EXÁMEN DE TEORÍA DISEÑO DEL SOFTWARE

Diciembre 2018

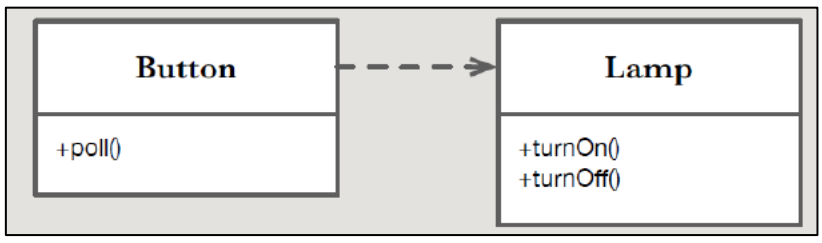
1. Una de las consecuencias del patrón **Strategy** es eliminar sentencias condicionales. Ejemplo de escenario de uso, diagrama UML después de aplicar el patrón identificando cada uno de sus componentes y ejemplo de código antes y después de aplicar el patrón. Ventajas de aplicarlo frente a la herencia estática mediante polimorfismo.

2. Diferencias entre el patrón **Decorator y Strategy**. Representar un diagrama de clases en tiempo de ejecución para ambos patrones.
   Diferencias entre el patrón **Decorator y Composit**e. Representar un diagrama UML identificando los participantes de cada patrón.

3. Tenemos un reproductor para cada tipo de elemento multimedia: canciones y videos. Aplicar un patrón para eliminar las sentencias condicionales. Identificar el patrón aplicado, estructura y componentes, cómo quedaría el código al eliminar las sentencias condicionales.

```
public class ListaDeReproduccion {
...
      public void reproducir() {
            Iterator<Elemento> iterador = elementos.iterator();
            while (iterator.hasNext()) {
                  Elemento elemento = iterator.next();
                  Reproductor reproductor;
                  if (elemento instanceof Video)
                        reproductor = new ReproductorDeVideo();
                  else if (elemento instanceof Cancion)
                        reproductor = new ReproductorDeAudio();
                  reproductor.reproducir(elemento);
            }
      }
...
}
```

4. Define el **principio de inversión de dependencias**. Aplícalo para mejorar el siguiente diseño en el que un botón permite encender o apagar una lámpara (representa cómo quedaría el diagrama de clases tras aplicarlo). ¿Qué se ha logrado?



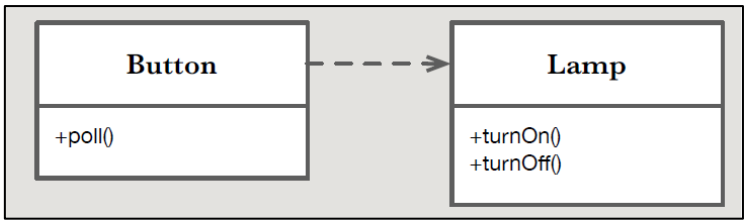
5. **Interfaces y Clases Abstractas**. Para qué se usan, su posición en la jerarquía e importancia de sus cambios.

EXÁMEN DE TEORÍA DISEÑO DEL SOFTWARE

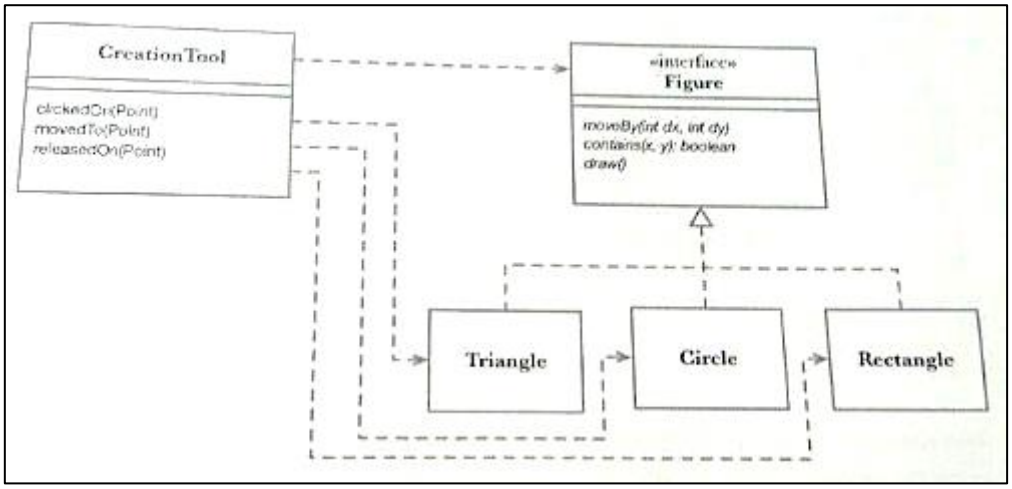
1. Tabla Interfaces vs. Clases Abstractas:

	Interfaz	Clase abstracta
Situación en que se originan		
Creador		
¿Raíz?		
Importancia		
Operaciones		
Importancia cambios		

2. Define el principio de inversión de dependencias. Aplícalo para mejorar el siguiente diseño en el que un botón permite encender o apagar una lámpara (representa cómo quedaría el diagrama de clases tras aplicarlo). ¿Qué se ha logrado?



3. A continuación, se representa el diseño actual de la parte de editor gráfico encargada de crear los distintos tipos de figuras, junto con un esbozo del código de la clase CreationTool. ¿Qué hay de malo en el código? Aplica un patrón que permita eliminar la lógica condicional a la hora de crear unas figuras u otras empleando polimorfismo.
   Representa un diagrama de clases con la solución y escribe cómo quedaría el código con el nuevo diseño (tanto la clase original como al menos el correspondiente a la creación de una figura determinada, así como la creación de los tres objetos que representan las distintas herramientas de creación). ¿De qué patrón se trata? ¿Qué se ha conseguido al aplicarlo?
   Haz lo mismo con el patrón Prototype.



   Esbozo de la clase actual CreationTool, donde reside el código encargado de crear las distintas figuras al soltar el botón en un punto dado:

```
public class CreationTool {
      public enum FigureType {RECTANGLE, CIRCLE, TRIANGLE}
      private FigureType type;
      private Editor editor;
      private Point initialPoint;

      public void releasedOn(Point endPoint) {
            Figure newFigure;
            if(type == FigureType.RECTANGLE)
                  newFigure = new Rectangle
                  (initialPoint,endPoint);
            else if(type == FigureType.CIRCLE) {
                  newFigure = new Circle
                  (initialPoint,endPoint);
            ... TRIANGLE ...
            editor.getDrawing().addFigure(newFigure);
            editor.toolDone();
      }
}
```

Mientras que la creación de los tres objetos representando las distintas herramientas de creación, en esta primera versión, sería algo así como:

```
CreationTool rectangleTool = new CreationTool(CreationTool.
      FigureType.RECTANGLE);
CreationTool circleTool = new CreationTool(CreationTool.
      FigureType.CIRCLE);
CreationTool triangleTool = new CreationTool(CreationTool.
      FigureType.TRIANGLE);
```

4. Define el **patrón Strategy**, su intención, y representa su estructura con un diagrama de clases UML donde aparezcan todos los participantes y métodos relevantes. ¿Cuándo es aplicable y cuáles son sus consecuencias? ¿En qué se diferencia del State? ¿Y del Decorator?

5. **Patrón Observer**: cuál es su intención, cuándo es aplicable, qué se consigue al aplicarlo. Representa la estructura del patrón mediante un diagrama UML y explica el papel de los distintos participantes y los métodos relevantes del patrón. Igualmente, dibuja un diagrama de secuencia donde se vea un escenario de uso típico del patrón en tiempo de ejecución, representando todas las colaboraciones entre los objetos participantes.