Departamento de Informática | Universidad de Valladolid

Programación II (I.T.I de Gestión)

Eiffel estructurado

Félix Prieto

Curso 2009/10

Introducción

- Lenguaje Orientado a Objetos puro
- Fuertemente tipado
- Dotado de genericidad
- Con un sistema de tipos uniforme
- Dotado con un sistema de contratos
- Posibilidad de utilizar herencia múltiple
- Creado en 1980 por Bertrand Meyer

Introducción (II)

- No es un lenguaje procedimiental, aunque los métodos contenidos en sus clases utilizan programación estructurada
- Las estructuras básicas de control tienen representación en el lenguaje
- Podemos «pervertir» el lenguaje para escribir programas estructurados
- Si lo hacemos será más fácil la transición a la programación Orientada a Objetos
- Sin embargo *Eiffel no es el lenguaje adecuado para hacer programación estructurada*

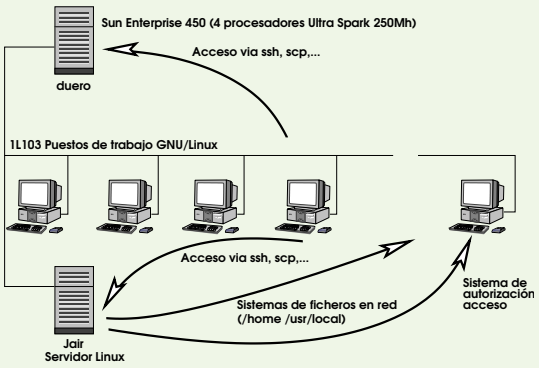
El compilador SmartEiffel

- Versión 1.1 (Ojo, existen diferencias entre las versiones)
- Algunas viejas versiones se denominaban SmallEiffel
- No utilizaremos las versiones más modernas (como 2.3)
- Licencia GNU, disponible desde la página de la asignatura:
  - Código fuente Eiffel del propio compilador, que debe ser compilado
  - Versión compilada para Windows
  - Existen otras versiones compiladas
- Genera código C intermedio para luego producir el ejecutable

Instalación en el laboratorio

- El compilador está instalado en duero, jair, las estaciones de trabajo SUN y todos los personales con GNU/Linux
- Lo natural es utilizar las estaciones de trabajo GNU/Linux
- Se puede editar en local y compilar en jair (buscar la configuración más rápida o cómoda)
- Se puede utilizar desde cualquier otro laboratorio de la escuela
- Dos directorios `home` disponibles, uno para las máquinas SUN y otro para las máquinas GNU/Linux

Instalación en el laboratorio (II)



Primer ejemplo

```
indexing
  description:"Primer ejemplo en Eiffel"
class HOLA_MUNDO
create make
feature
  make is
    do
      std_output.put_string("Hola mundo %N")
    end

end -- class HOLA_MUNDO
```


Primer ejemplo (II)

- Almacenar en un fichero llamado `hola_mundo.e`
- Se puede editar utilizando `vim` o `gvim`
- Compilar mediante la orden `compile hola_mundo`
- Si el método de creación no se llama `make` hay que añadir su nombre como segundo parámetro
- Ejecutar con `./a.out`
- Limpiar los ficheros intermedios con `clean hola_mundo`
- Borrar el ejecutable

Primer ejemplo (III)

- Se ejecutará el método de creación
- En Eiffel no existen funciones definidas a nivel del lenguaje
- Para imprimir una cadena hay que enviar un mensaje a un objeto
- *std_output* es un objeto predefinido, al que enviamos el mensaje *put_string* con el parámetro adecuado
- Los mensajes que entiende un objeto están publicados en la forma corta de su clase
- La forma corta de una clase se obtiene mediante el comando *short*. En nuestro caso *short std_output*

Primer ejemplo (IV)

```
class interface STD_OUTPUT
  -- To use the standard output file. As for UNIX, the default standard
  -- output is the screen
  -- Notes: - the predefined 'std_output' should be use to have only
  -- one instance of the class STD_OUTPUT,
  -- - to do reading or writing at the same time on the screen,
  -- see STD_INPUT_OUTPUT,
..
  put_string (s: STRING)
    -- Output 's' to current output device
    require
      is_connected;
      s /= Void
..
end of STD_OUTPUT
```


Comentarios sobre vim

- *gvim* (o *vim -g*) abre una ventana nueva para la edición
- Para que ilumine la sintaxis *:sintax on*
- Para ver número de fila y columna *set ru*
- Para que nos muestre su ayuda: *:help*
- Buscar algo */algo*. Si queremos substituir *:1,100s/aglo/algo/g*
- El fichero *$HOME/.vimrc* puede contener las opciones por defecto para el editor
- Se pueden añadir al fichero *$HOME/.profile* comandos como `alias vi='vim'`
- Más consejos en la página de la asignatura

Tipos predefinidos

- No existen tipos básicos predefinidos en el lenguaje
- Sin embargo existen clases que modelan los tipos básicos de otros lenguajes
- Podemos utilizar *INTEGER*, *REAL*, *DOUBLE* entre los numéricos, además de *CHARACTER* o *BOOLEAN*, pero *STRING* requiere precauciones especiales
- Para descubrir las posibilidades de estos «tipos predefinidos» es preciso recurrir a la forma corta
- Admiten una sintaxis natural mediante operadores al uso
- Además de los operadores lógicos habituales disponemos de «`and then`» «`or else`» e «`implies`»

Entidades

- En lugar de variables, en Eiffel utilizamos «entidades»
- Podemos declarar entidades locales al método de creación
- También son entidades los argumentos formales de los procedimientos y funciones o los atributos de las clases
- Todas las entidades deben ser declaradas
- El lenguaje dispone de un chequeo fuerte de tipos
- Las entidades se conectan mediante el operador *:=*

Entidades y conexión

```
indexing
  description:"Entidades␣locales␣a␣un␣método"
class ENTIDADES
create make
feature
  make is
    local
      i,j:INTEGER
      a:CHARACTER
    do
      a:='a'; std_output.put_character(a)
      std_output.put_string("%N")
      i:=1; j:=2
      std_output.put_string((i+j).to_string+"%N")
    end

end -- class ENTIDADES
```


Estructura alternativa

```
-- Simple
if <condicion> then
  <sentencia>
end
-- Doble
if <condicion> then
  <sentencia>
else
  <sentencia>
end
```

```
-- Múltiple
if <condicion> then
  <sentencia>
elseif <condicion> then
  <sentencia>
else
  <sentencia>
end
```


Estructura alternativa (II)

```
inspect
  [expr-entera | expr-caracter]
when value,value then
  <sentencia>
when value..value then
  <sentencia>
else
  <sentencia>
end
```

Estructura iterativa

```
from
   <inicializacion>
until
   <condicion>
loop
   <cuerpo>
end
```

- Un solo tipo de iteración
- Siempre con condición al principio
- Podemos simular con ella la iteración con condición al final
- Es conveniente escribir las sentencias de inicialización en su lugar
- La sintaxis completa se verá al tratar la programación bajo contrato

Procedimientos

```
nombre [<parametros>] is
      -- Comentario adecuado
   local
      <entidades locales>
   do
      <sentencias>
   end
```

- No juegan el papel de los módulos en Eiffel
- Pueden utilizarse como los procedimientos en Pascal
- Todos los argumentos son por valor, aunque al hablar de objetos y referencias veremos que eso no es un problema

Funciones

```
nombre [<parametros>]:<tipo> is
      -- Comentario adecuado
   local
      <entidades locales>
   do
      <sentencias>
   end
```

- Los mismos matices que en procedimientos
- La entidad predefinida `Result` permite devolver el resultado

¿Cómo leer un número?

```
std_input.read_integer
a:= std_input.last_integer
```

- Debemos «pedir» a un objeto especializado que lea el número
- Luego podemos consultar al objeto cuál fue el resultado de la lectura

Cálculo de divisores:Método de creación

```
indexing
   description:"Calcula los divisores de un numero"
class DIVISORES
create make
feature
   make is
      local
         numero:INTEGER
      do
         -- Leer el número
         std_output.put_string("Introduce un número entero >1 ")
         std_input.read_integer
         numero:=std_input.last_integer
         if numero<=1 then
            std_output.put_string("Error: Número <=1 %N")
         else
            -- Cálculo y escritura de los factores primos
            std_output.put_string(numero.to_string+"=1")
            divisores(numero)
         end
      end
```


Cálculo de divisores:divisores

```
divisores(n:INTEGER) is
      -- Escribe todos los divisores de 'n'
   local
      i,factor:INTEGER
   do
      from i:=n
      until i=1
      loop
         -- Menor divisor de 'i'
         factor:=divisor(i)
         -- Continúa el cálculo con el cociente
         i:=i//factor
         -- Escribe el resultado
         std_output.put_string("*"+factor.to_string)
      end
      std_output.put_string("%N")
   end -- divisores
```


Cálculo de divisores:divisor

```
   divisor(n:INTEGER):INTEGER is
         -- Menor de los divisores de 'n'
      local
         i:INTEGER
      do
         from i:=2
         until (n \\ i) =0
         loop
            i:=i+1
         end
         Result:=i
      end -- divisor
end -- class DIVISORES
```