# Tecnologías orientadas a objetos

Tarea No. 1

Tomás Balderas Contreras
tbaldera@mail.cs.buap.mx
Coordinación de Ciencias de la Computación
Instituto Nacional de Astrofísica, Óptica y Electrónica

29 de agosto, 2002

# Contenido



## 1 Requisitos para que un lenguaje sea orientado a objetos

Grady Booch propone la siguiente definición de programación orientada a objetos en [1]:

> La *programación orientada a objetos* es un método de implantación en el que los programas se organizan como colecciones cooperativas de objetos, cada uno de los cuales representa una instancia de alguna clase, y cuyas clases son, todas ellas, miembros de una jerarquía de clases unidas mediante relaciones de herencia.

En esta definición distinguimos tres componentes: (1) los *objetos*, (2) las *clases* y (3) la *herencia*. Tomando como base esta definición y opiniones diversas de otros autores establecemos que un lenguaje de programación es orientado a objetos si y solo si satisface los siguientes requerimientos:

- Soporta objetos,
- Los objetos son instancias de una clase,
- Las clases pueden heredar atributos de otras clases (superclases).

Si un lenguaje no soporta la herencia directamente, entonces no es orientado a objetos. A tales lenguajes que no ofrecen mecanismos de herencia se les denomina *basados en objetos*.

## 2 Diferencias entre algunos lenguajes de programación

En esta sección se describen, muy brevemente, algunas diferencias significativas entre los lenguajes de programación Smalltalk, Java y Objective C. Esta descripción considera varios aspectos generales como la definición de clases, la creación de instancias, el envío de mensajes y el polimorfismo.

## 2.1 Descripción

**Smalltalk.** Es un lenguaje orientado a objetos puro, desarrollado por investigadores del Xerox PARC (Palo Alto Research Center). Pasó por un proceso de desarrollo continuo durante la década de los 70 y hasta la fecha ha sido portado a diferentes plataformas por distintas compañías. En sus etapas inciales de desarrollo fue el lenguaje de programación de un completo sistema de cómputo personal concebido por uno de los pioneros de la programación orientada a objetos y de la computación personal, el doctor Alan Kay [2].

**Objective C.** Es un lenguaje orientado a objetos creado por Brad Cox y Tom Love a principios de la década de los 80 como un superconjunto de C, por lo que se considera un lenguaje híbrido. Tiene una gran influencia de Smalltalk. El lenguaje se popularizó cuando fue elegido como herramienta de desarrollo principal del ambiente NeXTSTEP. Actualmente es soportado por ambientes como MacOS X y por compiladores como *gcc* de GNU [5].

**Java.** Es un lenguaje orientado a objetos, creado por James Gosling en los laboratorios de Sun Microsystems. Java se ha vuelto popular porque conjunta y soporta características como independencia de plataforma, interpretación, creación de hilos, etc. Java fue innovador no por definir estos conceptos, que han sido conocidos y empleados desde hace muchos años, sino por reunirlos. Además el lenguaje permite desarrollar toda clase de aplicaciones para la World Wide Web [3].

## 2.2 Sobre las clases

- Cada uno de estos tres lenguajes tiene una jerarquía de clases asociada. Estas jerarquías tienen una clase inicial, la clase *Object*, que define el comportamiento básico de todos los objetos dentro del lenguaje. A pesar de esta similitud el Application Kit, la jerarquía que acompaña a Objective C en el ambiente NeXTSTEP, es muy diferente a la jerarquía que acompaña a Smalltalk y Java, las cuales son a su vez diferentes entre sí.

- La sintaxis para definir clases en cada lenguaje varía mucho. En Smalltalk se lleva a cabo enviando un mensaje a la superclase:

```
Object subclass: nombre de la clase
       instanceVariableNames: 'variables instancia'
       classVariableNames: '...'
       poolDictionaries: '...'
       category: '...'
```

  En Objective C la sintaxis es la siguiente:

```
@interface nombre de la clase: Object
{
  variables instancia
}
@end

- metodo 1;
```

```
- metodo 2;
     :
- metodo n;
```

La sintaxis de Java es similar a la de C++:

```
public class nombre de la clase
{
  variables instancia

  constructor

  public metodo 1
  public metodo 2
         :
  public metodo n
}
```

- El enfoque orientado a objetos fomenta el ocultamiento de información, sin embargo a veces es necesario cambiar el ámbito de los componentes de una clase. Tanto Objective C como Java proporcionan esta flexibilidad permitiendo declarar variables instancia y/o métodos como *private*, *protected* o *public*. Smalltalk, sin embargo, es mas rígido y tanto las variables instancia como la implantación de los métodos de una clase son privados.

## 2.3 Sobre las instancias

- La forma de crear nuevas instancias de una clase en Smalltalk y en Objective C es enviando el mensaje `new` al objeto que representa a dicha clase. En Java el esquema es similar, sin embargo aquí `new` es un operador, no un método.

- Smalltalk es un lenguaje sin tipos, toda variable es un apuntador a un objeto y puede hacer referencia a cualquier clase de objeto en cualquier momento durante la ejecución del programa. Java es un lenguaje con tipos, una variable puede apuntar a objetos de la clase con la que fue declarada o a instancias de subclases de dicha clase. Objective C soporta ambos esquemas.

## 2.4 Sobre los mensajes

La sintaxis para el envío de mensajes en Smalltalk y Objective C es prácticamente la misma. La siguiente tabla muestra las diferentes clases de mensajes en estos lenguajes:

| **Smalltalk** | **Objective C** | **Tipo de mensaje** |
|---|---|---|
| `aButton label.` | `[aButton label];` | Mensaje unitario |
| `aString at:1 put:'s'.` | `[aString at:1 put:'s'];` | Mensaje de palabra clave |
| `pointA + pointB.` | N.A. | Mensaje binario |

Sin embargo la sintaxis de Java difiere de la anterior y es idéntica a la de C++.

Como Smalltalk es un lenguaje puro todo dentro de él está definido en términos de objetos y mensajes entre objetos. Por ejemplo, en este lenguaje tanto las expresiones aritméticas como las estructuras de control están implantadas como mensajes a ciertos objetos, por el contrario en Java y en Objective C estas se expresan en la forma tradicional de los lenguajes procedurales.

### 2.5 Sobre el polimorfismo

Los tres lenguajes soportan polimorfismo. Instancias de diferentes clases pueden responder al mismo mensaje, cada una a su manera. Además es posible redefinir métodos entre una clase y su superclase. Sin embargo existe una sutil diferencia que tiene que ver con la *sobrecarga de operadores*. De manera natural es posible en Smalltalk asignar cualquiera de los operadores `+`, `-`, `*` y `/` como selector o nombre de un método, que es una característica no presente en Objective C.

## 3 Resumen de artículos

Esta sección resume el contenido del artículo *Do Object-Oriented Languages Need Special Hardware Support?* escrito por Urs Hölzle y David Ungar [4].

Por muchos años han existido implantaciones de sistemas orientados a objetos que dependen en gran medida del soporte que les proporciona el hardware, en especial la arquitectura del microprocesador. Al mismo tiempo ha habido estudios que establecen que el diseño de hardware especial para los sistemas orientados a objetos incrementa su rendimiento. Sin embargo el estudio reportado en este artículo emplea el lenguaje Self para demostrar que, empleando compiladores optimizadores de código, un programa escrito en un lenguaje orientado a objetos puede tener un rendimiento similar al de aplicaciones en C empleando hardware normal. Los motivos para basar el estudio en el lenguaje Self son los siguientes:

- Es un lenguaje orientado a objetos completamente puro. Si el comportamiento de un programa compilado desde un lenguaje orientado a objetos fuera diferente del comportamiento de un programa compilado desde un lenguaje procedural entonces un programa en Self exhibiría esta característica mucho mas que un programa escrito en un lenguaje híbrido.

- Los programas escritos en Self son compilados empleando varias técnicas de optimización de código, por lo que tienen un mejor rendimiento que los programas escritos en Smalltalk y no son tan lentos como los programas compilados desde C.

Uno de los objetivos de este documento es comparar la ejecución de programas compilados desde Self con la ejecución de programas compilados desde C y C++. Las herramientas y recursos usados para realizar las mediciones son:

- Un conjunto de aplicaciones escritas en Self de gran y mediano tamaño en una variedad de estilos de programación y de dominios de aplicación,

- Un conjunto de aplicaciones escritas en C para valores enteros (benchmarks SPECint89),

- Un simulador de conjuntos de instrucciones (*Shade*),
- Un simulador de memoria caché (*Dinero*).

Todos los programas son compilados para la arquitectura SPARC V8 [6], que es la arquitectura simulada por *Shade*.

La distribución de los distintos tipos de instrucciones en los programas en SELF es muy similar a la de los programas en SPECint89, también en ambos casos el tamaño de los bloques básicos es virtualmente idéntico. Este artículo reporta que las instrucciones especializadas de la arquitectura SPARC, como las instrucciones aritméticas con etiqueta (`taddcc`, `tsubcc`, etc.), incrementan el rendimiento de los programas solo en un factor del 2% o 3%.

Debido a que la invocación de métodos ocurre muy frecuentemente en los lenguajes puros seguramente el soporte de hardware para esta operación acelerará la ejecución. Sin embargo el costo involucrado en el despacho de un mensaje consiste en varias instrucciones que llevan a cabo operaciones muy simples que no requieren de hardware de propósito específico. Este costo puede reducirse empleando optimizaciones de código en el compilador.

Las mediciones de las frecuencias de fallos de la memoria caché de instrucciones, para varios valores del tamaño de la memoria, muestran que los fallos son mas frecuentes en los programas escritos en SELF que en el programa mas grande del conjunto SPECint89, el compilador *gcc*. Esta diferencia muestra que la propiedad de localidad de referencia temporal no tiene tanta influencia en los programas orientados a objetos como en los programas en C. Sin embargo, como es de esperar, ocurre que la frecuencia de fallos disminuye para todos los programas conforme aumenta el tamaño de la memoria caché para instrucciones. Con respecto al comportamiento de la memoria caché de datos se encontró que la frecuencia de fallos de la memoria tanto de lectura como de escritura es baja.

Como trabajo a futuro se propone investigar el impacto de la ejecución de instrucciones en forma superescalar, fuera de orden, predicción de saltos y otros avances arquitecturales genéricos.

# 4 Métodos y procedimientos

Dependiendo del lenguaje la invocación de un método puede ser una actividad dinámica o estática, en este caso no diferiría de una llamada a un procedimiento, como sucede, en muchas ocasiones, con lenguajes con tipos como C++ y Java. En estos casos si un objeto es declarado como una instancia de una clase en particular entonces cualquier mensaje enviado al objeto produce una simple llamada al procedimiento que implanta la respuesta, puesto que se conoce el tipo del objeto en tiempo de compilación. Este esquema permite además la detección temprana de errores. En lenguajes sin tipos esto no sucede, la invocación de un método ocurre en tiempo de ejecución ya que es necesario determinar primero la clase de la instancia y después buscar el método, de ser necesario a lo largo de la jerarquía de clases.

El uso del polimorfismo tiene como consecuencia, en todos los casos, que existan dos o mas procedimientos que pueden ser invocados como respuesta a un mensaje, la elección del procedimiento correcto depende de la clase del objeto receptor del mensaje y se puede realizar ya sea en tiempo de compilación o de ejecución.

## Referencias


[1] Booch, G. 1996. *Análisis y diseño orientado a objetos con aplicaciones*. Segunda edición. México: Addison-Wesley Longman.

[2] Goldberg, A. y D. Robson. 1985. *Smalltalk-80: the language and its implementation*. Reading, Massachusetts: Addison-Wesley Publishing Company.

[3] Gosling, J., B. Joy, G. Steele y G. Bracha. 2000. *The Java$^{TM}$ language specification*. Segunda edición. Reading, Massachusetts: Addison-Wesley Publishing Company.
`http://java.sun.com`

[4] Hölzle, U. y D. Ungar. 1995. Do Object-Oriented Languages Need Special Hardware Support?. En *Proceedings of the 9th European Conference on Object Oriented Programming*. ECOOP'95. ed. W. Olthoff. 283–302.

[5] NeXT Computer, Inc. 1993. *The Objective C language*. Reading, Massachusetts: Addison-Wesley Publishing Company.

[6] SPARC International, Inc. 1992. *The SPARC Architecture Manual. Version 8*. New Jersey: Prentice-Hall, Inc.
`http://www.sparc.org`