

# Guía Integral de Programación II

---

## 1. Objetivos de la guía

- Comprender y aplicar los principios fundamentales de la Programación Orientada a Objetos.
  - Desarrollar habilidades para implementar proyectos escalables usando el patrón MVC.
  - Incorporar los principios SOLID como guía de diseño profesional.
  - Implementar operaciones CRUD y persistencia de datos.
  - Ejercitar el uso de C# como lenguaje base para crear software robusto y mantenible.
- 

## 2. Contenidos teóricos clave (para repasar)

### a. Programación Orientada a Objetos (POO)

#### Definiciones clave:

- **Clase:** Plano o plantilla que define un tipo de objeto.
- **Objeto:** Instancia concreta de una clase.
- **Encapsulamiento:** Ocultar detalles internos y exponer sólo lo necesario.
- **Herencia:** Permite crear nuevas clases basadas en las clases existentes.
- **Polimorfismo:** Capacidad de ejecutar comportamientos diferentes con la misma interfaz.

**Consejo:** Pensá en objetos como entidades del mundo real que tienen *atributos* (propiedades) y *acciones* (métodos). Las clases son entonces moldes que utilizamos para crear esos objetos y donde definimos todas sus características.

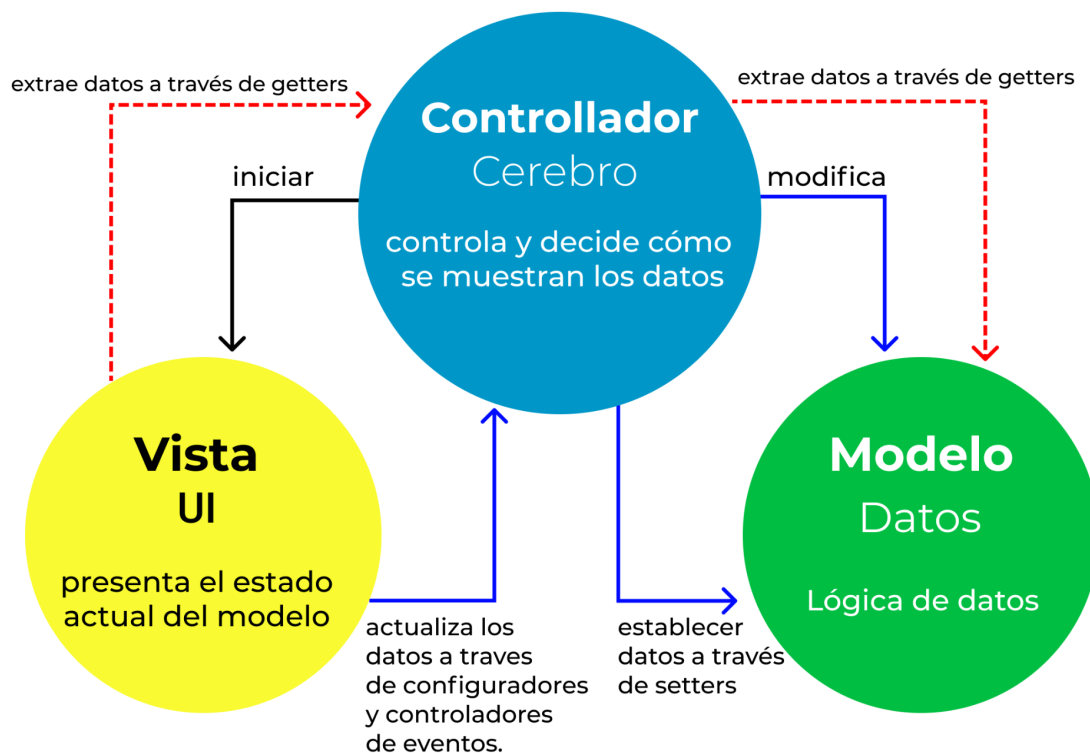
---

### b. Los principios SOLID.

Principio	Descripción	Pregunta guía
Single Responsibility	Cada clase debe tener una única razón de cambio.	¿Mi clase hace más de una cosa?
Open/Closed	Abierta a extensión, cerrada a modificación.	¿Puedo agregar comportamiento sin modificar código existente?
Liskov Substitution	Las subclasses deben poder reemplazar a las clases base.	¿Una subclase se comporta como su padre?
Interface Segregation	Usar interfaces pequeñas y específicas.	¿Mis interfaces son demasiado grandes?
Dependency Inversion	Depender de abstracciones, no de implementaciones.	¿Estoy acoplando mi clase a otra concreta?

## C. Patrón MVC (Modelo - Vista - Controlador)

# Patrones de Arquitectura MVC



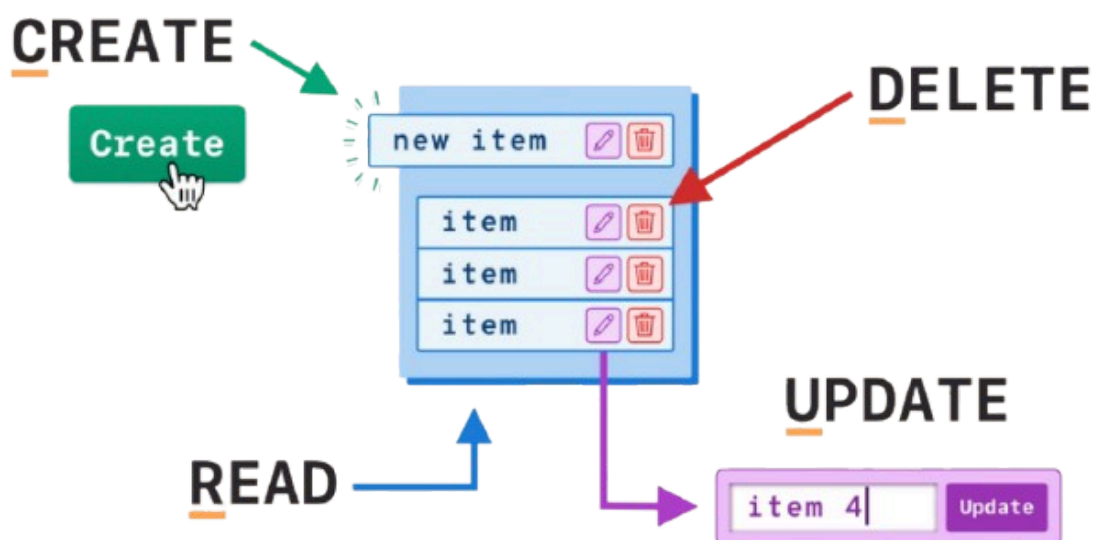
- **Modelo:** Lógica del negocio, características de las entidades y comportamiento de las mismas.
- **Vista:** Presentación de la información o ingreso de la información (aunque en consola es textual).
- **Controlador:** Orquestador que recibe inputs, controla la logica y actualiza el modelo y la vista.

**Objetivo:** Separar responsabilidades te permite escalar (mejorar) el proyecto sin romperlo. Permite que cada parte tenga su lógica, que sea fácil de solucionar o encontrar bugs, o reemplazar funciones específicas.

---

#### d. CRUD (Create, Read, Update, Delete)

- Operaciones fundamentales para manejar datos.
- Implementar en el **Modelo** usando una capa de **Repositorio** para abstraer el acceso a los datos.
- Persistencia sugerida: archivo `.json` usando `System.Text.Json`



### 3. Herramientas útiles y necesarias en C#:

- a. `List<T>`: lista dinámica de objetos.

- b. `File.ReadAllText()` / `File.WriteAllText()`: manejo de archivos.
- c. `JsonSerializer.Serialize()` / `Deserialize<T>()`: trabajar con JSON.
- d. `switch`, `try-catch`, `foreach`: estructuras comunes que debés dominar.
- e. `interface`, `abstract class`, `virtual`, `override`: pilares de POO avanzada sumamente necesarios.

**Consejo:** Revisa los links en la categoría POO de ETNA para acceder a documentación oficial sobre estos recursos.

---

## 4. Ejercitación práctica:

### Ejercicio 1: Fundamentos de POO

**Narrativa:** La municipalidad local te ha contratado para desarrollar un pequeño sistema de gestión de ciudadanos para su base de datos.

**Consigna:** Desarrollá una clase `Ciudadano` que contenga:

- Nombre completo (`string`)
- DNI (`string`)
- Edad (`int`)
- Método `Saludar()` que devuelva un string personalizado con el nombre y edad del ciudadano.

**Extras sugeridos:**

- Validá que la edad sea mayor o igual a 0.
- Agregá un método que indique si el ciudadano es mayor de edad.

**Objetivo:** Practicar encapsulamiento, métodos, constructores y lógica básica.

---

### Ejercicio 2: Herencia y Polimorfismo

**Narrativa:** Estás desarrollando un software para una veterinaria que trabaja con diferentes tipos de animales.

**Consigna:**

1. Creá una clase base `Animal` con:
  - Propiedad `Nombre`.

- Método virtual `EmitirSonido()`.
- 2. Creá dos clases que hereden de `Animal`:
- `Perro`: sobrescribí `EmitirSonido()` devolviendo "Guau!"
- `Gato`: devolvé "Miau!"
- 3. Agregá un método `Presentarse()` que devuelva por ejemplo:  
"Soy un gato llamado (nombre) y hago Miau!"

**Objetivo:** Aplicar herencia, polimorfismo y sobrescritura de métodos.

---

### Ejercicio 3: Principios SOLID

**Narrativa:** Un sistema de facturación actual hace todo desde una sola clase: genera la factura, la imprime y la guarda. Te piden refactorizarlo.

**Consigna:**

1. Recibís esta clase `Factura`, que hace todo esto:

```
class Factura {  
    public void CalcularTotal() { /* ... */ }  
    public void ImprimirFactura() { /* ... */ }  
    public void GuardarEnArchivo() { /* ... */ }  
}
```

2. Refactorizala aplicando **SRP**:
  - `FacturaCalculator`, `FacturaPrinter`, `FacturaSaver`.
3. Luego agregá una interfaz `IImprimible` y aplicá **ISP** para separar impresión digital de impresión papel.

**Objetivo:** Aplicar SRP e ISP. Iniciar el pensamiento en capas y separación de responsabilidades.

---

### Ejercicio 4: CRUD + JSON + Repository

**Narrativa:** Una ferretería quiere registrar productos y realizar operaciones básicas con ellos desde una app de consola.

**Consigna:**

1. Crear clase **Producto** con: **Id**, **Nombre**, **Precio**, **Stock**.
2. Implementá un CRUD por consola:
  - Alta (nuevo producto)
  - Baja (eliminar)
  - Modificación (editar un campo)
  - Consulta (mostrar todos)
3. Usá una clase **ProductoRepository** para manejar una lista interna de productos y serializarla en **productos.json**.
4. Al iniciar el programa, cargá los datos existentes (si hay) y al finalizar, guardalos.

**Objetivo:** Comprender el ciclo de vida de un dato, uso de listas, persistencia simple y separación de lógica de negocio y de acceso a datos.

---

## Ejercicio 5: Proyecto Final MVC

**Narrativa:** Una biblioteca barrial necesita una app en consola que le permita registrar libros y gestionar préstamos.



**Consigna:**




1. Modelo:
  - **Libro**: título, autor, ISBN, disponibilidad.
  - **Usuario**: nombre, email.
  - **Prestamo**: libro, usuario, fecha.
2. Controladores:
  - Permite prestar libros, devolverlos, listar libros disponibles, etc.
3. Vista:
  - Menú simple en consola que permita navegar entre opciones.
4. Separación clara entre carpetas: **Models**, **Controllers**, **Views**.

**Objetivo:** Aplicar todos los conceptos previos, incluyendo separación por capas, lógica de negocio, persistencia y diseño limpio.

---

## 5. Consejos y buenas prácticas

-  **Nombrá bien las clases y métodos:** Que el nombre diga claramente lo que hace.
-  **Mantené tu código limpio:** Comentarios innecesarios y duplicación son enemigos del mantenimiento.

-  **Dividí en capas:** No pongas lógica de negocio en la vista ni acceso a datos en el controlador.
  -  **Probá frecuentemente:** Cada pequeño cambio debe funcionar por sí solo.
  -  **Organizá tus carpetas:** Models, Views, Controllers, Data, Utils.
-