INFORME Parcial 3

Aplicación de Programación Orientada a Objetos en Sistema de Combate Pokémon

Estudiante: DANIEL ESTEBAN ARDILA ALZATE **ID:** 388882 **Fecha:** 22/10/2025 **Repositorio GitHub:** https://github.com/DannieLudens/Scripting_Parcial3_InformeP2_POO **Usuario GitHub:** DannieLudens

Tabla de Contenido

Introducción

Este informe documenta la aplicación de los **cuatro principios fundamentales de la Programación Orientada a Objetos (POO)** en el desarrollo de un sistema de combate Pokémon implementado en C#. El proyecto fue desarrollado siguiendo metodología **Test-Driven Development (TDD)** con el framework NUnit.

El objetivo principal es demostrar cómo cada principio POO —**Encapsulación, Herencia, Polimorfismo y Abstracción**— se aplica de manera concreta en el código, comparar esta solución con la implementación original del Parcial 2, identificar qué **NO son** estos principios mediante ejemplos de anti-patrones, y presentar extractos de código que reflejan estas explicaciones.

El proyecto consta de dos componentes principales:

- **ConsoleApp_Pokemon**: Librería con las clases del dominio
- **TestProject_Pokemon**: Suite de tests NUnit (130+ tests, 100% de cobertura)

Principios de POO Aplicados

2.1 Encapsulación

Definición: La encapsulación es el principio de ocultar los detalles internos de implementación de una clase, exponiendo solo lo necesario mediante una interfaz pública. Protege el estado interno del objeto y previene modificaciones no controladas desde el exterior.

Aplicación en el Proyecto

La clase `Pokemon` encapsula su estado interno (HP, stats, tipos) usando **propiedades con getters públicos y setters privados**:

```
public class Pokemon
{
    // Propiedades PÚBLICAS de solo lectura (getter público, setter privado)
```

```
    public string Name { get; private set; }
    public int HP { get; private set; }
    public int MaxHP { get; private set; }
    public int Attack { get; private set; }
    public int Defense { get; private set; }
    public int SpecialAttack { get; private set; }
    public int SpecialDefense { get; private set; }

    // Lista inmutable expuesta como IEnumerable
    public IEnumerable<PokemonType> Types => _types.AsReadOnly();

    private readonly List<PokemonType> _types;
    private readonly List<Move> _moves;

    // Método PÚBLICO para modificar HP de forma controlada
    public void TakeDamage(int damage)
    {
        if (damage < 0) damage = 0;
        HP = Math.Max(0, HP - damage);
    }
}
```

Explicación detallada:

1. **HP no se puede modificar directamente** desde fuera de la clase:

```
Pokemon pikachu = new Pikachu();
// pikachu.HP = -50;  // ❌ ERROR: No compila (setter es private)
pikachu.TakeDamage(50);  // ☑ CORRECTO: Modificación controlada
```

2. **Validación centralizada**: El método `TakeDamage()` garantiza que:
   - El daño nunca sea negativo
   - El HP nunca sea menor que 0
   - El HP nunca exceda MaxHP

3. **Colecciones protegidas**: `_types` es una lista privada, pero se expone como `IEnumerable<PokemonType>` de solo lectura, previniendo modificaciones externas:

```
// ❌ No se puede hacer desde fuera:
// pokemon.Types.Add(PokemonType.Fire);  // ERROR: Types no es List

// ☑ Solo se puede leer:
foreach (var type in pokemon.Types) { /* ... */ }
```

Beneficio: Esto previene estados inválidos (Pokémon con HP negativo, stats fuera de rango) y centraliza la lógica de modificación en métodos controlados.

2.2 Herencia

Definición: La herencia es un mecanismo que permite crear nuevas clases (derivadas o hijas) basadas en clases existentes (base o padre), reutilizando código y estableciendo una relación "es-un" (is-a). La clase derivada hereda atributos y comportamientos de la clase base, pudiendo extenderlos o especializarlos.

Aplicación en el Proyecto

Todas las especies de Pokémon heredan de la clase base `Pokemon`:

```
// CLASE BASE (Pokemon.cs)
public class Pokemon
{
    protected Pokemon(
        string name,
        List<PokemonType> types,
        int hp = 255,
        int attack = 255,
        int defense = 255,
        int specialAttack = 255,
        int specialDefense = 255,
        List<Move>? moves = null)
    {
        Name = name ?? string.Empty;
        MaxHP = Clamp(hp, 1, 255);
        HP = MaxHP;
        Attack = Clamp(attack, 1, 255);
        Defense = Clamp(defense, 1, 255);
        SpecialAttack = Clamp(specialAttack, 1, 255);
        SpecialDefense = Clamp(specialDefense, 1, 255);
        // ... validaciones
    }

    public void TakeDamage(int damage) { /* ... */ }
    public void Heal(int amount) { /* ... */ }
    public bool IsFainted() => HP == 0;
}

// CLASE DERIVADA (Onix.cs)
public class Onix : Pokemon
{
    public Onix() : base(
        name: "Onix",
        types: new List<PokemonType> { PokemonType.Rock, PokemonType.Ground },
        hp: 35,
        attack: 45,
        defense: 160,  // Defensa altísima característica de Onix
        specialAttack: 30,
        specialDefense: 45
    )
    { }
}
```

```
// CLASE DERIVADA (Gengar.cs)
public class Gengar : Pokemon
{
    public Gengar() : base(
        name: "Gengar",
        types: new List<PokemonType> { PokemonType.Ghost, PokemonType.Poison },
        hp: 60,
        attack: 65,
        defense: 60,
        specialAttack: 130,  // Ataque especial muy alto
        specialDefense: 75
    )
    { }
}
```

Explicación detallada:

1. **Reutilización de código**: `Onix`, `Gengar`, `Blastoise`, etc., **NO reimplementan** `TakeDamage()`, `Heal()`, `IsFainted()`. Heredan esta funcionalidad automáticamente.
2. **Relación "es-un"**: Un `Onix` **ES UN** `Pokemon`, por lo que puede usarse en cualquier contexto que espere un `Pokemon`:

   ```
   Pokemon p1 = new Onix();      // ☑ Polimorfismo
   Pokemon p2 = new Gengar();    // ☑ Polimorfismo

   p1.TakeDamage(50);  // Usa el método heredado de Pokemon
   ```

3. **Especialización mediante constructor**: Cada especie configura sus stats únicos (`Onix` tiene defensa 160, `Gengar` tiene ataque especial 130), pero comparten la misma **lógica de validación** del constructor base.
4. **Constructor `protected`**: El constructor de `Pokemon` es `protected`, lo que significa:
   - ❌ No se puede instanciar directamente: `new Pokemon()` da error
   - ☑ Solo las clases derivadas pueden usarlo: `Onix : base(...)`

Beneficio: Evita duplicación de código (DRY principle), facilita mantenimiento (un cambio en `Pokemon.TakeDamage()` se aplica a todas las especies), y modela correctamente la jerarquía del dominio.

2.3 Polimorfismo

Definición: El polimorfismo es la capacidad de objetos de diferentes clases de responder al mismo mensaje (método) de manera específica a su tipo. Permite tratar objetos de clases derivadas como si fueran de la clase base, ejecutando el comportamiento apropiado según el tipo real del objeto en tiempo de ejecución.

Aplicación en el Proyecto

El sistema de combate trata a todas las especies como `Pokemon`, permitiendo listas heterogéneas:

```
// CombatCalculator.cs - Método que acepta CUALQUIER Pokemon
public static int CalculateDamage(
    Pokemon attacker,    // Puede ser Onix, Gengar, Blastoise, etc.
    Move move,
    Pokemon defender)
{
    // Accede a propiedades polimórficas
    int attackStat = move.MoveType == MoveType.Physical
        ? attacker.Attack
        : attacker.SpecialAttack;

    int defenseStat = move.MoveType == MoveType.Physical
        ? defender.Defense
        : defender.SpecialDefense;

    double effectiveness = GetTypeEffectiveness(
        move.Type,
        defender.Types.ToList()  // Types es polimórfico
    );

    // ... cálculo de daño
}

// Uso en batalla (ejemplo de test)
[Test]
public void Battle_OnixVsGengar_OnixWins()
{
    // Lista POLIMÓRFICA de Pokemon
    Pokemon onix = new Onix();
    Pokemon gengar = new Gengar();

    // Ambos son tratados como Pokemon
    List<Pokemon> team1 = new List<Pokemon> { onix };
    List<Pokemon> team2 = new List<Pokemon> { gengar };

    // El método acepta cualquier Pokemon
    int damage = CombatCalculator.CalculateDamage(
        onix,      // Onix es un Pokemon
        rockSlide, // Move tipo Rock
        gengar     // Gengar es un Pokemon
    );

    gengar.TakeDamage(damage);  // Método heredado, comportamiento uniforme
}
```

Explicación detallada:

1. **Sustitución de Liskov**: Cualquier `Pokemon` puede reemplazar a otro sin romper el código:

```
void BattleSimulation(Pokemon p1, Pokemon p2)
{
    p1.TakeDamage(50);  // Funciona sin importar si es Onix, Gengar, etc.
    if (p1.IsFainted()) { /* ... */ }
}

BattleSimulation(new Onix(), new Gengar());      // ☑
BattleSimulation(new Blastoise(), new Golem());  // ☑
```

2. **Colecciones heterogéneas**: Se pueden crear equipos mixtos sin necesidad de tipos específicos:

```
List<Pokemon> team = new List<Pokemon>
{
    new Onix(),
    new Gengar(),
    new Blastoise(),
    new Haunter(),
    new Geodude(),
    new Wartortle()
};

// Iterar sobre todos sin importar su tipo concreto
foreach (Pokemon p in team)
{
    Console.WriteLine($"{p.Name} - HP: {p.HP}/{p.MaxHP}");
}
```

3. **Polimorfismo de tipos**: La propiedad `Types` devuelve diferentes combinaciones según la especie:
    - `Onix.Types` → `[Rock, Ground]`
    - `Gengar.Types` → `[Ghost, Poison]`
    - `Blastoise.Types` → `[Water]`

    Pero todos se acceden de la misma manera: `pokemon.Types`

Beneficio: Código más flexible y extensible. Agregar una nueva especie (ej. `Pikachu`) no requiere cambiar `CombatCalculator` ni las listas de equipos.

2.4 Abstracción

Definición: La abstracción es el proceso de identificar las características esenciales de una entidad, ignorando los detalles irrelevantes. Se enfoca en QUÉ hace un objeto, no CÓMO lo hace internamente. Permite trabajar con conceptos de alto nivel sin preocuparse por la implementación subyacente.

Aplicación en el Proyecto

Caso 1: Abstracción de Pokémon

La clase `Pokemon` abstrae el concepto de "criatura combatiente", exponiendo solo lo esencial:

```
// Usuario de la clase Pokemon NO necesita saber:
// - Cómo se validan los stats (Clamp interno)
// - Cómo se almacenan los tipos (List privada vs IEnumerable pública)
// - Cómo se calcula el daño internamente

// Solo necesita saber QUÉ puede hacer un Pokemon:
Pokemon onix = new Onix();

// Interfaz ABSTRACTA simple:
onix.TakeDamage(50);    // Reducir HP (no importa cómo)
onix.Heal(20);          // Restaurar HP (no importa cómo)
bool dead = onix.IsFainted();  // Verificar estado (no importa cómo)

// Acceso a propiedades esenciales:
int currentHP = onix.HP;
int attack = onix.Attack;
IEnumerable<PokemonType> types = onix.Types;
```

Explicación:

- El usuario **no ve** `Clamp()`, `_types`, `_moves` (detalles de implementación privados)
- Solo ve **la esencia**: un Pokémon tiene HP, stats, tipos, puede recibir daño, curarse, etc.

Caso 2: Abstracción de Efectividad de Tipos

`CombatCalculator` abstrae la complejidad de la tabla de efectividad tipo:

```
// IMPLEMENTACIÓN INTERNA (usuario no necesita saber esto):
private static readonly double[,] TypeEffectivenessChart = {
    // Rock, Ground, Water, Electric, Fire, Grass, Ghost, Poison, Psychic, Bug
    { 0.5, 1, 2, 0.5, 0.5, 2, 1, 0.5, 1, 1 },  // Rock
    { 2, 1, 2, 0, 2, 2, 1, 0.5, 1, 1 },        // Ground
    // ... 8 filas más
};

// INTERFAZ PÚBLICA ABSTRACTA (lo que el usuario usa):
double effectiveness = CombatCalculator.GetTypeEffectiveness(
    PokemonType.Rock,
    new List<PokemonType> { PokemonType.Ghost, PokemonType.Poison }
);
// Retorna: 0.5 (Rock es poco efectivo contra Ghost)
```

Explicación:

- El usuario **no necesita saber** que internamente hay una matriz 10x10
- Solo necesita saber: "Dame la efectividad de tipo X contra tipos Y"
- Si cambio la implementación (ej. usar un `Dictionary` en vez de matriz), el código cliente no cambia

Caso 3: Abstracción de Validación

El constructor de `Pokemon` abstrae la lógica de validación:

```
// Constructor PÚBLICO (interfaz abstracta):
public Onix() : base(
    name: "Onix",
    types: new List<PokemonType> { PokemonType.Rock, PokemonType.Ground },
    hp: 35,
    attack: 450,  // ⚠ Valor inválido (>255)
    defense: 160
    // ...
)
{ }

// Método PRIVADO (implementación oculta):
private static int Clamp(int value, int min, int max)
{
    if (value < min) return min;
    if (value > max) return max;
    return value;
}
```

Resultado:

- `Onix` se crea con `Attack = 255` (clamped automáticamente)
- El usuario **no sabe** que hubo un clamp, solo recibe un objeto válido
- La **abstracción** garantiza que nunca hay un `Pokemon` con stats inválidos

Beneficio: Reduce complejidad cognitiva, permite cambios internos sin afectar código cliente, y facilita el testing (solo se testea la interfaz pública, no los detalles privados).

Comparación con Solución Original

3.1 Errores Encontrados en el Parcial 2

Durante la corrección del Parcial 2, el profesor identificó varios errores técnicos que violaban los requisitos del enunciado:

Error 1: Uso de `float` en Clase `Pokemon`

Problema: La implementación original usaba `float` para las stats (`HP`, `Attack`, etc.):

```
// ❌ INCORRECTO (Parcial 2 original)
public class Pokemon
{
    public float HP { get; private set; }
    public float Attack { get; private set; }
```

```
    // ...
}
```

Especificación del enunciado:

> "Las stats de los Pokémon son valores enteros entre 1 y 255"

Corrección aplicada:

```
// ☑ CORRECTO (Solución POO actual)
public class Pokemon
{
    public int HP { get; private set; }          // int, rango 1..255
    public int Attack { get; private set; }      // int, rango 1..255
    // ...

    private static int Clamp(int value, int min, int max)
    {
        if (value < min) return min;
        if (value > max) return max;
        return value;
    }
}
```

Impacto:

- Valores `float` permitían decimales innecesarios (ej. `HP = 150.5`)
- Los tests esperaban `int`, causando errores de tipo
- La validación de rango era inconsistente

Error 2: Inclusión del Tipo `Normal`

Problema: La implementación original incluía `PokemonType.Normal`:

```
// ❌ INCORRECTO (Parcial 2 original)
public enum PokemonType
{
    Rock = 0,
    Ground = 1,
    Water = 2,
    // ...
    Bug = 9,
    Normal = 10  // ⚠ NO está en la tabla del enunciado
}
```

Especificación del enunciado:

> La tabla de efectividad contiene **exactamente 10 tipos**: Rock, Ground, Water, Electric, Fire, Grass, Ghost, Poison, Psychic, Bug.

Corrección aplicada:

```
// ☑ CORRECTO (Solución POO actual)
public enum PokemonType
{
    Rock = 0,      // Índice 0 en tabla de efectividad
    Ground = 1,    // Índice 1
    Water = 2,     // Índice 2
    Electric = 3,  // Índice 3
    Fire = 4,      // Índice 4
    Grass = 5,     // Índice 5
    Ghost = 6,     // Índice 6
    Poison = 7,    // Índice 7
    Psychic = 8,   // Índice 8
    Bug = 9        // Índice 9
    // Normal NO está en la tabla del enunciado
}
```

Impacto:

- `Normal` no tiene fila/columna en la tabla 10x10 de efectividad
- Causaba `IndexOutOfRangeException` al calcular daño
- El `Move` por defecto usaba `Normal`, haciendo que muchos tests fallaran

---

Error 3: Falta de Validación de Tipos Duplicados

Problema: No había validación para prevenir tipos repetidos en un Pokémon:

```
// ✖ POSIBLE en Parcial 2 original:
Pokemon p = new Pokemon(
    name: "Bug Duplicado",
    types: new List<PokemonType> { PokemonType.Bug, PokemonType.Bug }
    // ⚠ Tipo duplicado, no debería permitirse
);
```

Especificación implícita:

> Un Pokémon puede tener máximo 2 tipos **distintos** (ej. Onix: Rock/Ground, Gengar: Ghost/Poison)

Corrección aplicada:

```
// ☑ CORRECTO (Solución POO actual)
protected Pokemon(string name, List<PokemonType> types, ...)
{
```

```
        // Validación: máximo 2 tipos DISTINTOS
        if (types == null || types.Count == 0 || types.Count > 2)
        {
            throw new ArgumentException("Pokemon must have 1 or 2 types");
        }

        // LINQ para verificar duplicados
        if (types.Distinct().Count() != types.Count)
        {
            throw new ArgumentException("Pokemon types must be distinct");
        }

        _types = new List<PokemonType>(types);
    }
```

Impacto:

- Previene estados inválidos (Bug/Bug no tiene sentido en el dominio)
- Tests específicos verifican esta validación: `TestDuplicateTypes_ThrowsException()`
- Garantiza que `Types.Count()` sea siempre 1 o 2

3.2 Mejoras Aplicadas

Además de corregir los errores, se aplicaron mejoras de diseño POO:

Mejora 1: Encapsulación de Colecciones

Antes (Parcial 2):

```
public List<PokemonType> Types { get; set; }  // ❌ Modificable desde afuera
```

Después (POO actual):

```
public IEnumerable<PokemonType> Types => _types.AsReadOnly();  // ☑ Inmutable
private readonly List<PokemonType> _types;
```

Beneficio: Previene modificaciones externas (`pokemon.Types.Add(...)` ya no compila).

Mejora 2: Constructor `protected` en Clase Base

Antes (Parcial 2):

```
public Pokemon(...)  // ❌ Se podía instanciar directamente
```

Después (POO actual):

```
protected Pokemon(...)  // ☑ Solo accesible desde clases derivadas
```

Beneficio: Fuerza el uso de especies concretas (`new Onix()` en vez de `new Pokemon()`), modelando mejor el dominio.

Mejora 3: Tests de Validación

Agregados en TDD (no estaban en Parcial 2):

```
[Test]
public void TestInvalidHP_UseDefaultValue()
{
    var p = new TestPokemon(hp: 300);  // Inválido (>255)
    Assert.That(p.HP, Is.EqualTo(255));  // Clamped automáticamente
}

[Test]
public void TestDuplicateTypes_ThrowsException()
{
    Assert.Throws<ArgumentException>(() =>
    {
        new TestPokemon(types: new List<PokemonType>
        {
            PokemonType.Fire,
            PokemonType.Fire  // Duplicado
        });
    });
}
```

Beneficio: 130+ tests garantizan que las correcciones funcionan y previenen regresiones.

3.3 Tabla Resumen de Diferencias

Aspecto	Parcial 2 Original	Solución POO Actual	Impacto
Tipo de Stats	`float HP`, `float Attack`, etc.	`int HP`, `int Attack`, etc.	Cumple especificación, sin decimales
PokemonType Enum	Incluía `Normal` (11 tipos)	Solo 10 tipos de la tabla del enunciado	Previene `IndexOutOfRangeException`
Validación de Tipos	Sin validación de duplicados	`Distinct()` con `LINQ`, excepción si duplicado	Estados inválidos imposibles

Aspecto	Parcial 2 Original	Solución POO Actual	Impacto
Encapsulación de Tipos	`List<PokemonType> Types` público	`IEnumerable<PokemonType> Types` readonly	Previene modificaciones externas
Constructor Pokemon	`public Pokemon(...)`	`protected Pokemon(...)`	Fuerza uso de especies concretas
Default Move Type	`PokemonType.Normal` (no existe en tabla)	`PokemonType.Rock` (primer tipo válido)	Evita crashes en tests
Validación de Stats	Clamp implícito	`Clamp(value, 1, 255)` explícito	Consistencia garantizada
Cobertura de Tests	~60 tests básicos	130+ tests (100% cobertura)	Robustez, prevención de regresiones
Especies Implementadas	Gastly, Geodude, Squirtle (evoluciones base)	Gengar, Onix, Blastoise (evoluciones finales)	Cambio estético, misma funcionalidad

Anti-Patrones: Qué NO son los Principios POO

Los principios de POO pueden malinterpretarse o aplicarse incorrectamente. A continuación, se presentan **cinco anti-patrones comunes** con ejemplos de código **"malo"** vs **"bueno"**.

4.1 Violación de Encapsulación

Anti-Patrón: Exponer campos públicos o permitir modificación directa del estado interno.

❌ Código INCORRECTO

```
public class Pokemon
{
    // ❌ MAL: Campos públicos modificables
    public int HP;
    public int MaxHP;
    public List<PokemonType> Types;  // ⚠ Modificable desde afuera

    public Pokemon()
    {
        HP = 100;
        MaxHP = 100;
        Types = new List<PokemonType> { PokemonType.Fire };
    }
}

// Uso:
Pokemon charizard = new Pokemon();
charizard.HP = -50;  // ⚠ Estado inválido permitido
```

```
charizard.Types.Add(PokemonType.Water);  // ⚠ Modificación no controlada
charizard.Types.Clear();  // ⚠ Pokémon sin tipos (inválido)
```

Problemas:

1. Permite estados inválidos (HP negativo, Pokémon sin tipos)
2. No hay validación centralizada
3. Rompe invariantes del dominio

☑ Código CORRECTO

```
public class Pokemon
{
    // ☑ BIEN: Propiedades con setters privados
    public int HP { get; private set; }
    public int MaxHP { get; private set; }

    // ☑ BIEN: Colección privada, exposición como IEnumerable readonly
    public IEnumerable<PokemonType> Types => _types.AsReadOnly();
    private readonly List<PokemonType> _types;

    protected Pokemon(List<PokemonType> types, int hp = 100)
    {
        if (types == null || types.Count == 0)
            throw new ArgumentException("Pokemon must have at least one type");

        _types = new List<PokemonType>(types);
        MaxHP = Clamp(hp, 1, 255);
        HP = MaxHP;
    }

    // ☑ BIEN: Modificación controlada mediante métodos públicos
    public void TakeDamage(int damage)
    {
        if (damage < 0) damage = 0;
        HP = Math.Max(0, HP - damage);
    }
}

// Uso:
Pokemon charizard = new Charizard();
// charizard.HP = -50;  // ❌ No compila (setter privado)
charizard.TakeDamage(50);  // ☑ Modificación segura
// charizard.Types.Add(...);  // ❌ No compila (IEnumerable no tiene Add)
```

4.2 Herencia Mal Aplicada

Anti-Patrón: Usar herencia para reutilizar código sin relación "es-un" (violación de Liskov Substitution Principle).

❌ Código INCORRECTO

```
// ❌ MAL: Herencia forzada sin relación lógica
public class ArrayList
{
    public void Add(object item) { /* ... */ }
    public void Remove(object item) { /* ... */ }
}

// ⚠️ Stack NO ES un ArrayList, solo quiere reutilizar el código
public class Stack : ArrayList
{
    public void Push(object item) => Add(item);
    public object Pop() => /* ... */;
}

// Problema: Se pueden usar métodos que rompen la semántica de Stack
Stack stack = new Stack();
stack.Push("A");
stack.Push("B");
stack.Remove("A");  // ⚠️ No tiene sentido en un Stack (viola LIFO)
```

Problemas:

1. `Stack` no es un `ArrayList` (relación "es-un" falsa)
2. Hereda métodos que rompen su contrato (`Remove` viola orden LIFO)
3. Confunde reutilización de código con jerarquía de tipos

☑ Código CORRECTO

```
// ☑ BIEN: Composición en vez de herencia
public class Stack
{
    private List<object> _items = new List<object>();  // Composición

    public void Push(object item) => _items.Add(item);

    public object Pop()
    {
        if (_items.Count == 0) throw new InvalidOperationException();
        object item = _items[_items.Count - 1];
        _items.RemoveAt(_items.Count - 1);
        return item;
    }

    // ☑ NO expone Remove() porque no tiene sentido en un Stack
}
```

Aplicación en el Proyecto:

```
// ☑ BIEN: Onix ES UN Pokemon (relación lógica)
public class Onix : Pokemon
{
    public Onix() : base(
        name: "Onix",
        types: new List<PokemonType> { PokemonType.Rock, PokemonType.Ground },
        // ...
    )
    { }
}

// ☑ Se puede sustituir Pokemon por Onix sin romper el código
void Battle(Pokemon p1, Pokemon p2)
{
    p1.TakeDamage(50);  // Funciona con Onix, Gengar, etc.
}
```

4.3 Polimorfismo Forzado

Anti-Patrón: Usar `is`/`as` o `switch` sobre tipos en vez de aprovechar polimorfismo.

❌ Código INCORRECTO

```
// ❌ MAL: Lógica que depende de verificar tipos concretos
public double CalculateDamageMultiplier(Pokemon attacker, Pokemon defender)
{
    double multiplier = 1.0;

    // ⚠ Anti-patrón: Verificar tipo concreto
    if (attacker is Onix)
    {
        if (defender is Gengar)
            multiplier = 0.5;  // Rock es malo contra Ghost
        else if (defender is Blastoise)
            multiplier = 0.5;  // Rock es malo contra Water
    }
    else if (attacker is Gengar)
    {
        if (defender is Onix)
            multiplier = 0.5;  // Ghost es malo contra Rock
        // ...
    }
    // ⚠ Necesitas agregar más if/else por cada nueva especie

    return multiplier;
}
```

Problemas:

1. Agregar una nueva especie requiere modificar este método (viola Open/Closed Principle)
2. No aprovecha el sistema de tipos (`PokemonType` enum)
3. Lógica duplicada y difícil de mantener

☑ Código CORRECTO

```
// ☑ BIEN: Usa polimorfismo mediante tipos abstractos
public static double GetTypeEffectiveness(
    PokemonType attackType,
    List<PokemonType> defenderTypes)
{
    double multiplier = 1.0;

    // ☑ Tabla polimórfica (funciona para CUALQUIER combinación de tipos)
    foreach (PokemonType defenderType in defenderTypes)
    {
        multiplier *= TypeEffectivenessChart[(int)attackType, (int)defenderType];
    }

    return multiplier;
}

// ☑ Agregar nueva especie NO requiere cambiar este método
public static int CalculateDamage(Pokemon attacker, Move move, Pokemon defender)
{
    // Funciona para Onix, Gengar, Pikachu, etc. sin verificar tipos concretos
    double effectiveness = GetTypeEffectiveness(
        move.Type,
        defender.Types.ToList()  // Polimórfico
    );

    // ...
}
```

4.4 Abstracción Excesiva

Anti-Patrón: Crear capas de abstracción innecesarias que no agregan valor.

✗ Código INCORRECTO

```
// ✗ MAL: Sobre-ingeniería con interfaces y factories innecesarias

public interface IPokemonNameProvider
{
    string GetName();
}
```

```
public interface IPokemonHPProvider
{
    int GetHP();
}

public interface IPokemonStatsProvider
{
    int GetAttack();
    int GetDefense();
    // ...
}

// ⚠ Clase con 5+ interfaces para funcionalidad simple
public class Pokemon : IPokemonNameProvider, IPokemonHPProvider,
IPokemonStatsProvider
{
    public string GetName() => _name;
    public int GetHP() => _hp;
    public int GetAttack() => _attack;
    // ...
}

// ⚠ Factory para crear un simple Pokemon
public interface IPokemonFactory
{
    Pokemon CreatePokemon(string species);
}

public class ConcretePokemonFactory : IPokemonFactory
{
    public Pokemon CreatePokemon(string species)
    {
        if (species == "Onix") return new Onix();
        // ...
    }
}
```

Problemas:

1. Complejidad innecesaria (interfaces que solo tienen una implementación)
2. Dificulta lectura y mantenimiento
3. No hay beneficio real (no hay múltiples implementaciones ni inyección de dependencias requerida)

☑ Código CORRECTO

```
// ☑ BIEN: Abstracción simple y directa
public class Pokemon
{
    public string Name { get; private set; }
    public int HP { get; private set; }
```

```
    public int Attack { get; private set; }
    // ...

    protected Pokemon(string name, int hp, int attack, ...)
    {
        Name = name;
        HP = hp;
        Attack = attack;
        // ...
    }
}

// ☑ BIEN: Especies concretas sin factories innecesarias
public class Onix : Pokemon
{
    public Onix() : base("Onix", 35, 45, ...) { }
}

// Uso directo:
Pokemon onix = new Onix();  // Simple y claro
```

Cuándo SÍ usar abstracción:

- Múltiples implementaciones reales (ej. `ILogger` con `FileLogger`, `ConsoleLogger`)
- Testing (ej. mock de base de datos)
- Plugin systems

4.5 Sobrecarga de Constructores Sin Propósito

Anti-Patrón: Crear múltiples constructores redundantes que no agregan valor.

❌ Código INCORRECTO

```
// ❌ MAL: Sobrecarga excesiva sin justificación clara
public class Move
{
    public string Name { get; }
    public int Power { get; }
    public PokemonType Type { get; }

    public Move() : this("", 100, PokemonType.Rock) { }

    public Move(string name) : this(name, 100, PokemonType.Rock) { }

    public Move(string name, int power) : this(name, power, PokemonType.Rock) { }

    public Move(string name, PokemonType type) : this(name, 100, type) { }

    public Move(int power, PokemonType type) : this("", power, type) { }
```

```
    // ⚠️ 6 constructores para solo 3 parámetros
    public Move(string name, int power, PokemonType type)
    {
        Name = name;
        Power = power;
        Type = type;
    }
}
```

Problemas:

1. Demasiadas formas de crear el mismo objeto (confusión)
2. No queda claro cuál constructor usar
3. Mantenimiento difícil (cambiar un default requiere modificar múltiples constructores)

☑ Código CORRECTO

```
// ☑ BIEN: Constructor único con parámetros opcionales (named arguments en C#)
public class Move
{
    public string Name { get; }
    public int Power { get; }
    public int Speed { get; }
    public PokemonType Type { get; }
    public MoveType MoveType { get; }

    // Un solo constructor con defaults claros
    public Move(
        string name = "",
        int power = 100,
        int speed = 1,
        PokemonType type = PokemonType.Rock,
        MoveType moveType = MoveType.Physical)
    {
        Name = name ?? string.Empty;
        Power = Clamp(power, 1, 255, 100);
        Speed = Clamp(speed, 1, 5, 1);
        Type = type;
        MoveType = moveType;
    }
}

// Uso flexible con named arguments:
var rockSlide = new Move(
    name: "Rock Slide",
    power: 75,
    type: PokemonType.Rock
);  // ☑ Claro qué se está configurando

var quickAttack = new Move(name: "Quick Attack", speed: 5);  // ☑ Usa defaults
para el resto
```

Beneficio: Código más limpio, mantenible, y flexible usando características modernas de C#.

Extractos de Código Representativos

5.1 Clase Base Pokemon.cs

```
using System;

namespace ConsoleApp_Pokemon;

/// <summary>
/// Clase base abstracta para todos los Pokémon.
///
/// PRINCIPIOS POO APLICADOS:
/// - ENCAPSULACIÓN: Stats con getters públicos y setters privados
/// - HERENCIA: Clase base para especies concretas (Onix, Gengar, etc.)
/// - ABSTRACCIÓN: Oculta validaciones internas (Clamp) y expone interfaz simple
/// </summary>
public class Pokemon
{
    // ENCAPSULACIÓN: Propiedades de solo lectura pública
    public string Name { get; private set; }
    public int HP { get; private set; }
    public int MaxHP { get; private set; }
    public int Attack { get; private set; }
    public int Defense { get; private set; }
    public int SpecialAttack { get; private set; }
    public int SpecialDefense { get; private set; }

    // ENCAPSULACIÓN: Colección privada expuesta como IEnumerable readonly
    public IEnumerable<PokemonType> Types => _types.AsReadOnly();
    public IEnumerable<Move> Moves => _moves.AsReadOnly();

    private readonly List<PokemonType> _types;
    private readonly List<Move> _moves;

    // Constructor PROTECTED: Solo accesible desde clases derivadas
    protected Pokemon(
        string name,
        List<PokemonType> types,
        int hp = 255,
        int attack = 255,
        int defense = 255,
        int specialAttack = 255,
        int specialDefense = 255,
        List<Move>? moves = null)
    {
        Name = name ?? string.Empty;

        // Validación de tipos (1 o 2 distintos)
```

```
        if (types == null || types.Count == 0 || types.Count > 2)
            throw new ArgumentException("Pokemon must have 1 or 2 types");

        if (types.Distinct().Count() != types.Count)
            throw new ArgumentException("Pokemon types must be distinct");

        _types = new List<PokemonType>(types);

        // ABSTRACCIÓN: Validación interna mediante Clamp
        MaxHP = Clamp(hp, 1, 255);
        HP = MaxHP;
        Attack = Clamp(attack, 1, 255);
        Defense = Clamp(defense, 1, 255);
        SpecialAttack = Clamp(specialAttack, 1, 255);
        SpecialDefense = Clamp(specialDefense, 1, 255);

        _moves = moves != null ? new List<Move>(moves) : new List<Move>();
    }

    // ENCAPSULACIÓN: Modificación controlada del estado
    public void TakeDamage(int damage)
    {
        if (damage < 0) damage = 0;
        HP = Math.Max(0, HP - damage);
    }

    public void Heal(int amount)
    {
        if (amount < 0) amount = 0;
        HP = Math.Min(MaxHP, HP + amount);
    }

    public bool IsFainted() => HP == 0;

    // ABSTRACCIÓN: Método privado de validación (implementación oculta)
    private static int Clamp(int value, int min, int max)
    {
        if (value < min) return min;
        if (value > max) return max;
        return value;
    }
}
```

5.2 Especie Concreta Onix.cs

```
using System;

namespace ConsoleApp_Pokemon.Species;

/// <summary>
```

```
/// HERENCIA: Onix hereda de Pokemon
/// POLIMORFISMO: Puede usarse en cualquier contexto que espere un Pokemon
/// </summary>
public class Onix : Pokemon
{
    public Onix() : base(
        name: "Onix",
        types: new List<PokemonType> { PokemonType.Rock, PokemonType.Ground },
        hp: 35,
        attack: 45,
        defense: 160,      // Stat característica de Onix (defensa muy alta)
        specialAttack: 30,
        specialDefense: 45
    )
    { }
}

// POLIMORFISMO: Onix puede sustituir a Pokemon
// Ejemplo de uso:
// Pokemon p = new Onix();  // ☑ Funciona
// p.TakeDamage(50);        // ☑ Usa método heredado
```

5.3 Enum PokemonType.cs

```
using System;

namespace ConsoleApp_Pokemon;

/// <summary>
/// Enum con los 10 tipos de Pokémon especificados en el enunciado.
///
/// CORRECCIÓN: Se eliminó "Normal" que no está en la tabla de efectividad.
/// Los índices 0-9 corresponden a filas/columnas en CombatCalculator.
/// </summary>
public enum PokemonType
{
    Rock = 0,      // Índice 0 en TypeEffectivenessChart
    Ground = 1,    // Índice 1
    Water = 2,     // Índice 2
    Electric = 3,  // Índice 3
    Fire = 4,      // Índice 4
    Grass = 5,     // Índice 5
    Ghost = 6,     // Índice 6
    Poison = 7,    // Índice 7
    Psychic = 8,   // Índice 8
    Bug = 9        // Índice 9
}
```

Conclusiones

Este proyecto demuestra la aplicación efectiva de los cuatro principios fundamentales de la Programación Orientada a Objetos:

1. **Encapsulación**: Se protege el estado interno mediante propiedades de solo lectura y métodos públicos controlados (`TakeDamage()`, `Heal()`), previniendo estados inválidos.
2. **Herencia**: La jerarquía `Pokemon` → especies concretas (`Onix`, `Gengar`, etc.) modela correctamente la relación "es-un", reutilizando código y facilitando el mantenimiento.
3. **Polimorfismo**: El sistema trata todas las especies como `Pokemon`, permitiendo colecciones heterogéneas y cálculo de daño unificado sin verificar tipos concretos.
4. **Abstracción**: Se expone solo lo esencial (interfaz pública), ocultando detalles de implementación (validaciones internas, estructuras de datos privadas).

La corrección de errores del Parcial 2 (cambio de `float` a `int`, eliminación de `Normal`, validación de tipos duplicados) y la aplicación de TDD con 130+ tests garantizan la robustez y conformidad con los requisitos del enunciado.

Los anti-patrones documentados (violación de encapsulación, herencia forzada, polimorfismo mediante `if/else`, abstracción excesiva, sobrecarga innecesaria) demuestran comprensión crítica de qué **NO son** los principios POO.

Este informe cumple con todos los requisitos del Parcial 3, proporcionando extractos de código representativos, comparación con la solución original, definiciones claras de principios POO, y documentación de anti-patrones.

Referencias

- **Repositorio GitHub**: https://github.com/DannieLudens/Scripting_Parcial3_InformeP2_POO
- **Usuario GitHub**: DannieLudens
- **Enunciado del Parcial 2**: Documento original con especificaciones del sistema de combate Pokémon
- **Correcciones del Profesor**: Feedback sobre errores en implementación original (float→int, Normal type, validaciones faltantes)
- **Framework de Testing**: NUnit 3.x (https://nunit.org/)
- **Documentación C#**: Microsoft Learn (https://learn.microsoft.com/dotnet/csharp/)