

Introducción

En este laboratorio plantearán varios ejercicios utilizando los patrones de diseño: 1.-Simple Factory, 2.- Observer y 3.-Adapter.

El código de este laboratorio (y los ficheros UML) los puedes encontrar en el siguiente repositorio github: <https://github.com/jononekin/labpatterns>.

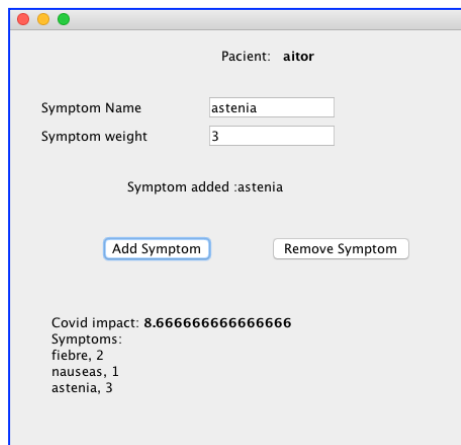
Haz una copia del proyecto (github) y cuando finalices deja el código actualizado (commit).

Entrega (por grupos)

Unicamente la URL del repositorio del laboratorio. En este repositorio estará el código, y un documento labpatterns.pdf. En este documento, para cada ejercicio, se deberá responder con detalle a las preguntas que se formulan y completarlo en su caso, con el código correspondiente a las modificaciones efectuadas.

1.- Simple Factory

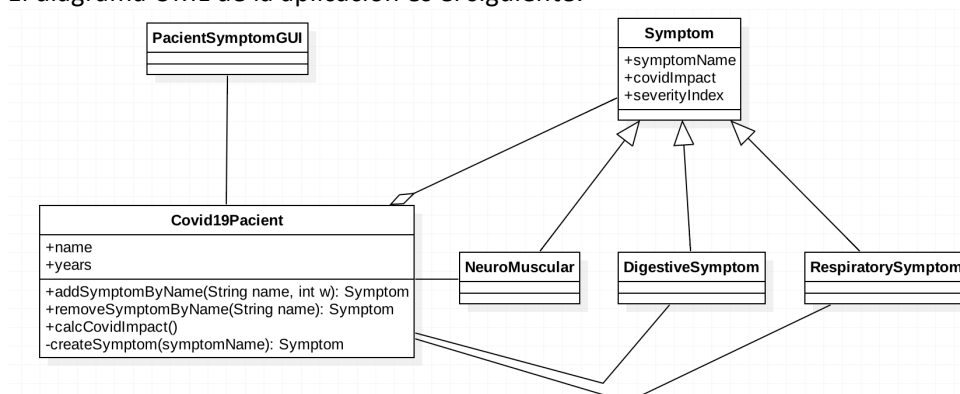
Se ha desarrollado una aplicación para actualizar los síntomas de un paciente a través de una interfaz. La clase PatientSymptomGUI es la que permite dicha actualización.



El método main de la clase Main que se ejecuta es el siguiente:

```
public static void main(String[] args) {  
    Covid19Pacient p1=new Covid19Pacient("aitor", 35);  
    PatientSymptomGUI psGUI1 = new PatientSymptomGUI(p1);  
}
```

El diagrama UML de la aplicación es el siguiente:



La clase PatientSymptomGUI actualiza la información de un usuario, indicando un nombre de síntoma y un peso y cuando se presionan los botones "Add Symptom" y "Remove Symptom", se llaman respectivamente a los métodos addSymptom y removeSymptom realizando las tareas pertinentes. El código addSymptom es el siguiente:

```
(1) public void addSymptom(Covid19Pacient p, String symptomName) {  
(2)     Symptom s;  
(3)     if (isNumeric(weightField.getText())) {  
(4)         if (p.getSymptomByName(symptomName)==null) {  
(5)             if (Integer.parseInt(weightField.getText())<=3) {  
(6)                 s=p.addSymptomByName(symptomName,  
Integer.parseInt(weightField.getText()));  
(7)                 if (s!=null) {  
(8)                     errorLabel.setText("Symptom added :"+symptomName);  
(9)                     reportLabel.setText(createReport());  
(10)                } else errorLabel.setText("ERROR, Symptom  
"+symptomName+ " does not exist ");  
(11)                } else errorLabel.setText("ERROR, Weight between  
[1..3]");  
(12)                } else errorLabel.setText("ERROR, Symptom  
"+symptomName+" already assigned ");  
(13)            } else errorLabel.setText("ERROR, weight must be an  
integer");  
(14)        }
```

Primero se verifica si el paciente tiene ese síntoma (llamando al método getSymptomByName (línea (4)) y, si aún no existe (==null), crea un nuevo síntoma llamando al método addSymptomByName del objeto Covid19Pacient (línea 6). La descripción del método addSymptomByName es la siguiente:

```
class Covid19Pacient {  
    public Symptom addSymptomByName(String symptom, Integer w){  
        Symptom s;  
        s=createSymptom(symptom);  
        if (s!=null)  
            symptoms.put(s,w);  
        return s;  
    }  
}
```

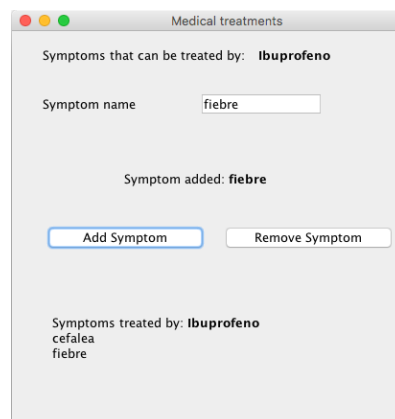
Primero crea un nuevo síntoma llamando al método privado createSymptom. A continuación, si existe se lo añade con su peso a los síntomas del paciente. La implementación del método createSymptom es la siguiente:

```
private Symptom createSymptom(String symptomName) {
    List<String> impact5 = Arrays.asList("fiebre", "tos seca", "astenia",
                                         "expectoracion");
    List<Double> index5 = Arrays.asList(87.9, 67.7, 38.1, 33.4);
    List<String> impact3 = Arrays.asList("disnea", "dolor de garganta",
                                         "cefalea", "mialgia", "escalofrios");
    List<Double> index3 = Arrays.asList(18.6, 13.9, 13.6, 14.8, 11.4);
    List<String> impact1 = Arrays.asList("nauseas", "vómitos", "diarrea",
                                         "congestión nasal", "hemoptisis", "congestion conjuntival");
    List<Double> index1 = Arrays.asList(5.0, 4.8, 3.7, 0.9, 0.8);
    int impact=0;
    double index=0;
    if (impact5.contains(symptomName)) {
        impact=5; index= index5.get(impact5.indexOf(symptomName));
    }
    else if (impact3.contains(symptomName)) {
        impact=3; index= index3.get(impact3.indexOf(symptomName));
    }
    else if (impact1.contains(symptomName)) {
        impact=1; index= index1.get(impact1.indexOf(symptomName));
    }

    if (impact!=0) return new Symptom(symptomName,(int)index, impact);
    else return null;
}
```

La aplicación estaría bien diseñada si los objetos de la clase Symptom solo se usaran en Covid19Pacient, pero ¿y si otras clases quisieran crear objetos de la clase Symptom?

Por ejemplo, supongamos que queremos ampliar la aplicación con una GUI diferente. En esa GUI (MedicamentGUI), queremos tener asociados los síntomas para los cuales está recomendado un medicamento (p.ej. Ibuprofeno). La interfaz de esta aplicación se muestra en la siguiente imagen:

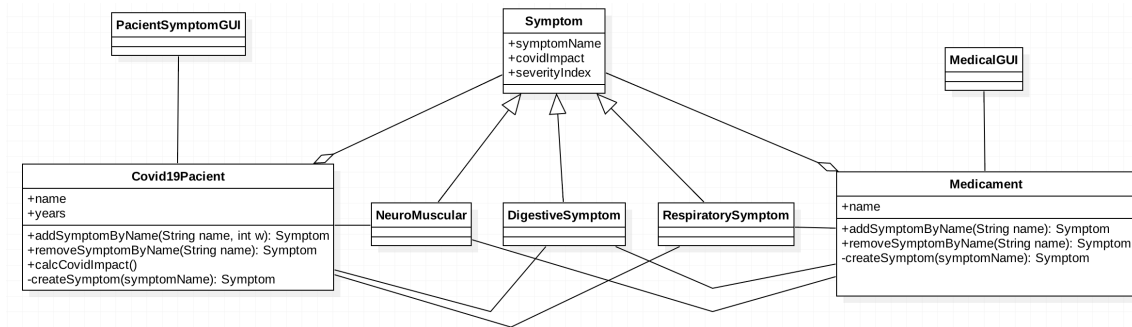


Para ver la interfaz, cambia el programa principal y añade el siguiente código:

```
public static void main(String[] args) {
    ...
    Medicament m=new Medicament("Ibuprofeno");
    MedicalGUI mgui=new MedicalGUI(m);
}
```

Cuando se presiona el botón "Add Symptom", se llama al método addSymptomByName de la clase Medicament que crea un síntoma llamando a su propio método privado createSymptom, pero ... Sí, como te imaginas, es el mismo método que el de Covid10Pacient. Este es el diseño de toda la aplicación hasta ahora:

Ingeniería del Software II
LAB Tema 3- Diseño Avanzado: 3.2- Patrones de diseño



Hay muchas vulnerabilidades en esta aplicación:

- ¿Qué sucede si aparece un nuevo síntoma (por ejemplo, mareos)?
- ¿Cómo se puede crear un nuevo síntoma sin cambiar las clases existentes (principio OCP)?
- ¿Cuántas responsabilidades tienen las clases de Covid19Patient y Medicament (principio SRP)?

Se pide:

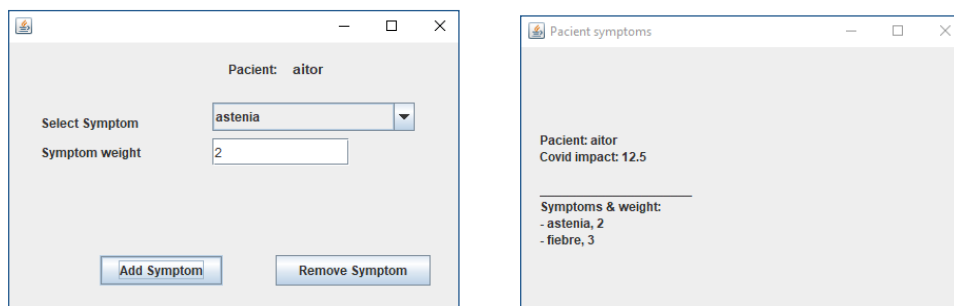
- Realiza un nuevo diseño de la aplicación (diagrama UML) aplicando el patrón Simple Factory para eliminar vulnerabilidades anteriores y mejorar el diseño en general. Describe con claridad los cambios realizados.
- Implementa la aplicación y agrega el nuevo síntoma "mareos" asociado a un tipo de impacto 1.
- Cómo se puede adaptar la clase Factory, para que los objetos Symptom que utilicen las clases Covid19Patient y Medicament sean únicos. Es decir, para cada síntoma sólo exista un objeto. (Si hay x síntomas en el sistema, haya únicamente x objetos Symptom)

2.- Patrón Observer

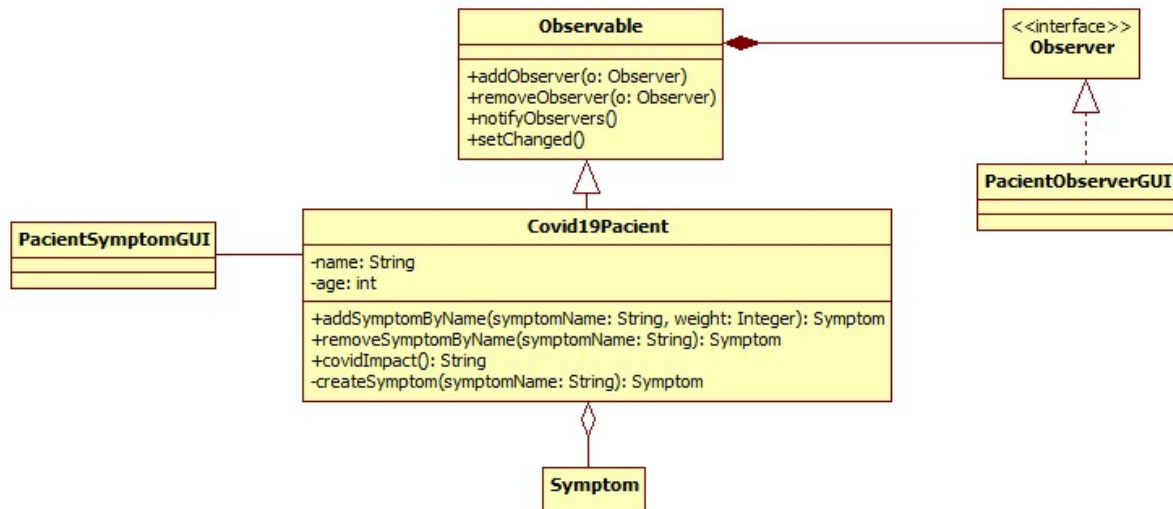
Vamos a modificar un poco la aplicación Covid anterior. Ahora el objetivo es desarrollar una aplicación para agregar los síntomas de un paciente a través de la GUI (PatientSymptomGUI) y, al mismo tiempo ir actualizando una pantalla de síntomas de dicho paciente (PatientObserverGUI).

Primer Prototipo

En la siguiente figura se muestra qué ocurre cuando se añaden varios síntomas:



Para ello, vamos a desarrollar una aplicación que siga un patrón Observer tal y como se muestra en el siguiente diseño:



La clase Observable y la interfaz Observer de la librería `java.util` nos ofrecen las funcionalidades necesarias para desarrollar el patrón Observer. Hay que señalar que, aunque ambos están en estado deprecated desde la versión de java 9, creemos que es un buen ejemplo para trabajar con el patrón observer. La clase **Covid19Pacient** se caracteriza por el conjunto de síntomas que presenta un paciente con sus pesos. De la actualización de estas clases están interesadas otras clases, así que será una clase Observable. La clase **PacientObserverGUI** visualizará la información del paciente con todos sus síntomas y pesos. Por ello es necesario que esté al tanto de cualquier actualización del paciente, por lo que implementará la interfaz Observer para alertar de la actualización. La clase **PacientSymptomGUI** nos sirve para actualizar los síntomas de un paciente junto con sus pesos.

Puedes encontrar un primer prototipo incompleto de esta aplicación en el package **observer** del código que se ha dado para este laboratorio. Estas son las 4 clases que hay que cambiar para desarrollar el patrón Observer.

Paso 1. Clase CovidPacient (extends Observable).

Copia la clase **CovidPacient** en el package **observer**, para no modificar la original de domain. Deberás cambiar las referencias a esa clase para que no tome la definición de domain. Primero tienes que extender **CovidPacient** con la clase **Observable** (es decir, añadir en la definición de clase **extends Observable**). Cada vez que se añade o elimina un síntoma, se debe informar a sus subscriptores, por lo que en los métodos `addSymptomByName` y `removeSymptomBuName` deberán añadirse las siguientes instrucciones: `setChanged()`; `notifyObservers()`;

Paso 2. PacientObserverGUI (implements Observer).

Se deben realizar tres modificaciones:

- Implementa la clase **Observer**, es decir, añadir a la definición de clase **implements Observer**.
- Añade a la constructora un parámetro **Observable** (`obs`, por ejemplo). Este parámetro será el objeto que queremos subscribir. Por ello, añadiremos en la constructora la sentencia `obs.addObserver(this)`;
- La clase debe añadir el método `update` (más abajo) que implementa la interfaz **Observer**. Este método se ejecutará cuando se modifique el objeto al que estamos suscritos (**Observable**). Para nosotros el valor del prototipo `args` es `null`.

```
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();  
Symptom p2;  
while (i.hasNext()) {  
    p2=i.next();  
    s=s+ " - " + p2.toString()+" , "+p.getWeight(p2)+"<br>";  
}  
s=s+"</html>";  
symptomLabel.setText(s);  
}
```

El método crea un texto HTML en una variable String a través de la información del paciente observable (o). El texto contiene el nombre del paciente, su impacto Covid y sus síntomas (obtiene los síntomas del paciente). Finalmente, actualiza la etiqueta symptomLabel de la ventana.

Paso 3. Clase PacientSymptomGUI

Esta clase recoge a través de la pantalla ya mostrada, los síntomas de un paciente Covid19Pacient y lo actualiza. Los cambios a realizar son poner acciones en los listeners de ambos botones para añadir o eliminar un síntoma al objeto paciente, es decir:

```
p.addSymptomByName(((Symptom)symptomComboBox.getSelectedItem()).getName(),  
                    Integer.parseInt(weightField.getText()));  
  
p.removeSymptomByName(((Symptom)symptomComboBox.getSelectedItem()).getName());
```

Paso 4. Actualiza el programa principal

Para terminar, actualizaremos el programa principal para conectar todos los objetos. Ya está creado un objeto Covid19Pacient (model). A continuación, crearemos un objeto de la clase PacientObserverGUI pasándole como modelo a través de la constructora el objeto anterior, y para terminar crearemos la ventana PacientSymptomGUI para actualizar los síntomas del paciente:

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

Ejecuta el programa principal y comprueba que la aplicación funciona correctamente.

Se pide: cambia el programa principal para crear 2 pacientes Covid19Pacient con sus interfaces PacientSymptomGUI y PacientObserverGUI.

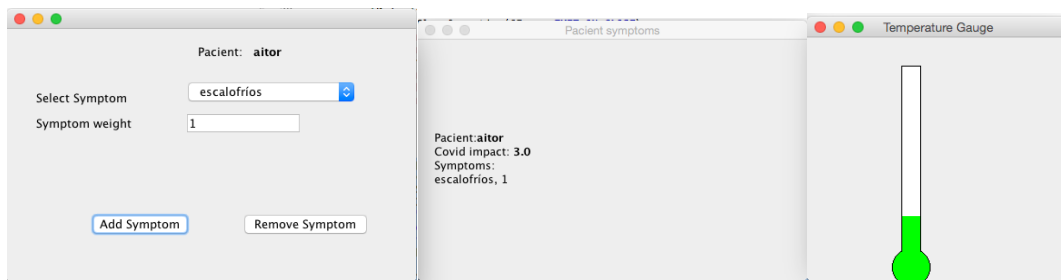
Segundo prototipo:

En este prototipo queremos lograr que cuando se agregue un síntoma, se actualice la interfaz anterior, pero también el gráfico del termómetro con color según el valor del covidImpact (si covidImpact <5 será verde, amarillo si 5 <= covidImpact <=10 y rojo si covidImpact > 10). A continuación, se muestran tres ejemplos:

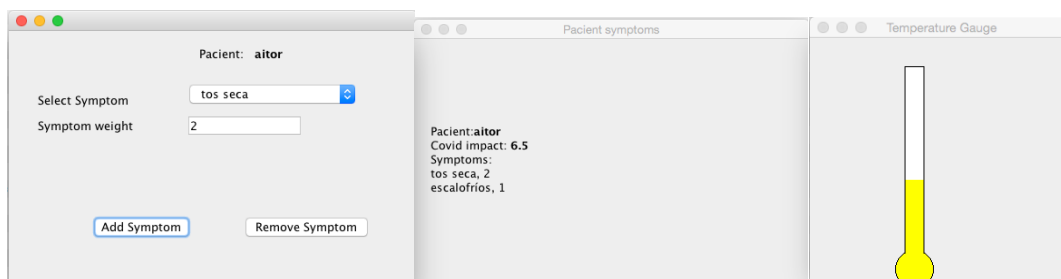
Ingeniería del Software II

LAB Tema 3- Diseño Avanzado: 3.2- Patrones de diseño

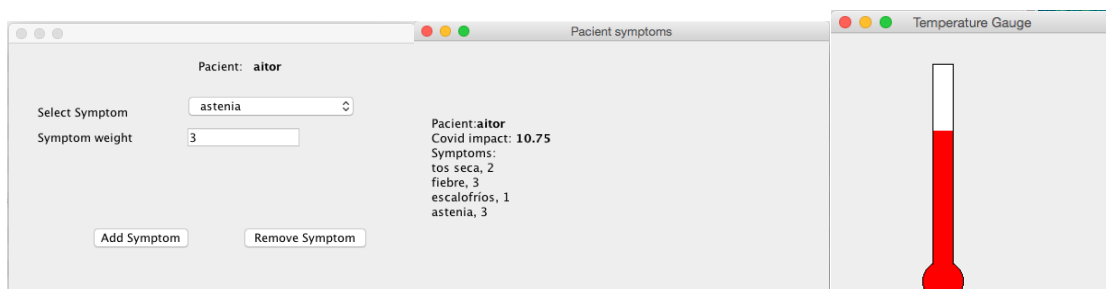
Primero le añadimos un síntoma (escalofríos, 1 pisuarekin), y como su covidImpact es 3, el termómetro aparece el verde.



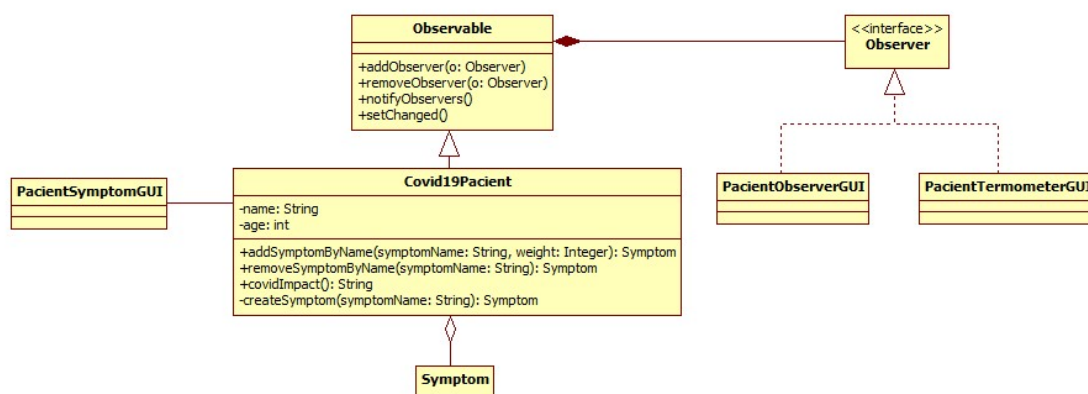
A continuación se añade otro síntoma (tos seca, 2). Ahora su covidImpact ha subido a 6, y por lo tanto su color ha pasado a amarillo:



Finalmente, añadimos (fiebre, 3) y (astenia, 3):



Para completar esta aplicación crearemos una nueva clase en la estructura del patrón Observer. Es de destacar que el patrón Observer nos permite extender la aplicación sin cambiar ninguna clase desarrollada previamente. Este es el nuevo diseño:

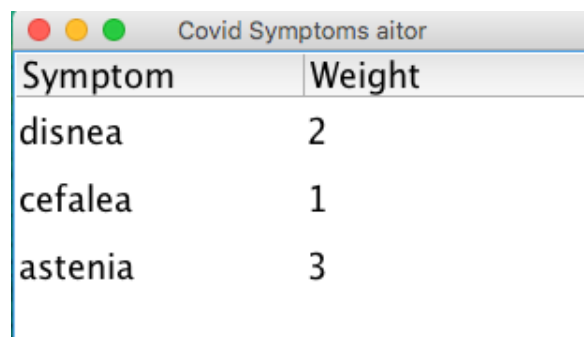


Solo hay que actualizar la clase PatientThermometerGUI. Con los pasos semejantes a los realizados para PatientObserverGUI y la implementación del método update es:

```
public void update(Observable o, Object args) {  
    Covid19Pacient p=(Covid19Pacient) o;  
    // Obtain the current covidImpact to paint  
    int farenheit = (int) p.calcCovid19Impact();  
    // temperature gauge update  
    gauges.set(farenheit);  
    gauges.repaint();  
}
```

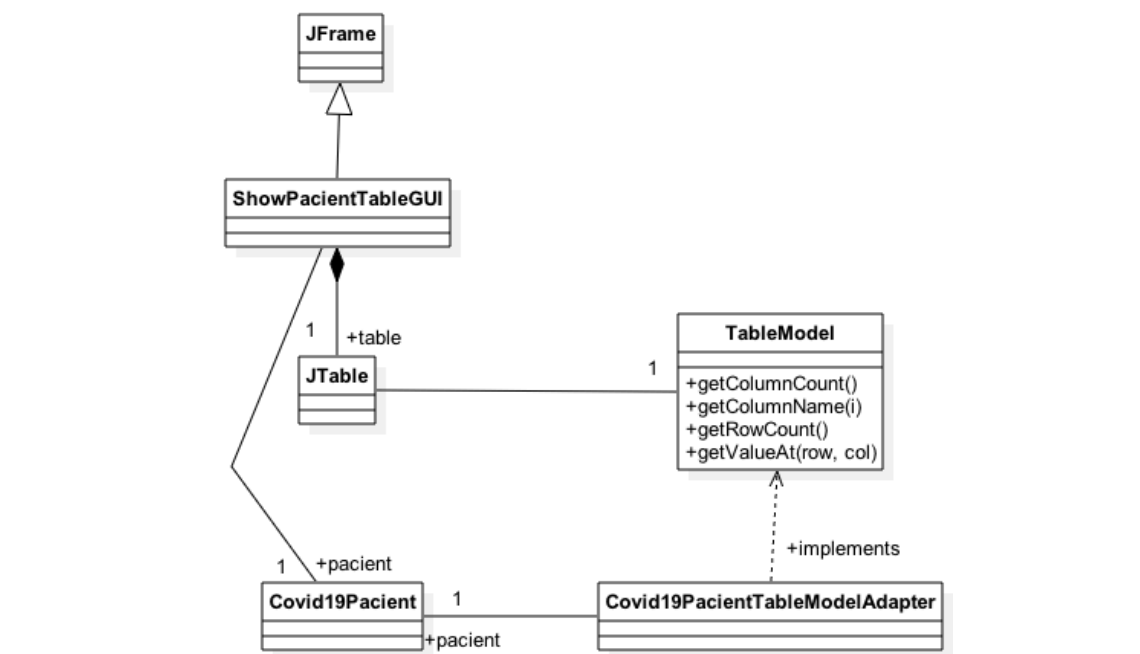
3.- Patrón Adapter

Queremos realizar una aplicación para visualizar los síntomas de un paciente en una tabla, tal y como se muestra en la siguiente figura:



Symptom	Weight
disnea	2
cefalea	1
astenia	3

Para realizar esta actividad utilizaremos el clase JTable de Swing. Esta clase visualiza en una tabla la información que contenga un objeto de tipo TableModel. En nuestro caso, lo que queremos visualizar es un objeto Covid19Pacient, por lo que lo tendremos que crear una clase adaptadora para adaptar un objeto de tipo Covid19Pacient a un objeto de tipo TableModel (Covid19PacientTableModelAdapter). El siguiente diagrama UML muestra la adaptación:



En el paquete adapter2, tienes todas las clases incompletas necesarias para ejecutar la aplicación.

Se pide:

- Añade el código necesario en la clase Covid19PatientTableModelAdapter y ejecuta la aplicación para comprobar que funciona correctamente.
- Añade otro paciente con otro síntomas, y ejecuta la aplicación para que aparezcan los 2 pacientes con sus síntomas.
- (opcional). 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.

4.- Patrón Iterator y Adapter

Nuestra clase Sorting tiene el método sortedIterator¹ que a partir de un objeto InvertedIterator y un objeto Comparator, devuelve un Iterator donde los elementos están ordenados en base al criterio establecido en el Comparator. Su implementación es la siguiente:

```
public interface InvertedIterator {  
    // return the actual element and go to the previous  
    public Object previous();  
  
    // true if there is a previous element  
    public boolean hasPrevious();  
  
    // It is placed in the last element  
    public void goLast();  
}
```

```
public class Sorting {  
    public static Iterator<Object> sortedIterator(InvertedIterator it,  
                                                  Comparator<Object> comparator) {  
        List<Object> list = new ArrayList();  
        it.goLast();  
        while (it.hasPrevious()) {  
            Symptom s= (Symptom)it.previous();  
            list.add(s);  
        }  
        Collections.sort(list, comparator);  
        return list.iterator();  
    }  
}
```

¹ Método inspirado de <http://stackoverflow.com/questions/16434526/sort-an-iterator-of-strings>. último acceso Octubre 2022.

Donde Iterator y Comparator son dos interfaces definidas en la librería de java.util. Su definición es la siguiente:

Interface Iterator: public interface Iterator<E>

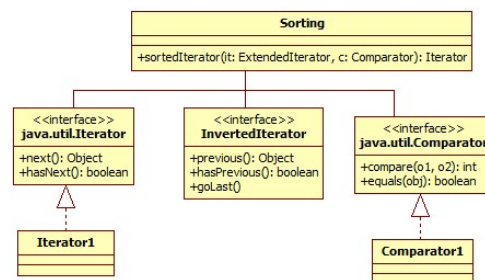
Modifier and Type	Method and Description
boolean	hasNext() Returns true if the iteration has more elements.
E	next() Returns the next element in the iteration.

Interface Comparator: public interface Comparator<T>

Modifier and Type	Method and Description
int	compare(T o1, T o2) Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.
boolean	equals(Object obj) Indicates whether some other object is "equal to" this comparator.

La clase Sorting se implementa usando el patrón de diseño Strategy. Esto es, para recorrer los elementos en el método sortedIterator, se usa como estrategia una implementación de InvertedIterator, y de igual manera los ordena según una estrategia que implementa Comparator.

De esta forma, al definir una clase que implemente la interfaz Comparator que ofrece java.util y usar el iterador InvertedIterator para recorrer sus elementos, podemos llamar al método sortedIterator.



Se pide:

Crea un programa principal utilizando el método Sorting.sortedIterator que imprima los 5 síntomas que debe tener un paciente Covid19Pacient. Se imprimirá primero ordenando por symptomName y luego por severityIndex.

- Crea un paciente Covid19Pacient con cinco síntomas. La clase Covid19Pacient NO PUEDE CAMBIARSE NADA.
- Implementa las interfaces Comparator: una para la ordenación por symptomName, y otra para la ordenación según severityIndex.
- Crea el patrón adapter sobre la clase Covid19Pacient, implementando la interfaz InvertedIterator. Recuerda crear una constructora adecuada para enviarle la información del paciente.
- TAMPOCO SE PUEDE MODIFICAR Sorting.sortedIterator.

