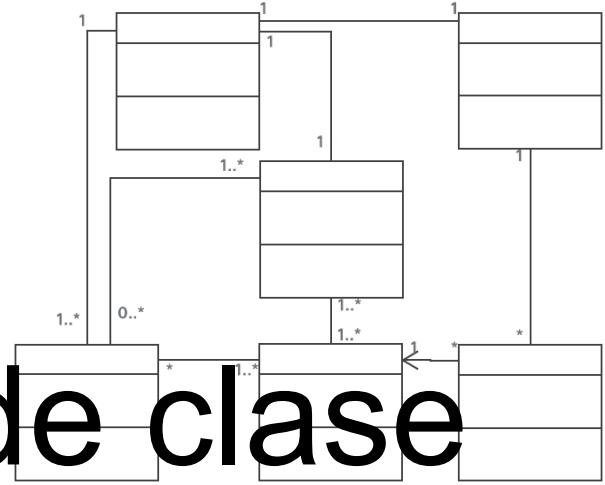


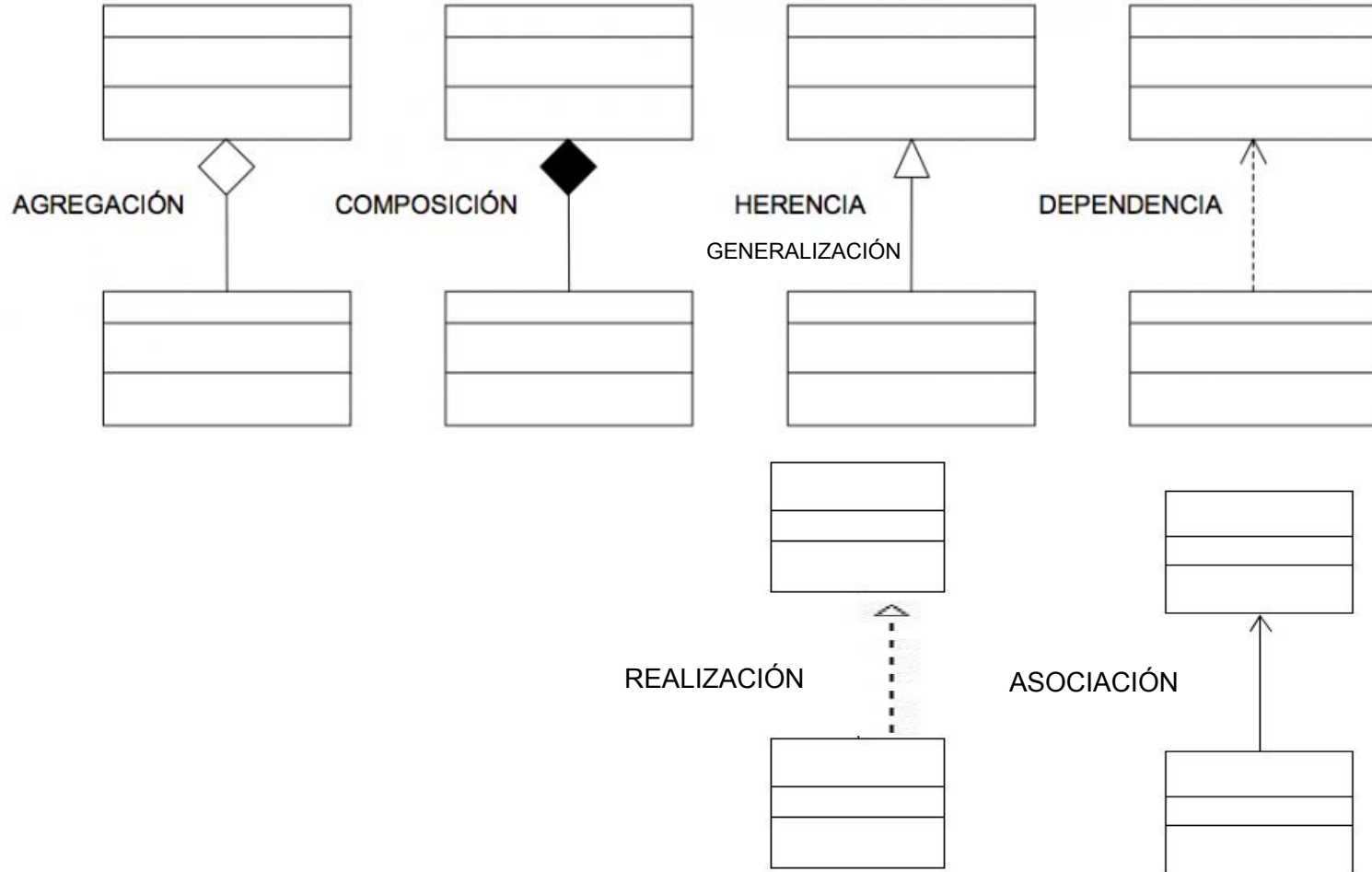
# UML Diagramas de clase

Sus relaciones



# Relaciones en los diagramas de clases

# Resumen



# RELACIÓN DE ASOCIACIÓN

# Asociación

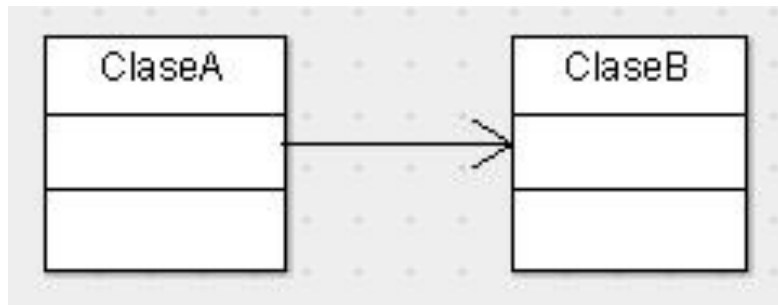
Es una relación de estructura entre clases, es decir, una entidad se construye a partir de otra u otras.

Aunque este tipo de relación es más fuerte que la [Dependencia](#) es más débil que la [Agregación](#), ya que el tiempo de vida de un objeto no depende de otro.

Se representa con una flecha continua que parte desde una clase y apunta a otra. El sentido de la flecha nos indica la clase que se compone (base de la flecha) y sus componentes (punta de la flecha).

Del diagrama anterior podemos observar que:

- La ClaseA depende de la ClaseB.
- La ClaseA está asociada a la ClaseB.
- La ClaseA conoce la existencia de la ClaseB pero la ClaseB desconoce que existe la ClaseA.
- Todo cambio en la ClaseB podrá afectar a la ClaseA.

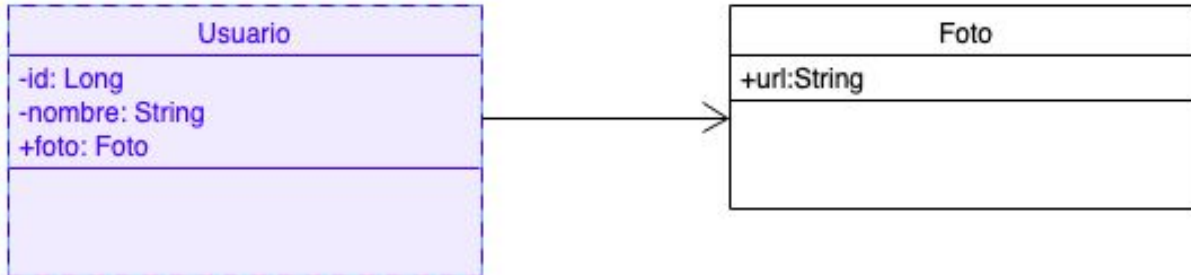


# Asociación

Es una relación estructural que describe una conexión entre objetos.

Se muestra como una línea continua

Puede o no tener flecha

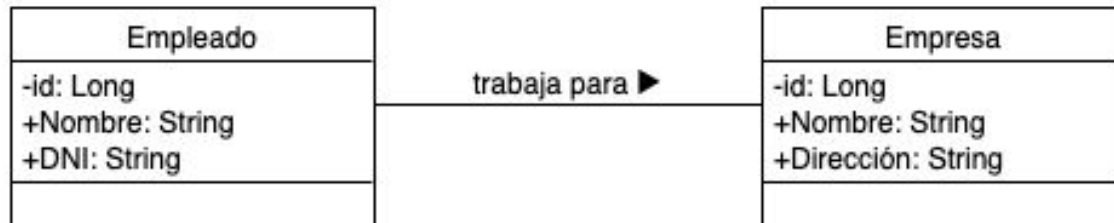


# Asociación (Association)

Por defecto la relaciones son bidireccional. Podemos entender esta relación como:

*El empleado trabaja para la empresa y en la empresa trabajan empleados*

Encima de la flecha podemos colocar un rol (trabaja\_para), que nos identifica mejor el tipo de relación.



# Asociaciones con multiplicidad

Podemos indicar la cardinalidad máxima y mínima (en cada sentido) de la relación: 1, 1..n, 0..n, m..n, \*, ...

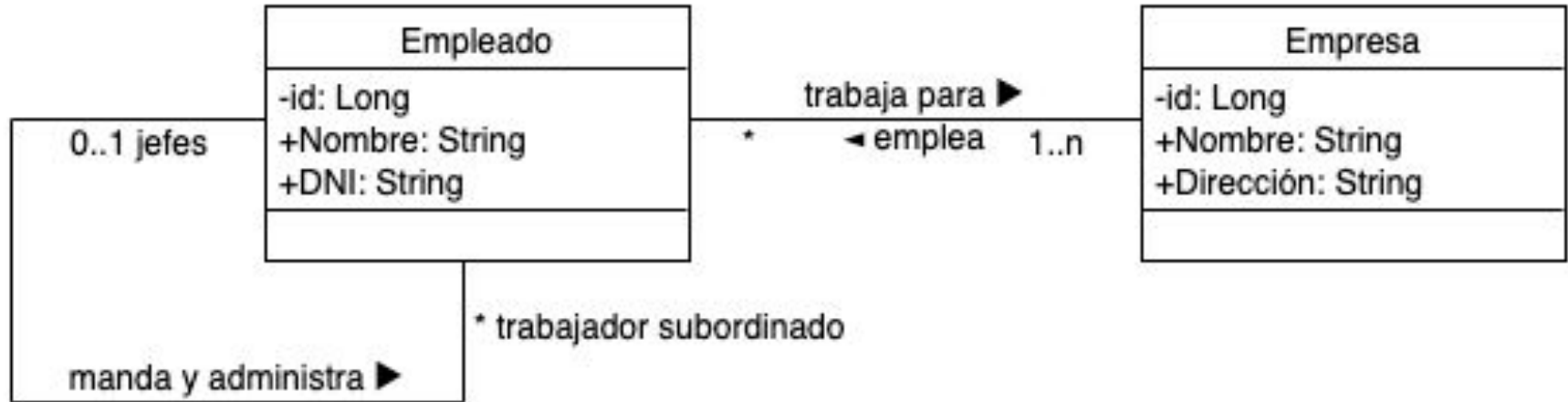


En este caso podemos leer: una empleado trabaja para 1 o más empresas, y en una empresa emplea a muchos empleados.



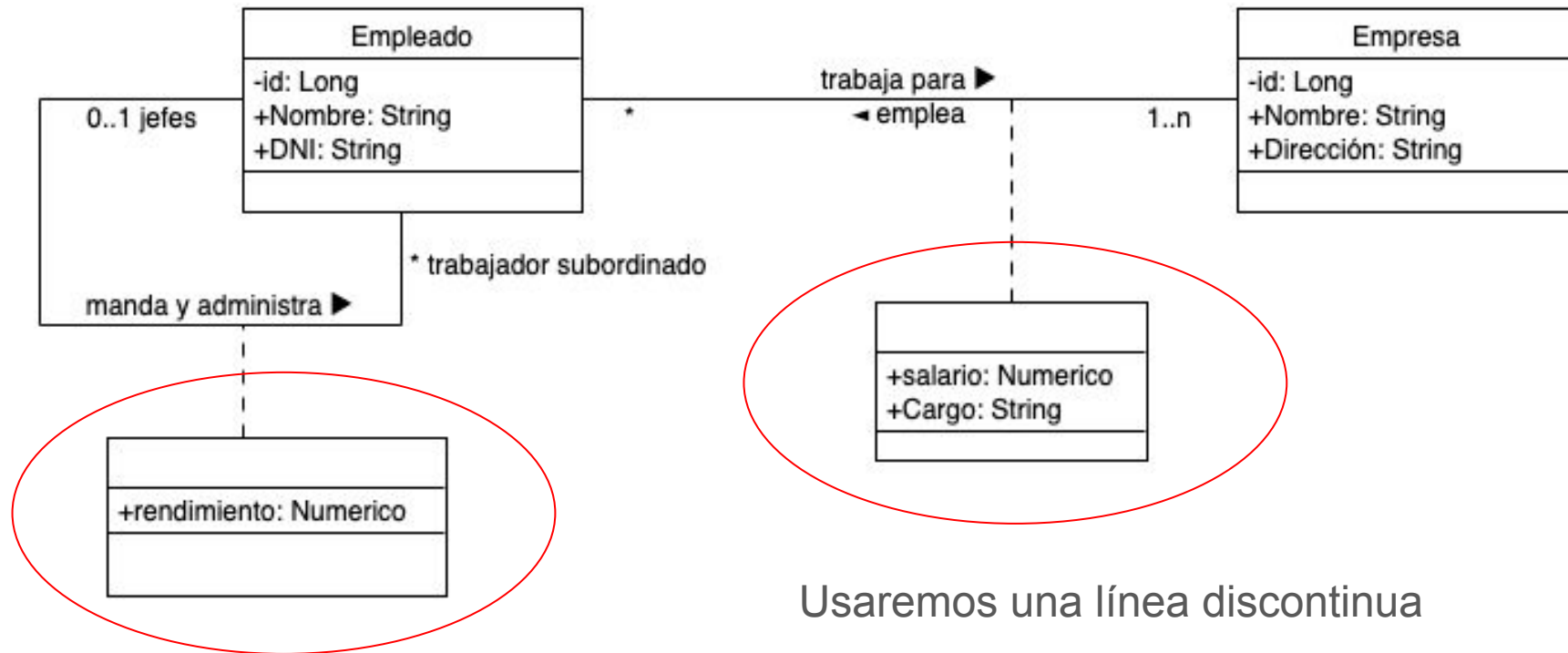
# Asociación consigo misma

Una clase puede estar asociada consigo misma. En este caso, estamos representando *que un empleado jefe manda y administra varios empleados subordinados*.



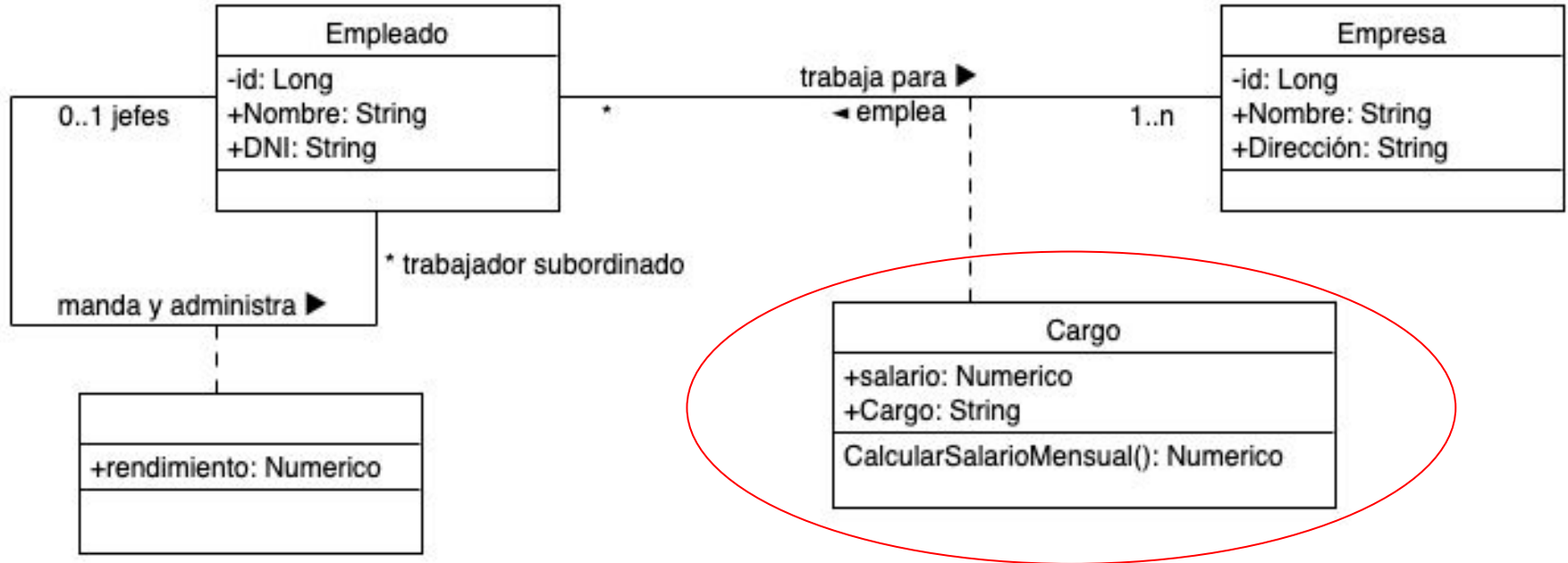
# Asociación con atributos de enlace

Podemos añadir propiedades que no pertenecen directamente a las clases del modelo, pero si pertenecen a la propia relación.

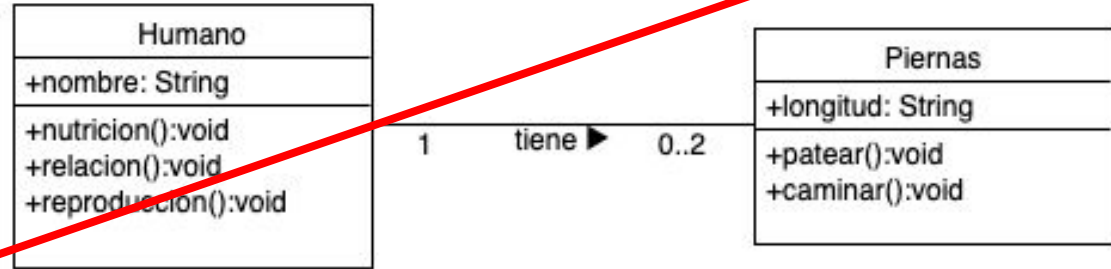


# Asociación con "clase asociación"

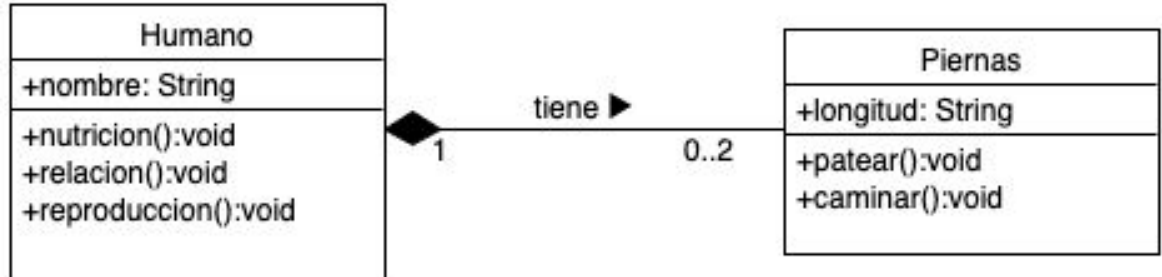
Básicamente consiste en convertir esos simples atributos en clases más complejas (que poseen atributos, métodos y otras relaciones con otros objetos)



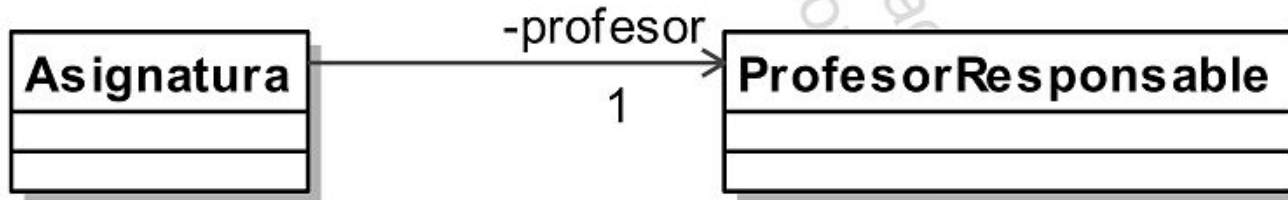
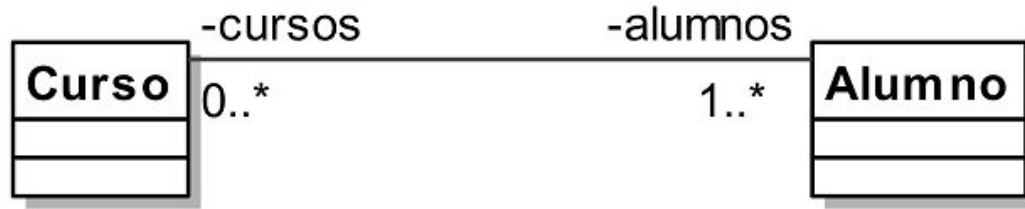
# No te confundas



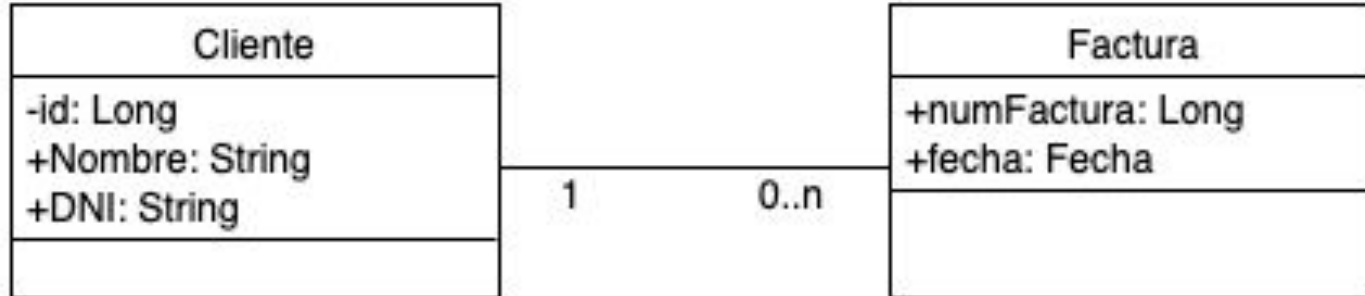
Este tipo de relación no sería de ~~asociación~~, sería de tipo **composición** (que veremos después).



# Ejemplos asociación

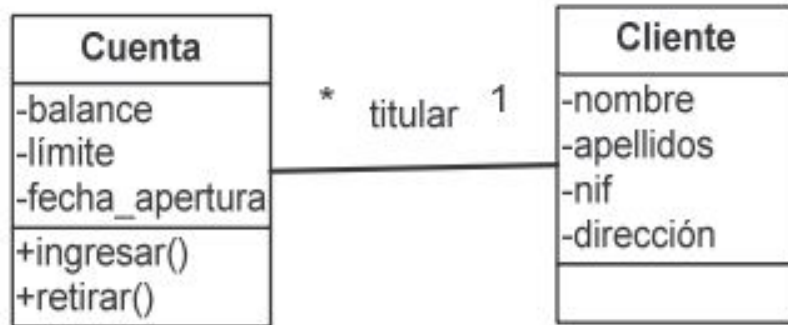


# Ejemplos asociación

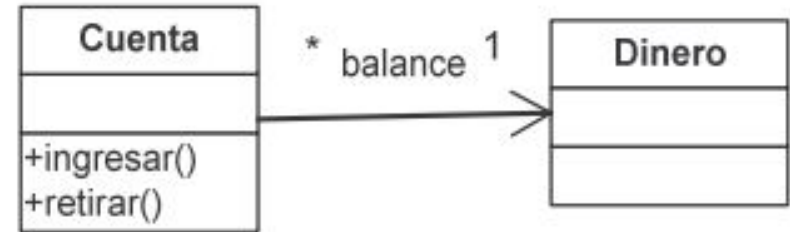


# Ejemplo asociación Bidireccional/unidireccional

Aunque suelen ser bidireccionales(ambos sentidos) podemos especificar que la relación sea unidireccional (restringir su navegación en un solo sentido)



**Asociación bidireccional**

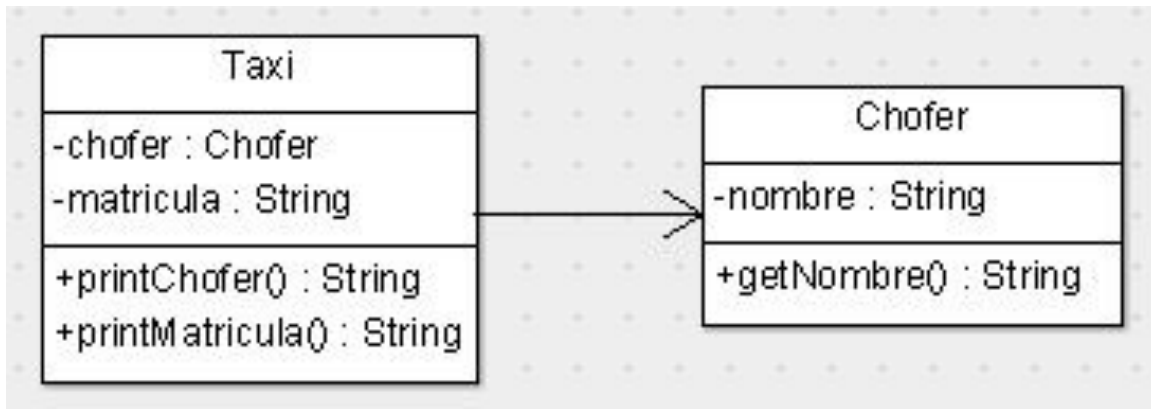


**Asociación unidireccional**

# Ejemplo Asociación unidireccional

```
class Chofer {  
    private String nombre;  
  
    public Chofer(String nombre) {  
        this.nombre = nombre;  
    }  
  
    public String getNombre() {  
        return this.nombre;  
    }  
}  
  
class Taxi {  
    private Chofer chofer;  
    private String matricula;  
  
    public Taxi(Chofer chofer, String matricula) {  
        this.chofer = chofer;  
        this.matricula = matricula;  
    }  
    public void printMatricula() {  
        System.out.println(this.matricula);  
    }  
    public void printChofer() {  
        String nombreChofer = this.chofer.getNombre();  
        System.out.println(nombreChofer);  
    }  
}
```

1. Tenemos una clase Taxi con un atributo matrícula.
2. Tenemos una clase Chofer con un atributo nombre.
3. Cada Taxi necesita ser conducido por un Chofer.
4. Taxi necesita acceder a algunos de los atributos de su Chofer (por ejemplo, su nombre).

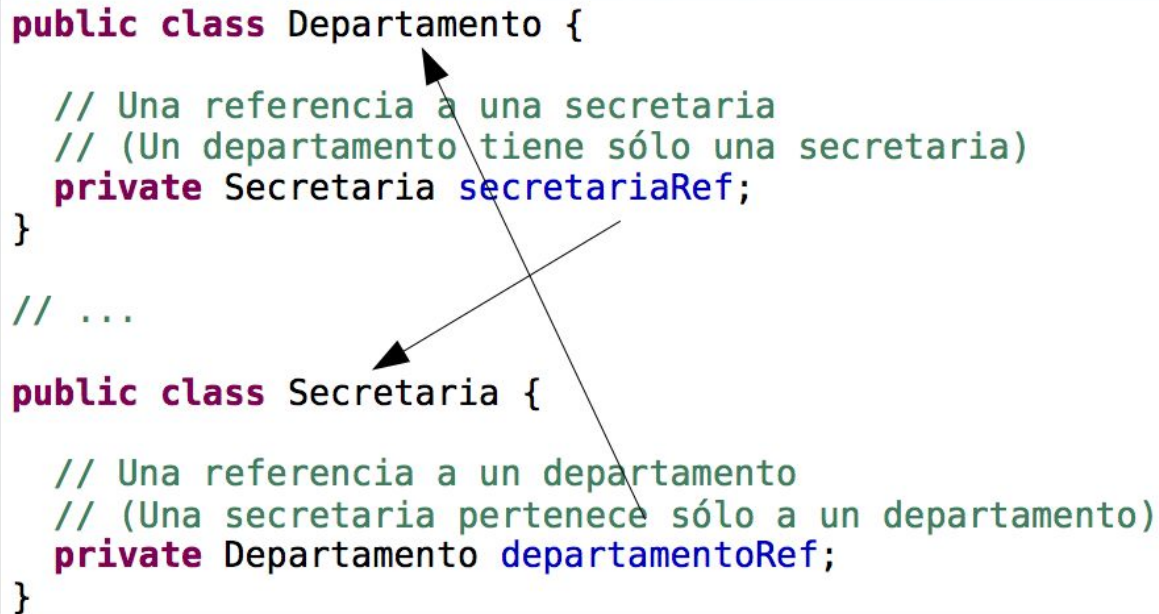


```
Chofer miChofer = new Chofer("Pedro");  
Taxi miTaxi = new Taxi(miChofer, "AHJ-1050");  
miTaxi.printChofer();  
miTaxi.printMatricula();
```



# Ejemplo asociación bidireccional

```
public class Departamento {  
    // Una referencia a una secretaria  
    // (Un departamento tiene sólo una secretaria)  
    private Secretaria secretariaRef;  
}  
  
// ...  
  
public class Secretaria {  
    // Una referencia a un departamento  
    // (Una secretaria pertenece sólo a un departamento)  
    private Departamento departamentoRef;  
}
```



# Enumeraciones

Se puede indicar que una clase es una enumeración, indicándolo con el estereotipo <<enumeration>>

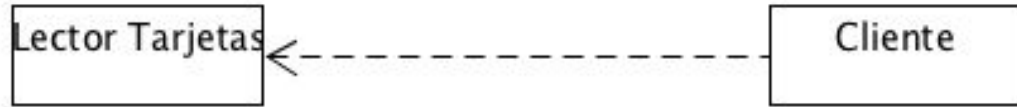


# RELACIÓN DE DEPENDENCIA

# Dependencia (dependency)

Es una **asociación de uso**, es decir que **una clase utiliza a otra**. Y si esta **última se altera, la anterior se puede ver afectada**.

En código (ej java) se suelen traducir principalmente como las clases donde se hace la instanciación de un objeto.



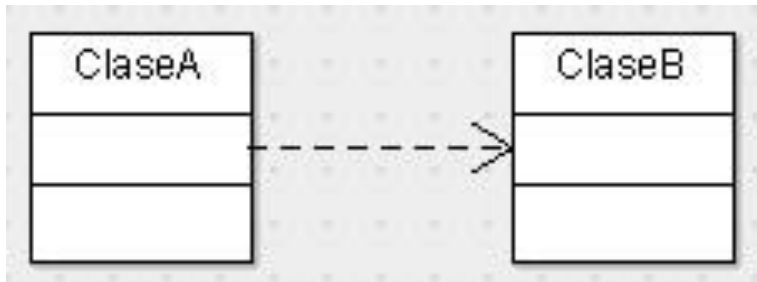
Una relación de dependencia se utiliza entre dos clases o entre una clase y una interfaz, e indica que una clase requiere de otra para proporcionar alguno de sus servicios.

# Dependencia

Se representa con una **flecha discontinua** que parte desde una clase y apunta a otra. **El sentido de la flecha nos indica quien usa a quien.**

Del diagrama anterior podemos observar que:

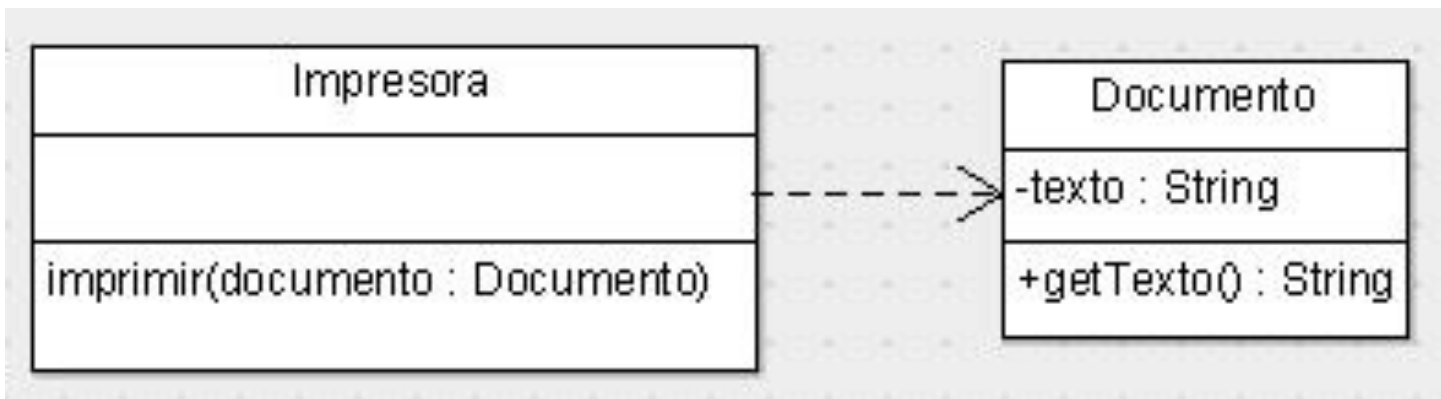
- La ClaseA usa a la ClaseB.
- La ClaseA depende de la ClaseB.
- Dada la dependencia, todo cambio en la ClaseB podrá afectar a la ClaseA.
- La ClaseA conoce la existencia de la ClaseB pero la ClaseB desconoce que existe la ClaseA.



# Ejemplo dependencia

Veamos un ejemplo para entender mejor este tipo de relación. Supongamos lo siguiente:


1. Tenemos una clase Impresora..
2. Tenemos una clase Documento con un atributo texto.
3. La clase Impresora se encarga de imprimir los Documentos. Tiene un método imprimir que tiene como parámetro un documento.



# Ejemplo dependencia (continuación)

```
class Documento {  
    private String texto;  
  
    public Documento(String texto) {  
        this.texto = texto;  
    }  
  
    public String getTexto() {  
        return this.texto;  
    }  
}
```

```
class Impresora {  
  
    public Impresora() {  
  
    }  
  
    public void imprimir(Documento documento) {  
        String texto = documento.getTexto();  
        System.out.println(texto);  
    }  
}
```



```
Documento miDocumento = new Documento("Hello World!");  
Impresora milImpresora = new Impresora();  
milImpresora.imprimir(miDocumento);
```

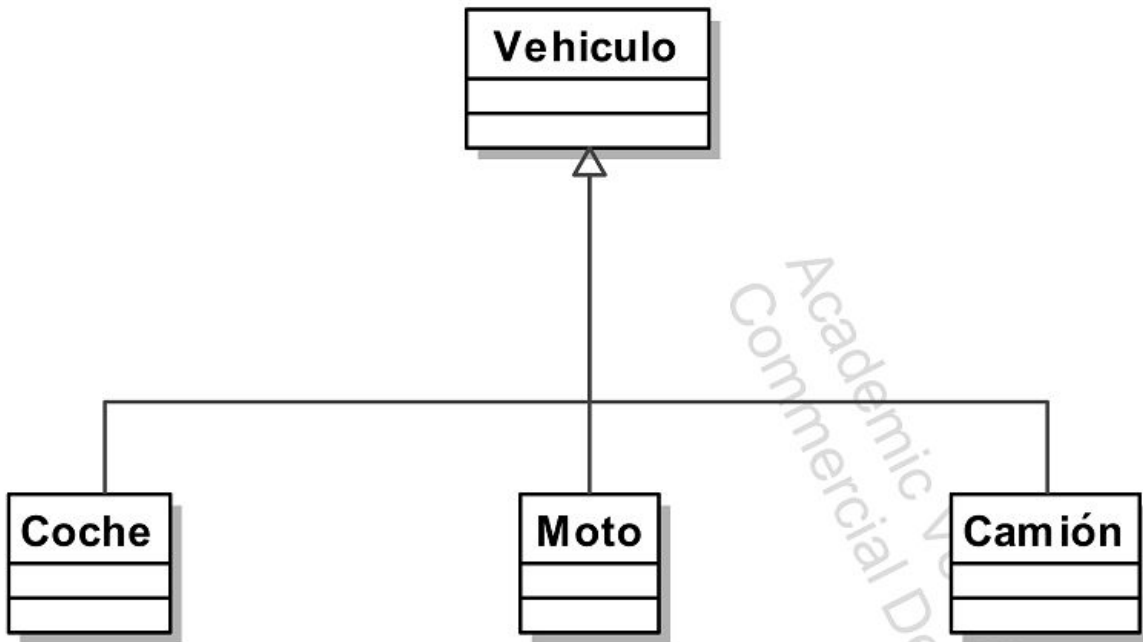
# RELACIÓN DE GENERALIZACIÓN



# Generalización

Esta relación **representa la herencia** o la extensión de una clase de otra. En la siguiente imagen podemos ver un ejemplo.

Las subclases heredan características de las clases de las que se derivan y añaden características específicas que las diferencian.



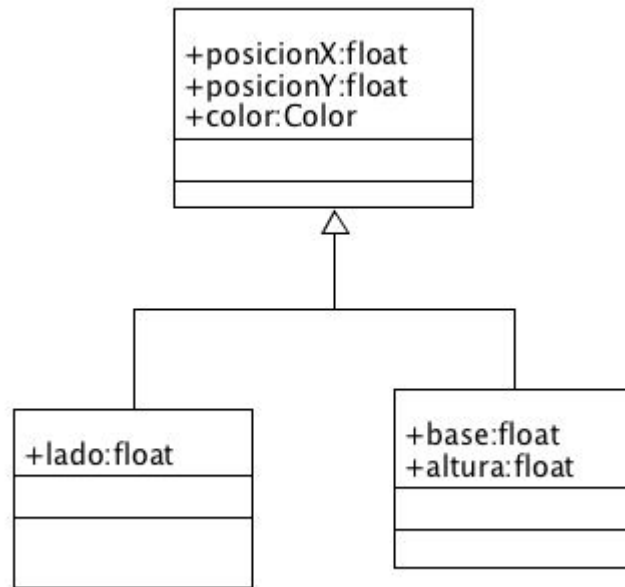
# Ejemplo generalización

```
import java.awt.Color;
```

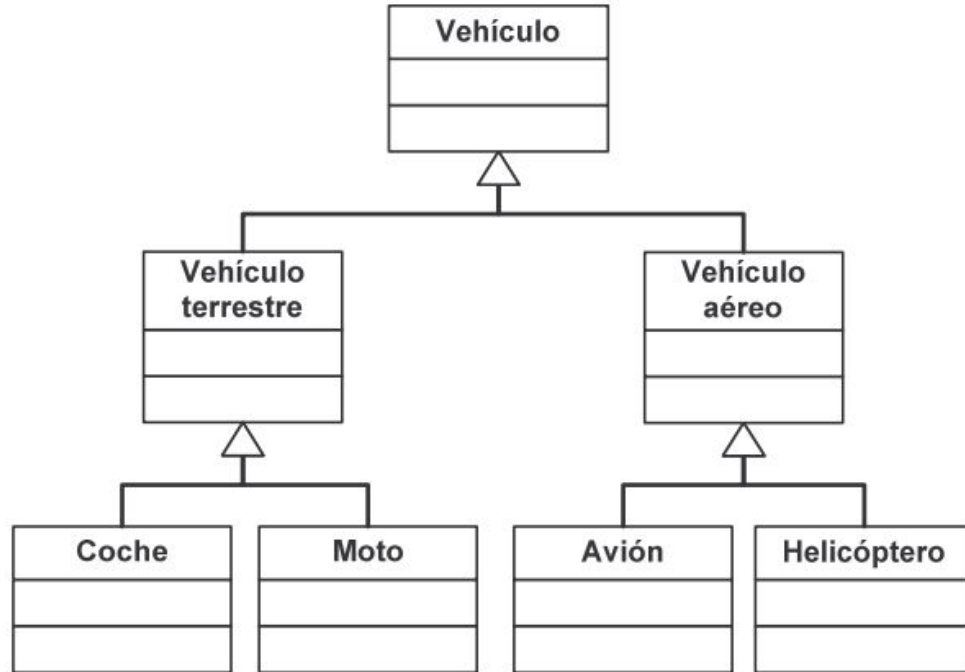
```
public class Forma {  
    public float posicionX;  
    public float posicionY;  
    public Color color;  
}
```

```
class cuadrado extends Forma{  
    public float lado;  
}
```

```
class rectangulo extends Forma{  
    public float base;  
    public float altura;  
}
```



# Jerarquía de clases



**Las clases se organizan  
en una estructura jerárquica formando una taxonomía.**

# Realización

# Realización

Es una relación de contrato con otra clase.

Se la utiliza para implementar una **interfaz**.

En lenguajes como java o php utilizamos la palabra reservada “implements”

Se representa por una flecha discontinua con punta cerrada.



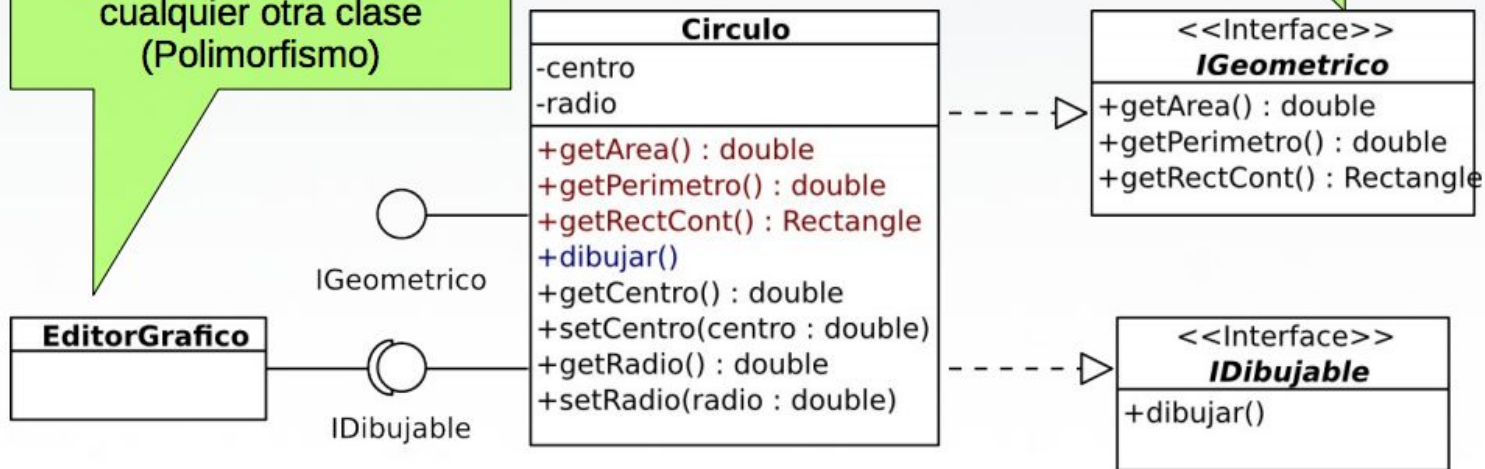
```
interface Dibujable{  
    void dibujarPantalla();  
}  
public class Cuadrado implements Dibujable{  
    void dibujarPantalla(){  
        ...  
    }  
}
```

# Interfaz

**Interfaz:** Clase asociada que describe su comportamiento visible. Conjunto de métodos que describen el comportamiento visible de una clase

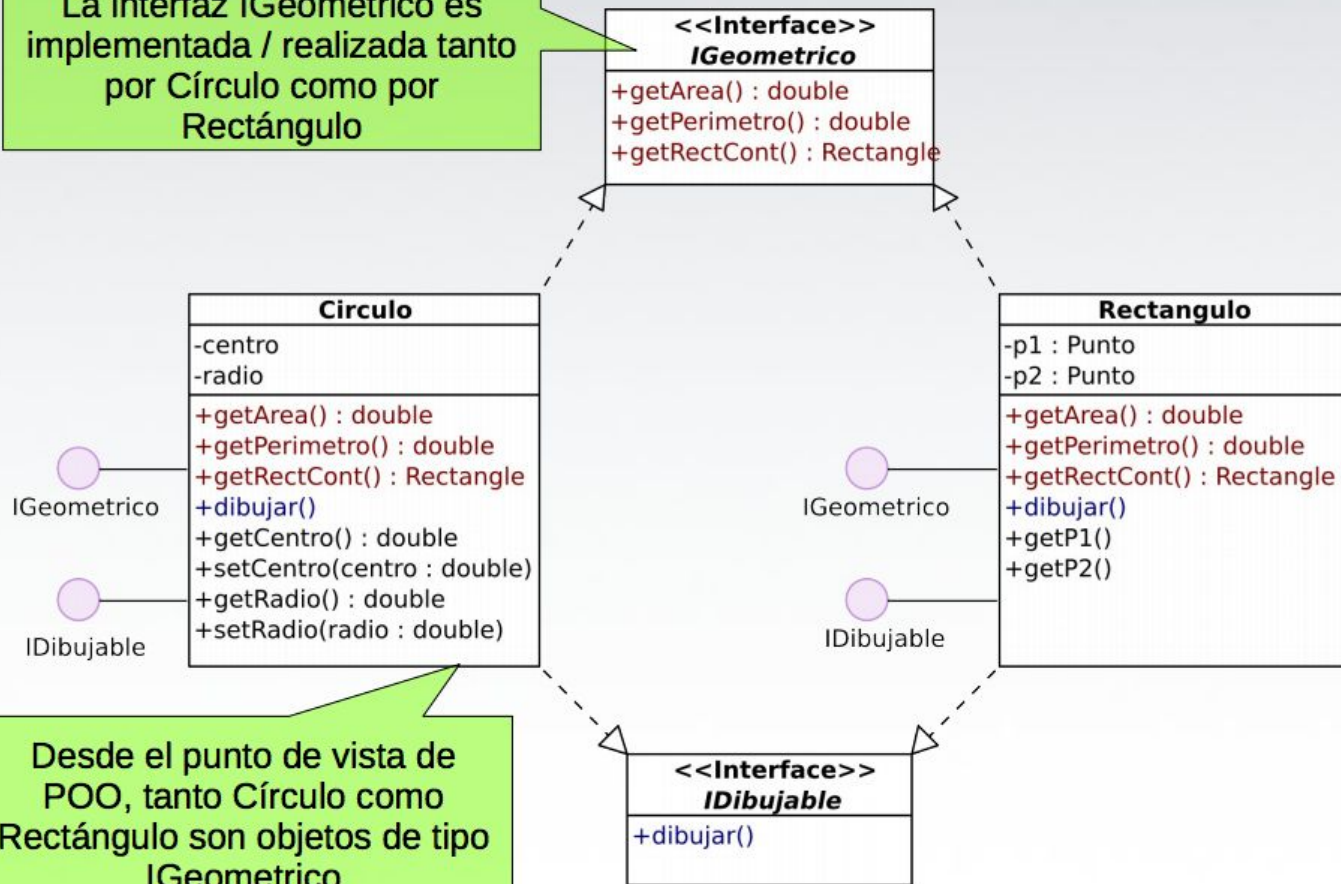
EditorGrafico es una clase que usa la interfaz Idibujable, independientemente que la implemente un Círculo o cualquier otra clase (Polimorfismo)

<<interface>> es un estereotipo



# Interfaz

La interfaz IGeometrico es implementada / realizada tanto por Círculo como por Rectángulo



Desde el punto de vista de POO, tanto Círculo como Rectángulo son objetos de tipo IGeometrico

# Interfaz ejemplo

```
import java.awt.Point;
import java.awt.Rectangle;

public class Circulo implements IGeometrico, IDibujable {

    private double centro;
    private double radio;

    public double      getArea()          { /* de IGeometrico */ }
    public double      getPerimetro()     { /* de IGeometrico */ }
    public Rectangle   getRectCont()      { /* de IGeometrico */ }

    public void dibujar()                  { /* de IDibujable */ }

    public Point       getCentro()         { /* de circulo */ }
    public void        setCentro(...)      { /* de circulo */ }

    public double      getRadio()          { /* de circulo */ }
    public void        setRadio(...)       { /* de circulo */ }
}
```



# Interfaz ejemplo

```
import java.awt.Rectangle;

public interface IGeometrico {

    public double getArea();

    public double getPerimetro();

    public Rectangle getRectCont();
}
```

---

```
public interface IDibujable {

    public void dibujar();

}
```

# Interfaz ejemplo

```
List<IDibujable> elementosDibujar;  
  
// ...  
  
for (IDibujable dibujable :  
    elementosDibujar) {  
    // No importa si dibujable es  
    // un círculo, rectángulo, etcétera  
    // Los puedo manejar a todos igual  
    // porque tienen una interfaz en común  
    dibujable.dibujar();  
}
```

Algunos de estos  
son círculos, otros  
son rectángulos,  
estrellas, líneas,  
etcétera...  
Pero todos  
implementan la  
interfaz IDibujable

¡El acto de magia de las interfaces y el polimorfismo!

# RELACIÓN DE AGREGACIÓN Y COMPOSICIÓN

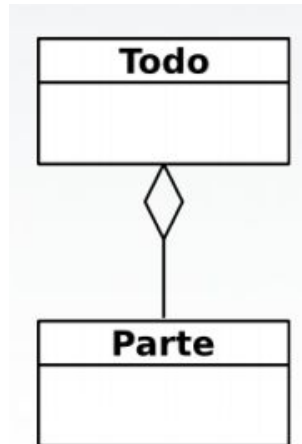
# Relación de agregación

# Relación de agregación

Es parecido a la *asociación*, pero **con más semántica...**

La relación representa: **está formado por...**

**En este caso los componentes son reutilizados por distintos compuestos**



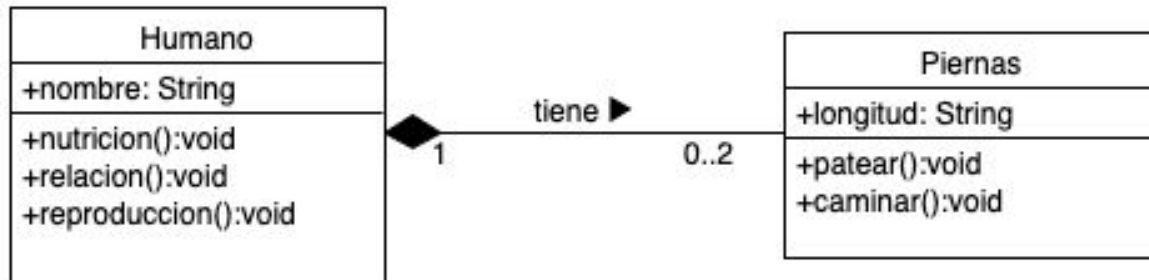
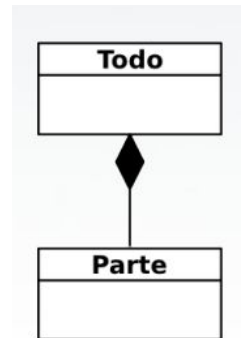
# Relación de composición

Es parecido a la *asociación*, pero **con más semántica...**

La relación representa: **esta compuesto por...**

**Un componente solo puede pertenecer a un compuesto**

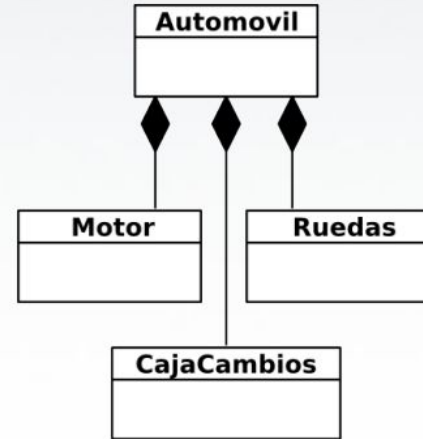
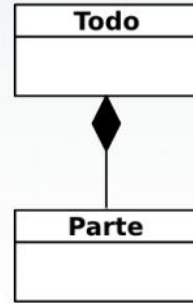
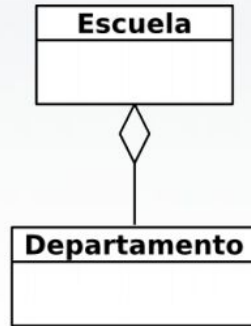
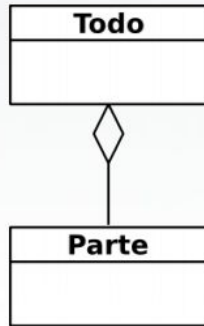
**La destrucción del compuesto implica la destrucción de sus componentes**



# Agregación VS composición

**Agregación:** Es una relación en la que una de las clases representa un todo y la otra representa parte de ese todo

**Composición:** Es una forma más fuerte de la agregación, en la que el todo no puede existir sin sus partes



¿Cómo se implementan?

¿Cuál es la diferencia con las asociaciones?

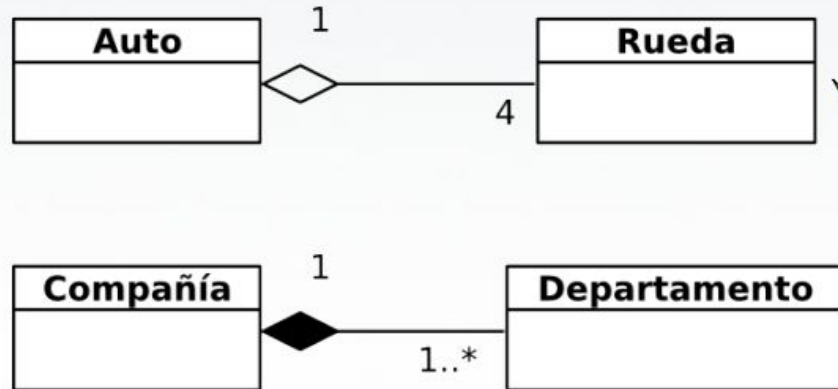
# Composición VS agregación

**Composición:** Las partes no pueden existir sin el todo

***En contradicción con el ejemplo anterior:***

**Composición:** El todo no puede existir sin las partes

***(Ejemplo Anterior)***



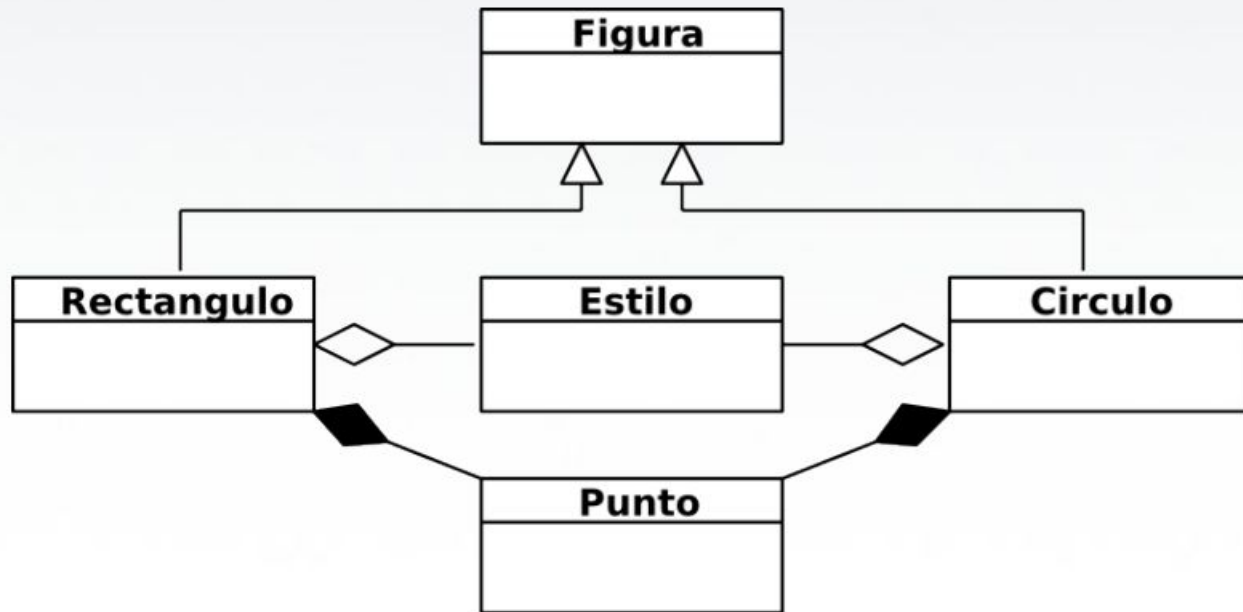
¿La parte (La rueda) puede existir sin el todo?



# Agregación y composición

**Agregación:** ¡Las partes pueden ser compartidas por varios todos!

**Composición:** ¡Las partes NO pueden ser compartidas por varios todos!



# RELACIÓN DE AGREGACIÓN

# Agregación

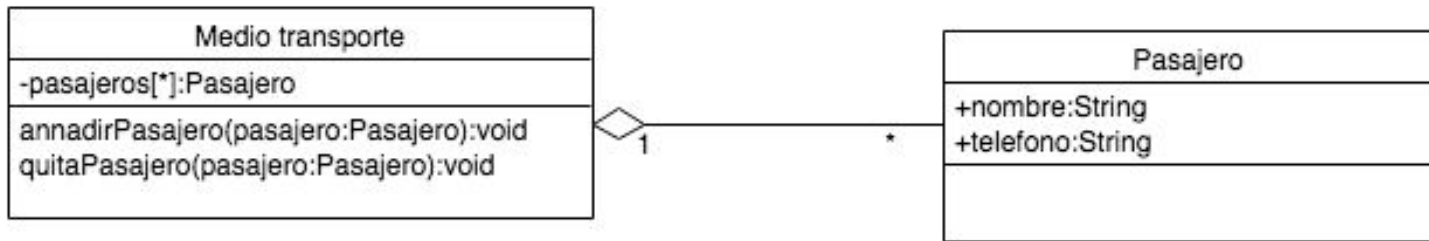
Se representa con una flecha que parte de una clase a otra en cuya base hay un rombo de color blanco.

¡La Agregación son siempre colecciones, o arrays! O algo que sirva de contenedor para "agregar" más de un objeto, aunque agreguemos uno solo. (si no sería settear y no agregar, add)

La Agregación cuenta con dos métodos: uno para "agregar" un solo objeto a la lista, y el otro para quitarlo de la misma.

- Tiene objetos que son pasados como parámetros a dicho método, son instanciados fuera de la clase.

La agregación puede, como no, tener los métodos setter y getter, mientras que la Asociación siempre los tiene y obtienen una variable de referencia del mismo tipo de la variable de instancia o de clase.



# Agregación (aggregation)

## Composición débil

Es una relación que se derivó de la asociación, por ser igualmente estructural, es decir que contiene un atributo, que en todos los casos, será una colección, es decir un Array, Vector, Collections, etc, y además de ello **la clase que contiene la colección debe tener un método que agregue los elementos a la colección**. También se puede leer como que un medio de transporte tiene varios pasajeros.

Nos está diciendo que los objetos pasajero forman parte del objeto medio de transporte. Pero, **su ciclo de vida no está atado al del objeto medio de transporte**. Es decir si el Autobús se destruye los pasajeros pueden seguir existiendo independientemente, ( o por lo menos por eso rezaríamos)

# Ejemplo Agregación

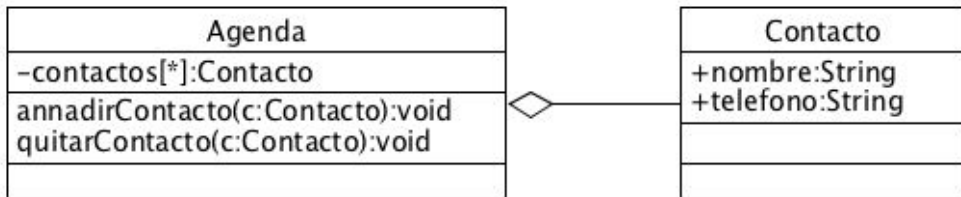
```
import java.util.ArrayList;
```

```
public class Contacto {  
    public String nombre;  
    public String telefono;  
    Contacto(String nombre, String telefono){  
        this.nombre=nombre;  
        this.telefono=telefono;  
    }  
}
```

```
}
```

```
class Agenda {  
    private ArrayList<Contacto> contactos;  
  
    public void annadirContacto(Contacto c){  
        contactos.add(c);  
    }  
  
    public void quitarContacto(Contacto c){  
        contactos.remove(c);  
    }  
}
```

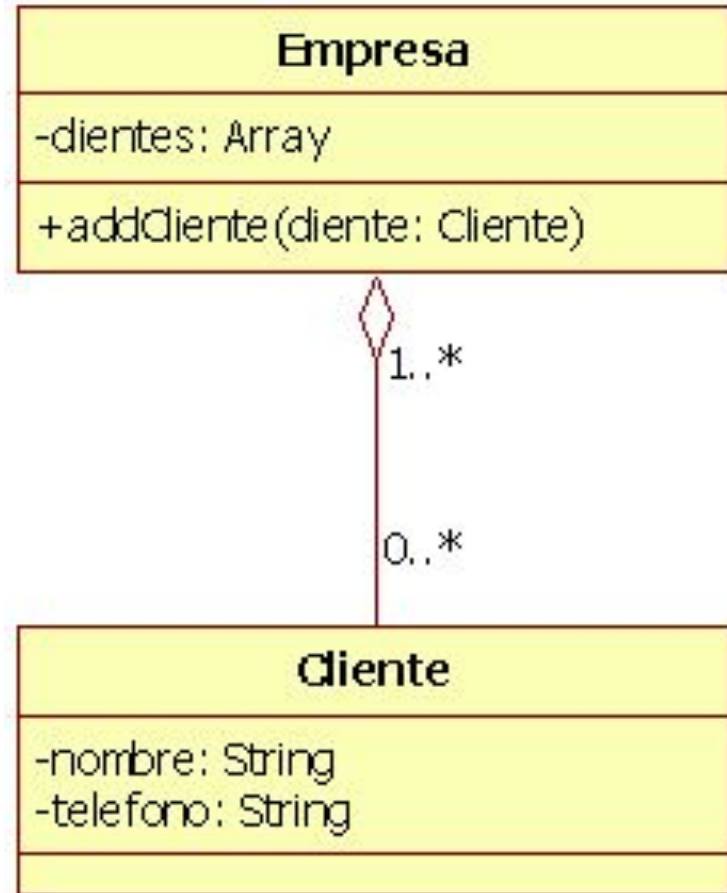
- Tenemos una clase Agenda.
- Tenemos una clase Contacto.
- Una Agenda agrupa varios Contactos



```
Contacto c1 = new Contacto("Angel","111");  
Contacto c2 = new Contacto("Pepe","222");
```

```
Agenda agen = new Agenda();  
agen.annadirContacto(c1);  
agen.annadirContacto(c2);
```

# Agregación ejemplo



# RELACIÓN DE COMPOSICIÓN

# Composición (composition)

Se representa con una flecha que parte de una clase a otra en cuya base hay un rombo de color negro.

¡La composición **son siempre colecciones, o arrays!** O algo que sirva de contenedor para "agregar" más de un objeto, aunque agreguemos uno solo. (si no sería settear y no agregar, add)

La composición cuenta con dos métodos: uno para "agregar" un solo objeto a la lista, y el otro para quitarlo de la misma.

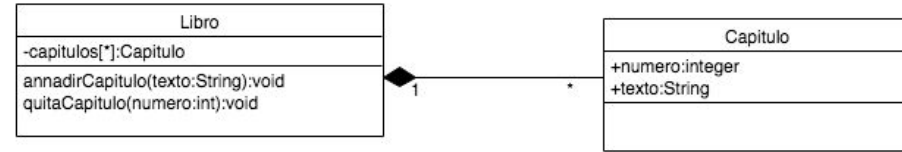
- A diferencia de la agregación, esta clase **SI se hace responsable de la creación de los objetos** de las clases a las cuales apuntan.



# Composición (composition)

Es un tipo de asociación, pero podemos decir que **son agregaciones fuertes**. La diferencia con las agregaciones es que no tiene sentido que el objeto contenedor siga existiendo si no existen los objetos contenidos.

Mucho se ha discutido acerca de las agregaciones y las composiciones, ya que algunos sostienen que los lenguajes orientados a objetos, tienen garbage collector, por lo que no necesitan métodos de destrucción de los objetos



# Ejemplo Composición

```
import java.util.ArrayList;
```

```
public class Contacto {  
    public String nombre;  
    public String telefono;  
    Contacto(String nombre, String telefono){  
        this.nombre=nombre;  
        this.telefono=telefono;  
    }  
}
```

```
class Agenda {  
    private ArrayList<Contacto> contactos;  
  
    public void annadirContacto(String nombre){  
        Contacto c = new Contacto();  
        c.nombre = nombre;  
        contactos.add(c);  
    }  
    public void quitarContacto(int indice){  
        contactos.remove(indice);  
    }  
}
```

- Tenemos una clase Agenda.
- Tenemos una clase Contacto.
- Una Agenda agrupa varios Contactos



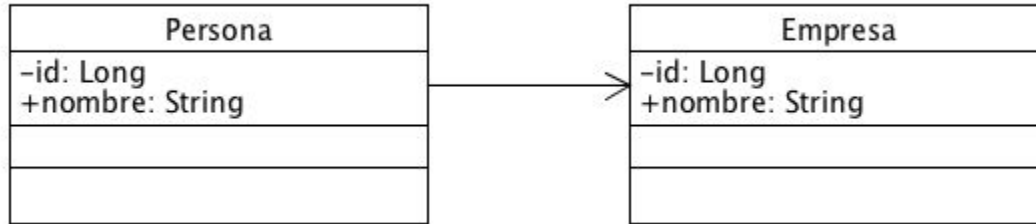
```
Agenda agen = new Agenda();  
agen.annadirContacto("Angel");  
agen.annadirContacto("Jose");
```

Ángel González M.

# Modificadores en las relaciones

# Propiedad (Navegabilidad)

La navegación desde una clase a la otra se representa poniendo una flecha sin relleno en el extremo de la línea, indicando el sentido de la navegación.

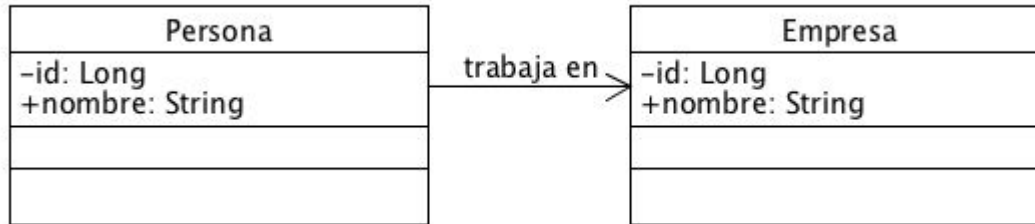


# Propiedad (Orden)

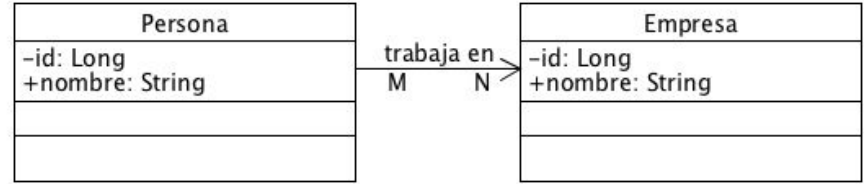
Orden: Se puede especificar si las instancias guardan un orden con la palabra clave '{ordered}'. Si el modelo es suficientemente detallado, se puede incluir una restricción que indique el criterio de ordenación.

# Propiedad (Rol)

Rol, o nombre de la asociación, que describe la semántica de la relación en el sentido indicado. Por ejemplo, la asociación entre Persona y Empresa recibe el nombre de trabaja para, como rol en ese sentido.



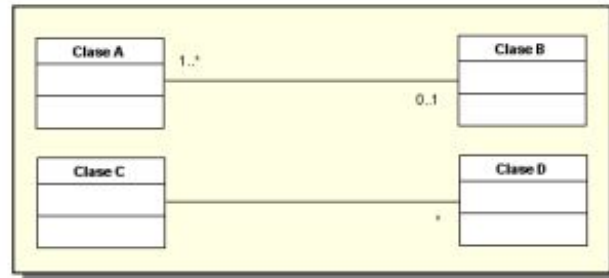
# Propiedad (Multiplicidad)



*Multiplicidad*, que describe la cardinalidad de la relación, es decir, especifica cuántas instancias de una clase están asociadas a una instancia de la otra clase. Los tipos de multiplicidad son: Uno a uno, uno a muchos y muchos a muchos.

## Multiplicidad

- 1** Un elemento relacionado.
- 0..1** Uno o ningún elemento relacionado.
- 0..\*** Varios elementos relacionados o ninguno.
- 1..\*** Varios elementos relacionados pero al menos uno.
- \*** Varios elementos relacionados.
- M..N** Entre M y N elementos relacionados.



# Ejemplo completo



Conductor

Pasejero

Butaca

Autobus

Medio de transporte

Tren

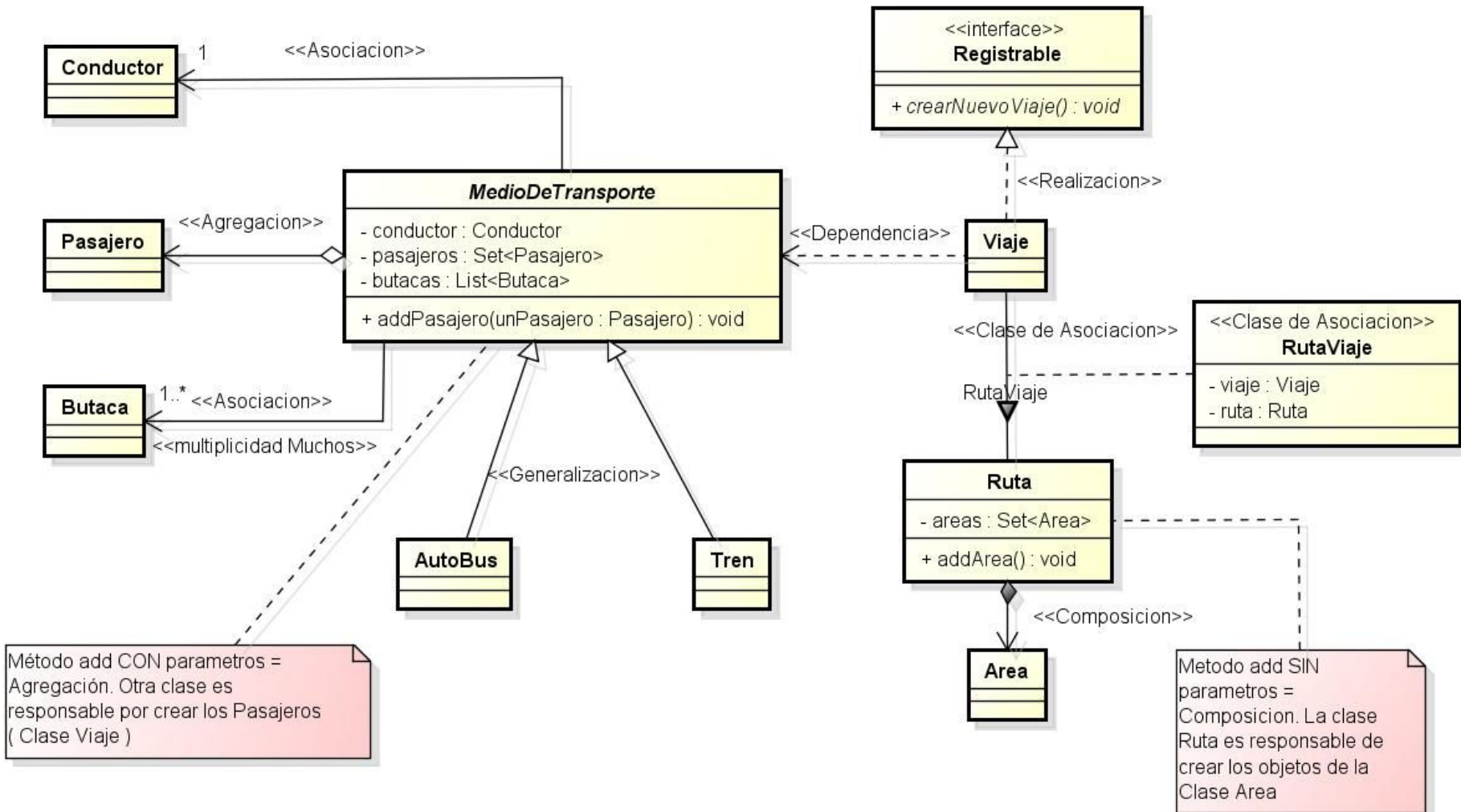
Registrable

Viaje

Ruta

Area

RutaViaje



# Asociación



Es generalmente, una relación estructural entre clases, es decir, que en el ejemplo, existe un atributo de la clase medio de transportes, que es del tipo Conductor.

La navegabilidad nos muestra dónde está ubicado el atributo. La multiplicidad en una Asociación dice bastante, ya que de eso dependerá si el atributo, es una colección o simplemente una variable de referencia a un objeto.

# Agregación



Es una relación que se derivó de la asociación, por ser igualmente estructural, es decir que contiene un atributo, que en todos los casos, será una colección, es decir un Array, Vector, Collections, etc, y además de ello la clase que contiene la colección **debe tener un método que agregue los elementos a la colección.**

También se puede leer como que un medio de transporte tiene varios pasajeros. Nos está diciendo que los objetos pasajero forman parte del objeto medio de transporte. Pero, **su ciclo de vida no está atado al del objeto medio de transporte.** Es decir si el Autobús se destruye los pasajeros pueden seguir existiendo independientemente.

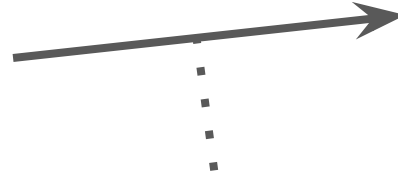
# Composición



Al igual que en la agregación, es una relación estructural pero se le suma, que tiene un método de destrucción de los objetos. Y a diferencia de la asociación, el ciclo de vida del objeto área está relacionado con el del objeto ruta. Es decir que si la ruta de viaje se levanta, las áreas que surgían a partir de ella desaparecen. También se puede leer como que una ruta tiene varias áreas de cobertura.

- Hay mucho debate acerca de esto.
- Algunos sostienen que los lenguajes orientados a objetos, tienen garbage collector, por lo que no necesitan métodos de destrucción de los objetos (relacionados a los ciclos de vida en la composición).
- La diferencia es meramente conceptual entre una y otras. Es más existen varias interpretaciones.

# Clase de asociación



Es una Clase que surge de una multiplicidad de muchos a muchos, y fue incorporada en UML para dar soporte a este caso. Se sacan los atributos de las clases involucradas y se los incorpora a una clase a parte. Al igual que las anteriores hace referencia a una relación estructural. En el ejemplo son los objetos viaje y ruta

# Realización



Es una relación de contrato con otra clase. Se la utiliza para implementar una interfaz. En lenguajes como java o php utilizamos la palabra reservada “implements”

```
public class Viaje implements Registrable{...}
```

# Generalización



Es una relación de herencia. En nuestro ejemplo: “un Autobus es un tipo de Medio de transporte”.

Es entre una clase hijo y su clase padre. En la codificación podemos encontrar la palabra “extends” que hace referencia a esta relación.

Además podemos encontrar palabras claves tales como “this” y “super” ( Java ), hija(padre) en python o "self" y “parent” ( PHP ). Para darnos cuenta que existe una relación de este tipo involucrada.

```
public class Autobus extends MedioDeTransporte{...}
```



# Dependencia



Es una relación de uso, es decir que una clase utiliza a otra. Y si esta última se altera, la anterior se puede ver afectada.

En código se suelen traducir principalmente como las clases donde se hace la instanciación de un objeto. En nuestro ejemplo la clase Viaje realiza los “new” de los distintos objetos.

En este momento puede que te preguntes cómo puede hacer un new de una clase abstracta. No realiza los new de la clase abstracta, si no de sus hijas. Sería algo así como

```
MedioDeTransporte medio = new Autobus();
```

También se sostiene que este tipo de relación hace referencias, a los parámetros que se pasan en un método, bajo este concepto, en java, podría ser algo así como:

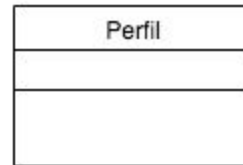
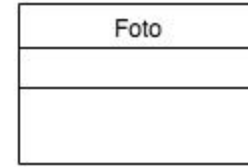
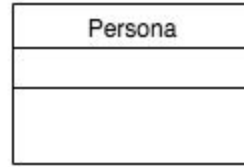
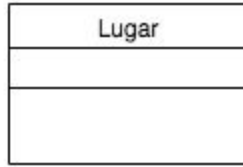
```
public void crearViaje(MedioDeTransporte medio){}
```

Por ultimo también se sostiene que podemos codificar esta relación realizando un “return” del tipo de dato en algún método.

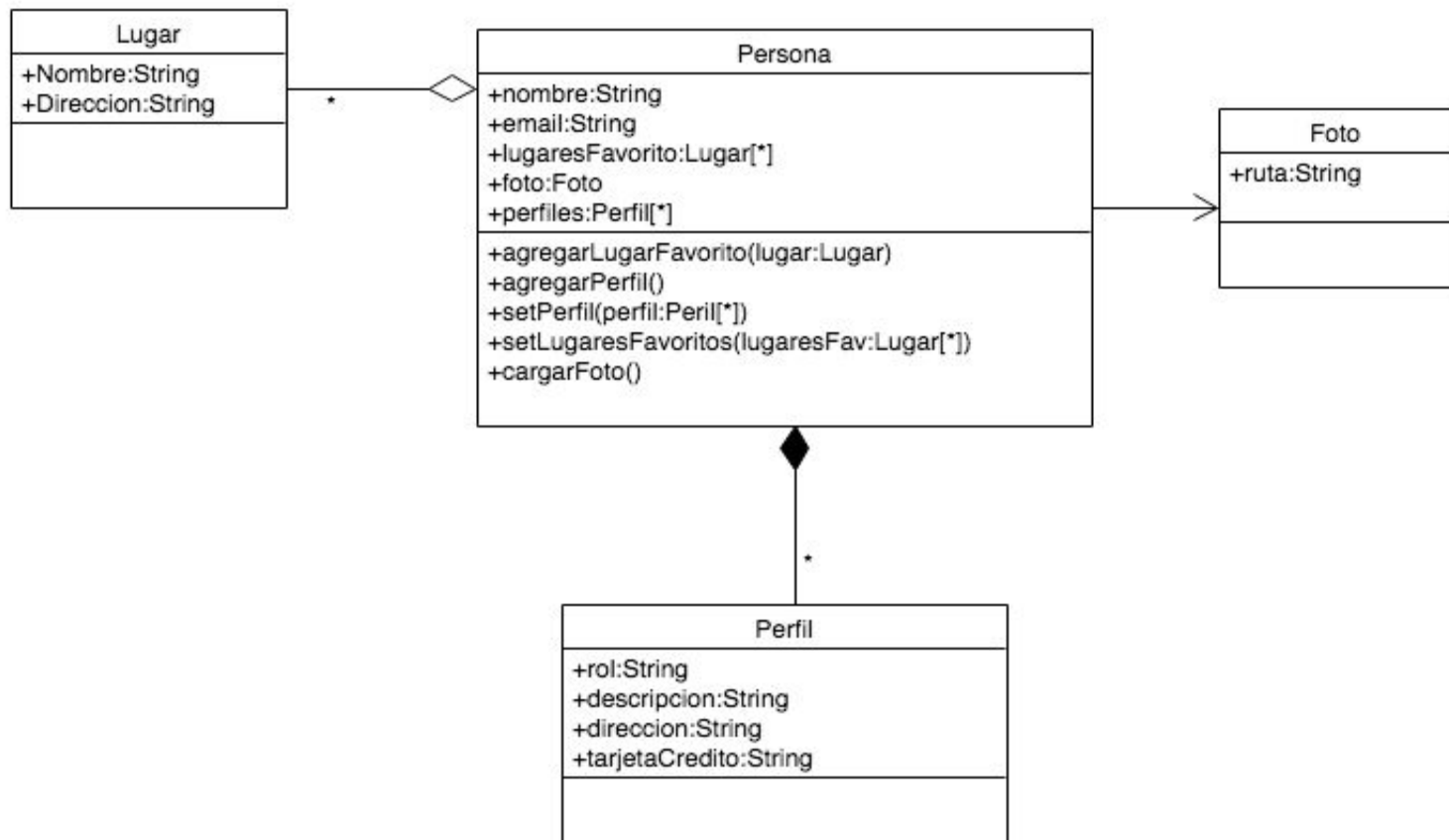
*Ángel González M.*

# Otro ejemplo completo

Partiendo de las siguientes clases, crea el diagrama completo de clases con sus dependencias.



Una persona tiene una foto  
Una persona tiene varios perfiles  
Una persona tiene varios lugares favoritos



# Asociación:



Es la relación que existe entre la clase Persona y la Clase Foto, es decir que que existe una variable de instancia (o atributo) en la clase Persona del tipo Foto. Solo un objeto, el cual se setea a partir del método setFoto(), como variable de instancia a la cual se le crea un setter.

Este puede ser instanciado (`new Foto()`) en cualquier otra parte del código.

# Agregación:



Es la relación que existe entre la clase Persona y la clase Lugar a través del atributo "lugaresFrecuentes", que es una colección (array, colection, vector, etc) de objetos Lugar.

Y que además, la clase Persona cuenta con un método "agregarLugaresFrecuentes(Lugar lugar)" Mediante el cual se agregan un solo objeto del tipo Lugar a la colección "lugaresFrecuentes". Estos objetos, que son pasados como parámetros a dicho método, son instanciados fuera de la clase Persona. No debe confundirse con el método setLugaresFrecuentes(Lugar[] lugaresFrecuentes), que tiene solo por ser un propiedad (getters y setters).

# Composición:



Es la relación que existe entre la clase Persona y la clase Perfil a través de la propiedad "perfiles" que es una colección del tipo Perfil.

Además, de ser una relación estructural, es meramente semántica, ya que lo que pretende es darnos la idea de la responsabilidad que tiene la clase Persona sobre el ciclo de vida del objeto de la clase Perfil. Es por ello que también en el ejemplo grafique una Dependencia (esta no se grafica generalmente, con la composición basta. Pero UML sirve para comunicarnos.)

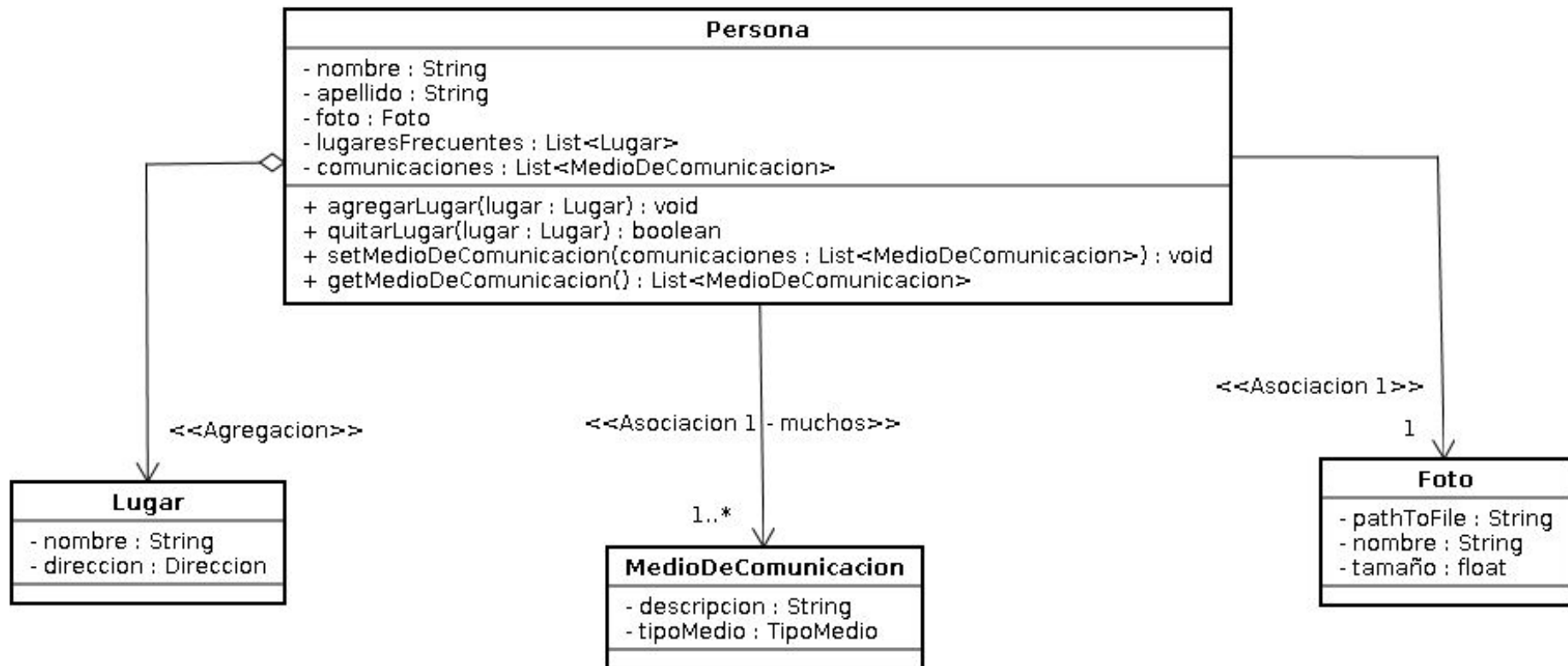
Es idéntica a la agregación pero con la diferencia que la clase Persona se hace responsable de la creación del objeto Perfil. Es decir que la instancia (`new Perfil()`) se crea dentro del método `agregarPerfil()`, y después de ello, dicho objeto se guarda en la colección "perfiles".

El modelo nos está diciendo, que en ninguna otra parte del código deberíamos escribir un `new Perfil()`. Ya que coloquialmente, el perfil (sea este psicologico, personal, etc) de una persona solo existe a partir de la persona, muerta la persona el perfil no sirve de nada, y solo a partir de la existencia de la misma se puede crear uno, por lo que este método de creación debe de ser un método de instancia.

*Ángel González M.*

# Otro ejemplo completo





¡La Agregación son siempre colecciones, o arrays! O algo que sirva de contenedor para "agregar" más de un objeto, aunque agreguemos uno solo. (si no sería settear y no agregar, add)

La Agregación cuenta con dos métodos: uno para "agregar" un solo objeto a la lista, y el otro para quitarlo de la misma.

La agregación puede, como no, tener los metodos setter y getter, mientras que la Asociación siempre los tiene, que ponen y obtienen una variable de referencia del mismo tipo de la variable de instancia o de clase, en este caso List.

```
import java.util.List;

public class Persona {

    private String nombre;
    private String apellido;

    private Foto foto;
    private List lugaresFrecuentes;
    private List comunicaciones;

    public String getNombre() {return nombre;}
    public void setNombre(String nombre) {this.nombre = nombre;}
    public String getApellido() {return apellido;}
    public void setApellido(String apellido) {this.apellido = apellido;}

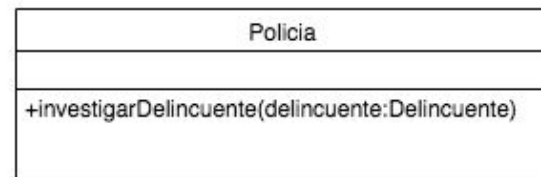
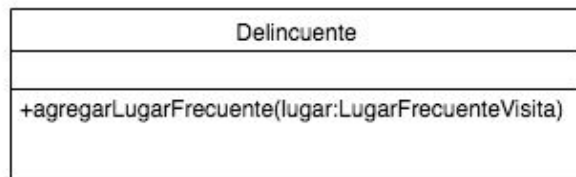
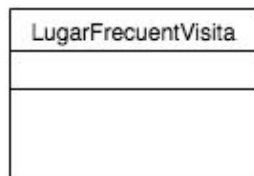
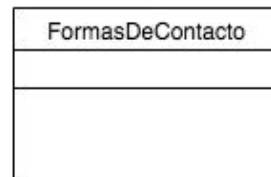
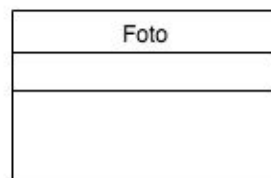
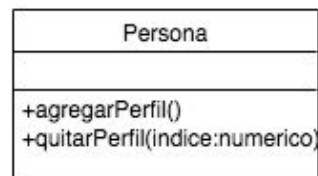
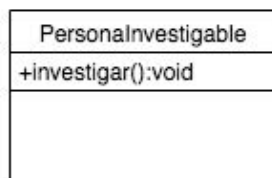
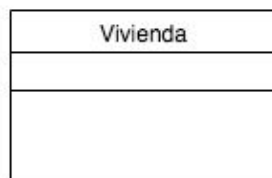
    //Asociacion Foto
    public Foto getFoto() {return foto;}
    public void setFoto(Foto foto) {this.foto = foto;}

    //public List getLugaresFrecuentes() {return lugaresFrecuentes;}
    //public void setLugaresFrecuentes(List lugaresFrecuentes) {this.lugaresFrecuentes = lugaresFrecuentes;}
```

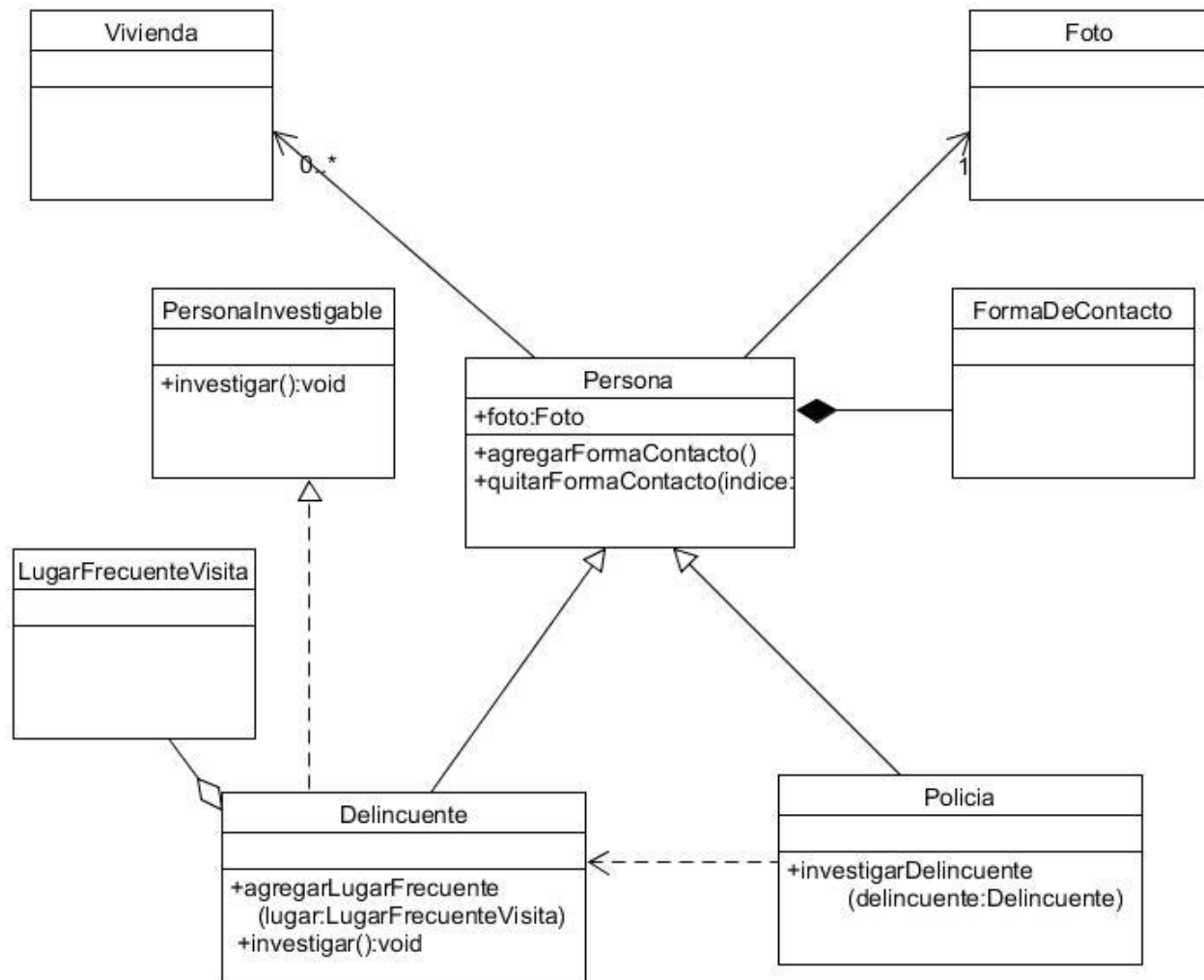
# Otro ejemplo completo

Partiendo de las siguientes clases, crea el diagrama completo de clases con sus dependencias.

Una persona puede tener muchas viviendas de residencia  
Una persona solo tiene una foto de perfil  
Una persona puede tener muchas formas de contacto(email, carta,...)  
Los delincuentes poseen una lista de lugares frecuentes que visitan  
La policia puede detener delincuentes



# Solución

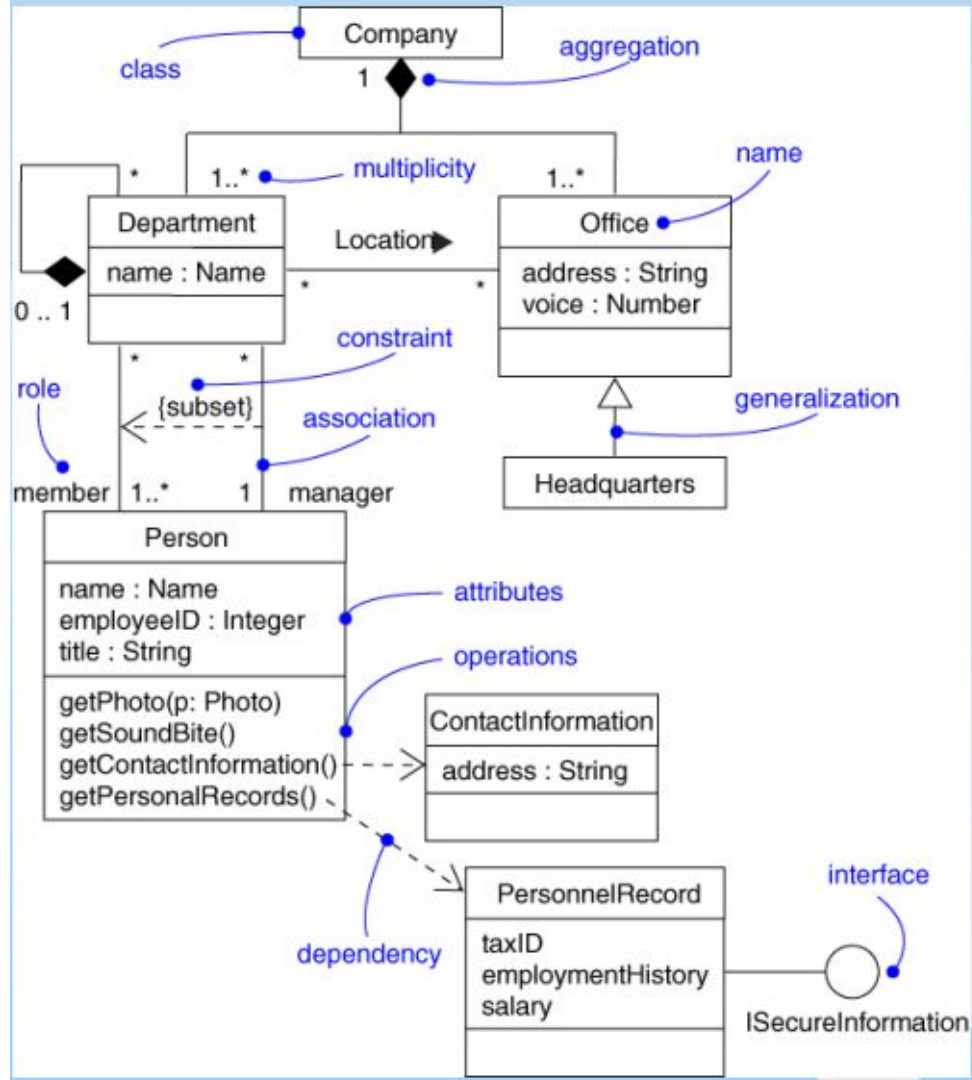


# Ejercicio

Convierte este diagrama en un programa de java

<< GUI >> Formulario de Reservas	
+ título : Titulo	
+ prestatario: Informacion_prestatario	
+ botonBuscarTitulo_Pulsado ( ) + botonBuscarPrestatario_Pulsado( ) + botonOk_Pulsado ( ) + botonCancelar_Pulsado ( ) + títuloResultado ( ) + prestatarioResultado ( ) - comprobarEstado ( ) + FormularioDeReservas ( ) # botonEliminarTitulo ( )	

# Ejercicio





# Ejercicio

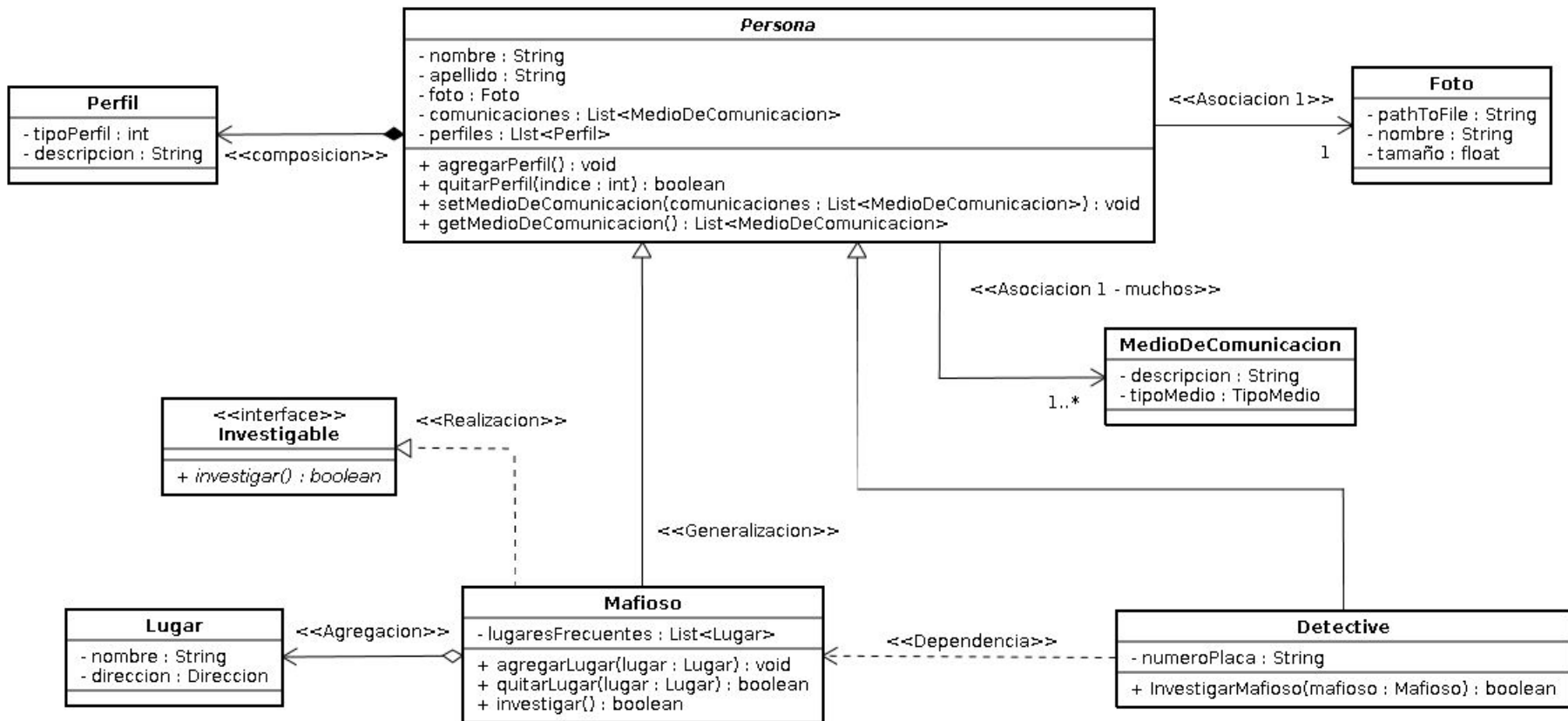
Crea la relaciones correspondiente partiendo de las siguientes clases.

Ejer1:

Camioneta, Autobús, Vehículo

Ejer2:

Empresa, cliente



# Ejercicio ¿Asociación, agregación o composición?

```
class Dinero{
    public float total;
    public String tipoMoneda;
}

public class Cuenta {
    private Dinero balance;

    public void ingresar(Dinero cantidad){
        this.balance.total += cantidad.total;
    }
    public void retirar(Dinero cantidad){
        this.balance.total -= cantidad.total;
    }
    public Dinero getSaldo() {
        return balance;
    }

    public void setBalance(Dinero balance) {
        this.balance = balance;
    }
}
```

# Ejercicio ¿Asociación, agregación o composición?

```
import java.util.Date;
```

```
public class Cuenta{  
    private float balance;  
    private float limite;  
    private Date fechaApertura;  
    public Cliente titular;  
  
    public void ingresar(float cantidad){  
        balance += cantidad;  
    }  
    public void retirar(float cantidad){  
        balance -= cantidad;  
    }  
}
```

```
class Cliente {  
    public String nombre;  
    public String apellidos;  
    public String direccion;  
    public String nif;  
    public Cuenta[] cuenta;  
}
```