4. Solid, buenos y malos diseños

Criterios de buen diseño

> *Análisis*: modelamos **qué** necesitamos
> *Diseño*: modelamos *cómo* resolveremos lo necesitado

Diseño

Teniendo (como resultado del análisis) el output que nos proponemos, nos interesa *cómo generamos ese output*.

Buen diseño vs Mal diseño

BUEN DISEÑO

> Motivación de un *buen diseño*:
> - manejar bien el cambio
> - lidiar con la complejidad
> - concretizar rápidamente

> Características de un *buen diseño*:
> - alta cohesión:
> - cada módulo tiene una responsabilidad completamente definida
> - las responsabilidades se relacionan entre sí.
> - ejm: un validado de email solo valida emails. no envia emails y los guarda en la bdd.
> - bajo acoplamiento:
> - depende lo menos posible de otros módulos
> - si cambia internamente no afecta a otros módulos

- ejm: un UserControler no debería de saber cómo guarda los datos el UserRepository.

MAL DISEÑO

Características de un *mal diseño*:

- Rigidez:
  - el cambio es laborioso y dificil
  - cualquier cambio requiere ajustes en múltiples partes del sistema.
  - puede deberse a un alto acoplamiento entre componente (dependencias fuertes)
- Fragilidad:
  - el cambio es arriesgado
  - un cambio rompe otra funcionalidades que parecen no estar relacionadas.
  - puede deberse a efectos secundarios ocultos o mala encapsulación
- Inmobilidad:
  - dependiente del resto del código
  - no se puede reusar en otros proyectos (El diseño es muy específico)
- Viscosidad
  - aplicar hacks o parches (código malo) es aplicar una refactorización

Causas: incorrectas dependencias entre módulos.

Cómo lograr un buen diseño: solid

*S*RP (única responsabilidad).

- solo porque puedes no implica que debas
- mantiene una sola responsabilidad

*O*CP (open/closed)

- abierto a la extensicón, cerrado a la modificación

- se debería poder agregar funcionalidades nuevas sin modificar el código existente,

*L*SP (liskov Substitution)

- las subclases deben poder reemplazar a sus clases padre sin romper el código.
- cada subclase debe poder implementar *todos* los comportamientos esperados de la clase padre (De otra forma rompería o lanzaría una excepción dependiendo de qué subclase de emplea de fondo).
- ejemplo:
  - un stack no puede derivar de un vector porque al tratar de indexar el stack generaría un comportamiento indeseado.
  - Por otro lado un vector sí puede ser una implementación de un stack porque el stack sí puede ser reemplazado con un stack.

*I*SP (Interface Segregation)

- interfaces específicas
- no forzamos que una clase deba implementar métodos que no usan.

*D*IP (Dependency inversion)

- Las clases dependen de abstracciones (no de clases concretas)
- Si hacemos que dependa de una interfaz entonces podemos cambias facilmente la clase concreta que se usa de fondo.
- Ejemplo: la conexión entre un tomacorriente y una lámpara debe ser a travéz de una interfaz. el tomacorrientes no está anclado a solo funcionar con la lámpara., se pueden enchufar otros electrodomésticos