Principio de responsabilidad única

Este artículo, dedicado al llamado "Principio de responsabilidad única", es el primero de una serie de cinco artículos que descubren cinco principios fundamentales del paradigma de la Programación Orientada a Objetos (POO).

Fue en los preparativos del Code Camp de Tarragona '2009 cuando surgió la idea de escribir esta serie de artículos para describir cinco principios fundamentales de la POO cuyas iniciales conforman las siglas **SOLID**. La comprensión de dichos principios nos permitirá mejorar la percepción del no siempre fácil campo de la POO, evitando así malas prácticas que la gran flexibilidad que ofrece esta metodología otorga, fundamentalmente a través de los lenguajes y herramientas que la soportan.

Las herramientas de desarrollo rápido (*Rapid Application Development*, RAD), como Visual Studio 2010, ofrecen al desarrollador un conjunto de funcionalidades que aumentan, ya sea a través de asistentes o mediante "arrastrar y soltar", la productividad en el desarrollo de aplicaciones, y le permiten focalizarse únicamente en la utilización de propiedades o eventos específicos, sin tener que preocuparse en muchas ocasiones del código generado "por debajo". Sin embargo, en ocasiones acabamos pagando un precio muy elevado, ya que estas herramientas dejan tras de sí "cajas negras lógicas" de código difíciles de modificar o reutilizar, en las que los conceptos clave de la orientación a objetos son sacrificados en aras de la productividad; pero hablar de productividad es hablar de facilidad de mantenimiento y calidad del software, con lo que dicho sacrificio, sencillamente, no tiene o debería tener cabida.

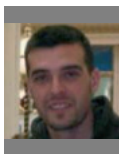
Cohesión

Para tratar de comprender el término "nivel de cohesión", vamos a utilizar como ejemplo el teléfono móvil. Como todos sabéis, un dispositivo móvil está compuesto por una serie de componentes, tales como la pantalla, el teclado, la radio GPRS/3G, el módulo Bluetooth, etc. Todos estos componentes tienen una responsabilidad específica, ya sean los módulos de entrada y salida de datos o los módulos de conectividad y comunicación, etcétera, y por tanto existe una cohesión entre todos los componentes, ya que no hay ningún componente que haga funciones que se solapen con las de otros componentes, ni ninguna función básica que no quede descubierta por un determinado componente. A la hora de buscar un nuevo móvil, miraremos las características y especificaciones técnicas del dispositivo, y además de contemplar las especificaciones que deseamos, también esperaremos que los componentes sean de alta calidad; no es viable que un móvil esté compuesto por los últimos componentes electrónicos del mercado y se venda con una pantalla no táctil en "blanco y negro" de 3 pulgadas, o que, por ejemplo, no tenga algo tan básico como un micrófono.

En un sistema informático también tenemos componentes (además de módulos, clases, etc.), y todos estos componentes lógicos tienen su responsabilidad dentro del sistema. Pensemos pues en el nivel de cohesión de una aplicación no como en una suma de los componentes, sino como el conjunto de los mismos.

Acoplamiento

Siguiendo con el ejemplo del teléfono móvil, un ejemplo de acoplamiento lo encontramos en los cargadores de nuestros móviles. Son cada vez más




José Miguel Torres y Hadi Hariri

MVP de Device Application Development y Technical Evangelist en JetBrains

los fabricantes que optan por adaptadores de corriente estándar en lugar de crear adaptadores propietarios que casi siempre acaban tirados en el contenedor de reciclaje cuando sustituimos el móvil. Se entiende que se opta por la universalidad de los dispositivos de corriente para su reaprovechamiento en otros dispositivos, incluso de diferentes fabricantes. En la ingeniería del software, el acoplamiento entre módulos, clases o cualquier otro tipo de entidad lógica es el grado de dependencia entre ellos. Cuanto más estándar sea la relación de una entidad lógica con otras, mayor reaprovechamiento podremos hacer de ella.

Encapsulación

Seguramente se habrá dado cuenta de que la parte interna de un móvil no es fácilmente accesible; es decir, no tenemos acceso a la electrónica interna. La idea de la encapsulación es la de abstraer determinadas funciones para que, además de ser reutilizables, no requieran que los usuarios tengan los conocimientos del diseñador. La complejidad de un dispositivo móvil o de cualquier otro dispositivo electrónico es muy elevada, y la gran mayoría de usuarios son capaces de sacarle el máximo provecho sin tener nociones específicas sobre su arquitectura interna. La radio Bluetooth o el módem GPRS/3G son los mismos para varios modelos, incluso de diferentes fabricantes. Esta es precisamente la idea de la encapsulación de funciones o características, que lo único que requiere es que el usuario sepa qué se puede hacer con esa función y no cómo está diseñada.

Aplicaciones SOLIDas

Siguiendo el modelo del teléfono móvil y la idea subyacente de estos tres aspectos, podríamos afirmar que el teléfono ideal sería aquel que estuviera compuesto por los mejores componentes del mercado, cuyas interfaces de conexión para la sincronización de datos y recarga de la batería fueran estándares, y cuyas funcionalidades pudiéramos conocer en profundidad sin necesidad de tener que consultar detalles en la documentación técnica para saber cómo han sido diseñadas y así poder sacarles el máximo provecho.

Extrapolando este ejemplo a nuestro mundo, el mundo del software, tenemos que tener siempre presentes estos tres aspectos fundamentales desde el momento mismo en que empezamos el diseño de una nueva aplicación. Es ahí donde entran en escena los cinco principios descritos por el acrónimo mnemotécnico SOLID y presentados a principio de esta década por **Robert C. Martin** (figura 1).

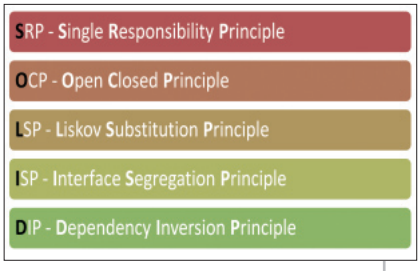


Figura 1. Los cinco principios SOLID

Los principios SOLID pretenden ser una guía a seguir durante la fase de desarrollo para facilitar el mantenimiento de las aplicaciones y tratar de eliminar el impacto de las inevitables modificaciones que éstas sufren durante su ciclo de vida, además de facilitar el uso de las unidades de testeo, entre otras ventajas.

Principio de responsabilidad única

El Principio de responsabilidad única (*Single Responsability Principle* - SRP) fue acuñado por Robert C. Martin en un artículo del mismo título y popularizado a través de su conocido libro [1]. SRP tiene que ver con el nivel de acoplamiento entre módulos dentro de la ingeniería del software. En términos prácticos, este principio establece que:

Una clase debe tener una y solo una única causa por la cual puede ser modificada.

Si una clase tiene dos responsabilidades, entonces asume dos motivos por los cuales puede ser modificada. Por ejemplo, supongamos una clase llamada **Factura**, la cual dentro de un contexto determinado ofrece un método para calcular el importe total, tal y como muestra la figura 2.

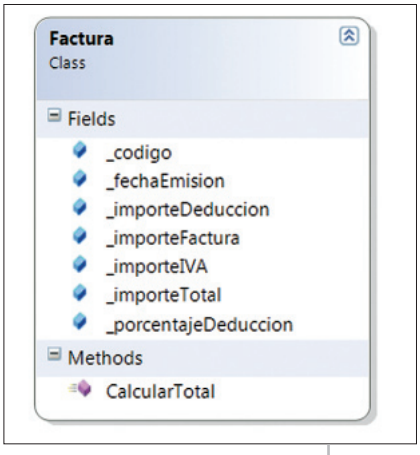


Figura 2

Detectando responsabilidades

La piedra angular de este principio es la identificación de la responsabilidad real de la clase. Según SRP, una responsabilidad es "un motivo de cambio"; algo que en ocasiones es difícil de ver, ya que estamos acostumbrados a pensar un conjunto de operaciones como una sola responsabilidad.

Si implementamos la clase **Factura** tal y como se muestra en el listado 1,

> Los principios SOLID pretenden ser una guía a seguir durante la fase de desarrollo para facilitar el mantenimiento de las aplicaciones

```
public class Factura
{
   public string _codigo;
   public DateTime _fechaEmision;

   public decimal _importeFactura;
   public decimal _importeIVA;
   public decimal _importeDeduccion;
   public decimal _importeTotal;
   public ushort _porcentajeDeduccion;

   // Método que calcula el total de la factura
   public void CalcularTotal()
   {
      // Calculamos la deducción
      _importeDeduccion = (_importeFactura * _porcentajeDeduccion) / 100;
      // Calculamos el IVA
      _importeIVA = _importeFactura * 0.16m;
      // Calculamos el total
      _importeTotal = (_importeFactura - _importeDeduccion) + _importeIVA;
   }
}
```

Listado 1

podríamos decir que la responsabilidad de esta clase es la de calcular el total de la factura y que, efectivamente, la clase cumple con su cometido. Sin embargo, no es cierto que la clase contenga una única responsabilidad. Si nos fijamos detenidamente en la implementación del método **CalcularTotal**, podremos ver que, además de calcular el importe base de la factura, se está aplicando sobre el importe a facturar un descuento o deducción y un 16% de IVA. El problema está en que si en el futuro tuviéramos que modificar la tasa de IVA, o bien tuviéramos que aplicar una deducción en base a una tarifa por cliente, tendríamos que modificar la clase **Factura** por cada una de dichas razones; por lo tanto, con el diseño actual las responsabilidades quedan acopladas entre sí, y la clase violaría el principio SRP.

Separando responsabilidades

El primer paso para solucionar este problema es separar las responsabilidades; para separarlas, primero hay que identificarlas. Enumeremos de nuevo los pasos que realiza el método **CalcularTotal**[1]:

- Aplica una deducción. En base a la base imponible se calcula un descuento porcentual.
- Aplica la tasa de IVA del 16% en base a la base imponible.
- Calcula el total de la factura, teniendo en cuenta el descuento y el impuesto.

En este método se identifican tres responsabilidades. Recuerde que una responsabilidad no es una acción, sino un motivo de cambio, y por lo tanto se deberían extraer las responsabilidades de deducción e impuestos en dos clases específicas para ambas operaciones; estableciendo por un lado la clase **IVA** y por otro la clase **Deduccion**, tal y como se presenta en el listado 2.

```
public class IVA
{
   public readonly decimal _iva = 0.16m;

   public decimal CalcularIVA(decimal importe)
   {
      return importe * _iva;
   }
}

public class Deduccion
{
   private decimal _deduccion;

   public Deduccion(ushort porcentaje)
   {
      _deduccion = porcentaje;
   }

   public decimal CalcularDeduccion(decimal importe)
   {
      return (importe * _deduccion) / 100;
   }
}
```

Listado 2

[1] Es muy probable que si algún contable o inspector financiero viera estas operaciones, como mínimo nos llevaríamos un tirón de orejas; pero démoslas por buenas para los fines de este artículo.

```
public class Factura
{
  public string _codigo;
  public DateTime _fechaEmision;

  public decimal _importeFactura;
  public decimal _importeIVA;
  public decimal _importeDeduccion;
  public decimal _importeTotal;
  public ushort _porcentajeDeduccion;

  // Método que calcula el total de la factura
  public void CalcularTotal()
  {
     // Calculamos la deducción
     Deduccion deduccion =
           new Deduccion(_porcentajeDeduccion);
     _importeDeduccion =
           deduccion.CalcularDeduccion(
                 _importeFactura);
      // Calculamos el IVA
      IVA iva = new IVA();
      _importeIVA = iva.CalcularIVA(
                 _importeFactura);
      // Calculamos el total
      _importeTotal = (_importeFactura -
                 _importeDeduccion) +
                 _importeIVA;
    }
}
```

Listado 3

Ambas clases contienen datos y un método y se responsabilizan únicamente en calcular el IVA y la deducción, respectivamente, de un importe. Además, con esta separación logramos una mayor cohesión y un menor acoplamiento, al aumentar la granularidad de la solución. La correcta aplicación del SRP simplifica el código y se traduce en facilidad de mantenimiento, mayores posibilidades de reutilización de código y de crear unidades de testeo específicas orientadas a cada clase/responsabilidad. El listado 3 muestra la nueva versión de la clase **Factura**, que hace uso de las dos nuevas clases **IVA** y **Deduccion**.

> La correcta aplicación del SRP simplifica el código y se traduce en facilidad de mantenimiento, mayores posibilidades de reutilización de código y de crear unidades de testeo específicas para cada responsabilidad

Ampliando el abanico de "responsabilidades"

Comentábamos anteriormente que no es fácil detectar las responsabilidades, ya que generalmente tendemos a agruparlas. No obstante, existen escenarios o casuísticas en los que "se permite" una cierta flexibilidad. Robert C. Martin expone un ejemplo utilizando la interfaz **Modem**:

```
interface Modem
{
  void dial(int pNumber);
  void hangup();
  void send(char[] data);
  char[] receive();
}
```

En este ejemplo se detectan dos responsabilidades, relacionadas con la gestión de la comunicación (**dial** y **hangup**) y la comunicación de datos (**send** y **receive**). Efectivamente, cada una de las funciones puede cambiar por diferentes motivos; sin embargo, ambas funciones se llamarán desde distintos puntos de la aplicación y no existe una dependencia entre ellas, con lo que no perderíamos la cohesión del sistema.

Conclusión

Pensemos siempre en el ciclo de vida de una aplicación, y no únicamente en su diseño y desarrollo. Toda aplicación sufre modificaciones a causa de cambios en los requisitos o arreglo de fallos existentes, y el equipo de desarrollo puede variar; si a ello le sumamos que el código es poco *mantenible*, los costes de mantenimiento se dispararán, y cualquier modificación se presentará como una causa potencial de errores en entidades relacionadas dentro del sistema.

Referencias

[1] **Martin, Robert C.** "Agile Software Development: Principles, Patterns, and Practices". Prentice-Hall, 2002.

[2] **Centro de Arquitectura de MSDN.** http://msdn.microsoft.com/es-ES/architecture.