

TODOS-los-EJERCICIOS-de-ASOCIACI...



JesusLagares



Programación Orientada a Objetos



2º Grado en Ingeniería Informática



**Escuela Superior de Ingeniería
Universidad de Cádiz**

The image is a promotional graphic for the Disney+ series "Ms. Marvel". It features Iman Vellani as Ms. Marvel in her red and blue superhero suit, standing in front of a vibrant, colorful wall covered in graffiti and posters. One poster prominently displays the word "LOVE". The "MARVEL STUDIOS" logo is visible above the title. The title "MS MARVEL" is written in large, stylized, metallic letters with a lightning bolt through the 'S'. A yellow "12" rating box is in the top right corner. To the right, the Disney+ logo is displayed with the text "Serie Original Ya disponible". At the bottom right, there is small fine print: "© 2022 MARVEL. Requiere suscripción. Se aplican términos y condiciones."

TOMARSE UN
CAFELITO YA HECHO
FEELS LIKE...

HACER UNA FOTO A LA PIZARRA
EN VEZ DE TOMAR APUNTES.

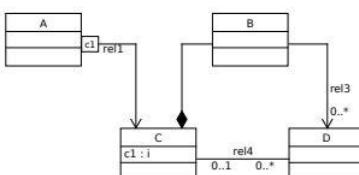
EJERCICIOS DE ASOCIACIONES

¡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 ejercicios que he encontrado por ahí para que entiendas todos los conceptos relacionados con asociaciones para 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

Ejercicio Examen

1. Considere el siguiente diagrama de clases.



- a) Escriba para cada clase los atributos estrictamente indispensables para la implementación de las relaciones en las que participa.
- b) Defina los constructores que estime oportunos para la clase D.
- c) Suponga que se añade un atributo de enlace de tipo X a rel3 ¿cómo cambiarían los miembros de datos de B?

2. Implemente la rel4 del ejercicio anterior mediante una clase de asociación. Para ello:

- a) Defina la clase con los atributos que estime oportuno declarando dos métodos `asocia()` y otros dos llamados `asociados()`, una pareja para cada sentido de la relación.
- b) Defina las funciones miembro `asocia()` de tal forma que ambas permitan crear/modificar el doble enlace entre un objeto de C y otro de D. Si D ya está asociado a un C, se desvinculará del mismo y se enlazará al nuevo.
- c) Defina las dos funciones miembro `asociados()`.

100% INGREDIENTES NATURALES



```

// [EJERCICIO 1]
// Apartado a)
#include <iostream>
#include <map>
#include <set>
using namespace std ;

typedef int c1 ;

class A
{
public:
    typedef map<c1 , C*> Rel1 ;
private:
    Rel1 rel1 ;
} ;

class B
{
public:
private:
    set<D*> rel3 ;
} ;

class C
{
public :
private :
    B b ;
    c1 c1_ ;
    set<D*> rel4 ;
} ;

class D
{
public :

```

```

class rel4
{
public :
    void asocia(C& c , D& d) ;
    void asocia(D& d , C& c ) ;
    set<D*> asociados(C& c) const ;
    C asociados(D& d) const ;

private :
    map<C* , set<D*>> relacionDirecta ;
    map<D* , C*> relacionInversa ;
} ;

void rel4::asocia(C& c , D& d)
{
    relacionDirecta[c].insert(&d) ;
    relacionInversa[&d] = &c ;
}

void rel4::asocia(D& d , C& c )
{
    asocia(c , d) ;
}

```

```

// Apartado b)
// Sería el constructor por defecto, ya que no hay que inicializar nada,
// así que no implementamos nada

// apartado c)
class X // Atributo de enlace
{
public :
private :
} ;

// En rel3 intervienen la clase B y la clase D; sin embargo,
// la navegabilidad es B -> D => Tan solo hay que modificar la clase B
// Al añadir el atributo de enlace, habrá que modificar el set por un map
class B
{
public:
private:
    map<D* , X> rel3 ;
} ;

```



12

MARVEL STUDIOS

MSMARVEL

Serie Original
Ya disponible solo en



© 2022 MARVEL. Requiere suscripción. Se aplican términos y condiciones.

Ejercicio Septiembre

Ejercicio 2 septiembre (1,5 puntos)

El CAC (Centro de Atención al Cliente) de cierta empresa recibe llamadas telefónicas de sus clientes. Una llamada es inicialmente asignada a un operador; si este no puede resolver la cuestión, la transfiere a un compañero especialista, el cual procede de la misma forma: atiende al cliente y si no puede resolver el asunto, transfiere la llamada a otro operador. Las llamadas se transfieren entre operadores. Un operador atenderá diversas llamadas durante su jornada laboral, quedando registradas tanto la fecha (string), hora de inicio (string) y duración de la atención del operador a cada llamada. Así mismo, se registrarán los operadores que atendieron una llamada.

Implemente las clases **Operador** y **Llamada** con los atributos necesarios para implementar la relación entre ellas, así como los siguientes métodos:

- o **recibir()**: un operador recibe una llamada.
- o **asignar()**: una llamada se asigna a un operador.
- o **llamadas()**: llamadas atendidas por un operador.
- o **asistente()**: operadores que atendieron una llamada.

Instrucciones de entrega:

- Escriba a mano su respuesta en papel.
- Coloque su DNI o tarjeta universitaria sobre cada página escrita (sin que tape el texto) y capture una imagen nítida.
- Renombré el fichero de cada imagen con su nombre y apellidos y añada un número correlativo de página.
- Suba cada fichero de imagen.

Tiempo máximo 20 minutos

```
#include <iostream>
#include <string>
#include <map>
#include <set>
using namespace std;

/*
  1 operador atiende N llamadas
  Cada llamada tiene un
*/
class Operador
{
public:
private:
};

class Llamada
{
public:
private:
};

class InfoLlamada // Creamos un atributo de enlace para guardar la información de la llamada
{
public:
private:
    string fecha, horaInicio;
    int duracion;
};

class ClaseDeAsociacion
{
public:
    void recibir( Operador& operador , Llamada& llamada , InfoLlamada& infollamada ) ;
    void asignar( Llamada& llamada , Operador& operador , InfoLlamada& infollamada ) ;
    map<Llamada* , InfoLlamada*> llamadas(Operador& operador) const ;
    map<Operador* , InfoLlamada*> asistente(Llamada& llamada) const ;
};

private :
    map<Operador* , map<Llamada* , InfoLlamada*>> operadorLlamada ;
    map<Llamada* , map<Operador* , InfoLlamada*>> llamadaOperador ;
};

void ClaseDeAsociacion::recibir( Operador& operador , Llamada& llamada , InfoLlamada& infollamada )
{
    OperadorLlamada[&operador].insert( make_pair(&llamada , &infollamada ) );
}

void ClaseDeAsociacion::asignar( Llamada& llamada , Operador& operador , InfoLlamada& infollamada )
{
    LlamadaOperador[&llamada].insert( make_pair(&operador , &infollamada ) );
}

map<Llamada* , InfoLlamada*> ClaseDeAsociacion::llamadas(Operador& operador) const
{
    map<Operador* , map<Llamada* , InfoLlamada*>>::const_iterator iteradorOperadorLlamada = OperadorLlamada.find(&operador) ;
    if ( iteradorOperadorLlamada != OperadorLlamada.end() )
    {
        return iteradorOperadorLlamada->second ;
    }
    else
    {
        return map<Llamada* , InfoLlamada*>() ;
    }
}

map<Operador* , InfoLlamada*> ClaseDeAsociacion::asistente(Llamada& llamada) const
{
    map<Llamada* , map<Operador* , InfoLlamada*>>::const_iterator iteradorLlamadaOperador = LlamadaOperador.find(&llamada) ;
    if ( iteradorLlamadaOperador != LlamadaOperador.end() )
    {
        return iteradorLlamadaOperador->second ;
    }
    else
    {
        return map<Operador* , InfoLlamada*>() ;
    }
}
```

Ejercicio Febrero 2021

Ejercicio 2 febrero 2021 (1,5 puntos)

El CAC (Centro de Atención al Cliente) de cierta empresa recibe llamadas telefónicas de sus clientes. La dirección del CAC está interesada en mantener un histórico de todas las llamadas recibidas por su plantilla de operadores. Un operador atenderá diversas llamadas durante su jornada laboral, a cada una de las cuales se le asigna un código único (*string*) y quedan registradas la fecha (*string*), hora de inicio (*string*) y duración de cada llamada, así como el operador que la atiende.

Implemente la relación entre las clases **Operador** y **Llamada** con los atributos necesarios para implementar la relación entre ellas (usando el código de llamada como calificador), así como los siguientes métodos:

- **recibir()**: un operador recibe una llamada.
- **asignar()**: una llamada se asigna a un operador.
- **asistente()**: operador que atendió una llamada.
- **atendidas()**: llamadas atendidas por un operador.

Instrucciones de entrega:

- Escriba su respuesta a mano en papel.
- Coloque su DNI o tarjeta universitaria sobre cada página escrita (sin que tape el texto) y capture una imagen nítida.
- Renombre el fichero de cada imagen con su nombre y apellidos y añada un número correlativo de página.
- Suba cada fichero de imagen.

```
#include <iostream>
#include <string>
#include <map>
#include <set>
using namespace std ;

/*
  1 operador atiende N llamadas
  Cada llamada tiene un código único, fecha, hora de inicio y duración
  El código único actúa como calificador
 */

typedef string CódigoÚnico ;
class Operador ;
// [CLASS LLAMADA]
class Llamada
{
public :
    CódigoÚnico código() const ;
    void asignar(Operador& operador) ;
    const Operador& asistente() const ;
private :
    Operador* operador ;
    CódigoÚnico códigoÚnico ;
} ;
void Llamada::asignar(Operador& operador)
{
    this->operador = &operador ;
}
const Operador& Llamada::asistente() const
{
    return *operador ;
}

// [CLASS OPERADOR]
class Operador
{
public :
    typedef map<CódigoÚnico , Llamada*> Llamadas ;
    void recibir(Llamada& llamada) ;
    map<CódigoÚnico , Llamada*> atendidas() const ;
private :
    Llamadas llamadas ;
} ;
void Operador::recibir(Llamada& llamada)
{
    llamadas.insert( make_pair( llamada.código() , &llamada ) );
}
Operador::Llamadas Operador::atendidas() const
{
    return llamadas ;
}
```

**TOMARSE UN
CAFELITO YA HECHO
FEELS LIKE...**

**REPARTIRSE A QUÉ CLASE VAIS CADA
UNO PARA DESPUÉS UNIR APUNTES.**

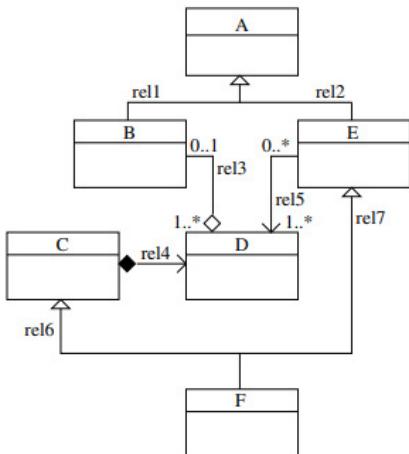
Ejercicio Mayo 2008

Curso 2007-08

Jueves 29 de mayo de 2008

1. Dado el siguiente diagrama, defina las clases que aparecen escribiendo exclusivamente los miembros imprescindibles para implementar las relaciones.

5,0 p



```

#include <iostream>
#include <set>
using namespace std;

class A
{
public :
private :
};

class B: public A
{
public:
    void rel3(D& d) ;
private:
    set<D*> rel3_ ;
};

class C
{
public :
private :
    D rel4 ; // Objeto, ya que es una composición
};

class D
{
public :
    void rel3(B& d) ;

private :
    B* rel3_ ;
};

class E: public A
{
public :
private :
    set <D*> rel5 ;
};

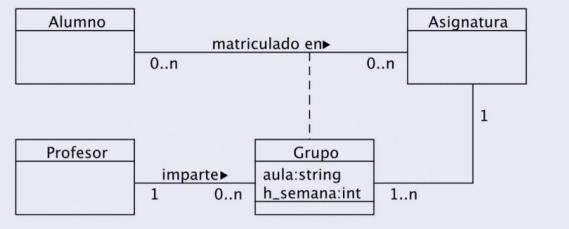
class F: public C, public E
{
public :
private :
};
  
```

**100% INGREDIENTES
NATURALES**



Ejercicios Seminario

Sea el diagrama de clases siguiente:



Ejercicio 1

Implemente las clases del diagrama, declarando exclusivamente los miembros imprescindibles para implementar las relaciones.

```
#include <iostream>
#include <string>
#include <map>
#include <set>

using namespace std ;

class Asignatura ;
// [CLASE GRUPO]
class Grupo
{
public :
    void asocia(Asignatura& asignatura) ;
    const Asignatura& asocia() const ;
private :
    string aula ;
    int h_semana ;
    Asignatura* asignatura_ ;
} ;
void Grupo::asocia(Asignatura& asignatura)
{
    asignatura_ = &asignatura ;
}
const Asignatura& Grupo::asocia() const
{
    return *asignatura_ ;
}

// [CLASE PROFESOR]
class Profesor
{
public :
    typedef set<Grupo*> Grupos ;
    void asocia (Grupo& grupo) ;
    const Grupos& asocia() const ;
private :
    Grupos grupos ;
} ;
void Profesor::asocia (Grupo& grupo)
{
    grupos.insert(&grupo) ;
}
const Profesor::Grupos& Profesor::asocia() const
{
    return grupos ;
}

// [CLASE ALUMNO]
class Alumno
{
public :
    typedef map<Asignatura* , Grupo*> Asignaturas ;
    void asocia(Asignatura& asignatura , Grupo& grupo) ;
    const Asignaturas& asocia() const ;
private :
    Asignaturas asignaturas ;
} ;
void Alumno::asocia(Asignatura& asignatura , Grupo& grupo)
{
    asignaturas.insert( make_pair(&asignatura , &grupo) ) ;
}
const Alumno::Asignaturas& Alumno::asocia() const
{
    return asignaturas ;
}

// [CLASE ASIGNATURA]
class Asignatura
{
public :
    typedef set<Grupo*> Grupos ;
    void asocia(Grupo& grupo) ;
    const Grupos& asocia() const ;
private :
    Grupos grupos ;
} ;
void Asignatura::asocia(Grupo& grupo)
{
    grupos.insert(&grupo) ;
}
const Asignatura::Grupos& Asignatura::asocia() const
{
    return grupos ;
}
```

Ejercicio 3

Defina los dos métodos siguientes:

- Alumno::matriculado_en() para matricular a un alumno en una asignatura asignándole un grupo.
- Profesor::imparte() para vincular un profesor a un grupo.

```
// [CLASE ALUMNO]
class Alumno
{
public :

    typedef map<Asignatura* , Grupo*> Asignaturas ;

    void matriculado_en(Asignatura& asignatura , Grupo& grupo) ;
    const Asignaturas& asocia() const;

private :
    Asignaturas asignaturas ;

} ;

void Alumno::matriculado_en(Asignatura& asignatura , Grupo& grupo)
{
    asignaturas.insert( make_pair(&asignatura , &grupo) ) ;
}
```

```
// [CLASE PROFESOR]
class Profesor
{
public :

    typedef set<Grupo*> Grupos ;

    void imparte (Grupo& grupo) ;
    const Grupos& asocia() const ;

private :

    Grupos grupos ;

} ;

void Profesor::imparte (Grupo& grupo)
{
    grupos.insert(&grupo) ;
}
```

Ejercicio 4

Declare una clase de asociación `Alumno_Asignatura` para la relación `matriculado_en`. Para ello declare los atributos que considere necesarios y dos métodos `matriculado_en()` y `matriculados()`. El primero registra a un alumno en una asignatura asignándole el grupo y el segundo devuelve todas las asignaturas (y los correspondientes grupos) en que se encuentre matriculado un alumno. Declare sobrecargas de estos dos métodos para el otro sentido de la asociación.

```
class AlumnoAsignatura
{
public :
    typedef map<Alumno* , map<Asignatura* , Grupo*>> RelacionDirecta ;
    typedef map<Asignatura* , map<Alumno* , Grupo*>> RelacionInversa ;

    void matriculado_en(Alumno& alumno , Asignatura& asignatura , Grupo& grupo) ;
    void matriculado_en(Asignatura& asignatura , Alumno& alumno , Grupo& grupo) ;
    map<Asignatura* , Grupo*> matriculados(Alumno& alumno) const ;
    map<Alumno* , Grupo*> matriculados(Asignatura& asignatura) const ;

private :
    RelacionDirecta directa;
    RelacionInversa inversa;

};

void AlumnoAsignatura::matriculado_en(Alumno& alumno , Asignatura& asignatura , Grupo& grupo)
{
    directa[&alumno].insert( make_pair( &asignatura , &grupo ) );
    inversa[&asignatura].insert( make_pair( &alumno , &grupo ) );
}

void AlumnoAsignatura::matriculado_en(Asignatura& asignatura , Alumno& alumno , Grupo& grupo)
{
    matriculado_en(alumno , asignatura , grupo) ;
}

map<Asignatura* , Grupo*> AlumnoAsignatura::matriculados(Alumno& alumno) const
{
    map<Alumno* , map<Asignatura* , Grupo*>>::const_iterator iteradorDirecta = directa.find(&alumno) ;
    if (iteradorDirecta != directa.end() )
    {
        return iteradorDirecta->second ;
    }
    else
    {
        return map<Asignatura* , Grupo*>() ;
    }
}

map<Alumno* , Grupo*> AlumnoAsignatura::matriculados(Asignatura& asignatura) const
{
    map<Asignatura* , map<Alumno* , Grupo*>>::const_iterator iteradorInversa = inversa.find(&asignatura) ;
    if (iteradorInversa != inversa.end() )
    {
        return iteradorInversa->second ;
    }
    else
    {
        return map<Alumno* , Grupo*>() ;
    }
};
```

**TOMARSE UN
CAFELITO YA HECHO
FEELS LIKE...**

**HACER UNA FOTO A LA PIZARRA
EN VEZ DE TOMAR APUNTES.**

Ejercicio 5

Declare una clase de asociación `Profesor_Grupo` para la relación `imparte`. Incluya en ella los atributos oportunos y dos métodos `imparte()` e `impartidos()`. El primero enlaza un profesor con un grupo y el segundo devuelve todos los grupos que imparte un profesor. Sobrecargue ambas funciones miembro para el sentido inverso de la relación.

```
class Profesor_Grupo
{
public :

    typedef map<Profesor* , set<Grupo*>> RelacionDirecta ;
    typedef map<Grupo* , Profesor*> RelacionInversa ;

    void imparte(Profesor& profesor , Grupo& grupo) ;
    void imparte(Grupo& grupo , Profesor& profesor) ;
    const set<Grupo*> impartidos(Profesor& profesor) const ;
    const Profesor& impartidos(Grupo& grupo) const ;

private :

    RelacionDirecta directa ;
    RelacionInversa inversa ;

};

void Profesor_Grupo::imparte(Profesor& profesor , Grupo& grupo)
{
    directa[&profesor].insert(&grupo) ; // Ingresamos el profesor en el set
    inversa.insert( make_pair(&grupo , &profesor) ) ; // Inseretaos la pareja Profesor - Grupo en el map
}

void Profesor_Grupo::imparte(Grupo& grupo , Profesor& profesor)
{
    imparte(profesor , grupo) ;
}

const set<Grupo*> Profesor_Grupo::impartidos(Profesor& profesor) const
{
    map<Profesor* , set<Grupo*>>::const_iterator iteradorDirecta = directa.find(&profesor) ;

    if ( iteradorDirecta != directa.end() )
    {
        return iteradorDirecta->second ;
    }
    else
    {
        return set<Grupo*>() ;
    }
}

const Profesor& Profesor_Grupo::impartidos(Grupo& grupo) const
{

    map<Grupo* , Profesor*>::const_iterator iteradorInversa = inversa.find(&grupo) ;

    if ( iteradorInversa != inversa.end() )
    {
```

**100% INGREDIENTES
NATURALES**



Ejercicio 6

Defina el método `Alumno_Asignatura::matriculado_en()` (y su sobrecarga) para matricular a un alumno en una asignatura asignándole un grupo. Esta función también permitirá cambiar el grupo al que pertenece un alumno ya matriculado en la asignatura.

```
void AlumnoAsignatura::matriculado_en(Alumno& alumno , Asignatura& asignatura , Grupo& grupo)
{
    // Relación Directa
    auto iteradorDirecta = directa.find(&alumno) ;

    if ( iteradorDirecta != directa.end() ) // Si el alumno ya está en el maps hay que cambiar el grupo al que pertenece
    {
        // Miramos si está la asignatura
        auto iteradorInternoDirecta = iteradorDirecta->second.find(&asignatura) ;

        if ( iteradorInternoDirecta != iteradorDirecta->second.end() ) // Si está la asignatura en el map
        {
            iteradorInternoDirecta->second = &grupo ; // Cambiamos el grupo
        }
        else // La asignatura no está en el maps
        {
            iteradorDirecta->second.insert( make_pair(&asignatura , &grupo) ) ; // La insertamos
        }
    }
    else // Si no está, lo añadimos directamente
    {
        map<Asignatura* , Grupo*> mapAuxiliar ;
        mapAuxiliar.insert( make_pair(&asignatura , &grupo) ) ;

        directa.insert( make_pair(&alumno , mapAuxiliar) ) ;
    }

    // Relación Inversa
    auto iteradorInversa = inversa.find(&asignatura) ;

    if ( iteradorInversa != inversa.end() ) // Si la asignatura ya está en el maps hay que cambiar el grupo al que pertenece
    {
        // Miramos si está el alumno
        Alumno &alumno
        auto iteradorInternoInversa = iteradorInversa->second.find(&alumno) ;

        if ( iteradorInternoInversa != iteradorInversa->second.end() ) // Si está el alumno en el map
        {
            iteradorInternoInversa->second = &grupo ; // Cambiamos el grupo
        }
        else // El alumno no está en el maps
        {
            iteradorInversa->second.insert( make_pair(&alumno , &grupo) ) ; // Lo insertamos
        }
    }
    else // Si no está, lo añadimos directamente
    {
        map<Asignatura* , Grupo*> mapAuxiliar ;
        mapAuxiliar.insert( make_pair(&asignatura , &grupo) ) ;

        directa.insert( make_pair(&alumno , mapAuxiliar) ) ;
    }
}

void AlumnoAsignatura::matriculado_en(Asignatura& asignatura , Alumno& alumno , Grupo& grupo)
{
    matriculado_en(alumno , asignatura , grupo) ;
}
```

Ejercicio 7

Escriba la definición de `Profesor_Grupo::imparte()`. Si el grupo ya tiene un profesor asociado, se deberá desvincular del mismo y enlazarlo con el nuevo.

```
void Profesor_Grupo::imparte(Profesor& profesor , Grupo& grupo)
{
    // Relación Inversa
    auto iteradorInversa = inversa.find(&grupo) ;
    if ( iteradorInversa != inversa.end() ) // El profesor ya está
    {
        iteradorInversa->second = &profesor ; // Se vincula con el nuevo
    }
    else // Si no está
    {
        inversa.insert( make_pair(&grupo , &profesor) ) ;
    }

    // Relación Directa
    auto iteradorDirecta = directa.find(&profesor) ;
    if ( iteradorDirecta != directa.end() ) // Si el profesor existe
    {
        iteradorDirecta->second.insert(&grupo) ; // Lo vinculamos al nuevo grupo
    }
    else // Si no
    {
        set<Grupo*>* setGrupo ;
        setGrupo.insert(&grupo) ;
        directa.insert( make_pair(&profesor , setGrupo) ) ;
    }
}

void Profesor_Grupo::imparte(Grupo& grupo , Profesor& profesor)
{
    imparte(profesor , grupo) ;
}
```

Ejercicio 8

Defina Profesor_Grupo::impartidos().

```
const set<Grupo*> Profesor_Grupo::impartidos(Profesor& profesor) const
{
    map<Profesor*, set<Grupo*>>::const_iterator iteradorDirecta = directa.find(&profesor) ;

    if ( iteradorDirecta != directa.end() )
    {
        return iteradorDirecta->second ;
    }
    else
    {
        return set<Grupo*>() ;
    }
} ;
const Profesor& Profesor_Grupo::impartidos(Grupo& grupo) const
{

    map<Grupo* , Profesor*>::const_iterator iteradorInversa = inversa.find(&grupo) ;

    if ( iteradorInversa != inversa.end() )
    {
        return *(iteradorInversa)->second ;
    }
    else
    {
        return nullptr ;
    }
} ;
```