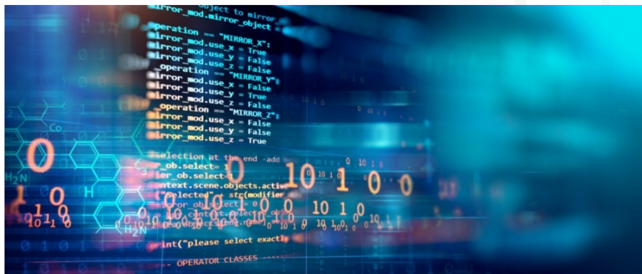
Herencia – Técnicas avanzadas

Tecnología de Programación



Adolfo Muñoz – Juan Magallón

Grado en Ingeniería Informática

Universidad Zaragoza

1542

Escuela de Ingeniería y Arquitectura
Universidad Zaragoza

1542

Departamento de Informática e Ingeniería de Sistemas
Universidad Zaragoza

2023-02-10

Clases Abstractas

(...en capítulos anteriores...)

Una clase pensada como raíz de una jerarquía de herencia implementa una serie de métodos del interfaz común que ofrece.

Las clases derivadas los redefinen para especializarlos.

```cpp
class Task {
public:
  virtual void run() const {};
};

void execute(const Task& task) {
  task.run();
}
```

```
class Task {
public:
  virtual void run() const {};
};

class Inc : public Task {
  int* i;
public:
  Inc(int* i) : i(i) {}
  void run() const override {
    ++(*i);
  }
};
```

```
void execute(const Task& task) {
  task.run();
}

int main() {
  int x = 0;
  Inc inc(&x);
  execute(inc);

  Task task;
  execute(task);
}
```

(*...en capítulos anteriores...*)

Una clase pensada como raíz de una jerarquía de herencia **implementa** una serie de métodos del interfaz común que ofrece...

(*...en capítulos anteriores...*)

Una clase pensada como raíz de una jerarquía de herencia **implementa** una serie de métodos del interfaz común que ofrece...

... o **quizás no**

Una **clase abstracta** tiene una parte que sólo especifica un cierto comportamiento a ofrecer pero **no lo implementa**.

Ese comportamiento se define mediante un tipo especial de métodos:

- C++: método virtual puro
- Java: método abstracto

que sólo se especifican, no se implementan.
(mismo concepto, distinta nomenclatura)

Una clase abstracta:

- Define e implementa una parte del comportamiento de la clase.
- Define pero no implementa otra parte.
- No se puede instanciar.
- Sirve como prototipo para definir otras clases mediante herencia.

```cpp
class Task {
public:
  virtual void run() const = 0;
};

void execute(const Task& task) {
  task.run();
}
```

```cpp
class Task {
public:
  virtual void run() const = 0;
};

class Inc {
  int* i;
public:
  Inc(int* i) : i(i) {}
  void run() const override {
    ++(*i);
  }
};
```

```cpp
void execute(const Task& task) {
   task.run();
}

int main() {
   int x = 0;
   Inc inc(&x);
   execute(inc);

   Task task; //Error!
   execute(task);
}
```

```cpp
#pragma once
#include <iostream>
using namespace std;

class Vehicle {
public:
  Vehicle() { }
  virtual void info() {
    cout << "I'm a vehicle" << endl;
  }
};
```

```cpp
#pragma once
#include "vehicle.h"

class MotorVehicle : public Vehicle
{
protected:
  int id;
public:
  MotorVehicle() : id(1234) { }
  virtual void info() {
    cout << "I'm a motor vehicle ";
    cout << "with plate " << id << endl;
  }
};
```

```cpp
#pragma once
#include "vehicle.h"

class Bike : public Vehicle {
public:
  Bike() { }

  virtual void info() {
    cout << "I'm a bike" << endl;
  }
};
```

```cpp
#pragma once
#include "motorvehicle.h"

class Car : public MotorVehicle {
protected:
  int seats;
public:
  Car() :seats(4) { }
  virtual void info() {
    cout << "I'm a car with plate ";
    cout << id <<" and ";
    cout << seats << " seats" << endl;
  }
};
```

```cpp
class Vehicle
{
public:
  Vehicle() { }
  virtual void info() = 0;
};
```

```cpp
class MotorVehicle: public Vehicle {
protected:
  int id;
public:
  MotorVehicle() : id(1234) { }
};
```

```cpp
class Bike: public Vehicle {
public:
  Bike() { }
  void info() {
    cout << "I'm a bike" << endl;
  }
};
```

```cpp
class Car: public MotorVehicle {
  int seats;
public:
  Car() : seats(4) { }
  void info() {
    cout<<"I'm a car with plate " << id;
    cout<<" and "<<seats<<" seats"<< endl;
  }
};
```

```java
class Vehicle {
  Vehicle() { }
  void info() {
    System.out.println("I'm a vehicle");
  }
}
```

```java
class Bike extends Vehicle {
  Bike() { }
  void info() {
    System.out.println("I'm a bike");
  }
}
```

```java
class MotorVehicle extends Vehicle {
  int id = 1234;
  MotorVehicle() { }
  void info() {
    System.out.println(
      "I'm a motor vehicle "
      + "with plate " + id);
  }
}
```

```java
class Car extends MotorVehicle {
  int seats = 4;
  Car() { }
  void info() {
    System.out.println(
    "I'm a car "
    + "with plate " + id
    + " and " + seats + " seats");
    }
}
```

```java
abstract class Vehicle {
  Vehicle() { }
  abstract void info();
}
```

```java
abstract class MotorVehicle extends Vehicle {
  int id = 1234;
  MotorVehicle() { }
}
```

```java
class Bike extends Vehicle {
  Bike() { }
  void info() {
    System.out.println("I'm a bike");
  }
}
```

```java
class Car extends MotorVehicle {
  int seats = 4;
  Car() { }
  void info() {
    System.out.println(
    "I'm a car "
    + "with plate " + id
    + " and " + seats + " seats");
  }
}
```

Una clase puede llegar a ser tan abstracta que puede no contener implementación alguna.

```java
abstract class Sprite
{
  String  name;

  abstract void draw( );
  abstract float area( );
  abstract float perimeter( );
}
```

```java
class Circle extends Sprite
{
  int   x,y;
  int   radius;

  void  draw( ) { }
  float area( ) { }
  float perimeter( ) { }
}
```

En realidad sólo especifica un **interfaz público** a implementar.

```
class Sprite
{
  String  name;
}
```

```
interface Shape
{
  void  draw( );
  float area( );
  float perimeter( );
}
```

```
class Circle
      extends Sprite
      implements Shape
{
  int x,y;
  int radius;

  public void  draw( ) { }
  public float area( ) { }
  public float perimeter( ) { }
}
```

Una clase abstracta obliga a sus descendientes a implementar un interfaz.

```
1  interface Shape
2  {
3    void  draw( );
4    float area( );
5    float perimeter( );
6  }
7  abstract class Sprite
8    implements Shape
9  {
10   String  name;
11 }
```

```
1  class Circle extends Sprite
2  {
3    int   x,y;
4    int   radius;
5
6    public void  draw( ) { }
7    public float area( ) { }
8    public float perimeter( ) { }
9  }
```

Los interfaces obligan a ser implementados por la clase.

```java
interface Talkative {
  void talk();
}

abstract class Animal
  implements Talkative {
}
```

```java
class Dog extends Animal {
  public void talk() {
    System.out.println("Woof!");
  }
}
class Cat extends Animal {
  public void talk() {
    System.out.println("Meow!");
  }
}
```

```java
class Interrogator {
  static void chat(Animal subject) {
    subject.talk();
  }
}
```

Los interfaces funcionan como tipos de datos.

```java
interface Talkative {
  void talk();
}

abstract class Animal
  implements Talkative {
}
```

```java
class Dog extends Animal {
  public void talk() {
    System.out.println("Woof!");
  }
}
class Phone implements Talkative {
  public void talk() {
    System.out.println("Ring!");
  }
}
```

```java
class Interrogator {
  static void chat(Talkative subject) {
    subject.talk();
  }
}
```

Herencia múltiple

Remember *Jurassic Park*...

```
1  abstract class Animal {
2    abstract void talk();
3  }
```

```
1  class Frog extends Animal {
2    void talk() {
3      System.out.println("Croak!");
4    }
5  }
6
7  class Dinosaur extends Animal {
8    void talk() {
9      System.out.println("Roarr!");
10   }
11 }
```

¿Podemos combinarlos genéticamente mediante herencia?

Herencia múltiple

Remember *Jurassic Park*...

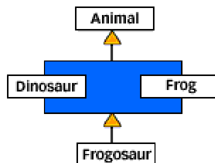
```java
abstract class Animal {
  abstract void talk();
}
```

```java
class Frog extends Animal {
  void talk() {
    System.out.println("Croak!");
  }
}

class Dinosaur extends Animal {
  void talk() {
    System.out.println("Roarr!");
  }
}
```

¿Podemos combinarlos genéticamente mediante herencia?

```java
// Esto es _ilegal_ en Java
class Frogosaur extends Frog, Dinosaur { }
```

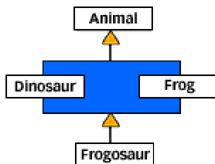
Problema del rombo (*diamond problem*, in english...):




```
1  Animal f = new Frogosaur();
2  f.talk();
```

¿ Croa o ruge ?

Problema del rombo (*diamond problem*, in english...):



- **Solución Java**: se puede heredar el interfaz, no la implementación.
- **Solución C++**: se puede elegir heredar múltiples clases, siempre y cuando se resuelvan las ambigüedades.

Herencia múltiple (C++)

```cpp
class Animal {
  string name;
  Animal(const string& n) : name(n) {};
  virtual void talk() const = 0;
};

class Frog : public Animal {
  Frog() : Animal("frog") {};
  void talk() const override
  { cout << "croak" << endl; }
};

class Dinosaur : public Animal {
  Dinosaur() : Animal("dino") {};
  void talk() const override
  { cout << "roarr" << endl; }
};
```

```cpp
#include "animal.h"

class Frogosaur :
    public Frog, public Dinosaur
{
};
```

```
1  #include "frogosaur.h"
2
3  int main() {
4    Frog      f;
5    Dinosaur  d;
6    Frogosaur g;
7
8    cout << f.name <<" -> "; f.talk();
9    cout << d.name <<" -> "; d.talk();
10   // No es posible. ¿ Por qué ?
11   //cout << g.name <<" -> "; g.talk();
12
13   cout << g.Frog::name << " -> ";
14     g.Frog::talk();
15   cout << g.Dinosaur::name << " -> ";
16     g.Dinosaur::talk();
17
18   Frog& fg(g);
19   cout << fg.name << " -> "; fg.talk();
20   Dinosaur& dg(g);
21   cout << dg.name << " -> "; dg.talk();
22
23   return 0;
24 }
```

```
1  > ./main
2  frog -> croak
3  dino -> roarr
4  frog -> croak
5  dino -> roarr
6  frog -> croak
7  dino -> roarr
```

Pregunta

¿ Cuántos Animal-es hay en un Frogosaur-io ?

Herencia múltiple virtual (C++)

```cpp
class Animal {
  string name;
  Animal(const string& n) : name(n) {};
  virtual void talk() const = 0;
};

class Frog : public virtual Animal {
  Frog() : Animal("frog") {};
  void talk() const override
  { cout << "croak" << endl; }
};

class Dinosaur : public virtual Animal {
  Dinosaur() : Animal("dino") {};
  void talk() const override
  { cout << "roarr" << endl; }
};
```

```cpp
#include "animal.h"

class Frogosaur :
    public Frog, public Dinosaur
{
  Frogosaur() : Animal("frogosaur") {};

  void talk() const override
  { cout << "croarr" << endl; };
};
```

```cpp
#include "frogosaur.h"

int main() {
  Frog      f;
  Dinosaur  d;
  Frogosaur g;

  cout << f.name <<" -> "; f.talk();
  cout << d.name <<" -> "; d.talk();
  // Ahora sí.
  cout << g.name <<" -> "; g.talk();

  cout << g.Frog::name << " -> ";
    g.Frog::talk();
  cout << g.Dinosaur::name << " -> ";
    g.Dinosaur::talk();

  Frog& fg(g);
  cout << fg.name << " -> "; fg.talk();
  Dinosaur& dg(g);
  cout << dg.name << " -> "; dg.talk();

  return 0;
}
```

```
> ./main
frog -> croak
dino -> roarr
frogosaur -> croarr
frogosaur -> croak
frogosaur -> roarr
frogosaur -> croarr
frogosaur -> croarr
```

Constructores y Destructores

En una jerarquía de herencia, cada una de las clases puede necesitar sus propios constructores y destructores.

¿ Cómo se gestionan en el proceso de herencia ?

En una jerarquía de herencia, cada una de las clases puede necesitar sus propios constructores y destructores.

¿ Cómo se gestionan en el proceso de herencia ?

Cada clase es responsable de sus propios datos (memoria, ficheros, etc).

Ejemplo

Escritura de datos en un fichero, con dos implementaciones:

- clase `Writer`: gestión común del fichero (abrir,cerrar), método abstracto de escritura.
- clase `WriterDirect`: escritura directa.
- clase `WriterCached`: escritura a través de una *cache*.

Writer

```cpp
class Writer
{
protected:
  ofstream os;

public:
  Writer();
  ~Writer();

  virtual void put(int data) = 0;
};
```

```cpp
Writer::Writer()
{
  cout << __PRETTY_FUNCTION__ << endl;
  os.open("/dev/null",ofstream::binary);
}

Writer::~Writer()
{
  cout << __PRETTY_FUNCTION__ << endl;
  os.close();
}
```

WriterDirect

```
class WriterDirect : public Writer
{
public:
  WriterDirect();
  ~WriterDirect();

  void put(int data) override;
};
```

```
WriterDirect::WriterDirect()
{
  cout << __PRETTY_FUNCTION__ << endl;
}

WriterDirect::~WriterDirect()
{
  cout << __PRETTY_FUNCTION__ << endl;
}

void WriterDirect::put(int data)
{
  cout << __PRETTY_FUNCTION__ << endl;
  os.write((char*)&data,sizeof(int));
}
```

WriterCached

```
1   class WriterCached : public Writer
2   {
3   private:
4     int*  cache;
5     int   size;
6     int   ndat;
7
8   public:
9     WriterCached(int sz);
10    ~WriterCached();
11
12    void put(int data) override;
13  };
```

```
1   WriterCached::WriterCached(int sz) : size(sz), ndat(0)
2   {
3     cout << __PRETTY_FUNCTION__ << endl;
4     cache = new int[size];
5   }
6
7   WriterCached::~WriterCached()
8   {
9     cout << __PRETTY_FUNCTION__ << endl;
10    if (ndat>0)
11      os.write((char*)cache,ndat*sizeof(int));
12    delete[] cache;
13  }
14
15  void WriterCached::put(int data)
16  {
17    cout << __PRETTY_FUNCTION__ << endl;
18    cache[ndat++] = data;
19    if (ndat==size) {
20      os.write((char*)cache,ndat*sizeof(int));
21      ndat = 0;
22    }
23  }
```

Constructores y Destructores

```cpp
int main()
{
  WriterDirect w;

  w.put(33);

  return 0;
}
```

```
> ./main
Writer::Writer()
WriterDirect::WriterDirect()
virtual void WriterDirect::put(int)
WriterDirect::~WriterDirect()
Writer::~Writer()
```

Funcionamiento correcto:

- Se ejecutan los constructores, desde la clase base hacia la derivada.
- Se ejecutan los destructores, desde la clase derivada hacia la clase base.

```cpp
int main()
{
  WriterCached w(128);

  w.put(33);

  return 0;
}
```

```
> ./main
Writer::Writer()
WriterCached::WriterCached(int)
virtual void WriterCached::put(int)
WriterCached::~WriterCached()
Writer::~Writer()
```

Funcionamiento correcto:

- Se ejecutan los constructores, desde la clase base hacia la derivada.
- Se ejecutan los destructores, desde la clase derivada hacia la clase base.

```cpp
int main()
{
  WriterDirect* w = new WriterDirect;

  w->put(33);

  delete w;

  return 0;
}
```

```
> ./main
Writer::Writer()
WriterDirect::WriterDirect()
virtual void WriterDirect::put(int)
WriterDirect::~WriterDirect()
Writer::~Writer()
```

Funcionamiento correcto:

- Se ejecutan los constructores, desde la clase base hacia la derivada.
- Se ejecutan los destructores, desde la clase derivada hacia la clase base.

```cpp
int main()
{
  WriterCached* w = new WriterCached(128);

  w->put(33);

  delete w;

  return 0;
}
```

```
> ./main
Writer::Writer()
WriterCached::WriterCached(int)
virtual void WriterCached::put(int)
WriterCached::~WriterCached()
Writer::~Writer()
```

Funcionamiento correcto:

- Se ejecutan los constructores, desde la clase base hacia la derivada.
- Se ejecutan los destructores, desde la clase derivada hacia la clase base.

```cpp
int main()
{
  Writer* w = new WriterDirect;

  w->put(33);

  delete w;

  return 0;
}
```

```
> ./main
Writer::Writer()
WriterDirect::WriterDirect()
virtual void WriterDirect::put(int)
Writer::~Writer()
```

OOPS !!!

- No demasiado grave, el destructor no hace nada...

Pero, ¿ Por qué ?

```
1  int main()
2  {
3    Writer* w = new WriterCached(128);
4
5    w->put(33);
6
7    delete w;
8
9    return 0;
10 }
```

```
1  > ./main
2  Writer::Writer()
3  WriterCached::WriterCached(int)
4  virtual void WriterCached::put(int)
5  Writer::~Writer()
```

OOPS !!!

- Problema serio: no se vuelca la *cache*
- Problema serio: no se libera la memoria

¿ Por qué ?

```
1   Writer* w = new WriterCached(128);
2   ...
3   delete w;
```

El destructor se selecciona según el tipo declarado de 'w' no según el objeto real al que está apuntando:
asociación **estática**.

¿ Qué os recuerda ?

```
1   Writer* w = new WriterCached(128);
2   ...
3   delete w;
```

El destructor se selecciona según el tipo declarado de 'w' no según el objeto real al que está apuntando:

asociación **estática**.

¿ Qué os recuerda ?

El destructor debe ser **virtual**.

```
class Writer
{
protected:
  ofstream os;

public:
  Writer();
  virtual ~Writer();

  virtual void put(int data) = 0;
};
```

```
> ./main
Writer::Writer()
WriterCached::WriterCached(int)
virtual void WriterCached::put(int)
virtual WriterCached::~WriterCached()
virtual Writer::~Writer()
```

Funcionamiento correcto, con una peculiaridad. ¿ Cuál ?

```cpp
class Writer
{
protected:
  ofstream os;

public:
  Writer();
  virtual ~Writer();

  virtual void put(int data) = 0;
};
```

```
> ./main
Writer::Writer()
WriterCached::WriterCached(int)
virtual void WriterCached::put(int)
virtual WriterCached::~WriterCached()
virtual Writer::~Writer()
```

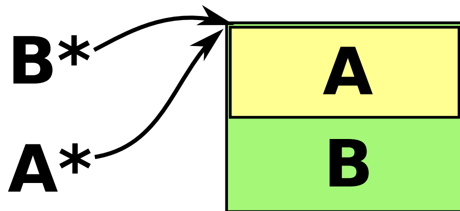
Funcionamiento correcto, con una peculiaridad. ¿ Cuál ?

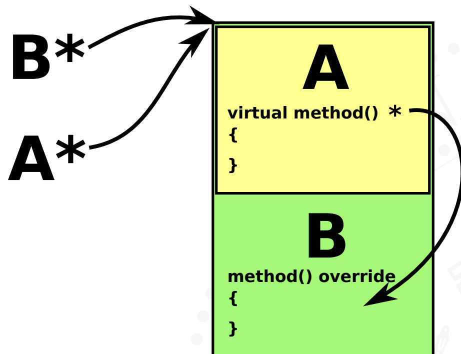
Un destructor virtual no reemplaza completamente al destructor base, sólo redefine el *punto de entrada* en el proceso de destrucción.

Constructores y Destructores

Recomendación: Una clase diseñada como clase base de una jerarquía de herencia **siempre debería tener un destructor virtual**, aunque sea vacío.

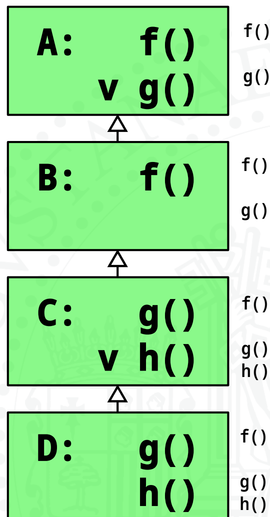
Implementación


B*
A*
A
B


B*
A*
A
virtual method() *
{
}
B
method() override
{
}

 Implementación

```
1  class A {
2              void f();
3    virtual void g();
4  };
5  class B : public A {
6              void f();
7  };
8  class C : public B {
9              void g();
10   virtual void h();
11 };
12 class D : public C {
13             void g();
14             void h();
15 };
```



 Implementación

La llamada a los métodos virtuales debe funcionar bajo múltiples condiciones:

- Debe ser eficiente, no implicar ningún tipo de búsqueda, sólo quizás con una mínima sobrecarga.
- Se debe poder realizar sin conocer el tipo real del objeto.
- Debe funcionar incluso con clases todavía no existentes, mientras se adhieran a las condiciones de la herencia.
- Debe soportar la herencia múltiple, y las especificaciones de ámbito.

La llamada a los métodos virtuales se realiza a través de una

tabla de métodos virtuales

Cada método virtual (y sólo los virtuales) se asocia a una entrada de esa tabla.

La tabla es única para cada clase.

Todos los objetos almacenan un puntero a su tabla de métodos virtuales.

Pregunta:

¿ Dónde ?

Todos los objetos almacenan un puntero a su tabla de métodos virtuales.

Pregunta:
¿ Dónde ?

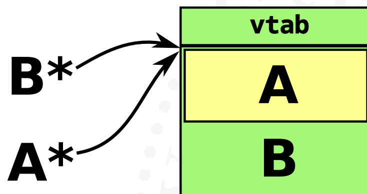
Posibilidades

- como primer atributo
- como último atributo
- ????

Todos los objetos almacenan un puntero a su tabla de métodos virtuales.

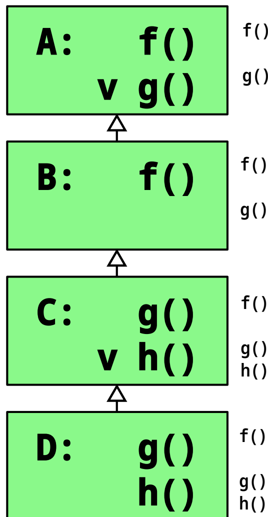
Respuesta:

Antes del objeto en sí mismo



Esto tiene implicaciones en la gestión de la memoria dinámica (`new` y `delete`).

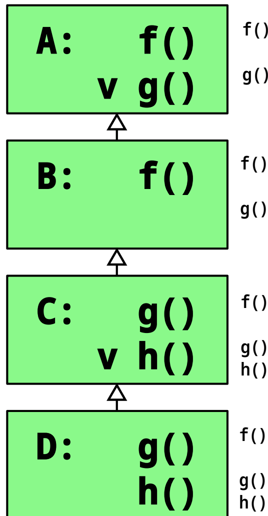
 Implementación



A::vtab

0	A_g()

```
A* a = new A;
a->f() => A_f(a)
a->g() => a->vtab[0](a) => A_g(a)
```



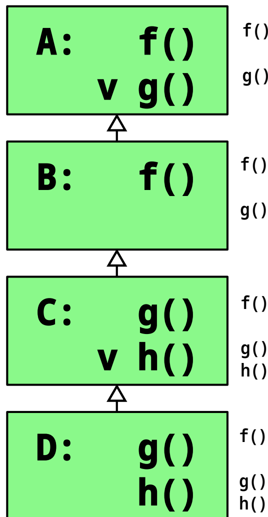
B::vtab

0	A_g()

```
B* b = new B;
b->f() => B_f(b)
b->g() => b->vtab[0](b) => A_g(b)
```

```
A* a = new B;
a->f() => A_f(a)
a->g() => a->vtab[0](a) => A_g(a)
```

Implementación

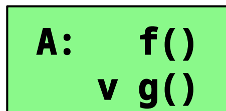


C::vtab

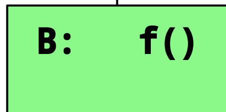
0	C_g()
1	C_h()

```
C* c = new C;
c->f() => B_f(c)
c->g() => c->vtab[0](c) => C_g(c)
c->h() => c->vtab[1](c) => C_h(c)

A* a = new C;
a->f() => A_f(a)
a->g() => a->vtab[0](a) => C_g(a)
a->h() => ???
```

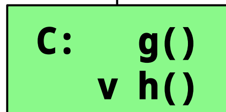
 Implementación



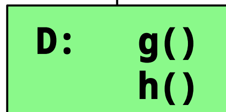
f()
g()



f()
g()



f()
g()
h()



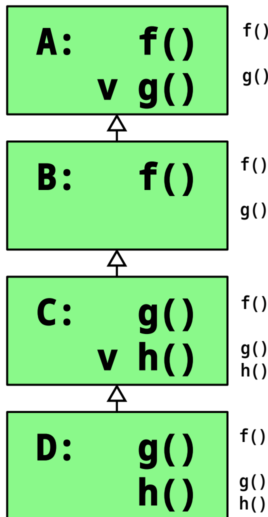
f()
g()
h()

D::vtab

0	D_g()
1	D_h()

```
D* d = new D;
d->f() => B_f(d)
d->g() => d->vtab[0](d) => D_g(d)
d->h() => d->vtab[1](d) => D_h(d)

A* a = new D;
a->f() => A_f(a)
a->g() => a->vtab[0](a) => D_g(a)
a->h() => ???
```

Implementación

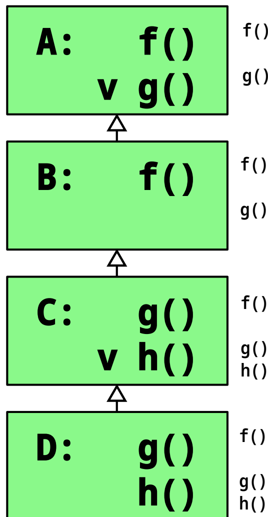


D::vtab

0	D_g()
1	D_h()

```
D* d = new D;
d->f() => B_f(d)
d->g() => d->vtab[0](d) => D_g(d)
d->h() => d->vtab[1](d) => D_h(d)

C* c = new D;
c->f() => B_f(c)
c->g() => c->vtab[0](c) => D_g(c)
c->h() => c->vtab[1](c) => D_h(c)
```

Implementación

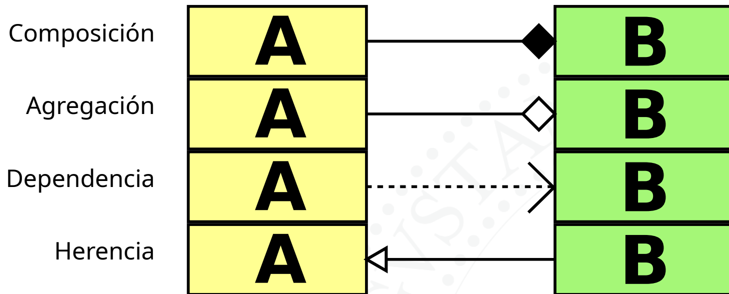


D::vtab

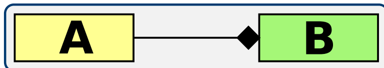
0	D_g()
1	D_h()

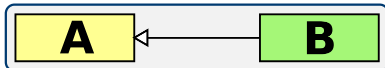
```
D* d = new D;
d->f() => B_f(d)
d->g() => d->vtab[0](d) => D_g(d)
d->h() => d->vtab[1](d) => D_h(d)

C* c = new D;
c->f() => B_f(c)
c->B::g() => c->B::vtab[0](c) => A_g(c)
c->C::h() => c->C::vtab[1](c) => C_h(c)
```

Relaciones



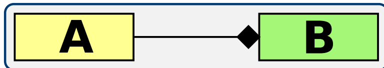
Se pueden combinar

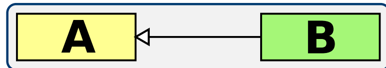





```cpp
struct Node {
  virtual void show( ) const = 0;
  virtual ~Node( ){}
};
class Leaf : public Node {
 int element;
public:
 void show( ) const override {
   std::cout<<element<<" ";
 }
};
```

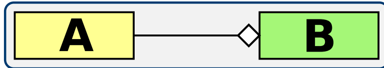
```cpp
class Branch : public Node {
  Node* left;
  Node* right;
public:
 void show( ) const override {
   left->show( ); right->show( );
 }
 ~Branch( ){
   delete left;
   delete right;
 }
};
```

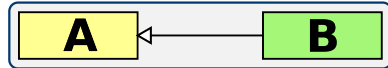





```
1  struct Node {
2    virtual void show( ) const = 0;
3    virtual ~Node( ){}
4  };
```

```
1  class Tree {
2    Node* root;
3  public:
4    //Constructor should fill tree
5   void show( ) {
6      node->show( );
7   }
8   ~Tree( ){
9      delete root;
10  }
11 };
```






```cpp
class Task {
public:
  virtual void run() const = 0;
};
```

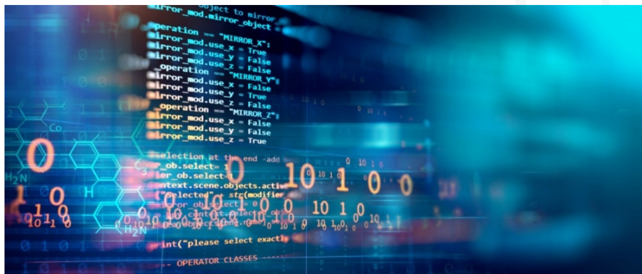
```cpp
class Seq : public Task {
  Task* tasks[]; int n;
public:
  void run() const override
  {
    for(int i=0; i<n; ++i)
      tasks[i]->run();
  }
};
```

La **explosión combinatoria** de relaciones entre clases permite resolver problemas de formas muy diversas. La dificultad radica en encontrar aquella combinación de mecanismos que maximice la escalabilidad y reutilización de código y minimice el riesgo de error.

Esta explosión combinatoria crece con cada concepto nuevo de Programación Orientada a Objetos.
El próximo: la **programación genérica.**

Herencia – Técnicas avanzadas

Tecnología de Programación



Adolfo Muñoz – Juan Magallón

Grado en Ingeniería Informática

Universidad Zaragoza
1542

Escuela de Ingeniería y Arquitectura
Universidad Zaragoza

1542

Departamento de Informática e Ingeniería de Sistemas
Universidad Zaragoza

2023-02-10