Práctica 2

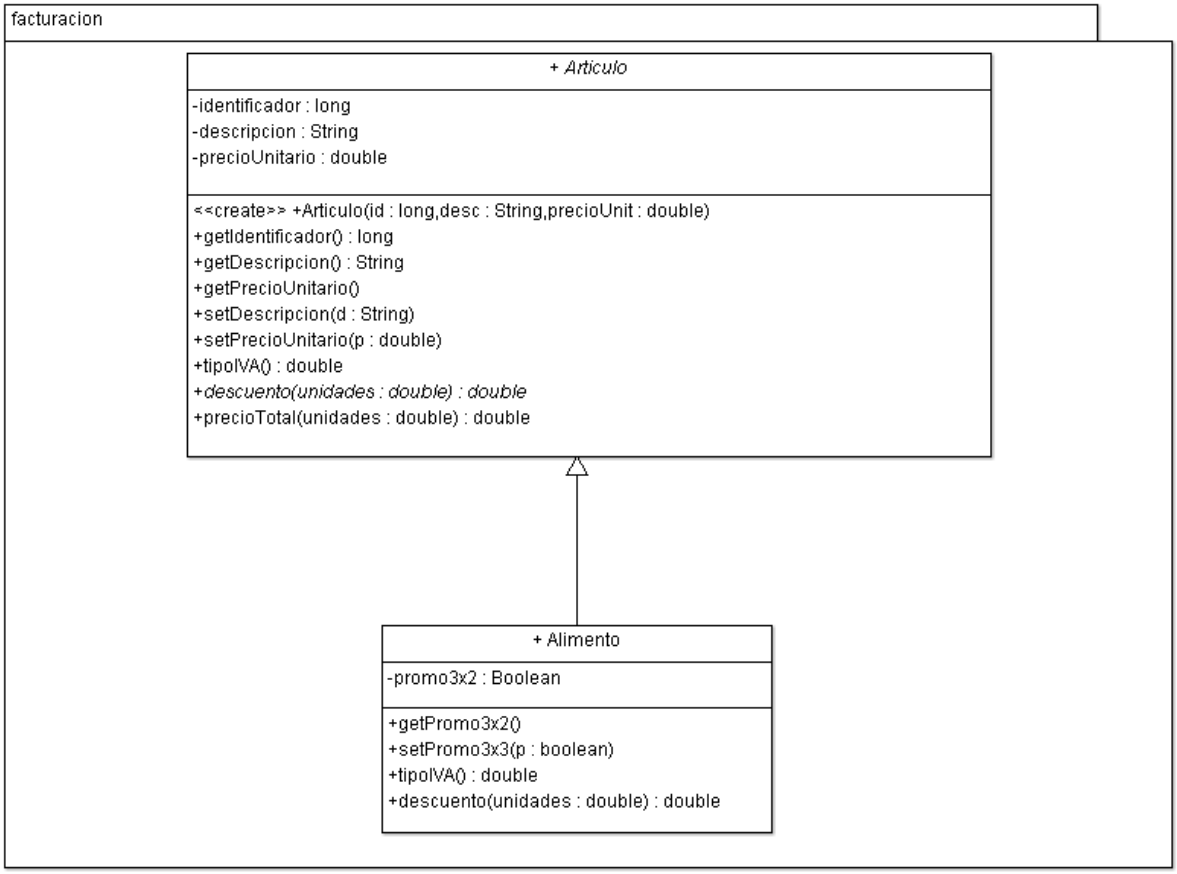
Introducción al Diseño Orientado a Objetos

Inicio: Semana del 4 de febrero.
Duración: 2 semanas.
Entrega: Semana del 18 de febrero (el día anterior a la siguiente práctica, según grupo)**, en parejas.**
Peso de la práctica: 15%

El objetivo de esta práctica es introducir los conceptos básicos de orientación a objetos, desde el punto de vista del diseño, así como de su correspondencia en Java, con particular énfasis en el diseño de clases con relaciones de herencia, asociación y agregación mediante diagramas de clases en UML.

Apartado 1 (4 puntos):

Sea una aplicación que almacena información los artículos de venta en grandes superficies. El siguiente diagrama de clases describe dos clases: una clase base **abstracta** `Articulo` que almacena el identificador, la descripción y el precio unitario de cada artículo (como **atributos privados**), y una subclase `Alimento` que modela los artículos de alimentación con un atributo privado adicional indicando si se les debe aplicar la promoción 3x2. Además del constructor y los métodos *getters* y *setters*, la clase `Articulo` tiene otros tres **métodos públicos**: el método *tipoIVA()* de la clase base `Articulo` devuelve el porcentaje de IVA que corresponderá aplicar sobre su precio según el tipo de artículo de que se trate; el método **abstracto** *descuento(unidades)* de la clase `Articulo` devuelve el descuento aplicable al precio sin impuestos del artículo; y el método *precioTotal(unidades)* calcula y devuelve el precio final a facturar por ese número de unidades del producto, incluyendo la aplicación del descuento y del IVA que le corresponda.



A continuación se muestra la implementación completa de dichas clases en Java. Como puedes ver, las clases se han incluido en el paquete "facturacion". Los paquetes sirven para organizar las clases que intervienen en programas grandes, con muchas clases y varios paquetes. De esta manera, clases más relacionadas entre sí se declararán en el mismo paquete. La estructura de los paquetes es jerárquica y se ha de corresponder con la estructura física de directorios (es decir, las clases han de almacenarse en un directorio "facturacion").
Los "constructores" tienen el mismo nombre que la clase, y son invocados mediante **new** para construir un objeto (crear una instancia de la clase) e inicializar sus atributos. Así, el constructor de `Articulo` toma tres parámetros e inicializa los atributos identificador, descripción y precio unitario. Fíjate también en cómo el constructor de la clase `Alimento`, que es **subclase** de `Articulo`, llama mediante **super(…)** al constructor de su **superclase** `Articulo` antes de inicializar el atributo promo3x2 específico de la subclase `Alimento`. Finalmente, observa cómo la clase `Alimento` sobrescribe el método *tipoIVA()* heredado de su superclase o clase padre, y cómo implementa el método abstracto *descuento(unidades)* que no tenía implementación en la clase `Articulo`.

Clase Articulo

```
package facturacion;

public abstract class Articulo {
	private long identificador;
	private String descripcion;
	private double precioUnitario;

	public Articulo(long id, String desc, double precio) {
		identificador = id; descripcion = desc; precioUnitario = precio;
	}
	public long getIdentificador() { return identificador; }
	public String getDescripcion() { return descripcion; }
	public double getPrecioUnitario() { return precioUnitario; }
	public void setDescripcion(String desc) { descripcion = desc; }
	public void setPrecioUnitario(double precio) { precioUnitario = precio; }

	// el tipo IVA general es 21% aplicable salvo que se redefina en una subclase
	public double tipoIVA() { return 0.21; }

	// cada subclase de articulo calculará el descuento que corresponda
	public abstract double descuento(double unidades);

	// el precio total siempre se calcula de la misma forma
	public double precioTotal(double unidades) {
		return ((precioUnitario * unidades) - descuento(unidades))
			* (1.0 + tipoIVA());
	}
}
```

Clase Alimento

```
package facturacion;

public class Alimento extends Articulo {
	private boolean promo3x2;
	public Alimento(long id, String desc, double precio, boolean promo) {
		super(id, desc, precio);
		promo3x2 = promo;
	}
	public boolean getPromo3x2() { return promo3x2; }
	public void setPromo3x2(boolean promo) { promo3x2 = promo; }
	public double tipoIVA() { return 0.10; }
	public double descuento(double unidades) {
		if (promo3x2) {
		    return getPrecioUnitario() * (int) (unidades / 3);
		} else {
		    return 0.0;
		}
	}
}
```

El siguiente programa ejemplo utiliza las dos clases anteriores para calcular e imprimir el precio final de facturación de 7 unidades de chocolate a la taza con un precio unitario de 2.5 y en promoción 3x2, así como el de 4 unidades de yogur líquido a 1.25 la unidad, sin promoción. Como puedes ver, los objetos se crean mediante **new** seguido del nombre de la clase concreta con los parámetros del constructor (más adelante veremos que una clase puede tener más de un constructor). En cambio, el tipo usado en la declaración de la variable puede ser una clase abstracta o concreta. El compilador comprueba que la clase del objeto asignado a una variable sea compatible con la clase usada en la declaración de esa variable.

Programa ejemplo

```
import facturacion.*;

public class Ejemplo1 {
  public static void main(String args[]) {
       Articulo a1 = new Alimento(990034, "Chocolate a la taza", 2.5, true);
       System.out.println("Precio total: "
                          + a1.precioTotal(7)); // 13.75 = (7 * 2.5 – 2 * 2.5) * 1.10
       Alimento a2 = new Alimento(990268, "Yogur liquido", 1.25, false);
       System.out.println("Precio total: "
                          + a2.precioTotal(4)); // 5.5 = (4 * 1.25 – 0.0) * 1.10

  }
}
```

La salida del programa anterior es la siguiente:

```
Precio total: 13.750000000000002
Precio total: 5.5
```

Se pide:

Añade dos clases al diagrama UML y escribe su código Java para modelar los **libros** y el **tabaco** como otras dos formas de artículos en venta cumpliendo los siguientes requisitos. Cada libro tiene un atributo que identifica a qué categoría pertenece: “LibroDeTexto”, “Coleccion”, “Best-sellers”, etc. Se debe poder cambiar la categoría de un libro una vez creado. A los libros se les aplica un tipo de IVA reducido (4%) y su descuento se calcula según su categoría: 15% para libros de texto, y 50% para la tercera unidad y sucesivas de los libros de colección. Respecto al tabaco, el precio final se calcula sin descuento y aplicando el tipo de IVA general. Completa la clase `Articulo` con un método toString() de forma que, con el código de tus dos nuevas clases, la ejecución de este programa

```
import facturacion.*;

public class Ejemplo2 {
  public static void main(String args[]) {
    Articulo a3 = new Libro(940111, "Dibujo Técnico", 15, "LibroDeTexto");
    Articulo a4 = new Libro(940607, "Grandes Clasicos", 10, "Coleccion");
    Libro a5 = new Libro(940607, "Fifty fifty", 3.25, "SinClasificar");
    Tabaco a6 = new Tabaco(690023, "Fortuna", 1.75);

    int cant = 2;
    System.out.println(cant + " unidades de " + a3 + " Precio final: " + a3.precioTotal(cant));
    cant = 5;
    System.out.println(cant + " unidades de " + a4 + " Precio final: " + a4.precioTotal(cant));
    cant = 4;
    System.out.println(cant + " unidades de " + a5 + " Precio final: " + a5.precioTotal(cant));
    cant = 1;
    System.out.println(cant + " unidades de " + a6 + " Precio final: " + a6.precioTotal(cant));
  }
}
```

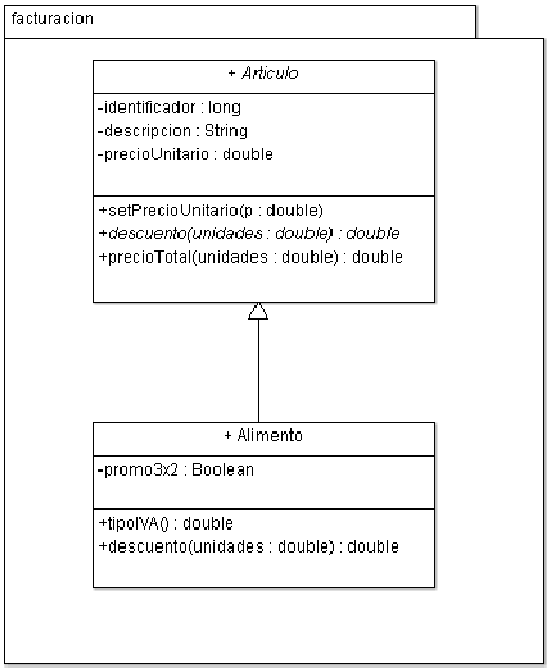
produzca la siguiente salida:

```
2 unidades de Id:940111 Dibujo Técnico Precio: 15.0 Precio: 26.52
5 unidades de Id:940607 Grandes Clasicos Precio: 10.0 Precio: 36.4
4 unidades de Id:940607 Fifty fifty Precio: 3.25 Precio: 13.52
1 unidades de Id:690023 Fortuna Precio: 1.75 Precio: 2.1174999999999997
```

Nota:

En realidad, un diagrama de clases es una abstracción del código final, así que normalmente y para facilitar su comprensión no se suelen incluir todos los detalles necesarios para la codificación. De esta manera se suelen omitir

los métodos “setters” y “getters”, y los constructores. El siguiente diagrama muestra el diagrama de clases anterior, usando este mayor nivel de abstracción.



Por si quieres saber más...:

El método *precioTotal()* de la clase `Articulo` es un ejemplo del patrón de diseño “Template Method”, que consiste en escribir código en una clase padre que llama a métodos (quizá abstractos) que se implementan en la subclase o clase hija. Tienes más información sobre patrones de diseño en:
Patrones de diseño elementos de software orientado a objetos reutilizable. Gamma, E., Helm, R., Johnson, R., Vlissides, J. INF/681.3.06/PAT. Addison-Wesley, 2003.

Apartado 2 (4 puntos):

Para el taller de un fabricante de automóviles se va a desarrollar una aplicación de control de existencias de los recambios disponibles para reparaciones o para venta directa al cliente. Realiza el diagrama de clases en UML para dicha aplicación a partir de la descripción dada a continuación. Es conveniente que, después de una primera lectura completa de toda la descripción, vayas construyendo el diagrama progresivamente con las ampliaciones o cambios que se requieran para cada punto de la descripción.

1. La aplicación contiene un *catálogo de recambios* que hace accesible la información de *todos* los recambios, ya sean recambios básicos o compuestos, tengan o no existencias en almacén, e incluso aquellos que han dejado de fabricarse. La aplicación permitirá que se añadan nuevos recambios al catálogo y que se eliminen recambios de él.

2. Cada *recambio* se identifica mediante un código numérico y una descripción, y tendrá asociada una lista de modelos de automóvil para los que el recambio es válido. Si se trata de un *recambio básico*, es decir, que no está compuesto de otras partes o recambios, se registrará también su precio unitario, la fecha de creación, y la fecha de descatalogación (si es que se ha dejado de fabricar). Además, para cada producto básico debe poder conocerse cuántas unidades hay en el almacén. Existen también *recambios compuestos* que, a pesar de tener su propio código y descripción, están formados por otros recambios (básicos o compuestos).

3. Por ejemplo, cada *llanta* es un recambio básico y cada *tornillo simple para llanta* también; el *juego de 4 tornillos de seguridad* también es un recambio básico porque no se venden estos tornillos sueltos. En cambio, el *juego de 4 tornillos simples* es un recambio compuesto. También pueden existir recambios compuestos como el siguiente: “*un juego de 4 llantas con un juego de 4 tornillos de seguridad y tres juegos de 4 tornillos simples”*, que está formado por 5 recambios básicos y 3 compuestos.

4. La aplicación llevará el control de existencias en almacén para todos los recambios básicos. Para los recambios compuestos, la aplicación debe poder calcular cuántas unidades hay disponibles en almacén, a partir de las existencias de sus componentes. Así mismo, el precio de los recambios compuestos se calculará a partir de los recambios que los componen. Algunos recambios compuestos podrán tener un porcentaje de reducción aplicar sobre el precio total de sus componentes.

5. El control de existencias se lleva a cabo mediante el registro de entradas y salidas de recambios. Las entradas se producen cuando el taller recibe de la fábrica los recambios básicos que había pedido. No se piden recambios compuestos a fábrica. Las salidas corresponden a ventas de recambios a clientes del taller o al consumo propio para reparaciones realizadas en el propio taller.

Se pide:
Realizar el diagrama de clases en UML para la aplicación descrita arriba. No es necesario incluir constructores, ni métodos *getters* y *setters*. Tampoco se necesario entregar su implementación en Java.

Apartado 3 (2 puntos):

Se va a construir una aplicación para la gestión de personal de las universidades públicas de Madrid, aunque en este apartado nos centraremos solo en la parte del diagrama de clases que refleja los siguientes aspectos.

El personal *funcionario* se identifica mediante un número de registro de personal y además tiene como características principal su fecha de ingreso en el cuerpo de funcionario y su categoría salarial (un entero que sirve para calcular su sueldo base). Algunos funcionarios ocupan puestos de responsabilidad administrativa, como es el de *administrador de centro*. Para cada administrador de centro debemos conocer la universidad a la que está adscrito y el centro (facultad o escuela) del que es administrador, así como un complemento salarial que se debe sumar al sueldo para calcular su sueldo final. Por otro lado, se debe almacenar información relativa al personal *docente* de las universidades. Cada docente se identifica mediante su número de DNI y pertenece a un departamento (el departamento es unidad independiente de universidad y el centro, y puede haber departamentos relacionados con varias universidades o varios centros); además para cada docente debemos saber si es doctor o no (ya que esto influye en el cómputo del sueldo de todos los docentes) y debemos mantener un histórico de las actividades docentes que ha tenido asignadas, en el que para cada año y cuatrimestre se registre las asignaturas que ha impartido, con su tipo de docencia (teoría, práctica, o proyecto) y el número de horas por semanas que ha impartido en cada una de esas asignaturas en ese cuatrimestre. Los *profesores titulares* son docentes que además pertenecen al cuerpo de funcionarios, y como características adicionales debemos conocer el número de quinquenios de evaluación docente positiva y el número de sexenios de evaluación investigadora positiva, ya que ambos aportan complementos salariales. Finalmente, existen *profesores asociados*, que no son funcionarios sino trabajan en empresas privadas y colaboran (normalmente con dedicación a tiempo parcial) en las tareas docentes de las universidades. Para éstos debemos conocer el nombre de su empresa, su número de seguridad social, y el porcentaje de su dedicación parcial a la docencia, ya que de este porcentaje depende en parte su sueldo.

Se pide:
a) Realiza un diagrama de clases UML para dicha aplicación. No es necesario incluir constructores, ni métodos *getters* y *setters*. Tampoco se necesario entregar su implementación en Java.

b) Comenta razonadamente si convendría hacer algún cambio en ese diagrama de clases en el supuesto de que se fuese a implementar la aplicación en Java, y en caso afirmativo entrega también el nuevo diagrama.

Apartado 4 (Opcional, 1 punto):

Realiza un nuevo diagrama de clases UML, partiendo del entregado en el apartado 2 pero aparte de éste, para cumplir con las siguientes características adicionales de la aplicación:

La aplicación para el taller de un fabricante de automóviles también debe incluir un catálogo con todos los modelos de vehículo que comercializa este fabricante, así como un registro de cada vehículo que ha pasado por el taller para algún tipo de servicio. El objetivo de dicho registro es llevar un control histórico de las distintas revisiones y reparaciones aplicadas a cada vehículo, incluyendo los recambios que se hayan empleado en cada una. Para cada modelo de vehículo se almacenará una identificación alfanumérica, una breve descripción y una lista de las revisiones de mantenimiento programado para ese modelo. Las revisiones de mantenimiento se realizan en cuanto se alcance alguna de estas dos condiciones: determinado kilometraje realizado o cierto número de años trascurridos

desde el alta del vehículo. Cada revisión programada se caracteriza por el kilometraje previsto realizado, años previstos transcurridos, precio estimado, duración de la revisión, y lista de recambios previsiblemente necesarios. Otras revisiones de mantenimiento no programado se pueden realizan a petición del propietario mediante solicitud explicita a través del servicio de recepción del taller. Cuando el propietario de un vehículo solicita una revisión para su vehículo, debe indicar si se trata de un mantenimiento programado o, en caso contrario, debe dar una descripción del motivo para la revisión (p.ej.: pasar la ITV, preparar para invierno, las marchas no entran bien, se desvía al frenar fuerte, …) Al recibir una solicitud de mantenimiento programado se debe poder comprobar si el taller tiene disponibles todos los recambios necesarios. En el caso de las revisiones o reparaciones no programadas, será después de realizadas cuando se registren los recambios utilizados. De esta forma, la aplicación debe permitir consultar, para cualquier vehículo, el histórico de recambios sustituidos, con indicación de fecha y kilometraje del vehículo en el momento de la sustitución.
Crea y entrega el nuevo diagrama de clases de esta aplicación, aparte del entregado en el apartado 3, incluyendo las clases y métodos necesarios, añadiendo la justificación razonada de las decisiones o suposiciones que hayas tenido en cuenta sobre el funcionamiento del periódico.

Normas de Entrega:

- Se deberán entregar los apartados 1, 2 y 3 (y opcionalmente el 4)
- La entrega la realizará uno de los alumnos de la pareja, a través de Moodle.
- Si el ejercicio pide código Java, se ha de entregar además la documentación generada con *javadoc*. Si el ejercicio pide un diagrama de diseño, se deberá entregar en PDF junto con una breve explicación (dos o tres párrafos a lo sumo).
- Se debe entregar un único fichero ZIP / RAR con todo lo solicitado, que deberá llamarse de la siguiente manera: GR<numero_grupo>_<nombre_estudiantes>.zip. Por ejemplo Marisa y Pedro, del grupo 2261, entregarían el fichero: GR2261_MarisaPedro.zip.
- La estructura de los ficheros entregados deberá estructurarse en 3 ó 4 directorios, uno por cada apartado.