ADSOF: Examen final. Convocatoria Ordinaria 28/05/2020 Versión 2 Continua

Ejercicio 2 (3 puntos)

Una aplicación escolar gestiona *tareas evaluables*, todas ellas identificadas mediante un nombre único. Algunas tareas están basadas en una *pregunta* de opción múltiple y otras consisten en un *ejercicio* formado por varias tareas. Cada pregunta de opción múltiple tiene, además de su nombre, el *número de opciones* y la *dificultad temática* cuya *escala de dificultades* es: cero, baja, normal, alta y máxima. Los ejercicios se crean vacíos (solo con su nombre) para irles añadiendo tareas una a una, evitando añadir *de manera directa* dos tareas con igual nombre en un mismo ejercicio.

Debemos poder calcular la *dificultad real* de cada tarea. Para una pregunta con 2 opciones, su dificultad real es igual a su dificultad temática, pero si tiene más de 2 opciones, la dificultad real es la inmediatamente superior a su dificultad temática en la escala de dificultades. Para un ejercicio, su dificultad real es la más alta entre todas las tareas 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 requitsitos 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:

```
EJ-1:[Intro(2,CERO), T2.pr1(2,ALTA)] dificultad real: ALTA
EJ-2:[EJ-1:[Intro(2,CERO), T2.pr1(2,ALTA)], T2.pr2(3,MAXIMA), Intro(2,CERO)] dificultad real: MAXIMA
```

```java
public class Ej2v2esCont {
  public static void main(String[] args) {
        Tarea pr1 = new Pregunta("Intro", 2, Dificultad.CERO);
        Tarea pr2 = new Pregunta("Intro", 3, Dificultad.BAJA);
        Tarea pr3 = new Pregunta("T2.pr1", 2, Dificultad.ALTA);
        Tarea pr4 = new Pregunta("T2.pr2", 3, Dificultad.ALTA); // dificultad real: MAXIMA
        Tarea ej1 = new Ejercicio("EJ-1").add(pr1).add(pr3)  // dificultad real: ALTA
                                         .add(pr2); // no se añadirá
        Tarea ej2 = new Ejercicio("EJ-2").add(ej1).add(pr4) // dificultad real: MAXIMA
                                         .add(ej1) // no se añade
                                         .add(pr1); // sí se añade
        System.out.println( ej1 + " dificultad real: " + ej1.dificultadReal() );
        System.out.println( ej2 + " dificultad real: " + ej2.dificultadReal() );
  }
}
```

SOLUCIÓN, Ejercicio 2, Versión 2 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 **Task/Tarea** es la clase abstracta **Component** del patrón.
La clase **Question/Pregunta** es la clase **Leaf** del patrón.
La clase **Exercise/Ejercicio** es la clase **Composite** del patrón.

`[1]` El método dificultadReal() 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 Difficulty {
   CERO, BAJA, NORMAL, ALTA, MAXIMA;  [1] // mejor sin valores internos
   public Difficulty nextHigher() {   [2] // metodo para obtener el siguiente
     return Difficulty.values()[ Math.min(Difficulty.values().length-1, this.ordinal()+1) ];
   }
}

abstract class Task {  [1] // Task/Tarea es la clase Component en el patrón Composite

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

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

       public abstract Difficulty dificultadReal(); [1] //operation() en Component del patrón

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

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

```
class Exercise extends Task {// Exercise/Ejercicio es la clase Composite en patrón Composite
	private Set<Task> components = new LinkedHashSet<>();

	public Exercise(String descr) { super(descr); }  [2]

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

	@Override
	public Difficulty dificultadReal() {//operation() implementado en Composite del patrón
		Difficulty resultado = Difficulty.values()[0]; [1] // mejor que Risk.CERO;
		for (Task t : components) {                    [1]
			Difficulty aux = t.dificultadReal();
			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.CERO;
		*/
	}

	@Override
	public String toString() {   [1]
		 return super.toString() // mejor que getDescription() en super clase
		            + ":" + this.components.toString(); }
}

class Question extends Task {  // Question/Pregunta es la clase Leaf en el patrón Composite

	private int numOptions;
	private Difficulty subjectDifficulty;

	public Question(String descr, int nOptions, Difficulty d) {// constructor 3 parámetros
		super(descr);    [2] // usar super() para tener atributos private en superclase
		this.numOptions = nOptions; subjectDifficulty = d; [2]
	}

	@Override
	public Difficulty dificultadReal() { // operation() implementado en Leaf patrón
	   return (this.numOptions == 2) ?
	                    this.subjectDifficulty : this.subjectDifficulty.nextHigher(); [1]
	}

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