codemanship

Code Craft Driving Test FxCop Rules

Introduction

In the Codemanship Code Craft “Driving Test”, you will be asked to implement a set of features to pass some customer acceptance tests.

We have created a suite of custom FxCop code rules that your source code will have to pass that check for issues relating to Clean Code.

- Readable
- Free of duplication (except when that makes it less readable)
- Made of simple parts that are:
  - Loosely coupled
  - Swappable
  - Communicating through client-specific interfaces

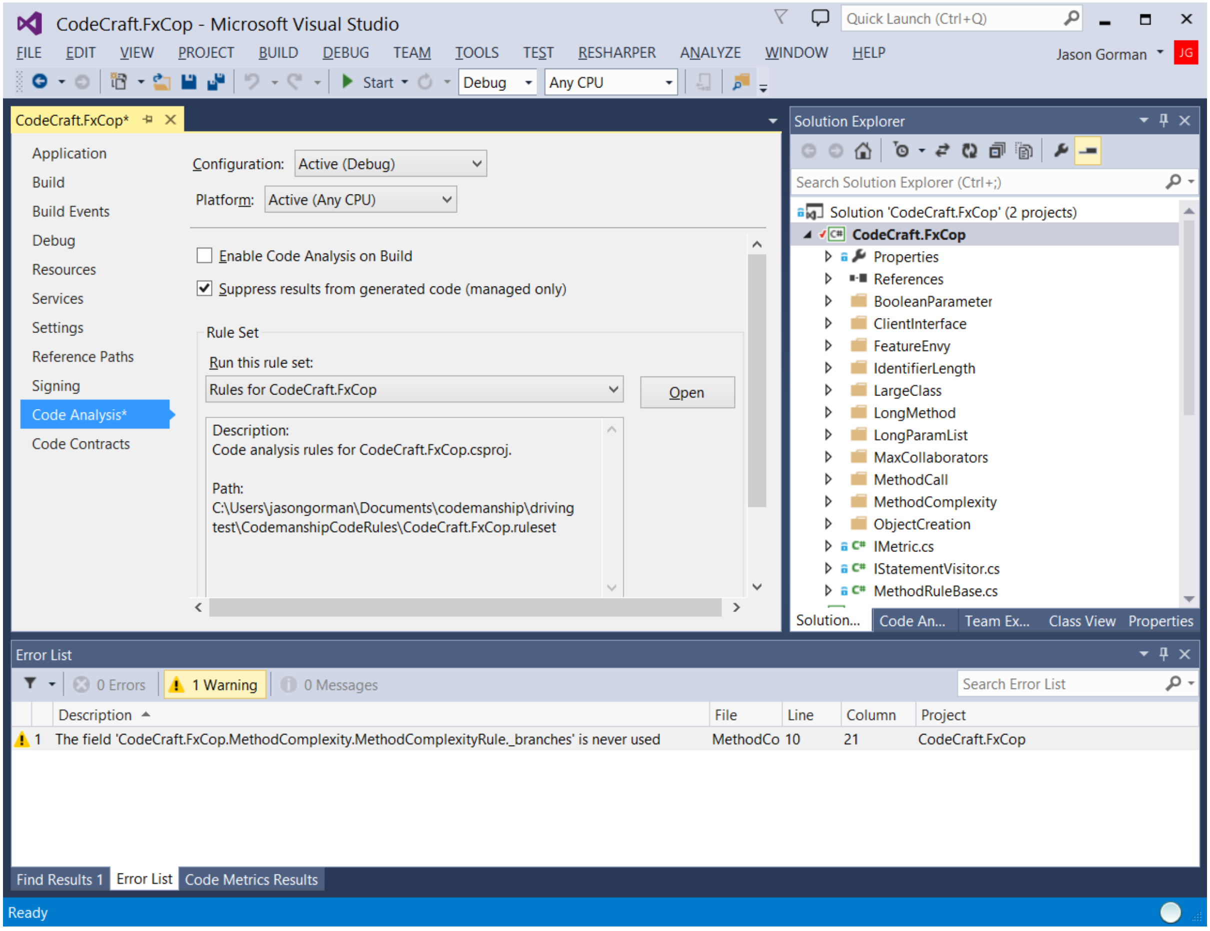
All of these rules will be applied to your source code, but not to your test code.

Installing the CodeCraft.FxCop rules

1. If it’s open, close Visual Studio.
2. Copy **CodeCraft.FxCop.dll** from the appropriate folder for the version of Visual Studio you’ll be using.
3. Find the **Team Tools\Static Analysis Tools\FxCop\Rules** folder for your installation of Visual Studio. (E.g. C:\Program Files (x86)\Microsoft Visual Studio 14.0\Team Tools\Static Analysis Tools\FxCop\Rules) and paste the DLL into that folder.
4. Copy **CodeCraft.FxCop.ruleset** into the **Team Tools\Static Analysis Tools\Rule Sets** folder of your VS installation.
5. Start Visual Studio.

Using the CodeCraft.FxCop rules

Select the Visual Studio project you want to apply the rules to, and open its *Properties* page. Select the *Code Analysis* tab.



Configuration: Active (Debug)

Platform: Active (Any CPU)

☐ Enable Code Analysis on Build

☑ Suppress results from generated code (managed only)

Rule Set

Run this rule set:

Rules for CodeCraft.FxCop

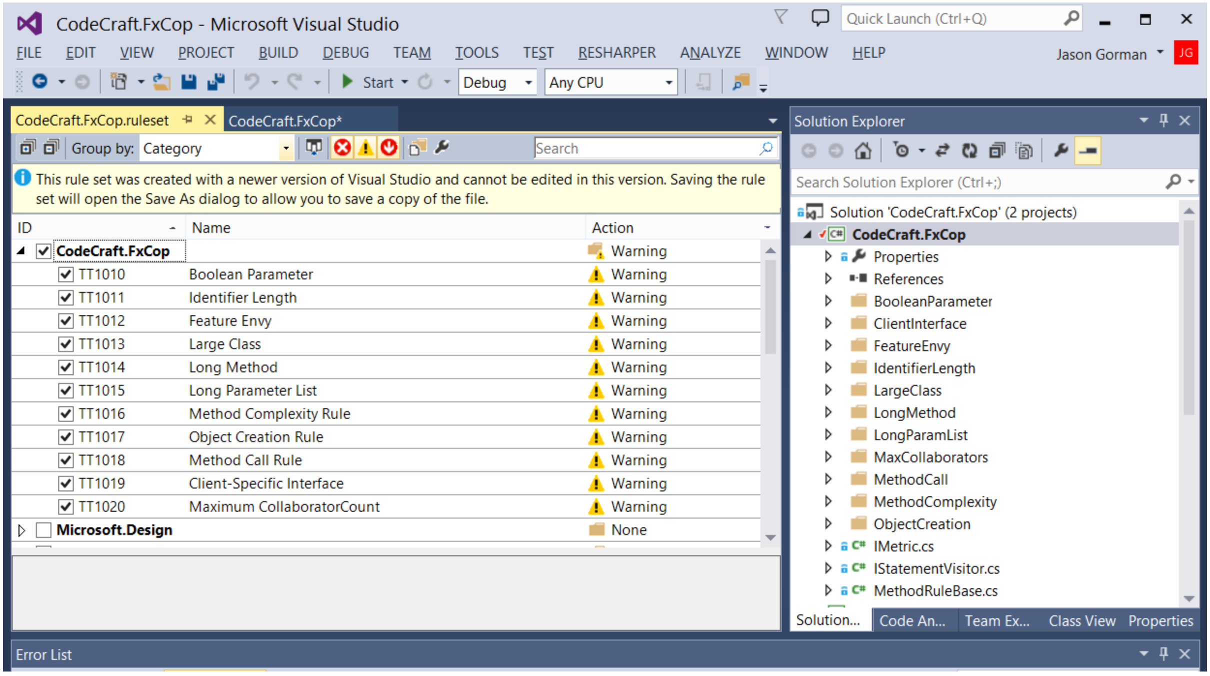
Description:
Code analysis rules for CodeCraft.FxCop.csproj.

Path:
C:\Users\jasongorman\Documents\codemanship\driving test\CodemanshipCodeRules\CodeCraft.FxCop.ruleset

	Description	File	Line	Column	Project
⚠ 1	The field 'CodeCraft.FxCop.MethodComplexity.MethodComplexityRule._branches' is never used	MethodCo	10	21	CodeCraft.FxCop

From the *Run this rule set* drop down, select the **CodeCraft.FxCop** ruleset. (If this ruleset isn't listed, select *Browse…* from the list and find the ruleset's file.)

Once it's selected, click *Open* to view the ruleset's configuration.



This rule set was created with a newer version of Visual Studio and cannot be edited in this version. Saving the rule set will open the Save As dialog to allow you to save a copy of the file.

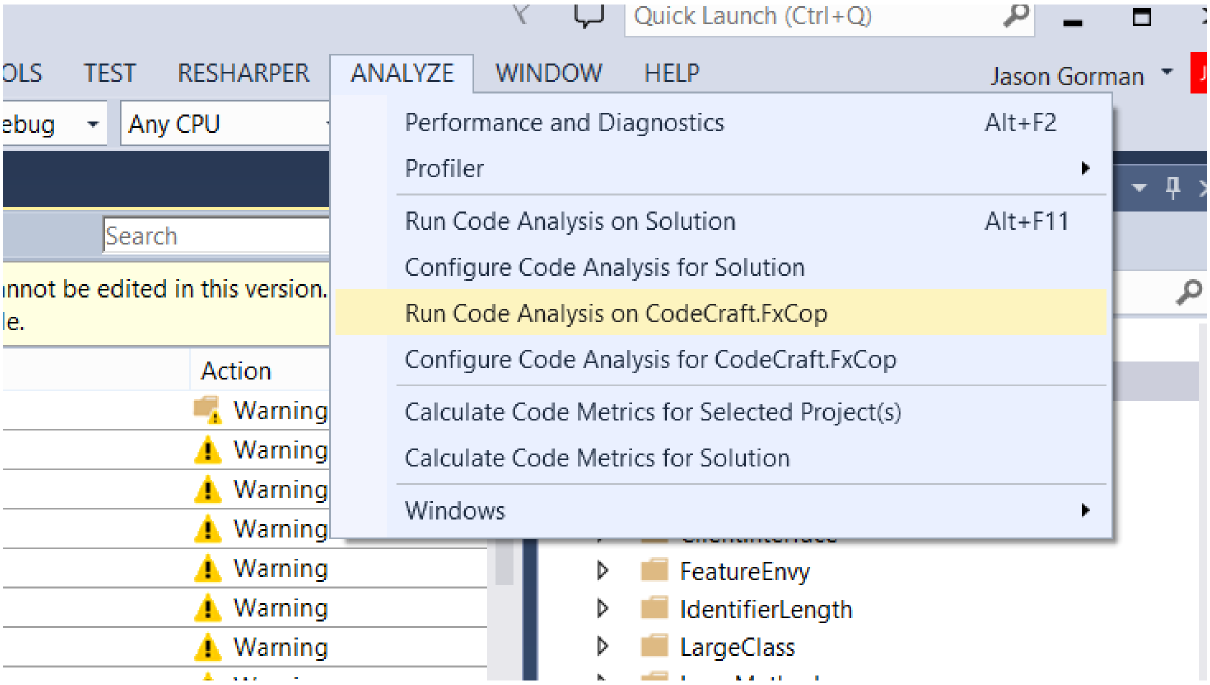
ID	Name	Action
☑ CodeCraft.FxCop		Warning
☑ TT1010	Boolean Parameter	Warning
☑ TT1011	Identifier Length	Warning
☑ TT1012	Feature Envy	Warning
☑ TT1013	Large Class	Warning
☑ TT1014	Long Method	Warning
☑ TT1015	Long Parameter List	Warning
☑ TT1016	Method Complexity Rule	Warning
☑ TT1017	Object Creation Rule	Warning
☑ TT1018	Method Call Rule	Warning
☑ TT1019	Client-Specific Interface	Warning
☑ TT1020	Maximum CollaboratorCount	Warning
□ Microsoft.Design		None

Confirm that all of the *CodeCraft.FxCop* rules are checked.

Save the project’s Properties.

Running Code Analysis

To see if any of your code breaks the CodeCraft.FxCop rules, go to the Analyze menu, and select to run code analysis on your project.


Quick Launch (Ctrl+Q)
OLS TEST RESHARPER ANALYZE WINDOW HELP
Jason Gorman
ebug Any CPU
Search
nnot be edited in this version.
le.
Action
Warning
Warning
Warning
Warning
Warning
Warning
Warning
Performance and Diagnostics Alt+F2
Profiler
Run Code Analysis on Solution Alt+F11
Configure Code Analysis for Solution
Run Code Analysis on CodeCraft.FxCop
Configure Code Analysis for CodeCraft.FxCop
Calculate Code Metrics for Selected Project(s)
Calculate Code Metrics for Solution
Windows
FeatureEnvy
IdentifierLength
LargeClass

Understanding the CodeCraft.FxCop Rules

Boolean Parameter

Boolean method parameters are usually an indicator of a code smell. Typically, methods that take Booleans are doing more than one thing (or are just plain setters, which indicate we may be breaking the *Tell, Don't Ask* design principle.)

```
public void DoFoo(bool isFoo)
{
    if (isFoo)
    {
        // do Foo stuff
    }
    // do other stuff
}
```

It's often better to have the client call a different method instead of the same method with a Boolean flag to indicate which behaviour they want.

```
public void DoFoo()
{
    // do Foo stuff
}

public void DoOther()
{
    // do other stuff
}
```

Identifier Length

Just as too many words can make prose less easy to read, too many words in class names, method names, field and variable names and so on can make code harder to read. For the driving test, you are constrained to a maximum of 20 characters in identifiers.

This is to help you avoid monstrosities like *IdentifierLengthRuleUtilityFactory*.

You will need to think harder than usual about the names you choose.

Feature Envy

According to Martin Fowler's book Refactoring, Feature Envy is when a method of one class has an unhealthy fascination for features of another class. This indicates that the method may be in the wrong class.

In practice, Feature Envy is an ambiguous concept, and is interpreted in many different ways.

For the purposes of the driving test, Feature Envy is when a method of one class uses two or more methods of another class that's declared in the same Visual Studio project.

```
public class Client
{
    public void Foo(Supplier supplier)
    {
        supplier.Fee();
        supplier.Fum();
    }
}
```

The rule is implemented to count property gets and sets as methods, so they are included in our definition of Feature Envy.

Large Class

Classes that are too big, or that do too much, are a code smell. If we build our software out of the simplest parts, then they are easier to understand, easier to test, easier to change, and easier to reuse by composing new collaborations with other simple classes.

In the driving test, classes are allowed to have a maximum of 8 methods (including property gets and sets, and constructors).

Long Method

Likewise, classes should be built out of the simplest methods. In the driving test, methods are restricted to a maximum length of 10 lines of executable code. Blank lines and curly braces aren't counted.

This means that a class is restricted to a maximum of 80 lines of code.

Long Parameter List

Methods that have lots of parameters are usually doing more than one thing. Also, long parameter lists have a tendency to get longer, so their signatures are frequently changing, breaking client code.

In the driving test, methods can have a maximum of 3 parameters.

Method Complexity Rule

Complex methods that have many possible paths through the code are difficult to understand, hard to test and easy to break.

In the driving test, methods can have a maximum of 2 branches (a maximum *cyclomatic complexity* of 3).

Branches include if statements, loops, combinations of conditionals, and switch cases.

Object Creation Rule

In the driving test, we require that dependencies between classes in your project be swappable by *dependency injection*. To enforce this, we have added a rule that prevents you from instantiating your own classes.

This would not be allowed:

```
public class Client
{
    public void Foo()
    {
        new Supplier().Fee();
    }
}
```

Instead, use dependency injection:

```
public class Client
{
    public void Foo(ISupplier supplier)
    {
        supplier.Fee();
    }
}
```

EXEMPTION: This rule does not apply in factory or builder methods. That is, methods that start with "Create" or "Build" and have an abstract return type.

Method Call Rule

Clean code means constructing our software out of simple *swappable* parts. In C#, for parts to be swappable, that means we must be able to substitute a different implementation for every method of another project class that's used.

```
public class Client
{
    public void Foo(Supplier supplier)
    {
        supplier.Fee();
    }
}

public class Supplier
{
    public void Fee()
    {
        // implementation A
    }
}
```

In this example, we invoke a concrete method *Fee()* on supplier. That's not easily swappable with a different implementation. To make it swappable, *Fee()* – as Client sees it – needs to be abstract or virtual.

```
public class Client
{
    public void Foo(ISupplier supplier)
    {
        supplier.Fee();
    }
}

public interface ISupplier
{
    void Fee();
}

public class SupplierA : ISupplier
{
    public void Fee()
    {
        // implementation A
    }
}
```

In this example, we invoke a concrete method *Fee()* on supplier. That's not easily swappable with a different implementation. To make it swappable, *Fee()* – as Client sees it – needs to be abstract or virtual.

An implication of this is that, in the driving test, you can't invoke constructors on classes that are declared in your project. (You can, of course, invoke concrete methods and constructors on, say, core library classes.)

EXEMPTION: This rule does not apply in factory or builder methods. That is, methods that start with "Create" or "Build" and return an abstract type. E.g.

```
public class Client
{
    public void Foo(ISupplierFactory factory)
    {
        factory.CreateSupplier().Fee();
    }
}

public interface ISupplierFactory
{
    ISupplier CreateSupplier();
}

class SupplierAFactory : ISupplierFactory
{
    public ISupplier CreateSupplier()
    {
        return new SupplierA();
    }
}
```

Invocations of the factory or builder methods themselves must be swappable.

Client-Specific Interface

The Interface Segregation Principle (the “I” in SOLID) recommends that our classes present interfaces that include only those methods (including property gets and sets) that the client needs to use.

```
public class Client
{
    public void Foo(ISupplier supplier)
    {
        supplier.Fee();
    }
}

public interface ISupplier
{
    void Fee();
    void Fum();
}

public class Supplier : ISupplier
{
    public void Fee()
    {
        //
    }

    public void Fum()
    {
        //
    }
}
```

This code breaks the principle, because *ISupplier* includes a method that *Client* doesn’t use. We need to break up the interface to make it client-specific:

```
public class Client
{
    public void Foo(IFeeable supplier)
    {
        supplier.Fee();
    }
}

public interface IFeeable
{
    void Fee();
}

public interface IFummable
{
    void Fum();
}

public class Supplier : IFeeable, IFummable
{
```

Max Collaborator Count

To limit class coupling, the driving test requires that none of your classes collaborate with more than 3 other types in your project.

This includes fields, parameters and local variables. It does not include base types.

HALL PASSES - SuppressMessage

The aim of the driving test is to challenge you to produce working software under tight code quality constraints. But we appreciate that blindly following rules, even when they make the code worse, can lead to problems.

FxCop provides a mechanism for telling the code analysis tool to ignore certain rules for a specific class, field or method.

```
#define CODE_ANALYSIS
using System.Diagnostics.CodeAnalysis;
using CodeCraft.FxCop.ObjectCreation;
using Microsoft.FxCop.Sdk;

namespace CodeCraft.FxCop.MethodCall
{
    public class MethodCallRule : MethodRuleBase
    {
        [SuppressMessage("CodeCraft.FxCop", "TT1018:MethodCallRule")]
        public MethodCallRule() :
            base(
            "MethodCallRule", "CodeCraft.FxCop.MethodCall.MethodCallRuleMetadata.xml",
            typeof (ObjectCreationRule).Assembly)
        {
        }
```

We can adorn the code object we wish to use our Hall Pass on with the **SuppressMessage** attribute (found in the *System.Diagnostics.CodeAnalysis* namespace).

Remember to start the code module with the **#define CODE_ANALYSIS** pre-processor directive.

In the driving test, you can use a maximum of 3 Hall Passes.