Arquitectura del Software

SOLID vs CUPID

Daniel Fernández Bernardino

Hugo Gutiérrez Tomás

Lucas Martínez Rego

Paula Puerta González

04/03/2022

Índice de contenidos

Propiedades CUPID

CUPID son una serie de propiedades que tienen como objetivo general obtener un código fácil de usar, modificar y seguir desarrollando por otras personas.

Composable: Software creado para usarse y modificarse fácilmente. Se dan una serie de técnicas para cumplir este propósito: API limitada, con la intención de evitar el sobre aprendizaje y ser menos propenso a fallos e inconsistencias. Hay que encontrar el punto medio entre una API demasiado grande y una demasiado pequeña. Intención reveladora, referida al nombrado del código y métodos, estos deben revelar qué es lo que pretenden hacer.

Unix-Philosophy: Utilizada en la mayoría de los servidores comerciales y en la nube, su modelo simple y consistente establece que los componentes que componen un sistema deben realizar una sola operación, pero está deben hacerla de forma totalmente satisfactoria. A su vez, los componentes deben poder comunicarse entre ellos, lo que nos proporciona un sistema cuyos componentes tienen un propósito único y, al estar bien comunicados con el resto de los componentes, el sistema sigue teniendo mucho potencial.

Predictable: El código debe hacer lo que parece que hace. No debe haber "sorpresas desagradables" en tiempo de ejecución y el nivel de consistencia debe ser alto. Se suelen usar dos técnicas: Se comporta como se espera, se suelen utilizar pruebas para cumplir este propósito (se deberían de hacer pruebas de todo tipo y el todo el sistema para probarlo correctamente). Determinista, referido al software, este debe hacer lo mismo cada vez. Gracias a esto aumentamos la robustez, fiabilidad y resistencia del código y hacemos que el punto anterior sea más fácil de cumplir.

Idiomatic: Seguir una serie de nociones, tanto lingüísticas como idiomáticas, para tener un código parecido en todas las clases del sistema.

Domain-Based: Se deben seguir, tanto en el lenguaje (nombrado de métodos) como en la estructura (estructura de directorios y sus relaciones), el modelo de dominio. De esta forma el código va a transmitir mejor lo que está haciendo en el sistema y añadir nuevos componentes va a ser una tarea más fácil.

Principios SOLID

Solid es un acrónimo acuñado por Robert C. Martin y define los 5 principios básicos de la programación orientada a objetos. Se trata de una filosofía que hará que nuestro código sea más mantenible, reutilizable y escalable a futuro.

Principio de Responsabilidad Única (SRP) : Las clases deben tener solo un motivo para cambiar. Entendemos el motivo para el cambio como una responsabilidad.

Principio de abierto-cerrado(OCP): Las clases deben estar abiertas para la extensión, pero cerradas para la modificación. Los cambios en las clases deben ser añadiendo nuevo código, no modificando el existente que ya funciona.

Principio de sustitución de Liskov(LSP): Los subtipos deben ser capaces de sustituir a sus tipos bases sin alterar el correcto funcionamiento del sistema. Debemos tener cuidado al implementar los subtipos de modo que no rompamos el contrato con su clase base.

Principio de inversión de dependencias(DIP): Los módulos de alto nivel no deben depender de los de bajo nivel, ambos deben depender de las abstracciones. Del mismo modo, las abstracciones no deben depender de los detalles, sino estos de las abstracciones.

Principio de segregación de interfaces(ISP): Es mejor hacer varias interfaces distintas para varios clientes que una única interfaz de propósito general.

Por qué SOLID está mal según CUPID

El principal motivo es que, en el momento en que se crearon o escribieron, éstos eran útiles, y en lo general ciertos. Aunque con el paso del tiempo, la mayoría de los factores que limitaban el desarrollo de código y que daban veracidad a estos principios han cambiado y evolucionado mucho.

SOLID propone soluciones a problemas que en lo general funcionan, aunque como Dan North defiende, éstos son patrones: tienen beneficios y detrimentos, por lo que, aunque realmente son efectivos, también surgen problemas a raíz de aplicarlos.

Los principios **SOLID** son realmente patrones que, aplicados en un contexto adecuado, son generalmente útiles, aunque muy fácilmente pueden implementarse de forma errónea. Como respuesta a esto, Dan North propone una serie de principios, denominados **CUPID**, explicados anteriormente. Podrían resumirse con una sola frase: **escribir código más simple**.

Se debe escribir código que hace lo que parece que hace, es fácilmente comprensible y lo suficientemente compacto como para poder conformar estructuras más complejas y de mayor tamaño.

Comparativa SOLID vs CUPID

Como bien indica su nombre los principios SOLID, son **reglas** que es necesario cumplir y cuya violación produce un error; por otro lado, CUPID nos transmite propiedades, que no es más que unas **cualidades finales** que es **deseable** alcanzar.

Mientras que SOLID está enfocado en el **rendimiento del código**, CUPID pone foco en que la **codificación** sea **fácilmente entendida** por otras personas. Siguiendo por el mismo camino, la primera necesita de unos conocimientos teóricos para ser aplicada correctamente, usualmente, requiere de ejemplos para que sus principios sean comprendidos, lo que los hace más **difícil de aplicar** de forma natural por **inexpertos**; al contrario que CUPID, que se muestra más **intuitivo**, sin precisar del uso de tecnicismos al centrarse en la simplicidad, lo que lo convierte su aplicación más **ingenua** y accesible por noveles.

Debido a que los principios SOLID se basan en las tecnologías existentes en los años 90, una de sus máximas es **evitar el cambio**, ya que en esa época la potencia de procesamiento era ínfima comparada con la actual, este es un gran signo de la desactualización de SOLID

respecto a su rival CUPID que ha tenido en cuenta los avances realizados en materia de hardware y trata el **cambio** de lo ya existente como algo **factible**.

Para concluir, la rigidez de los principios SOLID hace que el saber si se está aplicando o no sea claro, pero no tiene en cuenta la complejidad que se puede generar en el código, que al fin y al cabo será lo que haga que un código sea duradero y que no se enmarañe con los cambios de distintos desarrolladores. Y, tal y como se ha mencionado, la **falta de actualización de SOLID** lo hace cada vez menos útil.