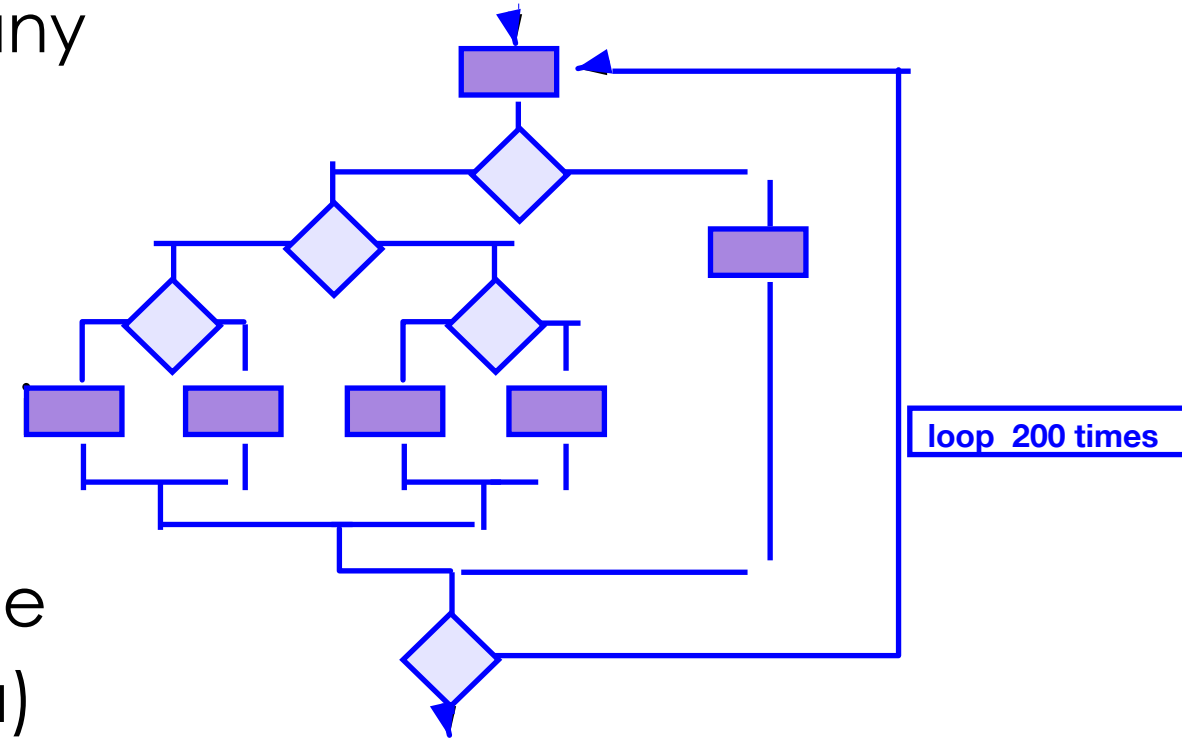
# Software Engineering

Part (XI)- Software Testing

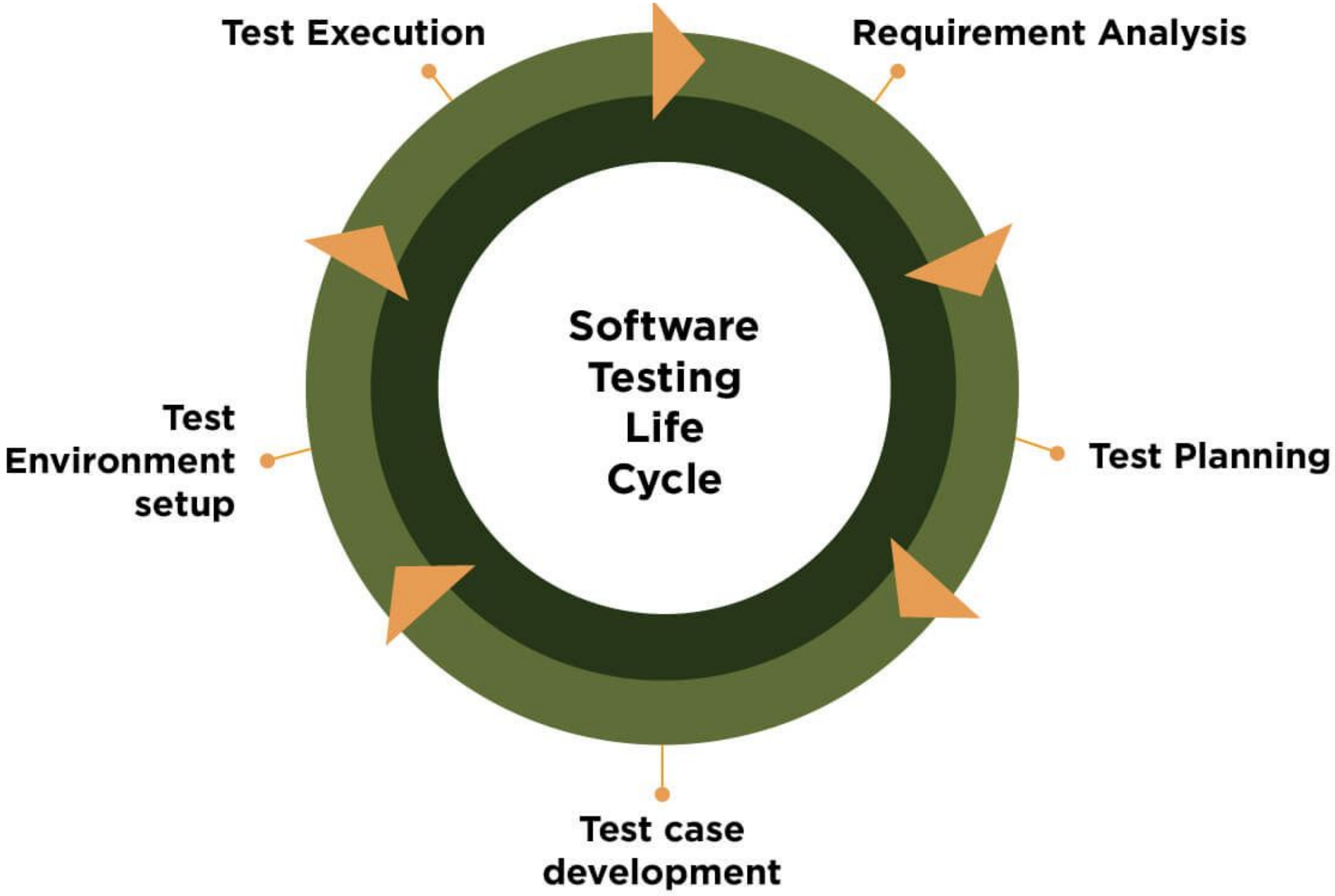
By: Mehran Alidoost Nia
Shahid Beheshti University, Fall 2023

# Testing

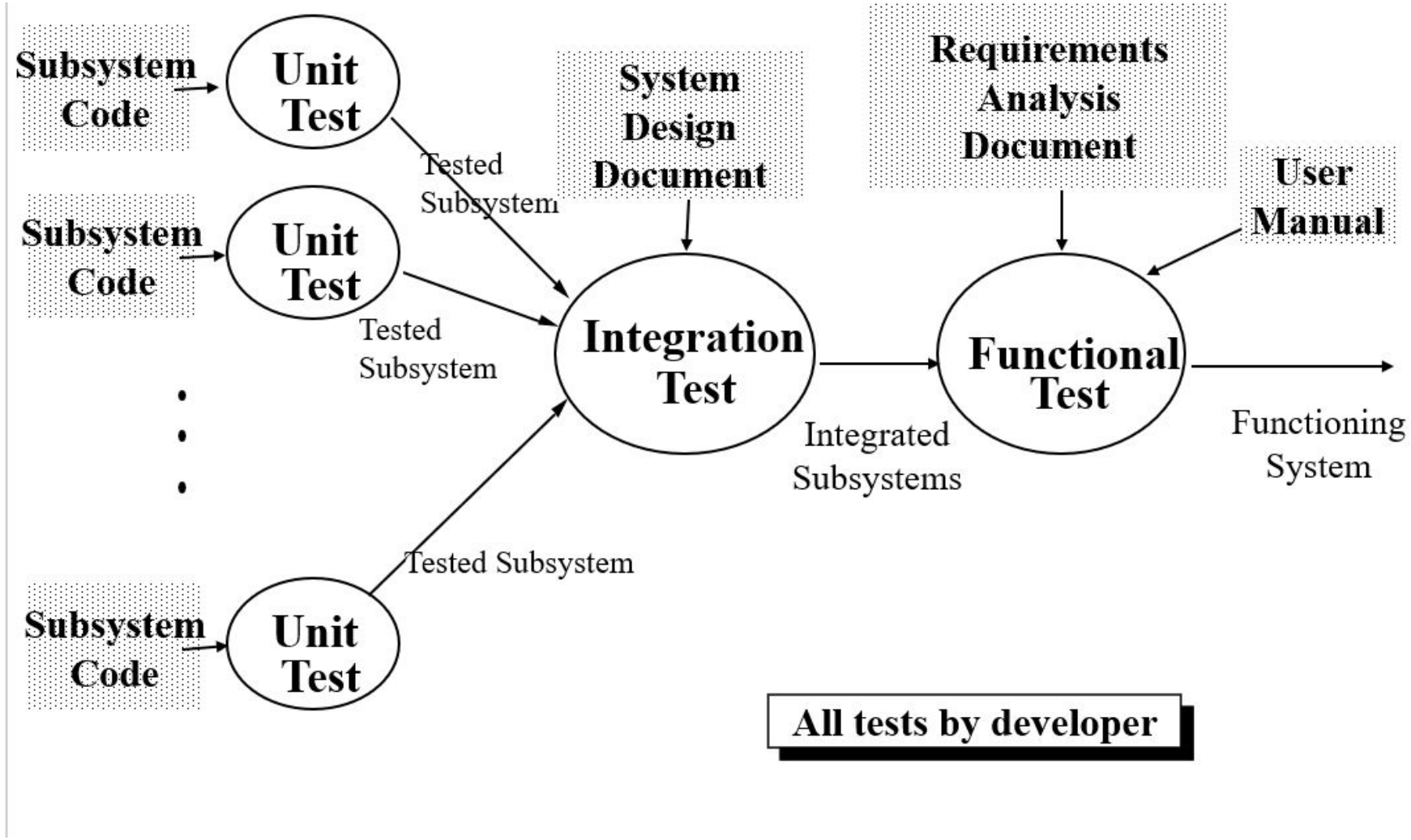
- It is impossible to completely test any nontrivial module or any system
  - Theoretical limitations: Halting problem
  - Practical limitations: Prohibitive time and cost
- Testing can only show the presence of bugs, not their absence (Dijkstra)



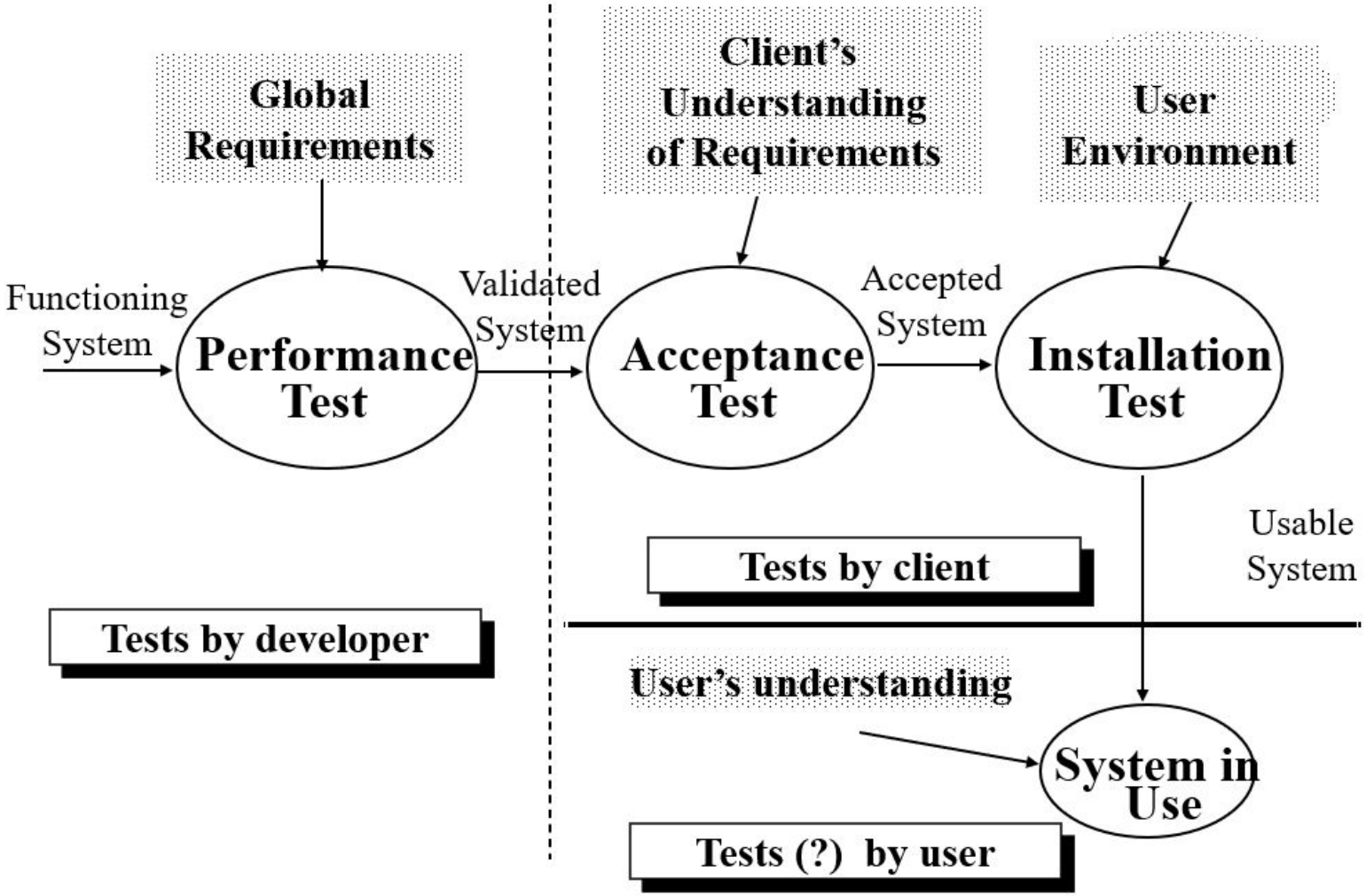

**total number of execution paths?**


Test Execution
Requirement Analysis
Software
Testing
Life
Cycle
Test
Environment
setup
Test Planning
Test case
development

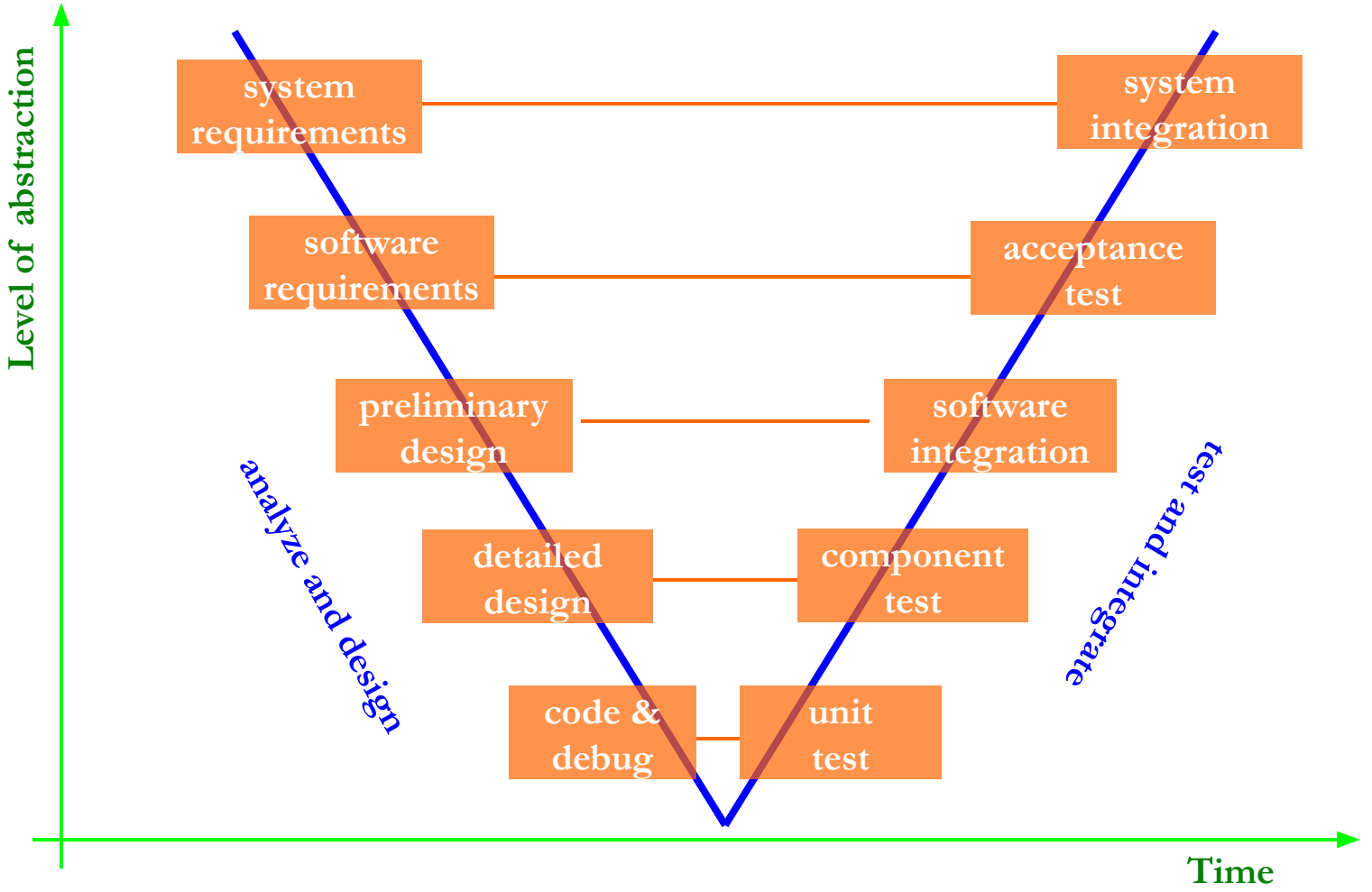
# Testing Activities



# Testing Activities



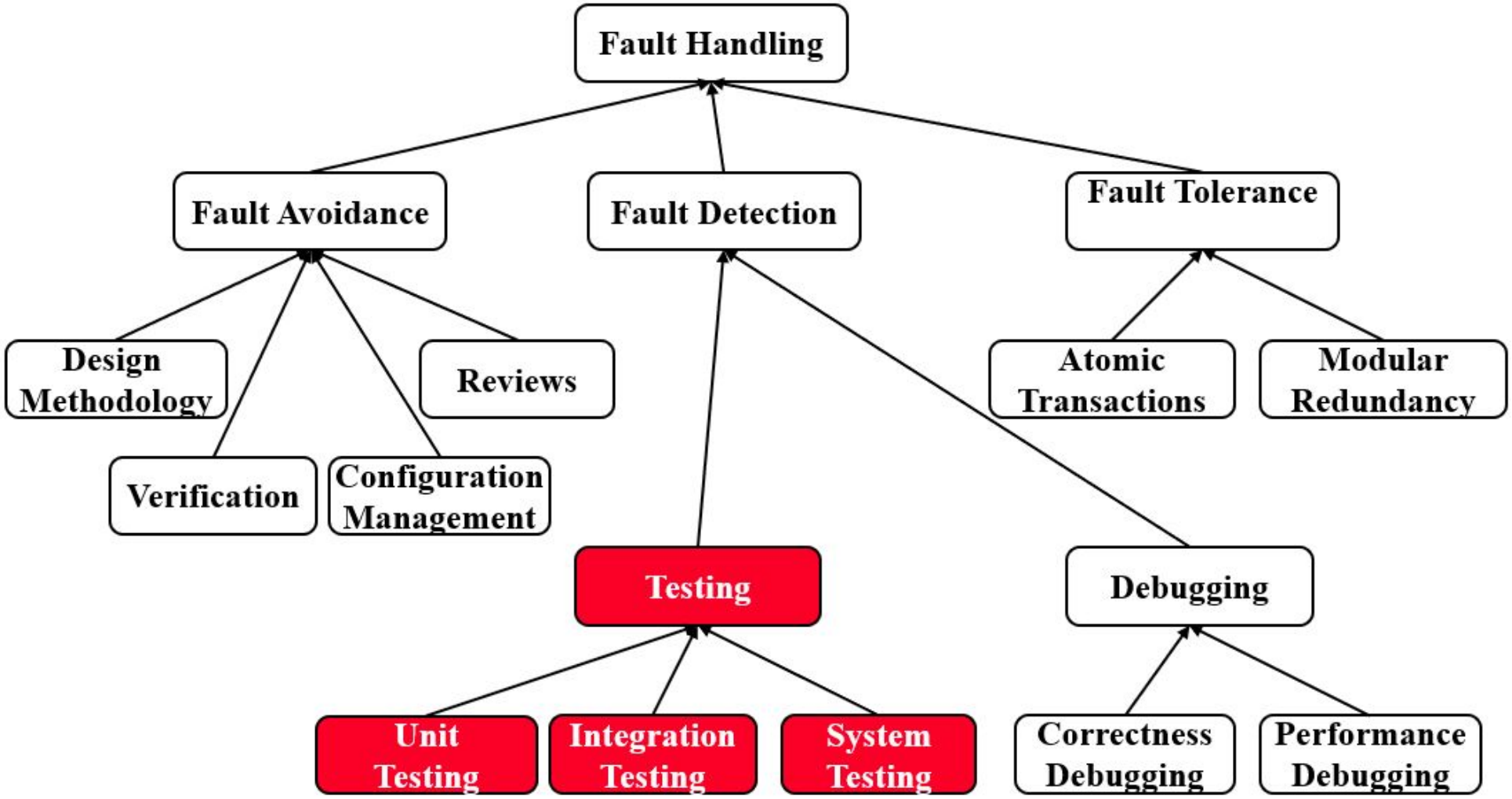
# Levels of Testing in V Model



# Test Planning

- A Test Plan:
  - covers all types and phases of testing
  - guides the entire testing process
  - who, why, when, what
  - developed as requirements, functional specification, and high-level design are developed
  - should be done before implementation starts
- A test plan includes:
  - test objectives
  - schedule and logistics
  - test strategies
  - test cases
    - procedure
    - data
    - expected result
  - procedures for handling problems

# Fault Handling Techniques



# Types of Testing

- Unit Testing:
  - Individual *subsystem*
  - Carried out by developers
  - Goal: Confirm that subsystems is correctly coded and carries out the intended functionality
- Integration Testing:
  - Groups of subsystems (collection of classes) and eventually the entire system
  - Carried out by developers
  - Goal: Test the *interface* among the subsystem

# System Testing

- System Testing:
  - The entire system
  - Carried out by developers
  - Goal: Determine if the system meets the requirements (functional and global)
- Acceptance Testing:
  - Evaluates the system delivered by developers
  - Carried out by the client. May involve executing typical transactions on site on a trial basis
  - Goal: Demonstrate that the system meets customer requirements and is ready to use
- Implementation (Coding) and testing go hand in hand

# *Unit Testing*

- Informal:
  - Incremental coding

Write a little, test a little

- Static Analysis:
  - Hand execution: Reading the *source code*
  - Walk-Through (informal presentation to others)
  - Code Inspection (formal presentation to others)
  - Automated Tools checking for
    - syntactic and semantic errors
    - departure from coding standards
- Dynamic Analysis:
  - Black-box testing (Test the input/output behavior)
  - *White-box* testing (Test the internal logic of the subsystem or object)
  - Data-structure based testing (Data types determine test cases)

Which is more effective, static or dynamic analysis?

# Black-box Testing

- Focus: I/O behavior. If for any given input, we can predict the output, then the module passes the test.
  - Almost always impossible to generate all possible inputs ("test cases")
- Goal: Reduce number of test cases by equivalence partitioning:
  - Divide input conditions into equivalence classes
  - Choose test cases for each equivalence class. (Example: If an object is supposed to accept a negative number,  testing one negative number is enough)

- If x  = 3 then …
- If x > -5 and x < 5 then …

What would be the equivalence classes?

# Black-box Testing (Continued)

- Selection of equivalence classes (No rules, only guidelines):
  - Input is valid across range of values. Select test cases from  3 equivalence classes:
    - Below the range
    - Within the range
    - Above the range
  - Input is valid if it is from a discrete set. Select test cases from 2 equivalence classes:
    - Valid discrete value
    - Invalid discrete value
- Another solution to select only a limited amount of test cases:
  - Get knowledge about the inner workings of the unit being tested => white-box testing

Are these complete?

# White-box Testing

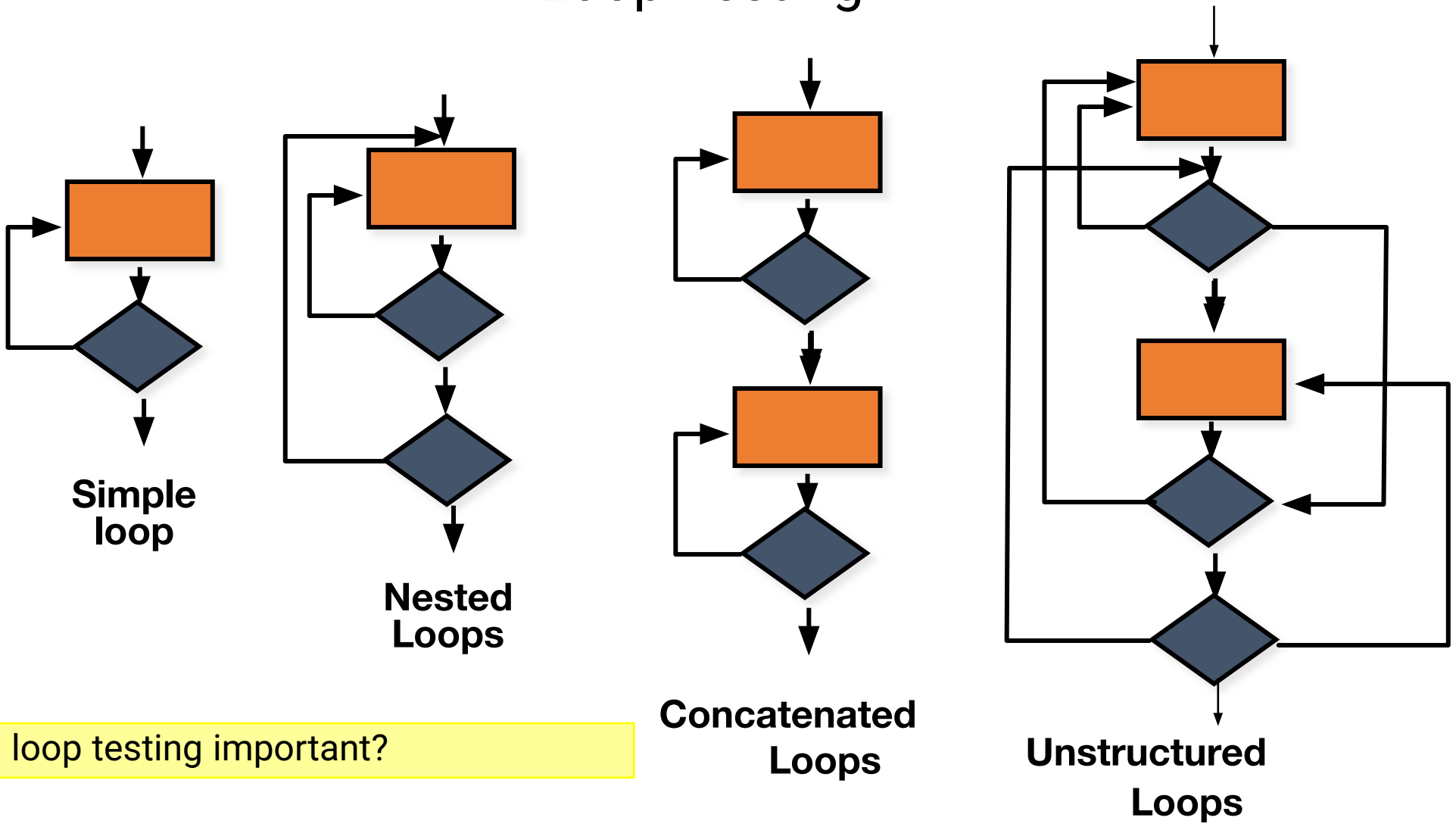
- Focus: Thoroughness (Coverage). Every statement in the component is executed at least once.
- Four types of white-box  testing
  - Statement Testing
  - Loop Testing
  - Path Testing
  - Branch Testing

# White-box Testing (Continued)

- Statement Testing (Algebraic Testing): Test single statements
- Loop Testing:
  - Cause execution of the loop to be skipped completely (Exception: Repeat loops).
  - Loop to be executed exactly once
  - Loop to be executed more than once
- Path testing:
  - Make sure all paths in the program are executed
- Branch Testing (Conditional Testing): Make sure that each possible outcome from a condition is tested at least once

```
if ( i =    TRUE) printf("YES\n");          else  printf("NO\n");
Test cases: 1) i = TRUE; 2) i = FALSE
```

# Loop Testing

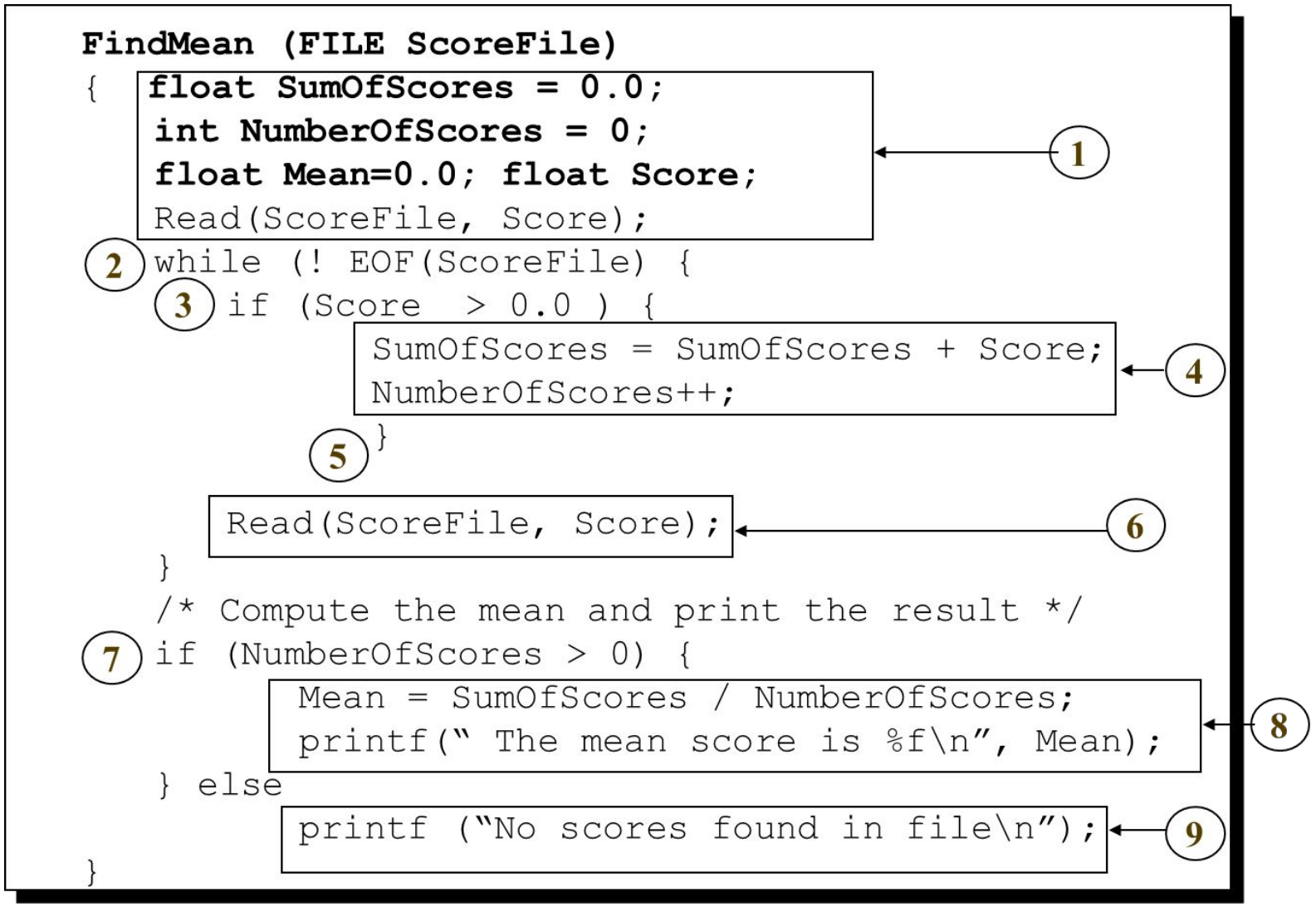



Why is loop testing important?

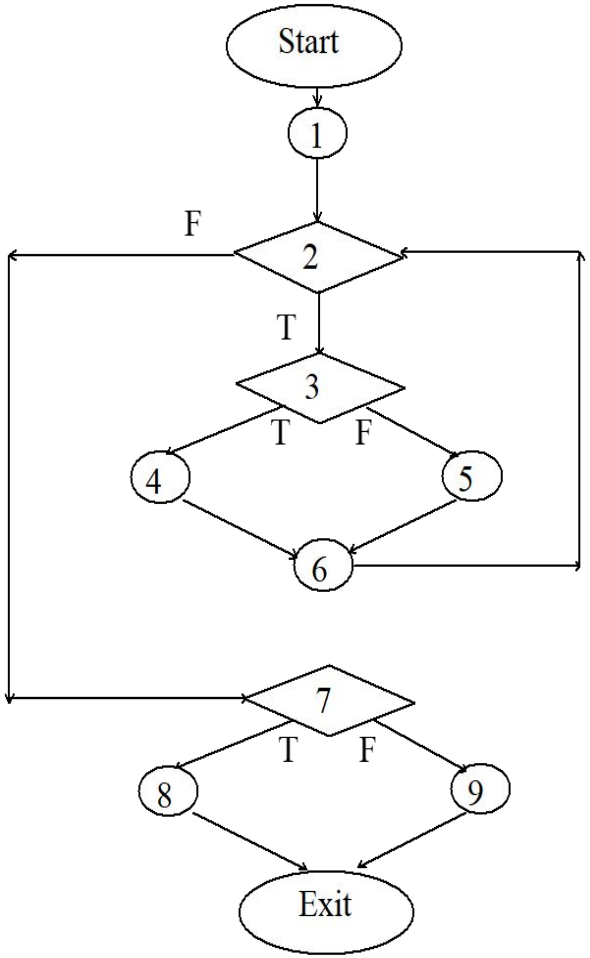
# White-box Testing

```
FindMean(float Mean, FILE ScoreFile)
{ SumOfScores = 0.0; NumberOfScores = 0; Mean = 0;
 Read(ScoreFile, Score);  /*Read in and sum the scores*/
 while (! EOF(ScoreFile) {
         if ( Score > 0.0 ) {
               SumOfScores = SumOfScores + Score;
              NumberOfScores++;
          }
          Read(ScoreFile, Score);
 }
 /* Compute the mean and print the result */
 if (NumberOfScores > 0 ) {
              Mean = SumOfScores/NumberOfScores;
              printf("The mean score is %f \n",  Mean);
 } else
              printf("No scores found in file\n");
}
```

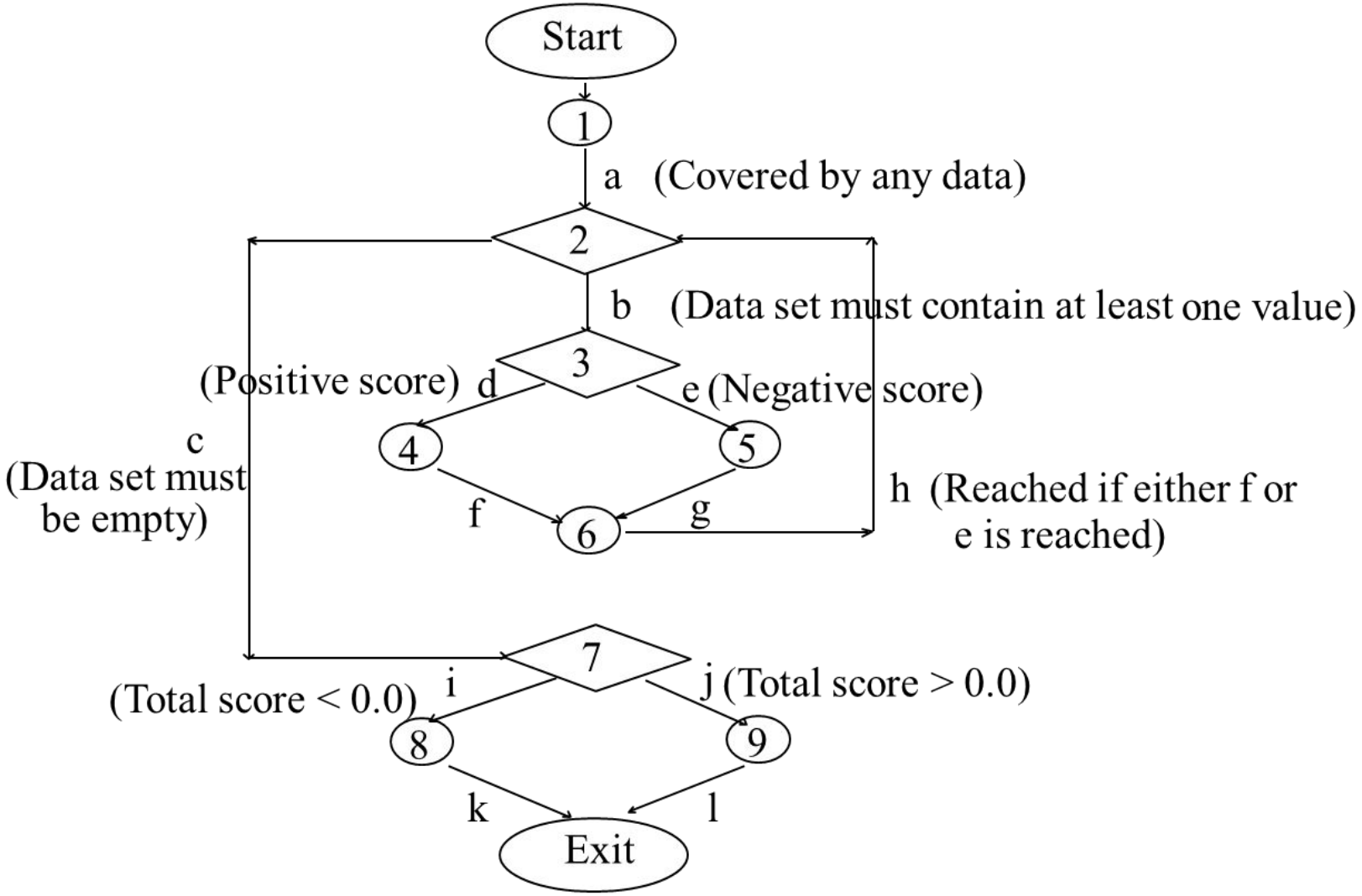
# White-box Testing



# Constructing the Logic Flow Diagram



# Constructing the Logic Flow Diagram



# White-box vs. Black-box

- White-box Testing:
  - Potentially infinite number of paths have to be tested
  - White-box testing often tests what is done, instead of what should be done
  - Cannot detect missing use cases
- Black-box Testing:
  - Potential combinatorial explosion of test cases (valid & invalid data)
  - Often not clear whether the selected test cases uncover a particular error
  - Does not discover extraneous use cases ("features")

- Both types of testing are needed
- White-box testing and black box testing are the extreme ends of a testing continuum.
- Any choice of test case lies in between and depends on the following:
  - Number of possible logical paths
  - Nature of input data
  - Amount of computation
  - Complexity of algorithms and data structures

Self reading

# Guidance for Test Case Selection

- Use analysis knowledge about functional requirements (black-box testing):
  - Use cases
  - Expected input data
  - Invalid input data
- Use design knowledge about system structure, algorithms, data structures (white-box testing):
  - Control structures
    - Test branches, loops, ...
  - Data structures
    - Test records fields, arrays, ...

- Use implementation knowledge about algorithms:
  - Examples:
  - Force division by zero
  - Use sequence of test cases for interrupt handler

Self reading

# Unit-testing Heuristics

1. Create unit tests as soon as object design is completed:
   - Black-box test: Test the use cases & functional model
   - White-box test: Test the dynamic model
   - Data-structure test: Test the object model
2. Develop the test cases
   - Goal: Find the minimal number of test cases to cover as many paths as possible
3. Cross-check the test cases to eliminate duplicates
   - Don't waste your time!
4. Desk check your source code
   - Reduces testing time
5. Create a test harness
   - Test drivers and test stubs are needed for integration testing
6. Describe the test oracle
   - Often the result of the first successfully executed test
7. Execute the test cases
   - Don't forget regression testing
   - Re-execute test cases every time a change is made.
8. Compare the results of the test with the test oracle
   - Automate as much as possible

# OOT Strategy

- class testing is the equivalent of unit testing
  - operations within the class are tested
  - the state behavior of the class is examined

…if there is no nesting of classes

- integration applied three different strategies/levels of abstraction
  - thread-based testing—integrates the set of classes required to respond to one input or event
  - use-based testing—integrates the set of classes required to respond to one use case
  - cluster testing—integrates the set of classes required to demonstrate one collaboration

Recall: model-driven software development

# OOT Methods: Behavior Testing

- The tests to be designed should achieve all state coverage.
- That is, the operation sequences should cause the Account class to make transition through all allowable states

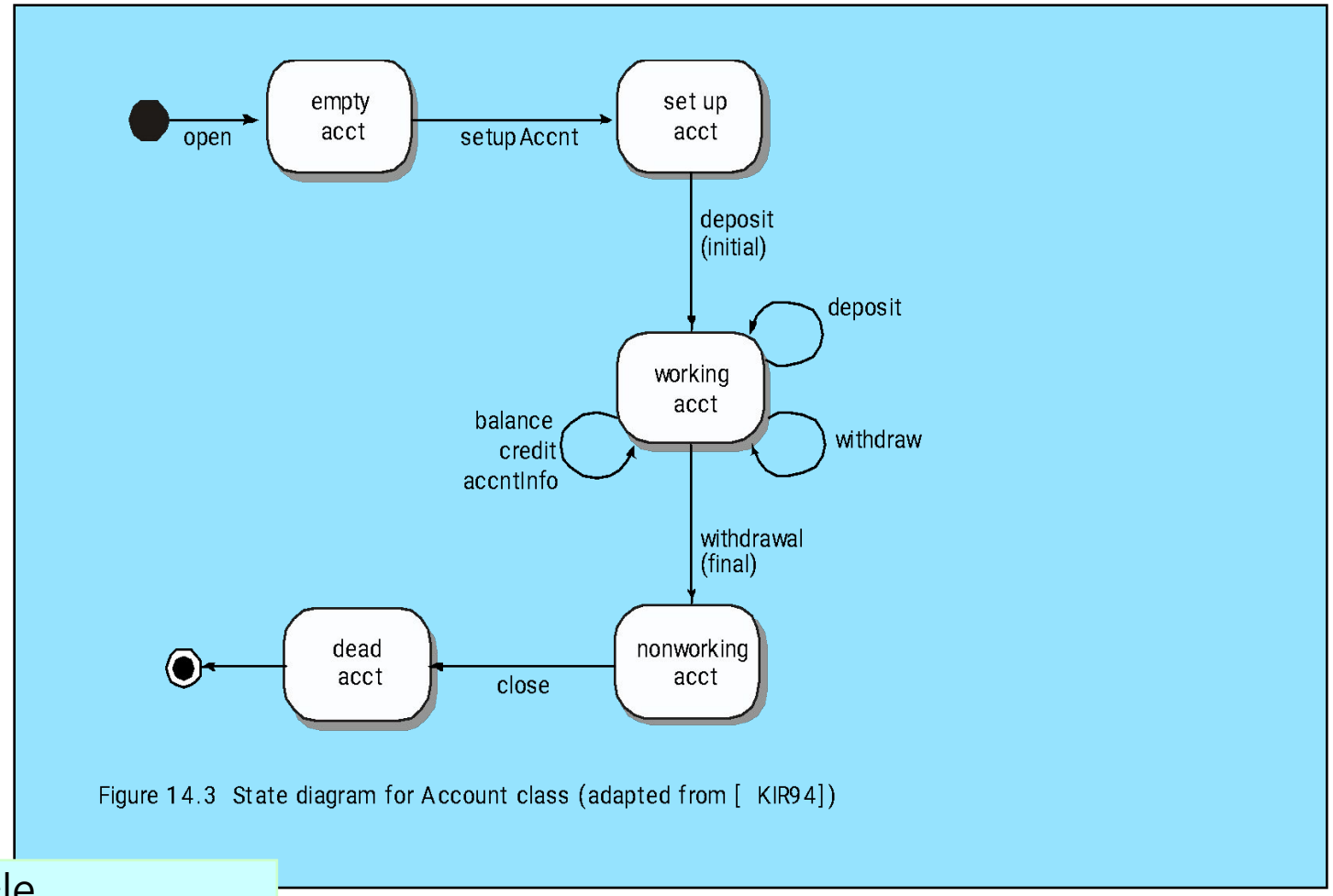



Figure 14.3 State diagram for Account class (adapted from [ KIR94])

This can act as an oracle

# Who Tests the Software?

developer

Understands the system

but, will test "gently"

and, is driven by "delivery"

independent tester

Must learn about the system,

but, will attempt to break it

and, is driven by quality

# Test Metrics: Planning

- Passed test case percentage = Total number of passed test cases / Total number of test cases x 100%
- Failed test case percentage = Total number of failed test cases / Total number of test cases x 100%
- Blocked test case percentage = Total number of blocked test cases / Total number of test cases x 100%
- Fixed defects percentage = Total number of defects fixed / Total number of defects reported x 100%
- Accepted defects percentage = Total number of defects accepted as valid / Total number of defects reported x 100%
- Defects rejected percentage = Total number of defects rejected as invalid / Total number of defects reported x 100%

# Test Metrics: Test Efforts

- Tests run per period = Total number of tests run / Total time taken
- Test design efficiency =  Total number of tests designed / Total time taken
- Test review efficiency = Total number of tests reviewed / Total time taken
- Defects per test hour = Total number of defects / Total number of test hours
- Bugs per test = Total number of bugs found / Total number of tests
- Time to test a bug = Total time taken between defect fix to retest for all defects / Total number of bugs found

# Test Metrics: Coverage

- Code coverage:
  - Code Coverage = (the number of lines of code tested / the total number of lines in the codebase) x 100.
- Test coverage:
  - Test Coverage = (Total number of requirements mapped to test cases / Total number of requirements) x 100.

# Quote of the Day

# Readings

- Software Engineering: A Practitioner's Approach,  Roger Pressman and Bruce Maxim, 9th Edition, Chapters 15, 16, 20 and 23.
- Clean Architecture: A Craftsman's Guide to Software Structure and Design, Robert Martin, Chapter 28, 2017.
- Software Engineering at Google: Lessons Learned from Programming Over Time, Titus Winters, Tom Manshreck and Hyrum Wright, Chapter 12, 2020.