The Specification Pattern: A Primer

Posted March 25th 2005 by Matt Berther

The Specification pattern is a very powerful design pattern which can be used to remove a lot of cruft from a class's interface while decreasing coupling and increasing extensibility. It's primary use is to select a subset of objects based on some criteria, and to refresh the selection at various times.

For example, I've seen a lot of classes that have interfaces that look something similar to this:

```
public class User
{
    public string Company;
    public string Name;
    public string City;
}

public class UserProvider
{
    public User[] GetUserByName(string name)
    {
    }

    public User[] GetUsersByCity(string name)
    {
    }

    public User[] GetUsersByCompany(string company)
    {
    }
}
```

Using this model, you can see that every time you want to add a new condition for user retrieval, you have to add a method to the UserProvider class which obfuscates the interface.

Now, lets look at the same example using the specification pattern.

```
public class User
{
    public string Company;
    public string Name;
    public string City;
}

public class UserSpecification
{
    public virtual bool IsSatisfiedBy(User user)
    {
        return true;
    }
}

public class UserProvider
{
    public User[] GetBySpecification(UserSpecification spec)
    {
        ArrayList list = new ArrayList();

        UserCollection coll = SomeMethodToPopulateTheUserCollection();
        foreach (User user in coll)
        {
            if (spec.IsSatisfiedBy(user))
            {
                list.Add(user);
            }
        }
```

```
        return (User[])list.ToArray(typeof(User));
    }
}

class UserCompanySpecification : UserSpecification
{
    private readonly string companyName;

    public UserCompanySpecification(string companyName)
    {
        this.companyName = companyName;
    }

    public override bool IsSatisfiedBy(User user)
    {
        return user.Company.Equals(companyName);
    }
}
```

Using the specification pattern, we have removed all of the specialized methods from the UserProvider class. Also, because of the loose coupling, any time we want an additional condition for user retrieval, we need to only implement a new UserSpecification and pass this instance off to the GetBySpecification method, rather than polluting the existing interface.

This allows the calling code to determine exactly how it wants to filter any given collection, rather than the provider code assuming that it knows how the user wants it.

Of course, there is nothing preventing an API designer from putting a few commonly used specifications into the API itself.

This pattern is very powerful, but like anything can be overused. Make sure to review the consequences in the linked description of the pattern for when you should and shouldnt use this pattern.

The Specification Pattern

Hot on the heels of my devastatingly fantastic post on an implementation of the Snapshot Pattern, I give you my next *piece du resistance*. In this little post, I'd like to delve into the Specification

So what the heck is it? Matt Berther provided a pretty good introduction where he states:

> It's primary use is to select a subset of objects based on some criteria...

That pretty much sums it up. What we want to do is extract out a *specification* for a subset of objects we might be interested in. We do this by creating specification objects.

You'll see something like this in a lot of applications:

```
view plain  copy to clipboard  print  ?
01. public List GetHighPricedSaleProducts(){
02.   List list = new ArrayList();
03.   for(Product p in Products){
04.     if(p.IsOnSale && p.Price > 100.0){
05.       list.Add(p);
06.     }
07.   }
08.   return list;
09. }
```

This is fine in small doses, but your definition of a highly priced sale product might change over time, and we want to avoid having our logic for what *IsOnSale* and what a highly priced object act sprinkled throughout our code. One way to avoid this is to extract our logic into a Specification object like so:

```
view plain  copy to clipboard  print  ?
01. public class ProductOnSaleSpecification : Specification<Product>
02. {
03.   public override bool IsSatisfiedBy(Product product)
04.   {
05.     return product.IsOnSale;
06.   }
07. }
```

Now the first loop I wrote can be written like so:

```
view plain  copy to clipboard  print  ?
01. public List GetHighPricedSaleProducts(){
02.   List list = new ArrayList();
03.   for(Product p in Products){
04.     if(new ProductOnSaleSpecification().isSatisfiedBy(p)
05.       && p.Price > 100.0){
06.       list.Add(p);
07.     }
08.   }
09.   return list;
10. }
```

This is a slight improvement... There's actually more code to write, but now we can separately unit test each specification we create, without worrying about the loop:

```
view plain  copy to clipboard  print  ?
01. [Test]
02. public void TestIsSatisfiedBy_ProductOnSaleSpecification()
03. {
04.   bool isOnSale = true;
05.   Product saleProduct = new Product(isOnSale);
06.   Product notOnSaleProduct = new Product(!isOnSale);
07.   ProductOnSaleSpecification spec =
08.     new ProductOnSaleSpecification();
09.   Assert.IsTrue(spec.IsSatisfiedBy(saleProduct));
10.   Assert.IsFalse(spec.IsSatisfiedBy(notOnSaleProduct));
11. }
```

Ok, so that's interesting, but we haven't even gone halfway, here. Why don't we refine that loop I wrote to use the new Generic collections in .NET 2.0:

```
view plain  copy to clipboard  print  ?
01. public List<Product> GetSaleProducts(){
02.   return Products.FindAll(
03.     new ProductOnSaleSpecification().IsSatisfiedBy);
04. }
```

Wow, now there's some serious savings on lines of code. "But you're missing the bit about the high priced products from the first example!?!" I hear you saying. Fear not, let's extract that into a specification like so:

```
view plain  copy to clipboard  print  ?
01. public class ProductPriceGreaterThanSpecification : Specification<Product>
02. {
03.     private readonly double _price;
04.     public ProductPriceGreaterThanSpecification(double price)
05.     {
06.         _price = price;
07.     }
08.     public override bool IsSatisfiedBy(Product product)
09.     {
10.         return product.Price > _price;
11.     }
12. }
```

We're still left with one problem, though. How do we tell the generic list of all products that we want the products that are *both* on sale and over a certain price? Let's try extracting our functionali Specification superclass first. This is what our *ProductOnSaleSpecification* and *ProductPriceGreaterThanSpecification* will inherit from. Once that's over with, we can create a *CompositeSpecifi*

abstract, and allows us to pass in the left and right sides of a specification "equation." We can then implement yet another subclass (this time of CompositeSpecification) that we'll call *AndSpec* it is:

```
view plain  copy to clipboard  print  ?
01. public class AndSpecification<T> : CompositeSpecification<T>
02. {
03.   public AndSpecification(Specification<T> leftSide,
04.     Specification<T> rightSide)
05.       : base(leftSide, rightSide)
06.   {}
07.   public override bool IsSatisfiedBy(T obj)
08.   {
09.     return _leftSide.IsSatisfiedBy(obj)
10.       && _rightSide.IsSatisfiedBy(obj);
11.   }
12. }
```

Now our original loop that looks for highly priced products that are on sale looks like this:

```
view plain  copy to clipboard  print  ?
01. public List<Product> GetSaleProducts(){
02.   AndSpecification spec = new AndSpecification(
03.     new ProductOnSaleSpecification(),
04.     new ProductPriceGreaterThanSpecification());
05.   return Products.FindAll(spec.IsSatisfiedBy);
06. }
```

We're getting there, but we're *still* not done. The code we just wrote is soooo .NET 1.1. Let's get fluent with our interfaces and add some sweet sugar to our Specification base class...

```
view plain  copy to clipboard  print  ?
01. public abstract class Specification<T>
02. {
03.   public abstract bool IsSatisfiedBy(T obj);
04.   public AndSpecification<T> And(Specification<T> specification)
05.   {
06.     return new AndSpecification<T>(this, specification);
07.   }
08.   public OrSpecification<T> Or(Specification<T> specification)
09.   {
10.     return new OrSpecification<T>(this, specification);
11.   }
12.   public NotSpecification<T> Not(Specification<T> specification)
13.   {
14.     return new NotSpecification<T>(this, specification);
15.   }
16. }
```

I've just added some convenience methods to Specification that will let us chain together any specifications we create. Therefore, our original loop ascends to a new level of sexiness...

```
view plain  copy to clipboard  print  ?
01. public List<Product> GetSaleProducts(){
02.   return Products.FindAll(
03.     new ProductOnSaleSpecification().And(
04.       new ProductPriceGreaterThanSpecification(100)).IsSatisfiedBy);
05. }
```

Friday, July 22, 2011

Specifications Pattern with LINQ

Recently I was reading two books Domain Driven Design and Refactoring to Patterns. Both these books have references to and also examples related to a pattern called Specification Pattern. This pattern also finds its way in the Patterns of Enterprise Application and Architecture by Martin Fowler. Here is my attempt to adapt this pattern using the new features available in DotNet like LINQ, Lambda Expression, Generics etc.

Code without Specification Pattern in place

To keep things simple, I'll try to reuse the example in the book **Refactoring to Patterns**. This example talks about a product finder functionality using a **ProductRepository**. There are a set of products and we need to apply filter based on different criteria. It can be a single criteria like the color, price or size of the product. The product can also be searched using composite like color and price, size and price or also products which do not meet certain criteria like color. Here is a product repository having various methods to filter the products.

```
public class ProductRepositoryWithoutSpecification
    {
        private IList<Product> _products = new List<Product>();

        public void Add(Product product)
        {
            _products.Add(product);
        }

        public IList<Product> FindProductsByColor(ProductColor color)
        {
            return _products.Where(product => product.Color == color).ToList();
        }

        public IList<Product> FindProductsByPrice(double price)
        {
            return _products.Where(product => product.Price == price).ToList();
        }

        public IList<Product> FindProductsBelowPrice(double price)
        {
            return _products.Where(product => product.Price < price).ToList();
        }

        public IList<Product> FindProductsAbovePrice(double price)
        {
            return _products.Where(product => product.Price > price).ToList();
        }

        public IList<Product> FindProductsByColorAndBelowPrice(ProductColor color, double price)
        {
            return _products.Where(product => product.Color == color && product.Price < price).ToList();
        }

        public IList<Product> FindProductsByColorAndSize(ProductColor color, ProductSize size)
        {
            return _products.Where(product => product.Color == color && product.Size == size).ToList();
        }

        public IList<Product> FindProductsByColorOrSize(ProductColor color, ProductSize size)
        {
            return _products.Where(product => product.Color == color || product.Size == size).ToList();
        }

        public IList<Product> FindProductsBySizeNotEqualTo(ProductSize size)
        {
            return _products.Where(product => product.Size != size).ToList();
        }
    }
```

As can be seen from the above code, there are 8 different versions of Find methods. This is only a subset and you can extend this to different permutations and combinations of various attributes of the product which include Color, Size, Price. The domain model contains the Product class which has very minimul properties. But you can imagine a real life product which can have numerous attributes like IsInStock, IsOnPromotion etc. etc. The attributes I have defined are sufficient for time being to demonstrate the refactoring towards the Specification Pattern. Before we start with the refactoring,

I want to build a suite of test cases to verify that the filtering works as expected before and after the code has been refactored. Here is the list of test cases I have built for the above 8 filters.

```
private Product _fireTruck;

        private Product _barbieClassic;

        private Product _frisbee;

        private Product _baseball;

        private Product _toyConvertible;

        private ProductRepositoryWithoutSpecification _productRepositoryWithoutSpecification;

        [TestInitialize]
        public void Setup()
        {
            _fireTruck = new Product
            {
                Id = "f1234",
                Description = "Fire Truck",
                Color = ProductColor.RED,
                Price = 8.95,
                Size = ProductSize.MEDIUM
            };

            _barbieClassic = new Product
            {
                Id = "b7654",
                Description = "Barbie Classic",
                Color = ProductColor.YELLOW,
                Price = 15.95,
                Size = ProductSize.SMALL
            };

            _frisbee = new Product
            {
                Id = "f4321",
                Description = "Frisbee",
                Color = ProductColor.GREEN,
                Price = 9.99,
                Size = ProductSize.LARGE
            };

            _baseball = new Product
            {
                Id = "b2343",
                Description = "Baseball",
                Color = ProductColor.WHITE,
                Price = 8.95,
                Size = ProductSize.NOT_APPLICABLE
            };

            _toyConvertible = new Product
            {
                Id = "p1112",
                Description = "Toy Porsche Convertible",
                Color = ProductColor.RED,
                Price = 230,
                Size = ProductSize.NOT_APPLICABLE
            };

            _productRepositoryWithoutSpecification = new ProductRepositoryWithoutSpecification();
            _productRepositoryWithoutSpecification.Add(_fireTruck);
            _productRepositoryWithoutSpecification.Add(_barbieClassic);
            _productRepositoryWithoutSpecification.Add(_frisbee);
            _productRepositoryWithoutSpecification.Add(_baseball);
            _productRepositoryWithoutSpecification.Add(_toyConvertible);
        }

        [TestMethod]
```

```
        public void ProductRepositoryConstructorTest()
        {
            Assert.IsNotNull(_productRepositoryWithoutSpecification);
        }
```

I have added 5 different products to the repository. Each product has a different combination of attributes to ensure that we cover different scenarios covered by the filters. Each of the repository method is tested using the following tests

```
[TestMethod]
        public void FindProductsByColorTest()
        {
            IList<Product> filteredProducts =
                _productRepositoryWithoutSpecification.FindProductsByColor(ProductColor.RED);

            Assert.AreEqual(2, filteredProducts.Count);
            Assert.AreEqual("Fire Truck", filteredProducts.First().Description);
            Assert.AreEqual("Toy Porsche Convertible", filteredProducts.Last().Description);
        }

        [TestMethod]
        public void FindProductsByPriceTest()
        {
            IList<Product> filteredProducts =
                _productRepositoryWithoutSpecification.FindProductsByPrice(9.99);

            Assert.AreEqual(1, filteredProducts.Count);
            Assert.AreEqual("Frisbee", filteredProducts.First().Description);
            Assert.AreEqual(9.99, filteredProducts.First().Price);
        }

        [TestMethod]
        public void FindProductsBelowPriceTest()
        {
            IList<Product> filteredProducts =
                _productRepositoryWithoutSpecification.FindProductsBelowPrice(9);

            Assert.AreEqual(2, filteredProducts.Count);
            Assert.AreEqual("Fire Truck", filteredProducts.First().Description);
            Assert.AreEqual("Baseball", filteredProducts.Last().Description);
        }

        [TestMethod]
        public void FindProductsAbovePriceTest()
        {
            IList<Product> filteredProducts =
                _productRepositoryWithoutSpecification.FindProductsAbovePrice(9);

            Assert.AreEqual(3, filteredProducts.Count);
            Assert.AreEqual("Barbie Classic", filteredProducts.First().Description);
            Assert.AreEqual("Toy Porsche Convertible", filteredProducts.Last().Description);
        }

        [TestMethod]
        public void FindProductsByColorAndBelowPriceTest()
        {
            IList<Product> filteredProducts =
                _productRepositoryWithoutSpecification.FindProductsByColorAndBelowPrice(ProductColor.GREEN, 10);

            Assert.AreEqual(1, filteredProducts.Count);
            Assert.AreEqual("Frisbee", filteredProducts.First().Description);
        }

        [TestMethod]
        public void FindProductsByColorAndSizeTest()
        {
            IList<Product> filteredProducts =
                _productRepositoryWithoutSpecification.FindProductsByColorAndSize(ProductColor.GREEN, ProductSize.SMALL);

            Assert.AreEqual(0, filteredProducts.Count);
        }

        [TestMethod]
        public void FindProductsByColorOrSizeTest()
```

```
        {
            IList<Product> filteredProducts =
                _productRepositoryWithoutSpecification.FindProductsByColorOrSize(ProductColor.GREEN, ProductSize.SMALL);

            Assert.AreEqual(2, filteredProducts.Count);
            Assert.AreEqual("Barbie Classic", filteredProducts.First().Description);
            Assert.AreEqual("Frisbee", filteredProducts.Last().Description);
        }

        [TestMethod]
        public void FindProductsBySizeNotEqualToTest()
        {
            IList<Product> filteredProducts =
                _productRepositoryWithoutSpecification.FindProductsBySizeNotEqualTo(ProductSize.SMALL);

            Assert.AreEqual(4, filteredProducts.Count);
            Assert.AreEqual("Fire Truck", filteredProducts.First().Description);
            Assert.AreEqual("Toy Porsche Convertible", filteredProducts.Last().Description);
        }
```

In fact the repository is built using TDD. With all the tests passing we now have the working code. If you look carefully at the code, it is concise with precisely one line in each of the method. Each method filters the products from the collection based on single or multiple attributes. Right now we don't have filters which have more than two criteria. In real life applications you'll come across situation quite often where there are multiple filters. As the number of filters start growing the code starts to smell. Imagine a situation where the products are to be filtered based on following criteria

- Color = Green
- Price > 10
- Size = Small or Large

With every attribute added to product, the filters can grow exponentially. Very soon you might find it very difficult to maintain such code. This is where the Specifications Pattern comes to the rescue.

Specifications Pattern

A very simple definition of Specification Pattern is that it filters a subset of objects based on some criteria. The criteria is nothing but the Specification. The Specification follows the **Single Responsibility Principle**. Each specification in general filters the objects based on only one condition. Lets take an example. We'll take the first test and convert that into a specification. So our first test was to filter the products based on the Color. We can build a **ColorSpecification** which takes color to be filtered as a constructor argument.

```
[TestMethod]
        public void FindProductsByColorTest()
        {
            IList<Product> filteredProducts =
                _productRepositoryWithSpecification.FindProducts(new ColorSpecification(ProductColor.RED));

            Assert.AreEqual(2, filteredProducts.Count);
            Assert.AreEqual("Fire Truck", filteredProducts.First().Description);
            Assert.AreEqual("Toy Porsche Convertible", filteredProducts.Last().Description);
        }
```

We are using the FindProducts method which returns the filtered products based on the specification. Lets look at the ColorSpecification

```
public class ColorSpecification : Specification
    {
        private readonly ProductColor _productColor;

        public ColorSpecification(ProductColor productColor)
        {
            _productColor = productColor;
        }

        public override bool IsSatisfiedBy(Product product)
        {
            return product.Color.Equals(_productColor);
        }
    }
```

The ColorSpecification has a private variable for storing the color which is initialized through the constructor. There is only one method IsSatisfiedBy which returns boolean value based on whether the product color matches with the private variables value. The base Specification is an abstract class which has only one abstract method IsSatisfiedBy. We can run the test and verify that we get the same result as before.

Similar to the ColorSPecification are other specifications which depend on a single attribute like PriceSpecification, SizeSpecification, AbovePriceSpecification and BelowPriceSpecification. So we refactor the tests related to these attributes to use these specifications as shown below

```
        [TestMethod]
        public void FindProductsByPriceTest()
```

```
    {
        IList<Product> filteredProducts =
            _productRepositoryWithSpecification.FindProducts(new PriceSpecification(9.99));

        Assert.AreEqual(1, filteredProducts.Count);
        Assert.AreEqual("Frisbee", filteredProducts.First().Description);
        Assert.AreEqual(9.99, filteredProducts.First().Price);
    }

    [TestMethod]
    public void FindProductsBelowPriceTest()
    {
        IList<Product> filteredProducts =
            _productRepositoryWithSpecification.FindProducts(new BelowPriceSpecification(9));

        Assert.AreEqual(2, filteredProducts.Count);
        Assert.AreEqual("Fire Truck", filteredProducts.First().Description);
        Assert.AreEqual("Baseball", filteredProducts.Last().Description);
    }

    [TestMethod]
    public void FindProductsAbovePriceTest()
    {
        IList<Product> filteredProducts =
                    _productRepositoryWithSpecification.FindProducts(new AbovePriceSpecification(9));

        Assert.AreEqual(3, filteredProducts.Count);
        Assert.AreEqual("Barbie Classic", filteredProducts.First().Description);
        Assert.AreEqual("Toy Porsche Convertible", filteredProducts.Last().Description);
    }
```

Composite Specifications

So far so good. Lets now look at the next test which uses composite criteria. The test tries to filter the products by Color as well as below certain price. We need a combination of **ColorSpecification** and **BelowPriceSpecification**. We could apply the ColorSpecification first and then filter the records by applying the **BelowPriceSpecification**. But that doesn't look very good.

Instead we could use the Composite design pattern to compose a specification containing multiple Specifications. Composite pattern suggests that both the composite and the single specification should have the same interface. So in our composite specification we need to have the same IsSatisfiedBy method. Lets create a **AndSpecification** which can be used to filter products using any two Specification.

```
public class AndSpecification : Specification
{
    private readonly Specification _leftSpecification;

    private readonly Specification _rightSpecification;

    public AndSpecification(Specification leftSpecification, Specification rightSpecification)
    {
        _leftSpecification = leftSpecification;
        _rightSpecification = rightSpecification;
    }

    public override bool IsSatisfiedBy(Product product)
    {
        return _leftSpecification.IsSatisfiedBy(product) && _rightSpecification.IsSatisfiedBy(product);
    }
}
```

The constructor of AndSpecification takes two Specification instances and gives the result of the logical And operation between the two of them. The **OrSpecification** is exactly similar and returns the logical Or result of the two Specifications.

The interesting case is of the **NotSpecification** which is used for negating the result of a Specification. Instead of two Specifications we need only one Specification to negate its result. So the NotSpecification looks like

```
public class NotSpecification : Specification
{
    private readonly Specification _specification;

    public NotSpecification(Specification specification)
    {
        _specification = specification;
    }
```

```
        public override bool IsSatisfiedBy(Product product)
        {
            return !_specification.IsSatisfiedBy(product);
        }
    }
```

In fact the NotSpecification is not really a composite specification as it depends on a single specification. Because it involves a logical operation it is treated bit differently. The final result of building all these single and composite specifications is that the repository method becomes very simple. We no longer need different methods for performing filtering, one method does it for all cases.

```
    public class ProductRepositoryWithSpecification
    {
        private IList<Product> _products = new List<Product>();

        public void Add(Product product)
        {
            _products.Add(product);
        }

        public IList<Product> FindProducts(Specification specification)
        {
            return _products.Where(specification.IsSatisfiedBy).ToList();
        }
    }
```

The 8 methods earlier are no longer required. We have a single FindProducts which works with Specification instance.

Conclusion

As we saw in this blog, complex filtering can be simplified using Specifications Pattern. It can be used for a single value filter as well as composite filtering. We can add new attributes to the mode (Product in this case) and without modifying the FindProducts method of the repository, we can build additional filters using new Specifications. This keeps the design simple and follows the Open Closed Principle where in the repository is closed for modification but open for extension.

As always the complete working solution is available for download.

Until next time Happy Programming 

Posted by nilesh at 10:42 PM  +2 Recommend this on Google

Labels: AndSpecification, NotSpecification, OrSpecification, Specification, Specifications Pattern