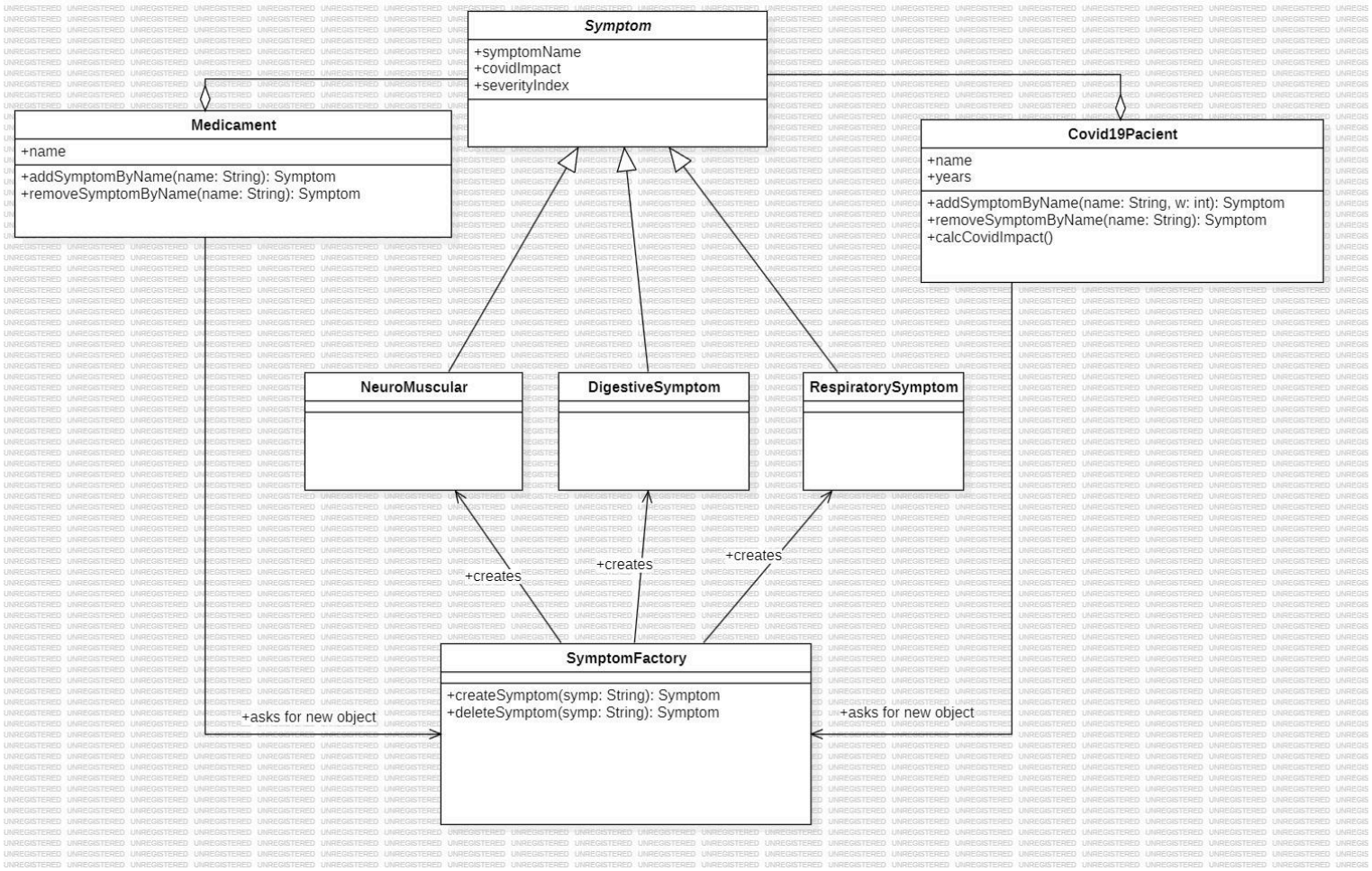
IS-2 / LABORATORIO

PATRONES DE DISEÑO

PABLO ALCOLEA, IKER GARCÍA, UNAI LEÓN

1. SIMPLE FACTORY



Cuestiones de la aplicación:

A. ¿Qué sucede si aparece un nuevo síntoma (por ejemplo, mareos)?

Habría que añadirlo en la ComboBox de manera individual añadiendo su nombre a la hora de crear un nuevo objeto Symptom cada vez, cambiando el código de todas las clases en las que se utilicen síntomas (Covid19Pacient y Medicament).

B. ¿Cómo se puede crear un nuevo síntoma sin cambiar las clases existentes (principio OCP)?

No se puede hacer en la versión inicial de la aplicación, no cumple el principio OCP. Debe cambiarse el código dentro de las clases que utilizan síntomas porque es en ellas donde está el método para crear los síntomas.

C. **¿Cuántas responsabilidades tienen las clases de Covid19Pacient y Medicament (principio SRP)?**

   Tienen las responsabilidades de gestión de síntomas para cada una de ellas, y además la responsabilidad de crear los síntomas. No cumplen el principio SRP.

Tareas a realizar:

1. **Realiza un nuevo diseño de la aplicación (diagrama UML) aplicando el patrón Simple Factory para eliminar vulnerabilidades anteriores y mejorar el diseño en general. Describe con claridad los cambios realizados.**

   Siguiendo el patrón Factory, se ha creado la clase SymptomFactory con la responsabilidad única de crear los objetos Symptom. Los objetos que necesiten crear síntomas tendrán un objeto SymptomFactory como atributo, y llamarán al método createSymptom de dicho atributo para crear los síntomas.

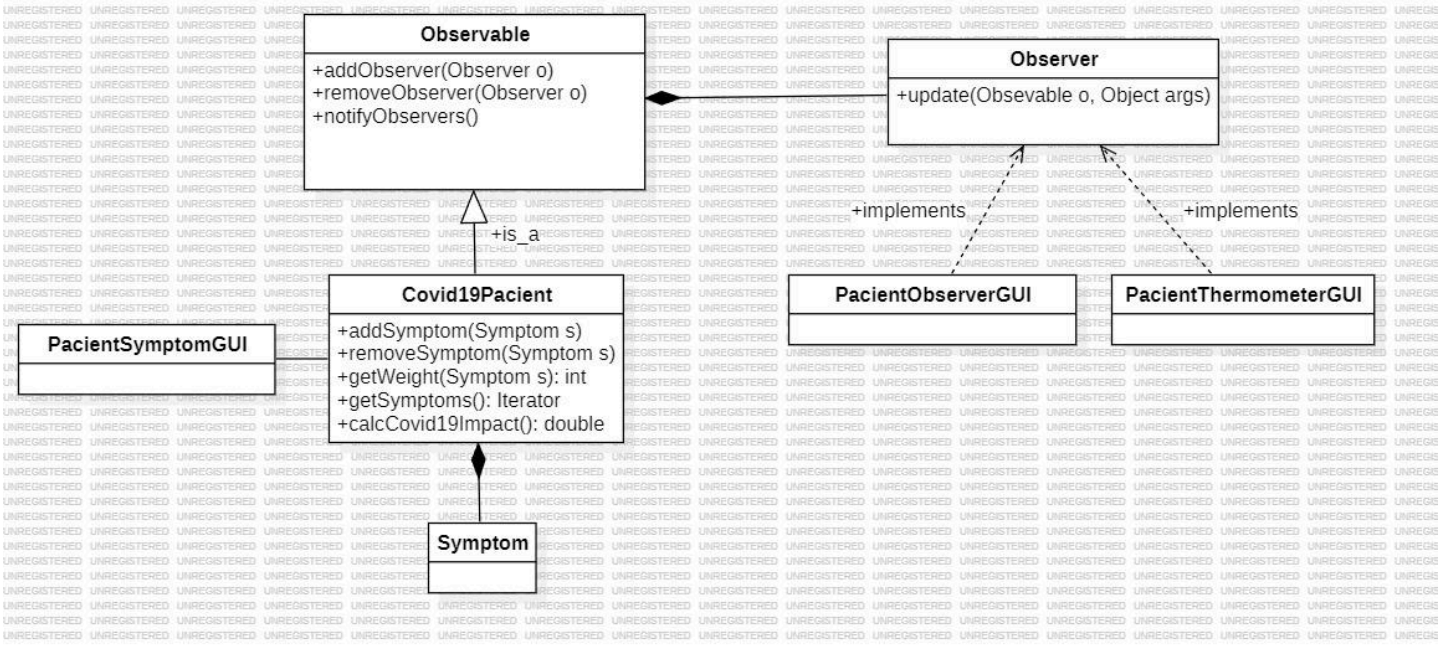
2. **Implementa la aplicación y agrega el nuevo síntoma "mareos" asociado a un tipo de impacto 1.**

   Para implementar los cambios, simplemente hemos movido el método createSymptom que se encontraba repetido en las clases Covid19Pacient y Medicament a la nueva clase SymptomFactory. Después hemos modificado dichas clases para que tengan un nuevo atributo de tipo SymptomFactory, y adaptado el método createSymptom de cada una para que solo llame a la versión implementada en SymptomFactory.

3. **Cómo se puede adaptar la clase Factory, para que los objetos Symptom que utilicen las clases Covid19Pacient y Medicament sean únicos. Es decir, para cada síntoma sólo exista un objeto. (Si hay x síntomas en el sistema, que haya únicamente x objetos Symptom).**

   Habría que implementar la clase SymptomFactory siguiendo el patrón Singleton. El método createSymptom debería comprobar si ya existe un objeto para el síntoma que se está intentando crear; y si existe, devolver el síntoma existente. Si no existe, lo crearía y a partir de entonces devolvería ese mismo objeto cuando se quiera volver a crear ese síntoma.

2. **OBSERVER**



Primero, se ha adaptado la clase Covid19Pacient para extender la clase Observable de Java. Para que notifique a los observadores suscritos, añadimos las llamadas a los métodos setChanged() y notifyObservers() en los métodos que hacen cambios a la lista de síntomas del paciente: addSymptomByName() y removeSymptomByName().

```
@SuppressWarnings("deprecation")
public Symptom addSymptomByName(String symptom, Integer w){
    Symptom s=getSymptomByName(symptom);
    if (s==null) {
        s=createSymptom(symptom);
        symptoms.put(s,w);
    }
    setChanged();
    notifyObservers();
    return s;
}

@SuppressWarnings("deprecation")
public Symptom removeSymptomByName(String symptomName) {
    Symptom s=getSymptomByName(symptomName);
    System.out.println("Simptom to remove: "+s);
    if (s!=null) symptoms.remove(s);
    setChanged();
    notifyObservers();
    return s;
}
```

Después, adaptamos las dos pantallas que van a observar el comportamiento del paciente para que implementen la interfaz Observer. Para ello, añadimos un objeto Observable como parámetro en la constructora, donde asignaremos el paciente a observar en el programa principal.

```
public PacientObserverGUI(Observable obs) {
    setTitle("Pacient symptoms");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setBounds(650, 100, 200, 300);
    contentPane = new JPanel();
    contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
    setContentPane(contentPane);
    contentPane.setLayout(null);
    symptomLabel.setBounds(19, 38, 389, 199);
    contentPane.add(symptomLabel);
    symptomLabel.setText("Still no symptoms");
    this.setVisible(true);
    obs.addObserver(this);
}
```

```
public PacientThermometerGUI(Observable obs) {
    super("Temperature Gauge");
    Panel Top = new Panel();
    add("North", Top);
    gauges = new TemperatureCanvas(0, 15);
    gauges.setSize(500, 280);
    add("Center", gauges);
    setSize(200, 380);
    setLocation(0, 100);
    setVisible(true);
    obs.addObserver(this);
}
```

Para que las pantallas implementen la interfaz Observer, deben implementar el método update() donde se indica qué hacer para reaccionar a los cambios del objeto que están observando.

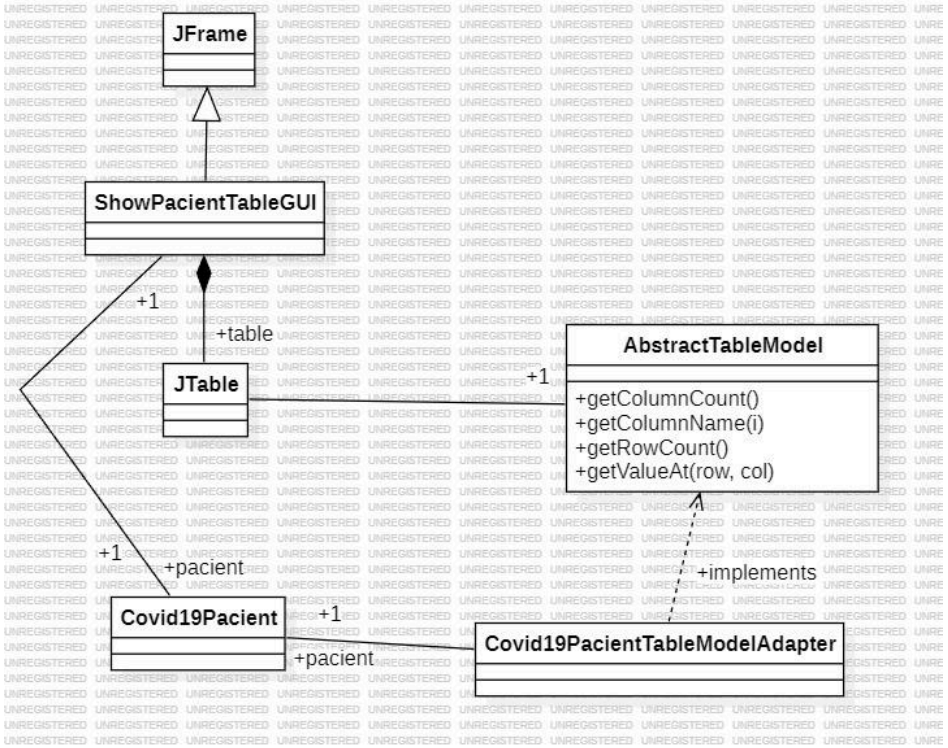
En el caso de PacientSymptomGUI, actualizamos el texto de la pantalla para que muestre todos los síntomas.

```
public void update(Observable o, Object arg) {
    Covid19Pacient p = (Covid19Pacient) o;
    String s = "<html>  Pacient: <b>" + p.getName() + "</b> <br>";
    s = s + "Covid impact:  <b>" + p.covidImpact() + "</b><br><br>";
    s = s + "                              <br>    Symptoms:   <br>";
    Iterator<Symptom> i = p.getSymptoms().iterator();
    Symptom p2;
    while (i.hasNext()) {
        p2 = i.next();
        s = s + "       - " + p2.toString() + ",     " + p.getWeight(p2) + "<br>";
    }
    s = s + "</html>";
    symptomLabel.setText(s);
}
```

En la clase PacientThermometerGUI actualizamos el dibujo del termómetro que aparece en la pantalla según el valor del impacto del Covid del paciente.

```
public void update(Observable o, Object args) {
    Covid19Pacient p = (Covid19Pacient) o;
    int farenheit = (int) p.covidImpact();
    gauges.set(farenheit);
    gauges.repaint();
}
```

3. ADAPTER



Para poder trabajar con la información de los síntomas de un paciente, creamos la clase adaptador Covid19PacientTableModelAdapter, que implementa los métodos de AbstractTableModel para poder gestionar los datos del paciente en tablas. Hay que implementar cuatro métodos.

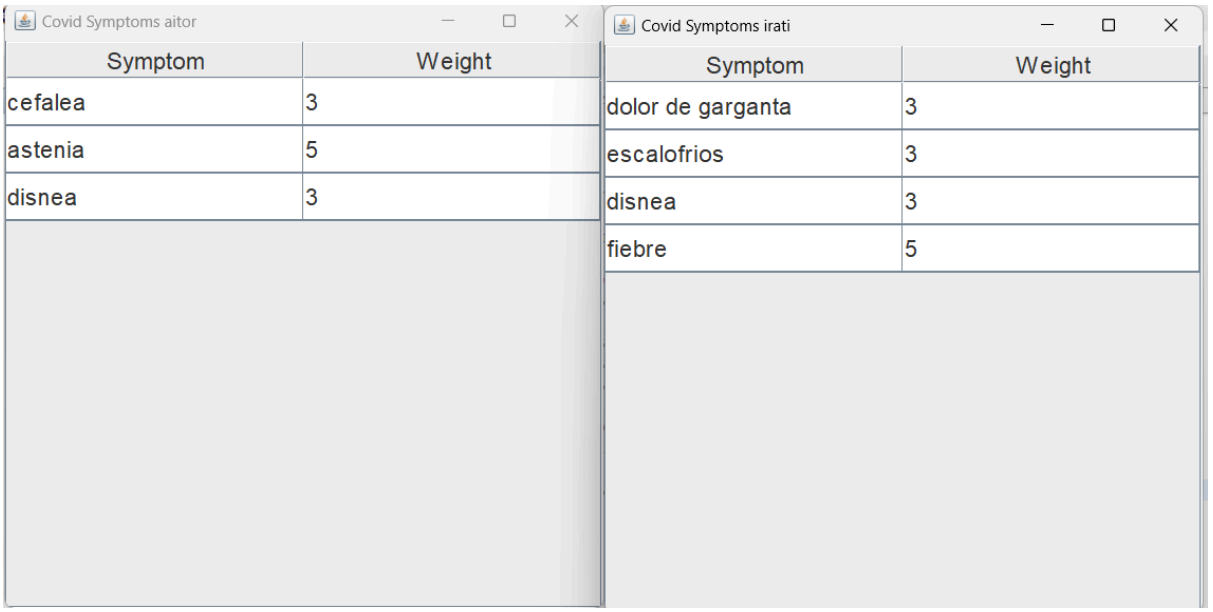
```
public class Covid19PacientTableModelAdapter extends AbstractTableModel {
    protected Covid19Pacient pacient;
    protected String[] columnNames = new String[] { "Symptom", "Weight" };
    protected ArrayList<Symptom> symptomList;

    public Covid19PacientTableModelAdapter(Covid19Pacient p) {
        this.pacient = p;
        this.symptomList = new ArrayList<Symptom>(p.getSymptoms());
    }

    public int getColumnCount() {
        return 2;
    }

    public String getColumnName(int i) {
        return columnNames[i];
    }

    public int getRowCount() {
        return pacient.getSymptoms().size();
    }

    public Object getValueAt(int row, int col) {
        Symptom symptom = symptomList.get(row);
        if (col == 0) {
            return symptom.getName();
        } else if (col == 1) {
            return symptom.getSeverityIndex();
        } else {
            return null;
        }
    }
}
```

El cambio principal es el atributo symptomList, donde guardamos la lista de síntomas del paciente cuando se construye la tabla. La lista de síntomas en la clase Covid19Pacient es un Set de síntomas, que contiene elementos desordenados: no hay forma de obtenerlos mediante índices. Convertimos el Set a ArrayList antes de añadir los elementos a la tabla, para poder indexarlos y obtenerlos de esa forma. Esto es importante para el método getValueAt().

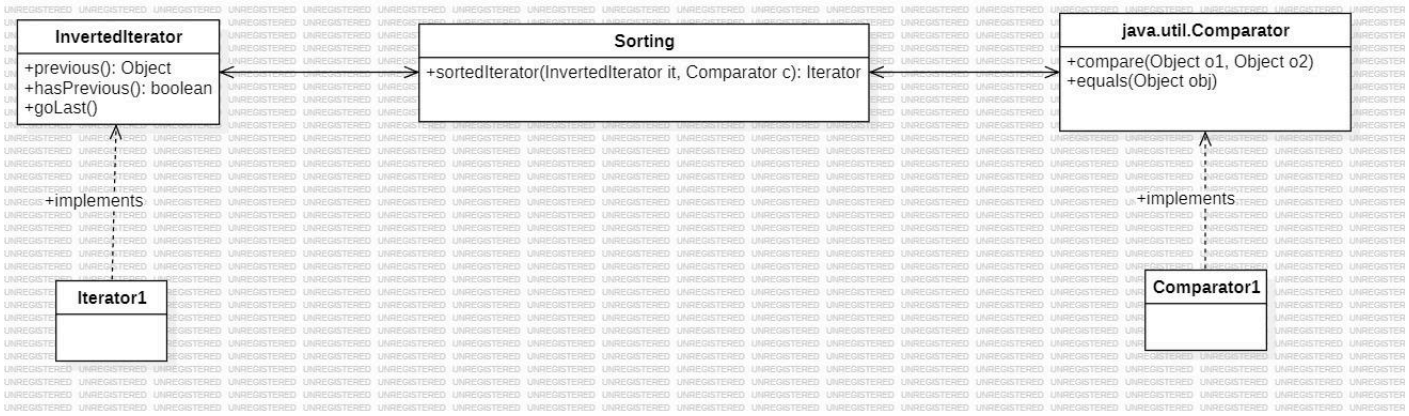
En el método principal añadimos un nuevo paciente con sus síntomas para probar el modelo de la tabla.

```
public static void main(String[] args) {
    Covid19Pacient pacient=new Covid19Pacient("aitor", 35);

    pacient.addSymptomByName("disnea", 2);
    pacient.addSymptomByName("cefalea", 1);
    pacient.addSymptomByName("astenia", 3);

    ShowPacientTableGUI gui=new ShowPacientTableGUI(pacient);
    gui.setVisible(true);

    Covid19Pacient pacient2 = new Covid19Pacient("irati", 44);

    pacient2.addSymptomByName("dolor de garganta", 2);
    pacient2.addSymptomByName("disnea", 1);
    pacient2.addSymptomByName("fiebre", 2);
    pacient2.addSymptomByName("escalofrios", 3);

    ShowPacientTableGUI gui2 = new ShowPacientTableGUI(pacient2);
    gui2.setVisible(true);

}
```

Finalmente, al ejecutar el método, obtenemos la tabla con la información de ambos pacientes.



4. ITERATOR + ADAPTER



Tareas a realizar:

- En la creación de la clase Main, se ha creado un Covid19Pacient, se ha creado un paciente con nombre “Ane” de 29 años. A ese objeto, se han añadido 5 síntomas con distintos nombres, grados de severidad e impacto Covid:

```
public class Main {

    public static void main(String[] args) {
        Covid19Pacient p=new Covid19Pacient("Ane", 29);
        p.addSymptom(new Symptom("fatiga", 10, 5), 1);
        p.addSymptom(new Symptom("tos", 8, 7), 2);
        p.addSymptom(new Symptom("fiebre", 9, 9), 3);
        p.addSymptom(new Symptom("dolor de cabeza", 6, 4), 4);
        p.addSymptom(new Symptom("dificultad para respirar", 10, 10), 5);
```

- Se han implementado dos clases ComparatorBySymptomName y ComparatorBySeverityIndex, que implementan la interfaz Comparator<Object>:

```java
public class ComparatorBySymptomName implements Comparator<Object> {

//  @Override
    public int compareSymptom(Symptom s1, Symptom s2) {
        return s1.getName().compareTo(s2.getName());
    }

    @Override
    public int compare(Object o1, Object o2) {
        return this.compareSymptom((Symptom) o1, (Symptom) o2);
    }
}
```

```java
public class ComparatorBySeverityIndex implements Comparator<Object>{

//  @Override
    public int compareSymptom(Symptom s1, Symptom s2) {
        return Integer.compare(s1.getSeverityIndex(), s2.getSeverityIndex());
    }

    @Override
    public int compare(Object o1, Object o2) {
        return this.compareSymptom((Symptom) o1, (Symptom) o2);
    }
}
```

En ellas, se ha implementado el método compare() de la interfaz Comparator y los método compareSymptom() para poder realizar el ejercicio y no tener problemas de casting. En los métodos compareSymptom() se realizan las comparaciones requeridas para realizar el ejercicio.

- Se ha creado la clase Covid19PacientAdapter, que implementa la interfaz InvertedIterator.

```java
public class Covid19PacientAdapter implements InvertedIterator{

    private List<Symptom> symptoms;
    private int currentPosition;

    public Covid19PacientAdapter(Covid19Pacient p) {
        this.symptoms = new ArrayList<>(p.getSymptoms());
        Collections.reverse(this.symptoms);
        this.currentPosition = this.symptoms.size() - 1;
    }

    @Override
    public Object previous() {
        if (hasPrevious()) {
            return symptoms.get(currentPosition--);
        }
        return null;
    }

    @Override
    public boolean hasPrevious() {
        return currentPosition >= 0;
    }

    @Override
    public void goLast() {
        currentPosition = symptoms.size() - 1;
    }

}
```

En ella se han implementado los métodos previous(), hasPrevious() y goLast() propios de la interfaz. Además, se ha creado una constructora que recibe un Covid19Pacient como parámetro para acceder a la información del paciente.

- Finalmente, este es el resultado de la clase Main completa y de la ejecución de la misma.

```java
public class Main {

    public static void main(String[] args) {
        Covid19Pacient p=new Covid19Pacient("Ane", 29);
        p.addSymptom(new Symptom("fatiga", 10, 5), 1);
        p.addSymptom(new Symptom("tos", 8, 7), 2);
        p.addSymptom(new Symptom("fiebre", 9, 9), 3);
        p.addSymptom(new Symptom("dolor de cabeza", 6, 4), 4);
        p.addSymptom(new Symptom("dificultad para respirar", 10, 10), 5);



        Covid19PacientAdapter invertedIterator = new Covid19PacientAdapter(p);

        System.out.println("Síntomas ordenados por nombre: \n");
        Iterator<Object> sortName = Sorting.sortedIterator(invertedIterator, new ComparatorBySymptomName());
        int count = 0;

        while(sortName.hasNext() && count < 5) {
            Symptom s = (Symptom) sortName.next();
            System.out.println(s.getName() + "\n");
            count++;
        }

        invertedIterator.goLast();

        System.out.println("\n");
        System.out.println("Síntomas ordenados por índice de severidad: \n");
        Iterator<Object> sortInd = Sorting.sortedIterator(invertedIterator, new ComparatorBySeverityIndex());
        count = 0;

        while(sortInd.hasNext() && count < 5) {
            Symptom s = (Symptom) sortInd.next();
            System.out.println(s.getName() + " - Severidad: " + s.getSeverityIndex() + "\n");
            count++;
        }
    }
}
```

Como se puede ver, los síntomas añadidos están ordenados alfabéticamente primero y por índice de severidad (ascendente) después.

```
Síntomas ordenados por nombre:

dificultad para respirar

dolor de cabeza

fatiga

fiebre

tos


Síntomas ordenados por índice de severidad:

dolor de cabeza - Severidad: 4

fatiga - Severidad: 5

tos - Severidad: 7

fiebre - Severidad: 9

dificultad para respirar - Severidad: 10
```