PRÁCTICA 3: LOS LADRONES DE CUERPOS[1]

1. Introducción

Desde el punto de vista de la calidad externa, los productos software creados mediante la utilización de *patrones de diseño* no se distinguen de los creados sin su utilización. Por ejemplo, supongamos que se creasen dos productos software con exactamente la misma funcionalidad, pero donde un producto se ha creado utilizando patrones de diseño y el otro producto sin ellos. En este caso, ambos productos ofrecerían tanto unas capacidades como un rendimiento similar. En adelante denominaremos al producto que utiliza los patrones de diseño como el producto *patronizado* y al que los utiliza como el producto *sin patronizar*.

Estos productos no serían sólo iguales desde un punto de visto externo, sino que incluso, en algunas situaciones muy particulares[2], el producto *patronizado* podría ser ligeramente más lento y/o consumir más memoria que el producto *sin patronizar*. Es decir, desde este punto de vist, el producto *patronizado* se puede considerar peor que el producto que *sin patronizar*.

Por tanto, si, desde el punto de vista de la calidad externa, ambos productos son prácticamente indistinguibles y la utilización de los patrones de diseño sólo parece servir para empeorar el producto, ¿cuál es entonces la ventaja aportada por su utilización?

La principal ventaja que suelen aportar los *Patrones de Diseño* está relacionada con la calidad interna de un producto software y, más concretamente, con su facilidad de evolución y adaptación al cambio. Para ello los patrones de diseño suelen introducir en un producto software *puntos de variación* bien definidos que permiten introducir nuevas funcionalidades en dicho producto de manera cómoda, sencilla y elegante, sin necesidad de modificar el producto ya existente.

Por tanto, volviendo a nuestro ejemplo, aunque externamente el producto *patronizado* pueda parecer exactamente igual que el producto *sin patronizar*, el producto patronizado es en realidad mucho más fácil de adaptar a nuevas situaciones y extender con nuevas funcionalidades. Es decir, en resumen, el producto *patronizado* es mucho más barato de mantener.

El primer objetivo de esta práctica es que el alumno entienda cómo y por qué los patrones de diseño ayudan a mejorar la calidad interna de un producto software. Para alcanzar este objetivo, el alumno deberá aplicar el patrón *Visitor* en una situación concreta. Se ha escogido el patrón *Visitor* por un ser un claro ejemplo de la situación descrita con anterioridad.

[1] En homenaje al clásico de la ciencia ficción "*La invasión de los ladrones de cuerpos*" (Don Siegel,1956), donde unas extrañas criaturas se introducen en los cuerpos de los habitantes de un pequeño pueblo estadounidense, los cuales mantienen su apariencia externa pero cambian su comportamiento.
[2] Dadas las características del hardware actual, este fenómeno es prácticamente inapreciable en casi la totalidad de situaciones.

Siguiendo los principios del patrón *Visitor* podemos conseguir que sea posible incorporar nuevas funcionalidades a una jerarquía o estructura concreta de clases, sin necesidad de tener que modificar dichas clases.

Sin embargo, para aplicar el patrón *Visitor* debemos utilizar de un complejo y exótico mecanismo de *double dispatching*. Este mecanismo puede ser complejo de entender, sobre todo por parte de programadores noveles poco familiarizados con este patrón. Además, el *double dispatching* introduce una serie de niveles de indirección que ralentizarían la ejecución de la aplicación, aunque este fenómeno es inapreciable con las prestaciones de las computadoras actuales.

Resumiendo, el objetivo de esta práctica es que el alumno comprenda parte de las ventajas e inconvenientes asociados a los patrones de diseño mediante la aplicación del patrón *Visitor*. Para alcanzar dicho objetivo, el alumno deberá satisfacer los subobjetivos que se detallan en la siguiente sección.

2. Objetivos

Los objetivos concretos de esta práctica son:

1. Comprender el funcionamiento del *patrón Visitor*.
2. Comprender el concepto de *punto de variación* y *variante*.
3. Ser capaz de, utilizando el patrón *Visitor*, introducir *puntos de variación* dentro de una estructura de clases de manera que se puedan incorporar de nuevas funcionalidades a dicha estructura.
4. Ser capaz de, utilizando el patrón *Visitor*, extender la funcionalidad de una estructura de clases *visitable* mediante la implementación de nuevas variantes.

Para alcanzar dichos objetivos, el alumno deberá aplicar el patrón *Visitor* en el contexto que se describe a continuación.

5. Visualización del Sistema de Archivos *Sparrow*

El sistema de archivos *Sparrow* (ver Práctica del Patrón Composite) debe permitir visualizar su estructura en forma arbórea conforme a, por el momento, dos formatos distintos: el formato *compacto* y el formato *extendido*.

En ambos formatos, sólo se debe mostrar un elemento por línea. Por cada elemento, además, se debe mostrar su nombre. Como es normal, cuando se muestre un directorio, deberá mostrarse debajo su contenido. Dicho contenido se deberá mostra tabulado con tres espacios con respecto al nombre del directorio.

Si se utiliza el formato compacto, el contenido de los archivos comprimidos no se visualizaría. Por el contrario, cuando se utiliza el formato desplegado, si se mostraría el contenido de un archivo comprimido. Al igual que en el caso de los directorios, este contenido estará tabulado con tres espacios con respecto al nombre del archivo comprimido.

Además, en el formato extendido, delante el nombre de cada elemento se mostrará una letra que especifique el tipo de elemento del que se trata. Como letra se utilizará la "d" para los directorios, la "e" para los enlaces directos, la "c" para los archivos comprimidos, y la "f" para los archivos.

La Figura 1 muestra un ejemplo de visualización para el formato desplegado.

```
d Raiz
   d Directorio Vacio
   d Directorio Con Archivo Unico
      f foto001.jpg
   d Directorio Con Archivo Comprimido Simple
      f foto002.jpg
      e foto001.jpg
      c ccSimple.zip
         d Directorio Vacio En Archivo Comprimido
         f foto003.jpg
         e foto001.jpg
   d Directorio con Directorio Anidado
      f foto004.jpg
      e ccSimple.zip
      e Directorio Vacio
      d Directorio con Archivo Comprimido Complejo
         f foto005
         f foto006
         c ccComplejo.zip
            c ccAnidada.zip
               f foto007.jpg
            f foto008.jpg
```

Figura 1. Ejemplo de visualización de Sistema de Archivos.

Para aplicar el patrón *Visitor* a este problema concreto, se deberán realizar las actividades que se describen en la siguiente sección.

3. Actividades

El alumno, para poder alcanzar los objetivos perseguidos, deberá completar las siguientes actividades:

1. Hacer que jerarquía de clases creada en la práctica dedicada al patrón *Composite* sea *visitable* por las funciones de impresión descritas. Para hacer que la jerarquía sea visitable, el alumno deberá crear un visitante abstracto y añadir a la jerarquía los métodos para aceptar visitantes, con su correspondiente implementación. Las funciones de impresión deberán serializar un sistema de archivos como una cadena de caracteres, de manera que dicha cadena se pueda imprimir en diferentes salidas, tales como un monitor o una impresora. Por tanto, las funciones de impresión o visualización deben retornar una cadena de caracteres.
2. Implementar un visitante concreto para el formato compacto, sin tener en cuenta la necesidad de tabular el contenido de los directorios.

3. Crear un programa de prueba que verifique el correcto funcionamiento del visitante implementado para el sistema de archivos de la Figura 1.
4. Implementar un visitante concreto para el formato extendido, sin tener en cuenta la necesidad de tabular el contenido de los directorios.
5. Crear un programa de prueba que verifique el correcto funcionamiento del visitante implementado para el sistema de archivos de la Figura 1.
6. Modificar los visitantes implementados para que tengan en cuenta el requisito relativo a la tabulación de los elementos contenidos en directorios o archivos comprimidos.
7. Con los programas de prueba implementados anteriormente, verificar el correcto funcionamiento de las modificaciones realizadas para soportar las tabulaciones.

Las actividades 1 a 5 representan las habilidades mínimas y básicas que el alumno debe adquirir para poder superar esta práctica. Las actividades 6 y 7 están enfocadas a aquellos alumnos que quieran profundizar un poco más en las características y ventajas del patrón *Visitor*.

Para aplicar el patrón Visitor, se recomienda seguir los pasos que se describen a continuación.

5. Pasos para Aplicar el Patrón *Visitor*

La siguiente sección describe, de manera resumida, una secuencia de pasos a realizar cuando se debe instanciar el patrón *Visitor*.

1. Identificar la familia de operaciones a introducir en la jerarquía. Identificar claramente el tipo del parámetro de retorno[3].
2. Crear una clase abstracta o interfaz que contenga tantos métodos *visit* como clases concretas haya en la jerarquía[4]. Estos métodos visit aceptan como argumento la clase concreta que corresponda y devuelve el tipo del parámetro de retorno indetificado.
3. Introducir en la jerarquía de clases un método *accept* que acepte un visitante abstracto y devuelva el tipo del parámetro de retorno identificado.
4. Implementar los métodos *accept* para cada clase concreta de la jerarquía. Los métodos *accept* deben simplemente delegar en el método *visit* que corresponda a la clase donde se encuentran.
5. Crear un visitante concreto por cada elemento de la familia de operaciones a implementar.
6. Implementar los métodos *visit* de cada visitante concreto. Para invocar otros métodos *visit,* delegar en los *accept* de la jerarquía de clases.

Pablo Sánchez Barreiro

[3] En el caso de que existan varios parámetros de retorno, se puede utilizar como parámetro de retorno una clase que guarde los resultados como atributos, o almacenar los resultados en el propio visitante abstracto.

[4] Si hubiese clases concretas, hijas de una misma clase padre, que comparten una implementación idéntica de los métodos *visit*, se puede implementar un solo método *visit* en dicha clase padre, común a todos los hijos.