Orientación a Objetos 2 – Práctica 3

Ejercicio 1: Patrón Adapter

Lea el capítulo 4 del libro Design Patterns de Gamma et al. y responda a las siguientes preguntas:

1. ¿Qué es un patrón estructural?
2. El capítulo menciona dos formas de implementar el patrón Adapter: como patrón estructural de clase y como patrón estructural de objeto. ¿Cuáles son esas dos formas? ¿Cuál de ellas no es implementable en Smalltalk? ¿Por qué?

Ejercicio 2: Adapter

Ud. ha implementado un exitoso cliente desktop que permite publicar en Facebook su estado de ánimo.

En su sistema la clase `Facebook` implementa el mensaje `#post:`, donde recibe un string (sin límite de longitud) para ser publicado.

Dada la popularidad de Twitter ahora ud. desea que su cliente también permita publicar en dicha red social. Lamentablemente se encuentra con 2 problemas:

- Los tweets son strings de a lo sumo 140 caracteres.
- La clase Twitter no entiende el mensaje `#post:` sino `#publish:`. También recibe como parámetro un String.

Además, como su software funciona correctamente ud. desea introducir el menor número posible de modificaciones. A la vez, la clase Twitter pertence a una libría de terceros y no debería modificarla.

Tareas:

1. Diseñe la aplicación original.
2. Discuta con el ayudante la inclusión del soporte a Twitter
3. Explique cómo el patrón resuelve los problemas planteados.
4. Implemente en Smalltalk.

Ejercicio 3: Adaptadores de recorrido

A diferencia del paradigma estructurado, en la programación orientada a objetos las estructuras como los árboles o los grafos no se suelen modelar explícitamente, sino que surgen como resultado de las relaciones de conocimiento entre objetos. A modo de ejemplo, tomemos el caso de un *File System*. En VisualWorks un *File System* es accedido por medio de la clase `Filename`, que implementa el mensaje `#directoryContents`. Este mensaje retorna una colección

con los nombres de archivos y subdirectorios del objeto receptor. Luego, a pesar de que un *File System* tiene una estructura de árbol, no es necesario modelar una clase "Arbol" ya que la estructura se deriva de las relaciones entre los objetos. En este ejercicio usted deberá implementar la clase `TreeCollector`. Un `TreeCollector` se utiliza para recorrer cualquier red de objetos que posea una estructura de árbol subyacente. Para esto un `TreeCollector` se instancia con el mensaje `#block:aBlock` y reconoce los mensajes: `#preOrderCollect:aTree`, `#inOrderCollect:aTree` y `#postOrderCollect:aTree`. Cada uno de estos mensajes recorre la estructura de árbol (siguiendo los algoritmos correspondientes de pre-order, in-order y post-order) y al pasar por cada nodo evalúa el bloque con el nodo actual como parámetro. Cada mensaje retorna una colección, con el resultado de evaluar el bloque en cada nodo del árbol. Por ejemplo, el código para pre-order es algo similar a:

```
TreeCollector>>preOrderCollect: aTree
result:= OrderedCollection new.
self preOrderCollect:aTree with:result.
^result

TreeCollector>> preOrderCollect:aTree with:result
result add: (self block value: aTree).
aTree children do: [:subtree|
                    self preOrderCollect: subtree with:result] .
```

1. Implemente los test cases para la clase `TreeCollector` que verifiquen el buen funcionamiento del protocolo público. Para independizarse del caso concreto en el que el `TreeCollector` vaya a ser aplicado, cree una clase `MockTree` que entienda el mensaje `#children` retornando una colección (probablemente vacía) de árboles hijos y donde cada nodo tenga asociado un valor numérico. Esto último le permitirá testear el orden en el que se realizan los recorridos.
2. Implemente la clase `TreeCollector`.
3. Aplique el pattern Adapter para poder utilizar un `TreeCollector` sobre el árbol que representa la jerarquía de clases de Smalltalk, empezando por `Object`. Para obtener las subclases de una clase, se le envía el mensaje `#subclasses`. Por ejemplo, `Collection subclasses` retorna las subclases directas de la clase Collection.
4. Aplique el pattern adapter para poder utilizar un `TreeCollector` sobre la estructura de directorios a partir de c:/temp.

Ejercicio 4: Topografía

Un objeto Topografía representa la distribución de agua y tierra de una región cuadrada del planeta, la cual está formada por porciones de "agua" y de "tierra". La figura 1a muestra el aspecto de una topografía formada únicamente por agua, otra formada sólamente por tierra (fig. 1b) y una topografía mixta (fig. 1c).

Una topografía mixta está formada por partes de agua y partes de tierra (4 partes en total). Éstas a su vez podrían descomponerse en 4 más y así consecutivamente.

(a) Agua (b) Tierra (c) Topografía Mixta

Figura 1: Distintas topografías.

Como puede verse en los ejemplos, hay una relación muy estrecha entre la proporción de agua o tierra de una topografía mixta y la proporción de agua o tierra de sus componentes (compuestas o no). Por ejemplo:

- La proporción de agua de una topografía sólo agua (fig. 1a) es 1.
- La proporción de agua de una topografía sólo tierra (fig. 1b) es 0.
- La proporción de agua de una topografía compuesta está dada por la suma de la proporción de agua de sus componentes dividida por 4.

En la figura 1c, la proporción de agua es: $0 + 1/4 + 1/4 + 0 = 1/2$. La proporción siempre es un valor entre 0 y 1.

1. Implemente las clases necesarias para que sea posible:
   - crear Topografías,
   - calcular su proporción de agua y tierra,
   - comparar igualdad entre topografías (dada por igual proporción de agua y tierra e igual distribución).
2. Instancie la topografía compuesta del ejemplo y pruebe la funcionalidad implementada mediante *test cases*.

Nota: sólo se descomponen topografías para conseguir combinaciones. No es correcto construir una topografía compuesta por cuatro topografías del mismo tipo.

Ejercicio 5: Modele el comportamiento de un FileSystem

Un *file system* contiene un conjunto de directorios y archivos organizados jerárquicamente mediante una relación de inclusión. De cada archivo se conoce el nombre, fecha de creación y tamaño en bytes. De un directorio se conoce el nombre, fecha de creación y contenido (el tamaño es siempre 32kb). Modele el *file system* y provea la funcionalidad para consultar de un directorio:

Código 1: Interfaz a implementar

```
 #tamanoTotalOcupado
"Retorna el espacio total ocupado en KB, incluyendo su contenido."

#listadoDeContenido
"Imprime en el Transcript el listado del contenido del directorio siguiendo el
modelo
presentado a continuacion:
- Directorio A
--- Directorio A.1
------ Directorio A.1.1    3 archivos
------ Directorio A.1.2    2 archivos
--- Directorio A.2
- Directorio B "

#archivoMasGrande
"Retorna el archivo con mayor cantidad de bytes en cualquier nivel del
filesystem
contenido por directorio receptor"

#archivoMasNuevo
"retorna el archivo con fecha de creacion mas recienteen cualquier nivel del
filesystem
contenido por directorio receptor"
```

Tareas:

- Diseñe y represente un modelo UML de clases de su aplicacion, identifique el patrón de diseño empleado (utilice estereotipos UML para indicar los roles de cada una de las clases en ese patrón)
- Implemente completamente en Smalltalk
- Diseñe, implemente y ejecute test cases para verificar el funcionamiento de su aplicación

Ejercicio 6: Fórmulas con objetos.

Veremos que es sencillo modelar formulas aplicando los conocimientos de patrones. Considerando las siguientes reglas modele e implemente las fórmulas

- Las constantes son términos.
- Las variables son términos.
- Una función cuyos parámetros sean términos es un término.

A partir de una fórmula es necesario saber la cantidad de términos atómicos (constantes o variables).

Implemente completamente en Smalltalk incluyendo los test cases.