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

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

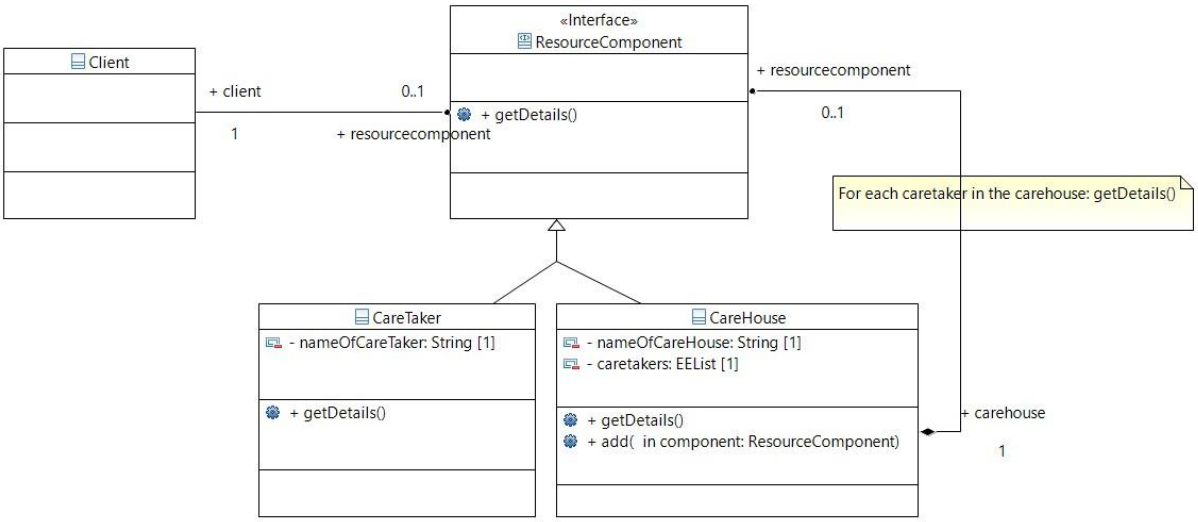
we would need to explicitly check whether an object is instance of which class and need to

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



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);

    }

}
```

```java
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.

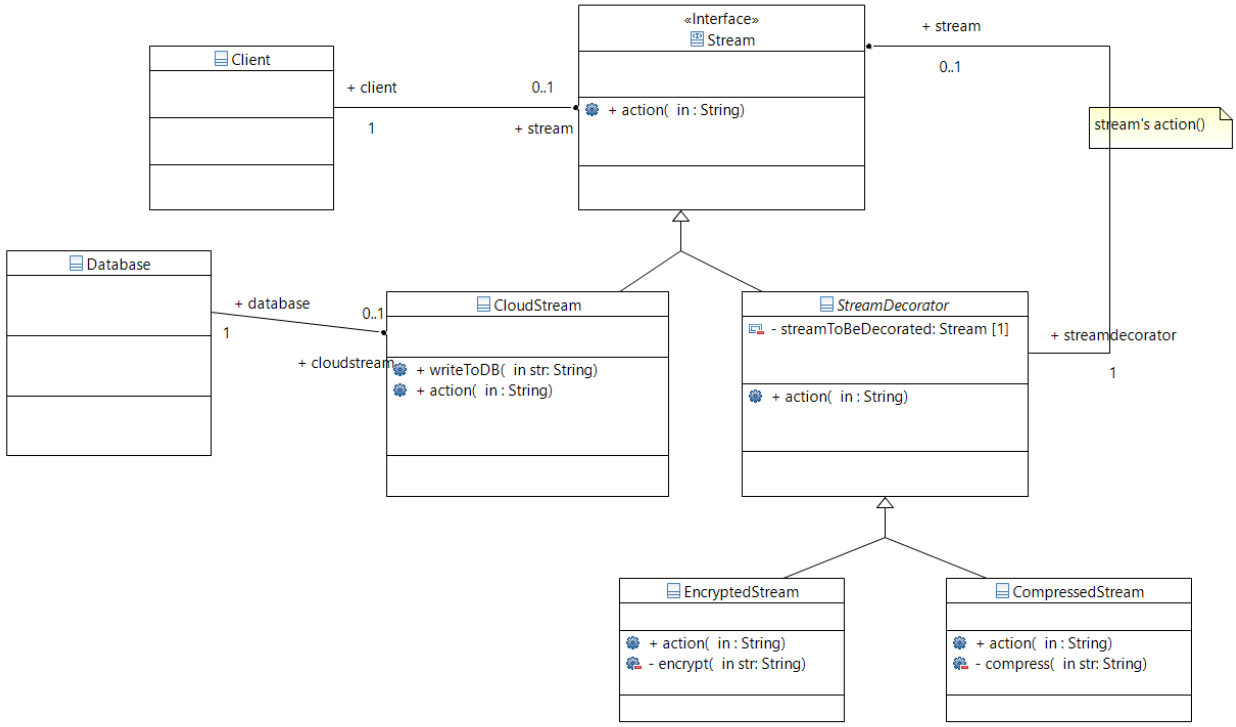
We use the decorator pattern to efficiently decorate data before the major operation of writing it to the database.

Problem and Solution

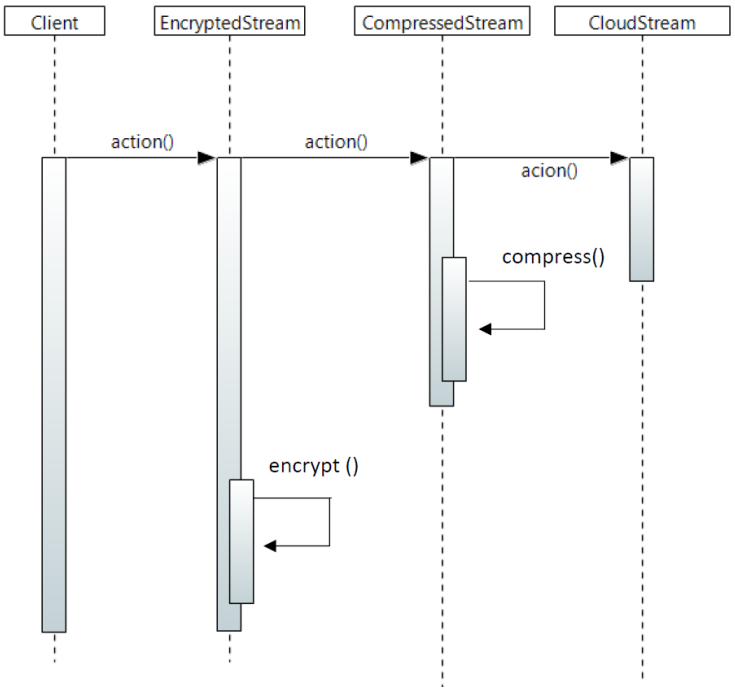
The problem associated with the basic way of defining write methods to database within each different type of modified writing process is not maintainable. To extend the code to add a combination of modifications like encrypted data being compressed would result in creating new classes making it less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



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");


    }
}
```

Façade Pattern

Problem

The most important part of the smart wheelchair system for the hospital / care house staff are the desktop application and the mobile app.

But letting them access to the complex classes and all other things of the implementation is not a good solution. They will not like this automatic system at all if that happens. This is not a good situation for us as well. Therefore, we need to hide this complexity and provide the users with a simpler interface.

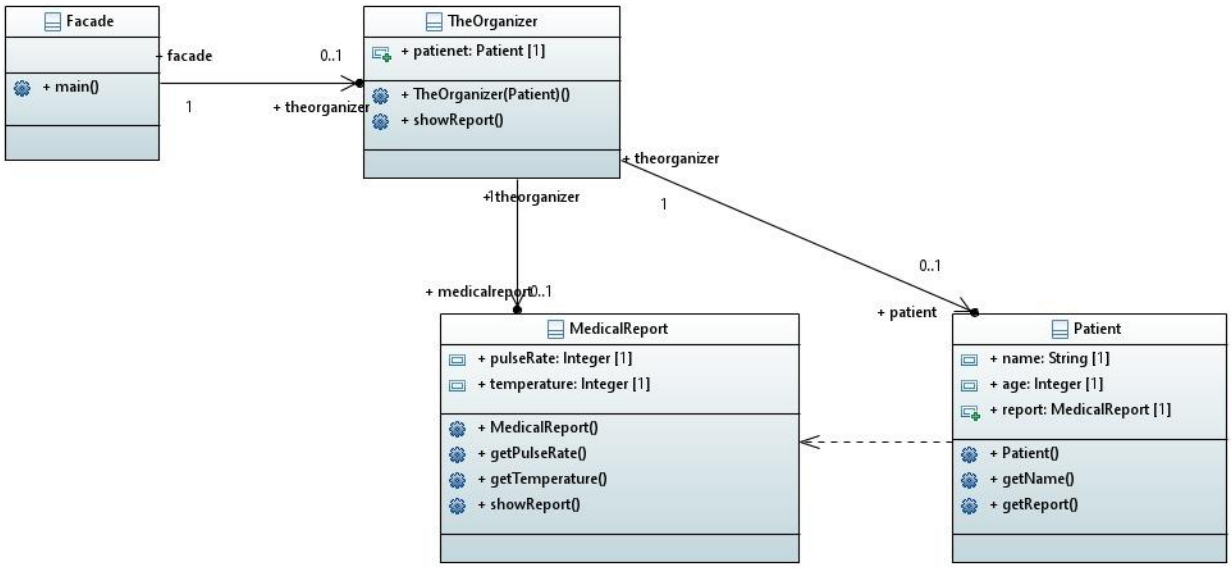
Every implementation details will be hidden in this interface and simple functions will be provided to get the job done. It is to make this interface easier, that we will develop a desktop GUI application. So that the users do not even have to work with the functions.

Solution

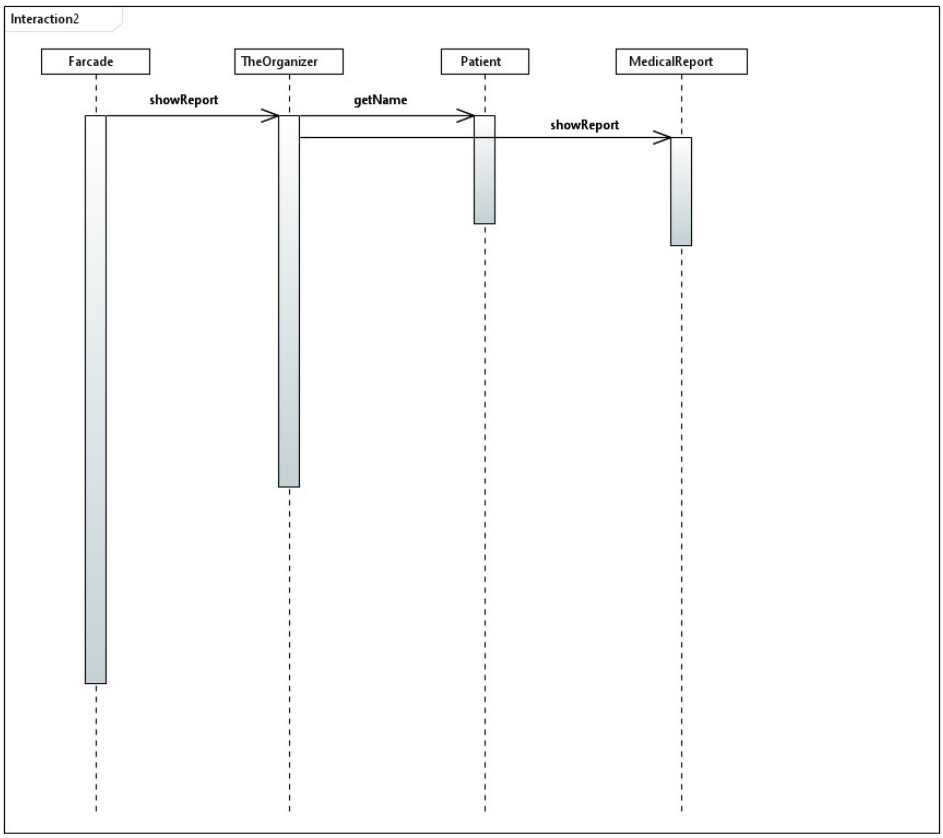
With this background, we decided to employee Façade design pattern to create an interface for the large subsystem of the wheelchair.

The class will help to decouple the source code from the interface and make the code base more arranged while making the code base easier to handle.

Class Diagram


Facade
+ main()
+ facade
1
0..1
+ theorganizer
TheOrganizer
+ patienet: Patient [1]
+ TheOrganizer(Patient)()
+ showReport()
+ theorganizer
1
+theorganizer
0..1
+ medicalreport
0..1
+ patient
MedicalReport
+ pulseRate: Integer [1]
+ temperature: Integer [1]
+ MedicalReport()
+ getPulseRate()
+ getTemperature()
+ showReport()
Patient
+ name: String [1]
+ age: Integer [1]
+ report: MedicalReport [1]
+ Patient()
+ getName()
+ getReport()

Sequence Diagram



Code snippet

```
class Patient{
    String name;
    int age;
    MedicalReport report;

    public Patient(String name, int age, MedicalReport report){
        this.name = name;
        this.age = age;
        this.report = report;
    }
```

```
    public String getName(){ return name;}
    public MedicalReport getReport() { return report;}
}

class MedicalReport{
    int pulseRate;
    int humidity;
    double bloodOxygenLevel;
    int sugarLevel;
    int temperature;

    public MedicalReport(){
        // in the actual implementation,
        //all the data initialized with the real time data.
        // the data that are collected from the sensors
        // will be read throught an interface using an "Adapter"
        // === dummy data ===
        pulseRate = 72;
        humidity = 23;
        bloodOxygenLevel = 21.2;
        sugarLevel = 102;
        temperature = 37;
    }

    public int getPulseRate(){
        return pulseRate;
    }

    public int getHumidity(){
        return humidity;
    }

    public double getBloodOxygenLevel(){
        return bloodOxygenLevel;
    }

    public int getSugarLevel(){
        return sugarLevel;
    }

    public int getTemperature(){
        return temperature;
    }
```

```
    public void showReport(){
        System.out.println("Pulse Rate = " + pulseRate);
        System.out.println("Humidity = " + humidity);
        System.out.println("Blood Oxygen Level = " + bloodOxygenLevel);
        System.out.println("Sugar Level = " + sugarLevel);
        System.out.println("Temperature = " + temperature);
    }

}

class TheOrganizer{
    private Patient patient;

    public TheOrganizer(Patient patient){
        this.patient=patient;
    }

    public void showReport(){
        System.out.println("============ MEDICAL REPORT ============");
        System.out.println("Name of the Patient : "+patient.getName());
        System.out.println("");
        patient.report.showReport();
    }
}

public class Facade{
    public static void main(String[] args) {
        MedicalReport report = new MedicalReport();
        Patient patient = new Patient("thilina",20,report);
        TheOrganizer organizer = new TheOrganizer(patient);
        organizer.showReport();
    }
}
```