* The cyclomatic number of a *strongly connected directed graph* G = (V, E) is given by v(G) = |E| - |V| + p, where p is the number of strong components in the graph.
* v(G) is also equal to the number of bounded areas defined by the graph.

**e1**

**e3**

**e4**

**e5**

**e6**

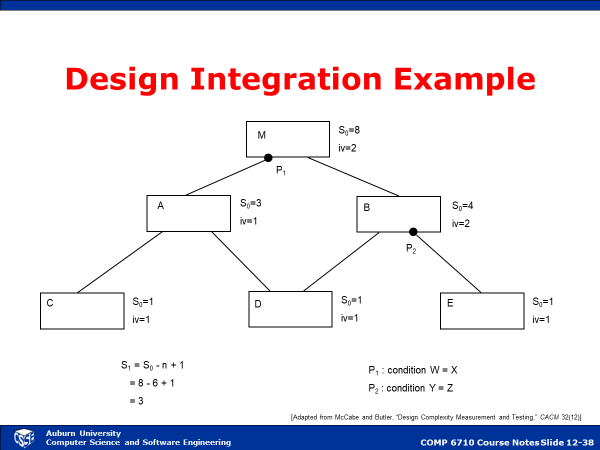
**e2**

**e7**

**e8**

**e9**

* A path is a sequence of instructions or statements from the entry point to the exit point.
* D-D
  + Partial paths
  + If all D-D paths are covered all nodes (statements) are covered
* **Statement coverage is the bare minimum**
* Basis Path
  + The cyclomatic number of the program graph gives the upper bound on the number of paths in the basis set.
  + Always start with the shortest path
* D =( c1 and c2 and c3) or not c1
* T1
  + X = 10
  + C1 = C2 = C3 = T
  + D = T
* T2
  + X = 0
  + C1 = C2 = C3 = F
  + D = T
* Achieved condition coverage but not decision/ branch coverage
* T3
  + X = 4
  + C1 = F, C2 = T, C3 = F
  + D = F
* Full Branch coverage now
* Grading Sys
  + T,L (Test & Lab)
  + Total(TTL) = .4T + .6L
  + To pass
    - T >= 60
    - L >= 60
* Program Slice must be executable
  + Don’t remove lines that limit usability
* No bias with Perform Structural testing
* Path coverage
  + Statement
  + IP – path
  + DU – path
  + P\*
* Fault-based testing
  + Mutatin testing
  + Operand vs operator
  + (a > = .5)
  + (a >= .45) (operand mutation)
  + (a > .5)
* Strong vs weak mutation
  + a >= .501 weak
  + a =< .5 strong
* False positive
  + Not a defect, but identified as one
* Test driver simulates a calling module
* Bottom-up Integration
  + The way the software was designed will influence the appropriateness of bottom-up integration.
  + While it is normally appropriate for object-oriented systems, bottom-up integration has disadvantages for functionally-decomposed systems:
  + Top-level components are usually the most important, but the last to be tested.
  + The upper levels are more general while the lower levels are more specific. Thus, by testing from the bottom up the discovery of major faults can be delayed.
  + Top-level faults are more likely to reflect design errors, which should obviously be discovered as soon as possible and are likely to have wide-ranging consequences.
  + In timing-based systems, the timing control is usually in the top-level components.
* Need simulators/Stubs needs both drivers and stubs
  + The correctness of the stub will influence the validity of the test.
  + Many stubs could be required, particularly when there are many general-purpose components in the lowest layer.
* Subtree a combination of modules that are exercised
* Integration testing
  + What combinations of modules to test
* Integration complexity
  + S1 = s0 – n + 1



* Complexity
  + Amt of info
    - # of var
      * Ia
    - # of input
      * Ib
    - # of class
      * Ic
  + Amt of comp
    - # of operator
      * Ca
    - # of operands
      * Cb
  + Amt of branching
    - # of cond
      * Ba

Comp (P) = WIa Ia + WIb Ib + WIc Ic + WCa Ca + WCb Cb + WBa Ba + …

Weights: importance W.R.T complexity

Use empirical data

* Lines of code (LOC)
  + - * Function Points (FP)
        + estimate of the SIZE of the functionality
        + Subjective, indirect measure
        + To be measured early in the life cycle (e.g. during requirements analysis), but can be measured at various points.
        + Measures the functionality of software, with the intent of estimating a project’s size (e.g., Total FP) and monitoring a project’s productivity (e.g., Cost per FP, FP per person-month)
        + Developed at IBM and rooted in classic information systems applications
        + Software Productivity Research, Inc. (SPR) developed a FP superset known as “Feature Points” to incorporate software that is high in algorithmic complexity but low in input/output.
        + A program’s FP metric is computed based on the program’s information domain and functionality complexity, with empirically-derived weighting factors.
        + Five factors

Inputs to the application

Outputs generated by the application

User inquiries

Data files to be accessed by the application

Interfaces to other applications

* + - * Reliability Metrics
      * Complexity Metrics
        + Halstead Metrics
        + McCabe Metrics
        + Complexity Profile Graph
      * Maintainability Index
        + 171 – 5.2\*ln(aveV) – 0.23\*aveV(g’) - 16.2\*ln(aveLOC) – 50\*sin(sqrt(2.4\*perCM))
        + aveV = average Halstead volume V per module
        + aveV(g’) = average cyclomatic complexity per module
        + aveLOC = average LOC per module
        + perCM = average percent of comment lines per module
      * Since the number of defects found in requirements and designs outnumber coding defects, leading companies and leading projects are very thorough in gathering requirements and in producing specifications.
      * As a result, formal inspections are among the most effective methods of defect prevention.
      * When formal inspections are added to the cycle, defect potentials gradually drop below 3.0 per function point while defect removal efficiency levels routinely top 95% and may hit 99%. This combination yields shorter development schedules and lower development costs than testing alone.
      * A more subtle problem with lines of code is that this metric penalizes high-level languages such as Java and Ruby and makes older low-level languages such as C and assembly language look better than they really are.
      * The “cost per defect” metric penalizes quality and tends to achieve the lowest result for the buggiest applications.
      * OOP
        + Encapsulation
        + Polymorphism/inheritance
      * Ck metrics
        + Weight methods per class (WMC)

Size of complexity of all methods of a given class

Use cyclomatic complexity to sum up

WMC should be kept low

Avg cyclomatic number is 10

* + - * + Depth of inheritance tree (DIT)

The length of the longest path from the root of the inheritance hierarchy to a leaf class.

* + - * + Number of children (NOC)

A count of the number of classes immediately subordinate to a given class in the hierarchy.

* + - * + Coupling between classes (CBO)

The amount of collaboration and interaction between a given class and the other classes in the system.

As the CBO value increases, reusability decreases.

Also, a high CBO indicates potential difficulty in modifying the class and the subsequent testing of the modifications.

CBO should be kept low.

* + - * + Response for a class (RFC)

The number of methods that can potentially be executed in response to a message received by an object of a given class.

As the RFC value increases, testing effort and design complexity also increase.

RFC should be kept low.

* + - * + Lack of cohesion in methods (LCOM)

The number of methods in each class that access one or more of the same instance variables.

The higher the LCOM value, the lower the cohesion of methods, and greater the coupling.

A high LCOM value could indicate the need to break the class apart into multiple classes.

* + - * + Class size (CS)

The total number of methods (both inherited and local) plus the total number of attributes (both inherited and local) encapsulated by a given class.

Inherited members should be weighted more heavily than local members.

Large values of CS could indicate that the class is too large; that is, that it encapsulates too much behavior, structure, and responsibility.

High CS values may also indicate lower reusability.

* + - * + Number of operations overridden by a subclass (NOO)

A count of the methods in subclasses that have been redefined.

Large NOO values could indicate a design problem, since the model of the class seems to be violated.

* + - * + Number of operations added by a subclass (NOA)

A count of the new methods appearing in subclasses.

A large NOA value could indicate a design abstraction violation.

As CK DIT increases, NOA should decrease.

* + - * + Specialization index (SI)

The degree to which subclasses are differentiated from superclasses.

SI is computed as NOO multiplied by the level at which the class resides in the inheritance hierarchy divided by the total number of methods defined by the class.

A high SI could indicate a lack of conformance to superclass abstractions.

SI = NOO \* depth / total # of methods