Programación Orientada a Objetos

Departamento de Computación, FCEyN, UBA

Objetos - La metáfora

- Todo programa es una simulación. Cada entidad del sistema que se está simulando se representa en el programa a través de una entidad u objeto.
  - Asociar los objetos físicos o conceptuales de un dominio del mundo real con objetos del dominio del programa
  - los objetos en el programa tienen las características y capacidades del mundo real que nos interese modelar.
- Todas las componentes de un sistema son objetos

Modelo de Computación - La metáfora

- El modelo de computación consiste en el envío de mensajes: Un sistema está formado por *objetos* que comunican a través del intercambio de mensajes.
- Un mensaje es una solicitud para que un objeto lleve a cabo una de sus operaciones
- El receptor, es decir, el objeto que recibe el mensaje, determina cómo llevar a cabo la operación.

Ejemplo: `"unCirculo radio"`/`unCirculo.radio`

- `unCirculo` es el objeto receptor
- `radio` es el mensaje que se le envía.

Objetos

- El conjunto de mensajes a los que un objeto responde se denomina interfaz o protocolo
- La forma en que un objeto lleva a cabo una operación está descripta por un método (describe la implementación de las operaciones)
- La forma en que un objeto lleva a cabo una operación puede depender de un estado interno
  - El estado se representa a través de un conjunto de colaboradores internos (también llamados atributos o variables de instancia)

unRectangulo

- interfaz: `area`
- atributos: `alto` y `ancho`
- método: `area = function (){return alto*ancho}`

Objetos

La única manera de interactuar con un objeto es a través del envío de mensajes

- la implementación de un objeto no puede depender de los detalles de implementación de otros objetos
- principio básico heredado de los Tipos Abstractos de Datos, antecesores de los Objetos:

Principio de ocultamiento de la información

- El estado de un objeto es privado y solamente puede ser consultado o modificado por sus métodos.
- No todos los lenguajes imponen esta restricción.

Detrás del modelo de cómputo: Method dispatch

- La interacción entre objetos se lleva a cabo a través de envío de mensajes
- Al recibir un mensaje se activa el método correspondiente
- Para poder procesar este mensaje es necesario hallar la declaración del método que se pretende ejecutar
- 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

Corrientes

¿Quién es responsable de conocer los métodos de los objetos?

2 Alternativas conocidas:

- Clasificación
- Prototipado

Clasificación

Clases

- Modelan conceptos abstractos del dominio del problema a resolver
- Se utilizan principios de diseño para decidir cuando crearlas
- Definen el comportamiento y la forma de un conjunto de objetos (sus instancias)
- Todo objeto es instancia de alguna clase

Ejemplo

clase Point

Variables de instancia 'xCoord' e 'yCoord'

Métodos

```
x
    ^xCoord
y
    ^yCoord

dist: aPoint
"Answer the distance between aPoint and the receiver."
    | dx dy |
    dx := aPoint x - xCoord.
    dy := aPoint y - yCoord.
    ^ (dx * dx + (dy * dy)) sqrt
```

Componentes de una clase

Componentes de una clase

- un nombre
- definición de variables de instancia
- métodos de instancia
- por cada método se especifica
  - su nombre
  - parámetros formales
  - cuerpo

Ejemplo

clase INode

Métodos de clase

```
l: leftchild r:rightchild
    "Creates an interior node"
    ...
```

Vars. de instancia `'left right'`

Métodos de instancia

```
sum
    ^ left sum + right sum
```

clase Leaf

Métodos de clase

```
new: anInteger
    "Creates a leaf"
    ...
```

Vars. de instancia `'value'`

Métodos de instancia

```
sum
    ^value
```

Ejemplos

```
1) Leaf new: 5
2) (INode l: (Leaf new: 3) r: (Leaf new: 4)) sum
```

Self

Pseudo variable que, durante la evaluación de un método, referencia al receptor del mensaje que activó dicha evaluación.

- no puede ser modificada por medio de una asignación.
- se liga automáticamente al receptor cuando comienza la evaluación del método.

clase INode

Métodos de clase

```
l: leftchild r:rightchild
    "Creates an interior node"
    ...
```

Var. de instancia `'left right'`

Métodos de instancia

```
l
    ^left
r
    ^right
sum
    ^ (self l) sum + (self r) sum
```

Jerarquía de clases

- Es común que nuevas clases aparezcan como resultado de la extensión de otras existentes incluyendo
  - adición o cambio del comportamiento de uno o varios métodos
  - adición de nuevas variables de instancia o clase
- Una clase puede heredar de o extender una clase pre-existente (la superclase)
- La transitividad de la herencia da origen a las nociones de ancestros y descendientes

Ejemplo

```
Object subclass: #Point
instanceVarNames: 'xCoord yCoord'
```

Métodos de clase

```
x: p1 y: p2
    ^self new setX: p1 setY: p2
```

Métodos de instancia

```
x
    ^xCoord
y
    ^yCoord

setX: xValue setY:yValue
   xCoord :=  xValue.
   yCoord :=  yValue.
```

```
Point subclass: #ColorPoint
instanceVarNames: 'color'
```

Métodos de clase

```
x: p1 y: p2 color: aColor
    |instance|
    instance := self x: p1 y: p2.
    instance color: aColor.
    ^instance
```

Métodos de instancia

```
color: aColor
    color := aColor

color
    ^color
```

uso

```
ColorPoint x: 10 y: 20 color: red.
```

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
- La herencia múltiple complica el proceso de method dispatch

Inconveniente con herencia múltiple

- Supongamos que
  - clases A y B son incomparables y C es subclase de A y B
  - A y B definen (o heredan) dos métodos diferentes para m
  - se envía el mensaje m a una instancia C
- ¿Qué método debe ejecutarse?
- Dos soluciones posibles:
  - Establecer un orden de búsqueda sobre las superclases de una clase
  - Si se heredan dos métodos diferentes para el mismo mensaje debe ser redefinidos en la clase nueva

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 es requerido, más allá de cuestiones de eficiencia
- Un ejemplo es `super`

Method Dispatch Estático

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

```
Object subclass: #Point
Métodos de instancia
setX: xValue setY: yValue
  xCoord:= xValue.
  yCoord:= yValue.

Point subclass: #ColorPoint
Métodos de instancia
setX: xValue setY: yValue setColor: aColor
  xCoord:= xValue.
  yCoord:= yValue.
  color:= aColor.
```

¡`setX: setY: setColor:` duplica código innecesariamente!

Method Dispatch Estático

- Notar que estamos repitiendo código y además deberiamos adaptar el código si la super clase cambia para dejarlo consistente
- Deberíamos recurrir al código ya existente del método `setX:setY:` de `Point` para que se encargue de la inicialización de `x` e `y`

```
Object subclass: #Point
Métodos de instancia
setX: xValue setY: yValue
  xCoord:= xValue.
  yCoord:= yValue.

Point subclass: #ColorPoint
Métodos de instancia
setX: xValue setY:yValue setColor: aColor
   self setX: xValue setY: yValue.
   color:= aColor.
```

Method Dispatch Estático

- ¿La siguiente variante funciona?

```
Object subclass: #Point
Métodos de instancia
setX: xValue setY: yValue
  xCoord:= xValue.
  yCoord:= yValue.

Point subclass: #BluePoint
instanceVarNames: 'color'

Métodos de instancia
setX: xValue setY: yValue
   self setX: xValue setY: yValue.
   color:= 'azul'.
```

Method Dispatch Estático

Super

- Pseudovariable que referencia al objeto que recibe el mensaje
- Cambia el proceso de activación al momento del envío de un mensaje.
- Una expresión de la forma `"super msg"` que aparece en el cuerpo de un método `m` provoca que el method lookup se haga desde el padre de la clase anfitriona de `m`

Código corregido

```
Object subclass: #Point
Métodos de instancia
setX: xValue setY: yValue
  xCoord:= xValue.
  yCoord:= yValue.
```

```
Point subclass: #BluePoint
Métodos de instancia
setX: xValue setY: yValue
   super setX: xValue setY: yValue.
   color:= 'azul'.
```

Ejemplo - Super/Self

```
Object subclass: #C1
Métodos de instancia
m1
  ^self m2
m2
  ^13
```

```
C1 subclass: #C2
Métodos de instancia
m1
  ^22
m2
  ^23
m3
  ^super m1
```

```
C2 subclass: #C3
Métodos de instancia
m1
  ^32
m2
  ^33
```

Sigamos la ejecución de

`(C2 new) m3.` ----¿ Qué valor devuelve? 23

`(C3 new) m3.` ----¿ Qué valor devuelve? 33

Variante de super en algunos lenguajes

```
super[A] n(...)
```

- Similar super ya visto
- Salvo que la búsqueda comienza desde la clase A
- A debe ser una superclase de la clase anfitriona de m, el método que contiene la expresión super

Lenguajes basados en Objetos

- Caracterizados por la ausencia de clases
- Constructores para la creación de objetos particulares

```
let celda = {
    contenido : 0,
    get : function () {return this.contenido;},
    set : function (n) {this.contenido = n;},
}
```

- Procedimientos para generar objetos

```
Celda = function () {
    this.contenido = 0;
    this.get = function ()
            {return this.contenido;};
    this.set = function (n)
            {this.contenido = n;};
}

otracelda = new Celda ();
```

Prototipado

- Construye instancias concretas que se interpretan como representantes canónicos de instancias (llamados prototipos)
- Otras instancias se generan por clonación (copia shallow)

```
celdaClonada = Object.create(celda)
```

- los clones se pueden cambiar

```
celdaClonada.set = function (n){
        this.contenido = this.contenido + n;
        }
```

- Herencia a través de prototipos (más en la próxima clase)