TP 1 – POO y Patrones de Software

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. Ejemplos de problemas según cada patrón:
11. **Adapter**:

Imagina que estás creando una aplicación de monitoreo del mercado de valores. La aplicación descarga la información de bolsa desde varias fuentes en formato XML para presentarla al usuario con bonitos gráficos y diagramas.

En cierto momento, decides mejorar la aplicación integrando una inteligente biblioteca de análisis de una tercera persona. Pero hay una trampa: la biblioteca de análisis solo funciona con datos en formato JSON.

Podrías cambiar la biblioteca para que funcione con XML. Sin embargo, esto podría descomponer parte del código existente que depende de la biblioteca. Y, lo que es peor, podrías no tener siquiera acceso al código fuente de la biblioteca, lo que hace imposible esta solución.

1. **Bridge**:

Digamos que tienes una clase geométrica Forma con un par de subclases: Círculo y Cuadrado. Deseas extender esta jerarquía de clase para que incorpore colores, por lo que planeas crear las subclases de forma Rojo y Azul. Sin embargo, como ya tienes dos subclases, tienes que crear cuatro combinaciones de clase, como CírculoAzul y CuadradoRojo.

Añadir nuevos tipos de forma y color a la jerarquía hará que ésta crezca exponencialmente. Por ejemplo, para añadir una forma de triángulo deberás introducir dos subclases, una para cada color. Y, después, para añadir un nuevo color habrá que crear tres subclases, una para cada tipo de forma. Cuanto más avancemos, peor será.

1. **Composite**:

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.

1. **Decorator**:

Imagina que estás trabajando en una biblioteca de notificaciones que permite a otros programas notificar a sus usuarios acerca de eventos importantes.

La versión inicial de la biblioteca se basaba en la clase Notificador que solo contaba con unos cuantos campos, un constructor y un único método send. El método podía aceptar un argumento de mensaje de un cliente y enviar el mensaje a una lista de correos electrónicos que se pasaban a la clase notificadora a través de su constructor. Una aplicación de un tercero que actuaba como cliente debía crear y configurar el objeto notificador una vez y después utilizarlo cada vez que sucediera algo importante.

En cierto momento te das cuenta de que los usuarios de la biblioteca esperan algo más que unas simples notificaciones por correo. A muchos de ellos les gustaría recibir mensajes SMS sobre asuntos importantes. Otros querrían recibir las notificaciones por Facebook y, por supuesto, a los usuarios corporativos les encantaría recibir notificaciones por Slack.

No puede ser muy complicado ¿verdad? Extendiste la clase Notificador y metiste los métodos adicionales de notificación dentro de nuevas subclases. Ahora el cliente debería instanciar la clase notificadora deseada y utilizarla para el resto de notificaciones.

Pero entonces alguien te hace una pregunta razonable: “¿Por qué no se pueden utilizar varios tipos de notificación al mismo tiempo? Si tu casa está en llamas, probablemente quieras que te informen a través de todos los canales”.

Intentaste solucionar ese problema creando subclases especiales que combinaban varios métodos de notificación dentro de una clase. Sin embargo, enseguida resultó evidente que esta solución inflaría el código en gran medida, no sólo el de la biblioteca, sino también el código cliente.

Debes encontrar alguna otra forma de estructurar las clases de las notificaciones para no alcanzar cifras que rompan accidentalmente un récord Guinness.

1. **Facade**:

Imagina que debes lograr que tu código trabaje con un amplio grupo de objetos que pertenecen a una sofisticada biblioteca o framework. Normalmente, debes inicializar todos esos objetos, llevar un registro de las dependencias, ejecutar los métodos en el orden correcto y así sucesivamente.

Como resultado, la lógica de negocio de tus clases se vería estrechamente acoplada a los detalles de implementación de las clases de terceros, haciéndola difícil de comprender y mantener.

1. **Flyweight**:

Para divertirte un poco después de largas horas de trabajo, decides crear un sencillo videojuego en el que los jugadores se tienen que mover por un mapa disparándose entre sí. Decides implementar un sistema de partículas realistas que lo distinga de otros juegos. Grandes cantidades de balas, misiles y metralla de las explosiones volarán por todo el mapa, ofreciendo una apasionante experiencia al jugador.

Al terminarlo, subes el último cambio, compilas el juego y se lo envias a un amigo para una partida de prueba. Aunque el juego funcionaba sin problemas en tu máquina, tu amigo no logró jugar durante mucho tiempo. En su computadora el juego se paraba a los pocos minutos de empezar. Tras dedicar varias horas a revisar los registros de depuración, descubres que el juego se paraba debido a una cantidad insuficiente de RAM. Resulta que el equipo de tu amigo es mucho menos potente que tu computadora, y esa es la razón por la que el problema surgió tan rápido en su máquina.

El problema estaba relacionado con tu sistema de partículas. Cada partícula, como una bala, un misil o un trozo de metralla, estaba representada por un objeto separado que contenía gran cantidad de datos. En cierto momento, cuando la masacre alcanzaba su punto culminante en la pantalla del jugador, las partículas recién creadas ya no cabían en el resto de RAM, provocando que el programa fallara.

1. **Proxy**:

¿Por qué querrías controlar el acceso a un objeto? Imagina que tienes un objeto enorme que consume una gran cantidad de recursos del sistema. Lo necesitas de vez en cuando, pero no siempre.

Puedes llevar a cabo una implementación diferida, es decir, crear este objeto sólo cuando sea realmente necesario. Todos los clientes del objeto tendrán que ejecutar algún código de inicialización diferida. Lamentablemente, esto seguramente generará una gran cantidad de código duplicado.

En un mundo ideal, querríamos meter este código directamente dentro de la clase de nuestro objeto, pero eso no siempre es posible. Por ejemplo, la clase puede ser parte de una biblioteca cerrada de un tercero.

1. El rol que juegan las interfaces, en relación a los patrones, es el de facilitar la modularidad de la aplicación, permitiendo poder añadir funcionalidad o componentes que con solamente implementar tal interfaz permite que no se tenga que modificar el código existente. Varios patrones utilizan las interfaces para resolver problemas, algunos de ellos son abstract factory, factory, adapter, bridge.
2. Una limitación que presenta la herencia es que es estática, es decir que no se puede alterar la funcionalidad de un objeto existente durante el tiempo de ejecución. Sólo se puede sustituir el objeto completo por otro creado a partir de una subclase diferente. Las subclases sólo pueden tener una clase padre. En la mayoría de lenguajes, la herencia no permite a una clase heredar comportamientos de varias clases al mismo tiempo. Por ello, la agregación o la composición se considera una de las formas de superar tal limitación, dado que ambas funcionan de la siguiente manera: un objeto tiene una referencia a otro y le delega parte del trabajo, mientras que con la herencia, el propio objeto puede realizar ese trabajo, heredando el comportamiento de su superclase. Con esta solución se puede sustituir fácilmente el objeto “ayudante” vinculado por otro, cambiando el comportamiento del contenedor durante el tiempo de ejecución. Un objeto puede usar el comportamiento de varias clases con referencias a varios objetos, delegándoles todo tipo de tareas.
3. La diferencia entre herencia de clase e implementación de interfaz, es que la primera se usa cuando las clases están estrechamente relacionadas y es conveniente reutilizar la implementación de comportamientos y atributos previos. En cambio, la segunda se utiliza en clases que no están estrechamente relacionadas pero requieren que compartan un comportamiento y/o atributos, pero no se puede reutilizar la implementación, la interfaz te obliga a implementarlo de forma específica para cada clase.