

UML

Diagramas de Clases

(UML ilustrado)

Universidad de Los Andes

Demián Gutierrez

Marzo 2011

La **estructura estática** del sistema modelado
(piense en el plano estructural de un ingeniero civil)

Las **relaciones** que existen entre las distintas
clases y objetos del sistema

Las **clases y objetos** del sistema
y su estructura interna

Se concentran en los elementos del sistema de
forma **independiente del tiempo**
(Muestran aspectos estáticos y no dinámicos)

Realizar la abstracción de un **dominio** y formalizar el análisis de los **conceptos relacionados** al mismo
(*Modelo de Dominio*)
(...o de cualquier tipo de conceptos)

Definir / Documentar una **solución de diseño**, es decir, la **estructura** del sistema que se va a implementar en términos de **clases y objetos**

Definir / Documentar **modelado de datos**
(*Cumplen la misma función en este sentido de los diagramas ERE*)

Advertencia

El hecho de que exista cierta característica en un diagrama (ej: la declaración de métodos) no significa que de forma obligatoria se deba usar, simplemente son herramientas que están disponibles.

Cuando usted arregla algo, no usa todas las herramientas de su caja de herramientas, sólo usa lo que necesita para realizar el trabajo.

Igual ocurre con UML y las herramientas de modelado, use sólo las herramientas (diagramas / constructos) que necesita para una situación particular y no “sobre use” las herramientas, tratando de usarlas sólo porque si...

**¿Qué es un
Dominio de
Aplicación?**

**¿Qué es un Modelo
de Dominio?**

La mayoría de los conceptos
que se presentan en las
siguientes transparencias
están relacionados con los
conceptos de programación
orientada a objetos (POO)
vistos en PR2

¿Qué es una Clase?

Clase / Clasificador: Definición de la *estructura* y el *comportamiento* de un *conjunto de objetos* que tienen (comparten) el *mismo patrón estructural y de comportamiento*

Un ejemplo de una clase “número complejo”:

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

Base de Datos: ¿No les suena esto al concepto de tipo de entidad?

Atributos:

Propiedades relevantes de un clase
Representan su estructura
Pueden ser simples o compuestos

Métodos:

Comportamiento asociado a una
clase

PedidoComida

-correlativo
-fecha
-hora

+tomarNota(Mesa)
+cocinar()
+servir()
+cobrar()

**La relación que existe
entre el código y una
clase en UML es muy
importante**

**¡Necesito que hablemos
el mismo idioma!**

Diagramas de Clases

[¿Que es una Clase?]

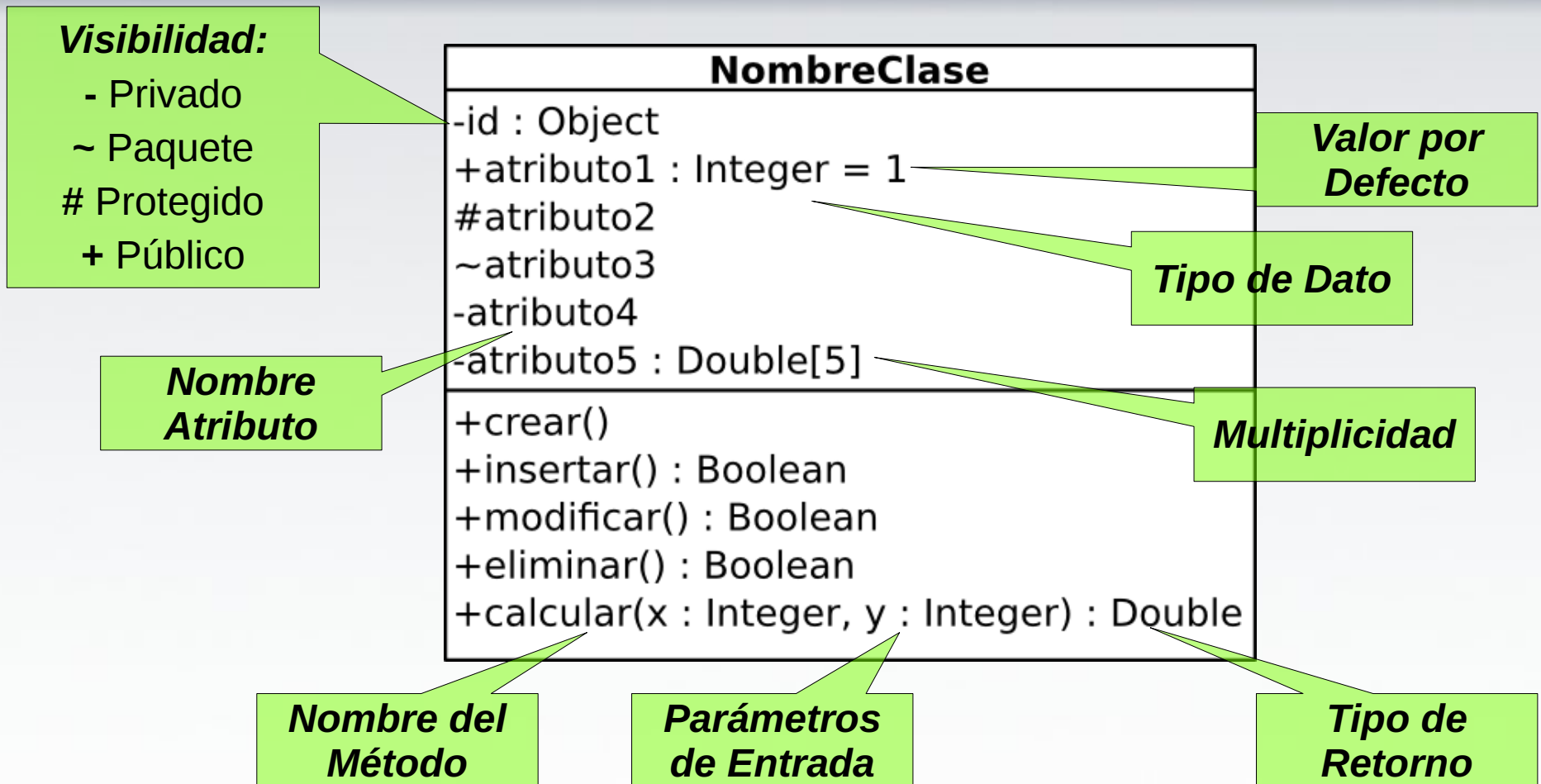
```
public class ComplexNumber {  
  
    private double r;  
    private double i;  
  
    public ComplexNumber(double r, double i) {  
        this.r = r;  
        this.i = i;  
    }  
  
    public double norm() {  
        return Math.sqrt(r * r + i * i);  
    }  
}
```

El código es
Java ;-)

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

Diagramas de Clases

(¿Que es una Clase?)



Base de Datos: Generalmente, cuando se desarrolla un modelo de datos no se utiliza toda esta complejidad. Por ejemplo, generalmente no se definen métodos

Para los Atributos:

[visibilidad] [/] nombre [:tipo] [multiplicidad] [=valor por omisión] [{**propiedad**}]

Para los Métodos:

[visibilidad] nombre [(lista de parámetros)] [{**propiedad**}]

Donde un parámetro es:

[dirección (in/out/inout)] nombre: tipo [multiplicidad] [=valor por omisión]

Las **propiedades** pueden una o mas de las siguientes:

readOnly, isQuery, Concurrent, Guarded, Sequential, etcétera

... o cualquier otra predefinida ...

¿qué es un objeto?

¿qué es una instancia?

¿qué es instanciar?

Instancia:

Cada objeto que pertenece a una clase

Instanciación / Instanciar:

Proceso de generación o creación de las instancias (objetos) de una clase

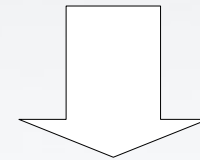
Objeto:

Representación de algo que se describe mediante un identificador, una estructura y un comportamiento.

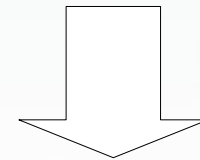
“Instancia de una Clase”

Persona

-cedula
-nombre



pedro = **new** Persona()



pedro : Persona


```
public class Persona {  
  
    private String nombre;  
    private char   sexo;  
    private Date   fechaNac;  
    private String profesion;  
  
    public Persona(  
        String nombre, char sexo, Date fechaNac, String profesion) {  
        this.nombre     = nombre;  
        this.sexo       = sexo;  
        this.fechaNac   = fechaNac;  
        this.profesion  = profesion;  
    }  
}
```

Persona
-nombre -sexo -fechaNac -profesion
+Persona(nombre, sexo, fechaNac, profesion)

Existe una diferencia muy importante entre un **Objeto** y una **Clase**

Persona
-nombre -sexo -fechaNac -profesion
+Persona(nombre, sexo, fechaNac, profesion)

```
Persona p1 = new Persona(  
    "Pedro", 'M', new Date(16, 7, 1988), "Actor" );  
  
Persona p2 = new Persona(  
    "Andrea", 'F', new Date(14, 4, 1980), "Ceramista");  
  
Persona p3 = new Persona(  
    "María", 'F', new Date(23, 11, 1960), "Médico" );  
  
Persona p4 = new Persona(  
    "Luis", 'M', new Date(12, 1, 1977), "Ingeniero");
```

Crear Instancias (Instanciar)

Pedro : Persona
sexo = M fechaNac = 16/07/1988 profesion = Actor

Andrea : Persona
sexo = F fechaNac = 14/04/1980 profesion = Ceramista

María : Persona
sexo = F fechaNac = 23/11/1960 profesion = Médico

Luis : Persona
sexo = M fechaNac = 12/01/77 profesion = Ingeniero

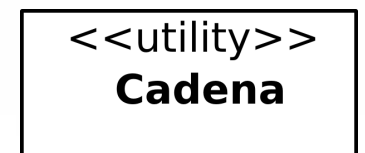
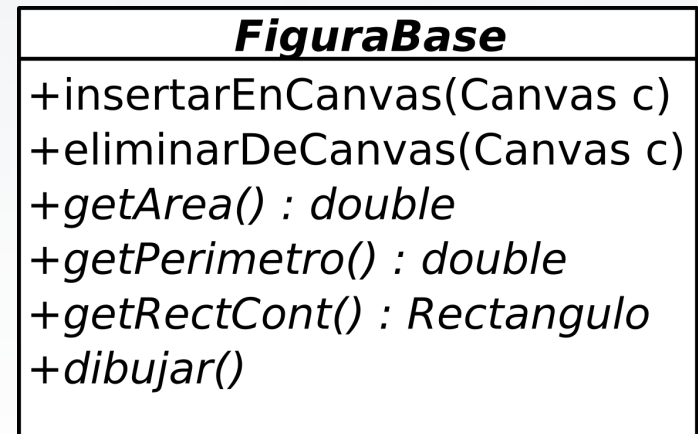
Diagramas de Clases

(Parametrizables / Abstractas / Utilitarias)

Clases Parametrizables: Plantillas de clases que se pueden parametrizar con uno o más tipos de datos según sea necesario (Clases Genéricas)

Clases Abstractas: Clases que no tienen implementación para todos los métodos definidos

Clases Utilitarias: Clases que contienen librerías de funciones (no interesa mucho la implementación)



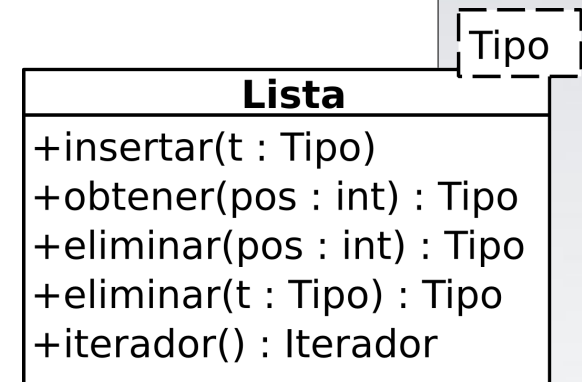
Diagramas de Clases

{Parametrizables / Abstractas / Utilitarias}

```
public class Lista<Tipo> {  
  
    public void insertar (Tipo t)  
        { /* código */ }  
  
    public void eliminar (Tipo t)  
        { /* código */ }  
  
    public Tipo eliminar (int pos)  
        { /* código */ }  
  
    public Tipo obtener (int pos)  
        { /* código */ }  
  
    public Iterador iterador ()  
        { /* código */ }  
}
```

// La clase se usa de la siguiente forma:

```
Lista<int> listaDeEnteros = new Lista<int>();  
Lista<Persona> listaDePersonas = new Lista<Persona>();
```



Diagramas de Clases

{Parametrizables / Abstractas / Utilitarias}

```
public abstract class FiguraBase {  
  
    public void insertarEnCanvas(Canvas c) { /* código */ }  
    public void eliminarDeCanvas(Canvas c) { /* código */ }  
  
    // Los métodos siguientes son abstractos,  
    // es decir, no tienen implementación  
  
    public abstract double getArea();  
    public abstract double getPerimetro();  
    public abstract double getRectCont();  
    public abstract double getDibujar();  
}
```

FiguraBase

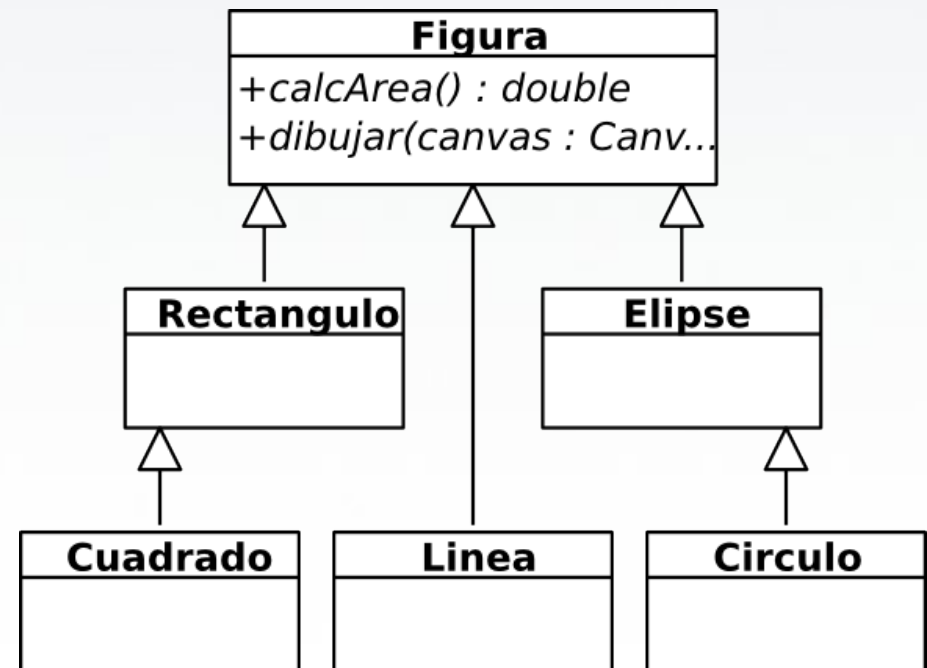
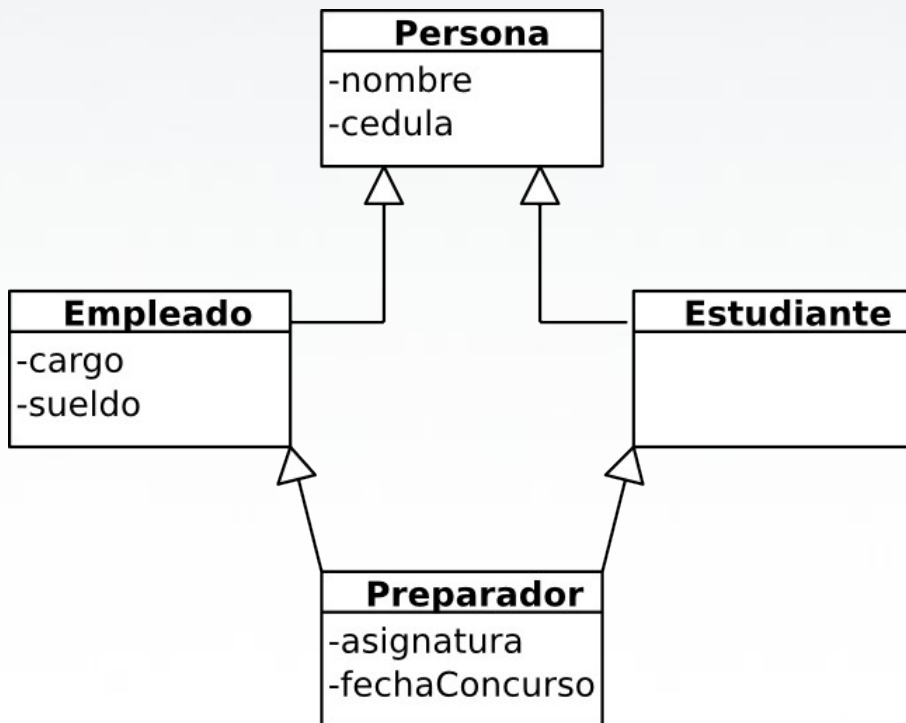
```
+insertarEnCanvas(Canvas c)  
+eliminarDeCanvas(Canvas c)  
+getArea() : double  
+getPerimetro() : double  
+getRectCont() : Rectangulo  
+dibujar()
```

Diagramas de Clases

[Especialización / Generalización / Herencia]

Jerarquía de Clases:
Relación ES-UN(A),
abstracciones de
generalización /
especialización de clases

Herencia: Propiedad que
tienen las clases de heredar
de sus superclases estructura
y/o comportamiento (Simple /
Múltiple)

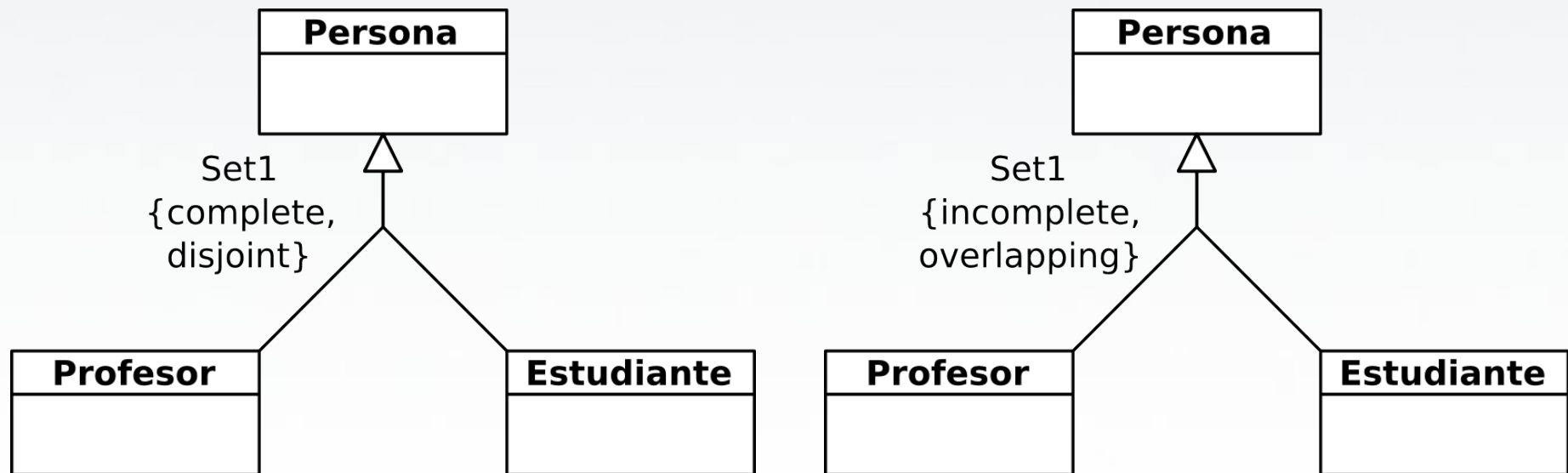


Diagramas de Clases

[Especialización / Generalización / Herencia]

Herencia:

Disjunta / Traslapada
Total / Parcial



Diagramas de Clases

(Especialización / Generalización / Herencia)

```
public abstract class Figura {  
    public abstract double calcArea ();  
    public abstract void dibujar (Canvas canvas);  
}
```

```
public class Rectangulo  
    extends Figura {  
    // ...  
}
```

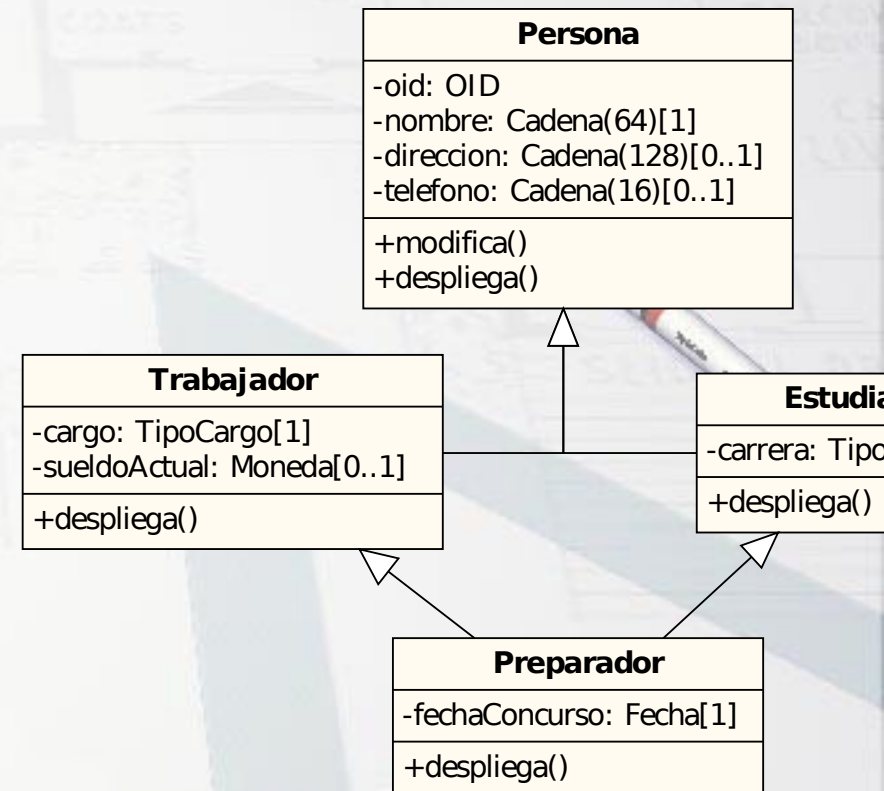
```
public class Elipse  
    extends Figura {  
    // ...  
}
```

```
public class Cuadrado  
    extends Rectangulo {  
    // ...  
}
```

```
public class Circulo  
    extends Elipse {  
    // ...  
}
```


Vista lógica o estructural

- **Polimorfismo:** se puede usar el mismo nombre para la definición de un método en varias clases sin importar la relación entre las mismas.
- **Reescritura o sobrecarga:** permite nombrar código diferente con el mismo nombre para más de una clase de objetos.
- **Encadenamiento tardío:** permite seleccionar el código adecuado al objeto definido en la invocación del método.

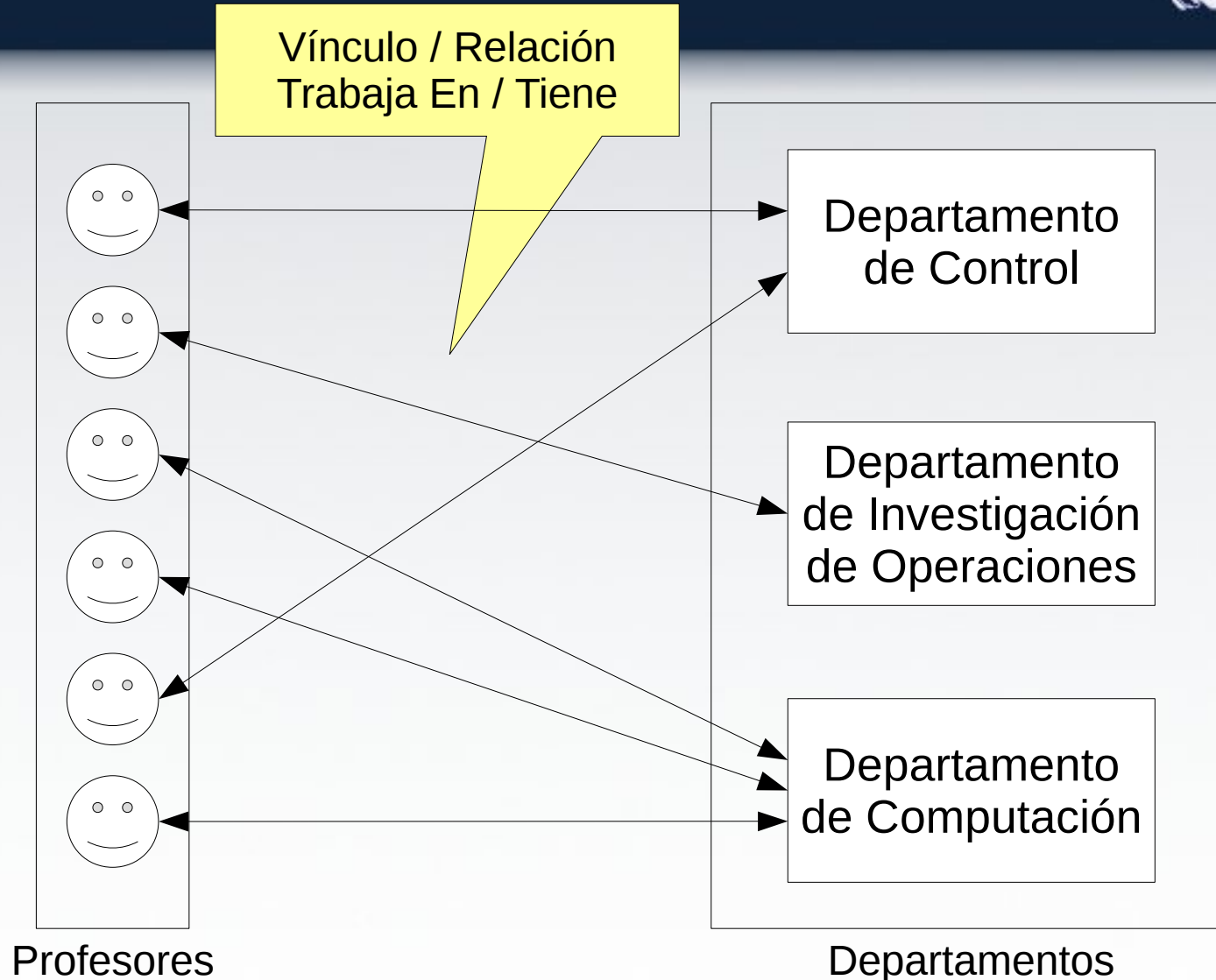


¿Asociaciones?
¿1:1, 1:N y N:M?

Eso se puede ver
mejor con un ejemplo

Relaciones (Vínculos)

1:N

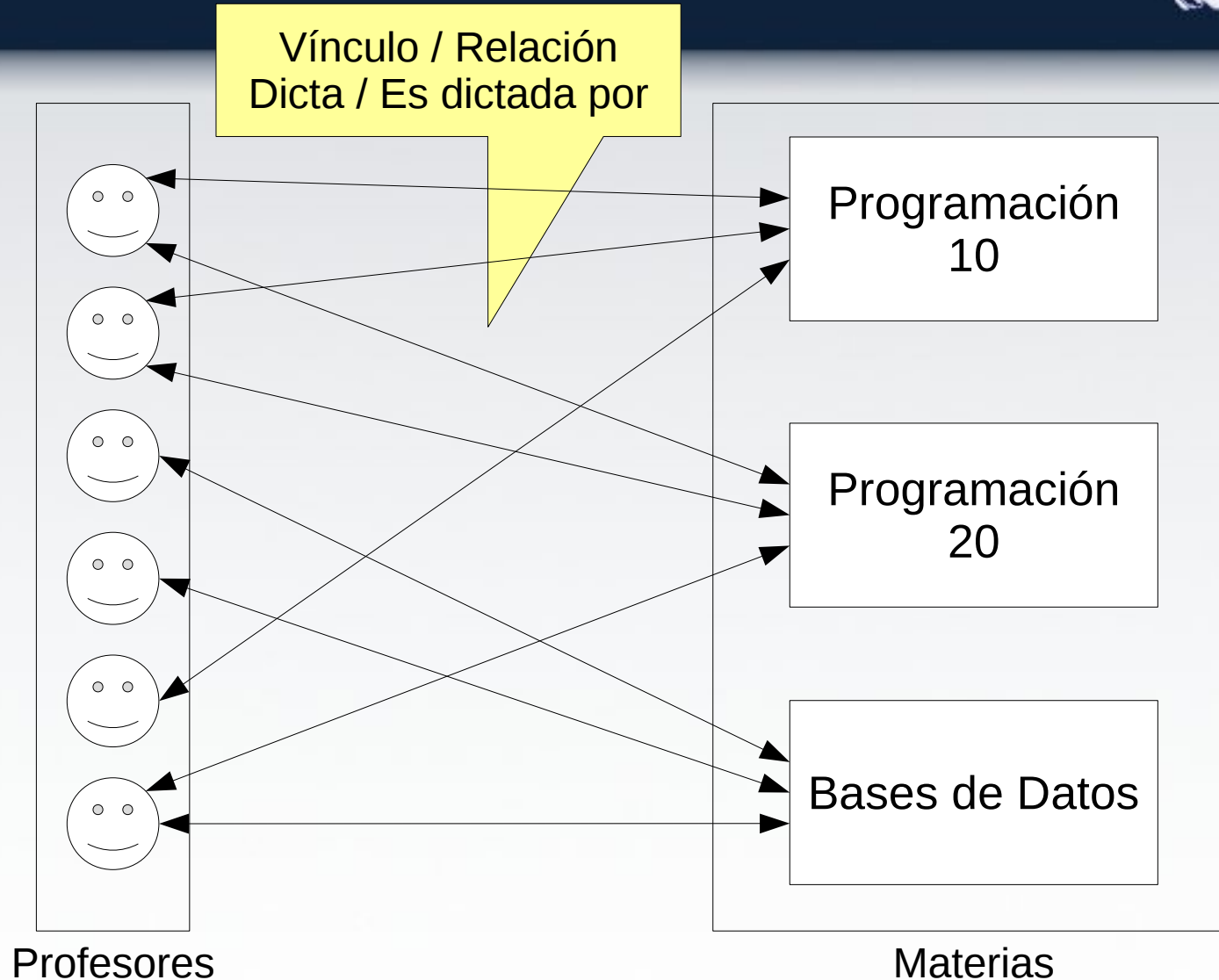


¿Cuántos profesores puedo tener en el conjunto de entidades "Profesores"?
¿Y en "Departamentos"?

¿Con cuantos profesores puede estar asociado un departamento?
¿Y al contrario?

Relaciones (Vínculos)

N:M

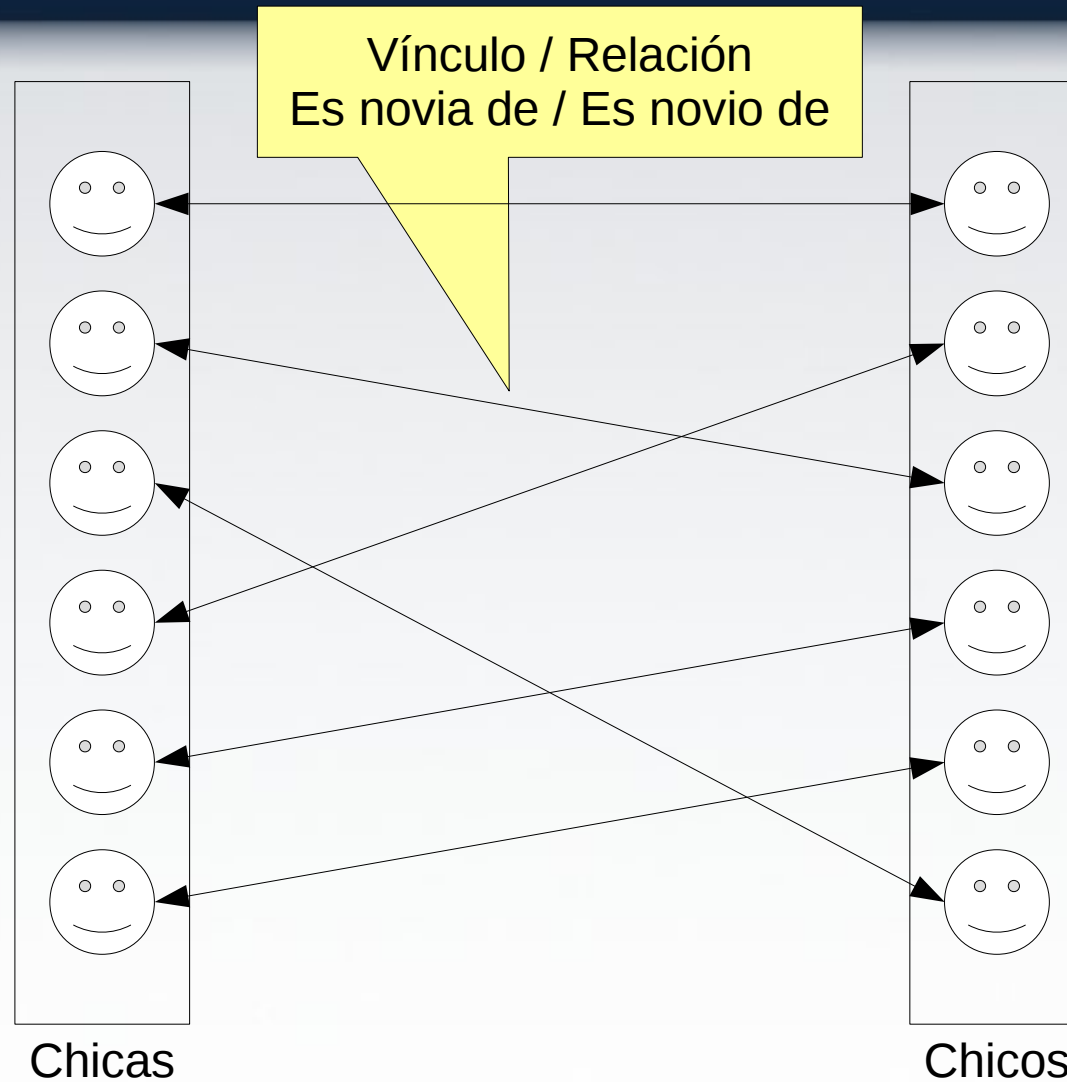


¿Cuántos profesores puedo tener en el conjunto de entidades "Profesores"?
¿Y en "Materias"?

¿Con cuántos profesores puede estar asociado una materia? ¿Y al contrario?

Relaciones (Vínculos)

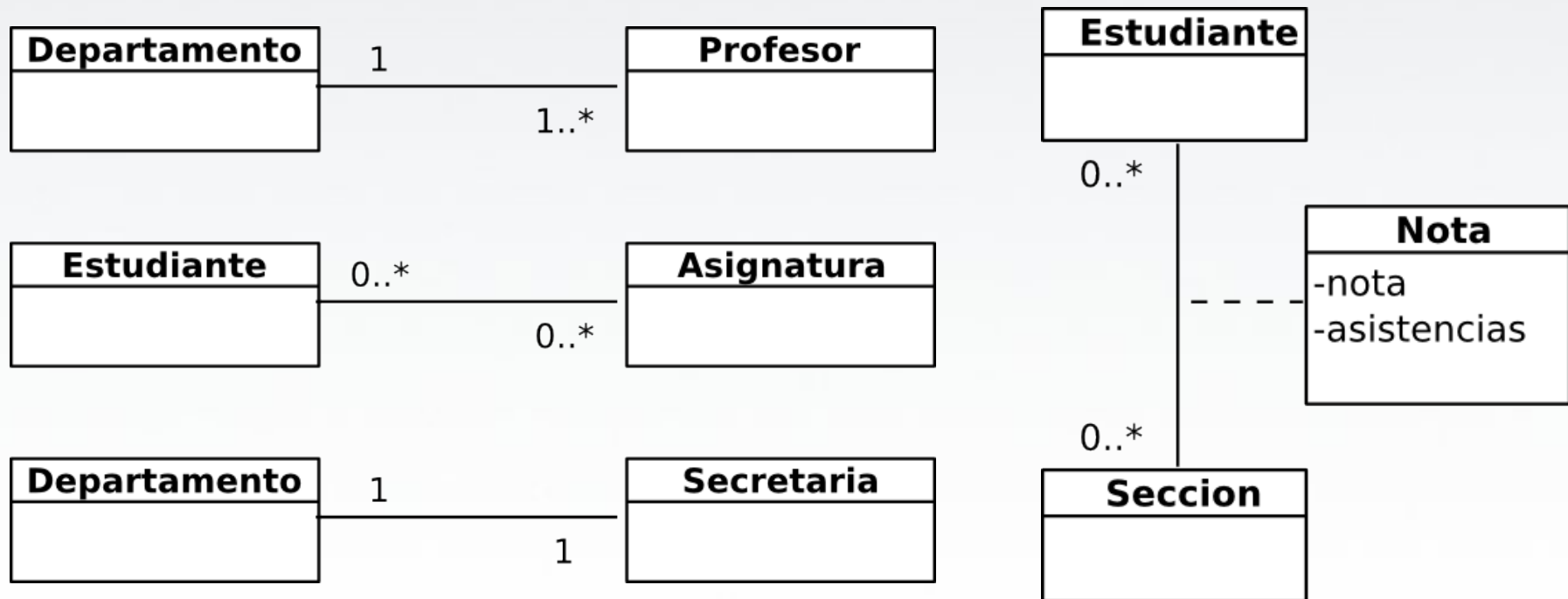
1:1



¿Cuántos muchachos puedo tener en el conjunto de entidades “Chicos”?
¿Y en “Chicas”?

¿Con cuántos Chicos puede estar asociados (ser novios) de una Chica en particular? ¿Y al contrario?

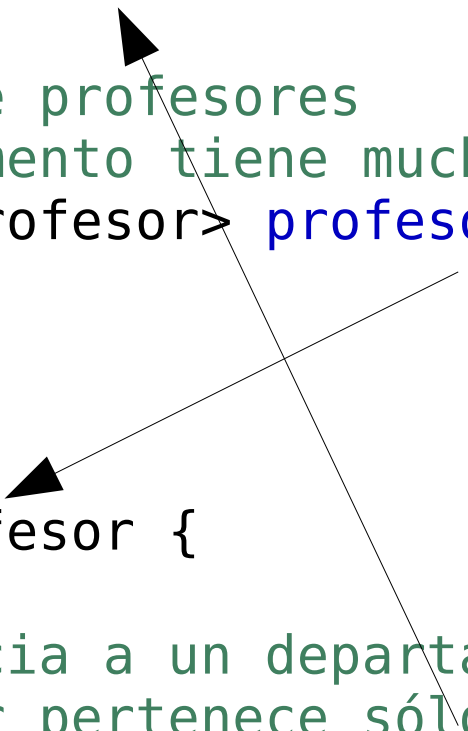
Asociaciones: Representan relaciones estructurales entre las clases (la forma en que están relacionadas entre si las clases)



¿Cómo se implementan?

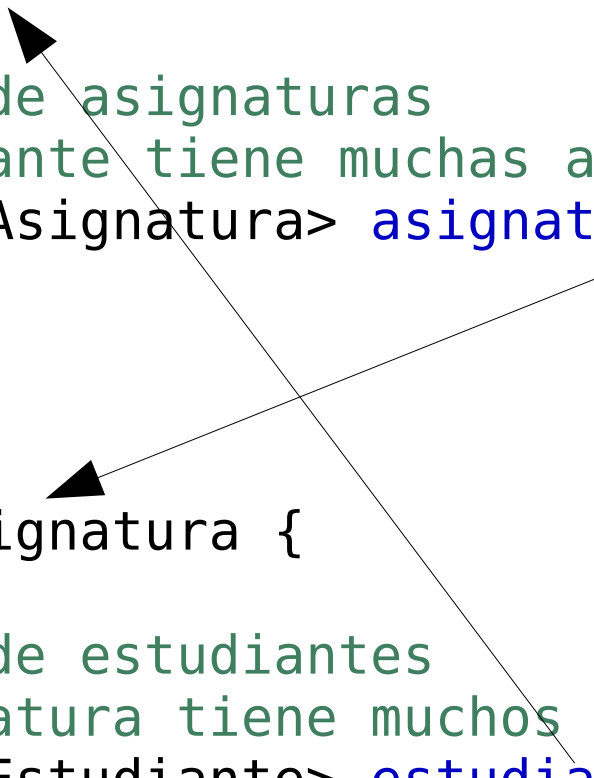
Diagramas de Clases (Asociaciones)

```
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;  
}
```



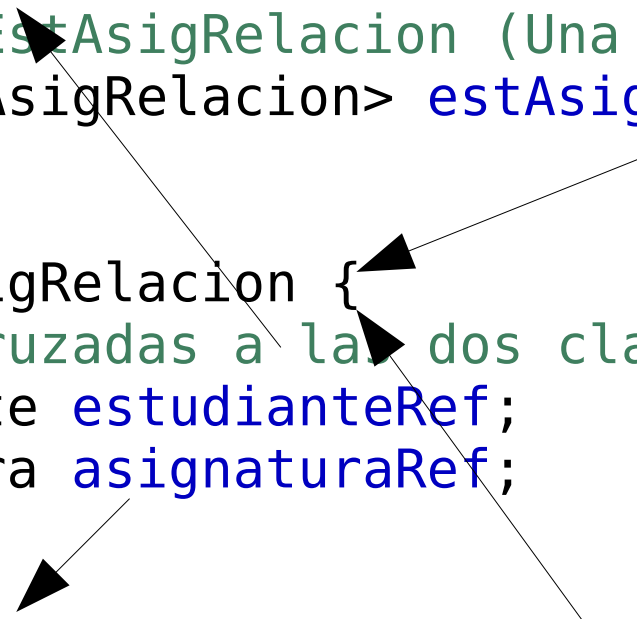
Diagramas de Clases (Asociaciones)

```
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;  
}
```

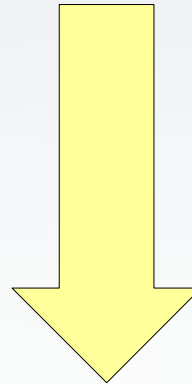


Diagramas de Clases (Asociaciones)

```
public class Estudiante {  
    // Una lista de EstAsigRelacion (Una clase relaci3n)  
    private List<EstAsigRelacion> estAsigRelacionList;  
}  
  
public class EstAsigRelacion {  
    // referencias cruzadas a la dos clases relacionadas  
    private Estudiante estudianteRef;  
    private Asignatura asignaturaRef;  
}  
  
public class Asignatura {  
    // Una lista de EstAsigRelacion (Una clase relaci3n)  
    private List<EstAsigRelacion> estAsigRelacionList;  
}
```



Diagramas de Clases (Asociaciones)

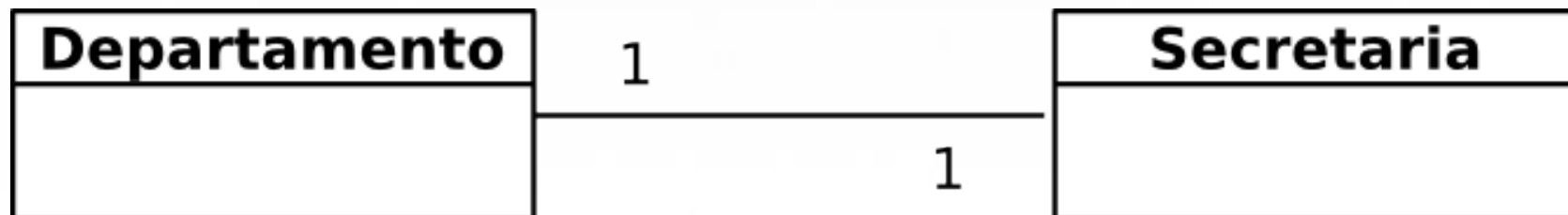
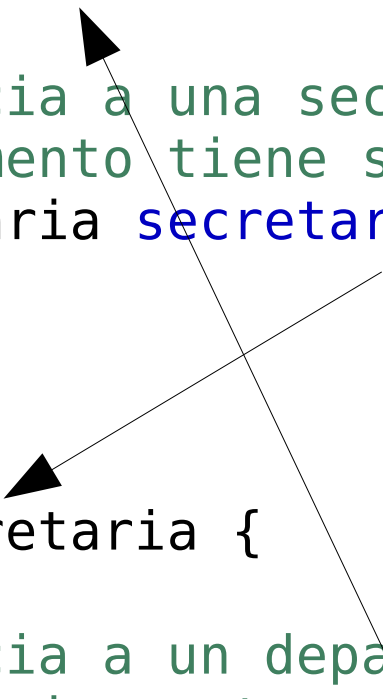


Una relación
muchos a muchos
se puede ver como
dos relaciones uno a
muchos



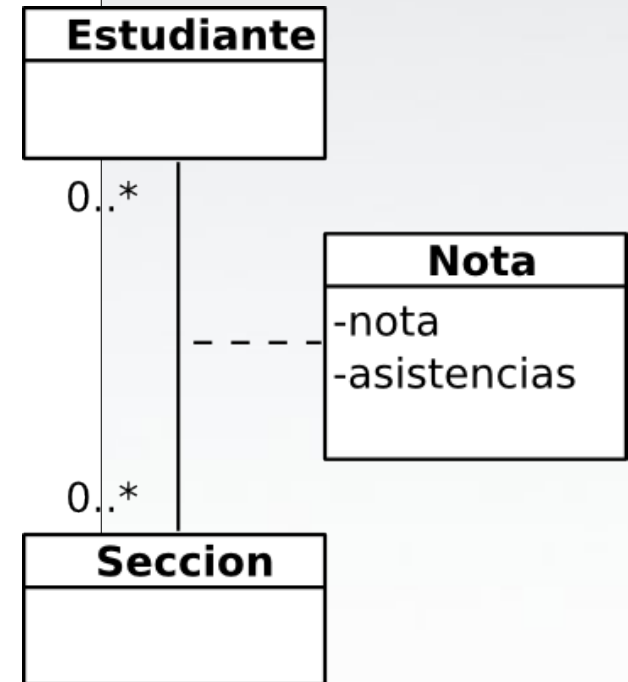
Diagramas de Clases (Asociaciones)

```
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;  
}
```



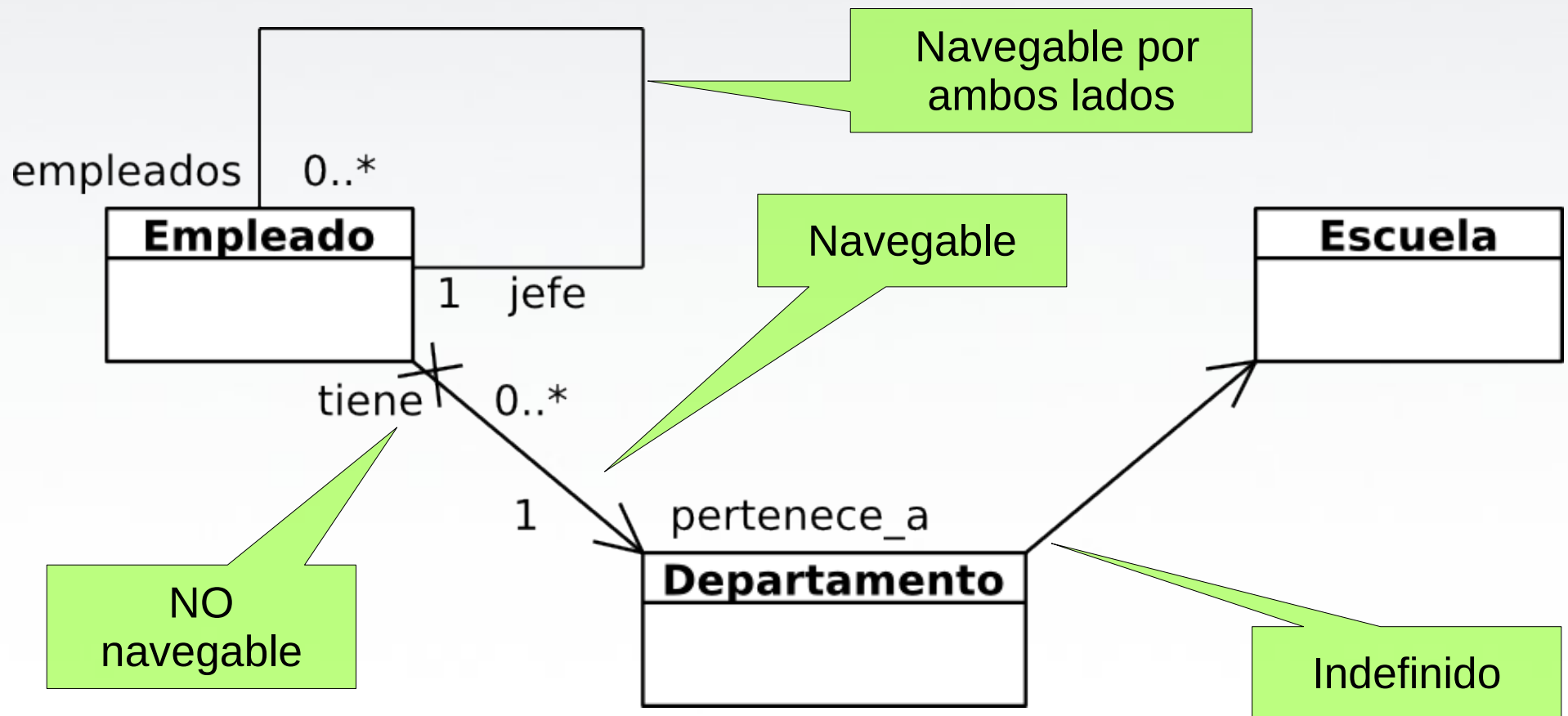
Diagramas de Clases (Asociaciones)

```
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;  
}
```



Diagramas de Clases (Asociaciones / Navegabilidad)

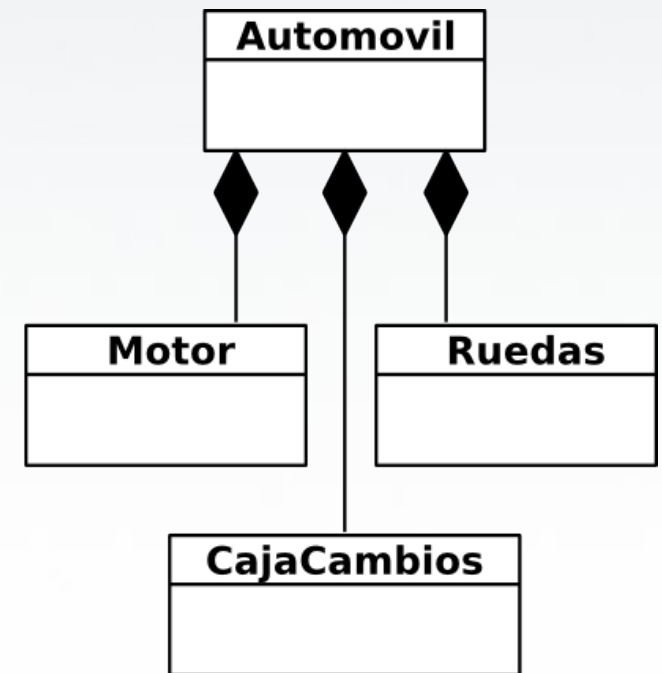
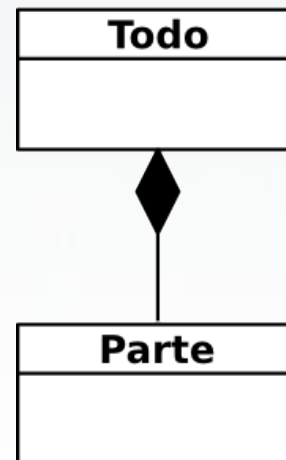
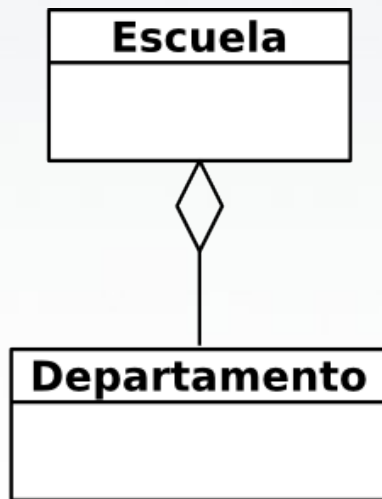
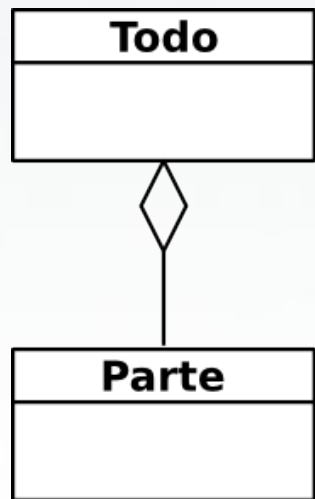
Navegabilidad: Representan relaciones estructurales entre las clases (la forma en que están relacionadas entre si las clases)



Diagramas de Clases (Agregación / 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?

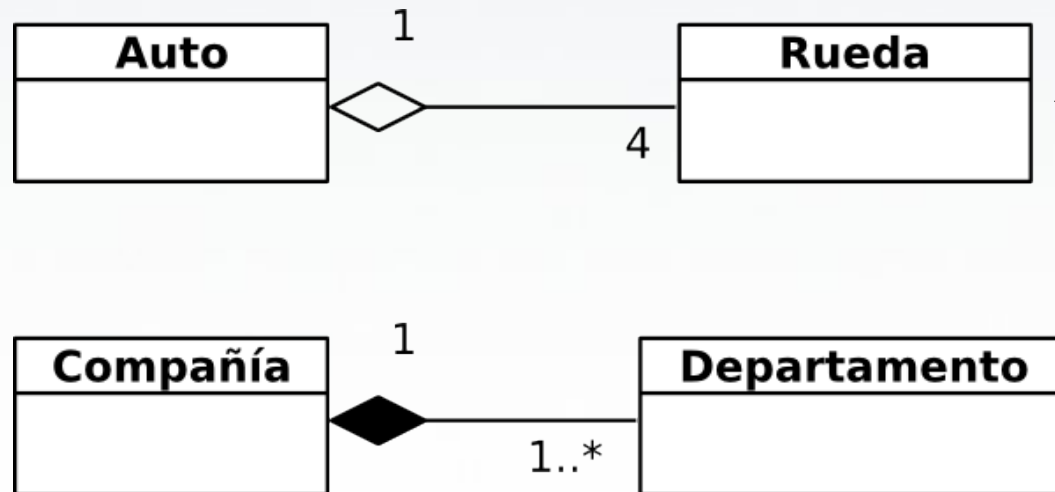
Diagramas de Clases (Agregación / Composición)

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)



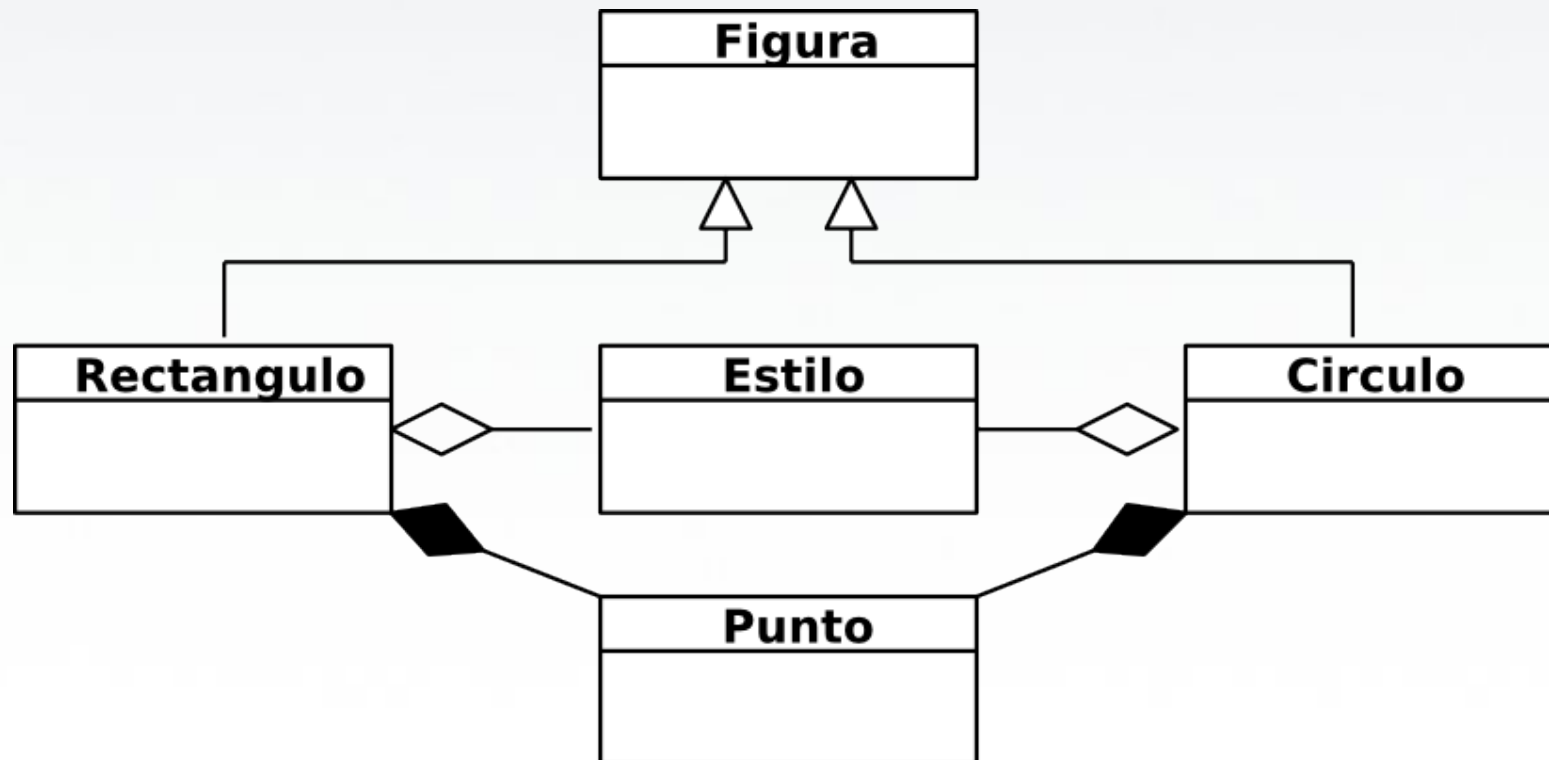
¿La parte (La rueda) puede existir sin el todo?

Diagramas de Clases (Agregación / Composición)

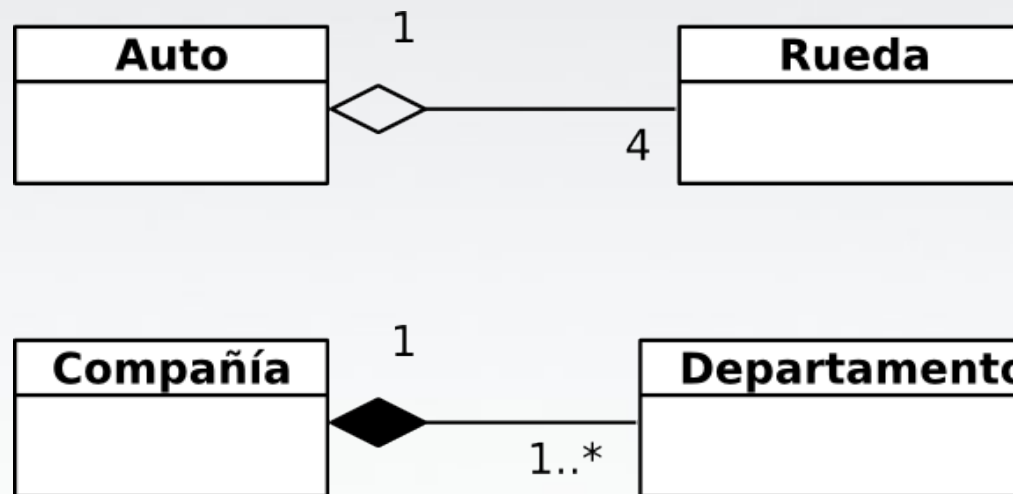
Peor aún...

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

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

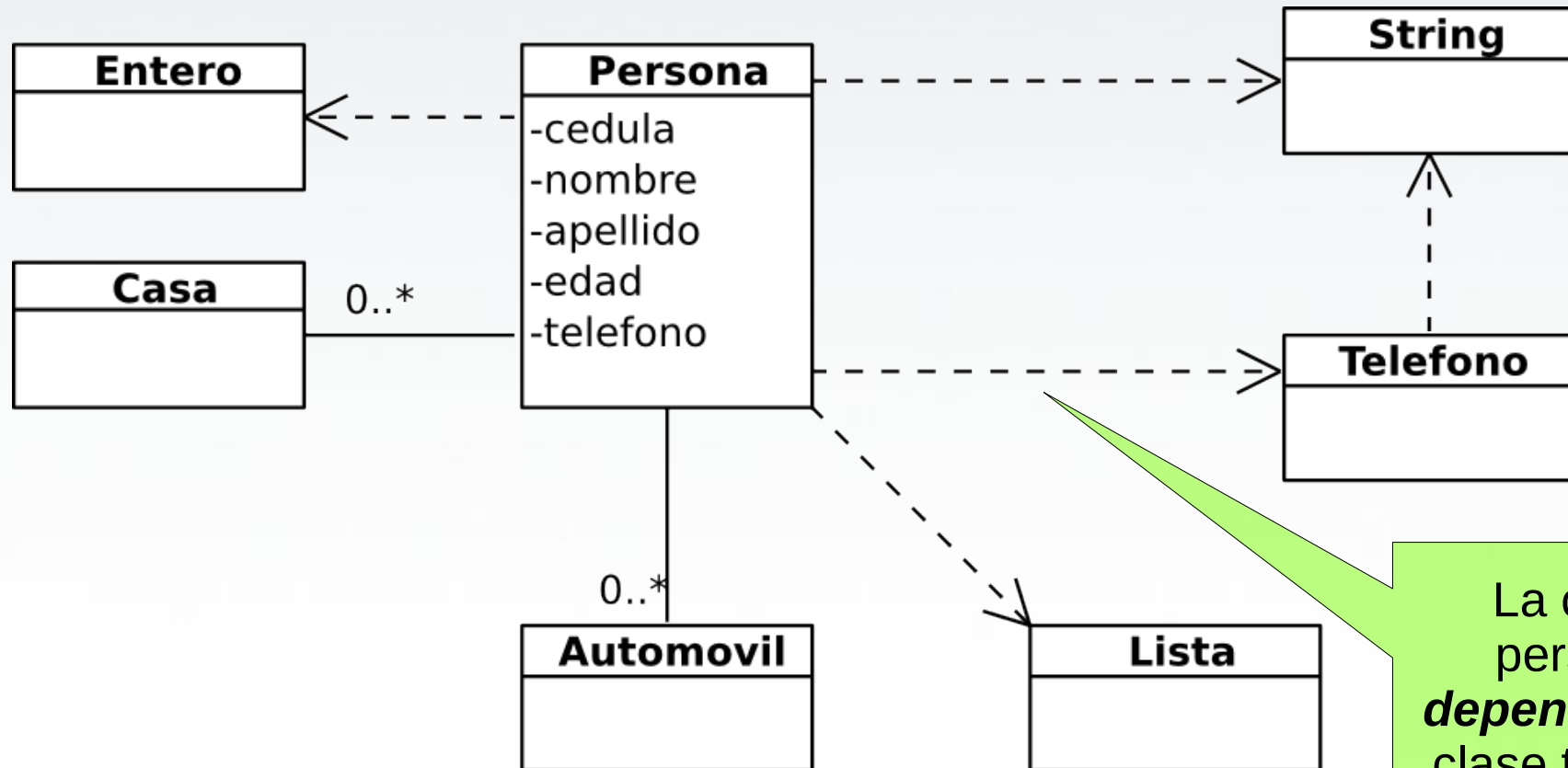


*“Precise semantics of **shared aggregation** varies by application area and modeler”*



*“Indicates that the property is aggregated **compositely**, i.e., the composite object has responsibility for the existence and storage of the composed objects (parts)”*

Dependencia: Relación en la que una clase necesita (requiere) a otra para poder funcionar



La clase
persona
depende de la
clase teléfono

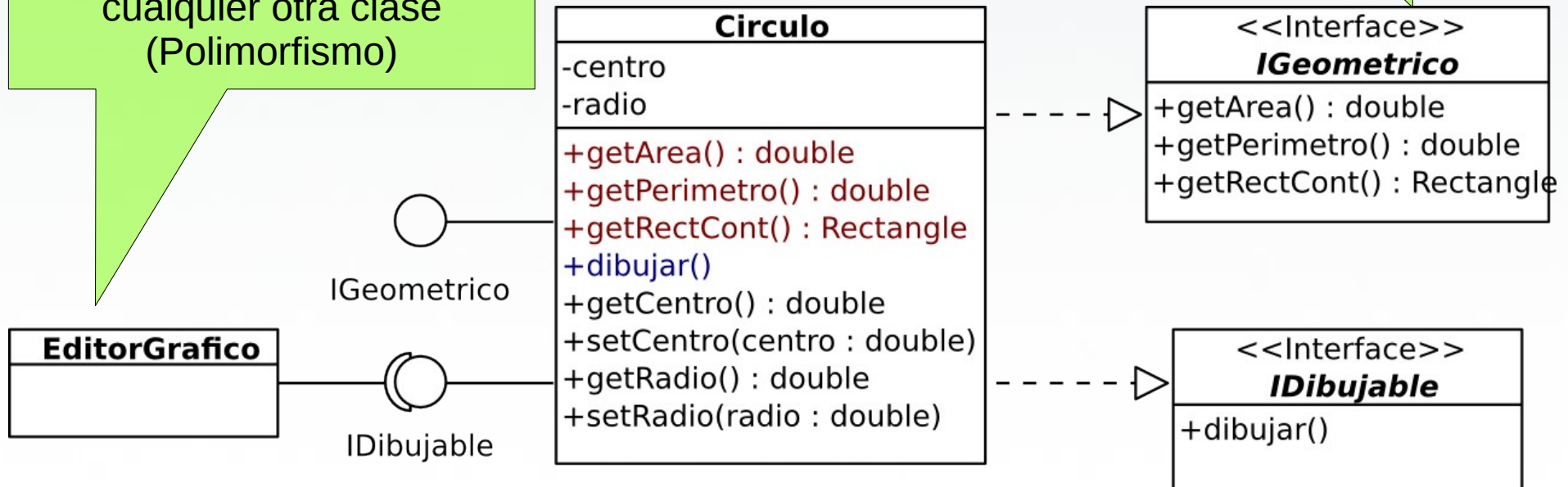
**¿Qué / Cuál es la
Interfaz de una clase?**

Diagramas de Clases (Interfaces / Realizaciones)

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

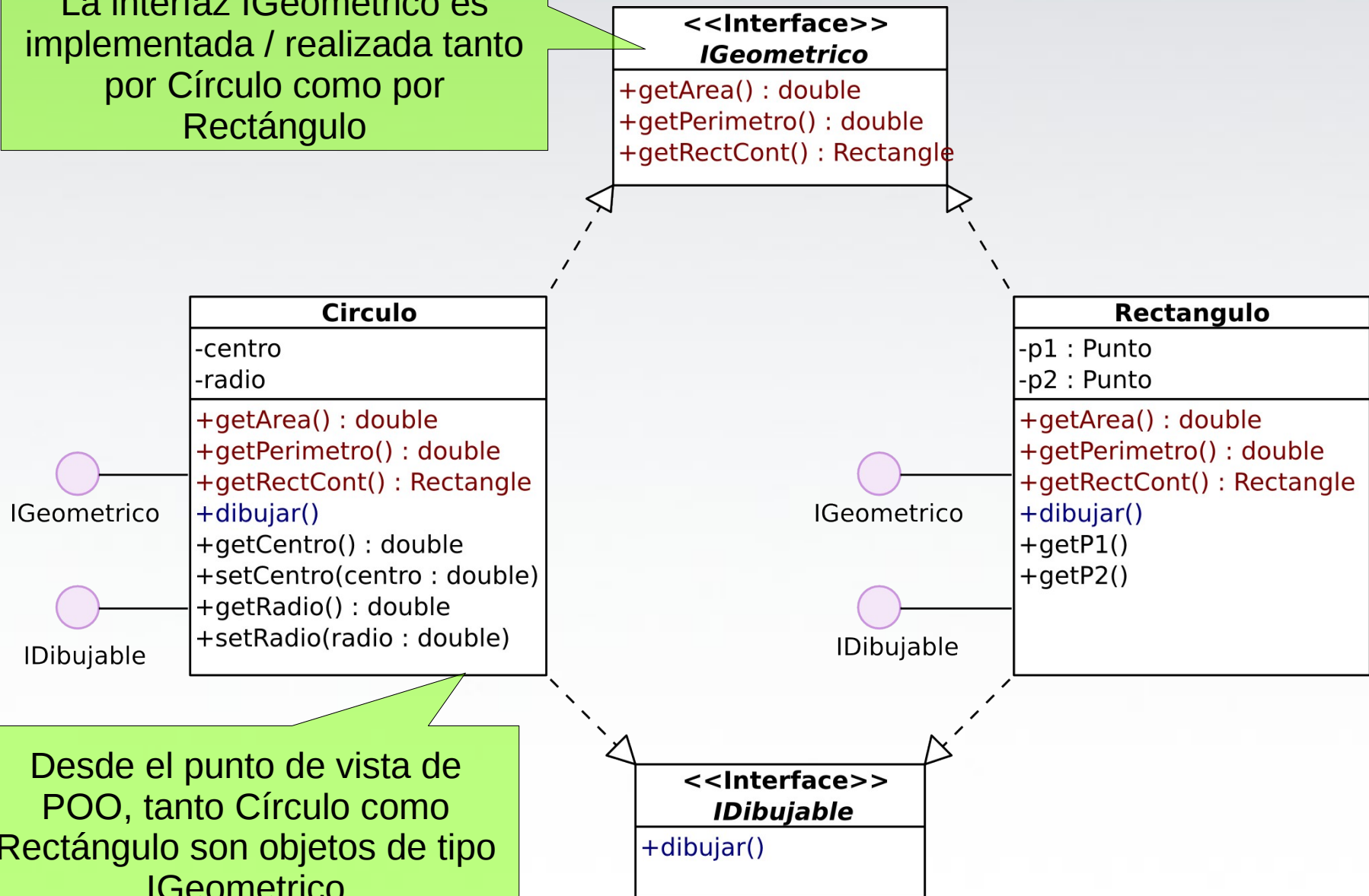
EditorGrafico es una clase que usa la interfaz Idibujable, independientemente que la implemente un Círculo o cualquier otra clase (Polimorfismo)

<<interface>> es un estereotipo



Diagramas de Clases (Interfaces / Realizaciones)

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

Diagramas de Clases

[Interfaces / Realizaciones]

```
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 */ }
}
```

Diagramas de Clases

[Interfaces / Realizaciones]

```
import java.awt.Rectangle;

public interface IGeometrico {

    public double getArea();

    public double getPerimetro();

    public Rectangle getRectCont();
}
```

```
public interface IDibujable {
    public void dibujar();
}
```

Diagramas de Clases (Interfaces / Realizaciones)

```
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!