TEMA 2

FUNDAMENTOS DE LA PROGRAMACIÓN ORIENTADA A OBJETOS

Cristina Cachero, Pedro J. Ponce de León

(3 sesiones)
Versión 0.6
(Curso 09/10)

Depto. Lenguajes y Sistemas Informáticos

Universitat d'Alacant
Universidad de Alicante

Del tema anterior…

- **ABSTRACCIÓN**
- **OCULTACIÓN DE INFORMACIÓN.**
- **ENCAPSULACIÓN**
- **Objetos (instancias, agentes)**
- **Clases**
- **Jerarquías de clases (herencia)**
- **Responsabilidades**
- **Paso de mensajes**
- **Métodos**
- **Polimorfismo**
- **Enlace dinámico**
- **etc…**

Indice

- **Objetos**
- Clases
- Atributos
- Operaciones
- Constructores y destructores
- UML y el diagrama de clases
- Relaciones entre objetos
- Diseño O.O.

Objeto
Definición

- Un objeto es cualquier cosa a la que podamos asociar unas determinadas **propiedades** y **comportamiento.**

- Desde el punto de vista del analista: un objeto representa una entidad (real o abstracta) con un papel bien definido en el dominio del problema.

- Desde el punto de vista del programador: un objeto es una estructura de datos sobre la cual podemos realizar un conjunto bien definido de operaciones.

Objeto
Definición

- Según *Grady Booch*: Un objeto tiene un estado, un comportamiento y una identidad:
  - **Estado**: conjunto de propiedades del objeto y valores actuales de esas propiedades.
  - **Comportamiento**: modo en que el objeto actúa y reacciona ante los mensajes que se le envían (con posibles cambios en su estado). Viene determinado por la **clase** a la que pertenece el objeto.
  - **Identidad**: propiedad que distingue a unos objetos de otros (nombre único de variable)

Objeto
Estado y Comportamiento

- El **estado** de un objeto puede influir en su **comportamiento**

- Ejemplo: una cuenta bancaria puede variar su comportamiento en función de su saldo:
  - *Cuenta con saldo negativo*: no se puede retirar dinero pero sí ingresar. Se le cobra comisión. Se le envía notificación de reclamo de puesta al día.
  - *Cuenta con saldo positivo:* puede retirar e ingresar dinero. No se cobra comisión de mantenimiento.

Indice

- Objetos
- **Clases**
- Atributos
- Operaciones
- Constructores y destructores
- UML y el diagrama de clases
- Relaciones entre objetos
- Diseño O.O.

Clase
Definición

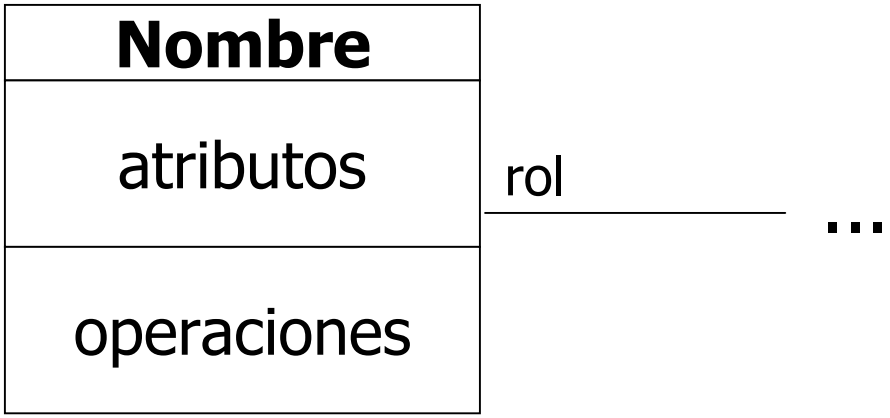
- Abstracción de los atributos (características), operaciones, relaciones y semántica comunes a un conjunto de objetos.

- Así, una clase representa al conjunto de objetos que comparten una estructura y un comportamiento comunes. Todos ellos serán **instancias** de la misma clase.

- Elemento central del paradigma OO.
  - No hay noción de programa principal, y los subprogramas no existen como unidades modulares independientes, sino que forman siempre parte de alguna clase.
    - El *programa principal* se convierte en un simple punto de entrada al programa, y no debería contener lógica de control.

Clase
Partes de la Definición

- **Identificador de Clase**: nombre
- **Propiedades**
  - **Atributos** o *variables*: datos necesarios para describir los objetos (*instancias*) creados a partir de la clase.
    - La combinación de sus valores determina el estado de un objeto.
  - **Roles**: relaciones que una clase establece con otras clases.
  - Operaciones, métodos, servicios (funciones miembro en C++): acciones que un objeto conoce cómo ha de ejecutar.




```
class Nombre  {
  tipo1 atributo1;
  tipo2 atributo2;
…
  tipoX operacion1();
  tipoY operacion2(…);
 …
}; // C++
```

Clases y Objetos
Ejemplos

- Objetos
  - En una aplicación de nóminas:
    - Habrá objetos que representan a cada empleado individual.
  - En un sistema de matriculación universitario:
    - Habrá objetos que representan a cada alumno, cada asignatura, cada profesor, etc…
  - En una aplicación de control de una fábrica automatizada:
    - Habrá objectos que representan a cada línea de ensamblaje, cada robot, cada producto que se fabrica.

- Clases
  - En una aplicación de nóminas:
    - 'Empleado' es la clase de las personas que trabajan para la empresa.
  - En un sistema de matriculación universitario:
    - 'Alumno', 'Asignatura' y 'Profesor' son clases de objetos del sistema
  - En una aplicación de control de una fábrica automatizada:
    - 'LineaEnsamblaje', 'Robot', 'Producto' son clases que representan diferentes tipos de objetos de la fábrica.

¿Objetos o clases?

- Película **CLASE**
- Carrete de película **CLASE**
- Carrete con nº de serie 123456 **OBJETO**
- Pase de la película 'La vida de Brian' en el cine Muchavista a las 19:30 **OBJETO**

En general:
- Algo será una clase si puede tener instancias.
- Algo será un objeto si es algo único que comparte características con otras cosas similares

¿Clases u objetos?

- ¿Cuál de los items siguientes es una clase y cual un objeto? Para los que sean clases, nombra algún objeto de esa clase. Para los que son objetos, nombra una clase a la que podrían pertenecer.

1. Leche Pascual
2. Airbus 360
3. Juan Albeniz
4. Juego de mesa
5. Asignatura POO 9190
6. La final del Abierto de EEUU de tenis del año 2009
7. Fabricante de automóviles
8. Estudiante de informática
9. El coche con nº de serie UGA12345678
10. Juego
11. Ajedrez
12. Avión

Convenio de nombres

- Clases:
  - Nombres en singular. Mayúscula inicial.

OK:

- JuegoDeMesa
- Fabricante
- Estudiante
- Juego
- Avión

- Juego_de_mesa
- fabricante
- DatosEstudiante
- InformacionJuego
- RegistroAvión

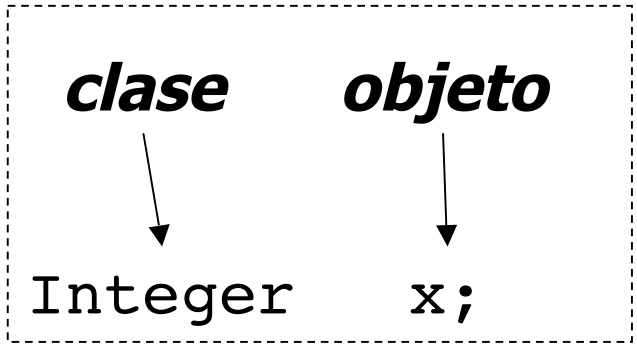
: mal

- Objetos:
  - Minúscula inicial.

- lechePascual
- unAirbus
- juanAlbeniz
- pooII
- miCoche

Objeto
Objetos y clases en un lenguaje de programación

- Clase ⟷ Tipo
- Objeto ⟷ Variable o constante



Clase: caracterización de un conjunto de objetos que comparten propiedades

Objeto
Creación e Inicialización

- Antes de comenzar a trabajar con un objeto, éste debe estar:
  - **Creado**: reserva de memoria y enlazado de ese espacio de memoria a un nombre.
  - **Inicializado**: establecer las condiciones iniciales necesarias para la manipulación de un objeto.

  - En algunos lenguajes, como C++, el proceso de creación va unido al proceso de nombrar una variable:

```
Naipe AsDeBastos;
```

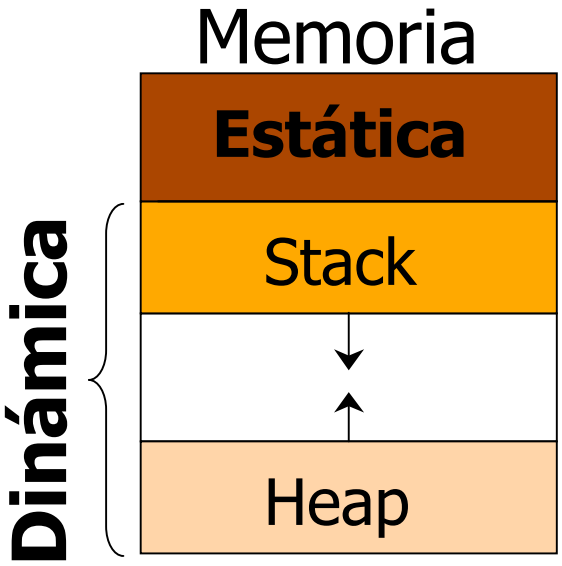
  - En otros, como *Java,* ambas operaciones se separan:

```
Naipe AsDeBastos; // declaración de referencia
AsDeBastos = new Naipe(As, Bastos);
```

Objeto
Punteros y ubicación en memoria

- Todos los lenguajes utilizan punteros para la representación interna de objetos.

- **C++:** Distinguimos entre:
  - ***variables automáticas*** (memoria ***stack)*** : desaparece cuando la variable sale de ámbito de manera automática.
  - ***punteros*** (memoria ***heap)***: los objetos se crean con el operador **`new`**. Deben eliminarse explícitamente con el operador **`delete`**.



Objeto
Liberación de memoria

- En C++ es responsabilidad del programador liberar la memoria *heap* ocupada por un objeto cuando ya no hay más referencias a ella.

- Otros lenguajes OO utilizan un ***recolector automático de basura*** *(garbage collector):* Java, C#, SmallTalk, Clos,…

- Este recolector automático es más costoso (usa tiempo de ejecución) (menor eficiencia) pero da lugar a código más flexible al evitar:
  - Quedarse sin memoria al olvidar liberarla.
  - Que el programador intente usar memoria después de que haya sido liberada.
  - Que el programador intente liberar la misma memoria más de una vez.

Indice

- Objetos
- Clases
- **Atributos**
- Operaciones
- Constructores y destructores
- UML y el diagrama de clases
- Relaciones entre objetos
- Diseño O.O.

Atributos
Definición

- **Atributo** (*dato miembro* o *variable de instancia*)
  - Porción de información que un objeto posee o conoce de sí mismo.
    - Suelen ser a su vez objetos
    - Se declaran como 'campos' de la clase.
  - **V*isibilidad*** de un atributo
    - *Indica desde donde se puede acceder a él.*
    - **+** Pública (interfaz) -> desde cualquier lugar
    - **-** Privada (implementación) -> sólo desde la propia clase
    - **#** Protegida (implementación) -> desde clases derivadas
    - **~** De paquete (en Java) -> desde clases definidas en el mismo paquete

Es habitual que los atributos formen parte de la implementación (parte oculta) de una clase, pues conforman el estado de un objeto.

Atributos
Tipos de Atributo

- **Constantes** / **Variables**
  - Constante: ej., una casa se crea con un número determinado de habitaciones (característica estable):
    - `const int numHab;`
  - Variable: ej., una persona puede variar su sueldo a lo largo de su vida:
    - `int sueldo;`
- **De instancia** / **De clase**
  - De instancia**:** atributos o características de los objetos representados por la clase. Se guarda espacio para una copia de él por cada objeto creado:
    - `int nombre; // nombre de un Empleado`
  - De clase: características de una clase comunes a todos los objetos de dicha clase:
    - `static int numEmpleados; // de la clase Empleado`

Atributos
Atributos de clase

- Implican una sola zona de memoria reservada para todos los objetos de la clase, y no una copia por objeto, como sucede con las variables de instancia.
  - **Sirven para:**
    - **Almacenar características comunes (constantes) a todos los objetos**
      - Número de ejes de un coche
      - Número de patas de una araña
    - **Almacenar características que dependen de todos los objetos**
      - Número de estudiantes en la universidad
  - Un atributo estático puede ser accedido desde cualquier objeto de la clase, ya que es un dato miembro de la clase.

Indice

- Objetos
- Clases
- Atributos
- **Operaciones**
- Constructores y destructores
- UML y el diagrama de clases
- Relaciones entre objetos
- Interfaces
- Diseño O.O.

Operaciones
Definición

- **Operación** (función miembro, método o servicio de la clase)
  - Acción que puede realizar un objeto en respuesta a un mensaje. Definen el **comportamiento** del objeto.
  - Tienen asociada una ***visibilidad*** (como los atributos)
  - Pueden ser ***constantes o variables*** y ***de clase o de instancia*** (como los atributos)
  - Pueden modificar el estado del sistema (**órdenes**) o no (**consultas**)
  - **Signatura de una operación en C++**:

```
TipoRetorno
NombreClase::NombreFuncionMiembro(parametros)
```

Operador de Resolución de Ámbito (ORA)

Operaciones
Tipos de Operaciones

- **De instancia/De clase**
  - ***Operaciones de instancia***:
    - Operaciones que pueden realizar los objetos de la clase.
    - Pueden acceder directamente a atributos tanto de instancia como de clase.
    - Normalmente actúan sobre el objeto receptor del mensaje.

```
Circulo c;

c.setRadio(3);
c.getRadio();
c.pintar();
```

```
void Circulo::setRadio(double r) {
  if (r > 0.0) radio = r;
  else radio = 0.0;
}
```

Operaciones
Tipos de Operaciones

- **De instancia/De clase**
  - ***Operaciones de clase***:
    - Operaciones que acceden exclusivamente a atributos de clase.
    - No existe receptor del mensaje (a menos que se pase explícitamente como parámetro).
    - Se pueden ejecutar sin necesidad de que exista ninguna instancia, mediante el ORA.

```
class Circulo {
  private:
    static const double pi=3.141592;
  public:
    static double getRazonRadioPerimetro()
    {
      return 2*pi;
    }
…
};
```

Operaciones
Tipos de Operaciones

- **Constantes/No constantes**
  - **Órdenes** *(operaciones **no constantes**)*
    - Pueden modificar el estado del objeto receptor. No pueden ser invocadas por objetos constantes.

```
Circulo c;
c.setRadio(3); // modifica el radio de 'c'
```

  - ***Consultas*** *(operaciones **constantes**)*
    - Pueden ser invocadas por cualquier objeto
    - Dentro de ellas no se puede modificar al objeto receptor.

```
c.getRadio(); // consulta el radio de 'c'
```

Operaciones sobrecargadas

- Algunos LOO soportan la **sobrecarga** de operaciones.
  - Consiste en la existencia, dentro de un mismo ámbito, de más de una operación definida con el mismo nombre (selector), pero diferente número y/o tipo de argumentos.

```
class Circulo {
  public:
    // pinta sin relleno
    void pintar();
    // pinta con relleno
    void pintar(Color);
};
```

```
Circulo c;
c.pintar();
c.pintar(azul);
```

Operaciones
Referencia al objeto receptor

- En muchos LOO, en los métodos el receptor es un argumento implícito.
- Para obtener una referencia a él dentro de un método de instancia, existe una *pseudo-variable*:
  - En C++ y Java, se llama **`this.`**
    - En C++ es un puntero declarado así: **`const Clase* this;`**
  - En otros lenguajes, es **`self`**

- Ejemplo en C++:

receptor.selector(this ,<argumentos>)

```
class Autoreferencia {
  int x;
  public:
     A& autoref() { return *this; }
     int getX()   { return x; }
     int getX2()  { return this->x; }
     int getX3()  { return getX(); }
     int getX4()  { return this->getX(); }
};
```

Indice

- Objetos
- Clases
- Atributos
- Operaciones
- **Constructores y destructores**
- UML y el diagrama de clases
- Relaciones entre objetos
- Diseño O.O.

Constructor

- Función miembro de la clase cuyo objetivo es crear e inicializar objetos.
  - Se invoca siempre que se crea un objeto, bien directamente o bien mediante el operador **new** (en C++).
  - El enlazado de creación e inicialización asegura que un objeto nunca puede ser utilizado antes de que haya sido correctamente inicializado.
  - En Java y C++ tienen el mismo nombre que la clase y no devuelven nada (ni siquiera void).

```
class Circulo {
  public:
    Circulo();          // Constructor por defecto
    Circulo(double r); // Constructor sobrecargado
};
```

```
Circulo c();       // invocación implícita
Circulo c2(10);    // invocación implícita
Circulo *c = new Circulo();  //invocación explícita
```

Constructor

- ***Constructor por defecto:***
  - Es conveniente definir siempre uno que permita la inicialización **sin parámetros** de un objeto, donde los atributos de éste se inicialicen con valores por defecto.

```
Circulo::Circulo()
{
    radio = 1.0;
}
```

- En C++, si no se define ninguno de manera explícita, el compilador genera uno con visibilidad pública, llamado **constructor de oficio.**

```
Circulo::Circulo()
{
   /* cuerpo vacío */
}
```

Constructor de copia

- Por definición, el ***constructor de copia*** tiene ***un único argumento*** que es una ***referencia constante a un objeto de la clase***.

- Su utilidad es crear objetos que son copias (clones) de otros de la misma clase.

```
class Vector {
 public:
  Vector();
  Vector(const Vector&);

 private:
  int longi;
  int *elementos;
};
```

Constructor de copia

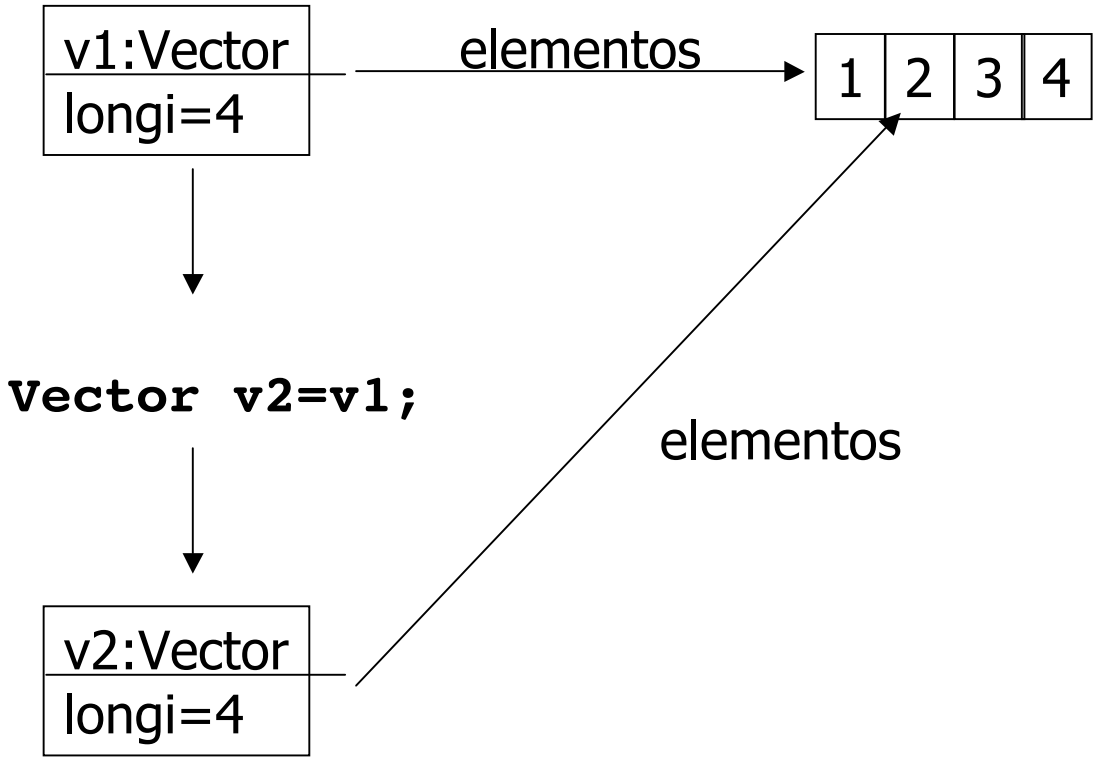
- Se invoca de forma implícita en estas situaciones:
  - Inicialización de un objeto como copia de otro:

    ```
    TCoche a(b); TCoche a=b;
    ```

  - Paso de un objeto por valor
  - Devolución de un objeto por valor
- En C++, existe un **constructor de copia de oficio** que realiza una copia bit a bit del objeto origen.

Constructor de copia de oficio

```
Vector::Vector(const Vector &v) {
{
  longi = v.longi;
  elementos = v.elementos;
}
```

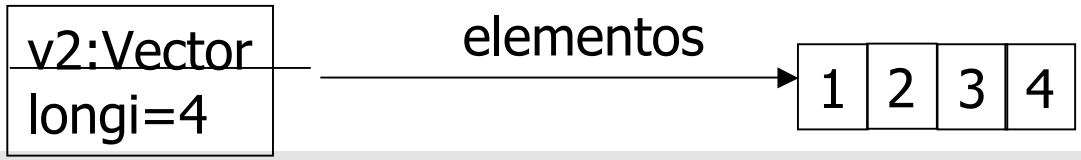


Constructor de copia explícito

```
Vector::Vector(const Vector &v)
{
  longi = v.longi;
  int i;
  elementos = new int[longi];
  for (i=0;i<longi;i++)
    elementos[i]=v.elementos[i];
}
```

¿por qué?

Vector v2=v1;



Ejercicio: clase Alumno

- Suponed que tenemos definida una clase Alumno de la siguiente manera:

```
class Alumno{
    public:
        ...
    private:
        char * nombre;
        long dni; //sin letra
};
```

- Implementad los métodos necesarios para que el siguiente código funcione correctamente.

```
int main(){
    Alumno a("Luis Pérez",33243234);
    Alumno b=a;
    b.setNombre("Pepito Grillo");
    b.setDni(58372829):
    cout << a.getNombre() << a.getDni() <<endl; // sacar datos de Luis;
    cout << b.getNombre() << b.getDni() <<endl; // sacar datos de Pepito;
}
```

Destructor (C++)

- Función miembro de la clase cuyo principal objetivo es liberar los recursos que el objeto haya reservado (p. ej., memoria dinámica).
  - En C++ tiene el mismo nombre que la clase, pero precedido por el símbolo **~**.
  - En Java y Eiffel (con garbage collectors) se denomina **`finalize()`**
- Es único, no recibe ningún argumento ni devuelve nada (ni siquiera `void`). Se suele definir con visibilidad pública.

```
Alumno::~Alumno() {
  delete [] nombre;
  nombre = NULL;
  dni = 0;
};
```

Destructor
Invocación en C++

- Variables automáticas: se invoca automáticamente cuando se libera la memoria de un objeto al salir éste de su ámbito:

```
int Suma() {Vector a; …/*aquí se invoca*/}
```

- Variables dinámicas: se deben liberar siempre explícitamente

```
Vector *a=new Vector;
delete a; a = NULL;
```

- El destructor en C++ puede ser invocado explíticamente
  - Se considera una mala práctica (desaconsejado)

```
Vector a;
…
A.~Vector();
```

Ejemplo de uso de Constructores y Destructores
Trazas

```
#include <iostream.h>
#include <stdlib.h>
#include <string>
class Traza {
    public:
        Traza(string s):texto(s){cout<<"Entrando en "<<texto<<endl;};
        ~Traza(){cout<<"Saliendo de "<<texto<<endl;};
    private:
        string texto;
};
class Dummy {
    public:
        Dummy(){Traza t("Constructor Dummy");}
        ~Dummy(){Traza t ("Destructor Dummy");}
        int getX() const{
                Traza t("Método getX Dummy");
                if (...){Traza t("cond x en método getX dummy");...}}
    private:
        int x;
};
int main () {
  Dummy d;
  d.getX();}
```

¿Cuál será la salida del programa? ¿Cómo eliminaríais la traza?

Forma canónica ortodoxa de una clase

- Todas las clases deberían definir al menos cuatro funciones importantes
  - Constructor por defecto
  - Constructor de copia
  - Operador de asignación
  - Destructor

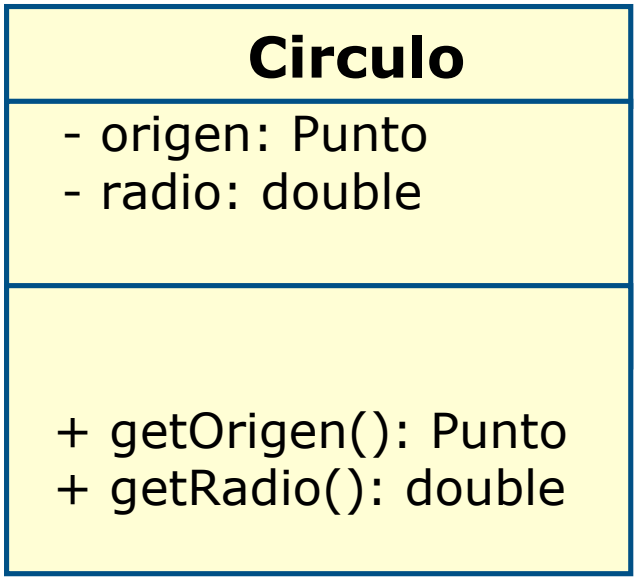
- Operaciones definidas de oficio en C++

Indice

- Objetos
- Clases
- Atributos
- Operaciones
- Constructores y destructores
- **UML y el diagrama de clases**
- Relaciones entre objetos
- Diseño O.O.

Notación UML de Clases, Atrib. y Oper.
Equivalencia C++

- Distintos LOO poseen distinta sintaxis para referirse a los mismos conceptos.
  - Las líneas de código no son un buen mecanismo de comunicación
- UML resuelve este problema y homogeneiza el modo en que se comunican conceptos OO




```
// C++
class Circulo{
public:
   Punto getOrigen();
   double getRadio();
 private:
   Punto origen;
   double radio;
};
```

Notación UML de Clases, Atrib. y Oper.
Otras equivalencias

Circulo
- origen: Punto - radio: double
+ getOrigen(): Punto + getRadio(): double

```
// Java y C#
class Circulo{
  public Punto getOrigen()
    {return origen;}
  public double getRadio()
    {return radio;}
  private Punto origen;
  private double radio;
}
```

Ejercicios
Visibilidad y Encapsulación

- Dadas las siguientes definiciones de la clase Asignatura,
  - ¿con cuál sería más rápido implementar un método de matriculación?
  - ¿cuál de ellas podemos decir que preserva el principio de encapsulación?
  - ¿Cómo podríamos hacer que el atributo nAlum fuera solo-lectura con la versión 1? ¿Y con la versión 2?
  - (propuesta) Proporcionad un ejemplo en el que se vean las ventajas de preservar el principio de encapsulación.

Asignatura
-string profesor - int nAlum - string alumnos[500]
+ void matricular(string); + int getAlumnos(); + void setProf(string)

Asignatura
+ string profesor + int nAlum + string alumnos[500]

Ejercicios
Visibilidad y Encapsulación

Asignatura
- string profesor - int nAlum - string alumnos[500]

```
int main(){
…
Asignatura a;
a.matricular("Juan");
…
If (a.getAlumnos()>200)
     a.setProf("Ana");
…
}
```

Asignatura
+ string profesor + int nAlum + string alumnos[500]

```
 int main(){
…
Asignatura a;
a.alumnos[a.nalum++]="Juan";
…
If a.nAlum>200
     a.profesor="Ana";
…
}
```

¿Qué pasaría si intentásemos compilar estos programas en un entorno que no tuviese la librería <string>?

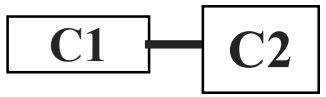
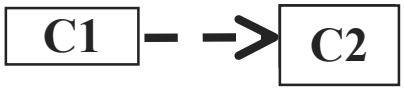
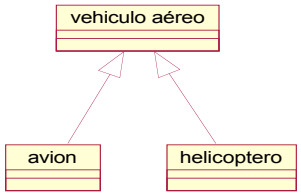
Indice

- Objetos
- Clases
- Atributos
- Operaciones
- Constructores y destructores
- UML y el diagrama de clases
- **Relaciones entre objetos**
- Diseño O.O.

Relaciones entre Clases y Objetos

- Booch: Un objeto por sí mismo es soberanamente aburrido.
- La resolución de un problema exige la colaboración de objetos.
  - Esto exige que los agentes ***se conozcan***
- El conocimiento entre agentes se realiza mediante el establecimiento de **relaciones.**
- Las relaciones se pueden establecer **entre clases** o **entre objetos**.
- Además, a nivel de objetos, podemos encontrar dos tipos de relaciones:
  - **Persistentes*:*** recogen caminos de comunicación entre objetos que se almacenan de algún modo y que por tanto pueden ser reutilizados en cualquier momento.
  - **No persistentes*:*** recogen caminos de comunicación entre objetos que desaparecen tras ser utilizados.

Relaciones entre Clases y Objetos

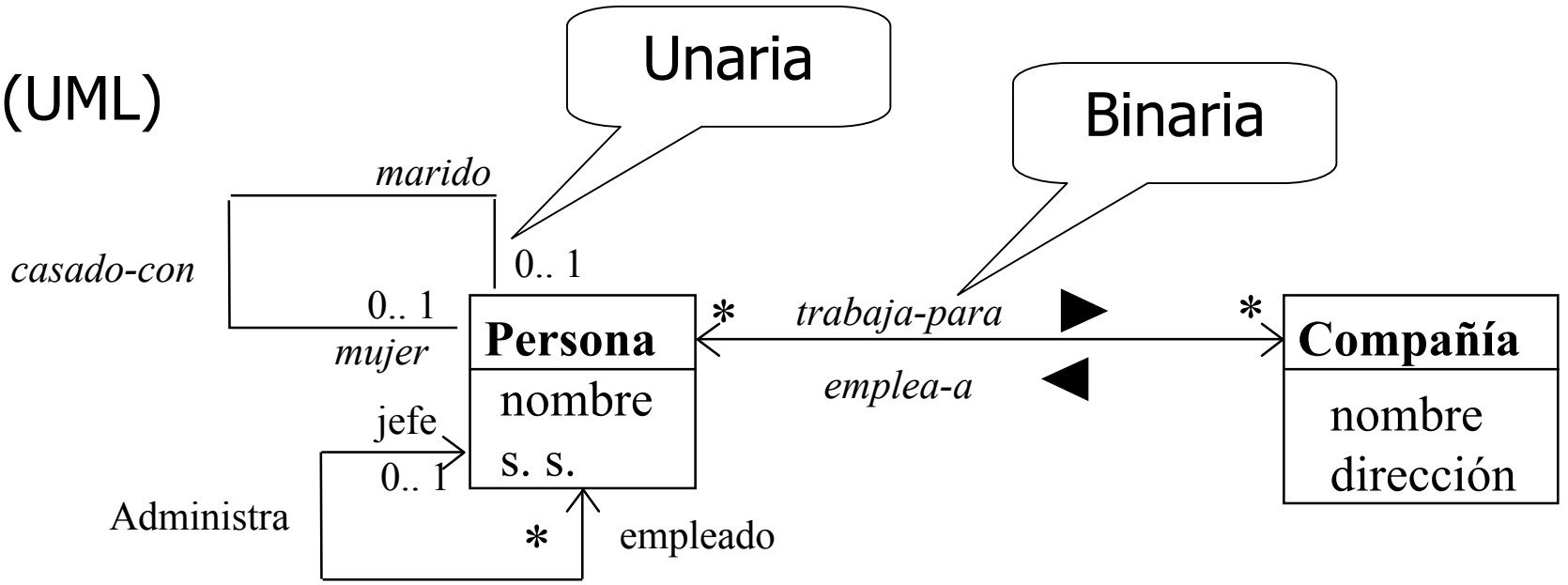
	Persistente	No persist.
Entre objetos	▪ **Asociación**  ▪ **Todo-Parte** ▪ Agregación ▪ Composición 	▪ **Uso (depend)** 
Entre clases	▪ **Generalización (Tema 3)** 	

Relaciones entre Objetos
Asociación

- Expresa una relación (unidireccional o bidireccional) entre los objetos instanciados a partir de las clases conectadas.

- El sentido en que se recorre la asociación se denomina **navegabilidad** de la asociación:

(UML)



Relaciones entre Objetos
Asociación

- Cada extremo de la asociación se caracteriza por:
  - **Rol**: papel que juega el objeto situado en cada extremo de la relación en dicha relación
    - Implementación: nombre del puntero o referencia
  - **Multiplicidad**: número de objetos mínimo y máximo que pueden relacionarse con un objeto del extremo opuesto de la relación.
    - Por defecto 1
    - Formato: *(mínima..máxima)*
      - Ejemplos (notación UML)

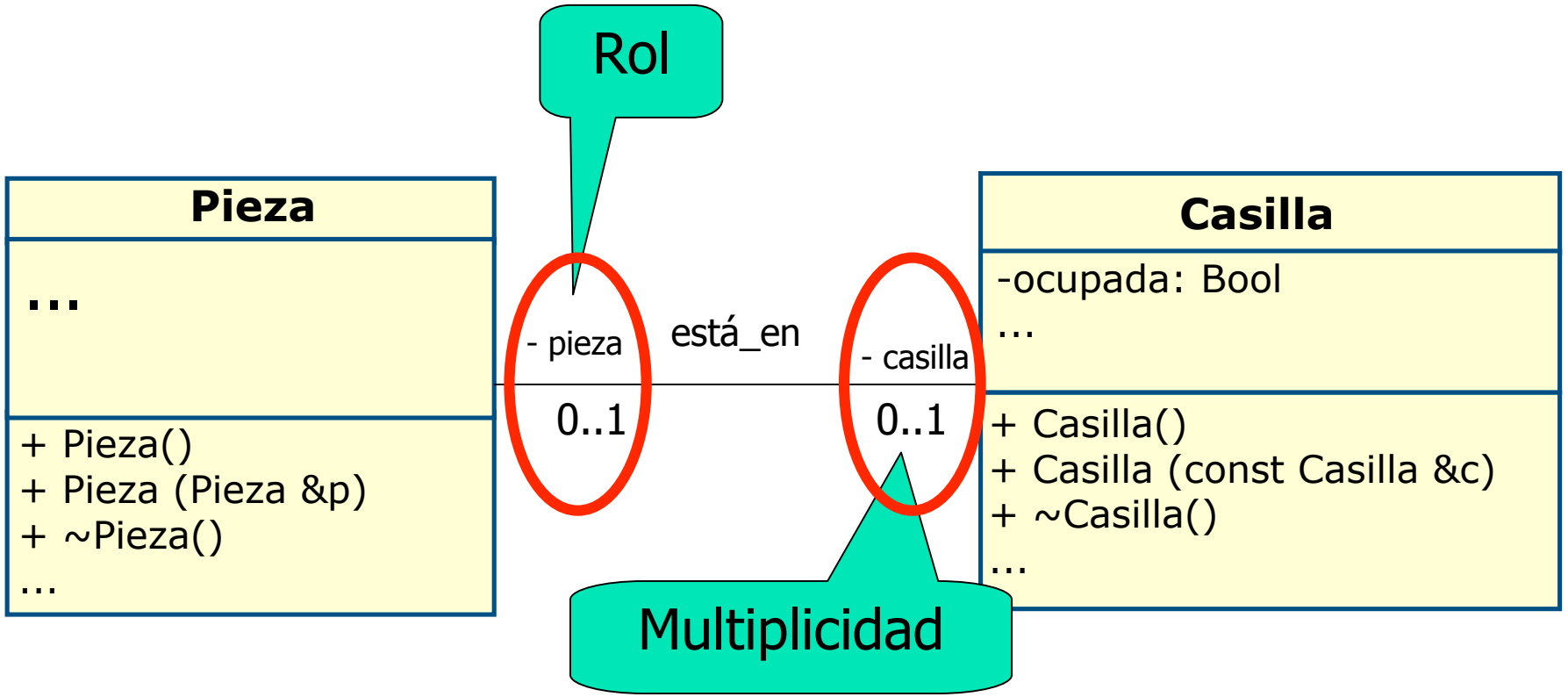
1	Uno y sólo uno (por defecto)
0..1	Cero a uno. También (0,1)
M..N	Desde M hasta N (enteros naturales)
*****	Cero a muchos
0..*	Cero a muchos
1..*	Uno a muchos (al menos uno)
1,5,9	Uno o cinco o nueve

Relaciones entre Objetos
Asociación

- 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 destrucción de la relación entre ellos y nunca la creación o destrucción del otro objeto.
- Implementación en C++
  - Un solo puntero o un vector de punteros del tamaño indicado por la cardinalidad *máxima.*
    - La decisión sobre en qué clase se introduce el nuevo dato miembro depende de la navegabilidad de la asociación.
    - Si el máximo es *: array dinámico de punteros o estructura dinámica similar.

Relaciones entre Objetos
Asociación: Ejemplo

- A partir del dibujo de la Fig., define la clase Pieza (.h)
  - Una pieza se relaciona con 0..1 casillas
  - Una casilla se relaciona con 0..1 piezas

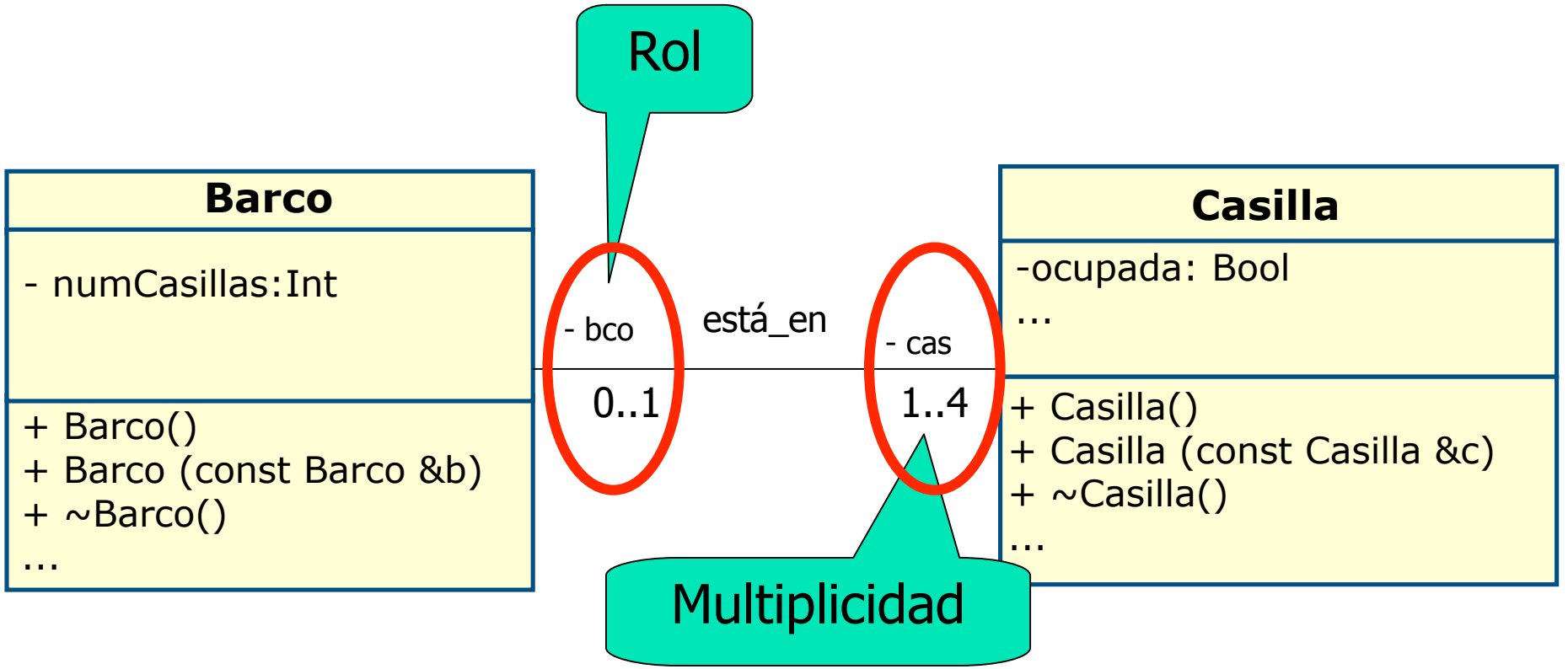


Relaciones entre Objetos
Asociación: Ejemplo

```
class Pieza{
  public:
    Pieza() {casilla=NULL;}  // Constructor por def.
    ~Pieza(){casilla=NULL;}  // Destructor
    Pieza(const Pieza &p){  // Ctor. de copia
      casilla=p.casilla;
    }
    Casilla& getCasilla()
      { return *casilla; }
    void setCasilla(Casilla &mcasilla)
      { casilla = &mcasilla; }
  private:
    Casilla* casilla;
    ...
 };
```

Relaciones entre Objetos
Asociación: Ejemplo

- A partir del dibujo de la Fig., define la clase Barco (.h)
  - Un barco se relaciona con 1..4 casillas
  - Una casilla se relaciona con 0..1 barcos



Relaciones entre Objetos
Asociación: Ejemplo

```
class Barco{
  private:
    static const int MAX_CAS=4;
    Casilla *cas[MAX_CAS];
    int numCasillas;
  public:
    Barco(){
      numCasillas=0;
      for (int x=0;x<MAX_CAS;x++)
        cas[x]=NULL;
    }
    Barco(const Barco& b){
      for (int x=0;x<MAX_CAS;x++)
        cas[x]=b.cas[x];
      numCasillas=b.numCasillas;
    }
    ~Barco(){
       for (int x=0;x<MAX_CAS;x++)
         cas[x]=NULL;
    }
};
```

- **¿Detectáis alguna posible inconsistencia que no controle este código? Modifica lo que sea necesario.**

- **¿Cambiaría en algo el código de la clase Casilla definida en el ejercicio anterior (aparte del nombre del puntero)?**

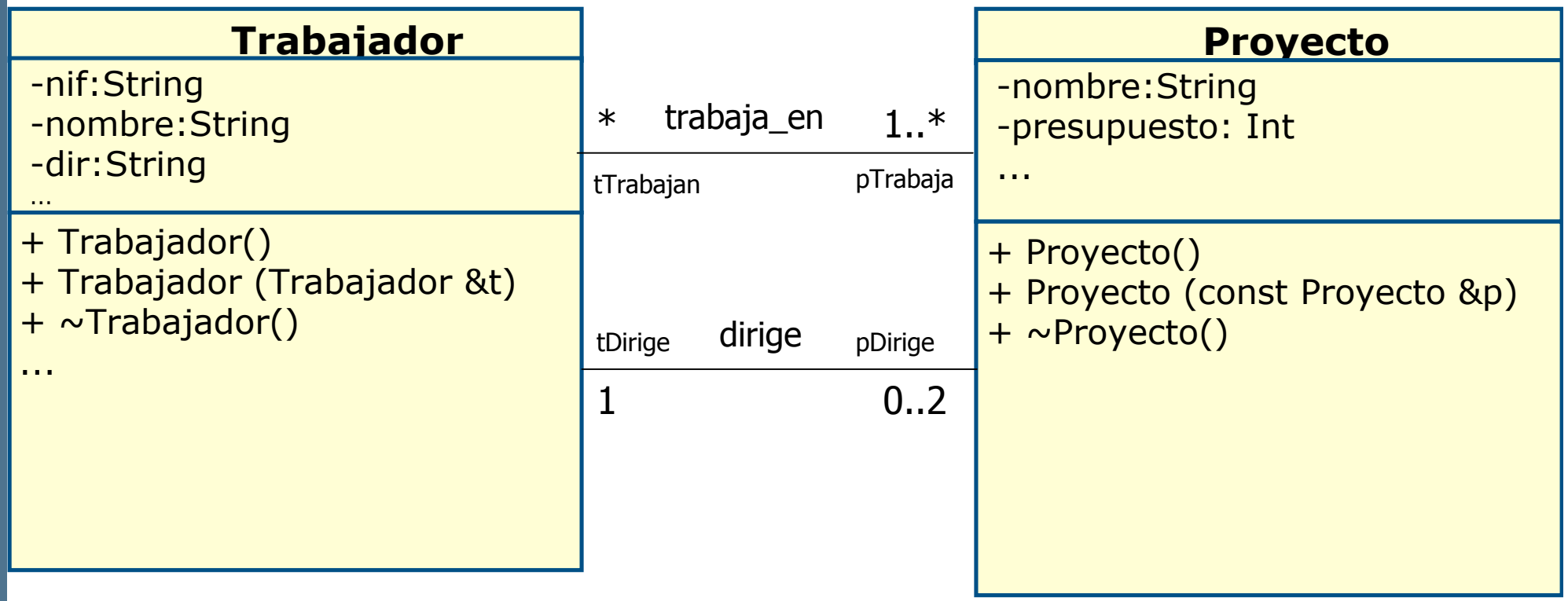
Relaciones entre Objetos
Asociación: Ejemplo

```
class Barco{
  private:
    static const int MAX_CAS=4;
    vector<Casilla*> cas;
  public:
    Barco(vector<Casilla*> c) {
      if (c.size() <= MAX_CAS)
        cas = c;
    }
    Barco(const Barco& b){
      cas = b.cas;
    }
    ~Barco(){
      cas.clear();
    }
};
```

- **Usando `vector` de la STL**

Relaciones entre Objetos
Asociación: Ejercicio

- A partir del dibujo de la Fig., define los ficheros de cabecera de la clase 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

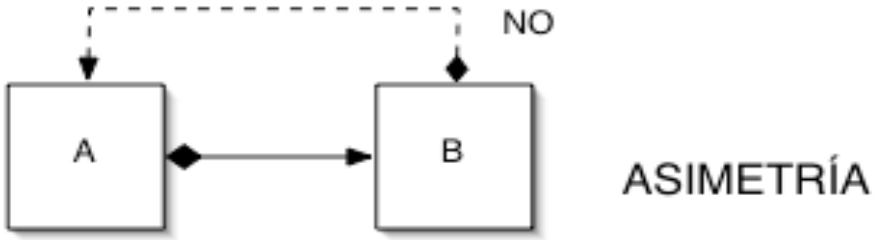


Relaciones entre Objetos
Asociación: Ejercicio

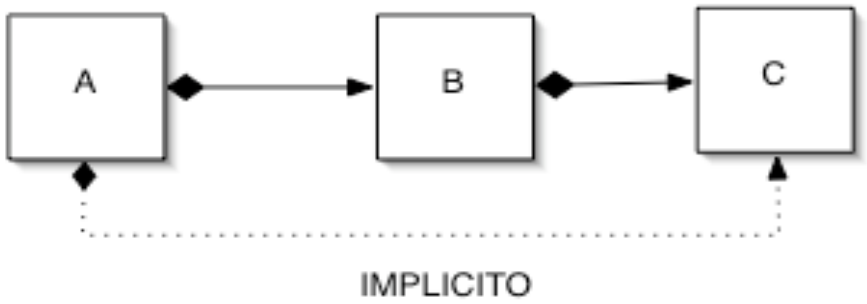
- Define el constructor, el constructor de copia y el destructor de la clase Trabajador y de la clase Proyecto.

Relaciones entre Objetos
Todo-Parte

- Una relación Todo-Parte es una relación en la que un objeto forma parte de la naturaleza de otro.
  - Se ve en la vida real: 'A está compuesto de B', 'A tiene B'
- Asociación vs. Todo-Parte
  - La diferencia entre asociación y relación todo-parte radica en la **asimetría** y **transitividad** presentes en toda relación todo-parte.



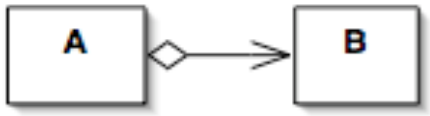
ASIMETRÍA



TRANSITIVIDAD

Relaciones entre Objetos
Todo-Parte

- A nivel teórico se distinguen dos tipos de relación todo-parte:
  - **Agregación**



    - Asociación binaria que representa una relación todo-parte ('tiene-un', 'es-parte-de', 'pertenece-a')

  - **Composición**
    - Agregación 'fuerte':

      - Una instancia 'parte' está relacionada, como máximo, con una instancia 'todo' en un instante dado.
      - Cuando un objeto 'todo' es eliminado, también son eliminados sus objetos 'parte' (Es responsabilidad del objeto 'todo' disponer de sus objetos 'parte')

Relaciones entre Objetos
¿Agregación o composición?

- Agregación vs. Composición a nivel práctico

  - Utilizaremos **agregación** siempre que deseemos que la relación se materialice mediante referencias (lo que permite que p. ej. un componente esté referenciado en más de un compuesto)
    - Así, a nivel de implementación una agregación no se diferencia de una asociación binaria
    - Ejemplo: Un equipo y sus miembros
  - Utilizaremos **composición** cuando deseemos que la relación se materialice en una inclusión por valor (lo que implica que un componente está como mucho en un compuesto, pero no impide que haya objetos componentes no relacionados con ningún compuesto)
    - Si destruyo el compuesto destruyo sus componentes
    - Ejemplo: Un libro y sus capítulos

Relaciones entre Objetos
Caracterización Todo-Parte

1. ¿Puede el objeto parte comunicarse directamente con objetos externos al objeto agregado?
   - No => inclusiva (+ Composición)
   - Sí => no inclusiva (+ Agregación)
2. ¿Puede cambiar la composición del objeto agregado una vez creado?
   - Sí => compuesto dinámico (+ Agregación)
   - No => compuesto estático (+ Composición)
3. ¿Puede el objeto parte cambiar de objeto agregado?
   - Sí => componente dinámico (+ Agregación)
   - No => componente estático (+ Composición)
4. **¿Puede el objeto parte ser compartido por más de un objeto agregado?**
   - No => **disjunta** (+ Composición)
   - Sí => no disjunta (+ Agregación)
5. **¿Puede existir un objeto parte sin ser componente de un objeto agregado?**
   - Sí => flexible (+ Agregación)
   - No => **estricta** (+ Composición)

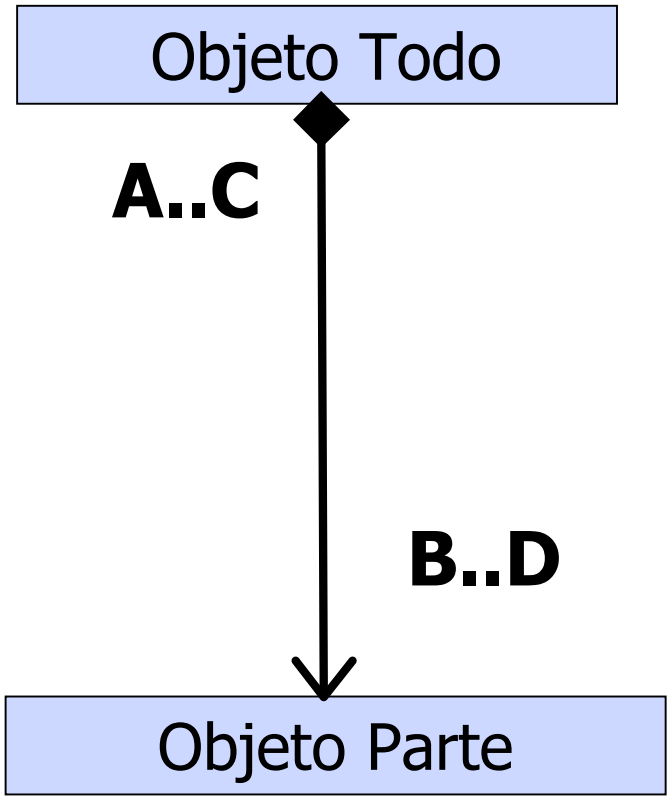
Relaciones entre Objetos
Caracterización Todo-Parte

6. ¿Cuántos objetos de una clase componente puede tener asociados un objeto agregado?
   - Más de uno => multivaluada (indiferente)
   - Máximo uno => univaluada (indiferente)

7. ¿Puede el objeto agregado no tener objetos de una clase componente en algún instante?
   - Sí => con nulos permitidos (+ Agregación)
   - No => con nulos no permitidos (+ Composición)

Relaciones entre Objetos
Caracterización Todo-Parte

- Las caracterizaciones 4, 5, 6 y 7 están incluidas en el concepto más amplio de multiplicidad



A: Multiplicidad Mínima (5)
- 0 → flexible
- > 0 → estricta

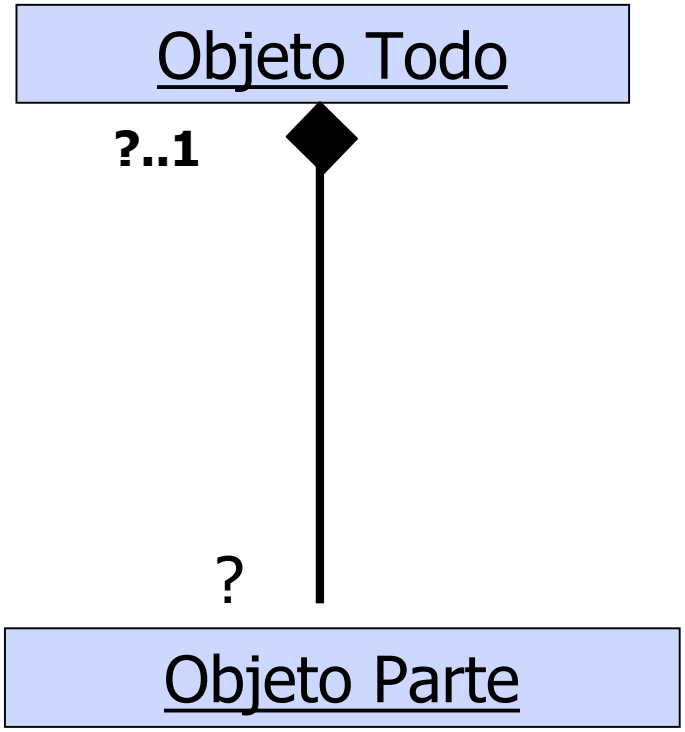
B: Multiplicidad Mínima (7)
- 0 → nulos permitidos
- > 0 → nulos no permitidos

C: Multiplicidad Máxima (4)
- 1 → disjunto
- > 1 → no disjunto

D: Multiplicidad Máxima (6)
- 1 → univaluado
- > 1 → multivaluado

Relaciones entre Objetos
Caracterización composición

- Si tenemos en cuenta las restricciones 4 y 5, una composición se caracteriza por una cardinalidad máxima de 1 en el objeto compuesto



Relaciones entre Objetos
Caracterización composición

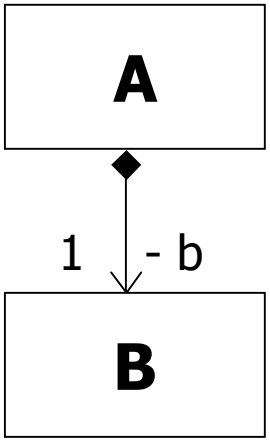
- Algunos autores consideran la composición como una agregación **disjunta** (el objeto parte no puede ser compartido por más de un objeto agregado en un momento determinado) y **estricta** (no puede existir un objeto parte que no pertenezca a un objeto agregado).

- Otros autores sin embargo no imponen esta restricción, y consideran que la única diferencia entre ambos conceptos radica en su implementación; así una composición sería una 'Agregación por valor'

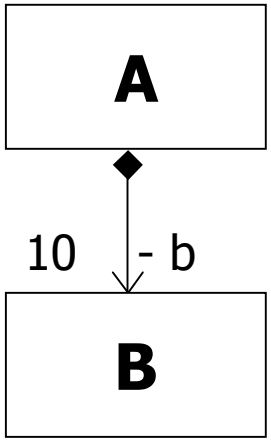
Relaciones todo-parte
Implementación (C++)

- Implementación en C++ de la relación TODO-PARTE: mediante *layering*
  - Una clase A contiene/agrega elementos de una clase B cuando la clase A incluye en su declaración algún dato miembro de tipo B.
  - Agregación: Se implementa como una asociación
  - Composición: Dijimos que
    - Es responsabilidad del objeto 'todo' disponer de sus objetos 'parte'

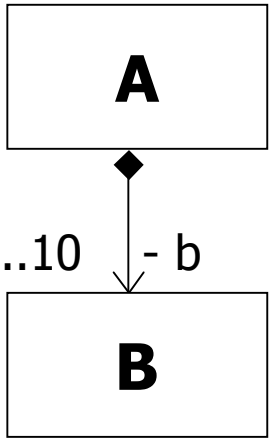
Relaciones todo-parte
Implementación de la composición (C++)



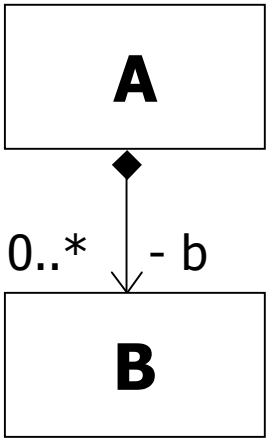

```
class A {
  private:
    B b;
// b es un
// subobjeto
…};
```




```
class A {
  private:
    B b[10];
…};
// o B *b[10];
// si B tiene
// clases derivadas
```

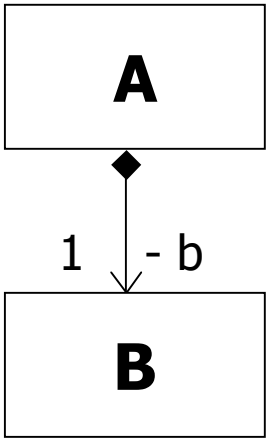



```
class A {
  private:
    B *b[10];
    int num_b;
…};
```

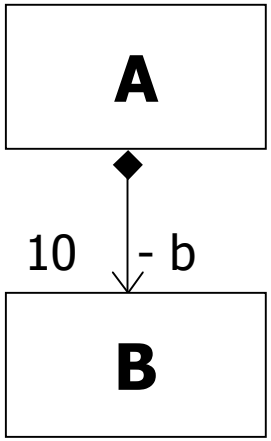



```
class A {
  private:
    B **b;
    int num_b;
…};
```

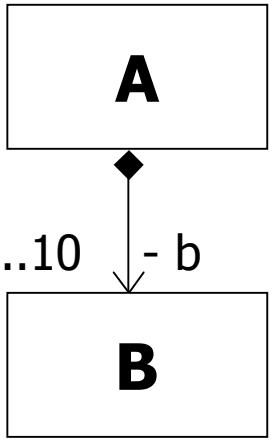
Relaciones todo-parte
Implementación de la composición (C++)



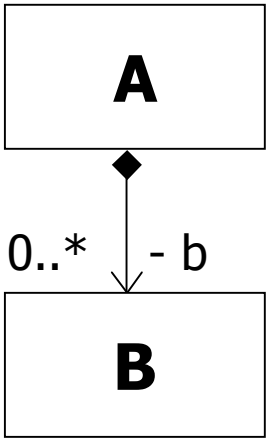

```
class A {
  private:
    B b;
// b es un
// subobjeto
…};
```




```
class A {
 private:
  static int MAXB=10;
  vector<B> b;
 public:
  A() : b(MAXB)
  {…}
…};
// o vector<B*> b;
// si B tiene
// clases derivadas
```

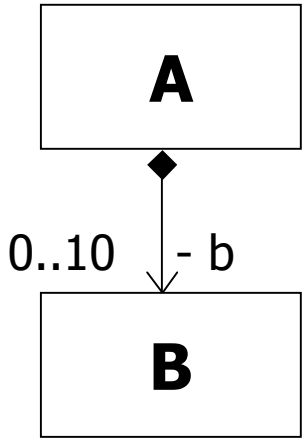



```
class A {
 private:
  static int MAXB=10;
  vector<B*> b;
…};
// también, según
// la aplicación,
// vector<B> b;
```




```
class A {
 private:
  vector<B*> b;
…};
```

Relaciones todo-parte
Implementación de la composición (C++)



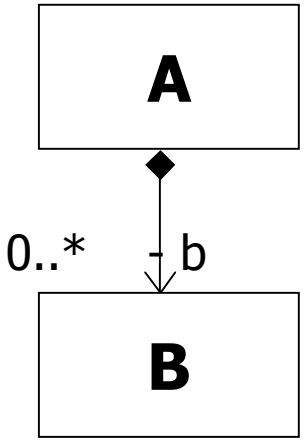

```
class A {
 private:
  static int MAXB=10;
  vector<B*> b;
…};
```

```
A::A() : b() {}


A::addB(const B& unB) {
…  if (b.size()<MAXB)
     b.push_back(new B(unB));
…}

 A::~A() {
   …
   for (int i=0; i<b.size(); i++)
   {    delete b[i]; b[i]=NULL; }
   b.clear();
 …}
```

Relaciones todo-parte
Implementación de la composición (C++)




```
class A {
 private:
  vector<B*> b;
…};
```

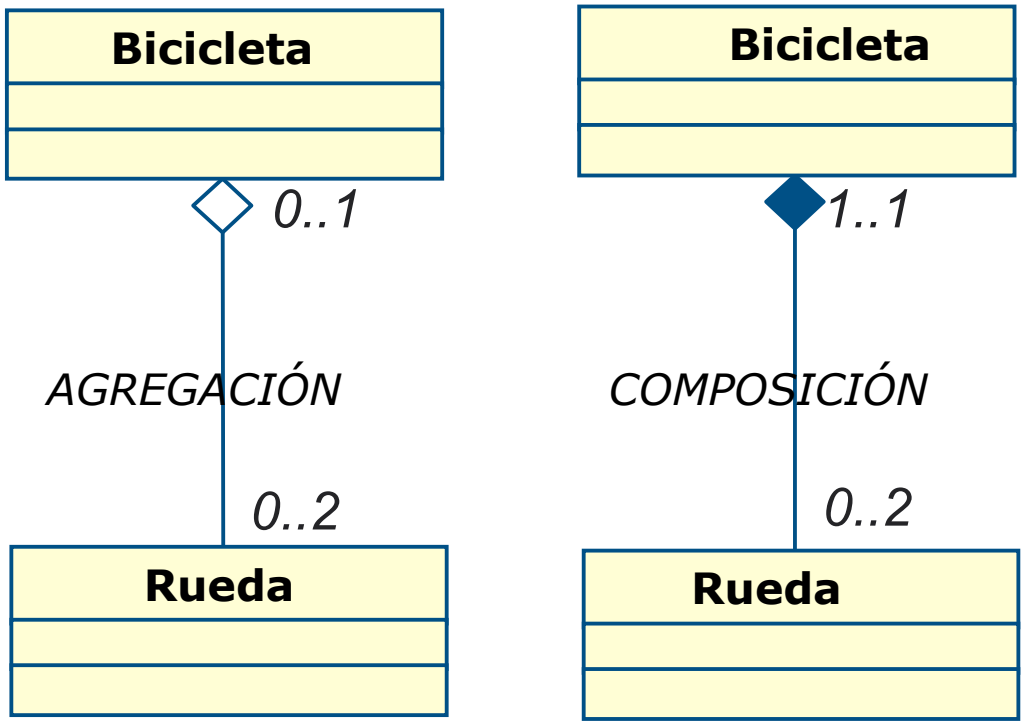
```
A::A() : b() {}


A::addB(const B& unB) {
… b.push_back(new B(unB));
…}
```

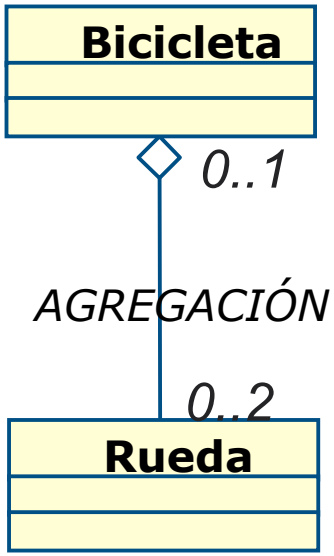
```
A::~A() {
   …
   for (int i=0; i<b.size(); i++)
   {    delete b[i]; b[i]=NULL; }
   b.clear();
 …}
```

Relaciones todo-parte
Ejemplo Bicicleta

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



Relaciones todo-parte
Ejemplo Bicicleta

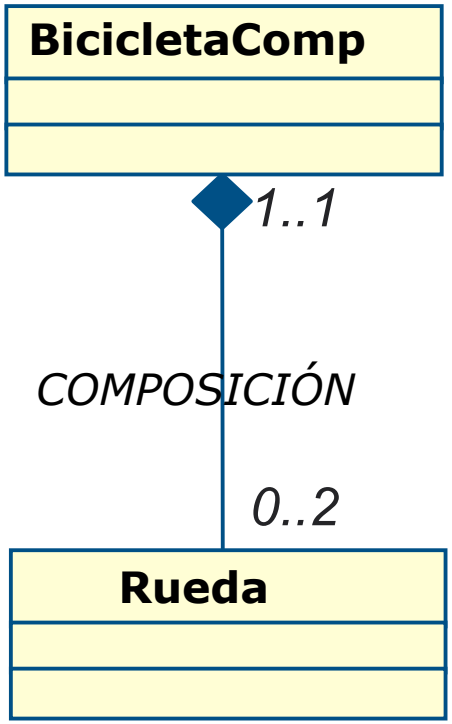



```
int main(){
Rueda *r, *r2, *r3;
Rueda *r= new Rueda("primera");
Rueda *r2= new Rueda("segunda");
Rueda *r3=new Rueda("tercera");
Bicicleta b1(r1,r2);
b1.cambiarRueda(0,r3);
//b1.~Bicicleta();
//¿Qué pasaría con las ruedas?
}
```

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

class Bicicleta {
private:
  vector<Rueda*> r;
  static const int MAXR=2;
public:
  Bicicleta(Rueda *r1, Rueda *r2){
    r.push_back(r1);
    r.push_back(r2);
  }
  void cambiarRueda(int pos, Rueda *raux){
    if (pos>=0 && pos<MAXR) r[pos]=raux;
  }
  ~Bicicleta(){
    r.clear(); //no destruye las ruedas
  }
};
```

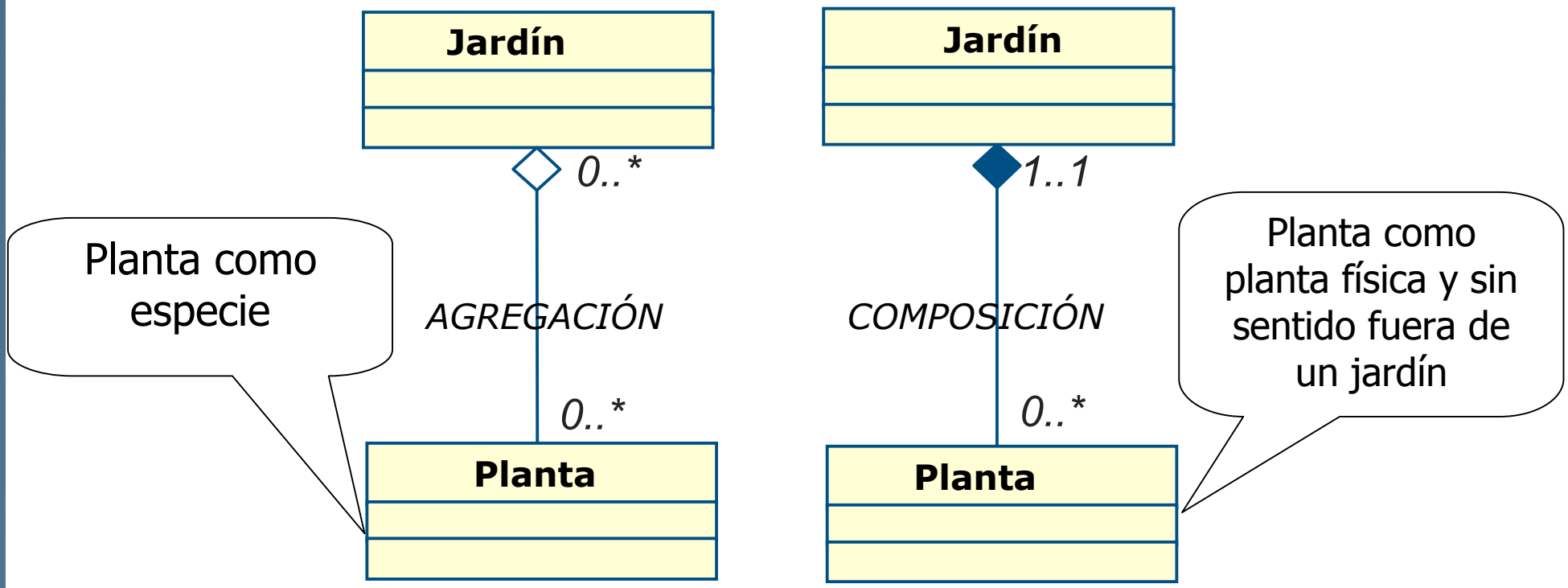
Relaciones todo-parte
Ejemplo Bicicleta




```
class BicicletaComp{
    private:
        static const int MAXR=2;
        vector<Rueda*> r;
    public:
        BicicletaComp(string p,string s){
          r.push_back(new Rueda(p));
          r.push_back(new Rueda(s));
         }
        ~BicicletaComp(){
          for (int i=0; i<r.size(); i++)
            {     delete r[i]; r[i]=NULL; }
          r.clear();
         }
};
int main(){
  BicicletaComp b2("tercera","cuarta");
  BicicletaComp b3("tercera","cuarta");
  //son ruedas distintas aunque con el mismo nombre
}
```

Relaciones todo-parte
Ejemplo Jardín

Observad el diferente significado de la clase Planta en función del tipo de relación que mantiene con Jardín



Relaciones todo-parte
Ejemplo Jardín

Supongamos que tenemos el código

```
class Planta {
 public:
   Planta(string, string);
   ~Planta();
   string getNombre();
   string getEspecie();
   string getTemporada();
   void
    setTemporada(string);
 private:
    string nombre;
    string especie,
    string temporada;
};
```

```
class Jardin {
 public:
   Jardin(string);
   ~Jardin();
   Jardin(const Jardin &);
   void Plantar(string, string,
    string);
   void Arrancar(string, string);
private:
   vector<Planta *> p;
   string emplazamiento;
};
```

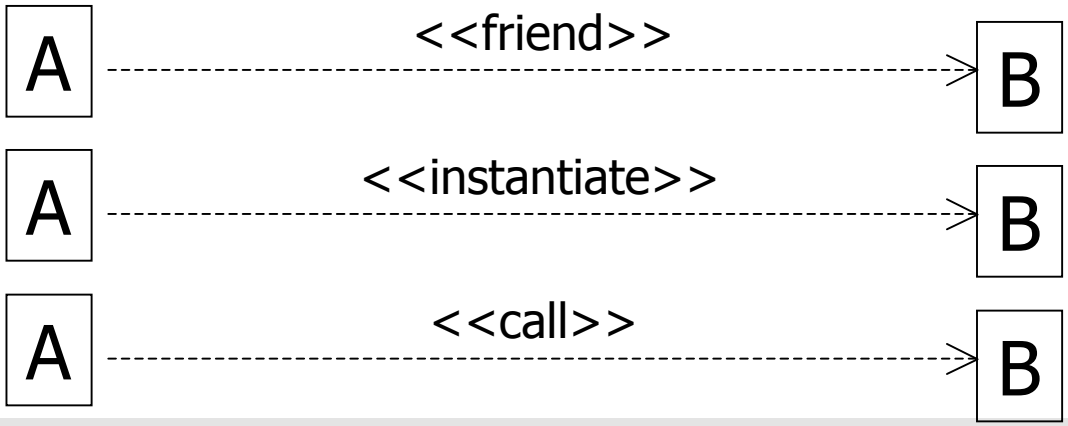
```
Jardin::Jardin(string lugar){
  emplazamiento=lugar;
}

Jardin::~Jardin(){
  for(int i=0;i<p.size();i++)
  {
      delete(p[i]);
      p[i]=NULL;
  }
  p.clear();
}
```

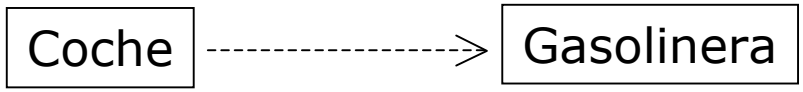
¿Qué relación existe entre Jardín y planta?

Relaciones entre Objetos
Relación de Uso (Dependencia)

- Una clase A **usa** una clase B cuando no contiene datos miembros del tipo especificado por la clase B pero:
  - Utiliza alguna **instancia de la clase B como parámetro** (o variable local) en alguno de sus métodos para realizar una operación.
  - **Accede a sus variables privadas** (clases amigas)
  - Usa algún **método de clase** de B.
- En UML este tipo de relaciones se diseñan mediante **dependencias**.



Relaciones entre Objetos
Relación de Uso (Dependencia): Ejemplo

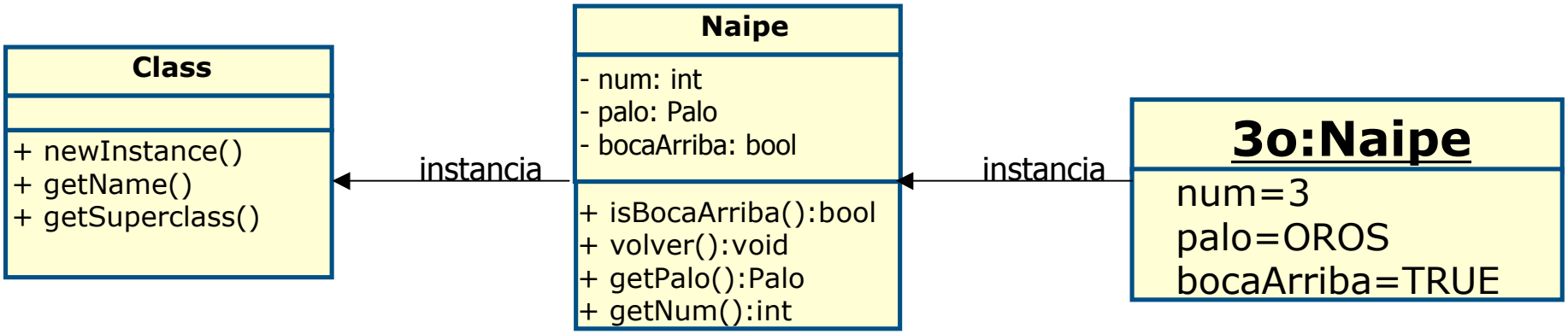


- Supongamos que no nos interesa guardar las gasolineras en las que ha repostado un coche: no es asociación.
- Sin embargo sí existe una interacción:

```
float
TCoche::respostar(Gasolinera& g, float litros){
 float importe=g.dispensarGaso(litros,tipoC);
 if (importe>0.0) //si éxito dispensar
   lGaso=lGaso+litros;
 return (importe);
}
```

Metaclases

- Existen métodos que se asocian no con métodos sino con clases
  - `new, delete`
  - Métodos estáticos
- En Smalltalk, Java y otros lenguajes una clase es una instancia de otra clase, llamada **metaclase**.
  - Por tanto, las clases en sí mismas pueden responder a ciertos mensajes, como es el mensaje de creación de objetos new



Indice

- Objetos
- Clases
- Atributos
- Operaciones
- Constructores y destructores
- UML y el diagrama de clases
- Relaciones entre objetos
- **Diseño O.O.**

El diseño de aplicaciones OO

- Objetivo principal: conseguir crear un universo de agentes lo más independientes posible entre sí.
  - Una posible técnica a utilizar es la denominada ***Diseño Dirigido por Responsabilidades*** (*responsibility-driven design*) [Wirfs-Brock].
  - Recordemos: responsabilidad implica **no interferencia** por parte de otros objetos de la comunidad.

El diseño de aplicaciones OO
Interfaz e Implementación

- El énfasis en caracterizar un componente software por su comportamiento tiene una consecuencia fundamental: separación de interfaz (*qué*) e implementación (*cómo*).

- ***Principios de Parnas***
  - El desarrollador de un componente sw C debe proporcionar al usuario de C toda la info necesaria para hacer un uso efectivo de los servicios de C y no debería proporcionar ninguna otra información.

  - El desarrollador de un componente sw C debe recibir toda la información necesaria para realizar las responsabilidades necesarias asignadas al componente y ninguna otra información.

El diseño de aplicaciones OO
Métricas de calidad

- **Acoplamiento**: relación entre componentes software
  - Interesa bajo acoplamiento. Éste se consigue moviendo las tareas a quién ya tiene habilidad para realizarlas.
- **Cohesión**: grado en que las responsabilidades de un solo componente forman una unidad significativa.
  - Interesa alta cohesión. Ésta se consigue asociando a un solo componente tareas que están relacionadas en cierta manera, p. ej., acceso a los mismos datos.
- Componentes con bajo acoplamiento y alta cohesión facilitan su utilización y su interconexión

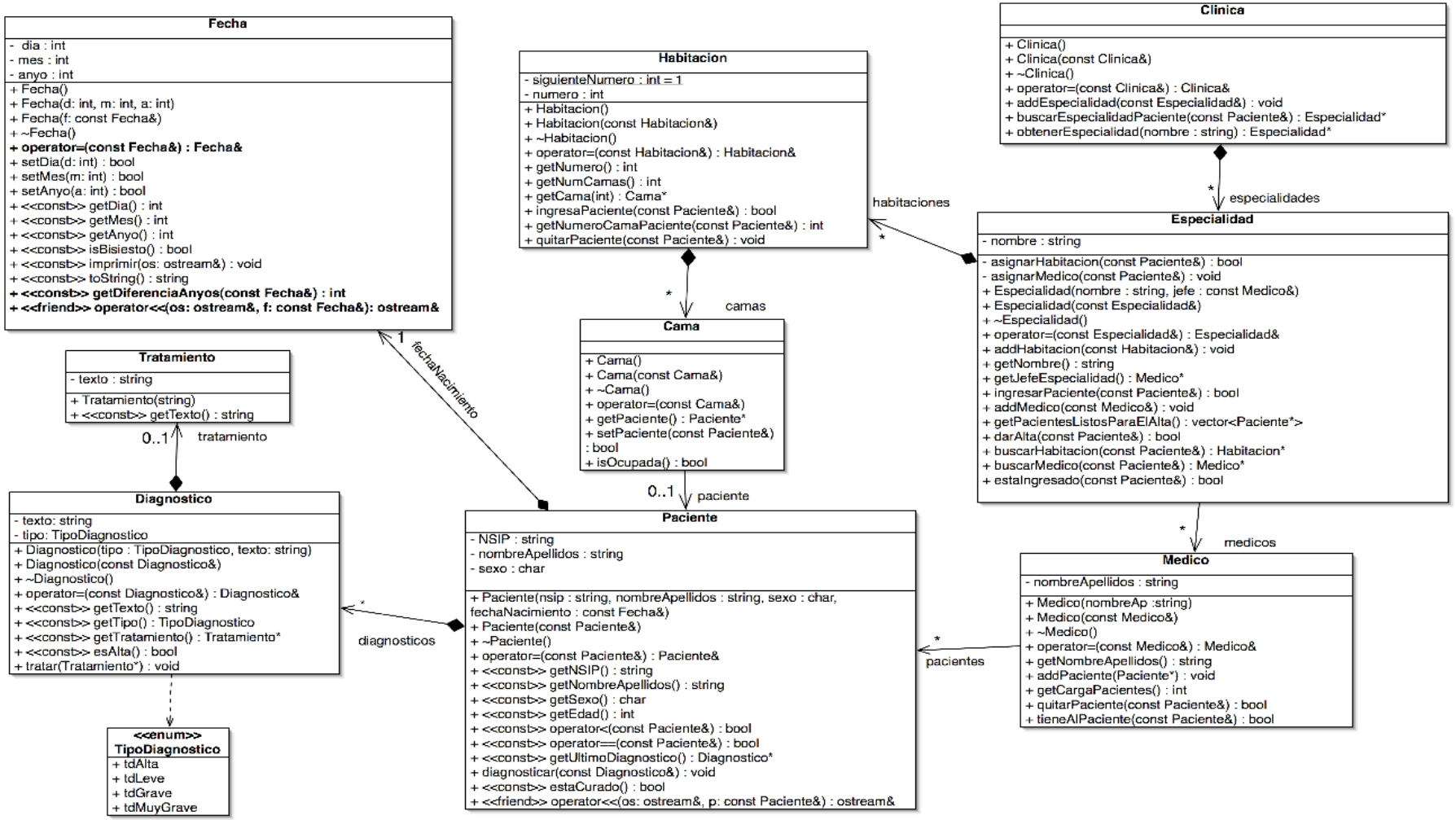
El diseño de Aplicaciones OO
Diagrama de Clases (UML)

- Un diagrama de clases define las clases, sus propiedades y cómo se relacionan unas con otras.

- Proporciona una vista estática de los elementos que conforman el software.
  - Se ven las partes que componen la aplicación y cómo se ensamblan, pero no cómo se comportan cuando el sistema se ejecuta.

- El diagrama de clases es el diagrama principal de análisis y diseño.

El diseño de Aplicaciones OO
Diagrama de Clases (UML)



Bibliografía

- Cachero et. al.
  - ***Introducción a la programación orientada a Objetos***
    - Capítulo 2
- T. Budd.
  - ***An Introduction to Object-oriented Programming, 3rd ed.***
    - Cap. 4 y 5; cap. 6: caso de estudio en varios LOO
- G. Booch.
  - ***Object Oriented Analysis and Design with Applications***
    - Cap. 3 y 4
- G. Booch et. al.
  - ***El lenguaje unificado de modelado***. Addison Wesley. 2000
    - Sección 2 (cap. 4-8)