TEMA 5

Herencia Múltiple

Facultad de Informática

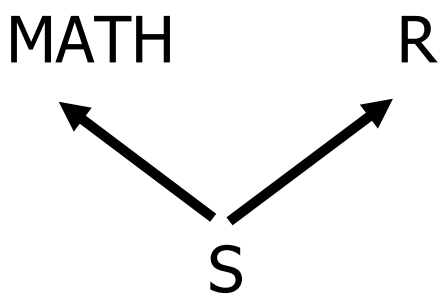
Universidad de Murcia

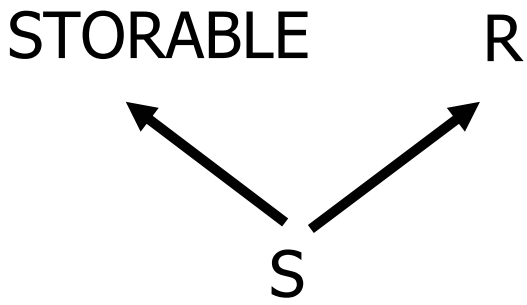
Índice

1.- Introducción

- Las clases pueden necesitar mas de una clase padre
- Más difícil de usar e implementar que la herencia simple.
- Algunos consideran que produce más inconvenientes que ventajas. Según B. Meyer:
  - *"No puedo pensar en una aplicación seria escrita en Eiffel que no use la herencia múltiple en un modo significativo"*
  - *"Para discutir cuestiones conceptuales sobre H.M. es necesario considerar el doble aspecto de la herencia: subtipos y extensión modular"*

Herencia múltiple y Extensión modular





- S es una especialización de R
- MATH incluye un paquete de rutinas matemáticas
- STORABLE ofrece servicios de persistencia de objetos

Herencia Múltiple y Subtipos

Ejemplo1: Implementación de los menús de una aplicación *(1/2)*

i) Los menús son **simples**

ii) Los menús contienen **submenús (Menú compuesto)**

1) Cambiar el tipo de la colección

MENU ⟹ **TREE [OPCION_MENU]**

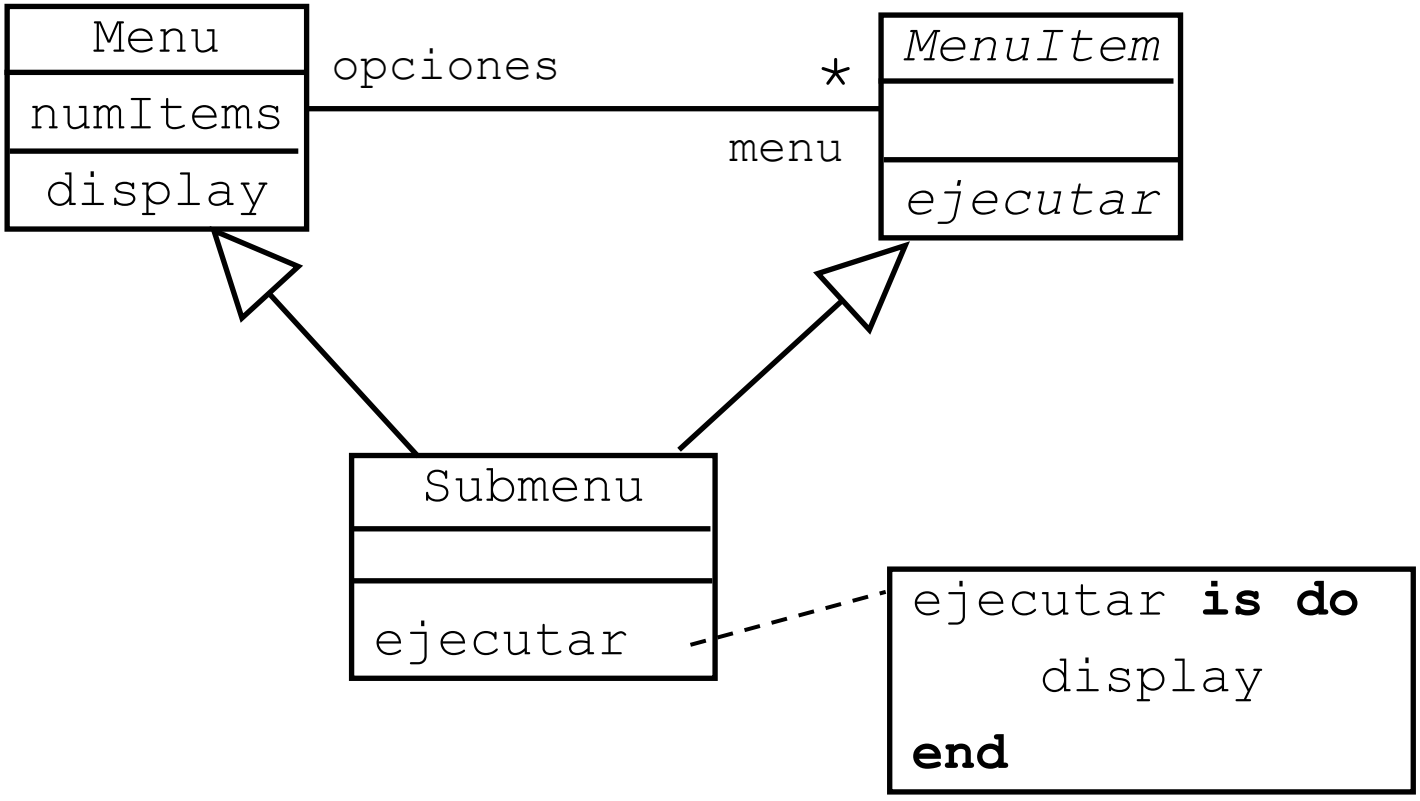
"provocaría muchos cambios"

2) Aprovechar la implementación de i) con herencia múltiple

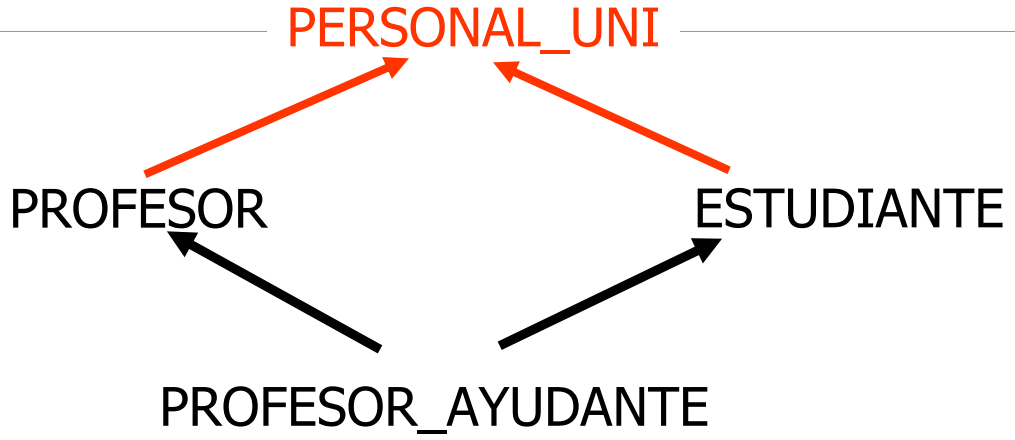
Herencia Múltiple y Subtipos

Ejemplo1: Implementación de los menús de una aplicación *(2/2)*

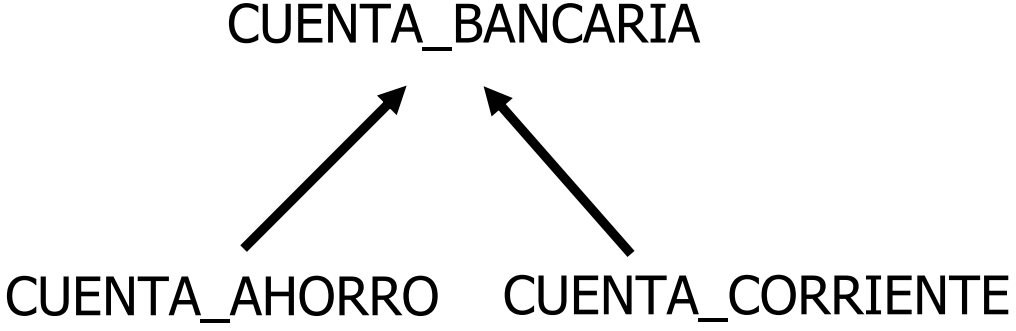
ii) Los menús contienen **submenús (Menú compuesto)**



Ejemplo2: Profesores ayudantes están matriculados de los cursos de doctorado

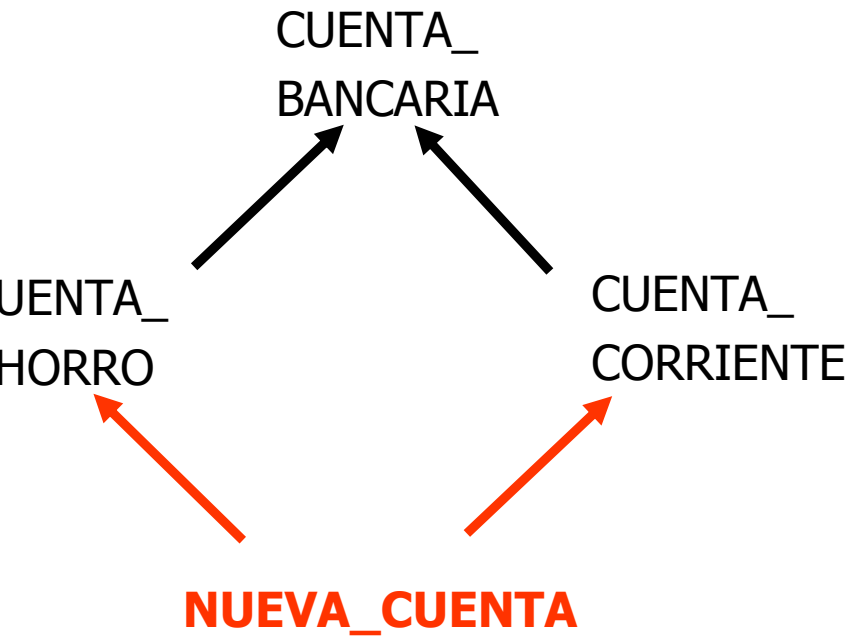


Ejemplo3: Añadir un nuevo tipo de cuenta que comparte a la vez propiedades de cuenta de ahorro y cuenta corriente

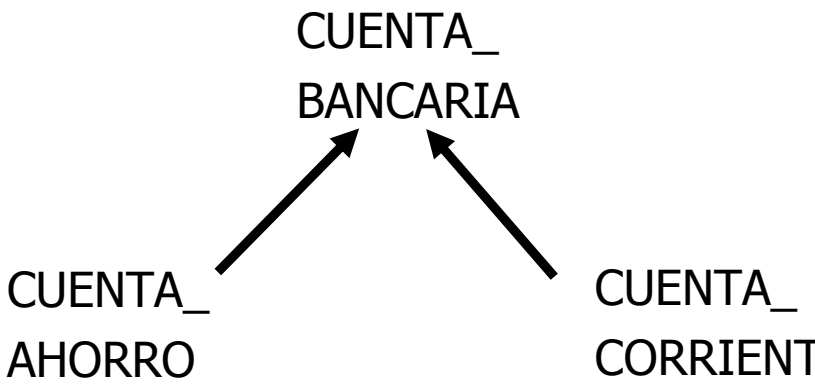


SOLUCIÓN:

Con herencia múltiple

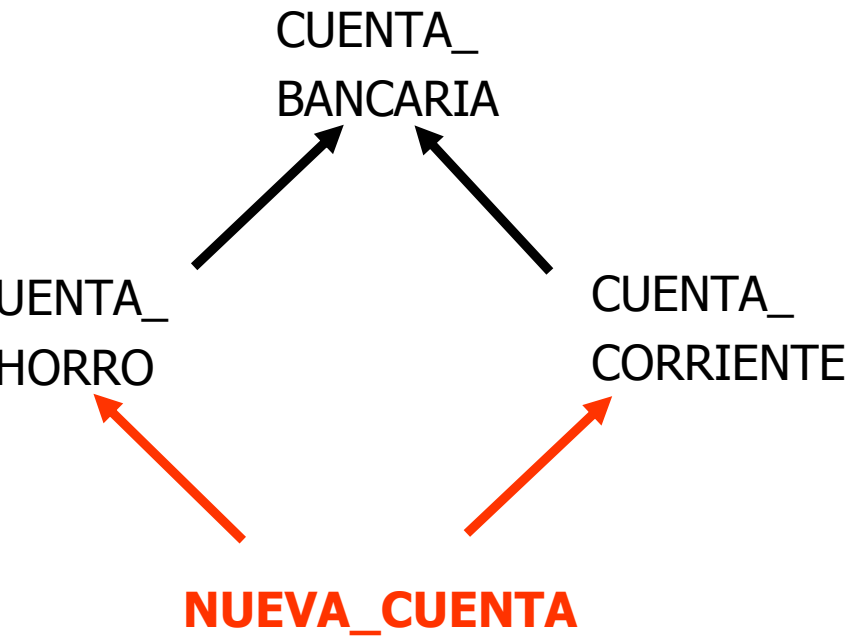


Sin herencia múltiple

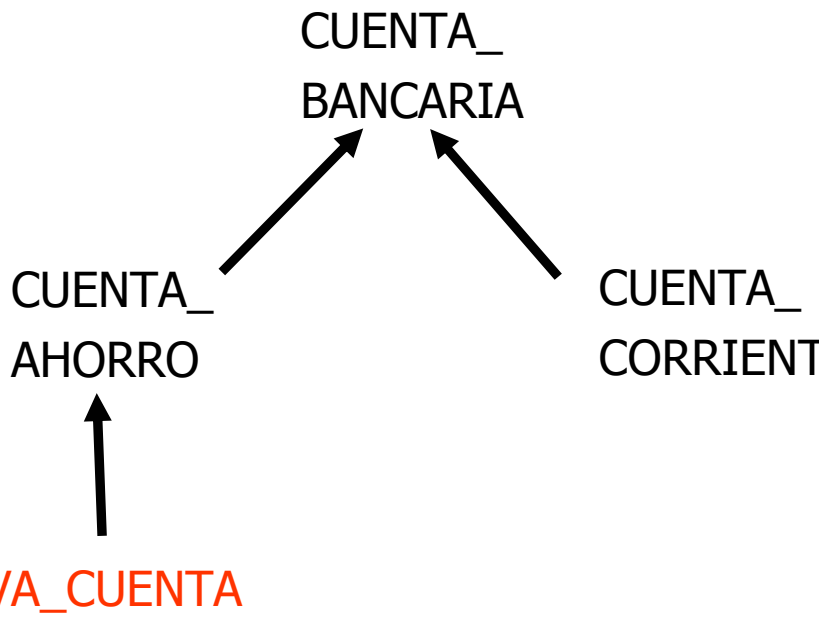


SOLUCIÓN:

Con herencia múltiple



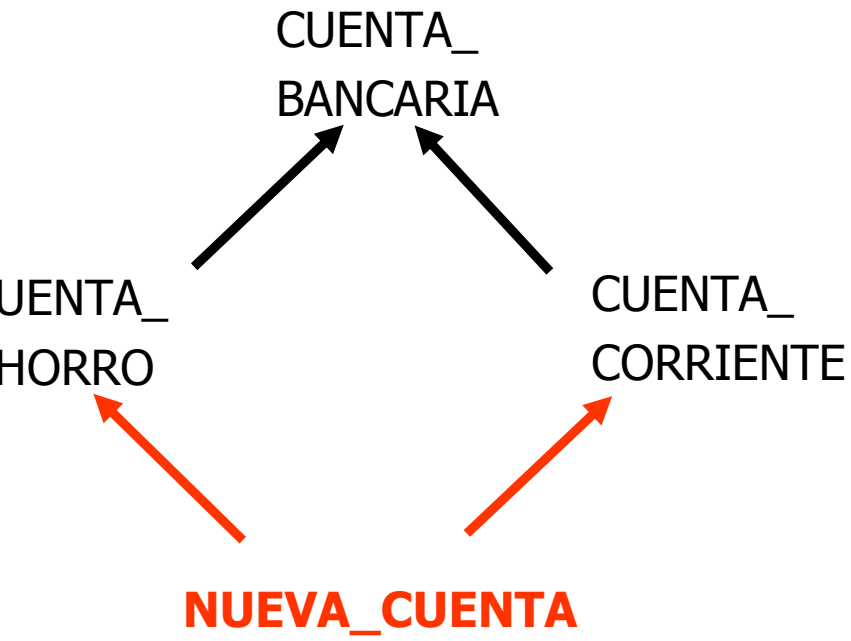
Sin herencia múltiple



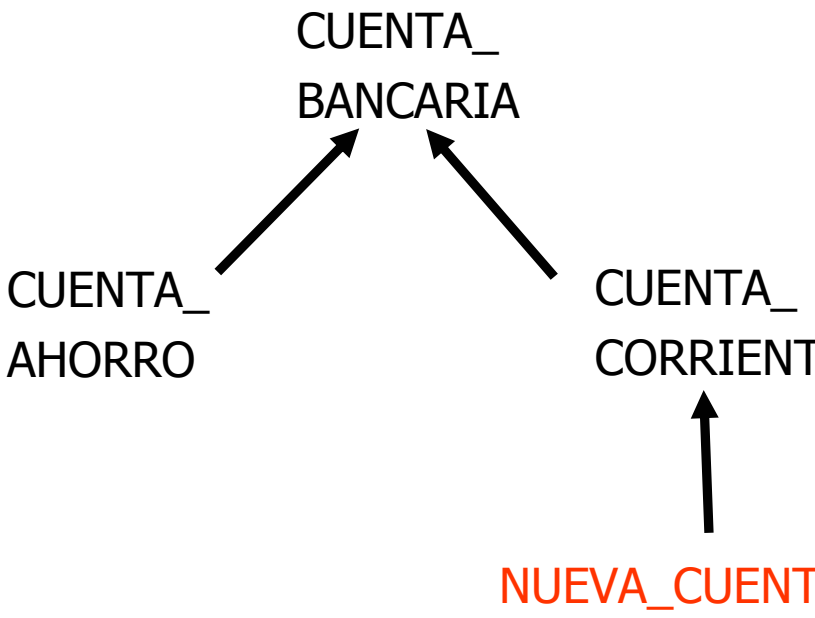
Debemos añadir a **NUEVA_CUENTA**
las propiedades de
CUENTA_CORRIENTE

SOLUCIÓN:

Con herencia múltiple



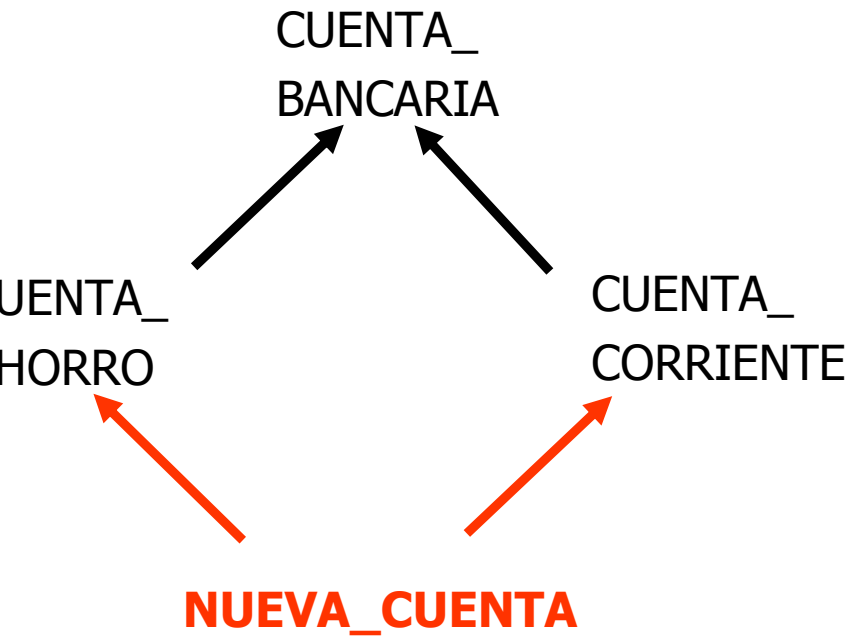
Sin herencia múltiple



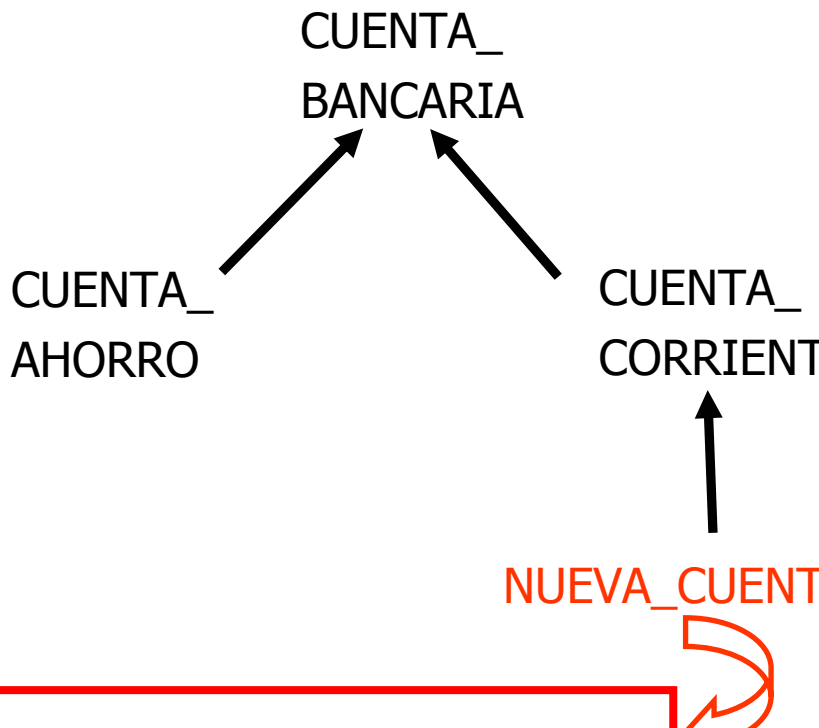
Debemos añadir a **NUEVA_CUENTA**
las propiedades de
CUENTA_AHORRO

SOLUCIÓN:

Con herencia múltiple



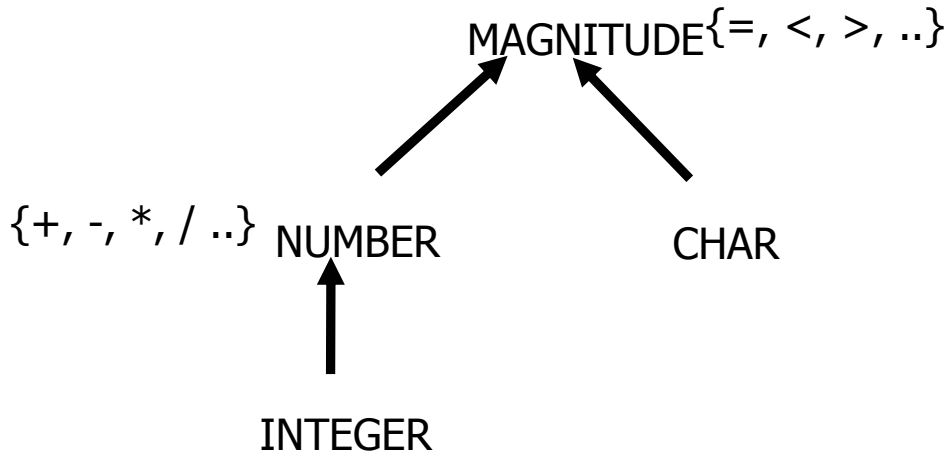
Sin herencia múltiple



Perdemos:

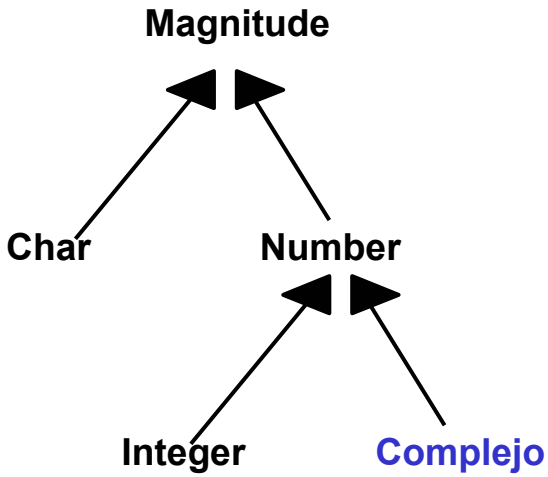
- Polimorfismo
- Principio de Abierto-Cerrado
- Beneficios de la reutilización de la herencia

Ejemplo4: *"Añadir a la jerarquía Smalltalk una clase para representar números complejos"*



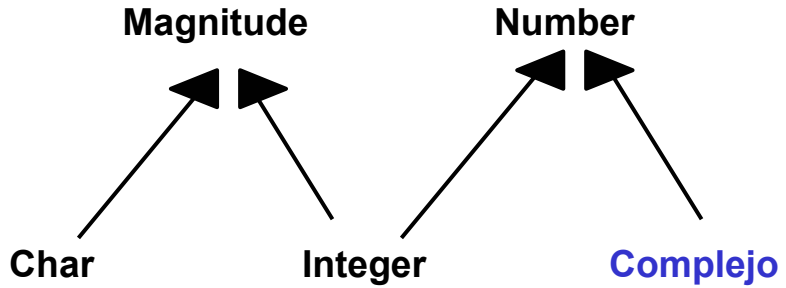
Soluciones:

OPCION A



{=, >, <, ERROR}

OPCION B



2.- Utilidades de la herencia múltiple

A) Combinar abstracciones de tipos (padres simétricos)

B) Matrimonio de conveniencia

- Herencia de implementación

C) Herencia de estructura

D) Herencia de facilidades

A) Combinación de interfaces no relacionadas

Ejemplo1:

```
class Ventana inherit
    TREE [Ventana]
    Rectangulo
feature
    ....
end
```

"Una ventana es un objeto gráfico y un árbol de ventanas"

```
class VentanaTexto inherit
    WINDOW
    STRING
feature
    ....
end
```

"Una ventana de texto es una ventana que manipula texto"

A) Combinación de interfaces no relacionadas

Ejemplo2:

```
class TREE [G] inherit
    LIST [G]
    LINKABLE [G]
feature
    ....
end
```

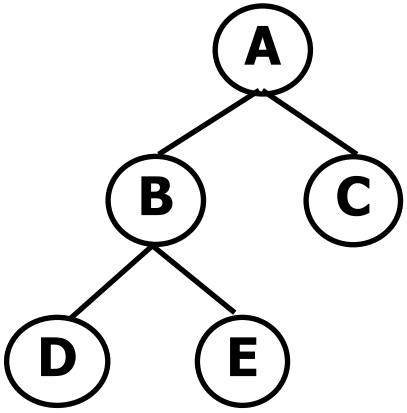
IST permite obtener los hijos de un odo, añadir/eliminar un hijo,...

INKABLE permite obtener el valor e un nodo, su hermanos, su padre, ñadir un hermano...

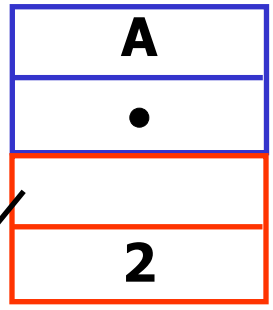
"Un árbol es una lista, la lista de sus hijos, pero también es un elemento potencial de la lista (un subárbol de otro árbol)

"Un árbol es una lista que es también un elemento de la lista"

TREE[G] gráficamente

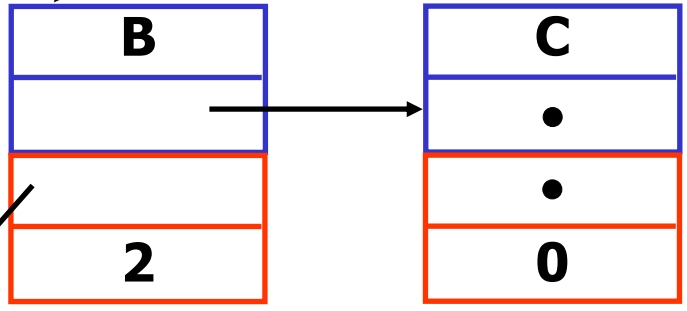


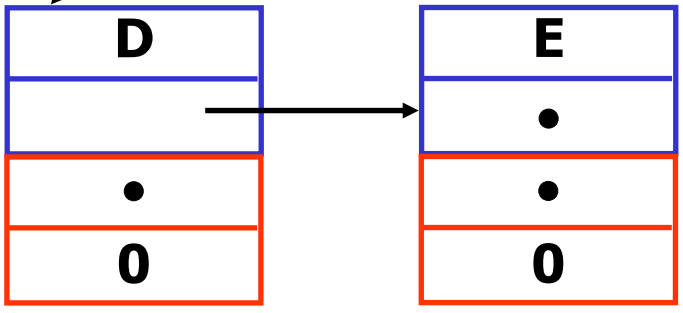
ARBOL es_un NODO



valor
siguiente
cabeza
numElementos

ARBOL es_una LISTA



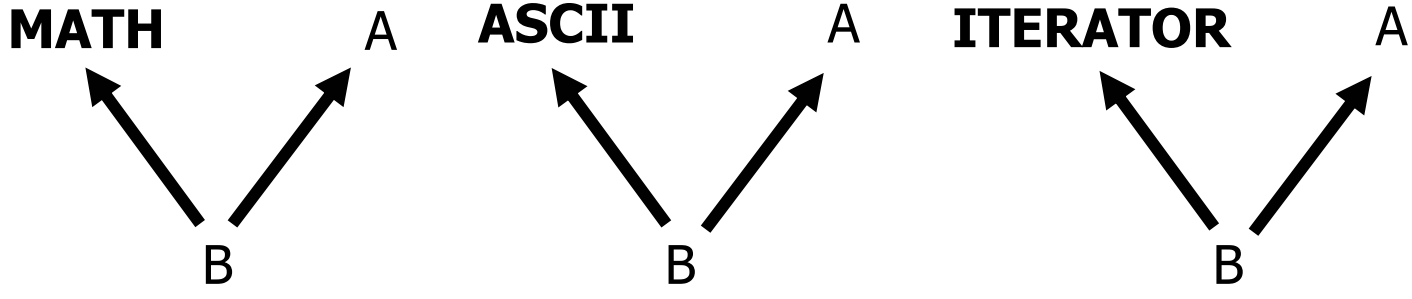


C) Herencia de Estructura:

Se desea que una clase posea ciertas propiedades además de la abstracción que representa.

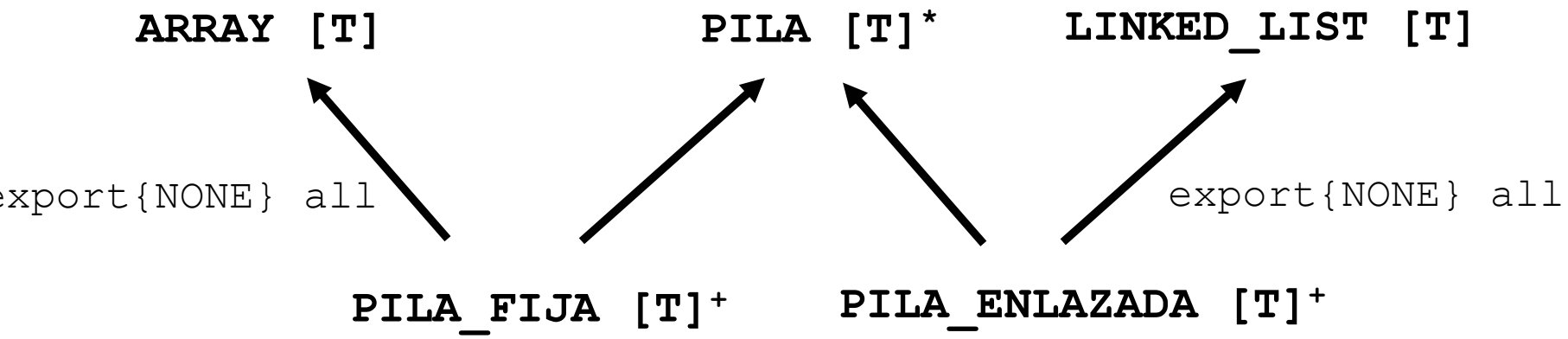
D) Herencia de Facilidades:

Existen clases que existen con el único propósito de ofrecer unos servicios a sus descendientes



3) Matrimonio por conveniencia

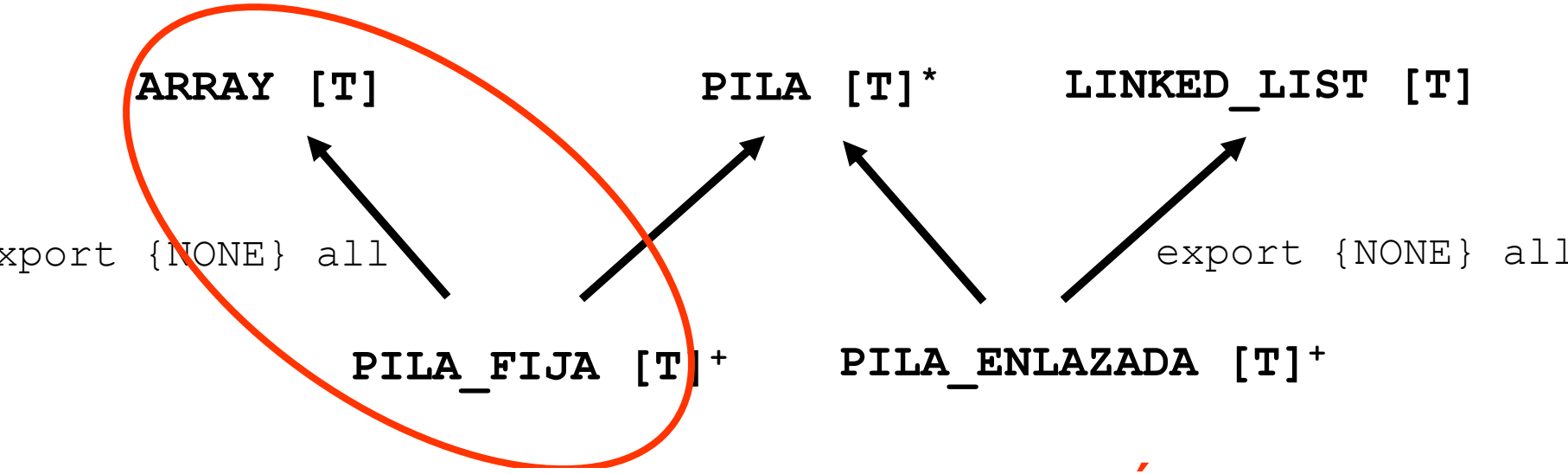
Proporcionar una implementación para una abstracción definida por una clase diferida usando las facilidades que proporciona una clase efectiva



- La clase `PILA_FIJA[T]` sólo exporta las características exportadas por `PILA[T]` y oculta las propiedades de `ARRAY[T]`.
- La clase `PILA_ENLAZADA[T]` sólo exporta las características exportadas por `PILA [T]` y oculta las propiedades de `LINKED_LIST[T]`.

3) Matrimonio por conveniencia

Proporcionar una implementación para una abstracción definida por una clase diferida usando las facilidades que proporciona una clase efectiva



Implementación de Pilas usando arrays 1/2

```
class PILA_FIJA [T] inherit
   PILA [T]
   ARRAY [T] export {NONE} all
             rename put as array_put,
                    make as array_make,
                    count as capacity
             end
   creation make
   feature
       count: INTEGER;
   --hace efectiva como atributo una
   --característica diferida
       make (n: INTEGER) is
             require   tamaño_no_negativo: n>=0;
             do
             array_make (1,n)
             ensure
             capacidad: capacity = n;
             vacia:  count = 0
             end
```

Implementación de Pilas usando arrays 2/2

```
    full: BOOLEAN is do
        --¿Está llena la representación de la
    pila?
                Result:= (count = capacity)
    end;

    put (x: T) is
        -- Pone x en la cima de la pila
                require  not full
                do
                count:= count + 1;
                array_put (x, count)
    end;

    invariant
        count>=0 ;
        count <= capacity
                ...
end -- PILA_FIJA [T]
```

Ejemplo: Figuras Compuestas

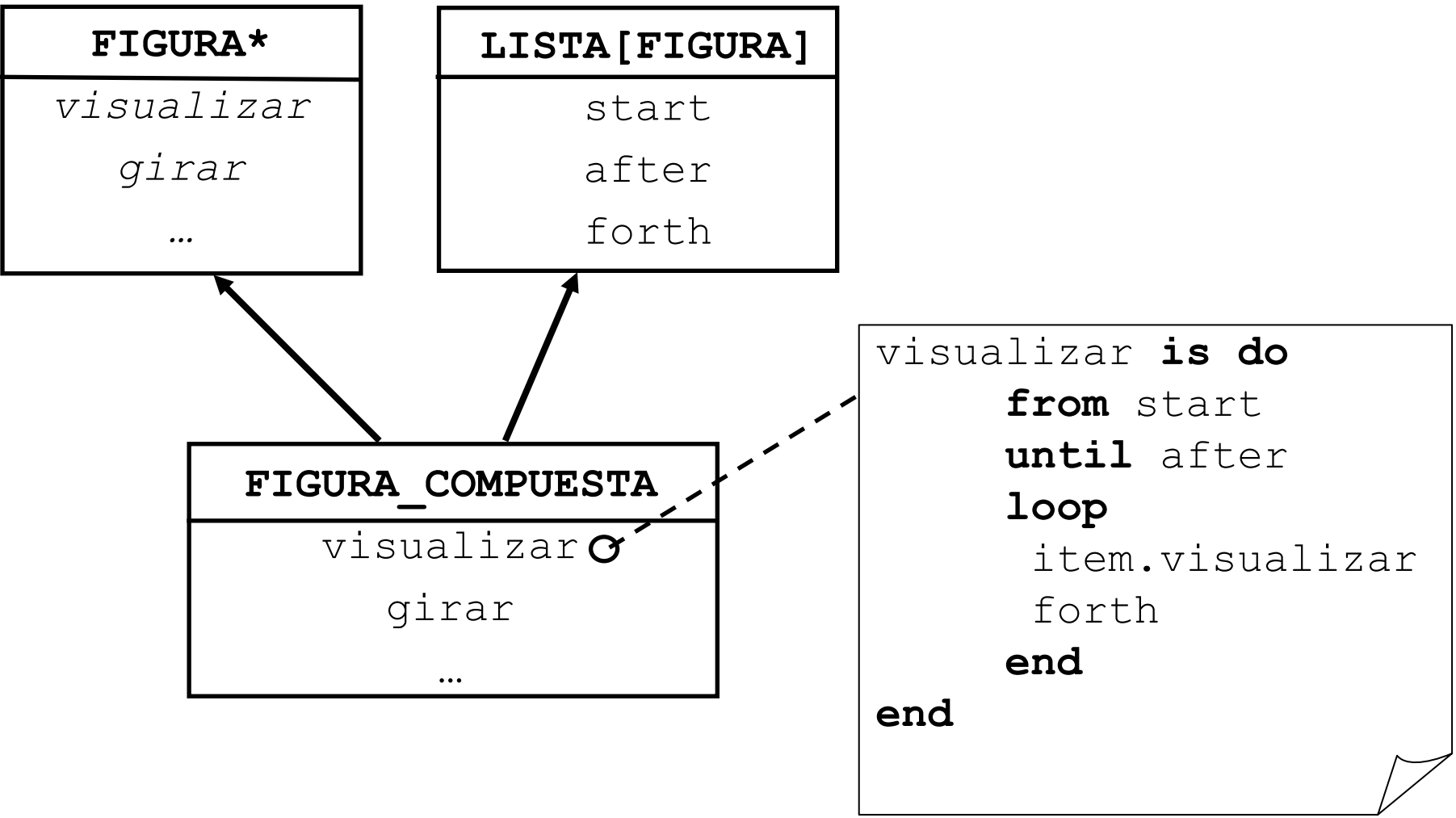
Vamos a ver un **patrón de diseño** general (de utilidad en muchas áreas) que describe estructuras compuestas a través de la herencia múltiple, usando una clase contenedora (lista) como una de sus clases padre, **PATRÓN COMPOSITE**.

```
class FIGURA_COMPUESTA inherit
                          FIGURA
                          LINKED_LIST [FIGURA]
feature
        …
end
```

"Una figura compuesta es una figura"

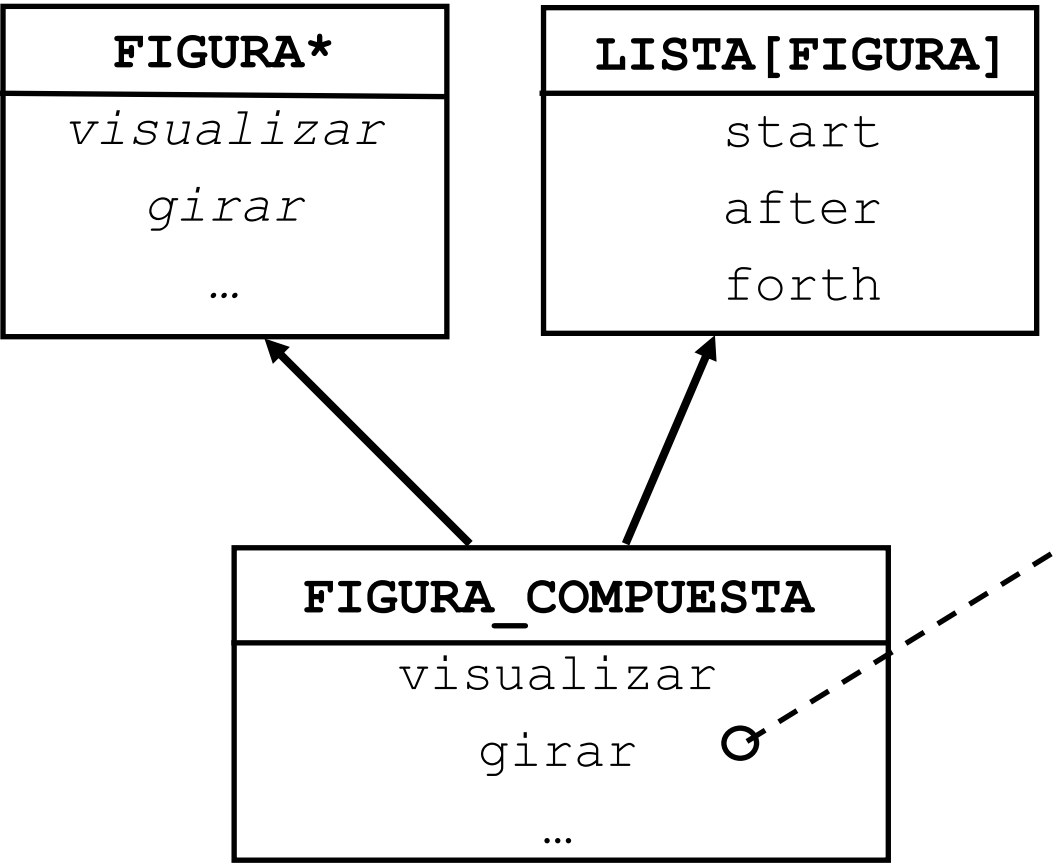
"Una figura compuesta es una lista de figuras"

Figura Compuesta con herencia múltiple



"Muchas rutinas con esta estructura" (rotar, trasladar, …)

Figura Compuesta con herencia múltiple




```
girar(…) is do
        from start
        until after
        loop
          item.girar(…)
          forth
        end
end
```

"Muchas rutinas con esta estructura" (rotar, trasladar, …)

Patrón Composite: Herencia simple

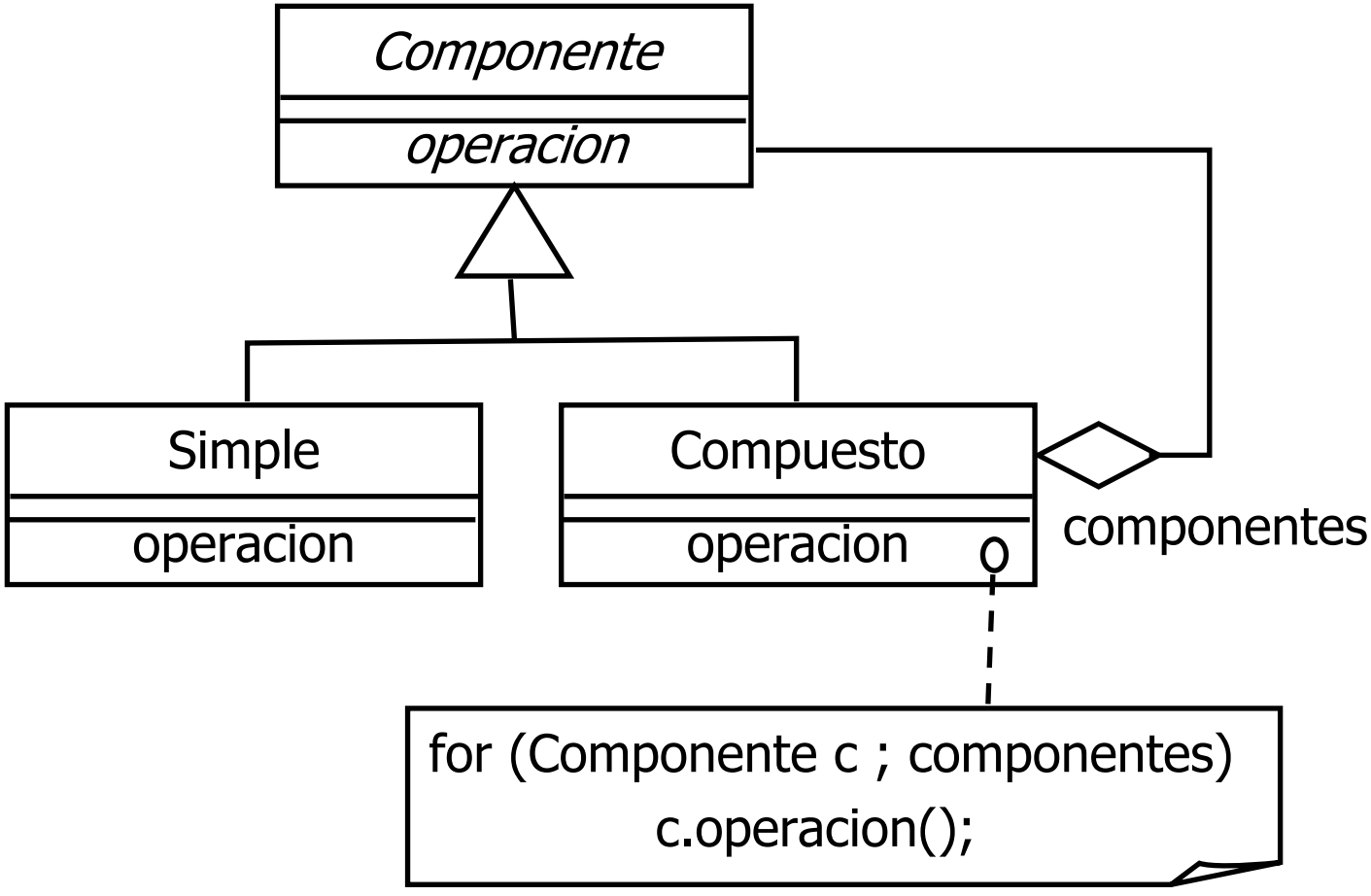
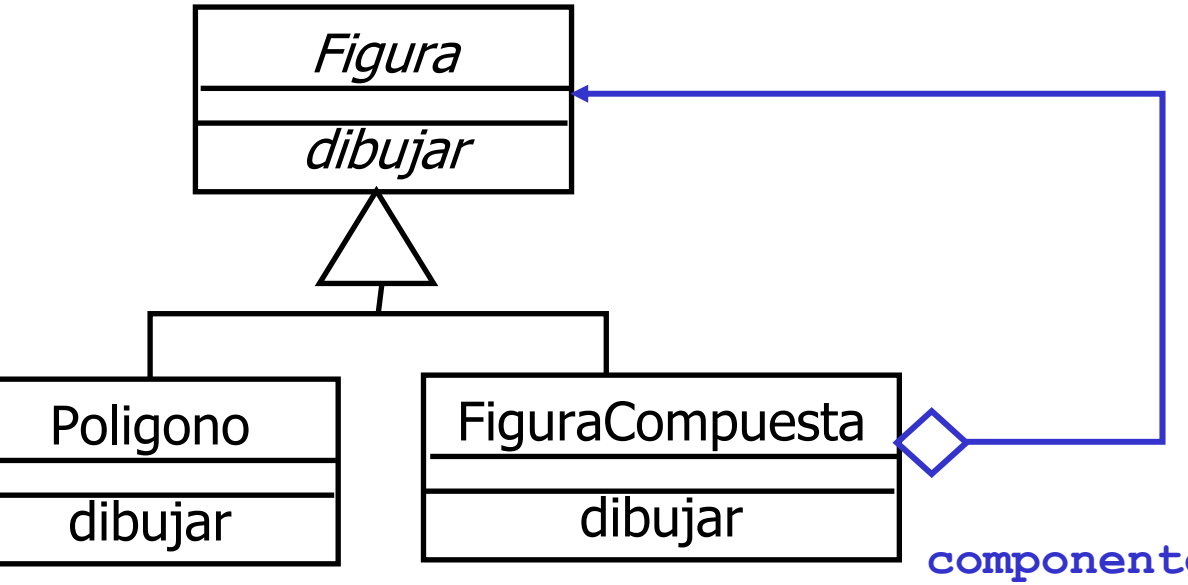
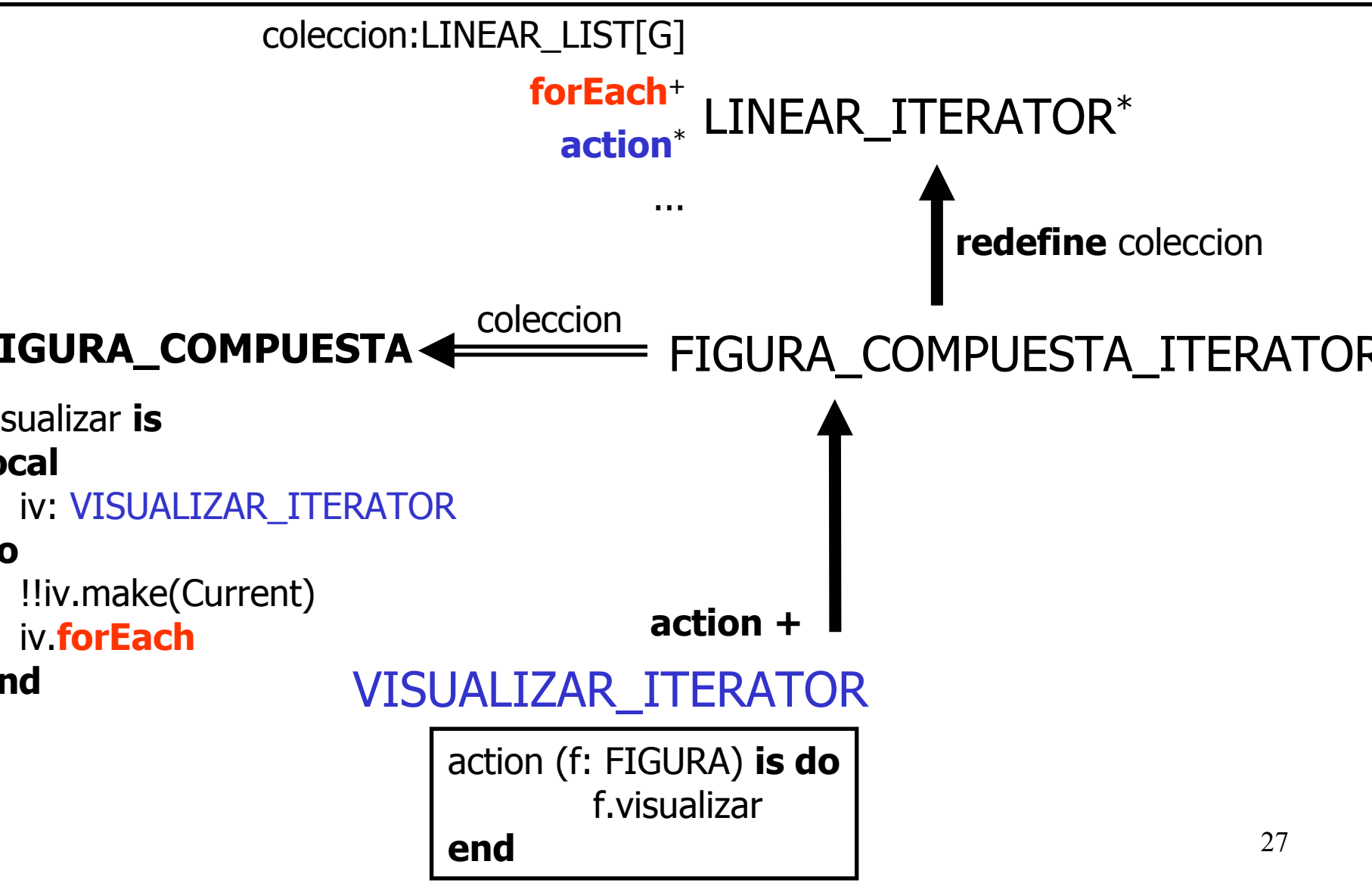


Figura Compuesta con Herencia simple

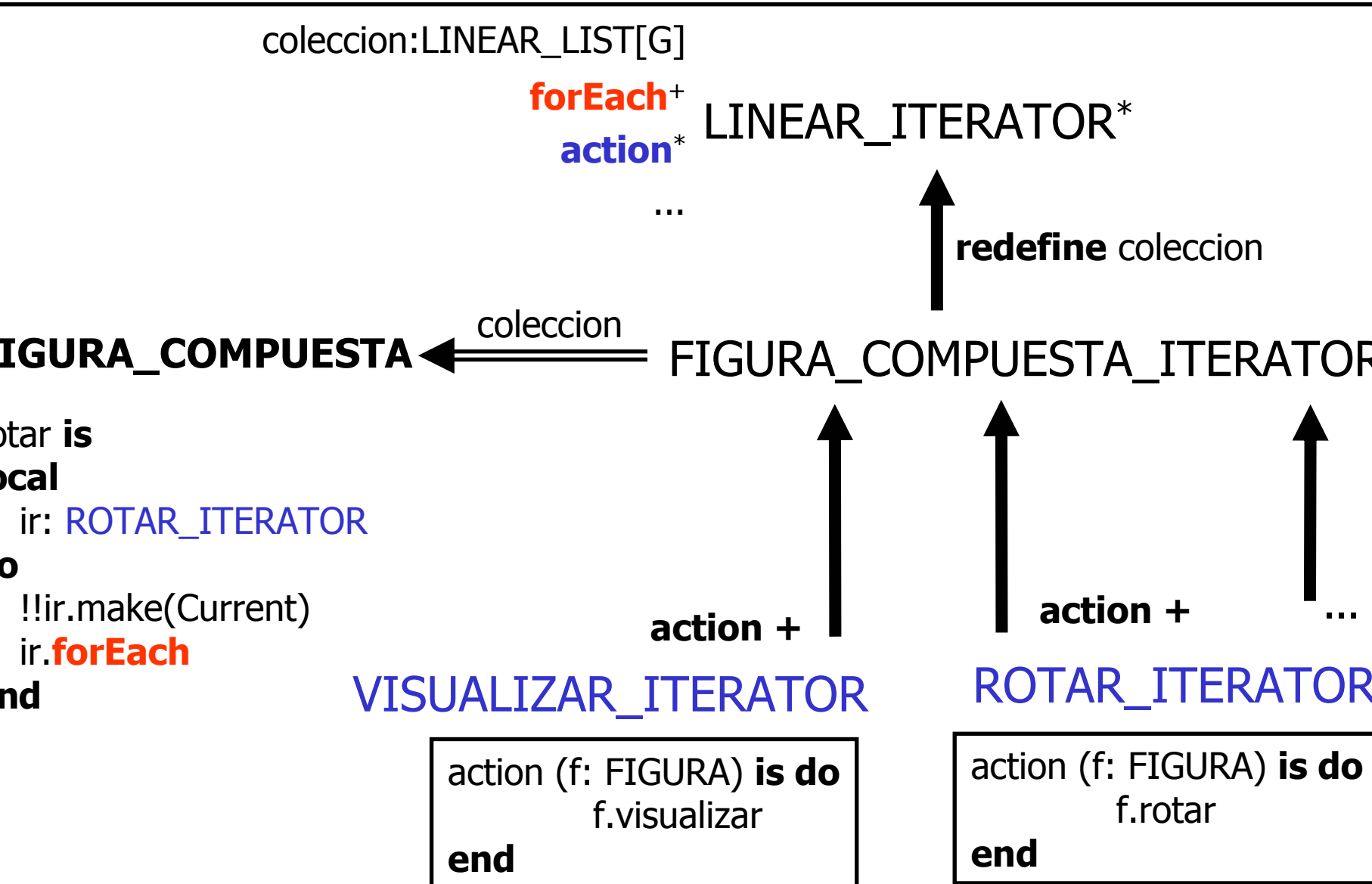



```
componentes: LINKED_LIST[FIGURA
dibujar is do
    from componentes.start
    until componentes.after
    loop
        componentes.item.dibuja
        componentes.forth
    end
end
```

Figuras Compuestas e Iteradores



Figuras Compuestas e Iteradores



```
deferred class FIGURA_COMPUESTA_ITERATOR inherit
      LINEAR_ITERATOR [FIGURA_COMPUESTA]
            redefine coleccion end
creation make
feature
   coleccion: FIGURA_COMPUESTA
   -- Colección sobre la que itera (lista de figuras)
   -- antes era Linear_List[G]
…
end
```

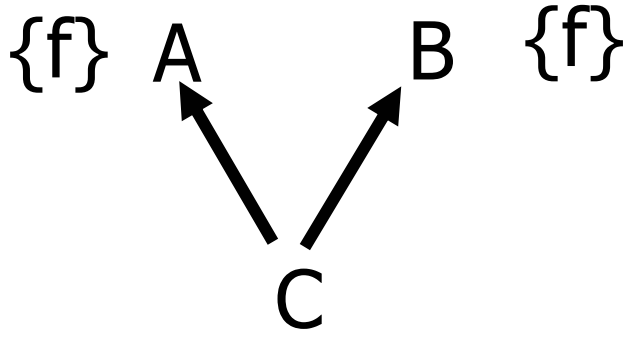
```
class VISUALIZAR_ITERATOR inherit
   FIGURA_COMPUESTA_ITERATOR
            redefine action end
creation make
feature
   action(f: FIGURA) is do
       f.visualizar
   end
end
```

```
class FIGURA_COMPUESTA inherit
                    FIGURA
                    LINKED_LIST[FIGURA]
feature
   ...
   visualizar is
   local
       iv: VISUALIZAR_ITERATOR
   do
       !!iv.make (Current)
       -- colecciOn:= Figura_compuesta actual
       iv.forEach
   end
   ...
end
```

"NO HAY NADA MALO EN TENER CLASES TAN PEQUEÑAS"
"NO ES ACEPTABLE PASAR RUTINAS COMO ARGUMENTOS"
(B. Meyer)

3.- Problemas con la Herencia Múltiple

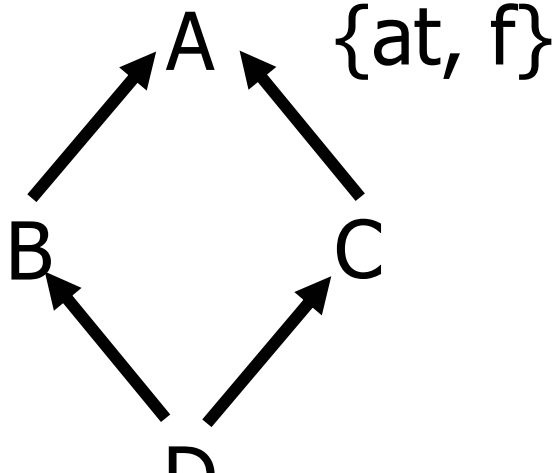
(3.a) Colisión de nombres



Solución:

Renombrar

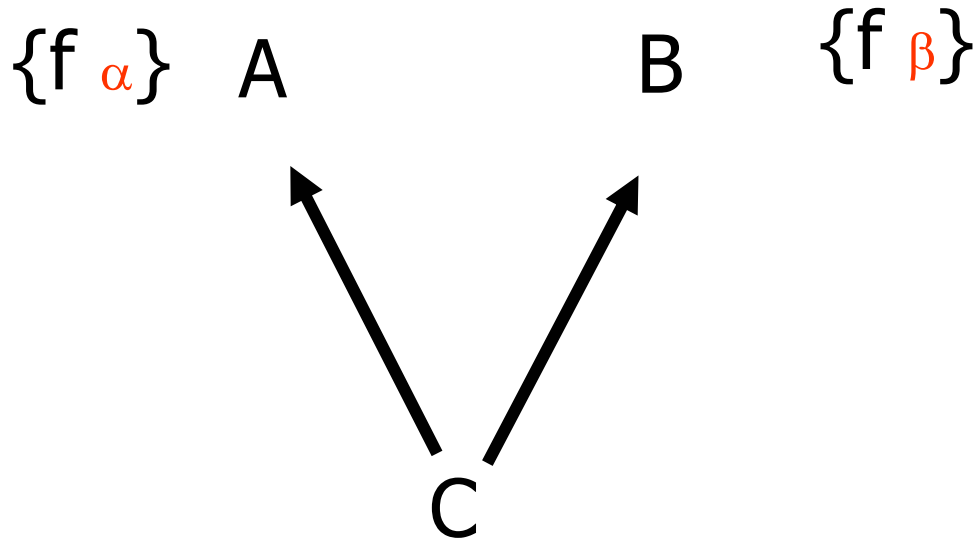
(3.b) Herencia repetida



Solución:

¿compartir o duplicar?

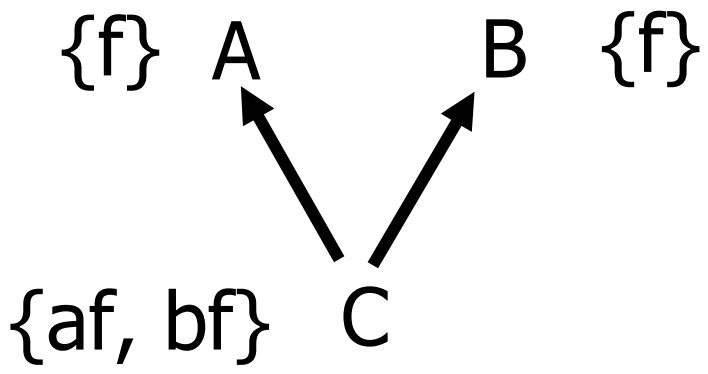
3.a) Colisión de nombres



- Heredamos dos funciones efectivas con el mismo nombre y diferentes implementaciones

Colisión de nombres en Eiffel

- Se considera un problema sintáctico
- Debe ser resuelto por la clase que hereda
- **Solución:** **RENOMBRAR** en la clase C, al menos una de las dos características heredadas que colisionan



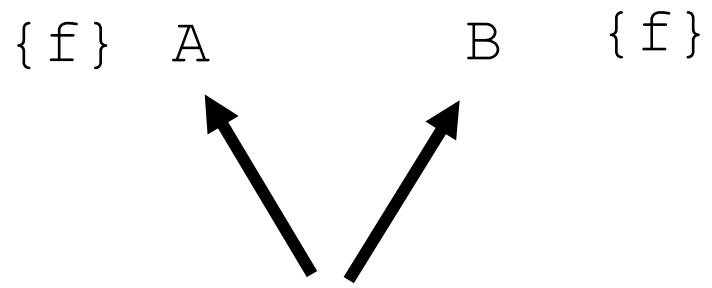

```
class C inherit
    A rename f as af
    B rename f as bf
feature
...
end
```

- No hay colisión de nombres si:
  - i) colisiona una rutina diferida con otra efectiva
  - ii) hay herencia repetida con compartición

Colisión de nombres en C++

- No es posible renombrar
- ¿No sería suficiente con la **sobrecarga**? NO
  - La sobrecarga no se aplica entre diferentes alcances de clase
  - Las ambigüedades entre funciones de clases bases diferentes no se resuelven basándose en el tipo de los parámetros
- **Solución:** Calificación de las rutinas
  - poco adecuada para los clientes
  - elimina la **Ligadura Dinámica** (A::f puede aplicarse desde cualquier clase)

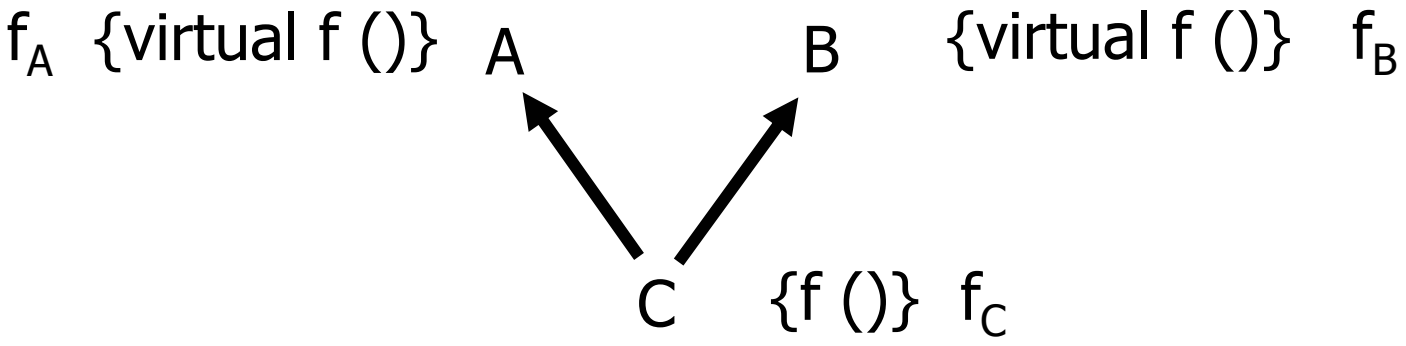
Colisión de nombres en C++




```
lass C: private A, private B {
 public:
     void af ( ) {return A::f ()};
     void bf ( ) {return B::f ()};
/ reexportar las otras funciones de A y B no renombradas
..
```

Colisión de nombres en C++

¿Y si las funciones son **virtual** y quiero conservar ambas porque tienen significados distintos?

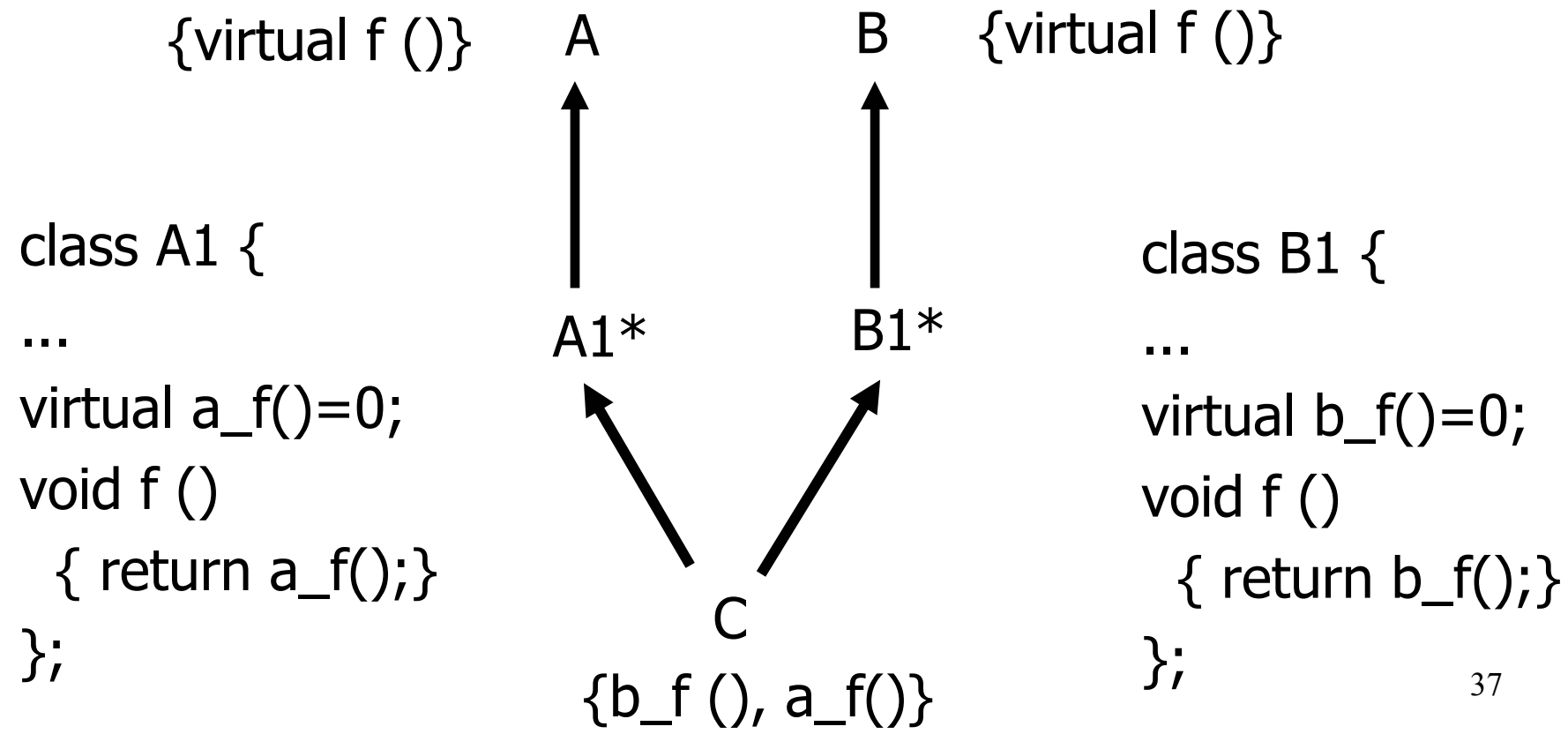



```
C* pc = new C(); A* pa = pc;  B* pb = pc;

pa -> f (); pb -> f (); pc -> f ()
```

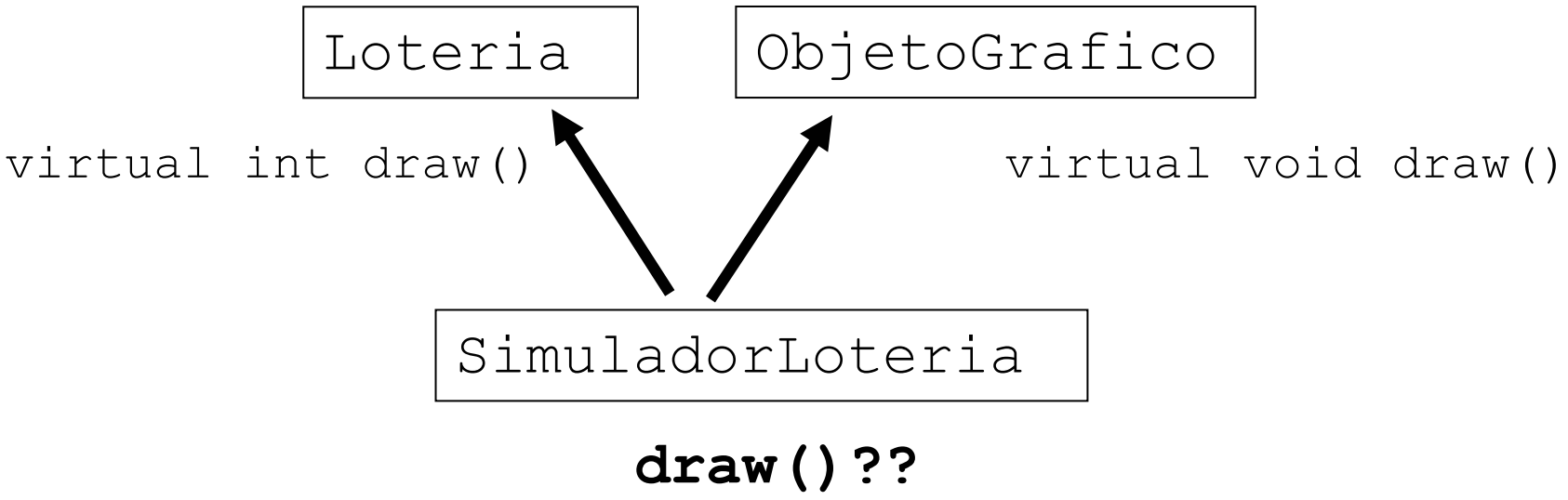
¡¡¡ Todos los mensajes invocan a `C::f()` !!! y la redefinición de ambas funciones con una única función f en C sería erróneo porque a pesar del nombre común las funciones f no están relacionadas.

Cambio de nombre en C++

Solución: Introducir una **clase extra** por cada clase que tenga un
función virtual a la que se quiera cambiar el nombre e
la que se define el **nuevo nombre** de la función.



Ejemplo: quiero redefinir los métodos de las clases bases y mantener los dos.

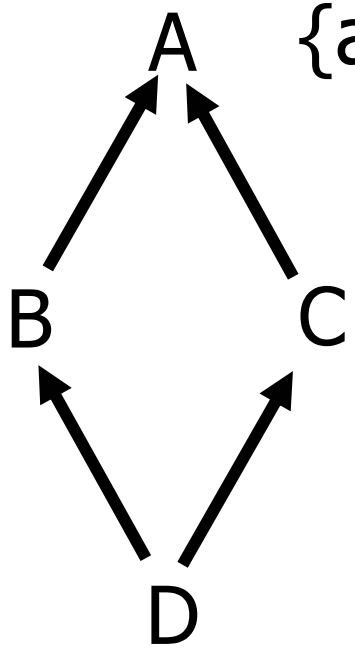


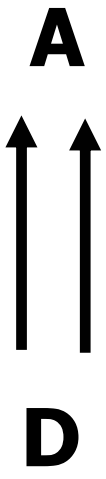
- `Loteria::draw` y `ObjetoGrafico::draw` tiene significados distintos (sacar y dibujar)

Ejemplo: quiero redefinir los métodos de las clases bases y mantener los dos.

```
class Lloteria:public Loteria{
   virtual int drawLoteria()=0;
   int draw() {
        return drawLoteria(); //redefine Loteria::draw
   }
};

class OObjetoGrafico: public ObjetoGrafico{
   virtual void drawGrafico()=0;
   void draw(){
        drawGrafico();       //redefine ObjetoGrafico::draw
   }
};

class SimuladorLoteria: public LLoteria,
                        public OObjetoGrafico{
   //...
    int drawLoteria(); //redefine y cambia el nombre de Loteria::draw
    void drawGrafico(); //redefine y cambia el nombre de
                        //ObjetoGrafico::draw
};
```

(3.b) Herencia repetida

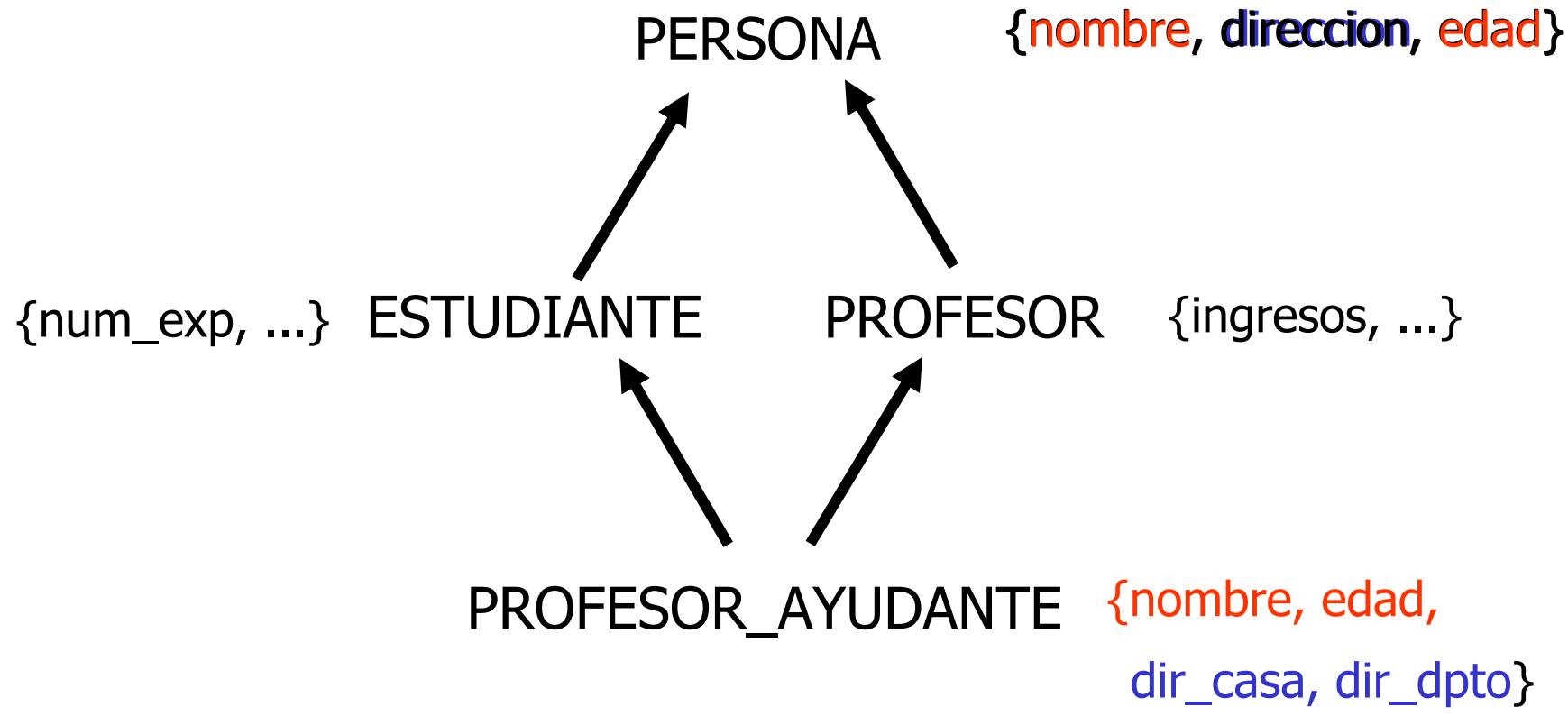




¿Qué sucede con las propiedades heredadas más de una vez?

¿**DUPLICAR** o **COMPARTIR?**

Ejemplo: Herencia repetida



- **edad** ⇒ Compartición
- **direccion** ⇒ ¿particular o del Dpto? ⇒ Duplicación

Herencia repetida: ¿Duplicar o Compartir?

Sea la clase **D** y $\mathbf{B_1}$, .., $\mathbf{B_n}$ ($n \geq 1$) son ascendientes de **D** que tienen la clase **A** como ascendiente común; sean $\mathbf{f_1}$, .., $\mathbf{f_n}$, características de $\mathbf{B_1}$, .., $\mathbf{B_n}$, respectivamente, que tienen como "semilla" la propiedad **f** de **A**, entonces:

1) Cualquier subconjunto de $\mathbf{f_1}$, .., $\mathbf{f_n}$ heredado **bajo el mismo nombre** final genera una única característica en **D**

(COMPARTICIÓN)

2) Cualesquiera dos de las $\mathbf{f_1}$, .., $\mathbf{f_n}$ heredadas **bajo diferente nombre**, generan características diferentes en **D**

(DUPLICACIÓN)

El primer caso es lo que normalmente se necesita

Herencia repetida: ¿Duplicar o Compartir?

Sea la clase **D** y $\mathbf{B_1}$, .., $\mathbf{B_n}$ (n≥1) son ascendientes de **D** que tienen la clase **A** como ascendiente común; sean $\mathbf{f_1}$, .., $\mathbf{f_n}$, características de $\mathbf{B_1}$, .., $\mathbf{B_n}$, respectivamente, que tienen como "semilla" la propiedad **f** de **A**, entonces:

atributos y rutinas

1) Cualquier subconjunto de $\mathbf{f_1}$, .., $\mathbf{f_n}$ heredado **bajo el mismo nombre** final genera una única característica en **D**

(COMPARTICIÓN)

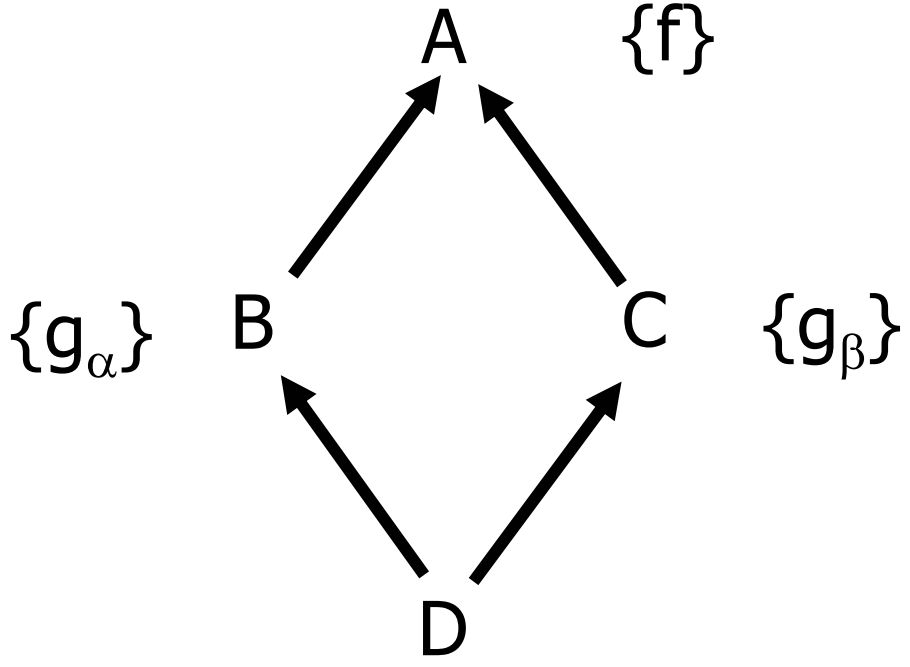
2) Cualesquiera dos de las $\mathbf{f_1}$, .., $\mathbf{f_n}$ heredadas **bajo diferente nombre**, generan características diferentes en **D**

(DUPLICACIÓN)

El primer caso es lo que normalmente se necesita

Regla del nombre único

Dos **características efectivas** diferentes de una misma clase **NO** pueden tener el mismo nombre final.

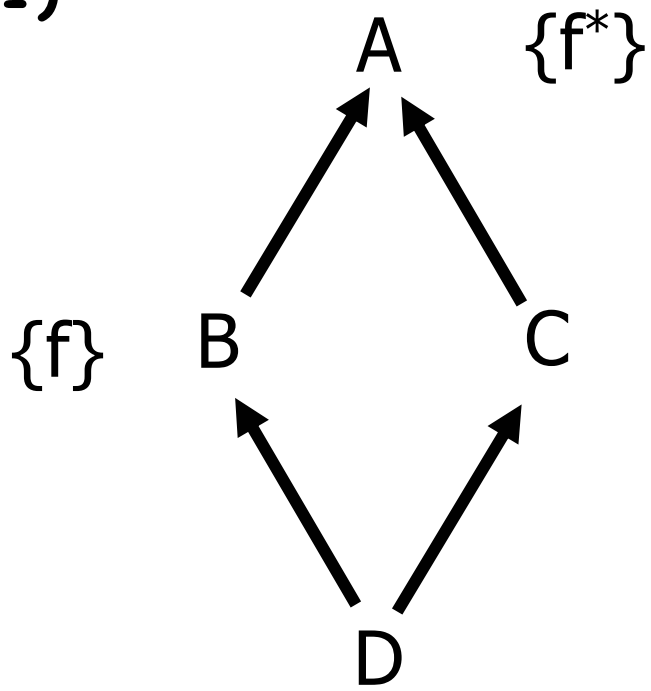


- f no provoca conflicto (compartición)
- g causa conflicto

b.1) Conflicto cuando hay compartición

Si se heredan dos características bajo el mismo nombre puede ocurrir:

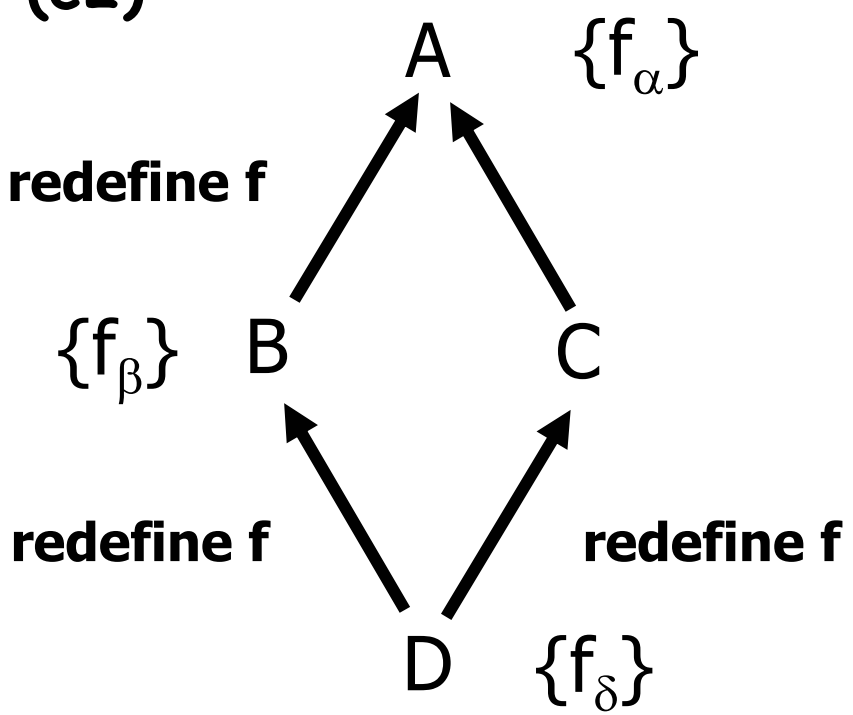
(c1)



LEGAL

Solo hay conflicto entre dos efectivas

(c2)

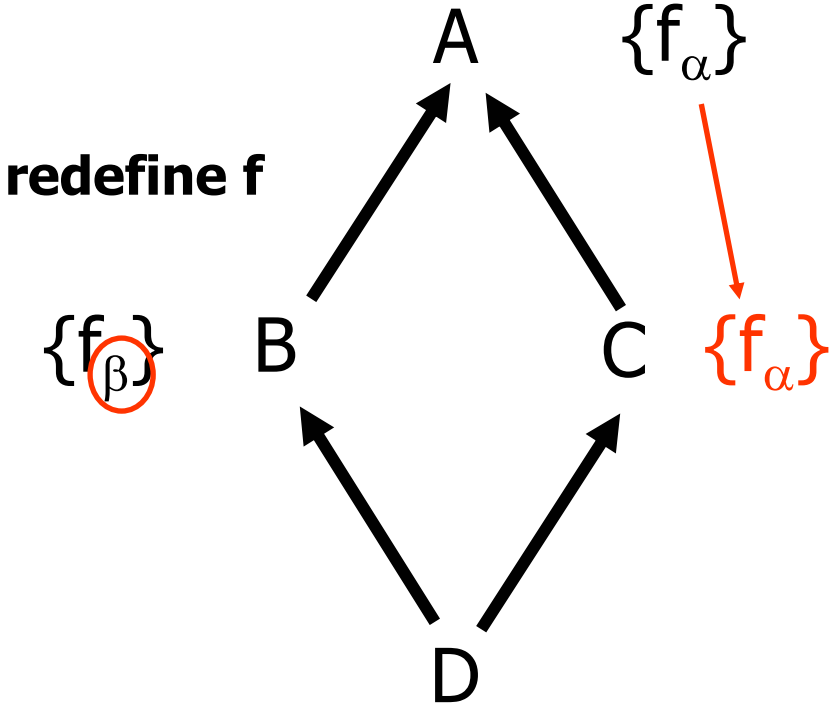


LEGAL

Ambas versiones se funden en una nueva

Conflictos cuando hay compartición:

(c3) Ambas versiones efectivas y ambas no se redefinen:



ILEGAL

Viola la Regla del nombre único

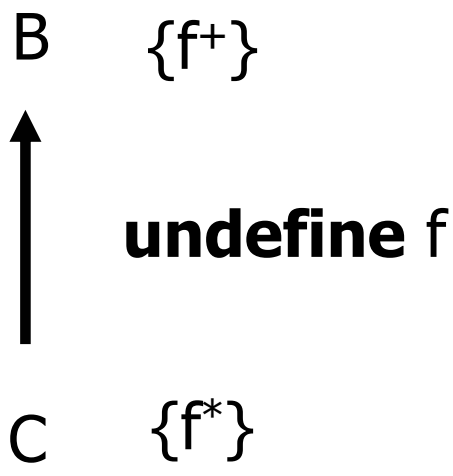
Soluciones:

- **rename** ⇒ duplicación
- **undefined** ⇒ (c1)

Conversión en diferida

- Dejar que una de las variantes se imponga sobre las otras
- Es posible al heredar transformar una rutina efectiva en diferida.

```
class C inherit
        B
         undefine f
feature
        ...
end
```

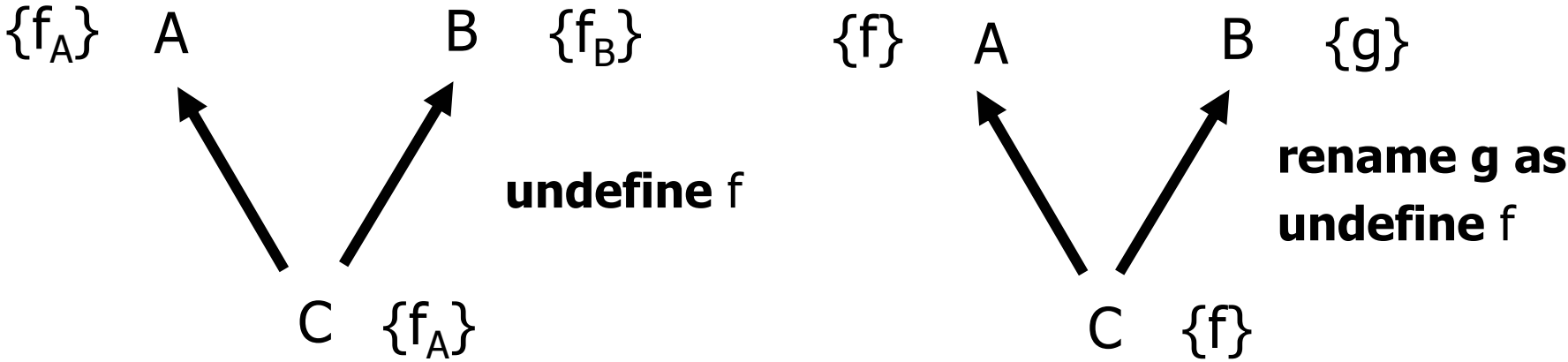


- Viene después de **rename** (se aplica al nombre final) pero antes de **redefine**.

Undefinición y unión

- El mecanismo **undefine** proporciona la capacidad de unir características bajo herencia múltiple (no necesariamente repetida).

- **Ejemplo:** Se desea que C trate a f y g como una única característica (requiere semántica y signatura compatibles)

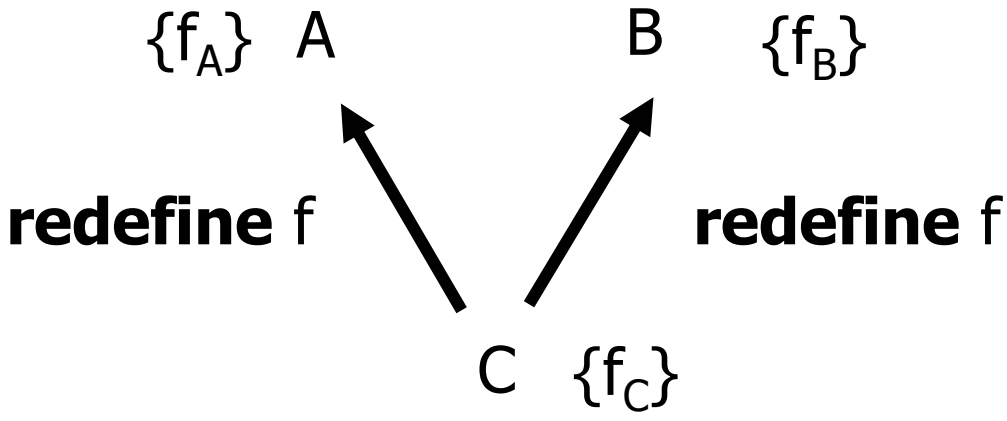


A impone la característica

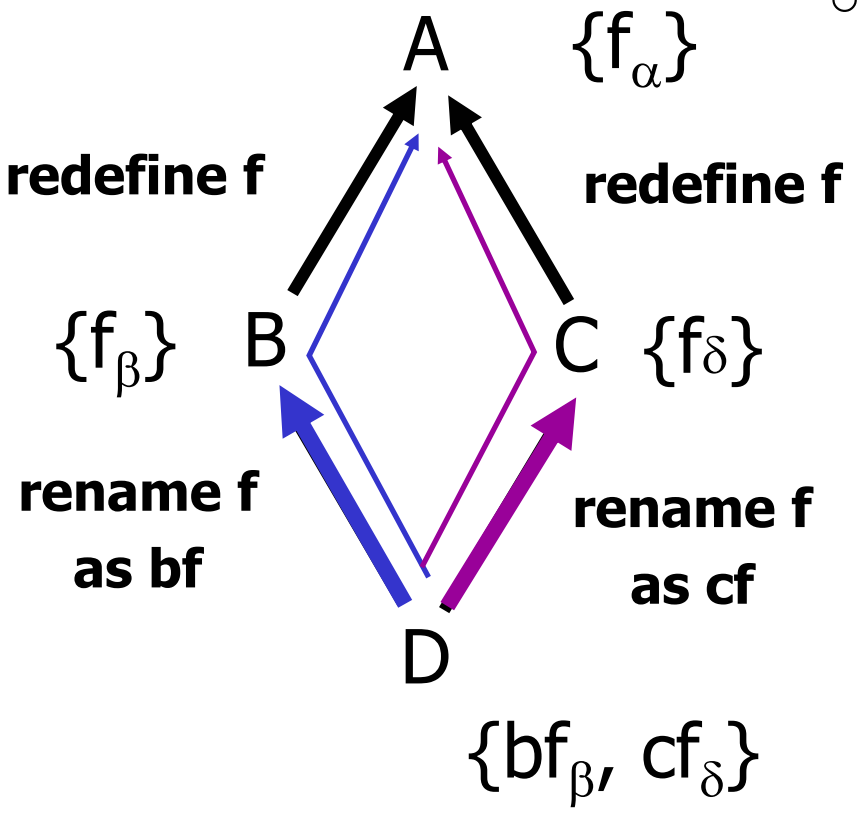
Combinación de propiedades

Todas las restantes combinaciones son posibles:

- tomar la característica de A y el nombre de B
- renombrar f y g y darle un nombre nuevo en C
- Reemplazar ambas versiones por una nueva (caso (c2))
  - ambas tienen el mismo nombre final (utilizar rename si no)



o.2) Conflicto cuando hay duplicación




```
oa:A; ob:B; oc:C; od:D; !!od
```

```
ob:= od;
ob. f          {β}

oc:= od;
oc. f          {δ}

oa:= od
oa.f       {β o δ}
```

No hay conflicto de nombres pero surge un nuevo problema debido a la **ligadura dinámica**.

Solución al conflicto con la duplicación

```
class D inherit
      B
         rename f as bf;
         select bf          -- elimina ambigüedad
      C
         rename f as cf;

feature
      ...
end
```

La clausula **select** debe aparecer después de rename, undefine y redefine.

Solución al conflicto con la duplicación

```
class D inherit
      B
         rename f as bf;

      C
         rename f as cf;
         select cf          -- elimina ambigüedad
feature
      ...
end
```

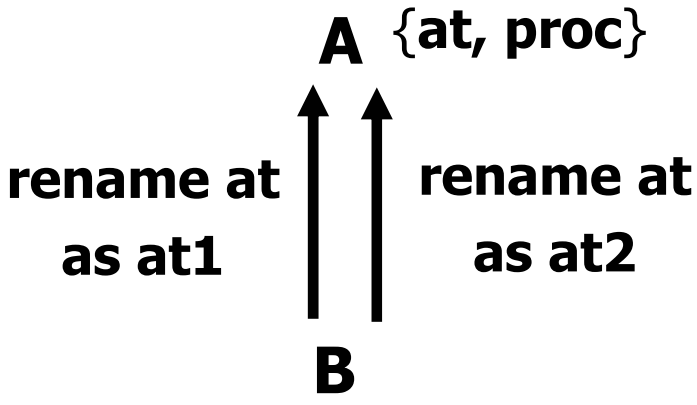
Regla del Select

Una clase, que de un antecesor repetido, hereda dos o más versiones efectivas diferentes de una característica y no las redefine a ambas, debe entonces incluir a una de las dos en una cláusula select

La clausula **select** debe aparecer después de rename, undefine y redefine.

Ejemplo: Conflicto con la duplicación

```
class A feature
    at: INTEGER
    proc is do
       print(at)
    end;
end

class B inherit
    A
       rename at as at1;
    select at1
    A
       rename at as at2
feature
 ...
end
```

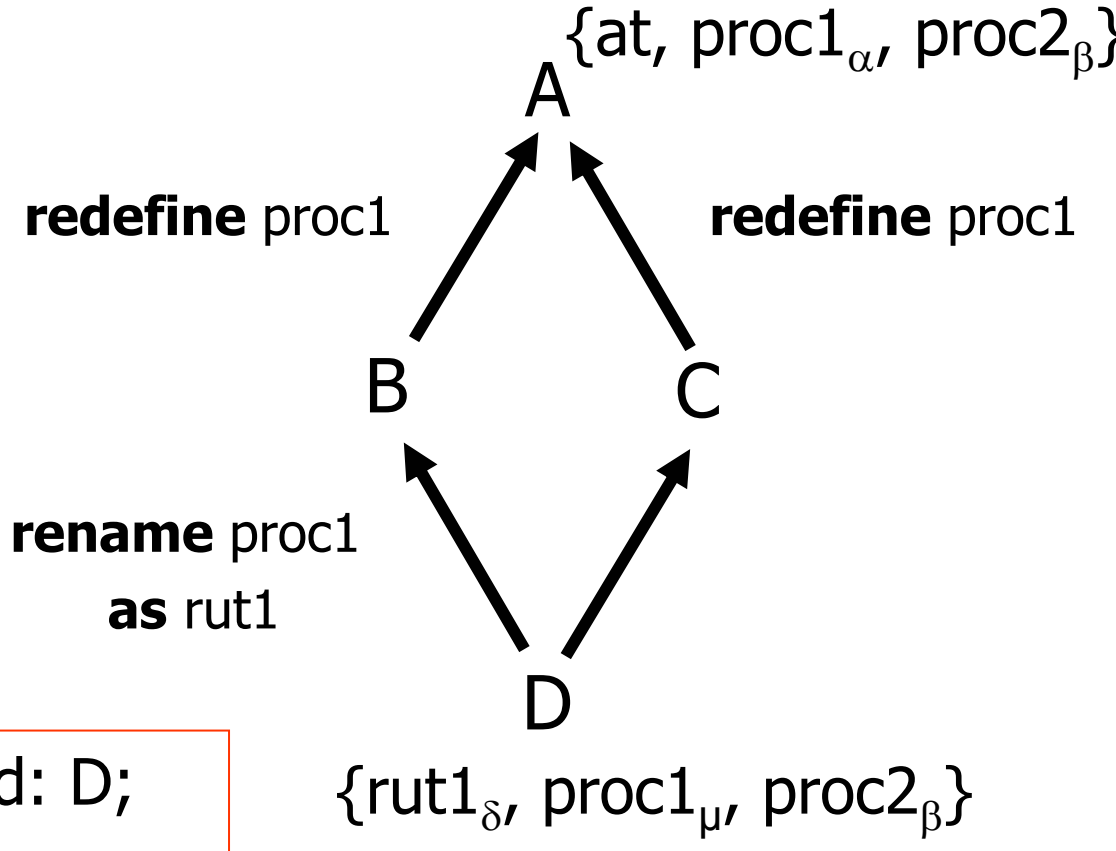



```
oa: A; ob: B
!!ob
oa:= ob
oa. proc
```

¿se imprime at1 o at2?

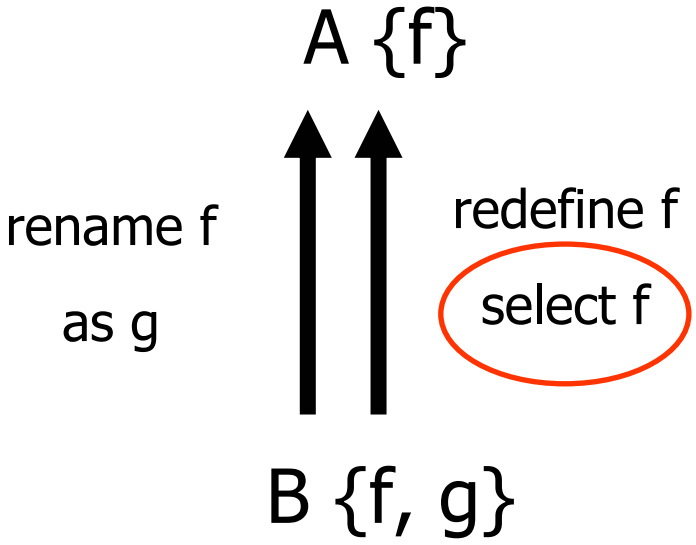
Ejemplo: Conflicto con la duplicación

```
class A feature
    at: INTEGER
    proc1 is do .. end;
    proc2 is do
          proc1
          print(at)
    end;
end
```




```
oa:A; od: D;
!!od;
oa:=od;
oa. proc2 ??
```

Utilidad de la herencia repetida: mantener la versión original junto con la redefinida.

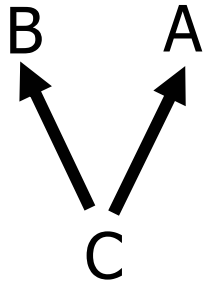



```
class B inherit
    A
        redefine f
        select f
    A
        rename f as g
feature
    ...
end
```

- Antes de introducir **`Precursor`** era la técnica usada para que una rutina redefinida pudiese invocar a la rutina original.

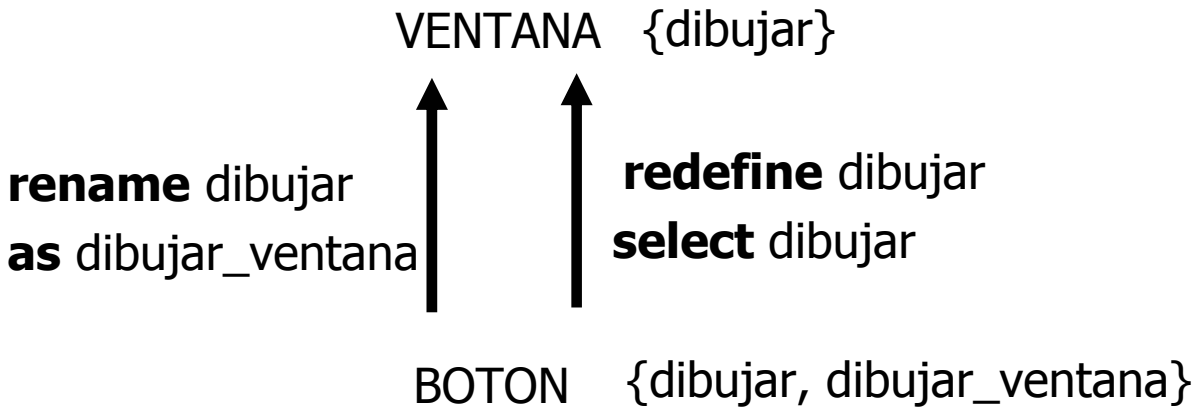
Entidad "Precursor" (Eiffel)

- ¿Que sucede si hay herencia múltiple y una rutina tiene varios precursores ?



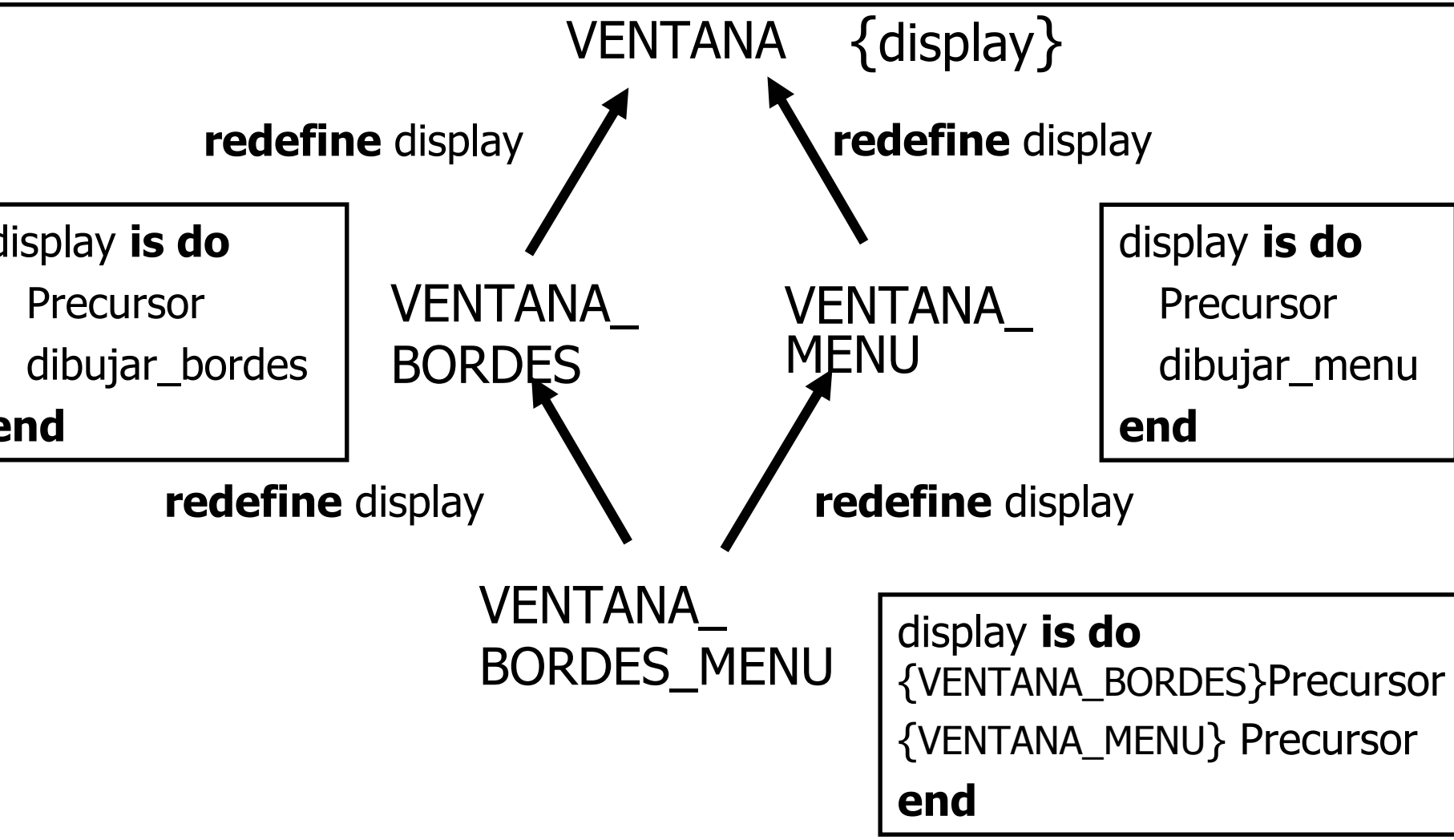
{A} Precursor
{B} Precursor

- ¿Que hacemos si queremos disponer en la nueva clase de la rutina original y la redefinida?



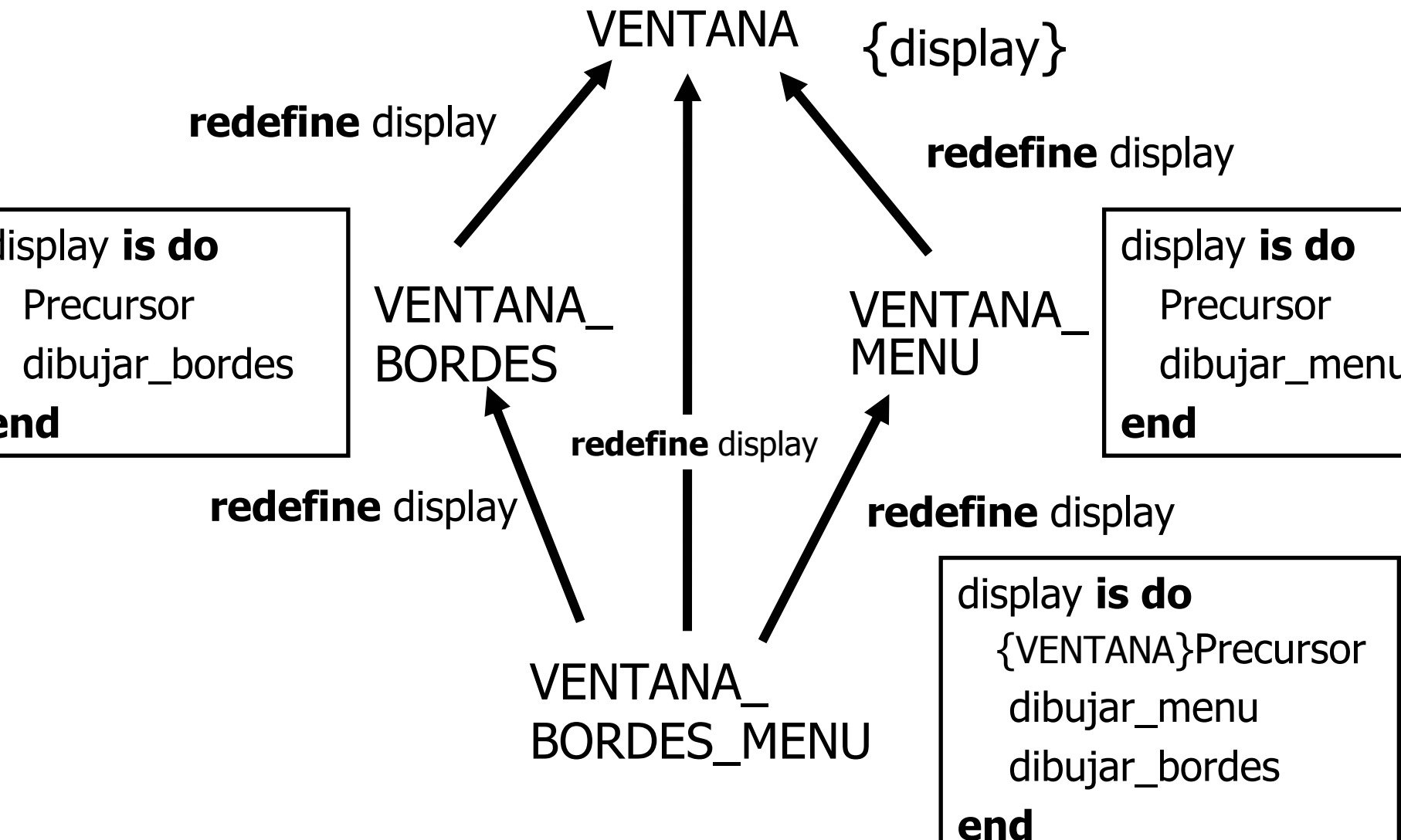
¿En JAVA?

Ejemplo:



display de ventana se llama dos veces !!

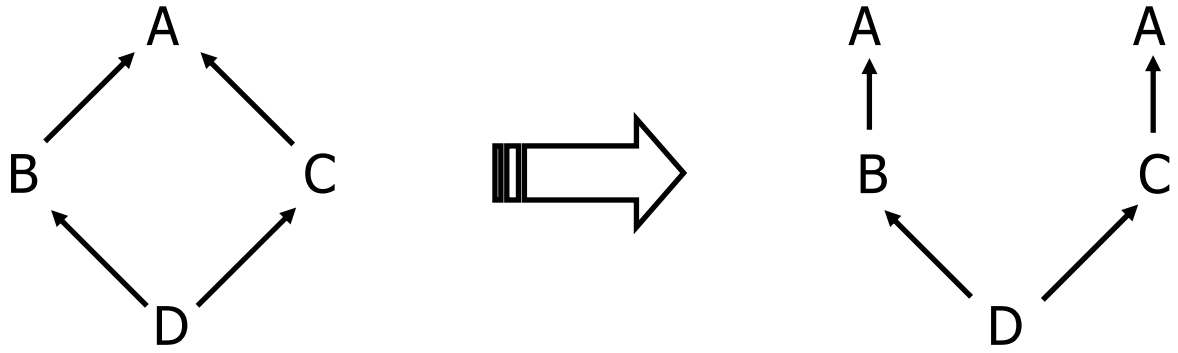
Solución: no será necesario el uso de la cláusula select en VENTANA_BORDES_MENU ¿por qué?



Herencia repetida en C++: Replicación

- Por defecto, **NO HAY COMPARTICIÓN, SE DUPLICA TODO**

```
class A {int at1; ...}
class B: public A {...};
class C: public A {...}
class D: public B, public C {...}
```

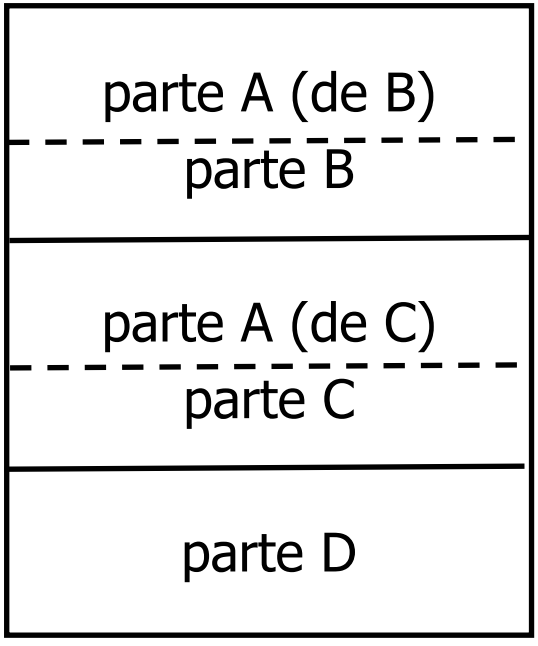


- El nivel de granularidad para decidir si compartir o duplicar es la **clase**
- **Ambigüedad** se resuelve con **calificación de nombres**:

C::at1; B::at1

Herencia repetida en C++

- Hay dos objetos A en un objeto D
- La asignación entre un puntero a A y otro a D no está permita, es ambigua



Estructura del objeto D

```
D* od = new D();
A* oa = od;          //ERROR: ambiguo

oa=(A*)od;           //ERROR: ambiguo

oa=(A*)(B*)od;       //OK!!
                     //asigna el
                     //subobjeto A de B
```

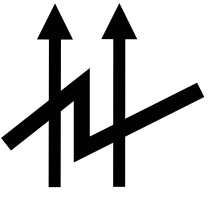
Herencia repetida en C++

class B {public: int b; ...}

class D: **public** B, **public** B {...} {**ILEGAL**}

No se pueden resolver las ambigüedades.

B {int b}



D

```
void f (D *p)
{
        p->b=7;          //ambiguo
}
```

Herencia repetida en C++: Compartición

Si queremos que la clase derivada herede sólo una copia de la clase "abuelo" entonces las clases intermedias tienen que declarar que su **herencia** de la clase paterna es **virtual**.

Clase base virtual

```
class A {char* at1; int f (); ...}
class B: virtual public A {...};
class C: virtual public A {...}
class D: public B, public C {...}
```

- Significado distinto en este contexto que en el de las funciones
- **No** significa que todas las funciones son virtuales

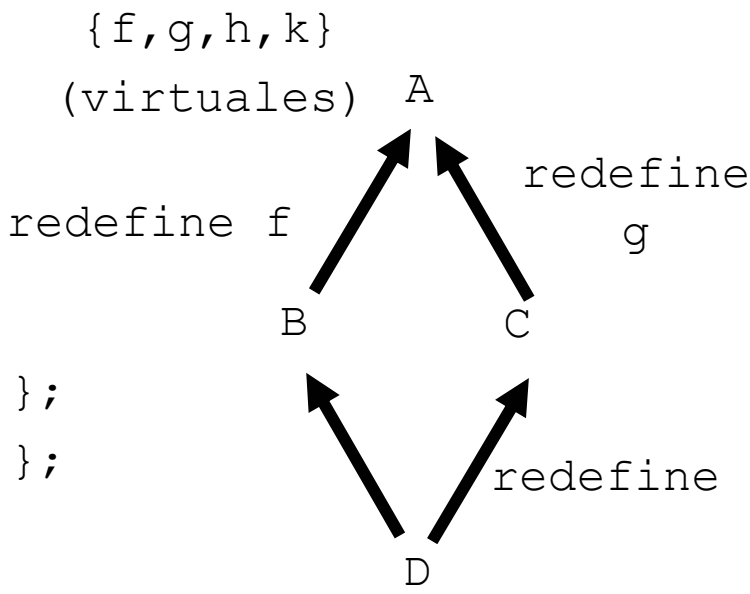
Mecanismo que se opone a la extensibilidad

Redefinición de funciones de una clase base virtual

- Cuando se utilizan **clases base virtuales** se puede llegar a la misma característica a través de mas de un camino **sin** que exista **ambigüedad**.
- La versión redefinida **domina** a la virtual
- La misma situación con clases base no virtuales produce **ambigüedad** (implica más de un objeto)
- ¿Qué pasaría si clases derivadas diferentes redefiniesen la misma función?
  - Es válido si y sólo si las dos funciones que redefinen la función se unen en otra en una clase derivada común.

Redefinición de funciones de una clase base virtual

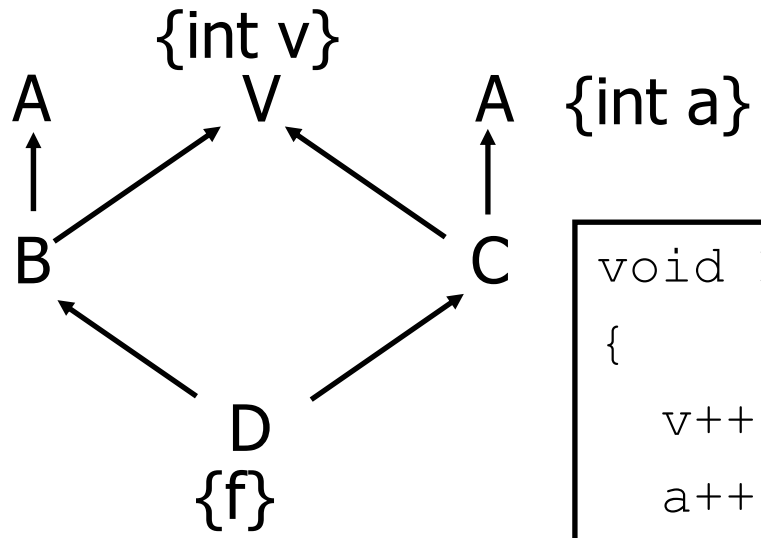
```
class A{ public virtual f();
         public virtual g();
         public virtual h();
         public virtual k();
};
class B: public virtual A{ public f();};
class C: public virtual A{ public g();};
class B: public B , C{ public h();};
```




```
D* od= new D();
od->f();     //B::f()
od->g();     //C::g()
od->h();     //D::h()

C* oc = od;
oc->f();     //B::f()
```

Herencia repetida en C++ y clases virtuales

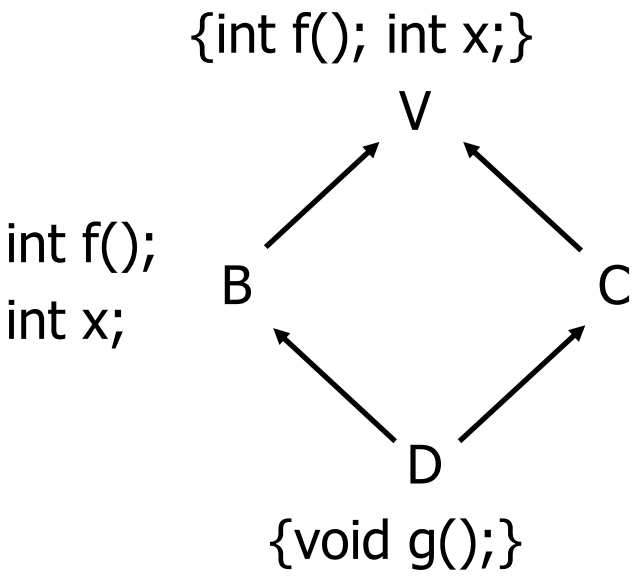
```
class V {public: int v};
class A {public: int a};
class B:public A, public virtual V {…};
class C:public A, public virtual V {…};
classD : public B, public C {public: void f();};
```




```
void D::f()
{
  v++;  //correcto sólo hay un 'v'
  a++;  //ambiguo hay dos 'a'
}
```

Herencia repetida en C++ y clases virtuales

```
class V {public: int x; int f();};
class B: public virtual V {
        public: int x; int f();};
class C: public virtual V {…};
classD : public B, public C {void g();};
```




```
void D::g()
{
   x++;  //OK!  B::x domina a V::x
   f();  //OK!  B::f() domina a V::f()
}
```

2.-Herencia múltiple en Java

- Java soporta **herencia simple**.
- Java soporta **herencia múltiple de interfaces**.
- Java permite que una clase pueda heredar de más de una INTERFACE
- **Interface** es el modo de declarar un tipo formado sólo por ***métodos abstractos (públicos***) y ***constantes***, permitiendo que se escriba cualquier implementación para estos métodos.

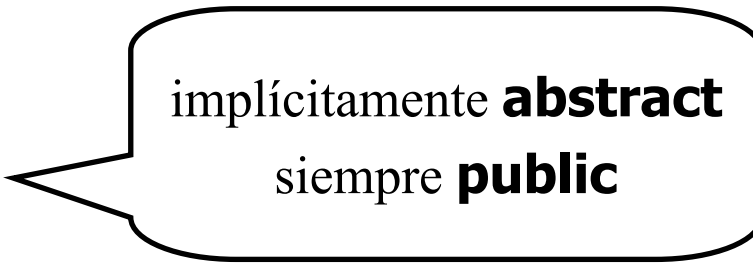
Interfaces vs clases abstractas

- **Una interface no es una clase abstracta**.
  - Una clase abstracta puede estar parcialmente implementada.
  - Una clase puede heredar de una única clase, incluso si sólo tiene métodos abstractos.

- **Importante**:
  - Es conveniente que el programador Java declare interfaces para las clases que cree, ya que NO EXISTE HERENCIA MULTIPLE.

Ejemplo:

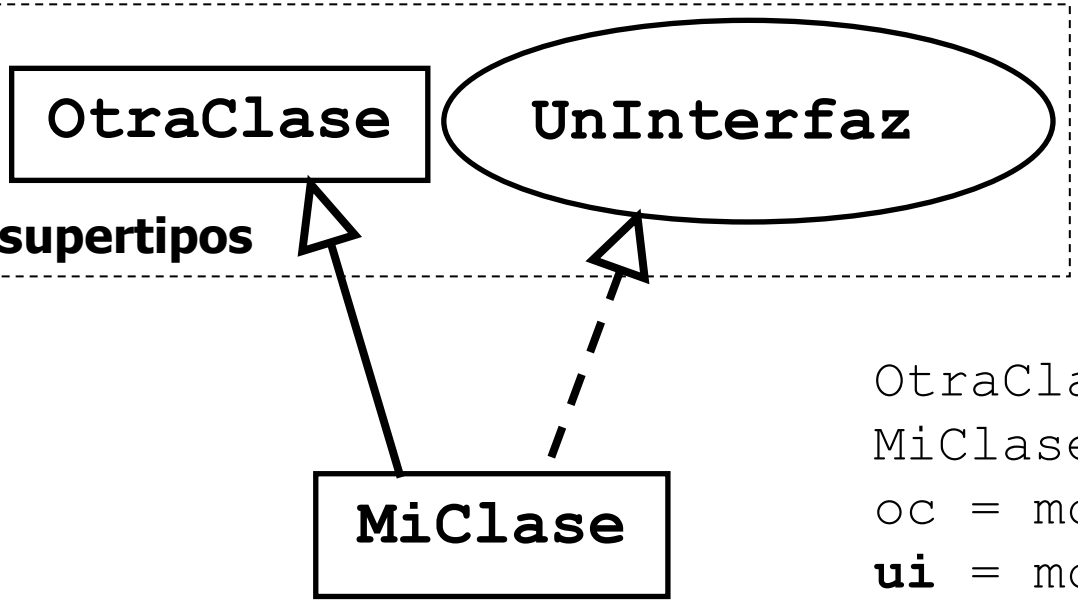
```
interface Pila<T> {
  void push(T elem);
  T pop();
  T top();
  boolean empty():
}
```



Varias implementaciones posibles:

```
class PilaFija<T> implements Pila<T> {…}
class PilaEnlazada<T> implements Pila<T> {…
```

Interfaces




```
OtraClase oc; UnInterfaz ui;
MiClase mc = new MiClase();
oc = mc;
ui = mc;
```

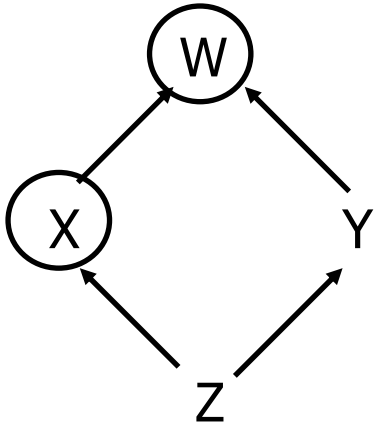
- Una interfaz puede utilizarse como nombre de tipo.
- `mc` incluye todos sus supertipos (clases e interfaces).
- A `ui` se le puede asignar cualquier objeto que implemente la interfaz.

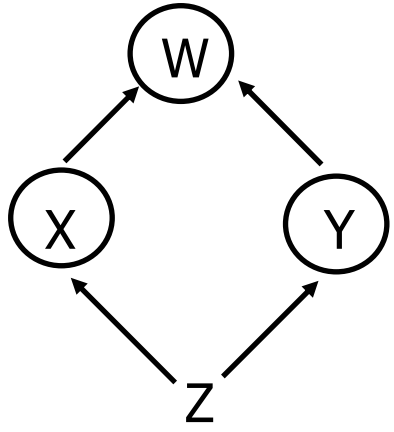
Colisión de nombres entre una interfaz y una clase

```
ublic class OtraClase {
  public void met(){
     //hace algo
   }


ublic interface UnInterface {
     void met();


ublic class MiClase extends OtraClase
                        implements UnInterface {
/La implementación del método met de la interface es la
/que se está heredando de la clase UnaClase porque
/coinciden las signaturas, si no habría sobrecarga
/da error si sólo difieren en el valor de retorno
```

Herencia de interfaces

- A diferencia de las clases una interfaz puede heredar de más de una interfaz
- Se utiliza herencia múltiple si se quiere que una clase implemente un interfaz y herede de otra clase






```
interface W { }
interface X extends W { }
class Y implements W { }
class Z extends Y implements X { }
```

```
interface W { }
interface X extends W { }
interface Y extends W { }
class Z implements X, Y { }
```

Colisión de nombres: herencia múltiple de interfaces

```
ublic interface Interfaz1 {
 int CTE = 1;
 void met();


ublic interface Interfaz2 {
 int CTE = 789;
 void met();


ublic class Clase implements Interfaz1, Interfaz2{

 public void met(){ //única semántica del método
   System.out.println("Única implementación de met");
   System.out.println("El valor de la cte es" +
 Interfaz1.CTE);
 }
```

Utilización de un interface

- Simplemente heredando de la clase que implementa el interfaz

- ¿Que hacemos si una clase hereda de otra y desea utilizar cierta implementación de una interface?

!NO PODEMOS HEREDAR DOS VECES!

- Podemos simular la herencia múltiple mediante delegación

Utilización de un interface

Solución: crear un objeto de la clase de implementación y remitirle los métodos de la interface

```
public class B extends A implements X {
  private Ximplem at1;
  …
  public void metodo1(…){
    at1.metodo1(…);
  }
  public int metodo2(…) {
    return at1.metodo2(…);
  }
}
public class Ximplem implements X {…}
```

Ejemplo: uso de interfaces 1/2

- Abstracción de una máquina que se mueve con motor

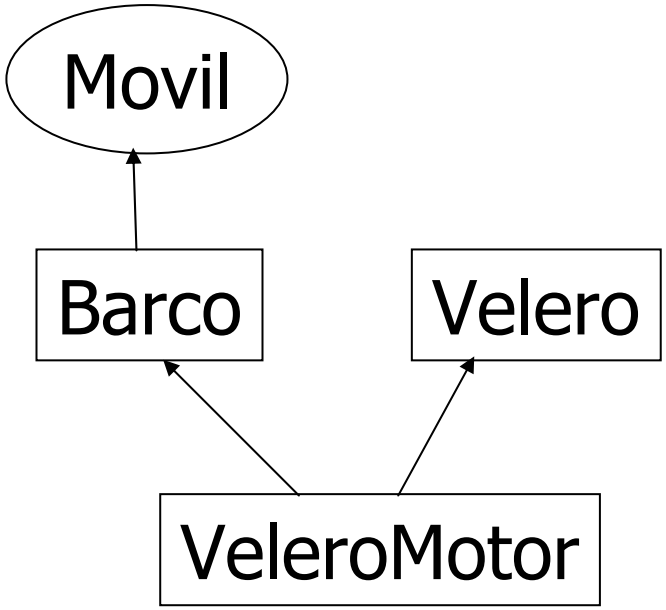
```
interface Movil{
   void enMarcha();
   void alto();
   void girar(double grados);
   void cambiaVelocidad(double kmHora);
   boolean quedaCombustible();
}
```

- Un Barco que se mueve a motor

```
public class Barco implements Movil { … }
```

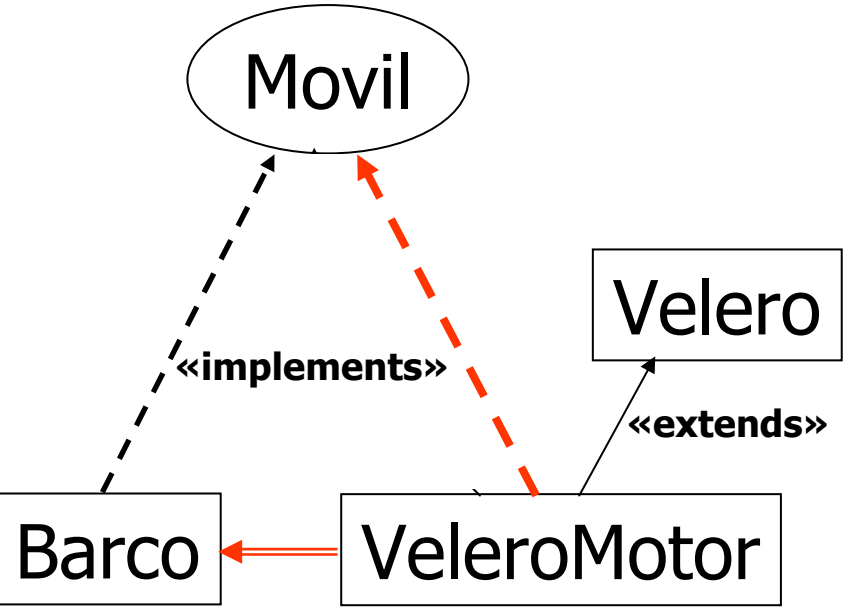
- Suponiendo que existe la clase `Velero` ¿Cómo modelaríamos un `VeleroMotor` que debe heredar de `Barco` y de `Velero`?

Ejemplo: uso de interfaces 2/2



¡¡¡No podemos heredar de dos clases!!!

Ejemplo: uso de interfaces 2/2




```
class VeleroMotor extends Velero
                        implements Movil{
    Barco b;

    void enMarcha(){
        b.enMarcha();
    }

    …

    boolean quedaCombustible(){
        return b.quedaCombustible();
    }
}
```

Conclusión: OO y Objetivos Iniciales

La combinación de **clases**, **genericidad**, **herencia**, **redefinición**, **polimorfismo**, **ligadura dinámica** y **clases diferidas** permite satisfacer:

- los principios/ reglas/ criterios de *modularidad*
- los requisitos para la *reutilización*

planteados en el primer tema.