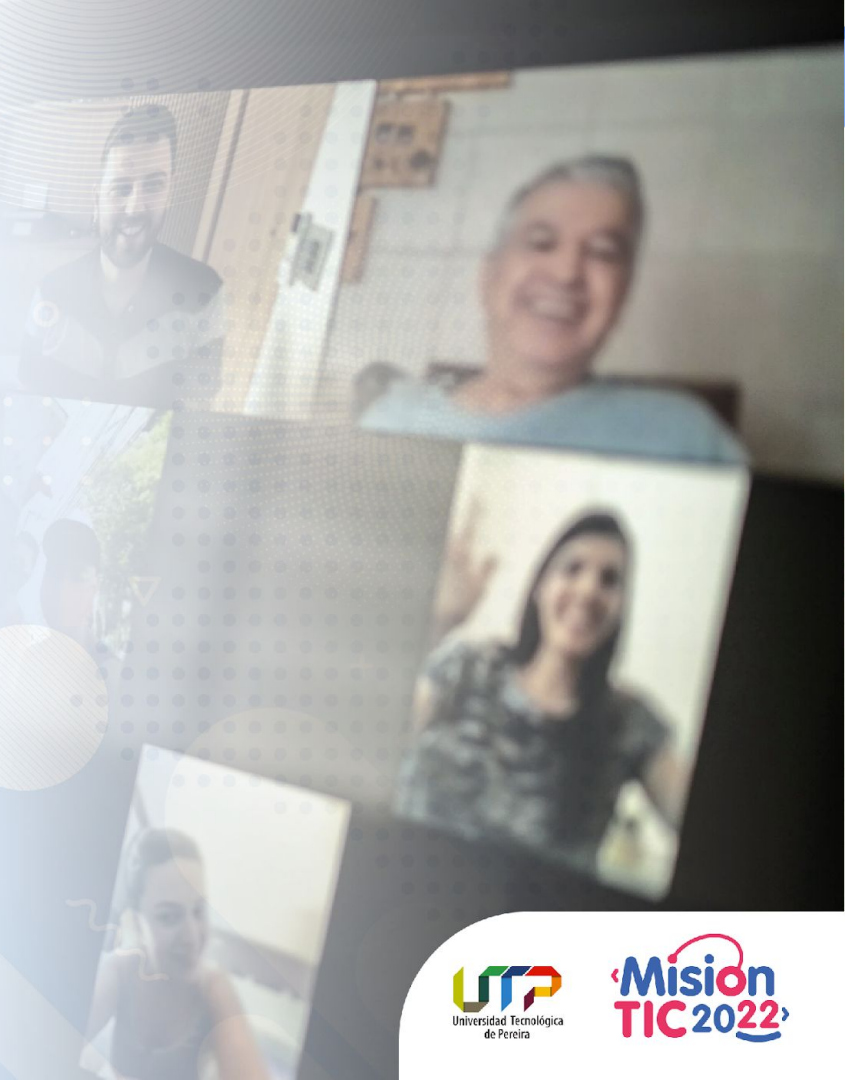
El futuro digital es de todos

MinTIC

Unidad 6

17 - Patrones de Arquitectura Principios POO



Universidad Tecnológica de Pereira

MisionTIC2022

El futuro digital es de todos

MinTIC

Patrones de Arquitectura

Universidad Tecnológica de Pereira

MisiónTIC 2022

¿Que es patrón de arquitectura?

Según Wikipedia:

- Los **patrones arquitectónicos**, o **patrones de arquitectura**, ofrecen soluciones a problemas de arquitectura de software en ingeniería de software.
- Dan una descripción de los elementos y el tipo de relación que tienen junto con un conjunto de restricciones sobre cómo pueden ser usados.
- Un patrón arquitectónico expresa un esquema de organización estructural esencial para un sistema de software, que consta de subsistemas, sus responsabilidades e interrelaciones.

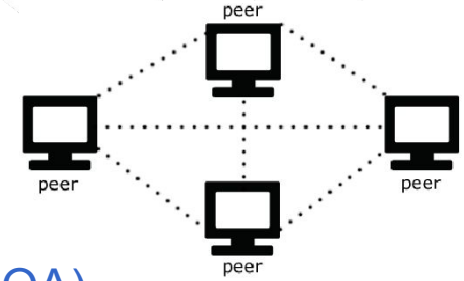
https://es.wikipedia.org/wiki/Patrones_de_arquitectura

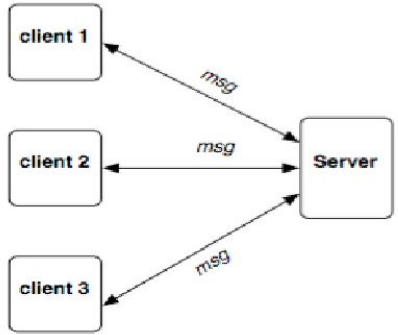
¿Que es patrón de arquitectura?

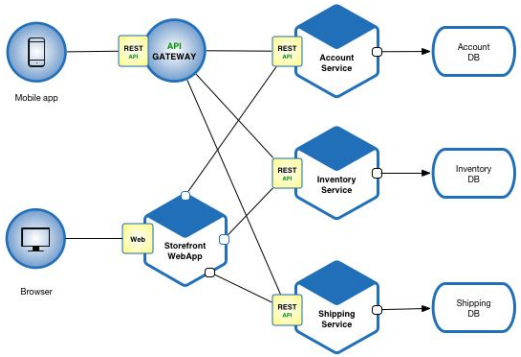
- Uno de los aspectos más importantes de los patrones arquitectónicos es que encarnan diferentes **atributos de calidad**. Por ejemplo, algunos patrones representan soluciones a problemas de rendimiento y otros pueden ser utilizados con éxito en sistemas de alta disponibilidad.
- A primeros de la **fase de diseño**, un arquitecto de software escoge qué patrones arquitectónicos mejor ofrecen las calidades deseadas para el sistema.

Ejemplos de patrones arquitectónicos

- **Multi-capas**
- Cliente / Servidor
- Arquitectura en pizarra
- Arquitectura dirigida por eventos
- Peer-to-peer (P2P)
- Arquitectura orientada a servicios (SOA)
- Microservicios
- **Puertos y Adaptadores (Hexagonal)**
- **Modelo Vista Controlador (MVC)**
- **Modelo – Vista – Modelo de Vista (MVVM)**

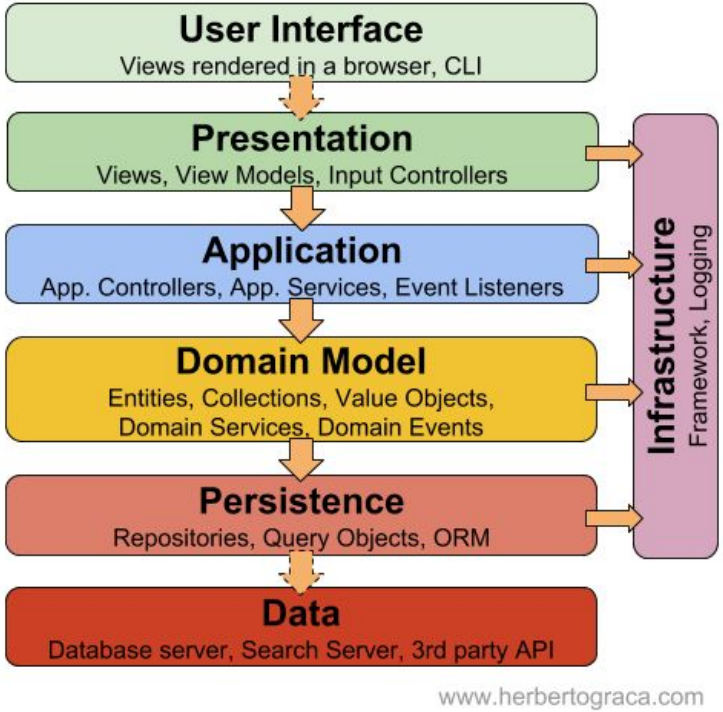






Patrón Multi-capa (N-capas)

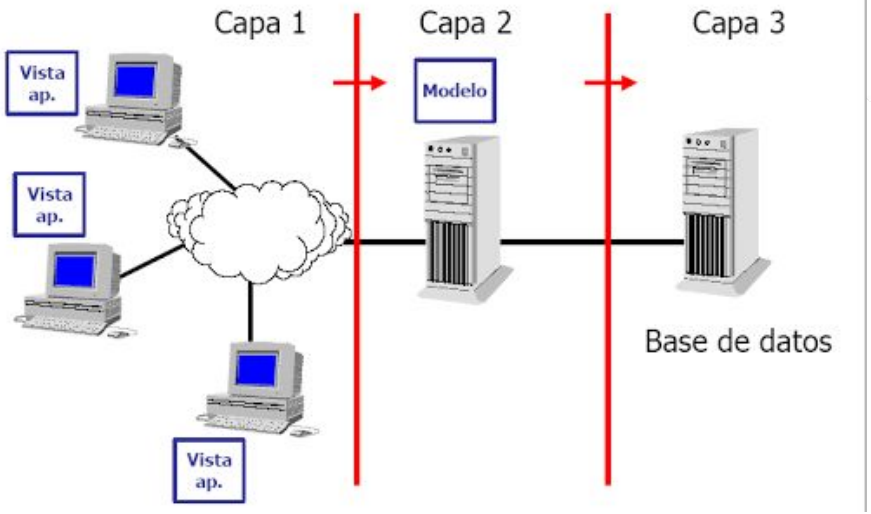
- Se enfoca en la distribución de roles y responsabilidades de forma jerárquica proveyendo una forma muy efectiva de separación de responsabilidades.
- El rol indica el modo y tipo de interacción con otras capas, y la responsabilidad indica la funcionalidad que está siendo desarrollada.
- Este estilo ha evolucionado desde la aproximación basada en componentes generalmente usando métodos específicos de comunicación asociados a una plataforma en vez de la aproximación basada en mensajes.

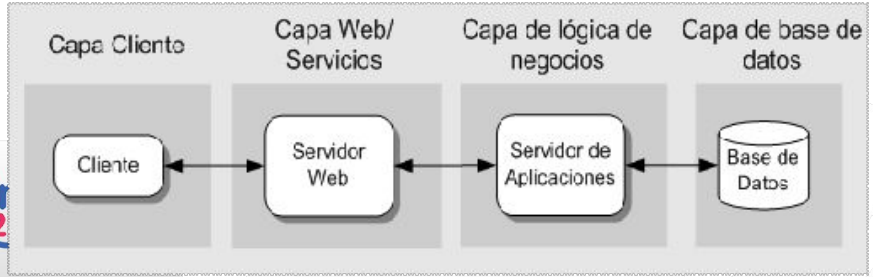


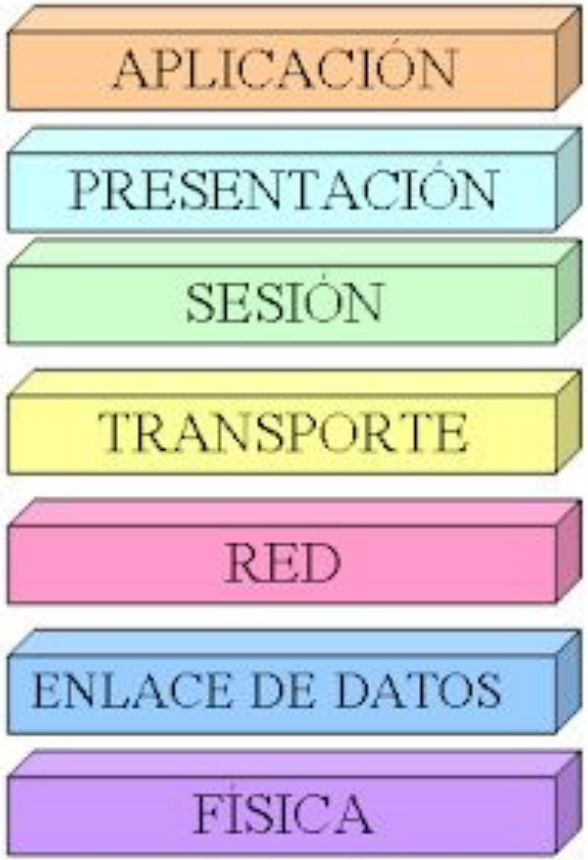

SUÉÑALO
CRÉELO
&
HÁCELO
Capa N + 1
Capa N
Capa N - 1
Capa 1
Serie de patrones de arquitectura
ARQUITECTURA POR CAPAS

Multi-capa

Ejemplos Multi-capa







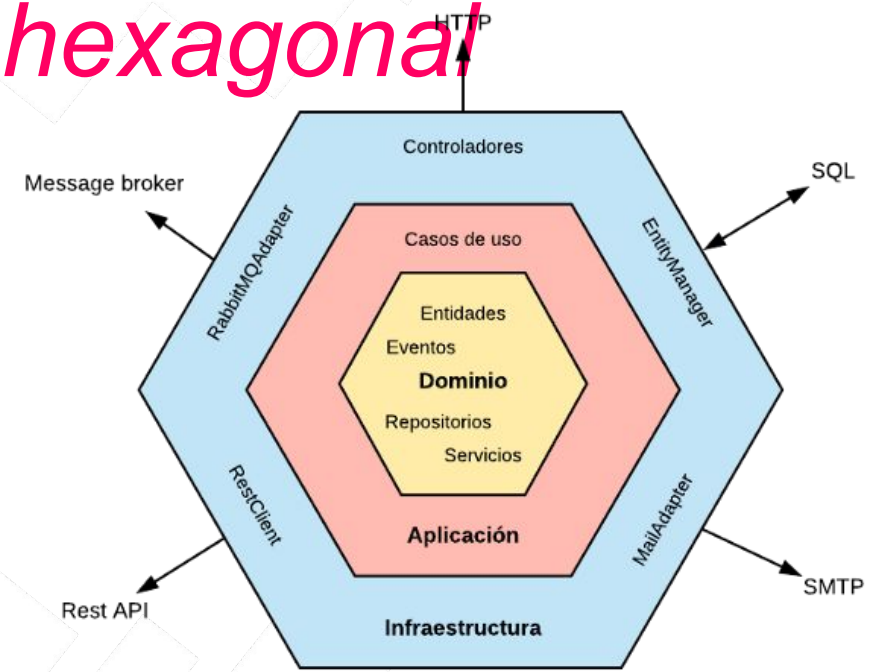
Patrón puertos y adaptadores
Arquitectura hexagonal

- La intención no es otra que permitir que una aplicación sea usada de la misma forma por usuarios, programas, pruebas automatizadas o scripts, y sea desarrollada y probada de forma aislada de sus eventuales dispositivos y bases de datos en tiempo de ejecución.
- Una de las grandes pesadillas en las aplicaciones de software ha sido la infiltración de la lógica del negocio en el código de la interfaz de usuario.
  - Dificulta la automatización de pruebas.
  - Impide el cambio de uso de la aplicación.
  - Dificulta o impide el uso por otro programa.
- Otro gran problema es el acoplamiento con detalles de infraestructura como la base de datos.

Patrón puertos y adaptadores

Arquitectura hexagonal

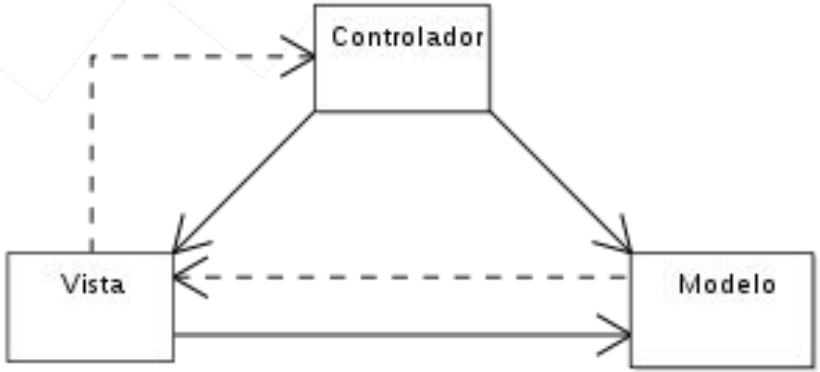
- Propone que nuestro dominio sea el núcleo de las capas y que este no se acople a nada externo.
- En lugar de hacer uso explícito y mediante el principio de inversión de dependencias nos acoplamos a contratos (interfaces o puertos) y no a implementaciones concretas.
- **Puerto**: definición de una interfaz pública.
- **Adaptador**: especialización de un puerto para un contexto concreto.




ARQUITECTURA
HEXAGONAL

Modelo Vista Controlador (MVC)

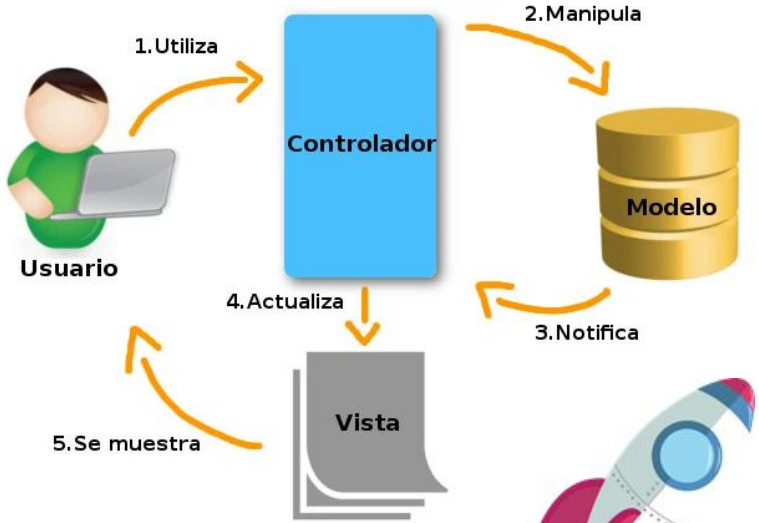
- Es un patrón que separa los datos y principalmente lo que es la lógica de negocio de una aplicación de su representación y el módulo encargado de gestionar los eventos y las comunicaciones.

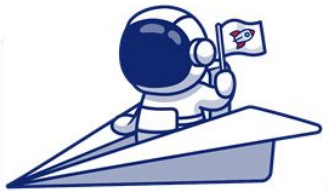


- MVC propone la construcción de tres componentes distintos que son el **modelo**, la **vista** y el **controlador**, es decir, por un lado define componentes para la representación de la información, y por otro lado para la interacción del usuario.

Modelo Vista Controlador (MVC)

- El **Modelo**: Es la representación de la información, por lo tanto gestiona todos los accesos a dicha información, tanto consultas como actualizaciones, implementando también los privilegios de acceso que se hayan descrito en las especificaciones de la aplicación (lógica de negocio).
- El **Controlador**: Responde a eventos (usualmente del usuario) e invoca peticiones al 'modelo' cuando se hace alguna solicitud sobre la información. También puede enviar comandos a su 'vista' asociada si se solicita un cambio en la forma en que se presenta el 'modelo'.
- La **Vista**: Presenta el 'modelo' (información) en un formato adecuado para interactuar usualmente con el usuario.

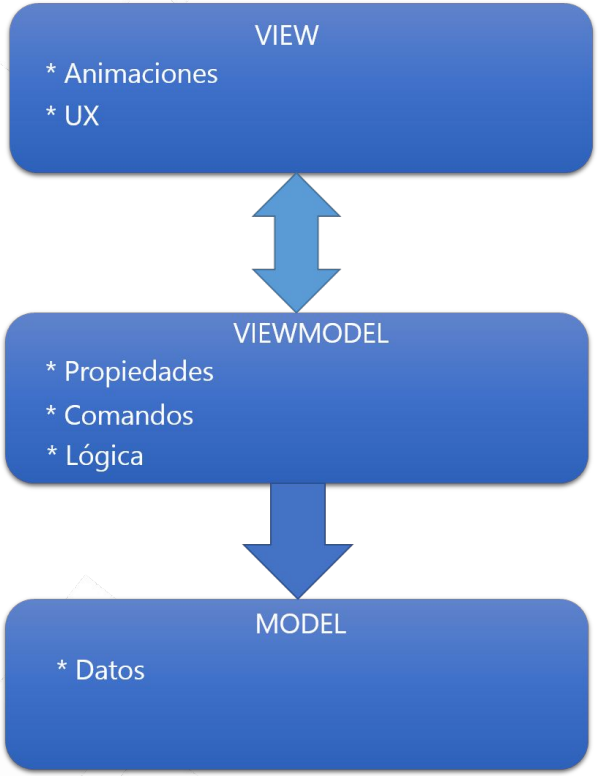





¿QUÉ ES
MVC?
CONTROLADOR
MÓDELO
VISTA

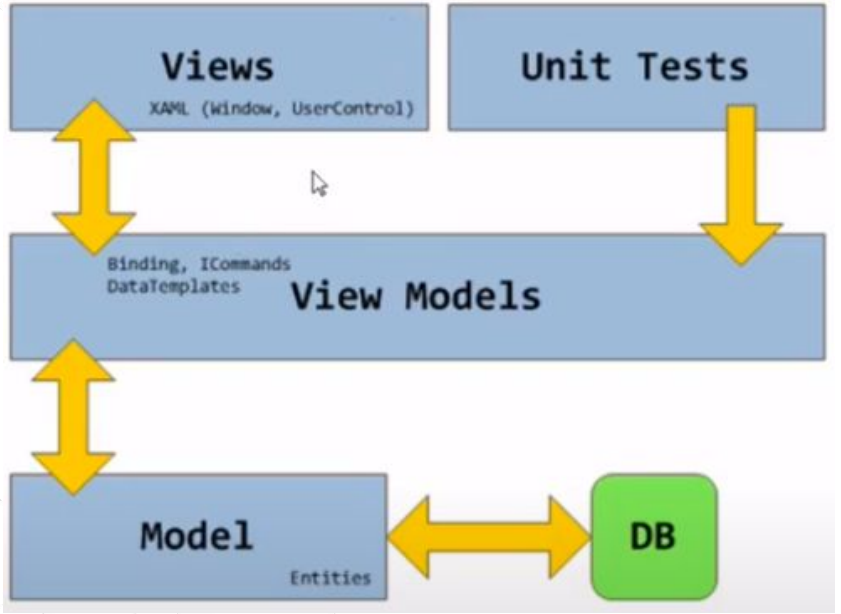
Modelo Vista Vista de Modelo (MVVM)

- En el año 2004, un grupo de desarrollo de Microsoft trabajaba en un proyecto denominado “Avalon”, cuyo propósito era permitir el desarrollo de aplicaciones de escritorio más completas y con un aspecto visual mucho más logrado y complejo de lo que era posible con Windows Forms.
- John Gossman en 2005, en un artículo de la MSDN, mostraba al público el patrón MVVM.
- En dicho artículo, MVVM se presenta como una variación del patrón MVC ajustado a “WPF” y a su sistema de enlace a datos.



Modelo Vista Vista de Modelo (MVVM)

- El **modelo** representa la capa de datos y/o la lógica de negocio. En ningún caso tiene dependencia alguna con la vista.
- La **vista** representa la información a través de los elementos visuales que la componen, son activas y contienen comportamientos, eventos y enlaces a datos que necesitan.
- El **modelo de vista** es un actor intermediario entre el modelo y la vista, contiene toda la lógica de presentación y se comporta como una abstracción de la interfaz. La comunicación entre la vista y el viewmodel se realiza por medio los enlaces de datos (***databinding***).



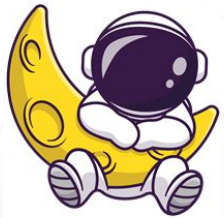

C#
José Manuel Montero
¿QUÉ ES EL MODELO MVVM?

Aplicaciones MVVM

- .NET Framework
  - WPF
  - Xamarin Forms
- Android
  - Jetpack (Java, Kotlin)
- IOS
  - Swift
- Javascript
  - Vue

¿Solo se puede usar una a la vez?

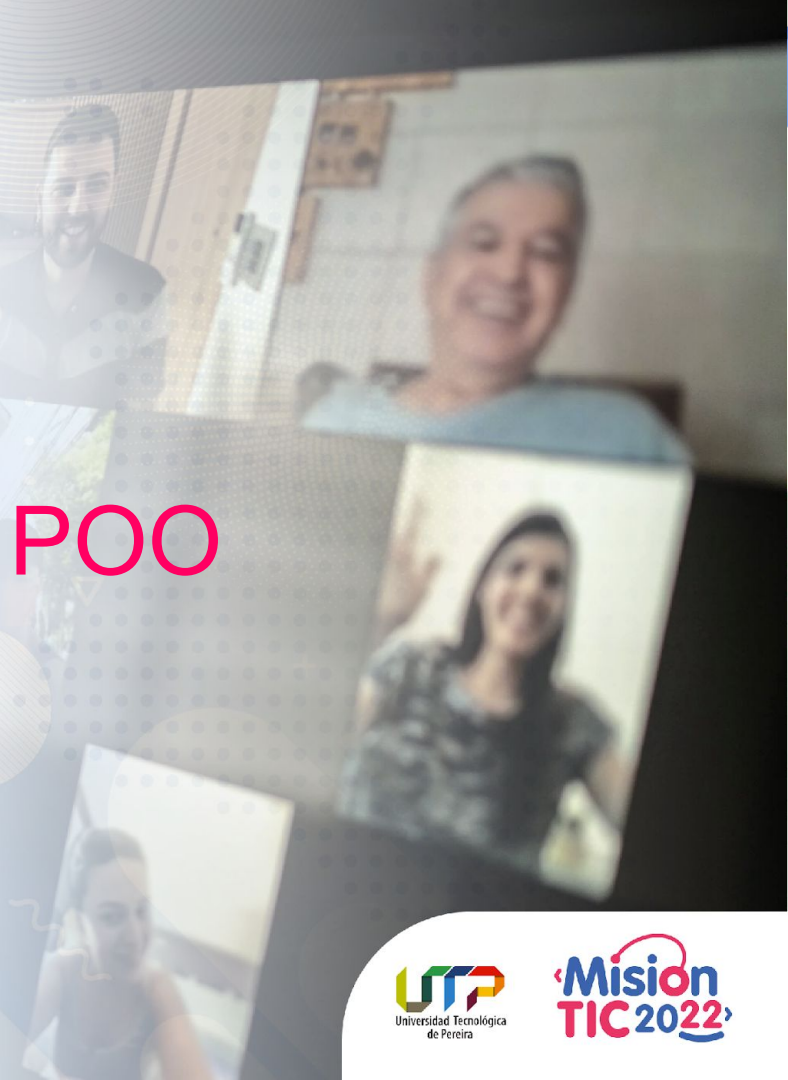
- **Capa Entidades**: se encarga de almacenar la entidades del negocio, es decir, guardar la información que traemos o enviamos a la base de datos. (Paciente, Médico, Especialidad, etc.).
- **Capa Componentes Comunes**: van las clases que sirven de ayuda o que tienen un uso genérico dentro de nuestro proyecto, como por ejemplo: clases para cifrar y descifrar cadenas, clases con métodos de extensión, clases para validar o parsear datos entre otros.
- **Capa Acceso a Datos**: van todos los métodos que sirven para traer o enviar datos a la Base de Datos (Grabar, Modificar, Eliminar, Listar, etc.).
- **Capa Lógica de Negocio**: van las validaciones del negocio, por ejemplo: al momento de registrar una cita, validar que la cita se registre dentro del rango de horas en la que atenderá el médico.
- **Capa Presentación**: se va a desarrollar en entorno web por lo que usaremos MVC



El futuro digital es de todos

MinTIC

Principios de POO



Universidad Tecnológica de Pereira

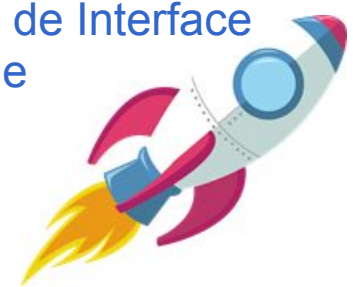
MisionTIC2022

S.O.L.I.D

SOLID es el acrónimo que acuñó Michael Feathers, basándose en los principios de la programación orientada a objetos que Robert C. Martin (tío Bob) había recopilado en el año 2000 en su paper **"Design Principles and Design Patterns"**.

Los 5 principios SOLID de diseño de aplicaciones de software son:

- **S** – Single Responsibility Principle (**SRP**) - Principio de Responsabilidad Unica
- **O** – Open/Closed Principle (**OCP**) - Principio Abierto / Cerrado
- **L** – Liskov Substitution Principle (**LSP**) - Principio de Substitución de Liskov
- **I** – Interface Segregation Principle (**ISP**) - Principio de Segregación de Interface
- **D** – Dependency Inversion Principle (**DIP**) - Principio de Inversión de Dependencias



SRP - Single Responsibility Principle

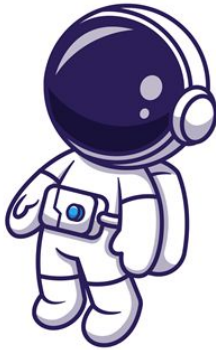
There should never be more than one reason for a class to change.

No debería haber nunca más de una razón para cambiar una clase.

Una clase debería concentrarse sólo en hacer una cosa de tal forma que cuando cambie algún requisito en mayor o menor medida dicho cambio sólo afecte a dicha clase por una razón.

Estel principio es el más importante y fundamental de SOLID, muy sencillo de explicar, pero el más difícil de seguir en la práctica.

El propio Bob resume cómo hacerlo: ***"Reúne las cosas que cambian por las mismas razones. Separa aquellas que cambian por razones diferentes"***.



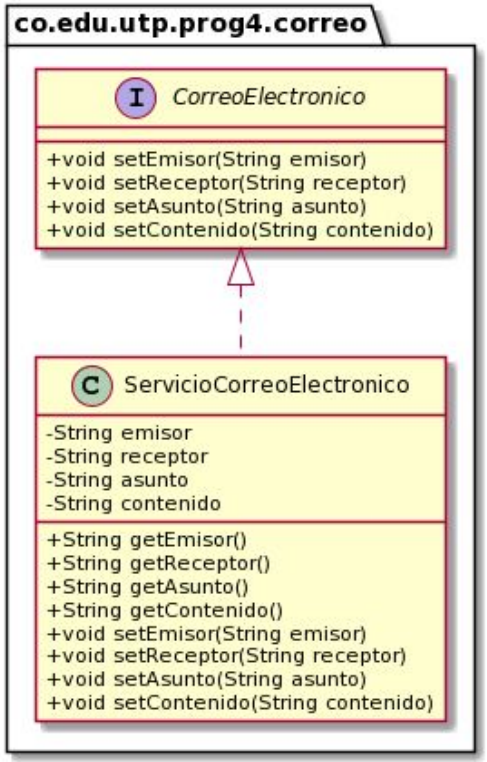
SRP - Single Responsibility Principle

Ejemplo:

Supongamos que tenemos un paquete de software para la gestión de correos electrónicos.

En algún lugar de dicho paquete, antes del envío de los correos en sí, queremos establecer los distintos parámetros:

- Emisor
- Receptor
- Asunto
- Contenido



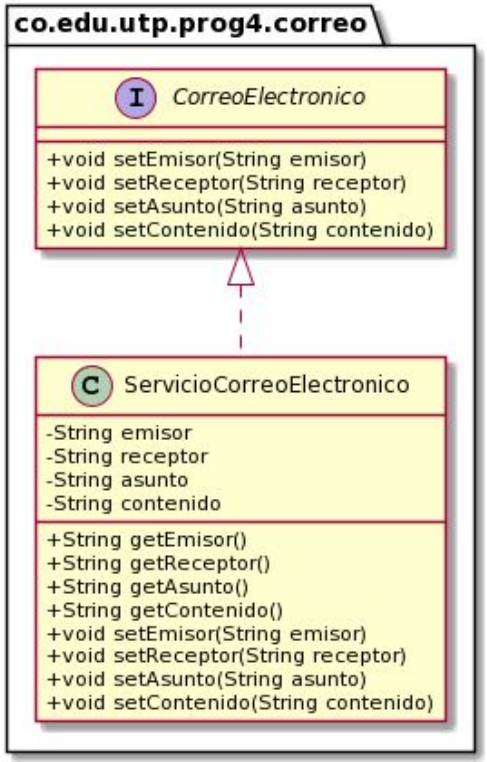
SRP - Single Responsibility Principle
Por qué incumple?

SRP nos dice que sólo debería haber un motivo para cambiar una clase o una interfaz y, por ende, las clases que la implementan.

En éste caso la traducción literal es clara:

- **Si cambiará el contenido** admitiera más tipos, habría que modificar el código cada vez que añadamos algún tipo de contenido
- **Si cambiará el protocolo** y quisiéramos añadir más campos también habría que modificar la clase

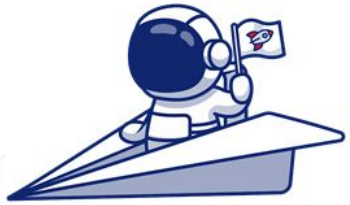
Por tanto hay más de un motivo por el que tendríamos que modificar la clase **ServicioCorreoElectronico**.

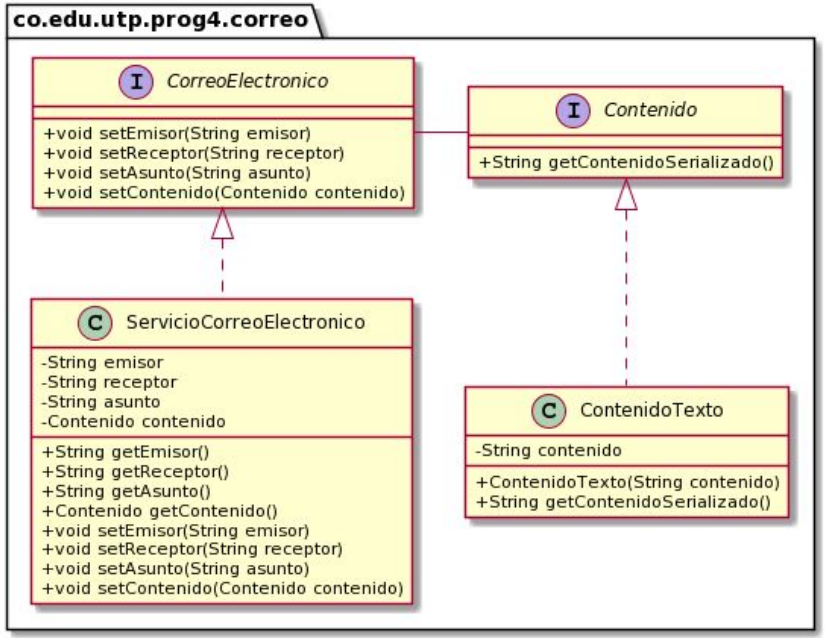


SRP - Single Responsibility Principle

Solución:

Si en vez de un contenido String utilizamos una interfaz **Contenido** que suponga un contrato para hacer viable cualquier tipo de contenido cada vez que nos pidan que añadamos un tipo de contenido sólo tendremos que modificar dicha interfaz y/o las clases que lo implementen, a lo que afecte el cambio, y no ya a la clase de **ServicioCorreoElectrónico** y a su interfaz.





OCP - Open/Closed Principle

Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification

Las entidades de software (clases, módulos, funciones, etcétera) deberían estar abiertas a la extensión pero cerradas a la modificación.

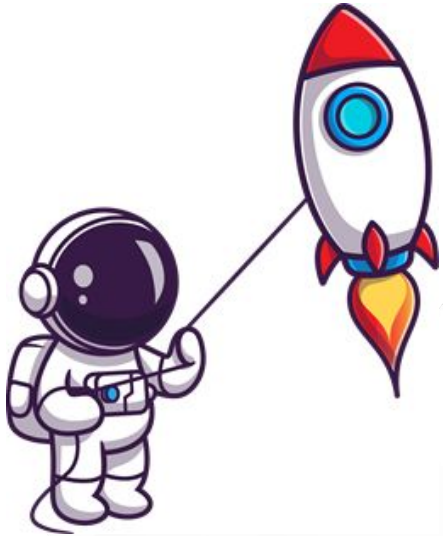
Este principio lo que nos dice es que los métodos ya creados no pueden ser modificados y adaptados. En su lugar deben crearse nuevos métodos que realicen esa nueva funcionalidad. De esta forma, las pruebas seguirán pasando, la aplicación seguirá funcionando y podremos darle más valor a nuestro código con funcionalidades nuevas.

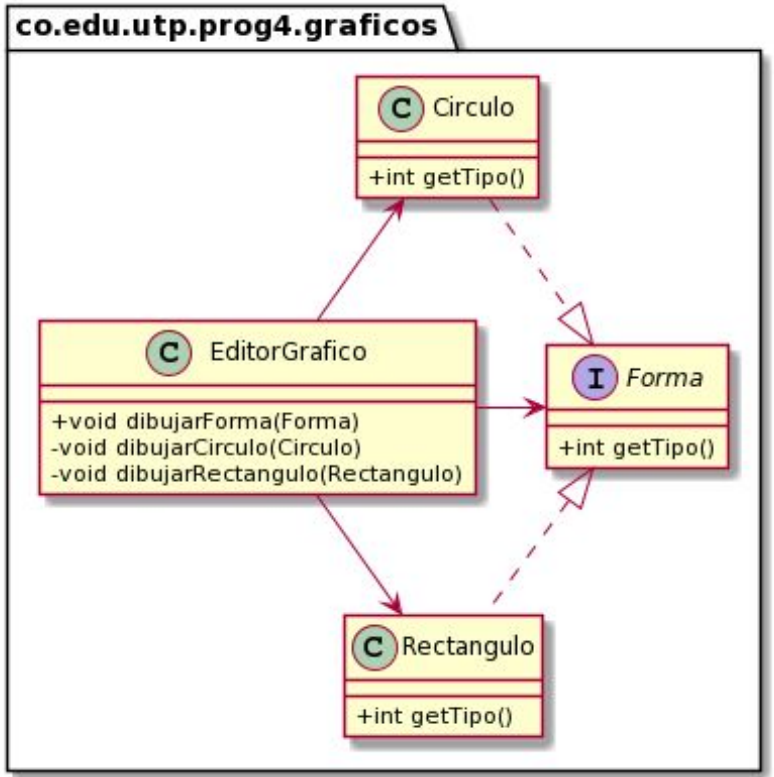
Esto es, anticípate al cambio, prepara tu código para que los posibles cambios en el comportamiento de una clase se puedan implementar mediante herencia, polimorfismo y composición.

OCP - Open/Closed Principle

Ejemplo:

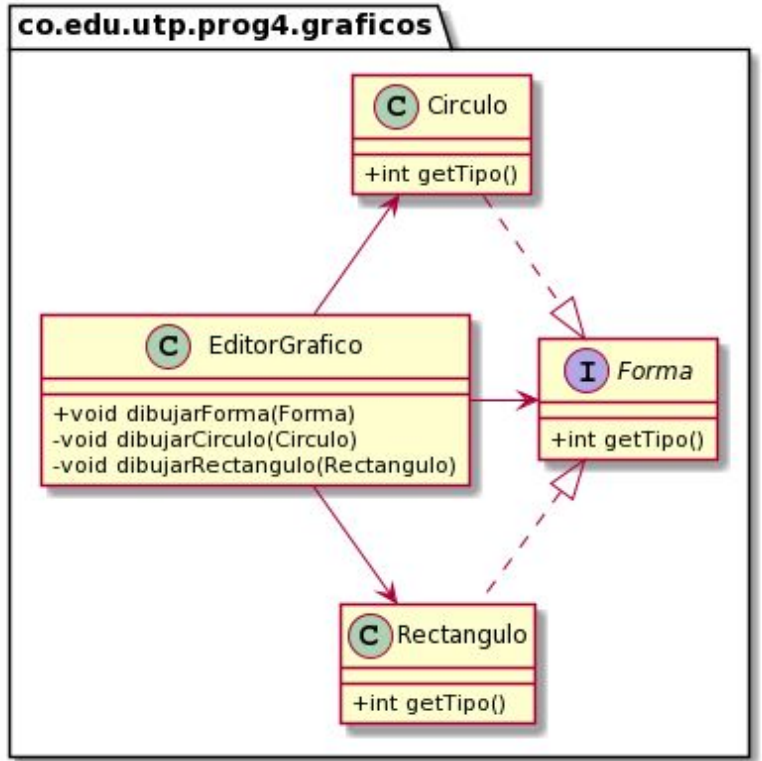
Diseñar un editor gráfico capaz de dibujar un círculo y un rectángulo

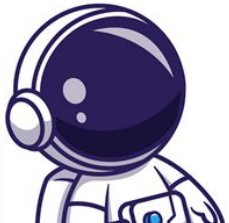




OCP - Open/Closed Principle
Por que incumple?

- La clase **EditorGrafico** no está abierta a la extensión
  - Si quisiéramos añadir una nueva figura geométrica para su pintado, habría que crear la nueva clase y habría que modificar el método público y añadir uno privado para el pintado de dicha nueva forma
  - Añadir una nueva funcionalidad como por ejemplo ***borrarForma()***, ***editarForma()*** o cualquier otra cosa que queramos que haga nuestro editor gráfico, entonces ya tenemos un segundo motivo para el cambio
- Por tanto éste ejemplo incumple los dos primeros principios.

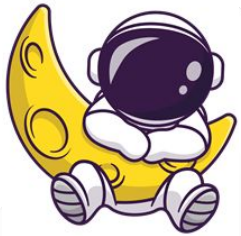


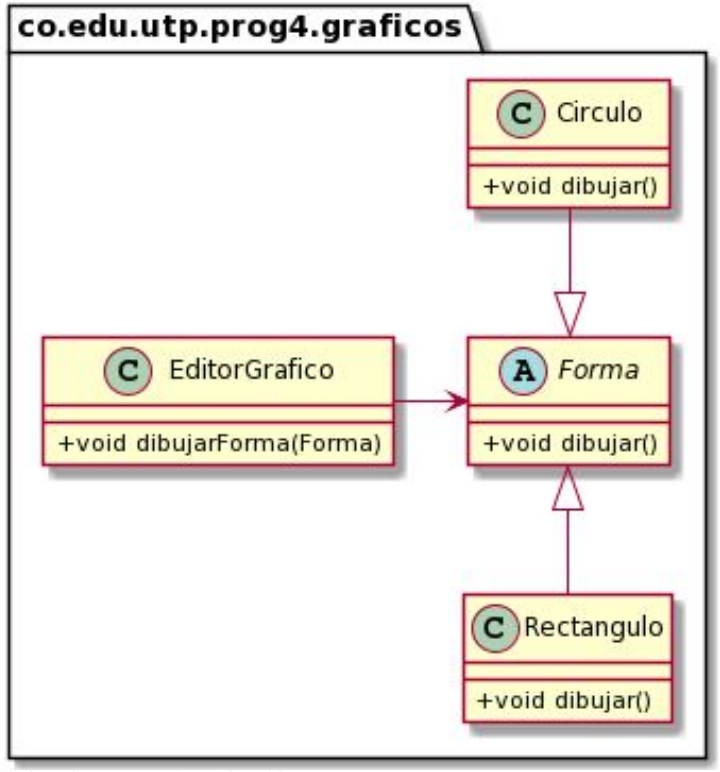


OCP - Open/Closed Principle

Solución:

- Si en vez de una interfaz tuviéramos una clase abstracta **Forma** de la que heredaran las demás e implementaran un método abstracto ***dibujar()***, se quitaría el problema de la extensión y el problema del motivo de cambio.
- Siempre que tenga comportamientos que dependen del tipo piensa en sí se puede implementar con una clase abstracta.





LSP - Liskov Substitution Principle

Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

Las funciones que utilizan punteros o referencias a clases base deben ser capaces de usar objetos de clases derivadas de éstas sin saberlo.

Este principio habla de crear clases derivadas para que puedan ser usarse como una implementación base.

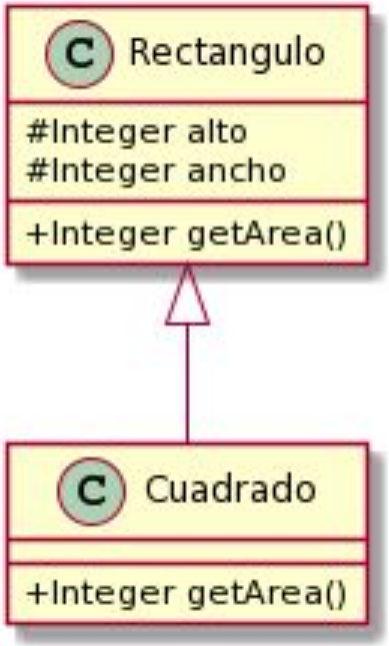
De esta forma, deberemos implementar clases derivadas sin volver a re-implementar lógica y asegurando que no rompemos nada al usar la clase base.

Las subclases deben comportarse adecuadamente cuando sean usadas en lugar de sus clases base.

LSP - Liskov Substitution Principle

Ejemplo:

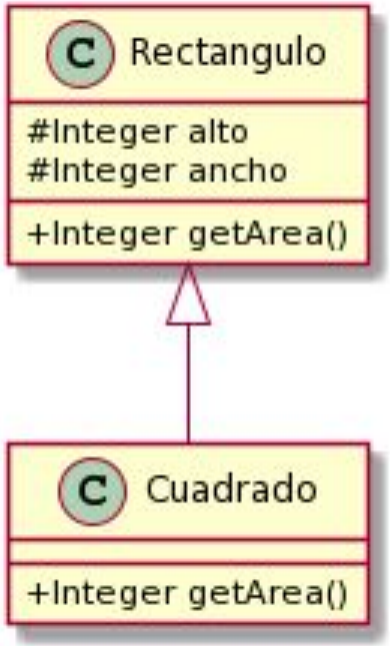
Un cuadrado es un rectángulo con la peculiaridad de que sus lados miden exactamente lo mismo, al menos eso es lo que ha deducido el tan altamente cualificado y estimado grupo de análisis que nos proporciona los requisitos para éste tan poco original ejemplo y quien nos proporciona a su vez el siguiente diagrama estático de clases:



LSP - Liskov Substitution Principle
Por qué incumple?

¿Se ha comportado adecuadamente la subclase en sustitución de la clase padre?

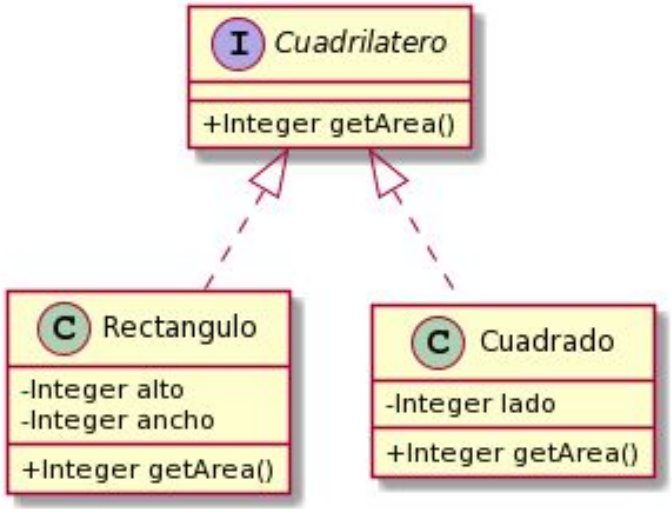
Puede que sea cierto que un cuadrado es un caso particular de un rectángulo en muchos aspectos pero, ¿en el caso del área no se comportan igual?

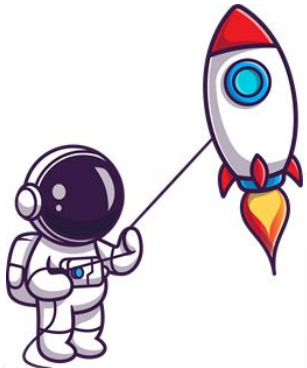


LSP - Liskov Substitution Principle

Solución:

Debido a que el comportamiento de un cuadrado no es coherente con el del rectángulo, la mejor solución es no tener un cuadrado heredando de un rectángulo sino crear una interfaz de la que hereden tanto **Cuadrado** como **Rectángulo**.





ISP - Interface Segregation Principle

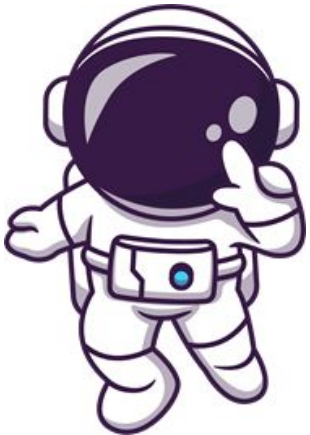
Clients should not be forced to depend upon interfaces that they do not use.

Los clientes no deberían ser forzados a depender de interfaces que no utilizan.

Este principio describe que debemos definir interfaces específicas a una lógica concreta.

Es preferible tener muchas interfaces y poder agruparlas por herencia que no tener interfaces con muchos métodos.

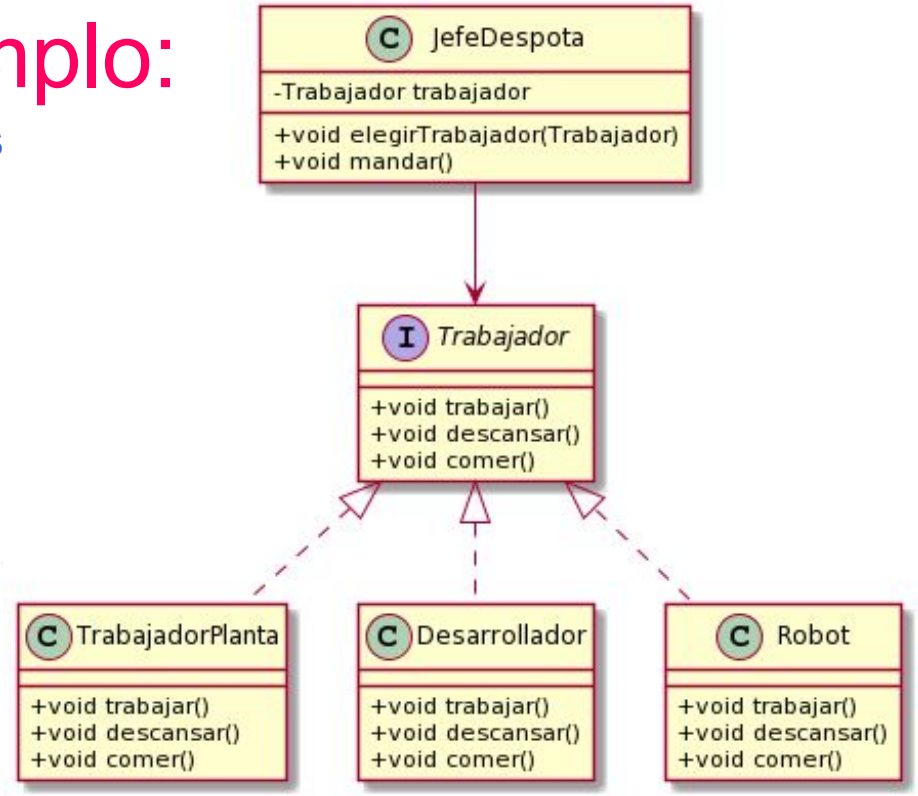
Al definir interfaces, la facilidad de aprovechar la lógica es muy grande y nos facilita el acceso a los métodos de cualquier interfaz desde cualquier clase de nuestro programa.



ISP - Interface Segregation Principle

Ejemplo:

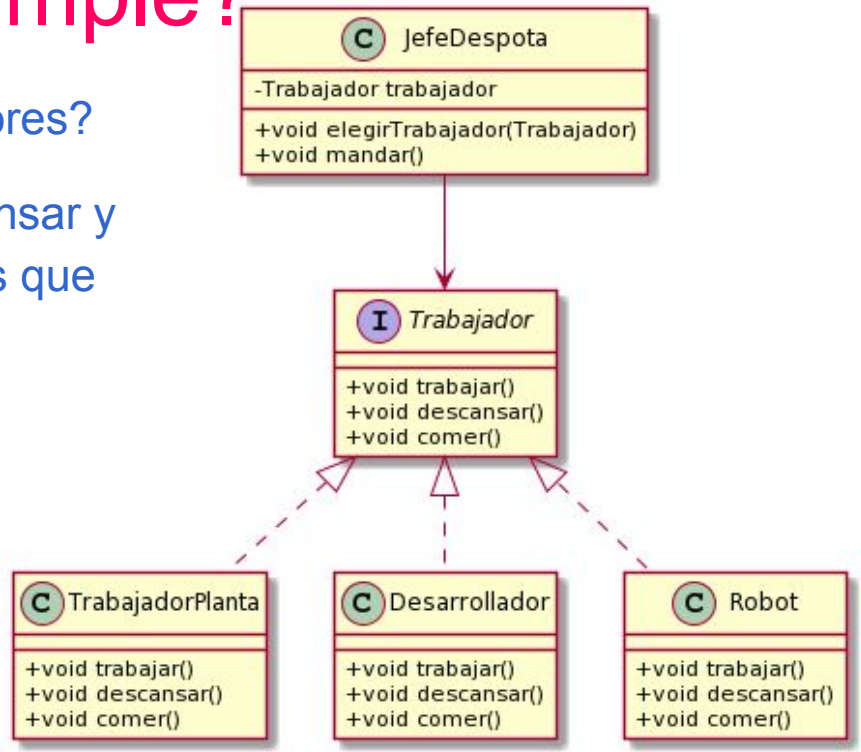
Analizar el siguiente diagrama de clases donde se puede identificar una interface Trabajador que tiene unos métodos que deben cumplir todos los trabajadores en la empresa.



ISP - Interface Segregation Principle
Por qué incumple?

¿Qué sucede con las implementaciones anteriores?

¿Por qué el robot tiene que implementar descansar y comer si nunca va a descansar y mucho menos que coma nada?



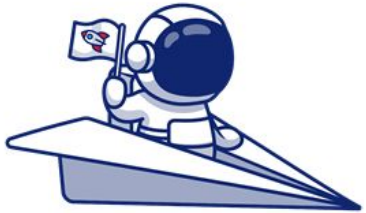
ISP - Interface Segregation Principle

Solución:

¿Cómo podemos separar unas cosas de otras?

¿Le suena por ejemplo ISerializable? Significa que cualquier clase que implemente ISerializable será serializable.

Hagamos lo mismo con **Trabajar**, **Comer** y **Descansar**.



DIP - Dependency Inversion Principle

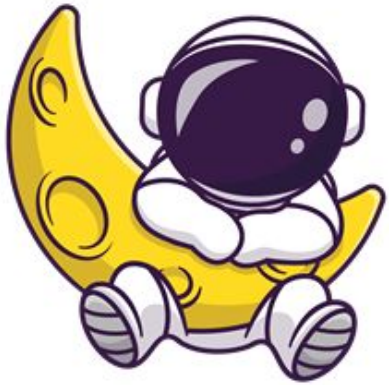
A. High level modules should not depend upon low level modules. Both should depend upon abstractions.

B. Abstractions should not depend upon details. Details should depend upon abstractions.

A. Módulos de alto nivel no deberían depender de módulos de bajo nivel. Ambos deberían depender de abstracciones.

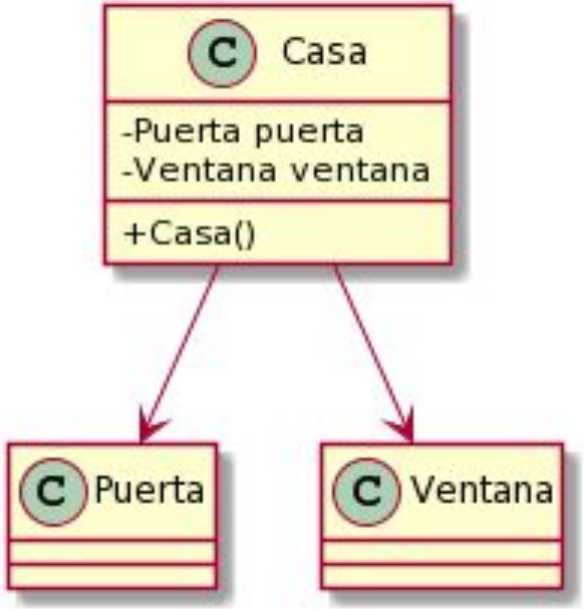
B. Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones.

Para conseguir robustez y flexibilidad y para posibilitar la reutilización haga que tu código **dependa de abstracciones y no de clases concretas**, esto es, utiliza muchas interfaces y muchas clases abstractas.



DIP - Dependency Inversion Principle
Por qué incumple?

En este ejemplo **Casa** depende de **Puerta** y de **Ventana**, es decir, de clases concretas y no de abstracciones.

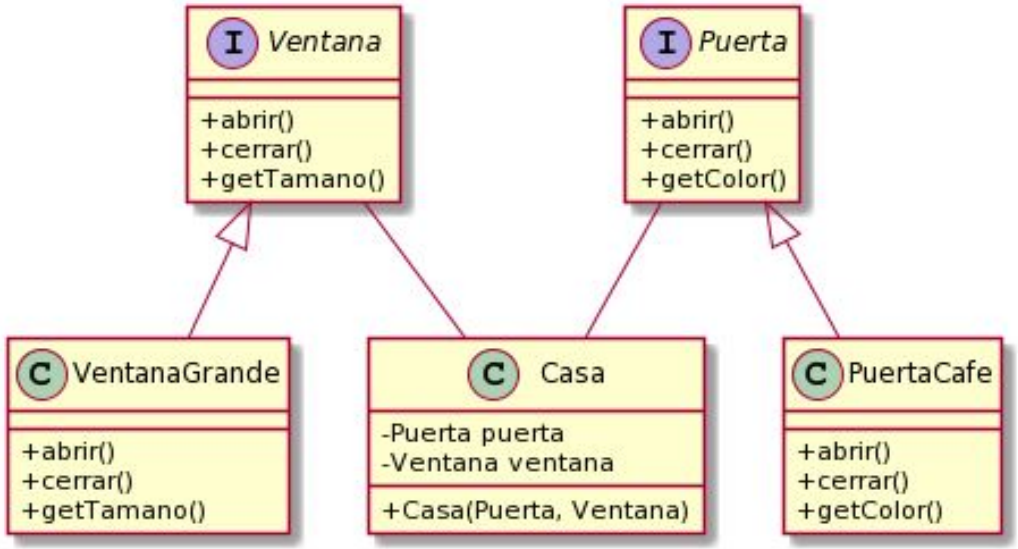


DIP - Dependency Inversion Principle Solución:

Al exponer dos propiedades que son Interfaces estamos dependiendo de abstracciones en vez de en concreciones.

En el constructor hagamos lo que se conoce como “Inyección de dependencias” o “Dependency Injection” (DI)

DI: No crear las instancias de **Ventana** y **Puerta** en la clase en **Casa**, sino que debe recibirlas de alguien más.



Y.A.G.N.I - You Aren’t Gonna Need It

Literalmente traducido por **no vas a necesitarlo**.

Otra fuente común de errores es el código escrito ***“porque podríamos necesitarlo algún día”*** o en ***“creo que lo usaremos más adelante”***.

En general, el desarrollo de cualquier funcionalidad basada en la especulación es una bomba que tarde o temprano estallará.

Ayuda a no ***intentar hacer una mega-solución*** que solucione los problemas del mañana porque estos pueden cambiar, y, en lugar de esto, ***centrarse en únicamente lo necesario***, que por otra parte es por lo que nos pagan.

K.I.S.S - *Keep It Simple, Stupid!*

Aunque se ha traducido normalmente por **manténlo simple, idiota!**, el autor, Kelly Johnson lo ha traducido sin coma, lo que no insulta al ingeniero encargado sino que refuerza la idea de sencillez.

Este patrón lo que nos dice es que cualquier sistema va a funcionar mejor si se mantiene sencillo que si se vuelve complejo.

La sencillez tiene que ser una meta en el desarrollo y la complejidad innecesaria debe ser eliminada.

KISS

D.R.Y - *Don't Repeat Yourself*
D.I.E - *Duplication Is Evil*

Se pueden agrupar estos dos conceptos porque su significado es el mismo, **no te repitas** y **la duplicación es el mal**.

Una fuente común de errores es el código repetido que hace cosas muy similares. Los errores se introducen en los sistemas como código repetido ***cuando se agrega una nueva funcionalidad y solo se actualiza en una de las dos partes de código similar***.

Peor aún peor, si hay un error existente, ***es posible que solo se solucione en un lado***, lo que hace que ***el error se vuelva a abrir debido a la corrección incompleta***.