Principios S.O.L.I.D.

SRP:El principio de responsabilidad única:

Una clase debería tener solo una razón para cambiar.

Durante la mayoría de su desarrollo la clase `Juego` estaba manejando dos responsabilidades. Se estaba ocupando de rastrear el marco actual y calcular la puntuación. Al final, RCM y RSK separaron estas dos responsabilidades en dos clases. La clase `Juego` mantuvo la responsabilidad de rastrear los marcos, y la clase `Puntuación` tuvo la responsabilidad de calcular la puntuación

¿Por qué era importante separar estas dos responsabilidades en clases distintas? Porque cada responsabilidad es un eje de cambio. Cuando cambian los requerimientos, ese cambio se manifestará a través de un cambio en las responsabilidades entre las clases. Si una clase asume más de una responsabilidad, entonces habrá más de una razón para que cambie.

Si una clase tiene más de una responsabilidad, las responsabilidades se acoplan. Los cambios en una responsabilidad pueden afectar o inhibir la capacidad de la clase para cumplir con las otras. Este tipo de acoplamiento lleva a diseños frágiles que se rompen de maneras inesperadas cuando se cambian.

Por ejemplo, considera el diseño en la Figura 8-1. La clase `Rectángulo` tiene dos métodos mostrados. Uno dibuja el rectángulo en la pantalla, el otro calcula el área del rectángulo.

Dos aplicaciones diferentes utilizan la clase `Rectángulo`. Una aplicación hace geometría computacional. Utiliza `Rectángulo` para ayudar con las matemáticas de las formas geométricas. Nunca dibuja el rectángulo en la pantalla. La otra aplicación es de naturaleza gráfica. También puede hacer algo de geometría computacional, pero definitivamente dibuja el rectángulo en la pantalla.

Este diseño viola el SRP. La clase Rectángulo tiene dos responsabilidades. La primera responsabilidad es proporcionar un modelo matemático de la geometría de un rectángulo. La segunda responsabilidad es renderizar el rectángulo en una interfaz gráfica de usuario.

La violación del SRP causa varios problemas. En primer lugar, debemos incluir la GUI en la aplicación de geometría computacional. En .NET, el ensamblaje de la GUI tendría que construirse y desplegarse junto con la aplicación de geometría computacional.

En segundo lugar, si un cambio en la `AplicaciónGráfica` hace que el `Rectángulo` cambie por alguna razón, ese cambio puede obligarnos a reconstruir, volver a probar y redesplegar la `AplicacióndeGeometríaComputacional`. Si olvidamos hacer esto, esa aplicación puede romperse de maneras impredecibles.

Un mejor diseño es separar las dos responsabilidades en dos clases completamente diferentes, como se muestra en la Figura 8-2. Este diseño mueve las partes computacionales de `Rectángulo` a la clase `RectánguloGeométrico`. Ahora los cambios realizados en la forma en que se renderizan los rectángulos no pueden afectar a la `AplicacióndeGeometríaComputacional`.

¿Qué es una Responsabilidad?

En el contexto del Principio de Responsabilidad Única (SRP), definimos una responsabilidad como "una razón para el cambio". Si puedes pensar en más de un motivo para cambiar una clase, entonces esa clase tiene más de una responsabilidad. Esto a veces es difícil de ver. Estamos acostumbrados a pensar en responsabilidades en grupos. Por ejemplo, considera la interfaz `Modem` en la Listado 8-1. La mayoría de nosotros estaremos de acuerdo en que esta interfaz parece perfectamente razonable. Las cuatro funciones que declara son funciones que pertenecen a un módem.

Sin embargo, aquí se muestran dos responsabilidades. La primera responsabilidad es la gestión de conexiones. La segunda es la comunicación de datos. Las funciones de `marcar` y `colgar` gestionan la conexión del módem, mientras que las funciones de `enviar` y `recibir` comunican datos.

¿Deberían separarse estas dos responsabilidades? Eso depende de cómo cambié la aplicación. Si la aplicación cambia de maneras que afectan la firma de las funciones de conexión, entonces el diseño olerá a **Rigidez**, porque las clases que llaman a `enviar` y `leer` tendrán que recompilarse y redesplegarse más a menudo de lo que nos gustaría. En ese caso, las dos responsabilidades deberían separarse, como se muestra en la Figura 8-3. Esto evita que las aplicaciones cliente acoplen las dos responsabilidades.

Si, por otro lado, la aplicación no está cambiando de maneras que causen que las dos responsabilidades cambien en momentos diferentes, entonces no hay necesidad de separarlas. De hecho, separarlas podría oler a **Complejidad Innecesaria**.

Hay un corolario aquí. *Un eje de cambio es solo un eje de cambio si los cambios realmente ocurren*. No es prudente aplicar el SRP, o cualquier otro principio, si no hay síntomas.

Separando responsabilidades acopladas.

Nota que en la Figura 8-3 mantuve ambas responsabilidades acopladas en la clase `MódemImplementación`. Esto no es deseable, pero puede ser necesario. A menudo hay razones, relacionadas con los detalles del hardware o el sistema operativo, que nos obligan a acoplar cosas que preferiríamos no acoplar. Sin embargo, al separar sus interfaces, hemos desacoplado los conceptos en lo que respecta al resto de la aplicación.

Podemos ver la clase `MódemImplementación` como un parche o una imperfección; sin embargo, observa que todas las dependencias fluyen *fuera* de ella. Nadie necesita depender de esta clase. Nadie excepto `main` necesita saber que existe. Así, hemos puesto lo feo detrás de una valla. Su fealdad no necesita filtrarse y contaminar el resto de la aplicación.

Persistencia.

La Figura 8-4 muestra una violación común del SRP. La clase `Empleado` contiene reglas de negocio y control de persistencia. Estas dos responsabilidades casi nunca deberían mezclarse. Las reglas de negocio tienden a cambiar con frecuencia, y aunque la persistencia puede no cambiar tan a menudo, cambia por razones completamente diferentes. Vincular las reglas de negocio al subsistema de persistencia es pedir problemas.

Afortunadamente, como vimos en el Capítulo 4, la práctica del desarrollo guiado por pruebas generalmente obligará a separar estas dos responsabilidades mucho antes de que el diseño comience a oler.

Sin embargo, en casos donde las pruebas no forzaron la separación, y los olores de **Rigidez** y **Fragilidad** se vuelven fuertes, el diseño debe ser refactorizado utilizando los patrones FACADE, DAO o PROXY para separar las dos responsabilidades.

Conclusión

El SRP es uno de los principios más simples y uno de los más difíciles de aplicar correctamente. Unir responsabilidades es algo que hacemos de manera natural. Encontrar y separar esas responsabilidades es gran parte de lo que realmente trata el diseño de software. De hecho, los demás principios que discutiremos regresan a este tema de una forma u otra.

OCP:El principio de abierto-cerrado:

Este es el primero de mis artículos de la sección "Engineering Notebook" para *The C++ Report*. Los artículos que aparecerán en esta columna se centrarán en el uso de C++ y el diseño orientado a objetos (OOD), y abordarán temas de ingeniería de software. Me esforzaré por ofrecer artículos que sean pragmáticos y directamente útiles para el ingeniero de software en el campo. 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.

Existen muchas heurísticas asociadas con el diseño orientado a objetos. Por ejemplo, "todas las variables de miembro deben ser privadas", "se deben evitar las variables globales", o "usar 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 su ciclo 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 frente al cambio y que duren más que la primera versión? Bertrand Meyer nos dio orientación hace mucho tiempo, en 1988, cuando acuñó el ahora famoso principio abierto-cerrado. Parafraseando:

LAS ENTIDADES DE SOFTWARE (CLASES, MÓDULOS, FUNCIONES, ETC.) DEBEN ESTAR ABIERTAS A LA EXTENSIÓN, PERO CERRADAS A LA MODIFICACIÓN.

Cuando un solo cambio en un programa resulta en una cascada de cambios en módulos dependientes, ese programa exhibe los atributos indeseables que hemos llegado a asociar con un "mal" diseño. El programa se vuelve frágil, rígido, impredecible y no reutilizable. El principio abierto-cerrado aborda esto de una manera muy directa. Dice que debes diseñar módulos que nunca cambien. Cuando cambian los requisitos, extiendes el comportamiento de tales módulos agregando nuevo código, no cambiando el código antiguo que ya funciona.

Descripción.

Los módulos que cumplen con el principio abierto-cerrado tienen dos atributos principales.

1. **Están "Abiertos a la Extensión"**.
   Esto significa que el comportamiento del módulo puede extenderse. Que podemos hacer que el módulo se comporte de nuevas y diferentes maneras a medida que cambian los requisitos de la aplicación, o para satisfacer las necesidades de nuevas aplicaciones.
2. **Están "Cerrados a la Modificación"**.
   El código fuente de tal módulo es inviolable. Nadie puede hacer cambios en el código fuente.

Parece que estos dos atributos están en desacuerdo entre sí. La forma normal de extender el comportamiento de un módulo es hacer cambios en ese módulo. Un módulo que no se puede cambiar normalmente se considera que tiene un comportamiento fijo. ¿Cómo se pueden resolver 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, sin embargo, 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 a la modificación ya que depende de una abstracción que es 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 `Client` como `Server` son concretas. No hay garantía de que las funciones miembro de la clase `Server` sean virtuales. La clase `Client` utiliza la clase `Server`. Si deseamos que un objeto `Client` use un objeto `Server` diferente, entonces la clase `Client` debe cambiar para nombrar la nueva clase `Server`.

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 `Client` utiliza esta abstracción. Sin embargo, los objetos de

la clase `Client` utilizarán objetos de la clase derivada `Server`. Si queremos que los objetos `Client` usen una clase server diferente, entonces se puede crear una nueva derivada de la clase `AbstractServer`. La clase `Client` puede permanecer sin cambios.

La abstracción de Forma.

Considera el siguiente ejemplo. Tenemos una aplicación que debe ser capaz de dibujar círculos y cuadrados en una GUI estándar. Los círculos y cuadrados deben dibujarse en un orden particular. Se creará una lista de los 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 la Lista 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 uno es un código de tipo que identifica la estructura de datos como un círculo o un cuadrado. La función `DrawAllShapes` recorre un arreglo de punteros a estas estructuras de datos, examinando el código de tipo y luego llamando a la función apropiada (ya sea `DrawCircle` o `DrawSquare`).

La función `DrawAllShapes` no cumple con el principio abierto-cerrado porque no puede cerrarse contra nuevos tipos de formas. Si quisiera extender 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 instrucción `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 tal aplicación significa buscar en cada lugar donde existan tales instrucciones `switch` (o cadenas `if/else`) y agregar la nueva forma a cada uno. Además, es muy poco probable que todas las instrucciones `switch` y cadenas `if/else` estén tan bien estructuradas como la de `DrawAllShapes`. Es mucho más probable que los predicados de las instrucciones if se combinen con operadores lógicos, o que las cláusulas `case` de las instrucciones `switch` se combinen para "simplificar" la toma de decisiones local. Así, el problema de encontrar y entender todos los lugares donde la nueva forma necesita ser agregada puede ser no trivial.

La Lista 2 muestra el código para una solución al problema cuadrado/círculo que cumple con el principio abierto-cerrado. En este caso, se crea una clase abstracta `Shape`. Esta clase abstracta tiene una única función puramente virtual llamada `Draw`. Tanto `Circle` como `Square` son derivadas de la clase `Shape`.

Nota que si queremos extender el comportamiento de la función `DrawAllShapes` en la Lista 2 para dibujar un nuevo tipo de forma, todo lo que necesitamos hacer es agregar una nueva derivada de la clase `Shape`. La función `DrawAllShapes` no necesita cambiar. Por lo tanto, `DrawAllShapes` cumple con el principio abierto-cerrado. Su comportamiento puede extenderse sin modificarlo.

En el mundo real, la clase `Shape` 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.

Dado que los programas que cumplen con el principio abierto-cerrado se cambian agregando nuevo código, en lugar de modificar el código existente, no experimentan la cascada de cambios exhibida por los programas no conformes.

Cierre Estratégico.
Debería quedar claro que ningún programa significativo puede estar 100% cerrado. Por ejemplo, considera lo que sucedería con la función `DrawAllShapes` si decidimos que todos los `Circle` deben ser dibujados antes que cualquier `Square`. La función `DrawAllShapes` no está cerrada contra un cambio como este. En general, sin importar cuán "cerrado" esté un módulo, siempre habrá algún tipo de cambio contra el cual no esté 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 cierta capacidad de previsión derivada de la experiencia. El diseñador experimentado conoce a los usuarios y la industria lo suficientemente bien como para juzgar la probabilidad de diferentes tipos de cambios. Luego se asegura de que se invoque el principio de abierto-cerrado para los cambios más probables.

Usando la Abstracción para Ganar Cierre Explícito.
¿Cómo podríamos cerrar la función `DrawAllShapes` contra cambios en el orden de dibujo? Recuerda que el cierre se basa en la abstracción. Así, para cerrar `DrawAllShapes` contra el orden, necesitamos algún tipo de "abstracción de orden". El caso específico del orden mencionado se refería a 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 debe ser dibujado primero. Por lo tanto, podemos definir un método en la clase `Shape` llamado `Precedes` que toma otro `Shape` como argumento y devuelve un resultado booleano. El resultado es verdadero si el objeto `Shape` que recibe el mensaje debe ser ordenado antes que el objeto `Shape` pasado como argumento.

En C++, esta función podría representarse mediante una función sobrecargada `operator<`. Ahora que tenemos una forma de determinar el orden relativo de dos objetos `Shape`, podemos ordenarlos y luego dibujarlos en orden.
Esto nos da un medio para ordenar objetos `Shape` y para dibujarlos en el orden apropiado. Pero aún no tenemos una abstracción de ordenamiento adecuada. Tal como está, los objetos `Shape` individuales tendrán que sobrescribir el método `Precedes` para especificar el orden. ¿Cómo funcionaría esto? ¿Qué tipo de código escribiríamos en `Circle::Precedes` para asegurar que los `Circles` se dibujen antes que los `Squares`? Considera el Listado 5. Debería ser muy claro que esta función no cumple con el principio

de abierto-cerrado. No hay forma de cerrarla frente a nuevos derivados de `Shape`. Cada vez que se crea un nuevo derivado de `Shape`, esta función necesitará ser cambiada.

Usando un Enfoque "Impulsado por Datos" para Lograr Cierre.
El cierre de los derivados de `Shape` puede lograrse utilizando un enfoque basado en tablas que no obligue a realizar cambios en cada clase derivada.
Al adoptar este enfoque, hemos cerrado con éxito la función `DrawAllShapes` contra problemas de orden en general y cada uno de los derivados de `Shape` contra la creación de nuevos derivados de `Shape` o un cambio en la política que reordena los objetos `Shape` por su tipo. (Por ejemplo, cambiar el orden para que los `Squares` se dibujen primero).

El único elemento que no está cerrado contra el orden de las varias `Shapes` es la tabla misma. Y esa tabla 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.

Extendiendo el Cierre Aún Más.
Esto no es el final de la historia. Hemos logrado cerrar la jerarquía de `Shape` y la función `DrawAllShapes` contra un orden que depende del tipo de la forma. Sin embargo, los derivados de `Shape` no están cerrados contra políticas de orden que no tienen nada que ver con los tipos de forma. Parece probable que queramos ordenar el dibujo de las formas de acuerdo con alguna estructura de nivel superior. Una exploración completa de estos problemas está más allá del alcance de este artículo; sin embargo, el lector ambicioso podría considerar cómo abordar este problema utilizando una clase abstracta `OrderedObject` contenida en la clase `OrderedShape`, que se deriva tanto de `Shape` como de `OrderedObject`.

Heurísticas y Convenciones.
Como se mencionó al inicio de este artículo, el principio abierto-cerrado es la motivación principal detrás de muchas de las heurísticas y convenciones que se han publicado sobre OOD a lo largo de los años. Aquí están algunas de las más importantes:

Hacer todos los Miembros Privados.
Esta es una de las convenciones más comúnmente sostenidas de OOD. Las variables miembro de las clases deben ser conocidas sólo por los métodos de la clase que las definen. Las variables miembro nunca deben ser conocidas por ninguna otra clase, incluidas las clases derivadas. Por lo tanto, deben ser declaradas `private`, en lugar de `public` o `protected`.
En vista 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 ser cambiada. Por lo tanto, ninguna función que dependa de una variable puede estar cerrada respecto a esa variable.

En OOD, esperamos que los métodos de una clase no estén cerrados a cambios en las variables miembro de esa clase. Sin embargo, ***esperamos*** que cualquier otra clase, incluidas las subclases, estén cerradas contra cambios a esas variables. A esta expectativa la llamamos: *encapsulación*.

¿Qué pasaría si tuvieras una variable miembro que sabes que nunca cambiará? ¿Hay alguna razón para hacerla `privada`? Por ejemplo, la clase `Device` que tiene una variable `bool status` que contiene el estado de la última operación. Si esa operación tuvo éxito, entonces `status` será verdadero; de lo contrario, será falso.

Sabemos que el tipo o significado de esta variable nunca cambiará. Así que, ¿por qué no hacerla `pública` y dejar que el código cliente examine simplemente su contenido? Si realmente esta variable nunca cambia, y si todos los demás clientes obedecen las reglas y sólo consultan el contenido de `status`, entonces el hecho de que la variable sea pública no hace daño. Sin embargo, considera lo que sucede si incluso un cliente se aprovecha de la naturaleza escribible de `status` y cambia su valor. De repente, este cliente podría afectar a todos los demás clientes de `Device`. Esto significa que es imposible cerrar cualquier cliente de `Device` contra cambios de este módulo mal comportado. Esto probablemente representa un riesgo demasiado grande para asumir.

Por otro lado, supongamos que tenemos la clase `Time`. ¿Cuál es el daño que causan las variables miembro públicas en esta clase? Ciertamente, son muy poco probables de cambiar. Además, no importa si alguno de los módulos cliente realiza cambios en las variables, se supone que las variables deben ser cambiadas por los clientes. También es muy poco probable que una clase derivada quiera atrapar el establecimiento de una variable miembro particular. Entonces, ¿hay algún daño?

Una queja que podría hacer sobre esta clase es que la modificación de la hora no es atómica. Es decir, un cliente puede cambiar la variable `minutes` sin cambiar la variable `hours`. Esto puede resultar en valores inconsistentes para un objeto `Time`. Preferiría que hubiera una función única para establecer la hora que tomara tres argumentos, haciendo que el establecimiento de la hora sea atómico. Pero este es un argumento muy débil.

No sería difícil pensar en otras condiciones en las que la naturaleza `pública` de estas variables causa algunos problemas. A largo plazo, sin embargo, no hay una razón concluyente para hacer estas variables `privadas`. Aún considero que es un mal *estilo* hacerlas `públicas`, pero probablemente no es un mal *diseño*. Lo considero un mal estilo porque es muy barato crear las funciones miembro apropiadas en línea; y el costo casi seguro vale la pena la protección contra el ligero riesgo de que surjan problemas de cierre.

Así, en esos raros casos donde el principio abierto-cerrado no se viola, la proscripción de variables `públicas` y `protegidas` depende más del estilo que de la sustancia.

No Variables Globales - - Jamás.
El argumento contra las variables globales es similar al argumento contra las variables miembro `públicas`. Ningún módulo que dependa de una variable global puede estar cerrado contra cualquier otro módulo que pudiera escribir en esa variable. Cualquier módulo que use la variable de una manera que los otros módulos no esperan, romperá esos otros módulos. Es demasiado arriesgado que muchos módulos estén sujetos a la caprichosa naturaleza de uno mal comportado.

Por otro lado, en casos donde una variable global tiene muy pocos dependientes, o no puede usarse de manera inconsistente, hacen poco daño. El diseñador debe evaluar cuánto cierre se sacrifica por un global y determinar si la conveniencia ofrecida por el global vale el costo.

Nuevamente, hay cuestiones de estilo que entran en juego. Las alternativas al uso de globals suelen ser muy baratas. En esos casos, es de mal estilo utilizar una técnica que arriesga incluso una pequeña cantidad de cierre sobre una que no conlleva tal riesgo. Sin embargo, hay casos donde la conveniencia de un global es significativa. Las variables globales `cout` y `cin` son ejemplos comunes. En tales casos, si el principio abierto-cerrado no se viola, entonces la conveniencia puede valer la violación del estilo.

El RTTI es Peligroso.
Otra proscripción muy común es la que se refiere a `dynamic_cast`. A menudo se afirma que `dynamic_cast`, o cualquier forma de identificación de tipo en tiempo de ejecución (RTTI), es intrínsecamente peligrosa y debe ser evitada. El caso que a menudo se cita es similar a una implementación que claramente viola el principio abierto-cerrado. Sin embargo, una implementación que usa `dynamic_cast`, pero no viola el principio abierto-cerrado, muestra que su uso puede ser seguro.

La diferencia entre estos dos es que el primero, el listado 9, *debe* ser cambiado cada vez que se deriva un nuevo tipo de `Shape`. (Sin mencionar que es simplemente ridículo). Sin embargo, nada cambia en el Listado 10 cuando se crea un nuevo derivado de Shape. Por lo tanto, el Listado 10 no viola el principio de abierto-cerrado.
Como regla general, si un uso de RTTI no viola el principio de abierto-cerrado, es seguro.

Conclusión.
Se podría decir mucho más sobre el principio de abierto-cerrado. En muchos aspectos, este principio está en el corazón del diseño orientado a objetos. La conformidad con este principio es lo que ofrece los mayores beneficios que se reclaman para la tecnología orientada a objetos; es decir, la reutilización y la mantenibilidad. Sin embargo, la conformidad con este principio no se logra simplemente usando un lenguaje de programación orientado a objetos. Más bien, requiere una 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: *The Principles and Patterns of OOD*, que se publicará pronto 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 en relación con la implementación en C++. Estudiaremos el papel de las categorías de clases de Booch en C++, y su aplicabilidad como espacios de nombres en C++. Definiremos lo que significan “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.

LSP:El principio de sustitución de Liskov:

Esta es la segunda de mis columnas del Engineering Notebook para The C++ Report. Los artículos que aparecerán en esta columna se centrarán en el uso de C++ y el diseño orientado a objetos (OOD), y abordarán cuestiones de ingeniería de software. Me esforzaré por ofrecer artículos que sean pragmáticos y directamente útiles para el ingeniero de software en el terreno. En estos artículos, haré uso de la nueva notación unificada de Booch y Rumbaugh (Versión 0.8) para documentar diseños orientados a objetos. La barra lateral proporciona un breve léxico de esta notación.

Introducción.

Mi última columna (enero de 1996) trató sobre el principio de abierto-cerrado. Este principio es la base para construir código que sea mantenible y reutilizable. Establece que un código bien diseñado puede ser extendido sin modificaciones; que en un programa bien diseñado, las nuevas funciones se agregan mediante la adición de nuevo código, en lugar de modificar el código antiguo que ya funciona.

Los mecanismos primarios detrás del principio de abierto-cerrado son la abstracción y el polimorfismo. En lenguajes de tipado estático como C++, uno de los mecanismos clave que soporta 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 rigen este uso particular de la herencia? ¿Cuáles son las características de las mejores jerarquías de herencia? ¿Cuáles son las trampas que nos harán crear jerarquías que no se ajusten al principio de abierto-cerrado? Estas son las preguntas que abordará este artículo.

El principio de sustitución de Liskov.

LAS FUNCIONES QUE USAN PUNTEROS O REFERENCIAS A CLASES BASE DEBEN PODER USAR OBJETOS DE CLASES DERIVADAS SIN SABERLO.

Lo anterior es una paráfrasis del Principio de Sustitución de Liskov (LSP). Barbara Liskov lo escribió por primera vez de la siguiente manera hace casi 8 años:
Lo que se desea aquí es algo 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 hace obvia cuando consideras las consecuencias de violarlo. Si hay una función que no se ajusta al LSP, entonces esa función usa un puntero o referencia a una clase base, pero debe conocer todos los derivados de esa clase base. Tal función viola el principio de abierto-cerrado porque debe ser modificada cada vez que se crea un nuevo derivado de la clase base.

Un ejemplo sencillo de violación del LSP.

Una de las violaciones más evidentes de este principio es el uso de la Información de Tipo en Tiempo de Ejecución de C++ (RTTI) para seleccionar una función basada en el tipo de un objeto. Por ejemplo:

```
void DrawShape(const Shape& s)
{
if (typeid(s) == typeid(Square))
DrawSquare(static_cast<Square&>(s)); else if (typeid(s) ==
typeid(Circle))
DrawCircle(static_cast<Circle&>(s));
}
```

[Nota: `static_cast` es uno de los nuevos operadores de conversión. En este ejemplo funciona exactamente como una conversión regular. Por ejemplo: `DrawSquare((Square&)s);`. Sin embargo, la nueva sintaxis tiene reglas más estrictas que la hacen más segura de usar y es más fácil de localizar con herramientas como grep. Por lo tanto, se prefiere.]
Claramente, la función `DrawShape` está mal formada. Debe conocer cada posible derivado de la clase `Shape`, y debe ser cambiada cada vez que se crean nuevos derivados de `Shape`. De hecho, muchos ven la estructura de esta función como un anatema para el diseño orientado a objetos.

Square y Rectangle, una violación más sutil.
Sin embargo, hay otras formas, mucho más sutiles, de violar el LSP. Considera una aplicación que usa la clase `Rectangle` como se describe a continuación:

```
class Rectangle
{
public:
void SetWidth(double w) {itsWidth=w;}
void SetHeight(double h) {itsHeight=w;}
double GetHeight() const {return itsHeight;}
double GetWidth() const {return itsWidth;}
private:
double itsWidth;
double itsHeight;
};
```

Imagina que esta aplicación funciona bien y está instalada en muchos sitios. Como sucede con todo software exitoso, a medida que cambian las necesidades de sus usuarios, se necesitan nuevas funciones. Imagina que un día los usuarios demandan la capacidad de manipular cuadrados además de rectángulos.

A menudo se dice que, en C++, la herencia es la relación ISA. En otras palabras, si un nuevo tipo de objeto puede decirse que cumple con la relación ISA con un viejo tipo de objeto, entonces la clase del nuevo objeto debería derivarse de la clase del viejo objeto. Claramente, un cuadrado es un rectángulo para todos los efectos normales. Dado que se cumple la relación ISA, es lógico modelar la clase `Square` como derivada de `Rectangle`. (Véase la Figura 1.)

Este uso de la relación ISA es considerado por muchos como una de las técnicas fundamentales del Análisis Orientado a Objetos. Un cuadrado es un rectángulo, por lo que la clase `Square` debería derivarse de la clase `Rectangle`. Sin embargo, este tipo de pensamiento puede llevar a problemas sutiles, pero significativos. Generalmente, estos problemas no se prevén hasta que realmente intentamos codificar la aplicación.

Nuestra primera pista podría ser el hecho de que un `Square` no necesita tanto sus variables miembro `itsHeight` como `itsWidth`. Sin embargo, las heredará de todos modos. Claramente, esto es derrochador. Además, si vamos a crear cientos de miles de objetos `Square` (por ejemplo, en un programa CAD/CAE en el que cada pin de cada componente de un circuito complejo se dibuja como un cuadrado), este desperdicio podría ser extremadamente significativo.

Sin embargo, supongamos que no estamos muy preocupados por la eficiencia de la memoria. ¿Hay otros problemas? ¡De hecho! `Square` heredará las funciones `SetWidth` y `SetHeight`. Estas funciones son completamente inapropiadas para un `Square`, ya que el ancho y la altura de un cuadrado son idénticos. Esto debería ser una pista significativa de que hay un problema con el diseño. Sin embargo, hay una manera de eludir el problema. Podríamos sobrescribir `SetWidth` y `SetHeight` de la siguiente manera:

```
void Square::SetWidth(double w)
{
Rectangle::SetWidth(w);
Rectangle::SetHeight(w);
}
void Square::SetHeight(double h)
{
Rectangle::SetHeight(h);
Rectangle::SetWidth(h);
}
```

Ahora, cuando alguien establece el ancho de un objeto `Square`, su altura cambiará en consecuencia. Y cuando alguien establece su altura, el ancho cambiará también. Así, los invariantes del `Square` se mantienen intactos. El objeto `Square` seguirá siendo un cuadrado matemáticamente correcto.

```
Square s;
s.SetWidth(1); // Fortunately sets the height to 1 too.
s,SetHeight(2); // sets width and heigt to 2, good thing.
But consider the following function:
void f(Rectangle& r)
{
r.SetWidth(32); // calls Rectangle::SetWidth
}
```

Si pasamos una referencia a un objeto `Square` a esta función, el objeto `Square` se corromperá porque la altura no se cambiará. Esta es una clara violación del LSP. La función f no funciona para los derivados de sus argumentos. La razón del fallo es que `SetWidth` y `SetHeight` no se declararon como `virtuales` en `Rectangle`.

Podemos solucionar esto fácilmente. Sin embargo, cuando la creación de una clase derivada nos obliga a realizar cambios en la clase base, a menudo implica que el diseño es defectuoso. De hecho, viola el principio de abierto-cerrado. Podríamos contrarrestar esto con el argumento de que olvidar hacer `SetWidth` y `SetHeight virtuales` fue el verdadero defecto de diseño, y solo lo estamos corrigiendo ahora. Sin embargo, esto es difícil de justificar ya que establecer la altura y el ancho de un rectángulo son operaciones sumamente primitivas. ¿Con qué razonamiento las haríamos `virtuales` si no anticipábamos la existencia de `Square`?

Aún así, supongamos que aceptamos el argumento y corregimos las clases. Terminamos con el siguiente código:

```
class Rectangle
{
public:
virtual void SetWidth(double w) {itsWidth=w;}
virtual void SetHeight(double h) {itsHeight=h;}
double GetHeight() const {return itsHeight;}
double GetWidth() const {return itsWidth;}
private:
double itsHeight;
double itsWidth;
};
class Square : public Rectangle
{
public:
virtual void SetWidth(double w);
virtual void SetHeight(double h);
};
void Square::SetWidth(double w)
{
Rectangle::SetWidth(w);
Rectangle::SetHeight(w);
}
void Square::SetHeight(double h)
{
Rectangle::SetHeight(h);
Rectangle::SetWidth(h);
}
```

El problema real.

En este punto, tenemos dos clases, `Square` y `Rectangle`, que parecen funcionar. No importa lo que hagas con un objeto `Square`, seguirá siendo consistente 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 acepte un puntero o referencia a un `Rectangle`, y el `Square` seguirá comportándose como un cuadrado y se mantendrá consistente.

Por lo tanto, podríamos concluir que el modelo es ahora autoconsistente y correcto. Sin embargo, esta conclusión sería errónea. 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 verdadero problema: ¿Estaba justificado el programador que escribió esa función al asumir *que cambiar el ancho de un `Rectangle` no cambia 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 con objetos `Square`. Estas funciones exponen una violación del Principio de Sustitución de Liskov (LSP). La adición del derivado `Square` de `Rectangle` ha roto estas funciones; y, por lo tanto, se ha violado el principio de abierto/cerrado.

La validez no es intrínseca.
Esto nos lleva a una conclusión muy importante. Un modelo, visto en aislamiento, no puede validarse de manera significativa. La validez de un modelo solo puede expresarse en términos de sus clientes. Por ejemplo, cuando examinamos la versión final de las clases Square y Rectangle en aislamiento, 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 particular es adecuado o no, no se debe simplemente ver la solución en aislamiento. Debe verse en términos de las suposiciones razonables que harán los usuarios de ese diseño.

¿Qué salió mal?
Entonces, ¿qué sucedió? ¿Por qué el modelo aparentemente razonable de `Square` y `Rectangle` salió mal? Después de todo, ¿no es un `Square` un `Rectangle`? ¿No se cumple la relación "es un"?
¡No! Un cuadrado puede 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 cuadrado no es un rectángulo! Y el *comportamiento* es realmente de lo que trata el software.

El LSP deja claro que en el diseño orientado a objetos, la relación "es un" se refiere al *comportamiento*. No al comportamiento privado intrínseco, sino al comportamiento público extrínseco; el comportamiento del que dependen los clientes. Por ejemplo, el autor de la función g arriba dependía del hecho de que los `Rectangle` se comportaran de tal manera que su altura y ancho variaran independientemente el uno del otro. Esa independencia de las dos variables es un comportamiento público extrínseco del que es probable que dependan otros programadores.

Para que se cumpla el LSP, y con él el principio de abierto/cerrado, todos los derivados deben conformarse al comportamiento que los clientes esperan de las clases base que utilizan.

Diseño por contrato
Existe una relación estrecha entre el LSP y el concepto de Diseño por Contrato, tal como lo expone Bertrand Meyer. Usando 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:

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

En otras palabras, cuando se usa un objeto a través de la interfaz de su clase base, el usuario solo conoce las precondiciones y postcondiciones de la clase base. Así, los objetos derivados no deben esperar que tales usuarios cumplan con precondiciones que sean más fuertes que las requeridas por la clase base. Es decir, deben aceptar cualquier cosa que la clase base pudiera aceptar. Además, las clases derivadas deben cumplir con todas las postcondiciones de la base.

Claramente, la postcondición de `Square::SetWidth(double w)` es más débil que la postcondición `de Rectangle::SetWidth(double w)`, 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.
Ciertos lenguajes, como Eiffel, tienen soporte directo para precondiciones y postcondiciones. En C++, aunque no se puede declarar explícitamente, podemos considerar manualmente estas condiciones y asegurarnos de que no se violen. Además, es útil documentarlas en los comentarios de cada método.

Un Ejemplo Real.
¡Suficiente de cuadrados y rectángulos! ¿Tiene el Principio de Sustitución de Liskov (LSP) relevancia en el software real? Veamos un estudio de caso que proviene de un proyecto en el que trabajé hace algunos años.

Motivación.
No estaba satisfecho con las interfaces de las clases de contenedores que estaban disponibles a través de terceros. No quería que el código de mi aplicación dependiera de

forma crítica de estos contenedores, ya que preveía que querría reemplazarlos por clases mejores más adelante. Así que envolví los contenedores de terceros en mi propia interfaz abstracta. (Ver Figura 2)

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

Esta estructura unificaba las variedades de conjuntos delimitados y no delimitados de dos proveedores de terceros y permitía acceder a ellos a través de una interfaz común. Así, algún cliente podía aceptar un argumento de tipo `Set<T>&` sin importarle si el conjunto en el que trabajaba era del tipo `Bounded` o `Unbounded`. (Ver la función `PrintSet` en el listado).

La ventaja de poder ignorar el tipo exacto del conjunto es enorme. Esto significa que el programador puede decidir qué tipo de conjunto necesita en cada instancia particular. Ninguna de las funciones cliente se verá afectada por esa decisión. El programador puede elegir un `BoundedSet` cuando la memoria es limitada y la velocidad no es crítica, o un `UnboundedSet` cuando hay memoria disponible y la velocidad es prioritaria. Las funciones cliente manipularán estos objetos a través de la interfaz de la clase base `Set`, y por lo tanto no sabrán ni les importará qué tipo de conjunto están utilizando.

Problema.
Quería añadir un `PersistentSet` a esta jerarquía. Un `PersistentSet` es un conjunto que puede escribirse en un stream y luego leerse más tarde, posiblemente por una aplicación diferente. Desafortunadamente, el único contenedor de terceros que ofrecía persistencia no era una clase plantilla. En su lugar, aceptaba objetos derivados de la clase base abstracta `PersistentObject`.

Así creé la jerarquía mostrada en la Figura 3.
A primera vista, esto podría parecer correcto. Sin embargo, hay una implicación que es bastante fea. Cuando un cliente añade miembros al `Set`, ¿cómo se supone que debe asegurarse de que solo añade derivados de `PersistentObject` si el conjunto es un `PersistentSet`?
Considere el código para `PersistentSet::Add:`

```
template <class T>
void PersistentSet::Add(const T& t)
{
PersistentObject& p =
dynamic_cast<PersistentObject&>(t); // throw bad_cast
itsThirdPartyPersistentSet.Add(p);
}
```

Este código deja claro que si algún cliente intenta añadir un objeto que no derive de `PersistentObject` a mi `PersistentSet`, se producirá un error en tiempo de ejecución. El `dynamic_cast` lanzará un `bad_cast`, y ninguna de las funciones existentes de la clase base abstracta `Set` espera que se lancen excepciones en `Add`. Teniendo en

cuenta que las funciones van a ser confundidas por un derivado de `Set`, este cambio en la jerarquía viola el LSP.

¿Es esto un problema? Ciertamente. Las funciones que nunca antes fallaron al recibir un derivado de `Set`, ahora causarán errores en tiempo de ejecución al recibir un `PersistentSet`. Depurar este tipo de problema es relativamente difícil, ya que el error en tiempo de ejecución ocurre muy lejos del verdadero fallo lógico. El fallo lógico puede estar en la decisión de pasar un `PersistentSet` a la función que falla, o en la decisión de agregar un objeto al `PersistentSet` que no esté derivado de `PersistentObject`. En cualquier caso, la decisión real podría estar a millones de instrucciones de distancia de la invocación del método `Add`. Encontrarlo puede ser un reto. Arreglarlo puede ser aún peor.

Una Solución No Conforme al LSP.
¿Cómo resolvemos este problema? Hace varios años, lo resolví mediante una convención. Es decir, no lo solucioné en el código fuente. En lugar de eso, establecí una convención en la cual `PersistentSet` y `PersistentObject` no eran conocidos por toda la aplicación, sino solo por un módulo en particular. Este módulo era responsable de leer y escribir todos los contenedores. Cuando un contenedor necesitaba ser escrito, su contenido se copiaba en `PersistentObjects` y luego se agregaba a `PersistentSets`, los cuales se guardaban en un flujo. Cuando un contenedor necesitaba ser leído de un flujo, el proceso se invertía. Se leía un `PersistentSet` del flujo, luego los `PersistentObjects` se eliminaban del `PersistentSet` y se copiaban en objetos regulares (no persistentes), que luego se agregaban a un `Set` regular.

Esta solución puede parecer excesivamente restrictiva, pero fue la única forma en que pude pensar para evitar que los objetos PersistentSet aparecieran en la interfaz de funciones que quisieran agregarles objetos no persistentes. Además, rompía la dependencia del resto de la aplicación respecto a la noción completa de persistencia.

¿Funcionó esta solución? No realmente. La convención fue violada en varias partes de la aplicación por ingenieros que no entendían su necesidad. Ese es el problema con las convenciones: tienen que ser reintroducidas continuamente a cada ingeniero. Si el ingeniero no está de acuerdo, la convención se violará. Y una sola violación arruina toda la estructura.

Una Solución Conforme al LSP.
¿Cómo lo resolvería ahora? Reconocería que `PersistentSet` no tiene una relación ISA con `Set`; que no es un derivado adecuado de `Set`. Por lo tanto, separaría las jerarquías, pero no por completo. Hay características que `Set` y `PersistentSet` tienen en común.

De hecho, sólo el método `Add` es el que causa la dificultad con el Principio de Sustitución de Liskov (LSP). Así que crearía una jerarquía en la que tanto `Set` como `PersistentSet` fueran hermanos bajo una interfaz abstracta que permitiera pruebas de pertenencia, iteración, etc. (Ver Figura 4). Esto permitiría que los objetos `PersistentSet` se pudieran iterar y probar para verificar la pertenencia, pero no permitiría la adición de objetos que no fueran derivados de `PersistentObject` a un `PersistentSet`.

```
template <class T>
```

```
void PersistentSet::Add(const T& t)
{
itsThirdPartyPersistentSet.Add(t);
// This will generate a compiler error if t is
// not derived from PersistentObject.
}
```

Como muestra el listado anterior, cualquier intento de agregar un objeto que no esté derivado de `PersistentObject` a un `PersistentSet` resultará en un error de compilación. (La interfaz del conjunto persistente de terceros espera un `PersistentObject&`).

Conclusion.
El principio de Abierto-Cerrado está en el corazón de muchas de las afirmaciones hechas sobre el Diseño Orientado a Objetos (OOD). Cuando este principio está en efecto, 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 de Abierto-Cerrado. Solo cuando los tipos derivados son completamente sustituibles por sus tipos base, las funciones que usan esos tipos base pueden ser reutilizadas sin problemas, y los tipos derivados pueden ser modificados sin inconvenientes.

Este artículo es una versión extremadamente condensada de un capítulo de mi nuevo libro: Patrones y Principios Avanzados del Diseño Orientado a Objetos, que será publicado pronto 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 su implementación en C++. Analizaremos el rol de las categorías de clases de Booch en C++ y su aplicabilidad como espacios de nombres en C++. Definiremos lo que significan “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.

ISP: El principio de segregación de interfaces:

Este es el cuarto de mis artículos para la columna de Engineering Notebook en *The C++ Report*. Los artículos que aparecen en esta columna se enfocan en el uso de C++ y la programación orientada a objetos (OOD), y abordan temas relacionados con la ingeniería de software. Me esfuerzo por ofrecer artículos que sean pragmáticos y directamente útiles para el ingeniero de software que trabaja en el campo. En estos artículos, utilizó el nuevo Lenguaje Unificado de Modelado (UML Versión 0.8) de Booch y Rumbaugh para documentar diseños orientados a objetos. La barra lateral proporciona un breve léxico de esta notación.

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 los módulos que implementan detalles. En lugar de eso, ambos tipos de módulos deben depender de abstracciones. De manera más concisa 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 TEMPLATE METHOD del libro de GOF. En este patrón, un algoritmo de alto nivel se codifica en una clase base abstracta y hace uso de 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 trata las desventajas de las interfaces "gordas". Las clases que tienen interfaces "gordas" son aquellas cuyos interfaces no son cohesivos. En otras palabras, las interfaces de la clase pueden dividirse en grupos de funciones miembro. Cada grupo sirve a un conjunto diferente de clientes. Así, algunos clientes utilizan un grupo de funciones miembro, y otros clientes utilizan otros grupos.

El ISP reconoce que existen objetos que requieren interfaces no cohesivas; sin embargo, sugiere que los clientes no deberían conocerlas como una única clase. En su lugar, los clientes deberían conocer clases base abstractas que tienen 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 "gordas" o "contaminadas". Mostraremos cómo se crean estas interfaces y cómo diseñar clases que las oculten. Finalmente, presentaremos un estudio de caso en el que surge naturalmente una interfaz "gorda" y aplicaremos el ISP para corregirla.

Contaminación de Interfaces.
Consideremos un sistema de seguridad. En este sistema hay objetos `Door` que pueden bloquearse y desbloquearse, y que saben si están abiertas o cerradas. (Ver Listado 1).

Listado 1 Security Door

```
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 de Puerta, sin tener que depender de implementaciones particulares de Door.

Ahora consideremos una de esas implementaciones. `TimedDoor` necesita sonar una alarma cuando la puerta ha estado abierta por demasiado tiempo. Para hacer esto, el objeto `TimedDoor` se comunica con otro objeto llamado Timer. (Ver Listado 2).

Listado 2 Timer

```
class Timer
{
public:
void Regsiter(int timeout, TimerClient* client);
};
class TimerClient
{
public:
virtual void TimeOut() = 0;
};
```

Cuando un objeto desea ser notificado sobre un tiempo de espera, llama a la función Register del Timer. Los argumentos de esta función son el tiempo de espera y un puntero a un objeto TimerClient cuya función TimeOut será llamada cuando el tiempo de espera expire.

¿Cómo podemos hacer que la clase TimerClient se comunique con la clase TimedDoor para que el código en TimedDoor pueda ser notificado del tiempo de espera? Existen varias alternativas. La Figura 1 muestra una solución común. Obligamos 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.

Figura 1.

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 la temporización. Si se crean derivadas de Door que no necesitan temporización, esas derivadas tendrán que proporcionar implementaciones vacías para el método TimeOut. Además, las aplicaciones que utilicen esas derivadas tendrán que incluir la definición de la clase TimerClient, aunque no la usen.

La Figura 1 muestra un síndrome común en el diseño orientado a objetos en lenguajes de tipo estático como C++. Este es el síndrome de la contaminación de interfaces. La interfaz de Door ha sido contaminada con una interfaz que no requiere. Ha sido forzada a incorporar esta interfaz únicamente para beneficio de una de sus subclases. Si se sigue esta práctica, cada vez que una derivada necesite una nueva interfaz, esa interfaz se añadirá a la clase base. Esto contaminará aún más la interfaz de la clase base, haciéndola "gorda".

Además, cada vez que se añada una nueva interfaz a la clase base, esa interfaz deberá 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 tengan

que 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 que son utilizadas por clientes completamente diferentes. Timer usa TimerClient y las clases que manipulan puertas usan Door. Dado que los clientes son diferentes, las interfaces también deberían mantenerse separadas. ¿Por qué? Porque, como veremos en la siguiente sección, los clientes ejercen fuerzas sobre las interfaces de sus servidores.

La fuerza inversa aplicada por los clientes sobre las interfaces.
Cuando pensamos en 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 a todos los usuarios de TimerClient, si la interfaz de TimerClient cambiara. Sin embargo, hay una fuerza que opera en la dirección opuesta. 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. Consideremos el 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 ese tiempo de espera expire, la puerta se cierra; permanece cerrada un tiempo y luego se vuelve a abrir. Esto nos lleva a registrar una nueva solicitud de tiempo de espera antes de que la anterior haya expirado. Finalmente, la primera solicitud de tiempo de espera expira y se invoca la función TimeOut de TimedDoor. Y la puerta suena la alarma falsamente.

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

Listado 3 Timer con ID
```
class Timer
{
public:
void Regsiter(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 del timeOutId es un error que necesita corregirse. 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 *afectar* a los clientes de las derivadas de Door que no requieren temporización? Este tipo de interdependencia extraña es la que aterroriza a los clientes y gerentes. Cuando un cambio en una parte del programa afecta a otras partes completamente no relacionadas del programa, los costos y repercusiones de los cambios se vuelven impredecibles, y el riesgo de consecuencias aumenta dramáticamente.

Pero es solo una recompilación. Cierto. Pero las recompilaciones pueden ser muy costosas por varias razones. En primer lugar, toman tiempo. Cuando las recompilaciones toman demasiado tiempo, los desarrolladores comienzan a tomar atajos. Pueden hacer un cambio en el lugar "equivocado", en lugar de hacer el cambio en el lugar "correcto", porque el lugar "correcto" implicaría una recompilación masiva. En segundo lugar, una recompilación significa un nuevo módulo de objeto. En esta era de bibliotecas de enlaces dinámicos y cargadores incrementales, generar más módulos de objeto de los necesarios puede ser una desventaja significativa. Cuantos más DLL se vean afectados por un cambio, mayor será el problema de distribuir y gestionar dicho cambio.

El principio de segregación de interfaces.

LOS CLIENTES NO DEBEN SER FORZADOS A DEPENDER DE INTERFACES QUE NO UTILIZAN.

Cuando los clientes se ven obligados a depender de interfaces que no utilizan, dichos clientes están sujetos a los cambios de 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 usa, pero que otros clientes sí utilizan, ese cliente se verá afectado por los cambios que esos otros clientes impongan a la clase. Nos gustaría evitar tales acoplamientos cuando sea posible, por lo que queremos separar las interfaces donde sea posible.

Interfaces de Clase vs Interfaces de Objeto.
Consideremos nuevamente la TimedDoor. 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 radica en el hecho de que los clientes de un objeto no necesitan acceder a él a través de la interfaz del objeto. En su lugar, pueden acceder a él mediante delegación o a través de una clase base del objeto.

Separación mediante delegación.
Podemos emplear la forma de *objeto* del patrón ADAPTER para el problema de TimedDoor. La solución es crear un objeto adaptador que derive de TimerClient y delegue en TimedDoor. La figura 2 muestra esta solución.

Cuando TimedDoor quiere registrar una solicitud de tiempo de espera con el Timer, crea un DoorTimerAdapter y lo registra con el Timer. Cuando el Timer envía el mensaje de TimeOut al DoorTimerAdapter, este último 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 realizara el cambio en Timer mostrado en el listado 3, ninguno de los usuarios de Door se vería afectado. Además, TimedDoor no tiene que tener la misma interfaz exacta que TimerClient. El DoorTimerAdapter puede traducir la interfaz de TimerClient en la interfaz de TimedDoor. Por lo tanto, esta es una solución de propósito muy general. (Ver listado 4)

Listado 4 Forma de objeto del patrón adapter

```
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 poco elegante. Implica la creación de un nuevo objeto cada vez que deseamos registrar un tiempo de espera. Además, la delegación requiere una pequeña, aunque no nula, cantidad de tiempo de ejecución y memoria. Existen dominios de aplicaciones, como sistemas de control en tiempo real embebidos, en los que el tiempo de ejecución y la memoria son lo suficientemente escasos como para que esto sea una preocupación.

Separación mediante herencia múltiple.

La figura 3 y el listado 5 muestran cómo la herencia múltiple puede utilizarse, en la forma de clase del patrón ADAPTER, para cumplir con el ISP. En este modelo, TimedDoor hereda tanto de Door como de TimerClient. Aunque los clientes de ambas clases base pueden utilizar TimedDoor, ninguno depende realmente de la clase TimedDoor. Así, usan el mismo objeto a través de interfaces separadas.

Figura 3.

Listado 5 Forma de clase del patrón adapter

```
class TimedDoor : public Door, public TimerClient
{
public:
```

```
virtual void TimeOut(int timeOutId);
};
```

Esta solución es mi preferencia habitual. La herencia múltiple no me asusta. De hecho, la encuentro muy útil en casos como este. La única vez que elegiría la solución de la figura 2 sobre la de la figura 3 es si la traducción realizada por el objeto DoorTimerAdapter fuera necesaria, o si se necesitaran traducciones diferentes en distintos momentos.

El ejemplo de la Interfaz de Usuario del cajero automático (ATM)
Ahora consideremos un ejemplo ligeramente más significativo: el problema tradicional del Cajero Automático (ATM). La interfaz de usuario de una máquina ATM necesita ser muy flexible. La salida puede necesitar traducirse a muchos idiomas diferentes. Puede necesitar presentarse en una pantalla, en una tableta Braille o ser pronunciada por un sintetizador de voz. Claramente, esto puede lograrse creando una clase base abstracta que tenga funciones virtuales puras para todos los mensajes que necesitan ser presentados por la interfaz.

Figura 4.

Consideremos también que cada transacción diferente que puede realizar el ATM está encapsulada como un derivado de la clase Transaction. Así, podríamos tener clases como DepositTransaction, WithdrawlTransaction, TransferTransaction, etc. Cada uno de estos objetos emite mensajes a la UI. 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 de la figura 5.

Observemos que esta es precisamente la situación que el ISP nos dice que evitemos. Cada una de las transacciones está utilizando una parte de la UI que ningún otro objeto utiliza. Esto crea la posibilidad de que los cambios en uno de los derivados de Transaction obliguen a un cambio correspondiente en la UI, afectando así a todos los demás derivados de Transaction y a todas las demás clases que dependen de la interfaz UI.

Este acoplamiento desafortunado puede evitarse segregando la interfaz UI en clases base abstractas individuales como DepositUI, WithdrawUI y TransferUI. Estas clases base abstractas pueden heredarse de forma múltiple en la clase UI abstracta final. La figura 6 y el listado 6 muestran este modelo.

Es cierto que, cada vez que se crea un nuevo derivado de la clase Transaction, será necesaria una clase base correspondiente para la clase UI abstracta. Por lo tanto, la clase UI y todos sus derivados deben cambiar. Sin embargo, estas clases no son ampliamente utilizadas. De hecho, probablemente solo sean utilizadas por main, o por cualquier proceso que arranque el sistema y cree la instancia concreta de UI. Así que el impacto de agregar nuevas clases base de UI está contenido.

Figura 5.

Listado 6 Interfaces segregadas del cajero automático (ATM)

```
class DepositUI
{
public:
virtual void RequestDepositAmount() = 0;
};
class class DepositTransation : public Transaction
{
public:
DepositTransaction(DepositUI& ui)
: itsDepositUI(ui)
{}
virtual void Execute()
{
...
itsDepositUI.RequestDepositAmount();
...
}
private:
DepositUI& itsDepositUI;
};
class WithdrawlUI
{
public:
virtual void RequestWithdrawlAmount() = 0;
};
class class WithdrawlTransation : public Transaction
{
public:
WithdrawlTransaction(WithdrawlUI& ui)
: itsWithdrawlUI(ui)
{}
virtual void Execute()
{
...
itsWithdrawlUI.RequestWithdrawlAmount();
...
}
private:
WithdrawlUI& itsWithdrawlUI;
};
class TransferUI
{
public:
virtual void RequestTransferAmount() = 0;
};
class class TransferTransation : public Transaction
{
```

```
public:
TransferTransaction(TransferUI& ui)
: itsTransferUI(ui)
{}
virtual void Execute()
{
...
itsTransferUI.RequestTransferAmount();
...
}
private:
TransferUI& itsTransferUI;
};
class UI : public DepositUI,
, public WithdrawlUI,
, 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. Observemos que cada transacción debe conocer su versión particular de la UI. DepositTransaction debe conocer DepositUI; WithdrawTransaction debe conocer WithdrawUI, etc. En el listado 6, he abordado este problema forzando que cada transacción se construya con una referencia a su UI particular. Observemos que esto me permite emplear el siguiente patrón:

Listado 7 Patrón de inicialización de interfaz

```
UI Gui; // global object;
void f()
{
DepositTransaction dt(Gui);
}
```

Esto es conveniente, pero también obliga a que cada transacción contenga un miembro de referencia a su UI. Otra forma de abordar este problema es crear un conjunto de constantes globales como se muestra en el listado 8. Como descubrimos al discutir el Principio de Abierto/Cerrado en la edición de enero del 96, las variables globales no siempre son un síntoma de un mal diseño. En este caso, proporcionan la clara ventaja de un acceso fácil. Y como son referencias, es imposible cambiarlas de ninguna manera, por lo tanto, no pueden ser manipuladas de una manera que sorprenda a otros usuarios.

Listado 8 Punteros globales separados

```
// in some module that gets linked in
// to the rest of the app.
```

```
static UI Lui; // non-global object;
DepositUI& GdepositUI = Lui;
WithdrawlUI& GwithdrawlUI = Lui;
TransferUI& GtransferUI = Lui;
// In the depositTransaction.h module
class class WithdrawlTransation : public Transaction
{
public:
virtual void Execute()
{
...
GwithdrawlUI.RequestWithdrawlAmount();
...
}
};
```

Uno podría sentirse tentado a poner todas las variables globales del listado 8 en una sola clase para evitar la contaminación del espacio de nombres global. El listado 9 muestra este enfoque. Sin embargo, esto tiene un efecto desafortunado. Para usar UIGlobals, se debe hacer #include ui_globals.h.
Esto, a su vez, hace #include depositUI.h, withdrawUI.h, y transferUI.h. Esto significa que cualquier módulo que desee usar alguna de las interfaces de UI depende de todas ellas de manera transitiva; exactamente la situación que el ISP nos advierte evitar. Si se realiza un cambio en cualquiera de las interfaces de UI, todos los módulos que hagan #include ui_globals.h se ven obligados a recompilar. La clase UIGlobals ha vuelto a combinar las interfaces que nos habíamos esforzado tanto en segregar.

Listado 9 Encapsulando las variables globales en una clase

```
// in ui_globals.h
#include “depositUI.h”
#include “withdrawlUI.h”
#include “transferUI.h”
class UIGlobals
{
public:
static WithdrawlUI& withdrawl;
static DepositUI& deposit;
static TransferUI& transfer
};
// in ui_globals.cc
static UI Lui; // non-global object;
DepositUI& UIGlobals::deposit = Lui;
WithdrawlUI& UIGlobals::withdrawl = Lui;
TransferUI& UIGlobals::transfer = Lui;
```

El Poliádico vs. el Monádico.
Considera una función g que necesita acceso tanto a DepositUI como a TransferUI. Considera también que deseamos pasar las UIs a esta función. ¿Deberíamos escribir el

prototipo de la función así: `void g(DepositUI&, TransferUI&);`? ¿O deberíamos escribirlo así: `void g(UI&);`?

La tentación de escribir la última forma (monádica) es fuerte. Después de todo, sabemos que en la primera forma (poliádica), ambos argumentos se referirán al mismo objeto. Además, si usáramos la forma poliádica, su invocación sería algo así: g(ui, ui); De alguna manera, esto parece perverso.

Perverso o no, la forma poliádica es preferible a la forma monádica. La forma monádica obliga a que “g” dependa de todas las interfaces incluidas en UI. Así, cuando WithdrawUI cambia, “g” y todos los clientes de g tendrían que recompilarse. ¡Esto es más perverso que g(ui, ui);! Además, no podemos estar seguros de que ambos argumentos de “g” siempre se referirán al mismo objeto. En el futuro, podría ser que los objetos de la interfaz se separen por alguna razón. Desde el punto de vista de la función “g”, el hecho de que todas las interfaces estén combinadas en un solo objeto es información que “g” no necesita saber. Por lo tanto, prefiero la forma poliádica para tales funciones.

Conclusión.
En este artículo hemos discutido las desventajas de las "interfaces gordas"; es decir, interfaces que no son específicas para un solo cliente. Las interfaces gordas conducen a acoplamientos inadvertidos entre clientes que de otro modo deberían estar aislados. Haciendo uso del patrón ADAPTER, ya sea mediante delegación (forma de objeto) o herencia múltiple (forma de clase), las interfaces gordas pueden segregarse en clases base abstractas que rompan los acoplamientos no deseados entre los clientes.

Este artículo es una versión extremadamente condensada de un capítulo de mi nuevo libro: *Patrones y Principios Avanzados de Diseño Orientado a Objetos*, que será publicado pronto 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 su implementación en C++. Estudiaremos el papel de las categorías de clase de Booch en C++, y su aplicabilidad como espacios de nombres en C++. Definiremos qué significan "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.

DIP: El principio de inversión de dependencias:

Esta es la tercera de mis columnas del Engineering Notebook para The C++ Report. Los artículos que aparecerán en esta columna se centrarán en el uso de C++ y OOD, y abordarán cuestiones de ingeniería de software. Me esforzaré por obtener artículos que sean pragmáticos y directamente útiles para el ingeniero de software en las trincheras. En estos artículos haré uso de la nueva notación unificada de Booch y Rumbaugh (Versión 0.8) para documentar diseños orientados a objetos. La barra lateral proporciona un breve léxico de esta notación.

Introducción.

Mi último artículo (marzo del 96) hablaba del principio de sustitución de Liskov (LSP). Este principio, cuando se aplica a C++, proporciona una guía para el uso de la herencia pública. Establece que toda función que opera sobre una referencia o puntero a una clase base, debería poder operar sobre derivadas de esa clase base sin saberlo. Esto significa que las funciones miembro virtuales de las clases derivadas no deben esperar más que las funciones miembro correspondientes de la clase base; y no deberían prometer menos. También significa que las funciones miembro virtuales que están presentes en las clases base también deben estar presentes en las clases derivadas; y deben hacer un trabajo útil. Cuando se viola este principio, las funciones que operan sobre punteros o referencias a clases base necesitarán verificar el tipo de objeto real para asegurarse de que puedan operar sobre él correctamente. Esta necesidad de verificar el tipo viola el Principio Abierto-Cerrado (OCP) que discutimos en enero pasado.

En esta columna, discutimos las implicaciones estructurales del OCP y el LSP. La estructura que resulta del uso riguroso de estos principios puede generalizarse en un principio por sí solo. Yo lo llamo "El principio de inversión de dependencia" (DIP).

¿Qué es lo que está mal con el software?
La mayoría de nosotros hemos tenido la desagradable experiencia de tratar de lidiar con un software que tiene un “mal diseño”. Algunos de nosotros incluso hemos tenido la experiencia mucho más desagradable de descubrir que fuimos los autores del software con el “mal diseño”. ¿Qué es lo que hace que un diseño sea malo? La mayoría de los ingenieros de software no se proponen crear "malos diseños". Sin embargo, la mayoría del software eventualmente se degrada hasta el punto en que alguien declarará que el diseño no es sólido. ¿Por qué pasó esto? ¿Fue el diseño deficiente para empezar, o el diseño realmente se degradó como un trozo de carne podrida? En el centro de este problema está nuestra falta de una buena definición de trabajo de "mal" diseño.

La Definición de un “Diseño Malo”
¿Alguna vez ha presentado un diseño de software del que estaba especialmente orgulloso para que lo revisara un compañero? ¿Dijo ese compañero, en una mueca burlona y burlona, algo como: "¿Por qué lo hiciste de esa manera?". Ciertamente esto me ha pasado a mí, y lo he visto pasar a muchos otros ingenieros también. Claramente, los ingenieros en desacuerdo no están usando los mismos criterios para definir qué es un “mal diseño”. El criterio más común que he visto utilizado es el criterio TNTWIWHDI o "Esa no es la forma en que lo hubiera hecho".

Pero hay un conjunto de criterios con los que creo que todos los ingenieros estarán de acuerdo. Una pieza de software que cumpla con todos sus requisitos y, sin embargo, exhiba alguno o todos de los siguientes tres rasgos, tiene un mal diseño:

Es difícil cambiar porque cada cambio afecta a muchas otras partes del sistema. (Rigidez)
Cuando realiza un cambio, partes inesperadas del sistema se rompen. (Fragilidad)
Es difícil de reutilizar en otra aplicación porque no se puede desenredar de la aplicación actual. (Inmovilidad)
Además, sería difícil demostrar que un software que no presenta ninguno de esos rasgos, es decir, es flexible, robusto y reutilizable, y que además cumple con todos sus requisitos,

tenga un mal diseño. Por lo tanto, podemos utilizar estos tres rasgos como una forma de decidir sin ambigüedades si un diseño es "bueno" o "malo".

La causa del "mal diseño".
¿Qué es lo que 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 a un software fuertemente interdependiente inicia una cascada de cambios en los módulos dependientes. Cuando los diseñadores o mantenedores no pueden predecir el alcance de esa cascada de cambios, no se puede estimar el impacto del cambio.

Esto hace que el costo del cambio sea imposible de predecir. Los gerentes, ante tal imprevisibilidad, se vuelven reacios a autorizar cambios. Así, 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 una relación conceptual con el área que se cambió. Tal fragilidad disminuye en gran medida la credibilidad de la organización de diseño y mantenimiento. Los usuarios y gerentes no pueden predecir la calidad de su producto. Los cambios simples en una parte de la aplicación provocan fallas en otras partes que parecen no tener ninguna relación. Arreglar esos problemas conduce a aún más problemas, y el proceso de mantenimiento comienza a parecerse a un perro persiguiendo su cola.

Un diseño es inmóvil cuando las partes deseables del diseño dependen en gran medida de otros detalles que no se desean. Los diseñadores encargados de investigar el diseño para ver si se puede reutilizar en una aplicación diferente pueden quedar impresionados con lo bien que funcionaría el diseño en la nueva aplicación. Sin embargo, si el diseño es altamente interdependiente, esos diseñadores también se sentirán intimidados por la cantidad de trabajo necesario para separar la parte deseable del diseño de las otras partes del diseño 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 más alto que el costo de remodelación del diseño.

Ejemplo: El programa de Copiado.
Un ejemplo sencillo puede ayudar a aclarar este punto. Considere un programa simple que se encarga de la tarea de copiar caracteres escritos en un teclado a una impresora. Suponga, además, que la plataforma de implementación no tiene un sistema operativo que admita la independencia del dispositivo. Podríamos concebir una estructura para este programa que se parezca a la Figura 1.

La Figura 1 es un "diagrama de estructura". Muestra que hay tres módulos o subprogramas en la aplicación. El módulo "Copiar" (Copy) llama a los otros dos. Uno puede imaginarse fácilmente un bucle dentro del módulo "Copiar". (Ver Listado 1). El cuerpo de ese bucle llama al módulo “Leer desde Teclado” (Read Keyboard) para buscar un carácter del teclado, luego envía ese carácter al módulo “Escribir en Impresora” (Write Printer) que imprime el carácter.

Los dos módulos de bajo nivel son muy reutilizables. Se pueden utilizar 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.

Listado 1 The copy program.

```
void Copy()
{
int c;
while ((c = ReadKeyboard()) != EOF)
WritePrinter(c);
}
```

Sin embargo, el módulo "Copy" no es reutilizable en ningún contexto que no implique un teclado o una impresora. Es una pena, ya que en este módulo se mantiene la inteligencia del sistema. Es el módulo "Copy" el que encapsula una política muy interesante que nos gustaría reutilizar.

Por ejemplo, considere un nuevo programa que copia los caracteres del teclado en un archivo de 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 sumidero. Desafortunadamente, el módulo "Copy" depende del módulo "Write Printer", por lo que no se puede reutilizar en el nuevo contexto.

Sin duda, podríamos 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" dependiendo de algún tipo de marca. 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 plagado de declaraciones if/else y dependerá de muchos módulos de nivel inferior. Eventualmente se volverá rígido y frágil.

Listado 2. The “enhanced” copy program.

```
enum OutputDevice {printer, disk};
void Copy(outputDevice dev)
{
int c;
while ((c = ReadKeyboard()) != EOF)
if (dev == printer)
WritePrinter(c);
else
WriteDisk(c);
}
```

Inversión de dependencia.

Una forma de caracterizar el problema anterior es notar 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, WritePrinter() y ReadKeyboard()). Si pudiéramos encontrar una

manera de hacer que el módulo Copy() sea independiente de los detalles que controla, entonces podríamos reutilizarlo libremente.

Podríamos producir otros programas que usaran este módulo para copiar caracteres desde cualquier dispositivo de entrada a cualquier dispositivo de salida. La orientación a objetos (OOD) nos da un mecanismo para realizar esta inversión de dependencia.

Considere el simple diagrama de clases en la Figura 2. Aquí tenemos una clase "Copy" que contiene una clase abstracta de "Reader" y una clase abstracta de "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" (Lector de teclado) ni del "Printer Writer" (Escritor de Impresora). En consecuencia, las dependencias se han invertido; la clase "Copy" depende de abstracciones, y los lectores (readers) y escritores (writers) detallados (concretos) dependen de las mismas abstracciones.

Listado 3: The OO Copy Program

```
class Reader
{
public:
virtual int Read() = 0;
};
class Writer
{
public:
virtual void Write(char) = 0;
};
void Copy(Reader& r, Writer& w)
{
int c;
while((c=r.Read()) != EOF)
w.Write(c);
}
```

Ahora podemos reutilizar la clase "Copy", independientemente del "Keyboard Reader" y el "Printer Writer". Podemos inventar nuevos tipos derivados (concretos) de "Reader" y "Writer" (abstractos), que podemos suministrar a la clase "Copy" (esto es inyección de dependencias).

Además, no importa cuántos tipos de "lectores" (readers) y "escritores" (writers) se creen, la clase "Copy" no dependerá de ninguno de ellos. No habrá interdependencias que hagan que el programa sea frágil o rígido. Y Copy() en sí mismo se puede utilizar en muchos contextos detallados diferentes. Esto es modular.

Independencia de Dispositivo.
A estas alturas, algunos de ustedes probablemente se estarán diciendo a sí mismos que podrían obtener los mismos beneficios escribiendo Copy() en C, usando la independencia de dispositivo inherente a stdio.h; es decir, getchar y putchar (Ver Listado 4). Si considera

los “Listado 3 y 4” cuidadosamente, se dará cuenta de que los dos son lógicamente equivalentes. Las clases abstractas en la Figura 2 han sido reemplazadas por un tipo diferente de abstracción en el Listado 4. Es cierto que el Listado 4 no usa clases y funciones virtuales puras, sin embargo, todavía usa abstracción y polimorfismo para lograr sus fines. Además, ¡todavía usa inversión de dependencia! El programa Copy del Listado 4 no depende de ninguno de los detalles que controla. Más bien depende de las comodidades abstractas declaradas en stdio.h. Además, los controladores IO (entrada y salida) que finalmente se invocan también dependen de las abstracciones declaradas en stdio.h. Por lo tanto, la independencia del dispositivo dentro de la biblioteca stdio.h es otro ejemplo de inversión de dependencia.

Listado 4: Copy using stdio.h

```
#include <stdio.h>
void Copy()
{
int c;
while((c = getchar()) != EOF)
putchar(c);
}
```

Ahora que hemos visto algunos ejemplos, podemos establecer la forma general del DIP (Principio de Inversión de Dependencias).

The Dependency Inversion Principle (El Principio de Inversión de Dependencias).

A. High level modules should not depend upon low level modules. Both should depend upon abstractions.

(Módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de las abstracciones.)

B. Abstractions should not depend upon details. Details should depend upon abstractions.

(Las abstracciones no deben depender de los detalles. Los detalles deben depender de abstracciones.)

Uno podría preguntarse por qué utilizo la palabra "inversión". Francamente, se debe a que los métodos de desarrollo de software más tradicionales, como el Análisis y Diseño Estructurados, tienden a crear estructuras de software en las que los módulos de alto nivel dependen de 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 realizan llamadas a los módulos de bajo nivel. La figura 1 es un buen ejemplo de esta jerarquía. Por lo tanto, la estructura de dependencia de un programa orientado a objetos bien diseñado se "invierte" con respecto a la estructura de dependencia que normalmente resulta de los métodos de procedimiento tradicionales.

Considere 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 políticas

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 puede obligarlos a cambiar.

¡Esta situación es absurda! Son los módulos de alto nivel los que deberían obligar a los módulos de bajo nivel a cambiar. Son los módulos de alto nivel los que deben tener prioridad sobre los módulos de nivel inferior. 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 módulos de bajo nivel, se vuelve muy difícil reutilizar esos módulos de alto nivel en diferentes contextos. Sin embargo, cuando los módulos de alto nivel son independientes de los módulos de bajo nivel, los módulos de alto nivel se pueden reutilizar de forma bastante sencilla.

Este es el principio que está en el corazón del diseño de frameworks.

Layering (División en capas).
Según Booch, "... todas las arquitecturas orientadas a objetos bien estructuradas tienen capas claramente definidas, y cada capa proporciona 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 Figura 3.

En este diagrama, la clase Policy Layer (Capa de Políticas) de alto nivel utiliza una Mechanism Layer (Capa de Mecanismos) de nivel inferior, que a su vez utiliza una clase Utility Layer (Capa de Utilidades) de nivel detallado (concreto). Si bien esto puede parecer apropiado, tiene la característica insidiosa de que Policy Layer es sensible a los cambios en toda la Utility Layer. La dependencia es transitiva. Policy Layer depende de algo que depende de Utility Layer, por lo que Policy Layer depende transitivamente de Utility Layer. Esto es muy lamentable.

La figura 4 muestra un modelo más apropiado. Cada una de las capas de nivel inferior está representada por una clase abstracta. Las capas concretas se derivan entonces de estas clases abstractas. Cada una de las clases de nivel superior utiliza la siguiente capa más baja a través de la interfaz abstracta. Por tanto, ninguna de las capas (concretas) depende de ninguna de las otras capas (concretas). En cambio, las capas dependen de clases abstractas. No solo se rompe la dependencia transitiva de Policy Layer sobre Utility Layer, sino que incluso se rompe la dependencia directa de Policy Layer sobre Mechanism Layer.

Un ejemplo simple.
Con este modelo, la clase Policy Layer no se ve afectada por ningún cambio en Mechanism Layer o en Utility Layer. Además, Policy Layer se puede reutilizar en cualquier contexto que defina módulos de nivel inferior que se ajusten a la interfaz de Mechanism Layer. Así, al invertir las dependencias, hemos creado una estructura que es a la vez más flexible, duradera y móvil.

Separar la interfaz de la implementación en C++.
Uno podría quejarse de que la estructura de la Figura 3 no presenta los problemas de dependencia y de dependencia transitiva que afirmé. Después de todo, Policy Layer depende únicamente de la interfaz de Mechanism Layer. ¿Por qué un cambio en la implementación de Mechanism Layer tendría algún efecto en Policy Layer?

En algunos lenguajes orientados a objetos, esto sería cierto. En tales lenguajes, la interfaz se separa de la implementación automáticamente. Sin embargo, en C++, no hay separación entre interfaz e implementación. Más bien, en C++, la separación es 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 (o .cpp para nosotros). 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 una simple interfaz. Todas las funciones de utilidad y variables privadas que necesita 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 del que deben depender todos los usuarios de la clase. Por lo tanto, en C++, la implementación no se separa automáticamente de la interfaz.

Esta falta de separación entre la interfaz y la implementación en C++ puede tratarse utilizando clases puramente abstractas. Una clase puramente abstracta es una clase que no 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 de la Figura 4 están destinadas a ser puramente abstractas, de modo que cada una de las capas depende únicamente de la interfaz de la capa siguiente.

Un ejemplo simple.
La inversión de dependencia se puede aplicar siempre que una clase envíe un mensaje a otra. Por ejemplo, considere el caso del objeto Button (Botón) y el objeto Lamp (Lámpara). El objeto Button 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 (Graphical User Interface), un botón físico que se presiona con un dedo humano o incluso un detector de movimiento en un sistema de seguridad del hogar. El objeto Button detecta que un usuario lo ha activado o desactivado. El objeto Lamp afecta el entorno externo. Al recibir un mensaje TurnOn, enciende una luz de algún tipo. Al recibir un mensaje TurnOff, apaga esa luz. El mecanismo físico no es importante. Podría ser un LED en la consola de una computadora, una lámpara de vapor de mercurio en un estacionamiento o incluso el láser en una impresora.

¿Cómo podemos diseñar un sistema de modo que el objeto Button controle el objeto Lamp? La Figura 5 muestra un modelo ingenuo.

El objeto Button simplemente envía los mensajes TurnOn y TurnOff a la lámpara. Para facilitar esto, la clase Button usa una relación "contiene" para contener una instancia de la clase Lamp.

El Listado 5 (Listing 5) muestra el código C++ que resulta de este modelo. Tenga en cuenta que la clase Button depende directamente de la clase Lamp. De hecho, el módulo button.cc incluye (#include) el módulo lamp.h. Esta dependencia implica que la clase del botón debe cambiar, o al menos volver a compilarse, siempre que cambie la clase Lamp. Además, no será posible reutilizar la clase Button para controlar un objeto Motor, por ejemplo.

Listado 5: Naive Button/Lamp Code

```
--------------lamp.h----------------
class Lamp
{
public:
void TurnOn();
void TurnOff();
};
-------------button.h---------------
class Lamp;
class Button
{
public:
Button(Lamp& l) : itsLamp(&l) {}
void Detect();
private:
Lamp* itsLamp;
};
-------------button.cc--------------
#include "button.h"
#include "lamp.h"
void Button::Detect()
{
bool buttonOn = GetPhysicalState();
if (buttonOn)
itsLamp->TurnOn();
else
itsLamp->TurnOff();
}
```

La Figura 5 y el Listado 5 violan el principio de inversión de dependencia. 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.
¿Qué es la política de alto nivel? Son las abstracciones las que subyacen a la aplicación, las verdades que no varían cuando se cambian los detalles. En el ejemplo de Button/Lamp, la abstracción subyacente es detectar un gesto (evento) de encendido/apagado de un usuario y transmitir ese gesto a un objeto de destino. ¿Qué mecanismo se utiliza para detectar el gesto del usuario? ¡Irrelevante! ¿Cuál es el objeto de destino? ¡Irrelevante! Estos

son detalles que no afectan la abstracción. Para ajustarse al principio de inversión de dependencia, debemos aislar esta abstracción de los detalles del problema. Entonces debemos dirigir las dependencias del diseño de manera que los detalles dependan de las abstracciones. La Figura 6 muestra tal diseño.

En la Figura 6, hemos aislado la abstracción de la clase Button, de su implementación detallada. El Listado 6 muestra el código correspondiente. Tenga en cuenta que la política de alto nivel se captura por completo dentro de la clase de botón abstracto. La clase Button no sabe nada del mecanismo físico para detectar los gestos del usuario; y no sabe nada de la lámpara. Estos detalles se aíslan dentro de los derivados concretos: Implementación de Button y de Lamp. 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 afectado por cambios en los mecanismos de bajo nivel. Por lo tanto, es robusto en presencia de cambios, flexible y reutilizable.

Listado 6: Inverted Button Model

```
----------byttonClient.h---------
class ButtonClient
{
public:
virtual void TurnOn() = 0;
virtual void TurnOff() = 0;
};
-----------button.h---------------
class ButtonClient;
class Button
{
public:
Button(ButtonClient&);
void Detect();
virtual bool GetState() = 0;
private:
ButtonClient* itsClient;
};
---------button.cc----------------
#include button.h
#include buttonClient.h
Button::Button(ButtonClient& bc)
: itsClient(&bc) {}

void Button::Detect()
{
bool buttonOn = GetState();
if (buttonOn)
itsClient->TurnOn();
else
itsClient->TurnOff();
}
```

```
----------lamp.h----------------
class Lamp : public ButtonClient
{
public:
virtual void TurnOn();
virtual void TurnOff();
};
---------buttonImp.h-------------
class ButtonImplementation
: public Button
{
public:
ButtonImplementaton(
ButtonClient&);
virtual bool GetState();
};
```

Extendiendo más la abstracción.
Alguna vez se podría presentar una queja legítima sobre el diseño en la Figura/Listado 6. El dispositivo controlado por el botón debe derivarse de ButtonClient. ¿Qué pasa si la clase Lamp proviene de una biblioteca de terceros y no podemos modificar el código fuente?

La Figura 7 muestra cómo se puede utilizar el patrón de diseño Adaptador (Adapter) para conectar un objeto Lamp de terceros al modelo. La clase LampAdapter simplemente traduce el mensaje TurnOn y TurnOff heredado de ButtonClient, en cualquier mensaje que la clase Lamp necesite ver.

Conclusión.
El principio de inversión de dependencias está en la raíz de muchos de los beneficios que se atribuyen a la tecnología orientada a objetos. Su correcta aplicación es necesaria para la creación de frameworks reutilizables. También es de vital importancia para la construcción de un código resistente al cambio. Y, dado que las abstracciones y los detalles están aislados unos de otros, el código es mucho más fácil de mantener.

Este artículo es una versión extremadamente condensada de un capítulo de mi nuevo libro: Patrones y principios avanzados de OOD, que pronto publicará 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++. Estudiaremos el papel de las categorías de clases de Booch en C++ y su aplicabilidad como espacios de nombres de C++. Definiremos qué significan "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.