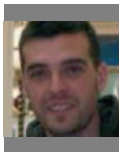
Principio de Segregación de Interfaces

Como parte de la serie dedicada a presentar los cinco principios SOLID de programación, este mes nos centramos en el Principio de Segregación de Interfaces (*Interface Segregation Principle, ISP*). Este principio trata sobre las desventajas de las interfaces "pesadas", y guarda una estrecha relación con el nivel de cohesión de las aplicaciones. En este artículo veremos qué perjuicios ocasionan las interfaces "pesadas", qué impacto tienen en la cohesión del sistema y cómo en muchas ocasiones vulneran el Principio de Responsabilidad Única, y cómo ISP ofrece una solución a estos problemas.



José Miguel Torres

MVP de Device Application Development

El Principio de Segregación de Interfaces

El Principio de Segregación de Interfaces fue utilizado por primera vez por **Robert C. Martin** durante unas sesiones de consultoría en Xerox. Por aquella época, Xerox estaba diseñando una impresora multifuncional. El software diseñado para la impresora funcionaba y se adaptaba perfectamente a las necesidades iniciales de la impresora; sin embargo, conforme fue evolucionando, y por lo tanto cambiando, se hizo cada vez más difícil de mantener. Cualquier modificación tenía un gran impacto global sobre el sistema. La utilización de ISP permitió reducir los riesgos de las modificaciones y otorgó una mayor facilidad al mantenimiento.
El ISP declara que:

> *Los clientes no deben ser forzosamente dependientes de las interfaces que no utilizan.*

A continuación veremos la utilidad de ISP y su significado, una vez que identifiquemos qué es una interfaz "pesada" y qué problemas lleva asociados.

Interfaces "pesadas"

Observemos el diagrama de clases de la figura 1. Básicamente, consta de dos modelos de impresoras representadas por las clases `Modelo1998` y `Modelo2000`, ambas herederas de la clase abstracta `ImpresoraMultifuncional`.

Inicialmente, la clase abstracta `ImpresoraMultifuncional` declaraba los métodos correspondientes a las funciones típicas que realiza una impresora multifuncional, como son la propia impresión, el escaneado, el envío de fax y la cancelación de cualquier operación. La impresora `Modelo1998` fue el primer modelo en basarse en esta interfaz; poco después se añadió un nuevo modelo, `Modelo2000`, que además de las funciones anteriores añadía la posibilidad de hacer fotocopias.

Posteriormente, surgió un nuevo modelo (`Modelo2002`) que se basaba en la misma clase abstracta `ImpresoraMultifuncional` e incorporaba el soporte para comunicaciones TCP/IP en lugar del servicio de fax; este modelo permitía enviar un documento directamente por correo electrónico, evitando así los altos costes de telefonía. El problema se presenta al implementar en `Modelo2002` el método heredado `EnviarFax`, ya que dicho modelo prescinde de dicha funcionalidad. Una posible implementación sería la que se presenta en el listado 1.

El método `EnviarFax` no se implementa, y por consiguiente una llamada al método generaría una excepción del tipo `NotImplementedException`. Sí, es cierto que podríamos quitar dicha excepción y podríamos dejar el método vacío; pero entonces el programador que utilice la clase se encontrará con un método que sencilla-

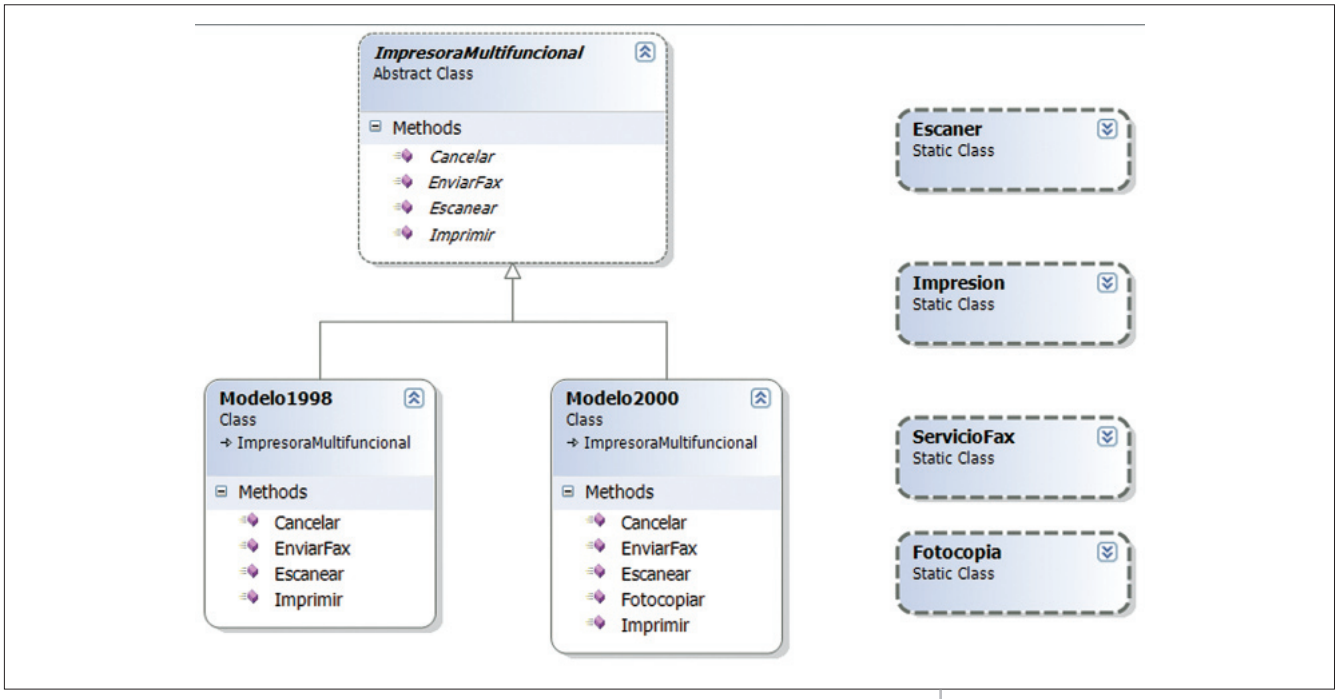


Figura 1

mente no hace nada. Esto podríamos intentar solucionarlo de varias formas: mediante documentación, indicando que el método no es funcional; mediante comentarios en el código (pero es posible que un programador que utilice la clase no tenga acceso al código), etc. En cualquier caso, el problema seguiría existiendo, y solo estaríamos ocultándolo.

```
class Modelo2002 : ImpresoraMultifuncional
{
    public override void Imprimir()
    {
        Impresion.EnviarImpresion();
    }

    public override void Escanear()
    {
        Escaner.DigitalizarAFormatoPng();
    }

    public override void Cancelar()
    {
        Impresion.CancelarImpresion();
    }

    public override void EnviarFax()
    {
        throw new System.NotImplementedException();
    }

    public void EnviarEMail()
    {
        // Enviamos por correo electrónico
    }
}
```

Listado 1

De "pesada" a confusa

Es importante que nos concienciemos de este problema. En nuestro ejemplo, se trata de un único método, y eludir el problema puede ser bastante obvio; pero si la clase abstracta implementara una docena de métodos y únicamente utilizáramos tres o cuatro de ellos en un contexto no tan claro como el de las impresoras multifuncionales, el problema se haría más complejo.

Un ejemplo de esto lo tenemos en el propio .NET Framework. La clase abstracta **`System.Web.Security.MembershipProvider`** contiene todos los métodos necesarios para la autenticación ASP.NET: valida credenciales accediendo a algún mecanismo de almacenamiento (SQL Server, sistema de archivos, etc.), bloquea usuarios, gestiona contraseñas, etc. Si implementamos nuestro propio proveedor de autenticación e implementamos la clase **`MembershipProvider`**, seguramente solo utilizaremos algunos de los métodos heredados (es decir, implementaremos únicamente ciertas funcionalidades disponibles en la clase base), y en este caso no es tan evidente cuáles

de esos métodos deberemos redefinir. ¿Cómo sabría el programador que utilice nuestra clase qué métodos ésta implementa? ¿Qué comportamiento debería tener nuestra clase ante la llamada a un método no implementado? En definitiva, ¿qué valor real tienen los métodos que están disponibles pero no implementados? La respuesta es: ninguno, aparte de crear confusión.

En nuestro ejemplo, la clase **Modelo2000** implementa, al contrario que **Modelo1998**, la característica de **Fotocopiar**. Dicho método se implementa en la propia clase **Modelo2000**, y quizás nos hayamos preguntado por qué no hemos añadido el método a la clase abstracta **ImpresoraMultifuncional**. El motivo puede ser bien dispar. Quizás quién diseñó el sistema decidió no tocar la clase abstracta y extender la clase **Modelo2000**; sin embargo, resulta que a partir de **Modelo2000** todas las impresoras tienen soporte de fotocopia y por lo tanto todos los modelos deberán implementar el método **Fotocopiar**. Utilizando esta estrategia, tenemos que vulnerar el principio DRY (*Don't Repeat Yourself*); y lo que es más preocupante, otro programador puede tener la ocurrencia o la necesidad de modificar el contrato o el nombre del método. En definitiva, ello complica el mantenimiento del sistema; cualquier modificación sobre del método **Fotocopiar** implicará buscarlo por todo el código, aumentando por tanto el riesgo de error. Es contradictorio que tengamos encapsulados en una clase abstracta miembros que no usamos, por ejemplo, en **Modelo2002**, mientras que otros que sí serían firmes candidatos a serlos, como el caso del método **Fotocopiar**, no lo son.

> Para solucionar problemas similares al del ejemplo que presentamos aquí, debemos segregar las operaciones en pequeñas interfaces.

Segregación de interfaces

Para solucionar el problema, debemos segregar las operaciones en pequeñas interfaces. Una interfaz es un contrato que debe cumplir una clase, y tales contratos deben ser específicos, no genéricos; esto nos proporcionará una forma más ágil de definir una única responsabilidad por interfaz - de otra forma, violaríamos además el Principio de Responsabilidad Única (*Single Responsibility Principle*, SRP).

```
public interface IImprimible
{
    void Imprimir();
}

public interface IFotocopiable
{
    void Fotocopiar();
}

public interface IEscaneable
{
    void Escanear();
}

public interface IFaxCompatible
{
    void EnviarFax();
    void RecibirFax();
}

public interface ITcpIpCompatible
{
    void EnviarEMail();
}

class Modelo1998 : IImprimible, IEscaneable, IFaxCompatible
{
    // ...
}

class Modelo2000 : IImprimible, IEscaneable, IFaxCompatible,
IFotocopiable
{
    // ...
}

class Modelo2002 : IImprimible, IEscaneable, IFotocopiable,
ITcpIpCompatible
{
    // ...
}
```

Listado 2

Retomando de nuevo el ejemplo práctico, volvamos a replantear el sistema y separemos las responsabilidades por interfaces. En el listado 2 podemos ver el resultado de dicha segregación.

Conclusión

Mediante la segregación de interfaces, el planteamiento del diseño otorga una mayor cohesión al sistema, lo que se traduce, por una parte, en un menor coste de mantenimiento, y por otra, en un menor riesgo de errores y una mejor localización de los mismos.