Chidamber & Kemerer object-oriented metrics suite

The Chidamber & Kemerer metrics suite originally consists of 6 metrics calculated for each class: WMC, DIT, NOC, CBO, RFC and LCOM1. The original suite has later been amended by RFC´, LCOM2, LCOM3 and LCOM4 by other authors.

See also Object-oriented metrics

WMC Weighted Methods Per Class

Despite its long name, WMC is simply the method count for a class.

> **WMC = number of methods defined in class**

Keep WMC down. A high WMC has been found to lead to more faults. Classes with many methods are likely to be more more application specific, limiting the possibility of reuse. WMC is a predictor of how much time and effort is required to develop and maintain the class. A large number of methods also means a greater potential impact on derived classes, since the derived classes inherit (some of) the methods of the base class. Search for high WMC values to spot classes that could be restructured into several smaller classes.

What is a good WMC? Different limits have been defined. One way is to limit the number of methods in a class to, say, 20 or 50. Another way is to specify that a maximum of 10% of classes can have more than 24 methods. This allows large classes but most classes should be small.

> A study of 30 C++ projects suggests that an increase in the average WMC increases the density of bugs and decreases quality. The study suggests "optimal" use for WMC but doesn't tell what the optimum range is. It sounds safe to assume that a high WMC is detrimental in VB as well. *Misra & Bhavsar: Relationships Between Selected Software Measures and Latent Bug-Density: Guidelines for Improving Quality. Springer-Verlag 2003.*

> **Implementation details.** Project Analyzer counts each Sub, Function, Operator and Property accessor into WMC. Constructors and event handlers are also counted as methods. Event definitions, Custom Events, API Declare statements and <DLLImport> procedures are not counted in WMC. WMC includes only those methods that are defined in the class, not any inherited methods. Overriding and shadowing methods defined in the class are counted, since they form a new implementation of a method.
>
> Each property accessor (Get, Set, Let) is counted separately in WMC. The reasoning behind this is that each of Get, Set and Let provides a separate way to access to the underlying property and is essentially a method of the class. The alternative way to using properties would be to write accessor functions: Function getProperty and Sub setProperty. Both of these would also be counted in WMC the same way we count each property accessor.
>
> Even though Project Analyzer counts WMC as a simple method count, it *could* be a weighted count. In principle, we *could* weigh each method by its size or complexity. This is not implemented, though.

Readings

- Shyam R. Chidamber, Chris F. Kemerer. A Metrics suite for Object Oriented design. M.I.T. Sloan School of Management E53-315. 1993. http://uweb.txstate.edu/~mq43/CS5391/Papers/Metrics/OOMetrics.pdf
- Victor Basili, Lionel Briand and Walcelio Melo. A Validation of Object-Oriented Design Metrics as Quality Indicators. IEEE Transactions on Software Engineering. Vol. 22, No. 10, October 1996. http://www.cs.umd.edu/users/basili/publications/journals/J60.pdf

DIT Depth of Inheritance Tree

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

The deeper a class is in the hierarchy, the more methods and variables it is likely to inherit, making it more complex. Deep trees as such indicate greater design complexity. Inheritance is a tool to manage complexity, really, not to not increase it. As a positive factor, deep trees promote reuse because 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. Glasberg et al. have found out that the most fault-prone classes are the ones in the middle of the tree. According to them, root and deepest classes are consulted often, and due to familiarity, they have low fault-proneness compared to classes in the middle.

A recommended DIT is 5 or less. The Visual Studio .NET documentation recommends that DIT <= 5 because excessively deep class hierachies are complex to develop. Some sources allow up to 8.

> A study of 30 C++ projects suggests that an increase in DIT increases the density of bugs and decreases quality. The study suggests "optimal" use for DIT but doesn't tell what the optimum is. It sounds safe to assume that a deep inheritance tree is detrimental in VB as well. *Misra & Bhavsar: Relationships Between Selected Software Measures and Latent Bug-Density: Guidelines for Improving Quality. Springer-Verlag 2003.*

> **Implementation details.** Project Analyzer takes only implementation inheritance (Inherits statement, not Implements statement) into account when calculating DIT.

Special cases. When a class inherits directly from System.Object (or has no Inherits statement), DIT=1. For a class that inherits from an unknown (unanalyzed) class, DIT=2. This is because the unknown class eventually inherits from System.Object and 2 is the minimum inheritance depth. It could also be more.

For all VB Classic classes, DIT = 0 because no inheritance is available.

Readings

- Shyam R. Chidamber, Chris F. Kemerer. A Metrics suite for Object Oriented design. M.I.T. Sloan School of Management E53-315. 1993. http://uweb.txstate.edu/~mg43/CS5391/Papers/Metrics/OOMetrics.pdf
- Victor Basili, Lionel Briand and Walcelio Melo. A Validation of Object-Oriented Design Metrics as Quality Indicators. IEEE Transactions on Software Engineering. Vol. 22, No. 10, October 1996. http://www.cs.umd.edu/users/basili/publications/journals/J60.pdf
- Daniela Glasberg, Khaled El Emam, Walcelio Melo, Nazim Madhavji: Validating Object-Oriented Design Metrics on a CommercialJava Application. 2000. http://iit-iti.nrc-cnrc.gc.ca/iit-publications-iti/docs/NRC-44146.pdf
- Khaled El Emam, Walcelio Melo, Javam, C. Machado. The prediction of faulty classes using object-oriented design metrics. The Journal of Systems and Software 56 (2001) 63-75.

NOC Number of Children

NOC = number of immediate sub-classes of a class

NOC equals the number of immediate child classes derived from a base class. In Visual Basic .NET one uses the Inherits statement to derive sub-classes. In classic Visual Basic inheritance is not available and thus NOC is always zero.

NOC measures the breadth of a class hierarchy, where maximum DIT measures the depth. Depth is generally better than breadth, since it promotes reuse of methods through inheritance. NOC and DIT are closely related. Inheritance

levels can be added to increase the depth and reduce the breadth.

A high NOC, a large number of child classes, can indicate several things:

- 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.

High NOC has been found to indicate fewer faults. This may be due to high reuse, which is desirable.

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. A redesign may be required.

Not all classes should have the same number of sub-classes. Classes higher up in the hierarchy should have more sub-classes then those lower down.

Readings

- Shyam R. Chidamber, Chris F. Kemerer. A Metrics suite for Object Oriented design. M.I.T. Sloan School of Management E53-315. 1993. http://uweb.txstate.edu/~mg43/CS5391/Papers/Metrics/OOMetrics.pdf
- Victor Basili, Lionel Briand and Walcelio Melo. A Validation of Object-Oriented Design Metrics as Quality Indicators. IEEE Transactions on Software Engineering. Vol. 22, No. 10, October 1996. http://www.cs.umd.edu/users/basili/publications/journals/J60.pdf
- Radu Marinescu. Using Object-Oriented Metrics for Automatic Design Flaws Detection in Large Scale Systems. http://loose.upt.ro/download/papers/ecoop98.pdf

CBO Coupling between Object Classes

CBO = number of classes to which a class is coupled

Two classes are coupled when methods declared in one class use methods or instance variables defined by the other class. The uses relationship can go either way: both uses and used-by relationships are taken into account, but only once.

Multiple accesses to the same class are counted as one access. Only method calls and variable references are counted. Other types of reference, such as use of constants, calls to API declares, handling of events, use of user-defined types, and object instantiations are ignored. 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.

High CBO is undesirable. Excessive coupling between object classes is detrimental to modular design and prevents reuse. The more independent a class is, the easier it is to reuse it in another application. 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 sensitivity to changes in other parts of the design, and therefore maintenance is more difficult. A high coupling has been found to indicate fault-proneness. Rigorous testing is thus needed. — How high is too high? CBO>14 is too high, say Sahraoui, Godin & Miceli in their article (link below).

A useful insight into the 'object-orientedness' of the design can be gained from the system wide distribution of the class fan-out values. For example a system in which a single class has very high fan-out and all other classes have low or zero fan-outs, we really have a structured, not an object oriented, system.

Implementation details. The definition of CBO deals with the *instance variables* and *all the methods* of a class. In VB.NET terms, this means non-Shared variables and Shared & non-Shared methods. Thus, Shared variables (class variables) are not taken into account. On the contrary, all method calls are taken into account, whether Shared or not. This distinction does not seem to make any sense, but we follow the original definition.

If a call is polymorphic in that it is to an Interface method in .NET, this is not taken as a coupling to either the Interface or the classes that implement the interface. If a call is polymorphic in that it is to a method defined in a VB Classic interface class (base class), it's a coupling to the interface class, but not to any classes that implement the interface. This is a limitation of the implementation, not the definition of CBO.

In this implementation of CBO, when a child class calls its own inherited methods, it is coupled to the parent class where the methods are defined. The original CBO definition does not define if inheritance should be treated in any specific way. Therefore, we follow the definition and treat inheritance as if it was regular coupling.

Readings

- Shyam R. Chidamber, Chris F. Kemerer. A Metrics suite for Object Oriented design. M.I.T. Sloan School of Management E53-315. 1993. http://uweb.txstate.edu/~mg43/CS5391/Papers/Metrics/OOMetrics.pdf
- Victor Basili, Lionel Briand and Walcelio Melo. A Validation of Object-Oriented Design Metrics as Quality Indicators. IEEE Transactions on Software Engineering. Vol. 22, No. 10, October 1996. http://www.cs.umd.edu/users/basili/publications/journals/J60.pdf
- Roger Whitney: Course material. CS 696: Advanced OO. Docs 6 & 8, Metrics. Spring Semester, 1997. San Diego State University. http://www.eli.sdsu.edu/courses/spring97/cs696/notes/metrics/metrics.html and http://www.eli.sdsu.edu/courses/spring97/cs696/notes/metrics2/metrics2.html
- Lionel C. Briand, John W. Daly, and Jürgen Wüst: A Unified Framework for Coupling Measurement in Object-Oriented Systems. Fraunhofer Institute for Experimental Software Engineering. Kaiserslautern, Germany. 1996. http://www.iese.fraunhofer.de/network/ISERN/pub/technical_reports/isern-96-14.pdf
- Houari A. Sahraoui, Robert Godin, Thierry Miceli: 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

RFC and RFC´ Response for a 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. Testing and debugging is complicated. A worst case value for possible responses will assist in appropriate allocation of testing time.

> A study of 30 C++ projects suggests that an increase in RFC increases the density of bugs and decreases quality. The study suggests "optimal" use for RFC but doesn't tell what the optimum is. It sounds safe to assume that a high RFC is detrimental in VB as well. *Misra & Bhavsar: Relationships Between Selected*

Software Measures and Latent Bug-Density: Guidelines for Improving Quality. Springer-Verlag 2003.

RFC is the original definition of the measure. It counts only the first level of calls outside of the class. RFC' measures the full response set, including methods called by the callers, recursively, until no new remote methods can be found. If the called method is polymorphic, all the possible remote methods executed are included in R and R'.

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 not 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 code executed.

> **Implementation details.** Project Analyzer calculates RFC and RFC´ from the procedure forward call tree. It regards all subs, functions, properties and API declares as methods, whether in classes or other modules. Calls to property Set, Let and Get are all counted separately. Calls to VB library functions, such as print, are not counted. Calls are counted to declared API procedures. If COM libraries were included in the analysis, calls to procedures in those libraries are also counted. The counting stops at the API call or COM procedure, no recursion is done because the callees of an API or COM procedure are unknown. Dead methods and their callees are included in the measure. Although they're not executed at the moment, they could become live by a change in the code.
>
> **Limitation of implementation.** RFC doesn't include calls via RaiseEvent. RFC' does.

Readings

- Shyam R. Chidamber, Chris F. Kemerer. A Metrics suite for Object Oriented design. M.I.T. Sloan School of Management E53-315. 1993. http://uweb.txstate.edu/~mq43/CS5391/Papers/Metrics/OOMetrics.pdf
- Victor Basili, Lionel Briand and Walcelio Melo. A Validation of Object-Oriented Design Metrics as Quality Indicators. IEEE Transactions on Software Engineering. Vol. 22, No. 10, October 1996. http://www.cs.umd.edu/users/basili/publications/journals/J60.pdf
- Roger Whitney: Course material. CS 696: Advanced OO. Docs 6 & 8, Metrics. Spring Semester, 1997. San Diego State University. http://www.eli.sdsu.edu/courses/spring97/cs696/notes/metrics/metrics.html and http://www.eli.sdsu.edu/courses/spring97/cs696/notes/metrics2/metrics2.html
- Lionel C. Briand, John W. Daly, and Jürgen Wüst: A Unified Framework for Coupling Measurement in Object-Oriented Systems. Fraunhofer Institute for Experimental Software Engineering. Kaiserslautern, Germany. 1996. http://www.iese.fraunhofer.de/network/ISERN/pub/technical_reports/isern-96-14.pdf

LCOM1 Lack of Cohesion of Methods

The 6th metric in the Chidamber & Kemerer metrics suite is LCOM (or LOCOM), the lack of cohesion of methods. This metric has received a great deal of critique and several alternatives have been developed. In Project Metrics we call the original Chidamber & Kemerer metric LCOM1 to distinguish it from the alternatives. LCOM1 is described among the other cohesion metrics.

Reference values

A study by NASA reports the following average values for Chidamber & Kemerer metrics. The study analyzed 3 systems and classified their quality.

System analyzed	Java	Java	C++
Classes	46	1000	1617
Lines	50,000	300,000	500,000
Quality	"Low"	"High"	"Medium"

CBO	2.48	1.25	2.09
LCOM1	447.65	78.34	113.94
RFC	80.39	43.84	28.60
NOC	0.07	0.35	0.39
DIT	0.37	0.97	1.02
WMC	45.7	11.10	23.97

The figures suggest that the higher the CBO and WMC, the lower the quality of the system. This seems to hold for LCOM1 as well, but note the shortcomings of LCOM1 discussed above.

NASA study

- Laing, Victor & Coleman, Charles: Principal Components of Orthogonal Object-Oriented Metrics. White Paper Analyzing Results of NASA Object-Oriented Data. SATC, NASA, 2001. http://satc.gsfc.nasa.gov/support/OSMASAS_SEP01/Principal_Components_of_Orthogonal_Object_Oriented_Metrics.pdf Comment: The writers suggest RFC=WMC+CBO, even though they seem to mean linear regression like RFC=β_{WMC} WMC + β_{CBO} CBO + c. You cannot sum WMC+CBO as WMC counts methods and CBO counts classes.

See also Object-oriented metrics