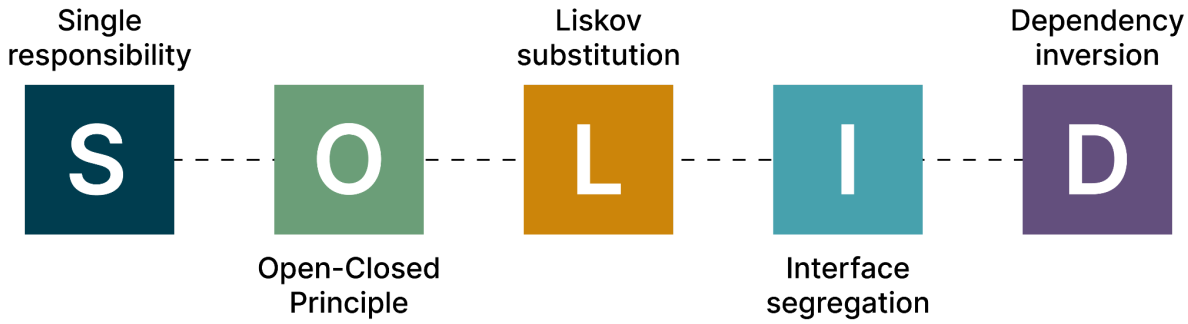
Principios SOLID aplicados

Codely Tv Pro

1. Qué son los principios SOLID (Huyendo de STUPID) 🦄

1.1. 🦄 Qué son los principios SOLID (Huyendo de STUPID)

- ¿Qué son los Principios SOLID?
  - Son principios o convenciones de Diseño de Software ampliamente aceptados en la industria.
    - Los principios SOLID juegan un papel crucial en la OPP.

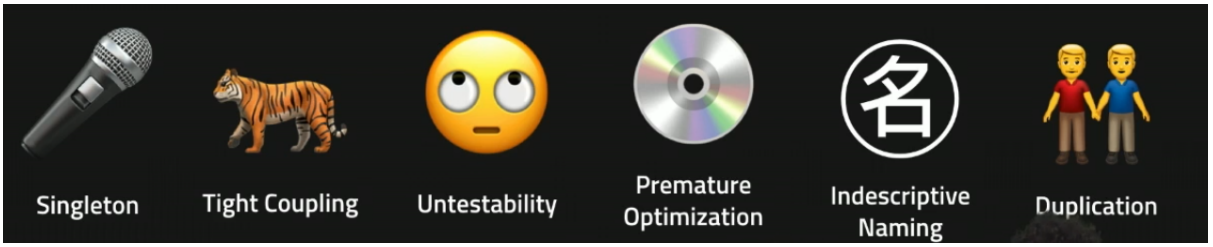


  - Son aplicables en términos de micro y macro diseño de software:
    - Micro-diseño: Referente a clases del código específico.
    - Macro-diseño: A nivel de arquitectura de software y estructura de microservicios. Más complejo y avanzado.
  - Romper el Rol del Arquitecto de Software y entender el diseño de software de calidad como una habilidad y no como un rol.

- Ayudan a hacer un código más mantenible y tolerante a cambios
  - Indispensable para los sistemas complejos actuales, en donde el software se encuentra en constante evolución.

- ¿Qué veremos en el curso?
  - Introducción conceptual
    - Saber el porqué de los principios SOLID, saber cuándo es necesario aplicarlos y cuándo conviene ahorrarnos esa complejidad.

- **<u>STUPID vs SOLI</u>**



  - **S**ingleton
    - Patrón de diseño usado hace años que nos evita exponer colaboradores de clases lo cual hace mucho más difícil el detectar el acoplamiento entre clases.
    - Aunque permite mantener un estado global (en el que sabemos todo sobre el - malo xq no está desacoplado), esto se consigue de forma más eficiente y limpia con la inyección de dependencias.
    - Esto perjudica la testeabilidad haciéndola más difícil.

  - **T**ight Coupling
    - Código fuertemente acoplado en el que realizar cambios es muy engorroso.
      - Si cambias algo, debes modificar varias clases porque el código está fuertemente acoplado.

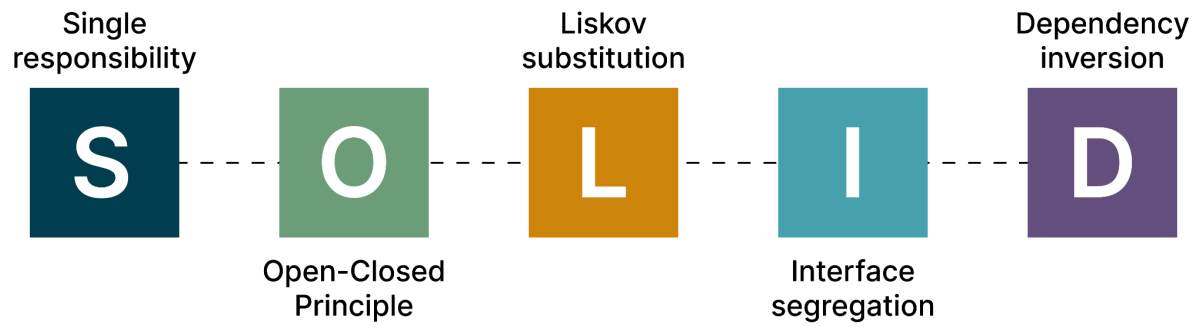
- **U**ntestability
  - Derivado del hecho de No inyectar las dependencias via Constructor, sino de methos estáticos acoplados.

- **P**remature Optimization
  - Anticiparnos a los requisitos de nuestro software desarrollando abstracciones innecesarias que añaden complejidad.

- **I**ndescriptive Naming
  - Nombres descriptivos para nuestras variables.

- **D**uplication
  - Duplicidad de código. Algo solucionable si aplicamos el Principio de Responsabilidad Única de SOLID e inyección de dependencias como veremos en este curso

- SOLID:



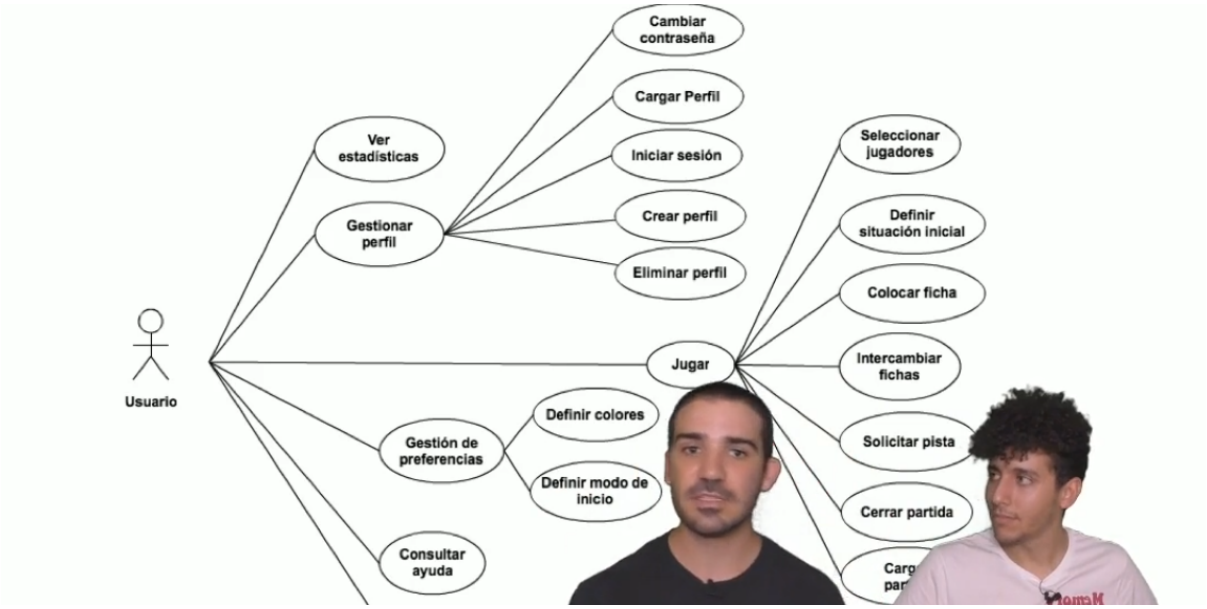
2. UML, ese gran denostado

2.1. UML

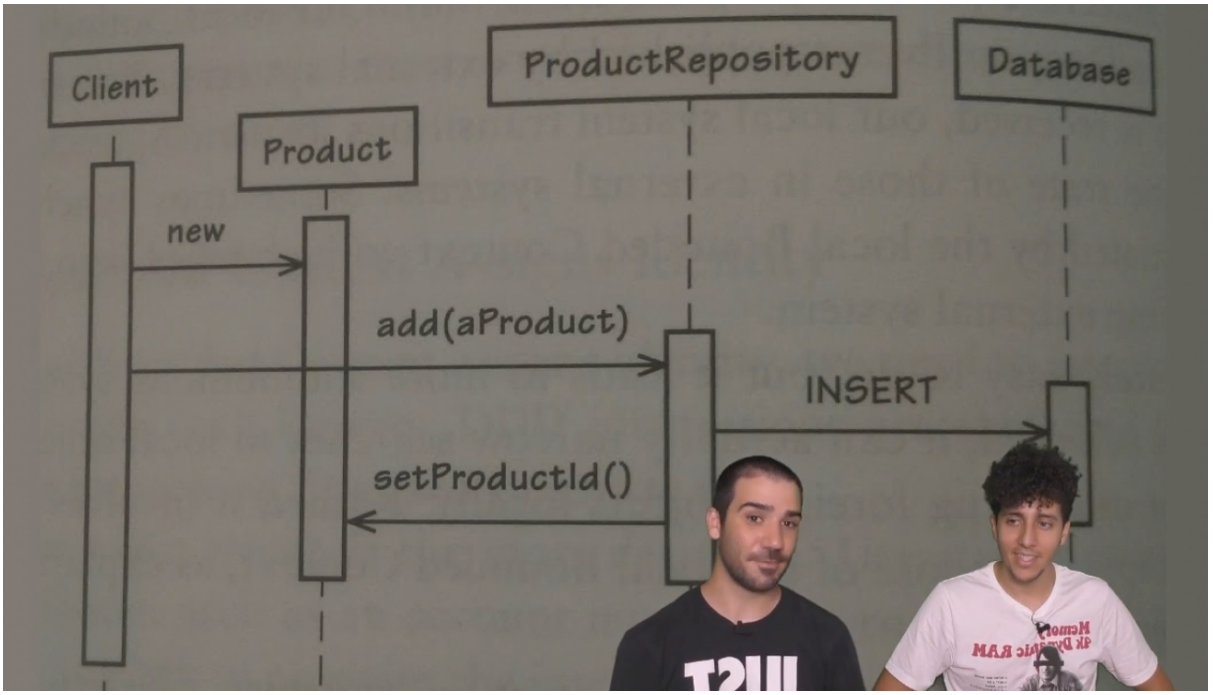
- ¿Qué es UML?
  - Lenguaje de diagramas ilustrativos para nuestros diseños de software.
    - Con 4 garabatos tenemos una idea de lo que queremos hacer.
  - Riguroso: Permite identificar acoplamiento entre clases y sus relaciones sin ser verboso (LP)

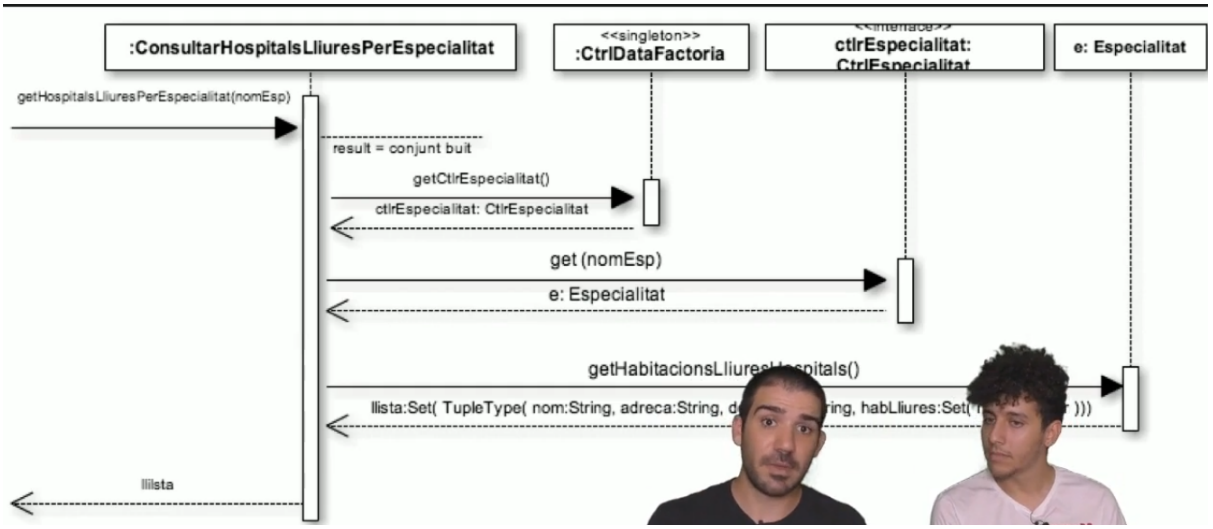
  - Agnostico: Agnostico al LP

- ¿Qué tipos hay?
  - Casos de uso, Secuencia y Clases
    - Casos de usos: Capa de aplicación. Visualizar las funcionalidades de nuestra app.

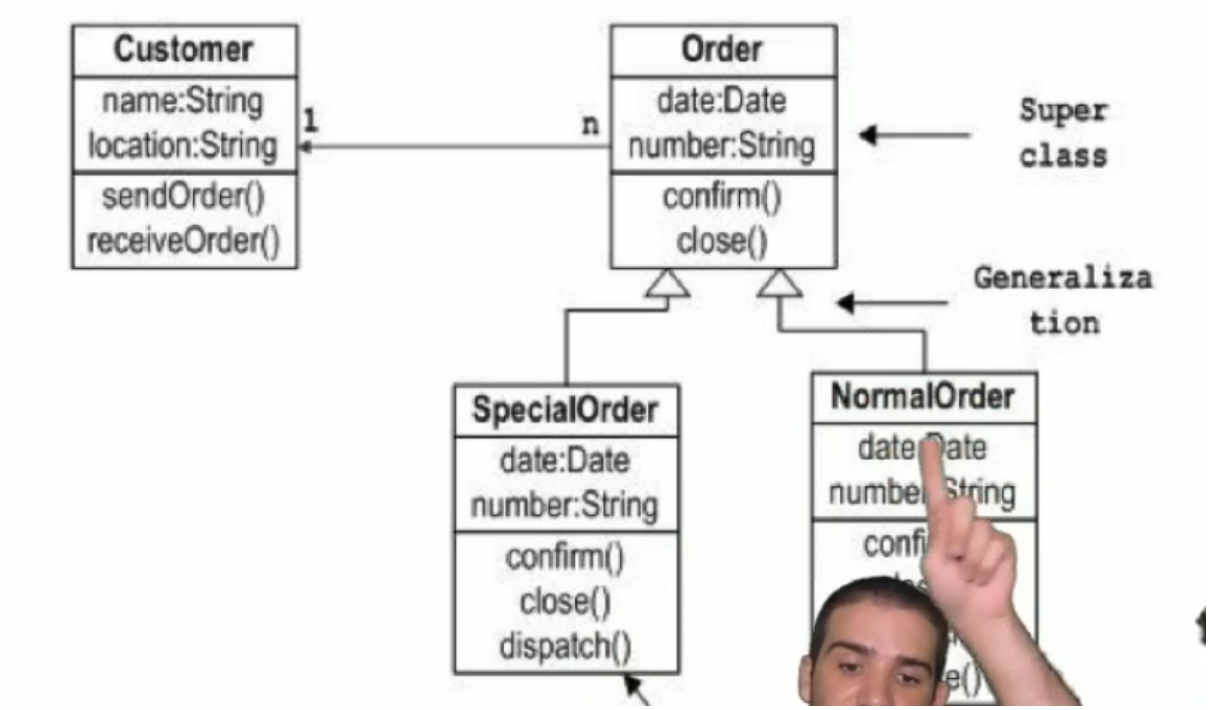


    - Secuencia: Flujo de nuestra App. Representa como interaccionan nuestras clases.

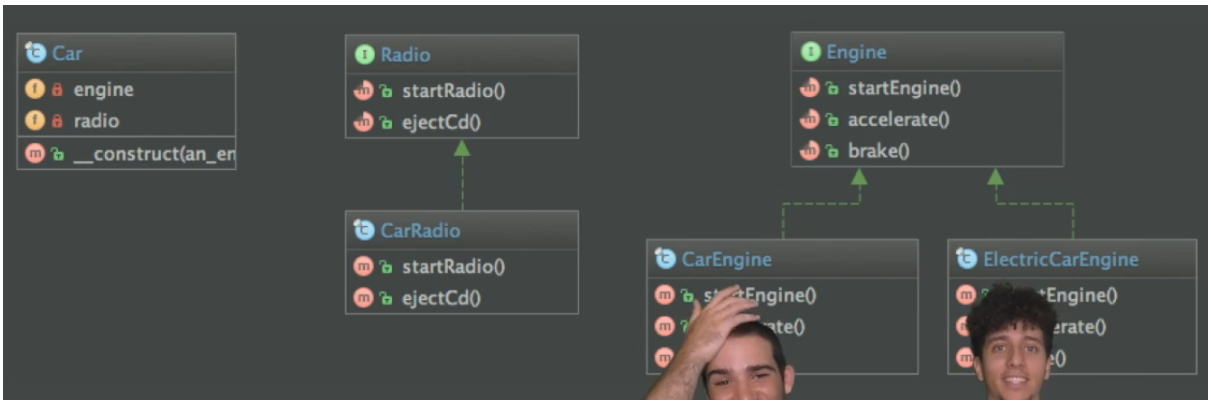




- Clases: Muy popular porque permite ver atributos, methods de cada clase, herencia, interfaces e implementaciones de estas.
  - Cada rectángulo representa una clase
    - Nombre de la clase > Atributos/Propiedades/Methods
    - Herencia con las flechas sin relleno y línea continua.



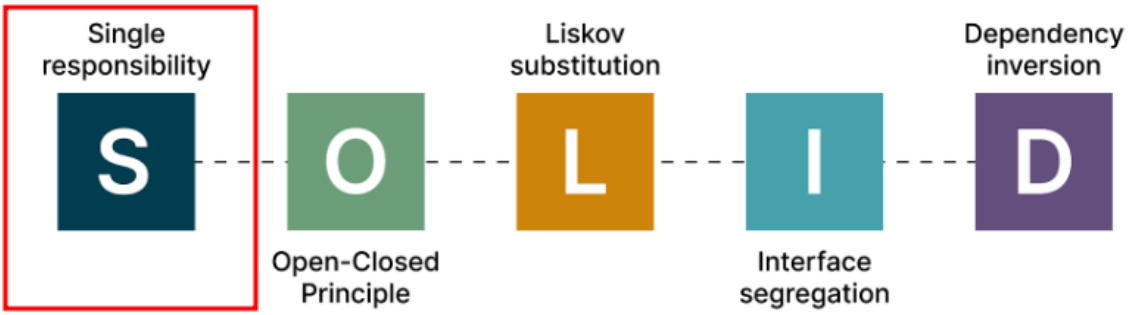
IntelliJ hace esto en automático con las clases existentes:



- Interfaces - Bolitas verdes con la I
- Clases finales - Bolitas azules con la C
  - Atributos/fields - Naranja con la f
  - Methods - Rosado con la m
    - Si tiene el candado es un Public Method (privacidad del method/atributo)

3. Principio de Responsabilidad Única 🕺

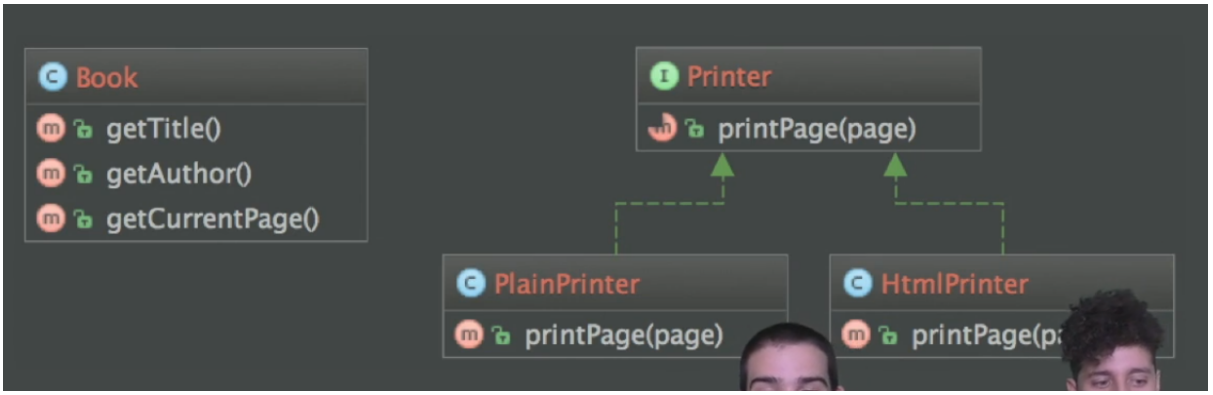
3.1. Introducción conceptual al Principio de Responsabilidad Única (Single Responsibility Principle – SRP)



- Introducción Conceptual
  - Concepto de SRP:
    - Una Class debe representar un único Concepto y Responsabilidad. Una Class debería tener solo 1 razón por la cual debería cambiar.
  - Como:
    - Clases de servicios pequeñas con objetivos acotados.

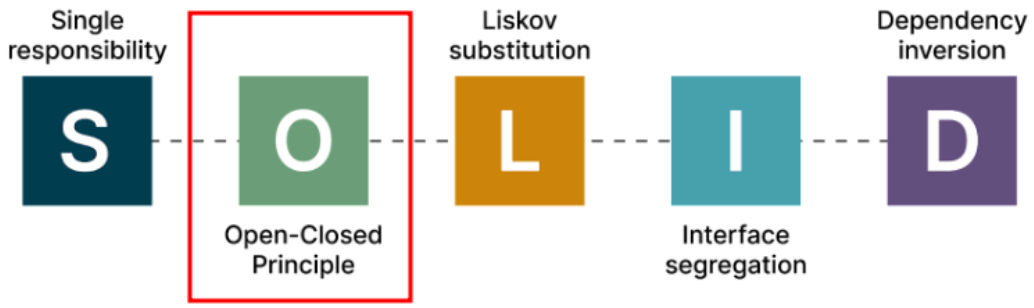
    - Modelos de dominio: Modela lo que tenemos en el sistema. Como es el User, Carrito de compra, pedidos, etc. Clases que nos sirven para definir qué atributos (Properties) y comportamientos (Methods) deben tener cada uno de ellas.
    - Servicios: Un servicio es algo que orquesta una serie de pasos interaccionando con otros elementos de nuestro sistema.
      - Verificar que se respete el SRP: Verificar si un Service tiene >1 public methods. Si tiene >1 public method hace 2 cosas diferentes, por tanto no respeta el SRP.
  - Finalidad:
    - Alta cohesión y robustez: Código tolerante al cambio
    - Permite composición de clases: Al tener todo tan segmentado, cuando se requiera se puede aplicar una inyección de una clase para no tener que copiar código.
    - Evita duplicidad: Con la inyección de clases segmentadas. Respetando la Dependency Inversion Principle.

- Nivel de granularidad
  - Order | User
    - Son Modelos de Dominio, NO services.
  - NO definir nombres muy generalizados/ambiguos.
    - Nombres como OrderAnalyzer | OrderProcessor son nombres genéricos que llevan a más de 1 responsabilidad.
  - Definir nombres más característicos/significantes
    - OrderMarginCalculator ← Sabemos de inmediato que es un servicio que únicamente va a calcular el margen de ganancia en una orden.



4. Principio de Abierto/Cerrado 🤯

4.1. Introducción conceptual al Principio de Abierto Cerrado (Open/Closed Principle – OCP)



- Introducción Conceptual
  - Concepto:
    - El Software debería estar ***abierto a expansión y cerrado a modificación***.
      - Si tenemos un Switch para gestionar los Roles de Usuario, si se agrega un nuevo rol debemos '*modificar*' agregando un nuevo case en todos los sitios que hagan uso de esta validación. Esto es una violacion del principio OCP.
  - Como:
    - NO depender de implementaciones específicas
      - *En lugar sl Switch tener Interfaces bien definidas*
        - Con lo cual, si agregamos un nuevo rol, en lugar de tocar case en cada switch, SIMPLEMENTE implementamos esa ***Interface*** y ya está.
  - Finalidad:
    - Facilidad a la hora de agregar nuevos casos de uso en nuestra app.

- Interfaces vs Abstract Classes
  - Beneficios *Interface*
    - No modifica el árbol de jerarquía
    - Permite implementar N interfaces
    - En TS no son transpiladas a Código
  - Beneficios *Abstract Class*:
    - Permite el patrón *template method*
      - Dificulta la trazabilidad
        - Deberíamos navegar entre clases hijas y abstract classes para saber cómo se comporta cierto Method
    - Permite tener Getters privados
    - Aplicado a través de la *Inheritance* entre clases.

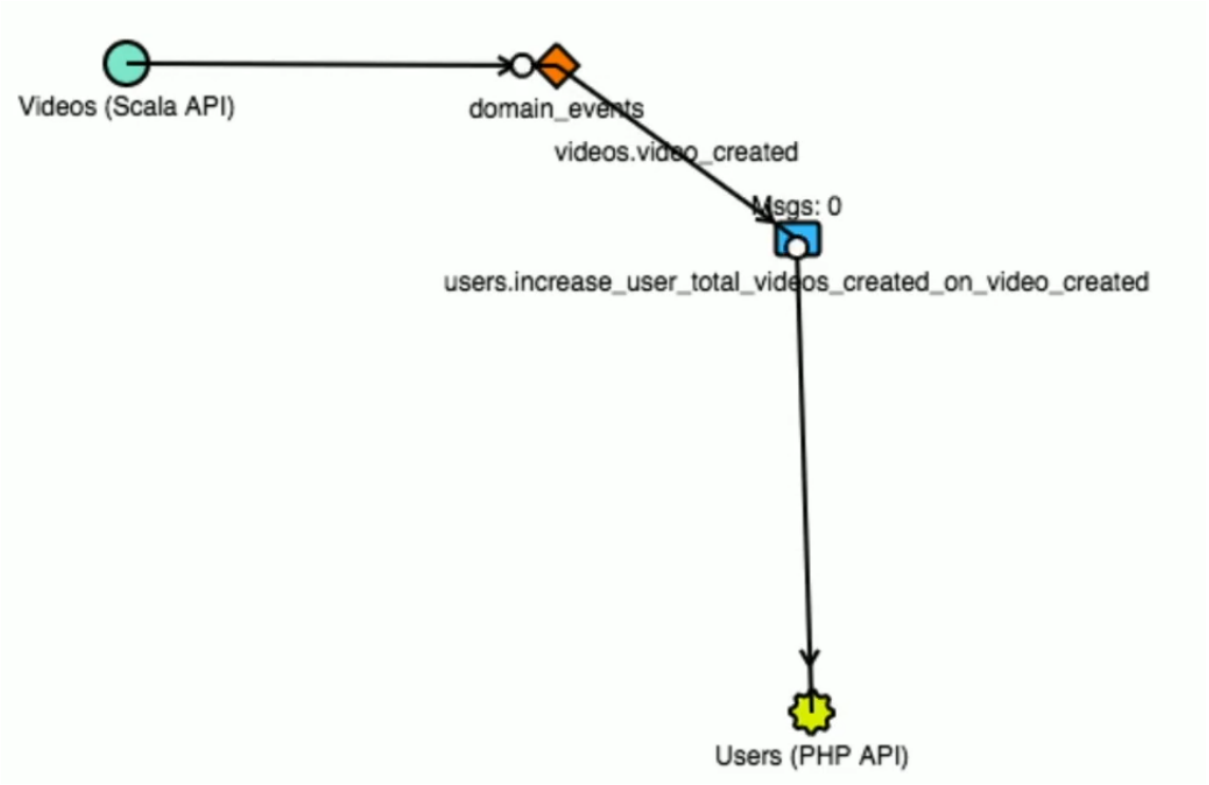
  - Interfaces
    - Desacoplar entre capas (Arquitectura Hexagonal)
  - Abstract Clases:
    - Determinados casos para Modelos de Dominios.

4.2. Keep it real

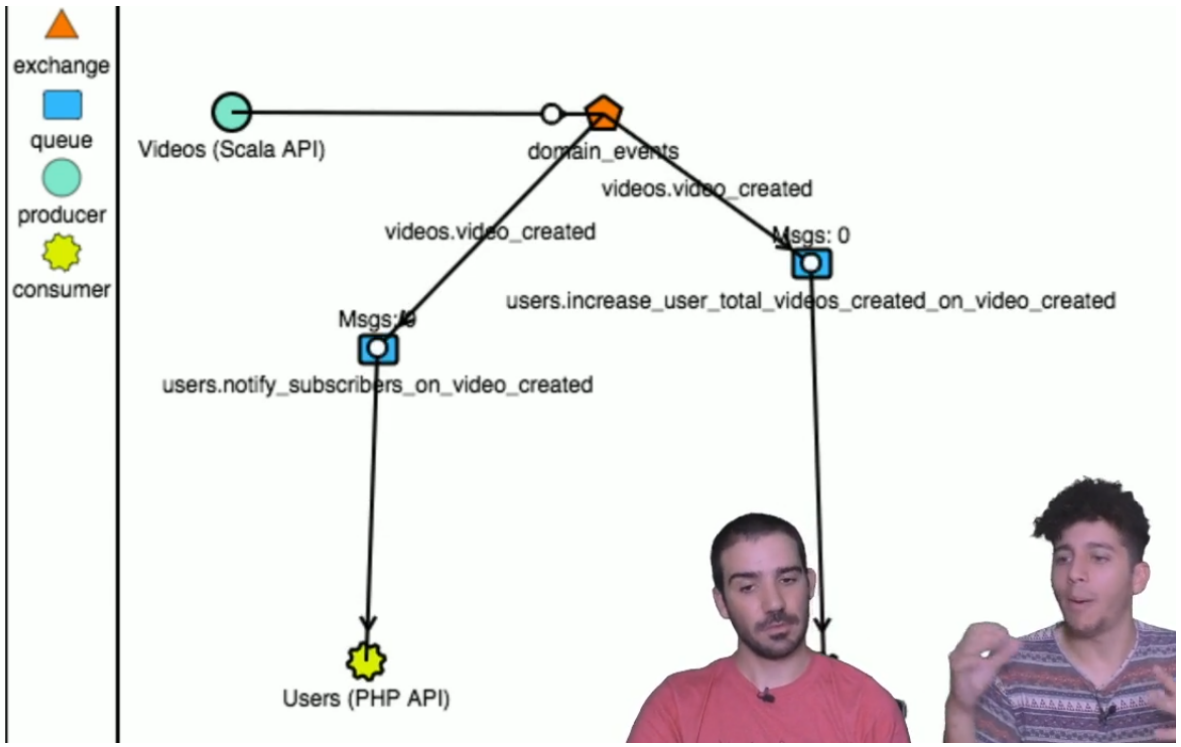
- Me quedo con la acción core de la Clase. Para el caso de uso Crear un nuevo video lo indispensable es guardarlo en DB.
  - En este caso cuando creo un nuevo curso, lo que más me interesa es que se guarde en DB. En tal contexto, solo necesito Publicar un Evento (de dominio) que informe que se ha guardado en DB (Programación Reactiva). Otros Servicios escuchan este evento y reaccionan (Ver curso de *Comunicación entre Microservicios*).
    - s
      - s

  - Tenemos un sistema productor de eventos (API en Scala)
    - Las bolitas blancas son el evento creado (JSON)
      - JSON con todos los datos que representan el video (ID, descripción, subtitle, etc.)
      - Ese JSON se publica en un Sistema de Colas (Queue) para que cualquiera que esté interesado se suscriba a él.

    - El User es otro sistema en otro lado escrito en cualquier otro LP
      - Consume esos eventos xq está suscrito a la Queue y hace lo que tenga que hacer.



- Hacer todo esto nos permite una fácil extensibilidad. Podemos agregar más funcionalidad al sistema Únicamente SUSCRIBIENDO este a la Queue.
  - Con esto, cada vez que tenga un nuevo caso de uso únicamente creamos un suscriptor más que esté escuchando el evento creado.
    - Aun teniendo un Monolito (sin estructura de micro-servicios) es Importante tener un Sistema de Colas (Queue)
  - Las letras de SOLID no siempre van solas, en este caso la S y la O van muy juntas.

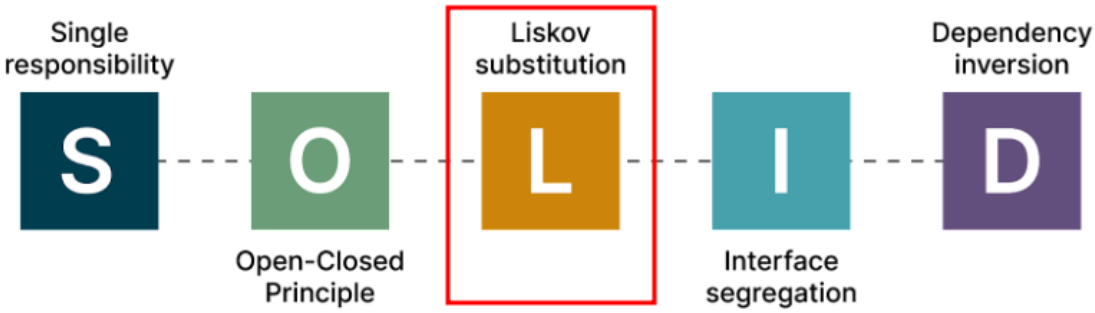


- Evento Vídeo creado publicado en un Sistema de Colas al que están suscritos otros sistemas que reaccionen en consecuencia.
  - Tenemos un Evento Crear vido
    - Un sistema (caso de uso) que escucha ese evento e incrementa el contador de videos totales.
    - Un sistema (caso de uso) que notifica a los usuarios de la app.

  - Esto nos da más flexibilidad (importante una Queue aunque sea un monolito).

5. Principio de sustitución de Liskov 🔁

5.1. Introducción conceptual al Principio de Sustitución de Liskov (Liskov Substitution Principle – LSP)

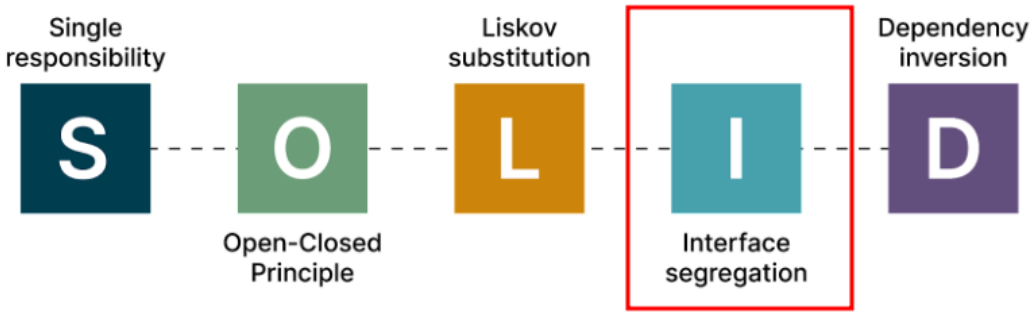


- Introducción Conceptual
  - Concepto
    - Si **S** es un *subtipo* de **T**, instancias de **T** deberían poderse sustituir por instancias de **S** sin alterar las propiedades del programa.
      - *Clases Hijas respeten el contrato de las Parent Classes.*
  - Como:
    - El comportamiento de las subclases debe respetar el contrato de la superclase.
  - Finalidad:
    - Proveer el escenario para aplicar OCP

- Resumen:
  - SRP para Clases pequeñas y acotadas (rompiendo el acoplamiento) de responsabilidad, y LSP son la premisa para poder aplicar OCP.
    - Que las Clases sean pequeñas y que haya un contrato robusto que se mantenga a lo largo de la jerarquía.

6. Principio de segregación de interfaces 💥

6.1. Introducción conceptual al Principio de segregación de interfaces (Interface Segregation Principle)



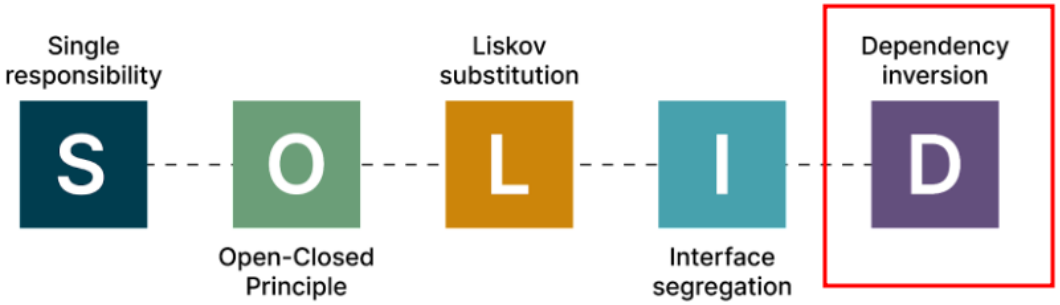
- Introducción conceptual
  - Concepto:
    - Ningún cliente debería verse forzado a depender de methods que no usa.
      - En ese contexto el Cliente es el Caso de Uso (Class)
  - Como:
    - Definir contratos de *interfaces* basándose en los clientes que las usan y no en las implementaciones que pudiéramos tener.
      - Evitar *Heder Interfaces* promoviendo *Role Interface*.
    - Las Interfaces pertenecen a los Clientes.
  - Finalidad:
    - Alta cohesión y bajo acoplamiento estructural

- Ejemplo:
  - Queremos poder enviar notificaciones vía email, Slack, o fichero txt ¿Qué firma tendrá la interface? 📨
    - notifier(content)
      - Por descarte es esta, pero lo mejor es tener un evento al que están suscritos subscribers diferentes (email, slack).

- Las Interfaces pertenecen a los Clientes
  - Debemos definir las interfaces en base a los Casos de Uso.

7. Principio de inversión de dependencias 🤹‍♀️

7.1. Introducción conceptual del Principio de Inversión de Dependencias (Dependency Inversion Principle)



- Intro
  - Concepto:
    - Módulos de alto nivel NO deberían depender de los de bajo nivel. Ambos deberían depender de abstracciones.
      - El Caso de Uso (Class/módulo) que estaría a *Alto Nivel* y la Interfaz para ese Caso de uso que también estaría a alto nivel. Estarían al mismo nivel, hablando de igual a igual.
        - Ej.: Si tenemos un Notificador de Slack, mi Caso de Uso NO debería depender del notificador de Slack sino de la Interface Notifyer.
        - Aquí se establece un alto nivel, bajo nivel entendido como las capas de la arquitectura. Esto queda mucho más claro en el curso de Arquitectura Hexagonal.

  - Como:
    - Inyectar dependencias (parámetros recibidos idealmente en el Constructor)
    - Depender de las Interfaces (contratos) de estas dependencias y NO de las implementaciones concretas.
    - LSP como premisa: Los contratos que establecen nuestras Interfaces se deben respetar a lo largo de todas las Subclases.

  - Finalidad:
    - Facilitar la modificación y sustitución de implementaciones.
    - Mejor estabilidad de clases.