



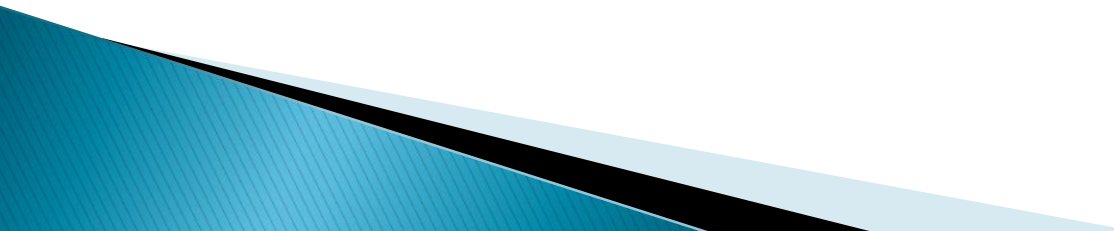
# Análisis y Diseño de Sistemas 2

2015

Ing. Luis Alberto Arias Solórzano

Unidad 3

# Diseño orientado a Objetos

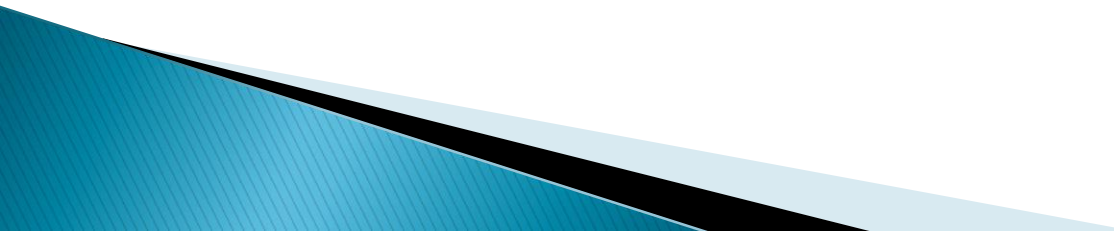
- ▶ Diseño orientado a objetos guía a los programadores a pensar en términos de objetos, en vez de procedimientos, cuando planifican su código.
  - ▶ Un objeto agrupa datos encapsulados y procedimientos para representar una entidad.
  - ▶ La interfaz del objeto son las formas de interactuar con el objeto, también se definen en esta etapa. Un programa orientado a objetos se caracteriza por la interacción de esos objetos.
  - ▶ El diseño orientado a objetos es la disciplina que define los objetos y sus interacciones para resolver un problema de negocio que fue identificado y documentado durante un análisis siempre orientado a objetos.
- 

# SOLID (object-oriented design)

En ingeniería de software, SOLID (Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion) es un acrónimo mnemónico introducido por *Robert C. Martin* a comienzos de los años 2000; en su libro *Agile Software Development, Principles, Patterns, and Practices*.

Representa cinco principios básicos de la programación orientada a objetos y el diseño. Cuando estos principios se aplican en conjunto es más probable que un desarrollador cree un sistema que sea fácil de mantener y ampliar en el tiempo.

Los principios SOLID son guías que pueden ser aplicadas en el desarrollo de software para eliminar código ineficaz o sucio, provocando que el programador tenga que refactorizar el código fuente hasta que sea legible y extensible.



# Single Responsibility Principle (SRP)

- ▶ En la programación orientada a objetos, el principio de responsabilidad única declara que: Cada clase tiene una única responsabilidad, y que esa responsabilidad debería ser completamente encapsulada por la clase. Todos sus servicios deben ser estrechamente alineados con la responsabilidad asignada.
- ▶ Basado en el principio de cohesión, define la responsabilidad como una razón de cambio. Concluye que una clase o módulo debe tener una, y sólo una, razón para cambiar. **Ejemplo:** considere el módulo que imprime un informe. Dicho módulo se puede cambiar por dos razones. En primer lugar, el contenido del informe puede cambiar. En segundo lugar, el formato del informe se puede cambiar. Estos dos aspectos cambian por muy diferentes causas; uno sustantivo y el otro cosmético.
- ▶ El principio de la responsabilidad única, dice que estos dos aspectos del problema son realmente dos responsabilidades distintas, y deben estar en Por lo tanto en clases o módulos distintos. Sería un mal diseño para acoplar dos cosas que cambian por razones diferentes en distintos momentos.

# Single Responsibility Principle (SRP)

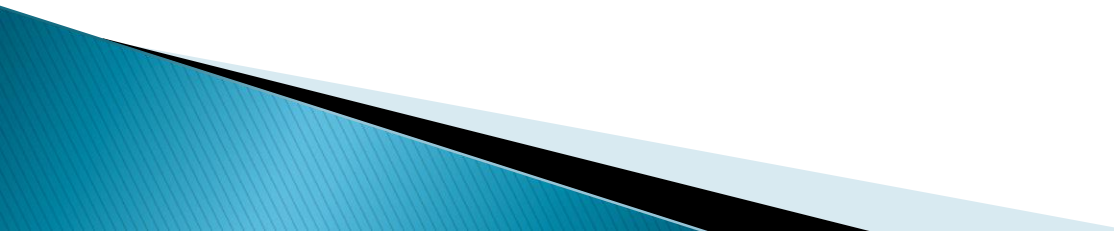
- ▶ La razón por la que es importante mantener a la clase enfocada en una sola responsabilidad es que hace a la clase más robusta. Si hay un cambio en un proceso critico, hay un mayor riesgo que el código dependiente falle si esta en la misma clase.



## SINGLE RESPONSIBILITY PRINCIPLE

Every object should have a single responsibility, and all its services should be narrowly aligned with that responsibility.

# Open/ Closed Principle (OCP)

- ▶ En la programación orientada a objetos, el –open/closed principle– dice que: “Las entidades de software (Clases, módulos, funciones, etc.) deben estar abiertas para su extensión, pero cerrados a la modificación”.
  - ▶ Es decir, una entidad puede permitir que su comportamiento sea modificado sin alterar su código fuente. Esto es especialmente valioso en un entorno de producción, donde los cambios en el código fuente pueden requerir revisiones de código, pruebas unitarias y otros procedimientos para calificar para su uso en el producto: El código que obedece el principio no cambia cuando se extiende y por lo tanto no necesita de tal esfuerzo.
  - ▶ El principio abierto/cerrado ha sido utilizado de dos maneras. De ambas formas se usa la herencia para resolver el aparente dilema, pero los objetivos, las técnicas y los resultados son diferentes.
- 

# Open/ Closed Principle (OCP)

## Meyer's open/closed principle:

- ▶ Una vez terminada, la implementación de una clase sólo puede ser modificado para corregir errores; características nuevas o modificadas requerirían que se creará una clase diferente. Esa clase podría volver a utilizar la codificación de la clase original (padre) a través de la herencia. La subclase derivada podría o no podría tener la misma interfaz que la clase original. Los atributos pueden ser reutilizados por herencia, pero las especificaciones de la interfaz no.

## Polymorphic open/closed principle:

- ▶ Redefinido para referirse al uso de interfaces abstractas, donde las implementaciones se puede cambiar y múltiples implementaciones podrían ser creadas y polimórficamente sustituirse entre sí.
- ▶ En contraste con el uso de Meyer, esta definición aboga por la herencia de las clases base abstractas. Aquí la interfaz puede ser reutilizada a través de la herencia, pero su aplicación no siempre. La interfaz existente se cierra a modificaciones y las nuevas implementaciones deben, como mínimo, implementarla.




# Liskov Substitution Principle (LSP)

- ▶ La sustituibilidad es un principio en la programación orientada a objetos. Afirma Que, en un programa de computadora, si S es un subtipo de T, entonces los objetos de tipo T se pueden reemplazar con objetos de tipo S (es decir, los objetos de tipo S pueden sustituir los objetos de tipo T), sin alterar ninguna de las deseables propiedades del programa.
- ▶ Más formalmente, el principio de sustitución Liskov (LSP) es una definición particular de una relación de subtipos. Se trata de una semántica mas que de una relación sintáctica, tiene la intención de garantizar la interoperabilidad semántica de los tipos en la herencia y los tipos de objeto en particular.
- ▶ Barbara Liskov:
- ▶ “Sea  $q(x)$  sea una propiedad demostrable sobre objetos  $x$  de tipo T. Entonces  $q(y)$  debe ser demostrable para los objetos  $y$  de tipo S, donde S es un subtipo de T.”



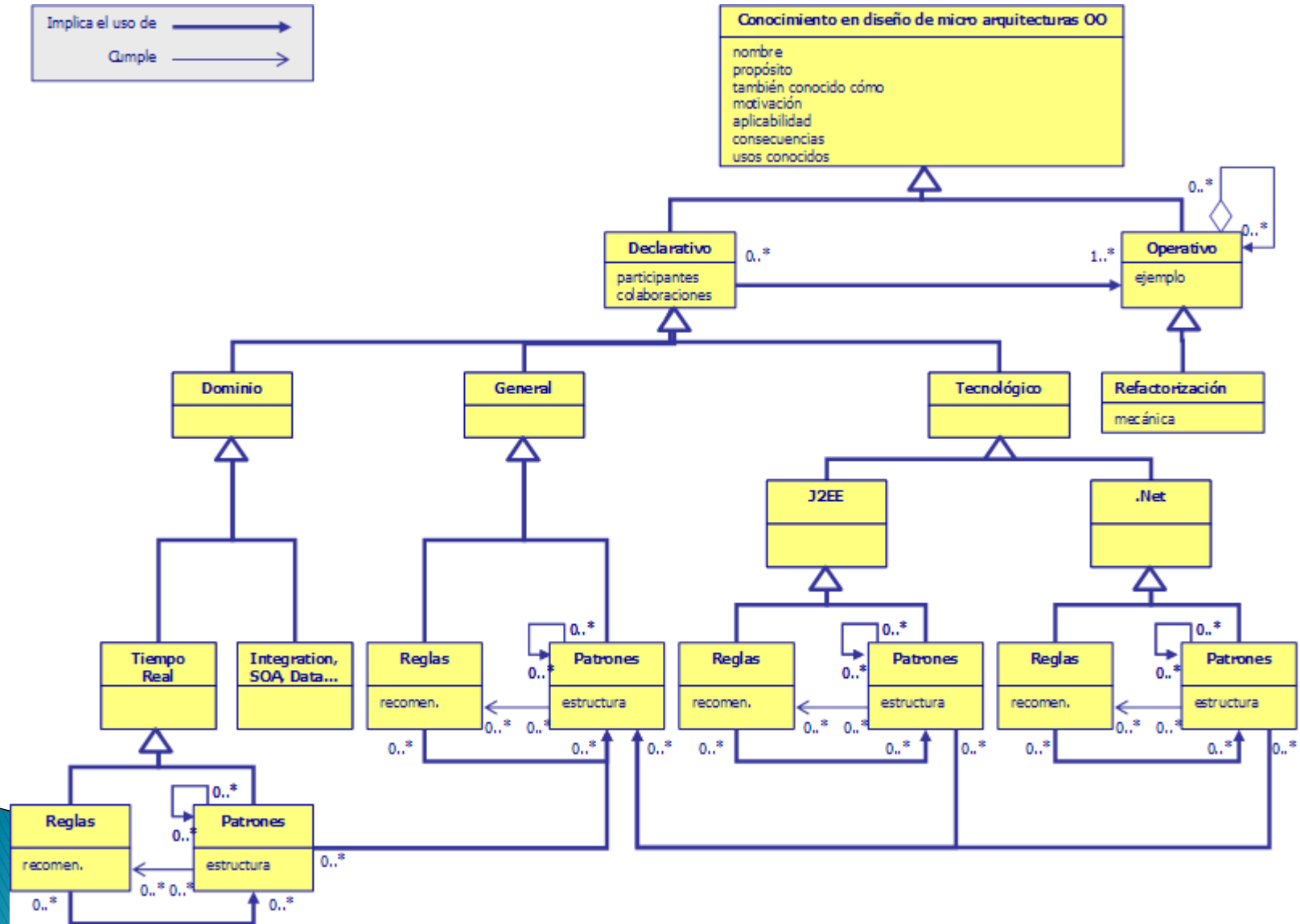
# Interface Segregation Principle (ISP)

- ▶ El principio de la segregación de interfaz (ISP) declara que los clientes no deben ser forzados a depender de métodos que no utilizan.
  - ▶ ISP separa grandes divisiones en otras más pequeñas y más específicas para que los clientes sólo tendrán que saber acerca de la métodos de que son de interés para ellos. ISP tiene la intención de mantener el sistema lo mas desacoplado posible y fácil de refactorizar, modificar y redistribuir.
  - ▶ Dentro del diseño orientado a objetos, las interfaces proporcionan capas de abstracción que facilitar la explicación conceptual del código y crean una barrera que evita dependencias. El uso de interfaces para describir con más detalle el propósito de que el software es a menudo una buena practica.
  - ▶ Los sistemas pueden llegar a ser tan unidos a múltiples niveles que ya no es posible hacer cambios en un solo lugar sin necesidad de muchos cambios adicionales. El uso de una interfaz o una clase abstracta puede prevenir este efecto secundario.
- 

# Dependency inversion principle (DIP)

- ▶ En la programación orientada a objetos, el principio de inversión de dependencia se refiere a una forma específica del desacoplamiento de los módulos de software. Siguiendo este principio, las relaciones de dependencia convencionales son establecidas a alto nivel, las políticas de definición de módulos a bajo nivel, los módulos de dependencia se invierten (es decir, al revés), este modo de representación independiente de alto nivel de los módulos de los detalles de implementación del módulo de bajo nivel. El principio:
  - ▶ A. Módulos de alto nivel no debe depender de los módulos de bajo nivel. Ambos deben depender de las abstracciones.
  - ▶ B. Las abstracciones no deben depender de los detalles. Detalles deben depender de las abstracciones.
- ▶ El principio Invierte la forma en que algunas personas pueden pensar en el diseño orientado a objetos, que dicta tanto de nivel alto y bajo objetos deben depender de la misma abstracción.

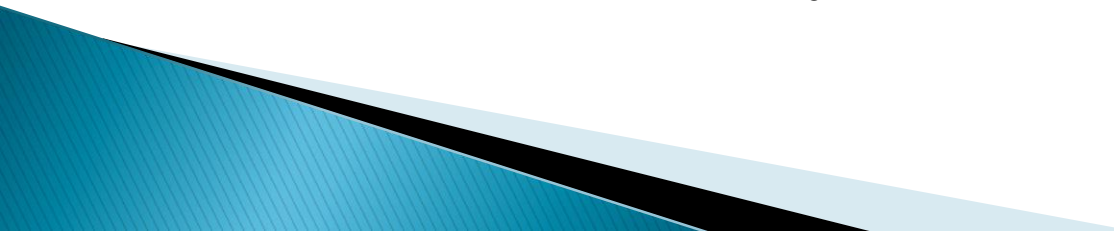
# Otros principios de Diseño



# Don't repeat yourself (DRY)

- ▶ En la ingeniería de software, “Don't repeat yourself ” (DRY) es un principio de desarrollo de software, el principio esta destinado a la reducción repetición de la información de todo tipo, es especialmente útil en las arquitecturas multi-nivel.
- ▶ El principio DRY se dice: "Cada pieza de información debe tener una única e inequívoca representación y autoridad dentro del sistema."
- ▶ Aplica bastante a los esquemas de bases de datos, planes de prueba, el sistema de construcción, incluso la documentación.
- ▶ Cuando se aplica correctamente el principio DRY, la modificación de un solo elemento del sistema no requiere un cambio en otra lógica elementos no relacionados. Además, los elementos de que están relacionadas lógicamente tendrán todo cambio de forma predecible y uniforme, se mantienen en sincronía Así.

# Inversion of control(IOC)

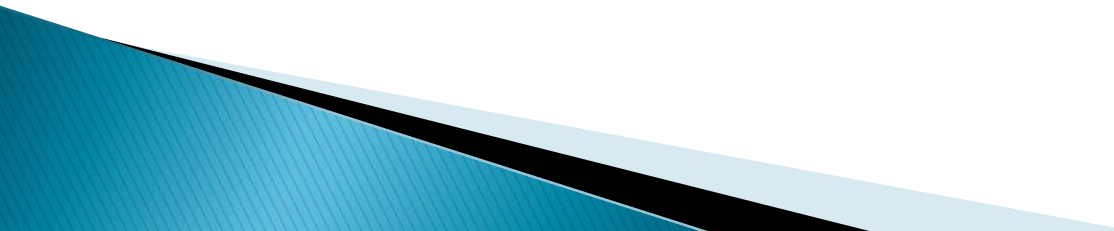
- ▶ En la ingeniería de software, la inversión de control (IOC). Describe el diseño en porciones custom-written del programa. Y recibe el flujo de control de una biblioteca reutilizable y genérica.
  - ▶ El diseño de la arquitectura de software con este principio invierte el control con respecto a la programación de procedimientos tradicional: En la programación tradicional, el código personalizado que expresa el propósito del programa llama a las bibliotecas reutilizables para cuidar de tareas genéricas, pero con la inversión de control, es el código reutilizable que llama a la cola de código de propósito específico.
  - ▶ La inversión de control se utiliza para aumentar la modularidad del programa y hacer que sea extensible y tiene aplicaciones en la programación orientada a objetos y otros paradigmas de programación.
  - ▶ El término está relacionado pero es diferente del principio de inversión de la dependencia, que se ocupa de las dependencias de desacoplamiento entre las capas de alto nivel y de bajo nivel a través de abstracciones compartidas.
- 

# Inversión of control(IOC)

Inversión de Control lleva la fuerte connotación que el código reutilizable y el código específico se desarrollan de forma independiente, aunque operan juntos en una aplicación.

- ▶ “Software frameworks”, “callbacks”, “schedulers”, “event loops” y “dependency injection” son ejemplos de patrones de diseño que siguen la inversión del principio de control, aunque el término es más comúnmente utilizado en el contexto de la programación orientada a objetos.

Propósitos:

- ▶ Para desacoplar la ejecución de la implementación.
  - ▶ El módulo se enfoca en la tarea para la que está diseñado.
  - ▶ Para liberar a los módulos de las suposiciones acerca de cómo otros sistemas hacen lo que hacen.
  - ▶ Reducir efectos secundarios al reemplazar un módulo.
  - ▶ Se refiere en broma al "Principio de Hollywood: No nos llames, nosotros te llamamos".
- 

# Lecturas

- ▶ [http://en.wikipedia.org/wiki/Inversion\\_of\\_control](http://en.wikipedia.org/wiki/Inversion_of_control)
- ▶ [http://programmer.97things.oreilly.com/wiki/index.php/Don't\\_Repeat\\_Yourself](http://programmer.97things.oreilly.com/wiki/index.php/Don't_Repeat_Yourself)
- ▶ <http://www.blackwasp.co.uk/SOLIDPrinciples.aspx>
- ▶ [http://en.wikipedia.org/wiki/Liskov\\_substitution\\_principle](http://en.wikipedia.org/wiki/Liskov_substitution_principle)



## The SOLID Principles

This is the first article in a series of six that describe the SOLID principles of object-oriented design and programming. The SOLID principles provide five guidelines that, when followed, can dramatically enhance the maintainability of software.



## Single Responsibility Principle

The second article in the SOLID Principles series describes the Single Responsibility Principle (SRP). The SRP states that each class or similar unit of code should have one responsibility only and, therefore, only one reason to change.



## Open / Closed Principle

The third article in the SOLID Principles series describes the Open / Closed Principle (OCP). The OCP states that all classes and similar units of source code should be open for extension but closed for modification.



## Liskov Substitution Principle

The fourth article in the SOLID Principles series describes the Liskov Substitution Principle (LSP). The LSP specifies that functions that use pointers of references to base classes must be able to use objects of derived classes without knowing it.



## Interface Segregation Principle

The fifth article in the SOLID Principles series describes the Interface Segregation Principle (ISP). The ISP specifies that clients should not be forced to depend upon interfaces that they do not use. Instead, those interfaces should be minimised.



## Dependency Inversion Principle

The sixth and final article in the SOLID Principles series describes the Dependency Inversion Principle (DIP). The DIP states that high level modules should not depend upon low level modules and that abstractions should not depend upon details.



**Gracias**