Principio SOLID

1. Single Responsaility principle
   Principio de responsabilidad unica→ habla de que los componentes de software en especial las clases deben teber un único motivo para cambiar eso es una única responsabilidad
2. Open/ close principle

   Las clases deberian estar abiertas para extender pero cerradas para modificar

   Esto significa que las clases solo deberían modificarse si tuvieran un problema de funcionalidad, un bug. Pero si nosotros quisiéramos extender esas clases deberíamos hacerlo a partir de uno de los métodos de rehúso como herencia o composición de objetos.

3. Liskov Substitucion principle
   Principio de substitution de Liskov→ la relación de las clases deberían de ser siempre el mayor nivel de abstracción y de esta manera en donde encontremos una clase de alto nivel, podremos reemplazar esa clase por cualquier otra clase del mismo tipo mucho más abajo en la jerarquía.
4. Interface segregation principle
   Este principio hace referencia a que las clases no deberían tener la obligación de implementar operaciones o interfaces que realmente no utilicen.
5. Dependency inversion principle
   Hace referencia a que la relacion que deben tener las clases siempre al mayor nivel de abstracción y a partir de esto invertir las dependencias entre estos objetos logrando que todo el código sea mucho más reutilizable y mucho menos dependiente, con esto también logramos bajar el acoplamiento de nuestros sistemas.

Receta para un código excelente

- Identación
- Nombres de variables claros y concisos
- 100% de cobertura en las pruebas
- Código mantenible

- Cualquier código que no sea mantenible y que no pueda adaptarse a los contextos cambiantes con facilidad, es un código que solo espera volverse obsoleto.

Single Responsibility principle

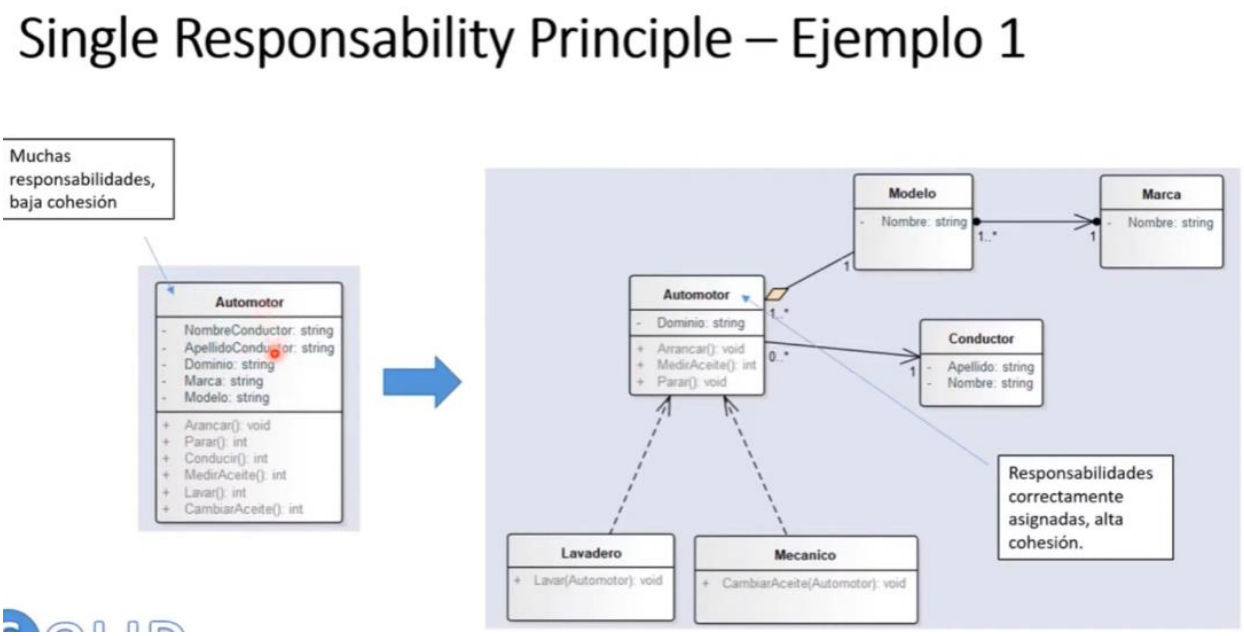
Una clase debe tener solo una razón para cambiar, hace referencia a una única responsabilidad

- Una responsabilidad tiene que ver con
  - ¿Qué hace?
  - ¿A quién conoce?

Se centra en mantener la cohesión

Cada clase debe tener responsabilidad sobre una sola parte de la funcionalidad proporcionada por el software, y esa responsabilidad debe estar completamente encapsulada por la clase.

Ejemplo1

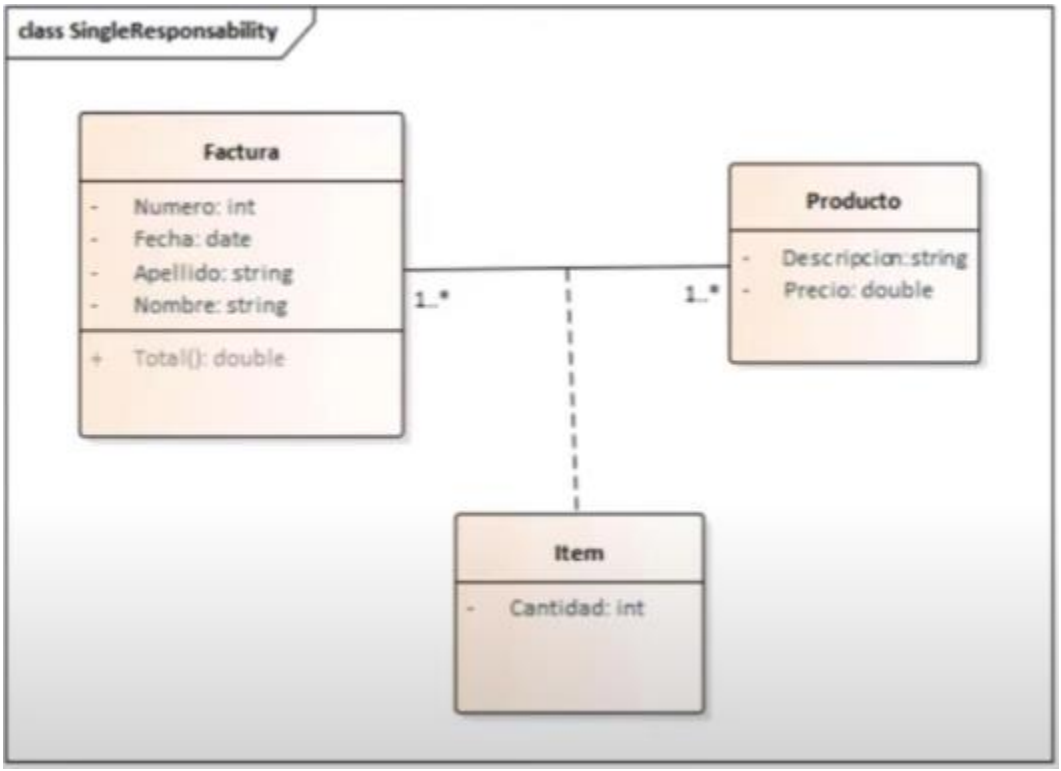


La clase primera clase automotor tiene muchas responsabilidades y por tanto baja cohesión, esta clase podemos estructurarla definiendo clases satélites que realmente se encarguen de ciertas responsabilidades como la case Conductor para el nombre y el apellido de conductor o las clases de lavadero y mecanico que

hace funciones realmente fuera del automotor ya que nos las hace el propio automotor. Dejando la clase automotor exclusivamente con responsabilidades propias del automotor ya que las hace el como el arrancar o el parar.

Ejemplo 2

Proyecto original



Resultado en proyecto: `namespace SOLIDEjemplo2`

Open-closed Principle

Este principio nos dice que las clases deberían estar abiertas para extender pero cerradas para modificar.

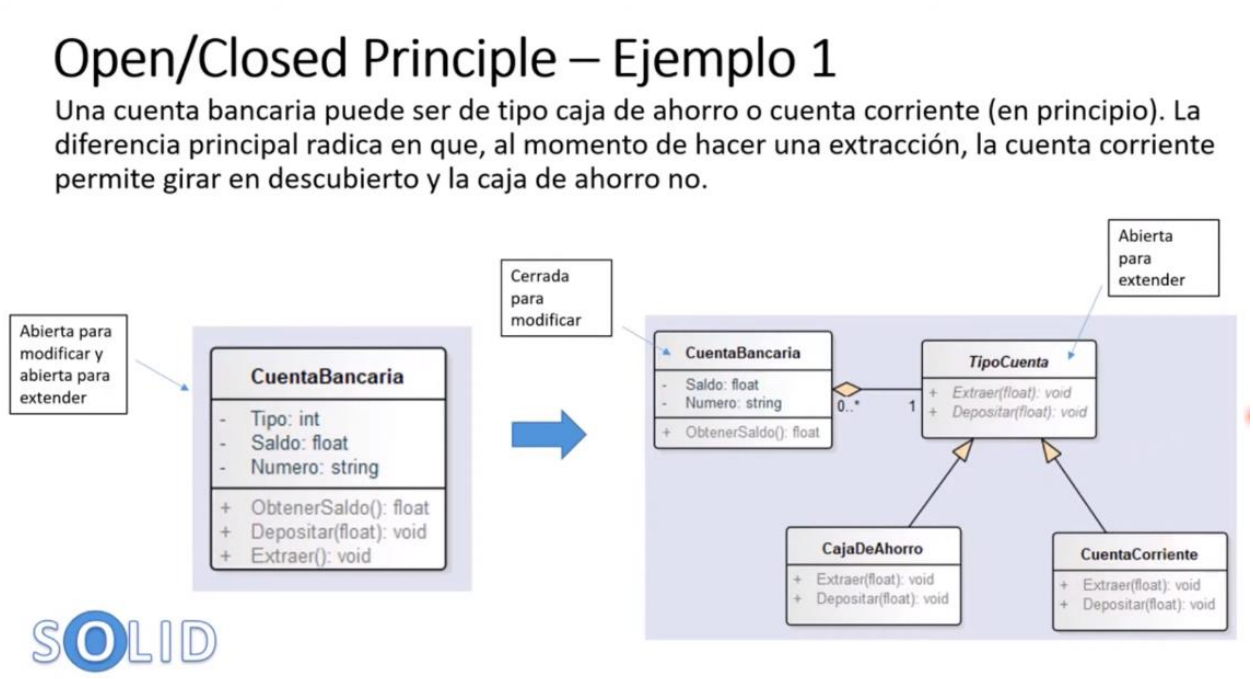
El comportamiento de una entidad debe poder ser alterado sin tener que modificar su propio código fuente.

Una clase no se puede modificar, pero si se puede extender haciendo uso de la herencia.

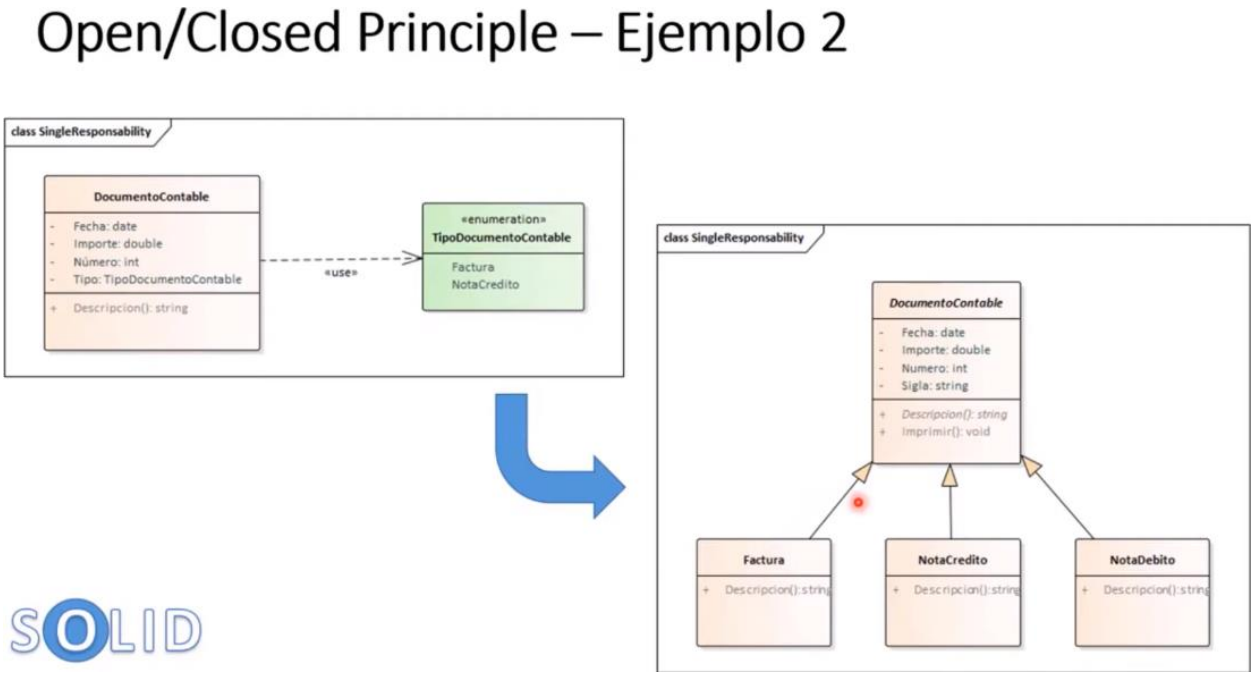
Una clase solo debe ser modificada si existe un bug, para no romper funcionalidades en módulos dependientes.

Se centra en mantener bajo el acoplamiento.

EJEMPLO 1



Ejemplo 2



En el segundo ejemplo si queremos agregar un nuevo tipo de factura tendríamos que agregar el nuevo tipo al enumerado y modificar el método de descripción para poder obtener la info del nuevo tipo de factura. No nos permite extender la clase de la forma que está programada. Por eso siempre hay que pensar en el rehúso.

Posible Solución→ Documento contable pasa a ser una clase abstracta sin el atributo Tipo ya que lo eliminamos, el método Descripción pasa a ser abstracto ya que cada clase que herede de DocumentoContable hara su propia implementación.

En lugar del enumerado Tipo, creamos clases para los diferentes tipos de factura donde heredamos de DocumentoFactura e implementamos el método Descripción acorde al nuevo tipo de factura.

En caso de tener que añadir un nuevo tipo de factura simplemente creamos una nueva clase que herede de DocumentoFactura sin tener que modificar las clases bases, solo haciendo extensiones mediante herencia.

Ejemplo Solución SOLID.OpenClose

Principio de sustitución de LISKOV

Se basa en mantener las clases de tal manera que la jerarquía de las clases sea correcto.

Es una extensión del principio Open/closed,

Una clase derivada puede ser reemplazada por cualquier otra que use su clase base sin alterar el correcto funcionamiento de un programa.

Si una función espera como parámetro una clase base esta puede ser reemplazada por cualquier clase derivada.

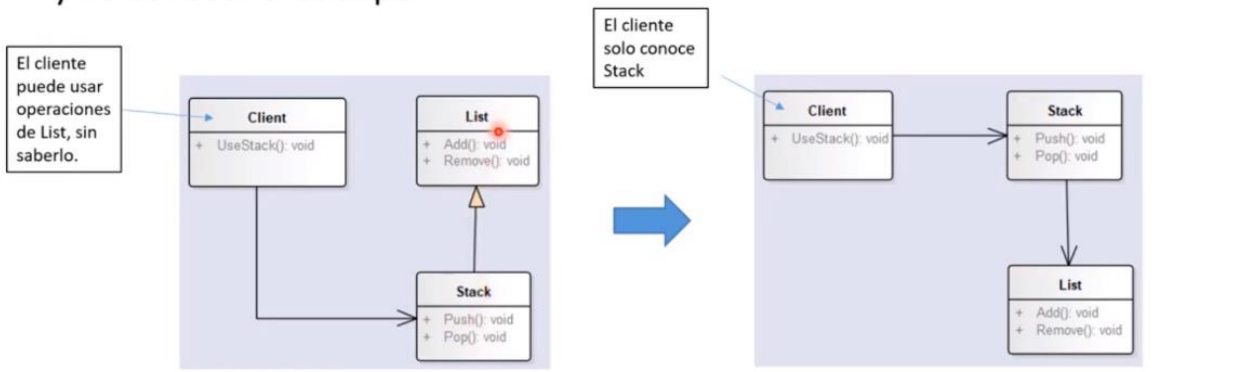
Una subclase no debe remover ni modificar comportamiento de la clase base, no debe conocer a los demás subtipos.

Si un subtipo hace algo que el cliente del supertipo no espera, es una violación al principio.

Liskov Substitution Principle – Ejemplo 1

¿Qué pasa si se ejecuta Remove()?

Corresponde a List y no a Stack. Debería tener una referencia a la clase BASE y no conocer el subtipo.



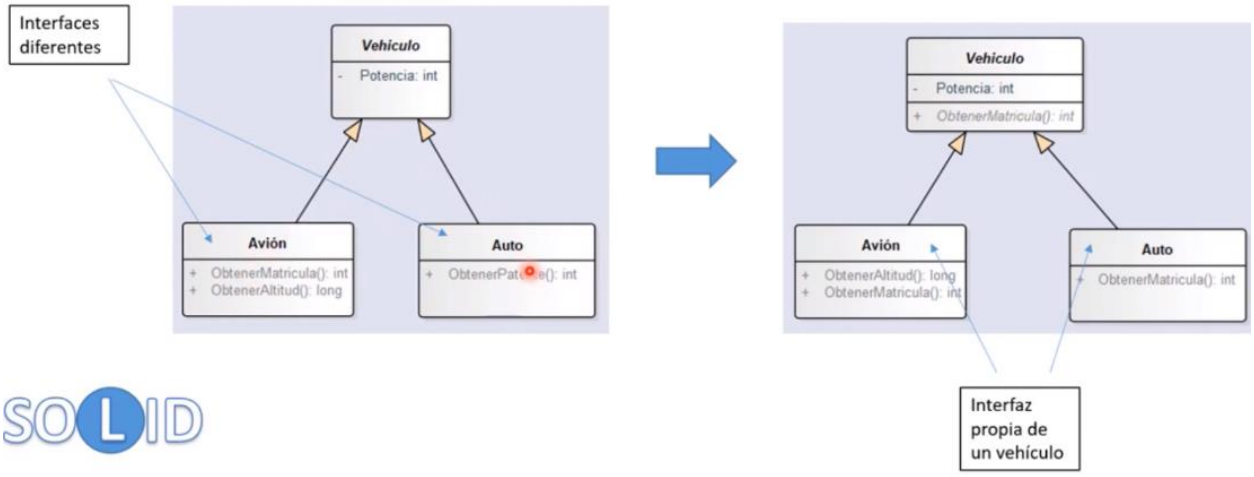
La clase cliente tiene dependencia de stack y stack al heredar de List cliente puede ver lo que puede hacer list a travez de stack, esto no debería ser asi ya que entra en conflicto.

La solución a la derecha toda son clases de alto nivel ya que cliente puede usar Stack pero no ve lo que puede hacer List

Ejemplo 2

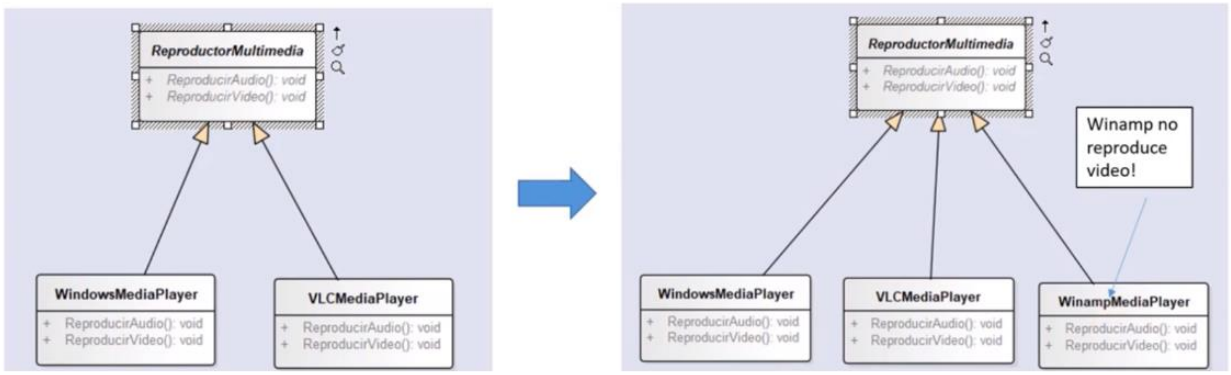
Liskov Substitution Principle – Ejemplo 2

Si tenemos dos objetos de tipos diferentes que derivan de una misma clase base, deberíamos poder reemplazar cada uno de los tipos allí dónde el tipo base esté implementado.

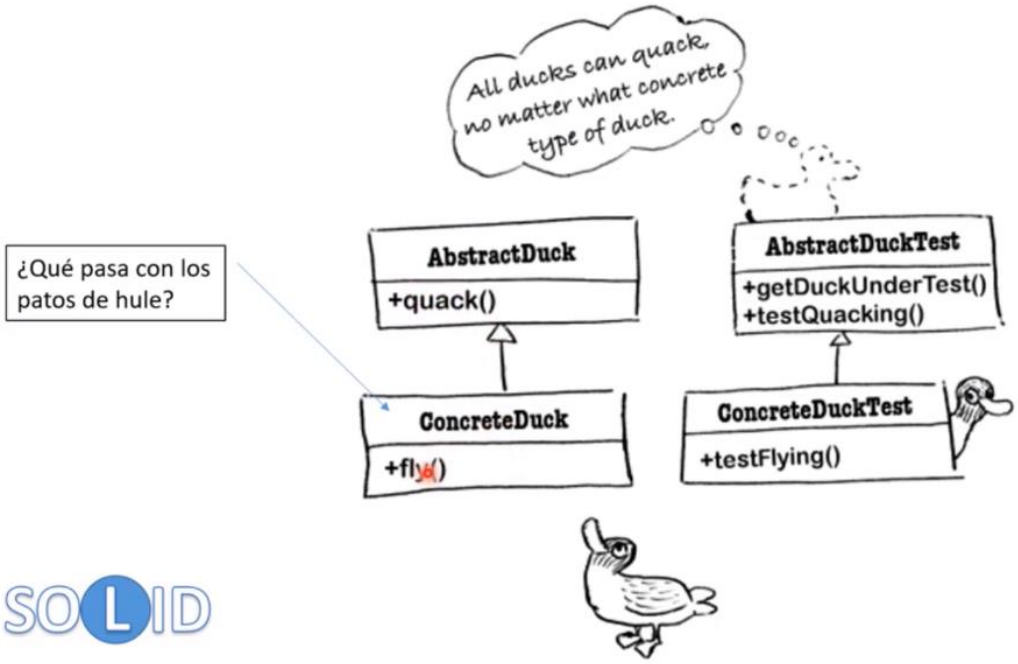


Liskov Substitution Principle – Ejemplo 3

Si tenemos un sistema que reproduce multimedia en cualquier tipo de reproductor, debemos identificar correctamente las abstracciones.

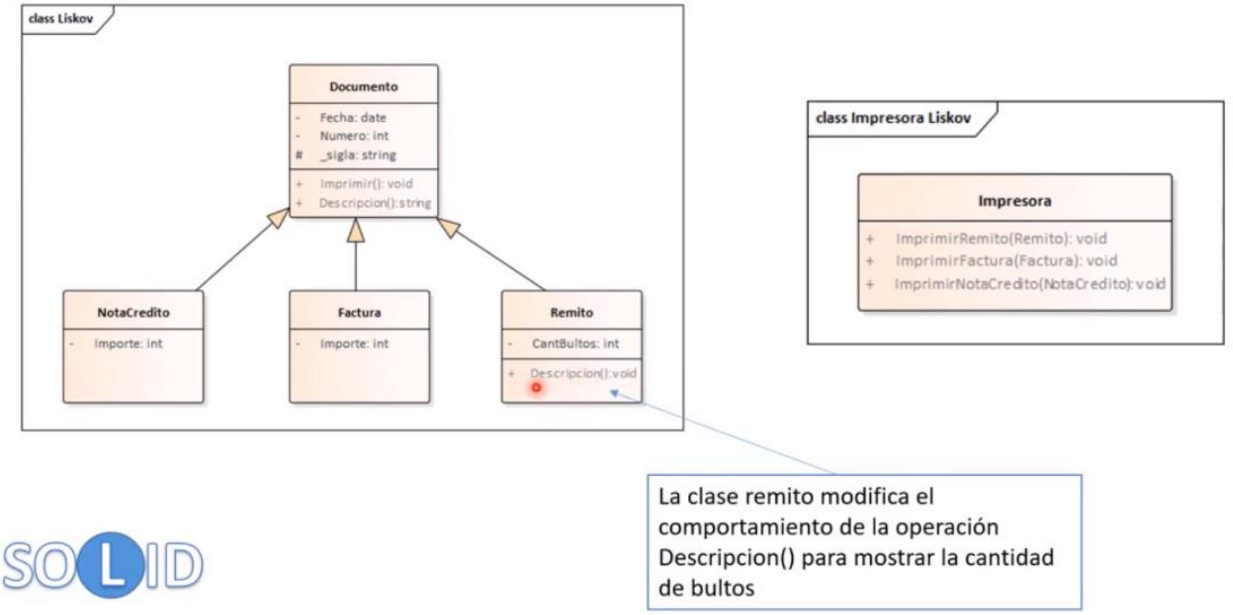


Liskov Substitution Principle – Ejemplo 4



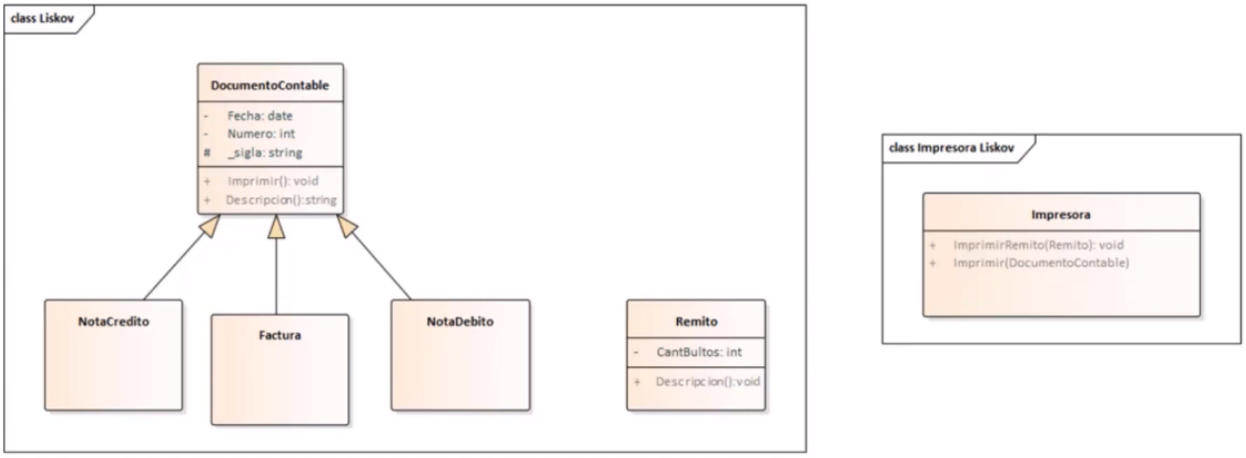
SOLID

Liskov Substitution Principle – Ejemplo 5



Despues del refactor →

Liskov Substitution Principle – Ejemplo 5



SOLID: Interface Segregation Principle

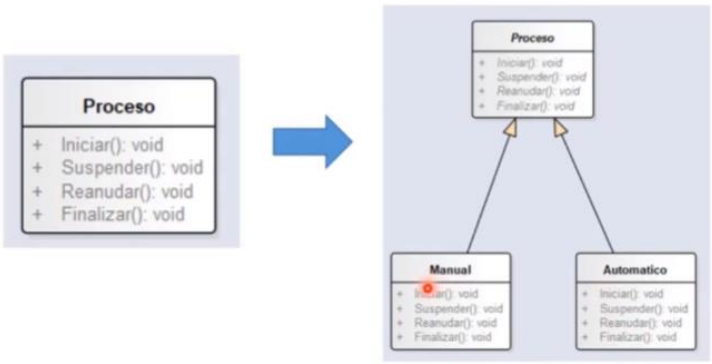
Muchas veces hay clases que implementan interfaces que no usan.

Principios

- Los clientes no deben estar forzados a usar interfaces que no usan
- Guarda la relación con la cohesión de las aplicaciones
- Las clases que implementan una interfaz o una clase abstracta no deberían estar obligadas a utilizar partes que no van a utilizar
- Los clientes no deben estar obligados a implementar y/o depender de una interface que luego no van a usar

Interface Segregation Principle – Ejemplo 1.1

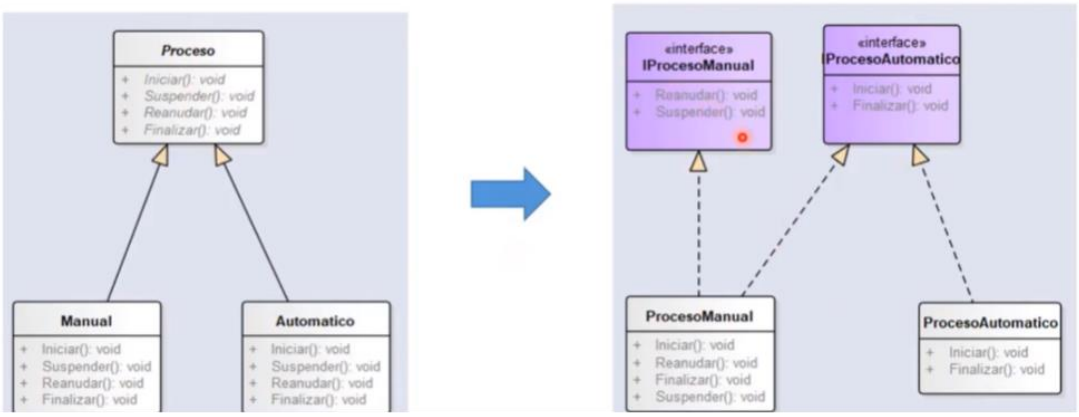
Una empresa tiene, históricamente, procesos que efectúan los operarios. Al llegar a la mañana inician el proceso, al medio día suspender para almorzar, luego lo reanudan para finalizarlo al final de la jornada laboral. Años después deciden adquirir un robot para automatizar ciertos procesos…



El proceso de la izquierda es válido ya que el operario como ser humano para para descansar y luego reanuda.
Sin embargo cuando el proceso es hecho por un robot pasa a ser un proceso automático por lo que el robot no necesita descansar por tanto no necesita Suspender y Reanudar.
Refactoring:

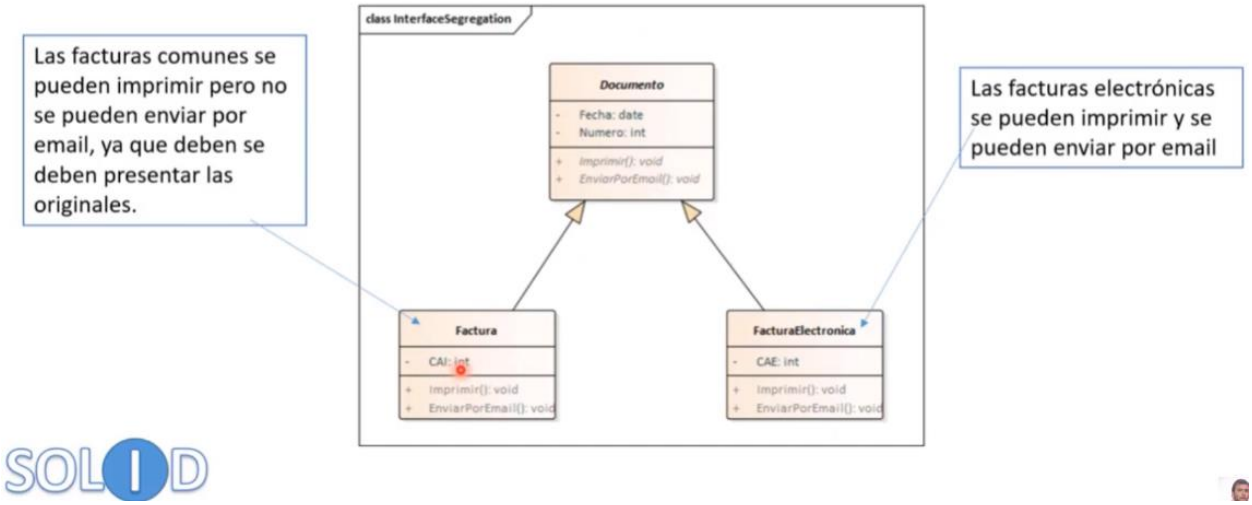
Interface Segregation Principle – Ejemplo 1.1

El inconveniente surge por que el Robot no necesita para para almorzar...

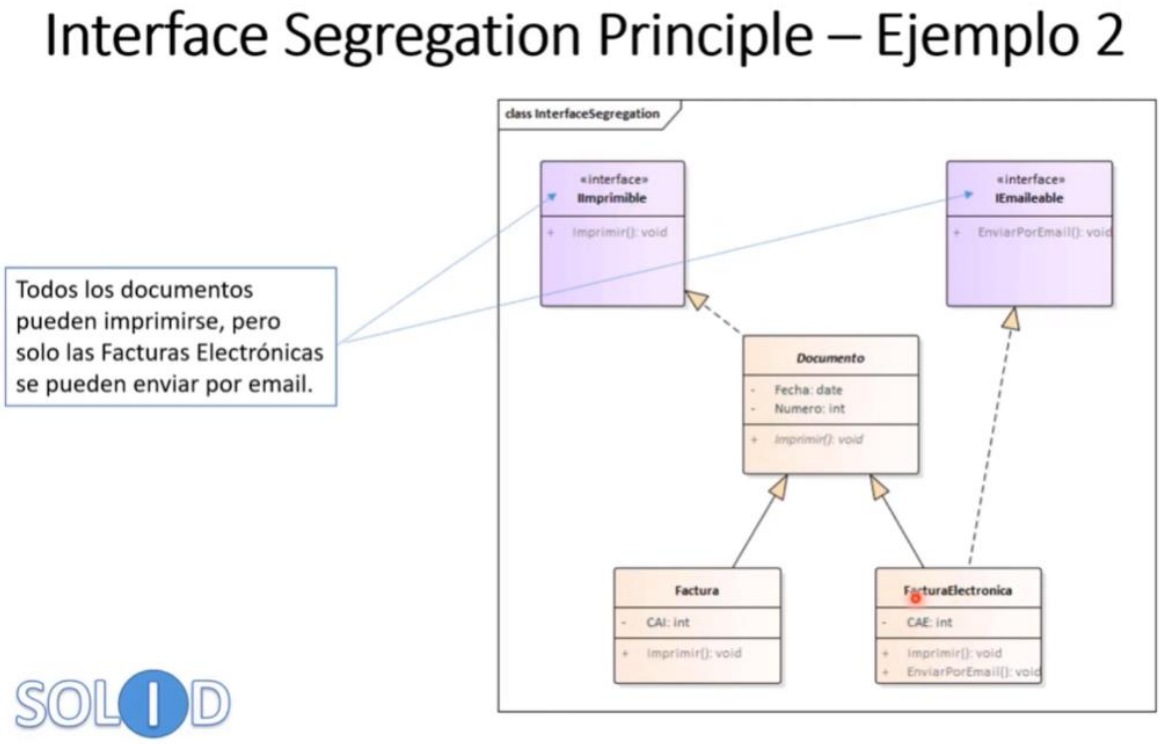


Después del refactor, segregamos las funcionalidades del proceso en dos interfaces, IprocesoManual e IProcesoAutomatico, así para un Proceso manual hecho por una persona implementara las dos interfaces y para el proceso automático del robot solo implementa la interfaz automática.
Asi cumplimos con la segregación de interfaces, ya que no obligamos a implementar operaciones que no serán utilizadas.

Interface Segregation Principle – Ejemplo 2



Solucion:



Ejemplo en código
Solid.Interfaces

SOLID inversión de dependencia

Se utilizan para desacoplar módulos de software

Las clases de alto nivel no deberían depender de las clases de bajo nivel. Ambas deberían depender de las abstracciones

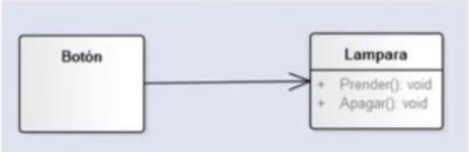
Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones.

Al diseñar la interacción entre el módulo de alto nivel y uno de bajo nivel, la interacción debe considerarse como una interacción abstracta entre ellos.

Ejemplo 1.1

Dependency Inversión Principle – Ejemplo 1.1

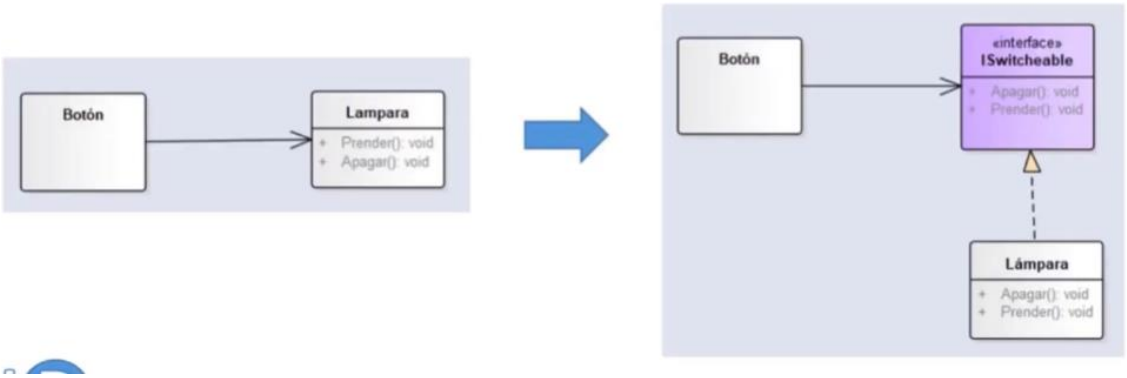
Un botón es capaz de detectar si el usuario desea prender o apagar una lámpara. Una vez que lo detecta, procede a apagarla o prenderla según corresponda.



Ejemplo 1.2

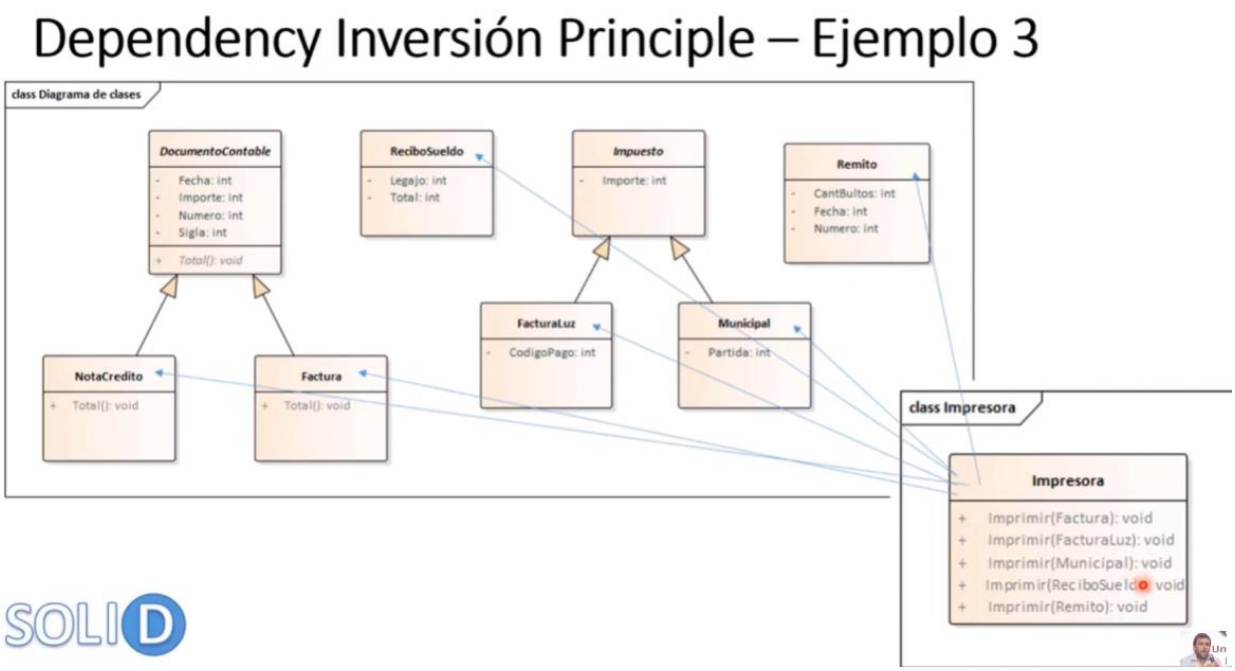
Dependency Inversión Principle – Ejemplo 1.2

- No se puede reusar el botón ya que depende exclusivamente de la lámpara, aunque podría tener otros usos.
- El botón no debería depender de la lámpara, sino de una abstracción.

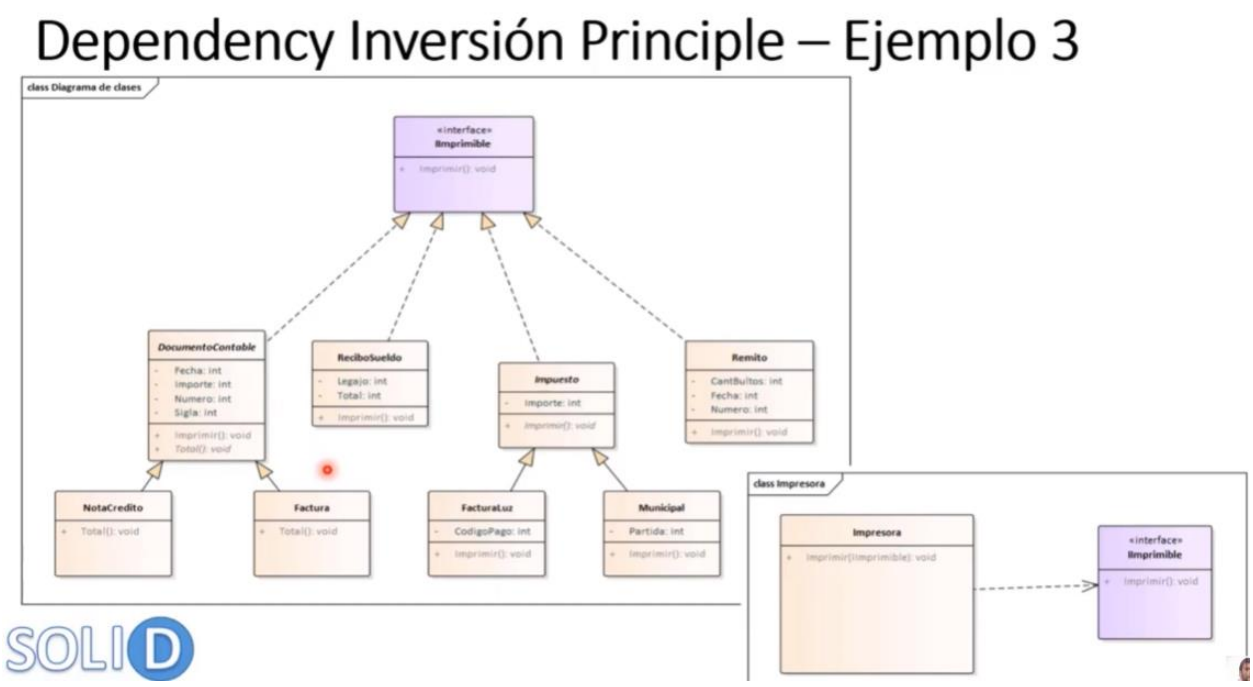


Con esta abstracción podríamos usar la interfaz no solo en una lámpara sino también en un ventilador.

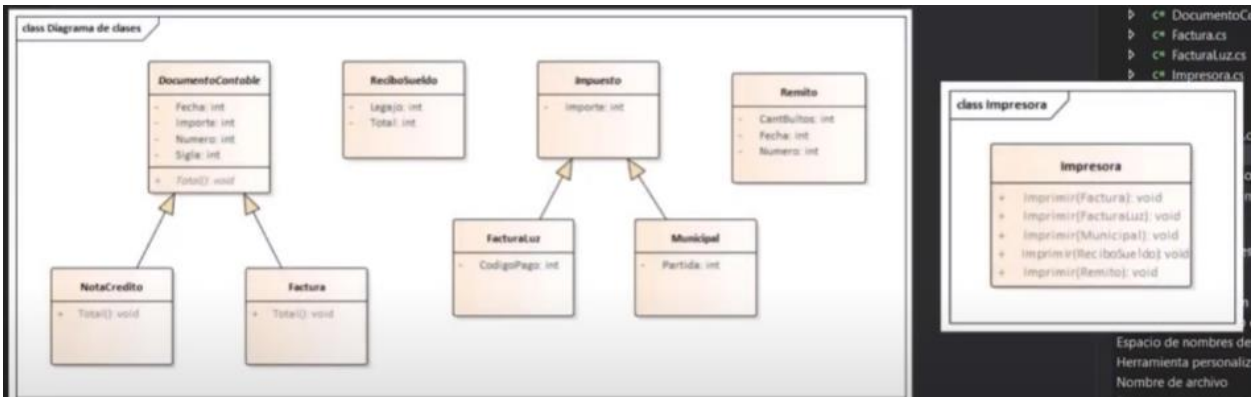
Ejemplo 3



Ejemplo 3 Refactoring



Ejemplo 4



Refactoring en código