UML Sí o Sí

Bienvenidos

75.07 Algoritmos y Programación III

Facultad de Ingeniería
Universidad de Buenos Aires

Pablo Rodriguez Massuh

¿ QUÉ NECESITAMOS SABER ?

1. ¿Qué es UML? ¿Para qué sirve?
2. Diagramas de clases
3. Diagramas de secuencia
4. ¿Hay otros diagramas UML?

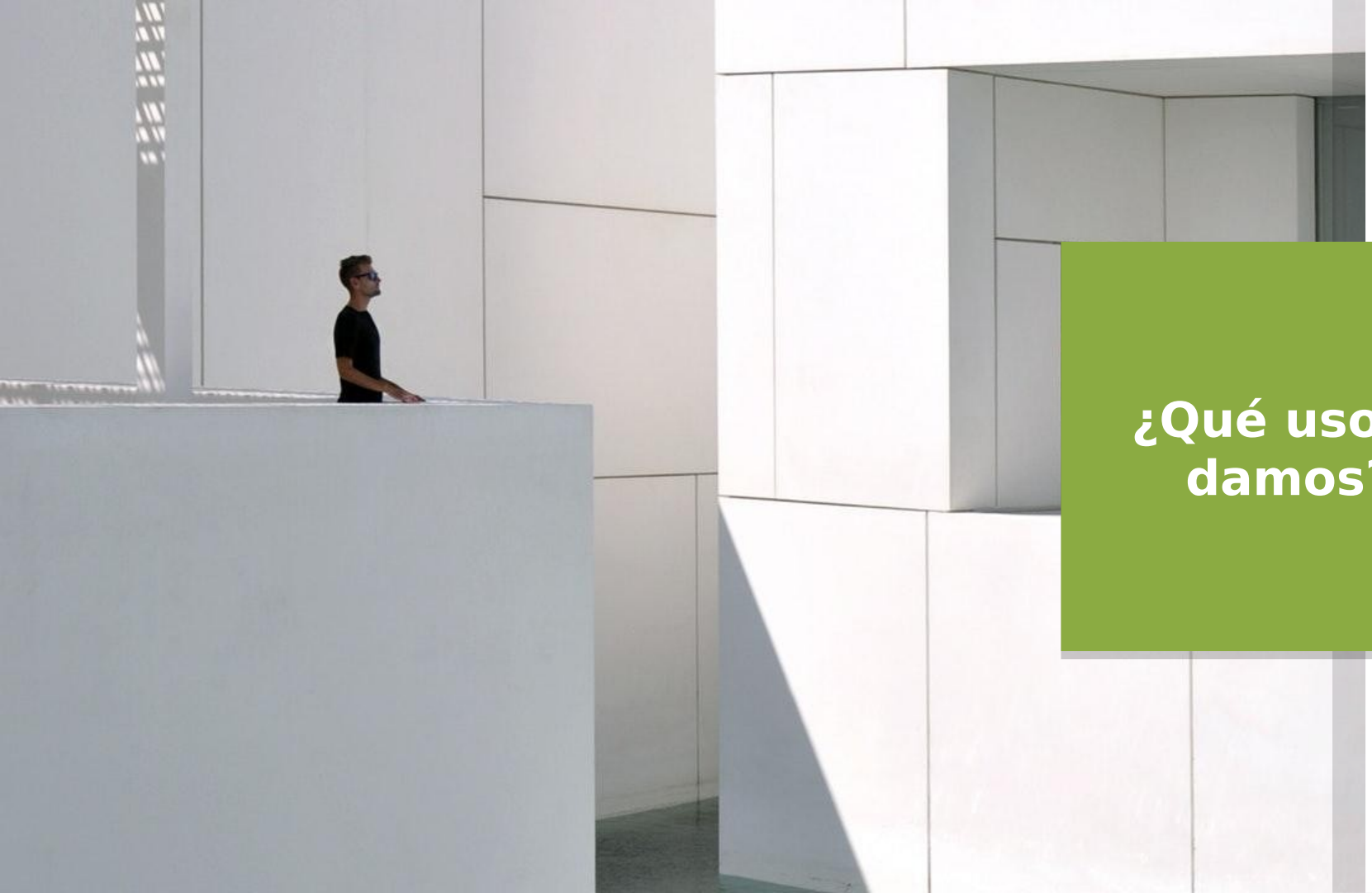
1. UML

¿Qué es y para qué sirve?

UML es un lenguaje para la visualización, especificación y documentación de software, por lo que resulta independiente del método que se utilice para el

O sea ...

- No es un *método*, sino una **notación**.
- No especifica un proceso.
- Describe el resultado de alguna actividad de desarrollo mediante una serie de diagramas.
- La idea es centrarse más en los objetos que en los procesos o algoritmos.



¿Qué uso le damos?

COMUNICAR

De una forma clara, simple y uniforme para toda la comunidad de desarrollo.

COMUNICAR… ¿QUÉ? ¿EN QUÉ CONTEXTOS?

Diseños

Cuando haya que transmitir aspectos del diseño de una aplicación a un equipo de trabajo, para que lo materialicen en el producto.

Acuerdos

Cuando 2 o más personas necesiten ponerse de acuerdo sobre un diseño, o desean discutir alternativas, y esperan visualizarlo mejor en forma gráfica.

Documentación

Cuando se necesite dejar documentación de diseño de un proyecto ya terminado.

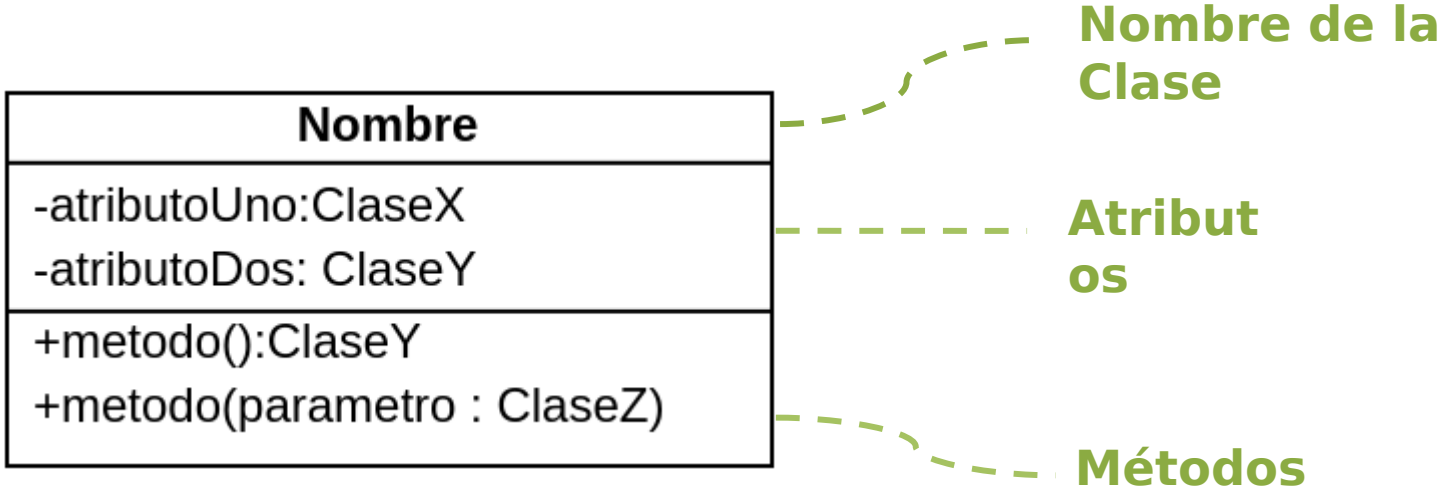
2. DIAGRAMA DE CLASES

“

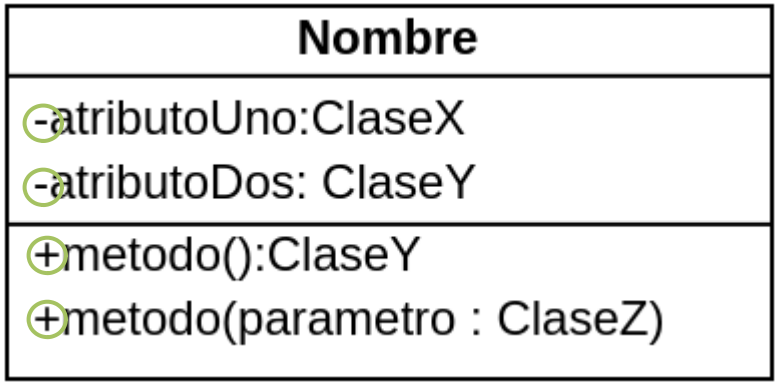
Es un modelo estático del sistema a construir o de una parte del mismo. En él se muestran clases y relaciones entre las

¿ Cómo está compuesto ?

Tiene 3 partes



Visibilidad I



Puede ser:

- **+** Pública
- **-** Privada
- **#** Protegida
- **~** Paquete

Visibilidad II

Pública

El atributo o método puede ser accedido desde afuera del objeto.

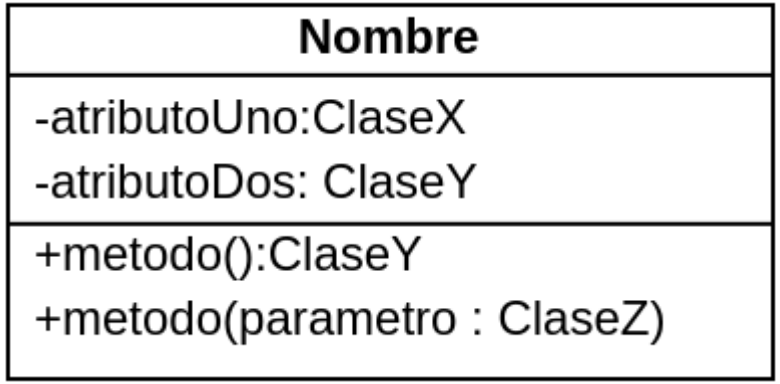
Privada

El atributo o método solo puede ser accedido desde el mismo objeto. No es posible acceder tanto desde afuera como de las clases hijas.

Protegida

El atributo o método solo puede ser accedido desde el mismo objeto **y** desde sus clases hijas.

Atributos



Nomenclatura:

```
visibilidad identificador : Tipo
```

Métodos

Nombre
-atributoUno:ClaseX -atributoDos: ClaseY
+metodo():ClaseY +metodo(parametro : ClaseZ)

Si el método devuelve un objeto, se identifica de qué tipo (clase) es ese objeto que devuelve.

Nomenclatura:

```
visibilidad identificador() : Tipo
visibilidad identificador(parametro:Tipo) :
Tipo
visibilidad identificador(p1:Tipo, p2:Tipo) :
Tipo
```

Pará, pará, pará, pará Turquito ... ¿Y con Smalltalk cómo hago?

Métodos Smalltalk

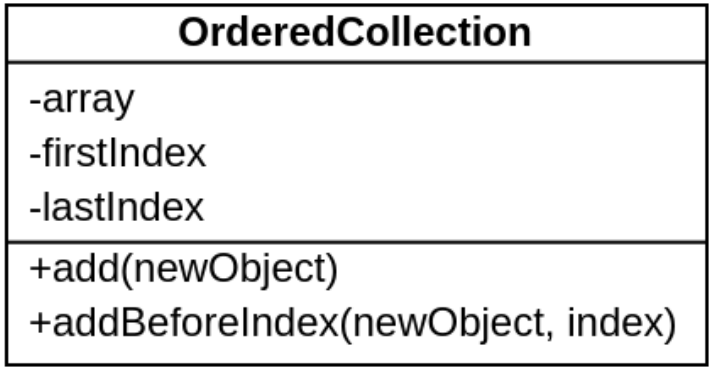
```
|collection|
collection := OrderedCollection new.
collection add: 'Massuh'.
collection add: 'Pablo' beforeIndex: 1.
```

¿Cómo quedaría la representación UML del mensaje:

```
add: newObject beforeIndex: index
```

?

Métodos Smalltalk



Dada la limitación natural propia de la notación UML *(que utiliza la convención del lenguaje* **C** *para los métodos)* lo que se busca es poder ***transmitir*** de la mejor manera la firma del método de modo tal que se pueda realizar un rápido paralelismo entre lo que muestra el diagrama y el código.

ÁMBITOS I

Miembro de Instancia

Su ámbito es *una* instancia específica.

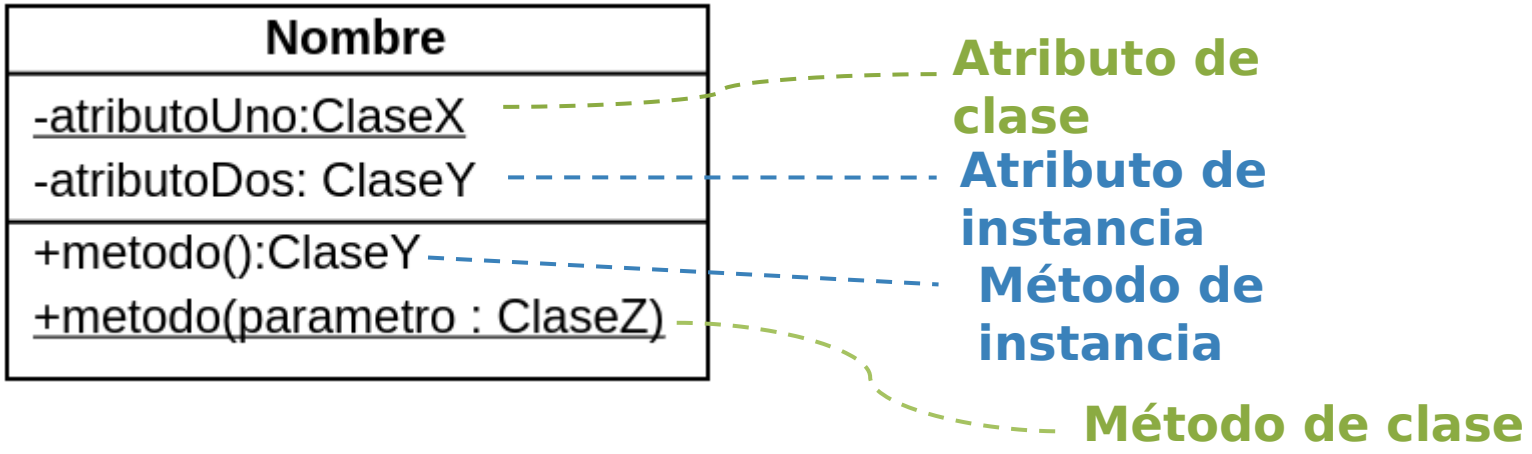
- Los valores de los atributos pueden variar entre instancias.
- La invocación de métodos puede afectar al estado de las instancias *(es decir, cambiar el valor de sus atributos)*

Miembro de clase o estático

Su ámbito es la propia clase.

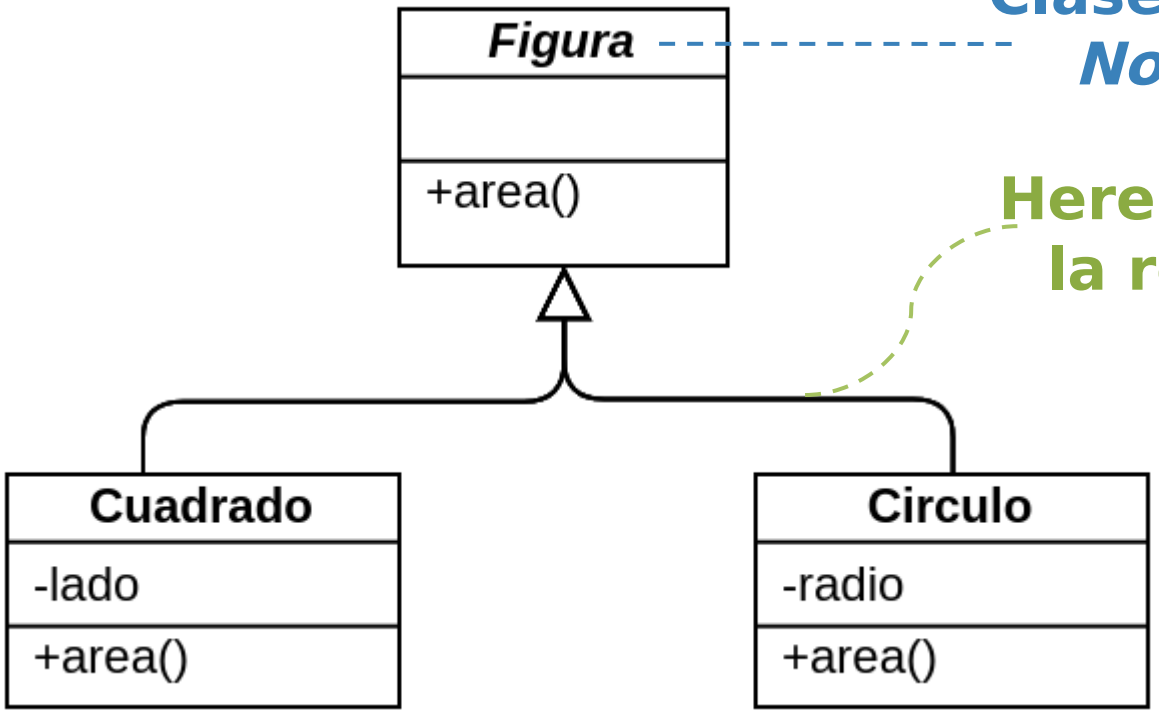
- Los valores de los atributos son los mismos en todas las instancias.
- La invocación de métodos no afecta al estado de las instancias

ÁMBITOS II


Nombre
-atributoUno:ClaseX
-atributoDos: ClaseY
+metodo():ClaseY
+metodo(parametro : ClaseZ)
Atributo de clase
Atributo de instancia
Método de instancia
Método de clase

Relaciones entre clases

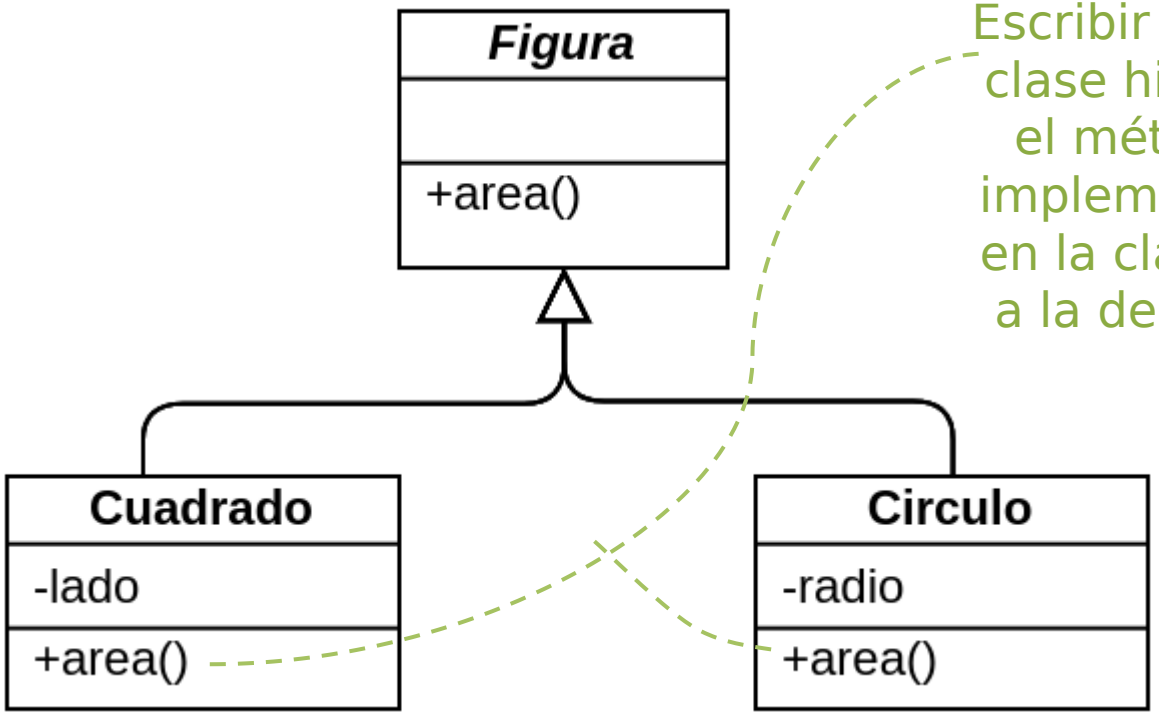
Herencia I



Clase abstracta: *Nombre en itálica*

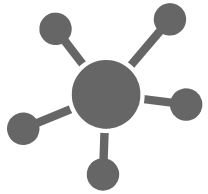
Herencia: cumple la relación “es un”

Herencia II



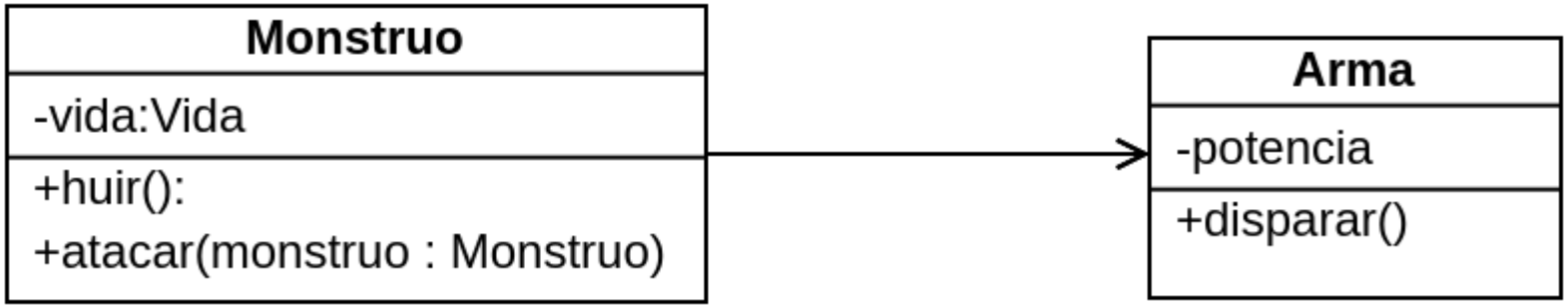
Escribir el método en la clase hija significa que el método tiene una implementación propia en la clase hija *distinta* a la de la clase Madre

ASOCIACIÓN



En POO un objeto puede estar relacionado con otro para usar los servicios *(métodos)* proporcionados por ese objeto. Esta relación entre dos objetos se conoce como asociación y es representada por una flecha en

Asociación I



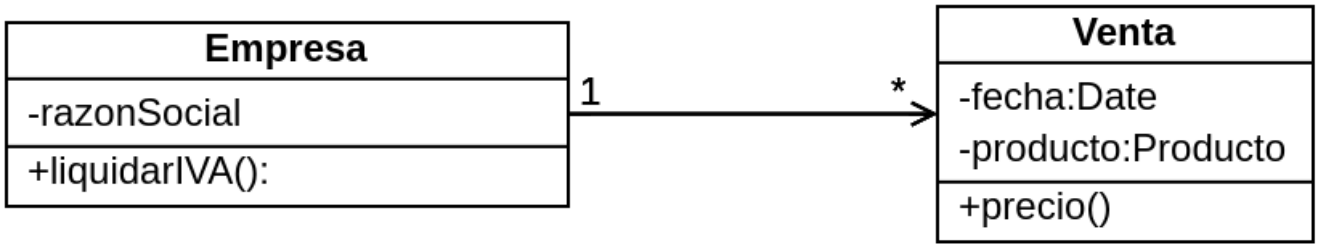
Se lee, interpreta como: **“Un Monstruo tiene / usa un Arma”** .

Importante: Nótese como dentro de Monstruo **no se encuentra**:

```
-arma:Arma
```

Ésto es así ya que la flecha misma de asociación me está indicando que un monstruo tiene un arma, con lo cual no

Asociación II : Colecciones

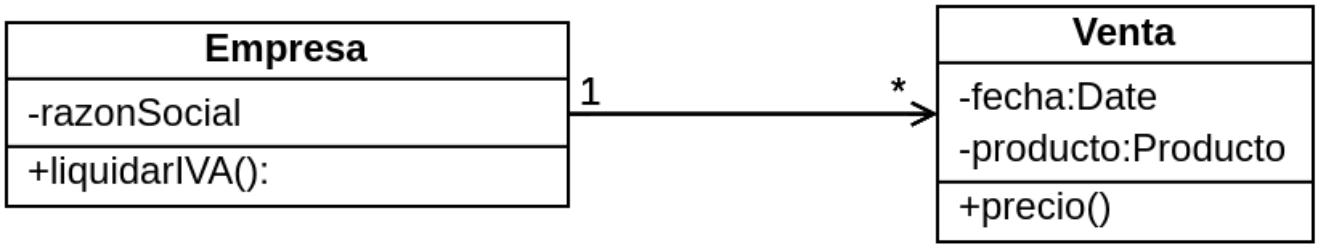


Lo mismo sucede con las colecciones. Se lee: **“Una Empresa tiene muchas Ventas”**. No importa como es la implementación, es decir no importa si es:

`-ventas:OrderedCollection` **ó** `-ventas:LinkedList` **ó** `-ventas: ...`

Lo que importa es ***comunicar*** que una instancia de **Empresa** contiene
muchas de **Venta**. Y como está la flecha es porque claramente Empresa tiene guardadas las Ventas de alguna forma que no interesa saber. **Recordemos**: tienen que ser simples, de fácil

Asociación III

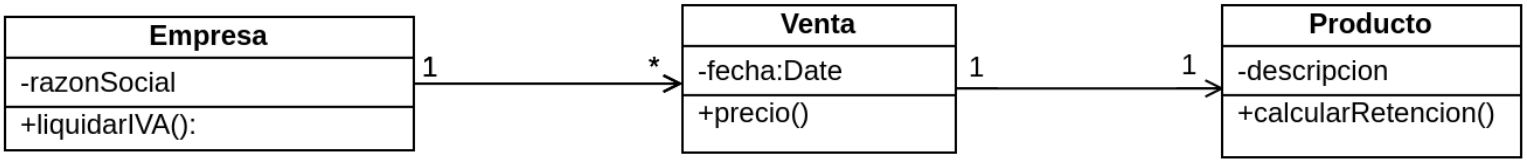


Nótese que en este diagrama **sí** observamos:

```
-producto:Producto
```

Esto se debe a que como en este diagrama ***no*** se encuentra la clase **Producto** y queremos comunicar que **Venta** tiene una relación con **Producto** entonces en esta situación lo agregamos como atributo en la clase **Venta** para evidenciarlo.

Asociación III



Si incluimos **Producto** en el diagrama entonces ya **no** está presente:

```
-producto:Producto
```

Porque como aclaramos, eso sería duplicar la información.

Multiplicidad

Definición

La multiplicidad de una asociación indica *cuántos* objetos de cada tipo intervienen en la relación.
O sea, el número de instancias de una clase que se relacionan con **una** instancia de otra clase.

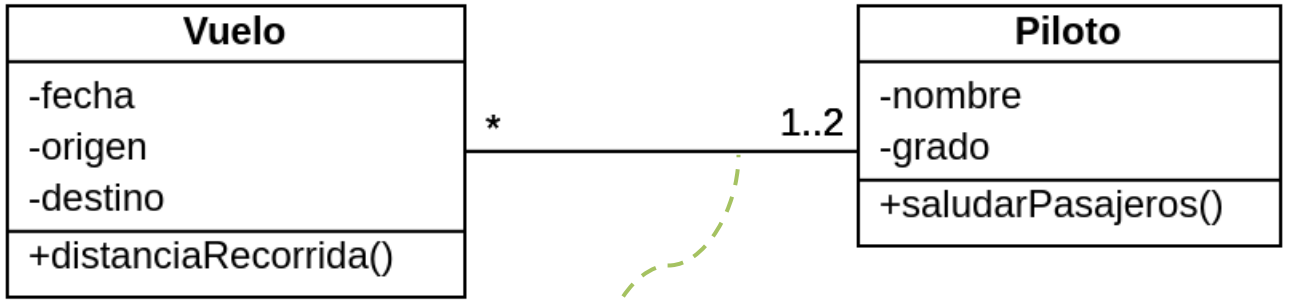
Multiplicidad I

- Cada asociación tiene **2** multiplicidades
  - *Una para cada extremo de la relación*
- Para especificar la multiplicidad de una asociación se debe indicar la multiplicidad mínima y la multiplicidad máxima

Multiplicidad II : Notación

Multiplicidad	*Significado*
1	Uno y sólo uno
0..1	Cero o uno
n..m	Desde **n** hasta **m**
*	Cero o varios
0..*	Cero o varios
1..*	Uno o varios (por lo menos uno)

Ejemplo de multiplicidad



He aquí una asociación bidireccional, es decir que se puede recorrer en ambos sentidos. Si tenemos *una* instancia de **Vuelo** vemos que la misma tendrá *una* o *dos* instancia/s de **Piloto**. Luego en el sentido contrario, una instancia de **Piloto** puede tener ninguna o muchas instancias de **Vuelo**.

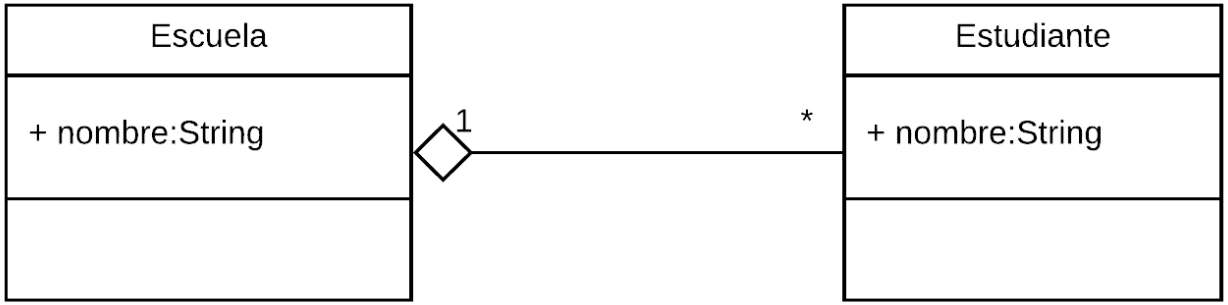
Agregación / Composición

Agregación y Composición

Son casos *particulares* de asociaciones:

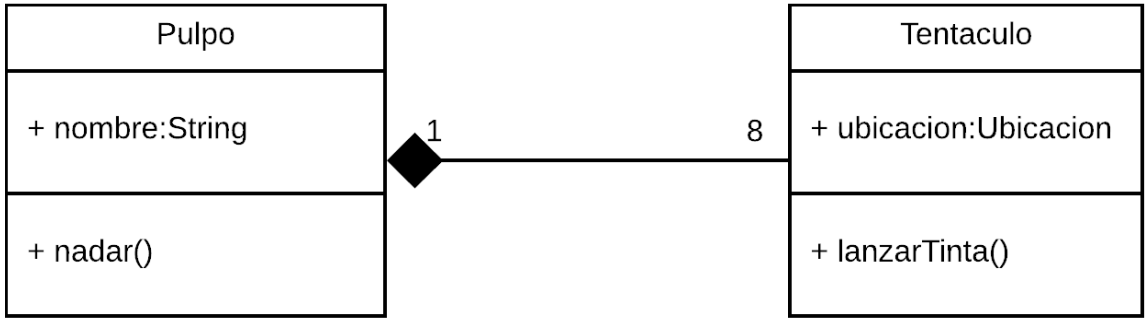
- Indican una relación entre un todo y sus partes.
- Gráficamente se muestran como asociaciones pero con un rombo en uno de los extremos.

Agregación



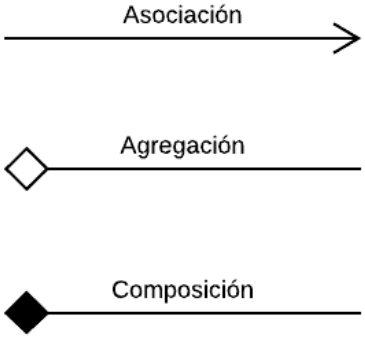
Un ejemplo de **agregación**: Los estudiantes de una Escuela. Cuando la escuela cierra, los estudiantes siguen existiendo (incluso pueden asistir a otra escuela).

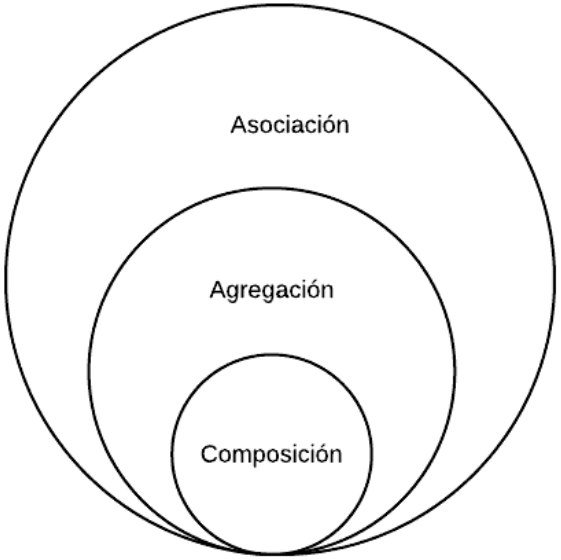
Composición



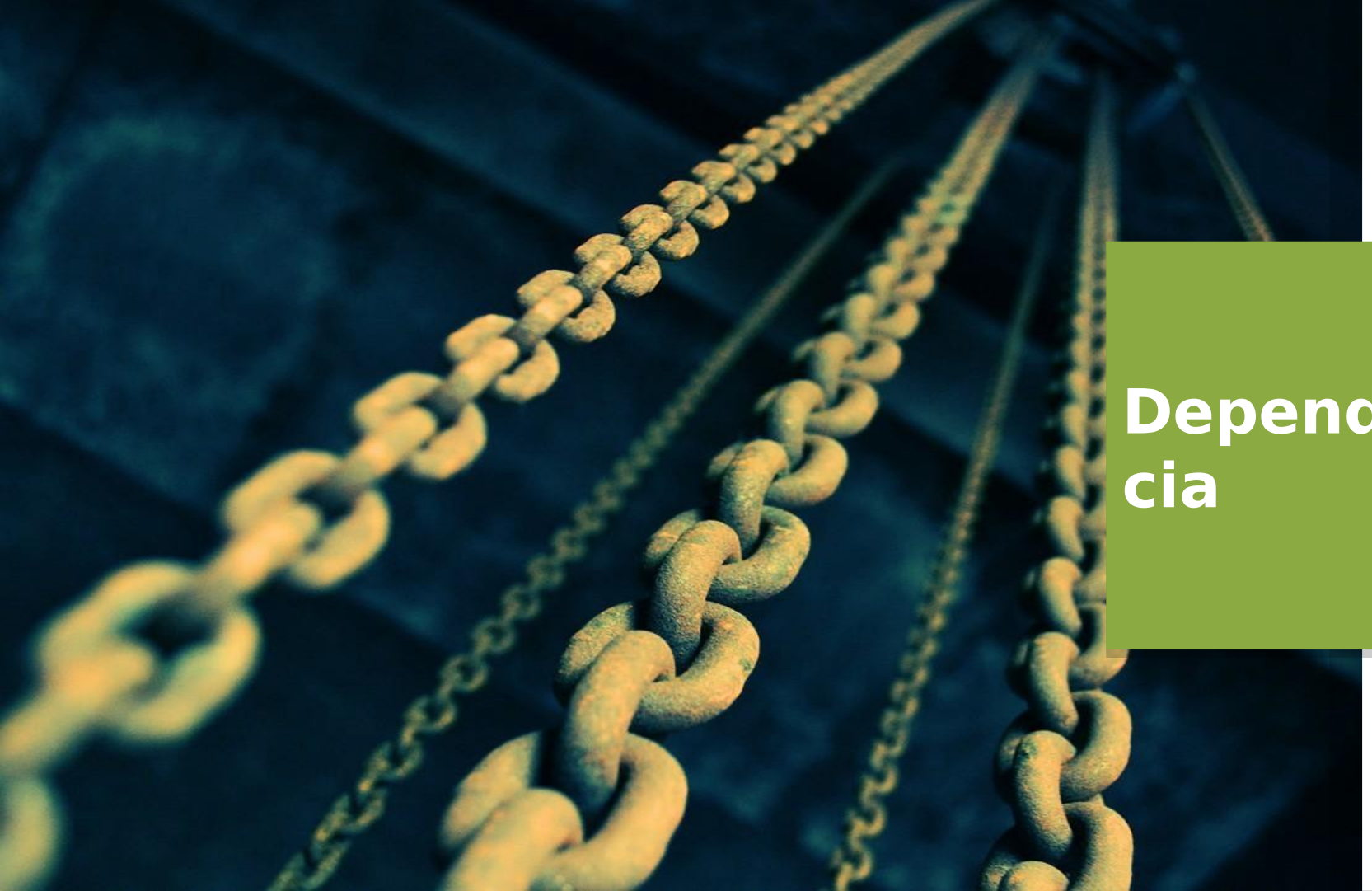
Un ejemplo de **composición**: Un pulpo y sus tentáculos. Los tentáculos **no** funcionan por sí sólos si el pulpo es destruido. La composición es más fuerte que la agregación. Una asociación es referida como *composición* cuando un objeto es el ***dueño*** de otro. En cambio en una *agregación* un objeto usa al otro.

Asociación - Agregación - Composición





Las 3 denotan una relación entre objetos y sólo difieren en su fuerza. La **composición** representa la forma **más fuerte** y la *asociación* la *más general.*



Dependencia

Dependencia

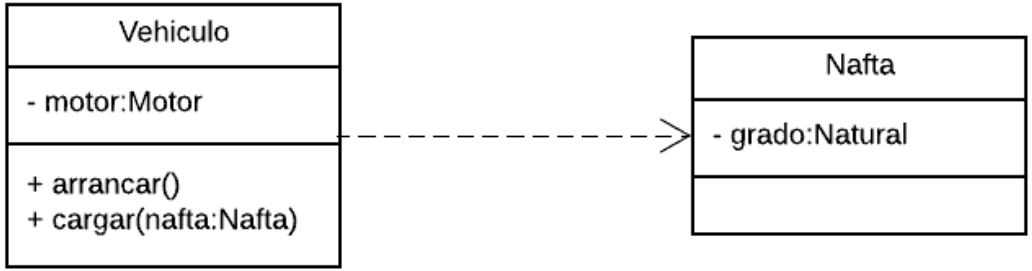
Implica también una relación entre 2 objetos. La misma *es más débil que una asociación*.

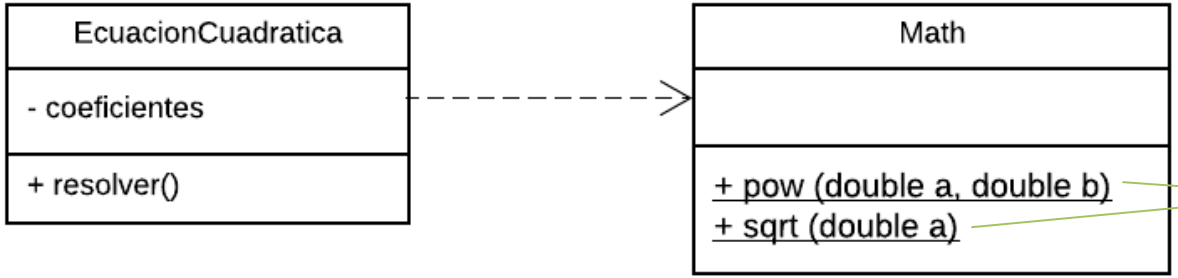
A diferencia de una asociación, en el caso de una dependencia un objeto **no tiene al otro como atributo**, sino que lo utiliza ya sea porque lo recibe como parámetro en un método o porque lo crea (lo instancia) y lo devuelve como respuesta de uno de sus métodos.

También puede darse el caso que un objeto use al otro mediante el llamado a métodos estáticos.

La flecha utilizada para una dependencia es como la de asociación, pero punteada.

Dependencia: ejemplos





Métodos de clase (estáticos)

Interfaces

Interfaces

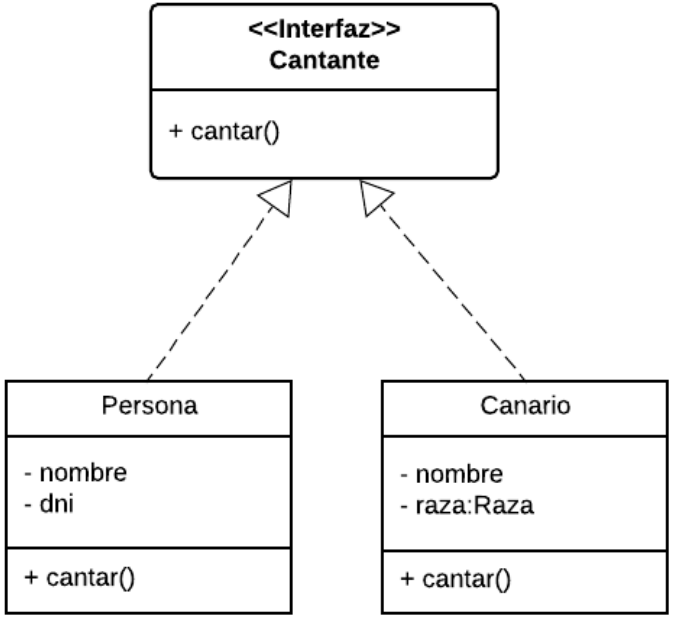
Una interfaz es un tipo *abstracto* sin estado que se utiliza para especificar un comportamiento.

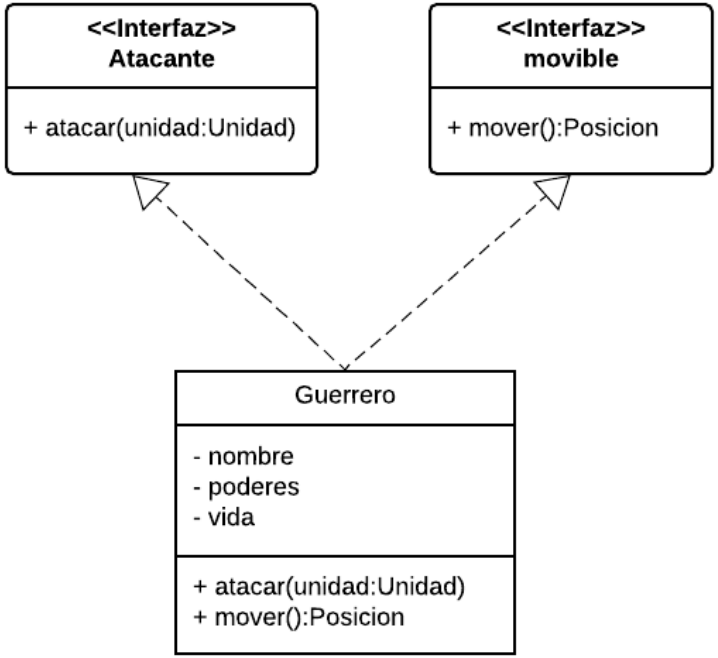
Básicamente es una colección de declaraciones de métodos *sin implementar* (sólo posee la firma de los mismos).

Cuando una clase dice que implementa una interfaz en **UML** se denota con una flecha punteada que posee la misma punta que la flecha de herencia.

Una misma clase puede implementar **múltiples** interfaces.

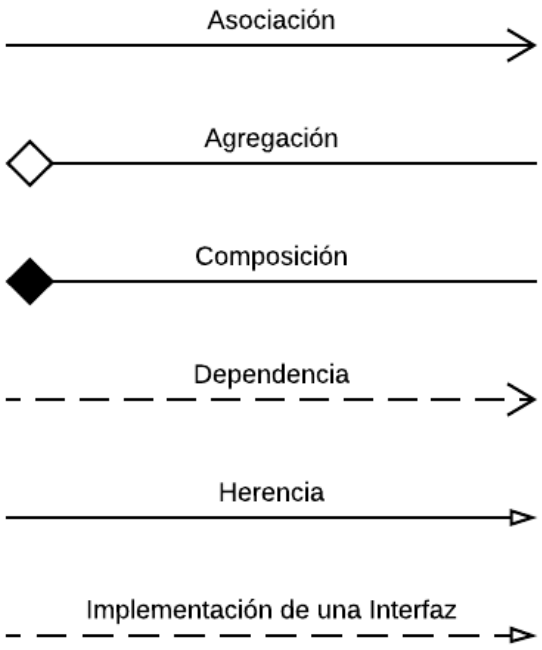
Interfaces: ejemplos





¿ Cuantas “ *flechas* “ conocimos ?

Son 6 relaciones distintas



Cada una representa una situación específica.

Por eso es tan importante conocer el significado de cada una de ellas.

3. DIAGRAMA DE SECUENCIAS

“

Un diagrama de secuencia muestra la interacción de un *conjunto* de objetos en una aplicación a través del tiempo.

mmm...

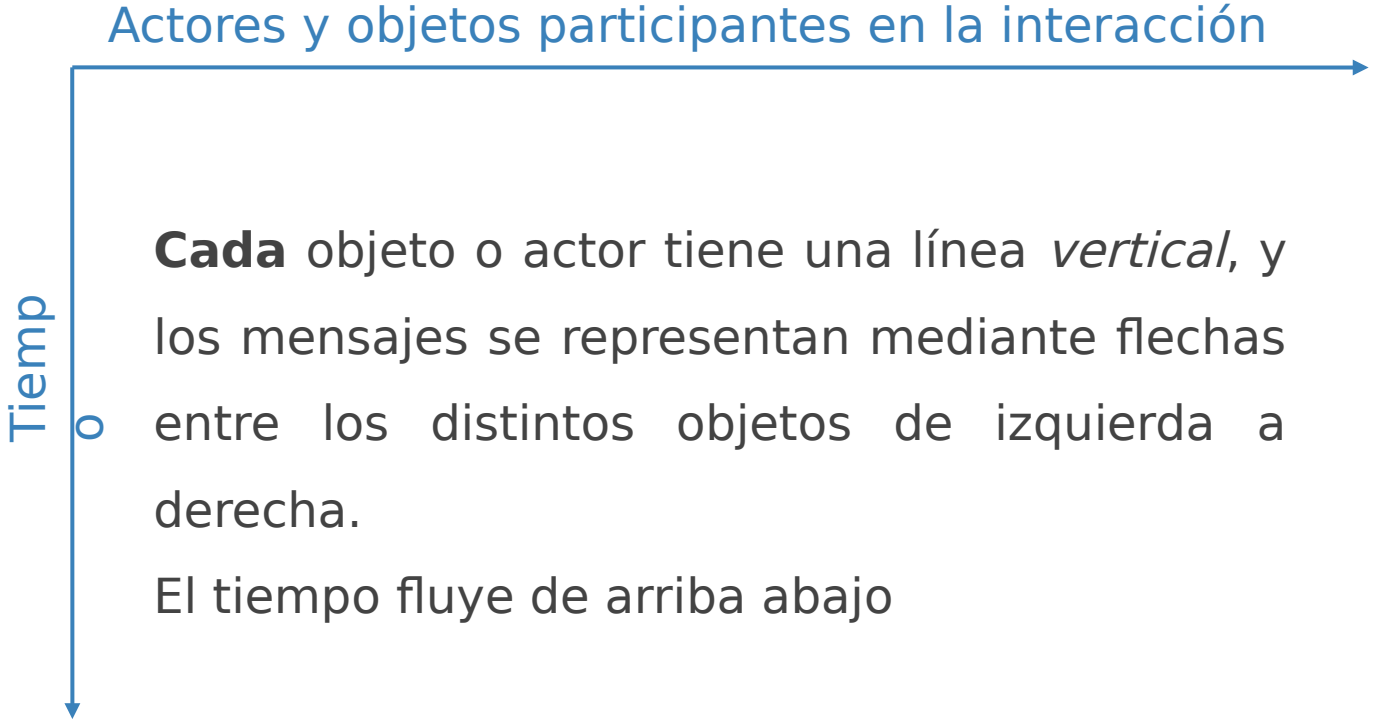
“

O sea, es un diagrama dinámico que muestra cómo los objetos se envían mensajes entre sí a lo largo del tiempo para

Elementos disponibles en un diagrama de secuencia

- Actores
- Objetos
- Clases
- Mensajes
- Mensajes de retorno
- Activaciones
- Líneas de vida
- Ciclos
- Creación de objetos
- Destrucción de objetos

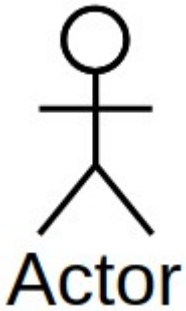
Estructura



Cada objeto o actor tiene una línea *vertical*, y los mensajes se representan mediante flechas entre los distintos objetos de izquierda a derecha.

El tiempo fluye de arriba abajo

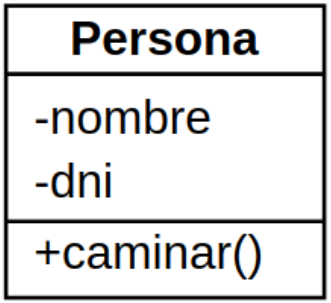
¿Qué es un Actor?



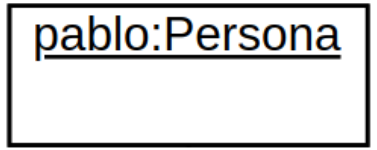
- Representa un tipo de rol que interactúa con el sistema y sus objetos.
- Siempre está fuera del alcance del sistema *que pretendemos* modelar.
- Usamos actores para representar varios roles, incluidos los usuarios humanos y otros sujetos externos.
- Representamos a un actor utilizando una notación de persona palito.
- Podemos tener múltiples actores en un diagrama de secuencia.

Instancias de objetos

Diagrama de clases



Representación del objeto “pablo”.
Es una instancia de la clase Persona



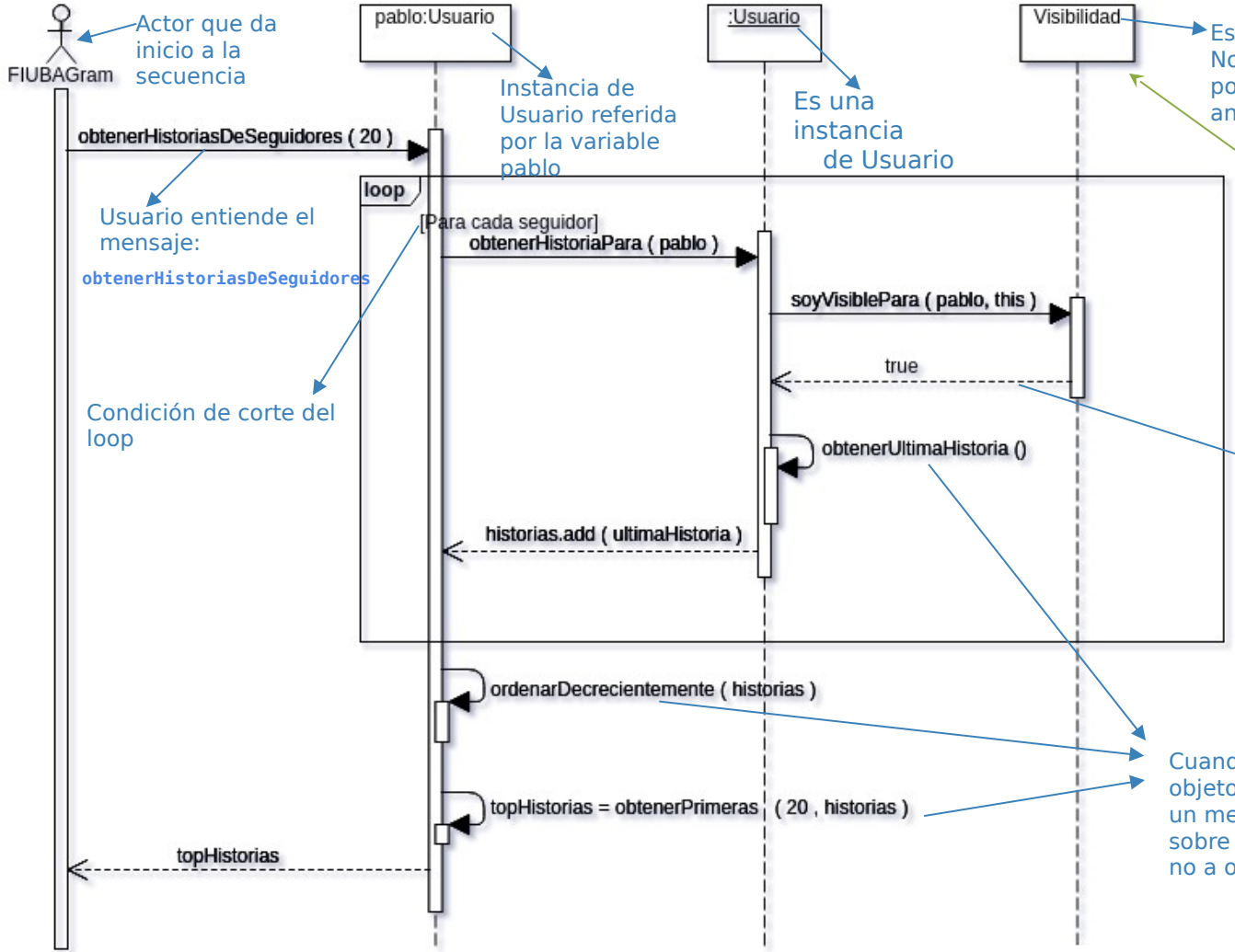

```
pablo = new Persona();
pablo := Persona new.
```

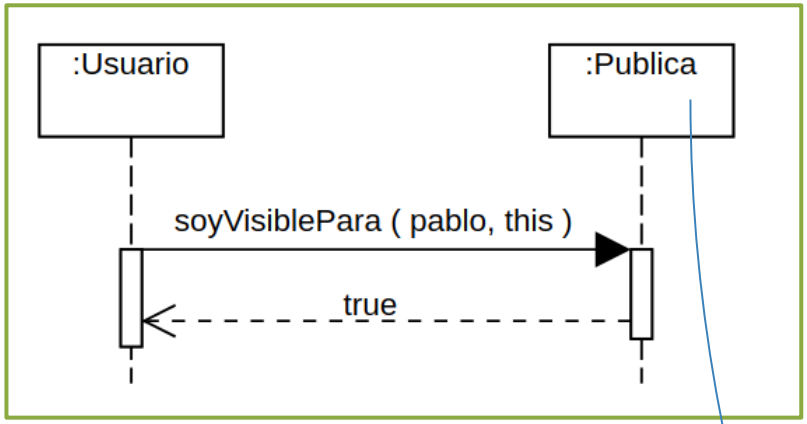
Datos a tener en cuenta

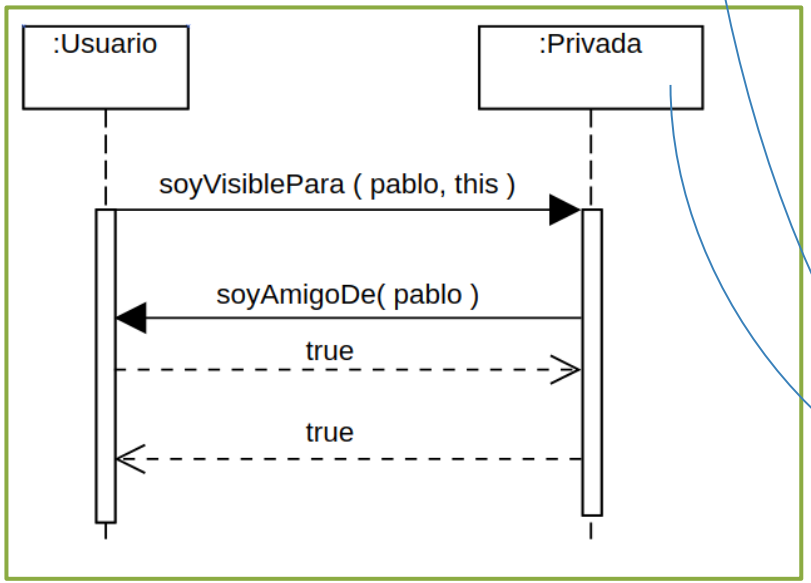
- Al hacer un diagrama de secuencias es **importantísimo** señalar a qué escenario particular se refiere. Es decir indicar qué problemática *puntual* se está resolviendo en la secuencia. Se lo suele indicar poniéndole un **título** al diagrama que lo explique.
- ¿Cómo haría un “ **if** ” en un diagrama de secuencias? Si bien existe una notación *específica* para modelar dicha situación, desde la cátedra pregonamos que se realicen **2** diagramas de secuencias. En uno se mostrará la secuencia de objetos y llamados de mensajes para el caso “*true*” y en el otro diagrama se mostrarán las secuencias para el caso “*false*”. Naturalmente cada uno de éstos 2 diagramas debe ser acompañado con su respectivo título indicativo acerca del escenario

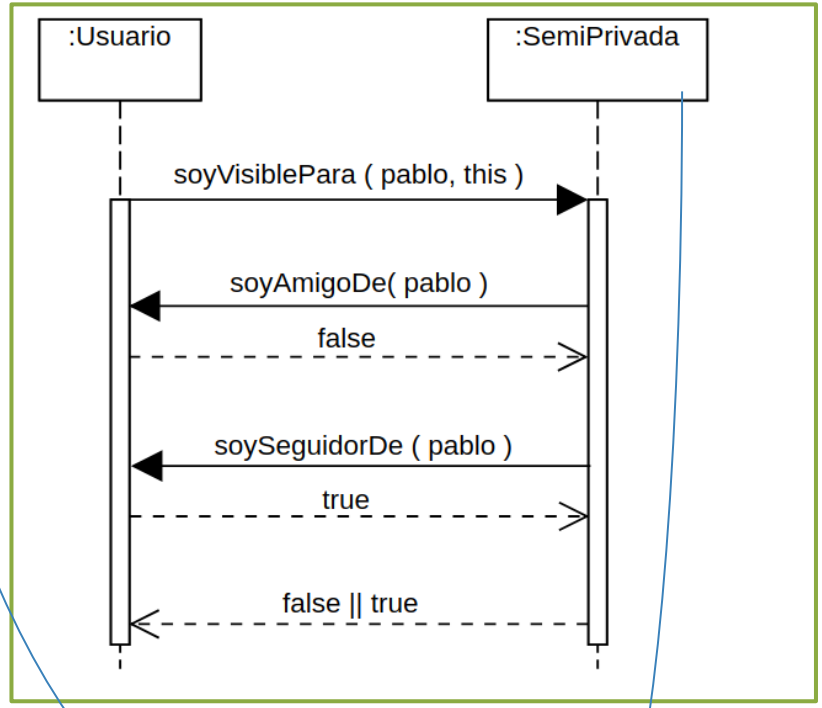
tomados en exámenes

¿Qué títulos les pondrían?

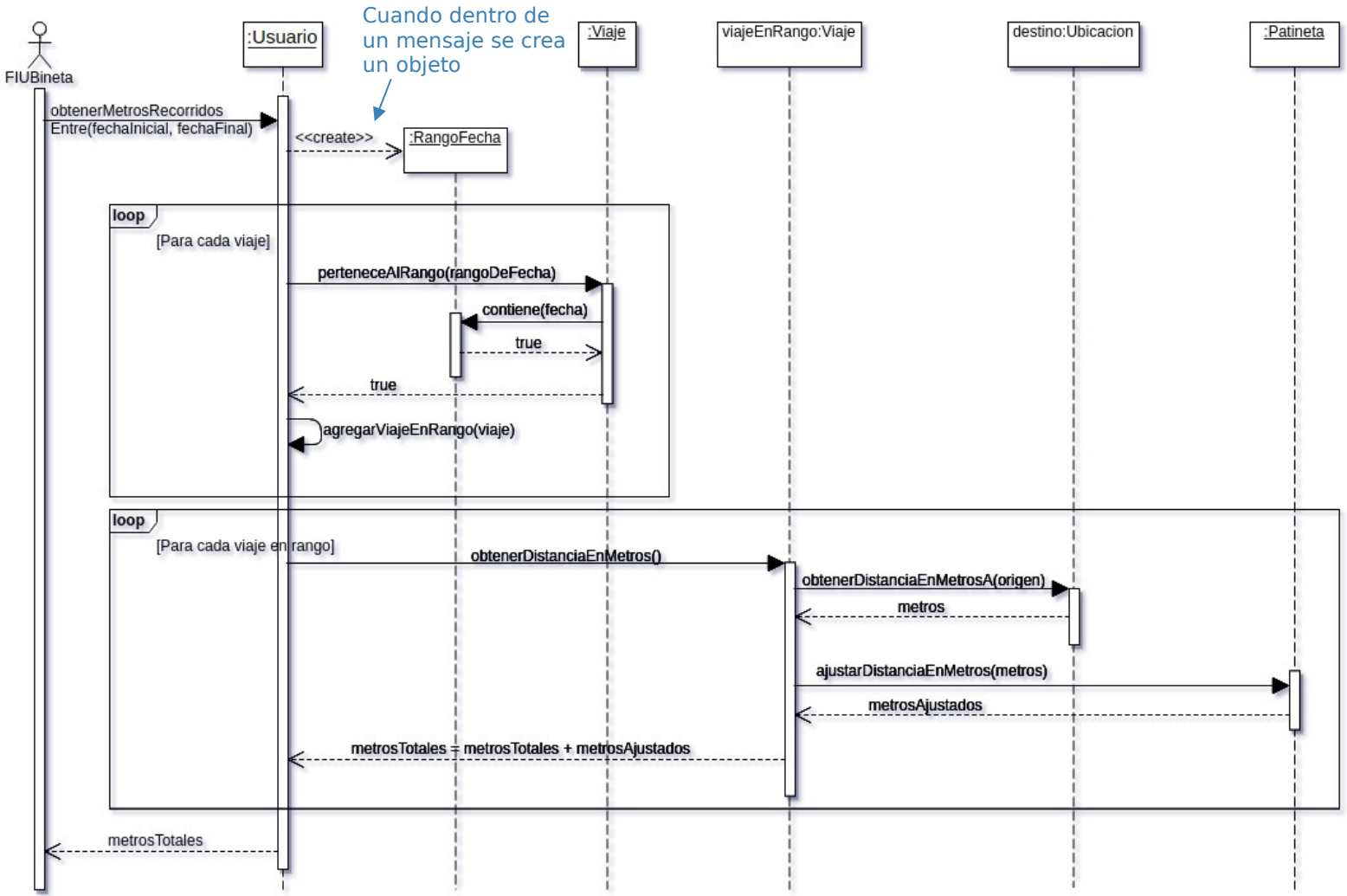

FIUBAGram
Actor que da inicio a la secuencia
pablo:Usuario
Instancia de Usuario referida por la variable pablo
:Usuario
Es una instancia de Usuario
Visibilidad
Es una clase. OJO. Nos damos cuenta porque no la antecede los :
obtenerHistoriasDeSeguidores (20)
Usuario entiende el mensaje:
obtenerHistoriasDeSeguidores
loop
[Para cada seguidor]
obtenerHistoriaPara (pablo)
soyVisiblePara (pablo, this)
true
Condición de corte del loop
obtenerUltimaHistoria ()
historias.add (ultimaHistoria)
ordenarDecrecientemente (historias)
topHistorias = obtenerPrimeras (20 , historias)
topHistorias
Así son los return: con una línea punteada, a diferencia de la línea sólida y punta completa que implica llamado a un mensaje de un objeto.
Cuando un objeto llama a un mensaje sobre sí mismo y no a otro objeto.
Representa una Abstracción







Representa una clase concreta la cual tiene la implementació


Cuando dentro de un mensaje se crea un objeto
FIUBineta
:Usuario
:Viaje
viajeEnRango:Viaje
destino:Ubicacion
:Patineta
obtenerMetrosRecorridos Entre(fechaInicial, fechaFinal)
<<create>>
:RangoFecha
loop
[Para cada viaje]
perteneceAlRango(rangoDeFecha)
contiene(fecha)
true
true
agregarViajeEnRango(viaje)
loop
[Para cada viaje en rango]
obtenerDistanciaEnMetros()
obtenerDistanciaEnMetrosA(origen)
metros
ajustarDistanciaEnMetros(metros)
metrosAjustados
metrosTotales = metrosTotales + metrosAjustados
metrosTotales

4. OTROS DIAGRAMAS UML

Diagramas de comportamiento

1. Diagramas de actividad.
2. Diagramas de casos de uso.
3. Diagramas de interacción.
4. Diagramas de tiempo.
5. Diagramas de estados.
6. Diagramas de comunicación o

requerido en el TP2

Diagramas de estructura

1. Diagramas de objetos.
2. Diagramas de componentes.
3. Diagrama de composición estructural.
4. Diagramas de despliegue.
5. Diagramas de paquetes.
6. Profile Diagram.

requerido en el TP2

GRACIAS!

¿ Dudas / Preguntas / Consultas ?

- Utilizar el campus !