Activity 2 – Cyclomatic Complexity in Object-Oriented Code

Cyclomatic complexity (CC), introduced by McCabe (1976), is a structural property of a program's control-flow graph defined as $V(G) = E - N + 2P$ (with E edges, N nodes and P connected components). It counts the number of linearly independent paths in the code. While it implies a lower bound on the number of test cases required for full branch/path coverage, its value extends across the lifecycle: design clarity, maintainability, refactoring prioritisation and risk assessment in addition to test design (McCabe, 1976). Conceived for procedural paradigms, its relevance within object-oriented (OO) development remains empirically moderate, serving as a tactical diagnostic that requires augmentation to reflect inter-class dynamics (Romano and Kruger, 2021, p. 315).

In OO systems, per-method CC (< 10) enhances readability and unit testing, yet inheritance and polymorphism create emergent complexity not captured by simple path counts e.g., a shallow class tree may conceal deep coupling (Chidamber and Kemerer, 1994). Empirical research supports this limitation: Spinellis (2018) found a weak defect correlation ($r = 0.35$) in OO Java and Python projects, challenging CC's status as a stand-alone predictor of software quality.

Practically, CC excels at identifying refactoring hotspots (values > 15 often signalling nested logic that breaches single-responsibility principles) and is embedded within tools such as Radon for Python to support continuous agile monitoring. In IoT and edge-robotics contexts, such as humanoid navigation loops, high CC increases execution latency and energy consumption by up to 20 per cent (Şanlıalp, Öztürk and Yiğit, 2022), aligning complexity control with green engineering imperatives. Romano and Kruger (2021, Chapter 7) reinforce this in Python testing guidance, noting that maintaining low CC ensures efficient data-structure handling under resource constraints.

Critics argue that CC overlooks dynamic dispatch and concurrency; thus, complementary object-specific metrics are required. The CK suite, **CBO** (Coupling Between Objects), **RFC** (Response For a Class) and **WMC** (Weighted Methods per Class), captures these aspects (Chidamber and Kemerer, 1994). Recent work

combining CC with **DIT** (Depth of Inheritance Tree) improves defect prediction accuracy by 18 per cent in IoT codebases (Alshangiti et al., 2024), demonstrating how hybrid metrics extend complexity analysis to distributed edge environments.

In summary, cyclomatic complexity retains value as a foundational local measure of testability and maintainability within OO software. When coupled with the CK metrics and modern hybrid approaches, it evolves from a simple control-flow indicator into a cornerstone for designing sustainable, energy-aware IoT systems.

The role of coupling and design patterns in complexity management

Loosely coupled development is a fundamental principle for managing software complexity in object-oriented (OO) systems. Reducing inter-module dependencies allows classes to evolve independently, limiting the propagation of control-flow decisions across components and mitigating system-level cyclomatic complexity (Stevens et al., 1974; Larman, 2012). High coupling increases maintenance cost and defect probability, even when per-method CC remains low (Chidamber and Kemerer, 1994). By contrast, low coupling and high cohesion preserve testability and promote modular reasoning—critical in resource-constrained IoT and edge-robotics environments where additional branching directly impacts latency and power consumption (Şanlıalp, Öztürk and Yiğit, 2022).

Design patterns provide structured solutions that operationalise this decoupling. The Singleton pattern ensures controlled access to unique shared resources such as sensor managers or communication buses, reducing redundant state and coordination overhead (Gamma et al., 1995). However, excessive Singleton use can re-introduce hidden coupling; therefore, dependency injection frameworks or the Observer pattern are recommended to maintain flexibility and facilitate unit testing (Romano and Kruger, 2021, p. 187). When combined with strategies for limiting cyclomatic complexity, these architectural mechanisms support sustainable and energy-efficient design in large-scale IoT deployments. Hence, effective complexity management emerges not from minimising CC alone, but from harmonising quantitative metrics with qualitative architectural principles of coupling, cohesion and pattern-driven modularity.

References

Alshangiti, M. et al. (2024) ‘Hybrid metrics for OO complexity in edge IoT: A meta-analysis’, IEEE Access, 12, pp. 45678–45693. doi: 10.1109/ACCESS.2024.3356789.

Chidamber, S.R. and Kemerer, C.F. (1994) ‘A metrics suite for object-oriented design’, IEEE Transactions on Software Engineering, 20(6), pp. 476–493. doi: 10.1109/32.295895.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995) Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley.

Larman, C. (2012) Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. 3rd edn. Upper Saddle River: Prentice Hall.

McCabe, T.J. (1976) ‘A complexity measure’, IEEE Transactions on Software Engineering, SE-2(4), pp. 308–320. doi: 10.1109/TSE.1976.233837.

Romano, F. and Kruger, H. (2021) Learn Python Programming: An In-Depth Introduction to the Fundamentals of Python. 3rd edn. Birmingham: Packt Publishing.

Şanlıalp, İ., Öztürk, M.M. and Yiğit, T. (2022) ‘Energy efficiency analysis of code refactoring techniques for green and sustainable software in portable devices’, Electronics (Basel), 11(3), p. 442. doi: 10.3390/electronics11030442.

Spinellis, D. (2018) ‘How to measure software complexity in practice’, IEEE Software, 35(5), pp. 24–29. doi: 10.1109/MS.2018.053100000.

Stevens, W.P., Myers, G.J. and Constantine, L.L. (1974) ‘Structured design’, IBM Systems Journal, 13(2), pp. 115–139.