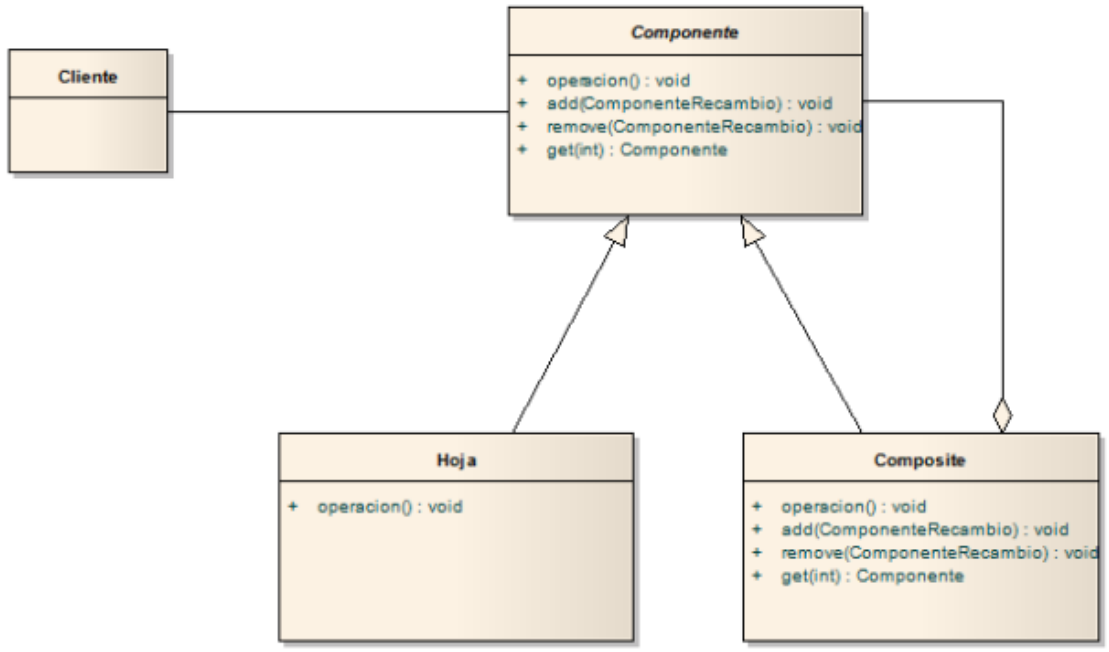
PATRÓN COMPOSITE

Objetivo:

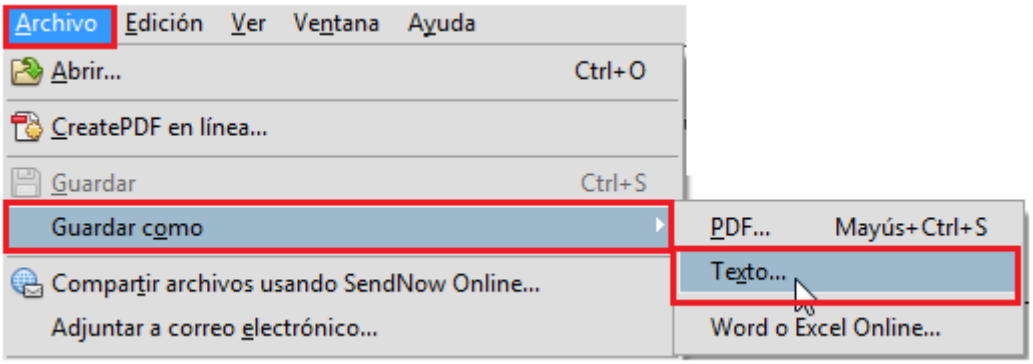
«Componer objetos en árboles para representar jerarquías todo-parte. Composite permite a los clientes tratar objetos individuales y objetos compuestos de una manera uniforme».



El patrón *Composite* se aleja un poco de la línea tradicional de los patrones vistos hasta ahora, ya que rompe uno de los principios de la programación orientada a objetos: **una clase, una responsabilidad**.

Cuando diseñamos debemos tener claro que la idea principal es alcanzar un equilibrio entre muchos factores como por ejemplo presupuesto, usabilidad y facilidad para que nuestro código sea reutilizable y pueda ser fácilmente mantenible en un futuro. El objetivo de este patrón es la **facilidad de uso**.

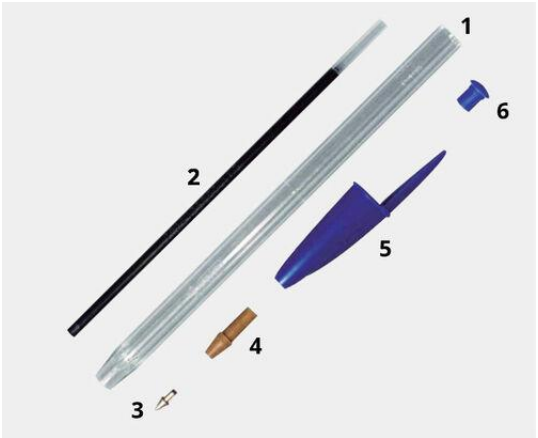
A grandes rasgos, el patrón *Composite* **permite crear una jerarquía de elementos anidados unos dentro de otros**. Cada elemento permitirá alojar una colección de elementos del mismo tipo, hasta llegar a los elementos «reales» que se corresponderán con los nodos «Hoja» del árbol. Un ejemplo del concepto de la jerarquía que se pretende modelar sería el de los menús de una aplicación:



En este ejemplo tenemos un **menú (Archivo)** que contiene varios elementos, que pueden ser «hojas» que ejecutan una operación (Abrir, CreatePDF en línea.., Compartir archivos usando SendNow Online…, Adjuntar a correo electrónico…) **o bien otro menú** (Guardar como) que a su vez contiene más elementos «hoja» (PDF…, Texto…, Word o Excel Online…).

Creo que el ejemplo es lo suficientemente ilustrativo como para entender el concepto: poder anidar menús que puedan contener o bien otros menús, o bien directamente nodos hoja que ejecuten operaciones. Este ejemplo se aproxima bastante al concepto de *Composite*, pero no se ajusta exactamente a su filosofía, ya que le falta una funcionalidad: el submenú debería ser capaz de ejecutar una operación que se encargaría de iterar sobre todos los subelementos que contiene, ejecutando la operación de cada uno de ellos. Sin embargo, deja bastante claro el esquema lógico que seguirá este patrón.

Otro ejemplo casero: empresa que fabrica biromes, y tiene que actualizar el precio de las partes de la birome, y como consecuencia el precio de la birome:



1. Cuerpo de plástico del bolígrafo
2. Tubo de tinta
3. Punta de latón y bola de carburo de tungsteno
4. Conector
5. Capuchón antiasfixia
6. Terminal de plástico

Llevando el auto al taller

Para ilustrar el patrón, acudiremos a un taller, que tuvo a bien contratar a un ingeniero de software para que le diseñara una aplicación para realizar el inventariado de las piezas de recambio. El dueño del taller le expuso al ingeniero la dificultad que tenía

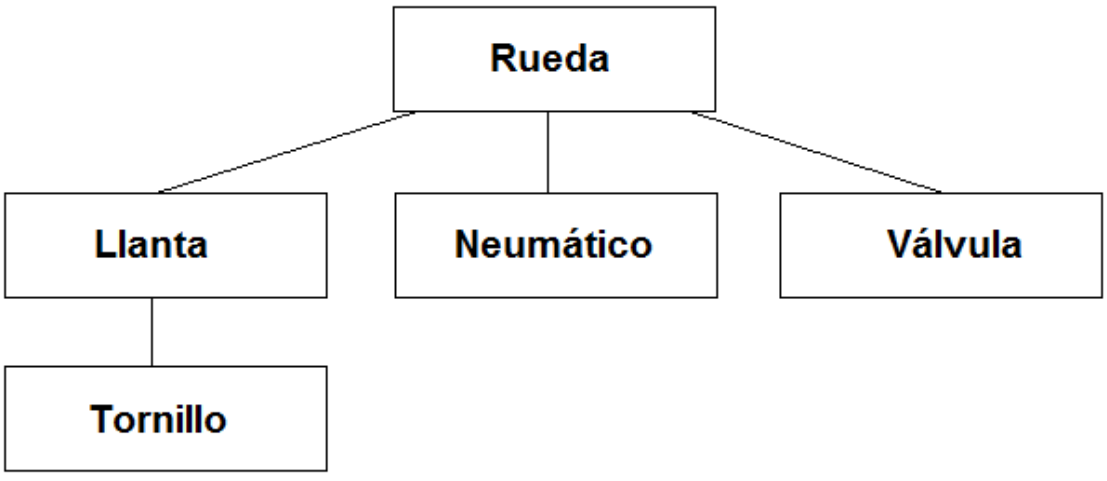
para establecer los precios de los recambios, ya que dependiendo de la rotura, a veces era necesario cambiar piezas enteras mientras que en otras ocasiones bastaba con sustituir un pequeño componente. Nuestro cliente había almacenado en una base de datos cada uno de los componentes con su respectivo precio, pero cuando el proveedor le aumentaba el precio de una pieza que formaba parte de otra, tenía que preocuparse de actualizar, una a una, todas las piezas de grano más grueso en las cuales estaba contenida esta pieza. Por poner un ejemplo, **si el precio de un tornillo para una rueda aumentaba**, nuestro mecánico tenía que acceder a su catálogo y modificar el precio total en:

- Tornillo para rueda
- Llantas (en todos y cada uno de los modelos que usaran los tornillos anteriores)
- Ruedas (en todos y cada uno de los modelos que usaran las llantas anteriores)

Entónces, se desarrolló un sistema que realizara el cálculo del precio de forma automática a la vez que modelaba el almacén de recambios con una estructura de árbol. En lugar de disponer del siguiente modelo:

Nombre	Descripción	Precio
Tornillo llanta	Tornillo llanta marca ACME	0.21
Llanta ACME 15'	Llanta ACME de 15'	42.22
Llanta ACME 16'	Llanta ACME de 16'	51.00
Llanta ACME 17'	Llanta ACME de 17'	54.00
Neumático 15'	Neumático Michelin de 15'	13.42
Neumático 16'	Neumático Michelin de 16'	14.98
Neumático 17'	Neumático Michelin de 17'	17.22
Válvula	Válvula neumático genérica	0.49
Rueda 15'	Rueda de 15' con llanta ACME de 15' y neumático Michelin	56,97
Rueda 16'	Rueda de 16' con llanta ACME de 16' y neumático Michelin	67,31
Rueda 17'	Rueda de 17' con llanta ACME de 17' y neumático Michelin	72,55

Le proporcionamos este (**...los componentes de una rueda!**):



Aplicando Composite

Comenzaremos creando nuestro elemento abstracto o interfaz correspondiente al «Componente». Esta clase o interfaz debe exponer los métodos comunes tanto a los elementos compuestos como a los elementos «hoja». Lo verdaderamente importante es que este elemento sea una abstracción (depender de abstracciones, no de concreciones).

- Si optamos por la interfaz, simplemente definiremos sus operaciones.
- Si optamos por la clase abstracta, además de definir sus operaciones, añadiremos un comportamiento por defecto en el que lanzaremos una excepción que indique que la operación no está soportada. De este modo, las clases derivadas deberán encargarse de proporcionar la funcionalidad. En caso de no proporcionarla, lanzarán una excepción (por ejemplo, el método *add* en un elemento *Hoja* no tiene sentido, por lo que deberá tener este comportamiento por defecto).

Ver clase ComponenteRecambio.cs

Sobre esta clase trabajaremos creando otras dos clases que simbolizarán los dos tipos de elemento que puede contener nuestro patrón ***Composite*: elementos hoja (ElementoRecambio) y elementos compuestos (Recambio).** Así, nuestros elementos «hoja» serán los que incorporen la verdadera funcionalidad, que será invocada por los elementos compuestos en caso de que fuera necesario.

Hemos utilizado «nomenclatura Java» (getters y setters en lugar de una propiedad para ambas operaciones) porque a la hora de sobrecargar los métodos será más intuitivo si las operaciones se encuentran separadas, como veremos más adelante. (Queda claro que también es posible usar propiedades virtuales y sobrecargarlas del mismo modo que hacemos con estos dos métodos).

La clase *ElementoRecambio* almacenará información como el nombre, la descripción y el precio:

Ver clase ElementoRecambio.cs

Seguramente, a estas alturas ya nos habremos dado cuenta de lo que comentábamos al principio del artículo respecto a la ruptura del principio de responsabilidad única. La clase *Elemento* implementa un conjunto de métodos, pero deja otro conjunto sin

implementar (los métodos *add, remove* y *getElemento*), que lanzarán una excepción si se utilizan. Esos métodos serán implementados por la siguiente clase: *Recambio*.

Ver clase Recambio.cs 1ERA PARTE

En esta primera etapa, hemos codificado la parte que se encarga de añadir, eliminar y consultar otros elementos. El método es sencillo: a través de un ArrayList interno, los métodos *add, remove* y *getElemento* realizarán operaciones sobre él.

A continuación codificaremos los métodos *get* que recuperarán el contenido de los atributos de cada *ElementoRecambio*: nombre, descripción y precio. Para ello iteraremos sobre los elementos contenidos dentro de cada *Recambio*.

Ver clase Recambio.cs 2DA PARTE

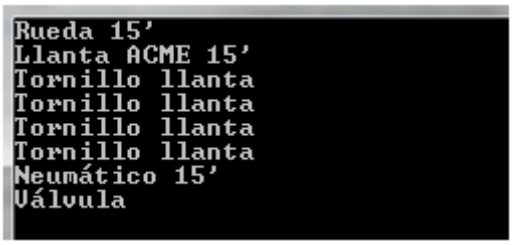
Uso de Composite desde código cliente

Recorriendo todos los elementos contenidos dentro de cada recambio podremos obtener la información almacenada tanto en un objeto nodo (***Recambio***) como en un objeto hoja (***ElementoRecambio***). Ambos se tratarán de la misma manera, aunque la implementación de los métodos sea distinta.

Y dado que ambos objetos son intercambiables, estamos consiguiendo lo que declaramos en primera instancia: componer objetos de forma de árbol respetando la jerarquía todo-parte permitiendo que ambos elementos se traten de forma uniforme.

Ver código del Main

Si consultamos el método *getNombre()* de nuestra rueda, se mostrará el contenido del atributo privado *nombre* para, a continuación, iterar sobre cada elemento del ArrayList invocando a su vez el método *getNombre()* de cada elemento de forma recursiva. Así, obtendríamos lo siguiente: Lo cual nos daría la siguiente salida:



Si hacemos lo propio con el precio, obtendríamos la suma de todos los precios de sus elementos hoja. Si después de esto realizamos una modificación en el precio de uno de los elementos contenidos dentro del objeto compuesto, el cambio se propagará de forma automática sin necesidad de modificar el precio total de cada rueda.

Purismo vs. Transparencia

Nos acabamos de encontrar con el primer patrón que viola deliberadamente uno de los principios de la programación orientada a objetos. El motivo ha sido que el incremento en la transparencia (y usabilidad) es tan importante que merece la pena el sacrificio de permitir que una clase adquiera más de una responsabilidad. Recordemos que los patrones de diseño son **soluciones genéricas a problemas concretos**, por lo que su objetivo es la de facilitar el desarrollo de software. Hay ocasiones, por tanto, en las que las que las ventajas de romper las normas sobrepasan de largo seguirlas a rajatabla.

¿Cuándo utilizar este patrón? Ejemplos reales

Los escenarios en los que este patrón suele utilizarse son, principalmente:

- Como su propia definición indica, cuando se requiere representar jerarquías todo-parte que superen cierto tamaño.
- Cuando se desea que los clientes puedan ignorar la diferencia entre colecciones de objetos y objetos individuales, haciendo que ambos se traten de la misma manera.

  Otros ejemplos

https://refactoring.guru/es/design-patterns/composite#:~:text=El%20uso%20del%20patr%C3%B3n%20Composite,n%C3%BAmero%20de%20Cajas%20m%C3%A1s%20peque%C3%B1as.