Teórica 11 : Programación Orientada a Objetos (OOP)

Tomás Felipe Melli

June 26, 2025

Índice

1 Introducción

El paradigma de **Programación Orientada a Objectos** está inspirado en otras disciplinas como la biología en donde los organismos vivos se adaptan e interactúan entre sí. Esta motivación hace que los programas estén conformados por **componentes**, que denominamos **Objetos, que interactúan entre sí intercambiando mensajes.** Es decir, que las entidades físicas o conceptuales del dominio del problema que se desea modelar se representarán con `Objetos`. La idea será entonces que esos objetos reflejen apropiadamente los aspectos que nos interesan de las entidades reales.
La programación orientada a objetos aparece alrededor de 1970 para abstraer técnicas comunes de la programación procedural como :

- Pasaje de registros para permitir código reentrante (Alternativa superadora a las variables globales). Esto es, en los 60' era común usar variables globales por lo tanto había problemas de concurrencia, entre otros. Esta técnica de pasaje de registros es básicamente agrupar datos relacionados en una estructura y que eso se pase como parámetro a las funciones que la manipulan. Esto es importante porque las funciones no dependen de datos globales y por tanto, sean `reentrantes` (se pueden llamar muchas veces de forma segura, hasta concurrente). Esto es el concepto de **instancia**, donde los métodos manipulan una instancia.
- Agrupamiento de las funciones en módulos, esta idea es la de **Clase**. Se encapsulan los métodos y atributos.
- Polimorfismo por indirección, era una manera de manipular dinámicamente el comportamiento de una función según qué estructura la utilizaba. Precedente claro del **Polimorfismo dinámico** donde se utiliza una **Interface** común para distintas clases y cada una implementa comportamientos diferentes.

Algunos ejemplos de estos primeros lenguajes son : **Smalltalk, Simula, Self**.

2 Conceptos fundamentales de la OOP

- Decimos que un entorno Orientado a Objetos se compone de **Objetos**.
- Un **Mensaje** es una solicitud que le hace un objeto a otro para que lleve a cabo una operación.
- La **Interfaz** de un objeto es el conjunto de mensajes que es capaz de responder (`Protocolo`).
- Un **Método** es el procedimiento que usa un objeto para responder un mensaje. La implementación de la operación solicitada.
- La forma en la que un objeto lleva a cabo una operación puede depender de**colaboradores externos**(argumentos que recibe en el mensaje) así como de su estado interno, dado por un conjunto de **colaboradores internos**(los atributos o variables de instancia)

2.1 Encapsulamiento

El **principio de encapsulamiento** nos dice que sólo se puede interactuar con un objeto a través de su interfaz. El estado interno de un objeto es inaccesible desde el exterior. Esto nos permite :

- Alternar entre dos representaciones de una misma entidad sin modificar el comportamiento observable. Es decir, el usuario ve siempre lo mismo aunque la implementación varíe.
- **Duck Typing** es cuando un objeto se puede intercambiar por otro que implemente la misma interfaz (que sepa responder los mismos mensajes).

2.2 Diferentes entornos OO

Los entornos OO pueden tener distintas características:

- El envío de mensajes puede ser tanto **sincrónico**(el emisor espera a que el receptor termine de ejecutar el método antes de continuar.) como **asincrónico**(el emisor no espera la respuesta inmediata.).
- El envío de mensajes puede ser **con respuesta** como **sin respuesta**(se ejecuta el método pero no se espera un resultado `fire-and-forget`).
- Los objetos pueden ser **Mutables**(pueden cambiar su estado interno después de ser creados.) o **Inmutables**.
- Cómo se definen los objetos : **Clasificación** (basado en **Clases**) o **Prototipado**(se crean objetos directamente, que se clonan y se modifican.)

- La herencia puede ser **simple**(una clase sólo puede tener una única clase padre) o **múltiple**(puede heredar de varias clases al mismo tiempo.)

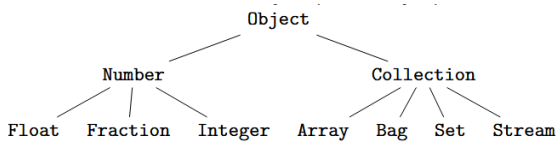
Vamos a usar **Smalltalk** que donde los objetos se definen por clasificación con herencia simple, pueden cambiar y el envío de mensajes es sincrónico y se espera una respuesta.

2.3 Clases e Instancias

Todo objeto es instancia de una clase. Una clase es un objeto que abstrae el comportamiento común de todas sus instancias. Todas las instancias de una clase tienen los mismos atributos. Todas las instancias de una clase usan el mismo método para responder un mismo mensaje.

2.4 Subclasificación y Herencia

Cada clase es **subclase** de alguna otra. Esto nos lleva a estructurar las clases en una **Jerarquía** :



Una clase **hereda** todos los métodos de su **superclase**. Una clase puede decidir si sobreescribe (*override*) un método definido en la superclase por otro más específico.
Existen las llamadas **clases abstractas** que son las que sólo abstraen el comportamiento de sus subclases pero no deben estar nunca instanciadas, como `Number` en el árbol de la imagen.

3 Introducción a la OOP en Smalltalk

3.1 Ejemplo : Clase Par

Creamos el objeto con las variables de instancia `x` e `y`.

```
Object subclass: #Par
    instanceVariableNames: 'x y'
    classVariableNames: ''
    package: 'Ejemplo'
```

Ahora vamos a mandarle un mensaje a la clase `Par` de manera que pueda crear una instancia de `Par`. Acá el `yourself` lo que hace es que en vez de responder lo que responde el nuevo objeto cuando responde el mensaje `y: argumento`, asegura que devolvemos el nuevo objeto recién creado:

```
Par class >> x: unObjeto y: otroObjeto
    ^self new
        x: unObjeto;
        y: otroObjeto;
        yourself
```

Tenemos que definirle "setters" lo cual no es recomendable ya que violaría el encapsulamiento (estos modifican la estructura interna) :

```
Par >> x: unValor
    x := unValor

Par >> y: otroValor
    y := otroValor
```

Podemos evitar eso si modificamos el mensaje de creación de instancia de esta manera :

```
Par class >> x: unObjeto y: otroObjeto
    ^ self new initializeWith: unObjeto y: otroObjeto

Par >> initializeWith: unX y: unY
    x := unX.
    y := unY.
    ^ self
```

Los mensajes tipo "getter" de la instancia, donde el objeto sabe responder quién es su primer coordenada y quién es su segunda:

```
Par >> x
    ^x

Par >> y
    ^y
```

Y definimos la suma :

```
Par >> + otroPar
    ^Par x: (self x + otroPar x) y: (self y + otroPar y)
```

3.2 Ejercicio : `map`

Agregar a la clase Collection un método map: aBlock. Devuelve una colección, de la misma "especie", que resulta de aplicar el bloque a cada elemento de la colección receptora.

```
Collection >> map: aBlock
    | resultado |
    resultado := self new.
    self do: [ :cadaElemento | resultado add: (aBlock value: cadaElemento) ].
    ^ resultado
```

Acá como recibimos un bloque tenemos que evaluarlo con `value` y pasarle el argumento que será manipulado dentro del bloque, `cadaElemento`. Es como la lambda, igualito. Con el `add:` agregamos a la `Collection resultado` la modificación del elemento que impone el bloque que recibimos como argumento. (Cuando escribimos `| variable |` estamos definiendo una `variable temporal`)

3.3 Ejercicio : `mínimo`

```
Collection >> minimo: aBlock
    | minimoValor |
    self isEmpty ifTrue: [ ^nil ].

    minimoValor := aBlock value: self first.
    self allButFirst do: [ :cadaElemento |
        | valor |
        valor := aBlock value: cadaElemento.
        valor < minimoValor ifTrue: [ minimoValor := valor ].
    ].
    ^minimoValor
```

3.4 Sintaxis de expresiones y comandos

Expr ::=	x	`(nombre local)`
	| X	(nombre global)
	| Expr `msg`	`(mensaje unario)`
	| Expr ⟨op⟩ Expr	`(mensaje binario)`
	| Expr $\texttt{msg}_1$: $\texttt{Expr}_1$ ... $\texttt{msg}_N$: $\texttt{Expr}_N$	`(mensaje keyword)`
	| x := Expr	`(asignación)`
Cmd ::=	Expr	`(expresión)`
	| ∧ Expr	`(retorno)`
	| Expr . Cmd	`(secuencia)`

```
contador                "x: nombre local"
Array                   "X: nombre global"
miLista isEmpty         "Expr msg: mensaje unario"
3 + 4                   "Expr    op    Expr: mensaje binario"
punto x: 10 y: 20       "Expr msg1:Expr1 ... msgN:ExprN: mensaje keyword"
total := 3 * 2          "x := Expr: asignaci n"

miLista add: 5          "Expr: comando simple"
^total                  "  Expr  : retorno"
x := 3 + 2.             "Expr . Cmd: secuencia"
```

3.5 Sintaxis en Smalltalk

- Ya hablamos de como declarar `variables locales` con `|x1 ... xn|`
- El operador **;** se usa para encadenar múltiples mensajes al mismo receptor.
- Hay 6 palabras reservadas :

  `nil true false self super thisContext`

- Se incluye notación para distintos tipos de literales : constantes numéricas, caracteres($a), símbolos (#), strings ('hola mundo')

3.6 Polimorfismo

Una misma operación puede operar con objetos que implementan la misma interfaz de distinta manera. Esta característica de la POO se puede aprovechar para construir programas genéricos, que operan con objetos independientemente de sus características específicas.

```
z := OrderedCollection new.
z add: Gato new; add: Perro new; add: Pato new.
z do: [:animal | animal hablar]. "miau guau cuac"
```

Esto sucede ya que cada objeto implementa su propio método para responder al mensaje `hablar`.

3.7 Polismorfimo y Estructuras de control : implementación condicional

```
Object subclass: #ValorDeVerdad
    instanceVariableNames: ''
    classVariableNames: ''
    package: 'Condicional'

ValorDeVerdad subclass: #Verdad
    instanceVariableNames: ''
    classVariableNames: ''
    package: 'Condicional'

ValorDeVerdad subclass: #Falsedad
    instanceVariableNames: ''
    classVariableNames: ''
    package: 'Condicional'

Verdad >> siEsVerdadero: bloqueVerdadero siEsFalso: bloqueFalso
    ^bloqueVerdadero value

Falsedad >> siEsVerdadero: bloqueVerdadero siEsFalso: bloqueFalso
    ^bloqueFalso value
```

Si evaluamos :

```
Verdad new siEsVerdadero: 1 siEsFalso: Misil new lanzar
```

Se rompe porque `siEsVerdadero` recibe *aBlock*

3.8 Bloques (`closures`)

Es un objeto que representa un comando, es decir, una **secuencia de envíos de mensajes**. La sintaxis es :

Expr ::= ...
| `[Cmd]` `(bloque sin parámetros)`
| `[:x1 ... xN | Cmd]` `(bloque con parámetros)`

Los bloques sin parámetros los evaluamos con *aBlock* `value` y los bloques con parámetros como sigue :

aBlock `value: arg1 value: arg2 ... value: argN`

Ejemplo

`[1 + 1] value` $\rightsquigarrow 2$

`[:x :y | x + y] value: 1 value:2` $\rightsquigarrow 3$

3.9 Ejemplo : recorrido sobre árboles

Armamos la jerarquía :

```
Object subclass: #ArbolBinario
    instanceVariableNames: ''.

ArbolBinario subclass: #Nil
    instanceVariableNames: ''.

ArbolBinario subclass: #Bin
    instanceVariableNames: 'izq raiz der'.
```

Tenemos que definir los "constructores" o mensajes de creación de instancia :

```
Nil class >> new
    ^ super new

Bin class >> izq: unArbol raiz: unDato der: otroArbol
    ^ self new
        izq: unArbol;
        raiz: unDato;
        der: otroArbol;
        yourself
```

Super

La palabra reservada super se refiere al mismo objeto que self. Pero super m indica que la búsqueda del método que implementa el mensaje m debe comenzar desde la superclase de la clase actual.
Seguimos ... con los setters :

```
Bin >> izq: unArbol
    izq := unArbol.

Bin >> raiz: unDato
    raiz := unDato.

Bin >> der: unArbol
    der := unArbol.
```

Para evitar setters podemos encapsular todo en un `initializeWith: unArbol raiz: unDato der: otroArbol`. Esto evita exponer la modificación de la estructura interna, es mejor:

```
Bin class >> izq: unArbol raiz: unDato der: otroArbol
    ^ self new initializeWith: unArbol raiz: unDato der: otroArbol

Bin >> initializeWith: unIzq raiz: unRaiz der: unDer
    izq := unIzq.
    raiz := unRaiz.
    der := unDer.
    ^ self
```

Para el `recorrido inorder`

```
Nil >> do: unBloque
    "No hace nada porque no hay elementos."

Bin >> do: unBloque
    izq do: unBloque.
    unBloque value: raiz.
    der do: unBloque.
```

Para `size`

```
ArbolBinario >> size
    | contador |
    contador := 0.
    self do: [:cadaElemento | contador := contador + 1].
    ^ contador
```

3.10 Manejo de mensajes incomprendidos `#doesNotUnderstand`

Cuando le mandás un mensaje a un objeto, y ese objeto no tiene un método con ese selector, el sistema Smalltalk responde enviando a ese objeto el mensaje especial, `doesNotUnderstand: aMessage` Donde `aMessage` es una instancia de la clase Message, que encapsula: el selector (el nombre del mensaje), los argumentos (si los hay), y el receptor original.

3.11 Algoritmo de resolución de métodos `method dispatch`

Recordar que en Smalltalk todo es un objeto y todo sucede enviando mensajes entre ellos. El `method dispatch` es el proceso por el cual Smalltalk (u otro lenguaje OOP) decide qué método ejecutar cuando le envía un mensaje a un objeto. Pensemos en lo siguiente :

```
persona nombre
```

Al **objeto (O) persona** le deseamos enviar el mensaje **nombre (S) que es el selector** (el selector es el nombre del mensaje, en caso de tener argumentos como `at:  put:` es sólo eso, sin `at:1 put:100` sus argumentos). Este *selector* es el nombre que va a buscar en la **clase (C)** para poder ejecutarlo. Formalmente

```
Entrada :  O (objeto al que se le desea enviar un mensaje), S (selector), C(clase en la que se
busca)
Salida :  M (método a ejecutar) o NotUnderstood si no existe
Procedimiento :  si C define un método M para S, devolver M. Si no, sea C' la superclase de C.
Si C' es nil, devolver NotUnderstood, si no, asignar C :0 C' y volver al paso 1
```

En general, cuando se envía un mensaje se utiliza el `method dispatch`, cuando no es porque :

- `Caso Particular :  se envía un mensaje a self, O es el mismo objeto receptor`
- `Excepción:  cuando se envía un mensaje a super, O es el mismo objeto receptor mientras que C es la superc`
  `de la clase del método que contiene el envío de mensaje a super`

3.12 Ejercicio : `streams`

Un stream es un objeto que representa una sucesión infinita. Acepta un mensaje `prox`, que devuelve el elemento actual y avanza al siguiente elemento. Definimos una jerarquía de clases:

```
Object subclass: #Sucesion
Sucesion subclass: #Progresion
Sucesion subclass: #Cons
Sucesion subclass: #Intercalacion
```

Para `progresion desde:  x aplicando:  bloque`

```
Progresion class >> desde: x aplicando: unBloque
    ^ self new initializeWith: x generador: unBloque

Progresion >> initializeWith: x generador: unBloque
    inicial := x.
    generador := unBloque.
    ^ self

Progresion >> prox
    | actual |
    actual := inicial.
    inicial := generador value: actual.
    ^ actual -> self
```

Para el caso de la clase `cons` y el mensaje `cabeza:  unElemento cola:  unStream`

```
Cons class >> cabeza: unElemento cola: unStream
    ^ self new initializeWith: unElemento cola: unStream

Cons >> initializeWith: unaCabeza cola: unaCola
    cabeza := unaCabeza.
    cola := unaCola.
    ^ self

Cons >> prox
    ^ cabeza -> cola
```