# Chapter 14
# Testing Tactics

Moonzoo Kim
CS Division of EECS Dept.
KAIST
moonzoo@cs.kaist.ac.kr
http://pswlab.kaist.ac.kr/courses/cs550-07

# Overview of Ch14. Testing Tactics

- 14.1 Software Testing Fundamentals
- 14.2 Blackbox and White-Box Testing
- 14.3 White-Box Testing
- 14.4 Basis Path Testing
  - Glow Graph Notation
  - Independent Program Paths
  - Deriving Test Cases
  - Graph Matrices
- 14.5 Control Structure Testing
  - Condition Testing
  - Data Flow Testing
  - Loop Testing

# Testability

- Operability
  - it operates cleanly
- Observability
  - the results of each test case are readily observed
- Controllability
  - the degree to which testing can be automated and optimized
- Decomposability
  - testing can be targeted
- Simplicity
  - reduce complex architecture and logic to simplify tests
- Stability
  - few changes are requested during testing
- Understandability
  - of the design

- Modular design provides good testability
- Let’s think about embedded SW
  - mobile phone software
  - Linux kernel

# What is a “Good” Test?

- A good test has a high probability of finding an error
- A good test is not redundant.
- A good test should be “best of breed”
- A good test should be neither too simple nor too complex

# Designing Unique Tests (pg423)

- **The scene:**
  - Vinod's cubical.
- **The players:**
  - Vinod, Ed

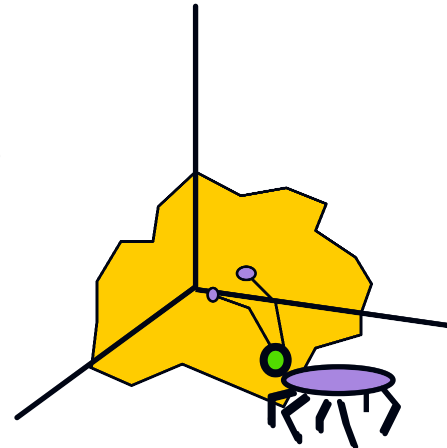
    members of the *SafeHome* software engineering team.
- **The conversation:**
- **Vinod:** So these are the test cases you intend to run for the *password Validation* operation.
- **Ed:** Yeah, they should cover pretty much all possibilities for the kinds of passwords a user might enter.
- **Vinod:** So let's see ... you note that the correct password will be 8080, right?
- **Ed:** Uh huh.
- **Vinod:** And you specify passwords 1234 and 6789 to test for errors in recognizing invalid passwords?
- **Ed:** Right, and I also test passwords that are close to the correct password, see ... 8081 and 8180.
- **Vinod:** Those are okay, but I don't see much point in running both the 1234 and 6789 inputs. They're redundant . . . test the same thing, don't they?

- **Ed:** Well, they're different values.
- **Vinod:** That's true, but if 1234 doesn't uncover an error ... in other words ... the *password Validation* operation notes that it's an invalid password, it is not likely that 6789 will show us anything new.
- **Ed:** I see what you mean.
- **Vinod:** I'm not trying to be picky here ... it's just that we have limited time to do testing, so it's a good idea to run tests that have a high likelihood of finding new errors.
- **Ed:** Not a problem ... I'll give this a bit more thought.

# Test Case Design

"Bugs lurk in corners and congregate at boundaries ..."

*Boris Beizer*

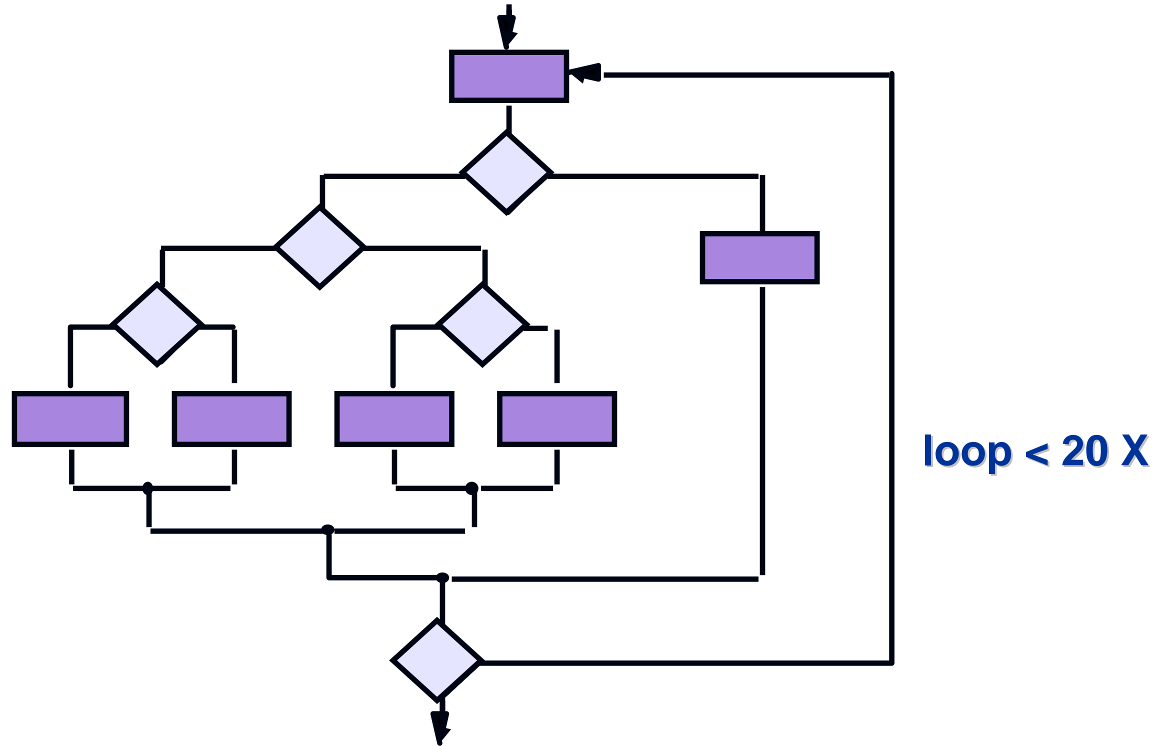


***OBJECTIVE*** to uncover errors

***CRITERIA*** in a complete manner

***CONSTRAINT*** with a minimum of effort and time

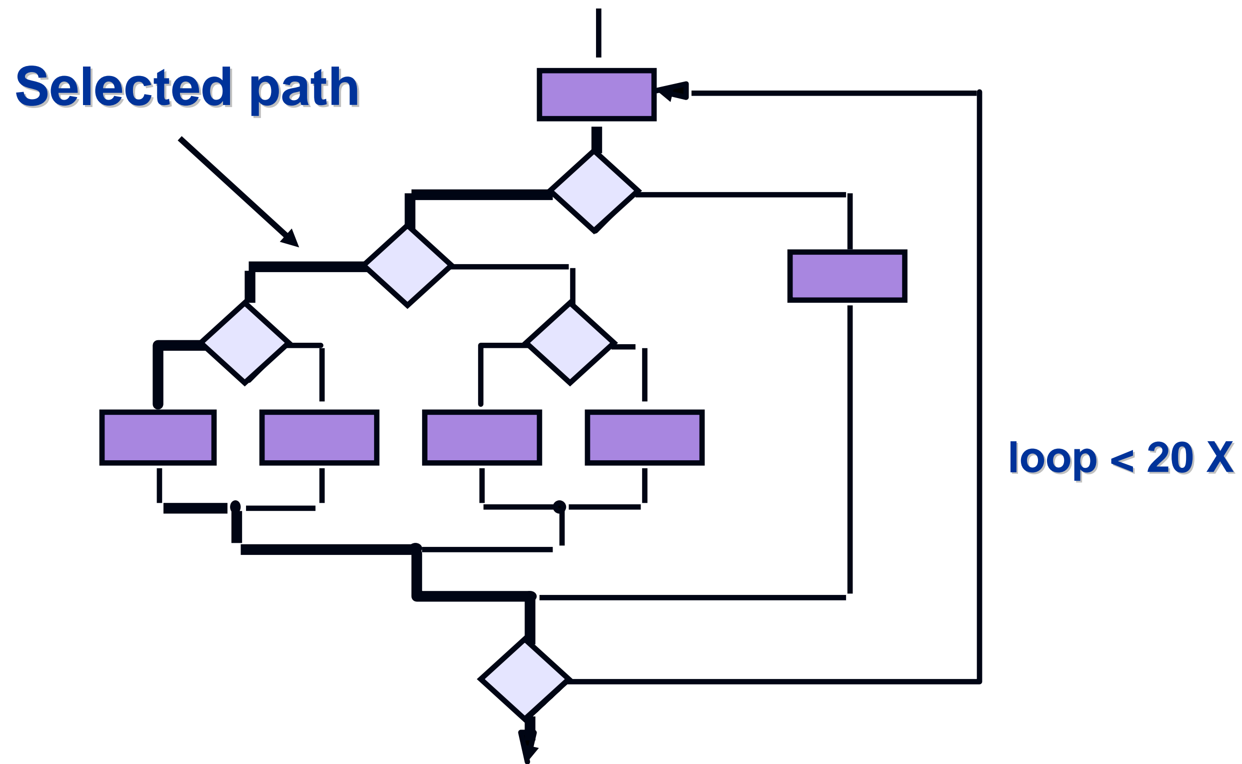
# Exhaustive Testing



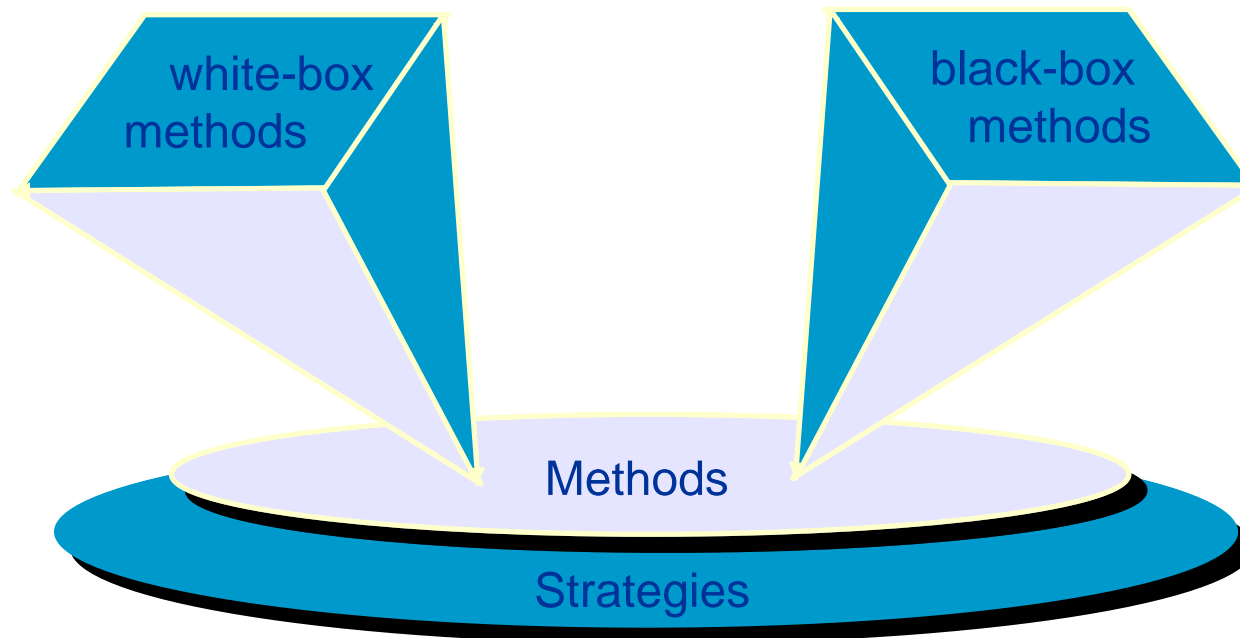
loop < 20 X

There are $10^{14}$ possible paths! If we execute one test per millisecond, it would take 3,170 years to test this program!!

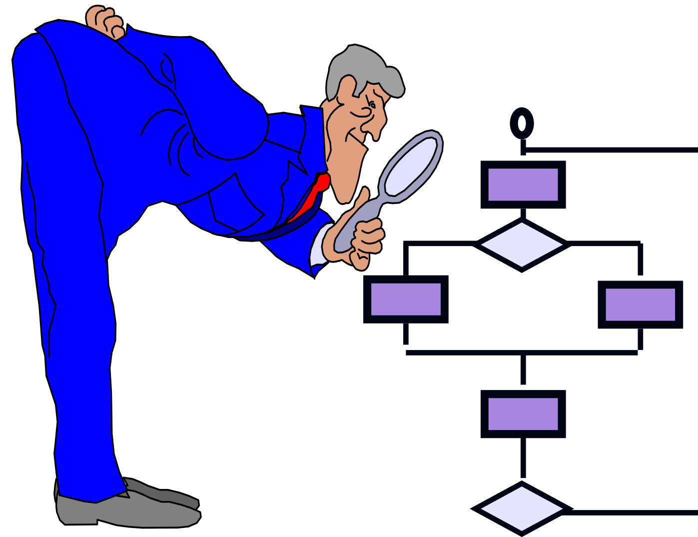
# Selective Testing



# Software Testing



# White-Box Testing



**... our goal is to ensure that all statements and conditions have been executed at least once ...**

# Why Cover?

- logic errors and incorrect assumptions are inversely proportional to a path's execution probability
- we often believe that a path is not likely to be executed; in fact, reality is often counter intuitive
- typographical errors are random; it's likely that untested paths will contain some

However, testing for statement coverage is never sufficient

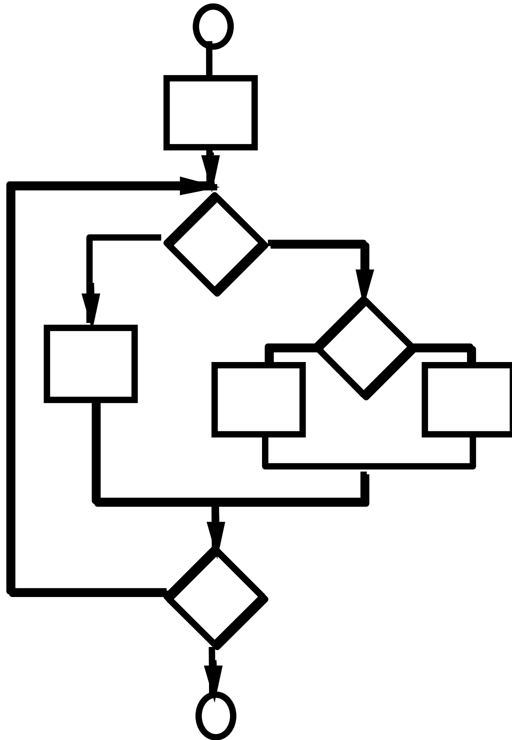
# Why More than Coverage?

- A flow graph does not reflect a real imperative program
  - A state of a real imperative program consists of values of variables while graph theory considers a node as a simple entity
- Most complicated error is caused from loop construct
  - Coverage test does not consider loop
- Therefore, coverage testing should be considered as a minimum requirement

**Formal verification techniques can help**

**$10^{14}$ paths can be handled by CPU**

# Basis Path Testing



First, we compute the cyclomatic complexity:

- number of simple decisions + 1

- number of edge – number of node +2

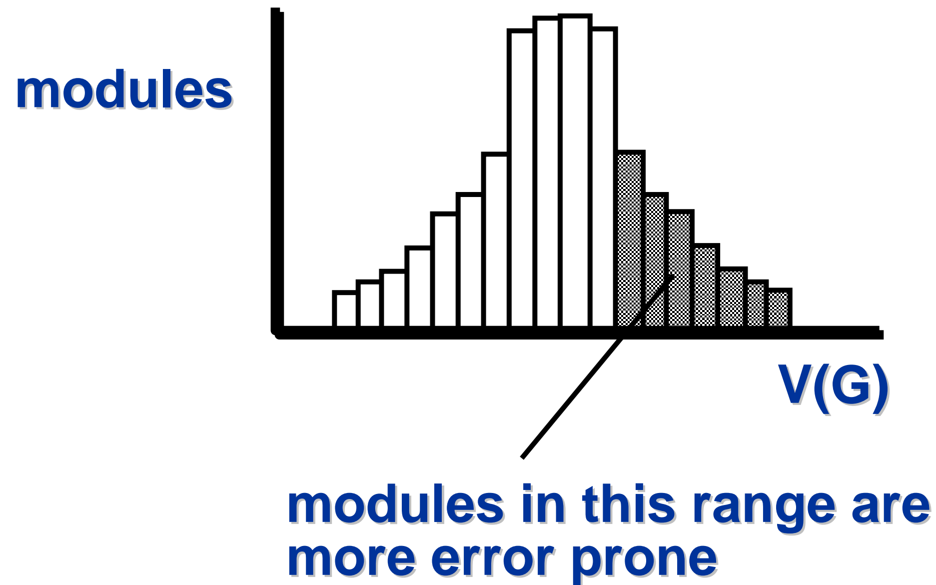
- number of enclosed areas + 1

- In this case, V(G) = 4

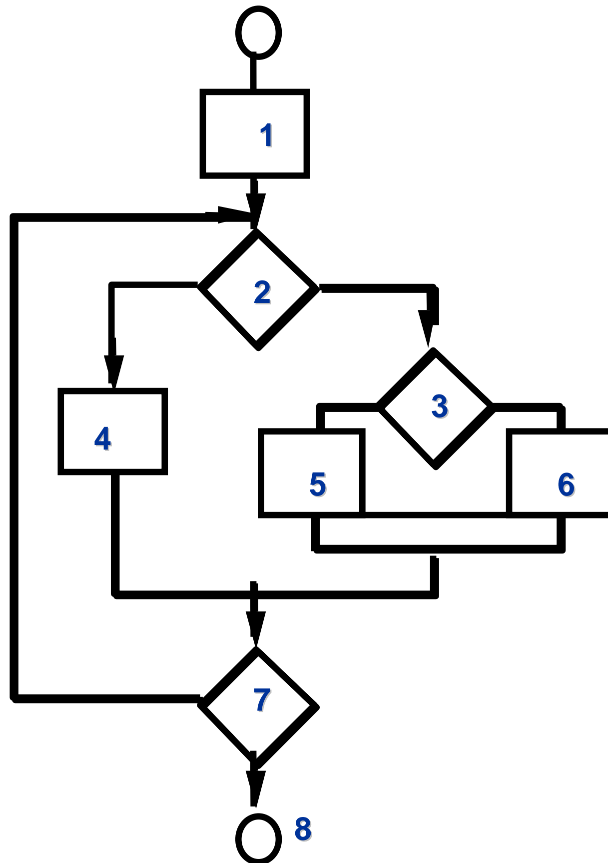
V(G) is the upper bound for the # of independent paths for complete coverage

# Cyclomatic Complexity

A number of industry studies have indicated that the higher V(G), the higher the probability or errors.



# Basis Path Testing




Next, we derive the independent paths:
(paths containing a new edge)

Since V(G) = 4,
there are four paths

Path 1: 1,2,3,6,7,8
Path 2: 1,2,3,5,7,8
Path 3: 1,2,4,7,8
Path 4: 1,2,4,7,2,4,...7,8

Finally, we derive test cases to exercise these paths.

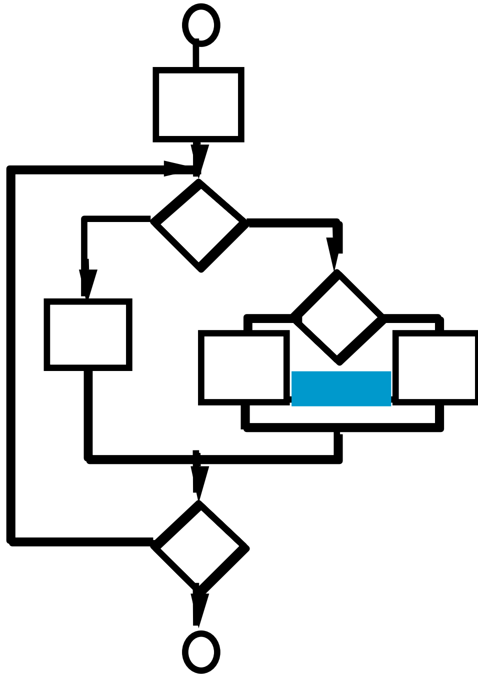
# Using Cyclomatic Complexity (pg428)

- **The scene:**
  - Shakira's cubicle.
- **The players:**
  - Vinod,Shakira

    members of the *SafeHome* software engineering team who are working on test planning for the security function.
- **The conversation:**
- **Shakira:** Look ... I know that we should unit test al! the components for the security function, but there are a lot of 'em and if you consider the number of operations that have to be exercised, I don't know ... maybe we should forget white-box testing, integrate everything, and start running black-box tests.
- **Vinod:** You figure we don't have enough time to do component tests, exercise the operations, and then integrate?
- **Shakira:** The deadline for the first increment is getting closer than I'd like ... yeah, I'm concerned.
- **Vinod:** Why don't you at least run white-box tests on the operations that are likely to be the most error prone?

- **Shakira (exasperated):** And exactly how do I know which are likely to be the most error prone?
- **Vinod:** V of *G*.
- **Shakira:** Huh?
- **Vinod:** Cyclomatic complexity--V of *G*. Just compute *V(G)* for each of the operations within each of the components and see which have the highest values for V(G). They're the ones that are most likely to be error prone.
- **Shakira:** And how do I compute *V of G?*
- **Vinod:** It's really easy. Here's a book that describes how to do it.
- **Shakira (leafing through the pages):** Okay, it doesn't look hard. I'll give it a try. The ops with the highest *V(G)* will be the candidates for white-box tests.
- **Vinod:** Just remember that there are no guarantees. A component with a low *V(G)* can still be error prone.
- **Shakira:** Alright. But at least this'll help me to narrow down the number of components that have to undergo white-box testing.

# Basis Path Testing Notes



- you don't need a flow chart, but the picture will help when you trace program paths
- count each simple logical test, compound tests count as 2 or more
- basis path testing should be applied to critical modules

# Graph Matrices

- A graph matrix is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on a flow graph
- Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes.
- By adding a *link weight* to each matrix entry, the graph matrix can become a powerful tool for evaluating program control structure during testing

# Control Structure Testing

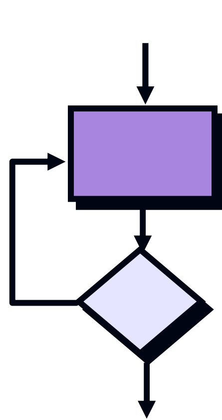
- Condition testing
  - a test case design method that exercises the logical conditions contained in a program module
- Data flow testing
  - selects test paths of a program according to the locations of definitions and uses of variables in the program

# Data Flow Testing

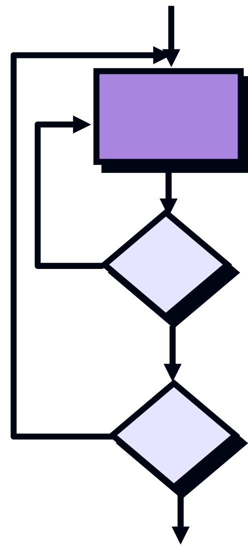
- For a statement S
  - DEF(S) = {X| statement S contains a definition of X}
  - USE(S) = {X| statement S contains a use of X}
- A definition-use (DU) chain of variable X is of the form [X,S,S'] where S and S' are statement, X is in DEF(S) and USE(S')
  - [x,s1,s3] is a DU chain
  - [y,s1,s3] is NOT a DU chain
- A branch is not guaranteed to be covered by DU testing

```
void f() {
s1:   int x = 10, y;
s2:   if ( …) {
          …
s3:       y = x + 1;
      }
```

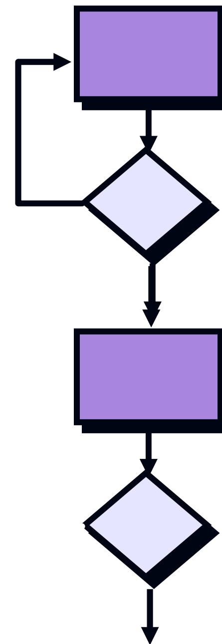
# Loop Testing



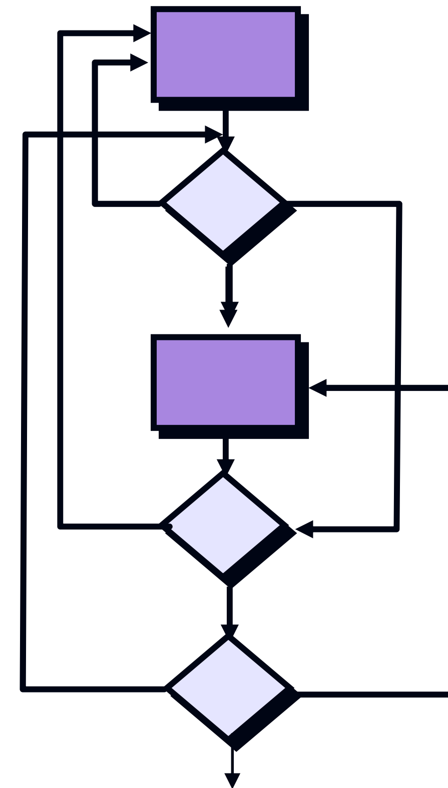
Simple loop



Nested Loops



Concatenated Loops



Unstructured Loops

# Loop Testing: Simple Loops

***Minimum conditions—Simple Loops***

1. skip the loop entirely
2. only one pass through the loop
3. two passes through the loop
4. m passes through the loop $m < n$
5. (n-1), n, and (n+1) passes through the loop

where n is the maximum number of allowable passes

# Loop Testing: Nested Loops

***Nested Loops***

**Start at the innermost loop. Set all outer loops to their minimum iteration parameter values.**

**Test the min+1, typical, max-1 and max for the innermost loop, while holding the outer loops at their minimum values.**

**Move out one loop and set it up as in step 2, holding all other loops at typical values. Continue this step until the outermost loop has been tested.**

***Concatenated Loops***

**If the loops are independent of one another**
**then treat each as a simple loop**
**else* treat as nested loops**
**endif***

***for example, the final loop counter value of loop 1 is used to initialize loop 2.***