ADSOF: Examen final. Convocatoria Ordinaria 28/05/2020 Versión 1 Continua

Ejercicio 2 (3 puntos)

Una aplicación médica gestiona *terapias*, todas ellas identificadas mediante un nombre único. Algunas terapias están basadas en una *receta* de un medicamento y otras consisten en un *tratamiento* formado por varias terapias. Cada receta de medicamento tiene, además de su nombre, el *número de dosis* y el *riesgo base* cuya *escala de riesgos* es: nulo, bajo, medio, alto y extremo. Los tratamientos se crean vacíos (solo con su nombre) para irles añadiendo terapias una a una, evitando añadir *de manera directa* dos terapias con igual nombre en un mismo tratamiento.

Debemos poder calcular el *riesgo real* de cada terapia. Para una receta con dosis 1, su riesgo real es igual a su riesgo base, pero si tiene dosis mayor que 1, el riesgo real es el inmediatamente superior al riesgo base en la escala de riesgos. Para un tratamiento, su riesgo real es el más alto entre todas las terapias incluidas en él.

Nota: puede ignorarse el control de errores por números negativos o sin sentido, strings vacíos o nulos, y estructuras cíclicas.

Se pide:

(a) **Diseñar e implementar en Java, el código necesario** para resolver los anteriores requisitos haciendo que el programa dado abajo **produzca la salida esperada.** Se valorará especialmente el uso de **principios de orientación a objetos en el diseño, así como su generalidad, reusabilidad, extensibilidad y la concisión y claridad** del código. [2,5 puntos]

(b) **Indicar qué patrón(es)** has usado en tu diseño, **identificando los roles que desempeñan las clases, métodos y atributos de tu diseño en cada patrón.** [0,5 puntos]

Salida esperada:

```
TR-1:[AAS(1,NULO), IBUPRO(1,ALTO)] riesgo real: ALTO
TR-2:[TR-1:[AAS(1,NULO), IBUPRO(1,ALTO)], ANFE(2,EXTREMO), AAS(1,NULO)] riesgo real: EXTREMO
```

```java
public class Ej2v1esCont {
  public static void main(String[] args) {
        Terapia te1 = new Receta("AAS", 1, Riesgo.NULO);
        Terapia te2 = new Receta("AAS", 3, Riesgo.BAJO);
        Terapia te3 = new Receta("IBUPRO", 1, Riesgo.ALTO);
        Terapia te4 = new Receta("ANFE", 2, Riesgo.ALTO); // riesgo real: EXTREMO
        Terapia tr1 = new Tratamiento("TR-1").add(te1)
                                             .add(te3)  // riesgo real: ALTO
                                             .add(te2); // no se añadirá
        Terapia tr2 = new Tratamiento("TR-2").add(tr1)
                                             .add(te4) // riesgo real: EXTREMO
                                             .add(tr1) // no se añade
                                             .add(te1); // sí se añade
        System.out.println( tr1 + " riesgo real: " + tr1.riesgoReal() );
        System.out.println( tr2 + " riesgo real: " + tr2.riesgoReal() );
  }
}
```

SOLUCIÓN Y PUNTUACIÓN, Ejercicio 2, Versión 1 Continua, 28 Mayo 2020, 3 puntos.

El reparto de puntos se refleja con la siguiente notación:

`[n] = valor aproximado sobre 30, a dividir por 10 para 3 puntos`

Además de las puntuaciones `[n]` asignadas a cada parte de la solución, se aplican penalizaciones por defectos relativos **principios de orientación a objetos en el diseño, así como su generalidad, reusabilidad, extensibilidad y la concisión y claridad,** como por ejemplo, código repetido innecesariamente, instrucciones if/else en cascada (o switch) para valores individuales de la enumeración, atributos no privados sin justificación válida, soluciones innecesariamente más complejas, etc.

Apartado (b): 0,5 puntos

`[1]` Se usa el patrón **Composite.**

`[2]` La clase **Therapy/Terapia** es la clase abstracta **Component** del patrón.
La clase **Drug/Receta** es la clase **Leaf** del patrón
La clase **Treatment/Tratamiento** es la clase **Composite** del patrón.

`[1]` El método actualRisk() es el método **operation**() del patrón.
El método add() es el método **add**() del patrón.

`[1]` El atributo **components** de Treatment es **children** en el patrón.

Apartado (a): 2,5 puntos

```
enum Risk {
      NULO, BAJO, MEDIO, ALTO, EXTREMO;  [1] // mejor sin valores internos
      public Risk nextHigher() {         [2] // metodo para obtener el siguiente
            return Risk.values()[ Math.min(Risk.values().length-1,  this.ordinal()+1) ];
      }
}

abstract class Therapy {  [1] // Terapia es la clase Component en el patrón Composite

      private String name;    [1] // atributo privado sin repetir en subclases
      public Therapy(String descr) { name = descr; }

      public Therapy add(Therapy t) { return this; }  [1] // add() en Component del patrón

      public abstract Risk riesgoReal(); [1] // operation() en Component del patrón

      @Override
      public final boolean equals(Object obj) { [2]
          return (obj instanceof Therapy) && this.name.equals( ((Therapy)obj).name );
      }
      @Override
      public final int hashCode() { return this.name.hashCode(); } [1]

    @Override public String toString() { return name; } [1]
}
```

```
class Treatment extends Therapy {// Tratamiento es la clase Composite en el patrón Composite
	private Set<Therapy> components = new LinkedHashSet<>();
	public Treatment(String descr) { super(descr); } [2]

	@Override
	public Therapy add(Therapy t) {
		this.components.add(t); [1]
		return this;            [1]
	}

	@Override
	public Risk riesgoReal() {//operation() implementado en Composite del patrón
		Risk resultado = Risk.values()[0]; [1] // mejor que Risk.NULO;
		for (Therapy t : components) {     [1]
			Risk aux = t.riesgoReal();
			if (aux.compareTo(resultado) > 0) [1]// mejor que comparar ordinal
				resultado = aux;
		}
		return resultado;
		/* O también en estilo funcional, pero sin olvidar orElse()
		return this.components.stream().map(Therapy::actualRisk)
		                               .max(Comparator.naturalOrder())
		                               .orElse(Risk.values()[0]); / mejor que Risk.NULO;
		*/
	}
	@Override
	public String toString() {    [1]
	       return super.toString() // mejor que getDescription() en superclase
	              + ":" + this.components.toString(); }
}

class Drug extends Therapy {  // Receta/Drug es la clase Leaf en el patrón Composite
	private int doses;
	private Risk baseRisk;
	public Drug(String descr, int dosis, Risk r) { // constructor principal 3 parámetros
		super(descr);  [2] // usar super() para tener atributos private en superclase
		this.doses = dosis; baseRisk = r;  [2]
	}

	@Override public Risk riesgoReal()  { // operation() implementado en Leaf del patrón
		return (this.doses == 1) ?  this.baseRisk : this.baseRisk.nextHigher(); [1]
	}

	@Override public String toString() {  [1]
	        return super.toString() // mejor que getDescription() en superclase
	               + "(" + this.doses + "," + this.riesgoReal()+")"; }
}
```