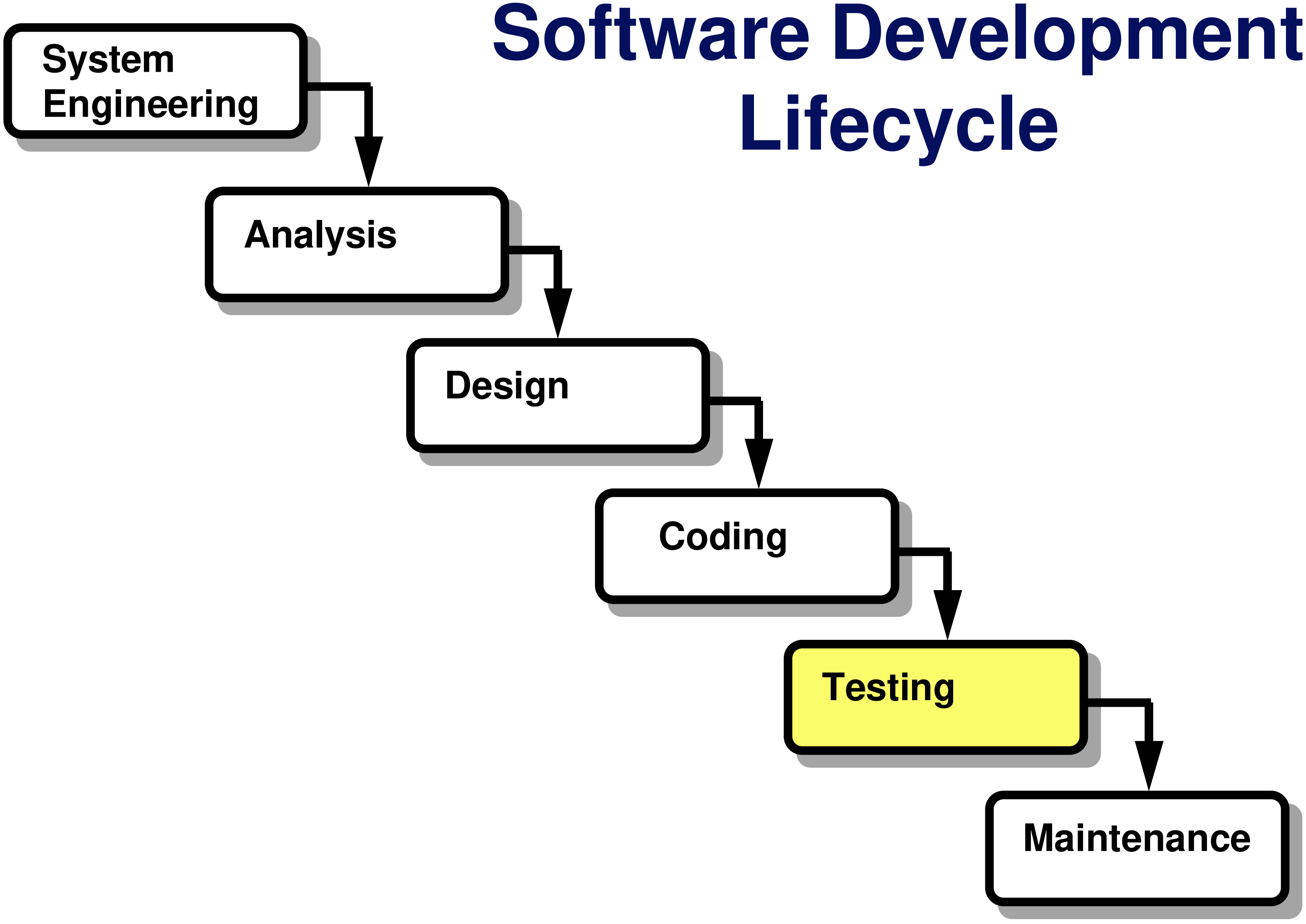
SOFTWARE TESTING

- Quality Assurance
  - Software Quality
  - Software Reviews
  - Software Quality Metrics
  - Formal SQA Approaches
  - Software Reliability
  - SQA Plan
- Testing Techniques
  - Black Box Testing
  - White Box Testing
- Testing Strategies
  - Unit Testing
  - Integration Testing
  - Validation Testing
  - System Testing
  - Debugging

Software Development Lifecycle


System Engineering

Analysis

Design

Coding

Testing

Maintenance

Software Quality Assurance

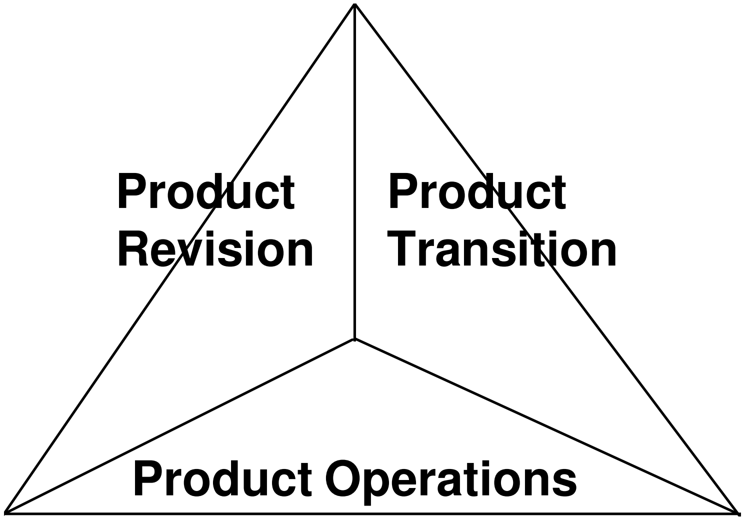
Conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software.

-- a definition of *Software Quality*, Pressman, Page 550

Software Quality Factors

- **Directly Measured**
  - **Errors**
  - **Lines of Code**
  - **Execution Time of Unit**
- **Indirectly Measured**
  - **Usability**
  - **Maintenance**

Software Quality Factors



Product Revision

Product Transition

Product Operations

McCall, J., P. Richards, and G. Walters, "Factors in Software Quality," three volumes, NTIS AD-A049-014, 015, 055, November 1977

Software Quality Checklists

Quality Factor	*Spec*	*Design*	*Impl*	*Test*	*Support*
Functionality					
Usability					
Reliability					
Performance					
Supportability					

Enter 0 (very poor) to 10 (outstanding) in each block to indentify quality

Grady, R.B., and D.L. Caswell, *Software Metrics: Establishing a Company-Wide Program*, Prentice-Hall, 1987

Software Quality Assurance (SQA)

- **SQA is a "planned and systematic pattern of actions" to ensure quality in software.**
- **SQA is essential for any business which produces software products used by others.**
- **The SQA group serves as an in-house representative of the customers.**

Software Reviews

Formal Technical Reviews (FTR)

- Uncover errors in function, logic, and implementation for any representation of the software
- Verify that software meets specifications
- Ensure that software conforms to standards
- Ensure that software is developed in a uniform manner
- Ensure that the project is manageable

Class of Reviews

- Code Walkthroughs
- Code Inspections
- Round-Robin Reviews
- Others

Formal Technical Review

Constraints

- 3-5 people in meeting -- developer, 2-3 reviewers, SQA representative, and recorder
- < 2 hours preparation time per person (pre-review before the meeting)
- < 2 hours for the meeting duration

During the Meeting

- Focus on a small, specific part of the software
- Review is initiated by SQA after the developer is done
- Developer talks through the product
- Recorder keeps notes on errors, issues, resolutions, and action items
- All attendees sign off on the team's findings

Software Quality Metrics

- **U.S. Air Force Systems Command Pamphlet 800-14: Design Structure Quality Index**
- **IEEE Standard 982.1-1988: Software Maturity Index**
- **Halstead's Software Science**
- **McCabe's Complexity Metric**

AFSCP 800-14 Design Structure Quality Index (DSQI)

Three Steps:

1. **Obtain specific information about the program (S1-S7)**
2. **Determine intermediate values (D1-D6)**
3. **Compute DSQI:**

$$DSQI = \sum w_i D_i$$

wi is the relative weight of Di

DSQI is used by comparing it with previous DSQI's. If much lower than expected, there is a need to do more design and review.

- **Based on database and data flow items**

IEEE Software Maturity Index (SMI)

M_T = # modules in current release
F_c = # modules in current release that have changed
F_a = # modules in current release that have been added
F_d = # modules from preceding release that were deleted in current release

$$SMI = \frac{M_T - (F_a + F_c + F_d)}{M_T}$$

As SMI approaches 1.0, the product is stabilizing.

- **Based on changes because of software updates**

Halstead Software Science

Given:

n_1 = # distinct operators in program
n_2 = # distinct operands in program
N_1 = # operator occurrences
N_2 = # operand occurrences

Program Length	$N = n_1 \log_2 n_1 + n_2 \log_2 n_2$	**statements**
Volume	$V = N \log_2 (n_1 + n_2)$	**bits to represent algorithm**
Volume Ratio	$L = \frac{2}{n_1} * \frac{n_2}{N_2}$	**min volume relative to actual volume possible**

Example of Halstead's Metrics

Program

```
      SUBROUTINE SORT (X, N)
      DIMENSION X(N)
      IF (N .LT. 2) RETURN
      DO 20 I=2,N
         DO 10 J=1,I
            IF (X(I) .GE. X(J)) GOTO 10
            SAVE = X(I)
            X(I) = X(J)
            X(J) = SAVE
10       CONTINUE
20    CONTINUE
      RETURN
      END
```

Example of Halstead's Metrics, Continued

Operators

	Operator	*Count*
1	End of statement	7
2	Array subscript	6
3	=	5
4	IF ()	2
5	DO	2
6	,	2
7	End of program	1
8	.LT.	1
9	.GE.	1
10	GOTO 10	1

n1 = 10

N1 = 28

Example of Halstead's Metrics, Continued

Operands

Operand		*Count*
1	X	6
2	I	5
3	J	4
4	N	2
5	2	2
6	SAVE	2
7	1	1

n2 = 7

N2 = 22

Example of Halstead’s Metrics, Continued

$$N = 10\log_2 10 + 7\log_2 7 = 52.871$$

$$V = \log_2(10+7) = 4.0875$$

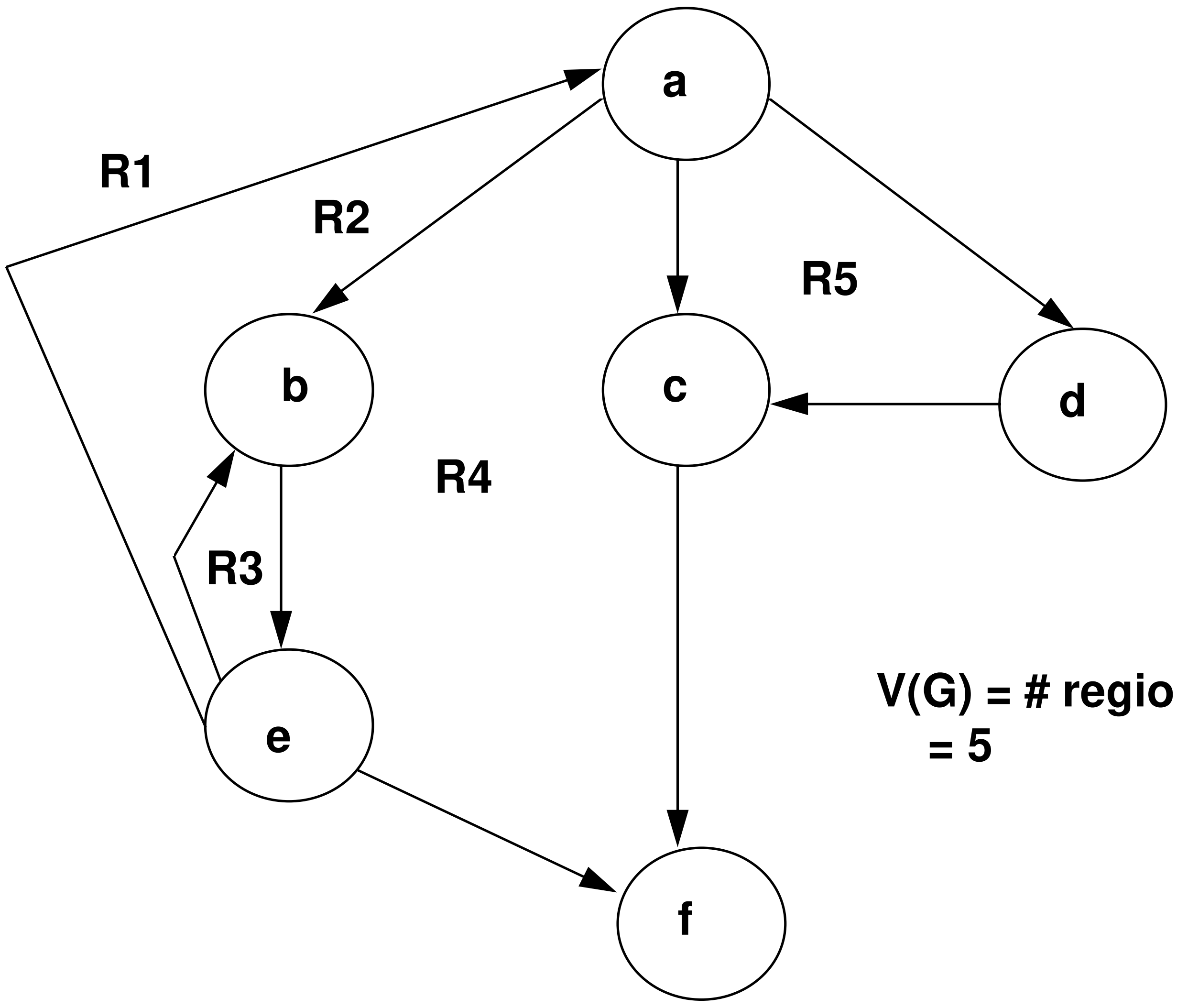
$$L = \left(\frac{2}{10}\right)\left(\frac{7}{22}\right) = \frac{14}{220} = 0.06364$$

McCabe’s Complexity Metric

- **Create program graph, G**
- **Determine cyclomatic complexity, V(G)**
- **Useful for estimating testing difficulty**

  V(G) > 10 indicates tough testing

Program Graph and V(G)

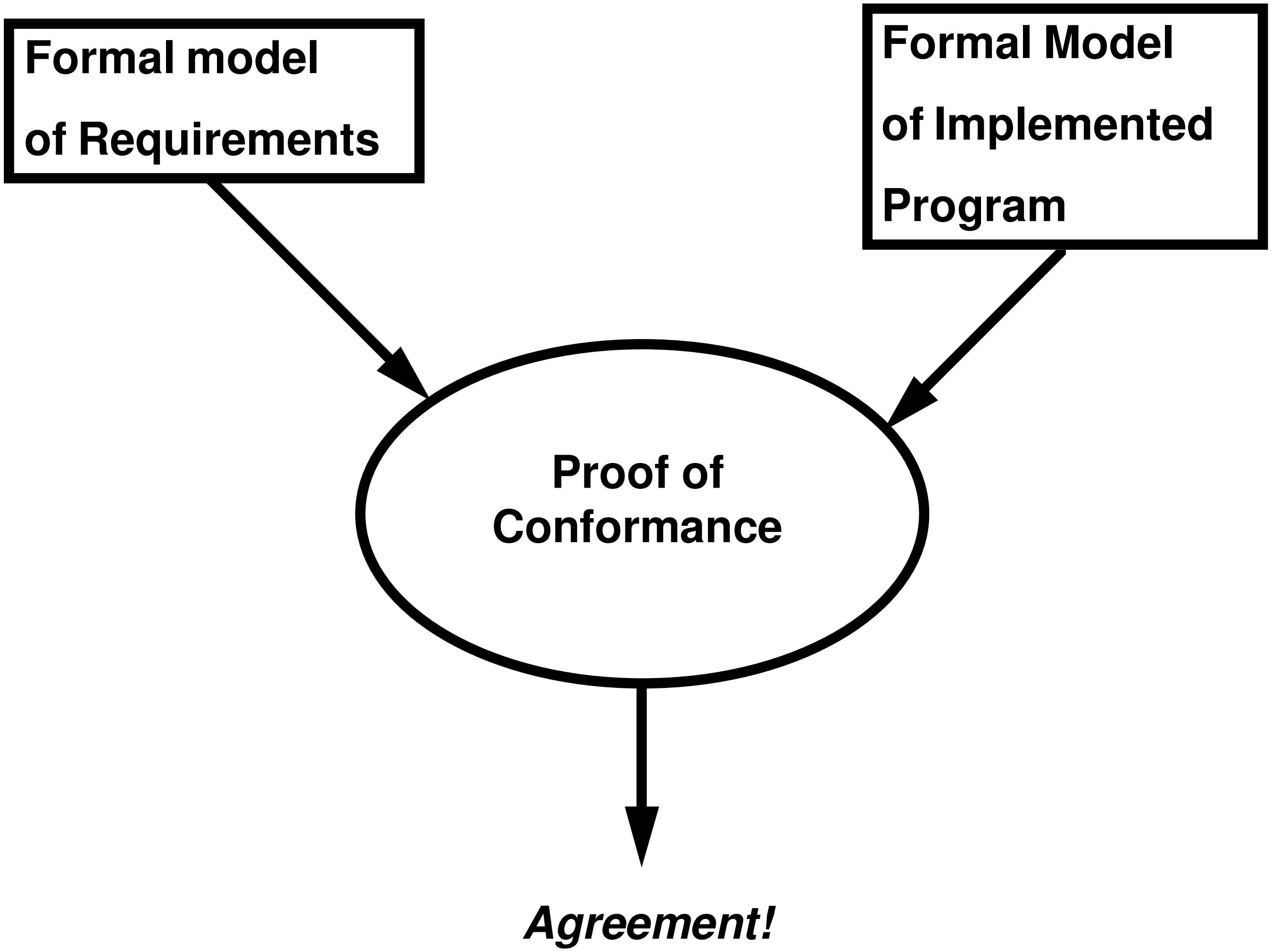

a
b
c
d
e
f
R1
R2
R3
R4
R5

V(G) = # regions in planar graph
= 5

Formal Approaches to SQA

1. **Proof of Correctness**
2. **Statistical Quality Assurance**
3. **Cleanroom Process**

Proof of Correctness



Formal model of Requirements

Formal Model of Implemented Program

Proof of Conformance

Agreement!

Proof of Correctness

```
Stmt    Code
1       procedure RANDOM (SEED : in FLOAT) return FLOAT is
2       begin
3           assert (SEED > 0 and SEED < MAX.FLOAT)
...         ...
n-2         assert (RESULT > 0.0 and RESULT < 1.0)
n-1         return RESULT;
n       end RANDOM;
```

Statistical Quality Assurance

1. **Software defect information is collected.**
2. **Trace each defect to its cause.**
3. **Identify the 20% "vital few" defects.**
4. **Correct the "vital few" defects.**

Data Collection for Statistical SQA

Example:

	Total		*Serious*		*Moderate*		*Minor*	
Error	*No.*	*%*	*No.*	*%*	*No.*	*%*	*No.*	*%*
IES	205	22	34	27	68	18	103	24
MCC	156	17	12	9	68	18	76	17
IDS	48	5	1	1	24	6	23	5
VPS	25	3	0	0	15	4	10	2
EDR	130	14	26	20	68	18	36	8
IMI	58	6	9	7	18	5	31	7
EDL	45	5	14	11	12	3	19	4
IET	95	10	12	9	35	9	48	11
other	180	19	20	16	71	18	89	20
Totals	942		128		379		435	

Defect Index

D_i = # defects uncovered in ith step of software engineering process

S_i = # serious defects

M_i = # moderate defects

T_i = # minor defects

PS = size of product (LOC, pages of doc, etc.)

W_j = weighting factor (j=1 for serious defect, 2 for moderate defect, 3 for minor defect)

$$PI_i = W1\left(\frac{S_i}{D_i}\right) + W2\left(\frac{M_i}{D_i}\right) + W3\left(\frac{T_i}{D_i}\right)$$

$$DI = \frac{\Sigma(i * PI_i)}{PS} = \frac{PI_1 + 2PI_2 + 3PI_3 + \ldots}{PS}$$

Cleanroom Software Engineering

- **Software developed under statistical quality control**
- **Goal is defect prevention rather than defect removal**
- **Proof of correctness is used to prevent defects**
- **Statistical QA used to certify the quality of the software**

- **Cleanroom approach has been shown to remove 90% of all defects prior to first tests**
- **General use of method would require substantial changes in management and technical approaches in industry**

Software Testing

1. Introduction
2. White Box Testing
3. Black Box Testing
4. Test Strategies

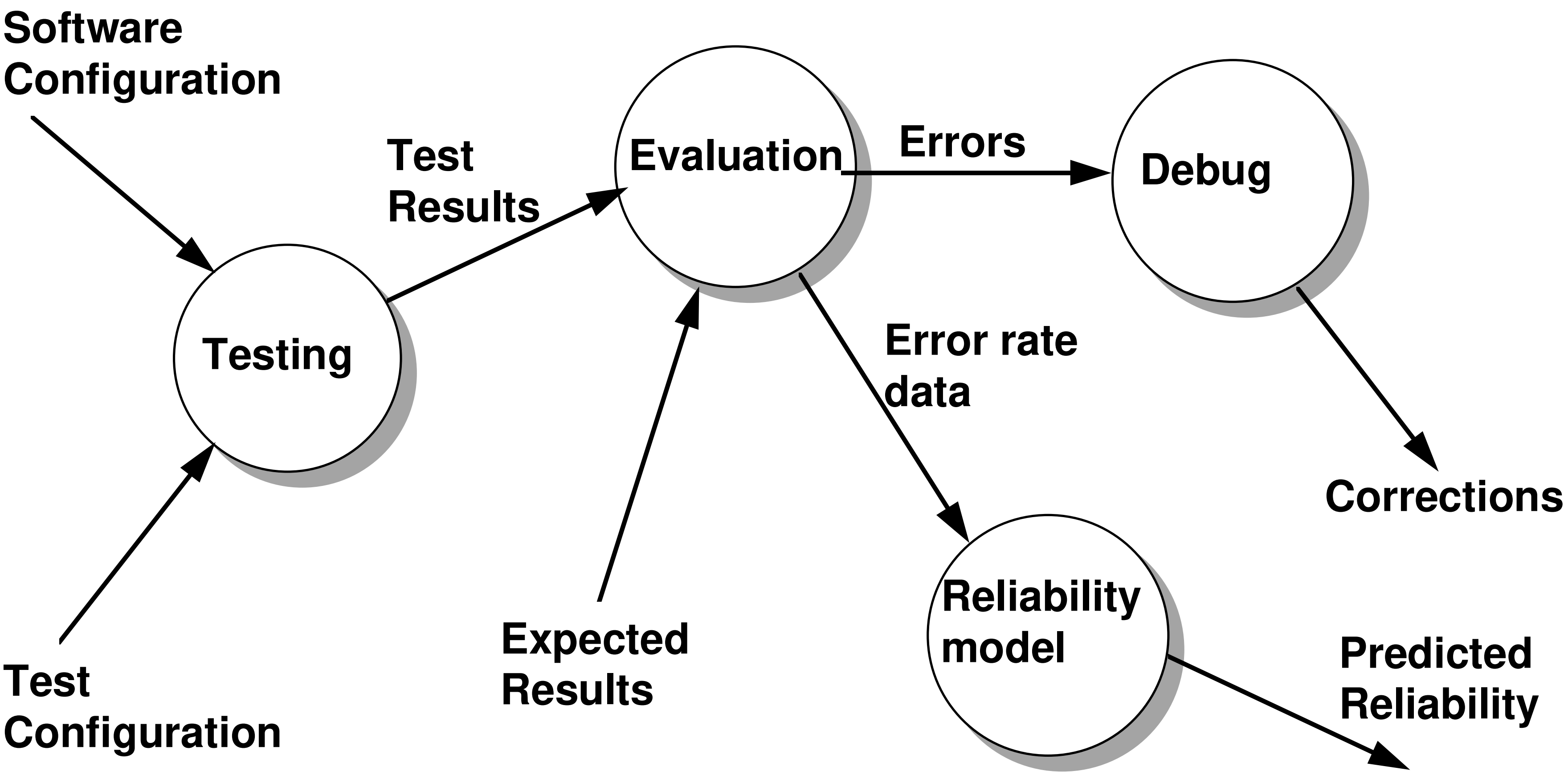
Software Fundamentals

Testing objectives

1. We test to find errors
2. A good test case has a high probability of finding an as yet undiscovered error
3. A successful test uncovers an as yet undiscovered error

Testing cannot show the absence of defects, it can only show that software defects are present.

Test Flow


Software Configuration
Test Results
Evaluation
Errors
Debug
Testing
Error rate data
Corrections
Reliability model
Expected Results
Test Configuration
Predicted Reliability

White and Black Box Testing

White Box Testing

Uses the control structure of the procedural design to derive test cases

1. Basis Path Testing
2. Control Structure Testing

Black Box Testing

Uses functional requirements including input/output relations to derive tests.

1. Equivalence Partitioning
2. Boundary Value Analysis
3. Cause-Effect Graphing Techniques
4. Comparison Testing

White Box Testing

1. White box tests exercise all
   - independent paths with a module at least once
   - logical decisions on their true and false sides
   - loops at their boundaries and within their operational bounds
   - internal data structures to ensure their validity

2. Why test as white box rather than black box (which is easier)?
   - Logic errors and incorrect assumptions are inversely proportional to the probability that a program path will be executed.
   - We often believe that a logical path is not likely to be executed when, in fact, it may be executed on a regular basis.
   - Typographical errors are random.

Basis Path Testing

Test derived from a basis set of execution paths.

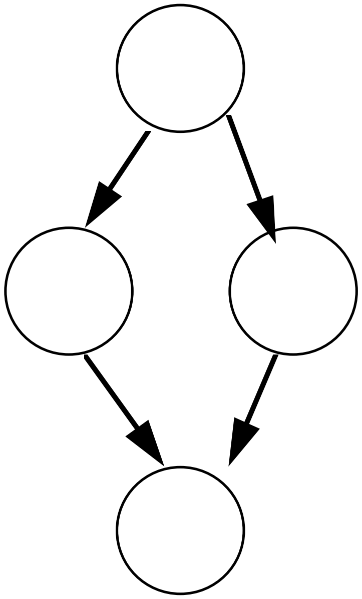
Cyclomatic number V(G) of the program graph is the upper bound of the size of the basis set.

The size of the basis set is the number of tests that must be designed and executed to guarantee coverage of all program statements.

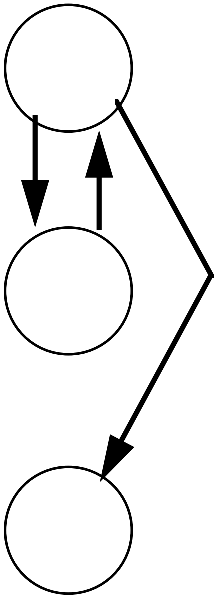
Procedure:

1. **Using the design or code as a foundation, draw a corresponding flow graph.**
2. **Determine the cyclomatic complexity of the resultant flow graph.**
3. **Determine a basis set of linearly independent paths**
4. **Prepare test cases that will force execution of each path in the basis set.**

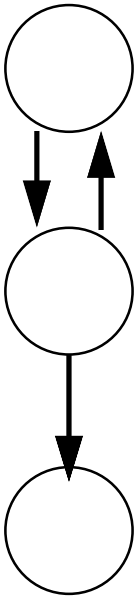
Creating Program Graphs



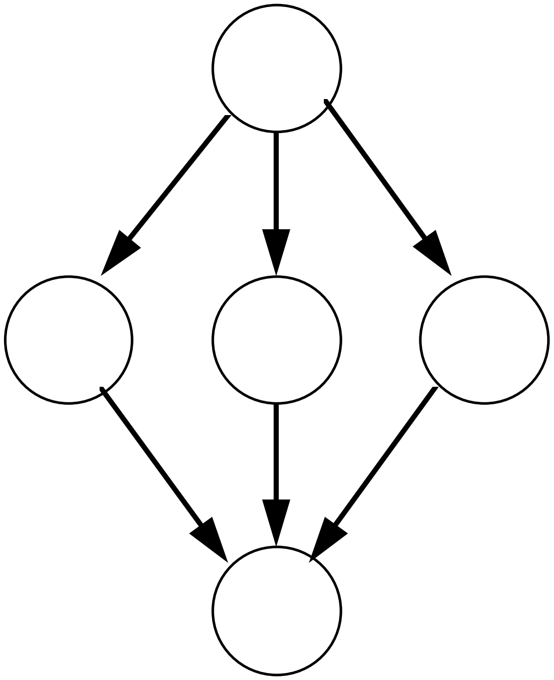
If



While

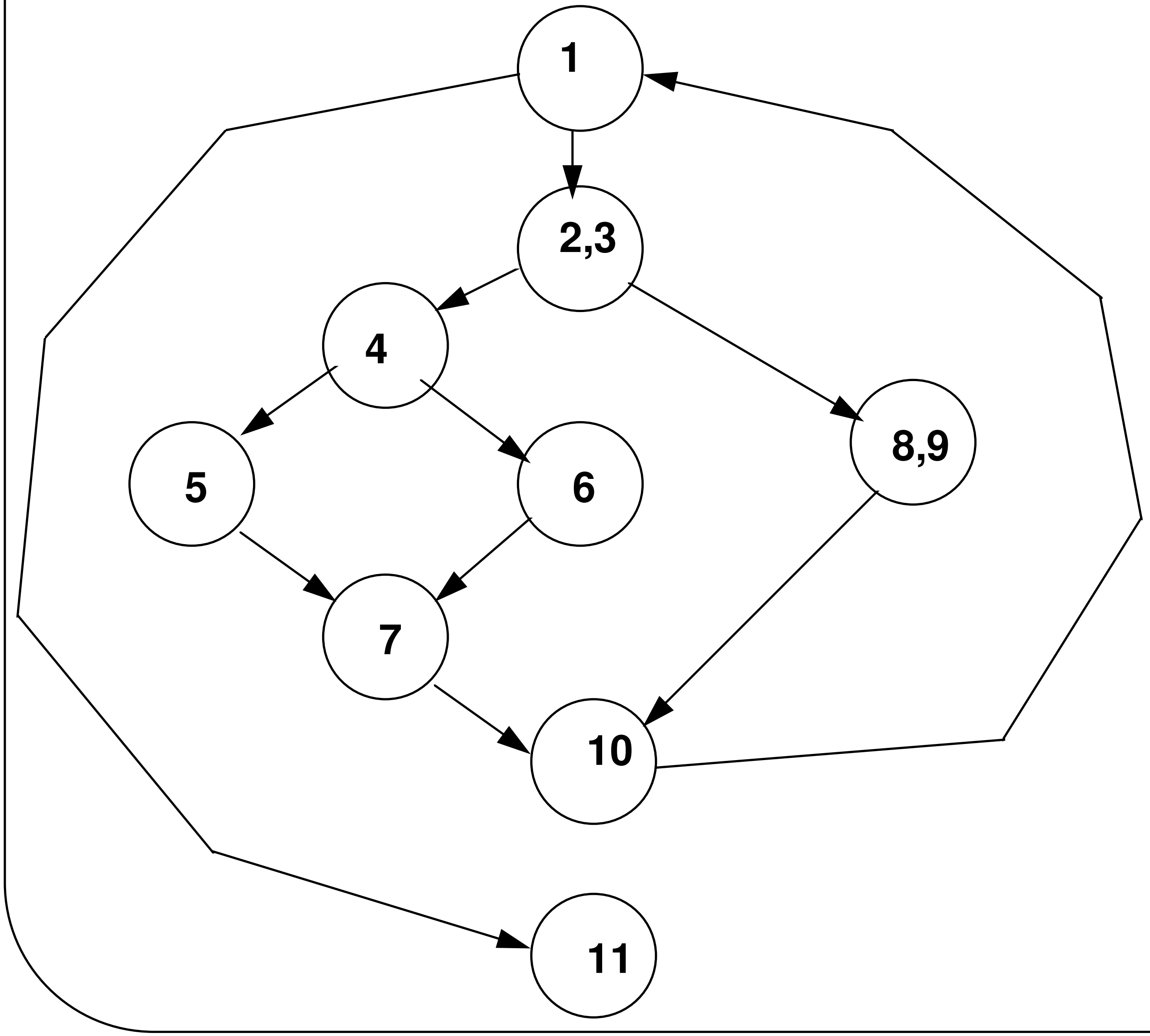


Until



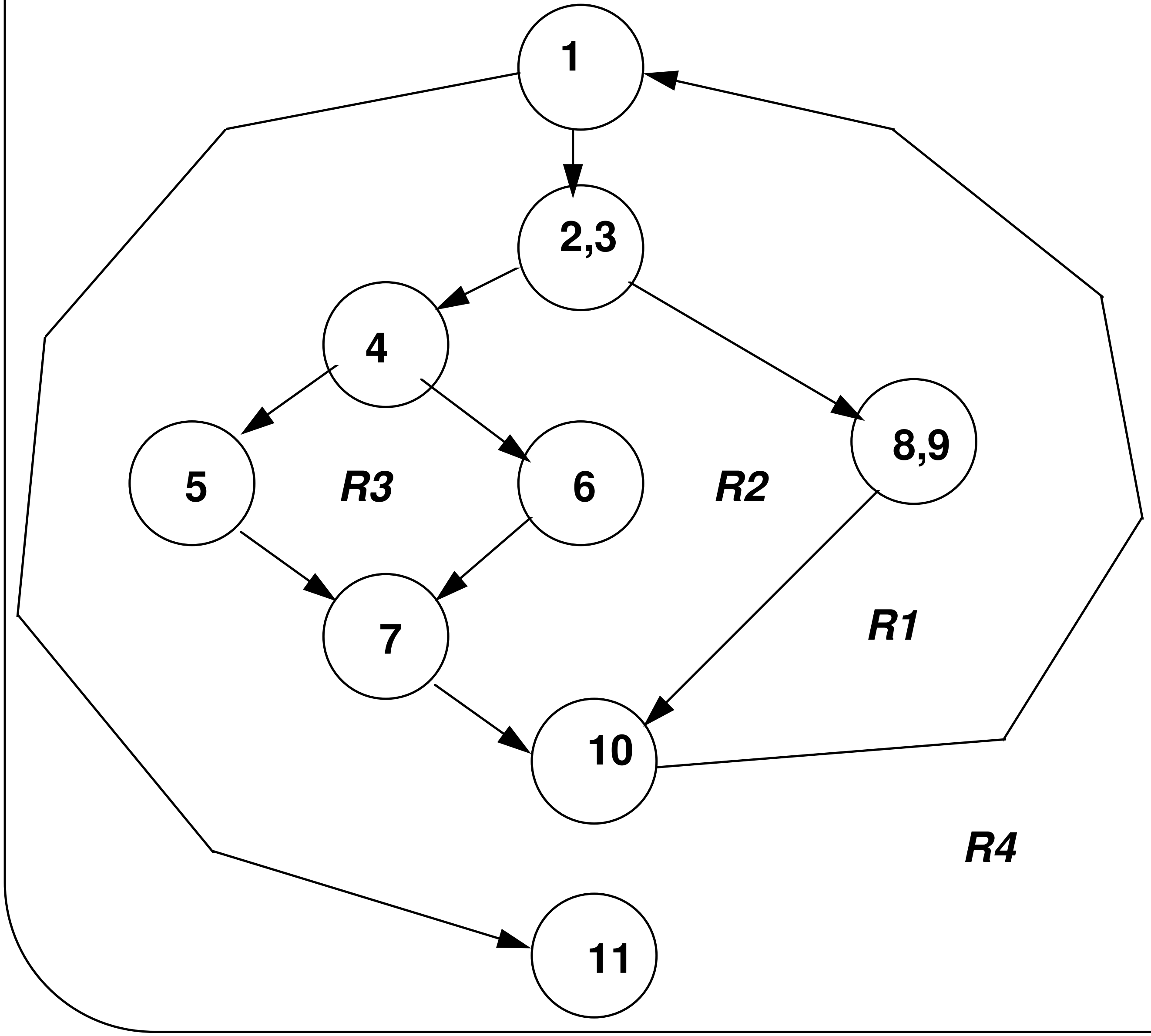
Case

Example Program Graph



```
1   while x>0
2         y:=1
3         if x>10 then
4,5            if z then x:=-2
6                     else x:=-3
7              end if
8         else y:=4
9                x:=-1
10        end if
    end loop
11 return
```

Deriving Independent Paths


1
2,3
4
5
6
7
8,9
10
11
R1
R2
R3
R4

Independent Paths

(basis set)

1. 1,11
2. 1,2,3,8,9,10,1,11
3. 1,2,3,4,6,7,10,1,11
4. 1,2,3,4,5,7,10,1,11

V(G) = E-N+2
V(G) = 4

Deriving Test Cases

Routine:

```
1    while x>0
2        y:=1
3        if x >10 then
4,5          if z then x:=-2
6                else x:=-3
7            end if
8        else y:=4
9            x:=-1
10       end if
     end loop
11   return
```

Tests:

Path 1,11

input: $x < 1$

output: unchanged x,y

Path 1,2,3,8,9,10,1,11

input: $x > 0$ and $x < 10$

output: y:=4, x:=-1

Path 1,2,3,4,6,7,10,1,11

input: $x > 10$ and z = false

output: y:=1, x:=-3

Path 1,2,3,4,5,7,10,1,11

input: $x > 10$ and z = true

output: y:=1, x:=-2

Control Structure Testing

The basis path testing technique previously described is one of a number of techniques for Control Structure Testing.

Basis path testing is simple and effective, but it is not sufficient in and of itself. Other variations on Control Structure Testing include:

- **Loop Testing**
- **Condition Testing**
- **Data Flow Testing**

Condition Testing

Condition Testing exercises the logical conditions contained in a program module.

A *simple condition* is a boolean variable or a relational expression, possibly preceded with one NOT operator.

A *relational expression* takes the form

E1 <relational-operator> E2

where E1 and E2 are arithmetic expressions and <relational-operator> is one of the following:

< <= = /= (inequality) > >=

A *compound condition* is composed of two or more simple conditions, boolean operators, and parentheses. It is assumed that boolean operators are used in a compound condition.

A *boolean expression* is a condition without relational expressions.

Data Flow Testing

Data Flow Testing involves the selection of test paths of a program according to the locations of definitions and uses of variables in the program.

With X representing a variable and S representing the number of a statement, we define:

DEF(S) = {X | statement S contains a definition of X}
USE(S) = {X | statement S contains a use of X}

A *definition-use chain* (or DU chain) of variable X is of the form [X, S, S'], where S and S' are statement numbers, X is in DEF(S) and USE(S'), and the definition of X in statement S is live at statement S'.

The *DU testing strategy* requires that every DU chain be covered at least once.

Loop Testing

Loop testing **is a white box testing technique that focuses exclusively on the validity of loop constructs. Four different classes of loops can be defined:**

- **Nested loops**
- **Concatenated loops**
- **Simple loops**
- **Unstructured loops**

Black Box Testing

Black box testing methods focus on the functional requirements of the software. A set of input conditions is derived which fully exercises all functional requirements for a program or code fragment in black box testing.

Black box testing attempts to find errors in the following categories:

- **incorrect or missing functions**
- **interface errors**
- **errors in data structures or external database access**
- **performance errors**
- **initialization and termination errors**

Black Box Testing Methods

- ***Equivalence Partitioning*** **- divides the input domain of a program into classes of data from which test cases can be derived**

- ***Boundary Value Analysis*** **- selects test cases that exercise bounding values**

- ***Cause-Effect Graphing Techniques*** **- provide concise representations of logical conditions and corresponding actions**

- ***Comparison Testing*** **- develop software redundantly, using separate software development teams for the same module, and compare the results generated by the independent modules**

Kinds of Automated Testing Tools

- Static analyzers
- Code auditors
- Assertion processors
- Test file generators
- Test data generators
- Test verifiers
- Test harnesses
- Output comparators
- Symbolic execution systems
- Environment simulators
- Data flow analyzers