Laboratorio Patrones de diseño

Ingeniería de software II

Alejandro Sanchez Hernandez

Proyecto Github:

1) Factory

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

Si aparece un nuevo síntoma se tendría que modificar el método createSymtom de la clase Covid19Pacient lo cual rompe el principio ocp

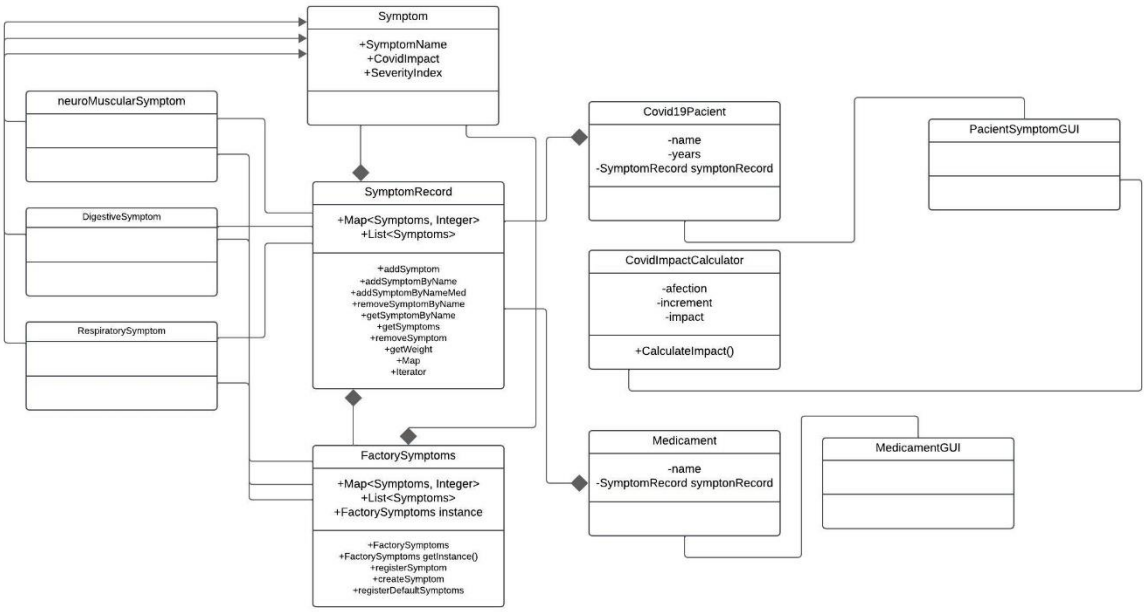
B. ¿Cómo se puede crear un nuevo síntoma sin cambiar las clases existentes (principio OCP)?

En la primera instancia del codigo esto no se puede hacer ya que toca modificar directamente la clase Covid19Pacient

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

En primera instancia la clase tiene 4 responsabilidades, las cuales son, calcular el impacto de los síntomas, administrar el usuario, creacion de síntomas y administrar la logica de los diferentes síntomas.

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.



Se crean tres clases nuevas FactorySymptoms, SymptomsRecord, CovidImpactCalculator.
FactorySymptoms: es el encargado de la instaciacion de los nuevos síntomas que se crean, como también permite agregar por medio de un método nuevos síntomas al sistema y también hace uso de una única instanciación para cada síntoma usando singleton lo cual evita la repetición de instancias de un mismo síntoma

SymptomsRecord: esta clase es la encargada de la administración de las funciones de síntomas, la cual permite crear, agregar, eliminar y consultar las diferentes características de los síntomas por medio de los métodos en su interior y extrayendo esta responsabilidad de la clase Covid19Pacient

CovidImpactCalculator: esta clase es la encargada de realizar el calculo de el impacto final dependiendo de los síntomas y el peso que tenga el paciente, esta clase se extrajo de la clase Covid19Impact aislando la responsabilidad del calculo del impacto del paciente

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

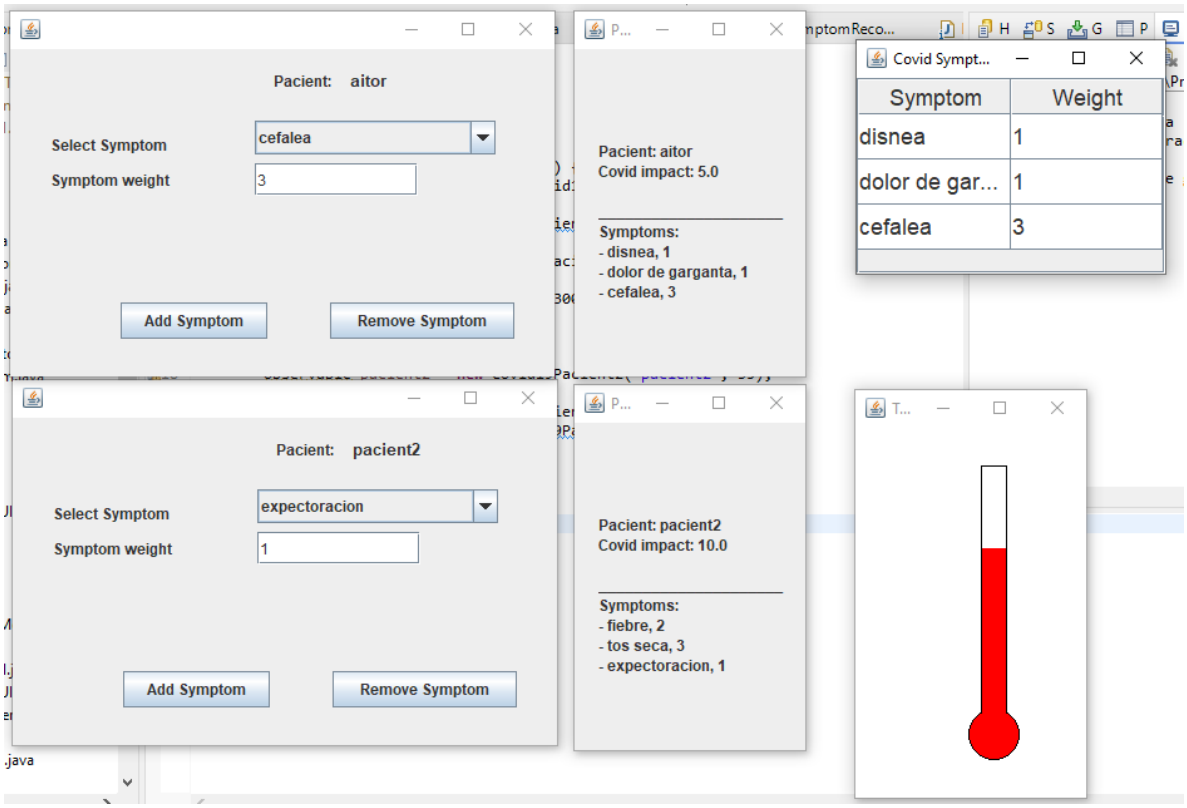
```
FactorySymptoms factory = FactorySymptoms.getInstance();
factory.registerSymptom("mareos", 2, 1, DigestiveSymptom.class);
```

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, haya únicamente x objetos Symptom)

Para hacer esto en la clase factory se uso una instancia singleton lo cual permite una única creacion para cada objeto.

2) Observer

```
public class Main {
    public static void main(String args[]) {
        Covid19Pacient2 pacient = new Covid19Pacient2("aitor", 35);
        new PacientObserverGUI(pacient);
        new PacientSymptomGUI((Covid19Pacient2) pacient);

        ShowPacientTableGUI gui=new ShowPacientTableGUI(pacient);
        gui.setPreferredSize(
                  new java.awt.Dimension(300, 200));
        gui.setVisible(true);


        Observable pacient2 = new Covid19Pacient2("pacient2", 35);
        new PacientObserverGUI(pacient2);
        new PacientSymptomGUI((Covid19Pacient2) pacient2);
        new PacientThermometerGUI((Covid19Pacient2) pacient2);
```



Aquí me adelanto y muestro la implementación de la tabla ShowPacientTableGUI del siguiente patron adapter el cual deja como opción implementarla también para el patron observer

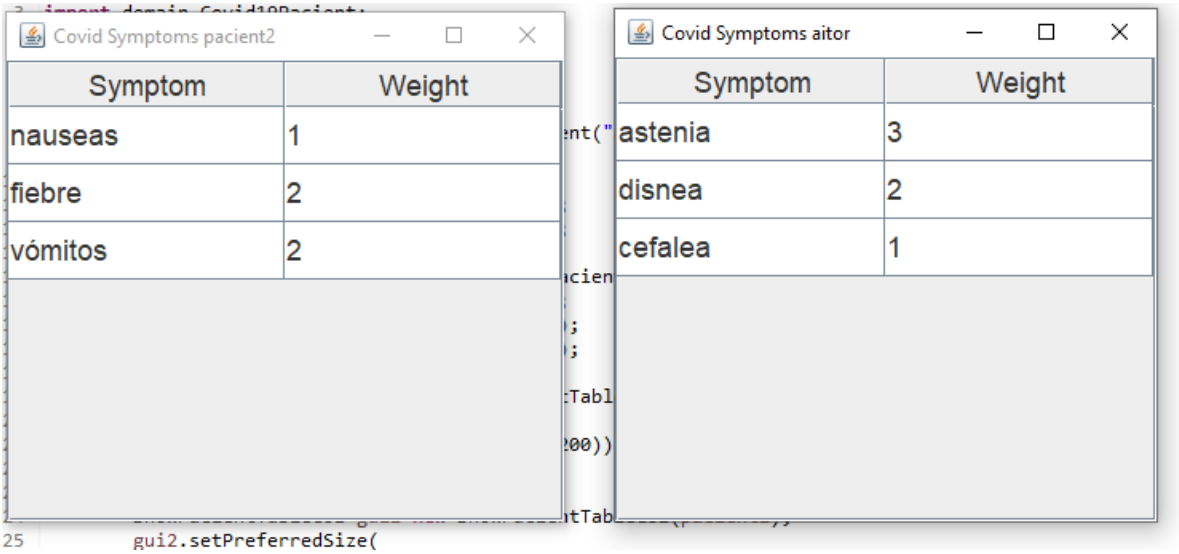
3) Adapter

Añade el código necesario en la clase Covid19PacientTableModelAdapter y ejecuta la aplicación para comprobar que funciona correctamente.

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

@Override
public int getColumnCount() {
    return columnNames.length; // Dos columnas: "Symptom" y "Weight"
}

@Override
public String getColumnName(int i) {
    return columnNames[i]; // Nombre de la columna según el índice
}

@Override
public int getRowCount() {
    return symptoms.size(); // Número de filas es el número de síntomas
}

@Override
public Object getValueAt(int row, int col) {
    Symptom symptom = symptoms.get(row); // Obtener el síntoma de la fila actual
    switch (col) {
        case 0:
            return symptom.getName(); // Nombre del síntoma
        case 1:
            return pacient.getWeight(symptom); // Peso del síntoma obtenido del paciente
        default:
            return null;
    }
}
```

Añade otro paciente con otro síntomas, y ejecuta la aplicación para que aparezcan los 2 pacientes son sus síntomas.

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

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

    Covid19Pacient pacient2 = new Covid19Pacient("pacient2", 30);
    pacient2.addSymptomByName("fiebre", 2);
    pacient2.addSymptomByName("nauseas", 1);
    pacient2.addSymptomByName("vómitos", 2);

    ShowPacientTableGUI gui=new ShowPacientTableGUI(pacient);
    gui.setPreferredSize(
              new java.awt.Dimension(300, 200));
    gui.setVisible(true);

    ShowPacientTableGUI gui2=new ShowPacientTableGUI(pacient2);
    gui2.setPreferredSize(
              new java.awt.Dimension(300, 200));
    gui2.setVisible(true);

}
```



(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.

```
public class Covid19PacientTableModelAdapter extends AbstractTableModel implements Observer {
    protected Covid19Pacient2 pacient;
    protected String[] columnNames = new String[] {"Symptom", "Weight"};
    private List<Symptom> symptoms;


    public Covid19PacientTableModelAdapter(Covid19Pacient2 p) {
        this.pacient = p;
        this.symptoms = new ArrayList<>(pacient.getSymptoms());
        pacient.addObserver(this);
    }

  @Override
  public void update(Observable o, Object arg) {
       symptoms = new ArrayList<>(pacient.getSymptoms()); // Actualiza la lista de síntomas
          fireTableDataChanged();

  }
```

Se muestra el resultado en el punto anterior

4) Iterator y Adapter

Crea un paciente Covid19Pacient con cinco síntomas. La clase Covid19Pacient NO PUEDE CAMBIARSE NADA.

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

    Covid19Pacient pacient = new Covid19Pacient("John Doe", 30);
    pacient.addSymptom(new Symptom("Cough", 5, 3), 3);
    pacient.addSymptom(new Symptom("Fever", 7, 4), 4);
    pacient.addSymptom(new Symptom("Fatigue", 3, 2), 2);
    pacient.addSymptom(new Symptom("Headache", 2, 1), 1);
    pacient.addSymptom(new Symptom("Loss of Smell", 6, 5), 5);

    // Crear el adaptador del paciente
    InvertedIterator pacientAdapter = new Covid19PacientAdapter(pacient);

    // Ordenar e imprimir por nombre de síntoma
    System.out.println("Ordenado por symptomName:");
    Iterator<Object> sortedByName = Sorting.sortedIterator(pacientAdapter, new ComparatorSymptomName());
    sortedByName.forEachRemaining(System.out::println);

    // Ordenar e imprimir por índice de severidad
    System.out.println("\nOrdenado por severityIndex:");
    Iterator<Object> sortedBySeverity = Sorting.sortedIterator(pacientAdapter, new ComparatorSeverityIndex());
    sortedBySeverity.forEachRemaining(System.out::println);
}
```

Implementa las interfaces Comparator: una para la ordenación por symptomName, y otra para la ordenación según severityIndex.

```
public class ComparatorSymptomName implements Comparator<Object> {
    @Override
    public int compare(Object o1, Object o2) {
        Symptom s1 = (Symptom) o1;
        Symptom s2 = (Symptom) o2;
        return s1.getName().compareTo(s2.getName());
    }

public class ComparatorSeverityIndex implements Comparator<Object> {
    @Override
    public int compare(Object o1, Object o2) {
        Symptom s1 = (Symptom) o1;
        Symptom s2 = (Symptom) o2;
        return Integer.compare(s1.getSeverityIndex(), s2.getSeverityIndex());
    }
```

Crea el patrón adapter sobre la clase Covid19Pacient, implementando la interfaz InvertedIterator. Recuerda crear una constructora adecuada para enviarle la información del paciente.

```java
public class Covid19PacientAdapter implements InvertedIterator {
    private List<Symptom> symptomsList;
    private int currentIndex;

    public Covid19PacientAdapter(Covid19Pacient pacient) {
        this.symptomsList = new ArrayList<>(pacient.getSymptomsMap().keySet());
        goLast();
    }

    @Override
    public void goLast() {
        currentIndex = symptomsList.size() - 1;
    }

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

    @Override
    public Object previous() {
        return symptomsList.get(currentIndex--);
    }
```