INSTITUTO POLITECNICO NACIONAL

Segundo Parcial. Principios de Diseño Orientado a Objetos

Análisis y Diseño OO

12-Nov-21

Grupo:2CV13

Emilio Flores Castañeda

Diaz Rosales Mauricio Yael

Guillermo Cardona Gómez

Diaz Gayosso Dafny Saúl

David Emanuel Meza Benitez

1.- ¿Qué es diseño pobre?

El diseño pobre se refiere a la incapacidad de poder desarrollare software que tenga una buena estructura esto conlleva que sea ágil, flexible, mantenible y reusable.

Es importante que para que un equipo pueda desarrollar un buen software que siempre mantengan una vista amplia y completa del software y que esta evolucione con cada iteración tomando en cuenta los requerimientos del ahora y no enfocarse en cosas que aun no se pueden realizar.

Síntomas del diseño pobre

Los síntomas del diseño pobre son los siguientes:

Rigidez: El diseño es difícil de cambiar

Fragilidad: El diseño es fácil de romper (susceptible a errores).

Inmovilidad: Es difícil de reutilizar en otra aplicación porque no se puede desenredar de la aplicación actual.

Viscosidad: Es difícil hacer lo correcto

Complejidad Innecesaria: Sobre complicado

Complejidad de repetición: Abuso de mouse

Opacidad: Expresiones desorganizadas

Estos síntomas son presentados por el diseño a alto nivel y no en el código como tal y se notan en la estructura general de el software no solo en un pedazo.

Podemos considerar que tenemos un diseño pobre cuando cumplimos con múltiples de los síntomas mencionados anteriormente

Conclusión de Diseño Pobre

Consideramos que es muy importante tener en cuenta estos síntomas para poder detectar cuando empecemos a desviarnos de un buen desarrollo y comencemos a caer en varios síntomas de el diseño pobre y poder reconocer en que partes estamos fallando y como.

Bad Design Smell ó Bad Smell the odors of Rotting software

El bad design smell son aquellas características que un software en mal estado empieza a exhibir y estos suelen ser indicadores de que es necesario una reevaluación del software las características de un software en mal estado son los mismos de los síntomas de un diseño pobre

¿Que estimula el empoderamiento del software?

El software se empieza a degradar por que los requerimientos de diseño cambian y se deforman A menudo estos cambios ocurren de manera rápida e imprevista.

Un buen equipo no permitirá que esto suceda y siempre estará al pendiente de los cambios en el desarrollo de software y como abordarlos para poder prevenir que el software se pudra los desarrolladores deben estar muy al pendiente de las señales “olores de pudredumbre” para poder identificarlo y atacarlos inmediatamente

SOLID

¿Qué Es SOLID?

Son las siglas en inglés de los 5 principios del Diseño Orientado a objetos

1. Single Responsability
2. Open-Closed
3. Liskov Substitution
4. Interface Segregation
5. Dependency Invertion

Se toman las iniciales para que se haga más facil recordar cada uno de los principios que se manejan

El principio de diseño que decidimos tomar para esta investigacion es el principio de inversión de dependencia, o “Dependency Invertion”

Cuando diseñamos aplicaciones de software podemos considerar las clases de bajo nivel las clases que implementan operaciones básicas y primarias (acceso a disco, protocolos de red, ...) y las clases de alto nivel las clases que encapsulan lógica compleja (flujos de negocios, ...). Los últimos se basan en las clases de bajo nivel. Una forma natural de implementar tales estructuras sería escribir clases de bajo nivel y una vez que las tengamos escribir las complejas clases de alto nivel. Dado que las clases de alto nivel se definen en términos de otras, esta parece la forma lógica de hacerlo.

Pero este no es un diseño flexible. ¿Qué sucede si necesitamos reemplazar una clase de bajo nivel? Tomemos el ejemplo clásico de un módulo de copia que lee caracteres del teclado y los escribe en el dispositivo de impresión. La clase de alto nivel que contiene la lógica es la clase Copiar. Las clases de bajo nivel son KeyboardReader e PrinterWriter.

En un mal diseño, la clase de alto nivel se usa directamente y depende en gran medida de las clases de bajo nivel. En tal caso, si queremos cambiar el diseño para dirigir la salida a una nueva clase FileWriter, tenemos que hacer cambios en la clase Copiar. (Supongamos que es una clase muy compleja, con mucha lógica y realmente difícil de probar). Para evitar tales problemas, podemos introducir una capa de abstracción entre clases de alto nivel y clases de bajo nivel. Dado que los módulos de alto nivel

contienen la lógica compleja, no deben depender de los módulos de bajo nivel, por lo que la nueva capa de abstracción no debe crearse basándose en módulos de bajo nivel.

Los módulos de bajo nivel se crearán basándose en la capa de abstracción. De acuerdo con este principio, la forma de diseñar una estructura de clases es comenzar desde módulos de alto nivel hasta módulos de bajo nivel: Clases de alto nivel -> Capa de abstracción -> Clases de bajo nivel

Finalmente, con este ejemplo podemos decir que no está bien que los módulos de alto nivel dependan de los de bajo nivel, intuitivamente pensamos que la lógica superior se debe de adaptar a la lógica anterior, pero esto nos puede limitar de una manera muy notable, así que la solución es invertir esta dependencia, si la dependencia era así: Modulo Alto------Depende----->Modulo bajo
Ahora será: Modulo Bajo------Depende----->Modulo Alto

Así cualquier funcionalidad nueva se puede añadir y el cambio en el módulo alto será mínimo

Después de todo este razonamiento el enunciado con el que se generaliza todo este proceso y con el cual se define este principio es:

"***Los módulos de alto nivel no deben depender de los módulos de bajo nivel, ambos deben depender de sus abstracciones"*** y "***Las abstracciones no deberían depender de los detalles, los detalles deberían depender de las abstracciones"***

Aquí tenemos un ejemplo de como no se aplica este principio (Código en C#)

```
using System;
namespace Ejemplo
{
    public class Trabajador{
        public void trabajar(){
            //Trabaja
        }
    }
    public class Gerente{
        Trabajador trabajador {get;set;}

        public void hacerTrabajar(){
            trabajador.trabajar();
        }
    }
}
```

Parece que en un inicio si respeta el principio de no dependencia a módulos bajos, podemos ver que la clase gerente es el módulo de alto nivel, y podemos ver que la clase trabajador es el de bajo nivel, así que por el momento la lógica de la clase Gerente no presenta ningún problema si el trabajador cambia su trabajo

Pero qué pasaría si nosotros queremos añadir un trabajador nuevo

```
public class Trabajador{
        public void trabajar(){
            //Trabaja
        }
    }
    public class TrabajadorSistemas{
        public void trabajar(){
            //Trabaja
        }
    }
    public class Gerente{
        Trabajador trabajador {get;set;}
        TrabajadorSistemas ts{get;set;}
        public void hacerTrabajar(){
            trabajador.trabajar();
        }
        public void hacerTrabajarSistemas(){
            ts.trabajar();
        }
    }
```

Aquí ya estamos violando el principio de inversión de dependencia, porque el gerente está dependiendo de que haya más y más trabajadores, lo cual no debería de ser así, es poco eficiente añadir un método por cada clase, así que invertiremos esta dependencia con el uso de interfaces como capa de abstracción

A continuación, se muestra el código que admite el principio de inversión de dependencia. En este nuevo diseño, se agrega una nueva capa de abstracción a través de la interfaz Trabajador. Ahora los problemas del código anterior están resueltos (considerando que no hay cambios en la lógica de alto nivel): La clase de administrador no requiere cambios al agregar otros Trabajadores. Riesgo minimizado de afectar la funcionalidad antigua presente en la clase Manager ya que no la cambiamos. No es necesario rehacer las pruebas unitarias para la clase Gerente

```
public class Trabajador:ITrabajador{
        public void trabajar(){
            //Trabaja
        }
    }
    public class TrabajadorSistemas:ITrabajador{
        public void trabajar(){
            //Trabaja
        }
    }
    public interface ITrabajador{
        public abstract void trabajar();
    }
    public class Gerente{
        ITrabajador trabajador {get; set;}
        public void hacerTrabajar(){
            trabajador.trabajar();
        }
    }
```

Por supuesto, usar este principio implica un mayor esfuerzo, dará como resultado más clases e interfaces a mantener, en pocas palabras en un código más complejo, pero más flexible. Este principio

no debe aplicarse a ciegas para todas las clases o todos los módulos. Si tenemos una funcionalidad de clase que es más probable que permanezca sin cambios en el futuro, no es necesario aplicar este principio.

Conclusiones

Según la información recabada en clases, las plataformas de moodle, teams y otras fuentes, el diseño pobre, en cuanto a desarrollo de software compete, es una problemática que involucra a quienes están detrás de los proyectos, esto se relaciona con la falta de protocolos y modelados íntegros previos a la creación y despliegue de los diferentes proyectos, aplicaciones y, en general, soluciones. Se recomienda tener o, al menos, buscar bases sólidas en diseño que, hoy en día, tenemos al alcance de una búsqueda para toparnos con artículos, ejemplos, tutoriales y guías de cualquier nivel para mejorar y hacer más eficaz el desarrollo de software

Diagramas de objetos y clases realizados en clase.