El Principio Abierto/Cerrado

Francisco José García Peñalvo
*Licenciado en Informática. Profesor del Área de Lenguajes y Sistemas Informáticos de la **Universidad de Burgos**.*
fgarcia@.ubu.es

Carlos Pardo Aguilar
*Licenciado en Informática. Profesor del Área de Lenguajes y Sistemas Informáticos de la **Universidad de Burgos***
cpardo@.ubu.es

El principio abierto/cerrado es, sin duda alguna, una de las referencias más importantes para realizar un buen Diseño Orientado a Objetos, de forma que las clases diseñadas para un determinado software puedan ser extendidas en otros desarrollos, pero sin modificar el código fuente de dichas clases, aprovechando de esta forma las facilidades de extensión y reutilización que ofrece la orientación a objeto

En la introducción que en [1] se hizo sobre el Diseño Orientado a Objetos (DOO), se presentaba éste como el proceso de modelado del dominio de la solución, teniendo como principal responsabilidad el refinamiento de las clases semánticas identificadas en la fase de Análisis Orientado a Objetos (AOO). Estos conceptos están perfectamente aceptados y asimilados por la comunidad de la orientación a objeto, aunque a las personas que estén dando sus primeros pasos en la orientación a objeto seguramente les parecerán bastantes abstractos y poco clarificadores.

Para aclarar este tipo de aseveraciones podríamos recurrir a generar listas de recomendaciones a seguir en el DOO (*aumentar cohesión y disminuir el acoplamiento entre las clases, creación del mayor número de clases abstractas posibles en las jerarquías de herencia, mantenimiento de las signaturas de los métodos consistentes…*). Pero, a nuestro modesto entender, este tipo de *recetarios* queda bastante inconexo, no dejando claros los motivos por los que deben aplicarse las heurísticas de diseño que recogen. Por este motivo, desde estas páginas vamos a centrarnos en aspectos concretos del DOO, explicándolos en detalle e ilustrando su aplicación práctica mediante ejemplos sencillos, obteniéndose como conclusiones de los mismos las heurísticas de diseño que se recogen en los diferentes métodos de DOO.

Como inicio de esta serie de artículos dedicados al DOO vamos a centrarnos en el que puede ser el principio elemental del DOO, el denominado principio abierto/cerrado.

El principio abierto/cerrado

Que un sistema software cambia a lo largo de su ciclo de vida, es algo que se tiene que tener presente cuando se desarrolla un sistema, si no se quiere que el ciclo de vida de dicho sistema se reduzca a la primera y única versión.

La forma de realizar diseños software, para que permanezcan estables ante los cambios, es el tema que aborda el principio abierto/cerrado que fue enunciado por el genial y siempre controvertido y polémico Bertrand Meyer [2], y que dice lo siguiente: "***Las entidades software (clases, módulos, funciones…) deben estar abiertas para su extensión, pero cerradas para su modificación***".

El principio abierto/cerrado expresa la necesidad de que ante un cambio de los requisitos, el diseño de las entidades existentes permanezca inalterado, recurriéndose a la extensión del comportamiento de dichas entidades añadiendo nuevo código, pero nunca cambiando el código ya existente.

La naturaleza, simultáneamente abierta y cerrada de los sistemas orientados al objeto da soporte a la facilidad de mantenimiento, de reutilización y de verificación. Así, un sistema reutilizable debe estar abierto, en el sentido de que tiene que ser fácil de extender, y cerrado en el sentido de que debe estar listo para ser utilizado sin tocar el código existente.

Cuando un único cambio en un programa produce una cascada de cambios en los módulos dependientes, el programa exhibe unas características no deseables debidos a un mal diseño. De esta forma, el programa se convierte en un programa frágil, rígido, impredecible y en consecuencia **No Reutilizable**.

Las entidades software que conforman el principio abierto/cerrado cumplen dos propiedades primarias:

1. **Están abiertas para su extensión**

   Esto implica que el comportamiento de estas entidades software puede ser extendido. Se puede hacer que una entidad se comporte de otras formas cuando los requisitos de la aplicación cambien.

2. **Están cerradas para su modificación**

   El código fuente de la entidad software es inalterable. No se permite realizar cambios en el código fuente existente.

La clave de este principio está en la abstracción. Se puede definir abstracción como la representación de las características esenciales de algo sin incluir antecedentes o detalles irrelevantes [3].

Se pueden crear abstracciones que sean fijas y que representen un grupo ilimitado de posibles comportamientos. Las abstracciones son un conjunto de clases base, y el grupo ilimitado de los posibles comportamientos viene representado por todas las posibles clases derivadas. De esta forma, un módulo está cerrado para su modificación al depender de una abstracción que es fija. Pero, el comportamiento del módulo puede ser extendido mediante la creación de clases derivadas.

El párrafo anterior encierra la explicación de muchas de las heurísticas propias del DOO. En primer lugar, nos está estableciendo la necesidad de que la base de una jerarquía de clases esté formada por clases abstractas. Recordemos que una clase abstracta es una clase que no representa completamente un objeto, sino que en su lugar representa un amplio rango de diferentes clases de objetos, más concretamente, recoge las características comunes a todas sus clases derivadas, por tanto, una clase abstracta sólo ofrece una descripción parcial de sus objetos. Como consecuencia inmediata de esto se tiene que una clase debe mantener consistente la signatura de los métodos que fueron introducidos por sus predecesoras abstractas en la jerarquía de clases en la que se encuentra inmersa, y cuyo comportamiento va a redefinirse en dicha clase, siendo esto necesario para poder tomar las ventajas que ofrece el polimorfismo en tiempo de ejecución.

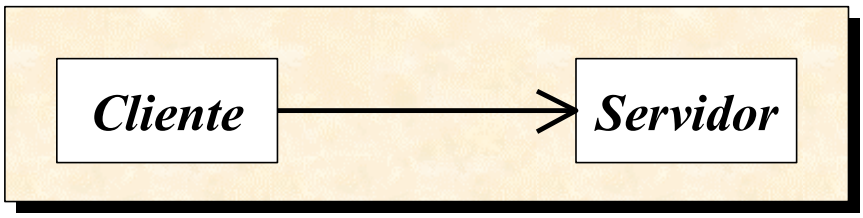

Figura A. Diseño que no se ajusta al principio abierto/cerrado.

En la **Figura A** se tiene el esquema básico de un diseño que no cumple el principio abierto/cerrado. El diseño de esta figura (*utilizando notación UML 1.1 [4]*) muestra dos clases concretas **Cliente** y **Servidor**, de forma que entre ambas clases existe una relación de uso (*la clase **Cliente** usa a la clase **Servidor***). Si por un cambio en los requisitos se desea que los objetos de la clase **Cliente** puedan usar objetos de otros servidores habría que modificar la definición de la clase **Cliente** para referenciar a la nueva clase **Servidor**, lo cual iría contra el principio abierto/cerrado.

Sin embargo, en la **Figura B** se presenta un diseño que sí conforma el principio abierto/cerrado. En este caso, la clase **ServidorAbstracto** es una clase abstracta que define la interfaz de sus servicios pero no su implementación (*hablando en termiología C++, se tiene una clase que contiene métodos virtuales puros*). La clase **Cliente** usa esta abstracción, y por tanto los objetos de la clase **Cliente** utilizarán los objetos de las clases derivadas de la clase **ServidorAbstracto**. Si se desea que los objetos de la clase **Cliente** utilicen diferentes clases **Servidor**, se deriva una nueva clase de la clase **ServidorAbstracto**, y la clase **Cliente** permanece inalterada.

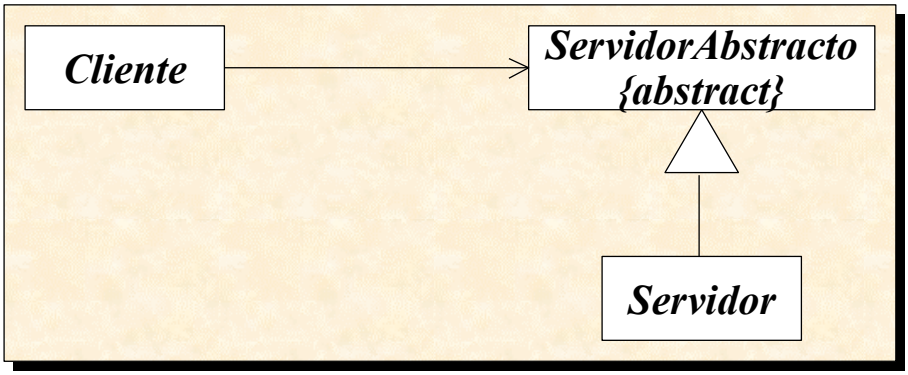

Figura B. Diseño que cumple el principio abierto/cerrado.

Ejemplo del principio abierto/cerrado

Para ilustrar el uso práctico del principio abierto/cerrado se va a recurrir a un ejemplo clásico, la abstracción de la figura, que debido a su sencillez lo hacen ideal para los fines didácticos que se persiguen en el presente artículo.

La idea es realizar una aplicación que sea capaz de dibujar círculos y cuadrados. Los círculos y los cuadrados deben ser dibujados en un determinado orden, para lo cual se creará una cola de figuras. Un determinado cliente recorrerá dicha lista y dibujará las figuras según el tipo de figura.

Se van a realizar dos implementaciones, la primera de ellas se corresponderá con una solución procedural, donde no se va a cumplir el principio abierto/cerrado (**Listado 1**), mientras que la segunda implementación conforma el diseño presentado en la **Figura B (Listado 2**), y por tanto cumplirá el principio abierto/cerrado.

La primera implementación está realizada en lenguaje **C** y, como ya se indicó, no se ajusta al principio abierto/cerrado. Se puede observar que tanto la estructura **Cuadrado** como la estructura **Circulo** tienen el primer elemento en común, que no es más que un identificador del tipo de figura. La función **DibujaFiguras** recorre la matriz de punteros a elementos de tipo **Figura**, examinando el tipo de figura y llamando explícitamente a la función apropiada de dibujo para cada tipo de figura.

Por tanto, la función **DibujaFiguras** del **Listado 1** no cumple el principio abierto/cerrado porque no está cerrada a nuevos tipos de figuras, esto es, si se quiere extender esta función para dibujar por ejemplo rectángulos, se tiene que modificar la función **DibujaFiguras**, añadiéndose un nuevo caso al *switch* para contemplar los rectángulos, necesitándose en general un nuevo caso para cada nuevo tipo de figura que se quiera añadir.

En el **Listado 2** se presenta una solución orientada a objetos, implementada en **C++,** que cumple el principio abierto/cerrado. En este caso, se ha definido la clase abstracta **Figura**, la cual cuenta con un método virtual puro denominado **Dibuja**. Las clases **Circulo** y **Cuadrado** son clases concretas, derivadas de la clase **Figura**. Ambas clases redefinen el método **Dibuja** con la implementación adecuada a cada clase, pero manteniendo intacta la signatura del método.

La función **DibujaFiguras** del **Listado 2** no necesita modificarse en el caso de que se añadan nuevas figuras a la jerarquía, porque está función invoca siempre al servicio **Dibuja**, determinándose en tiempo de ejecución cual es la implementación que le corresponde, dependiendo del tipo de la instancia apuntada desde la posición i-ésima de la lista de figuras.

De acuerdo con esto se puede extender el comportamiento de la función **DibujaFiguras** sin modificar nada en absoluto de su código, cumpliendo así el principio abierto/cerrado.

Conclusiones

Hemos presentado uno de los principios básicos del DOO, que todo diseñador debe tener presente para buscar la reutilización de sus diseños y en consecuencia del código fuente asociado a estos diseños.

Pero, como prometíamos al principio de este artículo, vamos a detallar las implicaciones que conlleva crear diseños que conformen el principio abierto/cerrado.

- En primer lugar citar la importancia de la abstracción dentro de una jerarquía de herencia. Esto implica que al menos la clase base de una jerarquía de herencia debe ser abstracta, y cuantos más niveles de dicha jerarquía tengan clases abstractas, más se potenciará a la reutilización de dicha jerarquía de clases.
- Todas las clases concretas deben mantener las signaturas de los métodos introducidos por las clases abstractas, de forma que se potencie el uso del polimorfismo en tiempo de ejecución. Esta implicación está directamente relacionada con otro principio básico del DOO, el principio de sustitución de Liskov, que será objeto de un próximo artículo.
- Todos los atributos de una clase deben estar ocultos al exterior de la clase. Esta es una de las convenciones del DOO más difundidas. Los atributos de una clase deben ser manejados exclusivamente por los métodos de la propia clase. Este uso del encapsulamiento tan propio de la orientación a objeto, es una de las formas más adecuadas para lograr una fuerte cohesión dentro de la clase y reducir el acoplamiento entre clases, algo fundamental en cualquier diseño.

- No deben utilizarse variables globales. De nuevo nos encontramos con otra de las heurísticas de diseño más difundidas con independencia del paradigma de programación en el que se trabaje. Esta aseveración es similar a la de tener atributos públicos. Ningún módulo que dependa de una variable global puede estar cerrado con respecto a otro módulo que modifique el valor de dicha variable, existiendo un acoplamiento entre estos módulos.

Como comentario final al principio abierto/cerrado, se puede decir que nunca se tiene un programa completamente cerrado. En general, no importa lo cerrado que esté una entidad software, siempre existe algún tipo de cambio que demuestra que no estaba cerrada. Entonces, dado que una entidad software nunca estará cerrada al 100%, el cierre debe ser estratégico, el diseñador debe seleccionar los cambios para los cuales estará cerrado su diseño.

```
#include <stdio.h>
#include <stdlib.h>
#define MAXIMO 10
#define ELEMENTOS 5

enum TipoFigura {circulo, cuadrado};
struct Figura {
        TipoFigura Tipo;
};
struct Punto {
        float x, y;
};
struct Circulo {
        TipoFigura Tipo;
        float Radio;
        struct Punto Centro;
};
struct Cuadrado {
        TipoFigura Tipo;
        float Lado;
        struct Punto SuperiorIzquierda;
};
typedef struct Figura *PunteroFigura;
void DibujaCuadrado(struct Cuadrado *);
void DibujaCirculo(struct Circulo*);
void DibujaFiguras(PunteroFigura lista[], int);

void DibujaCuadrado(struct Cuadrado *c) {
        printf("\nCuadrado");
        printf("\nPunto superior izquierda: %f %f", c->SuperiorIzquierda.x,
                        c->SuperiorIzquierda.y);
        printf("\nLongitud del lado: %f", c->Lado);
}
void DibujaCirculo(struct Circulo *c) {
        printf("\nCírculo");
        printf("\nCentro: %f %f", c->Centro.x, c->Centro.y);
        printf("\nLongitud del radio: %f", c->Radio);
}
void DibujaFiguras(PunteroFigura lista[], int n) {
        int i;
        struct Figura *f;
        for (i=0; i<n; i++) {
                f = lista[i];
                switch (f->Tipo) {
                case cuadrado:
                        DibujaCuadrado((struct Cuadrado*)f);
                break;
                case circulo:
                        DibujaCirculo((struct Circulo*)f);
                break;
                }
        }
}
void main(void) {
        struct Cuadrado *c1;
        struct Circulo *c2;
        int i;
        PunteroFigura lista[MAXIMO];

        for (i=0; i<ELEMENTOS; i++) {
                if (i%2==0) {
                        c1 = (struct Cuadrado *) malloc (sizeof(struct Cuadrado));
                        c1->Tipo = cuadrado;
                        c1->SuperiorIzquierda.x = c1->SuperiorIzquierda.y = i*2;
                        c1->Lado = i*2.5;
                        lista[i]= (struct Figura *)c1;
                }
                else {
                        c2 = (struct Circulo *) malloc (sizeof(struct Circulo));
                        c2->Tipo = circulo;
                        c2->Centro.x = c2->Centro.y = 5.0+i;
                        c2->Radio = 10.0/i;
                        lista[i]= (struct Figura *)c2;
                }
        }
        DibujaFiguras(lista, ELEMENTOS);
}
```

Listado 1. No cumple el principio abierto/cerrado (figuras.c)

```
#include <iostream.h>
#define MAXIMO 10
#define ELEMENTOS 5

class Figura {
public:
        virtual void Dibuja() = 0;
};
class Punto {
        float x, y;
public:
        Punto(float x1, float y1) {x=x1; y=y1;}
        float LeerX(void) { return x;}
        float LeerY(void) { return y;}
};
class Circulo: public Figura {
        float Radio;
        Punto Centro;
public:
        Circulo(float x1, float y1, float r1):Centro(x1,y1) { Radio = r1;}
        virtual void Dibuja() {
                cout << "\nCírculo";
                cout << "\nCentro: " << Centro.LeerX() << " " << Centro.LeerY();
                cout << "\nLongitud del radio:" << Radio;
        }
};
class Cuadrado:public Figura{
        float Lado;
        Punto SuperiorIzquierda;
public:
        Cuadrado(float x1,float y1,float l1):SuperiorIzquierda(x1, y1) { Lado = l1;}
        virtual void Dibuja()  {
                cout << "\nCuadrado";
                cout << "\nPunto superior izquierda: " <<SuperiorIzquierda.LeerX() << " "<<
                        SuperiorIzquierda.LeerY();
                cout << "\nLongitud del lado: " << Lado;
        }
};
void DibujaFiguras(Figura *lista[], int);

void DibujaFiguras(Figura *lista[], int n) {for (int i=0; i<n; lista[i++]->Dibuja());}

void main(void) {
        Figura *lista[MAXIMO];
        for (char i=0; i<ELEMENTOS; i++) {
                if (i%2==0)
                        lista[i] = new Cuadrado (i*2.0, i*2.0, i+5.0);
                else
                        lista[i] = new Circulo (i*3.0, i*2.5, i + 10.0);
        }
        DibujaFiguras(lista, ELEMENTOS);
}
```

Listado 2. Código que si cumple el principio abierto/cerrado (figura.cpp).

Bibliografía

[1] García Peñalvo, Francisco José y Pardo Aguilar, Carlos. "*Introducción al Análisis y Diseño Orientado a Objetos*". RPP, Nº37, pp. 64-70. Editorial América Ibérica. Febrero 1998.

[2] Meyer, Bertrand. "*Object Oriented Software Construction*". 2nd Edition. Prentice-Hall. 1997.

[3] Graham, Ian. "*Métodos Orientado a Objetos*". 2ª Ed. Addison-Wesley/Díaz de Santos, 1996.

[4] García Peñalvo. Francisco José y Pardo Aguilar, Carlos. "*Diagramas de Clase en UML 1.1*". RPP, Nº38. Editorial América Ibérica. Marzo 1998.