Software design and modularity

Equipo Es2-02

Biografía de John Ousterhout:

John Kenneth Ousterhout nació el 15 de octubre de 1954 en EE.UU., es el presidente de Electric Cloud, profesor de ciencias computacionales de Standford y es el creador del lenguaje de secuencias de comandos Tcl y del conjunto de herramientas Tk.
También dirigió el equipo que diseño el sistema operativo Sprite y el primer registro de estructura del sistema de archivos, además de esto es el autor del libro “A Philosophy of Software Design” (del cual se habla en el episodio del podcast) y creador de la Dicotomía de Ousterhout (que consiste en una división básica de los lenguajes de programación en dos grupos lenguajes de programación de sistemas y lenguajes de scripting).

Metodologías de trabajo:

Como malas practicas se pueden identificar las siguientes:

- Programar sin preocuparse por el diseño:

Seria el peor caso y vendría a ocurrir cuando se empieza a hacer un proyecto sin pensar en el diseño de este

- Desarrollo basado en pruebas

No lo recomienda puesto que puede incitar a cumplir los test de forma separada sin preocuparse por el diseño (lo que haría que se acabe con un mal diseño) o que cuando se creen nuevas partes del código estas queden sin testear

- Diseñar primero todo el proyecto y programar después

Es cuando se diseña el proyecto entero primero antes de empezar a desarrollar generalmente poniendo ese diseño como algo inamovible

Como buenas prácticas menciona:

un diseño “iterativo” en el que sugiere ir intercalando periodos de desarrollo y de diseño en los que se empieza a desarrollar conforme a un diseño básico inicial y conforme vallan surgiendo problemas o nuevas ideas ir rediseñando/ampliando el diseño original

Empatía técnica/código

También habla del concepto de empatía técnica/código empático lo cual consiste en hacer tu código siguiendo un buen diseño que piense en el futuro para que este se pueda entender fácilmente por otros en el futuro y así no los entorpezcas, esto también es bueno para uno mismo ya que estarías haciendo que en el futuro también te sea más fácil trabajar con este

Principio del mártir:

Este principio consiste cuando un programador o un grupo de estos se encargan de mantener una parte muy compleja del código ocultando esta complejidad para facilitar la vida al resto de desarrolladores

En su contra se encontraría el desarrollo “perezoso”:

Esto es cuando se programa sin pensar que se va ha hacer en el futuro con ese código lo que seguramente va a causar problemas a otros desarrolladores e incluso a ti mismo o dicho de otra forma es cuando se programa con la mentalidad de “eso es problema del yo de mañana”

Tornado táctico:

Es un concepto que usan para referirse a las personas que en un proyecto hacen cualquier código funcional rápido y como sea sin preocuparse de hacer un buen diseño es la mentalidad de hacer cualquier cosa para conseguir algo de progreso aunque a la larga esto cause daño al sistema y genere deuda técnica, suelen hacer una gran parte del proyecto y irse con lo que luego otros tienen que ir a solucionar los problemas que introdujo y otro problema que tienen es que suelen ser “alabados” por los jefes debido a que “hacen su trabajo” en poco tiempo

Complejidad incremental:

Este concepto consiste en que cuando algo llega a ser innecesariamente/muy complicado suele ser debido a que se han ido acumulando pequeños fallos a lo largo del tiempo que normalmente de forma individual no supondrían ningún problema, lo que también causa que posteriormente sea mucho más difícil de solucionar puesto que ese problema no esta localizado en un solo trozo del código si no distribuido en pequeños fragmentos a lo largo de todo el código del proyecto, también es importante mantener la complejidad baja puesto que la complejidad en si misma es un factor limitante significativo en un proyecto ya que es necesario entender que se esta haciendo para poder hacerlo y si un proyecto tiene una complejidad demasiado alta se corre el riesgo a que se deje de entender el proyecto en si y que se quiere hacer

Complejidad hacia arriba:

Esto ocurre a que cuando se encuentra un problema no solucionarlo “ahí” y dejar que pase a la siguiente “capa” un ejemplo exagerado de esto podría ser que los de la capa de acceso a datos deleguen que base de datos usar a la capa de negocio, estos a la interfaz los cuales lo delegan en el usuario lo cual sería un mal diseño bastante claro

Complejidad hacia abajo:

Esto vendría a ser justo lo contrario a lo anterior y consiste en encapsular esa complejidad y evitar que se propague hacia “arriba” por ejemplo que los de la capa de negocio no tengan que saber que base de datos se usa ni como hay que acceder a esta para poder hacer su parte del trabajo, otro ejemplo también seria el principio del mártir

Errores/casos especiales:

En ocasiones los errores pueden ser “solucionados” y no tiene por qué parar la aplicación y lanzar una excepción o estos no tienen por qué ser un error real por ejemplo borrar un archivo/variable que no existe o en Linux que cuando borras un archivo en uso no te da error y simplemente” lo borra” falsamente para que no se vea y espera a que deje de estar en uso para borrarlo realmente.

También menciona que hay que tener cuidado de no meterlos en un try catch y no tratar estos errores y simplemente ignorarlos y en otras ocasiones puede ser interesante mostrar los errores al usuario

Modularidad:

EL diseño orientado a módulos consiste en ocultar la complejidad usando una interfaz sencilla para algo que en realidad es muy complejo o con mucha funcionalidad, mostrando así lo importante y ocultando lo que no es consiguiendo interfaces “finas y profundas” (pocos métodos mucha funcionalidad) en vez de “gordas y superficiales”

Comentarios en el código:

Comenta que para los comentarios considera malas prácticas no comentar nada, creer que hacer el código de forma que se entienda sin hacer comentarios o comentar excesivamente.

Como buenas practicas el recomienda comentar lo necesario y fundamental evitando comentar funciones/métodos que no lo necesiten por ejemplo una función que calcule Fibonacci