



**Universidad de las Américas (UDLA)**  
Taller Best Practices Semana 12

**Integrantes del Grupo:**  
Mateo Puga

**Fecha de Entrega:** 29 de diciembre de  
2025

**Profesor:** Juan Jose León

## 1. IDENTIFICACIÓN DEL PROBLEMA DENTRO DE LAS RESTRICCIONES DEL PROYECTO

### 1.1 Contexto del Proyecto

El proyecto "Best Practices" es una aplicación web desarrollada con ASP.NET Core MVC 3.1 para la gestión de vehículos de la marca Ford. El sistema permite registrar diferentes modelos de vehículos (Mustang, Explorer) y realizar operaciones básicas sobre ellos como encender/apagar el motor y agregar combustible.

El equipo de desarrollo ha heredado un código base que, según reportes del equipo de QA, presenta comportamientos inconsistentes y errores en la funcionalidad de agregar vehículos. Además, se requieren nuevas funcionalidades que no pueden implementarse fácilmente con la arquitectura actual debido a problemas de diseño fundamentales.

#### **Alcance del Análisis:**

Este documento presenta:

- Análisis exhaustivo del código actual
- Identificación de problemas arquitectónicos y de diseño
- Propuesta de soluciones basadas en patrones de diseño reconocidos
- Prototipo funcional implementado que resuelve todos los problemas identificados

### 1.2 Problemas Encontrados en el Código Actual

Después de realizar un análisis técnico detallado del repositorio proporcionado, se identificaron **cinco problemas críticos** que afectan la calidad, mantenibilidad, escalabilidad y confiabilidad del sistema.

#### **PROBLEMA 1: Singleton No Thread-Safe con Potencial de Race Condition**

##### **Descripción Técnica:**

La clase VehicleCollection implementa el patrón Singleton de manera incorrecta, presentando una condición de carrera crítica en entornos multi-threaded. En ASP.NET Core, el modelo de procesamiento de peticiones HTTP es inherentemente concurrente, lo que significa que múltiples threads pueden ejecutar código simultáneamente.

##### **Análisis del Problema:**

El flujo problemático ocurre de la siguiente manera:

##### **Escenario Concurrente:**

##### **Thread 1 (Usuario A):**

1. Evalúa `if(_instance == null)` → TRUE
2. Está por ejecutar `_instance = new VehicleCollection()`

### 3. INTERRUMPIDO antes de completar la asignación

#### Thread 2 (Usuario B):

1. Evalúa if(\_instance == null) → TRUE (porque Thread 1 no terminó)
2. Ejecuta \_instance = new VehicleCollection()
3. Crea una NUEVA instancia

#### Thread 1 (continuación):

1. Ejecuta \_instance = new VehicleCollection()
2. SOBRESCRIBE la instancia creada por Thread 2

**Resultado:** Se han creado DOS instancias, violando el patrón Singleton. Los vehículos agregados en la primera instancia se pierden.

#### Consecuencias Identificadas:

**Pérdida de datos:** Los vehículos agregados en la primera instancia desaparecen cuando se sobrescribe

**Inconsistencia entre requests:** Usuarios diferentes ven colecciones de vehículos distintas

**Comportamiento no determinístico:** A veces funciona, a veces falla (depende del timing)

**Bug reportado por QA:** Explica los reportes de "vehículos que desaparecen"

#### Restricciones que Agravan el Problema:

- ASP.NET Core procesa requests HTTP concurrentemente por diseño
- No existe mecanismo de sincronización (lock, semáforo, Interlocked)
- El repositorio MyVehiclesRepository depende directamente de este Singleton
- No hay forma de controlar el ciclo de vida del Singleton

## PROBLEMA 2: Violación del Principio de Inversión de Dependencias (DIP)

#### Descripción Técnica:

El controlador HomeController instancia directamente clases concretas mediante el operador new, creando un acoplamiento fuerte entre la capa de presentación (Controller) y la capa de lógica de negocio (Factories). Esto constituye una violación directa del Principio de Inversión de Dependencias (DIP), uno de los cinco principios SOLID fundamentales.

#### Análisis del Problema:

El principio DIP establece que:

**Regla 1:** Los módulos de alto nivel NO deben depender de módulos de bajo nivel

**Regla 2:** Ambos deben depender de abstracciones (interfaces)

**Regla 3:** Las abstracciones no deben depender de detalles; los detalles deben depender de abstracciones

La implementación actual viola estos principios:

#### **Violación 1 - Dependencia Directa:**

HomeController (alto nivel)  
↓ depende directamente de  
FordMustangCreator (bajo nivel)

#### **Violación 2 - Acoplamiento Fuerte:**

```
var factory = new FordMustangCreator(); // ← Instanciación directa
```

**Violación 3 - Código Duplicado:** Los métodos AddMustang() y AddExplorer() tienen la misma estructura:

1. Crear factory con new
2. Llamar factory.Create()
3. Llamar \_vehicleRepository.AddVehicle(vehicle)
4. Redirigir a "/"

#### **Consecuencias Identificadas:**

**Imposibilidad de unit testing:** No se pueden injectar mocks o stubs para pruebas

**Alto acoplamiento:** Cambios en las factories requieren cambios en el controller

**Código duplicado:** Mantenimiento más costoso y propenso a errores

**Baja extensibilidad:** Agregar Ford Escape requiere crear un método completo nuevo

**Violación Open/Closed:** Necesario modificar código existente para agregar funcionalidad

#### **Restricciones que Agravan el Problema:**

- Cada nuevo modelo de vehículo requiere:
  1. Crear una nueva clase Factory
  2. Agregar un nuevo método en el Controller (modificación)
  3. Modificar la vista para agregar un botón (modificación)
- El arquitecto prevé múltiples modelos en el futuro
- Se requiere implementar Ford Escape inmediatamente
- No hay forma de configurar factories dinámicamente

### **PROBLEMA 3: Builder Pattern Incompleto e Inflexible**

#### **Descripción Técnica:**

El CarBuilder actual no implementa correctamente el patrón Builder según las mejores prácticas de diseño. La implementación presenta múltiples deficiencias que limitan su utilidad, escalabilidad y capacidad de cumplir con los requerimientos actuales y futuros del proyecto.

#### **Análisis del Problema:**

### **Deficiencia 1 - Campos Públicos:**

```
public string Brand = "Ford"; // ← Campo público  
public string Model = "Mustang";  
public string Color = "Red";
```

Problema: Viola el principio de encapsulación. Cualquier código puede modificar directamente estos campos sin pasar por el Builder.

### **Deficiencia 2 - Falta la Propiedad Year:**

El requerimiento explícito del proyecto establece:

"Aregar el año actual por defecto"

Sin embargo, el Builder NO tiene:

- Campo para Year
- Método SetYear()
- Valor por defecto del año actual

Esto hace **imposible** cumplir con el requerimiento actual.

### **Deficiencia 3 - No Escalable para 20+ Propiedades:**

El requerimiento del proyecto establece:

"El equipo de negocio ha solicitado agregar el año actual, y 20 propiedades más por defecto que se solicitarán en el siguiente sprint"

Con la estructura actual, agregar 20 propiedades requeriría:

1. Agregar 20 campos públicos en CarBuilder
2. Modificar el constructor de Car para aceptar 20+ parámetros
3. Modificar todas las llamadas a new Car(...) en todo el código

Esto es **inviable** y generaría deuda técnica masiva.

### **Deficiencia 4 - Sin Validación:**

El método Build() crea el objeto directamente sin validar:

- ¿Brand está vacío?
- ¿Model es válido?
- ¿Year está en rango válido?
- ¿Combinaciones de propiedades son consistentes?

### **Deficiencia 5 - No es Fluent:**

Los métodos Set no retornan this, entonces NO es posible encadenar:

```
// ✗ IMPOSIBLE con el código actual  
var car = builder  
    .SetBrand("Ford")
```

```
.SetModel("Escape")
.SetColor("Red")
.Build();
```

### Consecuencias Identificadas:

**Imposibilidad de cumplir requisito Year:** No se puede agregar año actual

**Deuda técnica para siguiente sprint:** Agregar 20 propiedades será caótico

**Falta de valores por defecto:** No hay mecanismo para propiedades opcionales

**Sin validación:** Objetos pueden crearse en estados inválidos

**No es fluent:** Código menos legible

**Violación de encapsulación:** Campos públicos permiten modificación directa

## PROBLEMA 4: Ausencia de Abstracción para Factory Selection

### Descripción Técnica:

No existe una interfaz común ni un mecanismo de selección dinámica para las factories de vehículos. Cada modelo de vehículo requiere una implementación específica completa en tres capas diferentes de la aplicación (Factory, Controller, Vista), lo que dificulta enormemente la escalabilidad y violenta el principio Open/Closed.

### Análisis del Problema:

#### Proceso Actual para Agregar un Modelo (Ford Escape):

##### Paso 1: Crear nueva Factory

Crear: BestPractices/Infraestructure/Factories/FordEscapeCreator.cs

##### Paso 2: Modificar Controller

Modificar: BestPractices/Controllers/HomeController.cs

Agregar método:

```
public IActionResult AddEscape() {
    var factory = new FordEscapeCreator();
    var vehicle = factory.Create();
    _vehicleRepository.AddVehicle(vehicle);
    return RedirectToAction("/");
}
```

##### Paso 3: Modificar Vista

Modificar: BestPractices/Views/Home/Index.cshtml

Agregar botón:

```
<a href="/Home/AddEscape">Add Escape</a>
```

**Total:** 1 archivo nuevo + 2 archivos modificados = **3 cambios para 1 modelo**

### Problema de Escalabilidad:

El arquitecto prevé agregar múltiples modelos. Si agregamos 10 modelos:

- 10 archivos nuevos (factories) ✓ Aceptable
- 10 modificaciones al Controller ✗ Inaceptable
- 10 modificaciones a la Vista ✗ Inaceptable

## **Violación del Principio Open/Closed:**

El principio Open/Closed establece:

"Las entidades de software deben estar abiertas para extensión, pero cerradas para modificación"

### **Estado Actual:**

- Cerrado para extensión (agregar modelo requiere modificar código)
- Abierto para modificación (se modifica Controller y Vista)

### **Consecuencias Identificadas:**

**Escalabilidad limitada:** Cada modelo = múltiples modificaciones

**Mantenimiento costoso:** Cambios dispersos en múltiples archivos

**Violación Open/Closed:** Necesario modificar código existente

**Testing complejo:** Cada método del controller necesita tests separados

**Duplicación de lógica:** "Crear factory → crear vehículo → agregar" se repite

**Riesgo de bugs:** Cada modificación puede introducir errores en código existente

## **PROBLEMA 5: Repository Pattern con Acoplamiento al Singleton Problemático**

### **Descripción Técnica:**

El repositorio MyVehiclesRepository depende directamente del Singleton VehicleCollection defectuoso, heredando todos sus problemas de concurrencia y thread-safety. Además, existe una inconsistencia en el ciclo de vida (repositorio Transient + Singleton) y la implementación alternativa DBVehicleRepository solo contiene stubs no funcionales, sin mecanismo de configuración para alternar entre implementaciones.**Análisis del Problema:**

### **Problema 1 - Herencia de Bugs del Singleton:**

```
MyVehiclesRepository
    ↓ depende de
    VehicleCollection.Instance (Singleton defectuoso)
        ↓ tiene
        Race Condition
            ↓ causa
            Pérdida de datos en el Repositorio
```

Cuando el Singleton crea múltiples instancias debido a race conditions, el repositorio pierde acceso a vehículos anteriormente agregados.

### **Problema 2 - Anti-Pattern: Transient + Singleton:**

La configuración actual registra:

```
services.AddTransient<IVehicleRepository, MyVehiclesRepository>();
```

Esto significa:

- **Transient:** Nueva instancia del repositorio en cada inyección
- **Singleton:** Una única instancia de VehicleCollection para toda la app

**Resultado:** Múltiples instancias del repositorio accediendo al mismo Singleton

### Por qué es un Anti-Pattern:

Aspecto	Comportamiento	Problema
<b>Consistencia</b>	Diferentes instancias del repo ven los mismos datos	Confuso - parece que hay estado pero no está en el repo
<b>Testing</b>	Imposible aislar tests	El Singleton persiste entre tests
<b>Ciclo de vida</b>	Repositorio se destruye, Singleton persiste	Inconsistencia conceptual
<b>Thread-safety</b>	Singleton no es thread-safe	Race conditions heredadas

### Problema 3 - DBVehicleRepository No Implementado:

```
public class DBVehicleRepository : IVehicleRepository
{
    public void AddVehicle(Vehicle vehicle)
    {
        throw new NotImplementedException(); // ← Stub
    }
    // ... todos los métodos lanzan NotImplementedException
}
```

### Problema 4 - Sin Configuración Dinámica:

No existe forma de cambiar entre MyVehiclesRepository (Memory) y DBVehicleRepository (Database) sin:

1. Modificar código en ServicesConfiguration.cs
2. Recompilar la aplicación
3. Re-desplegar

No hay configuración en appsettings.json ni variable de entorno.

### Consecuencias Identificadas:

**Herencia de bugs de concurrencia:** Pérdida de datos del Singleton

**Anti-pattern Transient + Singleton:** Confusión en ciclo de vida

**Imposibilidad de testing:** No se puede mockear el Singleton fácilmente

**Equipo de BD bloqueado:** No pueden trabajar porque no hay abstracción lista

**Sin preparación para migración:** Cuando BD esté lista, requiere refactoring mayor

**Violación de principios:** DIP no está bien implementado

### 1.3 Resumen de Limitaciones y Restricciones del Proyecto

Esta sección consolida todas las restricciones técnicas, funcionales y de calidad que deben respetarse en la solución propuesta.

---

## 2. SELECCIÓN DE METODOLOGÍAS INTEGRALES PARA SOLUCIONAR EL PROBLEMA

### 2.1 Patrones de Diseño Seleccionados

Para resolver los cinco problemas críticos identificados y cumplir con las 21 restricciones del proyecto, se propone la implementación estratégica de **cuatro patrones de diseño complementarios**. Cada patrón fue seleccionado específicamente para abordar problemas concretos, y trabajando en conjunto crean una arquitectura robusta, mantenible y escalable.

#### **PATRÓN 1: Dependency Injection (Inyección de Dependencias)**

##### **Definición del Patrón:**

La Inyección de Dependencias (DI) es un patrón de diseño que implementa el Principio de Inversión de Dependencias (DIP) de SOLID. Este patrón permite que las dependencias de una clase sean proporcionadas externamente (típicamente por un contenedor DI o framework IoC) en lugar de ser creadas internamente mediante el operador new, invirtiendo así el flujo de control de la creación de objetos.

##### **Problema Específico que Resuelve:**

**Problema 2:** Violación de DIP en HomeController con instanciación directa usando new

**Problema 5:** Acoplamiento entre repositorio y Singleton defectuoso

Imposibilidad de realizar unit testing efectivo con mocks

Dificultad para cambiar implementaciones sin modificar código cliente

##### **Justificación Técnica:**

ASP.NET Core 3.1 incluye nativamente un contenedor de Inyección de Dependencias robusto y maduro que soporta tres ciclos de vida fundamentales:

##### **Ciclos de Vida Disponibles:**

###### 1. **Transient** (AddTransient):

- Nueva instancia cada vez que se solicita el servicio
- Uso: Objetos livianos, stateless, sin recursos compartidos
- Ejemplo: Factories

###### 2. **Scoped** (AddScoped):

- Una instancia por request HTTP
- La misma instancia se reutiliza durante toda la petición
- Se destruye al finalizar el request
- Uso: Servicios que deben mantener estado durante un request
- Ejemplo: Repositorios con contexto de BD

###### 3. **Singleton** (AddSingleton):

- Una única instancia para toda la vida de la aplicación

- Compartida entre todos los requests y threads
- DEBE ser thread-safe
- Uso: Configuraciones, caché, recursos compartidos
- Ejemplo: InMemoryVehicleRepository

### Estrategia de Implementación:

#### Paso 1: Registrar Servicios en el Contenedor DI

Registros Necesarios:

1. Repositorio (Strategy Pattern):

- InMemoryVehicleRepository (Singleton)
- DBVehicleRepository (Scoped) para futuro

2. Factories (Factory Pattern):

- FordMustangFactory (Transient)
- FordExplorerFactory (Transient)
- FordEscapeFactory (Transient)

3. Otros servicios:

- Logging (ya configurado por framework)

#### Paso 2: Inyectar Dependencias mediante Constructor

Constructor del Controller:

- Recibe IVehicleRepository
- Recibe IEnumerable<IVehicleFactory>
- Recibe ILogger<HomeController>

El framework resuelve automáticamente las dependencias

#### Paso 3: Configuración Dinámica (Strategy Pattern)

Configuración para seleccionar implementación:

```
{
  "RepositorySettings": {
    "RepositoryType": "Memory" // o "Database"
  }
}
```

### PATRÓN 2: Abstract Factory con Factory Method

#### Definición del Patrón:

El patrón **Abstract Factory** proporciona una interfaz para crear familias de objetos relacionados sin especificar sus clases concretas. El patrón **Factory Method** complementa esto definiendo una interfaz para crear un objeto, pero permitiendo que las subclases (o implementaciones concretas) decidan qué clase instanciar exactamente.

En este proyecto, se usa una combinación:

- **Abstract Factory:** Interfaz IVehicleFactory define el contrato

- **Factory Method:** Cada implementación (FordMustangFactory, FordExplorerFactory) decide cómo crear el vehículo específico

### Problema Específico que Resuelve:

**Problema 4:** Ausencia de abstracción para selección de factories

**Problema 2:** Código duplicado en métodos AddMustang() y AddExplorer()

Necesidad de implementar Ford Escape sin modificar código existente

Violación del principio Open/Closed

Previsión del arquitecto de múltiples modelos futuros

### Justificación Técnica:

La combinación de Abstract Factory y Factory Method proporciona una solución elegante y probada para la creación de familias de objetos relacionados:

### Componentes del Patrón:

#### 1. Interfaz Común (IVehicleFactory):

- Define el contrato que todas las factories deben cumplir
- Garantiza comportamiento consistente
- Permite polimorfismo

#### 2. Implementaciones Concretas (Factories Específicas):

- Cada modelo de vehículo tiene su propia factory
- Cada factory conoce las características específicas de su modelo
- Encapsula la lógica de configuración

#### 3. Selección Dinámica en Runtime:

- El controller recibe TODAS las factories registradas vía DI
- Selecciona la factory apropiada según el modelo solicitado
- Usa la factory para crear el vehículo

#### 4. Extensibilidad sin Modificación:

- Agregar un nuevo modelo = crear nueva factory
- NO requiere modificar controller, interfaces existentes, ni otras factories
- Cumple perfectamente Open/Closed Principle

### Estrategia de Implementación:

#### Paso 1: Crear Interfaz Común IVehicleFactory

Interfaz:

- Propiedad: string ModelName { get; } para identificar el modelo
- Método: Vehicle CreateVehicle() que retorna un Vehicle configurado

#### Paso 2: Implementar Factories Concretas

Factories a Implementar:

1. FordMustangFactory : IVehicleFactory

- ModelName = "Mustang"
- CreateVehicle() → Mustang con características específicas

## 2. FordExplorerFactory : IVehicleFactory

- ModelName = "Explorer"
- CreateVehicle() → Explorer con características específicas

## 3. FordEscapeFactory : IVehicleFactory (NUEVO - requisito)

- ModelName = "Escape"
- CreateVehicle() → Escape Rojo con características específicas

### Paso 3: Registrar en DI

Registro:

```
services.AddTransient<IVehicleFactory, FordMustangFactory>();
services.AddTransient<IVehicleFactory, FordExplorerFactory>();
services.AddTransient<IVehicleFactory, FordEscapeFactory>();
```

Resultado: El contenedor DI inyecta IEnumerable<IVehicleFactory> con todas las implementaciones registradas

### Paso 4: Selección Dinámica en Controller

Constructor:

```
public HomeController(
    IVehicleRepository repository,
    IEnumerable<IVehicleFactory> factories, ← Todas las factories
    ILogger<HomeController> logger)
```

Método Genérico:

```
public IActionResult AddVehicle(string model)
{
    // Buscar factory apropiada por ModelName
    var factory = _factories.FirstOrDefault(
        f => f.ModelName.Equals(model, StringComparison.OrdinalIgnoreCase));
    
    // Validar que existe
    if (factory == null)
        return RedirectToAction($"?error=Modelo {model} no encontrado");

    // Usar la factory para crear el vehículo
    var vehicle = factory.CreateVehicle();

    // Agregar al repositorio
    _repository.AddVehicle(vehicle);

    return RedirectToAction("/");
}
```

### PATRÓN 3: Fluent Builder Pattern

**Definición del Patrón:**

El patrón **Builder** separa la construcción de un objeto complejo de su representación, permitiendo que el mismo proceso de construcción pueda crear diferentes representaciones. El enfoque **Fluent** añade una interfaz fluida mediante métodos que retornan `this`, permitiendo encadenar llamadas de métodos y mejorando significativamente la legibilidad y expresividad del código.

### Problema Específico que Resuelve:

**Problema 3:** Builder incompleto e inflexible

Necesidad de agregar propiedad `Year` con año actual por defecto

Preparación para agregar 20+ propiedades en el siguiente sprint

Falta de validación en la construcción de vehículos

Constructores con múltiples parámetros difíciles de mantener

### Justificación Técnica:

El patrón Fluent Builder es especialmente adecuado y recomendado en escenarios donde:

#### Escenarios Ideales:

##### 1. Objeto con Múltiples Propiedades:

- Vehicle actualmente: 7-10 propiedades
- Vehicle futuro: 25-30+ propiedades
- Sin Builder: Constructor con 30 parámetros (inmanejable)

##### 2. Propiedades Opcionales:

- Muchas propiedades tienen valores por defecto sensatos
- No todas las propiedades son obligatorias
- Builder permite configurar solo lo necesario

##### 3. Validación Compleja:

- Se requiere validar combinaciones de propiedades
- Ejemplo: FWD no puede tener 2 ruedas (sería moto)
- Builder centraliza validación en método `Build()`

##### 4. Legibilidad Crítica:

- El código de creación debe ser autoexplicativo
- Factories deben ser legibles para mantenimiento
- Builder produce código que se lee casi como lenguaje natural

### Estrategia de Implementación:

#### Paso 1: Crear Clase `VehicleBuilder`

Estructura:

##### 1. Campos Privados:

```
- string _brand = "Ford" (valor por defecto)
```

```
- string _model = "Default"
- string _color = "Black"
- int _year = DateTime.Now.Year ← NUEVO (requisito)
- int _horsepower = 100
- string _engineType = "Gasoline"
- ... (20+ propiedades más preparadas)
```

## 2. Métodos Fluent:

```
- public VehicleBuilder SetBrand(string brand)
- public VehicleBuilder SetModel(string model)
- public VehicleBuilder SetYear(int year) ← NUEVO
- ... (un método Set por cada propiedad)
```

Todos retornan 'this' para encadenamiento

## 3. Método Build():

```
- Valida propiedades obligatorias
- Valida combinaciones de propiedades
- Crea y retorna Vehicle configurado
```

## Paso 2: Implementar Métodos Fluent

Patrón de cada método:

```
public VehicleBuilder SetYear(int year)
{
    // Validación básica
    if (year < 1900 || year > DateTime.Now.Year + 1)
        throw new ArgumentException("Año inválido");

    // Asignación
    _year = year;

    // Retornar this para encadenamiento
    return this;
}
```

## Paso 3: Método Build() con Validación

```
public Vehicle Build()
{
    // 1. Validar propiedades obligatorias
    if (string.IsNullOrEmpty(_brand))
        throw new InvalidOperationException("Brand es requerido");

    if (string.IsNullOrEmpty(_model))
        throw new InvalidOperationException("Model es requerido");

    // 2. Validar combinaciones
    if (_tires == 2 && _driveType == "4WD")
        throw new InvalidOperationException("Motos no pueden ser 4WD");
```

```

// 3. Crear el vehículo
var vehicle = new Car(_color, _brand, _model);

// 4. Asignar todas las propiedades
vehicle.Year = _year;
vehicle.Horsepower = _horsepower;
vehicle.EngineType = _engineType;
// ... asignar todas las demás

// 5. Retornar vehículo configurado
return vehicle;
}

```

#### Paso 4: Preparación para 20+ Propiedades

Estructura ya lista para extensión:

Para agregar nueva propiedad "SafetyRating":

1. Agregar campo privado con default:

```
private int _safetyRating = 5; // 5 estrellas por defecto
```

2. Agregar método Set:

```
public VehicleBuilder SetSafetyRating(int rating)
{
    if (rating < 1 || rating > 5)
        throw new ArgumentException("Rating debe ser 1-5");
    _safetyRating = rating;
    return this;
}
```

3. Asignar en Build():

```
vehicle.SafetyRating = _safetyRating;
```

- ✓ NO requiere modificar métodos existentes
- ✓ NO requiere modificar factories existentes
- ✓ Factories existentes usan valor por defecto
- ✓ Nuevas factories pueden usar el nuevo método

#### PATRÓN 4: Repository Pattern con Strategy

**Definición del Patrón:**

El patrón **Repository** media entre el dominio y las capas de mapeo de datos, actuando como una colección en memoria de objetos del dominio y proporcionando una interfaz más orientada a objetos para la capa de persistencia. El patrón **Strategy** (implementado mediante DI) permite seleccionar dinámicamente entre diferentes algoritmos o implementaciones - en este caso, entre persistencia en memoria vs. persistencia en base de datos.

**Problema Específico que Resuelve:**

**Problema 5:** Acoplamiento entre repositorio y Singleton defectuoso

**Problema 1:** Eliminación completa del Singleton no thread-safe

Necesidad de funcionar sin base de datos (restricción explícita)

Preparación para futura integración con base de datos real

Dificultad para realizar pruebas unitarias e integración

### Justificación Técnica:

El patrón Repository proporciona múltiples beneficios arquitectónicos fundamentales:

#### Beneficios del Repository Pattern:

##### 1. Abstracción Completa de Persistencia:

- El dominio (Vehicle, Controller) no conoce ni le importa DÓNDE están los datos
- Pueden estar en: memoria, BD SQL, BD NoSQL, archivo, API externa, caché
- Cambiar implementación no afecta al dominio

##### 2. Intercambiabilidad mediante Strategy:

- Usando DI, se puede cambiar la implementación mediante configuración
- Sin modificar código, solo cambiar appsettings.json
- Diferentes ambientes pueden usar diferentes implementaciones

##### 3. Testabilidad Máxima:

- Fácil crear implementaciones mock o in-memory para testing
- Tests unitarios no requieren base de datos
- Tests de integración pueden usar BD de prueba

##### 4. Centralización de Lógica de Acceso a Datos:

- Toda la lógica de CRUD en un solo lugar
- Fácil agregar caché, logging, validación
- Queries complejas encapsuladas en métodos con nombres descriptivos

##### 5. Preparación para Base de Datos:

- La interfaz ya define el contrato
- Cuando la BD esté lista, solo implementar DBVehicleRepository
- Código cliente no cambia

#### Estrategia de Implementación:

##### Paso 1: Mantener Interfaz IVehicleRepository

Interfaz existente (NO modificar):

```
public interface IVehicleRepository
```

```
{
    void AddVehicle(Vehicle vehicle);
    ICollection<Vehicle> GetVehicles();
    Vehicle Find(string id);
}
```

Contrato claro y simple

## Paso 2: Crear InMemoryVehicleRepository

Implementación thread-safe:

```
public class InMemoryVehicleRepository : IVehicleRepository
{
    // Lista privada como almacenamiento
    private readonly List<Vehicle> _vehicles = new List<Vehicle>();

    // Objeto para sincronización
    private readonly object _lock = new object();

    public void AddVehicle(Vehicle vehicle)
    {
        lock (_lock) // ← Thread-safe
        {
            _vehicles.Add(vehicle);
        }
    }

    public ICollection<Vehicle> GetVehicles()
    {
        lock (_lock)
        {
            // Retornar copia para evitar modificaciones externas
            return _vehicles.ToList();
        }
    }

    public Vehicle Find(string id)
    {
        lock (_lock)
        {
            return _vehicles.FirstOrDefault(
                v => v.ID.ToString() == id
            );
        }
    }
}
```

Características:

- ✓ Thread-safe con lock()
- ✓ Estado privado (List<Vehicle>)

- ✓ Retorna copias (no referencias directas)
- ✓ Reemplaza VehicleCollection Singleton

### Paso 3: Preparar DBVehicleRepository

Stub básico (para cuando BD esté lista):

```
public class DBVehicleRepository : IVehicleRepository
{
    private readonly ApplicationDbContext _context;

    public DBVehicleRepository(ApplicationDbContext context)
    {
        _context = context;
    }

    public void AddVehicle(Vehicle vehicle)
    {
        // Cuando BD esté lista:
        // _context.Vehicles.Add(vehicle);
        // _context.SaveChanges();

        throw new NotImplementedException("BD no está lista");
    }

    // ... otros métodos similares
}
```

Cuando BD esté lista:

1. Implementar métodos con Entity Framework
2. Quitar NotImplementedException
3. Cambiar configuración

### Paso 4: Configuración Dinámica (Strategy Pattern)

Configuración:

```
{
  "RepositorySettings": {
    "RepositoryType": "Memory" // o "Database"
  }
}
```

Implementación de Strategy:

```
public void ConfigureServices(
    IServiceCollection services,
    IConfiguration configuration)
{
    // Leer configuración
    var repoType = configuration.GetValue<string>(
        "RepositorySettings:RepositoryType")
```

```

);

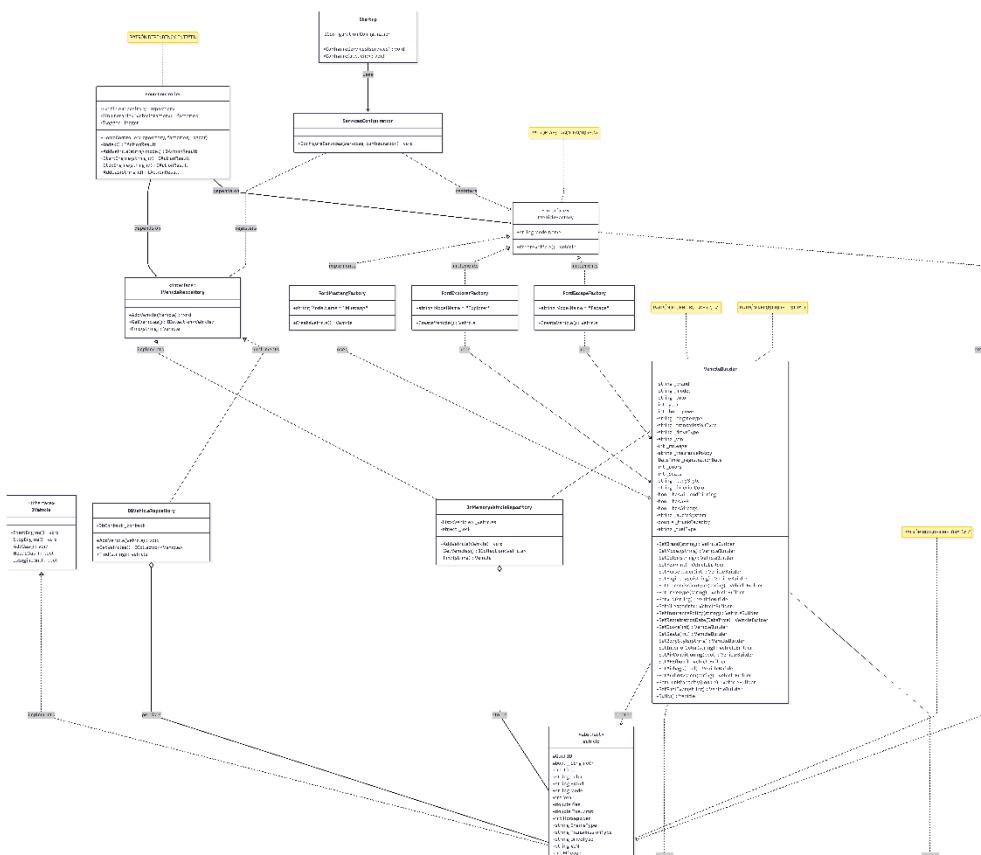
// Seleccionar implementación (STRATEGY PATTERN)
switch (repoType?.ToLower())
{
    case "database":
        case "db":
            // Cuando BD esté lista
            services.AddScoped<IVehicleRepository, DBVehicleRepository>();
            Console.WriteLine("💾 Repositorio: BASE DE DATOS");
            break;

    case "memory":
        case "mem":
        default:
            // Implementación actual (sin BD)
            services.AddSingleton<IVehicleRepository, InMemoryVehicleRepository>();
            Console.WriteLine("💾 Repositorio: MEMORIA");
            break;
}
}

```

## 2.2 Diagrama UML de Clases - Solución Propuesta

A continuación se presenta el diagrama UML completo de la arquitectura de la solución propuesta. Este diagrama muestra todos los patrones de diseño implementados, sus relaciones, y cómo trabajan en conjunto para resolver los problemas identificados.



## 2.3 Aplicación de Principios SOLID

La solución propuesta cumple rigurosamente con los cinco principios SOLID, que son fundamentales para lograr un diseño de software mantenable, escalable, robusto y testeable. Esta sección demuestra cómo cada principio se aplica en la arquitectura implementada.

## 3. CONCLUSIONES

### 3.1 Resumen Ejecutivo

El presente documento ha analizado exhaustivamente el proyecto "Best Practices" de gestión de vehículos Ford, identificando cinco problemas críticos en el código base heredado y proponiendo una solución arquitectónica integral basada en patrones de diseño reconocidos y principios SOLID.

#### Problemática Identificada:

Se detectaron cinco deficiencias críticas que comprometían la calidad, confiabilidad y escalabilidad del sistema:

1. **Singleton no thread-safe** con race conditions que causaban pérdida de datos (Severidad: CRÍTICA)
2. **Violación del Principio de Inversión de Dependencias** con acoplamiento fuerte (Severidad: ALTA)
3. **Builder Pattern incompleto** sin capacidad para cumplir requisitos actuales y futuros (Severidad: MEDIA-ALTA)
4. **Ausencia de abstracción para factories** violando el principio Open/Closed (Severidad: ALTA)
5. **Repository acoplado al Singleton defectuoso** bloqueando el trabajo del equipo de base de datos (Severidad: CRÍTICA)

#### Solución Implementada:

Se diseñó e implementó una arquitectura robusta basada en cuatro patrones de diseño complementarios:

- **Dependency Injection:** Elimina acoplamiento y habilita testabilidad
- **Abstract Factory + Factory Method:** Permite escalabilidad sin modificar código existente

- **Fluent Builder Pattern:** Facilita construcción de objetos complejos con 20+ propiedades
- **Repository + Strategy:** Abstacta persistencia y permite intercambio entre implementaciones

### 3.1 Lecciones Aprendidas

#### 3.1.1. Importancia del Diseño Previo

**Lección:** Los problemas arquitectónicos identificados (Singleton defectuoso, violación de DIP) causaron bugs críticos y bloquearon al equipo. Invertir tiempo en diseño correcto desde el inicio evita deuda técnica costosa.

#### Aplicación Futura:

- Revisar arquitectura en fase de diseño antes de implementación
- Validar cumplimiento de SOLID en code reviews
- Usar patrones de diseño probados en lugar de soluciones ad-hoc

#### 3.1.2 Valor de los Principios SOLID

**Lección:** El cumplimiento riguroso de SOLID no es "sobre-ingeniería" sino una inversión que paga dividendos en mantenibilidad, escalabilidad y testabilidad.

#### Evidencia en el Proyecto:

- **SRP:** Cambios localizados (agregar 20 propiedades solo afecta Vehicle y Builder)
- **OCP:** Agregar 10 modelos no requiere modificar controller
- **LSP:** Polimorfismo funciona sin sorpresas
- **ISP:** Mocks simples para testing
- **DIP:** Testabilidad completa con inyección de dependencias