Ministerio de Educación Superior, Ciencia y Tecnología (Mescyt)

Centro de Tecnología y Educación Permanente TEP

Pontificia Universidad Católica Madre y Maestra (PUCMM)

Diplomado en Programación en Lenguaje JAVA

Facilitador: Ing. Eudris Cabrera

Diseño de Clases y Buenas Prácticas

¿Cómo sabemos si nuestro diseño es correcto?

Existen algunos síntomas que nos indican que el diseño de un sistema es bastante pobre:

RIGIDEZ (las clases son difíciles de cambiar)

FRAGILIDAD (es fácil que las clases dejen de funcionar)

INMOVILIDAD (las clases son difíciles de reutilizar)

VISCOSIDAD (resulta difícil usar las clases correctamente)

COMPLEJIDAD INNECESARIA (sistema “sobrediseñado”)

REPETICIÓN INNECESARIA (abuso de “copiar y pegar”)

OPACIDAD (aparente desorganización)

Afortunadamente, también existen algunos principios heurísticos que nos ayudan a eliminar los síntomas arriba enumerados:

- Principio de responsabilidad única **(Single-responsiblity principle)**
- Principio abierto-cerrado **(Open-closed principle)**
- Principio de sustitución de Liskov **(Liskov substitution principle)**
- Principio de segregación de interfaces **(Interface segregation principle)**
- Principio de inversión de dependencias **(Dependency Inversion Principle)**

Descripción de los síntomas de un diseño “mejorable”

Rigidez:

El sistema es difícil de cambiar porque cualquier cambio, por simple que sea, fuerza otros muchos cambios en cascada.

¿Por qué es un problema?. “Es más complicado de lo que pensé”.

Cuando nos dicen que realicemos un cambio, puede que nos encontremos con que hay que hacer más cosas de las que, en principio, habíamos pensado que harían falta.

Cuantos más módulos haya que tocar para hacer un cambio, más rígido es el sistema.

Fragilidad:

Los cambios hacen que el sistema deje de funcionar (incluso en lugares que, aparentemente, no tienen nada que ver con lo que hemos tocado).

¿Por qué es un problema?. Porque la solución de un problema genera nuevos problemas...

Conforme la fragilidad de un sistema aumenta, la probabilidad de que un cambio ocasione nuevos quebraderos de cabeza también aumenta.

Inmovilidad:
La reutilización de componentes requiere demasiado esfuerzo.

¿Por qué es un problema?. Porque la reutilización es siempre la solución más rápida.

Aunque un diseño incorpore elementos potencialmente útiles, su inmovilidad hace que sea demasiado difícil extraerlos.

Viscosidad:

Es más difícil utilizar correctamente lo que ya hay implementado que hacerlo de forma incorrecta (e, incluso, que reimplementarlo).

¿Por qué es un problema?.
Cuando hay varias formas de hacer algo, no siempre se elige la mejor forma de hacerlo y el diseño tiende a degenerarse.

Complejidad innecesaria:
El diseño contiene infraestructura que no proporciona beneficios.

¿Por qué es un problema?.
A veces se anticipan cambios que luego no se producen, lo que conduce a más código (que hay que depurar y mantener).

La complejidad innecesaria hace que el software sea más difícil de entender.

Repetición innecesaria:

"Copiar y pegar" resulta perjudicial a la larga.

¿Por qué es un problema?.

Porque el mantenimiento puede convertirse en una pesadilla. El código duplicado debería unificarse bajo una única abstracción.

Opacidad:

Código enrevesado difícil de entender.

¿Por qué es un problema?.
Porque la "entropía" del código tiende a aumentar si no se toman las medidas oportunas.

Escribimos código para que otros puedan leerlo (y entenderlo).

El principio de responsabilidad única (Single-responsiblity principle)
Una clase debe tener un único motivo para cambiar.

En este contexto, una responsabilidad equivale a "una razón para cambiar".

Si se puede pensar en más de un motivo por el que cambiar una clase,entonces la clase tiene más de una responsabilidad.

El principio abierto-cerrado (Open-closed principle)
Los módulos deben ser a la vez abiertos y cerrados: abiertos para ser extendidos y cerrados para ser usados.

1. Debe ser posible extender un módulo para ampliar su conjunto de operaciones y añadir nuevos atributos a sus estructuras de datos.

2. Un módulo ha de tener un interfaz estable y bien definido (que no se modificará para no afectar a otros módulos que dependen de él).

Si se aplica correctamente este principio, los cambios en un sistema se realizan siempre añadiendo código, sin tener que modificar la parte del código que ya funciona.

El principio abierto-cerrado es clave en el diseño orientado a objetos.

Si nos ajustamos a él, obtenemos los mayores beneficios que se suelen atribuir a la orientación a objetos (flexibilidad, mantenibilidad, reutilización).

No obstante, pese a que la abstracción y el polimorfismo lo hacen posible, el principio abierto-cerrado no se garantiza simplemente con utilizar un lenguaje de programación orientado a objetos.

De la misma forma, tampoco tenemos que crear abstracciones arbitrariamente:

No todo tiene que ser abstracto:
El código debe acabar haciendo algo concreto. Las abstracciones se han de utilizar en las zonas que exhiban cierta propensión a sufrir cambios.

El principio de sustitución de Liskov (Liskov substitution principle)
Los tipos base siempre se pueden sustituir por sus subtipos. Si S es un subtipo de T, cualquier programa definido en función de T debe comportarse de la misma forma con objetos de tipo S.

El principio de inversión de dependencias (Dependency Inversion Principle)
a) Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones.

b) Las abstracciones no deben depender de detalles. Los detalles deben depender de las abstracciones.

La programación estructurada tiende a crear estructuras jerárquicas en las que el comportamiento de los módulos de alto nivel (p.ej. el programa principal) depende de detalles de módulos de un nivel inferior (p.ej. la opción del menú seleccionada por el usuario).

La estructura de dependencias en una aplicación orientada a objetos bien diseñada suele ser la inversa.

Si se mantuviese la estructura tradicional:

ß Cuando se cambiasen los módulos de nivel inferior, habría que modificar los módulos de nivel superior.

ß Además, los módulos de nivel superior resultarían difíciles de reutilizar si dependiesen de módulos inferiores.

Cuestiones de estilo

Escribimos código para que lo puedan leer otras personas, no sólo para que lo traduzca el compilador.

Convenciones get y set

Muchas clases suelen incluir métodos públicos que permiten acceder y modificar las variables de instancia desde el exterior de la clase.

Por convención, los métodos se denominan:

- getX para obtener la variable de instancia X, y
- setX para establecer el valor de la variable de instancia X.

No es obligatorio, pero resulta conveniente, especialmente si queremos desarrollar componentes

reutilizables (denominados JavaBeans) y facilitar su uso posterior.

Identificadores

Los identificadores deben ser descriptivos

p, i, s...

precio, izquierda, suma...

En ocasiones, se permite el uso de nombres cortos para variables locales cuyo significado es evidente (p.ej. bucles controlados por contador)

```
for (elemento=0; elemento<N; elemento++ )...

for (i=0; i<N; i++) ...
```

Constantes

Se considera una mala costumbre incluir literales de tipo numérico (“números mágicos”) en medio del código. Se prefiere la definición de constantes simbólicas (con final).

```
for (i=0; i<79; i++) ...

for (i=0; i<columnas-1; i++) ...
```

Expresiones

Expresiones booleanas:

Es aconsejable escribirlas como se dirían en voz alta.

```
if ( !(bloque<actual) ) ...

 if ( bloque >= actual ) ...
```

Expresiones complejas:

Es aconsejable dividirlas para mejorar su legibilidad

```
  x += ( xp = ( 2*k<(n-m) ? c+k: d-k ));

       if ( 2*k < n-m ){

           xp = c+k;

      } else {

        xp = d-k;

       x += xp;
}
```

Comentarios

Comentarios descriptivos:

Los comentarios deben comunicar algo. Jamás se utilizarán para "parafrasear" el código y repetir lo que es obvio.

```
ß    i++;    /* Incrementa el contador */

     /* Recorrido secuencial de los datos*/

     for (i=0; i<N; i++) ...
```

Estructuras de control Sangrías:

Conviene utilizar espacios en blanco o separadores para delimitar el ámbito de las estructuras de control de nuestros programas.

Líneas en blanco:

Para delimitar claramente los distintos bloques de código en nuestros programas dejaremos líneas en blanco entre ellos. Salvo en la cabecera de los bucles for, sólo incluiremos una sentencia por línea de código.

Sean cuales sean las convenciones utilizadas al escribir código (p.ej. uso de sangrías y llaves), hay que ser consistente en su utilización.

```
while (...) {
   ...
}

for (...;...;...) {
   ...
}

if (...) {
     ...
}
```

```
while (...)
{
   ...
}
for (...;...;...)
{
   ...
}
if (...)
{
   ...
 }
```

El código bien escrito es más fácil de leer, entender y mantener (además, seguramente tiene menos errores)

Eudris Cabrera Rodriguez

Ingeniero Telemático

Consultor / Desarrollador Informático

LinkedIn: http://www.linkedin.com/in/eudriscabrera

Mayo 2016, Santiago de los Caballeros, R. D.