Programación II
Inyección de dependencias

24 de junio de 2025

Imaginemos a una máquina de café. Para funcionar, necesita granos, agua caliente y una taza. Si la propia máquina tiene dentro de sí el depósito de agua y los granos, cada vez que se requiera otro tipo de café, tendríamos que abrir la máquina y cambiar piezas internas.

En cambio, si la máquina recibe el agua desde una tubería externa y los granos desde un contenedor aparte, nosotros solo conectariamos lo que necesite: agua filtrada o granos de café a elección. Así, la misma máquina sirve para muchos cafés, sin tener que desarmarla.

En programación, la máquina es tu clase, y los ingredientes (granos, agua) son sus dependencias: otros objetos o servicios que necesita para trabajar. Si la clase crea sus propias dependencias, queda atada a implementaciones específicas.

Si en lugar de eso las recibe del exterior, vos decidís en cada proyecto qué “ingredientes” inyectarle, sin cambiar la clase. Esto hace tu código mucho más flexible, fácil de probar (podes “inyectar” sustitutos ligeros en tests) y mantenible.

Teoría de Inyección de Dependencias (DI)

Inversión de Control (IoC):

El principio clave es: **no dejes que tu clase “mande” sobre sus dependencias**, sino que sea el código externo quien las “pase” dentro.

- Antes: la clase va y busca o crea lo que necesita.
- Ahora: la clase **espera** que le den lo que necesita.

Por eso hablamos de **Inversión de Control**: el control de la creación y suministro de objetos se invierte y sale de la clase.

Principio de Dependencias (DIP):

Dentro de SOLID, el DIP dice dos cosas:

1. **Las clases de alto nivel** (las que contienen la lógica de tu aplicación) no deben depender de clases de bajo nivel (implementaciones concretas). Ambos deben depender de **abstracciones** (interfaces).
2. Las **abstracciones** no deben depender de detalles; **los detalles** deben depender de **las abstracciones**.

En la práctica: tus clases hablan siempre con **interfaces**, nunca con clases concretas.

¿Cómo hacer inyección de dependencias?

En C#, sin frameworks, solemos usar tres caminos:

1. **Constructor Injection**: La forma más fuerte y segura: si no me das la dependencia, la clase no se crea.

   Ejemplo sencillo:

```
public interface IPrinter
{
    void Print(string message);
}

public class ConsolePrinter : IPrinter
{
    public void Print(string message)
    {
        Console.WriteLine(message);
    }
}

public class ReportGenerator
{
    private readonly IPrinter _printer;

    public ReportGenerator(IPrinter printer)
    {
        // Validación: nunca nulo
```

```
        _printer = printer ?? throw new
ArgumentNullException(nameof(printer));
    }

    public void Generate()
    {
        _printer.Print(" --- Reporte Generado --- ");
    }
}

// En tu Main:
var printer = new ConsolePrinter();
var report = new ReportGenerator(printer);
report.Generate();
```

2. **Setter (Property) Injection:** Ideal si la dependencia es opcional o se puede cambiar en tiempo de ejecución.

   Ejemplo:

```
public class AlertService
{
    // Propiedad pública que alguien "setea" después
    public IPrinter Printer { get; set; }

    public void Alert(string text)
    {
        if (Printer == null)
            Console.WriteLine("¡Cuidado! No hay impresora configurada.");
        else
            Printer.Print($"[ALERTA] {text}");
    }
}

// En Main:
var alert = new AlertService();
alert.Alert("Sistema iniciado");    // aún sin Printer
alert.Printer = new ConsolePrinter();
alert.Alert("Error crítico");       // ahora imprime
```

3. **Interface Injection:** Menos común en C#, consiste en que la propia interfaz expone un método para inyectar la dependencia.
   Ejemplo:

```
public interface IConfigurable<T>
{
    void Configure(T dependency);
}

public class Logger : IConfigurable<IPrinter>
{
    private IPrinter _printer;
    public void Configure(IPrinter printer)
    {
        _printer = printer;
    }

    public void Log(string msg)
    {
        _printer?.Print("[LOG] " + msg);
    }
}
```

¿Por qué usar Inyección de dependencias?

- **Desacoplamiento**: tu clase solo conoce la interfaz, no la implementación.
- **Testabilidad**: en tests puedes inyectar un "falso" `IPrinter` que recoja mensajes en memoria.
- **Reemplazo fácil**: para otro entorno o tecnología, basta con cambiar la instancia que pasas en el arranque.

Aplicando la DI en C#

Empecemos por crear dos servicios de ejemplo:

```
public interface IMessageService
{
    void Send(string recipient, string body);
}
```

```
public class EmailService : IMessageService
{
    public void Send(string recipient, string body)
    {
        Console.WriteLine($"[Email] A: {recipient} -> \"{body}\"");
    }
}

public class SmsService : IMessageService
{
    public void Send(string recipient, string body)
    {
        Console.WriteLine($"[SMS] A: {recipient} -> \"{body}\"");
    }
}
```

Creamos la clase que va a utilizar estas dependencias:

```
public class NotificationManager
{
    private readonly IMessageService _service;

    // Constructor Injection
    public NotificationManager(IMessageService service)
    {
        _service = service ?? throw new
ArgumentNullException(nameof(service));
    }

    public void Notify(string user, string message)
    {
        // Lógica adicional antes de enviar...
        _service.Send(user, message);
    }
}
```

En la composición, o main:

```
class Program
{
    static void Main(string[] args)
    {
        // Aquí elegimos la implementación:
        IMessageService service = new EmailService();
        // IMessageService service = new SmsService();

        // Inyectamos en nuestro manager
        var notifier = new NotificationManager(service);

        // ¡Listo para usar!
        notifier.Notify("juan@correo.com", "¡Bienvenido al sistema!");
    }
}
```

¿Qué pasaría si mañana quiero usar SMS?
Sólo se cambia la asignación en `Main()`. El resto del código ni se entera.

Me perdí... ¿En la composición?
La **composición** es el punto único en tu programa donde creas y conectas las implementaciones concretas de tus códigos (por ejemplo, en `Main()`), de modo que el resto del proyecto solo recibe esas llamadas o dependencias sin saber cómo se construyen.

Pero no nos quedemos en el medio, vamos a ver cómo podemos mejorar esto:

```
public class DynamicNotifier
{
    public IMessageService Service { get; set; }

    public void Alert(string msg)
    {
```

```
        if (Service == null)
        {
            Console.WriteLine("[ALERT] Sin servicio configurado: " + msg);
        }
        else
        {
            Service.Send("soporte@empresa.com", msg);
        }
    }
}

// En Main:
var dynamic = new DynamicNotifier();
dynamic.Alert("Arrancando");           // mensaje sin servicio
dynamic.Service = new SmsService();
dynamic.Alert("Error crítico");        // ahora envía SMS
```

La clase `DynamicNotifier` ilustra el uso de **Setter Injection**: en lugar de recibir su dependencia por el constructor, expone una propiedad pública (`Service`) que puede asignarse en cualquier momento.. Esto permite cambiar o retrasar la configuración del servicio de notificaciones durante la ejecución.

¿Cuándo usar entonces cada tipo de inyección?

- **Constructor Injection** para dependencias **obligatorias**.
- **Setter Injection** para dependencias **opcionales** o configurables en el ciclo de vida.
- **Interface Injection** en casos especiales.

¡A practicar!

Ejercicio 1: Constructor Injection Básico

1. **Objetivo**
   – Definir una interfaz `ILogger` y dos implementaciones: `ConsoleLogger` (usa

`Console.WriteLine`) y `FileLogger` (simula escribiendo en archivo).

2. **Pasos**

    - Crea la interfaz `ILogger` con método `void Log(string message)`.
    - Implementa `ConsoleLogger` y `FileLogger`.
    - Crea una clase `AppService` que reciba obligatoriamente un `ILogger` por constructor y en su método `Run()` escriba un mensaje de bienvenida usando `Log`.
    - En tu `Main()`, inyecta `ConsoleLogger` y llama a `Run()`.

3. **Chequeo**
   – Al ejecutar deberías ver la salida en consola. Cambia en `Main()` a `FileLogger` y confirma que el texto “grabado” en archivo se “simula” en la salida.

Ejercicio 2: Setter Injection

1. **Objetivo**
   – Practicar la inyección a través de una propiedad para dependencias opcionales.
2. **Pasos**

    - A partir de `AppService` del ejercicio anterior, añade una propiedad pública `ILogger OptionalLogger { get; set; }`.
    - En `Run()`, después de escribir el mensaje principal, si `OptionalLogger != null` escribe un mensaje extra (“Modo opcional activo”).
    - En `Main()`,
        1. Primero crea y ejecuta `AppService` **sin** asignar `OptionalLogger`.
        2. Luego vuelve a crear, asigna `OptionalLogger = new ConsoleLogger()`, y vuelve a ejecutar.

3. **Chequeo**
   – Deberías ver la segunda línea de log solo en la segunda ejecución.

Ejercicio 3: Interface Injection

1. **Objetivo**
   – Experimentar la inyección vía método de configuración.

2. **Pasos**

   - Define `IConfigurable<T>` con `void Configure(T dep)`.
   - Crea una clase `ReportGenerator` que implemente `IConfigurable<ILogger>`.
   - En `Configure`, guarda el `ILogger` interno; en `GenerateReport()`, usa el logger para escribir “Reporte listo”.
   - En `Main()`,
     1. Instancia `ReportGenerator`.
     2. Llama a `GenerateReport()` (antes de configurar).
     3. Llama a `Configure(new ConsoleLogger())`.
     4. Vuelve a llamar a `GenerateReport()`.

3. **Chequeo**
   – Deberías ver que la primera llamada no imprime nada (o muestra un aviso opcional) y la segunda sí.

Ejercicio 4: Punto de Composición Central

1. **Objetivo**
   – Refuerza el concepto de “composition root”.

2. **Pasos**

   - Crea una nueva clase estática `CompositionRoot` con un método `public static NotificationManager Build()`.
   - Dentro de `Build()`, decide si usar `EmailService` o `SmsService` (por ejemplo, según un valor booleano).
   - `Build()` retorna un `NotificationManager` con la dependencia inyectada.
   - En `Main()`, reemplaza la construcción directa con `var nm = CompositionRoot.Build(); nm.Notify(...);`.

3. **Chequeo**
   – Cambiando sólo `CompositionRoot`, la lógica de notificaciones (cliente) no debe tocarse.

Ejercicio 5: DI y Pruebas Unitarias

1. **Objetivo**
   – Ver cómo DI facilita el testing.

2. **Pasos**

   - Usa tu clase `AppService` de ejercicios anteriores.
   - Crea un **mock simple** de `ILogger` implementando la interfaz y capturando los mensajes en una lista interna.
   - En un proyecto de test (o en un método `Main()` extra), inyecta ese mock y ejecuta `Run()`.

- Comprueba que la lista de mensajes contiene exactamente el texto esperado.

3. **Chequeo**

   – Si la lista de tu mock no coincide, revisa la inyección y la llamada a `Log`.

¿Mock?
Un **mock** es un objeto “falso” o simulado que implementa la misma interfaz que una clase real, pero cuya lógica interna simplemente registra las llamadas (o devuelve valores predefinidos) en lugar de ejecutar comportamiento real.