FACULTAD DE INFORMÁTICA

Herencia

SUPLEMENTO

Programación orientada a objetos — Unidad 6

Autor: Luis Hernández Yáñez

FdI UCM

Contenido

FdI UCM

Un tercer modo de derivación

También existe el modo de derivación protegido.
Los miembros públicos y protegidos se convierten en protegidos.

```
clase Persona
protegido:
   _nif        _nombre
   _apellidos  _edad
público:
   nif(string)   nombre(string)
   apellidos(string)   edad(int)
   nif()   nombre()   apellidos()
   edad()   leer()   mostrar()
   nombreCompleto()
   felizCumple()

class Alumno : protected Persona
...
```

Protegido en **Alumno**. Accesible en sus funciones miembro. Inaccesible desde fuera de los objetos.

Protegido en **Alumno**. Accesible en sus funciones miembro. Inaccesible desde fuera de los objetos.

FdI UCM

Herencia múltiple

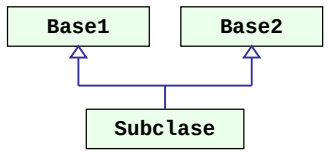
Se puede hacer que una clase se derive de más de una superclase.

Basta con incluir varias clases en la lista de derivación, cada una con su modo de derivación:

```
class Base1 {
  ...
};

class Base2 {
  ...
};

class Subclase : public Base1, public Base2 {
  ...
};
```



Herencia múltiple

Por supuesto, la subclase adopta las características de todas sus superclases.

Un problema que puede surgir es la *colisión de identificadores*:

```
class Base1 {
  void mostrar();
};
class Base2 {
  void mostrar()
};
class Subclase : public Base1, public Base2 {
  ...
};
```

Si la **Subclase** redefine los métodos **mostrar()** heredados, será su propia versión la que se use con sus objetos

Pero, ¿qué pasa si no?

Herencia múltiple: colisión de identificadores

Si la subclase no redefine los métodos heredados, se puede resolver la ambigüedad con el operador de resolución de ámbito:

```
Subclase obj1;
  ...
obj1.Base1::mostrar();
obj1.Base2::mostrar();
```

Si el compilador no puede resolver la ambigüedad, como en

```
obj1.mostrar();
```

notificará un error.

Herencia múltiple: un ejemplo

En el taller de la Unidad 4 creamos una clase **FechaHora** que servía para manipular instantes históricos. La creamos a partir de las clases **Fecha** y **Hora**. Como todavía no habíamos visto la herencia, construimos la clase **FechaHora** como una composición de las clases **Fecha** y **Hora**, de forma que cada objeto de clase **FechaHora** contuviera un objeto de clase **Fecha** y otro objeto de clase **Hora**. Esos objetos eran los atributos de la clase **FechaHora**.

Ahora que ya conocemos la herencia, podemos crear la clase **FechaHora** de otra forma. Dado que los objetos **FechaHora** son tanto **Fecha**s como **Hora**s, podemos hacer que la nueva clase herede de esas dos. Como derivaremos de más de una clase, lo que aplicaremos será herencia múltiple.

Para poder comparar después las dos alternativas, hemos dotado a la subclase **FechaHora** de un conjunto equivalente de servicios al que establecimos en aquel taller.

Herencia múltiple: un ejemplo

La clase **Fecha**

```
// Clase Fecha - Archivo de cabecera "fecha.h"

#ifndef fecha_h // Evitar inclusiones múltiples
#define fecha_h

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

class Fecha {
public:
  Fecha(int = 1, int = 1, int = 1900);
    // Constructor (día, mes, año)
  Fecha(const Fecha&);
  Fecha& operator=(const Fecha&);
  ~Fecha();

  int dia() const;
  int mes() const;  // Accedentes
  int anio() const;
```

(*continúa*)

Herencia múltiple: un ejemplo

```
  void dia(int);
  void mes(int);    // Mutadores
  void anio(int);

  long int operator-(Fecha) const;
    // Días transcurridos entre la fecha y la proporcionada
  Fecha operator+(long int) const; // Fecha más días
  Fecha operator-(long int) const; // Fecha menos días
  Fecha& operator+=(long int); // Abreviatura para la suma
  Fecha& operator-=(long int); // Abreviatura para la resta
  Fecha& operator++(); // Día siguiente
  Fecha& operator++(int); // (variante postfija)
  Fecha& operator--(); // Día anterior
  Fecha& operator--(int); // (variante postfija)

  bool operator==(Fecha) const;
  bool operator!=(Fecha) const;
  bool operator<(Fecha) const;   // Operadores relacionales
  bool operator<=(Fecha) const;
  bool operator>(Fecha) const;
  bool operator>=(Fecha) const;
```

(*continúa*)

Herencia múltiple: un ejemplo

```
  bool bisiesto() const;
  string diaSemana() const; // Devuelve el día de la semana
  string completa() const;
    // Devuelve la fecha en formato largo:
    // <día de la semana> <día> de <nombre del mes> de <año>
    // Por ejemplo: sabado 1 de abril de 2000
  string reducida() const;
    // Devuelve la fecha en formato corto:
    // <día>/<mes>/<año> (día y mes con 2 dígitos)
    // Por ejemplo: 01/04/2000
  void leer();

private:
  static int dias[12]; // días de cada mes - atributo de clase
  int _dia, _mes, _anio;
  void valida();
  long int diasDesdeInicio() const; // días desde el 01/01/1900
  void fechaTras(long int); // fecha desde el 01/01/1900 + días
};

#endif
```

>>> fecha.h

Herencia múltiple: un ejemplo

La clase **Hora**

```
// Clase Hora - Archivo de cabecera "hora.h"

#ifndef hora_h // Evitar inclusiones múltiples
#define hora_h

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

class Hora {
public:
   Hora(int = 0, int = 0, int = 0);
     // Constructor (horas, minutos, segundos)
  Hora(const Hora&);
  Hora& operator=(const Hora&);
  ~Hora();
  Hora(long int); // Otro constructor:construye la hora
    // desde la medianoche pasados los segundos indicados.
```

(*continúa*)

Herencia múltiple: un ejemplo

```
  void horas(int);
  void minutos(int);   // Mutadores
  void segundos(int);

  int horas() const;
  int minutos() const; // Accedentes
  int segundos() const;

  long int operator-(Hora) const;
    // Segundos transcurridos entre la hora y la proporcionada
  Hora operator+(long int) const; // Hora más segundos
  Hora operator-(long int) const; // Hora menos segundos
  Hora& operator+=(long int); // Abreviatura para la suma
  Hora& operator-=(long int); // Abreviatura para la resta
  Hora& operator++(); // Un segundo más
  Hora& operator++(int); // (versión postfija)
  Hora& operator--(); // Un segundo menos
  Hora& operator--(int); // (versión postfija)
```

(*continúa*)

Herencia múltiple: un ejemplo

```
  bool operator==(Hora) const;
  bool operator!=(Hora) const;
  bool operator<(Hora) const;    // Operadores relacionales
  bool operator<=(Hora) const;
  bool operator>(Hora) const;
  bool operator>=(Hora) const;

  string hora() const; // Devuelve la hora en una cadena
  void leer();

private:
  int _horas, _minutos, _segundos;
  void valida();
  void deSegundos(long int);
  long int aSegundos() const;
};

#endif
```

>>> hora.h

Herencia múltiple: un ejemplo

La subclase **FechaHora**

```
// FechaHora.h
#ifndef fechahora_h
#define fechahora_h

#include <iostream>
#include <string>
using namespace std;
#include "fecha.h"
#include "hora.h"

class FechaHora : public Fecha, public Hora {
public:
  FechaHora(); // Constructor predeterminado
               // 00:00:00 del 1-1-1900, por defecto
  FechaHora(const FechaHora&); // Constructor de copia
  FechaHora& operator=(const FechaHora&); // Op. asig.
  ~FechaHora(); // Destructor
```

Herencia múltiple

(*continúa*)

Herencia múltiple: un ejemplo

```
  // Otros constructores:
  FechaHora(Fecha f);
  FechaHora(Hora h);
  FechaHora(Fecha f, Hora h);

  // Fechas y horas
  Fecha fecha() const;
  Hora hora() const;
  void fecha(Fecha);
  void hora(Hora);

  // Operadores aritméticos:
  long int operator-(FechaHora) const;
  // Segundos entre la FechaHora y la proporcionada

  // long int operator-(Hora) const; se hereda de Hora
  // long int operator-(Fecha) const; se hereda de Fecha
```

(*continúa*)

Herencia múltiple: un ejemplo

```
  FechaHora operator+(long int) const; // + segundos
  FechaHora operator-(long int) const; // - segundos
  FechaHora& operator++(); // Un segundo más
  FechaHora& operator--(); // Un segundo menos

  // Operadores relacionales:
  bool operator==(FechaHora) const;
  bool operator!=(FechaHora) const;
  bool operator<(FechaHora) const;
  bool operator<=(FechaHora) const;
  bool operator>(FechaHora) const;
  bool operator>=(FechaHora) const;

  // Otros métodos:
  long int aDias(long int); // Segundos a días
  long int aSegundos(long int); // Días a segundos
```

(*continúa*)

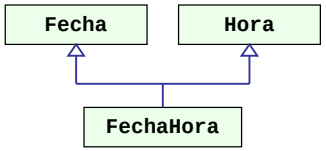
Herencia múltiple: un ejemplo

```
    // string diaSemana() const; se hereda de Fecha

    string completa() const;
    // Cadena con la fecha completa seguida de la hora
    string reducida() const;
    // Cadena con la fecha reducida seguida de la hora
    void leer();

private:
    static long int SPD; // segundos por día (86400)
};

#endif
```



>>> fechahora.h

Herencia múltiple: un ejemplo

La subclase **FechaHora** adopta todos los atributos de sus superclases **Fecha** y **Hora** como atributos propios:

_horas, **_minutos**, **_segundos**, **_dia**, **_mes**, **_anio**, **_dias** (de clase).

Tan sólo declara un único atributo adicional, **SPD** (de clase).

La clase **FechaHora** adopta todos los métodos de sus superclases **Fecha** y **Hora**.

De la interfaz deseada (la misma que establecimos en el taller de la Unidad 4), la clase **FechaHora** hereda directamente tres métodos: **operator-(Fecha)**, **operator-(Hora)** y **diaSemana()**.

Herencia múltiple: un ejemplo

La clase **FechaHora** redefine los siguientes métodos de sus superclases **Fecha** y **Hora**:

De **Fecha** (además de los operadores relacionales):

+ - ++ -- completa() reducida() leer()

De **Hora** (además de los operadores relacionales):

+ - ++ -- aSegundos() leer()

Veamos cómo es la implementación de la clase **FechaHora**. Como merece la pena comparar esta alternativa, la herencia múltiple, frente a la composición del taller de la Unidad 4, a la derecha, con fondo gris, se muestran las implementaciones de aquel taller.

Herencia múltiple: un ejemplo

En gris, con composición

Implementación de **FechaHora**

```
#include "fechahora.h"

long FechaHora::SPD = 86400; // Igual en la otra implementación

FechaHora::FechaHora() { } // Igual en la otra implementación

FechaHora::FechaHora(const FechaHora& otra)
  : Fecha(otra), Hora(otra) { }                    { _fecha = otra._fecha;
                                                     _hora = otra._hora; }

FechaHora& FechaHora::operator=(const FechaHora& otra) {
  Fecha::operator=(otra);                          { _fecha = otra._fecha;
  Hora::operator=(otra);                             _hora = otra._hora;
  return *this;                                      return *this; }
}

FechaHora::~FechaHora() { } // Igual en la otra implementación

FechaHora::FechaHora(Fecha f) : Fecha(f) { }      : _fecha(f) { }
```

(*continúa*)

Herencia múltiple: un ejemplo

En gris, con composición

```
FechaHora::FechaHora(Hora h) : Hora(h) { }          : _hora(h) { }

FechaHora::FechaHora(Fecha f, Hora h) : Fecha(f), Hora(h) { }
                                          : _fecha(f), _hora(h) { }
Fecha FechaHora::fecha() const
{ return Fecha(dia(), mes(), anio()); }          { return _fecha; }

Hora FechaHora::hora() const
{ return Hora(horas(), minutos(), segundos()); { return _hora; }

void FechaHora::fecha(Fecha f) {
  dia(f.dia()); mes(f.mes()); anio(f.anio());
}
                                                 { _fecha = f; }

void FechaHora::hora(Hora h) {
  horas(h.horas()); minutos(h.minutos());
  segundos(h.segundos());                        { _hora = h; }
}
```

(*continúa*)

Herencia múltiple: un ejemplo

En gris, con composición

```
long int FechaHora::operator-(FechaHora otra) const {
// Segundos entre la FechaHora y la proporcionada
  return Fecha::operator-(otra) * SPD + Hora::operator-(otra);
}
 { return (_fecha - otra._fecha) * SPD + (_hora - otra._hora); }

// long int operator-(Hora otra) const; se hereda de Hora
                                          {  return _hora - otra; }

// long int operator-(Fecha otra) const; se hereda de Fecha
                                          { return _fecha - otra; }
```

Los métodos redefinidos todavía se pueden acceder

Hora:: **Fecha::**

(*continúa*)

Herencia múltiple: un ejemplo

En gris, con composición

```
FechaHora FechaHora::operator+(long int seg) const {
  Fecha f = fecha();
  Hora h = hora();                  { FechaHora fh;
  f = f + seg / SPD;                  fh._fecha = _fecha + seg / SPD;
  seg = seg % SPD;                    seg = seg % SPD;
  h = h + seg;                        fh._hora = _hora + seg;
  if(h < hora()) f++;                 if(fh._hora < _hora) fh._fecha++;
  return FechaHora(f, h);             return fh;
}                                   }

FechaHora FechaHora::operator-(long int seg) const {
  Fecha f = fecha();
  Hora h = hora();                  { FechaHora fh;
  f = f - seg / SPD;                  fh._fecha = _fecha - seg / SPD;
  seg = seg % SPD;                    seg = seg % SPD;
  h = h - seg;                        fh._hora = _hora - seg;
  if(h > hora()) f--;                 if(fh._hora > _hora) fh._fecha--;
  return FechaHora(f, h);             return fh;
}                                   }
```

(*continúa*)

Herencia múltiple: un ejemplo

En gris, con composición

```
FechaHora& FechaHora::operator++() { // Un segundo más
  Hora::operator++();
  if(hora() == Hora(0,0,0)) Fecha::operator++(); // Día sig.
  return *this;
}                                   { _hora++;
                                      if(_hora == Hora(0,0,0)) _fecha++;
                                      return *this; }

FechaHora& FechaHora::operator--() { // Un segundo menos
  if(hora() == Hora(0,0,0)) Fecha::operator--(); // Día ant.
  Hora::operator--();
  return *this;                     { if(_hora == Hora(0,0,0)) _fecha--;
}                                     _hora--;
                                      return *this; }

bool FechaHora::operator==(FechaHora otra) const {
  return Fecha::operator==(otra) && Hora::operator==(otra);
}
   { return (_fecha == otra._fecha) && (_hora == otra._hora); }
```

(*continúa*)

Herencia múltiple: un ejemplo

En gris, con composición

```
bool FechaHora::operator!=(FechaHora otra) const {
  return Fecha::operator!=(otra) || Hora::operator!=(otra);
}
```

```
{ return (_fecha != otra._fecha) || (_hora != otra._hora); }
```

```
bool FechaHora::operator<(FechaHora otra) const {
  return Fecha::operator<(otra) ||
         Fecha::operator==(otra) && Hora::operator<(otra);
}
```

```
{ return (_fecha < otra._fecha) ||
         (_fecha == otra._fecha) && (_hora < otra._hora); }
```

```
bool FechaHora::operator<=(FechaHora otra) const {
  return Fecha::operator<(otra) ||
         Fecha::operator==(otra) && Hora::operator<=(otra);
}
```

```
{ return (_fecha < otra._fecha) ||
         (_fecha == otra._fecha) && (_hora <= otra._hora); }
```

(*continúa*)

Herencia múltiple: un ejemplo

En gris, con composición

```
bool FechaHora::operator>(FechaHora otra) const {
  return Fecha::operator>(otra) ||
         Fecha::operator==(otra) && Hora::operator>(otra);
}
```

```
{ return (_fecha > otra._fecha) ||
         (_fecha == otra._fecha) && (_hora > otra._hora); }
```

```
bool FechaHora::operator>=(FechaHora otra) const {
  return Fecha::operator>(otra) ||
         Fecha::operator==(otra) && Hora::operator>=(otra);
}
```

```
{ return (_fecha > otra._fecha) ||
         (_fecha == otra._fecha) && (_hora >= otra._hora); }
```

```
long int FechaHora::aDias(long int seg) { return seg / SPD; }
// Igual en la otra implementación

long int FechaHora::aSegundos(long int dias)
{ return dias * SPD; } // Igual en la otra implementación
```

(*continúa*)

Herencia múltiple: un ejemplo

En gris, con composición

```
string FechaHora::completa() const {
// Cadena con la fecha completa seguida de la hora
  return Fecha::completa() + " " + Hora::hora();
}
```

```
{ return _fecha.completa() + " " + _hora.hora(); }
```

```
string FechaHora::reducida() const {
// Cadena con la fecha reducida seguida de la hora
  return Fecha::reducida() + " " + Hora::hora();
}
```

```
{ return _fecha.reducida() + " " + _hora.hora(); }
```

```
void FechaHora::leer() {
  Fecha::leer();
  Hora::leer();
}
```

```
{ _fecha.leer();
  _hora.leer(); }
```


Herencia múltiple: un ejemplo

En la clase **FechaHora** se produce una colisión de identificadores: método **valida()** de **Fecha** y método **valida()** de **Hora**.

En este caso no se producen problemas, ya que cada uno de esos métodos se invoca sólo desde algún otro método de su misma clase, por lo que la ambigüedad no llega a producirse.
Además, fuera de los objetos de la clase **FechaHora** no se pueden acceder esos métodos **valida()**.

En realidad hay otras colisiones, pero quedan resueltas por las redefiniciones efectuadas en la subclase:

Por ejemplo, el operador suma (**+**) está definido en las dos superclases, pero como también se ha definido en la subclase, siempre se ejecutará el propio.

Herencia frente a composición

Hemos visto dos implementaciones de la clase **FechaHora**: mediante composición (taller de la Unidad 4) y con herencia múltiple (aquí).

Mediante herencia, cada objeto de la clase **FechaHora** es una **Fecha** y también es una **Hora**.

Mediante composición, el objeto contiene una **Fecha** y una **Hora**.

Mediante herencia, la nueva clase parte ya con una funcionalidad heredada, funcionalidad que se amplía o reemplaza en su interfaz.

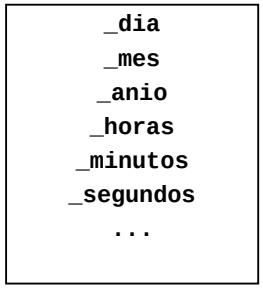
Mediante composición, la nueva clase parte de cero en cuanto a funcionalidad se refiere, funcionalidad que debe construirse por completo en su interfaz.

Herencia frente a composición

El enfoque es claramente distinto:

FechaHora como subclase de **Fecha** y de **Hora**

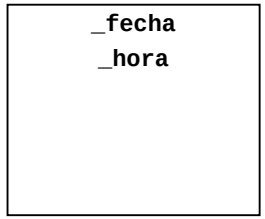
unaFechaHora



Adopta todos los servicios de las clases **Fecha** y **Hora**

FechaHora con un atributo de **Fecha** y otro de **Hora**

unaFechaHora



Los servicios de las clases **Fecha** y **Hora** están disponibles sólo a través de los atributos

Herencia frente a composición

La implementación de los métodos en el enfoque de composición se basa en pasos de mensajes a los atributos (métodos de las clases **Fecha** y **Hora**), mientras que con herencia se usan directamente los métodos heredados de las clases **Fecha** y **Hora**.

FechaHora como subclase de **Fecha** y de **Hora**:

```
bool FechaHora::operator==(FechaHora otra) const
{
  return Fecha::operator==(otra) &&
         Hora::operator==(otra);
}
```

FechaHora con atributos de **Fecha** y de **Hora**:

```
bool FechaHora::operator==(FechaHora otra) const
{
  return (_fecha==otra._fecha) && (_hora==otra._hora);
}
```


Herencia frente a composición

Los dos enfoques también se distinguen por los servicios que finalmente presta la clase **FechaHora** en cada caso.

Hemos desarrollado la clase **FechaHora** mediante herencia de forma que proporcionara los mismos servicios que la desarrollada mediante composición en el taller de la Unidad 4. Con composición tuvimos que crear todos los métodos. Con herencia nos ahorramos tres de ellos, ya que se heredan directamente de las superclases (la resta de una **Fecha**, la resta de una **Hora** y **diaSemana()**).

Y con el enfoque de herencia, la subclase **FechaHora** dispone de unos servicios adicionales heredados que son los siguientes:

dia() **mes()** **anio()** **horas()** **minutos()** **segundos()**
dia(int) **mes(int)** **anio(int)** **horas(int)** **minutos(int)**
segundos(int) **bisiesto()** **hora()**

Y los seis operadores relacionales para comparar con una **Fecha** o con una **Hora**.