

SOLID

S **Single Responsability**:

cada clase tiene que tener una sola responsabilidad

O **Open/Closed:**

las clases tienen que están abiertas para su extensión pero cerradas para su modificación

L **Liskov sustitución:**

cuando una clase hija puede sustituir al padre y que el código siga funcionando igual ya que no modifica el comportamiento de la clase padre

I **Segregación de interfaces:**

se tienen que crear interfaces pequeñas y específicas

D **Inversión de dependencia:**

los módulos de alto nivel no tienen que depender de los de bajo nivel, ambos tienen que depender de las abstracciones.

GRASP:

**Information Expert:** se le da la responsabilidad a la clase que tiene la información necesaria para cumplirla

**Creator:** una clase tiene que crear instancias de otra si cumple con algunos criterios como contener o usar muchas instancias de ella

**Controller:** asigana la responsabilidad de manejar los eventos de un sistema a una clase que represente el caso de uso o un controlador general

**Low Coupling:**

Mantén el acoplamiento entre clases lo más bajo posible para que el sistema sea más flexible y fácil de mantener.

**High Cohesion** (Alta Cohesión):

Asigna responsabilidades de manera que las clases se centren en tareas relacionadas, lo que hace el diseño más claro y eficiente.

**Polymorphism** (Polimorfismo):

Usa polimorfismo para que los comportamientos relacionados se gestionen a través de interfaces comunes, permitiendo extensibilidad sin cambiar el código existente.

**Pure Fabrication** (Fabricación Pura):

Crea clases que no representan entidades del dominio pero que ayudan a mantener baja la cohesión y el acoplamiento.

**Indirection** (Indirección):

Usa intermediarios para desacoplar elementos del sistema, permitiendo que interactúen sin estar directamente vinculados.

**Protected Variations** (Variaciones Protegidas):

Protege las partes del sistema que son susceptibles a cambios usando abstracciones y encapsulamientos.

**Strategy** porque te permitió separar la lógica de diferentes comportamientos o algoritmos en clases independientes, facilitando la modificación y la extensión del código sin tener que tocar la lógica principal. Esto ayuda a mantener el código más limpio, organizado y fácil de mantener a largo plazo.

TEATRO ES **Singleton** para asegurarte de que una clase tenga una única instancia en todo el programa