Principio de Inversión de Dependencias

Este es el tercero 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, abordando problemas de ingeniería de software. Me esforzaré por presentar artículos pragmáticos y directamente útiles para el ingeniero de software en la trinchera. En estos artículos, haré uso de la nueva notación unificada de Booch y Rumbaugh (Versión 0.8) para documentar diseños orientados a objetos. El cuadro lateral proporciona un breve léxico de esta notación.

Introducción

Mi último artículo (marzo de 96) trató sobre el Principio de Sustitución de Liskov (LSP). Este principio, cuando se aplica a C++, proporciona orientación para el uso de la herencia pública. Establece que cada función que opera sobre una referencia o puntero a una clase base debe poder operar sobre derivadas de esa clase base sin saberlo. Esto significa que las funciones miembro virtuales de las clases derivadas deben esperar no más de lo que hacen las funciones miembro correspondientes de la clase base y no deben prometer menos. También significa que las funciones miembro virtuales presentes en las clases base también deben estar presentes en las clases derivadas y deben realizar un trabajo útil. Cuando este principio se viola, las funciones que operan sobre punteros o referencias a clases base deberán verificar el tipo del objeto real para asegurarse de que pueden operar correctamente. Esta necesidad de verificar el tipo viola el Principio Abierto/Cerrado (OCP) que discutimos el pasado enero. En este artículo, discutimos las implicaciones estructurales del OCP y el LSP. La estructura que resulta del uso riguroso de estos principios puede generalizarse en un principio en sí mismo. Lo llamo "El Principio de Inversión de Dependencias" (DIP).

Cuadro Lateral: Notación Unificada 0.8

Clase Base

Derivada 1 Derivada 2

Poseída por Referencia

Poseída por Valor

Usada

Qué sale mal con el software? 2 Qué sale mal con el software? La mayoría de nosotros ha tenido la desagradable experiencia de tratar con un software que tiene un "diseño deficiente". Algunos de nosotros incluso hemos tenido la experiencia mucho más desagradable de descubrir que éramos los autores del software con "diseño deficiente". ¿Qué hace que un diseño sea malo? La mayoría de los ingenieros de software no tienen la intención de crear "diseños deficientes". Sin embargo, la mayoría del software eventualmente degrada hasta el punto en que alguien declarará que el diseño no es sólido. ¿Por qué sucede esto? ¿El diseño fue pobre desde el principio, o el diseño realmente se degradó como un pedazo de carne podrida? En el corazón de este problema está nuestra falta de una buena definición práctica de diseño "deficiente". La Definición de un "Diseño Deficiente" ¿Alguna vez has presentado un diseño de software, del cual estabas especialmente orgulloso, para su revisión por parte de un colega? ¿Dijo ese colega, con un tono de desprecio y queja, algo como: "¿Por qué lo hiciste de esa manera?". Ciertamente, esto me ha sucedido a mí, y he visto que le sucede a muchos otros ingenieros también. Claramente, los ingenieros que no están de acuerdo no están utilizando los mismos criterios para definir qué es un diseño "deficiente". El criterio más común que he visto es el criterio TNTWI-WHDI o "Esa no es la forma en que yo lo hubiera hecho". Pero hay un conjunto de criterios con el que creo que todos los ingenieros estarán de acuerdo. Un software que cumple con sus requisitos y aún exhibe cualquiera o todas las siguientes tres características tiene un diseño deficiente.

1. Es difícil de cambiar porque cada cambio afecta a demasiadas otras partes del sistema. (Rigidez)
2. Cuando realizas un cambio, partes inesperadas del sistema se rompen. (Fragilidad)
3. Es difícil de reutilizar en otra aplicación porque no se puede desentrañar de la aplicación actual. (Inmovilidad) Además, sería difícil demostrar que un software que no exhibe ninguna de esas características, es decir, es flexible, robusto y reutilizable, y que también cumple con todos sus requisitos, tiene un diseño deficiente. Por lo tanto, podemos usar estas tres características como una forma de decidir de manera inequívoca si un diseño es "bueno" o "malo".

La Causa del "Diseño Deficiente". ¿Qué hace que un diseño sea rígido, frágil e inmóvil? Es la interdependencia de los módulos dentro de ese diseño. Un diseño es rígido si no se puede cambiar fácilmente. Tal rigidez se debe al hecho de que un solo cambio en un software altamente interdependiente inicia una cascada de cambios en módulos dependientes. Cuando los diseñadores o mantenedores no pueden prever la magnitud de esa cascada de cambios, no pueden estimar el impacto del cambio. Esto hace que el costo del cambio sea imposible de predecir. Los gerentes, enfrentados a tal imprevisibilidad, se muestran reacios a autorizar cambios. Así es como el diseño se vuelve oficialmente rígido. La fragilidad es la tendencia de un programa a romperse en muchos lugares cuando se realiza un solo cambio. A menudo, los nuevos problemas se encuentran en áreas que no tienen relación conceptual con el área que se cambió. Tal fragilidad disminuye enormemente la credibilidad del diseño y de la organización de mantenimiento. Los usuarios y los gerentes no pueden prever la calidad de su producto. Cambios simples en una parte de la aplicación llevan a fallas en otras partes que parecen estar completamente no relacionadas. Solucionar esos problemas conduce a aún más problemas, y el proceso de mantenimiento comienza a parecerse al juego del gato y el ratón. Un diseño es inmóvil cuando las partes deseables del diseño dependen en gran medida de otros detalles que no son deseables. Los diseñadores encargados de investigar el diseño para ver si se puede reutilizar en una aplicación diferente pueden quedar impresionados por lo bien que el diseño funcionaría en la nueva aplicación. Sin embargo, si el diseño es altamente interdependiente, esos diseñadores también se desanimarán por la cantidad de trabajo necesario para separar la parte deseable del diseño de las otras partes que no son deseables. En la mayoría de los casos, estos diseños no se reutilizan porque se considera que el costo de la separación es mayor que el costo de volver a desarrollar el diseño.

Ejemplo: el programa "Copy". Un ejemplo sencillo puede ayudar a ilustrar este punto. Considera un programa simple que tiene la tarea de copiar caracteres escritos en un teclado a una impresora. Supongamos, además, que la plataforma de implementación no tiene un sistema operativo que admita la independencia de dispositivos. Podríamos concebir una estructura para este programa que se ve como la Figura 1: La Figura 1 es un "diagrama de estructura"1. Muestra que hay tres módulos o subprogramas en la aplicación. El módulo "Copy" llama a los otros dos. Uno puede imaginar fácilmente un bucle dentro del módulo "Copy" (ver Listado 1). El cuerpo de ese bucle llama al módulo "Read Keyboard" para obtener un carácter del teclado, luego envía ese carácter al módulo "Write Printer" que imprime el carácter.

1. Ver: The Practical Guide To Structured Systems Design, de Meilir Page-Jones, Yourdon Press, 1988

La Figura 1. Programa Copy.

Copy

Read Keyboard

Write Printer

Inversión de Dependencias 4

Los dos módulos de bajo nivel son fácilmente reutilizables. Se pueden usar en muchos otros programas para acceder al teclado y a la impresora. Este es el mismo tipo de reutilización que obtenemos de las bibliotecas de subrutinas. Sin embargo, el módulo "Copy" no es reutilizable en ningún contexto que no involucre un teclado o una impresora. Es una lástima, ya que la inteligencia del sistema se mantiene en este módulo. Es el módulo "Copy" el que encapsula una política muy interesante que nos gustaría reutilizar. Por ejemplo, considera un nuevo programa que copia caracteres de un teclado a un archivo en disco. Ciertamente, nos gustaría reutilizar el módulo "Copy" ya que encapsula la política de alto nivel que necesitamos, es decir, sabe cómo copiar caracteres de una fuente a un destino. Desafortunadamente, el módulo "Copy" depende del módulo "Write Printer" y, por lo tanto, no se puede reutilizar en el nuevo contexto. Podríamos ciertamente modificar el módulo "Copy" para darle la nueva funcionalidad deseada (ver Listado 2). Podríamos agregar una declaración 'if' a su política y hacer que seleccione entre el módulo "Write Printer" y el módulo "Write Disk" según algún tipo de indicador. Sin embargo, esto agrega nuevas interdependencias al sistema. A medida que pasa el tiempo y más y más dispositivos deben participar en el programa de copia, el módulo "Copy" estará lleno de declaraciones 'if/else' y dependerá de muchos módulos de bajo nivel. Eventualmente se volverá rígido y frágil.

Inversión de Dependencias Una forma de caracterizar el problema anterior es darse cuenta de que el módulo que contiene la política de alto nivel, es decir, el módulo "Copy", depende de los módulos detallados de bajo nivel que controla (es decir, "Write Printer" y "Read Keyboard"). Si pudiéramos encontrar una manera de hacer que el módulo "Copy" fuera independiente de los detalles que controla, entonces podríamos reutilizarlo libremente. Podríamos producir otros programas que usaran este módulo para copiar caracteres de cualquier dispositivo de entrada a cualquier dispositivo de salida. OOD nos proporciona un mecanismo para realizar esta inversión de dependencias. Considera el simple diagrama de clases en la Figura 2. Aquí tenemos una clase "Copy" que contiene una clase abstracta "Reader" y una clase abstracta "Writer". Uno puede imaginar fácilmente un bucle dentro de la clase "Copy" que obtiene caracteres de su "Reader" y los envía a su "Writer" (ver Listado 3). Sin embargo, esta clase "Copy" no depende en absoluto del "Keyboard Reader" ni del "Printer Writer". Así es como se han invertido las dependencias; la clase "Copy" depende de abstracciones, y los lectores y escritores detallados dependen de las mismas abstracciones. Ahora podemos reutilizar la clase "Copy", independientemente del "Keyboard Reader" y del "Printer Writer". Podemos inventar nuevos tipos de derivados de "Reader" y "Writer" que podemos suministrar a la clase "Copy". Además, no importa cuántos tipos de "Readers" y "Writers" se creen, "Copy" no dependerá de ninguno de ellos. No habrá interdependencias para hacer que el programa sea frágil o rígido. Y "Copy" en sí mismo se puede usar en muchos contextos diferentes detallados. Es móvil.

Independencia de Dispositivos A estas alturas, algunos de ustedes probablemente se estén diciendo a sí mismos que podrían obtener los mismos beneficios escribiendo Copy() en C, usando la independencia de dispositivos inherente a stdio.h; es decir, getchar y putchar (ver Listado 4). Si consideras detenidamente los Listados 3 y 4, te darás cuenta de que son lógicamente equivalentes. Las clases abstractas en la Figura 3 han sido reemplazadas por un tipo diferente de abstracción en el Listado 4. Es cierto que el Listado 4 no utiliza clases y funciones virtuales puras, pero aún así utiliza abstracción y polimorfismo para lograr sus objetivos. Además, ¡todavía utiliza la inversión de dependencias! El programa Copy en el Listado 4 no depende de ninguno de los detalles que controla. Más bien, depende de las instalaciones abstractas declaradas en stdio.h. Además, los controladores de E/S que finalmente se invocan también dependen de las abstracciones declaradas en stdio.h. Así que la independencia de dispositivos dentro de la biblioteca stdio.h es otro ejemplo de inversión de dependencias.

Ahora que hemos visto algunos ejemplos, podemos declarar la forma general del DIP.

Principio de Inversión de Dependencias

A. LOS MÓDULOS DE ALTO NIVEL NO DEBERÍAN DEPENDER DE LOS MÓDULOS DE BAJO NIVEL. AMBOS DEBERÍAN DEPENDER DE ABSTRACCIONES. B. LAS ABSTRACCIONES NO DEBERÍAN DEPENDER DE DETALLES. LOS DETALLES DEBERÍAN DEPENDER DE LAS ABSTRACCIONES.

Uno podría preguntarse por qué uso la palabra "inversión". Francamente, es porque los métodos de desarrollo de software más tradicionales, como Análisis y Diseño Estructurado, tienden a crear estructuras de software en las que los módulos de alto nivel dependen de los módulos de bajo nivel, y en las que las abstracciones dependen de los detalles. De hecho, uno de los objetivos de estos métodos es definir la jerarquía de subprogramas que describe cómo los módulos de alto nivel hacen llamadas a los módulos de bajo nivel. La Figura 1 es un buen ejemplo de tal jerarquía. Así, la estructura de dependencia de un programa orientado a objetos bien diseñado está "invertida" con respecto a la estructura de dependencia que normalmente resulta de los métodos tradicionales de procedimientos.

Consideremos las implicaciones de los módulos de alto nivel que dependen de los módulos de bajo nivel. Son los módulos de alto nivel los que contienen las decisiones de política importantes y los modelos comerciales de una aplicación. Son estos modelos los que contienen la identidad de la aplicación. Sin embargo, cuando estos módulos dependen de los módulos de nivel inferior, los cambios en los módulos de nivel inferior pueden tener efectos directos sobre ellos y pueden forzarlos a cambiar. ¡Esta situación es absurda! Deberían ser los módulos de alto nivel los que obliguen a cambiar a los módulos de bajo nivel. Deberían tener prioridad sobre los módulos de bajo nivel. Los módulos de alto nivel simplemente no deberían depender de los módulos de bajo nivel de ninguna manera.

Además, son los módulos de alto nivel los que queremos poder reutilizar. Ya somos bastante buenos reutilizando módulos de bajo nivel en forma de bibliotecas de subrutinas. Cuando los módulos de alto nivel dependen de los módulos de bajo nivel, se vuelve muy difícil reutilizar esos módulos de alto nivel en contextos diferentes. Sin embargo, cuando los módulos de alto nivel son independientes de los módulos de bajo nivel, entonces se pueden reutilizar bastante fácilmente.

Esta es la idea fundamental detrás del Principio de Inversión de Dependencias. Es la base misma del diseño de frameworks.

**Capas**

Según Booch, "todas las arquitecturas orientadas a objetos bien estructuradas tienen capas claramente definidas, siendo cada capa responsable de un conjunto coherente de servicios a través de una interfaz bien definida y controlada." Una interpretación ingenua de esta afirmación podría llevar a un diseñador a producir una estructura similar a la de la Figura 3. En este diagrama, la clase de política de alto nivel utiliza un Mecanismo de nivel inferior, que a su vez utiliza una clase de utilidad de nivel detallado. Aunque esto puede parecer apropiado, tiene la característica insidiosa de que la Capa de Política es sensible a cambios en la Capa de Utilidad. La dependencia es transitiva. La Capa de Política depende de algo que depende de la Capa de Utilidad, por lo que la Capa de Política depende transitivamente de la Capa de Utilidad. Esto es muy desafortunado.

La Figura 4 muestra un modelo más apropiado. Cada una de las capas de nivel inferior está representada por una clase abstracta. Las capas reales se derivan luego de estas clases abstractas. Cada una de las clases de nivel superior utiliza la capa siguiente más baja a través de la interfaz abstracta. Así, ninguna de las capas depende de las otras capas. En cambio, las capas dependen de clases abstractas. No solo se rompe la dependencia transitoria de la Capa de Política sobre la Capa de Utilidad, sino que incluso la dependencia directa de la Capa de Política sobre la Capa de Mecanismo también se rompe.

**Separar Interfaz de Implementación en C++**

Alguien podría quejarse de que la estructura en la Figura 3 no exhibe los problemas de dependencia y dependencia transitoria que afirmé. Después de todo, la Capa de Política depende solo de la interfaz de la Capa de Mecanismo. ¿Cómo afectaría en absoluto un cambio en la implementación de la Capa de Mecanismo a la Capa de Política?

En algunos lenguajes orientados a objetos, esto sería cierto. En tales lenguajes, la interfaz está separada de la implementación automáticamente. Sin embargo, en C++, no hay separación entre interfaz e implementación. En cambio, en C++, la separación se encuentra entre la definición de la clase y la definición de sus funciones miembro.

En C++, generalmente separamos una clase en dos módulos: un módulo .h y un módulo .cc. El módulo .h contiene la definición de la clase, y el módulo .cc contiene la definición de las funciones miembro de esa clase. La definición de una clase, en el módulo .h, contiene declaraciones de todas las funciones miembro y variables miembro de la clase. Esta información va más allá de la simple interfaz. Todas las funciones de utilidad y las variables privadas necesarias para la clase también se declaran en el módulo .h. Estas utilidades y variables privadas son parte de la implementación de la clase, pero aparecen en el módulo en el que todos los usuarios de la clase deben depender. Por lo tanto, en C++, la implementación no se separa automáticamente de la interfaz.

Esta falta de separación entre interfaz e implementación en C++ se puede manejar mediante el uso de clases puramente abstractas. Una clase puramente abstracta es una clase que contiene nada más que funciones virtuales puras. Tal clase es pura interfaz, y su módulo .h no contiene implementación. La Figura 4 muestra tal estructura. Las clases abstractas en la Figura 4 están destinadas a ser puramente abstractas para que cada una de las capas dependa solo de la interfaz de la capa siguiente.

**Un Ejemplo Simple**

La Inversión de Dependencias se puede aplicar siempre que una clase envíe un mensaje a otra. Por ejemplo, considera el caso del objeto Botón y el objeto Lámpara.

El objeto Botón detecta el entorno externo. Puede determinar si un usuario lo ha "presionado" o no. No importa cuál sea el mecanismo de detección. Podría ser un ícono de botón en una GUI, un botón físico presionado por un dedo humano, o incluso un detector de movimiento en un sistema de seguridad doméstico. El objeto Botón detecta que un usuario lo ha activado o desactivado. El objeto lámpara afecta el entorno externo. Al recibir un mensaje "Encender", ilumina una luz de algún tipo. Al recibir un mensaje "Apagar", apaga esa luz. El mecanismo físico no es importante. Podría ser un LED en una consola de computadora, una lámpara de vapor de mercurio en un estacionamiento, o incluso el láser en una impresora láser.

¿Cómo podemos diseñar un sistema de manera que el objeto Botón controle el objeto Lámpara? La Figura 5 muestra un modelo ingenuo. El objeto Botón simplemente envía el mensaje "Encender" y "Apagar" a la Lámpara. Para facilitar esto, la clase Botón usa una relación de "contiene" para mantener una instancia de la clase Lámpara.

El Listado 5 muestra el código en C++ que resulta de este modelo. Observa que la clase Botón depende directamente de la clase Lámpara. De hecho, el módulo button.cc #incluye el módulo lamp.h. Esta dependencia implica que la clase botón debe cambiar, o al menos ser recompilada, cada vez que cambia la clase Lámpara. Además, no será posible reutilizar la clase Botón para controlar un objeto Motor.

La Figura 5 y el Listado 5 violan el principio de inversión de dependencias. La política de alto nivel de la aplicación no se ha separado de los módulos de bajo nivel; las abstracciones no se han separado de los detalles. Sin tal separación, la política de alto nivel depende automáticamente de los módulos de bajo nivel, y las abstracciones dependen automáticamente de los detalles.

**Encontrar la Abstracción Subyacente**

¿Cuál es la política de alto nivel? Son las abstracciones que subyacen en la aplicación, las verdades que no varían cuando cambian los detalles. En el ejemplo de Botón/Lámpara, la abstracción subyacente es detectar un gesto de encendido/apagado de un usuario y transmitir ese gesto a un objeto objetivo. ¿Qué mecanismo se utiliza para detectar el gesto del usuario? ¡Irrelevante! ¿Cuál es el objeto objetivo? ¡Irrelevante! Estos son detalles que no afectan la abstracción.

Para cumplir con el principio de inversión de dependencias, debemos aislar esta abstracción de los detalles del problema. Luego debemos dirigir las dependencias del diseño de manera que los detalles dependan de las abstracciones. La Figura 6 muestra un diseño de este tipo.

En la Figura 6, hemos aislado la abstracción de la clase Botón de su implementación detallada. El Listado 6 muestra el código correspondiente. Observa que la política de alto nivel está completamente capturada dentro de la clase de botón abstracta. La clase Botón no sabe nada del mecanismo físico para detectar los gestos del usuario, y no sabe nada en absoluto sobre la lámpara. Esos detalles están aislados dentro de las derivaciones concretas: BotónImplementacion y Lámpara.

La política de alto nivel en el Listado 6 es reutilizable con cualquier tipo de botón y con cualquier tipo de dispositivo que necesite ser controlado. Además, no se ve afectada por cambios en los mecanismos de bajo nivel. Por lo tanto, es robusta en presencia de cambios, flexible y reutilizable.

**Ampliando la Abstracción aún más**

Se podría hacer una queja legítima sobre el diseño en la Figura/Listado 6. El dispositivo controlado por el botón debe derivarse de ButtonClient. ¿Y si la clase Lámpara proviene de una biblioteca de terceros y no podemos modificar el código fuente?

Conclision:

El principio de inversión de dependencias está en la raíz de muchos de los beneficios reclamados para la tecnología orientada a objetos. Su aplicación adecuada es necesaria para la creación de frameworks reutilizables. También es de vital importancia para la construcción de código que sea resistente al cambio. Y, dado que las abstracciones y los detalles están todos aislados entre sí, el código es mucho más fácil de mantener.

Este artículo es una versión extremadamente condensada de un capítulo de mi nuevo libro: "Patterns and Advanced Principles of OOD", que será publicado próximamente por Prentice Hall. En artículos posteriores 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, discutiremos muchos otros temas interesantes.