[DevExperto - Formación Android & Kotlin](https://www.youtube.com/@devexperto)

Inversión de dependencias

https://www.youtube.com/watch?v=90uW1\_l0Y3I

Gracias a este principio se puede hacer que el código, el núcleo de la aplicación, no tenga que depender de factores externos, como detalles de implementación: el framework utilizado, la BBDD a la que te conectes, el servidor que se haya utilizado.

Todos estos aspectos se definen mediante interfaces.

En una modelo tradicional, cuando un objeto depende de otro, simplemente se crea una nueva instancia del objeto dentro del otro y se lo utiliza. Esto puede traer los problemas mencionados arriba. Además, no quedan claras las dependencias. A la vez, es muy complicado hacer tests.

Cómo darse cuenta si se viola este principio:

Cualquier instanciación de clases complejas dentro de otra clase.

Ejemplo de todo lo que está mal

Luego de elegir los productos en la clase shopping, se le pasan por parámetro a Shopping Canasta. En esa clase se crean instancias de SqlDatabase y de TarjetaDeCredito y se les pasa por parámetro la data.

Dice que ShoppingCanasta sería una clase de alto nivel, que depende de TarjetaDeCredito y SqlDatabase, que son de bajo nivel. Asimismo, las instancias de dichas clases se crean en la misma clase ShoppingCanasta, lo cual también está mal (reitera que eso dificulta el testeo).

public class Shopping

{

}

public class ShoppingCanasta

{

public void Comprar(Shopping shopping)

{

var db = new SqlDatabase();

db.Save(shopping);

var creditCard = new TarjetaDeCredito();

creditCard.Pagar(shopping);

}

}

public class SqlDatabase

{

public void Save (Shopping shopping)

{

}

}

public class TarjetaDeCredito

{

public void Pagar (Shopping shopping)

{

}

}

Ejemplo aplicando inversión de dependencias

Primero crea una interfaz para la determinar la forma de pago y otra para determinar la forma en que la información se guardará. Las clases que implementan esas interfaces, deben cumplir con dicho contrato. Se ve ahí cuan fácil es agregar otra forma persistir la información, basta con hacer que una nueva clase cumpla con dicho contrato.

Además, las instancias de dichas clases no se crean en la clase ShoppingCanasta. Ella, las recibe por constructor y después ejecuta sus métodos, independientemente de la forma de pago o de la persistencia.

En este ejemplo las instancias se crean en la clase main (con un ejemplo concreto debería hacerse desde Shopping).

namespace DependencyInversion

{

class Program

{

static void Main(string[] args)

{

ShoppingCanasta SC = new ShoppingCanasta(new DocTxT(), new TarjetaDeCredito());

SC.Comprar(new Shopping());

Console.WriteLine("-----------------------------------\n");

ShoppingCanasta SC2 = new ShoppingCanasta(new SqlDatabase(), new TarjetaDeCredito());

SC2.Comprar(new Shopping());

Console.ReadKey();

}

}

public class Shopping

{

}

public class ShoppingCanasta

{

private IPersistencia persistencia;

private IFormaDePago formaDePago;

public ShoppingCanasta(IPersistencia persistencia, IFormaDePago formaDePago)

{

this.formaDePago = formaDePago;

this.persistencia = persistencia;

}

public void Comprar(Shopping shopping)

{

persistencia.Save(shopping);

formaDePago.Pagar(shopping);

}

}

public interface IPersistencia

{

void Save(Shopping shopping);

void InformaPersistencia();

}

public class DocTxT : IPersistencia

{

public void Save(Shopping shopping)

{

InformaPersistencia();

}

public void InformaPersistencia()

{

Console.WriteLine("Se está 'persistiendo' desde un Doc.Txt");

}

}

public class SqlDatabase:IPersistencia

{

public void Save (Shopping shopping)

{

InformaPersistencia();

}

public void InformaPersistencia()

{

Console.WriteLine("Se está 'persistiendo' desde una BBDD Sql");

}

}

public interface IFormaDePago

{

void Pagar(Shopping shopping);

void InformaMedioDePago();

}

public class TarjetaDeCredito:IFormaDePago

{

public void Pagar (Shopping shopping)

{

InformaMedioDePago();

}

public void InformaMedioDePago()

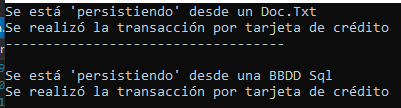
{

Console.WriteLine("Se realizó la transacción por tarjeta de crédito");

}

}

}



Segregación de interfaces

El problema es cuando las interfaces intentan definir mas cosas de las que necesitan. Muchas de las clases hijas no sabrán que hacer con gran parte de los métodos que deben implementar.

¿Cómo darse cuenta cuándo se viola este principio?

Cuando al implementar una interfaz uno o varios métodos no tienen sentido, y hay que dejarlos vacíos y que lancen excepciones.

Lo ideal es dividir esa interfaz en varias, para poder utilizar todos los métodos definidos en las mismas.

Sustitución de Liskov

Las clases hijas deben poder sustituir a la padre, y que el programa siga funcionando.

Si un método sobrescrito no hace nada o lanza una excepción, muy probablemente estemos violando este principio.

Open / Closed

Se suele resolver usando polimorfismo.

¿Cómo detectar si se viola este principio? Si cada vez que hay algo nuevo, se tienen que modificar las mismas viejas clases.

Dice que no siempre es recomendable, ya que complejiza bastante y no siempre es necesario hacerlo (cuando un área del código no cambia demasiado).

Principio de responsabilidad única

Una clase debe tener una única responsabilidad (=razón para cambiar).