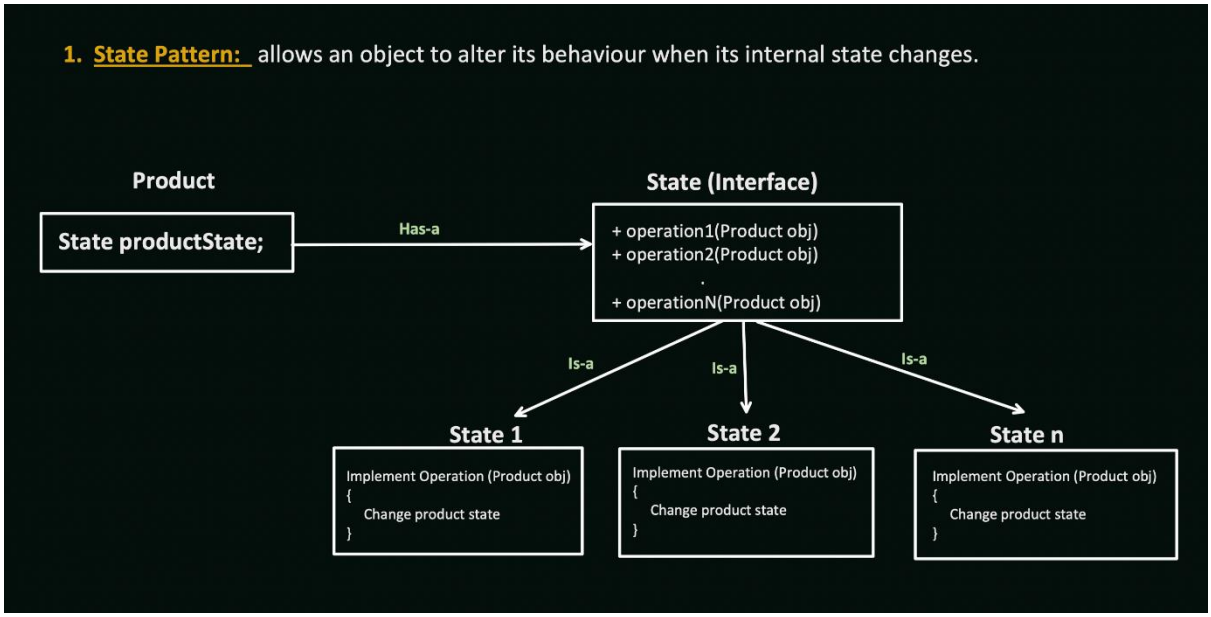
```java
public class Client {

    public static void main(String args[]){

        DBConnectionPoolManager poolManager =  DBConnectionPoolManager.getInstance();

        DBConnection dbConnection1 = poolManager.getDBConnection();
        DBConnection dbConnection2 = poolManager.getDBConnection();
        DBConnection dbConnection3 = poolManager.getDBConnection();
        DBConnection dbConnection4 = poolManager.getDBConnection();
        DBConnection dbConnection5 = poolManager.getDBConnection();
        DBConnection dbConnection6 = poolManager.getDBConnection();
        poolManager.getDBConnection();
        poolManager.releaseDBConnection(dbConnection6);

    }
}
```

```java
public class DBConnectionPoolManager {

    private List<DBConnection> freeConnectionsInPool = new ArrayList<>();
    private List<DBConnection> connectionsCurrentlyInUse = new ArrayList<>();
    private static final int INITIAL_POOL_SIZE = 3;
    private static final int MAX_POOL_SIZE = 6;
    private static DBConnectionPoolManager dbConnectionPoolManagerInstance = null;

    private DBConnectionPoolManager() {
        for (int i = 0; i < INITIAL_POOL_SIZE; i++) {
            freeConnectionsInPool.add(new DBConnection());
        }
    }

    public static DBConnectionPoolManager getInstance() {
        if(dbConnectionPoolManagerInstance == null) {
            synchronized (DBConnectionPoolManager.class) {
                if(dbConnectionPoolManagerInstance == null) {
                    dbConnectionPoolManagerInstance = new DBConnectionPoolManager();
                }
            }
        }
        return dbConnectionPoolManagerInstance;
    }

    public synchronized DBConnection getDBConnection() {
        if (freeConnectionsInPool.isEmpty() && connectionsCurrentlyInUse.size() < MAX_POOL_SIZE) {
            freeConnectionsInPool.add(new DBConnection());
        } else if (freeConnectionsInPool.isEmpty() && connectionsCurrentlyInUse.size() >= MAX_POOL_SIZE) {
            return null;
        }
        DBConnection dbConnection = freeConnectionsInPool.remove( index: freeConnectionsInPool.size() - 1);
        connectionsCurrentlyInUse.add(dbConnection);
        return dbConnection;
    }

    public synchronized void releaseDBConnection(DBConnection dbConnection) {
        if (dbConnection != null) {
            connectionsCurrentlyInUse.remove(dbConnection);
            freeConnectionsInPool.add(dbConnection);
        }
    }
}
```

```java
public class DBConnection {

    Connection mysqlConnection;

    DBConnection() {
        try {
            mysqlConnection = DriverManager.getConnection( url: "url",  user: "username",  password: "password");
        } catch (Exception e) {
            //handle exception here
        }
    }
}
```

All Behavioral Patterns (Concept and Coding)





```java
public class VendingMachine {

    VendingState machineState;

    public VendingState getMachineState() {
        return machineState;
    }

    public void setMachineState(VendingState machineState) {
        this.machineState = machineState;
    }
}
```

Has-a

```java
public interface VendingState {

    void insertCoin(VendingMachine product);

    void dispenseItem(VendingMachine product);
}
```

is-a

```java
public class IdleState implements VendingState{

    @Override
    public void insertCoin(VendingMachine product) {
        //insert coin logic
        System.out.println("Coin Inserted");
        product.setMachineState(new WorkingState());
    }

    @Override
    public void dispenseItem(VendingMachine product) {
        //not doing anything here
    }
}
```

is-a

```java
public class WorkingState implements VendingState{

    @Override
    public void insertCoin(VendingMachine product) {
        //not doing anything here
    }

    @Override
    public void dispenseItem(VendingMachine product) {
        System.out.println("Product dispensed");
        //set any other state if applicable
    }
}
```

2. **Observer Pattern:** in this an object (Observable) maintains a list of its dependents (observers) and notifies them of any changes in its state.

Observable <<Interface>>

```
List<Observer> observerList;

+ addObserver(Observer obj);
+ removeObserver(Observer obj);
+ notify();
+ setData();
```

Has-a

Observer<<Interface>>

```
+ update()
```

is-a

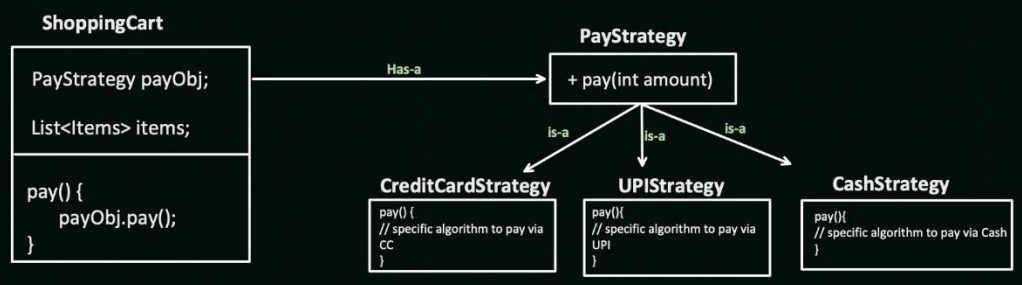
Observable Concrete Class

```
addObserver(Observer obj) {
        observerList.add(obj);
}

removeObserver(Observer obj) {
        observerList.remove(obj);
}

notify() {
   for(Observer obj: observerList) {
                obj.update();
        }
}

setData() {
        notify();
}
```

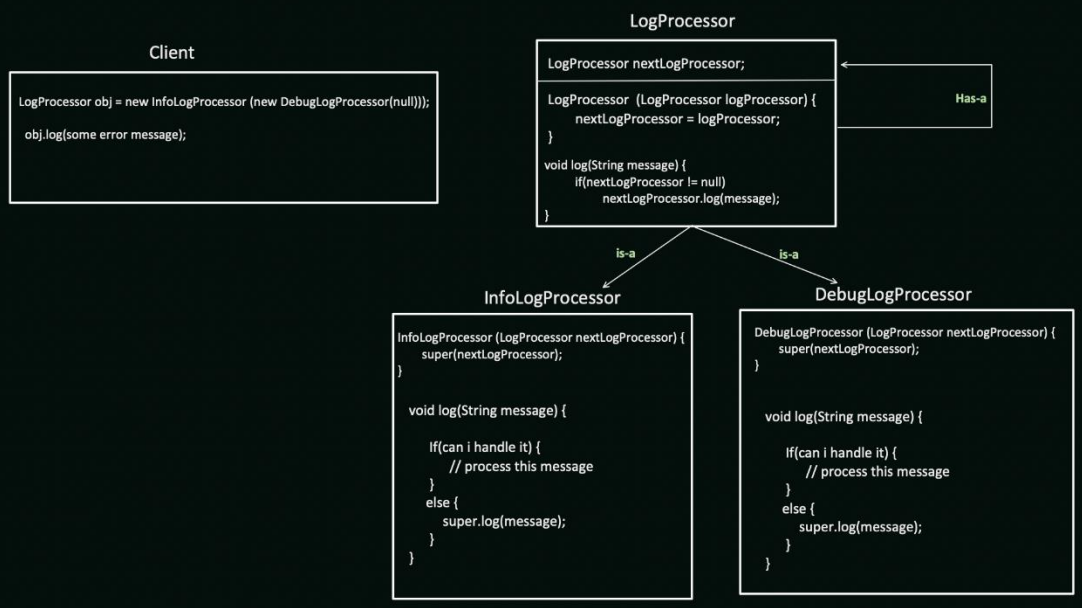
Observer Concrete Class

```
update() {
//do something
}
```

3. **Strategy Pattern:** helps to define multiple algorithm for the task and we can select any algorithm depending on the situation.



4. **Chain of Responsibility Pattern:** allows multiple objects to handle a request without the sender needing to know which object will ultimately process it.



5. Template Method Pattern: when you want all classes to follow specific steps to process the tasks but provide flexibility that each class can have their own logic in that specific step.

Client

```
PaymentFlow obj = new PayToFriend();
obj .sendMoney();
```

```java
public abstract class PaymentFlow {

    public abstract void validateRequest();
    public abstract void calculateFees();
    public abstract void debitAmount();
    public abstract void creditAmount();

    //this is Template method: which defines the order of steps to execute the task.
    public final void sendMoney(){
        //step1
        validateRequest();

        //step2
        debitAmount();

        //step3
        calculateFees();

        //step4
        creditAmount();
    }
}
```

is-a

```java
public class PayToFriend extends PaymentFlow{

    @Override
    public void validateRequest() {
        //specific validation for PayToFriend flow
        System.out.println("Validate logic of PayToFriend");
    }

    @Override
    public void debitAmount() {
        //debit the amount
        System.out.println("Debit the Amount logic of PayToFriend");
    }

    @Override
    public void calculateFees() {
        //specific Fee computation logic for PayToFriend flow
        System.out.println("0% fees charged");
    }

    @Override
    public void creditAmount() {
        // credit the amount logic
        System.out.println("Credit the full amount");
    }
}
```

is-a

```java
public class PayToMerchantFlow extends PaymentFlow{

    @Override
    public void validateRequest() {
        //specific validation for PayToFriend flow
        System.out.println("Validate logic of PayToMerchantFlow");
    }

    @Override
    public void debitAmount() {
        //debit the amount
        System.out.println("Debit the Amount logic of PayToMerchantFlow");
    }

    @Override
    public void calculateFees() {
        //specific Fee computation logic for PayToFriend flow
        System.out.println("2% fees charged");
    }

    @Override
    public void creditAmount() {
        // credit the amount logic
        System.out.println("Credit the remaining amount");
    }
}
```

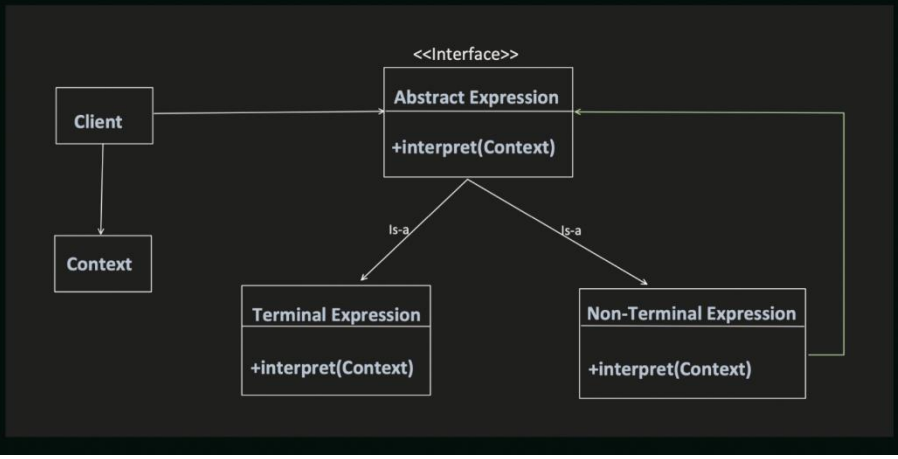
6. **Interpreter Pattern:** Defines a grammar for interpreting and evaluating an expression.

Some can interpret this:

- STOP
- Hi
- Number 5

Etc..

based on CONTEXT interpret happens.




```java
public class Client {

    public static void main(String args[]) {

        //initialize the context
        Context context = new Context();
        context.put( strVariable: "x",  intValue: 15);
        context.put( strVariable: "y",  intValue: 42);

        //use
        AbstractExpression expression1 = new MultiplyNonTerminalExpression(
                new NumberTerminalExpression( stringVal: "x"), new NumberTerminalExpression( stringVal: "y"));

        System.out.println(expression1.interpret(context));
    }
}
```

```java
public interface AbstractExpression {

    int interpret(Context context);
}
```

```java
public class Context {

    Map<String, Integer> contextMap = new HashMap<>();

    public void put(String strVariable, int intValue) {
        contextMap.put(strVariable, intValue);
    }

    public int get(String strVariable) {
        return contextMap.get(strVariable);
    }
}
```

```java
public class NumberTerminalExpression implements AbstractExpression{

    String stringValue;

    NumberTerminalExpression(String stringVal) {
        this.stringValue = stringVal;
    }

    @Override
    public int interpret(Context context) {
        return context.get(stringValue);
    }
}
```

```java
public class MultiplyNonTerminalExpression implements AbstractExpression{

    AbstractExpression leftExpression;
    AbstractExpression rightExpression;

    public MultiplyNonTerminalExpression(AbstractExpression leftExpression, AbstractExpression rightExpression) {
        this.leftExpression = leftExpression;
        this.rightExpression = rightExpression;
    }

    @Override
    public int interpret(Context context) {
        return leftExpression.interpret(context) * rightExpression.interpret(context);
    }
}
```

7. **Command Pattern:** Turns requests (commands) into objects, allowing you to either parametrized or queue them. This will help to decouple the request Sender and receiver.

Problem with below code:

process of turning on AC is simple, but if there are more steps, client has to aware all of that, which is not good. So Sender and Receiver are not decoupled.

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

        AirConditioner ac = new AirConditioner();
        ac.turnOnAC();
        ac.setTemperature(24);
        ac.turnOffAC();
    }
}
```

```java
public class AirConditioner {

    boolean isOn;
    int temperature;

    public void turnOnAC(){
        isOn = true;
        System.out.println("AC is ON");
    }

    public void turnOffAC(){
        isOn = false;
        System.out.println("AC is OFF");
    }

    public void setTemperature(int temp){
        this.temperature = temp;
        System.out.println("Temperature changed to:" + temperature) ;
    }
}
```

Solution with Command Pattern:

Invoker

```java
public class MyRemoteControl {

    ICommand command;
    MyRemoteControl(){
    }

    public void setCommand(ICommand command){
        this.command = command;
    }

    public void pressButton() {
        command.execute();
    }
}
```

Command

```java
public interface ICommand {

    public void execute();
}
```

```java
public class TurnACOnCommand implements ICommand{

    AirConditioner ac;
    TurnACOnCommand(AirConditioner ac){
        this.ac = ac;
    }

    @Override
    public void execute() {
        ac.turnOnAC();
    }
}
```

```java
public class TurnACOffCommand implements ICommand{

    AirConditioner ac;
    TurnACOffCommand(AirConditioner ac){
        this.ac = ac;
    }

    @Override
    public void execute() {
        ac.turnOffAC();
    }
}
```

Receiver

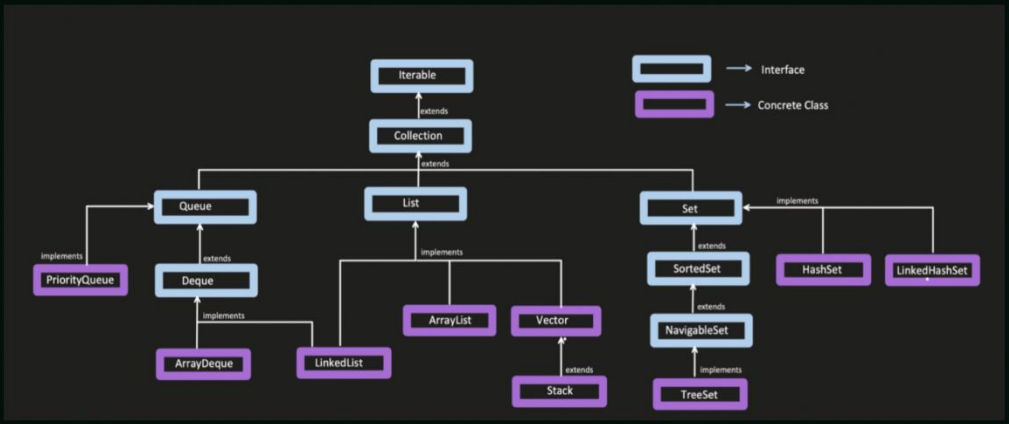
```java
public class AirConditioner {

    boolean isOn;
    int temperature;

    public void turnOnAC(){
        isOn = true;
        System.out.println("AC is ON");
    }

    public void turnOffAC(){
        isOn = false;
        System.out.println("AC is OFF");
    }

    public void setTemperature(int temp){
        this.temperature = temp;
        System.out.println("Temperature changed to:" + temperature) ;
    }
}
```

Client

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

        //AC object
        AirConditioner airConditioner = new AirConditioner();

        //remote
        MyRemoteControl remoteObj = new MyRemoteControl();

        //create the command and press the button
        remoteObj.setCommand(new TurnACOnCommand(airConditioner));
        remoteObj.pressButton();
    }
}
```

8. **Iterator Pattern:** that provides a way to access element of a Collection sequentially without exposing the underlying representation of the collection.

Understand the Need for an ITERATOR Pattern:




```java
public class LinkedHashSetExample {

    public static void main(String args[]){

        Set<Integer> intSet = new LinkedHashSet<>();
        intSet.add(2);
        intSet.add(77);
        intSet.add(82);
        intSet.add(63);
        intSet.add(5);

       Iterator<Integer> iterable =  intSet.iterator();
       while(iterable.hasNext()){
           int val = iterable.next();
           System.out.println(val);
       }
    }
}
```

Iterator UML with an Example:

```java
public interface Iterator {
    boolean hasNext();
    Object next();
}
```

is-a

```java
public class Library {

    private List<Book> booksList;

    public Library(List<Book> booksList) {
        this.booksList = booksList;
    }

    public Iterator createIterator() {
        return new BookIterator(booksList);
    }
}
```

has-a

```java
public class BookIterator implements Iterator {
    private List<Book> books;
    private int index = 0;

    public BookIterator(List<Book> books) {
        this.books = books;
    }

    @Override
    public boolean hasNext() {
        return index < books.size();
    }

    @Override
    public Object next() {
        if (this.hasNext()) {
            return books.get(index++);
        }
        return null;
    }
}
```

```java
public class Client {

        public static void main(String[] args) {
            List<Book> booksList = Arrays.asList(
                    new Book( price: 100,  bookName: "Science"),
                    new Book( price: 200,  bookName: "Maths"),
                    new Book( price: 300,  bookName: "GK"),
                    new Book( price: 400,  bookName: "Drawing")
            );

            Library lib = new Library(booksList);
            Iterator iterator = lib.createIterator();

            while (iterator.hasNext()) {
                Book book = (Book) iterator.next();
                System.out.println(book.getBookName());
            }
        }
}
```

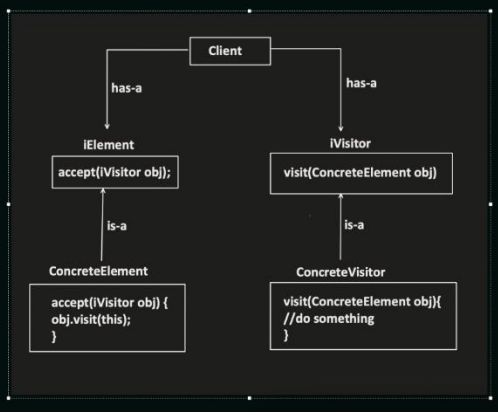
```java
public class Book {

    private int price;
    private String bookName;

    Book(int price, String bookName){
        this.price = price;
        this.bookName = bookName;
    }
    public int getPrice() {
        return price;
    }

    public String getBookName() {
        return bookName;
    }
}
```

9. **Visitor Pattern:** Allows adding new operations to existing classes without modifying them and encourage OPEN/CLOSED principle.

What's the problem with the below class?

```java
public class HotelRoom {

    public void getRoomPrice(){
        //price computation logic
    }

    public void initiateRoomMaintenance(){
        //start room maintenance
    }

    public void reserveRoom(){
        //perform operation to reserve the room
    }

    //many more operations can come over the time
}
```

UML of Visitor Pattern




```java
public class Client {

    public static void main(String args[]){

        RoomElement singleRoomObj = new SingleRoom();
        RoomElement doubleRoomObj = new DoubleRoom();
        RoomElement deluxeRoomObj = new DeluxeRoom();

        //performing an operation on the  objects
        RoomVisitor pricingVisitorObj = new RoomPricingVisitor();
        singleRoomObj.accept(pricingVisitorObj);
        System.out.println((SingleRoom)singleRoomObj).roomPrice);

        doubleRoomObj.accept(pricingVisitorObj);
        System.out.println((DoubleRoom)doubleRoomObj).roomPrice);

        deluxeRoomObj.accept(pricingVisitorObj);
        System.out.println((DeluxeRoom)deluxeRoomObj).roomPrice);

        //performing another operation on the  objects
        RoomVisitor maintenanceVisitorObj = new RoomMaintenanceVisitor();
        singleRoomObj.accept(maintenanceVisitorObj);

        doubleRoomObj.accept(maintenanceVisitorObj);

        deluxeRoomObj.accept(maintenanceVisitorObj);
    }
}
```

```java
public interface RoomElement {

    public void accept(RoomVisitor visitor);
}
```

is-a

```java
public class SingleRoom implements RoomElement{

    public int roomPrice = 0;

    @Override
    public void accept(RoomVisitor visitor) {
        visitor.visit( singleRoomObj, this);
    }
}
```

```java
public class DoubleRoom implements RoomElement{

    public int roomPrice = 0;

    @Override
    public void accept(RoomVisitor visitor) {
        visitor.visit( doubleRoomObj, this);
    }
}
```

```java
public class DeluxeRoom implements RoomElement{

    public int roomPrice = 0;

    @Override
    public void accept(RoomVisitor visitor) {
        visitor.visit( deluxeRoomObj, this);
    }
}
```

```java
public interface RoomVisitor {

    public void visit(SingleRoom singleRoomObj);
    public void visit(DoubleRoom doubleRoomObj);
    public void visit(DeluxeRoom deluxeRoomObj);
}
```

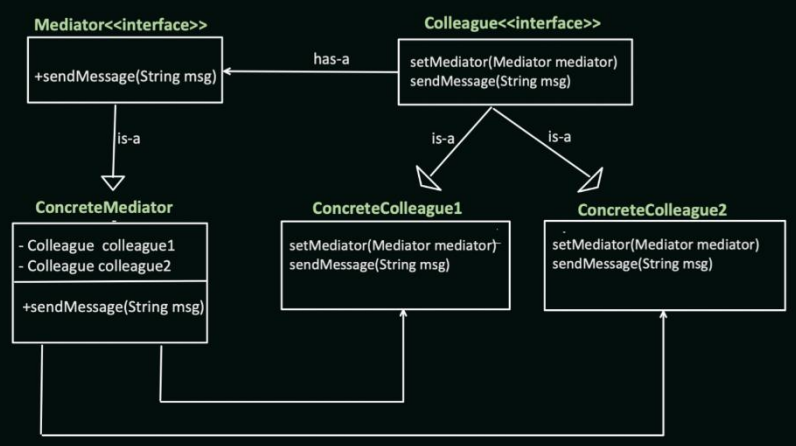
is-a

```java
public class RoomPricingVisitor implements RoomVisitor{

    @Override
    public void visit(SingleRoom singleRoomObj) {
        System.out.println("Pricing computation logic of SingleRoom");
        singleRoomObj.roomPrice = 1000;
    }

    @Override
    public void visit(DoubleRoom doubleRoomObj) {
        System.out.println("Pricing computation logic of DoubleRoom");
        doubleRoomObj.roomPrice = 2000;
    }

    @Override
    public void visit(DeluxeRoom deluxeRoomObj) {
        System.out.println("Pricing computation logic of DeluxeRoom");
        deluxeRoomObj.roomPrice = 5000;
    }
}
```

```java
public class RoomMaintenanceVisitor implements RoomVisitor{

    @Override
    public void visit(SingleRoom singleRoomObj) {
        System.out.println("Performing maintenance of SingleRoom");
    }

    @Override
    public void visit(DoubleRoom doubleRoomObj) {
        System.out.println("Performing maintenance of DoubleRoom");
    }

    @Override
    public void visit(DeluxeRoom deluxeRoomObj) {
        System.out.println("Performing maintenance of DeluxeRoom");
    }
}
```

10. **Mediator Pattern:** It encourage loose coupling by keeping objects from referring to each other explicitly and allows them to ***communicate through a mediator*** object.

UML



Lets use, Online Auction System Example to understand the UML

```java
//this is Mediator Interface
public interface AuctionMediator {
    void addBidder(Colleague bidder);
    void placeBid(Colleague bidder, int bidAmount);
}
```

has-a

```java
public interface Colleague {

    void placeBid(int bidAmount);
    void receiveBidNotification(int bidAmount);
    String getName();
}
```

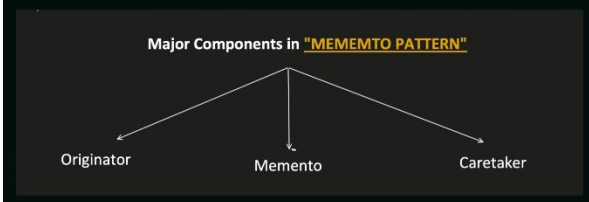
is-a

```java
//Mediator Concrete Class
public class Auction implements AuctionMediator {

    List<Colleague> colleagues = new ArrayList<>();

    @Override
    public void addBidder(Colleague bidder) {
        colleagues.add(bidder);
    }

    @Override
    public void placeBid(Colleague bidder, int bidAmount) {

        for(Colleague colleague : colleagues){
            if(!colleague.getName().equals(bidder.getName())){
                colleague.receiveBidNotification(bidAmount);
            }
        }
    }
}
```

is-a

```java
public class Bidder implements Colleague {

    String name;
    AuctionMediator auctionMediator;

    Bidder(String name, AuctionMediator auctionMediator) {
        this.name = name;
        this.auctionMediator = auctionMediator;
        auctionMediator.addBidder(this);
    }

    @Override
    public void placeBid(int bidAmount) {
        auctionMediator.placeBid( bidder: this, bidAmount);
    }

    @Override
    public void receiveBidNotification(int bidAmount) {
        System.out.println("Bidder: " + name + " got the notification that someone has put bid of : " + bidAmount);
    }

    @Override
    public String getName(){
        return name;
    }
}
```

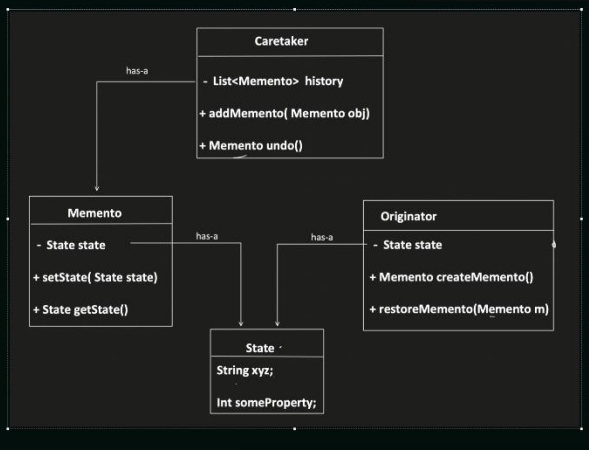
11. **Memento Pattern:** Provides an ability to revert an object to a previous state i.e. UNDO capability.
and it does not expose the object internal implementation.



Originator:
- It represents the object, for which state need to be saved and restored.
- Expose Methods to Save and Restore its state using Memento object.

Memento:
- It represents an Object which holds the state of the Originator.

Caretaker:
- Manages the list of States (i.e. list of Memento)



```java
//Caretaker
public class ConfigurationCareTaker {

    List<ConfigurationMemento> history = new ArrayList<>();

    public void addMemento(ConfigurationMemento memento) {
        history.add(memento);
    }

    public ConfigurationMemento undo() {
        if (!history.isEmpty()) {
            int lastMementoIndex = history.size() - 1;
            //get the last memento from the list
            ConfigurationMemento lastMemento = history.get(lastMementoIndex);
            //remove the last memento from the list now
            history.remove(lastMementoIndex);
            return lastMemento;
        }
        return null;
    }
}
```

```java
//Memento
public class ConfigurationMemento {

    int height;
    int width;

    public ConfigurationMemento(int height, int width){
        this.height = height;
        this.width = width;
    }

    public int getHeight() {
        return height;
    }

    public int getWidth() {
        return width;
    }
}
```

```java
//Originator
public class ConfigurationOriginator {

    int height;
    int width;

     ConfigurationOriginator(int height, int width){
        this.height = height;
        this.width = width;
    }

    public void setHeight(int height) {
        this.height = height;
    }

    public void setWidth(int width) {
        this.width = width;
    }

    public ConfigurationMemento createMemento(){
        return new ConfigurationMemento(this.height, this.width);
    }

    public void restoreMemento(ConfigurationMemento mementoToBeRestored){
        this.height = mementoToBeRestored.height;
        this.width = mementoToBeRestored.width;
    }
```

LLD: Command Design Pattern

LLD: Command Design Pattern

Thursday, 3 August 2023 3:58 PM

Pattern Category: It's a behavioral pattern.

Lets take the use-case of Remote control which can control various home appliances and with that lets understand the problem, then we will go with this pattern.

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

        AirConditioner ac = new AirConditioner();
        ac.turnOnAC();
        ac.setTemperature(24);
        ac.turnOffAC();
    }
}
```

```java
public class AirConditioner {

    boolean isOn;
    int temperature;

    public void turnOnAC(){
        isOn = true;
        System.out.println("AC is ON");
    }

    public void turnOffAC(){
        isOn = false;
        System.out.println("AC is OFF");
    }

    public void setTemperature(int temp){
        this.temperature = temp;
        System.out.println("Temperature changed to:" + temperature) ;
    }
}
```

Problem with above implementation:

- Lack of Abstraction :

Today, process of turning on AC is simple, but if there are more steps, client has to aware all of that, which is not good.

- Undo/Redo Functionality:

What if I want to implement the undo/redo capability. How it will be handled.

- Difficulty in Code Maintenance:

What if in future, we have to support more commands for more devices example Bulb. Ma

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

        AirConditioner ac = new AirConditioner();
        ac.turnOnAC();
        ac.setTemperature(24);
        ac.turnOffAC();

        Bulb bulbObj = new Bulb();
        bulbObj.turnOnBulb();
        bulbObj.turnOffBulb();
    }
}
```

```java
public class Bulb {

    boolean isOn;

    public void turnOnBulb(){
        isOn = true;
        System.out.println("Bulb is ON");
    }

    public void turnOffBulb(){
        isOn = false;
        System.out.println("Bulb is OFF");
    }
}
```

```java
public class AirConditioner {

    boolean isOn;
    int temperature;

    public void turnOnAC(){
        isOn = true;
        System.out.println("AC is ON");
    }

    public void turnOffAC(){
        isOn = false;
        System.out.println("AC is OFF");
    }

    public void setTemperature(int temp){
        this.temperature = temp;
        System.out.println("Temperature changed to:" + temperature) ;
    }
}
```

How COMMAND DESIGN PATTERN Solves it?
It separates the logic of:

- Receiver
- Invoker and
- Command

Invoker

```java
public class MyRemoteControl {

    ICommand command;
    MyRemoteControl(){
    }

    public void setCommand(ICommand command){
        this.command = command;
    }

    public void pressButton() {
        command.execute();
    }
}
```

Command

```java
public interface ICommand {

    public void execute();
}
```

```java
public class TurnACOnCommand implements ICommand{

    AirConditioner ac;
    TurnACOnCommand(AirConditioner ac){
        this.ac = ac;
    }

    @Override
    public void execute() {
        ac.turnOnAC();
    }
}
```

```java
public class TurnACOffCommand implements ICommand{

    AirConditioner ac;
    TurnACOffCommand(AirConditioner ac){
        this.ac = ac;
    }

    @Override
    public void execute() {
        ac.turnOffAC();
    }
}
```

Receiver

```java
public class AirConditioner {

    boolean isOn;
    int temperature;

    public void turnOnAC(){
        isOn = true;
        System.out.println("AC is ON");
    }

    public void turnOffAC(){
        isOn = false;
        System.out.println("AC is OFF");
    }

    public void setTemperature(int temp){
        this.temperature = temp;
        System.out.println("Temperature changed to:" + temperature) ;
    }
}
```

Client

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

        //AC object
        AirConditioner airConditioner = new AirConditioner();

        //remote
        MyRemoteControl remoteObj = new MyRemoteControl();

        //create the command and press the button
        remoteObj.setCommand(new TurnACOnCommand(airConditioner));
        remoteObj.pressButton();
    }
}
```

```java
import java.util.Stack;

public class MyRemoteControl {

    Stack<ICommand> acCommandHistory = new Stack<>();
    ICommand command;

    MyRemoteControl(){
    }

    public void setCommand(ICommand command){
        this.command = command;
    }

    public void pressButton()
    {
        command.execute();
        acCommandHistory.add(command);
    }

    public void undo(){

        if(!acCommandHistory.isEmpty()){
            ICommand lastCommand = acCommandHistory.pop();
            lastCommand.undo();
        }
    }
}
```

```java
public interface ICommand {

    public void execute();
    public void undo();

}
```

```java
public class AirConditioner {

    boolean isOn;
    int temperature;

    public void turnOnAC(){
        isOn = true;
        System.out.println("AC is ON");
    }

    public void turnOffAC(){
        isOn = false;
        System.out.println("AC is OFF");
    }

    public void setTemperature(int temp){
        this.temperature = temp;
        System.out.println("Temperature changed to:" + temperature) ;
    }
}
```

```java
public class TurnACOnCommand implements ICommand{

    AirConditioner ac;
    TurnACOnCommand(AirConditioner ac){
        this.ac = ac;
    }

    @Override
    public void execute() {
        ac.turnOnAC();
    }

    @Override
    public void undo() {
        ac.turnOffAC();
    }
}
```

```java
public class TurnACOffCommand implements ICommand{

    AirConditioner ac;
    TurnACOffCommand(AirConditioner ac){
        this.ac = ac;
    }

    @Override
    public void execute() {
        ac.turnOffAC();
    }

    @Override
    public void undo() {
        ac.turnOnAC();
    }
}
```

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

        //AC object
        AirConditioner airConditioner = new AirConditioner();

        //remote
        MyRemoteControl remoteObj = new MyRemoteControl();

        //create the command and press the button
        remoteObj.setCommand(new TurnACOnCommand(airConditioner));
        remoteObj.pressButton();

        //undo the last operation
        remoteObj.undo();
    }
}
```

LLD: All Structural Design Patterns

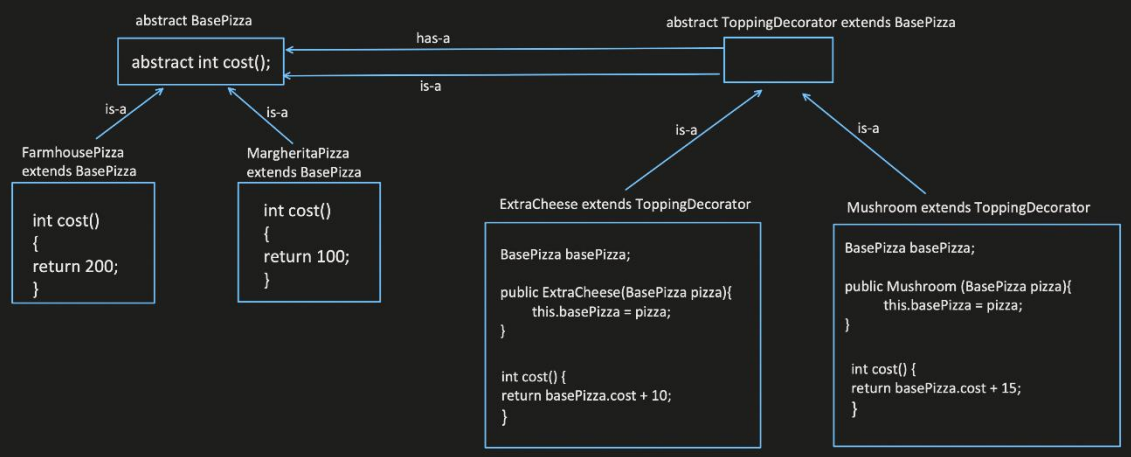
Structural Design Pattern is a way to combine or arrange different classes and objects to form a complex or bigger structure to solve a particular requirement.

Types:
---

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

1. Decorator Pattern:

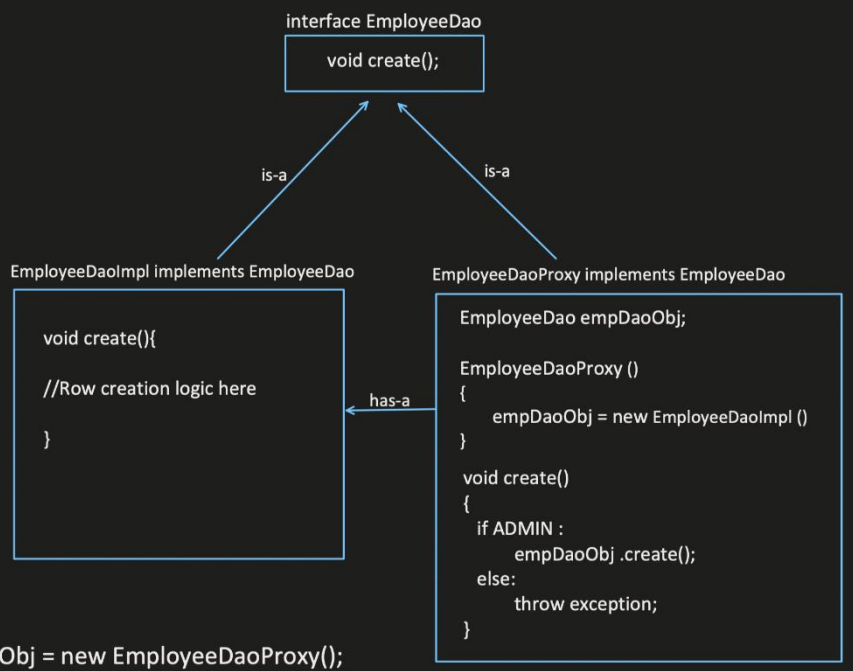
This pattern helps to add more functionality to existing object, without changing its structure.




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

2. Proxy Pattern:

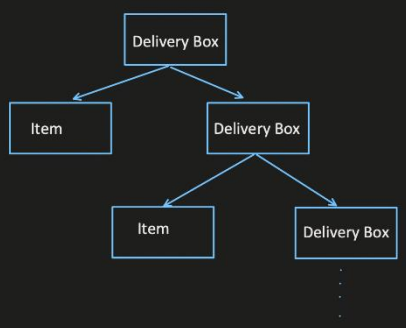
This pattern helps to provide control access to original object.

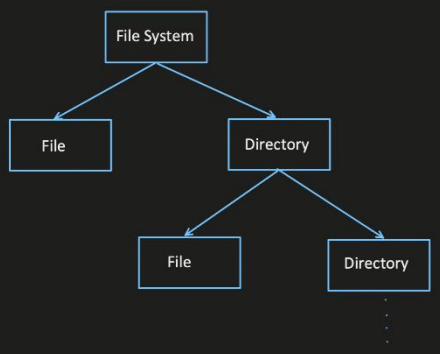


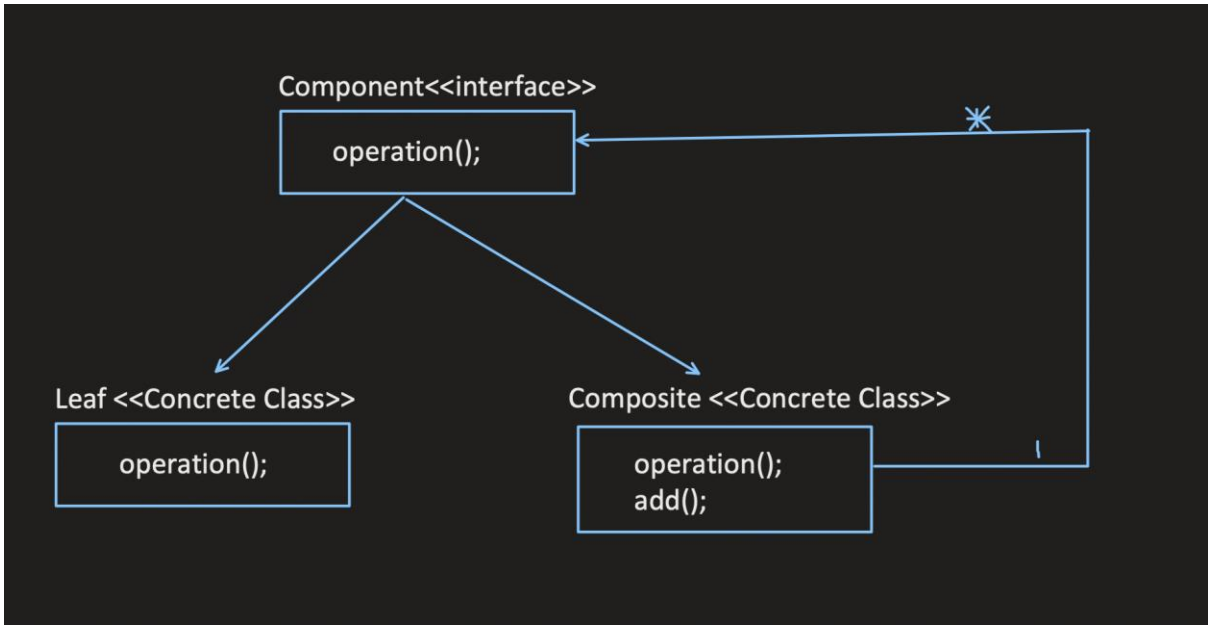

```
EmployeeDao empProxyObj = new EmployeeDaoProxy();
empProxyObj .create();
```

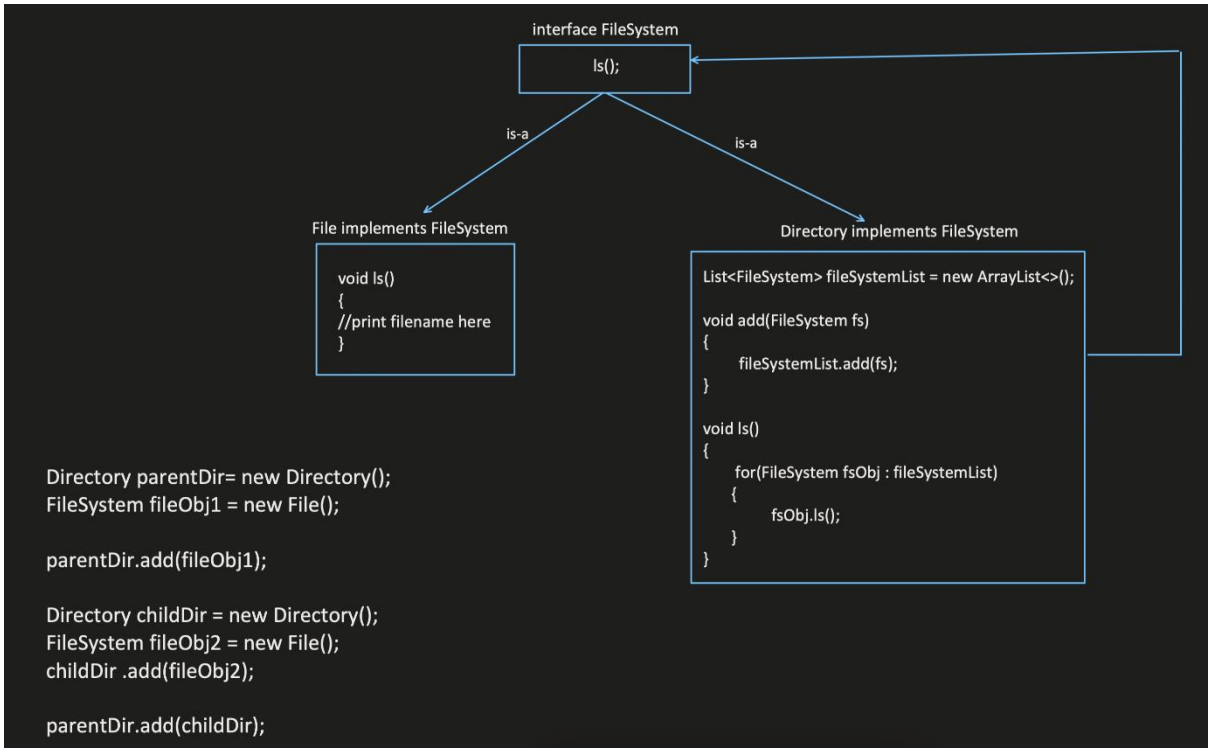
3. Composite Pattern:

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



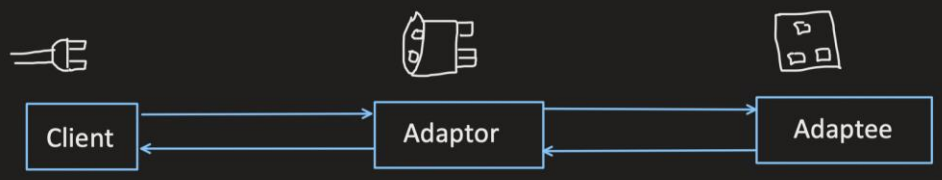



Component<<interface>>
operation();
*
Leaf <<Concrete Class>>
operation();
Composite <<Concrete Class>>
operation();
add();
1


interface FileSystem
ls();
is-a
is-a
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);

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();
```

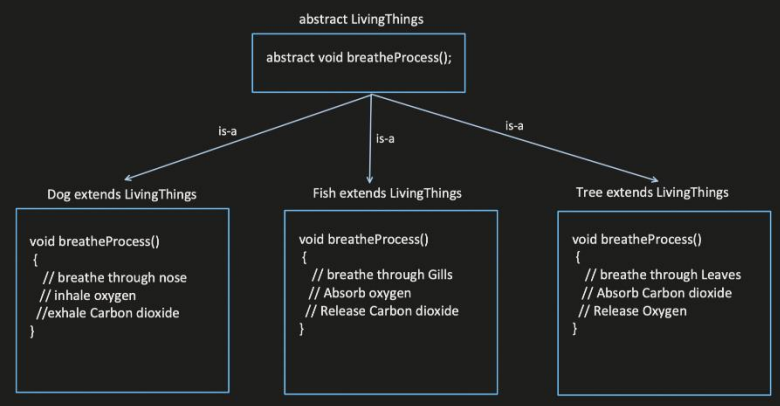
is-a

WeightMachineImpl implements WeightMachine

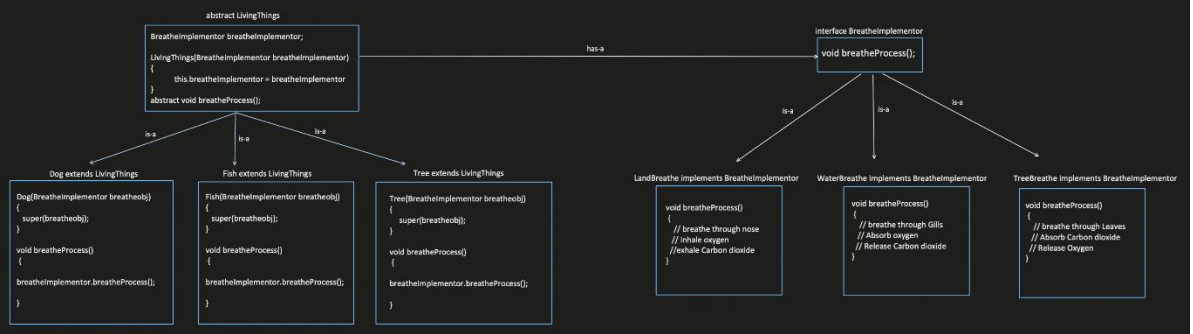
```
int getWeightInPounds()
{
      return 30;
}
```

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?

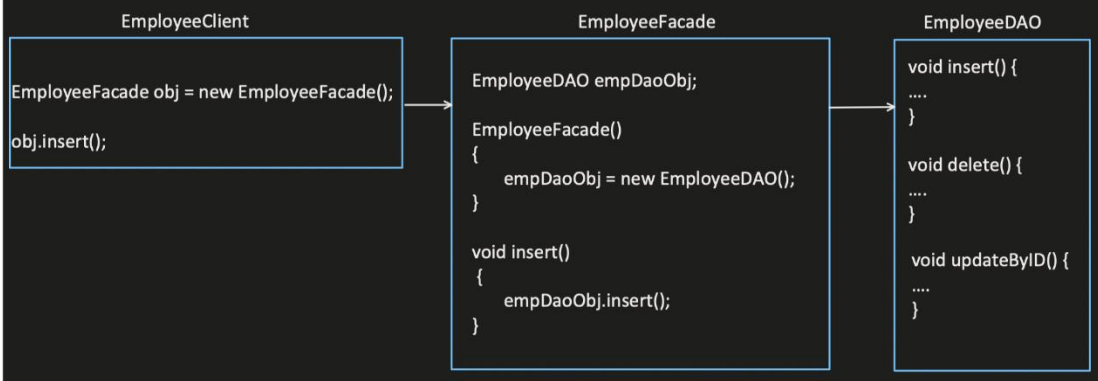



```
LivingThings fishObj= new Fish(new WaterBreathe());
fishObj.breatheProcess();
```

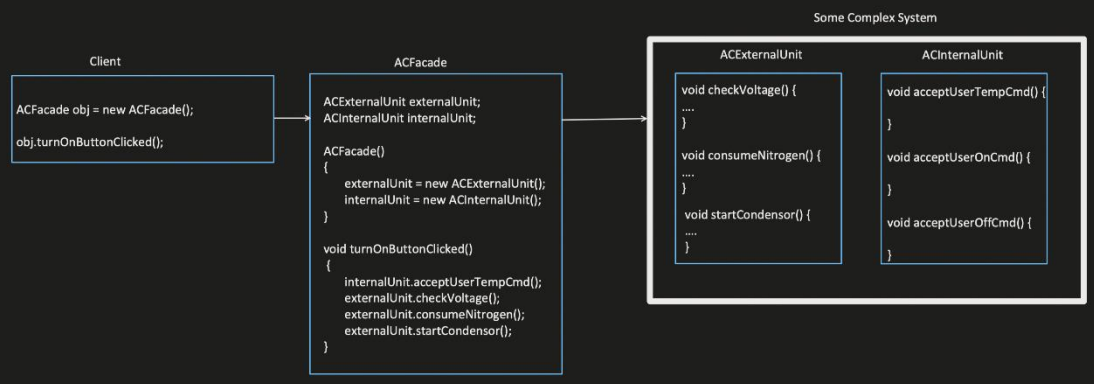
6. Facade Pattern:

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



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);

}
```

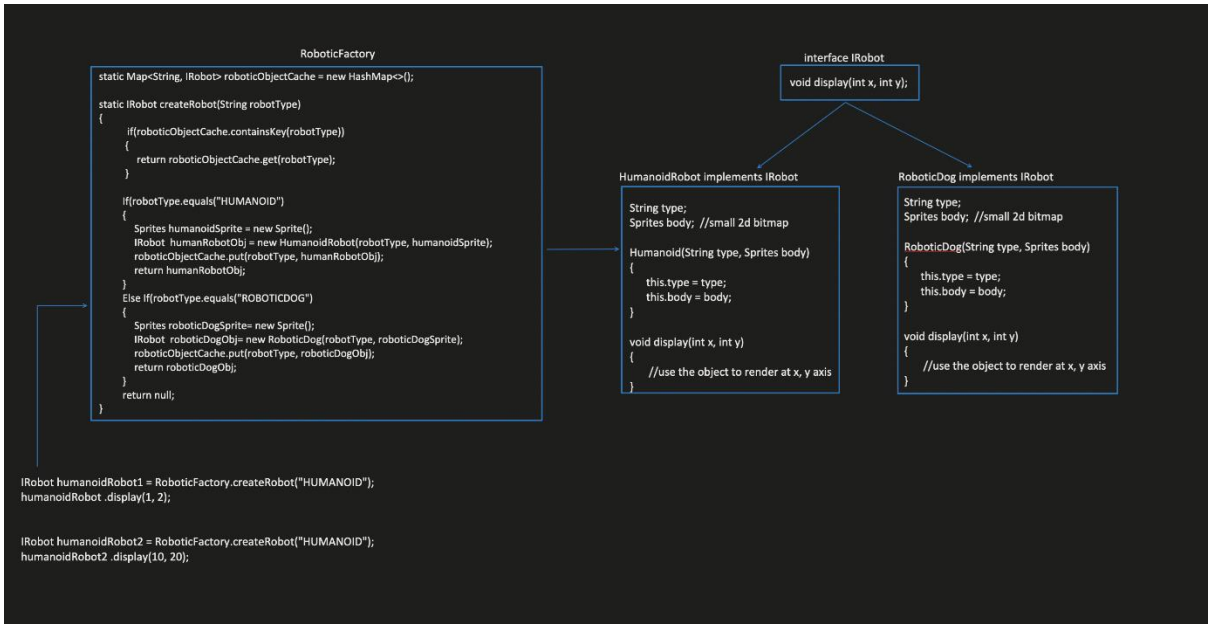
= 10Lakh * ~31KB = 31GB

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);