**LSP: El Principio de Sustitución de Liskov**

Este es el segundo de mis artículos para la columna Engineering Notebook en The C++ Report. Los artículos que aparecerán en esta columna se centrarán en el uso de C++ y OOD (Diseño Orientado a Objetos) y abordarán cuestiones de ingeniería de software. Me esforzaré por escribir artículos pragmáticos y directamente útiles para el ingeniero de software "en la trinchera". En estos artículos, utilizaré la nueva notación unificada de Booch y Rumbaugh (Versión 0.8) para documentar diseños orientados a objetos. La barra lateral proporciona un breve glosario de esta notación.

Introducción Mi último artículo (enero '96) hablaba del principio Abierto-Cerrado. Este principio es la base para construir código que sea mantenible y reutilizable. Afirma que un código bien diseñado puede ser ampliado sin modificaciones; que en un programa bien diseñado, las nuevas características se agregan mediante la adición de nuevo código, en lugar de cambiar código antiguo que ya funciona.

Los mecanismos principales detrás del principio Abierto-Cerrado son la abstracción y el polimorfismo. En lenguajes tipados estáticamente como C++, uno de los mecanismos clave que respalda la abstracción y el polimorfismo es la herencia. Es mediante el uso de la herencia que podemos crear clases derivadas que se ajustan a las interfaces polimórficas abstractas definidas por funciones virtuales puras en clases base abstractas. ¿Cuáles son las reglas de diseño que gobiernan este uso particular de la herencia? ¿Cuáles son las características de las mejores jerarquías de herencia? ¿Cuáles son las trampas que nos llevarán a crear jerarquías que no se ajustan al principio Abierto-Cerrado? Estas son las preguntas a las que este artículo tratará de responder.

Clase Base

Derivada 1 Derivada 2

Poseído por Referencia

Poseído por Valor

Usado Barra lateral: Notación Unificada 0.8

El Principio de Sustitución de Liskov 2 El Principio de Sustitución de Liskov LAS FUNCIONES QUE UTILIZAN PUNTEROS O REFERENCIAS A CLASES BASE DEBEN PODER UTILIZAR OBJETOS DE CLASES DERIVADAS SIN SABERLO. La frase anterior es una paráfrasis del Principio de Sustitución de Liskov (LSP). Barbara Liskov lo escribió por primera vez hace casi 8 años1:

Lo que se quiere aquí es algo así como la siguiente propiedad de sustitución: si para cada objeto o1 de tipo S hay un objeto o2 de tipo T tal que para todos los programas P definidos en términos de T, el comportamiento de P no cambia cuando o1 se sustituye por o2, entonces S es un subtipo de T.

La importancia de este principio se vuelve obvia cuando se consideran las consecuencias de violarlo.

El Problema Real

En este momento, tenemos dos clases, Square y Rectangle, que parecen funcionar. No importa lo que hagas con un objeto Square, seguirá siendo coherente con un cuadrado matemático. Y sin importar lo que hagas con un objeto Rectangle, seguirá siendo un rectángulo matemático. Además, puedes pasar un Square a una función que acepta un puntero o referencia a un Rectangle, y el Square seguirá actuando como un cuadrado y permanecerá coherente.

Por lo tanto, podríamos concluir que el modelo es ahora autoconsistente y correcto. Sin embargo, esta conclusión sería equivocada. Un modelo que es autoconsistente no es necesariamente consistente con todos sus usuarios. Considera la función g a continuación.

void g(Rectangle& r)

{

r.SetWidth(5);

r.SetHeight(4);

assert((r.GetWidth() \* r.GetHeight()) == 20);

}

Esta función invoca los miembros SetWidth y SetHeight de lo que cree que es un Rectangle. La función funciona perfectamente para un Rectangle, pero declara un error de aserción si se le pasa un Square. Entonces, aquí está el problema real: ¿El programador que escribió esa función estaba justificado al asumir que cambiar el ancho de un Rectangle deja inalterada su altura?

Claramente, el programador de g hizo esta suposición muy razonable. Pasar un Square a funciones cuyos programadores hicieron esta suposición resultará en problemas. Por lo tanto, existen funciones que toman punteros o referencias a objetos Rectangle, pero no pueden operar correctamente sobre objetos Square. Estas funciones exponen una violación del LSP. La adición de la derivada Square de Rectangle ha roto estas funciones, y así se ha violado el principio Abierto-Cerrado.

La Validez no es Intrínseca

Esto nos lleva a una conclusión muy importante. Un modelo, visto de manera aislada, no puede validarse de manera significativa. La validez de un modelo solo se puede expresar en términos de sus clientes. Por ejemplo, cuando examinamos la versión final de las clases Square y Rectangle de manera aislada, encontramos que eran autoconsistentes y válidas. Sin embargo, cuando las miramos desde el punto de vista de un programador que hizo suposiciones razonables sobre la clase base, el modelo se desmoronó.

Por lo tanto, al considerar si un diseño en particular es apropiado o no, no se debe ver simplemente la solución de manera aislada. Se debe ver en términos de las suposiciones razonables que harán los usuarios de ese diseño.

¿Qué Salió Mal?

Entonces, ¿qué pasó? ¿Por qué el modelo aparentemente razonable de Square y Rectangle se volvió malo? Después de todo, ¿no es un Square un Rectangle? ¿No se cumple la relación ISA?

¡No! Un cuadrado podría ser un rectángulo, pero un objeto Square definitivamente no es un objeto Rectangle. ¿Por qué? Porque el comportamiento de un objeto Square no es consistente con el comportamiento de un objeto Rectangle. ¡Comportamentalmente, un Square no es un Rectangle! Y es el comportamiento en lo que realmente se trata el software.

El LSP deja claro que, en OOD, la relación ISA se refiere al comportamiento. No al comportamiento intrínseco privado, sino al comportamiento público extrínseco; comportamiento en el que los clientes dependen. Por ejemplo, el autor de la función g anterior dependía del hecho de que los Rectangles se comportan de manera que su altura y ancho varían independientemente entre sí. Esa independencia de las dos variables es un comportamiento público extrínseco en el que es probable que dependan otros programadores.

Para que se cumpla el LSP y, con él, el principio Abierto-Cerrado, todas las derivadas deben cumplir con el comportamiento que los clientes esperan de las clases base que utilizan.

Diseño por Contrato

Hay una fuerte relación entre el LSP y el concepto de Diseño por Contrato expuesto por Bertrand Meyer. Utilizando este esquema, los métodos de las clases declaran precondiciones y postcondiciones. Las precondiciones deben ser verdaderas para que el método se ejecute. Al completarse, el método garantiza que la postcondición será verdadera.

Podemos ver la postcondición de Rectangle::SetWidth(double w) como:

assert((itsWidth == w) && (itsHeight == old.itsHeight));

Ahora, la regla para las precondiciones y postcondiciones para las derivadas, como indica Meyer, es:

"... al redefinir una rutina [en una derivada], solo puedes reemplazar su precondición por una más débil, y su postcondición por una más fuerte."

En otras palabras, al usar un objeto a través de su interfaz de clase base, el usuario conoce solo las precondiciones y postcondiciones de la clase base. Por lo tanto, las clases derivadas no deben esperar que los usuarios cumplan con precondiciones más fuertes de las requeridas por la clase base. Es decir, deben aceptar cualquier cosa que la clase base pueda aceptar. Además, las clases derivadas deben cumplir con todas las postcondiciones de la clase base. Es decir, sus comportamientos y salidas no deben violar ninguna de las restricciones establecidas para la clase base. Los usuarios de la clase base no deben confundirse con la salida de la clase derivada.

Claramente, la postcondición de Square::SetWidth(double w) es más débil que la postcondición de Rectangle::SetWidth(double w) anterior, ya que no cumple con la cláusula de la clase base "(itsHeight == old.itsHeight)". Por lo tanto, Square::SetWidth(double w) viola el contrato de la clase base.

Algunos lenguajes, como Eiffel, tienen soporte directo para precondiciones y postcondiciones. Puedes declararlos y hacer que el sistema en tiempo de ejecución los verifique por ti. C++ no tiene esa característica. Sin embargo, incluso en C++, podemos considerar manualmente las precondiciones y postcondiciones de cada método y asegurarnos de que no se viole la regla de Meyer. Además, puede ser muy útil documentar estas precondiciones y postcondiciones en los comentarios de cada método.

Un Ejemplo Real

Suficiente de cuadrados y rectángulos. ¿El LSP tiene relevancia en el software real? Echemos un vistazo a un estudio de caso que proviene de un proyecto en el que trabajé hace algunos años.

Motivación

No estaba contento con las interfaces de las clases contenedor que estaban disponibles a través de terceros. No quería que el código de mi aplicación dependiera en gran medida de estos contenedores porque sentía que querría reemplazarlos con clases mejores más adelante. Por lo tanto, envolví los contenedores de terceros en mi propia interfaz abstracta. (Ver Figura 2)

Tenía una clase abstracta llamada Set que presentaba funciones puramente virtuales Add, Delete y IsMember.

cpp

template <class T>

class Set

{

public:

virtual void Add(const T&) = 0;

virtual void Delete(const T&) = 0;

virtual bool IsMember(const T&) const = 0;

};

Esta estructura unificaba las variedades Ilimitada y Acotada de los dos conjuntos de terceros y les permitía ser accedidos a través de una interfaz común. Por lo tanto, algún cliente podría aceptar un argumento de tipo Set<T>& y no le importaría si el Set real en el que trabajaba era de tipo Acotado o Ilimitado. (Ver la función PrintSet).

Esta capacidad de no conocer ni preocuparse por el tipo de Set en el que está operando es una gran ventaja. Significa que el programador puede decidir qué tipo de Set se necesita en cada instancia particular. Ninguna de las funciones del cliente se verá afectada por esa decisión. El programador puede elegir un BoundedSet cuando la memoria es escasa y la velocidad no es crítica, o puede elegir un UnboundedSet cuando la memoria es abundante y la velocidad es crítica. Las funciones del cliente manipularán estos objetos a través de la interfaz de la clase base Set y, por lo tanto, no sabrán ni se preocuparán por qué tipo de Set están usando.

Problema

Quería agregar un PersistentSet a esta jerarquía. Un conjunto persistente es un conjunto que se puede escribir en un flujo y luego leer más tarde, posiblemente por una aplicación diferente. Desafortunadamente, el único contenedor de terceros al que tenía acceso y que también ofrecía persistencia no era una clase de plantilla. En cambio, aceptaba objetos que eran PersistentObject.

template <class T>

void PrintSet(const Set<T>& s)

{

for (Iterator<T> i(s); i; i++)

cout << (\*i) << endl;

}

Derivé PersistentSet de la clase base abstracta PersistentObject. Creé la jerarquía que se muestra en la Figura 3.

A simple vista, esto puede parecer correcto. Sin embargo, hay una implicación que es bastante fea. Cuando un cliente está agregando miembros a la clase base Set, ¿cómo se supone que ese cliente garantice que solo agrega derivadas de PersistentObject si Set resulta ser un PersistentSet?

Considera el código para PersistentSet::Add:

template <class T>

void PersistentSet::Add(const T& t)

{

PersistentObject& p = dynamic\_cast<PersistentObject&>(t); // lanza bad\_cast

itsThirdPartyPersistentSet.Add(p);

}

Este código deja claro que si algún cliente intenta agregar un objeto que no es derivado de la clase PersistentObject a mi PersistentSet, se producirá un error en tiempo de ejecución. El dynamic\_cast lanzará bad\_cast (uno de los objetos de excepción estándar). Ninguno de los clientes existentes de la clase base Set espera que se lancen excepciones en Add. Dado que estas funciones se confundirán con una derivada de Set, este cambio en la jerarquía viola el LSP.

¿Es esto un problema? Ciertamente. Las funciones que nunca antes fallaron cuando se les pasó una derivada de Set ahora causarán errores en tiempo de ejecución cuando se les pase un PersistentSet. Depurar este tipo de problema es relativamente difícil, ya que el error en tiempo de ejecución ocurre muy lejos de la falla lógica real. La falla lógica es la decisión de pasar un PersistentSet a la función fallida, o es la decisión de agregar un objeto al PersistentSet que no es derivado de PersistentObject. En ambos casos, la decisión real podría estar a millones de instrucciones de distancia de la invocación real del método Add. Encontrarlo puede ser un problema. Arreglarlo puede ser peor.

Una Solución que no Conforma con el LSP

¿Cómo solucionamos este problema? Hace varios años, lo resolví por convención. Es decir, no lo resolví en el código fuente. En cambio, establecí una convención por la cual PersistentSet y PersistentObject no eran conocidos por la aplicación en su conjunto. Solo eran conocidos por un módulo en particular

**Conclusión**

El principio Abierto-Cerrado es el corazón de muchas afirmaciones realizadas en favor de la Programación Orientada a Objetos (POO). Es cuando este principio está en efecto que las aplicaciones son más mantenibles, reutilizables y robustas. El Principio de Sustitución de Liskov (también conocido como Diseño por Contrato) es una característica importante de todos los programas que cumplen con el principio Abierto-Cerrado. Solo cuando los tipos derivados son completamente sustituibles por sus tipos base, las funciones que utilizan esos tipos base pueden ser reutilizadas sin temor, y los tipos derivados pueden ser cambiados sin inconvenientes.

Este artículo es una versión extremadamente condensada de un capítulo de mi nuevo libro: "Patrones y Principios Avanzados de OOD", que será publicado pronto por Prentice Hall. En artículos subsiguientes, exploraremos muchos de los otros principios del diseño orientado a objetos. También estudiaremos varios patrones de diseño y sus fortalezas y debilidades con respecto a la implementación en C++. Analizaremos 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.