Principios de diseño

Simplicidad

En esta sección veremos algunos principios de simplicidad del software.

La simplicidad en cualquier diseño de software es un atributo deseable. Esto no quiere decir que tenemos que ignorar la complejidad propia de lo que estamos haciendo, pero debemos hacerlo los más simple posible. Como dice una cita atribuida a Albert Einstein “Make everything as simple as possible, but not simpler“, es decir, “Hacer todo lo más simple posible, pero no más simple”.

KISS

El principio KISS, del inglés *Keep It Short and Simple* o mantenlo corto y simple, establece que la mayoría de sistemas funcionan mejor si se mantienen simples que si se hacen complejos. La simplicidad debe ser mantenida como un objetivo clave del diseño, y cualquier complejidad innecesaria debe ser evitada.

YAGNI

El principio *Yagni*, del inglés You aren’t gonna need it (No lo vas a necesitar), es otro principio que está muy relacionado con el principio KISS. Este principio nos advierte sobre escribir código de la forma más general, para “anticipar cambios futuros” , la intención es buena, pero en la mayoría de los casos suele ser código inútil.

En otras palabras este principio nos dice que sólo se desarrolle lo que se vaya a usar en el momento, que se piense si se va a usar la funcionalidad y si se va a necesitar en un futuro inmediato. No se debe desarrollar algo que no se vaya a usar pronto, aunque se sepa con certeza que se va a tener que implementar en un futuro.

The Hollywood Principle

Por otro lado, tenemos el principio Hollywood: “No nos llames, nosotros te llamamos”. También conocido como principio de **inversión de dependencia**.

Por el 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 se utiliza, la base de datos, cómo se realiza la conexión a un servidor, etc.

Single Responsibility Principle (SRP)

El Principio de responsabilidad única, del inglés Single responsibility principle (SRP), nos dice que “Un objeto debe realizar un única cosa”. En caso contrario debemos dividir la responsabilidad entre varios objetos.

Por ejemplo, si un objeto representa un triángulo; y sabe responder los mensajes *area* y *dibujar*. Este objeto tiene dos responsabilidades:

1. Calcular su área.
2. Dibujarse en pantalla.

Entonces para cumplir el principio de responsabilidad única, debo separar la responsabilidad de dibujarse en pantalla, siendo otro objeto el que sepa dibujar el triángulo en pantalla.

Consistencia

El principio de consistencia radica en que el software sea consistente con el dominio de problema y que no haya redundancia que puede traer conflictos por falta de consistencia.

Por una lado, es recomendable utilizar metáforas de cómo se haría lo que queremos hacer sin un sistema. Por ejemplo, si queremos hacer un facturador de llamadas telefónicas, nos podemos situar en los años en que no había sistemas y este trabajo lo hacían personas manualmente. Entonces, habría una persona que tenía un listado de llamadas y agrupaba los llamados por cliente, otra persona buscaba los costos de cada llamado según su duración y tipo de llamada, otra persona sumaba los costos de cada llamada, otra persona armaba la factura, etc.

De esa manera podemos encontrar mejor los objetos que forman parte del modelo por sus responsabilidades, siendo consistentes con el dominio de problema.

DRY

El principio DRY, del inglés Don't repeat yourself (No te repitas), ayuda también a mantener la consistencia. Frecuentemente en trabajos colaborativos hay redundancia y repetición de código, debido a la falta de comunicación y a la falta de especificaciones de diseño. Esto puede causar ciertos problemas que luego dificultan hacer cambios y mantener la consistencia.

Si seguimos este principio, en todas las etapas del desarrollo, mantendremos la consistencia y evitaremos esfuerzos adicionales.

Entendible

Una característica importante del software es que sea entendible. Esto se logra de dos manera principalmente. Por un lado, que el código sea legible, prolijo, etc. Por otro lado, es que haya un mapeo entre el dominio de problema y el diseño. Idealmente que sea uno a uno. Esto es porque si alguien conoce el dominio de problema, es más sencillo que entienda el software.

Máxima cohesión

La máxima cohesión significa que un objeto sea bien funcional, es decir, cuando el objeto realiza una única cosa. En definitiva si cumplimos el principio de responsabilidad única tendremos objetos altamente cohesivos.

Si tengo objetos poco cohesivos al querer introducir cambios en el software surgen muchas complicaciones porque hay muchas funcionalidades mezcladas. Hacer un cambio cuesta mucho más porque hay que entender que hay que cambiar y donde.

Además, como las funcionalidades están mezcladas, por un cambio que tengo que hacer se rompe otra funcionalidad que no tiene ninguna relación con el cambio que estoy haciendo. Con lo cual, también se suma una complicación más y es que tengo que probar más funcionalidades, y no solo la que estoy cambiando.

Mínimo acoplamiento

El mínimo acoplamiento significa que un objeto depende de la menor cantidad de objetos posibles.

Un objeto siempre está relacionado con otros objetos, sino no serviría de mucho. En otras palabras, un objeto con cero acoplamiento es inútil. Por eso lo que se busca es minimizar el acoplamiento y no anularlo.

Si tengo un objeto con muchas dependencias (es decir, muy acoplado), se produce lo que se llama “ripple effect” o efecto dominó, porque al cambiar un objeto deben cambiar sus dependencias.

Entonces mientras menos dependencias haya más fácil es introducir cambios en el software.