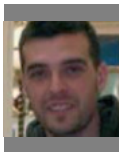
Principio Open/Closed

Este es el segundo de una serie de cinco artículos que describen los principios SOLID y su aplicación en la Programación Orientada a Objetos. Después de examinar en nuestra entrega anterior el Principio de Responsabilidad Única, este mes nos adentramos en otro de los principios, que además guarda una estrecha relación con el primero: el Principio Open/Closed.



José Miguel Torres
MVP de Device Application Development

Todas las aplicaciones cambian durante su ciclo de vida, y siempre vendrán nuevas versiones tras la primera *release*. No por ello debemos adelantarnos a desarrollar características que el cliente podría necesitar en el futuro; si nos pusiéramos en el papel de adivinos, seguramente fallaríamos y probablemente desarrollaríamos características que el cliente nunca necesitará. El principio YAGNI ("*You Ain't Gonna Need It*" o "No vas a necesitarlo"), utilizado en la Programación Extrema, previene de implementar nada más que lo que realmente se requiera. La idea es desarrollar ahora sobre los requisitos funcionales actuales, no sobre los que supongamos que aparecerán dentro de un mes.

La actitud de adelantarnos a los acontecimientos es un mecanismo de defensa que en ocasiones acusamos los desarrolladores para prevenir lo que tarde o temprano será inevitable: la modificación. Lo único que podemos hacer es minimizar el impacto de una futura modificación en nuestro sistema, y para ello es imprescindible empezar con un buen diseño, ya que la modificación de una clase o módulo de una aplicación mal diseñada generará cambios en cascada sobre las clases dependientes que derivarán en unos efectos indeseables. La aplicación se convierte, así, en rígida, impredecible y no reutilizable.

Ahora bien, ¿cómo debemos plantear nuestras aplicaciones para que se mantengan estables ante cualquier modificación?

El Principio Open/Closed

El **Principio Open/Closed** (*Open/Closed Principle*, OCP) fue acuñado por el **Dr. Bertrand Meyer** en su libro "*Object Oriented Software Construction*" [1] y afirma que:

Una clase debe estar abierta a extensiones, pero cerrada a las modificaciones.

OCP es la respuesta a la pregunta que hacíamos anteriormente, ya que argumenta que deberíamos diseñar clases que **nunca cambien**, y que cuando un requisito cambie, lo que debemos hacer es **extender el comportamiento** de dichas clases añadiendo código, no modificando el existente.

Las clases que cumplen con OCP tienen dos características:

- Son **abiertas** para la extensión; es decir, que la lógica o el comportamiento de esas clases puede ser extendida en nuevas clases.
- Son **cerradas** para la modificación, y por tanto el código fuente de dichas clases debería permanecer inalterado.

Podría parecer que ambas características son incompatibles, pero eso no es así. Veamos un ejemplo de una clase que rompe con OCP. Supongamos un sistema de gestión de proyectos al estilo de Microsoft Project. Obviemos de momento la complejidad real que existe en dicho sistema, y centrémonos únicamente en la entidad `Tarea`, tal y como muestra la figura 1. Dicha clase viene determinada por uno de los estados `Pendiente`, `Finalizada` o `Cancelada`, representados mediante la enumeración `EstadosTarea`. Además, la clase implementa dos métodos, `Cancelar` y `Finalizar` que cambian, si es posible, el estado de la tarea. En el listado 1

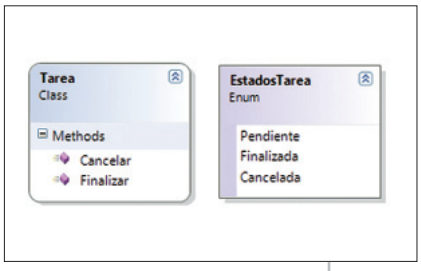


Figura 1

podemos ver la implementación inicial del método **Finalizar**.

Un cambio típico solicitado por el cliente de la aplicación sería la adición de un nuevo estado para controlar las tareas que se han pospuesto, con lo que la adaptación a esta modificación podría ser la expuesta en el listado 2. Aparentemente, parece una modificación trivial; sin embargo, este cambio puede replicarse en otros métodos o clases que utilicen la enumeración **EstadosTarea**, de forma que en nuestro caso también deberíamos modificar el método **Cancelar** (listado 3).

En definitiva, por cada nuevo estado que implementemos tendremos que identificar todas las clases que lo utilizan (tanto la clase **Tarea** como las clases lógicamente involucradas) y modificarlas, violando no únicamente OCP sino también el Principio DRY ("*Don't Repeat Yourself*", "No te repitas"), otro principio que pretende reducir al máximo cualquier tipo de duplicación. En este tipo de modificaciones existe una alta probabilidad de olvidar modificar algún método relacionado con el nuevo estado implementado en el enumerador **EstadosTarea**, lo que elevaría la probabilidad de aparición de un nuevo *bug*.

Fundamentos de la orientación a objetos

La cuestión se centra en cómo minimizar el impacto de una modificación en nuestro sistema, sin comprometer OCP; esto es, manteniendo la "simbiosis" entre las dos características del principio: abierto en extensión y cerrado en modificación.

Volvamos a la entidad **Tarea** del ejemplo anterior. Por lo que hemos podido ver, los métodos dependen en gran medida del estado de la tarea. Así,

```
public void Finalizar()
{
   switch (_estadoTarea)
   {
       case EstadosTarea.Pendiente:
           // finalizamos
           break;
       case EstadosTarea.Finalizada:
           throw new ApplicationException("Tarea ya finalizada");
       case EstadosTarea.Cancelada:
           throw new ApplicationException("Imposible finalizar. Tarea cancelada");
       default:
           throw new ArgumentOutOfRangeException();
   }
}
```

Listado 1

```
public void Finalizar()
{
    switch (_estadoTarea)
    {
        case EstadosTarea.Pendiente:
            // finalizamos
            break;
        case EstadosTarea.Finalizada:
            throw new ApplicationException("Tarea ya finalizada");
        case EstadosTarea.Cancelada:
            throw new ApplicationException("Imposible finalizar. Tarea cancelada");
        case EstadosTarea.Pospuesta:
            throw new ApplicationException("Imposible finalizar. Tarea no completada");
        default:
            throw new ArgumentOutOfRangeException();
    }
}
```

Listado 2

```
public void Cancelar()
{
    switch (_estadoTarea)
    {
        case EstadosTarea.Pendiente:
            // cancelamos
            _estadoTarea = EstadosTarea.Cancelada;
            break;
        case EstadosTarea.Finalizada:
            throw new ApplicationException("Imposible cancelar. Tarea finalizada");
        case EstadosTarea.Cancelada:
            throw new ApplicationException("Tarea ya cancelada");
        case EstadosTarea.Pospuesta:
            // cancelamos
            _estadoTarea = EstadosTarea.Cancelada;
            break;
        default:
            throw new ArgumentOutOfRangeException();
    }
}
```

Listado 3

```
class EstadosTareaHelper
{
    public virtual void Finalizar(EstadosTarea estado)
    {
        switch ( estado) {
            case EstadosTarea.Pendiente:
                // finalizamos
            case EstadosTarea.Pospuesta:
                throw new ApplicationException("Imposible finalizar. Tarea no completada");
            default:
                throw new ArgumentOutOfRangeException();
        }
    }

    public virtual void Cancelar(EstadosTarea estado)
    {
        switch (estado) {
                // ...
                // cancelamos
        }
    }

    public virtual void Posponer(EstadosTarea estado)
    {
        switch (estado) {
                // ...
                // posponemos
        }
    }
}
```

Listado 4

> Cuando un requisito cambie, lo que debemos hacer es extender el comportamiento añadiendo código, y no modificando el existente

una tarea podrá finalizarse o cancelarse dependiendo de su estado previo, pues no podremos cancelar una tarea que haya sido finalizada. De la misma forma, introduciendo el nuevo estado **EstadosTarea.Pospuesta** implementaríamos un nuevo método llamado **Posponer**, cuya lógica sería obvia: únicamente podría posponerse una tarea que estuviera en estado pendiente. En definitiva, todo gira alrededor del estado de la tarea, y por tanto el comportamiento de la misma dependerá del estado en que se encuentre. Una opción sería encapsular dicho estado en una clase auxiliar e implementar en ella los métodos **Finalizar**, **Cancelar** y **Posponer**, mediante los cuales definimos el comportamiento, tal y como se muestra en el listado 4, para luego delegar los métodos del objeto **Tarea** hacia dicha clase.

Pese a que hayamos extraído y aislado el estado de la entidad **Tarea**, aún no hemos resuelto el problema. De hecho, ahora hemos aislado la responsabilidad en la clase **EstadosTareaHelper**; sin embargo, estamos algo más cerca de la solución. Estudiemos de nuevo los estados -métodos- de la clase **EstadosTareaHelper**. La lógica de cada acción está escrita en todos los métodos y por tanto se repite; es decir, todos los métodos contemplan la opción de **Finalizar** una tarea, y en base a ello actúan de una forma u otra. La operación **Posponer** no podrá ejecutarse si el estado de la tarea es **Cancelada**, y la operación **Cancelar** únicamente podrá ejecutarse si el estado es **Pendiente**. A través de este razonamiento, podemos detectar un patrón: un mismo contrato –los métodos– y diferentes comportamientos en base a un estado. Esto en OO puede ser solucionado mediante polimorfismo, como se muestra en el listado 5.

Básicamente, lo que hemos hecho es crear una clase por cada estado en lugar de tener una única clase cuyos métodos están basados en sentencias condicionadas por el estado de la tarea (**switch** o **if**). Además, con esta nueva implementación hemos delegado la responsabilidad de finalizar, cancelar o posponer a una nueva clase

```
abstract class EstadoTareaBase
{
    protected Tarea _tarea;

    public abstract void Finalizar();
    public abstract void Cancelar();
    public abstract void Posponer();
}

class EstadoTareaPendiente : EstadoTareaBase
{

    public override void Finalizar()
    {
        // finalizamos
    }

    public override void Cancelar()
    {
        // cancelamos
    }

    public override void Posponer()
    {
        // posponemos
    }
}

class EstadoTareaFinalizada : EstadoTareaBase
{
    public override void Finalizar()
    {
        throw new ApplicationException("Tarea ya finalizada");
    }

    public override void Cancelar()
    {
        throw new ApplicationException("Imposible cancelar. Tarea finalizada");
    }

    public override void Posponer()
    {
        throw new ApplicationException("Imposible posponer. Tarea finalizada");
    }
}

class EstadoTareaCancelada : EstadoTareaBase
{

    public override void Finalizar()
    {
        throw new ApplicationException("Imposible finalizar. Tarea cancelada");
    }

    public override void Cancelar()
    {
        throw new ApplicationException("Tarea ya cancelada");
    }

    public override void Posponer()
    {
        throw new ApplicationException("Imposible posponer. Tarea cancelada");
    }
}

class EstadoTareaPospuesta : EstadoTareaBase
{
    public override void Finalizar()
```

```
    {
        throw new ApplicationException("Imposible posponer. Tarea finalizada");
    }

    public override void Cancelar()
    {
        // cancelamos
    }

    public override void Posponer()
    {
        throw new ApplicationException("Tarea ya pospuesta");
    }
}

class Tarea
{
    private EstadoTareaBase _estadoTarea;

    public Tarea()
    {
        _estadoTarea = new EstadoTareaPendiente();
    }

    public void Finalizar()
    {
        _estadoTarea.Finalizar();
    }

    public void Cancelar()
    {
        _estadoTarea.Cancelar();
    }

    public void Posponer()
    {
        _estadoTarea.Posponer();
    }
}
```

Listado 5

EstadoTareaBase que hemos marcado como abstracta. La clase **Tarea** implementará sus propios métodos y delegará la responsabilidad a través de las clases-estados que heredan de **EstadoTareaBase**. Debido a que la clase **Tarea** gira en torno a un estado, asumimos que el estado inicial por defecto es **Pendiente**, y así lo especificamos en el constructor, instanciando **EstadoTareaPendiente**.

En realidad, hemos aplicado un patrón ya conocido, el patrón de diseño **State**, ya que el comportamiento de la clase cambia dependiendo del estado, en este caso, de la tarea, y por lo tanto hemos abstraído cada uno de los estados como entidades independientes. Ante un nuevo requisito en el que intervenga un nuevo estado, lo único que deberemos hacer es crear una nueva clase que herede de **EstadoTareaBase** e implementar los métodos virtuales, extendiendo así el comportamiento de la aplicación sin comprometer el código existente.

Conclusión

Cuando hablábamos el mes pasado del Principio de Responsabilidad Única, argumentamos la importancia de que cada clase tuviera una y solo una responsabilidad dentro del sistema, de forma que cuanto menos impacto tenga una clase en el conjunto global del sistema, menos repercusión global tendrá una modificación de la clase en dicho sistema. Este mismo argumento es la línea que pretende seguir el Principio Open/Closed, que pese a ser relativamente sencillo de comprender conceptualmente, no sucede lo mismo cuando se aplica. Las claves para la correcta aplicación de este principio son la abstracción y el polimorfismo, como hemos podido ver en el ejemplo.

Referencias

[1] **Meyer, Bertrand**. "*Object Oriented Software Construction*", Prentice-Hall, 1988. Edición encastellano: "Construcción de Software Orientado a Objetos", Pearson Educación, 1998 (traducido por **Luis Joyanes**, **Miguel Katrib**, **Rafael García Bermejo** y **Salvador Sánchez**).

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