WMC - Weighted Methods for Class
Weighted methods for Class (WMC) was originally proposed by C&K as the sum of all the complexities of the methods in the class. Rather than use Cyclomatic Complexity they assigned each method a complexity of one making WMC equal to the number of methods in the class. Most 'classic' implementations follow this rule. C&Ks view of WMC was -

1. The number of methods and the complexity of methods involved is a predictor of how much time and effort is required to develop and maintain the class
2. The larger the number of methods in a class the greater the potential impact on children, since children will inherit all the methods defined in the class
3. Classes with large numbers of methods are likely to be more application specific, limiting the possibility of reuse.

I have quite a few issues with this metric, starting with the name - don't call a metric **Weighted** Methods per class if you are immediately going to assign each method an equal and arbitrary value of 1. Their view that classes with more methods are less likely to be reused seems strange in that, potentially, there is more there to reuse! I think that number of methods is a better name for this method and that Total Cyclomatic

DIT - Depth of Inheritance Tree
Depth of Inheritance Tree (DIT) is a count of the classes that a particular class inherits from. C&K suggested the following consequences based on the depth of inheritance -

 The deeper a class is in the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behavior
 Deeper trees constitute greater design complexity, since more methods and classes are involved
 The deeper a particular class is in the hierarchy, the greater the potential reuse of inherited methods

The first two points seem to be interpreted as 'bad' indicators while the third seems to be suggested as 'good' - making it difficult to decide whether DIT should be large or small. I would suggest that a deep hierarchy is a good thing - you don't really want to punish OO programmers for using inheritance.

There are a couple of considerations here -

- Should the Inheritance Tree stop at the boundary of the system under examination or should it also include library classes (even whether Object should be included since all classes ultimately inherit from this)? As C&K took the view that it was the numbers of inherited methods that increased the potential complexity then possibly all of the classes (inside and outside the code under examination) should be included
- Java classes have a single inheritance tree but should this also include interfaces that are implemented and may include data defined in these interfaces such as statics? See the point about the longest chain below.

- Java interfaces have multiple inheritance should the depth of the tree be a count of all the interfaces or simply be the longest chain of interefaces? C&K use the longest chain in their calculations but if the issue is the greater number of inherited functionality (data and methods) how does this make sense.

NOC - Number of Children

Number of Children (NOC) is defined by C&K the number of immediate subclasses of a class. C&K's view was that -

- The greater the number of children, the greater the level of reuse, since inheritance is a form of reuse.
- The greater the number of children, the greater the likelihood of improper abstraction of the parent class. If a class has a large number of children, it may be a case of misuse of subclassing.
- The number of children gives an idea of the potential influence a class has on the design. If a class has a large number of children, it may require more testing of the methods in that class.
- So their view was that increasing the number of children was improving reusability but if you didn't do it correctly you could make a mess of your design and increase the burden on your testers. This would suggest that this metric might point you towards code that you should look at - but you were then going to have to make a qualitative judgement about that code.
- The calculation itself seems very simple. But should you perhaps include all the subclasses that ultimately have that class as their parent. And what about interfaces? - what relationship does a class that implements the interface have to that interface - is it a child or an instance? And if you include classes that implement the interface do you include the classes that implement the child interfaces that inherit from the root interface?

CBO - Coupling between Objects

Coupling between objects (CBO) is a count of the number of classes that are coupled to a particular class i.e. where the methods of one class call the methods or access the variables of the other. These calls need to be counted in both directions so the CBO of class A is the size of the set of classes that class A references and those classes that reference class A. Since this is a set - each class is counted only once even if the reference operates in both directions i.e. if A references B and B references A, B is only counted once.

- C&K viewed that CBO should be as low as possible for three reasons -
- Increased coupling increases interclass dependancies, making the code less modular and less suitable for reuse. In other words if you wanted to package up the code for reuse you might end up having to include code that was not really fundamental to the core functionality.
- More coupling means that the code becomes more difficult to maintain since an alteration to code in one area runs a higher risk of affecting code in another (linked) area.
- The more links between classes the more complex the code and the more difficult it will be to test

- The only consideration in calculating this metric is whether only the classes written for the project should be considered or whether library classes should be included.

RFC - Response For Class
This is the size of the Response set of a class. The Response set for a class is defined by C&K as 'a set of methods that can potentially be executed in response to a message received by an object of that class'. That means all the methods in the class and all the methods that are called by methods in that class. As it is a set each called method is only counted once no matter how many times it is called. C&K's view was that -

- If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes more complicated since it requires a greater level of understanding required on the part of the tester
- The larger the number of methods that can be invoked from a class, the greater the complexity of the class
- A worst case value for possible responses will assist in appropriate allocation of testing time
- This is probably the most clear cut of the metrics - a high RFC suggests that something is wrong. There might be some debate about what a high level of RFC is but if you started at the classes with the highest RFC level in your code and worked down you would probably eliminate a lot of 'bad smells'.

LCOM - Lack of Cohesion of Methods
LCOM is probably the most controversial and argued over of the C&K metrics. In their original incarnation C&K defined LCOM based on the numbers of pairs of methods that shared references to instance variables. Every method pair combination in the class was assessed. If the pair do not share references to any instance variable then the count is increased by 1 and if they do share any instance variables then the count is decreased by 1. LCOM is viewed as a measure of how well the methods of the class co-operate to achieve the aims of the class. A low LCOM value suggests the class is more cohesive and is viewed as better. C&K's rationale for the LCOM method was as follows -

- Cohesiveness of methods within a class is desirable, since it promotes encapsulation.
- Lack of cohesion implies classes should probably be split into two or more subclasses.
- Any measure of disparateness of methods helps identify flaws in the design of classes.
- Low cohesion increases complexity, thereby increasing the likelihood of errors during the development process.

One of the difficulties with the original C&K LCOM measurement was that it's maximum value was dependant on the number of method pairs - with 3 methods you can have 3 method pairs but with 4 you can have 6. This means that to assess whether your value of LCOM is low you have to take into account the number of method pairs - an LCOM of 2 with 3 method pairs should be considered differently to an LCOM of 2 with 100 message pairs. This was addressed by Henderson-Sellars when he introduced LCOM* (sometimes known as LCOM3 with C&Ks LCOM being known as LCOM1). LCOM* is calculated by (numMethods - numAccesses/numInstVars)/(numMethods-1) where numMethods is the number of methods in

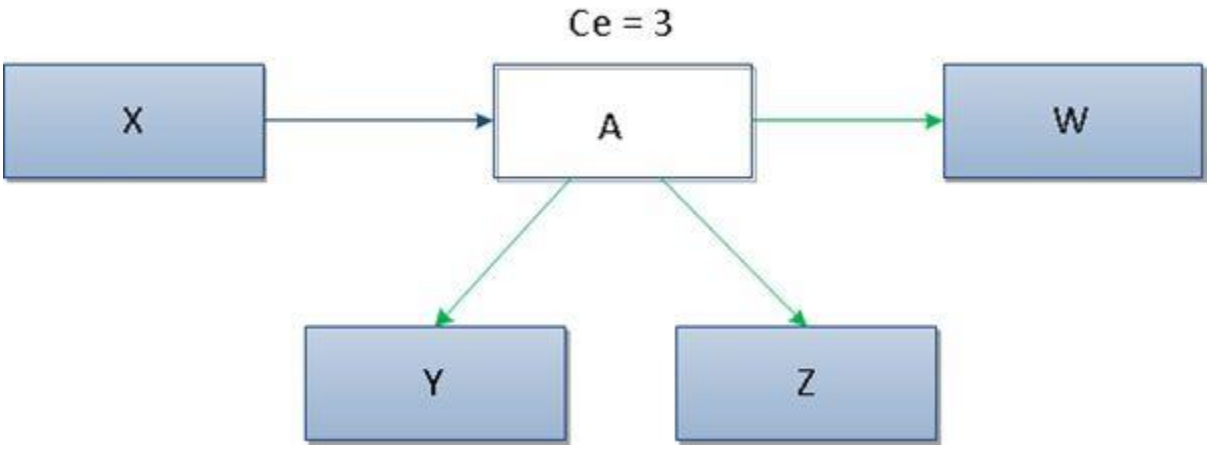
the class, numAccesses is the number of methods in a class that access an instance variable and numInstvars is the number of instance variables. If the number of attributes or methods is zero then LCOM* is zero. LCOM* can be in the range 0 to 2 with values over 1 being viewed as being suggestive of poor design.

There are several other calculations of LCOM and these are reviewed by Etzhorn et al. This covers areas such as the inclusion of inherited variables in the calculation and the inclusion of constructor and destructor methods and they looked at eight possible implementations of LCOM. Their conclusion was that a version of LCOM produced by Li and Henry that included constructor methods, but not inherited variables, gave a result that was closest to cohesion assessed visually from the code.

Although there is a fair amount of debate about how to calculate LCOM and it features in a lot of metrics sets an increasing number of researchers (including Henderson-Sellars) suggest that it is not a particularly useful metric. Perhaps this is also reflected in there being a fair amount of debate about how to calculate LCOM but very little on how to interpret it and how it fits in with other metrics.

EFFERENT COUPLING (CE)

This metric is used to measure interrelationships between classes. As defined, it is a number of classes in a given package, which depends on the classes in other packages. It enables us to measure the vulnerability of the package to changes in packages on which it depends.

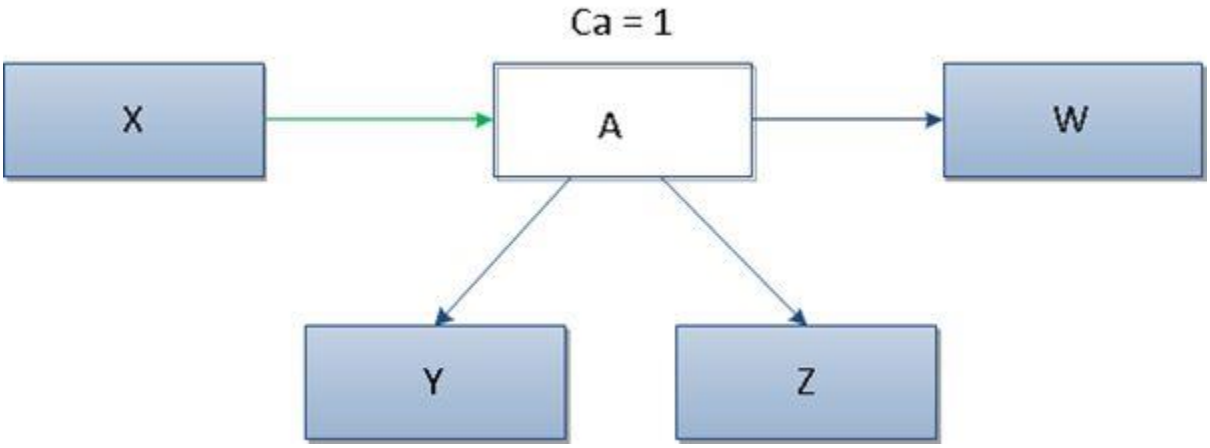


In the pic.1 it can be seen that class A has outgoing dependencies to 3 other classes, that is why metric Ce for this class is 3.

The high value of the metric Ce> 20 indicates instability of a package, change in any of the numerous external classes can cause the need for changes to the package. Preferred values for the metric Ce are in the range of 0 to 20, higher values cause problems with care and development of code.

AFFERENT COUPLING (CA)

This metric is an addition to metric Ce and is used to measure another type of dependencies between packages, i.e. incoming dependencies. It enables us to measure the sensitivity of remaining packages to changes in the analysed package.



In the pic.2 it can be seen that class A has only 1 incoming dependency (from class X), that is why the value for metrics Ca equals 1.

High values of metric Ca usually suggest high component stability. This is due to the fact that the class depends on many other classes. Therefore, it can't be modified significantly because, in this case, the probability of spreading such changes increases.

Preferred values for the metric Ca are in the range of 0 to 500.

NPM - Number of Public Methods

The NPM metric simply counts all the methods in a class that are declared as public. It can be used to measure the size of an API provided by a package.

LCOM3 -Lack of cohesion in methods.

LCOM3 varies between 0 and 2.

m - number of procedures (methods) in class

a - number of variables (attributes in class

μ(A) - number of methods that access a variable (attribute)

$$LCOM3 = \frac{\left(\frac{1}{a}\sum_{j=1}^{a}\mu(A_j)\right) - m}{1 - m}$$

LOC - Lines of Code.

The lines are counted from java binary code and it is the sum of number of fields, number of methods and number of instructions in every method of given class.

AMC: Average Method Complexity

Max _CC: Maximum McCabe's Cyclomatic Complexity values of methods in the same class

Avg _CC Mean McCabe's : Cyclomatic Complexity values of methods in the same class

Bug Number of bugs detected in the class