Tema 5: Principios Generales de Diseño

Parte 1 - Principios de Diseño SOLID

Evolución y Adaptación de Software

Carlos E. Cuesta, ETSII, URJC

Universidad
Rey Juan Carlos

Índice

Principios Generales de Diseño
Introducción

- Aunque se basan en principios bien conocidos (y generales), centramos su definición en diseño OO
  - Reformulan el clásico *Cohesión vs. Acoplamiento*
  - Tienen una relación directa con los patrones de diseño
  - Son clásicos en el diseño OO
    - Hoy también considerados como principios "ágiles"
- Su formulación más conocida se basa en la definición de Robert Martin (no la más "académica")
  - SOLID = Single (SRP) + Open (OCP) + Liskov (LSP)+ Interface (ISP) + Dependency (DIP)

Principio de Responsabilidad Única (Single Responsibility Principle, SRP)

> *Toda clase debería tener un único motivo para cambiar*
>
> *R. Martin, 1996*

- Una reformulación del concepto clásico de cohesión
- Martin define "responsabilidad" como un "motivo para cambiar" a nivel de diseño
  - No de evolución, aunque la conexión es evidente
  - Responsabilidad: "encargo asignado a un único actor, con respecto a una tarea única de negocio"
- Si hay más de un motivo para cambiar, la clase se encarga de más de una cosa, y por tanto debería haber más de una clase

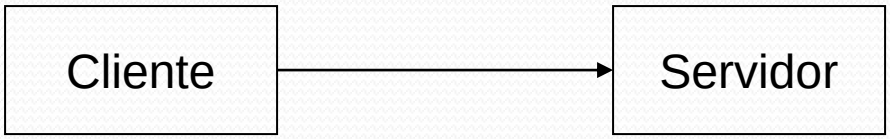
Principio Abierto-Cerrado (Open-Closed Principle. OCP)

> Las entidades software deben estar abiertas para su extensión pero cerradas para su modificación.
>
> *B. Meyer, 1988 / R. Martin, 1996*

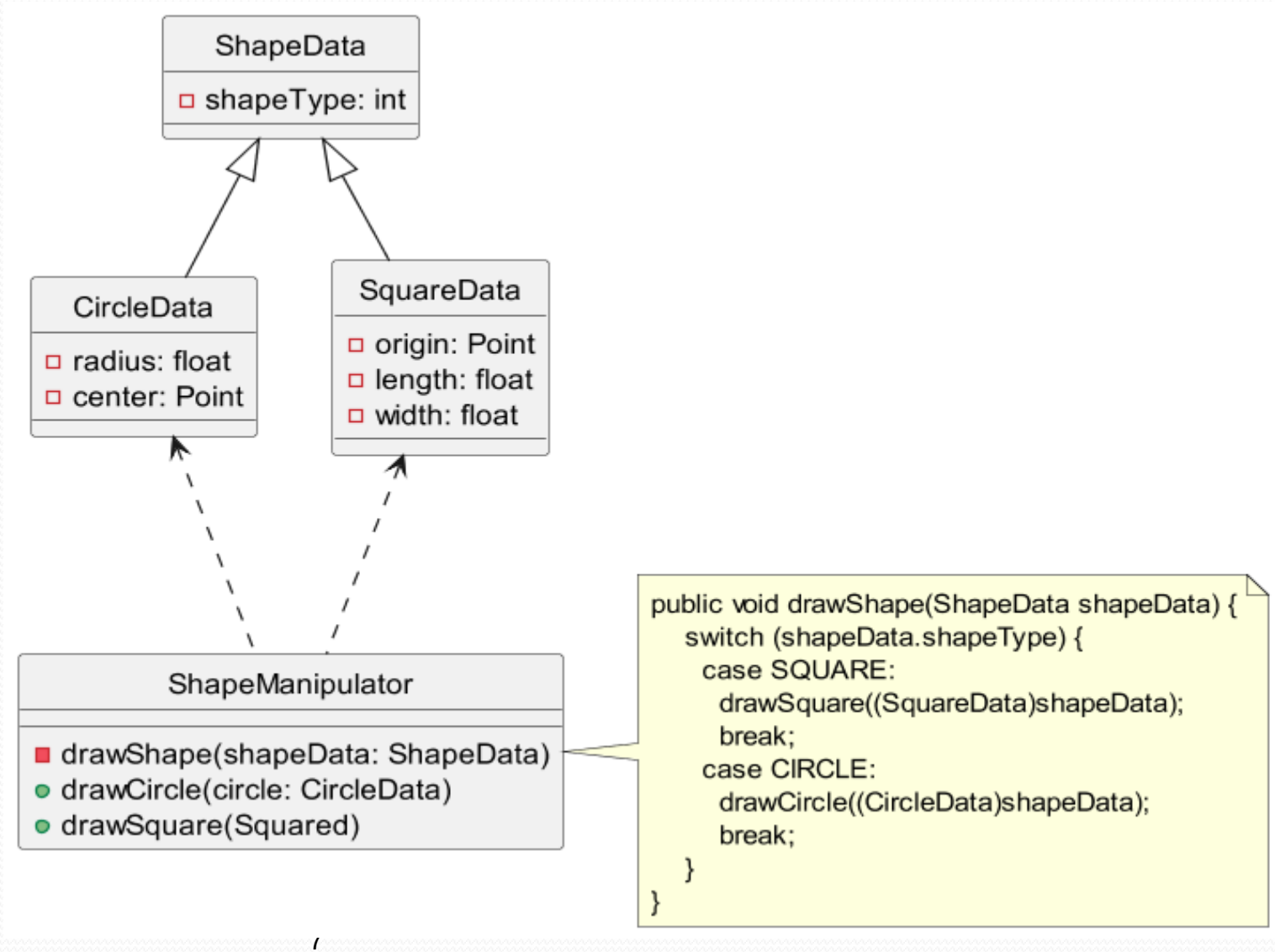
- Deben ser abiertos para extensión
  - El comportamiento del módulo puede ser extendido.
- Deben ser cerrados para modificación
  - El código fuente del módulo es inalterable.
- Los módulos han de ser escritos de forma que puedan extenderse sin que sea necesario modificarlos.
- ¿Cómo?
  - Abstracción y polimorfismo.

Principio Abierto Cerrado. (2)

- La clave del principio está en la abstracción.
- Se pueden crear abstracciones que sean fijas y que representen un número ilimitado de posibles comportamientos.
- Las abstracciones son un conjunto de clases base, y el grupo ilimitado de los posibles comportamientos viene representado por todas las posibles clases derivadas.
- Un módulo está cerrado para su modificación al depender de una abstracción que es fija. Pero el comportamiento del módulo puede ser extendido mediante la creación de clases derivadas.
- El siguiente diseño NO cumple el Principio Abierto/Cerrado.
  - Las clases cliente y servidor concretas. La clase cliente usa a la clase servidor.
  - Si se desea que un objeto cliente use un objeto servidor diferente, se debe cambiar la clase cliente para nombrar la nueva clase servidor.



Principio Abierto-Cerrado. Ejemplo.

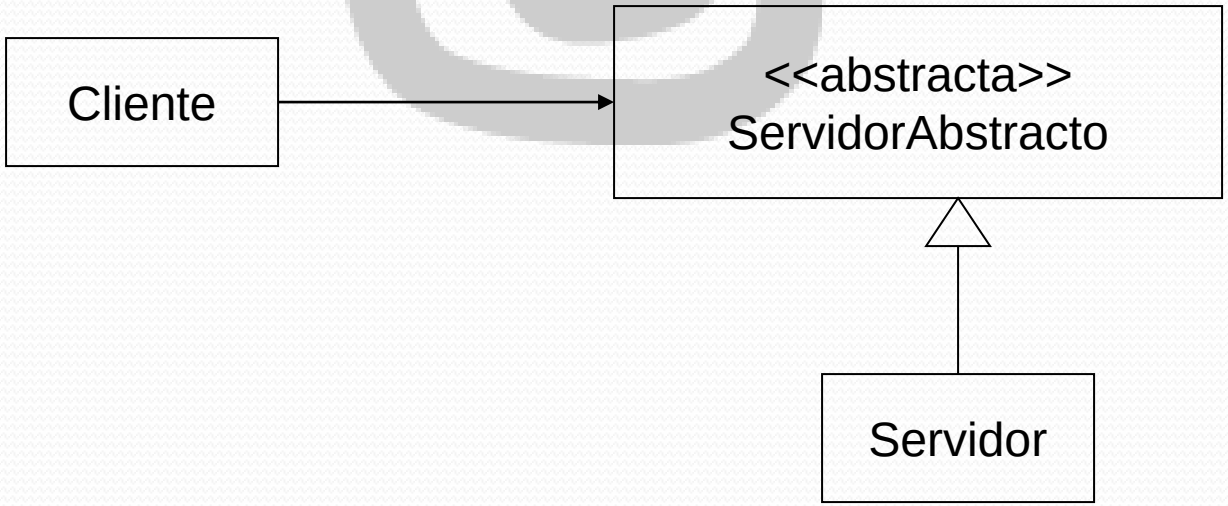


Principio Abierto-Cerrado. Ejemplo. Discusión

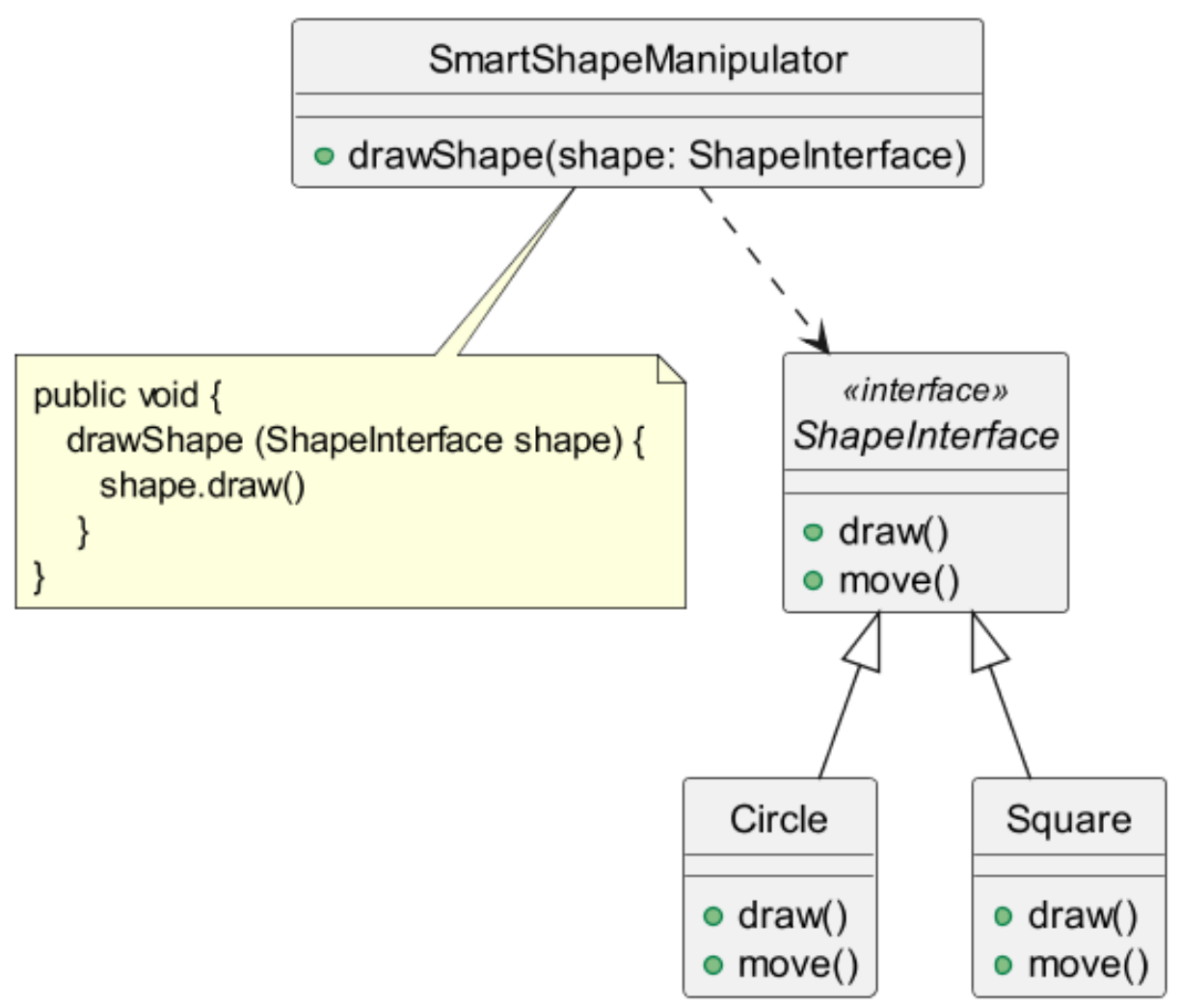
- Si se necesita crear una nueva figura, un Triángulo por ejemplo, necesitamos modificar la función: ‘drawShape()‘.
- En una aplicación compleja la sentencia switch/case se repite una y otra vez para cada tipo de operación que tenga que realizarse sobre una figura.
- Peor aún, cada módulo que contenga este tipo de sentencia switch/case retiene una dependencia sobre cada posible figura que haya que dibujar, por lo tanto, cuando se realice cualquier tipo de modificación sobre una de las figuras, los módulos necesitarán una nueva compilación y posiblemente una modificación.
- Si los módulos de una aplicación cumplen con el principio abierto/cerrado, las nuevas características pueden añadirse a la aplicación añadiendo nuevo código en lugar de cambiar el código en funcionamiento.

Principio Abierto-Cerrado. Ejemplo.

- Este diseño cumple el principio abierto/cerrado.
  - La clase ServidorAbstracto es una clase abstracta
  - La clase Cliente usa esa abstracción y, por lo tanto, los objetos de la clase Cliente utilizarán los objetos de las clases derivadas de la clase ServidorAbstracto.
  - Si se desea que los objetos de la clase Cliente utilicen diferentes clases Servidor, se deriva una nueva clase de la clase ServidorAbstracto y la clase Cliente permanece inalterada.



Principio Abierto-Cerrado. Ejemplo.



Principio de Sustitución de Liskov (LSP).

- Los elementos clave del OCP son: Abstracción y Polimorfismo
  - Implementados mediante herencia
  - ¿Cómo medimos la calidad de la herencia?

La herencia ha de garantizar que cualquier propiedad que sea cierta para los objetos supertipo también lo sea para los objetos subtipo.

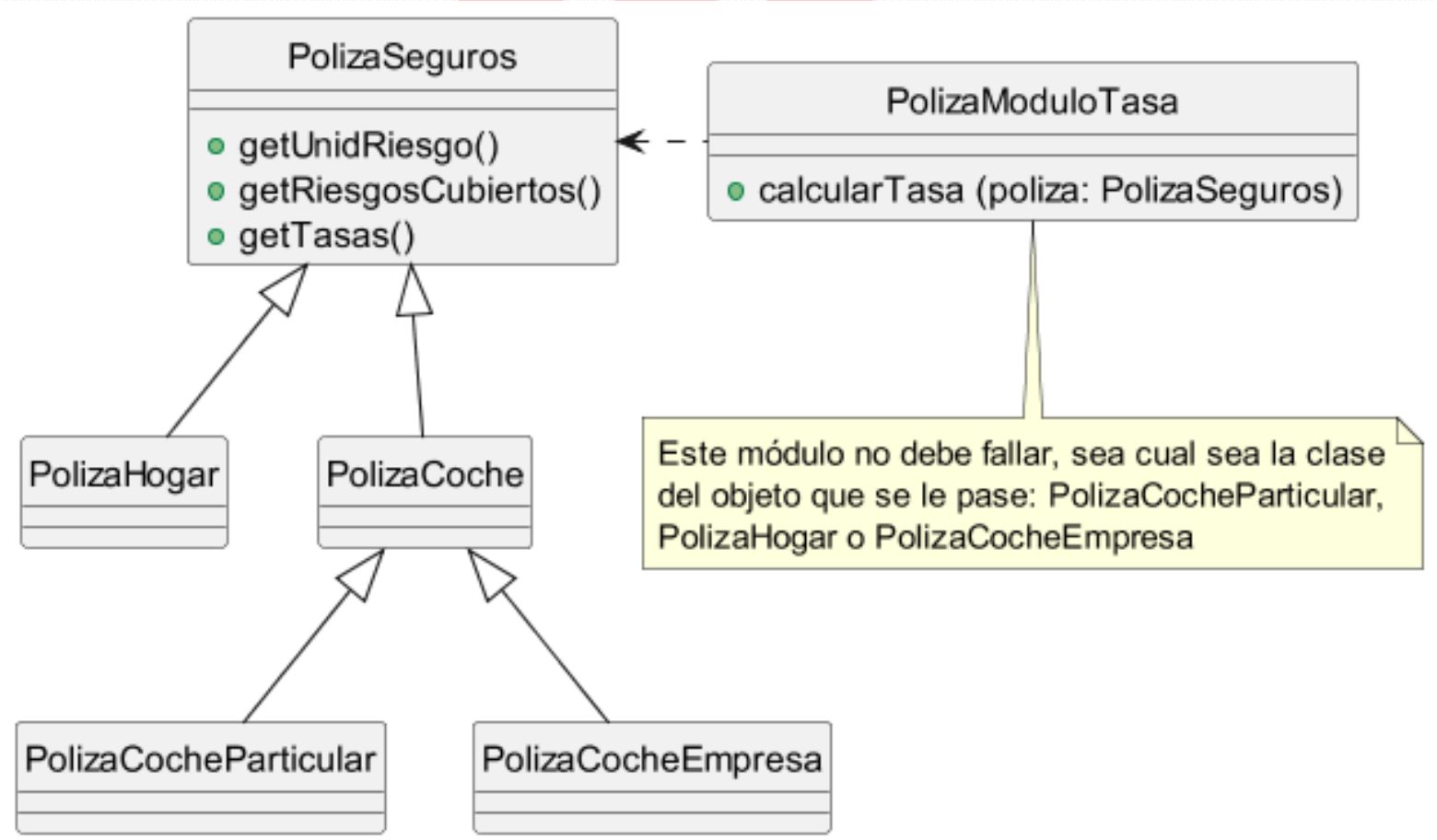
B. Liskov, 1987

Las clases derivadas deber ser utilizables a través de la interfaz de la clase base sin necesidad de que el usuario conozca la diferencia.

R. Martin, 1997

En resumen: Las subclases deben ser sustituibles por sus clases base

Principio de Sustitución de Liskov. Ejemplo



Principio de Sustitución de Liskov

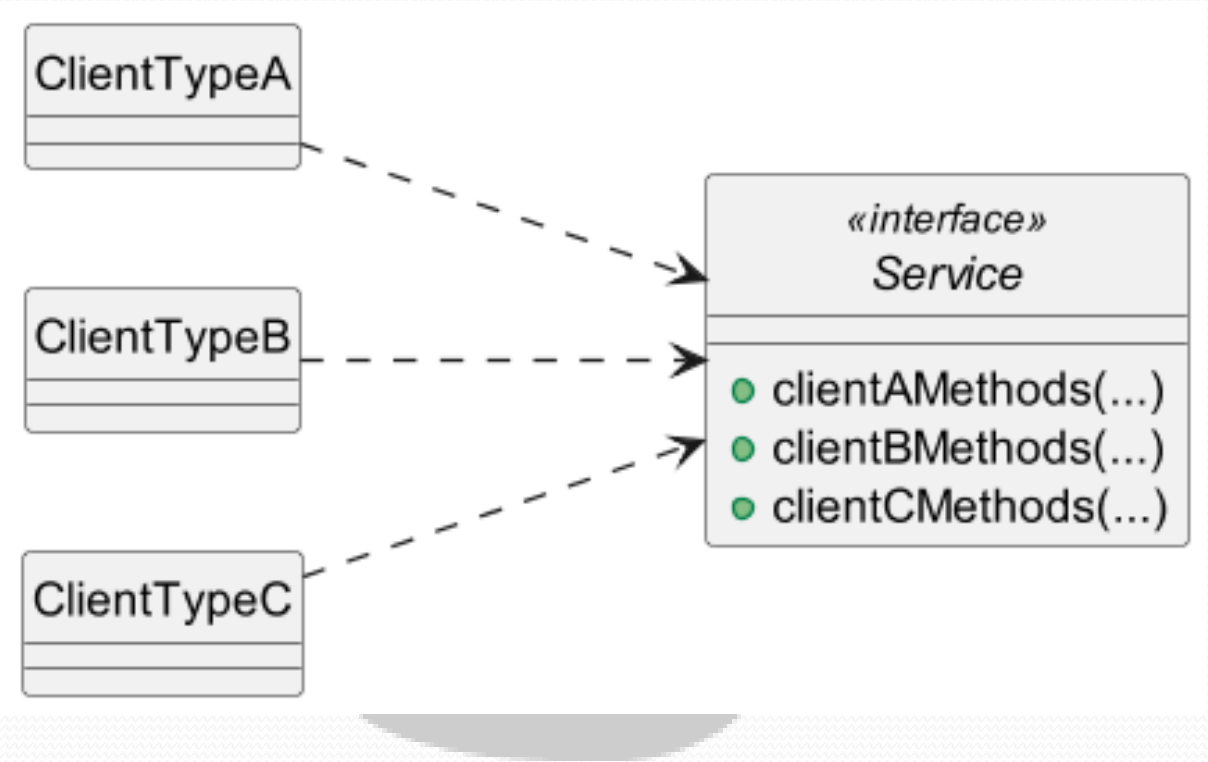
- Un cliente de la clase base debe seguir funcionando adecuadamente si una clase derivada de esa clase base se le pasa a dicho cliente.
  - En otras palabras: Si alguna función toma un argumente del tipo PolizaSeguros, debería ser legal pasarle una instancia de PolizaCocheParticular ya que deriva directa o indirectamente de PolizaSeguros.
- El principio de Liskov establece que la relación IS_A en DOO se refiere al comportamiento externo público, que es comportamiento del que dependen los clientes.
  - Dilema círculo/elipse.
- Las violaciones del principio de Liskov son violaciones enmascaradas del principio abierto/cerrado.
  - Ésta es, precisamente, la razón por la que este principio (aparentemente tan genérico) es relevante en este contexto

Principio de Separación de la Interfaz (ISP)

> Es mejor muchas interfaces de clientes específicos que una sola interfaz de propósito general.
>
> Los clientes no deben ser forzados a depender de interfaces que no utilizan
>
> *R. Martin, 1996*

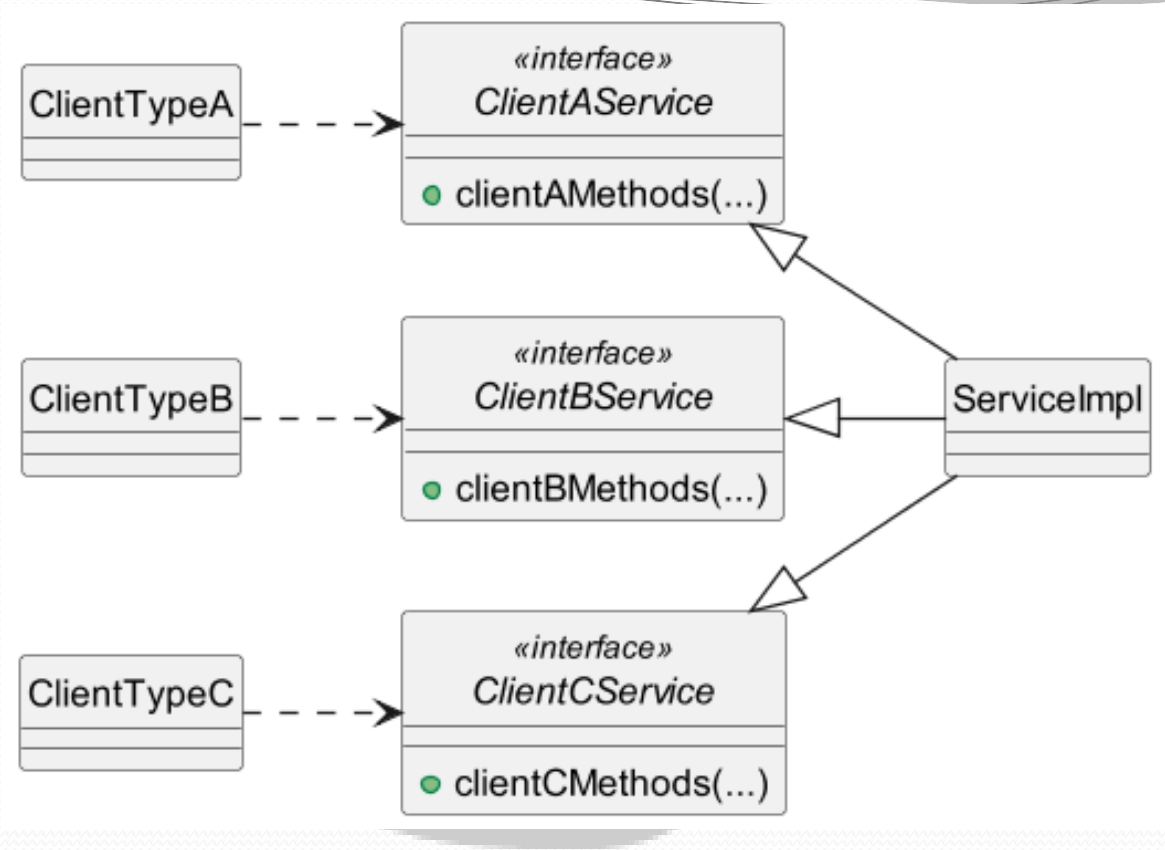
- Principio estructural que combate las desventajas de las clases con interfaces grandes.
- Las clases con interfaces grandes son clases cuyas interfaces no están cohesionadas.
- Cuando los clientes se ven forzados a depender de interfaces que no utilizan, éstos se ven afectados por los cambios de dichas interfaces. El resultado es un acoplamiento entre todos los clientes.

Principio de Separación de la Interfaz (ISP)



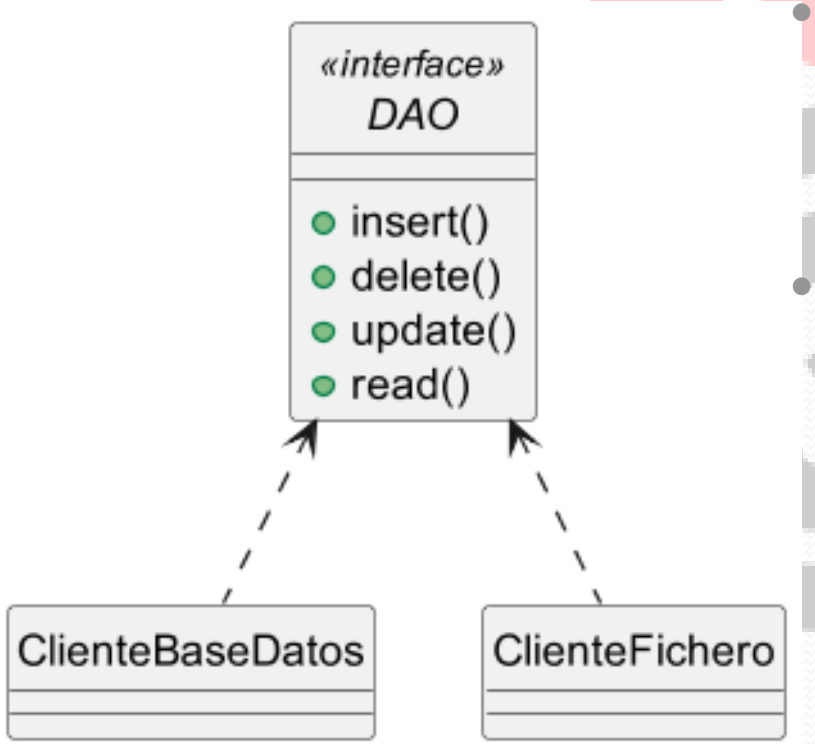
- Si se tiene una clase con varios clientes, en lugar de cargar la clase con todos los métodos que los clientes necesitan, crear interfaces específicos para cada tipo de cliente y que la clase herede de todos ellos (herencia múltiple)

Principio de Separación de la Interfaz (ISP)



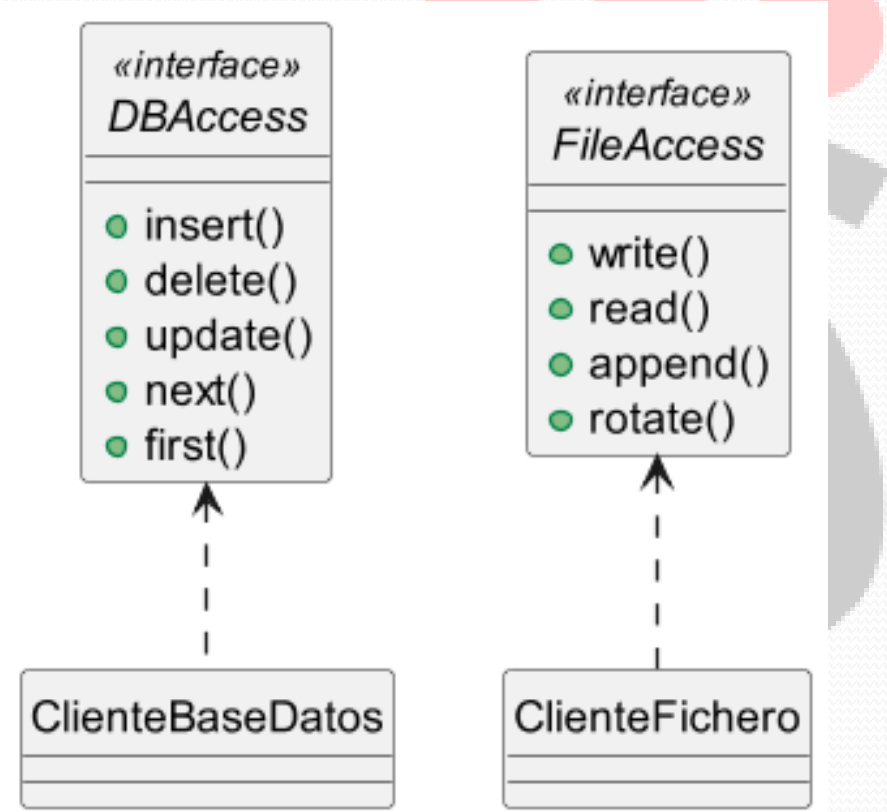
- El ISP **no recomienda** que cada clase que utilice un servicio tenga su propia clase interfaz especial del cual herede el servicio. Lo que indica es que los clientes han de categorizarse por tipo y crear una interfaz para cada tipo de cliente. Si dos o mas tipos diferentes de cliente necesitan el mismo método, el método ha de ser añadido en todas las interfaces.

Principio de Separación de la Interfaz. Ejemplo



- DAO – Objeto Acceso a Datos.
- ¿Qué pasaría si la fuente de datos fuese sólo de lectura?
  - insert() y update() sobrarían
  - El objeto que implemente DAO tiene que proporcionar una implementación nula.
- ¿Y si necesitamos cambiar de fichero una vez que cierta cantidad de datos se escriban en un fichero?
  - Se necesitan métodos adicionales.
  - Hay que añadir flexibilidad para que ClienteFichero pueda incorporar la característica de añadir datos tras una rotación.
  - Pero la implementación de ClienteBaseDatos se rompería, se necesita cambiarla para proporcionar una implementación nula de la nueva característica. Se violaría el OCP.

Principio de Separación de la Interfaz. Ejemplo



Todos los problemas desaparecen si las interfaces, que se usan de manera diferente, están separadas

Este ejemplo puede parecer extraño, porque se centra en lo que necesita el *cliente*, no en la implementación de las interfaces

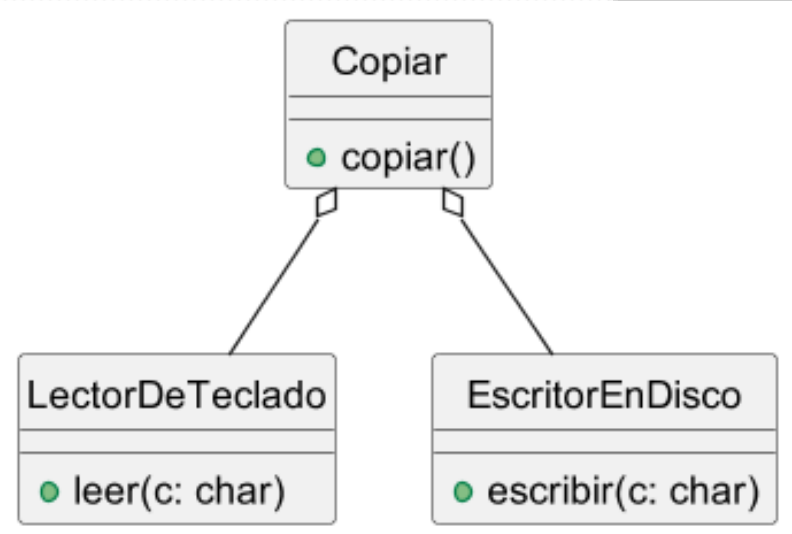
Principio de Inversión de la Dependencia (DIP)

A. Los módulos de alto nivel no deben depender de los módulos de bajo nivel. Ambos deben depender de las abstracciones.
B. Las abstracciones no deben depender de los detalles. Los detalles deben depender de las abstracciones.

R. Martin, 1996

- El OCP establece el objetivo, el DIP establece el mecanismo.
- Indica la dirección que tienen que tomar todas las dependencias en un diseño orientado al objeto.
- La Inversión de Dependencia es la estrategia de depender de interfaces o funciones y clases abstractas, en lugar de depender de funciones y clases concretas.
- Cada dependencia en el diseño deberá tener como destino una interfaz o una clase abstracta. Ninguna dependencia debería tener como destino una clase concreta.

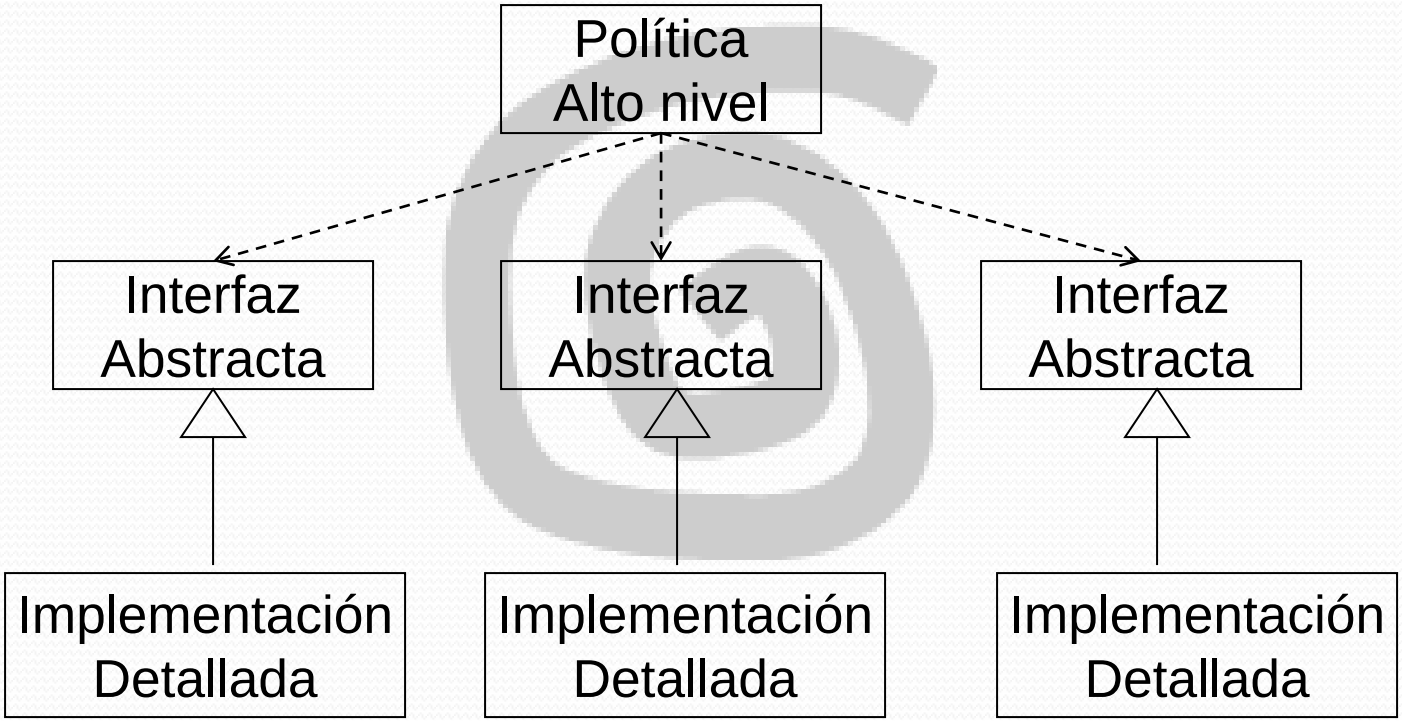
Principio de Inversión de la Dependencia (DIP)



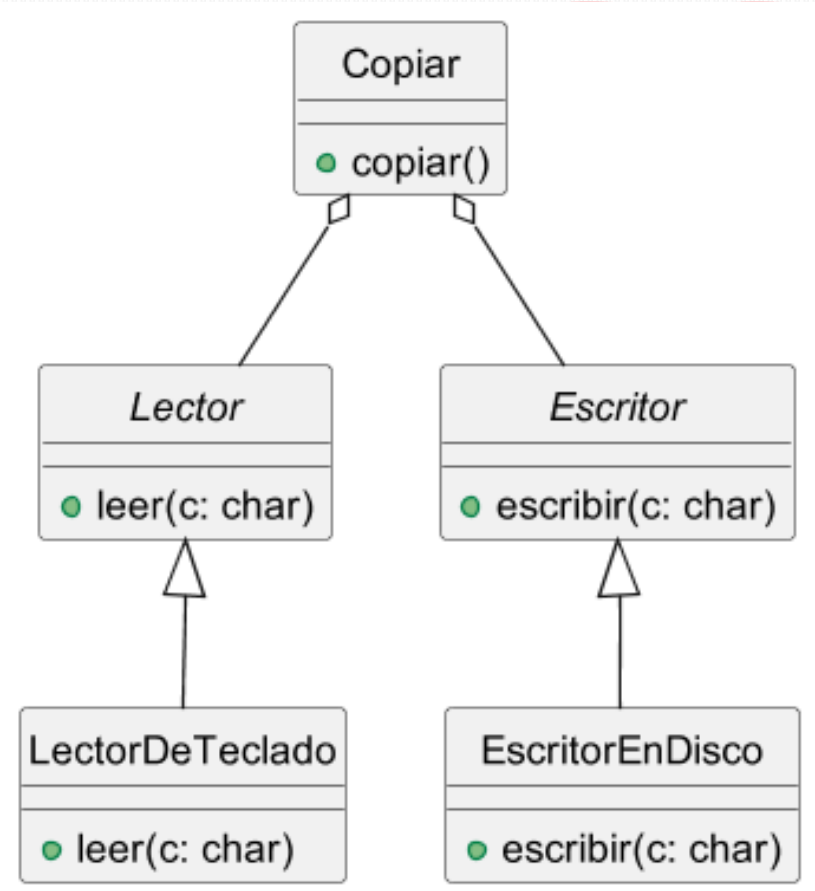
- Supongamos el caso siguiente
- Diseñamos la clase Copiar para que actúe como un keylogger: cada vez que se introduce un carácter por teclado (LectorDeTeclado), se graba ese carácter en el disco, p.e. en un fichero (EscritorEnDisco).
- Obviamente, la estrategia de Copiar es genérica, pero depende de dos implementaciones de (muy) bajo nivel. Cualquier cambio en éstas obliga a cambiar a aquélla.
- Pero las políticas generales no deben depender de los detalles, sino de las abstracciones

Principio de Inversión de la Dependencia (DIP)

Estructura de las dependencias en una arquitectura Orientada al Objeto



Principio de Inversión de la Dependencia (DIP)



- La clase Copiar obtiene un carácter del Lector y se la manda al Escritor independientemente de los módulos de bajo nivel.
- La clase Copiar depende de abstracciones y los Lectores y Escritores especializados dependen de las mismas abstracciones.
- Se puede reutilizar Copiar independientemente de los dispositivos físicos.
- Se pueden añadir nuevos tipos de *lectores* y *escritores* sin que la clase copiar dependa en absoluto de ellos.

Principio de Inversión de Dependencia.Consideraciones

- La motivación que subyace en el DIP es prevenir dependencias de módulos volátiles.
- Normalmente, las cosas concretas cambian con más frecuencia que las cosas abstractas.
- Las abstracciones son "puntos de enganche" que representan los "sitios" sobre los que el diseño puede ser aplicado o extendido, sin que se modifiquen (OCP)
- El DIP está relacionado con la heurística de DOO: "Diseñe interfaces, no implementaciones"
- Hay literalmente decenas de implicaciones del DIP. Se puede considerar como el principio fundamental
  - Se implica también en la *inversión de control (IoC)*

Principio de Inversión de Dependencia. Niveles.

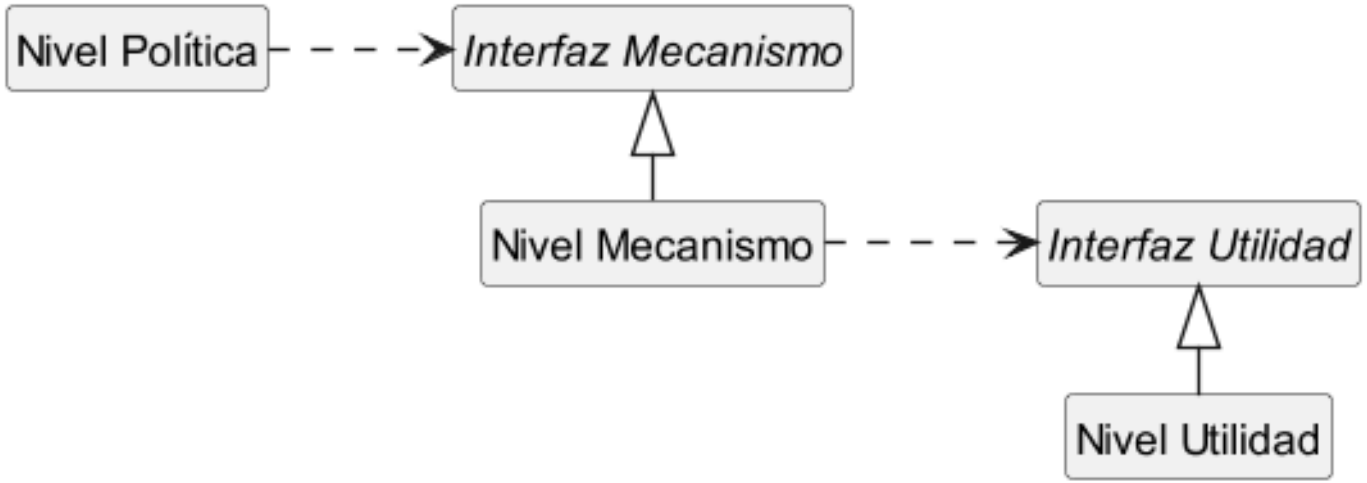
> "... todas las arquitecturas orientadas al objeto bien estructuradas tienen niveles claramente definidos, donde cada nivel ofrece algún conjunto de servicios coherentes a través de una interfaz bien definida y controlada".
>
> *G. Booch, 1996*

- Heurística: Evitar dependencias transitivas.
  - Evitar estructuras en las que los niveles altos dependan de abstracciones de los niveles bajos.
- Solución
  - Utilizar herencia y clases abstractas ancestro para eliminar de forma efectiva las dependencias transitivas.

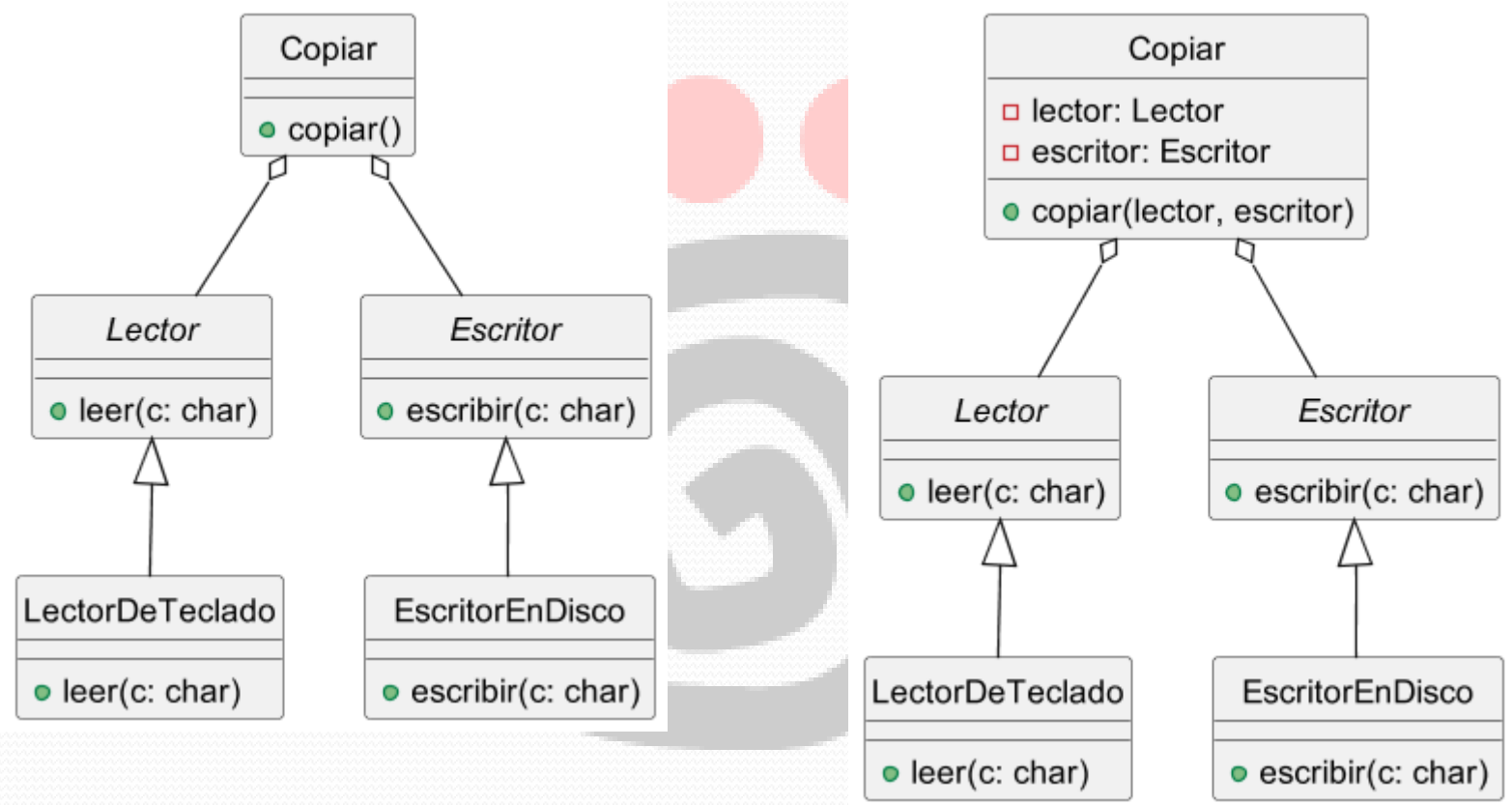
Principio de Inversión de Dependencia. Niveles.

Estructura incorrecta de niveles

Solución con niveles abstractos



Principio de Inversión de la Dependencia (DIP)



- Un comentario final: no es lo mismo una *inversión* de dependencias que una *inyección* de dependencias
  - Toda inyección implica una inversión (o debería, si se está usando bien); pero lo contrario no es cierto

Principios SOLID y Patrones de Diseño

- Cuando se siguen los principios de diseño expuestos se descubre que las estructuras se repiten una y otra vez.
  - Estas estructuras repetitivas es lo que se conoce como patrón de diseño.

- Por otra parte, los patrones son soluciones probadas a los problemas de diseño más comunes con los que se enfrentan los diseñadores en orientación al objeto.
  - Gestión de dependencias
  - Diseño de sistemas que puedan evolucionar a medida que los requisitos cambien.
  - Maximizar la reutilización de diseños.
  - ….
  - Esto ya debería sonarnos de algo ☺

Principios vs. Patrones de Diseño

Los patrones de diseño ayudan a evitar rediseños al asegurar que un sistema *pueda cambiar* de forma concreta

- Causas Comunes de Rediseño
  1. Crear un objeto indicando la clase
  2. Dependencia de operaciones específicas
  3. Dependencia de plataformas hardware o software
  4. Dependencia sobre representación de objetos.
  5. Dependencias de algoritmos
  6. Acoplamiento fuerte entre clases
  7. Extender funcionalidad mediante subclases
  8. Incapacidad de cambiar clases convenientemente

- Patrones de Diseño
  1. Abstract factory, Method factory, Prototype
  2. Chain of Responsability, Command
  3. Abstract factory, Bridge
  4. Abstract factory, Bridge, Memento, Proxy,
  5. Builder, Iterator, Strategy, Template Method, Visitor
  6. Abstract factory, Bridge, Chain of Responsability, Command, Facade, Mediator, Observer
  7. Bridge, Chain of Responsability, Composite, Decorator, Observer, Strategy
  8. Adapter, Decorator, Visitor