Ejercicio 7.1:
Implementar un framework de validación

Empecemos por implementar un framework de validación de dominio que contenga las abstracciones y servicios para que las aplicaciones que lo consuman puedan crear clases de validación que satisfagan reglas de negocio.

La validación puede ser realizada a través de una serie de instrucciones ***If*** o puede ser implementada utilizando alguna herramienta propia o de terceros. En esta lección implementaremos el patrón de diseño ***Specification*** para crear el framework de validación.

El patrón ***Specification*** es un patrón de diseño de software que nos permite encapsular reglas de negocio en una sola unidad: *la Especificación* (*Specification*). Una vez encapsuladas, las reglas de negocio pueden ser reutilizadas en diferentes partes del código y además pueden ser modificadas fácilmente. El patrón *Specification* se utiliza con frecuencia en el contexto del diseño orientado al dominio.

En el dominio, una *Especificación* (*Specification*) recibe una entidad como parámetro y nos dice si satisface o no las reglas de negocio que encapsula. Las reglas de negocio no son más que una serie de condiciones que ciertas entidades deben cumplir.

El patrón ***Specification*** es explicado detalladamente en el documento "***Specifications***" elaborado por *Eric Evans* y *Martin Fowler*.

Specifications
https://martinfowler.com/apsupp/spec.pdf

Es probable que, para sistemas pequeños, la estrategia de validación recomendada sea *Anotaciones de Datos* debido a su facilidad de implementación. Sin embargo, para sistemas más grandes y complejos, la recomendación es separar la responsabilidad de la validación utilizando objetos de validación con código.

Tarea 1: Crear la solución

Para implementar el framework de validación de dominio, crearemos una nueva solución. Posteriormente, el framework podrá incorporarse como parte de la capa de *Reglas de Negocio Empresariales* o como parte de la capa de *Reglas de Negocio de Aplicación*.

1. Utiliza la plantilla *Class Library* para crear un nuevo proyecto llamado ***DomainValidation*** en una solución con el mismo nombre.

2. Elimina el archivo ***Class1.cs***.

Tarea 2: Agregar los objetos de valor

1. Agrega un nuevo directorio llamado ***ValueObjects*** en la raíz del proyecto.
2. Agrega un nuevo archivo de clase llamado ***SpecificationError.cs*** en el directorio *ValueObjects* y remplaza el contenido de la clase por el siguiente.

```
public class SpecificationError(string propertyName, string errorMessage)
{
    public string PropertyName => propertyName;
    public string ErrorMessage => errorMessage;
}
```

La clase *SpecificationError* nos permitirá representar un error de validación de una propiedad de una entidad.

3. Agregar un nuevo archivo de clase llamado ***ValidationResult.cs*** en el directorio *ValueObjects* y remplaza el contenido de la clase por el siguiente.

```
public class ValidationResult(IEnumerable<SpecificationError> errors)
{
    public bool IsValid => errors?.Any() != true;
    public IEnumerable<SpecificationError> Errors => errors;
}
```

La clase *ValidationResult* representa el resultado de una validación de especificaciones. Si la colección *Errors* contiene algún elemento, significa que la entidad validada no cumple con una o más especificaciones.

Tarea 3: Agregar las Interfaces

1. Agrega un nuevo directorio llamado ***Interfaces*** en la raíz del proyecto.
2. Agrega un nuevo archivo de tipo *Interface* llamado ***ISpecification.cs*** en el directorio *Interfaces* y remplaza el contenido de la interface por el siguiente.

```
public interface ISpecification<T>
{
    IEnumerable<SpecificationError> Errors { get; }
    bool IsSatisfiedBy(T entity);
}
```

La implementación de esta interface permitirá definir una especificación que una entidad debe cumplir. El método `IsSatisfiedBy` contendrá la lógica de validación, mientras que la colección `Errors` almacenará los errores encontrados cuando la entidad no cumpla con las reglas de validación. Utilizaremos esta interface principalmente para definir especificaciones sobre las propiedades de una entidad. Por ejemplo, podremos indicar que una propiedad es obligatoria, establecer la longitud permitida para una cadena de texto o validar que el valor de una propiedad tenga un formato de correo electrónico válido, entre otras reglas de validación.

3. Importa los espacios de nombres requeridos en el archivo *GlobalUsings.cs*.
4. Agrega un nuevo archivo de tipo Interface llamado ***IDomainSpecification.cs*** en el directorio *Interfaces* y remplaza el contenido de la interface por el siguiente.

```
public interface IDomainSpecification<T>
{
    bool EvaluateOnlyIfNoPreviousErrors { get; }
    bool StopOnFirstEntitySpecificationError { get; }
    IEnumerable<SpecificationError> Errors { get; }
    Task<bool> ValidateAsync(T entity);
}
```

Utilizaremos esta interface para definir un conjunto de especificaciones que una entidad debe cumplir.

- La propiedad ***EvaluateOnlyIfNoPreviousErrors***, cuando su valor es ***true***, indica al motor de validación que esta especificación solo debe evaluarse si no se han encontrado errores en otras especificaciones de la entidad. Esto resulta útil en casos donde la validación requiere consultar una base de datos. Por ejemplo, si una especificación debe verificar la existencia de la clave de un cliente en la base de datos, podemos evitar consultas innecesarias si previamente se ha detectado que los datos ingresados por el usuario son inválidos.
- La propiedad ***StopOnFirstEntitySpecificationError***, cuando su valor es ***true***, indica que la validación de esta especificación debe detenerse tan pronto como se encuentre el primer error dentro del conjunto de especificaciones de la entidad.
- La colección ***Errors*** almacenará los errores de validación detectados durante el proceso de validación, el cual se inicia al invocar el método ***ValidateAsync***.

5. Agrega un nuevo archivo de tipo *Interface* llamado ***IDomainSpecificationsValidator.cs*** en el directorio *Interfaces* y remplaza el contenido de la interface por el siguiente.

```
public interface IDomainSpecificationsValidator<T>
{
    Task<ValidationResult> ValidateAsync(T entity);
```

```
}
```

Esta interface facilita el inicio del proceso de validación de una o más especificaciones de dominio para una entidad, asegurando que todas las reglas definidas se apliquen correctamente.

6. Agrega un nuevo archivo de tipo Interface llamado ***IPropertySpecificationsTree.cs*** en el directorio *Interfaces* y remplaza el contenido de la interface por el siguiente.

```
public interface IPropertySpecificationsTree<T>
{
    string PropertyName { get; }
    List<ISpecification<T>> Specifications { get; }
    bool StopOnFirstPropertySpecificationError { get; }
    object GetPropertyValue(T entity);
}
```

La interface *IPropertySpecificationsTree* permite definir las reglas de validación para las propiedades de una entidad, organizando las especificaciones de una sola propiedad. Una especificación de dominio podrá contener un grupo de *IPropertySpecificationsTree*.

- La propiedad ***PropertyName*** contiene el nombre de la propiedad que está asociada con las especificaciones.
- La colección ***Specifications*** almacena las especificaciones que deben cumplirse para dicha propiedad.
- Si ***StopOnFirstPropertySpecificationError*** tiene un valor ***true***, el proceso de validación se detendrá al encontrar el primer error en una especificación de la propiedad, sin evaluar el resto de las especificaciones.
- El método ***GetPropertyValue*** obtiene el valor de la propiedad durante el proceso de validación. El tipo devuelto es ***object*** para manejar propiedades de diferentes tipos en el modelo ***T***. Por ejemplo, una propiedad puede devolver un ***int*** mientras que otra podría devolver un *string*, un ***double*** o algún otro tipo.

Tarea 4: Agregar la implementación de ISpecification

1. Agrega un nuevo directorio llamado ***Core*** en la raíz del proyecto.
2. Agrega un nuevo archivo de clase llamado ***Specification.cs*** en el directorio *Core* y remplaza el contenido de la clase por el siguiente.

```
public class Specification<T>(
    Func<T, IEnumerable<SpecificationError>> validationRule) : ISpecification<T>
```

```
{
    public IEnumerable<SpecificationError> Errors { get; private set; }
    public bool IsSatisfiedBy(T entity)
    {
        Errors = validationRule(entity);

        return Errors?.Any() != true;
    }
}
```

La clase `Specification<T>`, que implementa la interfaz `ISpecification<T>`, recibe como parámetro un delegado `Func<T, IEnumerable<SpecificationError>>`, encargado de definir la lógica de validación de la especificación. Este delegado toma una instancia de la entidad a validar y devuelve una colección de `SpecificationError` si la entidad no cumple con los criterios establecidos.

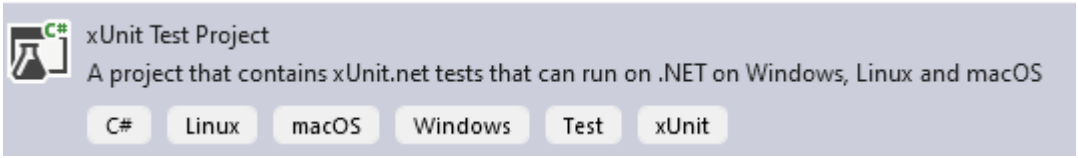
El método `IsSatisfiedBy` ejecuta el delegado de validación, almacena el resultado en la propiedad `Errors` y devuelve `true` si la colección `Errors` es `null` o está vacía, indicando que la entidad satisface la especificación.

3. Importa los espacios de nombres requeridos en el archivo *GlobalUsings.cs*.

Tarea 5: Agregar un proyecto de pruebas unitarias

Ahora incorporaremos un proyecto de pruebas unitarias para verificar el correcto funcionamiento de las implementaciones que iremos desarrollando. Además de validar la funcionalidad, el código de pruebas servirá para ilustrar los diferentes escenarios de uso del framework de validación.

1. Utiliza la plantilla ***xUnit Test Project*** para agregar a la solución un nuevo proyecto llamado ***DomainValidation.Tests***.



2. Elimina el archivo ***UnitTest1.cs***.
3. Agrega una referencia del proyecto *DomainValidation*.
4. Agrega un nuevo directorio llamado ***Models*** en la raíz del proyecto.
5. Agrega un nuevo archivo de clase llamado ***CreateOrder.cs*** en el directorio *Models* y remplaza el contenido de la clase por el siguiente.

```
public class CreateOrder
{
```

```
    public const string CustomerIdIsRequired =
        "Customer ID is required";
    public string CustomerId { get; set; }
    public string ShipAddress { get; set; }
    public string ShipCity { get; set; }
    public string ShipCountry { get; set; }
    public string ShipPostalCode { get; set; }
}
```

6. Agrega un nuevo directorio llamado ***SpecificationTest*** en la raíz del proyecto.
7. Agrega un nuevo archivo de clase llamado ***SpecificationTest.cs*** en el directorio *SpecificationTest* y remplaza el contenido de la clase por el siguiente.

```
public class SpecificationTest
{

}
```

8. Agrega el siguiente código a la clase para definir una especificación.

```
ISpecification<CreateOrder> Specification =
    new Specification<CreateOrder>(entity =>
{
    List<SpecificationError> Errors = null;

    if (string.IsNullOrWhiteSpace(entity.CustomerId))
    {
        Errors = new List<SpecificationError>
        {
            new SpecificationError("CustomerId",
            CreateOrder.CustomerIdIsRequired)
        };
    }

    return Errors;
});
```

El código anterior crea una instancia de `Specification<CreateOrder>` con una regla de validación que verifica si la propiedad `CustomerId` de la entidad `CreateOrder` tiene un valor asignado. Si el campo está vacío o compuesto únicamente por espacios en blanco, se genera una lista de errores de validación que contiene un `SpecificationError` indicando que `CustomerId` es obligatorio.

9. Agrega el siguiente código a la clase para definir el caso de prueba que verifica que el método `IsSatisfiedBy` devuelva `false` cuando la validación falla. En este escenario, la propiedad `CustomerId` de la entidad `CreateOrder` es `null`, lo que provoca que la regla de validación genere un error.

```
[Fact]
public void IsSatisfiedBy_ShouldReturnFalse_WhenValidationFails()
{
    // Arrange
    var Entity = new CreateOrder { CustomerId = null };

    // Act
    var Result = Specification.IsSatisfiedBy(Entity);

    // Assert
    Assert.False(Result);
    Assert.Single(Specification.Errors);
    Assert.Equal(CreateOrder.CustomerIdIsRequired,
        Specification.Errors.First().ErrorMessage);
}
```

Esta prueba asegura que, cuando la validación no se cumple, la especificación registre un único error en la colección *Errors* y que el mensaje de error sea el esperado.

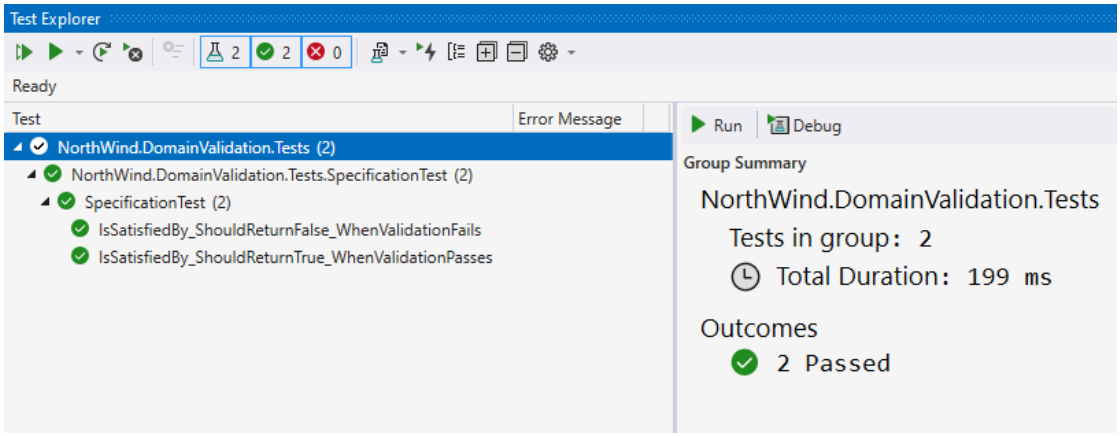
10. Agrega el siguiente código a la clase para definir el caso de prueba que verifica que el método `IsSatisfiedBy` devuelva `true` cuando la entidad cumple con la regla de validación. En este escenario, la propiedad `CustomerId` de `CreateOrder` tiene un valor válido, por lo que no se generan errores de validación.

```
[Fact]
public void IsSatisfiedBy_ShouldReturnTrue_WhenValidationPasses()
{
    // Arrange
    var Entity = new CreateOrder { CustomerId = "ALFKI" };

    // Act
    var Result = Specification.IsSatisfiedBy(Entity);

    // Assert
    Assert.True(Result);
    Assert.True(Specification.Errors == null ||
        !Specification.Errors.Any(), "Errors should be null or empty");
}
```

Esta prueba garantiza que cuando la entidad satisface la especificación, el resultado sea `true` y la colección `Errors` sea `null` o esté vacía.

11. Selecciona la opción ***Test > Run All Tests*** de la barra de menús de *Visual Studio* para ejecutar las pruebas unitarias.

    Deberás ver un resultado similar al siguiente indicando que las pruebas se ejecutaron con éxito.



Tarea 6: Agregar un método de extensión

Regresemos ahora con las implementaciones en el proyecto *DomainValidation*.

1. Agrega un nuevo directorio llamado **Extensions** en la raíz del proyecto.
2. Agrega un nuevo archivo de clase llamado ***ExpressionExtensions.cs*** en el directorio *Extensions* y remplaza el contenido de la clase por el siguiente.

```
internal static class ExpressionExtensions
{
    public static string GetPropertyName<T, TProperty>(
        this Expression<Func<T, TProperty>> propertyExpression)
    {
        string PropertyName = null;

        var Body = propertyExpression.Body;

        if (Body is UnaryExpression UnaryExpression)
        {
            Body = UnaryExpression.Operand;
        }

        if (Body is MemberExpression MemberExpression)
        {
            PropertyName = MemberExpression.Member.Name;
        }

        return PropertyName;
    }
}
```

La clase *ExpressionExtensions* contiene una extensión para expresiones que permite obtener el nombre de una propiedad de una entidad a partir de una expresión lambda. El método *GetPropertyName* acepta una expresión `Func<T, TProperty>`, que representa el acceso a una

propiedad, y extrae su nombre. Si la expresión es una `UnaryExpression` (como una conversión de tipo), se extrae su operando. Luego, si la expresión resultante es un `MemberExpression`, se obtiene el nombre de la propiedad a través de su propiedad `Member.Name`.

Este método es útil para obtener dinámicamente el nombre de una propiedad a partir de una expresión de acceso a propiedad.

3. Importa el espacio de nombres requerido en el archivo *GlobalUsings.cs*.

Tarea 7: Agregar la implementación de IPropertySpecificationsTree

1. Agrega un nuevo archivo de clase llamado ***PropertySpecificationsTree.cs*** en el directorio *Core* y remplaza el contenido de la clase por el siguiente.

```
public class PropertySpecificationsTree<T, TProperty>(
    Expression<Func<T, TProperty>> propertyExpression,
    bool stopOnFirstPropertySpecificationError = false)
    : IPropertySpecificationsTree<T>
{
    readonly Func<T, TProperty> PropertyExpressionDelegate =
        propertyExpression.Compile();

    public string PropertyName { get; } =
        propertyExpression.GetPropertyName();

    public List<ISpecification<T>> Specifications { get; } = [];
    public bool StopOnFirstPropertySpecificationError =>
        stopOnFirstPropertySpecificationError;
    public object GetPropertyValue(T entity) =>
        PropertyExpressionDelegate(entity);
}
```

La clase `PropertySpecificationsTree<T, TProperty>` implementa la interface `IPropertySpecificationsTree<T>`. Su constructor recibe `propertyExpression`, una expresión de acceso a una propiedad, y `stopOnFirstPropertySpecificationError`, un parámetro opcional, que indica si la validación debe detenerse en el primer error encontrado.

La propiedad `PropertyExpressionDelegate` almacena el delegado resultante de compilar la expresión de acceso a la propiedad, lo que permite obtener su valor en tiempo de ejecución a través del método `GetPropertyValue`.

La propiedad `PropertyName` expone el nombre de la propiedad evaluada, obtenido mediante la ejecución del método de extensión `GetPropertyName` sobre la expresión recibida.

La colección `Specifications` almacena las especificaciones asociadas a la propiedad, mientras que `StopOnFirstPropertySpecificationError` indica si la validación debe finalizar en el primer error encontrado.

2. Importa los espacios de nombres requeridos en el archivo *GlobalUsings.cs*.

Tarea 8: Agregar un método auxiliar

1. Agrega un nuevo directorio llamado ***Helpers*** en la raíz del proyecto.
2. Agrega un nuevo archivo de clase llamado ***PropertySpecificationTreeHelper.cs*** en el directorio *Helpers* y remplaza el contenido de la clase por el siguiente.

```
internal static class PropertySpecificationTreeHelper
{
    public static List<SpecificationError> Validate<T, TProperty>(T entity,
        PropertySpecificationsTree<T, TProperty> tree,
        Action<List<SpecificationError>, TProperty> validator)
    {
        List<SpecificationError> Errors = [];

        var Value = (TProperty)tree.GetPropertyValue(entity);

        validator(Errors, Value);

        return Errors;
    }
}
```

La clase `PropertySpecificationTreeHelper` proporciona un método auxiliar `Validate` para la validación de especificaciones asociadas a una propiedad.

El método `Validate` encapsula la lógica necesaria para evaluar una especificación de una entidad dada. Primero, obtiene el valor de la propiedad a validar utilizando el árbol de especificaciones `tree`. Luego, delega la validación al `validator`, el cual aplica las reglas de negocio sobre el valor de la propiedad y almacena los errores detectados en la lista `Errors`. Finalmente, devuelve la lista de errores encontrados.

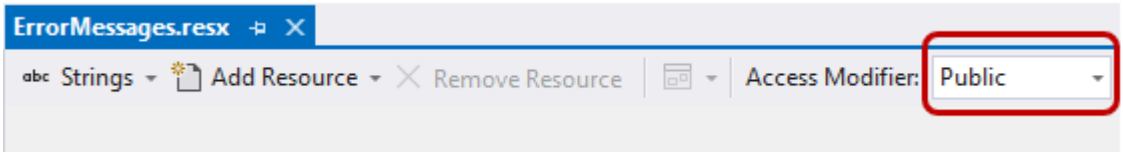
Este enfoque centraliza la validación de propiedades, facilitando la reutilización del código y asegurando una estructura consistente en la evaluación de reglas de negocio.

3. Importa el espacio de nombres requerido en el archivo *GlobalUsings.cs*.

Tarea 9: Agregar un archivo de recursos

Agreguemos ahora un archivo de recursos donde almacenaremos los mensajes de error predeterminados de las validaciones que implementaremos.

1. Agrega un nuevo directorio llamado ***Resources*** en la raíz del proyecto.
2. Agrega un nuevo archivo de tipo *Resources File* llamado ***ErrorMessages.resx*** en el directorio *Resources*.
3. Establece el modificador de acceso del archivo como ***Public***.



4. Agrega los siguientes valores al archivo de recursos.

Nombre	Valor
EmailAddress	Invalid email format.
Equal	The values are not equal.
GreaterThan	The value must be greater than {0}.
GreaterThanOrEqualTo	The value must be greater than or equal to {0}.
HasFixedLength	{0} characters are required.
HasMaxLength	The maximum length is {0} characters.
HasMinLength	The minimum length is {0} characters.
IsRequired	This information is required.
Matches	Must satisfy the expression {0}
NotEmpty	Required information.

Tarea 10: Agregar extensiones de validación de PropertySpecificationsTree

Ahora implementaremos las validaciones más comunes como métodos de extensión de la clase *PropertySpecificationsTree*. Esto permitirá aplicar reglas de validación de manera flexible y reutilizable.

1. Agrega un nuevo directorio llamado ***PropertySpecificationTreeExtensions*** en el directorio *Extensions*.

Tarea 10.1: Implementar la validación IsRequired

1. Agrega un nuevo archivo de clase llamado ***IsRequiredExtension.cs*** en el directorio *PropertySpecificationTreeExtensions* y remplaza el código de la clase por el siguiente.

```
public static class IsRequiredExtension
{
    public static PropertySpecificationsTree<T, TProperty>
        IsRequired<T, TProperty>(
        this PropertySpecificationsTree<T, TProperty> tree,
        string errorMessage = default)
    {
        tree.Specifications.Add(new Specification<T>(entity =>
        PropertySpecificationTreeHelper.Validate(entity, tree, (errors, value) =>
        {
            if (value == null ||
                value is string str && string.IsNullOrWhiteSpace(str))
            {
                errors.Add(new SpecificationError(tree.PropertyName,
                    errorMessage ?? ErrorMessages.IsRequired));
            }
        })));

        return tree;
    }
}
```

La clase estática `IsRequiredExtension` define un método de extensión `IsRequired` para la clase `PropertySpecificationsTree<T, TProperty>`. Este método permite agregar una validación que exige que una propiedad no sea `null` ni una cadena vacía o compuesta únicamente de espacios en blanco.

Este enfoque facilita la incorporación de nuevas reglas de validación sin modificar el código existente.

2. Importa los espacios de nombres requeridos en el archivo *GlobalUsings.cs*.

Agreguemos ahora las pruebas unitarias para este método de extensión.

3. Agrega un nuevo directorio llamado ***PropertySpecificationTreeExtensionsTests*** en la raíz del proyecto de pruebas.
4. En el directorio *PropertySpecificationTreeExtensionsTests*, agrega un nuevo archivo de clase llamado ***IsRequiredTests.cs*** y remplaza el contenido de la clase por el siguiente.

```
public class IsRequiredTests
{
}
```

5. Agrega el siguiente código a la clase *IsRequiredTests* para definir una expresión de acceso a la propiedad *CustomerId* de una instancia de *CreateOrder*.

```
Expression<Func<CreateOrder, string>> PropertyExpression =
    x => x.CustomerId;
```

6. Agrega el siguiente código a la clase para definir una prueba unitaria.

```
[Theory]
[InlineData(null, false)]
[InlineData("   ", false)]
[InlineData("ALFKI", true)]
public void IsRequired_ShouldReturnExpectedResult_WhenValueIsChecked(
    string customerId, bool expectedResult)
{
    // Arrange
    var Tree = new PropertySpecificationsTree<CreateOrder, string>(
        PropertyExpression);

    Tree.IsRequired();

    var Entity = new CreateOrder { CustomerId = customerId };

    // Act
    bool Result = Tree.Specifications[0].IsSatisfiedBy(Entity);

    // Assert
    Assert.Equal(expectedResult, Result);

    if (!expectedResult)
    {
        Assert.Single(Tree.Specifications[0].Errors);
    }
    else
    {
        Assert.True(Tree.Specifications[0].Errors == null ||
            !Tree.Specifications[0].Errors.Any());
    }
}
```

Este código define una prueba unitaria parametrizada mediante ***Theory*** e ***InlineData*** en ***xUnit***, validando el comportamiento del método ***IsRequired*** en *PropertySpecificationsTree*.

La prueba se ejecuta con diferentes valores de *customerId* y su resultado esperado *expectedResult*:

- *null* → *false* (debe fallar la validación).
- " " (solo espacios en blanco) → *false* (debe fallar la validación).
- "ALFKI" → *true* (debe pasar la validación).

En la fase *Arrange*:

- Se crea una instancia de *PropertySpecificationsTree*, inicializada con la expresión *PropertyExpression* que apunta a *CustomerId*.
- Se aplica la validación *IsRequired()*, agregando una especificación que verifica que *CustomerId* no sea null ni una cadena vacía o con solo espacios.
- Se instancia un objeto *CreateOrder* con el valor de *customerId* recibido como parámetro.

En la fase *Act*, se evalúa la especificación (*IsSatisfiedBy*) para la entidad *Entity*, almacenando el resultado en *Result*.

En la fase *Assert*:

- Se compara *Result* con *expectedResult* para verificar si la validación se comporta como se espera.
- Si *expectedResult* es *false* (la validación falla), se verifica que haya exactamente un error en la lista de errores.
- Si *expectedResult* es *true* (la validación pasa), se comprueba que la lista de errores sea *null* o esté vacía.

7. Agrega el siguiente código a la clase para definir una nueva prueba unitaria para validar que el método *IsRequired* de *PropertySpecificationsTree* utiliza el mensaje de error proporcionado por el usuario cuando la validación falla.

```
[Fact]
public void IsRequired_ShouldUseProvidedErrorMessage_WhenValidationFails()
{
    // Arrange
    var Tree = new PropertySpecificationsTree<CreateOrder, string>(
        PropertyExpression);
    string ExpectedErrorMessage = CreateOrder.CustomerIdIsRequired;

    Tree.IsRequired(ExpectedErrorMessage);

    var Entity = new CreateOrder { CustomerId = null };

    // Act
    bool Result = Tree.Specifications[0].IsSatisfiedBy(Entity);

    // Assert
    Assert.False(Result);
    Assert.Equal(ExpectedErrorMessage,
        Tree.Specifications[0].Errors.First().ErrorMessage);
}
```

8. Agrega el siguiente código a la clase para definir una nueva prueba unitaria para validar que el método *IsRequired* de *PropertySpecificationsTree* utiliza el mensaje de error predeterminado cuando no se proporciona un mensaje de error personalizado y la validación falla.

```
[Fact]
public void IsRequired_ShouldUseDefaultErrorMessage_WhenValidationFails()
{
    // Arrange
    var Tree = new PropertySpecificationsTree<CreateOrder, string>(
        PropertyExpression);
    string ExpectedErrorMessage = ErrorMessages.IsRequired;

    Tree.IsRequired();

    var Entity = new CreateOrder { CustomerId = null };

    // Act
    bool Result = Tree.Specifications[0].IsSatisfiedBy(Entity);

    // Assert
    Assert.False(Result);
    Assert.Equal(ExpectedErrorMessage,
        Tree.Specifications[0].Errors.First().ErrorMessage);
}
```

9. Ejecuta las pruebas y verifica que pasen con éxito.

Tarea 10.2: Implementar la validación HasFixedLength

1. En el directorio *PropertySpecificationTreeExtensions*, agrega un nuevo archivo de clase llamado ***HasFixedLengthExtension.cs*** y remplaza el contenido de la clase por el siguiente.

```
public static class HasFixedLenghtSpecification
{
    public static PropertySpecificationsTree<T, string> HasFixedLength<T>(
        this PropertySpecificationsTree<T, string> tree,
        int length, string errorMessage = null)
    {
        tree.Specifications.Add(new Specification<T>(entity =>
        PropertySpecificationTreeHelper.Validate(entity, tree, (errors, value) =>
        {
            if (value == null || value.Length != length)
            {
                errors.Add(new SpecificationError(tree.PropertyName,
                    errorMessage ??
                    string.Format(ErrorMessages.HasFixedLength, length)));
            }
        })));

        return tree;
    }
```

```
}
```

Este código define un método de extensión *HasFixedLength* para la clase *PropertySpecificationsTree<T, string>*, que agrega una validación que asegura que el valor de una propiedad tenga una longitud fija especificada. Si la propiedad no cumple con esta longitud, se agrega un mensaje de error (ya sea el proporcionado por el usuario o uno predeterminado).

2. Agrega el siguiente código a la clase *CreateOrder* del proyecto de pruebas.

```
public const string CustomerIdFixedLength =
    "The length of the client Id must be 5.";
```

3. En el directorio *PropertySpecificationTreeExtensionsTests*, agrega un nuevo archivo de clase llamado ***HasFixedLengthTests.cs*** y remplaza el contenido de la clase por el siguiente para implementar las pruebas unitarias del método *HasFixedLength*. Puedes notar que el código es similar al implementado previamente para el método *IsRequired*.

```
public class HasFixedLenghtTests
{
    Expression<Func<CreateOrder, string>> PropertyExpression =
        (x => x.CustomerId);

    [Theory]
    [InlineData(null, 5, false)]
    [InlineData("", 5, false)]
    [InlineData("ALF", 5, false)]
    [InlineData("ALFKI", 5, true)]
    public void HasFixedLenght_ShouldReturnExpectedResult_WhenValueIsChecked(
        string customerId, int length, bool expectedResult)
    {
        // Arrange
        var Tree = new PropertySpecificationsTree<CreateOrder, string>(
            PropertyExpression);
        Tree.HasFixedLength(length);

        var Entity = new CreateOrder { CustomerId = customerId };

        // Act
        bool Result = Tree.Specifications[0].IsSatisfiedBy(Entity);

        // Assert
        Assert.Equal(expectedResult, Result);

        if (!expectedResult)
        {
            Assert.Single(Tree.Specifications[0].Errors);
        }
        else
        {
            Assert.True(Tree.Specifications[0].Errors == null ||
```

```
                !Tree.Specifications[0].Errors.Any());
        }
    }

    [Fact]
    public void
HasFixedLenght_ShouldUseProvidedErrorMessage_WhenValidationFails()
    {
        // Arrange
        var Tree = new PropertySpecificationsTree<CreateOrder, string>(
            PropertyExpression);
        string ExpectedMessage = CreateOrder.CustomerIdFixedLength;

        Tree.HasFixedLength(5, ExpectedMessage);

        var Entity = new CreateOrder { CustomerId = "" };

        // Act
        bool Result = Tree.Specifications[0].IsSatisfiedBy(Entity);

        // Assert
        Assert.False(Result);
        Assert.Equal(ExpectedMessage,
            Tree.Specifications[0].Errors.First().ErrorMessage);
    }

    [Fact]
    public void HasFixedLenght_ShouldUseDefaultErrorMessage_WhenValidationFails()
    {
        // Arrange
        var Tree = new PropertySpecificationsTree<CreateOrder, string>(
            PropertyExpression);
        string ExpectedMessage = string.Format(ErrorMessages.HasFixedLength, 5);

        Tree.HasFixedLength(5);

        var Entity = new CreateOrder { CustomerId = "" };

        // Act
        bool Result = Tree.Specifications[0].IsSatisfiedBy(Entity);

        // Assert
        Assert.False(Result);
        Assert.Equal(ExpectedMessage,
            Tree.Specifications[0].Errors.First().ErrorMessage);
    }
}
```

4. Ejecuta las pruebas y verifica que pasen con éxito.

Tarea 10.3: Implementar la validación HasMaxLength

1. En el directorio *PropertySpecificationTreeExtensions*, agrega un nuevo archivo de clase llamado ***HasMaxLengthExtension.cs*** y remplaza el contenido de la clase por el siguiente.

```
public static class HasMaxLengthExtension
{
    public static PropertySpecificationsTree<T, string> HasMaxLength<T>(
        this PropertySpecificationsTree<T, string> tree,
        int maxLength, string errorMessage = default)
    {
        tree.Specifications.Add(new Specification<T>(entity =>
        PropertySpecificationTreeHelper.Validate(entity, tree, (errors, value) =>
        {
            if (value != null && value.Length > maxLength)
            {
                errors.Add(new SpecificationError(tree.PropertyName,
                    errorMessage ??
                    string.Format(ErrorMessages.HasMaxLength, maxLength)));
            }
        })));

        return tree;
    }
}
```

Este código define un método de extensión *HasMaxLength* para *PropertySpecificationsTree<T, string>*, que agrega una validación para garantizar que una propiedad no exceda una longitud máxima especificada. Si la validación falla, se registra un mensaje de error personalizado o uno predeterminado.

2. Agrega el siguiente código a la clase *CreateOrder*.

```
public const string CustomerIdHasMaxLength =
    "The length of the client ID must be a maximum of 5 characters.";
```

3. El en directorio *PropertySpecificationTreeExtensionsTests*, agrega un nuevo archivo de clase llamado ***HasMaxLengthTests.cs*** y remplaza el contenido de la clase por el siguiente.

```
public class HasMaxLengthTests
{
    Expression<Func<CreateOrder, string>> PropertyExpression =
        (x => x.CustomerId);

    [Theory]
    [InlineData("ALFKIS", 5, false)]
    [InlineData(null, 5, true)]
    [InlineData("", 5, true)]
    [InlineData("ALF", 5, true)]
    [InlineData("ALFKI", 5, true)]
    public void HasMaxLength_ShouldReturnExpectedResult_WhenValueIsChecked(
        string customerId, int length, bool expectedResult)
    {
        // Arrange
        var Tree = new PropertySpecificationsTree<CreateOrder, string>(
            PropertyExpression);
        Tree.HasMaxLength(length);
```

```
        var Entity = new CreateOrder { CustomerId = customerId };

        // Act
        bool Result = Tree.Specifications[0].IsSatisfiedBy(Entity);

        // Assert
        Assert.Equal(expectedResult, Result);

        if (!expectedResult)
        {
            Assert.Single(Tree.Specifications[0].Errors);
        }
        else
        {
            Assert.True(Tree.Specifications[0].Errors == null ||
                !Tree.Specifications[0].Errors.Any());
        }
    }

    [Fact]
    public void HasMaxLength_ShouldUseProvidedErrorMessage_WhenValidationFails()
    {
        // Arrange
        var Tree = new PropertySpecificationsTree<CreateOrder, string>(
            PropertyExpression);
        string ExpectedMessage = CreateOrder.CustomerIdHasMaxLength;

        Tree.HasMaxLength(5, ExpectedMessage);

        var Entity = new CreateOrder { CustomerId = "ALFKIS" };

        // Act
        bool Result = Tree.Specifications[0].IsSatisfiedBy(Entity);

        // Assert
        Assert.False(Result);
        Assert.Equal(ExpectedMessage,
            Tree.Specifications[0].Errors.First().ErrorMessage);
    }

    [Fact]
    public void HasMaxLength_ShouldUseDefaultErrorMessage_WhenValidationFails()
    {
        // Arrange
        var Tree = new PropertySpecificationsTree<CreateOrder, string>(
            PropertyExpression);
        string ExpectedMessage = string.Format(ErrorMessages.HasMaxLength, 5);

        Tree.HasMaxLength(5);

        var Entity = new CreateOrder { CustomerId = "ALFKIS" };

        // Act
        bool Result = Tree.Specifications[0].IsSatisfiedBy(Entity);

        // Assert
        Assert.False(Result);
```

```
        Assert.Equal(ExpectedMessage,
            Tree.Specifications[0].Errors.First().ErrorMessage);
    }
}
```

4. Ejecuta las pruebas y verifica que pasen con éxito.

Tarea 10.4: Implementar la validación HasMinLenght

1. En el directorio *PropertySpecificationTreeExtensions*, agrega un nuevo archivo de clase llamado ***HasMinLengthExtension.cs*** y remplaza el contenido de la clase por el siguiente.

```
public static class HasMinLenghtExtension
{
    public static PropertySpecificationsTree<T, string> HasMinLength<T>(
        this PropertySpecificationsTree<T, string> tree,
        int minLength, string errorMessage = default)
    {
        tree.Specifications.Add(new Specification<T>(entity =>
        PropertySpecificationTreeHelper.Validate(entity, tree, (errors, value) =>
        {
            if (value == null || value.Length < minLength)
            {
                errors.Add(new SpecificationError(tree.PropertyName,
                    errorMessage ??
                    string.Format(ErrorMessages.HasMinLength, minLength)));
            }
        })));

        return tree;
    }
}
```

Este código define un método de extensión para validar que una propiedad de tipo *string* tenga una longitud mínima dentro de un árbol de especificaciones de propiedad.

2. Agrega el siguiente código a la clase *CreateOrder*.

```
public const string CustomerIdHasMinLength =
    "The length of the client ID must be at least 5 characters.";
```

3. El en directorio *PropertySpecificationTreeExtensionsTests*, agrega un nuevo archivo de clase llamado ***HasMinLengthTests.cs*** y remplaza el contenido de la clase por el siguiente.

```
public class HasMinLengthTests
{
    Expression<Func<CreateOrder, string>> PropertyExpression =
        (x => x.CustomerId);
```

```
    [Theory]
    [InlineData(null, 5, false)]
    [InlineData("", 5, false)]
    [InlineData("ALF", 5, false)]
    [InlineData("ALFKI", 5, true)]
    [InlineData("ALFKIS", 5, true)]
    public void HasMinLength_ShouldReturnExpectedResult_WhenValueIsChecked(
        string customerId, int length, bool expectedResult)
    {
        // Arrange
        var Tree = new PropertySpecificationsTree<CreateOrder, string>(
            PropertyExpression);
        Tree.HasMinLength(length);

        var Entity = new CreateOrder { CustomerId = customerId };

        // Act
        bool Result = Tree.Specifications[0].IsSatisfiedBy(Entity);

        // Assert
        Assert.Equal(expectedResult, Result);

        if (!expectedResult)
        {
            Assert.Single(Tree.Specifications[0].Errors);
        }
        else
        {
            Assert.True(Tree.Specifications[0].Errors == null ||
                !Tree.Specifications[0].Errors.Any());
        }
    }

    [Fact]
    public void HasMinLength_ShouldUseProvidedErrorMessage_WhenValidationFails()
    {
        // Arrange
        var Tree = new PropertySpecificationsTree<CreateOrder, string>(
            PropertyExpression);
        string ExpectedMessage = CreateOrder.CustomerIdHasMinLength;

        Tree.HasMinLength(5, ExpectedMessage);

        var Entity = new CreateOrder { CustomerId = "ALF" };

        // Act
        bool Result = Tree.Specifications[0].IsSatisfiedBy(Entity);

        // Assert
        Assert.False(Result);
        Assert.Equal(ExpectedMessage,
            Tree.Specifications[0].Errors.First().ErrorMessage);
    }

    [Fact]
    public void HasMinLength_ShouldUseDefaultErrorMessage_WhenValidationFails()
    {
```

```
        // Arrange
        var Tree = new PropertySpecificationsTree<CreateOrder, string>(
            PropertyExpression);
        string ExpectedMessage = string.Format(ErrorMessages.HasMinLength, 5);

        Tree.HasMinLength(5);

        var Entity = new CreateOrder { CustomerId = "ALF" };

        // Act
        bool Result = Tree.Specifications[0].IsSatisfiedBy(Entity);

        // Assert
        Assert.False(Result);
        Assert.Equal(ExpectedMessage,
            Tree.Specifications[0].Errors.First().ErrorMessage);
    }
}
```

4. Ejecuta las pruebas y verifica que pasen con éxito.

Tarea 10.5: Implementar la validación Matches

1. En el directorio *PropertySpecificationTreeExtensions*, agrega un nuevo archivo de clase llamado ***MatchesExtension.cs*** y remplaza el contenido de la clase por el siguiente.

```
public static class MatchesExtension
{
    public static PropertySpecificationsTree<T, string> Matches<T>(
        this PropertySpecificationsTree<T, string> tree,
        string regularExpression, string errorMessage = default)
    {
        tree.Specifications.Add(new Specification<T>(entity =>
        PropertySpecificationTreeHelper.Validate(entity, tree, (errors, value) =>
        {
            if (value == null || !Regex.IsMatch(value, regularExpression))
            {
                errors.Add(new SpecificationError(tree.PropertyName,
                    errorMessage ??
                    string.Format(ErrorMessages.Matches, regularExpression)));
            }
        })));

        return tree;
    }
}
```

Este código define un método de extensión para validar si una propiedad de tipo *string* cumple con una expresión regular dentro de un árbol de especificaciones de propiedad.

2. Agrega el siguiente código a la clase *CreateOrder*.

```
public const string CustomerIdMatch =
    "The customer ID must have 5 digits.";
```

3. El en directorio *PropertySpecificationTreeExtensionsTests*, agrega un nuevo archivo de clase llamado ***MatchesTests.cs*** y remplaza el contenido de la clase por el siguiente.

```
public class MatchesTests
{
    Expression<Func<CreateOrder, string>> PropertyExpression =
        x => x.CustomerId;

    [Theory]
    [InlineData(null, "^([0-9]{5})$", false)]
    [InlineData("", "^([0-9]{5})$", false)]
    [InlineData("123", "^([0-9]{5})$", false)]
    [InlineData("123456", "^([0-9]{5})$", false)]
    [InlineData("12345", "^([0-9]{5})$", true)]
    public void Matches_ShouldReturnExpectedResult_WhenValueIsChecked(
        string customerId, string regularExpression, bool expectedResult)
    {
        // Arrange
        var Tree = new PropertySpecificationsTree<CreateOrder, string>(
            PropertyExpression);
        Tree.Matches(regularExpression);

        var Entity = new CreateOrder { CustomerId = customerId };

        // Act
        bool result = Tree.Specifications[0].IsSatisfiedBy(Entity);

        // Assert
        Assert.Equal(expectedResult, result);

        if (!expectedResult)
        {
            Assert.Single(Tree.Specifications[0].Errors);
        }
        else
        {
            Assert.True(Tree.Specifications[0].Errors == null ||
                !Tree.Specifications[0].Errors.Any());
        }
    }

    [Fact]
    public void Matches_ShouldUseProvidedErrorMessage_WhenValidationFails()
    {
        // Arrange
        var Tree = new PropertySpecificationsTree<CreateOrder, string>(
            PropertyExpression);
        string ExpectedErrorMessage = CreateOrder.CustomerIdMatch;
        string RegularExpression = "^([0-9]{5})$";

        Tree.Matches(RegularExpression, ExpectedErrorMessage);
```

```
        var Entity = new CreateOrder { CustomerId = "" };

        // Act
        bool Result = Tree.Specifications[0].IsSatisfiedBy(Entity);

        // Assert
        Assert.False(Result);
        Assert.Equal(ExpectedErrorMessage,
            Tree.Specifications[0].Errors.First().ErrorMessage);
    }

    [Fact]
    public void Matches_ShouldUseDefaultErrorMessage_WhenValidationFails()
    {
        // Arrange
        var Tree = new PropertySpecificationsTree<CreateOrder, string>(
            PropertyExpression);
        string RegularExpression = "^([0-9]{5})$";
        string ExpectedErrorMessage =
            string.Format(ErrorMessages.Matches, RegularExpression);

        Tree.Matches(RegularExpression);

        var Entity = new CreateOrder { CustomerId = "" };

        // Act
        bool Result = Tree.Specifications[0].IsSatisfiedBy(Entity);

        // Assert
        Assert.False(Result);
        Assert.Equal(ExpectedErrorMessage,
            Tree.Specifications[0].Errors.First().ErrorMessage);
    }
}
```

4. Ejecuta las pruebas y verifica que pasen con éxito.

Tarea 10.6: Implementar la validación GreaterThan

1. En el directorio *PropertySpecificationTreeExtensions*, agrega un nuevo archivo de clase llamado ***GreaterThanExtension.cs*** y remplaza el contenido de la clase por el siguiente.

```
public static class GreaterThanExtension
{
    public static PropertySpecificationsTree<T, TProperty>
        GreaterThan<T, TProperty>(
        this PropertySpecificationsTree<T, TProperty> tree,
        TProperty comparisonValue,
        string errorMessage = default) where TProperty : IComparable<TProperty>
    {
        tree.Specifications.Add(new Specification<T>(entity =>
        PropertySpecificationTreeHelper.Validate(entity, tree, (errors, value) =>
        {
            var ComparableValue =
```

```
                (TProperty)Convert.ChangeType(value, typeof(TProperty));

            if (ComparableValue.CompareTo(comparisonValue) <= 0)
            {
                errors.Add(new SpecificationError(tree.PropertyName,
                    errorMessage ??
                    string.Format(ErrorMessages.GreaterThan, comparisonValue)));
            }
        }));

        return tree;
    }
}
```

Este código define un método de extensión que permite agregar una validación para verificar si el valor de una propiedad es mayor que un valor de comparación (comparisonValue).

TProperty, el tipo de la propiedad a validar debe implementar *IComparable<TProperty>* para permitir comparaciones.

2. Agrega un nuevo archivo de clase llamado ***CreateOrderDetail.cs*** en el directorio *Models* del proyecto de pruebas unitarias y remplaza el contenido de la clase por el siguiente.

```
public class CreateOrderDetail
{
    public const string ProductIdMessage = "Required product.";
    public const string UnitPriceMessage = "Price required.";
    public const string QuantityMessage = "Quantity required.";

    public int ProductId { get; set; }
    public double UnitPrice { get; set; }
    public short Quantity { get; set; }
}
```

3. El en directorio *PropertySpecificationTreeExtensionsTests*, agrega un nuevo archivo de clase llamado ***GreaterThanTests.cs*** y remplaza el contenido de la clase por el siguiente.

```
public class GreaterThanTests
{
    Expression<Func<CreateOrderDetail, int>> PropertyExpression =
        (x => x.ProductId);

    [Theory]
    [InlineData(0, 0, false)]
    [InlineData(1, 0, true)]
    public void GreaterThan_ShouldReturnExpectedResult_WhenValueIsChecked(
        int productId, int comparisonValue, bool expectedResult)
    {
        // Arrange
        var Tree = new PropertySpecificationsTree<CreateOrderDetail, int>(
            PropertyExpression);
        Tree.GreaterThan(comparisonValue);
```

```
        var Entity = new CreateOrderDetail { ProductId = productId };

        // Act
        bool Result = Tree.Specifications[0].IsSatisfiedBy(Entity);

        // Assert
        Assert.Equal(expectedResult, Result);

        if (!expectedResult)
        {
            Assert.Single(Tree.Specifications[0].Errors);
        }
        else
        {
            Assert.True(Tree.Specifications[0].Errors == null ||
                !Tree.Specifications[0].Errors.Any());
        }
    }

    [Fact]
    public void GreaterThan_ShouldUseProvidedErrorMessage_WhenValidationFails()
    {
        // Arrange
        var Tree = new PropertySpecificationsTree<CreateOrderDetail, int>(
            PropertyExpression);
        string ExpectedMessage = CreateOrderDetail.ProductIdMessage;

        Tree.GreaterThan(0, ExpectedMessage);

        var Entity = new CreateOrderDetail { ProductId = 0 };

        // Act
        bool Result = Tree.Specifications[0].IsSatisfiedBy(Entity);

        // Assert
        Assert.False(Result);
        Assert.Equal(ExpectedMessage,
            Tree.Specifications[0].Errors.First().ErrorMessage);
    }

    [Fact]
    public void GreaterThan_ShouldUseDefaultErrorMessage_WhenValidationFails()
    {
        // Arrange
        var Tree = new PropertySpecificationsTree<CreateOrderDetail, int>(
            PropertyExpression);
        string ExpectedMessage = string.Format(ErrorMessages.GreaterThan, 0);

        Tree.GreaterThan(0);

        var Entity = new CreateOrderDetail { ProductId = 0 };

        // Act
        bool Result = Tree.Specifications[0].IsSatisfiedBy(Entity);

        // Assert
        Assert.False(Result);
```

```
        Assert.Equal(ExpectedMessage,
            Tree.Specifications[0].Errors.First().ErrorMessage);
    }
}
```

4. Ejecuta las pruebas y verifica que pasen con éxito.

Tarea 10.7: Implementar la validación GreaterThanOrEqualTo

1. En el directorio *PropertySpecificationTreeExtensions*, agrega un nuevo archivo de clase llamado ***GreaterThanOrEqualToExtension.cs*** y remplaza el contenido de la clase por el siguiente.

```
public static class GreaterThanOrEqualToExtension
{
    public static PropertySpecificationsTree<T, TProperty>
        GreaterThanOrEqualTo<T, TProperty>(
        this PropertySpecificationsTree<T, TProperty> tree,
        TProperty comparisonValue,
        string errorMessage = default) where TProperty : IComparable<TProperty>
    {
        tree.Specifications.Add(new Specification<T>(entity =>
        PropertySpecificationTreeHelper.Validate(entity, tree, (errors, value) =>
        {
            var ComparableValue =
                (TProperty)Convert.ChangeType(value, typeof(TProperty));

            if (ComparableValue.CompareTo(comparisonValue) < 0)
            {
                errors.Add(new SpecificationError(tree.PropertyName,
                    errorMessage ?? string.Format(
                        ErrorMessages.GreaterThanOrEqualTo, comparisonValue)));
            }
        })));

        return tree;
    }
}
```

Este código define un método de extensión que permite agregar una validación para verificar si el valor de una propiedad es mayor o igual que un valor de comparación (*comparisonValue*).

TProperty, el tipo de la propiedad a validar debe implementar *IComparable<TProperty>* para permitir comparaciones.

2. El en directorio *PropertySpecificationTreeExtensionsTests*, agrega un nuevo archivo de clase llamado ***GreaterThanOrEqualToTests.cs*** y remplaza el contenido de la clase por el siguiente.

```
public class GreaterThanOrEqualToTests
{
```

```
    Expression<Func<CreateOrderDetail, int>> PropertyExpression =
        (x => x.ProductId);

    [Theory]
    [InlineData(0, 1, false)]
    [InlineData(1, 1, true)]
    [InlineData(2, 1, true)]
    public void
GreaterThanOrEqualTo_ShouldReturnExpectedResult_WhenValueIsChecked(
        int productId, int comparisonValue, bool expectedResult)
    {
        // Arrange
        var Tree = new PropertySpecificationsTree<CreateOrderDetail, int>(
            PropertyExpression);
        Tree.GreaterThanOrEqualTo(comparisonValue);

        var Entity = new CreateOrderDetail { ProductId = productId };

        // Act
        bool Result = Tree.Specifications[0].IsSatisfiedBy(Entity);

        // Assert
        Assert.Equal(expectedResult, Result);

        if (!expectedResult)
        {
            Assert.Single(Tree.Specifications[0].Errors);
        }
        else
        {
            Assert.True(Tree.Specifications[0].Errors == null ||
                !Tree.Specifications[0].Errors.Any());
        }
    }

    [Fact]
    public void
GreaterThanOrEqualTo_ShouldUseProvidedErrorMessage_WhenValidationFails()
    {
        // Arrange
        var Tree = new PropertySpecificationsTree<CreateOrderDetail, int>(
            PropertyExpression);
        string ExpectedMessage = CreateOrderDetail.ProductIdMessage;

        Tree.GreaterThanOrEqualTo(1, ExpectedMessage);

        var Entity = new CreateOrderDetail { ProductId = 0 };

        // Act
        bool Result = Tree.Specifications[0].IsSatisfiedBy(Entity);

        // Assert
        Assert.False(Result);
        Assert.Equal(ExpectedMessage,
            Tree.Specifications[0].Errors.First().ErrorMessage);
    }

    [Fact]
```

```
    public void
GreaterThanOrEqualTo_ShouldUseDefaultErrorMessage_WhenValidationFails()
    {
        // Arrange
        var Tree = new PropertySpecificationsTree<CreateOrderDetail, int>(
            PropertyExpression);
        string ExpectedMessage =
            string.Format(ErrorMessages.GreaterThanOrEqualTo, 1);

        Tree.GreaterThanOrEqualTo(1);

        var Entity = new CreateOrderDetail { ProductId = 0 };

        // Act
        bool Result = Tree.Specifications[0].IsSatisfiedBy(Entity);

        // Assert
        Assert.False(Result);
        Assert.Equal(ExpectedMessage,
            Tree.Specifications[0].Errors.First().ErrorMessage);
    }
}
```

3. Ejecuta las pruebas y verifica que pasen con éxito.

Tarea 10.8: Implementar la validación Equal

1. En el directorio *PropertySpecificationTreeExtensions*, agrega un nuevo archivo de clase llamado ***EqualExtension.cs*** y remplaza el contenido de la clase por el siguiente.

```
public static class EqualExtension
{
    public static PropertySpecificationsTree<T, TProperty> Equal<T, TProperty>(
        this PropertySpecificationsTree<T, TProperty> tree,
        TProperty comparisonValue,
        string errorMessage = default) where TProperty : IComparable<TProperty>
    {
        AddSpecification(tree, (entity) => comparisonValue, errorMessage);

        return tree;
    }

    public static PropertySpecificationsTree<T, TProperty> Equal<T, TProperty>(
        this PropertySpecificationsTree<T, TProperty> tree,
        Expression<Func<T, TProperty>> ComparisonProperty,
        string errorMessage = default) where TProperty : IComparable<TProperty>
    {
        AddSpecification(tree, (entity) =>
            ComparisonProperty.Compile().Invoke(entity), errorMessage);

        return tree;
    }

    static void AddSpecification<T, TProperty>(
```

```
        PropertySpecificationsTree<T, TProperty> tree,
        Func<T, TProperty> getcomparisonValue,
        string errorMessage) where TProperty : IComparable<TProperty>
    {
        tree.Specifications.Add(new Specification<T>(entity =>
        PropertySpecificationTreeHelper.Validate(entity, tree, (errors, value) =>
        {
            TProperty ComparisonValue = getcomparisonValue(entity);
            bool AddError;

            if (value == null && ComparisonValue == null)
            {
                AddError = false;
            }
            else if (value == null || ComparisonValue == null)
            {
                AddError = true;
            }
            else
            {
                // Evitar conversión innecesaria si value ya es del tipo correcto
                TProperty ComparableValue = value is TProperty TPropertyValue
                    ? TPropertyValue
                    : (TProperty)Convert.ChangeType(value, typeof(TProperty));

                AddError = ComparableValue.CompareTo(ComparisonValue) != 0;
            }

            if (AddError)
            {
                errors.Add(new SpecificationError(tree.PropertyName,
                    errorMessage ?? ErrorMessages.Equal));
            }
        })));
    }
}
```

La clase *EqualExtension* define dos métodos de extensión que permiten validar si el valor de una propiedad es igual a un valor dado (primer método) o a otra propiedad de la entidad (segundo método).

El método auxiliar *AddSpecification* se encarga de implementar la lógica de validación. A través del delegado *getcomparisonValue* puede obtener el valor de comparación que puede ser un valor proporcionado en el primer método o el valor de una propiedad proporcionado en el segundo método.

2. Agrega un nuevo archivo de clase llamado ***UserRegistration.cs*** en el directorio *Models* del proyecto de prueba y remplaza el contenido de la clase por el siguiente.

```
public class UserRegistration
{
    public const string EmailErrorMessage =
        "Invalid email format";
```

```
    public const string IsRequiredErrorMessage =
        "Required data";
    public const string HasMinLengthErrorMessage =
        "At least 6 characters are required";
    public const string UppercaseCharactersAreRequiredErrorMessage =
        "Capital characters are required";
    public const string LowercaseCharactersAreRequiredErrorMessage =
        "Lowercase characters are required";
    public const string DigitsAreRequiredErrorMessage =
        "Digits are required";
    public const string SymbolsAreRequiredErrorMessage =
        "Symbols are required";
    public const string PasswordConfirmationDoesNotMatchErrorMessage =
        "Password confirmation does not match";

    public string Email { get; set; }
    public string Password { get; set; }
    public string ConfirmPassword { get; set; }
}
```

3. El en directorio *PropertySpecificationTreeExtensionsTests*, agrega un nuevo archivo de clase llamado ***EqualTests.cs*** y remplaza el contenido de la clase por el siguiente.

```
public class EqualTests
{
    [Theory]
    [InlineData(0, 1, false)]
    [InlineData(1, 1, true)]
    public void Equal_ShouldReturnExpectedResult_WhenNumericValueIsChecked(
        int productId, int comparisonValue, bool expectedResult)
    {
        // Arrange
        var Tree = new PropertySpecificationsTree<CreateOrderDetail, int>(
            x => x.ProductId);
        Tree.Equal(comparisonValue);

        var Entity = new CreateOrderDetail { ProductId = productId };

        // Act
        bool Result = Tree.Specifications[0].IsSatisfiedBy(Entity);

        // Assert
        Assert.Equal(expectedResult, Result);

        if (!expectedResult)
        {
            Assert.Single(Tree.Specifications[0].Errors);
        }
        else
        {
            Assert.True(Tree.Specifications[0].Errors == null ||
                !Tree.Specifications[0].Errors.Any());
        }
    }

    [Theory]
```

```
    [InlineData(null, "ALFKI", false)]
    [InlineData("ALFKI", null, false)]
    [InlineData("ALF", "ALFKI", false)]
    [InlineData(null, null, true)]
    [InlineData("", "", true)]
    [InlineData("ALFKI", "ALFKI", true)]
    public void Equal_ShouldReturnExpectedResult_WhenStringValueIsChecked(
        string customerId, string comparisonValue, bool expectedResult)
    {
        // Arrange
        var Tree = new PropertySpecificationsTree<CreateOrder, string>(
            x => x.CustomerId);
        Tree.Equal(comparisonValue);

        var Entity = new CreateOrder { CustomerId = customerId };

        // Act
        bool Result = Tree.Specifications[0].IsSatisfiedBy(Entity);

        // Assert
        Assert.Equal(expectedResult, Result);

        if (!expectedResult)
        {
            Assert.Single(Tree.Specifications[0].Errors);
        }
        else
        {
            Assert.True(Tree.Specifications[0].Errors == null ||
                !Tree.Specifications[0].Errors.Any());
        }
    }

    [Theory]
    [InlineData(null, "Mm1234567$", false)]
    [InlineData("Mm1234567$", null, false)]
    [InlineData("Mm1234567$", "Mm1234567", false)]
    [InlineData(null, null, true)]
    [InlineData("", "", true)]
    [InlineData("Mm1234567$", "Mm1234567$", true)]
    public void Equal_ShouldReturnExpectedResult_WhenPropertiesAreChecked(
        string password, string confirmPassword, bool expectedResult)
    {
        // Arrange
        var Tree = new PropertySpecificationsTree<UserRegistration, string>(
            x => x.Password);

        Tree.Equal(x => x.ConfirmPassword);

        var Entity = new UserRegistration
        {
            Password = password,
            ConfirmPassword = confirmPassword
        };

        // Act
        bool Result = Tree.Specifications[0].IsSatisfiedBy(Entity);
```

```
        // Assert
        Assert.Equal(expectedResult, Result);

        if (!expectedResult)
        {
            Assert.Single(Tree.Specifications[0].Errors);
        }
        else
        {
            Assert.True(Tree.Specifications[0].Errors == null ||
                !Tree.Specifications[0].Errors.Any());
        }
    }

    [Fact]
    public void Equal_ShouldUseProvidedErrorMessage_WhenValidationFails()
    {
        // Arrange
        var Tree = new PropertySpecificationsTree<UserRegistration, string>(
            x => x.ConfirmPassword);
        string ExpectedMessage =
            UserRegistration.PasswordConfirmationDoesNotMatchErrorMessage;

        Tree.Equal(x => x.Password, ExpectedMessage);

        var Entity = new UserRegistration
        {
            Password = "Mm1234567$",
            ConfirmPassword = "Mm1234567"
        };

        // Act
        bool Result = Tree.Specifications[0].IsSatisfiedBy(Entity);

        // Assert
        Assert.False(Result);
        Assert.Equal(ExpectedMessage,
            Tree.Specifications[0].Errors.First().ErrorMessage);
    }

    [Fact]
    public void Equal_ShouldUseDefaultErrorMessage_WhenValidationFails()
    {
        // Arrange
        var Tree = new PropertySpecificationsTree<UserRegistration, string>(
            x => x.ConfirmPassword);
        string ExpectedMessage = ErrorMessages.Equal;

        Tree.Equal(x => x.Password);

        var Entity = new UserRegistration
        {
            Password = "Mm1234567$",
            ConfirmPassword = "Mm1234567"
        };

        // Act
        bool Result = Tree.Specifications[0].IsSatisfiedBy(Entity);
```

```
        // Assert
        Assert.False(Result);
        Assert.Equal(ExpectedMessage,
            Tree.Specifications[0].Errors.First().ErrorMessage);
    }
}
```

4. Ejecuta las pruebas y verifica que pasen con éxito.

Tarea 10.9: Implementar la validación EmailAddress

1. En el directorio *PropertySpecificationTreeExtensions*, agrega un nuevo archivo de clase llamado ***EmailAddressExtension.cs*** y remplaza el contenido de la clase por el siguiente.

```
public static partial class EmailAddressExtension
{
    public static PropertySpecificationsTree<T, string> EmailAddress<T>(
        this PropertySpecificationsTree<T, string> tree,
        string errorMessage = default)
    {
        tree.Specifications.Add(new Specification<T>(entity =>
        PropertySpecificationTreeHelper.Validate(entity, tree, (errors, value) =>
        {
            if (string.IsNullOrWhiteSpace(value) ||
                !EmailRegex().IsMatch(value))
            {
                errors.Add(new SpecificationError(
                    tree.PropertyName, errorMessage ??
                    ErrorMessages.EmailAddress));
            }
        })));

        return tree;
    }

    [GeneratedRegex(@"^[^@\s]+@[^@\s]+\.[^@\s]+$")]
    private static partial Regex EmailRegex();
}
```

La clase *EmailAddressExtension* define un método de extensión que permite validar si una propiedad de tipo *string* contiene una dirección de correo electrónico válida según una expresión regular.

`[GeneratedRegex]` es un atributo introducido en .NET 7 que permite al compilador generar una implementación optimizada de una expresión regular en tiempo de compilación. Para evitar errores indicando que el método `partial` debe implementarse, es necesario guardar los cambios en el archivo y permitir que el compilador genere automáticamente el código correspondiente.

2. El en directorio *PropertySpecificationTreeExtensionsTests*, agrega un nuevo archivo de clase llamado ***EmailAddressTests.cs*** y remplaza el contenido de la clase por el siguiente.

```
public class EmailAddressTests
{
    Expression<Func<UserRegistration, string>> PropertyExpression =
        x => x.Email;

    [Theory]
    [InlineData(null, false)]
    [InlineData("  ", false)]
    [InlineData("@hotmail.com", false)]
    [InlineData("name@hotmail", false)]
    [InlineData(".com", false)]
    [InlineData("name@hotmail.com", true)]
    public void EmailAddress_ShouldReturnExpectedResult_WhenValueIsChecked(
        string email, bool expectedResult)
    {
        // Arrange
        var Tree = new PropertySpecificationsTree<UserRegistration, string>(
            PropertyExpression);
        Tree.EmailAddress();

        var Entity = new UserRegistration { Email = email };

        // Act
        bool result = Tree.Specifications[0].IsSatisfiedBy(Entity);

        // Assert
        Assert.Equal(expectedResult, result);

        if (!expectedResult)
        {
            Assert.Single(Tree.Specifications[0].Errors);
        }
        else
        {
            Assert.True(Tree.Specifications[0].Errors == null ||
                !Tree.Specifications[0].Errors.Any());
        }
    }

    [Fact]
    public void EmailAddress_ShouldUseProvidedErrorMessage_WhenValidationFails()
    {
        // Arrange
        var Tree = new PropertySpecificationsTree<UserRegistration, string>(
            PropertyExpression);
        string ExpectedErrorMessage = UserRegistration.EmailErrorMessage;

        Tree.EmailAddress(ExpectedErrorMessage);

        var Entity = new UserRegistration { Email = "name" };

        // Act
        bool Result = Tree.Specifications[0].IsSatisfiedBy(Entity);
```

```
        // Assert
        Assert.False(Result);
        Assert.Equal(ExpectedErrorMessage,
            Tree.Specifications[0].Errors.First().ErrorMessage);
    }

    [Fact]
    public void EmailAddress_ShouldUseDefaultErrorMessage_WhenValidationFails()
    {
        // Arrange
        var Tree = new PropertySpecificationsTree<UserRegistration, string>(
            PropertyExpression);
        string ExpectedErrorMessage = ErrorMessages.EmailAddress;

        Tree.EmailAddress();

        var Entity = new UserRegistration { Email = "name" };

        // Act
        bool Result = Tree.Specifications[0].IsSatisfiedBy(Entity);

        // Assert
        Assert.False(Result);
        Assert.Equal(ExpectedErrorMessage,
            Tree.Specifications[0].Errors.First().ErrorMessage);
    }
}
```

3. Ejecuta las pruebas y verifica que pasen con éxito.

Tarea 10.10: Implementar la validación NotEmpty

1. En el directorio *PropertySpecificationTreeExtensions*, agrega un nuevo archivo de clase llamado ***NotEmptyExtension.cs*** y remplaza el contenido de la clase por el siguiente.

```
public static class NotEmptyExtension
{
    public static PropertySpecificationsTree<T, TProperty>
        NotEmpty<T, TProperty>(
        this PropertySpecificationsTree<T, TProperty> tree,
        string errorMessage = default)
    {
        tree.Specifications.Add(new Specification<T>(entity =>
        PropertySpecificationTreeHelper.Validate(entity, tree, (errors, value) =>
        {
            if (value == null || value is IEnumerable collection &&
                !collection.Cast<object>().Any())
            {
                errors.Add(new SpecificationError(tree.PropertyName,
                    errorMessage ?? ErrorMessages.NotEmpty));
            }
        })));

        return tree;
```

```
    }
}
```

La clase *NotEmptyExtension* define un método de extensión que valida que una propiedad de tipo *IEnumerable* no esté vacía.

2. Agrega el siguiente código a la clase *CreateOrder* del directorio *Models* del proyecto de pruebas unitarias.

```
public IEnumerable<CreateOrderDetail> OrderDetails { get; set; }
```

3. Agrega el siguiente código a la clase *CreateOrder*.

```
public const string NotEmpty =
    "At least one item of the order detail is required.";
```

4. El en directorio *PropertySpecificationTreeExtensionsTests*, agrega un nuevo archivo de clase llamado ***NotEmptyTests.cs*** y remplaza el contenido de la clase por el siguiente.

```
public class NotEmptyTests
{
    static Expression<Func<CreateOrder, IEnumerable<CreateOrderDetail>>>
        PropertyExpression = x => x.OrderDetails;

    [Theory]
    [MemberData(nameof(GetTestData))]
    public void NotEmpty_ShouldReturnExpectedResult_WhenValueIsChecked(
        IEnumerable<CreateOrderDetail> details, bool expectedResult)
    {
        // Arrange
        var Tree = new PropertySpecificationsTree<CreateOrder,
            IEnumerable<CreateOrderDetail>>(PropertyExpression);

        Tree.NotEmpty();

        var Entity = new CreateOrder { OrderDetails = details };

        // Act
        bool result = Tree.Specifications[0].IsSatisfiedBy(Entity);

        // Assert
        Assert.Equal(expectedResult, result);

        if (!expectedResult)
        {
            Assert.Single(Tree.Specifications[0].Errors);
        }
        else
        {
            Assert.True(Tree.Specifications[0].Errors == null ||
```

```
                !Tree.Specifications[0].Errors.Any());
        }
    }

    [Fact]
    public void NotEmpty_ShouldUseProvidedErrorMessage_WhenValidationFails()
    {
        // Arrange
        var Tree = new PropertySpecificationsTree<CreateOrder,
            IEnumerable<CreateOrderDetail>>(PropertyExpression);
        string ExpectedErrorMessage = CreateOrder.NotEmpty;

        Tree.NotEmpty(ExpectedErrorMessage);

        var Entity = new CreateOrder { OrderDetails = null };

        // Act
        bool Result = Tree.Specifications[0].IsSatisfiedBy(Entity);

        // Assert
        Assert.False(Result);
        Assert.Equal(ExpectedErrorMessage,
            Tree.Specifications[0].Errors.First().ErrorMessage);
    }

    [Fact]
    public void NotEmpty_ShouldUseDefaultErrorMessage_WhenValidationFails()
    {
        // Arrange
        var Tree = new PropertySpecificationsTree<CreateOrder,
            IEnumerable<CreateOrderDetail>>(PropertyExpression);
        string ExpectedErrorMessage = ErrorMessages.NotEmpty;

        Tree.NotEmpty();

        var Entity = new CreateOrder { OrderDetails = null };

        // Act
        bool Result = Tree.Specifications[0].IsSatisfiedBy(Entity);

        // Assert
        Assert.False(Result);
        Assert.Equal(ExpectedErrorMessage,
            Tree.Specifications[0].Errors.First().ErrorMessage);
    }

    public static IEnumerable<object[]> GetTestData()
    {
        yield return new object[]
        {
            null, false
        };

        yield return new object[]
        {
            new List<CreateOrderDetail>(), false
        };
```

```
        yield return new object[]
        {
            new List<CreateOrderDetail>(){new() }, true
        };
    }
}
```

5. Ejecuta las pruebas y verifica que pasen con éxito.

Tarea 10.11: Implementar la validación Must

1. En el directorio *PropertySpecificationTreeExtensions*, agrega un nuevo archivo de clase llamado ***MustExtension.cs*** y remplaza el contenido de la clase por el siguiente.

```
public static class MustExtension
{
    public static PropertySpecificationsTree<T, TProperty> Must<T, TProperty>(
        this PropertySpecificationsTree<T, TProperty> tree,
        Func<T, bool> predicate, string errorMessage)
    {
        tree.Specifications.Add(new Specification<T>(entity =>
        PropertySpecificationTreeHelper.Validate(entity, tree, (errors, value) =>
        {
            if (!predicate(entity))
            {
                errors.Add(
                    new SpecificationError(tree.PropertyName, errorMessage));
            }
        })));

        return tree;
    }

    public static PropertySpecificationsTree<T, TProperty> Must<T, TProperty>(
        this PropertySpecificationsTree<T, TProperty> tree,
        Func<TProperty, bool> predicate, string errorMessage)
    {
        tree.Specifications.Add(new Specification<T>(entity =>
            PropertySpecificationTreeHelper.Validate(
                entity, tree, (errors, value) =>
            {
                if (value is TProperty propertyValue &&
                    !predicate(propertyValue))
                {
                    errors.Add(
                        new SpecificationError(tree.PropertyName, errorMessage));
                }
            })));

        return tree;
    }
}
```

La clase *MustExtension* proporciona 2 métodos de extensión que permiten agregar validaciones personalizadas a toda una entidad (primer método) o al valor de una propiedad (segundo método).

2. El en directorio *PropertySpecificationTreeExtensionsTests*, agrega un nuevo archivo de clase llamado ***MustTests.cs*** y remplaza el contenido de la clase por el siguiente.

```
public class MustTests
{
    Expression<Func<UserRegistration, string>> PropertyExpression =
        (x => x.Password);

    [Theory]
    [InlineData("", false)]
    [InlineData("12345", false)]
    [InlineData("M", true)]
    public void Must_ShouldReturnExpectedResult_WhenValueIsChecked(
        string passwordValue, bool expectedResult)
    {
        // Arrange
        var Tree = new PropertySpecificationsTree<UserRegistration, string>(
            PropertyExpression);
        Tree.Must(p => p.Any(c => char.IsUpper(c)),
            UserRegistration.UppercaseCharactersAreRequiredErrorMessage);

        var Entity = new UserRegistration { Password = passwordValue };

        // Act
        bool Result = Tree.Specifications[0].IsSatisfiedBy(Entity);

        // Assert
        Assert.Equal(expectedResult, Result);

        if (!expectedResult)
        {
            Assert.Single(Tree.Specifications[0].Errors);
        }
        else
        {
            Assert.True(Tree.Specifications[0].Errors == null ||
                !Tree.Specifications[0].Errors.Any());
        }
    }

    [Theory]
    [InlineData("Admin", "Admin", false)]
    [InlineData("Admin@name.com", "Password", true)]
    public void Must_ShouldReturnExpectedResult_WhenEntityIsChecked(
        string email, string password, bool expectedResult)
    {
        // Arrange
        var Tree = new PropertySpecificationsTree<UserRegistration, string>(
            PropertyExpression);

        Tree.Must((UserRegistration t) =>
```

```
            t.Email == "Admin@name.com" && t.Password == "Password",
            "Error");

        var Entity = new UserRegistration
        {
            Email = email,
            Password = password
        };

        // Act
        bool Result = Tree.Specifications[0].IsSatisfiedBy(Entity);

        // Assert
        Assert.Equal(expectedResult, Result);

        if (!expectedResult)
        {
            Assert.Single(Tree.Specifications[0].Errors);
        }
        else
        {
            Assert.True(Tree.Specifications[0].Errors == null ||
                !Tree.Specifications[0].Errors.Any());
        }
    }

    [Fact]
    public void Must_ShouldUseProvidedErrorMessage_WhenValidationFails()
    {
        // Arrange
        var Tree = new PropertySpecificationsTree<UserRegistration, string>(
            PropertyExpression);

        string ExpectedMessage =
            UserRegistration.UppercaseCharactersAreRequiredErrorMessage;

        Tree.Must(p => p.Any(c => char.IsUpper(c)), ExpectedMessage);

        var Entity = new UserRegistration { Password = "12345" };

        // Act
        bool Result = Tree.Specifications[0].IsSatisfiedBy(Entity);

        // Assert
        Assert.False(Result);
        Assert.Equal(ExpectedMessage,
            Tree.Specifications[0].Errors.First().ErrorMessage);
    }
}
```

3. Ejecuta las pruebas y verifica que pasen con éxito.

Tarea 11: Agregar la implementación de IDomainSpecification

Incorporemos ahora el código base que toda especificación de dominio debe implementar. Esto simplificará la creación de nuevas especificaciones de dominio.

1. Agrega un nuevo archivo de clase llamado ***DomainSpecificationBase.cs*** en el directorio *Core* y remplaza el contenido de la clase por el siguiente.

```
public abstract class DomainSpecificationBase<T> : IDomainSpecification<T>
{
}
```

   Implementaremos esta clase como una base abstracta que solo pueda heredarse, pero no instanciarse directamente.

2. Agrega el siguiente código a la clase para definir un constructor que, opcionalmente, reciba y asigne el valor de la propiedad *EvaluateOnlyIfNoPreviousErrors*.

```
public DomainSpecificationBase(bool evaluateOnlyIfNoPreviousErrors = false)
{
    EvaluateOnlyIfNoPreviousErrors = evaluateOnlyIfNoPreviousErrors;
}

public bool EvaluateOnlyIfNoPreviousErrors { get; }
```

3. Agrega el siguiente código a la clase para definir una propiedad que indique si la evaluación de la especificación de dominio debe detenerse al encontrar el primer error en la validación de una propiedad.

```
public bool StopOnFirstEntitySpecificationError { get; protected set; } = false;
```

4. Agrega el siguiente código a la clase para definir una propiedad que almacenará la lista de errores de la especificación.

```
public IEnumerable<SpecificationError> Errors { get; protected set; }
```

5. Agrega el siguiente código a la clase para definir una variable que almacenará los distintos conjuntos de especificaciones asociados a las propiedades de la entidad en la especificación de dominio.

```
List<IPropertySpecificationsTree<T>> PropertySpecificationsForest = [];
```

6. Agrega el siguiente código a la clase para definir un método que permita crear y registrar un conjunto de especificaciones para una propiedad específica de la entidad.

```
protected PropertySpecificationsTree<T, TProperty> Property<TProperty>(
    Expression<Func<T, TProperty>> propertyExpression,
    bool stopOnFirstPropertySpecificationError = false)
{
    var Tree = new PropertySpecificationsTree<T, TProperty>(
        propertyExpression, stopOnFirstPropertySpecificationError);

    PropertySpecificationsForest.Add(Tree);

    return Tree;
}
```

Puedes notar que el método recibe como parámetro una expresión que accede a una propiedad y un valor booleano opcional que indica si la evaluación de sus especificaciones debe detenerse al encontrar el primer error.

7. Agrega el siguiente código a la clase para crear el método donde implementaremos la validación de los distintos conjuntos de especificaciones.

```
List<SpecificationError> ValidatePropertySpecificationsForest(T entity)
{

}
```

8. Agrega el siguiente código al método para crear una lista vacía que almacenará los errores encontrados durante la validación.

```
List<SpecificationError> SpecificationErrors = [];
```

9. Agrega el siguiente código después del código anterior para obtener un enumerador que utilizaremos para recorrer los distintos conjuntos de especificaciones contenidos en el campo *PropertySpecificationsForest*.

```
var TreesEnumerator = PropertySpecificationsForest.GetEnumerator();
```

10. Agrega el siguiente código después del código anterior para definir una bandera que nos permitirá controlar si se debe continuar con la validación de más árboles cuando se haya encontrado un error de validación en un árbol.

```
bool ContinueValidatingTrees = true;
```

11. Agrega el siguiente código después del código anterior para definir un ciclo que recorra cada conjunto de especificaciones de propiedad definidos para la especificación. La evaluación puede detenerse si la variable *ContinueValidatingTrees* se establece en *false*.

```
while (TreesEnumerator.MoveNext() && ContinueValidatingTrees)
{

}
```

12. Agrega el siguiente código dentro del cuerpo del ciclo *while* para obtener el enumerador que nos permitirá recorrer las especificaciones del conjunto de especificaciones del árbol actual.

```
var TreeSpecificationsEnumerator =
    TreesEnumerator.Current.Specifications.GetEnumerator();
```

13. Agrega el siguiente código después del código anterior para definir una bandera que nos permita controlar si se siguen validando las especificaciones del conjunto actual de especificaciones cuando se haya encontrado un error de validación de alguna especificación.

```
bool ContinueValidatingTreeSpecifications = true;
```

14. Agrega el siguiente código después del código anterior para recorrer todas las especificaciones del conjunto de especificaciones actual. La evaluación puede detenerse si la variable *ContinueValidatingTreeSpecifications* se establece en *false*.

```
while (TreeSpecificationsEnumerator.MoveNext() &&
    ContinueValidatingTreeSpecifications)
{

}
```

15. Dentro del nuevo ciclo *while*, agrega el siguiente código para evaluar si la especificación actual no se satisface con la entidad dada.

```
if (!TreeSpecificationsEnumerator.Current.IsSatisfiedBy(entity))
{

}
```

16. Dentro del cuerpo del *if* anterior, agrega el siguiente código para agregar los errores a la lista *SpecificationErrors* y para que, en el caso de que el conjunto de especificaciones tenga el valor *true* en la propiedad *StopOnFirstPropertySpecificationError*, establezca el valor *false* a la

variable *ContinueValidatingTreeSpecifications* que hará que la validación dentro de ese árbol sea interrumpida.

```
SpecificationErrors.AddRange(
    TreeSpecificationsEnumerator.Current.Errors);

if (TreesEnumerator.Current.StopOnFirstPropertySpecificationError)
    ContinueValidatingTreeSpecifications = false;
```

17. Agrega el siguiente código después de la llave que cierra el cuerpo del segundo *while* para establecer el valor de la variable *ContinueValidatingTrees* en *false* cuando se hayan encontrado errores de validación y la propiedad *StopOnFirstEntitySpecificationError* de la especificación tenga un valor *true* que indica que la validación de la especificación debe detenerse ante el primer error encontrado.

```
if (SpecificationErrors.Count != 0 && StopOnFirstEntitySpecificationError)
    ContinueValidatingTrees = false;
```

18. Agrega el siguiente código después de la llave que cierra el cuerpo del primer *while* para devolver la lista de errores.

```
return SpecificationErrors;
```

El código completo del método será similar al siguiente.

```
List<SpecificationError> ValidatePropertySpecificationsForest(T entity)
{
    List<SpecificationError> SpecificationErrors = [];

    var TreesEnumerator = PropertySpecificationsForest.GetEnumerator();
    bool ContinueValidatingTrees = true;

    while (TreesEnumerator.MoveNext() && ContinueValidatingTrees)
    {
        var TreeSpecificationsEnumerator =
            TreesEnumerator.Current.Specifications.GetEnumerator();
        bool ContinueValidatingTreeSpecifications = true;

        while (TreeSpecificationsEnumerator.MoveNext() &&
            ContinueValidatingTreeSpecifications)
        {
            if (!TreeSpecificationsEnumerator.Current.IsSatisfiedBy(entity))
            {
                SpecificationErrors.AddRange(
                    TreeSpecificationsEnumerator.Current.Errors);

                if (TreesEnumerator.Current
                    .StopOnFirstPropertySpecificationError)
```

```
                    ContinueValidatingTreeSpecifications = false;

            }
        }

        if (SpecificationErrors.Count != 0 &&
            StopOnFirstEntitySpecificationError)
            ContinueValidatingTrees = false;
    }

    return SpecificationErrors;
}
```

La implementación de validación que hemos realizado, valida únicamente especificaciones `ISpecification<T>`. No valida, por ejemplo, una lógica que realice una búsqueda en una base de datos para verificar si un valor existe.

19. Agrega el siguiente código a la clase para definir un método que las clases derivadas puedan sobrescribir para agregar lógica de validación personalizada, por ejemplo, una lógica que realice una búsqueda en una base de datos para verificar si un valor existe.

```
protected virtual Task<List<SpecificationError>> ValidateSpecificationsAsync(
    T entity) => Task.FromResult(new List<SpecificationError>());
```

20. Agrega el siguiente código a la clase para definir el método encargado de ejecutar la validación de la especificación.

```
public async Task<bool> ValidateAsync(T entity)
{
    Errors = [
        .. ValidatePropertySpecificationsForest(entity),
        .. await ValidateSpecificationsAsync(entity) ?? []
        ];

    return !Errors.Any();
}
```

El método *ValidateAsync* invoca a los dos métodos de validación que definimos previamente y emplea la sintaxis de propagación (`..`) para insertar los elementos de las listas de errores devueltas por estos métodos en la colección *Errors*.

Agreguemos ahora las pruebas unitarias.

21. Agrega un nuevo directorio llamado ***DomainSpecificationsTests*** en la raíz del proyecto de pruebas.

22. Agrega un nuevo archivo de clase llamado ***PropertyWithoutStopOnFirstErrorSpecification.cs*** en el directorio *DomainSpecificationsTests* y remplaza el contenido de la clase por el siguiente.

```
internal class PropertyWithoutStopOnFirstErrorSpecification :
    DomainSpecificationBase<UserRegistration>
{
    public PropertyWithoutStopOnFirstErrorSpecification()
    {
        Property(u => u.Password)
            .HasMinLength(6, UserRegistration.HasMinLengthErrorMessage)
            .Must(p => p.Any(c => char.IsUpper(c)),
                UserRegistration.UppercaseCharactersAreRequiredErrorMessage)
            .Must(p => p.Any(c => char.IsLower(c)),
                UserRegistration.LowercaseCharactersAreRequiredErrorMessage)
            .Must(p => p.Any(c => char.IsDigit(c)),
                UserRegistration.DigitsAreRequiredErrorMessage)
            .Must(p => p.Any(c => !char.IsLetterOrDigit(c)),
                UserRegistration.SymbolsAreRequiredErrorMessage);
    }
}
```

Esta clase es un ejemplo de una especificación de dominio que incluye un conjunto de especificaciones para una propiedad. De manera predeterminada, el valor del parámetro *stopOnFirstPropertySpecificationError* para este conjunto de especificaciones está establecido en *false*, lo que permitirá que todas las especificaciones se evalúen sin interrumpir la validación al encontrar el primer error.

23. Agrega un nuevo archivo de clase llamado ***PropertyWithStopOnFirstErrorSpecification.cs*** en el directorio *DomainSpecificationsTests* y remplaza el contenido de la clase por el siguiente.

```
internal class PropertyWithStopOnFirstErrorSpecification :
    DomainSpecificationBase<UserRegistration>
{
    public PropertyWithStopOnFirstErrorSpecification()
    {
        Property(u => u.Password, true)
            .HasMinLength(6, UserRegistration.HasMinLengthErrorMessage)
            .Must(p => p.Any(c => char.IsUpper(c)),
                UserRegistration.UppercaseCharactersAreRequiredErrorMessage)
            .Must(p => p.Any(c => char.IsLower(c)),
                UserRegistration.LowercaseCharactersAreRequiredErrorMessage)
            .Must(p => p.Any(c => char.IsDigit(c)),
                UserRegistration.DigitsAreRequiredErrorMessage)
            .Must(p => p.Any(c => !char.IsLetterOrDigit(c)),
                UserRegistration.SymbolsAreRequiredErrorMessage);
    }
}
```

Al establecer el valor del parámetro *stopOnFirstPropertySpecificationError* en *true* en esta especificación de dominio, la evaluación de las especificaciones se detendrá tan pronto como se encuentre el primer error de validación.

24. Agrega un nuevo archivo de clase llamado ***PropertyStopOnFirstErrorTests.cs*** en el directorio *DomainSpecificationsTests* y remplaza el contenido de la clase por el siguiente.

```
public class PropertyStopOnFirstErrorTests
{
    [Theory]
    [MemberData(nameof(GetTestData))]
    public async Task ValidateAsync_ShouldReturnExpectedResult_WhenValidate(
        IDomainSpecification<UserRegistration> specification,
        UserRegistration entity,
        bool expectedResult,
        IEnumerable<SpecificationError> expectedErrors)
    {

        // Act
        var Result = await specification.ValidateAsync(entity);

        // Assert
        Assert.Equal(expectedResult, Result);

        if (Result == false)
        {
            var ExpectedErrorsOrdered =
                expectedErrors.OrderBy(e => e.PropertyName)
                .ThenBy(e => e.ErrorMessage);
            var ActualErrorsOrdered =
                specification.Errors.OrderBy(e => e.PropertyName)
                .ThenBy(e => e.ErrorMessage);

            Assert.Collection(
                ActualErrorsOrdered,
                ExpectedErrorsOrdered
                .Select(expected => (Action<SpecificationError>)(actual =>
                {
                    Assert.Equal(expected.PropertyName, actual.PropertyName);
                    Assert.Equal(expected.ErrorMessage, actual.ErrorMessage);
                })
                ).ToArray()
            );
        }
        else
        {
            Assert.True(specification.Errors == null ||
                !specification.Errors.Any());
        }
    }

    public static IEnumerable<object[]> GetTestData()
    {
        yield return new object[]
        {
```

```
            new PropertyWithoutStopOnFirstErrorSpecification(),
            new UserRegistration()
            {
                Email = "name@hotmail.com",
                Password = "Mm1234567$",
                ConfirmPassword="Mm1234567$"
            },
            true,
            new List<SpecificationError>()
        };

        yield return new object[]
        {
            new PropertyWithoutStopOnFirstErrorSpecification(),
            new UserRegistration(){Password = ""},
            false,
            new List<SpecificationError>() {
                new SpecificationError("Password",
                    UserRegistration.HasMinLengthErrorMessage),
                new SpecificationError("Password",
                    UserRegistration.UppercaseCharactersAreRequiredErrorMessage),
                new SpecificationError("Password",
                    UserRegistration.LowercaseCharactersAreRequiredErrorMessage),
                new SpecificationError("Password",
                    UserRegistration.DigitsAreRequiredErrorMessage),
                new SpecificationError("Password",
                    UserRegistration.SymbolsAreRequiredErrorMessage)
            }
        };

        yield return new object[]
        {
            new PropertyWithStopOnFirstErrorSpecification(),
            new UserRegistration()
            {
                Email = "name@hotmail.com",
                Password = "Mm1234567$",
                ConfirmPassword="Mm1234567$"},
            true,
            new List<SpecificationError>()
        };

        yield return new object[]
        {
            new PropertyWithStopOnFirstErrorSpecification(),
            new UserRegistration(){Password = "1234567"},
            false,
            new List<SpecificationError>() {
                new SpecificationError("Password",
                UserRegistration.UppercaseCharactersAreRequiredErrorMessage)
            }
        };

        yield return new object[]
        {
            new PropertyWithStopOnFirstErrorSpecification(),
            new UserRegistration(){Password = "MmABCDEF"},
            false,
```

```
            new List<SpecificationError>() {
                new SpecificationError("Password",
                UserRegistration.DigitsAreRequiredErrorMessage)
            }
        };
    }
}
```

25. Ejecuta las pruebas y verifica que pasen con éxito.

    Agreguemos ahora las pruebas para verificar el comportamiento del valor de la propiedad *StopOnFirstEntitySpecificationError* de la especificación de dominio.

26. En el directorio *DomainSpecificationsTests*, agrega un nuevo archivo de clase llamado ***SpecificationWithoutStopOnFirstErrorSpecification.cs*** y remplaza el contenido de la clase por el siguiente.

```
internal class SpecificationWithoutStopOnFirstErrorSpecification :
    DomainSpecificationBase<UserRegistration>
{
    public SpecificationWithoutStopOnFirstErrorSpecification()
    {
        Property(e => e.Email)
            .IsRequired(UserRegistration.IsRequiredErrorMessage);

        Property(u => u.Password)
            .HasMinLength(6, UserRegistration.HasMinLengthErrorMessage);

        Property(u => u.ConfirmPassword)
            .Equal(u => u.Password,
            UserRegistration.PasswordConfirmationDoesNotMatchErrorMessage);
    }
}
```

    Esta clase es un ejemplo de una especificación de dominio que incluye varios conjuntos de especificaciones para distintas propiedades. De manera predeterminada, el valor de *StopOnFirstEntitySpecificationError* de esta especificación de dominio está establecido en *false*, lo que permitirá que todos los conjuntos de especificaciones se evalúen sin interrumpir la validación al encontrar el primer error.

27. En el directorio *DomainSpecificationsTests*, agrega un nuevo archivo de clase llamado ***SpecificationWithStopOnFirstErrorSpecification.cs*** y remplaza el contenido de la clase por el siguiente.

```
internal class SpecificationWithStopOnFirstErrorSpecification :
    DomainSpecificationBase<UserRegistration>
{
    public SpecificationWithStopOnFirstErrorSpecification()
    {
```

```
        StopOnFirstEntitySpecificationError = true;

        Property(e => e.Email)
           .IsRequired(UserRegistration.IsRequiredErrorMessage);

        Property(u => u.Password)
            .HasMinLength(6, UserRegistration.HasMinLengthErrorMessage);

        Property(u => u.ConfirmPassword)
            .Equal(u => u.Password,
            UserRegistration.PasswordConfirmationDoesNotMatchErrorMessage);
    }
}
```

El valor de la propiedad *StopOnFirstEntitySpecificationError* de esta especificación de dominio está establecido en *true*, lo que hará que la validación se detenga al encontrar el primer error.

28. Agrega un nuevo archivo de clase llamado ***SpecificationStopOnFirstErrorTests.cs*** en el directorio *DomainSpecificationsTests* y remplaza el contenido de la clase por el siguiente.

```
public class SpecificationStopOnFirstErrorTests
{
    [Theory]
    [MemberData(nameof(GetTestData))]
    public async Task ValidateAsync_ShouldReturnExpectedResult_WhenValidate(
        IDomainSpecification<UserRegistration> specification,
        UserRegistration entity,
        bool expectedResult,
        IEnumerable<SpecificationError> expectedErrors)
    {
        // Act
        var Result = await specification.ValidateAsync(entity);

        // Assert
        Assert.Equal(expectedResult, Result);

        if (Result == false)
        {
            var ExpectedErrorsOrdered =
                expectedErrors.OrderBy(e => e.PropertyName)
                .ThenBy(e => e.ErrorMessage);
            var ActualErrorsOrdered =
                specification.Errors.OrderBy(e => e.PropertyName)
                .ThenBy(e => e.ErrorMessage);

            Assert.Collection(
                ActualErrorsOrdered,
                ExpectedErrorsOrdered
                .Select(expected => (Action<SpecificationError>)(actual =>
                {
                    Assert.Equal(expected.PropertyName, actual.PropertyName);
                    Assert.Equal(expected.ErrorMessage, actual.ErrorMessage);
                })
                ).ToArray()
            );
```

```
        }
        else
        {
            Assert.True(specification.Errors == null ||
                !specification.Errors.Any());
        }
    }

    public static IEnumerable<object[]> GetTestData()
    {
        yield return new object[]
        {
            new SpecificationWithoutStopOnFirstErrorSpecification(),
            new UserRegistration()
            {
                Email = "name@hotmail.com",
                Password = "Mm1234567$",
                ConfirmPassword="Mm1234567$"
            },
            true,
            new List<SpecificationError>()
        };

        yield return new object[]
        {
            new SpecificationWithoutStopOnFirstErrorSpecification(),
            new UserRegistration()
            {
                Email = "",
                Password = "123",
                ConfirmPassword = "456"
            },
            false,
            new List<SpecificationError>()
            {
                new SpecificationError("Email",
                    UserRegistration.IsRequiredErrorMessage),
                new SpecificationError("Password",
                    UserRegistration.HasMinLengthErrorMessage),
                new SpecificationError("ConfirmPassword",
                  UserRegistration.PasswordConfirmationDoesNotMatchErrorMessage),
            }
        };

        yield return new object[]
        {
            new SpecificationWithStopOnFirstErrorSpecification(),
            new UserRegistration()
            {
                Email = "name@hotmail.com",
                Password = "Mm1234567$",
                ConfirmPassword="Mm1234567$"
            },
            true,
            new List<SpecificationError>()
        };

        yield return new object[]
```

```
        {
            new SpecificationWithStopOnFirstErrorSpecification(),
            new UserRegistration()
            {
                Email = "",
                Password = "123",
                ConfirmPassword="456"
            },
            false,
            new List<SpecificationError>()
            {
                new SpecificationError("Email",
                UserRegistration.IsRequiredErrorMessage),
            }
        };

        yield return new object[]
        {
            new SpecificationWithStopOnFirstErrorSpecification(),
            new UserRegistration()
            {
                Email = "name@hotmail.com",
                Password = "123",
                ConfirmPassword="456"
            },
            false,
            new List<SpecificationError>()
            {
                new SpecificationError("Password",
                    UserRegistration.HasMinLengthErrorMessage),
            }
        };

        yield return new object[]
        {
            new SpecificationWithStopOnFirstErrorSpecification(),
            new UserRegistration()
            {
                Email = "name@hotmail.com",
                Password = "Mm1234567$",
                ConfirmPassword="$Mm1234567"
            },
            false,
            new List<SpecificationError>()
            {
                new SpecificationError("ConfirmPassword",
                  UserRegistration.PasswordConfirmationDoesNotMatchErrorMessage),
            }
        };
    }
}
```

29. Ejecuta las pruebas y verifica que pasen con éxito.

Tarea 12: Agregar la implementación de IDomainSpecificationsValidator

1. Agrega un nuevo archivo de clase llamado ***DomainSpecificationsValidator.cs*** en el directorio *Core* y remplaza el contenido de la clase por el siguiente.

```
public class DomainSpecificationsValidator<T>(
    IEnumerable<IDomainSpecification<T>> specifications) :
IDomainSpecificationsValidator<T>
{
    public async Task<ValidationResult> ValidateAsync(T entity)
    {

    }
}
```

2. Agrega el siguiente código en el método *ValidateAsync* para definir una lista que almacenará los distintos errores de validación que puedan encontrarse.

```
List<SpecificationError> Errors = [];
```

3. Agrega el siguiente código después del anterior para definir una variable que almacene las especificaciones que deben validarse siempre.

```
var UnconditionalSpecifications =
    specifications.Where(spec => !spec.EvaluateOnlyIfNoPreviousErrors);
```

4. Agrega el siguiente código después del anterior para definir una variable que almacene las especificaciones que deben validarse mientras no se haya encontrado un error de validación en alguna especificación previa.

```
var ConditionalSpecifications =
    specifications.Where(spec => spec.EvaluateOnlyIfNoPreviousErrors);
```

5. Agrega el siguiente código después del anterior para realizar las validaciones de las especificaciones que deban evaluarse siempre.

```
foreach (var Specification in UnconditionalSpecifications)
{
    if (!await Specification.ValidateAsync(entity))
        Errors.AddRange(Specification.Errors);
}
```

6. Agrega el siguiente código después del anterior para evaluar las especificaciones que deben evaluarse mientras no se encuentre un error de validación.

```
if (Errors.Count == 0)
{
    var Enumerator = ConditionalSpecifications.GetEnumerator();
    bool IsValid = true;

    while (Enumerator.MoveNext() && IsValid)
    {
        IsValid = await Enumerator.Current.ValidateAsync(entity);
        if (!IsValid)
            Errors.AddRange(Enumerator.Current.Errors);
    }
}
```

7. Agrega el siguiente código después del anterior para devolver la lista de posibles errores de validación encapsulada en una instancia de *ValidationResult*.

```
return new ValidationResult(Errors);
```

El código completo del método será similar al siguiente.

```
public async Task<ValidationResult> ValidateAsync(T entity)
{
    List<SpecificationError> Errors = [];

    var UnconditionalSpecifications =
        specifications.Where(spec => !spec.EvaluateOnlyIfNoPreviousErrors);

    var ConditionalSpecifications =
        specifications.Where(spec => spec.EvaluateOnlyIfNoPreviousErrors);

    foreach (var Specification in UnconditionalSpecifications)
    {
        if (!await Specification.ValidateAsync(entity))
            Errors.AddRange(Specification.Errors);
    }

    if (Errors.Count == 0)
    {
        var Enumerator = ConditionalSpecifications.GetEnumerator();
        bool IsValid = true;

        while (Enumerator.MoveNext() && IsValid)
        {
            IsValid = await Enumerator.Current.ValidateAsync(entity);
            if (!IsValid)
                Errors.AddRange(Enumerator.Current.Errors);
        }
    }
```

```
    return new ValidationResult(Errors);
}
```

8. Agrega un nuevo directorio llamado ***DomainSpecificationsValidatorTests*** en la raíz del proyecto de pruebas.
9. En el directorio *DomainSpecificationsValidatorTests*, agrega un nuevo archivo de clase llamado ***UnConditionalEmailSpecification.cs*** y remplaza el contenido de la clase por el siguiente.

```
internal class UnConditionalEmailSpecification :
    DomainSpecificationBase<UserRegistration>
{
    public UnConditionalEmailSpecification()
    {
        Property(u => u.Email)
            .IsRequired(UserRegistration.IsRequiredErrorMessage);
    }
}
```

10. En el directorio *DomainSpecificationsValidatorTests*, agrega un nuevo archivo de clase llamado ***UnConditionalPasswordSpecification.cs*** y remplaza el contenido de la clase por el siguiente.

```
internal class UnConditionalPasswordSpecification :
    DomainSpecificationBase<UserRegistration>
{
    public UnConditionalPasswordSpecification()
    {
        Property(u => u.Password)
            .HasMinLength(6, UserRegistration.HasMinLengthErrorMessage);
    }
}
```

11. En el directorio *DomainSpecificationsValidatorTests*, agrega un nuevo archivo de clase llamado ***UnConditionalConfirmPasswordSpecification.cs*** y remplaza el contenido de la clase por el siguiente.

```
internal class UnConditionalConfirmPasswordSpecification :
    DomainSpecificationBase<UserRegistration>
{
    public UnConditionalConfirmPasswordSpecification()
    {
        Property(u => u.ConfirmPassword)
            .Equal(u => u.Password,
                UserRegistration.PasswordConfirmationDoesNotMatchErrorMessage);
    }
}
```

12. En el directorio *DomainSpecificationsValidatorTests*, agrega un nuevo archivo de clase llamado ***ConditionalEmailSpecification.cs*** y remplaza el contenido de la clase por el siguiente.

```
internal class ConditionalEmailSpecification :
    DomainSpecificationBase<UserRegistration>
{
    public ConditionalEmailSpecification() : base(true)
    {
        Property(u => u.Email)
            .IsRequired(UserRegistration.IsRequiredErrorMessage);
    }
}
```

13. En el directorio *DomainSpecificationsValidatorTests*, agrega un nuevo archivo de clase llamado ***ConditionalPasswordSpecification.cs*** y remplaza el contenido de la clase por el siguiente.

```
internal class ConditionalPasswordSpecification :
    DomainSpecificationBase<UserRegistration>
{
    public ConditionalPasswordSpecification() : base(true)
    {
        Property(u => u.Password)
            .HasMinLength(6, UserRegistration.HasMinLengthErrorMessage);
    }
}
```

14. En el directorio *DomainSpecificationsValidatorTests*, agrega un nuevo archivo de clase llamado ***ConditionalUniqueEmailSpecification.cs*** y remplaza el contenido de la clase por el siguiente.

```
internal class ConditionalUniqueEmailSpecification :
    DomainSpecificationBase<UserRegistration>
{
    public const string DuplicateEmailMessage =
        "The email provided already exists.";
    public const string TestEmail = "user@northwind.com";
    public ConditionalUniqueEmailSpecification() : base(true) { }

    protected override async Task<List<SpecificationError>>
        ValidateSpecificationsAsync(
        UserRegistration entity)
    {
        List<SpecificationError> Errors = [];

        // Simular una operación asíncrona, por ejemplo,
        // una búsqueda en una base de datos
        await Task.Delay(1000);
        bool DuplicateEmail = TestEmail.Equals(entity.Email);

        if (DuplicateEmail)
            Errors.Add(new SpecificationError("Email", DuplicateEmailMessage));
```

```
        return Errors;
    }
}
```

15. En el directorio *DomainSpecificationsValidatorTests*, agrega un nuevo archivo de clase llamado ***TestData.cs*** y remplaza el contenido de la clase por el siguiente.

```
internal static class TestData
{
    public static IEnumerable<Object[]> GetTestData()
    {
        yield return new Object[]
        {
            new IDomainSpecification<UserRegistration>[]
            {
                new UnConditionalEmailSpecification()
            },
            new UserRegistration
            {
                Email= "name@hotmail.com"
            },
            new ValidationResult([])
        };

        yield return new Object[]
        {
            new IDomainSpecification<UserRegistration>[]
            {
                new UnConditionalEmailSpecification()
            },
            new UserRegistration
            {
                Email= ""
            },
            new ValidationResult([
                new SpecificationError("Email",
                    UserRegistration.IsRequiredErrorMessage)
                ])
        };

        yield return new Object[]
        {
            new IDomainSpecification<UserRegistration>[]
            {
                new UnConditionalEmailSpecification(),
                new UnConditionalPasswordSpecification(),
                new UnConditionalConfirmPasswordSpecification()
            },
            new UserRegistration
            {
                Email= "name@hotmail.com",
                Password="Mm1234567$",
                ConfirmPassword="Mm1234567$"
            },
            new ValidationResult([])
        };
```

```
        yield return new Object[]
        {
            new IDomainSpecification<UserRegistration>[]
            {
                new UnConditionalEmailSpecification(),
                new UnConditionalPasswordSpecification(),
                new UnConditionalConfirmPasswordSpecification()
            },
            new UserRegistration
            {
                Email= null,
                Password="123",
                ConfirmPassword="456"
            },
            new ValidationResult([
                new SpecificationError("Email",
                    UserRegistration.IsRequiredErrorMessage),
                new SpecificationError("Password",
                    UserRegistration.HasMinLengthErrorMessage),
                new SpecificationError("ConfirmPassword",
                   UserRegistration.PasswordConfirmationDoesNotMatchErrorMessage)
                ])
        };

        yield return new Object[]
        {
            new IDomainSpecification<UserRegistration>[]
            {
                new ConditionalEmailSpecification(),
                new ConditionalPasswordSpecification(),
                new ConditionalUniqueEmailSpecification()
            },
            new UserRegistration
            {
                Email= "name@hotmail.com",
                Password="Mm1234567$"
            },
            new ValidationResult([])
        };

        yield return new Object[]
        {
            new IDomainSpecification<UserRegistration>[]
            {
                new ConditionalEmailSpecification(),
                new ConditionalPasswordSpecification(),
                new ConditionalUniqueEmailSpecification()
            },
            new UserRegistration
            {
                Email= null,
                Password="123"
            },
            new ValidationResult([
                new SpecificationError("Email",
                    UserRegistration.IsRequiredErrorMessage)
                ])
```

```
        };

        yield return new Object[]
        {
            new IDomainSpecification<UserRegistration>[]
            {
                new UnConditionalEmailSpecification(),
                new ConditionalPasswordSpecification(),
                new ConditionalUniqueEmailSpecification()
            },
            new UserRegistration
            {
                Email= "name@hotmail.com",
                Password="Mm1234567$"
            },
            new ValidationResult([])
        };

        yield return new Object[]
        {
            new IDomainSpecification<UserRegistration>[]
            {
                new UnConditionalEmailSpecification(),
                new ConditionalPasswordSpecification(),
                new ConditionalUniqueEmailSpecification()
            },
            new UserRegistration
            {
                Email= null,
                Password="123"
            },
            new ValidationResult([
                new SpecificationError("Email",
                    UserRegistration.IsRequiredErrorMessage)
                ])
        };

        yield return new Object[]
        {
            new IDomainSpecification<UserRegistration>[]
            {
                new UnConditionalEmailSpecification(),
                new ConditionalPasswordSpecification(),
                new ConditionalUniqueEmailSpecification()
            },
            new UserRegistration
            {
                Email= ConditionalUniqueEmailSpecification.TestEmail,
                Password="123"
            },
            new ValidationResult([
                new SpecificationError("Password",
                    UserRegistration.HasMinLengthErrorMessage)
                ])
        };

        yield return new Object[]
        {
```

```
            new IDomainSpecification<UserRegistration>[]
            {
                new UnConditionalEmailSpecification(),
                new ConditionalPasswordSpecification(),
                new ConditionalUniqueEmailSpecification()
            },
            new UserRegistration
            {
                Email= ConditionalUniqueEmailSpecification.TestEmail,
                Password="Mm1234567$"
            },
            new ValidationResult([
                new SpecificationError("Email",
                ConditionalUniqueEmailSpecification.DuplicateEmailMessage)
                ])
        };
    }
}
```

16. En el directorio *DomainSpecificationsValidatorTests*, agrega un nuevo archivo de clase llamado ***DomainSpecificationsValidatorTests.cs*** y remplaza el contenido de la clase por el siguiente.

```
public class DomainSpecificationsValidatorTests
{
    [Theory]
    [MemberData(nameof(TestData.GetTestData),
        MemberType = typeof(TestData))]
    public async Task
ValidateAsync_ShouldReturnExpectedResult_WhenValidateDomainSpecifications(
        IDomainSpecification<UserRegistration>[] specifications,
        UserRegistration user,
        ValidationResult expectedResult)
    {
        // Arrange
        IDomainSpecificationsValidator<UserRegistration> Validator =
            new DomainSpecificationsValidator<UserRegistration>(specifications);

        // Act
        var Result = await Validator.ValidateAsync(user);

        // Arrange
        Assert.Equal(expectedResult.IsValid, Result.IsValid);

        if (Result.IsValid == false)
        {
            var ExpectedErrorsOrdered =
                expectedResult.Errors.OrderBy(e => e.PropertyName)
                    .ThenBy(e => e.ErrorMessage);
            var ActualErrorsOrdered =
                Result.Errors.OrderBy(e => e.PropertyName)
                    .ThenBy(e => e.ErrorMessage);

            Assert.Collection(
                ActualErrorsOrdered,
                ExpectedErrorsOrdered
                .Select(expected => (Action<SpecificationError>)(actual =>
```

```
                {
                    Assert.Equal(expected.PropertyName, actual.PropertyName);
                    Assert.Equal(expected.ErrorMessage, actual.ErrorMessage);
                })
                ).ToArray()
            );
        }
        else
        {
            Assert.True(Result.Errors == null || !Result.Errors.Any());
        }
    }
}
```

17. Ejecuta las pruebas y verifica que pasen con éxito.

Tarea 13: Implementar la validación SetValidator

Ahora agregaremos una extensión de validación para permitir la validación de los elementos de una colección dentro de una entidad.

1. En el directorio *PropertySpecificationTreeExtensions*, agrega un nuevo archivo de clase llamado ***SetValidatorExtension.cs*** y remplaza el contenido de la clase por el siguiente.

```
public static class SetValidatorExtension
{
    public static PropertySpecificationsTree<T, IEnumerable<TElement>>
        SetValidator<T, TElement>(
        this PropertySpecificationsTree<T, IEnumerable<TElement>> tree,
        IDomainSpecificationsValidator<TElement> validator)
    {
        tree.Specifications.Add(new Specification<T>(entity =>
        PropertySpecificationTreeHelper.Validate(entity, tree, (errors, value) =>
        {
            if (value is IEnumerable<TElement> Collection)
            {
                int Index = 0;
                foreach (var Item in Collection)
                {
                    var Result = validator.ValidateAsync(Item)
                        .GetAwaiter().GetResult();

                    if (!Result.IsValid)
                    {
                        foreach (var Error in Result.Errors)
                        {
                            errors.Add(new SpecificationError(
                            $"{tree.PropertyName}[{Index}].{Error.PropertyName}",
                            Error.ErrorMessage));
                        }
                    }

                    Index++;
```

```
                }
            }
        }));

        return tree;
    }
}
```

El método *SetValidator* extiende *PropertySpecificationsTree*, permitiendo agregar una validación personalizada para cada elemento de la colección utilizando una instancia de *IDomainSpecificationsValidator*.

Durante la validación, se obtiene el valor de la propiedad correspondiente y se verifica si es una colección (*IEnumerable<TElement>*). Se recorre cada elemento de la colección, validándolo de forma individual con `validator.ValidateAsync(Item)`.

El método `GetAwaiter().GetResult()` se utiliza para bloquear la ejecución de forma síncrona hasta obtener el resultado de la validación asíncrona, evitando la necesidad de utilizar código `async/await` en el contexto síncrono. Este enfoque se prefiere sobre el uso de `validator.ValidateAsync(Item).Result` ya que `.GetAwaiter().GetResult()` lanza directamente la excepción original en caso de error, mientras que `.Result` envuelve la excepción en un `AggregateException`, lo que complica el manejo de errores.

Si el resultado de la validación indica errores, cada error se agrega a la lista de errores, incluyendo:

- El nombre de la propiedad de la colección.
- El índice del elemento dentro de la colección.
- El nombre de la propiedad del error específico.
- El mensaje de error correspondiente.

Por ejemplo, si *tree.PropertyName* es "*OrderDetails*" y el segundo elemento de la colección tiene un error en la propiedad "*ProductId*", el nombre de propiedad con error será `OrderDetails[1].ProductId`.

Este enfoque garantiza que cada error de validación se relacione correctamente con su posición dentro de la colección, facilitando la identificación y corrección de problemas.

2. Agrega un nuevo directorio llamado ***SetValidatorTests*** en la raíz del proyecto de pruebas unitarias.
3. Agrega un nuevo archivo de clase llamado ***CreateOrderDetailSpecification.cs*** en el directorio *SetValidatorTests* y remplaza el contenido de la clase por el siguiente.

```
internal class CreateOrderDetailSpecification :
DomainSpecificationBase<CreateOrderDetail>
```

```
{
    public CreateOrderDetailSpecification()
    {
        Property(d => d.ProductId)
            .GreaterThan(0, CreateOrderDetail.ProductIdMessage);

        Property(d => d.Quantity)
            .GreaterThan((short)0, CreateOrderDetail.QuantityMessage);

        Property(d => d.UnitPrice)
            .GreaterThan(0, CreateOrderDetail.UnitPriceMessage);
    }
}
```

4. Agrega un nuevo archivo de clase llamado ***CreateOrderSpecification.cs*** en el directorio *SetValidatorTests* y remplaza el contenido de la clase por el siguiente.

```
internal class CreateOrderSpecification : DomainSpecificationBase<CreateOrder>
{
    public CreateOrderSpecification(
        IDomainSpecificationsValidator<CreateOrderDetail> orderDetailsValidator)
    {
        Property(o => o.CustomerId)
            .IsRequired();

        Property(o => o.OrderDetails)
            .NotEmpty();

        Property(o => o.OrderDetails)
            .SetValidator(orderDetailsValidator);
    }
}
```

5. Agrega un nuevo archivo de clase llamado ***SetValidatorTestData.cs*** en el directorio *SetValidatorTests* y remplaza el contenido de la clase por el siguiente.

```
internal class SetValidatorTestData
{
    public static IEnumerable<object[]> GetTestData()
    {
        yield return new object[]
        {
            new CreateOrder
            {
                CustomerId = "ALFKI",
                OrderDetails = [
                    new CreateOrderDetail
                    {
                        ProductId = 1,
                        Quantity = 2,
                        UnitPrice = 3}
                    ]},
            new ValidationResult([])
```

```
        };

        yield return new object[]
        {
            new CreateOrder
            {
                CustomerId = "" ,
                OrderDetails = [
                    new CreateOrderDetail
                    {
                        ProductId = 1,
                        Quantity = 2,
                        UnitPrice = 3
                    }
                    ]},
            new ValidationResult([
                new SpecificationError("CustomerId",
                    ErrorMessages.IsRequired)
                ])
        };


        yield return new object[]
        {
            new CreateOrder
            {
                CustomerId = "ALFKI",
                OrderDetails = null
            },
            new ValidationResult([
                new SpecificationError("OrderDetails",
                    ErrorMessages.NotEmpty)
                ])
        };

        yield return new object[]
        {
            new CreateOrder
            {
                CustomerId = "ALFKI",
                OrderDetails = []
            },
            new ValidationResult([
                new SpecificationError("OrderDetails",
                    ErrorMessages.NotEmpty)
                ])
        };

        yield return new object[]
        {
            new CreateOrder
            {
                CustomerId = "ALFKI",
                OrderDetails = [
                    new CreateOrderDetail
                    {
                        ProductId = 1,
                        Quantity = 2,
```

```
                        UnitPrice = 3
                    },
                    new CreateOrderDetail
                    {
                        ProductId = 0,
                        Quantity = 2,
                        UnitPrice = 3
                    },
                    new CreateOrderDetail
                    {
                        ProductId = 1,
                        Quantity = 0,
                        UnitPrice = 3
                    },
                    new CreateOrderDetail
                    {
                        ProductId = 1,
                        Quantity = 2,
                        UnitPrice = 0
                    }
                    ]},
            new ValidationResult([
                new SpecificationError("OrderDetails[1].ProductId",
                    CreateOrderDetail.ProductIdMessage),
                new SpecificationError("OrderDetails[2].Quantity",
                    CreateOrderDetail.QuantityMessage),
                new SpecificationError("OrderDetails[3].UnitPrice",
                    CreateOrderDetail.UnitPriceMessage),
            ])
        };
    }
}
```

6. Agrega un nuevo archivo de clase llamado ***SetValidatorTests.cs*** en el directorio *SetValidatorTests* y remplaza el contenido de la clase por el siguiente.

```
public class SetValidatorTests
{
    [Theory]
    [MemberData(nameof(SetValidatorTestData.GetTestData),
    MemberType = typeof(SetValidatorTestData))]
    public async Task
ValidateAsync_ShouldReturnExpectedResult_WhenValidateDomainSpecifications(
        CreateOrder order,
        ValidationResult expectedResult)
    {
        // Arrange
        IDomainSpecificationsValidator<CreateOrderDetail> OrderDetailValidator =
            new DomainSpecificationsValidator<CreateOrderDetail>([
                new CreateOrderDetailSpecification()
                ]);

        IDomainSpecificationsValidator<CreateOrder> Validator =
            new DomainSpecificationsValidator<CreateOrder>([
                new CreateOrderSpecification(OrderDetailValidator)
                ]);
```

```
        // Act
        var Result = await Validator.ValidateAsync(order);

        // Assert
        Assert.Equal(expectedResult.IsValid, Result.IsValid);

        if (Result.IsValid == false)
        {
            var ExpectedErrorsOrdered =
                expectedResult.Errors.OrderBy(e => e.PropertyName)
                .ThenBy(e => e.ErrorMessage);
            var ActualErrorsOrdered =
                Result.Errors.OrderBy(e => e.PropertyName)
                .ThenBy(e => e.ErrorMessage);

            Assert.Collection(
                ActualErrorsOrdered,
                ExpectedErrorsOrdered
                .Select(expected => (Action<SpecificationError>)(actual =>
                {
                    Assert.Equal(expected.PropertyName, actual.PropertyName);
                    Assert.Equal(expected.ErrorMessage, actual.ErrorMessage);
                })
                ).ToArray()
            );
        }
        else
        {
            Assert.True(Result.Errors == null || !Result.Errors.Any());
        }
    }
}
```

7. Ejecuta las pruebas y verifica que pasen con éxito.

Tarea 14: Agregar métodos de extensión para registro de servicios

Ahora que hemos implementado el framework de validación, agreguemos métodos de extensión que simplifiquen el uso de los servicios de validación.

1. Agrega el paquete NuGet ***Microsoft.Extensions.DependencyInjection.Abstractions*** en el proyecto *DomainValidation.*
2. Agrega un nuevo archivo de clase llamado ***DependencyContainer.cs*** en la raíz del proyecto *DomainValidation* y remplaza el contenido de la clase por el siguiente.

```
public static class DependencyContainer
{
    public static IServiceCollection AddDomainSpecificationsValidator(
        this IServiceCollection services)
    {
        services.TryAddScoped(typeof(IDomainSpecificationsValidator<>),
```

```
            typeof(DomainSpecificationsValidator<>));

        return services;
    }
}
```

3. Importa los espacios de nombres requeridos en el archivo *GlobalUsings.cs*.

Tarea 15: Agregar una excepción personalizada

Implementemos una excepción personalizada que se lance cuando se detecten errores de validación.

1. Agrega un nuevo directorio llamado ***Exceptions*** en la raíz del proyecto *DomainValidation*.
2. Agrega un nuevo archivo de clase llamado ***DomainValidationException.cs*** en el directorio *Exceptions* y remplaza el código de la clase por el siguiente.

```
public class DomainValidationException : Exception
{
    public IReadOnlyList<SpecificationError> Errors { get; }
    public DomainValidationException() { }
    public DomainValidationException(string message) : base(message) { }
    public DomainValidationException(string message, Exception innerException)
        : base(message, innerException) { }
    public DomainValidationException(
        IEnumerable<SpecificationError> errors, string message = default)
        : base(message)
    {
        Errors = [.. errors];
    }
}
```

Tarea 16: Agregar una cláusula *Guard* de validación

Una cláusula ***Guard*** es un bloque de código que interrumpe la ejecución de un método al evaluar una condición. Si la condición no se cumple, el método se detiene mediante una sentencia *return* o lanzando una excepción.

Ahora, agregaremos una cláusula *Guard* para interrumpir la ejecución de un método y lanzar una excepción en caso de que una especificación no supere la validación**.**

1. Agrega un nuevo directorio llamado ***Guards*** en la raíz del proyecto.
2. Agrega un nuevo archivo de clase llamado ***DomainValidationGuard.cs*** en el directorio *Guards* y remplaza el código de la clase por el siguiente.

```
public static class DomainValidationGuard
{
```

```
    public static async Task AgainstInvalidSpecification<T>(
        IDomainSpecificationsValidator<T> validator, T entity,
        string message = default)
    {
        var ValidationResult = await validator.ValidateAsync(entity);

        if (!ValidationResult.IsValid)
            throw new DomainValidationException(
                ValidationResult.Errors, message);
    }
}
```

El framework está listo para utilizarse en los procesos de validación.

Para utilizar el framework debes crear tus especificaciones y registrarlas en el contenedor de inyección de dependencias junto con el servicio de validación de especificaciones de dominio.

El siguiente es un ejemplo de unas especificaciones de dominio.

```
internal class CreateOrderDetailDtoSpecification :
    DomainSpecificationBase<CreateOrderDetailDto>
{
    public CreateOrderDetailDtoSpecification()
    {
        Property(d => d.ProductId)
            .GreaterThan(0,
                CreateOrderMessages.ProductIdGreaterThanZero);

        Property(d => d.Quantity)
            .GreaterThan((short)0,
                CreateOrderMessages.QuantityGreaterThanZero);

        Property(d => d.UnitPrice)
            .GreaterThan(0,
                CreateOrderMessages.UnitPriceGreaterThanZero);
    }
}
```

```
internal class CreateOrderDtoSpecification :
    DomainSpecificationBase<CreateOrderDto>
{
    public CreateOrderDtoSpecification(
        IDomainSpecificationsValidator<CreateOrderDetailDto> detailValidator)
    {
        Property(c => c.CustomerId)
            .NotEmpty(CreateOrderMessages.CustomerIdRequired)
            .HasFixedLength(5, CreateOrderMessages.CustomerIdRequiredLength);

        Property(c => c.ShipAddress)
            .NotEmpty(CreateOrderMessages.ShipAddressRequired)
            .HasMaxLength(60, CreateOrderMessages.ShipAddressMaximumLength);
```

```
        Property(c => c.ShipCity)
            .NotEmpty(CreateOrderMessages.ShipCityRequired)
            .HasMinLength(3, CreateOrderMessages.ShipCityMinimumLength)
            .HasMaxLength(15, CreateOrderMessages.ShipCityMaximumLength);

        Property(c => c.ShipCountry)
            .NotEmpty(CreateOrderMessages.ShipCountryRequired)
            .HasMinLength(3, CreateOrderMessages.ShipCountryMinimumLength)
            .HasMaxLength(15, CreateOrderMessages.ShipCountryMaximumLength);

        Property(c => c.ShipPostalCode)
            .HasMaxLength(10, CreateOrderMessages.ShipPostalCodeMaximumLength);

        Property(c => c.OrderDetails)
            .NotEmpty(CreateOrderMessages.OrderDetailsNotEmpty);

        Property(c => c.OrderDetails)
            .SetValidator(detailValidator);
    }
}
```

El siguiente es un ejemplo de un método de extensión que registra las especificaciones.

```
public static class DependencyContainer
{
    public static IServiceCollection AddDomainSpecifications(
        this IServiceCollection services)
    {
        services.AddScoped<IDomainSpecification<CreateOrderDetailDto>,
            CreateOrderDetailDtoSpecification>();

        services.AddScoped<IDomainSpecification<CreateOrderDto>,
            CreateOrderDtoSpecification>();

        return services;
    }
}
```

El siguiente es un ejemplo de un método de extensión que registra los servicios de validación.

```
public static class DependencyContainer
{
    public static IServiceCollection AddSalesServices(
        this IServiceCollection services)
    {
        services.AddUseCasesServices()
            .AddDomainSpecificationsValidator()
            .AddDomainSpecifications();

        return services;
    }

}
```

Puedes notar que estamos utilizando el método *AddDomainSpecificationsValidator* que definimos en nuestro framework de validación para registrar la clase que implementa la validación.

El siguiente es un ejemplo que muestra la inyección del servicio de validación en una clase.

```
internal class CreateOrderInteractor(
    ICreateOrderOutputPort outputPort,
    ICommandsRepository repository,
    IDomainSpecificationsValidator<CreateOrderDto> validator) :
    ICreateOrderInputPort
```

El siguiente es un ejemplo del uso del servicio de validación sin el uso de la cláusula *Guard* o la excepción personalizada.

```
var ValidationResult = await validator.ValidateAsync(orderDto);

if (!ValidationResult.IsValid)
{
    string Errors = string.Join(" ", ValidationResult.Errors
        .Select(e => $"{e.PropertyName}: {e.ErrorMessage}"));

    throw new Exception(Errors);
}
```

El siguiente es un ejemplo del uso del servicio de validación utilizando la cláusula *Guard*.

```
await DomainValidationGuard.AgainstInvalidSpecification(validator, orderDto,
    CreateOrderMessages.ValidationExceptionMessage);
```