

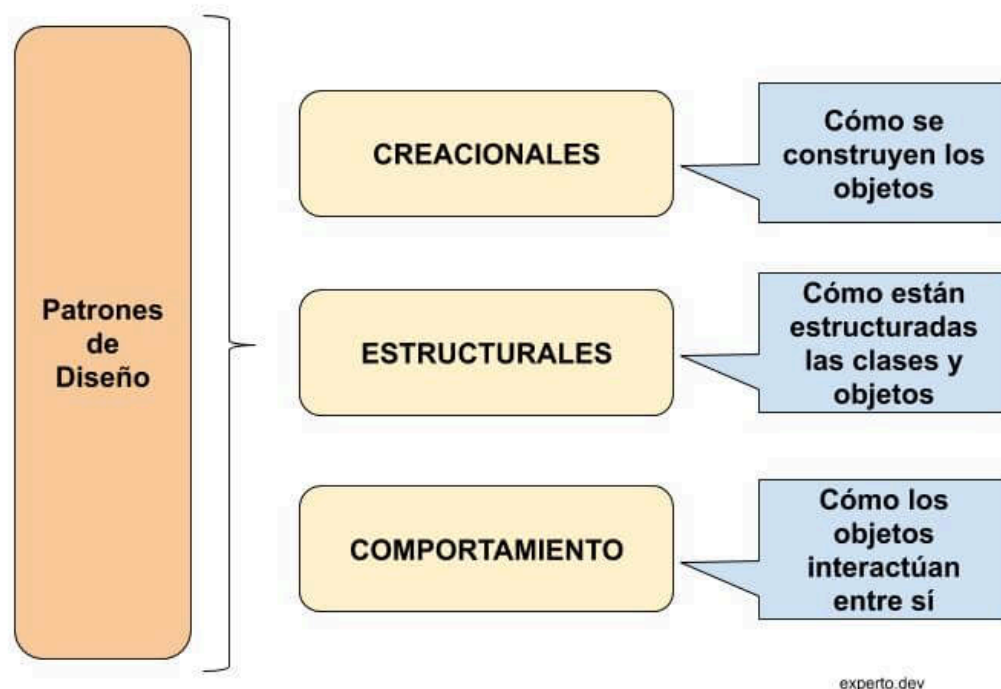
Patrones de diseño

Los patrones de diseño son técnicas de creación de código que evita que tengas que reinventar la rueda. Muchas problemáticas a la hora de programar ya fueron resueltas y esto es en realidad una buena noticia. Puede ayudarte a **agilizar tiempos** utilizando técnicas sumamente probadas y que le darán **estabilidad** y **confianza** a tu código.

¿De dónde salen? ¿Quién los inventó? Quizá a esta altura empieces a notar un patrón, ya que al igual que los principios SOLID: los patrones de diseño salen de un libro. En 1994, la “banda de los cuatro” (Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides), largan el libro “*Pattern Desings*” introduciendo 23 patrones de diseño que han sido referentes en el mundo del software. Además de este libro, también te recomiendo “*Head First Design Patterns*” que podes descargarlo desde [acá](#), aunque lamentablemente está en inglés.

Pero como siempre, en la programación la única manera de aprender es **practicando**. No existe libro mágico, ni teoría milagrosa, ni herramienta salvadora que pueda enseñarte y convertirte en un buen programador.

Para entender mejor qué son los patrones vamos a definirlos primero en tres categorías:



- Los patrones **creacionales** proporcionan mecanismos de creación de objetos que incrementan la flexibilidad y la reutilización de código existente.
- Los patrones **estructurales** explican cómo ensamblar objetos y clases en estructuras más grandes a la vez que se mantiene la flexibilidad y eficiencia de la estructura.
- Los patrones de **comportamiento** se encargan de una comunicación efectiva y la asignación de responsabilidades entre objetos.

Dentro de cada categoría existen diferentes patrones que son utilizados por su fama y aceptación entre la comunidad informática. Nosotros no seremos la excepción, vamos a ver que son cada uno y cómo se utilizan.

Lamentablemente no existe una enciclopedia donde puedas **buscar el problema que tengas y que automáticamente te recomiende un patrón** como solución. La experiencia te va a dar una visión donde empieces a encontrar similitudes en diferentes problemas, pero los mismos pueden ser muy variados y no podemos acotarlos en una búsqueda mecánica. Además,

los patrones no buscan solucionar bugs directamente, sino que plantean estructuras base o implementaciones “*from scratch*” que van a **evitar que estos errores** surjan. Si existen algunos específicos que intentan resolver casos puntuales, pero es imperativo que aprendas a incorporarlos desde el minuto uno.

Aprendamos ahora sobre los patrones **creacionales**, vamos a enfocarnos en tres:

1. **Abstract Factory:** Vas a crear una especie de “fábrica” general para crear grupos de objetos relacionados (productos) sin que tu código principal tenga que saber exactamente qué clases específicas está creando. Básicamente, le pedís los objetos a esta fábrica, en lugar de crearlos directamente vos y si en algún momento necesitas cambiar todas las variables, con factory es cuestión de una sola línea.
 - a. ¿Cuándo usar? Cuando necesitas variantes de productos de la misma “familia”. Ejemplo: “Windows”, “Mac”, “Android”, “Linux” (sistemas operativos)

```
using System;
// Necesitamos System para usar Console.WriteLine.

// -----
// ♦ Interfaces de productos (contratos comunes)
// -----

// Interface para el "Botón"
public interface IButton
{
    void Render(); // Método para dibujar o renderizar el
    botón
}

// Interface para el "Checkbox"
public interface ICheckbox
{
```

```
        void Toggle(); // Método para cambiar el
estado (marcar/desmarcar)
    }

    // -----
    // ♦ Productos concretos: versión Windows
    // -----
    public class WinButton : IButton
    {
        // Implementa Render de manera específica para Windows
        public void Render() => Console.WriteLine("Render
WinButton");
    }

    public class WinCheckbox : ICheckbox
    {
        // Implementa Toggle de manera específica para Windows
        public void Toggle() => Console.WriteLine("Toggle
WinCheckbox");
    }

    // -----
    // ♦ Productos concretos: versión Mac
    // -----
    public class MacButton : IButton
    {
        public void Render() => Console.WriteLine("Render
MacButton");
    }

    public class MacCheckbox : ICheckbox
    {
        public void Toggle() => Console.WriteLine("Toggle
MacCheckbox");
    }

    // -----
    // ♦ Fábrica abstracta
```

```
// -----  
// Define los métodos que todas las fábricas concretas deben tener  
// (qué productos deben saber crear).  
public interface IGuiFactory  
{  
    IButton CreateButton();    // Crear botón  
    ICheckbox CreateCheckbox(); // Crear checkbox  
}  
  
// -----  
// ♦ Fábricas concretas  
// -----  
  
// Fábrica de elementos estilo Windows  
public class WinFactory : IGuiFactory  
{  
    public IButton CreateButton() => new WinButton();  
    public ICheckbox CreateCheckbox() => new WinCheckbox();  
}  
  
// Fábrica de elementos estilo Mac  
public class MacFactory : IGuiFactory  
{  
    public IButton CreateButton() => new MacButton();  
    public ICheckbox CreateCheckbox() => new MacCheckbox();  
}  
  
// -----  
// ♦ Cliente (Application)  
// -----  
// La clase Application NO sabe qué tipo de botón/checkbox está usando.  
// Solo conoce las interfaces IButton e ICheckbox.  
// La fábrica concreta se pasa por parámetro al construir la app.  
public class Application  
{
```

```
    private readonly IButton _button;        //
Referencia al botón
    private readonly ICheckbox _checkbox; // Referencia al
checkbox

    // El cliente recibe una fábrica concreta, pero trabaja
solo con la interfaz IGuiFactory
    public Application(IGuiFactory factory)
    {
        _button = factory.CreateButton();
        _checkbox = factory.CreateCheckbox();
    }

    // Método que usa los productos sin importar su
implementación real
    public void Run()
    {
        _button.Render();
        _checkbox.Toggle();
    }
}

// -----
// ♦ Demo
// -----
public static class Program
{
    public static void Main()
    {
        // Elegimos qué fábrica usar (WinFactory o MacFactory)
        IGuiFactory factory = new WinFactory();
        // Cambiá por 'new MacFactory()' y verás que la app
usa productos estilo Mac.


        var app = new Application(factory);
        app.Run(); // Ejecuta y muestra mensajes de productos
según la fábrica elegida.
    }
}
```

```
}
```

2. Builder: Separa la construcción paso a paso de un objeto complejo de su representación final. El mismo proceso de construcción puede crear variantes (distintas configuraciones) sin una “telaraña” de constructores.

- a. ¿Cuándo usar? Cuando tengas objetos con muchos parámetros opcionales o combinaciones o cuando tengas que **reutilizar** el proceso para producir variantes (p. ej., “Básico”, “Premium”).

```
using System;
using System.Text;

// =====
//  Producto complejo: Informe
// =====
// Este es el "objeto final" que queremos construir de forma
// prolija.
// Tiene varias partes opcionales (Resumen, Anexos, etc.).
public class Informe
{
    // 'internal set' → se pueden setear desde el mismo
    // ensamblado,
    // pero no desde afuera del todo. El Builder vive acá, así
    // que puede.
    public string Titulo    { get; internal set; } = "";
    public string Resumen  { get; internal set; } = "";
    public string Contenido { get; internal set; } = "";
    public string Anexos   { get; internal set; } = "";

    // ToString "lindo": armamos el texto final del informe.
    public override string ToString()
    {
        var sb = new StringBuilder();
        sb.AppendLine($"== {Titulo} ==");
        // Mostramos solo si hay algo escrito (evitamos
        // ruido).
        if (!string.IsNullOrEmpty(Resumen))
```

```

sb.AppendLine($"Resumen: {Resumen}");
    sb.AppendLine(Contenido);
    if (!string.IsNullOrEmpty(Anexos))
sb.AppendLine($"Anexos: {Anexos}");
    return sb.ToString();
}
}

// =====
// 🛠 Builder (contrato)
// =====
// Define el "flujo" para construir el Informe paso a paso.
// Notá que devuelve IInformeBuilder para encadenar llamadas
// (fluent API).
public interface IInformeBuilder
{
    IInformeBuilder ConTitulo(string titulo);
    IInformeBuilder ConResumen(string resumen);
    IInformeBuilder ConContenido(string contenido);
    IInformeBuilder ConAnexos(string anexos);
    Informe Build(); // Cierra la construcción y devuelve el
    producto final.
}

// =====
// 🖋 Implementación concreta del Builder
// =====
// Acá se guarda el estado parcial mientras armamos el
// Informe.
// La idea: evitar constructores monstruosos y poder validar
// al final.
public class InformeTecnicoBuilder : IInformeBuilder
{
    // Cada builder arranca con un Informe vacío.
    private readonly Informe _informe = new();

    public IInformeBuilder ConTitulo(string titulo)
    {
        _informe.Titulo = titulo;
        return this; // devolvemos 'this' para encadenar:
        builder.ConTitulo(...).ConContenido(...)
    }
}

```



```
public IInformeBuilder ConResumen(string
resumen)
{
    _informe.Resumen = resumen;
    return this;
}

public IInformeBuilder ConContenido(string contenido)
{
    _informe.Contenido = contenido;
    return this;
}

public IInformeBuilder ConAnexos(string anexos)
{
    _informe.Anexos = anexos;
    return this;
}

public Informe Build()
{
    // Validaciones finales para que no salga un
    // Frankenstein.
    if (string.IsNullOrEmpty(_informe.Titulo))
        throw new InvalidOperationException("El informe
    debe tener título.");
    if (string.IsNullOrEmpty(_informe.Contenido))
        throw new InvalidOperationException("El informe
    debe tener contenido.");

    // Devolvemos el producto ya listo.
    return _informe;
}

// 💡 Nota didáctica:
// Si quisieras reutilizar el MISMO builder para armar
// OTRO informe,
// conviene agregar un método Reset() o crear un builder
// nuevo.
}

// =====
// 🧑‍🎓 Director (opcional pero va excelente)
// =====
```

```
// El Director define "recetas" armadas: Básico,
Completo, etc.
// Reutiliza el mismo proceso sin que el cliente sepa los
detalles.
public class DirectorDeInformes
{
    public Informe CrearInformeBasico(IInformeBuilder
builder)
        => builder
            .ConTitulo("Informe Básico")
            .ConContenido("Contenido mínimo.")
            .Build();

    public Informe CrearInformeCompleto(IInformeBuilder
builder)
        => builder
            .ConTitulo("Informe Completo")
            .ConResumen("Resumen ejecutivo del informe.")
            .ConContenido("Contenido detallado y secciones
técnicas.")
            .ConAnexos("Gráficos y tablas.")
            .Build();
}

// =====
// 🚀 Demo de uso
// =====
public static class Program
{
    public static void Main()
    {
        // 1) Creamos un builder concreto (podrías tener
otros: InformeComercialBuilder, etc.)
        var builder = new InformeTecnicoBuilder();

        // 2) El Director trae "recetas" listas.
        var director = new DirectorDeInformes();

        // 3) Informe rápido con receta básica
        var basico = director.CrearInformeBasico(builder);
        Console.WriteLine(basico);

        // 4) Informe completo con otro builder (recomendado
```

```
para no mezclar estados)
    var completo = director.CrearInformeCompleto(new
InformeTecnicoBuilder());
    Console.WriteLine(completo);

    // 5) Construcción manual, a tu gusto, sin Director
(sos libre)
    var custom = new InformeTecnicoBuilder()
        .ConTitulo("Informe Custom")
        .ConResumen("Breve overview.")
        .ConContenido("Cuerpo del informe.")
        .Build();

    Console.WriteLine(custom);
}
}
```

- 3. Singleton:** Garantiza que exista una única instancia de una clase y provee un punto global de acceso a ella. Útil para recursos compartidos (configuración, logger, cache simple).
- ¿Cuándo se usa? Cuando hay exactamente un objeto que coordina algo global (p. ej., configuraciones de la app). O simplemente quieres armar un **control centralizado** y acceso simple.

```
using System;
// Usamos System porque ahí vive Lazy<T> y Console.

// 'sealed' = clase sellada: nadie puede heredarla.
// En un Singleton esto ayuda a que no aparezcan "variantes"
raras.
public sealed class Configuracion
{
    // Campo estático y de solo lectura (readonly): existe una
única copia para toda la app.
    // Lazy<Configuracion> = "creación perezosa": no se crea
hasta que alguien la pida.
```

```
// Además, Lazy es thread-safe por defecto, o sea, si vienen varios hilos a la vez,
// igual se crea UNA sola instancia sin que nada se rompa.
private static readonly Lazy<Configuracion> _instancia =
    new Lazy<Configuracion>(() => new Configuracion());
    // Esa lambda "new Configuracion()" es la receta que Lazy ejecuta la PRIMERA vez.

// Punto de acceso global al singleton:
// Instancia llama a _instancia.Value → si aún no está creada, Lazy la crea en ese momento.
public static Configuracion Instancia =>
_instancia.Value;

// Estado de ejemplo que manejaría tu configuración de app.
// 'private set' = desde afuera solo se puede leer; modificar, solo desde dentro de la clase.
public string CadenaConexion { get; private set; } =
"Server=.;Database=App;Trusted_Connection=True;";
public bool ModoDebug { get; private set; } = false;

// Constructor privado: nadie puede hacer 'new Configuracion()' desde afuera.
// Solo la propia clase (vía Lazy) puede crear la instancia.
private Configuracion()
{
    // Acá podrías cargar config desde un JSON, variables de entorno, etc.
    // Por ahora, dejamos valores por defecto.
}

// Métodos públicos para cambiar el estado de forma controlada (API clara y segura).
public void ActivarDebug() => ModoDebug = true;
public void EstablecerConexion(string cadena) =>
CadenaConexion = cadena;
}

// --- Demo simple para ver el flujo ---
public static class Program
{
```

```
public static void Main()
{
    // 1) Primera vez que tocamos 'Instancia' → Lazy
    todavía no creó nada.
    // Al acceder a .Value (implícito en la prop), Lazy
    ejecuta la lambda y crea el objeto.
    var cfg = Configuracion.Instancia;

    // 2) Leemos y mostramos la cadena de conexión por
    defecto.
    Console.WriteLine(cfg.CadenaConexion);

    // 3) Cambiamos el estado usando la API pública (no
    podemos setear directo por el 'private set').
    Configuracion.Instancia.ActivarDebug();

    // 4) Leemos el flag y comprobamos que quedó en true.
    Console.WriteLine($"Debug:
    {Configuracion.Instancia.ModoDebug}");
}
}
```

Cada patrón cumple un rol a la hora de estructurar el código, y como se mencionaba anteriormente estas técnicas no están para solucionar bugs si no para evitar que se formen.

Con estos patrones de diseño te aseguras cubrir los requisitos de un programa robusto y escalable.

Ahora, vamos a ejercitar un poco. Para cada patrón se ofrece una guía paso a paso de como crearlo, deberas recrear estas instrucciones en C# y si el programa ejecuta y es escalable, entonces dominaste lo esencial. La idea es ir por etapas claras, con pasos cortitos y objetivos concretos. No va código listo para pegar: vas a ir creando archivos y escribiendo lo justo en cada uno. Al final de cada práctica hay pruebas y cómo escalar agregando entidades sin romper nada.

1. Abstract Factory - Mapas

Objetivo: Armar una familia de componentes de “mapas” compuesta por dos productos relacionados:

1. *IMapa* (mostrar mapa)
2. *IGeocoder* (buscar direcciones)

Vas a tener dos familias completas: GoogleMaps y OpenStreetMap. La app cliente no conoce clases concretas, solo interfaces y la fábrica. Queremos cambiar toda la familia con una sola línea:

- Cambiar del stack “Google” al stack “OSM” sin tocar el código del cliente.
- Agregar nuevas familias (por ejemplo, “Mapbox”) y/o nuevos productos (por ejemplo, IRuteador) sin romper nada.

Instrucciones:

1. Proyecto de consola C#: MapasAbstractFactory (Dejá Program.cs vacío por ahora.)
2. Definir contratos (interfaces) de los productos
 - Archivo IMapa.cs
 - Interface IMapa con un método Render(string ubicacion).
 - La idea: “dibujar” el mapa de una ubicación (podés simular con Console.WriteLine).
 - Archivo IGeocoder.cs
 - Interface IGeocoder con un método Buscar(string textoDireccion) que devuelva (por ahora) un string simulando coords.
3. Definir la fábrica abstracta
 - Archivo IMapServicesFactory.cs
 - Interface IMapServicesFactory con:
 - IMapa CreateMapa()
 - IGeocoder CreateGeocoder()

4. Implementar familia GoogleMaps

- Archivo GoogleMapa.cs
 - Clase que implementa IMapa. En Render imprime algo tipo: "GoogleMaps mostrando: {ubicacion}".
- Archivo GoogleGeocoder.cs
 - Clase que implementa IGeocoder. En Buscar imprime y devuelve un string tipo "Coords GM: -38.00, -57.55 para {textoDireccion}".
- Archivo GoogleMapServicesFactory.cs
 - Clase que implementa IMapServicesFactory y retorna GoogleMapa/GoogleGeocoder.

5. Implementar familia OpenStreetMap (OSM)

- Archivo OsmMapa.cs
 - Implementa IMapa. Render imprime: "OSM mostrando: {ubicacion}".
- Archivo OsmGeocoder.cs
 - Implementa IGeocoder. Buscar devuelve "Coords OSM: -38.01, -57.54 para {textoDireccion}".
- Archivo OsmMapServicesFactory.cs
 - Implementa IMapServicesFactory y retorna OsmMapa/OsmGeocoder.

6. Cliente que usa la fábrica sin conocer concreciones

- Archivo Navegador.cs
 - Clase Navegador con dos campos privados: IMapa y IGeocoder.
 - Constructor recibe IMapServicesFactory, y crea ambos productos con esa fábrica.
 - Método Explorar(string ubicacion, string consultaDireccion):
 - Usa IMapa.Render(ubicacion)
 - Usa IGeocoder.Buscar(consultaDireccion) y lo muestra por consola

7. Probar en Program.cs

- Instanciá GoogleMapServicesFactory, pasala a Navegador y corré Explorar con una ubicación y una dirección.
- Cambiá a OsmMapServicesFactory y corré lo mismo. La salida debe cambiar de “Google” a “OSM” sin tocar la clase Navegador.

Salida esperada (referencial, no te cases con el texto exacto)

- Con Google:
 - “GoogleMaps mostrando: Mar del Plata”
 - “Coords GM: ... para Av. Colón 1234”
- Con OSM:
 - “OSM mostrando: Mar del Plata”
 - “Coords OSM: ... para Av. Colón 1234”

Escalado:

A) Nueva familia: Mapbox

- Crear MapboxMapa, MapboxGeocoder y MapboxMapServicesFactory.
- En Program, cambiar la fábrica por Mapbox y validar que todo siga andando.

B) Nuevo producto en la familia: IRuteador

- Agregar interface IRuteador con CalcularRuta(string desde, string hasta).
- Agregar CreateRuteador() a IMapServicesFactory.
- Implementar GoogleRuteador, OsmRuteador (y el de Mapbox si te animás).
- Ajustar Navegador para usar también el ruteador. Todas las fábricas deben proveerlo.

2. Builder: Construyendo hamburguesas.

Objetivo: Construir objetos complejos (Hamburguesa) con muchas opciones sin crear constructores kilométricos. Usar Builder para tener una API encadenable, validaciones y recetas.

- Crear hamburguesas “a medida” con métodos `Con*()` y un `Build()` que valida.
- Poder agregar más ingredientes y recetas sin tocar el código existente.

Instrucciones:

1. Crear un proyecto de consola llamado: `HamburguesaBuilder` (dejar `Program.cs` vacío por ahora)
2. Definir el producto
 - Crear archivo: `Hamburguesa.cs`
 - Clase `Hamburguesa` con propiedades:
 - `Pan (string)`, `Carne (string)`
 - `Queso, Lechuga, Tomate, Cebolla (bool)`
 - `Salsa (string)`
 - `ToString` que arme un texto con los datos (usá `StringBuilder` si querés).
 - `Setters` internos o públicos controlados según prefieras.
3. Definir el contrato del Builder
 - Crear archivo: `IHamburguesaBuilder.cs`
 - Interface con métodos encadenables:
 - `ConPan(string)`, `ConCarne(string)`
 - `ConQueso(bool = true)`, `ConLechuga(bool = true)`, `ConTomate(bool = true)`, `ConCebolla(bool = true)`
 - `ConSalsa(string)`
 - `Build()` que devuelve `Hamburguesa`

4. Implementar el Builder concreto

- Crear archivo: `HamburguesaClasicaBuilder.cs`
 - Mantener un objeto `Hamburguesa` privado que se va completando.
 - Cada método `Con*` setea el dato y devuelve `this`.
 - En `Build()` validar que haya Pan y Carne (mínimo). Si falta algo, tirar `InvalidOperationException`.
 - Devolver la `Hamburguesa` y (opcional) resetear el estado interno para reuso.

5. (Opcional pero muy útil) Director con recetas

- Crear archivo: `Cocinero.cs`
 - Métodos que reciban `IHamburguesaBuilder` y devuelvan `Hamburguesa`:
 - `Clasica`: pan clásico, carne simple, queso sí, lechuga sí, tomate sí, salsa ketchup.
 - `DobleCheddar`: pan brioche, carne doble, queso sí, cebolla sí, salsa cheddar.

6. Probar en `Program.cs`

- Crear una hamburguesa con el `Cocinero` y el `Builder`;
- Crear otra hamburguesa manual encadenando métodos `Con*` y terminando con `Build()`.
- Mostrar `ToString` por consola.
- Ver impresas 2 o 3 hamburguesas con sus ingredientes.
- Si provocás un error (por ejemplo, no seteás `Carne`) el `Build()` debe tirar excepción.

Escalado

- Agregar ingredientes nuevos (`Pepinillos`, `Bacon`). Sumar propiedades al producto y métodos `ConPepinillos/ConBacon` al builder.
- Agregar nuevas recetas en el `Cocinero` (`Veggie`, `TripleConBacon`).

- Agregar varias “familias” de builders (por ejemplo, HamburguesaGourmetBuilder), creá una nueva clase que implemente IHamburguesaBuilder con defaults diferentes.

3. Singleton

Objetivo: Crear un registro de logs central para toda la app. Debe existir una única instancia accesible desde cualquier parte. Queremos inicialización perezosa, acceso global y una API simple para loggear mensajes.

- Garantizar que siempre trabajás con el mismo Logger.
- Extender con niveles de log, destinos y formatos sin romper a los consumidores.

Instrucciones

1. Proyecto de consola C#: LoggerSingleton (Dejá Program.cs vacío por ahora.)
2. Diseñar la clase Singleton
 - Archivo Logger.cs
 - Marcá la clase como sealed.
 - Constructor privado (nadie hace new desde afuera).
 - Campo estático que mantenga la instancia única.
 - Sugerencia simple: Lazy<Logger> para no pelearte con multi-hilo.
 - Propiedad estática Instancia que devuelva esa instancia.
3. Definir el estado mínimo y la API
 - En Logger.cs, agregá:
 - Una lista interna de strings (o similar) para almacenar mensajes (historial básico).
 - Método Info(string mensaje) que:

- Agregue el mensaje al historial con un prefijo tipo “[INFO] ...”
- Lo muestre por consola.
- Método Dump() que liste todo el historial por consola (para inspección).

La idea: que cualquiera pueda hacer `Logger.Instancia.Info("...")` desde cualquier parte del programa.

4. Probar la unicidad en Program.cs

- Tomar una referencia a `Logger.Instancia` en una variable “a”.
- Llamar a `Info` varias veces.
- Tomar otra referencia a `Logger.Instancia` en una variable “b”.
- Llamar a `Dump` desde “b”.
- Mostrar por consola si a y b referencian al mismo objeto (`ReferenceEquals`).

Salida esperada (referencial no te cases!!)

- Se ven los mensajes con prefijo “[INFO] ...”.
- El `Dump` lista todos los mensajes previos.
- La comparación de referencias da “true”.

Escalado

A) Niveles de log

- Agregar métodos `Warning(string)`, `Error(string)` con prefijos diferentes.
- Opcional: un enum `LogLevel` y un método genérico `Log(LogLevel, string)`.

B) Destinos alternativos

- Agregar un “destino” seleccionable:

- Por defecto, consola.
- Alternativa: archivo de texto (crear/append).
- Mínimo: un flag interno tipo “UsarArchivo” y una ruta. Agregar métodos para configurarlo en tiempo de ejecución:
 - UsarConsola()
 - UsarArchivo(string ruta)

C) Formato y contexto

- Agregar fecha/hora a cada línea.
- Agregar un campo “Contexto” opcional (por ejemplo, el nombre del módulo) y métodos para setearlo:
 - SetContexto(string nombre)
 - El log imprime “[INFO][HH:mm:ss][Contexto] mensaje”
- Verificar que los cambios de configuración impactan en toda la app (es la misma instancia).

D) Prueba de “múltiples lugares”

- Crear una clase cualquiera (por ejemplo, ProcesadorDePedidos) con un método Ejecutar().
- Desde adentro de Ejecutar, loguear varios mensajes usando Logger.Instancia.
- Llamar a Ejecutar desde Program y verificar que el historial es único y compartido.