**ISP: Principio de Segregación de Interfaces**

Este es el cuarto de mis artículos en el cuaderno de ingeniería para The C++ Report. Los artículos que aparecen en esta columna se centran en el uso de C++ y OOD, abordando cuestiones de ingeniería de software. Me esfuerzo por escribir artículos pragmáticos y directamente útiles para el ingeniero de software en la trinchera. En estos artículos, hago uso del Lenguaje de Modelado Unificado (UML) unificado de Booch y Rumbaugh (Versión 0.8) para documentar diseños orientados a objetos.

Introducción En mi última columna (mayo de 1996), discutí el principio de Inversión de Dependencias (DIP). Este principio establece que los módulos que encapsulan políticas de alto nivel no deben depender de módulos que implementen detalles. Ambos tipos de módulos deben depender de abstracciones. Para ser más conciso y simplista, las clases abstractas no deben depender de clases concretas; las clases concretas deben depender de clases abstractas. Un buen ejemplo de este principio es el patrón de Método de Plantilla (TEMPLATE METHOD) del libro "Design Patterns" (GOF). En este patrón, un algoritmo de alto nivel está codificado en una clase base abstracta y utiliza funciones virtuales puras para implementar sus detalles. Las clases derivadas implementan esas funciones virtuales detalladas. Así, la clase que contiene los detalles depende de la clase que contiene la abstracción.

En este artículo examinaremos otro principio estructural: el Principio de Segregación de Interfaces (ISP). Este principio aborda las desventajas de las interfaces "gruesas". Las clases que tienen interfaces "gruesas" son clases cuyas interfaces no son cohesivas. En otras palabras, las interfaces de la clase se pueden dividir en grupos de funciones miembro. Cada grupo sirve a un conjunto diferente de clientes. Por lo tanto, algunos clientes utilizan un grupo de funciones miembro y otros clientes utilizan los otros grupos.

El ISP reconoce que hay objetos que requieren interfaces no cohesivas; sin embargo, sugiere que los clientes no deben conocerlas como una única clase. En su lugar, los clientes deben conocer clases base abstractas que tengan interfaces cohesivas. Algunos lenguajes se refieren a estas clases base abstractas como "interfaces", "protocolos" o "firmas".

En este artículo, discutiremos las desventajas de las interfaces "gruesas" o "contaminadas". Mostraremos cómo se crean estas interfaces y cómo diseñar clases que las oculten. Finalmente, presentaremos un estudio de caso en el cual una interfaz "gruesa" ocurre naturalmente y emplearemos el ISP para corregirla.

**Contaminación de Interfaces** Considera un sistema de seguridad. En este sistema, hay objetos de tipo **Door** que pueden bloquearse y desbloquearse, y que saben si están abiertos o cerrados

// Listado 1

// Puerta de seguridad

class Door

{

public:

virtual void Lock() = 0;

virtual void Unlock() = 0;

virtual bool IsDoorOpen() = 0;

};

Esta clase es abstracta para que los clientes puedan usar objetos que cumplan con la interfaz **Door** sin depender de implementaciones particulares de **Door**. Ahora considera que una de esas implementaciones, **TimedDoor**, necesita activar una alarma cuando la puerta ha estado abierta durante demasiado tiempo. Para hacer esto, el objeto **TimedDoor** se comunica con otro objeto llamado **Timer**

// Listado 2

// Temporizador

class Timer

{

public:

void Register(int timeout, TimerClient\* client);

};

class TimerClient

{

public:

virtual void Timeout() = 0;

};

Cuando un objeto desea ser informado sobre un temporizador, llama a la función Register del Timer. Los argumentos de esta función son el tiempo de expiración y un puntero a un objeto TimerClient cuya función Timeout se llamará cuando expire el temporizador.

¿Cómo podemos hacer que la clase TimerClient se comunique con la clase TimedDoor para que el código en TimedDoor pueda ser notificado del temporizador? Hay varias alternativas. La Figura 1 muestra una solución común. Forzamos a Door, y por lo tanto a TimedDoor, a heredar de TimerClient. Esto asegura que TimerClient pueda registrarse con el Timer y recibir el mensaje Timeout.

Aunque esta solución es común, no está exenta de problemas. El principal de ellos es que la clase Door ahora depende de TimerClient. No todas las variedades de Door necesitan temporización. De hecho, la abstracción original de Door no tenía nada que ver con el tiempo. Si se crean derivados de Door sin temporización, esos derivados tendrán que proporcionar implementaciones nulas para el método Timeout. Además, las aplicaciones que usan esos derivados deberán incluir la definición de la clase TimerClient, aunque no se use.

La Figura 1 muestra un síndrome común de diseño orientado a objetos en lenguajes con tipos estáticos como C++. Este es el síndrome de contaminación de interfaces. La interfaz de Door se ha contaminado con una interfaz que no necesita. Se ha visto obligada a incorporar esta interfaz únicamente para beneficiar a una de sus subclases. Si se sigue esta práctica, cada vez que un derivado necesita una nueva interfaz, esa interfaz se agregará a la clase base. Esto contaminará aún más la interfaz de la clase base, haciéndola "gruesa".

Además, cada vez que se agrega una nueva interfaz a la clase base, esa interfaz debe implementarse (o permitirse por defecto) en las clases derivadas. De hecho, una práctica asociada es agregar estas interfaces a la clase base como funciones virtuales nulas en lugar de funciones virtuales puras, específicamente para que las clases derivadas no se vean obligadas a implementarlas. Como aprendimos en el segundo artículo de esta columna, tal práctica viola el Principio de Sustitución de Liskov (LSP), lo que lleva a problemas de mantenimiento y reutilización.

Clientes Separados Significan Interfaces Separadas Door y TimerClient representan interfaces utilizadas por clientes completamente diferentes. Timer utiliza TimerClient, y las clases que manipulan puertas usan Door. Dado que los clientes son distintos, las interfaces también deberían permanecer separadas. ¿Por qué? Porque, como veremos en la siguiente sección, los clientes ejercen fuerzas sobre las interfaces de sus servidores.

**La fuerza contraria aplicada por los clientes sobre las interfaces.**

Cuando pensamos en las fuerzas que causan cambios en el software, normalmente pensamos en cómo los cambios en las interfaces afectarán a sus usuarios. Por ejemplo, nos preocuparíamos por los cambios en todos los usuarios de TimerClient si la interfaz de TimerClient cambiara. Sin embargo, hay una fuerza que opera en la otra dirección. Es decir, a veces es el usuario quien fuerza un cambio en la interfaz.

Por ejemplo, algunos usuarios de Timer registrarán más de una solicitud de tiempo de espera. Considera la TimedDoor. Cuando detecta que la puerta se ha abierto, envía el mensaje Register al Timer, solicitando un tiempo de espera. Sin embargo, antes de que expire ese tiempo de espera, la puerta se cierra, permanece cerrada durante un tiempo y luego se abre nuevamente. Esto nos lleva a registrar una nueva solicitud de tiempo de espera antes de que expire la antigua. Finalmente, expira la primera solicitud de tiempo de espera y se invoca la función TimeOut de TimedDoor. Y la alarma de la puerta suena falsamente.

Podemos corregir esta situación utilizando la convención mostrada en el Listado 3. Incluimos un código único de timeOutId en cada registro de tiempo de espera y repetimos ese código en la llamada TimeOut al TimerClient. Esto permite que cada derivado de TimerClient sepa a qué solicitud de tiempo de espera se está respondiendo.

// Listado 3

// Timer con ID

class Timer

{

public:

void Register(int timeout, int timeOutId, TimerClient\* client);

};

class TimerClient

{

public:

virtual void TimeOut(int timeOutId) = 0;

};

Claramente, este cambio afectará a todos los usuarios de TimerClient. Aceptamos esto ya que la falta de timeOutId es un descuido que necesita corrección. Sin embargo, el diseño en la Figura 1 también hará que Door y todos los clientes de Door se vean afectados (es decir, al menos recompilados) por esta corrección. ¿Por qué debería un error en TimerClient tener algún efecto en los clientes de las derivadas de Door que no requieren temporización? Es este tipo de interdependencia extraña lo que aterra a los clientes y gerentes. Cuando un cambio en una parte del programa afecta a otras partes completamente no relacionadas del programa, el costo y las repercusiones de los cambios se vuelven impredecibles; y el riesgo de consecuencias del cambio aumenta drásticamente.

Pero es solo una recompilación. Es cierto. Pero las recompilaciones pueden ser muy costosas por varias razones. En primer lugar, llevan tiempo. Cuando las recompilaciones llevan demasiado tiempo, los desarrolladores comienzan a tomar atajos. Pueden realizar un cambio en el "lugar equivocado" en lugar de realizar un cambio en el "lugar correcto"; porque el "lugar correcto" obligará a una gran recompilación. En segundo lugar, una recompilación significa un nuevo módulo de objeto. En esta época de bibliotecas vinculadas dinámicamente y cargadores incrementales, generar más módulos de objeto de los necesarios puede ser una desventaja significativa. Cuantas más DLL se vean afectadas por un cambio, mayor será el problema de distribuir y gestionar el cambio.

**Principio de Segregación de Interfaces (ISP)** **LOS CLIENTES NO DEBEN SER OBLIGADOS A DEPENDER DE INTERFACES QUE NO UTILIZAN.** Cuando los clientes se ven obligados a depender de interfaces que no utilizan, entonces esos clientes están sujetos a cambios en esas interfaces. Esto resulta en un acoplamiento involuntario entre todos los clientes. Dicho de otra manera, cuando un cliente depende de una clase que contiene interfaces que el cliente no utiliza, pero que otros clientes sí utilizan, entonces ese cliente se verá afectado por los cambios que esos otros clientes fuerzan en la clase. Nos gustaría evitar tales acoplamientos cuando sea posible, y por eso queremos separar las interfaces cuando sea posible.

**Interfaces de Clase vs Interfaces de Objeto** Considera la TimedDoor nuevamente. Aquí hay un objeto que tiene dos interfaces separadas utilizadas por dos clientes diferentes: Timer y los usuarios de Door. Estas dos interfaces deben implementarse en el mismo objeto, ya que la implementación de ambas interfaces manipula los mismos datos. Entonces, ¿cómo podemos cumplir con el ISP? ¿Cómo podemos separar las interfaces cuando deben permanecer juntas?

La respuesta a esto radica en el hecho de que los clientes de un objeto no necesitan acceder a través de la interfaz del objeto. Más bien, pueden acceder a través de la delegación o a través de una clase base del objeto.

**Separación a través de Delegación** Podemos emplear la forma de objeto del patrón ADAPTER para abordar el problema de TimedDoor. La solución es crear un objeto adaptador que herede de TimerClient y delegue a TimedDoor. La Figura 2 muestra esta solución.

Cuando TimedDoor desea registrar una solicitud de tiempo de espera con el Timer, crea un DoorTimerAdapter y lo registra en el Timer. Cuando el Timer envía el mensaje TimeOut al DoorTimerAdapter, el DoorTimerAdapter delega el mensaje de nuevo a TimedDoor.

Esta solución cumple con el ISP y evita el acoplamiento de los clientes de Door con Timer. Incluso si se hiciera el cambio a Timer mostrado en el Listado 3, ninguno de los usuarios de Door se vería afectado. Además, TimedDoor no tiene que tener la misma interfaz que TimerClient. El DoorTimerAdapter puede traducir la interfaz de TimerClient a la interfaz de TimedDoor. Por lo tanto, esta es una solución muy versátil.

// Listado 4

// Forma de Objeto del Patrón Adaptador

class TimedDoor : public Door

{

public:

virtual void DoorTimeOut(int timeOutId);

};

class DoorTimerAdapter : public TimerClient

{

public:

DoorTimerAdapter(TimedDoor& theDoor)

: itsTimedDoor(theDoor)

{}

virtual void TimeOut(int timeOutId)

{

itsTimedDoor.DoorTimeOut(timeOutId);

}

private:

TimedDoor& itsTimedDoor;

};

Sin embargo, esta solución también es algo inelegante. Involucra la creación de un nuevo objeto cada vez que queremos registrar un tiempo de espera. Además, la delegación requiere una cantidad muy pequeña, pero aún no nula, de tiempo de ejecución y memoria. Hay dominios de aplicación, como los sistemas de control en tiempo real integrado, en los que el tiempo de ejecución y la memoria son escasos lo suficiente como para que esto sea una preocupación.

**Separación a través de la Herencia Múltiple** La Figura 3 y el Listado 5 muestran cómo se puede utilizar la Herencia Múltiple, en la forma de clase del patrón ADAPTER, para lograr el ISP. En este modelo, TimedDoor hereda tanto de Door como de TimerClient. Aunque los clientes de ambas clases base pueden hacer uso de TimedDoor, ninguno depende realmente de la clase TimedDoor. Por lo tanto, utilizan el mismo objeto a través de interfaces separadas.

// Listado 5

// Forma de Clase del Patrón Adaptador

class TimedDoor : public Door, public TimerClient

{

public:

virtual void TimeOut(int timeOutId);

};

Esta solución es mi preferencia normal. La Herencia Múltiple no me asusta. De hecho, la encuentro bastante útil en casos como este. La única vez que elegiría la solución en la Figura 2 sobre la Figura 3 es si la traducción realizada por el objeto DoorTimerAdapter fuera necesaria, o si se necesitaran traducciones diferentes en momentos diferentes.

**Ejemplo de Interfaz de Usuario del Cajero Automático (ATM)** Ahora consideremos un ejemplo ligeramente más significativo. El tradicional problema del Cajero Automático (ATM). La interfaz de usuario de una máquina ATM necesita ser muy flexible. La salida puede necesitar ser traducida a muchos idiomas diferentes. Puede necesitar presentarse en una pantalla, en una tableta braille o ser hablada por un sintetizador de voz. Claramente, esto se puede lograr creando una clase base abstracta que tenga funciones virtuales puras para todos los mensajes diferentes que necesita presentar la interfaz.

Considera también que cada transacción diferente que puede realizar el cajero automático se encapsula como una derivada de la clase Transaction. Por lo tanto, podríamos tener clases como DepositTransaction, WithdrawalTransaction, TransferTransaction, etc. Cada uno de estos objetos emite mensajes a la interfaz de usuario. Por ejemplo, el objeto DepositTransaction llama a la función miembro RequestDepositAmount de la clase UI. Mientras que el objeto TransferTransaction llama a la función miembro RequestTransferAmount de UI. Esto corresponde al diagrama en la Figura 5.

Observa que esta es precisamente la situación que el ISP nos dice que evitemos. Cada una de las transacciones está utilizando una porción de la interfaz de usuario que ningún otro objeto usa. Esto crea la posibilidad de que los cambios en una de las derivadas de Transaction obliguen a un cambio correspondiente en la interfaz de usuario, afectando así a todas las demás derivadas de Transaction y a cada otra clase que depende de la interfaz de usuario.

Este acoplamiento desafortunado se puede evitar segregando la interfaz de usuario en clases base abstractas individuales, como DepositUI, WithdrawUI y TransferUI. Estas clases base abstractas pueden heredarse luego en la clase abstracta final de la interfaz de usuario. La Figura 6 y el Listado 6 muestran este modelo.

Es cierto que, siempre que se cree una nueva derivada de la clase Transaction, se necesitará una clase base correspondiente para la interfaz de usuario abstracta. Por lo tanto, la clase de interfaz de usuario y todas sus derivadas deben cambiar. Sin embargo, estas clases no se usan ampliamente. De hecho, probablemente solo se utilicen en main, o cualquier proceso que inicie el sistema y cree la instancia concreta de la interfaz de usuario. Por lo tanto, el impacto de agregar nuevas clases base de interfaz de usuario está contenido.

// Listado 6

// Interfaces de Cajero Automático (ATM) Segregadas

class DepositUI

{

public:

virtual void RequestDepositAmount() = 0;

};

// Transacción Abstracta

class Transaction

{

public:

virtual void Execute() = 0;

};

// Clases de Transacción Derivadas

class DepositTransaction : public Transaction

{

public:

DepositTransaction(DepositUI& ui)

: itsDepositUI(ui)

{}

virtual void Execute()

{

//...

itsDepositUI.RequestDepositAmount();

//...

}

private:

DepositUI& itsDepositUI;

};

// Clase de Interfaz de Usuario Abstracta

class UI : public DepositUI,

public WithdrawUI,

public TransferUI

{

public:

virtual void RequestDepositAmount();

virtual void RequestWithdrawlAmount();

virtual void RequestTransferAmount();

};

Un examen cuidadoso del Listado 6 mostrará uno de los problemas con la conformidad al ISP que no era evidente en el ejemplo de TimedDoor. Observa que cada transacción debe saber de alguna manera sobre su versión específica de la interfaz de usuario. DepositTransaction debe conocer DepositUI; WithdrawTransaction debe conocer WithdrawUI, etc. En el Listado 6, he abordado este problema obligando a cada transacción a construirse con una referencia a su interfaz de usuario específica.

// Listado 7

// Idiom de Inicialización de Interfaz

UI Gui; // objeto global;

void f()

{

DepositTransaction dt(Gui);

}

Esto es conveniente, pero también obliga a cada transacción a contener una referencia a su interfaz de usuario. Otra forma de abordar este problema es crear un conjunto de constantes globales como se muestra en el Listado 8. Como descubrimos cuando discutimos el Principio Abierto-Cerrado en el número de enero de 96, las variables globales no siempre son un síntoma de un diseño deficiente. En este caso, proporcionan la ventaja distintiva de un fácil acceso. Y dado que son referencias, es imposible cambiarlas de ninguna manera, por lo tanto, no se pueden manipular de una manera que sorprendería a otros usuarios.

Conclusión :

En este artículo, hemos discutido las desventajas de las "interfaces extensas"; es decir, interfaces que no son específicas para un solo cliente. Las interfaces extensas llevan a acoplamientos inadvertidos entre clientes que de otra manera deberían estar aislados. Al hacer uso del patrón ADAPTADOR, ya sea a través de la delegación (en forma de objeto) o de la herencia múltiple (en forma de clase), las interfaces extensas pueden ser segregadas en clases base abstractas que rompen el acoplamiento no deseado entre clientes.

Este artículo es una versión extremadamente condensada de un capítulo de mi nuevo libro: "Patrones y Principios Avanzados de OOD" que será publicado próximamente por Prentice Hall. En artículos subsecuentes, exploraremos muchos de los otros principios del diseño orientado a objetos. También estudiaremos varios patrones de diseño, así como sus fortalezas y debilidades con respecto a su 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. Después de eso, discutiremos muchos otros temas interesantes.

**Principio del formulario**