Lenguajes de Programación, 2017-1
Nota de clase 14: Introducción al Paradigma Orientado a Objetos [*]

Favio E. Miranda Perea Lourdes del Carmen González Huesca
Facultad de Ciencias UNAM

19 de noviembre de 2018

1. Introducción

Contestar a la pregunta ¿Qué es la programación orientada a objetos? no es sencillo, y depende del punto de vista particular de quién responda, sin embargo existen ciertas características importantes del paradigma que deberían estar incluidas en cualquier respuesta a esta pregunta y que resumimos a continuación

1.1. Representaciones Multiples

Cuando una operación se invoca en un objeto particular, el objeto mismo determina que código ejecutar. Dos objetos que responden a un mismo conjunto de operaciones, es decir que tienen la misma interfaz, pueden usar representaciones distintas, siempre y cuando cuenten con una implementación de las operaciones que funcione con la representación particular. Estas implementaciones son los *métodos* del objeto. Invocar una operación en un objeto -lo cual se conoce como invocación del método o envio de mensajes- involucra la búsqueda del nombre de la operación en tiempo de ejecución en una tabla de métodos asociada al objeto, este proceso se llama *despacho dinámico*[1].

1.2. Encapsulado

La representación interna de un objeto se esconde, por lo general, fuera de la definición del objeto mismo. Sólo los métodos del objeto pueden manipular o inspeccionar sus campos. Esto implica que los cambios a la representación interna de un objeto afectan sólo a una región pequeña y fácilmente identificable del programa. Esta restricción mejora considerablemente el mantenimiento y legibilidad de sistemas grandes.

1.3. Subtipos (herencia de interfases)

El tipo de un objeto, es decir su interfaz, es simplemente el conjunto de nombres y tipos de sus operaciones. La representación interna del objeto no figura en su tipo puesto que no afecta el conjunto de cosas que podemos hacer directamente con él. Las interfaces generan una relación

[*] Estas notas se basan en el libro de Pierce y en material de Ralph Hinze y Gerwin Klein.
[1] *Dynamic dispatch*

natural de subtipos: un objeto puede usarse en un contexto en el que sólo una parte de sus métodos se necesiten. De esta manera podemos definir funciones polimórficas capaces de manejar objetos distintos de manera uniforme.

1.4. Herencia (de implementaciones)

La herencia permite factorizar la implementación de objetos distintos de manera que el comportamiento común se implemente una única vez. La mayoría de lenguajes orientados a objetos permiten este reuso de comportamientos mediante estructuras llamadas clases las cuales proporcionan plantillas[2] a partir de las cuales se instancian objetos. Un mecanismo de *subclases* permite derivar nuevas clases a partir de otras agregando implementaciones para nuevos métodos y dando prioridad, en caso de ser necesario, a las implementaciones de los métodos viejos. En algunos lenguajes se utiliza la técnica de *delegación* en lugar de subclases.

1.5. Recursión abierta

Un método de un objeto puede llamar a otro dentro del mismo objeto con ayuda de las variables especiales *self* ó *this*. La variable *self* se liga dinámicamente de manera que se puedan llamar métodos definidos en subclases.

2. Modelando Objetos

En su forma más simple un objeto es una estructura de datos que encapsula cierto estado interno ofreciendo acceso a este mediante una colección de métodos. El estado interno se organiza generalmente mediante un número de atributos, llamados también variables de instancia o campos, los cuales se comparten entre los métodos y son inaccesibles para el resto del programa. Una forma de obtener este comportamiento de manera directa es usando celdas de referencia para el estado interno de un objeto y un record de funciones para los métodos. Dado el uso de referencias y asignaciones nos referiremos a esta clase de objetos primitivos como objetos imperativos.

Como ejemplo de como modelar objetos implementamos un contador muy simple cuyo estado interno actual es 1:

$$
\begin{array}{rcl}
c & =_{def} & \mathsf{let}\ x = \mathsf{ref}\ \ 1\ \mathsf{in} \\
 & & \quad \{\, get = \lambda\_ : \mathtt{Void}.\, !\, x, \\
 & & \quad\ \ inc = \lambda\_ : \mathtt{Void}.\, x := \mathsf{suc}(!\, x)\} \\
 & & \mathsf{end}
\end{array}
$$

Tenemos $\vdash c : \mathsf{Counter}$ donde $\mathsf{Counter} =_{def} (\, get : \mathtt{Void} \to \mathsf{Nat},\ inc : \mathtt{Void} \to \mathtt{Void})$

Observemos que los métodos se escriben como funciones con parámetros triviales dado que no figuran en los cuerpos. El uso de una abstracción bloquea la evaluación del método en el momento en que éste es creado permitiendo una evaluación repetida más tarde al aplicarse al valor trivial () cuantas veces se requiera. Además la variable x se comparte en los métodos y, al ser local, está encapsulada y es inaccesible al resto del programa.

[2] *Templates*

Para invocar un método simplemente lo extraemos del record y lo aplicamos a un argumento apropiado, por ejemplo:

$$c.inc\,() \,;c.get\,() \rightarrow^{\star} 2 \quad \text{después de lo cual se tiene} \quad c.inc\,() \,;c.inc\,() \,;c.get\,() \rightarrow^{\star} 4$$

Una vez representado el objeto y su tipo podemos utilizarlo en otros programas, por ejemplo:

$$inc3 =_{def} \lambda c : \mathsf{Counter}.(c.inc\,() \,;c.inc\,() \,;c.inc())$$

Esta función incrementa en 3 el estado interno, por ejemplo se tiene $inc3\,c \,;c.get\,() \rightarrow^{\star} 7$

3. Ejemplos

Un programa orientado a objetos consta de una sucesión de definiciones de clases seguida de una expresión a evaluarse (correspondiente al programa principal o *main*), por ejemplo

```
class C {
 field a
 method C() {a := 5}
 method m() {return a}
 }

// main
let x = new C() in
 x.m()
end
```

Se observa que el campo o atributo es `a`, mientras que los métodos son `C` correspondiente al constructor y `m`. El constructor de lenguaje `new` indica la creación de un nuevo objeto de la clase `C`, esto se hace mediante la invocación inmediata del metodo `C`, el cual asigna valores iniciales a los atributos del nuevo objeto. La expresión `x.m()` indica un paso o envio de mensajes del objeto `x` al método `m`. Esto puede también escribirse como `send x m()` como en los antiguos lenguajes orientados a objetos (SMALLTALK), pero en nuestro caso preferimos usar notación más actual. En este ejemplo el programa principal se evalua a 5.

Consideremos otro ejemplo:

```
class C {
 field i
 field j
 method C(x) {
   i := x;
   j := 0 -x
 }
 method add(d) {
   i := i+d;
```

```
   j:= j-d
 }
 method getstate() {
  return [i,j]
 }
}
```

En este caso hay dos atributos y el constructor recibe un parámetro. Los objetos de esta clase preservarán el invariante `i + j = 0`. Considérese ahora el siguiente programa principal, recordando que estos ejemplos no involucran un tipado correcto en el sentido estudiado anteriormente.

```
let a = 0
    b = 0
    x = new c(5)
in
  a:= x.getstate();
  x.add(3);
  b := x.getstate();
  [a,b]
end
```

En este caso la creación de objetos requiere de un parámetro, en este caso 5. El programa se evalua a la lista `[[5,-5],[8,-8]]`

3.1. Recursión abierta

En el siguiente programa se ejemplifica el mecanismo de recursión abierta mediante el uso de la variable `this`, también conocida como `self`.

```
class OddEven {
  method OddEven() {1}
  method even(n) { if n = 0 then  1 else this.odd(n-1)}
  method odd(n) { if n = 0 then 0 else this.even(n-1)}
 }

new OddEven().odd(17)
```

En este caso se definen mediante recursión mutua los dos métodos `even,odd` usando la variable `this` para referirse al objeto mismo que se está definiendo. Se observa que el constructor no juega ningun papel importante de hecho podría omitirse pues la clase no tiene atributos.

3.2. Despacho dinámico

Una de las características más importantes del paradigma orientado a objetos es la capacidad de los objetos para enviarse mensajes entre si. Considérese el siguiente programa que implementa árboles binarios.

```
Class Node {
  field left
  field right
  method Node(x,y){ left := x; right := y}
  method sum() { return (left.sum() + right.sum()) }
}

Class Leaf {
 field value
 method Leaf(v) { value := v }
 method sum() { return value }
}

let x = new Node(New Node(new Leaf(13),new Leaf(4)),
                 New Leaf(5))
   in
     x.sum()
end
```

Se observa en la definición del método `sum` en la clase `Node` la llamada al mismo método `sum` por parte de los atributos `left,right`. Esto no debe entenderse como una recursión incorrecta. La implementación esta asumiendo que los atributos serán objetos de una clase que proporcionará un método `sum` en su interfaz, por ejemplo, otros elementos de la clase `Node` o bien instancias de la clase `Leaf`. En general, si un objeto "sabe" que otro objeto debe incluir cierto método en su interfaz entonces el primero puede llamar al método del segundo objeto cuyo nombre es el mismo. En este caso el programa se evalua a 12.

Para que la evaluación funcione de esta manera, la mayoría de los lenguajes orientados a objetos utilizan el despacho dinámico[3] de métodos. Es el objeto en tiempo de ejecución quien decide que implementación de un método particular utilizar.

3.3. Herencia

El mecanismo de herencia permite definir nuevas clases mediante una modificación sencilla a clases existentes, por ejemplo agregando nuevos atributos o métodos a una clase. Veamos un ejemplo.

```
class Point {
  field x
  field y
  method Point(initx,inity) { x:=initx; y := inity}
  method move(dx,dy) { x:=x+dx; y:=y+dy}
  method getLocation() { return (x,y) }

class ColorPoint extends Point {
 field color
 method setColor(c) { color := c }
```

[3] dynamic dispatch

```
 method getColor() { return color }
}

let p  = new Point(3,4)
    cp = new ColorPoint(10,20)
in
 p.move(3,4);
 cp.setColor(Red);
 cp.move(10,20);
 [p.getLocation(),cp.getLocation(),cp.getColor()]
end
```

Los métodos `move,getLocation` no están definidos en la clase `ColorPoint` sin embargo se heredan de la clase `Point` por lo que el resultado del programa es `[(6,8),(20,40),Red]`.

El uso de herencia no obliga a usar las definiciones existentes en la superclase, por ejemplo la subclase puede redeclarar un atributo de la superclase:

```
class C1 {
 field x
 field y
 method C1() {1}
 method setx1(v) { x:=v }
 method sety1(v) { y:=v }
 method getx1() { return x }
 method gety1() { return y }
}

class C2 extends C1 {
 field y
 method sety2(v) { y:=v }
 method getx2() { return x }
 method gety2() { return y }
}
```

El programa principal

```
let o = new C2() in
 o.setx1(101);
 o.sety1(102);
 o.sety2(999);
 [ o.getx1(), o.gety1(), o.getx2(), o.gety2() ]
end
```

se evalua a `[101,102,101,999]` debido a que una instancia de la subclase siempre buscará sus atributos dentro de su misma definición y de no encontrarlos los buscará en la superclase. De manera que el atributo `y` se evalua a 999.

3.4. Redefinición de métodos (*Overriding*)

Si una subclase redefine un método ya presente en la superclase o en alguna clase de la que herede la superclase decimos que el nuevo método tiene predominancia sobre el anterior (*overrides*).

```
class C1{
 method C1() {return 1}
 method m1() {1}
 method m2() { return this.m1() }
}

class C2 extends C1{
 method m1() {return 2}
}

let o1=new C1()
    o2=new C2()
in
 [o1.m1(), o2.m1(),o2.m2()]
end
```

¿Qué implementación de `m1` se utiliza en el cuerpo del método `m2` dentro de un objeto instancia de `C2`? de acuerdo al despacho dinámico nos referimos al `m1` definido en `C2` por lo que el programa se evalua a `[1,2,2]`. Si el despacho de métodos fuera estático, como en C++, el resultado sería `[1,2,1]`.

3.5. Llamada a métodos de la superclase

Hay situaciones donde se necesita una llamada a un método de la superclase, a pesar de que dicho método se haya redefinido. Por ejemplo podemos definir la clase de puntos coloreados de manera que tenga su propio método de inicialización

```
class ColorPoint extends Point {
 field color
 method ColorPoint(initx,inity,initcolor){ x:= init x;
                                           y:= init y;
                                           color := initcolor
                                         }
 method setColor(c) { color := c }
 method getColor() { return color}
}
```

En lugar de repetir la inicialización de los campos `x,y` de la superclase se puede usar la variable `super` que llama a los métodos correspondientes en la superclase.

```
class ColorPoint extends Point {
 field color
```

```
 method ColorPoint(initx,inity,initcolor){ super.Point(initx,inity);
                                           color := initcolor
                                         }
 method setColor(c) { color := c }
 method getColor() { return color}
}
```

De esta manera concluimos nuestra discusión informal acerca de las principales características del paradigma orientado a objetos. En la siguiente nota nos dedicamos a estudiar un lenguaje formal base para este paradigma, conocido como Java Peso Pluma JPP[4].

[4] Featherweight Java