Diseño de software y modularidad

Arquitectura del Software

Universidad de Oviedo

Curso 2022/23

Francisco Coya Abajo – UO257239

Rubén Caño Domínguez – UO284647

Introducción

El desarrollo de software es un proceso de creación y modelado de un sistema software que satisfaga los requisitos funcionales de éste. Aplicar un buen diseño de software es una tarea compleja y con numerosos retos. Supone tomar decisiones (a veces no tan acertadas) que sean funcionales y efectivas. Este documento es una extensión de la presentación realizada en el seminario, donde se ahondará un poco más en profundidad los aspectos de diseño de software citados en ésta.

Motivación

John Ousterhout es profesor en la Universidad de Standford. El tema tratado se basa en el episodio número 512 del programa de radio Software Engineering Radio, en el que John habla sobre los principios de un buen diseño de software basándose en su libro "A Philosophy of Software Design".

Principios de un buen diseño de software

A continuación, se presentarán los principales aspectos a la hora de realizar un buen diseño del software.

Simplicidad

Se trata de minimizar la complejidad del sistema. Por ejemplo, si lo comparamos con la arquitectura (de edificios), es posible realizar diseños muy vanguardistas, que llamen mucho la atención, pero sean poco habitables. Por el contrario, un diseño estándar como puede ser un chalet o un bloque de pisos, donde se da más importancia al bienestar y comodidad de las personas. Es decir, no por realizar un diseño muy complejo, aplicando ("a calzador") muchos patrones, vamos a conseguir un producto mejor.

Claridad

Es una extensión de explicado en el apartado anterior. Un código claro, sin errores y fácil de entender el uno de los objetivos que todo desarrollador debería fijar. Hace que sea fácil de mantener y evita ambigüedades en el código escrito. Es necesario mantener un orden y unos estándares a la hora de trabajar en el equipo, pudiendo resolver incidencias de forma más rápida y eficiente. En consecuencia, se reducen los fallos y la deuda técnica. Para ello:

- Escribir código legible.
- Establecer un estándar o políticas de diseño antes de comenzar el proyecto.
- Mantener una buena organización.
- Pero, que el código sea fácil de entender no implica que sea fácil de programar. En ocasiones, puede ser un verdadero reto.

Abstracción para manejar la complejidad

La complejidad está presente en todo sistema de información. Identificar y analizar los aspectos más complejos es una tarea del ingeniero del software. Para ello, es necesario abstraer dicha complejidad para que el usuario final del producto no se vea afectado e, importante, para reducir la deuda técnica. Para ello:

- Ocultar detalles de la implementación. Programando para interfaces, no para la implementación.
- Si un diseño parece complejo, afróntalo desde otro punto de vista.
- Es muy importante conocer el dominio del problema y su contexto.
- Escribir (buena) documentación en aquellos puntos del código que así lo requieran.

Anticiparse al cambio

El software evoluciona constantemente. Analizar los puntos de mejora actuales, ayudará a que el producto evolucione de una manera favorable, sin demasiados obstáculos. Como informáticos, es necesario estar al día de las novedades del sector. Sin embargo, es imposible ser experto en todo. Identificar los beneficios y riesgos de utilizar el nuevo software es parte de las decisiones a tomar. Por ejemplo, para ayudar a afrontar el cambio es posible acudir a patrones de diseño y técnicas que se conocen.

¿En qué parte del desarrollo se realiza el diseño?

Para John hay tres métodos posibles:

- Modelo cascada:
    - Se trata de hacer todo el diseño final antes de empezar con la implementación
    - No suele funcionar bien, pues hay sistemas muy complejos en los que no es posible visualizar todas las consecuencias de las decisiones de diseño
- El otro extremo es empezar directamente con la implementación e ir diseñando sobre la marcha. Esto suele acabar en diseños muy pobres, lo que los acaba haciendo sistemas muy complejos de entender y ampliar.
- El diseño iterativo:

- Se hace un diseño inicial, y al implementar te darás cuenta de los errores que tiene el diseño, volverás a rediseñar, y así iras iterativamente hasta tener un diseño robusto.

John dice que el suele hacer unas 3 iteraciones de diseño, en el primero se da cuenta muy rápidamente de los errores, en el segundo mejora imperfecciones y el tercero ya suele ser un buen diseño.

Definir errores fuera de existencia

Definir errores fuera de existencia se refiere a que no siempre hay que manejar un error pues a veces consideramos como error algo que no lo es.

Un ejemplo de esto sería, al crear un método que elimine un archivo, es lógico pensar que hay que lanzar una excepción si recibe un nombre de un archivo que no existe, sin embargo, es más correcto pensar que si lo que tiene que hacer el método es eliminar un archivo y el archivo ya está eliminado porque no existe, entonces no es necesario lanzar una excepción, simplemente el método ya ha cumplido su trabajo y finaliza.

Sin embargo, tampoco se puede llevar esto al extremo de definir todos los errores fuera de existencia. John dice que después de hablarles de este principio a sus alumnos, empezaron a ignorar muchos errores importantes dejando bloques catch vacíos y justificándolo con que estaban definiendo errores fuera de existencia lo cual es una mala interpretación de este principio pues hay errores importantes que son imprescindibles manejarlos para detectarlos rápidamente en caso de que fallen.

Modularidad como forma de reducir la complejidad

El objetivo de los módulos es el mismo que el de la abstracción, dar una forma sencilla de pensar en algo complicado. Al encapsular mucha funcionalidad compleja dentro de un módulo estamos reduciendo la complejidad que tenía esa funcionalidad a la complejidad de la interfaz del módulo.

John identifica dos tipos de módulos:

- Módulos profundos: Son los módulos perfectos, encierran una gran cantidad de funcionalidad y su interfaz es muy pequeña
- módulos superficiales: Son módulos que tienen poca funcionalidad, pero una interfaz muy grande

Un gran ejemplo de modulo profundo serían los recolectores de basura, estos no tienen que ser llamados para hacer su trabajo, están por detrás del código sin hacer notar su presencia. Son un ejemplo de modulo que ni si quiera tiene una interfaz, y logra reducir la complejidad de liberar memoria en desuso a prácticamente 0.