**Specification**. Es un patrón de diseño de software que se utiliza para encapsular la lógica de negocio en un objeto separado, haciéndola reutilizable y componible.

**¿Qué es el patrón Specification?**

En esencia, el patrón Specification permite definir un conjunto de reglas o criterios (la "especificación") de una manera que puede ser:

* **Reutilizable:** Puedes usar la misma especificación en diferentes partes de tu aplicación.
* **Componible:** Puedes combinar múltiples especificaciones para crear reglas más complejas (usando operadores lógicos como AND, OR, NOT).
* **Testeable:** La lógica de las reglas es fácil de testear de forma aislada.
* **Separada del dominio:** Ayuda a mantener la lógica de filtrado o validación fuera de tus entidades de dominio, haciéndolas más limpias y centradas en su responsabilidad principal.

**¿Cómo funciona?**

Normalmente, el patrón Specification implica:

1. **Una interfaz Specification:** Define un método, como isSatisfiedBy(T candidate), que devuelve un booleano indicando si un objeto dado cumple con la especificación.
2. **Implementaciones concretas de Specification:** Cada una encapsula una regla de negocio específica. Por ejemplo, una IsPremiumCustomerSpecification o una OrderHasEnoughItemsSpecification.
3. **Especificaciones combinables:** A menudo, se incluyen métodos en la interfaz o clases auxiliares para combinar especificaciones existentes (e.g., And(Specification other), Or(Specification other)).

**¿Cuándo se utiliza?**

El patrón Specification es muy útil en escenarios como:

* **Filtrado de colecciones:** Seleccionar objetos de una lista que cumplen ciertos criterios. Por ejemplo, encontrar todos los productos en stock que están en oferta.
* **Validación de objetos:** Comprobar si un objeto cumple con un conjunto de reglas antes de realizar una operación. Por ejemplo, validar si los datos de un usuario son correctos antes de guardarlos.
* **Selección de reglas de negocio:** Aplicar diferentes reglas o flujos de trabajo basándose en ciertas condiciones.
* **Consultas de bases de datos:** Aunque la implementación puede variar, el concepto subyacente de "especificación" puede ser usado para construir consultas dinámicas.

**Ventajas principales:**

* **Mayor legibilidad y mantenimiento:** La lógica de las reglas está claramente definida y separada.
* **Mayor flexibilidad:** Fácilmente puedes añadir nuevas reglas o combinar las existentes sin modificar el código existente (principio de abierto/cerrado).
* **Mejor testabilidad:** Las especificaciones son unidades pequeñas y aisladas que se pueden testear individualmente.

**Ejemplo de su Implementación**

### La Interfaz ISpecification<T>

Primero, definimos una interfaz genérica que representará nuestra especificación. El tipo T será el tipo de objeto que queremos evaluar.

public interface ISpecification<T>

{

bool IsSatisfiedBy(T candidate);

}

### Implementaciones Básicas de ISpecification<T>

Ahora, creamos algunas implementaciones concretas para reglas específicas.

#### Ejemplo 1: Especificación para Clientes Premium

public class IsPremiumCustomerSpecification : ISpecification<Customer>

{

public bool IsSatisfiedBy(Customer customer)

{

// Suponiendo que Customer tiene una propiedad IsPremium

return customer.IsPremium;

}

}

#### Ejemplo 2: Especificación para Productos en Oferta

public class IsOnSaleProductSpecification : ISpecification<Product>

{

public bool IsSatisfiedBy(Product product)

{

// Suponiendo que Product tiene una propiedad DiscountPercentage

return product.DiscountPercentage > 0;

}

}

### Componiendo Especificaciones

Una de las mayores ventajas del patrón Specification es la capacidad de combinar especificaciones. Para ello, podemos extender la interfaz o crear clases base abstractas. Es común agregar métodos a la interfaz o, lo que es más flexible, crear una clase base CompositeSpecification.

Aquí tienes una implementación usando una clase base abstracta que ya sabe cómo combinar las especificaciones.

public abstract class CompositeSpecification<T> : ISpecification<T>

{

public abstract bool IsSatisfiedBy(T candidate);

public ISpecification<T> And(ISpecification<T> other)

{

return new AndSpecification<T>(this, other);

}

public ISpecification<T> Or(ISpecification<T> other)

{

return new OrSpecification<T>(this, other);

}

public ISpecification<T> Not()

{

return new NotSpecification<T>(this);

}

}

// Implementaciones concretas para los operadores lógicos

public class AndSpecification<T> : CompositeSpecification<T>

{

private readonly ISpecification<T> \_left;

private readonly ISpecification<T> \_right;

public AndSpecification(ISpecification<T> left, ISpecification<T> right)

{

\_left = left;

\_right = right;

}

public override bool IsSatisfiedBy(T candidate)

{

return \_left.IsSatisfiedBy(candidate) && \_right.IsSatisfiedBy(candidate);

}

}

public class OrSpecification<T> : CompositeSpecification<T>

{

private readonly ISpecification<T> \_left;

private readonly ISpecification<T> \_right;

public OrSpecification(ISpecification<T> left, ISpecification<T> right)

{

\_left = left;

\_right = right;

}

public override bool IsSatisfiedBy(T candidate)

{

return \_left.IsSatisfiedBy(candidate) || \_right.IsSatisfiedBy(candidate);

}

}

public class NotSpecification<T> : CompositeSpecification<T>

{

private readonly ISpecification<T> \_specification;

public NotSpecification(ISpecification<T> specification)

{

\_specification = specification;

}

public override bool IsSatisfiedBy(T candidate)

{

return !\_specification.IsSatisfiedBy(candidate);

}

}

**Nota:** Es importante que tus especificaciones concretas (como IsPremiumCustomerSpecification) hereden de CompositeSpecification<T> para poder usar los métodos And(), Or(), Not().

### Usando las Especificaciones

Aquí un ejemplo de cómo podrías usar estas especificaciones con algunas clases de dominio simples:

// Clases de Dominio (simplificadas para el ejemplo)

public class Customer

{

public int Id { get; set; }

public string Name { get; set; }

public bool IsPremium { get; set; }

public DateTime RegistrationDate { get; set; }

}

public class Product

{

public int Id { get; set; }

public string Name { get; set; }

public decimal Price { get; set; }

public decimal DiscountPercentage { get; set; }

public int Stock { get; set; }

}

public class Program

{

public static void Main(string[] args)

{

// Crear algunos datos de ejemplo

var customers = new List<Customer>

{

new Customer { Id = 1, Name = "Alice", IsPremium = true, RegistrationDate = new DateTime(2023, 1, 1) },

new Customer { Id = 2, Name = "Bob", IsPremium = false, RegistrationDate = new DateTime(2024, 6, 15) },

new Customer { Id = 3, Name = "Charlie", IsPremium = true, RegistrationDate = new DateTime(2022, 11, 20) }

};

var products = new List<Product>

{

new Product { Id = 101, Name = "Laptop", Price = 1200, DiscountPercentage = 0, Stock = 50 },

new Product { Id = 102, Name = "Mouse", Price = 25, DiscountPercentage = 0.10m, Stock = 100 },

new Product { Id = 103, Name = "Keyboard", Price = 75, DiscountPercentage = 0.20m, Stock = 0 },

new Product { Id = 104, Name = "Monitor", Price = 300, DiscountPercentage = 0, Stock = 30 }

};

// --- Uso de Especificaciones de Cliente ---

Console.WriteLine("--- Clientes ---");

var isPremium = new IsPremiumCustomerSpecification();

Console.WriteLine("Clientes Premium:");

foreach (var customer in customers.Where(c => isPremium.IsSatisfiedBy(c)))

{

Console.WriteLine($"- {customer.Name} (Premium: {customer.IsPremium})");

}

// Ejemplo de una especificación combinada para clientes

// Clientes premium registrados antes de 2024

var registeredBefore2024 = new RegisteredBeforeDateSpecification(new DateTime(2024, 1, 1));

var premiumAndOldCustomer = (registeredBefore2024 as CompositeSpecification<Customer>).And(isPremium);

Console.WriteLine("\nClientes Premium registrados antes de 2024:");

foreach (var customer in customers.Where(c => premiumAndOldCustomer.IsSatisfiedBy(c)))

{

Console.WriteLine($"- {customer.Name} (Premium: {customer.IsPremium}, Reg: {customer.RegistrationDate.ToShortDateString()})");

}

// --- Uso de Especificaciones de Producto ---

Console.WriteLine("\n--- Productos ---");

var isOnSale = new IsOnSaleProductSpecification();

var isInStock = new IsInStockProductSpecification(); // Necesitas crear esta clase

Console.WriteLine("Productos en Oferta:");

foreach (var product in products.Where(p => isOnSale.IsSatisfiedBy(p)))

{

Console.WriteLine($"- {product.Name} (Oferta: {product.DiscountPercentage:P0})");

}

var onSaleAndInStock = (isOnSale as CompositeSpecification<Product>).And(isInStock);

Console.WriteLine("\nProductos en Oferta Y en Stock:");

foreach (var product in products.Where(p => onSaleAndInStock.IsSatisfiedBy(p)))

{

Console.WriteLine($"- {product.Name} (Oferta: {product.DiscountPercentage:P0}, Stock: {product.Stock})");

}

}

}

// Clases de Especificación adicionales necesarias para el ejemplo completo

public class RegisteredBeforeDateSpecification : CompositeSpecification<Customer>

{

private readonly DateTime \_date;

public RegisteredBeforeDateSpecification(DateTime date)

{

\_date = date;

}

public override bool IsSatisfiedBy(Customer customer)

{

return customer.RegistrationDate < \_date;

}

}

public class IsInStockProductSpecification : CompositeSpecification<Product>

{

public override bool IsSatisfiedBy(Product product)

{

return product.Stock > 0;

}

}

### Consideraciones Adicionales

* **Expresiones Linq (Expression<Func<T, bool>>):** Para escenarios donde necesitas traducir tus especificaciones a consultas para bases de datos (como con Entity Framework Core), puedes extender el patrón para que IsSatisfiedBy devuelva una Expression<Func<T, bool>>. Esto permite que el ORM traduzca la lógica de tu especificación a SQL. Es un poco más complejo de implementar para los operadores lógicos, pero muy potente.
* **Separación de Responsabilidades:** El patrón Specification te ayuda a mantener la lógica de negocio de validación o filtrado fuera de tus repositorios o servicios, haciéndolos más limpios y enfocados en sus propias responsabilidades.
* **Testabilidad:** Cada especificación es una unidad de lógica de negocio que se puede probar de forma independiente, lo que mejora la calidad del código.

Espero que esta guía detallada te sea muy útil para implementar el patrón Specification en tus proyectos de C#! ¿Hay algún escenario específico en el que estés pensando usarlo?

**Explicación**

¡Excelente! Ahora que el código compila, vamos a desglosar el patrón **Specification** paso a paso para que cada pieza tenga sentido.

### Paso 1: La Interfaz ISpecification<T> (El "Contrato")

Imagina que tienes una regla que quieres aplicar a diferentes objetos. Esa regla debe poder decirte si un objeto cumple o no con el criterio. La interfaz ISpecification<T> define este contrato fundamental:

C#

namespace \_01\_Specification

{

public interface ISpecification<T>

{

// Este método es el corazón del patrón.

// Recibe un 'candidate' (el objeto a evaluar)

// y devuelve 'true' si el objeto satisface la especificación,

// o 'false' si no la satisface.

bool IsSatisfiedBy(T candidate);

}

}

* **ISpecification<T>:** La <T> al lado de ISpecification significa que es una **interfaz genérica**. Esto es crucial porque nos permite definir especificaciones para cualquier tipo de objeto (clientes, productos, pedidos, etc.) sin tener que crear una interfaz diferente para cada uno. T es un marcador de posición para el tipo real que usaremos.
* **IsSatisfiedBy(T candidate):** Este es el único método que todas las especificaciones deben implementar. Es donde reside la lógica de la regla.

### Paso 2: CompositeSpecification<T> (La "Base" para Combinar Reglas)

Aquí es donde el patrón se vuelve realmente poderoso. CompositeSpecification<T> no es una especificación de regla por sí misma, sino una **clase abstracta** que proporciona la funcionalidad para **combinar** otras especificaciones usando operadores lógicos (AND, OR, NOT).

C#

namespace \_01\_Specification

{

public abstract class CompositeSpecification<T> : ISpecification<T>

{

// Esta es una implementación obligatoria de la interfaz ISpecification<T>.

// Las clases que hereden de CompositeSpecification deberán definir

// su propia lógica de "satisfacción".

public abstract bool IsSatisfiedBy(T candidate);

// --- Métodos para combinar especificaciones ---

// Método AND: Crea una nueva especificación que será verdadera

// solo si esta especificación actual Y la 'other' son verdaderas.

public ISpecification<T> And(ISpecification<T> other)

{

return new AndSpecification<T>(this, other);

}

// Método OR: Crea una nueva especificación que será verdadera

// si esta especificación actual O la 'other' son verdaderas.

public ISpecification<T> Or(ISpecification<T> other)

{

return new OrSpecification<T>(this, other);

}

// Método NOT: Crea una nueva especificación que es la negación

// de esta especificación actual.

public ISpecification<T> Not()

{

return new NotSpecification<T>(this);

}

}

}

* **abstract class CompositeSpecification<T> : ISpecification<T>:** Es abstract porque no tiene sentido crear una instancia de CompositeSpecification directamente; siempre la usarás como base para crear tus reglas específicas. Hereda de ISpecification<T> porque, al final, una especificación compuesta también es una especificación.
* **public abstract bool IsSatisfiedBy(T candidate);**: Esto obliga a las clases hijas (tus reglas concretas) a implementar su lógica de IsSatisfiedBy.
* **And(), Or(), Not():** Estos son los métodos clave. En lugar de procesar la lógica de combinación directamente, delegan la tarea a clases auxiliares (AndSpecification<T>, OrSpecification<T>, NotSpecification<T>). **Esto es programación orientada a objetos en acción: cada clase tiene una única responsabilidad.**

### Paso 3: Las Especificaciones de Operadores Lógicos (Los "Combinadores")

Estas clases son los "engranajes" que hacen posible la combinación. Cada una encapsula la lógica de un operador lógico binario (AND, OR) o unario (NOT).

C#

namespace \_01\_Specification

{

// AND: ambas deben ser verdaderas

public class AndSpecification<T> : CompositeSpecification<T>

{

private readonly ISpecification<T> \_left; // La primera especificación

private readonly ISpecification<T> \_right; // La segunda especificación

public AndSpecification(ISpecification<T> left, ISpecification<T> right)

{

\_left = left;

\_right = right;

}

public override bool IsSatisfiedBy(T candidate)

{

// Retorna verdadero solo si AMBAS especificaciones son satisfechas.

return \_left.IsSatisfiedBy(candidate) && \_right.IsSatisfiedBy(candidate);

}

}

// OR: al menos una debe ser verdadera

public class OrSpecification<T> : CompositeSpecification<T>

{

private readonly ISpecification<T> \_left;

private readonly ISpecification<T> \_right;

public OrSpecification(ISpecification<T> left, ISpecification<T> right)

{

\_left = left;

\_right = right;

}

public override bool IsSatisfiedBy(T candidate)

{

// Retorna verdadero si ALGUNA de las especificaciones es satisfecha.

return \_left.IsSatisfiedBy(candidate) || \_right.IsSatisfiedBy(candidate);

}

}

// NOT: la opuesta debe ser verdadera

public class NotSpecification<T> : CompositeSpecification<T>

{

private readonly ISpecification<T> \_specification; // La especificación a negar

public NotSpecification(ISpecification<T> specification)

{

\_specification = specification;

}

public override bool IsSatisfiedBy(T candidate)

{

// Retorna verdadero si la especificación original NO es satisfecha.

return !\_specification.IsSatisfiedBy(candidate);

}

}

}

* **Heredan de CompositeSpecification<T>:** Esto es clave porque estas clases también son "especificaciones" y, por lo tanto, pueden ser usadas en combinaciones más complejas (e.g., (A AND B) OR C).
* **Encapsulan la lógica del operador:** Cada una sabe exactamente cómo combinar el resultado de sus especificaciones internas.

### Paso 4: Tus Especificaciones Concretas (Tus "Reglas de Negocio")

Aquí es donde realmente defines las reglas de tu dominio. Estas son las clases que usas directamente en tu aplicación para expresar un criterio específico.

C#

namespace \_01\_Specification

{

// Una regla para determinar si un Cliente es Premium

public class IsPremiumCustomerSpecification : CompositeSpecification<Customer> // ¡Importante: hereda de CompositeSpecification!

{

public override bool IsSatisfiedBy(Customer customer)

{

return customer.IsPremium; // La lógica de la regla

}

}

// Una regla para determinar si un Producto está en oferta

public class IsOnSaleProductSpecification : CompositeSpecification<Product> // ¡Importante: hereda de CompositeSpecification!

{

public override bool IsSatisfiedBy(Product product)

{

return product.DiscountPercentage > 0; // La lógica de la regla

}

}

// Una regla para determinar si un Cliente se registró antes de una fecha

public class RegisteredBeforeDateSpecification : CompositeSpecification<Customer>

{

private readonly DateTime \_date; // Una propiedad para que la regla sea configurable

public RegisteredBeforeDateSpecification(DateTime date)

{

\_date = date;

}

public override bool IsSatisfiedBy(Customer customer)

{

return customer.RegistrationDate < \_date;

}

}

// Una regla para determinar si un Producto tiene stock

public class IsInStockProductSpecification : CompositeSpecification<Product>

{

public override bool IsSatisfiedBy(Product product)

{

return product.Stock > 0;

}

}

}

* **Heredan de CompositeSpecification<T>:** Esto les da acceso a los métodos And(), Or(), Not() directamente, lo que nos permite combinarlas fácilmente.
* **Implementan IsSatisfiedBy():** Aquí es donde pones la lógica real de tu negocio. Son muy específicas y hacen una única cosa bien.

### Paso 5: Usando las Especificaciones (La "Aplicación Práctica")

Finalmente, así es como juntas todo para aplicar tus reglas en tu código.

C#

using \_01\_Specification; // Asegúrate de tener este 'using'

using System;

using System.Collections.Generic;

using System.Linq; // Necesario para el método .Where()

public class Program

{

public static void Main(string[] args)

{

// ... (definición de Customer y Product, y datos de ejemplo como antes) ...

// 1. Crear instancias de tus reglas:

var isPremium = new IsPremiumCustomerSpecification();

var registeredBefore2024 = new RegisteredBeforeDateSpecification(new DateTime(2024, 1, 1));

// 2. Usar una regla simple para filtrar:

Console.WriteLine("Clientes Premium:");

foreach (var customer in customers.Where(c => isPremium.IsSatisfiedBy(c)))

{

Console.WriteLine($"- {customer.Name}");

}

// 3. Combinar reglas para crear una más compleja:

// isPremium y registeredBefore2024 ya son CompositeSpecification<Customer>,

// por lo que puedes llamar a .And() directamente.

var premiumAndOldCustomer = registeredBefore2024.And(isPremium);

Console.WriteLine("\nClientes Premium registrados antes de 2024:");

foreach (var customer in customers.Where(c => premiumAndOldCustomer.IsSatisfiedBy(c)))

{

Console.WriteLine($"- {customer.Name}");

}

// Lo mismo para los productos:

var isOnSale = new IsOnSaleProductSpecification();

var isInStock = new IsInStockProductSpecification();

var onSaleAndInStock = isOnSale.And(isInStock); // Combinando reglas de producto

Console.WriteLine("\nProductos en Oferta Y en Stock:");

foreach (var product in products.Where(p => onSaleAndInStock.IsSatisfiedBy(p)))

{

Console.WriteLine($"- {product.Name}");

}

}

}

* **Instanciación:** Creas objetos de tus clases de especificación.
* **Aplicación:** Llamas a IsSatisfiedBy() para verificar si un objeto cumple con la regla.
* **Composición:** Usas los métodos And(), Or(), Not() en tus instancias de especificación para construir reglas compuestas de manera fluida y legible.

### ¿Por qué este patrón es útil?

* **Legibilidad:** Las reglas de negocio se leen casi como lenguaje natural (e.g., "clientes que son premium Y que se registraron antes de 2024").
* **Reutilización:** Puedes reutilizar IsPremiumCustomerSpecification en cualquier parte de tu código donde necesites ese criterio, sin duplicar la lógica.
* **Mantenibilidad:** Si la definición de "cliente premium" cambia, solo tienes que modificar una clase (IsPremiumCustomerSpecification), no muchos lugares dispersos en el código.
* **Testabilidad:** Cada especificación es una pequeña unidad de lógica que puedes probar de forma aislada, lo que simplifica tus pruebas unitarias.
* **Separación de Conceptos:** La lógica de "qué" se valida o filtra (la especificación) está separada del "cómo" se usa esa validación o filtrado (el código que llama a IsSatisfiedBy).

Espero que esta explicación paso a paso te ayude a comprender el patrón Specification en profundidad. ¿Qué te parece? ¿Hay alguna parte que te gustaría que ampliara o algún escenario específico en el que te gustaría aplicarlo?

**TRAZA**

Ejemplo concreto del flujo para validar si un producto está en stock (usando IsInStockProductSpecification), pero también mostraré cómo se combinan las especificaciones y cómo se usan en el Program.

## 📝 TRAZABILIDAD COMPLETA: FLUJO DE EJECUCIÓN

### 1️⃣ Punto de Entrada: Main en la clase Program

public static void Main(string[] args)

Aquí comienza todo. Se ejecuta el método Main.

### 2️⃣ Creación de Datos de Prueba:

var products = new List<Product>

{

new Product { Id = 101, Name = "Laptop", Price = 1200, DiscountPercentage = 0, Stock = 50 },

new Product { Id = 102, Name = "Mouse", Price = 25, DiscountPercentage = 0.10m, Stock = 100 },

new Product { Id = 103, Name = "Keyboard", Price = 75, DiscountPercentage = 0.20m, Stock = 0 },

new Product { Id = 104, Name = "Monitor", Price = 300, DiscountPercentage = 0, Stock = 30 }

};

Se crean algunos productos en memoria.

### 3️⃣ Creación de Especificaciones:

var isInStock = new IsInStockProductSpecification();

Aquí se **crea una instancia** de la especificación IsInStockProductSpecification.

📌 Esta clase hereda de CompositeSpecification<Product> e implementa el método:

public override bool IsSatisfiedBy(Product product)

{

return product.Stock > 0;

}

👉 Esta es la regla: **Un producto está en stock si su Stock es mayor a 0.**

### 4️⃣ Aplicación de la Especificación:

foreach (var product in products.Where(p => isInStock.IsSatisfiedBy(p)))

{

Console.WriteLine($"- {product.Name} (Stock: {product.Stock})");

}

🔍 Aquí ocurre lo siguiente paso a paso:

| **PASO** | **ACCIÓN** |
| --- | --- |
| 4.1 | Por cada product en la lista products... |
| 4.2 | Se llama isInStock.IsSatisfiedBy(product). |
| 4.3 | Se entra a la clase IsInStockProductSpecification y se evalúa la expresión product.Stock > 0. |
| 4.4 | Si devuelve true, el producto se muestra en pantalla. |

✅ Por ejemplo:

* Producto **Laptop**: Stock = 50 → true
* Producto **Keyboard**: Stock = 0 → false

### 5️⃣ ¿Qué pasa por dentro? (Llamadas entre clases)

Program.Main()

|

|---> Crea instancia de IsInStockProductSpecification

|

|---> Llama IsInStockProductSpecification.IsSatisfiedBy(Product)

|

|---> Evalúa: ¿product.Stock > 0?

|

|---> Devuelve true o false

✅ Es un flujo **directo**:

1. El Program crea la especificación.
2. La especificación recibe un producto.
3. Evalúa si cumple o no la condición.

### 6️⃣ Especificaciones Compuestas

En el caso de un compuesto como este:

var onSaleAndInStock = (isOnSale as CompositeSpecification<Product>).And(isInStock);

👉 Aquí el flujo cambia un poco:

| **PASO** | **ACCIÓN** |
| --- | --- |
| 6.1 | Se llama .And() sobre una especificación (CompositeSpecification crea una nueva AndSpecification<Product>) |
| 6.2 | Se genera un nuevo objeto que encapsula las dos reglas: isOnSale **Y** isInStock. |
| 6.3 | Cuando se llama .IsSatisfiedBy(product), se evalúan las dos especificaciones originales y se combinan con && (AND lógico). |

📌 Internamente usa esta clase:

public override bool IsSatisfiedBy(T candidate)

{

return \_left.IsSatisfiedBy(candidate) && \_right.IsSatisfiedBy(candidate);

}

### 7️⃣ Resultado Visual en Consola:

Ejemplo de salida esperada:

Productos en Oferta Y en Stock:

- Mouse (Oferta: 10%, Stock: 100)

## 🗺 RESUMEN DE TRAZABILIDAD (Paso a paso general):

| **PASO** | **DESCRIPCIÓN** | **CLASE / MÉTODO** |
| --- | --- | --- |
| 1 | Inicia ejecución | Program.Main |
| 2 | Se crean listas de productos y clientes | Program |
| 3 | Se crean especificaciones | IsInStockProductSpecification, IsOnSaleProductSpecification |
| 4 | Se aplican especificaciones simples o combinadas | .IsSatisfiedBy() |
| 5 | Evaluación lógica en especificación concreta | Dentro de clases como IsInStockProductSpecification |
| 6 | Se imprime el resultado según especificación | Console.WriteLine |