**Unit 9: Packaging and Testing**

# e-Portfolio Activities:

## *1. How relevant is the cyclomatic complexity in object oriented systems? Which alternative metrics do you consider to be more reflective of the complexity of a piece of code, in comparison to the number of independent paths through a program? Support your response using reference to the related academic literature.*

Cyclomatic complexity is an essential software metric used to quantify the complexity of a program by examining the number of linearly independent paths through its source code. It helps in assessing the program's maintainability, understandability, and potential error-proneness, providing significant insights into the complexity of control flow within a program (Gupta & Singhal, 2014). However, while cyclomatic complexity serves as a valuable tool for method-level control flow analysis, it is essential to recognize that it is not the sole indicator of software complexity and quality, particularly in object-oriented systems.

I would argue that cyclomatic complexity can be suited to very specific cases, particularly when evaluating parts of a system that don't depend on other object classes. Therefore, it can have utility when applied to methods that don't have any coupling, since this is one major issue with the cyclomatic approach (Gupta & Singhal, 2014). Critics argue that complexity changes with modularization of code, and if technology shifts from linear programming to OOP, the complexity will also change (Ahmad, 2012). The fundamental problem is that cyclomatic complexity lacks the calculation of coupling between object classes and only measures conditional statements, missing the core sources of OOP complexity: inheritance hierarchies, polymorphism, inter-class dependencies, and encapsulation violations.

Several alternative metrics offer a more holistic view of code complexity compared to just counting independent paths within a program. The Chidamber and Kemerer metrics suite provides OOP-specific alternatives—particularly CBO (Coupling Between Objects), DIT (Depth of Inheritance Tree), and RFC (Response for Class)—which better capture object-oriented complexity by measuring inter-class relationships, inheritance structures, and method interaction complexity (Chidamber & Kemerer, 1996). Beyond these quantitative metrics, qualitative factors such as readability, maintainability, testability, scalability, performance, robustness, and compatibility play a crucial role in assessing software quality alongside cyclomatic complexity. These qualitative aspects emphasize the human experience of code quality, focusing on simplicity, maintainability, and effectiveness over purely numerical metrics.

Therefore, while cyclomatic complexity remains a valuable metric for measuring software complexity, it should be used in conjunction with OOP-specific alternative metrics and qualitative assessments to provide a comprehensive evaluation of a piece of code's complexity and quality. This balanced approach, considering both quantitative and qualitative factors, enables developers to make informed decisions about code structure, testing strategies, and overall software maintainability in object-oriented systems.

**References:**

Ahmad, S. (2012) 'Cyclomatic Complexity for WCF: A Service Oriented Architecture', *ResearchGate*. Available at: https://www.researchgate.net/publication/261492648_Cyclomatic_Complexity_for_WCF_A_Service_Oriented_Architecture (Accessed: 25 September 2025).

Chidamber, S.R. and Kemerer, C.F. (1996) 'Chidamber and Kemerer's metrics suite: A measurement theory perspective', *ResearchGate*. Available at: https://www.researchgate.net/publication/3187794_Chidamber_and_Kemerer's_metrics_suite_A_measurement_theory_perspective (Accessed: 25 September 2025).

Gupta, V. and Singhal, P. (2014) 'An approach for improving the concept of Cyclomatic Complexity for Object-Oriented Programming', *arXiv preprint arXiv:1412.6216*. Available at: https://arxiv.org/abs/1412.6216 (Accessed: 25 September 2025).

## *2. To what extent is cyclomatic complexity relevant when developing objectoriented code?*

Cyclomatic complexity has limited relevance as a primary metric when developing object-oriented code. While it provides insights into method-level control flow complexity, it fails to capture the interrelationships between classes, inheritance hierarchies, and polymorphic behavior that define OOP systems (Gupta & Singhal, 2014).

During OOP development, developers must consider coupling between objects, which cyclomatic complexity entirely ignores (Gupta & Singhal, 2014). High cyclomatic complexity values within individual methods may indicate potential areas for refactoring, but this represents only a fraction of the complexity developers face in object-oriented design. Therefore, while cyclomatic complexity can serve as a supplementary metric for assessing method-level maintainability and testability, it should not be relied upon as a primary indicator of code quality in object-oriented systems. Instead, OOP-specific metrics such as CBO, DIT, and RFC provide more comprehensive insights into actual system complexity (Chidamber & Kemerer, 1996).

**References:**

Chidamber, S.R. and Kemerer, C.F. (1996) 'Chidamber and Kemerer's metrics suite: A measurement theory perspective', *ResearchGate*. Available at: https://www.researchgate.net/publication/3187794_Chidamber_and_Kemerer's_metrics_suite_A_measurement_theory_perspective (Accessed: 25 September 2025).

Gupta, V. and Singhal, P. (2014) 'An approach for improving the concept of Cyclomatic Complexity for Object-Oriented Programming', *arXiv preprint arXiv:1412.6216*. Available at: https://arxiv.org/abs/1412.6216 (Accessed: 25 September 2025).

***3. What is the cyclomatic complexity of the following piece of code?***

```
public static string IntroducePerson(string name, int age)
{     var response = $"Hi! My name is {name} and I'm {age} years old.";
     if (age >= 18)          response
+= " I'm an adult.";
     if (name.Length > 7)          response
+= " I have a long name.";
     return
response;
}
```

The method contains two decision points. The first conditional statement evaluating age >= 18, and the second evaluating name.Length > 7. Each conditional branch represents an independent decision that affects program flow. Starting with a base complexity of 1 for the method entry point, we add 1 for each decision point, resulting in a cyclomatic complexity of 3.

Although there are four possible execution paths through this method (all combinations of the two boolean conditions), cyclomatic complexity measures linearly independent paths, of which there are three:

**Path 1**: Both conditions false → returns base introduction only

**Path 2**: First condition true, second false → adds adult status

**Path 3**: Second condition true → adds long name notification

***4. Extend the following program to test accuracy of operations using the assert statement.***

```
# PythonStringOperations

str1 = 'Hello'
str2 ='World!'

# using + print('str1 + str2 = ',
str1 + str2)

# using * print('str1 * 3
=', str1 * 3)
```

Answer:

```
# Python String Operations
str1 = 'Hello'
str2 = 'World!'

# using +
concatenation = str1 + str2
print('str1 + str2 = ', result concatenation)
assert concatenation == 'HelloWorld!',"Concatenation result is
incorrect"

# using *
repetition = str1 * 3
print('str1 * 3 =', result repetition)
assert repetition == 'HelloHelloHello', "Repetition result is
incorrect"
```