CS2242 Lab 4

Advanced Smart Wheelchair System

Systems Requirements Specification Document

Prepared by -

1. I.T.A. Ilesinghe-*190238U*
2. N.M. Hafeel-*190211G*
3. M.H.A. Ahmed-*190028C*

Introduction

This document is based on a smart wheel chair embedded system project . The main purpose of proposed smart wheelchair is to automate the process of taking care of patients. The other main goal we have is to use the smart wheelchair to pay continuous attention to the patient without missing out any other important things.
Also, we wish to make the lives of the patients better with our project.

The requirements of the customer are as follows.

- Desktop application – shows all the patients' health details and other details.
- Android app – monitor one or few specific patients' health.
- Database – collection of real time details about patients.
- RFID &GPS locator – Collecting of real time locations of the wheelchair.
- Voice recognizer- Control the motion of the wheelchair.
- Gesture Control Unit – controls the motion of the wheelchair with hand gestures.
- Sensors – Detecting and avoiding obstacles.
  Collecting the real time details of a specific patient.
- Motors- To control the motion of a wheelchair
- power supply (chargeable and battery)

The purpose of this system is to design and build a Smart Wheelchair system which mainly focuses on automation and health concern of wheelchair users. In this implementation the smart wheelchairs will collect health related details and the location details and update it to a database. A desktop application and mobile app are designed in which access is given to the respective individuals. Voice command recognition will be used to achieve the auto motion requirement of the wheelchair.
The databases and access authorities can be modified using the desktop application.

To meet these requirements, we are willing to use an IoT implementation with the wheelchair. We will also employee various kind of sensors, controllers in the project.

The chair will have the ability to move on voice commands, gestures, and manually as well. This design is fail-safe since the manual control is also there. The data that is being gathered by the chair will be uploaded to a data base and update all the relevant parties simultaneously. The receivers of the data will expect responses at certain time intervals and will inform any unexpected behaviors hence making the system fail-safe. Health data will be checked with the normal health conditions and launch alerts to all the relevant parties if the condition gets abnormal. System will be able to identify some conditions like heart attacks directly. Also, a notification will be sent if the battery level of the wheelchair is low.

The patient will also be able to use hand gestures or voice commands to move chair. Also, there will be an emergency button for the patient to call the relevant parties.
We are trying to add a compartment to keep simple things for patient’s use as well.

Domain Class Diagram

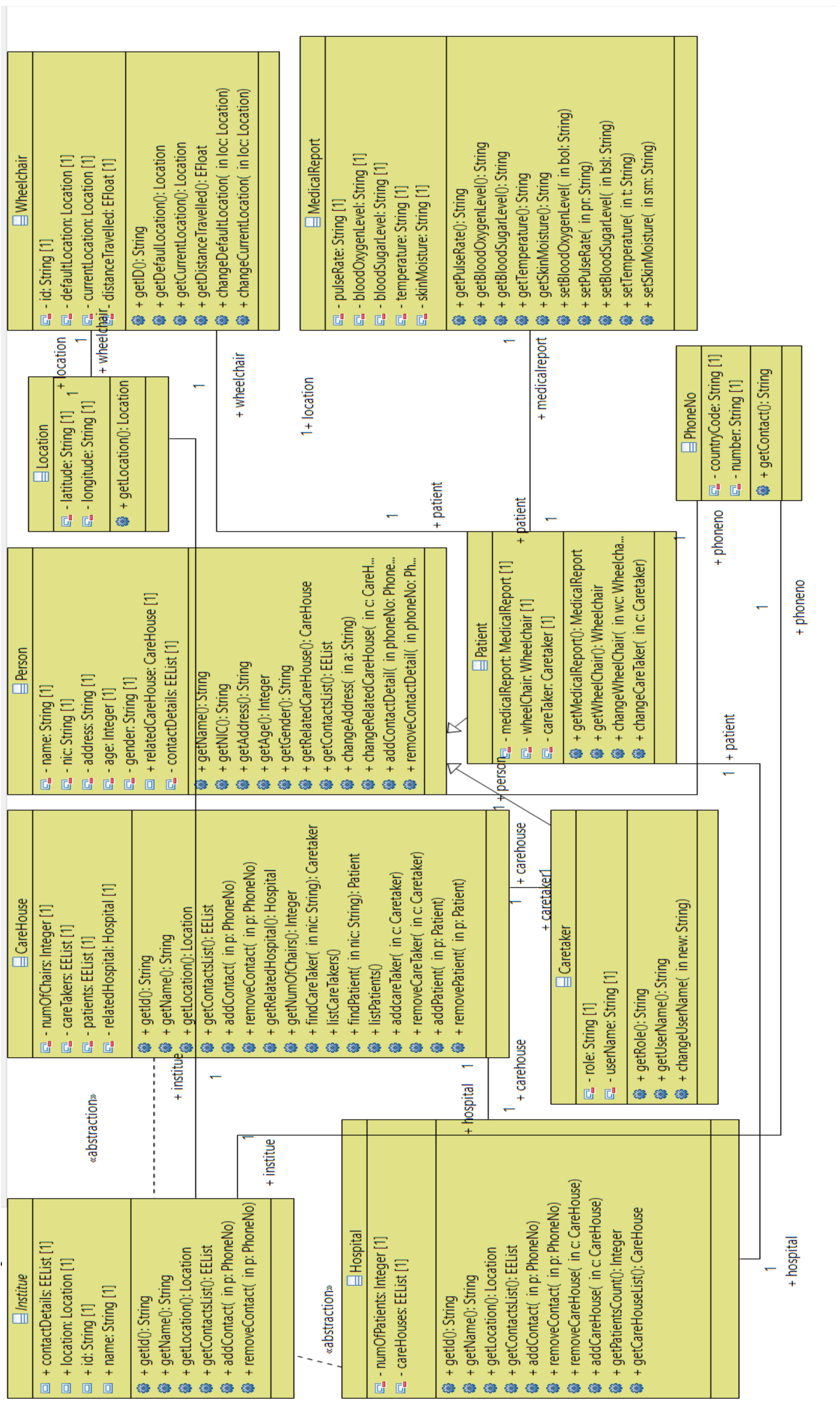

Wheelchair
- id: String [1]
- defaultLocation: Location [1]
- currentLocation: Location [1]
- distanceTravelled: EFloat [1]
+ getID(): String
+ getDefaultLocation(): Location
+ getCurrentLocation(): Location
+ getDistanceTravelled(): EFloat
+ changeDefaultLocation(in loc: Location)
+ changeCurrentLocation(in loc: Location)
MedicalReport
- pulseRate: String [1]
- bloodOxygenLevel: String [1]
- bloodSugarLevel: String [1]
- temperature: String [1]
- skinMoisture: String [1]
+ getPulseRate(): String
+ getBloodOxygenLevel(): String
+ getBloodSugarLevel(): String
+ getTemperature(): String
+ getSkinMoisture(): String
+ setBloodOxygenLevel(in bol: String)
+ setPulseRate(in pr: String)
+ setBloodSugarLevel(in bsl: String)
+ setTemperature(in t: String)
+ setSkinMoisture(in sm: String)
Location
- latitude: String [1]
- longitude: String [1]
+ getLocation(): Location
PhoneNo
- countryCode: String [1]
- number: String [1]
+ getContact(): String
Person
- name: String [1]
- nic: String [1]
- address: String [1]
- age: Integer [1]
- gender: String [1]
+ relatedCareHouse: CareHouse [1]
- contactDetails: EEList [1]
+ getName(): String
+ getNIC(): String
+ getAddress(): String
+ getAge(): Integer
+ getGender(): String
+ getRelatedCareHouse(): CareHouse
+ getContactsList(): EEList
+ changeAddress(in a: String)
+ changeRelatedCareHouse(in c: CareH...
+ addContactDetail(in phoneNo: Phone...
+ removeContactDetail(in phoneNo: Ph...
Patient
- medicalReport: MedicalReport [1]
- wheelChair: Wheelchair [1]
- careTaker: Caretaker [1]
+ getMedicalReport(): MedicalReport
+ getWheelChair(): Wheelchair
+ changeWheelChair(in wc: Wheelcha...
+ changeCareTaker(in c: Caretaker)
CareHouse
- numOfChairs: Integer [1]
- careTakers: EEList [1]
- patients: EEList [1]
- relatedHospital: Hospital [1]
+ getId(): String
+ getName(): String
+ getLocation(): Location
+ getContactsList(): EEList
+ addContact(in p: PhoneNo)
+ removeContact(in p: PhoneNo)
+ getRelatedHospital(): Hospital
+ getNumOfChairs(): Integer
+ findCareTaker(in nic: String): Caretaker
+ listCareTakers()
+ findPatient(in nic: String): Patient
+ listPatients()
+ addCareTaker(in c: Caretaker)
+ removeCareTaker(in c: Caretaker)
+ addPatient(in p: Patient)
+ removePatient(in p: Patient)
Caretaker
- role: String [1]
- userName: String [1]
+ getRole(): String
+ getUserName(): String
+ changeUserName(in new: String)
Institue
+ contactDetails: EEList [1]
+ location: Location [1]
+ id: String [1]
+ name: String [1]
+ getId(): String
+ getName(): String
+ getLocation(): Location
+ getContactsList(): EEList
+ addContact(in p: PhoneNo)
+ removeContact(in p: PhoneNo)
Hospital
- numOfPatients: Integer [1]
- careHouses: EEList [1]
+ getId(): String
+ getName(): String
+ getLocation(): Location
+ getContactsList(): EEList
+ addContact(in p: PhoneNo)
+ removeContact(in p: PhoneNo)
+ removeCareHouse(in c: CareHouse)
+ addCareHouse(in c: CareHouse)
+ getPatientsCount(): Integer
+ getCareHouseList(): CareHouse
+ location
+ wheelchair
+ wheelchair
1 + location
+ patient
+ medicalreport
+ patient
+ phoneno
+ phoneno
+ patient
+ person
+ carehouse
+ caretaker
+ carehouse
+ institue
+ hospital
«abstraction»
+ institue
«abstraction»
+ hospital
1

Figure 01 : Class Diagram of a smart wheelchair system

Use Case Diagram

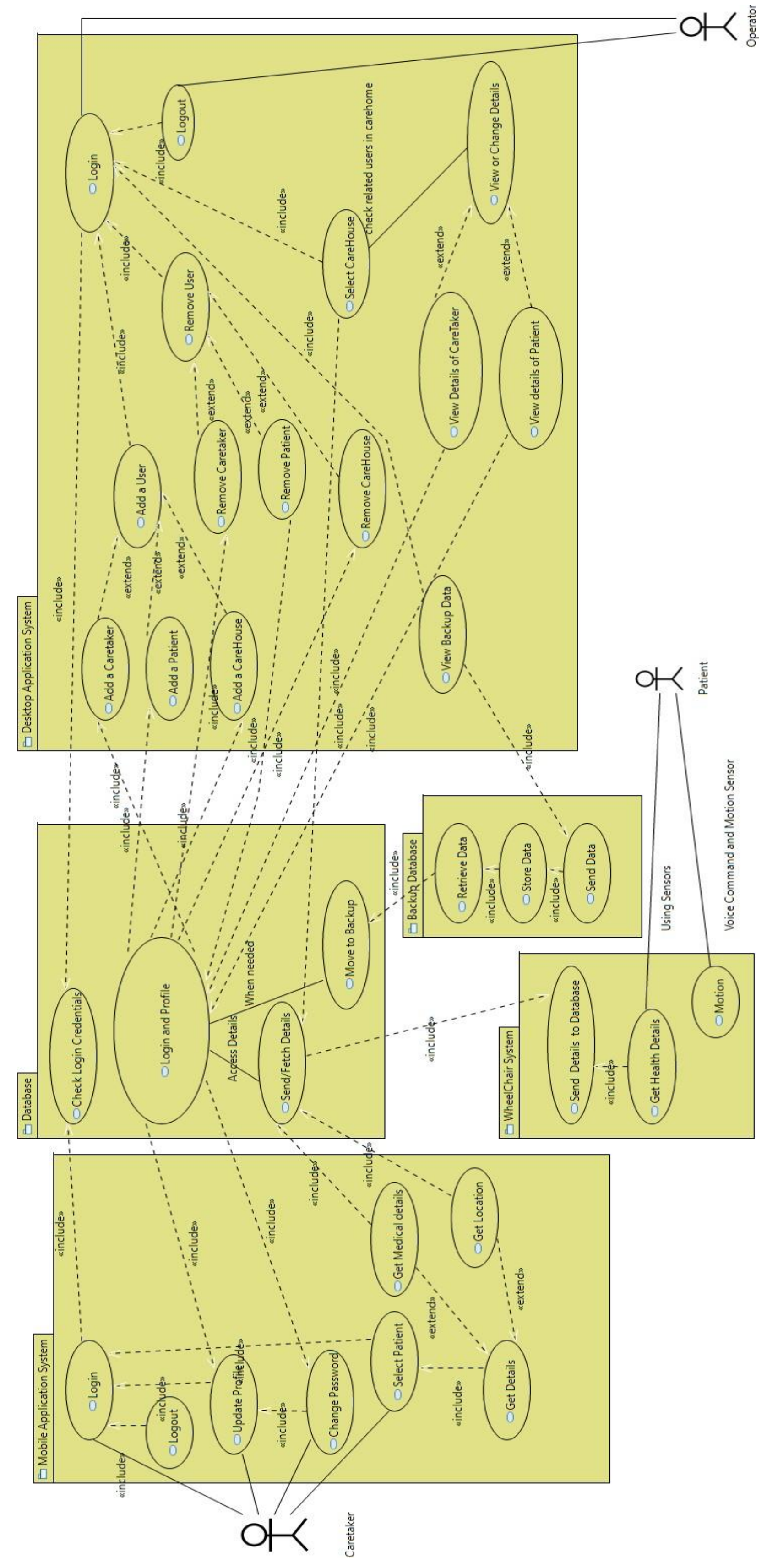


Figure 02: Use Case Diagram of a smart wheelchair system

Detailed Use Case Descriptions

Mobile Application System Package

USE CASE	**Login**
DESCRIPTION	Logging into the application to access the features of the mobile application.
GOAL	The caretaker can select a patient to view details.
ACTORS	Caretaker
CONDITIONS	➢ Need to be pre-registered to the system. ➢ Valid Login credentials should be used
FLOW OF EVENTS	The use case starts when the user opens the application. The user is prompted to input his login credentials and based on the validity of the login credentials , the user is logged into the application to access his profile .
ISSUES	➢ Unable to add a patient to the system. ➢ Unable to alter details of the database.
CONNECTED PACKAGES	Database

USE CASE	**Logout**
DESCRIPTION	Logging out of the application. Thus, the features of the mobile application are not accessible.
GOAL	The caretaker can stop from accessing the application's features.
ACTORS	Caretaker
CONDITIONS	➢ Need to be logged into the system.
FLOW OF EVENTS	The use case starts when the logged in user presses the logout button. Then, the user is logged out of the application to back to the application's login page.
ISSUES	None

USE CASE	**Update Profile**
DESCRIPTION	Change the details of the caretaker's profile . The new profile details would be update in the database.
GOAL	The caretaker can change his details. ➢ Add/remove a contact detail ➢ Change his username and password. ➢ Change his profile picture.
ACTORS	Caretaker
CONDITIONS	➢ Need to be logged into to the system. ➢ Need to press the 'Edit profile' button for redirection to the intended page of the application.
FLOW OF EVENTS	The use case starts when the user presses the 'Edit Profile' button the application. Then the user can change the intended details.
ISSUES	➢ Unable to change the related hospital and other fixed details.
CONNECTED PACKAGES	Database

USE CASE	**Change Password**
DESCRIPTION	Changing the login password of his/her profile if the existing password was exposed or as per need of the user. The new password would be updated in the database.
GOAL	The caretaker can create an alphanumeric password and confirm the password.
ACTORS	Caretaker
CONDITIONS	➢ Need to know the current password to be qualified for password changing feature of the application.
FLOW OF EVENTS	The use case starts when the user presses the change password button in the profile page. Then the caretaker is redirected to change password page .

ISSUES	➢ Get a code to change the password to the contact numbers provided to change the password upon forgetting the password.
CONNECTED PACKAGES	Database

USE CASE	**Select Patient**
DESCRIPTION	Select a patient out of all the patients under the care of the caretaker.
GOAL	The caretaker can select the patient of whom he needs to get details of.
ACTORS	Caretaker
CONDITIONS	➢ Need to know the name of the patient of which the user needs the detail.
FLOW OF EVENTS	The use case starts when the user presses the select patient button. Then , the user selects a patient and views his / her details.
ISSUES	None

USE CASE	**Get Details**
DESCRIPTION	Get location and medical details of the selected patient from the local buffer storage. The details in the local buffer storage are those retrieved from the database at refreshed intervals.
GOAL	The caretaker can view the details of the patient.
ACTORS	Caretaker
CONDITIONS	None
FLOW OF EVENTS	The use case starts when the user presses the button with the patient’s name. Then, the user views the patient’s details.

EXTENSIONS	➢ Get Medical Details: This would provide the details of the past seven days. ➢ Get Location Details: This would provide the recent distinct locations.
CONNECTED PACKAGES	Database

Database Package

USE CASE	**Check Login Credentials**
DESCRIPTION	The validity of the login credentials for both the mobile and desktop applications is checked to grant access to the features of the system.
GOAL	Check whether the login credentials entered by the user matches with the login credentials in the database if exist.
ACTORS	No direct connection with any of the actors. Indirect connection with Caretaker and Operator
CONDITIONS	Need to be updated with latest login credentials.
FLOW OF EVENTS	The use case starts when the user enters the login credentials and presses the login button of the application systems' interface. Then the validity of the credentials are checked if exiting.
ISSUES	None
CONNECTED PACKAGES	➢ Mobile Application System ➢ Desktop Application System

USE CASE	**Login and Profile**
DESCRIPTION	Updating of the database with the latest changes in profile of a user of the application systems.
GOAL	Updating the database instantly upon any changes to a user profile.
ACTORS	No direct connection with any of the actors. Indirect connection with Caretaker and Operator
CONDITIONS	Need to be updated with latest profile information instantly.
FLOW OF EVENTS	The use case starts when the user submits the changes to his/her profile in the application systems. Then, the database is updated with the changes.
ISSUES	None
CONNECTED PACKAGES	➢ Mobile Application System ➢ Desktop Application System

USE CASE	**Send/Fetch Details**
DESCRIPTION	The details from the wheelchair are updated to the database and retrieved to local buffer storages of the devices using the application system.
GOAL	Updating the database instantly upon any changes in sensors mounted on wheelchairs that monitor the patient.
ACTORS	No direct connection with any of the actors. Indirect connection with Caretaker, Patient and Operator
CONDITIONS	Need to be updated with latest medical information instantly.
FLOW OF EVENTS	The use case starts when changes are observed in a wheelchair sensors. Then, the database is updated with the changes.
ISSUES	None

CONNECTED PACKAGES	➢ Mobile Application System ➢ Desktop Application System ➢ Wheelchair System

USE CASE	**Move to Backup**
DESCRIPTION	The details from the primary database are moved to the backup database upon a delete request from the desktop Application System.
GOAL	Transferring of data from primary database to backup database for later view. Make the user unable to access the features of the system.
ACTORS	No direct connection with any of the actors. Indirect connection with Operator
CONDITIONS	Need to store only required details.
FLOW OF EVENTS	The use case starts when the operator has requested to delete a user of the system. The details of the user is moved to the backup database.
ISSUES	None
CONNECTED PACKAGES	Backup Database

Backup Database Package

USE CASES	**Retrieve Data, Store Data and Send Data**
DESCRIPTION	The details from the primary database are retrieved to the backup database and stored. The Stored details are accessed by the operator if needed.
GOAL	Use the details stored when needed.
ACTORS	No direct connection with any of the actors.
CONDITIONS	Need to store only required details.
FLOW OF EVENTS	The use case starts when the operator has requested to delete a user of the system and the move to backup is initiated. The details of the user are moved to the backup database .
ISSUES	The storage capacity of the backup database should be concerned.
CONNECTED PACKAGES	Database

Wheelchair Package

USE CASE	**Send Details to Database**
DESCRIPTION	The details from the wheelchair are updated to the database instantly.
GOAL	Updating the database instantly upon any changes in sensors mounted on wheelchairs that monitor the patient.
ACTORS	No direct connection with any of the actors. Indirect connection with Patient
CONDITIONS	Need to be updated with latest medical information instantly.

FLOW OF EVENTS	The use case starts when changes observed in the wheelchair sensors are received . Then, the database is updated with the changes.
ISSUES	None
CONNECTED PACKAGES	Database

USE CASE	**Get Details**
DESCRIPTION	The health details of the patient are obtained using sensors. The blood oxygen level, blood sugar level, temperature, skin moisture and pulse rate are obtained from the patient along with the location of the wheelchair from the wheelchair itself.
GOAL	Monitor the patient's health and location of the wheelchair.
ACTORS	Patient
CONDITIONS	Need to monitor the status of each sensor. Need to be signaling whether the state has changed at fixed time intervals.
FLOW OF EVENTS	The use case starts when changes are observed in a wheelchair sensor. Then, the database is updated with the changes.
ISSUES	Would probably have to connect the sensors to a central server which would monitor the status of each sensor. Dependent on how the sensors worked they would either send a signal when their state changed, at fixed time intervals, or the server would have to poll input. Once a change has been sent/received by the server it would update a database of all sensor states.
CONNECTED PACKAGES	None

USE CASE	**Motion**

DESCRIPTION	The motion of the wheelchair is monitored using voice commands from the patient.
GOAL	➢ Auto obstacle detection and avoiding ➢ Monitor the motion using specific voice commands ➢ Tracking the distance travelled
ACTORS	Patient
CONDITIONS	Motion cannot exceed a specified speed limit.
FLOW OF EVENTS	The use case starts when changes are observed in a wheelchair sensor. Then, the database is updated with the changes.
ISSUES	Obstacle detection should be accurate and responsive.
CONNECTED PACKAGES	None

Desktop Application System Package

USE CASE	**Login**
DESCRIPTION	Logging into the application to access the features of the desktop application.
GOAL	The operator can view details of , add or remove Care house and users from the system.
ACTORS	Operator
CONDITIONS	➢ Valid Login credentials should be used
FLOW OF EVENTS	The use case starts when the user opens the application. The user is prompted to input his login credentials and based on the validity of the login credentials, the user is logged into the application to access .
ISSUES	None

CONNECTED PACKAGES	Database

USE CASE	**Logout**
DESCRIPTION	Logging out of the application. Thus, the features of the desktop application are not accessible.
GOAL	The operator can stop from accessing the application's features.
ACTORS	Operator
CONDITIONS	➢ Need to be logged into the system.
FLOW OF EVENTS	The use case starts when the logged in user presses the logout button. Then, the user is logged out of the application to back to the application's login page.
ISSUES	None

USE CASE	**Add User**
DESCRIPTION	Addition of a new user or user collection to the system database.
GOAL	➢ Add the user to the database inputting the details needed to do so.
ACTORS	Operator
CONDITIONS	Need to collect details of user prior to adding a user.
FLOW OF EVENTS	The use case starts when operator presses the add User button. Then, the operator is prompted to add the required details and proceed.
ISSUES	None
EXTENSIONS	➢ Add Caretaker:

	The personal details of the caretaker and the care house to which he/she is assigned is added to the database. ➢ Add Patient: The personal details of the patient and the caretaker to which he/she is assigned is added to the database. ➢ Add CareHouse: The details of the care house to which he/she is assigned is added to the database.
CONNECTED PACKAGES	Database

USE CASE	**Remove User**
DESCRIPTION	Removal of an existing user or user collection from the system database.
GOAL	➢ Remove the user from the database inputting the details needed to do so.
ACTORS	Operator
CONDITIONS	Need to know the details of user prior to removal of a user.
FLOW OF EVENTS	The use case starts when operator presses the remove User button. Then, the operator is prompted to add the required details and proceed.
ISSUES	The removal of details from the primary database are moved to the backup database.
EXTENSIONS	➢ Remove Caretaker:

	The details of the caretaker is moved from the primary database to the backup database and the details of care house to which he/she is assigned is altered in the database. ➢ Remove Patient: The personal details of the patient is moved from the primary database to the backup database and the details of care taker to which he/she is assigned is altered in the database. ➢ Remove CareHouse: The details of the care house is removed altering the details of caretakers and patients. This could be done only upon confirmation of from then care house as it affects all the users assigned under the care house.
CONNECTED PACKAGES	Database

USE CASE	**View Backup data**
DESCRIPTION	Viewing of the data in the backup database
GOAL	Viewing of the data in the backup database upon any emergency need.
ACTORS	Operator
CONDITIONS	None
FLOW OF EVENTS	The use case starts when operator presses the view backup button. Then, the operator is prompted to confirm and view details.
ISSUES	The removal of details from the primary database are moved to the backup database.
CONNECTED PACKAGES	Backup database

USE CASE	View Backup data
DESCRIPTION	Viewing of the data in the backup database
GOAL	Viewing of the data in the backup database upon any emergency need.
ACTORS	Operator
CONDITIONS	None
FLOW OF EVENTS	The use case starts when operator presses the view backup button. Then, the operator is prompted to confirm and view details.
ISSUES	The removal of details from the primary database are moved to the backup database.
CONNECTED PACKAGES	Backup database

USE CASE	**Select CareHouse**
DESCRIPTION	Selecting a care house to view/change details .
GOAL	➢ View or change details of users within the care house.
ACTORS	Operator
CONDITIONS	Need to know the NIC of user prior to view or change.
FLOW OF EVENTS	The use case starts when operator presses the select CareHouse button. Then, the operator is prompted to view or change details .
ISSUES	None
CONNECTED PACKAGES	Database

USE CASE	**View/Change Details**
DESCRIPTION	View/Change details of caretakers or patients
GOAL	View/Change details of caretakers or patients
ACTORS	Operator
CONDITIONS	Need to know the NIC of user prior to proceed.
FLOW OF EVENTS	The use case starts when operator presses the view/change details button. Then, the operator is prompted to select user type and proceed.
ISSUES	Only Certain details are permitted to be changed/
EXTENSIONS	➢ View/Change details of caretaker: The details of the caretaker is viewed or changed from/in the primary database. ➢ View/Change details of patient: The details of the patient is viewed or changed from/in the primary database.
CONNECTED PACKAGES	Database

System Sequence Diagram

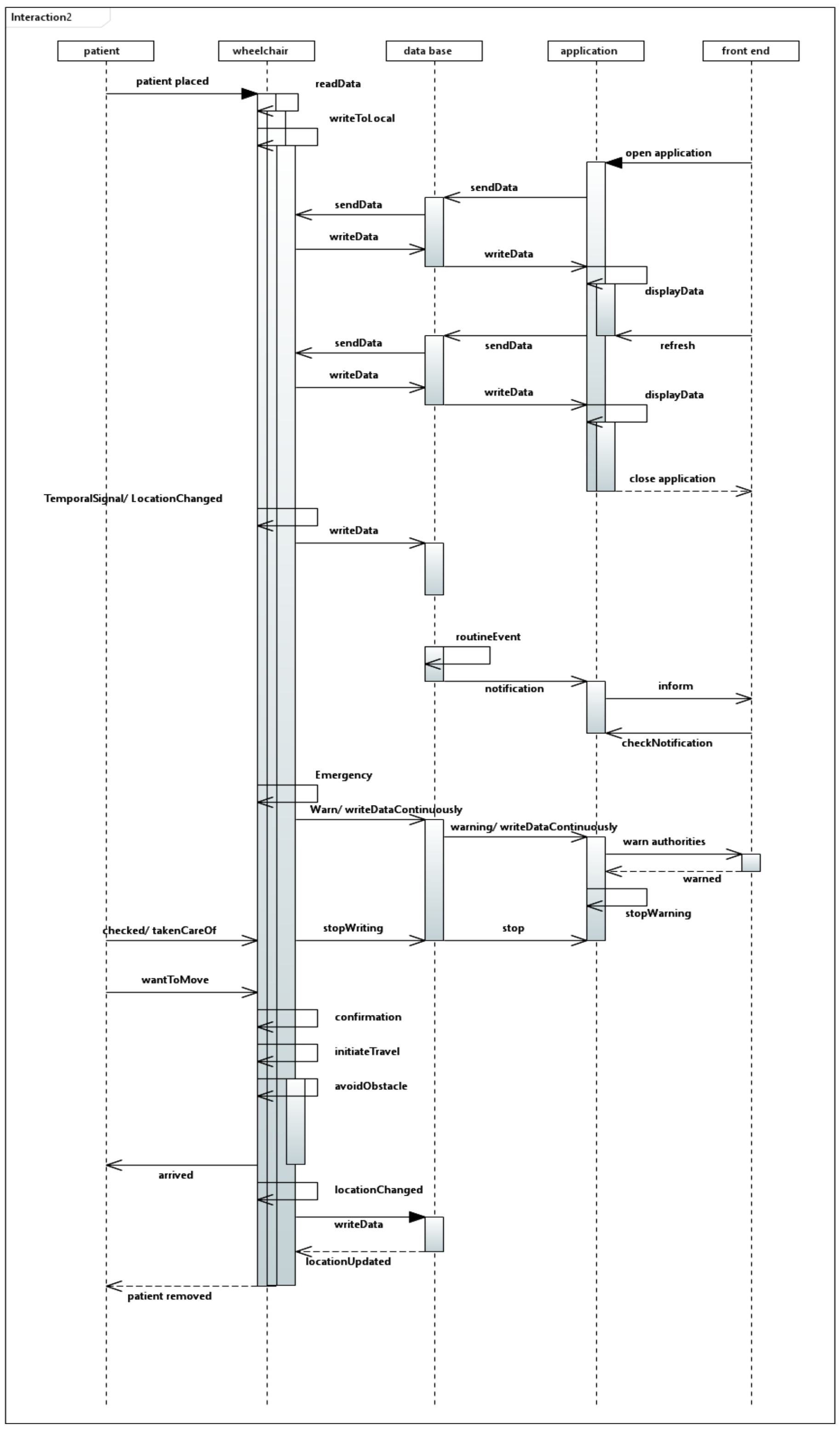


Figure 03: System Sequence Diagram of a smart wheelchair system

State Chart

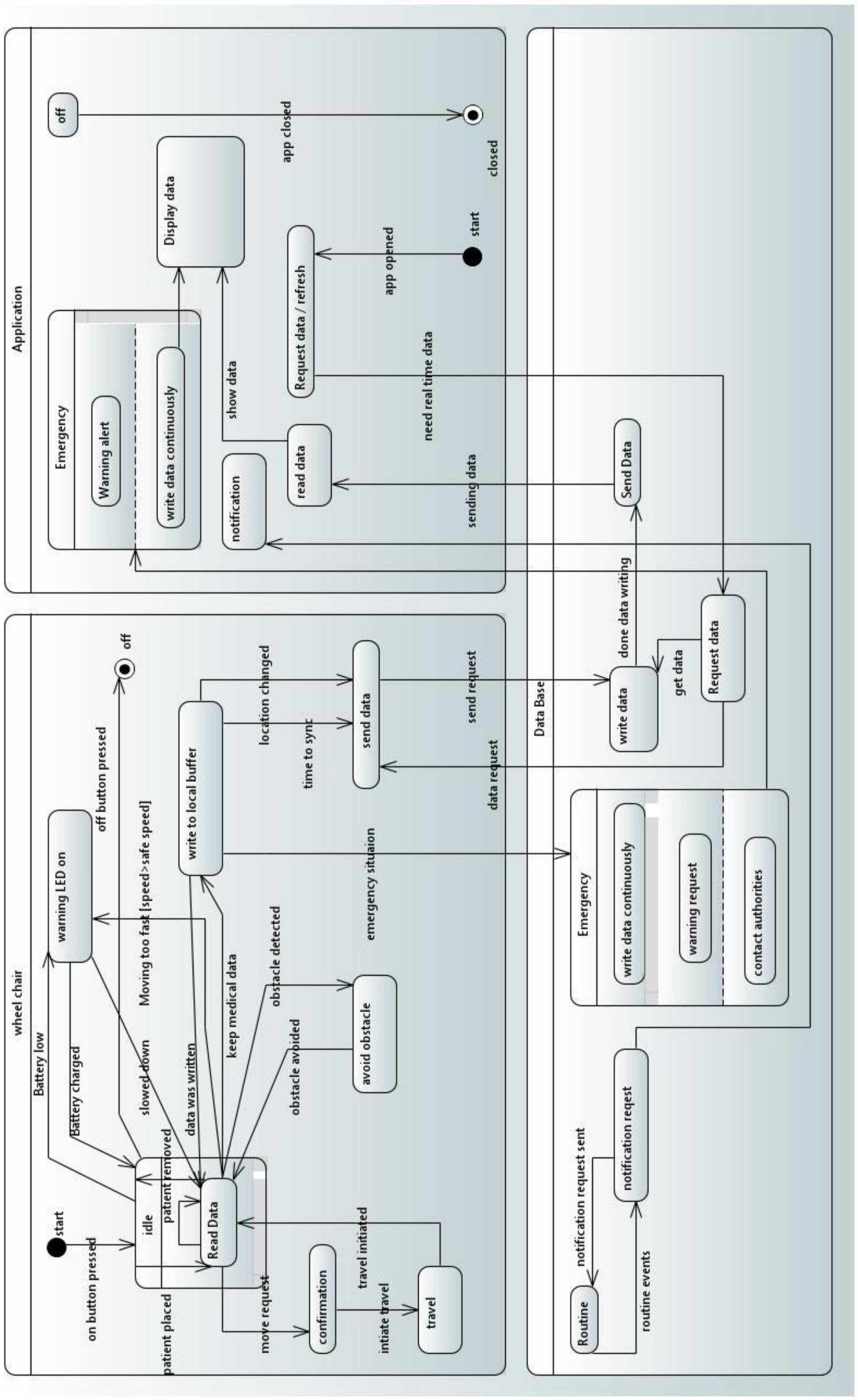


Figure 04: State Chart of a smart wheelchair system

Activity Diagrams

Add Patient Activity

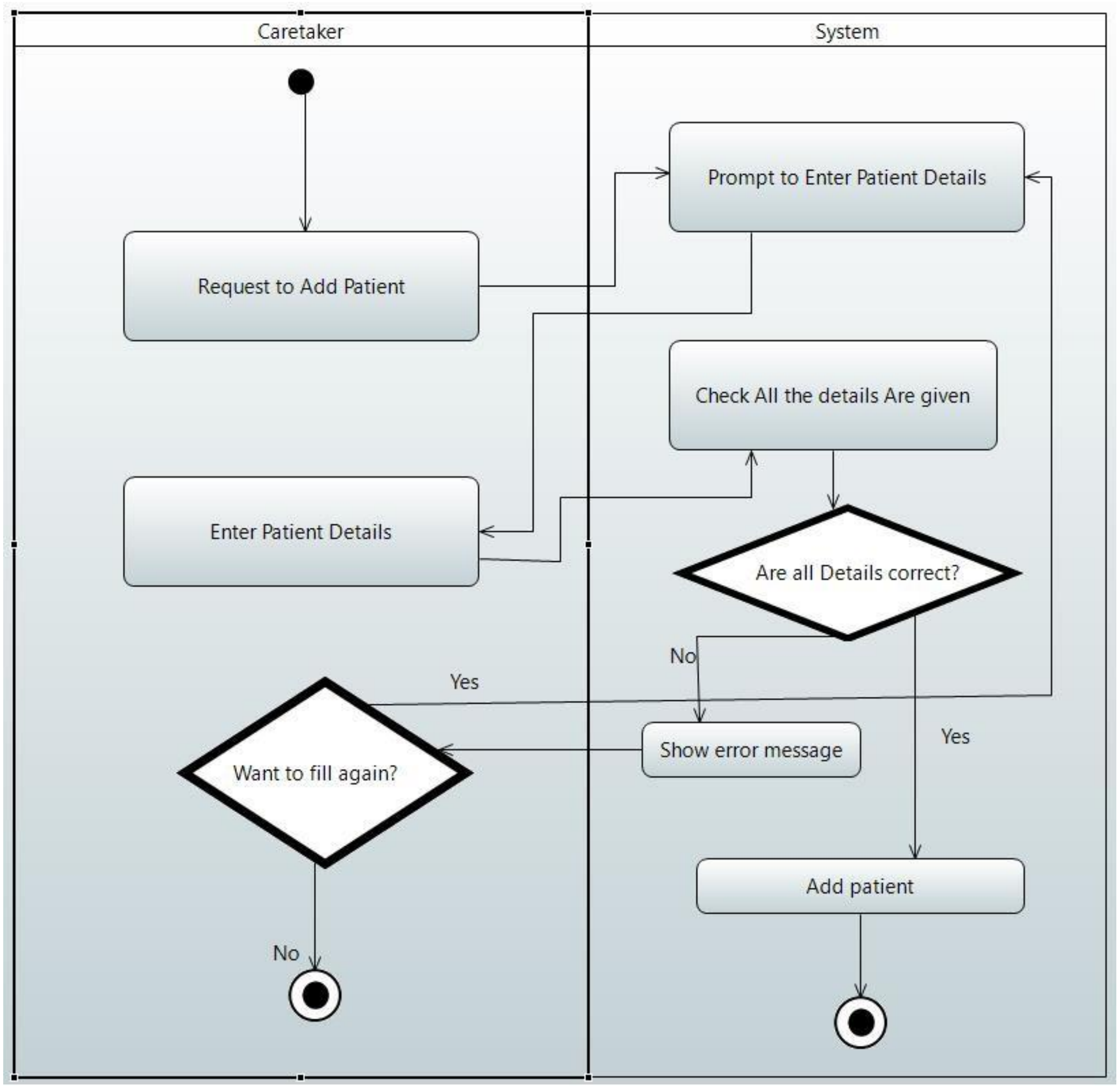


Figure 05: Activity Diagram for adding a patient to the system

Create User Account Activity

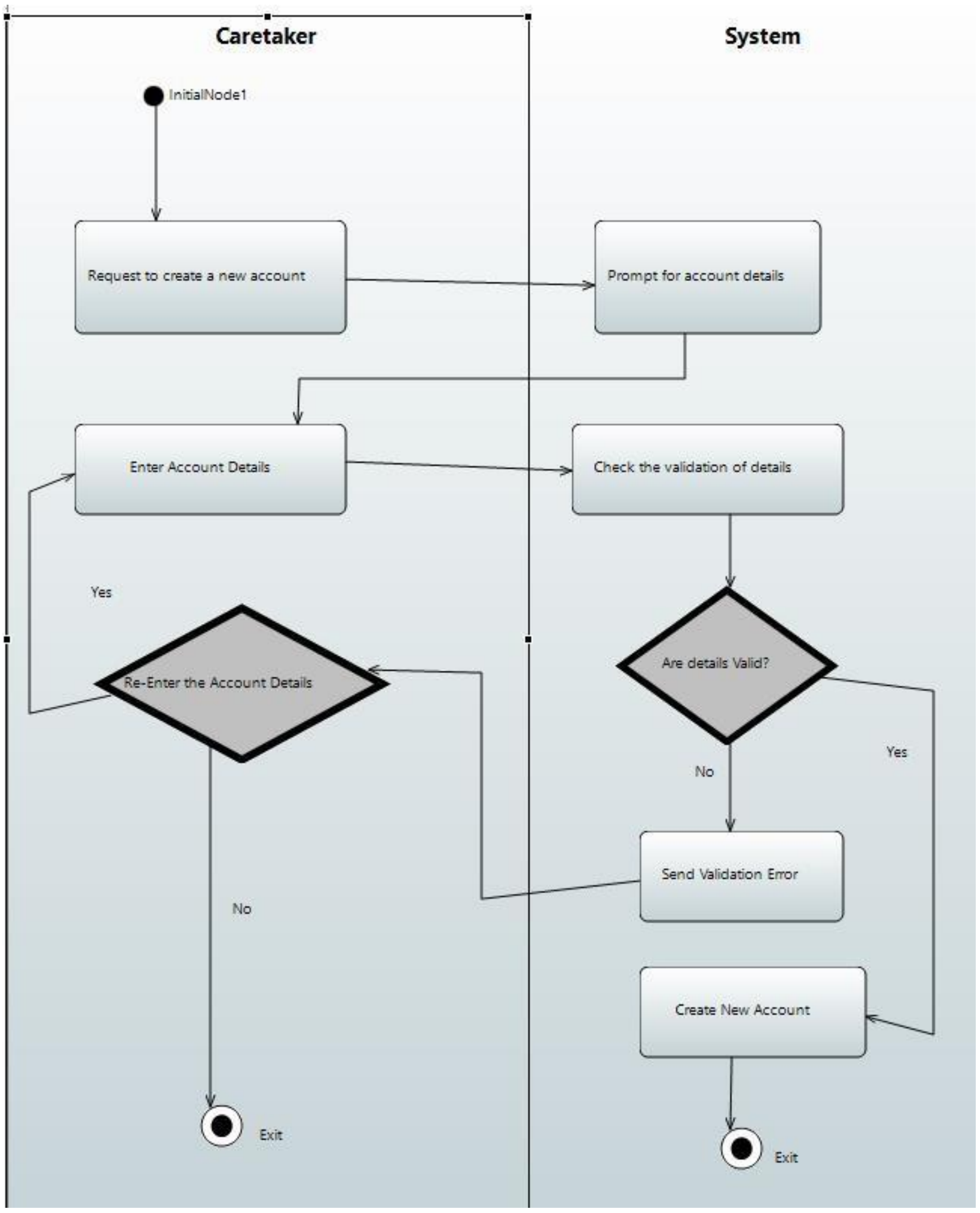


Figure 06: Activity Diagram for Creating an account for a user of the system

Login page Activity

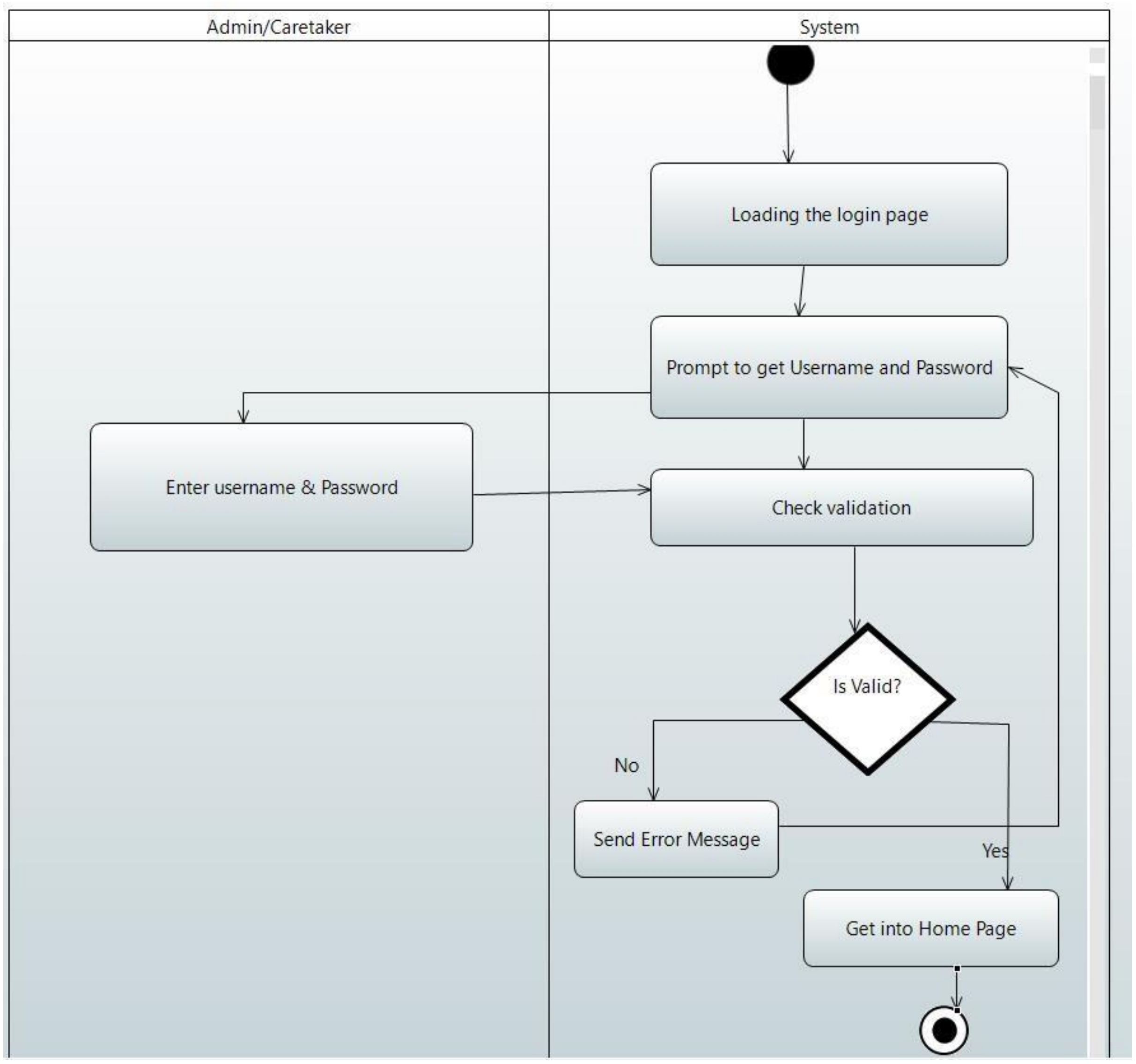


Figure 07: Activity Diagram for logging of a user into the system

Remove Patient Activity

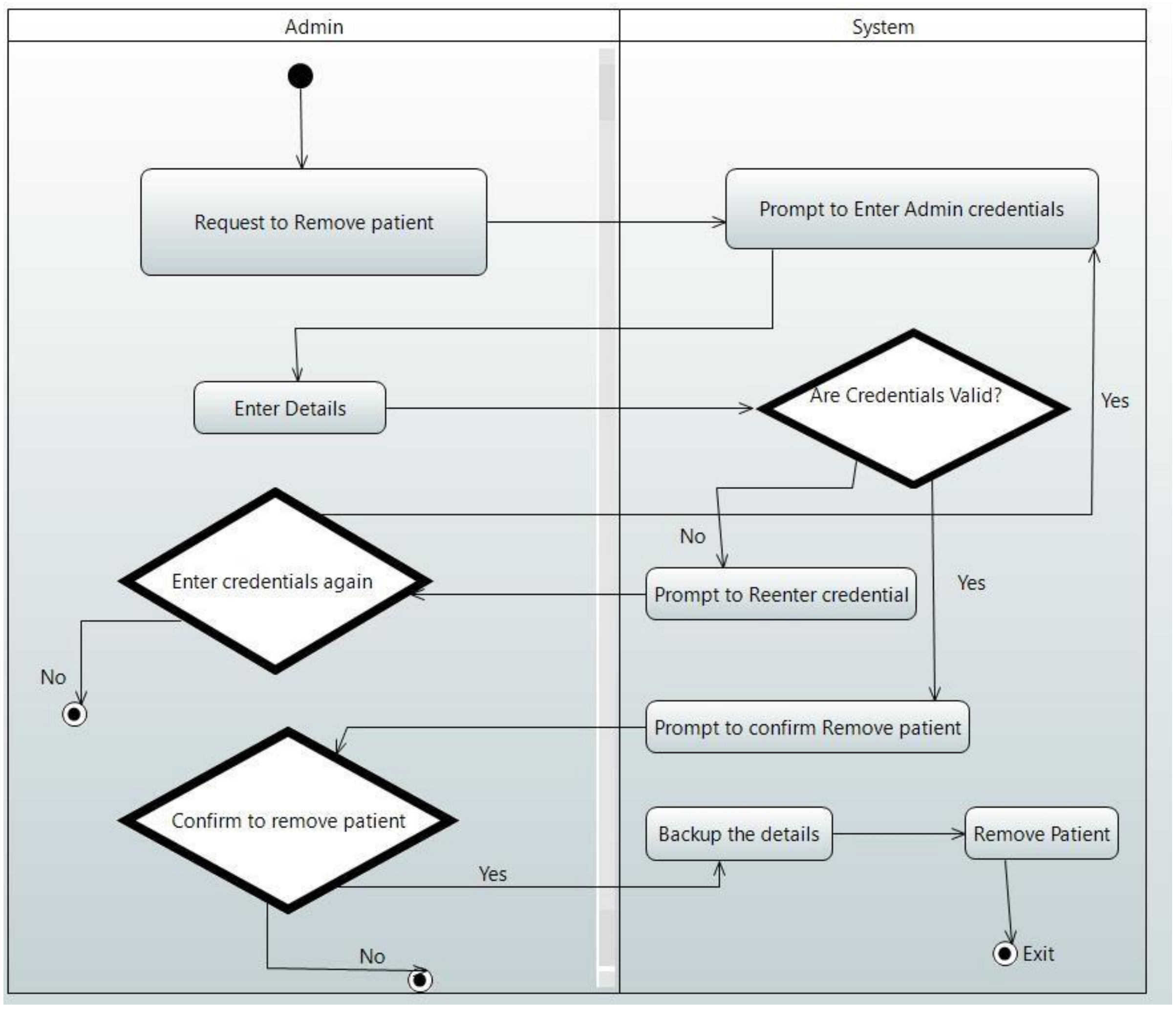

Admin
System
Request to Remove patient
Prompt to Enter Admin credentials
Enter Details
Are Credentials Valid?
Yes
No
Enter credentials again
Prompt to Reenter credential
Yes
No
Prompt to confirm Remove patient
Confirm to remove patient
Backup the details
Remove Patient
Yes
No
Exit

Figure 08: Activity Diagram for removing a patient from the system

Remove User Account Activity

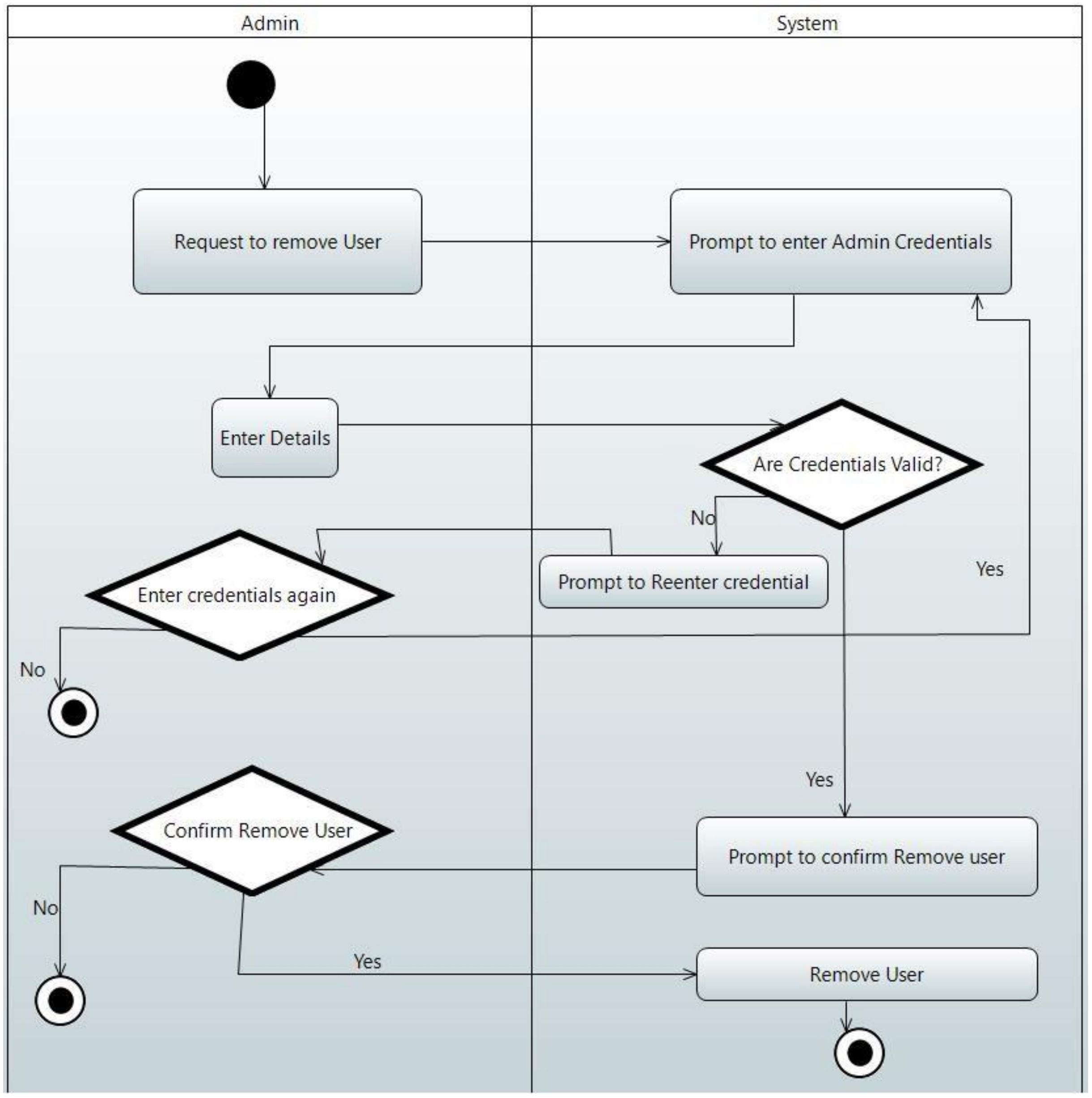


Figure 09: Activity Diagram for removing an account of a user from the system

Update Patient Activity

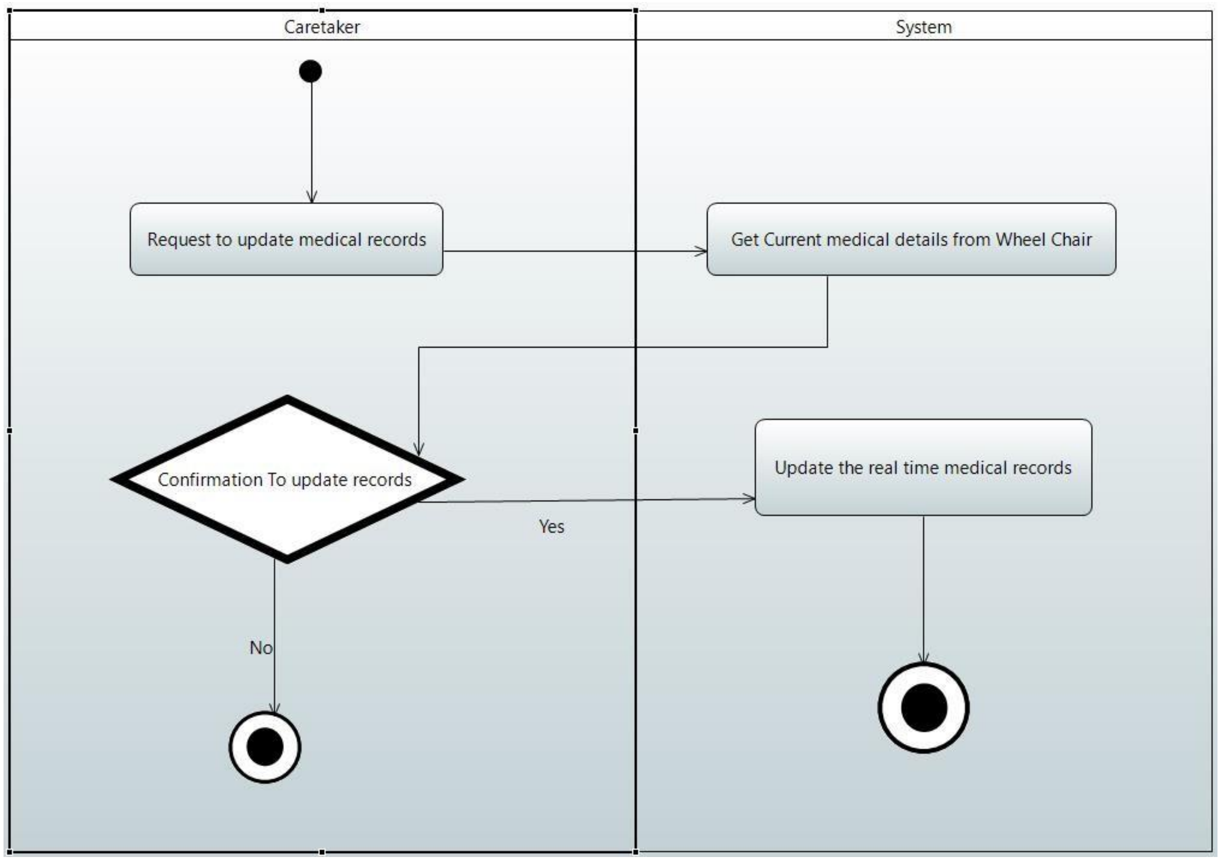

Caretaker
System
Request to update medical records
Get Current medical details from Wheel Chair
Confirmation To update records
Update the real time medical records
Yes
No

Figure 10: Activity Diagram for updating medical records to the system

View Patient's Details Activity

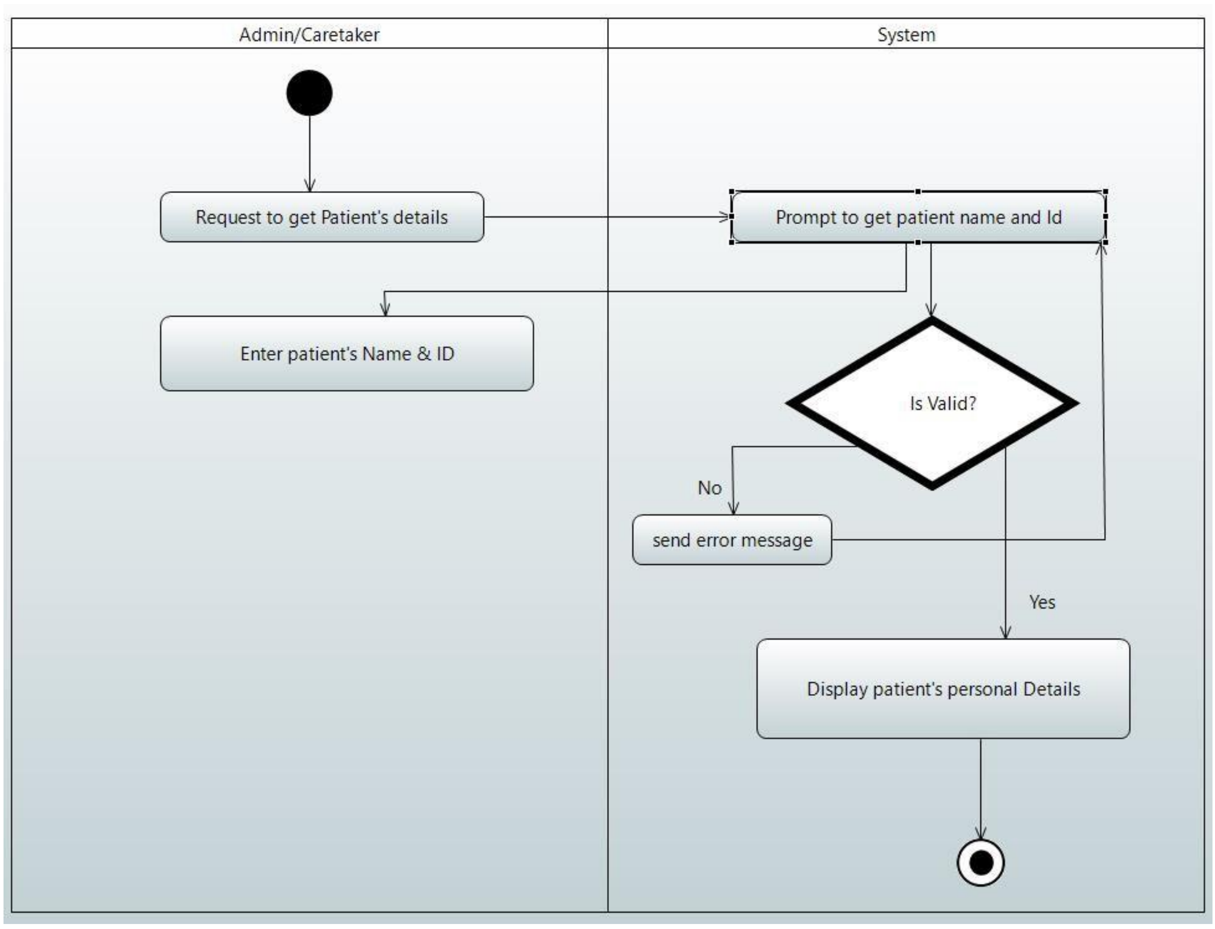

Admin/Caretaker
System
Request to get Patient's details
Prompt to get patient name and Id
Enter patient's Name & ID
Is Valid?
No
send error message
Yes
Display patient's personal Details

Figure 11: Activity Diagram for viewing patient’s personal details from the system

View Patient's Location Activity

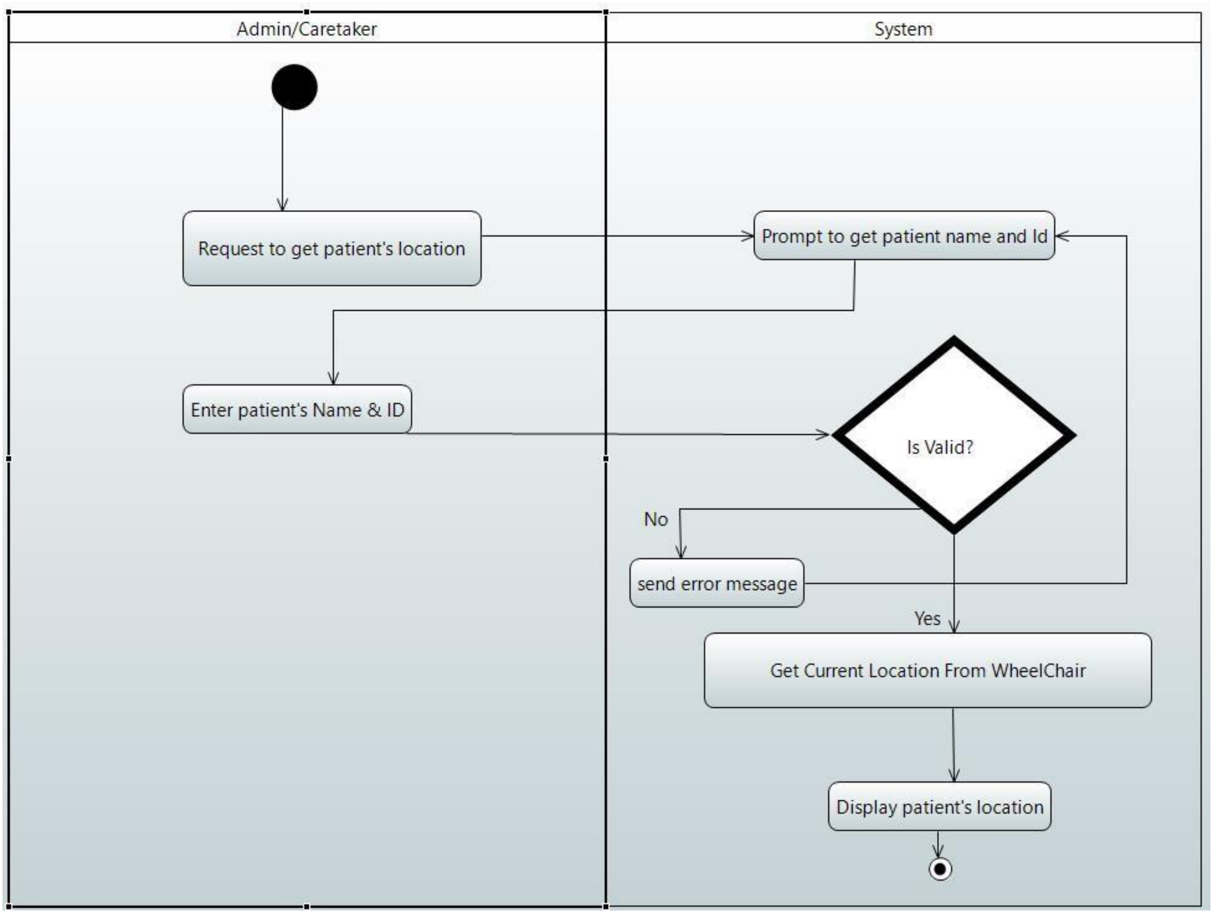

Admin/Caretaker
System
Request to get patient's location
Prompt to get patient name and Id
Enter patient's Name & ID
Is Valid?
No
send error message
Yes
Get Current Location From WheelChair
Display patient's location

Figure 12: Activity Diagram for viewing location of patient from the system

View Patient's Medical Record Activity

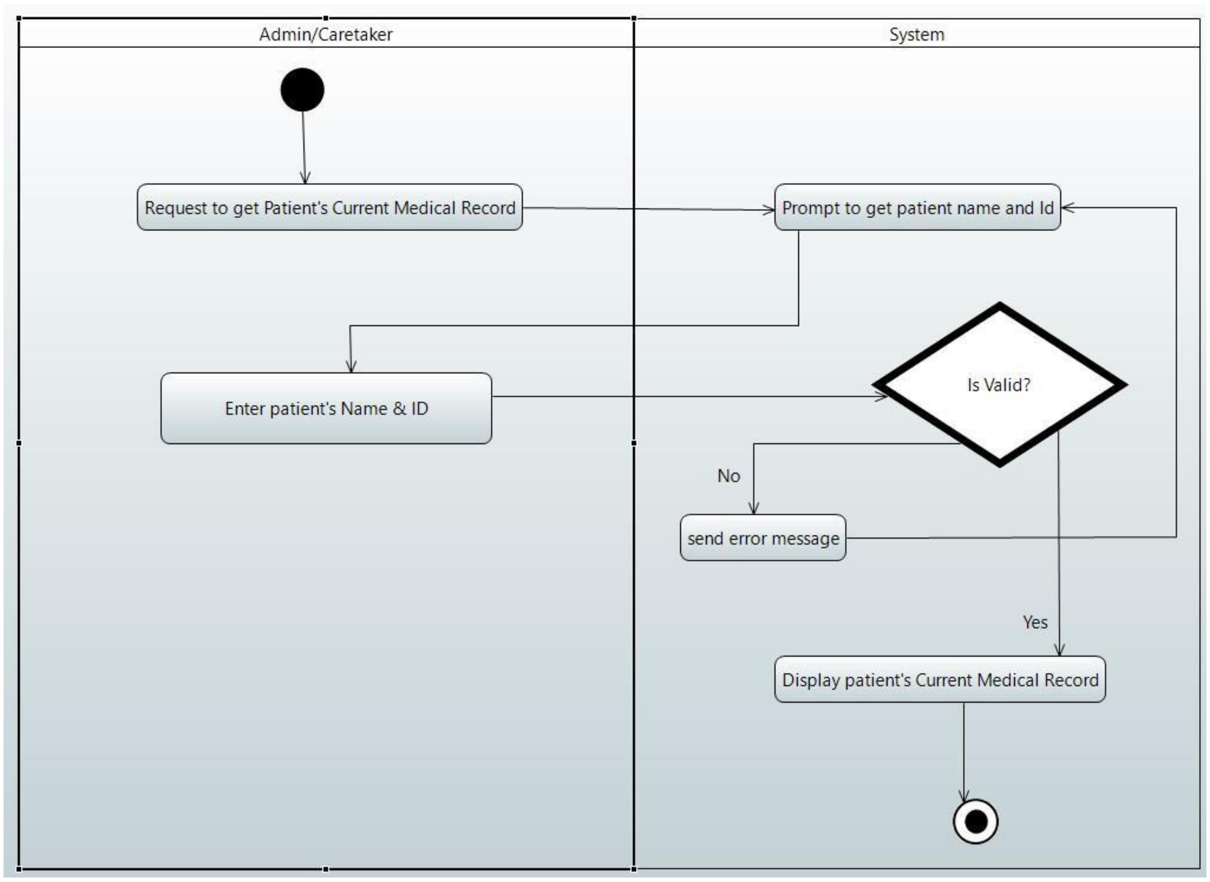
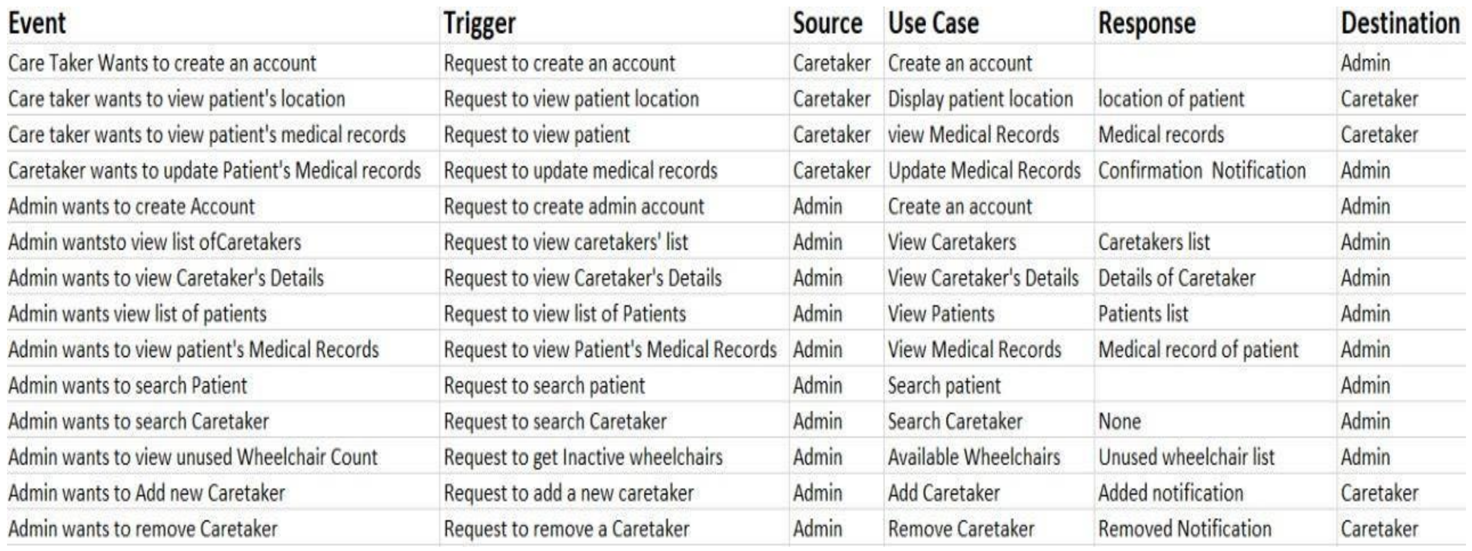

Admin/Caretaker
System
Request to get Patient's Current Medical Record
Prompt to get patient name and Id
Enter patient's Name & ID
Is Valid?
No
send error message
Yes
Display patient's Current Medical Record

Figure 13: Activity Diagram for viewing medical records from the system

Event Tables

Table 01: Event Table of a smart wheelchair system



Event	Trigger	Source	Use Case	Response	Destination
Care Taker Wants to create an account	Request to create an account	Caretaker	Create an account		Admin
Care taker wants to view patient's location	Request to view patient location	Caretaker	Display patient location	location of patient	Caretaker
Care taker wants to view patient's medical records	Request to view patient	Caretaker	view Medical Records	Medical records	Caretaker
Caretaker wants to update Patient's Medical records	Request to update medical records	Caretaker	Update Medical Records	Confirmation Notification	Admin
Admin wants to create Account	Request to create admin account	Admin	Create an account		Admin
Admin wantsto view list ofCaretakers	Request to view caretakers' list	Admin	View Caretakers	Caretakers list	Admin
Admin wants to view Caretaker's Details	Request to view Caretaker's Details	Admin	View Caretaker's Details	Details of Caretaker	Admin
Admin wants view list of patients	Request to view list of Patients	Admin	View Patients	Patients list	Admin
Admin wants to view patient's Medical Records	Request to view Patient's Medical Records	Admin	View Medical Records	Medical record of patient	Admin
Admin wants to search Patient	Request to search patient	Admin	Search patient		Admin
Admin wants to search Caretaker	Request to search Caretaker	Admin	Search Caretaker	None	Admin
Admin wants to view unused Wheelchair Count	Request to get Inactive wheelchairs	Admin	Available Wheelchairs	Unused wheelchair list	Admin
Admin wants to Add new Caretaker	Request to add a new caretaker	Admin	Add Caretaker	Added notification	Caretaker
Admin wants to remove Caretaker	Request to remove a Caretaker	Admin	Remove Caretaker	Removed Notification	Caretaker

Creational Design Patterns

Builder Pattern

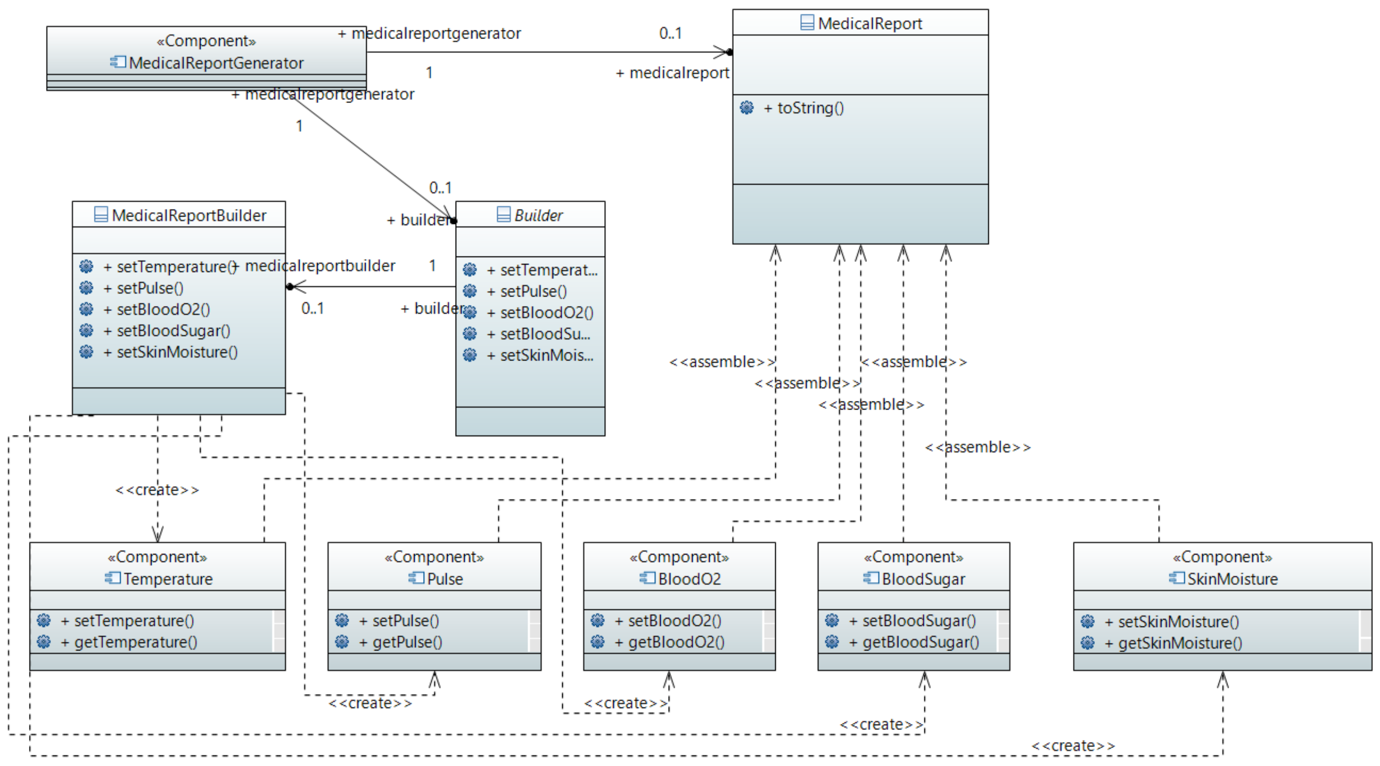
Problem

The construction of the Medical Report needs to be implemented with the medical data available. The medical report needs to be separated from the construction allowing the same building process to create various report representations.

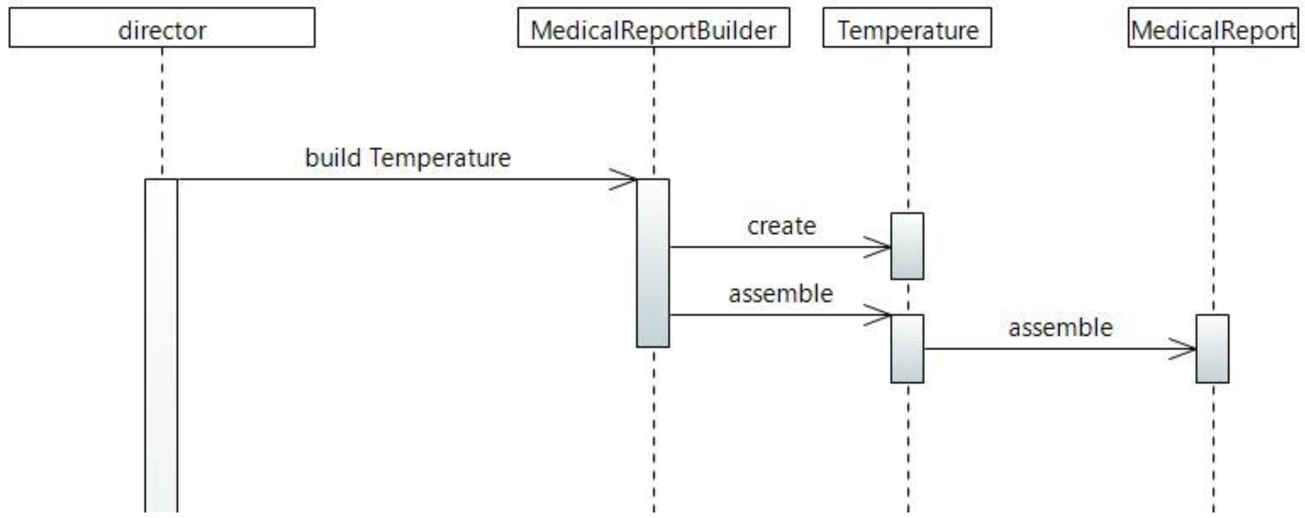
Solution

We could use the builder design pattern to solve this problem. Here, we could encapsulate creating and assembling the parts of a Medical Report in a separate Builder object.
This pattern would also provide control over steps in the building process of the medical report.

Class Diagram



System Sequence Diagram



Similar sequence diagram is observed for each component used to build the medical report.

Code

```
interface Builder {
    public MedicalReportBuilder setTemperature(String temperature);

    public MedicalReportBuilder setPulse(String pulse);

    public MedicalReportBuilder setBloodO2(String bloodO2);

    public MedicalReportBuilder setBloodSugar(String bloodSugar);

    public MedicalReportBuilder setSkinMoisture(String skinMoisture);
}

class MedicalReportBuilder {
    TemperatureComp temperature;
    PulseComp pulse;
    BloodO2Comp bloodO2;
    BloodSugarComp bloodSugar;
    SkinMoistureComp skinMoisture;

    public MedicalReportBuilder setTemperature(TemperatureComp temperature) {
        this.temperature = temperature;
        return this;
    }

    public MedicalReportBuilder setPulse(PulseComp pulse) {
        this.pulse = pulse;
        return this;
    }
    public MedicalReportBuilder setBloodO2(BloodO2Comp bloodO2) {
        this.bloodO2 = bloodO2;
        return this;
    }

    public MedicalReportBuilder setBloodSugar(BloodSugarComp bloodSugar) {
        this.bloodSugar = bloodSugar;
        return this;
    }

    public MedicalReportBuilder setSkinMoisture(SkinMoistureComp skinMoisture) {
        this.skinMoisture = skinMoisture;
        return this;
    }

    public MedicalReport getMedicalReport() {
        return new MedicalReport(temperature, pulse, bloodO2, bloodSugar, skinMoisture);
    }
}

class MedicalReport {
    TemperatureComp temperature;
    PulseComp pulse;
    BloodO2Comp bloodO2;
    BloodSugarComp bloodSugar;
    SkinMoistureComp skinMoisture;

    public MedicalReport(TemperatureComp temperature, PulseComp pulse, BloodO2Comp
bloodO2, BloodSugarComp bloodSugar,
            SkinMoistureComp skinMoisture) {
        this.temperature = temperature;
        this.pulse = pulse;
        this.bloodO2 = bloodO2;
        this.bloodSugar = bloodSugar;
        this.skinMoisture = skinMoisture;
    }

    @Override
    public String toString() {
        return "Temperature: " + temperature + "\n" + "Pulse Rate: " + pulse + "\n" +
"Blood O2 Level: " + bloodO2
                + "\n" + "Blood Sugar Level: " + bloodSugar + "\n" + "Skin Moisture" +
skinMoisture + "\n";
    }

}

class TemperatureComp {
    private String temp;
    public TemperatureComp(String temp) {
        setTemp(temp);
    }

    public String getTemp() {
        return temp;
    }

    public void setTemp(String temp) {
        this.temp = temp;
    }

    @Override
    public String toString() {
        return temp;
    }
}

class PulseComp {
    private String pulse;

    public PulseComp(String pulse) {
        setPulse(pulse);
    }

    public String getPulse() {
        return pulse;
    }

    public void setPulse(String pulse) {
        this.pulse = pulse;
    }

    @Override
    public String toString() {
        return pulse;
    }
}

class BloodO2Comp {
    private String O2;

    public BloodO2Comp(String O2) {
        setO2(O2);
    }

    public String getO2() {
        return O2;
```

```java
    }

    public void setO2(String o2) {
        this.O2 = o2;
    }

    @Override
    public String toString() {
        return O2;
    }
}

class BloodSugarComp {
    private String sugar;

    public BloodSugarComp(String sugar) {
        setSugar(sugar);
    }

    public String getSugar() {
        return sugar;
    }

    public void setSugar(String sugar) {
        this.sugar = sugar;
    }

    @Override
    public String toString() {
        return sugar;
    }
}

class SkinMoistureComp {
    private String moisture;

    public SkinMoistureComp(String mositure) {
        setMoisture(moisture);
    }

    public String getMoisture() {
        return moisture;
    }

    public void setMoisture(String moisture) {
        this.moisture = moisture;
    }

    @Override
    public String toString() {
        return moisture;
    }
}

class MedicalReportGenerator {
    public static void main(String[] args) {
        MedicalReportBuilder mrb = new MedicalReportBuilder();
        MedicalReport mr = mrb.setTemperature(new TemperatureComp("36.9 Degree
Celcius")).getMedicalReport();
        MedicalReport mr2 = mrb.setBloodSugar(new BloodSugarComp("7.8
mmol/L")).getMedicalReport();
        System.out.println(mr);
        System.out.println(mr2);

    }
}
```

Structural Design Patterns

Composite Pattern

Description

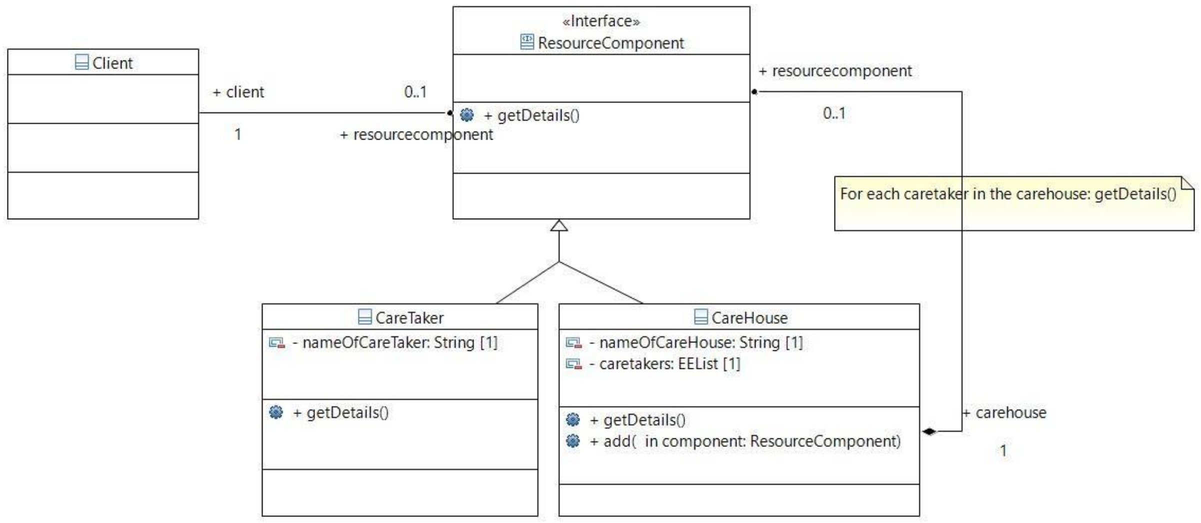
In the smart wheelchair system we observe the CareHouse and CareTakers to be important components of the system
A carehouse consists of number of caretakers and we observe that the a need to merge carehouses to a single one could occur
In this situation a need to treat individual objects(CareTaker objects) 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);

    }

}

class CareHouse implements ResourceComponent { // Composite Collection

    private String nameOfCareHouse;

    CareHouse(String nameOfCareHouse) {
        this.nameOfCareHouse = nameOfCareHouse;
```

```java
    }

    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 institute) {
        caretakers.add(institute);
    }

}

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

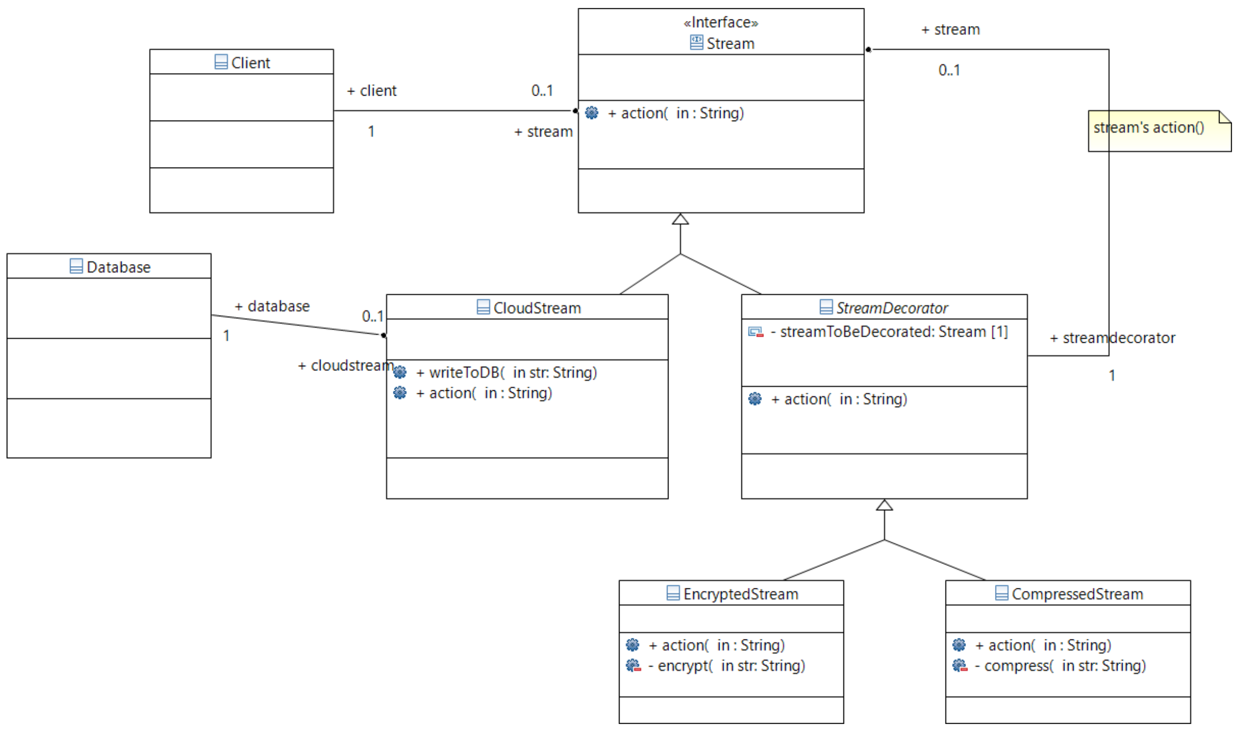
Description

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.

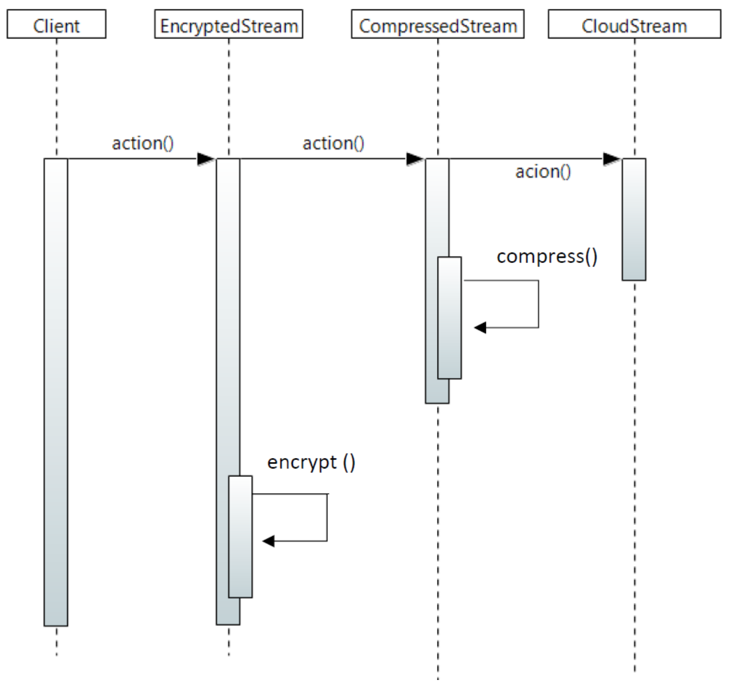
Problem and Solution

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

```java
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);
    }
    @Override
    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 = encrypt(str);
        super.action(str);

    }

    private String encrypt(String str) {
        return "Encrypted(" + str + ")";
    }
}
class CompressedStream extends StreamDecorator {

    CompressedStream(Stream streamToBeDecorated) {
        super(streamToBeDecorated);
    }

    public void action(String str) {
        str = compress(str);
        super.action(str);
    }

    private String compress(String str) {
        return "Compressed(" + 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

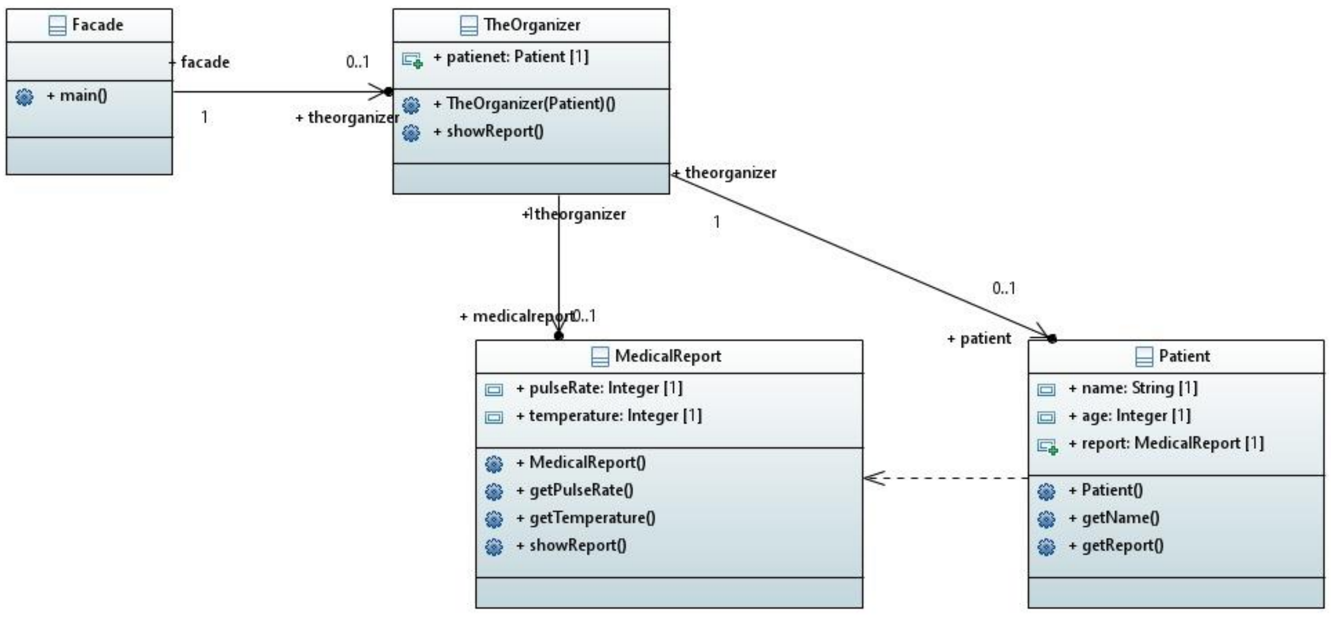
Problem

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.

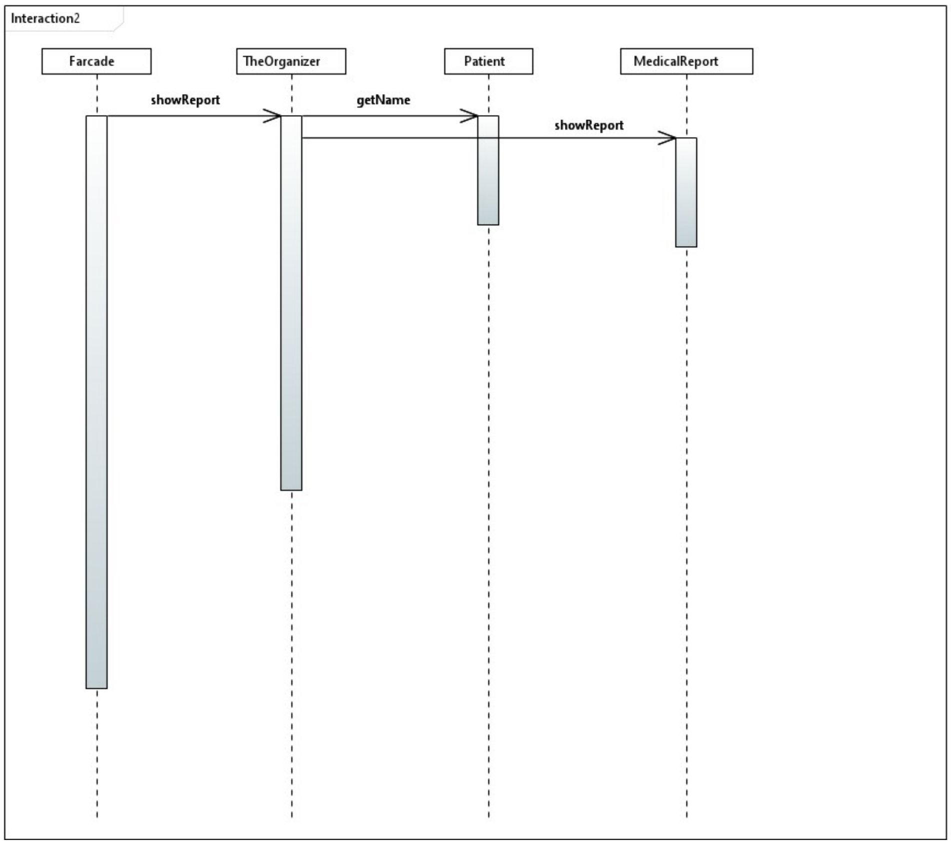
Solution

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



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;}
    }
```

```java
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();
}
}
```

Behavioral design Patterns

Chain Responsibility Pattern

Scenario (Problem)

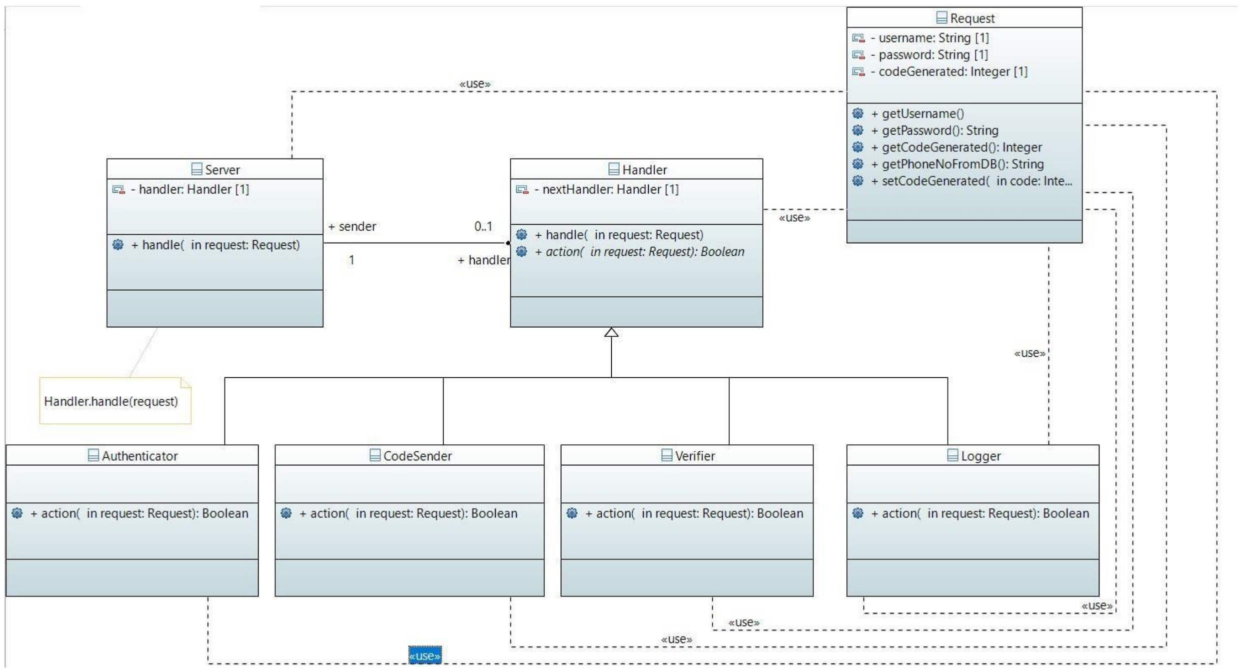
The request for verification is made upon attempt to login from a device. The authentication using phone number is needed to be an added protection. This added protection is only needed under certain instances like login attempt from a new device while logged in on another device, while it could be skipped under instances like a recently logged in device. In such an instance, we would need to handle the request step by step before the logging in happens. The order of the steps would differ, thus we need to have control of the order of the request handlers as well. There could be needs to add new protection features as well in the future , so the code must be extensible for new protection features.

Solution

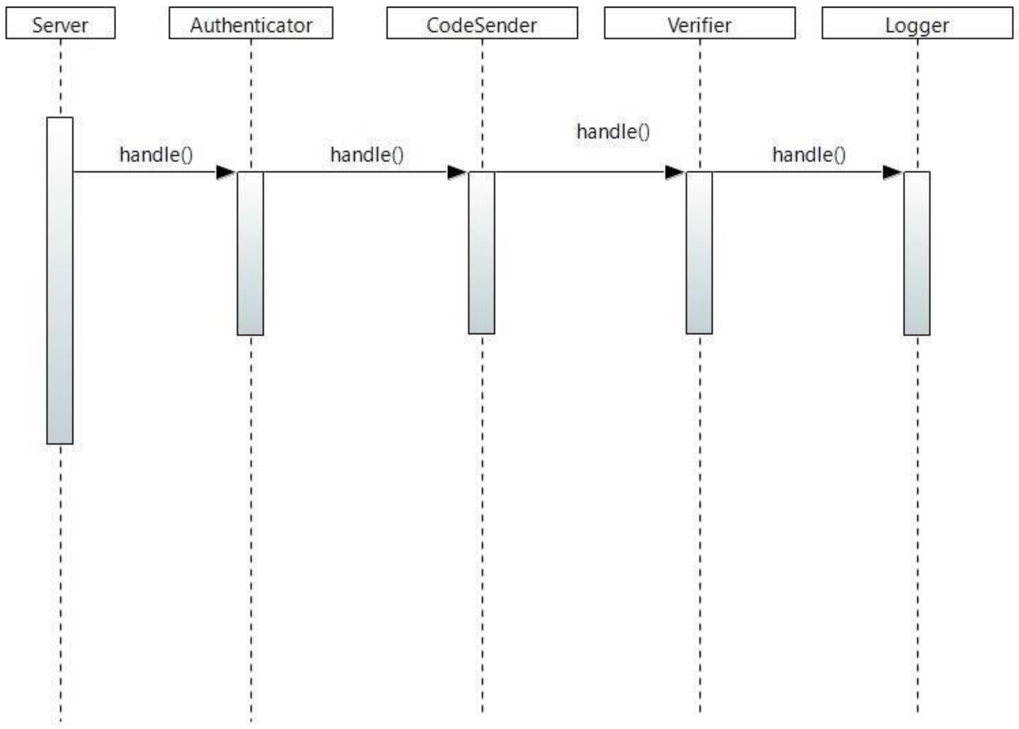
The chain responsibility pattern is used to achieve loose coupling in software design. This pattern provides the possibility of easily changing the order of request handling steps without altering the codes of the concrete classes. That is, we could omit the additional code to phone number protection feature when required without altering the Authenticator class in the code below. Also, a new feature like captcha verification or secret question verification could also be added without altering any other part of the code. We would only need to add a class inheriting the Handler abstract class and coding the action to be performed within the override action method in it.

Here, the server acts as the sender while the authenticator, code sender, verifier and Logger are the receivers.

Class Diagram


Request
- username: String [1]
- password: String [1]
- codeGenerated: Integer [1]
+ getUsername()
+ getPassword(): String
+ getCodeGenerated(): Integer
+ getPhoneNoFromDB(): String
+ setCodeGenerated(in code: Inte...
Server
- handler: Handler [1]
+ handle(in request: Request)
Handler
- nextHandler: Handler [1]
+ handle(in request: Request)
+ action(in request: Request): Boolean
+ sender
0..1
1
+ handler
«use»
Handler.handle(request)
Authenticator
CodeSender
Verifier
Logger
+ action(in request: Request): Boolean

Sequence Diagram



Code

```java
import java.util.Scanner;
import java.lang.System;
class Request{
    // A database query is a request for data from a database.
    private String username,password;
    private int codeGenerated;
    Request(String username,String password){
        this.username = username;
        this.password = password;
    }
    public String getUsername() {
        return this.username;
    }
    public String getPassword() {
        return this.password;
    }
    public String getPhoneNoFromDB() {
        //  get from database
        return "1234567890";
    }
    public int getCodeGenerated() {
        return this.codeGenerated;
    }
    public void setCodeGenerated(int codeGenerated) {
        this.codeGenerated = codeGenerated;
    }

}
abstract class Handler{
    private Handler nextHandler;
    Handler(Handler nextHandler){
        this.nextHandler = nextHandler;
    }
    void handle(Request request){
        if (action(request) && nextHandler!=null){
            nextHandler.handle(request);
        }
        System.out.println("Done");
        System.exit(0);

    }
    abstract boolean action(Request request);
}
class Server {
    private Handler handler;
    Server(Handler handler){
        this.handler = handler;
    }
    void handle(Request request){
        handler.handle(request);
    }
}
class Authenticator extends Handler{
    Authenticator(Handler nextHandler){
       super(nextHandler);
    }
    @Override
    boolean action(Request request) {
        boolean isValid = false;
        if (request.getUsername().equals("admin") &&
request.getPassword().equals("password")){
            System.out.println("Authenticated");
            isValid = true;
        }
        System.out.println("VALID : "+isValid);
        return isValid; // if valid go to next step
    }

}

class CodeSender extends Handler{

    CodeSender(Handler nextHandler) {
        super(nextHandler);
    }

    @Override
    boolean action(Request request) {
        int codeGenerated = 111222; //can use random to generate
        System.out.println("Code Generated = 111 222");

        //  Assume always successful
        boolean success = true;
        request.setCodeGenerated(codeGenerated);
        System.out.println("Sending code to "+request.getPhoneNoFromDB()+" : "+success);
        return success;
    }


}


class Verifier extends Handler{
    Verifier(Handler nextHandler) {
        super(nextHandler);
    }

    @Override
    boolean action(Request request) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter pin code to verify: ");
        int code = sc.nextInt();
        System.out.println("Verified Code");
        sc.close();
        return (code == request.getCodeGenerated());
    }


}

class Logger extends Handler{

    Logger(Handler nextHandler) {
        super(nextHandler);
    }
    @Override
    boolean action(Request request) {
        System.out.println("Logging In");
        return true;
    }

}
class Scenario1{
    void test(){
        Logger log = new Logger(null);
        Verifier verify = new Verifier(log);
        CodeSender codeSender = new CodeSender(verify);
        Authenticator auth = new Authenticator(codeSender);
        Request request =  new Request("admin", "password");
        Server server = new Server(auth);
        server.handle(request);
    }
}

public class chainofResponsibilty {
    public static void main(String[] args) {
        Scenario1 scene1 = new Scenario1();
        scene1.test();
    }
}
```

Observer Pattern

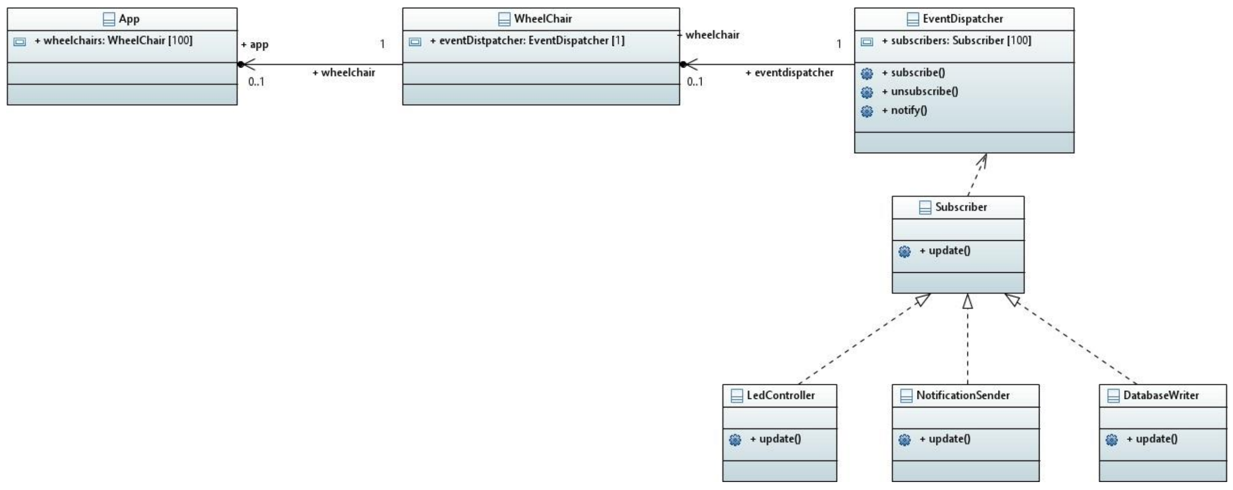
Scenario (Problem)

While the wheelchair is operating, some activities, alerts and notification has to be issued in order to keep the care takers and the hospitals updated. to do so, the app has to either constantly check whether they are ready or not or the app has to send the updates whether they are required or not. but in both cases, a lot of resources are being wasted. it is better if the app is notified when these are required or ready. So that, app could issue the relevent alerts and notifications when they are required or ready.the solution is a publisher subscriber model which is described by the observer design pattern.

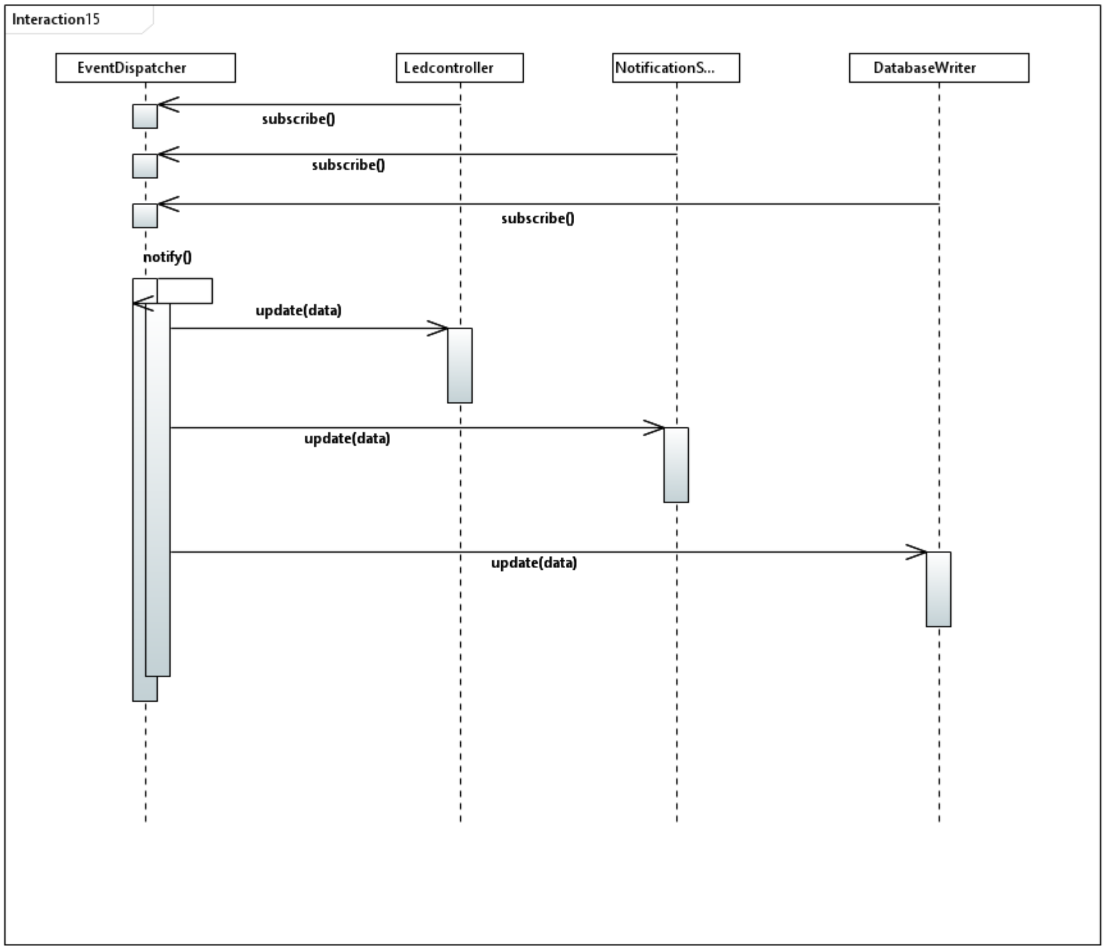
Solution

A publisher class (EventDispatcher in this case) is created and it will keep the Subscribers (Listners) lists for each event. this could as well be implemented with a hash map. when there is a change of state that triggers the event the publisher will notify all teh Listners via "notify" method that is declared in the Listener interface which is implemented by all the concrete Listeners. Adding a new Listner will not cause of changing the code of the app or the publisher since all the listeners are connected via a common interface.

Class Diagram



Sequence Diagram



Code

```
import java.util.*;

enum Event{EMERGENCY, ROUTINE, LOCATION_CHANGED, BATTERY_LOW}

class EventDispatcher{     // this is the publisher
   private Map<Event,List<Listener>> listeners = new HashMap<>();

   public EventDispatcher(){
      listeners.put(Event.EMERGENCY, new ArrayList<>());
      listeners.put(Event.ROUTINE, new ArrayList<>());
      listeners.put(Event.LOCATION_CHANGED, new ArrayList<>());
      listeners.put(Event.BATTERY_LOW, new ArrayList<>());
   }

   public void subscribe(Listener listener,  Event event){
      listeners.get(event).add(listener);
   }
```

```java
   public void unsubscribe(Listener listener, Event event){
      listeners.get(event).remove(listener);
   }

   public void notify(Event event, String data){
      List<Listener> event_listeners = listeners.get(event);
      for(Listener listener: event_listeners){
         listener.update(data);
      }
   }
}

class WheelChair{
   public EventDispatcher eventDispatcher;

   public WheelChair(EventDispatcher eventDispatcher){
      this.eventDispatcher = eventDispatcher;
   }
}

interface Listener{
   public void update(String data);
}

class LedController implements Listener{
   private boolean ledState;

   public void update(String data){
      // the data will not be used in this function
      ledState = true;
      System.out.println("Warning LED turned on");
   }
}

class NotificationSender implements Listener{
   String notification;

   public void update(String data){
      // invoke the functions to send the notification
      // data that has passed will be used in the process.
      System.out.println("Notification sent");
   }
}

class DataBaseWriter implements Listener{
   boolean state;

   public void update(String data){
      //call the necessary functions to write to a database
      // data that has passed will be used in the process.
      System.out.println("Writing to database");
   }
}

public class ObserverTestApp{
   public static void main(String[] args) {

      EventDispatcher eventDispatcher = new EventDispatcher();
      WheelChair wheelChair = new WheelChair(eventDispatcher);

      LedController ledController = new LedController();
      NotificationSender notificationSender = new NotificationSender();
      DataBaseWriter dataBaseWriter = new DataBaseWriter();

      eventDispatcher.subscribe(ledController, Event.EMERGENCY);
      eventDispatcher.subscribe(ledController, Event.BATTERY_LOW);
      eventDispatcher.subscribe(notificationSender, Event.EMERGENCY);
      eventDispatcher.subscribe(notificationSender, Event.BATTERY_LOW);
      eventDispatcher.subscribe(notificationSender, Event.ROUTINE);
      eventDispatcher.subscribe(dataBaseWriter, Event.EMERGENCY);
      eventDispatcher.subscribe(dataBaseWriter, Event.ROUTINE);
      eventDispatcher.subscribe(dataBaseWriter, Event.BATTERY_LOW);
      eventDispatcher.subscribe(dataBaseWriter, Event.LOCATION_CHANGED);

      eventDispatcher.notify(Event.BATTERY_LOW, "data");
   }
}
```

Iterator Pattern

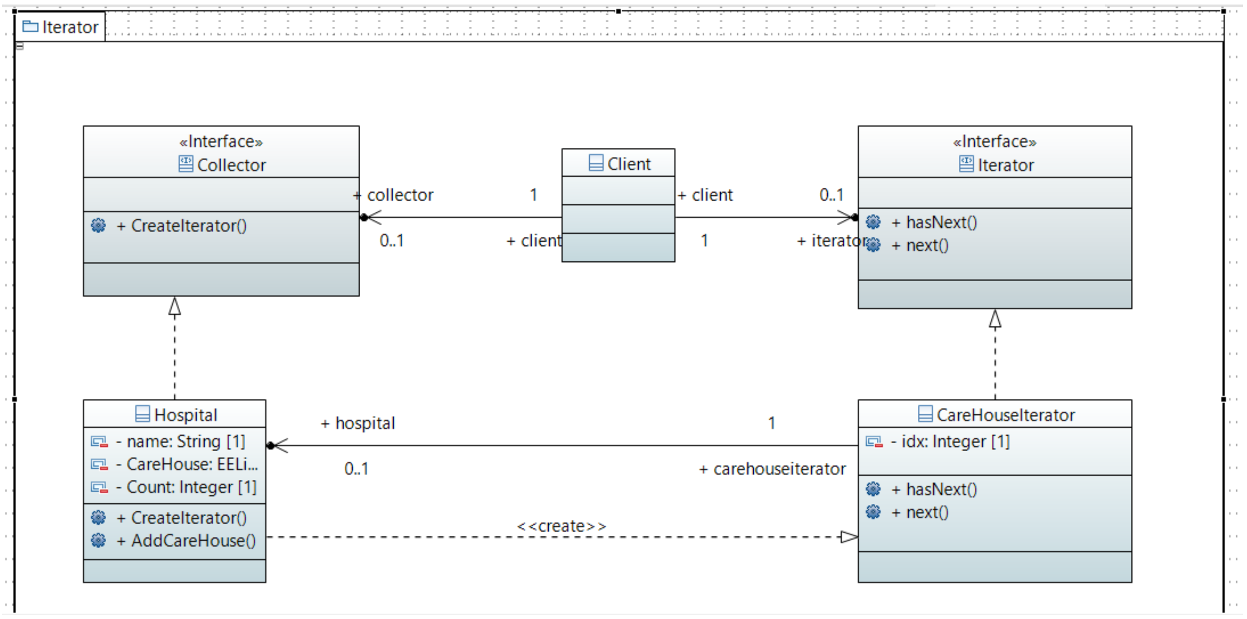
Scenario (Problem)

There occurs a need to have a collection of carehouses attached with the hospital and each carehouse could also have a collection of patients responsible for. Sometimes, we would need to change the underlying structure that holds carehouse objects in a carehouse class and that holds Patient objects in a CareTaker class. In such an instance the general way of iteration would lead to changes in client any changes are made to the underlying storage structure. The underlying storage collection structure would also be exposed to client in such an instance.

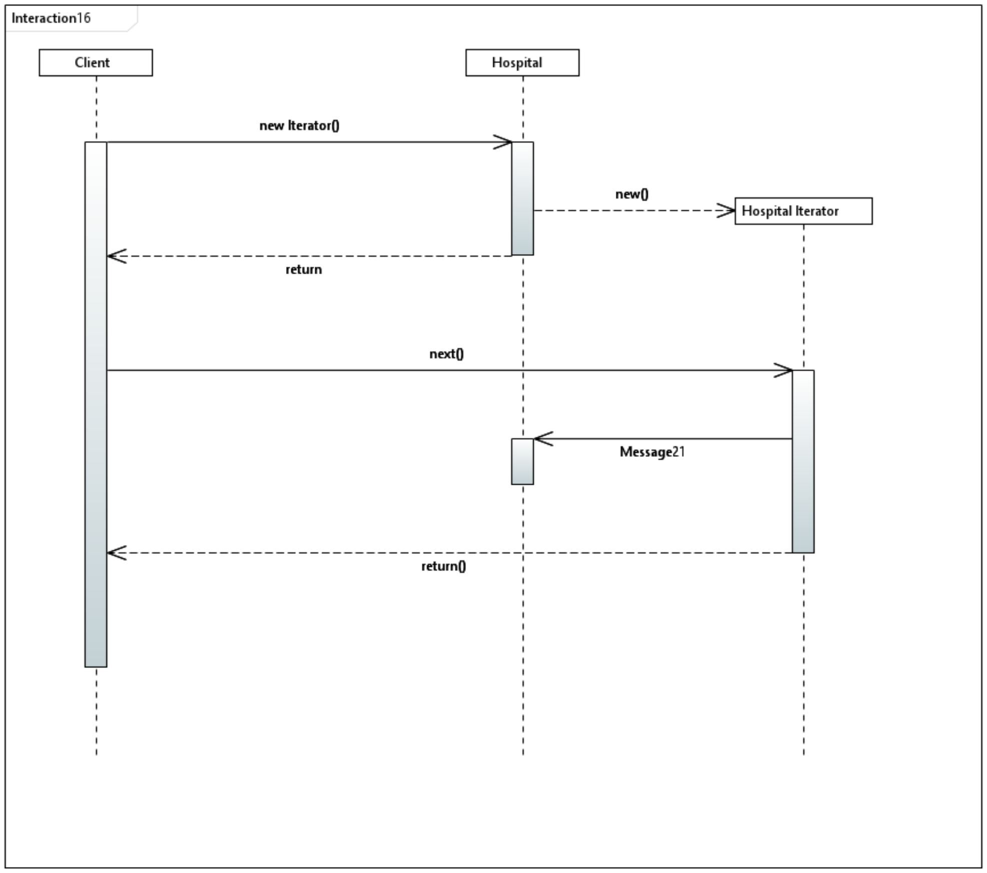
Solution

This could be solved using the iterator pattern. here we use a nested Iterator class within an aggregate class, so that the changes would need to be made only withing the aggregate class, but not in the client .This way, we could hide the storage structure and change the structure of it without making changes to the client.

Class Diagram



Sequence Diagram



Code

```java
import java.util.ArrayList;
import java.util.List;

interface Collector { // Aggregate interface
    Iterator createIterator();
}

class CareHouses {
    private static int i = 0;
    private int id;
    private String name;

    public CareHouses(String name) {
        i++;
        this.id = i;
        this.name = name;
    }
    @Override
    public String toString() {
        return this.getClass().getSimpleName()+" : id = "+id+" Name: "+name;
    }
    // more methods and attributes available

}

interface Iterator {
    boolean hasNext();
    Object next();

}

class Hospital implements Collector {
    private String name;
    private List<CareHouses> careHouses = new ArrayList<>();
    private int count = 0; // count of carehouses in above list

    public Hospital(String name) {
        this.name = name;
    }

    public class CareHouseIterator implements Iterator {
        private int idx;

        public CareHouseIterator() {
            this.idx = 0;
        }
        @Override
        public Object next() {
            return careHouses.get(idx++);

        }

        @Override
        public boolean hasNext() {
            if (idx < count) {
                return true;
            } else
                return false;
        }

    }


    @Override
    public Iterator createIterator() {
        return new CareHouseIterator();
    }

    public Hospital addCareHouse(CareHouses c) {
        careHouses.add(c);
        count++;
        return this;

    }

    // more methods and attributes available

}
// Now consider CareTaker objects within a Carehouse

public class IteratorTest {
    public static void main(String[] args) {
        Hospital H = new Hospital("Hospital_01");
        H.addCareHouse(new CareHouses("CareHouse_01"));
        H.addCareHouse(new CareHouses("CareHouse_02"));
        H.addCareHouse(new CareHouses("CareHouse_03"));
        H.addCareHouse(new CareHouses("CareHouse_04"));
        H.addCareHouse(new CareHouses("CareHouse_05"));
        Iterator iter = H.createIterator();
        while (iter.hasNext())
            System.out.println(iter.next());
    }
}
```

Command Pattern

Scenario (Problem)

This could be solved using the iterator pattern. here we use a nested Iterator class within an aggregate class, so that the changes would need to be made only withing the aggregate class, but not in the client .This way, we could hide the storage structure and change the structure of it without making changes to the client. The voice recognition system that commands the motion of the wheel chair would need to be implemented.

Here, it would be needed to perform different operations based on different voice commands, thus a need to decide which command to perform needs to be determined at real time.

The commands that could operate the wheel chair could be :

1. Move Forward
2. Turn Around
3. Turn Right
4. Turn Left
5. Stop

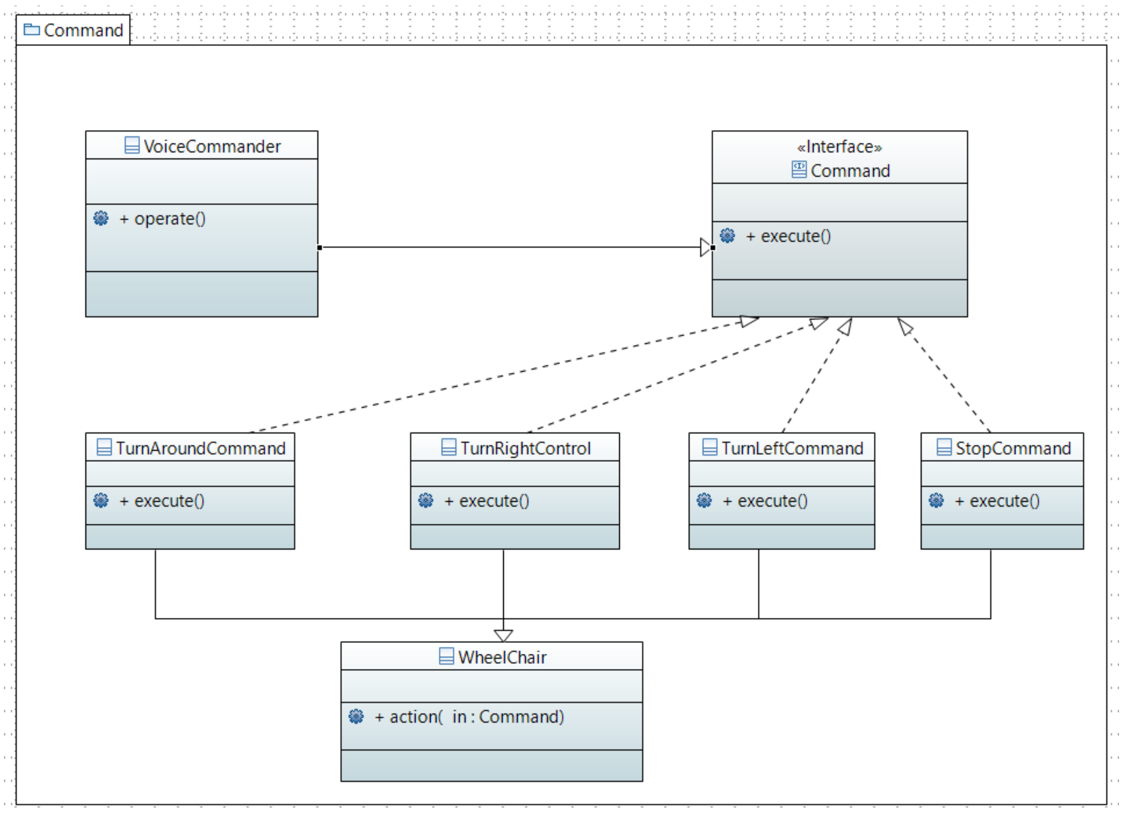
Solution

The command design pattern could be used to solve this problem.

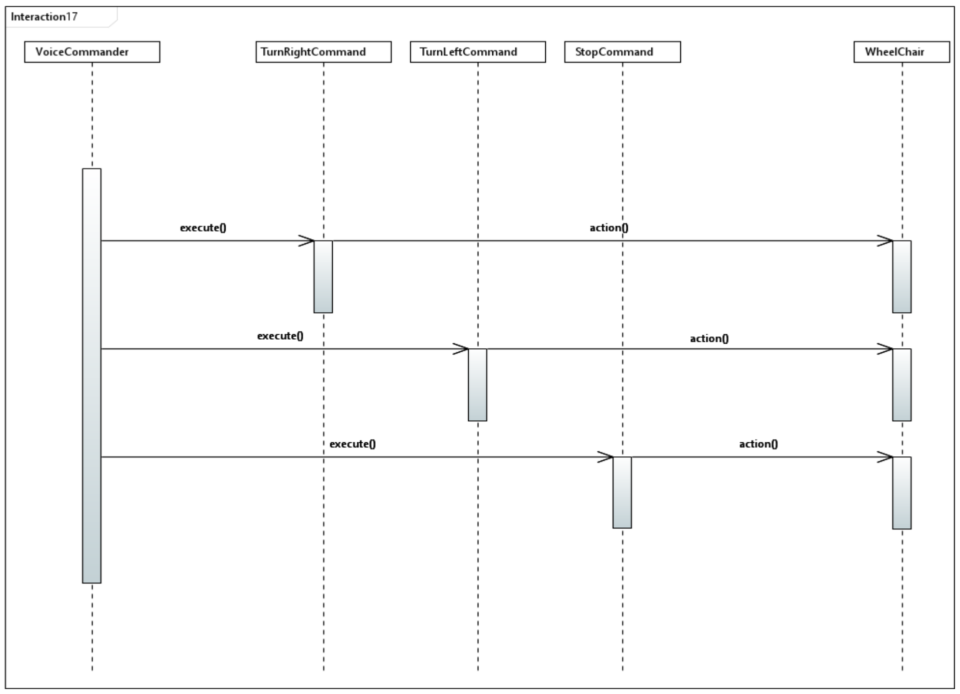
The implementation of a command interface which could be implemented in each separated concrete command classes , so that the Invoker(Command Executer) is not directly involved with the reciever(). Here, we need not keep track of changes , but if a need appears we could keep track of the movement of the wheel chair (to determine the movement done based on voice commands) , so that a real time movement of the wheel chair could be watched for selected patients.

We do not undo commands in this system though that is possible feature implemented efficiently using a command pattern.

Class Diagram



Sequence Diagram



Code

```java
//Command
interface Command{
    void execute();
}

//Concrete Commands
class MoveForwardCommand implements Command{
    private WheelChair chair;
    public MoveForwardCommand(WheelChair chair) {
        this.chair = chair;
    }
    @Override
    public void execute() {
        System.out.println(".................................");
        System.out.println("The System is moving forwards");
        chair.action(this);

    }

}
class TurnAroundCommand implements Command{
    private WheelChair chair;
    public TurnAroundCommand(WheelChair chair) {
        this.chair = chair;
    }
    @Override
    public void execute() {
        //  This could be implemented using 2 turn right/left commands at a slow pace.
        System.out.println(".................................");
        System.out.println("The System is turned around");
        chair.action(this);

    }

}
class TurnRightCommand implements Command{
    private WheelChair chair;
    public TurnRightCommand(WheelChair chair) {
        this.chair = chair;
    }
    @Override
    public void execute() {
        System.out.println(".................................");
        System.out.println("The System is turned right");
        chair.action(this);
    }

}
class TurnLeftCommand implements Command{
    private WheelChair chair;
    public TurnLeftCommand(WheelChair chair) {
        this.chair = chair;
    }
    @Override
    public void execute() {
        System.out.println("................................");
        System.out.println("The System is turned left");
        chair.action(this);
    }

}

class StopCommand implements Command{
    private WheelChair chair;
    public StopCommand(WheelChair chair) {
        this.chair = chair;
    }
    @Override
    public void execute() {
        System.out.println("................................");
        System.out.println("The System is stopped");
        chair.action(this);

    }

}
// Invoker
class VoiceCommander{
    private WheelChair chair;
    public VoiceCommander(WheelChair chair) {
        this.chair = chair;
    }
    VoiceCommander(){}
    public void operate(Command command){
        command.execute();
    }
}

//Reciever
class WheelChair{
    private int id;
    WheelChair(int id ){
        this.id = id;
    }
    public int getId() {
        return id;
    }
    public void action(Command command){
        System.out.println(command.getClass().getSimpleName()+"'s action taking place in wheelchair");
        System.out.println(".................................");
    }
}

public class CommandTest{
    public static void main(String[] args) {
        WheelChair wc  =  new  WheelChair(1);
        VoiceCommander vc = new VoiceCommander(wc);

        MoveForwardCommand mvfCommand = new MoveForwardCommand(wc);
        vc.operate(mvfCommand);
        TurnAroundCommand taCommand = new TurnAroundCommand(wc);
        vc.operate(taCommand);
        TurnRightCommand trCommand = new TurnRightCommand(wc);
        vc.operate(trCommand);
        TurnLeftCommand tlCommand = new TurnLeftCommand(wc);
        vc.operate(tlCommand);

    }
}
```

State Pattern

Scenario (Problem)

We observe that the wheel chair could be moved at different surfaces, so we could change the force on wheel chair according to that to maintain the speed.

For e.g.:

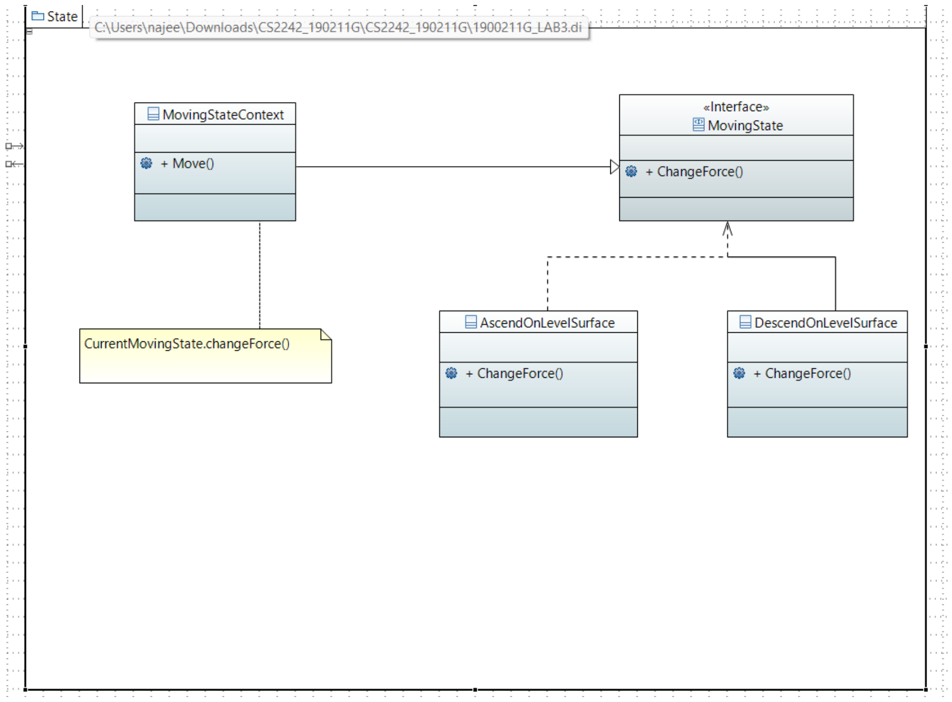
> If we are ascending , we would need to increase the force on wheel chair in the direction of movement to maintain a speed. If we are descending , we would need to increase the force on wheel chair opposite to the direction of movement to maintain a speed. If we are moving on a level surface then, a constant force that produces the required speed is maintained,

In this scenario, we observe that there could be more additional states like moving in descending or ascending a stair case feature added to the wheel chair. In such an addition of states , we need to easily add such feature/state without modifying much of the code. Thus, open-closed (open for extension and closed for modification) principle needs to be followed

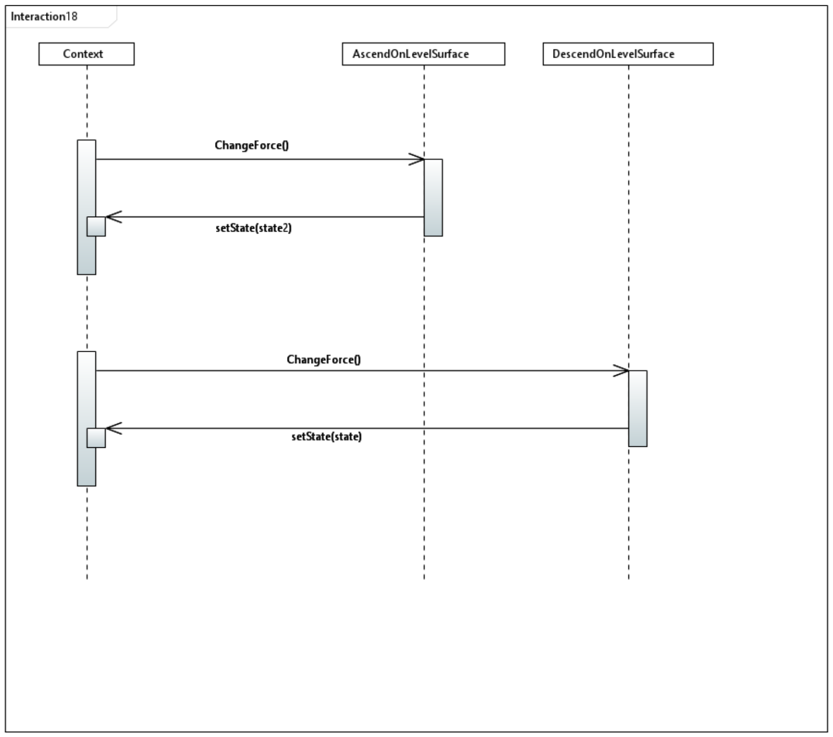
Solution

The state pattern defines an interface that specify a state-specific behavior and have a class that implement the specific interface. The state interface is implemented by the different states of the wheel chair object and state-specific behavior is implemented here within the class that wants to implement the state-specific behavior. In such case, any new state to be added could be done easily by creating a new class that implements the predefined interface and provide the state specific behavior within the class method.

Class Diagram



Sequence Diagram



Code

```
class WheelChair{
  // methods and attributes available

}
interface MovingState{
  void changeForce();
}
class MovingStateContext{
  private MovingState currentMovingState;
  private WheelChair context;
  MovingStateContext(WheelChair context){
    this.context = context;
  }
  public void setCurrentMovingState(MovingState currentMovingState) {
    this.currentMovingState = currentMovingState;
  }
  void move(){
    if (currentMovingState!=null){
      currentMovingState.changeForce();
    }else{
      System.out.println("No change of force required!");
    }
  }
```

```java
  public WheelChair getContext() {
      return this.context;
  }


}
class AscendOnLevelSurface implements MovingState{

  @Override
  public void changeForce() {
    //angles are detected and force needed are calculated prior
    System.out.println("Force increased as per angle of elevation");

  }

}
class DescendOnLevelSurface implements MovingState{

  @Override
  public void changeForce() {
    System.out.println("Force decreased as per angle of depression");

  }

}
public class StatePatternTest{
  public static void main(String[] args) {
    WheelChair chair = new WheelChair();// has arguments
    MovingStateContext context = new MovingStateContext(chair);

    context.setCurrentMovingState(new AscendOnLevelSurface());
    context.move();

    context.setCurrentMovingState(new DescendOnLevelSurface());
    context.move();


  }
}
```

Strategy Pattern

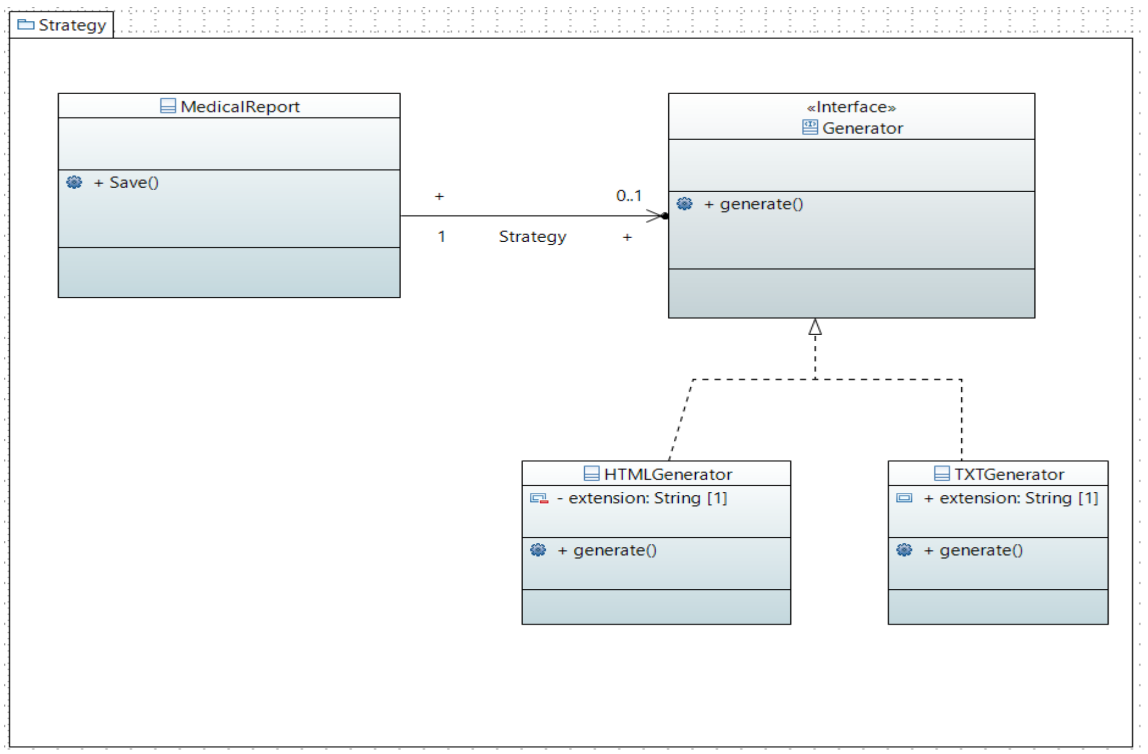
Scenario (Problem)

The instance where we generate a medical report could be generated in different formats like : HTML ,Text files. Here, we could generate each format based on selection within a class itself, but this would not abide the open-close principle. The software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification. When a new format needs to be generated, we would need to modify the code and make changes.

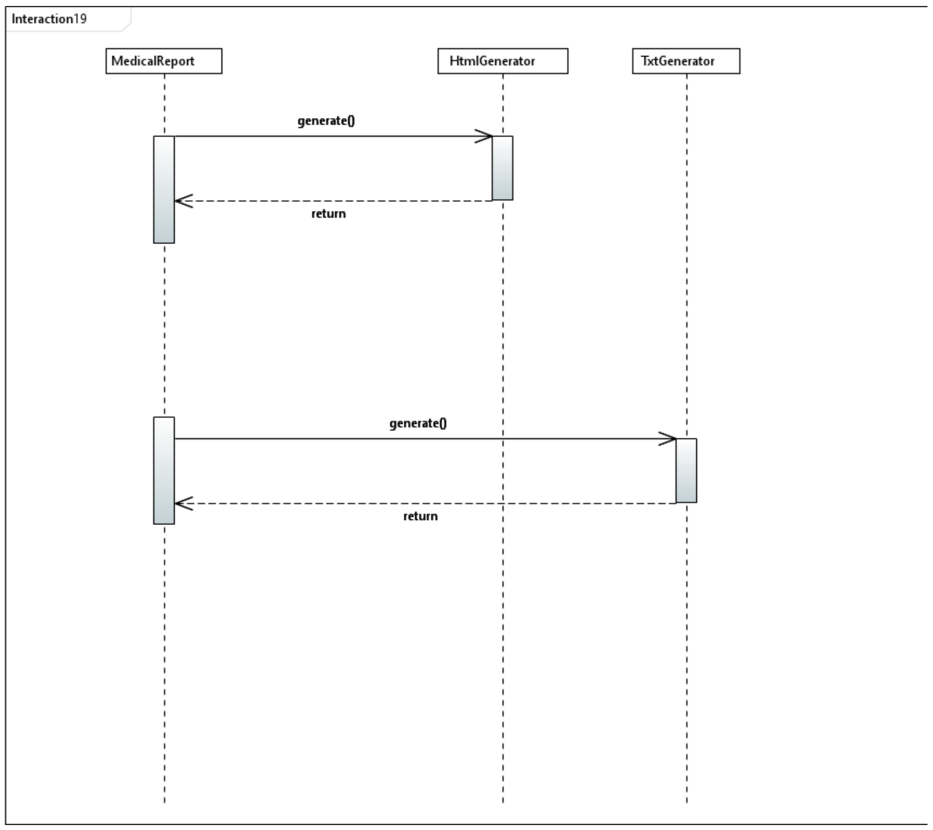
Solution

The strategy pattern uses composition instead of inheritance with different behaviors defined as separate interfaces with specific classes implementing these interfaces. Thus, the changes could be made to the respective concrete strategy classes (HTMLGenerator, TXTGenerator), but not the context class (ReportSaver) and new file format generators could be added easily by creating a new class implementing the interface.

Class Diagram



Sequence Diagram



Code

```
class MedicalReportBuilder{
    String temperature,pulse,bloodO2,bloodSugar,skinMoisture;

    public MedicalReportBuilder setTemperature(String temperature) {
        this.temperature = temperature;
        return this;
    }
    public MedicalReportBuilder setPulse(String pulse) {
        this.pulse = pulse;
        return this;
    }
    public MedicalReportBuilder setBloodO2(String bloodO2) {
        this.bloodO2 = bloodO2;
        return this;
    }
    public MedicalReportBuilder setBloodSugar(String bloodSugar) {
        this.bloodSugar = bloodSugar;
        return this;
    }
    public MedicalReportBuilder setSkinMoisture(String skinMoisture) {
        this.skinMoisture = skinMoisture;
        return this;
    }
    public MedicalReport getMedicalReport(){
        return new
MedicalReport(temperature,pulse,bloodO2,bloodSugar,skinMoisture);
    }
}

class MedicalReport{
    String temperature,pulse,bloodO2,bloodSugar,skinMoisture;
    public MedicalReport(String temperature, String pulse, String bloodO2, String
bloodSugar,String skinMoisture) {
        this.temperature = temperature;
        this.pulse = pulse;
        this.bloodO2=bloodO2;
        this.bloodSugar=bloodSugar;
        this.skinMoisture=skinMoisture;
    }
    @Override
```

```java
    public String toString() {
        return "Temperature: "+temperature+"\n"+"Pulse Rate: "+pulse+"\n"+"Blood
O2 Level: "+bloodO2+"\n"+"Blood Sugar Level: "+bloodSugar+"\n"+"Skin Moisture:
"+skinMoisture+"\n";
    }
    public void save(Generator generator,String fileName){
        String extension = generator.generate();
        System.out.println("File saved as : "+fileName+"."+ extension);
        System.out.println("\nDetails in saved file :\n"+this);
        System.out.println("..................................");

    }

}


interface Generator{ // Strategy
    public String generate();
}
class HTMLGenerator implements Generator{ // Concrete Strategy
    private String extension = "html";

    @Override
    public String generate() {
        System.out.println("Generating HTML File");
        return  extension;

    }

}
class TXTGenerator implements Generator{ // Concrete Strategy
    private String extension = "txt";

    @Override
    public String generate() {
        System.out.println("Generating TXT File");
        return  extension;


    }

}

public class StrategyOrPolicyPatternTest{
    public static void main(String[] args) {
        Generator html = new HTMLGenerator();

        MedicalReportBuilder mrb1 = new MedicalReportBuilder();
        mrb1 = mrb1.setTemperature("36.9 Degree Celcius");
        mrb1 = mrb1.setBloodSugar("7.8 mmol/L");
        MedicalReport mr1 = mrb1.getMedicalReport();
        mr1.save(html,"Report_01");

        Generator txt = new TXTGenerator();
        MedicalReportBuilder mrb2 = new MedicalReportBuilder();
        mrb2 = mrb2.setTemperature("37.4 Degree Celcius");
        MedicalReport mr2 = mrb2.getMedicalReport();
        mr2.save(txt,"Report_02");

    }
}
```

Conclusion

Non-functional requirements

Usability

Usability is the ability of a product to be used easily.
To achieve this we have provided the caretakers with a desktop application and a mobile application. So that they can interact with the device without knowing much of the details about the device. In these applications, we always used global symbols on every occasion. For example, we have used a plus mark for create-buttons since it is globally recognized as a symbol for creation. We have also taken care of the usability issues that come with the performances as described below.
We have used QT creator to develop desktop and mobile applications. Since it is a cross-platform, both desktop and mobile applications have the same interface. Therefore it is easy to switch between desktop and mobile applications.

Reliability

Reliability is referred to describe the ability to output the same functionality even after extensive use.
With numerous tests, we have calculated and minimized the probability of failure of the software. We have included a fail-safe design for the wheelchair as well. The wheelchair issues warnings for probable failures and it stores data in a local buffer before sending it. If something happens to the data that has been sent, the data in the local buffer can be used.
We also accept customer feedbacks after issuing the wheelchairs, hence we can work on the bugs that might not have been detected in the testing phase. With this feedback, we can give more reliable updates to the software.

Performance and speed

Performance is the term used to indicate how fast a software responds to the action of a user.
We have achieved fast synchronization via a common cloud-based database. With this, every device can gather data through the database without directly contacting any particular device or any other party. Software components are highly independent due to the high decoupling in the software design. Therefore a failure in one component does not affect the performance of another component. The system uses Wi-Fi to update the database with details from the wheelchair and is expected to provide instant and real-time information for the caretakers and other authorities. The security of the wheelchair user is of utmost importance and the sensors of a considerable obstacle detection range would serve this purpose.

Safety and security

Security is referred to describe the ability to
The databases will be secure and access to the system would only be provided to respective individuals. The authority to terminate a user of the system would be provided to a higher authority.
There will be an accounts system for the above task. Also, the passwords for the accounts will be validated in the account creation. So that the passwords will not be easy to guess. To provide more security in a case of a forgotten password, the system will have the capability to ask and store some security questions.
Also, the local stores do not have personal information or anything except the medical data.

There will not be any interface open to the outside as well. So that no one can read even the medical data.

Compatibility

Compatibility is the ability to work with many levels of hardware capabilities. The lesser capabilities a software uses, the better the compatibility of that software.
Since the apps are using storage as a cloud service

Summary of contribution

- M.H. AKEEL AHMED – 190028C

I created the use case diagrams and the class diagrams to identify functions and how the roles interact with them. Using the use case diagrams, I was able to highlight the roles that interact with the system and the functionality provided by the system without going deep into the inner workings of the system. I was able to provide an overview of the system using the class diagram so that the system's structure could be illustrated in a detailed way showing the attributes, operations as well as inter relationships. Also, there were instances where we needed to build the system in an effective way so that the system is closed for modification while easily extended. I focused on the instance of Medical Report creation where it was needed to create the report as a complex object consisting of various details. The construction of the Medical Report needed to be implemented with the medical data available. The medical report needed to be separated from the construction allowing the same building process to create various report representations. So, I used the builder design pattern to accomplish this need. Also, composite design pattern was identified to be of much effective to solve the problem of merging the care houses as a later need. This would automatically update the details within each care house to be merged into the newly generated care house object. The logging was observed to be done using the chain of responsibility done. Request needs to be handled by the authenticator, code sender, verifier and logger and needs to be passed within those handlers based on the result of the previous handler. The voice recognition is the preferred choice of control of the automation of the wheelchair. So, predefined commands specified by the system developer could be used by the wheelchair user to do movement operations. Using the command pattern here, we would be able to add new commands to the voice control motion system easily as an extension of the available system.

- I.T.A. ILESINGHE – 190238U

I created the state diagram and system sequence for the system. I created the state diagrams to visualize the entire life cycle of objects and thus help to provide a better understanding of state-based systems. The sequence diagram was designed to show the interaction logic between the objects in the system in the time order that the interactions take place. I used the façade pattern to decouple the source code from the interface and make the code base more arranged while making the code base easier to handle. Thus, I was able to hide the complexity of the system and provide the users with a simpler interface. I also observed that the observer pattern is an efficient way to design the part where alerts and notifications are issued to keep the components of the system updated. The constantly sending approach of relevant alerts and notifications to the components would be a less efficient way as resources would be in continuous use. So that, app could issue the relevant alerts and notifications when they are required or ready. I used the publisher subscriber model which is described by the observer design pattern to achieve this purpose. The wheel chair has different states of motion, thus I used the State pattern to accomplish the state specific behavior so that new states could be added easily by creating new classes that that implements the predefined interface and provide the state specific behavior within the class method.

- N.M. HAFEEL – 190211G

I created activity diagrams and event table for this system to show the software processes as a progression of actions and to collect the events that occur in this system and illustrate an overview of what happens in those events. 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 compression feature would also be needed. I used the decorator pattern to efficiently decorate data before the major operation of writing it to the database. Also, it was needed to have a collection of care houses within a hospital. Upon displaying in applications , we would need to iterate through the collection to display details. Thus, the iteration needs to be made possible irrespective of the underlying structure. This was a possible instance for the iterator design pattern to be used so that the underlying storage collection structure would not be exposed to client and it could be changed easily adding a different suitable iterator method . The generation of the Medical report needs to be generated in multiple formats. The strategy pattern uses composition instead of inheritance with different behaviors defined as separate interfaces with specific classes implementing these interfaces. So, I used the strategic pattern to define the separate behaviors of each different generator so that it could be easily extended with a new generator type.