CODE SMELLS

Table of contents

1. Avoid types too big

1.1 Description

This rule matches types with more than 200 lines of code. **Only lines of code in JustMyCode methods are taken account.**

Types where *NbLinesOfCode > 200* are extremely complex to develop and maintain.

See the definition of the NbLinesOfCode metric here
http://www.cppdepend.com/Metrics#NbLinesOfCode

Maybe you are facing the **God Class** phenomenon: A **God Class** is a class that controls way too many other classes in the system and has grown beyond all logic to become *The Class That Does Everything*.

1.2 CQLinq Query

```
warnif count > 0 from t in JustMyCode.Types where

// First filter on type to optimize
t.NbLinesOfCode > 200


// What matters is the # lines of code in JustMyCode
let locJustMyCode = t.MethodsAndContructors.Where(m => JustMyCode.Contains(m)).Sum(m
=> m.NbLinesOfCode)
where locJustMyCode > 200

let isStaticWithNoMutableState = (t.IsStatic && t.Fields.Any(f => !f.IsImmutable))
let staticFactor = (isStaticWithNoMutableState ? 0.2 : 1)

orderby locJustMyCode descending
select new {
t,
locJustMyCode,
t.Methods,
t.Fields,

Debt = (staticFactor*locJustMyCode.Linear(200, 1, 2000, 10)).ToHours().ToDebt(),

// The annual interest varies linearly from interest for severity major for 300 loc
// to interest for severity critical for 2000 loc
AnnualInterest = staticFactor*(locJustMyCode.Linear(
200,  Severity.Medium.AnnualInterestThreshold().Value.TotalMinutes,
2000,
Severity.Critical.AnnualInterestThreshold().Value.TotalMinutes)).ToMinutes().ToAnnualInt
erest() }
```

1.3 How to Fix Issues

Types with many lines of code should be split in a group of smaller types.

To refactor a God Class you'll need patience, and you might even need to recreate everything from scratch. Here are a few refactoring advices:

- The logic in the God Class must be splitted in smaller classes. These smaller classes can eventually become private classes nested in the original God Class, whose instances objects become composed of instances of smaller nested classes.
- Smaller classes partitioning should be driven by the multiple responsibilities handled by the God Class. To identify these responsibilities it often helps to look for subsets of methods strongly coupled with subsets of fields.
- If the God Class contains way more logic than states, a good option can be to define one or several static classes that contains no static field but only pure static methods. A pure static method is a function that computes a result only from inputs parameters, it doesn't read nor assign any static or instance field. The main advantage of pure static methods is that they are easily testable.
- Try to maintain the interface of the God Class at first and delegate calls to the new extracted classes. In the end the God Class should be a pure facade without its own logic. Then you can keep it for convenience or throw it away and start to use the new classes only.
- Unit Tests can help: write tests for each method before extracting it to ensure you don't break functionality.
- The estimated Debt, which means the effort to fix such issue, varies linearly from 1 hour for a 200 lines of code type, up to 10 hours for a type with 2.000 or more lines of code.
- In Debt and Interest computation, this rule takes account of the fact that static types with no mutable fields are just a collection of static methods that can be easily splitted and moved from one type to another.

2. Avoid types too too many methods

2.1 Description

This rule matches types with more than 20 methods. Such type might be hard to understand and maintain.
Notice that methods like constructors or property and event accessors are not taken account.
Having many methods for a type might be a symptom of too many responsibilities implemented.
Maybe you are facing the God Class phenomenon: A God Class is a class that controls way too many other classes in the system and has grown beyond all logic to become The Class That Does Everything.

2.2 CQLinq Query

```
warnif count > 0 from t in JustMyCode.Types

// Optimization: Fast discard of non-relevant types
where t.Methods.Count() > 20

// Don't match these methods
let methods = t.Methods.Where(
m => !(m.IsGeneratedByCompiler ||
m.IsConstructor || m.IsClassConstructor))

where methods.Count() > 20
orderby methods.Count() descending

let isStaticWithNoMutableState = (t.IsStatic && t.Fields.Any(f => !f.IsImmutable))
let staticFactor = (isStaticWithNoMutableState ? 0.2 : 1)

select new {
t,
nbMethods = methods.Count(),
instanceMethods = methods.Where(m => !m.IsStatic),
staticMethods = methods.Where(m => m.IsStatic),

t.NbLinesOfCode,

Debt = (staticFactor*methods.Count().Linear(20, 1, 200, 10)).ToHours().ToDebt(),

// The annual interest varies linearly from interest for severity major for 30 methods
// to interest for severity critical for 200 methods
AnnualInterest = (staticFactor*methods.Count().Linear(
20,  Severity.Medium.AnnualInterestThreshold().Value.TotalMinutes,
```

```
200,
Severity.Critical.AnnualInterestThreshold().Value.TotalMinutes)).ToMinutes().ToAnnualInt
erest() }
```

2.3 How to Fix Issues

To refactor properly a God Class please read HowToFix advices from the default rule Types to Big. // The estimated Debt, which means the effort to fix such issue, varies linearly from 1 hour for a type with 20 methods, up to 10 hours for a type with 200 or more methods.

In Debt and Interest computation, this rule takes account of the fact that static types with no mutable fields are just a collection of static methods that can be easily splitted and moved from one type to another.

3. Avoid types with too many fields

3.1 Description

This rule matches types with more than 15 fields. Such type might be hard to understand and maintain.

Notice that constant fields and static-readonly fields are not counted. Enumerations types are not counted also.

Having many fields for a type might be a symptom of too many responsibilities implemented.

3.2 CQLinq Query

```
warnif count > 0 from t in JustMyCode.Types

// Optimization: Fast discard of non-relevant types
where !t.IsEnumeration &&
t.Fields.Count() > 15

// Count instance fields and non-constant static fields
let fields = t.Fields.Where(f =>
!f.IsGeneratedByCompiler &&
!(f.IsStatic ) &&
JustMyCode.Contains(f) )

where fields.Count() > 15

let methodsAssigningFields = fields.SelectMany(f => f.MethodsAssigningMe)

orderby fields.Count() descending
select new {
t,
instanceFields = fields.Where(f => !f.IsStatic),
staticFields = fields.Where(f => f.IsStatic),
methodsAssigningFields ,

Debt = fields.Count().Linear(15, 1, 200, 10).ToHours().ToDebt(),

// The annual interest varies linearly from interest for severity major for 30 methods
// to interest for severity critical for 200 methods
AnnualInterest = fields.Count().Linear(15,
Severity.Medium.AnnualInterestThreshold().Value.TotalMinutes,
200,
Severity.Critical.AnnualInterestThreshold().Value.TotalMinutes).ToMinutes().ToAnnualInte
rest() }
```

3.3 How to Fix Issues

To refactor such type and increase code quality and maintainability, certainly you'll have to group subsets of fields into smaller types and dispatch the logic implemented into the methods into these smaller types.

More refactoring advices can be found in the default rule Types to Big, HowToFix section.

The estimated Debt, which means the effort to fix such issue, varies linearly from 1 hour for a type with 15 fields, to up to 10 hours for a type with 200 or more fields.

4. Avoid methods too big, too complex

4.1 Description

This rule matches methods where ILNestingDepth > 2 and (NbLinesOfCode > 35 or CyclomaticComplexity > 20 Such method is typically hard to understand and maintain.

Maybe you are facing the God Method phenomenon. A "God Method" is a method that does way too many processes in the system and has grown beyond all logic to become The Method That Does Everything. When need for new processes increases suddenly some programmers realize: why should I create a new method for each processe if I can only add an if.

See the definition of the CyclomaticComplexity metric here:
http://www.cppdepend.com/Metrics#CC

4.2 CQLinq Query

```
warnif count > 0 from m in JustMyCode.Methods where
(m.NbLinesOfCode > 35 ||
m.CyclomaticComplexity > 20)

let complexityScore = m.NbLinesOfCode/2 + m.CyclomaticComplexity

orderby complexityScore descending,
m.CyclomaticComplexity descending
select new {
m,
m.NbLinesOfCode,
m.CyclomaticComplexity,
complexityScore,

Debt = complexityScore.Linear(30, 40, 400, 8*60).ToMinutes().ToDebt(),

// The annual interest varies linearly from interest for severity minor
// to interest for severity major
AnnualInterest = complexityScore .Linear(30,
Severity.Medium.AnnualInterestThreshold().Value.TotalMinutes,
200,
2*(Severity.High.AnnualInterestThreshold().Value.TotalMinutes)).ToMinutes().ToAnnualInte
rest() }
```

4.3 How to Fix Issues

A large and complex method should be split in smaller methods, or even one or several classes can be created for that.

During this process it is important to question the scope of each variable local to the method. This can be an indication if such local variable will become an instance field of the newly created class(es).

Large switch…case structures might be refactored through the help of a set of types that implement a common interface, the interface polymorphism playing the role of the switch cases tests.

Unit Tests can help: write tests for each method before extracting it to ensure you don't break functionality.

The estimated Debt, which means the effort to fix such issue, varies from 40 minutes to 8 hours, linearly from a weighted complexity score.

5. Avoid methods with too many parameters

5.1 Description

This rule matches methods with more than 8 parameters. Such method is painful to call and might degrade performance. See the definition of the NbParameters metric here: http://www.cppdepend.com/Metrics#NbParameters

5.2 CQLinq Query

```
warnif count > 0 from m in JustMyCode.Methods where
m.NbParameters >= 7
orderby m.NbParameters descending
select new {
m,
m.NbParameters,

Debt = m.NbParameters.Linear(7, 1,  40, 6).ToHours().ToDebt(),

// The annual interest varies linearly from interest for severity Minor for 7 parameters
// to interest for severity Critical for 40 parameters
AnnualInterest = m.NbParameters.Linear(7,
Severity.Medium.AnnualInterestThreshold().Value.TotalMinutes,
40,
Severity.Critical.AnnualInterestThreshold().Value.TotalMinutes).ToMinutes().ToAnnualInte
rest() }
```

5.3 How to Fix Issues

More properties/fields can be added to the declaring type to handle numerous states. An alternative is to provide a class or a structure dedicated to handle arguments passing.

The estimated Debt, which means the effort to fix such issue, varies linearly from 1 hour for a method with 7 parameters, up to 6 hours for a method with 40 or more parameters.

6. Avoid methods with too many local variables

6.1 Description

This rule matches methods with more than 15 variables.

Methods where NbVariables > 8 are hard to understand and maintain. Methods where NbVariables > 15 are extremely complex and must be refactored.

See the definition of the Nbvariables metric here: http://www.cppdepend.com/Metrics#Nbvariables

6.2 CQLinq Query

```
warnif count > 0 from m in JustMyCode.Methods where
m.NbVariables > 15
orderby m.NbVariables descending
select new {
m,
m.NbVariables,

Debt = m.NbVariables.Linear(15, 1, 80, 6).ToHours().ToDebt(),

// The annual interest varies linearly from interest for severity Minor for 15 variables
// to interest for severity Critical for 80 variables
AnnualInterest = m.NbVariables.Linear(15,
Severity.Medium.AnnualInterestThreshold().Value.TotalMinutes,
80,
Severity.Critical.AnnualInterestThreshold().Value.TotalMinutes).ToMinutes().ToAnnualInte
rest() }
```

6.3 How to Fix Issues

To refactor such method and increase code quality and maintainability, certainly you'll have to split the method into several smaller methods or even create one or several classes to implement the logic.

During this process it is important to question the scope of each variable local to the method. This can be an indication if such local variable will become an instance field of the newly created class(es).

The estimated Debt, which means the effort to fix such issue, varies linearly from 10 minutes for a method with 15 variables, up to 2 hours for a methods with 80 or more variables.

7. Avoid types too too many methods

7.1 Description

Method overloading is the ability to create multiple methods of the same name with different implementations, and various set of parameters.

This rule matches sets of methods with 6 overloads or more.

Such method set might be a problem to maintain and provokes coupling higher than necessary.

See the definition of the NbOverloads metric here:
http://www.cppdepend.com/Metrics#NbOverloads

7.2 CQLinq Query

```
warnif count > 0 from m in JustMyCode.Methods where
m.NbOverloads >= 6 &&
!m.IsOperator // Don't report operator overload
orderby m.NbOverloads descending
let overloads =
m.IsConstructor ? m.ParentType.Constructors :
m.ParentType.Methods.Where(m1 => m1.SimpleName == m.SimpleName)
select new {
m,
overloads,
Debt = 2.ToMinutes().ToDebt(),
Severity = Severity.Medium }
```

7.3 How to Fix Issues

Typically the too many overloads phenomenon appears when an algorithm takes a various set of in-parameters. Each overload is presented as a facility to provide a various set of in-parameters. In such situation, the C# and VB.NET language feature named Named and Optional arguments should be used.

The too many overloads phenomenon can also be a consequence of the usage of the visitor design pattern http://en.wikipedia.org/wiki/Visitor_pattern since a method named Visit() must be provided for each sub type. In such situation there is no need for fix.

Sometime too many overloads phenomenon is not the symptom of a problem, for example when a numeric to something conversion method applies to all numeric and nullable numeric types.

The estimated Debt, which means the effort to fix such issue, is of 2 minutes per method overload.

8. Avoid methods potentially poorly commented

8.1 Description

This rule matches methods with less than 20% of comment lines and that have at least 20 lines of code. Such method might need to be more commented.

See the definitions of the Comments metric here:
http://www.cppdepend.com/Metrics#PercentageComment
http://www.cppdepend.com/Metrics#NbLinesOfComment

Notice that only comments about the method implementation (comments in method body) are taken account.

8.2 CQLinq Query

```
warnif count > 0 from m in JustMyCode.Methods where
m.PercentageComment < 20 &&
m.NbLinesOfCode > 20

let nbLinesOfCodeNotCommented = m.NbLinesOfCode - m.NbLinesOfComment

orderby nbLinesOfCodeNotCommented descending

select new {
m,
m.PercentageComment,
m.NbLinesOfCode,
m.NbLinesOfComment,
nbLinesOfCodeNotCommented,

Debt = nbLinesOfCodeNotCommented .Linear(20, 2, 200, 20).ToMinutes().ToDebt(),

// The annual interest varies linearly from interest for severity major for 300 loc
// to interest for severity critical for 2000 loc
AnnualInterest = m.PercentageComment.Linear(
0,  8 *(Severity.Medium.AnnualInterestThreshold().Value.TotalMinutes),
20,
Severity.Medium.AnnualInterestThreshold().Value.TotalMinutes).ToMinutes().ToAnnualIntere
st() }
```

8.3 How to Fix Issues

Typically add more comment. But code commenting is subject to controversy. While poorly written and designed code would needs a lot of comment to be understood, clean code doesn't need that much comment, especially if variables and methods are properly named and convey enough information. Unit-Test code can also play the role of code commenting.

However, even when writing clean and well-tested code, one will have to write hacks at a point, usually to circumvent some API limitations or bugs. A hack is a non-trivial piece of code, that doesn't make sense at first glance, and that took time and web research to be found. In such situation comments must absolutely be used to express the intention, the need for the hacks and the source where the solution has been found.

The estimated Debt, which means the effort to comment such method, varies linearly from 2 minutes for 10 lines of code not commented, up to 20 minutes for 200 or more, lines of code not commented.

9. Avoid types with poor cohesion

9.1 Description

This rule is based on the LCOM code metric, LCOM stands for Lack Of Cohesion of Methods.
hSee the definition of the LCOM metric here

http://www.cppdepend.com/Metrics#LCOM

The LCOM metric measures the fact that most methods are using most fields. A class is considered utterly cohesive (which is good) if all its methods use all its instance fields.

Only types with enough methods and fields are taken account to avoid bias. The LCOM takes its values in the range [0-1].

This rule matches types with LCOM higher than 0.8. Such value generally pinpoints a poorly cohesive class.

9.2 CQLinq Query

```
warnif count > 0 from t in JustMyCode.Types where
t.LCOM > 0.8  &&
t.NbFields > 10 &&
t.NbMethods >10

let poorCohesionScore = 1/(1.01 - t.LCOM)
orderby poorCohesionScore descending

select new {
t,
t.LCOM,
t.NbMethods,
t.NbFields,
poorCohesionScore,

Debt = poorCohesionScore.Linear(5, 5, 50, 4*60).ToMinutes().ToDebt(),

// The annual interest varies linearly from interest for severity Minor for low
poorCohesionScore
// to 4 times interest for severity Major for high poorCohesionScore
AnnualInterest = poorCohesionScore.Linear(5,
Severity.Medium.AnnualInterestThreshold().Value.TotalMinutes,
50,
4*(Severity.High.AnnualInterestThreshold().Value.TotalMinutes)).ToMinutes().ToAnnualInte
rest() }
```

9.3 How to Fix Issues

To refactor a poorly cohesive type and increase code quality and maintainability, certainly you'll have to split the type into several smaller and more cohesive types that together, implement the same logic.

The estimated Debt, which means the effort to fix such issue, varies linearly from 5 minutes for a type with a low poorCohesionScore, up to 4 hours for a type with high poorCohesionScore.