Software design and modularity

¿Cuándo diseñar?

En un extremo encontramos el “big design up front”, se hace todo el diseño al principio del proyecto, antes de implementar nada, y tras implementarla se corrige el diseño. Por supuesto este enfoque no es muy conveniente. En el otro extremo está el no diseñar, solo desarrollar el código e ir arreglando los fallos que aparezcan, un “diseño por depuración”.

Lo ideal sería un proceso iterativo, un diseño inicial abarcando una complejidad que se pueda pensar, sin sobrediseñar, empezar el desarrollo tratando de construir lo establecido en el diseño inicial, rediseñar y ampliar lo especificado en el diseño inicial, etc. El diseño madura a medida que avanza el proyecto.

La complejidad

El diseño del software consiste en “gestionar la complejidad” (John Ousterhout), entendiendo como complejidad todo aquello relacionado con la estructura de un sistema software que lo hace difícil de entender o modificar.

Es un factor primordial, pues el mayor limitante a la hora de construir software es nuestra habilidad para entender los sistemas creados, que son demasiado grandes para una buena comprensión. Ya no es el tamaño de las memorias ni la velocidad de los procesos.

Es por esto que en el diseño del software estudiamos las formas de manejar la complejidad, de intentar reducirla, aplicar patrones que nos faciliten encapsularla o esconderla, etc.

Principios de diseño

En el libro *A Philosophy of Software Design*, John Ousterhout nos propone 15 principios de diseño:

1. Complejidad es incremental: “ignorar” los detalles pequeños.

   La complejidad no viene de un fallo fundamental que se pueda resolver y así eliminarla de un proyecto, sino de la acumulación de los pequeños detalles, por los que “no pasa nada”, pero que un programador introduce a un ritmo de 12 por semana multiplicado por los cientos de programadores de un proyecto. Es entonces cuando se alcanza una alta complejidad difícil de eliminar, esparcida en miles de puntos diferentes.

2. Código que solo funcione no es suficiente.

   Es importante que el código sea fácil de entender y mantener, y esté bien optimizado si no, podría llevar a problemas a la hora de actualizar o arreglarlo.

3. Hacer pequeños incrementos continuos para mejorar el diseño del sistema.

   Dedicar un 10% o un 20% del tiempo a intentar mejorar el diseño ahorrará tiempo de desarrollo en el futuro, podría verse como una inversión.

4. Los módulos deben de ser profundos.

Los módulos profundos hacen que el código sea más fácil de entender y mantener, pues al estar más definidos y enfocados en tareas específicas, es más fácil detectar errores y su funcionalidad queda más clara.

5. Las interfaces han de ser diseñadas para hacer que el uso más común sea lo más simple posible.

   La simplicidad en la interfaz puede reducir la complejidad y el costo del desarrollo, además de permitir reducir los costes asociados a cosas como el soporte técnico.

6. Es más importante que un módulo tenga una interfaz simple que una implementación simple.
7. Los módulos de propósito general son más profundos.

   Los módulos de propósito general suelen ser utilizados en diferentes partes del sistema y por diferentes personas, por tanto, tienen que ser más flexibles y tener mayor capacidad de adaptación a diferentes contextos, teniendo mayor profundidad y complejidad.

8. Separar código de propósito general y el de propósito específico.

   Permite una mayor modularidad y flexibilidad en el diseño del software, facilitando la reutilización de código y la solución de problemas de manera más eficiente.

9. Diferentes capas deben de tener diferentes abstracciones.

   Cada capa debe tener una interfaz claramente definida, y las capas inferiores deben ser menos abstractas que las superiores. Capas más bajas, detalles de bajo nivel. Capas superiores, lógica de negocio e interfaz de usuario.

10. Arrastrar complejidad.

    En lugar de propagar la complejidad en todo el código, hay que intentar limitarla en todo a una capa específica (capa de servicios).

11. Definir errores y casos especiales para eliminarlos.

    Al definirlos, se pueden anticipar como se comportará el sistema en esas situaciones específicas, diseñando así el software de manera más efectiva para manejarlas.

12. Diseñarlo dos veces.

    Diseñar el software al menos 2 veces, (uno de alto nivel, para los requisitos y arquitectura) y otro detallado (requisitos y arquitectura del código) antes de codificarlo ayuda a garantizar que el software se diseñe de manera completa y coherente desde el principio.

13. Comentarios deben describir cosas que no son obvias en el código.
14. Software debería ser diseñado con la idea de facilidad de lectura, no de escritura.
15. Los incrementos del desarrollo deben de ser abstracciones, no funcionalidades.

    Construir abstracciones que puedan ser reutilizadas en múltiples lugares en lugar de centrarse en agregar nuevas funcionalidades.

Red flags

A la hora de poder comprobar que estamos realizando un buen diseño, o comprobar si un código existente esta correctamente diseñado, podemos buscar por unas “red flags”. Algunas de las “red flags” más importantes mencionadas en el libro:

- **Modulo llano** (Shallow Module): la interfaz creada no es más simple que la implementación de esta.
- **Fuga de información** (Information Leakage): se usa el mismo conocimiento en dos o más sitios diferentes.
- **Descomposición temporal**: la estructura del sistema está basada en el orden en el que se ejecutan las operaciones.
- **Sobreexposición**: cuando una API está diseñada de tal manera que se obligue a los usuarios a conocer funcionalidades de uso poco frecuente para poder usar funcionalidades de uso frecuente.
- **Método Pass-Through**: un método que lo único que hace es pasar los argumentos a otro método. Significa que no hay una división de responsabilidades.
- **Repetición**: una pieza de código no trivial es repetida una y otra vez, quiere decir que no has encontrado la abstracción adecuada aún.
- **Mezcla especial-general**: cuando un fragmento de código de propósito general también contiene código específico al módulo donde se encuentra
- **Métodos conjuntos**: los métodos han de ser entendibles independientemente, sin necesidad de entender otros adicionales
- **Comentarios repitiendo código**: la información en los comentarios es algo que se puede inferir del propio código que está comentando.
- **Documentación de la implementación contamina la interfaz**: la documentación de la interfaz (como la de los métodos) describe datos de implementación que no son necesarios para usar lo implementado
- **Nombre impreciso**: el nombre de un método o variable puede significar diferentes cosas
- **Difícil escoger un nombre**: es difícil encontrar un nombre de una clase o método que pueda crear una imagen clara del objeto
- **Difícil de describir**: el código que acompaña a un método o variable ha de ser simple y completo.
- **Código no obvio**: el código ha de ser entendible con una lectura rápida