ESCOM - IPN

Principios de Diseño

Análisis y Diseño Orientado a Objetos

Oscar Andrés Rosas Hernandez

Mayo 2018

Índice

1. Principios de Diseño

2. Principios SOLID

La importancia del principio sólido es simple, nos indican la mejor práctica para escribir códigos en módulos, con aislamiento y una manera robusta para que más adelante pueda agregar fácilmente un nuevo código o nuevas funcionalidades sin cambiar la arquitectura del código.

Estos 5 principios han sido introducidos en una serie de artículos escritos en los años 90 por el Bob Martin en una revista (discontinuada) llamada "The C ++ Report".

Es una alternativa (solo en la forma, no tanto en la sustancia) a los conceptos difundidos de un buen diseño de POO (débilmente acoplado, cohesivo, encapsulado, no redundante, etc) se define este conjunto de 5 principios:

SRP Principio de Responsabilidad Individual (AKA Cohesión): Üna clase debe tener solo una razón para cambiar".

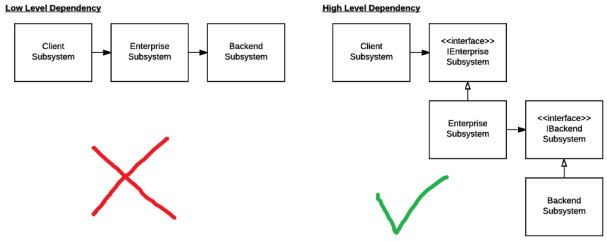
Bob había publicado en su antiguo sitio web http://objectmentor.com un extracto sobre él en su obra maestra, vinculado también en las notas a pie de página de sus libros más recientes. Desafortunadamente ese sitio web ya no está en línea.

OCP Principio Cerrado Abierto:

"Las entidades de software (clases, módulos, funciones, etc.) deben estar abiertas para la extensión pero cerradas para su modificación".

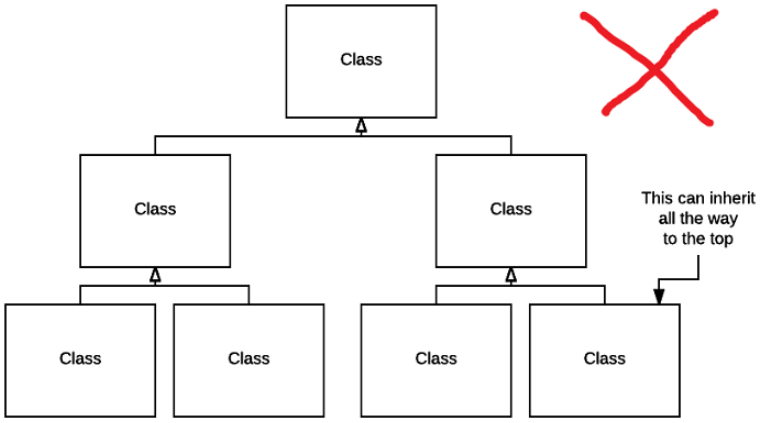
Se necesita que tu puedas ampliar tus clases / funciones sin modificarlas. ¿Cómo?

En mi experiencia, diría que aplicando el patrón de Inyección de Dependencia. Hay que tener en cuenta que la extensión no estamos hablando de que va a extender una clase, por el contrario, puede optar por optar por una clase / método final que no se extenderá y se ampliará la lógica mediante la adición de nuevas implementaciones en el contexto de los patrones de diseño de Composición / Estrategia (o incluso Comando).

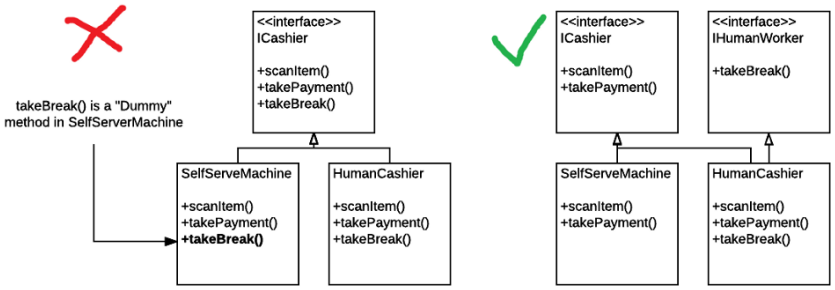


LSP Principio de sustitución de Liskov: "Las subclases deben ser sustituibles por sus tipos base".

La violación de este principio rompe el contrato del tipo base / superclase. Este principio es una especie de desincentivo al uso de la herencia sobre la composición, al menos en algunas situaciones.



ISP Principio de segregación de la interfaz: "No se debe forzar a los clientes a depender de interfaces que no usan", Que se traduce en "definir interfaces pequeñas lo más esenciales posible, una interfaz que se ajuste a todo solo causará problemas".



DIP Principio de inversión de dependencia: Definido por tío bob como "Los módulos de alto nivel no deberían depender de módulos de bajo nivel. Ambos deberían depender de las abstracciones. Las abstracciones no deberían depender de los detalles. Los detalles deben depender de las abstracciones."

Suena al principio un poco .ªbstractoçomo concepto ...

Normalmente lo interpreto como üse interfaces y evite el cableado en las clases concretas de código (implementación / detalles / módulos de bajo nivel) tanto como pueda".

Creo que esto está muy bien acoplado con Dependency Injection y frameworks DI / IOC como Spring, que te permitirá definir tus interfaces y no te preocupes por pasar manualmente la implementación concreta.

Desde una perspectiva de prueba, el uso de interfaces como dependencias también podría ofrecer muchas ventajas, tanto desde un aspecto de verificación (proporcionaré un objeto que se burla de una interfaz dada y verificar las interacciones de mi prueba en clase con él) como desde una dependencia aspecto de gestión (no necesito la implementación real de X como lo requiere mi método / clase de prueba, cualquier otra implementación de stub servirá para el propósito de esta prueba).

Desafortunadamente, al menos en Java, los frameworks de ayudantes de prueba modernos (por ejemplo, Mockito) son demasiado buenos con los desarrolladores y permiten simular clases concretas, por lo que debemos recordar preferir interfaces incluso si pudiéramos simplemente usar clases concretas porque las herramientas nos lo permiten.

3. Definición Real

Acerca del diseño del software podemos reducirlo a:

- Rápido
- Escalable
- Seguro
- Pequeña
- Legible

Muchos libros te enseñaran una gran cantidad de cosas que hacer y qué no hacer, pero al final, en la industria, todo resume a estas 5 cualidades que todo lo que hagas debe tener.

No tomar en cuanta alguno de ellos significa que lo estámos haciendo mal.

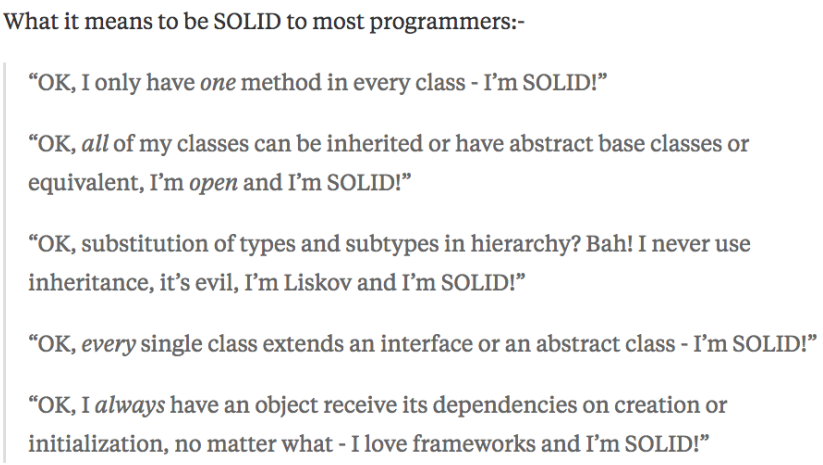


Figura 1: Un mensaje que vale la pena leer

4. ¿Valen la pena aprenderlo?

Recordemos:

- Principio de Responsabilidad Individual
- Principio abierto / cerrado
- Principio de sustitución Liskov
- Principio de segregación de interfaz
- Principio de inversión de dependencia

Si bien muchos frameworks modernos se basan en muchos de estos conceptos, y es difícil trabajar en contra de ellos, siguen siendo conceptos de diseño de software muy válidos.

Ayudan cuando preguntas "¿a dónde pertenece este método? o incluso tiene preguntas sobre el flujo y la presentación de datos.

Si bien es probable que no te vayan a interrogar sobre esto en la entrevista deltrabajo

Es similar a decir "Voy a desperdiciar miles de años de experiencia en ingeniería y arquitectura cuando construya esta casa y simplemente lo haga como yo quiera".

Párate sobre los hombros de los gigantes que te precedieron.

Referencias

[1] http://tekina.info/solid-principles/

[2] https://www.quora.com/Object-Oriented-Design-What-is-the-significance-of-the-SOLID-principles

[3] https://medium.com/@sadatnazrul/software-development-design-principles-79d15ef765f3