Principios y patrones

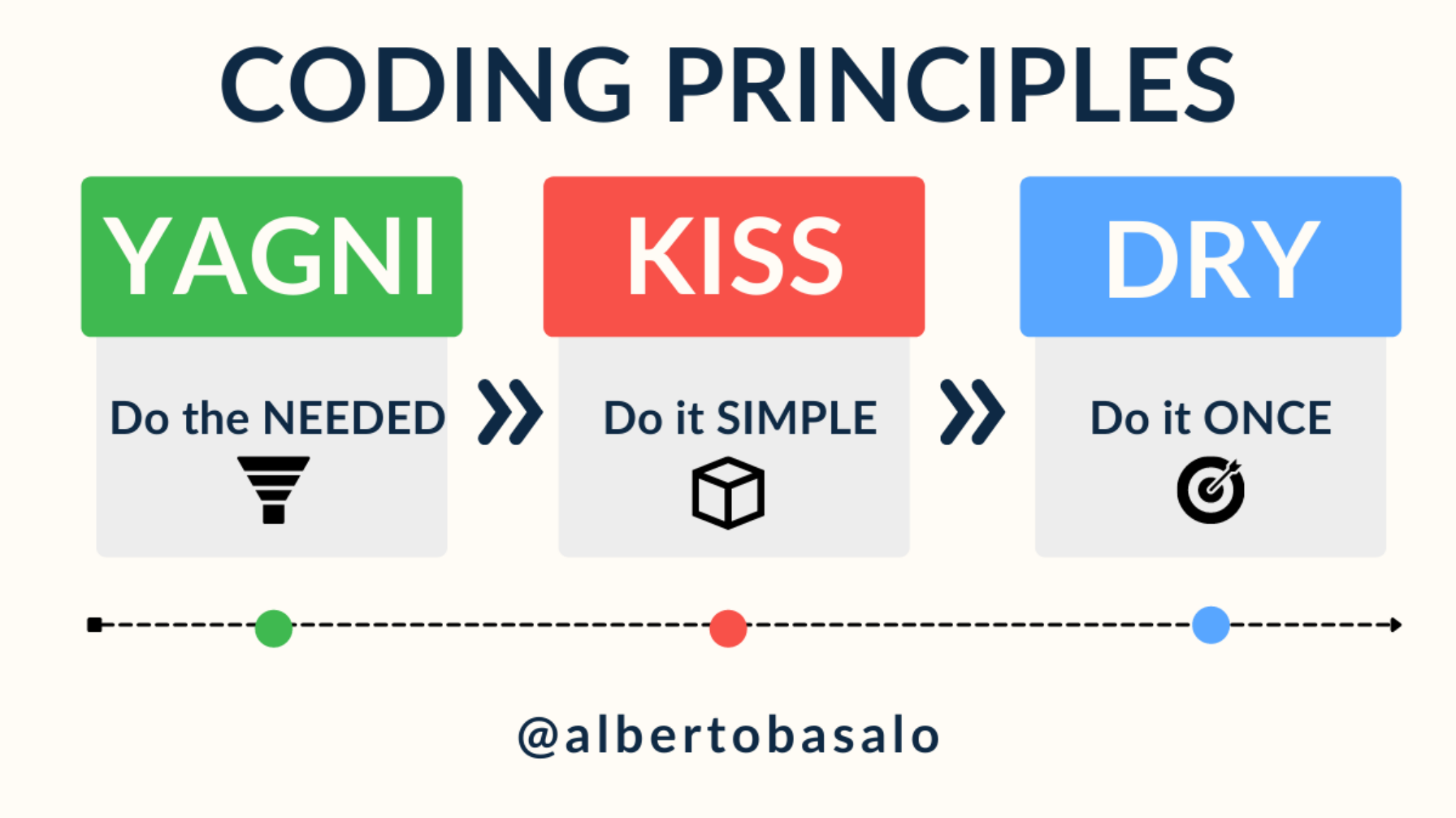
Código limpio avanzado

Alberto Basalo

Contenido

1. Principles of Software Design
2. SOLID Principles
3. Design Patterns

YAGNI, KISS, DRY



Obstacles for change

Rigidez

Un cambio afecta a muchas partes.

Al cambiar un objeto hay que cambiar otros muchos.

Fragilidad

Los 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.

📊 Metrics, Preventers and Facilitators

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

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**
  - Usa tus propiedades, (más que las de los demás)
- **Inappropriate Intimacy**
  - Reduce el conocimiento necesario para usar dependencias
- **Primitive obsession**
  - Usa estructuras o clases que aporten cohesión y reglas básicas 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 el número de llamadas para una tarea
- ✓**Law of Demeter**
  - ✓Don’t talk to strangers
- ✓**Command-Query Separation**
  - ✓Cada método debe ser 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.

S.O.L.I.D. principles

🎯 Single responsibility

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

😈 Clases grandes, muy tentador seguir añadiendo **funcionalidad**

 Clases pequeñas, sencillas reutilizables y 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.*  Debería poder sustituir una cosa por otra, si se declara que esas cosas se comportan de la misma manera.

 Herencia en lugar de composición

 Mejor la composición, envolviendo y usando otros… componentes

 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. Ofrece capacidades.

🙃 Dependency Inversion

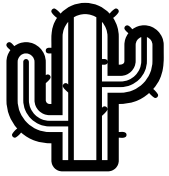
 *Depende de abstracciones, no de concreciones.*

😈 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

- **Builder:** facilita o dirige la creación de objetos complejos.
- **Factory Method:** crea instancias de distintas clases pero que cumplen una interfaz.
- **Prototype:** crea una copia (clon) de un objeto existente aplicando mutaciones controladas.
- **Singleton:** asegura una instancia única de una clase, evita usar variables globales.

Estructurales

ENSAMBLAN OBJETOS Y CLASES EN ESTRUCTURAS MÁS GRANDES

- **Adapter:** desacopla código legacy o de terceros del desarrollo principal
- **Bridge:** permite que subsistemas complejos evolucionen independientemente
- **Decorator:** agrega funcionalidad a una clase sin modificarla
- **Façade:** proporciona un acceso simple a un sistema de objetos complejo.

De comportamiento

CONTROL DE COMUNICACIÓN Y ASIGNACIÓN DE RESPONSABILIDADES

- **Command:** lleva la definición de acciones de métodos a clases incrementando su versatilidad
- **Observer:** desacopla emisores y procesadores de eventos notificando cambios a suscriptores.
- **Strategy:** escoge el algoritmo según las circunstancias de ejecución
- **Template:** asegura un algoritmo común permitiendo variaciones.