SOLID y GRASP

Buenas prácticas hacia el éxito en el desarrollo de software.

Juan García Carmona

d.jgc.it@gmail.com

15 de noviembre de 2012

Éste documento contiene la práctica totalidad de lo expuesto por el autor durante el simposio celebrado el día 09 de Noviembre en la Escuela Técnica Superior de Ingeniería Informática de la Universidad de Sevilla organizado por el grupo de investigación de Ingeniería Web y Testing Temprano (IWT2). En él se analizan estos dos acrónimos, "SOLID" y "GRASP", y se ven y ponen en práctica algunos de los principios básicos del diseño y la programación orientada a objetos con el objetivo de tomar conciencia de diversas buenas prácticas que harán que los sistemas que diseñemos y los productos de software que desarrollemos sean de la mas alta calidad.

Contenido

CREATIVE COMMONS Attribution-NonCommercial-NoDerivs 3.0 Unported

SOLID y GRASP

Buenas prácticas hacia el éxito en el desarrollo de software.

PRINCIPIOS SOLID

Los principios SOLID son cinco principios enunciados por Robert C. Martin alrededor del año 2000 enfocados a la elaboración de software de calidad. Una clara definición de los principios SOLID la podemos encontrar en Wikipedia:

> En ingeniería de software, **SOLID** (**S**ingle responsibility, **O**pen-closed, **L**iskov substitution, **I**nterface segregation and **D**ependency inversion) es un acronimo mnemónico introducido por **Robert C. Martin** a comienzos de la decada del 2000 que representa cinco principios básicos de la **programación orientada a objetos** y el diseño. Cuando estos principios se aplican en conjunto es más probable que un desarrollador cree un sistema que sea **fácil de mantener y ampliar en el tiempo**. Los principio SOLID son guías que pueden ser aplicadas en el desarrollo de software para **eliminar código sucio** provocando que el programador tenga que refactorizar el código fuente hasta que sea legible y extensible. Debe ser utilizado con el desarrollo guiado por pruebas o **TDD**, y forma parte de la **estrategia global** del desarrollo ágil de software y programación adaptativa.

Ésta definición es perfectamente válida y se podría resumir diciendo que son cinco principios que hay que tener siempre presentes si queremos desarrollar un software de calidad, legible, entendible y fácilmente testeable.

¿Por qué fácilmente testeable?

Principalmente porque nos va a obligar a sacar muchas interfaces, cosa que favorecerá la utilización de Mocking, y hará que el trabajo de testear una clase sea conciso, además favorece el TDD porque se prepara el código para los sucesivos cambios propios del TDD…

Como bien dice la definición de Wikipedia los cinco principios son:

- **SRP: Single Responsibility Principle**
- **OCP: Open/Closed Principle**
- **LSP: Liskov Substitution Principle**
- **ISP: Interfaze Segregation Principle**
- **DIP: Dependency Inversion Principle**

En ésta sesión de trabajo veremos, uno por uno, cada principio, siguiendo éste guion con cada uno de ellos

1. Definición original, en inglés.
2. Traducción literal.
3. Interpretación.

4. Un ejemplo que no cumple el principio y una explicación de por qué no lo cumple que incluye diagramas estáticos UML y código C#.
5. Trabajo en grupo para arreglar dicho ejemplo aplicando el principio en cuestión.

Así que manos a la obra…

S → SRP → Single Responsibility Principle

Principio de única responsabilidad

Enunciado original: "*There should never be more than one reason for a class to change*

Traducción literal: "No debería haber nunca más de una razón para cambiar una clase."

Interpretación: Una clase debería concentrarse sólo en hacer una cosa de tal forma que cuando cambie algún requisito en mayor o menor medida dicho cambio sólo afecte a dicha clase por una razón.

Ejemplo 1

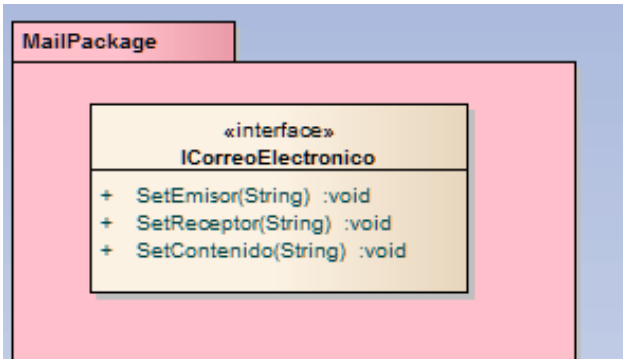
Enunciado:

Supongamos que tenemos un paquete de software para la gestión de correos electrónicos. En algún lugar de dicho paquete, antes del envío de los correos en sí, queremos establecer los distintos parámetros que, por ahora, son:

- **Emisor**
- **Receptor**
- **Contenido**

UML ORIGINAL:

De una forma muy sencilla la interpretación de dicho enunciado podría ser ésta:



CÓDIGO ORIGINAL:

Y la implementación es ésta:

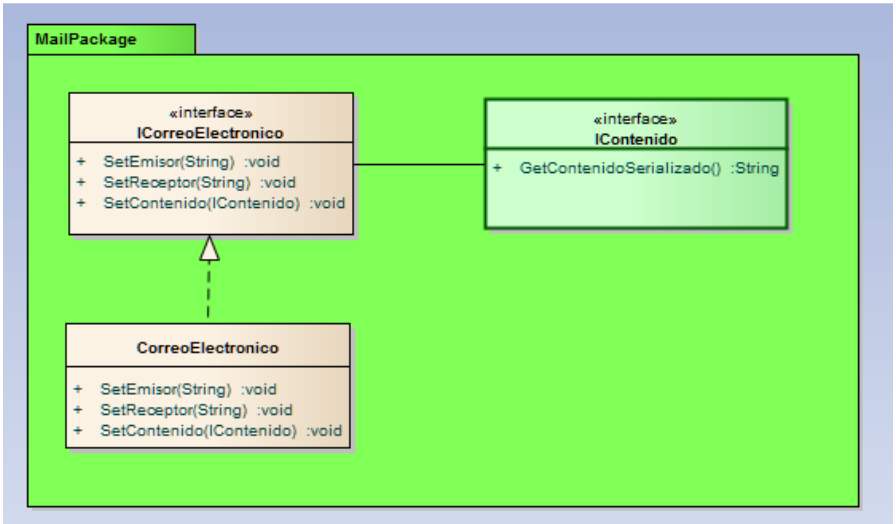
```
public interface ICorreoElectronico
{
    void SetEmisor(String emisor);
    void SetReceptor(String receptor);
    void SetContenido(String contenido);
}

public class CorreoElectronico : ICorreoElectronico
{
    public void SetEmisor(String emisor)
    {
    }
    public void SetReceptor(String receptor)
    {
    }
    public void SetContenido(String contenido)
    {
    }
}
```

¿Por qué incumple el principio de única responsabilidad?

SRP nos dice que sólo debería haber un motivo para cambiar una clase o una interfaz y, por ende, las clases que la implementan. ¿Qué motivos veis que puedan hacer que la clase cambie? En éste caso la traducción literal es clara, si cambiara el contenido porque, por ejemplo, admitiera más tipos, habría que modificar el código cada vez que añadamos algún tipo de contenido o si, por ejemplo, cambiara el protocolo y quisiéramos añadir más campos también habría que modificar la clase email. Por tanto hay más de un motivo por el que tendríamos que modificar ésta clase. Veamos otro ejemplo que no cumple con el SRP.

Solución:

UML



CÓDIGO

```
interface ICorreoElectronico
{
   void SetEmisor(String emisor);
   void SetReceptor(String receptor);
   void SetContenido(IContenido content);
}

interface IContenido
{
   String GetContenidoSerializado();    // para serializar el contenido
}

class CorreoElectronico : ICorreoElectronico
{
   public void SetEmisor(String sender) { }
   public void SetReceptor(String receiver) { }
   public void SetContenido(IContenido content) { }
}
```

Si en vez de un contenido String utilizamos una interfaz IContenido que suponga un contrato para hacer viable cualquier tipo de contenido cada vez que nos pidan que añadamos un tipo de contenido sólo tendremos que modificar dicha interfaz y/o las clases que lo implementen, a lo que afecte el cambio, y no ya a la clase de CorreoElectrónico y a su interfaz. ¿Qué os parece?

O → OCP → Open / Closed Principle

Principio abierto/cerrado

Enunciado original: "Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification."

Traducción literal: "Las entidades de software (clases, módulos, funciones, etcétera) deberían estar abiertas a la extensión pero cerradas a la modificación."

Interpretación: Cambia el comportamiento de una clase mediante herencia, polimorfismo y composición.

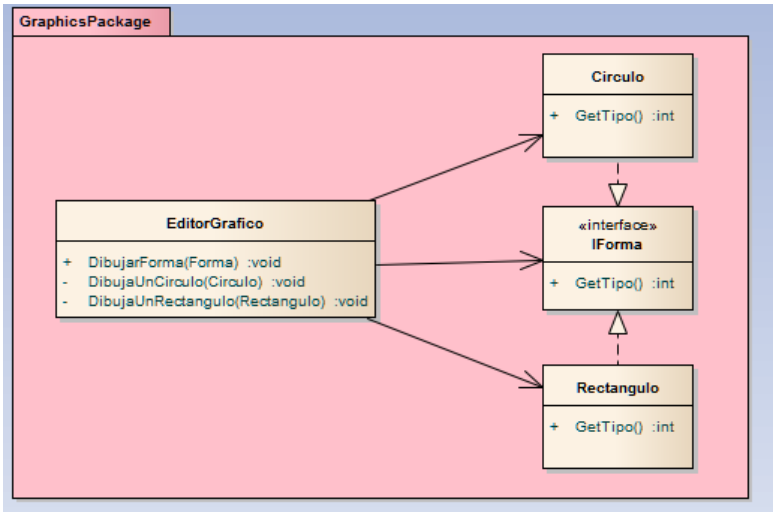
Esto es, anticípate al cambio, prepara tu código para que los posibles cambios en el comportamiento de una clase se puedan implementar mediante herencia y composición.

Ejemplo:

Enunciado:

Diseñar un editor gráfico capaz de dibujar un círculo y un rectángulo

UML ORIGINAL:



CÓDIGO ORIGINAL:

```
public interface IForma
{
   int GetTipo();
}

public class Rectangulo : IForma
{
   public int GetTipo()
   {
      return 1;
   }
}

public class Circulo : IForma
{
   public int GetTipo()
   {
      return 2;
   }
}

public class EditorGrafico

  public void DibujarForma(IForma forma)
  {
     switch (forma.GetTipo())
     {
        case 1:
           DibujaUnRectangulo((Rectangulo)forma);
           break;
        case 2:
           DibujaUnCirculo((Circulo)forma);
           break;
     }
  }

  private void DibujaUnCirculo(Circulo c)
  {//pinta un círculo...
```

```
  }

  private void DibujaUnRectangulo(Rectangulo r)
  {//pinta un rectángulo
  }
}
```

Vamos siendo profesionales y ya hemos visto sólo con leer los requisitos que circulo y rectángulo son formas así que extraemos una interfaz IForma y en nuestro editor tenemos un método DibujarForma el cual, dependiendo del tipo de la forma que se le pase como parámetro dibuja una forma u otra pero…

¿Por qué incumple el principio abierto / cerrado?

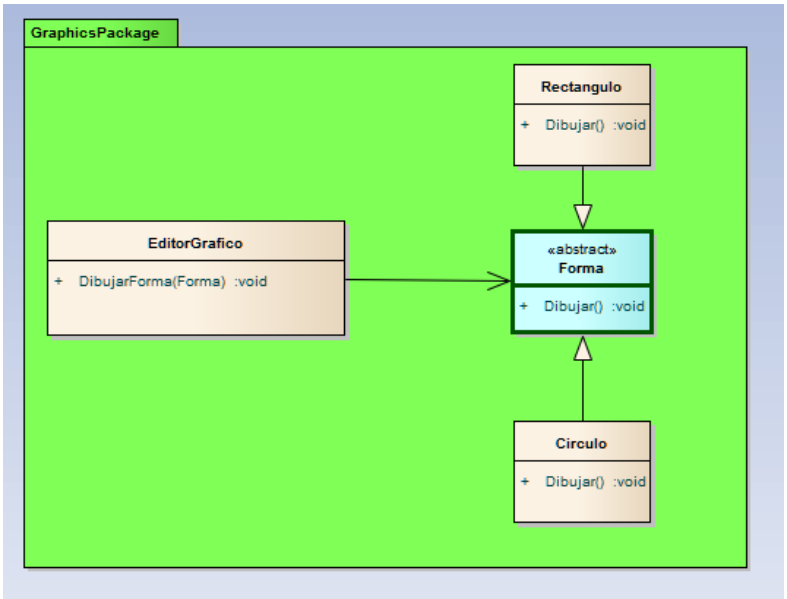
Por varias razones, primero, atendiendo a éste principio ésta clase editor gráfico no está abierta a la extensión, y si quisiéramos añadir una nueva figura geométrica para su pintado... habría que crear la nueva clase y habría que modificar el método público y añadir uno privado para el pintado de dicha nueva forma... vale, y además si nos fijamos en el primer principio añadir una nueva forma sería un motivo para el cambio y añadir una nueva funcionalidad como por ejemplo BorrarForma, EditarForma o cualquier otra cosa que queramos que haga nuestro editor gráfico, entonces ya tenemos un segundo motivo para el cambio... Por tanto éste ejemplo incumple los dos primeros principios (y alguno más...)

Bueno, que no cunda el pánico... Somos o vamos a ser analistas, arquitectos o ingenieros de software... Pensemos un poco. Yo aquí veo que las distintas formas que ahora tenemos y probablemente las que tengamos si nos piden que añadamos alguna otra tendrán una responsabilidad, la de dibujarse a sí mismas. Si retorciéramos todo éste código y en vez de una interfaz tuviéramos una clase abstracta Forma de la que heredaran las demás e implementaran un método abstracto Dibujar nos quitaríamos de un plumazo el problema de la extensión y el problema del motivo de cambio...

Solución:

¿Cuál es la responsabilidad principal del Editor Gráfico? ¿Debería el Editor conocer cómo se dibuja cada forma? Una buena pista sería: “Abstraeros un poco”…

UML



Lo que buscamos es esto, más simple, más sencillo, más robusto, más cerrado a las modificaciones pues un círculo siempre se pinta igual y un rectángulo lo mismo pero a la vez abierto a la extensión porque podemos añadir las formas que queramos sin tener que modificar código anterior en el EditorGráfico. ¿Os gusta? Siempre que tengáis comportamientos que dependan del tipo pensad en si se puede implementar con una clase abstracta... Hay muchos ejemplos en distintos lenguajes sobre éste tipo de soluciones...

CÓDIGO

```
public abstract class Forma
{
   public abstract void Dibujar();

   protected void DibujarComun()
   {
      // Parte común en todos los métodos dibujar para todas las formas...
   }
}

public class Rectangulo : Forma
{
   public override void Dibujar()
   {
      // ¿Hay algo común en todos los métodos dibujar?
      DibujarComun();
      // Dibuja un Rectangulo
   }
}

public class Circulo : Forma
{
   public override void Dibujar()
```

```
    {
        // ¿Hay algo común en todos los métodos dibujar?
        DibujarComun();
        // Dibuja un círculo
    }
}

public class EditorGrafico
{
    public void DibujarForma(Forma forma)
    {
        forma.Dibujar();
    }
}
```

El código de esto es realmente simple... Además, una de las ventajas de las clases abstractas es que podemos colocar en ellas todo el código que es común y nos evitamos tener que mantener bloques iguales de código en distintos sitios, cosa que suele ser una auténtica pesadilla, amén de lo que supone esto si estamos haciendo TDD pues lo común se testea una vez y lo particular también...

¿No es genial? De una forma elegante hemos ganado en simplicidad, sólo hemos utilizado una clase abstracta con un método abstracto que será implementado por cada clase hija. Voy a insistir en mi propia traducción de éste principio, si has de cambiar una clase hazlo usando herencia y polimorfismo. No debes temer el uso de clases abstractas ni interfaces sino al revés, potenciarlo, ya que son una ayuda contra el acoplamiento de tu software que te protegen frente a cambios.

L → LSP → Liskov Substitution Principle

Principio de sustitución de Liskov (Bárbara Liskov)

Enunciado original: "Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it."

Traducción literal: "Las funciones que utilicen punteros o referencias a clases base deben ser capaces de usar objetos de clases derivadas de éstas sin saberlo."

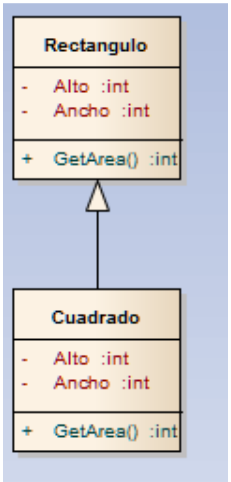
Interpretación: Las subclasses deben comportarse adecuadamente cuando sean usadas en lugar de sus clases base.

Ejemplo 3

Enunciado:

Un cuadrado es un rectángulo con la peculiaridad de que sus lados miden exactamente lo mismo, al menos eso es lo que ha deducido el tan altamente cualificado y estimado grupo de análisis que nos proporciona los requisitos para éste tan poco original ejemplo y quien nos proporciona a su vez el siguiente diagrama estático de clases:

UML ORIGINAL:



CÓDIGO ORIGINAL:

Nosotros, desarrolladores bien mandados, cumplimos a rajatabla dicho diagrama y escribimos el siguiente código:

```
public class Rectangulo
{
   public int Alto { get; set; }

   public int Ancho { get; set; }

   public int GetArea()
   {
      return Alto * Ancho;
   }
}
public class Cuadrado : Rectangulo
{
   public override int Alto
   {
      get { return base.Alto; }
      set { base.Alto = base.Ancho = value; }
   }

   public override int Ancho
   {
      get { return base.Ancho; }
      set { base.Ancho = base.Alto = value; }
   }
}

public class Prueba
{
   private static Rectangulo GetNewRectangulo()
   {
      // Podría ser cualquier objeto que también sea un rectángulo...
      // por ejemplo un cuadrado... ¿No?
      return new Cuadrado();
   }
```

```
    public static void Main()
    {
        var r = GetNewRectangulo();

        // El usuario "sabe" que r es un rectángulo
        // y asume que puede darle valor al alto y al ancho...
        r.Alto = 2;
        r.Ancho = 3;

        Console.WriteLine(r.GetArea());
            // Y al ver la consola el usuario dice: WTF!
    }
}
```

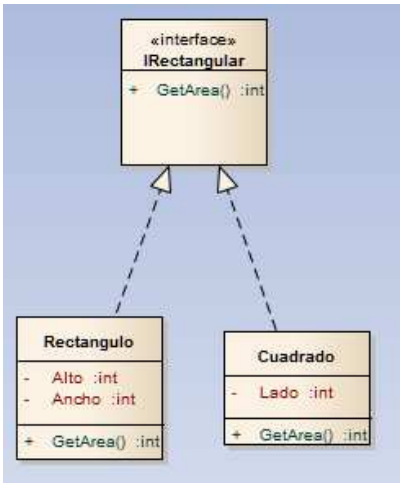
¿POR QUÉ INCUMPLE EL PRINCIPIO?

¿Se ha comportado adecuadamente la subclase en sustitución de la clase padre? Puede que sea cierto que un cuadrado es un caso particular de un rectángulo en muchos aspectos pero, ¿en el caso del área no se comportan igual?

Solución:

Debido a que el comportamiento de un cuadrado no es coherente con el del rectángulo, la mejor solución es no tener un cuadrado heredando de un rectángulo sino crear una interfaz de la que hereden tanto cuadrado como rectángulo... ¡Benditas interfaces!

UML



CÓDIGO

```
public interface IRectangular
{
   void SetAlto(int alto);
   void SetAncho(int ancho);
   int GetArea();
}

public class Rectangulo : IRectangular
{
   private int _alto;
   private int _ancho;

   public void SetAlto(int alto)
   {
      _alto = alto;
   }

   public void SetAncho(int ancho)
   {
      _ancho = ancho;
   }

   public int GetArea()
   {
      return _alto * _ancho;
   }
}


public class Cuadrado : IRectangular
{
   private int _lado;

   public void SetAlto(int alto)
   {
      _lado = alto;
   }

   public void SetAncho(int ancho)
   {
      _lado = ancho;
   }

   public int GetArea()
   {
      return _lado * _lado;
   }
}
```

I → ISP → Interface Segregation Principle

Principio de Segregación de Interfaces

Enunciado original: "Clients should not be forced to depend upon interfaces that they do not use."

Traducción literal: "Los clientes no deberían ser forzados a depender de interfaces que no utilizan."

Interpretación: Mantén las interfaces pequeñas y cohesivas, que puedan coexistir unas con otras.

Ejemplo 4

Enunciado:

Supongamos la siguiente interfaz:

```
public interface ITrabajador
{
   void Trabajar();
   void Descansar();
   void Comer();
}
```

Imaginemos las siguientes implementaciones de dicha interfaz:

```
public class TrabajadorPlanta : ITrabajador
{
   public void Trabajar()
   {
      // Trabaja mucho
   }

   public void Descansar()
   {
      // Descansa poco
   }

   public void Comer()
   {
      // Nada, como en casa y aquí no se contempla que un trabajador de planta pueda comer…
      // ¿Para qué?
   }
}
```

```
public class Desarrollador : ITrabajador
{
   public void Trabajar()
   {
      // Parece que bastante
   }

   public void Descansar()
   {
      // "… es que estoy compilando el código…"
   }
```

```
    public void Comer()
    {
        // y también beber café
    }
}
```

```
public class Robot : ITrabajador
{
    public void Trabajar()
    {
        // Soy el que mas trabajo, de eso no cabe duda
    }

    public void Descansar()
    {
        // ¿Descansar  yo?
    }

    public void Comer()
    {
        // ¿?
    }
}
```

Y para "controlar" a éstas implementaciones ¿qué tenemos? Pues un…

```
public class JefeDespota
{ // éste no falla…
    private ITrabajador _trabajador;

    public void ElegirTrabajador(ITrabajador trabajador)
    { // ¿Y si me elige a mí?
        _trabajador = trabajador;
    }

    public void Mandar()
    { // ¡A ver qué pide ahora!
        _trabajador.Trabajar();
    }
}
```

¿Por qué incumple el principio de segregación de interfaces?

¿Qué significa segregar? Pues la RAE dice:

segregar. (Del lat. *segregāre*).

1. tr. Separar o apartar algo de otra u otras cosas.

2. tr. Separar y marginar a una persona o a un grupo de personas por motivos sociales, políticos o culturales.

3. tr. Secretar, excretar, expeler.

¿Qué sucede con las implementaciones anteriores? ¿Por qué el robot tiene que implementar Descansar y comer si nunca va a descansar y dudo mucho que coma nada?

Solución:

Nos quedamos con la primera definición de la RAE, pero ¿cómo podemos separar unas cosas de otras? ¿Os suena por ejemplo ISerializable? Significa que cualquier clase que implemente ISerializable será serializable, hagamos lo mismo con Trabajar, Comer y Descansar, podemos crear varias interfaces que se llamen ITrabajar o ITrabajable…

CÓDIGO

```
public interface ITrabajar
{
   void Trabajar();
}


interface IDescansar
{
   void Descansar();
}


public interface IComer
{
   void Comer();
}

public class TrabajadorJornadaCompleta:ITrabajar, IDescansar, IComer
{
   public void Trabajar()
   {

   }

   public void Descansar()
   {

   }

   public void Comer()
   {

   }
}

public class TrabajadorMediaJornada : ITrabajar, IDescansar
{
   public void Trabajar()
   {

   }
```

```
    public void Descansar()
    {

    }
}

public class Robot:ITrabajar
{
    public void Trabajar()
    {

    }
}
```

D → DIP → Dependency Inversion Principle

Principio de Inversión de Dependencias

Enunciado original: "A. High level modules should not depend upon low level modules. Both should depend upon abstractions.

B. Abstractions should not depend upon details. Details should depend upon abstractions."

Traducción literal: "A. Módulos de alto nivel no deberían depender de módulos de bajo nivel. Ambos deberían depender de abstracciones.

B. Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones."

Interpretación: para conseguir robustez y flexibilidad y para posibilitar la reutilización haz que tu código dependa de abstracciones y no de concreciones, esto es, utiliza muchas interfaces y muchas clases abstractas y, sobretodo, expón, por constructor o por parámetros, las dependencias que una clase pueda tener.

Ejemplo

Enunciado:

Nuestro cliente nos da el siguiente requisito:

"*Quiero que la casa tenga puerta y ventana*"

Los clientes SIEMPRE son así, parece muy simple y en realidad lo es, hasta que luego te dicen:

"*También quiero que las puertas y las ventanas sean personalizables*"…

CÓDIGO ORIGINAL:

Codificar esto no tiene ninguna complicación, ¿no es cierto?:

```
public class Door
{
}

public class Window
{
}

public class House
{
   private Door _door;
   private Window _window;

   public House()
   {
      _door = new Door();
      _window = new Window();
   }
}
```

¿Por qué incumple el principio de inversión de dependencias?

En este ejemplo House depende de Door y de Window, es decir, de clases concretas y no de abstracciones.

Solución:

Podríamos mejorarlo así:

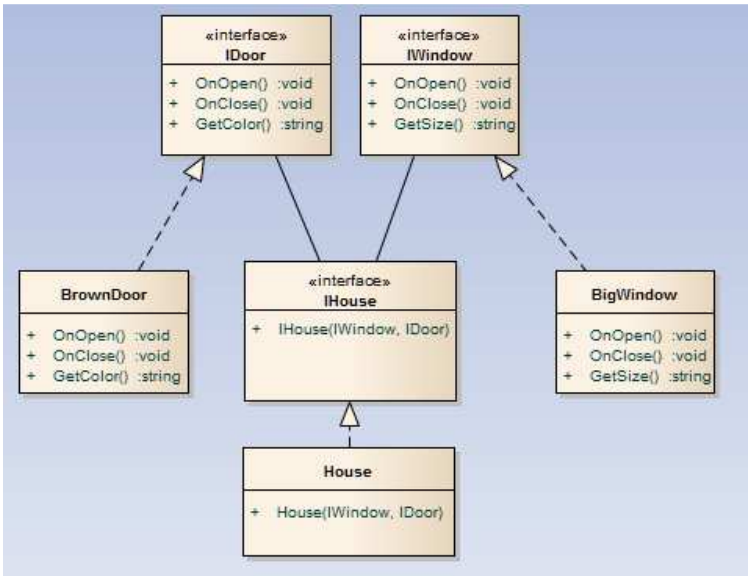
```
public class House
{
   private IDoor _door;
   private IWindow _window;

   public House()
   {
      _door = new Door();
      _window = new Window();
   }

   public IDoor Door;
   public IWindow Window;
}
```

Pero ésta solución es medio buena, ya que aunque al exponer dos propiedades que son Interfaces estamos dependiendo de abstracciones en vez de en concreciones en

el constructor hemos concretado creando una nueva puerta y una nueva ventana… Mejoremos esto aún más y hagamos lo que se conoce como "Inyección de dependencias" o "Dependency Injection" (DI)

```
public interface IDoor
{
   string GetColor();
   void OnOpen();
   void OnClose();
}

public class BrownDoor : IDoor
{
   public void OnOpen()   {   }
   public void OnClose()   {   }
   public string GetColor()
   {
      return "Soy una puerta marrón";
   }
}

public interface IWindow
{
   string GetSize();
   void OnOpen();
   void OnClose();
}

public class BigWindow : IWindow
{
   public void OnOpen()   {   }
   public void OnClose()   {    }
   public string GetSize()
   {
      return "Soy una ventana grande";
   }
}
public class House
{
   private IDoor _door;
   private IWindow _window;

   public House(){}

   public House(IDoor door, IWindow window)
   {
      _door = door;
      _window = window;
   }

   public IDoor Door;
   public IWindow Window;
}
```

UML



RESUMEN

Antes de tomarnos un pequeño y merecido descanso.

1º Espero no haberos aburrido demasiado con cosas tan básicas… Seguro que ya sabíais todo esto.

2º Espero que, al menos a grandes rasgos, hayáis entendido qué es lo que persigue cada principio de SOLID y que veáis las ventajas que conllevan la aplicación de estos principios tanto en las fases de análisis como durante el desarrollo… ¿Cuáles de éstos principios creéis que ayudan más a un desarrollo ágil practicando TDD? Desde luego la inversión (e inyección) de dependencias es ideal para hacer Mocking en tests unitarios y el principio de única responsabilidad acota mucho las clases de testeo, el principio abierto/cerrado, Liskov y la segregación de interfaces ayudan mucho a disminuir el acoplamiento y en general, la suma de todas estas prácticas nos preparan frente a los cambios… Mi resumen total de SOLID es que aplicar SOLID es como tener una malla de protección a la hora de hacer un triple salto mortal, quizá sin protección consiga dar el salto pero con protección, como las probabilidades de caer son altas, si me caigo, estoy seguro. Sin protección, si caigo, estoy muerto. Mi experiencia me dice que, hasta en desarrollos propios en los que el cliente soy yo, el cliente SIEMPRE quiere cambios y, por tanto, el cambio debe ser bienvenido.

3º Que después de este rollazo estéis dispuestos a seguir con otro tema parecido: los patrones GRASP (General Responsibility Assignment Software Patterns), o patrones de asignación de responsabilidades.

Es el momento de tomar un descanso…

GRASP: Patrones generales de asignación de responsabilidades

En el mundo anglosajón gustan mucho los juegos de palabras. Esto lo digo porque, de igual forma que estoy seguro de que SOLID, que significa sólido, no fue elegido al azar, no creo que sea casualidad que se haya elegido el acrónimo GRASP para designar a los "General Responsibility Assignment Software Patterns", o lo que es lo mismo, patrones generales de asignación de responsabilidades en software ya que la palabra 'grasp' significa comprender, entender o alcanzar.

GRASP son una serie de buenas prácticas enfocadas a la calidad del software, calidad "estructural" por llamarla de alguna manera, ya que no se ha centrado en pruebas sino en la estructura y las responsabilidades de las clases que componen el software que desarrollamos. GRASP hay que considerarlo como una serie de buenas prácticas, unos pocos consejos, que hay que seguir si se quiere hacerlo bien.

GRASP son siete patrones o “consejos”:

1. Alta cohesión y bajo acoplamiento
2. Controlador
3. Creador
4. Experto en información
5. Fabricación pura
6. Indirección
7. Polimorfismo
8. Variaciones protegidas

Al igual que con SOLID vamos a ir uno por uno viendo primero la definición, vamos a entenderla y vamos a aplicar cada patrón de una manera lo más práctica posible. En Internet probablemente los encontréis en otro orden, en mi opinión el orden alfabético, en castellano, es tan bueno como cualquier otro. Vamos a ello:

Alta cohesión y bajo acoplamiento

¿Qué significa? ¿Qué es cohesión y acoplamiento?

El grado de cohesión mide la coherencia de una clase, esto es, lo coherente que es la información que almacena una clase con las responsabilidades y relaciones que ésta tiene con otras clases.

El grado de acoplamiento indica lo vinculadas que están unas clases con otras, es decir, lo que afecta un cambio en una clase a las demás y por tanto lo dependientes que son unas clases de otras.

Como se ve claramente, los conceptos de cohesión y acoplamiento están íntimamente relacionados. Un mayor grado de cohesión implica uno menor de acoplamiento. Maximizar el nivel de cohesión intramodular en todo el sistema resulta en una minimización del acoplamiento intermodular.

Alta cohesión

Nos dice que la información que almacena una clase debe de ser coherente y debe estar, en la medida de lo posible, relacionada con la clase. Los puritanos y teóricos diferencian 7 tipos de cohesión:

1. **Cohesión coincidente:** el módulo realiza múltiples tareas pero sin ninguna relación entre ellas.
2. **Cohesión lógica:** el módulo realiza múltiples tareas relacionadas pero en tiempo de ejecución sólo una de ellas será llevada a cabo.
3. **Cohesión temporal:** las tareas llevadas a cabo por un módulo tienen, como única relación el deber ser ejecutadas al mismo tiempo.
4. **Cohesión de procedimiento:** la única relación que guardan las tareas de un módulo es que corresponden a una secuencia de pasos propia del producto.
5. **Cohesión de comunicación:** las tareas corresponden a una secuencia de pasos propia del producto y todas afectan a los mismos datos.
6. **Cohesión de información:** las tareas llevadas a cabo por un módulo tienen su propio punto de arranque, su codificación independiente y trabajan sobre los mismos datos. El ejemplo típico: OBJETOS
7. **Cohesión funcional**: cuando el módulo ejecuta una y sólo una tarea, teniendo un único objetivo a cumplir.

Bajo acoplamiento

Es la idea de tener las clases lo menos ligadas entre sí que se pueda, de tal forma que, en caso de producirse una modificación en alguna de ellas, tenga la mínima repercusión posible en el resto de clases, potenciando la reutilización, y disminuyendo la dependencia entre las clases. También hay varios tipos de acoplamiento.

1. **Acoplamiento de contenido:** cuando un módulo referencia directamente el contenido de otro módulo. (hoy en día es difícil verlo, quizá es más fácil verlo en entornos de programación funcional)
2. **Acoplamiento común:** cuando dos módulos acceden (y afectan) a un mismo valor global.
3. **Acoplamiento de control:** cuando un módulo le envía a otro un elemento de control que determina la lógica de ejecución del mismo.

Pero basta de palabrería, pasemos a los ejemplos. Para éste primer principio tengo varios ejemplos malos y uno bueno.

Sin conocer nada de C#,

```
public class ClaseDeLogicaDeNegocio
{
   public void DoSomething()
   {
      // Coge parámetros de configuración
      var umbral = int.Parse(ConfigurationManager.AppSettings["umbral"]);
      var connectionString = ConfigurationManager.AppSettings["connectionString"];

      // Vamos a por datos...
      var sql = @"select * from cosas like parametro > ";
      sql += umbral;
```

```
        using (var connection = new SqlConnection(connectionString))
        {
            connection.Open();
            var command = new SqlCommand(sql, connection);
            using (var reader = command.ExecuteReader())
            {
                while (reader.Read())
                {
                    var nombre = reader["Nombre"].ToString();
                    var destino = reader["Detino"].ToString();

                    // Haz algo más de lógica de negocio por otro lado...
                    HacerMasLogicaDeNegocio(nombre, destino, connection);
                }
            }
        }
    }
    public void HacerMasLogicaDeNegocio(string nombre, string destino, SqlConnection conexion)
    {
    }
}
```

¿No huele un poco mal éste código? Aunque éste código podría haberlo escrito yo, cosas peores he escrito, leo esto y tengo que soltar un tremendo WTF! ¿Quién tiene que arreglar esto? Nadie nace sabiendo…

Otro ejemplo, imagina que tenemos la clase mensaje:

```
class Mensaje
{
    private string _para;
    private string _asunto;
    private string _mensaje;

    public Mensaje(string to, string subject, string message)
    {
        _para = to;
        _asunto = subject;
        _mensaje = message;
    }
    public void Enviar()
    {
        // envia el mensaje...
    }
}
```

Imagina que después alguien pide que se haga login antes del envío y el desarrollador hace lo siguiente:

```
class MensajeVersion2
{
    private string _para;
    private string _asunto;
    private string _mensaje;
    private string _nombreUsuario;

    public MensajeVersion2(string to, string subject, string message)
    {
        _para = to;
        _asunto = subject;
```

```
        _mensaje = message;
    }

    public void Enviar()
    {
        // envia el mensaje...
    }

    public void Login(string nombreUsuario, string contraseña)
    {
        _nombreUsuario = nombreUsuario;
        // code to login
    }
}
```

De nuevo: WTF!

¿Quién ha tirado éste código? ¿Cuántos principios SOLID incumple? ¿Se ha hecho mediante TDD? (lo dudo mucho) ¿Quién lo va a mantener? Espero que no me toque a mí porque tendré que refactorizarlo y darle la vuelta del todo para que sea mantenible… ¿Cómo lo hago?

Una buena solución podría ser la inversión de dependencias, la D de SOLID así:

```
public interface IServicioEmail
{
    void Enviar(string asunto, string mensaje);
}
public interface IServicioLogin
{
    void Login(string nombreUsuario, string contraseña);
}

public interface IUserController
{
    void Registrar();
}

public class UserController : IUserController
{
    private readonly IServicioEmail _servicioEmail;
    private readonly IServicioLogin _servicioLogin;

    public UserController(IServicioEmail servicioEmail, IServicioLogin servicioLogin)
    {
        // Se harían las debidas comprobaciones y después:
        _servicioEmail = servicioEmail;
        _servicioLogin = servicioLogin;
    }

    public void Registrar()
    {   // Autenticar y enviar:
        _servicioLogin.Login("usuario", "contraseña");
        _servicioEmail.Enviar("asunto", "mensaje");
    }
    // Etcétera…
}
```

¿Qué principios SOLID se ven en éste código?

S: cada clase/interfaz cumple un único cometido.

O: UserController está abierto a la extensión pero cerrada al cambio (O), lo cuál no significa que por requisitos tenga que crecer…

L: se puede buscar que la jerarquía de las implementaciones de los servicios cumpla con el principio de sustitución de Liskov (L) y siempre puedas sustituir un servicio por su clase padre

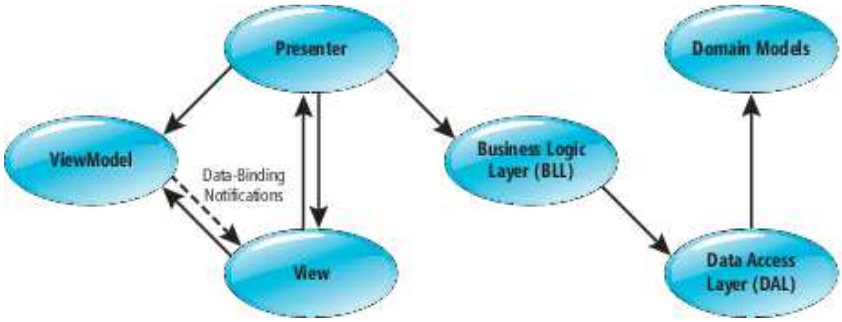
I: hay segregación de interfaces (I) y por eso se consiguen otros principios

D: también hay inversión (inyección) de dependencias (D).

Al hacerlo bien hemos aplicado todos los principios SOLID y tenemos un código muy cohesivo, es decir, la información de cada clase es coherente con ella misma, y muy desacoplado, esto es, las clases están muy desligadas unas de otras habiendo potenciado esto mediante el uso de interfaces y demás…

Controlador

El patrón controlador es un patrón que sirve como intermediario entre una determinada interfaz y el algoritmo que la implementa, de tal forma que es el controlador quien recibe los datos del usuario y quien los envía a las distintas clases según el método llamado. Este patrón sugiere que la lógica de negocio debe estar separada de la capa de presentación, lo que aumenta la reutilización de código y permite a la vez tener un mayor control. Hay muchas arquitecturas de aplicación que se basan en esto, desde el famoso MVC que ya vi estudiando la carrera hasta la arquitectura MVVM tan utilizada últimamente en aplicaciones de escritorio y la última que he practicado profesionalmente, MVVMP, que no es más que un refinamiento sutil de la anterior.



Se recomienda dividir los eventos del sistema en el mayor número de controladores para poder aumentar así la cohesión y disminuir el acoplamiento.

Un Controlador:

- Debería ser el primer objeto llamado después de un cambio en la interfaz de usuario.
- Controla/ejecuta un caso de uso. No hace demasiado por si solo, controla, coordina.
- Pertenece a la capa de aplicación o a la de servicios.

Éste ejemplo se centra en que el controlador debe ser el primer objeto llamado por objetos de la UI:

```
public class AlbumView
{
}

public class AlbumPresenter
{
   AlbumView _vista;
   AlbumController _controlador;

   public void EtiquetaAdded(string etiqueta)
   {
      _controlador.EtiquetarFoto(etiqueta);
   }
}

public class AlbumController
{
   public void EtiquetarFoto(string newTag)
   {
   }
}
```

Este otro ejemplo se centra en que el controlador no debería hacer demasiado, tan sólo coordinar:

```
public class AlbumView
{
}

public class AlbumPresenter
{
   AlbumView _vista;
   AlbumController _controlador;

   public void EtiquetaAdded(string etiqueta)
   {
      _controlador.EtiquetarFoto(etiqueta);
   }
}

public class AlbumController
{
   private RepositorioDeFotos _repository;

   public void EtiquetarFoto(string nuevaEtiqueta)
   {
      var foto = _repository.ReadFoto();
      foto.AddEtiqueta(nuevaEtiqueta);
```

```
        _repository.UpdateFoto(foto);
    }
}

public class RepositorioDeFotos
{
   // Operaciones CRUD...
    public Foto ReadFoto()
    {
        return new Foto();
    }

    public void UpdateFoto(Foto foto)
    {
    }
}

public class Foto
{
    public void AddEtiqueta(string tag)
    {
    }
}
```

Creador

La creación de instancias es una de las actividades más comunes en un sistema orientado a objetos. En consecuencia es útil contar con un principio general para la asignación de las responsabilidades de creación. Si se asignan bien el diseño puede soportar un bajo acoplamiento, mayor claridad, encapsulamiento y reutilización.

El patrón creador nos ayuda a identificar quién debe ser el responsable de la creación o instanciación de nuevos objetos o clases. Éste patrón nos dice que la nueva instancia podrá ser creada por una clase si:

- Contiene o agrega la clase.
- Almacena la instancia en algún sitio (por ejemplo una base de datos)
- Tiene la información necesaria para realizar la creación del objeto (es 'Experta')
- Usa directamente las instancias creadas del objeto

Una de las consecuencias de usar este patrón es la visibilidad entre la clase creada y la clase creadora. Una ventaja es el bajo acoplamiento, lo cual supone facilidad de mantenimiento y reutilización.

Veamos, con una serie de ejemplos de cómo aplicar éste patrón correctamente bajo distintas circunstancias.

Ejemplo base:

```
public class ListaDeClientesPresenter
{
    Cliente _clientes;

    public void AddButtonClicked()
```

```
    {
    }
}

public class Cliente
{
    List<Pedido> _pedidos;
}

public sealed class Pedido
{
}
```

Donde quiera que se llame al método AddButtonClicked se generará un nuevo pedido y se lo "daremos" a una instancia de un cliente, pero ¿quién debería crear ese pedido? Lo que nos dice éste principio es que esa responsabilidad debería recaer en Cliente ya que contiene Pedidos.

Ejemplo 2:

```
public class ListaDeClientesPresenter
{
    Cliente _cliente;

    public void AddButtonClicked()
    {
        _cliente.AddPedido();
    }
}

// En éste caso contiene o agrega la clase y además maneja varias instancias...
public class Cliente
{
    List<Pedido> _pedidos;

    public void AddPedido()
    {
        var nuevoPedido = new Pedido();
        _pedidos.Add(nuevoPedido);
    }
}

public sealed class Pedido
{
}
// Pero hay más posibilidades...
```

Ejemplo 3:

```
// Si la clase guarda (persiste) la clase creada...
// Lo más fácil es implementar un repositorio:

public class ListaDeClientesPresenter
{
    private RepositorioDePedidos repositorioDePedidos;

    public void AddButtonClicked()
    {
```

```
    }
}

public class Cliente
{
    List<Pedido> _pedidos;
}

public class RepositorioDePedidos
{
    Stream _pedidosXml;

    public Pedido LoadOrders()
    {
        // Cargo el pedido y lo devuelvo
        return pedido;
    }
}

public sealed class Pedido
{
    string _id;

    public Pedido(string id)
    {
        _id = id;
    }
}
```

Ejemplo 4:

```
// La clase tiene la información necesaria para crear la nueva clase
// Por ejemplo si queremos que Pedido tenga el Id de cliente...
public class ListaDeClientesPresenter
{
    Cliente _cliente;
    public void AddButtonClicked()
    {
        _cliente.AddPedido();
    }
}

public class Cliente
{
    List<Pedido> _pedidos;
    int _id;

    public void AddPedido()
    {
        var nuevoPedido = new Pedido(_id);
        _pedidos.Add(nuevoPedido);
    }
}

public sealed class Pedido
{
    private int _clienteId;
    public Pedido(int clienteId)
    {
        _clienteId = clienteId;
    }
}
```

Ejemplo 5:

```
// La clase usa directamente la clase creada
// Por ejemplo si queremos que el Cliente gestione Pedido/s
public class ListaDeClientesPresenter
{
   Cliente _cliente;

   public void AddButtonClicked()
   {
   }
}

public class Cliente
{
   Pedido _pedido;

   // La responsabilidad de crear esta clase que él mismo maneja tantas veces recae sobre él aquí
o en cualquier otro lugar...

   public Cliente()
   {
      _pedido = new Pedido();
   }

   public void DeleteOrder()
   {
      _pedido.Borar();
   }

   public void RenameOrder(string newName)
   {
      _pedido.Modificar();
   }
}

public sealed class Pedido
{
   public void Borar(){}
   public void Modificar(){}
}
```

De nuevo no hay mucho más que decir sobre éste patrón, tan solo que hay que tener en cuenta esos cuatro puntos a la hora de instanciar un objeto y preguntarse si el lugar en el que se está realizando es el idóneo o no. Darse cuenta de que no se está cumpliendo con el patrón creador es un buen motivo para refactorizar nuestro código.

Experto en información

Éste es el principio básico de asignación de responsabilidades, la S de SOLID, y para mí es de los más importantes de los patrones o principios GRASP.

Experto en información nos dice que la responsabilidad de la creación de un objeto o la implementación de un método, debe recaer sobre la clase que conoce toda la información necesaria para crearlo o ejecutarlo, de este modo obtendremos un

diseño con mayor cohesión y cuya información se mantiene encapsulada, es decir, disminuye el acoplamiento.

```
public class Informe
{
   public int[] Parciales { get; set; }
}
public class InformePresenter
{
   Informe _informe;

   public void CalculateTotalButtonClicked()
   {
      var total = 0;
      foreach (var parcial in _informe.Parciales)
      {
         total = total + parcial;
      }

      // TIP: éste bucle lo podemos convertir en un una expresión funcional usando Linq:
      // var total = _informe.Parciales.Aggregate(0, (current, parcial) => current + parcial);
   }
}
```

De nuevo: ¿qué falla aquí? ¿Por qué viola éste principio?

Por lo mismo que viola la S de SOLID, la responsabilidad de InformePresenter es, como su propio nombre indica, presentar un informe y en éste caso está calculando el total. ¡Mal! El total debería dárselo la clase Informe, que es la que tiene toda la información al respecto, ella es la experta y quien debería calcularlo.

```
public class Informe
{
   public int[] Parciales { get; set; }

   public void CalcularTotal()
   {
      var total = 0;
      foreach (var parcial in Parciales)
      {
         total = total + parcial;
      }
   }
}
public class InformePresenter
{
   private Informe _informe;

   public void CalculateTotalButtonClicked()
   {
     _informe.CalcularTotal();
   }
}
```

Regla de oro:

Éste es el principio que se debería aplicar más a menudo, el que hay que tener más en cuenta. La suma de "Experto en información" y SRP es que una clase sólo debería tener un motivo para cambiar y debería ser ella misma la encargada de crear los objetos e implementar los métodos sobre los que es experta.

Fabricación Pura

La fabricación pura se da en las clases que no representan un ente u objeto real del dominio del problema sino que se han creado intencionadamente para disminuir el acoplamiento, aumentar la cohesión y/o potenciar la reutilización del código.

La fabricación pura es la solución que surge cuando el diseñador se encuentra con una clase poco cohesiva y que no tiene otra clase en la que implementar algunos métodos. Es decir que se crea una clase "inventada" o que no existe en el problema como tal, pero que, añadiéndola, logra mejorar estructuralmente el sistema. Como contraindicación deberemos mencionar que al abusar de este patrón suelen aparecer clases función o algoritmo, esto es, clases que tienen un solo método.

Un ejemplo de porqué tendríamos que inventarnos una clase fuera del dominio del problema, imaginemos que estamos programando el famoso Angry Birds:

```
public class PajaroEnfadado
{
   public void Volar()
   {
   }

   public void Mostrar()
   {
      // En una interfaz de usuario
   }
}
```

El método Mostrar "acopla" al pájaro a la interfaz de usuario y como no queremos enfadarlo aún más haríamos lo siguiente:

```
public class PajaroEnfadado
{
   public void Volar()
   {
   }
}

public class PajaroEnfadadoPresenter
{
   public PajaroEnfadadoPresenter(PajaroEnfadadoView view)
   {
      view.Mostrar(new PajaroEnfadado());
   }
}
```

```
public class PajaroEnfadadoView
{
   public void Mostrar(PajaroEnfadado pajaroEnfadado)
   {
   }
}
```

Ahora tenemos dos nuevas clases, view y presenter. Estas dos clases no existen en el dominio real del problema pero desacoplan enormemente la implementación del pájaro de la interfaz de usuario... Y nos dejan que la implementación del pájaro enfadado se centre en la lógica de negocio y en nada más, una única responsabilidad...

¿No te suena? Seguro que si pero quizá nunca te habías preguntado el porqué de las arquitecturas de n capas, me da igual que sea MVC que MVP, que MVVM o MVVMP. Es una idea simple e intuitiva a la que ahora puedes ponerle un nombre y decir que se basa en el patrón GRASP de "Fabricación pura".

Indirección

Voy llegando al final de los patrones o principios GRASP, éste, la indirección, es fundamental y, como buenos profesionales, deberíamos tenerlo muy presente en nuestros análisis y decisiones arquitecturales.

El patrón de indirección nos permite mejorar el bajo acoplamiento entre dos clases asignando la responsabilidad de la mediación entre ellos a una clase intermedia.

Problema: ¿dónde debo asignar responsabilidades para evitar o reducir el acoplamiento directo entre elementos y mejorar la reutilización? Es decir, si sabemos que un objeto utiliza otro que muy posiblemente va a cambiar, ¿cómo protegemos a un objeto frente a cambios previsibles?

Solución: asignar la responsabilidad a un objeto que medie entre los elementos para proteger al primer objeto de los posibles cambios del segundo.

Imaginemos que estamos desarrollando un componente de software para hacer logs dentro de una gran arquitectura…

```
public class CualquierPresentador
{
   Log4Net _logger;

   public void AlgoHaSucedido()
   {
      _logger.Log("Algo ha sucedido y hemos tratado en el presentador dicho algo...");
   }
}

public class Log4Net
{
   public void Log(string message)
   {
```

```
    }
}
```

En éste caso Log4Net podría cambiar, de hecho esperamos que cambie porque lo estamos utilizando, si, pero se nos queda cortos. ¿Cómo aplicamos la indirección?

¿Qué tal si creamos un servicio de loggin intermedio? Así el presentador utilizará el servicio y con un poco de suerte los cambios en Log4Net no afectarán demasiado al resto del programa.

```
public class CualquierPresentador
{
    private ServicioLog _logger;

    public void AlgoHaSucedido()
    {
        _logger.Log("Algo ha sucedido y hemos tratado en el presentador dicho algo...");
    }
}

public class ServicioLog
{
    private Log4Net _logger;

    public void Log(string message)
    {
        _logger.Log(message);
    }
}

public class Log4Net
{
    public void Log(string message)
    {
    }
}
```

Este patrón es fundamental para crear abstracciones ya que nos permite introducir API's externas en la aplicación sin que tengan demasiada influencia en el código que tendría que cambiar cuando cambiara la API. Además, la primera clase podría cambiarse fácilmente para que en vez de ésta API utilizara cualquier otra.

Polimorfismo

A lo largo de todos de ésta sesión ha aparecido la palabra polimorfismo alrededor de una decena de veces y he dado por hecho que todo el mundo sabía de qué estaba hablando, pero ¿qué es polimorfismo?

Polimorfismo, en programación orientada a objetos, es un concepto muy simple con el que la gente a veces se lía, polimorfismo es permitir que varias clases se comporten de manera distinta dependiendo del tipo que sean.

Siempre que se tenga que llevar a cabo una responsabilidad que dependa de un tipo, se tiene que hacer uso del polimorfismo, es decir, asignaremos el mismo nombre a servicios implementados en diferentes objetos acordes con cada tipo.

Como de costumbre, veamos esto con un ejemplo "malo", es decir, mejorable, la explicación y un ejemplo que lo soluciona.

```
public enum TipoDeLog
{
   Debug,
   Error
}

public class Log
{
   StreamWriter _ficheroLog;

   public void Registrar(string mensaje, TipoDeLog tipoDeLog)
   {
      switch (tipoDeLog)
      {
         case TipoDeLog.Debug:
            _ficheroLog.WriteLine("[DEBUG];{0}", mensaje);
            break;
         case TipoDeLog.Error:
            _ficheroLog.WriteLine("[ERROR]:{0}", mensaje);
            break;
      }
   }
}

public class CualquierPresentador
{
   Log _log;
   public void AlgoHaSucedido()
   {
      _log.Registrar("Algo ha sucedido y queremos dejar constancia en un registro.",
TipoDeLog.Debug);
   }
}
```

¿Por qué está mal? ¿Por qué hemos de refactorizar éste código?

Se ve claramente la dependencia de la clase Log y el método registrar con el TipoDeLog. Podemos evitar esto creando una interfaz para el mensaje implementada de dos formas distintas, una por cada tipo, y sólo tendremos que pedir a la clase Log que registre un IMensaje. Y dependiendo de qué mensaje queramos registrar, quien sepa de qué tipo debe ser registrado, es decir, el experto en información, que cree una instancia concreta y llame a Log.Registrar.

```
public interface IMensajeDelLog
{
   string Valor { get; }
}

public class MensajeDebug : IMensajeDelLog
```

```
{
   readonly string _mensaje;

   public MensajeDebug(string mensaje)
   {
      _mensaje = mensaje;
   }

   public string Valor
   {
      get
      {
         return string.Format("[DEBUG];{0}", _mensaje);
      }
   }
}

public class MensajeError : IMensajeDelLog
{
   readonly string _mensaje;

   public MensajeError(string mensaje)
   {
      _mensaje = mensaje;
   }

   public string Valor
   {
      get
      {
         return string.Format("[ERROR];{0}", _mensaje);
      }
   }
}

public class Log
{
   StreamWriter _ficheroLog;

   public void Registrar(IMensajeDelLog message)
   {
      _ficheroLog.WriteLine(message.Valor);
   }
}

public class CualquierPresentador
{
   Log _log;

   public void AlgoHaSucedidoYQueremosRegistrarUnError()
   {
      _log.Registrar(new MensajeError("Algo ha sucedido y queremos dejar constancia en un
registro."));
   }

   public void AlgoHaSucedidoYQueremosRegistrarloParaDepurar()
   {// Por ejemplo una excepción
      _log.Registrar(new MensajeDebug("Algo ha sucedido y queremos dejar constancia en un
registro."));
   }
}
```

Variaciones protegidas

Siempre se ha dicho que, aplicando metodologías ágiles al desarrollo de software, el cambio es bienvenido, ahora bien, el cambio es bienvenido si lo esperamos. ¿Cuántos proyectos de software se habrán ido a pique y cuantos profesionales habrán visto su reputación por los suelos cuando el cliente ha pedido un pequeño cambio cuyas implicaciones obligaron a arquitectos y desarrolladores a empezar de cero? No sé la respuesta pero seguro que han sido demasiados.

El cambio debe ser bienvenido, claro, porque los clientes siempre quieren cambios y les cobraremos por desarrollarlos, pero no deben ser motivo de desesperación. Variaciones protegidas, es el principio fundamental de protegerse frente al cambio. Esto quiere decir que lo que veamos en un análisis previo que es susceptible de modificaciones lo envolvamos en una interfaz y utilicemos el polimorfismo para crear varias implementaciones y posibilitar implementaciones futuras de manera que quede lo menos ligado posible a nuestro sistema. De ésta forma, cuando se produzca la variación o el cambio que esperamos, dicho cambio nos repercuta lo mínimo.Este principio está muy relacionado con el polimorfismo y la indirección.

Imaginemos el siguiente código:

```
public class ImagenJpeg
{
   public void Redimensionar(int nuevoAlto, int nuevoAncho)
   {
      // Resize
   }
}

public class MostrarImagenController
{
   public void BotonRedimensionarClicked(ImagenJpeg image)
   {
      image.Redimensionar(10, 20);
   }
}
```

¿Qué problemas pueden surgir? ¿Qué podría variar? ¿Cómo lo mejoramos?

¿Qué problemas pueden surgir? Simple, puede suceder que en vez de una imagen jpeg nos manden otro tipo de imagen. ¿Y cómo lo solucionamos? También simple si has seguido toda la serie de artículos sobre GRASP y las distintas refactorizaciones que hemos hecho en cada caso, basta con añadir una interfaz IImagen con un método Redimensionar, dejando el código así:

Solución:

```
public interface IImagen
{
   void Redimensionar(int nuevoAlto, int nuevoAncho);
}
public class ImagenJpeg : IImagen
{
   public void Redimensionar(int nuevoAlto, int nuevoAncho)
   {
      // Resize
   }
}

public class MostrarImagenController
{
   public void BotonRedimensionarClicked(IImagen image)
   {
      image.Redimensionar(10, 20);
   }
}
```

¡Genial! Ya no nos tendría que importar que cambiara el tipo de imagen a redimensionar, al menos a éste nivel en nuestra aplicación, y éste ejemplo es extrapolable a casi cualquier situación.

Bueno, aquí acaba la sesión sobre SOLID y GRASP en los que hemos ido viendo las mejores formas de asignar responsabilidades a clases y métodos para ganar en cohesión, disminuir acoplamiento, aplicar mejor los principios SOLID, protegernos frente al cambio y, en definitiva, ser mejores desarrolladores.

De nuevo espero que haya servido de algo, que no os hayáis aburrido de mí y mis explicaciones y que, si creéis que todo esto es útil le deis algo de divulgación.

NOTAS FINALES

Merece la pena comentar:

- ¿Por qué SOLID y GRASP en unas charlas sobre TDD?
  - Beneficios de la extracción de interfaces y clases abstractas para Mocking en Test unitarios
  - Nomenclatura When – ItShould en Tests unitarios reforzada gracias a la separacación de responsabilidades y a la limpieza en nuestro código
- Otros princios:
  - DRY → Don't Repeat Yourself
  - KISS → Keep It Simple, Stupid!
  - YAGNI → You ain't gonna need it
- Más información:
  - Robert C Martin ("Dios" cuando hablamos de POO, autor a tener en cuenta como referencia…)
  - Libros recomendados:
    - Clean Code
    - Pragmatic Programmer