Programación Orientada a Objetos

Matilde Fernández Azuela

Madrid, septiembre de 2008

Relaciones entre clases

El lenguaje de modelado estándar

UML es un lenguaje gráfico formal que orienta la realización de diagramas a fin de que éstos representen sin ambigüedad los diferentes aspectos a definir en el desarrollo del sofware

La especificación del lenguaje está disponible en http://www.uml.org/ .

El lenguaje de modelado estándar. Tipos de diagramas

- **Modelado de datos.**

  Diagramas de clases, objetos, componentes, despliegue, paquetes.

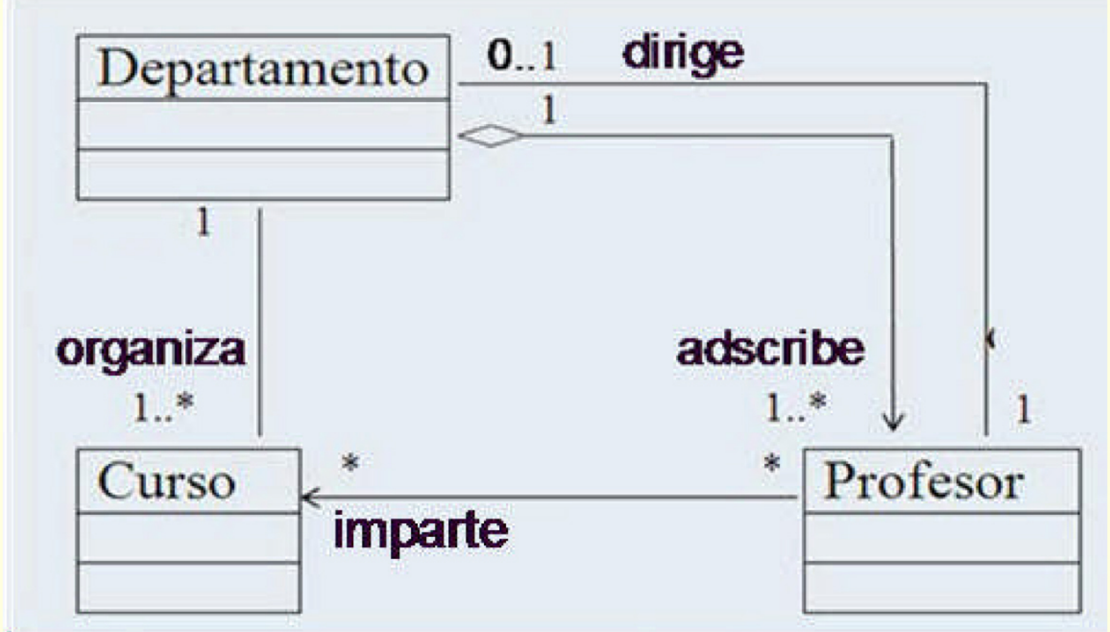
- **Modelado de comportamiento**

  Diagrama de casos de uso, actividades, estados, colaboración, secuencia.

El lenguaje de modelado estándar. Diagramas de clases

El diagrama de clases resulta fundamental en el análisis y diseño del sistema. El diagrama de clases es una representación gráfica que presenta las clases con sus relaciones estructurales y de herencia.

Por ej.:



Perspectivas para un diagrama de clases

- **Conceptual**, considera las clases, sus relaciones y la cardinalidad pero puede no detallar la navegabilidad
- **De especificación**, añade los métodos públicos a la perspectiva conceptual y adquiere relevancia la navegabilidad

Representación de clases

Símbolos visibilidad:

Publico +
Privado -
Protegido #

<Nombre de la clase>
<Atributos>
<Operaciones>

Sintaxis de los atributos:
visibilidad nombre: tipo = valor_inicial

Sintaxis de las operaciones:
visibilidad nombre (lista_parámetros): tipo_retorno

Forma simplificada

<Nombre de la clase>

Ejemplo: la clase circulo en UML

Circulo
-radio: **double** -x: **double** -y: **double**
+Area(): **double** +Perímetro(): **double**

Formas simplificadas de representación:

Circulo

Circulo

La colaboración entre objetos

Los objetos contribuyen al comportamiento de un sistema colaborando entre si. La colaboración se logra a través de las relaciones definidas entre las clases.

En una jerarquía de clases, las subclases de una clase base heredan todas sus relaciones y, además, cada subclase puede participar en relaciones adicionales.

Relaciones entre clases

Asociación

Agregación

Composición

Generalización

Dependencia

Rol: semántica de la asociación

Cardinalidad

0 a muchos: *

uno a muchos: 1..*

m a n: m..n

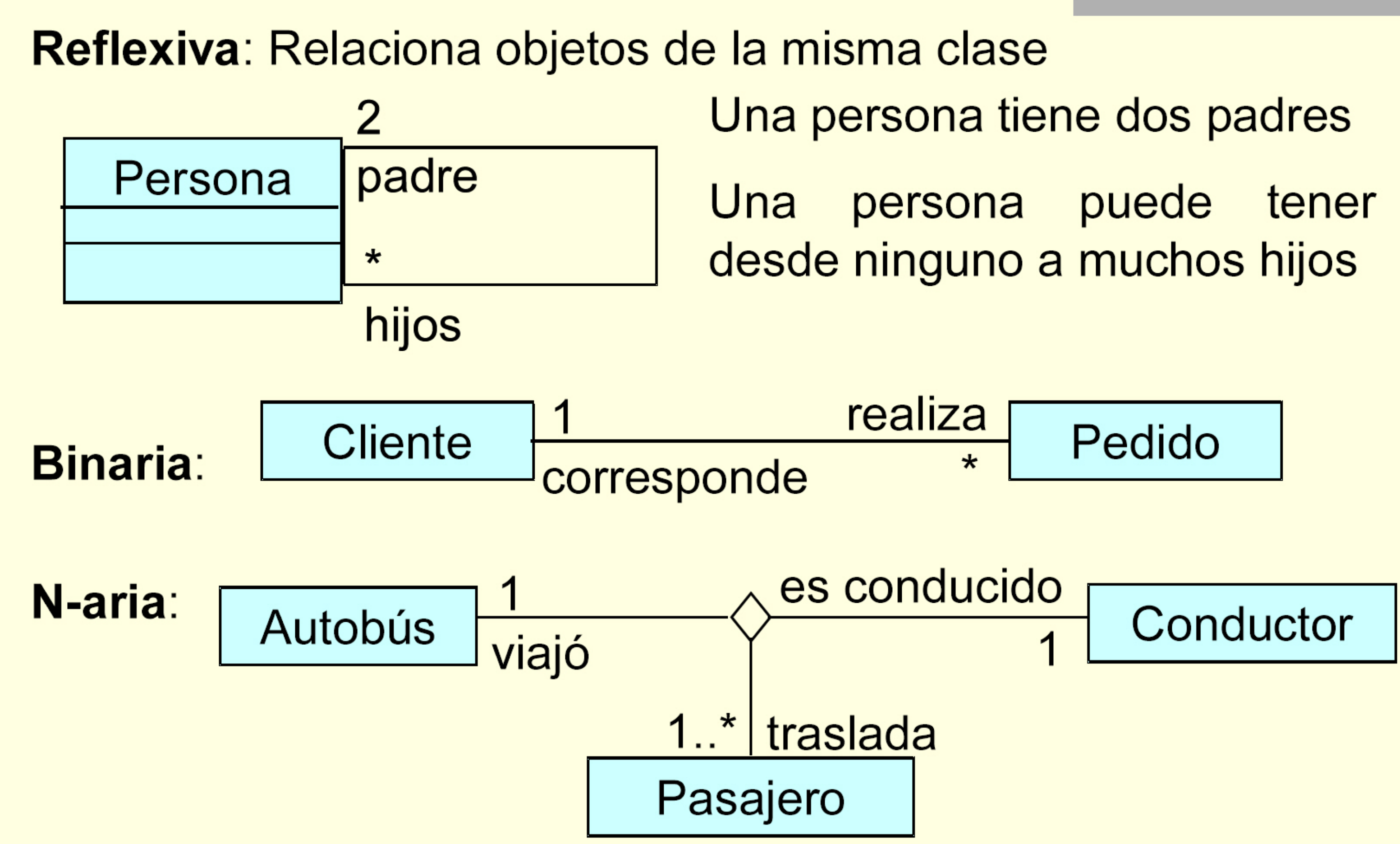
número fijo: n

Relación de asociación

Una asociación es una relación entre clases que representa una conexión entre sus objetos.

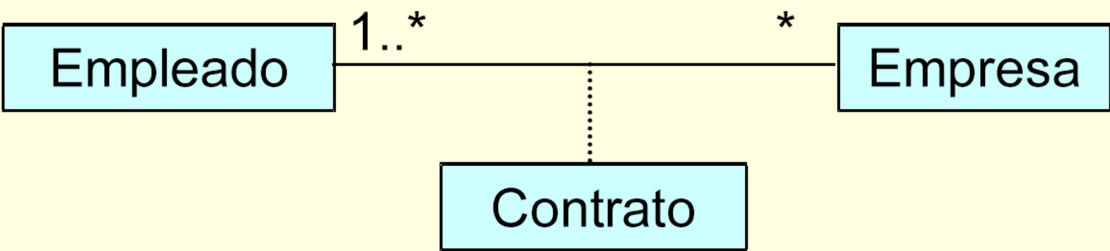
- Una asociación puede tener etiquetas que indiquen los roles..
- La comunicación (navegabilidad) puede ser tanto uni como bi-direccional. La comunicación implica que la clase origen puede referirse a los métodos de la clase destino (normalmente por medio de un atributo en la clase origen que constituye una referencia a la clase destino)
- La cardinalidad (multiplicidad) debe ser siempre especificada, indica cual es el número máximo y mínimo de instancias de una clase que pueden asociarse a una instancia de otra y aclara si una asociación es obligatoria u opcional

Tipos de asociación: reflexiva, binaria, n-aria

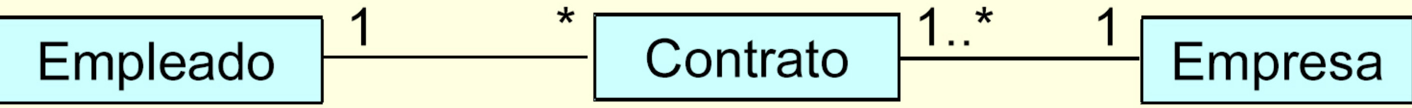


Clase asociación

Si una asociación necesita atributos u operaciones propias deberá definirse como una clase, surgiendo así el concepto de clase asociación.



Las clases asociación evolucionan durante la fase de diseño a una clase interpuesta entre otras dos con multiplicidad 1 desde ella a las clases ligadas



Implementación de asociaciones

- En relaciones con multiplicidad 1 ó 0..1 se declara como atributo (dato miembro) un puntero a un objeto de la clase relacionada.
- Para relaciones con multiplicidad superior a 1 se declaran arrays, vectores o listas de punteros a objetos de la clase relacionada.
- La conexión entre los objetos relacionados se establece pasando como parámetro un puntero o referencia de un objeto existente a una operación del otro
- Los objetos son creados y destruidos de manera independiente.

Ejemplo de implementación de asociaciones

```
class Empleado
{  //arrays dinámicos de
   //punteros a objeto
   Contrato** contratos;
   Empresa** empresas;
  public:
   Empleado();
   //...                 };
```

```
class Empresa
{  //arrays dinámicos de
   //punteros a objeto
   Contrato** contratos;
   Empleado** empleados;
  public:
   Empresa();
   //...                 };
```

```
class Contrato
{ //puntero a objeto
  Empresa* unaEmpresa;
  //puntero a objeto
  Empleado* unEmpleado;
 public:
  Contrato();
  //...                 };
```

Relación de dependencia

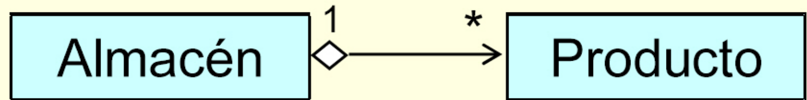
Implica una colaboración temporal, surgida con determinado propósito, que requiere una conexión entre las clases más débil que la asociación.

Las clase dependiente conoce a la independiente (recibe como parámetro, declara como variable local, etc. un objeto de ella) y utiliza sus métodos en una determinada tarea. Cambios en la clase independiente afectan a la dependiente.

Ej.: la clase persona utiliza la clase fecha localmente, en uno de sus métodos, para calcular la edad

Relación de agregación

La agregación es una forma especial de asociación que denota posesión, representa la relación de un todo con sus partes.



- Es una relación transitiva y antisimétrica.
- El todo y las partes son objetos con tiempos de vida independientes.
- Se implementa de forma análoga a la asociación. Las diferencias entre estos dos tipos de relaciones no son a nivel de código y solo residen en la existencia o no existencia de la posibilidad de considerar una clase como “todo” y la otra como “parte” de la anterior.

Relación de composición

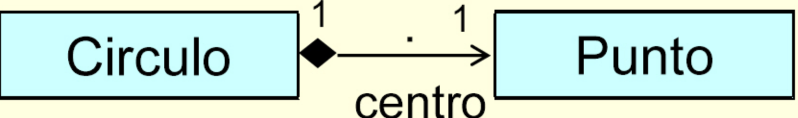
La composición es es un tipo de agregación donde el todo asume absoluta responsabilidad sobre sus partes.

- el objeto contenido es parte vital del que lo contiene.
- si la cardinalidad es 1 la parte y el todo se crean al mismo tiempo.
- los objetos contenidos no pueden ser compartidos
- al destruir el objeto compuesto deben destruirse sus partes o bien traspasar su responsabilidad a otros objetos.

Implementación de la composición I

- El componente se declara dentro del objeto compuesto y se inicializa en el constructor.
- Como el objeto contenido es un dato miembro del compuesto la creación y destrucción de ambos se encuentra ligada.

Ejemplo: todo circulo tiene un centro



Implementación de la composición II

```
class Punto
{ float x, y;
  public:
  Punto(float a=0, float b=0)
  { x = a; y = b; }
};
```

```
class Circulo
{ Punto centro;
  float radio;
  public:
  Circulo(float abcisa, float ordenada, float radio)
    : centro(abcisa, ordenada)
   { this->radio = radio; }
};
```

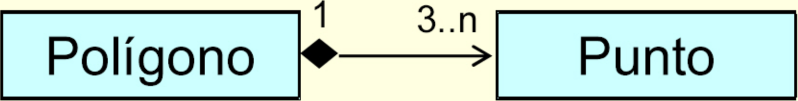
Implementación de la composición III

```
class Circulo
{ Punto* centro;    //de forma dinámica
  float radio;
  public:
  Circulo(float abcisa, float ordenada, float radio)
    {
        centro = new Punto(abcisa, ordenada);
        this->radio = radio;
    }

   ~Circulo()
    {
        delete centro;
    }
};
```

Implementación de la composición IV

Si la composición es múltiple se requiere el uso de arrays, vectores o listas.




```
class Poligono
{  int numv;
   Punto* vertices;
   public:
   Poligono(Punto [], int);
   ~Poligono();
};
```

```
Poligono::~Poligono()
{
  delete [] vertices;
}
```

Implementación de la composición V

```
Poligono::Poligono (Punto p[], int n)
{
  //verifica que n es mayor que dos
  //...
  numv = n;
  vertices = new Punto[numv];
  //comprueba que  el vector p
  //no tiene puntos repetidos
  //...
  //se hace una COPIA de los puntos
  //los puntos de esta copia se destruyen
  //cuando se destruye el polígono
  for (int i = 0; i < numv; i++)
     vertices[i]= p[i];
}
```

Generalización/Especialización

- La generalización pretende agrupar la estructura o los comportamientos comunes de un conjunto de clases en una clase más general.
- La especialización pretende crear subclases en las que la estructura o comportamiento de la clase base se modifica y refina.

Desarrollo de jerarquías

La generalización y especialización son técnicas que se usan en el desarrollo de jerarquías y se implementan mediante herencia.

La jerarquía será necesaria cuando se desee posibilitar una actuación polimórfica de objetos pertenecientes a las clases involucradas.

Estará bien diseñada si la case base observa una perfecta relación de inclusión respecto a la(s) clase(s) derivada(s) y puede aplicarse un objeto de la clase base donde aparece un objeto de la(s) clase(s) derivada(s).

Clases abstractas

- Son representantes de la generalización que se colocan como clase base en una jerarquía.
- Han de tener al menos una clase derivada para poder ser utilizadas.
- Son clases que no pueden tener instancias directas (no es posible declarar objetos de estas clases, pero sí punteros o referencias)
- En UML se escriben en cursiva tanto las clases como las operaciones abstractas

Clases y operaciones abstractas en C++

Una clase abstracta es la que tiene, al menos, una operación abstracta.

Las operaciones abstractas en C++ son funciones virtuales puras declaradas en una clase base desde la que definen el interfaz de la operación común para todas las clases derivadas. Las clases derivadas deberán implementar estas operaciones.

```
class Vehiculo
{ // ...
   virtual void desplazarse () = 0;
 };

 //las clases derivadas han de implementar el método
 class Barco : public Vehiculo { ... };
 class Avion : public Vehiculo { ... };
```

¿Preguntas?