UNIVERSIDAD PRIVADA DE TACNA
INGENIERIA DE SISTEMAS

TITULO:

"Principios de Diseño y DDD"

CURSO:

CALIDAD Y PRUEBAS DE SOFTWARE

DOCENTE:

Ing. Patrick Cuadros Quiroga

Integrantes:

Maldonado Cancapi, Carlos Alejandro	(2018000660)
Villanueva Yucra, Josue Joel	(2018000722)
Contreras Murguia, Jose Manuel	(2016056346)
Rojas Bedregal, Brian Erik	(2018060904)
Mamani Laura, Juan Carlos	(2017059565)

Tacna - Perú
2021

Resumen

Todas las aplicaciones cambian su ciclo de vida, y siempre vendran nuevas versiones tras la primera release. No por ello debemos adelantarnos a desarrollar caracteristicas que el cliente podria necesitar en el futuro; si nos pusieramos en el papel de adivinos, seguramente fallariamos y probablemnte desarrollariamos caracteristicas que el cliente nunca necesitará, Por suerte existen principios que se llamaron S.O.L.I.D. que al aplicarlos facilitará mucho el trabajo, tanto propio como ajeno.

Palabras Clave: Principios S.O.L.I.D, ciclo de vida

Abstract

All applications change their life cycle, and new versions will always come after the first release. Not for this reason we must anticipate developing characteristics that the client you might need in the future; If we put ourselves in the role of fortune tellers, we would surely fail and probably develop characteristics that the client will never need. Fortunately, there are principles that were called SOLID. that by applying them it will greatly facilitate the work, both your own and that of others.

Keywords: S.O.L.I.D Principles, Lifecycle.

1 Introducción

S.O.L.I.D. es el acrónimo de los cinco principios básicos de la programación orientada a objetos.¿ Dentro de esos principios, unos de los más conocidos fueron justamente los cinco princicpios S.O.L.I.D. definidos por Robert C. Martin. El conocer y aplicar estos principios ofrecen una serie de ventajas al desarrollador a la hora de programar, independientemente del lenguaje utilizado. El código resultante estará menos acoplado por lo que será más fácil de extender, de mantener, más legible, reutilizable y más sencillo realizar test unitarios. En conclusión tendremos un mejor diseño en la arquitectura de la aplicación y un código de mayor calidad.

2 Desarrollo

Los 5 principios SOLID de diseño de aplicaciones de software son:

2.1 Single Responsibility Principle (SRP)

La S del acrónimo del que hablamos hoy se refiere a Single Responsibility Principle (SRP). Según este principio "una clase debería tener una, y solo una, razón para cambiar". Es esto, precisamente, "razón para cambiar", lo que Robert C. Martin identifica como "responsabilidad".

El principio de Responsabilidad Única es el más importante y fundamental de SOLID, muy sencillo de explicar, pero el más difícil de seguir en la práctica.

2.1.1 ¿Cómo detectar si estamos violando el Principio de Responsabilidad Única?

La respuesta a esta pregunta es bastante subjetiva. Sin necesidad de obsesionarnos con ello, podemos detectar situaciones en las que una clase podría dividirse en varias:

- En una misma clase están involucradas dos capas de la arquitectura: esta puede ser difícil de ver sin experiencia previa. En toda arquitectura, por simple que sea, debería haber una capa de presentación, una de lógica de negocio y otra de persistencia. Si mezclamos responsabilidades de dos capas en una misma clase, será un buen indicio.
- El número de métodos públicos: Si una clase hace muchas cosas, lo más probable es que tenga muchos métodos públicos, y que tengan poco que ver entre ellos. Detecta cómo puedes agruparlos para separarlos en distintas clases. Algunos de los puntos siguientes te pueden ayudar.
- Los métodos que usan cada uno de los campos de esa clase: si tenemos dos campos, y uno de ellos se usa en unos cuantos métodos y otro en otros cuantos, esto puede estar indicando que cada campo con sus correspondientes métodos podrían formar una clase independiente. Normalmente esto estará más difuso y habrá métodos en común, porque seguramente esas dos nuevas clases tendrán que interactuar entre ellas.
- Por el número de imports: Si necesitamos importar demasiadas clases para hacer nuestro trabajo, es posible que estemos haciendo trabajo de más. También ayuda fijarse a qué paquetes pertenecen esos imports. Si vemos que se agrupan con facilidad, puede que nos esté avisando de que estamos haciendo cosas muy diferentes.
- Nos cuesta testear la clase: si no somos capaces de escribir tests unitarios sobre ella, o no conseguimos el grado de granularidad que nos gustaría, es momento de plantearse dividir la clase en dos.
- Cada vez que escribes una nueva funcionalidad, esa clase se ve afectada: si una clase se modifica a menudo, es porque está involucrada en demasiadas cosas.
- Por el número de líneas: a veces es tan sencillo como eso. Si una clase es demasiado grande, intenta dividirla en clases más manejables.

En general no hay reglas de oro para estar 100 por ciento seguros. La práctica te irá haciendo ver cuándo es recomendable que cierto código se mueva a otra clase, pero estos indicios te ayudarán a detectar algunos casos donde tengas dudas.

2.2 Open/Closed Principle (OCP)

El segundo principio de SOLID lo formuló Bertrand Meyer en 1988 en su libro "Object Oriented Software Construction" y dice: "Deberías ser capaz de extender el comportamiento de una clase, sin modificarla". En otras palabras: las clases que usas deberían estar abiertas para poder extenderse y cerradas para modificarse.

En su blog Robert C. Martin defendió este principio que a priori puede parecer una paradoja. Es importante tener en cuenta el Open/Closed Principle (OCP) a la hora de desarrollar clases, librerías o frameworks.

Este principio nos dice que una entidad de software debería estar abierta a extensión pero cerrada a modificación. ¿Qué quiere decir esto? Que tenemos que ser capaces de extender el comportamiento de nuestras clases sin necesidad de modificar su código. Esto nos ayuda a seguir añadiendo funcionalidad con la seguridad de que no afectará al código existente. Nuevas funcionalidades implicarán añadir nuevas clases y métodos, pero en general no debería suponer modificar lo que ya ha sido escrito.

2.2.1 ¿Cómo detectar que estamos violando el principio Open/Closed?

Una de las formas más sencillas para detectarlo es darnos cuenta de qué clases modificamos más a menudo. Si cada vez que hay un nuevo requisito o una modificación de los existentes, las mismas clases se ven afectadas, podemos empezar a entender que estamos violando este principio

2.2.2 ¿Cuándo debemos cumplir con este principio?

Hay que decir que añadir esta complejidad no siempre compensa, y como el resto de principios, sólo será aplicable si realmente es necesario. Si tienes una parte de tu código que es propensa a cambios, plantéate hacerla de forma que un nuevo cambio impacte lo menos posible en el código existente. Normalmente esto no es fácil de saber a priori, por lo que puedes preocuparte por ello cuando tengas que modificarlo, y hacer los cambios necesarios para cumplir este principio en ese momento. Intentar hacer un código 100 por ciento Open/Closed es prácticamente imposible, y puede hacer que sea ilegible e incluso más difícil de mantener. No me cansaré de repetir que las reglas SOLID son ideas muy potentes, pero hay que aplicarlas donde corresponda y sin obsesionarnos con cumplirlas en cada punto del desarrollo. Casi siempre es más sencillo limitarse a usarlas cuando nos haya surgido la necesidad real.

2.3 Liskov Substitution Principle (LSP)

La L de SOLID alude al apellido de quien lo creó, Barbara Liskov, y dice que "las clases derivadas deben poder sustituirse por sus clases base".

Esto significa que los objetos deben poder ser reemplazados por instancias de sus subtipos sin alterar el correcto funcionamiento del sistema o lo que es lo mismo: si en un programa utilizamos cierta clase, deberíamos poder usar cualquiera de sus subclases sin interferir en la funcionalidad del programa.

Según Robert C. Martin incumplir el Liskov Substitution Principle (LSP) implica violar también el principio de Abierto/Cerrado.

El principio de sustitución de Liskov nos dice que si en alguna parte de nuestro código estamos usando una clase, y esta clase es extendida, tenemos que poder utilizar cualquiera de las clases hijas y que el programa siga siendo válido. Esto nos obliga a asegurarnos de que cuando extendemos una clase no estamos alterando el comportamiento de la padre. Este principio viene a desmentir la idea preconcebida de que las clases son una forma directa de

modelar la realidad. Esto no siempre es así, y el ejemplo más típico es el de un rectángulo y un cuadrado.

2.3.1 ¿Cómo detectar que estamos violando el principio de sustitución de Liskov?

Seguro que te has encontrado con esta situación muchas veces: creas una clase que extiende de otra, pero de repente uno de los métodos te sobra, y no sabes que hacer con él. Las opciones más rápidas son bien dejarlo vacío, bien lanzar una excepción cuando se use, asegurándote de que nadie llama incorrectamente a un método que no se puede utilizar. Si un método sobrescrito no hace nada o lanza una excepción, es muy probable que estés violando el principio de sustitución de Liskov. Si tu código estaba usando un método que para algunas concreciones ahora lanza una excepción, ¿cómo puedes estar seguro de que todo sigue funcionando? Otra herramienta que te avisará fácilmente son los tests. Si los tests de la clase padre no funcionan para la hija, también estarás violando este principio.

2.3.2 CONCLUSION

El principio de Liskov nos ayuda a utilizar la herencia de forma correcta, y a tener mucho más cuidado a la hora de extender clases. En la práctica nos ahorrará muchos errores derivados de nuestro afán por modelar lo que vemos en la vida real en clases siguiendo la misma lógica. No siempre hay una modelización exacta, por lo que este principio nos ayudará a descubrir la mejor forma de hacerlo.

2.4 Interface Segregation Principle (ISP)

El principio de segregación de interfaces viene a decir que ninguna clase debería depender de métodos que no usa. Por tanto, cuando creemos interfaces que definan comportamientos, es importante estar seguros de que todas las clases que implementen esas interfaces vayan a necesitar y ser capaces de agregar comportamientos a todos los métodos. En caso contrario, es mejor tener varias interfaces más pequeñas. Las interfaces nos ayudan a desacoplar módulos entre sí. Esto es así porque si tenemos una interfaz que explica el comportamiento que el módulo espera para comunicarse con otros módulos, nosotros siempre podremos crear una clase que lo implemente de modo que cumpla las condiciones. El módulo que describe la interfaz no tiene que saber nada sobre nuestro código y, sin embargo, nosotros podemos trabajar con él sin problemas.

La problemática surge cuando esas interfaces intentan definir más cosas de las debidas, lo que se denominan fat interfaces. Probablemente ocurrirá que las clases hijas acabarán por no usar muchos de esos métodos, y habrá que darles una implementación. Muy habitual es lanzar una excepción, o simplemente no hacer nada. Pero, al igual que vimos en algún ejemplo en el principio de sustitución de Liskov, esto es peligroso. Si lanzamos una excepción, es más que probable que el módulo que define esa interfaz use el método en algún momento, y esto hará fallar nuestro programa. El resto de implementaciones “por defecto” que podamos dar pueden generar efectos secundarios que no esperemos, y a los que sólo podemos responder conociendo el código fuente del módulo en cuestión, cosa que no nos interesa

2.4.1 ¿Cómo detectar que estamos violando el Principio de segregación de interfaces?

Si al implementar una interfaz ves que uno o varios de los métodos no tienen sentido y te hace falta dejarlos vacíos o lanzar excepciones, es muy probable que estés violando este principio. Si la interfaz forma parte de tu código, divídela en varias interfaces que definan comportamientos más específicos. Recuerda que no pasa nada porque una clase ahora necesite implementar varias interfaces. El punto importante es que use todos los métodos definidos por esas interfaces.

2.4.2 CONCLUSION

El principio de segregación de interfaces nos ayuda a no obligar a ninguna clase a implementar métodos que no utiliza. Esto nos evitará problemas que nos pueden llevar a errores inesperados y a dependencias no deseadas. Además nos ayuda a reutilizar código de forma más inteligente.

2.5 Dependency Inversion Principle (DIP)

Este principio es una técnica básica, y será el que más presente tengas en tu día a día si quieres hacer que tu código sea testable y mantenible. Gracias al principio de inversión de dependencias, podemos hacer que el código que es el núcleo de nuestra aplicación no dependa de los detalles de implementación, como pueden ser el framework que utilices, la base de datos, cómo te conectes a tu servidor... Todos estos aspectos se especificarán mediante interfaces, y el núcleo no tendrá que conocer cuál es la implementación real para funcionar.

La definicion que se suele dar es:

- Los módulos de alto nivel no deberían depender de módulos de bajo nivel. Ambos deberían depender de abstracciones.
- Las abstracciones no deberían depender de los detalles.
- Los detalles deberían depender de las abstracciones.

El objetivo del Dependency Inversion Principle (DIP) consiste en reducir las dependencias entre los módulos del código, es decir, alcanzar un bajo acoplamiento de las clases.

En la programación vista desde el modo tradicional, cuando un módulo depende de otro módulo, se crea una nueva instancia y la utiliza sin más complicaciones. Esta forma de hacer las cosas, que a primera vista parece la más sencilla y natural, nos va a traer bastantes problemas posteriormente, entre ellos:

- Las parte más genérica de nuestro código (lo que llamaríamos el dominio o lógica de negocio) dependerá por todas partes de detalles de implementación. Esto no es bueno, porque no podremos reutilizarlo, ya que estará acoplado al framework de turno que usemos, a la forma que tengamos de persistir los datos, etc. Si cambiamos algo de eso, tendremos que rehacer también la parte más importante de nuestro programa.
- No quedan claras las dependencias: si las instancias se crean dentro del módulo que las usa, es mucho más difícil detectar de qué depende nuestro módulo y, por tanto, es más

difícil predecir los efectos de un cambio en uno de esos módulos. También nos costará más tener claro si estamos violando algunos otros principios, como el de responsabilidad única.

- Es muy complicado hacer tests: Si tu clase depende de otras y no tienes forma de sustituir el comportamiento de esas otras clases, no puedes testarla de forma aislada. Si algo en los tests falla, no tendrías forma de saber de un primer vistazo qué clase es la culpable.

2.6 Domain-Driven Design - Patron

Este enfoque para desarrollo de software definido por Eric Evans en su libro Domain-Driven Design: Tackling Complexity in the Heart of Software, en el que expone un modelo rico, expresivo y en constante evolución que busca resolver problemas del dominio de una forma semántica

El libro publicado en el 2004 recopila una serie de patrones y recomendaciones centradas en diseñar un modelo de dominio que ayude a tratar la complejidad de las aplicaciones.

El DDD no es una tecnología ni una metodología, es una técnica que está estructurada por varias prácticas que podrán ayudarte a tomar decisiones de diseño con el fin de enfocar y acelerar el manejo de dominios complejos durante el desarrollo de proyectos digitales.

2.6.1 ¿Qué problema resuelve?

Depende del caso o proyecto nos podemos encontrar con que la complejidad de muchas aplicaciones no está en la parte técnica sino en la lógica del negocio o dominio.

El dilema empieza cuando intentamos resolver problemas del dominio con tecnología. Eso provoca que, aunque la aplicación funcione, no haya nadie capaz de entender realmente cómo lo hace. Es habitual que surja el anti-patrón "Modelo del Dominio Anémico", en inglés (Anemic Domain Model). Si esto sucede, nos encontraremos con objetos que llevan nombres sacados del dominio y forman una estructura que, a primera vista, parece un modelo del dominio pero la realidad es que estos objetos son solo un conjunto de datos sin comportamiento, implementados por la lógica en objetos servicio.

Al hacer DDD evitamos esta y otras malas prácticas, previniendo así que acabemos con una aplicación complicada al momento de mantenerla.

2.6.2 Pre-Requisitos para hacer DDD

- El desarrollo debe ser iterativo. Esto será necesario para ir refinando el modelo del dominio continuamente a medida que aprendemos más sobre este y avanzamos.

- Debe existir una estrecha relación entre los desarrolladores y los expertos del dominio. Como dije antes, el conocimiento profundo del dominio es esencial, al igual que la colaboración con los expertos de desarrollo durante la vida del proyecto; esto evitará malos entendidos entre las partes del equipo y ofrecerá la oportunidad de obtener un conocimiento más profundo del dominio.

Esta técnica fue ideada para el desarrollo de aplicaciones complejas y está orientada a proyectos que usen metodologías ágiles.

Como debes saber, uno de los aspectos más complicados de los proyectos de software complejos está en el dominio del mundo real al que sirve el software y no en su implementación; es en este punto en el que el DDD podrá ayudar a tener una mayor visión y enfoque para evolucionar a través de sucesivas iteraciones del diseño.

3 Conclusiones

Los principios SOLID son eso: principios, es decir, buenas prácticas que pueden ayudar a escribir un mejor código: más limpio, mantenible y escalable.

Como indica el propio Robert C. Martin en su artículo "Getting a SOLID start" no se trata de reglas, ni leyes, ni verdades absolutas, sino más bien soluciones de sentido común a problemas comunes. Son heurísticos, basados en la experiencia: "se ha observado que funcionan en muchos casos; pero no hay pruebas de que siempre funcionen, ni de que siempre se deban seguir."

4 Recomendaciones

Hay que ser algo pragmático al momento de la implementación de un requerimiento, buscando la solución que mejor rendimiento entregue, que tenga un menor costo y genere más dinero a la organización. No se usan metodologías, Frameworks o tecnologías arbitrariamente (Recomendación: Hacer un TradeOff para este tipo de decisiones).

Referencias

[1] Manu Pijierro (2017) Principios SOLID: Principio de Responsabilidad Única. Recuperado 16 de Abril 2021, de https://mpijierro.medium.com/principios-solid-principio-de-responsabilidad-

[2] Carlos Macías Martín (2019).Principios SOLID.Recuperado 16 de Abril 2021, de https://enmilocalfunciona.io/principios-solid/

[3] Carlos Blé Jurado, Diseño Ágil con TDD (2019)

[4] Antonio Leiva (2016) Principios SOLID, Guía rápida para aprender qué son y cómo aplicarlos en tu día a día

[5] María José Martín (2018) SOLID: los 5 principios que te ayudarán a desarrollar software de calidad. Recuperado 16 de Abril 2021, de https://profile.es/blog/principios-solid-desarrollo-software-calidad/

[6] Miriam Martinez Canelo ¿Qué son los patrones de diseño de software?. Recuperado 16 de Abril 2021, de https://profile.es/blog/patrones-de-diseno-de-software/

[7] Matt Carroll (2016) Book Review: Domain-Driven Design

[8] Alex Sosa (2018) Solución ejercicios de principios S.O.L.I.D, Recuperado 16 de Abril 2021, de https://github.com/alejandrososa/ddd-tutorial/tree/master/Clase1

[9] Javier Pardo (2020) Domain Driven Design y arquitectura Onion, todo lo que necesitas saber. Recuperado 16 de Abril 2021, de https://www.paradigmadigital.com/techbiz/domain-driven-design-y-arquitectura-onion/

[10] Halil İbrahim Kalkan (2021) Implementing Domain Driven Design, Recuperado 16 de Abril 2021, de https://docs.abp.io/en/abp/latest/Domain-Driven-Design-Implementation-Guide