Code Complexity, Test Strategy and Case Design

- featuring system testing in a Microservices architecture

Huang Luohua Locke, Software QA Engineer
- https://luohuahuang.org/about/

Topics

- Is COMPLETE Testing Possible?
- System Testing in a Microservices Architecture
- Path Testing (Structural Testing)
- Cyclomatic Complexity (McCabe’s Structural Metric)
- Derive Test Cases
- Case Study: Routine & Integration Complexity, and Correlation
- Reference

Is COMPLETE Testing Possible?

- Three different approaches can be used to demonstrate that a program is correct:
  - **Functional Testing**. Test based on business function
    - Verify that the correct outcome is produced for every input
  - **Structural Testing**. Test based on code structure
    - Verify that every path through the routine is exercised at least once
  - ~~Correctness Proofs~~. Test based on formal proofs of correctness
    - Build a mathematics model and use it to verify that the routine will produce the correct outcome for all possible input sequences. Applies only to numerical routines or crucial software such as system security kernel or compilers

System Testing in a Microservices Architecture

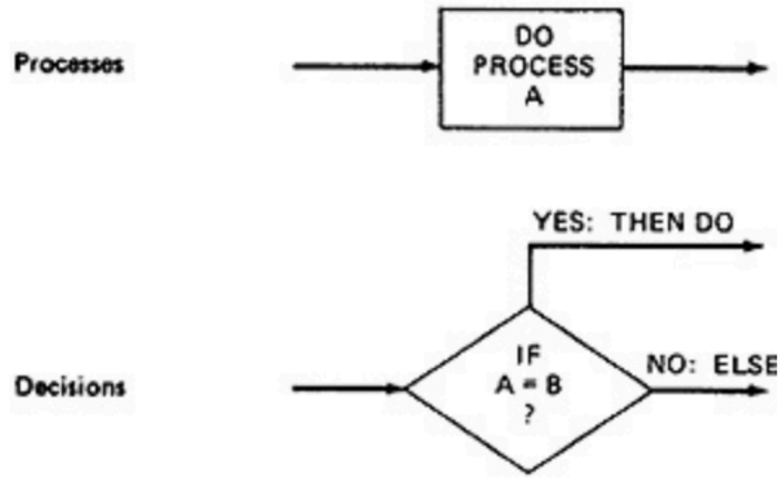
- System Testing (a.k.a APIs Testing) consists of
  - Parameter/input checks
  - Functional checks
- API Design/Test Principle: ~~Large Set of Unknown Developers (LSUDs)~~ VS. Small Set of Known Developers (SSKDs)
  - Typically SSKDs are in-house development
    - Share a common data format (input & output) interface: json & protobuf
- No more monolithic, code complexity is analysis-able/doable for each micro service

System Testing in a Microservices Architecture - Cont'd

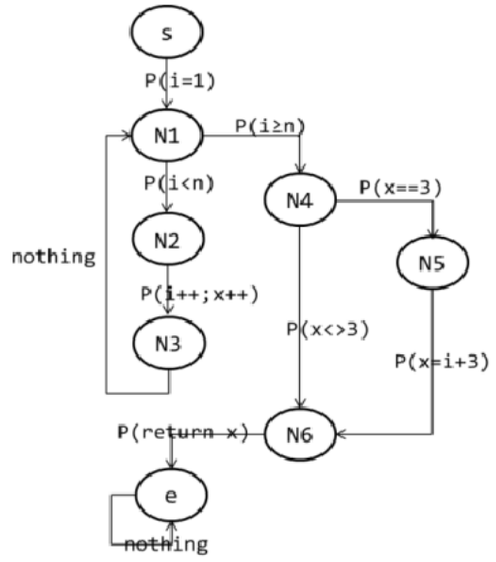
- System Testing Strategy
  - Testing based on SSKDs
  - Functional Testing which focus on design of input combination from business view to verify correct outcome is not efficient
  - Structural Testing which leverages code complexity analysis to conduct path testing is efficient
- Summary: choose Structural Testing to conduct system testing in a Microservices architecture

Path Testing (Structural Testing)

- Path Testing is about Process, Decisions and Case Statements, Junctions, Entry/Exit

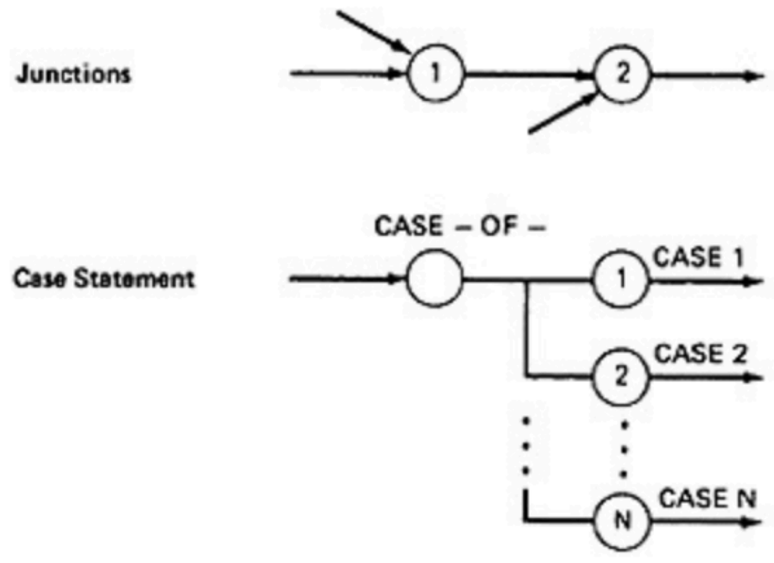



```
int foo(int x, int n) {
     int i=1;
     while (i<n) {
          i++;
          x++;
      }
      if (x==3) x=i+3;
      return x;
}
```



(a) (b)

Program Example and its Simplified Flowgraph Notation



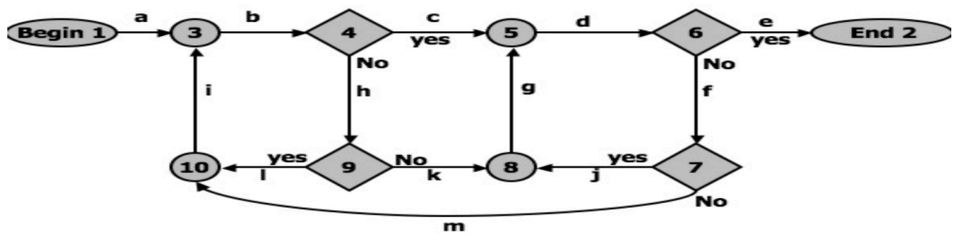
Flowgraph Elements

Path Testing - Cont’d

- Path Selection Criteria
    - Exercise every path from entry to exit
    - Exercise every statement or instruction at least once
    - Exercise every branch and case statement, in each direction, at least once
- Path Testing Criteria
    - Statement Testing (C1) - Execute all statements in the program at least once under some test
        - This is the weakest criterion. For any testing should ensure C1 is honoured
    - Branch Testing (C2) - Execute enough tests to assure that every branch alternative has been exercised at least once under some test

Path Testing - Cont'd

- Path Selection Example
  - Does every decision have a YES and a NO in its column? (C2)
  - Has every case of all case statements been marked? (C2)
  - Is every three - way branch (less, equal, greater) covered? (C2)
  - Is every link (process) covered at least once? (C1)



Simplified Flowgraph Notation

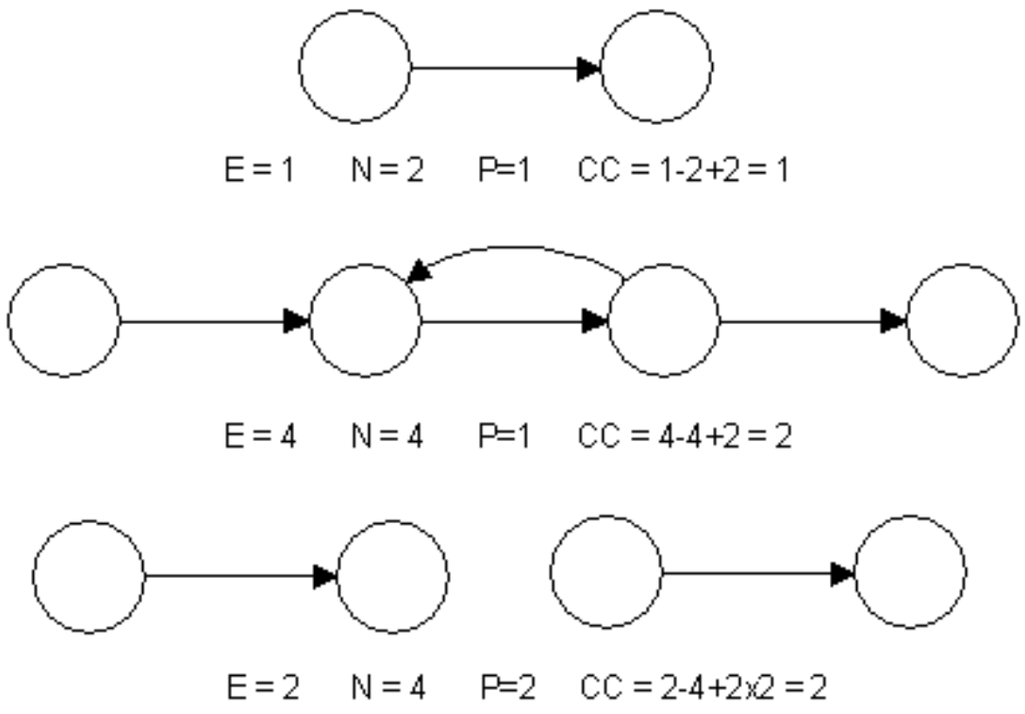
PATHS	DECISIONS				PROCESS—LINK												
	4	6	7	9	a	b	c	d	e	f	g	h	i	j	k	l	m
abcde	YES	YES			✓	✓	✓	✓	✓								
abhkgde	NO	YES		NO	✓	✓		✓	✓		✓	✓			✓		
abhlibcde	NO,YES	YES		YES	✓	✓	✓	✓	✓			✓	✓			✓	
abcdfjgde	YES	NO,YES	YES		✓	✓	✓	✓	✓	✓	✓			✓			
abcdfmibcde	YES	NO,YES	NO		✓	✓	✓	✓	✓	✓			✓				✓

Pathing Testing

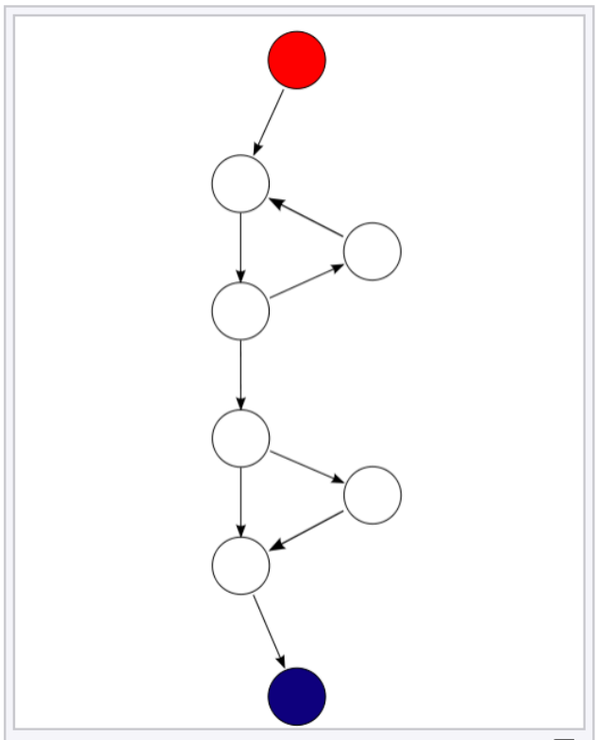
Cyclomatic Complexity (McCabe's Structural Metric)

- The cyclomatic complexity (CC) of a structured program[a] is defined with reference to the control flow graph of the program, a directed graph containing the basic blocks of the program, with an edge between two basic blocks if control may pass from the first to the second. The complexity **M** is then defined as
    - ***M* = *E* – *N* + 2*P***,
        - *E* = the number of edges of the graph
        - *N* = the number of nodes of the graph
        - *P* = the number of connected components
- CC is Comprehensibility, Testing Effort Estimation, Reliability
    - Quantifies the logical complexity
    - Measures the minimum effort for testing
    - Guides the testing process
    - Useful for finding sneak paths within the logic
    - Aids in verifying the integrity of control flow
    - Used to test the interactions between code constructs

Cyclomatic Complexity - Cont'd



CC Calculation 1



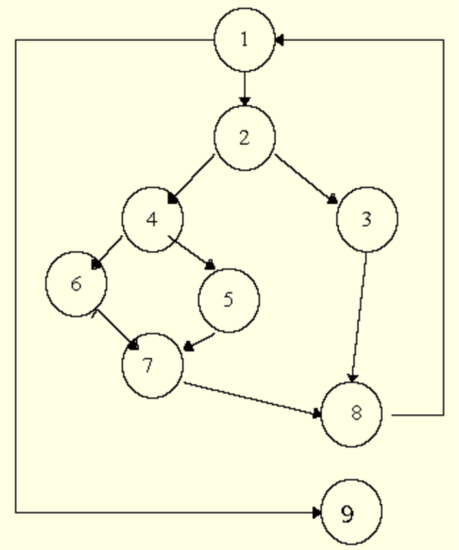
A control flow graph of a simple program. The program begins executing at the red node, then enters a loop (group of three nodes immediately below the red node). On exiting the loop, there is a conditional statement (group below the loop), and finally the program exits at the blue node. This graph has 9 edges, 8 nodes, and 1 connected component, so the cyclomatic complexity of the program is 9 - 8 + 2*1 = 3.

CC Calculation 2

Derive Test Case

- Cyclomatic Complexity provides upper bound for number of tests required to guarantee coverage of all program statements
    - Using the design or code, draw the corresponding flow graph.
    - Determine the Cyclomatic Complexity of the flow graph
    - Determine a basis set of independent paths
    - Prepare test cases that will force execution of each path in the basis set

```
1:      WHILE NOT EOF LOOP
2:         Read Record;
2:         IF field1 equals 0 THEN
3:            Add field1 to Total
3:            Increment Counter
4:         ELSE
4:             IF field2 equals 0 THEN
5:                 Print Total, Counter
5:                 Reset Counter
6:             ELSE
6:                 Subtract field2 from Total
7:             END IF
8:         END IF
8:         Print "End Record"
9:      END LOOP
9:      Print Counter
```



Example has:

- Independent Paths:
    1. 1, 9
    2. 1, 2, 3, 8, 1, 9
    3. 1, 2, 4, 5, 7, 8, 1, 9
    4. 1, 2, 4, 6, 7, 8, 1, 9
- Cyclomatic Complexity of 4; computed using any of these 3 formulas:
    1. #Edges - #Nodes + #terminal vertices (usually 2)
    2. #Predicate Nodes + 1
    3. Number of regions of flow graph.

M = E - N + 2P = 11 - 9 + 2 = 4

Derive Test Case - Cont’d

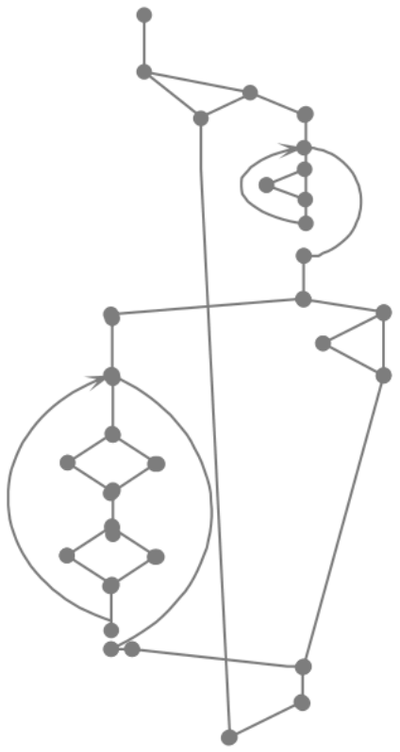
- Test Case Review. Count how many test cases are written. If the # of test cases is less than Cyclomatic Complexity,
    - You haven’t calculated the complexity correctly. Did you miss a decision?
    - Coverage is not really complete; there’s a link that hasn’t been covered
    - Coverage is complete, but it can be done with a few more but simpler paths
    - It might be possible for simplify the routine

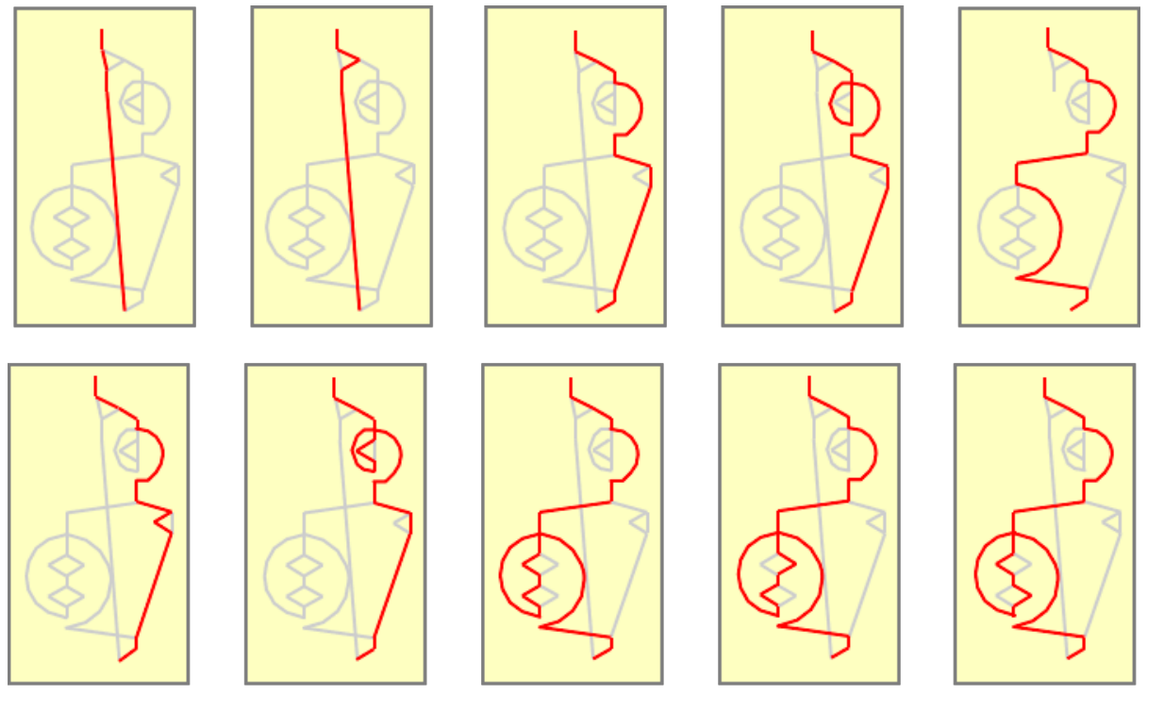
Case Study - Routine Complexity

- Complexity M = 10
  (*E – N + 2P, 39-31+2*1=10*)

  - Means that 10 Minimum test cases will:

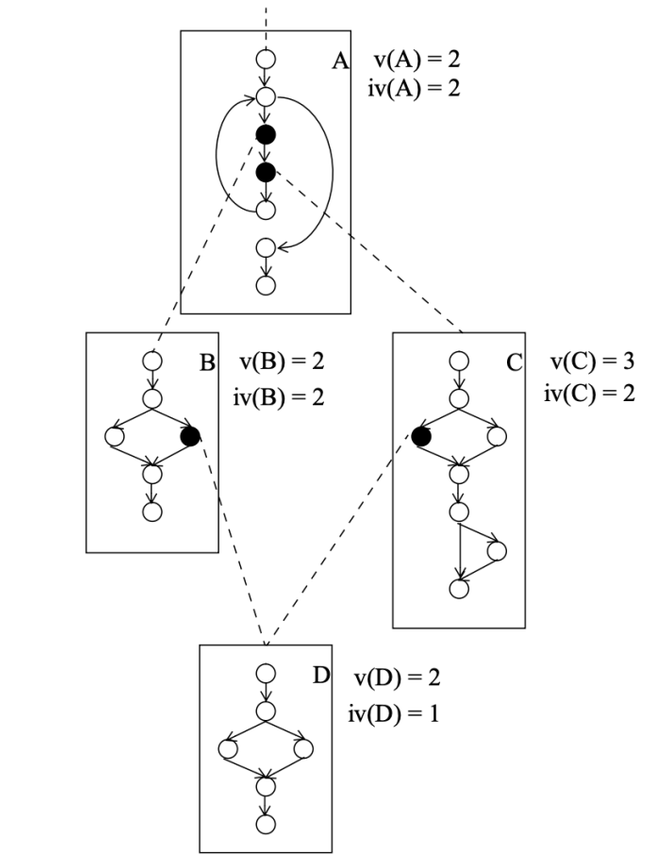
    - Cover all the code
    - Test decision logic
    - Test the interaction between code constructs





Case Study - Integration Complexity

- The integration complexity, S1, is defined for a program with n modules (G1 through Gn) by the formula:
  - ***S1 = (Σ iv(Gi)) - n + 1***
    - iv(Gi) is the M of Gi component
    - n is the # of components
- Integration complexity measures the number of independent integration tests through an entire program's design



Integration Complexity

Case Study - Correlation

- Complexity= Cyclomatic Complexity & Reliability Risk
  - 1 – 10: Simple procedure, little risk
  - 11- 20: More Complex, moderate risk
  - 21 – 50: Complex , high risk
  - >50: Untestable, VERY HIGH RISK
- Cyclomatic Complexity & Bad Fix Probability
  - 1 – 10: 5%
  - 20 –30: 20%
  - >50: 40%
  - Approaching 100: 60%
- Note: Some researchers who have studied the area question the validity of the methods used by the studies finding no correlation. E.g, https://www.researchgate.net/publication/237502126_Software_Metrics_and_Risk_FESMA_99_2nd_European_Software_Measurement_Conference_8_October_1999 - It uses a new approach, Bayesian nets for defects prediction

Reference

- Retrospect
    - Firstly it discusses there are three approaches ‘Functional’, ‘Structural’ and ‘Correctness Proofs’ to verify a system’s correctness
    - It introduces the features of system testing in a Microservices architecture and discusses why we choose ‘Structural’ (Path) testing
    - It introduces Path Testing in details and introduce how can we leverage Cyclomatic Complexity (McCabe metric) to conduct testing
    - At last It introduces how to derive test cases from Cyclomatic Complexity analysis and provides two case studies
- Reference

    - https://en.wikipedia.org/wiki/Cyclomatic_complexity
    - Software testing techniques (2nd ed.) - Boris Beizer: https://dl.acm.org/citation.cfm?id=79060
    - National Institute of Standards and Technology Special Publication 500-235: http://www.mccabe.com/pdf/mccabe-nist235r.pdf
    - Department of Homeland Security - Software Assurance Working Group http://www.mccabe.com/ppt/SoftwareQualityMetricsToIdentifyRisk.ppt
    - https://thenextweb.com/dd/2013/12/17/future-api-design-orchestration-layer/
    - https://www.researchgate.net/publication/237502126_Software_Metrics_and_Risk_FESMA_99_2nd_European_Software_Measurement_Conference_8_October_1999 - A new approach, using Bayesian nets for defects prediction
        - Linguistic Metrics: https://en.wikipedia.org/wiki/Halstead_complexity_measures