# CMSC 128

## Introduction to Software Engineering
## Second Semester AY 2007-2008

jachermocilla@uplb.edu.ph

# Software Testing

- Engineer creates a series of test cases that are intended to demolish the software that has been built-an anomaly?
- Requires that the developer discards preconceived notion of the correctness of software just developed and overcome a conflict of interest that occurs when errors are uncovered
- Is testing destructive?Does it instill guilt?
  - NO!

# Testing Objectives

- Testing is the process of executing a program with the intent of finding an error
- A good test case is one that has a high probability of finding an as-yet undiscovered error
- A successful test is one that uncovers an as-yet undiscovered error

# Testing…

- There is a common belief that testing is one which no errors are found…but testing…
- …cannot show the absence of defects, it can only show that software errors are present..

# Testing Principles

- All tests should be traceable to customer requirements
- Test should be planned long before testing begins, before any code is written
- Pareto principle applies to testing
  - 80% of uncovered errors will be traceable to 20% of all program modules
- Should begin in the small and progress towards in the large

# Testing Principles

- Exhaustive testing is not possible
- To be most effective, testing should be done by an independent third party

# Testability

- How easily can a computer program be tested?
  - Design and implement programs that are testable
- Checklist for testable software
  - Operability
    - The better it works, the more efficiently it can be tested
  - Observability
    - What you see is what you test

# Testability

- Checklist for testable software
  - Controllability
    - The better we can control the software, the more the testing can be automated and optimized
  - Decomposability
    - By controlling the scope of testing, we can more quickly isolate problems and perform smart retesting
  - Simplicity
    - The less there is to test, the more quickly we can test it

# Testability

- Checklist for testable software
  - Stability
    - The fewer the changes, the fewer the disruptions to testing
  - Understandability
    - The more information we have, the smarter we will test

# Good Tests

- Has a high probability of finding an error
  - Testers must develop a mental picture of how the software might fail
- A good test is not redundant
  - Testing time and resources are limited
  - Tests should have different purposes
- Should be “best of breed”
  - Use tests that will most likely uncover an error

# Good Tests

- Neither too simple nor too complex
  - Possible to combine a series of tests into a single test case
  - In general, each test should be executed separately

# Test Case Design

- Black-box testing
  - Knowing the special function that a product has been designed to perform, test can be conducted that demonstrate each function is fully operational
- White-box testing
  - Knowing the internal workings of a product, tests can be conducted to ensure that internal operations performs according to specification

# White-box Testing

- Also called glass-box testing
  - Uses control structure of the procedural design to derive test cases
- Characteristics of test cases derived using WBT
  - Guarantee that all independent paths within a module have been exercised at least once
  - Exercise all logical decisions on their true and false sides

# White-box Testing

- Characteristics of test cases derived using WBT
  - Execute all loops at their boundaries and within their operational bounds
  - Exercise internal data structures to assure their validity

# White-box Testing

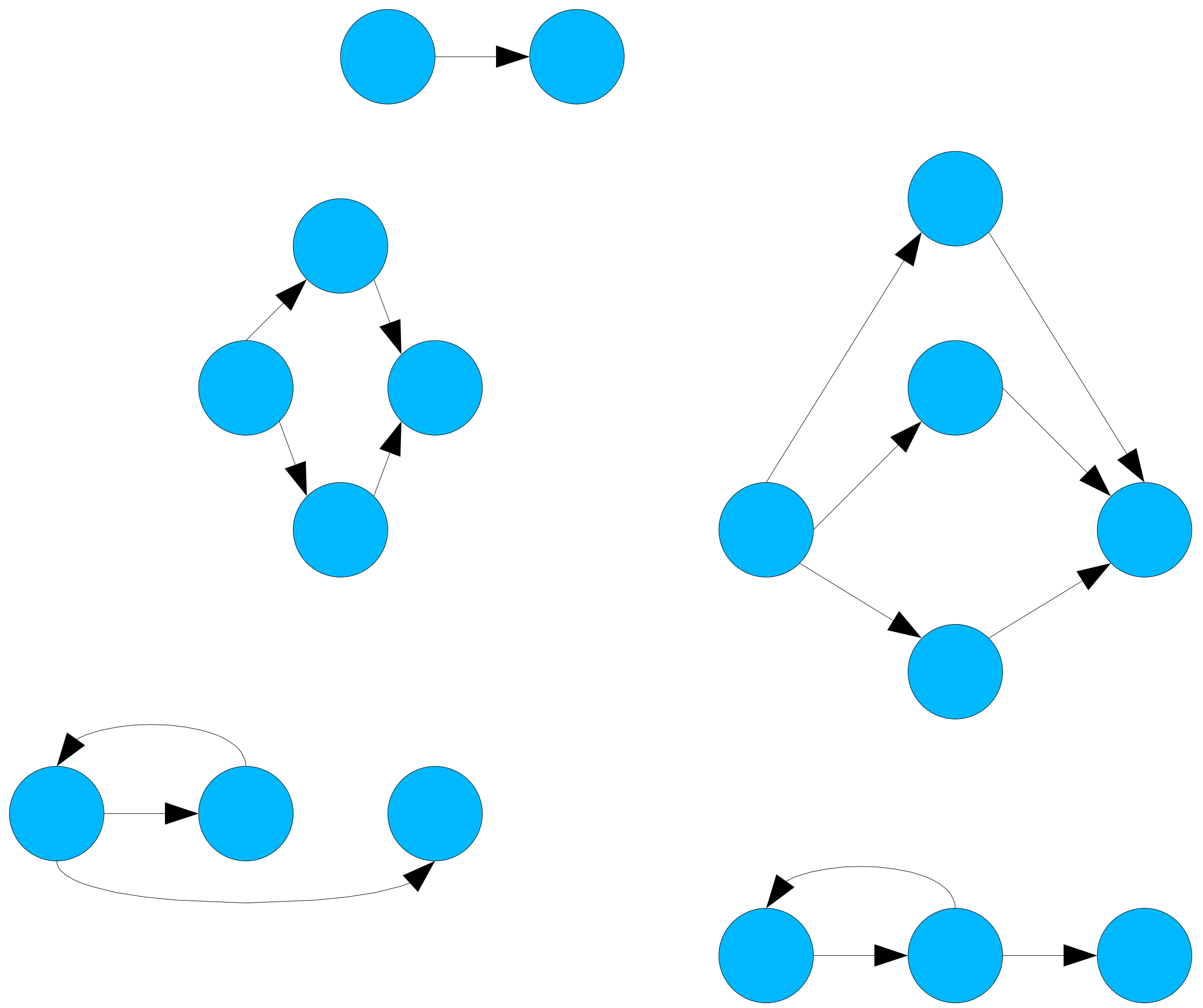
- Why spend time and on WBT?Why not focus just on BBT?Answer lies in the nature of defects
  - Logic errors and assumptions are inversely proportional to the probability that a program path will be executed
  - We often believe that a logical path is not likely to be executed, when in fact, it may be executed on a regular basis
  - Typographical errors are random

# Basis Path Testing

- Enables test designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a *basis set* of execution paths
- Test cases derived to execute the *basis set* are guaranteed to execute every statement in the program at least one time during testing

# Flow Graph Notation

- Sequence
- If
- While
- Until
- Case



# Flow Graph Notation

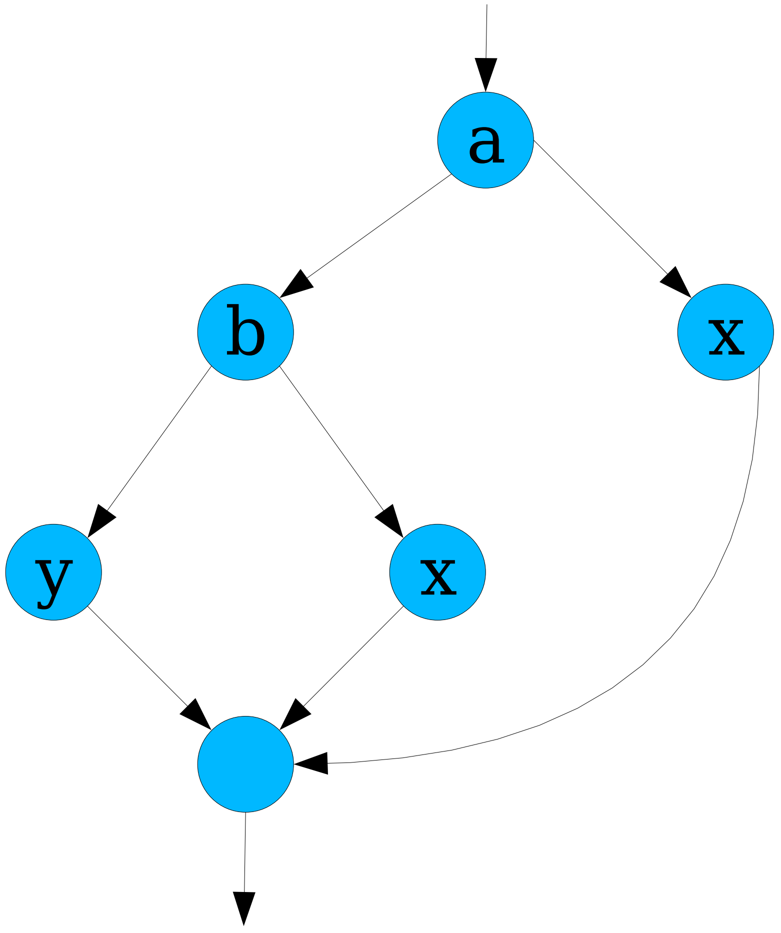
IF a OR b THEN

procedure x

ELSE

procedure y

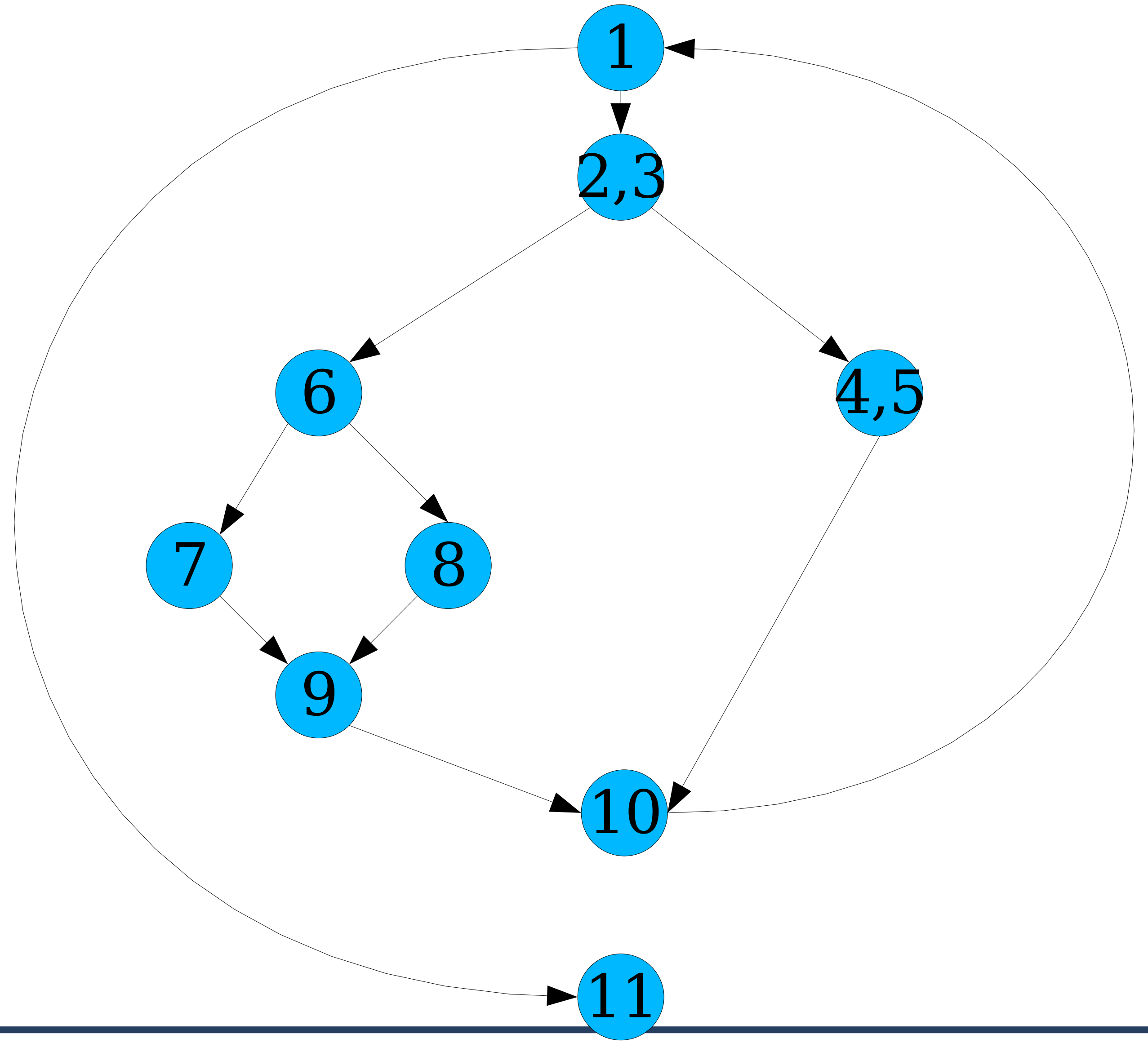
ENDIF



# Cyclomatic Complexity

- Software metric that provides a quantitative measure of the logical complexity of a program
- Independent path
  - A unique path from start to end
- Basis set
  - A set of independent paths

# Example



# Cyclomatic Complexity

- Independent paths
  - Path 1: 1-11
  - Path 2: 1-2-3-4-5-10-1-11
  - Path 3: 1-2-3-6-8-9-10-1-11
  - Path 4: 1-2-3-6-7-9-10-1-11
- Cyclomatic Complexity
  - Cardinality of basis set
  - $V(G) = E - N + 2$
  - Upperbound on the number of tests that must be executed

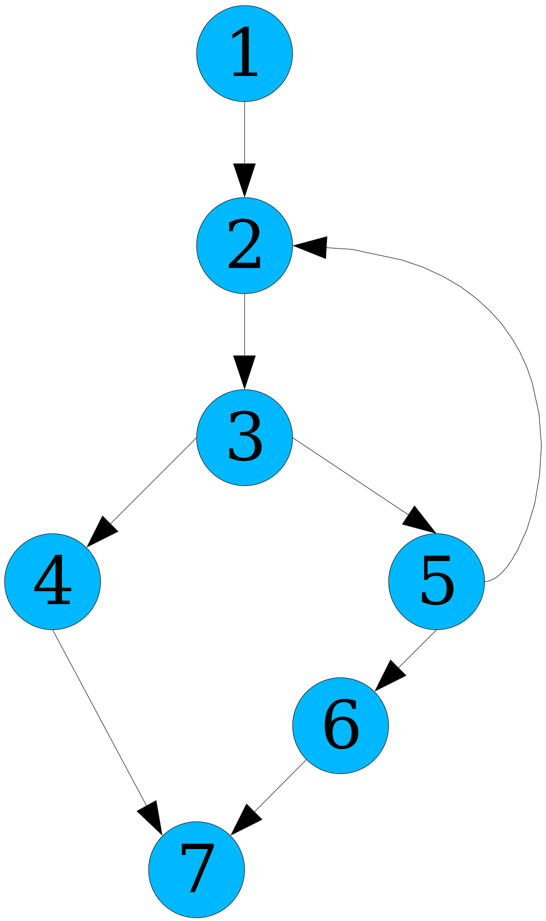
# Designing Test Cases

1. Review procedural design
2. Derive a flow graph
3. Determine cyclomatic complexity
4. Determine a basis set
5. Prepare test cases that will force execution of each path in the basis set

# Example

```
PROCEDURE search
INTERFACE RETURNS position
INTERFACE ACCEPTS data, key
TYPE data[1:5] IS SCALAR ARRAY;
TYPE position, key,i IS INTEGER;
i=1;                    1
DO WHILE (i <= 5)       2
    IF (data[i] == key) 3
        position = i;   4
        RETURN;
    ENDIF
    i = i + 1;          5
ENDDO
position = 0            6
END search              7
```

# Example



# Example

- V(G) = 8 – 7 + 2 = 3
- Path 1: 1-2-3-4-7
- Path 2: 1-[2-3-5]*-2-3-4-7
- Path 3: 1-[2-3-5]*-6-7

# Example

- Path 1 Test Case
  - data : [1,2,3,4,5], key: 1
  - expected: 1
- Path 2 Test Case
  - data : [1,2,3,4,5], key: 5
  - expected: 5
- Path 3 Test Case
  - data : [1,2,3,4,5], key: 6
  - expected: 0

# Condition Testing

- Tests the logical conditions contained in a program module
- Definitions
  - simple condition : boolean variable, relational expression
  - relational expression : E1(relational-operator)E2
  - relational operator : <,<=,==,!=,>,>=
  - compound condition : composed of two or more simple conditions, boolean operators, parenthesis

# Condition Testing

- Definitions
  - boolean operators : OR (|), AND (&&), NOT(!)
  - boolean expression : condition without relational expressions
  - components in a condition
    - boolean operator, boolean variable, pair of boolean parenthesis(surrounding a simple or compound condition), relational operator, or arithmetic expression

# Condition Testing

- A condition is incorrect if at least one component of the condition is incorrect
- Types of error in a condition
  – boolean operator error
  – boolean variable error
  – boolean parenthesis error
  – relational operator error
  – arithmetic expression error

# Condition Testing

- Testing strategies
  - Branch Testing
    - For a compound condition C, the true and false branches of C and every simple condition in C need to be executed at least once
  - Domain Testing
    - Given a relational expression: E1(relational-operator)E2
      - test for E1{<,=,>}E2, three tests
  - Exhaustive testing for n boolean expressions
    - useful if n is small (2 to the n) combinations!

# Data Flow Testing

- Selects test paths of a program according to the location of the definitions and uses of variables in the program
- For a statement with S as statement number
  - DEF(S) = { X | statement S contain a definition of X }
  - USE(S) = { X | statement S contain a use of X}
  - live(X,S,S') : there is a path from S to S' with no other definition of X

# Data Flow Testing

- For a statement with S as statement number
  - Definition-Use chain (DU chain) of variable X is of the form [X, S, S']
    - X member of DEF(S)
    - X member of USE(S')
    - live(X,S,S')
- Data flow testing requires that every DU chain be covered at least once

# Loop Testing

- Focuses on the validity of loop constructs
- Classes of loops
  - simple loops
    - skip loop entirely
    - only one pass through the loop
    - two passes through the loop
    - m passes through the loop where $m < n$
    - $n-1$, $n$, $n + 1$ passes through the loop

# Loop Testing

- Classes of loops
  - nested loops
    - Start at the innermost loop. Set all other loops to minimum values
    - Work outward
    - Continue until all loops have been tested
  - Concatenated loops
    - use approach for simple loops
  - Unstructured loops (GOTO-full)
    - redesign!

# Black-Box Testing

- Test the functional requirements
- Not an alternative to WBT
- Attempts to find errors on the following categories
  - incorrect or missing functions
  - interface errors
  - errors in data structures or external database access
  - performance errors
  - initialization/termination errors

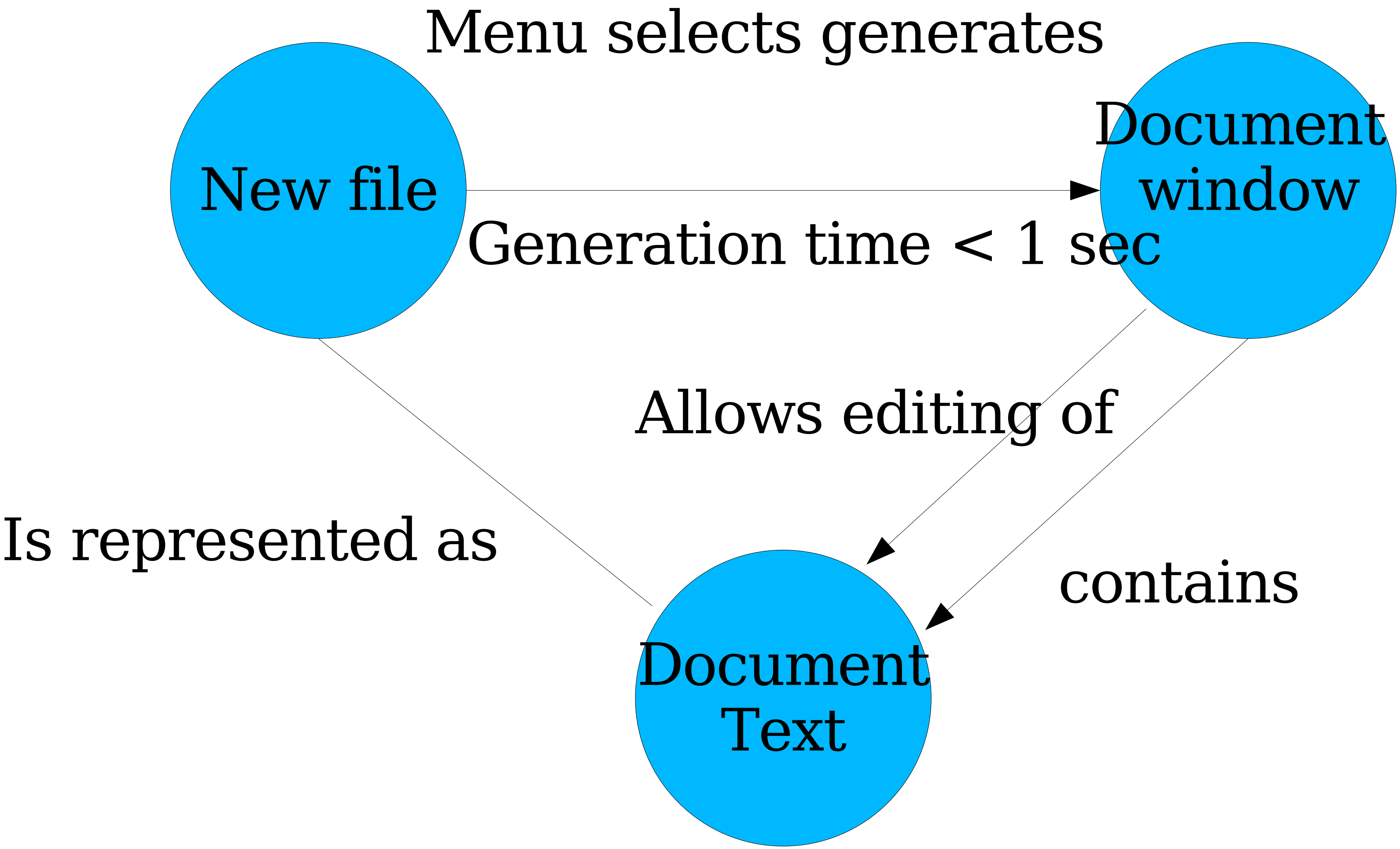
# Black-Box Testing

- Performed late in the testing process
- Focus on the information domain
  - How is functional validity tested?
  - What classes of input will make good test cases?
  - Is the system sensitive to certain input values
  - How are boundaries of a data class isolated
  - What data rates and data volume can the system tolerate
  - What effect will specific combination of data?

# Graph-Based Testing

- Model objects (program or data) and their relationships as a graph
- Nodes represent objects
- Links/Edges represent relationships
  - May be labeled
- Test cases are derived by traversing the graph and covering each of the nodes and relationships shown in the graph
- Relationship may be: symmetric, transitive, and reflexive

# Graph-Based Testing



# Equivalence Partitioning

- Divides the input domain of a program into classes of data from which test cases can be derived
- An ideal test case single-handedly uncovers a class of errors that might otherwise require many cases to be executed
- Goal is to define a test case that uncovers classes of errors

# Equivalence Partitioning

- An equivalence class is present when a relationship is symmetric, transitive, and reflexive
- Given a set X and an equivalence relation ~ on X, the equivalence class of an element a in X is the subset of all elements in X which are equivalent to a
  - If X is the set of all cars, and ~ is the equivalence relation "has the same color as", then one particular equivalence class consists of all green cars

# Equivalence Partitioning

- EC: Valid and invalid states for input condition
- Guidelines for creating equivalence classes given type of input condition
  - *range*: one valid, 2 invalid classes
    - Within range, lesser than minimum, greater than maximum
  - *value*: one valid, two invalid
  - *member of a set*: one valid and one invalid
  - *boolean*: one valid and one invalid

# Boundary Value Analysis

- Errors tend to occur at the boundaries of the input domain than in the center
- Related to equivalence partitioning
  - Test cases selected at the “edges” of the class
- Output domain is also tested

# Boundary Value Analysis

- Guidelines
  - Range: [a,b], test <a, a,…,b,>b
  - Number of values: test minimum, maximum, <minimum, >maximum
  - Apply tests above to output conditions
  - Test boundary values in data structures: array

# Comparison Testing

- Used in critical software, high reliability requirements
- Different implementation for same specification
- Similar tests are applied for each implementation
- Not full-proof since specification may contain error

# Specialized Tests

- GUI
  - Windows
  - Menus
  - Data entry
- Client/Server Architectures
- Documentation and Help Facilities
- Real-time systems
  - Task testing. Behavioral, Intertask, System

# Summary

- Derive a set of tests that have the highest likelihood of uncovering errors
- White-box test focus on program control structure-testing in the small
- Black-box tests are designed to uncover errors in functional requirements without regard to the internal workings of a program
- Testing never ends, its transferred to customers

# Reference

- Roger S. Pressman.Software Engineering: A Practitioner's Approach, 4th Ed.McGraw-Hill,1997. Chapter 16