ARQUITECTURA DE SOFTWARE

Por Ricardo Cordonado

1. Introducción a la arquitectura de software:

a. Atributos de calidad / Requisitos no funcionales

i. ¿Qué es la calidad de software?

El término calidad de software se refiere al grado de desempeño de las principales características con las que debe cumplir un sistema computacional durante su ciclo de vida, dichas características de cierta manera garantizan que el cliente cuente con un sistema confiable, lo cual aumenta su satisfacción frente a la funcionalidad y eficiencia del sistema construido.

El concepto de calidad de software, según Pressman (2010) se asocia a la “concordancia con los requisitos funcionales y de rendimiento explícitamente establecidos con los estándares de desarrollo plenamente documentados y con las características implícitas que se espera de todo software desarrollado profesionalmente”.

El instituto de ingenieros eléctricos y electrónicos (IEEE, 1990) define calidad de software como “el grado con el que un sistema, componente o proceso cumple los requerimientos especificados y las necesidad o expectativas del cliente o usuario”

ii. ¿Cuáles son los atributos de calidad de un software?

Una de las cosas que aprendí de la asignatura “Teoría de sistemas”, fue que los sistemas relacionados con la elaboración de productos o servicios tienen a desarrollar atributos dependiendo de las demandas de sus consumidores en el tiempo, sin embargo, el atributo “calidad” suele reinar en cada uno de estos sistemas sobre cualquier otro atributo.

La siguiente imagen modela bien este fenómeno:

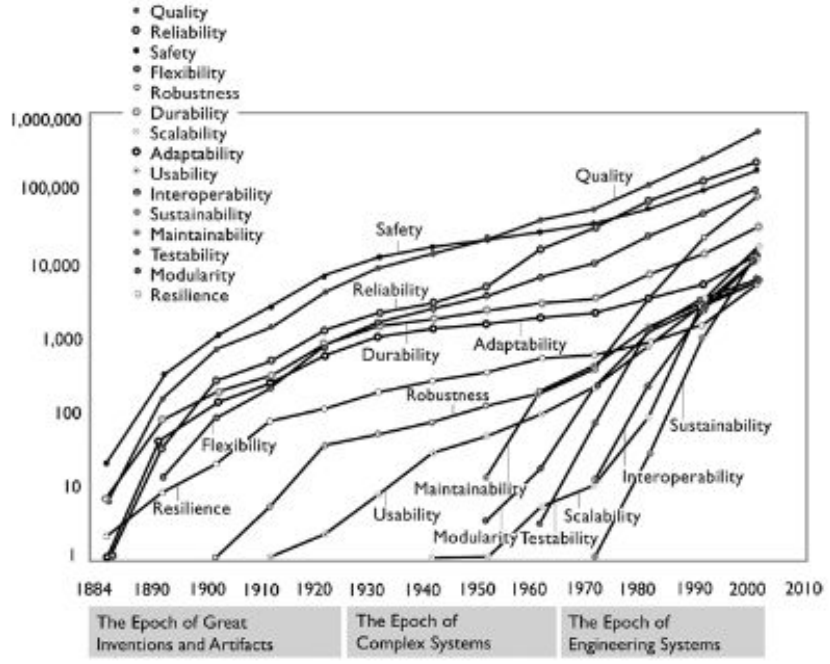

Figura 1: Número acumulativo de artículos de revistas en donde algún atributo (ility) aparece en el título o el abstracto. El eje de las abscisas presenta el año, mientras que el de las ordenadas presenta el número de artículos publicados, [Imagen sacada de General System Theory by Lars Skyttner]

Cuando hablamos de atributos de calidad de software nos referimos a que el software presente: Simplicidad, escalabilidad, mantenibilidad, extensibilidad, seguridad, interoperabilidad, fiabilidad, testeabilidad, etc.

Finalmente, entregar una lista con todos los atributos de calidad resultaría una tarea imposible, esto porque a medida que transcurre el tiempo surgen nuevos atributos de calidad. A su vez, la importancia de ciertos atributos es variante en el tiempo.

iii. ¿Es deseable cumplir con todos los atributos de calidad? ¿Por qué?

Ciertamente, la situación ideal sería cumplir con todos los atributos de calidad, de esta forma podremos tener un producto impecable, sin embargo, considero que el estándar de calidad del software es subjetivo al desarrollador, por esta razón es que existen modelos de calidad de software que evalúan la calidad a nivel de producto, de uso y proceso.
De esta forma, sometiendo nuestro software a estas pruebas, es posible cerciorarse de que nuestro producto satisface las expectativas de nuestros clientes y es compatible con los estándares de calidad.

iv. ¿Qué es un requisito no funcional? ¿Cuál es la diferencia con un requisito funcional?

Se entiende por requisito no funcional a todas aquellas propiedades del software que permiten el buen funcionamiento del sistema, como por ejemplo, una interfaz gráfica intuitiva que permita una interacción armoniosa entre el humano y computador (simplicidad e intuitividad). En este sentido, los requisitos no funcionales competen a todos aquellos atributos de calidad que deseamos que formen parte de nuestro software.
La diferencia entre un requisito no funcional y funcional radica en que el segundo responde a los requisitos de usuario y busca satisfacer un problema(s) o idea(s) en específico.

v. ¿Cuál es el rol del arquitecto de software?

El arquitecto de software tiene la responsabilidad global de dirigir las principales decisiones técnicas, expresadas como la arquitectura de software. Esto habitualmente incluye la identificación y la documentación de los aspectos arquitectónicamente significativos del sistema, que incluye las "vistas" de requisitos, diseño, implementación y despliegue del sistema.

El arquitecto también es responsable de proporcionar el fundamento de estas decisiones, equilibrando las preocupaciones de los diferentes interesados, reduciendo los riesgos técnicos, y garantizando que las decisiones se comunican, y validan con eficacia, y que se acatan.

b. OAuth2

i. ¿Qué es?

OAuth 2.0 proporciona flujos de autorización específicos para aplicaciones web, aplicaciones de escritorio y teléfonos móviles

[https://tools.ietf.org/html/rfc6749#section-1.2]

ii. ¿Quién lo utiliza?

Mayoritariamente es utilizado por redes sociales como Facebook, twitter, instagram o Google.

iii. ¿Por qué se utiliza?

Surge como respuesta ante el envío continuo de credenciales entre el cliente y servidor para la integración con aplicaciones de terceros.
Con OAuth2 el usuario delega la capacidad de realizar ciertas acciones, no todas, a las cuales da su consentimiento para hacerlas en su nombre, por ejemplo, publicar algún dato.

iv. ¿Cuáles son sus limitaciones?

1. En caso de ceder privilegios a algún sitio y este es víctima de vulnerabilidades de seguridad o ataque, nuestros datos estarán parcialmente expuestos.
2. Usuarios menos experimentados pueden ser víctimas de phishing.
3. En caso de logs, realizar una mantención continua frente a posibles actualizaciones de los sistemas interconectados.

2. Principios SOLID

A. ¿Qué son los principios SOLID?

SOLID es un acrónimo mnemónico introducido que representa cinco principios básicos de la programación orientada a objetos y el diseño. Cuando estos principios se aplican en conjunto es más probable que un desarrollador cree un sistema mantenible y escalable en el tiempo.

a. Principio de responsabilidad única (Single responsibility principle)

“Una clase debe tener solo una razón para cambiar”.

Establece que cada módulo o clase debe tener responsabilidad sobre una sola parte de la funcionalidad proporcionada por el software y esta responsabilidad debe estar encapsulada en su totalidad por la clase. Todos los servicios deben estar estrechamente alineados con esa responsabilidad.

b. Principio de abierto/cerrado (Open/closed principle)

Establece que una entidad de software debe quedar abierta para su extensión, pero cerrada para su modificación. Es decir, se debe poder extender el comportamiento de tal entidad sin modificar su código fuente.

c. Principio de sustitución de Liskov (Liskov substitution principle)

*“Si **S** es un subtipo de **T**, entonces los objetos de tipo **T** en un programa pueden ser sustituidos por objetos de tipo **S** (es decir, los objetos de tipo **S** pueden sustituir objetos de tipo **T**), sin alterar ninguna de las propiedades deseables de ese programa (la corrección, la tarea que realiza, etc.)”.*

Establece que cada clase que hereda de otra puede usarse como su padre sin la necesidad de conocer las diferencias entre ellas,

d. Principio de segregación de interfaz (Interface segregation principle)

Establece que los clientes de un programa dado solo deberían conocer de este aquellos métodos que realmente usan, y no aquellos que no necesitan usar.

e. Principio de inversión de la dependencia (Dependency inversion principle)

El principio establece:

1. Los módulos de alto nivel no deberían depender de los módulos de bajo nivel. Ambos deberían depender de abstracciones (p.ej., interfaces).
2. Las abstracciones no deberían depender de los detalles. Los detalles (implementaciones concretas) deben depender de abstracciones.

B. ¿Cuáles son los argumentos en favor y en contra de SOLID?

Para una mayor profundidad recomiendo visitar:
https://news.ycombinator.com/item?id=19955467

a. A favor

i. Hace que el código sea fácil de mantener en el futuro debido a que es fácil de entender, concreto, testable
ii. Fácil de extender
iii. Aplicar los principios profesionaliza el software
iv. Forma parte de los estándares de calidad de software
v. https://blog.ndepend.com/defense-solid-principles/
vi. https://www.entropywins.wtf/blog/2017/02/17/why-every-single-argument-of-dan-north-is-wrong/

b. En contra

i. Los principios SOLID son vagos o ambiguos
ii. SOLID implica complejizar el código y aumentar los tiempos en el proceso de desarrollo
iii. Los principios SOLID son utópicos
iv. SOLID es un truco de marketing
v. https://jamesmccaffrey.wordpress.com/2016/08/24/the-solid-design-principles-absolute-nonsense/

vi. https://speakerdeck.com/tastapod/why-every-element-of-solid-is-wrong/

C. ¿A qué requisitos no funcionales responden? ¿A qué requisitos no funcionales no responde?

a. Responde

i. Calidad
ii. Testabilidad
iii. Mantenibilidad
iv. Escalabilidad
v. Flexibilidad
vi. Extensibility
vii. Solidez
viii. Atomicidad

b. No responde

i. Seguridad
ii. Portabilidad
iii. Simplicidad
iv. Agilidad