Tip: Static methods are a code smell

by Benjamin Day (benday@benday.com)

Ever heard of code smells? A code "smell" is a way of saying that there is a potentially a problem with your code. It doesn't mean that you've definitely made a mistake or that you're definitely wrong for doing something wrong, it just means that you *might* be doing something wrong. At the moment, Wikipedia says that it is "any symptom in the source code of a program that possibly indicates a deeper problem."

One of the common smells that I come across is static method and variables. (In VB.NET these are known as "Shared" methods.) If you mark a method as static, this means that the method now belongs to the type rather than an instance.

Let's take a simple example of a class called Person and a class called PersonApp.

```
public class Person
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public static Person CreateNew()
    {
        Person temp = new Person();

        temp.Id = -1;
        temp.FirstName = String.Empty;
        temp.LastName = String.Empty;

        return temp;
    }

    public override string ToString()
    {
        return String.Format("{0} {1}", FirstName, LastName);
    }
}

public class PersonApp
{
    public void DoSomething()
    {
        // static method
        Person temp = Person.CreateNew();

        temp.FirstName = "Ben";
        temp.LastName = "Day";

        // instance method
        Console.WriteLine(temp.ToString());
    }
}
```

On the Person class, we've encapsulated some Factory Pattern creation logic inside of a static method called CreateNew() and there is also an instance method implementation of ToString(). At this point, the static method seems harmless but let's assume that, like most software, the application grows and changes over time. What's the harm in that static method? The answer: maintenance and testability.

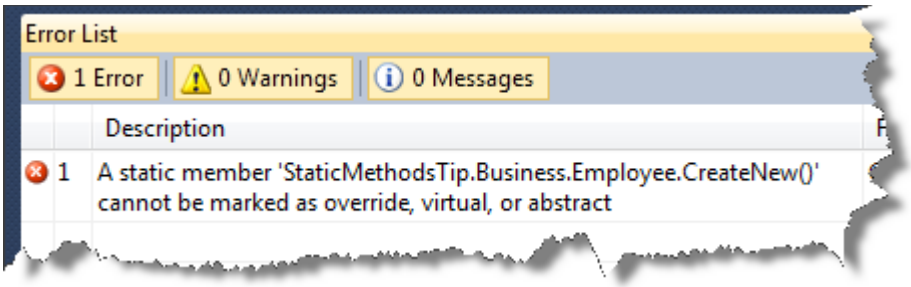
Why are static methods bad for maintenance? Well, static methods cannot be overridden or extended through inheritance. Pretend that we've been handed a new requirement and now we need a class called Employee that extends from Person and introduces two new properties: Boss and Department. As part of the requirements for the Employee class, the ToString() and CreateNew() methods need to be updated.

```
public class Employee : Person
{
    public string Boss { get; set; }
    public string Department { get; set; }

    public override string ToString()
    {
        return String.Format("{0} {1} -- {2}", FirstName, LastName, Department);
    }
}
```

Providing this new implementation of ToString() on Employee is easy. All you have to do is override ToString() again and it replaces the implementation that's on Person. So, we get to keep all the properties from Person and add on the additional Employee fields and then change the ToString() implementation. Very easy, very basic object-oriented programming.

The implementation of CreateNew() for Employee isn't going to be as smooth. What we really want to be able to do is to take the implementation of CreateNew() from the base class (Person) and extend it so that it initializes the Boss and Department properties. The problem is that we get a compiler error if we mark a static method with override, virtual, or abstract.



This pretty much leaves us no choice. We have to duplicate the code from Person.CreateNew() in Employee.CreateNew(). That's definitely a worst practice.

```
public class Employee : Person
{
    public string Boss { get; set; }
    public string Department { get; set; }

    public override string ToString()
    {
        return String.Format("{0} {1} -- {2}", FirstName, LastName, Department);
    }

    public static Employee CreateNew()
    {
        Employee temp = new Employee();

        temp.Id = -1;
        temp.FirstName=String.Empty;
        temp.LastName = String.Empty;
        temp.Boss = String.Empty;
        temp.Department = String.Empty;

        return temp;
    }
}
```

The other reason that I try to avoid static methods is because they're bad for testability and impossible to use with interface-driven programming. If you're using and writing unit tests, you'll quickly realize how much simpler and cleaner that your unit tests can be if you code against interfaces. If your application's code depends on interfaces rather than concrete objects, your application is going to be much more flexible. This interface flexibility will allow you to use "mock" or otherwise faked out implementations of your objects in your unit tests. Mock objects help to keep your tests focused on what you're trying to test by freeing you from having to set up and manage huge chains of object dependencies.

If you want to create an IPerson interface to represent a Person, there's no way to get that CreateNew() method on there because you can't have static methods on an interface. That's a big roadblock.

Why you might be tempted to write a lot of static methods

I tend to find developers getting in to the static method trap when they're trying to write a stateless application like a web application or a WCF service. In a stateless application, there's no in-memory state data hanging around between calls from the user. For example, in a WCF service application, a WCF client makes a call to a service method, the service processes the request, returns a result to the client, and is disposed. The lifetime of the object that services that WCF request is very short.

Another example of a short-lived object would be the codebehind object instances for an ASP.NET WebForm. For example, let's say that you have a form called PersonDetail.aspx and that form has a

Save button on it that is hooked up to Click event handler method on PersonDetail.aspx.cs. If the user clicks Save, the request goes up to the server, ASP.NET instantiates an instance of the PersonDetail class, and (eventually) passes the request to the click handler method. The click handler executes and exits and the PersonDetail object instance is released shortly afterward. PersonDetail instances don't last long at all.

These short-lived classes like the ASP.NET codebehind and the WCF service classes usually don't handle all of the work themselves. Ideally, it handles very little of the work and instead relies on a class in your business tier to do the real work of the request.

Here's where the static method temptation starts to leak in to the business tier. If the code that calls the business tier has a short lifetime and your application's state can be easily passed in as arguments to a business tier method, then it would make sense to make the business tier method static, right? I mean, why incur the overhead of instantiating your business object just to make a single method call?

This mental logic quickly becomes a "slippery slope". If the business tier method is short-lived and static then the inevitable calls to the data access tier classes might as well be static, too. Those data access calls just end up making more short-lived calls in to the database or in to stored procedures and they don't have much state anyway. And on and on and on until your whole application becomes a big collection of static methods. Now, guess what? – what you thought was an object-oriented application is now just a bunch of procedural code that's tough to maintain, test, and extend.

Summary

If you find yourself writing a lot of static methods, you're probably doing something wrong. Static methods work best as utility methods (think System.Math) or other very tightly scoped methods. If you can imagine ever needing to use object inheritance or needing to use polymorphism for your method, you should definitely skip the static and make it an instance method.