Cualidades de Diseño

Versión 2.0
Abril 2014

Índice

1 Sobre este texto

1.1 Versiones

Autores principales	Versión	Fecha	Observaciones
Franco Bulgarelli, Juan Zaffaroni	2.0	Abril 2014	Integración, ampliación y reescritura del apunte
Rodrigo Merino Leonardo Gassman	1.1	2007	Versión original del apunte
Nicolás Passerini, Carlos Lombardi, Fernando Dodino	1.1	2005	Versión original del apunte de cualidades de software

1.2 Breve reseña histórica

Este apunte constituye la segunda gran versión del apunte de cualidades de diseño, que se viene desarrollando desde el año 2004, integrado con el apunte de Cualidades de Software.

Es decir, si bien se trata de una completa reescritura del mismo, es en definitiva apunte con más de 10 años de trabajo.

El mundo de la ingeniería de software, tan dependiente de la tecnología, se caracteriza por constante cambio, advenimientos de nuevos lenguajes, metodologías y tipos de problemas. Y lo interesante es que si bien los ejemplos y explicaciones de este apunte se han actualizado o modificado, las ideas centrales se mantienen intactas. Creemos que hay valor en ello: poder mirar un instante más allá del momento actual.

Porque, como verán, cualidades de diseño se trata no de recitar un catálogo de recetas, sino de pensar, imaginar y aplicar criterio.

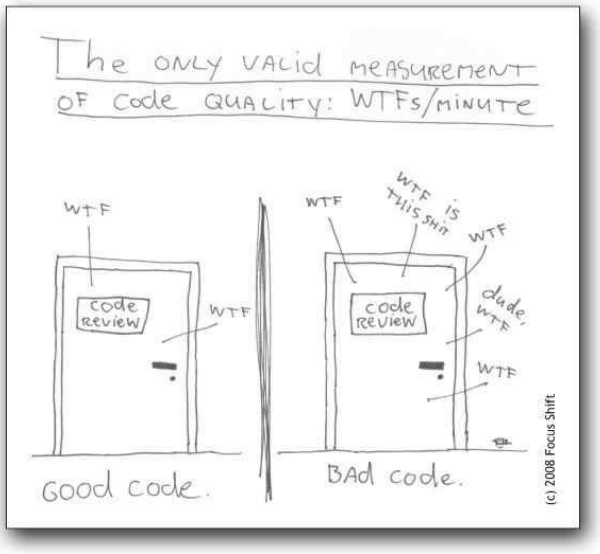
2 Introducción

2.1 Cualidades de diseño vs cualidades de software

En este apunte hablaremos de cualidades de diseño en términos generales, pero en particular las bajaremos a detalle en el contexto de la construcción de software. Es por ello que indistintamente las referiremos como cualidades de diseño o cualidades de software.

Queda para el lector hacer las extrapolaciones correspondientes al diseño de sistemas no software.

2.2 ¿Qué son las cualidades de diseño?



Cuando nos toca analizar un sistema, ya sea existente o aún tan sólo presente en nuestras mentes, frecuentemente nos encontraremos con aspectos de su diseño "nos hacen ruido": por ejemplo, muchas veces veremos partes difíciles de modificar, o excesivamente compleja, o que presentan abstracciones confusas.

Y si somos además responsables del uso, mantenimiento o construcción de dichos componentes, probablemente esto venga acompañado del recuerdo poco feliz de la familia de quien lo ideó.

Esto nos da una intuición de qué es la calidad del diseño. Lo que buscaremos ahora es justamente refinar esta idea[1], formar criterios más sofisticados sobre qué diferencia a un buen diseño de uno deficitario, que conoceremos como cualidades de diseño. Puesto en otros términos, nos ayudará a poder responder la pregunta: ¿es el diseño A mejor que B?

Lo interesante es que estos criterios nos permitirán analizar y tomar decisiones más formadas. No serán nuestras únicas guías, claro: el criterio, experiencia y conocimiento del Ingeniero de Software serán elementos clave. Por lo tanto, debemos interpretar a las cualidades de diseño como heurísticas antes que reglas.

Las cualidades de diseño, como veremos a continuación, estarán en general enunciadas en forma de principios más o menos genéricos, pero cuya interpretación será diferente según la tecnología empleada. Por ejemplo, analizar el acoplamiento entre dos componentes desarrollados bajo el paradigma funcional será diferente de hacerlo entre dos componentes desarrollados bajo el paradigma de objetos.

2.3 ¿Para qué sirven las cualidades de diseño?

Como adelantamos, las cualidades de diseño nos servirán para comparar diseños y tomar decisiones de diseño.

¿Y de qué forma nos ayudarán en estas tareas? Por un lado, expandirán nuestra mente y nos recordarán varios aspectos a tener en cuenta antes de tomar una decisión de diseño. Y por

[1] Diría Paenza: "la intuición es un músculo que se ejercita"

otro lado nos darán un vocabulario más rico que nos permitirá justificar mejor nuestras opiniones y decisiones.

Para hablar de cualidades de diseño, deberemos tener siempre un **diseño alternativo** en mente. Una pieza de software no es, por ejemplo, inherentemente simple, sino más simple que otra que resuelva la misma problemática.

Y en general también deberemos tener en cuenta **un contexto**: por ejemplo, un diseño para un componente no es más flexible que otro, sino más flexible ante un cierto escenario de cambio.

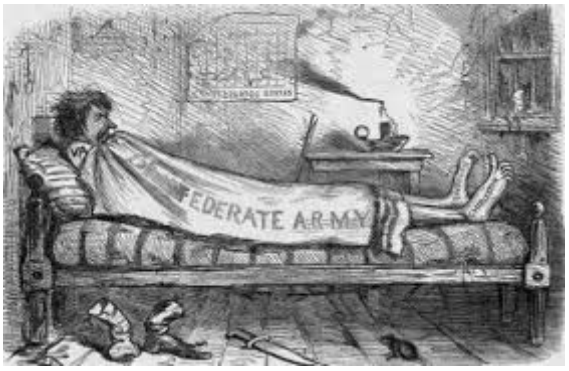
2.4 Tensiones entre las cualidades

Como veremos más adelante, muchas veces nos encontramos con que favorecer una cualidad de diseño en una solución perjudica a otra. O por el contrario, una mejora en una significa una mejora en otra.

Es decir, algunas cualidades tienen correlación positiva, como por ejemplo cohesión y abstracción, o desacoplamiento y testeabilidad.

Pero otras entran en conflicto y más bien, parecen oponerse.
Ejemplos: flexibilidad y robustez[2] (abrir vs. cerrar caminos, los deseos del usuario vs. el momento de implementar), o simplicidad vs. extensibilidad.

Es el síndrome de la frazada corta:



Estos casos son más interesantes, porque tenemos que elegir a cual cualidad le daremos preponderancia[3]. Y no sólo con conocimiento técnico sino también con conocimiento del negocio y del contexto humano del desarrollo.

Algunos ejemplos:

- si estamos construyendo un sistema provisorio cuyo tiempo de vida estimado es de 6 meses, no necesito pensar en la mantenibilidad.

[2] Didácticamente es inevitable la metáfora de la roca: la puedo ver como sólida, robusta, y como algo rígido, difícil de malear

[3] Diria un popular y soez cantautor, "*tenes que elegir mi amor, todo no se puede tener*"

- si estoy construyendo un prototipo para validar la idea de un producto o servicio, la simplicidad será clave.
- si soy responsable de un sistema de control de un avión que por diseño no va a cambiar, la flexibilidad no me parecerá importante pero la performance y robustez será crítica.
- si estoy desarrollando un sistema para un cliente, pero sé que en el futuro inmediato tendré que construir sistemas muy similares para otros clientes, probablemente valga la pena construir buenas abstracciones genéricas que me permitan reutilizar los componentes

2.5 Un ejemplo real

A modo de ejemplo, les presentamos la documentación del API de tiempo introducida en la versión 1.8 del Java Development Kit (JDK). Es interesante que los ingenieros que desarrollaron la misma se tomaron su tiempo para explicarnos no sólo qué hacen sus componentes sino sus decisiones de diseño.

1. Nos muestran cuales fueron las cualidades de diseño en las que más foco hicieron (notar ***extensiblidad****)*
2. Cómo repartieron las responsabilidades entre sus paquetes (con el objetivo de mantenerlos ***cohesivos***)
3. Cómo lograron una nomenclatura ***consistente***
4. Cómo construyeron ***buenas abstracciones*** que representen cada una de las posibles formas de tratar el tiempo (quizás a costa ***pérdida de simplicidad***)
5. Cómo proveyeron dos mecanismos para construir los objetos de la biblioteca: uno simple, y otro que permita la ***facilidad de prueba*** del código que use esta biblioteca.
6. Una comparación entre esta biblioteca la anterior versión de la misma (con un diseño diferente), en donde está implícita la idea de ***robustez*** (en el API anterior muchos errores no se reportaban tempranamente, por eso para esta nueva versión fue atacado aprovechando al sistema de tipos)

3 Cualidades que se pueden estudiar con cierta independencia tecnológica

Las primeras cualidades que atacaremos son aquellas que podríamos estudiar al comparar dos diseños con tan sólo un un nivel superficial o intermedio la tecnología sobre la que vamos a implementarlos.

3.1 Simplicidad

Dado que la idea de simplicidad es muy amplia, vamos a tomar la interpretación de KISS (Keep it simple, stupid) y YAGNI (You aren't gonna need it): no sobrediseñar, focalizándonos en las necesidades conocidas del sistema.

KISS → Muchas veces hay abstracciones que no son fundamentales, no surgen del negocio o su presencia no aporta a la solución. Lo que nos propone KISS es que cualquier complejidad **innecesaria** debería ser evitada.

YAGNI → Los requerimientos del hoy rara vez van a coincidir con los del mañana. Lo que nos propone YAGNI es no agregar funcionalidad nueva que no apunte a la **problemática actual**, es decir, no diseñar pensando en requerimientos en futuros hipotéticos, sino focalizarnos en las necesidades conocidas.

Esto es importante por dos motivos:

- **Por un factor económico**: Agregar funcionalidad no requerida para esa iteración, nos saca tiempo para hacer otras que sí lo son. Además, siempre está la posibilidad de agregar funcionalidad que no va a ser requerida en ninguna otra iteración, o que esté basada en conceptos que luego deberán ser cambiados. *Si hay algo peor que añadir funcionalidad prematuramente, es añadir funcionalidad que jamás será requerida, o que sea incorrecta.*
- **Por complejidad**: Agregar al modelo actual un requerimiento no solicitado, inyecta complejidad al mismo en la ventana de tiempo entre que se introdujo esa complejidad, y cuando realmente se precisó.

En definitiva, a medida que tengamos que mantener en nuestra mente más abstracciones para poder entender y predecir el comportamiento de un sistema, estamos ante diseños más complejos. La mayoría de los sistemas funcionan mejor cuanto más simples son, de ahí que:

- La Complejidad Accidental que proviene de nuestra propia solución (diseño) se debe evitar.
- La Complejidad Esencial propia del problema a solucionar, se debe manejar en nuestra solución de la forma más simple posible.

3.2 Robustez

La robustez nos dice que ante un uso inadecuado por parte del usuario, sistemas externos o ante fallas internas:

- El sistema no debe generar información o comportamiento inconsistente/errático.
- El sistema debe **reportar los errores** y volver a un estado consistente.
- El sistema debe facilitar tanto como sea posible la detección de la causa del problema.

Es decir, la robustez no se trata de evitar que un sistema falle, sino de la gracia con la que lidia con la situación excepcional.

Un principio que nos ayudará a mejorar esta cualidad en nuestras soluciones será el de Fail Fast (fallar rápido). Este nos propone que ante el indicio de un comportamiento incorrecto, el sistema debe abortar de forma ordenada la ejecución de su operatoria y reportar el error.

Fail Fast, entonces, minimizará las probabilidades de generar inconsistencias y facilitará encontrar la causa del problema (dado que el error se reportará próximo al momento y lugar en donde ocurrió), todo lo cual nos ayudará luego a volver a un estado conocido.

Esto contrasta con otra interpretación de robustez: “cuánta tranquilidad le da al usuario el uso de la aplicación”. Esto evidentemente no nos sirve para comparar tanto el diseño sino el producto final, dado que en esa sensación hay factores externos al mismo: calidad de la implementación, defectos en las tecnologías empleadas, el tiempo que se ha invertido en probar el sistema, y los propios prejuicios del usuario. Por tanto, esta interpretación no nos será de gran utilidad al diseñar.

3.3 Flexibilidad

Capacidad de reflejar cambios en el dominio de manera simple y sencilla. Podemos verlo en dos ejes: extensibilidad y mantenibilidad.

3.3.1 Extensibilidad

Es la capacidad de agregar nuevas características con poco impacto.

3.3.2 Mantenibilidad

Es la capacidad de modificar las características existentes con el menor esfuerzo posible.

3.4 (Des)acoplamiento

El acoplamiento es el grado de dependencia entre dos módulos/componentes, es decir, es el nivel de conocimiento que un módulo tiene sobre otro. Pensemos que cuanto mayor sea el acoplamiento, los cambios o errores de un módulo repercutirán en mayor medida sobre el otro módulo.

Buscaremos minimizar el acoplamiento para:

- Mejorar la mantenibilidad
- Aumentar la reutilización
- Evitar que un defecto en un módulo se propague a otros, haciendo dificultoso detectar dónde está el problema.
- Minimizar el riesgo de tener que tocar múltiples componentes ante una modificación, cuando sólo se debería modificar uno.

3.5 Facilidad de prueba (Testeabilidad)

La testeabilidad de un sistema nos permite asegurar que el código funciona correctamente y es mantenible. Verificar la testeabilidad de componentes pequeños, ayudará a mejorar el sistema en general.
Temas relacionados para ver durante la cursada:

- Pruebas Unitarias
- TDD

3.6 Cohesión

Un módulo o componente cohesivo tiende a tener todos sus elementos abocados a resolver el mismo problema. Puesto en otras palabras, la cohesión se trata de cuántas responsabilidades tiene el componente: cuantas más sean, menos cohesivo será.

En el caso de objetos, podemos ver fácilmente cuando un objeto o clase tiene dos métodos que apuntan a resolver, cada uno, tareas diferentes.

3.7 Abstracción

Podemos atacar a la idea de abstracción en, al menos, dos ejes: su calidad y su cantidad.

Por un lado, construir buenas abstracciones que definan metáforas consistentes y que encajen con nuestros modelos mentales sobre la realidad. Dicho informalmente, que la abstracción “cierre”, no “nos genere ruido”

Cuando tenemos mejores abstracciones estamos maximizando dos cualidades de diseño más:

- **Reusabilidad**: posibilidad de utilizar un módulo/componente construido anteriormente para resolver un problema nuevo.
- **Genericidad**: poder utilizar un módulo/componente definido anteriormente que se puede aplicar para resolver problemas distintos.

Ejemplo:

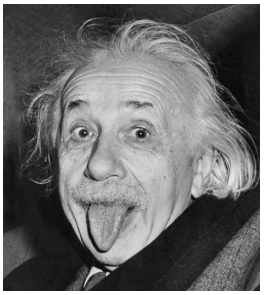
Una estructura de datos fundamental es la Pila, la cual es muy poderosa por su simplicidad, pero también por su proximidad al mundo real: un contenedor en el cual coloco y saco elementos por arriba, como en los portamonedas con resorte

Sus dos operaciones fundamentales son apilar y desapilar: push y pop. Qué pasaría si modeláramos una pila con un objeto, que entienda los mensajes push y pop, pero además le dieramos el método `insert(position, element)` *?*

Nuestra abstracción dejaría de “cerrar”, no porque haya perdido cohesión (push e insert son dos métodos orientados a lo mismo: agregar elementos al contenedor) sino porque la operación de inserción en un una posición arbitraria deja de encajar con la idea de una pila.

Parece un ejemplo tonto, pero miren como está modelada la clase Stack (deprecada en la práctica) de Java desde sus primeras versiones (presten atención a la herencia).

Por otro lado, podemos ver la cualidad de abstracción según cuántas de las abstracciones presentes en el modelo de negocio también están presentes en nuestra solución. Lo que vamos a buscar es que todas las abstracciones fundamentales del negocio que estamos modelando estén presentes, es decir, no perder abstracciones en el camino del diseño y construcción del sistema.



Acá estamos entrando en una aparente contradicción con la cualidad de simplicidad: parecería que por un lado planteamos maximizar la cantidad de abstracciones, y por el otro, minimizarla. La clave está en identificar cuales son abstracciones fundamentales para el diseño de la solución, y cuales son prescindentes (complejidades accidentales). Idea que queda resumida en el principio de "*Make things as simple as possible, but not simpler*"

Pero para eso no hay reglas, está en el **criterio de quien diseñe**, y eso no se puede enseñar, sino sólo ejercitar.

Recuerden: **no hay recetas para diseñar.**

3.8 Consistencia

Un diseño es consistente cuando ante problemas de similares, se tomaron decisiones de diseño similares. Se trata de aplicar los mismos criterios uniformemente a lo largo del diseño, haciéndolo más predecible para el lector ocasional y facilitando su comprensión.

3.9 Redundancia mínima

Un diseño presenta redundancia cuando el mismo conocimiento está presente en múltiples lugares, ya sea porque contempla múltiples mecanismos orientados a realizar la misma tarea, o porque la información que el sistema mantiene se encuentra directa o indirectamente duplicada.

Esto es un problema, porque:

- En el caso de la repetición de lógica entre diversos componentes, esta hace que cambiar el comportamiento del sistema sea más difícil, cometer errores sea más fácil y rastrearlos, más difícil.
- En el caso de la repetición de información, esta posibilidad la introducción de inconsistencias en los datos.

Entonces buscaremos minimizar la redundancia en la lógica entre los componentes de nuestro sistema, lo cual asociaremos al principio de **DRY** (*Don't repeat yourself*) / *Once and only once* y la redundancia entre nuestra información, lo cual asociaremos al proceso de **normalización**.

Puesto en otros términos, no será suficiente con crear buenas abstracciones y usarlas de forma consistente siempre que corresponda, sino además, deberemos evitar el solapamiento entre las mismas: **el conocimiento debe estar en un sólo lugar**.

3.10 Mutaciones controladas

Cuanto menos cambio de estado presentan mis componentes mientras el sistema se encuentra en funcionamiento, más fácil resulta razonar sobre el mismo[4]: podemos compartir, descartar o reemplazar a los componentes más fácilmente, y en general, minimizamos la probabilidad de cometer errores.

Por eso, un diseño que tiene más control sobre las mutaciones (es decir, las circunscribe y emplea sólo cuando son necesarios) es mejor que aquel que no lo hace.

Algunos principios derivados de esta idea general son:

- Favorecer la inmutabilidad: si es posible, diseñar los componentes del sistema de forma tal que sean inmutables, es decir, libres de cualquier tipo de cambio de estado interno. Si bien no es posible diseñar un sistema completamente libre de mutaciones, si es posible y valioso diseñar partes del mismo que sean inmutables.
- Minimizar la mutabilidad: Aún si mis componentes son mutables, realizar las mutaciones sólo cuando realmente es necesario, y no exponer en sus interfaces operaciones mutables que los requerimientos no justifiquen.

4 Cualidades que sólo se pueden estudiar con profundo conocimiento de la tecnología

A continuación trataremos algunas cualidades de diseño que podemos estudiar solo conociendo la tecnología y arquitectura sobre la que vamos a trabajar con profundo detalle.

4.1 Seguridad

Un sistema seguro debe impedir que agentes (personas / sistemas) externos no autorizados realicen acciones sobre el mismo, ó que agentes externos autorizados realicen acciones no permitidas sobre el mismo.

Analizar la seguridad del diseño (e implementación) es algo absolutamente dependiente de la tecnología. Por ejemplo, la forma de lograr un *exploit* es totalmente diferente en C, en Java o Ruby. El diablo está en los detalles.

Por ejemplo, el *bug* Heartbleed, presente en algunas versiones de la biblioteca de encriptación OpenL, permitía a un usuario malicioso robar información de claves de aquel sistema que utilizaba esta biblioteca. Este bug es del tipo desbordamiento de búfer, que si bien puede darse en cualquier tecnología, son mucho más fáciles de introducir en sistemas implementados en C (como lo era esta biblioteca).

El diseño de OpenSSL es en este sentido defectuoso, porque no contempla mecanismos que ayuden a evitar este tipo de situaciones **en esta tecnología**, como podemos ver incluso en uno de los *fixes* al *bug*.

[4] Controlas las mutaciones se trata en realidad de una idea más general: controlar los efectos (efectos colaterales)

4.2 Escalabilidad

Facilidad con la que un sistema pensado para una determinada carga puede ser adaptado para soportar una carga mayor.

Por ejemplo: *Caso caída de Whatsapp justo luego de la compra por Facebook, la gente se volcó ese día a darse de alta en Telegram que no estaba preparada para soportar esa carga toda junta. En definitiva se cayó también Telegram ese mismo día. Es decir, Telegram no estaba preparado para escalar a ese ritmo*

4.3 Eficiencia/Performance

Se trata de evaluar cuán buen uso hace el sistema de los recursos disponibles, el sistema tiene mejor eficiencia si requiere menos recursos para realizar una determinada tarea. Podemos pensarlo en tres niveles:

- Recursos Humanos necesarios para la construcción del sistema (hs. hombre)
- Recursos Humanos para la ejecución del sistema (hw / tiempo usuario)
- Recursos Hardware necesarios para la ejecución del sistema (memoria, procesador, almacenamiento)

La tercera acepción de eficiencia (recursos Hardware) es la más tradicional, sin embargo hoy hay menos interés en la misma que hace 20 ó 30 años, dado que el hardware es más barato, lo que lleva a que haya un mayor interés en la cualidad de escalabilidad.

5 Conclusiones

Como ya hemos señalado anteriormente, no existe un proceso mágico que me lleve a un buen diseño y un buen software. No hay reglas automatizables para tomar decisiones de diseño, y es justamente allí donde ponemos nuestro valor agregado.

Entonces, estas cualidades de diseño son sólo algunas heurísticas comúnmente aceptadas. Pero no son por nada del mundo reglas ni algoritmos que podamos seguir ciegamente. Nos deben servir para expandir nuestra mente, no para cerrarla.

El resto de los criterios para decidir los pondrán ustedes, en base a la experiencia que adquieran allá afuera.

6 Bibliografía

- Beautiful Architecture, Diomidis Spinellis, Capítulo 2
- Is Design Dead?, paper de Martin Fowler

- Effective Java, Joshua Bloch, Minimize mutability