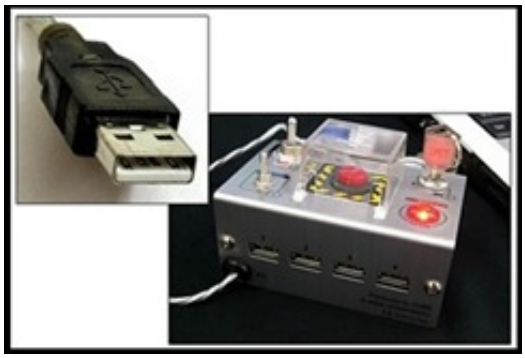
21st April 2011

Interface Segregation Principle o principio de segregación de interfaces



[http://lh6.ggpht.com/_bpKrBFNvmI0/TbB5jz6ac6I/AAAAAAAAAFo/E3c5ViW_5Y4/s1600-h/interface_segregation_principle12.jpg] Este principio de diseño dice “Los clientes no deben estar obligados a implementar una interface que luego no usarán”.

Este principio trata de que no crees interface “pesadas”.

La primer forma de hacer una interfaz pesada es crear una interfaz con mucha funcionalidad que solo usa una implementación y que las otras la heredan pero no hacen nada o como mucho elevan una excepción del estilo “Esto no esta permitido”.

El ejemplo típico en esta situación es la implementación de un sistema de membresía propio, aquel donde se leen los usuarios de una base de datos. Hay que heredar de la clase MembershipProvider [http://msdn.microsoft.com/es-es/library/system.web.security.membershipprovider.aspx] , del cual se pueden sobrescribir unos 20 métodos, cuando con solo sobrescribir validateUser ya puedes hacerlo. (sobretodo si no tienes el control para actualizar o crear nuevos usuarios).

Otro ejemplo practico, vemos que muchas veces cuando se habla del patrón repository se crea una clase base que hace un CRUD. Lo cierto es que a la hora de implementar un repository te encontras con que muchos de los datos son de solo lectura, y te obliga a implementar una interfaz o clase abstracta que hace todas las operaciones. En estos casos se pueden hacer dos Interfaces una para solo lectura y otra para escritura. Si es un repositorio de solo lectura usara solo la primer interfaz, sino usará las dos interfaces.

Otra forma de hacer una interfaz “pesada” es que tenga mas información de la que necesita.

Por ejemplo, si tengo que hacer un programa que envíe un mail a un contacto con información de un pedido, puedo pensar que la clase EnviarMail en el método enviar acepte el contacto como parámetro (12).

```
1:      public class Contacto
2:      {
3:          public string Nombre { get; set; }
4:
5:          public string CuentaBancaria { get; set; }
6:
```

```
 7:          public string Email { get; set; }
 8:      }
 9:
10:      public interface IMailSender
11:      {
12:          void Enviar(Contacto contacto, string cuerpoMensaje);
13:      }
```

Ahora bien, ¿Hace falta pasar todo el contacto cuando solo usará la dirección de mail y el nombre?

Las interfaces de la clase se puede dividir en pequeños grupos funcionales. Cada
grupo sirve a un cliente diferente. De esta manera, unos clientes conoce una interfaz con un grupo de funcionalidad y otro cliente conoce otro grupo.

Acá pongo una interfaz llamada IReceptorMail (1-5) que solo tenga el nombre y la dirección de correo electrónico. Luego coloco la interfaz sobre la clase contacto (7) y cambiamos el método de enviar (18) para que use la interfaz IReceptorMail en lugar de la clase contacto.

```
 1:      public interface IReceptorMail
 2:      {
 3:          string Nombre { get; set; }
 4:          string Email { get; set; }
 5:      }
 6:
 7:      public class Contacto : IReceptorMail
 8:      {
 9:          public string Nombre { get; set; }
10:
```

```
11:          public string CuentaBancaria { get; set; }
12:
13:          public string Email { get; set; }
14:      }
15:
16:      public interface IMailSender
17:      {
18:          void Enviar(IReceptorMail contacto, string cuerpoMensaje);
19:      }
```

Un buen barómetro para saber si estas dando en el clavo, en base a mi experiencia, es que cuando estas usando toda la interfaz es una interfaz que cumple con el principio ISP.

Conclusión

Hoy trate la I de los principios SOLID [http://en.wikipedia.org/wiki/Solid_%28object-oriented_design%29] .

Para que un sistema no tenga alta cohesión no solo basta con que se usen interfaces, sino que no exista dependencia entre ella. Por eso, este principio hace que nos centremos en que ellas cumplan con una funcionalidad, si lo hacen será mas simple cumplir con el principio de solo una responsabilidad [http://danielmazzini.blogspot.com/2010/10/single-responsibility-principle-srp-o.html] .

Publicado 21st April 2011 por Daniel Mazzini

Etiquetas: Patterns, SOLID