LAB 8

BEHAVIORAL DESIGN PATTERNS IN SMART WHEEL CHAIR SYSTEM

Prepared by :

Akeel Ahmed - 190028C

Thilina Ilesinghe - 190238U

Hafeel - 190211G

Contents

Chain Responsibility Pattern

Scenario (Problem)

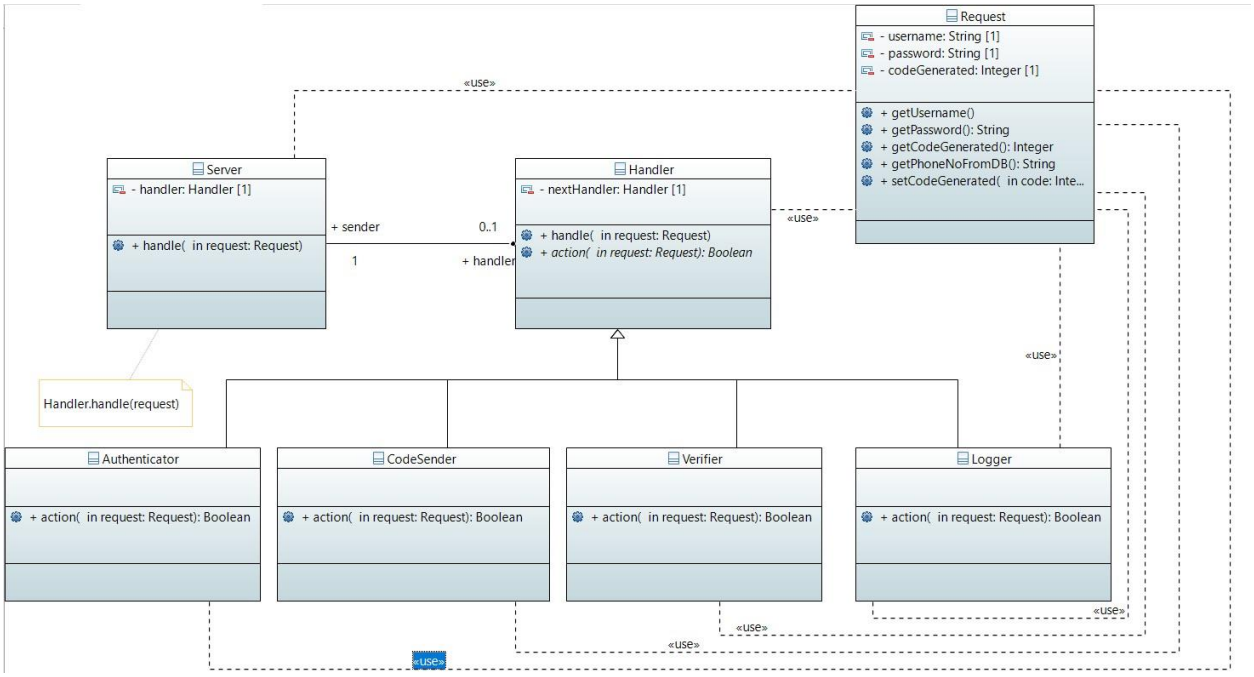
The request for verification is made upon attempt to login from a device. The authentication using phone number is needed to be an added protection. This added protection is only needed under certain instances like login attempt from a new device while logged in on another device, while it could be skipped 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

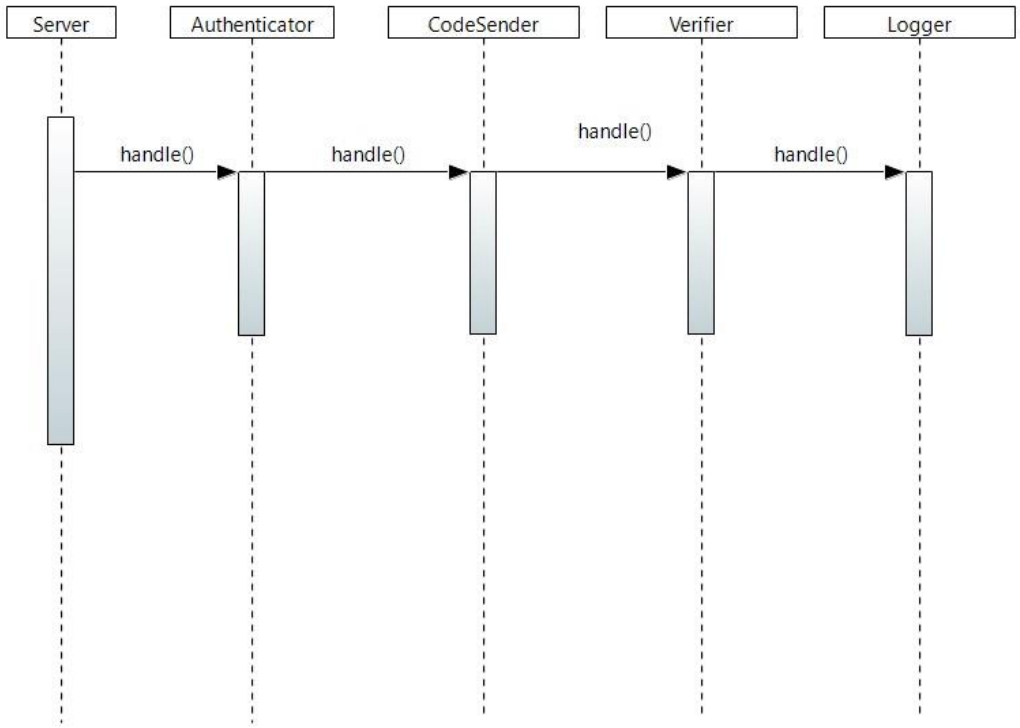
The chain responsibility pattern is used to achieve loose coupling in software design. This pattern provides the possibility of easily changing the order of request handling steps without altering the codes of the concrete classes. That is, we could omit the additional code to phone number protection 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



Sequence Diagram



Code

```
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

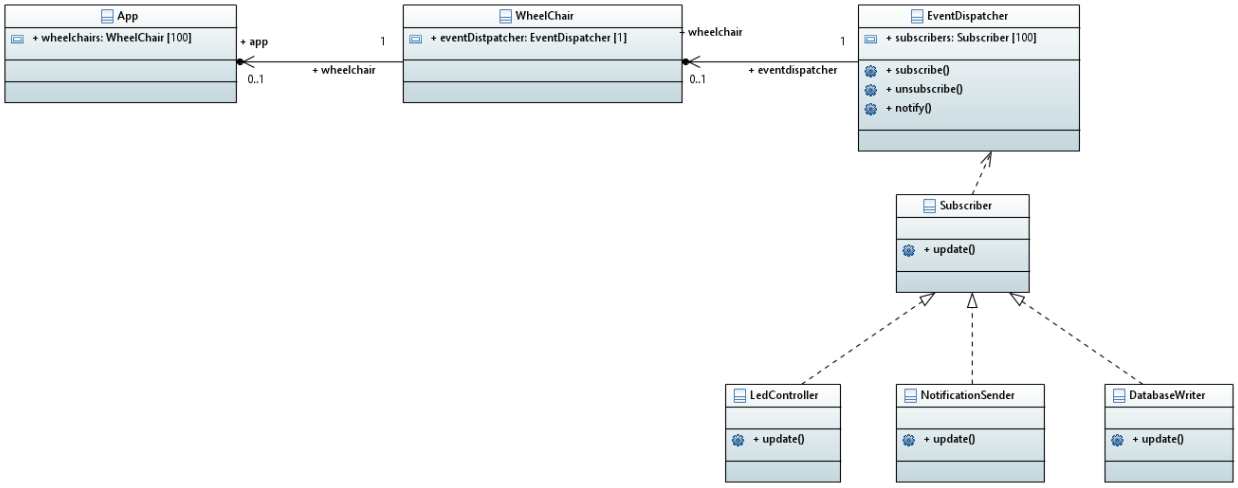
Scenario (Problem)

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.

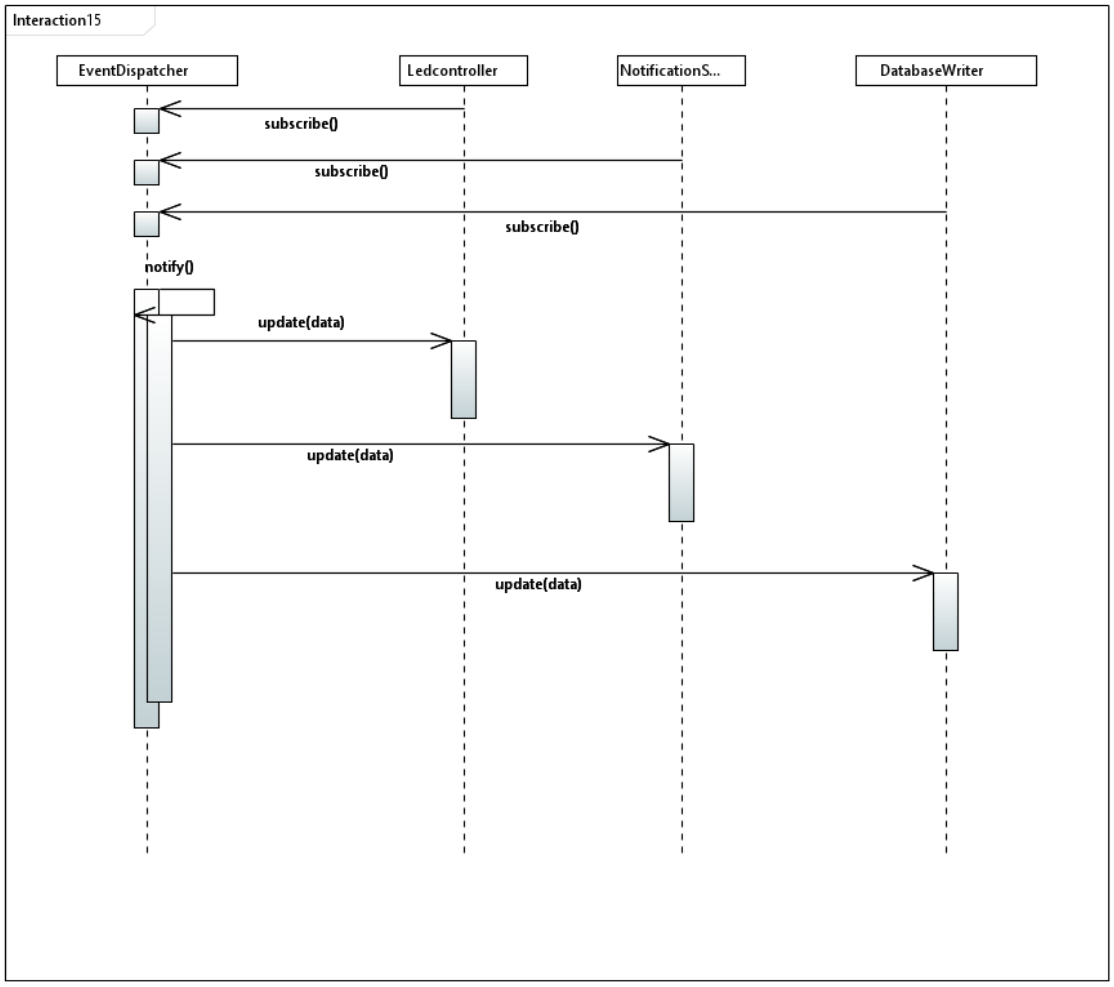
Solution

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);
    }

    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

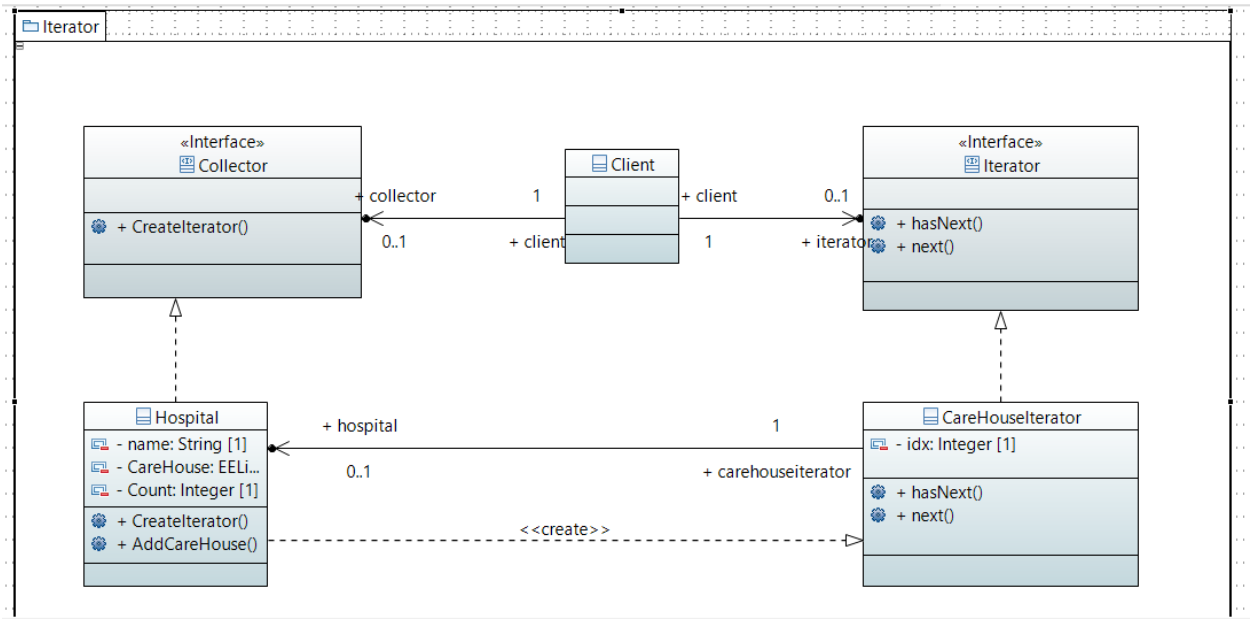
Scenario (Problem)

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.

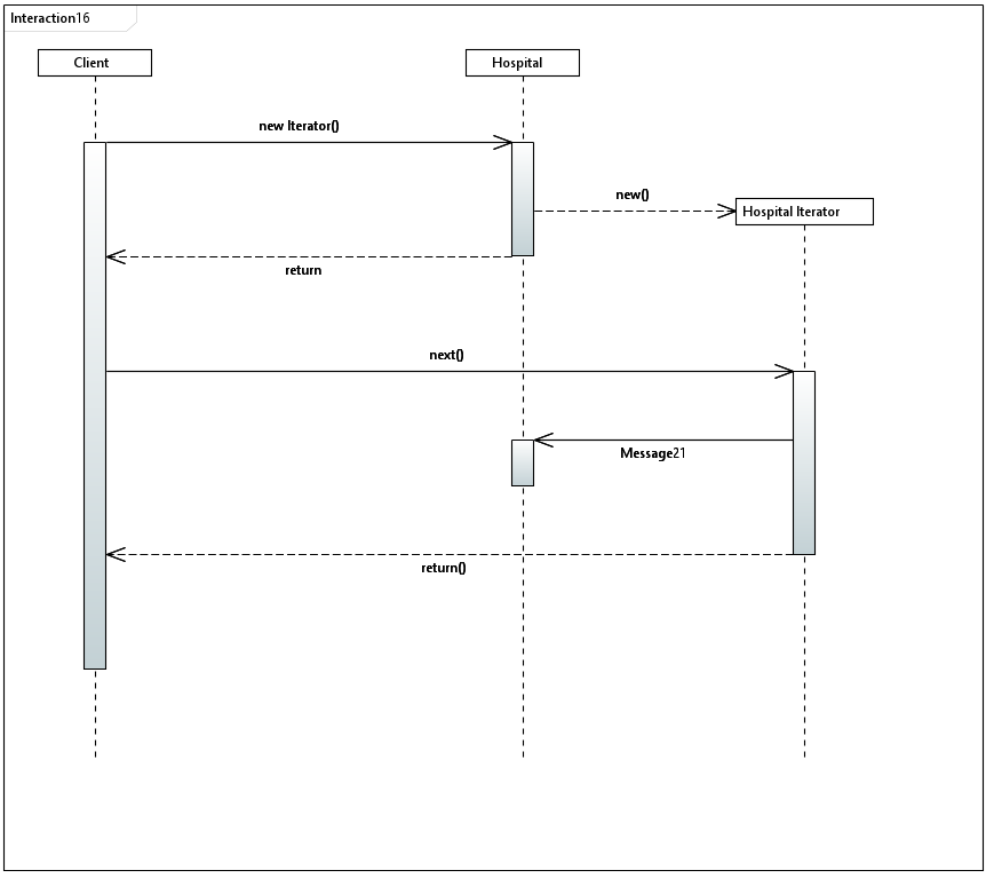
Solution

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

```
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

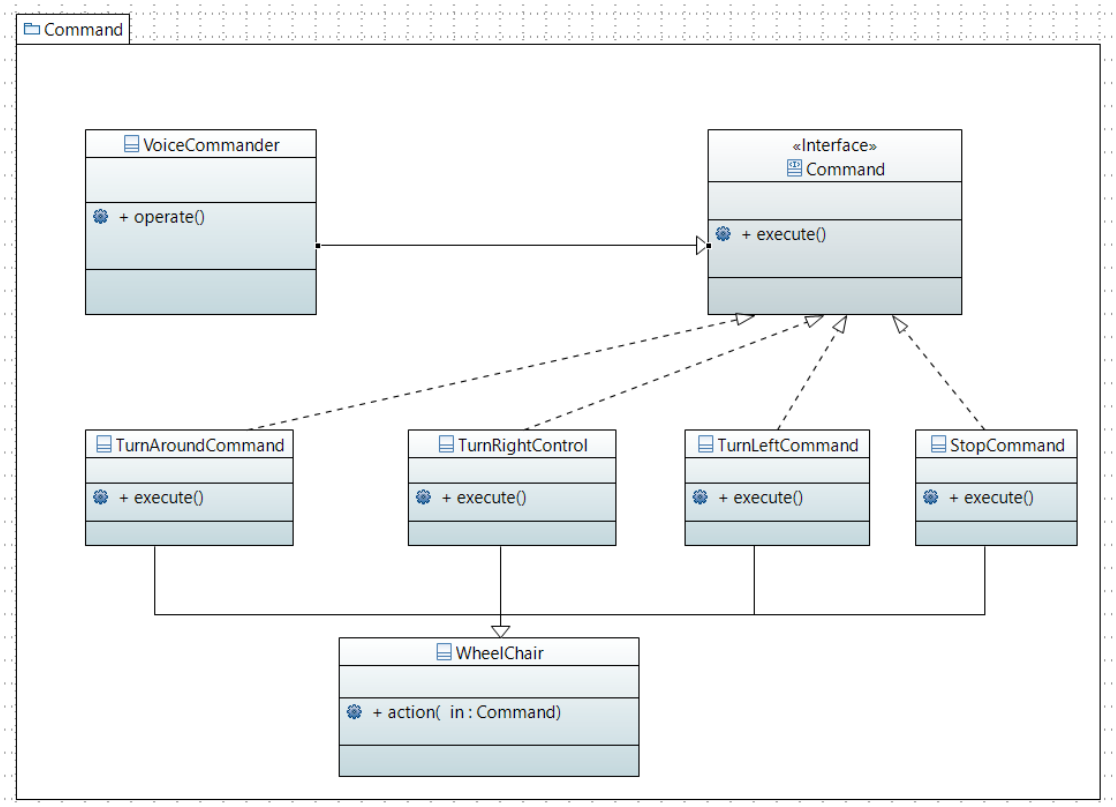
Solution

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

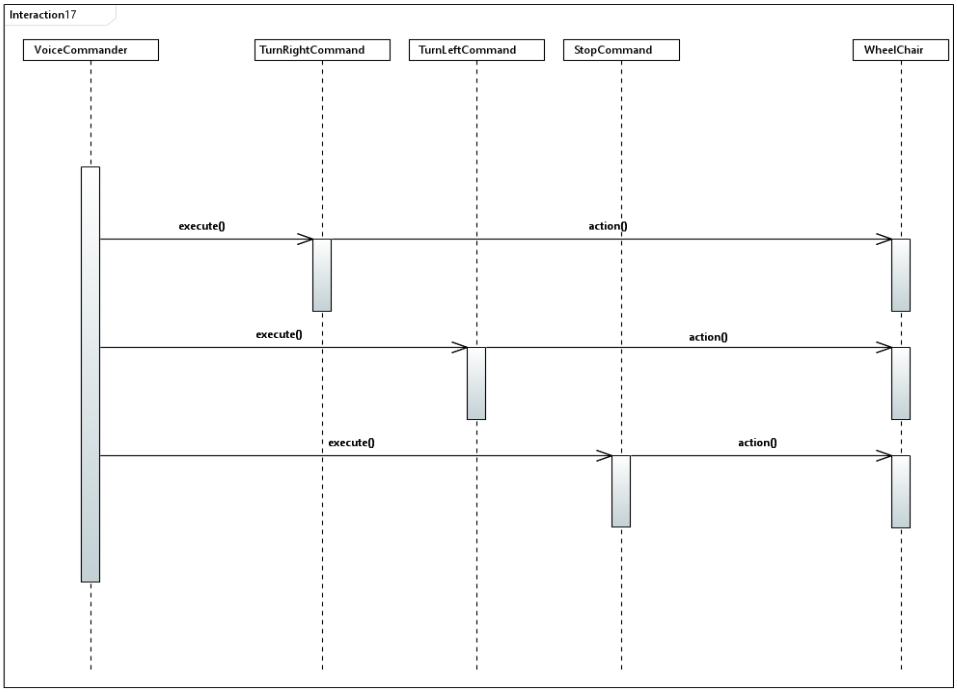
The implementation of a command interface which could be implemented in each separated concrete command classes , so that the Invoker(Command Executer) is not directly involved with the reciever(). Here, we need not keep track of changes , but if a need appears we could keep track of the movement of 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

```
//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.:

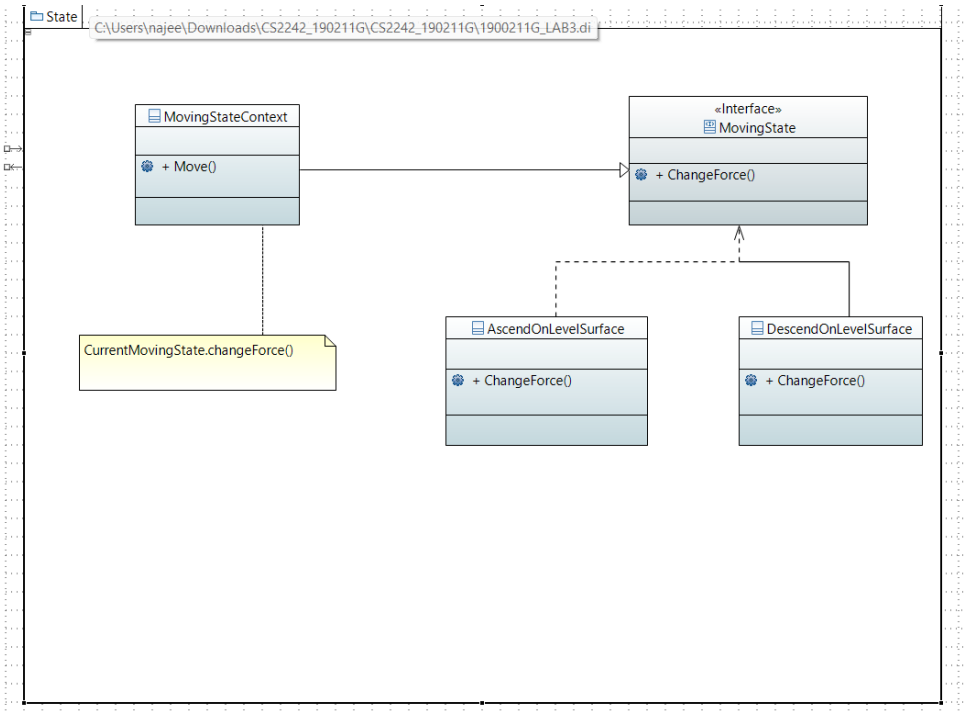
> If we are ascending , we would need to increase the force on wheel chair in the direction of movement to maintain a speed. If we are descending , we would need to increase the force on wheel chair opposite to the direction of movement to maintain a speed. If we are moving on a level surface then, a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

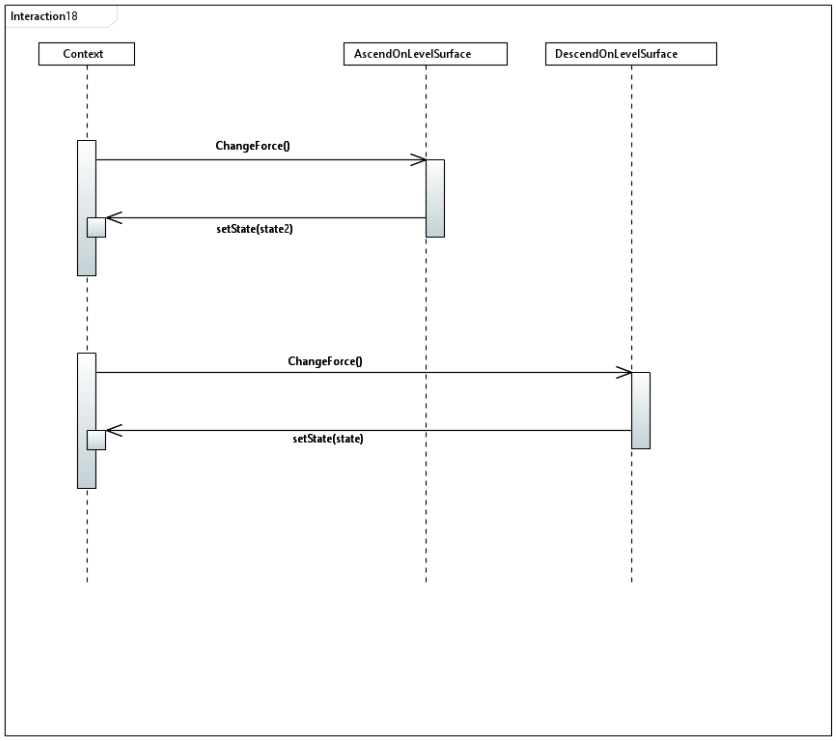
Solution

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!");
    }
  }
  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

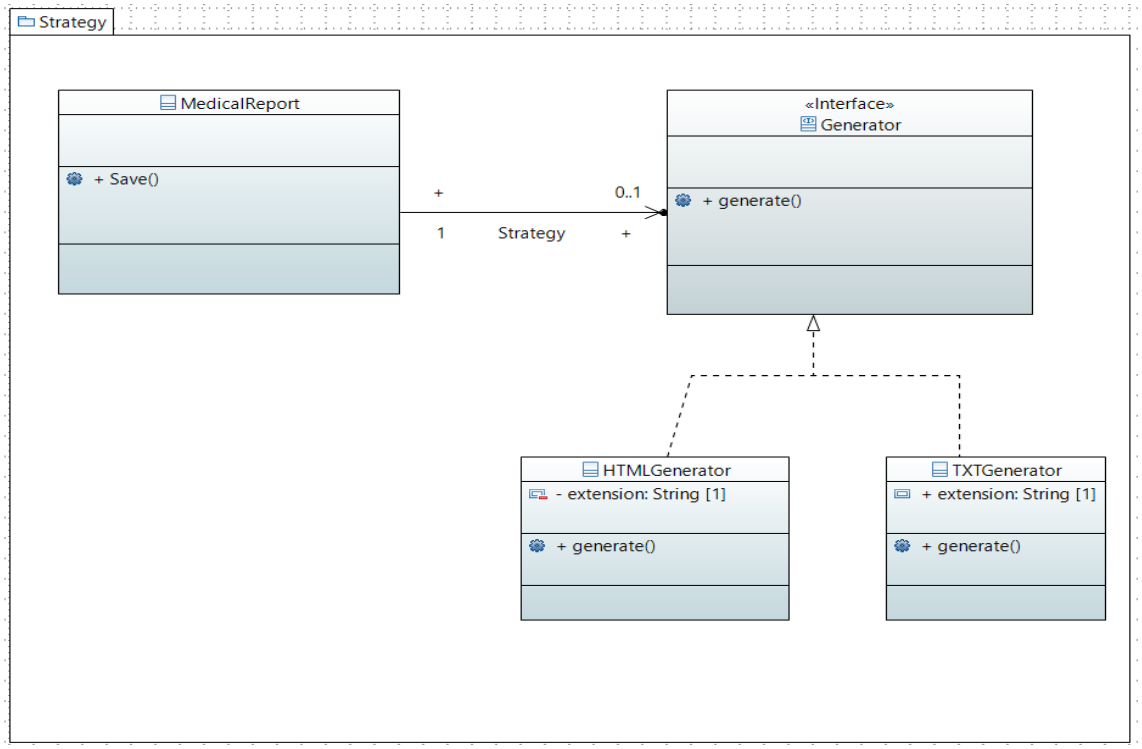
Scenario (Problem)

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.

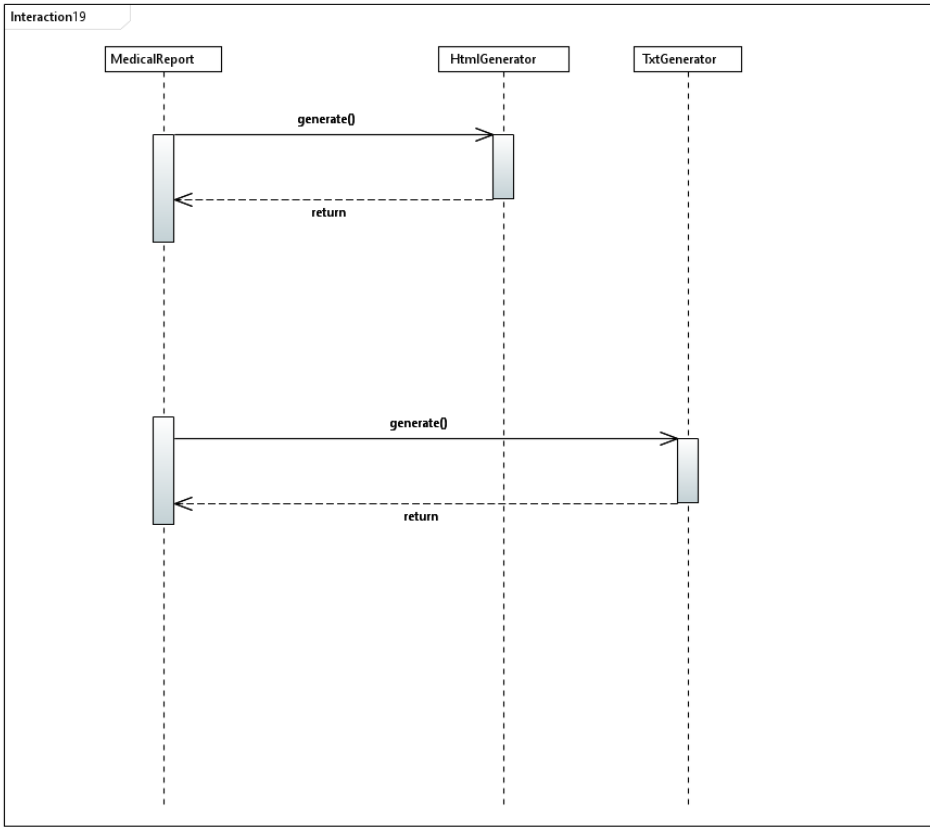
Solution

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
```

```
    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");

    }
}
```