Programación Orientada a Objetos

Eduardo Bonelli

Departamento de Computación
FCEyN
UBA

7 de noviembre de 2006

Índice temático

- Vamos a introducir los conceptos fundamentales del paradigma
  - Objetos, clases, modelo de cómputo, herencia
  - Polimorfismo de subclase y dynamic method dispatch
  - Super y static method dispatch
- Ilustraremos estos ejemplos con el lenguaje SOOL
  - Es una extensión de SFLA con orientación a objetos
  - Vamos a implementar un intérprete para SOOL

Índice temático

- Vamos a presentar los problemas principales asociados a los sistemas de tipos para lenguajes orientados a objetos
- Algunos conceptos relevantes son:
  - Sistema de tipos invariantes
  - Subsumption como subclassing
  - Covarianza y contravarianza
  - Falla de Type Safety para Java
  - El "Binary Method Problem"
- Completaremos la visión del paradigma usando Smalltalk (Práctica)

Eje de fundamentos

- Entre los temas más destacados de este eje se encuentra el de desarrollo de sistemas de tipos y semántica para lenguajes OO
- Dos textos sobre el tema:
  1. A Theory of Objects, Martín Abadi y Luca Cardelli, Monographs in Computer Science, David Gries and Fred B. Schneider ed., Springer-Verlag, 1996
  2. Foundations of Object-Oriented Programming Languages: Types and Semantics, Kim B. Bruce, The MIT Press, 2002

Eje de fundamentos

- Un survey (no tan reciente pero relevante) sobre sistemas de tipos para lenguajes OO:
  - The Development of Type Systems for Object-Oriented Languages, Kathleen Fisher y John C.Mitchell, Theory and Practice of Object Systems,1(3): 189-220, 1996
- Disponible en la página personal del primer autor

Sistemas Orientados a Objetos

Sistemas Orientados a Objetos

Un sistema orientado a objetos es un conjunto de objetos que interactúan entre sí para lograr algún objetivo predefinido

- Lenguajes orientados a objetos se usan para implementar sistemas orientados a objetos
- Para poder estudiar estos lenguajes:
  - Debemos comprender qué son los objetos, cómo se especifican y cómo se clasifican
  - Debemos, asimismo, conocer el modelo de cómputo: ¿De qué manera pueden interactúar los objetos?

Objetos

Un objeto se especifica a través de:

- Un conjunto de funciones (usualmente llamadas métodos) que determinan su interfase o protocolo
- Un conjunto de campos o atributos que representan su estado

- Principio básico heredado de los Tipos Abstractos de Datos, antecesores de los Objetos y Clases:

Principio de ocultamiento de la información

El estado de un objeto es privado y solamente puede ser consultado o modificado a través de los métodos provistos por su interfase

Clases

Para facilitar compartir métodos entre objetos de la misma naturaleza, los objetos se suelen especificar a través de clases.

- La clases son estructuras que especifican los campos y métodos de los objetos (llamados variables de instancia y métodos de instancia, resp.)
- Cada objeto es una instancia de alguna clase
- Las clases pueden verse como "moldes" de objetos, cada uno de los cuales es creado a su semejanza
- Las clases también pueden tener sus propios campos (variables de clase o estáticas) y métodos (métodos de clase)

Ejemplo

```
class point extends object
  field x
  field y
  method initialize (initx,inity)
    begin
      set x = initx;
      set y = inity
    end
  method move (dx,dy)
    begin
      set x = +(x,dx);
      set y = +(y,dy)
    end
  method get_location () list(x,y)
```

Clases

Cada clase consiste de

- un nombre y el nombre de la clase que extiende
- una lista de declaraciones de campos
- una lista de declaraciones de métodos
- por cada método se especifica
  - su nombre
  - parámetros formales
  - cuerpo
- `initialize` es un método especial que se invoca cuando un objeto es creado

Ejemplo

```
class point extends object
  field x
  field y
  method initialize (initx,inity)
    begin
      set x = initx;
      set y = inity
    end
  method move (dx,dy)
    begin
      set x = +(x,dx);
      set y = +(y,dy)
    end
  method get_location () list(x,y)
let p = new point(3,4) in p
```

Pasaje de mensajes como modelo de cómputo

- La interacción entre objetos se lleva a cabo a través de pasaje de mensajes
- Llamar al método de un objeto se interpreta como enviar un mensaje a ese objeto que consiste en:
  - el nombre del método
  - los argumentos o parámetros reales
- Por ejemplo, en la expresión

```
send pump depress(5)
```

  - `pump` es el objeto receptor
  - `depress(5)` es el mensaje que se envía
- En este caso, el mensaje consiste en un nombre de método (`depress`) y argumentos (5)

Method dispatch

Supongamos que un objeto envía el mensaje: `send o m(e1,e2)`

- Para poder realizar procesar este mensaje es necesario hallar la declaración del método que se pretende ejecutar

Method Dispatch

El proceso de establecer la asociación entre el mensaje y el método a ejecutar se llama method dispatch

- Si el method dispatch se hace en tiempo de
  - compilación (i.e. el método a ejecutar se puede determinar a partir del código fuente): se habla de method dispatch estático
  - ejecución: se habla de method dispatch dinámico

Ejemplo

```
class interior_node
   extends object
 field left, right
 method initialize (l,r)
   begin
     set left = l;
     set right = r
   end
 method sum() +(send left sum(),
    send right sum())
class leaf_node extends object
 field value
 method initialize (v) set value = v
 method sum () value
```

Ejemplo

```
class interior_node
   extends object
 field left,right
 method initialize (l,r)
   begin
     set left = l;
     set right = r;
   end
 method sum() +(send left sum(),
    send right sum())
class leaf_node extends object
 field value
 method initialize (v) set value = v
 method sum () value
```

```
let o1 = new interior_node(
                new interior_node(
                 new leaf_node(3),
                 new leaf_node(4)),
                new leaf_node(5))
in send o1 sum()
```

Jerarquía de clases

- Es común que nuevas clases aparezcan como resultado de la extensión de otras existentes a través de la
  - adición o cambio del comportamiento de un método
  - adición de nuevos campos
- Una clase puede heredar de o extender una clase pre-existente (la superclase)
- Si una clase c2 hereda de otra c1, todos los campos y métodos de c1 serán visibles desde c2, salvo que sean redeclarados
- Por ello, la herencia promueve el reuso de código
- La transitividad de la herencia da origen a las nociones de ancestros y descendientes

Herencia

- Hay dos tipos de herencia
  - Simple: una clase tiene una única clase padre (salvo la clase raíz `object`)
  - Múltiple: una clase puede tener más de una clase padre
- La gran mayoría de los lenguajes OO utilizan herencia simple
- Si bien en algunas situaciones puede ser útil, el mayor inconveniente con herencia múltiple es que una clase puede tener dos o más superclases con métodos del mismo nombre
- Determinar cuál de los métodos se heredan es, en el mejor de los casos, arbitrario

Ejemplo

```
class point extends object
  field x, y
  method initialize (initx,inity)
    begin
      set x = initx;
      set y = inity
    end
  method move (dx,dy)
    begin
      set x = +(x,dx);
      set y = +(y,dy)
    end
  method get_location () list(x,y)
class colorpoint extends point
  field color
  method set_color (c) set color = c
  method get_color () color
```

Ejemplo

```
class point extends object
  field x, y
  method initialize (initx,inity)
    begin                                let p = new point(3,4);
      set x = initx;                        cp = new colorpoint(10,20)
      set y = inity                      in begin
    end                                     send p move(3,4);
  method move (dx,dy)                       send cp set_color(87);
    begin                                   send cp move(10,20);
      set x = +(x,dx);                      list (send p get_location (),
      set y = +(y,dy)                        send cp get_location (),
    end                                      send cp get_color ()))
  method get_location () list(x,y)      end
class colorpoint extends point
  field color
  method set_color (c) set color = c
  method get_color () color
```

Ejemplo - Redeclaración

```
class c1 extends object
  field x,y
  method initialize () 1
  method setx1 (v) set x = v
  method sety1 (v) set y = v
  method getx1 () x
  method gety1 () y
class c2 extends c1
  field y
  method sety2 (v) set y = v
  method getx2 () x
  method gety2 () y
```

```
let o2 = new c2()
in begin
     send o2 setx1(100);
     send o2 sety1(102);
     send o2 sety2(999);
     list(send o2 getx1();
          send o2 gety1();
          send o2 getx2();
          send o2 gety2())
   end
```

Polimorfismo de subclase

Polimorfismo de subclase

Una instancia de cualquier descendiente de una clase puede utilizarse en lugar de una instancia de la clase misma

- Polimorfismo de subclase se implementa a través de method dispatch dinámico
  - esto es consecuencia de que el método a ejecutar no puede ser determinado sin conocer el objeto receptor del mensaje
  - esto último sólo puede conocerse en tiempo de ejecución

Ejemplo

```
class c1 extends object
  method initialize () 1
  method m1 () 1
  method m2 () send self m1()
class c2 extends c1
  method m1 () 2
let o1 = new c1()
    o2 = new c2()
in list(send o1 m1(),
        send o2 m1(),
        send o2 m2())
```

Nota: el identificador `self` se liga al objeto receptor del mensaje

Ejemplo

```
class c1 extends object
  method intialize () 1
  method m1 () 1
  method m2 () 100
  method m3 () send self m2()
class c2 extends c1
  method initialize () 1
  method m2 () 2
let o1 = new c1()
    o2 = new c2()
in list(send o1 m1(),
        send o1 m2(),
        send o1 m3(),
        send o2 m1(),
        send o2 m2(),
        send o2 m3())
```

Method Dispatch Estático

- Method dispatch dinámico es uno de los pilares de la POO (junto con la noción de clase y de herencia)
- Por cuestiones de eficiencia (o diseño, como el caso de C++) muchos lenguajes también cuentan con method dispatch estático
- Sin embargo, hay algunas situaciones donde method dispatch estático se requerido, más allá de cuestiones de eficiencia
- La sentencia que ejemplifica esto es el `super`

Super

Supongamos que queremos extender la clase `point` del siguiente modo:

```
class colorpoint extends point
  field color
  method initialize (initx, inity, initc)
    begin
       set x = initx;
       set y = inity;
       set color = initc
    end
  method set_color (c) ...
  method get_color () ...
```

Super

- ¡El cuerpo de `initialize` duplica código innecesariamente!
- Esto es un ejemplo claro de mala práctica de programación en función a la presencia de herencia
- Deberíamos recurrir al código ya existente del método `initialize` de `point` para que se encargue de la inicialización de `x` e `y`
- ¿La siguiente variante funciona?

```
class colorpoint extends point
  field color
  method initialize (initx, inity)
    begin
       send self initialize(initx,inity);
       set color = "azul"
    end
  method set_color (c) ...
  method get_color () ...
```

Super

- clase anfitriona de un método: clase donde se declara el método
- Una expresión de la forma `super s(...)` que aparece en el cuerpo de un método `m` invoca el método `s` del padre de la clase anfitriona de `m`
- El código correcto debería ser

```
class colorpoint extends point
  field color
  method initialize (initx, inity)
    begin
       super initialize(initx,inity);
       set color = "azul"
    end
  method set_color (c) ...
  method get_color () ...
```

Diferencia Super/Self

- Una expresión de la forma `super s(...)` se dice llamado super
- Un llamado super es similar a un llamado a `self` en el sentido que el objeto receptor es `self` en ambos casos
- Sin embargo, method dispatch es estático en el primero caso y dinámico en el último
- Vamos a ver un ejemplo que ilustre esta diferencia

Ejemplo - Super/Self

```
class c1 extends object
  method initialize () 1
  method m1 () send self m2()
  method m2 () 13
class c2 extends c1
  method m1 () 22
  method m2 () 23
  method m3 () super m1 ()
class c3 extends c2
  method m1 () 32
  method m2 () 33
let o3 = new c3()
in send o3 m3()
```

Method Overloading (Sobrecarga de Métodos)

- Facilidad que le permite a una clase tener múltiples métodos con el mismo nombre, siempre y cuando tengan diferente signaturas
- La signatura de un método típicamente consiste en
  - el nombre del método
  - el número de parámetros
  - el tipo de los parámetros y del resultado
- Esta noción también existe en lenguajes imperativos y funcionales
- A veces lleva el nombre de ad-hoc polymorphism

Method Overloading (Sobrecarga de Métodos)

```
class colorpoint extends point
  field color
  method initialize (initx, inity, initc)
    begin
       super initialize(initx,inity);
       set color = initc
    end
  method initialize (initx, inity)
    begin
       super initialize(initx,inity);
       set color = "azul"
    end
```

Resolviendo la sobrecarga

Main

```
A x;
x=new B();
System.out.print(x.m(5));
```

Output

20

```
public class A {
    public int m(float x) {
     return 10;
    };
}

public class B extends A {
    public int m(float x) {
     return 20;
    };
}
```

Resolviendo la sobrecarga

Main

```
A x;
x=new B();
System.out.print(x.m(5));
```

Output

10

```
public class A {
    public int m(int x) {
     return 10;
    };
}

public class B extends A {
    public int m(float x) {
     return 20;
    };
}
```

Method Override (Redefinición de Métodos)

- Facilidad que le permite a una clase redefinir un método o campo heredado
- El método redefinido, en general, agrega comportamiento específico de la subclase
- Es aconsejable que el método redefinido tenga relación lógica con el método que se redefine
- En la presencia de sistemas de tipos, la signatura del método redefinido debe "parecerse" a la del método a redefinir (veremos más detalles en la clase de sistemas de tipos para lenguajes OO)

Method Override (Redefinición de Métodos)

```
class A
  field x
  method A()
    begin
      set x = 1
    end
class B extends A
  method A()
    begin
       set x = 2
    end
let p = new B() in send p A()
```

SOOL

- Introduciremos el lenguaje SOOL
  - Es una extensión de SFLA con orientación a objetos
  - En particular agrega primitivas para
    - declarar clases, atributos y métodos
    - crear nuevas instancias de una clase
    - mandar mensajes a objetos
    - referirse a super y self
  - La clase que viene vamos a implementar un intérprete para SOOL

Sintaxis concreta de SOOL

Extendemos SFL con las siguientes producciones:

```
<program> ::= {<class-decl>}* <expr>

<class-decl> ::= class <ident> extends <ident>
                 {field <ident>}* {<method-decl>}*

<method-decl> ::=
  method <ident> ({<ident>}*(,)) <expr>

<expr> ::= ... | new <ident> ({<expr>}*(,))

<expr> ::= send <expr> <ident> ({<expr>}*(,))

<expr> ::= super <ident> ({<expr>}*(,))
```

Sintaxis concreta de SOOL

Observar que la nueva gramática:

- Incluye una sentencia begin-end

  ```
  <expr> :: = begin <expr> ; <expr>(*) end
  ```

  Las expresiones se evalúan en orden y se retorna el valor de la última de ellas

- Incluye primitivas de procesamiento de listas

  ```
  <primitive> ::= ... | list | cons | nil |
                  head | tail | null?
  ```

Valores expresados y denotados

Recordar

- Valores expresados: valores que pueden resultar de la evaluación de las expresiones del lenguaje
- Valores denotados: valores que se pueden asignar a las variables

En SOOL tenemos

Valores Expr. = Number + ProcVal + Obj + List(Valores Expr.)
Valores Den. = Ref(Expressed Value)