A SOLID validation class

By Alex Siepman 5. January 2014 09:33

Introduction

A long story today, but very instructive if you do not have much experience with SOLID. I will give you 2 clues in advance:

- Don't implement validations in other objects but give the other objects a list of validations (using Dependency inversion).
- When the validation uses a logical implication (if condition P is true then requirement Q needs to be true), do not implement the condition and the requirement in 1 property, but create a 2 properties (condition and requirement).

Example

Image you have the folowing data transfer class that needs to be validated:

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public bool ConsumesAlcohol { get; set; }
}
```

These validations needs to be created:

1. A person is only allowed to consume alcohol when his or her age is 18 or higher.
2. The age of a person must be 0 of higher.

Worst solution

```
public class AlcoholSeller
{
    // Other codes not shown

    private void Validate(Person person)
    {
        private const int MinimumAge = 18;

        // Previous code forces person can not be null

        if (person.Age < MinimumAge && person.ConsumesAlcohol)
        {
            throw new ValidationException(
                "{0} is not allowed to consume alcohol because his or her age ({1}) is not {2} or high
                person.Name, person.Age, MinimumAge);
        }
        if (person.Age < 0)
        {
            throw new ValidationException("Age of {0} must be higher than 0",
                person.Name);
        }
    }
}
```

using this exception:

```
public class ValidationException : Exception
{
    public ValidationException(string message, params object[] args)
        : base(String.Format(message, args))
    {
    }
}
```

This is not a good (SOLID) solution:

- The Validate method has multiple responsibilities (Single responsibility).
- The Validate method has to change when a new validation will be added (Open/close principle, Dependency inversion).
- It shows only the first message that is invalid.
- The conditions are inverted (negative condition is harder to read).
- This model can not stand when the Person has 20 validations (otherwise the method would be much to long).

An alternative solution would be: 1 class implements 1 validation. Advantages:

- Object orientation possible:
  - Inheritance: 2 different valdiations can use the logic of the same base class.
  - Reuse: For example: validating the age of a customer and a employee can be done with the same validation class: "Age must be 0 or higher".

About the author

I am a software architect at R
Developer. My main interests
LINQ and **generics**

Visit my personal homepage
about me.

Enter search term or APML ur

Month List

2012
2013
2014
2015

Page List

Blogroll

Eric Lippert
Fixing Random, Part 28
Fixing Random, Part 27
Porting Old Posts, Part...
The Morning Brew - Chri...
The Morning Brew #2738
The Morning Brew #2737
The Morning Brew #2736
Ynze Zuidema

  - Polyformism: A collection of validations can be handled easily.
- Methods, constants, variables, etc. only used by the validation can be placed (private) in a class. If a validation is no longer needed, just delete the class and all related code is deleted automatically.
- Unittesting is much easier.

Create a class for a validation

To make polyformism possible, let's use a small interface (interface segregation principle):

```
public interface IValidation
{
    bool IsValid { get; } // True when valid
    void Validate(); // Throws an exception when not valid
    string Message { get; } // The message when object is not valid
}
```

and then create a base class for (most of) the validations

```
public abstract class ValidationBase<T> : IValidation where T : class
{
    protected T Context { get; private set; }

    protected ValidationBase(T context)
    {
        if (context == null)
        {
            throw new ArgumentNullException("context");
        }
        Context = context;
    }

    public void Validate()
    {
        if (!IsValid)
        {
            throw new ValidationException(Message);
        }
    }

    public abstract bool IsValid { get; }
    public abstract string Message { get; }
}
```

The 2 validations can now be rewritten:

```
public class OnlyAdultsCanConsumeAlcoholValidation : ValidationBase<Person>
{
    private const int MinimumAge = 18;

    public OnlyAdultsCanConsumeAlcohol(Person context)
        : base(context)
    {
    }

    public override bool IsValid
    {
        get { return !Context.ConsumesAlcohol || Context.Age >= 18; }
    }

    public override string Message
    {
        get
        {
            return string.Format(
                "{0} is not allowed to consume alcohol because his or her age ({1}) not is {2} or higl
                Context.Name, Context.Age, MinimumAge);
        }
    }
}
```

and

```
public class Age0OrHigherValidation : ValidationBase<Person>
    {
        public Age0OrHigherValidation(Person context)
            : base(context)
        {
        }

        public override bool IsValid
        {
            get { return Context.Age >= 0; }
        }

        public override string Message
        {
            get
            {
                return string.Format("The Age {1} of {0} is not 0 or higher.",
                    Context.Name, Context.Age);
            }
        }
    }
```

Now it can be combined with a special List to make polyformism possible:

```
public class ValidationList : List<IValidation>, IValidationList
{
    public bool IsValid
    {
        get
        {
            return this.All(v => v.IsValid);
        }
    }

    public void Validate()
    {
        foreach (var validation in this)
        {
            validation.Validate();
        }
    }

    public IEnumerable<string> Messages
    {
        get
        {
            return this.Where(v => !v.IsValid).Select(v => v.Message);
        }
    }
}
```

Implementing this small interface (interface segregation principle):

```
public interface IValidationList
{
    bool IsValid { get; }
    void Validate();
    IEnumerable<string> Messages { get; }
}
```

Now the AlcoholSeller class can be loosely coupled from the validations applying Dependency inversion:

```
public class AlcoholSeller
{
    private readonly IValidationList _validationList;

    public AlcoholSeller(IValidationList validationList)
    {
        _validationList = validationList;
    }

    // Other code not shown

    private bool IsValid
    {
        get
        {
            return _validationList.IsValid;
        }
    }

    private IEnumerable<string> Messages
    {
        get
        {
            return _validationList.Messages;
        }
    }
}
```

When new validations are added to validationList, the AlcoholSeller class does not need any change (Open/close principle, Dependency inversion).

Unit tests

Unit test of the validations are now completely separated from the AlcoholSeller class. The unit tests of both the validation classes are separated as wel.

```
[TestClass]
public class OnlyAdultsCanConsumeAlcoholValidationTest
{
    [TestMethod]
    public void Person18AndConsumingAlcohol()
    {
        AssertHelper(18, true, true);
    }

    [TestMethod]
    public void Person17AndConsumingAlcohol()
    {
        AssertHelper(17, true, false);
    }

    [TestMethod]
    public void Person18AndNotConsumingAlcohol()
    {
        AssertHelper(18, false, true);
    }

    [TestMethod]
    public void Person17AndNotConsumingAlcohol()
    {
        AssertHelper(17, false, true);
    }

    private void AssertHelper(int age, bool consumesAlcohol, bool expected)
    {
        var person = new Person { Age = age, ConsumesAlcohol = consumesAlcohol };
        var validation = new OnlyAdultsCanConsumeAlcoholValidation(person);
        Assert.AreEqual(expected, validation.IsValid);
    }
}
```

and

```
[TestClass]
public class Age0OrHigherValidationTest
{
    [TestMethod]
    public void AgeIs0()
    {
        AssertHelper(0, true);
    }

    [TestMethod]
    public void AgeIsBelow0()
    {
        AssertHelper(-1, false);
    }

    private void AssertHelper(int age, bool expected)
    {
        var person = new Person { Age = age };
        var validation = new Age0OrHigherValidation(person);
        Assert.AreEqual(expected, validation.IsValid);
    }
}
```

Logical implication

The OnlyAdultsCanConsumeAlcoholValidation has a IsValid property containing the following implication: If a person consumes alcohol, the person's age needs to be at least 18. This is written as !Condition || requirement:

```
return !Context.ConsumesAlcohol || Context.Age >= 18;
```

I have seen that this line of code implemented by colleagues , but all used different variations. That makes it hard to review and multiple implementations are also a rich source of bugs. Some examples:

```
return !(Context.ConsumesAlcohol && Context.Age < 18);
```

```
if (!Context.ConsumesAlcohol)
{
    return true;
}
if (Context.Age >= 18)
{
    return false;
}
return true;
```

```
if (!Context.ConsumesAlcohol)
{
    return true;
}
return Context.Age < 18;
```

```
if (Context.ConsumesAlcohol)
{
    return Context.Age < 18;
}
return true;
```

```
if (Context.Age >= 18)
{
    return true;
}
return !Context.ConsumesAlcohol;
```

```
if (Context.Age >= 18)
{
    return true;
}
if (Context.ConsumesAlcohol)
```

```
{
    return false;
}
return true;
```

This makes the IsValid propery hard to read and a mistake can easily be made.

The final solution

Implements the implication in a sealed IsInvalid property and split the condition and the requirement in two different properties. The ValidationList and the AlcoholSeller class does not need any change.

The base class:

```
public abstract class ValidationBase<T> : IValidation where T : class
{
    protected T Context { get; private set; }

    protected ValidationBase(T context)
    {
        if (context == null)
        {
            throw new ArgumentNullException("context");
        }
        Context = context;
    }

    public void Validate()
    {
        if (!IsValid)
        {
            throw new ValidationException(Message);
        }
    }

    public virtual bool Condition
    {
        get
        {
            // If the condition is not overriden, the requirent always needs to be true
            return true;
        }
    }

    public abstract bool Requirement { get; }

    public bool IsValid { get { return Implication(Condition, Requirement); } }

    public abstract string Message { get; }

    private static bool Implication(bool condition, bool requirement)
    {
        return !condition || requirement;
    }
}
```

The first validation has an implication, but the implication is already implemented in the base class to improve readability. 3 Simple properties are left in this class:

```
public class OnlyAdultsCanConsumeAlcoholValidation : ValidationBase<Person>
{
    private const int MinimumAge = 18;

    public OnlyAdultsCanConsumeAlcoholValidation(Person context)
        : base(context)
    {
    }

    public override bool Requirement
    {
        get
        {
            return Context.Age >= 18;

        }
    }

    public override bool Condition
    {
        get
        {
            return Context.ConsumesAlcohol;
        }
    }

    public override string Message
    {
        get
        {
            return string.Format(
                "{0} is not allowed to consume alcohol because his or her age ({1}) not is {2} or high
                Context.Name, Context.Age, MinimumAge);
        }
    }
}
```

The other validation does not need to implement an implication so it has only 2 properties:

```
public class Age0OrHigherValidation : ValidationBase<Person>
{
    public Age0OrHigherValidation(Person context)
        : base(context)
    {
    }

    public override bool Requirement
    {
        get {return Context.Age >= 0;  }
    }

    public override string Message
    {
        get
        {
            return string.Format("The Age {1} of {0} is not 0 or higher.",
                Context.Name, Context.Age);
        }
    }
}
```

Tags: Generics | LINQ | SOLID

Submit to DotNetKicks... Permalink | Comments (2)

Related posts

Create your own navigation system (with a Graph, Node and Connection class)
Lots of people uses naviagtion systems like TomTom these days. As a C# deveoper you might want to kn...

Lazy<T> property caching alternative
Lazy Has some disadvantages when you use it for caching properties, If you use other members of a...

Simple IoC container, makes it easier to debug
Most IoC Containers uses reflection for inversion of control. This makes it hard to debug and the co...

Comments (2)

Frank Bakker 1/8/2014 9:30:48 PM #

Hi Alex, (as you know) I like your general aproach in creating these separate validation classes, especially if there are a lot of validations based on the same input. By creating some infrastructure for this you get a great benefit to enforce consistency in how validations are implemented!

One thing I am not sure about is the separation of the condition and the requirement, this seems to be quite arbitrary. I could say
- When you drink(condition) you should be at least 18 (requirement)
But I could also say
- If you are under 18 (condition) you are not allowed to drink (requirement)

Personally I would phrase it the second way, but that is besides the point. My point is that there is no good objective way to choose one over the other, so there is no real distinction to make between two parts of the same validation. Maybe your real world scenario's are better suitable for this, but I would be very carefull not to split all your validations in 2 parts just because you have two methods that can be overridden while some simple boolean logic will also do the trick.

Regards Frank

Reply

Admin 1/10/2014 10:09:38 AM #

Hi Frank,

The validation with a separate condition and requirement forces you to NOT implement the implication in the IsValid property . Maybe it is a bit overenginered now. Thanks for your comment .I will consider changing my production code.

Best regards,

Alex

Reply

Add comment

Name*	
E-mail*	
Country	Brazil ▾
5+5 =	

Comment Preview b i u quote

☐ Notify me when new comments are added

Save comment