|  |  |  |
| --- | --- | --- |
| Control de cambios: | | |
| Revisión | Fecha | Observaciones |
| 00 | 09/04/2025 | Documento inicial |

ÍNDICE

[1 Objeto 3](#__RefHeading___Toc1514_3823752326)

[2 Campo de aplicación 3](#__RefHeading___Toc1516_3823752326)

[3 Definiciones 3](#__RefHeading___Toc1637_3616825104)

[4 Responsabilidades 3](#__RefHeading___Toc1639_3616825104)

[4.1 Equipo de desarrollo 3](#__RefHeading___Toc1522_3823752326)

[5 Instrucciones 4](#__RefHeading___Toc1671_3823752326)

[5.1 SRP - Single Responsibility Principle 4](#__RefHeading___Toc1673_3823752326)

[5.2 OCP - Open/Closed Principle 4](#__RefHeading___Toc1673_3823752326_Copia_)

[5.3 LSP - Liskov Substitution Principle 5](#__RefHeading___Toc1673_3823752326_Copia1)

[5.4 ISP - Interface Segregation Principle 6](#__RefHeading___Toc1673_3823752326_Copia2)

[5.5 DIP - Dependency Inversion Principle 6](#__RefHeading___Toc1673_3823752326_Copia3)

[Anexo 1: Reglas prácticas de código limpio 8](#__RefHeading___Toc1562_3823752326)

[Anexo 2. Recursos 8](#__RefHeading___Toc1603_3823752326)

# Objeto

Establecer una guía clara sobre los principios SOLID propuestos por Robert C. Martin para la creación de software orientado a objetos mantenible, escalable y flexible. Este documento está enfocado en la correcta aplicación de cada principio mediante ejemplos prácticos y recomendaciones para el equipo de desarrollo.

# Campo de aplicación

Aplica a todos los desarrollos internos y externos de software orientado a objetos, en cualquier lenguaje que soporte programación orientada a objetos (OOP), como PHP, Java, C#, etc. Es especialmente relevante para los proyectos basados en Laravel y PHP dentro del entorno de Termosalud.

# Definiciones

* **SOLID**: Acrónimo que representa cinco principios fundamentales del diseño de software orientado a objetos:
* **S - Single** **Responsibility Principle** (Responsabilidad única)
* **O - Open/Closed Principle** (Abierto/cerrado)
* **L - Liskov Substitution Principle** (Sustitución de Liskov)
* **I - Interface Segregation Principle** (Segregación de interfaces)
* **D - Dependency Inversion Principle** (Inversión de dependencias)

# Responsabilidades

## Equipo de desarrollo

* **Aplicar los principios SOLID en el diseño de clases y módulos.**
* **Refactorizar código existente para adaptarlo progresivamente a estos principios.**

# Instrucciones

## SRP - Single Responsibility Principle

Principio de responsabilidad única.

* **Concepto**
  + Una clase debe representar un único concepto y responsabilidad
  + Una clase debería tener únicamente un motivo para cambiar
* **Como**
  + Clases de servicios pequeñas con objetivos acotados
* **Finalidad**
  + Cohesión y robustez
  + Permitir composición de clases
  + Evitar duplicidad

### Ejemplo

## OCP - Open/Closed Principle

// ❌ Mal  
class Report { generate(); print(); email(); }

// ✅ Bien  
class ReportGenerator, ReportPrinter, ReportMailer

Principio de abierto/cerrado.

* **Concepto**
  + Abierto a extensión y cerrado a modificación
  + Aplicable a clases, servicios, microservicios, clases de uso...
* **Como**
  + No depender de implementaciones específicas
  + Usamos herencia e interfaces
* **Finalidad**
  + Facilidad de añadir casos de uso

### Ejemplo

## LSP - Liskov Substitution Principle

// ✅ Bien  
interface Shape { area(); }  
class Circle implements Shape {...}  
class Square implements Shape {...}

Principio de sustitución de Liskov.

* **Concepto**
  + Si S es un subtipo de T, instancias de T deberían poderse sustituir por instancias de S sin alterar las propiedades del programa.
  + Una subclase debe poder ser ser usada como si fuera su clase padre sin que el programa falle.
* **Como**
  + El comportamiento de de las subclases debe respetar el contrato establecido en la superclase.
* **Finalidad**
  + Mantener correctitud funcional para poder aplicar OCP.

### Ejemplo

## ISP - Interface Segregation Principle

// ❌ Mal  
class Bird { fly(); }  
class Penguin extends Bird { fly() => throw error; }

Principio de segregación de interfaces

* **Concepto**
  + Ningún cliente debería verse forzado a depender de métodos que no usa
  + Es mejor tener muchas interfaces específicas que una general con métodos que no siempre se usan.
* **Como**
  + Definir contratos de interfaces basándonos en los clientes que las usan y no en las implementaciones que pudiéramos tener.
  + Evitar *header interfaces* promoviendo *role interfaces*.
* **Finalidad**
  + Alta cohesión y bajo acoplamiento estructural

### Ejemplo

## DIP - Dependency Inversion Principle

// ✅ Bien  
interface Workable { work(); }  
interface Eatable { eat(); }

Principio de inversión de dependencias

* **Concepto**
  + Módulos de alto nivel no deberían depender de los de bajo nivel. Ambos deberían depender de abstracciones.
* **Como**
  + Inyectar dependencias (parámetros recibidos idealmente en constructor)
  + Depender de las interfaces de estas dependencias y no de implementaciones concretas
  + LSP como premisa
* **Finalidad**
  + Facilitar la modificación y sustitución de implementaciones.
  + Mejorar testabilidad de clases

### Ejemplo

// ✅ Bien  
class AuthService {   
\_\_construct(UserRepositoryInterface $repo);   
}

# ****Anexo 1: Reglas prácticas de código limpio****

| Principio | Resumen |
| --- | --- |
| SRP | Una clase = una responsabilidad |
| OCP | Extensible sin modificar lo existente |
| LSP | Subclases deben funcionar como su padre |
| ISP | Interfaces pequeñas y específicas |
| DIP | Depender de abstracciones, no de concreciones |

# Anexo 2. Recursos

* **Libro: Clean Architecture, Robert C. Martin**
* **Cheat sheet:** <https://github.com/stevebrownlee/solid-principles>

# ****Anexo 3. ¿Qué significa el acrónimo STUPID?****

* ****S – Singleton**: Patrón de diseño ampliamente usado hace unos años. Evita exponer los colaboradores de una clase (algo negativo ya que hace más difícil de detectar el acoplamiento entre clases).  
  Provee mantener un estado global, algo que podemos conseguir vía inyección de dependencias.  
  Perjudica la testabilidad .**
* ****T - Tight Coupling**: Acoplamiento entre clases que dificulta la mantenibilidad y tolerancia al cambio.**
* ****U - Untestability**: Código difícil de testear.**
* ****P - Premature Optimization**: Anticiparnos a los requisitos de nuestro software desarrollando abstracciones innecesarias que añaden complejidad.**
* ****I - Indescriptive Naming**: Nomenclatura indescriptiva tanto a nivel de clases, como variables o atributos.**
* ****D - Duplication**: Duplicidad de código.**