**LAB 7**

**STRUCTURAL DESIGN PATTERNS**

**IN SMART**

**WHEEL CHAIR SYSTEM**

**Prepared by :**

**Akeel Ahmed - 190028C**

**Thilina Ilesinghe - 190238U**

**Hafeel - 190211G**

Table of Contents

[**Composite Pattern** 3](#_Toc87125095)

[**Description** 3](#_Toc87125096)

[**Problem and Solution** 3](#_Toc87125097)

[**Class Diagram** 4](#_Toc87125098)

[**Sequence Diagram** 4](#_Toc87125099)

[**Code Snippets** 4](#_Toc87125100)

[**Decorator Pattern** 6](#_Toc87125101)

[**Description** 6](#_Toc87125102)

[**Problem and Solution** 6](#_Toc87125103)

[**Class Diagram** 7](#_Toc87125104)

[**Sequence Diagram** 7](#_Toc87125105)

[**Code Snippets** 7](#_Toc87125106)

# **Composite Pattern**

## **Description**

In the smart wheelchair system we observe the CareHouse and CareTakers to be important components of the system

A carehouse consists of number of caretakers and we observe that the a need to merge carehouses to a single one could occur

In this situation a need to treat individual objects( CareTaker objects) and compositions of Objects( CareHouse Objects) to be treated uniformly.

So, we use the composite pattern for this purpose.

## **Problem and Solution**

In the above scenario we observe that the need to get details about a carehouse needs to

get details about each caretaker object as well. If we implement this need without using a composite pattern

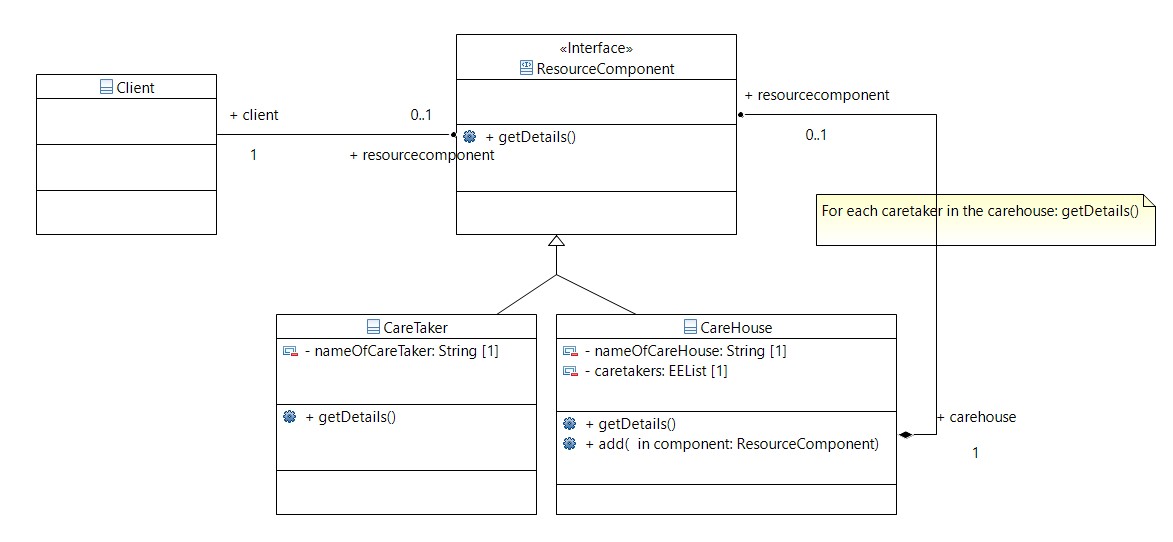
we would need to explicitly check whether an object is instance of which class and need to

invoke the getDetails() method of the respective class .

But, by using this design pattern , we need not do so.

Similarly we could use each carehouse as leaf nodes and hospitals as composite objects.

## **Class Diagram**



## **Sequence Diagram**

## **Code Snippets**

import java.util.ArrayList;

import java.util.List;

interface ResourceComponent {

    void getDetails();

}

class CareTaker implements ResourceComponent { // a leaf node

    private String nameOfCareTaker;

    CareTaker(String nameOfCareTaker) {

        this.nameOfCareTaker = nameOfCareTaker;

    }

    @Override

    public void getDetails() {

        System.out.println("CareTaker name: " + nameOfCareTaker);

    }

}

class CareHouse implements ResourceComponent { // Composite Collection

    private String nameOfCareHouse;

    CareHouse(String nameOfCareHouse) {

        this.nameOfCareHouse = nameOfCareHouse;

    }

    private List<ResourceComponent> caretakers = new ArrayList<ResourceComponent>();

    @Override

    public void getDetails() {

        System.out.println(nameOfCareHouse);

        for (ResourceComponent caretaker : caretakers) {

            caretaker.getDetails();

        }

    }

    public void add(ResourceComponent component) {

        caretakers.add(component);

    }

}

public class composite {

    public static void main(String[] args) {

        CareHouse ch1 = new CareHouse("CareHouse 01");

        ch1.add(new CareTaker("CareTaker 1"));

        ch1.add(new CareTaker("CareTaker 2"));

        CareHouse ch2 = new CareHouse("CareHouse 02");

        ch2.add(new CareTaker("CareTaker 3"));

        ch2.add(new CareTaker("CareTaker 4"));

        //Eg:  merging 2 carehouses into a single carehouse with a new name

        CareHouse merged = new CareHouse("CareHouse 03");

        merged.add(ch1);

        merged.add(ch2);

        merged.getDetails();

    }

}

# **Decorator Pattern**

## **Description**

In the smart wheel chair system , we have to write details to the database. Some details needs decoration or modification before writing it to the database.

The passwords need to be encrypted and if files of medical reports are stored in database the compress feature would also be needed.

We use the decorator pattern to efficiently decorate data before the major operation of writing it to the database.

## **Problem and Solution**

The problem associated with the basic way of defining write methods to database within each different type of modified writing process is not maintainable. To extend the code to add a combination of modifications like encrypted data being compressed would result in creating new classes making it less maintainable.

Thus, the decorator pattern is used so that a combination of modifications could done easily as portrayed in the code snippets using a decorator abstract class which could be inherited by any class that wants to add a new modification feature.

## **Class Diagram**

Diagram

Description automatically generated

## **Sequence Diagram**

## **Code Snippets**

import java.util.HashMap;

import java.util.Map;

class Database {

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

    // singleton creational design pattern

    private static Database instance = new Database();

    private static int id = 0;

    private Database() {

    }

    public static Database getDatabaseInstance() {

        return instance;

    }

    public void addToDatabase(String str) {

        id++;

        System.out.println("Added [ key = "+id+" : value = "+str+"] to the database");

        db.put(id, str);

    }

}

interface Stream {

    void action(String str);

}

class CloudStream implements Stream {

    public void writeToDB(String str) {

        Database database = Database.getDatabaseInstance();

        database.addToDatabase(str);

    }

    public void action(String str){

        writeToDB(str);

    };

}

abstract class StreamDecorator implements Stream{

    private final Stream streamToBeDecorated;

    StreamDecorator(Stream streamToBeDecorated){

        this.streamToBeDecorated =streamToBeDecorated;

    }

    @Override

    public void action(String str) {

        streamToBeDecorated.action(str);

    }

}

class EncryptedStream extends StreamDecorator {

    EncryptedStream(Stream streamToBeDecorated) {

        super(streamToBeDecorated);

    }

    public void action(String str) {

        str = "Encrypted(" + str + ")";

        super.action(str);

    }

}

class CompressedStream extends StreamDecorator {

    CompressedStream(Stream streamToBeDecorated) {

        super(streamToBeDecorated);

    }

    public void action(String str) {

        str = "Compressed(" + str + ")";

        super.action(str);

    }

}

public class decorator {

    public static void main(String[] args) {

        Stream stream = new CloudStream();

        stream.action("Name");

        Stream encryptedStream = new EncryptedStream(stream);

        encryptedStream.action("password");

        Stream compressedStream = new CompressedStream(stream);

        compressedStream.action("abcdefghijklmnopqrstuvwxyz");

        Stream ce\_Stream = new CompressedStream(new EncryptedStream(stream));

        ce\_Stream.action("...COMPRESS&thenENCRYPT");

        Stream ec\_Stream = new EncryptedStream(new CompressedStream(stream));

        ec\_Stream.action("...ENCRYPT&thenCOMPRESS");

    }

}