**AA2 PATRONES DE DISEÑO ESTRUCTURALES**

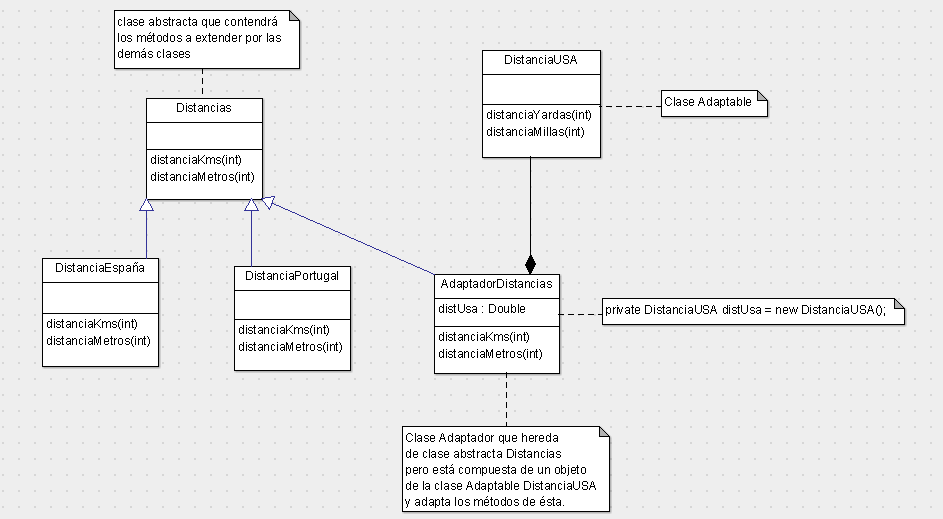
**1.Patrón Adapter**

1. Descripción del escenario.

Se quiere crear una aplicación que calcule distancias. Queremos calcular no solo las distancias entre el sistema internacional (metros y kilómetros) sino también pasar del sistema internacional al sistema anglosajón (yardas y millas). La solución es utilizar este patrón donde el cliente utilizará la clase adaptador para hacer las conversiones de sistema internacional a sistema anglosajón sin saber nada de la clase que de cálculo de sistema anglosajón. Esta clase adaptador heredará de la misma clase que las clases del sistema internacional, pero contendrá un objeto de la clase que contiene el sistema de cálculo anglosajón.

1. Diagrama de clases.

Se crea una clase abstracta Distancias con dos métodos que calculan la distancia introducida por el usuario en kilómetros y metros. Se crean, por ejemplo, dos clases más que extienden estos dos métodos DistanciaEspaña y DistanciaPortugal ya que en estos dos países se utiliza el sistema internacional. Ahora se crea una tercera clase con métodos distintos que calcula las distancias con el sistema anglosajón que es DistanciaUSA. ¿Cómo se adapta esta clase a la clase Distancias si los cálculos son distintos? Pues se crea la clase AdaptadorDistancias que hereda de la clase abstracta Distancias pero a su vez contiene un objeto de la clase DistanciaUSA y así consigue adaptar las medidas de un sistema a otro.



1. Código fuente.

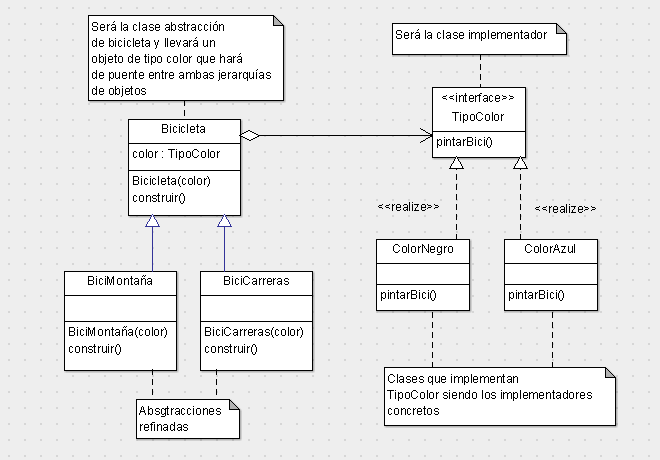
**2.Patrón Bridge**

1. Descripción del escenario.

Una empresa de cadena robótica de bicicletas quiere que le hagamos una aplicación (CreandoBicis) para indicarle a los robots el tipo de bici que queremos hacer en cada momento. Lo requisitos de la empresa son claros: quieren varios tipos de bicis y de distintos colores. Los tipos de bicis pueden ser: bici de montaña, bici de paseo, bici de carreras, etc. Y los distintos tipos de colores pueden ser también bastantes. Nos ponemos a crear la aplicación y tenemos un problema y es que tendríamos que crear una super clase Bici y de ella heredan bici de montaña de color azul, bici de montaña de color negro, bici de montaña de color verde… y lo mismo para los otros tantos tipos de bicis. Esto nos haría una jerarquía de clases inmanejable y tediosa. Como solución se utilizará el patrón Bridge separando los dos tipos de jerarquía: Tipo de bici y tipo de color y luego uniéndolas (puente) poniéndole a una de ellas un objeto del otro. Pudiendo de esta forma tener menos clases y de forma más fácil y extensible añadir los distintos colores y tipos de bicis.

1. Diagrama de clases.

A modo de ejemplo sencillo se crea para la jerarquía de colores la interfaz TipoColor (que será el implementador) con un solo método pintarBici() que luego implementarán a su modo las clases ColorAzul y ColorNegro. Para la otra jerarquía creo la clase Bicicleta (será la abstracción de bicicleta) y tendrá un atributo de TipoColor que actuará de puente entre ambas jerarquías de clases. Este atributo será utilizado por los constructores de la todas las clases que hereden de Bicicleta haciendo dos para el ejemplo BiciMontaña y BiciCarreras. Esta clase también tendrá un solo método construir() que hará la bici y después cogerá el atributo de color para pintarla.



1. Código fuente.

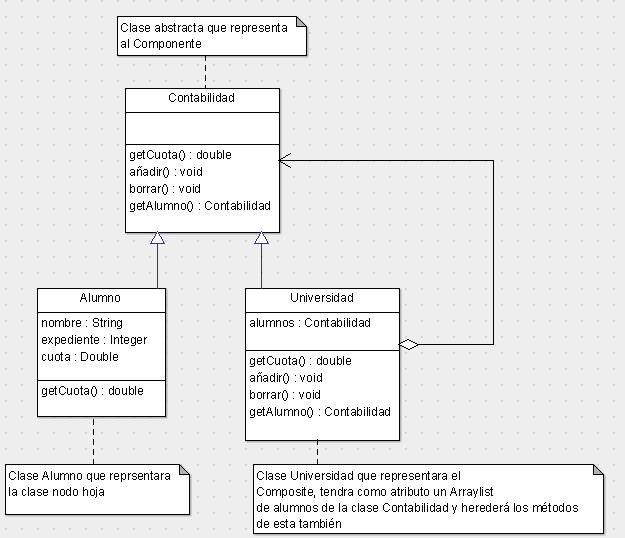
**3.Patrón Composite**

1. Descripción del escenario.

Una universidad nos pide que hagamos una aplicación (ContaUni) que calcule el total de ingresos generados al mes por las distintas cuotas de los alumnos. Para resolver esto vamos a utilizar el patrón Composite ya que podremos ver la cuota aportada por cada alumno mensualmente y ver también los ingresos totales que recibe la universidad, tratando al alumno como un objeto simple y a la universidad como un objeto compuesto. Utilizaremos la recursividad de clases, donde una clase se llamará a sí misma.

1. Diagrama de clases.

Como vemos en el diagrama se hará un ejemplo con pocas clases para nuestro árbol. Un clase Contabilidad que será una clase abstracta y actuará de componente. Un clase Alumno que heredará de esta clase y será un nodo hoja. Y la clase Universidad que hará de Composite que heredará de la clase abstracta y a la vez estará compuesta por ella. Se podrían añadir más clases de tipo nodo hoja como Empleado, Proveedor, etc. que actuarían como la clase Alumno pero a modo de ejemplo con esto nos valdría. Solo habría que añadir en la clase Composite Universidad más ArrayList, otro para los empleados, otro para los proveedores, etc.



1. Código fuente.