Portfolio Component: Exploring the Cyclomatic Complexity's Relevance Today

The Cyclomatic Complexity 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 appropriate to our quest to develop secure software? Justify all opinions that support your argument and share your responses with your team.

Teamwork

You can share your team responses with the tutor for formative feedback or discuss them in next week's seminar.

Learning Outcomes

- Identify and manage security risks as part of a software development project.
- Critically analyse development problems and determine appropriate methodologies, tools and techniques (including program design and development) to solve them.
- Systematically develop and implement the skills required to be an influential development team member in a virtual professional environment, adopting real-life perspectives on team roles and organisation.

My answers:

The measurement of the number of paths through code, known as Cyclomatic complexity (CC), remains controversial within the industry despite its widespread use. While academic research may not always consider practical applications, the industry values straightforward and actionable metrics. In their article, Ebert & Cain (2016) contend that CC has significant value as a metric. Ferrer et al. (2013) pointed out that measuring time and cost in software development is essential; using metrics is necessary for management control. For instance, assessing the complexity of software components through CC analysis can effectively forecast high defect rates and testing

challenges. CC is a simple yet effective metric that can offer insights into enhancing the overall quality of code. It can also closely align with the number of test cases required for path coverage, ultimately providing immense value to developers and project managers alike (Ferrer et al. 2013).

Software complexity can be directly evaluated through software attributes rather than relying on indirect factors such as project milestones or system failures. Various measures are available, ranging from straightforward to more specialised. Metrics should be independent of implementation details such as source code formatting and language syntax to ensure consistency across diverse projects and programming languages. Cyclomatic complexity satisfies this requirement by gauging decision logic within a function, irrespective of text formatting or programming language. Ideally, complexity measures should define software prone to errors and difficult to comprehend and provide strategies for managing it (Watson et al. 1996).

However, Shepperd (1988) states as more modules are added to a program, its complexity can increase. Conversely, removing redundant code can decrease complexity. However, the decision-counting metric used to measure complexity has faced theoretical objections and neglected other vital factors. Despite this, it can still help predict testing effort, error incidence, and code recall. Cyclomatic complexity, on the other hand, has limitations as a general complexity metric due to differing interpretations and a lack of an explicit model. Developing a complexity metric based on program properties is a challenging task, and it may be more effective to derive metrics from abstract notations and software design concepts. It's important to validate any complexity metric to ensure it is reliable. McCabe's metric doubts its usefulness, and constructing a complete model for a complexity metric is challenging given the non-formal nature of the real world.

In my opinion, Cyclomatic Complexity should not be viewed as a cure-all solution. While it can undoubtedly help flag possible issues, it is just one of many metrics to consider when striving for secure coding practices. The relevance of Cyclomatic Complexity today is debatable, with compelling arguments to be made for both sides. Nonetheless, I maintain that it remains a valuable metric for enhancing software quality, provided it is used with other metrics and its limitations are kept in mind.

In support of my objective to develop secure software, I believe cyclomatic complexity remains pertinent today. As Brainhub (N.D.) highlights, measuring cyclomatic complexity can enhance code quality, maintainability, and productivity. This helpful practice also facilitates defect reduction and improves testing procedures by identifying independent paths within the code. However, it is essential to note that excessive cyclomatic complexity can render code incomprehensible and difficult to test, as Brainhub (N.D.) cautions. This may indicate additional issues, such as unnecessary conditional logic or excessive nesting, which could increase the likelihood of defects.

References:

Ferrer, J., Chicano, F. & Alba, E. (2013). Estimating software testing complexity. *Information and Software Technology*, 55(12), pp.2125–2139. doi:https://doi.org/10.1016/j.infsof.2013.07.007.

Ebert, C. and Cain, J. (2016). Cyclomatic Complexity. *IEEE Software*, 33(6), pp.27–29. doi:https://doi.org/10.1109/ms.2016.147.

Watson, A.H., Wallace, D.R. & McCabe, T.J. (1996). *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric*. [online] *Google Books*. U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology. Available at: https://www.google.co.uk/books/edition/Structured_Testing/lysRzUZhc2QC?hl=en&gbpv=1&dq=Computer+Systems+Technology%2BA+Testing+Methodology+Using+the+Cyclomatic+Complexity+Metric&pg=PR10&printsec=frontcover [Accessed 9 Sep. 2023].

Shepperd, M. (1988). A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, 3(2), p.30. doi:https://doi.org/10.1049/sej.1988.0003.

Brainhub (N.D.). *Streamlining the Code: Pros and Cons of Cyclomatic Complexity*. [online] Available at: https://brainhub.eu/library/measuring-cyclomatic-complexity.