TareasTema3.pdf

Anónimo

Programación Orientada a Objetos

2º Grado en Ingeniería Informática

Escuela Superior de Ingeniería
Universidad de Cádiz

SEMINARIO – TAREA 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.

1. Dibujar el diagrama de clases
2. Implementar lo mínimo para que se reflejen las relaciones y clases
3. Clases: menú, opción y formulario.

- Un objeto Menú tiene varias Opciones, ejecuta una sola
- Un objeto Opción activa un Formulario
- Una Opción sólo está en un Menú → Composición
- Un Formulario puede activarse por varias Opciones

Si, así de simple

```
class Menu {
    public:
        typedef set<Opcion> Opciones;
    private:
        Opciones opciones;
}

class Opcion {
    public:

    private:
        Formulario* formulario;
}

class Formulario {
    public:
    private:
}
```

```
//Incorrecto
Class Menu {
    public:
        typedef set<Opcion> Opciones; //objetos, no punteros a objetos (composición)
        //...
        void ejecuta (Opcion& opcion); //enlaza con opcion
        const Opciones& ejecuta() const; //objetos Opcion enlazados
    private:
        Opciones opciones;
};
```

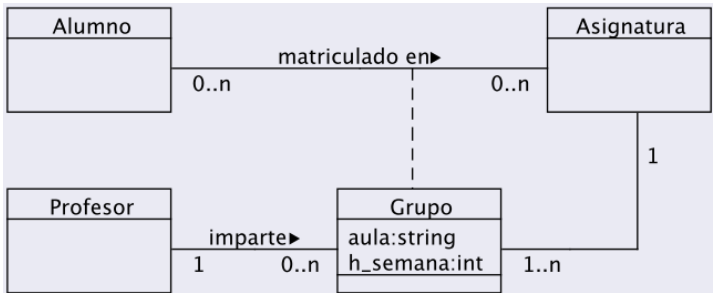
```
Class Formulario {
    public:
        typedef set<Opcion*> Opciones;
        //...
        void activado(Opcion& opcion); //enlaza con opcion
        const Opcion& activado() const; //objetos opcion enlazados
    private:
        Opciones opciones; //enlaces con objetos Opcion
};

class Opcion {
    public:
        void activa(Formulario& formulario); //enlaza con formulario
        Formulario& activa() const; //objeto formulario enlazado

        Al ser una composición, es unidireccional, no necesita saber el menú al que pertenece
        // void ejecutada(Menu& menu); //enlaza con menu
        // Menu& ejecutada() const; //objeto Menu enlazado
    private:
        Formulario* formulario;
        // Menu* menu;
}
```

Ejercicio 2

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




```
class Alumno{
    public:
        typedef map<Asignatura*, Grupo*> AsGs;
        void matriculado_en(Asignatura& as, Grupo& g);
    private:
        AsGs asgs;
}

class Asignatura{
    public:
        typedef map<Alumno*, Grupo*> AlGs;
        typedef set<Grupo*> Grupos;
        void se_matricula(Alumno& al, Grupo& g);
```

NUWE

Encuentra el trabajo de tus sueños

Participa en retos y competiciones de programación

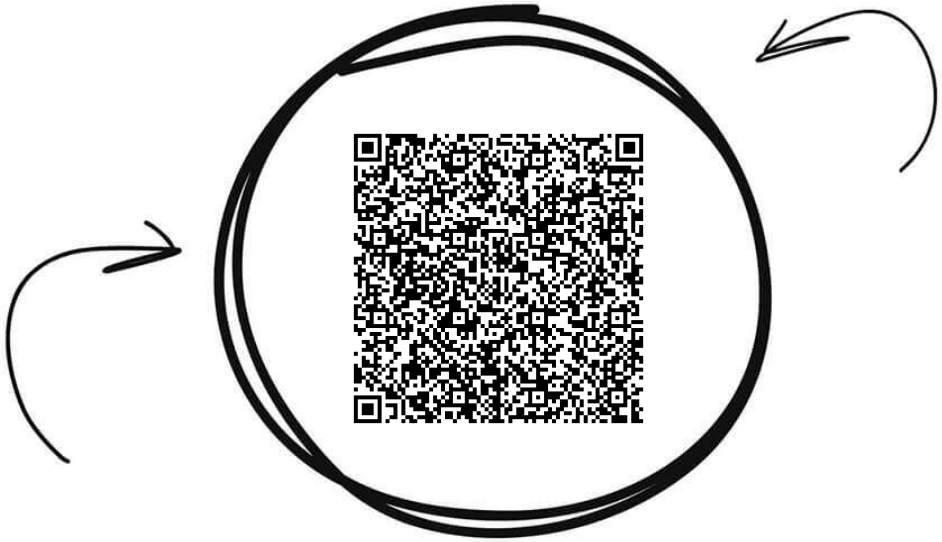
Ten contacto de calidad con empresas líderes en el sector tecnológico mientras vives una experiencia divertida y enriquecedora durante el proceso.

Únete ahora

Escanéame y obtén más info!!

hello@nuwe.io

Programación Orientada a Obj...



Banco de apuntes de la

WUOLAH

Comparte estos flyers en tu clase y consigue más dinero y recompensas

1. Imprime esta hoja
2. Recorta por la mitad
3. Coloca en un lugar visible para que tus compis puedan escanar y acceder a apuntes
4. Llévate dinero por cada descarga de los documentos descargados a través de tu QR

```
        void grup(Grupo& g);
    private:
         AlGs algs;
        Grupos grupos;
}

class Profesor{
    public:
        typedef set<Grupo*> Grupos;
        void imparte(Grupo& g);
    private:
        Grupos grupos;
}

class Grupo{
    public:
        void es_impartido(Profesor& p);
        void asig(Asignatura& a);
    private:
        Asignatura* asignatura;
        Profesor* profesor;
}
```

```
Class Alumno {
    public:
        typedef map<Asignatura*, Grupo*> Matriculados;
        void matriculado_en(Asignatura& asignatura, Grupo& grupo);
        const Asignaturas& matriculado_en() const;
    private:
        Matriculados matriculados;
}

Class Asignatura {
    public:
        typedef set<Grupo*> Grupos;
        void tiene(Grupo& grupo);
        const Grupos& tiene() const;
    private:
        Grupos grupos;
}
```

Alumno pertenece a 1 un grupo y solo 1 por cada asignatura en la que está matriculado.
Puede haber grupo sin alumnos.

```
Class Grupo {
    public:
        void pertenece(Asignatura& asignatura);
        Asignatura& pertenece() const;
    private:
        Asignatura* asignatura;
```

```
}

Class Profesor {
    public:
        typedef set<Grupo*> Grupos;
        void imparte(Grupo& grupo);
        const Grupos& imparte() const;
    private:
        Grupos grupos;
}
```

Ejercicio 3
Defina los dos métodos siguientes:

- **Alumno::matriculado_en() para matricular a un alumno en una asignatura asignándole un grupo.**

```
Alumno::matriculado_en(Asignatura& A, Grupo& G){
    matriculados.insert(make_pair(&A, &G));
}
```

- **Profesor::imparte() para vincular un profesor a un grupo.**

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

SEMINARIO T3

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 Alumno_Asignatura{
    public:
        void matriculado_en (Alumno& al, Asignatura& as, Grupo& g);
        void matriculado_en (Asignatura& as, Alumno& al, Grupo& g);
        map<Asignatura*, Grupo*> matriculados(Alumno& al) const;
        map<Alumno*, Grupo*> matriculados(Asignatura& as) const;
    private:
        std::map<Alumno*, std::map<Asignatura*, Grupo*>> directa;
        std::map<Asignatura*, std::map<Alumno*, Grupo*>> inversa;
};

void Alumno_Asignatura::matriculado_en( Alumno& al, Asignatura& as, Grupo& g)
{
    directa[&al].insert(make_pair(&as, &g));
```

```
    inversa[&as].insert(make_pair(&al, &g));

    //En set y map no comprobamos si está
    //directa.insert(make_pair(&al, map<Asignatura*, Grupo*>({make_pair(&as, &g)})));
    //inversa.insert(make_pair(&as, map<Alumno*, Grupo*>({make_pair(&al, &g)})));
}

void Alumno_Asignatura::matriculado_en (Asignatura& as, Alumno& al, Grupo& g)
{
    matriculado_en(al, as, g);
}

map<Asignatura*, Grupo*> Alumno_Asignatura::matriculados(Alumno& al) const
{
    typename std::map<Alumno*, std::map<Asignatura*, Grupo*>>::
        const_iterator i = directa.find(&al);
    if(i != directa.end()) return i→second;
    else return std:::map<Asignatura*, Grupo*>();
}

map<Alumno*, Grupo*> Alumno_Asignatura::matriculados(Asignatura& as) const
{
    typename std::map<Asignatura*, std::map<Alumno*, Grupo*>>::
        const_iterator i = inversa.find(&as);
    if(i != inversa.end()) return i→second;
    else return std:::map<Alumno*, Grupo*>();
}

-----------------------------------------------------------------------------------------------

class Alumno_Asignatura {
    public:
        void matriculado_en(Alumno& al, Asignatura& as, Grupo& g);
        void matriculado_en(Asignatura& as, Alumno& al, Grupo& g);
        std::map<Asignatura*, Grupo*> matriculados(Alumno& al);
        std::map<Alumno*, Grupo*> matriculados(Asignatura& as);
    private:
        std::map<Alumno*, std::map<Asignatura*, Grupo*>> AlAsGrups;
        std::map<Asignatura*, std::map<Alumno*, Grupo*>> AsAlGrups;
}

void Alumno_Asignatura::matriculado_en(Alumno& al, Asignatura& as, Grupo& g){
    //No permite modificar, en caso de que ya exista la asignatura/alumno y grupo, no hará nada
    AlAsGrups[&al].insert(make_pair(&as, &g);
    AlAsGrups[&as].insert(make_pair(&al, &g);

    //Permite modificar, siempre machacará el valor si existe y si no, lo crea
    AlAsGrups[&al][&as] = &g;
    AsAlGrups[&as][&al] = &g;

    //insert no hará nada si lo que buscas ya existe, mientras que corchetes siempre crea si no existe
y machaca si sí.
```

```
}

void Alumno_Asignatura::matriculado_en(Asignatura& as, Alumno& al, Grupo& g){
      matriculado_en(al, as, g);
}

std::map<Asignatura*, Grupo*> Alumno_Asignatura::matriculados(Alumno& al){
      auto i = directa.find(&al);
      if(i != directa.end()) return i→second;
      else return std::map<Asignatura*, Grupo*>();
}

std::map<Alumno*, Grupo*> matriculados(Asignatura& as){
      auto i = inversa.find(&as);
      if(i != inversa.end()) return i→second;
      else return std::map<Alumno*, Grupo*>();
}
```

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.

SET → UNO
MAP → PARES

```
class Profesor_Grupo {
      public:
            void imparte(Profesor& p, Grupo& g);
            void imparte(Grupo& g, Profesor& p);
            std::set<Grupo*> impartidos(Profesor& p);
            Profesor* impartidos(Grupo& p);

      private:
            std::map<Profesor*, set<Grupo*>> ProfGrups;
            std::map<Grupo*, Profesor*> GrupProf;
}


void Profesor_Grupo::imparte(Profesor& p, Grupo& g) {
      //Si el grupo no existe, lo crea y le asocia el profesor, en caso contrario, machaca el antiguo pro
      GrupProf[&g] = &p;
      //Si el prof existe le inserta el nuevo grupo, si no, lo crea y le inserta el nuevo grupo
      ProfGrups[&p].insert(&g)

      auto it = GrupProf.find(&g);   //Busco grupo en map si existe

      if(it !=  GrupProf.end())    //voy a su prof y le quito el grupo
            it->second.erase();
            it->second = &p;
```

```
    else
        GrupProf[&p]

    ProfGrups[&p].insert(&g);      //voy a nuevo profesor y le añado el grupo

    //voy al grupo y le quito el profesor
    //voy al grupo y le añado el nuevo profesor

    //Busco si dicho grupo ya tiene asignado un profesor (busco grupo en el map)
    //Si ya lo tiene asignado -> Borro dicha relación directa e inversamente y creo la nueva
    //Si no tiene asignado -> Creo la relación en ambas direcciones
    GrupProf[&g] = &p;
}
```

Romper el pacto reclamar mis 21gramos

```
void imparte(Grupo& g, Profesor& p){
    imparte(p, g);
}

std::set<Grupo*> impartidos(Profesor& p){
    auto it = ProfGrups.find(&p);
    if(it != ProfGrups.end()) return it→second;
    else return set<Grupo*>();
}

Profesor* impartidos(Grupo& p){
    auto it = GrupProfs.find(&p);
    if(it != GrupProfs.end()) return it→second;
                else { Profesor* p; return p; }
}
```

SEMINARIO T4

Ejercicio 1:
Primera opción es la correcta

Ejercicio 2:

```
class VentanaBarra(): public: Ventana {
    private:
        Barra b;
}
```

Es una ventana con cosas, herencia de Ventana.
¿Qué tipo de herencia? → Especialización
Ahora, ¿qué relación tiene VentanaBarra con Barra?
Relación de composición con Barra, es una Ventana especial compuesta de una Barra

Ejercicio 3:
Dadas las clases A y B, indicar qué asignaciones son correctas:

class A { /* … */ };
class B: public A { };

linea 8: correcta
linea 9:

main() {
5 A objA, *pA;
6 B objB, *pB;
7 pA = &objA;
8 pB = &objB;
9 objA = objB; → Correcto, conversión hacia arriba
10 objA = (A)objB; → Correcto, igual que en 9
11 objB = objA; → Mal, conversión hacia abajo, hay atributos que no tiene
12 objB = (B)objA; → Mal, sería correcto si tuvieramos un ctor especifico para ello
13 pA = pB; → Correcto, conversión hacia arriba
14 pB = pA; → Mal, conversión hacia abajo
15 pB = (B*)pA; → Correcto, caso especial porque conversión explícita, aunque podría dar errores
}

Ejercicio 4
Sea cierta clase base B y una derivada D . Ambas tienen de#nido un cierto método f() . Diga si el siguiente código es correcto y a qué método f() se llamaría.
El código es correcto, no hay problemas de compilación.

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

4 b.f(); → f() de B por ser la clase base

6 bp = &d; //puntero bp apunta a objeto de la clase D
7 bp→f(); → llama a f() de B por ser bp un puntero a B

9 dp = &d; ////puntero dp apunta a objeto de la clase D
10 dp→f(); → llama a f() de D por ser dp un puntero a D

dp→B::f(); → llama a f() de B gracias a la resolución de ámbito

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?
5 miembros: d, c, b, v.B::a y v.C::a.