**La inversión de dependencias es un principio**que describe un conjunto de técnicas destinadas a**disminuir el acoplamiento entre los componentes de una aplicación.**

El Principio de Inversión de Dependencias propone evitar las dependencias rígidas entre componentes mediante las siguientes técnicas:

* Utilizar abstracciones (interfaces) en lugar de referencias directas entre clases, lo que facilita que podamos reemplazar componentes con suma facilidad.
* Hacer que una clase reciba referencias a los componentes que necesite para funcionar, en lugar de permitir que sea ella misma quien los instancie de forma directa o a través de factorías.

 Observa el siguiente ejemplo, escrito en C#,aún sin usar inyección de dependencias, donde se muestra una clase InvoiceServices cuyo funcionamiento depende, como mínimo, de otras dos clases externas, InvoiceRepository e EmailNotifier:

public class InvoiceServices

{

...

public void Remove(int invoiceId)

{

using (var invoiceRepository = new InvoiceRepository())

{

var removed = invoiceRepository.Remove(invoiceId);

if (removed)

{

var notifier = new EmailNotifier();

notifier.NotifyAdmin($"Invoice {invoiceId} removed");

}

}

}

}

El código del método Remove(), aunque aparentemente correcto, presenta algunos **problemas**:

1. Tiene **bastantes líneas de código de "fontanería"**, dedicadas a instanciar y preparar las dependencias, en lugar de centrarse en su misión, que es eliminar una factura y notificar al administrador.
2. Observa además que, **muchas de esas líneas deberían repetirse en otros métodos de la clase** que requirieran los servicios de estos componentes. Por ejemplo, otros métodos, como Add() o Update(), probablemente necesitarían acceder al repositorio de facturas y quizás también al componente de notificación.
3. **Estamos atando inexorablemente la implementación** de InvoiceServices a InvoiceRepository e EmailNotifier. Cualquier modificación en estas últimas podría afectar a la primera, o incluso requerir cambios en ésta.
4. **Complicamos la reutilización de la clase**, puesto que siempre va a ir unida a los componentes de los que depende.
5. **No quedan claras las dependencias de la clase**. Para conocerlas deberíamos leer todo su código y ver qué clases externas utiliza. Aunque en ese ejemplo no es un problema porque es poco código, en clases más extensas sí sería bastante complicado determinarlas.
6. **Dificultamos la realización de**[pruebas unitarias](https://www.campusmvp.es/recursos/post/que-tipos-de-pruebas-de-software-son-habituales-para-un-desarrollador.aspx), puesto que no hay forma de probar únicamente el correcto funcionamiento del método Remove() de InvoiceServices sin probar al mismo tiempo el funcionamiento de las clases de las que depende.

Utilizando el principio de Inyección de Dependencias, el código anterior podríamos transformarlo en el siguiente:

public class InvoiceServices: IInvoiceServices

{

private readonly IInvoiceRepository \_invoiceRepository;

private readonly INotifier \_notifier;

public InvoiceServices (IInvoiceRepository invoiceRepository, INotifier notifier)

{

\_invoiceRepository = invoiceRepository;

\_notifier = notifier;

}

...

public void Remove(int invoiceId)

{

var removed = \_invoiceRepository.Remove(invoiceId);

if (removed)

{

\_notifier.NotifyAdmin($"Invoice {invoiceId} removed");

}

}

}

En la práctica podemos **utilizar un sistema de inyección de dependencias**. **Un sistema de inyección de dependencias es el encargado de instanciar las clases que necesitemos y suministrarnos ("inyectar") las dependencias** enviando los parámetros oportunos al constructor.

De esta forma conseguiremos las siguientes ventajas:

1. La implementación de InvoiceServices queda totalmente desacoplada, puesto que no depende de ningún componente específico para funcionar, sólo de contratos.
2. Para conocer las dependencias de la clase basta con echar un vistazo a su constructor o a las propiedades decoradas con el atributo usado por el marco de trabajo para marcar los miembros inyectables, por lo que simplificamos su lectura y facilitamos su comprensión.
3. Los métodos pueden centrarse ahora en lograr su cometido porque las dependencias ya están disponibles a nivel de instancia. Esto nos lleva a disponer de un código más conciso, limpio, fácil de escribir y de leer. Observa que puedes entender el método Remove() del segundo ejemplo de un rápido vistazo, mientras que en el primer ejemplo necesitabas una lectura algo más detenida.
4. La clase será mucho más reutilizable porque no depende de otros componentes, sino de abstracciones.
5. Podemos realizar fácilmente pruebas unitarias de esta clase de forma aislada, enviándole dependencias falsas o controladas (fakes, stubs, mocks...) desde los métodos de test.