

UT03.-DISEÑO ORIENTADO A OBJETOS. DIAGRAMAS ESTRUCTURALES.



UNIFIED MODELING LANGUAGE

3. UML

UML

- **UML** - *Unified Modeling Language* o *Lenguaje Unificado de Modelado*:
 - Conjunto de herramientas que permite **modelar, construir y documentar** los elementos que forman un sistema software orientado a objetos (todas las partes que forman parte del desarrollo del software).
 - Modelos que representan el sistema desde diferentes perspectivas o diagramas estandarizados.
 - Versiones de UML:
 - UML 1.X: desde finales de los 90.
 - UML 2.X: entorno a 2005

- ¿Por qué es útil?
 - Uso de un lenguaje común entre todo el equipo de desarrollo.
 - Permite la documentación del proceso de desarrollo del software (requisitos, arquitectura, pruebas, versiones,...).
 - Puede representar estructuras referentes a la arquitectura del sistema (relaciones entre módulos, nodos en los que se ejecuta en sistemas distribuidos).
 - Modelos precisos, no ambiguos y completos en las decisiones de análisis, diseño e implementación.
 - Conexión a lenguajes de programación mediante ingeniería directa e inversa.

Elementos de los diagramas UML.

- **Sistema** (definición de Sistema según el estándar UML)
 - ▣ Conjunto de modelos que describen sus diferentes perspectivas, implementados en una serie de diagramas (representaciones gráficas de una colección de elementos de modelado), a menudo dibujado como un grafo conexo de arcos (relaciones) y vértices (otros elementos del modelo).
- **Elementos:**
 - ▣ **Estructuras:** Son los nodos del grafo y definen el tipo de diagrama.
 - ▣ **Relaciones:** Son los arcos del grafo que se establecen entre los elementos estructurales.
 - ▣ **Notas:** Se representan como un cuadro donde podemos escribir comentarios que nos ayuden a entender algún concepto que queramos representar.
 - ▣ **Agrupaciones:** Se utilizan cuando modelamos sistemas grandes para facilitar su desarrollo por bloques.

Tipos de diagramas UML I

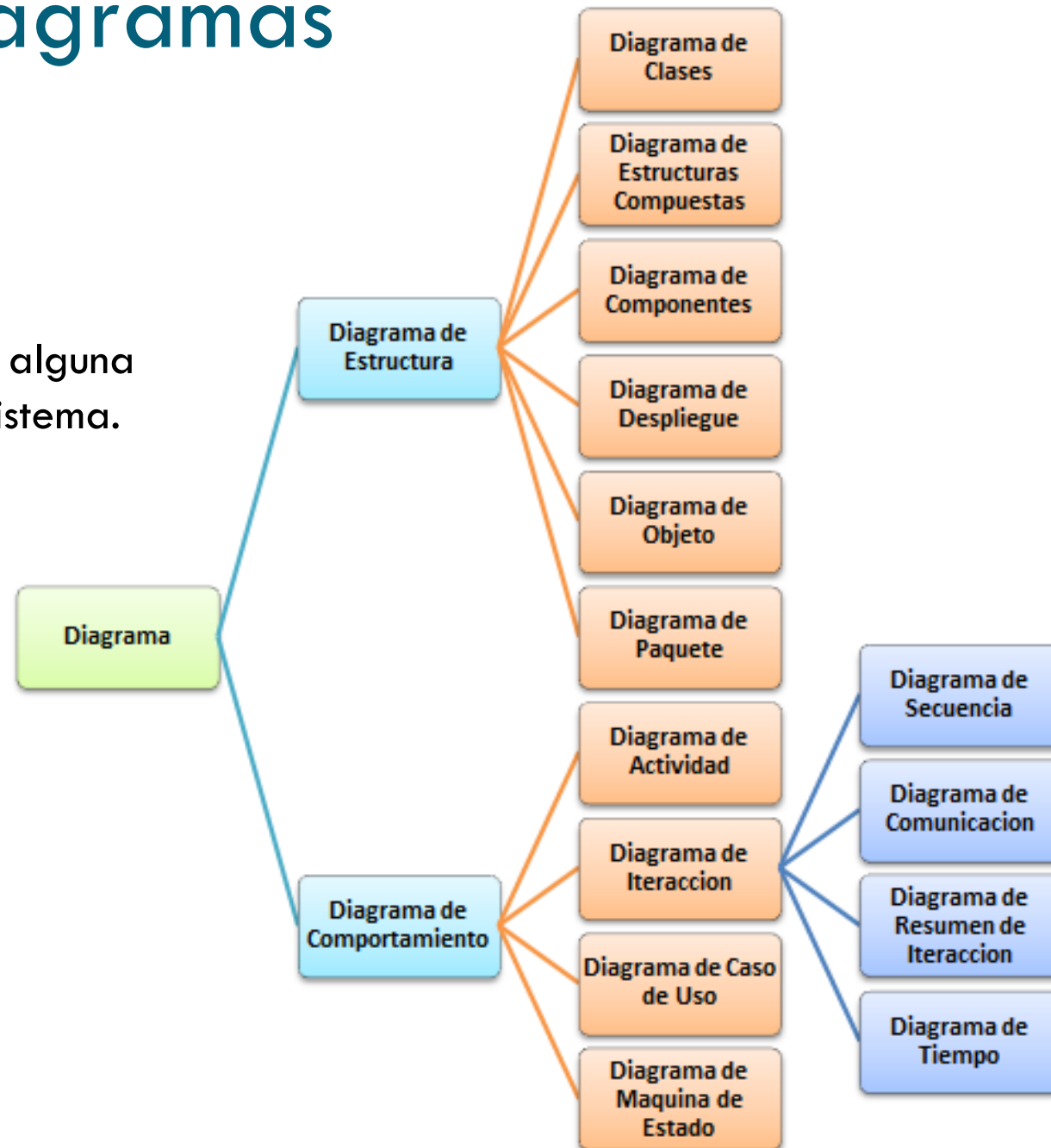
□ Clasificación:

- ▣ **Diagramas de estructura o estructurales:** representan la visión **estática** del sistema.
 - Especifican clases y objetos y como se distribuyen físicamente en el sistema.
- ▣ **Diagramas de comportamiento:** muestran lo que sucede **en tiempo de ejecución en el sistema**, tanto desde el punto de vista del sistema completo como de las instancias u objetos que lo integran.
 - ▣ Dentro de este grupo están los **diagramas de interacción**. Se centran en el flujo de control y de datos entre los elementos del sistema.

Tipos de diagramas

UML II

Cada diagrama representa alguna parte o punto de vista del sistema.



Tipos de diagramas UML III

□ Diagramas estructurales:

- ▣ **Diagramas de clases:** muestran los elementos del modelo estático abstracto, y está formado por un conjunto de clases y sus relaciones. Prioridad ALTA.
- ▣ **Diagrama de objetos:** conjunto de objetos y sus relaciones en un momento concreto. Prioridad ALTA.
- ▣ **Diagrama de componentes:** especifican la organización lógica de la implementación de una aplicación, sistema o empresa, indicando sus componentes, sus interrelaciones, interacciones y sus interfaces públicas y las dependencias entre ellos. Prioridad MEDIA.
- ▣ **Diagramas de despliegue:** representan la configuración del sistema en tiempo de ejecución. Muestra como los componentes de un sistema se distribuyen entre los ordenadores que los ejecutan. Se utiliza cuando tenemos **sistemas distribuidos**. Prioridad MEDIA.
- ▣ **Diagrama integrado de estructura (UML 2.0):** muestra la estructura interna de una clasificación (tales como una clase, componente o caso típico), e incluye los puntos de interacción de esta clasificación con otras partes del sistema. Tiene una prioridad BAJA.
- ▣ **Diagrama de paquetes:** Muestra cómo los elementos del modelo se organizan en paquetes, así como las dependencias entre esos paquetes. Suele ser útil para la gestión de sistemas de mediano o gran tamaño. Prioridad BAJA.

Tipos de diagramas UML IV

□ Diagramas de comportamiento:

- **Diagramas de casos de uso:** representan las **acciones a realizar** en el sistema desde el punto de vista de los usuarios (interacción con el usuario y otros sistemas). Prioridad MEDIA.
- **Diagramas de estado de la máquina:** describen el comportamiento de un sistema dirigido por **eventos** (estados que pueden tener un objeto o interacción, así como las transiciones entre dichos estados). Se denomina también diagrama de estado, diagrama de estados y transiciones o diagrama de cambio de estados. Prioridad MEDIA.
- **Diagrama de actividades:** muestran el orden en el que se van realizando tareas dentro de un sistema. En él aparecen los procesos de alto nivel de la organización. Incluye flujo de datos, o un modelo de la lógica compleja dentro del sistema. Prioridad ALTA.

Tipos de diagramas UML V

- ▣ **Diagramas de interacción:**

- ▣ **Diagramas de secuencia:** representan la ordenación temporal en el paso de mensajes. Prioridad ALTA.
- ▣ **Diagramas de comunicación/colaboración (UML 2.0):** resaltan la organización estructural de los objetos que se pasan mensajes. Ofrece las instancias de las clases, sus interrelaciones, y el flujo de mensajes entre ellas. Comúnmente enfoca la organización estructural de los objetos que reciben y envían mensajes. Prioridad BAJA.
- ▣ **Diagrama de interacción:** muestra un conjunto de objetos y sus relaciones junto con los mensajes que se envían entre ellos. Es una variante del diagrama de actividad que permite mostrar el flujo de control dentro de un sistema o proceso organizativo. Cada nodo de actividad dentro del diagrama puede representar otro diagrama de interacción. Tiene una prioridad BAJA.
- ▣ **Diagrama de tiempos:** muestra el cambio en un estado o una condición de una instancia o un rol a través del tiempo. Prioridad BAJA.

Herramientas para la elaboración de diagramas UML.

- Herramientas CASE que
 - ▣ Permiten desarrollo de los diagramas UML.
 - ▣ Cuentan con *un entorno wysiwyg* ("lo que ves es lo que obtienes")
 - ▣ Permiten documentar los diagramas.
 - ▣ Permiten integrarse con otros entornos de desarrollo.

Herramientas para la elaboración de diagramas UML II

□ Herramientas:

- **Rational Systems Developer de IBM:** Herramienta *proprietaria* que permite el desarrollo de proyectos software basados en la metodología UML. Desarrollada en origen por los creadores de UML ha sido recientemente absorbida por IBM. Ofrece versiones de prueba, y software libre para el desarrollo de diagramas UML.
- **Visual Paradigm for UML (VP-UML):** Incluye una versión para uso no comercial que se distribuye libremente sin más que registrarse para obtener un archivo de licencia. Incluye diferentes módulos para realizar desarrollo UML, diseñar bases de datos, realizar actividades de ingeniería inversa y diseñar con Agile. Es compatible con los IDE de Eclipse, Visual Studio .net, IntelliJDEA y NetBeans. Multiplataforma, incluye instaladores para Windows y Linux.
- **ArgoUML: herramienta de código abierto.** Soporta los diagramas de UML 1.4, y genera código para java y C++. Para poder ejecutarlo se necesita la plataforma java (JDK). Admite ingeniería directa e inversa.

[Descargar ArgoUML](#)

Diagramas de clases o estructuras.

- Estructura del sistema. Es un elemento estático.
- Describe el sistema mostrando sus clases y relaciones entre ellas. Sirve para visualizar las relaciones entre las clases del sistema.
- Elementos:
 - ▣ **Clases:** atributos, métodos y visibilidad.
 - ▣ **Relaciones,** relaciones reales entre los elementos del sistema a los que hacen referencia las clases. Pueden ser de asociación, agregación, composición, herencia ...
 - ▣ **Notas:** comentarios que nos ayuden a entender algún concepto que queramos representar.
 - ▣ **Elementos de agrupación:** las clases y sus relaciones se agrupan en paquetes, que a su vez se relacionan entre sí.

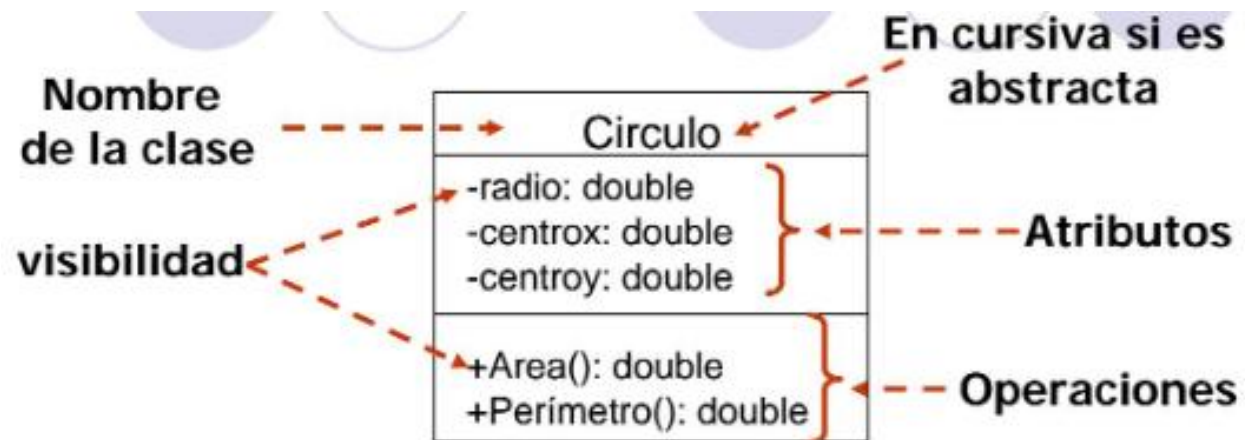
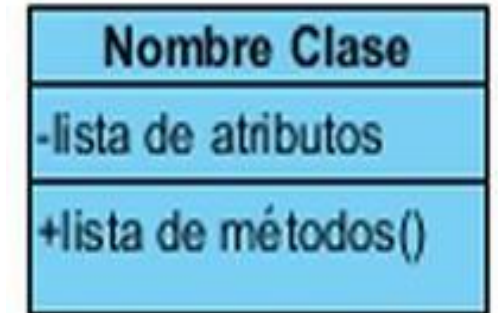
Diagramas de clases II

Un objeto es cualquier persona, lugar, cosa, concepto, acontecimiento, pantalla, o el informe correspondiente a su sistema. Los objetos tienen propiedades (tienen atributos) y hacen cosas (tienen métodos). **Una clase** es una representación de un objeto y, en muchos sentidos, es simplemente una plantilla a partir de la cual se crean los objetos. **Las clases son la unidad básica que encapsula toda la información de un objeto.**

Creación de clases

□ Clases

- Se representa como un rectángulo dividido en tres filas
 - Nombre de la clase
 - Atributos con su visibilidad
 - Métodos con su visibilidad

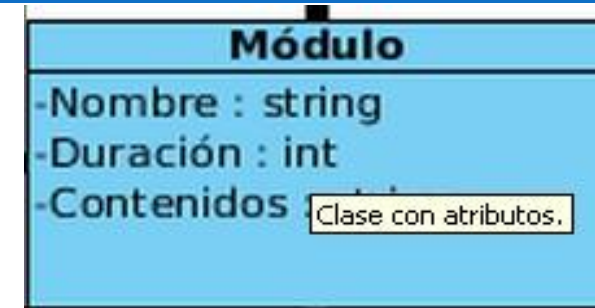


+	Público
-	Privado
#	Protegido
/	Derivado (se puede combinar con otro)
~	Paquete

Tipos de **visibilidad**
La clase también tiene una determinada visibilidad

Atributos

- Nombre
- Tipo de dato
 - ▣ Simple
 - ▣ Compuesto, pudiendo incluir otra clase.
 - Los tipos de datos básicos de UML son: Integer, String y Boolean. También se pueden indicar los tipos de cualquier lenguaje de programación.
- Visibilidad
 - ▣ **Público:** acceso desde cualquier clase y cualquier parte del programa. Visible para todos
 - ▣ **Privado:** acceso desde operaciones de la clase. Visible solo en la clase.
 - ▣ **Protegido:** acceso desde operaciones de la clase o de clases derivadas en cualquier nivel. Visible en la clase y las subclases de la clase
 - ▣ **Paquete:** acceso desde las operaciones de las clases que pertenecen al mismo paquete que la clase que estamos definiendo. Visible en las clases del mismo paquete



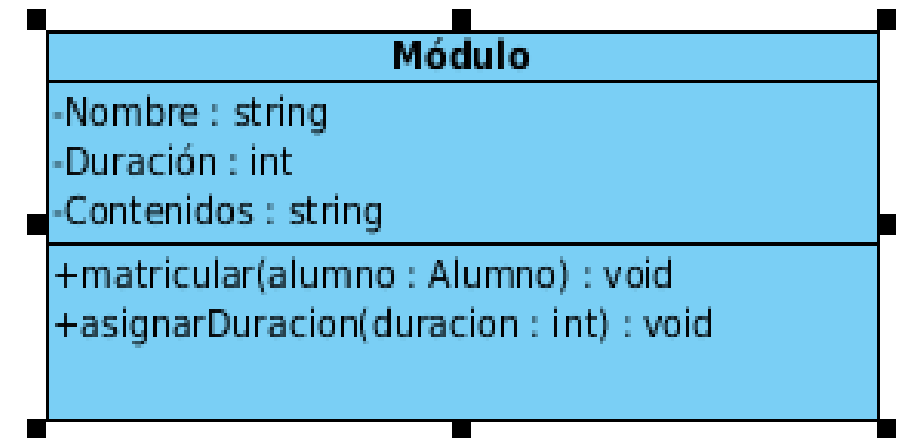
Clases UML y código Java

```
public class ComplexNumber {  
  
    private double r;  
    private double i;  
  
    public ComplexNumber(double r, double i) {  
        this.r = r;  
        this.i = i;  
    }  
  
    public double norm() {  
        return Math.sqrt(r * r + i * i);  
    }  
}
```

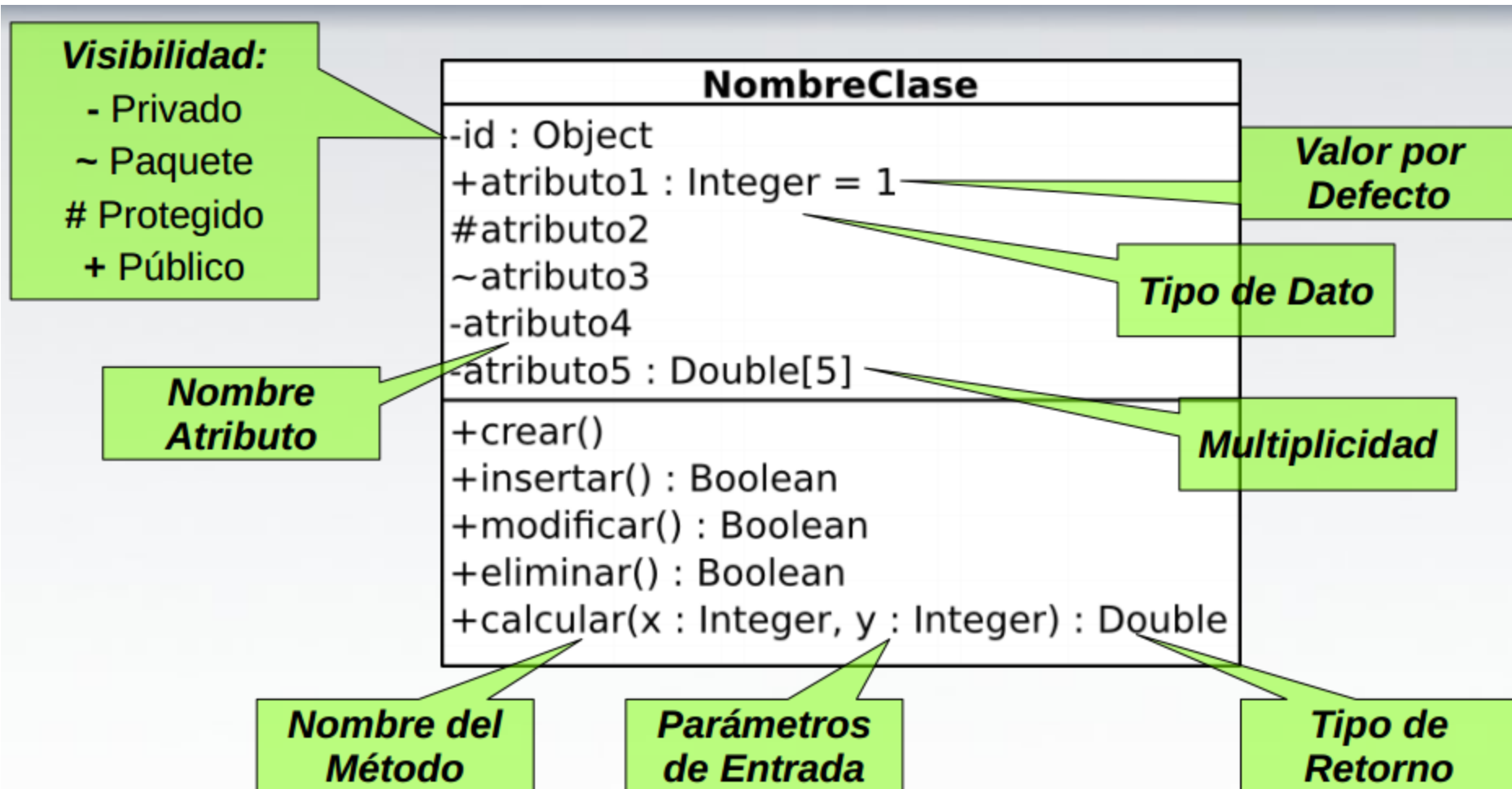
ComplexNumber
-r : double -i : double
+ComplexNumber(r : double, i : double) +norm() : double

Métodos

- Definir nombre, parámetros, el tipo que devuelve y su visibilidad y la descripción del método que aparecerá en la documentación que se genere del proyecto.
- **Constructor** de la clase
 - ▣ No devuelve ningún valor
 - ▣ Tiene el mismo nombre que la clase
 - ▣ Para instanciar objetos de clases
 - ▣ Existe una función especial para que el objeto deje de existir



Clases con métodos y atributos



Sin título - Diagrama de clase - ArgoUML *

Archivo Editar Visualizar Crear Organizar Generar Crítica Herramientas Ayuda

Orientada a paquetes

Profile Configuration
modelo sin título

Nombre de la clase

Crear clase en un diagrama

Atributos de la clase

Métodos de la clase

Sobre el modelo creamos un Diagrama

Estudiante

dni : String
nombre : String

altaEst()

As Diagram

Por prioridad 3 elementos

Alta
Media
Baja

Presentación Código fuente Restricciones Estereotipos Valores etiquetados Lista de control

Tarea pendiente

Propiedades

Dependencias del cliente
Dependencias del surtidor:
Generalización
Especialización:

Nombre Clase

Estudiante

Espacio de nombres

modelo sin título

Visibilidad:

☒ Pública ☐ Paquete ☐ Protegida ☐ Privada

modifiers

☐ isRoot ☐ isLeaf ☐ isAbstract ☐ isActive

Parámetros de la plantilla:

Atributos:
Operaciones:
Cabo de asociación
Elementos que posee:

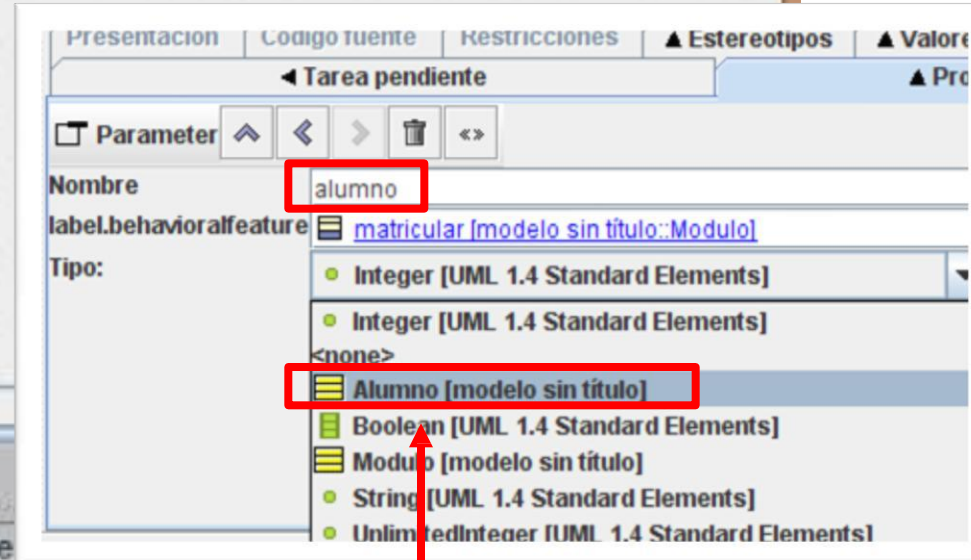
dni
altaEst

40M used of 508M max

Crear parámetro en un método

Seleccionar el método

1



Seleccionar, en Parámetro>> Crear parámetro, y rellenar el nombre y tipo

2

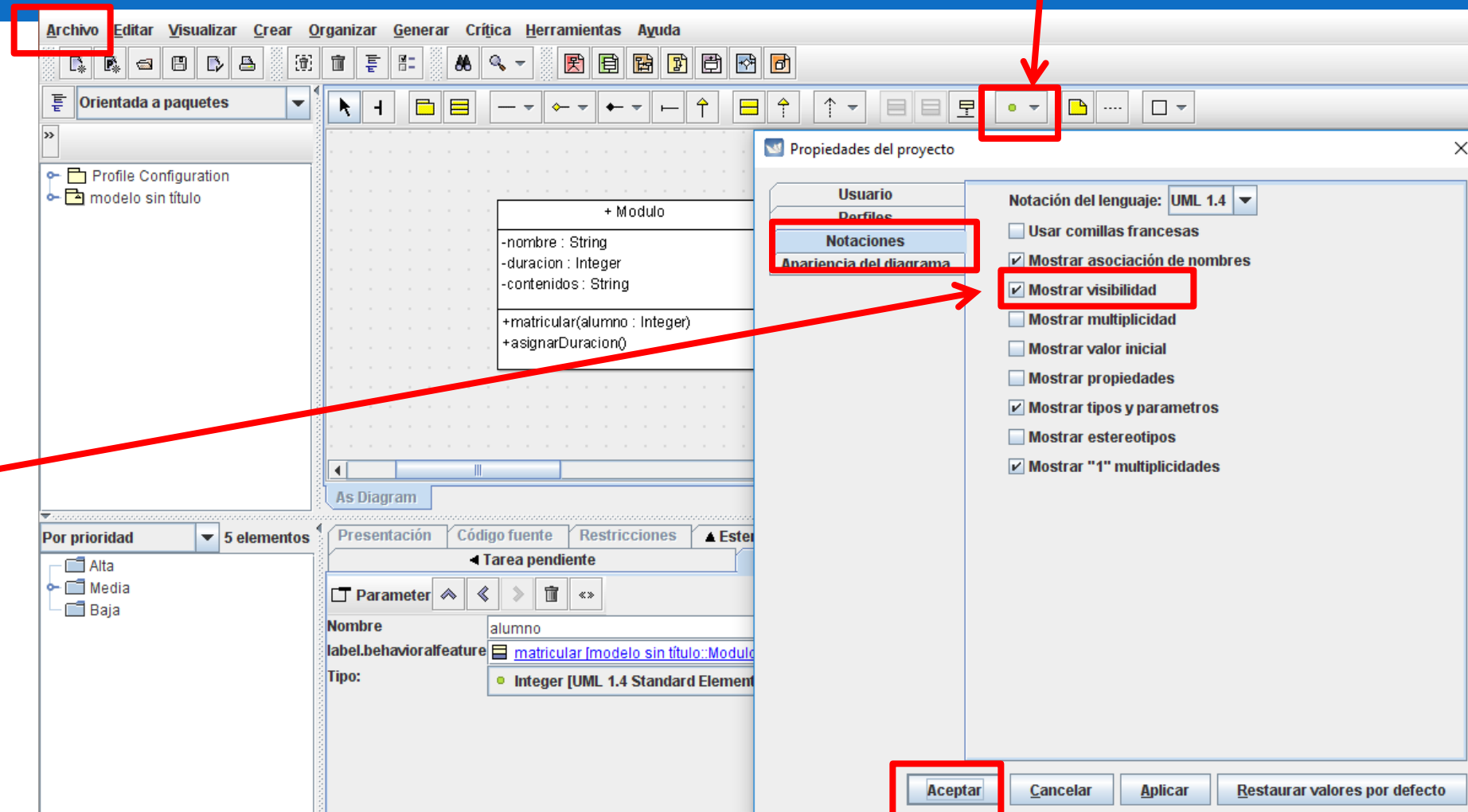


ArgoUML

- Para mostrar la visibilidad
 - Menú Archivo>>Propiedades>>Notaciones>>Mostrar visibilidad

[Vídeo explicativo](#)

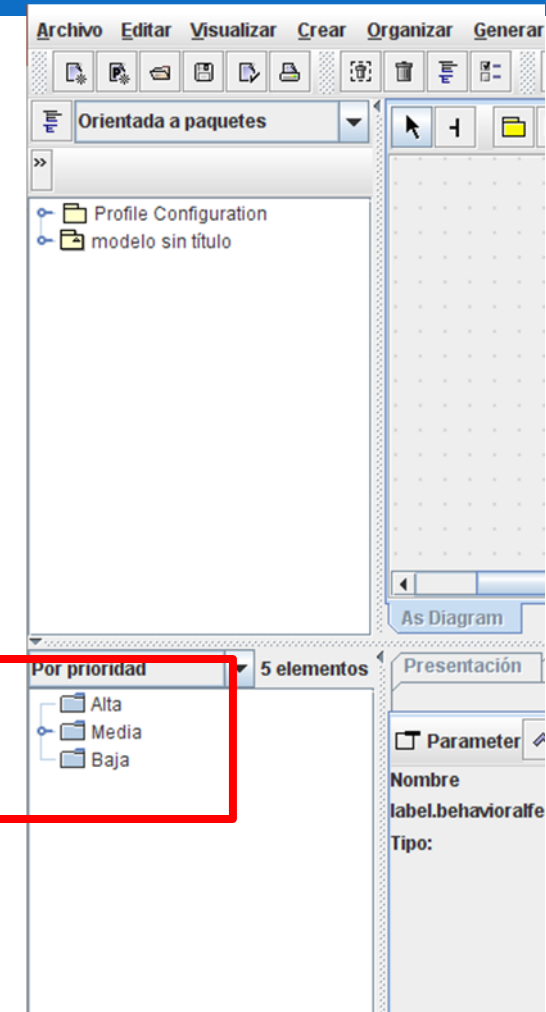
Crear tipos de datos



ArgoUML

❑ Criticas

- En la ventana a la izquierda de las propiedades, aparecen las críticas, que son recomendaciones que deberíamos seguir para obtener un buen diagrama UML.



Criticas

Seleccionamos la clase
Seleccionamos la critica
Siguiendo para ver la solución

The screenshot displays the Eclipse IDE interface. On the left, the 'Package Explorer' shows a project named 'modelo sin título' with a package 'Alta' and a sub-package 'Media'. The 'Media' package contains several tasks, with 'Añade constructor a Empleado' highlighted in red. The main editor area shows a UML class diagram with two classes: 'Empleado' and 'Departamento'. The 'Empleado' class has attributes '-codigo : Integer', '-nombre : String', '-oficio : String', and '-salario : Integer'. It has methods '+setCodigo(codigo : Integer)', '+setNombre(nombre : String)', '+setOficio(oficio : String)', '+setSalario(salario : Integer)', '+getCodigo() : Integer', '+getNombre() : String', '+getOficio() : String', and '+getSalario() : Integer'. The 'Departamento' class has attributes '-nombre : String', '-codigo : Integer', and '-localidad : String'. It has methods '+setNombre(nombre : String)', '+setCodigo(codigo : Integer)', '+setLocalidad(localidad : String)', '+getNombre() : String', '+getCodigo() : Integer', and '+getLocalidad() : String'. The diagram shows a 'trabajan' association between 'Empleado' (multiplicity 1..*) and 'Departamento' (multiplicity 1). There is also a 'es jefe' association between 'Empleado' (multiplicity 1) and 'Empleado' (multiplicity 1..*). The bottom panel shows a task description for 'Añade constructor a Empleado'. The task text is: 'Aun no has definido un constructor para la clase Empleado. Los constructores inicializan nuevas instancias dotando de valores válidos a sus atributos. Esta clase probablemente necesite un constructor porque no todos sus atributos tienen valores iniciales. Definir buenos constructores es clave para establecer class invariants, y class invariants son una poderosa ayuda para escribir código sólido. Para arreglar esto, pulsa el botón "Siguiendo>", o añade manualmente un constructor pulsando en Empleado en el panel de navegación y usando el menú Create para hacer un constructor nuevo.' The 'Siguiendo' button is highlighted in a red box.

UML Class Diagram:

```
classDiagram
    class Empleado {
        -codigo : Integer
        -nombre : String
        -oficio : String
        -salario : Integer
        +setCodigo(codigo : Integer)
        +setNombre(nombre : String)
        +setOficio(oficio : String)
        +setSalario(salario : Integer)
        +getCodigo() : Integer
        +getNombre() : String
        +getOficio() : String
        +getSalario() : Integer
    }
    class Departamento {
        -nombre : String
        -codigo : Integer
        -localidad : String
        +setNombre(nombre : String)
        +setCodigo(codigo : Integer)
        +setLocalidad(localidad : String)
        +getNombre() : String
        +getCodigo() : Integer
        +getLocalidad() : String
    }
    Empleado "1..*" -- "1" Departamento : trabajan
    Empleado "1" -- "1..*" Empleado : es jefe
```

Task List (Por prioridad):

- Alta
- Media
 - Revisa el nombre del paquete modelo sin título
 - Añade constructor a Empleado**
 - Añade constructor a Departamento
 - Añade variables de instancia a Alumno
 - Añade operaciones a Alumno
 - Añade variables de instancia a Alumno
 - Añade operaciones a Alumno
 - Escoge un nombre

Task Description (Tarea pendiente):

Aun no has definido un constructor para la clase Empleado. Los constructores inicializan nuevas instancias dotando de valores válidos a sus atributos. Esta clase probablemente necesite un constructor porque no todos sus atributos tienen valores iniciales.

Definir buenos constructores es clave para establecer class invariants, y class invariants son una poderosa ayuda para escribir código sólido.

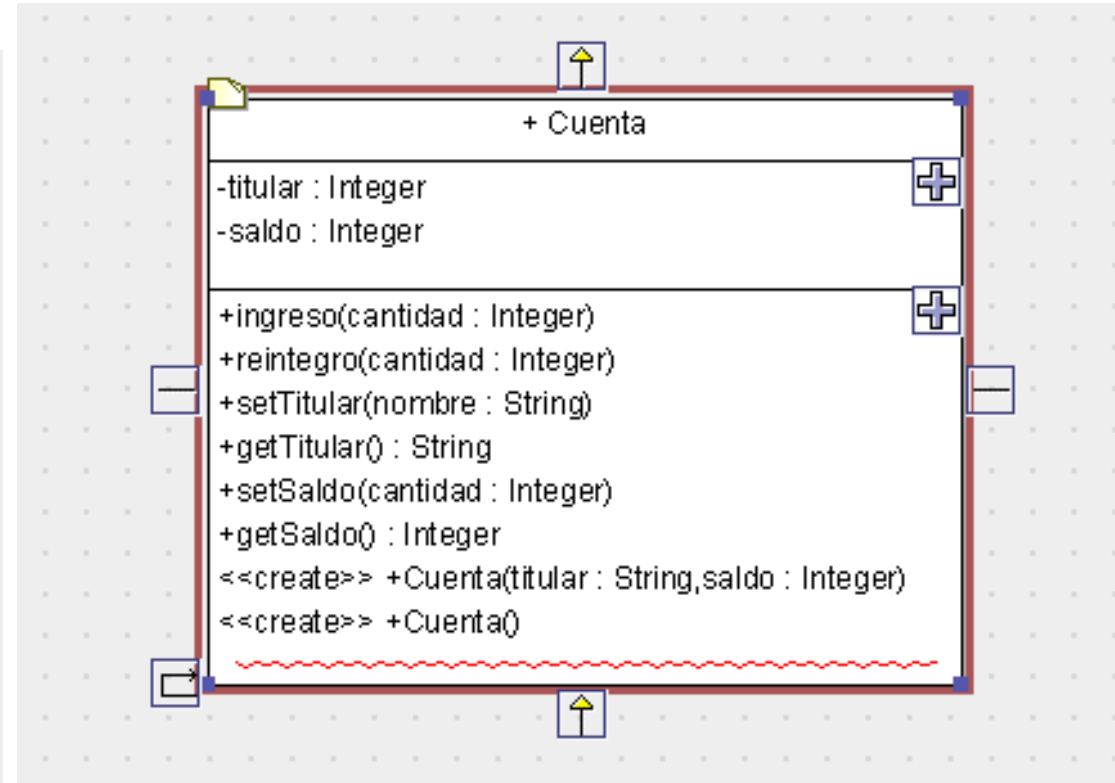
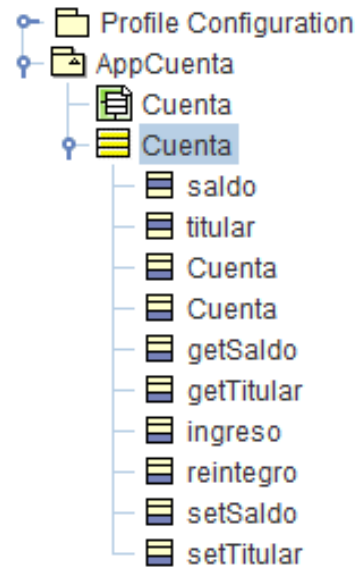
Para arreglar esto, pulsa el botón "Siguiendo>", o añade manualmente un constructor pulsando en Empleado en el panel de navegación y usando el menú Create para hacer un constructor nuevo.

Buttons: Retroceder, **Siguiendo**, Terminar, Ayuda

Ejercicio de clase

- Vamos a modelar la clase Cuenta desarrollada en el módulo de Programación.

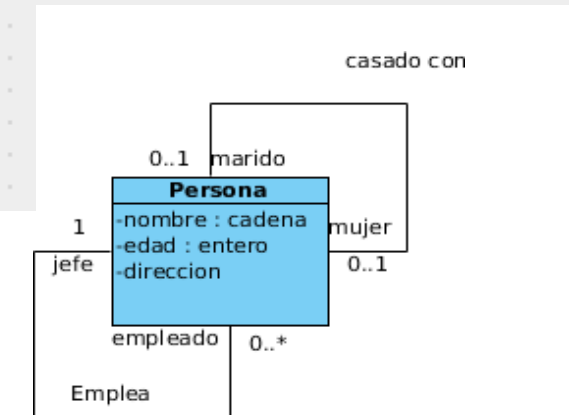
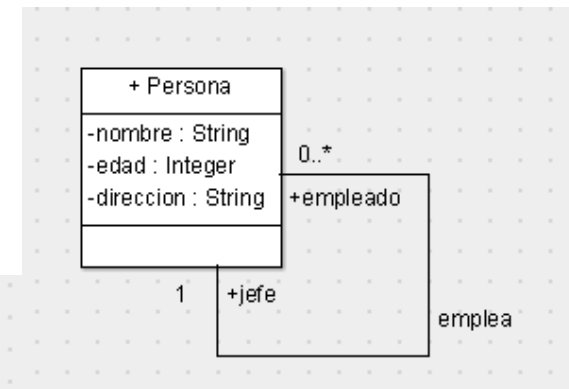
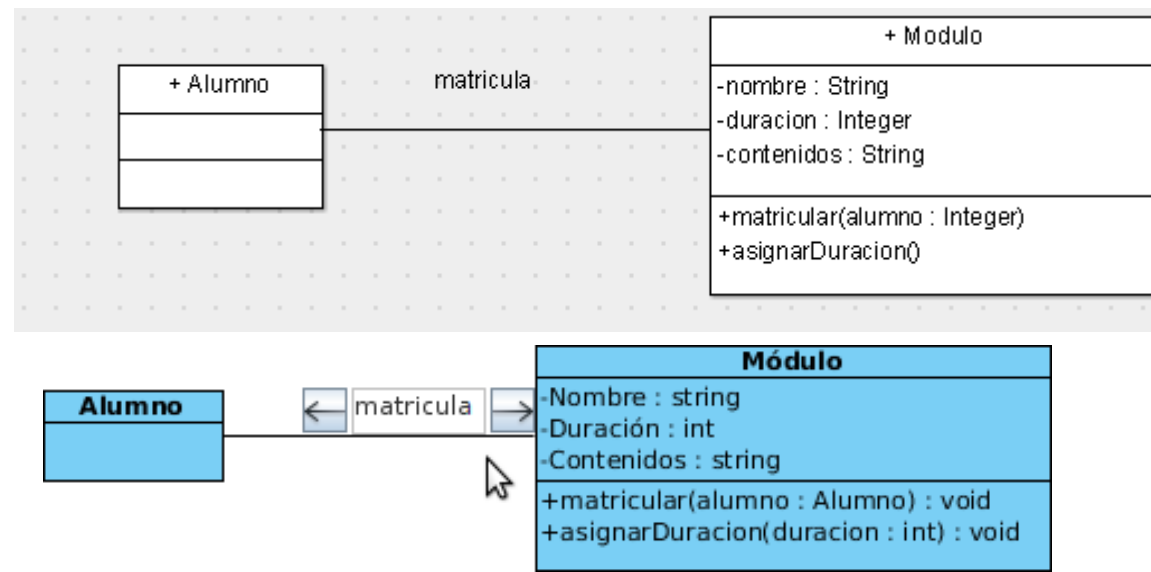
Ejercicio de clase. Solución



Relaciones entre clases.

- En el mundo real los objetos están relacionados entre sí. Por ejemplo, un alumno está relacionado con el módulo en el que está matriculado.
- Debemos representar la conexión entre dos clases cuando aparece algún tipo de relación entre ellas en el dominio del problema.
- Las relaciones tiene un nombre,
- Se representan mediante una línea continua (o discontinua) y
- Se caracterizan por su **cardinalidad** llamada **multiplicidad**
- Pueden ser
 - ▣ De herencia.
 - ▣ De asociación.
 - ▣ De composición.
 - ▣ De agregación.

[Video explicativo](#)



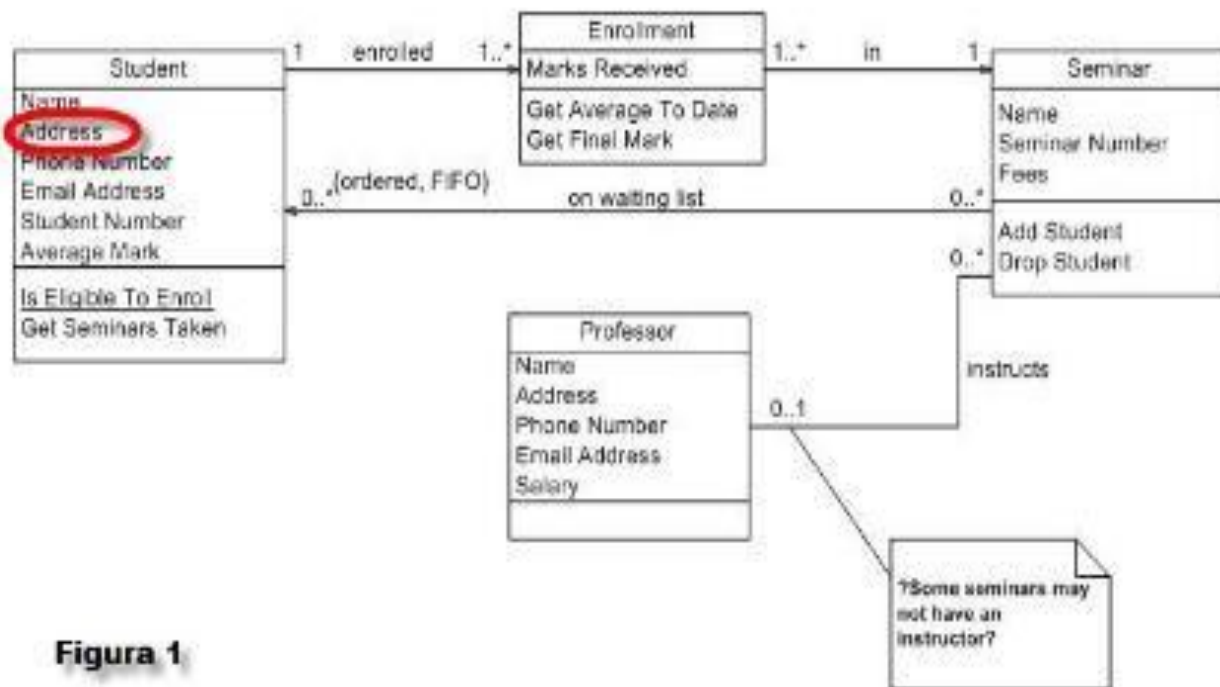


Figura 1

La clase *Student* de la figura 1 tiene un atributo llamado *dirección*. Las direcciones son complicadas, tienen muchos datos (ciudad ,calle,) por lo que puede ser una mejor manera de modelar una dirección representandola como en la figura 2, donde se modela la *dirección* como una clase.

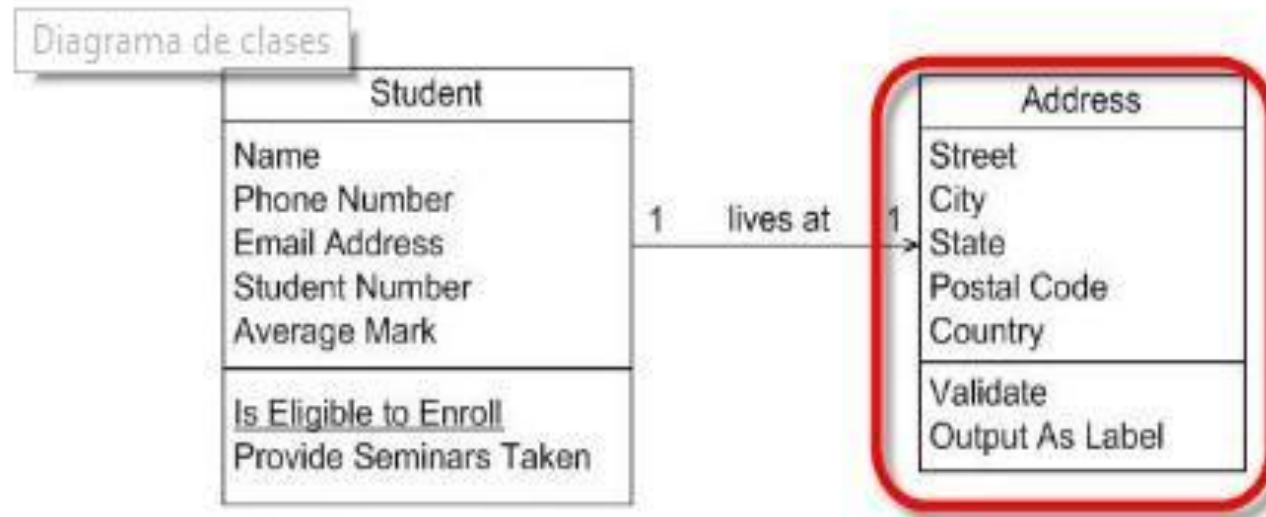


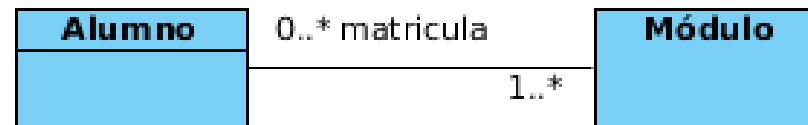
Figura 2

Un estudiante tiene una dirección y cada dirección pertenece a un estudiante.

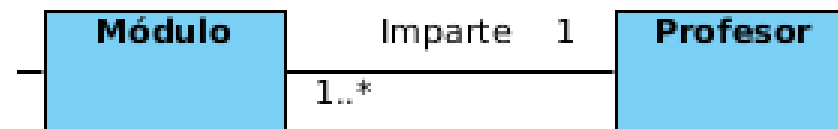
Cardinalidad o multiplicidad de la relación

- Cuántos objetos de una clase se van a relacionar con objetos de otra clase

Significado de las cardinalidades.	
Cardinalidad	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 (al menos uno)
N	N veces



Un alumno se matricula en uno o varios módulos y en un módulo están matriculados 0 o n alumnos

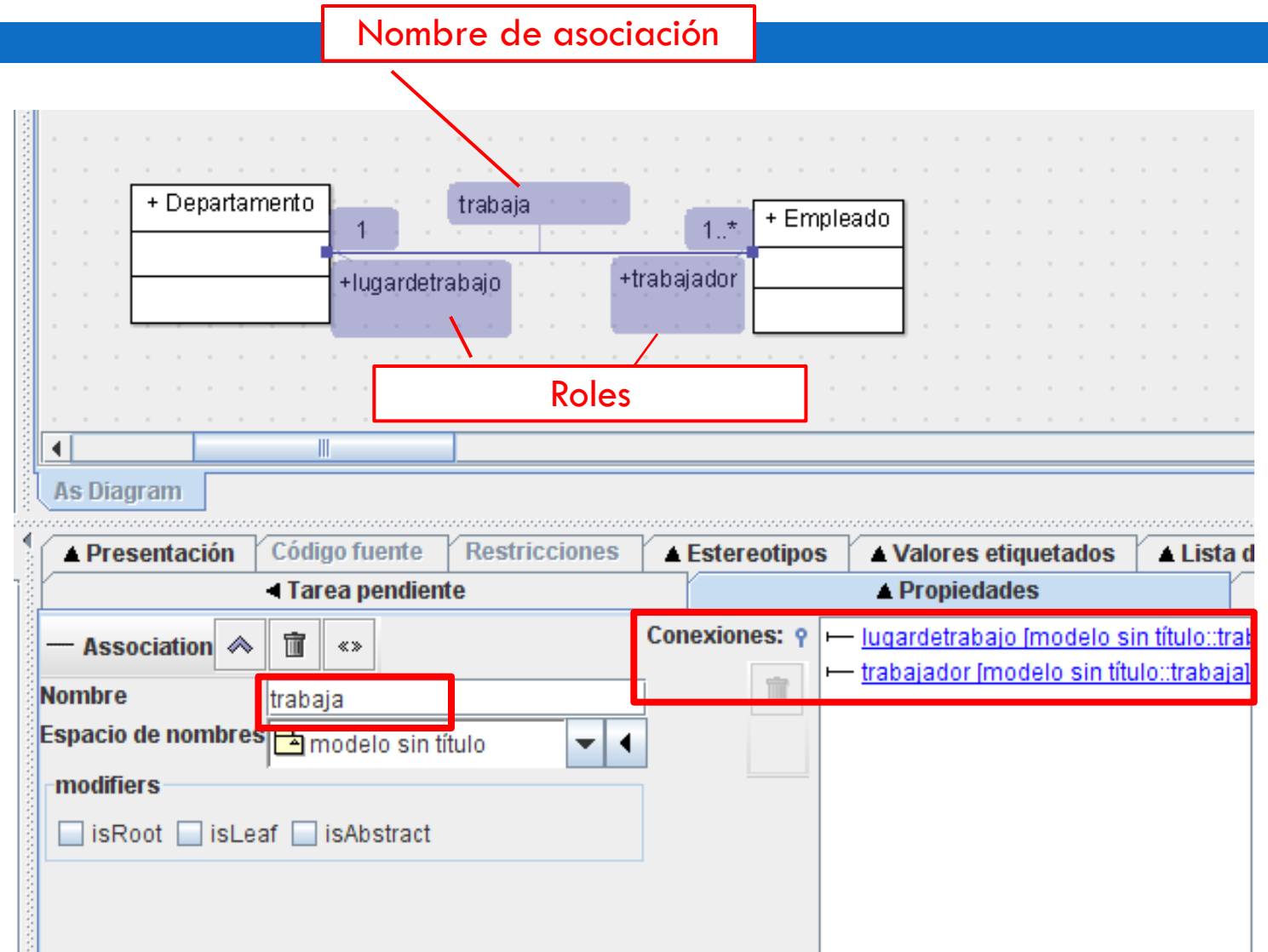


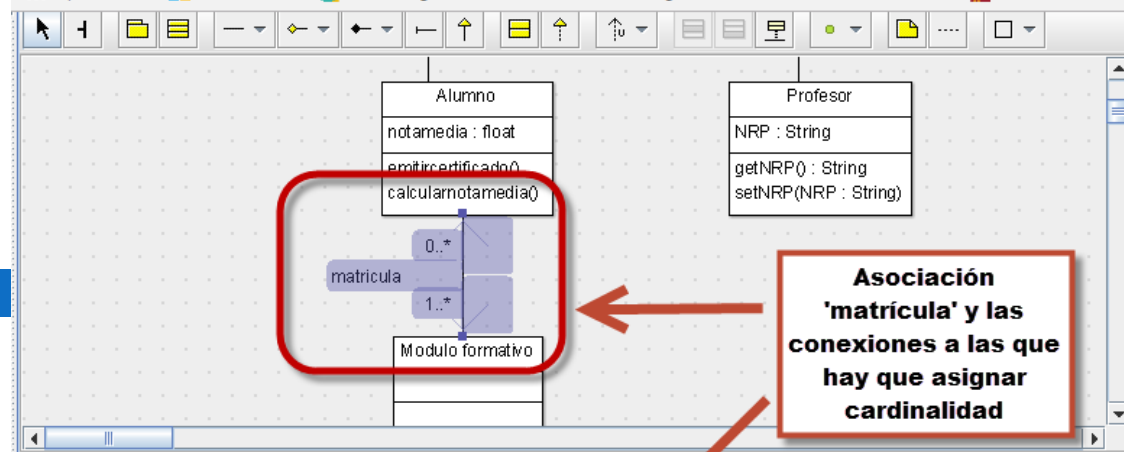
Un profesor imparte uno o varios módulos y un módulo es impartido por un único profesor

NOTA: Dependiendo del L.P las multiplicidades de destino mayores que 1 se pueden implementar como un atributo de tipo array o una colección.

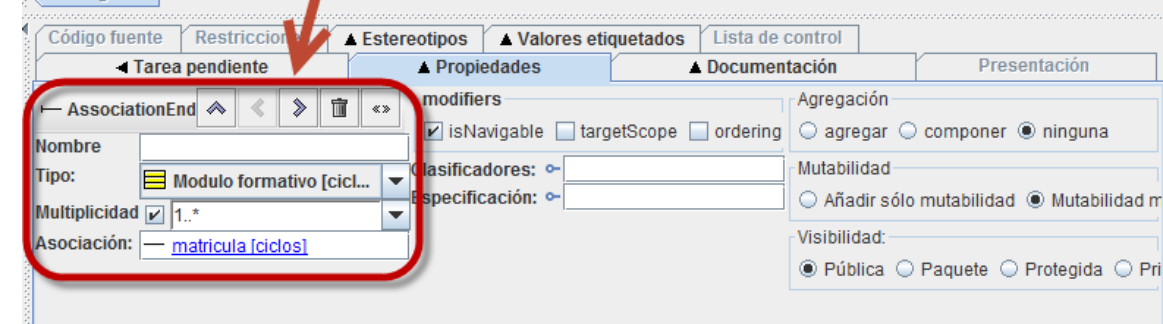
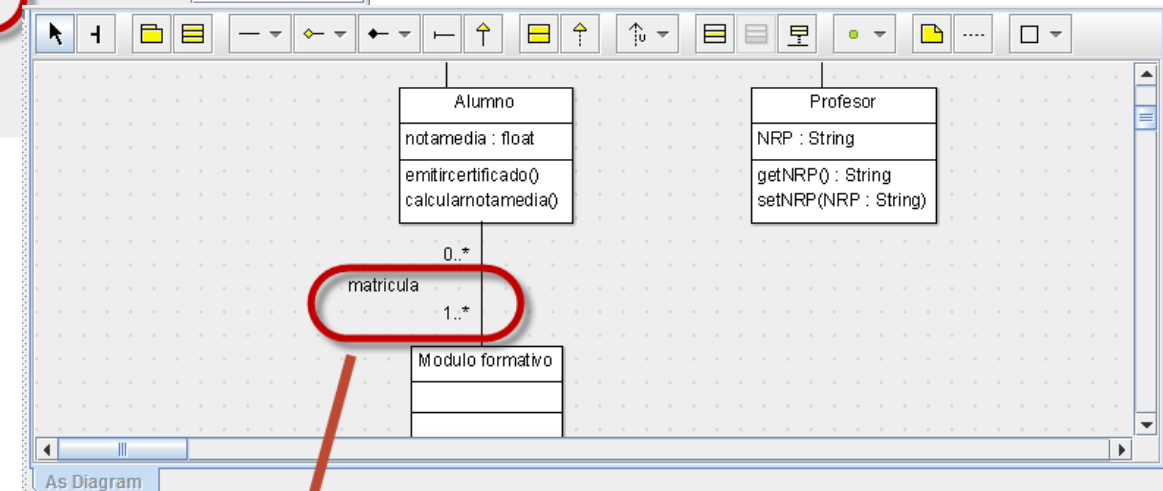
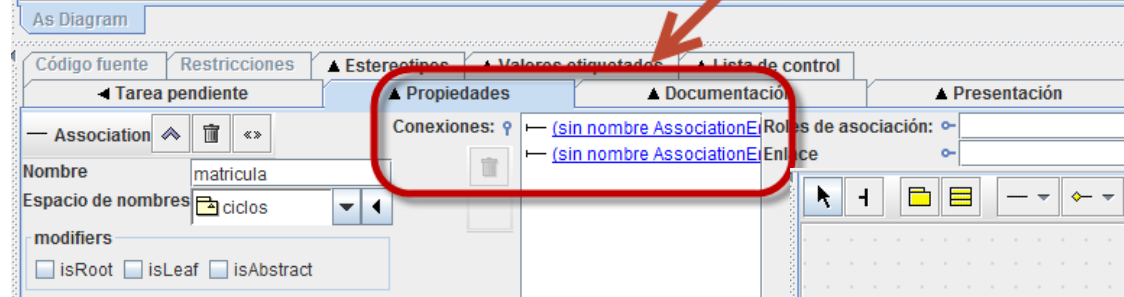
Cardinalidad o multiplicidad de la relación en ArgoUML

- Abrir la especificación de la relación >> doble-clic en la conexión >> establecer el apartado Multiplicidad a alguno de los valores que se indica o escribir el valor necesario.



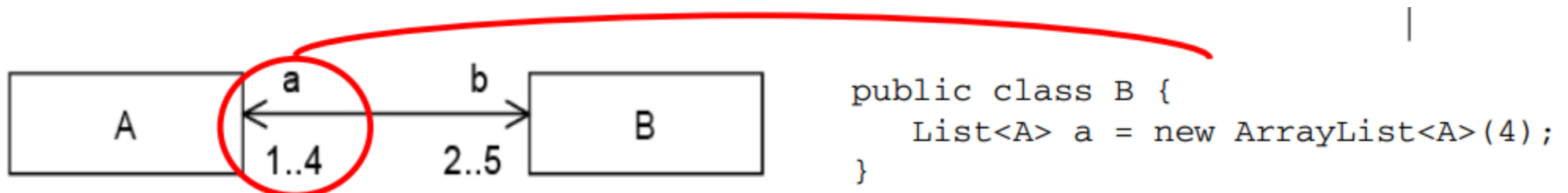


**Asociación
'matricula' y las
conexiones a las que
hay que asignar
cardinalidad**



Tipos de relaciones: Asociación

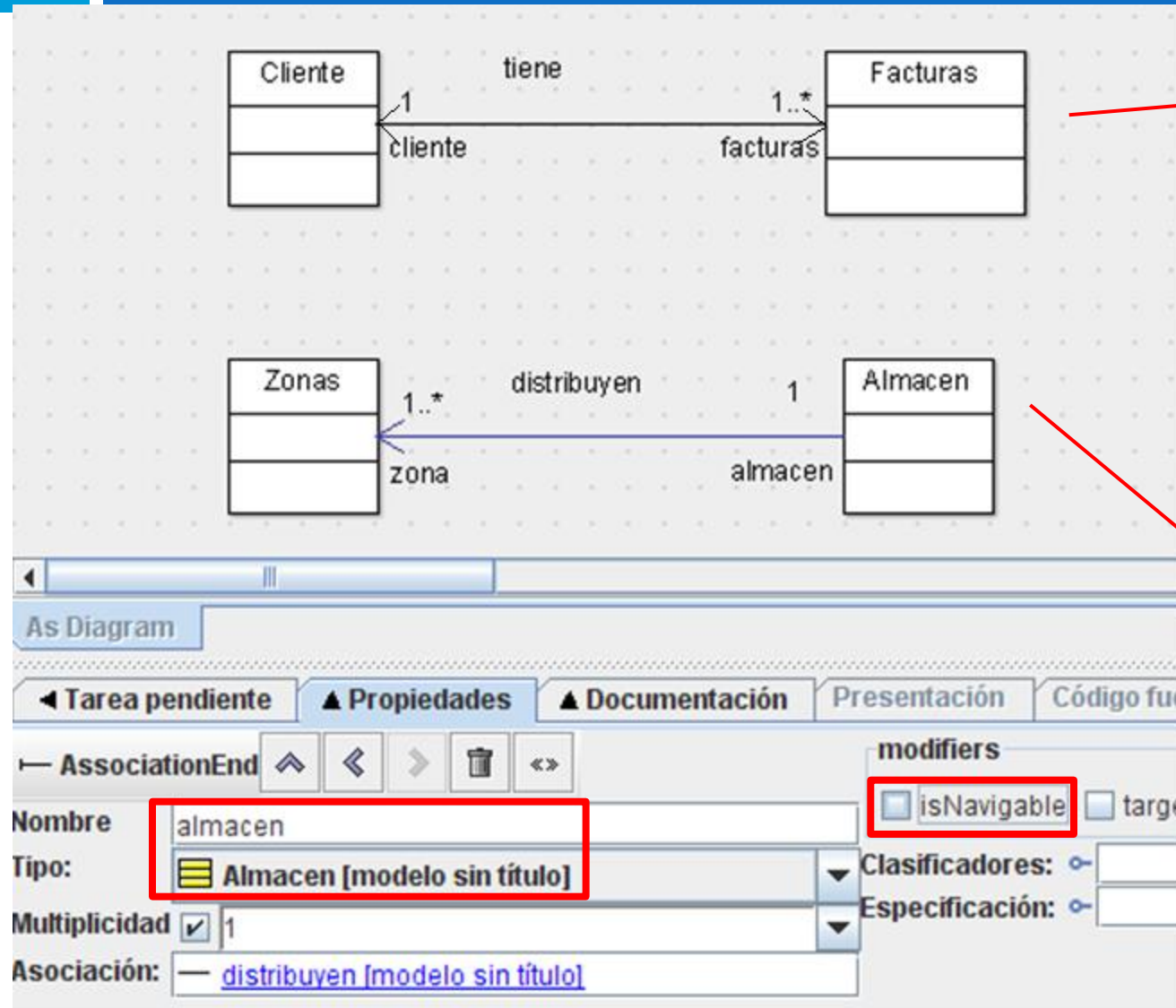
- Especifica que una clase utiliza otra clase.
- En una asociación, dos objetos A y B asociados entre sí existen de forma independiente → la creación o desaparición de uno de ellos implica únicamente la creación o eliminación de la relación entre ellos.
- Puede ser **unidireccional** o **bidireccional**, dependiendo de si ambas clases conocen o no la existencia de la otra. \longleftrightarrow
 \longrightarrow
- En una relación de asociación, cada clase juega un rol dentro de la relación que se indica en el extremo de la relación.
- La implementación en Java de dos clases unidas por una asociación **bidireccional**, implica que cada una de las clases tendrá un objeto o un conjunto de objetos de la otra clase, dependiendo de la multiplicidad entre ellas. (Navegabilidad en ambos sentidos).



Tipos de relaciones: Asociación

- En la asociación **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. (Navegabilidad en un sentido).
- **Navegabilidad.** Representa relaciones estructurales entre las clases. Cómo se relacionan. Hay asociaciones unidireccionales (navegabilidad de origen a destino) o bidireccionales.

Ejemplo Asociación



La asociación tiene: un cliente tiene muchas facturas y una factura es de un cliente.
Es bidireccional → ambas clases son navegables
Nota: La línea se puede representar sin flechas

La asociación distribuyen: un almacén distribuye en varias zonas y una zona es distribuida por un almacén.

Es unidireccional → es navegable de Almacén a Zona: Solo Almacén conoce la existencia de la clase Zonas, en cambio no es navegable de Zonas a Almacén.

Ejemplo Asociación. Código generado

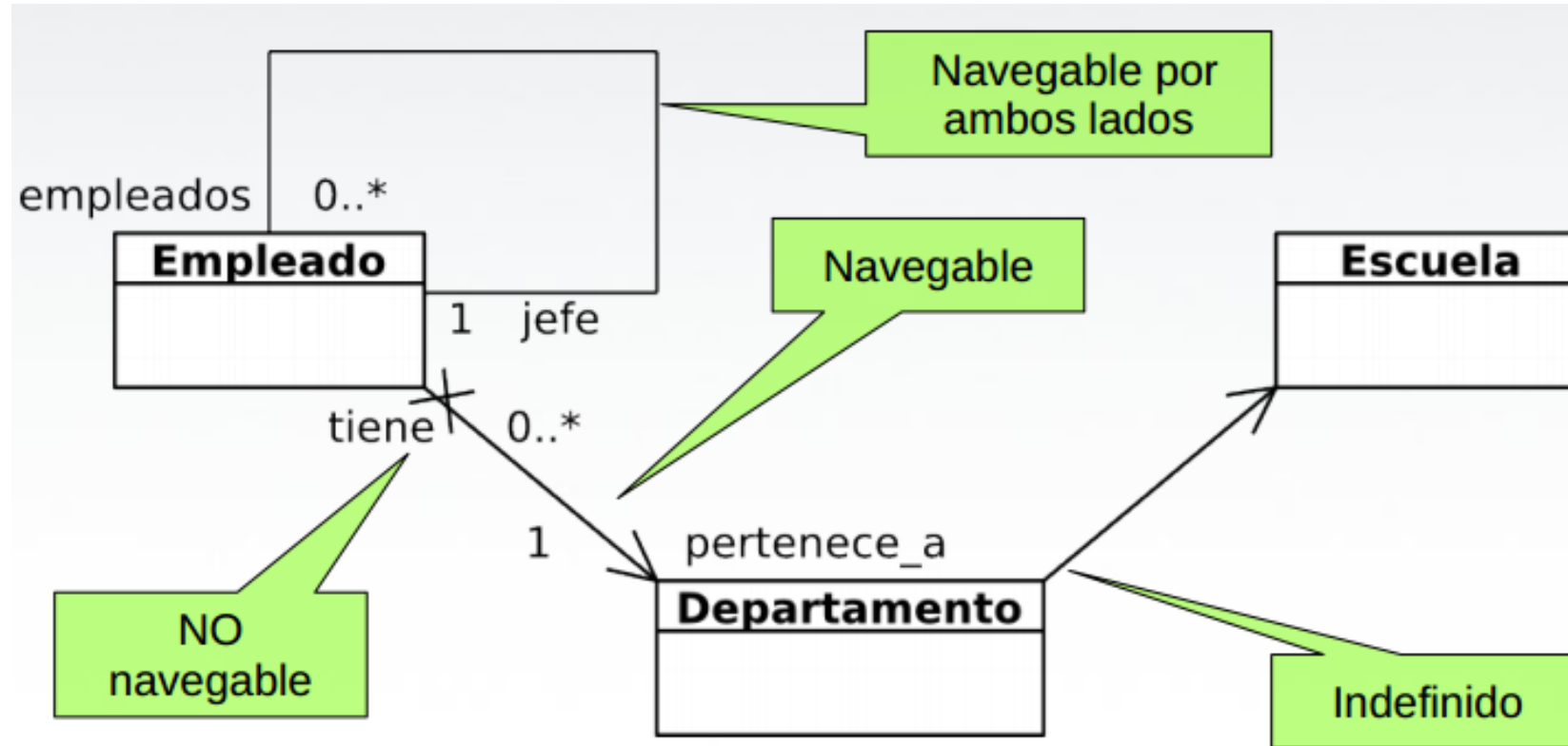
```
import java.util.Vector;
public class Cliente {
    public Vector facturas;
    ...
}

public class Facturas {
    public Cliente cliente;
    ...
}
```

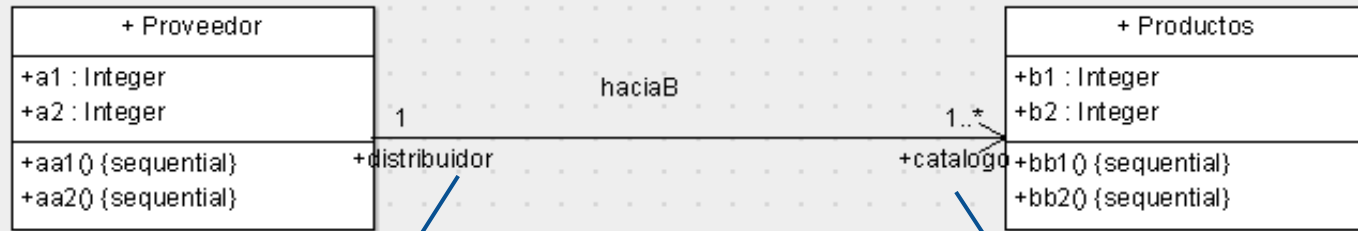
```
import java.util.Vector;
public class Almacen {
    public Vector zona;
    ...
}

public class Zonas {
    ...
}
```

Navegabilidad.



Ejemplo de navegabilidad.



No navegable

Navegable

```
import java.util.Vector;
```

```
public class Proveedor {
```

```
    public Integer a1;
```

```
    public Integer a2;
```

```
    /**
```

```
    *
```

```
    * @element-type Productos
```

```
    */
```

```
    public Vector catalogo;
```

```
    public void aa1() {
    }
```

```
    public void aa2() {
    }
```

```
}
```

```
public class Productos {
```

```
    public Integer b1;
```

```
    public Integer b2;
```

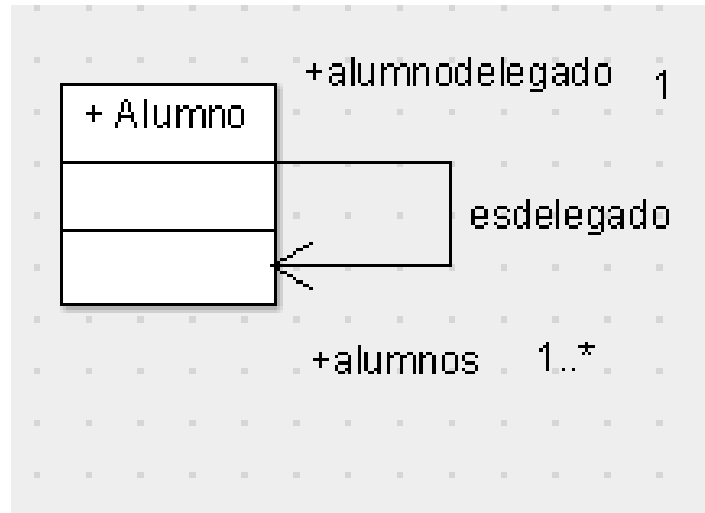
```
    public void bb1() {
    }
```

```
    public void bb2() {
    }
```

```
}
```

Asociación Reflexiva

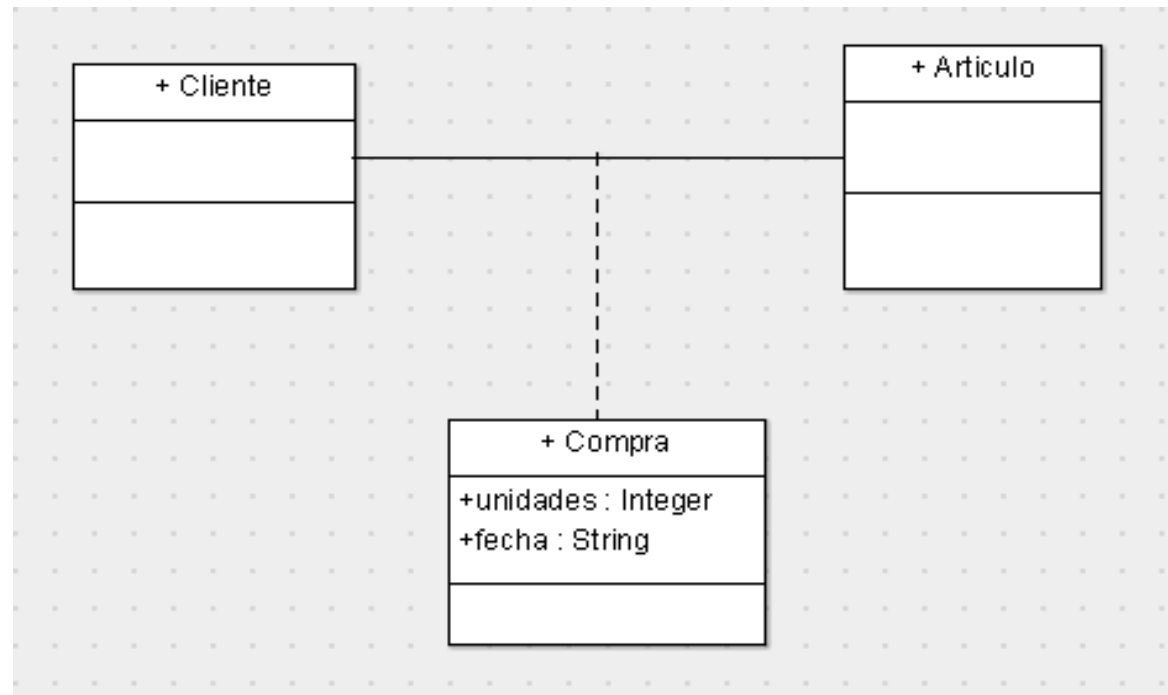
- Una clase puede asociarse consigo misma
- Estas asociaciones unen entre sí instancias de una misma clase.



Un alumno es delegado de muchos alumnos

Clase asociación

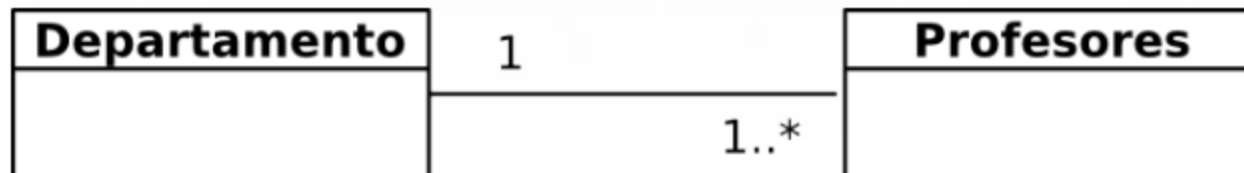
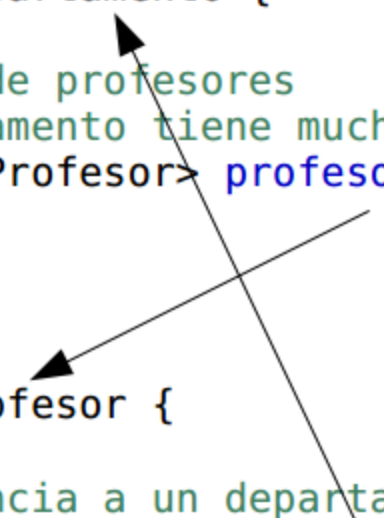
- Asociación entre dos clases que puede llevar información asociada
- En este caso la asociación es una clase (es como una relación N:M con atributos del modelo E/R)
- Similar a los atributos en relaciones N:M



Un cliente compra muchos artículos y cada artículo es comprado por muchos clientes y, de cada compra se necesita saber la fecha y las unidades compradas

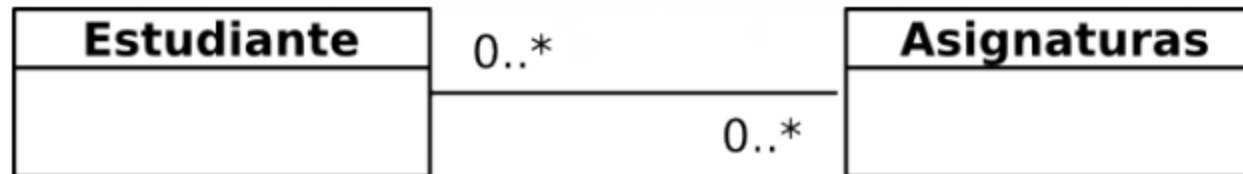
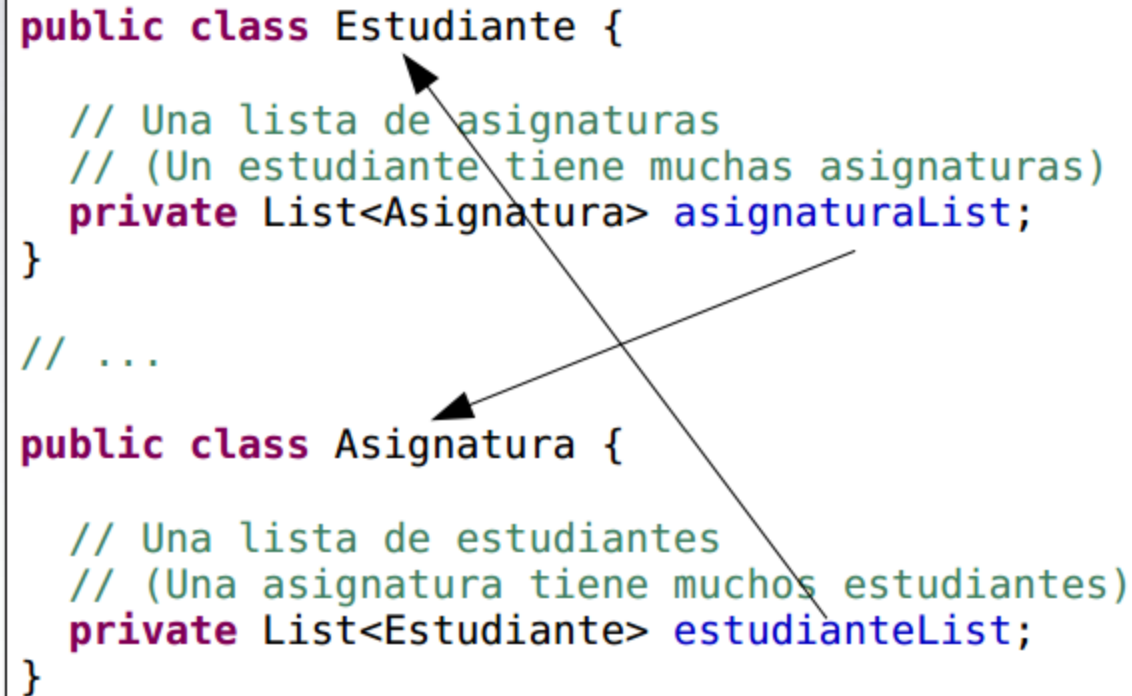
Ejemplos Asociaciones

```
public class Departamento {  
    // Una lista de profesores  
    // (Un departamento tiene muchos profesores)  
    private List<Profesor> profesorList;  
}  
  
// ...  
  
public class Profesor {  
    // Una referencia a un departamento  
    // (Un profesor pertenece sólo a un departamento)  
    private Departamento departamentoRef;  
}
```



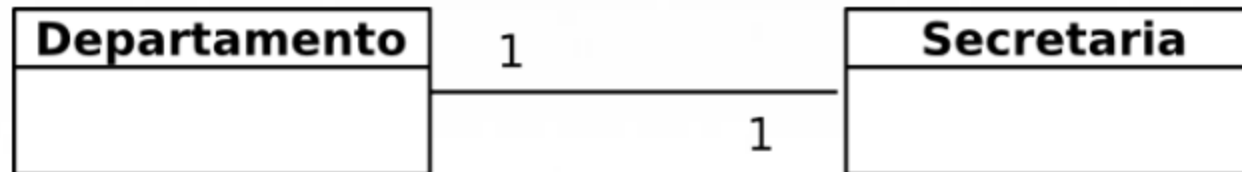
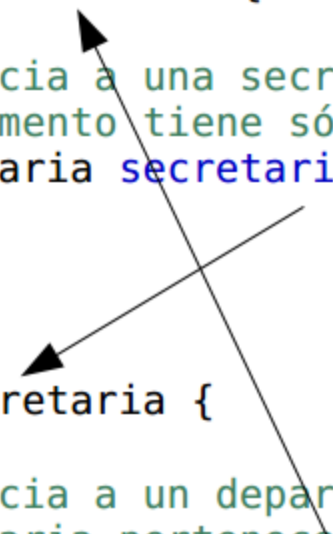
Ejemplos Asociaciones

```
public class Estudiante {  
    // Una lista de asignaturas  
    // (Un estudiante tiene muchas asignaturas)  
    private List<Asignatura> asignaturaList;  
}  
  
// ...  
  
public class Asignatura {  
    // Una lista de estudiantes  
    // (Una asignatura tiene muchos estudiantes)  
    private List<Estudiante> estudianteList;  
}
```



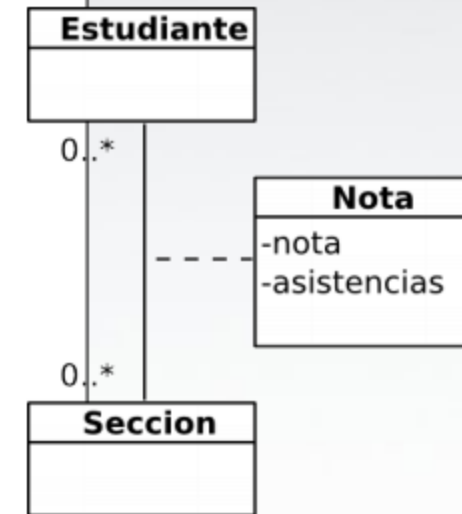
Ejemplos Asociaciones

```
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;  
}
```



Ejemplos Asociaciones

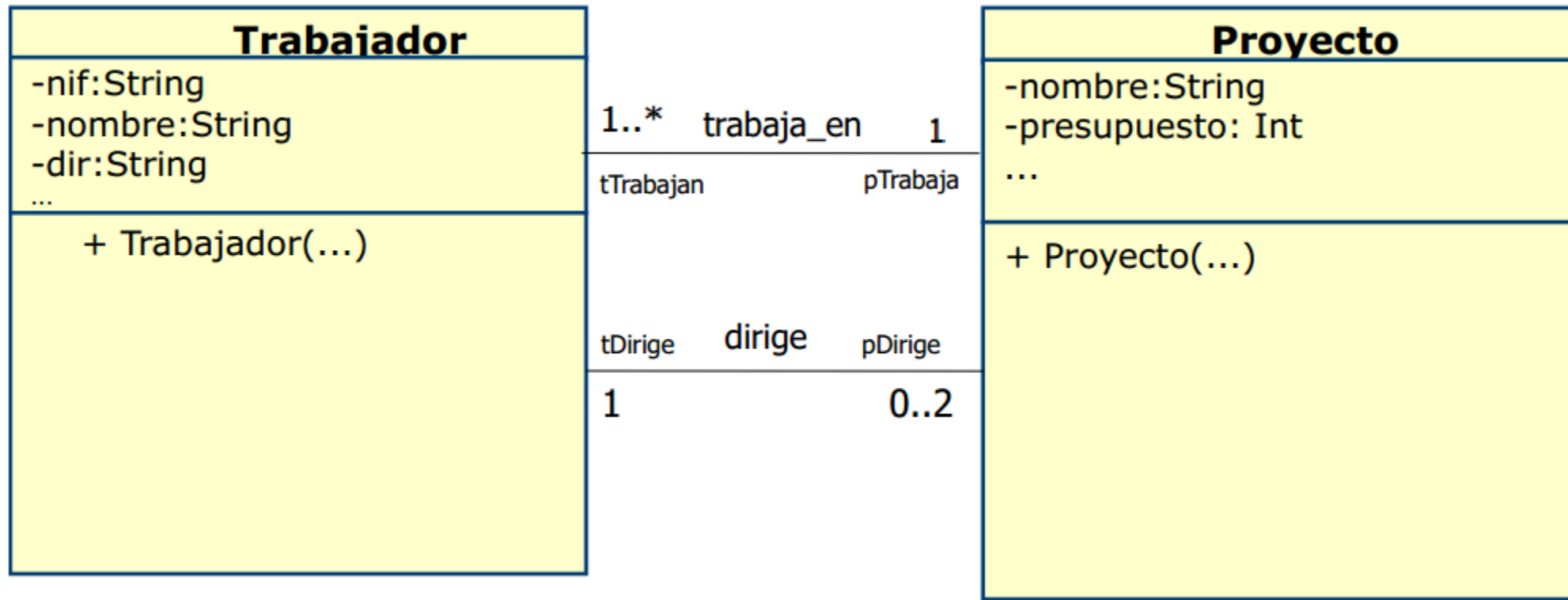
```
public class Estudiante {  
    // Una lista de Nota (Una clase asociación)  
    private List<Nota> notaList;  
}  
  
public class Nota {  
  
    // Datos de la asociación  
    private double nota;  
    private int asistencias  
  
    // referencias cruzadas a  
    // las dos clases relacionadas  
    private Estudiante estudianteRef;  
    private Seccion seccionRef;  
}  
  
public class Seccion {  
    // Una lista de Nota (Una clase asociación)  
    private List<Nota> notaList;  
}
```



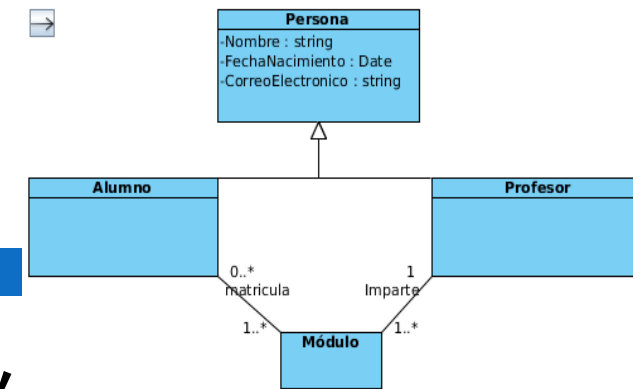
Ejercicio

A partir del dibujo de la Fig., define las clases Trabajador y Proyecto

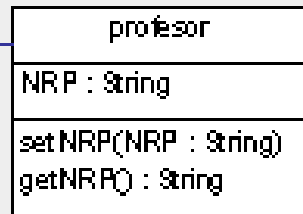
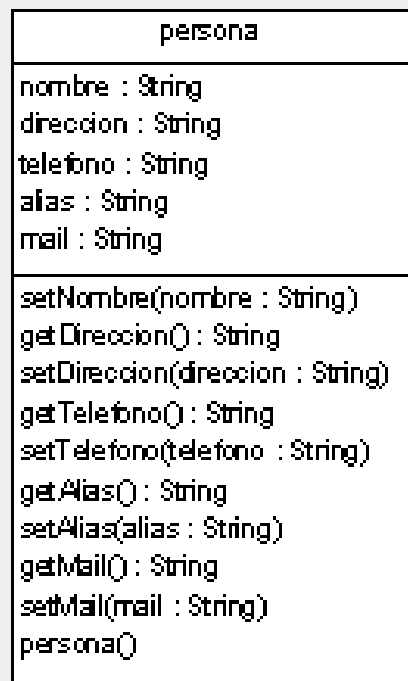
- Un trabajador debe trabajar siempre como mínimo en un proyecto, y dirigir un máximo de dos
- Un proyecto tiene n trabajadores, y siempre debe tener un director



Tipos de Relación. Herencia



- Es una abstracción para compartir similitudes entre clases, donde los atributos y operaciones comunes a varias clases se pueden compartir por medio de una superclase, más general.
- **Objetivo:** *reutilización de código*.
- Una **clase base** (**superclase** más generalizada) y una jerarquía de clases que contiene las **clases derivadas** (especializaciones).
- Tipos:
 - ▣ **Herencia simple:** Una clase puede tener sólo un ascendente.
 - ▣ **Herencia múltiple:** Una clase puede tener más de un ascendente inmediato.
- Representación:
 - ▣ En el diagrama de clases se representa como una asociación en la que el extremo de la clase base tiene un triángulo.

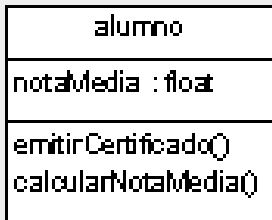


```
public class persona {
    private String nombre;
    private String direccion;
    private String telefono;
    private String alias;
    private String mail;
    public void setNombre(String nombre) { }
    public String getDireccion() { return null; }
    public void setDireccion(String direccion) { }
    public String getTelefono() { return null; }
    public void setTelefono(String telefono) { }
    public String getAlias() { return null; }
    public void setAlias(String alias) { }
    public String getMail() { return null; }
    public void setMail(String mail) { }
    public void persona() { }
}
```

```
public class alumno extends persona {

    private float notaMedia;
    public Vector matricula;
    public void emitirCertificado() { }
    public void calcularNotaMedia() { }
}
```

```
public class profesor extends persona {
    private String NRP;
    public void setNRP(String NRP) { }
    public String getNRP() { return null; }
}
```

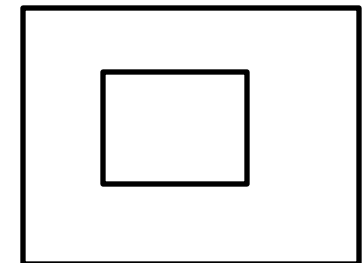


Relación Todo-parte

- Una relación **Todo-parte** es una relación en la que un objeto forma parte de la naturaleza del otro.
 - Un objeto puede estar compuesto por otros objetos.
 - A está compuesto de B, A tiene B.
- Se distinguen dos tipos de relación Todo-parte:
 - **Agregación:** relación débil
 - **Composición:** relación fuerte

Relación Todo-parte: Composición

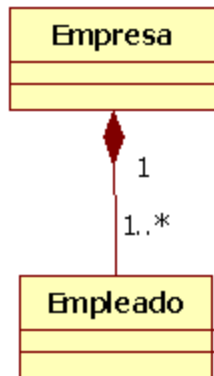
- Es una relación fuerte.
- Una instancia 'parte' está relacionada, como máximo, con una instancia 'todo' en un momento dado (un objeto solo puede estar contenido dentro de otro a la vez)
- Es decir, los objetos 'parte' forman parte del objeto 'todo', y no pueden ser compartidos por otros objetos 'todo'.
- La **cardinalidad** en la parte del 'todo' es 0...1 o 1. La copia o eliminación del objeto compuesto o 'todo' implica la eliminación de los objetos componentes o 'parte'.



Si el *Círculo* se copia o elimina, también lo hace el *Punto*

Relación Todo-parte: Composición

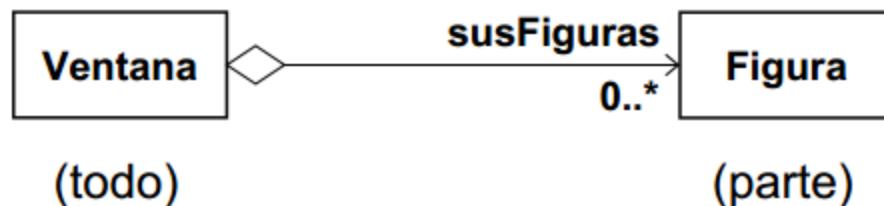
- Es un tipo de relación estática, en donde el tiempo de vida del objeto incluido está condicionado por el tiempo de vida del que lo incluye (el objeto base se construye a partir del objeto incluido, es decir, es parte/todo).



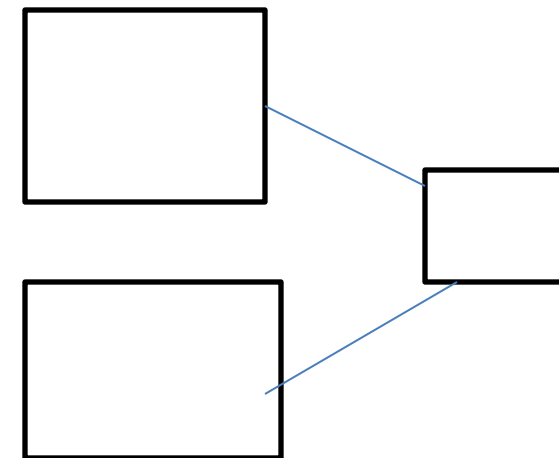
Un objeto empresa está compuesto a su vez por uno o varios objetos empleado.
El tiempo de vida de los objetos empleados depende del tiempo de vida del objeto empresa, ya que si no existe una empresa no pueden existir sus empleados.

Relación Todo-parte: Agregación

- La agregación es una relación débil.
- Es una asociación binaria que representa una relación entre un 'todo' y sus 'partes': 'pertenece a', 'tiene un', 'es parte de'.
- Los objetos 'parte' pueden ser compartidos por varios objetos 'todo'.
- La destrucción del objeto 'todo' **no** implica la destrucción de los objetos 'parte'.
- Es un tipo de relación dinámica, en donde el tiempo de vida del objeto incluido es independiente del que lo incluye.



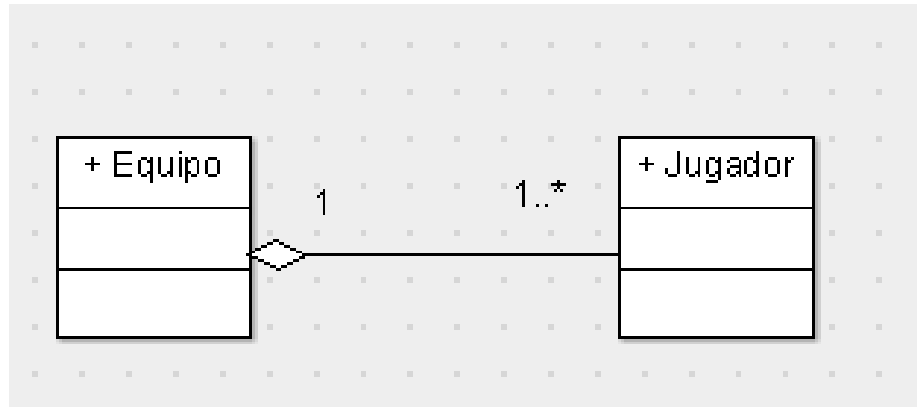
La *Ventana* contiene *Figuras*, pero cada una puede existir sin la otra



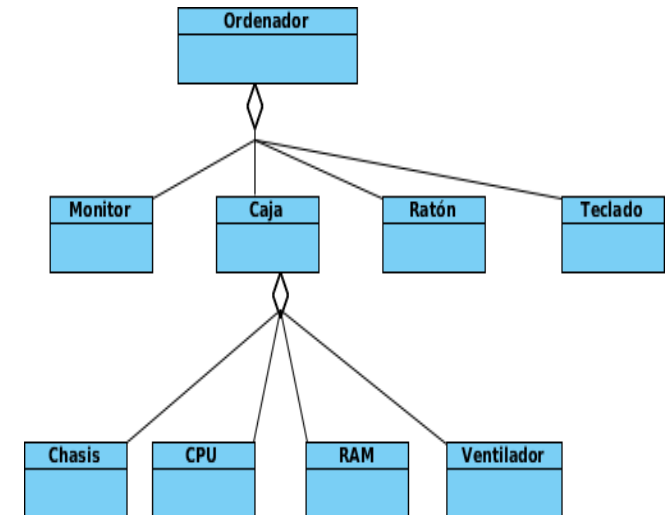
Relación Todo-parte: Agregación





- **Agregación:** se implementa como una asociación unidireccional
 - El objeto 'todo' mantiene referencias (quizás compartidas con otros objetos) a sus 'partes'.

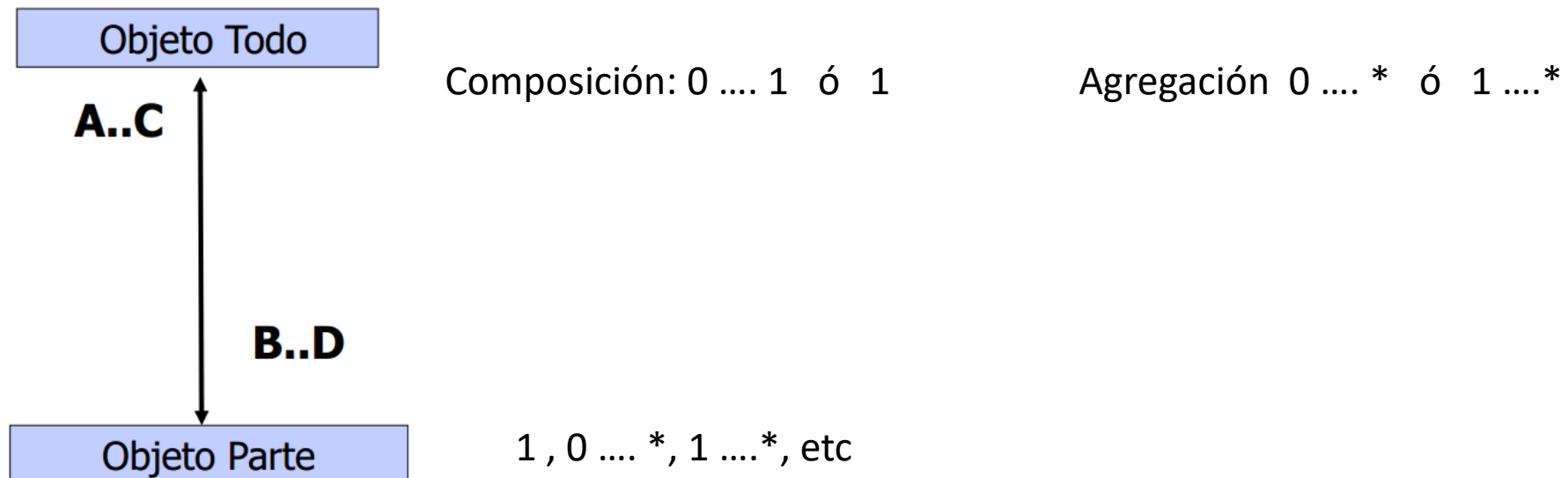


Un equipo está compuesto por jugadores, pero el jugador también puede jugar en otros equipos. Si desaparece el equipo, el jugador no desaparece

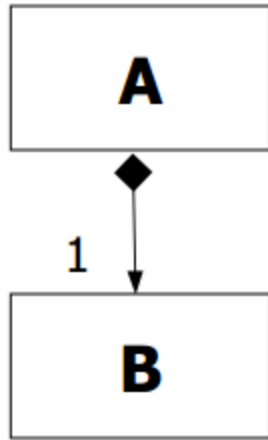


Diferencias entre agregación y composición

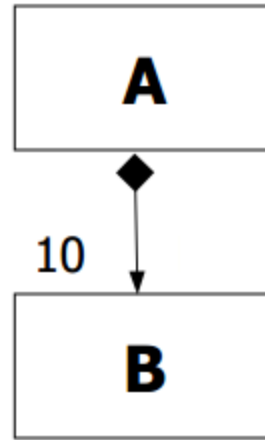
	Agregación	Composición
Símbolo		
Varias asociaciones comparten los componentes	SI	NO
Destrucción de los componentes al destruir el compuesto	NO	SI
Cardinalidad del compuesto	Cualquiera	0..1 o 1



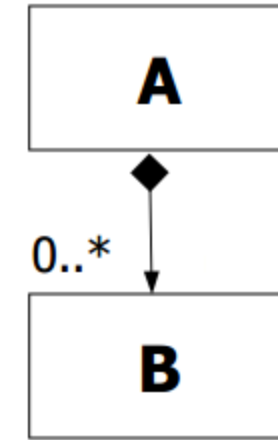
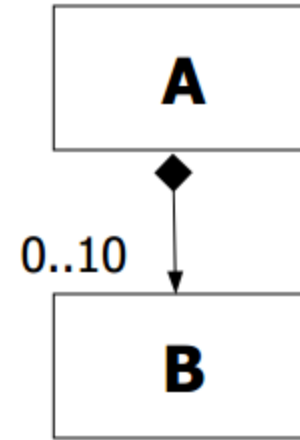
Relación Todo-parte: Implementación



```
class A {
    private B b;
    // b es un
    // subobjeto
    ...}
```



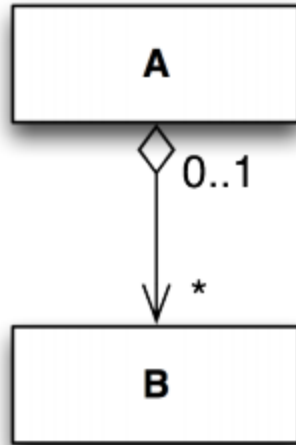
```
class A {
    private static final int MAX_B = 10;
    private B b[] = new B[MAX_B];
    ...}
```



```
class A {
    private
    Vector<B> b;
    ...};
```

La declaración de atributos es la misma para una agregación o una composición.

Relación Todo-parte: Implementación



A) El objeto B puede ser creado fuera de A, de forma que pueden existir referencias externas ('objB') al objeto agregado.

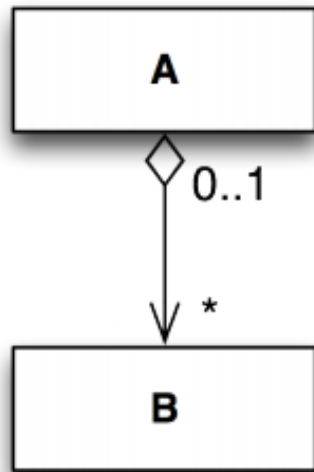
```
class A {
    private Vector<B> b = new Vector<B>();

    public A() {}
    public addB(B unB) {
        b.add(unB);
    }
    ...}
```

```
// En otro lugar (código cliente),
// quizás fuera de A...
B objB = new B();
if (...) {
    A objA = new A();
    objA.addB(objB);
}
```

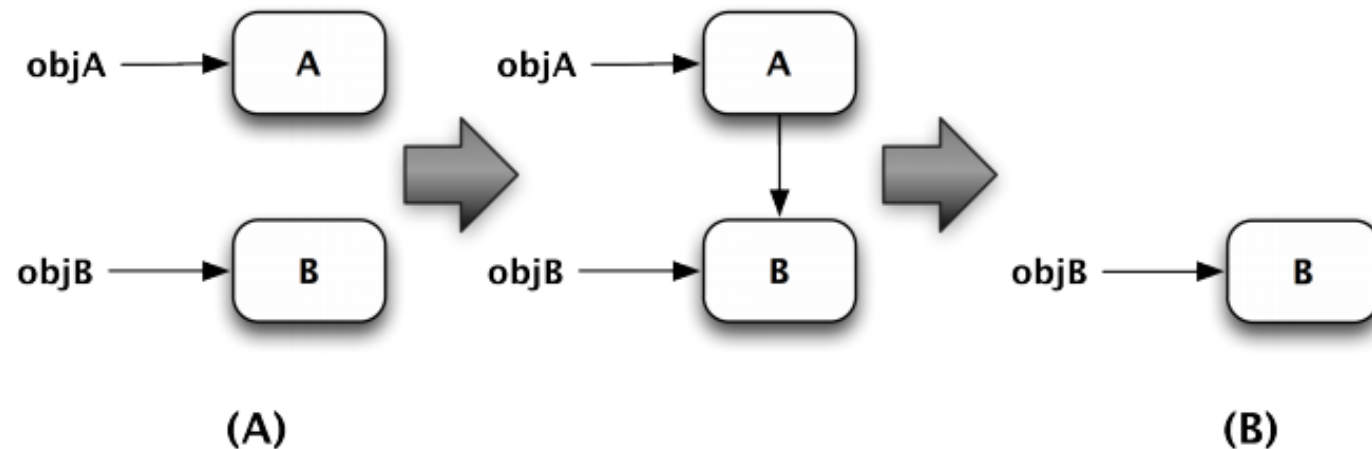
B) Cuando 'objA' desaparece, 'objB' sigue existiendo

Relación Todo-parte: Implementación

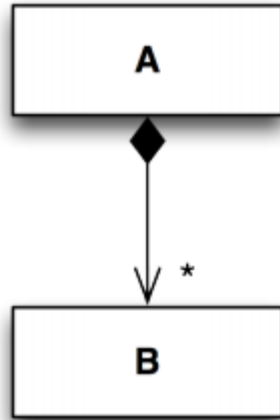


A) El objeto B puede ser creado fuera de A, de forma que pueden existir referencias externas ('objB') al objeto agregado.

B) Cuando 'objA' desaparece, el objeto B sigue existiendo, pues aún hay referencias a él ('objB').



Relación Todo-parte: Implementación



A) El objeto B es creado 'dentro' de A, de forma que A es el único que mantiene referencias a su componente B.

```
class A {
    private Vector<B> b = new Vector<B>();

    public A() {}
    public addB(...) {
        b.add(new B(...));
    } '...' es la información necesaria
    para crear B
}
```

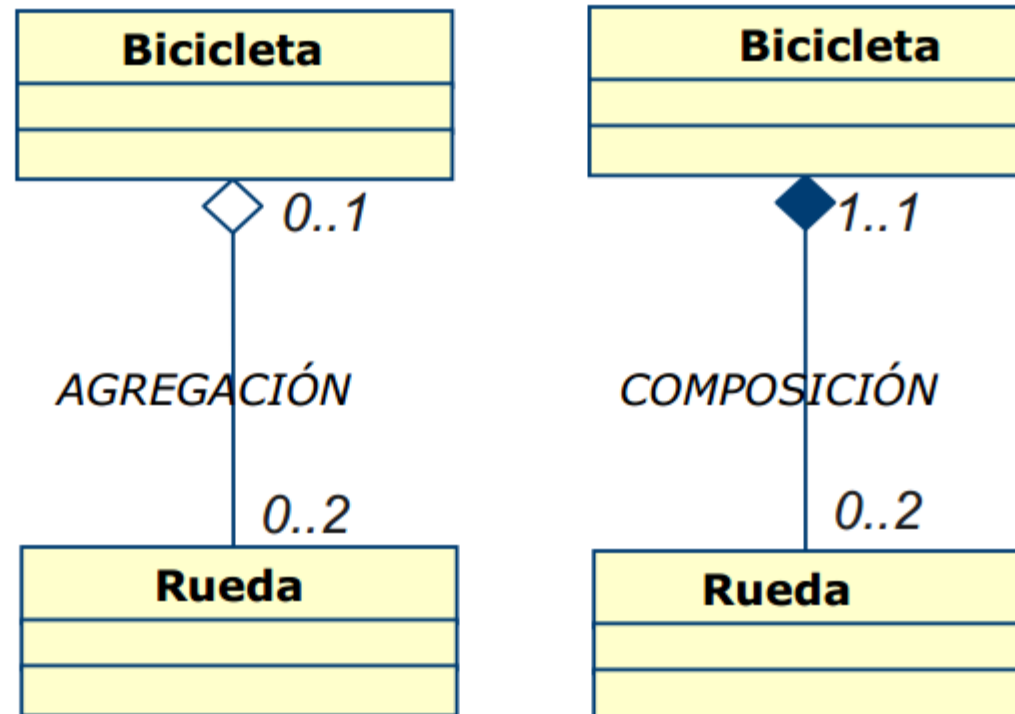
```
// En otro lugar (código cliente),
// fuera de A...
```

```
if (...) {
    A objA = new A();
    objA.addB(...);
}
```

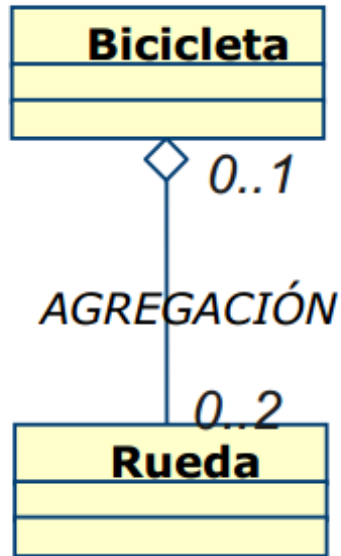
B) Cuando 'objA' desaparece, también desaparecen los objetos B que forman parte de él

Relación Todo-parte

Algunas relaciones pueden ser consideradas agregaciones o composiciones, en función del contexto en que se utilicen



Relación Todo-parte: Ejemplo bicicleta

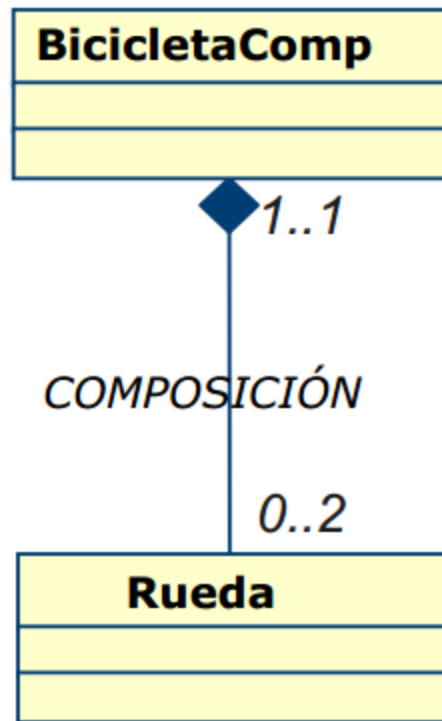


```
class Rueda {
    private String nombre;
    public Rueda(String n){nombre=n;}
}

class Bicicleta {
    private Vector<Rueda> r;
    private static final int MAXR=2;
    public Bicicleta(Rueda r1, Rueda r2){
        r.add(r1);
        r.add(r2);
    }
    public void cambiarRueda(int pos, Rueda raux){
        if (pos>=0 && pos<MAXR)
            r.set(pos,raux);
    }

    public static final void main(String[] args)
    {
        Rueda r1=new Rueda("1");
        Rueda r2=new Rueda("2");
        Rueda r3=new Rueda("3");
        Bicicleta b(r1,r2);
        b1.cambiarRueda(0,r3);
    }
}
```

Relación Todo-parte: Ejemplo bicicleta



```
class BicicletaComp{
    private static const int MAXR=2;
    private Vector<Rueda> r;
    public BicicletaComp(String p,String s){
        r.add(new Rueda(p));
        r.add(new Rueda(s));
    }
    public static final void main(String[] args) {
        BicicletaComp b2 = new BicicletaComp("1","2");
        BicicletaComp b3 = new BicicletaComp("1","2");

        //son ruedas distintas aunque con el mismo nombre
    }
}
```

Relación Todo-parte: Ejercicio 1

Supongamos que tenemos el código

```
class Planta {  
    public Planta(String n, String e)  
        {...}  
  
    public String getNombre() {...}  
    public String getEspecie() {...}  
    public String getTemporada() {...}  
    public void setTemporada(String t)  
        {...}  
  
    private String nombre;  
    private String especie;  
    private String temporada;  
}
```

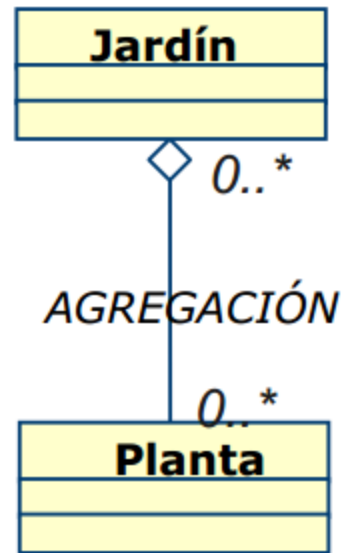
```
class Jardin {  
    public Jardin(String e) {...}  
  
    public Jardin clone() {...}  
    public void plantar(String n, String e,  
        String t)  
    {  
        Planta lp = new Planta(n, e);  
        lp.setTemporada(t);  
        p.add(lp);  
    }  
  
    private Vector<Planta> p  
        = new Vector<Planta>();  
    private String emplazamiento;  
}
```



¿Qué relación existe entre Jardín y planta?

Relación Todo-parte: Ejercicio 2

¿Cómo cambiaría el código si decidiésemos implementar el jardín como una agregación de plantas?

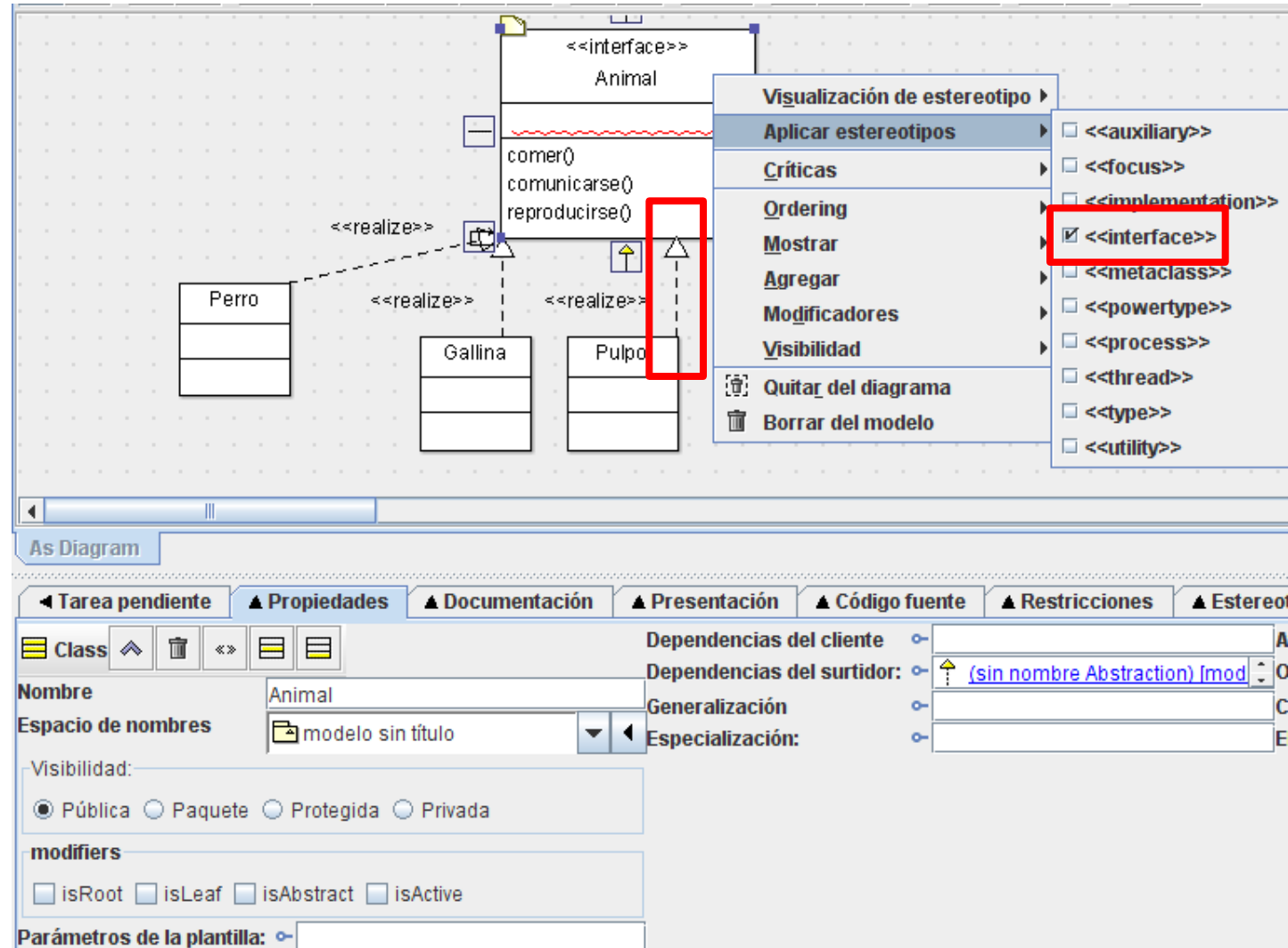
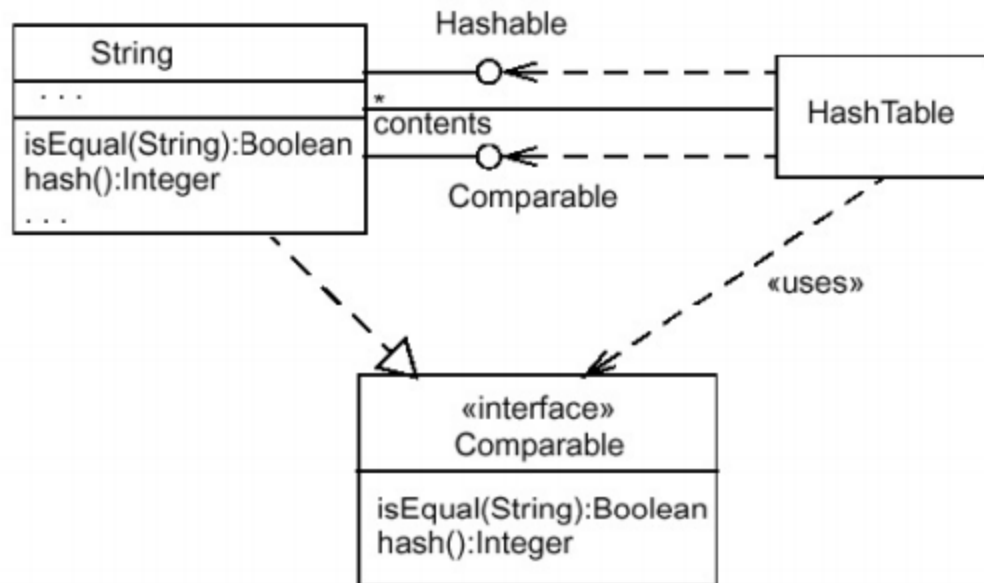


Tipo de relaciones: Realización

- ❑ Relación de herencia entre una clase INTERFACE y la subclase que implementa esa interface.
- ❑ Una **Interface** es una colección de operaciones que especifican un servicio de una clase. Es una clase abstracta
- ❑ Separa especificación e implementación de una abstracción.
- ❑ Incluyen:
 - ❑ Nombre
 - ❑ Operaciones sin implementación (métodos abstractos) y constantes
 - ❑ Relaciones de realización
 - ❑ Pueden tener relaciones de generalización.
- ❑ No incluyen:
 - ❑ Atributos
 - ❑ Asociaciones

Tipo de relaciones: Realización

- Clase con estereotipo de interface



Tipo de relaciones: Realización

- Una Interfaz es una clase totalmente abstracta, es decir, no tiene atributos y todos sus métodos son abstractos y públicos, sin desarrollar.

```
public interface Animal{  
    public void comer();  
    public void comunicarse();  
  
}
```

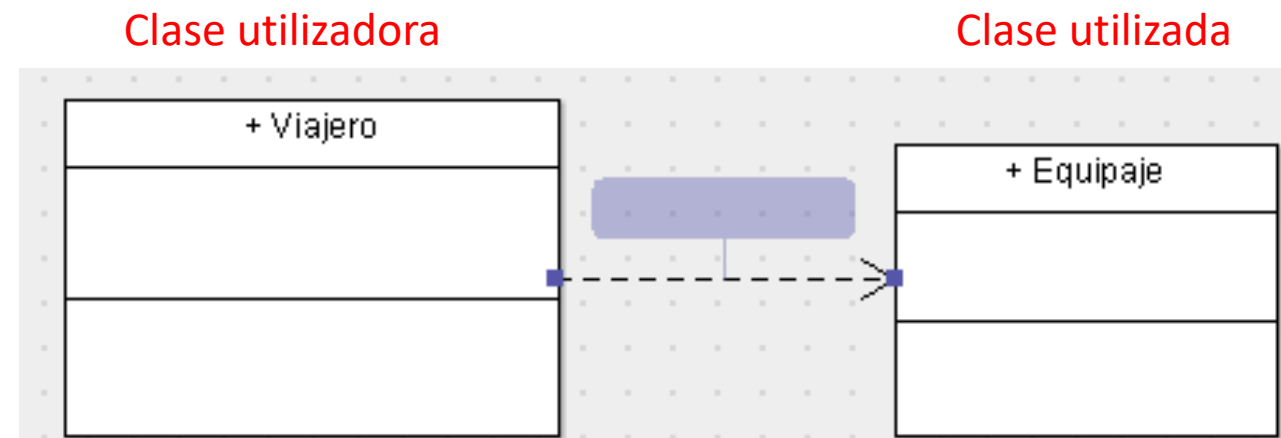
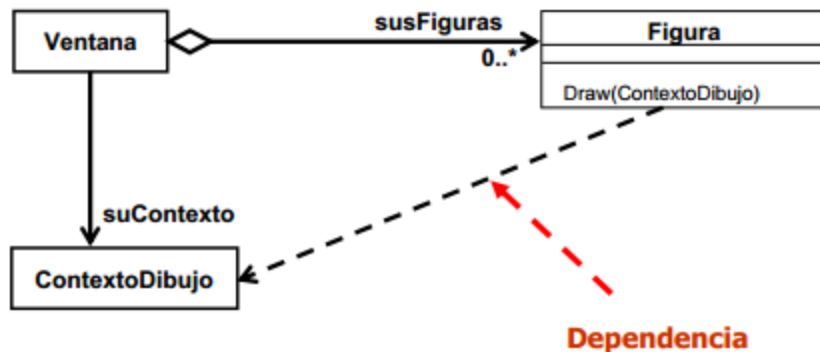
```
public class Pulpo implements Animal{  
    public void comer() { . . . }  
    public void comunicarse { . . . }  
}
```

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

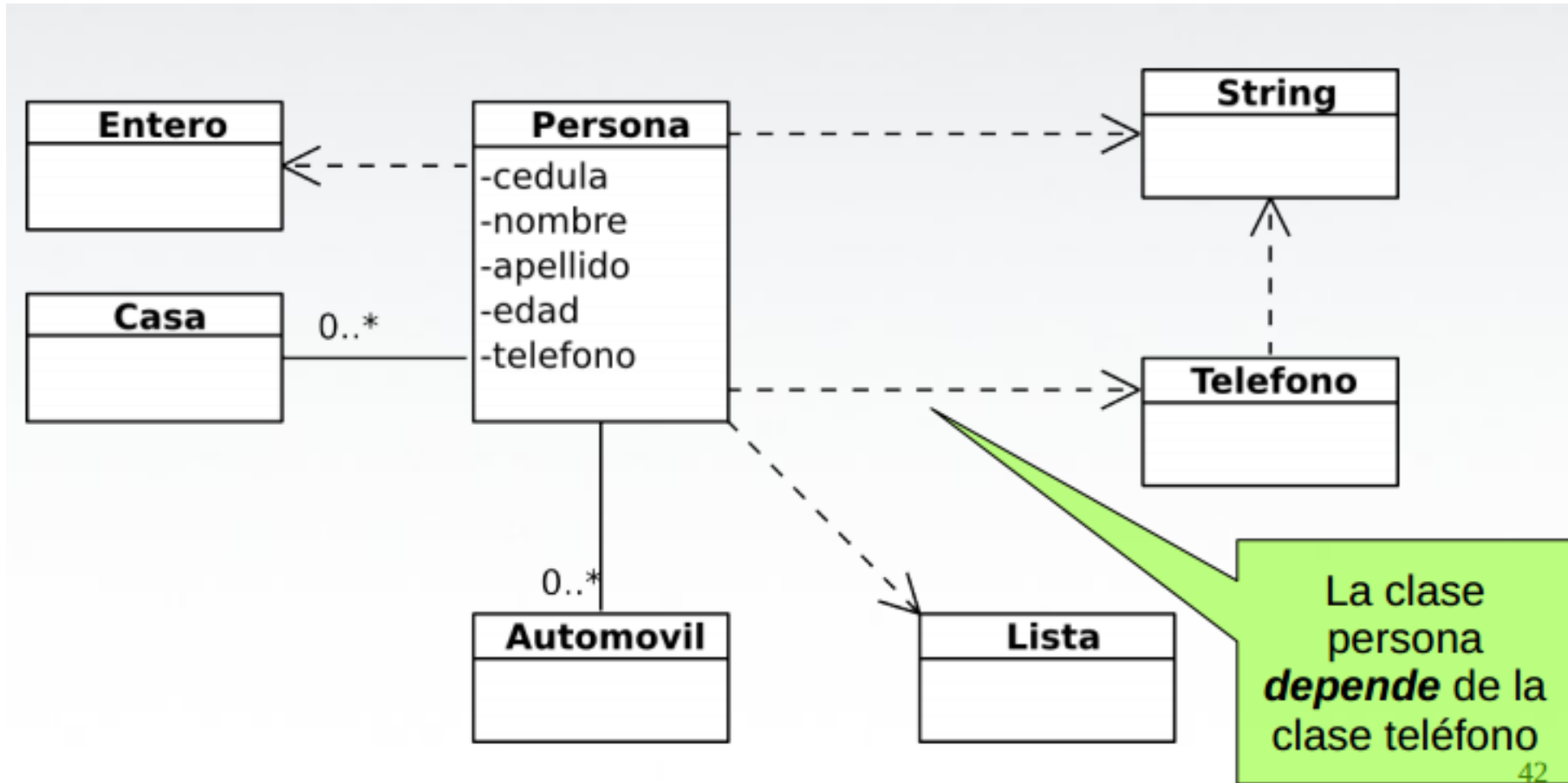
```
public class Gallina implements Animal{  
    public void comer() { . . . }  
    public void comunicarse { . . . }  
}
```


Tipos de relaciones: Dependencia

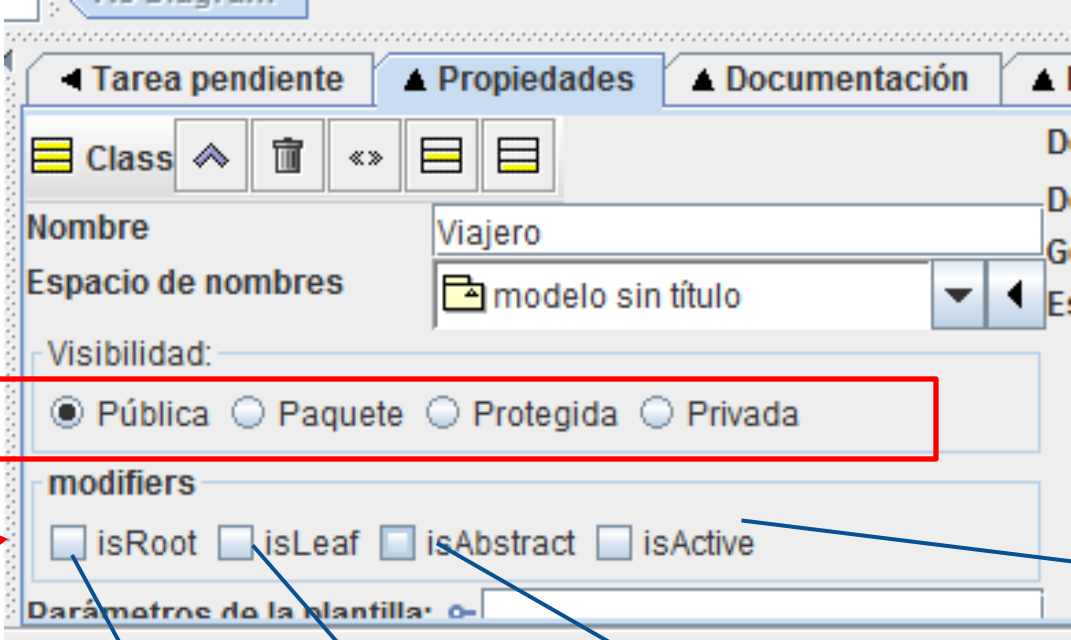
- ❑ Relación débil entre dos clases en la que una usa la otra, es decir, que la necesita para su cometido.
- ❑ Un cambio en la clase utilizada afecta al funcionamiento de la clase utilizadora, pero no al revés.
- ❑ Ejemplo. La clase Impresora imprime Documentos (la clase Impresora usa la clase Documento). Un Viajero necesita un Equipaje para viajar



Tipos de relaciones: Dependencia



Modificadores



The screenshot shows the 'Propiedades' (Properties) tab of a UML class editor. The class name is 'Viajero' and it belongs to the package 'modelo sin título'. The 'Visibilidad' (Visibility) section has four radio buttons: 'Pública' (selected), 'Paquete', 'Protegida', and 'Privada'. The 'modifiers' section has four checkboxes: 'isRoot', 'isLeaf', 'isAbstract', and 'isActive'. Red arrows point from the text 'Visibilidad de la clase' to the visibility section and from 'Modificadores' to the modifiers section. Blue boxes with labels are connected to the checkboxes: 'Clase raíz' to 'isRoot', 'Clase hoja' to 'isLeaf', 'Clase abstracta' to 'isAbstract', and 'Clase activa' to 'isActive'. The 'Clase activa' box contains a definition of an active class.

Visibilidad de la clase

Modificadores

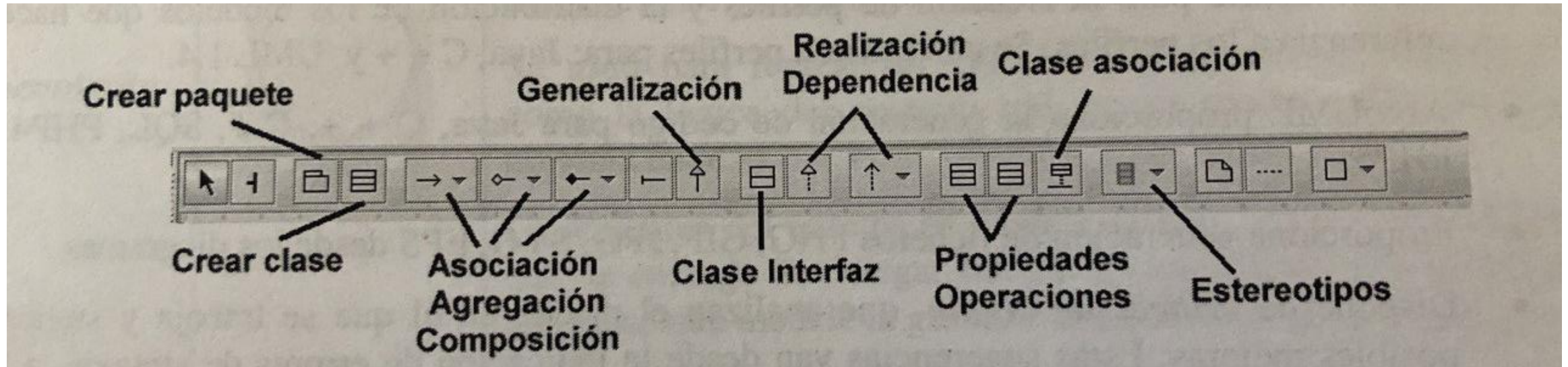
Clase raíz

Clase hoja

Clase abstracta

Clase activa:
clase que posee
uno o más
procesos o hilos
y pueden iniciar
una actividad de
control,

Barra de herramientas para la creación de un diagrama de clases

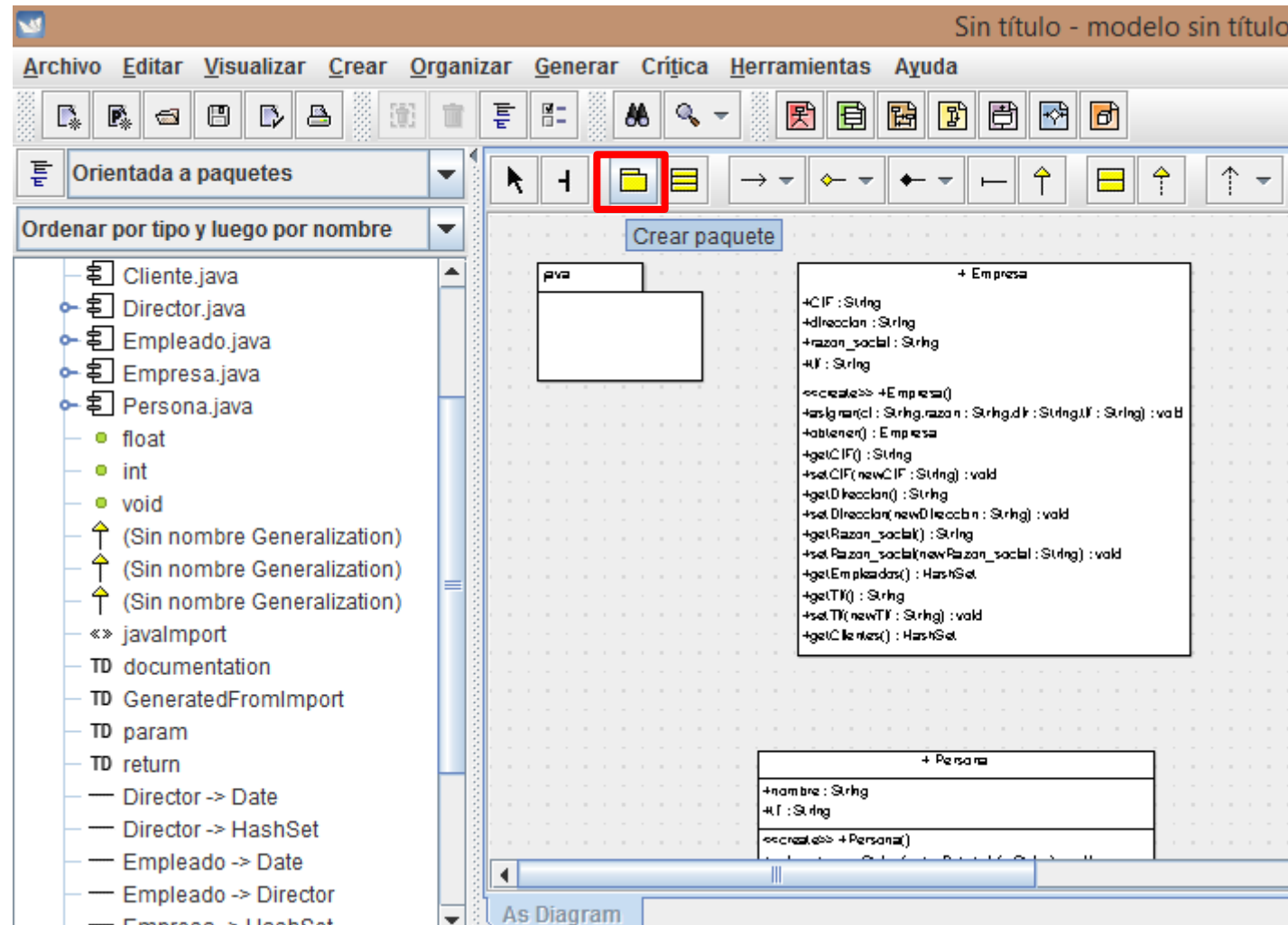


Ejercicios

Hacer el ejercicio 1 de la Tarea 01 UT03

- Se trata de realizar un diagrama de clases para representar las relaciones entre empleados y departamentos.
 - ▣ Consideramos que un empleado trabaja en un departamento y en el departamento trabajan muchos empleados
 - ▣ Datos de los empleados son código, nombre, oficio y salario
 - ▣ Datos de los departamentos son código, nombre y localidad
 - ▣ Además, un empleado puede ser jefe de varios empleados
 - ▣ Se necesita crear los métodos para asignar datos a los empleados y departamentos y devolverlos (setter and getter).

Creación de paquetes



Documentación

- Si se está creando el diseño, y no se cumple con las reglas establecidas, aparecerán las críticas y las sugerencias en el apartado de **críticas** (clic en las hojas de las clases) que nos ayudarán a corregir fallos.
- Para borrar, no solo de la vista, hay que borrar del modelo (Botón Papelera) de la barra de herramientas.

Paso de los requisitos de un sistema al diagrama de clases.

- Los sustantivos nos revelarán las clases del modelo y sus propiedades
- Los adjetivos nos revelan valores concretos de atributos
- Los verbos nos indicarán los métodos a definir
- Pasos:

- **Nombres:** Clases y sus propiedades
- **Adjetivos:** Valores de las propiedades
- **Verbos:** Comportamiento de las clases (métodos)

"El **coche** tiene **color rojo** y se **mueve**"

"El **documento** tiene **letra grande** y se **muestra**"

1. Identificar clases
2. Identificar y depurar relaciones
3. Identificar atributos de clases y relaciones
4. Añadir herencia
5. Comprobar casos de uso (iterar)
6. Añadir y simplificar métodos

Paso de los requisitos de un sistema al diagrama de clases II.

- Identificamos objetos que serán las clases del diagrama examinando el planteamiento del problema:
 - Subrayar cada nombre o cláusula nominal.
 - Los sinónimos deben destacarse.
 - **Se incluyen en una tabla**
 - Una vez aislados todos los nombres, buscamos **sustantivos** que se correspondan con las siguientes categorías:
 - **Entidades externas** que producen o consumen información a usar por un sistema computacional.
 - **Cosas** como informes, presentaciones, cartas, señales que son parte del dominio de información del problema.
 - **Ocurrencias o sucesos** que ocurren dentro del contexto de una operación del sistema.
 - **Papeles o roles** desempeñados por personas que interactúan con el sistema.
 - **Unidades organizacionales** que son relevantes en una aplicación.
 - **Lugares** que establecen el contexto del problema y la función general del sistema.
 - **Estructuras** que definen una clase de objetos o, en casos extremos, clases relacionadas de objetos.
- No incluir en la lista cosas que no sean objetos, como operaciones aplicadas a otro objeto.

Paso de los requisitos de un sistema al diagrama de clases III.

- Se incluyen también los posibles **atributos** que aparezcan
- Decidir si los elementos de la tabla se incluyen en el diagrama como objetos, para ello seguiremos los **Criterios**:
 1. La información del objeto es necesaria para que el sistema funcione.
 2. El objeto posee un **conjunto de atributos**. Si sólo aparece un atributo normalmente se rechazará y será añadido como atributo de otro objeto.
 3. El objeto tiene un **conjunto de operaciones**.
 4. Es una entidad externa que consume o produce información esencial para la producción de cualquier solución en el sistema.
- El objeto se incluye si cumple todos o casi todos los criterios.

Se debe tener en cuenta que la lista no incluye todo, habrá que añadir objetos adicionales para completar el modelo y también, que diferentes descripciones del problema pueden provocar la toma de diferentes decisiones de creación de objetos y atributos.

Obtención de atributos y operaciones.

□ **Atributos**

- Definen al objeto en el contexto del sistema
- *Deben contestar a la pregunta: ¿Qué elementos (compuestos y/o simples) definen completamente al objeto en el contexto del problema actual?*

□ **Operaciones**

- Describen el comportamiento del objeto y modifican sus características:
 - Manipulan los datos.
 - Realizan algún cálculo.
 - Monitorizan un objeto frente a la ocurrencia de un suceso de control.
- Se obtienen analizando verbos en el enunciado del problema.

Obtención de atributos y operaciones

□ Relaciones

- ▣ Se revisa de nuevo el enunciado para extraer las relaciones
- ▣ Buscar mensajes que se pasen entre objetos y las relaciones de composición y agregación.
- ▣ Las relaciones de herencia se suelen encontrar al comparar objetos semejantes entre sí, y constatar que tengan atributos y métodos comunes.
- Revisar el diagrama obtenido y ver si todo cumple con las especificaciones.
- Refinar el diagrama con aspectos obtenidos a través de entrevistas con los clientes o a nuestros conocimientos de la materia.

Ejercicio resuelto

Representa mediante un **diagrama de clases** la siguiente especificación:

- Una aplicación necesita almacenar información sobre empresas, sus empleados y sus clientes
- Empleados y clientes se caracterizan por su nombre y edad
- Los empleados tienen un sueldo bruto
- Los empleados que son directivos tienen una categoría y un conjunto de empleados subordinados
- De los clientes se necesita conocer su teléfono de contacto
- La aplicación necesita mostrar los datos de empleados y clientes

Ejercicio resuelto

- Una aplicación necesita almacenar información sobre empresas, sus empleados y sus clientes
- Empleados y clientes se caracterizan por su nombre y edad
- Los empleados tienen un sueldo bruto
- Los empleados que son directivos tienen una categoría y un conjunto de empleados subordinados
- De los clientes se necesita conocer su teléfono de contacto
- La aplicación necesita mostrar los datos de empleados y clientes

Ejercicio resuelto

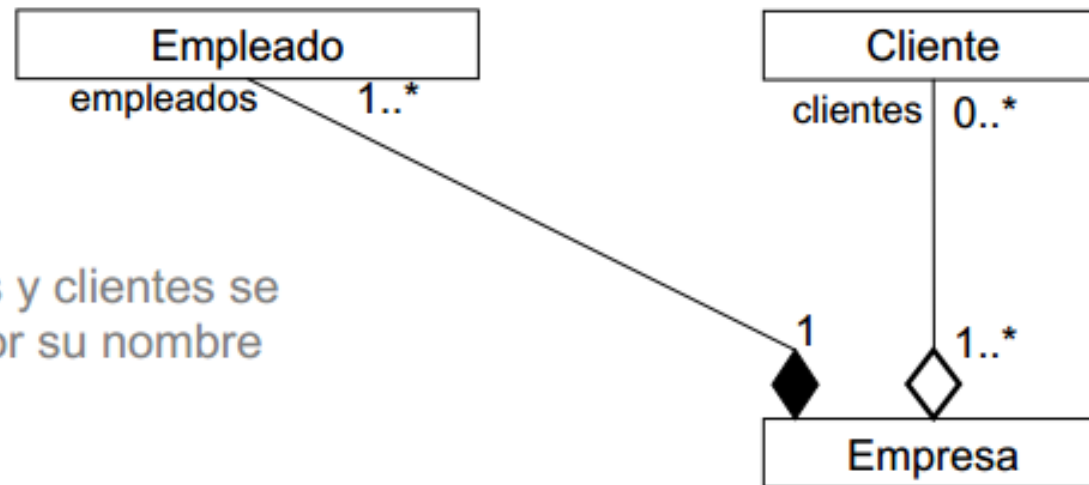
Clase/objeto	Categoría	Clase/objeto	Categoría
Empresas	Entidad externa	categoría	Atributo
Cientes	Entidad externa	Teléfono contacto	Atributo
empleados	Entidad externa		
Nombre	Atributo		
edad	Atributo		
Sueldo bruto	Atributo		
directivo	Rol/Entidad Externa		

Ejercicio resuelto

R1: una aplicación necesita almacenar información sobre empresas, sus empleados y sus clientes

R2: empleados y clientes se caracterizan por su nombre y edad

R3: los empleados tienen un sueldo bruto

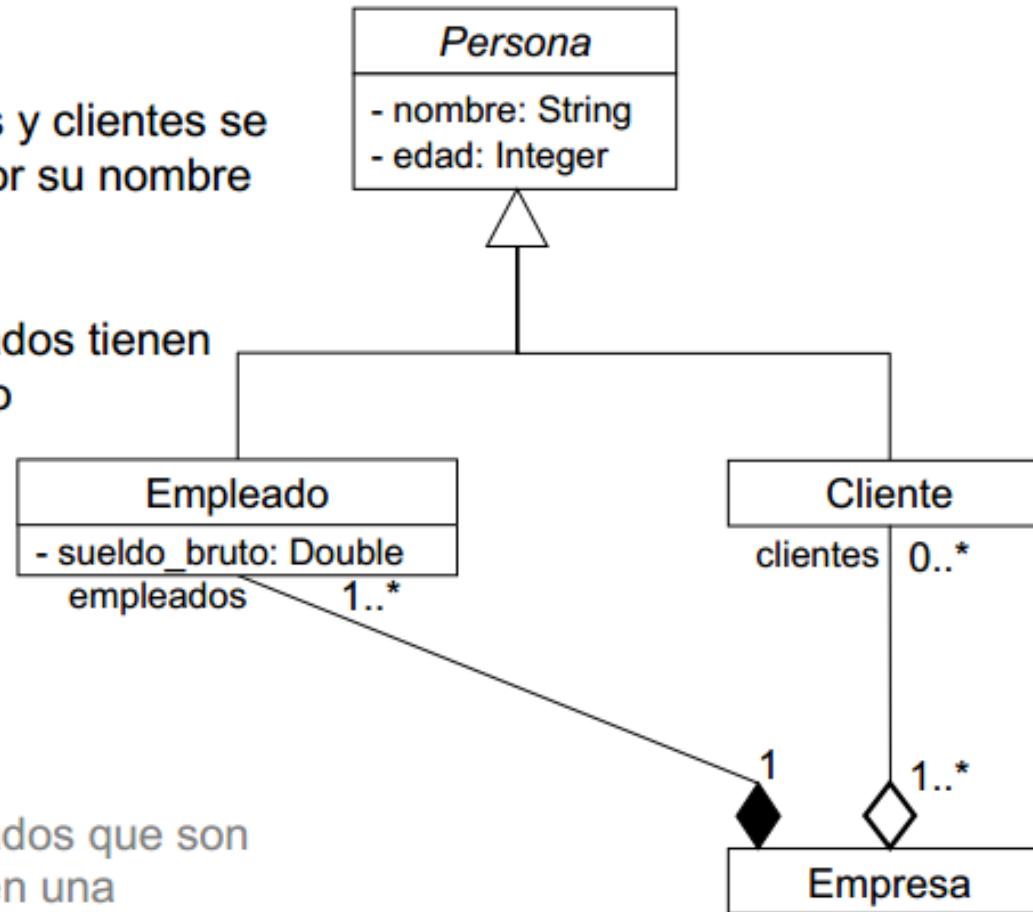


Ejercicio resuelto

R2: empleados y clientes se caracterizan por su nombre y edad

R3: los empleados tienen un sueldo bruto

R4: los empleados que son directivos tienen una categoría y un conjunto de empleados subordinados

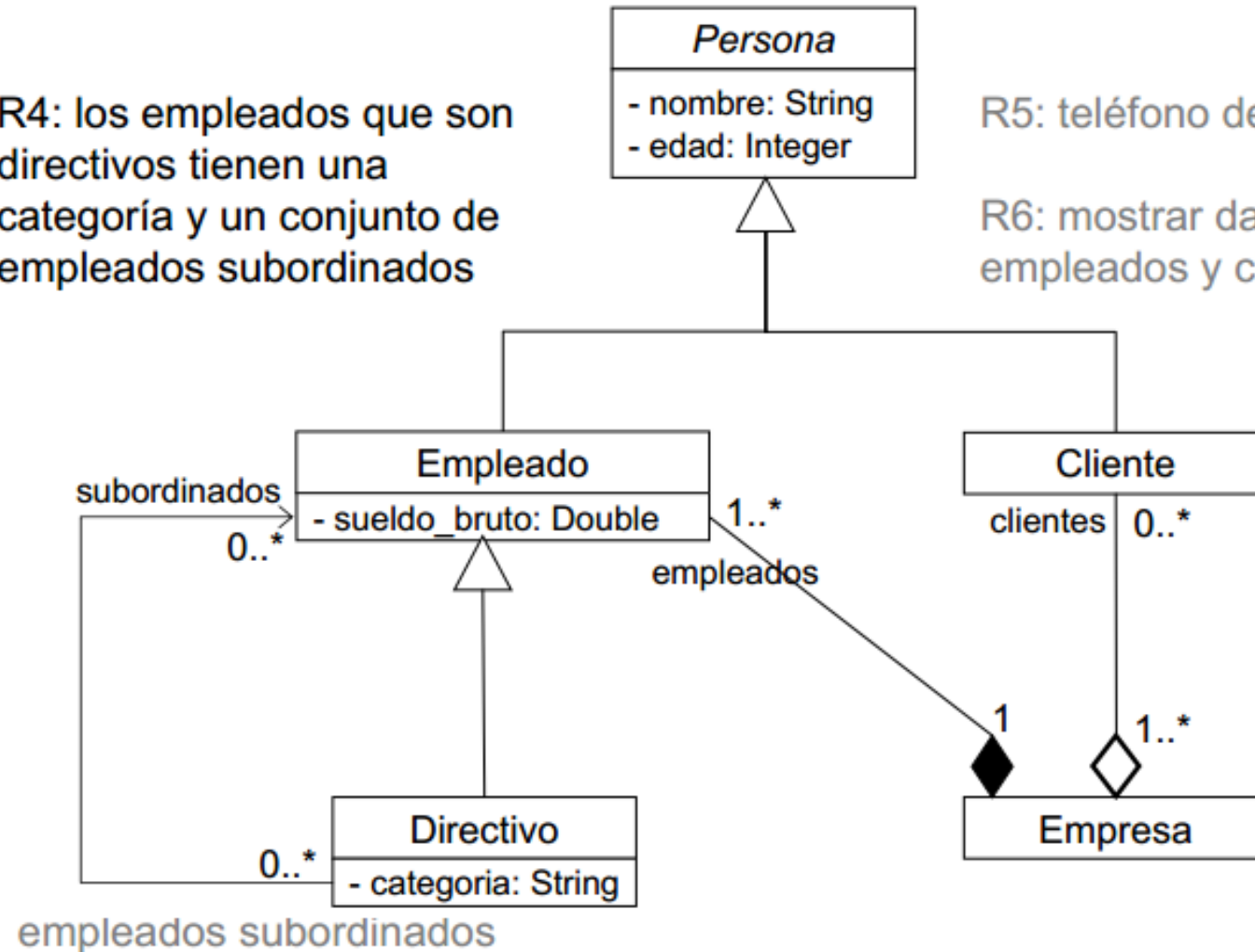


Ejercicio resuelto

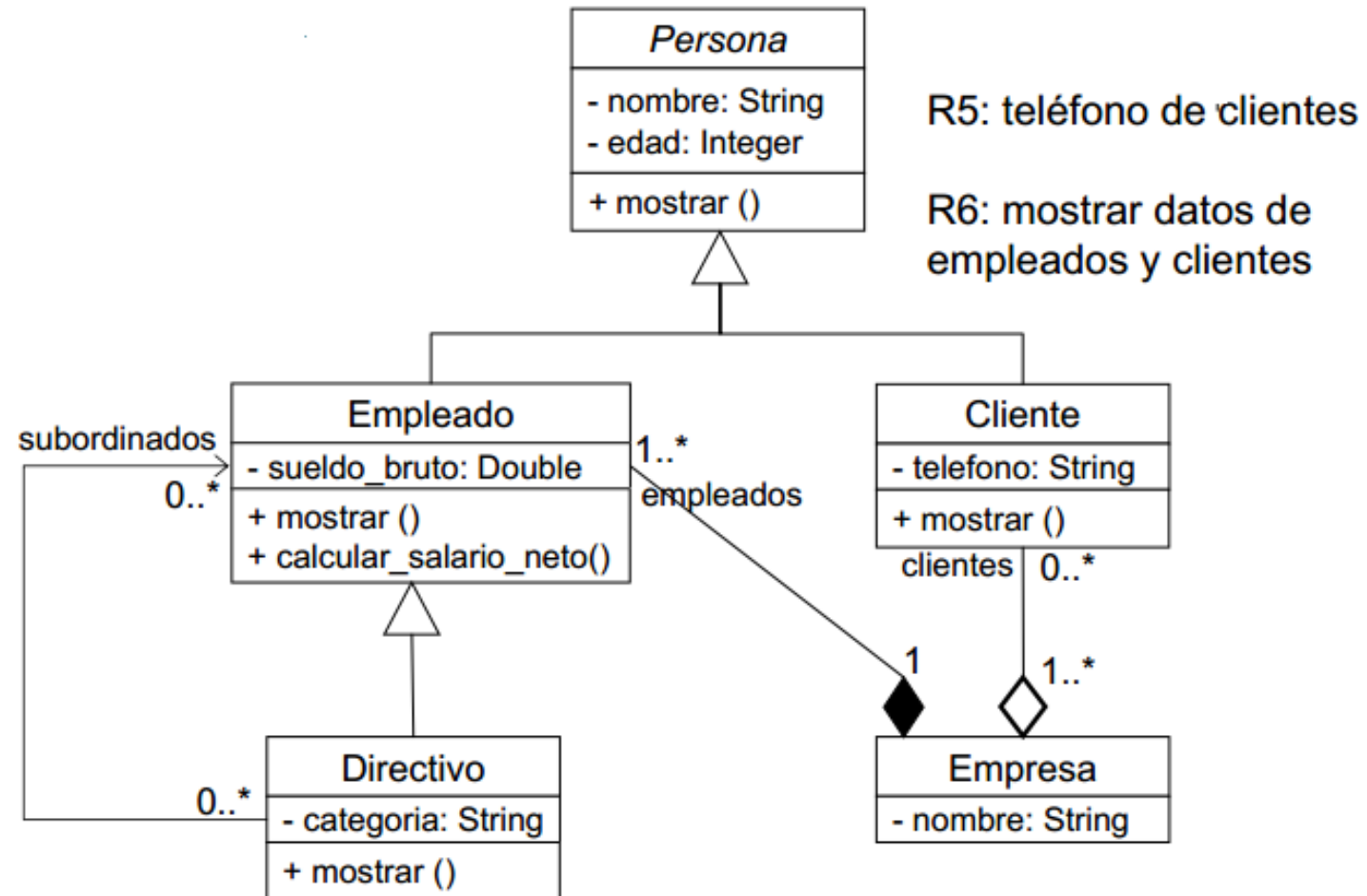
R4: los empleados que son directivos tienen una categoría y un conjunto de empleados subordinados

R5: teléfono de clientes

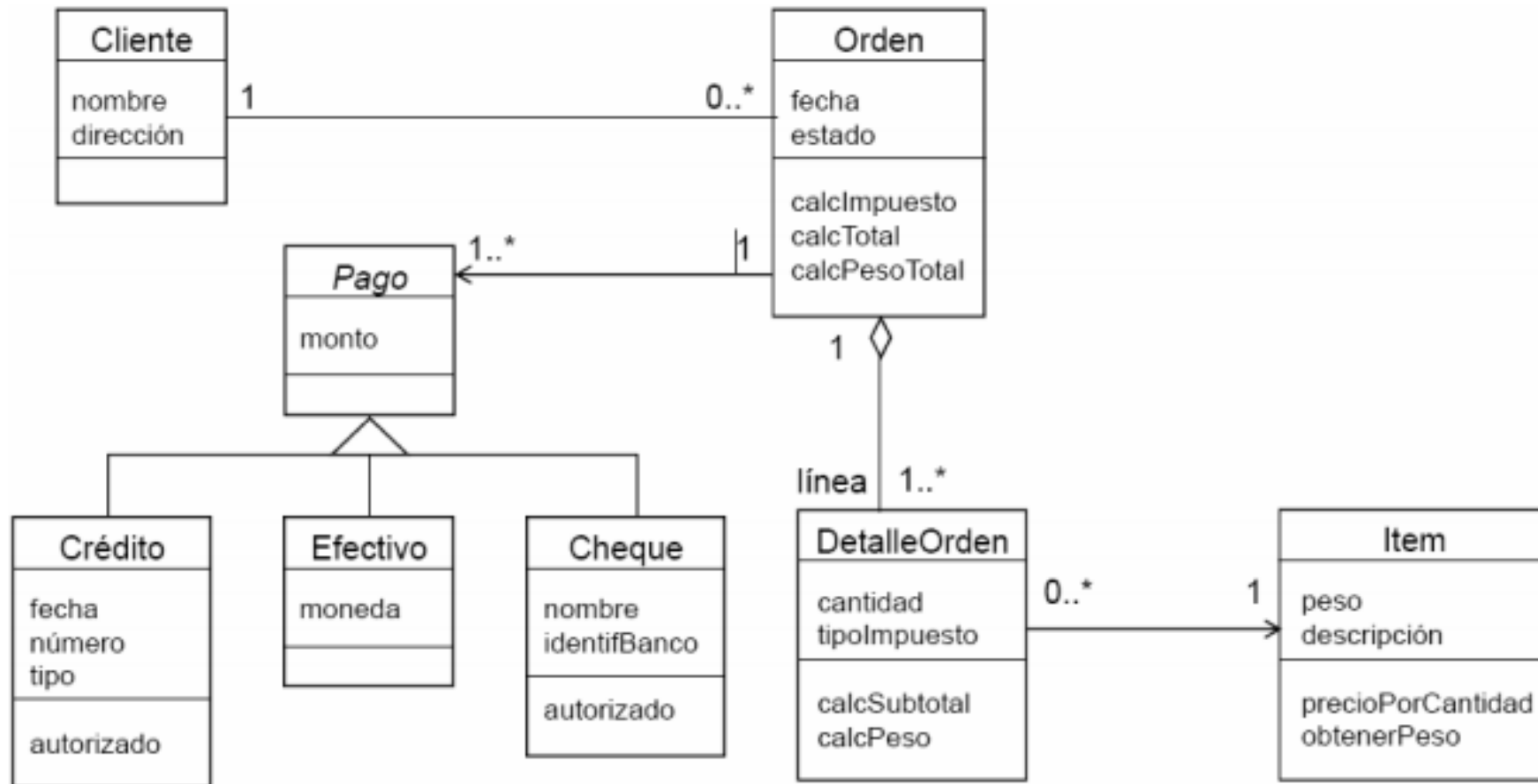
R6: mostrar datos de empleados y clientes



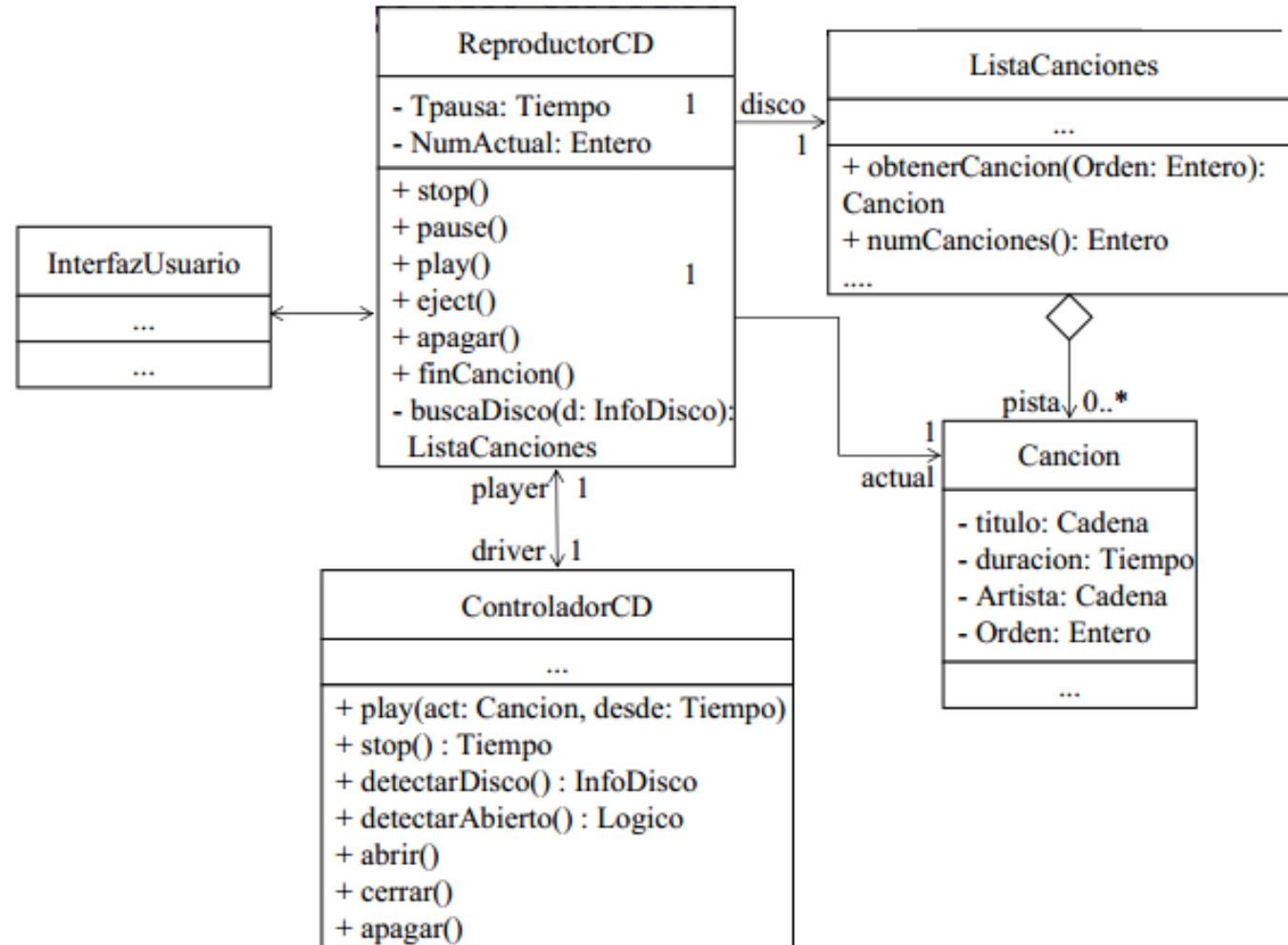
Ejercicio resuelto



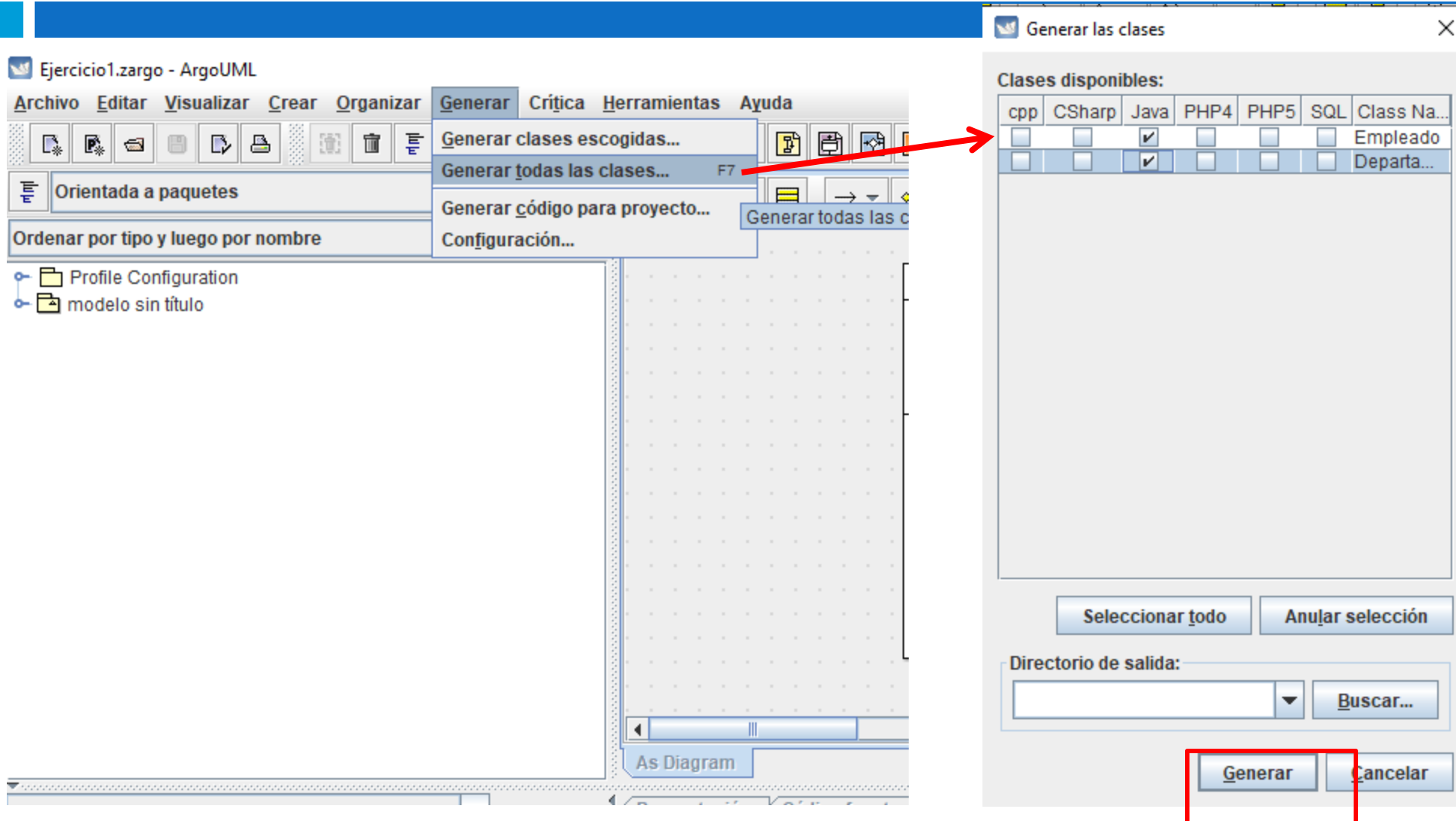
Otros ejemplos



Otros ejemplos

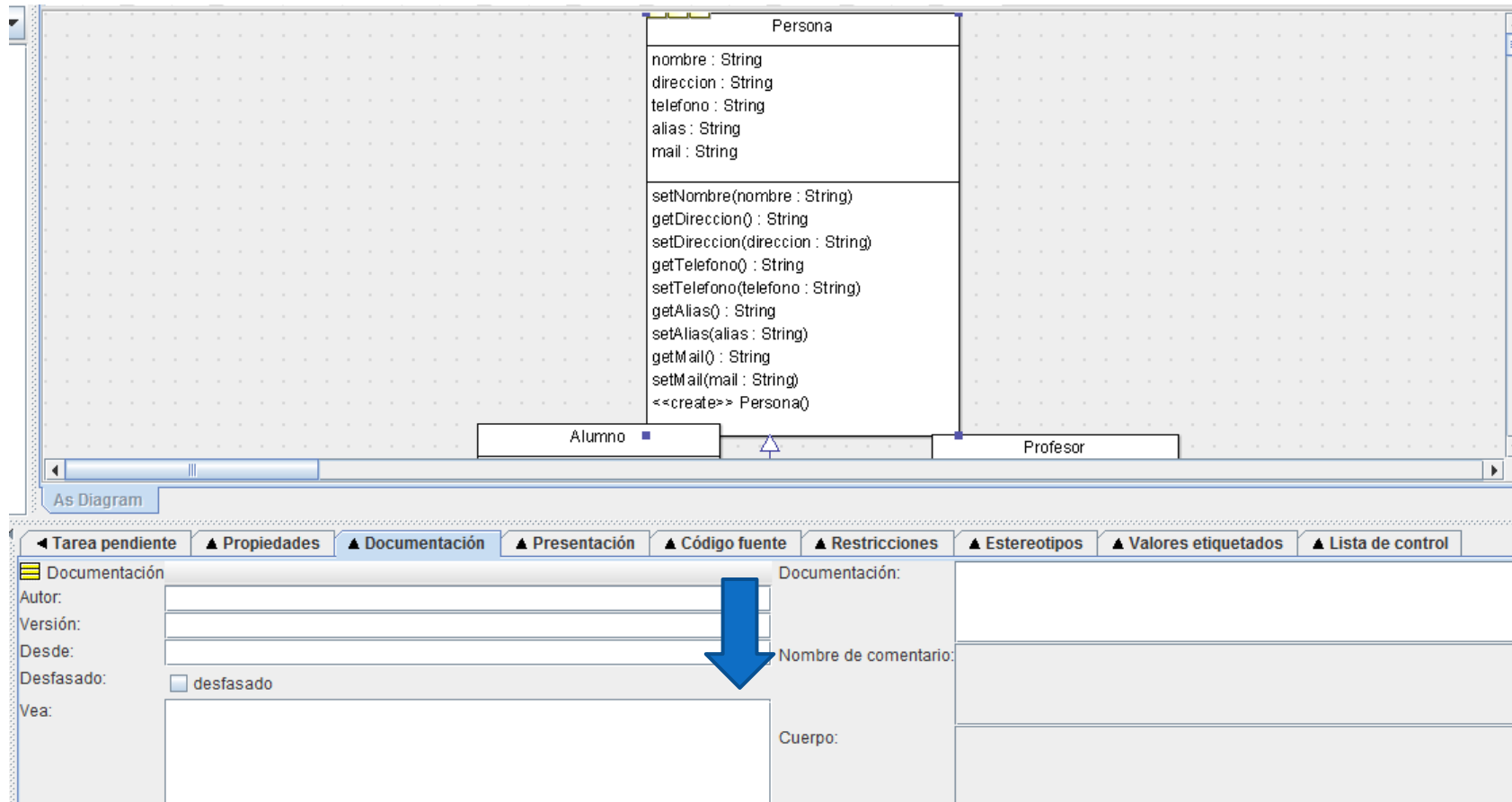


Generación de código a partir el diagrama de clases



[Vídeo explicativo](#)

Generar documentación en ArgoUML



Podemos hacer anotaciones abriendo la especificación de cualquiera de los elementos, clases o relaciones, o bien del diagrama en la pestaña “Documentación”.

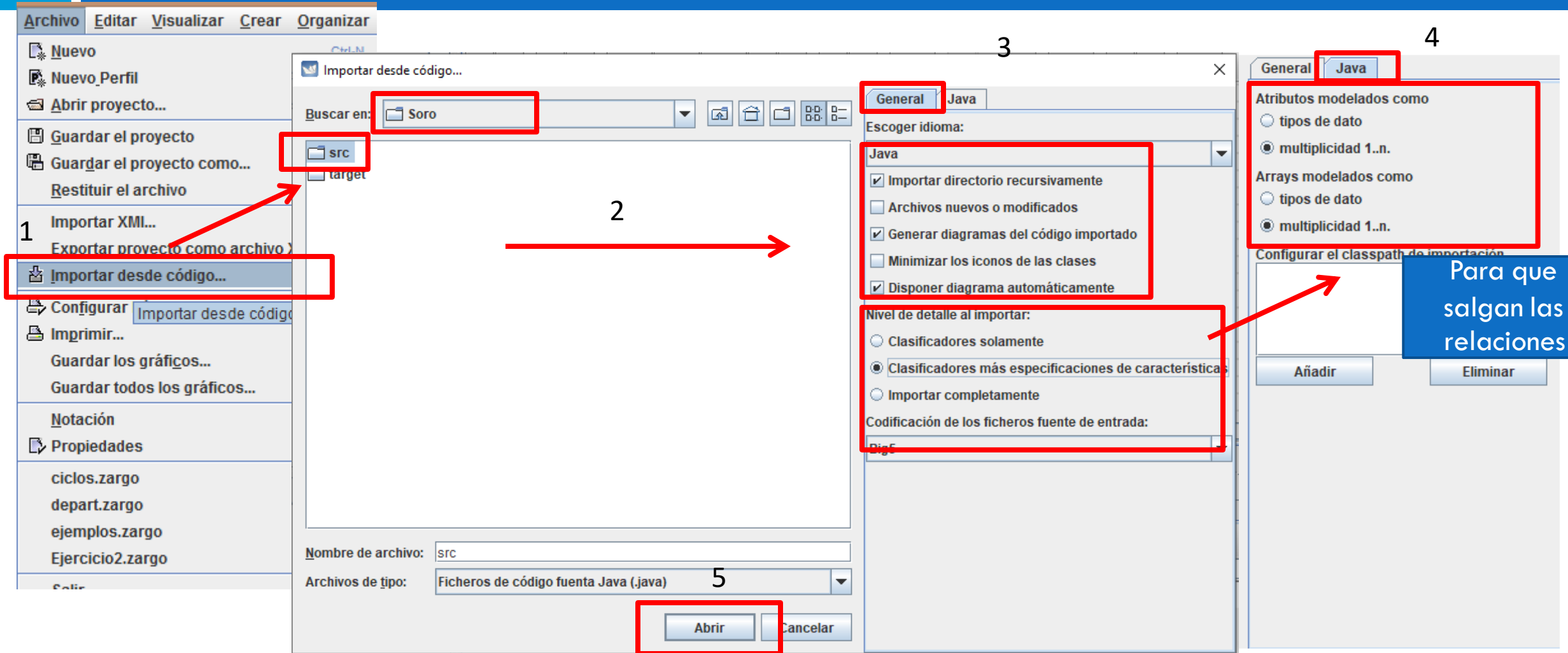
Ingeniería inversa

- Proceso de analizar código, documentación y comportamiento de una aplicación para identificar sus componentes actuales y sus dependencias y para extraer y crear una abstracción del sistema e información del diseño. El sistema en estudio no es alterado, sino que se produce un conocimiento adicional del mismo.
- *“Proceso que recorre hacia atrás el ciclo de desarrollo de software.”-*
P. Hall
- Dos opciones:
 - A partir del código fuente pero con documentación pobre o desactualizada.
 - A partir de código ejecutable => extraer el código fuente de un archivo ejecutable.

Ingeniería inversa. Tipos

- **Ingeniería inversa de datos:** Se aplica sobre algún código de bases datos (aplicación, código SQL, etc.) para obtener los modelos relacionales o sobre el modelo relacional para obtener el diagrama entidad-relación.
- **Ingeniería inversa de lógica o de proceso:** Cuando la ingeniería inversa se aplica sobre el **código** de un programa para averiguar su lógica (reingeniería), o sobre cualquier documento de diseño para obtener documentos de análisis o de requisitos.
- **Ingeniería inversa de interfaces de usuario:** Se aplica con objeto de mantener la lógica interna del programa para obtener los modelos y especificaciones que sirvieron de base para la construcción de la misma, con objeto de tomarlas como punto de partida en procesos de ingeniería directa que permitan modificar dicha interfaz.

Ingeniería inversa



Ingeniería inversa

The screenshot displays the Eclipse IDE interface for an inverse engineering project. The left-hand 'Project Explorer' pane shows a project named 'modelo sin título' which contains a 'Diagrama de clase' (Class Diagram) and a 'Diagrama de caso de uso' (Use Case Diagram). The 'modelo sin título_classes' package is selected, revealing a list of Java classes: 'Cliente.java', 'Director.java', 'Empleado.java', 'Empresa.java', and 'Persona.java'. Below these are primitive types (float, int, void) and several 'Generalization' elements. The bottom-left pane shows a 'Por prioridad' (By priority) view with 'Alta', 'Media', and 'Baja' categories. The main editor area on the right shows the 'modelo sin título_classes' class diagram, which is a UML class diagram for the 'Persona' class. The diagram includes attributes like '+ nombre : String' and '+ tlf : String', and methods such as '+ crear()', '+ asigna()', '+ borrar()', '+ getNombre()', '+ setNombre()', '+ getFecha_nacimiento()', '+ setFecha_nacimiento()', '+ getTlf()', and '+ setTlf()'. The bottom-right pane shows the 'Propiedades' (Properties) view for the selected class diagram, indicating the 'Nombre' (Name) is 'modelo sin título_classes' and the 'Modelo local' (Local model) is 'modelo sin título'.

Orientada a paquetes

Ordenar por tipo y luego por nombre

Profile Configuration

modelo sin título

- Diagrama de clase
- modelo sin título_classes
- Diagrama de caso de uso
- java
 - Cliente.java
 - Director.java
 - Empleado.java
 - Empresa.java
 - Persona.java
- float
- int
- void
- (Sin nombre Generalization)
- (Sin nombre Generalization)
- (Sin nombre Generalization)
- «» javaImport
- TD documentation
- TD GeneratedFromImport
- TD param

Por prioridad 14 elementos

- Alta
- Media
- Baja

As Diagram

Diagrama de clase

Nombre: modelo sin título_classes

Modelo local: modelo sin título

```
«class» + Persona
+ nombre : String
+ tlf : String
«methods» + Persona()
+ asigna( nom : String/fecha : Date/telf : String) : void
+ borrar() : Persona
+ getNombre() : String
+ setNombre(newNombre : String) : void
+ getFecha_nacimiento() : Date
+ setFecha_nacimiento(newFecha_nacimiento : Date) : void
+ getTlf() : String
+ setTlf(newTlf : String) : void
```

Ingeniería inversa. Ejercicio resuelto

- [Vídeo: Ingeniería Directa e Inversa. Ejercicio resuelto.](#)