

por Javier Carrasco

Entornos de desarrollo

Bloque 2

Tema 5: Concepto de clase y objeto. Diagrama de clases. Software específico. Otros diagramas útiles

2.5.1. Concepto de clase y objeto

En el ámbito de la Programación Orientada a Objetos (POO) estos dos conceptos van unidos y muy ligados a este tipo de programación, en realidad son la base de la POO. Para lenguajes de programación como Java, son su esencia, la base sobre la cual se construye el lenguaje.

Debemos entender una **clase** como una plantilla que define la forma de un objeto, especificando los datos y el código que actuará sobre ellos. Por tanto, una **clase** es básicamente una serie de mapas, o esquemas, que especifican como crear un objeto.

Los **objetos** son instancias específicas de una clase. Hasta que un objeto de una clase no es creado, no existe representación física de ésta en memoria.

Veamos como sería el formato general de una clase en Java.

```
class nombredeclase {  
    // declarar variables de instancia  
    tipo var1;  
    tipo var2;  
    // ...  
    tipo varN;  
  
    // declarar métodos  
    tipo método1 (parámetros) {  
        // cuerpo del método  
    }  
  
    tipo método2 (parámetros) {  
        // cuerpo del método  
    }  
    // ...  
    tipo métodoN (parámetros) {  
        // cuerpo del método  
    }  
}
```

Otro lenguaje POO muy vinculado a estos dos conceptos es C++, veamos como sería el formato general de una clase en este lenguaje.

```
class nombredeclase {
    private:
        // declarar variables de instancia
        tipo var1;
        tipo var2;
        // ...
        tipo varN;

    public:
        // declarar métodos
        tipo método1 (tipo &p1, tipo &p2, ...);
        tipo método2 (tipo &p1, tipo &p2, ...);
        // ...
        tipo métodoN (tipo &p1, tipo &p2, ...);
};

tipo nombredeclase::método1(tipo &p1, tipo &p2) {
    a2 = a;
    b2 = b;
}
// ...
```

Vamos a ilustrar utilizando Java estos conceptos mediante un ejemplo clásico. Desarrollaremos una clase que contenga información sobre vehículos (coches, furgonetas y camiones).

Definición de la clase **Vehicle**.

```
class Vehicle {
    int passengers; // número de pasajeros
    int fuelcap; // capacidad de combustible en litros
    int kml; // consumo de combustible en kilómetros por litro
}
```

Para instanciar el objeto utilizando la clase **Vehicle** utilizaríamos la siguiente línea.

```
// crea un objeto Vehicle con el nombre de minivan
Vehicle minivan = new Vehicle();
```

Tras ejecutar esta instrucción, **minivan** se convierte en una instancia de **Vehicle**, teniendo así una realidad física en memoria. Siempre que se crea una instancia de una clase, se crea un objeto que contiene su propia copia de cada una de las variables de instancia creadas en la clase.

Veamos ahora como podemos utilizar la clase **Vehicle** con unas cuantas líneas de código.

```
class ExampleVehicle {
    public static void main(String[] args) {
        // crea un objeto Vehicle con el nombre de minivan
    }
}
```

```
Vehicle minivan = new Vehicle();
int autonomia;

// asignamos los valores a los campos de minivan
minivan.passengers = 5;
minivan.fuelcap = 20;
minivan.kml = 3;

/*
 * calculamos la autonomía del vehículo,
 * asumiendo el depósito lleno
 */
autonomia = minivan.fuelcap * minivan.kml;
System.out.println("El vehículo minivan puede llevar "
    + minivan.passengers + " pasajeros con una autonomía de "
    + autonomia + " kilómetros.");
}
}
```

En la salida por consola deberemos ver lo siguiente.

```
El vehículo minivan puede llevar 5 pasajeros con una autonomía de 60
kilómetros.
```

Si los pensamos un poco, vemos que podemos aplicar una pequeña refactorización sobre el código. ¿En qué sentido? El cálculo de la autonomía de un vehículo puede ser un método interesante para tener disponible en el esquema de la clase **Vehicle**, de forma que podamos conocer la autonomía de cualquier objeto **Vehicle** simplemente invocando dicho método. Quedaría de la siguiente forma.

```
class Vehicle {
    int passengers; // número de pasajeros
    int fuelcap; // capacidad de combustible en litros
    int kml; // consumo de combustible en kilómetros por litro

    /*
     * calculamos la autonomía del vehículo, asumiendo el
     * depósito lleno, desde aquí fuelcap y kml se
     * utilizan directamente sin el operador punto.
     */
    void autonomia() {
        System.out.println("La autonomía es de " + fuelcap*kml
            + " kilómetros.");
    }
}

class ExampleVehicle {
    public static void main(String[] args) {
        // crea un objeto Vehicle con el nombre de minivan
    }
}
```

```
Vehicle minivan = new Vehicle();

// asignamos los valores a los campos de minivan
minivan.passengers = 5;
minivan.fuelcap = 20;
minivan.kml = 3;

// llamamos al método que nos indica la autonomía
minivan.autonomia();
}
}
```

Como puede verse en este ejemplo hemos añadido funcionalidad a la clase **Vehicle** mediante el método **autonomia**. Visto esto, añadiremos algunos conceptos que debemos conocer cuando hablemos de clases y objetos.

2.5.1.1. Constructores

Hasta aquí, las variables de instancia se han asignado manualmente, pero este no es un enfoque que se utilice en el ámbito profesional, ya que es muy dado a producir errores.

Los **constructores** se utilizan para inicializar los objetos al crearlos. Tienen el mismo nombre que su clase y son sintácticamente similares a un método, pero no tienen un tipo de devolución explícito.

Se utilizan para asignar valores iniciales a las variables de instancia definidas por la clase o para realizar otros procedimientos de inicio que sean necesarios para crear el objeto correctamente.

Veamos un ejemplo de constructor para la clase **Vehicle**, también se le conoce como **constructor por defecto** o **sin parámetros**. Éste será invocado en el momento de la creación del objeto.

```
class Vehicle {
    int passengers; // número de pasajeros
    int fuelcap; // capacidad de combustible en litros
    int kml; // consumo de combustible en kilómetros por litro

    Vehicle(){ // Constructor por defecto
        passengers = 0;
        fuelcap = 0;
        kml = 0;
    }

    /*
     * calculamos la autonomía del vehículo, asumiendo
     * el depósito lleno, desde aquí fuelcap y kml se
     * utilizan directamente sin el operador punto.
     */
    void autonomia() {
        System.out.println("La autonomía es de " + fuelcap * kml
            + " kilómetros.");
    }
}
```

```
}  
}  
  
class ExampleVehicle {  
    public static void main(String[] args) {  
        // crea un objeto Vehicle con el nombre de minivan  
        Vehicle minivan = new Vehicle();  
  
        minivan.autonomia();  
    }  
}
```

La salida que obtendremos por consola cambia, ya que no hemos inicializado las variables de instancia manualmente, sino mediante el constructor por defecto, estableciendo todas las variables a 0 como hemos puesto en el código.

La autonomía es de 0 kilómetros.

Pero, suele ocurrir que en ocasiones necesitemos que un constructor pueda aceptar uno o varios parámetros. Esto podrá hacerse añadiendo parámetros como si de un método cualquiera se tratase.

```
Vehicle(int p, int f, int k){ // Constructor con parámetros  
    passengers = p;  
    fuelcap = f;  
    kmL = k;  
}
```

La creación del objeto utilizando el constructor con parámetros sería como sigue.

```
Vehicle minivan = new Vehicle(5, 20, 3);
```

Hay que saber que podemos tener tantos constructores como nos haga falta, todos llevarán el mismo nombre, pero eso no es un problema, la diferencia entre ellos radicará en el número y tipo de parámetros que tengan. En este ejemplo ya tenemos un constructor con tres parámetros enteros, ya no podría existir otro, pero si uno con tres parámetros, dos enteros y un decimal, por ejemplo.

2.5.1.2. Modificadores de acceso

El control de acceso es algo que debemos tener muy en cuenta a la hora de diseñar una clase. El control de acceso a miembros se hará utilizando tres modificadores de acceso: **public**, **private** y **protected**, utilizado en casos de herencia, ésto se verá más adelante.

Si modificamos un miembro de una clase con el modificador **public**, cualquier código del programa podrá acceder a ese miembro, incluidos los miembros definidos en otras clases.

Si especificamos un miembro de una clase como **private**, sólo podrán acceder a este miembro otros miembros de su clase. Los métodos de otras clases no podrán acceder a un miembro *private* de otra clase.

```
class Vehicle {
    private int passengers; // número de pasajeros
    private int fuelcap; // capacidad de combustible en litros
    private int kml; // consumo de combustible en kilómetros por litro

    Vehicle(){ // Constructor por defecto
        passengers = 0;
        fuelcap = 0;
        kml = 0;
    }

    Vehicle(int p, int f, int k){ // Constructor con parámetros
        passengers = p;
        fuelcap = f;
        kml = k;
    }

    /*
     * calculamos la autonomía del vehículo, asumiendo
     * el depósito lleno, desde aquí fuelcap y kml se
     * utilizan directamente sin el operador punto.
     */
    private int autonomia() {
        return fuelcap * kml;
    }

    public void datVehicle() { // Muestra los datos del vehículo
        System.out.println("El vehículo puede transportar " + passengers
            + " y su autonomía es de " + autonomia() + " kilómetros.");
    }
}

class ExampleVehicle {
    public static void main(String[] args) {
        // crea un objeto Vehicle con el nombre de minivan
        Vehicle minivan = new Vehicle(5, 20, 3);

        // minivan.autonomia(); este método no es accesible.
        minivan.datVehicle();
    }
}
```

La salida por consola de este último ejemplo sería la siguiente.

El vehículo puede transportar 5 y su autonomía es de 60 kilómetros.

2.5.2. Diagrama de clases

Un diagrama de clases se utiliza para visualizar las relaciones existentes entre todas las clases que intervienen en el sistema, representando además sus estructuras y comportamientos para con el resto.

El diagrama de clases no representa los comportamientos temporales entre clases, para ello se suele utilizar el diagrama de transición de estados.

Elementos que componen los diagramas de clases.

Clase

Es la unidad básica y encapsula toda la información referente a un objeto. Mediante ella se podrá modelar el entorno de estudio.

<Nombre Clase>	En la parte superior colocaremos el nombre que identifique a la clase.
<Atributos>	La parte intermedia contiene los atributos, o variables de instancia, de la clase, recordar que estos pueden ser <i>public</i> (+), <i>private</i> (-) o <i>protected</i> (#).
<Métodos>	Por último, la parte inferior contendrá los métodos que permitirán al objeto interactuar con su entorno.

Veamos un ejemplo representado mediante **Dia**.

CuentaCorriente
-balance: int
+ingresar(in cantidad:float): void
+transferir(in cantidad:int): bool
+consultar(): float

El diseño de esta clase se corresponde a una cuenta corriente que posee una característica balance que almacena el capital de la cuenta, si nos fijamos en el signo que precede al atributo podemos ver que su ámbito es privado. Además, la clase puede realizar las operaciones de ingreso, transferencia y consultar el balance de la cuenta.

Atributos

Los atributos, características o variables de instancia, de un clase podrán ser de tres tipos, dependiendo del nivel de visibilidad.

- **public (+):** atributo visible, éste podrá accederse tanto desde dentro de la clase como desde fuera de la misma.
- **private (-):** atributo privado, únicamente podrá ser accesible desde dentro de la clase, sólo podrán utilizarlo los métodos de la propia clase.
- **protected (#):** atributo protegido, no podrá ser accedido desde fuera de la clase, pero si podrá ser accedido por métodos de la propia clase y por métodos de subclases derivadas (herencia).

Métodos


Los métodos, u operaciones, de una clase muestran la forma en como ésta interactuará con el entorno, además podrán ser de tres tipos, dependiendo del nivel de visibilidad.

- **public (+):** método visible, éste podrá utilizarse tanto desde dentro de la clase como desde fuera de la misma.
- **private (-):** método privado, únicamente podrá ser utilizado desde dentro de la clase, podrán utilizarlo otros métodos de la propia clase.
- **protected (#):** método protegido, no podrá ser utilizado desde fuera de la clase, pero si podrá ser usado otros métodos de la propia clase y por métodos de subclases derivadas (herencia).

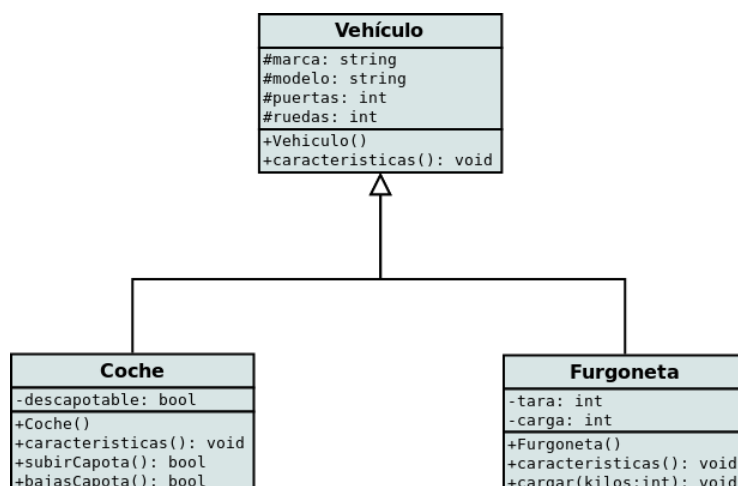
Relaciones entre clases

Ya tenemos claro, más o menos, el concepto de clase, veamos ahora como pueden relacionarse entre ellas. Expliquemos antes de empezar el concepto de **cardinalidad** de relaciones.

En UML, la cardinalidad entre las relaciones indicará el grado y nivel de dependencia, se anotará en cada extremo de la relación y podrán ser:

- **uno a muchos:** 1..* (1..n)
- **0 a mucho:** 0..* (0..n)
- **número fijo:** m (m indica el número)
- **Herencia (Especialización/Generalización):** 

Esto indicará que una **Subclase** podrá heredar los atributos y métodos de una **Super Clase**. Es más, una subclase podrá tener sus propios atributos y métodos además de los visibles de la super clase (*public* y *protected*).




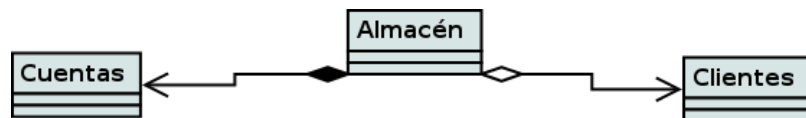
En este ejemplo podemos ver como las clases **Coche** y **Furgoneta** heredan de **Vehículo**, podemos ver como **Coche** tiene las características de Vehículo (marca, modelo, puertas, ruedas) y además tiene la característica propia **descapotable**. Lo mismo ocurre con **Furgoneta**, salvo que tiene sus propias características como **tara** y **carga**.

Agregación

En ocasiones necesitaremos modelar objetos complejos, y no basta con los tipos de datos básicos. Cuando se necesita componer objetos que son instancias de clases definidas por el desarrollador, tenemos dos posibilidades.


- **Por valor:** es una relación estática, el tiempo de vida del objeto incluido está condicionado por el tiempo de vida del que lo incluye, es decir, si se elimina el que lo incluye, se borra el incluido. Se conoce como **composición** (rombo relleno).
- **Por referencia:** es una relación dinámica, el tiempo de vida del objeto incluido es independiente del que lo incluye. Se conoce como **agregación** (rombo vacío).

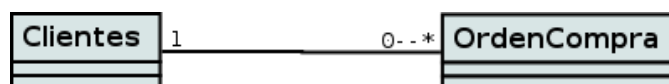
Este tipo de relación se representa por , donde la flecha indica la navegabilidad del objeto referenciado. Si no existiese esta particularidad se eliminaría la flecha. Veamos un ejemplo.



En este ejemplo podemos ver que **Almacén** tiene **Clientes** y **Cuentas**, podemos ver que los rombos van en el objeto que posee las referencias. Cuando se destruye el objeto Almacén también serán destruidos los objetos Cuentas asociados, pero no se verán afectados los objetos Clientes asociados.

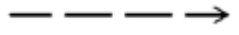
Asociación

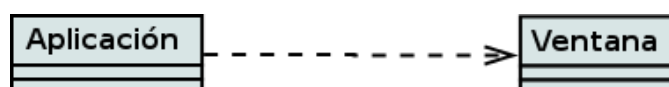
Esta relación entre clases permite asociar objetos que colaboran entre sí. No es una relación fuerte, es decir, el tiempo de vida de un objeto no depende del otro. Se representan mediante .



En este ejemplo se puede ver como un **Cliente** puede tener asociadas muchas **Ordenes de compra** o ninguna, pero una orden de compra únicamente puede tener asociado un cliente.

Dependencia o instanciación (uso)

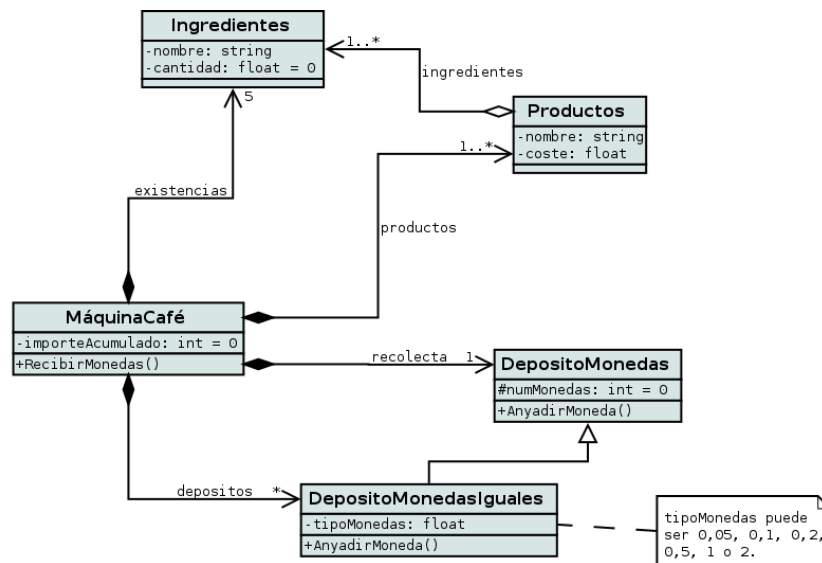
Esta es una relación de uso, es decir, una clase usa a la otra, es la relación más básica y comparada con el resto, es la relación más débil. Se representa mediante , el sentido de la flecha indicará que clase utiliza a la otra.



Este ejemplo escenifica la dependencia que tiene una clase de otra. La clase **Aplicación** instancia una **Ventana**, la creación del objeto ventana viene condicionada por la instanciación desde el objeto aplicación.

Es importante resaltar que el objeto creado (ventana) no se almacena dentro del objeto que lo crea (aplicación).

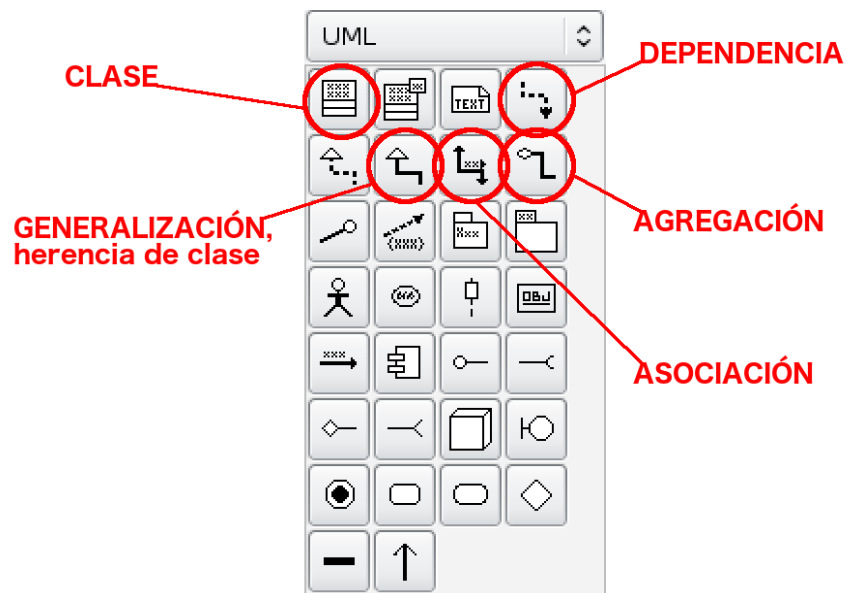
Para terminar este punto, veamos a continuación como ejemplo el diagrama de clases perteneciente a una máquina de café.



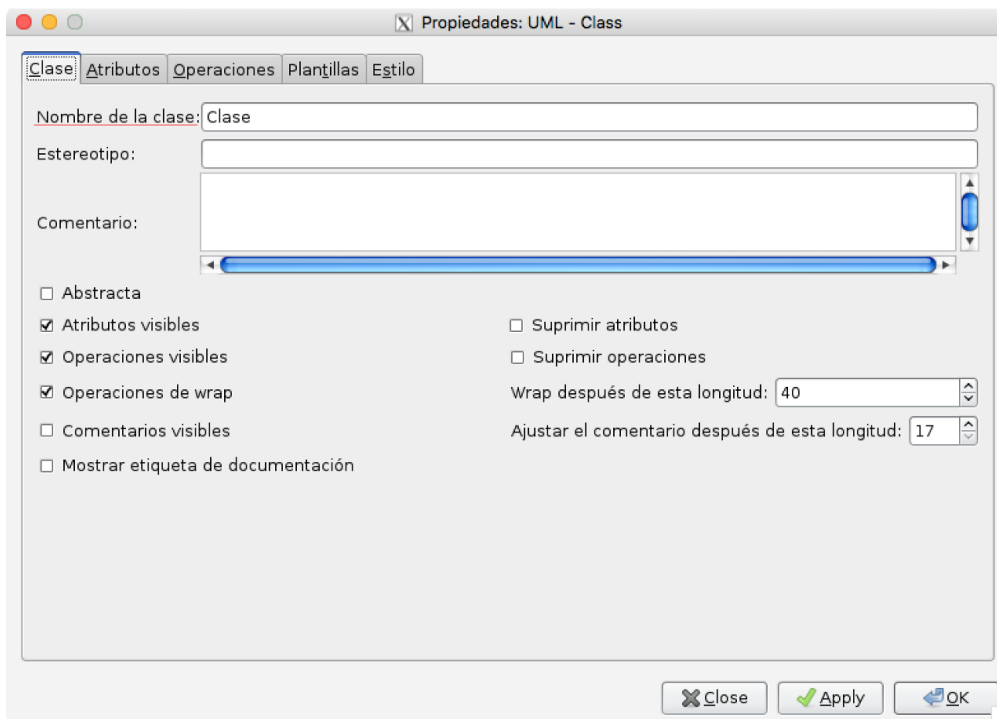
2.5.3. Software específico

En lo relativo al **software** específico, para crear diagramas de clases volvemos a centrarnos en **Dia** como ya se vio en temas anteriores. Llegados a este punto, ya deberíamos tener cierta soltura, así que vamos a ver los elementos que intervienen con mayor frecuencia en el diseño de un diagrama de clases.

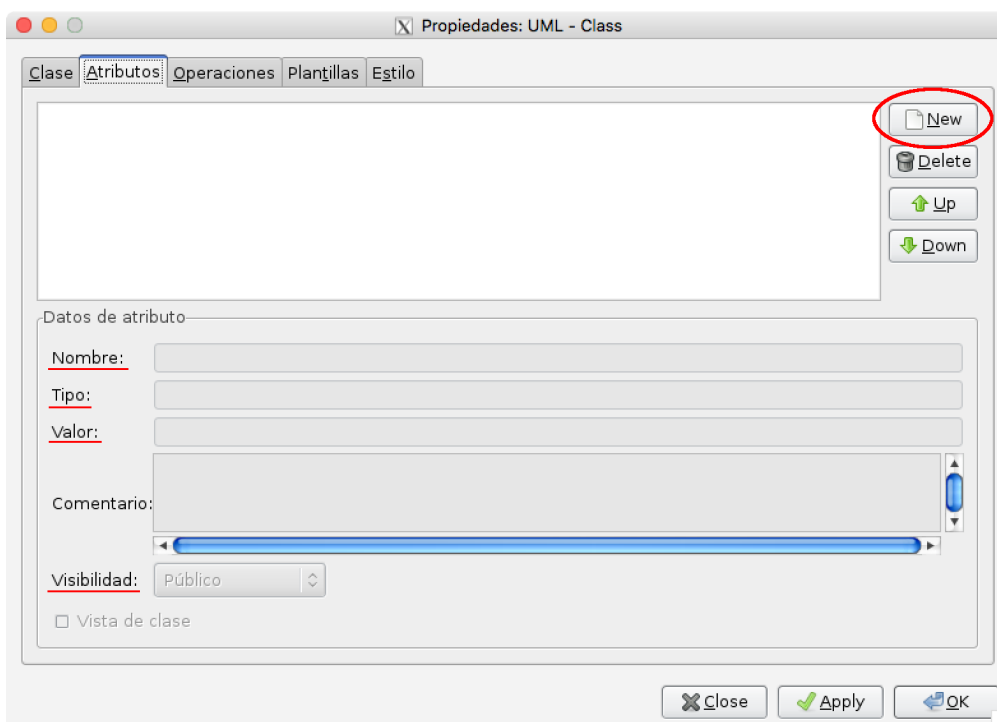
Los diagramas de clases forman parte de UML, por tanto debemos tener seleccionada la opción **UML** en el desplegable que encontramos bajo las herramientas en la parte izquierda de Dia.



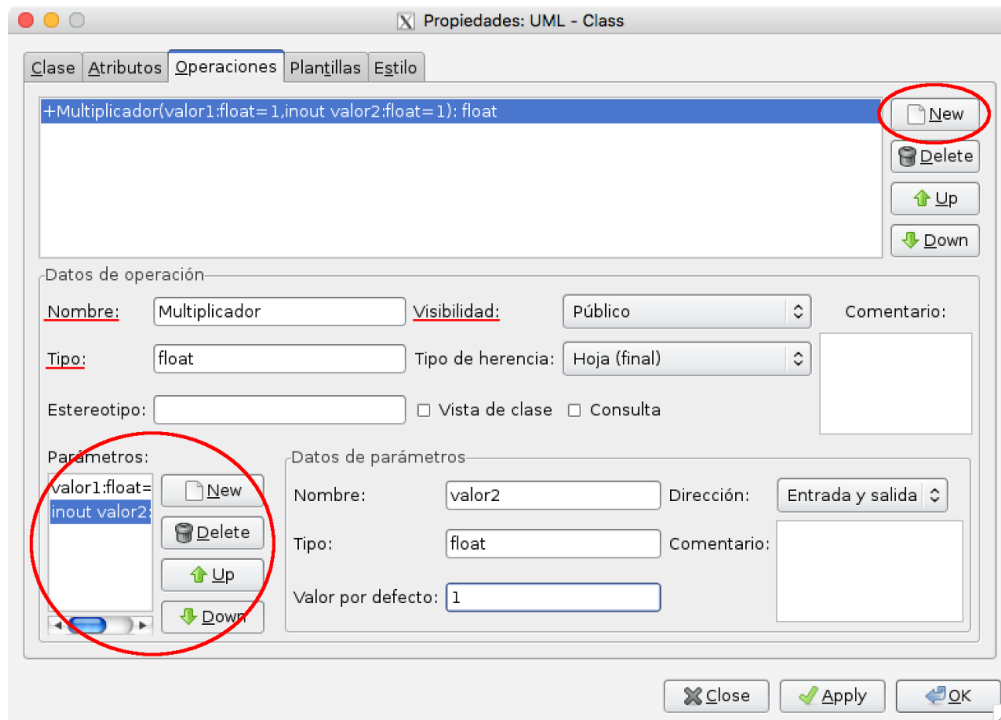
Cuando añadimos una clase nueva la debemos configurar, indicar su nombre, sus propiedades y sus métodos básicamente. Para ello, una vez ubicada en la zona de trabajo de **Dia** hacemos doble clic sobre ella y accedemos a sus propiedades.



En la pestaña **Clase** podemos poner el nombre de la clase, básicamente es lo más importante de esta pestaña, el resto suele dejarse tal cual.



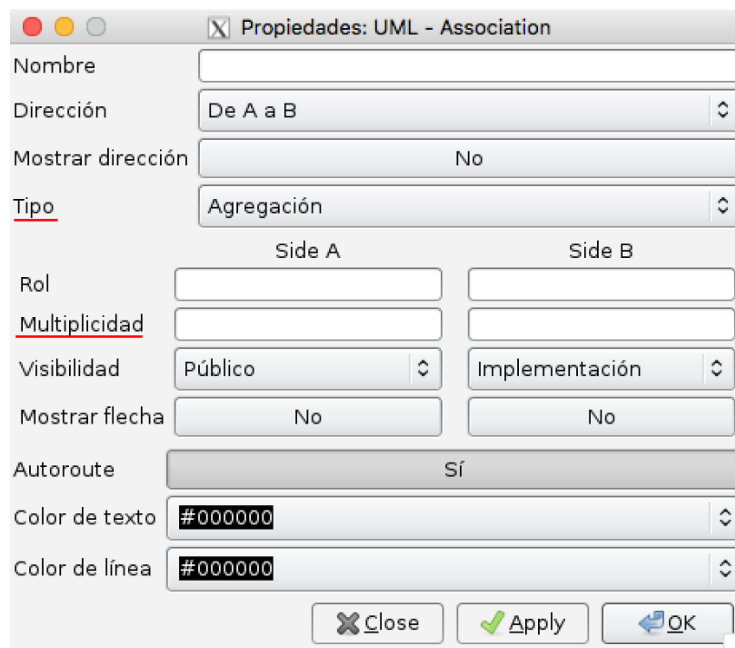
La siguiente pestaña, **Atributos**, nos permite añadir los atributos o propiedades de la clase, para ello, bastará con ir pulsando el botón **New** cada vez que queramos añadir uno nuevo. A continuación rellenamos los campos **Nombre** (nombre del atributo), **Tipo** (entero, cadena, etc), **Valor** (opcional) y seleccionar la **Visibilidad** (público, privado, protegido).



La pestaña **Operaciones** nos permitirá añadir los métodos de la clase, funciona exactamente igual que los atributos, con una diferencia, en los métodos podemos añadir **parámetros** a los métodos.

Destacar que la información que añadimos en los campos es de tipo texto, por lo que podemos poner cualquier información, por ejemplo, en los campos **Tipo** podemos poner *float*, *decimal*, *entero*, *string*, *bool*, *supercombo*, *ventilador*, etc, lo que prefiramos.

El siguiente cuadro de diálogo lo encontramos asociado a las relaciones de **Agregación** y **Asociación**.



La diferencia radica en el campo **Tipo**, si es una asociación, este campo tendrá el valor *Nada* y la opción **Mostrar dirección** a *Sí*. En caso de ser una agregación, el campo **Tipo** tendrá el valor *Agregación*, por referencia. Si lo cambiamos a *Composición* será por valor.

Otro campo interesante aquí es el de **Multiplicidad**, donde podremos indicar la cardinalidad en modo texto, como 1..*, n, 0..3, etc.

Ejercicios propuestos

2.5.3.1. Representa en un **diagrama de clases** la siguiente especificación:

Debemos representar las características y clases de los barcos (usaremos una muestra pequeña). Como características comunes de un **barco** nos quedaremos con su **nombre**, con el número de **anclas**, y el **material** de construcción (madera, metal, fibra de vidrio, etc) y su **eslora** (se mide en metros). Como tipos de barcos nos quedaremos con:

- **Portacontenedores**, deberemos registrar su capacidad en **TEU** (unidad de medida que equivale a un contenedor).
- Barcos **graneleros**, de los que registraremos el número de **escotillas** de carga (suelen ser un número impar).
- **Petroleros**, guardaremos si son o no de **doble casco** y su **capacidad** en toneladas.
- Barcos **pesqueros**, de los que guardaremos información sobre el **tipo de pesca** (comercial, artesanal o deportiva).

Añade los métodos necesarios para modificar las propiedades (*getters* y *setters*).

2.5.3.2. Representa en un **diagrama de clases** la siguiente especificación:

Supongamos que queremos desarrollar un videojuego, necesitamos personajes. Partiremos de la clase **personaje**, que tendrá como propiedades la **edad**, **altura**, **sexo** y nivel de **vida**. Según el papel del personaje en el juego, podrá ser:

- **Soldado**, cuya propiedad será el **arma** que porte (espada, arco, cañón, etc).
 - **Arquero**, sus propiedades serán, nivel de **vida**, tipo de **movimiento** y **daño**.
 - **Caballero**, sus propiedades serán, nivel de **vida**, tipo de **movimiento** y **daño**, nivel de vida de su **montura**.
- **Ciudadano**, de éstos, su propiedad será su **herramienta** (martillo, azada, legón, etc).
 - **Granjero**, sus propiedades serán, nivel de **vida**, tipo de **movimiento**, **trabajo** que debe desempeñar (recolectar, sembrar, etc) y el **nivel de trabajo**, que indicará lo que tardará en realizar una tarea.
 - **Minero**, sus propiedades serán, su nivel de **vida**, el tipo de **movimiento**, **trabajo** que deberá realizar (picar, derribar, extraer, etc) y su **nivel de cansancio**, tiempo que tardará en agotarse.

Añade los métodos necesarios para modificar las propiedades (*getters* y *setters*).

2.5.3.3. Se trata de un ejercicio de temas anteriores adaptado al tema en que nos encontramos.

Un **blog** tiene tres tipos de usuarios: administradores, editores y visitantes. Para ellos se guardarán el usuario y la contraseña. Los administradores pueden registrar a otros usuarios. Los editores pueden publicar **posts**, y los visitantes pueden comentar los **posts**. Además, existe otro tipo de usuario llamado autor, el cual es un subtipo de editor, que puede crear y publicar **posts**, pero siempre estará bajo supervisión de un editor.

Añade los **getters** y **setters** necesarios para cada clase y los métodos necesarios para añadir la funcionalidad requerida.

2.5.3.4. Intenta, utilizando el diagrama de clases de temas anteriores, la adaptación que se ha hecho para este tema.

Una organización cultural gestiona el préstamo de dos tipos de objetos: discos de música y libros. Para ambos objetos almacenamos información general, como identificación, título y autor. Con respecto a los libros, también almacenamos el número de páginas, y para los discos nos interesa la compañía discográfica. Existen muchos usuarios que vienen a esta organización, de los cuales almacenamos su ID (DNI) y su nombre. Pueden solicitar libros y discos de música (hasta 5 objetos simultáneamente), para lo cual almacenaremos la fecha de inicio y finalización del préstamo.

Añade los **getters** y **setters** necesarios para cada clase y los métodos necesarios para añadir la funcionalidad requerida.

2.5.4. Otros diagramas útiles

Diagrama de estructuras compuestas: Muestran la estructura interna de una clase.

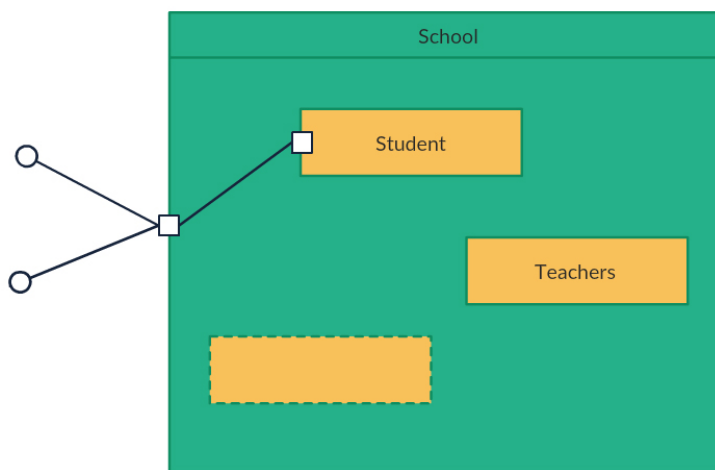
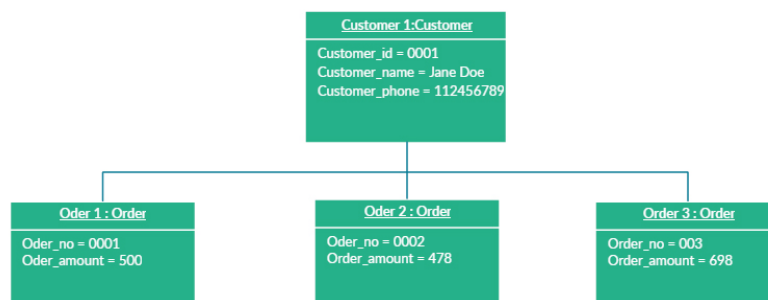


Diagrama de objetos: Similar al diagrama de clases pero con objetos con datos de ejemplo.



2.5.5. Para profundizar

Puedes ampliar información sobre este tema en los enlaces que tienes a continuación.

- [Java SE Development Kit 8 \(JDK\)](#).
- [Java 8](#). Herbert Schildt. Editorial Anaya.