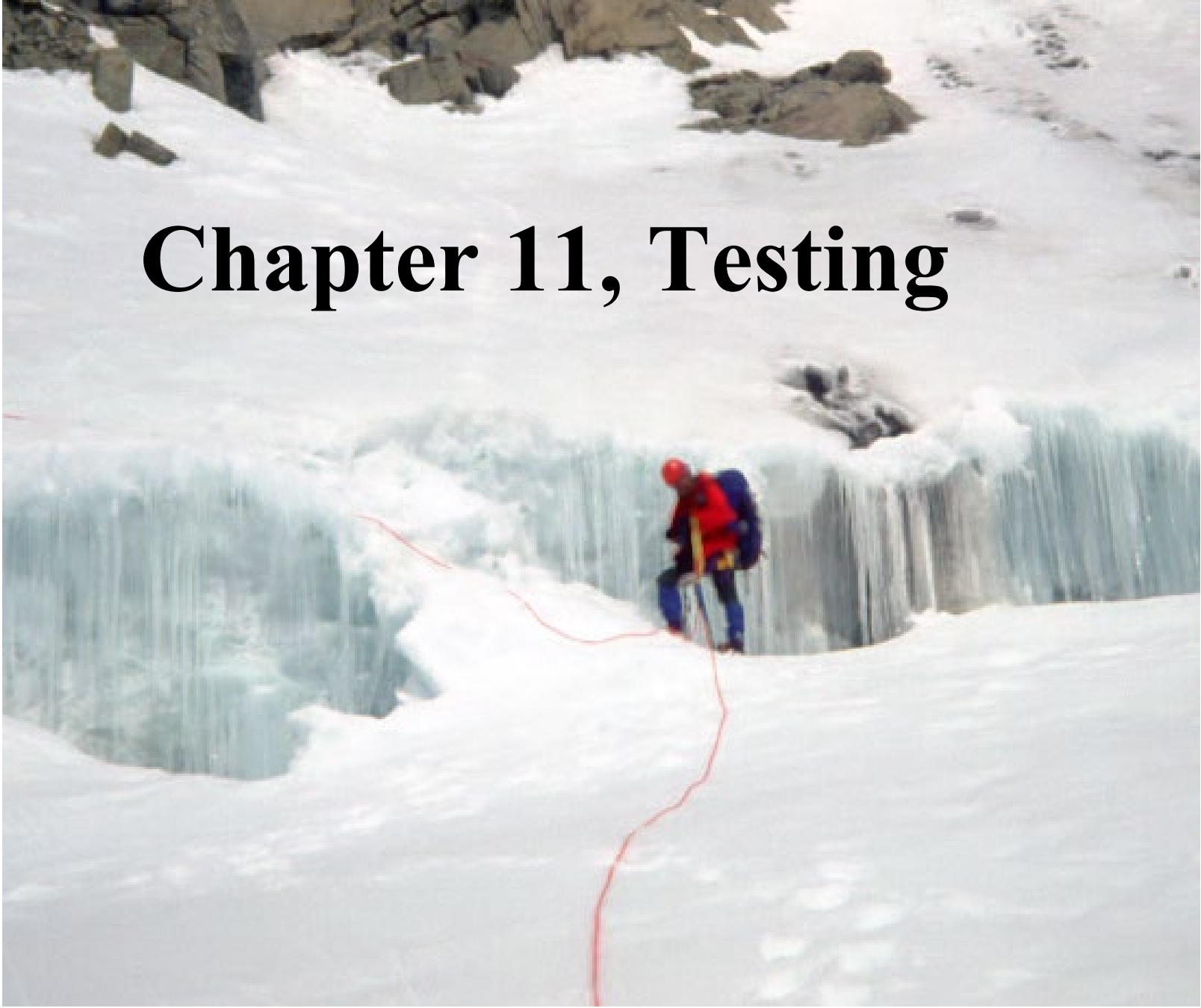


# Chapter 11, Testing

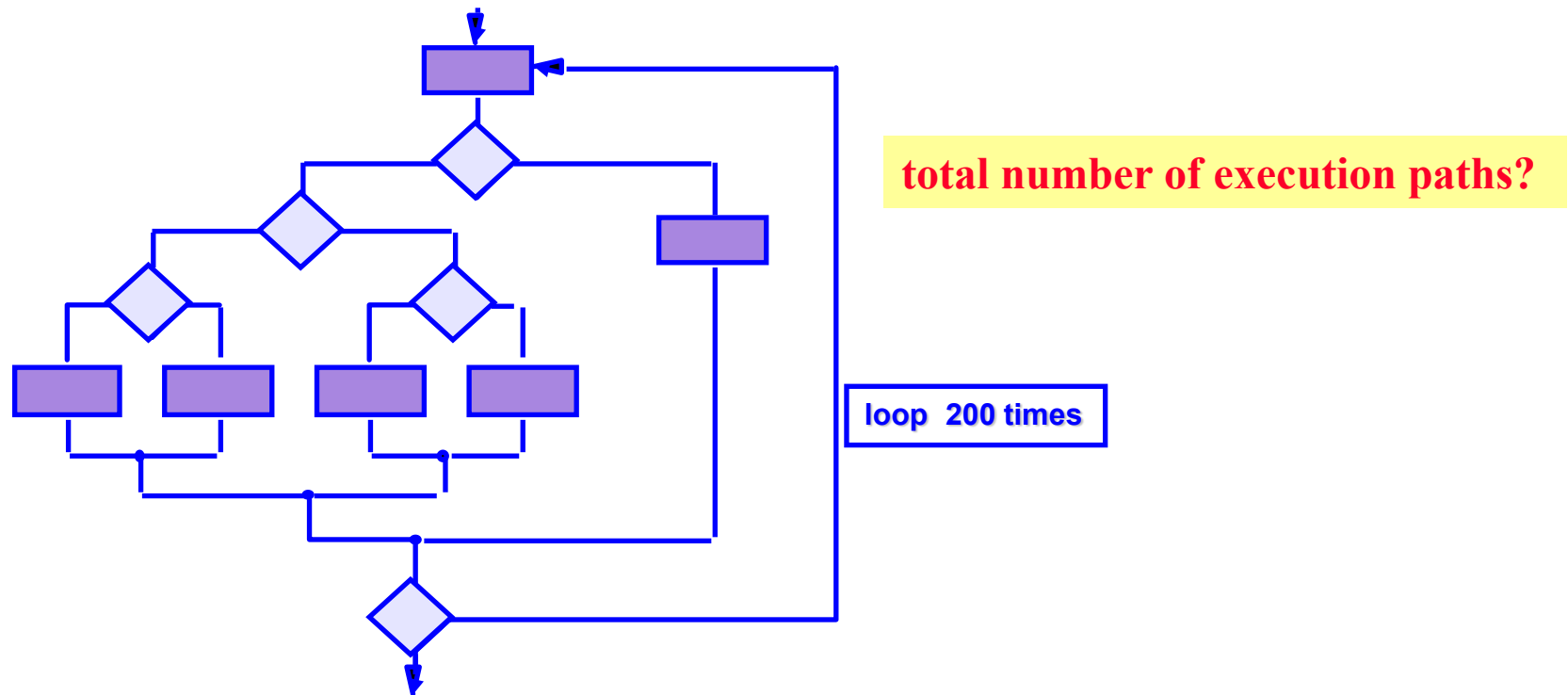


# *Outline*

- ◆ Terminology
- ◆ Types of errors
- ◆ Dealing with errors
- ◆ Quality assurance vs Testing
- ◆ Component Testing
  - ◆ **Unit testing**
  - ◆ **Integration testing**
- ◆ Testing Strategy
- ◆ Design Patterns & Testing
- ◆ System testing
  - ◆ **Function testing**
  - ◆ **Structure Testing**
  - ◆ **Performance testing**
  - ◆ **Acceptance testing**
  - ◆ **Installation testing**

# *Some Observations*

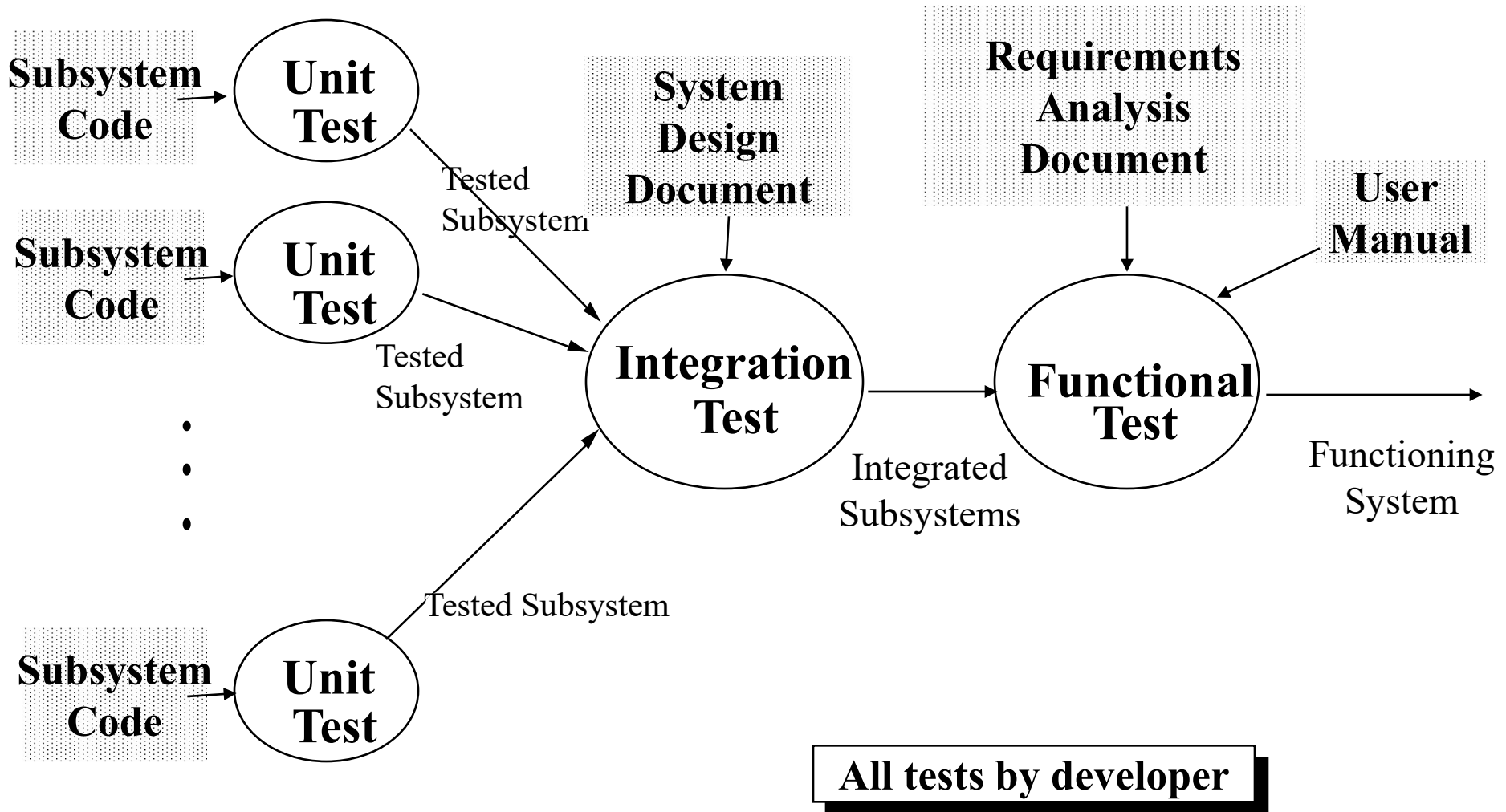
- ◆ It is impossible to completely test any nontrivial module or any system
  - ◆ Theoretical limitations: Halting problem ??
  - ◆ Practical limitations: Prohibitive in time and cost
- ◆ Testing can only show the presence of bugs, not their absence (Dijkstra)



# *Testing Takes **Creativity***

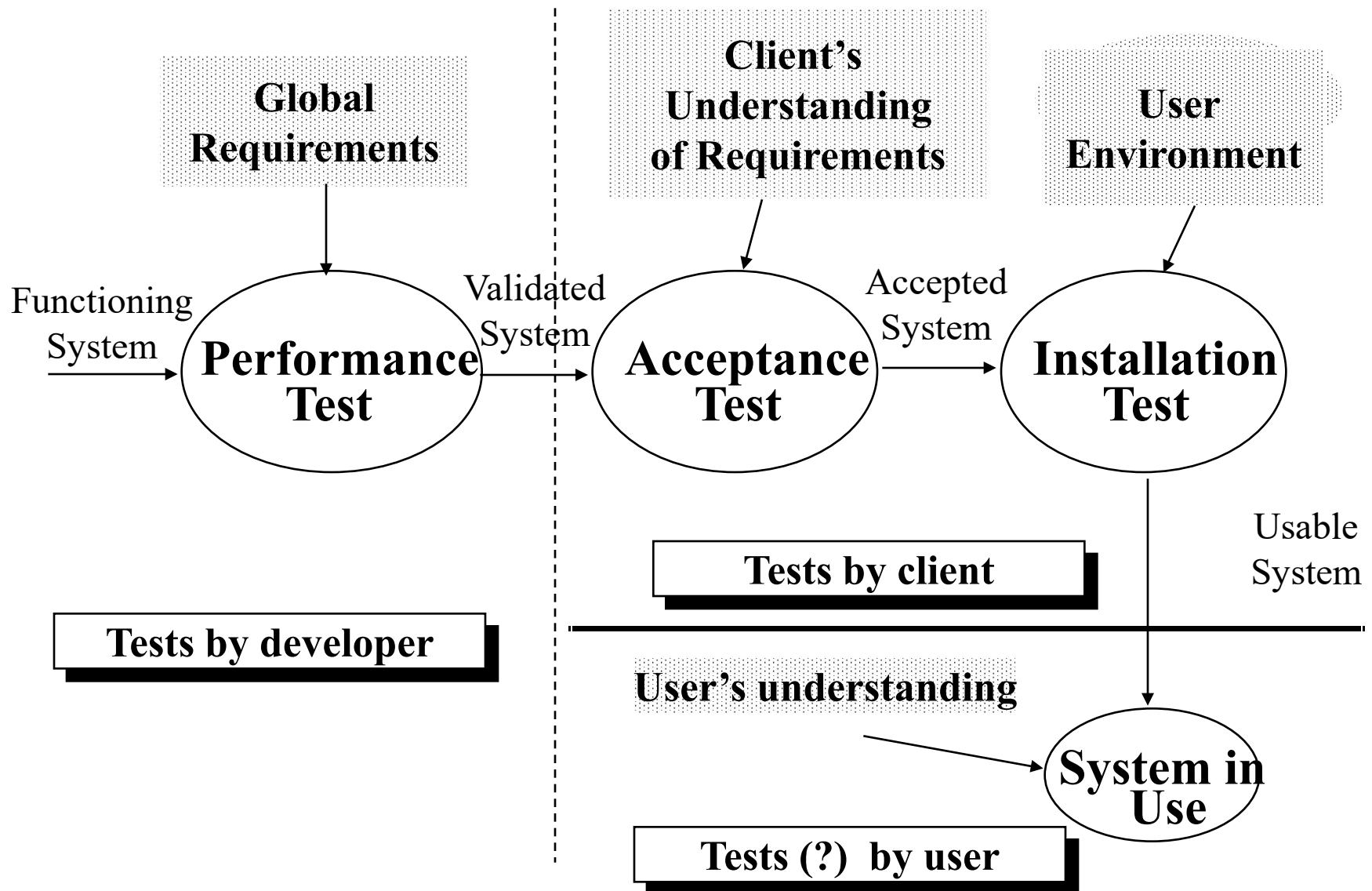
- ♦ Testing often viewed as dirty work.
- ♦ To develop an effective test, one must have:
  - ♦ Detailed understanding of the system
  - ♦ Knowledge of the testing techniques
  - ♦ Skill to apply these techniques in an effective and efficient manner
- ♦ Testing is done best by independent testers
  -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

# Testing Activities

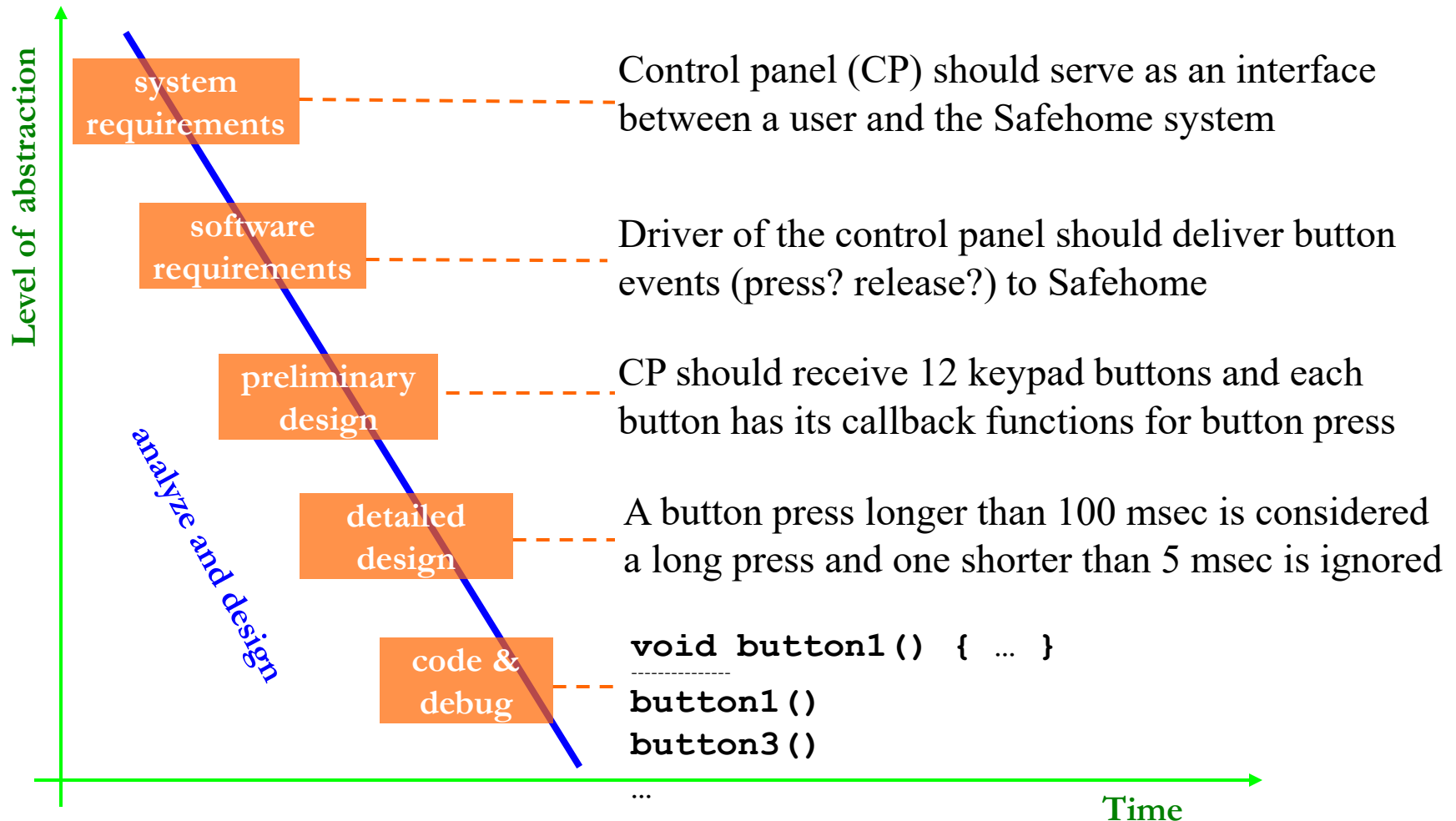


**Cf. levels of testing**

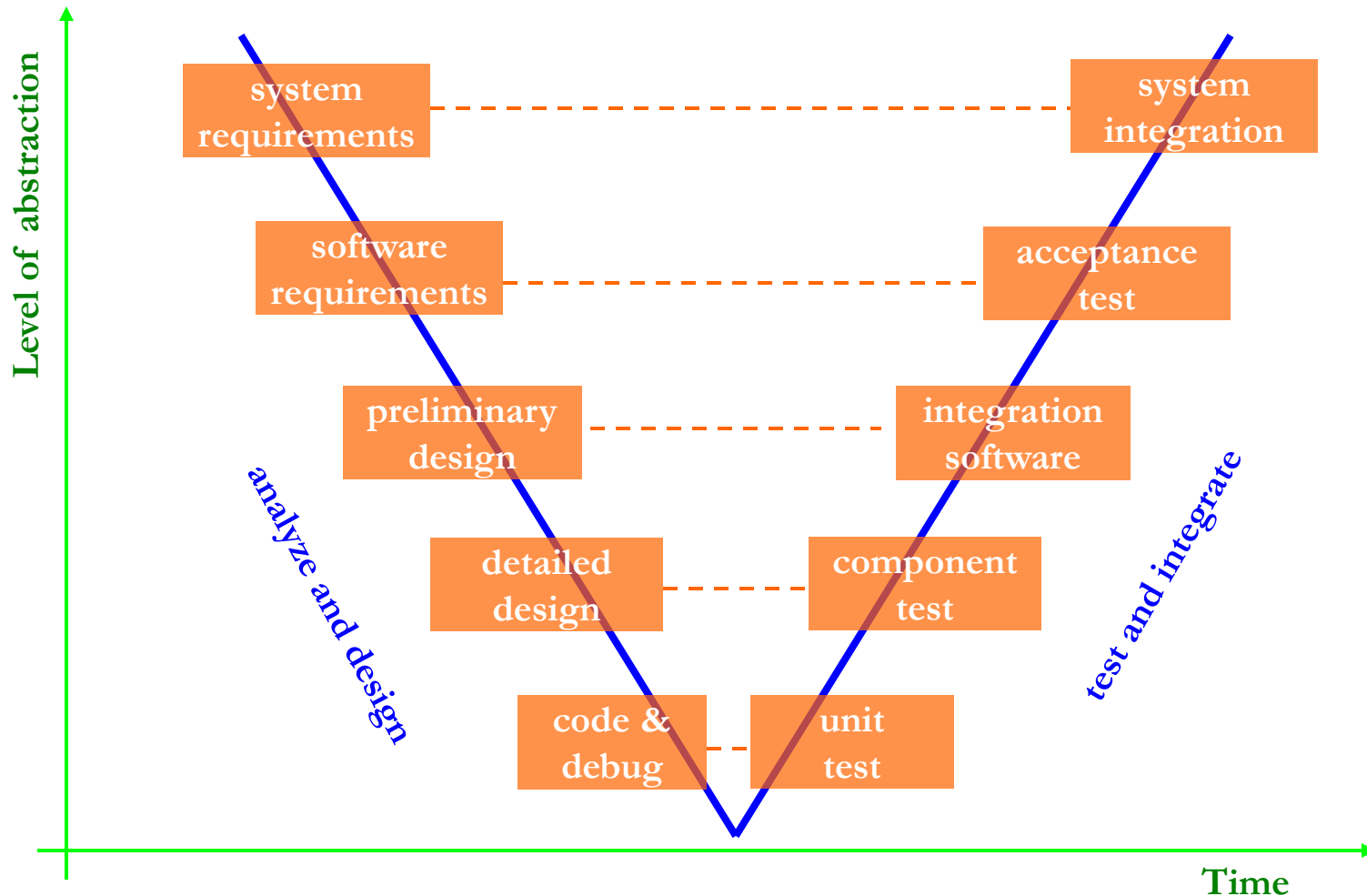
# Testing Activities continued



# *Safehome Example: Levels of Designs in V Model*



# Levels of Testing in V Model





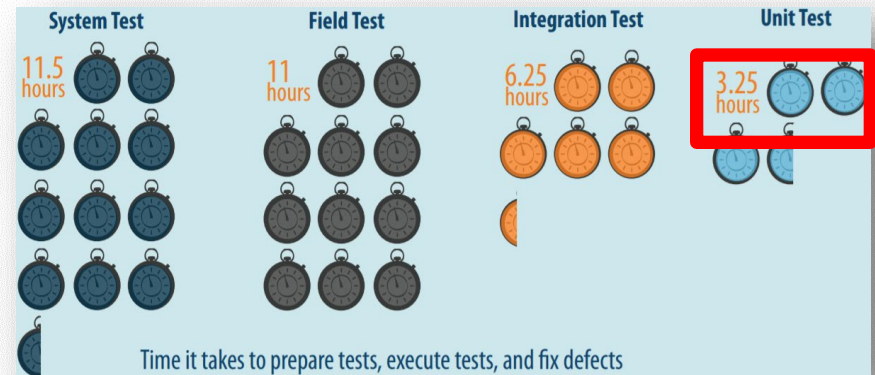
# Many Benefits of Unit Testing

- › **Bug correction cost: 7x cheaper** than system tests
  - › \$937 (unit test) vs \$7,136 (system test)

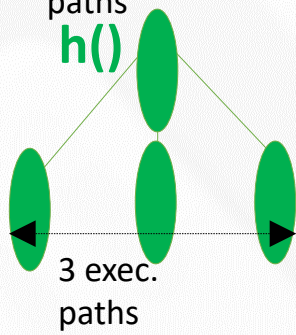
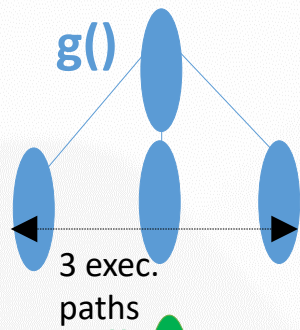
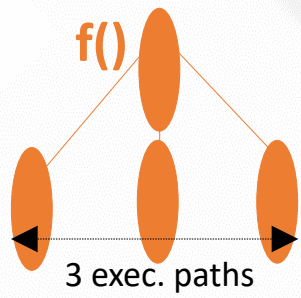


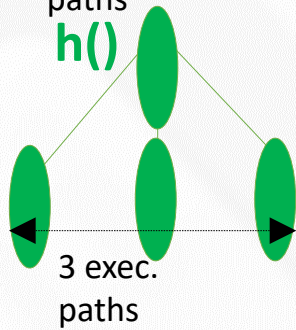
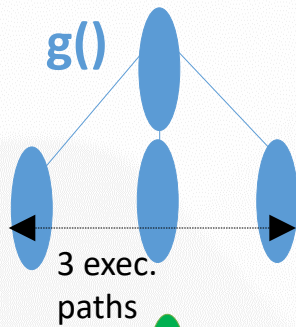
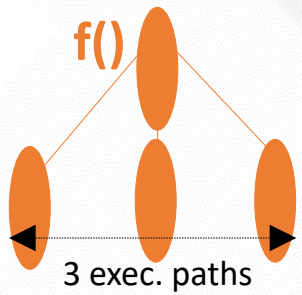
Source: B. Boehm and V. Basil, Software Defect Reduction Top 10 List, IEEE Computer, January 2001

- › **Bug correction time: 3x faster** than system testing
  - › 3.25 hours vs 11.5 hours

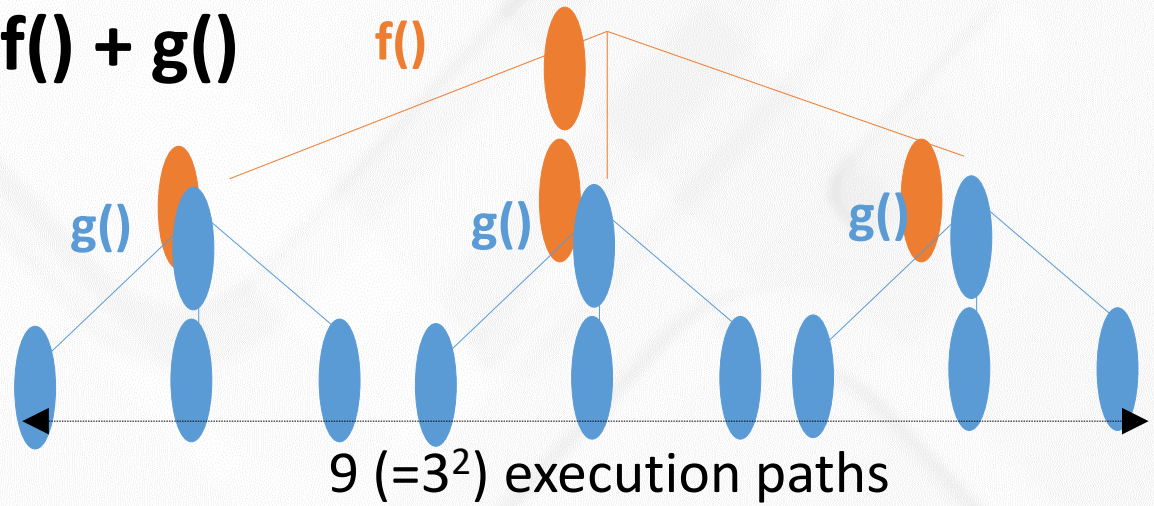


Source: Capers Jones, Applied Software Measurement: Global Analysis of Productivity and Quality

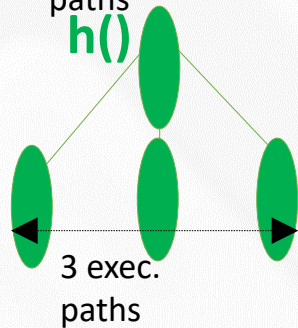
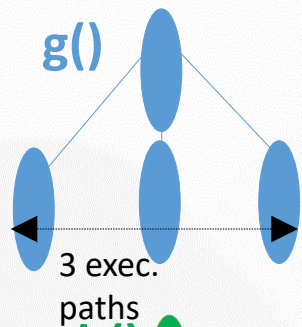
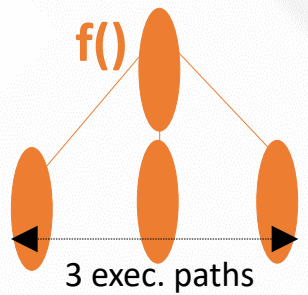




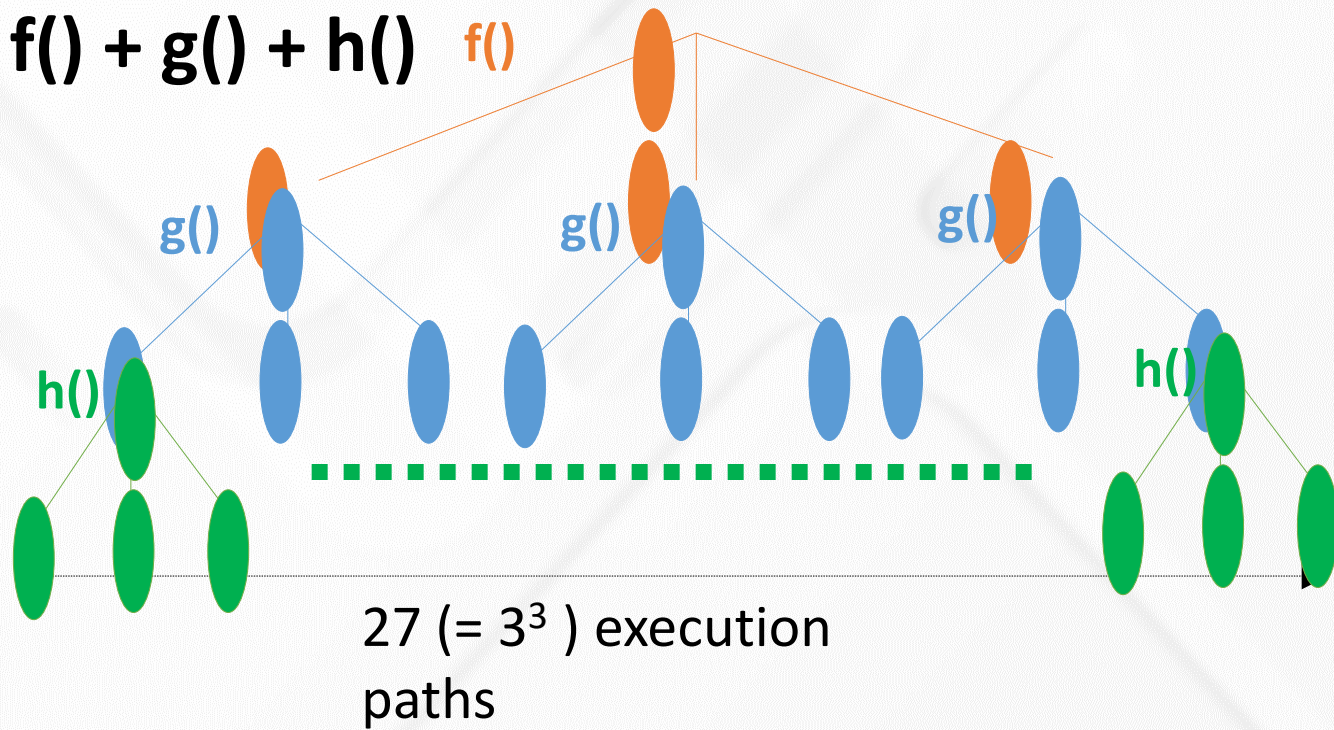
$f() + g()$







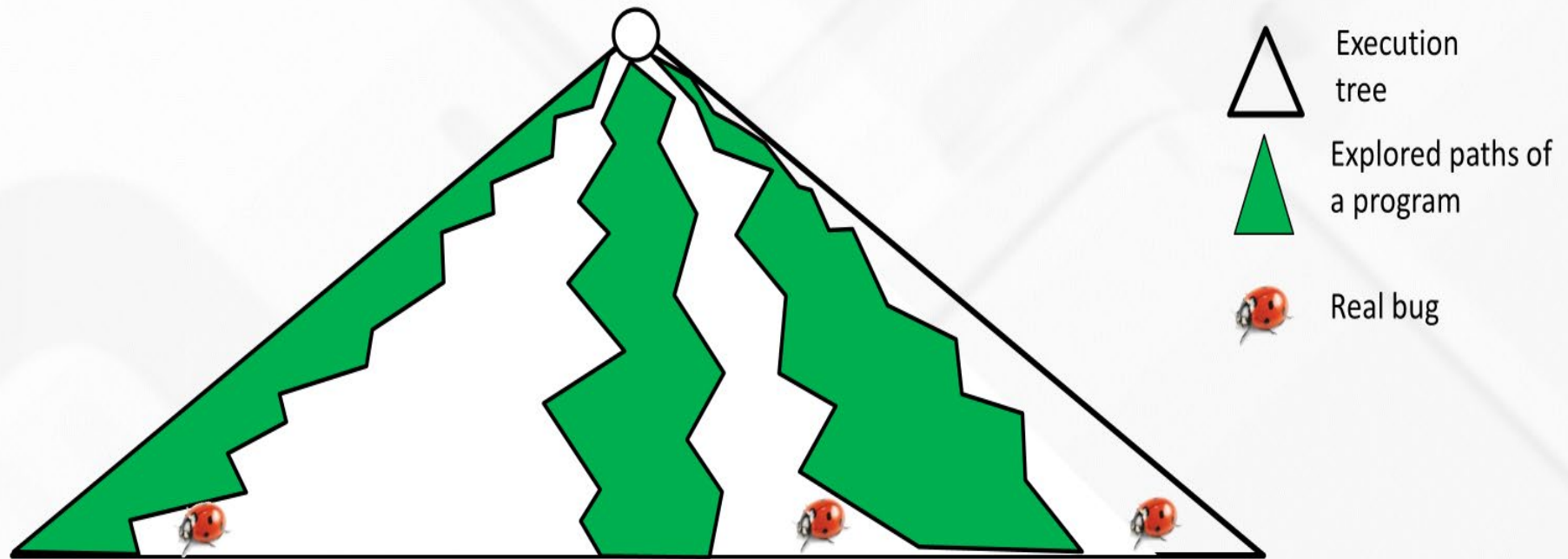
**f() + g() + h()** **f()**



# Pros and Cons of Auto. Test Gen. in System-level

› Pros: **No false alarms**

› Cons: Low bug detection power due to **large search space**



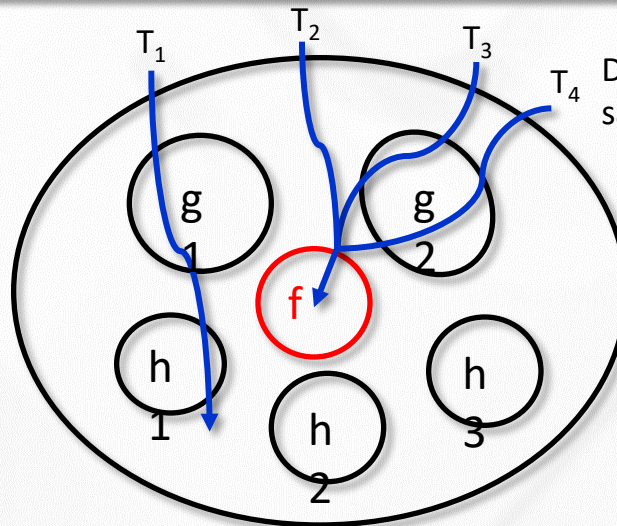
# Automated Test Generation in System-Level

## Pros

- + Can be easy to generate system TCs due to clear interface specification
- + No false alarm (i.e., no assert violation caused by infeasible execution scenario)

## Cons

- Low controllability of each unit
- Large and complex search space to explore in a limited time
- Hard to detect bugs in corner cases



Different system tests  $T_2$  to  $T_4$  exercise the same behavior of the target unit

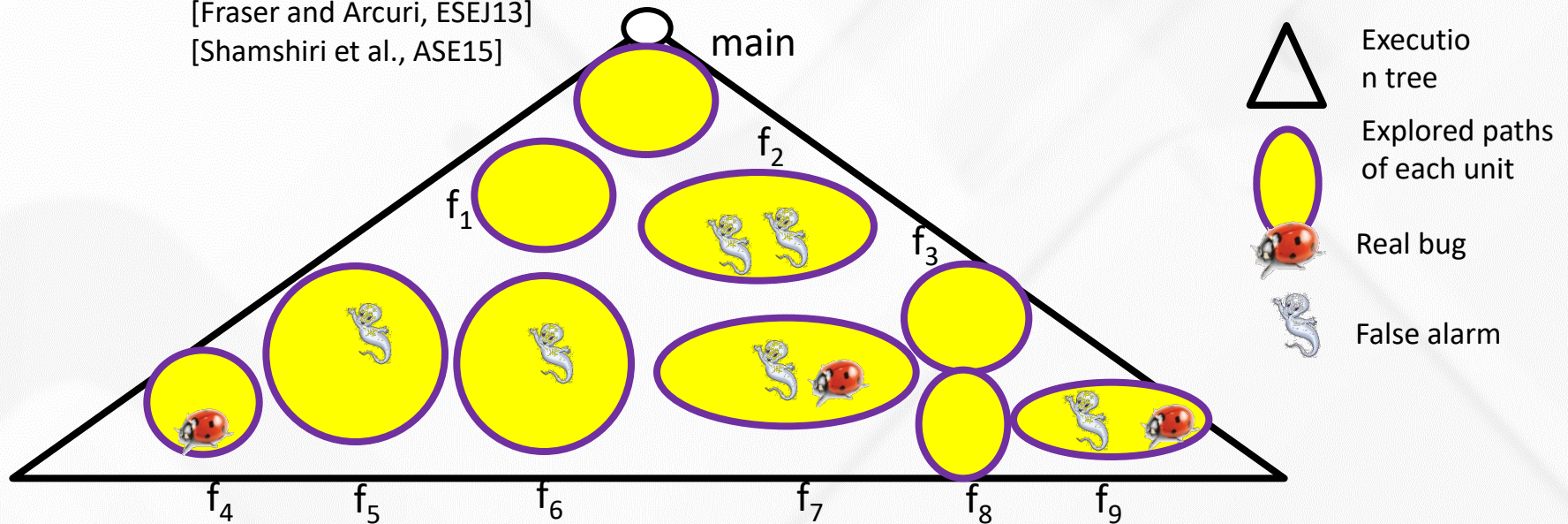
# Pros and Cons of Auto. Test Gen. in Unit-level

- › Pros: High bug detection power for **small search space**
- › Cons: **Many false alarms** due to over-approximated context of a unit

[Gross et al., ISSTA12]

[Fraser and Arcuri, ESEJ13]

[Shamshiri et al., ASE15]





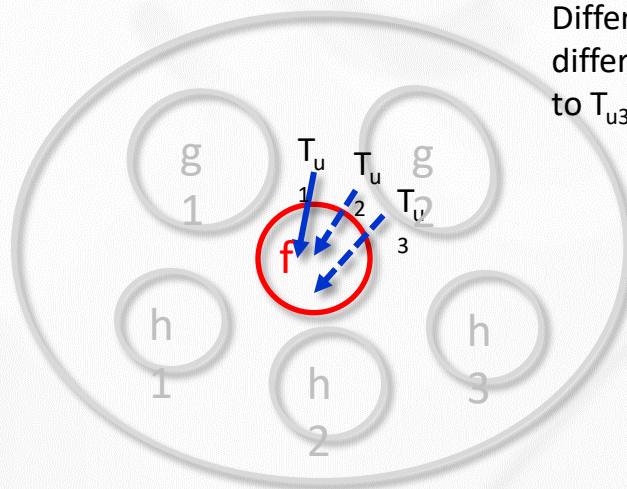
# Automated Test Generation in Unit-Level

## Pros

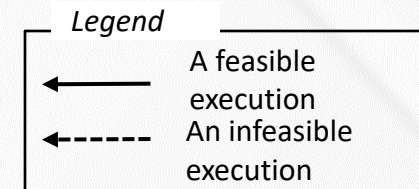
- + High controllability of a target unit
- + Smaller search space to explore than system testing
- + High effectiveness for detecting corner cases bugs

## Cons

- Hard to write down accurate unit test drivers/stubs due to unclear unit specification
- High false/true alarm ratio



Different unit tests  $T_{u1}$  to  $T_{u3}$  directly exercise different behaviors of the target unit, but  $T_{u2}$  to  $T_{u3}$  exercise infeasible paths



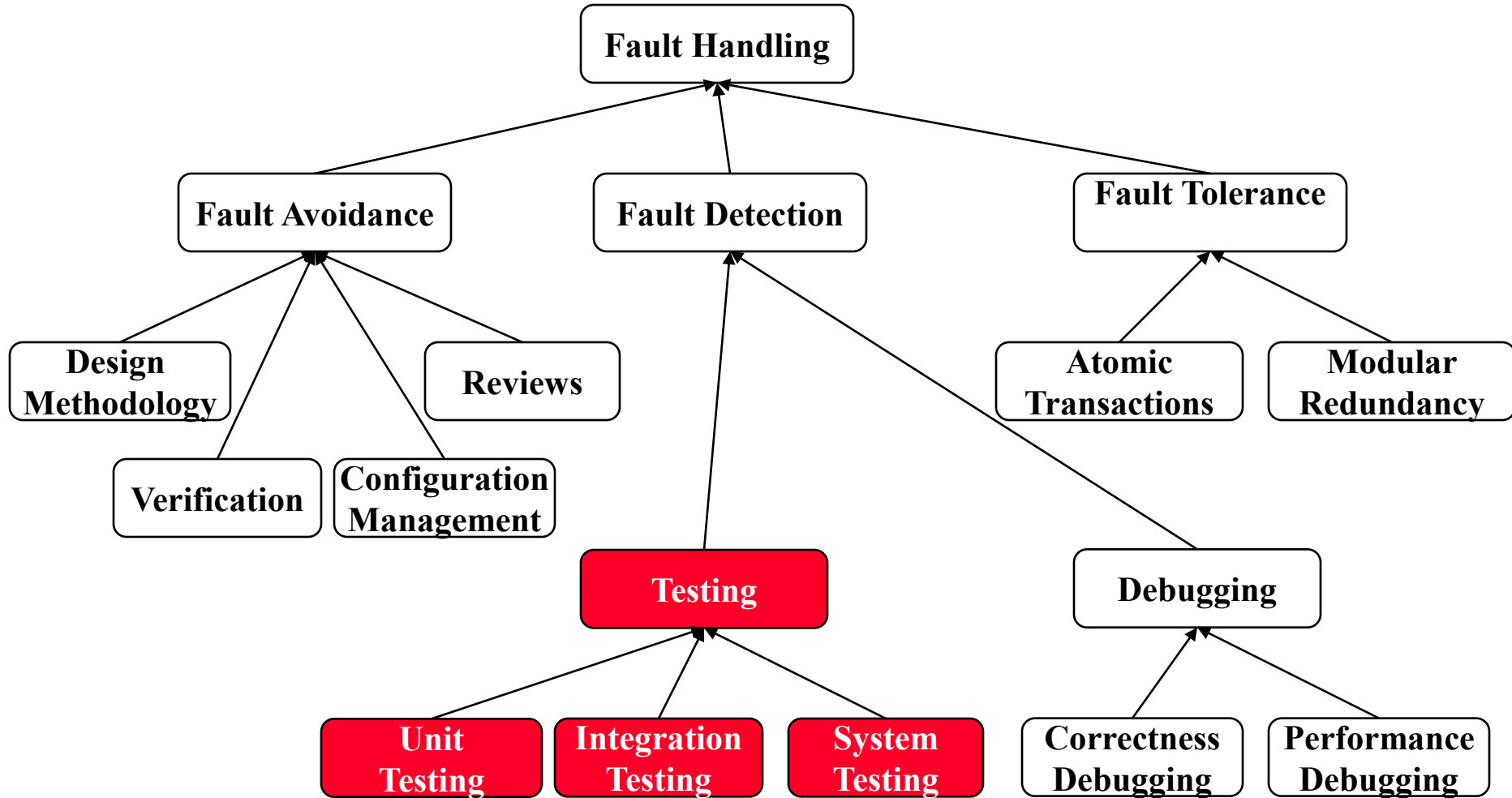


# *Test Planning*

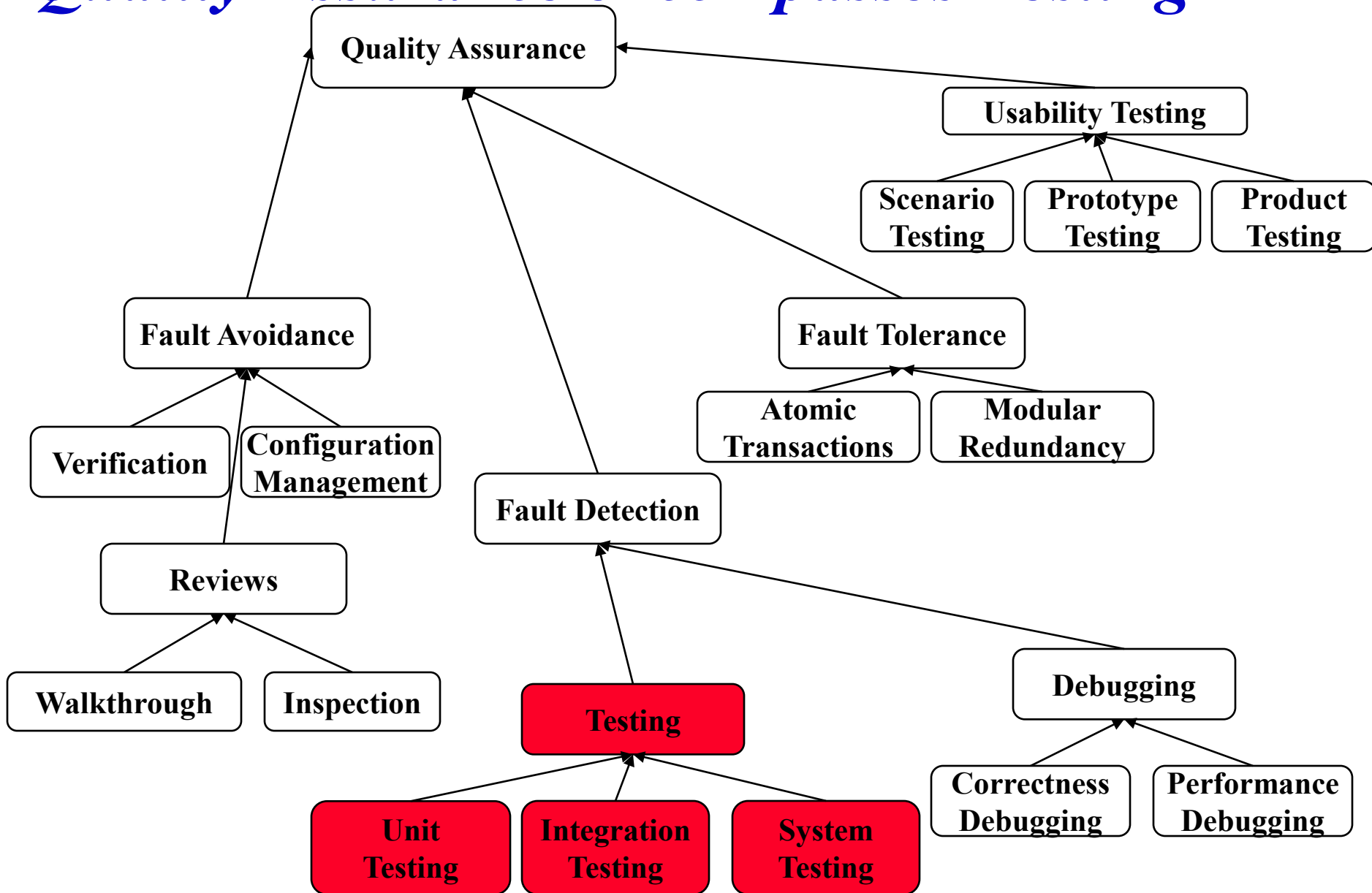
[Pressman]

- ◆ 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
    - ◆ I.e., test driven development (TDD)
- ◆ A test plan includes:
  - ◆ test objectives
  - ◆ schedule and logistics
  - ◆ test strategies
  - ◆ test cases
    - ◆ procedure
    - ◆ data
    - ◆ expected result
  - ◆ procedures for handling problems

# ***Fault Handling Techniques***



# *Quality Assurance encompasses Testing*



# *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 an 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, check points

# 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") **why?**
- ♦ 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**
  - ♦ **Branch Testing**
  - ♦ **Loop Testing**
  - ♦ **Path Testing**

## *White-box Testing (Continued)*

- ◆ Statement Testing: Test single statements
- ◆ Branch Testing (Conditional Testing): Make sure that each possible outcome from a condition is tested at least once

- ◆ 

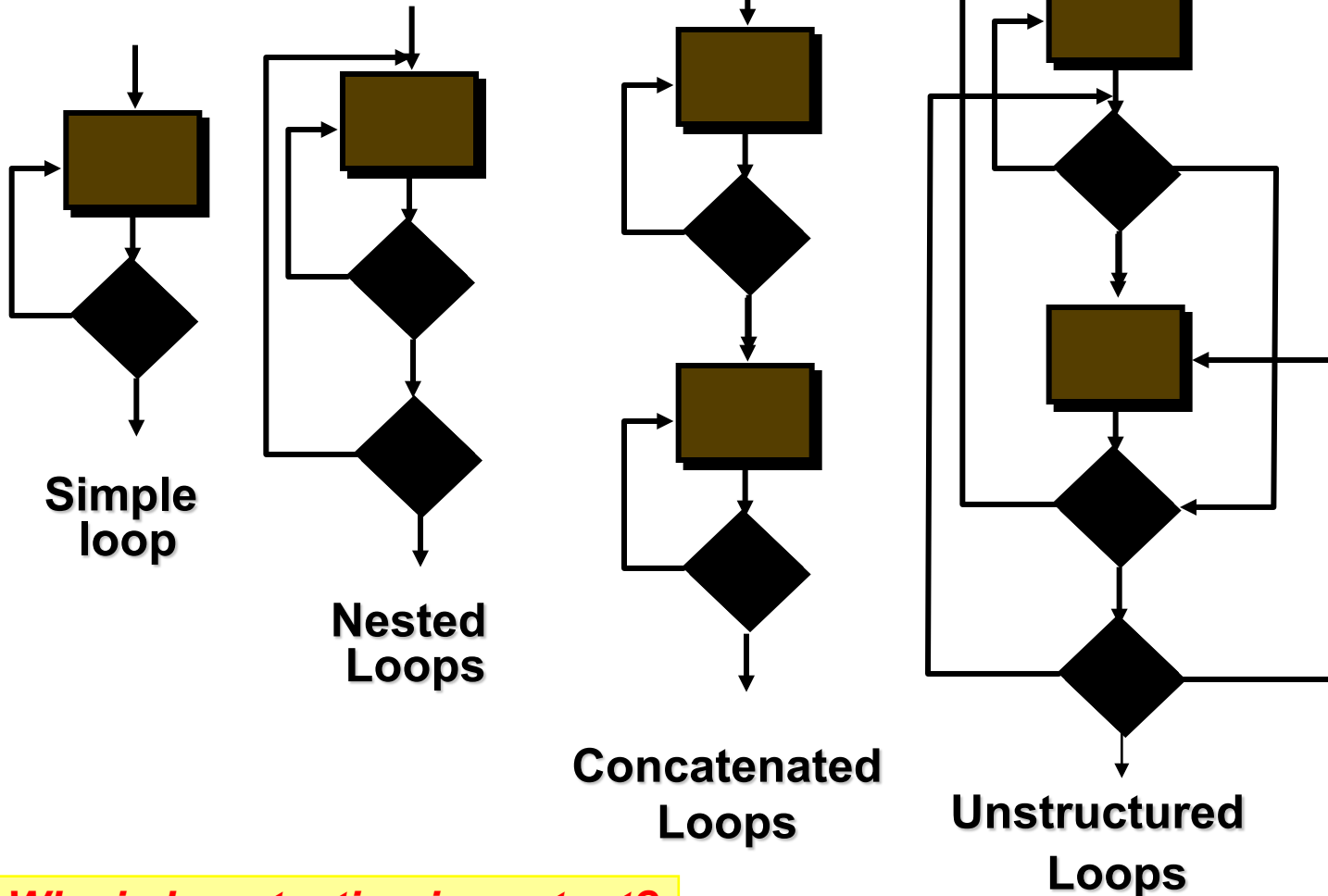
```
if ( i = TRUE) printf("YES\n"); else printf("NO\n");
```

**Test cases: 1) i = TRUE; 2) i = FALSE**

- ◆ Loop Testing:
  - ◆ 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

## *Loop Testing*

[Pressman]



***Why is loop testing important?***

# *Comparison of White & Black-box Testing 25.1.2002*

- ♦ 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 combinatorical 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**

# 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

# 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.**

*Big cost -> what should be done?*
8. Compare the results of the test with the test oracle
  - ♦ Automate as much as possible

# OOT Methods: Behavior Testing

[Pressman]

The tests to be designed should achieve **all state coverage** [KIR94]. That is, the operation sequences should cause the Account class to make transition through all allowable states

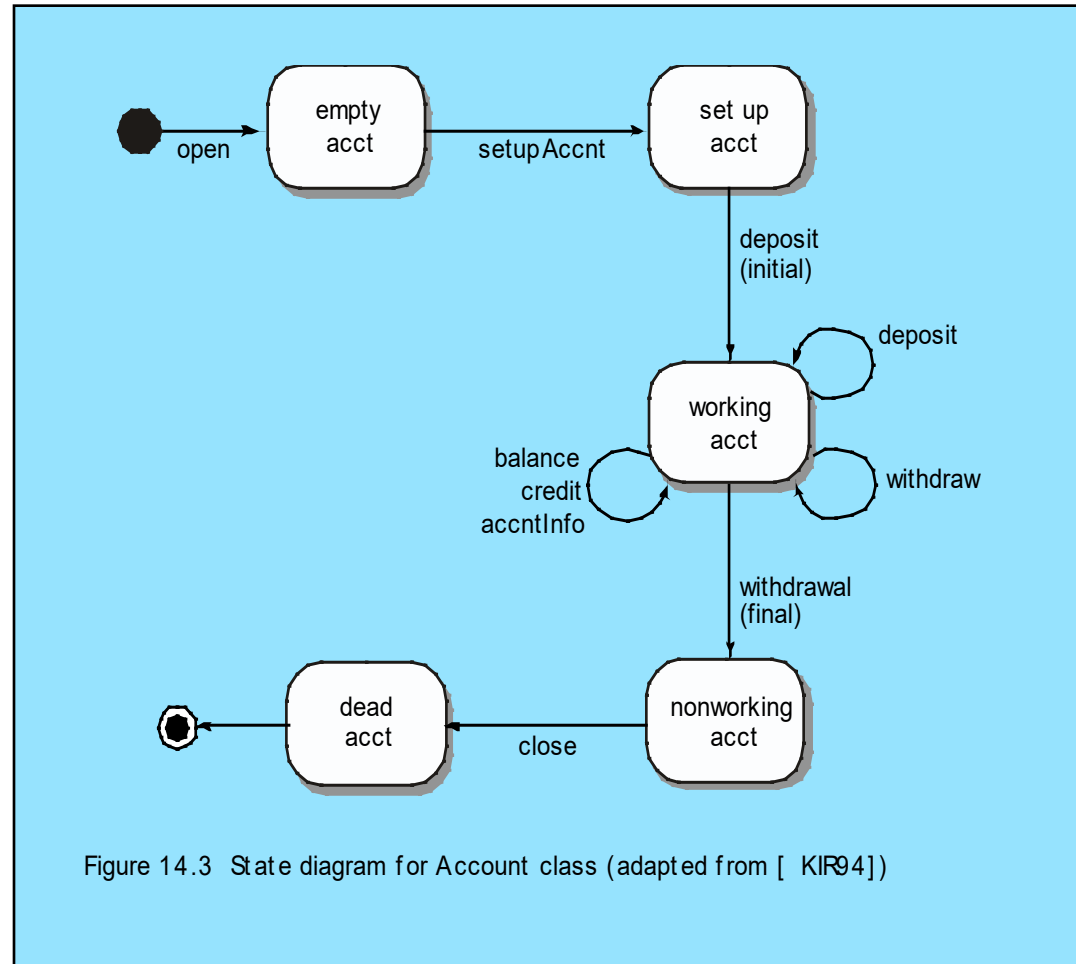


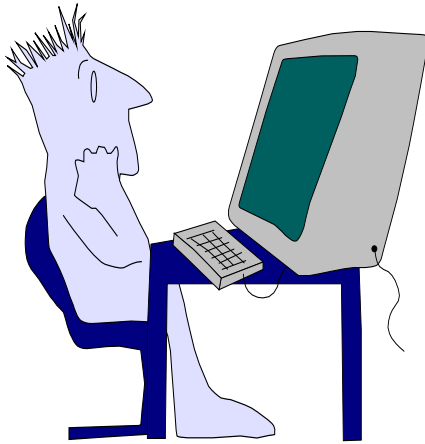
Figure 14.3 State diagram for Account class (adapted from [ KIR94])

***This can act as an oracle***



# Who Tests the Software?

[Pressman]



***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**

## Counting Bugs

- ♦ Sometimes reliability requirements take the form:  
*"The software shall have no more than X bugs/1K LOC"*  
But how do we measure bugs at delivery time?
- ♦ *Bebugging Process - based on a Monte Carlo technique for statistical analysis of random events.*
  1. before testing, a known number of bugs (seeded bugs) are secretly inserted.
  2. estimate the number of bugs in the system
  3. remove (both known and new) bugs.

$$\begin{aligned}\text{\# of detected seeded bugs} / \text{\# of seeded bugs} &= \text{\# of detected bugs} / \text{\# of bugs in the system} \\ \text{\# of bugs in the system} &= \text{\# of seeded bugs} \times \text{\# of detected bugs} / \text{\# of detected seeded bugs}\end{aligned}$$

**Example: secretly seed 10 bugs**  
*an independent test team detects 120 bugs (6 for the seeded)*  
 $\text{\# of bugs in the system} = 10 \times 120 / 6 = 200$   
 $\text{\# of bugs in the system after removal} = 200 - 120 - 4 = 76$

- ♦ But, deadly bugs vs. insignificant ones; not all bugs are equally detectable; ( Suggestion [Musa87]:  
*"No more than X bugs/1K LOC may be detected during testing"*  
*"No more than X bugs/1K LOC may be remain after delivery,*  
*as calculated by the Monte Carlo seeding technique"*

# 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, and ...
- ♦ OOT and architectural testing, still challenging
- ♦ User-oriented reliability modeling and evaluation not adequate
- ♦ Testing has its own lifecycle