

UT5 1 – UML. ELABORACIÓN DE DIAGRAMAS DE CLASES

| RESULTADOS DE APRENDIZAJE ASOCIADOS |
|---|
| 5. Genera diagramas de clases valorando su importancia en el desarrollo de aplicaciones y empleando las herramientas disponibles en el entorno. |
| CRITERIOS DE EVALUACIÓN |
| a) Se han identificado los conceptos básicos de la programación orientada a objetos |
| b) Se ha instalado el módulo del entorno integrado de desarrollo que permite la utilización de diagramas de clases |
| c) Se han identificado las herramientas para la elaboración de diagramas de clases. |
| d) Se ha interpretado el significado de diagramas de clases. |
| e) Se han trazado diagramas de clases a partir de las especificaciones de las mismas. |
| f) Se ha generado código a partir de un diagrama de clases. |
| g) Se ha generado un diagrama de clases mediante ingeniería inversa. |

UT5_1 – UML. ELABORACIÓN DE DIAGRAMAS DE CLASES

Índice de contenido

| | |
|--------------------------------------|----|
| 1.- Contenidos..... | 3 |
| 2.- Introducción | 3 |
| 3.- Diagramas de clases | 5 |
| 3.1- Clases | 6 |
| 3.2 – Atributos | 8 |
| 3.3 – Notas adjuntas..... | 10 |
| 3.4 – Métodos | 11 |
| 3.6 – Relaciones | 13 |
| 3.6.1 – Tipos de relaciones | 14 |
| 3.7 – Estereotipos | 31 |
| 5.- Referencias bibliográficas | 32 |

1.- Contenidos

El estudio de la programación orientada a objetos (POO) es fundamental en el desarrollo de aplicaciones modernas. Esta metodología se centra en representar entidades del mundo real como objetos con atributos y comportamientos. La creación de diagramas de clases es una herramienta esencial en este proceso, ya que permite visualizar la estructura y las relaciones entre los objetos de un sistema. En este contexto, es crucial comprender los conceptos básicos de la POO, instalar y utilizar las herramientas adecuadas para la elaboración de diagramas de clases, interpretar su significado y generar código a partir de ellos. Además, la ingeniería inversa facilita la comprensión de sistemas existentes mediante la generación de diagramas de clases a partir del código fuente.

2.- Introducción

La evolución de la programación ha marcado un cambio significativo a lo largo de la historia. En sus inicios, los programas se adherían a los principios de la **programación estructurada**, donde imperaba el lema "**divide y vencerás**". Incluso los romanos aplicaron esta técnica con su célebre "divide et impera".



Este enfoque consiste en descomponer el problema principal en módulos más pequeños, simplificando así su resolución mediante la creación de funciones que aborden estas subdivisiones.

Los programas que seguían este paradigma se centraban en la funcionalidad en lugar de en los datos, con el propósito claro para el programador de identificar cómo resolver el problema en cuestión.

Posteriormente, surgió la **Programación Orientada a Objetos (POO)**, revolucionando el panorama.

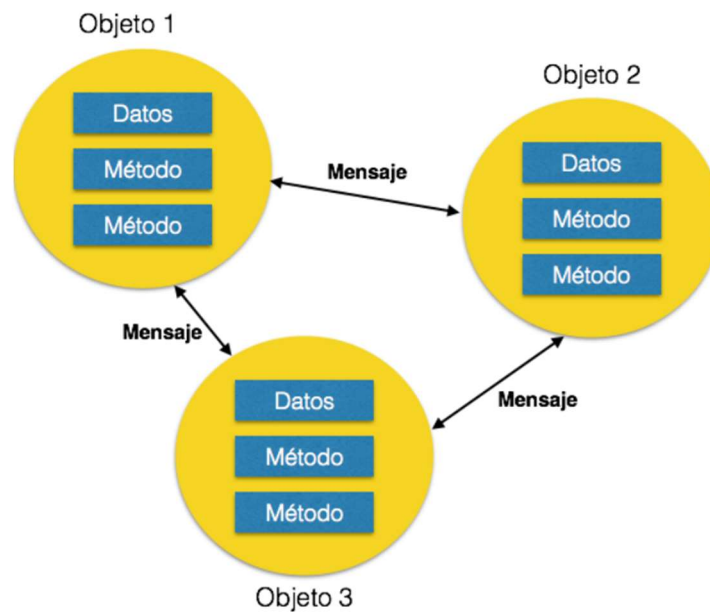


La orientación a objetos (OO) busca representar las partes o elementos del problema como entidades concretas. Este enfoque rompe con el paradigma anterior al basarse en la creación de objetos que poseen atributos (datos específicos del objeto) y métodos (comportamiento del objeto). Estos objetos interactúan entre sí, enviándose mensajes y respondiendo a las acciones del usuario.

De esta manera tan característica y diferente a la programación estructurada, funcionan los programas creados con la POO.

El ciclo de vida de un programa orientado a objetos es el siguiente:

- a) **Paso 1:** Creación de objetos conforme se necesiten.
- b) **Paso 2:** Interacción del usuario con el programa, donde los objetos intercambian mensajes entre sí. En ocasiones, al procesar información, pueden crearse y destruirse objetos.
- c) **Paso 3:** Eliminación de los objetos una vez que ya no son necesarios. Esta tarea puede ser responsabilidad del programador, incluyendo la destrucción del objeto en el código, o bien puede ser gestionada por un proceso del sistema, como el recolector de basura (*garbage collector*) en Java.



3.- Diagramas de clases

Los diagramas de clase representan la estructura estática del sistema, mostrando las clases del sistema, sus atributos, métodos y las relaciones entre ellas. Son fundamentales para comprender la arquitectura y la organización del sistema.

Las clases y objetos se representan comúnmente en diagramas utilizando la notación **UML** (*Unified Modeling Language*). Esta herramienta ofrece un conjunto de técnicas que facilitan la modelización, documentación y desarrollo de sistemas o aplicaciones orientadas a objetos. UML, concebido por los pioneros de la Programación Orientada a Objetos (POO) como *Grady Booch*, *Jim Rumbaugh* e *Ivar Jacobson*, se ha convertido en el estándar de la industria del software.



Hagamos una búsqueda rápida...

¿Qué diagramas UML existen? ¿Para qué sirve cada uno de ellos?



Curiosidad

Object Management Group (OMG)

Responsable, entre otros estándares, del UML, del inglés *Unified Modeling Language* (lenguaje de modelado unificado). Es un consorcio que se encarga de crear y cuidar estos estándares. Crea guías y especificaciones y está formado por varias organizaciones, las cuales tienen distintas funciones y privilegios.



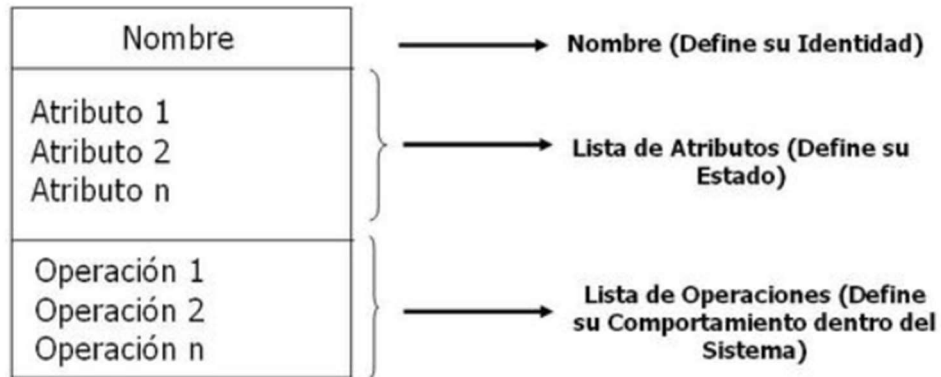
¿Por qué es necesario un estándar como el UML?

Cuando se desarrolla un sistema software, muchas veces, un equipo de personas (muchas o pocas dependiendo de la envergadura del proyecto) tienen que comunicarse durante su creación. Se generará un volumen ingente de documentación e información y **todo esto tiene que estar normalizado de alguna manera para que sea comprensible para todo el grupo**. El UML soluciona este problema y puede ser útil tanto en proyectos pequeños como en proyectos muy grandes.

3.1- Clases

Las clases, como se ha explicado previamente, representan categorías o tipos de objetos y se modelan siguiendo una estructura que se detalla a continuación:

Representación gráfica de una Clase en UML



Esta estructura consta de tres áreas distintas:

- En la **parte superior**, se encuentra el **nombre de la clase**, que en este ejemplo es "tensiómetro".
- En el **área central** se **definen los atributos de la clase**, que en este caso son la marca del tensiómetro, el modelo y la última medición realizada. Aunque podría contener más información según las necesidades específicas, para este caso se considera suficiente con estos tres atributos.
- Finalmente, en la **parte inferior** se **especifica el comportamiento de la clase**, es decir, **las acciones que puede realizar**. En este caso, el tensiómetro puede tomar la tensión, mostrar el resultado y registrar la última medición.



3.2 – Atributos

Los atributos de una clase suelen tener un tipo determinado. UML permite indicar el posible formato de un atributo especificando su tipo:

- **String o cadena de caracteres (texto).** Posibles valores: Med-Tenso, Avant Plus, etc.
- **Integer/int o número entero.** Posibles valores: 128, 32, -4, etc.
- **Float o número en coma flotante (número real).** Posibles valores: 3,141592; 1,98, etc.
- **Boolean o booleano.** Puede especificarse verdadero o falso.



Los atributos pueden tener valores por defecto o valores predeterminados.

Por ejemplo, puede determinarse que todos los tensiómetros que se generen tendrán como marca el valor *MedTenso* y que el valor por defecto para el atributo *últimaMedición* sea nulo.



Restricciones

Imagínese que, al tensiómetro anterior, quiere añadirse el atributo año de fabricación teniendo en cuenta que solamente ha sido fabricado durante los años 2017 y 2018.

Al atributo añoFabricación, podría añadirse una restricción indicando que solamente puede tomar los valores 2017 o 2018. En el diagrama UML, quedaría de la forma que se observa en la figura 5.6.

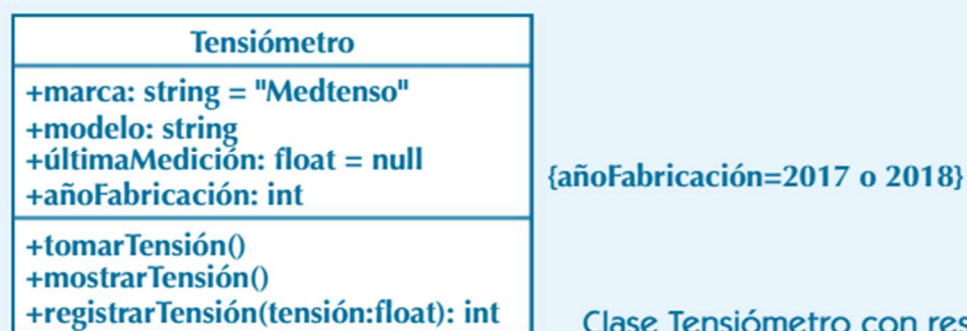


Figura 5.6

Clase Tensiómetro con restricciones.

Al crear el atributo se podrá indicar la **visibilidad** de este. Se distinguen los siguientes tipos:

- **public:** el atributo será público, visible tanto dentro como fuera de la clase. Sería accesible desde todos los lugares.



Se representa con un signo **+**

- **private:** el atributo sólo será accesible desde dentro de la clase (sólo sus métodos pueden acceder al atributo).



Se representa con un signo **-**

- **protected:** el método puede ser accedido por métodos de la clase además de los métodos de las subclases que deriven.

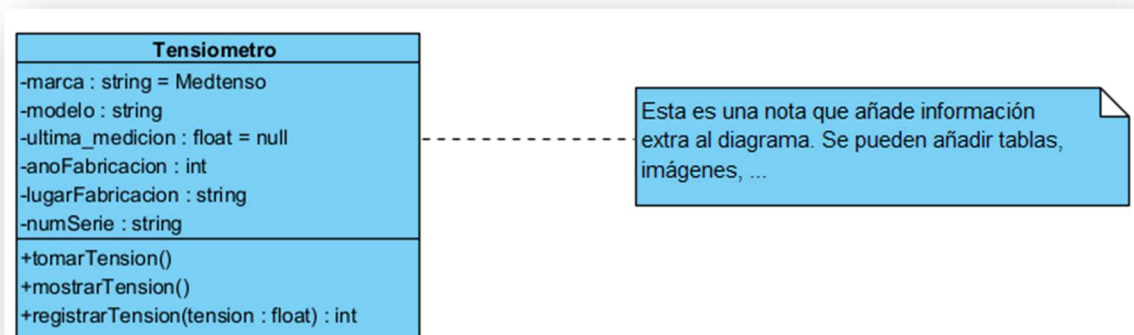


Se representa con un signo **#**

3.3 – Notas adjuntas

En ciertas ocasiones, es preciso agregar información adicional a la clase. Supongamos que nuestros tensiómetros poseen un número de serie y un lugar de fabricación. El número de serie sigue un formato específico que varía según el modelo, y el lugar de fabricación difiere según dicho modelo.

Esta información adicional o aclaratoria se incluirá en notas adjuntas.



Las notas adjuntas pueden llevar incluso imágenes y va a permitir al analista incluir información extra en el diagrama de clases que, luego, habrá que tener en cuenta en posteriores fases como la de codificación.



¡Paremos un momento y hagamos una prueba!

Haz clic en el siguiente enlace:

<https://online.visual-paradigm.com/es/>

Ahora sigue estos pasos:

- 1) **Crea un nuevo diagrama de clases**
- 2) **Diseña la clase de la imagen anterior y añade una nota como la del ejemplo.**




3.4 – Métodos

Las operaciones o métodos de una clase representan su aspecto dinámico. Al igual que con los atributos, se suele seguir la convención **camelCase**, y en ocasiones, pueden recibir una serie de parámetros y retornar un valor. Este valor se devuelve una vez que las acciones correspondientes han sido completadas.

Por ejemplo, el método **registrarTensión()** acepta un parámetro de tipo float llamado "tensión" y devuelve un valor entero (integer).

```
+registrarTension(tension : float) : int
```

Al igual que los atributos, en los métodos se podrá indicar la visibilidad de la siguiente manera:

- **public:** el atributo será público, visible tanto dentro como fuera de la clase. Sería accesible desde todos los lugares.
 Se representa con un signo **+**
- **private:** el atributo sólo será accesible desde dentro de la clase (sólo sus métodos pueden acceder al atributo).
 Se representa con un signo **—**
- **protected:** el método puede ser accedido por métodos de la clase además de los métodos de las subclases que deriven.
 Se representa con un signo **#**

Resumen...

| Marcador /signo | Visibilidad | Descripción |
|-----------------|-------------|--|
| + | Público | Todas las clases pueden ver la información. |
| — | Privado | La información está oculta a todas las clases fuera de la partición. |
| # | Protegido | La clase secundaria puede acceder a la información heredada de la clase principal. |



Hagamos un ejercicio rápido

En la vida real, el analista o desarrollador de software tiene una serie de entrevistas con el cliente y, de esas entrevistas, tiene que extraer información:

"El cliente comenta que, al buscar un libro en la aplicación de la biblioteca, quiere ver los detalles del libro como el título, autor y año de

publicación, y además, si el libro está disponible, poder reservarlo directamente desde la app."

En esta frase, el cliente está describiendo las características deseadas de un sistema de biblioteca, y de ella se pueden extraer las siguientes conclusiones:



Piensa por un momento... y vemos la solución:

- "**Libro**" sería una clase, ya que es un objeto con propiedades y comportamientos.
- "**Título**", "**autor**", "**año de publicación**" y "**disponibilidad**" son atributos de la clase Libro.
- "**ReservarLibro()**" podría ser un método asociado con la clase Libro, ya que describe una acción que se realiza sobre el libro.



Investiguemos...

¿Quién fue James Rumbaugh y cuáles fueron sus aportaciones al terreno de la informática?

3.6 – Relaciones

En un diagrama de clases, todas las clases están vinculadas entre sí mediante relaciones adecuadas. Estos enlaces ayudan al usuario a comprender a fondo la conexión entre diferentes entidades.

En UML, estos vínculos se describen mediante **asociaciones**, de igual modo que los objetos se describen mediante clases.

Las asociaciones tienen:

- Un **nombre**. Éste es un reflejo de los elementos de la asociación
- Una cardinalidad o **multiplicidad**. Representa el número de instancias de una clase que se relaciona con las instancias de otra clase diferente.

Esta cardinalidad es similar a la multiplicidad utilizada en el modelo Entidad-Relación

Se pueden especificar las multiplicidades mínimas y máximas de cada extremo. Para hacerlo, se utiliza la siguiente notación:

| Notación | Cardinalidad/Multiplicidad |
|----------|----------------------------|
| 0..1 | Cero o una vez |
| 1 | Una y sólo una vez |
| * | De cero a varias veces |
| 1..* | De una a varias veces |
| 0..* | De cero a varias veces |
| M..N | Entre M y N veces |
| N | N veces |

Ejemplo:



A continuación, se explican los diferentes tipos de relaciones existentes.

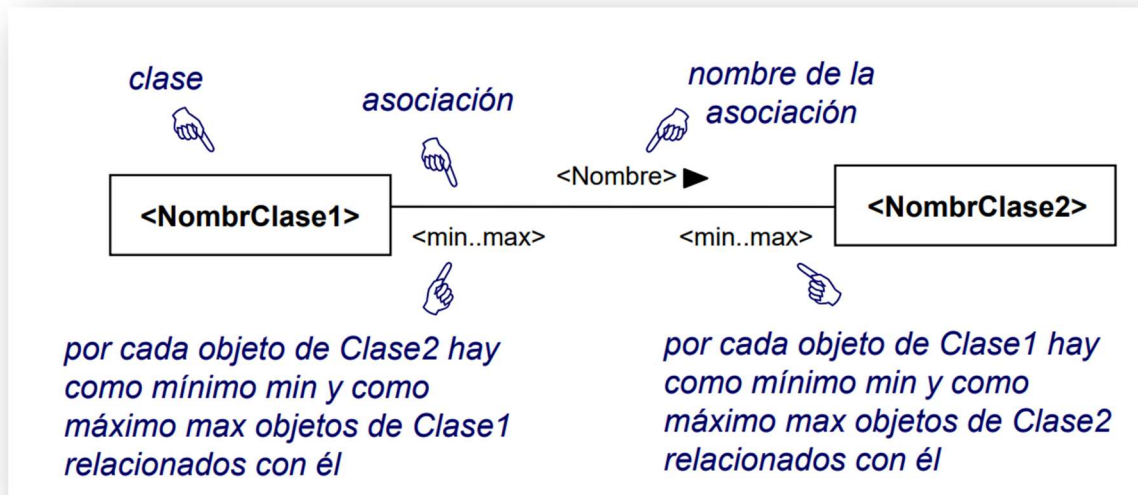
3.6.1 – Tipos de relaciones

Se distinguen los siguientes tipos de relaciones:

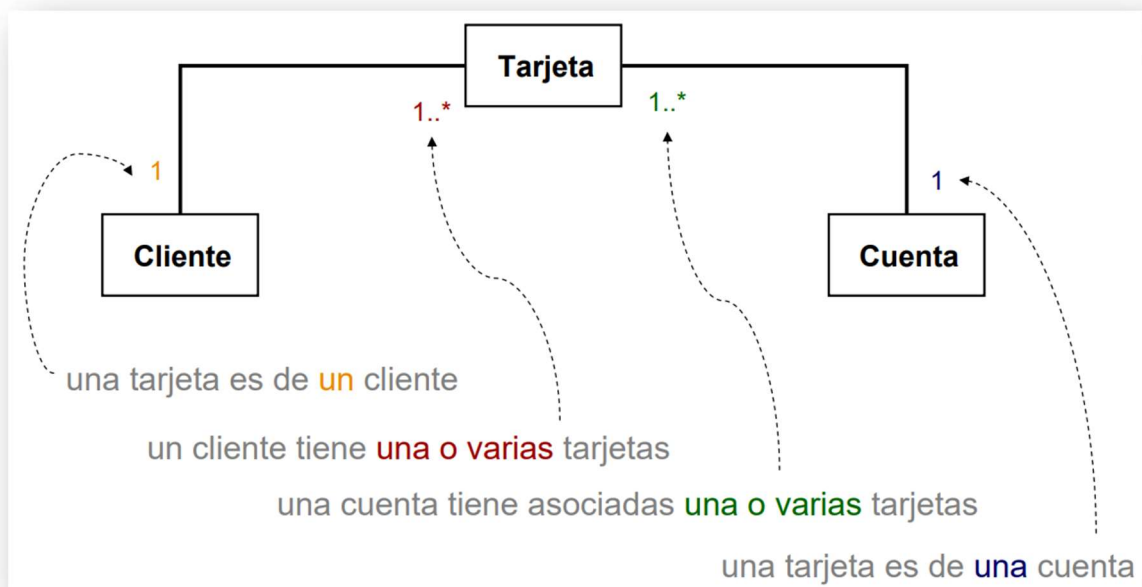
ASOCIACIÓN

Describe una relación genérica entre dos clases.

Representación gráfica:



Ejemplo:



Restricciones:

- Dentro de una misma clase, no se pueden repetir nombres de atributos
- La multiplicidad mínima no puede ser negativa
- La multiplicidad máxima tiene que ser mayor o igual que la mínima

Una asociación puede ser **Bidireccional** o **Unidireccional**, dependiendo de si ambas conocen la existencia la una de la otra o no.

Si se convierten a Java dos clases unidas por una asociación **Bidireccional**, cada una de las clases tendrá un objeto o un set de objetos, dependiendo de la multiplicidad entre ellas.

Ejemplos:

Bidireccional con cardinalidad 0..1 o 1



```

public class Cliente{
    private String nombre;
    public CtaCte cuenta;

    public Cliente(String n){
        nombre=n;
    }

    public void agregarCuenta(CtaCte c){
        cuenta=c;
    }
}
  
```

```

public class CtaCte{
    private double saldo;
    public Cliente dueño;

    public CtaCte(double s, Cliente d){
        saldo=s;
        dueño=d;
    }
}
  
```


Direccional con cardinalidad 0..1 o 1



```

public class Usuario{
    private String nombre;
    public Clave clave;

    public Usuario(String n, Clave c){
        nombre=n;
        clave=c;
    }
}
    
```

```

public class Clave{
    private int codigo;

    public Clave(int c){
        codigo=c;
    }
}
    
```

Bidireccional con cardinalidad n



```

public class Persona{
    private String nombre;
    public ArrayList<Perro> mascotas = new ArrayList();

    public Persona(String n){
        nombre=n;
    }

    public void agregarMascota(Perro p){
        mascotas.add(p);
    }
}
    
```

```

public class Perro{
    private String nombre;
    public Persona propietario;

    public Perro(String n){
        nombre=n;
    }

    public void asignarPropietario(Persona p){
        propietario=p;
    }
}
    
```

Bidireccional con cardinalidad n



```

public class Diputado{
    public ArrayList<Ley> voto = new ArrayList();

    public void agregarLey(Ley ley){
        voto.add(ley);
    }
}
    
```

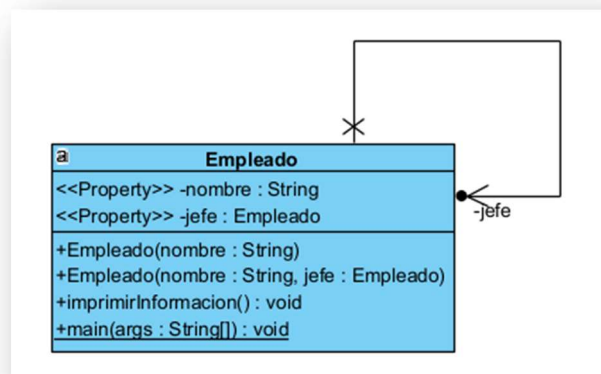
```

public class Ley{
    public ArrayList<Diputado> fueVotadaPor = new ArrayList();

    public void agregarDiputado(Diputado dip){
        fueVotadaPor.add(dip);
    }
}
    
```

Una clase puede asociarse consigo misma creando una **asociación reflexiva**, similares a las asociaciones reflexivas del modelo Entidad/Relación.

Ejemplo:



El código asociado sería el siguiente:

```

public class Empleado {
    private String nombre;
    private Empleado jefe; // La referencia al jefe, que
    también es un empleado.

    // Constructor para un empleado sin jefe (podría ser
    el jefe máximo).
    public Empleado(String nombre) {
        this.nombre = nombre;
        this.jefe = null; // No tiene jefe.
    }

    // Constructor para un empleado con un jefe.
    
```

```
public Empleado(String nombre, Empleado jefe) {
    this.nombre = nombre;
    this.jefe = jefe;
}

// Métodos getters y setters.
public String getNombre() {
    return nombre;
}

public void setNombre(String nombre) {
    this.nombre = nombre;
}

public Empleado getJefe() {
    return jefe;
}

public void setJefe(Empleado jefe) {
    this.jefe = jefe;
}

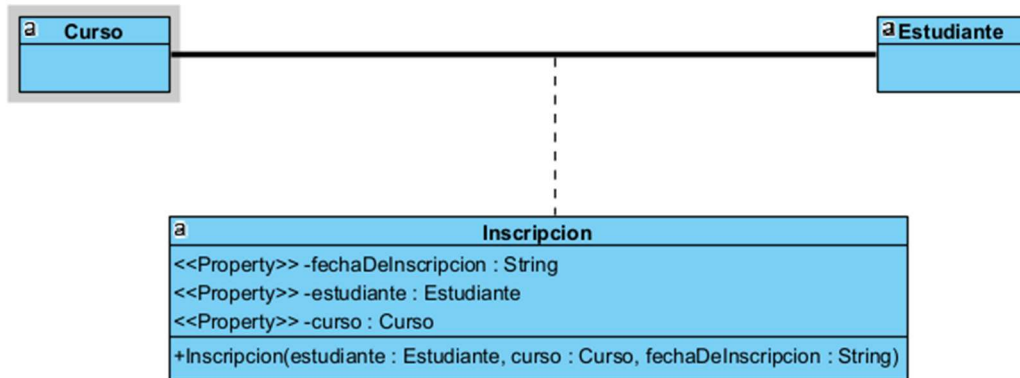
public void imprimirInformacion() {
    if (jefe != null) {
        System.out.println(nombre + " es un empleado
bajo el mando de " + jefe.getNombre() + ".");
    } else {
        System.out.println(nombre + " es el jefe
máximo, no tiene jefe.");
    }
}

public static void main(String[] args) {
    // Creación de empleados.
    Empleado jefe = new Empleado("Carlos"); // Carlos
es el jefe máximo.
    Empleado empleado1 = new Empleado("Ana", jefe);
// Ana reporta a Carlos.
    Empleado empleado2 = new Empleado("Luis", jefe);
// Luis también reporta a Carlos.

    // Impresión de la información de los empleados.
    jefe.imprimirInformacion();
    empleado1.imprimirInformacion();
    empleado2.imprimirInformacion();
}
}
```

CLASE ASOCIACIÓN

Una clase de asociación en UML es un tipo de clase que modela una relación entre otras clases y tiene sus propios atributos y métodos, añadiendo información adicional y comportamientos a esa relación.



El código en JAVA sería el siguiente:

```

public class Estudiante {
    private String nombre;

    public Estudiante(String nombre) {
        this.nombre = nombre;
    }

    // Getters y setters
    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
}

```

```

public class Curso {
    private String nombre;

    public Curso(String nombre) {
        this.nombre = nombre;
    }
}

```

```
// Getters y setters
public String getNombre() {
    return nombre;
}

public void setNombre(String nombre) {
    this.nombre = nombre;
}
}
```

```
public class Inscripcion {
    private Estudiante estudiante;
    private Curso curso;
    private String fechaDeInscripcion;

    public Inscripcion(Estudiante estudiante, Curso
curso, String fechaDeInscripcion) {
        this.estudiante = estudiante;
        this.curso = curso;
        this.fechaDeInscripcion = fechaDeInscripcion;
    }

    // Getters y setters
    public Estudiante getEstudiante() {
        return estudiante;
    }

    public void setEstudiante(Estudiante estudiante) {
        this.estudiante = estudiante;
    }

    public Curso getCurso() {
        return curso;
    }

    public void setCurso(Curso curso) {
        this.curso = curso;
    }

    public String getFechaDeInscripcion() {
        return fechaDeInscripcion;
    }

    public void setFechaDeInscripcion(String
fechaDeInscripcion) {
```

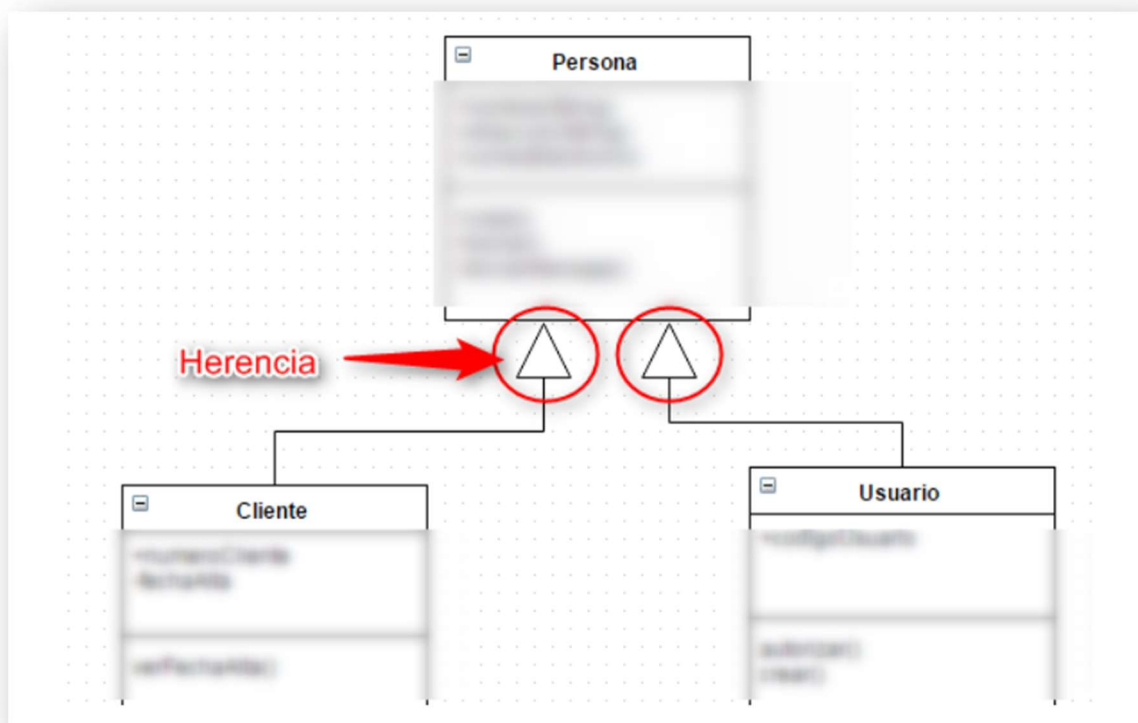
```

        this.fechaDeInscripcion = fechaDeInscripcion;
    }
}
    
```

Tutorial sobre cómo crear una clase asociación en Visual Paradigm
<https://knowhow.visual-paradigm.com/uml/association-class/>

HERENCIA

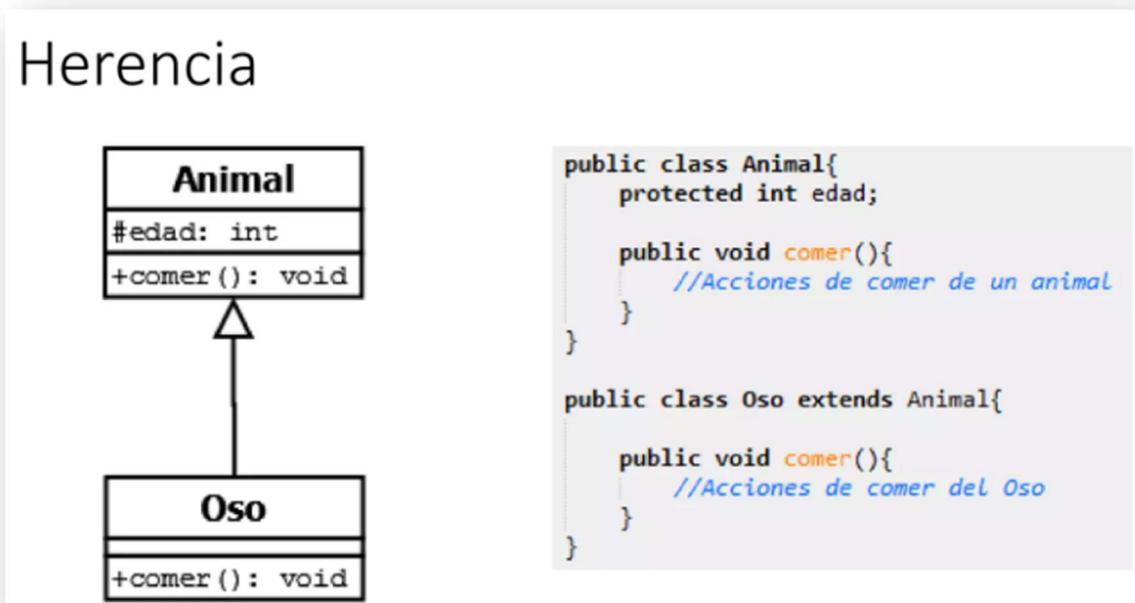
La herencia en los diagramas de clases de UML se utiliza para representar la relación "es un/a" entre clases, mostrando cómo una clase (subclase o clase derivada) puede heredar atributos y métodos de otra clase (superclase o clase base). Esta relación se ilustra mediante una línea que conecta las clases involucradas, con un triángulo relleno en el extremo que apunta hacia la superclase.



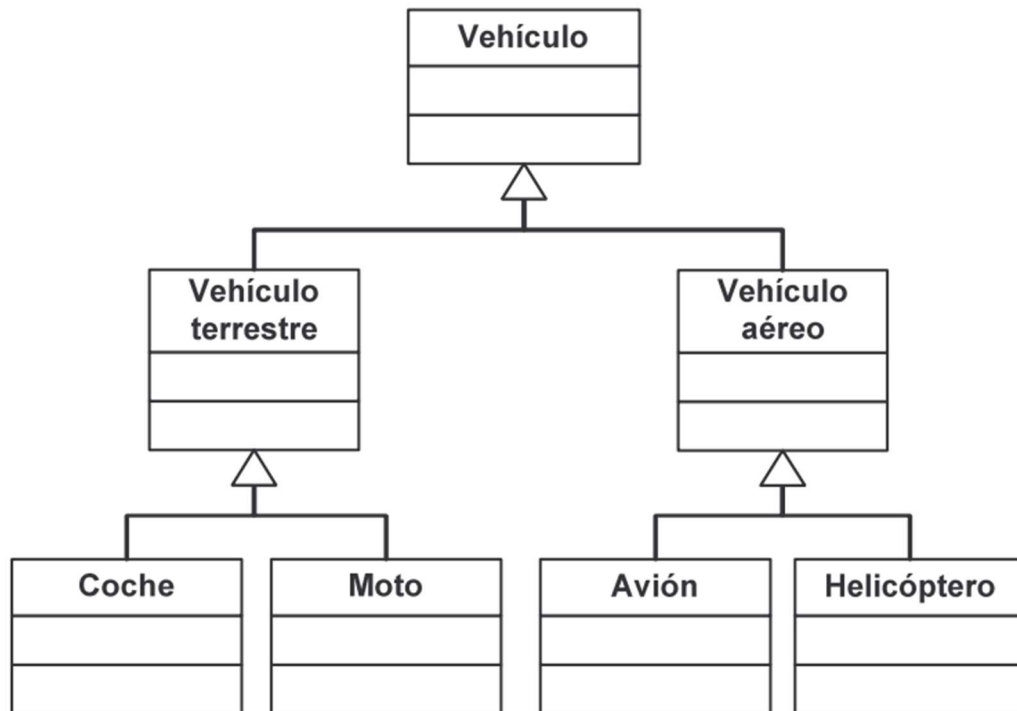
Características principales de la herencia en UML:

- **Reutilización:** La subclase hereda todos los atributos y métodos de la superclase, lo que promueve la reutilización del código y reduce la redundancia.
- **Polimorfismo:** La subclase puede sobrescribir métodos de la superclase, proporcionando su propia implementación. Esto permite el uso de objetos de la subclase donde se esperan objetos de la superclase.
- **Jerarquía:** La herencia establece una jerarquía de clases que refleja la relación "es un/a", facilitando la organización y el entendimiento de las relaciones entre las clases.

Ejemplo:



Un ejemplo de jerarquía algo más compleja:



COMPOSICIÓN

Un objeto puede estar compuesto por otros objetos, situación en la cual se establece una forma de asociación conocida como **composición**. Esta vinculación se da entre un objeto complejo y los objetos que lo conforman, sus componentes. Dentro de la composición, distinguimos dos variantes según la intensidad de la relación:

- **Composición (Fuerte):** En esta modalidad, la existencia de los objetos componentes está fuertemente ligada al objeto contenedor. Si el objeto contenedor deja de existir, sus componentes también son destruidos. Representa una dependencia total.


Ejemplo:

“Una persona **posee** órganos vitales”. Si la persona fallece, los órganos dejarían de funcionar. Esos objetos se eliminarían.



- **Agregación (Débil):** A diferencia de la composición fuerte, en la agregación, aunque un objeto contenedor agrupe a otros, la existencia de estos últimos no depende completamente del contenedor. Los componentes pueden existir independientemente del objeto agregador, indicando una asociación menos restrictiva.

Ejemplo:

“La persona usa un ordenador”. La computadora es independiente de la persona. Ese ordenador puede ser usado por otras personas y una persona podría utilizar otros ordenadores diferentes.

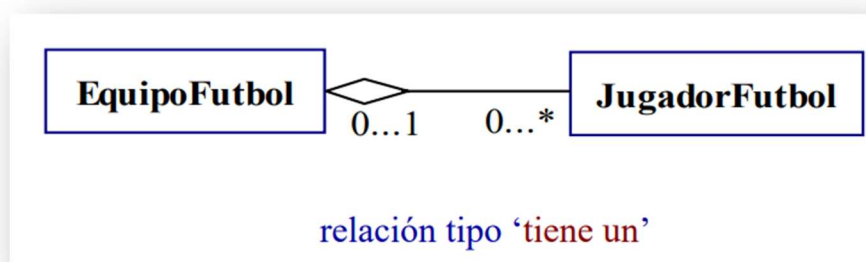


Resumen...

| | |
|---|---|
| Agregación: es una asociación que describe una relación entre un todo y sus partes de modo que las partes pueden existir por sí mismas |  |
| Composición: es una asociación que describe una relación entre un todo y sus partes de modo que las existencias de las partes se perciben como totalmente dependientes del todo |  |

Más ejemplos:

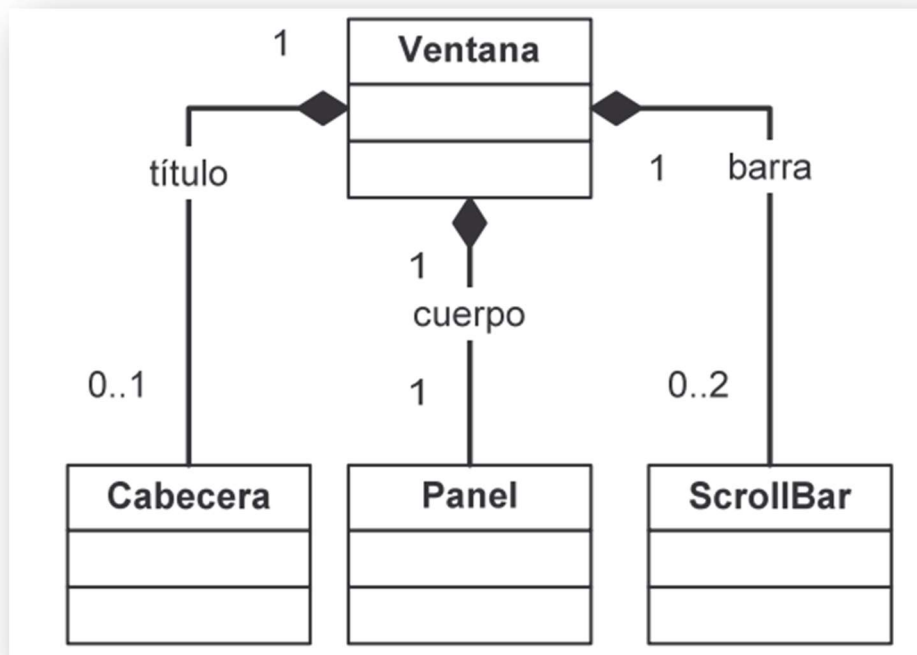
- **AGREGACIÓN**



La asociación de agregación entre una clase *EquipoFutbol* y *JugadorFutbol*:

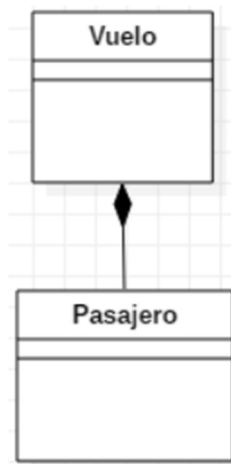
“Un Equipo está compuesto por jugadores, sin embargo, el jugador puede jugar también en otros equipos. **Si desaparece el equipo el jugador NO desaparece**”

- COMPOSICIÓN



Si se elimina la ventana, también se eliminan las clases *Cabecera*, *Panel* y *ScrollBar* de dicha ventana.

Otro ejemplo:

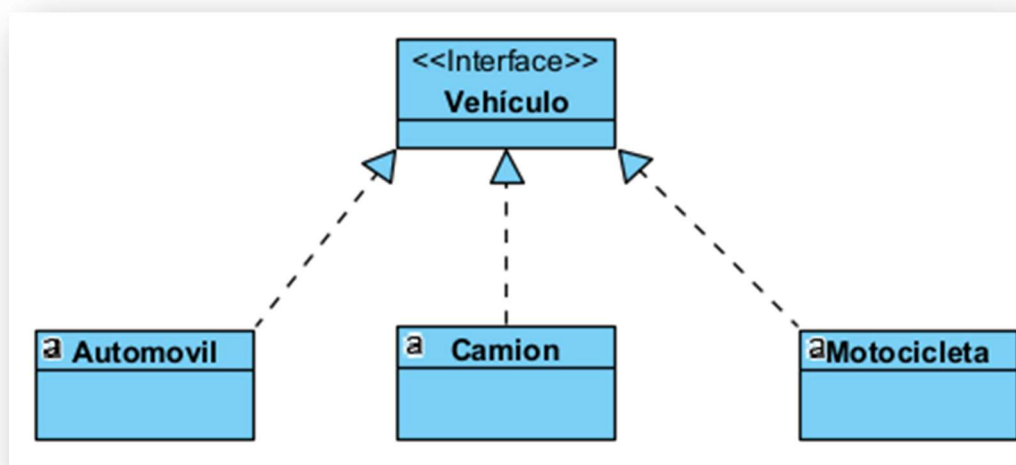


REALIZACIÓN

Una relación de realización es la relación de herencia existente entre una clase **interfaz** o **interface** y la subclase que implementa esa interfaz.

Esta relación de herencia se representa gráficamente mediante una flecha con línea discontinua en lugar de una línea completa.

Ejemplo:



El código en JAVA es el siguiente:

```
public interface Vehiculo {  
    void acelerar(int cuanto);  
    void frenar(int cuanto);  
}
```

```
public class Automovil implements Vehiculo {  
    private int velocidadActual;  
  
    @Override  
    public void acelerar(int cuanto) {  
        velocidadActual += cuanto;  
        System.out.println("Automóvil acelerando...  
Velocidad actual: " + velocidadActual + " km/h");  
    }  
  
    @Override  
    public void frenar(int cuanto) {  
        velocidadActual -= cuanto;  
        System.out.println("Automóvil frenando...  
Velocidad actual: " + velocidadActual + " km/h");  
    }  
}
```

```
public class Camion implements Vehiculo {  
    private int velocidadActual;  
  
    @Override  
    public void acelerar(int cuanto) {  
        velocidadActual += cuanto;  
        System.out.println("Camión acelerando...  
Velocidad actual: " + velocidadActual + " km/h");  
    }  
  
    @Override  
    public void frenar(int cuanto) {  
        velocidadActual -= cuanto;  
        System.out.println("Camión frenando... Velocidad  
actual: " + velocidadActual + " km/h");  
    }  
}
```

```
public class Motocicleta implements Vehiculo {
    private int velocidadActual;

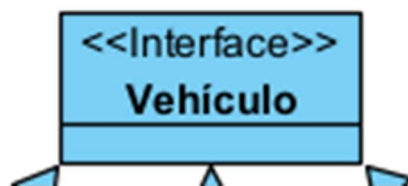
    @Override
    public void acelerar(int cuanto) {
        velocidadActual += cuanto;
        System.out.println("Motocicleta acelerando...
Velocidad actual: " + velocidadActual + " km/h");
    }

    @Override
    public void frenar(int cuanto) {
        velocidadActual -= cuanto;
        System.out.println("Motocicleta frenando...
Velocidad actual: " + velocidadActual + " km/h");
    }
}
```



Importante:

Una **interfaz** o **interface** es una clase totalmente abstracta, es decir, no tiene atributos y todos sus métodos son abstractos y públicos, sin desarrollar. Esas clases NO implementan ningún método. Gráficamente se representan como una clase con el estereotipo **<<interface>>**



Cómo crear una Clase (de tipo interface) en Visual Pradigm:

<https://forums.visual-paradigm.com/t/how-to-define-an-interface/14714>

DEPENDENCIA

En UML, una relación de dependencia entre dos clases **indica que una clase utiliza a otra de manera que un cambio en la clase utilizada** podría afectar a la clase que la utiliza. Esta relación se representa con una línea punteada que tiene una flecha en el extremo que apunta hacia la clase de la que se depende.

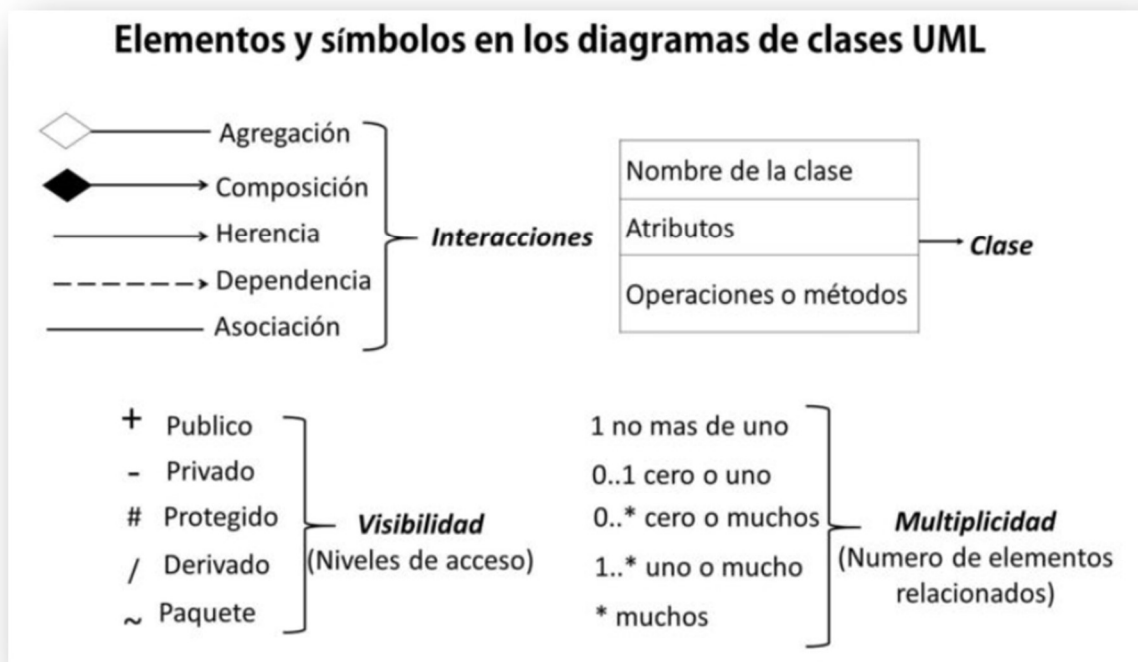
Ejemplo:

Imaginemos que tenemos una clase **Reporte** que genera reportes en diferentes formatos. También tenemos una clase **GeneradorPDF** que sabe cómo crear un PDF. La clase **Reporte** necesita a **GeneradorPDF** para generar reportes en formato PDF, pero **Reporte** no contiene a **GeneradorPDF** como un atributo; más bien, utiliza **GeneradorPDF** temporalmente para realizar su tarea.



```
public class Reporte {
    public void generarReportePDF() {
        GeneradorPDF generador = new GeneradorPDF();
        generador.crearPDF();
    }
}
```

```
public class GeneradorPDF {
    public void crearPDF() {
        // Lógica para crear un PDF
    }
}
```



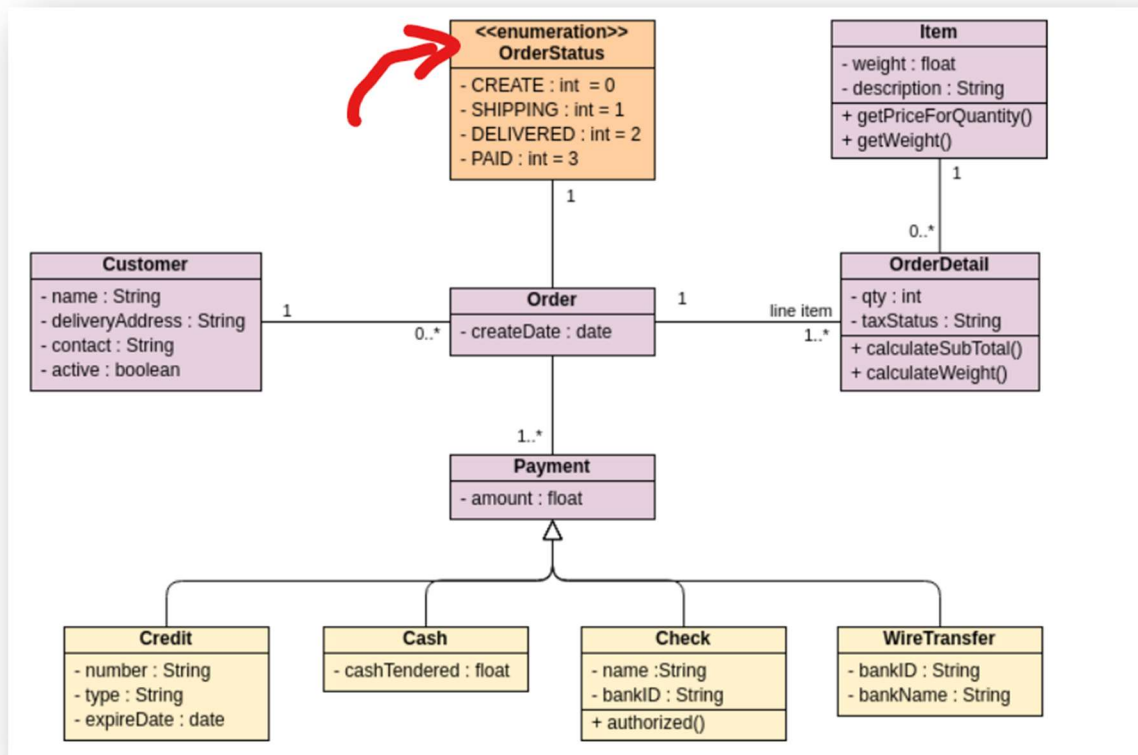
3.7 – Estereotipos

Los estereotipos en UML son una forma de extender el vocabulario del lenguaje de modelado unificado (UML) para crear nuevos elementos de modelado basados en los existentes, añadiendo más significado a los diagramas.

Se representan gráficamente con el nombre del estereotipo encerrado entre dos signos de guión angulares (<< >>).

Permiten clasificar y marcar los elementos de modelado para indicar roles especiales, restricciones o propiedades adicionales que estos puedan tener. Por ejemplo, en el caso de las clases, un estereotipo podría ser <<entity>> para indicar que la clase es parte de una capa de persistencia en una aplicación, o <<control>> para señalar que la clase maneja la lógica de negocio. Los estereotipos ofrecen una manera flexible de adaptar UML a diferentes dominios y metodologías de desarrollo.

Ejemplo:



5.- Referencias bibliográficas

- ❖ Moreno Pérez, J.C. *Entornos de desarrollo*. Editorial Síntesis.
- ❖ Ramos Martín, A. & Ramos Martín, M.J. *Entornos de desarrollo*. Grupo editorial Garceta.