Projecte de Programació

Implementación relaciones

Fase de Implementación

Implementación de las relaciones:

El lenguaje de programación que usaremos ya nos proporciona mecanismos para implementar clases (abstractas, genéricas, etc.), atributos, métodos, etc. Así pues, al estudiar el lenguaje de programación ya encontraremos cómo implementar las clases.

Fase de Implementación

Implementación de las relaciones:

Sin embargo, el diagrama de clases no son sólo las clases. Una parte muy importante de la información que nos proporciona un diagrama de clases está en las **relaciones** entre clases.

Algunos lenguajes de programación proporcionan mecanismos para implementar de manera directa alguna de estas relaciones. El caso más típico es la herencia.

Sin embargo, en general hará falta ver cómo implementamos estas relaciones. Eso se puede hacer de manera independiente del lenguaje de programación.

Fase de Implementación

Implementación de les relaciones: Recordemos que en PROP nos limitamos a las siguientes relaciones*:

- *Herencia* : "es-un"
- *Asociación* : "conoce / tiene"
- *Agregación* : "parte-todo / contiene-a"
- *Composición* : "parte-todo / contiene-a"
  (donde la parte no tiene sentido sin el todo)
- *Dependencia* : "usa"

*UML es capaz de expresar más relaciones. Con éstas tenemos suficiente en PROP

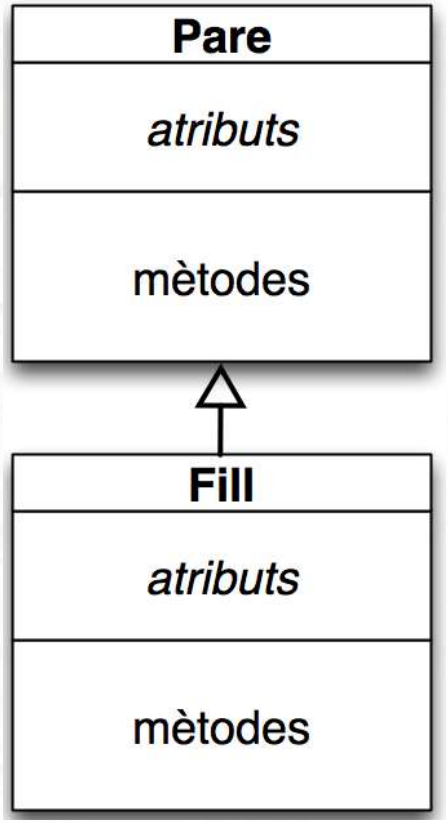
Fase de Implementación

Relación de Herencia:

- Representa diferentes niveles de abstracción

- Las subclases pueden tener atributos adicionales

- Las subclases pueden tener comportamientos diferentes y/o adicionales.



Fase de Implementación

Relación de Herencia:

Usualmente la herencia la implementan los compiladores/intérpretes del lenguaje de programación (LP).

Herencia Simple:

Para redefinir una operación, la subclase puede crear una operación con el mismo nombre y signatura (parámetros y tipo de retorno de la operación).

Para redefinir un atributo, la subclase puede crear un atributo con el mismo nombre y diferente tipo.

Algunos LPs permiten redefinir el nombre de los atributos (Java no)

Fase de Implementación

Relación de Herencia:

Herencia Simple:

Para acceder a la operación de la superclase redefinida en la subclase, desde la subclase existen diversos mecanismos dados por el LP. Un ejemplo es la palabra clave **`super`** (presente en Java, C++ o Smalltalk).

El comportamiento de los hijos (subclases) puede ser muy diferente del comportamiento de los padres (superclases) vía la redefinición de métodos y atributos.

Fase de Implementación

Relación de Herencia:

Herencia Múltiple:

Si el LP la soporta, la implementa el compilador. Semántica problemática (*diamond inheritance problems*).

LPs sin herencia múltiple pueden:

- Simularla vía el mecanismo de **delegación**
- Simular parte de su expresividad gracias a mecanismos *ad-hoc*: Interfaces en Java, Traits en Smalltalk-Pharo. La ventaja (sobre la herencia múltiple) es que no se dan problemas de ambigüedad semántica.

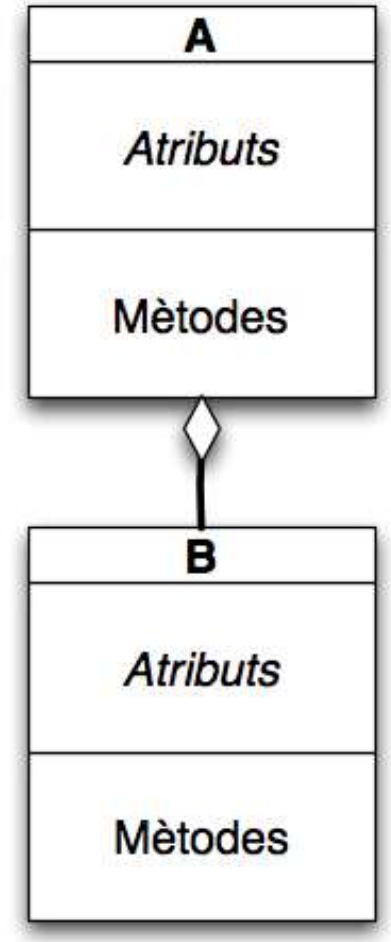
Fase de Implementación

Relación de Agregación:

- Dos clases A y B están relacionadas por una agregación si los objetos de la clase A actúan como contenedores de objetos de la clase B.

- Dos clases A y B relacionadas por una agregación NO están al mismo nivel de abstracción

- La relación puede tener:
  - Nombre
  - Multiplicidad (1, 1-10, 1-*, ... Por defecto 1-1)

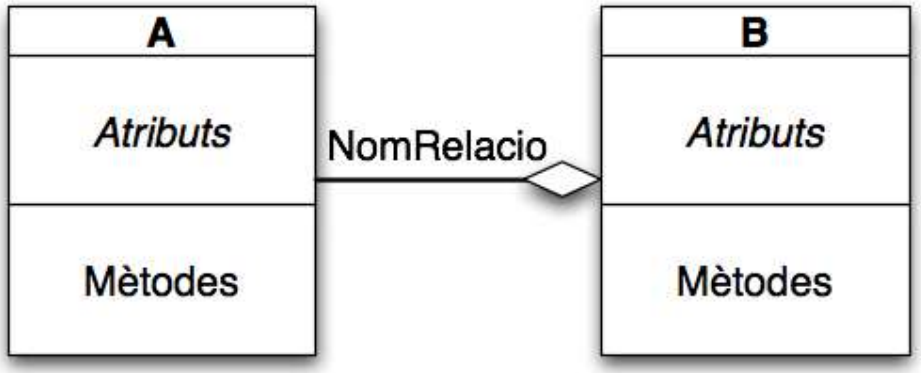


Fase de Implementación

Relación de Agregación:

Se implementará mediante la definición de atributos en cada clase. La *clase-todo*, contenedora, en una agregación definirá atributos para contener referencias a instancias de la *clase-parte*, tantas como requiera la multiplicidad.

Ejemplo: agregación de elementos de clase A dentro de clase B.



Fase de Implementación

Relación de Agregación:

Si la multiplicidad es 1, la clase B definirá un atributo privado de tipo A de nombre **`NomRelacio`**. En Java:

```
private A NomRelacio;
```

Si la multiplicidad es > 1 hará falta que el atributo privado sea una estructura homogénea de elementos de clase A, de nombre **`NomRelacio`**. En Java:

```
private A[] NomRelacio;

private Vector/ArrayList<A> NomRelacio;
```

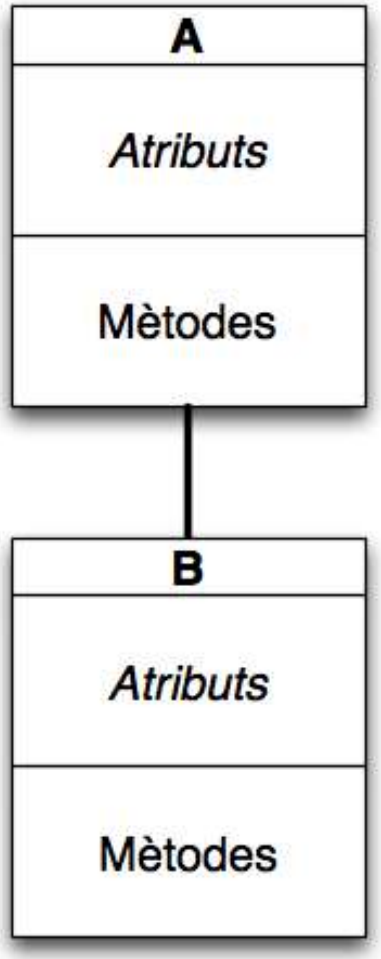
Fase de Implementación

Relación de Asociación:

- Dos clases A y B están relacionadas por una asociación si los objetos de la clase A han de conocer objetos de la clase B y/o viceversa.

- Dos clases A y B relacionadas por una asociación están al mismo nivel de abstracción

- La relación puede tener:
  - Nombre
  - Multiplicidad (1, 1-10, 1-*, … Por defecto 1-1)
  - Navegabilidad (por defecto bidireccional)

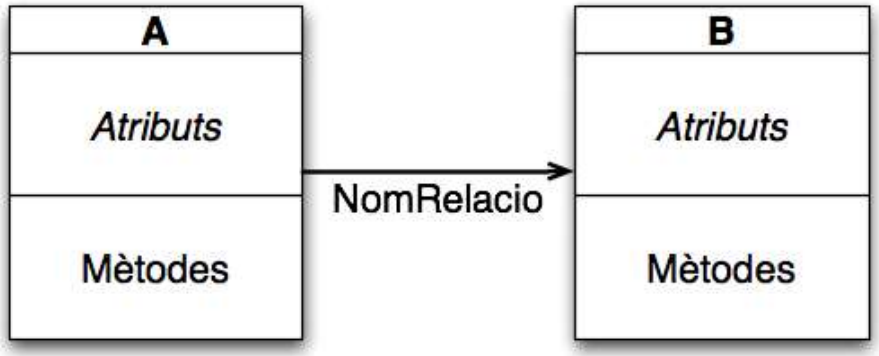


Fase de Implementación

Relación de Asociación:

Opción 1: implementar la asociación como una agregación. Problema: hacer persistente un objeto implicaría guardar todos los objetos asociados, creando fácilmente mucha redundancia y duplicidad en los objetos guardados en los ficheros (o BD).

Opción 2: *Utilizar identificadores únicos para los objetos*. Imaginemos instancias de las clases A y B, relacionadas por una asociación con dirección (navegabilidad):



Fase de Implementación

Relación de Asociación:

1) En la clase A definiríamos un atributo para identificar cada objeto:

```
private TipoIDClaseA idA
```

2) En la clase B definiríamos un atributo para identificar cada objeto:

```
private tipoIDClaseB idB
```

3) Para cada dirección de la asociación y dependiendo de la multiplicidad, tendríamos uno o más atributos de los tipos de los identificadores.

En el ejemplo, la relación tiene una sola dirección y la multiplicidad es 1, por lo que sólo en la clase A tendremos:

```
private tipoIDClaseB NomRelacio
```

Si la multiplicidad fuera > 1 haría falta como atributo de A una estructura homogénea de identificadores de la clase B:

```
private TipoIDClaseB[] NomRelacio
private Vector/ArrayList<TipoIDClaseB> NomRelacio
```

Fase de Implementación

Relación de Asociación:

Cómo se implementa la gestión de los identificadores?

1) Obliga a tener una clase **ConjuntoObjetos** con un contenedor con **acceso directo por identificador** de los objetos de la clase (recordemos que la idea es que los identificadores sean únicos). Por ejemplo, tendríamos que tener 2 clases ConjuntoObjetos para guardar las instancias de A y de B, donde pudiéramos acceder a los objetos a partir de su identificador.

2) Cada vez que se crea un objeto (vía constructor), se consulta ConjuntoObjetos para asignar el nuevo identificador, a la vez que se le pasa el objeto para que lo añada al contenedor, por ej. vía un método:

```
TipoId newId(Object o)
```

O dos métodos:

```
TipoId newId()
void addObject(Object o)
```

(**ConjuntoObjetos** será creado por el controlador del dominio a partir de los datos persistentes)

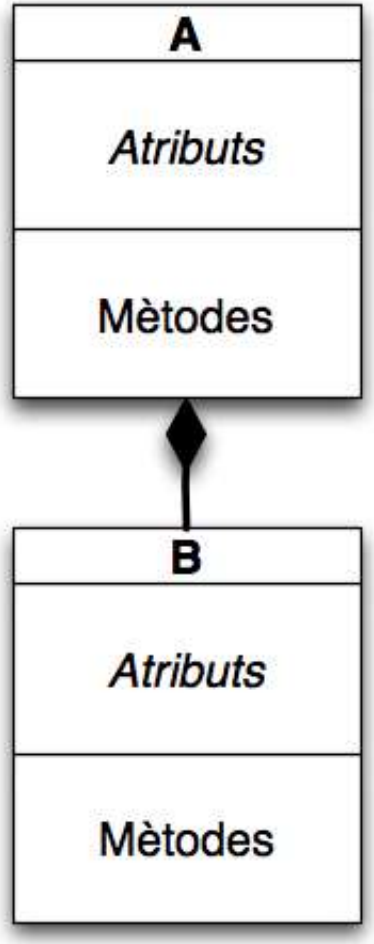
Fase de Implementación

Relación de Composición:

- Igual que la Agregación pero en este caso la parte no tiene ningún sentido sin el todo.

- *Ejemplo* : **`Casilla`** no tiene sentido sin **`Tablero`**

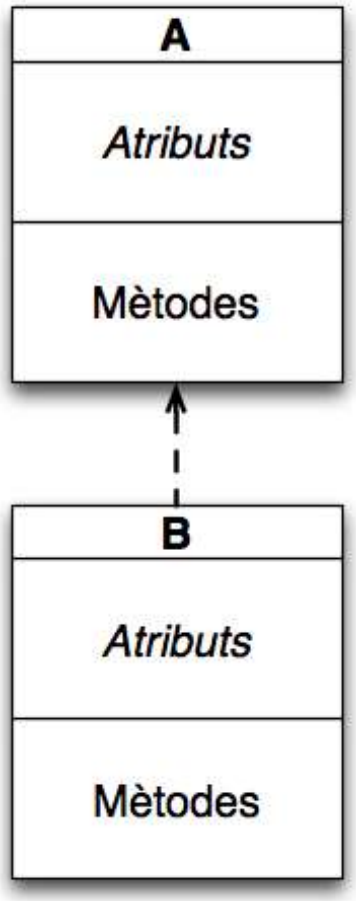
- En PROP no os pediremos diferenciar las composiciones de les agregaciones.



Fase de Implementación

Relación de Dependencia:

- Es una relación que existe para expresar que un objeto de una clase pueda usar los servicios de otra o pueda usar instancias de otra clase como variables locales y/o parámetros (aunque si el parámetro aparece a la cabecera de un método no hace falta poner explícitamente la relación).



Fase de Implementación

Relación de Dependencia:

Esta relación implica que instancias de la clase B necesitan instancias de A para llevar a cabo sus funcionalidades. Usualmente estas instancias son accesibles vía parámetros de mensajes enviados a instancias de B, o de otras maneras (las instancias de A no son atributos de B, eso no sería *dependencia*, sino *asociación* o *agregación*).

Así pues, la implementación de esta relación no requiere nada especial, se trata de invocar desde un método de B a un método de A (teniendo cuidado con la visibilidad).

Ojo, cualquier modificación de la interfície de A puede acabar *rompiendo* a la clase B.