**OCP: El Principio Abierto-Cerrado**

Este es el primero de mis artículos en el Cuaderno de Ingeniería para The C++ Report. Los artículos que aparecerán en esta columna se centrarán en el uso de C++ y OOD, y abordarán problemas de ingeniería de software. Me esforzaré por crear artículos pragmáticos y directamente útiles para el ingeniero de software en la trinchera. En estos artículos, utilizaré la notación de Booch para documentar diseños orientados a objetos. La barra lateral proporciona un breve léxico de la notación de Booch.

Hay muchas heurísticas asociadas con el diseño orientado a objetos. Por ejemplo, "todas las variables miembro deben ser privadas" o "se deben evitar las variables globales" o "usar la identificación de tipo en tiempo de ejecución (RTTI) es peligroso". ¿Cuál es la fuente de estas heurísticas? ¿Qué las hace verdaderas? ¿Siempre son ciertas? Esta columna investiga el principio de diseño que subyace a estas heurísticas: el principio abierto-cerrado.

Como dijo Ivar Jacobson: "Todos los sistemas cambian durante sus ciclos de vida. Esto debe tenerse en cuenta al desarrollar sistemas que se espera que duren más que la primera versión". ¿Cómo podemos crear diseños que sean estables ante el cambio y que duren más que la primera versión? Bertrand Meyer nos dio orientación ya en 1988 cuando acuñó el ahora famoso principio abierto-cerrado. Parafraseándolo:

ENTIDADES DE SOFTWARE (CLASES, MÓDULOS, FUNCIONES, ETC.) DEBERÍAN ESTAR ABIERTAS PARA LA EXTENSIÓN, PERO CERRADAS PARA LA MODIFICACIÓN.

Cuando un solo cambio en un programa resulta en una cascada de cambios en módulos dependientes, ese programa muestra atributos indeseables que asociamos con un diseño deficiente. El programa se vuelve frágil, rígido, impredecible e irremplazable. El principio abierto-cerrado aborda esto de una manera muy directa. Dice que debes diseñar módulos que nunca cambien. Cuando cambian los requisitos, amplías el comportamiento de esos módulos agregando nuevo código, no cambiando código antiguo que ya funciona.

Descripción 2

1. Son "Abiertos para la Extensión". Esto significa que el comportamiento del módulo puede ser ampliado. Podemos hacer que el módulo se comporte de nuevas y diferentes maneras a medida que cambien los requisitos de la aplicación, o para satisfacer las necesidades de nuevas aplicaciones.
2. Son "Cerrados para la Modificación". El código fuente de dicho módulo es inviolable. A nadie se le permite realizar cambios en el código fuente. Parecería que estos dos atributos están en desacuerdo entre sí. La forma normal de ampliar el comportamiento de un módulo es hacer cambios en ese módulo. Normalmente se piensa que un módulo que no puede cambiarse tiene un comportamiento fijo. ¿Cómo pueden resolverse estos dos atributos opuestos?

La abstracción es la clave. En C++, utilizando los principios del diseño orientado a objetos, es posible crear abstracciones que son fijas y, al mismo tiempo, representan un grupo ilimitado de comportamientos posibles. Las abstracciones son clases base abstractas, y el grupo ilimitado de comportamientos posibles está representado por todas las posibles clases derivadas. Es posible que un módulo manipule una abstracción. Dicho módulo puede estar cerrado para la modificación ya que depende de una abstracción que está fija. Sin embargo, el comportamiento de ese módulo puede extenderse creando nuevas derivadas de la abstracción.

La Figura 1 muestra un diseño simple que no cumple con el principio abierto-cerrado. Tanto las clases Cliente como Servidor son concretas. No hay garantía de que las funciones miembro de la clase Servidor sean virtuales. La clase Cliente utiliza la clase Servidor. Si deseamos que un objeto Cliente utilice un objeto Servidor diferente, entonces la clase Cliente debe modificarse para nombrar la nueva clase de servidor.

La Figura 2 muestra el diseño correspondiente que cumple con el principio abierto-cerrado. En este caso, la clase AbstractServer es una clase abstracta con funciones miembro puramente virtuales. La clase Cliente utiliza esta abstracción. Sin embargo, los objetos de la clase Cliente utilizarán objetos de la clase derivada Servidor. Si queremos que los objetos Cliente utilicen una clase de servidor diferente, entonces se puede crear una nueva derivada de la clase AbstractServer. La clase Cliente puede permanecer sin cambios.

Figura 1 Cliente Cerrado Cliente Servidor

3: El Principio Abierto-Cerrado

La Abstracción de Forma Considere el siguiente ejemplo. Tenemos una aplicación que debe poder dibujar círculos y cuadrados en una GUI estándar. Los círculos y cuadrados deben dibujarse en un orden particular. Se creará una lista de círculos y cuadrados en el orden apropiado y el programa debe recorrer la lista en ese orden y dibujar cada círculo o cuadrado.

En C, utilizando técnicas procedimentales que no cumplen con el principio abierto-cerrado, podríamos resolver este problema como se muestra en el Listado 1. Aquí vemos un conjunto de estructuras de datos que tienen el mismo primer elemento, pero son diferentes más allá de eso. El primer elemento de cada una es un código de tipo que identifica la estructura de datos como un círculo o un cuadrado. La función DrawAllShapes recorre una matriz de punteros a estas estructuras de datos, examinando el código de tipo y luego llamando a la función apropiada (DrawCircle o DrawSquare).

Figura 2 Cliente Abierto

Listado 1 Solución procedural al problema de cuadrados/círculos

enum TipoForma

{círculo, cuadrado};

struct Forma { TipoForma suTipo; };

struct Círculo { TipoForma suTipo; double suRadio; Punto suCentro; };

Cliente Abstracto Servidor

Servidor A

La Abstracción de Forma 4

La función DrawAllShapes no cumple con el principio abierto-cerrado porque no puede cerrarse contra nuevos tipos de formas. Si quisiera ampliar esta función para poder dibujar una lista de formas que incluya triángulos, tendría que modificar la función. De hecho, tendría que modificar la función para cualquier nuevo tipo de forma que necesitara dibujar.

Por supuesto, este programa es solo un ejemplo simple. En la vida real, la declaración switch en la función DrawAllShapes se repetiría una y otra vez en varias funciones en toda la aplicación, cada una haciendo algo un poco diferente. Agregar una nueva forma a dicha aplicación significa buscar en todos los lugares donde existen tales declaraciones switch (o cadenas if/else) y agregar la nueva forma a cada una. Además, es muy improbable que todas las declaraciones switch y cadenas if/else estén tan bien estructuradas como la de DrawAllShapes. Es mucho más probable que los predicados de las sentencias if se combinen con operadores lógicos, o que las cláusulas case de las sentencias switch se combinen de manera que "simplifiquen" la toma de decisiones local. Por lo tanto, el problema de encontrar y entender todos los lugares donde es necesario agregar la nueva forma puede no ser trivial.

El Listado 2 muestra el código de una solución al problema de cuadrados/círculos que cumple con el principio abierto-cerrado. En este caso, se crea una clase abstracta Forma. Esta clase abstracta tiene una sola función miembro puramente virtual llamada Dibujar. Tanto Círculo como Cuadrado son derivados de la clase Forma.

Obsérvese que si queremos ampliar el comportamiento de la función DrawAllShapes en el Listado 2 para dibujar un nuevo tipo de forma, todo lo que necesitamos hacer es agregar una nueva derivada de la clase Forma. La función DrawAllShapes no necesita cambiar. Así que DrawAllShapes cumple con el principio abierto-cerrado. Su comportamiento se puede ampliar sin modificarlo.

En el mundo real, la clase Forma tendría muchos más métodos. Sin embargo, agregar una nueva forma a la aplicación sigue siendo bastante simple, ya que todo lo que se requiere es crear la nueva derivada e implementar todas sus funciones. No es necesario buscar en toda la aplicación lugares que requieran cambios.

Listado 2 Solución OOD al problema de cuadrados/círculos.

class Forma

{ public: virtual void Dibujar() const = 0; }; class Cuadrado : public Forma {

public: virtual void Dibujar() const;

};

class Círculo : public Forma {

public: virtual void Dibujar() const;

};

void DrawAllShapes(Set<Forma\*>& lista) {

for (Iterador<Forma\*> i(lista); i; i++)

(\*i)->Dibujar();

}

Cierre Estratégico 6 Cierre Estratégico Debe quedar claro que ningún programa significativo puede ser 100% cerrado. Por ejemplo, imagine qué sucedería con la función DrawAllShapes del Listado 2 si decidiéramos que todos los Círculos deben dibujarse antes que cualquier Cuadrado. La función DrawAllShapes no está cerrada contra un cambio como este. En general, no importa cuán "cerrado" sea un módulo, siempre habrá algún tipo de cambio contra el cual no está cerrado.

Dado que el cierre no puede ser completo, debe ser estratégico. Es decir, el diseñador debe elegir los tipos de cambios contra los cuales cerrar su diseño. Esto requiere una cierta cantidad de previsión derivada de la experiencia. El diseñador experimentado conoce lo suficiente a los usuarios y a la industria como para juzgar la probabilidad de diferentes tipos de cambios. Luego se asegura de que se invoque el principio abierto-cerrado para los cambios más probables.

Usar la Abstracción para Obtener un Cierre Explícito. ¿Cómo podríamos cerrar la función DrawAllShapes contra cambios en el orden de dibujo? Recuerde que el cierre se basa en la abstracción. Entonces, para cerrar DrawAllShapes contra el orden, necesitamos algún tipo de "abstracción de orden". El caso específico de ordenar anteriormente tenía que ver con dibujar ciertos tipos de formas antes que otros tipos de formas.

Una política de orden implica que, dado cualquier par de objetos, es posible descubrir cuál debería dibujarse primero. Por lo tanto, podemos definir un método de Forma llamado Precede que toma otra Forma como argumento y devuelve un resultado bool. El resultado es verdadero si el objeto Forma que recibe el mensaje debería ordenarse antes que el objeto Forma pasado como argumento.

En C++, esta función podría representarse mediante una función de operador< sobrecargada. El Listado 3 muestra cómo podría lucir la clase Forma con los métodos de ordenación. Ahora que tenemos una forma de determinar el orden relativo de dos objetos Forma, podemos ordenarlos y luego dibujarlos en orden. El Listado 4 muestra el código C++ que hace esto.

Esto nos da un medio para ordenar objetos Forma y dibujarlos en el orden apropiado. Pero aún no tenemos una abstracción de orden decente. Como está, los objetos individuales de Forma tendrán que anular el método Precede para especificar el orden. ¿Cómo funcionaría esto? ¿Qué tipo de código escribiríamos en Circle::Precede para asegurarnos de que los Círculos se dibujen antes que los Cuadrados? Considere el Listado 5.

Listado 3 Forma con métodos de ordenación. class Forma { public: virtual void Dibujar() const = 0; virtual bool Precede(const Forma&) const = 0; bool operator<(const Forma& s) {return Precede(s);} }; Listado 4 DrawAllShapes con Ordenamiento. void DrawAllShapes(Set<Forma\*>& lista) { // copiar elementos en OrderedSet y luego ordenar. OrderedSet<Forma\*> listaOrdenada = lista; listaOrdenada.Sort(); for (Iterador<Forma\*> i(listaOrdenada); i; i++) (*i)->Dibujar(); } Listado 5 Ordenar un Círculo bool Circle::Precede(const Forma& s) const { if (dynamic\_cast<Cuadrado*>(&s)) return true; else return false; }

Cierre Estratégico 8 Listado 6 Mecanismo de ordenación de tipos basado en tabla

#include <typeinfo.h> #include <string.h>

enum {falso, verdadero};

typedef int bool; class Forma

{

public:

virtual void Dibujar() const = 0;

virtual bool Precede(const Forma&) const;

bool operator<(const Forma& s) const {return Precede(s);

}

private:

static char\* tablaOrdenTipos[];

};

char\* Forma::tablaOrdenTipos[] = { "Círculo", "Cuadrado", 0 }; // Esta función busca en una tabla los nombres de las clases. // La tabla define el orden en el que se // deben dibujar las formas. Las formas que no // se encuentran siempre preceden a las formas que se encuentran. // bool Forma::Precede(const Forma& s) const {

const char\* esteTipo = typeid(*this).name(); const char* tipoArg = typeid(s).name();

bool hecho = falso;

int esteOrd = -1;

int argOrd = -1;

for (int i=0; !hecho; i++) {

const char\* entradaTabla = tablaOrdenTipos[i];

if (entradaTabla != 0) {

if (strcmp(entradaTabla, esteTipo) == 0) esteOrd = i;

if (strcmp(entradaTabla, tipoArg) == 0) argOrd = i; if ((argOrd > 0) && (esteOrd > 0)) hecho = verdadero;

} else // entrada de la tabla == 0 hecho = verdadero;

} return esteOrd < argOrd;

} Listado 6 (Continuación) Mecanismo de ordenación de tipos basado en tabla

El único elemento que no está cerrado con respecto al orden de las diversas formas es la mesa misma. Y esa mesa puede colocarse en su propio módulo, separado de todos los demás módulos, de modo que los cambios en ella no afecten a ninguno de los otros módulos. Ampliando el cierre aún más. Esto no es el final de la historia. Hemos logrado cerrar la jerarquía de formas y la función DrawAllShapes con respecto a un orden que depende del tipo de la forma. Sin embargo, las derivadas de Shape no están cerradas respecto a políticas de ordenación que no tienen nada que ver con los tipos de formas. Es probable que deseemos ordenar el dibujo de formas de acuerdo con alguna estructura de nivel superior. Una exploración completa de estos problemas está fuera del alcance de este artículo; sin embargo, el lector ambicioso podría considerar cómo abordar este problema utilizando una clase OrderedObject abstracta contenida por la clase OrderedShape, que se deriva tanto de Shape como de OrderedObject.

Heurísticas y Convenciones

Como se mencionó al principio de este artículo, el principio abierto-cerrado es la motivación principal detrás de muchas de las heurísticas y convenciones que se han publicado con respecto al diseño orientado a objetos a lo largo de los años. Aquí hay algunas de las más importantes.

Hacer todas las variables miembro privadas.

Esta es una de las convenciones más comúnmente aceptadas en el diseño orientado a objetos. Las variables miembro de las clases solo deben ser conocidas por los métodos de la clase que las define. Las variables miembro nunca deben ser conocidas por ninguna otra clase, incluidas las clases derivadas. Por lo tanto, deben declararse como privadas en lugar de públicas o protegidas.

A la luz del principio abierto-cerrado, la razón de esta convención debería ser clara. Cuando las variables miembro de una clase cambian, cada función que depende de esas variables debe cambiarse. Por lo tanto, ninguna función que dependa de una variable puede estar cerrada con respecto a esa variable.

if ((argOrd > 0) && (thisOrd > 0))

done = true;

}

else // entrada de la tabla == 0

done = true;

}

return thisOrd < argOrd;

}

Conclusión

Hay mucho más que podría decirse sobre el principio abierto-cerrado. En muchos aspectos, este principio está en el corazón del diseño orientado a objetos. La conformidad con este principio es lo que produce los mayores beneficios reclamados para la tecnología orientada a objetos, es decir, la reutilización y la mantenibilidad. Sin embargo, la conformidad con este principio no se logra simplemente mediante el uso de un lenguaje de programación orientado a objetos. Más bien, requiere dedicación por parte del diseñador para aplicar la abstracción a aquellas partes del programa que el diseñador considera que estarán sujetas a cambios.

Este artículo es una versión extremadamente condensada de un capítulo de mi nuevo libro: "Los Principios y Patrones de OOD", que se publicará pronto por Prentice Hall. En artículos posteriores, exploraremos muchos de los otros principios del diseño orientado a objetos. También estudiaremos diversos patrones de diseño, así como sus fortalezas y debilidades en relación con la implementación en C++. Investigaremos el papel de las categorías de clases de Booch en C++ y su aplicabilidad como espacios de nombres en C++. Definiremos qué significa "cohesión" y "acoplamiento" en un diseño orientado a objetos, y desarrollaremos métricas para medir la calidad de un diseño orientado a objetos. Y después de eso, muchos otros temas interesantes.