**Chidamber and Kemerer Metrics Report**

# **Briefing**

Chidamber and Kemerer metrics pertain to information on object coupling, inheritance depth, cohesion between methods (or lack thereof), number of subclasses.. amongst other metrics.

The specific metrics discussed in this report (extracted using the MetricsReloaded plugin for IntelliJ IDEA) and their corresponding abbreviations, are as follows:

CBO - Coupling Between Objects

DIT - Depth of Inheritance Tree

LCOM - Lack of Cohesion of Methods

NOC - Number of Children

RFC - Response for Class

WMC - Weighted Method Complexity

# CBO - Coupling Between Objects

CBO = number of classes to which a class is coupled

Classes are considered coupled when methods declared in one class use methods or instance variables defined by another class.

Multiple accesses of one class to another class are counted as one access. Only method calls and variable references are counted - the use of constants, calls to API declares, handling of events, use of user-defined types, and object instantiations are not counted.

If a method call is polymorphic (either because of Overrides or Overloads), all the classes to which the call can go are included in the coupled count.

In general, high CBO is undesirable. Excessive coupling between object classes is detrimental to modular design and prevents reuse. In order to improve modularity and promote encapsulation, inter-object class couples should be kept to a minimum. The larger the number of couples, the higher the chance that changes in one class will imply changes in other coupled classes, and therefore natural refactoring that comes with growth becomes more difficult.

CBO > 14 is considered too high, as concluded in: [Can Metrics Help Bridging the Gap Between the Improvement of OO Design Quality and Its Automation?](http://www.iro.umontreal.ca/~sahraouh/papers/ICSM00.pdf)

# DIT - Depth of Inheritance Tree

DIT = maximum inheritance path from the class to the root class

The deeper a class is in an inheritance hierarchy, the more methods and variables it is likely to inherit, making it more complex.

Although deep trees promote reuse as a result of method inheritance, a high DIT has been found to increase faults. However, it’s not necessarily the classes deepest in the class hierarchy that have the most faults, usually the most fault-prone classes are the ones in the middle of the tree, since usually the root and deepest classes are consulted often, and, due to familiarity, have low fault-proneness compared to classes in the middle.

The Visual Studio .NET documentation recommends a DIT <= 5, other sources allow up to 8.

# LCOM - Lack of Cohesion of Methods

Take each pair of methods in the class. If they access disjoint sets of instance variables, increase P by one. If they share at least one variable access, increase Q by one.

LCOM = P − Q, if P > Q

LCOM = 0 otherwise

LCOM = 0 indicates a cohesive class.

LCOM > 0 indicates that the class can be split into two or more classes, since its variables belong in disjoint sets.

A high LCOM value indicates disparateness in the functionality provided by the class. This metric can be used to identify classes that are attempting to achieve many different objectives, and consequently are likely to behave in less predictable ways than classes that have lower LCOM values.

# NOC - Number of Children

NOC = number of immediate subclasses of a class

NOC measures the breadth of a class hierarchy, whereas DIT measures the depth.

Depth is generally better than breadth, since it promotes reuse of methods through inheritance. Inheritance levels can be added to increase the depth and reduce the breadth.

A high NOC may indicate:

* High reuse of base class. Inheritance is a form of reuse.
* Base class may require more testing.
* Improper abstraction of the parent class.
* Misuse of sub-classing. In such a case, it may be necessary to group related classes and introduce another level of inheritance.

A class with a high NOC and a high WMC indicates complexity at the top of the class hierarchy. The class is potentially influencing a large number of descendant classes. This can be a sign of poor design.

Not all classes should have the same number of subclasses. Classes higher up in the hierarchy should have more subclasses than those lower down.

# RFC - Response for Class

The response set of a class is a set of methods that can potentially be executed in response to a message received by an object of that class. RFC is simply the number of methods in the set.

RFC = M + R (First-step measure)

RFC’ = M + R’ (Full measure)

M = number of methods in the class

R = number of remote methods directly called by methods of the class

R’ = number of remote methods called, recursively through the entire call tree

A given method is counted only once in R (and R’) even if it is executed by several methods M.

Since RFC specifically includes methods called from outside the class, it is also a measure of the potential communication between the class and other classes.

A large RFC has been found to indicate more faults. Classes with a high RFC are more complex and harder to understand.

The use of RFC’ should be preferred over RFC. RFC was originally defined as a first-level metric because it was not practical to consider the full call tree in manual calculation. With an automated code analysis tool, getting RFC’ values is no longer problematic. As RFC’ considers the entire call tree and not just one first level of it, it provides a more thorough measurement of the executed code.

# WMC - Weighted Method Complexity

WMC is simply the method count for a class.

WMC = number of methods defined in class

A high WMC has been found to lead to more faults. Classes with many methods are likely to be more application specific, limiting the possibility of reuse. A large number of methods also means a greater potential impact on subclasses, since they inherit (all or some of) the methods of the base class.

A high WMC value may indicate a class that should be restructured into smaller classes.

A good WMC is not clear-cut. One way is to limit the number of methods in a class to a couple of dozen. Another way is to have a limit of 10% of classes that can have more than 24 methods. This allows large classes but most classes should be small.

# **Chidamber and Kemerer Metrics in JabRef**

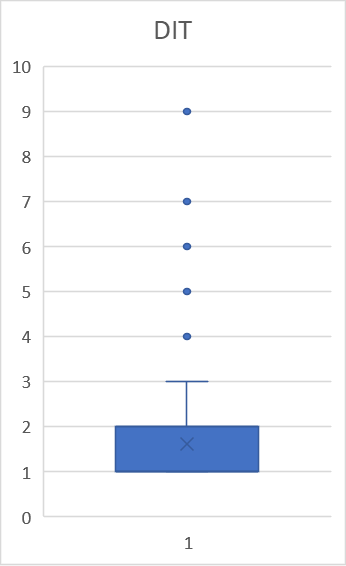
## CBO - Coupling Between Objects

As can be observed in the collected metrics sheet, although the average CBO (11.11) is below the recommended 14, a total of 354 classes have a CBO above that.

192 classes have a CBOin the 20-100 range, and 14 have a CBO > 100, the highest being the **BibEntry** class with 616.

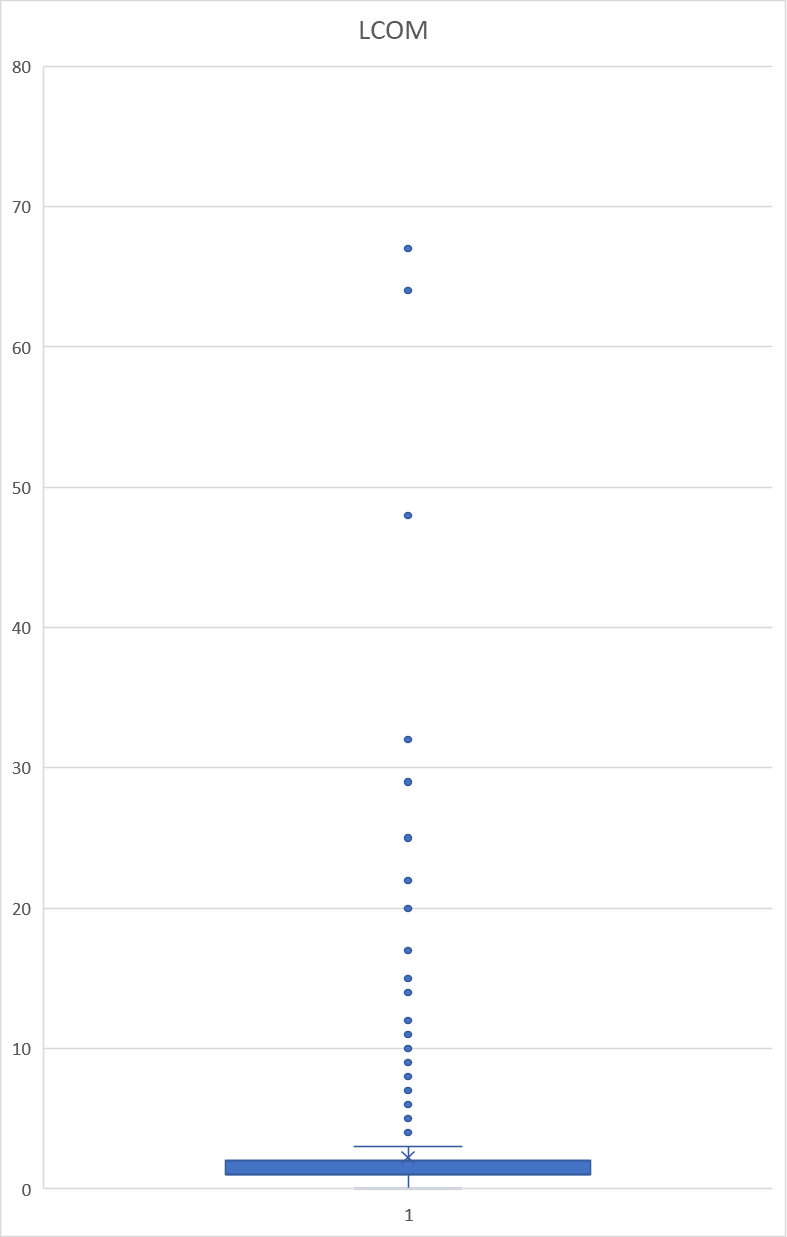
The amount of classes with absurdly high values for coupling are a clear indication of the **Shotgun Surgery**, **Inappropriate Intimacy** and, **Feature Envy** code smells, especially the first one, since changes/refactoring in one class may (in most of the aforementioned cases, *will*) require changes to be made to many different related classes.

## DIT - Depth of Inheritance Tree



Inheritance tree depth in JabRef doesn’t seem to be as big of an issue as CBO, seeing as the average DIT per class is only 1.62 and the highest values for this metric are 7 and 9 for 28 classes. Although, this might be an indication of some needed refactoring to the middle classes of a number of inheritance trees, in order to distribute responsibilities and deliver better abstraction overall.

## LCOM - Lack of Cohesion of Methods



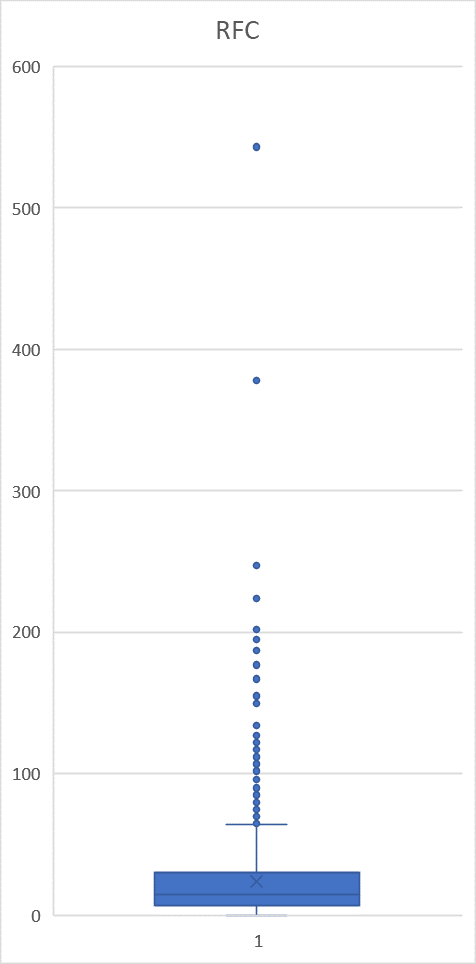
Although LCOM (also known as LCOM1) has some critiques for being too simplistic, and therefore spawned more intricate metrics (LCOM2, LCOM3..), there is a clear volume of classes (25) with an LCOM value above 10 (67 being the highest). This is an indication of the **Large Class** and **Long Parameter List** code smells, which indicate that a number of these classes should have their responsibilities divided into smaller classes. These were indeed some of the documented code smells in our report.

## NOC - Number of Children

The average value for each classes’ NOC is only 0.23, with a small number of classes having an extremely high value. Although not very problematic, this might indicate that another level of abstraction/inheritance should be added in some of these trees, increasing the average DIT (which is desirable up to a certain point) but greatly reducing the NOC, promoting reuse.

An outlier class, which happens to be the one with the highest NOC (80), is the **SimpleCommand** abstract class, since it is the base for an implementation of the Command Pattern (as documented in the Design Patterns section), and therefore is not necessarily an indication of poor design.

## RFC - Response for Class

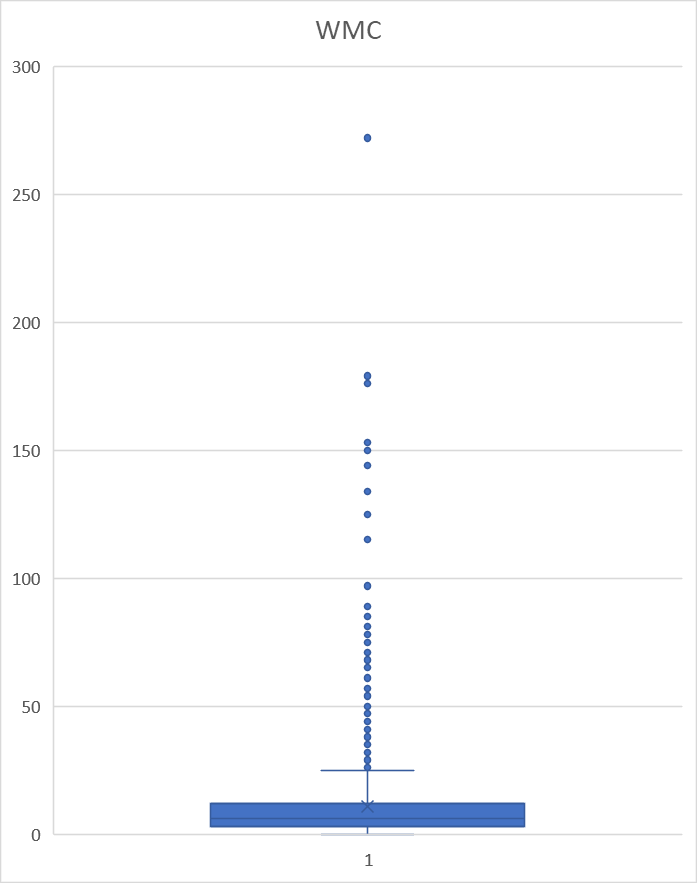


The average RFC per class is 23.86, with a large number of classes (47) holding values above 100 (543 being the highest).

Since RFC specifically includes methods called from outside the class, it is also a measure of the potential communication between the class and other classes, which may indicate code smells similar to the ones related with CBO.

Additionally, classes with a high RFC hold a higher level of complexity and are therefore harder to understand and make changes from the get go.

## WMC - Weighted Method Complexity



There are 173 total classes with a WMC above 25 (272 being the highest).

Although some classes in the system require a higher method count, the number of classes with an absurdly high WMC is a clear indication that their responsibilities should be subdivided into smaller classes (**Large Class** code smell, similar to the conclusion of the LCOM metric).