PATRONES DE DISEÑO

DISEÑO AVANZADO

1. SIMPLE FACTORY

Hay muchas vulnerabilidades en esta aplicación:

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

Si quisiéramos añadir un nuevo síntoma como mareos, habría que cambiar el método *createSympton()* en ambas clases (*Medicament* y *Covid19Pacient)*, esto incumple el principio abierto-cerrado (Open Closed Principle OCP).

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

Para cumplir con el principio OCP, la creación de síntomas debería ser responsabilidad desligada de ambas clases. Crear una clase *SymptomFactory* que se encargue de instanciar objetos tipo *Sympton*.

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

En ambas clases, las clases *Covid19Pacient* y *Medicament* tienen más de una responsabilidad, violando así el principio de una única responsabilidad (Single Responsibility Principle).

NUEVO DIAGRAMA UML Y EXPLICACIÓN:

Antes de la refactorización, como hemos mencionado en el apartado A, las clases Medicament y Covid19Pacient tenian el método createSympton(), entre la duplicación de código y la violación de los principio OCP y SRP.

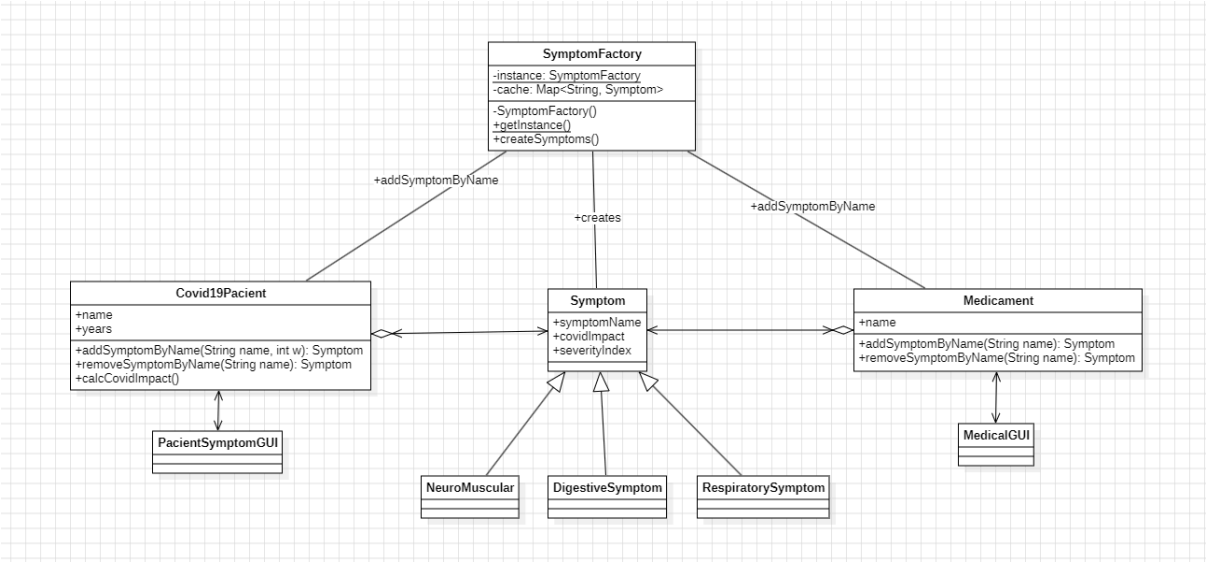
Para solventar el problema, se ha creado una nueva clase *SymptonFactory*, encargada de crear nuevas instancias de la clase *Sympton*, de esta manera cada vez que una clase necesite podrá delegar esa tarea a la clase creada, que mejorará el control de los síntomas creados.

Como la factoría será única y global, es conveniente que esta nueva clase sea de tipo Singleton.

IMPLEMENTACIÓN DEL NUEVO SÍNTOMA “MAREOS”:

Añadiendo el síntoma en la clase SymptomFactory con su correspondiente impacto.

ADAPTACIÓN CLASE FACTORY:



2. PATRÓN OBSERVER

El programa principal ahora crea 2 pacientes:

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

     Observable pacient2 = new Covid19Pacient("maria", 42);
     new PacientObserverGUI(pacient2);
     new PacientSymptomGUI((Covid19Pacient) pacient2);
}
```

Se abrirán cuatro ventanas: 2 de tipo PacientSymptomGUI (una para cada paciente)2 de tipo PacientObserverGUI (una para cada paciente). Ahora cuando se agregue o elimine síntomas en la interfaz de aitor,

solo se actualizará su ventana de observación. Lo mismo ocurrirá para maria.

Cambios para que PacientThermometerGui funcione:

En la clase implementamos la clase Observer, se añade a la constructora un parámetro Observable y la sentencia obs.addObserver(this);

```
public class PacientThermometerGUI extends Frame implements Observer{
    private TemperatureCanvas gauges;
    /**
     * @wbp.nonvisual location=119,71
     */
    private final JLabel label = new JLabel("New label");

    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);
    }
```

También añadimos el método que nos indica el enunciado:

```
public void update(Observable o, Object args) {
    Covid19Pacient p=(Covid19Pacient) o;
        Obtain the current covidImpact to paint
    int farenheit = (int) p.covidImpact();
        temperature gauge update
    gauges.set(farenheit);
    gauges.repaint();
    }
```

Y en el programa principal lo creamos:

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

    Observable pacient2 = new Covid19Pacient("maria", 42);
    new PacientObserverGUI(pacient2);
    new PacientSymptomGUI((Covid19Pacient) pacient2);
    new PacientThermometerGUI (pacient2);
}
```

3. PATRÓN ADAPTER

Añadimos el codigo necesario en la clase Covid19PacientTableModelAdapter:

```
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) {

    Set<Symptom> ls = pacient.getSymptoms();
    Object[] ss = ls.toArray();
    Symptom s = (Symptom)ss[row];

    if(col==0) return s.getName();
    if(col==1) return pacient.getWeight(s);
    return null;
}
```

Añadimos un segundo paciente con sus síntomas y utilizamos otra tabla para enseñarla:

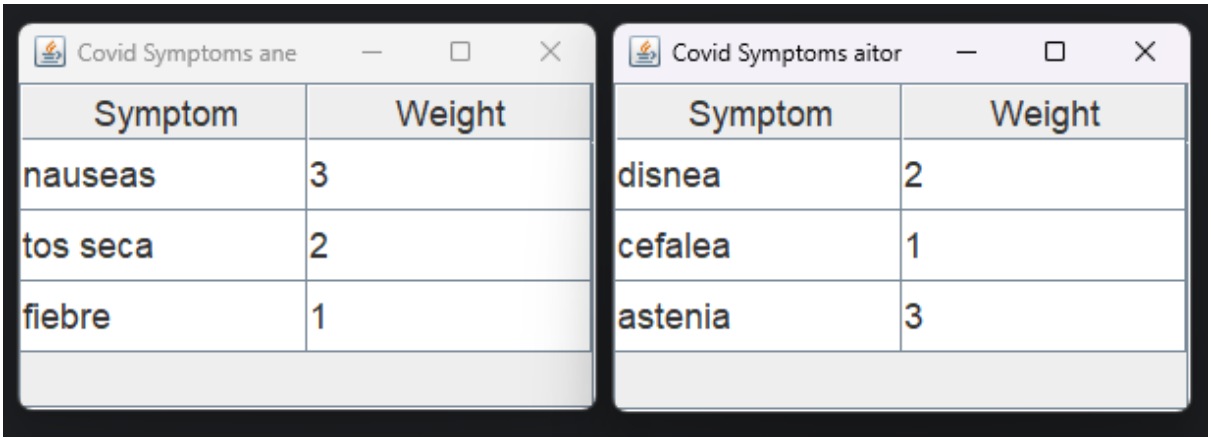
```
public static void main(String[] args) {
    Covid19Pacient pacient=new Covid19Pacient("aitor", 35);
    Covid19Pacient pacient2=new Covid19Pacient("ane", 31);

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

    pacient2.addSymptomByName("fiebre", 1);
    pacient2.addSymptomByName("nauseas", 3);
    pacient2.addSymptomByName("tos seca", 2);

    ShowPacientTableGUI gui=new ShowPacientTableGUI(pacient);
    gui.setPreferredSize(
              new java.awt.Dimension(1300, 1200));
    gui.setVisible(true);

    ShowPacientTableGUI gui2=new ShowPacientTableGUI(pacient2);
    gui2.setPreferredSize(
              new java.awt.Dimension(1300, 1200));
    gui2.setVisible(true);
}
```



Cómo podrías añadir esta nueva pantalla al ejercicio anterior del observer, de forma que cada vez que se añada un nuevo síntoma a un paciente, se actualice la tabla.

Para que se pueda actualizar la tabla se tendria que implementar Observer a la clase Covid19PacientTableModelAdapter, ahi, integramos en la constructora el Observer con pacient.addObserver(this); Después se integra el método update() haciendo un Override y que dentro llame a un método de AbstractTableModel diciendo que han cambiado los datos, cambiando en la vista los síntomas del paciente.

4. PATRÓN ITERATOR ADAPTER

Primero implementamos las Interfaces Comparator en dos nuevas clases, una para ordenar los síntomas por nombre y otra para ordenarlos por severidad.

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

    @Override
    public int compare(Object arg0, Object arg1) {
        Symptom s1 = (Symptom) arg0;
        Symptom s2 = (Symptom) arg1;
        return s1.getName().compareToIgnoreCase(s2.getName());

    }
}
```

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

    @Override
    public int compare(Object arg0, Object arg1) {
        Symptom s1 = (Symptom) arg0;
        Symptom s2 = (Symptom) arg1;
        return Integer.compare(s1.getSeverityIndex(), s2.getSeverityIndex());
    }

}
```

Creamos el patrón Adapter sobre la clase Covid19Pacient, como la clase no se puede cambiar, creamos una nueva con los métodos implementados y su constructora.

```java
public class Covid19PacientInvertedAdapter implements InvertedIterator{

    private List<Symptom> symptons;
    private int index;

    public Covid19PacientInvertedAdapter(Covid19Pacient pacient) {
        this.symptons = new ArrayList<>(pacient.getSymptoms());
        this.index = symptons.size();
    }
    @Override
    public Object previous() {
        if (hasPrevious()) {
            index--;
            return symptons.get(index);
        }
        return null;
    }

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

    @Override
    public void goLast() {
        index=symptons.size();
    }
```

Por último creamos el programa principal que utilizará el Sorting.sortedIterator para ordenar los síntomas tanto por nombre como por severidad del paciente creado.

```java
public class MainIteratorAdapter {
    public static void main(String[] args) {

        Covid19Pacient pacient = new Covid19Pacient("Aitor", 35);
        pacient.addSymptomByName("fiebre", 3);
        pacient.addSymptomByName("tos seca", 2);
        pacient.addSymptomByName("astenia", 1);
        pacient.addSymptomByName("cefalea", 1);
        pacient.addSymptomByName("nauseas", 2);

        InvertedIterator adapter = new Covid19PacientInvertedAdapter(pacient);

        System.out.println("=== Ordenado por nombre ===");
        Iterator<Object> itName = Sorting.sortedIterator(adapter, new SymptomNameComparator());
        while (itName.hasNext()) {
            System.out.println(itName.next());
        }

        System.out.println("\n=== Ordenado por severidad ===");
        adapter.goLast();
        Iterator<Object> itSeverity = Sorting.sortedIterator(adapter, new SymptomSeverityComparator());
        while (itSeverity.hasNext()) {
            System.out.println(itSeverity.next());
        }
    }
```

La salida nos dará los síntomas ordenados por nombre primero y luego por severidad, sin haber modificado la clase Covid19Pacient ni el método Sorting.sortedIterator