

Alumno: Pardo Sotelo, Daniel

Eric Gustavo Coronel Castillo

Análisis y diseño o.o.

Principios s.o.l.i.d.

Principios s.o.l.i.d.

Análisis y diseño o.o.

# principios básicos de diseño de clases

Son cinco principios fundamentales, uno por cada letra, que hablan del diseño orientado a objetos en términos de la gestión de dependencias. Las dependencias entre unas clases y otras son las que hacen al código más frágil o más robusto y reutilizable. El problema con el modelado tradicional es que no se ocupa en profundidad de la gestión de dependencias entre clases sino de la conceptualización. Quién decidió resaltar estos principios y darles nombre a algunos de ellos fue Robert C. Martin, allá por el año 1995.

# Single Responsibility Principle (SRP)

El principio que da origen a la S de S.O.L.I.D es el de una única responsabilidad y dice que cada clase debe ocuparse de un solo menester. Visto de otro modo, R. Martin dice que cada clase debería tener un único motivo para ser modificada.

Si estamos delante de una clase que se podría ver obligada a cambiar ante una modificación en la base de datos y a la vez, ante un cambio en el proceso de negocio, podemos afirmar que dicha clase tiene más de una responsabilidad o más de un motivo para cambiar, por poner un ejemplo.

Se aplica tanto a la clase como a cada uno de sus métodos, con lo que cada método también debería tener un solo motivo para cambiar. El efecto que produce este principio son clases con nombres muy descriptivos y por tanto largos, que tienen menos de cinco métodos, cada uno también con nombres que sirven perfectamente de documentación, es decir, de varias palabras: CalcularAreaRectangulo y que no contienen más de 15 líneas de código.

En la práctica la mayoría de mis clases tienen uno o dos métodos nada más. Este principio es quizás el más importante de todos, el más sencillo y a la vez el más complicado de llevar a cabo.

Existen ejemplos de código y una explicación más detallada del mismo en la propia web del autor. Martin también habla en profundidad sobre SRP en su libro. Hay una antigua técnica llamada Responsibility Driven Design (RDD), que viene a decir lo mismo que este principio, aunque es anterior a la aparición de SRP como tal. TDD es una excelente manera de hacer RDD o de seguir el SRP, como se quiera ver.

Allá por el año 1989, Kent Beck y Ward Cunningham usaban tarjetas CRC (Class, Responsibility, Collaboration) como ayuda para detectar responsabilidades y colaboraciones entre clases. Cada tarjeta es para una entidad, no necesariamente una clase. Desde que disponemos de herramientas que nos permiten el desarrollo dirigido por tests,las tarjetas CRC han pasado a un segundo plano pero puede ser buena idea usarlas parcialmente para casos donde no terminamos de ver claras las responsabilidades.

# Open-Closed Principle (OCP)

Una entidad software (una clase, módulo o función) debe estar abierta a extensiones pero cerrada a modificaciones. Puesto que el software requiere cambios y que unas entidades dependen de otras, las modificaciones en el código de una de ellas puede generar indeseables efectos colaterales en cascada.

Para evitarlo, el principio dice que el comportamiento de una entidad debe poder ser alterado sin tener que modificar su propio código fuente. ¿Cómo se hace esto?, Hay varias técnicas dependiendo del diseño, una podría ser mediante herencia y redefinición de los métodos de la clase padre, donde dicha clase padre podría incluso ser abstracta. La otra podría ser inyectando dependencias que cumplen el mismo contrato (que tienen la misma interfaz) pero que implementan diferente funcionamiento.

En próximos párrafos estudiaremos la inyección de dependencias. Como la totalidad del código no se puede ni se debe cerrar a cambios, el diseñador debe decidir contra cuáles protegerse mediante este principio. Su aplicación requiere bastante experiencia, no sólo por la dificultad de crear entidades de comportamiento extensible sino por el peligro que conlleva cerrar determinadas entidades o parte de ellas.

Cerrar en exceso obliga aescribir demasiadas líneas de código a la hora de reutilizar la entidad en cuestión. El nombre de Open-Closed se lo debemos a Bertrand Meyer y data del año 1988. En español podemos denominarlo el principio Abierto-Cerrado.

# Liskov Substitution Principle (LSP)

Introducido por Barbara Liskov en 1987, lo que viene diciendo es que si una función recibe un objeto como parámetro, de tipo X y en su lugar le pasamos otro de tipo Y, que hereda de X, dicha función debe proceder correctamente.

Por el propio polimorfismo, los compiladores e intérpretes admiten este paso de parámetros, la cuestión es si la función de verdad está diseñada para hacer lo que debe, aunque quien recibe como parámetro no es exactamente X, sino Y.

El principio de sustitución de Liskov está estrechamente relacionado con el anterior en cuanto a la extensibilidad de las clases cuando ésta se realiza mediante herencia o subtipos. Si una función no cumple el LSP entonces rompe el OCP puesto que para ser capaz de funcionar con subtipos (clases hijas) necesita saber demasiado de la clase padre y por tanto, modificarla. El diseño por contrato (Design by Contract) es otra forma de llamar al LSP.

# Interface Segregation Principle (ISP)

Cuando empleamos el SRP también empleamos el ISP como efecto colateral. El ISP defiende que no obliguemos a los clientes a depender de clases o interfaces que no necesitan usar. Tal imposición ocurre cuando una clase o interfaz tiene más métodos de los que un cliente (otra clase o entidad) necesita para sí mismo. Seguramente sirve a varios objetos cliente con responsabilidades diferentes, con lo que debería estar dividida en varias entidades.

En los lenguajes como Java y C# hablamos de interfaces pero en lenguajes interpretados como Python, que no requieren interfaces, hablamos de clases. No sólo es por motivos de robustez del software, sino también por motivos de despliegue. Cuando un cliente depende de una interfaz con funcionalidad que no utiliza, se convierte en dependiente de otro cliente y la posibilidad de catástrofe frente a cambios en la interfaz o clase base se multiplica.

# Dependency Inversión Principle (DIP)

La inversión de dependencias da origen a la conocida inyección de dependencias, una de las mejores técnicas para lidiar con las colaboraciones entre clases, produciendo un código reutilizable, sobrio y preparado para cambiar sin producir efectos bola de nieve.

DIP explica que un módulo concreto A, no debe depender directamente de otro módulo concreto B, sino de una abstracción de B. Tal abstracción es una interfaz o una clase (que podría ser abstracta) que sirve de base para un conjunto de clases hijas.

En el caso de un lenguaje interpretado no necesitamos definir interfaces, ni siquiera jerarquías pero el concepto se aplica igualmente. Veámoslo con un ejemplo sencillo: La clase Logica necesita de un colaborador para guardar el dato Dato en algún lugar persistente. Disponemos de una clase MyBD que es capaz de almacenar Dato en una base de datos MySQL y de una clase FS que es capaz de almacenar Dato en un fichero binario sobre un sistema de ficheros NTFS.

Si en el código de Logica escribimos literalmente el nombre de la clase MyBD como colaborador para persistir datos, ¿Cómo haremos cuando necesitamos cambiar la base de datos por ficheros binarios en disco?. No quedará otro remedio que modificar el código de Logica.

Si las clases MyDB y FS implementasen una misma interfaz IPersistor para guardar Dato, podríamos limitarnos a usar IPersistor (que es una abstracción) en el código de Logica. Cuando los requerimientos exigiesen un cambio de base de datos por ficheros en disco o viceversa, sólo tendríamos que preocuparnos de que el atributo \_myPersistor de la clase Logica, que es de tipo IPersistor contuviese una instancia de MyDB o bien de FS.

¿Cómo resolvemos esta última parte?. Con la inyección de dependencias, que vamos a ver dentro del siguiente apartado, Inversión del Control. En los próximos capítulos haremos mucho uso de la inyección de dependencias con gran cantidad de listados de código. No se preocupe si el ejemplo no le queda demasiado claro.