TEMA 5

Elaboración de diagramas de clases

1. INTRODUCCIÓN

- En el diseño orientado a objetos, un sistema se entiende como un conjunto de objetos que tienen propiedades y comportamientos.
- Un **objeto** consta de una estructura de datos y de una colección de métodos u operaciones que manipulan esos datos.
- Una **clase** es una plantilla para la creación de objetos.
- Cuando se crea un objeto, se ha de especificar de qué clase es el objeto instanciando, para que el compilador entienda las características del objeto.
- Para el análisis y diseño orientado a objetos se utiliza **UML** (Lenguaje unificado de modelado).

2. UML

- Es un lenguaje de modelado basado en diagramas que sirve para expresar modelos (representación de la realidad).
- El lenguaje de modelado unificado es un lenguaje gráfico para **visualizar**, **especificar** y **documentar** cada una de las partes que comprende el desarrollo del software.
- Este lenguaje se puede utilizar para modelar tanto sistemas de software como de hardware, como organizaciones del mundo real.
- Para ello, utiliza una serie de **diagramas** en los que se representan los distintos puntos de vista de modelado.

2. UML

- Se distinguen dos grandes versiones:
  - UML 1.X (finales de los 90)
  - UML 2.X (año 2005)

- UML 2.0 define 13 tipos de diagramas divididos en 3 categorías.

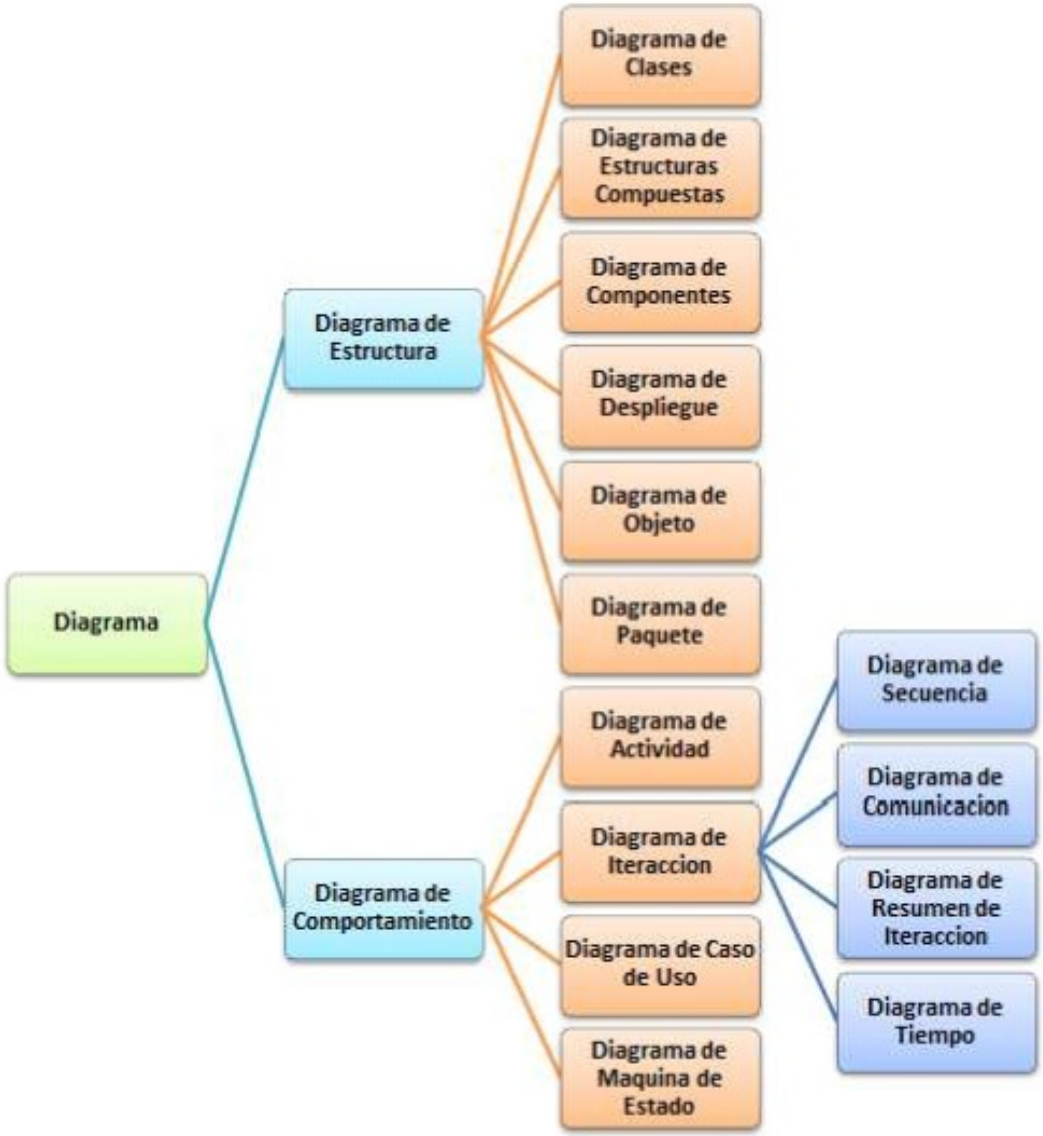
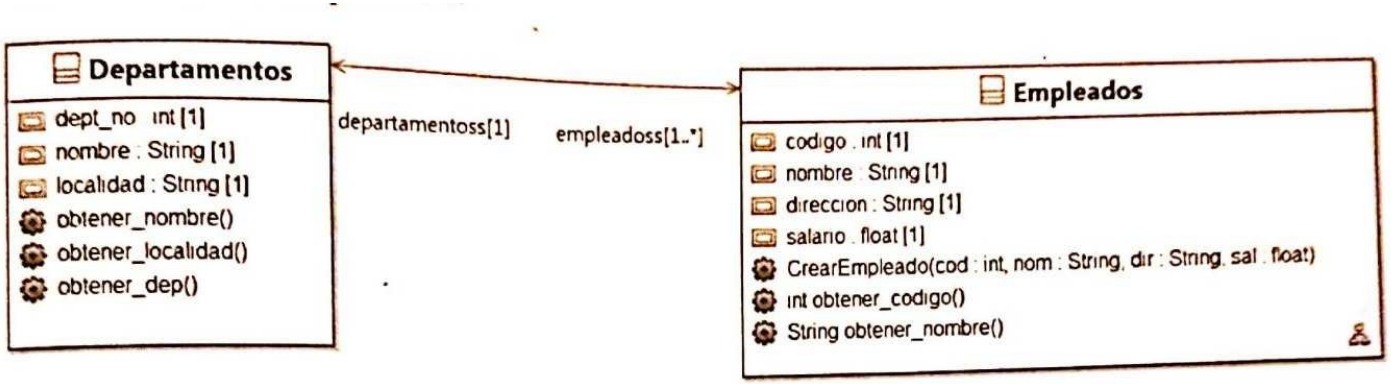


Diagrama
Diagrama de Estructura
Diagrama de Clases
Diagrama de Estructuras Compuestas
Diagrama de Componentes
Diagrama de Despliegue
Diagrama de Objeto
Diagrama de Paquete
Diagrama de Comportamiento
Diagrama de Actividad
Diagrama de Iteraccion
Diagrama de Caso de Uso
Diagrama de Maquina de Estado
Diagrama de Secuencia
Diagrama de Comunicacion
Diagrama de Resumen de Iteraccion
Diagrama de Tiempo

3. TIPOS DE DIAGRAMAS

- Cada diagrama UML representa alguna parte o punto de vista del sistema.
- Los diagramas más utilizados son los siguientes:
  - **Diagramas de clase:** muestran las diferentes clases que componen un sistema y cómo se relacionan unas con otras. En el ejemplo: se muestran las clases Departamentos y Empleados y su relación. Un departamento tiene muchos empleados y un empleado pertenece a un departamento.



3. TIPOS DE DIAGRAMAS

- **Diagramas de objeto:** representan objetos y sus relaciones. Muestran una serie de objetos y sus relaciones en un momento particular de la ejecución del sistema.
- **Diagramas de casos de uso:** se utiliza para entender el uso del sistema, muestran un conjunto de actores, las acciones (casos de uso) que se realizan en el sistema, y las relaciones entre ellos. En el ejemplo, se muestra un diagrama de casos de uso en el que el actor operador realiza operaciones sobre los datos de los empleados de una base de datos.

3. TIPOS DE DIAGRAMAS

- **Diagramas de secuencia:** son una representación temporal de los objetos y sus relaciones. Enfatiza la interacción entre los objetos y los mensajes que intercambian entre sí junto con el orden temporal de los mismos.

  En el ejemplo, se muestra un diagrama de secuencia de la inserción de datos de un empleado en la base de datos. El operador indica en la ventana principal que quiere realizar operaciones con empleados y pide abrir la ventana de empleados. Aquí se teclearán los datos, se validarán y el operador indicará si se graba el registro en la BD. De ser así, el registro creará un nuevo objeto Empleado en la base de datos con los datos tecleados.

3. TIPOS DE DIAGRAMA

- **Diagramas de estado:** se utiliza para analizar los cambios de estado de los objetos. Muestra los estados, eventos, transiciones y actividades de los diferentes objetos. En el ejemplo, se muestran los estados de un objeto trabajador. Los **estados** son los rectángulos redondeados. Las **transiciones** son las flechas de un estado a otro. Las condiciones que desencadenan las transiciones están escritas al lado de las flechas. El estado inicial (círculo relleno) se representa para iniciar la acción. El círculo relleno dentro de otro círculo se utiliza para indicar el final de la actividad.

3. TIPOS DE DIAGRAMA

- **Diagramas de actividad:** se utiliza para mostrar la secuencia de actividades. Muestran el flujo de trabajo desde un punto de inicio hasta el punto final detallando las decisiones que surgen en la progresión de los eventos contenidos en la actividad. En el ejemplo, se muestra el diagrama de actividad de la inserción de un empleado en la BD.

3. TIPOS DE DIAGRAMA

- **Diagramas de despliegue:** especifica el hardware físico sobre el que el sistema de software se ejecutará y también se especifica cómo el software se despliega en ese hardware.

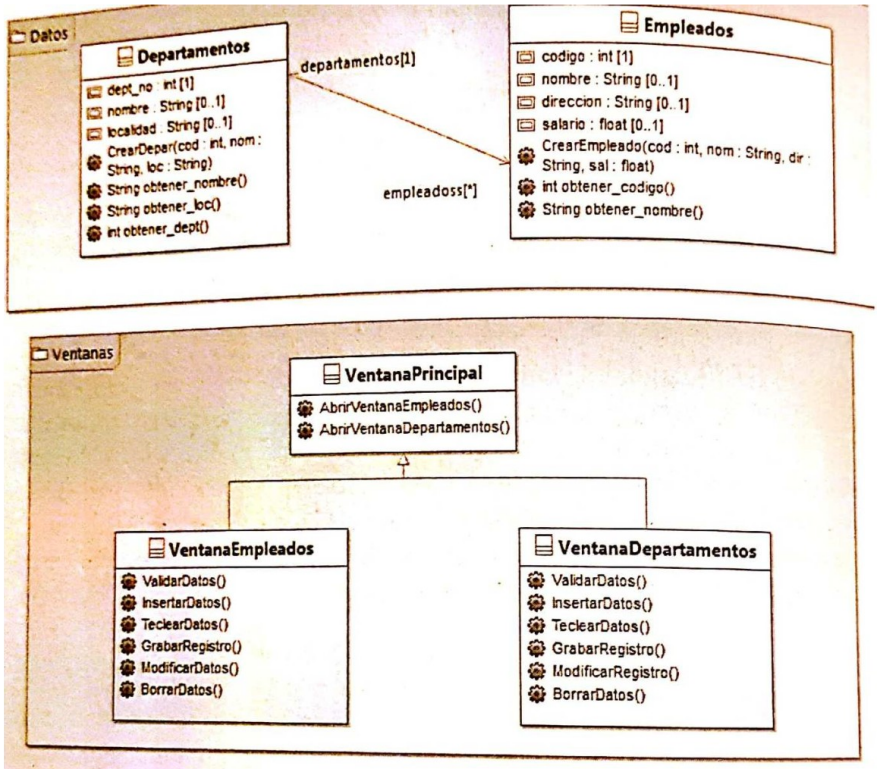
  En el ejemplo se muestra el diagrama de despliegue de gestión de empleados y departamentos. La arquitectura de este sistema está basada en un servidor y dos puestos clientes conectados al servidor mediante enlaces directos que representan la red. El servidor contiene tres ejecutables.

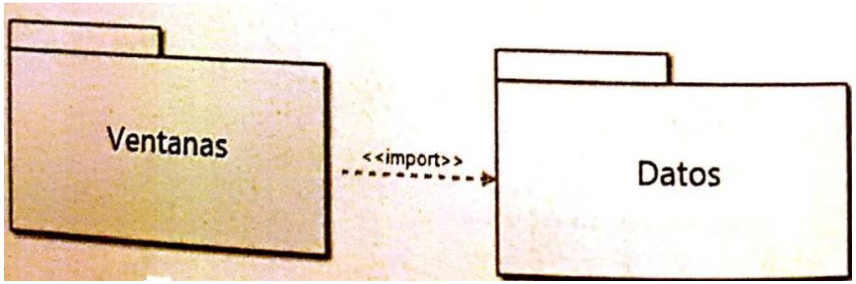
3. TIPOS DE DIAGRAMA

- **Diagramas de paquetes:** se usan para reflejar la organización de paquetes y sus elementos.

  En el ejemplo se muestra un diagrama de paquetes formado por 5 clases: las clases **Departamentos** y **Empleados** agrupadas en el paquete **Datos** y las clases **VentanaPrincipal**, **VentanaEmpleados** y **VentanaDepartamentos** agrupadas en el paquete **Ventanas**.

  Se muestra el diagrama de paquetes en el que se relacionan los dos paquetes mediante una asociación de ***importación***, las clases del paquete Ventanas importarán las clases del paquete Datos.


Datos
Departamentos
dept_no : int [1]
nombre : String [0..1]
localidad : String [0..1]
CrearDepar(cod : int, nom : String, loc : String)
String obtener_nombre()
String obtener_loc()
int obtener_dept()
departamentos[1]
empleadoss[*]
Empleados
codigo : int [1]
nombre : String [0..1]
direccion : String [0..1]
salario : float [0..1]
CrearEmpleado(cod : int, nom : String, dir : String, sal : float)
int obtener_codigo()
String obtener_nombre()
Ventanas
VentanaPrincipal
AbrirVentanaEmpleados()
AbrirVentanaDepartamentos()
VentanaEmpleados
ValidarDatos()
InsertarDatos()
TeclearDatos()
GrabarRegistro()
ModificarDatos()
BorrarDatos()
VentanaDepartamentos
ValidarDatos()
InsertarDatos()
TeclearDatos()
GrabarRegistro()
ModificarRegistro()
BorrarDatos()


Ventanas
<<import>>
Datos

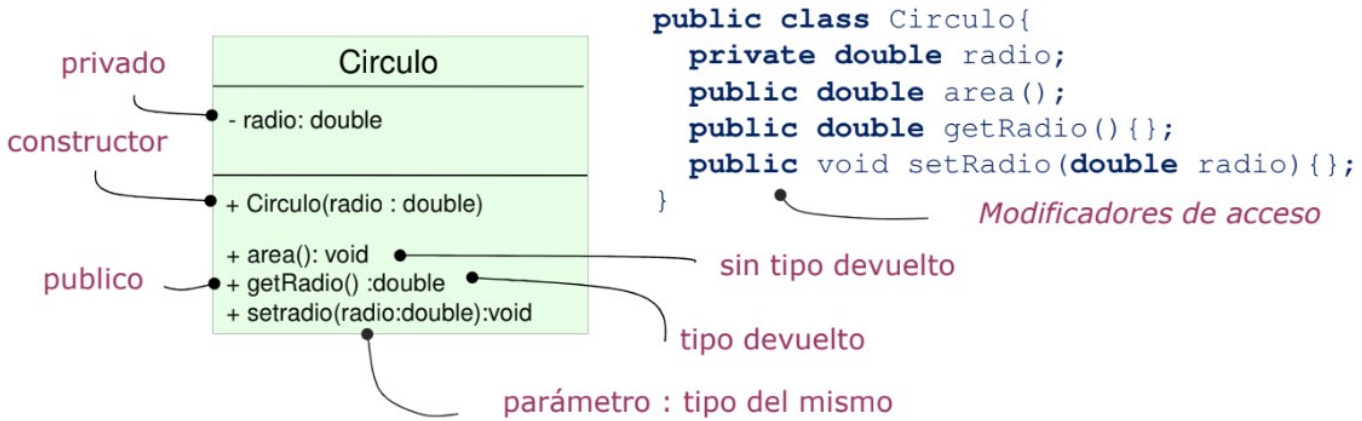
4. DIAGRAMA DE CLASES

- Un diagrama de clases es un tipo de diagrama de estructuras (estático) que describe la estructura de un sistema mostrando sus clases y las asociaciones entre ellas.
- Sirve para visualizar las relaciones entre las clases que componen el sistema.
- Un diagrama de clases está compuesto principalmente por los siguientes elementos:

. **Clases:** atributos, métodos y visibilidad
. **Relaciones:** asociación, herencia, agregación, composición, realización y dependencia.

4.1. CLASES

- Las clases son la unidad básica que encapsula toda la información de un objeto.
- En UML, una clase se representa por un rectángulo que posee tres divisiones:




```
public class Circulo{
  private double radio;
  public double area();
  public double getRadio(){};
  public void setRadio(double radio){};
}
```

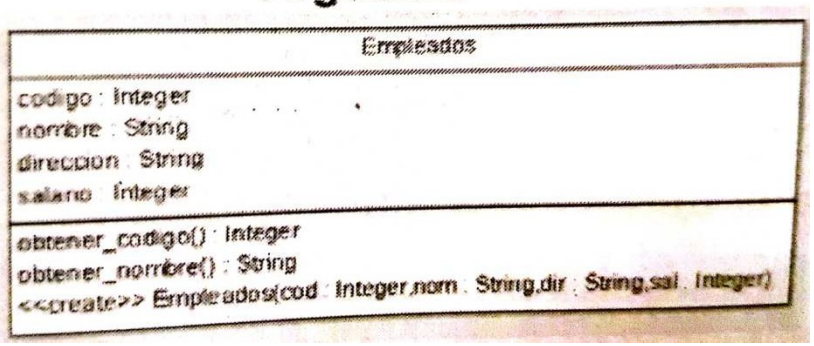
Modificadores de acceso

- La parte superior contiene el nombre de la clase
- La parte intermedia contiene los atributos y su visibilidad
- La parte inferior contiene los métodos y su visibilidad.

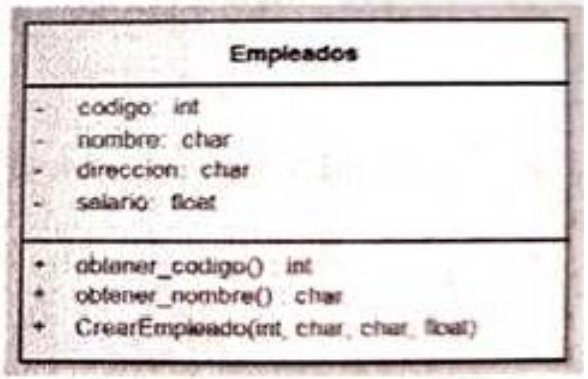
4.1. CLASES

- En la representación de una clase los atributos y métodos podrían omitirse.
- A continuación se muestra la misma clase representada con diferentes herramientas:

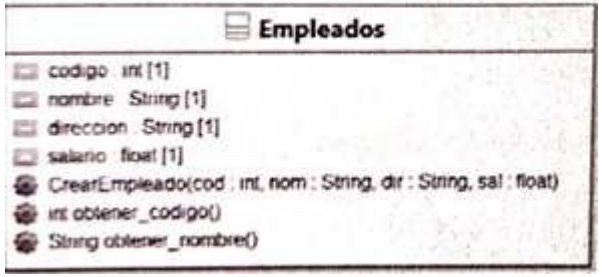
ArgoUML



Enterprise Architect



Eclipse UML



4.2. ATRIBUTOS

- Un atributo representa una propiedad de la clase que se encuentra en todas las instancias de la clase.
- Los atributos pueden representarse mostrando solamente su nombre o mostrando su nombre y su tipo e incluso su valor por defecto.
- La sintaxis de definición de atributos sigue el siguiente patrón:

visibilidad nombre:tipo = valor-inicial

4.2.1 Visibilidad

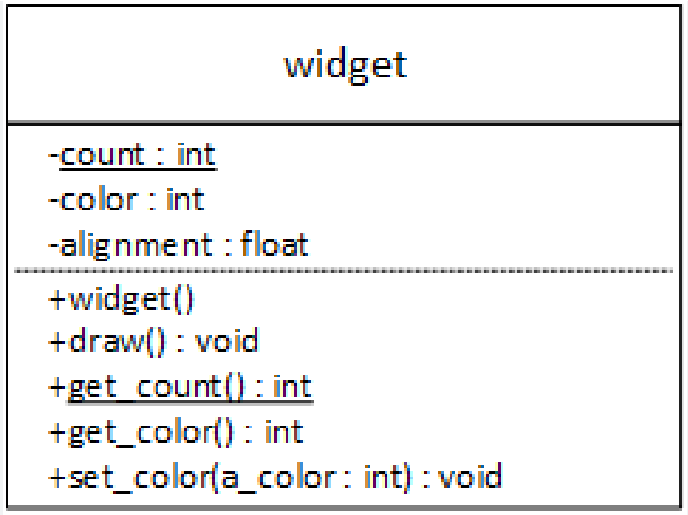
- Al crear el atributo se indicará su visibilidad con el entorno:
  - **public:** atributo visible dentro y fuera de la clase. **Signo +.**
  - **private:** sólo accesible desde dentro de la clase. **Signo -.**
  - **protected:** el atributo no será accesible desde fuera de la clase pero sí podrá ser accedido por métodos de la clase y de las subclases. **Carácter #.**
  - **package:** el atributo es visible en las clases del mismo paquete. **Carácter ~.**

UML	Java	Descripción
-	Private	Solo se puede acceder a los atributos y métodos de la clase desde la propia clase.
#	Protected	Solo se puede acceder a los atributos y métodos de la clase desde la propia clase o desde una clase que herede de ella.
+	Public	Se puede acceder a los atributos y métodos de la clase desde cualquier lugar.
~		Se puede acceder a los a los atributos y métodos de una clase desde cualquier clase del mismo paquete

4.2.2 Alcance

- Atributo **Static**: El valor del atributo es el mismo para todos los objetos (instancias de la clase.
- Método **Static**: El método se puede invocar sin tener que instanciar la clase (Ejemplo: Clase Math).

Se representa subrayando la característica estática.

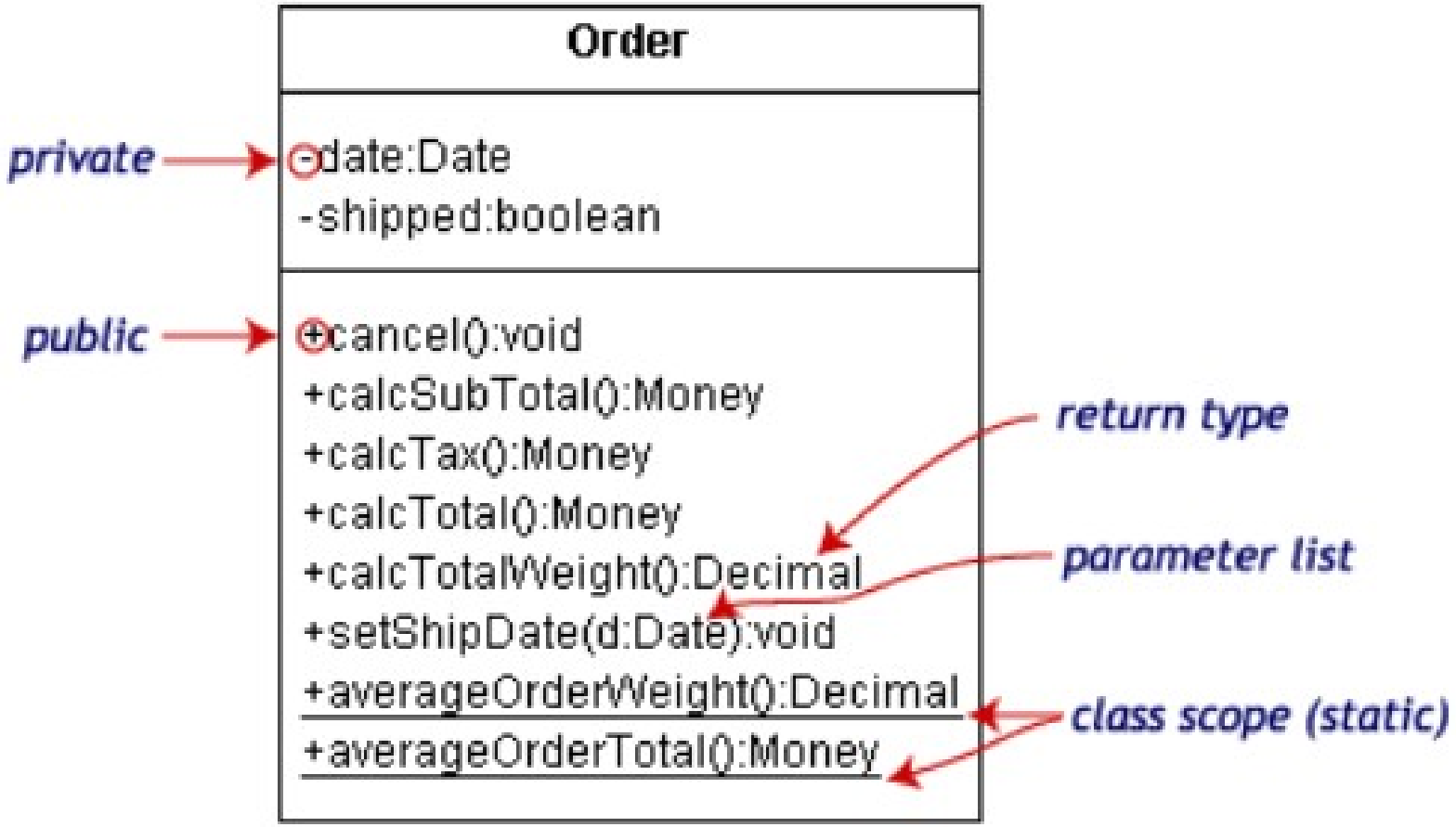


4.3 MÉTODOS

- Un método es la implementación de un servicio de la clase que muestra un comportamiento común a todos los objetos.
- Definen la forma de cómo la clase interactúa con su entorno.
- Igual que los atributos, los métodos pueden ser:
  Public: Signo +
  private: Signo -
  protected: Carácter #
  package: Carácter ~.
- La sintaxis de definición de métodos sigue el siguiente patrón:

  visibilidad nombre_metodo(lista_params):tipo_retorno

  lista params = (nombre parametro: tipo parametro, ...)


Order
-date:Date
-shipped:boolean
+cancel():void
+calcSubTotal():Money
+calcTax():Money
+calcTotal():Money
+calcTotalWeight():Decimal
+setShipDate(d:Date):void
+averageOrderWeight():Decimal
+averageOrderTotal():Money
private
public
return type
parameter list
class scope (static)

EJERCICIO

Representa la clase Artículo formada por los siguientes atributos y métodos:

Atributos

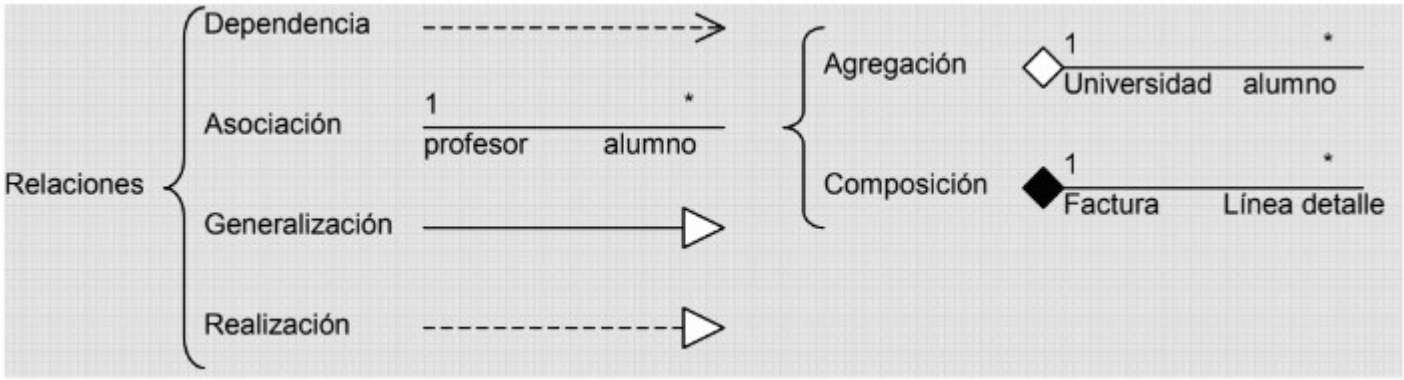
- código: entero. Visibilidad package
- denominación: char. Visibilidad public
- precioVenta: entero. Visibilidad protected. Alcance static.
- stockAlmacen: entero. Visibilidad private
- stockMinimo: entero. Visibilidad private

Métodos

- calcularStock(): retorna int. Visibilidad private
- calcularPVP(): retorna int. Visibilidad protected. Alcance static.
- obtenerCodigo(): retorna String. Visibilidad package. Parametros id tipo int, valor tipo double.

4.4 RELACIONES

- Se distinguen los siguientes tipos de relaciones:
  - Asociación
  - Clase asociación
  - Herencia
  - Composición
  - Agregación
  - Realización
  - Dependencia



4.4.1 ASOCIACIÓN

- En el mundo real muchos objetos están vinculados o relacionados entre sí.
- Los vínculos se corresponden con las **asociaciones** entre los objetos, por ejemplo, el vínculo existente entre un alumno y el curso en el que está matriculado, o el vínculo entre un profesor y el centro en el que trabaja.
- En UML, estos vínculos se describen mediante **asociaciones**, de igual modo que los objetos se describen mediante clases.
- Las asociaciones tienen un nombre y poseen una cardinalidad llamada **multiplicidad** que representa el número de instancias de una clase que se relacionan con las instancias de otra clase.

MULTIPLICIDAD

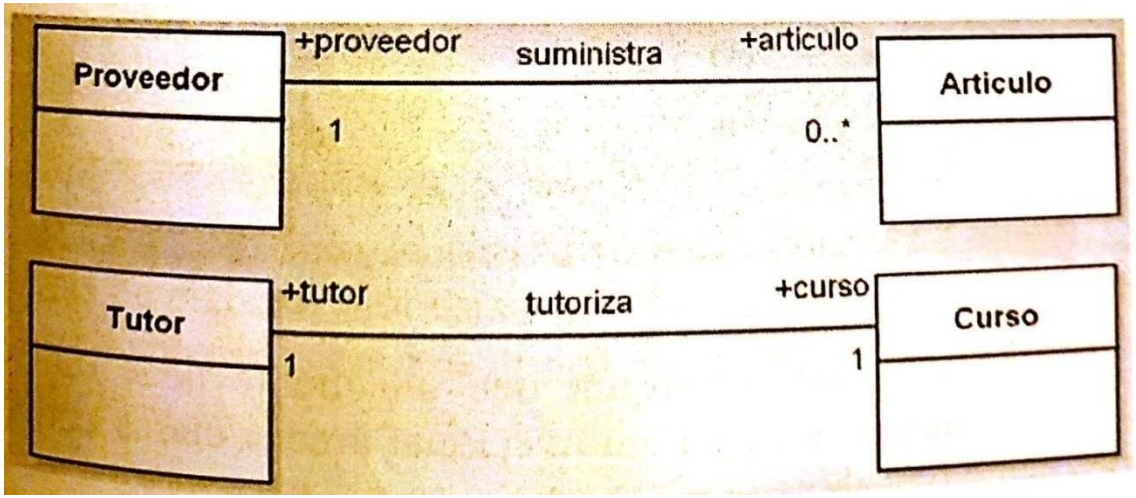
- La multiplicidad situada en un extremo de una asociación indica a cuántas instancias de la clase situada en ese mismo extremo está vinculada una instancia de la clase situada en el extremo opuesto.
- En los extremos de la asociación se especifica la multiplicidad mínima y máxima con el fin de indicar el intervalo de valores al que deberá pertenecer siempre la multiplicidad.
- Para expresar las multiplicidades se utiliza la siguiente notación:

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

EJEMPLOS:

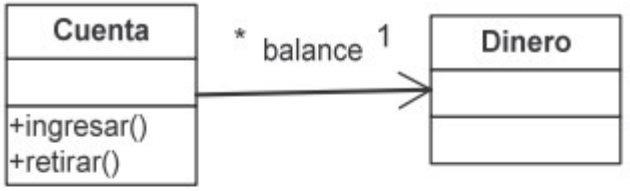
- Ejemplos:
  - Se muestra la asociación suministra entre Proveedor y Artículo. Un proveedor suministra 0 o muchos artículos. Un artículo es suministrado por un proveedor.

  - Se muestra la asociación tutoriza entre Tutor y Curso, un tutor tutoriza a un curso y un curso es tutorizado por un tutor.

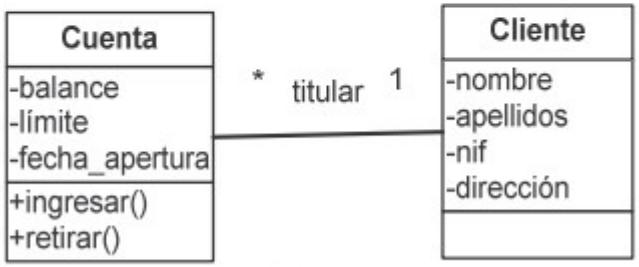


NAVEGAVILIDAD

- Puede ser **bidireccional** o **unidireccional** dependiendo de si ambas conocen la existencia la una de la otra o no.
- Dentro de una relación de asociación, cada clase juega un papel que se indica en la parte superior o inferior de la línea que conecta a dichas clases.



Asociación unidireccional



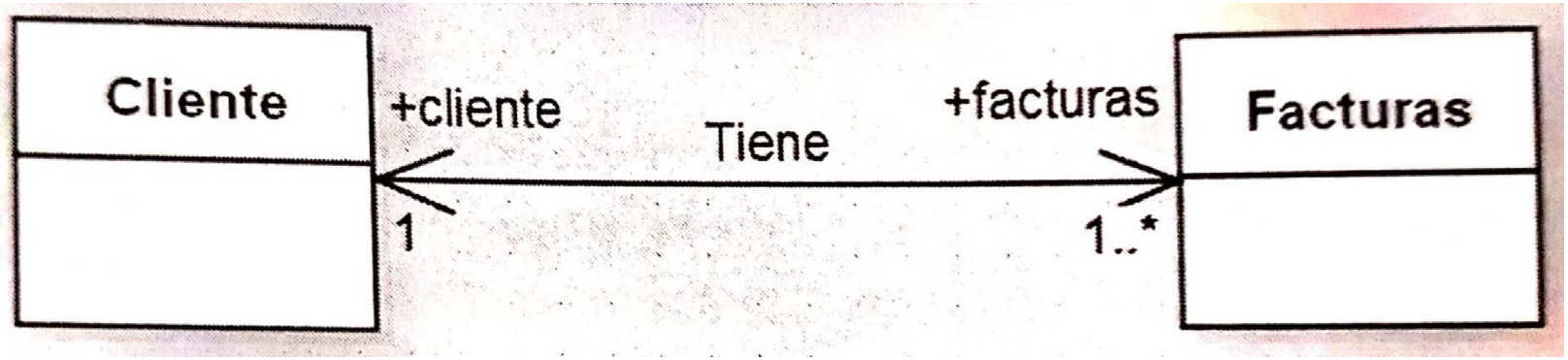
Asociación bidireccional

NAVEGAVILIDAD

- Si se convierten a JAVA dos clases unidas por una asociación bidireccional, cada una de las clases tendrá un objeto o set de objetos de la otra clase, dependiendo de la multiplicidad entre ellas.
- En cambio, si la asociación es unidireccional, la clase destino no sabrá de la existencia de la clase origen, y la clase origen contendrá un objeto o set de objetos de la clase destino.
- La **navegabilidad** entre clases nos muestra que es posible pasar desde un objeto de la clase origen a uno o más objetos de la clase destino dependiendo de la multiplicidad. En el caso de la asociación unidireccional, la navegabilidad va en un solo sentido, del origen al destino. El origen es navegable al destino, sin embargo, el destino no es navegable al origen.

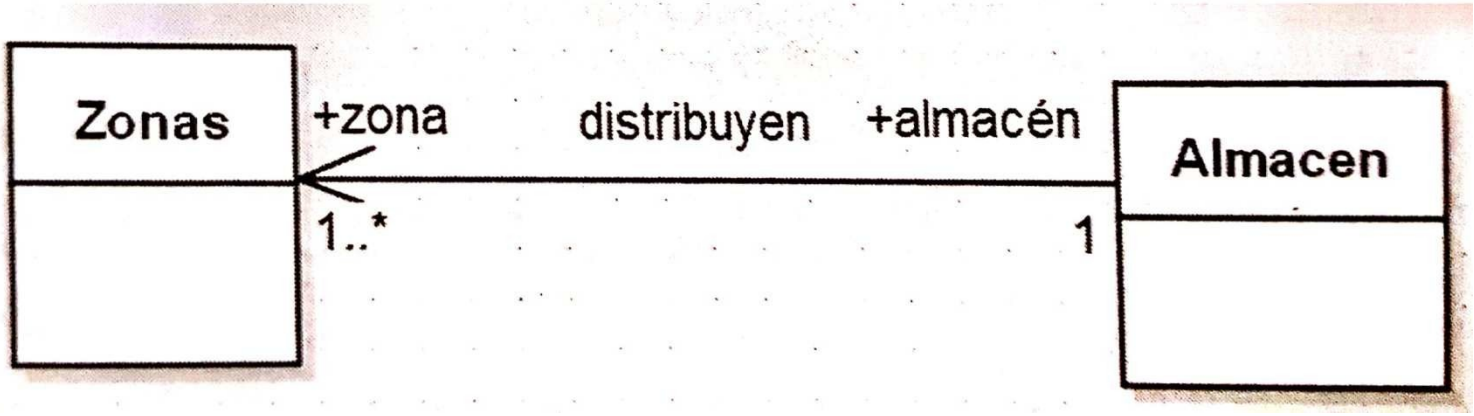
EJEMPLO

- La asociación ***Tiene*** muestra que un cliente tiene muchas facturas, y la factura es de un cliente. Como es bidireccional, ambas clases conocen su existencia. Ambas clases son navegables.



EJEMPLO

- En la asociación ***distribuyen***, un almacén distribuye artículos en varias zonas. La asociación es unidireccional, sólo la clase origen Almacén conoce la existencia de la clase destino Zonas. Almacén a zonas es navegable, en cambio, Zonas a Almacén no es navegable.



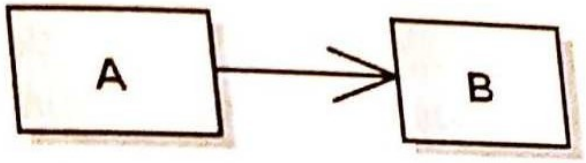
CÓDIGO

- El código JAVA generado con Eclipse UML correspondiente a estas asociaciones se muestra a continuación. Observa que se utiliza la clase HashSet para guardar colecciones de objetos en el caso de las multiplicidades 1..* ó 0..*. Es mas frecuente el uso de **List** para multiplicidades tipo *.

```
public class Cliente { // 1 cliente tiene muchas facturas (utiliza HashSet)
	public HashSet<Facturas> facturas = new HashSet<Facturas>();
	public Cliente() { // constructor
	}
	public HashSet<Facturas> getFacturas() {
		return this.facturas;
	}
	public void setFacturas(HashSet<Facturas> newFacturas) {
		this.facturas = newFacturas;
	}
}
```

```
public class Facturas {// 1 factura es de un cliente
      public Cliente cliente = null;
      public Facturas() { // constructor
      }
      public Cliente getCliente() {
             return this.cliente;
      }
      public void setCliente(Cliente newCliente) {
             this.cliente = newCliente;
      }
}

public class Zonas { // no sabe de la existencia de Almacen
      public Zonas() { // constructor
      }
}


public class Almacen { // 1 almacén distribuye en muchas zonas (HashSet)
      public HashSet<Zonas> zonas = new HashSet<Zonas>();
      public Almacen() { // constructor
      }
      public HashSet<Zonas> getZonass() {
             return this.zonas;
      }
      public void setZonass(HashSet<Zonas> newZonass) {
             this.zonas = newZonass;
      }
}
```

- **NOTA:** A la hora de elaborar diagramas de clases no todas las herramientas utilizan la misma notación para expresar la navegabilidad. En UML2 existen varias notaciones para expresar la navegabilidad, la práctica más estándar es utilizar la siguiente notación:

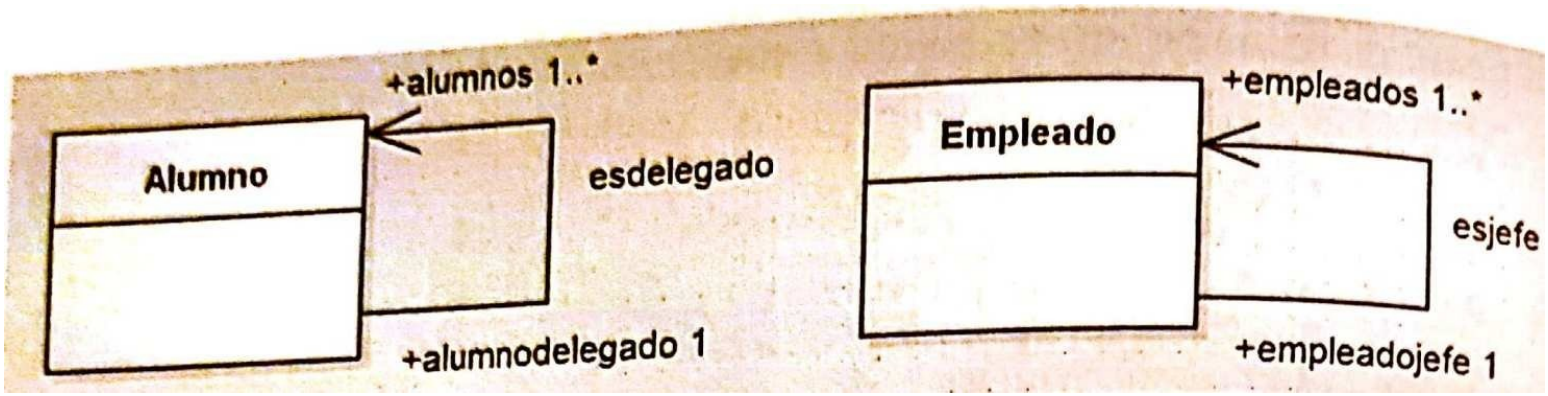


A a B es navegable
B a A no es navegable

A a B es navegable
B a A es navegable

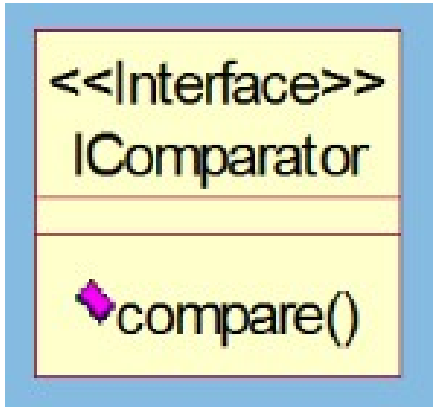
ASOCIACIONES REFLEXIVAS

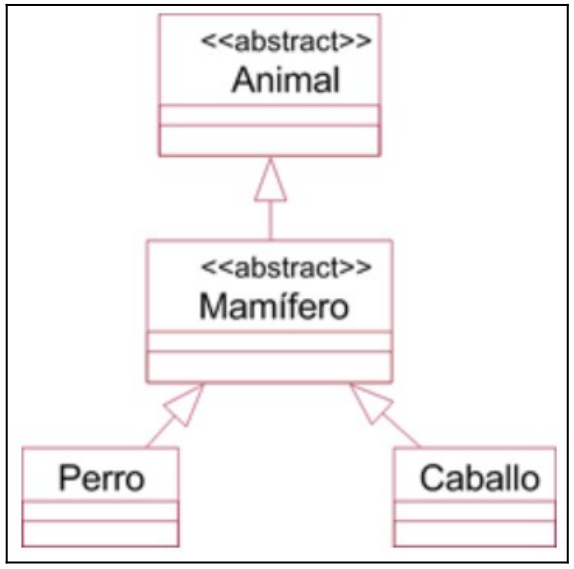
- Una clase puede asociarse consigo misma creando una asociación reflexiva, similar a las relaciones reflexivas del modelo Entidad/Relación. Estas asociaciones unen entre sí instancias de una misma clase.
- A continuación, se muestran dos asociaciones reflexivas, un alumno es delegado de muchos alumnos y un empleado es jefe de muchos empleados.



ESTEREOTIPOS

- Cuando un componente de un diagrama de clases tiene un significado especial, se le asigna una etiqueta que lo caracteriza con ese significado. Esta etiqueta recibe el nombre de estereotipo.
- Un estereotipo está formado por una palabra, que identifica ese significado especial, delimitada por comillas francesas, por ejemplo «interface».
- Se hará uso de los siguientes estereotipos:



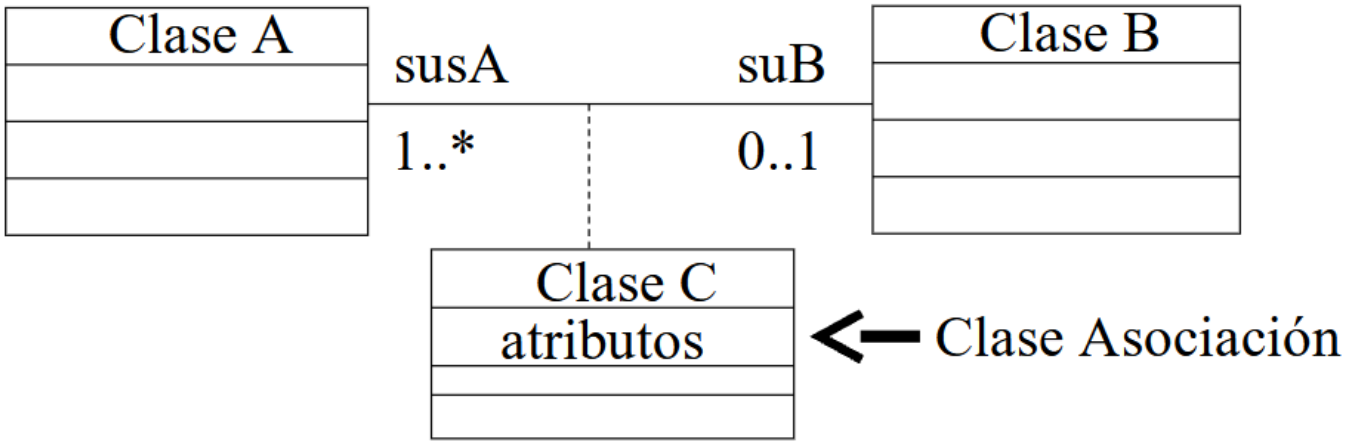


4.4.2 CLASE ASOCIACIÓN

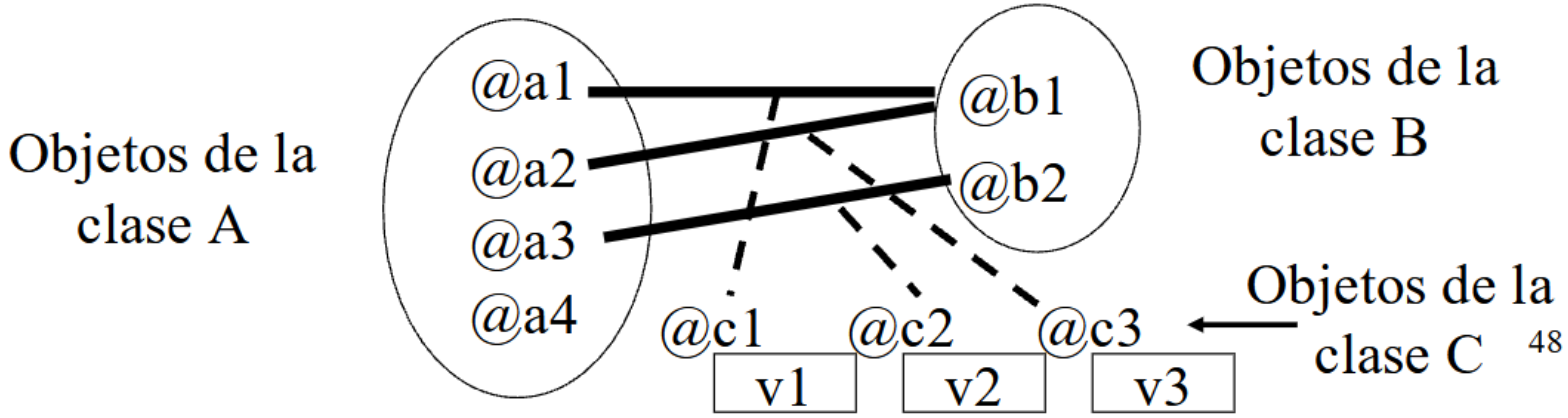
- Una asociación entre dos clases puede llevar información necesaria para esa asociación, a esto se le llama **clase asociación**, es como una relación M:N con atributos del modelo E/R.
- En este caso, la asociación recibe el estatus de clase y sus instancias son elementos de la asociación.
- Al igual que el resto, estas clases pueden estar dotadas de atributos y operaciones y estar vinculadas a otras clases a través de asociaciones.
- Una clase asociación añade una restricción:

“Sólo puede existir una instancia de la asociación entre cualquiera par de objetos participantes”

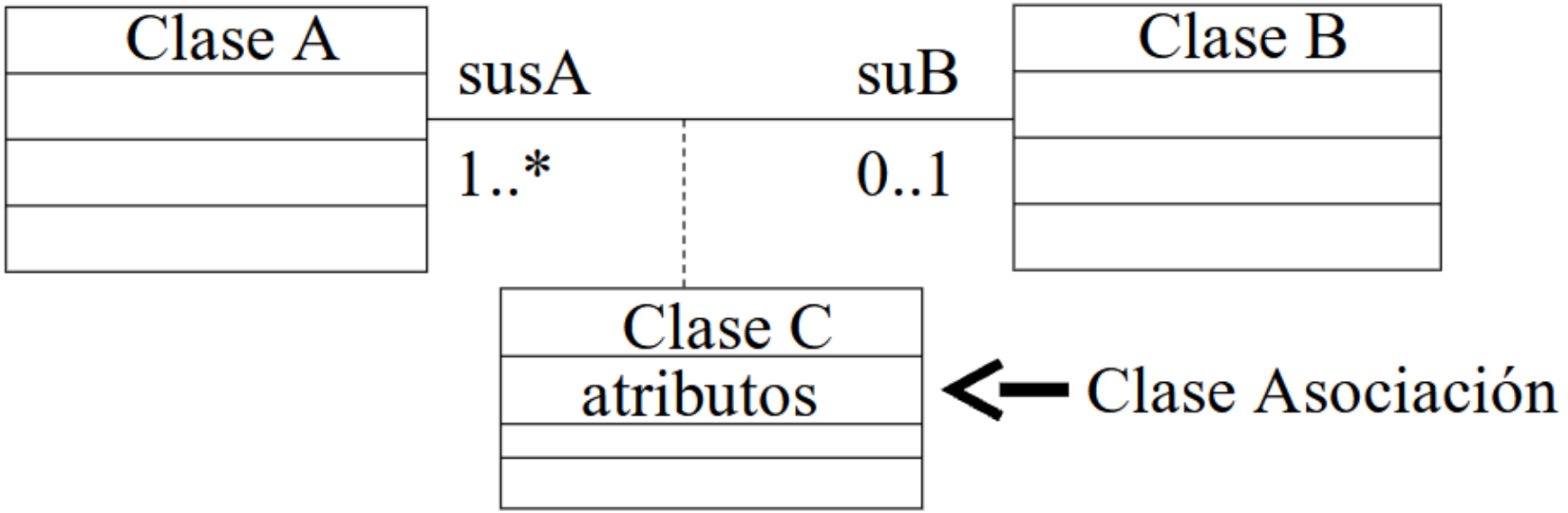
Clase Asociación



Para almacenar
<objeto de A, objeto de B, Atributos PROPIOS>

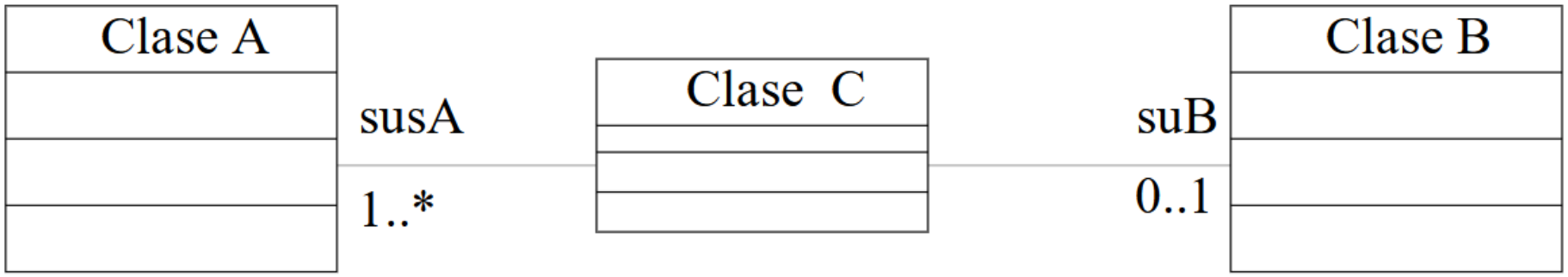


Clase Asociación



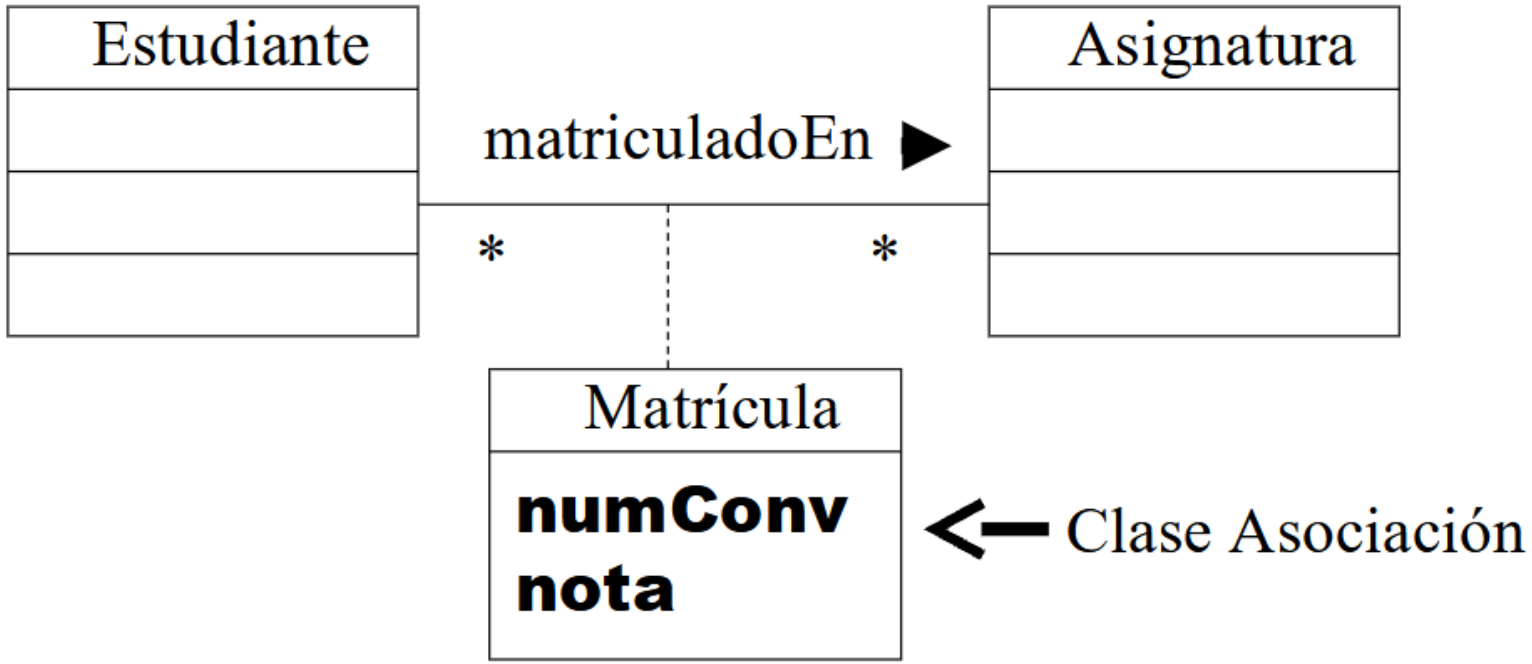
Cada objeto de C se refiere a un único objeto de A y a un único objeto de B.

Clase Asociación



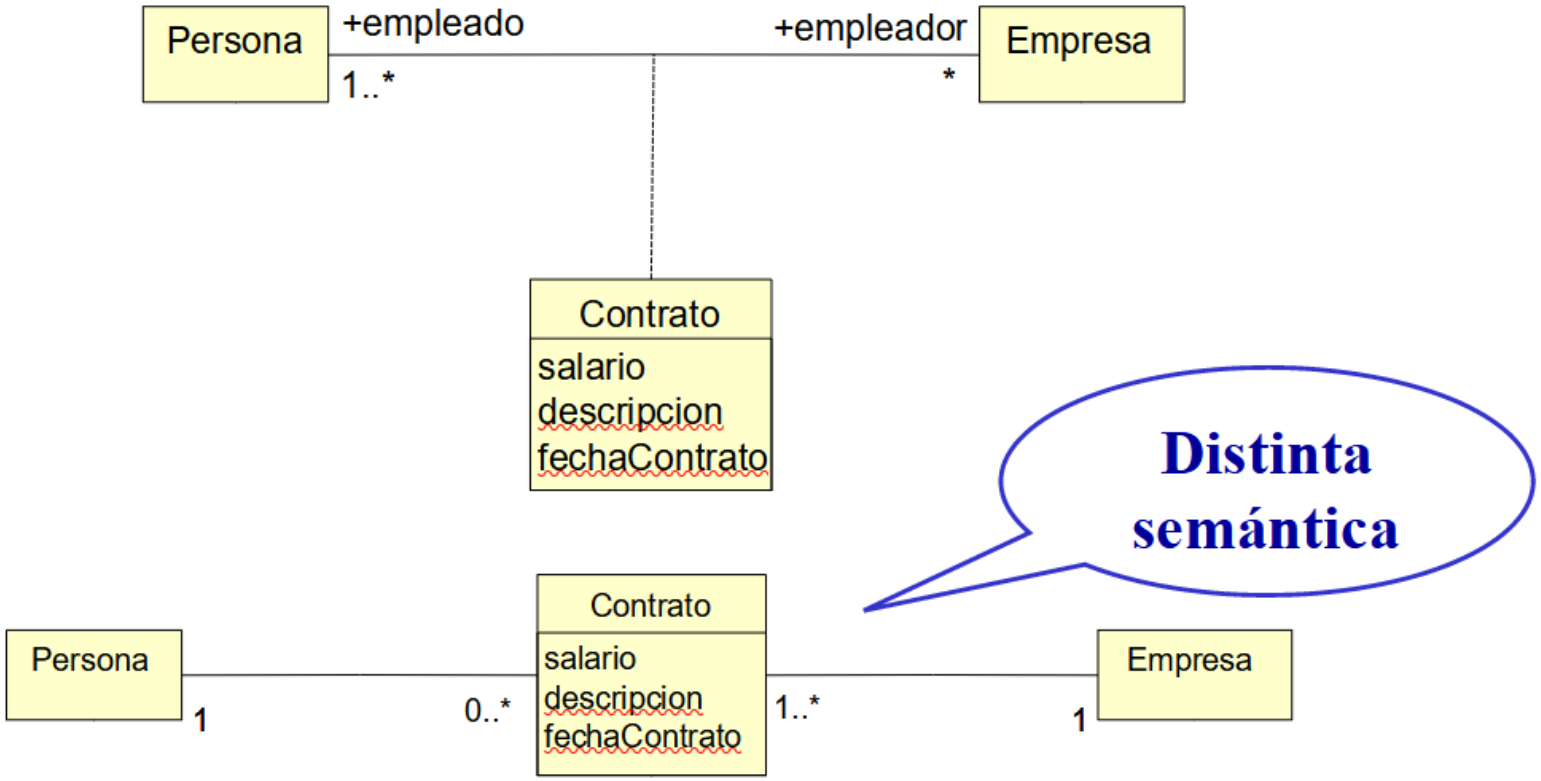
Si se quisiera que uno de C pudiera asociarse con varios de A y con 0 ó 1 de B, entonces no se puede usar una clase asociación sino una clase (normal) y 2 asociaciones.

Clase Asociación: Ejemplo

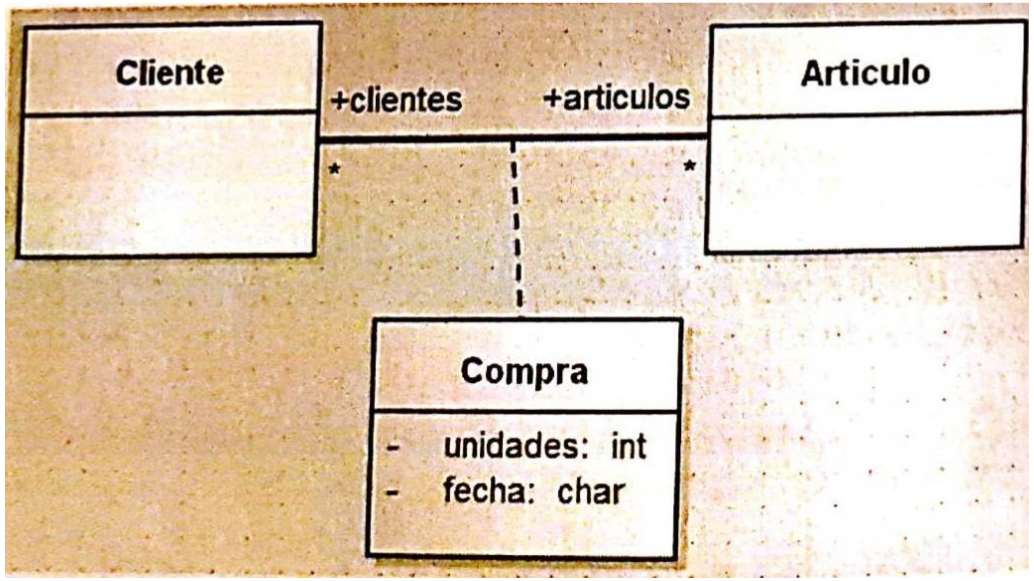


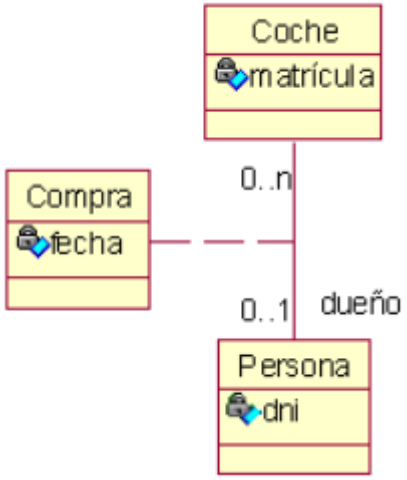
Si se desea poder almacenar el nº de convocatoria, nota obtenida, curso académico, etc.

Clase Asociación y Clase Normal: Ejemplo



Si se quiere modelar que una persona tiene diferentes contratos para una misma empresa a lo largo del tiempo, NO se debe diseñar mediante una clase asociación, sino mediante una clase normal.

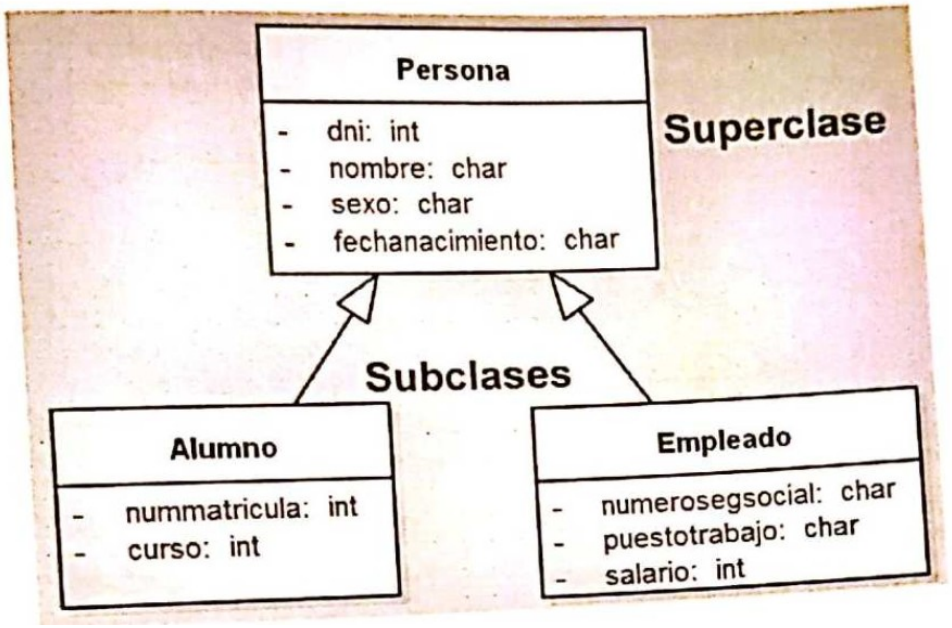

Cliente
+clientes
+articulos
Articulo
*
*
Compra
- unidades: int
- fecha: char

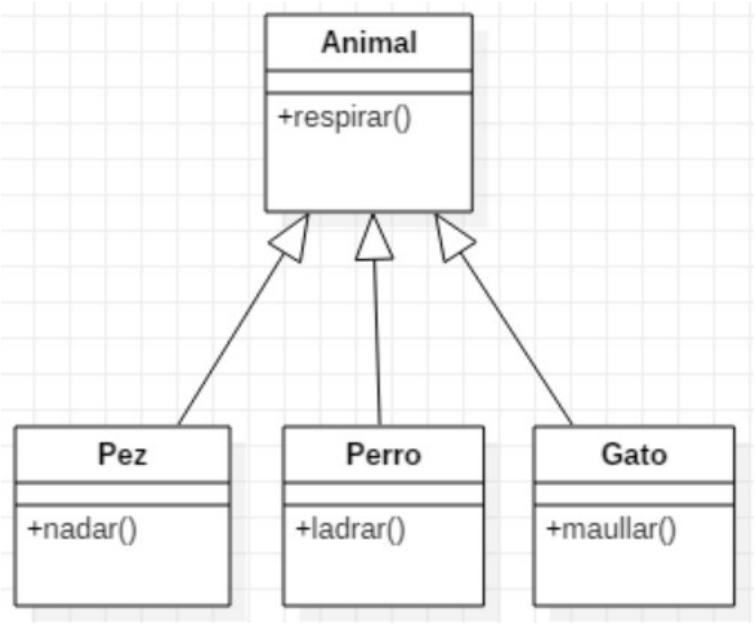

Coche
matrícula
0..n
Compra
fecha
0..1
dueño
Persona
dni

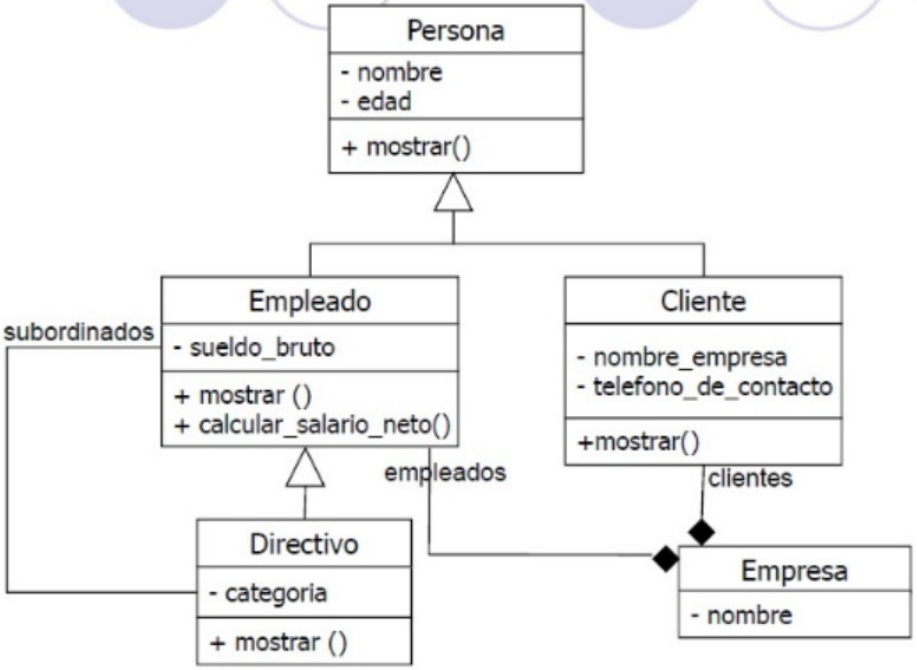

Compañía
Persona
Trabajo
fecha alta
salario

4.4.3 HERENCIA

- La herencia es una abstracción importante para compartir similitudes entre clases, donde todos los atributos y operaciones comunes a varias clases se pueden compartir por medio de la superclase.
- Para representar la herencia se utiliza una flecha con el extremo de la flecha apuntando a la superclase.
- A continuación, se muestra la herencia entre la clase Persona (superclase) y las clases Empleado y Alumno (subclases).
- Todas las clases comparten los atributos y métodos de Persona.


Persona
- dni: int
- nombre: char
- sexo: char
- fechanacimiento: char
Superclase
Subclases
Alumno
- nummatricula: int
- curso: int
Empleado
- numerosegsocial: char
- puestotrabajo: char
- salario: int

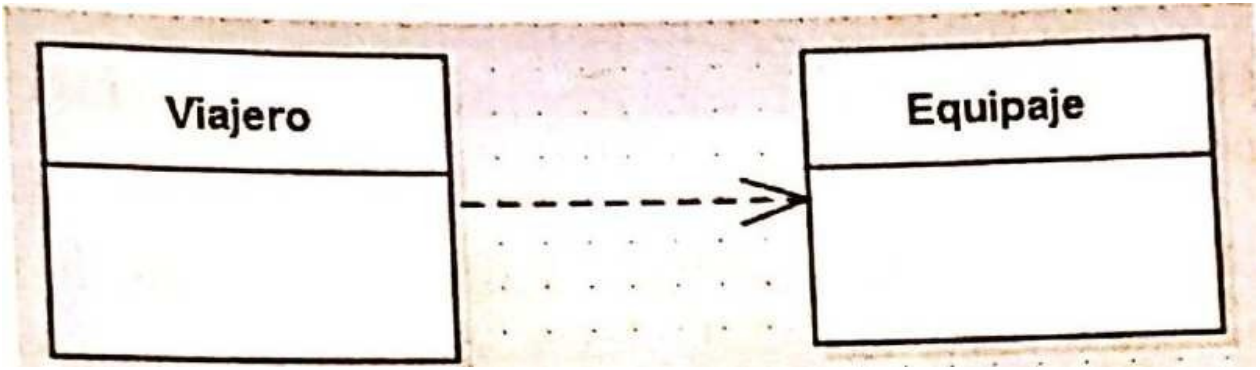

Animal
+respirar()
Pez
+nadar()
Perro
+ladrar()
Gato
+maullar()


Persona
- nombre
- edad
+ mostrar()
Empleado
- sueldo_bruto
+ mostrar ()
+ calcular_salario_neto()
Cliente
- nombre_empresa
- telefono_de_contacto
+mostrar()
subordinados
empleados
clientes
Directivo
- categoria
+ mostrar ()
Empresa
- nombre

4.4.4 DEPENDENCIA

- Es una relación que se establece entre dos clases cuando una clase usa a la otra, es decir, que la necesita para su cometido.
- Se representa con una flecha sin relleno discontinua que va desde la clase utilizadora a la clase utilizada (clase de la que depende).
- Con la dependencia mostramos que un cambio en la clase utilizada puede afectar al funcionamiento de la clase utilizadora, pero no al contrario.
- La relación de dependencia se da cuando una clase hace uso de otra de la siguiente manera:
  - En un método se pasa por parámetro un objeto de la clase independiente.
  - En un método se devuelve un objeto de la clase independiente.
  - Desde dentro de la clase dependiente se instancia un objeto de la clase independiente.

- Un ejemplo es la relación entre Viajero y Equipaje, el viajero necesita un equipaje para viajar.



- Otro ejemplo de esta relación lo podemos encontrar con una clase Impresora y una clase Documento. La impresora imprime documentos, entonces necesita al documento para imprimirlo.

```java
public class Impresora {

    private boolean estaEncendida;

    public void encender() {
        estaEncendida = true;
    }

    public void imprimir(Documento doc) {
        if (estaEncendida) {
            System.out.println("Imprimiendo doc " + doc.getTitulo());
            System.out.println("****************");
            System.out.println(doc.getCuerpo());
            System.out.println("****************");
        } else {
            System.out.println("Impresora apagada!");
        }
    }
}
```

```java
public class Documento {

    private String titulo;
    private String cuerpo;

    public Documento(String titulo, String cuerpo) {
        this.titulo = titulo;
        this.cuerpo = cuerpo;
    }

    public String getTitulo() {
        return titulo;
    }

    public String getCuerpo() {
        return cuerpo;
    }
}
```

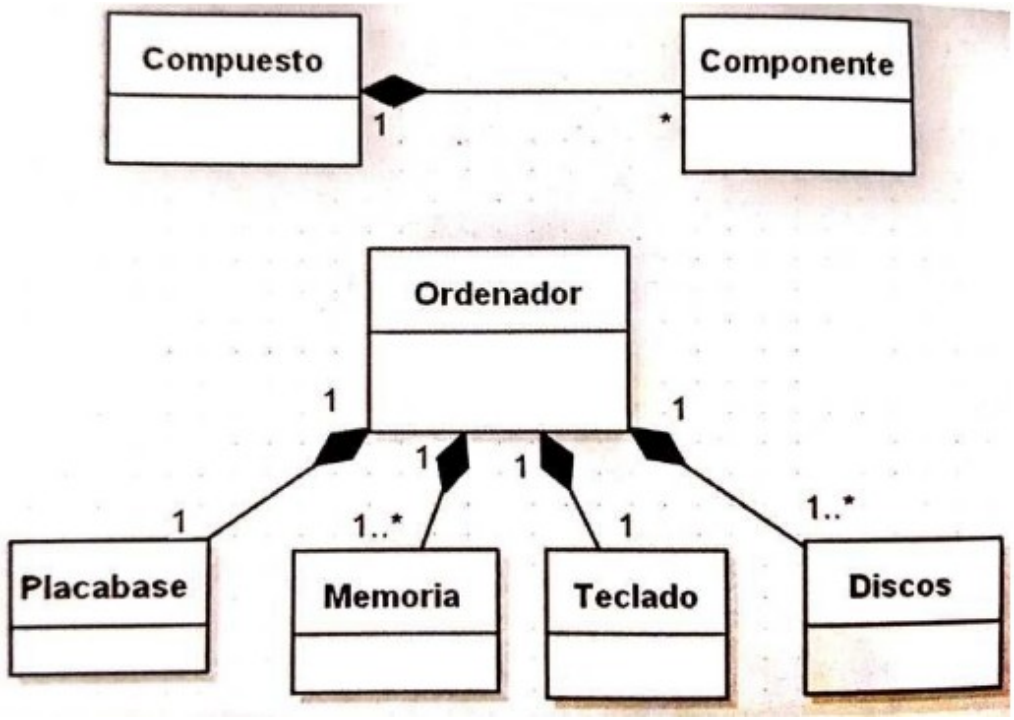
```java
public class Principal {

    public static void main(String[] args) {
        Documento doc1 = new Documento("Prueba", "Este es un\ntexto de prueba");
        Impresora imp = new Impresora();

        imp.encender();
        imp.imprimir(doc1);
    }

}
```

4.4.5 COMPOSICIÓN

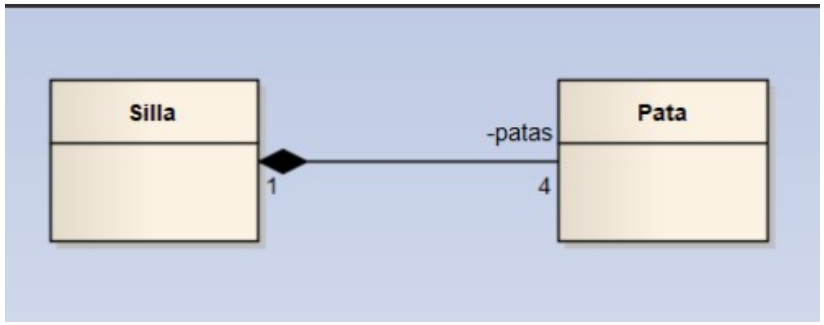
- Un objeto puede estar compuesto por otros objetos, en estos casos nos encontramos ante una asociación entre objetos llamada **Asociación de composición**.
- Esta asocia un objeto complejo con los objetos que lo constituyen, es decir, sus componentes.
- Existen dos formas de composición: fuerte o débil.
- La fuerte se la conoce como **composición** a secas y la débil como **agregación**.

- En la **composición fuerte** los componentes constituyen una parte del objeto compuesto y estos no pueden ser compartidos por otros objetos.
- Por tanto, la cardinalidad máxima a nivel del objeto compuesto es obligatoriamente uno.
- La supresión del objeto compuesto comporta la supresión de los componentes(dependencia).
- Una parte pertenece a un único agregado (exclusividad).
- Se representa con una línea con un rombo relleno.

- A continuación, se muestra la asociación de composición entre un ordenador y sus partes.
- Se considera que el ordenador se compone de una placa base, una o varias memorias, un teclado y uno o varios discos.



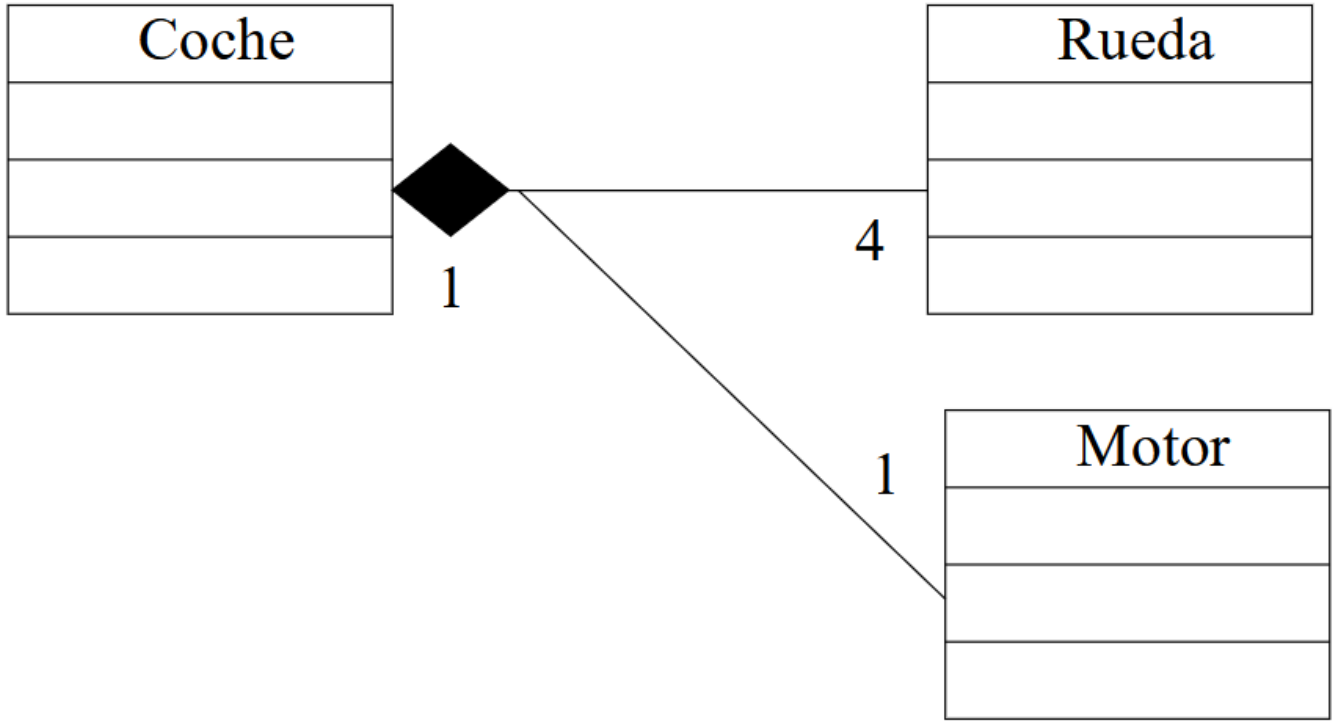
En este ejemplo tenemos la clase Silla que se compone de 4 patas.



Si el objeto Silla es destruido no queremos que sus patas anden “vagando” por la memoria, el objetivo es que la destrucción de silla elimine a su vez a todas las patas que componen la silla.

```java
public class Silla {
    private Pata[] patas;
    private int numPatas = 0;

    public Silla(){
        patas = new Pata[4];
    }

    public void agregarPata(String color, float peso){
        patas[numPatas] = new Pata(color, peso);
        numPatas++;
    }
}
```

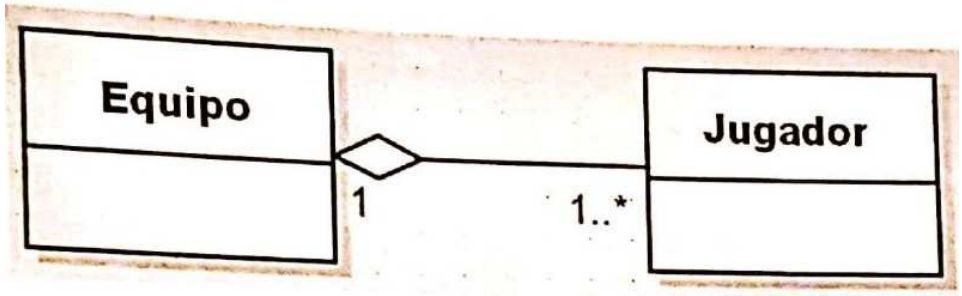
Composición: Ejemplo

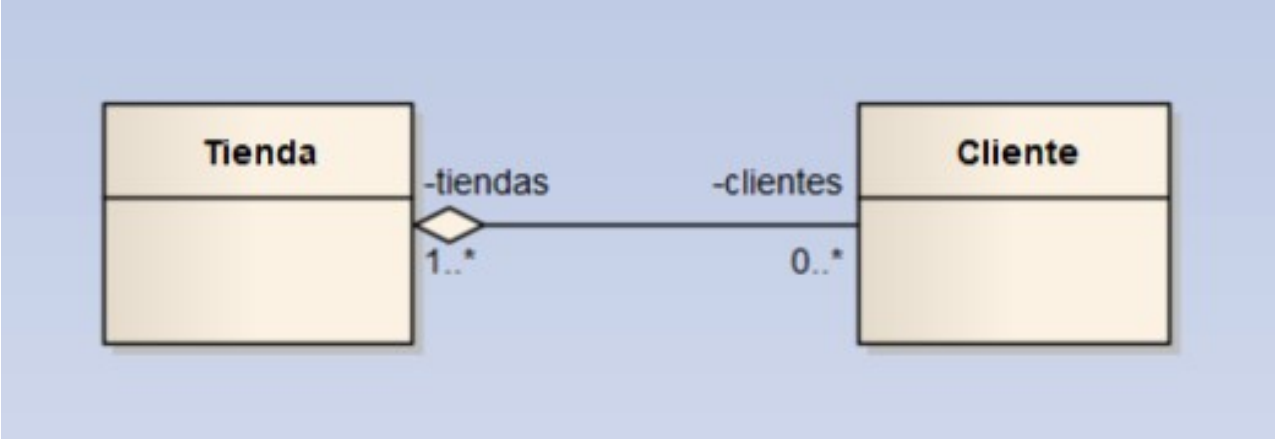


NO se permite tener “motores” ni “ruedas” sueltos; y al borrar el coche, se borra todo. Seguro que no se trata de un desguace.

4.4.6 AGREGACIÓN

- Es la **composición débil**.
- Los componentes pueden ser compartidos por varios compuestos y la destrucción del compuesto no implica la destrucción de los componentes.
- Se representa con un rombo vacío.
- A continuación, se muestra la asociación de agregación entre una clase Equipo, y la clase Jugador.
- Un Equipo está compuesto por jugadores, sin embargo, el jugador puede jugar también en otros equipos.
- Si desaparece el equipo, el jugador no desaparece.





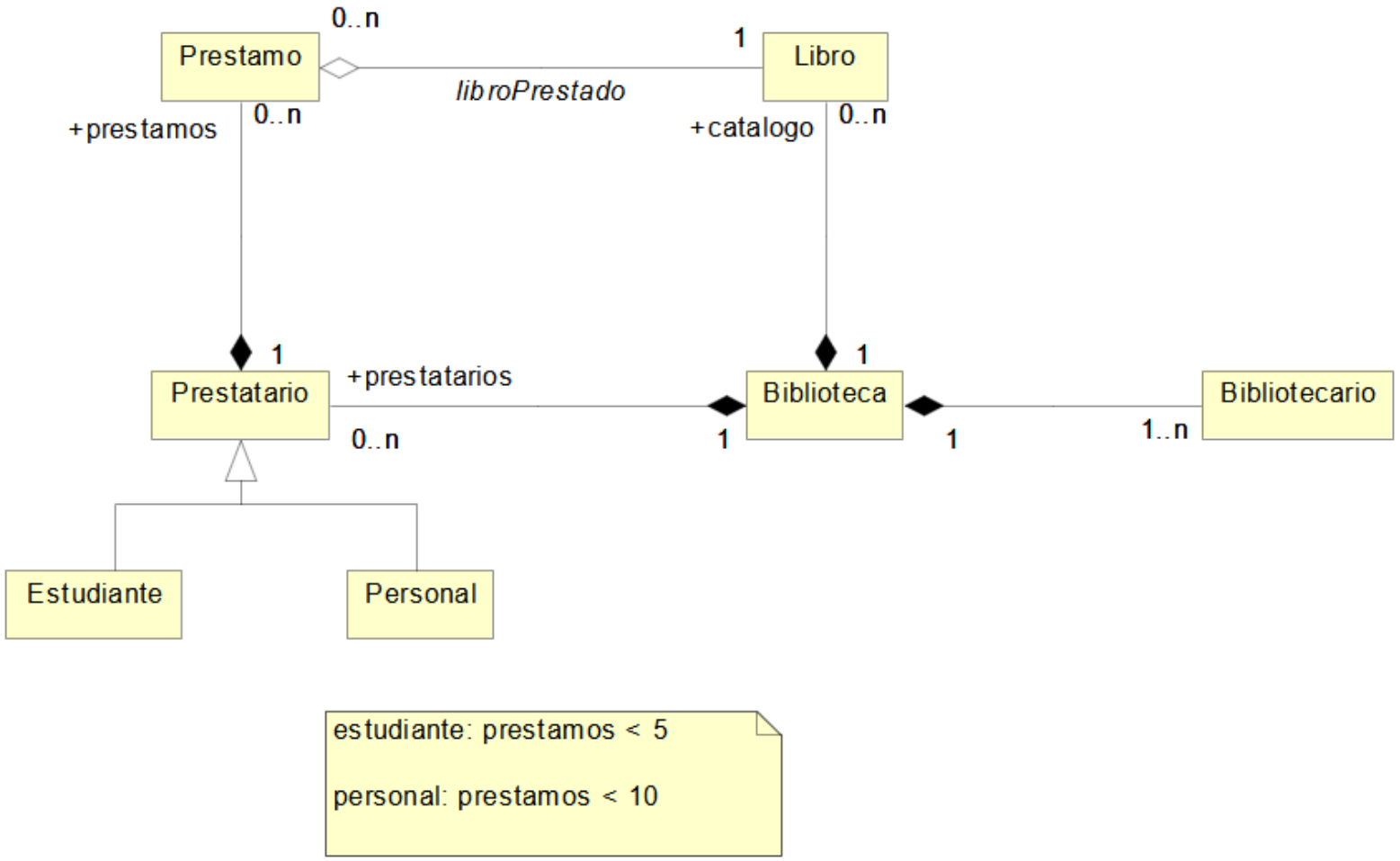
Código Java

```java
public class Tienda {
    private Cliente[] clientes = new Clientes[3];
    private int numClientes = 0;

    public Tienda(){}

    public void addCliente(Cliente cliente){
        clientes[numClientes] = cliente;
        numClientes++;
    }
}
```

- La siguiente tabla resume las diferencias entre la agregación y la composición

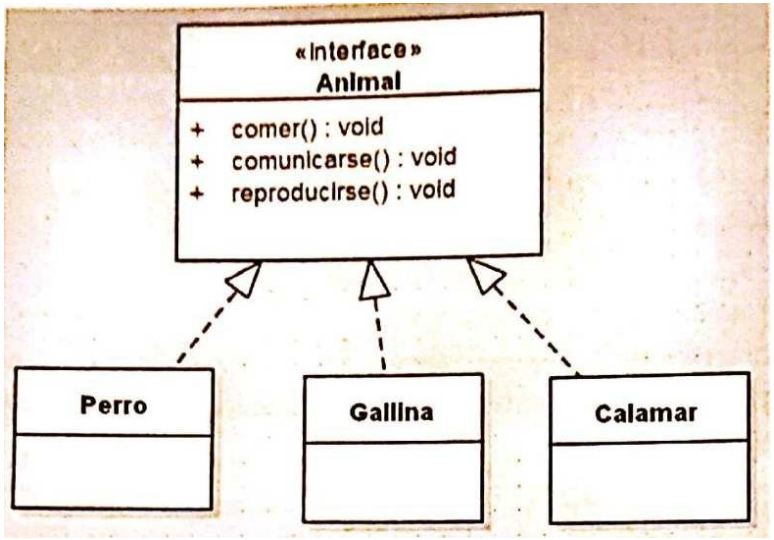
	Agregación	**Composición**
Varias asociaciones comparten los componentes	Sí	No
Destrucción de los componentes al destruir el compuesto	No	Sí
Cardinalidad a nivel de compuesto	Cualquiera	0..1 ó 1
Representación	Rombo transparente	Rombo negro

Clases Estructuradas: Ejemplo



4.4.6 REALIZACIÓN

- Una relación de realización es la relación de herencia existente entre una clase interfaz y la subclase que implementa de la interfaz.
- Esta relación de herencia se representa gráficamente mediante una flecha, con línea discontinua.
- En la figura se muestra una asociación de realización entre una clase interfaz Animal y las clases Perro, Gallina y Calamar.



4.4.6 REALIZACIÓN

- Cada subclase implementará los métodos de la interfaz.
- El código Java generado es el siguiente:

```
public interface Animal {

        public void comer();
        public void comunicarse();
        public void reproducirse();
}
```

```
public class Calamar implements Animal
{
    public Calamar(){  }
    public void reproducirse(){ }
    public void comunicarse(){ }
    public void comer(){ }
}
```

```
public class Perro implements Animal
{
   public Perro(){    }
   public void reproducirse(){  }
   public void comunicarse(){ }
   public void comer(){      }
}
```

```
public class Gallina implements Animal
{
        public Gallina(){   }
        public void reproducirse(){  }
        public void comunicarse(){   }
        public void comer(){  }
}
```

EJERCICIOS PROPUESTOS

1. Se desea diseñar un diagrama de clases sobre la información de las reservas de una empresa dedicada al alquiler de automóviles, teniendo en cuenta que:

- Un determinado cliente puede tener en un momento dado hechas varias reservas.
- De cada cliente se desean almacenar su DNI, nombre, dirección y teléfono. Además dos clientes se diferencian por un código único.
- Cada cliente puede ser avalado por otro cliente de la empresa.
- Una reserva la realiza un único cliente pero puede involucrar varios coches.
- Es importante registrar la fecha de inicio y final de la reserva, el precio del alquiler de cada uno de los coches, los litros de gasolina en el depósito en el momento de realizar la reserva, el precio total de la reserva y un indicador de si el coche o los coches han tenido algún desperfecto.
- Todo coche tiene siempre asignado un determinado garaje y una plaza que no puede cambiar. Del garaje se desea almacenar el nombre y la dirección. De la plaza se de almacenar el número y la planta.
- De cada coche se requiere la matricula, el modelo el color y la marca.
- Cada reserva se realiza en una determinada agencia.

2. El Ministerio de Defensa desea diseñar un sistema para llevar un cierto control de los soldados que realizan el servicio militar. Los datos significativos a tener en cuenta son:

- Un soldado se define por su código de soldado (único), su nombre y apellidos, y su graduación.
- Existen varios cuarteles, cada uno se define por su código de cuartel, nombre y ubicación.
- Hay que tener en cuenta que existen diferentes Cuerpos del Ejército (Infantería, Artillería, Armada, ...), y cada uno se define por un código de Cuerpo y denominación.
- Los soldados están agrupados en compañías, siendo significativa para cada una de éstas, el número de compañía y la actividad principal que realiza.
- Se desea controlar los servicios que realizan los soldados (guardias, imaginarias, cuarteleros, ...), y se definen por el código de servicio y descripción.

Consideraciones de diseño:

- Un soldado pertenece a un único cuerpo y a una única compañía, durante todo el servicio militar. A una compañía pueden pertenecer soldados de diferentes cuerpos, no habiendo relación directa entre compañías y cuerpos.
- Los soldados de una misma compañía pueden estar destinados en diferentes cuarteles, es decir, una compañía puede estar ubicada en varios cuarteles, y en un cuartel puede haber varias compañías. Eso si, un soldado sólo esta en un cuartel.
- Un soldado realiza varios servicios a lo largo de la milicia. Un mismo servicio puede ser realizado por más de un soldado (con independencia de la compañía), siendo significativa la fecha de realización.