

Patrón Builder - Sistema de Automóviles con Atributos y Reflexión

Ejercicio 1 - Arquitectura de Software (Versión Mejorada)

Descripción del Escenario

Contexto

Imagina que estás desarrollando una aplicación para una compañía automotriz que permite a los clientes personalizar y ordenar un automóvil. Un objeto Automóvil puede tener muchas configuraciones opcionales que van desde características básicas hasta opciones de lujo.

Características del Automóvil

El sistema debe permitir configurar:

Características Básicas (Obligatorias): - Marca y modelo - Tipo de automóvil (Básico, Familiar, Deportivo, De Lujo, SUV, Pickup) - Año de fabricación - Tipo de motor (Básico, Intermedio, Potente, Deportivo, Eléctrico) - Color del vehículo - Tipo de llantas - Tipo de transmisión (Manual, Automática, CVT, Semi-automática) - Tipo de faros (Halógeno, Xenón, LED, Láser) - Tipo de tapicería (Tela, Vinilo, Cuero, Alcantara)

Características Opcionales (Con Atributos): - Sistema de sonido personalizado - Detalles de interiores - Volante personalizado - Rines personalizados - Techo solar
[CaracteristicaOpcional("Techo Solar")] - Sistema GPS
[CaracteristicaOpcional("GPS")] - Aire acondicionado
[CaracteristicaOpcional("Aire Acondicionado")] - Cámara de reversa
[CaracteristicaOpcional("Cámara Reversa")] - Sensores delanteros y traseros
[CaracteristicaOpcional("Sensores Delanteros/Traseros")] - Vidrios eléctricos
[CaracteristicaOpcional("Vidrios Eléctricos")] - Espejos eléctricos
[CaracteristicaOpcional("Espejos Eléctricos")] - Baúl automático
[CaracteristicaOpcional("Baúl Automático")] - Polarizado de vidrios
[CaracteristicaOpcional("Polarizado")] - Frenos ABS
[CaracteristicaOpcional("Frenos ABS")] - Control de estabilidad
[CaracteristicaOpcional("Control de Estabilidad")] - Airbags laterales
[CaracteristicaOpcional("Airbags Laterales")] - Sistema de alarma
[CaracteristicaOpcional("Alarma")] - Bloqueo central
[CaracteristicaOpcional("Bloqueo Central")] - Pantalla Android Auto
[CaracteristicaOpcional("Pantalla Android Auto")] - Parlantes extra
[CaracteristicaOpcional("Parlantes Extra")] - DVD para asientos traseros

```
[CaracteristicaOpcional("DVD Para Atrás")] - Gancho de remolque
[CaracteristicaOpcional("Gancho de Remolque")] - Parrilla de techo
[CaracteristicaOpcional("Parrilla de Techo")] - Portavasos
[CaracteristicaOpcional("Portavasos")] - Soporte para celular
[CaracteristicaOpcional("Soporte para Celular")] - Luces interiores LED
[CaracteristicaOpcional("Luces Interiores LED")] - Sistema de sonido "tumba carro" [CaracteristicaOpcional("Sonido Tumba Carro")]
```

⚠ Problema Identificado

Constructor Telescópico

Crear un objeto Automóvil con múltiples configuraciones puede llevar a:

1. **Constructores con muchos parámetros** (el infame "constructor telescópico")
2. **Múltiples constructores sobrecargados** para diferentes combinaciones
3. **Dificultad de mantenimiento** y legibilidad del código
4. **Propensión a errores** al pasar parámetros en orden incorrecto
5. **Dificultad para omitir parámetros opcionales** sin crear subclases
6. **Código repetitivo** en métodos como GetOpcionesActivas()
7. **Mantenimiento manual** de listas de características

Ejemplo del Problema (Antes de las Mejoras)

```
// Constructor telescópico problemático
public Automovil(string marca, string modelo, string tipo, int anio,
                  string motor, string color, string llantas, string trans
mission,
                  string faros, string tapicería, string sonido, string in
teriores,
                  bool techoSolar, bool gps, bool aireAcondicionado,
                  bool camaraReversa, bool sensoresDelanteros, bool sensor
esTraseros,
                  bool vidriosElectricos, bool espejosElectricos, bool bau
lAutomatico,
                  bool polarizado, bool frenosABS, bool controlEstabilidad,
                  bool airbagsLaterales, bool alarma, bool bloqueoCentral,
                  bool pantallaAndroidAuto, bool parlantesExtra, bool dvdP
araAtras,
                  bool ganchoRemolque, bool parrillaTecho, bool portavasos,
                  bool soporteCelular, string rinesPersonalizados,
                  bool lucesInterioresLED, bool sonidoTumbaCarro)
{
    // Implementación compleja y propensa a errores
}
```

```
// Código repetitivo en GetOpcionesActivas()
private List<string> GetOpcionesActivas()
{
    var opciones = new List<string>();

    if (TechoSolar) opciones.Add("Techo Solar");
    if (GPS) opciones.Add("GPS");
    if (AireAcondicionado) opciones.Add("Aire Acondicionado");
    if (CamaraReversa) opciones.Add("Cámara Reversa");
    // ... 22 líneas más de ifs repetitivos

    return opciones;
}
```

Solución Propuesta

Patrón Builder + Atributos + Reflexión

La solución combina el **Patrón Builder** con **atributos personalizados** y **reflexión** para crear un sistema más robusto y mantenible:

1. Permite construir objetos complejos paso a paso
2. Evita el constructor telescopico
3. Separa la construcción de la representación
4. Facilita la creación de diferentes configuraciones
5. Permite omitir parámetros opcionales fácilmente
6.  Automatiza el manejo de características opcionales
7.  Elimina código repetitivo con reflexión
8.  Facilita la adición de nuevas características

Estructura de la Solución Mejorada

- **Automovil**: Producto complejo con atributos personalizados
 - **IAutomovilBuilder**: Interfaz que define los pasos de construcción
 - **AutomovilBuilder**: Implementación mejorada con diccionarios dinámicos
 - **Director**: Maneja configuraciones predefinidas con aplicación automática
 - **Atributos Personalizados**: Para características opcionales y configuraciones
 - **Enumeraciones**: Con descripciones legibles para usuarios
 - **EnumHelper**: Utilidades para trabajar con enums
-

🔍 Análisis del Patrón Builder Mejorado

Clasificación

Según el catálogo de [Refactoring.Guru](#):

- **Tipo:** Patrón Creacional
- **Propósito:** Construir objetos complejos paso a paso
- **Aplicabilidad:** Cuando un objeto tiene muchas configuraciones opcionales
- **Mejora:** Con atributos personalizados y reflexión para automatización

Componentes del Patrón Mejorado

1. Producto (Product)

- **Automovil:** Objeto complejo con propiedades marcadas con atributos
- **Inmutable:** Una vez creado, no se puede modificar
- **Configurable:** Permite diferentes combinaciones de características
- **Automatizado:** GetOpcionesActivas() usa reflexión para listar características

2. Builder (Constructor)

- **IAutomovilBuilder:** Interfaz que define los pasos de construcción
- **AutomovilBuilder:** Implementación mejorada con diccionarios dinámicos
- **Métodos encadenables:** Permiten construcción fluida
- **Método Build():** Construye usando reflexión para propiedades init-only
- **Reset() automático:** Usa reflexión para resetear todas las propiedades

3. Director (Director)

- **Director:** Maneja configuraciones predefinidas
- **Métodos especializados:** Para diferentes tipos de automóviles
- **Aplicación automática:** Usa reflexión para aplicar características por tipo
- **Configuraciones centralizadas:** Todas las características por tipo en un lugar

4. Atributos Personalizados (Nuevos)

- **CaracteristicaOpcionalAttribute:** Marca características opcionales
- **DescripcionEnumAttribute:** Proporciona descripciones legibles para enums
- **PropiedadBuilderAttribute:** Para automatizar el builder (futuro)
- **ConfiguracionVehiculoAttribute:** Para configuraciones automáticas (futuro)

5. Enumeraciones Mejoradas

- **Type Safety:** Previene errores de tipo
- **Opciones predefinidas:** Valores válidos para cada característica
- **Descripciones legibles:** Para mostrar al usuario final
- **Utilidades:** EnumHelper para trabajar con descripciones

6. Clase de Utilidades (Nueva)

- **EnumHelper**: Métodos estáticos para trabajar con enums y sus descripciones
 - **Conversión bidireccional**: Entre valores de enum y descripciones
 - **Funcionalidad reutilizable**: Para toda la aplicación
-



Mejoras Implementadas con Atributos y Reflexión

1. Atributos Personalizados

CaracteristicaOpcionalAttribute

```
[AttributeUsage(AttributeTargets.Property)]
public class CaracteristicaOpcionalAttribute : Attribute
{
    public string Nombre { get; }
    public CaracteristicaOpcionalAttribute(string nombre) => Nombre = nombre;
}

// Uso en Automovil
[CaracteristicaOpcional("Techo Solar")]
public bool TechoSolar { get; init; }

[CaracteristicaOpcional("GPS")]
public bool GPS { get; init; }
```

DescripcionEnumAttribute

```
[AttributeUsage(AttributeTargets.Property | AttributeTargets.Field)]
public class DescripcionEnumAttribute : Attribute
{
    public string Descripcion { get; }
    public DescripcionEnumAttribute(string descripcion) => Descripcion = descripcion;
}

// Uso en Enums
public enum TipoAutomovil
{
    [DescripcionEnum("Automóvil Básico")] Basico,
    [DescripcionEnum("Automóvil Familiar")] Familiar,
    [DescripcionEnum("Automóvil Deportivo")] Deportivo,
    [DescripcionEnum("Automóvil de Lujo")] DeLujo,
    [DescripcionEnum("SUV")] SUV,
    [DescripcionEnum("Pickup")] Pickup
}
```

2. Reflexión para Automatización

GetOpcionesActivas() Automatizado

```
// Antes: 25 líneas de ifs repetitivos
private List<string> GetOpcionesActivas()
{
    var opciones = new List<string>();
    var props = GetType().GetProperties(BindingFlags.Public | BindingFlags.Instance);

    foreach (var prop in props)
    {
        if (prop.PropertyType == typeof(bool) &&
            prop.GetCustomAttribute<CaracteristicaOpcionalAttribute>() is
{ } attr &&
            (bool)prop.GetValue(this)!)
        {
            opciones.Add(attr.Nombre);
        }
    }
    return opciones;
}
```

Builder con Diccionarios Dinámicos

```
public class AutomovilBuilder : IAutomovilBuilder
{
    private readonly Dictionary<string, object> _propiedades = new();

    public IAutomovilBuilder Reset()
    {
        _propiedades.Clear();

        // Usar reflexión para resetear automáticamente
        var tipoAutomovil = typeof(Automovil);
        var propiedades = tipoAutomovil.GetProperties(BindingFlags.Public
| BindingFlags.Instance);

        foreach (var prop in propiedades)
        {
            if (prop.PropertyType == typeof(bool))
                _propiedades[prop.Name] = false;
            else if (prop.PropertyType == typeof(string))
                _propiedades[prop.Name] = string.Empty;
            else if (prop.PropertyType == typeof(int))
                _propiedades[prop.Name] = 0;
        }
        return this;
    }
}
```

Director con Configuraciones Automáticas

```
public class Director
{
    public IAutomovilBuilder AplicarCaracteristicasPorTipo(TipoAutomovil
tipo)
    {
        var configuraciones = GetConfiguracionesPorTipo(tipo);

        foreach (var caracteristica in configuraciones)
        {
            // Usar reflexión para aplicar características automáticamente
e
            var metodo = _builder.GetType().GetMethod($"Set{caracteristic
a}");
            if (metodo != null)
            {
                metodo.Invoke(_builder, new object[] { true });
            }
        }
        return _builder;
    }
}
```

3. Utilidades para Enums

EnumHelper

```
public static class EnumHelper
{
    public static string GetDescripcion<T>(T enumValue) where T : Enum
    {
        var field = enumValue.GetType().GetField(enumValue.ToString());
        var attribute = field?.GetCustomAttribute<DescripcionEnumAttribut
e>();
        return attribute?.Descripcion ?? enumValue.ToString();
    }

    public static Dictionary<T, string> GetTodasLasDescripciones<T>() whe
re T : Enum
    {
        var resultado = new Dictionary<T, string>();
        var tipo = typeof(T);
        var valores = Enum.GetValues(tipo).Cast<T>();

        foreach (var valor in valores)
        {
            var field = tipo.GetField(valor.ToString());
            var attribute = field?.GetCustomAttribute<DescripcionEnumAttr
ibute>();
            resultado[valor] = attribute?.Descripcion ?? valor.ToString();
        }
    }
}
```

```
        return resultado;
    }
}
```

Diagrama de Clases Actualizado

El diagrama de clases muestra la estructura completa del patrón Builder mejorado con atributos y reflexión:

- **Clases principales:** Automovil, IAutomovilBuilder, AutomovilBuilder, Director, EnumHelper
 - **Atributos personalizados:** CaracteristicaOpcionalAttribute, DescripcionEnumAttribute, etc.
 - **Enumeraciones mejoradas:** Con descripciones legibles
 - **Relaciones:** Implementación, creación, uso y dependencias entre componentes
 -  **Reflexión:** Para automatización de procesos
-

Ejemplos de Uso Actualizados

Ejemplo 1: Construcción Manual (Sin Cambios)

```
IAutomovilBuilder builder = new AutomovilBuilder();
```

```
Automovil autoRenault = builder
    .Reset()
    .SetMarca("Renault")
    .SetModelo("Logan")
    .SetTipo(TipoAutomovil.Familiar)
    .SetAnio(2025)
    .SetMotor(TipoMotor.Basico)
    .SetColor("Negro")
    .SetLlantas("17\" Aleación")
    .SetTransmision(TipoTransmision.Automatica)
    .SetFaros(TipoFaros.Halogeno)
    .SetTapiceria(TipoTapiceria.Vinilo)
    .SetVolante("Cuero")
    .SetSonido("Estándar")
    .SetInteriores("Plástico")
    .SetAireAcondicionado(true)
    .SetGPS(true)
    .SetFrenosABS(true)
    .Build();

Console.WriteLine(autoRenault);
```

Ejemplo 2: Uso con Director (Mejorado)

```
IAutomovilBuilder builder = new AutomovilBuilder();
Director director = new Director(builder);

// Las características se aplican automáticamente según el tipo
Automovil autoDeportivo = director.AutomovilDeportivo().Build();
Console.WriteLine(autoDeportivo);
```

Ejemplo 3: Trabajar con Descripciones de Enums (Nuevo)

```
// Obtener descripción legible
string descripcion = EnumHelper.GetDescripcion(TipoMotor.Electrico);
Console.WriteLine(descripcion); // Output: "Motor Eléctrico"

// Obtener todas las descripciones
var tiposMotor = EnumHelper.GetTodasLasDescripciones<TipoMotor>();
foreach (var tipo in tiposMotor)
{
    Console.WriteLine($"{tipo.Key}: {tipo.Value}");
}
```

Ejemplo 4: Agregar Nueva Característica (Simplificado)

```
// Solo necesitas agregar la propiedad con el atributo
[CaracteristicaOpcional("Nueva Característica")]
public bool NuevaCaracteristica { get; init; }

// Y el método Set en el Builder
public IAutomovilBuilder SetNuevaCaracteristica(bool valor)
{
    _propiedades["NuevaCaracteristica"] = valor;
    return this;
}

// ¡Automáticamente aparece en ToString() cuando esté activa!
```

Comparación Antes vs Después

Aspecto	Antes	Después	Mejora
Líneas de código repetitivo	~200 líneas	~60 líneas	70% reducción
Mantenimiento de características	Manual en múltiples lugares	Automático con atributos	Muy alta
Riesgo de errores	Alto (olvidos en ifs)	Muy bajo (reflexión automática)	Eliminado

Aspecto	Antes	Después	Mejora
Extensibilidad	Difícil (tocar muchos métodos)	Trivial (solo agregar atributo)	Excelente
Legibilidad	Repetitivo y verboso	Limpio y declarativo	Mucho mejor
Experiencia de usuario	Nombres técnicos	Descripciones legibles	Mejorada
Configuraciones automáticas	Manual	Automática por tipo	Automatizada

🔗 Conclusiones

Resumen del Ejercicio Mejorado

El ejercicio demuestra exitosamente la aplicación del **Patrón Builder** combinado con **atributos personalizados y reflexión** para resolver múltiples problemas:

1. **Elimina la complejidad** del constructor telescopico
2. **Mejora significativamente** la legibilidad del código
3. **Proporciona flexibilidad** para diferentes configuraciones
4. **Mantiene la inmutabilidad** de los objetos creados
5. **Facilita el mantenimiento** y extensión del sistema
6. **NEW Automatiza procesos** repetitivos con reflexión
7. **NEW Reduce código** duplicado en un 70%
8. **NEW Mejora la experiencia** de usuario con descripciones legibles

Aplicabilidad del Patrón Mejorado

El **Patrón Builder con Atributos y Reflexión** es especialmente útil cuando:

- Un objeto tiene muchas configuraciones opcionales
- Se necesita crear diferentes variaciones del mismo objeto
- El proceso de construcción es complejo
- Se requiere validación durante la construcción
- **NEW Se necesita automatizar** el manejo de características
- **NEW Se requiere facilidad** para agregar nuevas características
- **NEW Se necesita mejorar** la experiencia del usuario final

Impacto en la Calidad del Código

- **Legibilidad:** Código más claro y expresivo
- **Mantenibilidad:** Fácil modificación y extensión
- **Robustez:** Validación y manejo de errores mejorado
- **Reutilización:** Componentes modulares y reutilizables

-  **Automatización**: Menos código repetitivo y manual
-  **Extensibilidad**: Fácil agregar nuevas funcionalidades
-  **UX**: Mejor experiencia para usuarios finales

Recomendaciones Actualizadas

1. **Usar enums** para opciones predefinidas (Type Safety)
 2. **Implementar validación** robusta en el método Build()
 3. **Documentar** claramente las configuraciones predefinidas
 4. **Considerar** el uso del Director para configuraciones comunes
 5. **Mantener** la inmutabilidad de los objetos producto
 6.  **Usar atributos personalizados** para características opcionales
 7.  **Aprovechar la reflexión** para automatizar procesos repetitivos
 8.  **Proporcionar descripciones legibles** para usuarios finales
 9.  **Centralizar configuraciones** por tipo de objeto
 10.  **Crear utilidades reutilizables** para patrones comunes
-



Referencias

- Refactoring.Guru - Patrón Builder
 - Catálogo de Patrones de Diseño
 - Patrones Creacionales
 -  **Atributos en C#**
 -  **Reflexión en C#**
-