

Seminarios-resueltos-UNIFICADOS-...



DoctorDS



Programación Orientada a Objetos



2º Grado en Ingeniería Informática



**Escuela Superior de Ingeniería
Universidad de Cádiz**



5 ECO TIPS FOREVER GREEN

SIGUE LOS CONSEJOS DE FOREVER GREEN
PARA CUIDAR EL MEDIO AMBIENTE

1

REUTILIZA

y recicla
todo lo que puedas.

utilizar siempre objetos
que puedan tener una
segunda vida

2

ENCIENDE

Las luces justas.

para ahorrar utiliza
los horarios más baratos:
- **llano** de 8:00 a 10:00 h,
de 14:00 a 18:00 horas y
de 22:00 a 00:00 horas
- **valle** de 00:00 a 08:00 h.

3

APROVECHA

todo el agua.

no desperdigies los
primeros litros esperando
a que salga caliente.



4

EVITA

contaminar el
entorno.

no depositar en el medio
ambiente materiales o
productos que son espe-
cialmente agresivos para
la salud y la naturaleza.

5

USA BICICLETA

transporte público o
comparte vehículo
cuando lo utilices.
Así se reduce el CO2

¿AÚN NO
NOS CONOCES?

VISÍTANOS SOMOS

FOREVER GREEN



Estudiar sin publi es posible.

Compra Wuolah Coins y que nada te distraiga durante el estudio.

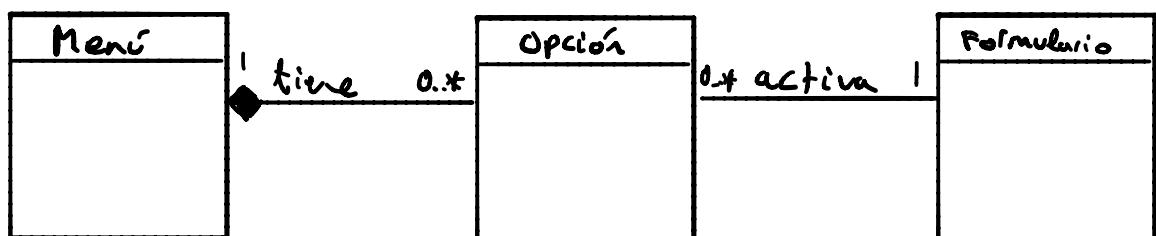


Seminario 3.1

Ejercicio 1

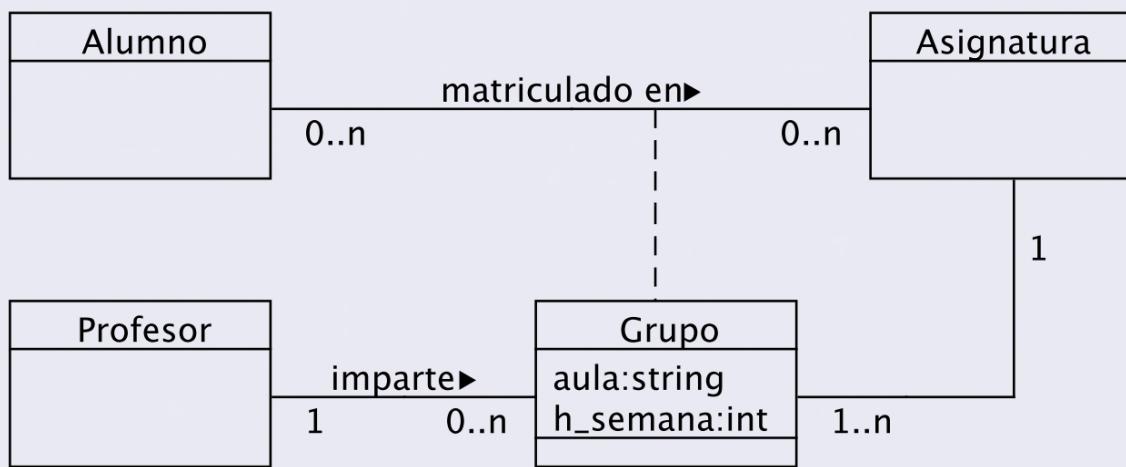
Suponga que hay que desarrollar la interfaz de usuario de una aplicación. Dicha interfaz estará formada por un **menú** con una serie de **opciones**. El único comportamiento a tener en cuenta es que desde cada menú se puede ejecutar cualquiera de sus opciones, desencadenando dicha opción la activación de un **formulario**. Una opción sólo puede aparecer en un menú, pero un mismo formulario puede ser compartido por varias opciones.

Describa e implemente las relaciones que se establecerán entre estas clases.



WUOLAH

Sea el diagrama de clases siguiente:



Ejercicio 2

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

```
class Alumno {
public:
    void matriculado_en(Asignatura* A, Grupo* G);
    void elimina(Asignatura* A);

private:
    std::map<Asignatura*, Grupo*> a_g;
};

void Alumno::matriculado_en(Asignatura* A, Grupo* G) {
    a_g.insert(std::make_pair(A, G));
}

void Alumno::elimina(Asignatura* A) {
    a_g.erase(A);
}
```

```
class Asignatura {
```

```
public:
```

```
    void asocia(Grupo* G);
```

```
    void elimina(Grupo* G);
```

```
private:
```

```
    std::set<Grupo*> g;
```

```
};
```

```
void Asignatura::asocia(Grupo* G) { g.insert(G); }
```

```
void Asignatura::elimina(Grupo* G) { g.erase(G); }
```

```
class Profesor {
```

```
public:
```

```
    void imparte(Grupo* G);
```

```
    void elimina(Grupo* G);
```

```
private:
```

```
    std::set<Grupo*> g;
```

```
};
```

```
void Profesor::imparte(Grupo* G) { g.insert(G); }
```

```
void Profesor::elimina(Grupo* G) { g.erase(G); }
```

Class Grupo {

public:

void asocia(Asignatura* A);

void elimina();

private:

Asignatura* a;

}

void Grupo::asocia(Asignatura* A) { a = A; }

void Grupo::elimina() { a = nullptr; }

Estudiar sin publi es posible.

Compra Wuolah Coins y que nada te distraiga durante el estudio.



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.

```
void Alumno::matriculado_en(Asignatura* A, Grupo* G){  
    a_g.insert(std::make_pair(A, G));  
}  
  
void Profesor::imparte(Grupo* G){g.insert(G);}
```

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.

La asociación ahora pasa a ser bidireccional.

```
class Alumno_Asignatura{
```

```
public:
```

```
    void matriculado_en(Alumno* A, Asignatura* A1, Grupo* G);  
    std::map<Asignatura*, Grupo*> matriculados(Alumno* A) const;  
  
    void matriculado_en(Asignatura* A1, Alumno* A, Grupo* G);  
    std::map<Alumno*, Grupo*> matriculados(Asignatura* A) const;
```



WUOLAH

private:

```
std::map<Alumno*, std::map<Asignatura*, Grupo*>> a_g;
```

```
std::map<Asignatura*, std::map<Alumno*, Grupo*>> sas_g;
```

};

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. Sobrecregue ambas funciones miembro para el sentido inverso de la relación.

La asociación pasa a ser bidireccional

```
class Profesor_Grupo {
```

public:

```
void imparte(Profesor* P, Grupo* G);
```

```
void imparte(Grupo* G, Profesor* P);
```

```
std::set<Grupo*> impartidos(Profesor* P) const;
```

```
Profesor* impartidos(Grupo* G) const;
```

private:

```
std::map<Profesor* P, std::set<Grupo* G>> p_g;
```

```
std::map<Grupo*, Profesor* p> g_p;
```

};

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 Alumno_Asignatura :: matriculado_en (Alumno* A ) {  
    Asignatura* AS, Grupo* G  
  
    auto it = a - g . find ( A ) ;  
    if ( it != a - g . end () ) {  
        it -> second . insert ( std :: make _pair ( AS , G )) ;  
    } else {  
        a - g . insert ( std :: make _pair ( A , std :: map < Asignatura* ,  
                                         Grupo* > ( { std :: make _pair ( AS , G ) } ) ) );  
    }  
  
    it = as - g . find ( AS ) ;  
    if ( it != as - g . end () ) {  
        it -> second . insert ( std :: make _pair ( A , G )) ;  
    } else {  
        as - g . insert ( std :: make _pair ( AS , std :: map < Alumno* ,  
                                         Grupo* > ( { std :: make _pair ( A , G ) } ) ) );  
    }  
}  
  
void Alumno_Asignatura :: matriculado_en ( Asignatura* AS , Alumno* A , Grupo* G ) {  
    matriculado_en ( A , AS , G );  
}
```

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* P, Grupo* G ) {  
    auto it = p-g->find( P );  
    if( it != p-g->end() ) {  
        auto it2 = it->second->find( G );  
        if( it2 == it->second->end() ) {  
            it->second->insert( G );  
        }  
    } else {  
        p-g->insert( std::make_pair( P, std::set< Grupo* >{ G } ) );  
    }  
  
    it = g-p->find( G );  
    if( it != g-p->end() ) {  
        it->second = P;  
    } else {  
        g-p->insert( std::make_pair( G, P ) );  
    }  
}
```

Estudiar sin publi es posible.

Compra Wuolah Coins y que nada te distraiga durante el estudio.



```
void ProfesorGrupos::imparte(Grupo* G, Profesor* P){  
    imparte(P,G);  
}
```

Reservados todos los derechos.
No se permite la explotación económica ni la transformación de esta obra. Queda permitida la impresión en su totalidad.



WUOLAH

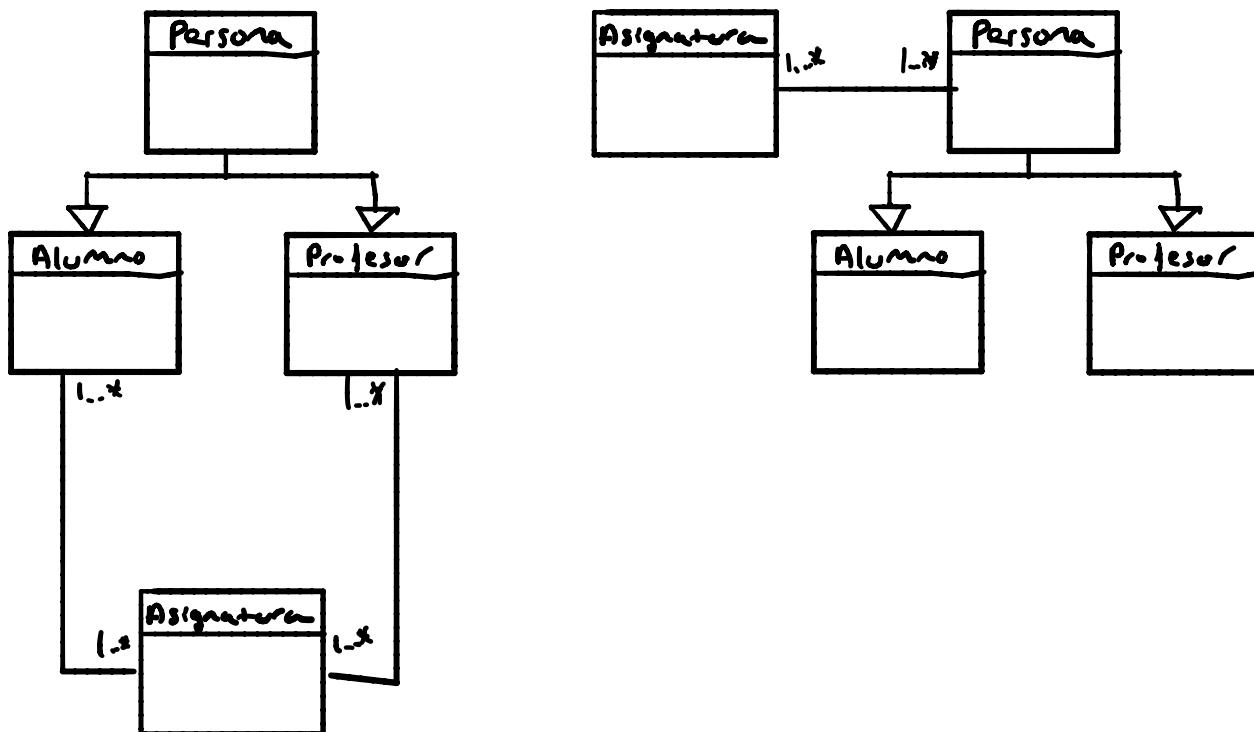
Seminario 3.2

Ejercicio 1

Se dispone de una clase base **Persona** y dos clases especializadas **Alumno** y **Profesor**. Se quiere saber qué alumnos están matriculados en qué asignaturas y qué profesores imparten qué asignaturas, y viceversa. Para ello hay dos opciones:

- Dos asociaciones bidireccionales varios a varios, una entre **Alumno** y **Asignatura**, y otra entre **Profesor** y **Asignatura**.
- Una única asociación bidireccional varios a varios entre las clases **Persona** y **Asignatura**.

¿Cuál de los dos opciones considera más conveniente?



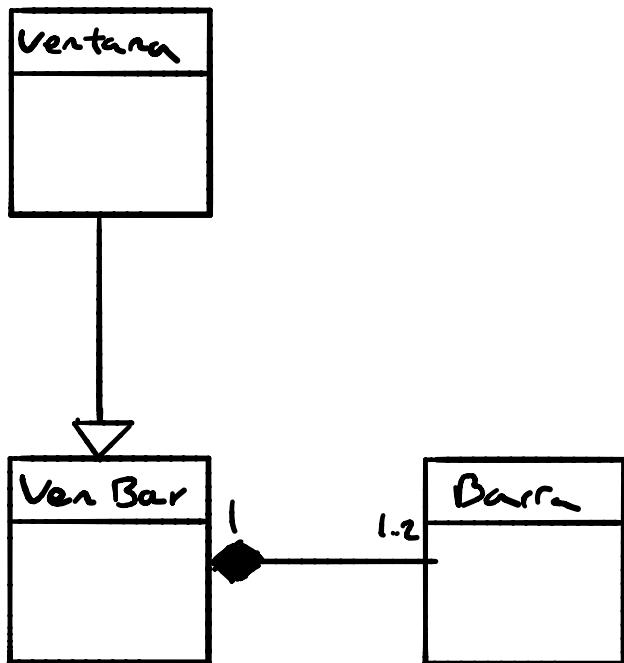
Sí.

No.

Hay que hacer distinciones entre clases, con la segunda opción no se podría.

Ejercicio 2

Supóngase que existen ya definidas dos clases `Ventana` (ventana gráfica), y `Barra` (barra de desplazamiento) y se quiere definir una nueva clase `VentanaBarra` (ventana con barra de desplazamiento). Indique si definiría la nueva clase utilizando alguna de las anteriores o como una nueva clase independiente. En caso de utilizar alguna de las ya definidas explique qué relaciones son las que se establecen entre ellas y cómo las codificaría. Razone la respuesta.



Una clase especializada de `Ventana`, ya que una `VentanaBarra` es una `Ventana`.

Ejercicio 3

Dadas las clases A y B, indicar qué asignaciones son correctas:

```
1 class A { /* ... */ };
2 class B: public A { /* ... */ };

4 main() {
5     A objA, *pA;
6     B objB, *pB;
7     pA = &objA; Bien
8     pB = &objB; Bien
9     objA = objB; Bien. Conversión implícita
10    objA = (A)objB; Bien. Conversión explícita
11    objB = objA; Error. Conversión hacia abajo.
12    objB = (B)objA; Error. Conversión hacia arriba.
13    pA = pB; Bien.
14    pB = pA; Error. Conversión hacia abajo.
15    pB = (B*)pA; Bien. Conversión explícita entre punteros.
16 }
```

Estudiar sin publi es posible.

Compra Wuolah Coins y que nada te distraiga durante el estudio.



Ejercicio 4

Sea cierta clase base B y una derivada D. Ambas tienen definido un cierto método f(). Diga si el siguiente código es correcto y a qué método f() se llamaría.

1 B b, *bp;
2 D d, *dp;

4 b.f(); *Método f() de B (b.B::f())*

6 bp = &d;
7 bp->f(); *Método f() de B*

9 dp = &d;
10 dp->f(); *Método f() de D*



Ejercicio 5

Dadas las siguientes declaraciones :

```
1 struct A { int a; };
2 struct B : public A { int b; };
3 struct C : public A { int c; };
4 struct D : public B, public C { int d; } v;
```

¿Cuántos miembros tiene el objeto v? ¿Cómo se accede a cada uno de ellos?

Tiene 5.

{ v. B::a
v. C::a
v. b
v. c
v. d }

Seminario 4.1

Ejercicio 1

¿Es correcto (compilará) el siguiente programa?

```
1 #include <iostream>
2
3 using namespace std;
4
5 void mostrar(int i) { cout << i << "[entero]" << endl; }
6 void mostrar(float f) { cout << f << "[real]" << endl; }
7
8 int main()
9 {
10    mostrar(2); 2 [entero]
11    mostrar(2.0); Error por ambigüedad de conversión
12    mostrar('a'); 97 [real]
13 }
```

Ejercicio 3

Sea cierta clase base B y una derivada D. Ambas tienen definido un cierto método f(). Diga si el siguiente código es correcto; y, si lo es, a qué método f() se llamaría, dependiendo de que B::f() sea o no virtual.

No es virtual

- 1 B b, *bp;
- 2 D d, *dp;

- 4 bp = &d;
- 5 bp->f(); **B::f()**

- 7 dp = &b; *Error. Conversión implícita hacia abajo.*
- 8 dp->f();

- 10 dp = &d;
- 11 dp->f(); **D::f()**

Ejercicio 3

Sea cierta clase base B y una derivada D. Ambas tienen definido un cierto método f(). Diga si el siguiente código es correcto; y, si lo es, a qué método f() se llamaría, dependiendo de que B::f() sea o no virtual.

Sí es virtual

- 1 B b, *bp;
- 2 D d, *dp;

- 4 bp = &d;
- 5 bp->f(); **D::f()**

- 7 dp = &b; *Error. Conversión implícita hacia abajo*
- 8 dp->f();

- 10 dp = &d;
- 11 dp->f(); **D::f()**

Estudiar sin publi es posible.

Compra Wuolah Coins y que nada te distraiga durante el estudio.



Ejercicio 4

Indique qué enviará exactamente a la salida estándar el siguiente programa al ejecutarse:

```
1 #include <iostream>

3 struct B {
4     B() { std::cout << "Constructor de B\n"; }
5     virtual ~B() { std::cout << "Destructor de B\n"; }
6 };

8 struct D: B {
9     D() { std::cout << "Constructor de D\n"; }
10    ~D() { std::cout << "Destructor de D\n"; }
11 };

13 int main() {
14     B *pb = new D;
15     delete pb;
16 }
```

¿Cambiaría algo si quitamos la palabra `virtual` del destructor de B?

Se imprime:

Constructor de B
Constructor de D
Destructor de D
Destructor de B

Si quitamos `virtual`, saldría

Constructor de B
Constructor de D
Destructor de B



Ejercicio 5

Si tenemos las siguientes clases, ¿qué destructores son virtuales?

```
1 class A { public: ~A(); };
2 class B : public A { public: virtual ~B(); };
3 class C : public B { public: virtual ~C(); };
4 class D : public C { public: ~D(); };
```

Gl de B, C y D.

Ejercicio 6

¿Cambiaría el comportamiento de la clase cuadrado si le quitamos el miembro area()?

```
1 class rectangulo {  
2 public:  
3     rectangulo(double a, double l): ancho(a), largo(l) {}  
4     virtual double area() { return ancho * largo; }  
5 private:  
6     double ancho, largo;  
7 };  
  
9 class cuadrado: public rectangulo {  
10 public:  
11     cuadrado (double l): rectangulo(l, l) {}  
12     double area() { return rectangulo::area(); }  
13 };
```

No, ya que el área de un cuadrado se calcula de la misma manera que de un rectángulo, y hereda su método area().

Seminario 4.2

Exercise 1

Consider a base class `B` and a derived class `D`. Both classes have a member function `f()`, which is defined as **pure virtual** in class `B`, so `D::f()` overrides `B::f()`.

- ① Write the definition of `B::f()` (it takes no parameters and it returns no value).
- ② What is the kind of class `B` denominated?
- ③ What happens when the below code is compiled and executed? Is there a static (at compile-time) or dynamic (at runtime) binding in line 4?

```
1 B b, *bp;  
2 D d;  
3 bp = &d;  
4 bp->f();
```

- 1) `virtual void f() = 0;`
- 2) Clase abstracta
- 3) Hay un error ya que no se pueden crear objetos de clases abstractas.

Si se suprime ese error, habría en la zanja dinámico (método virtual)

Estudiar sin publi es posible.

Compra Wuolah Coins y que nada te distraiga durante el estudio.



Exercise 2

Consider the following class hierarchy, in which x, y and z inherit publicly from v:

```
1 struct V {  
2     virtual void fv() = 0;  
3     virtual ~V() {}  
4 };  
5 struct X : V {  
6     void fv() {}  
7 };  
8 struct Y : V {  
9     void fv() {}  
10};  
11 struct Z : V {  
12     void fv() {}  
13};
```

Exercise 2 (cont.)

Think about a function f() that processes objects of class v:

```
1 void f(V& v)  
2 {  
3     if (typeid(v) == typeid(X)) {  
4         std::cout << "Processing object X...\n";  
5         // specific code for X  
6     }  
7     if (typeid(v) == typeid(Y)) {  
8         std::cout << "Processing object Y...\n";  
9         // specific code for Y  
10}   
11    if (typeid(v) == typeid(Z)) {  
12        std::cout << "Processing object Z...\n";  
13        // specific code for Z  
14    }  
15 }
```

Exercise 2 (cont.)

- ① What is the output of the following instructions?

```
X x; V* pv = new Y;  
f(x); f(*pv);
```

- ② Is this the best way to implement the polymorphic behavior of f()? Justify your answer. Depending on your answer, describe how this implementation can be improved and, if necessary, modify the code to get the same output.



WUOLAH

1) Processing object X

Processing object Y

2) Sería más correcto con dynamic_cast:

```
void f(v& v) {  
    if(X* px = dynamic_cast<X*>(v)) {  
        std::cout << "Processing object X ... \n";  
    }  
    else if(Y* py = dynamic_cast<Y*>(v)) {  
        std::cout << "Processing object Y ... \n";  
    }  
    else if(Z* pz = dynamic_cast<Z*>(v)) {  
        std::cout << "Processing object Z ... \n";  
    }  
}
```

Exercise 3

- ① Define a class template Buffer to represent a storage area for items of the same type. Its template parameters are:

- the type of the stored items (by default, the type of size 1 byte);
- the storage capacity (by default, 256 items).

Define the Buffer's main data member (based on the STL container vector) and the default constructor.

- ② Then, define an object of type Buffer composed of 128 items of type int, and another one composed of 256 items of the default type.

1) `template <typename T = char, unsigned int i = 256 >`
`Buffer {`

`public:`

`Buffer();`

`private:`

`std::vector<T> (i) bi`

`};`

`template < typename T, unsigned int i > Buffer<T, i>;`

`Buffer() {}`

2)

`Buffer<int, 128> B1;`

`Buffer B2;`

Exercise 4

- ① Write the output of the following program.
- ② If classes `B<id>` were non-polymorphic types, what would be the result?

Perezza. El de Queso lo veo crema. :)