Programación Orientada a Objetos (POO)

Introducción

La *programación orientada a objetos* es el paradigma en el cual los elementos de primer orden son los objetos.

Como *paradigma* es un derivado de la programación estructurada, por el cual se logró que los datos y los métodos que manipulaban esos datos se mantengan juntos en una unidad llamada **objeto**. De esta manera se propuso impedir el manejo de datos *salvo mediante llamadas a los métodos del objeto contenedor de los mismos.*
La localidad de los datos es la consecuencia ventajosa obvia, pero la más importante es el componente añadido de mantenibilidad, ya que puede cambiarse la estructura interna de un objeto sin afectar a aquellos que interactúen con él - siempre y cuando no se afecte su interfaz pública.

Objetos

El término 'objeto', como dice Meyer en *Touch of Class*, es uno muy vago y vulgar. En nuestro idioma sólo hay una forma más vaga y vulgar de referirse a los objetos, y es mediante la palabra 'cosa'.
Dado este pequeño contexto es que afirmamos que los objetos tienen la particularidad de poder amoldarse al concepto que se desee representar, sea éste de cualquier índole. No necesita existir en el mundo real, ser tangible, ni ser pequeño, o demasiado grande, y al poder tomarnos todas esas licencias al momento de idearlos es como encontramos uno de los potenciales del paradigma, *la programación orientada a objetos.*

En otras palabras, estamos ante la posibilidad de crear y dar forma a cualquier concepto que imaginemos, traduciéndolo a la apariencia de 'objetos' para poder utilizarlos como *pequeñas máquinas de software que fundamentalmente saben cosas, y saben hacer cosas.*

Nota: En rigor, a efectos de pensar y utilizar los objetos con los que nos encontremos (o diseñemos) sólamente debe importarnos **qué cosas pueden hacer los objetos por nosotros**, y dejamos de lado (por el momento o definitivamente, según el caso) el conocimiento de las cosas que 'saben' los objetos.

Miembros

Lo que hace a un objeto, en palabras de Meyer en *Touch of Class*, no es su contrapartida en el mundo físico, sino que podemos manipularlo a través de un grupo de operaciones bien definidas a las cuales llamamos miembros.

Los miembros de un objeto se invocan por medio de mensajes, y suele ser común confundir mensajes con miembros. Haremos una tosca pero práctica distinción:

- **Miembro** es cada una de las posibilidades de manipulación que nos proporciona un objeto. Se evidencia en tiempo de codificación.
- **Mensaje** es la invocación efectiva de esos miembros. Se evidencia en tiempo de ejecución.

Mensajes

En el año 1967 Ole-Johan Dahl y Kristen Nygaard lanzaron el lenguaje de programación Simula 67, el ancestro común de todos los lenguajes de programación orientada a objetos.

Una de las principales innovaciones que introdujeron a la anterior programación estructurada fue convertir esto

```
f(o,x)
```

en esto

```
o.f(x)
```

Lenguajes como Python toman de modo indistinto ambas invocaciones, pero éstos han llegado mucho tiempo después. La semántica introducida es el componente que cambia las reglas del juego.

> En el caso de `f(o,x)` estamos diciendo "realizar la función f con o y x". En cambio con `o.f(x)` estamos diciendo "objeto o, hacé f con x". Y aquí hemos cambiado un paradigma.

Para terminar de marcar la diferencia, Alan Kay definió la programación orientada a objetos de la siguiente manera:

> POO para mí significa sólo mensajería, retención local y protección y ocultamiento de estado y procesamiento, y vinculación extremadamente tardía de todas las cosas.

El rol que Kay le brinda al mensaje es sustancial: a partir de la programación orientada a objetos desacoplamos la dependencia entre funciones para pasar a depender de mensajes, que estratégicamente implementados nos derivan a los objetos adecuados sin necesidad de ejercer control directo.

> Cuando pasamos un mensaje, perdemos el control de quién lo va a interpretar. Sólo podemos esperar que el receptor reaccione apropiadamente. Ni el que envía depende del que recibe, ni viceversa.

Complicado como parece, simplemente estamos diciendo que bajamos el acoplamiento aumentando la mantenibilidad.

En las notas de la materia Object Oriented Programming de la Universidad de KwaZulu-Natal podemos ver de un modo muy claro cómo debe interpretarse el pasaje de mensajes, en el sentido práctico:

> La acción en la programación orientada a objetos se inicia por la transmisión de un mensaje a un agente (un objeto) responsable por las acciones. El mensaje codifica el pedido de una acción y se acompaña por cualquier información adicional (argumentos/parámetros) necesaria para llevar adelante el pedido. El receptor es el objeto al cual el mensaje está dirigido. Si éste acepta el mensaje, está aceptando la responsabilidad de llevar adelante la acción indicada. En respuesta a un mensaje, el receptor llevará a cabo un método para satisfacer el pedido.

Los mensajes propiamente dichos suelen dividirse en dos tipos: **consultas** y **comandos**.

Consultas

Los miembros que nos permiten obtener propiedades de un objeto se denominan consultas: estamos *consultando* al objeto por una propiedad en particular.

> Según el **principio de acceso uniforme**, todos los servicios ofrecidos por un módulo deben estár disponibles por medio de una notación uniforme, que no devele si son implementadas mediante almacenamiento o cálculo de lo requerido.

Tomaremos por ejemplo el siguiente caso:

```
class Persona {
  Integer calcularEdad() { ... }
  Date getFechaNacimiento() { ... }
}
```

Nos damos cuenta que traicionamos el ocultamiento de información: se sabe a ciencia cierta que lo que se almacena es la fecha de nacimiento, y que la edad se calcula.

Respetando el principio de acceso uniforme, deberíamos hacerlo de la siguiente manera:

```
class Persona {
  Integer getEdad() { ... }
  Date getFechaNacimiento() { ... }
}
```

A los efectos de los clientes de nuestras clases, no es importante saber qué dato se almacena y cuál se calcula en tanto y en cuanto se cumpla con la responsabilidad requerida.

Para devolver los valores las consultas pueden requerir de ciertos parámetros adicionales. Lo importante es que las consultas **no alteran el estado de los objetos**, por lo que eso las define.

Comandos

Los comandos son peticiones de acción hacia un objeto. Son la contrapartida de las consultas ya que se caracterizan por **modificar el estado de los objetos**. Un ejemplo podría ser el siguiente:

```
class Auto {
  void apagarLuces() { ... }
  void encenderLuces() { ... }
}
```

Como podemos ver, es evidente que algo cambiará en el estado interno del `Auto` para que ésto pueda llevarse a cabo. En un análisis más sutil, vemos que el cambio genera que la consecutiva invocación de uno de los miembros deje de tener sentido: *no sirve apagar una luz apagada.*

Ese cambio de estado distingue a los comandos, y nos alerta de que en diferentes instantes de tiempo las invocaciones a consultas o comandos puedan brindar resultados diferentes.

En una frase, el resultado de consultas y comandos siempre depende del estado del objeto, aunque no siempre será modificado.

Clases

Cuando nos referimos a objetos estamos contando a los individuos de un universo repleto de variedad: al hablar de una pelota, no hablamos del concepto general de pelota, sino de una pelota en particular. Será mi pelota, o puede ser otra: cada una con características particulares.

Lo mismo con los individuos: cuando nos referimos a cierta persona estamos englobando en esa distinción un montón de características que permiten diferenciarla del resto. Juan es alto, castaño y tiene barba. Mirta, en cambio, es delgada, pequeña y rubia. Y no tiene barba.

Ese conjunto de características son las que definen el tipo de entidad del que estamos hablando: distinguimos los individuos entre sí por la variabilidad de sus características, pero al referirnos al concepto genérico y colectivo estamos hablando de características sin pensar en ninguna en particular: toda persona tiene altura, aunque no sabemos cuánta hasta pensar en una en particular.

Esta distinción entre individuo y colectivo es la que debemos hacer al pensar en objetos y clases: los objetos son los individuos, y mediante una abstracción de características obtenemos las clases.

Las clases, en un sentido muy simplista, nos permiten tener un molde para crear objetos a partir de ellas. Imaginemos que una clase es una plantilla que posee algunos huecos para completar a gusto: los completaremos cuando apliquemos la plantilla para crear un nuevo objeto.

Las clases tienen tres características: nombre, atributos y operaciones. Cada individuo (que posee identidad) de determinada clase dará valores a esos atributos y comportamiento efectivo a esas operaciones.

Al definir una clase damos la posibilidad de la existencia de los objetos:

```
public class Estudiante {
  // atributos, constructor, métodos
}
```

Nos permite escribir:

```
Estudiante unEstudiante = new Estudiante("Juan");
```

> En esa simple sentencia tenemos la relación que existe entre un objeto y una clase. "Un estudiante es una nueva instancia de Estudiante, en este caso llamado Juan".

Los objetos tienen identidad y un estado interno particular. La clase no tiene identidad, ya que es un molde para todos los objetos, ni tiene valores sino "espacios" preparados para adquirirlos.
Adicionalmente, si bien los objetos son los que reciben los mensajes y ejecutan acciones, las clases imponen qué instrucciones deberán ejecutarse y de qué modo en cada caso.

Resumen

Suele decirse que la POO se basa en modelar conceptos del mundo real en forma de software, y en parte es un encunciado correcto. Sin embargo, la programación en general modela conceptos del mundo real en software, ya que para eso ha sido desarrollada.

También suele decirse que se define la POO por medio de sus pilares (que los veremos más adelante). Utilizar estos mecanismos es parte de la POO, pero no su esencia: podemos obtener software estructurado y procedimental utilizando las ventajas de los objetos, desperdiciando grandes oportunidades.

La esencia de la POO son los mensajes, y cómo serán interpretados cuando alcancen su destinatario. El cambio de dependencias en el flujo del control es lo que proporciona la ventaja fundamental de la orientación a objetos.

Estudiaremos a lo largo de estos capítulos los conceptos principales de la Programación Orientada a Objetos.

Bibliografía recomendada

- Bertrand Meyer. 1997. Object-Oriented Software Construction (2nd Ed.). Prentice-Hall, Inc., Upper Saddle River, NJ, USA.
- Bertrand Meyer. 2009. Touch of Class: Learning to Program Well with Objects and Contracts (1st Ed.). Springer Publishing Company, Incorporated.

Links de interés

- "Object-oriented programming", *Wikipedia, The Free Encyclopedia*. Wikimedia Foundation, 3 de Agosto de 2013 (accedido el 27 de junio de 2012)
- "Notes for the Computer Science Module Object Oriented Programming", *math.hws.edu*. Febrero de 2007 (accedido el 9 de abril de 2013)
- "Introduction to Object Oriented Programming Concepts and More", Nirosh L.W.C. *codeproject.com*. 25 de enero de 2011 (accedido el 9 de abril de 2013)

Por Lucas Videla. Versión original disponible en delucas/notas