Alberto Basalo Follow

May 19 · 5 min read

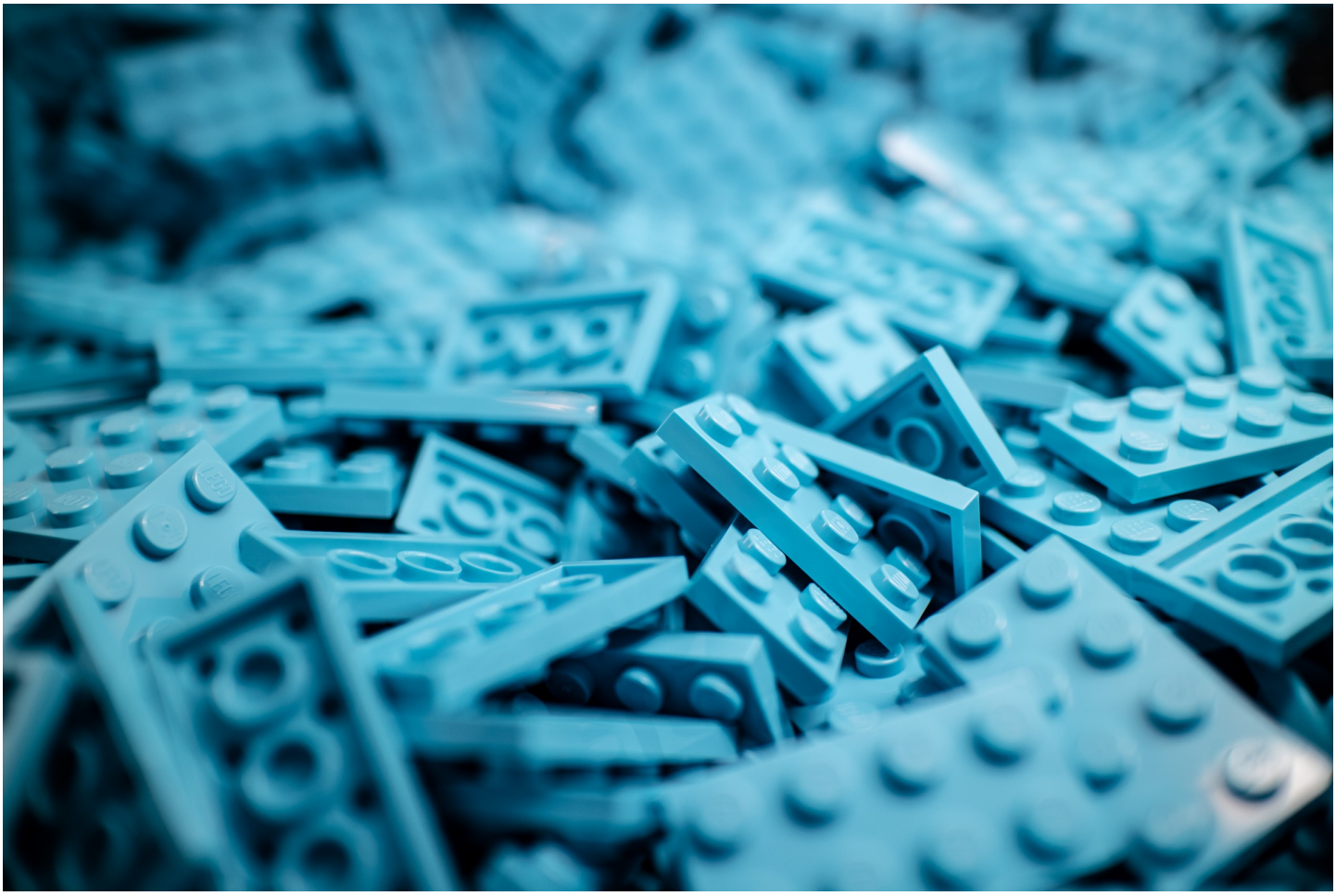

Photo by Iker Urteaga on Unsplash

En algún sitio he leído que *el software es como el plástico: maleable y costosamente duradero*. Lo primero lo distingue del hardware, algo que es costoso de cambiar. Lo segundo lo distingue de casi todo. El software como tal no caduca, ni siquiera se degrada; pero su mantenimiento es un quebradero de cabeza.

Con el tiempo y la lucidez de unos cuantos maestros del desarrollo, ha aparecido una disciplina para facilitar el mantenimiento de los programas: **la arquitectura del software**. Podríamos definirla como **el conjunto de decisiones que tomamos al diseñar un programa para facilitar su mantenimiento**.

Dichas decisiones han de ser proporcionadas al tamaño, y sobre todo, al tiempo de desarrollo y vida esperada después de la puesta en producción. Todas y cada una de esas decisiones tendrán un objetivo pero también un coste.

Hay muchos libros sobre el tema que desde hace años se incluye como materia troncal en estudios universitarios y másteres. No se aprende en un artículo en un blog o una newsletter. Lo que aquí pretendo es simplemente que veas la necesidad de la arquitectura y que uses lo que ya sabes, que aprendas lo que desconoces o que busques ayuda si lo necesitas.

Proyectos cortos.

Entre 1 y 2 años de tiempo de desarrollo y explotación. Equipos estables de menos de 5 integrantes.

Ejemplos: Producto mínimo viable en una start-up que no se sabe si vivirán lo suficiente. Proyectos para campañas o negocios de duración limitada y conocida. Herramientas *adhoc* para integración temporal entre sistemas. Otros desarrollos técnica y funcionalmente simples.

Situación: Los tiempos y presupuestos serán muy rigurosos, por tanto debemos abaratar y reducir el desarrollo. Los cambios funcionales serán muy frecuentes, aunque afortunadamente muchos ocurrirán antes de la puesta en producción con cliente y riesgo real. La reducción del coste del cambio está en la reducción del coste de entender y manipular el código.

Reglas:

- Código: Evitar los *code smells* mediante aplicación de reglas que lleven a un código limpio fácil de leer.
- Mantra: Muchas estructuras y funciones pequeñas y bien nombradas.
- Test: Garantizar que el software sigue funcionando a pesar de los frecuentes cambios mediante *smoke-test* o pruebas de integración sencillas.
- Componentes: Separar el código en capas lógicas (packages, namespaces, modules… según el lenguaje). Por ejemplo, siguiendo la arquitectura en tres capas horizontales: presentación -> lógica -> persistencia.

- Despliegue: Mantener mientras sea posible un despliegue sencillo, tendente al monolito en cada capa física. Por ejemplo : cliente—servidor

> *En estos proyectos el objetivo es **reutilizar código**, principio DRY, pero sin complicarlo demasiado para **facilitar el cambio** constante: principios YAGNI y KISS.*

Consejos:

Para empezar puedes formarte leyendo libros, asistir a seminarios o bien contratar puntualmente un consultor durante el inicio del proyecto. El resultado debe ser un conjunto de buenas prácticas y métricas que mantengan la base de código en unos parámetros de legibilidad razonables.

El equipo de desarrolladores se hará cargo de la arquitectura, la cual va emergiendo y evolucionando con las necesidades concretas del proyecto.

Proyectos medios.

Entre 2 y 5 años de tiempo de desarrollo y explotación. Equipos de 5 o más integrantes.

Ejemplos: Productos de *start-up* que ya han funcionado. Automatización de procesos de negocio de empresas consolidadas. Renovación de sistemas de información en organizaciones con sistemas *legacy*.

Situación: Si algo hay seguro para los próximos años es que las reglas del negocio informatizado van a cambiar. Por si fuera poco, lo harán ya con el sistema en producción dando servicio ininterrumpido a clientes o sistemas críticos para la empresa. Así que el cambio ha de integrarse de manera transparente y sin oposición. Impactando lo menos posible en el código ya hecho y en los paquetes ya desplegados.

Reglas:

- Código: Fomentar el cambio funcional mediante la aplicación de los principios SOLID al diseño de las clases.
- Mantra: Encapsular lo que varía y depender de interfaces en lugar de implementaciones concretas.

- Test: Garantizar que el software funciona unitariamente mediante pruebas a nivel de paquete desplegable.
- Componentes: Las tres capas lógicas por niveles son insuficientes. Para permitir un desarrollo paralelo e independiente debemos desacoplaras mediante abstracciones intermedias.
- Despliegue: Para reducir el impacto de un cambio, este debe afectar a partes y nunca a todo del sistema. Los componentes deben agruparse en silos funcionales verticales que no exijan el compilado y despliegue completo.

> *A medio plazo la clave es la **extensibilidad de un sistema en producción,** que se consigue facilitando el desarrollo y despliegue por partes conectadas pero independientes.*

Consejos:

Para definir e implantar esta arquitectura se necesita la participación de alguien experimentado. Según el calibre del problema y los recursos disponibles puede ser necesario personal interno dedicado o la contratación de un consulting externo; pero implicado durante todo el ciclo de vida.

El equipo debe participar en el diseño y elección constante de patrones. La imposición desde un pedestal de arquitecto de software no funciona si es vista sólo como una restricción en la libertad de acción del programador. Pero sin dichas restricciones el software es muy probable que degenere en una masa deforme muy difícil de cambiar.

Proyectos largos.

Más de 5 años de tiempo de desarrollo y explotación. Equipos de 20 o más integrantes con alta rotación.

Ejemplos: Software de gestión de grandes corporaciones, bancos y administraciones públicas. Continuidad de productos o servicios online de gran éxito.

Situación: Más allá de 5 años ya sólo una cosa será cierta; todo habrá cambiado. Además del negocio también cambiarán las tecnologías, las personas, y las demandas y expectativas de los usuarios. Así que, si algo va a durar una década, mejor que no dependa de pequeños detalles sin importancia como las bases de datos, las interfaces de usuario, los

protocolos de comunicaciones, los frameworks o los dispositivos físicos de última generación :-)

Reglas:

- La arquitectura clásica en capas horizontales y silos no es suficiente. Necesitas desacoplar completamente todos los paquetes manteniendo un núcleo lo más estable e independiente posible.
- Las arquitecturas más usadas son la hexagonal, la circular o cebolla con las entidades del dominio y las reglas en el centro o cúspide del árbol de dependencias.

Consejos:

Sólo hay un consejo posible. Un departamento de arquitectura interno o una consultora especializada deben pilotar todo el desarrollo todo el tiempo.