1: Introducción a los objetos

La génesis de la revolución de los computadores se encontraba en una máquina, y por ello, la génesis de nuestros lenguajes de programación tiende a parecerse a esa máquina.

Pero los computadores, más que máquinas, pueden considerarse como herramientas que permiten ampliar la mente ("bicicletas para la mente", como se enorgullece de decir Steve Jobs), además de un medio de expresión inherentemente diferente. Como resultado, las herramientas empiezan a parecerse menos a máquinas y más a partes de nuestra mente, al igual que ocurre con otros medios de expresión como la escritura, la pintura, la escultura, la animación o la filmación de películas. La programación orientada a objetos (POO) es una parte de este movimiento dirigido a utilizar los computadores como si de un medio de expresión se tratara.

Este capítulo introducirá al lector en los conceptos básicos de la POO, incluyendo un repaso a los métodos de desarrollo. Este capítulo y todo el libro, toman como premisa que el lector ha tenido experiencia en algún lenguaje de programación procedural (por procedimientos), sea C u otro lenguaje. Si el lector considera que necesita una preparación mayor en programación y/o en la sintaxis de C antes de enfrentarse al presente libro, se recomienda hacer uso del CD ROM formativo *Thinking in C: Foundations for C++ and Java*, que se adjunta con el presente libro, y que puede encontrarse también la URL, *http://www.BruceEckel.com*.

Este capítulo contiene material suplementario, o de trasfondo (*background*). Mucha gente no se siente cómoda cuando se enfrenta a la programación orientada a objetos si no entiende su contexto, a grandes rasgos, previamente. Por ello, se presentan aquí numerosos conceptos con la intención de proporcionar un repaso sólido a la POO. No obstante, también es frecuente encontrar a gente que no acaba de comprender los conceptos hasta que tiene acceso a los mecanismos; estas personas suelen perderse si no se les ofrece algo de código que puedan manipular. Si el lector se siente identificado con este último grupo, estará ansioso por tomar contacto con el lenguaje en sí, por lo que debe sentirse libre de saltarse este capítulo —lo cual no tiene por qué influir en la comprensión que finalmente se adquiera del lenguaje o en la capacidad de escribir programas en él mismo. Sin embargo, tarde o temprano tendrá necesidades ocasionales de volver aquí, para completar sus nociones en aras de lograr una mejor comprensión de la importancia de los objetos y de la necesidad de comprender cómo acometer diseños haciendo uso de ellos.

El progreso de la abstracción

Todos los lenguajes de programación proporcionan abstracciones. Puede incluso afirmarse que la complejidad de los problemas a resolver es directamente proporcional a la clase (tipo) y calidad de las abstracciones a utilizar, entendiendo por tipo "clase", *aquello que se desea abstraer*. El lenguaje

ensamblador es una pequeña abstracción de la máquina subyacente. Muchos de los lenguajes denominados "imperativos" desarrollados a continuación del antes mencionado ensamblador (como Fortran, BASIC y C) eran abstracciones a su vez del lenguaje citado. Estos lenguajes supusieron una gran mejora sobre el lenguaje ensamblador, pero su abstracción principal aún exigía pensar en términos de la estructura del computador más que en la del problema en sí a resolver. El programador que haga uso de estos lenguajes debe establecer una asociación entre el modelo de la máquina (dentro del "espacio de la solución", que es donde se modela el problema, por ejemplo, un computador) y el modelo del problema que de hecho trata de resolver (en el "espacio del problema", que es donde de hecho el problema existe). El esfuerzo necesario para establecer esta correspondencia, y el hecho de que éste no es intrínseco al lenguaje de programación, es causa directa de que sea difícil escribir programas, y de que éstos sean caros de mantener, además de fomentar, como efecto colateral (lateral), toda una la industria de "métodos de programación".

La alternativa al modelado de la máquina es modelar el problema que se trata de resolver. Lenguajes primitivos como LISP o APL eligen vistas parciales o particulares del mundo (considerando respectivamente que los problemas siempre se reducen a "listas" o a "algoritmos"). PROLOG convierte todos los problemas en cadenas de decisiones. Los lenguajes se han creado para programación basada en limitaciones o para programar exclusivamente mediante la manipulación de símbolos gráficos (aunque este último caso resultó ser excesivamente restrictivo). Cada uno de estos enfoques constituye una solución buena para determinadas clases (tipos) de problemas (aquéllos para cuya solución fueron diseñados), pero cuando uno trata de sacarlos de su dominio resultan casi impracticables.

El enfoque orientado a objetos trata de ir más allá, proporcionando herramientas que permitan al programador representar los elementos en el espacio del problema. Esta representación suele ser lo suficientemente general como para evitar al programador limitarse a ningún tipo de problema específico. Nos referiremos a elementos del espacio del problema, denominando "objetos" a sus representaciones dentro del espacio de la solución (por supuesto, también serán necesarios otros objetos que no tienen su análogo dentro del espacio del problema). La idea es que el programa pueda autoadaptarse al lingo del problema simplemente añadiendo nuevos tipos de objetos, de manera que la mera lectura del código que describa la solución constituya la lectura de palabras que expresan el problema. Se trata, en definitiva, de una abstracción del lenguaje mucho más flexible y potente que cualquiera que haya existido previamente. Por consiguiente, la POO permite al lector describir el problema en términos del propio problema, en vez de en términos del sistema en el que se ejecutará el programa final. Sin embargo, sigue existiendo una conexión con el computador, pues cada objeto puede parecer en sí un pequeño computador; tiene un estado, y se le puede pedir que lleve a cabo determinadas operaciones. No obstante, esto no quiere decir que nos encontremos ante una mala analogía del mundo real, al contrario, los objetos del mundo real también tienen características y comportamientos.

Algunos diseñadores de lenguajes han dado por sentado que la programación orientada a objetos, de por sí, no es adecuada para resolver de manera sencilla todos los problemas de programación, y hacen referencia al uso de lenguajes de programación *multiparadigma*[1].

[1] N. del autor: Ver *Multiparadigm Programming in Leda*, por Timothy Budd (Addison-Wesley, 1995).

Alan Kay resumió las cinco características básicas de Smalltalk, el primer lenguaje de programación orientado a objetos que tuvo éxito, además de uno de los lenguajes en los que se basa Java. Estas características constituyen un enfoque puro a la programación orientada a objetos:

1. **Todo es un objeto**. Piense en cualquier objeto como una variable: almacena datos, permite que se le "hagan peticiones", pidiéndole que desempeñe por sí mismo determinadas operaciones, etc. En teoría, puede acogerse cualquier componente conceptual del problema a resolver (bien sean perros, edificios, servicios, etc.) y representarlos como objetos dentro de un programa.

2. **Un programa es un cúmulo de objetos que se dicen entre sí lo que tienen que hacer mediante el envío de mensajes**. Para hacer una petición a un objeto, basta con "enviarle un mensaje". Más concretamente, puede considerarse que un mensaje en sí es una petición para solicitar una llamada a una función que pertenece a un objeto en particular.

3. **Cada objeto tiene su propia memoria, constituida por otros objetos**. Dicho de otra manera, uno crea una nueva clase de objeto construyendo un paquete que contiene objetos ya existentes. Por consiguiente, uno puede incrementar la complejidad de un programa, ocultándola tras la simplicidad de los propios objetos.

4. **Todo objeto es de algún tipo**. Cada objeto es *un elemento de una clase*, entendiendo por "clase" un sinónimo de "tipo". La característica más relevante de una clase la constituyen "el conjunto de mensajes que se le pueden enviar".

5. **Todos los objetos de determinado tipo pueden recibir los mismos mensajes**. Ésta es una afirmación de enorme trascendencia como se verá más tarde. Dado que un objeto de tipo "círculo" es también un objeto de tipo "polígono", se garantiza que todos los objetos "círculo" acepten mensajes propios de "polígono". Esto permite la escritura de código que haga referencia a polígonos, y que de manera automática pueda manejar cualquier elemento que encaje con la descripción de "polígono". Esta capacidad de *suplantación* es uno de los conceptos más potentes de la POO.

Todo objeto tiene una interfaz

Aristóteles fue probablemente el primero en estudiar cuidadosamente el concepto de *tipo*; hablaba de "la clase de los pescados o la clase de los peces". La idea de que todos los objetos, aún siendo únicos, son también parte de una clase de objetos, todos ellos con características y comportamientos en común, ya fue usada en el primer lenguaje orientado a objetos, Simula-67, que ya incluye la palabra clave **clase**, que permite la introducción de un nuevo tipo dentro de un programa.

Simula, como su propio nombre indica, se creó para el desarrollo de simulaciones, como el clásico del cajero de un banco, en el que hay cajero, clientes, cuentas, transacciones y unidades monetarias —un montón de "objetos". Todos los objetos que, con excepción de su estado, son idénticos durante la ejecución de un programa se agrupan en "clases de objetos", que es precisamente de donde proviene la palabra clave **clase**. La creación de tipos abstractos de datos (clases) es un concepto fun-

damental en la programación orientada a objetos. Los tipos abstractos de datos funcionan casi como los tipos de datos propios del lenguaje: es posible la creación de variables de un tipo (que se denominan *objetos* o *instancias* en el dialecto propio de la orientación a objetos) y manipular estas variables (mediante el *envío* o *recepción de mensajes*; se envía un mensaje a un objeto y éste averigua qué debe hacer con él). Los miembros (elementos) de cada clase comparten algunos rasgos comunes: toda cuenta tiene un saldo, todo cajero puede aceptar un ingreso, etc. Al mismo tiempo, cada miembro tiene su propio estado, cada cuenta tiene un saldo distinto, cada cajero tiene un nombre. Por consiguiente, los cajeros, clientes, cuentas, transacciones, etc. también pueden ser representados mediante una entidad única en el programa del computador. Esta entidad es el objeto, y cada objeto pertenece a una clase particular que define sus características y comportamientos.

Por tanto, aunque en la programación orientada a objetos se crean nuevos tipos de datos, virtualmente todos los lenguajes de programación orientada a objetos hacen uso de la palabra clave "clase". Siempre que aparezca la palabra clave "tipo" (*type*) puede sustituirse por "clase" (*class*) y viceversa[2].

Dado que una clase describe a un conjunto de objetos con características (datos) y comportamientos (funcionalidad) idénticos, una clase es realmente un tipo de datos, porque un número en coma flotante, por ejemplo, también tiene un conjunto de características y comportamientos. La diferencia radica en que un programador define una clase para que encaje dentro de un problema en vez de verse forzado a utilizar un tipo de datos existente que fue diseñado para representar una unidad de almacenamiento en una máquina. Es posible extender el lenguaje de programación añadiendo nuevos tipos de datos específicos de las necesidades de cada problema. El sistema de programación acepta las nuevas clases y las cuida, y asigna las comprobaciones que da a los tipos de datos predefinidos.

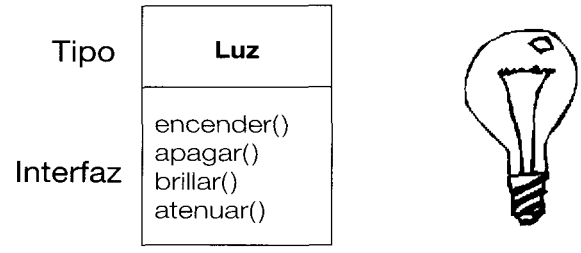
El enfoque orientado a objetos no se limita a la construcción de simulaciones. Uno puede estar de acuerdo o no con la afirmación de que todo programa es una simulación del sistema a diseñar, mientras que las técnicas de POO pueden reducir de manera sencilla un gran conjunto de problemas a una solución simple.

Una vez que se establece una clase, pueden construirse tantos objetos de esa clase como se desee, y manipularlos como si fueran elementos que existen en el problema que se trata de resolver. Sin duda, uno de los retos de la programación orientada a objetos es crear una correspondencia uno a uno entre los elementos del espacio del problema y los objetos en el espacio de la solución.

Pero, ¿cómo se consigue que un objeto haga un trabajo útil para el programador? Debe de haber una forma de hacer peticiones al objeto, de manera que éste desempeñe alguna tarea, como completar una transacción, dibujar algo en la pantalla o encender un interruptor. Además, cada objeto sólo puede satisfacer determinadas peticiones. Las peticiones que se pueden hacer a un objeto se encuentran definidas en su *interfaz*, y es el tipo de objeto el que determina la interfaz. Un ejemplo sencillo sería la representación de una bombilla:

---

[2] Algunas personas establecen una distinción entre ambos, remarcando que un tipo determina la interfaz, mientras que una clase es una implementación particular de una interfaz.




```
Luz lz = new Luz();
lz.encender();
```

La interfaz establece *qué* peticiones pueden hacerse a un objeto particular. Sin embargo, debe hacer código en algún lugar que permita satisfacer esas peticiones. Éste, junto con los datos ocultos, constituye la *implementación*. Desde el punto de vista de un lenguaje de programación procedural, esto no es tan complicado. Un tipo tiene una función asociada a cada posible petición, de manera que cuando se hace una petición particular a un objeto, se invoca a esa función. Este proceso se suele simplificar diciendo que se ha "enviado un mensaje" (hecho una petición) a un objeto, y el objeto averigua qué debe hacer con el mensaje (ejecuta el código).

Aquí, el nombre del tipo o clase es **Luz**, el nombre del objeto **Luz** particular es **lz**, y las peticiones que pueden hacerse a una **Luz** son encender, apagar, brillar o atenuar. Es posible crear una **Luz** definiendo una "referencia" (**lz**) a ese objeto e invocando a **new** para pedir un nuevo objeto de ese tipo. Para enviar un mensaje al objeto, se menta el nombre del objeto y se conecta al mensaje de petición mediante un punto. Desde el punto de vista de un usuario de una clase predefinida, éste es el no va más de la programación con objetos.

El diagrama anteriormente mostrado sigue el formato del Lenguaje de Modelado Unificado o *Unified Modeling Language* (UML). Cada clase se representa mediante una caja, en la que el nombre del tipo se ubica en la parte superior, cualquier dato necesario para describirlo se coloca en la parte central, y las *funciones miembro* (las funciones que pertenecen al objeto) en la parte inferior de la caja. A menudo, solamente se muestran el nombre de la clase y las funciones miembro públicas en los diagramas de diseño UML, ocultando la parte central. Si únicamente interesan los nombres de las clases, tampoco es necesario mostrar la parte inferior de la caja.

La implementación oculta

Suele ser de gran utilidad descomponer el tablero de juego en *creadores de clases* (elementos que crean nuevos tipos de datos) y *programadores clientes*[3] (consumidores de clases que hacen uso de los tipos de datos en sus aplicaciones). La meta del programador cliente es hacer uso de un gran repertorio de clases que le permitan acometer el desarrollo de aplicaciones de manera rápida. La

[3] Nota del autor: Término acuñado por mi amigo Scott Meyers.