# Open Closed Principle by Joe Ocampo

The open closed principle is one of the oldest principles of Object Oriented Design. I won't bore you with the history since you can find countless articles out on the net. But if you want a really comprehensive read, please checkout Robert Martin's excellent write up on the subject.

The open closed principle can be summoned up in the following statement.

The open/closed principle states "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification";[1] that is, such an entity can allow its behavior to be modified without altering its source code.

Sounds easy enough but many developers seem to miss the mark on actually implementing this simple extensible approach. I don't think it is a matter of skill set, as much as I feel that they have never been taught how to approach applying OCP to class design.

### A case study in OCP ignorance

*Scenario: We need a way to filter products based off the color of the product.*

All entities in a software development ecosystem behave a certain way that is dependent upon a governed context. In the scenario above, you realize that you are going to need a Filter class that accepts a color and then filters all the products that adhere to that color.

The filter class' responsibility is to filter products (its job), based off the action of filtering by color (its behavior). So your goal is to write a class that will always be able to filter products. (Work with me on this I am trying to get you into a mindset because that is all OCP truly is at its heart.)

To make this easier I like to tell developers to fill in the following template.

*The {class} is responsible for {its job} by {action/behavior}*

The ***ProductFilter*** is responsible for ***filtering products*** by ***color***

Now, let's write our simple class to do this:

```
public class ProductFilter
{
  public IEnumerable<Product> ByColor(IList<Product> products, ProductColor productColor)
  {
    foreach (var product in products)
    {
      if (product.Color == productColor)
        yield return product;
    }
  }
}
```

As you can see this pretty much does the job of filtering a product by color. Pretty simple, but imagine if you had the following typical conversation with one of your users.

User: "We need to also be able to filter by size."

Developer: "Just size alone or color and size? "

User: "Umm probably both."

Developer: "Great!"

So let's use our OCP scenario template again.

The ***ProductFilter*** is responsible for ***filtering products*** by ***color***

The ***ProductFilter*** is responsible for ***filtering products*** by ***size***

The ***ProductFilter*** is responsible for ***filtering products*** by ***color and size***

Now the code:

```
public class ProductFilter
{
  public IEnumerable<Product> ByColor(IList<Product> products, ProductColor productColor)
  {
    foreach (var product in products)
    {
      if (product.Color == productColor)
        yield return product;
    }
  }

  public IEnumerable<Product> ByColorAndSize(IList<Product> products,
```

```
                                ProductColor productColor,
                                ProductSize productSize)
  {
    foreach (var product in products)
    {
      if ((product.Color == productColor) &&
        (product.Size == productSize))
        yield return product;
    }
  }

  public IEnumerable<Product> BySize(IList<Product> products,
                              ProductSize productSize)
  {
    foreach (var product in products)
    {
      if ((product.Size == productSize))
        yield return product;
    }
  }
}
```

This is great but this implementation is violating OCP.

### Where'd we go wrong?

Let's revisit again what Robert Martin has to say about OCP.

Robert Martin says modules that adhere to Open-Closed Principle have 2 primary attributes:

1. "Open For Extension" - It is possible to extend the behavior of the module as the requirements of the application change (i.e. change the behavior of the module).

2. "Closed For Modification" - Extending the behavior of the module does not result in the changing of the source code or binary code of the module itself.

Let's ask the following question to ensure we **ARE** violating OCP.

Every time a user asks for new criteria to filter a product, do we have to modify the ProductFilter class?
***Yes! This means it is not CLOSED for modification.***

Every time a user asks for new criteria to filter a product, can we extend the behavior of the ProductFilter class to support the new criteria, without opening up the class file again and modifying it?
***No! This means it is not OPEN for extension.***

### Solutions

One of the easiest ways to implement OCP is utilize a template or strategy pattern. If we still allow the Product filter to perform its job of invoking the filtering process, we can put the implementation of filtering in another class. This is achieved by mixing in a little LSP.

Here is the template for the ProductFilterSpecification:

```
public abstract class ProductFilterSpecification
{
   public IEnumerable<Product> Filter(IList<Product> products)
   {
      return ApplyFilter(products);
   }

   protected abstract IEnumerable<Product> ApplyFilter(IList<Product> products);
}
```

Let's go ahead and create our first criteria, which is a color specification.

```
public class ColorFilterSpecification : ProductFilterSpecification
{
   private readonly ProductColor productColor;

   public ColorFilterSpecification(ProductColor productColor)
   {
      this.productColor = productColor;
   }

   protected override IEnumerable<Product> ApplyFilter(IList<Product> products)
   {
      foreach (var product in products)
      {
         if (product.Color == productColor)
            yield return product;
      }
   }
}
```

Now all we have to do is extend the actual ProductFilter class to accept our template ProductFilterSpecification.

```
public IEnumerable<Product> By(IList<Product> products, ProductFilterSpecification filterSpecification)
{
   return filterSpecification.Filter(products);
}
```

OCP goodness!

So let's make sure we are **NOT** violating OCP and ask the same questions we did before.

Every time a user asks for new criteria to filter a product do we have to modify the ProductFilter class?
**No! Because we have marshaled the behavior of filtering to the ProductFilterSpecification**.
"Closed for modification"

Every time a user asks for new criteria to filter a product can we extend the behavior of the ProductFilter class to support the new criteria, without opening up the class file again and modifying it?
**Yes! All we simply have to do is, pass in a new ProductFilterSpecification. "Open for extension"**

Now let's just make sure we haven't modified too much from our intentions of the ProductFilter. All we simply have to do is validate that our ProductFilter still has the same behavior as before.

The ***ProductFilter*** is responsible for ***filtering products*** by ***color: Yes it still does that!***

The ***ProductFilter*** is responsible for ***filtering products*** by ***size: Yes it still does that!***

The ***ProductFilter*** is responsible for ***filtering products*** by ***color and size: Yes it still does that!***

If you are a good TDD/BDD practitioner you should already have all these scenarios covered in your Test Suite.

Here is the final code:

```
namespace OCP_Example.Good
{
   public class ProductFilter
   {
      [Obsolete("This method is obsolete; use method 'By' with ProductFilterSpecification")]
      public IEnumerable<Product> ByColor(IList<Product> products, ProductColor productColor)
      {
         foreach (var product in products)
         {
            if (product.Color == productColor)
               yield return product;
         }
      }

      [Obsolete("This method is obsolete; use method 'By' with ProductFilterSpecification")]
      public IEnumerable<Product> ByColorAndSize(IList<Product> products,
                                    ProductColor productColor,
                                    ProductSize productSize)
```

```
    {
      foreach (var product in products)
      {
        if ((product.Color == productColor) &&
          (product.Size == productSize))
          yield return product;
      }
    }

    [Obsolete("This method is obsolete; use method 'By' with ProductFilterSpecification")]
    public IEnumerable<Product> BySize(IList<Product> products,
                                       ProductSize productSize)
    {
      foreach (var product in products)
      {
        if ((product.Size == productSize))
          yield return product;
      }
    }

    public IEnumerable<Product> By(IList<Product> products, ProductFilterSpecification filterSpecification)
    {
      return filterSpecification.Filter(products);
    }
}

public abstract class ProductFilterSpecification
{
    public IEnumerable<Product> Filter(IList<Product> products)
    {
      return ApplyFilter(products);
    }

    protected abstract IEnumerable<Product> ApplyFilter(IList<Product> products);
}

public class ColorFilterSpecification : ProductFilterSpecification
{
    private readonly ProductColor productColor;

    public ColorFilterSpecification(ProductColor productColor)
    {
      this.productColor = productColor;
    }

    protected override IEnumerable<Product> ApplyFilter(IList<Product> products)
    {
      foreach (var product in products)
      {
        if (product.Color == productColor)
          yield return product;
      }
```

```
    }
}

public enum ProductColor
{
    Blue,
    Yellow,
    Red,
    Gold,
    Brown
}

public enum ProductSize
{
    Small, Medium, Large, ReallyBig
}

public class Product
{
    public Product(ProductColor color)
    {
        this.Color = color;
    }

    public ProductColor Color { get; set; }

    public ProductSize Size { get; set; }
}

[Context]
public class Filtering_by_color
{
    private ProductFilter filterProduct;
    private IList<Product> products;

    [SetUp]
    public void before_each_spec()
    {
        filterProduct = new ProductFilter();
        products = BuildProducts();
    }

    private IList<Product> BuildProducts()
    {
        return new List<Product>
                    {
                        new Product(ProductColor.Blue),
                        new Product(ProductColor.Yellow),
                        new Product(ProductColor.Yellow),
                        new Product(ProductColor.Red),
                        new Product(ProductColor.Blue)
                    };
```

```
    }


    [Specification]
    public void should_filter_by_the_color_given()
    {
      int foundCount = 0;
      foreach (var product in filterProduct.By(products, new ColorFilterSpecification(ProductColor.Blue)))
      {
        foundCount++;
      }

      Assert.That(foundCount, Is.EqualTo(2));
    }
  }
}
```