# Programación Orientada a Aspectos

## Overview

Hay ciertos asuntos que no pertenecen al entorno de una aplicación pero deben ser tratados.

En la programación orientada a objetos, son los objetos quienes deben tratar estos asuntos,

es decir, hacer tareas que no le corresponden.

Es allí donde entra la programación orientada a aspectos, que es la que se encarga de tratar

estos asuntos de dejando que los objetos se dediquen solamente a la funcionalidad de la aplicación.

A continuación, se muestra un resumen de cómo afectan estos asuntos al desarrollo de aplicaciones:

* Baja cohesión: las clases realizan tareas que no le corresponden específicamente a ellas.
* Baja reusabilidad: Cuando una clase trata un asunto que no le corresponde, esta clase probablemente sea difícil de volver a usar en otro sistema.
* Peor calidad de código: El código se complica, y cuesta más entenderlo y mantenerlo.
* Evolución dificultosa: Se vuelve más costoso que el sistema evoluciones al tener código disperso y enredado por toda la aplicación.

## Importancia de POA

En esta parte explicaremos un ejemplo muy práctico a modo de ilustrar la importancia de la Programación Orientada a Aspectos. Para ser más didácticos se hará una implementación con la ya conocida Programación Orientada a Objetos que está realizada en java.

Se tiene una aplicación que simula las transacciones básicas de un banco, depósito, retiro, y transferencias de dinero, de primer momento se tendrá una clase con nombre es Cuenta, esta clase tiene un atributo llamado saldo que será de tipo **double** que indica el saldo en la cuenta. Además tendremos tres métodos: Uno llamado **hacerDeposito()** con un único parámetro de tipo **double** que indica la cantidad a depositar en la cuenta. Otro llamado hacerRetiro() que cuenta con un solo parámetro que indica la cantidad de dinero que se quiere extraer de la cuenta. Y el último llamado **hacerTransferencia()** que cuenta con dos parámetros, uno indica la cantidad de dinero y el otro es la cuenta a la que se quiere hacer la transferencia.

public void hacerDeposito(Double cantidad) {

if(cantidad <= 0) {

System.out.println("No se puede realizar esta transaccion por falta de dinero");

}

else {

Date hora = new Date();

SimpleDateFormat formatoDeFecha = new SimpleDateFormat("yyyy.MM.dd G 'at' HH:mm:ss");

this.saldo += cantidad;

System.out.println("Movimiento realizado a las " + formatoDeFecha.format(hora)) ;

}

}

public void hacerRetiro(Double cantidad) {

if(this.saldo < cantidad) {

System.out.println("No hay fondos suficientes para esta transaccion.");

}

else {

Date hora = new Date();

SimpleDateFormat formatoDeFecha = new SimpleDateFormat("yyyy.MM.dd G 'at' HH:mm:ss");

this.saldo -= cantidad;

System.out.println("Movimiento realizado a las " + formatoDeFecha.format(hora)) ;

}

}

Examinando un poco este primer intento donde se implementan los tres métodos observamos que comienzan con una verificación de la cantidad que se quiere extraer, depositar o transferir.

Esta verificación es necesaria es necesaria debido a que sería un error permitir depositar una cantidad negativa de dinero o extraer una cantidad mayor del dinero que se dispone en dicha cuenta. Por lo tanto evitamos estas situaciones.

Una vez que se comprueban las cantidades y que está todo bien, se procede a realizar el depósito o la extracción o la transferencia. Pero para esto como es normal en todo sistema donde se realicen transacciones bancarias necesitamos tener un registro de cada movimiento realizado en cada cuenta. Por eso creamos una variable llamada hora y la mostramos en pantalla.

Lo resaltante de todo esto y que tenemos que tener en cuenta es que los tres métodos repiten el proceso de registrar un movimiento. Lo cual es una mala práctica. No deberíamos de repetir código. Alguna solución a esto podría ser creando un cuarto método al que podríamos llamar **registrarMovimiento()** dentro de la misma clase. Este método se encargaría sólo de la parte del registro del movimiento por lo tanto cada uno de los tres métodos anteriores lo llamaría cuando ha concluido un movimiento. Tendríamos entonces lo siguiente:

private void registrarMovimiento() {

Date hora = new Date();

SimpleDateFormat formatoDeFecha = new SimpleDateFormat("yyyy.MM.dd G 'at' HH:mm:ss");

System.out.println("Movimiento realizado a las " + formatoDeFecha.format(hora)) ;

}

Entonces las funciones quedarían de la siguiente manera :

public void hacerRetiro(Double cantidad) {

if(this.saldo < cantidad) {

System.out.println("No hay fondos suficientes para esta transaccion.");

}

else {

this.saldo -= cantidad;

registrarMovimiento();

}

}

Y se haría lo mismo para las otras funciones de retiro y depósito.

Lo malo de esta solución sería si se diera el caso que se efectuará otro movimiento que no fuera dentro de la clase Cuenta sino dentro de otra clase, esta clase tendría que tener su propio método **registrarMovimiento()**, y otra vez estaríamos repitiendo código pero entre clases. Debido a esto podría surgir otra posible solución, creamos una clase que se encargue específicamente de registrar estos movimientos; por lo tanto cualquier método de cualquier clase puede instanciarla y llamar al método que registra movimientos. Pero aunque tengamos una clase aparte que se encargue de registrar movimientos tendríamos que cada clase debe instanciar y llamar al método de esa clase. Es decir, seguimos repitiendo código.

El registro debería hacerse aparte, no debería ser responsabilidad de un depósito, o extracción o una transferencia. Esta forma de registrar un movimiento, ensucia el código que se refiere a los métodos de la clase Cuenta. Lo ideal sería que estos métodos se vieran así:

public void hacerDeposito(Double cantidad) {

if(cantidad <= 0) {

System.out.println("No se puede realizar esta transaccion por falta de dinero");

}

else {

this.saldo += cantidad;

}

}

public void hacerRetiro(Double cantidad) {

if(this.saldo < cantidad) {

System.out.println("No hay fondos suficientes para esta transaccion.");

}

else {

this.saldo -= cantidad;

}

}

Un aspecto, entonces, es algo que entrecruza una o más clases. Dicho de otra forma, un aspecto es un concepto que ofrece funcionalidad transversal a los objetos del negocio.

Para su implementación se realizará con java con su complemento orientado a objetos, Aspect Java.

Para ver de una manera más clara un Aspecto, hay que imaginarlo como algo que vigila un programa a medida que este se ejecuta y a partir de ciertas acciones que se producen en el programa, este aspecto realiza acciones propias de él. Por ejemplo podemos decirle a un aspecto , que cuando se termine de ejecutar cualquier método que comience con la palabra “hacer” realice un registro de movimiento. Entonces la clase Cuenta de nuestro ejemplo , no se daría cuenta de la existencia de este aspecto nunca, pero este aspecto está atento a las ejecuciones **hacerAlgo..()** de la clase. Definimos el siguiente aspecto para poder visualizar mejor lo que estamos tratando decir:

public aspect RegistroDeMovimientos {

after():execution (public \* hacer\*(..)) {

Date hora = new Date();

SimpleDateFormat formatoDeFecha = new SimpleDateFormat("yyyy.MM.dd G 'at' HH:mm:ss");

System.out.println("Movimiento realizado a las " + formatoDeFecha.format(hora)) ;

}

}

Observamos este código y lo primero que vemos es la declaración del aspecto bajo el nombre de RegistroDeMovimientos. Por ende al igual que las clases de Java el archivo del aspecto debe llamarse RegistroDeMovimientos.aj . Si seguimos vemos la línea que dice:

after():execution (public \* hacer\*(..))

Esta línea es clave en forma práctica diremos que después de la ejecución de cualquier método público, sin importar el tipo que retome, que comience con hacer , y sin importar la cantidad y tipo de parámetros que requiera, hacer lo siguiente del código.

Otros aspectos que se verán más adelante errores en la verificación del saldo y en la sincronización de las tareas.