Instituto Politécnico Nacional

Escuela Superior de Cómputo

PRINCIPIOS DE DISEÑO

Investigación

Análisis y Diseño Orientado a Objetos

Profesor: Chadwick Carreto Arellano

Grupo: 2CM7

Fecha: 17 / Mayo /2018

Alumno:

Calva Hernández José Manuel 2017630201

Contenido

Introducción

¿Qué es el diseño de software?

Es un proceso de invención y selección de programas que cumplan los objetivos de un sistema software. La entrada incluye el entendimiento de los requisitos, las restricciones de entorno y los criterios de diseño. La salida del proceso de diseño está compuesta de una arquitectura de diseño que muestra como las piezas están interrelacionadas, de especificaciones de cualquier pieza nueva y de las definiciones de cualquier dato nuevo.

¿Por qué es importante?

El diseño permite modelar el sistema o producto que se va a construir, permite medir la calidad y su mejora antes de generar código y permite establecer la calidad del software.

¿Cuál es el proceso de diseño?

a) Diseño arquitectónico: Define la relación entre los elementos estructurales principales del software, los patrones de diseño que se pueden utilizar para lograr los requisitos que se han definido para el sistema, y las restricciones que afectan a la manera en que se pueden aplicar los patrones de diseño arquitectónicos.

b) Diseño de datos: Transforma el modelo del dominio de información creado en el análisis en las estructuras de datos necesarias para la implementación del software.

c) Diseño a nivel de componentes: Transforma los elementos estructurales de la arquitectura del software en una descripción procedimental de los componentes del software.

d) Diseño de la interfaz: Diseño de interfaces hombre-máquina para facilitar al usuario la utilización del sistema con el propósito de recoger información de estos hacia el sistema y ponerla a disposición de otros usuarios.

Principios de diseño

Son estandarizaciones usadas para organizar y organizar los componentes estructurales del diseño en la Ingeniería de Software. Los métodos en los que se aplican estos principios de diseño afectan el contenido expresado y el proceso de trabajo desde el principio.

Los principios de diseño ayudan a los diseñadores a construir consensos acerca del conocimiento arquitectónico, ayuda a la gente en los procesos con gran escala de ingeniería de software, ayuda a los principiantes a evitar trampas y no repetir fallos que han sido detectados en experiencias pasadas.

¿Por qué usar estos principios?

Todo desarrollo, antes o después, se ve sometido a cambios sobre la funcionalidad inicial o simplemente se requiere funcionalidad nueva.

Estos principios nos proporcionan unas bases para la construcción de aplicaciones mucho más sólidas y robustas. Permiten que los cambios y la evolución del software tenga un mínimo impacto en el código ya desarrollado tratando de evitar que lo que funciona deje de funcionar y por ello que el coste del mantenimiento sea mucho menor.

Principales principios de diseño (SOLID)

Single responsibility (Principio de responsabilidad única)

Este principio dice que cada clase tiene que tener una responsabilidad única y concreta, como dice Rober C. Martín “Una clase debería tener una y sólo una razón para cambiar”.

Es común que cuando empezamos a desarrollar se acabe tomando decisiones como meter en una clase un método a causa de que esa clase lo utiliza, el problema llega cuando más adelante ese método lo necesitamos usar en otras clases y vemos que pierde coherencia.

Supongamos que tenemos que realizar una aplicación para registrar las ofertas presentadas y las ventas cerradas. Se puede crear una clase vendedor con un método “GenerarOferta” que registrará las ofertas presentadas y otro “CerrarVenta” que registrará las ofertas ganadas.

```
public class Venta
{
    public void GenerarOferta() { }

    public void CerrarVenta() { }
}
```

A priori, el ejemplo, pude parecer correcto, pero en realidad estamos mezclando 2 conceptos (oferta y venta). La forma en que se realiza una oferta puede variar por diversos motivos, lo mismo ocurre con el procedimiento de venta que también puede cambiar, lo cual implicará modificar la clase por 2 motivos diferentes.

```
public class Oferta
{
    public void GenerarOferta() { }
}

public class Venta
{
    public void CerrarVenta(Oferta oferta) { }
}
```

Después de hacer las modificaciones, vemos que si ahora cambia la forma de realizar una oferta no impacta sobre la venta y si cambia la forma de cerrar la venta no impacta sobre la oferta.

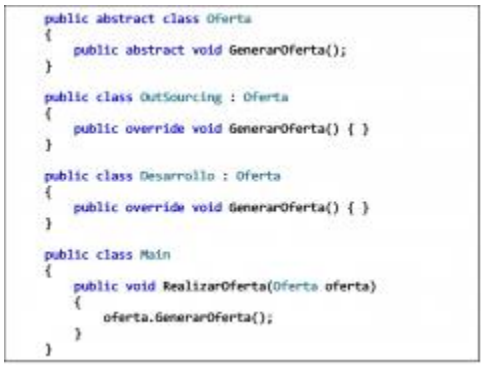
Open-Close (Principio de abierto-cerrado)

Lo que dice este principio es que no se debe modificar el código de una clase o entidad, sino que estas deben de ser extendidas y para que esto se cumpla, nuestro código debe estar bien diseñado. La forma más común para cumplir con este principio es el uso de la herencia y/o el de las interfaces. Cumplir el principio de responsabilidad única ayuda a que este otro principio también se cumpla, pero no significa que cumpliendo uno se cumpla el otro.

Sigamos con el ejemplo anterior, supongamos ahora que vamos a realizar la oferta y que dependiendo del tipo de servicio tendremos que generar una estructura diferente en cada caso. Añadiremos una propiedad de tipo enumerado en la clase Oferta que nos indique el tipo de servicio y un método por cada servicio que genere una estructura u otra, cuando realicemos la oferta tendremos que comprobar el tipo y en función de este llamar a uno u otro método.

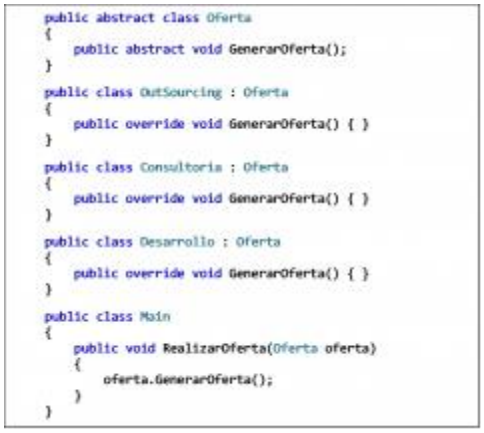
```
public enum Servicios
{
    OutSourcing,
    Desarrollo
}

public class Oferta
{
    public Servicios Servicio { get; set; }

    public void GenerarOfertaOutSourcing() { }

    public void GenerarOfertaDesarrollo() { }
}

public class Main
{
    public void RealizarOferta(Oferta oferta)
    {
        switch (oferta.Servicio)
        {
            case Servicios.OutSourcing:
                oferta.GenerarOfertaOutSourcing();
                break;
            case Servicios.Desarrollo:
                oferta.GenerarOfertaDesarrollo();
                break;
        }
    }
}
```

En este ejemplo, si queremos ofrecer un nuevo servicio nos vemos obligados a modificar la clase Oferta y el método “RealizarOferta”, por lo que estamos incumpliendo el principio abierto-cerrado. En el siguiente ejemplo mostramos como se podría solucionar.



```
public abstract class Oferta
{
    public abstract void GenerarOferta();
}

public class OutSourcing : Oferta
{
    public override void GenerarOferta() { }
}

public class Desarrollo : Oferta
{
    public override void GenerarOferta() { }
}

public class Main
{
    public void RealizarOferta(Oferta oferta)
    {
        oferta.GenerarOferta();
    }
}
```

Como podemos ver en el ejemplo, al crear por cada tipo del enumerado una clase que herede de Oferta e implemente su método “GenerarOferta”, conseguimos simplificar la funcionalidad de forma que no sea necesario realizar modificaciones si añadimos un nuevo servicio, ya que bastará con crear una nueva clase que herede de Oferta e implemente el método. En el siguiente ejemplo añadimos el servicio de consultoría.



```
public abstract class Oferta
{
    public abstract void GenerarOferta();
}

public class OutSourcing : Oferta
{
    public override void GenerarOferta() { }
}

public class Consultoria : Oferta
{
    public override void GenerarOferta() { }
}

public class Desarrollo : Oferta
{
    public override void GenerarOferta() { }
}

public class Main
{
    public void RealizarOferta(Oferta oferta)
    {
        oferta.GenerarOferta();
    }
}
```

Liskov Substitution (Principio de Sustitución de Liskov)

Este principio recibe su nombre por su creadora Barbara Liskov. Viene a decir que una clase derivada de otra debe poder ser sustituida por su clase base y debemos garantizar que los métodos de la primera no provoquen un mal funcionamiento de los métodos de la clase base.

Vamos a utilizar un ejemplo muy recurrido para explicar este principio, el del cuadrado y el rectángulo. Un cuadro no es más que un rectángulo con todos los lados iguales, intentemos llevar esto a la programación y calcular su área.

```
public class Rectangulo
{
    private int ancho;
    private int alto;

    public int getAncho()
    {
        return ancho;
    }

    public virtual void setAncho(int ancho)
    {
        this.ancho = ancho;
    }

    public int getAlto()
    {
        return alto;
    }

    public virtual void setAlto(int alto)
    {
        this.alto = alto;
    }

    public int calcularArea()
    {
        return ancho * alto;
    }
}

public class Cuadrado : Rectangulo
{
    public override void setAncho(int ancho)
    {
        base.setAncho(ancho);
        base.setAlto(ancho);
    }

    public override void setAlto(int alto)
    {
        base.setAncho(alto);
        base.setAlto(alto);
    }
}
```

Hemos creado una clase “Rectángulo” con unos métodos para obtener ancho y alto, otros para modificar el ancho y el alto, y un método que calcula el área. Hemos creado también una clase “Cuadrado” que hereda de “Rectángulo” y sobrescribe los métodos que cambian el ancho y alto para que mantengan una relación.

```
[TestMethod]
public void testArea()
{
    Rectangulo r = new Rectangulo();
    r.setAncho(5);
    r.setAlto(4);
    assertEquals(20, r.calcularArea());
}
```

El siguiente método lo usamos para testear nuestras clases. Si nos fijamos siempre que se pase un rectángulo funcionará correctamente, pero si cambiamos el rectángulo por el cuadrado no funcionará y por ello incumple el principio de Liskov.

```
public class Rectangulo
{
    public int ancho;
    public int alto;

    public Rectangulo(int ancho, int alto)
    {
        this.ancho = ancho;
        this.alto = alto;
    }

    public int calcularArea()
    {
        return ancho * alto;
    }
}

public class Cuadrado : Rectangulo
{
    public Cuadrado(int lado) : base(lado,lado) { }
}
```

Como se puede ver, hemos cambiado las clases para que sus constructores reciban ancho y alto por parámetros, el lado en caso del cuadrado. De esta forma estamos obligando en la definición del objeto a indicar el valor que tendrán sus lados, haciendo su uso más intuitivo y evitando que haya operaciones por detrás que nos lleven a confusión.

Interface Segregation (Principio de Segregación de Interfaces)

Mejor crear muchas interfaces que contengan pocos métodos, que crear pocas interfaces que definan demasiados métodos. El motivo de este principio es que ninguna clase debe de estar obligada a implementar métodos que no necesita, por ello es preferible crear varias interfaces que agrupen funcionalidad común, a englobar gran parte de esa funcionalidad en unas pocas interfaces y encontrarnos más adelante que necesitamos implementar una interface que nos obliga a implementar métodos que no necesitamos.

En el siguiente ejemplo volvemos a nuestro proyecto de las ofertas.

```
public interface IOferta
{
    void GenerarOferta();
    void EstablecerDesplazamiento();
    void EstablecerTarifa();
}

public class OutSourcing : IOferta
{
    public void GenerarOferta() { }

    public void EstablecerDesplazamiento() { }

    public void EstablecerTarifa() { }
}
public class Proyecto : IOferta
{
    public void GenerarOferta() { }

    public void EstablecerDesplazamiento()
    {
        throw new NotImplementedException();
    }

    public void EstablecerTarifa() { }
}
```

Esta vez, tendremos una interface “IOferta” que define una serie de métodos que se usan en las ofertas. Si nos fijamos, cuanto heredamos de esa interface en la clase “Proyecto” nos vemos obligados a implementar el método “EstablecerDesplazamiento”, que nos viene bien para el outsourcing, pero en proyectos que se van a desarrollar internamente no lo necesitamos y si lo usamos devolverá una excepción.

```
public interface IOferta
{
    void GenerarOferta();
    void EstablecerTarifa();
}

public interface IOutSourcing
{
    void EstablecerDesplazamiento();
}

public class OutSourcing : IOferta, IOutSourcing
{
    public void GenerarOferta() { }

    public void EstablecerDesplazamiento() { }

    public void EstablecerTarifa() { }
}

public class Proyecto : IOferta
{
    public void GenerarOferta() { }

    public void EstablecerTarifa() { }
}
```

Lo que hemos hecho para solucionarlo es crear una nueva interface que defina el método “EstablecerDesplazamiento” e implementarla solo donde se necesita. Ahora no tenemos métodos implementados que no necesitamos usar.

Dependency Inversion (Principio de Inversión de Dependencia)

Este principio busca que no existan un alto acoplamiento en las aplicaciones, ya que ello repercute en un difícil mantenimiento.

El principio quiere decir que las clases de alto nivel no tienen que depender de otras de bajo nivel, sino que ambas dependan de abstracciones, así como que las abstracciones no deben depender de los detalles, sino al contrario.

Imaginemos que tenemos que generar una oferta y que dependiendo del su tipo se generará con una estructura u otra.

```
public class Main
{
    public void GuardarOferta(IOferta oferta)
    {
        switch (oferta.GetType().Name)
        {
            case "OutSourcing":
                var docWord = new Word();
                docWord.Guardar(oferta);
                break;
            case "Proyecto":
                var docPDF = new PDF();
                docPDF.Guardar(oferta);
                break;
        }
    }
}
```

En este ejemplo disponemos de un método principal que recibe una oferta y en base al tipo de esta decidimos se guarda en Word o en PDF. El problema que encontramos es que hay un fuerte acoplamiento y dependencias entre las clases. Si más adelante se requiere que una oferta concreta se guarde en ambos formatos, nos vemos obligados a cambiar toda la lógica del método.

```
public interface IDocumento
{
    void Guardar(IOferta oferta);
}

public class Main
{
    public void GuardarOferta (IOferta oferta, IDocumento documento)
    {
        documento.Guardar(oferta);
    }
}
```

Para solucionar el problema se ha creado una Interface con la definición del método para guardar y que implementarán los diferentes tipos de documento, finalmente se ha añadido como parámetro al método principal la Interface. Con estos cambios podemos ver que ahora no dependemos del tipo de oferta, sino que hemos invertido la dependencia, será el método que llame a este el responsable de indicar el tipo de documento.

Referencias

Pressman, R. (2010). *Ingeniería de Software. Un enfoque práctico* (7th ed.). Ciudad de México: Mc Graw Hill Educación.

Bovet Derpich, J. (2015). Principios diseño del software. Retrieved from https://www.slideshare.net/josebovet/idss5501-principios-diseno-del-software

Gala, F. (2014). Los principios de diseño del software. Retrieved from https://rootear.com/desarrollo/principios-del-software

García Peñalvo, F., Conde González, M., & Bravo Martín, S. (2018). Principios del diseño del software. Retrieved from http://ocw.usal.es/ensenanzas-tecnicas/ingenieria-del-software/contenidos/Tema5-Principiosdeldisenodelsoftware-1pp.pdf

Principios básicos del diseño de software. (2016). Retrieved from https://mvpcluster.com/diseno-de-software-2/

Haoyu, Wang & Haili, Zhou. (2012). *Basic Design Principles in Software Engineering*, presented at Fourth International Conference on Computational and Information Sciences. Beijing, PRC: Beijing University of Posts and Telecommunications. Retrieved from https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6301346