

Laboratorio Patrones de diseño

(Mikel Calleja)

Links: <https://github.com/kroko8/labpatterns.git>

1. - Simple Factory

Preguntas:

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

-Solo hay que agregar su información en la fábrica (SymptomFactory) para que pueda ser creado sin cambiar las clases existentes.

B: ¿Cómo se puede crear un nuevo síntoma sin cambiar las clases existentes (principioOCP)?

- Extendiendo SymptomFactory para que reconozca el nuevo síntoma sin modificar las clases actuales.

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

- La clase Covid19Pacient tiene la responsabilidad de gestionar los síntomas y calcular el impacto de COVID en el paciente. La clase Medicament se encarga de gestionar los medicamentos y los síntomas que pueden tratar.

Cambios / Implementaciones:

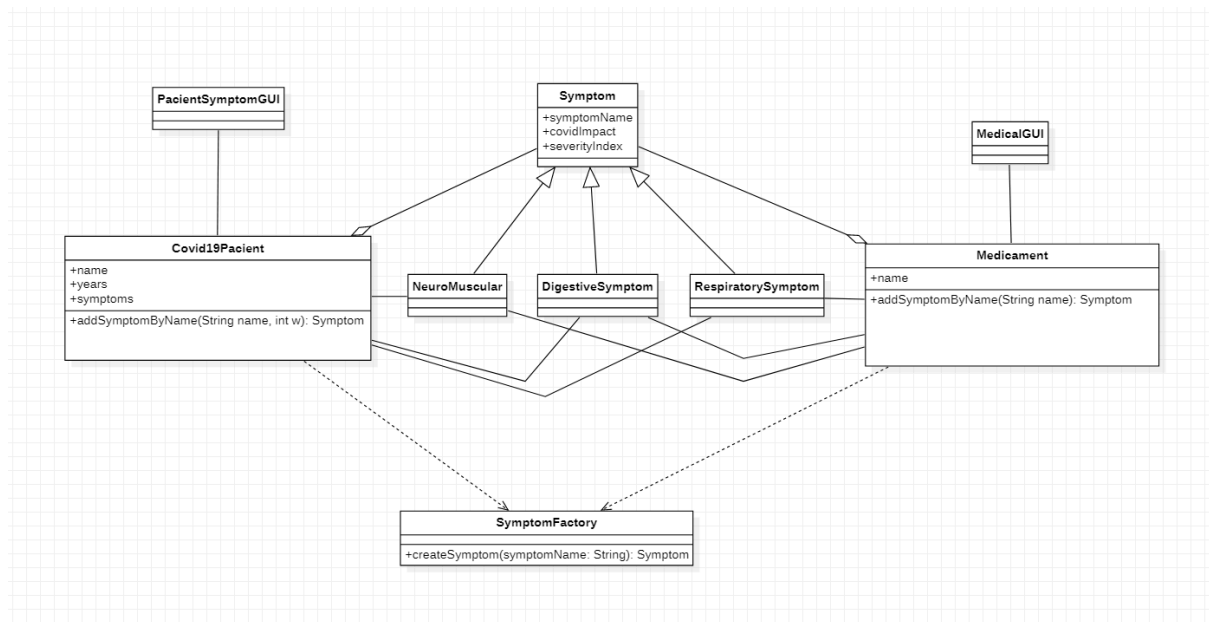
Cree la clase que funcionara como factory, llamada SymptomFactory

```
public class SymptomFactory {
    private static final Map<String, Symptom> symptomCache = new HashMap<>();
    public static Symptom createSymptom(String symptomName) {
        if (symptomCache.containsKey(symptomName)) {
            return symptomCache.get(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", "mareos");
        List<Double> index1 = Arrays.asList(5.0, 4.8, 3.7, 0.9, 0.8, 0.7, 0.5);
        List<String> digestiveSymptom = Arrays.asList("nauseas", "vómitos", "diarrea");
        List<String> neuroMuscularSymptom = Arrays.asList("fiebre", "astenia", "cefalea", "mialgia",
"escalofrios");
        List<String> respiratorySymptom = Arrays.asList("tos seca", "expectoracion", "disnea", "dolor de
garganta", "congestión nasal", "hemoptisis", "congestion conjuntival");
        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));
        }
        Symptom symptom = null;
        if (impact != 0) {
            if (digestiveSymptom.contains(symptomName)) {
                symptom = new DigestiveSymptom(symptomName, (int) index, impact);
            } else if (neuroMuscularSymptom.contains(symptomName)) {
                symptom = new NeuroMuscularSymptom(symptomName, (int) index, impact);
            } else if (respiratorySymptom.contains(symptomName)) {
                symptom = new RespiratorySymptom(symptomName, (int) index, impact);
            } else {
                symptom = new Symptom(symptomName, (int) index, impact); // Para cualquier síntoma
que no esté en categorías específicas
            }
        }
        if (symptom != null) {
            symptomCache.put(symptomName, symptom);
        }
        return symptom;}}
}
```

Básicamente se encarga de instanciar diferentes tipos de síntomas como DigestiveSymptom o NeuroMuscularSymptom. Posteriormente modifique las clases de dominio Covid19Pacient y Medicament en ellas borre la creación de síntomas y modifique los metodos addSymptomByName(String symptomName, int weight) y addSymptomByName(String symptomName) para que usase SymptomFactory.

```
public Symptom addSymptomByName(String symptomName, Integer weight) {
    Symptom symptom = getSymptomByName(symptomName);
    if (symptom == null) {
        symptom = SymptomFactory.createSymptom(symptomName);
        if (symptom != null) {
            symptoms.put(symptom, weight);
        }
    }
    return symptom;
}
```

```
public Symptom addSymptomByName(String symptomName) {
    Symptom symptom = getSymptomByName(symptomName);
    if (symptom == null) {
        symptom = SymptomFactory.createSymptom(symptomName);
        if (symptom != null) {
            symptoms.add(symptom);
        }
    }
    return symptom;
}
```



2. - Patrón Observer

Cambios / Implementaciones:

Cree una interfaz Observer, que define el método update(), que deben implementar todas las clases que actúen como observadores.

```
public interface Observer {  
    void update();  
}
```

También modifique la clase de semáforo y PatientObserverGUI que implementen la interfaz observer

En PatientObserverGUI añada el método de update:

```
@Override  
public void update() {  
    StringBuilder sb = new StringBuilder("<html>Symptoms:<br>");  
    Set<Symptom> symptoms = pacient.getSymptoms();  
    for (Symptom symptom : symptoms) {  
        sb.append("- ").append(symptom.getName()).append("<br>");  
    }  
    sb.append("</html>");  
    symptomLabel.setText(sb.toString());  
}
```

Y en la de semaphoreGUI también añada el método update y un método para actualizar los colores

```
@Override  
public void update() {  
    actualizarColor();  
}  
private void actualizarColor() {  
    double impact = pacient.covidImpact();  
    System.out.println("Impacto actual: " + impact); // Depurar  
    Color color;  
    if (impact < 5) {  
        color = Color.green;  
    } else if (impact < 10) {  
        color = Color.yellow;  
    } else {  
        color = Color.red;  
    }  
    getContentPane().setBackground(color);  
    repaint();  
}
```

Y por último modifique Covid19Pacient, en general añadi 3 métodos y un nuevo atributo

```
private List<Observer> observers = new ArrayList<>();
```

Los métodos addObserver(Observer observer) y removeObserver(Observer observer) los añadí para gestionar la lista de observadores del paciente.

```
public void addObserver(Observer observer) {  
    observers.add(observer);  
}  
public void removeObserver(Observer observer) {  
    observers.remove(observer);  
}
```

Y el último método que añadí fue el de notify() , recorre la lista de observadores y llama a update() en cada uno de ellos.

```
private void notifyObservers() {  
    for (Observer observer : observers) {  
        observer.update();  
    }  
}
```

Además en métodos como addSymptom y removeSymptom añadi la llamada a notifyObservers()

```
public void addSymptom(Symptom symptom, Integer weight) {  
    symptoms.put(symptom, weight);  
    notifyObservers();  
}
```

Con esta implementación de patrón Observer, todas las interfaces gráficas que muestran información sobre el paciente se actualizan automáticamente en tiempo real.

3. - Patrón Adapter

Cambios / Implementaciones:

Cree una nueva clase Covid19PacientInvertedIteratorAdapter, funciona como un Adapter que convierte los datos de Covid19Pacient a un formato compatible con JTable.

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

    protected List<Symptom> symptoms;
    public Covid19PacientTableModelAdapter(Covid19Pacient p) {
        this.pacient = p;
        this.symptoms = new ArrayList<>(p.getSymptoms());
    }
    @Override
    public int getColumnCount() {
        return columnNames.length;
    }
    @Override
    public String getColumnName(int col) {
        return columnNames[col];
    }
    @Override
    public int getRowCount() {
        return symptoms.size();
    }
    @Override
    public Object getValueAt(int row, int col) {
        Symptom symptom = symptoms.get(row);
        switch (col) {
            case 0:
                return symptom.getName();
            case 1:
                return pacient.getWeight(symptom);
            default:
                return null;
        }
    }
}
```

Y el main lo implemente con 2 usuario y 2 interfaces de la siguiente manera:

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

        //Paciente 1

        Covid19Pacient pacient=new Covid19Pacient("aitor", 35);

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

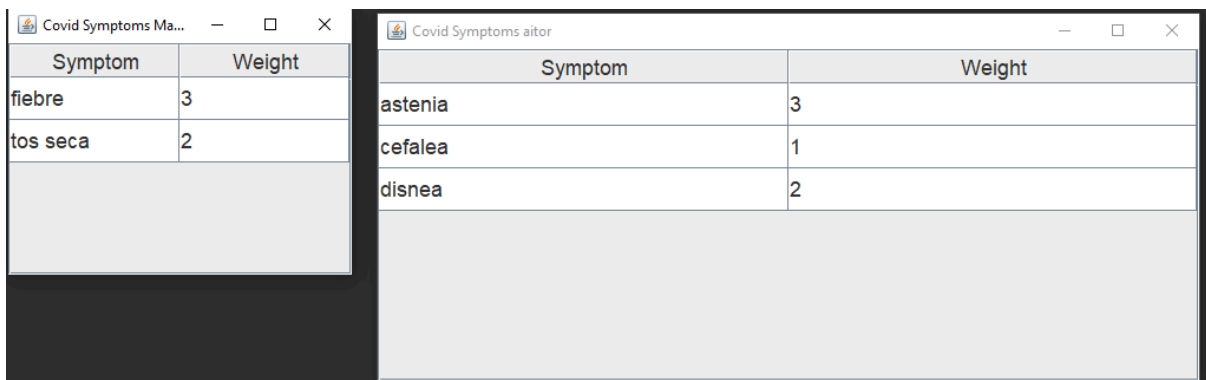
        ShowPacientTableGUI gui=new ShowPacientTableGUI(pacient);
        gui.setPreferredSize(
            new java.awt.Dimension(300, 200));
        gui.setVisible(true);

        //Paciente 2

        Covid19Pacient pacient2 = new Covid19Pacient("Maria", 42);
        pacient2.addSymptomByName("fiebre", 3);
        pacient2.addSymptomByName("tos seca", 2);

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

Resultados:



Symptom	Weight
fiebre	3
tos seca	2

Symptom	Weight
astenia	3
cefalea	1
disnea	2

4. - Patrón Iterator y Adapter

Cambios / Implementaciones:

Cree la clase Covid19PatientInvertedIteratorAdapter que adapta Covid19Patient para que funcione con InvertedIterator implementando la interfaz y sus métodos.

```
public class Covid19PatientInvertedIteratorAdapter implements InvertedIterator {
    private List<Symptom> symptoms;
    private int position;
    public Covid19PatientInvertedIteratorAdapter(Covid19Patient patient) {
        this.symptoms = new ArrayList<>(patient.getSymptoms());
        this.position = symptoms.size() - 1;
    }
    @Override
    public void goLast() {
        position = symptoms.size() - 1;
    }
    @Override
    public boolean hasPrevious() {
        return position >= 0;
    }
    @Override
    public Object previous() {
        Symptom symptom = symptoms.get(position);
        position--;
        return symptom;
    }
}
```

Y por último añadí 2 clases de comparadores SymptomNameComparator y SymptomSeverityComparator que ordenan los síntomas según el criterio.

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

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