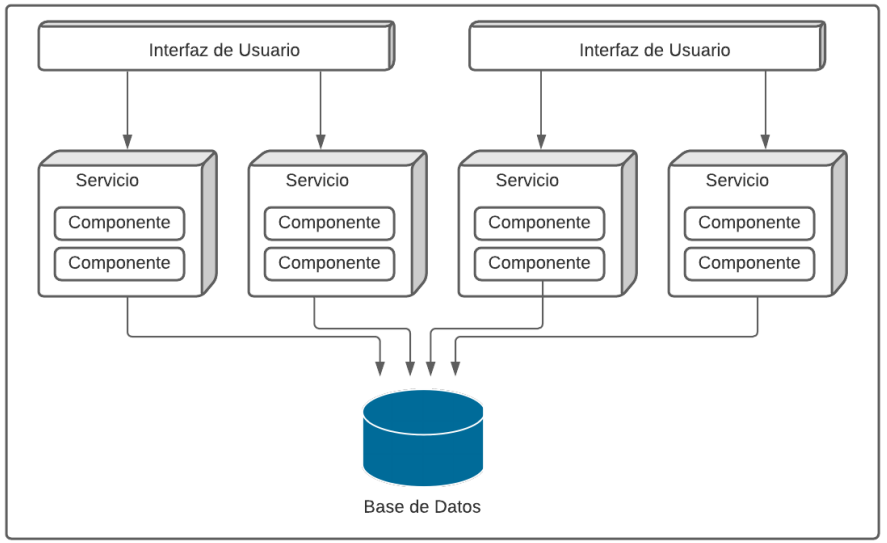
DISEÑO Y ARQUITECTURA DE SOFTWARE

Semana 04

Principios y procesos en las arquitecturas de software



Docente: Danyer A. Valencia LLamoca

Correo: c24454@utp.edu.pe

Logro de la Sesión

Al finalizar la sesión, el estudiante elabora un diseño arquitectónico basado en los requerimientos fundamentales del producto y los Estilos Arquitectónicos.

Utilidad

Al finalizar la unidad, el estudiante comprende los conceptos de arquitectura para poder realizar una correcta implementación de un sistema escalable.

.

CONTENIDO DE LA SESION

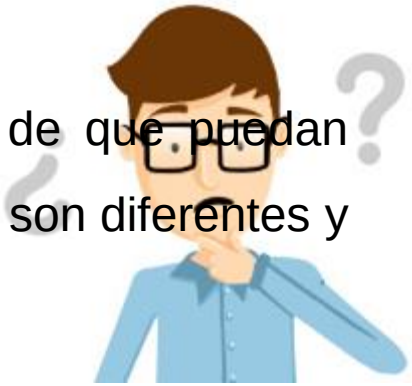
- Principios del Diseño.
- Acoplamiento y cohesión.
- Conceptos de Orientación a Objetos .
- Requerimientos de Software.
- Especificación de Requerimientos de Software (SRS).

RECORDEMOS

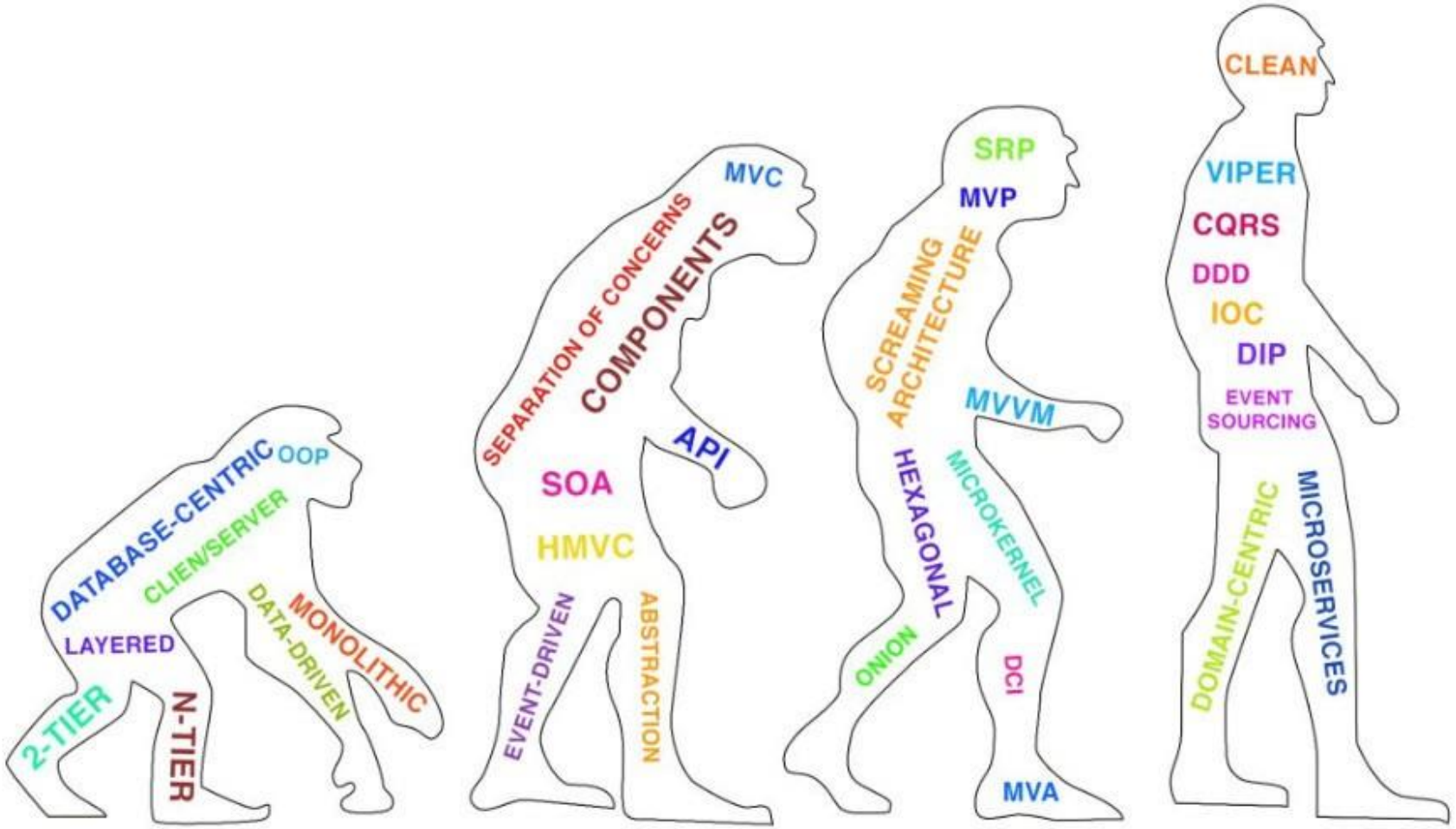
La **arquitectura de software** es el diseño de más alto nivel de la estructura de un sistema, el cual consiste en un conjunto de patrones y abstracciones que proporcionan un marco claro para la implementación del sistema.

Un **estilo arquitectónico** establece un marco de referencia a partir del cual es posible construir aplicaciones que comparten un conjunto de atributos y características mediante el cual es posible identificarlos y clasificarlos.

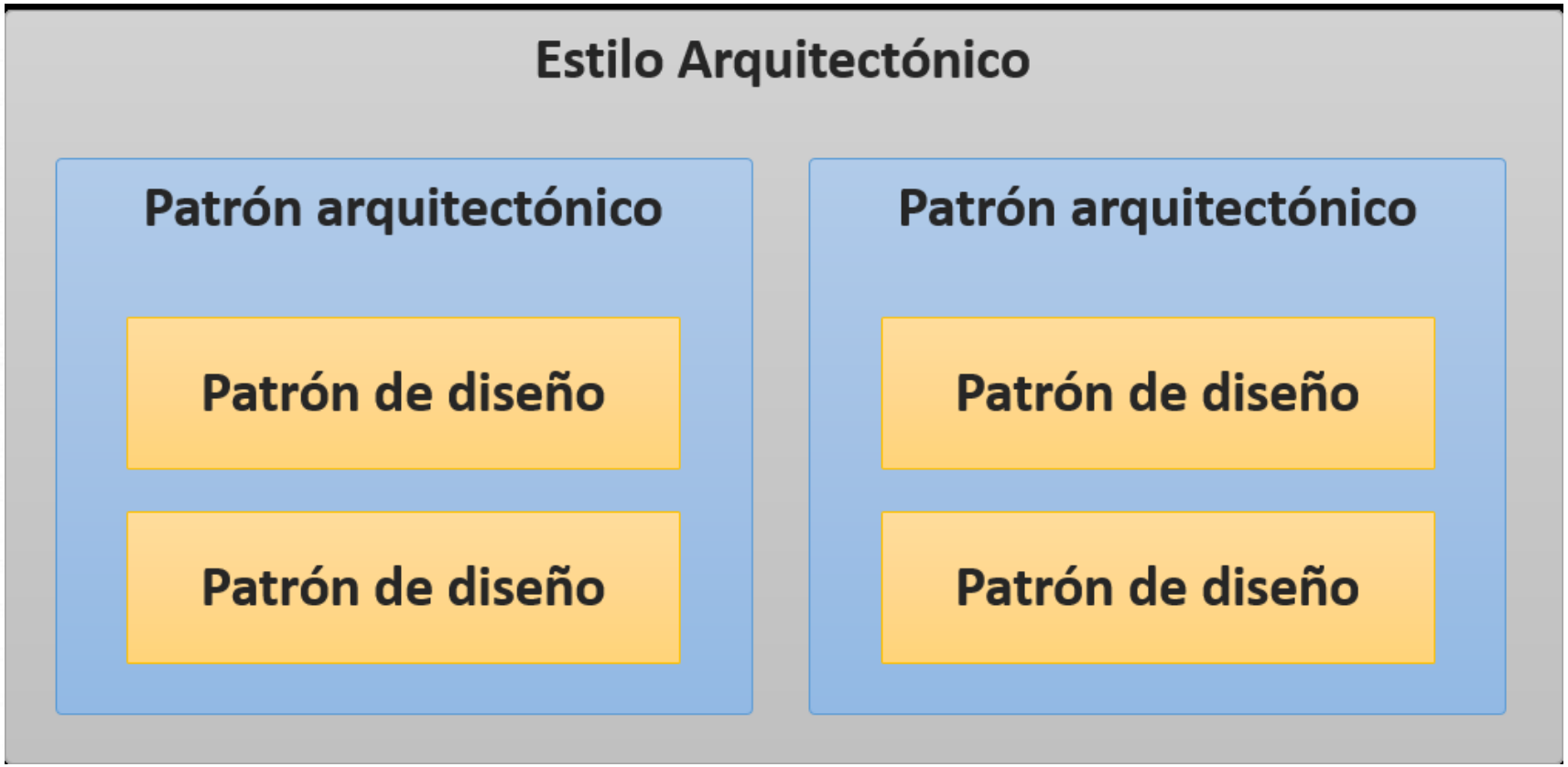
Los **estilos arquitectónicos son diferentes a los patrones arquitectónicos,** pesar de que puedan parecer similares por su nombre, los patrones arquitectónicos y los estilos arquitectónicos son diferentes y por ningún motivo deben de confundirse



RECORDEMOS


2-TIER
N-TIER
LAYERED
DATABASE-CENTRIC
OOP
CLIENT/SERVER
DATA-DRIVEN
MONOLITHIC
SEPARATION OF CONCERNS
COMPONENTS
MVC
API
SOA
HMVC
EVENT-DRIVEN
ABSTRACTION
SCREAMING ARCHITECTURE
SRP
MVP
MVVM
HEXAGONAL
MICROKERNEL
ONION
DCI
MVA
CLEAN
VIPER
CQRS
DDD
IOC
DIP
EVENT SOURCING
DOMAIN-CENTRIC
MICROSERVICES

RECORDEMOS


Estilo Arquitectónico
Patrón arquitectónico
Patrón de diseño
Patrón de diseño
Patrón arquitectónico
Patrón de diseño
Patrón de diseño

RECORDEMOS

Estilos arquitectónicos	Patrones arquitectónicos
Monolito	Data Transfer Object (DTO)
Cliente Servidor	Data Access Object (DAO)
Peer to Peer (P2P)	Load Balance
Arquiectura en Capas	Service Registry
Microkernel	Service Discovery
SOA	API Gateway
Microservicios	Access Token
Event Driver Architecture (EDA)	Single Sign On
Representate Station Transfer (REST)	Circuit Breaker

CONOCIMIENTOS PREVIOS

Design Smell

Dentro del desarrollo de software, son una manera de definir los síntomas que surgen con el avance del proyecto y crecimiento del software que influyen en la degradación del diseño.

CARACTERISTICAS

Síntoma	Definición	Consecuencias
Rigidez	Dificultad de realizar cambios en el software, incluso en tareas sencillas	• Las estimaciones son más abultadas • Añadir funcionalidades nuevas cuesta mucho, cuando antes era más rápido
Fragilidad	Tendencias a que el software se rompa en múltiples sitios cada vez que se realizar un cambio en el, incluso sin tener relación al cambio con el que se rompe	• Subir una nueva versión y que se rompan partes que aparentemente no tiene que ver con lo modificado • Aparición de mayor cantidad de bugs, aumentando el coste de esfuerzo y tiempo de corregirlos
Inmovilidad	Incapacidad de reutilizar componentes de código por el gran acoplamiento que tiene, haciendo que este sea inamovible cuan la funcionalidad es prácticamente la misma que la deseada	• Aumenta la complejidad del diseño y del código al introducir código duplicado • Complica el entendimiento del código y genera equivocaciones
Viscosidad	La tendencia de que en el ámbito del software, realizar la solución buena requiere mayor esfuerzo comparado con la solución mala y rápida, y en el ámbito del entorno de desarrollo, este es lento e ineficiente	• Provoca el efecto bola de nieve: cuanto más tarde se corrija esa deuda técnica mas difícil y costos será resolverlo • Perdidiza de tiempo, estrés, confusión, pasotismo, acciones inseguras, etc.

Principios de Diseño Software

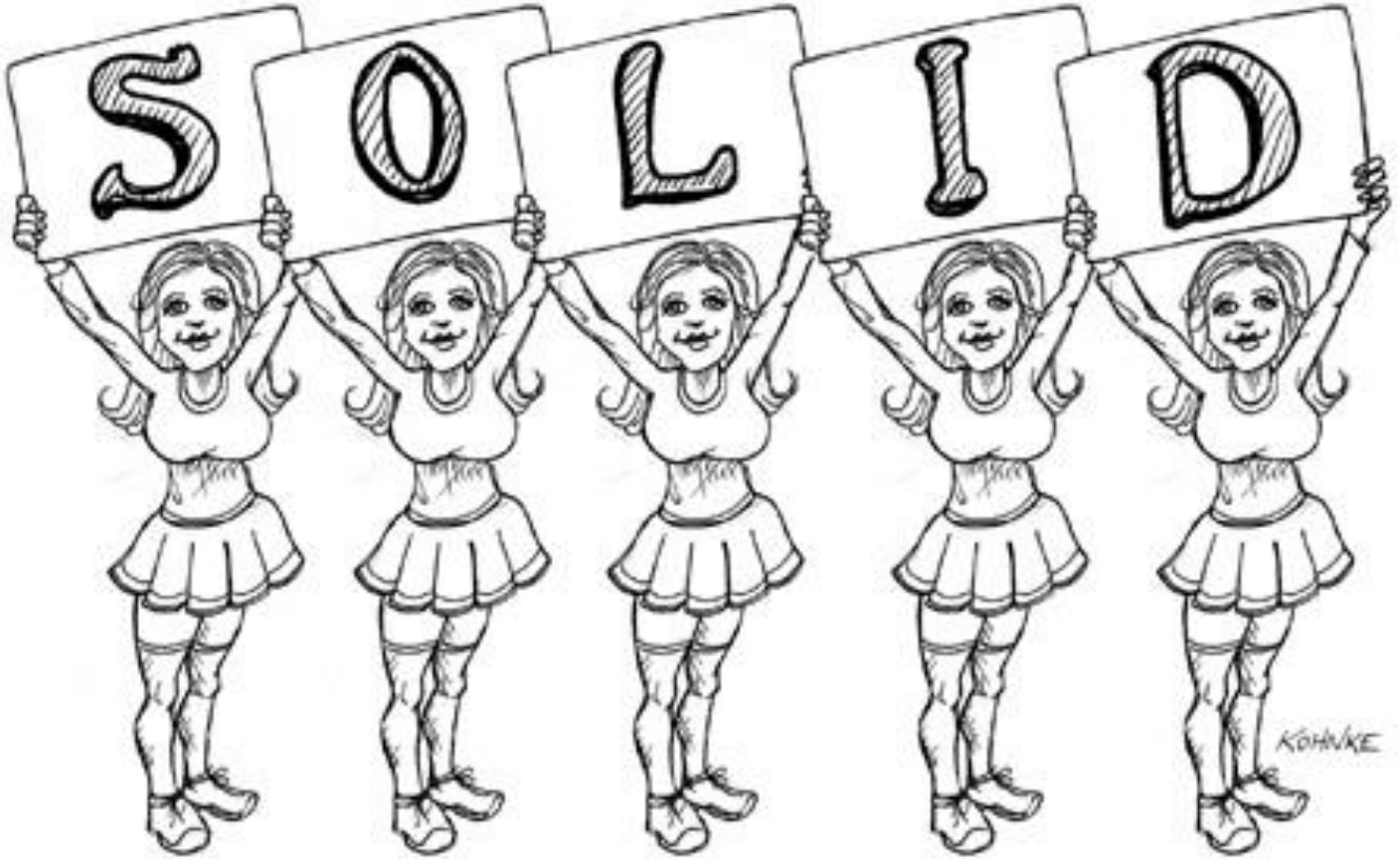
Los principios de desarrollo de software son una serie de **reglas** y **recomendaciones** específicas que los programadores deben seguir durante el desarrollo si quieren escribir un código limpio, comprensible y fácil de mantener.

No hay magia por medio de la cual se pueda transformar una combinación de clases, variables y métodos/funciones, en el código ideal, pero hay algunos consejos y sugerencias que pueden ayudar al programador a determinar si está haciendo las cosas bien y tratar de evitar las situaciones que a modo de ejemplo hemos narrado en el apartado anterior, y que seguro, que si llevamos el tiempo suficiente dedicados a esta maravillosa profesión del desarrollo de software, habremos vivido, sino igual, al menos de manera similar

Principios de Diseño Software

- S.O.L.I.D. (SRP, OCP, LSP, ISP, DIP)
- Don’t Repeat Yourselft (DRY)
- Inversion of Control (IoC)
- You Aren’t Gonna Need It (YAGNI)
- Keep It Simple, Stupid (KISS)
- Law of Demeter
- Strive for loosely coupled design between objects that interact
- Composition over inheritance
- Encapsulate what varies
- The four of simple design
- The boy scout rule
- Las Responsible Moment

Principio SOLID



1. Principio S.O.L.I.S.

Es la unión de 5 principios considerados más importantes de todos, este acrónimo fue introducido por primera vez a principios de la década del 2000 por Robert C. Martin (mejor conocido por muchos como “el tío Bob”), aunque se aplican a cualquier diseño orientado a objetos, los principios SOLID también pueden formar una filosofía central para metodologías como el desarrollo ágil o el desarrollo de software adaptativo. El acrónimo mnemónico SOLID hace referencia a:

- **S**: Single responsibility principle (SRP)
- **O**: Open/closed principle (OCP)
- **L**: Liskov substitution principle (LSP)
- **I**: Interface segregation principle (ISP)
- **D**: Dependency inversion principle (DIP)

1. Principio S.O.L.I.D.

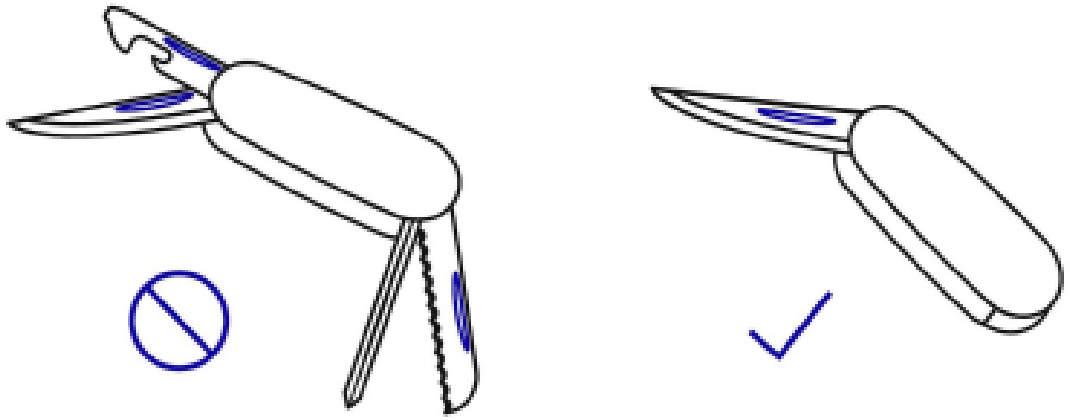
Aplicar estos principios facilitará mucho el trabajo, tanto propio como ajeno (es muy probable que nuestro código lo acaben leyendo muchos otros desarrolladores a lo largo de su ciclo de vida). Algunas de las ventajas de aplicarlo son:

- Facilitar el mantenimiento del código.
- Reducir la complejidad de añadir nuevas funcionalidades.
- Aumentar la reusabilidad de piezas y componentes.
- Mejorar la calidad del código y su comprensión.

1.1. Single Responsibility Principle (SRP)

El principio de **responsabilidad única o single responsibility** establece que un módulo de software debe tener una y solo una razón para cambiar . Esta razón para cambiar es lo que se entiende por responsabilidad.

Este principio está estrechamente relacionado con los conceptos de acoplamiento y cohesión. Queremos aumentar la cohesión entre las cosas que cambian por las mismas razones y disminuir el acoplamiento entre las cosas que cambian por diferentes razones. Este principio trata sobre limitar el impacto de un cambio.



"Reúna las cosas que cambian por las mismas razones. Separe las cosas que cambian por diferentes razones."

1.1. Single responsibility (SRP)

SOLID - Single Responsibility Principle

Una clase debe tener solo una razón para cambiar

El **Single Responsibility Principle (SRP)** o principio de responsabilidad única es un principio que defiende que una clase o módulo debería tener responsabilidad sobre una única funcionalidad del software.

CONCEPTO

Robert C. Martin define el principio con la siguiente frase: ***"Una clase debe tener solo una razón para cambiar."***

¿Pero qué entendemos como **"razón"**?

Robert aclara que el principio trata realmente sobre las **personas**. No queremos que una misma pieza de código tenga que cambiar debido a personas distintas ya que las personas son las que dirigen los cambios en el software.

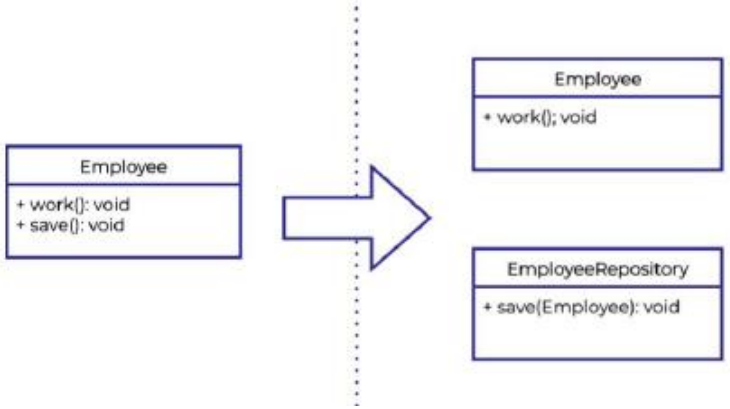
Por ejemplo, un cambio en *Employee* sobre su forma de trabajar vendrá requerido por una persona distinta del que vendrá un cambio sobre la tecnología de persistencia de datos.

Esto podría entenderse como **razones** distintas para cambiar.

EJEMPLO

Podemos identificar que la clase Employee tiene dos responsabilidades, la propia de un empleado que es trabajar y la de persistir su estado.

Podemos separar ambas responsabilidades en dos clases:



1.2. Open / closed principle (OCP)

Este principio nos dice que los módulos de software deben ser abiertos para su extensión pero cerrados para su modificación. ¿A qué se refiere con esto?

- **Abierto para la extensión**: esto significa que el comportamiento del módulo puede extenderse. A medida que cambian los requisitos de la aplicación, podemos ampliar el módulo con nuevos comportamientos que satisfagan esos cambios.
- **Cerrado por modificación**: un módulo estará cerrado si dispone de un contrato (interface) estable y bien definida. Extender el comportamiento de un módulo no debería afectar al código ya existente en el módulo, es decir, el código original del módulo no debería verse afectado y tener que modificarse

OCP fue acuñado en 1988 por Bertrand Meyer:

> "Un artefacto de software debe estar abierto para extensión pero cerrado para modificación".

En otras palabras, el comportamiento de un artefacto de software debería ser extensible, sin tener que modificar ese artefacto.

1.2. Open / closed principle (OCP)

SOLID - Principio Open/Closed

Abierto a la extensión, cerrado a la modificación

Representa la O de SOLID. Con este principio se pretende minimizar el efecto cascada que puede suceder cuando cambiamos el comportamiento de una clase. Si existen clientes que dependan de ella, es posible que tengan que cambiar su comportamiento también.

PLANTEAMIENTO

El principio consta de dos partes:

- Las clases deben estar **abiertas** a la extensión. La extensión se refiere a las modificaciones que pueden ocurrir cuando se plantean nuevos requisitos de software.
- Las clases deben estar **cerradas** a la modificación. No se puede cambiar el código fuente de una clase.

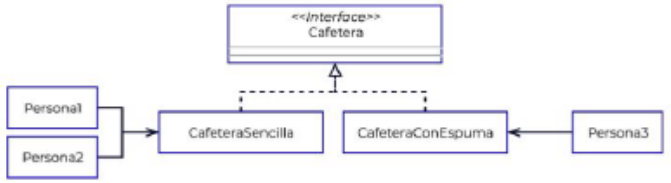
El polimorfismo es la herramienta principal para unir estos dos conceptos que a primera vista parecen contradecirse.

Este principio nos proporciona una mayor estabilidad de los clientes que dependan de la antigua clase. La antigua clase ya funciona, está probado y demostrado. Si incorporamos nuevas funcionalidades a ella, aumentamos la posibilidad de introducir bugs en el software. Por ello, cada vez que se quiera añadir o modificar un comportamiento, en vez de cambiar el código existente, se extiende la clase abstracta o la interfaz y se realizan los cambios en una nueva clase.

EJEMPLO

En una oficina se encuentra una cafetera utilizada por varias personas. Un día, alguien pidió que el café se pudiera preparar distinto. Esta solicitud implicaba un cambio en la configuración y comportamiento de la cafetera.

En vez de cambiar el comportamiento de la cafetera actual para intentar satisfacer a dos clientes distintos, aplicaremos el principio Open/Closed. Cerramos nuestra clase CafeteraSencilla ante cualquier modificación, pero creamos una interfaz para abrirla a extensiones:



Logramos cumplir con el nuevo requisito sin tener que arriesgar los anteriores. Es más, ahora ofrecemos la posibilidad de utilizar una cafetera o la otra, en caso de que cambien de opinión.

1.3. Liskov substitution (LSP)

Este principio dice lo siguiente: Sea Φ (x) una propiedad demostrable sobre los objetos x de tipo T. Entonces Φ (y) debe ser cierta para los objetos y de tipo S donde S es un subtipo de T.

Definición matemática que dio Barbara Liskov sobre este principio.

"Si se ve como un pato, hace cuac como un pato, pero necesita baterías, probablemente tengas la abstracción incorrecta."

1.3. Liskov substitution (LSP)

La sustitución de Liskov nos dice que los objetos de un programa deberían ser reemplazables por instancias de sus subtipos sin alterar el correcto funcionamiento del programa. Básicamente, si en alguna parte de nuestro código estamos usando una clase, y esta clase es extendida, tenemos que poder utilizar cualquiera de las clases hijas y que el programa siga siendo válido. Esto nos obliga a asegurarnos de que cuando extendemos una clase no estamos alterando el comportamiento de la clase padre.

Este principio nos ayuda a utilizar la herencia de forma correcta y nos muestra que no se debe mapear automáticamente el mundo real en un modelo orientado a objetos, ya que no existe una equivalencia unívoca entre ambos modelos.

"Si se ve como un pato, hace cuac como un pato, pero necesita baterías, probablemente tengas la abstracción incorrecta."

1.3. Liskov substitution (LSP)

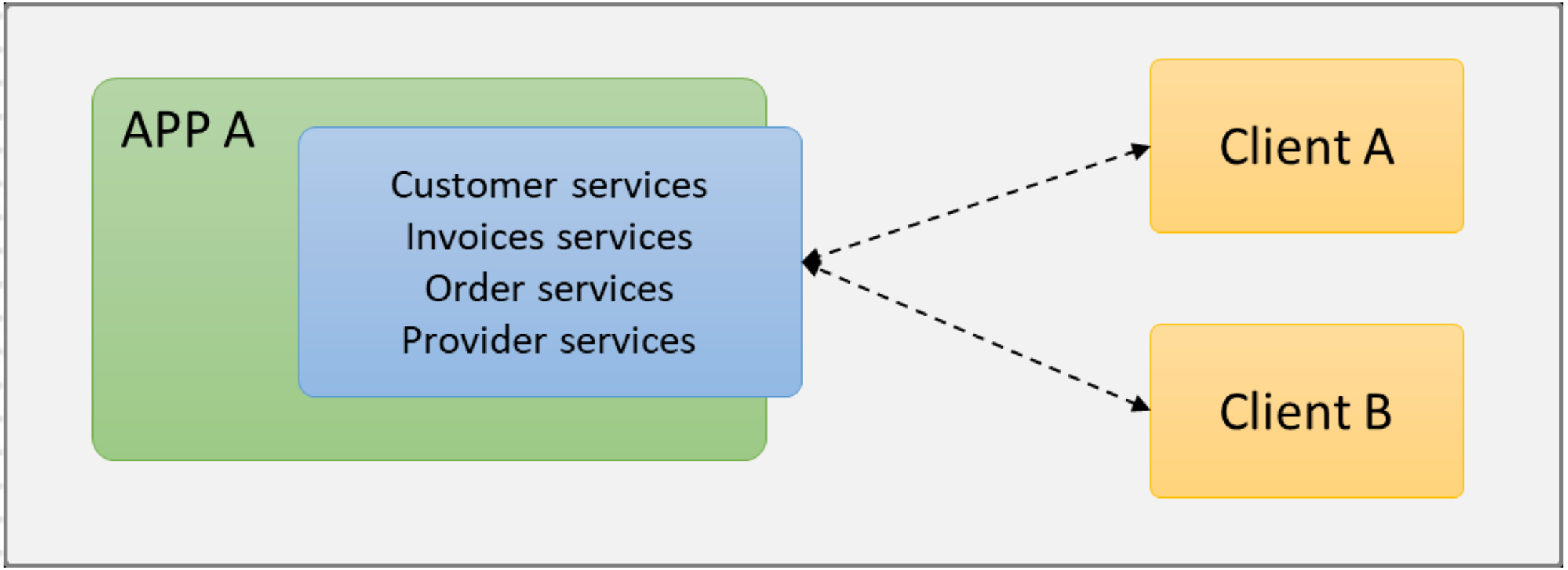
1.4. Interface segregation (ISP)

El Principio de Segregación de la Interfaz indica que el cliente de un programa solo debería de conocer del otro solo los métodos que realmente va a utilizar, evitando conocer una gran cantidad de método que no son necesarios.

La idea que propone es separar los métodos de una interfaz con una gran cantidad de métodos, en varias interfaces más pequeñas y cohesivas, de tal forma que cada interface tenga un rol especifico, permitiendo al cliente ignorar al resto de interfaces y centrarse únicamente en la que necesita, reduciendo el número de método y la complejidad de la misma. A este tipo de interface reducidas se les conoce como “Interfaces de rol”.

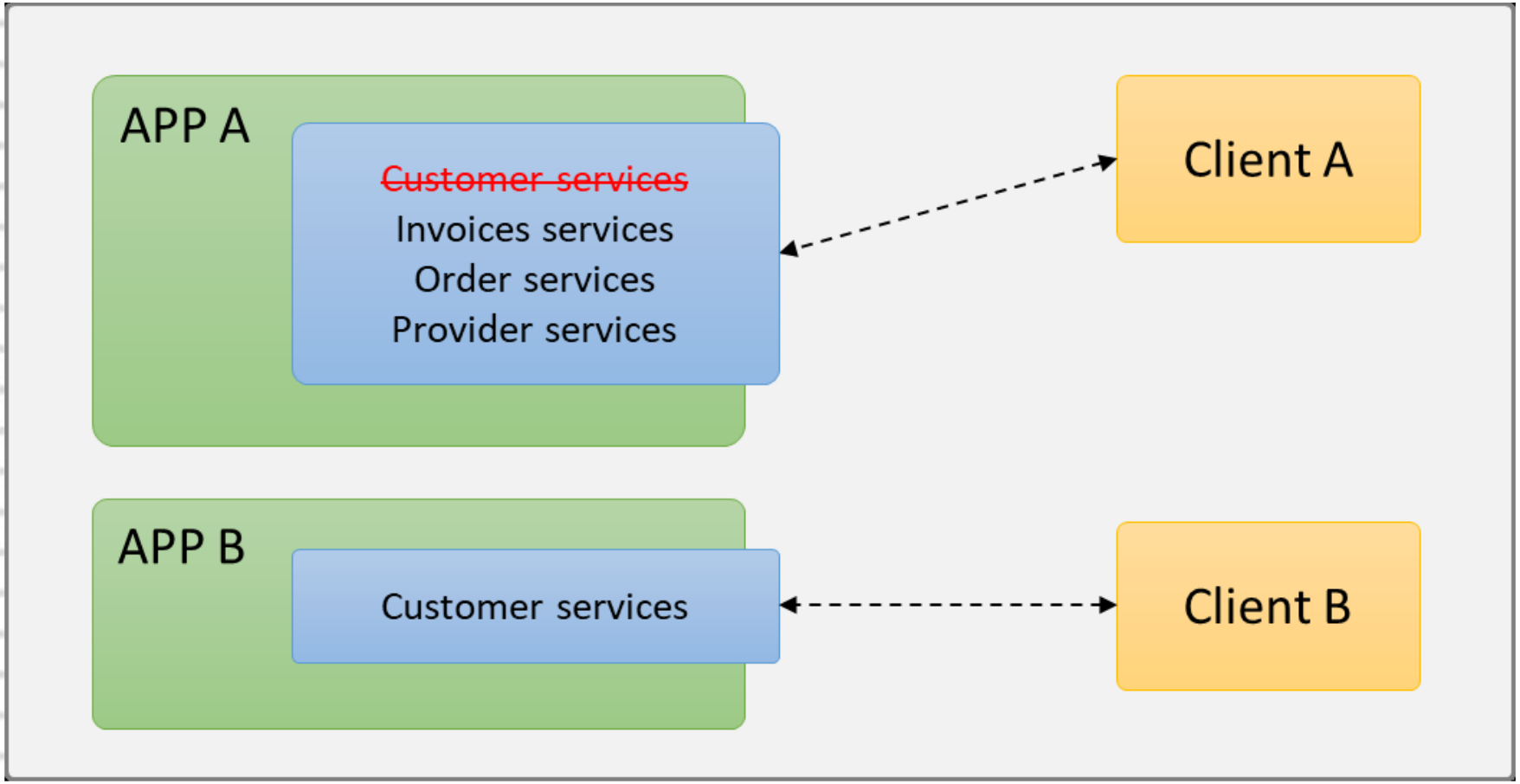
El objetivo que se busca con este principio es mantener desacoplado a los sistemas con respecto a los sistemas de los que depende. A mayor número de métodos mayor será la dependencia, lo que hará más difícil refactorizarlo, modificarlo y redesplegarlo.

1.4. Interface segregation (ISP)



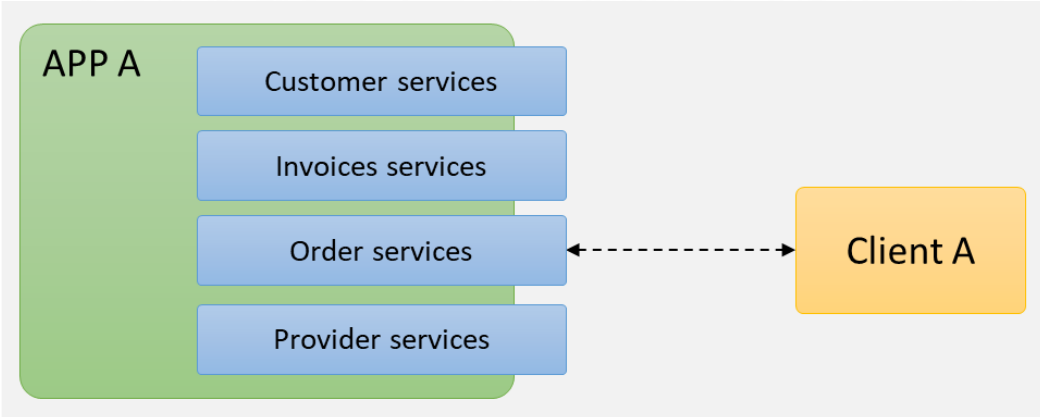
Aglomeración de métodos

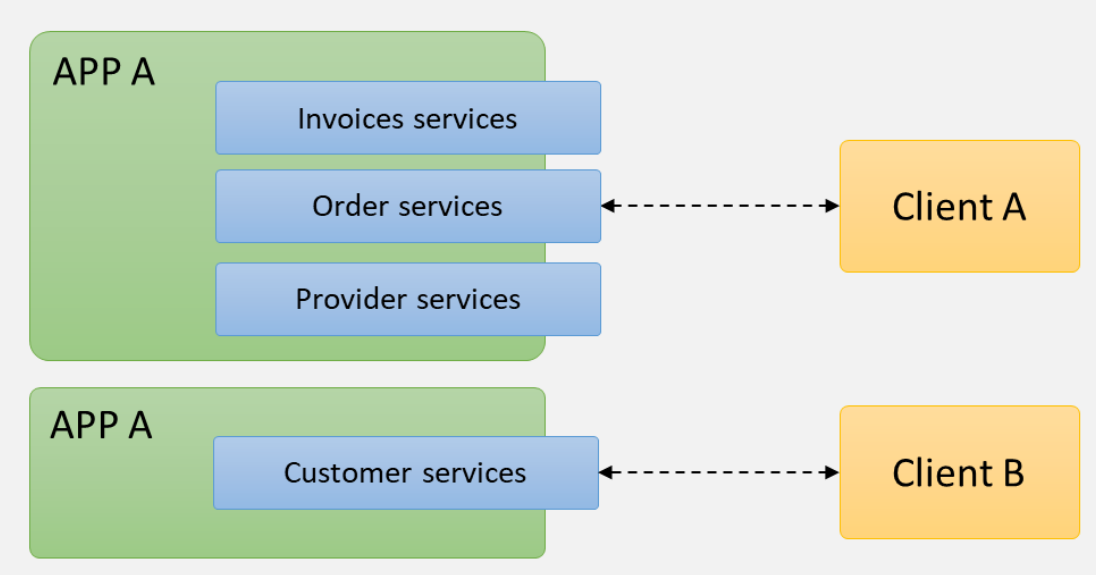
1.4. Interface segregation (ISP)



Separación de responsabilidades

1.4. Interface segregation (ISP)





1.4. Interface segregation (ISP)

SOLID - Interface Segregation Principle

¿Cómo lo aplico?

El **interface Segregation Principle (ISP)** o principio de segregación de interfaces defiende que ninguna clase debería depender de métodos que no usa.

CONCEPTO

Cuando creamos una interfaz debemos estar seguros de que la clase que va a implementar la interfaz va a poder implementar todos los métodos.

En caso contrario **debemos separar la interfaz en interfaces más pequeñas** con menos métodos.

A veces, anticipar qué métodos va realmente a necesitar las implementaciones no es sencillo. Si hemos aplicado correctamente el principio de responsabilidad única y el principio de sustitución de Liskov podremos tener una buena aproximación.

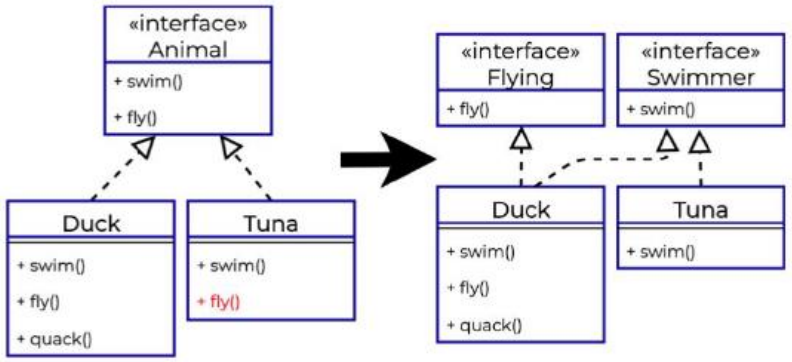
Por lo general siempre será preferible **muchas interfaces pequeñas** a una gran interfaz con muchos métodos.

EJEMPLO

En el ejemplo podemos observar que un Atún se ve forzado a tener un método volar, el cual no puede implementar, ya que un atún no puedo volar.

Podríamos separar ambas habilidades en distintas interfaces. Una para voladores y otra para nadadores.

De este modo los animales implementarán solo las interfaces que necesiten.

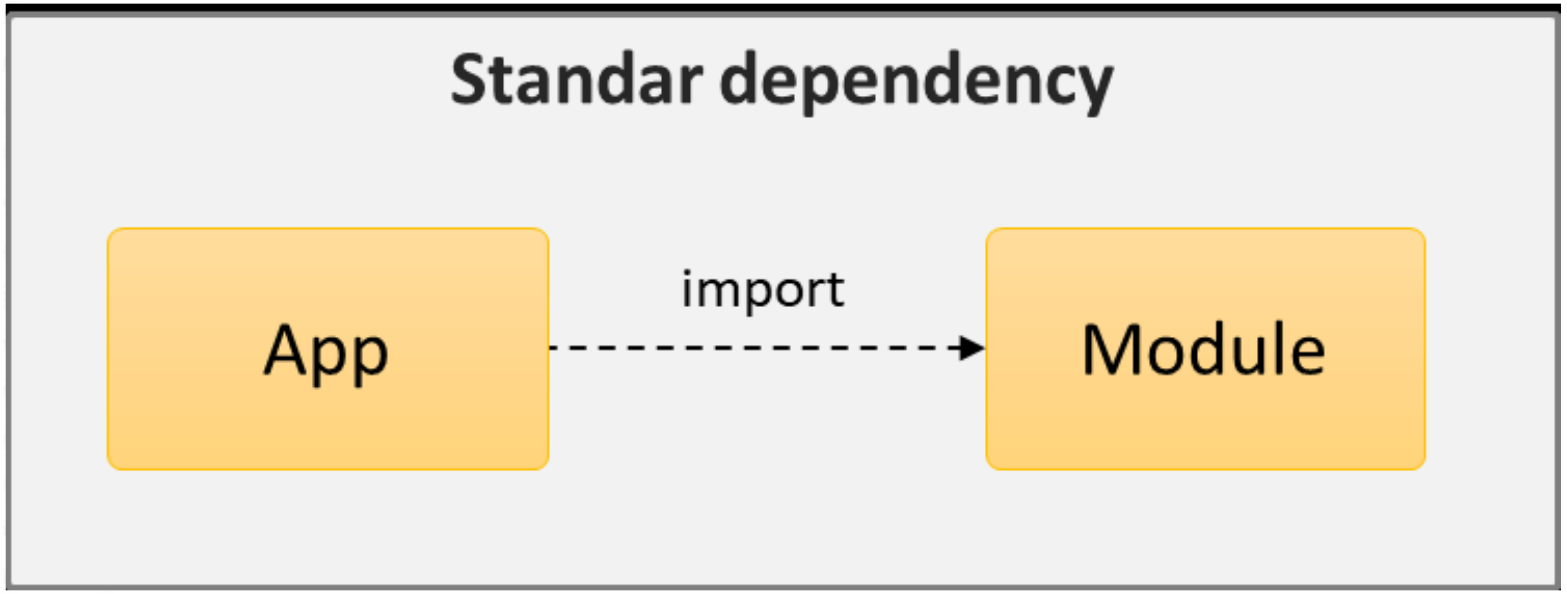


1.5. Dependency inversion principle (DIP)

Normalmente cuando desarrollamos software, solemos hacer que los software de más alto nivel, dependa de los más bajo nivel, de tal forma que para que un módulo completo funcione, requiere que otros más pequeños estén presentes para funcionar, muy parecido a la clásica arquitectura de 3 capas, en la que la capa de presentación depende de la capa de negocio y la capa de negocio depende de la capa de datos, sin embargo, el principio DIP viene a movernos un poco la jugada, pues lo que propone es lo siguiente:

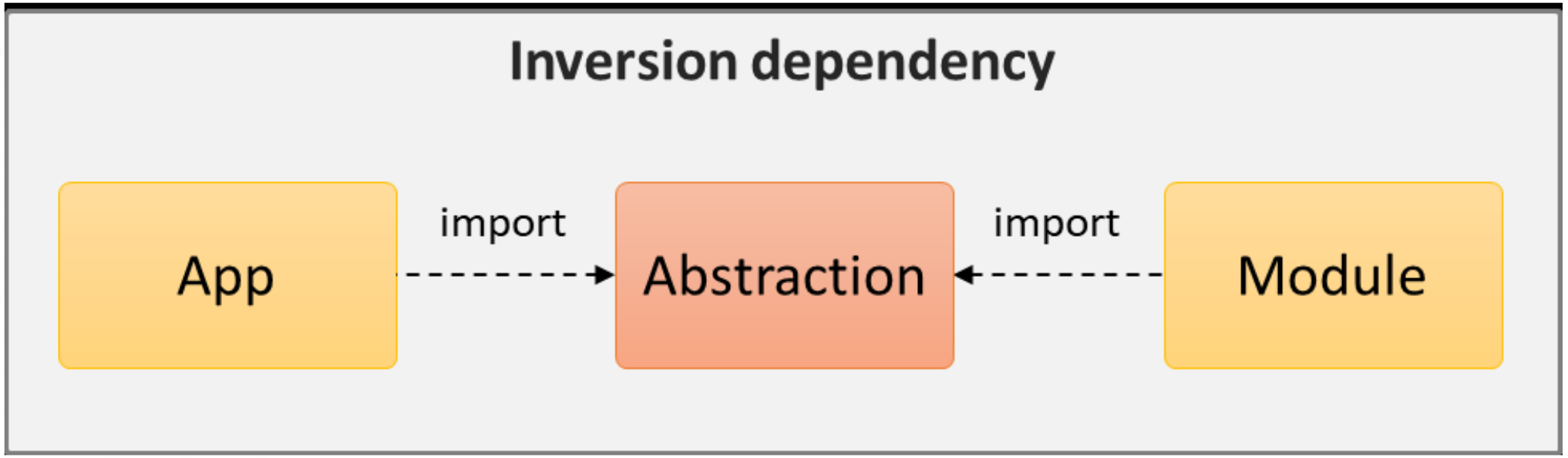
- Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de las abstracciones.
- Las abstracciones no deben depender de los detalles. Los detalles deben depender de las abstracciones.

1.5. Dependency inversion principle (DIP)



Forma habitual de dependencia

1.5. Dependency inversion principle (DIP)



Inversión de dependencias

1.5. Dependency inversion principle (DIP)

La abstracción es un módulo abstracto que solo implementa las interfaces o clases base que luego el módulo de bajo nivel (Module en el diagrama) deberá implementar y posteriormente el módulo de Alto nivel (App en el diagrama) deberá utilizar. Cabe mencionar que el módulo de bajo nivel deberá tener toda la funcionalidad que define la abstracción y siempre alineado a las interfaces o especificaciones de la abstracción, de tal forma que cuando el módulo de Alto necesite utilizar el módulo de bajo nivel lo haga a través de la abstracción, de esta forma, el Módulo de alto nivel no necesita saber la implementación concreta del módulo de bajo nivel, logrando con esto, la variación de cualquier parte sin afectar el funcionamiento, incluso, mediante este patrón, es posible tener múltiples implementaciones del módulo de bajo nivel.

1.5. Dependency inversion principle (DIP)

La abstracción es un módulo abstracto que solo implementa las interfaces o clases base que luego el módulo de bajo nivel (Module en el diagrama) deberá implementar y posteriormente el módulo de Alto nivel (App en el diagrama) deberá utilizar. Cabe mencionar que el módulo de bajo nivel deberá tener toda la funcionalidad que define la abstracción y siempre alineado a las interfaces o especificaciones de la abstracción, de tal forma que cuando el módulo de Alto necesite utilizar el módulo de bajo nivel lo haga a través de la abstracción, de esta forma, el Módulo de alto nivel no necesita saber la implementación concreta del módulo de bajo nivel, logrando con esto, la variación de cualquier parte sin afectar el funcionamiento, incluso, mediante este patrón, es posible tener múltiples implementaciones del módulo de bajo nivel.

1.5. Dependency inversion principle (DIP)

SOLID - Dependency Inversion

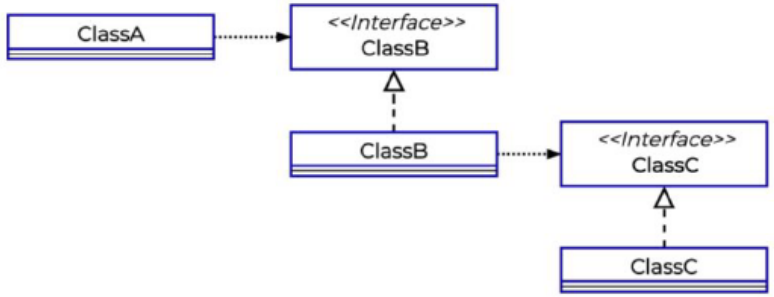
¿Qué es?

Representa la D de los principios S.O.L.I.D. El principio dice, los **módulos de alto nivel no deben depender de módulos de bajo nivel, ambos deben depender de abstracciones**. El principio tiene como fin evitar depender de concreciones para minimizar el grado de acoplamiento entre los componentes.

PROBLEMA - SOLUCIÓN

Normalmente, las capas de alto nivel (ClassA) consumen las de bajo nivel (ClassC), generando un fuerte acoplamiento.

Para evitar este problema, se introduce una capa de abstracción que permite reutilizar las capas de mayor nivel.



VENTAJAS

Si aplicamos correctamente los anteriores principios SOLID como el Open/Closed y el de Liskov, estamos implícitamente cumpliendo con la inversión de dependencias. Pero, ¿qué ventajas nos aporta?

- **Mayor flexibilidad haciendo testing**: gracias a que se depende de abstracciones, podemos reemplazar objetos reales por mocks que simulen el comportamiento deseado.
- **Reduce el acoplamiento** entre clases: al no depender de una implementación en concreto, no acoplamos nuestro código a ciertas clases específicas.
- Código más **limpio, legible y mantenible** en el tiempo: al no depender de implementaciones, estamos contribuyendo a un desarrollo más robusto y que no sea frágil a cualquier cambio.
- Al depender de abstracciones, tenemos la **posibilidad de elegir en tiempo de ejecución la implementación** adecuada.

2. Keep It Simple, Stupid (KISS)

Hay discrepancias sobre si esta frase significa: “Déjalo simple, estúpido” o “Mantenlo estúpidamente simple”. Este principio establece que el código, el diseño, la documentación, todo lo relacionado con el software, debe ser tan simple como sea posible.

Los developers tienden a complicar las cosas. Nos piden un formulario sencillo y queremos hacer un generador de formularios que soporte este y todos los formularios en el futuro. No tenemos ni 100 usuarios y ya queremos usar Kubernetes. Necesitamos un simple binding de datos y queremos meter Angular 16, Ionic 6 y 250 bibliotecas más.

- Nuestro software en general debería tener tan pocos componentes (y por lo tanto líneas) como sea posible.
- No deberíamos tener funcionalidades que no se ocupen actualmente “por si en el futuro se ocupan”.
- La documentación debe tener la información estrictamente necesaria.
- El código debe ser lo más obvio y sencillo posible. Se deben evitar esas líneas que sólo sirven para presumir lo inteligente que eres.
- El diseño debe mantener la estructura simple, siempre que se pueda.

2. Acoplamiento

El acoplamiento es nivel de dependencia que existe entre dos unidades de software, es decir, indica hasta qué grado una unidad de software puede funcionar sin recurrir a la otra.

Entonces podemos decir que el acoplamiento es el nivel de dependencia que una unidad de software tiene con la otra. Por ejemplo, imagina una aplicación de registro de clientes y su base de datos, dicho esto, ¿qué tan dependiente es la aplicación de la base de datos? ¿podría seguir funcionando la aplicación sin la base de datos? Seguramente todos estaremos de acuerdo que sin la base de datos la aplicación no funcionará en absoluto, pues es allí donde se guarda y consulta la información que vemos en el sistema, entonces podríamos decir que existe un Alto acoplamiento.

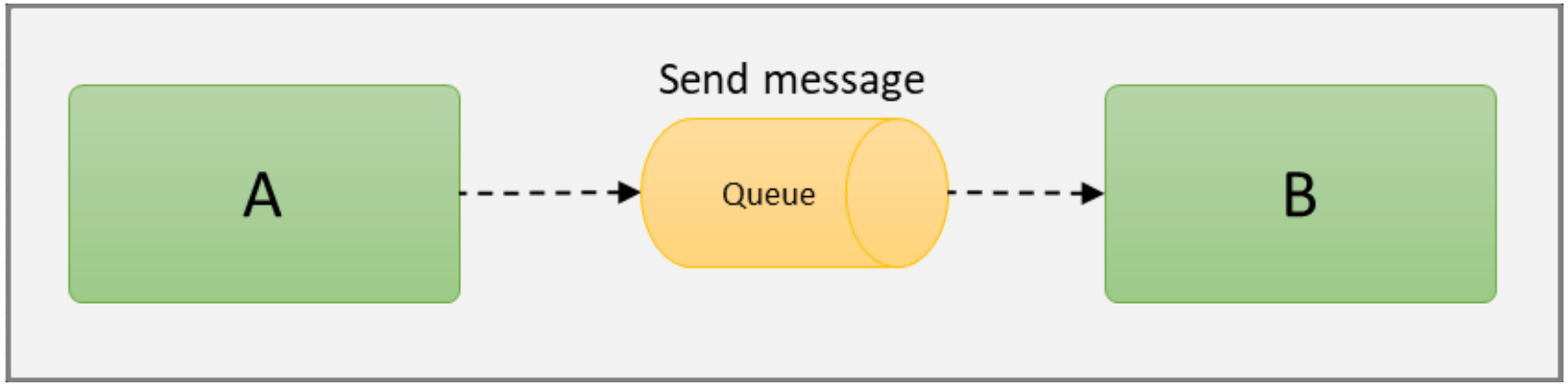
2. Acoplamiento

Cuando un arquitecto inexperto diseña una solución, por lo general crea relaciones de acoplamiento altas entre los módulos de la solución, no porque sea bueno o malo, sino porque es la única forma que conoce y que al mismo tiempo es la más fácil. Por ejemplo, si te pidiera que comunicaras dos sistemas, en el cual, la aplicación A debe mandar un mensaje a B por medio de un Webservices asíncrono, ¿cómo diseñarías la solución? La gran mayoría haría esto:

Alto acoplamiento.

2. Acoplamiento

El bajo acoplamiento se logra cuando un módulo no conoce o conoce muy poco del funcionamiento interno de otros módulos, evitando la fuerte dependencia entre ellos. Como buena práctica siempre debemos buscar el bajo acoplamiento, sin embargo, no siempre será posible y no deberemos forzar una solución para lograrlo:



Bajo acoplamiento.

3. Cohesión

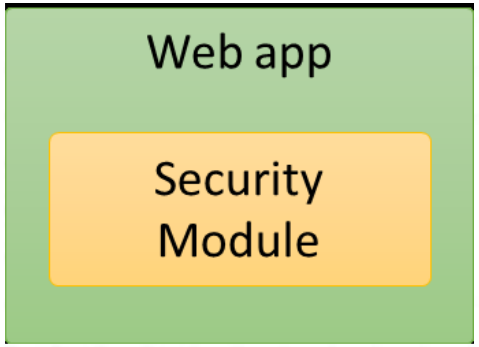
la cohesión es una medida del grado en que los elementos del módulo están relacionados funcionalmente, es decir, se refiere al grado en que los elementos de un módulo permanecen juntos.

En otras palabras, la cohesión mide que tan relacionados están las unidades de software entre sí, buscando que todas las unidades de software busquen cumplir un único objetivo o funcionalidad.

En la práctica, se busca que todos los compontes de software que construyamos sean altamente cohesivos, es decir, que el módulo esté diseñado para resolver una única problemática.

3. Cohesión

Imaginemos que estamos por construir una nueva aplicación web para un cliente; esta aplicación además de cumplir con todos los requerimientos de negocio, requiere de un módulo de seguridad en donde podamos administrar a los usuarios..

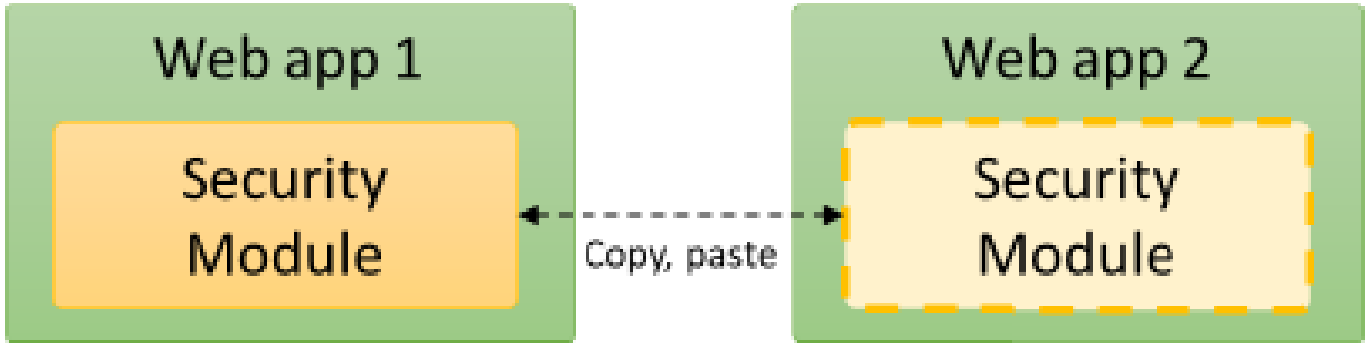


Baja cohesión

Para muchos, implementar el módulo de seguridad dentro de la aplicación es lo más normal, pues al final, necesitamos asegurar la aplicación, sin embargo, puede que en principio esto no tenga ningún problema, pero estamos mesclando la responsabilidad de negocio que busca resolver la aplicación, con la administración de los accesos, lo que hace que además de preocuparse por resolver una problemática de negocio, también nos tenemos que preocupar por administrar los accesos y privilegios.

3. Cohesión

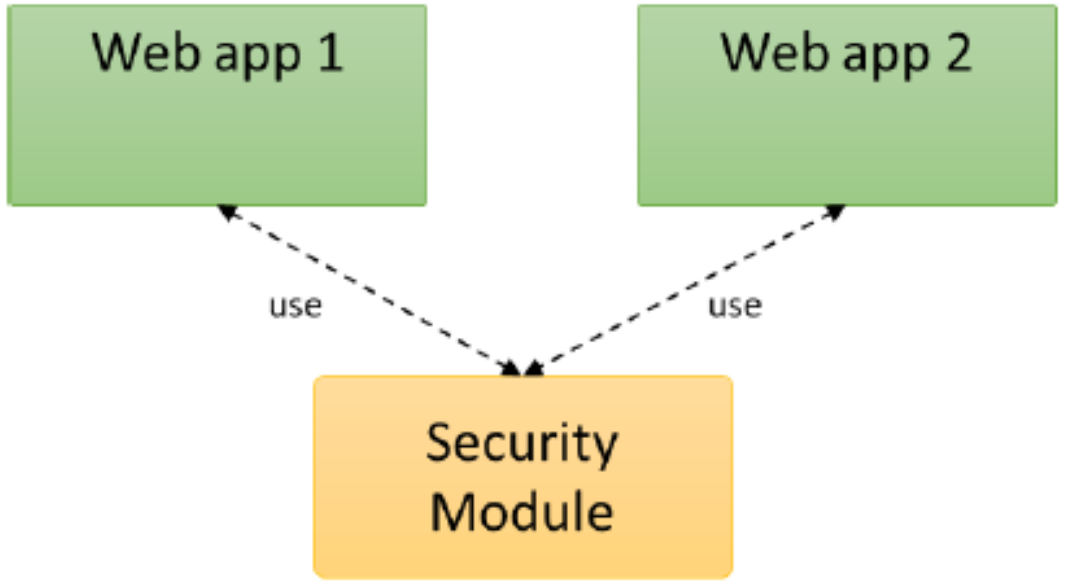
Si esta es la única aplicación que requiere autenticación, entonces podríamos decir que está bien, sin embargo, las empresas por lo general tienen más de una aplicación funcionando y que además requiere de autenticación. Entonces, ¿qué pasaría si el día de mañana nos piden otra aplicación que requiere autenticación? Lo más seguro es que terminemos copiando el código de esta aplicación y pegándolo en la nueva, lo que comprueba que la aplicación es bajamente cohesiva, pues buscó resolver más de una problemática..



Copy-Paste code

3. Cohesión

Ahora bien, que pasaría si nos damos cuenta que el módulo de seguridad es una problemática distinta a la que busca resolver la aplicación y que, además, podría ser utilizada por otras aplicaciones:



Alta cohesión

4. Keep It Simple, Stupid (KISS)

Este no es un principio que tenga una serie de pasos o acciones a seguir, sino más bien nos invita a reflexionar si lo que estamos haciendo es realmente necesario y si no estamos complicando las cosas más de lo que deberíamos.

Ahora bien, hacer las cosas simples, no es fácil, pues hacerlo simple no significa que dejemos de hacer cosas o que quitemos funcionalidad que el usuario requiera, sino más bien, idear como hacer que todo sea lo más fácil e intuitivo para el usuario, como reducir el número de pasos para terminar una transacción, reducir el número de campo solicitados, reducir el número de actores involucrados en un proceso, reducir el número de trámites, etc.

Dos de los ejemplos más exitoso de KISS son Google y Apple. Google invento un buscador súper minimalista, al entrar solo aparecía una barra de búsqueda y el logotipo, y eso era todo, los usuarios podían encontrar lo que buscaban si saber absolutamente nada de buscadores.

Por su parte Steve Jobs invento un teléfono súper intuitivo, con teclado integrado, multitouch y navegador integrado, lo que nos demuestra que hacer las cosas simples no es fácil, si no requiere conocer a tus usuarios para ofrecerles lo que buscan lo más simple posible.

Trabajo en clase

¿Que aprendimos en esta session?

SOLID

1) .....
2) .....
3) .....

Acoplamiento y Cohesion

1) .....
2) .....

Universidad
Tecnológica
del Perú