McGill

Introduction to Software Engineering

ECSE-321

Unit 17 – Quality assurance and testing

Agenda

- Software QA
  - SQA techniques
  - Verification
  - Principles of testing
- Debugging
- Testing
  - Component based testing
  - System/structure testing

Terminology

- Bugs and defects
  - Failure: deviation from specified behavior
  - Defect (fault, bug): cause of a failure
  - Error: the system is in a state where further processing would lead to failure
- Some sources also distinguish
  - Fault vs. defect: before or after release

SQA

- Quality = meeting the requirements
    - Functional
    - Technological
    - Budget
    - Time
- SQA
    - Much more than testing
    - Often a different team (QA team)

Types of SQA

- Verification
  - Meaning: the program conforms to specification

    "Are we building the product right?"
- Validation
  - Meaning: the specified system is what the customer wants built

    "Are we building the right product?"
- Fault prevention
  - Meaning: decrease the chance of occurrence of faults
- Fault detection
  - Meaning: finding the faults in the system
- Fault tolerance
  - Meaning: contain the damage of faults

Another View on How to Deal with Errors

- **Error prevention** (before the system is released):
  - Use good programming methodology to reduce complexity
  - Use version control to prevent inconsistent system
  - Apply verification to prevent algorithmic bugs
- **Error detection** (while system is running):
  - Testing: Create failures in a planned way
  - Debugging: Start with unplanned failures
  - Monitoring: Deliver information about state. Find performance bugs
- **Error recovery** (recover from failure once the system is released):
  - Data base systems (atomic transactions)
  - Modular redundancy
  - Recovery blocks

SQA Techniques

- Testing
  - Unit, integration, system, …
  - Pilot tests - Alpha, beta, …
  - Functional, performance, usability, …
- Manual checks
  - Reviews, inspections, walkthroughs, …
- Reliability measurement
- Modeling and prototyping
- Formal methods
- Defect management
- Debugging
  - Fault search, location, repair

Which technique works best?

1. Personal design checking 15%-70%
2. Design reviews 30%-60%
3. Design inspections 35%-75%
4. Code inspections 30%-70%
5. Prototyping 35%-80%
6. Unit testing 10%-50%
7. Group-test related routines 20%-55%
8. System testing 25%-60%
9. Field testing 35%-65%
10. Cumulative 93%-99%

[Programming Productivity - Jones 1986]

Observations

- Individually, none of these techniques has a definite advantage
- They tend to discover different types of faults
  - Testing: extreme cases and human oversights
  - Reviews: common errors

A combination of techniques is most effective

Verification

- Guaranteeing that the program conforms to specification

  “Are we building the product right?”
- Verification while developing
  - Making sure each stage finished successfully
- Non-execution tests
  - Walkthrough
  - Inspection
  - Peer review
- Automatic verification
  - “Proving” it works
- Integrating tests in the implementation

Walkthrough

- Carefully going over the products. Line by line
  - Requirement spec
  - Design
  - Code
- SQA + development team
- Objectives:
  - Discovering and noting faults, including bad conventions
  - Examining alternatives
  - Provide feedback to development team
  - Discussion forum

Inspection (Fagan 76')

- A wide review (more than a walkthrough)
  - Moderator, reviewers, owner.
  - Objective: finding errors, deviations, inefficiencies

- Five stages:

1. Overview – presented by the owner
2. Preparation – participants try to understand the document.
3. Inspection – going over document very carefully; looking for faults.

   Moderator writes down all faults
4. Rework – owner fixes faults, or addresses them
5. Follow-up – moderator checks all faults are fixed

Validation

- Checking the product or parts of it
- Execution-based testing
  - What can we test?
  - Principles of testing

What can we test?

- Effectiveness
  - Does the software meet the requirements?
    - Ease of use
    - Functionality
    - Cost/effectiveness
- Reliability
  - Frequency and severity
    - MTBF = mean time between failure
      - ALOHA
      - Disk drive – 5x10^?
  - Average time to repair
    - MTTR = mean time to repair

What can we test

- Robustness
  - Operational range
  - Possibility of unexpected results with legitimate input
  - Influence of erroneous input
- Performance
  - Meeting the requirements
    - Space
    - Time
  - Real-time

Agenda

- Software QA
  - SQA techniques
  - Verification
  - Principles of testing
- Debugging
- Testing
  - Component based testing
  - System/structure testing

Principles of testing

- Compliance with requirements
  - Failure = not meeting the requirements. Which requirements?
- Redesign
  - Plan your testing from the requirement phase
- Focus testing in selected subsystems
  - 80% of faults are in 20% of subsystems
- Start from specific subsystems and move to system wide tests
  - Don’t start with checking everything and hope for the best

Principles of testing

- No exhaustive test
  - Practically: cannot cover every possible situation
  - But can make sure every “big” logical condition is covered
- First check – the developer (Alpha)
- Comprehensive test by an outside source (Beta)
  - Increase efficiency
    - i.e. improve probability of fault detection
- A person cannot effectively check himself/herself

Testability

- Meaning:
  - How easy is it to test the software?
  - How much does a given set of tests cover the product
  - How realistic is it to test a system (and fix it) in time
- How to guarantee testability?
  - Follow some rules during development
  - Observability and decomposability

Observability

- What you see is what you test
- Different output for different input
- State and system variables are observable or queriable during runtime
- Past states and previous variables are observable or queriable
  - E.g. – transaction logs
- Everything that affects the output is observable
- Invalid input can be easily determined
- Internal mistakes are discovered by internal mechanisms
- Source code is accessible

Decomposability

- The system is built from isolated subsystems
- Each subsystem can be tested separately
  - Facilitate quicker isolation of faults
- How will we check preconditions to methods?
  - Before calling?
  - In the beginning of the method?

Testing techniques

- White box
  - Checking the internal structure of a module
  - Execution paths
  - Correctness of calculations
  - Correctness of control decision
- Black box
  - Check correctness w.r.t spec, implementation independ.
    - Correctness of output
    - Speed of reaction
- Test data
  - Different data files for different cases

Black box tests - principles

- Get tests from spec/use cases/ sequence diagrams
- Can be designed after defining the spec or from UML
- Each test must be tied to a scenario or a requirement
- Define “equivalence” between tests
- Check several cases and border-line/boundary cases
- Example: If in spec x:1..1000
  - Check: x = -8, 0, 1, 234, 999, 1000, 1001, 1060
- A test is usually more than a data file
  - Script
  - GUI description

Characteristic of a good test

- High probability to discover a bug
  - Tester should have a “mental” image of the software and have a good idea where a bug can be found
- Necessity
  - No redundancy
  - Every test has a different goal
  - The elements checked are different
- Best of breed
  - If multiple tests are available – choose the one with the best coverage
- Not too simple, not too complex
  - Several tests may sometimes be united. But don’t create monster tests

When are the tests completed?

- Never
- When we finish running all of them
  - And achieved complete coverage
- When the product matches the spec
  - E.g., MTBF is larger than some value
- By bug discovery
  - Rate is less than some predefined value
  - X% of bugs were discovered
    - Based on estimate
    - Based on “planted” bugs
    - Based on comparing two independent teams
- When we run out of money/time

So do we get perfect code?

No.

Statistics on defects left in code:

- Industry average: 15..50 defects/KLOC (including code produced using bad development practices)
- Best practices: 1..5 defects/KLOC
  - It is cheaper to build high-quality software than to fix low-quality software
- Reduced rates (0.1..0.5 defects/KLOC) for combinations of QA techniques and for “cleanroom process”
  - Justified in special applications

Reliability measurements

- Predict how software reliability should improve over time as faults are discovered and repaired

- Reliability growth models
- Equal steps: reliability grows by sudden jumps, by a constant amount after fixing each fault
- Normally distributed steps: non-constant jump
  - Negative steps: the reliability might actually decrease after fixing a fault
- Continuous models: focus on time as opposed to discrete steps
  - Recognize that it is increasingly difficult to find new faults
  - Calibration required for type of application
  - Target reliability

- No universally applicable model
  - Highly dependent on type of application, programming language, development process, testing/QA process

Modeling and prototyping

- Simplified version of the system for evaluation with end users or customer
- Evolutionary vs. throw-away prototypes
  - Evolutionary – get requirements right, but no deliverables
  - Throw-away – clarify requirements, but misleading (leaves out functionality)
- Horizontal vs. vertical prototypes
  - Horizontal prototype: UI
    - Validate the requirements
    - Vertical prototype: a complete use case
  - Vertical prototype: subset of functionality
    - Use case
    - Functional requirement
    - Project risk

Formal methods

- Guarantee complete coverage by a test suite
- Checks for deadlocks/livelocks
- System logic is specified using predicates in linear temporal logic - automata theory
- Exhaustive, partial, and sampling techniques
- Massive message passing, near real time operation
- Point out to missing tests
- Very useful in protocols and HW related systems, less so in UI centric systems

Defect management

- Track all known bugs – document and maintain status
- Assign a number to every bug
- Manage the list (add, merge, split, delete)
- Assign owner to every bug and someone who has to follow-up
- Great tools – Bugzilla, FogBugz (many other available)
- Accessible to developers, testers, management, end users
- Issue resolution – non reproducible bugs
- Critical in medium to large projects

Fault tolerance

- How do we achieve it in software?
  - Multi-process
  - Watchdogs
  - Graceful shutdown – detection of faults
  - Multi-server
    - Take over
    - Hand over
    - Cluster and dispatcher
  - Physical redundancy
- Data corruption
  - Rollbacks
  - CRC
- Atomic transactions
- Recovery modules – prepare for the worst

Agenda

- Software QA
  - SQA techniques
  - Verification
  - Principles of testing
- Debugging
- Testing
  - Component based testing
  - System/structure testing

Debugging

- Definition: Finding faults from an unplanned failure
- Correctness debugging: determine and repair deviations from specified functional requirements
- Performance debugging: address deviation from non-functional requirements
- Debugging requires skill and experience

Debugging Activities

- Fault search (finding the existence)
  - Unpredictable, costly
  - Should be replaced by other techniques wherever possible
- Fault location (a fault is found)
  - Can and should be done in a systematic manner
  - Use tool assistance
- Fault repair
  - May introduce new faults

Debugging Don’ts

- Panic
- Locate faults by guessing without a rational basis for the guess - “Superstition debugging”
  - Do not confuse with “educated guess”
- Fix the symptom without locating the bug
  - Trying to avoid the bug by avoiding “problematic input” will make it appear later
- Let your team member hang out to dry
- Become depressed if you can’t find the bug

- This can be avoided by staying in control with systematic techniques
- Individual programmer statistics: 20:1 differences in effectiveness at debugging!

Steps in locating a fault

- Stabilize the failure
  - Determine symptom: observed output ≠ expected output
  - Determine inputs on which the failure occurs predictably
- Simplify the failure
  - Experiment with simpler data
  - See if the failure still happens
- Progressively reduce the scope of the fault
  - Some form of binary search works best
  - Weighted binary trees
- The “scientific method” works for all of the above
  - This is how science is produced since ancient days
  - Elaborate “design of experiment” techniques in manufacturing QA

The “scientific method”

Steps:

1. Examine data that reveal a phenomenon
2. Form a hypothesis to explain the data
3. Design an experiment that can confirm or disprove the hypothesis
4. Perform the experiment and either adopt or discard the hypothesis
5. Repeat until a satisfactory hypothesis is found and adopted

Example:

- Hypothesis: the memory access violation occurs in module A
- Experiment: run with a breakpoint at the start of module A, or insert a print statement at the start of A

Example:

- Hypothesis: the fault was introduced by Joe
- Experiment: use version control to get previous version and check correctness.

Locating a fault

Example

- IntBag: contains unordered integers, some of which may be equal
  E.g. {12, 5, 9, 9, 9, -4, 100}
- Suppose that the following failure occurs for an IntBag object:
  Methods invoked (“input”):
  insert(5); insert(10); insert(10); insert(10); extract(10); extract(10); total()
- Failure symptom:
- expected return value for total() = 15; observed value = 5

- Debugging strategy
  - What would be an effective way to locate the fault?

Using debuggers

- Use one!
- Use debugger features:
- Control: step into, step over, continue, run to cursor, set variable, ...
- Observation: breakpoints, watches (expression displays)
- Advanced: stack, memory leaks, ...

- Combine debugging with your own reasoning about correctness
- Example

  Infer that i should ==n after “for (i = 2; i < n; i ++) {…}”

  Although some side effects may overwrite i

Step through the code with a debugger

  - Watches on
  - Assertions enabled

Fixing faults

Make sure you understand the problem before fixing it

- As opposed to patching up the program to avoid the symptom
- Fix the problem, not the symptom

- Always perform regression tests after the fix
  - I.e., use debugging in combination with systematic testing

- Always look for similar faults
  - E.g., by including the fault type on a review checklist

Tips

- Avoid debugging as much as you can!
  - Enlightened procrastination
  - When you have to debug, debug less and reason more
- Talk to others about the failure
- See debugging as opportunity
  - Learn about the program
  - Learn about likely kinds of mistakes
  - Learn about how to fix errors
- It will take as long as it will take

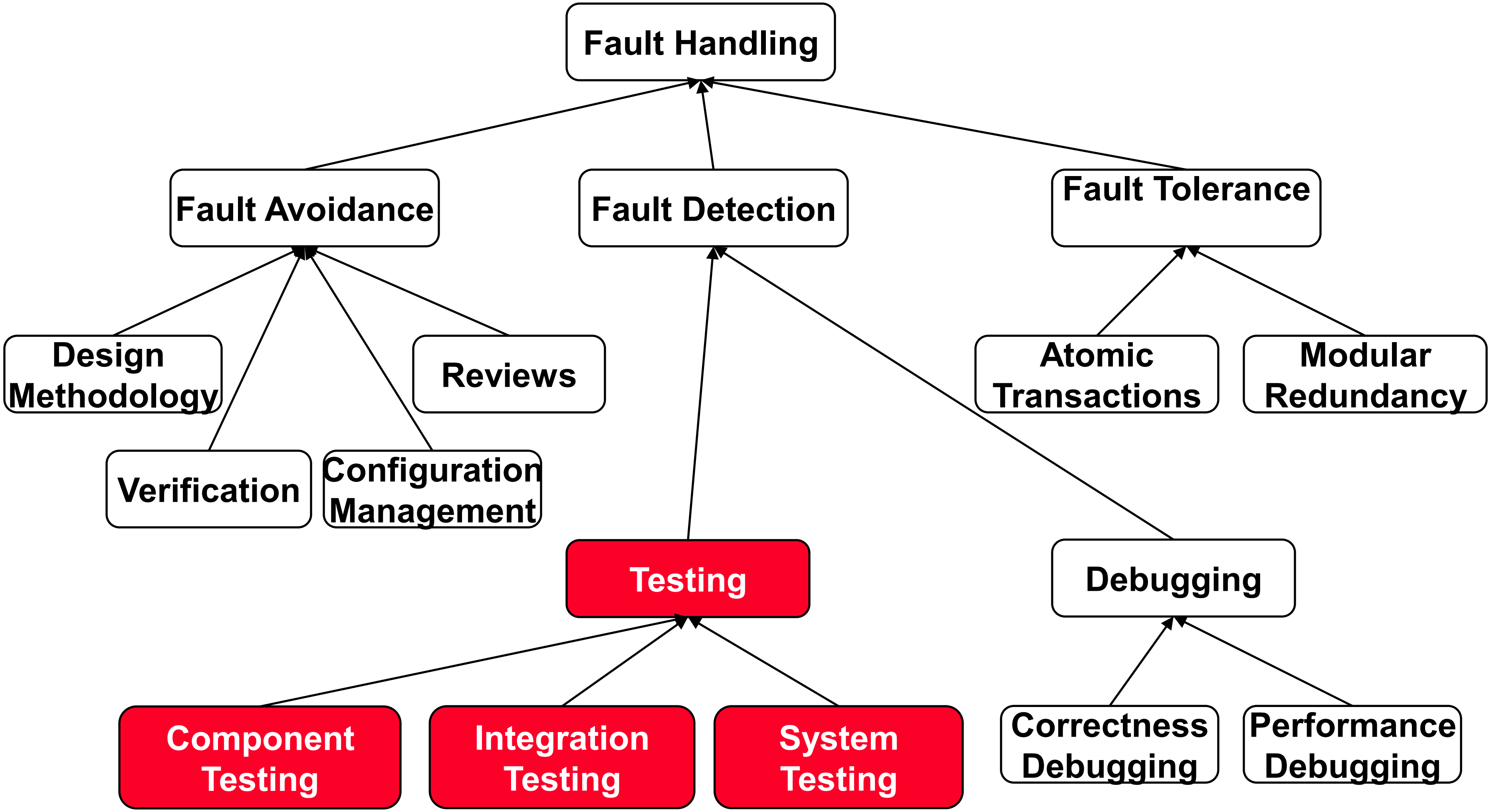
Agenda

- Software QA
  - SQA techniques
  - Verification
  - Principles of testing
- Debugging
- Testing
  - Component based testing
  - System/structure testing

Testing

- About 40% of the expenditure
- Do it well, do it once (or twice)
- Testing does not prove there are no bugs – just the we can’t find them
- Testing is never good enough
- Common engineering practice (cars)
- Hard to get used to
- Can save a lot of time and money

Fault Handling Techniques



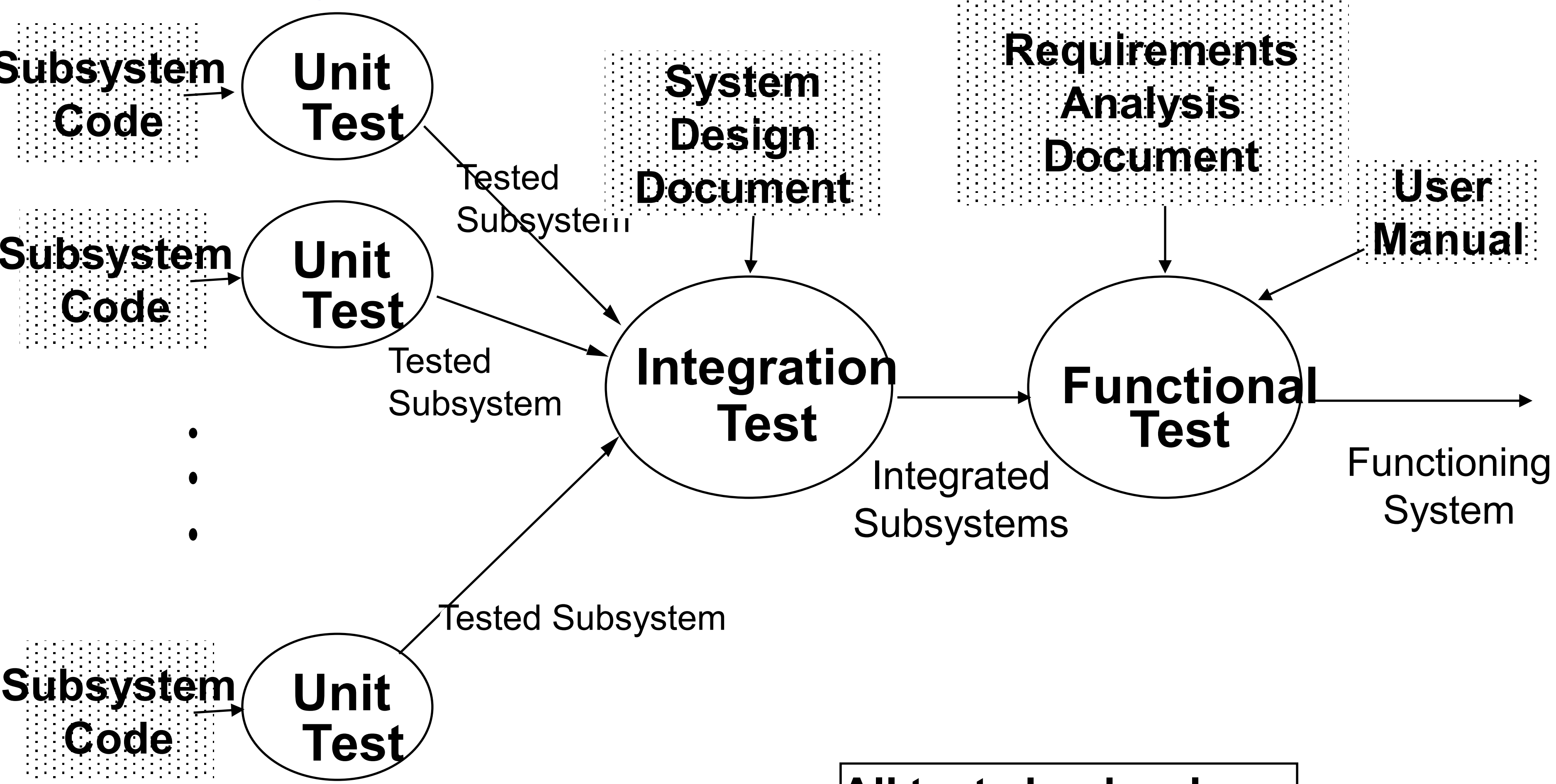
Testing takes creativity

- Testing often viewed as dirty work.
- To develop an effective test, one must have:
    - Detailed understanding of the system
    - Knowledge of the testing techniques
    - Skill to apply these techniques in an effective and efficient manner
- Testing is done best by independent testers
    - We often develop a certain mental attitude that the program should run in a certain way when in fact it does not.
- A program often does not work when tried by somebody else
    - Don't let this be the end-user

Testing takes creativity

- How do you test
  - Google
  - Grand Theft Auto

Testing Activities



Subsystem Code

Unit Test

Tested Subsystem

Subsystem Code

Unit Test

Tested Subsystem

Subsystem Code

Unit Test

Tested Subsystem

System Design Document

Integration Test

Integrated Subsystems

Requirements Analysis Document

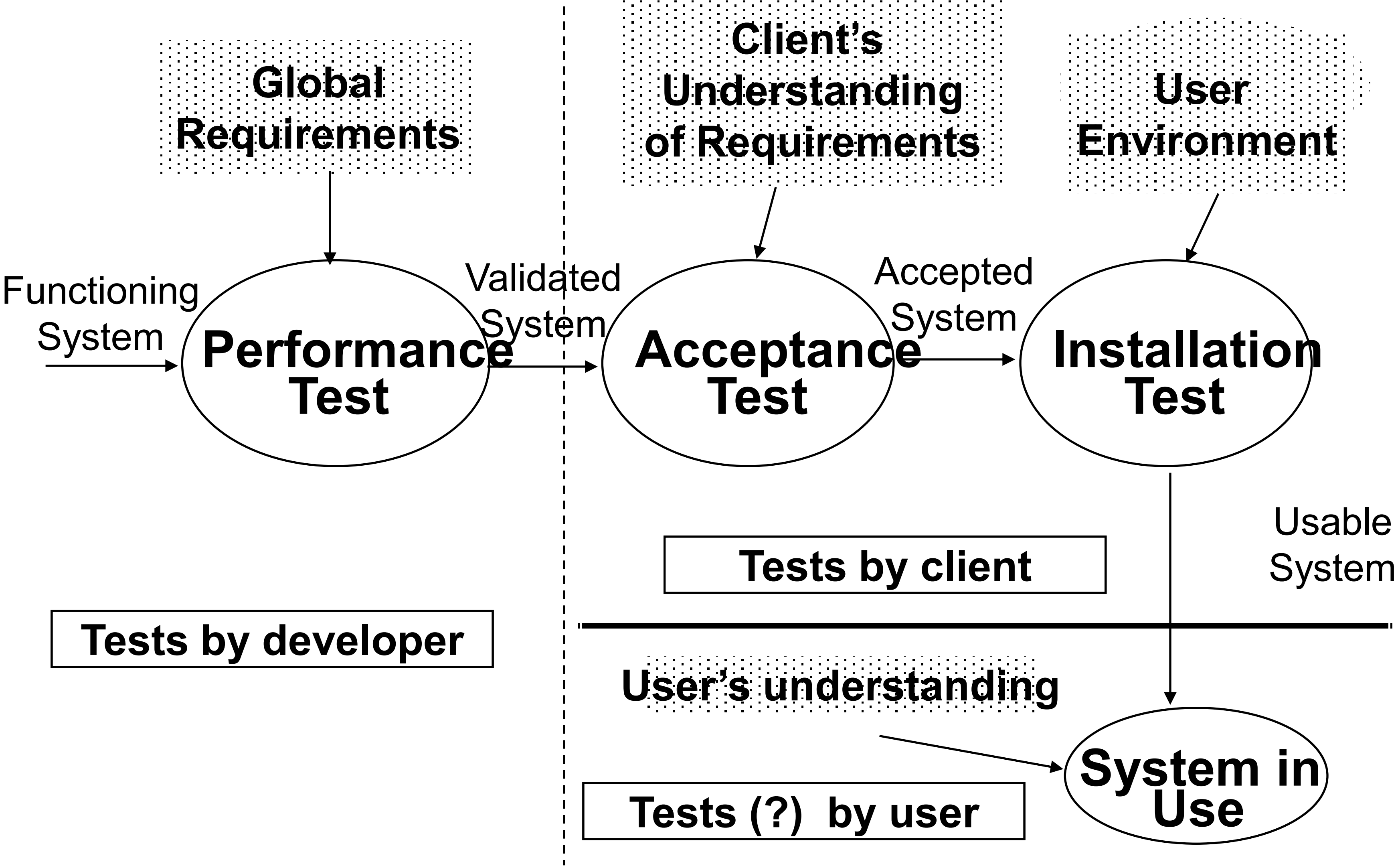
User Manual

Functional Test

Functioning System

All tests by developer

Testing Activities



Global Requirements

Client's Understanding of Requirements

User Environment

Functioning System

Performance Test

Validated System

Acceptance Test

Accepted System

Installation Test

Usable System

Tests by client

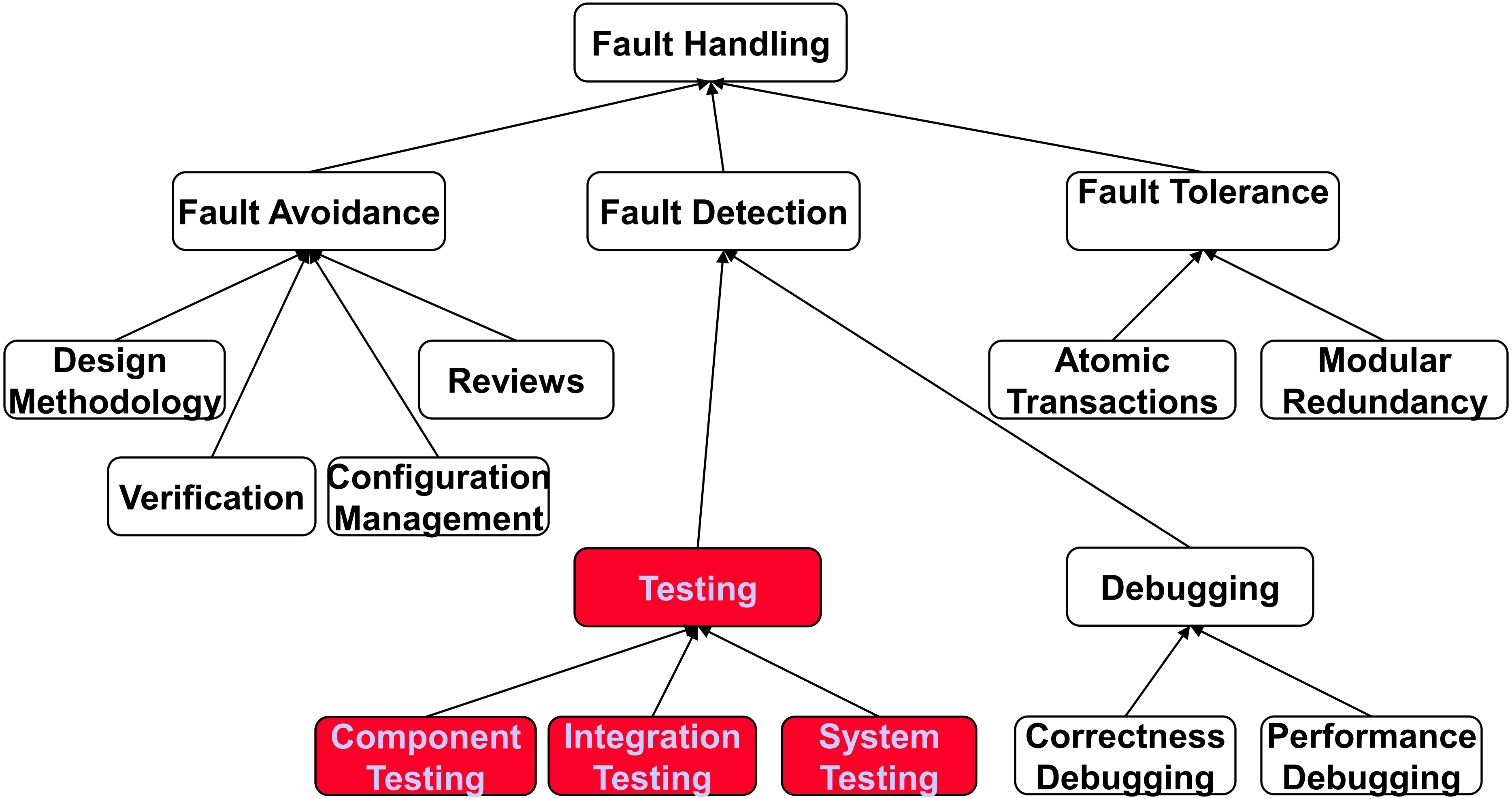
Tests by developer

User's understanding

System in Use

Tests (?) by user

Quality Assurance encompasses Testing



Fault Handling

Fault Avoidance

Fault Detection

Fault Tolerance

Design Methodology

Reviews

Verification

Configuration Management

Testing

Debugging

Atomic Transactions

Modular Redundancy

Component Testing

Integration Testing

System Testing

Correctness Debugging

Performance Debugging

What is a test?

Name and number:
Test items:
Input:
Expected Output:
Environmental Needs:
Special Procedural Requirements:
Inter-case Dependencies:

Undocumented tests = non existent tests!

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 subsystems

Types of testing

- System Testing:
  - The entire system
  - Carried out by developers
  - <u>Goal:</u> 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
  - <u>Goal:</u> 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
- 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)

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)

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

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

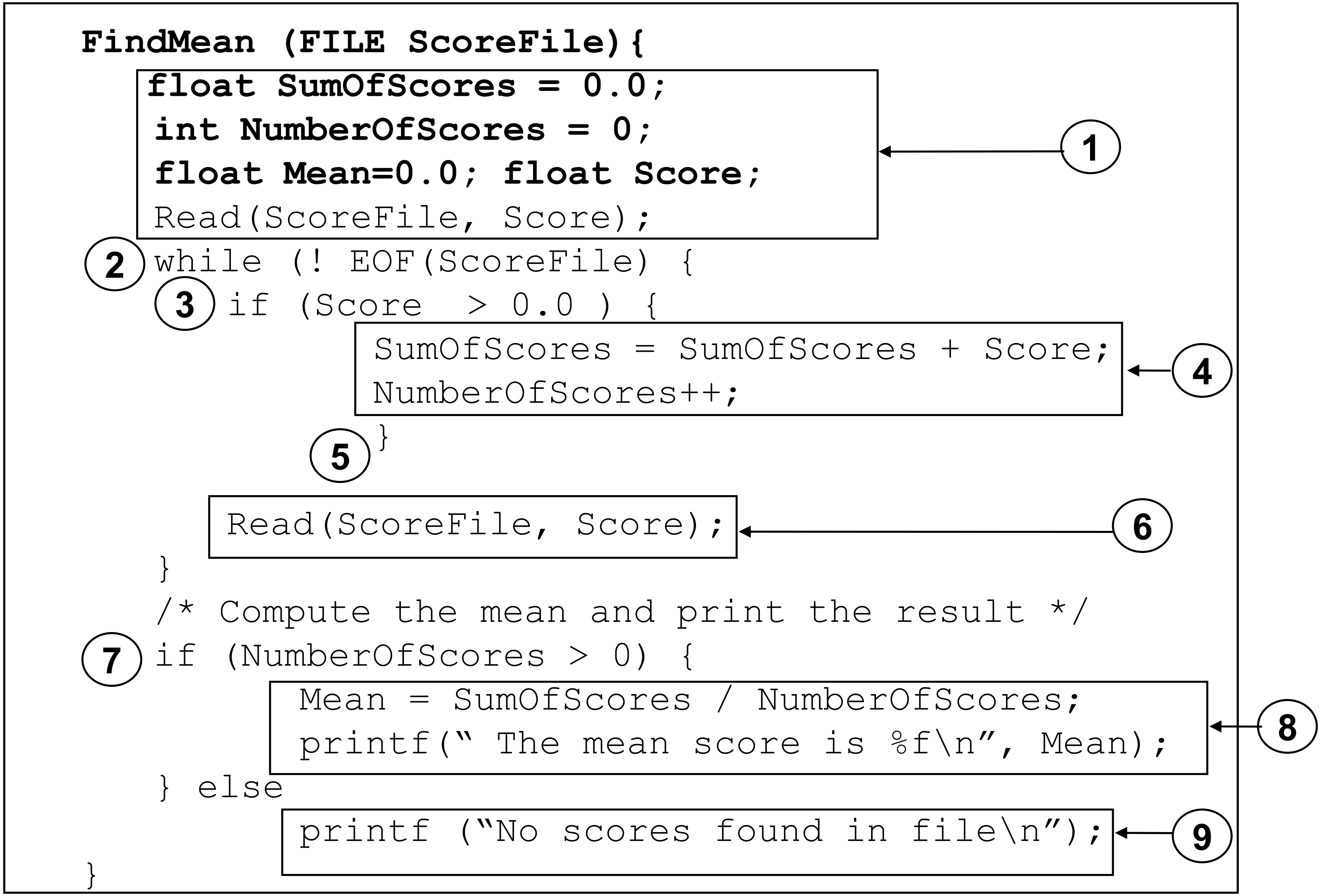
White-box Testing (Continued)

- Statement Testing (Algebraic Testing): Test single statements (Choice of operators in polynomials, etc)
- 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

White-box Testing Example

```
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");
}
```

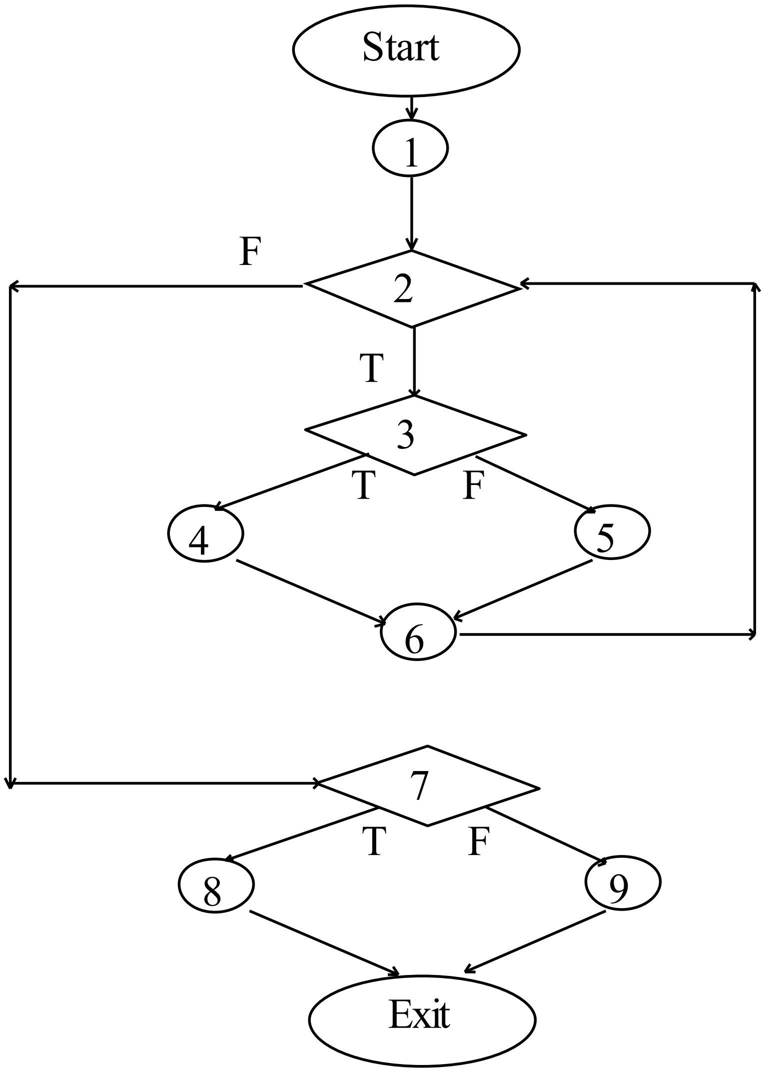
White-box Testing : Determining the Paths



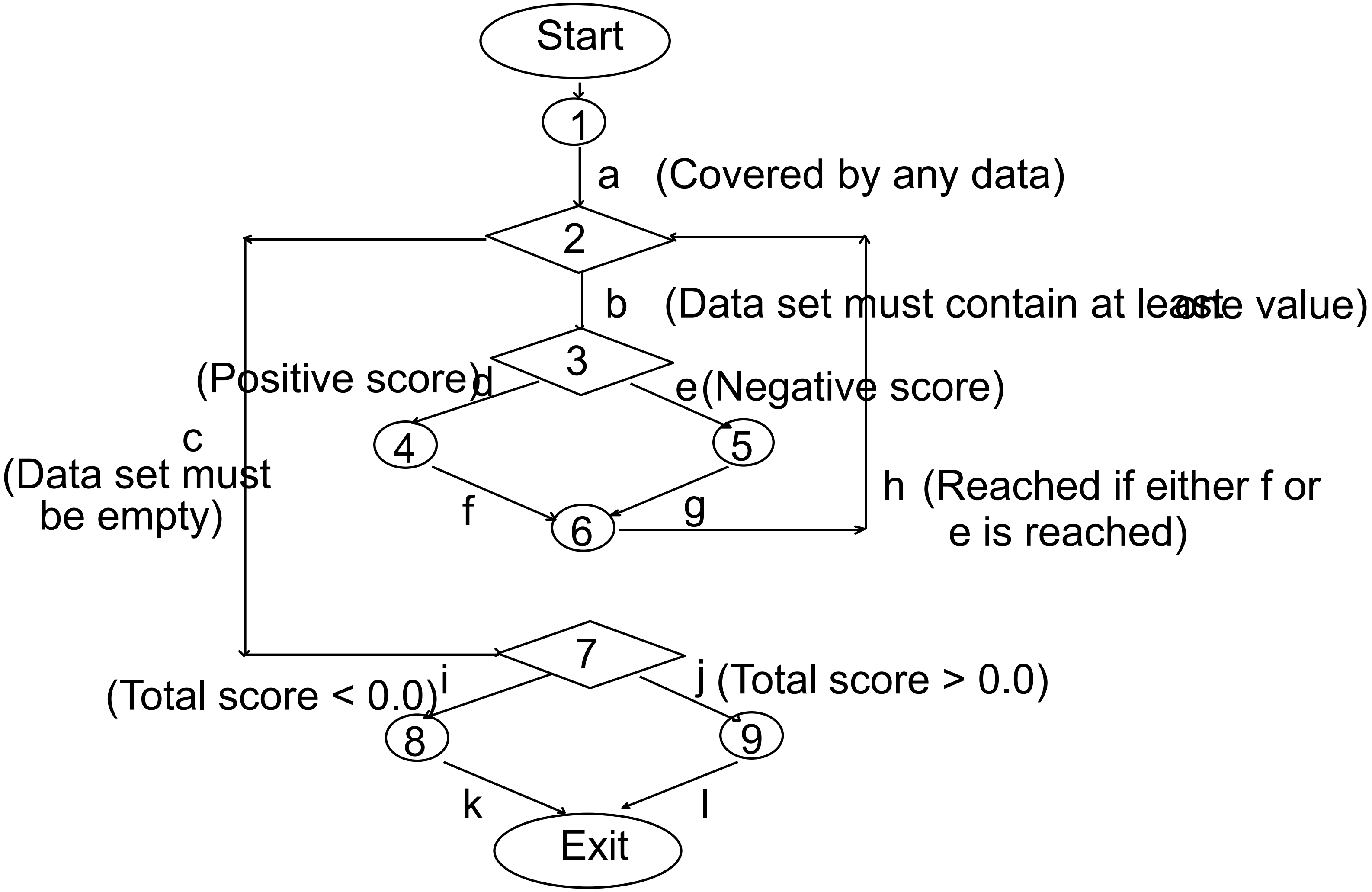
```
FindMean (FILE ScoreFile){
    float SumOfScores = 0.0;
    int NumberOfScores = 0;
    float Mean=0.0; float Score;
    Read(ScoreFile, Score);
    while (! EOF(ScoreFile) {
        if (Score  > 0.0 ) {
                SumOfScores = SumOfScores + Score;
                NumberOfScores++;
                }

        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”);
}
```

Constructing the Logic Flow Diagram



Finding the Test Cases



Test Cases

- Test case 1 : ? (To execute loop exactly once)
- Test case 2 : ? (To skip loop body)
- Test case 3: ?,? (to execute loop more than once)

These 3 test cases cover all control flow paths

Comparison of White & Black-box Testing

- 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

The 4 Testing Steps

1. Select what has to be measured
   - Completeness of requirements
   - Code tested for reliability
   - Design tested for cohesion
2. Decide how the testing is done
   - Code inspection
   - Proofs
   - Black-box, white box
   - Select integration testing strategy (big bang, bottom up, top down, sandwich)

3. Develop test cases
   - A test case is a set of test data or situations that will be used to exercise the unit (code, module, system) being tested or about the attribute being measured
4. Create the test oracle
   - An oracle contains of the predicted results for a set of test cases
   - The test oracle has to be written down before the actual testing takes place

Guidance for Test Case Selection

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

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

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 test 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

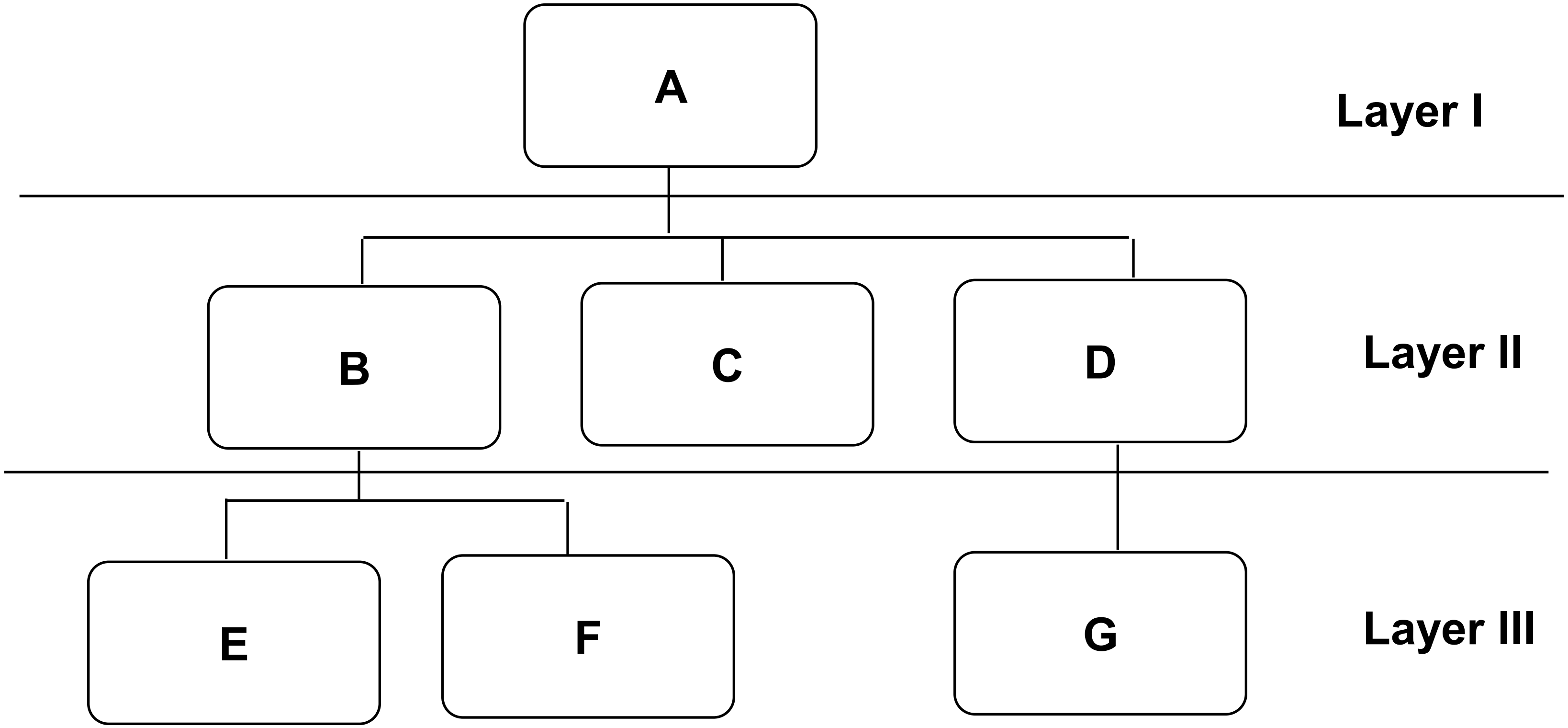
Agenda

- Software QA
  - SQA techniques
  - Verification
  - Principles of testing
- Debugging
- Testing
  - Component based testing
  - System/structure testing

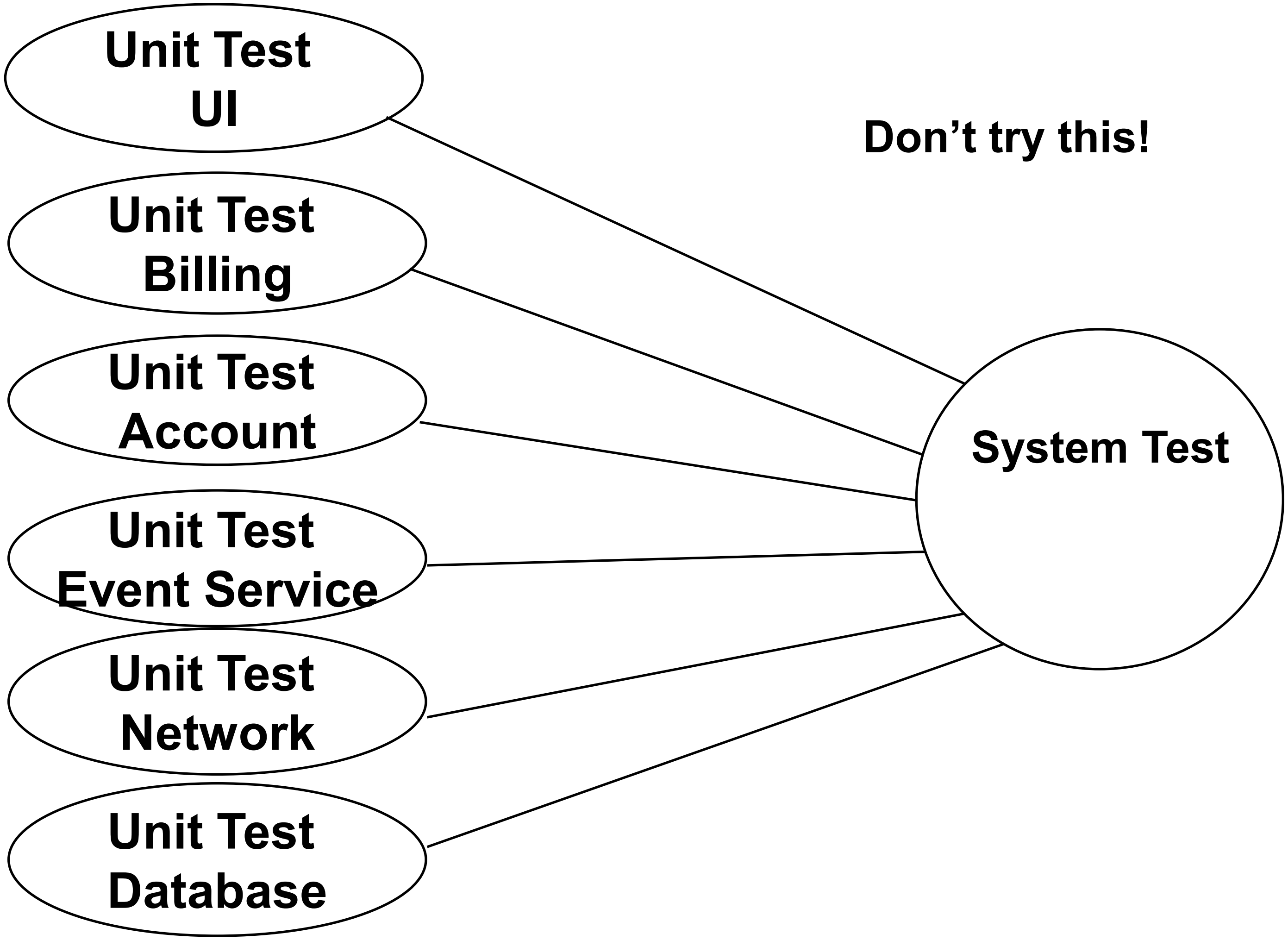
Component-Based Testing Strategy

- The entire system is viewed as a collection of subsystems (sets of classes) determined during the system and object design.
- The order in which the subsystems are selected for testing and integration determines the testing strategy
  - Big bang integration (Non-incremental)
  - Bottom up integration
  - Top down integration
  - Sandwich testing
  - Variations of the above
- For the selection use the system decomposition from the System Design

Example: Three Layer Call Hierarchy



Integration Testing: Big-Bang Approach



Unit Test
UI

Unit Test
Billing

Unit Test
Account

Unit Test
Event Service

Unit Test
Network

Unit Test
Database

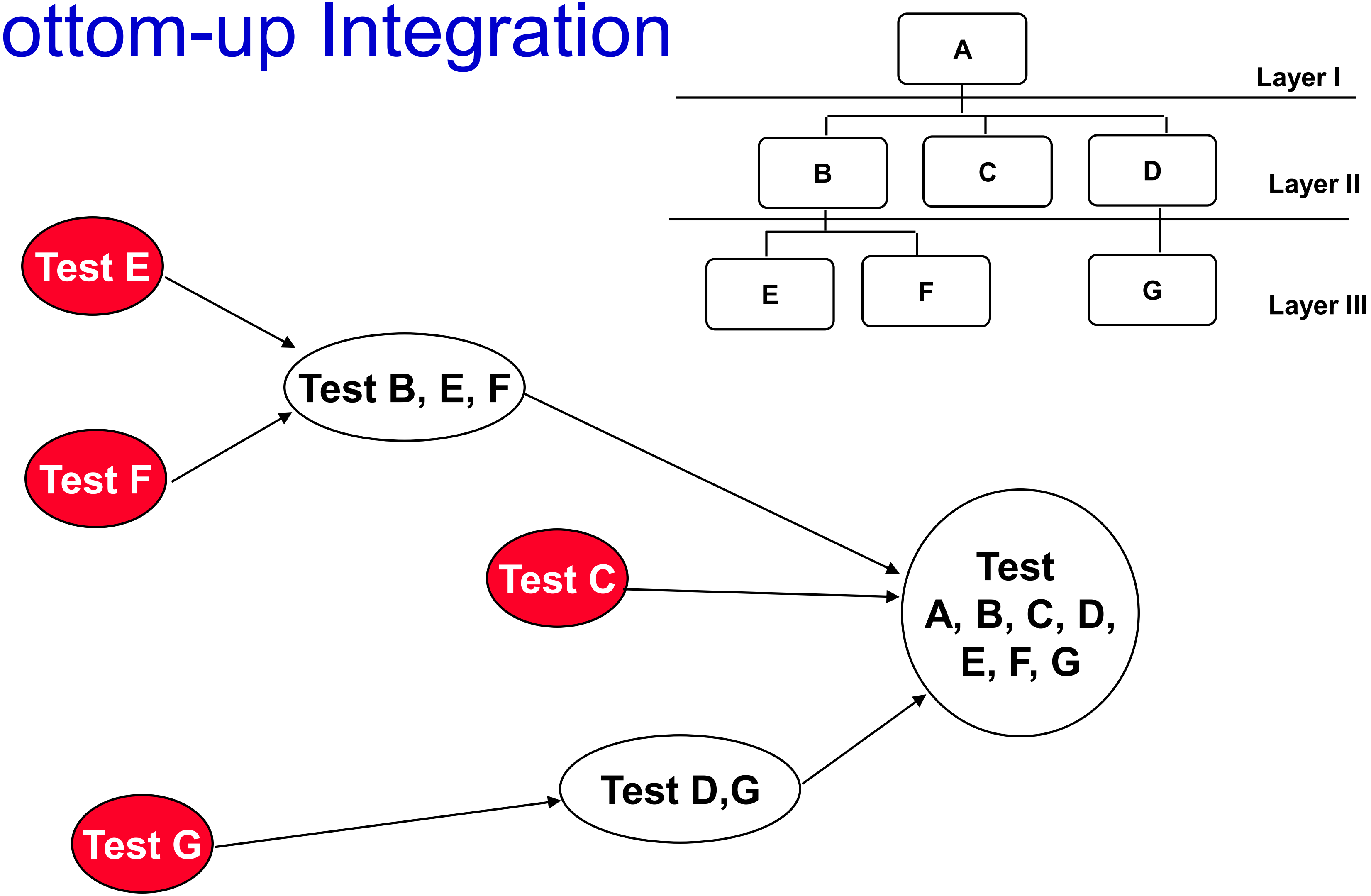
System Test

Don't try this!

Bottom-up Testing Strategy

- The subsystem in the lowest layer of the call hierarchy are tested individually
- Then the next subsystems are tested that call the previously tested subsystems
- This is done repeatedly until all subsystems are included in the testing
- Special program needed to do the testing, *Test Driver:*
  - A routine that calls a particular subsystem and passes a test case to it
  - Drivers may be tailored for specific tests

Bottom-up Integration



A

Layer I

B

C

D

Layer II

E

F

G

Layer III

Test E

Test F

Test B, E, F

Test C

Test G

Test D,G

Test
A, B, C, D,
E, F, G

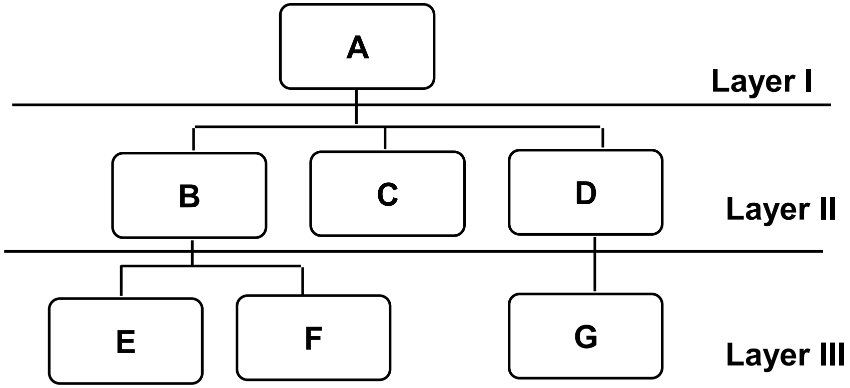
Pros and Cons of bottom up integration testing

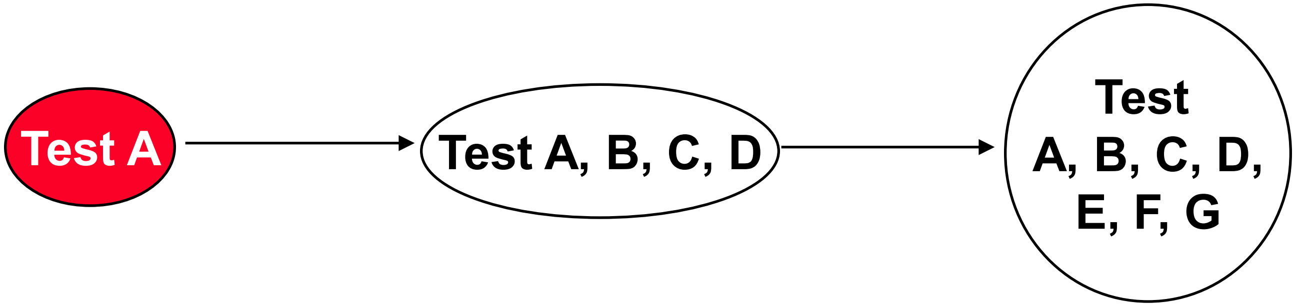
- Bad for functionally decomposed systems:
  - Tests the most important subsystem last
- Useful for integrating the following systems
  - Object-oriented systems
  - real-time systems
  - systems with strict performance requirements

Top-down Testing Strategy

- Test the top layer or the controlling subsystem first
- Then combine all the subsystems that are called by the tested subsystems and test the resulting collection of subsystems
- Do this until all subsystems are incorporated into the test
- Special program is needed to do the testing, *Test stub :*
  - A program or a method that simulates the activity of a missing subsystem by answering to the calling sequence of the calling subsystem and returning back fake data
  - Stubs may be tailored to specific tests

Top-down Integration Testing





Test A

Test A, B, C, D

Test A, B, C, D, E, F, G

Layer I

Layer I + II

All Layers

Pros and Cons of top-down integration testing

- Test cases can be defined in terms of the functionality of the system (functional requirements)

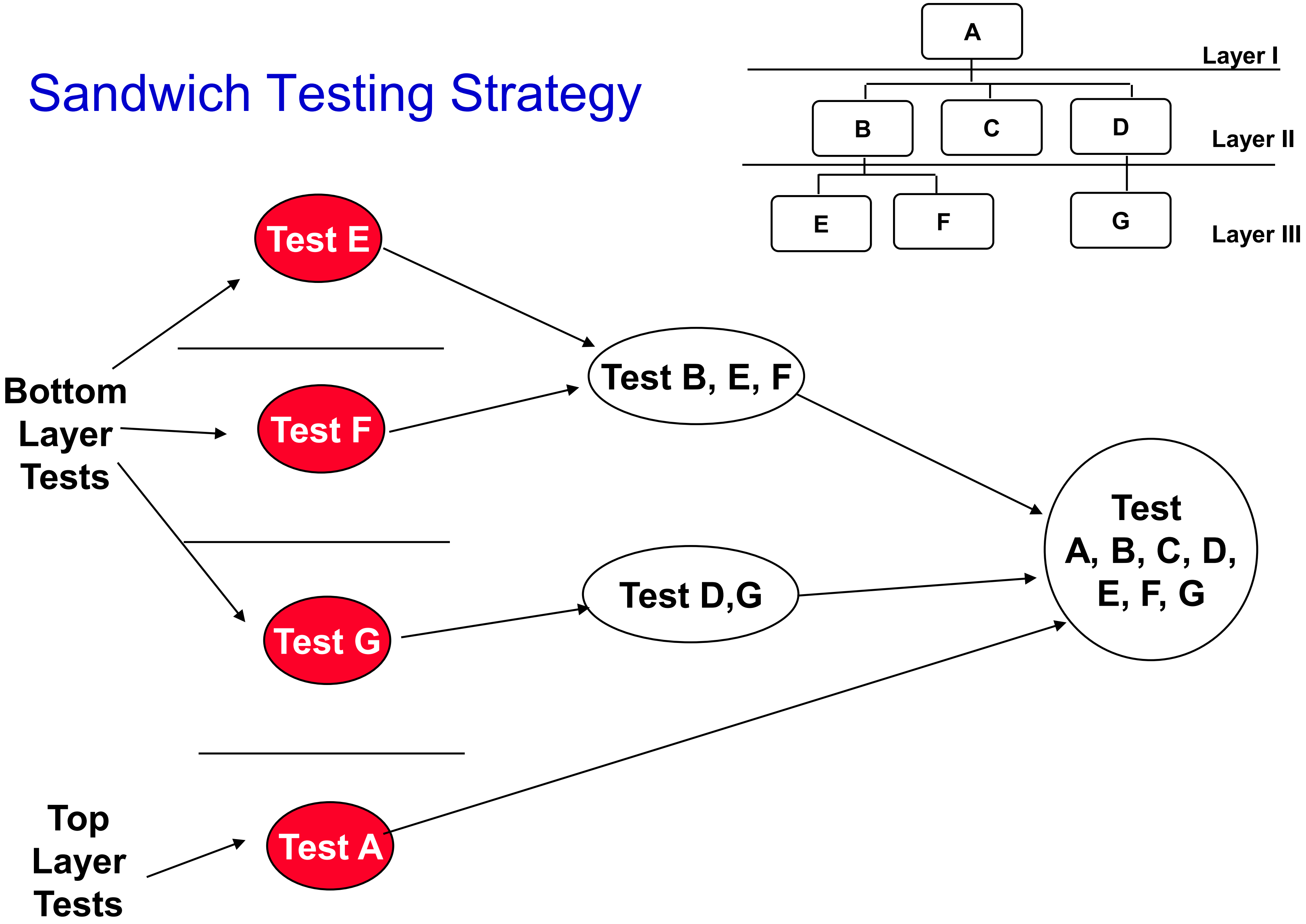
- Writing stubs can be difficult: Stubs must allow all possible conditions to be tested.

- Possibly a very large number of stubs may be required, especially if the lowest level of the system contains many methods.

Sandwich Testing Strategy

- Combines top-down strategy with bottom-up strategy
- *The system is view as having three layers*
  - A target layer in the middle
  - A layer above the target
  - A layer below the target
  - Testing converges at the target layer
- How do you select the target layer if there are more than 3 layers?
  - Heuristic: Try to minimize the number of stubs and drivers

Sandwich Testing Strategy



A

Layer I

B

C

D

Layer II

E

F

G

Layer III

Bottom Layer Tests

Test E

Test F

Test G

Test B, E, F

Test D,G

Top Layer Tests

Test A

Test A, B, C, D, E, F, G

Pros and Cons of Sandwich Testing

- Top and Bottom Layer Tests can be done in parallel

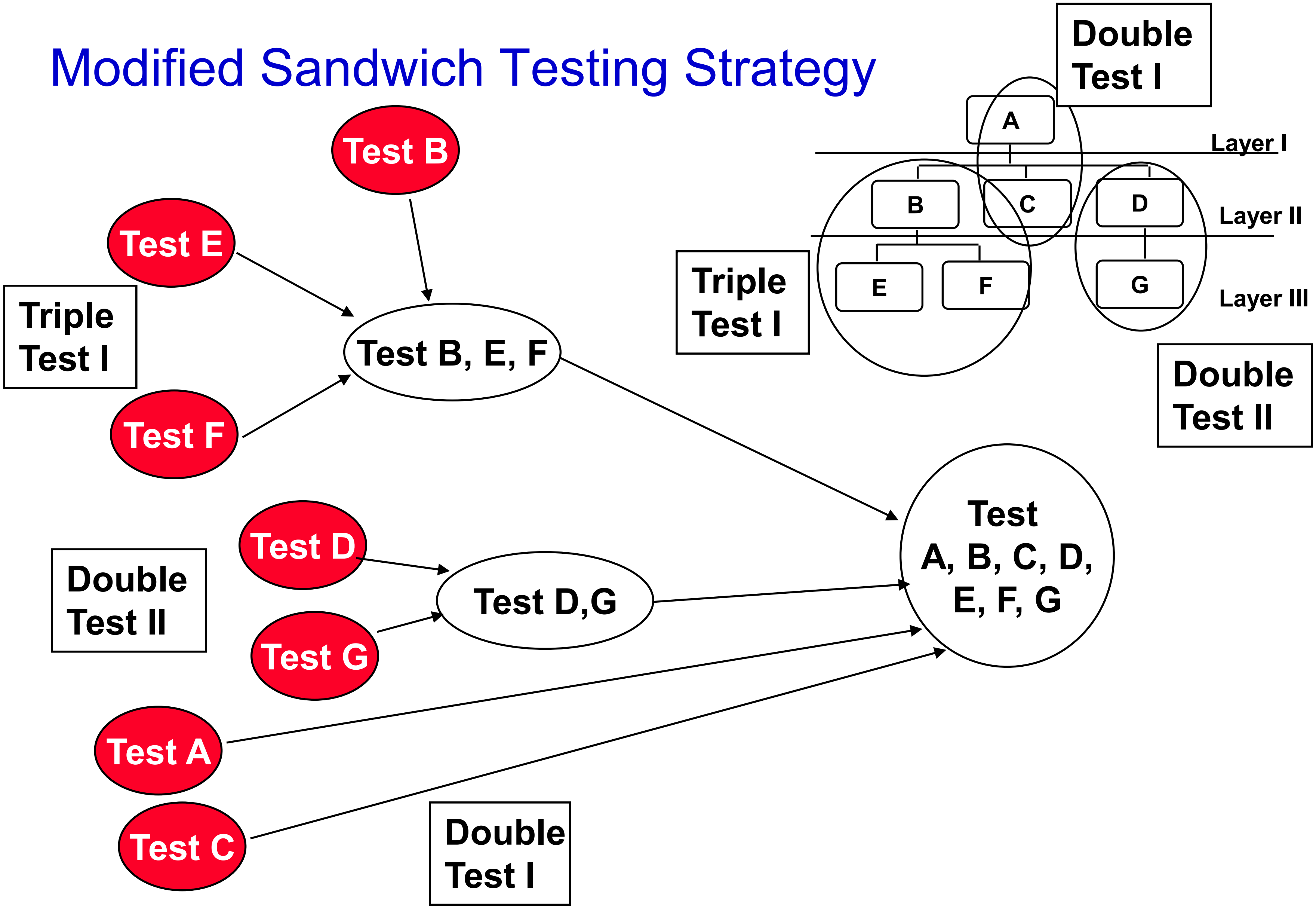
- Does not test the middle layers thoroughly before integration

- Solution: Modified sandwich testing strategy

Modified Sandwich Testing Strategy

- Test in parallel:
  - Middle layer with drivers and stubs
  - Top layer with stubs
  - Bottom layer with drivers
- Test in parallel:
  - Top layer accessing middle layer (top layer replaces drivers)
  - Bottom accessed by middle layer (bottom layer replaces stubs)

Modified Sandwich Testing Strategy



Steps in Component-Based Testing

1. Based on the integration strategy, *select a component* to be tested. **Unit test** all the classes in the component.
2. Put selected component together; do any **preliminary fix-up** necessary to make the integration test operational (drivers, stubs)
3. Do **functional testing***:* Define test cases that exercise all uses cases with the selected component
4. Do **structural testing***:* Define test cases that exercise the selected component
5. Execute **performance tests**
6. **Keep records** of the test cases and testing activities.
7. Repeat steps 1 to 7 until the full system is tested.

The primary *goal of integration testing is to identify errors* in the (current) component configuration.

Which Integration Strategy should you use?

- Factors to consider
  - Amount of test harness (stubs &drivers)
  - Location of critical parts in the system
  - Availability of hardware
  - Availability of components
  - Scheduling concerns
- Bottom up approach
  - good for object oriented design methodologies
  - Test driver interfaces must match component interfaces
  - ...Top-level components are usually important and cannot be neglected up to the end of testing
  - Detection of design errors postponed until end of testing
- Top down approach
  - Test cases can be defined in terms of functions examined
  - Need to maintain correctness of test stubs
  - Writing stubs can be difficult

Agenda

- Software QA
  - SQA techniques
  - Verification
  - Principles of testing
- Debugging
- Testing
  - Component based testing
  - System/structure testing

System Testing

- Functional Testing
- Structure Testing
- Performance Testing
- Acceptance Testing
- Installation Testing

Impact of requirements on system testing:

  - The more explicit the requirements, the easier they are to test
  - Quality of use cases determines the ease of functional testing
  - Quality of subsystem decomposition determines the ease of structure testing
  - Quality of nonfunctional requirements and constraints determines the ease of performance tests

Functional Testing

Essentially the same as black box testing

- Goal: Test functionality of system
- Test cases are designed from the requirements analysis document (better: user manual) and centered around requirements and key functions (use cases)
- The system is treated as black box.
- Unit test cases can be reused, but in end user-oriented new test cases have to be developed as well.

Test case example:

The use case:

Name: `PurchaseTicket`

Participating actor: `Passenger`

Entry condition:

- `Passenger` standing in front of ticket distributor.
- `Passenger` has sufficient money to purchase ticket.

Exit condition:

- `Passenger` has ticket.

Event flow:

1. `Passenger` selects the number of zones to be traveled
2. Distributor displays the amount due.
3. `Passenger` inserts money, of at least the amount due
4. Distributor returns change if passenger inserted more money then needed
5. Distributor issues ticket
6. Passenger picks up change and ticket

Test case example:

The test case:

Name: `Purchase2Tickets`

Entry condition:

1. The `Passenger` is in front of ticket distributor.
2. `Passenger` has a $10 bill

Flow of events:

1. `Passenger` presses the zones buttons 2,4,1, and 2 (in succession).
2. `Distributor` displays $1.25, $2.50, $1, $1.25
3. `Passenger` inserts a $10 bill
4. `Distributor` returns one $5 bill, three $1 bills and three quarters and issues a zone 2 ticket
5. `Passenger` presses zone button 4.
6. `Passenger` inserts a three $1 bills
7. `Distributor` returns two quarters and issues a zone 4 ticket

Exit condition:

- `Passenger` has zone 2 ticket and zone 4 ticket

Structure Testing

- *Essentially the same as white box testing*
- Goal: Cover all paths in the system design
  - Exercise all input and output parameters of each subsystems
  - Exercise all subsystems and all calls (each subsystem is called at least once and every subsystem is called by all possible callers)
  - Use conditional and iteration testing as in unit testing

Performance Testing

- Stress Testing
  - Stress limits of system (maximum # of users, peak demands, extended operation)
- Volume testing
  - Test what happens if large amounts of data are handled
- Configuration testing
  - Test the various software and hardware configurations
- Compatibility test
  - Test backward compatibility with existing systems
- Security testing
  - Try to violate security requirements
- Timing testing
  - Evaluate response times and time to perform a function
- Environmental test
  - Test tolerances for heat, humidity, motion, portability
- Quality testing
  - Test reliability, maintainability & availability of the system
- Recovery testing
  - Tests system’s response to presence of errors or loss of data.
- Human factors testing
  - Tests user interface with user

Test Cases for Performance Testing

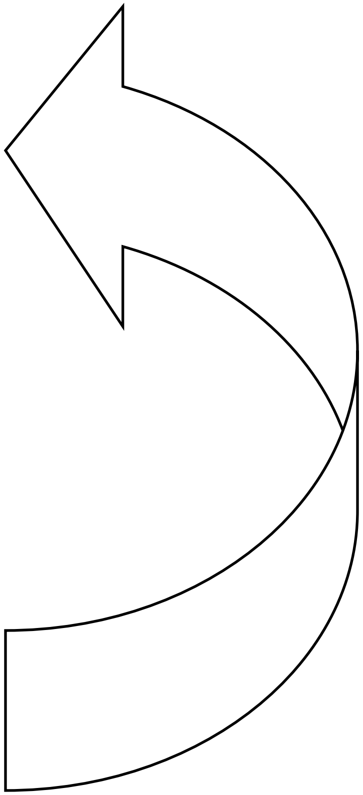
- **Push the (integrated) system to its limits.**
- **Goal: Try to break a subsystem**
- **Test how the system behaves when overloaded.**
  - Can bottlenecks be identified? (First candidates for redesign in the next iteration
- **Try unusual orders of execution**
  - Call a receive() before send()
- **Check the system's response to large volumes of data**
  - If the system is supposed to handle 1000 items, try it with 1001 items.
- **What is the amount of time spent in different use cases?**
  - Are typical cases executed in a timely fashion?

Acceptance Testing

- Goal: Demonstrate system is ready for operational use
  - Choice of tests is made by client/sponsor
  - Many tests can be taken from integration testing
  - Acceptance test is performed by the client, not by the developer.
- Many bugs typically found by the client after the system is in use. Therefore two kinds of additional tests:

- *Alpha test:*
  - Customer uses the software at the *developer's site.*
  - Software used in a controlled setting, with the developer always ready to fix bugs.
- *Beta test:*
  - Conducted at *sponsor's site* (developer is not present)
  - Software gets a realistic workout in target environment
  - Potential customer might get discouraged

Testing has its own Life Cycle



Establish the test objectives

Design the test cases

Write the test cases

Test the test cases

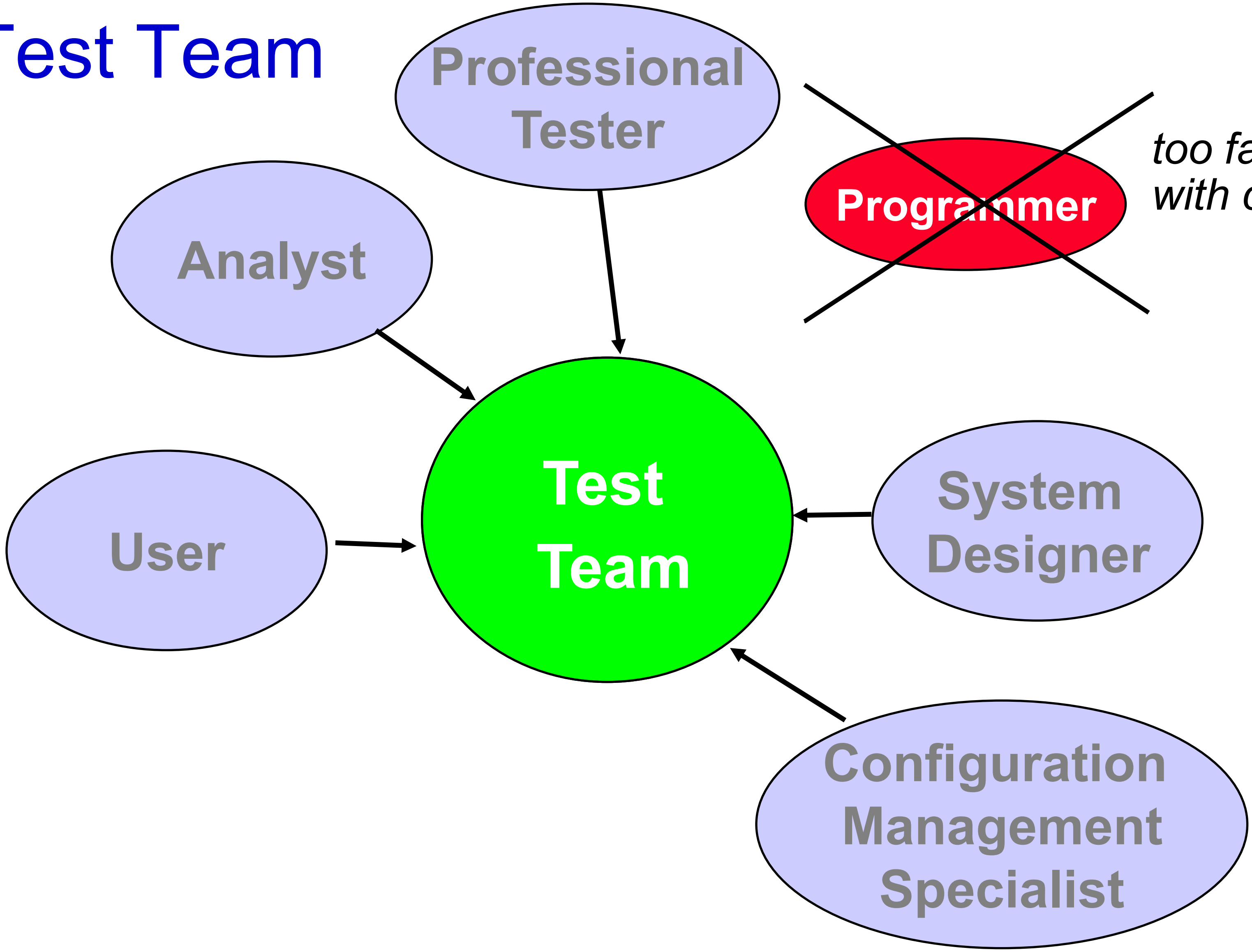
Execute the tests

Evaluate the test results

Change the system

Do regression testing

Test Team



too familiar with code

Summary

- Testing is still a black art, but many rules and heuristics are available
- Testing consists of component-testing (unit testing, integration testing) and system testing
- Design Patterns can be used for component-based testing
- Testing has its own lifecycle

Managing Testing

- Test plan – what is our plan?
  - Scope
  - Approach
  - Schedule
- Test case spec
  - Each test is documented
- Test incident report
- Test report summary – pass/fail + analysis

Test plan

1. Introduction
2. Relationship to other docs (RAD, SDD, ODD)
3. System overview
4. Features to be tested/not tested
5. Pass/fail criteria
6. General approach
7. Suspension/resumption
8. Resources
9. **Test cases – list of all tests (listed in the Test-case spec)**
10. Schedule

Regression testing

- A bug was discovered and fixed. What now?
  - New bugs were created
  - Old bugs rediscovered
- Regression testing
  - Dependent components
  - Risky use cases
  - Frequent use cases

Automated testing

- Extremely important in large projects
- Automatic execution (of tests, and checking results)
- Makes regression tests “cheap”
- Automatic test generation
- Easy for some application (HW), hard for others (GUI)
- JUnit