Programación Orientada a Objetos

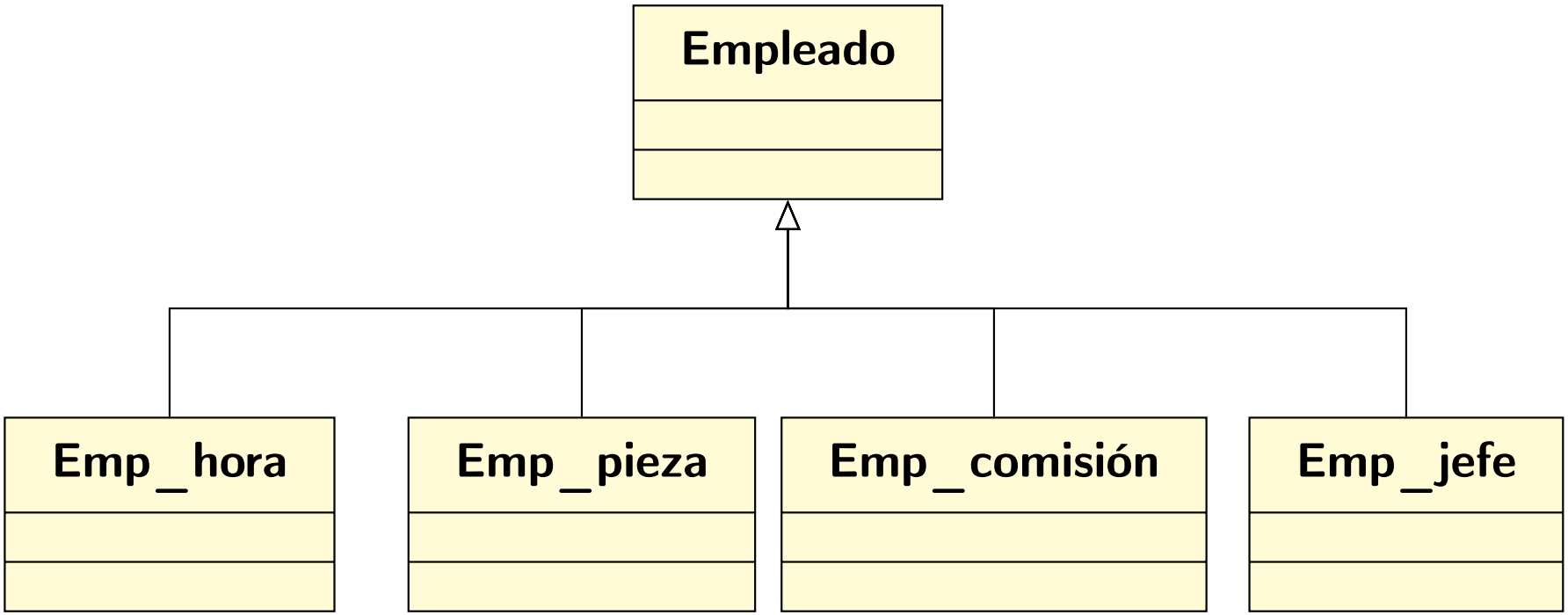
Tema 3. Relaciones entre clases. Parte II

José Fidel Argudo Argudo Francisco Palomo Lozano
Inmaculada Medina Bulo Gerardo Aburruzaga García

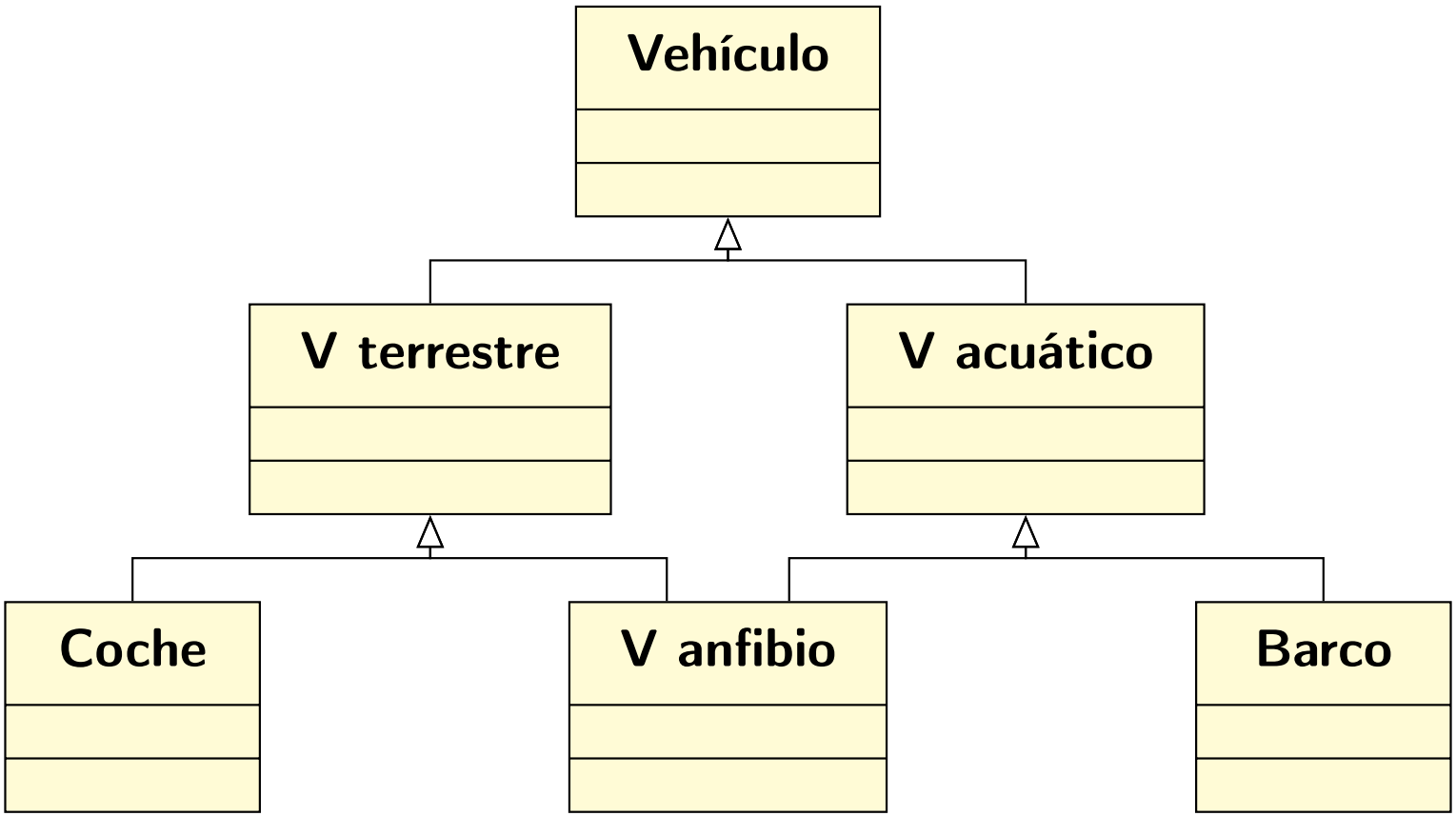
Versión 2.0

Índice

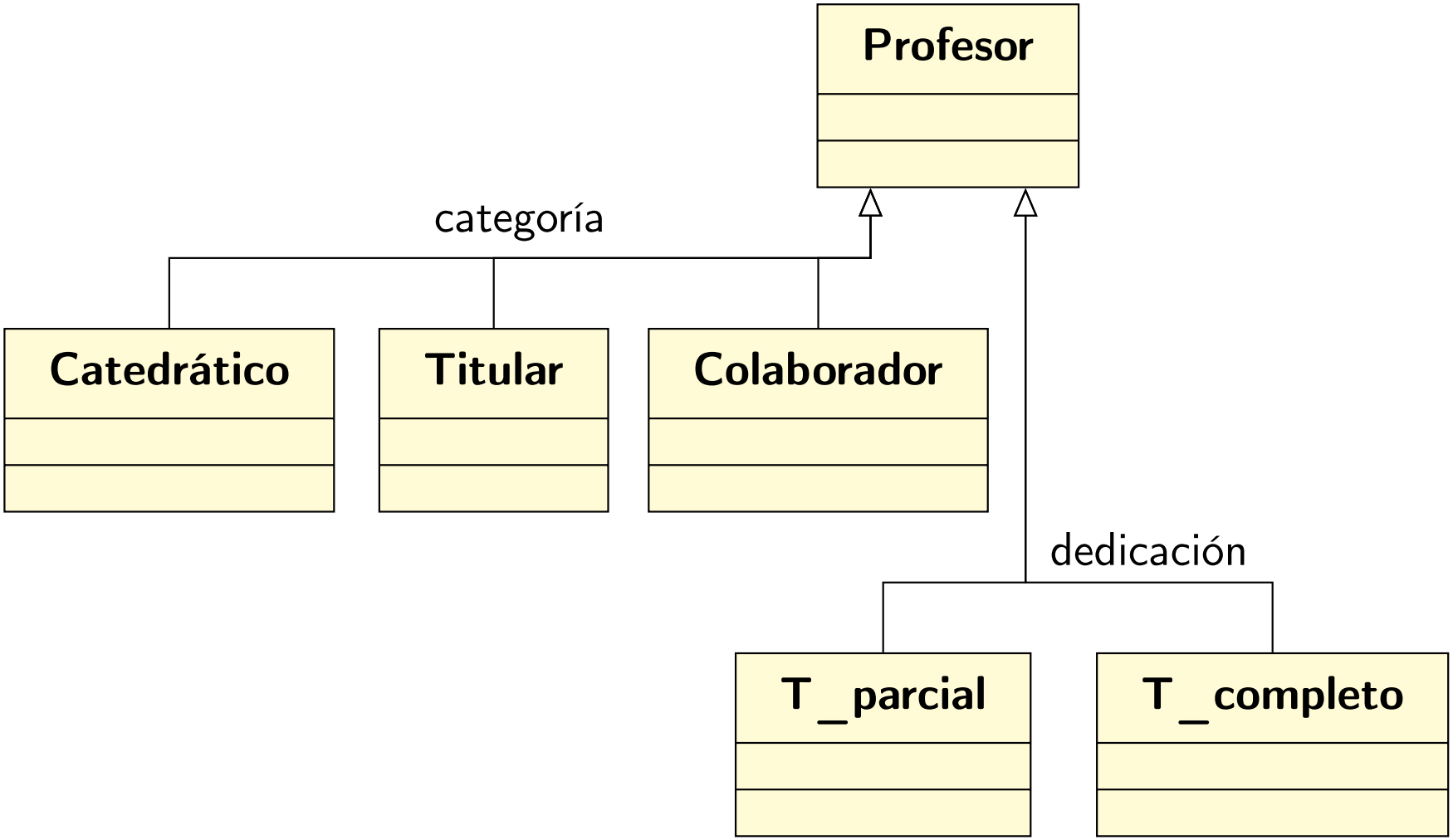
Generalización



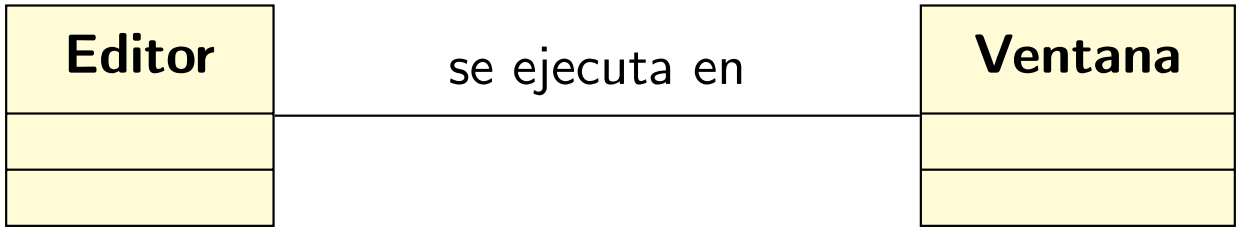
Generalización múltiple

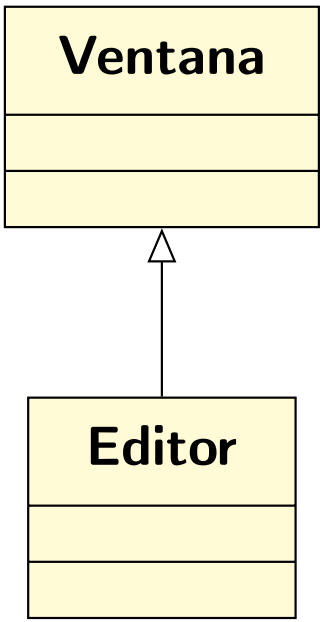


Generalización según criterios independientes



Asociación vs. generalización





Cabe plantearse algunas preguntas:

- ¿Es un editor un tipo especial de ventana, una ventana editable?
- ¿Es posible usar un editor en todas aquellas situaciones en que se emplea una ventana?
- ¿Y si un editor se va a ejecutar simultáneamente en dos o más ventanas, cada una mostrando una vista diferente del mismo?

Herencia simple

```
class clase-derivada [final]: [accesibilidad] clase-base
{
  // declaraciones de miembros
};
```

Una clase derivada hereda todos los miembros de la clase base menos:

- Constructores
- Destructor
- Operadores de asignación

Control de acceso a miembros

```
1  class C {
2  public:
3    int publico;    // Accesible desde el interior y exterior.
4
5  protected:         // Accesible desde el interior, para funciones
6    int protegido; // amigas de C y desde las clases derivadas de C
7                     // y sus funciones amigas.
8  private:
9    int privado;    // Accesible desde el interior y para las funciones
10                    // amigas de C.
11 };
```

Modos de acceso a los miembros heredados

Accesibilidad de la herencia	Un miembro... de la clase base	pasa a ser... en la derivada
`public` (por omisión en `struct`)	público protegido privado	público protegido inaccesible
`protected`	público protegido privado	protegido protegido inaccesible
`private` (por omisión en `class`)	público protegido privado	privado privado inaccesible

Herencia simple

Ejemplo

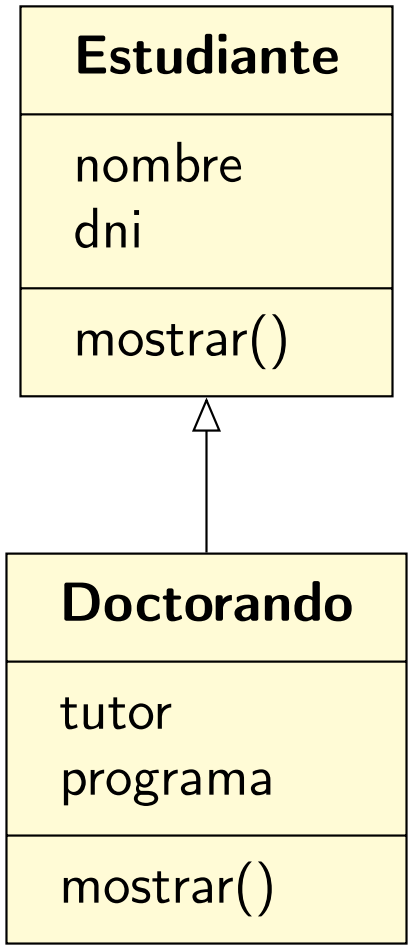
```
1  class Base {
2  public:
3    int publico;
4  protected:
5    int protegido;
6  private:
7    int privado;
8  };
```

```
10 class DerivadaPublica: public Base {
11   // publico es público
12   // protegido es protegido
13   // privado es  inaccesible
14 };

16 class DerivadaProtegida: protected Base {
17   // publico es protegido
18   // protegido es protegido
19   // privado es  inaccesible
20 };

22 class DerivadaPrivada: private Base {
23   // publico es privado
24   // protegido es privado
25   // privado es  inaccesible
26 };
```

Herencia pública

- La herencia pública representa una relación de generalización/especialización entre la clase base o *superclase* y las clases derivadas o *subclases*. La clase base es una generalización de sus derivadas, las cuales, a su vez, son especializaciones de la clase base.
- La relación entre las clases es visible desde el exterior por estar representada mediante herencia pública. Ello significa que un objeto de una subclase también *es un* objeto de la superclase. De hecho, la herencia pública define intrínsecamente la conversión de los tipos de las derivadas al tipo de la base, aunque esto conlleve, por lo general, pérdida de información.
- Con otras palabras, la herencia pública implica que un objeto de una clase derivada puede ser tratado como uno de su base y, por tanto, allí donde sea factible utilizar un objeto de la base puede sustituirse por otro de una de sus derivadas. (Principio de sustitución de Liskov)

Especialización de Estudiante en Doctorando



Especialización de Estudiante en Doctorando (estudiante.h)

```
1   #ifndef ESTUDIANTE_H_
2   #define ESTUDIANTE_H_

4   #include <iostream>
5   #include <string>
6   using namespace std;

8   class Estudiante {
9   public:
10    Estudiante(string nombre, int dni);
11    void mostrar() const;
12  private:
13    string nombre;   // nombre completo
14    int dni;         // DNI
15    // ...
16  };

18  #endif
```

Especialización de Estudiante en Doctorando (doctorando.h)

```
1   #ifndef DOCTORANDO_H_
2   #define DOCTORANDO_H_

4   #include "estudiante.h"
5   #include <string>
6   using namespace std;

8   class Doctorando final: public Estudiante {
9   public:
10    Doctorando(string nombre, int dni, string tutor, int programa);
11    void mostrar() const;
12  private:
13    string tutor;    // tutor en el programa de doctorado
14    int programa;    // código del programa
15    // ...
16  };

18  #endif
```

Especialización Estudiante–Doctorando (doctorando.cpp)

```
1   #include "doctorando.h"
2   #include <iostream>
3   #include <string>
4   using namespace std;

6   Doctorando::
7   Doctorando(string nombre, int dni, string tutor, int programa)
8     : Estudiante(nombre, dni), tutor(tutor), programa(programa) {}

10  void Doctorando::mostrar() const
11  {
12    // Muestra los datos que posee como estudiante
13    Estudiante::mostrar();
14    // Y los específicos de su condición de doctorando
15    cout << "Programa de doctorado: " << programa << "\n"
16         << "Tutor en el programa: " << tutor << endl;
17  }
```

Especialización de Estudiante en Doctorando (prueba.cpp)

```
1   #include "estudiante.h"
2   #include "doctorando.h"

4   int main()
5   {
6     Estudiante e("María␣Pérez␣Sánchez", 31682034);
7     Doctorando d("José␣López␣González", 32456790,
8                  "Dr.␣Juan␣Jiménez", 134);

10    Estudiante* pe = &e;
11    pe->mostrar();                  // e.mostrar()
12    pe = &d;                        // Conversión «hacia arriba»
13    pe->mostrar();                  // d.Estudiante::mostrar()

15    Doctorando* pd = &d;
16    pd->mostrar();                   // d.Doctorando::mostrar()
17    pd->Estudiante::mostrar();      // d.Estudiante::mostrar()
18    pd->Doctorando::mostrar();      // d.Doctorando::mostrar()
19  }
```

Especialización Estudiante–Doctorando (conversiones.cpp)

```
1   #include "estudiante.h"
2   #include "doctorando.h"

4   int main()
5   {
6     Estudiante e("María␣Pérez␣Sánchez", 31682034), *pe;
7     Doctorando d("José␣López␣González", 32456790,
8                  "Dr.␣Juan␣Jiménez", 134), *pd;
9     // Conversiones entre punteros
10    pe = &d;                                    // Bien
11    pd = pe;                                    // ERROR
12    pd = static_cast<Doctorando*>(pe);          // Bien
13    // Conversiones entre objetos
14    e = d;                                      // Bien
15    d = e;                                      // ERROR
16    d = Doctorando(e);                          // ERROR
17    d = static_cast<Doctorando>(e);             // ERROR
18    d = reinterpret_cast<Doctorando>(e);    // ERROR
19  }
```

Herencia protegida y privada

- La herencia privada (y también la protegida) representa una relación del tipo *se implementa como*: un objeto de la clase derivada *se implementa como* un objeto de la clase base, es decir, el subobjeto de la clase base es un detalle de implementación del objeto de la derivada.
- La relación de especialización es invisible desde el exterior por estar representada mediante herencia privada. Dentro de la derivada (y de sus amigas) ésta se ve como una subclase de la base y, por tanto, un objeto de la derivada puede ser convertido en un objeto de la base.
- Con herencia protegida la relación se hace visible para las derivadas, para las clases derivadas de las derivadas y también para sus respectivas amigas, pero no para el resto.
- La herencia privada (y protegida) se puede emplear como recurso para implementar una relación de composición.

Composición mediante herencia privada

```
1  #ifndef LISTA_H_
2  #define LISTA_H_
3  #include <deque>

5  class Lista {
6  public:
7    bool vacia() const;
8    int primero() const;
9    int ultimo() const;
10   void insertarPrincipio(int e);
11   void insertarFinal(int e);
12   void eliminarPrimero();
13   void eliminarUltimo();
14   void mostrar() const;
15 private:
16   std::deque<int> l;
17 };

19 #endif  // LISTA_H_
```

Composición mediante herencia privada

```
1  #ifndef PILA_H_
2  #define PILA_H_
3  #include "lista.h"
4  class Pila: private Lista {
5  public:
6    // Métodos de Lista que se hacen públicos
7    using Lista::vacia;
8    using Lista::mostrar;
9    // Métodos de Pila que delegan en los de Lista
10   int cima() const;
11   void apilar(int e);
12   void desapilar();
13 };
14 // Delegación de operaciones
15 inline int Pila::cima() const { return primero(); }
16 inline void Pila::apilar(int e) { insertarPrincipio(e); }
17 inline void Pila::desapilar() { eliminarPrimero(); }
18 #endif  // PILA_H_
```

Composición mediante herencia privada

```
1  #ifndef VECTOR_H
2  #define VECTOR_H
3  class Vector {
4  public:
5    using T = double;
6    explicit Vector(int n);
7    Vector(const Vector& V);
8    Vector& operator =(const Vector& V);
9    Vector(Vector&& V);
10   Vector& operator =(Vector&& V);
11   ~Vector();
12   T& operator [](int i); // Comprueba 0 <= i < n
13   const T& operator [](int i) const; // Comprueba 0 <= i < n
14   int longitud() const;
15   void mostrar() const;
16 protected: // Exponer atributos viola la ocultación de información
17   int n;    // longitud
18   T* v;     // ptro. al primer elto .
19 };
20 #endif    // VECTOR_H
```

Composición mediante herencia privada

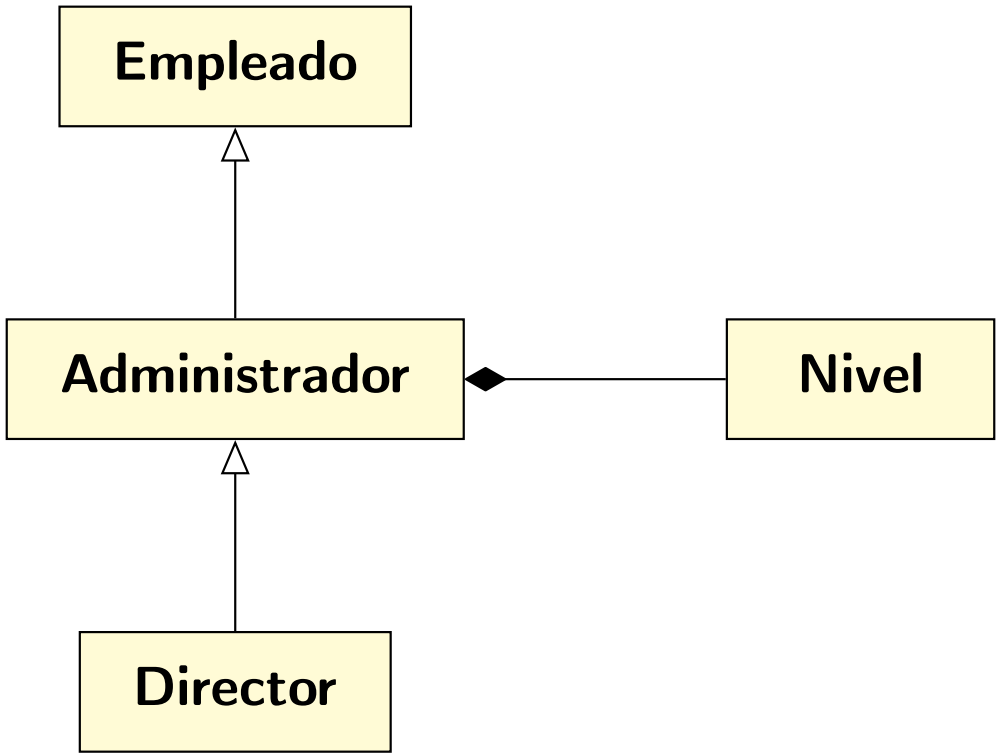
```
1  #ifndef SUPERVECTOR_H
2  #define SUPERVECTOR_H
3  #include "Vector.h"
4  class SuperVector: private Vector {
5  public:
6    explicit SuperVector(int i = 0, int s = 0)
7      : Vector{s - i + 1}, inf{i}, sup{s} {}
8    T& operator [](int i) {
9      // Delegación:
10     // return Vector::operator [ ](i - inf);
11     // Alternativa :
12     assert(i >= inf && i <= sup);
13     return v[i - inf];    // Bien, v es protegido
14   }
15   const T& operator [](int i) const {
16     assert(i >= inf && i <= sup);
17     return v[i - inf];
18   }
```

Composición mediante herencia privada

```
19    int limiteInferior() const { return inf; }
20    int limiteSuperior() const { return sup; }
21    using Vector::longitud;
22    using Vector::mostrar;
23  private:
24    int inf,    // límite inferior
25        sup;    // límite superior
26  };
27  #endif // SUPERVECTOR_H
```

Jerarquía de clases

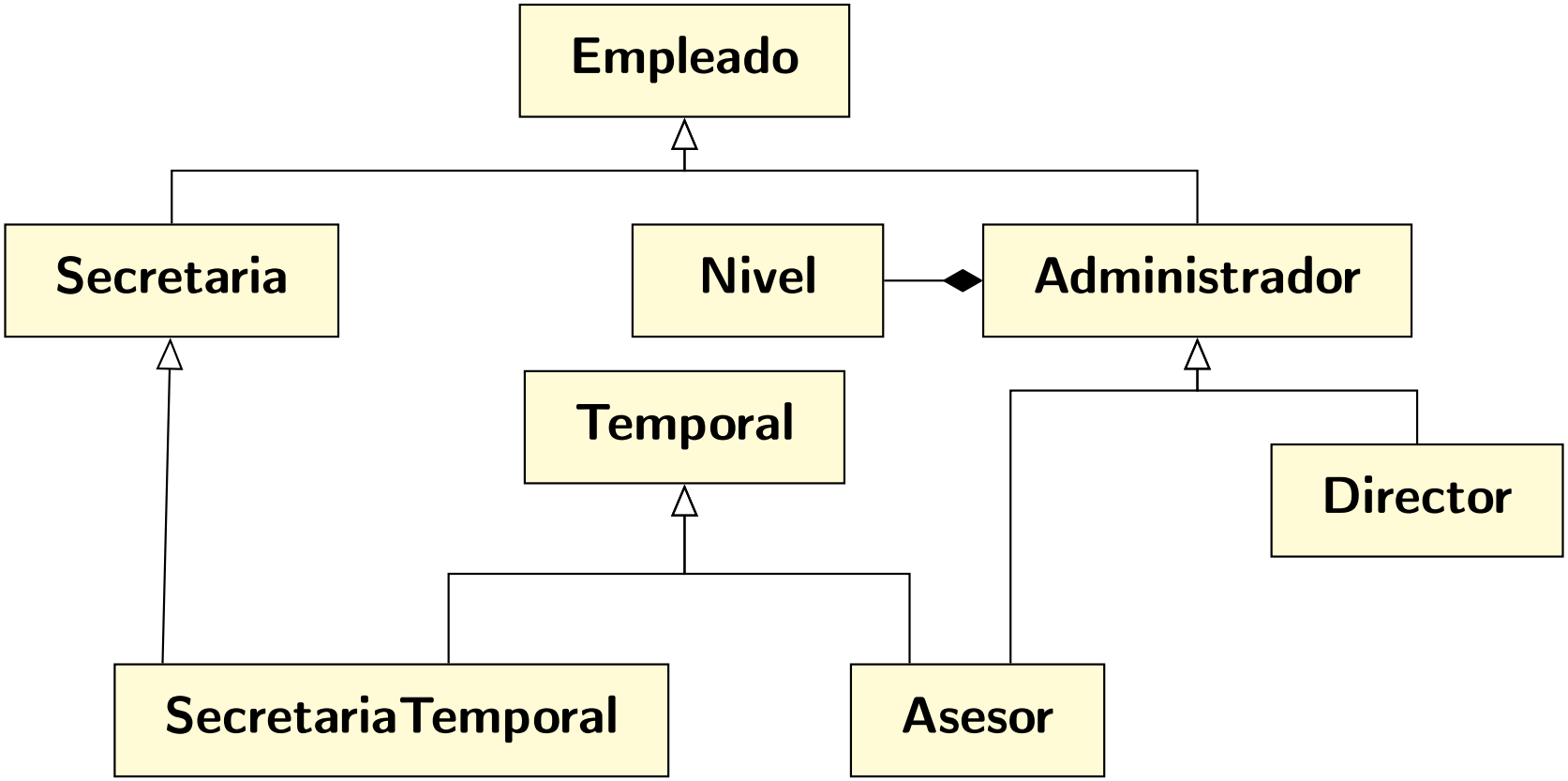
- La herencia simple permite definir una jerarquía de clases relacionadas, en la que cada una se deriva de una sola clase base. La cadena de derivación no puede ser circular.



Jerarquía de clases

```
1  class Empleado {
2    // ...
3  };

5  class Administrador: public Empleado {
6  protected:
7    Nivel nivel;
8    // ...
9  };

11 class Director final: public Administrador {
12   // ...
13 };
```

Herencia múltiple



Herencia múltiple

```
1  class Temporal {
2    // ...
3  };

5  class Secretaria: public Empleado {
6    // ...
7  };

9  class SecretariaTemporal final
10   : public Temporal, public Secretaria {
11   // ...
12 };

14 class Asesor final: public Temporal, public Administrador {
15   // ...
16 };
```

Herencia múltiple: Orden de inicialización

- Un objeto de una clase derivada mediante herencia múltiple se inicializa ejecutando los constructores en el siguiente orden:
  1. Constructores de las clases bases en el orden en que han sido declaradas en la lista de derivación.
  2. Constructores de los atributos en el orden en que aparecen declarados dentro de la clase (independientemente del orden de la lista de inicialización del constructor de la clase derivada).
  3. Por último se ejecuta el constructor de la clase derivada.
- Los destructores se ejecutan en orden inverso.

Herencia múltiple: Orden de inicialización

Ejemplo: Orden de constructores para definir un objeto `Asesor`

1. `Temporal()`

   `Empleado()`, por ser clase base de `Administrador`.

   `Administrador()`, que llamará a `Nivel()`.
2. Constructores de los atributos de `Asesor`.
3. `Asesor()`

Herencia múltiple: Ambigüedad al heredar miembros con nombres iguales

```
1  #include <iostream>

3  class B1 {
4  public:
5    void f() { std::cout << "B1::f()" << std::endl; }
6    int b;
7    // ...
8  };

10 class B2 {
11 public:
12   void f() { std::cout << "B2::f()" << std::endl; }
13   int b;
14   // ...
15 };

17 class D: public B1, public B2 {
18   // ...
19 };
```

Herencia múltiple: Ambigüedad al heredar miembros con nombres iguales

```
21 int main()
22 {
23   D d;
24   d.f();          // ERROR, ¿qué f(), el de B1 o el de B2?
25   d.b = 0;        // ERROR, ¿qué «b», el de B1 o el de B2?
26   d.B1::f();      // bien
27   d.B2::f();      // bien
28   d.B1::b = 0;    // bien
29   d.B2::b = 0;    // bien
30 }
```

Herencia múltiple: Ambigüedad al heredar miembros sobrecargados

```
1  #include <iostream>
2  using namespace std;

4  class B1 {
5  public:
6    void f(int i) { cout << "B1::f(int)" << endl; }
7    // ...
8  };

10 class B2 {
11 public:
12   void f(double d) { cout << "B2::f(double)" << endl; }
13   // ...
14 };

16 class D: public B1, public B2 {
17   // ...
18 };
```

Herencia múltiple: Ambigüedad al heredar miembros sobrecargados

```
20 int main()
21 {
22   D d;
23   d.f(0);          // ERROR, ¿qué f(), el de B1 o el de B2?
24   d.f(0.0);        // ERROR, ¿qué f(), el de B1 o el de B2?
25   d.B1::f(0);      // bien, B1::f()
26   d.B2::f(0.0);    // bien, B2::f()
27 }
```

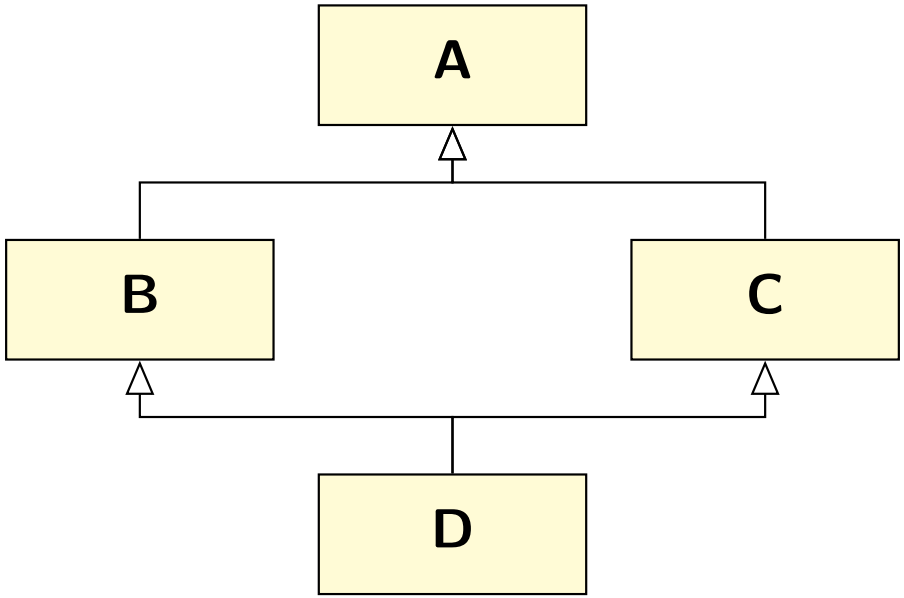
Herencia múltiple: Resolución de sobrecarga

```
1  #include <iostream>
2
3  class B1 {
4  public:
5    void f(char i) { std::cout << "B1::f(char)" << std::endl; }
6    // ...
7  };
8
9  class B2 {
10 public:
11   void f(int d) { std::cout << "B2::f(int)" << std::endl; }
12   // ...
13 };
14
15 class D: public B1, public B2 {
16 public:
17   using B1::f; using B2::f;
18   void f(double c) { std::cout << "D::f(double)" << std::endl; }
19   // ...
20 };
```

Herencia múltiple: Resolución de sobrecarga

```
22 int main()
23 {
24   D d;
25   d.f('A');    // B1::f(char)
26   d.f(0);      // B2::f(int)
27   d.f(0.0);    // D::f(double)
28 }
```

Herencia múltiple: Ambigüedad por herencia duplicada

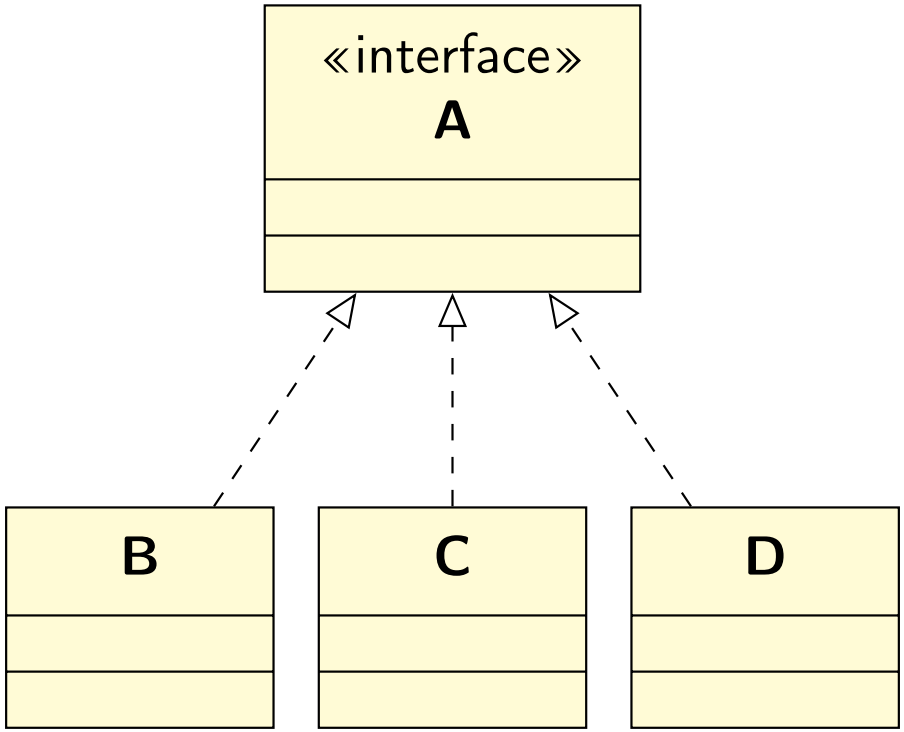



```
1 struct A { int a; };
2 struct B: A { int b; };
3 struct C: A { int c; };
4 struct D: B, C { int d; };

6 D d;
7 d.a = 0;       // ERROR, ambigüedad
8 d.B::a = 0;    // bien, resolución de la ambigüedad
9 d.C::a = 0;    // bien, resolución de la ambigüedad
```

Herencia virtual: Supresión de herencia duplicada

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

6 D d;
7 d.a = 0;        // bien, d sólo tiene un atributo a
```

Realización



- En C++, una interfaz se define mediante una clase abstracta sin atributos ni constructores, que solamente declara métodos polimórficos.
- Las implementaciones o realizaciones de una interfaz se definen como especializaciones de ella mediante herencia pública.