TP 1 – POO y Patrones de Software – Patrones Estructurales 002 Parte 2

Alumno: Nuñez Souto, Aaron Agustín

1. Los patrones estructurales se clasifican de la siguiente manera:
2. **Adapter**: es un patrón de diseño estructural que permite la colaboración entre objetos con interfaces incompatibles. Éste utiliza el principio de composición de objetos: el adaptador implementa la interfaz de un objeto y envuelve el otro.
3. **Bridge**: es un patrón de diseño estructural que te permite dividir una clase grande, o un grupo de clases estrechamente relacionadas, en dos jerarquías separadas (abstracción e implementación) que pueden desarrollarse independientemente la una de la otra.
4. **Composite**: es un patrón de diseño estructural que te permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales.
5. **Decorator**: es un patrón de diseño estructural que te permite añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores que contienen estas funcionalidades.
6. **Fecade**: es un patrón de diseño estructural que proporciona una interfaz simplificada a una biblioteca, un framework o cualquier otro grupo complejo de clases.
7. **Flyweight**: es un patrón de diseño estructural que te permite mantener más objetos dentro de la cantidad disponible de RAM compartiendo las partes comunes del estado entre varios objetos en lugar de mantener toda la información en cada objeto.
8. **Proxy**: es un patrón de diseño estructural que te permite proporcionar un sustituto o marcador de posición para otro objeto. Un proxy controla el acceso al objeto original, permitiéndote hacer algo antes o después de que la solicitud llegue al objeto original.
9. Los patrones de diseño estructurales resuelven aquellos problemas que requieren de composición (agregación) de clases y objetos. El concepto de herencia se utiliza para componer interfaces y definir formas de componer objetos para obtener nuevas funcionalidades.
10. Ejemplo en imagen externa.
11. Un ejemplo de patrón composite se puede observar en el siguiente problema: imagina que tienes dos tipos de objetos: Productos y Cajas. Una Caja puede contener varios Productos así como cierto número de Cajas más pequeñas. Estas Cajas pequeñas también pueden contener algunos Productos o incluso Cajas más pequeñas, y así sucesivamente.

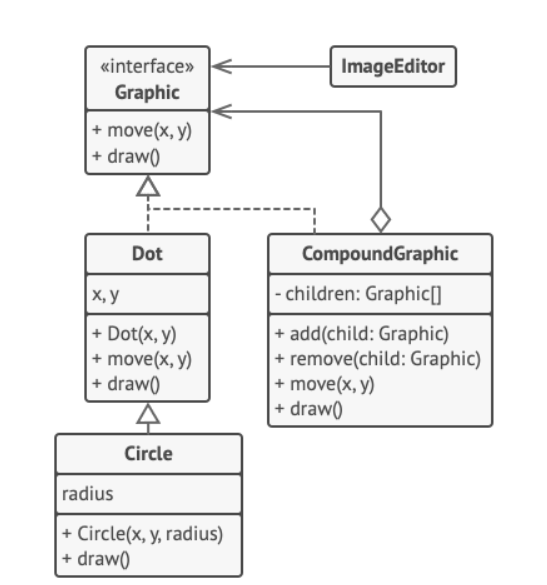
Digamos que decides crear un sistema de pedidos que utiliza estas clases. Los pedidos pueden contener productos sencillos sin envolver, así como cajas llenas de productos... y otras cajas. ¿Cómo determinarás el precio total de ese pedido?

Puedes intentar la solución directa: desenvolver todas las cajas, repasar todos los productos y calcular el total. Esto sería viable en el mundo real; pero en un programa no es tan fácil como ejecutar un bucle. Tienes que conocer de antemano las clases de Productos y Cajas a iterar, el nivel de anidación de las cajas y otros detalles desagradables. Todo esto provoca que la solución directa sea demasiado complicada, o incluso imposible.

La solución al problema, mediante el patrón Composite, es: El patrón Composite sugiere que trabajes con Productos y Cajas a través de una interfaz común que declara un método para calcular el precio total.

¿Cómo funcionaría este método? Para un producto, sencillamente devuelve el precio del producto. Para una caja, recorre cada artículo que contiene la caja, pregunta su precio y devuelve un total por la caja. Si uno de esos artículos fuera una caja más pequeña, esa caja también comenzaría a repasar su contenido y así sucesivamente, hasta que se calcule el precio de todos los componentes internos. Una caja podría incluso añadir costos adicionales al precio final, como costos de empaquetado.

La gran ventaja de esta solución es que no tienes que preocuparte por las clases concretas de los objetos que componen el árbol. No tienes que saber si un objeto es un producto simple o una sofisticada caja. Puedes tratarlos a todos por igual a través de la interfaz común. Cuando invocas un método, los propios objetos pasan la solicitud a lo largo del árbol.



1. Teniendo en cuenta que un mixin es un fragmento de una clase en el sentido de que su propósito es ser compuesto por otras clases o mixins, al querer llevarlo a la programación en C++ lo primero que hallamos es que este lenguaje no tiene soporte para los mixins, sino que para poder “hacerlos” hay que utilizar herencia múltiple, lo cual termina significando que en la práctica se sigue utilizando la herencia múltiple, pero artificialmente limitamos aquello que queremos permitir ser usado.
2. El programar con interfaces nos permite aumentar la modularidad de la aplicación, por lo tanto, también su capacidad de mantenimiento en ciclos de desarrollo posteriores. Esto se debe a que se puede definir una aplicación como una colección de módulos interacoplados, los cuales se pueden desconectar, reemplazar o actualizar, sin la necesidad de comprometer el contenido de los otros módulos; lo cual puede verse como una aplicación del principio de Open-Closed. A su vez, se reduce considerablemente la complejidad del sistema.