Departamento de Informática | Universidad de Valladolid

Programación III.I.T.I. de Sistemas

Introducción a la herencia

Félix Prieto

Curso 2008/09

Contenidos

- Contenidos
  - Definiciones
  - Jerarquías de herencia
  - Polimorfismo
  - Tipo estático y dinámico
  - Ligadura dinámica
  - Clases y métodos diferidos
  - Problemas asociados al polimorfismo
  - Genericidad restringida

Una clase ancestro

```
class POLIGONO
create ...
feature -- Acceso
 contador: INTEGER
   -- Número de vértices
 perimetro: DOUBLE is
   -- Suma de las longitudes de
   -- los lados
  do ... end
 muestra is
   -- Representación en pantalla
  do ... end
feature -- Modificación
 rota (c:PUNTO;a: DOUBLE) is
   -- en torno a 'c' con ángulo 'a'
  do ... end
 traslada (a,b:DOUBLE) is
   -- 'a' en horizontal y
   -- 'b' en vertical
  do ... end
feature {NONE}-- Implementación
 vertices: LINKED_LIST[POINT]
   -- Puntos que lo definen
end -- Class POLIGONO
```


Una clase descendiente

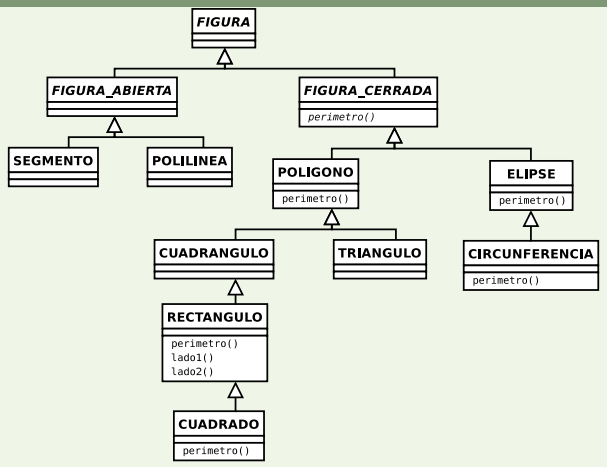
```
class RECTANGULO
inherit POLIGONO
 redefine perimetro
 end
create
 make
feature -- Creación
 make(c:PUNTO;s1,s2,a:DOUBLE) is
   -- Creación con centro 'c', lados
   -- 's1' y 's2' y ángulo 'a' con
   -- respecto a la horizontal
  do ... end
```

```
feature -- Acceso
 lado1, lado2 : DOUBLE
 diagonal : DOUBLE
   -- Longitudes específicas
 perimetro: DOUBLE is
   -- Suma de las longitudes de los
   -- lados
  do
   Result:= lado1*2 + lado2*2
  end
end -- Class RECTANGULO
```

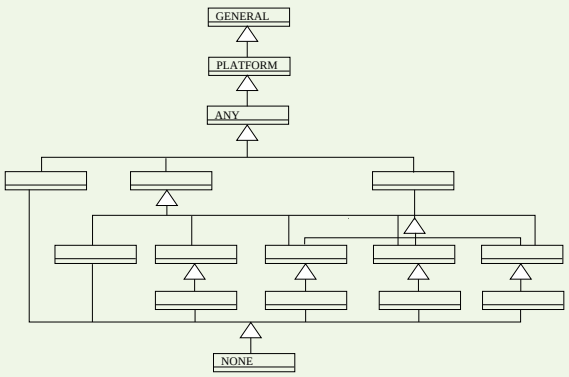

Primeras definiciones

- Llamaremos clase *descendiente* de C a:
  - la propia clase C
  - cualquier clase con una clausula `inherit` D donde D es descendiente de C
- Diremos que B es *descendiente propio* de C si es descendiente suyo y ambas son distintas
- También utilizaremos expresiones como *ancestro*
- Los descendientes disponen de todas las características de sus ancestros
- Los descendientes pueden redefinir características de sus ancestros (`redefine`), aunque con ciertas restricciones (`frozen`, compatibilidad entre interfaces, programación bajo contrato)
- La condición de método de creación en los ancestros no se mantiene en sus descendientes

Jerarquía de herencia de las figuras



Jerarquía de herencia de SmartEiffel



Polimorfismo

- *Polimorfismo* es la habilidad de *las referencias* para estar conectadas a objetos de tipos distintos
- El número de tipos a que puede estar conectada una referencia está limitado por las reglas que se deducen de la herencia
- Estas limitaciones pueden ser vistas desde distintos enfoques

Límites al polimorfismo (mensajes)

- En una llamada `x.f` donde el tipo de `x` está basado en una clase `C` debe ocurrir que `f` sea característica de uno de los ancestros de `C`
- Las restricciones al polimorfismo deben garantizar que al ejecutar un código como `x.f` el objeto referenciado por `x` sea capaz de responder al mensaje `f`
- Tampoco podemos mandar al objeto mensajes que no entendería `x`, esto es, los declarados en sus descendientes
- Al «disfrazar» un objeto mediante polimorfismo perdemos su tipo real. Podemos tratar de recuperarlo mediante un *intento de asignación* (`?=`)
- *`x:C` sólo puede usarse para referenciar a objetos creados a partir de una clase descendiente de `C`*

Límites al polimorfismo (conformidad)

- En caso de tipos *no genéricos* un tipo `T` *es conforme* con un tipo `U` si y sólo si la clase base de `T` es descendiente de la clase base de `U`
- En caso de tipos *genéricos* un tipo `T` *es conforme* con un tipo `U` si y sólo si la clase base de `T` es descendiente de la clase base de `U` y cada parámetro actual utilizado para derivar `T` es conforme con el correspondiente parámetro utilizado para derivar `U`
- *Una conexión de fuente `x` y objetivo `y` es válida si y sólo si el tipo de `x` es conforme con el tipo de `y`*

Límites al polimorfismo (instancias)

- Una *instancia directa* de una clase `C` es un objeto conseguido en tiempo de ejecución mediante una clausula de creación `create x`, con `x` de tipo `C`
- Una *instancia* de `C` es un objeto instancia directa de algún descendiente de `C`
- *Una entidad de tipo basado en `C` sólo puede ser conectada en tiempo de ejecución a instancias de `C`*

Tipo estático y dinámico

- Las *entidades*, identificadores en el texto de una clase, tienen tipo *estático* y *dinámico*. El primero es fijo mientras el segundo puede variar en tiempo de ejecución
- Las *referencias*, que sólo existen en tiempo de ejecución, tienen el tipo *dinámico* del objeto al que están conectadas (o `NONE` si la referencia apunta a `Void`). El tipo de las referencias es variable
- Los *objetos* apuntados por las referencias tienen sólo tipo dinámico, pero es el tipo con el que fueron creados y no puede cambiar

Formas de ligadura

- En el código de un método pueden aparecer llamadas como `p.perimetro` con `p:POLIGONO`
- Disponemos de dos definiciones del método, una en `POLIGONO` y otra en `RECTANGULO`
- Gracias al *polimorfismo* el objeto que recibe el mensaje puede ser de cualquiera de los dos tipos
- ¿Cuál de las dos versiones del método se ejecutará?
  - *ligadura estática:* La asociada al tipo estático de `x`
  - *ligadura dinámica:* La asociada al tipo dinámico de `x`
- Cada lenguaje etermina el tipo de ligadura que será utilizada
- En Eiffel se utiliza por defecto la *ligadura dinámica*
- Forzamos la ligadura estática utilizando `frozen`

Ligadura dinámica

- En una llamada `x.f` la versión del método realmente ejecutada se determina en tiempo de ejecución
- La versión del método elegida es la definida en la clase base del objeto que recibe la llamada
- Podemos «imaginar» que el objeto responde ejecutando el único método que conoce
- Pero es evidente que al compilar se almacenan en el ejecutable todas las versiones disponibles del método, para su posterior selección en tiempo de ejecución

Transformación sobre figuras

```
cambia is
  -- En el código de FIGURA
do
  rota(....)
  traslada(...)
end
-- o alternativamente
cambia is
  -- En el código de una
  -- clase con f:FIGURA
do
  f.rota(....)
  f.traslada(...)
end
```

- El código se basa en la «promesa» de que todas las figuras tendrán una implementación de `rota` y `traslada`
- Necesitamos una garantía «sintáctica» de que estas definiciones siempre estarán presentes
- Podemos escribir definiciones «sin código» en `FIGURA`, pero eso no garantiza que todos los descendientes redefinan los métodos con alguna implementación efectiva

Definiciones diferidas

```
-- En la clase figura
rota (c:PUNTO;a: DOUBLE) is
  -- en torno a 'c' con ángulo 'a'
deferred
end
traslada (a,b:DOUBLE) is
  -- 'a' en horizontal y
  -- 'b' en vertical
deferred
end
```

- Podemos añadir a `FIGURA` definiciones *diferidas* de `rota` y `traslada`
- Las definiciones diferidas determinan cómo será la interfaz de las características en los descendientes
- La *programación bajo contrato* nos permitirá fijar desde el ancestro parte del comportamiento de las características en los descendientes

Clases diferidas

- Diremos que una clase es *diferida* cuando alguna de sus características lo es, en caso contrario es *efectiva*
- La definición de las clases diferidas comienza obligatoriamente por `deferred Class` (incluso cuando todas sus características diferidas provengan de sus ancestros)
- Un tipo es diferido si está basado en una clase diferida
- El tipo de una instrucción de creación no puede ser diferido
- Aunque las entidades pueden ser de un tipo diferido, los objetos siempre son de un tipo efectivo
- En los diagramas las clases y características diferidas se identifican con letra cursiva

Sintáxis de la redeclaración

Tipos de redeclaración (declaración en `inherit`)		
	a diferida	a efectiva
de diferida	redefinición (`redefine...end`)	concreción ninguna
de efectiva	indefinición (`undefine...end`)	redefinición (`redefine...end`)
`export {<c>}` para cambiar la exportación		
`rename` y `select` para herencia repetida		
`Precursor` para ejecutar el código del padre		

Eliminando alternativas múltiples

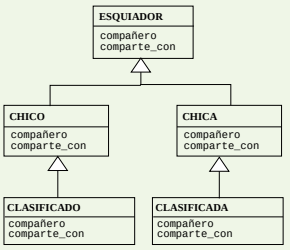
- Deseamos escribir una fecha en determinado idioma, variable en tiempo de ejecución
- Podemos encapsular el código de cada idioma en una alternativa múltiple
- Podemos utilizar polimorfismo, clases diferidas y ligadura dinámica para obtener el mismo efecto (ver `TIME_IN_SOME_LANGUAGE` y `TIME_IN_SPANISH`)
- Con esta solución minimizamos las operaciones de mantenimiento necesarias para introducir un nuevo idioma
- El responsable de la creación de cada objeto sabe con qué tipo lo hace, pero ni siquiera es seguro que un sólo objeto conozca todos los tipos posibles

Abreviaturas en la creación

```
local
   x,y:T
   a,b:U
do
   ...
   -- nueva sintáxis
   create a
   x:=a
   -- sintáxis tradicional
   !!b
   y:=b
   ...
end
```

```
local
   x,y:T
do
   ...
   -- nueva sintáxis
   create {U} x
   -- sintáxis tradicional
   !U!y
   ...
end
-- En ambos casos U debe
-- ser conforme con T
```


Dificultades con el polimorfismo



- El polimorfismo relaja el de chequeo de tipos
- Existen situaciones en que eso provoca errores en tiempo de ejecución
- Gestionamos el alojamiento de esquiadores en una competición
  - Los esquiadores comparten habitación con un compañero del mismo sexo
  - Los esquiadores clasificados disponen de habitación individual

Primer ejemplo problemático

```
class ESQUIADOR
feature
   compannero: ESQUIADOR
      -- Su compañero de habitación.
   comparte_con(otro:ESQUIADOR) is
      -- Fija el compañero del esquiador
      do
         compannero := otro
      end
   -- Otras características que no
   -- nos interesan para el ejemplo
end -- class ESQUIADOR
```


Primer ejemplo problemático(II)

```
class CHICO
inherit
   ESQUIADOR
      redefine
         compannero,
         comparte_con
      end
feature
   compannero: CHICO
   comparte_con(otro:CHICO) is
         -- Redefinimos la interfaz
      do
         compannero:=otro
         -- ¿ Podemos usar
         -- Precursor(otro) ?
      end
   -- Para distinguirlo
   distinto: INTEGER
end -- class CHICO
```

```
class CHICA
inherit
   ESQUIADOR
      redefine
         compannero,
         comparte_con
      end
feature
   compannero: CHICA
   comparte_con(otro:CHICA) is
         -- Redefinimos la interfaz
      do
         compannero:=otro
         -- ¿ Podemos usar
         -- Precursor(otro)?
      end
   -- para distinguirla
   distinto:REAL
end -- class CHICA
```


Primer ejemplo problemático(III)

```
class APLICACION
   -- Ejemplo de los problemas de la covarianza
creation
   make
feature
   make is
      -- Esquiva las limitaciones entre chicos y chicas
   local
      s: ESQUIADOR; b: CHICO ; g: CHICA
   do
      create b ; create g
      s:= b -- Disfrazo al chico
      s.comparte_con(g) -- no provoca error
      std_output.put_integer(b.compannero.distinto) -- error
   end
end -- class APLICACION
```

Interludio

- Al redefinir la interfaz de un método podemos sustituir un tipo por uno conforme con él
- Como hemos visto en el ejemplo anterior, esta redefinición no necesariamente evita errores en tiempo de ejecución
- Alternativamente podemos *anclar* el tipo de una entidad al de otra mediante expresión `like` *<tipo>* (`like` *Current* incluso)
- Con anclas no evitaremos el problema del ejemplo anterior
- Los tipos anclados son sólo conformes con su correspondiente ancla

Segundo ejemplo problemático

```
class ESQUIADOR
feature
  compannero: like Current
    -- Ahora anclamos el atributo
  comparte_con(otro:like Current) is
    -- Ahora anclamos el argumento
    do
      compannero := otro
    end
  -- Otras características que no
  -- nos interesan para el ejemplo
end -- class ESQUIADOR
```


Segundo ejemplo problemático(II)

```
class CHICO
inherit
  ESQUIADOR
feature
  -- Gracias a las anclas no
  -- necesitamos redefiniciones
  distinto: INTEGER
  -- Otras características que no
  -- nos interesan para el ejemplo
end -- class CHICO
```

```
class CHICA
inherit
  ESQUIADOR
feature
  -- Gracias a las anclas no
  -- necesitamos redefiniciones
  distinto: REAL
  -- Otras características que no
  -- nos interesan para el ejemplo
end -- class CHICA
```


Segundo ejemplo problemático(III)

```
class APLICACION
  -- Ejemplo de los problemas de la covarianza
creation
  make
feature
  make is
    -- Esquiva las limitaciones entre chicos y chicas
  local
    s: ESQUIADOR; b: CHICO ; g: CHICA
  do
    create b ; create g
    s:= b -- Disfrazo al chico
    s.comparte_con(g) -- no provoca error
    std_output.put_integer(b.compannero.distinto) -- error
  end
end -- class APLICACION
```


Interludio (II)

- El polimorfismo también puede evitar la detección de problemas en tiempo de compilación
- El nivel de exportación de las características impide a ciertos clientes solicitar determinadas tareas. El error sería detectado en tiempo de compilación
- Pero si un cliente tiene acceso legítimo a una característica, todos sus descendientes pueden «ser disfrazados» para conseguir el acceso,... al menos en tiempo de compilación
- Algunos compiladores avisan de esta situación mediante un Warning (SmartEiffel 2.3)

Tercer ejemplo problemático

```
class ESQUIADOR
feature
  compannero: like Current
    -- Anclamos el atributo
  comparte_con(otro:like Current) is
    -- Anclamos el argumento
    do
      compannero := otro
    end
  -- Otras características que no
  -- nos interesan para el ejemplo
end -- class ESQUIADOR
```


Tercer ejemplo problemático(II)

```
class CHICO
inherit
  ESQUIADOR
feature
  -- Gracias a las anclas no
  -- necesitamos redefiniciones
  -- Otras características que no
  -- nos interesan para el ejemplo
end -- class CHICO
```

```
class CLASIFICADO
inherit
  CHICO
  -- Pero nadie puede pedirle
  -- que comparta habitación
    export {NONE}
      comparte_con
    end
end -- class CLASIFICADO
```


Tercer ejemplo problemático(III)

```
class APLICACION
  -- Ejemplo de los problemas de la covarianza
creation
  make
feature
  make is
    -- Ejemplo del error
  local
    s: CHICO; b1,b2: CLASIFICADO
  do
    create b1 ; create b2
    s:= b1 -- Disfrazo al chico
    s.comparte_con(b2) -- no provoca error
    -- Pero ahora b1 comparte el cuarto
  end
end -- class APLICACION
```

Genericidad restringida

- En las clases genéricas las entidades cuyo tipo es el parámetro formal sólo pueden cualificar los mensajes comprendidos por `ANY`
- En otras palabras: Los parámetros formales son conformes sólo con `ANY` ¿No es esto una excepción?
- Al definir un parámetro formal podemos establecer el tipo con que será conforme (`G -> INTEGER` por ejemplo) con ello:
  - Aumentamos los mensajes que entienden los parámetros formales
  - Disminuimos los tipos utilizables como parámetros formales
- La genericidad definida como hasta ahora es una abreviatura de `G -> ANY`

Un vector sumable

```
class VECTOR[G -> NUMERIC]
inherit
  ARRAY[G]
create
  make,with_capacity,from_collection
feature {ANY}
  infix "+" (otro: like Current): like current is
    -- Suma con otro coherente con 'Current'
  require
    lower = otro.lower
    upper = otro.upper
  local   i:INTEGER
  do
    create Result.make(lower,upper)
    from i:=lower until i> upper loop
      Result.put(item(i)+otro.item(i),i);i:=i+1
    end
  end
end -- class VECTOR(G)
```