Cualidades de Diseño

Heurísticas para la toma de decisiones en la construcción de sistemas informáticos

Versión 3.0
Abril 2024

Índice

1 Sobre este texto

1.1 Versiones

Autores principales	Versión	Fecha	Observaciones
Franco Bulgarelli	3.0	Abril 2024	Ampliación de cualidades
Franco Bulgarelli, Federico Aloi	2.1	Septiembre 2021	Correcciones y mejoras de redacción
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 Contexto

Este texto constituye la tercera gran versión del apunte de Cualidades de Diseño: *Heurísticas para la toma de decisiones en la construcción de sistemas informáticos*, que se viene desarrollando desde el año 2004. El mismo, a su vez, ya había sido reescrito y ampliado en 2014, integrando en éste el apunte de Cualidades de Software. En otras palabras, este es un texto con casi 20 años de desarrollo.

En la actual versión se han incorporado cualidades nuevas (*trazabilidad* y *facilidad de uso*), aumentado las referencias bibliográficas y también se ha buscado vincular a las mismas con cuestiones que van más allá de lo exclusivamente técnico, incorporando nociones sociológicas como poder y opresión, provenientes de los enfoques decoloniales y feministas.

Ya en la segunda versión del apunte se reconocía que:

> *"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."*

Sin embargo, la afirmación anterior esconde una importante omisión: el software responde también a estructuras sociales, las crea, modifica, refuerza e invisibiliza. De allí que sea también necesario ponerlas de manifiesto para tener un mayor entendimiento del impacto de nuestras decisiones de diseño.

Esta observación no es novedosa y se puede ya entrever en las ideas germinales del software libre[1] [2] [3]. Pero la intención de las modestas propuestas de vinculación a continuación es ir más allá, para analizar de forma crítica todas las producciones de software, aún de código libre o abierto. Se busca dar un puntapié a la reflexión sobre cuál es el fin del software en sí, las condiciones bajo las que se produce, los paradigmas culturales en los que se ancla y los mensajes que nuestras abstracciones replican.

En este sentido, buscamos enmarcar la discusión de las cualidades bajo dos paradigmas complementarios: el de **opresión algorítmica**[4] y el de **pedagogía de la crueldad**[5].

Por un lado, la *opresión algorítmica* se trata de reconocer el poder de los algoritmos en la era del neoliberalismo y cómo las decisiones que toman refuerzan relaciones sociales opresivas y habilitan formas de perfilamiento racial, creando, enmascarando y profundizando condiciones de inequidad económica, racional y de género. Este concepto plantea que la omnipresencia de los algoritmos y el software, tanto en formas visibles como invisibles a la persona común,

[1] https://www.gnu.org/philosophy/shouldbefree.es.html
[2] https://www.gnu.org/philosophy/freedom-or-power.es.html
[3] https://www.gnu.org/philosophy/imperfection-isnt-oppression.en.html
[4] Safiya Umoja Noble, Algorithms of Oppression: How Search Engines Reinforce Racism
[5] Rita Segato, Contra-pedagogías de la Crueldad, Segunda Edición

demanda la inspección precisa de los valores que estos sistemas de tomas de decisiones priorizan.

Por otro lado, la *pedagogía de la crueldad* refiere a las prácticas que habitúan a convertir la vida y lo vital en cosas, estériles, inertes, mensurables, vendibles, comprables y obsolescentes. Entre estas prácticas se incluyen la precarización del empleo, el trabajo servil y esclavo, la depredación de los territorios comunales a fines extractivos, la obsolescencia tecnológica programada. Estas prácticas se enmarcan en el proyecto histórico del capital, la producción de individuos y su transformación en cosas, y se anteponen al proyecto histórico de los vínculos, que instan a la reciprocidad y a la generación de comunidades.

Al hacerlo, buscamos habilitar algunas preguntas: ¿a qué proyecto histórico responde el software que utilizamos? ¿Cómo debería ser el software que vamos diseñar? ¿Cómo podemos, desde nuestro rol en el diseño del software, combatir estas formas de opresión?

Esperamos entonces que el estudio de las cualidades de diseño, lejos de presentar un catálogo de recetas, rígidas, inapelables, inmutables, sea una invitación a pensar, imaginar, aplicar criterio y reflexionar críticamente, sobre cómo, qué, para qué y para quién vamos a diseñar y construir sistemas informáticos.

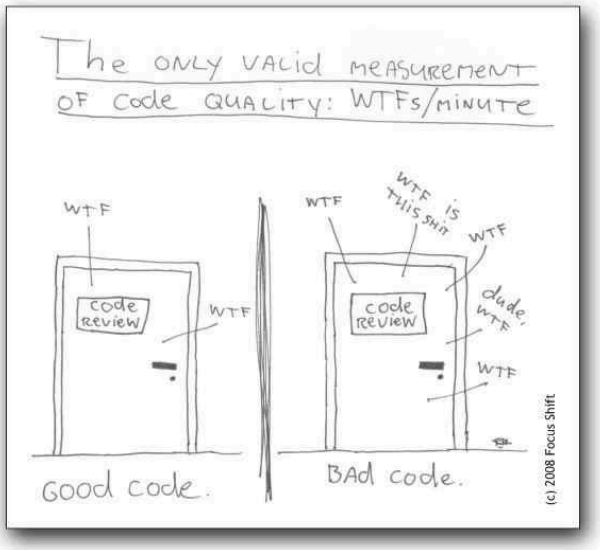
2 Introducción

2.1 Cualidades de diseño y 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.

Muchas de ellas podrán ser extrapoladas al diseño de sistemas no informatizados, pero dejamos dicha tarea a la persona lectora del apunte.

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 que “nos hacen ruido”: por ejemplo, muchas veces veremos partes difíciles de modificar, o excesivamente complejas, 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 grato de quien lo ideó.

Esto nos da una intuición de qué es la calidad del diseño: *maximizar la habitabilidad del espacio del software, ya sea en tanto personas constructoras o usuarias del mismo.* Lo que buscaremos ahora es justamente refinar esta idea[6], 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 de quien construya el software serán elementos clave. Por lo tanto, debemos interpretar las cualidades de diseño como *heurísticas* antes que reglas: nociones generales, imprecisas, que podemos usar como estrategias para buscar una solución dentro de un espacio de diseño con una gran cantidad de grados de libertad.

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 y la problemática a resolver. Por ejemplo, analizar el acoplamiento entre

[6] “la intuición es un músculo que se ejercita”: https://www.pagina12.com.ar/26399-la-paradoja-de-los-cajones-de-bertrand

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?

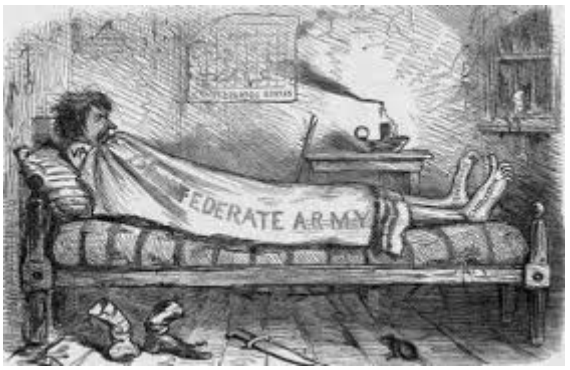
Como adelantamos, las cualidades de diseño nos servirán para comparar diseños, tomar decisiones sobre ellos y comunicar nuestros razonamientos a otras personas.

¿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 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 a continuación, muchas veces una mejora en una cualidad de diseño significa una mejora en otra. En otras ocasiones, sin embargo, nos encontramos con el escenario contrario: favorecer una cualidad de diseño en una solución perjudica a otra. Es decir, algunas cualidades, en contextos particulares, tendrán una correlación positiva, mientras que otras tienen correlación negativa.



Se trata del síndrome de la frazada corta. Estos casos son más interesantes, porque tenemos que elegir a qué cualidad le daremos preponderancia. Y no sólo con conocimiento técnico sino también del negocio y del contexto humano del desarrollo.

Veamos algunos ejemplos:

- Si estamos construyendo un sistema provisorio cuyo tiempo de vida estimado es de apenas algunos meses, la mantenibilidad no será una prioridad.
- Si estamos construyendo un prototipo para validar la idea de un producto o servicio, la simplicidad será clave.
- Si somos responsables del sistema de control de un avión, probablemente la flexibilidad será menos importante que el rendimiento y robustez.
- Si estamos desarrollando un sistema para un cliente y sabemos que en el futuro inmediato deberemos construir sistemas muy similares para otros, probablemente valga la pena construir buenas abstracciones genéricas que nos 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 quienes 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 ***extensibilidad****)*
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 nivel de conocimiento 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 basarnos en los conceptos de KISS (Mantenelo simple, o *Keep it simple* en inglés) y YAGNI (No vas a necesitarlo, o *You aren't gonna need it*)[7]: 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.

Todo esto 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.

[7] https://www.martinfowler.com/articles/designDead.html

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 intrínseca** (o esencial) propia del problema a solucionar, se debe manejar en nuestra solución de la forma más simple posible.

Es importante resaltar que siempre que simplificamos un problema, estamos haciendo un recorte de la realidad. Al extraer los aspectos esenciales de un concepto, estamos tomando la decisión de jerarquizar la información, determinando que algo es importante y otra cosa no. Pero esta decisión, aunque operativamente deseable y valiosa, no deja de pasar por lentes subjetivos y culturales. Para personas diferentes, con experiencias diferentes, lo que a unos ojos es nimio, para otros puede ser fundamental.

¿Cuál es la verdadera realidad, entonces, cuya esencia buscamos extraer? Más que buscarla en una mirada particular, podremos encontrarla en las intersubjetividades. A partir de allí proponemos la cualidad del **pluralismo**[8], es decir, la diversidad de voces, en los equipos de trabajo y el proceso de diseño.

3.2 Flexibilidad - Extensibilidad - Mantenibilidad

Existen varias definiciones posibles del concepto **flexibilidad**; aquí la interpretaremos como la medida de la facilidad con que el software puede adaptarse a cambios en los parámetros del diseño[9]. En este apunte, partiremos el concepto en dos ejes: extensibilidad y mantenibilidad:

- **Extensibilidad**: es la capacidad de agregar nuevas características con poco impacto.
- **Mantenibilidad**: es la capacidad de modificar las características existentes con el menor esfuerzo posible.

3.3 Robustez - Explicabilidad - Transparencia

La **robustez** nos dice que ante un uso inadecuado por parte de le usuarie, 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. El software no debe ser inmune a fallos, pero sí adaptarse a situaciones poco probables.[10]

[8] https://data-feminism.mitpress.mit.edu/pub/j8sdluic/release/1
[9] What Does It Mean to Be Flexible?, Object Design: Roles, Responsibilities and Collaborations, Rebecca Wirfs-Brock
[10] Understanding the Consequences of Failure, Object Design: Roles, Responsibilities and Collaborations, Rebecca Wirfs-Brock

Un principio que nos ayudará a mejorar esta cualidad en nuestras soluciones será el de **fail fast**[11] (*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.

Otra interpretación posible de robustez nos habla de “cuánta tranquilidad le da a le usuarie el uso de la aplicación”. Esta otra interpretación incluye otros aspectos que van más allá del diseño en sí, como por ejemplo, la calidad de la implementación, defectos en las tecnologías empleadas, el tiempo que se ha invertido en probar el sistema, y los propios prejuicios de le usuarie.

En cualquier caso, la robustez aborda la capacidad de los sistemas de presentar comportamientos consistentes y predecibles en situaciones de error, o en otras palabras, que los mismos informen adecuadamente qué los lleva a fallar: qué se esperaba, que se obtuvo, qué condiciones no se cumplieron.

Sin embargo, también resulta importante que los sistemas sean capaces de describir sus comportamientos en cualquier escenario, ya sea anómalo o esperado. Existen dos cualidades asociadas que abordan esta cuestión y que debemos observar:

- **explicabilidad**[12]: se trata del grado en que los sistemas de inteligencia artificial pueden fundamentar, ante cualquier persona, las decisiones que toman, o en otros términos, el grado en que se puede predecir cuál es el resultado que arrojarán ante una cierta entrada;
- **transparencia algorítmica**[13] [14]: se trata del grado en que la descripción de las reglas de negocio de un sistema están a disposición de cualquier parte interesada, a fines de poder monitorearlo, chequearlo, criticarlo o intervenirlo[15]. El software de código abierto, evidentemente, representa un paso hacia adelante en esta cualidad, aunque no una garantía por sí misma.

[11] https://www.martinfowler.com/ieeeSoftware/failFast.pdf
[12] https://dl.acm.org/doi/pdf/10.1145/3531146.3533090
[13] https://www.consejotransparencia.cl/wp-content/uploads/estudios/2020/10/Transparencia-Algoritmica.pdf
[14] https://institucional.us.es/revistas/Ius_Et_Scientia/VOL_7-1/Art_19.pdf
[15] “the disclosure of information about algorithms to enable monitoring, checking, criticism, or intervention by interested parties”: https://www.tandfonline.com/doi/abs/10.1080/21670811.2016.1208053

3.4 (Des)acoplamiento

El **acoplamiento** es el grado de dependencia entre dos componentes, es decir, es el nivel de conocimiento que un módulo tiene sobre otro. Podemos pensar 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 Validación - Facilidad de prueba (Testeabilidad) - Rendición de Cuentas

La **validación** de un sistema nos permite asegurar que el código funciona correctamente. Cuánta más cobertura presente el software, es decir, cuanto más probado ha sido, ya sea de forma automática o manual, más confianza tendremos en que estamos entregando un producto adecuado.

Por otro lado, la **facilidad de prueba** nos habla de cuán sencillo es incluir nuevas pruebas, y de que las mismas pueden evolucionar conforme las reglas del negocio evolucionan. En otros términos, de nada sirve que el sistema cuente con un alto grado de cobertura, si al actualizar dichas pruebas luego de una modificación a la lógica del mismo se torna imposible.

La facilidad de pruebas es una cualidad importante, dado que probar el software con rigurosidad antes de entregarlo a les usuaries es una actividad fundamental de la construcción de software. No se trata tan sólo de una técnica para optimizar el proceso de desarrollo, maximizar la calidad del producto generado o sus beneficios comerciales, sino de un *deber ético* de, como profesionales, ser responsables de detectar, corregir y reportar los errores en el software[16] [17].

Sin embargo, no debemos limitar el concepto de responsabilidad a una actitud individual, sino incluirla como una cualidad más del proceso de diseño: la **rendición de cuentas** (*accountability*), que trata de comunicar y participar a todas las partes en las decisiones de diseño, velando, en igual medida, por las necesidades de clientes, usuaries y personas afectadas por el sistema informático[18] [19]. Sólo así, al momento de construir pruebas, estaremos validando de forma ética y efectiva el software construido.

[16] https://ethics.acm.org/code-of-ethics/software-engineering-code/
[17] https://slate.com/human-interest/2018/04/mark-zuckerberg-is-not-a-child.html
[18] https://www.fastcompany.com/3066631/software-is-politics
[19] https://www.researchgate.net/publication/221248162_Designing_for_accountability

3.6 Cohesión

Un módulo o componente cohesivo tiende a tener todos sus elementos abocados a resolver el mismo problema[20]. 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. Podríamos incluso pensarlo a nivel de cada método, analizando cuántas tareas resuelve.

3.7 Abstracción - Reusabilidad - Genericidad - Humanización

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

Por un lado, la calidad de la abstracción se refiere al grado en que las abstracciones que definimos proponen metáforas consistentes y que encajan con nuestros modelos mentales sobre la realidad. Dicho informalmente, que la abstracción “cierre” y no “nos genere ruido”. Cuando tenemos abstracciones de buena calidad 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.

Por 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 es posible colocar y sacar elementos por arriba, como en un portamonedas.

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 diéramos el método `insert(position, element)`[21]?

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

[20] Beautiful Architecture, Diomidis Spinellis, Capítulo 2
[21] http://docs.oracle.com/javase/8/docs/api/java/util/Stack.html

Por otro lado, podemos entender la cualidad de abstracción según cuántas de las abstracciones presentes en el modelo de negocio también están presentes en nuestra solución. Desde este punto de vista, 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, entonces, en identificar cuáles de ellas son fundamentales para el diseño de la solución, y cuales son prescindentes (complejidades accidentales). Es que, en última instancia, crear abstracciones también significa simplificar: enfatizar unos aspectos y silenciar otros.

De todas formas, en este punto resulta conveniente realizar una advertencia: cuando dichas abstracciones refieren específicamente a personas, debemos tomar recaudos adicionales. Reducirlas a meros componentes, convertirlas en cosas, mercancías, commodities o elementos fácilmente reemplazables resultará peligroso. De hecho, ésta es una idea central de las economías de plataformas, de las que Amazon Mechanical Turk fue pionera y que aplicaciones como Uber popularizaron (de hecho, da origen al término *uberización*), en las que existe una llamativa correspondencia entre las abstracciones elegidas y las condiciones laborales[22] [23].

Por eso, en el proceso de diseño, resulta una cualidad fundamental rechazar la **deshumanización de los individuos**[24]. Es decir, debemos procurar que nuestras abstracciones y reglas de negocio no caigan en tales reduccionismos, y tener presente, en todo momento, que detrás de cada objeto, registro de base de datos o pieza de información de este tipo, hay una persona.

3.8 Consistencia

Un diseño es consistente cuando ante problemas de diseño similares, se toman 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.

---

[22] https://compass.onlinelibrary.wiley.com/doi/full/10.1111/soc4.13028

[23] https://www.researchgate.net/publication/339181813_AS_RELACOES_DE_TRABALHO_NA_ECONOMIA_GIG_E_O_IMPACTO_NA_UBERIZACAO_DO_TRABALHO_WORKING_RELATIONSHIPS_IN_THE_GIG_ECONOMY_AND_THE_IMPACT_ON_UBERIZATION_OF_WORK

[24] https://data-feminism.mitpress.mit.edu/pub/s7g5wna3/release/1

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 Almacenamiento mínimo - Recopilación de contradatos

Al diseñar es importante no **almacenar información irrelevante** para el problema, tanto a fines de reducir el espacio de almacenamiento como para resguardar la privacidad, proteger de los datos personales y minimizar el impacto de fallas de seguridad.

Sin embargo, se debe considerar también que sin datos no es posible fundamentar percepciones, tomar decisiones ni realizar cambios. Muchas veces la violencia se ejerce, justamente, a través del rechazo a recopilar información cuando ésta busca denunciar estructuras de poder existentes. Bajo este lente, en ocasiones sí nos interesará que nuestro sistema sea capaz de recopilar **contra-datos** de género, raza, etc, no a fines identificatorios o de personalización, sino estadísticos para la toma de decisiones.[25]

3.11 Mutaciones controladas

Cuanto menos cambios de estado presenten nuestros componentes durante la ejecución del sistema, más fácil resulta razonar sobre el mismo[26]: 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 necesarias) 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

[25] https://data-feminism.mitpress.mit.edu/pub/o6w62eho/release/1

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

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[27]:** aún si nuestros 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 requieren de conocimiento de la tecnología para su estudio

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

4.1 Seguridad - Protección de datos personales - Soberanía

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

Analizar la **seguridad** del diseño (e implementación) de forma cabal require conocimiento de la tecnología. Por ejemplo, la forma de lograr un *exploit* es totalmente diferente en C, en Java o Ruby.

Por ejemplo, el *bug* Heartbleed, presente en algunas versiones de la biblioteca de encriptación OpenSSL, permitía a un programa 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*.

Sin embargo, esta interpretación restrictiva del concepto seguridad es insuficiente. ¿Qué sucede si es el propio sistema, por diseño, el que comparte información con otros, aún sin el conocimiento o consentimiento de sus usuaries? ¿Qué protección tenemos ante el escenario en que quien opera el sistema decide negar el acceso a usuaries individuales, a colectivos o estados? ¿Qué podemos hacer si nuestra información queda almacenada y no contamos con operaciones para eliminarla? ¿Qué alternativas nos quedan cuando para utilizar un sistema particular, el mismo requiere de una cuenta en un cierto servicio gratuito que no queremos utilizar o de un dispositivo hardware que no podemos adquirir?

[27] Minimize mutability, Effective Java, Joshua Bloch, 2nd edition.

Para comprender y dar respuestas a estas cuestiones, la cualidad de la seguridad entonces debe ser complementada con otras cualidades de diseño:

- **protección de datos personales**[28]: se trata de garantizar a les usuaries el pleno control de los datos personales que ingresan o generan en un sistema, para lo cual el diseño debe contemplar mecanismos que que permitan consultarlos, gestionarlos y eliminarlos de forma permanente.
- **soberanía tecnológica**[29]**:** se trata del grado en que un sistema depende (o fuerza a sus usuaries a depender) de tecnologías que se generan mayormente o totalmente fuera de los confines de sus límites territoriales y su jurisdicción legal.
- **prevención del extractivismo de datos:** se trata de evitar consumir (u ofrecer) premios o servicios tecnológicos gratuitos o de bajo costo a cambio de datos personales o producciones digitales[30]. Usualmente, para continuar teniendo control sobre las mismas, es necesario abonar licencias de valor creciente.

4.2 Escalabilidad

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

Por ejemplo, luego de que Facebook comprara WhatsApp, la aplicación de mensajería dejó de funcionar por unas horas. En la desesperación por comunicarse, mucha gente se bajó la aplicación Telegram[31], plataforma que terminó también cayéndose por no poder estar preparada para soportar esa carga (mucho mayor a la habitual).

4.3 Eficiencia (Performance) - Sustentabilidad

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 (horas persona)
- Recursos Humanos para la ejecución del sistema (tiempo de uso)
- Recursos Hardware necesarios para la ejecución del sistema (memoria, procesamiento, almacenamiento)

La tercera acepción de eficiencia (recursos Hardware) es la más tradicional, pero con los costos decrecientes del hardware, el interés en la esta interpretación es menor que hace 30 años, lo que lleva a que haya un mayor interés en la cualidad de escalabilidad.

Ahora bien, si bien es cierto que el hardware se ha vuelto cada vez más económico, haciendo que la optimización del software no sea ya una prioridad, esto se trata de sólo un lado de la moneda.

[28] https://www.theguardian.com/technology/2016/may/12/facebook-free-basics-india-zuckerberg
[29] https://sur.conectas.org/wp-content/uploads/2018/07/sur-27-espanhol-renata-avila-pinto.pdf
[30] https://www.20minutos.es/tecnologia/actualidad/cuanto-vale-worldcoin-espana-escaneo-iris-5226624/
[31] https://techcrunch.com/2014/02/24/telegram-saw-8m-downloads-after-whatsapp-got-acquired/

Del otro lado, ejecutar código cada vez menos eficiente y más demandante en términos de uso de memoria, almacenamiento, red y procesamiento requiere de mayores volúmenes de cómputo, que en general sólo pueden ser provistos por grandes empresas extranjeras y, más grave aún, de grandes niveles de consumo energético[32] [33]. Esto debe representar una llamada de atención en un mundo donde el acceso a la energía y su medios de almacenamiento es motivo de conflictos geopolíticos[34] y donde el extractivismo y los terricidos [35] nos ha arrastrado a los efectos drásticos del cambio climático. De allí que la **sustentabilidad** y **responsabilidad ecológica** deba ser una cualidad de diseño[36]

En la misma línea, el constante cambio tecnológico nos arroja a ciclos constantes de actualización de hardware y software, aún cuando los dispositivos no han alcanzado el máximo de su vida útil, lo cual se conoce como *obsolescencia programada*[37]. Obligar a las actualizaciones constantes y a quitar soporte prematuro a dispositivos, si bien es conveniente para el proceso de desarrollo, a través de la necesidad de producción constante de nuevos aparatos y la generación de basura tecnológica, tiene un profundo impacto sobre la tierra, por lo que **extender la vida útil del software** también debe ser una cualidad a atender.

4.4 Trazabilidad

La **trazabilidad** se trata del grado en que podemos rastrear los orígenes de:

- las decisiones de diseño y sus cambios a lo largo del ciclo de vida del sistema informático, desde sus requerimientos, documentos hasta su implementación;
- las operaciones que ocurren en el sistema, para su posterior debugging y auditoría.

La trazabilidad nos deberá dar respuestas a preguntas como *cuándo, dónde, quién y por qué* se ejecutó una cierta acción. La trazabilidad también está vinculada al concepto de **transparencia** que mencionamos anteriormente: informar *quién, dónde, desde cuándo y por qué* administra o mantiene un sistema informático es parte, justamente, de la cualidad de transparencia.

[32] https://agupubs.onlinelibrary.wiley.com/doi/10.1029/2023EF003871
[33] https://www.france24.com/es/%C3%A1frica/20231222-las-%C3%BAltimas-tropas-francesas-abandonan-n%C3%ADger-poniendo-fin-a-una-d%C3%A9cada-de-misiones-en-el-sahel
[34] https://www.france24.com/es/%C3%A1frica/20230801-constituye-el-golpe-de-estado-en-n%C3%ADger-una-amenaza-para-las-centrales-nucleares-francesas
[35] https://ri.conicet.gov.ar/handle/11336/207758
[36] https://www.sciencedirect.com/science/article/pii/S266665962200004X
[37] https://www.gob.mx/profeco/es/articulos/obsolescencia-programada-disenados-para-morir?idiom=es

4.5 Usabilidad - Accesibilidad - Inclusión

La **usabilidad** se trata del grado en que les usuaries pueden comprender y utilizar el software de manera efectiva y la facilidad con la que pueden aprenderlo.

El problema con la noción de usabilidad es que la facilidad de uso no es algo universal: todas las personas son diferentes, con distintos recursos, culturas, historias y capacidades técnicas, cognitivas y físicas. De esta observación deriva la cualidad de **accesibilidad,** que trata de diseñar derribando estas barreras[38].

Por otro lado, más allá del avance en estándares técnicos para mitigar o eliminar barreras *capacitistas*, es fundamental que el software no sea, en general, excluyente para todas aquellas personas distintas a quienes lo construyeron. En otros términos, la exclusión no proviene de limitaciones técnicas, sino que son los sesgos culturales de quienes los diseñan los que generan dichas marginaciones: el *racismo*[39], *sexismo*[40] y *binarismo*[41], solo para enunciar algunos. Estos actos discriminatorios llegan en ocasiones a impedir el uso del software, por ejemplo:

- en 2021, el sistema de recorte automático de imágenes de Twitter tenía a eliminar a las personas negras[42], por lo que debió ser retirado;
- en 2018 Amazon lanzó un sistema de reclutamiento que tendía a rechazar las postulaciones de mujeres a puestos tecnológicos[43], y también debió ser luego retirado.

De hecho, la exclusión digital es aún más grave en contextos *interseccionales*[44], es decir, en personas que son víctimas de dos o más estigmas sociales a la vez. Por ejemplo, varias de las herramientas de reconocimiento facial más extendidas son significativamente menos precisas en mujeres negras que en hombres blancos[45].

Nuevamente, el **pluralismo** en la constitución de los equipos de trabajo y la **rendición de cuentas** a todas las partes involucradas surgen como cualidades con correlación positiva con la **inclusión**.

[38] https://www.w3.org/mission/accessibility/
[39] https://cajanegraeditora.com.ar/el-algoritmo-de-la-raza-notas-sobre-antirracismo-y-big-data/
[40] https://data-feminism.mitpress.mit.edu/pub/v874jd7x/release/1
[41] https://infanciastrans.org/la-apropiacion-de-la-vida-hablemos-de-opresiones-y-normatividades/
[42] https://edition.cnn.com/2021/05/19/tech/twitter-image-cropping-algorithm-bias/index.html
[43] https://www.reuters.com/article/idUSKCN1MK0AG.
[44] https://biblioteca-repositorio.clacso.edu.ar/bitstream/CLACSO/248817/1/Interseccionalidad.pdf
[45] https://sitn.hms.harvard.edu/flash/2020/racial-discrimination-in-face-recognition-technology/

5 Conclusiones

Como ya hemos señalado anteriormente, no existe un proceso *mágico* que nos lleve a un buen diseño y un buen software. Tampoco contamos con un proceso sistemático, taxativo o reproducible: no existen reglas automatizables para tomar decisiones de diseño y es justamente allí donde aportamos nuestra experiencia y criticidad, que no se pueden transmitir, pero sí ejercitar.

En particular hoy, en que día a día vemos más herramientas basadas en inteligencia artificial que proponen resolver tareas que otrora se pensaban imposibles por estar asociadas a la creatividad humana, es fundamental apartarse de un pensamiento simplista, productivista, mecanicista y algorítmico. Por el contrario, debemos reafirmar la importancia de las cualidades como heurísticas, que nos deben servir para expandir nuestra mente, no para cerrarla.

Las cualidades no nos serán útiles para alcanzar una solución particular, única y óptima, sino para poder fundamentar y comunicar las decisiones que se tomaron en el proceso. Oficiarán como punto de partida para generar discusiones, para formar criterios colectivos, en equipos de trabajo interdisciplinarios y plurales, cumpliendo con los requerimientos del cliente, pero, ante todo, creando al servicio de la comunidad.