Contenidos

Tecnologías de Desarrollo de Software IDE

Unidad: 8

"Características Avanzadas"

Capítulo: A

"Buenas Prácticas de Programación"

(Última Actualización: 1 Julio 2025)

Índice

Unidad 8: Características Avanzadas

Capítulo A: Buenas Prácticas de Programación

IMPORTANTE: los **contenidos** aquí desarrollados **no son necesariamente completos** en su total profundidad, por lo que **se sugiere consultar otras fuentes**, tales como la **Bibliografía** sugerida por la cátedra en el Programa Analítico y docentes.

Objetivo(s)

- Comprender la importancia de las buenas prácticas en el desarrollo de software.
- Aplicar buenas prácticas de programación utilizando tecnologías .NET.

Temas

1. Introducción a las Buenas Prácticas de Programación
2. Principios Fundamentales de Programación Limpia (Clean Code)
3. Principios SOLID en .NET
4. Manejo de Excepciones y Errores.
5. Uso de Patrones de Diseño Comunes en .NET.
6. Organización de Proyectos en .NET.
7. Manejo de Versionamiento y Control de Código Fuente.
8. Testing y Código de Calidad.
9. Buenas Prácticas Específicas de C# y .NET.
10. Principios de Seguridad en el Desarrollo de Software.
11. Performance y Optimización en Aplicaciones .NET.
12. Principios de Documentación y Comentarios.
13. CI/CD y Deployments en Entornos .NET.
14. Revisión de Código (Code Review).
15. Conclusiones y Recomendaciones Finales.

¿Qué son las buenas prácticas de programación?
¿Qué son las Convenciones de Nombre (o de Nomenclatura)?

IMPORTANTE: se encuentran desarrollados de forma introductoria algunos temas, dejando al estudiante profundizar los restantes utilizando las fuentes de información explicitadas.

Introducción

Las **Buenas Prácticas de Programación** son un conjunto de recomendaciones, normas y enfoques que buscan mejorar la calidad del código, haciéndolo más **legible, mantenible,**

escalable y seguro. No son imposiciones rígidas, pero seguirlas facilita el trabajo en equipo y la evolución de los sistemas en el tiempo.

Importancia en proyectos reales

- Permiten que otros desarrolladores (o el mismo en el futuro) puedan entender, corregir o extender el código de manera rápida y segura.
- Reducen la generación de bugs y problemas ocultos por malas prácticas.
- Mejoran la calidad de los despliegues y la experiencia del usuario final.
- Fomentan un estilo de trabajo profesional, ordenado y confiable, fundamental para empresas de desarrollo.

Consecuencias de no aplicarlas

- Código difícil de entender y modificar (código “espagueti”).
- Aumento de errores en producción y tiempos excesivos de mantenimiento.
- Retrabajos y pérdidas de tiempo en tareas simples.
- Frustración del equipo y baja calidad de producto.

Las buenas prácticas no son solo para “programar bonito”, sino para asegurar la calidad y viabilidad del software en entornos reales, donde la colaboración y los cambios son constantes.

No dude en consultar en Microsoft LEARN - Common C# code conventions: https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/coding-conventions

Principios fundamentales del Codigo Limpio (Clean Code)

La codificación limpia se basa en principios que buscan crear código fácil de entender, mantener y modificar. Se centra en la legibilidad, simplicidad y claridad. Es fundamental usar nombres descriptivos, dividir el código en partes lógicas y realizar pruebas unitarias entre otras.

Legibilidad y claridad del código

- El código debe ser fácil de leer como si fuera un texto narrativo, priorizando la claridad por sobre la optimización prematura.
- Un buen indicador es que otro desarrollador pueda comprender qué hace un bloque de código en menos de un minuto sin necesidad de adivinar.

```
// Ejemplo de Código Impuro
```

```
public class Op
{
   public static Calc(int a, int b)
   {
      return a + a * b;
   }
}
int x = Op.Calc(100,2);
```

En esta versión, el código utiliza nombres de variables genéricos (a, b, y x), lo que hace que no quede claro qué hace la función y qué representa cada parámetro.

Código limpio:

```
function calculateTotalPrice(basePrice, taxRate) {
  return basePrice + basePrice * taxRate
}
let totalPrice = calculateTotalPrice(100, 0.2)
```

En este ejemplo, los nombres de funciones y variables descriptivos ayudan a mejorar la legibilidad y el mantenimiento, lo que permite que los futuros desarrolladores los comprendan calculateTotalPricesin explicaciones adicionales.

```
// Ejemplo de Código Limpio
public class Factura
{
   public static CalcularTotal(int precioBace, int impuesto)
   {
      return precioBase + precioBase * impuesto;
   }
}
int x = Operaciones.CalcularTotal(100,2);
```

Convenciones de Nombres

Las Convenciones de Nombre (o Nomenclatura) son reglas estándar utilizadas para nombrar variables, métodos, clases y otros elementos en el código fuente y la documentación, con el objetivo de favorecer la legibilidad y coherencia.

Existen varios tipos de Notaciones, cuyo uso va a depender del contexto donde se lo utiliza. En

C# y .NET, Microsoft y la comunidad utilizan algunas en particular.

- **PascalCase**: utiliza en mayúscula la primera letra de cada palabra, sin guiones bajos. Conocida también como “UpperCamelCase”.
  Usado en: clases, interfaces, métodos públicos, propiedades.
  Ejemplo: CustomerService, ProcessPayment()

- **camelCase**: a excepción de la primera palabra usa en mayúsculas la primera letra de cada palabra subsiguiente.
  Usado en: variables locales, parámetros de métodos, campos privados (cuando no se usa prefijo _).
  Ejemplo: orderList, paymentAmount

- **_camelCase** (underscore prefix): variante de “camelCase” con un guión bajo de prefijo.
  Usado en: campos privados en clases (como alternativa a camelCase).
  Ejemplo: _repository, _connectionString

- **SCREAMING_SNAKE_CASE**: utiliza guión bajo para separar las palabras escritas en mayúsculas. Conocida también como “UPPER_CASE”.
  Usado en: constantes.
  Ejemplo: MAX_ATTEMPTS, DEFAULT_TIMEOUT

- **Prefijos y sufijos específicos en .NET:**
  - Interfaces: comienzan con I (Ejemplos: IRepository, ILogger)
  - Métodos asíncronos: terminan con Async (Ejemplo: GetUserAsync())

Otras Notaciones

Existen otras notaciones de nomenclatura que no necesariamente se utilizan en ámbitos de desarrollo .NET; se mencionan algunas a continuación.

- **snake_case**: utiliza guión bajo (underscore) para separar las palabras. Conocido también como "underscore case".
  Usado en: nombre de tablas, campos y claves en base de datos
  Ejemplo: ordenes_de_compra, precio_unitario, producto_id

- **Húngara**[1]: utiliza una serie de caracteres de prefijo, generalmente compuesto por una

[1] Su inventor, Charles Simonyi, nació en Hungría, de allí su nombre de “Notación húngara”.

abreviación de constantes. En IDEs que no cuentan/contaban con funciones de autocompletado, intellisense o tooltips para indicar el tipo de dato de la variable (Ej. strAlumno, intNota)
Usado en: controles y formularios de interfaz gráfica.
Ejemplo: btnAceptar, txtNombre, lblUsuario

- **Kebab**: usa guiones medios para separar cada palabra en minúsculas. No suele recomendarse su uso dado que en algunos lenguajes dicho símbolo guión medio (-) suele usarse para el operador resta.
  Usado en: atributos de clases y selectores en CSS
  Ejemplo: background-color, .mi-clase-principal, #mi-id-unico

Para más información consulte Microsoft LEARN:

C# identifier naming rules and conventions:
https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/identifier-names

Ejemplos de buenas y malas prácticas

✅ Correcto:
```
public class ProductService { }
public interface IProductRepository { }
private readonly IProductRepository _productRepository;
public string FirstName { get; private set; }
string firstName;
decimal totalAmount;
```

❌ **Incorrecto:**
```
public interface productrepository { }
public class productservice { }
private IProductRepository productrepository;
string Firstname;
decimal Total_amount;
```

Nombres significativos para variables, clases y métodos

- Los nombres deben reflejar con claridad qué representan o qué acción realizan.
- Evitar abreviaturas confusas (usar GetCustomerDetails en vez de GetCustDtl).

- Métodos con verbos que indiquen acción (CalculateTotal, SaveOrder).
- Clases con nombres de sustantivo que indique su responsabilidad (OrderProcessor, EmailSender).

Métodos cortos y específicos

- Cada método debe realizar una sola tarea clara.
- Debe tener nombres descriptivos que transmitan su propósito.
- Limitar su longitud para evitar que sea necesario “scrollear” mucho para comprenderlo.
- Si un método se vuelve demasiado largo o complejo, refactorizar en métodos privados más pequeños, concisos, específicos y descriptivos.

Existen algunos principios reconocidos por sus acrónimos en el ámbito del desarrollo de software como los siguientes:

Evitar código duplicado (DRY: Don’t Repeat Yourself)

- Extraer lógicas comunes en métodos reutilizables o servicios compartidos.
- El código duplicado aumenta la probabilidad de errores cuando se requiere un cambio.

Mantenerlo simple (KISS: Keep It Simple, S…)

- Priorizar la solución más simple que funcione correctamente.
- No complicar el código con estructuras innecesarias.

No vas a necesitarlo (YAGNI[2]: You Aren’t Gonna Need It)

- Evitar programar funcionalidades o capas extra que “podrían” usarse en el futuro, pero no son requeridas hoy.
- El software evoluciona con requerimientos reales, no con suposiciones prematuras.

Aplicar estos principios y convenciones de nombres mejora la calidad general del código, facilita la revisión en equipo y refuerza el profesionalismo de un desarrollador .NET.

Manejo de Excepciones y Errores

Una excepción es un evento que interrumpe el flujo normal de funcionamiento de un programa que generalmente se traduce en un error o es reconocido como tal.

[2] Acrónimo utilizado en el Desarrollo Ágil. Puede consultar en “Principio YAGNI: evita la sobreingeniería”.

Ejemplos de excepciones comunes en .NET:

- NullReferenceException: acceder a un objeto que es null.
- IndexOutOfRangeException: índice fuera de rango en un array o colección.
- FileNotFoundException: no se encuentra el archivo especificado.
- DivideByZeroException: división por cero.

Uso básico de una estructura try-catch-finally

La plataforma .NET, así como la mayoría de los lenguajes, cuentan con la estructura de control a lo que comúnmente se conoce como try-catch, que puede incluir de forma opcional al bloque finally como lo muestra el ejemplo siguiente:

```
try
{
    int[] numeros = {1, 2, 3};
    Console.WriteLine(numeros[5]);
}
catch (IndexOutOfRangeException ex)
{
    Console.WriteLine($"Error: {ex.Message}");
}
finally
{
    Console.WriteLine("Finalizó la ejecución del bloque try-catch.");
}
```

Buenas prácticas

- Capturar la excepción más específica posible.
- Crear una excepción personalizada cuando hay reglas de negocio no representadas una estándar.
- Cuando hay reglas de negocio no representadas por excepciones estándar, crear una personalizada.
- Evitar capturar Exception genérica a menos que sea para logging general.
- Utilizar bloque “finally” para liberar recursos.
- Registrar (Logging) las excepciones, para análisis posterior y mejora continua.
- Propagar excepciones (throw) de ser necesario, para mantener el stack trace original.

Ejemplo de Creación de excepciones personalizadas

```
public class InvalidOrderException : Exception
{
    public InvalidOrderException(string message) : base(message) { }
}

public class Order
{
    public void Validate()
    {
        throw new InvalidOrderException("El pedido no contiene productos.");
    }
}
```

Ejemplo de Propagación de excepciones y rethrow

```
try
{
    this.RealizarOperacion();
}
catch (InvalidOrderException ex)
{
    logger.LogError(ex, "Error al validar pedido");
}
finally
{
    throw; // Propaga manteniendo el stack trace
}
```

Frameworks comunes de registro (Logging) de excepciones:

- ILogger (Microsoft)
- NLog.
- Log4Net
- Serilog

Principios de Documentación y Comentarios

Es importante documentar y comentar el código dado que permiten:

- Facilitar la lectura y mantenimiento por parte de otros programadores (o uno mismo en el futuro).
- Explicar el por qué de ciertas decisiones de diseño.
- Generar documentación externa de manera automática con herramientas como XML Comments, Swagger en APIs, o DocFX.

Ejemplos de comentarios en C#

```
// Ejemplo de comentario de línea:
// Calcula el total del pedido
int total = cantidad * precio;

// Ejemplo de comentario de bloque:
/*
   Este bloque valida el pedido:
   - Verifica cantidad
   - Verifica stock
*/

// Ejemplo de comentario de documentación (XML):
// (se colocan sobre clases, métodos, propiedades, y son utilizados por
// herramientas como IntelliSense y generadores de documentación.)

/// <summary>
/// Calcula el total de un pedido con descuento.
/// </summary>
/// <param name="cantidad">Cantidad de unidades.</param>
/// <param name="precio">Precio unitario.</param>
/// <returns>Total calculado con descuento aplicado.</returns>
public decimal CalcularTotal(int cantidad, decimal precio)
{
   return cantidad * precio * 0.95m;
}
```

Buenas prácticas en comentarios:

- Comentar el propósito y la intención.
- No es recomendable comentar obviedades.
- Mantener los comentarios actualizados. Código desactualizado es malo, pero un comentario desactualizado es peor.
- Usar comentarios de documentación en todos los métodos públicos de bibliotecas o APIs.

Generación de documentación en .NET
Activar XML Documentation en el .csproj

```
<PropertyGroup>
  <GenerateDocumentationFile>true</GenerateDocumentationFile>
</PropertyGroup>
```

Luego, generar con herramientas como DocFX para documentación web o Swagger para APIs REST.

Arquitectura y Organización de Proyectos

Organizar correctamente la estructura de una solución .NET es fundamental para su **mantenibilidad, escalabilidad y colaboración** en equipo.

En .NET una **Solución** (.sln) es un contenedor que agrupa uno o más proyectos relacionados entre sí; en tanto que un **Proyecto** (.csproj) es una unidad de compilación que contiene archivos de código, configuraciones y dependencias. Cada proyecto tiene su propio archivo .csproj que define sus referencias y configuración de compilación.

Una vez definida la arquitectura de software a utilizar se utilizan diferentes tipos de proyectos para implementar y representar cada parte de la misma.

Existen una variedad de arquitectura que se pueden utilizar que van a depender del tamaño y complejidad del producto de software necesario entre las que podemos mencionar:

- Arquitectura Limpia (Clean) y sus variantes.
- Arquitectura MVC (Modelo Vista Controlador)
- Arquitectura en N capas.

Estructura típica en aplicaciones de mediana o gran complejidad con Clean Architecture[3]

- API: contiene los endpoints y configuración de presentación.
- Application: lógica de aplicación (casos de uso, interfaces de servicios).
- Domain: entidades de dominio y lógica pura del negocio.
- Infrastructure: implementaciones de repositorios, acceso a datos y servicios externos.

Estructura típica en proyectos MVP

[3] Propuesta por Robert C. Martin (Uncle Bob) "Cleaner Code".

/Controllers
/Models
/Views
/Services
/Repositories

Arquitectura por capas

- Presentation Layer: capa de presentación o interfaz de usuario (ej. Controllers en Web API).
- Application Layer: gestiona la lógica de aplicación (ej. servicios que orquestan entidades de dominio).
- Domain Layer: contiene las entidades, lógica y reglas de negocio puras, independientes de frameworks.
- Infrastructure Layer: implementa el acceso a datos, APIs externas, servicios de terceros, persistencia.

Conclusiones y Recomendaciones Finales

Las temas planteados hasta aquí, no intenten ser completos ni definitivos, existiendo otros aspectos relacionados de forma directa o indirecta sobre los que se pueden mencionar los siguientes:

- Patrones de Diseño y Principios SOLID.
- Manejo de Versionamiento y Control de Código Fuente.
- Testing y Código de Calidad.
- Revisión de Código (Code Review)
- Principios de Seguridad en el Desarrollo de Software
- Integración y Despliegue Continuo (CI/CD)
- Performance y Optimización de Aplicaciones

Complementar los temas aquí expuestos con aquellos temas abordados en otras asignaturas de la carrera tales como Análisis de Sistemas, Diseño de Sistemas, Base de Datos y Desarrollo de Software donde se abordan temáticas relacionadas sobre: Arquitecturas, Patrones, Pruebas, Calidad y Seguridad entre otras.

Por otra parte dado que la tecnología y el software en particular están en constante evolución y cambio, afecta a lo que son considerados buenas prácticas en un momento y contexto en particular es esperable que lo que son considerados como “buenas prácticas en el desarrollo de software” evolucionan y sufran cambios tambien por lo que se recomienda seguir investigando y consultando fuentes especializadas al respecto.

Resumen de la Unidad / Capítulo

IMPORTANTE: los **contenidos** desarrollados en esta sección **no son completos**, sino que son un mini resumen de conceptos generales de los temas de mayor importancia del capítulo a modo de un repaso general de los mismos una vez completada la lectura del mismo.

Buenas prácticas de programación: recomendaciones, normas y enfoques que buscan mejorar la calidad del código, haciéndolo más legible, mantenible, escalable y seguro.

<COMPLETAR con lo basico>

Bibliografía[4]

IMPORTANTE: Se sugiere revisar y ampliar con otras fuentes adicionales a las citadas, en el buscador disponible en Biblioteca UTN Rosario.

ALLS, Jason "Clean Code with C#: Refactor your legacy C# code base and improve application performance using best practices", Reino Unido, Packt Publishing, 2023.

MARTIN, Robert "Código limpio / Clean code: Manual de estilo para el desarrollo ágil de software / A Handbook of Agile Software Craftsmanship", Madrid, Anaya Multimedia, 2012.

Microsoft LEARN:
Common C# code conventions
https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/coding-conventions
C# identifier naming rules and conventions
https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/identifier-names
ASP.NET Core Best Practices
https://learn.microsoft.com/en-us/aspnet/core/fundamentals/best-practices?view=aspnetcore-9.0
Best practices for exceptions
https://learn.microsoft.com/en-us/dotnet/standard/exceptions/best-practices-for-exceptions
C# Best Practices : Dangers of Violating SOLID Principles in C#
https://learn.microsoft.com/en-us/archive/msdn-magazine/2014/may/csharp-best-practices-dangers-of-violating-solid-principles-in-csharp
Coding guidelines
https://learn.microsoft.com/en-us/previous-versions/mixed-reality/world-locking-tools/Documentation/HowTos/codingconventions

Google Style Guide https://google.github.io/styleguide/

Wikipedia

[4] La Bibliografía está organizada en una primer sección de libros, ordenados alfabéticamente por apellido del autor (Normas APA), algunos disponibles en Biblioteca Física o Virtual de UTN en cuyo caso aparecerá un enlace a los datos de la publicación que figuran en linea en el el sitio web de la biblioteca. Seguido se detallan aquellos recursos digitales ya sean otros libros disponibles en dicho formato que podrá encontrar en el aula virtual o de alguna otra fuente de información como pueden ser los contenidos en el sitio oficial de Microsoft y sitios especializados que poseen contenidos de valor.

https://es.wikipedia.org/wiki/Convenci%C3%B3n_de_nombres_(programaci%C3%B3n)
https://wikipedia.org/wiki/YAGNI

Blog ArSys.es
https://www.arsys.es/blog/principio-yagni-evita-la-sobreingenieria#:~:text=YAGNI%20es%20el%20acr%C3%B3nimo%20de,lo%20vas%20a%20necesitar%C2%BB).

Anexo A: Tema A

...

Historial de Versiones

Versión	Fecha	Autor	Detalle
0.1	01/07/25	Ezequiel Porta	Versión con estructura básica
0.2			

TODO List

1. Completar para poder publicar publicar