ASOCIACIONES-TODO-lo-que-DEBES-S...

JesusLagares

Programación Orientada a Objetos

2º Grado en Ingeniería Informática

Escuela Superior de Ingeniería
Universidad de Cádiz

ASOCIACIONES

¡Hola! ¿Me echabas de menos? Ya casi he terminado la carrera; no obstante, aún tengo que aprobar esta asignatura... Por eso me he dedicado a diseccionarla al 100% para entender perfectamente qué tengo que saber para poder aprobarla, de ahí la creación de este documento. En él encontrarás todos los conocimientos teóricos (con ejemplos hechos a mano) para que entiendas todos los conceptos relacionados con relaciones entre clases (centrándome en asociaciones) para que, puedas hacer los ejercicios, y aprobar el dichoso examen. No obstante, tengo que decirte que soy un alumno de informática como tú, por lo que, quizás en estos apuntes haya algún error o encuentres algo que se puede explicar mejor. En ambos casos, te animo a que me contactes para poder mejorar estos apuntes. Nada asegura que apruebes con ellos, pero he dado mi mejor esfuerzo para que así sea. Ah, y no lo olvides 😉

MUCHA SUERTE Y A FULL HD QUE SE APRUEBA CARAJO

Definición de Relaciones entre Clases

Las **clases** se relacionan entre sí para conformar la **funcionalidad del sistema**. Por ejemplo, si queremos modelar un sistema en el que se represente que N Profesores imparten M asignaturas, modelaríamos dos clases: Profesores y Asignaturas. Y podríamos decir que **estas clases se relacionan** entre sí.

Por supuesto, al igual que los objetos del mundo real que modelamos interactúan de diferentes maneras entre sí, dentro del mundo del *software* podemos **modelar las relaciones de diferentes formas**.

Podemos decir que **existen distintos tipos de relaciones**:

Tipos de Relaciones

Los diferentes tipos de relaciones que distinguimos son **dependencia**, **asociación**, **agregación/composición**, **generalización** y **realización**.

Dependencia

Cuando una clase se relaciona con otra de tal manera que, **un cambio en una puede afectar a la otra**, pero no necesariamente a la inversa, decimos que una clase depende de otra.

Es decir, si la clase A utiliza la clase B para implementar sus métodos, cambiar la implementación de la clase B cambiaría el comportamiento de A, pero no a la inversa (cambiar A no cambiaría B), por lo que podemos decir que **la clase A depende de B.**

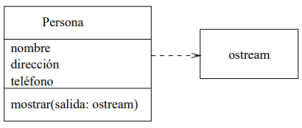


Figura 4.1: Dependencias

Jesús Lagares Galán

Asociación

Mientras que los enlaces expresan conexiones entre objetos, las **asociaciones expresan conexiones entre clases**. Cuando dos clases están **relacionadas/conectadas**, forman una asociación. Las asociaciones poseen:

- Cardinalidad: Indica el número de clases que intervienen en la asociación.
- Multiplicidad: Especifica el número de objetos que puede haber en cada extremo de la asociación.
- Navegabilidad. Determina el sentido en el que se puede recorrer la asociación.

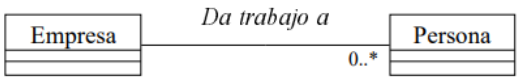


Figura 4.5: Asociación uno a varios

Agregación

Esta relación es una **forma especializada de asociación**. Se da cuando **un objeto de una clase se compone de objetos de otra**. Decimos que se da una agregación cuando:

- Una clase forma parte de otra. Los componentes (Clase 2) conforman el compuesto (Clase 1).

- Los objetos de una clase están subordinados a los de otra.

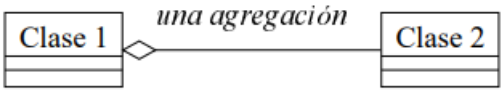


Figura 4.10: Agregación

Composición

Un caso particular de agregación es la **composición**.

La composición implica una **restricción sobre la multiplicidad** en el lado del compuesto (agregado). **Esta clase componente no puede existir sin el compuesto**

Serie Original
Ya disponible solo en

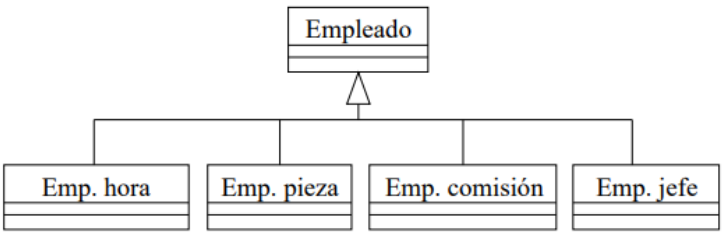
Disney+

Generalización

Cuando una clase es una **especialización de otra**, o una generalización de la otra, se está dando una relación de generalización. La que se está especializando se denomina superclase y las versiones especializadas se denominan subclases.

En C++ este tipo de relaciones se implementa mediante el mecanismo de herencia.

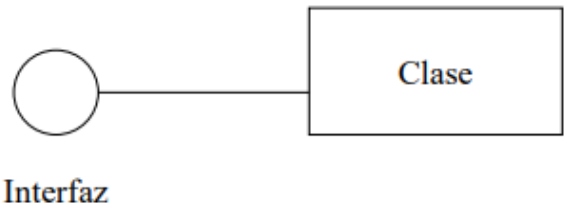
Hablo más detenidamente sobre esto en el documento de **HERENCIA – TODO lo que NECESITAS SABER para APROBAR.** Y si quieres más, siempre puedes revisar **TODOS los EJERCICIOS de HERENCIA**.



Realización

La realización es la **relación que se da entre una interfaz y una clase que implementa dicha interfaz.**

Conocer este concepto no hará que sepas resolver mejor o no los ejercicios de relaciones, de hecho, no hay que tomar ninguna decisión para modelar una relación. No obstante, si quieres ampliar sobre este concepto te recomiendo que consultes el documento de **HERENCIA – TODO lo que NECESITAS SABER para APROBAR.** Y si quieres más, siempre puedes revisar **TODOS los EJERCICIOS de HERENCIA**.

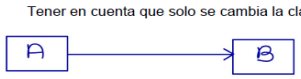


Implementación de las Asociaciones

A la hora de realizar la implementación de las asociaciones utilizaremos diversas técnicas en función de la **navegabilidad**, la **cardinalidad** y la **multiplicidad**: adición de atributos (punteros, maps, sets…), creación de clases de asociación, etc.

APUNTES A TENER EN CUENTA ANTES DE CONTINUAR:

- Si la navegabilidad de la relación es **monodireccional**; es decir, tan solo es posible navegar hacia una dirección, hay que tener en cuenta que **solo se realizarán cambios en la clase Origen.**



- Si la multiplicidad es **1**, se utilizará un **puntero/referencia a un objeto** de la clase asociada.
- Si la multiplicidad es **N** (varios), se utilizará un **set de punteros a objetos** de la clase asociada set<ClaseB*>.
- Si es una **composición**, se utilizarán **objetos y no punteros** o referencias.

Relaciones 1 - 1

Se añadirá como **miembro de datos de cada clase un puntero de la otra clase**.

Habrá que modificar las dos clases.

```
// Asociación 1 - 1
/*
    Se implementa añadiendo un puntero de la otra clase en cada extremo
    Ejemplo: 1 profesor imparte 1 asignatura
*/
class Asignatura  ;

// [CLASE PROFESOR]
class Profesor
{

    public :

        void imparte(Asignatura& asignatura) ;
        Asignatura& imparte() const ;

    private :

        Asignatura* asignatura ;

} ;

void Profesor::imparte(Asignatura& asignatura)
{
    this->asignatura = &asignatura ;
} ;
```

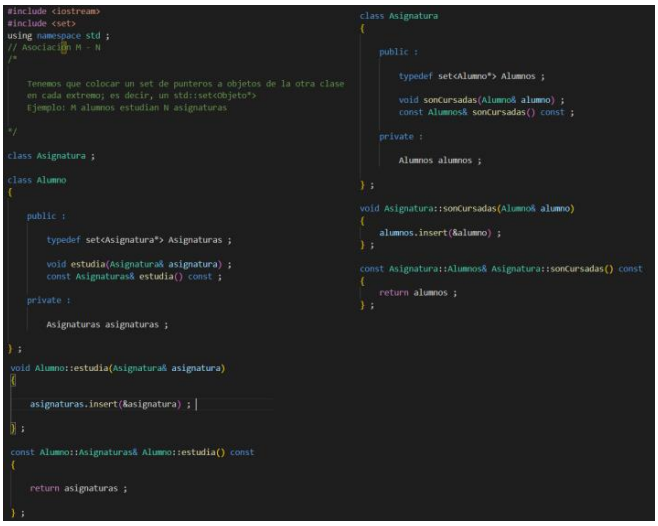
```
// [CLASE ASIGNATURA]
class Asignatura
{

    public :

        void esImpartida(Profesor& profesor) ;
        Profesor& esImpartida() const ;

    private :

        Profesor* profesor ;

} ;

void Asignatura::esImpartida(Profesor& profesor)
{
    this->profesor = &profesor ;
} ;

Profesor& Asignatura::esImpartida() const
```

Jesús Lagares Galán

Relaciones N – M

Se añadirá como **miembro de datos de cada clase set<Clase*>** de la otra clase.

Habrá que modificar las dos clases.



```cpp
#include <iostream>
#include <set>
using namespace std ;
// Asociación M - N
/*

    Tenemos que colocar un set de punteros a objetos de la otra clase
    en cada extremo; es decir, un std::set<Objeto*>
    Ejemplo: M alumnos estudian N asignaturas

*/

class Asignatura ;

class Alumno
{

    public :

        typedef set<Asignatura*> Asignaturas ;

        void estudia(Asignatura& asignatura) ;
        const Asignaturas& estudia() const ;

    private :

        Asignaturas asignaturas ;

} ;
void Alumno::estudia(Asignatura& asignatura)
{

    asignaturas.insert(&asignatura) ;

} ;

const Alumno::Asignaturas& Alumno::estudia() const
{

    return asignaturas ;

} ;
```

```cpp
class Asignatura
{

    public :

        typedef set<Alumno*> Alumnos ;

        void sonCursadas(Alumno& alumno) ;
        const Alumnos& sonCursadas() const ;

    private :

        Alumnos alumnos ;

} ;

void Asignatura::sonCursadas(Alumno& alumno)
{
    alumnos.insert(&alumno) ;
} ;

const Asignatura::Alumnos& Asignatura::sonCursadas() const
{
    return alumnos ;
} ;
```

Asociación Calificada

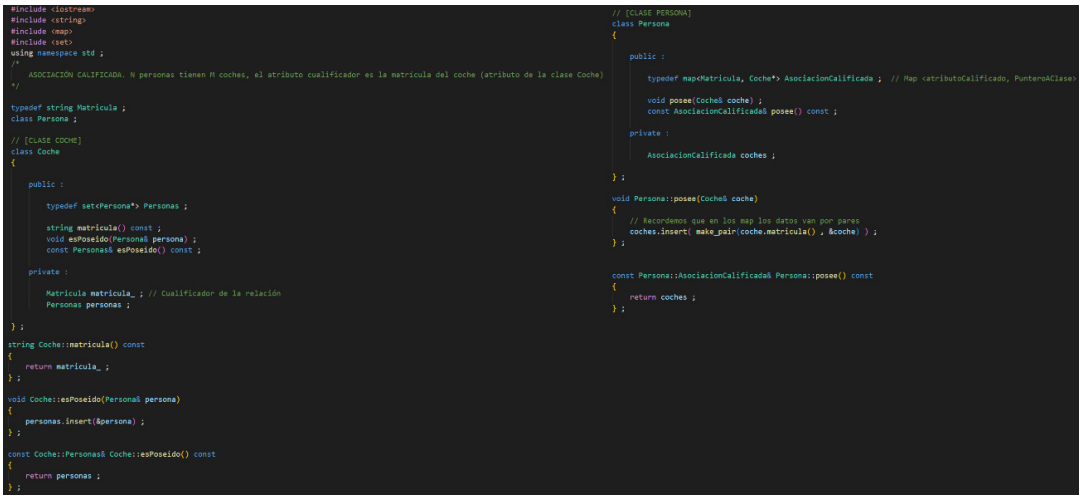
Si tenemos un **calificador** de la relación habrá que **enlazar cada referencia/puntero a la otra clase con su calificador**, por lo que utilizaremos **map** y **multimap**.

Tan solo tendrás que modificar la clase que tiene el calificador en el modelado.

La decisión para la implementación será:

- **Multiplicidad 1** (un calificador corresponde a un objeto): Se implementa utilizando un mal del tipo *map<Calificador , Clase*>*.

- **Multiplicidad N** (un calificador puede corresponder a varios objetos): Se implementa utilizando un multimap<Calificador , Clase*>.



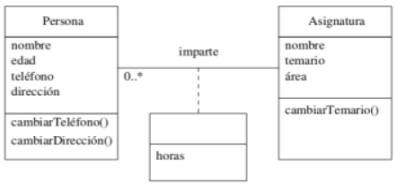
```cpp
#include <iostream>
#include <string>
#include <map>
#include <set>
using namespace std ;
/*
    ASOCIACIÓN CALIFICADA. N personas tienen M coches, el atributo cualificador es la matricula del coche (atributo de la clase Coche)
*/

typedef string Matricula ;
class Persona ;

// [CLASE COCHE]
class Coche
{

    public :

        typedef set<Persona*> Personas ;

        string matricula() const ;
        void esPoseido(Persona& persona) ;
        const Personas& esPoseido() const ;

    private :

        Matricula matricula_ ; // Cualificador de la relación
        Personas personas ;

} ;
string Coche::matricula() const
{
    return matricula_ ;
} ;

void Coche::esPoseido(Persona& persona)
{
    personas.insert(&persona) ;
} ;

const Coche::Personas& Coche::esPoseido() const
{
    return personas ;
} ;
```

```cpp
// [CLASE PERSONA]
class Persona
{

    public :

        typedef map<Matricula, Coche*> AsociacionCalificada ;  // Map <atributoCalificado, PunteroAClase>

        void posee(Coche& coche) ;
        const AsociacionCalificada& posee() const ;

    private :

        AsociacionCalificada coches ;

} ;

void Persona::posee(Coche& coche)
{
    // Recordemos que en los map los datos van por pares
    coches.insert( make_pair(coche.matricula() , &coche) ) ;
} ;


const Persona::AsociacionCalificada& Persona::posee() const
{
    return coches ;
} ;
```

Jesús Lagares Galán

Atributo de Enlace

La relación también puede quedar definida por un **atributo de enlace**; es decir, puede ser que, cada vez que una Persona imparta una Asignatura se cree una relación imparte, que crea un atributo horasImpartidas. Por tanto, cada vez que se cree un objeto de esta relación se implicará este atributo. Este atributo de enlace no tiene por qué ser únicamente un tipo básico del lenguaje (int hora, por ejemplo), sino que puede ser una **nueva clase que contenga diversos atributos**. Hay que tener en cuenta que es un **ATRIBUTO DE ENLACE** no un **ATRIBUTO DE CALIFICADOR**, por lo que no pertenece a ninguna de las clases, sino a la relación.



Se puede modelar de diferentes formas:

- Si es **1 – 1**, puedes almacenarlo en **cualquiera de las clases**
- Si es **N a 1**, puedes almacenarlo como **atributo de la clase varios**.
- Si es **N a M**, no se puede implementar, sino que hay que utilizar una **clase de asociación.** Hay que unir esta clase de asociación que contendrá los atributos con que unir cada puntero/referencia de la clase.
  Habrá que modificar las dos clases añadiendo un *map<Clase* , AtributoDeEnlace>* como miembro de datos.

```cpp
#include <iostream>
#include <string>
#include <map>
#include <set>
using namespace std ;
/*
    ATRIBUTO DE ENLACE. N profesores imparten M asignaturas, y el atributo de enlace de la asociación es el código de la asignatura
                        Hay que incluir dicho atributo en los métodos para asociar, ya que participa en toda la relación
*/

typedef int CodigoAsignatura ;
class Asignatura ;

// [CLASE PROFESOR]
class Profesor
{

    public :

        typedef map<Asignatura* , CodigoAsignatura> Asignaturas ;

        void imparte(Asignatura& persona , CodigoAsignatura codigoAsignatura) ;
        const Asignaturas& imparte() const ;

    private :

        Asignaturas asignaturas ;

} ;
void Profesor::imparte(Asignatura& persona , CodigoAsignatura codigoAsignatura)
{
                                CodigoAsignatura codigoAsignatura
    asignaturas.insert( make_pair(&persona , codigoAsignatura) ) ;

} ;
const Profesor::Asignaturas& Profesor::imparte() const
{

    return asignaturas ;

} ;
```

```cpp
// [CLASE ASIGNATURA]
class Asignatura
{

    public :

        typedef map<Profesor*, CodigoAsignatura> Profesores ;  // Map <atributoCalificado, PunteroAClase>

        void esImpartida(Profesor& profesor , CodigoAsignatura codigoAsignatura) ;
        const Profesores& esImpartida() const ;

    private :

        Profesores profesores ;

} ;

void Asignatura::esImpartida(Profesor& profesor , CodigoAsignatura codigoAsignatura)
{
    // Recordemos que en los map los datos van por pares
    profesores.insert( make_pair(&profesor , codigoAsignatura) ) ;
} ;


const Asignatura::Profesores& Asignatura::esImpartida() const
{
    return profesores ;
} ;
```

Jesús Lagares Galán

Clase de Asociación

A la hora de diseñar una relación podemos elegir **formar una clase de asociació**n (a veces también nos lo piden). Esto es muy útil para relacionar clases existentes que no podamos modificar.

Crearemos una nueva clase **ClaseDeAsociacion** que asociará ambas clases (o 3 si existe un atributo de enlace). Distinguiremos dos casos a la hora de implementar:

- Sin Atributo de Enlace (tan solo hay que asociar las 2 clases):

```cpp
#include <map>
#include <set>
using namespace std ;
/*
    Podemos modelar una clase de asociación para modelar una relación (sin atributos de enlace)
    Para esto se utilizarán 2 maps y 2 sets (para la asociación directa y la inversa)
    Hay que recordar que, cada vez que se modifique una de las asociaciones (la directa o la inversa) hay que modificar también la otra
    IMPORTANTE conocer cómo funcionan los set y los map
*/
class A
{
    public :

    private :
} ;

class B
{
    public :

    private :
} ;
void ClaseAsociacion::asocia(A& a , B& b)
{
    relacionDirecta[&a].insert(&b) ;
    relacionInversa[&b].insert(&a) ; // Hay que modificar las dos relaciones
} ;

void ClaseAsociacion::asocia(B& b , A& a)
{
    asocia(a , b) ; // Utilizamos el método anterior, tan solo hay que llamarlo cambiando el orden de los parámetros
} ;
set<A*> ClaseAsociacion::asociados(B& b) const
{
    map<B* , set<A*>>::const_iterator posicionB = relacionInversa.find(&b) ; // Encontramos la posición de a en el map

    if ( posicionB != relacionInversa.end() ) // Si existe
    {
        return posicionB->second ;
    }

    else
    {
        return set<A*>() ; // Devolvemos un set vació, ya que no existen los enlaces
    }
} ;
```

```cpp
class ClaseAsociacion
{
    public :

        void asocia(A& a , B& b) ;
        void asocia(B& b , A& a) ;
        set<B*> asociados(A& a) const ;
        set<A*> asociados(B& b) const;

    private :

        map<A*, set<B*> > relacionDirecta ;
        map<B*, set<A*> > relacionInversa ;
} ;
```

```cpp
set<B*> ClaseAsociacion::asociados(A& a) const
{
    map<A* , set<B*>>::const_iterator posicionA = relacionDirecta.find(&a) ; // Encontramos la posición de a en el map

    if ( posicionA != relacionDirecta.end() ) // Si existe
    {
        return posicionA->second ;
    }

    else
    {
        return set<B*>() ; // Devolvemos un set vació, ya que no existen los enlaces
    }
```

- Con Atributo de Enlace (hay que asociar las 2 clases con el atributo de enlace):

```cpp
#include <map>
#include <set>
using namespace std ;
/*
    Cuando tenemos un atributo de enlace, la ClaseAsociación será igual que la anterior pero cambiando los set por maps, ya que
    habrá que unir cada clase con el atributo de enlace que pertenece a la asociación
*/
class A
{
    public :

    private :
} ;

class B
{
    public :

    private :
}  ;

class AtributoEnlace // En esta clase estarán contenidos todos los atributos que conforman el AtributoDeEnlace
{
    public :

    private :
} ;
map<B*, ClaseAsociacion*> ClaseAsociacion::asociados(A& a) const
{
    map<A* , map<B* , ClaseAsociacion*>>::const_iterator iteradorDirecta = relacionDirecta.find(&a) ;

    if ( iteradorDirecta != relacionDirecta.end() )
    {
        return iteradorDirecta->second ;
    }

    else
    {

        return map<B*, ClaseAsociacion*>() ;

    }
}
```

```cpp
class ClaseAsociacion
{
    public :

        void asocia(A& a , B& b , ClaseAsociacion& claseAsociacion) ;
        void asocia(B& b , A& a , ClaseAsociacion& claseAsociacion) ;
        map<B*, ClaseAsociacion*> asociados(A& a) const ;
        map<A*, ClaseAsociacion*> asociados(B& b) const ;

    private :

        map<A* , map<B* , ClaseAsociacion*>> relacionDirecta ;
        map<B* , map<A* , ClaseAsociacion*>> relacionInversa ;

} ;
void ClaseAsociacion::asocia(A& a , B& b , ClaseAsociacion& claseAsociacion)
{
    relacionDirecta[&a].insert( make_pair(&b , &claseAsociacion) ) ;
    relacionInversa[&b].insert( make_pair(&a , &claseAsociacion) ) ;
}
void ClaseAsociacion::asocia(B& b , A& a , ClaseAsociacion& claseAsociacion)
{
    asocia(a , b , claseAsociacion) ;
}
map<A*, ClaseAsociacion*> ClaseAsociacion::asociados(B& b) const
{
    map<B* , map<A* , ClaseAsociacion*>>::const_iterator iteradorInversa = relacionInversa.find(&b) ;

    if ( iteradorInversa != relacionInversa.end() )
    {
        return iteradorInversa->second ;
    }

    else
    {

        return map<A*, ClaseAsociacion*>() ;

    }
}
```


Jesús Lagares Galán

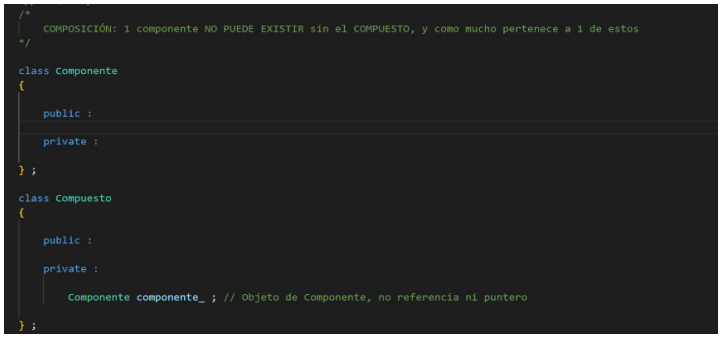
Implementación de las Relaciones de Agregación

Se implementan de igual forma que las asociaciones

- Suelen ser **monodireccionales.**

Implementación de las Relaciones de Composición

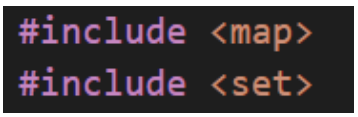
Como el componente no existe sin el compuesto, en lugar de punteros o referencias podemos **utilizar directamente objetos de la otra clase**. La implementación sería igual, en función de si la multiplicidad es 1 o N, pero **en vez de punteros se utilizarán objetos.**



Implementación de las Relaciones de Dependencia

No hay que tomar ninguna decisión a la hora de implementar relaciones de dependencia; sin embargo, estas **deben reducirse todo lo posible**, ya que, cuanta más dependencia, más acoplamiento tendrá nuestro código.

Un ejemplo de uso de dependencias es **cuando incluimos una librería en nuestro lenguaje**.



Implementación de las Generalizaciones

En C++ se implementan gracias al mecanismo de Herencia. Te recomiendo que consultes el documento de **HERENCIA – TODO lo que NECESITAS SABER para APROBAR.** Y si quieres más, siempre puedes revisar **TODOS los EJERCICIOS de HERENCIA.**

Implementación de las Relaciones de Realización

En algunos lenguajes no hay que tomar ninguna decisión a la hora de implementar.

En otros, hay que diseñar clases abstractas como interfaces y especializaciones de ellas para implementarlo. Este 2º caso es el caso de C++. Sin embargo, el conocimiento de esto no será necesario para aprobar la asignatura.

Jesús Lagares Galán