TEMA 4

Herencia: Conceptos básicos

Facultad de Informática

Universidad de Murcia

Índice

1.-Introducción

Las clases no son suficientes para conseguir los objetivos de:

(A) **REUTILIZACIÓN**

Necesidad de mecanismos para generar **código genérico:**

- Capturar aspectos comunes en grupos de estructuras similares
- Independencia de la representación
- Variación en estructuras de datos y algoritmos

(B) **EXTENSIBILIDAD**

Necesidad de mecanismos para favorecer:

- "Principio abierto-cerrado" y "Principio Elección Única"
- Estructuras de datos polimórficas.

Introducción

- Entre algunas clases pueden existir relaciones conceptuales:

  Extensión, Especialización, Combinación

EJEMPLO:

"Libros y Revistas tienen propiedades comunes"
"Una pila puede definirse a partir de una cola o viceversa"
"Un rectángulo es una especialización de polígono"
"Una ventana de texto es un rectángulo dónde se manipula texto"

¿Tiene sentido crear una clase a partir de otra?

Herencia { soporte para registrar y utilizar estas relaciones
posibilita la definición de una clase a partir de otra

Introducción. Jerarquías de clases

La herencia organiza las clases en una estructura jerárquica:

Jerarquías de clases

Ejemplos:

- No es tan solo un mecanismo para compartir código.
- Consistente con el sistema de tipos del lenguaje

Introducción

- Puede existir una clase "raíz" en la jerarquía de la cual heredan las demás directa o indirectamente.
- Incluye todas las características comunes a todas las clases

Eiffel:	clase `ANY`
Java:	clase `Object`
C++:	no existe
C#:	clase `System.Object`

Introducción

Si *B* hereda de *A* entonces *B* incorpora la **estructura** (atributos) y **comportamiento** (métodos) de la clase *A*, pero puede incluir **adaptaciones**:

- -B puede **añadir** nuevos **atributos**
- -B puede **añadir** nuevos **métodos**
- -B puede **REDEFINIR métodos**
  - •Refinar: Extender el uso original
  - •Reemplazar: Mejorar la implementación
- -B puede **renombrar** atributos o métodos
- -B puede implementar un método diferido en A

...

Adaptaciones dependientes del lenguaje

(Redefinición disponible en cualquier LPOO)

El proceso de la herencia es transitivo

A

↑

B hereda de A

C hereda de B y A

B

↑

B y C son descendientes (subclases) de A

B es un descendiente directo de A

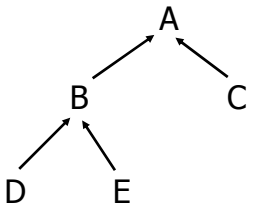
C es un descendiente indirecto de A

C

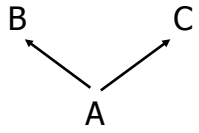
TERMINOLOGÍA

- B hereda de A
- B es descendiente de A (Eiffel)
- A es un ascendiente de B (Eiffel)
- B es subclase de A (Java)
- A es superclase de B (Java)
- B es una clase derivada de A (C++)
- A es la clase base de B (C++)

Tipos de herencia



- **Herencia simple**
  - Una clase puede heredar de una única clase.
  - Ejemplo: Java, C#



- **Herencia múltiple**
  - Una clase puede heredar de varias clases.
  - Clases forman un grafo dirigido acíclico
  - Ejemplos: Eiffel, C++

¿Cómo detectar la herencia durante el diseño?

- **Generalización (Factorización)**

  Se detectan clases con un comportamiento común (p.e. `Libro` y `Revista`)

- **Especialización (*Abstracción*)**

  Se detecta que una clase es un caso especial de otra (p.e. `Rectangulo` de `Poligono`)

No hay receta mágica para crear buenas jerarquías

Problemas con la evolución de la jerarquía

Ejemplo: Polígonos y Rectángulos

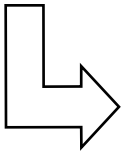
- Tenemos la clase **Poligono** y necesitamos representar rectángulos:

¿Debemos crear la clase **Rectangulo** partiendo de cero?

Podemos aprovechar la existencia de similitudes y particularidades entre ambas clases

Polígonos y Rectángulos

- Un rectángulo tiene muchas de las características de un polígono (*rotar*, *trasladar*, *vértices*,..)
- Pero tiene características especiales (*diagonal*) y propiedades especiales (*4 lados, ángulos rectos*)
- Algunas características de polígono pueden implementarse más eficientemente *(perímetro)*



```
class Rectangulo inherit
      Poligono
feature
      ...Características específicas
end
```

Clase Polígono 1/3

```
class POLIGONO creation ...
feature {NONE}              -- implementación
  vertices: LINKED_LIST [PUNTO];
      -- ptos sucesivos formando el polígono
feature                     -- acceso
  numero_vertices: INTEGER;
  rotar (centro:PUNTO; angulo:REAL) is do .. end;
  trasladar (a, b: REAL) is do .. end;
  visualizar is do .. end;
  perimetro : REAL is do .. end;
  …
invariant
  numero_vertices = vertices.count
  numero_vertices >=3
end -- class POLIGONO
```

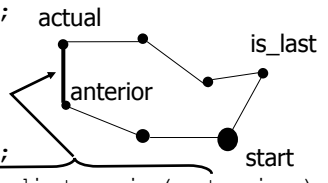

Clase Polígono 2/3

```
trasladar (a, b: REAL) is do
      -- desplaza a horizontal^te y b vertical^te
  from  vertices.start
  until vertices.after
  loop
        vertices.item.trasladar(a,b)
        vertices.forth
  end
end;

rotar (centro, angulo: REAL) is do
      -- rotar el angulo alrededor del centro
  from  vertices.start
  until vertices.after
  loop
        vertices.item.rotar(centro,angulo)
        vertices.forth
  end
end;
```

Clase Polígono 3/3

```
perimetro : REAL is
      -- suma de las longitudes de los lados
   local    actual, anterior : PUNTO
   do
      from  vertices.start;
            actual:= vertices.item ;
      until   vertices.is_last ;
      loop
       anterior := actual ;
       vertices.forth ;
       actual := vertices.item ;
       Result:= Result + actual.distancia(anterior);
      end;
   Result:=Result+actual.distancia(vertices.first);
end;
```



Clase Rectángulo

```
class RECTANGULO inherit POLIGONO
   redefine perimetro
creation crear
feature
   lado1, lado2 : REAL ;
   diagonal : REAL ;
   crear (centro:Punto; s1, s2, angulo:real) is
    do .. end;
   perimetro : REAL is do
    Result := 2 * ( lado1 + lado2 )
   end ;
end ;
```

Atributos nuevos

Todas las características de **Poligono** están disponibles automáticamente para la clase **Rectangulo**, no es necesario que se repitan.

Doble aspecto de la herencia

Clases	Herencia	
Modulo	Mecanismo de extensión	Reutilizar características
Tipo	Mecanismo de especialización	Clasificación de tipos

¿relación es-un?

Sean:

Prop(X) : Propiedades (atributos y métodos) de la clase X

dom(C) : Conjunto de instancias de una clase C

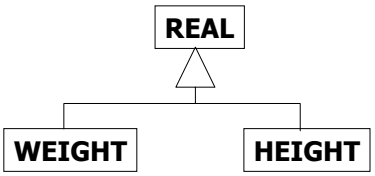
- **B extiende la clase A => Prop (A) ⊂ Prop (B)**
- Cualquier objeto de ***B*** puede ser considerado objeto de ***A***
- Siempre que se espera un objeto de ***A*** podemos recibir un objeto de ***B***, puesto que aceptaría todos sus mensajes
- **dom (B) ⊂ dom (A) =>B es un subtipo de A**

Herencia de Implementación vs. Herencia de Tipos

- No siempre se corresponden las clases con tipos
- Dos consideraciones:
  - ¿Cómo relacionamos los tipos?
    - **HERENCIA DE TIPOS o DE COMPORTAMIENTO**
    - Da lugar a jerarquías basadas en aspectos comunes
  - ¿Cómo organizamos las clases para reutilizar código?
    - **HERENCIA DE IMPLEMENTACIÓN o CÓDIGO**
    - Clases no relacionadas pero que tienen código similar
    - Las clases podrían parecer repositorios de código

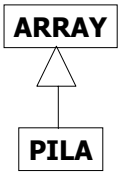
Ejemplos. Herencia de Tipos e Implementación

1) Coincide **herencia de tipos e implementación**

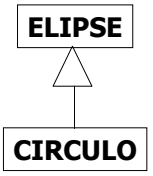


`Weight` y `Height` son tipos de medidas que tienen propiedades en común con los números reales

2) **Herencia de implementación**: todas las propiedades del padre pueden no aplicarse al hijo



3) **Herencia de comportamiento** (especializar un tipo)



¿Un mecanismo o más?

- "Esta división parece causar más daños que beneficios" [Meyer]:
  - Sólo dos categorías no es representativo
  - Discusiones metodológicas inútiles
  - Complejidad del lenguaje
    - Casi todos los mecanismos auxiliares (redefinición, renombramiento) son útiles en ambas visiones
- En **Eiffel** y **C#** existe un único mecanismo
- **C++** distingue entre herencia de tipos (`public`) y de implementación (`private`)
- **Java** también distingue entre herencia de tipos e implementación utilizando interfaces para los tipos y clases para implementación

2.- Polimorfismo

- El término **polimorfismo** significa que hay un nombre (variable, función o clase) y muchos significados diferentes (distintas definiciones).

- Formas de polimorfismo [Budd'02]:
  - Polimorfismo de asignación (*variables polimorfas*)
  - Polimorfismo puro (*función polimorfa*)
  - Polimorfismo ad hoc (*sobrecarga*)
  - Polimorfismo de inclusión (*redefinición*)
  - Polimorfismo paramétrico (*genericidad*)

Polimorfismo de asignación y puro

Capacidad de una entidad de referenciar en tiempo de ejecución a instancias de diferentes clases.

- **Es restringido por la herencia**
- Importante para escribir código genérico
- Sea las declaraciones:

```
ox: X; rutina1 (oy:Y)
```

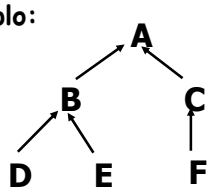
  - En un lenguaje con **monomorfismo** (Pascal, Ada, ..) en t.e. **ox** y **oy** denotarán valores de los tipos **X** e **Y**, respectivamente.
  - En un lenguaje con **polimorfismo** (Eiffel, C++, ..) en t.e. **ox** y **oy** podrán estar asociados a objetos de varios tipos diferentes:

tipo estático vs. tipo dinámico

Tipo estático y tipo dinámico

- **Tipo estático:**
  - Tipo asociado en la declaración
- **Tipo dinámico:**
  - Tipo correspondiente a la clase del objeto conectado a la entidad en tiempo de ejecución
- **Conjunto de tipos dinámicos:**
  - Conjunto de posibles tipos dinámicos de una entidad

Ejemplo:

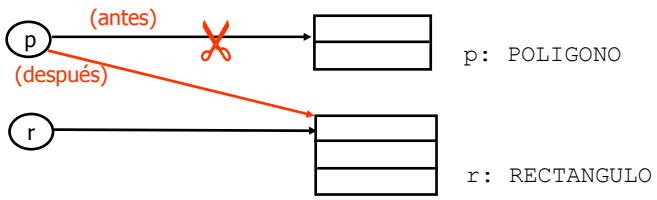


oa: A; ob: B; oc: C;

te(oa) = A	ctd(oa) = {A,B,C,D,E,F}
te(ob) = B	ctd(ob) = {B, D, E}
te(oc) = C	ctd(oc) = {C,F}

Entidades y rutinas polimorfas

Conexión polimorfa: el origen y el destino tiene tipos diferentes



a) asignación:

```
p := r;
```

-- p es una **entidad polimorfa**
(polimorfismo de asignación)

b) paso de parámetros:

```
f (p:POLIGONO) is do
...
end
```

-- f es una **rutina polimorfa**
(polimorfismo puro)

- **Sólo** se permite para entidades destino de **tipo referencia**

Polimorfismo puro vs. Sobrecarga

- Funciones sobrecargadas ≠ funciones polimórficas
- Sobrecarga:
  - Dos o mas funciones comparten el nombre y distintos argumentos (nº y tipo). **El nombre es polimórfico**.
  - Distintas definiciones y tipos (distintos comportamientos)
  - Función correcta se determina en **tiempo de compilación** según la signatura.
- Funciones polimórficas:
  - La función correcta se determina dinámicamente en **tiempo de ejecución.**
  - Una única definición y todos los tipos son subtipos del tipo principal (comportamiento uniforme).

Sobrecarga en C++ y Java

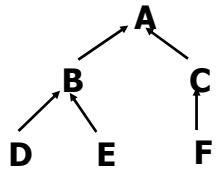
- En los lenguajes OO puede existir sobrecarga
  - dentro de una clase: C++ y Java
  - entre clases no relacionadas (es fundamental)
- En C++ existe sobrecarga si dos funciones se encuentran definidas en el mismo ámbito (clase)

```
class B{
    public void f(int v);
}
class D: B{
    public void f(float v);
}
```

- f no está sobrecargada en la clase `D`
- la `f` de `D` oculta a `f` de `B`

- En Java si estarían disponibles las dos funciones en la clase `D`

3.- Herencia y sistema de tipos




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

¿Son legales las siguientes asignaciones?

```
oa:= ob; oc:= ob; oa:= od
```

¿Es legal el mensaje `od.metodo1`?

NOTA:

- **Descendiente propio** de una clase:
  - Todas las clases que heredan de ella directa o indirectamente
  - Ej.: B, C, D E y F son descendientes propios de A
- **Descendiente** de una clase:
  - La clase y todos sus descendientes propios
  - Ej.: Los descendientes de A incluyen a A

Herencia y Sistema de Tipos

Un lenguaje OO tiene **comprobación estática de tipos** si está equipado con un cjto de **reglas de consistencia**, cuyo cumplimiento es controlado por los **compiladores**, y que si el código de un sistema las cumple se garantiza que ninguna ejecución de dicho sistema puede provocar una violación de tipos

- **Política pesimista**:

"al tratar de garantizar que ninguna operación fallará, el compilador puede rechazar código que tenga sentido en tiempo de ejecución"

Ejemplo: (Pascal) n:INTEGER; r: REAL ⇒ n:=r 💣

n:= 0.0	Podría funcionar
n:= -3.67	No funcionaría
n:= 3.67 - 3.67	Funcionaría

- Beneficios esperados: Fiabilidad, legibilidad y eficiencia

Reglas básicas

- La herencia es consistente con el sistema de tipos

Regla de llamada a una característica

En una llamada a una característica **`x.f`** donde el tipo de `x` se basa en una clase `C`, la característica `f` debe estar definida en uno de los antecesores de `C`.

Luego, sean las declaraciones

```
p : POLIGONO ; r : RECTANGULO
```

Mensajes legales:	**Mensajes ilegales:**

`p.perimetro; p.vertices ;`
`p.trasladar(..); p.rotar (..);`
`r.diagonal; r.lado1; r.lado2;`
`r.vertices; r.trasladar(..);`
`r.rotar (..); r.perimetro;`
	`p.lado1;p.lado2;`
	`p.diagonal;`
	 

Reglas básicas

- La herencia regula que conexiones polimorfas son permitidas

Definición: compatibilidad o conformidad de tipos

Un tipo **`U`** es compatible o conforme con un tipo **`T`** sólo si la clase base de `U` es un descendiente de la clase base de `T`, además, para los tipos derivados genéricamente, todo parámetro real de `U` debe (recursivamente) ser compatible con el correspondiente parámetro formal en `T`.

Es decir:

1) Ejemplo: `Rectangulo` es compatible con `Poligono`

2) `B[Y]` será compatible con `A[X]` si:

- `B` es descendiente de `A`
- `Y` es descendiente de `X`

Reglas básicas

Regla de compatibilidad de tipos

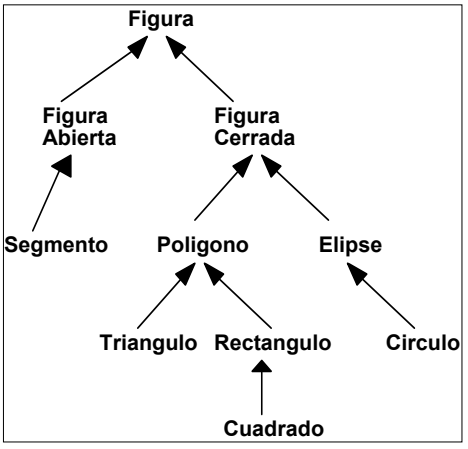
Una conexión con origen **y** y destino **x** (esto es, una asignación **x:=y**, o invocación **r(..,y,..)** a una rutina **r(.., x:T,..)** es válido solamente si el tipo de **y** es compatible con el tipo de **x**.

Regla de validez de un mensaje

Un mensaje **ox.rut (y),** supuesta la declaración **ox: X,** será legal si i) **X** incluye una propiedad con nombre final **rut**, ii) los argumentos son compatibles con los parámetros y coinciden en número, y iii) **rut** está disponible para la clase que incluye el mensaje.

Ejemplo

```
p: POLIGONO; r: RECTANGULO; t: TRIANGULO;... x: REAL
```



SERÍA CORRECTO EL CODIGO

```
x:=p.perimetro;
x:=r.perimetro;
x:=r.diagonal;
if "algún test" then p:=r
else p:=t
end ;
x:=p.perimetro;
```

SERÍA INCORRECTO

```
x:=p.diagonal;
r:=p;
```

¿Están las restricciones justificadas?

- ¿Tiene sentido que el compilador rechace el siguiente código:

  c1) `p:= r; r:= p`
  c2) `p:= r; x:= p.diagonal ?`

- Situaciones poco probables (sin sentido).

- Asignaciones como **`p:=r`** son normalmente ejecutadas como parte de alguna estructura de control que dependa de condiciones "run-time", tal como la entrada de un usuario.

- Esquemas más realistas son:

A)
```
p: POLIGONO; r: RECTANGULO;  t: TRIANGULO;... x: REAL;
  !!r.make ; !!t.make; ..
  pantalla.visualizar_iconos;
  pantalla.esperar_click_raton ;
  x := pantalla.posicion_raton;
  icono_elegido := pantalla.icono_donde_esta_es(x);
  if icono_elegido = icono_rectangulo then  p:=r
  elsif  … p:=t  …
  elsif  … p:="alguna otra clase de polígono"
      ...
  end;
  ...
```
{... Usos de **p**, **`p.visualizar`**; **`p.rotar(..)`** ; ...}

B)
```
una_rutina (p:POLIGONO) is do ... end
```

- En ambas situaciones, donde se desconoce el tipo exacto de `p`:
  - sólo tiene sentido aplicar operaciones generales de `POLIGONO`.
  - es cuando tiene sentido el uso de entidades polimorfas como `p`

4.- Genericidad y Polimorfismo

- **Estructuras de datos polimorfas**:

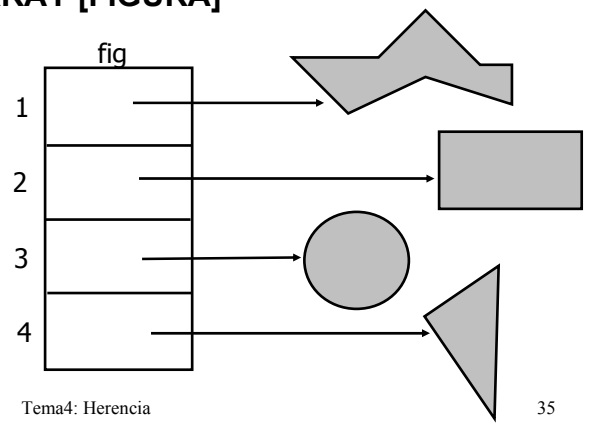
  Pueden contener instancias de una jerarquía de clases

- Ejemplo: **fig: ARRAY [FIGURA]**

p: POLIGONO;
r: RECTANGULO;
c: CIRCULO;
t:TRIANGULO;

!!p; !!r; !!c; !!t;

fig.put (p,1);
fig.put (r,2);
fig.put (c,3);
fig.put (t,4); ...



Intento de asignación

- La **Regla de Compatibilidad de Tipos** asume que tenemos toda la información sobre el tipo de los objetos
- Esta información se puede perder:
  - Procede de estructuras de datos polimorfas
  - Llegada del objeto desde el exterior (archivo, red)
- Hace falta una forma de **averiguar el tipo** de los objetos evitando estructuras "CASE"

```
if "f es de tipo RECTANGULO" then
      …
elseif "f es de tipo CIRCULO" then
      ...
```

- **Abierto-Cerrado**
- **Elección Única**

Solución: Intento de asignación

- **Sintaxis:**

```
ox:X ; oy:Y
ox ?= oy
```

- **Semántica:**

  1) Es legal si **X** es compatible con **Y**

  2) Se realiza la asignación en tiempo de ejecución si el tipo dinámico de **oy** es compatible con el tipo estático de **ox**, **X**.

  3) **ox** tendrá valor `void` si no se realiza la asignación

Ejemplo de intento de asignación.

Encontrar las mayor diagonal de los elementos de una lista

```
max_diagonal (lista_fig: LIST[FIGURA]): REAL is
  require lista_fig /= void
  local r: RECTANGULO
  do
   from lista_fig.start; Result:= -1.0
   until  lista_fig.after
   loop
      r?= lista_fig.item;
      if r /= void then Result:= Result.max(r.diagonal)
    end;
      lista_fig.forth
  end
end
```

En Java!!!

Ejemplo de intento de asignación. Java

```
public float maxDiagonal (LinkedList listaFig) {
   if (listaFig == null) throw new IllegalArgumentException();
   Rectangulo r;
   float actual,result=-1;
   for (int index=0;index<listaFig.size();++index){
    try{
       r= (Rectangulo)lista_fig.get(i);
       actual = r.getDiagonal();
       if (actual>result) result=actual;
    }catch (ClassCastException e){ }
   }
   return result;
}
```


Ejemplo de intento de asignación. Java

```
public float maxDiagonal (LinkedList listaFig) {
   if (listaFig == null) throw new IllegalArgumentException();
   Figura f;
   float actual,result=-1;
   for (int index=0;index<listaFig.size();++index){
        f = (Figura)lista_fig.get(i);
        if (f instanceof Rectangulo){
             actual = (Rectangulo)f.getDiagonal();
             if (actual>result) result = actual;
        }
    }
    return result;
}
```


¿Qué ocurre si quiero sumar vectores?

```
class VECTOR [G] feature
    count: INTEGER;
    item, infix "@" (i: INTEGER) : G is do .. end;
    put(v: G, i:INTEGER) is do .. end;
    infix "+" (other: VECTOR[G]): VECTOR[G] is
        require count = other.count
        local i: INTEGER
        do
            from i:=1 until i> count loop
                Result. put ( item(i) + other.item(i) , i)
                i:= i + 1
            end
        end
    ....
end
```

¿Es posible ampliar el nº de operaciones?

Solución: Genericidad Restringida

- Es posible restringir las clases a las que se puede instanciar el parámetro génerico formal de una clase genérica.

  class C [G -> R]

- Ejemplos:

  VECTOR [G - > NUMERIC]
  DICTIONARY [G, H - > COMPARABLE]
  ORDENACION [G - > COMPARABLE]

- Sólo es posible instanciar G con descendientes de la clase R.
- Las operaciones permitidas sobre entidades de tipo G son aquellas permitidas sobre una entidad de tipo R.

Genericidad Restringida

Ejemplo: **VECTOR [G -> NUMERIC]**

será posible aplicar operaciones de la clase **NUMERIC** sobre entidades de tipo **G**

- La genericidad no restringida equivale a **[G -> ANY]**
- ¿Sería legal la declaración **x: VECTOR [VECTOR [NUMERIC]]** ?
- ¿Es lo mismo **VECTOR[NUMERIC]** que **VECTOR[G ->NUMERIC]**?

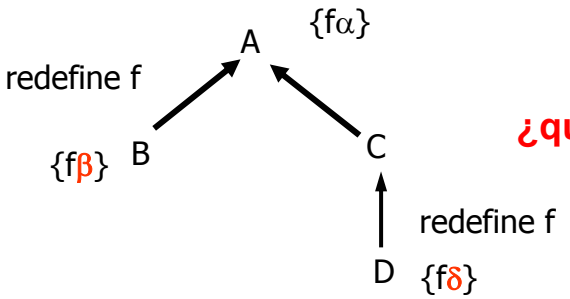
Genericidad restringida Eiffel vs. Java

- ¿Cuál es la diferencia, si la hay?

Eiffel	Java

class Vector[**G->Numeric**]
feature {NONE}
  contenido: ARRAY[G]
 	
feature
  insertar(n:**G**, i:INTEGER)is
  do
  ...
  end
  ...
end
	public class Vector{
	
	private **Numeric**[] contenido;
	
	public void insertar(**Numeric** n,
	  Integer i)
	{
	  ...
	}
	...
	}

5.- Ligadura dinámica




```
oa: A;
oa.f(..);
```

¿qué versión de f se ejecuta?

Regla de la ligadura dinámica

La forma dinámica del objeto determina la versión de la operación que se aplicará.

Ligadura dinámica

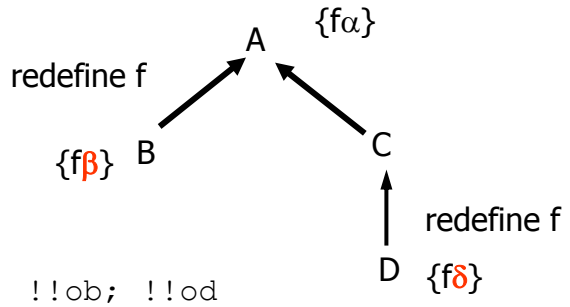
- La **versión** de una rutina en una clase es la introducida por la clase (redefinición u original) o la heredada.

- **Ejemplo 1:**

  p: POLIGONO; r: RECTANGULO;

  !!p; x:= p.perimetro → $perimetro_{POLIGONO}$
  !!r; x:= r.perimetro → $perimetro_{RECTANGULO}$
  !!r; p:= r; x:= p.perimetro → $perimetro_{RECTANGULO}$

Ejercicio: ¿qué versión se ejecuta?




```
oa: A; !!ob; !!od
oa:=ob;
oa.f;

oa:=od
oa.f;
```


Ligadura Dinámica

Ejemplo 2: visualizar (fig:Array[Figura]) **is do**

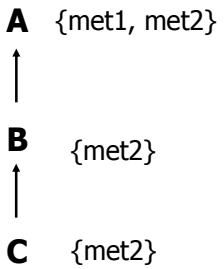
```
        from i:=1 until i= fig.size loop
                fig.item(i).dibujar
        end
end
```

¿Qué sucede si aparece un nuevo tipo de figura?

- ¿Qué relación existe entre ligadura dinámica y comprobación estática de tipos?

Sea el mensaje **x.f ()**, la **comprobación estática de tipos** garantiza que al menos existirá una versión aplicable para **f,** y la **ligadura dinámica** garantiza que se ejecutará la versión más apropiada

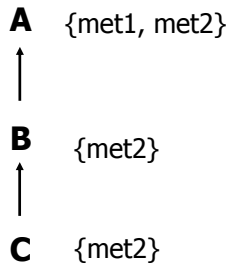
Patrones para conseguir código genérico




```
1) met1 is do
        ....
        current.met2(..)
                -- current es una entidad polimorfa
        ...
   end

2) met3 (p: A) is do
        ....
        p.met2(..)
        ....
   end
```


Patrones para conseguir código genérico




```
met1(p:A) is do
        ....
    current.met2()
    p.met2(..)
    ...
end

met3 (p: ARRAY[A]) is do
    from i:=1 until i=p.size loop
        p.item(i).met2(..)
    end
    ..
end
```

Un mismo código con diferentes implementaciones:

"variación en la implementación"

* Ligadura dinámica en Java

- Ligadura dinámica como política.
- Si utilizamos una referencia de superclase para referirnos a un objeto de subclase el programa escogerá el método de la subclase correcta en tiempo de ejecución (= dinámicamente)

* Ligadura dinámica en C++

- Por defecto la **ligadura es estática**. Como optimización!!!
- Es necesario declarar **explícitamente** sobre qué funciones se aplicará ligadura dinámica.

```
class A {                    class B: public A {        A *oa;  B *ob;
   public:                      public:                 ob = new B;
   virtual void f ();           void f ();              oa = ob;
... }                           ...}                    oa -> f ();
```


Ligadura dinámica y eficiencia

- **¿Tiene un coste de ejecución inaceptable?**

  `x.f(a,b,c,…)`

  - la característica `f` depende del tipo del objeto al que esté conectado `x`
  - el tipo de `x` no se puede predecir a partir del texto software
- Lenguajes SIN comprobación estática de tipos
  - Si `f` no está en el tipo del objeto al que está conectado `x` se busca en el padre recursivamente
  - Se puede tener que recorrer todo el camino hasta la raíz
  - la penalización **es** IMPREDECIBLE
  - crece con la profundidad de la estructura de herencia
  - conflicto entre reutilización y eficiencia

Ligadura dinámica y eficiencia

- Lenguajes **CON comprobación estática de tipos**
  - Los tipos posibles para `x` controlados por la herencia
  - se reduce a los descendientes del tipo estático de `x`
  - El compilador puede preparar una estructura basada en arrays que contenga la información de tipos necesaria
  - El coste de la ligadura dinámica es **CONSTANTE**
    - cálculo de índice y acceso a un array
  - No hay que preocuparse de eficiencia y reutilización
  - Cierto tanto para herencia simple como múltiple

6.- Clases diferidas

Sea la declaración

```
f: FIGURA;   p: POLIGONO
```

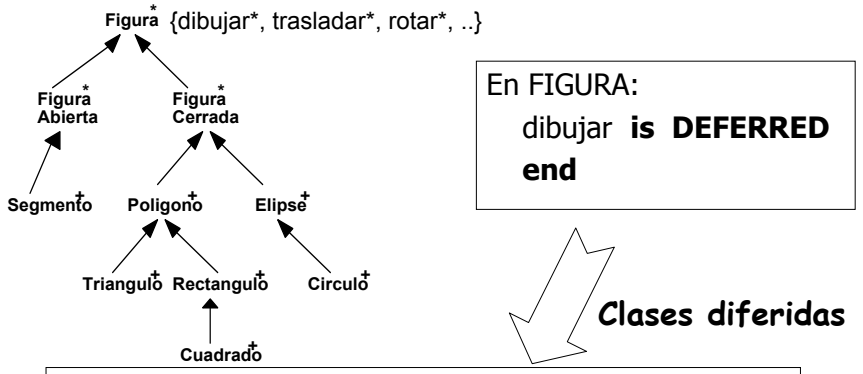
y el código

```
!!p;
f:=p;
f.dibujar
```

¿Sería legal?

- ¿Cómo se implementa **`dibujar`** en la clase **`FIGURA`**?
- La rutina **`dibujar`** no puede ser implementada en **`FIGURA`** pero **`f.dibujar`** es **¡dinámicamente correcto!**
- ¿Tendría sentido incluir **`dibujar`** en **`FIGURA`** como una rutina que no hace nada?

Solución: Rutinas diferidas



En FIGURA:
dibujar **is DEFERRED**
end

Clases diferidas

```
deferred class FIGURA feature ... end
```

- Una subclase de una clase diferida puede seguir siendo diferida

```
deferred class FIGURA_ABIERTA feature ... end
deferred class FIGURA_CERRADA feature ... end
```


Clases diferidas

- Contiene rutinas diferidas que deben ser implementadas en sus subclases.
- Especifica una **funcionalidad que es común** a un conjunto de subclases aunque no es completa.
- Puede ser total o parcialmente diferida.
- **No es posible crear instancias** de una clase diferida, pero si declarar entidades de estas clases.
- En Eiffel, C++ y Java son un elemento del lenguaje, pero no en Smalltalk.

* Clases abstractas en C++

- Una clase con funciones virtuales se puede convertir en abstracta al declarar como **pura** una de sus funciones virtuales:

virtual void f ( ) = 0;

Ejemplo:

```
class Figura {
    public:
        virtual void dibujar ( ) = 0;
        virtual void trasladar (...) = 0;
        ...
}
```


* Clases abstractas en Java

- Hacemos abstracta una clase declarándola con la palabra clave **abstract.**
- Son clases muy generales de las que no se desean instanciar objetos.
- Toda clase que contenga un método abstracto (heredado o no) será abstracta.

Ejemplo:

```
abstract class Figura {
      ...
      abstract void dibujar();
      ...
}
```

Clases parcialmente diferidas

- Contienen rutinas diferidas y efectivas.
- Las rutinas efectivas pueden hacer uso de las diferidas.
- Importante mecanismo para incluir **código genérico**.
- Incluyen comportamiento abstracto común a todos los descendientes.

"programs with holes"

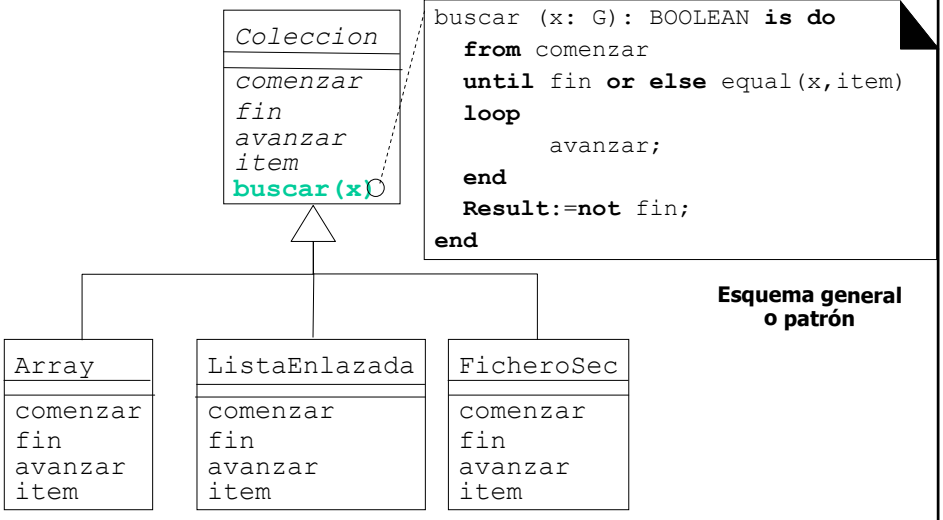
Clases parcialmente diferidas

- "Permiten **capturar lo conocido** sobre el comportamiento y estructuras de datos que caracterizan a cierta área de aplicación, **dejando una puerta abierta a la variación**:

  SON UNA CONTRIBUCION IMPORTANTE DE LA OO A LA REUTILIZACIÓN" [B. Meyer]

- Reciben el nombre de **CLASES COMPORTAMIENTO** aquellas clases que incluyen un **comportamiento común** a varias subclases ("familia de implementaciones de TAD´s")

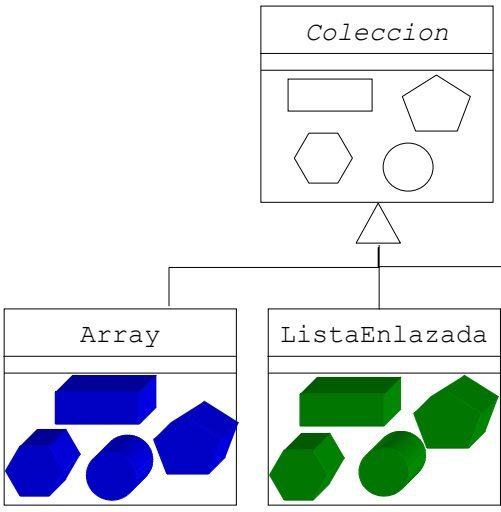
Clases parcialmente diferidas



Método plantilla - Clase comportamiento

```
deferred class Coleccion[G] feature
  buscar (x: G): BOOLEAN is do
     from comenzar
     until fin or else equals(x, item)
     loop
        avanzar;
     end
     Result:=not fin;
   end
   …
end;
```

"No nos llame, nosotros le llamaremos"




```
a:Array[Integer]
!!a
a.buscar(31)
```

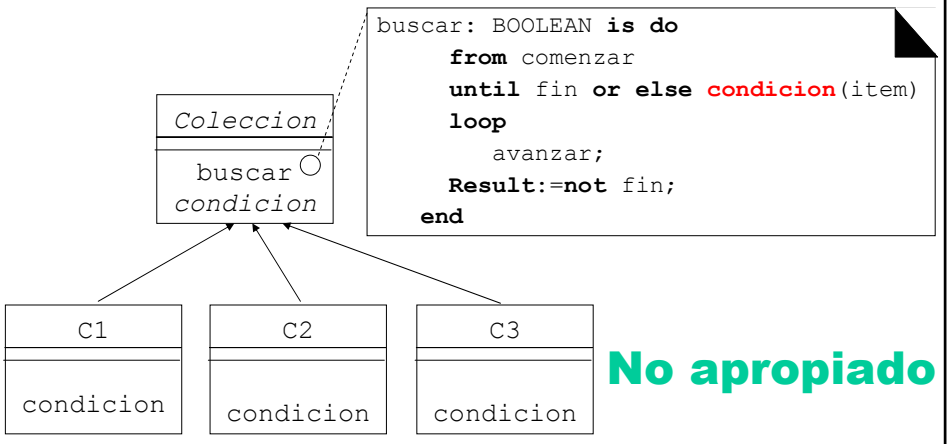

Parametrizar una rutina con una acción

- ¿Cómo podemos buscar si existe un elemento que cumple una condición?

  ```
  buscar(condicion)
  ```

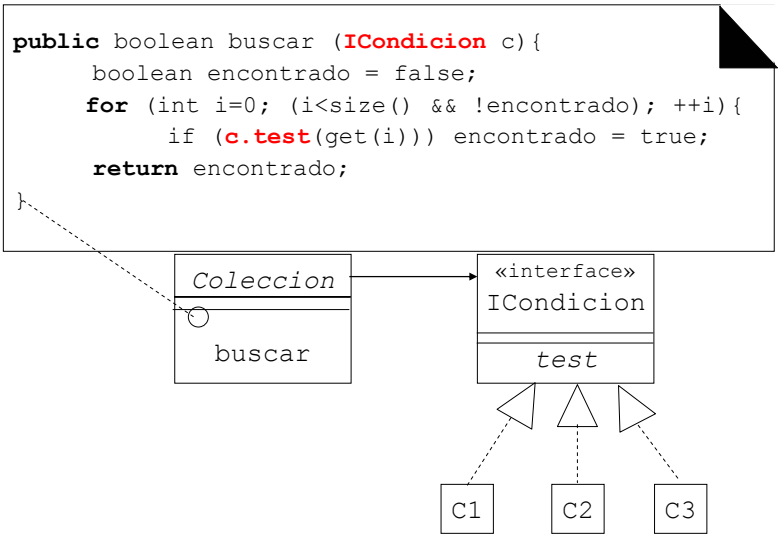
- Dos soluciones:
  a) Herencia → Método plantilla
  b) Composición

Parametrizar una rutina con una acción.
a) Método plantilla



No apropiado

Parametrizar una rutina con una acción.
b) Composición

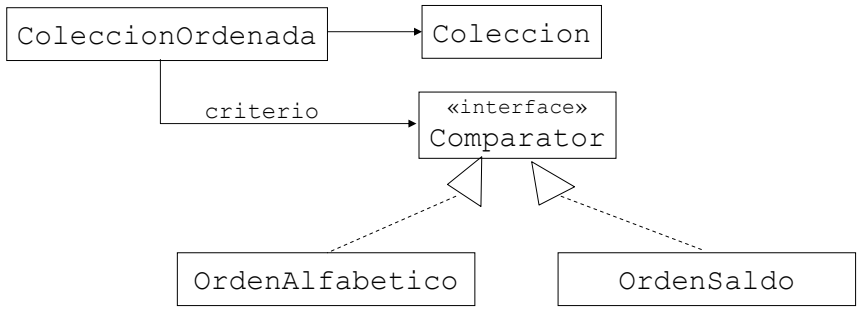


Parametrizar una rutina con una acción. b) Composición

```
class ExisteNombre implements ICondicion{
  private String nombre;
  public ExisteNombre(String n){
      nombre = n;
  }
  public boolean test (Object obj){
  return nombre.equals((Contacto)obj.getNombre());
  }
}

      boolean resp; List agenda=new LinkedList();
        …
      resp = agenda.buscar(new ExisteNombre("Yago"));
```


Ejemplo composición. ColeccionOrdenada




```
ColeccionOrdenada cuentas;
cuentas = new ColeccionOrdenada (new OrdenAlfabetico());
```

- **Ejercicio**: implementar **add** en **ColeccionOrdenada.**

Ejemplo herencia. Iteradores internos

"Iterar" significa ejecutar un cierto procedimiento (**accion**) sobre todos los elementos de una estructura de datos (**coleccion**) o sobre aquellos que cumplan una condición (**test**).

```
lc: LISTA [CUENTA]
from lc.comenzar
until lc.fin
loop
    cta:= lc.item
    if cta.numeros_rojos then
            print(cta.titular)
    end
    lc.avanzar
end;
```

```
forEach(accion)
do_if(test, accion)
```


Iteradores internos

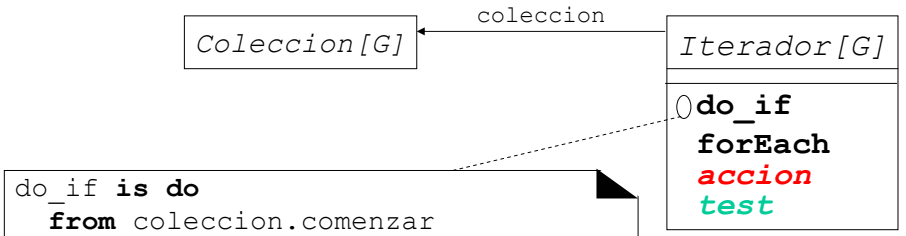
- Interesa capturar "**patrones o esquemas de recorrido** de estructuras de datos": reutilizar en vez de escribir de nuevo.
- Un sistema que haga uso de un mecanismo general para iterar debe ser capaz de aplicarlo para cualquier **accion** y **test** de su elección.
- El método de iteración debe estar parametrizado por la acción y la condición.
- En Java y Eiffel no es posible pasar una rutina como argumento.

Implementación de Iteradores Internos

A. Definir los métodos de iteración en la clase `Coleccion`
   → **NO**
   - Una iteración es una propiedad del cliente, no de la colección
   - Deberíamos crear descendientes de las clases que representan colecciones para crear diferentes esquemas de iteración.

B. Implementar la ***clase comportamiento*** `Iterador` →**SI**
   - Representa objetos con capacidad para iterar sobre colecciones.

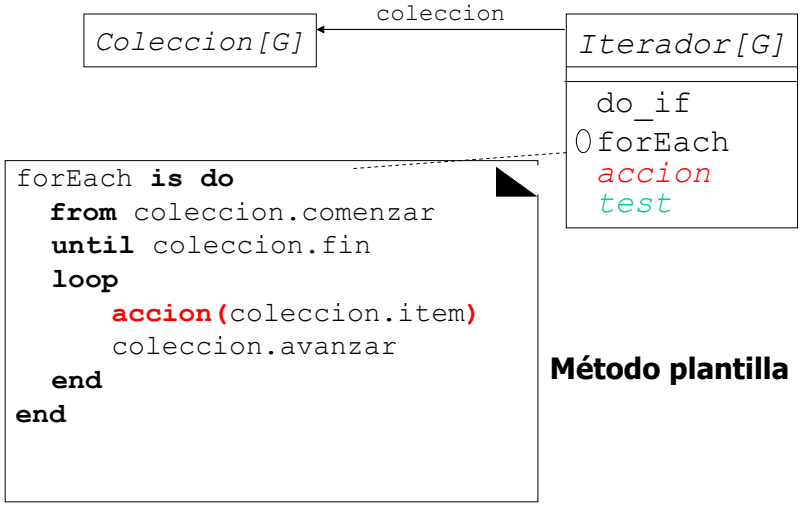
Iteradores internos




```
do_if is do
  from coleccion.comenzar
  until coleccion.fin
  loop
      if test(coleccion.item) then
            accion(colección.item)
      end
      coleccion.avanzar
  end
end
```

Método plantilla

Iteradores internos

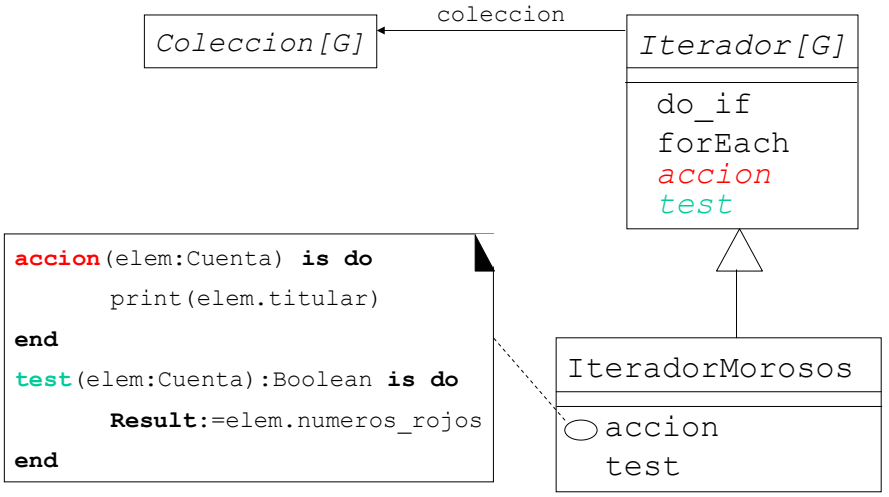


Método plantilla

Iteradores en Eiffel

```
deferred class LINEAR_ITERATOR [G] creation  make
feature
   coleccion: LINEAR_LIST [G]
   make (c: LINEAR_LIST [G]) is do
         coleccion:= c;
   end;
   action (v:G) is deferred end;
   test (v: G): BOOLEAN is deferred end;
   forEach is do     -- Ejecutar la acción sobre todos los elementos
         from coleccion.start  until coleccion. after  loop
                  action (coleccion.item)
                  coleccion. forth
         end
   end;
   do_if is do          -- Ejecutar la acción en los item que cumplan test
         from coleccion.start until coleccion.after loop
                  if test(coleccion.item) then accion(coleccion.item) end
                  coleccion.forth
         end
   end
end  - - LINEAR_ITERATOR[G]
```

Iteradores internos



Iteradores internos

```
cuentas:Lista[Cuenta];
iter: IteradorMorosos[Cuenta];
!!cuentas;
…
!!iter.make(cuentas);
iter.do_if;
```

- Siendo `make` el método de creación de la clase `Iterador[G]` que recibe como argumento la colección lineal sobre la que se va a iterar.

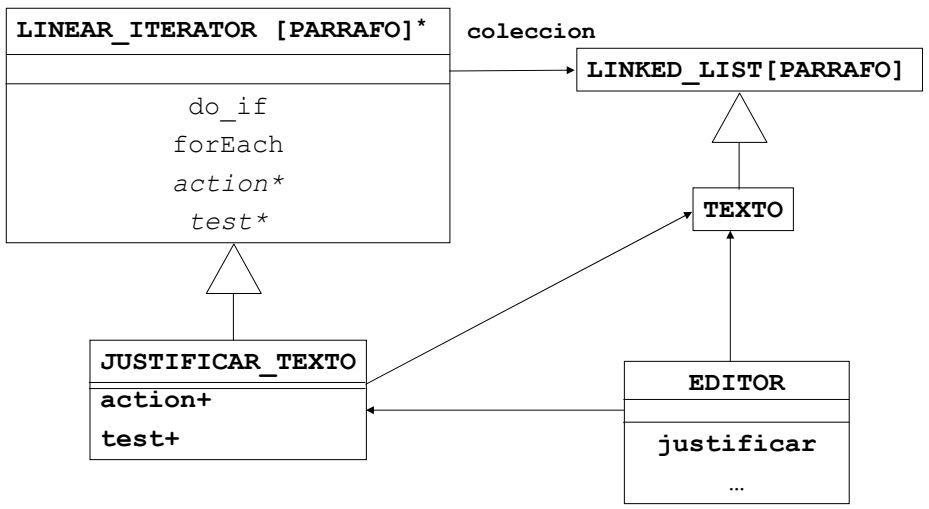
Iteradores en Eiffel

```
class IteradorSuma inherit LINEAR_ITERATOR [G]
     creation make
     feature {APLICACION}
       suma: INTEGER;
       action (v: INTEGER) is do
              suma:= suma + v
       end;
end
```

```
class APLICACION creation
  feature {NONE}
      listaEnteros: LINKED_LIST [INTEGER];

  feature
      sumarTodo: INTEGER is
      local  iterador: IteradorSuma;
      do
            !!iterador.make(listaEnteros);
            iterador.forEach
            Result:=iterador.suma
      end;
end
```


Ejemplo Iterador: Editor de texto (1/3)



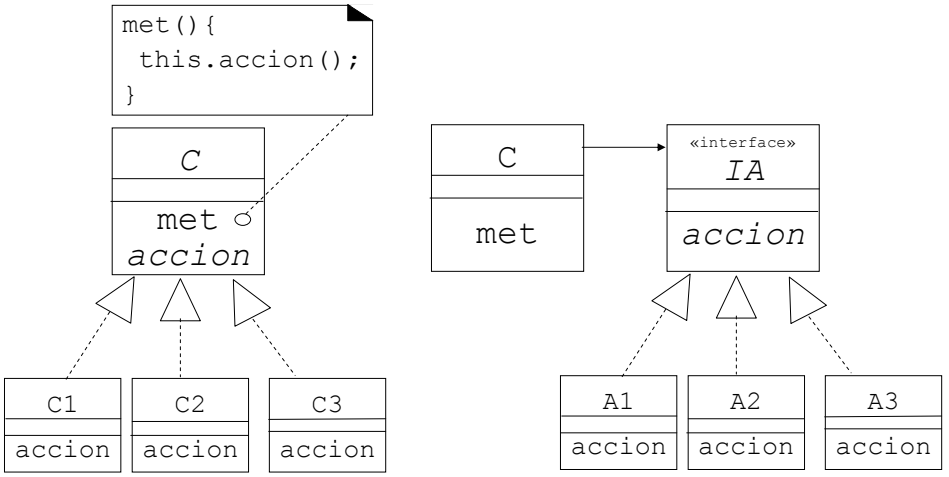
Ejemplo Iterador: Editor de texto (2/3)

```
class Justificar_Texto                    -- implementa un iterador
   inherit LINEAR_ITERATOR [PARRAFO]
   feature {EDITOR}
       justificarTodo (t: TEXT) is do
              make (t)      -- establece que se itera sobre t
              do_if         -- se cambia tamaño de cada párrafo
                            -- que se ha modificado
       end
   feature {NONE}
       test (p: PARRAFO): BOOLEAN is do
              Result:= p.sinFormato
       end;
       action (p: PARRAFO) is do
              p.justificar
       end
end
```


Ejemplo Iterador: Editor de texto (3/3)

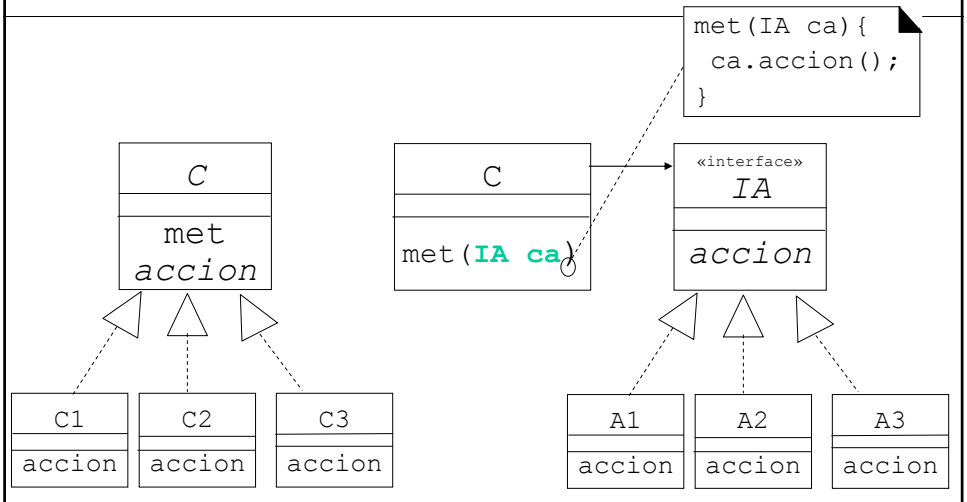
```
class Editor feature
  text: Texto
  …
  justificar is
  local jt: Justificar_Texto
  do
    !!jt
    jt.justificarTodo(text)
  end
end
```

Esquemas para parametrizar una rutina con una acción
→ `met(accion)`



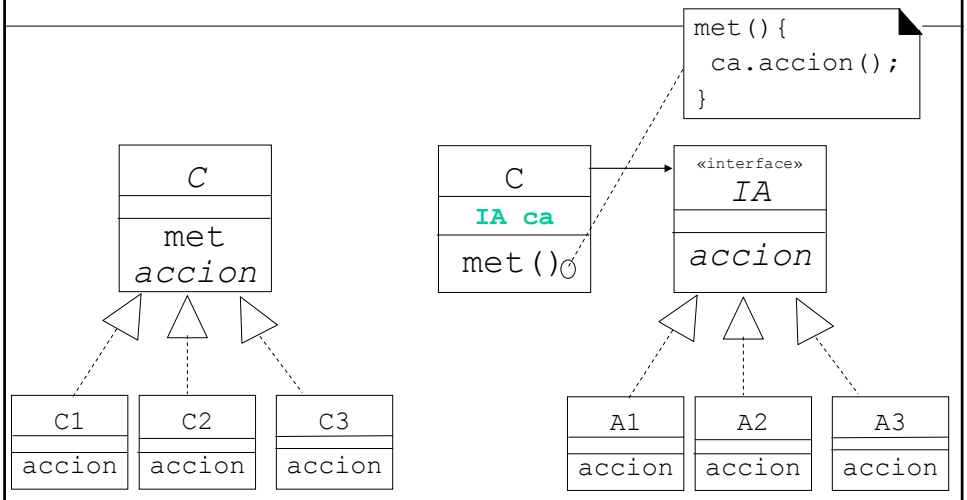
a) Método plantilla **b) Composición**

Esquemas para parametrizar una rutina con una acción
→ `met(accion)`



a) Método plantilla **b) Composición**

Esquemas para parametrizar una rutina con una acción
→ `met(accion)`



a) Método plantilla **b) Composición**

¿Quién controla la iteración?

- Iterador externo:
  - el cliente es el que controla la iteración.
  - el cliente es el que avanza en el recorrido y pide al iterador el siguiente elemento.
  - Ejemplo: `Iterator` de Java.
- Iterador interno:
  - el iterador es quien controla la iteración.
  - Es el iterador el que aplica una operación a cada elemento de la colección.
  - Ejemplo: clase `Linear_Iterator` de Eiffel.

Ejemplo: imprimir los objetos de una colección

```
public void imprimir(){
  Iterator it = coleccion.iterator();
   while (it.hasNext())
     IO.imprimir(it.next());
}
```

Iterador externo

```
public void imprimir(){
  IteratorImprimir it =
          new IteratorImprimir(coleccion);
  it.forEach();
}
```

Iterador interno

IteratorImprimir

```
public class IteratorImprimir extends
                              LinealIterator{

 public IteratorImprimir(List c){
     super(c);
 }
 public void accion (Object obj){
     IO.imprimir(obj);
 }
}
```

LinealIterator

```
/** Iterador sobre una colección secuencial */
public abstract class LinealIterator{
   private List coleccion;
   public LinealIterator(List c){
     coleccion=c;
    }
   public void forEach(){
    for (int i=0; i<coleccion.size(); ++i)
       accion(coleccion.get(i));
  }
  public abstract void accion(Object obj);
}
```


7.- Redefinición de características

- Reconcilia la **reutilización** con la **extensibilidad**:
  "Es raro reutilizar un componente software sin necesidad de cambios"
- Una clase hija puede "anular" un método de la clase padre por dos motivos:
  - **Reemplazo:**
    - Mejorar implementación. Ej: redefinir perímetro en la clase Rectangulo.
    - Otra diferente (aunque con la misma semántica). Ej: el método dibujar en la jerarquía de Figura.
  - **Refinamiento:**
    - Método del padre + acciones específicas. Ej. métodos de inicialización

Redefinición en Eiffel

- **¿Qué podemos cambiar?**
  - Tipo de las características (Regla covariante)
  - Pre y post- condiciones
  - Implementación **JAVA y C++**
- Solicita explícitamente cláusula **redefine**.
- No es necesario incluirlo cuando se redefine para hacer efectiva una característica.
- Los cuatro posibles casos se resumen en :

A \ DE	Diferido	Efectivo
Diferido	Redefinir	Indefinir
Efectivo	Hacer efectiva	Redefinir

REDECLARAR

Técnicas de Redeclaración

- Redeclarar una característica es: Redefinirla o hacerla efectiva
- Proporciona un estilo de desarrollo flexible e incremental
- **Técnicas que le añade potencia**:
  a) Redeclarar una función como un atributo
     - Aplicación del Principio de Acceso Uniforme
     - Un atributo no se puede redefinir como una función
  b) El uso de la versión original en una redeclaración
     - "Facilita la tarea al que redefine si el nuevo trabajo incluye el viejo".

a) Redefinir una función como un atributo

- Ejemplo1:

```
class Cuenta1 feature
    saldo: INTEGER is do
    Result:= ingresos.total -reintegros.total
    end
...
end

class Cuenta2 inherit
    Cuenta1 redefine saldo end
feature
    saldo:INTEGER
...
end
```

- Ejemplo 2: Hacer efectiva la función diferida de lista que calcula el número de elementos `count` en un atributo

b) Uso de la versión original en Eiffel

- Entidad **Precursor =** llamar a la versión de esta característica en la clase padre.
- Su uso **está limitado** al cuerpo de las rutinas redefinidas
- **Ejemplo:** Dibujar un botón es mostrarlo como una ventana y luego dibujar el borde


```
dibujar is do ... end

dibujar is do
        Precursor
        dibujar_borde
end
```

Redefinición en C++

- Clase base contiene una función virtual vf
- Clase que se deriva de ella contiene una función vf del mismo tipo (si son distintos se consideran funciones diferentes y no se invoca al mecanismo virtual)

```
class Base {
    virtual void vf1();    //Ligadura dinámica
    virtual void vf2();
    virtual void vf3();
    void f();              //Ligadura estática
};
class Derivada : public Base {
    void vf1();            //redefine vf1
    void vf2 (int);        //distinta a vf2()
    char vf3();            //error, difiere tipo devuelto
    void f();
};
```

```
void g()
{
Derivada *d;
Base* b;
        ...
b = d;
b -> vf1();        //Derivada::vf1
b -> vf2();        //Base::vf2
b -> f();          //Base::f
d-> vf2();         //ERROR
        ...
}
```

- Llamada al padre calificando las rutinas. Ej: **Base :: f()**

Redefinición en Java

- El método que tiene la misma signatura que un método de la superclase anulará siempre, automáticamente, el método de la superclase (no hay que indicarlo explícitamente).
- Para referenciar a la versión del padre ¿**super** = precursor?

```
public class Punto {
   ...
 public String toString(){
  return "[" + x+ "," + y + "]";
 }
   ...
}

public class Circulo extends Punto {
                    ...
 public String toString(){
   return "Centro=" + super.toString() + " ;Radio =" + radio;
 }
}
```

Siempre el padre
NO HAY
LIGADURA DINÁMICA

8.- Herencia y creación en Eiffel

Regla de creación en la herencia

En Eiffel el status de creación de una característica heredada de la clase padre no tiene que ver con su status de creación en la clase heredada.

- Si la subclase añade atributos también tienen que inicializarse.
- La subclase puede reforzar el invariante.
- Si se puede aplicar el procedimiento de creación del padre se indica en la clausula **`creation`**.
- Instrucciones de creación básicas: **`!!x`**

  `!!x.crear`

Ejemplo: Herencia y creación en Eiffel

- **Creación polimorfa:** Se debe poder permitir la creación directa de objetos de un tipo descendiente

  `!C!x.crear`

```
fig: FIGURA; tipo_icono_sel: INTEGER; p1, p2, p3: PUNTO;

tipo_icono_sel:= interface. icon_selected (mouse_position);
inspect tipo_icono_sel
   when segmento_icono  then !SEGMENTO! fig. make (p1,p2)
   when circulo_icono   then !CIRCULO! fig. make (p1,radio)
   when triangulo_icono then !TRIANGULO! fig. make (p1,p2,p3)
   ...
end;
       fig. visualizar
```

* Herencia y Creación en C++

- Un constructor de clase derivada siempre llamará primero al constructor de su clase base para inicializar los miembros de su clase padre.
- **Ejemplo:**

```
//Punto.cpp (x,y)
    ...
Punto::Punto (float a, float b){  //constructor
    x=a; x=b;
}

//Circulo.cpp;
class Circulo: public Punto {
    …
Circulo::Circulo(float r, float a, float b):Punto(a,b)
//llamada al constructor de la clase base
{
    radio=r;
}
```


* Herencia y creación en Java

- Se puede incluir una llamada **explícita** al constructor de la clase padre como primer enunciado del constructor de la subclase
- En otro caso, se llamará **implícitamente** al constructor por omisión.

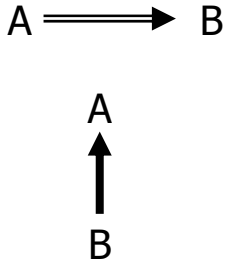
Ejemplo://Punto.java

```
public class Punto{
  …
  public Punto(double a, double b){  //constructor
    setPunto(a,b);
  }
  ...
}

public class Circulo extends Punto{
  …
  public Circulo(double r, double a, double b){
    super(a,b);   //llamada al constructor del padre
    setRadio(r);
  }
  ...
}
```


9.-Herencia y Ocultación de Información

- La herencia expone los detalles de implementación a las subclases:
  - **"La herencia rompe la ocultación de información"**
- **Relaciones entre módulos:**



Clientela (reutilización de "caja negra")

El autor de cada clase es responsable de dar permisos de acceso a los clientes

Herencia (reutilización de "caja blanca")

¿Qué pasa con el status de exportación cuando se pasa a los descendientes?

Herencia y Ocultación de Información en Eiffel

- Son mecanismos independientes
- Una clase B es libre de exportar o de esconder una característica f que hereda de un antecesor A.

f exportada en A y B

f privada en A y B

f exportada en A y oculta en B

f oculta en A y exportada en B

- Por defecto f mantiene el status de exportación que tenía en A, pero se puede cambiar mediante la clausula **export**.

¿Por qué dejar que cada descendiente escoja la política de exportación en Eiffel?

- Flexibilidad y Extensibilidad:
  - *La herencia es la clave del Principio Abierto-Cerrado*
- La propiedad básica de la herencia es permitir definir descendientes de una clase no previstos en su creación.
- ¿Cómo sabemos a priori que propiedades exportará o no una subclase no prevista?
- La herencia sólo debe estar limitada por los asertos y las reglas de tipado.

* Herencia y Ocultación de Información en C++

- **protected**:
  - sólo accesibles por la clase y **sus descendientes**
- Tipos de herencia:
  - **Herencia privada:** `class B: private A {...}`
    - todas las propiedades de **A** se heredan como privadas
    - se puede **mantener** el estatus calificando la rutina
  - **Herencia pública:** `class B: public A {...}`
    - se mantiene el status de las propiedades heredadas (por defecto)
    - un miembro `private` o `protected` no puede ser re-exportado, es decir, `protected` no puede ser `public` y `private` no puede ser ni `protected` ni `public`
    - "agujero" debido asignaciones polimórficas (público puede pasar a privado)

Herencia y exportación en C++

Se puede conservar el acceso si la herencia es privada, mencionando su nombre calificado en la parte `public` o `protected`. Por ejemplo:

```
class B{
  int a; //private
public:
  int bf();
protected:
  int c;
};
class D: private B{
public:
  B::bf();   //hace que bf sea público en D
  B::a;      //ERROR intenta conceder acceso
protected:
  B::bf();   //ERROR intenta reducir acceso
};
```

Todos los miembros de B son privados en D

* Herencia y Ocultación de Información en Java

- La herencia en Java no cambia el nivel de protección de los miembros de la clase base.
- No se puede especificar herencia pública o privada como en C++.
- La clase padre puede cambiar el acceso de los miembros de la superclase, pero sólo si proporciona más acceso:
  - public -> public
  - protected -> protected / public
- Hacer un miembro más restrictivo no tiene sentido puesto que podrías saltarte la restricción a través de las asignaciones polimórficas.

10.- Herencia y Aserciones. (Eiffel)

A) Invariante de clase

A $\{INV_A\}$

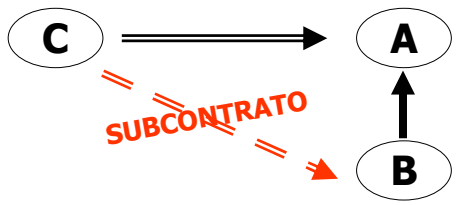
B $\{INV_B = INV_A$ and "restricciones para B"$\}$

"Siempre que espero una instancia de la clase A puedo recibir un objeto de la clase B"

B) PRE Y POST Condiciones. Subcontrato

La redefinición debe cambiar la implementación de una rutina, no su semántica

B) PRE Y POST Condiciones. Subcontrato



a:A;
...
a.r ó
if a.α then a.r

¿Si a tiene t.d. B?

r **is**
require α
...
ensure β
end

r^{++} **is**
require γ
...
ensure δ
end

La nueva versión (**r^{++}**):

- **acepta todas** las llamadas que aceptaba la original (pre igual o más débil)
- garantiza **al menos** lo que garantizaba la otra (post igual o más fuerte)

Herencia y assert en Java

- Recordemos que la facilidad de asertos de Java 1.4 **NO** proporciona el soporte suficiente para la técnica de Diseño por Contrato.
- **Carece de**:
  - la capacidad de distinguir entre los distintos tipos de asertos
  - **un mecanismo para permitir la herencia de asertos en las subclases**.

11.- Herencia y excepciones en Java

- Si redefinimos un método en la subclase, éste no puede lanzar más excepciones comprobadas que el método de la superclase que está redefiniendo.
- El método de la subclase puede lanzar menos.
- Si el método de la superclase no lanza ninguna excepción comprobada, tampoco puede hacerlo el método redefinido de la subclase.