Exploring the Cyclomatic Complexity's Relevance Today

Topic: Cyclomatic Complexity (CC) is commonly considered in modules on testing the validity of code design today. However, in your opinion, should it be? Does it remain relevant today? Specific to the focus of this module, is it relevant in our quest to develop secure software? Justify all opinions which support your argument and share your responses with your team.

Q: Is Cyclomatic complexity sufficient to ensure valid code design?

Yes, I believe that CC is a valid metric to test the validity of code design. In my opinion, CC helps identify the complexity of code paths and the "cleanness" of their software design choices. Such identification aids developers to adhere to software patterns that reduce path complexity and cognitive overload per module—Single Responsibility and Separation of Concerns. I shall say both Single Responsibility and Separation of Concerns are primary principles for the trend towards micro-services, as each micro-service maintains ownership over its data.

Selvarani et al. (2009) find a correlation between a system's design qualities and its likelihood for defects. Minimising such defects requires development and testing teams to understand the number of paths through a program. Such knowledge is time-consuming to obtain, given the dynamic nature of ever-evolving codebases. For this reason, cyclomatic complexity aims to identify these logical pathways through a program and thus rank a module accordingly. There are many metrics available to determine code quality as outlined by Butler and McCabe (2021):

Table 1. Software metric descriptions.

Metric	Acronym	Description	Author
Number of public methods	NPM	A count of public methods in a class	Bansiya & Davis
Data access metric	DAM	The ratio of private attributes to the total attributes declared in a class	Bansiya & Davis
Measure of aggregation	MOA	A count of the number of class fields who types are user defined classes; measures the part-whole relationship	Bansiya & Davis
Measure of functional abstraction	MFA	The ratio of the number of methods inherited by a class to the total number of methods accessible by the member methods on the class	Bansiya & Davis
Cohesion among methods of class	CAM	The summation of the number of different types of method parameters in every method divided by a multiplication of the number of different method parameter types in whole class and number of methods; the relatedness of methods of a class based on the parameter list of the methods	Bansiya & Davis
Weighted methods per class	WMC	The number of methods in the class	Chidamber & Kemerer
Depth of inheritance tree	WIT	The inheritance levels form the top object hierarchy top	Chidamber & Kemerer
Number of children	NOC	The number of immediate descendants of the class	Chidamber & Kemerer
Coupling between object classes	CBO	The number of classes coupled to a given class as a result of method calls, field accesses, inheritance, method arguments, return types, and exceptions	Chidamber & Kemerer
Response for a class	RFC	The number of different methods that can be executed when an object of that class receives a message	Chidamber & Kemerer
Lack of cohesion of methods	LCOM	The sets of methods a class that are not related through sharing of some of the class fields	Chidamber & Kemerer
Inheritance coupling	IC	The number of parent classes to which a given class is coupled; coupling occurs when 1) an inherited method uses an attribute that is defined in a new or redefined method, 2) one of its inherited methods calls a redefined method or 3) one of its inherited methods is called by a redefined method and uses a parameter that is defined in the redefined model	Tang, Kao & Chen
Coupling between methods	CBM	The total number of new or redefined methods to which all the inherited methods are coupled	Tang, Kao & Chen
Average method complexity	AMC	The average method size for each class; size is equal to the number of Java binary codes in the method	Tang, Kao & Chen

DOI: 10.4236/jsea.2021.141004 46 Journal of Software Engineering and Applications

Q: Is Cyclomatic Complexity valid for today's developers?

Yes. A large swathe of developers today use the Microsoft Visual Studio environment to develop software systems. This integrated development environment (IDE) has built-in support to calculate cyclomatic complexity for a given module or collection of modules (Microsoft, 2021). Watson and McCabe (1996) define cyclomatic complexity as "the amount of decision logic in a source code function" and recommend a CC limit of around 10.

Therefore after calculating a CC limit for each function within a codebase, developers can use the CC score to focus on high-risk functions. Thus, CC is a valuable tool that helps improve code design and maintainability by reducing logic complexity.

Q: Is Cyclomatic Complexity relevant in developing secure software?

CC is indeed relevant in developing secure software because it is imperative to detect and resolve software vulnerabilities as early as possible in the software development lifecycle. McGraw (2004) states that complexity is the enemy of software security, an idea that was investigated further by Moshtari et al. (2013). In their paper, they used McCabe's CC metric to determine unit complexity across various projects. They concluded that "complexity metrics are good predictors of vulnerability between different releases of a project".

Sin and Williams (2008) investigated the relationship between software complexity and secure software to determine whether or not complex code is less secure. In their evaluation, they referenced McCabe's CC metric. They found—in line with Moshtari et al. (2013)—that "fault prediction models that use complexity metrics also might be useful for vulnerability prediction in general." They conclude that there is little correlation that software complexity is the enemy of software security and that vulnerable code tends to be complex.

Therefore, I consider the CC metrics a valuable tool that developers can leverage to improve and better secure each new release of their system.

References

Butler, C.W. & McCabe, T.J. (2021) Cyclomatic Complexity-Based Encapsulation, Data Hiding, and Separation of Concerns. *Journal of Software Engineering and Applications*, *14*(1): 44-66.

Microsoft (2021) Code metrics – Cyclomatic complexity. Available from https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-cyclomatic-complexity?view=vs-2019 [Accessed 28 Sep. 2021]

McGraw, G. (2004) Software security. *IEEE Security & Privacy*, *2*(2):80-83.

McGraw, G. (2016). Four software security findings. *Computer*, *49*(1):84-87.

Moshtari, S., Sami, A. & Azimi, M. (2013). Using complexity metrics to improve software security. *Computer Fraud & Security, 2013(5)*:8-17. DOI: https://doi.org/10.1016/S1361-3723(13)70045-9

Shin, Y. & Williams, L. (2008). Is complexity really the enemy of software security?. *Proceedings of the 4th ACM workshop on Quality of protection*:47-50.

Watson, A. H., & McCabe, T. J. (1996). Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric (NIST Special Publication 500-235). Available from http://www.mccabe.com/pdf/mccabe-nist235r.pdf [Accessed 28 Sep. 2021]

Selvarani, R., Nair, T.R.G. & Prasad, V.K. (2009) Estimation of Defect Proneness Using Design Complexity Measurements in Object-Oriented Software. International Conference on Signal Processing Systems: 766-770. DOI: https://doi.org/10.1109/ICSPS.2009.163