

Clean Code Avanzado

Alberto Basalo, Vitae Digital

Contenido

1 Principles of Software Design

2 SOLID Principles

3 Design Patterns

4 Software Architecture

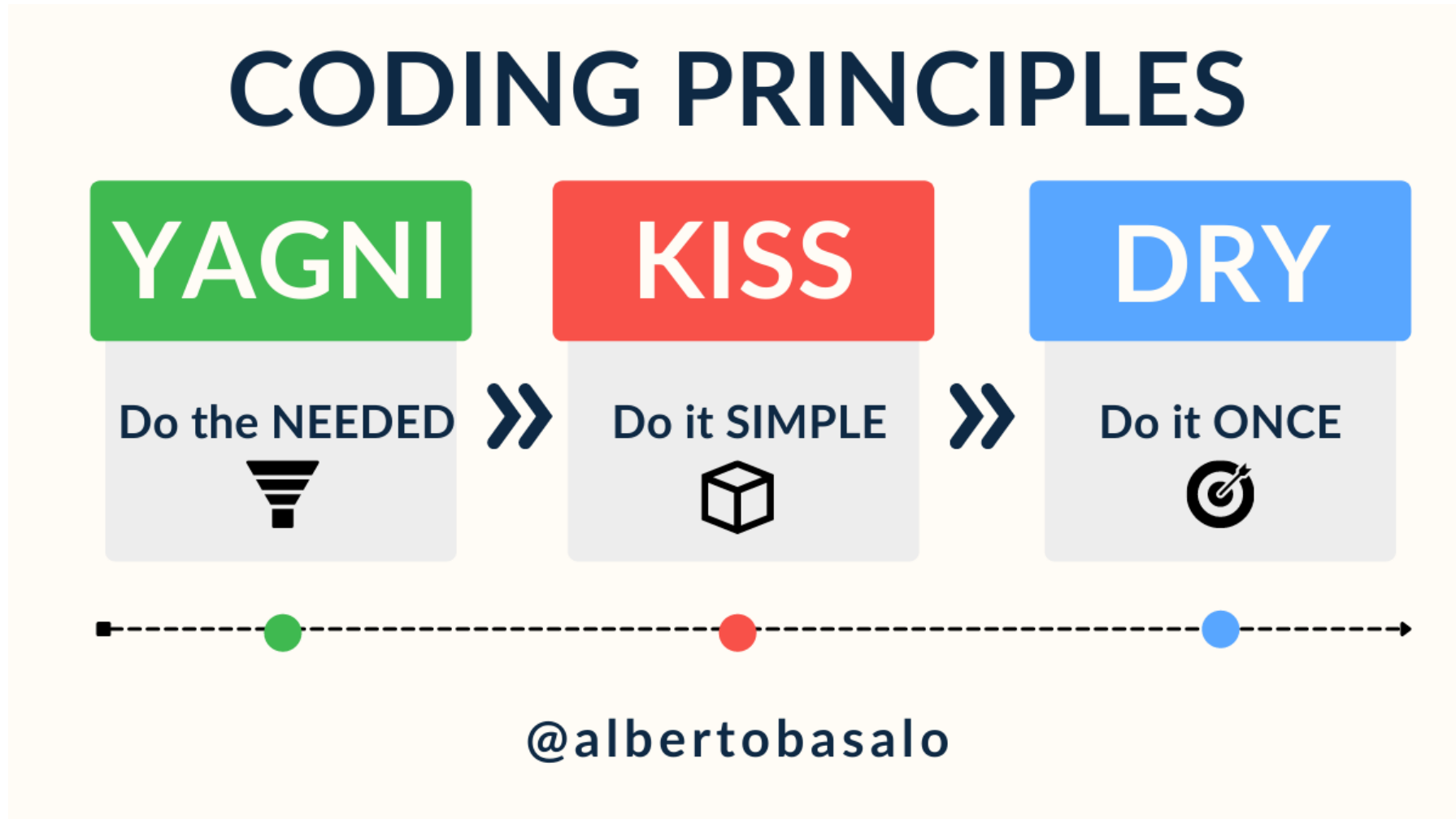
❶ Principles of Software Design

1 YAGNI, KISS, DRY

2 Obstacles for change

3 Metrics, preventers, and facilitators

1 YAGNI, KISS, DRY



1 2 Obstacles for change

Rigidez

Un cambio afecta a muchas partes. Al cambiar un objeto hay que cambiar otros muchos.


Fragilidad

Las errores saltan en lugares inesperados. Los cambios en un objeto tienen efectos en otros muchos.

Inmovilidad





No se puede reutilizar el código fuera de su entorno. Los cambios en un objeto dependen de otros muchos.

1 3 Metrics, Preventers and Facilitators

-  El **acoplamiento** es el principal enemigo del cambio
 - Se puede **medir**
 - Se puede **evitar**
 - Se puede **limpiar**



Reduce métricas de acoplamiento

- **Eferente** 
 - ¿Cuántas dependencias uso?
-  **Aferente**
 - ¿Cuántos dependientes me usan?
-  **Exposición**
 - ¿Cuántas funcionalidades usan?
-  **Tamaño**
 - ¿Cuántas instrucciones tengo?

✖ Change preventers

- **Feature envy**
 - Use your properties
- **Inappropriate Intimacy**
 - Reduce knowledge
- **Primitive obsession**
 - Aportar cohesión con invariantes
- Divergent change.
 - Una clase que se cambia por diferentes razones.
- Shotgun surgery.
 - Un cambio que requiere muchos cambios. Difícil encontrarlos, fácil olvidarse.
- Cyclomatic complexity.
 - Número de rutas únicas por anidamiento, switches y condiciones complejas

Change Facilitators

- **Tell don't Ask**
 - Reduce calls
- **Law of Demeter**
 - Don't talk to strangers
- **Command-Query Separation**
 - Cada método debe un comando o una consulta; pero no ambos.
- **P.O.L.A. Principle Of Least Astonishment.**
 - Ni me sorprendas ni me hagas pensar.
- **H.P. Hollywood principle.**
 - No nos llames, ya te llamaremos.
- **CoC Convention over configuration.**
 - Establecer convenios que minimicen decisiones.

2 SOLID principles

S.O.L.I.D.

Single responsibility

 *Nunca debe haber más de una razón para que una clase cambie.*


 Clases grandes, muy tentador seguir añadiendo funcionalidad

 Genera clases pequeñas de alta cohesión.



Open / Closed

 *Las entidades de software deben estar abiertas a la extensión, pero cerradas a la modificación.*

 Las condiciones if switch

 Agregar (o eliminar) módulos (clases) mejor que manipular líneas.

Liskov substitution

 *Si S es un subtipo de T , entonces los objetos de tipo T pueden reemplazarse con objetos de tipo S sin alterar ninguna de las propiedades deseables del programa.* 

 Herencia en lugar de composición

 Debería poder sustituir una cosa por otra, si se declara que esas cosas se comportan de la misma manera.

Interface segregation

 *Muchas interfaces específicas son mejores que una interfaz de uso general.*

 Pensar en términos de lo que soy, en lugar de lo que puedo hacer.

 Muestra a tus clientes sólo lo que necesitan ver.

🙄 Dependency Inversion

🧙 *Depende de abstracciones, no de concreciones.*

😈 El new

👶 Evita crear directamente tus dependencias.



Mandamientos

- ⊘ **No abrumes:** toda abstracción tiene un coste.
- ⊘ **No defraudes:** cumple tus contratos.
- ⊘ **No confundas:** mantén los niveles de abstracción.
- ⊘ **No líes:** junta lo que cambian por la misma razón.
- ⊘ **No expolies:** usa lo imprescindible.

Design patterns

Creacionales

¿Cómo **instanciar** objetos?

Estructurales

¿Cómo **relacionar** objetos?

De comportamiento

¿Cómo **comunicar** objetos?



Creacionales

PROPORCIONAN MECANISMOS DE CREACIÓN DE OBJETOS

- **Abstract Factory:** familia de factorías sin exponer nada concreto.
- **Builder:** crea objetos complejos a partir de otros.
- **Factory Method:** delega la creación de instancias a otros.
- **Prototype:** genera una instancia a partir de otra.
- **Singleton:** asegura una instancia única de una clase.



Estructurales

ENSAMBLAN OBJETOS Y CLASES EN ESTRUCTURAS MÁS GRANDES

- **Adapter:** envuelve un objeto para que sea compatible con otro.
- **Bridge:** separa el interfaz de la implementación para variar por separado.
- **Composite:** trata múltiples objetos individuales como uno solo.
- **Decorator:** agrega comportamiento dinámicamente.
- **Façade:** proporciona un acceso simple a un sistema de objetos complejo.
- **Flyweight:** reduce el consumo de memoria o CPU compartiendo recursos.
- **Proxy:** una clase actúa como representante de otra.

De comportamiento

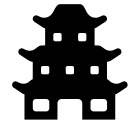
CONTROL DE COMUNICACIÓN Y ASIGNACIÓN DE RESPONSABILIDADES

- **Chain of Responsibility:** encadena llamadas entre objetos o métodos.
- **Command:** encapsula acciones en objetos.
- **Iterator** recorre los elementos de un conjunto sin revelar cómo.
- **Mediator:** desacopla dos objetos comunicándose con ambos.
- **Memento:** guarda el estado actual para un uso futuro.
- **Observer:** notifica cambios a suscriptores interesados.
- **Strategy:** cambia el algoritmo según las circunstancias.

4 Software Architectures

Decisiones de alto impacto

- ***¿Qué clase de solución se necesita?***
 - *Diseño estratégico*
- ***¿Cómo lo voy a construir?***
 - *Diseño táctico*
- ***¿Cuál será la estructura?***
 - *Detalles técnicos*



Layered

*Separación por **tecnología***

- **app** (*Presentación*)
 - **business** (*Lógica*)
 - **data** (*Persistencia*)
-
- Data centric
 - Anemic models (DTO, POJO)



Domain centric

Separación por estabilidad

- **domain** (*Entidades y reglas de negocio*)
- **detail** (*Presentación, frameworks e infraestructura*)
- Business centric
 - Entities, Aggregates, Value-Objects
- Decoupled
 - Events, Ports (Interfaces), Adapters (Implementations)
- Variantes: Hexagonal, Onion, Clean



C.Q.R.S.

Separación por responsabilidad

- **api**
 - **commands** (*domain centric*)
 - **queries** (*data centric*)
- Command Query Segregation ++
- Puede requerir dos bases de datos
- Puede requerir Event Driven Architectures



Microservicios

Separación por funcionalidad

- **api1** (*Resuelve un problema a su manera*)
 - **api2** (*quizá sea un simple layered, anémico*)
 - **api3** (*o requiera un dominio complejo con mucha lógica*)
 - **api4** (*e incluso gestión de eventos, asincronismo, orquestación, sagas...*)
-
- Cada servicio implementa **pocos casos de uso relacionados**
 - Cada servicio tiene **su propia base de datos**
 - Cada servicio se desarrolla y despliega **con tecnología propia**
 - Requiere **orquestación** de eventos y **sincronización** de datos



Code Labs

- Repositorio en GitHub
 - <https://github.com/LabsAdemy/CleanCodeAdvanced>
 - Ramas por lección (1, 2...)
- TypeScript
 - JavaScript con tipos; similar a Java y C#
 - Ejemplos síncronos y constructores estilo Java C#
 - Fakes para infraestructura (DB, HTTP y SMTP)
- Edición local
 - git clone <https://github.com/LabsAdemy/CleanCodeAdvanced.git>
 - npm i
 - git checkout 1_PSD

Astro Bookings

- Aplicación de reservas para empresa de viajes espaciales
 - <https://github.com/AstroBookings/.github/wiki/Context-Diagram>
 - <https://raw.githubusercontent.com/wiki/AstroBookings/.github/requirements/AstroBookings-event-storm.png>
- Funcionalidad:
 - Solicitar una reserva `bookings.solicite()`
 - Anular una reserva `bookings.annulate()`
 - Ofertar viaje `trips.offer()`
 - Cancela viaje `trips.cancel()`
 - Listar viajes `trips.getList()`

bookings.solicite()

1. Validar petición
2. Consultar disponibilidad operador
3. Obtener precio
4. Pagar con tarjeta
5. Confirmar reserva al operador
6. Notificar reserva al viajero