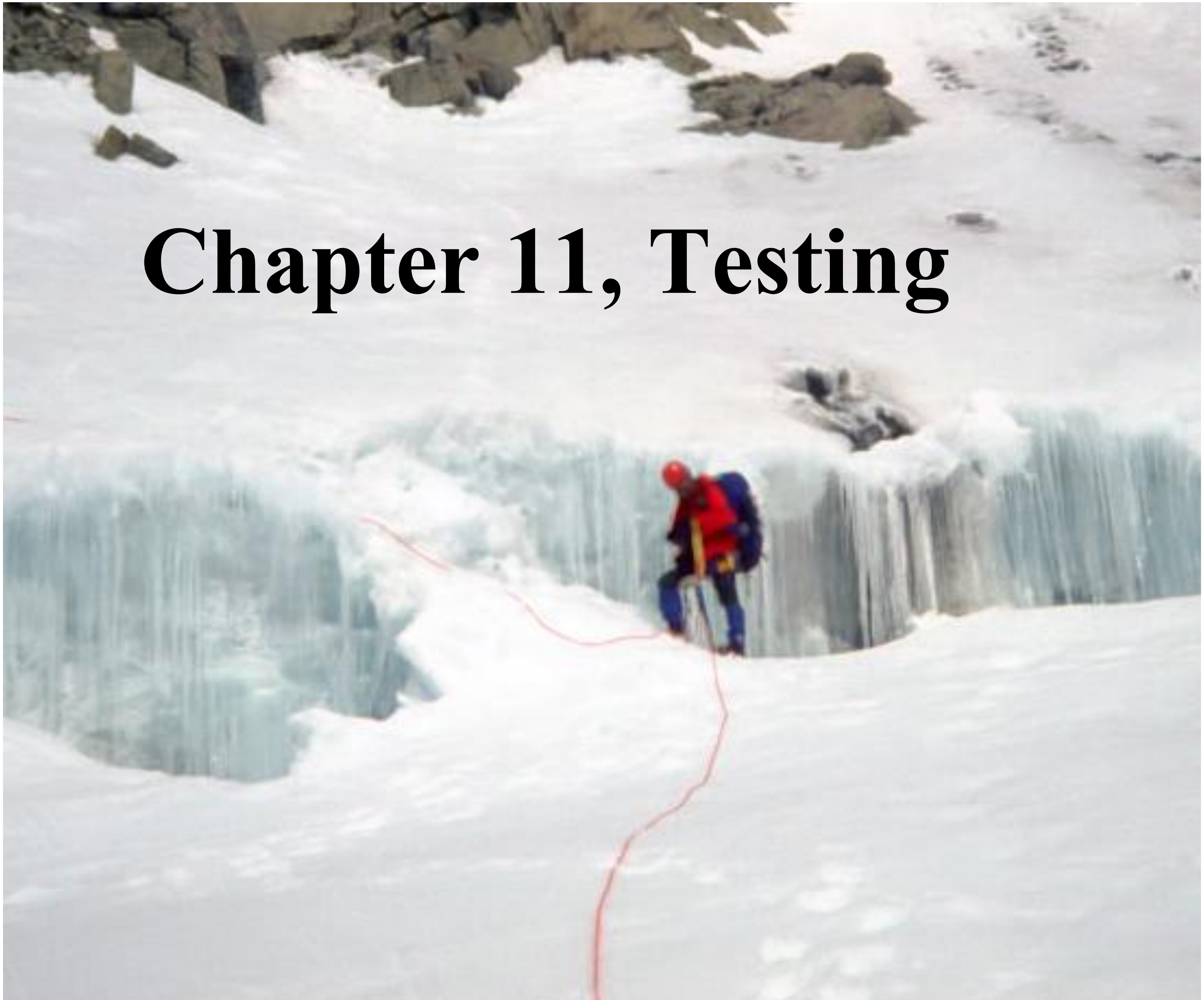
# Chapter 11, Testing



**Object-Oriented Software Engineering**
Using UML, Patterns, and Java

# *Outline*

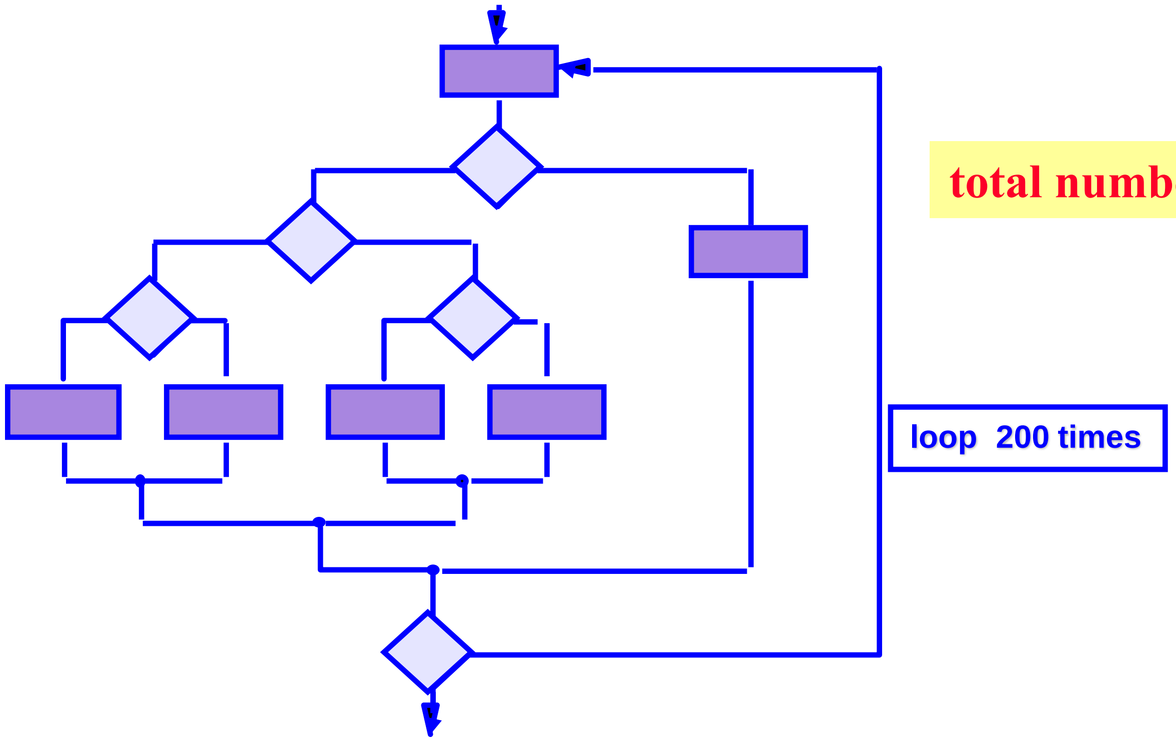
- Terminology
- Types of errors
- Dealing with errors
- Quality assurance vs Testing
- Component Testing
  - **Unit testing**
  - **Integration testing**
- Testing Strategy
- Design Patterns & Testing

- System testing
  - **Function testing**
  - **Structure Testing**
  - **Performance testing**
  - **Acceptance testing**
  - **Installation testing**

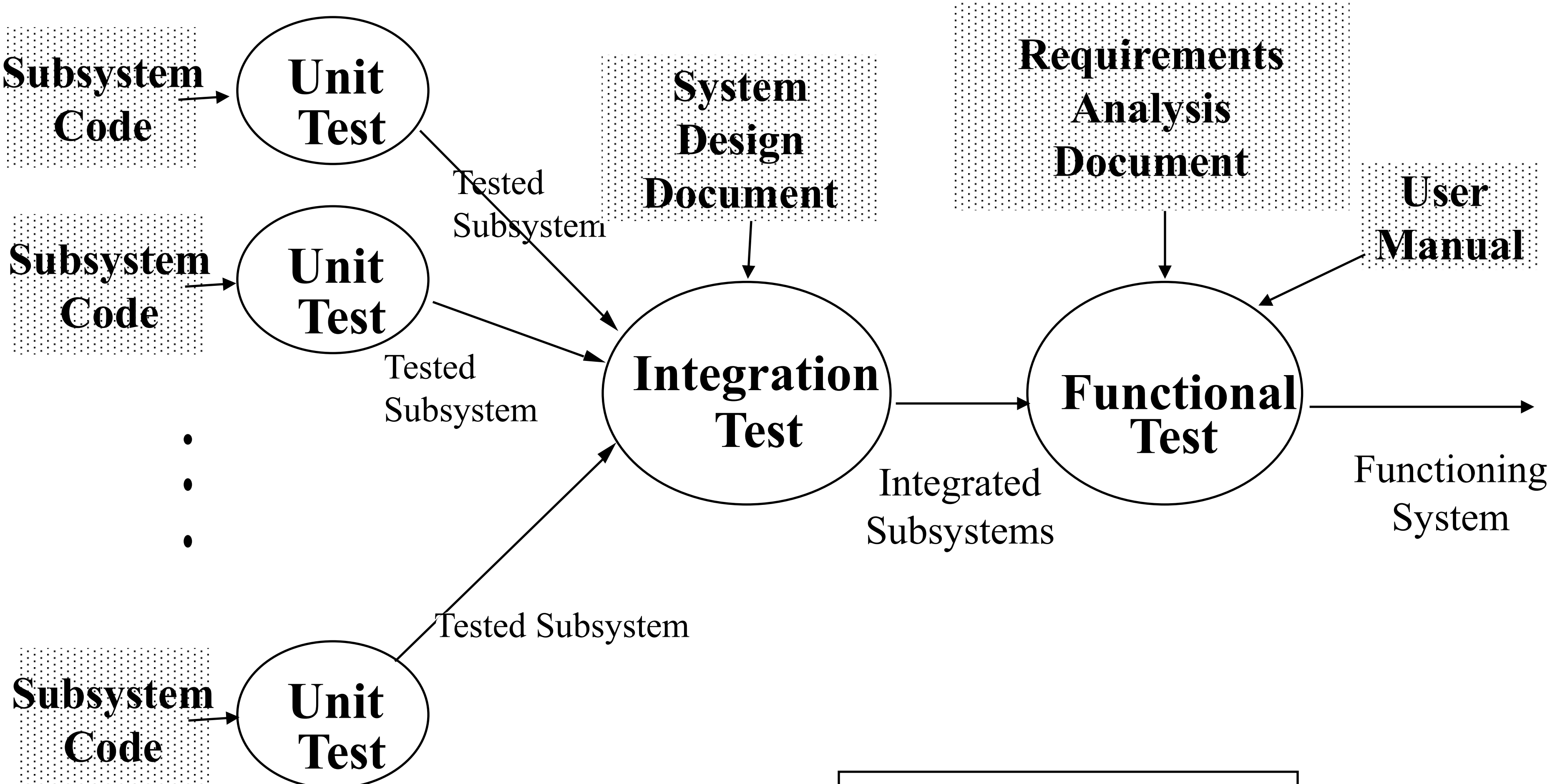
# *Some Observations*

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

**total number of execution paths?**



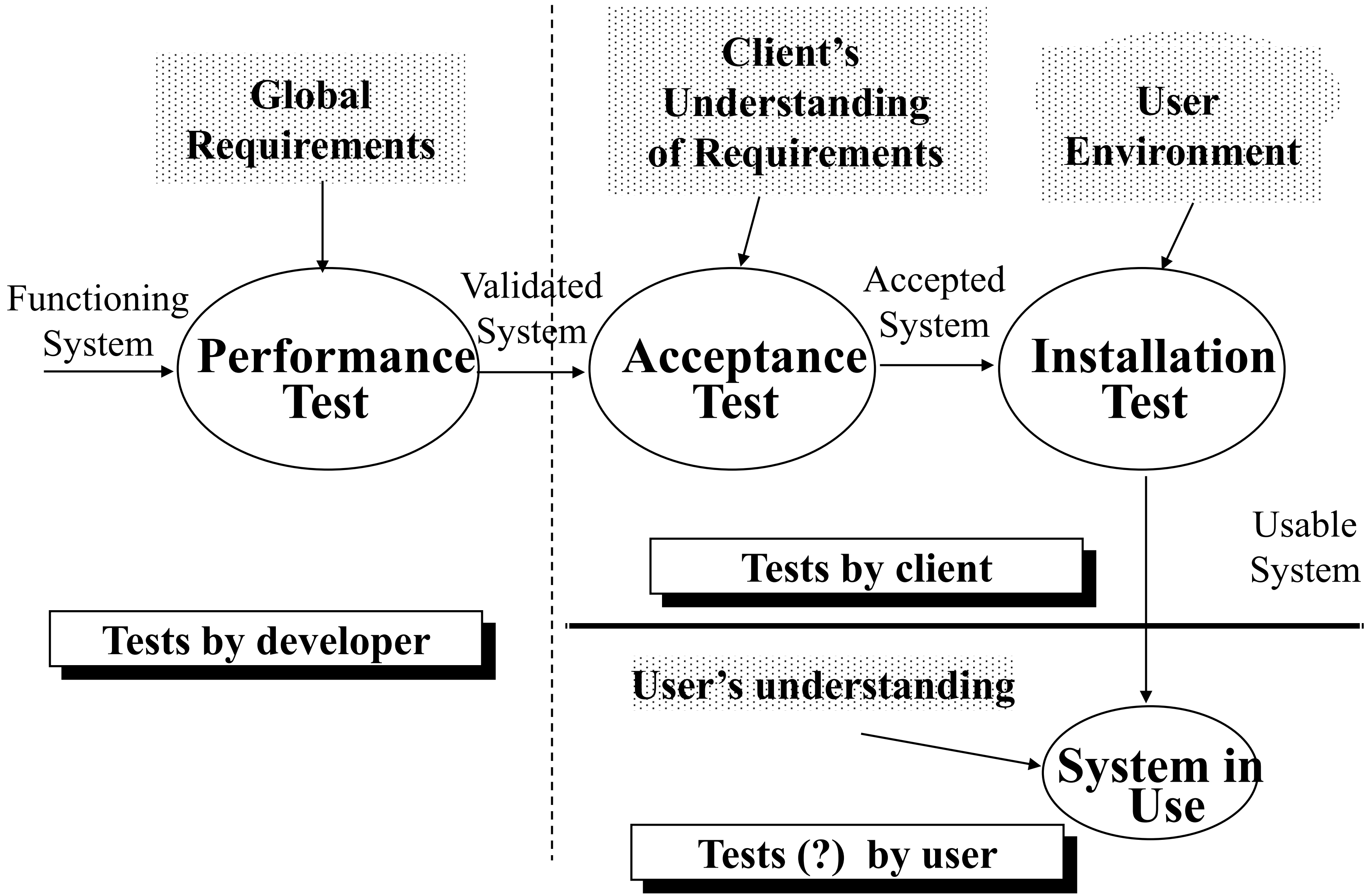
# *Testing Activities*



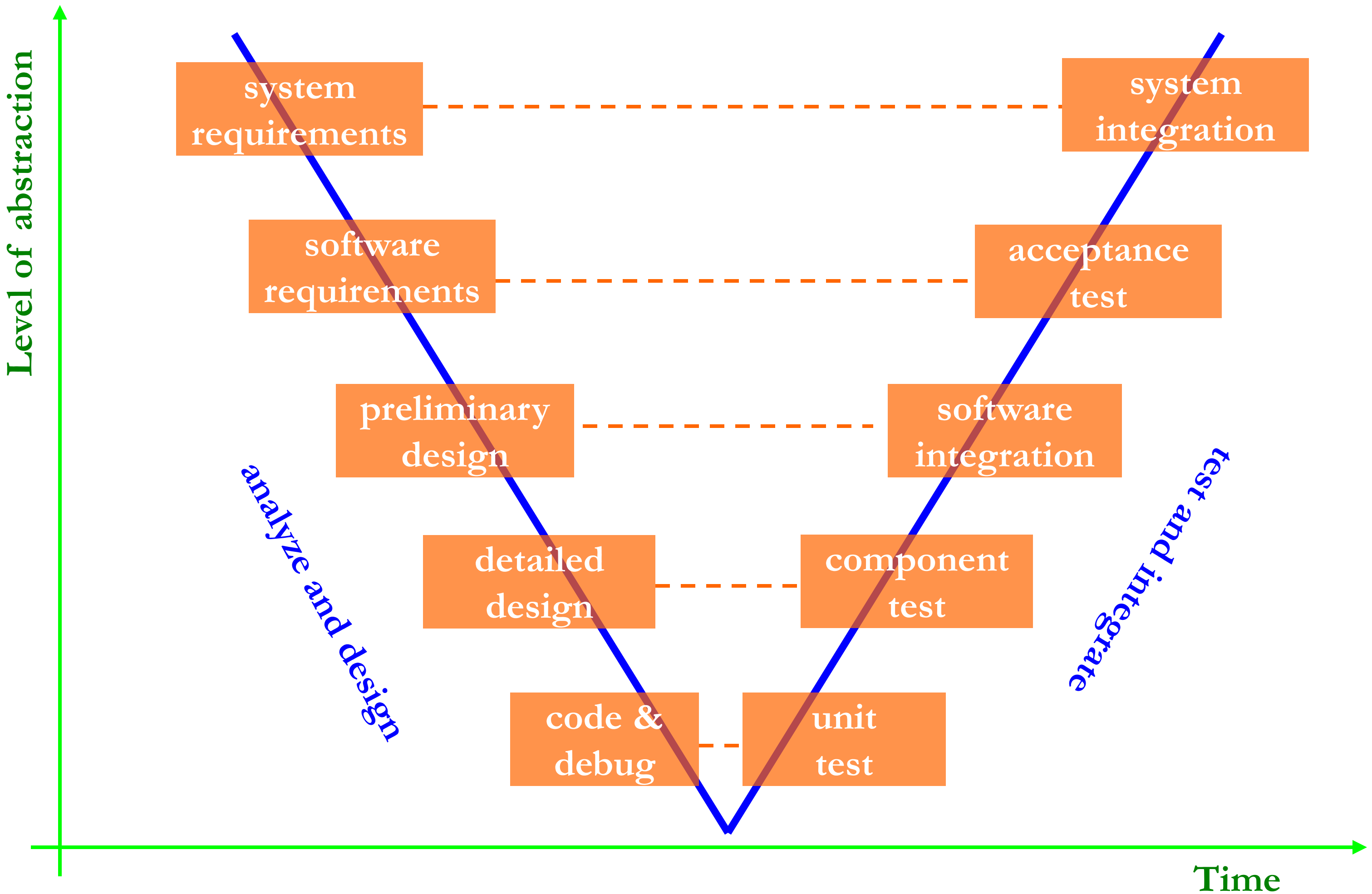

**All tests by developer**

**Cf. levels of testing**

# *Testing Activities continued*



# *Levels of Testing in V Model*



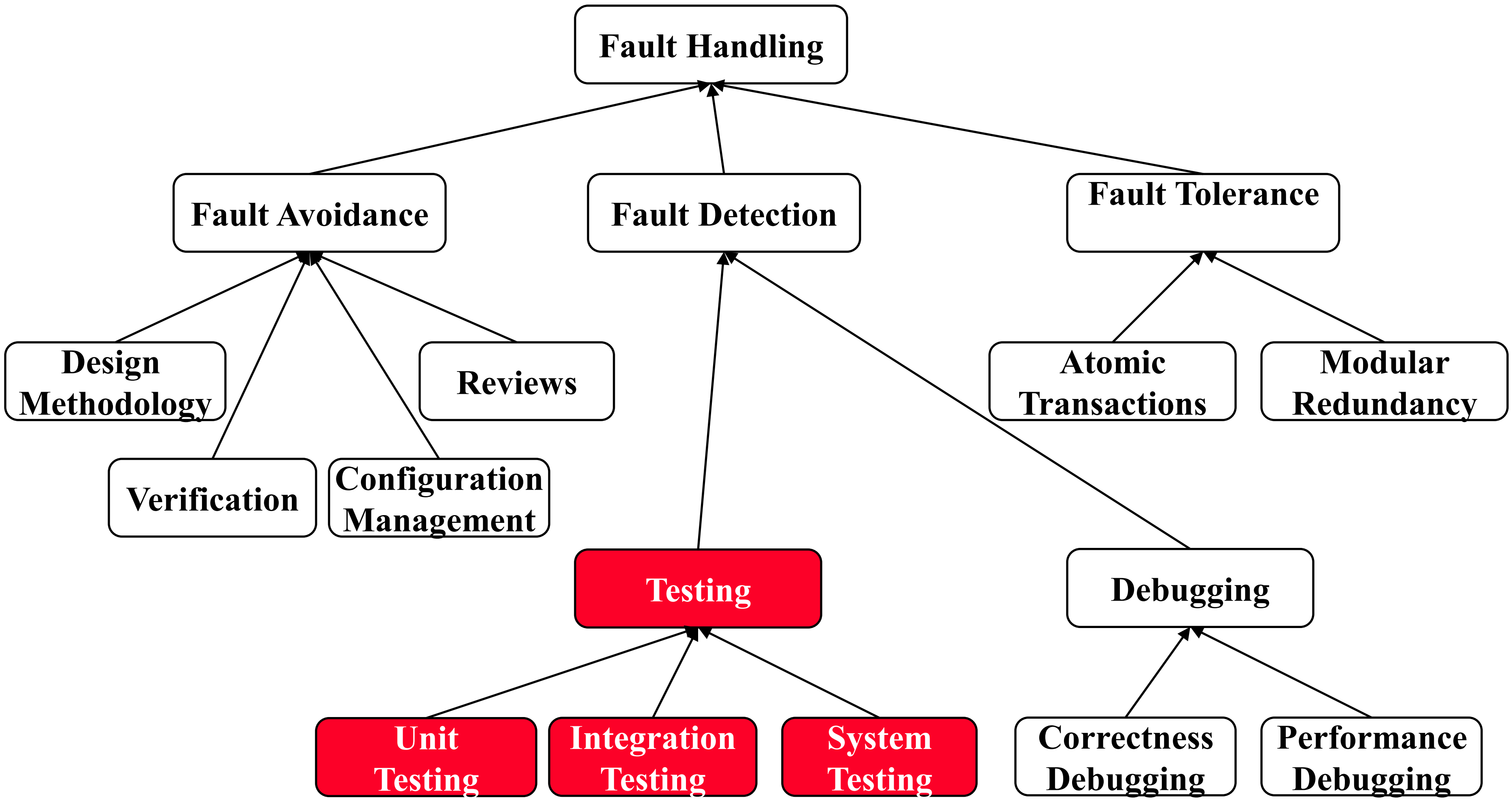

**N.B.: component test vs. unit test; acceptance test vs. system integration**

# *Test Planning*

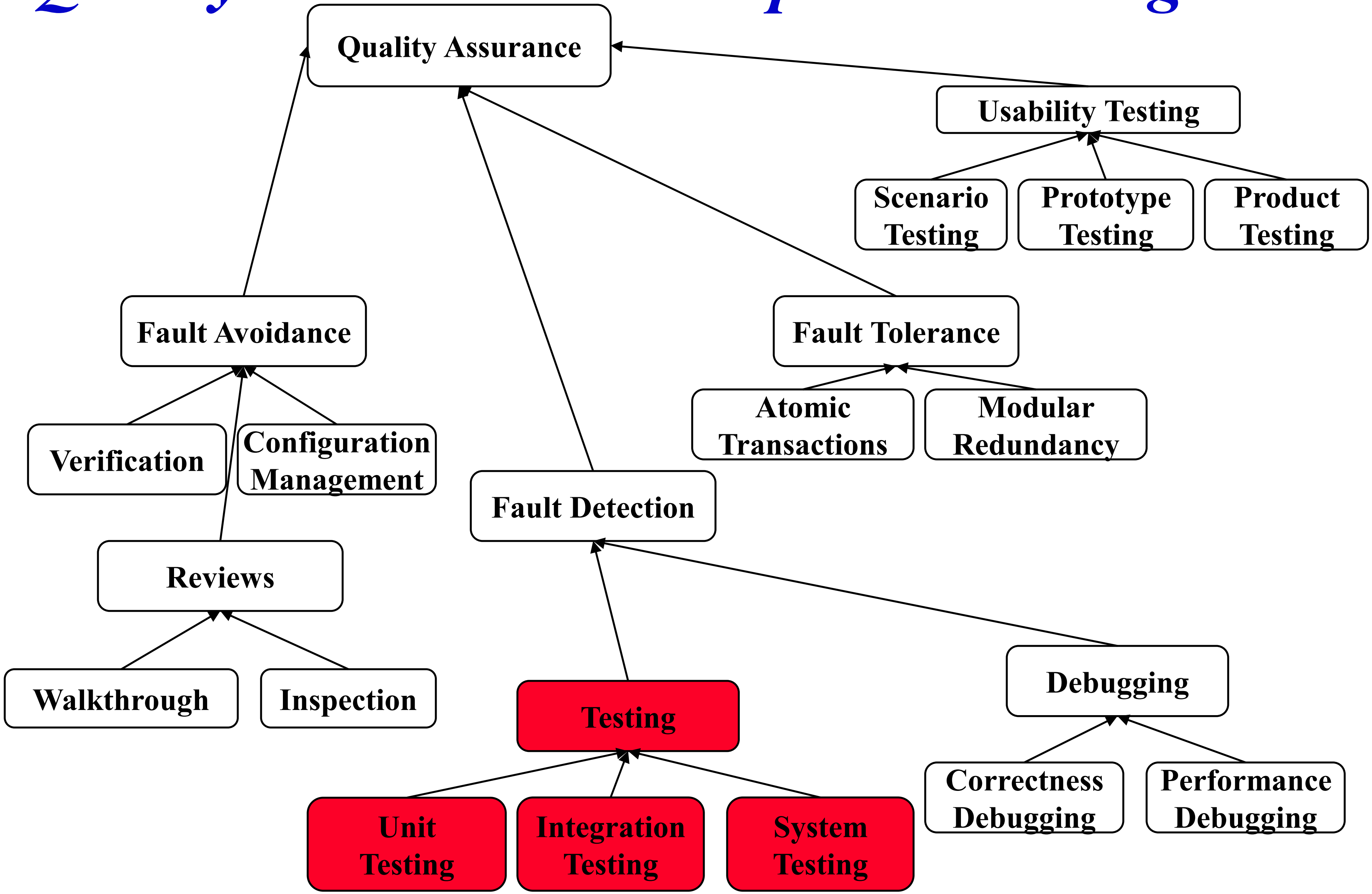
*[Pressman]*

- 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**
- 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



# *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*

Terminology:
system testing here = validation testing

- **System** Testing:
  - **The entire system**
  - **Carried out by developers**
  - **Goal: Determine if the system meets the *requirements* (functional and *global*)**

2 kinds of Acceptance testing

- **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 *requirement*s 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**

*Are these complete?*

  - **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**

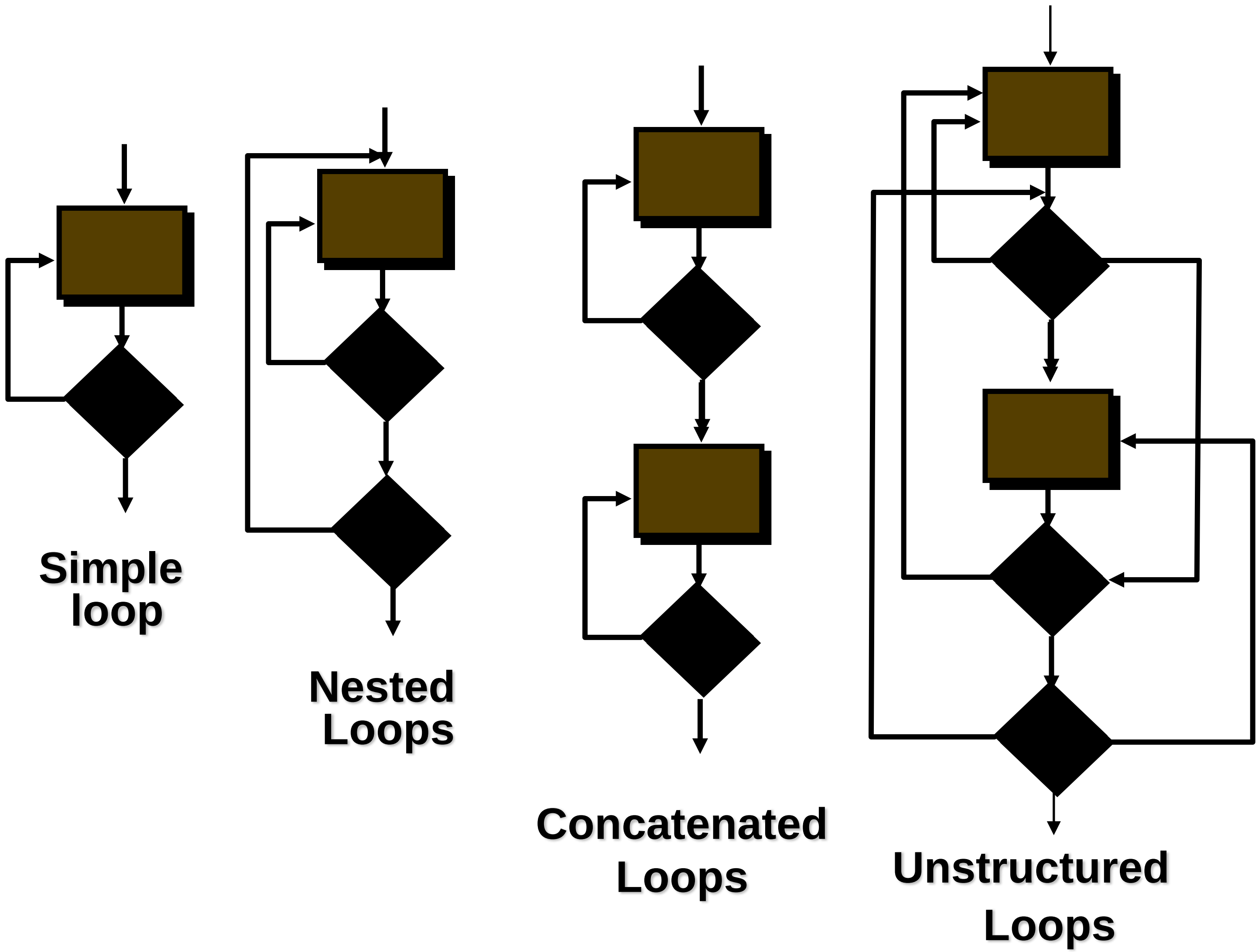
# *White-box* Testing (Continued)

- Statement Testing (Algebraic Testing): Test single statements
- 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

*[Pressman]*




*Why is loop testing important?*

# White-box Testing Example

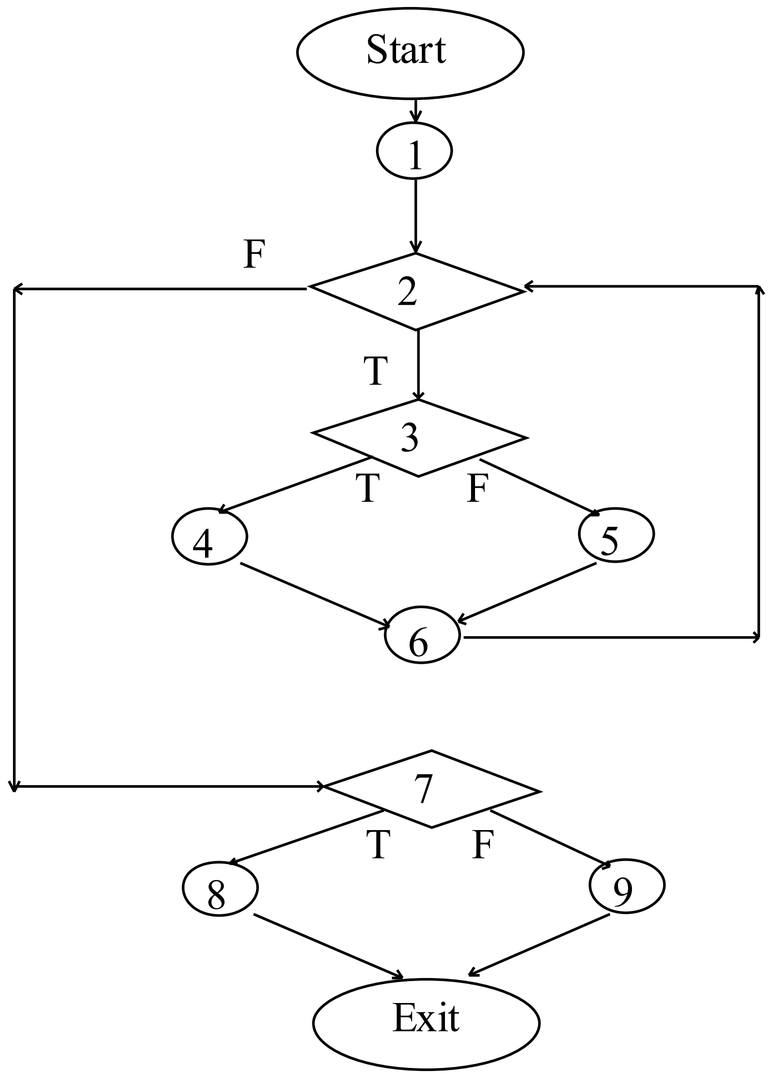
```
FindMean(float Mean, FILE ScoreFile)
{ SumOfScores = 0.0; NumberOfScores = 0; Mean = 0;
 Read(ScoreFile, Score);  /*Read in and sum the scores*/
 while (! EOF(ScoreFile) {
        if ( Score > 0.0 ) {
              SumOfScores = SumOfScores + Score;
             NumberOfScores++;
         }
         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 Example: Determining the Paths*

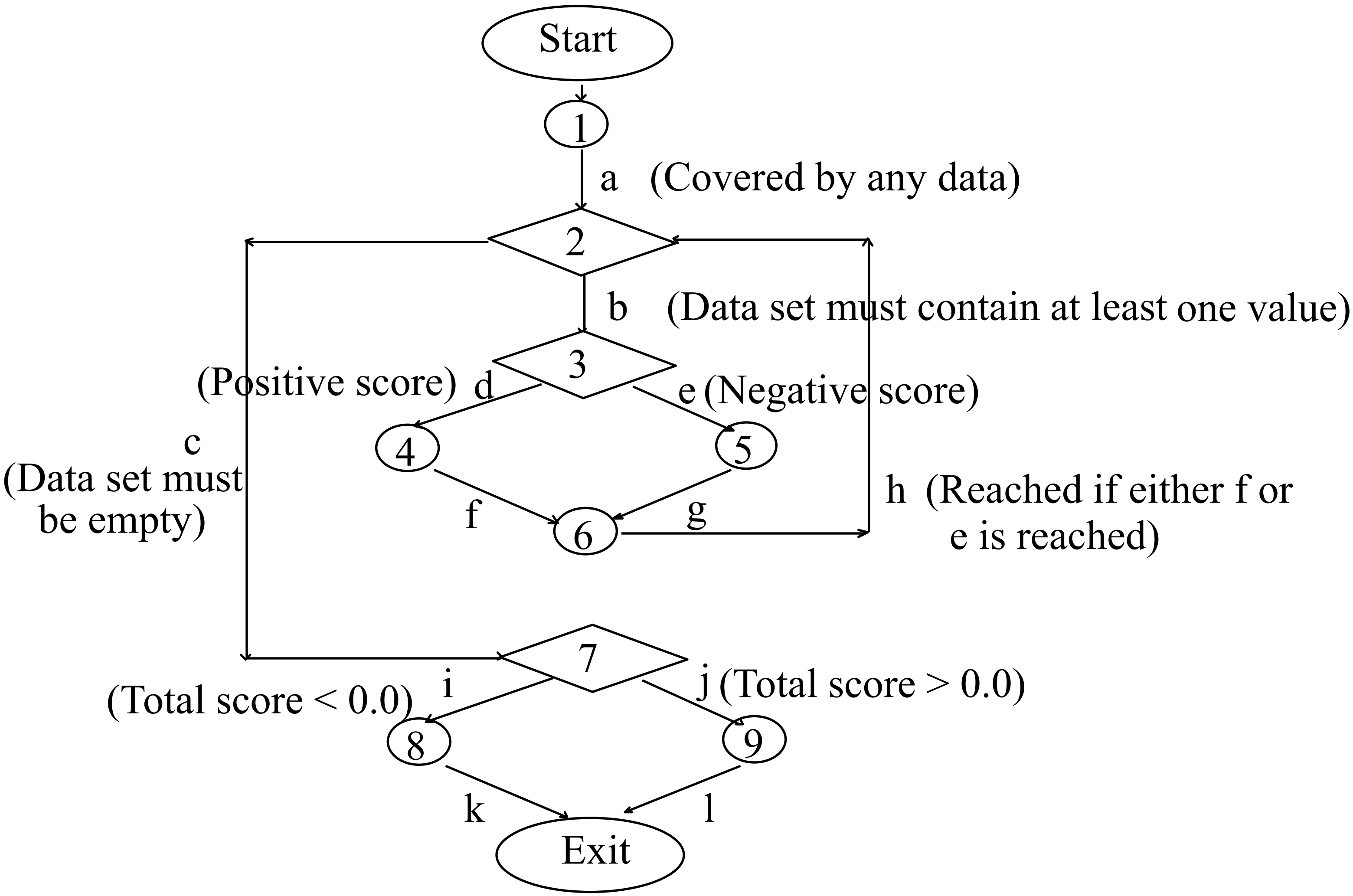
```
FindMean (FILE ScoreFile)
{   float SumOfScores = 0.0;
    int NumberOfScores = 0;                                    1
    float Mean=0.0; float Score;
    Read(ScoreFile, Score);
2   while (! EOF(ScoreFile) {
    3  if (Score  > 0.0 ) {
            SumOfScores = SumOfScores + Score;                 4
            NumberOfScores++;
        5   }

        Read(ScoreFile, Score);                                6
    }
    /* Compute the mean and print the result */
7   if (NumberOfScores > 0) {
          Mean = SumOfScores / NumberOfScores;                 8
          printf(" The mean score is %f\n", Mean);
    } else
          printf ("No scores found in file\n");                9
}
```

# Constructing the Logic Flow Diagram



# *Finding the Test Cases*



Self reading

# *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**

# *The 4 Testing Steps*

1. Select <u>what</u> has to be measured
   - **Analysis: Completeness of requirements**
   - **Design: tested for cohesion**
   - **Implementation: Code tests**
2. Decide <u>how</u> the testing is done
   - **Code inspection**
   - **Proofs (Design by Contract)**
   - **Black-box, white box,**
   - **Select integration testing strategy (big bang, bottom up, top down, sandwich)**

**Next module**

3. Develop <u>test cases</u>
   - **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 ***<u>test oracle</u>***
   - **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**

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.**

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

# *OOT Strategy*

*[Pressman]*

- class testing is the equivalent of unit testing
  - *operations* **within the class are tested**

*…if there is no nesting of classes*

  - **the** *state behavior* **of the class is examined**

- 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*

*…this is pushing…*

  - **cluster testing—integrates the set of classes required to demonstrate** *one collaboration*

***Recall: model-driven software development***

# *OOT—Test Case Design*

*[Pressman]*

**Berard [BER93] proposes the following approach:**

1. **Each test case should be uniquely identified and should be explicitly associated with the *class* to be tested,**
2. **A list of testing steps should be developed for each test and should contain [BER94]:**
   a. **a list of specified *states* for the object that is to be tested**
   b. **a list of *messages and operations* that will be exercised as a consequence of the test** ***how can this be done?***
   c. **a list of *exceptions* that may occur as the object is tested**
   d. **a list of *external conditions* (i.e., changes in the environment external to the software that must exist in order to properly conduct the test)**

*{people, machine, time of operation, etc.}*

***This is a kind of data structure testing***

# *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**

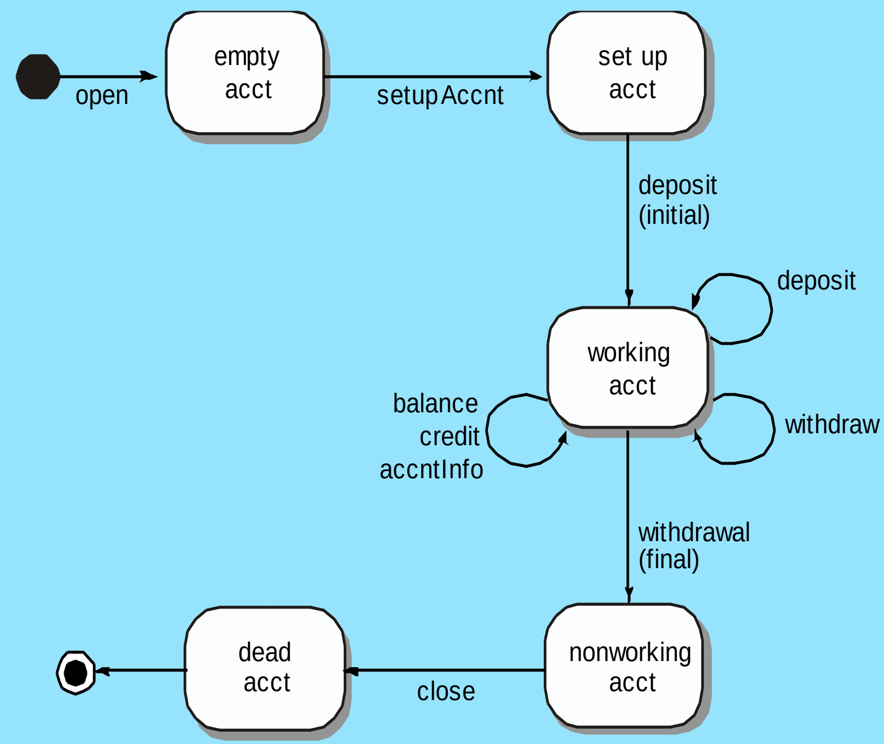
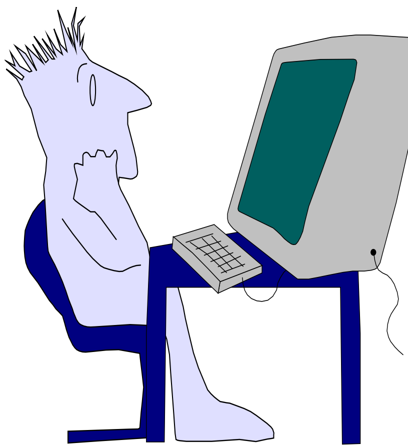



Figure 14.3 State diagram for Account class (adapted from [ KIR94])

***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***

# *NFRs: Reliability*

[Chung, RE Lecture Notes]]

## 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.**

  ***# of detected seeded bugs/ # of seeded bugs = # of detected bugs/ # of bugs in the system***
  ***# of bugs in the system = # of seeded bugs x # of detected bugs /# of detected seeded bugs***

  **Example:** ***secretely seed 10 bugs***
  ***an independent test team detects 120 bugs (6 for the seeded)***
  ***# of bugs in the system = 10 x 120/6 = 200***
  ***# of bugs in the system after removal = 200 - 120 - 4 = 76***

- **But, deadly bugs vs. insignifant 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

# *Additional Slides*

# *Terminology*

- **Reliability:** The measure of success with which the observed behavior of a system confirms to some specification of its behavior.
- **Failure:** Any deviation of the observed behavior from the specified behavior.
- **Error:** The system is in a state such that further processing by the system will lead to a failure.
- **Fault (Bug):** The mechanical or algorithmic cause of an error.

There are many different types of errors and different ways how we can deal with them.

# *Examples of Faults and Errors*

- Faults in the Interface specification
  - **Mismatch between what the client needs and what the server offers**
  - **Mismatch between requirements and implementation**
- Algorithmic Faults
  - **Missing initialization**
  - **Branching errors (too soon, too late)**
  - **Missing test for nil**

- Mechanical Faults (very hard to find)
  - **Documentation does not match actual conditions or operating procedures**
- Errors
  - **Stress or overload errors**
  - **Capacity or boundary errors**
  - **Timing errors**
  - **Throughput or performance errors**

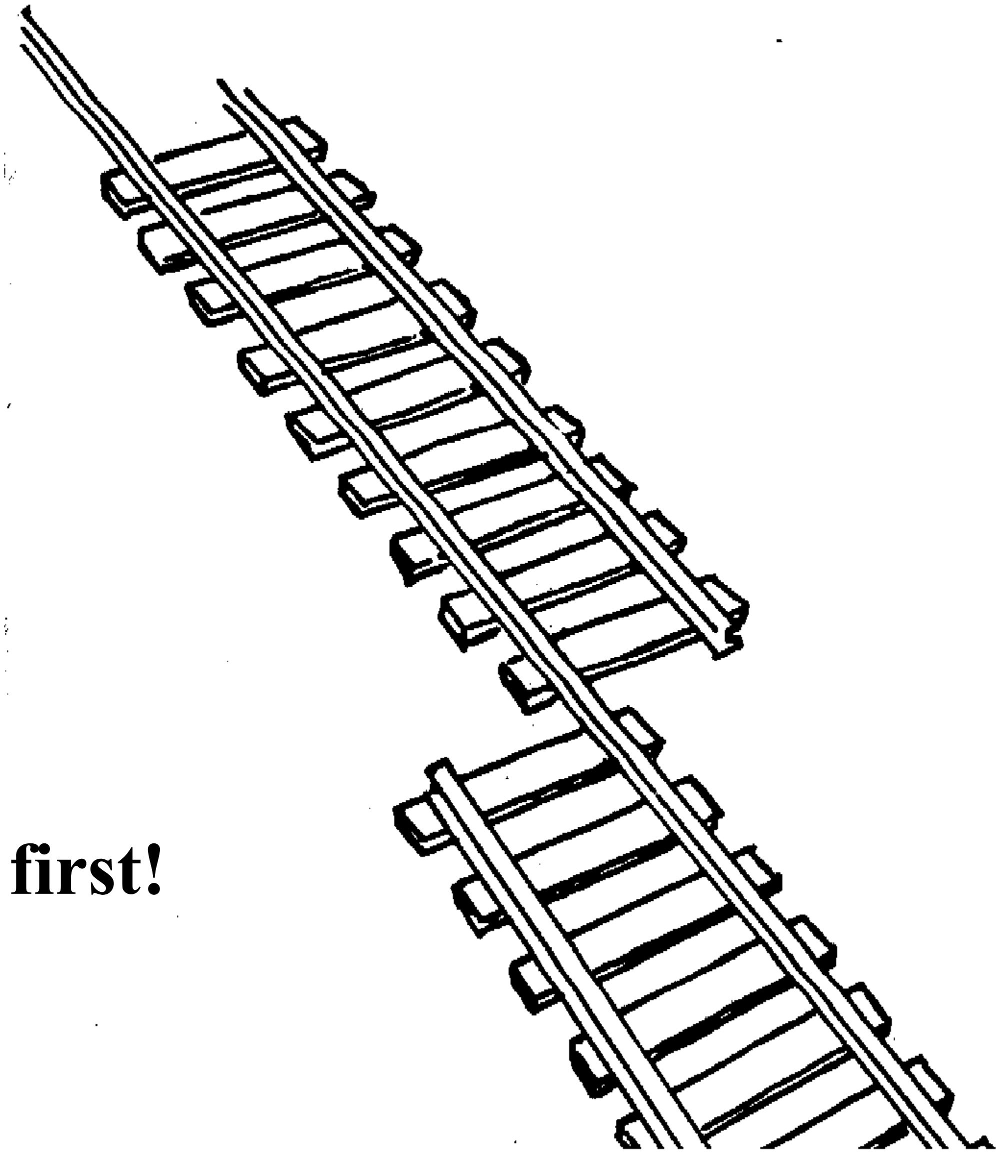
# *Dealing with Errors*

- Verification:
  - **Assumes hypothetical environment that does not match real environment**
  - **Proof might be buggy (omits important constraints; simply wrong)**
- Modular redundancy:
  - **Expensive**
- Declaring a bug to be a “feature”
  - **Bad practice**
- Patching
  - **Slows down performance**
- Testing (this lecture)
  - **Testing is never good enough**

# *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**

## *What is this?*



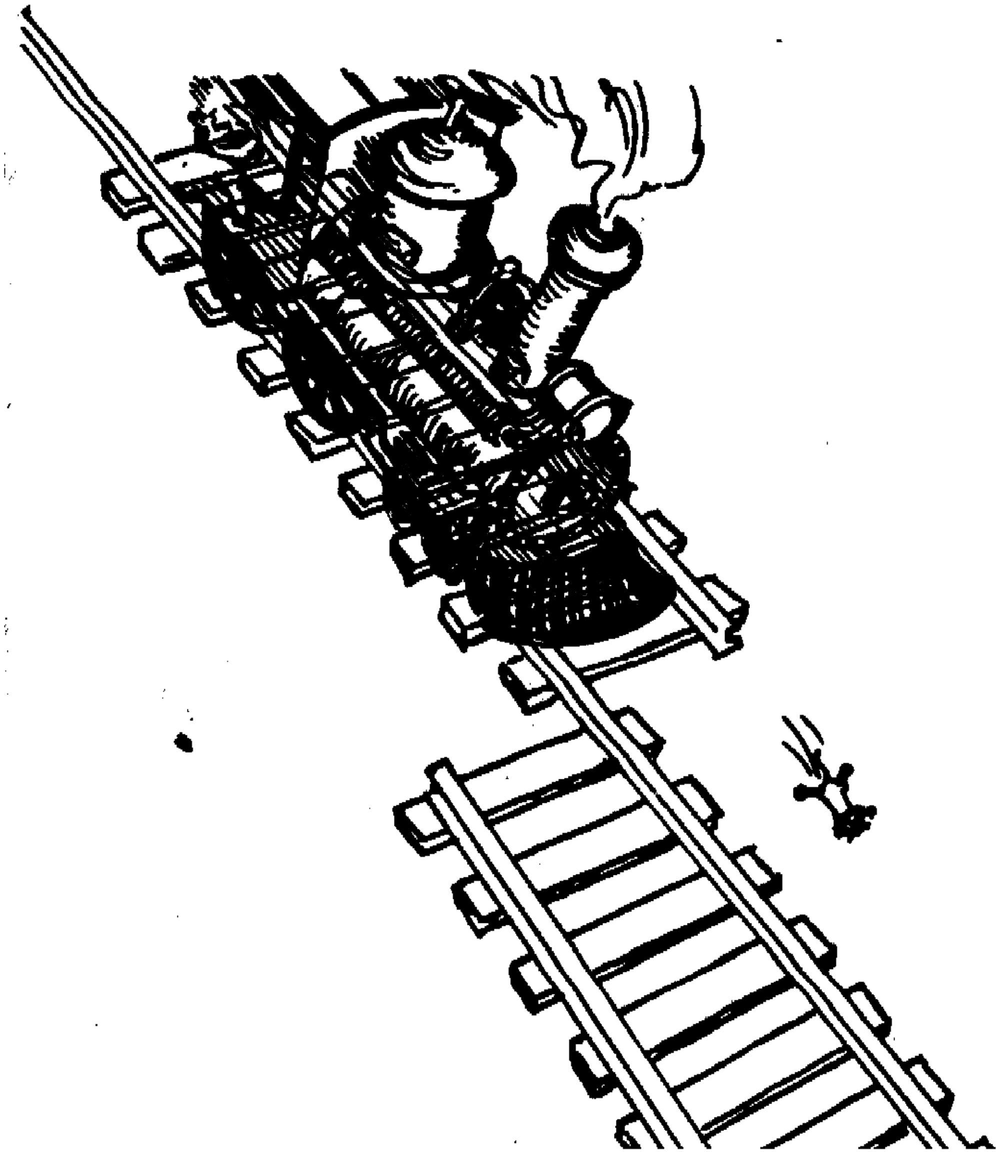
**A failure?**

**An error?**

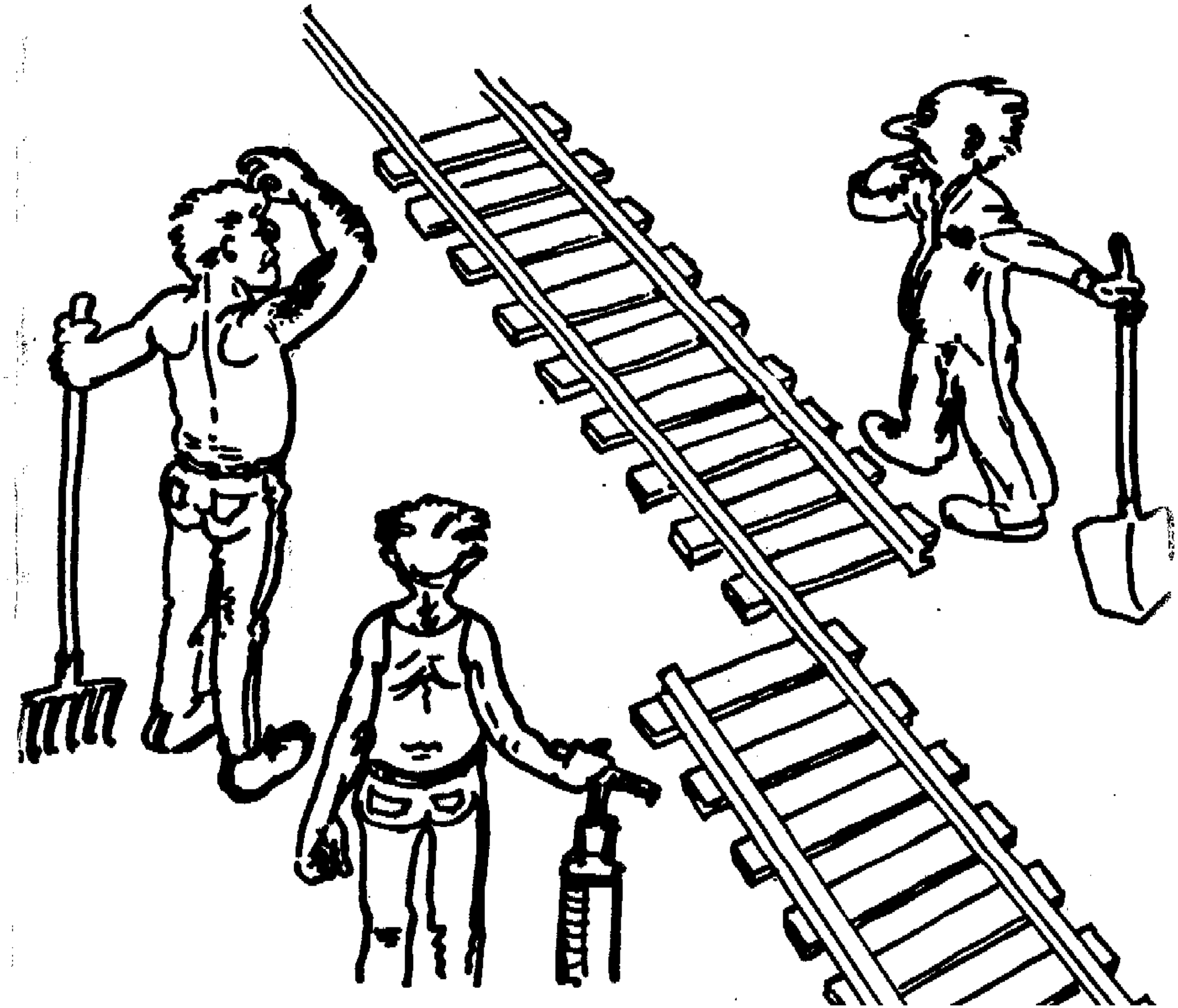
**A fault?**

**Need to specify the desired behavior first!**

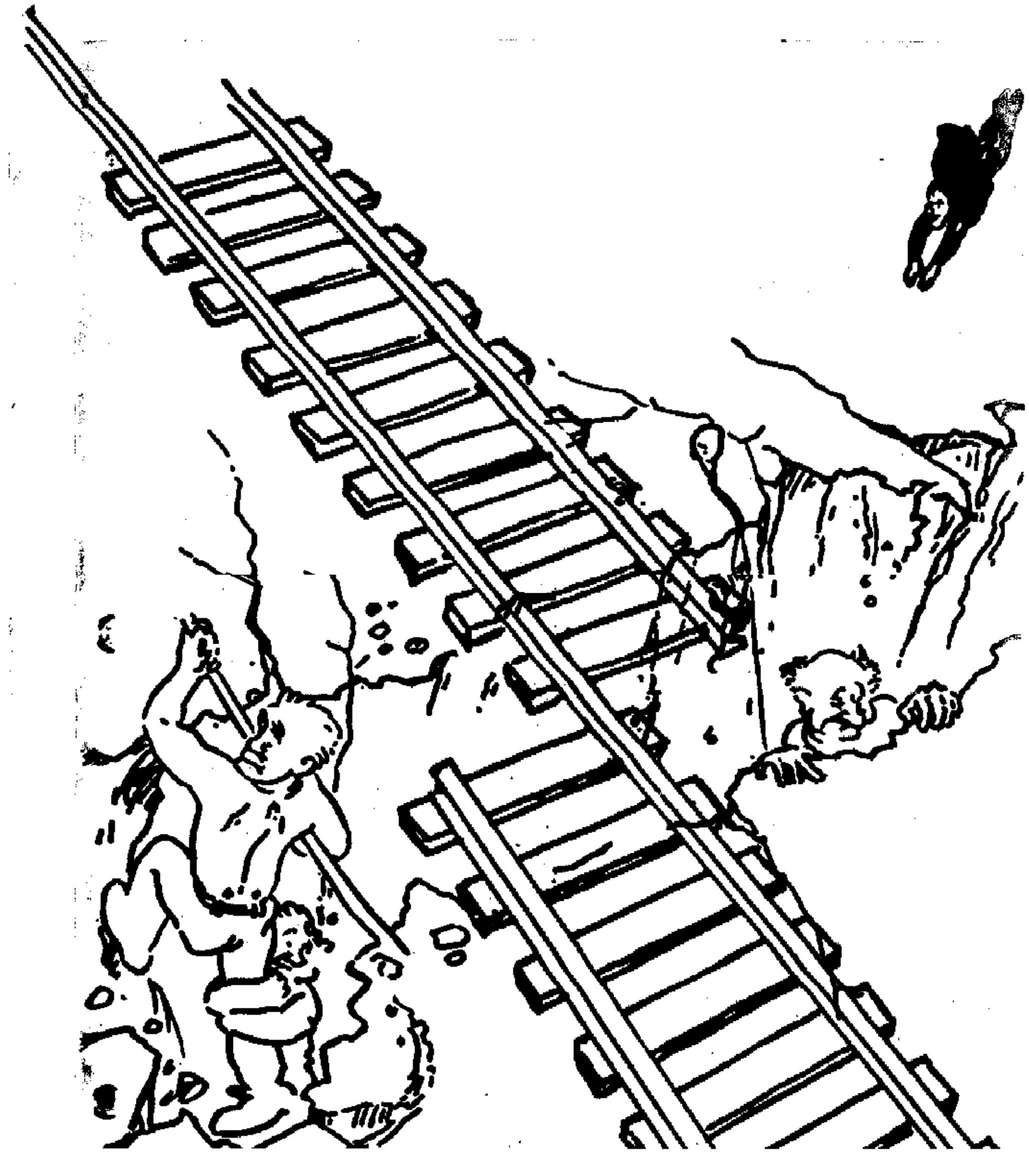
# *Erroneous State ("Error")*



# *Algorithmic Fault*

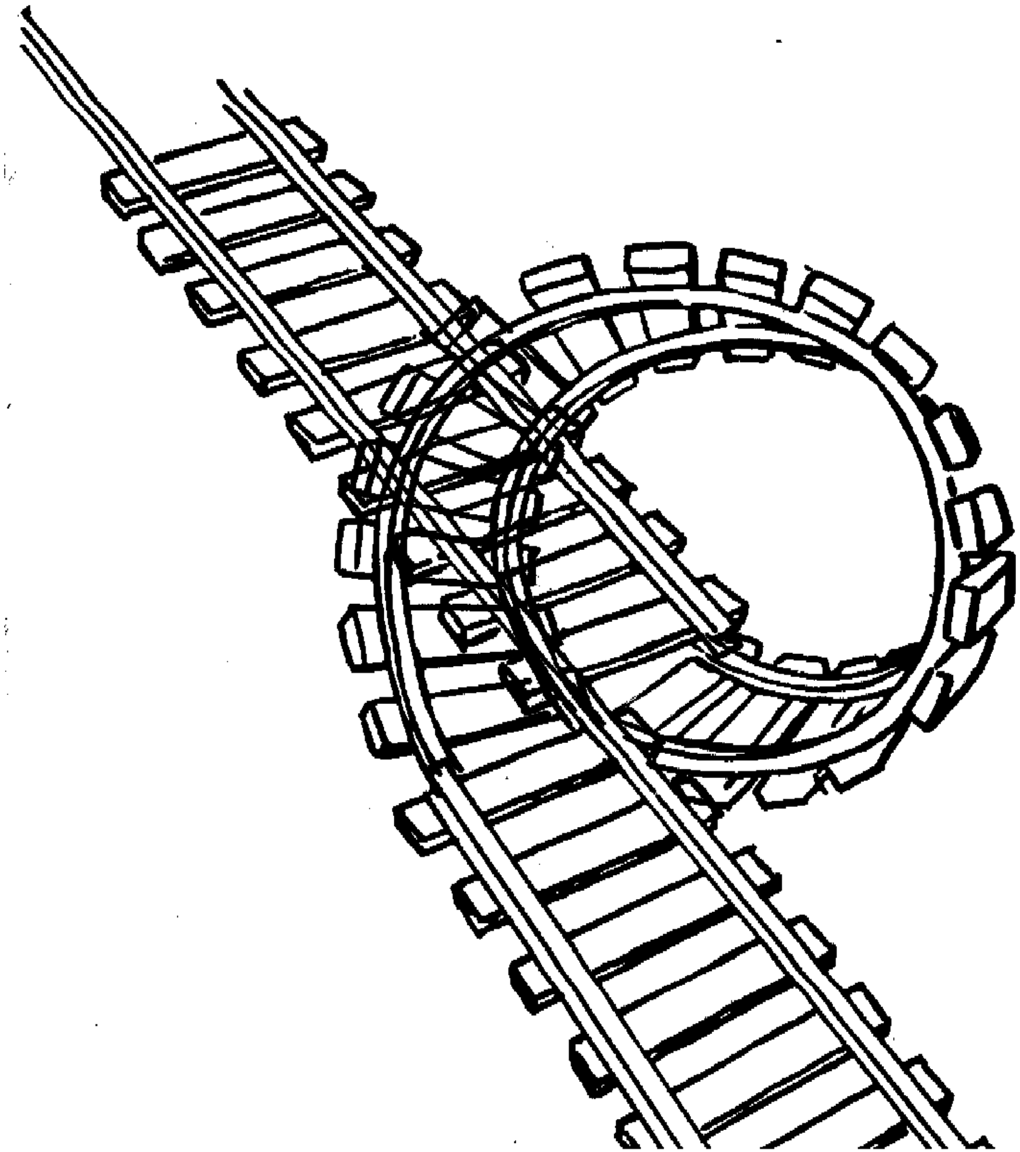


# *Mechanical Fault*

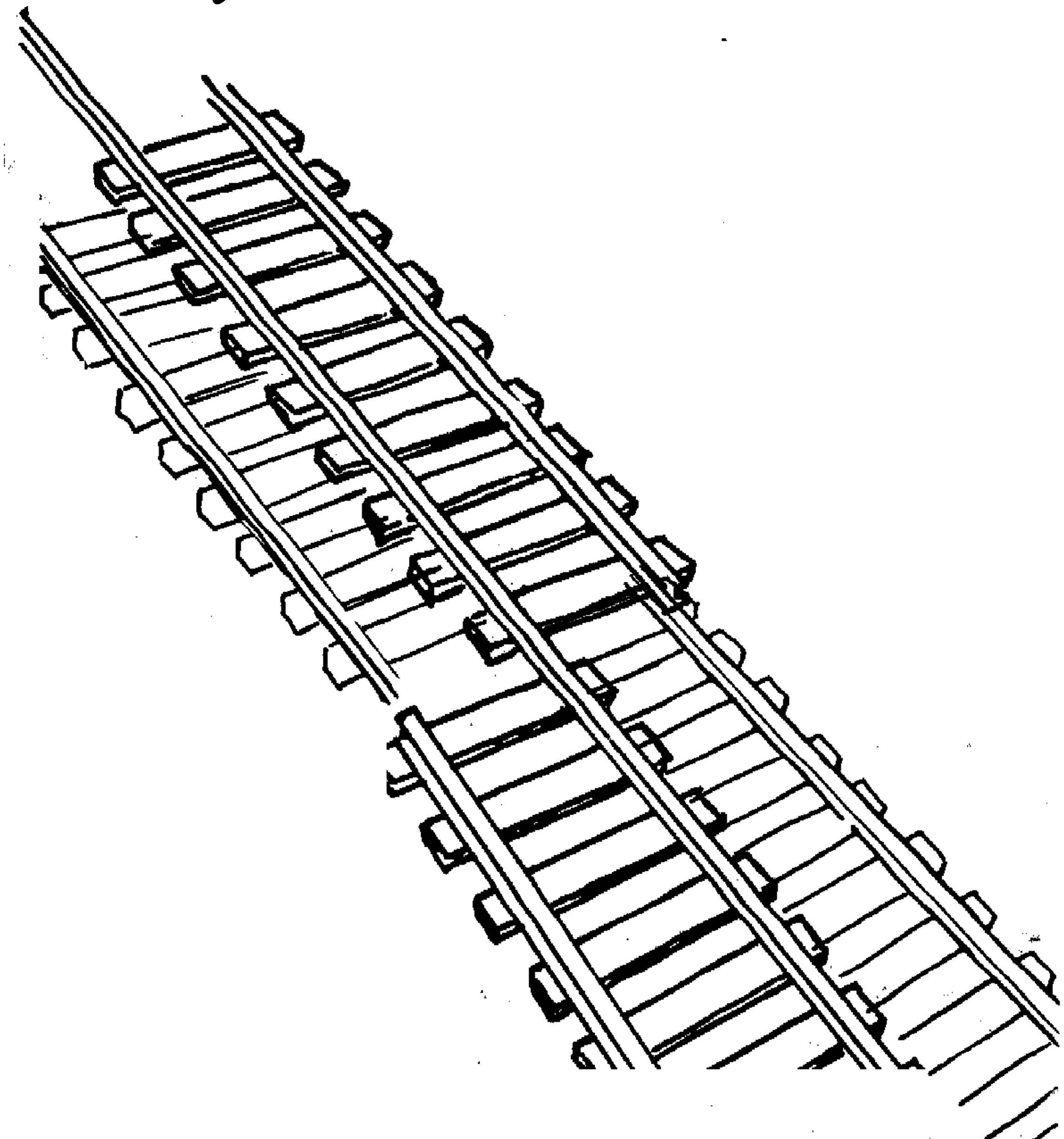


## *How do we deal with Errors and Faults?*

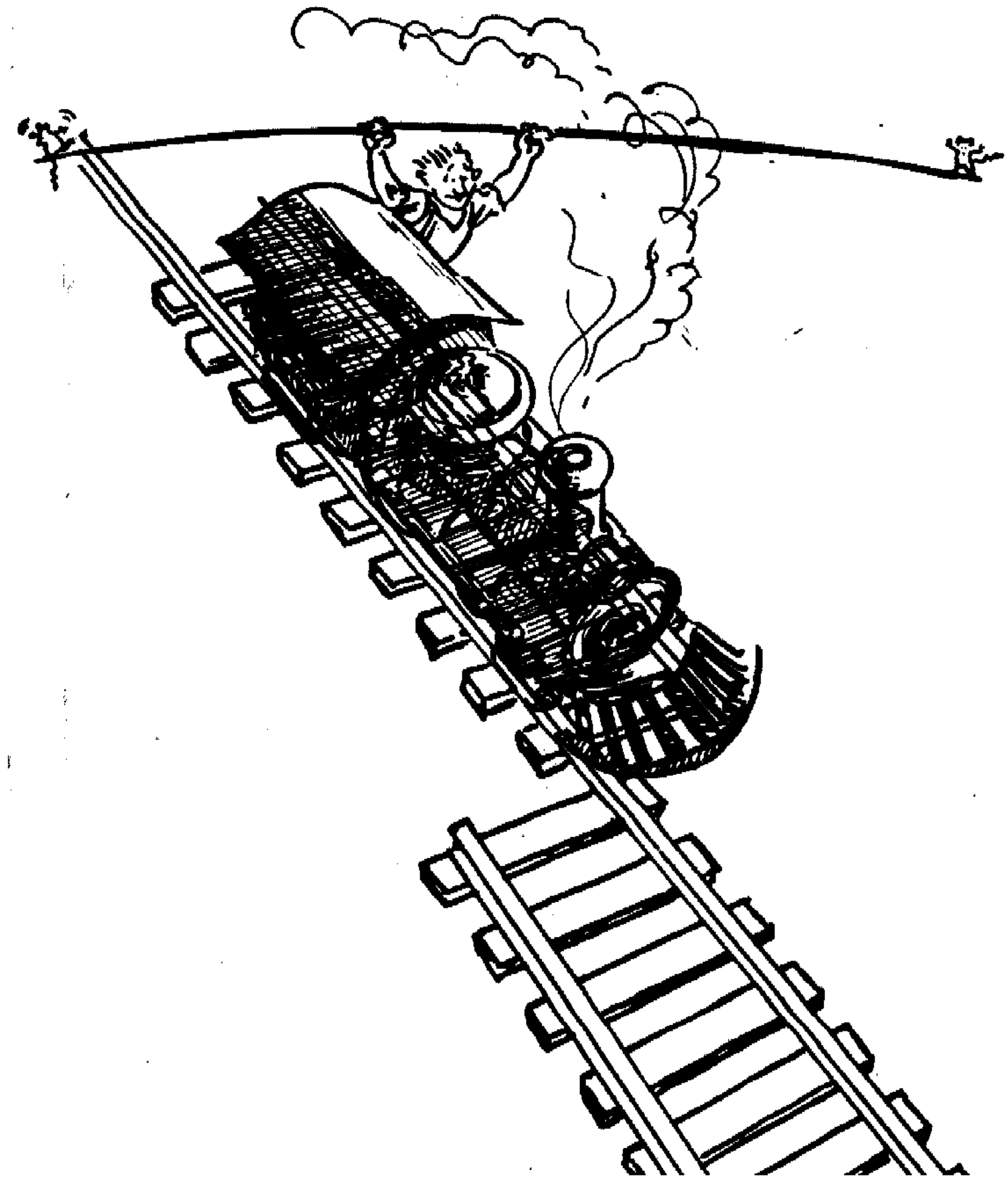
# *Verification?*



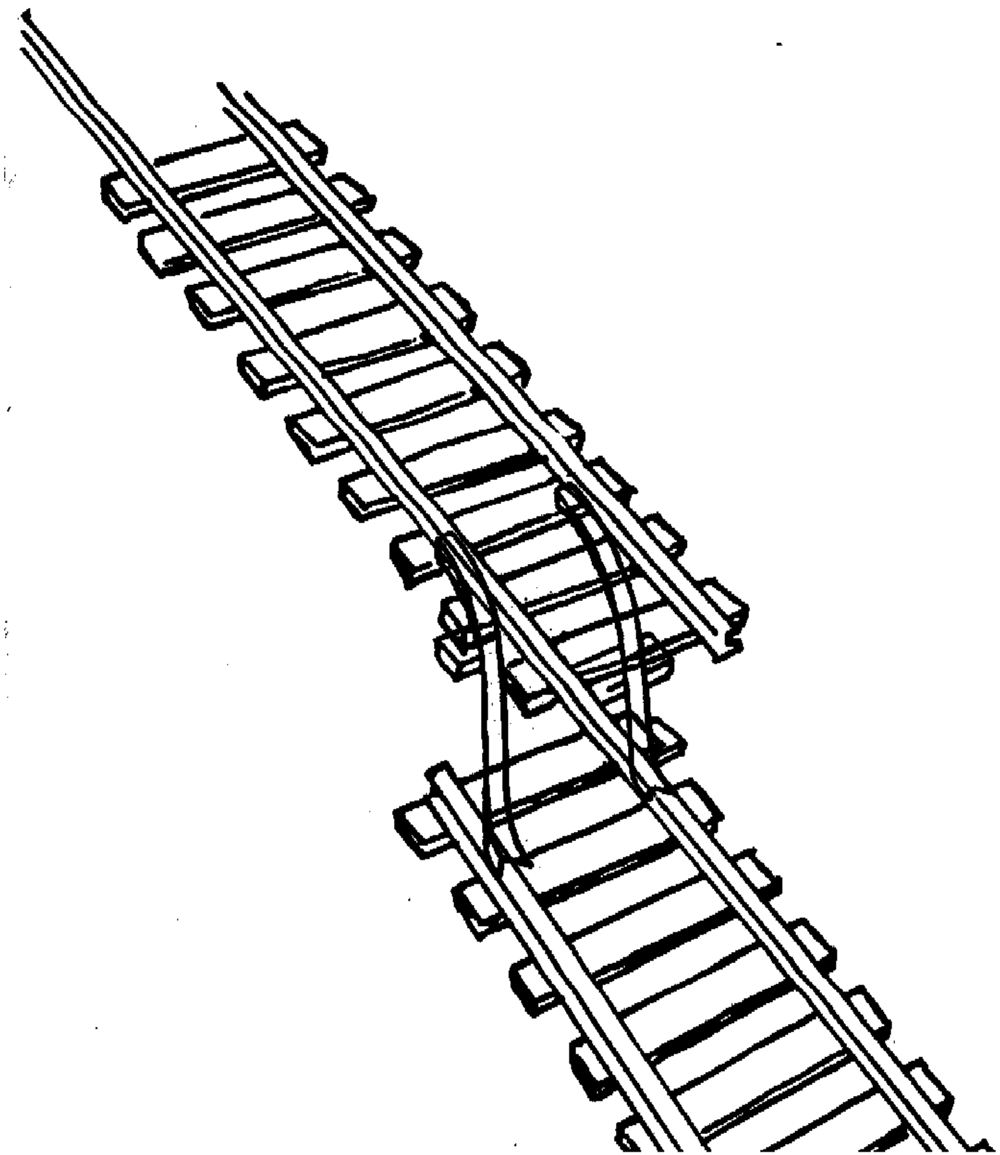
# *Modular Redundancy?*



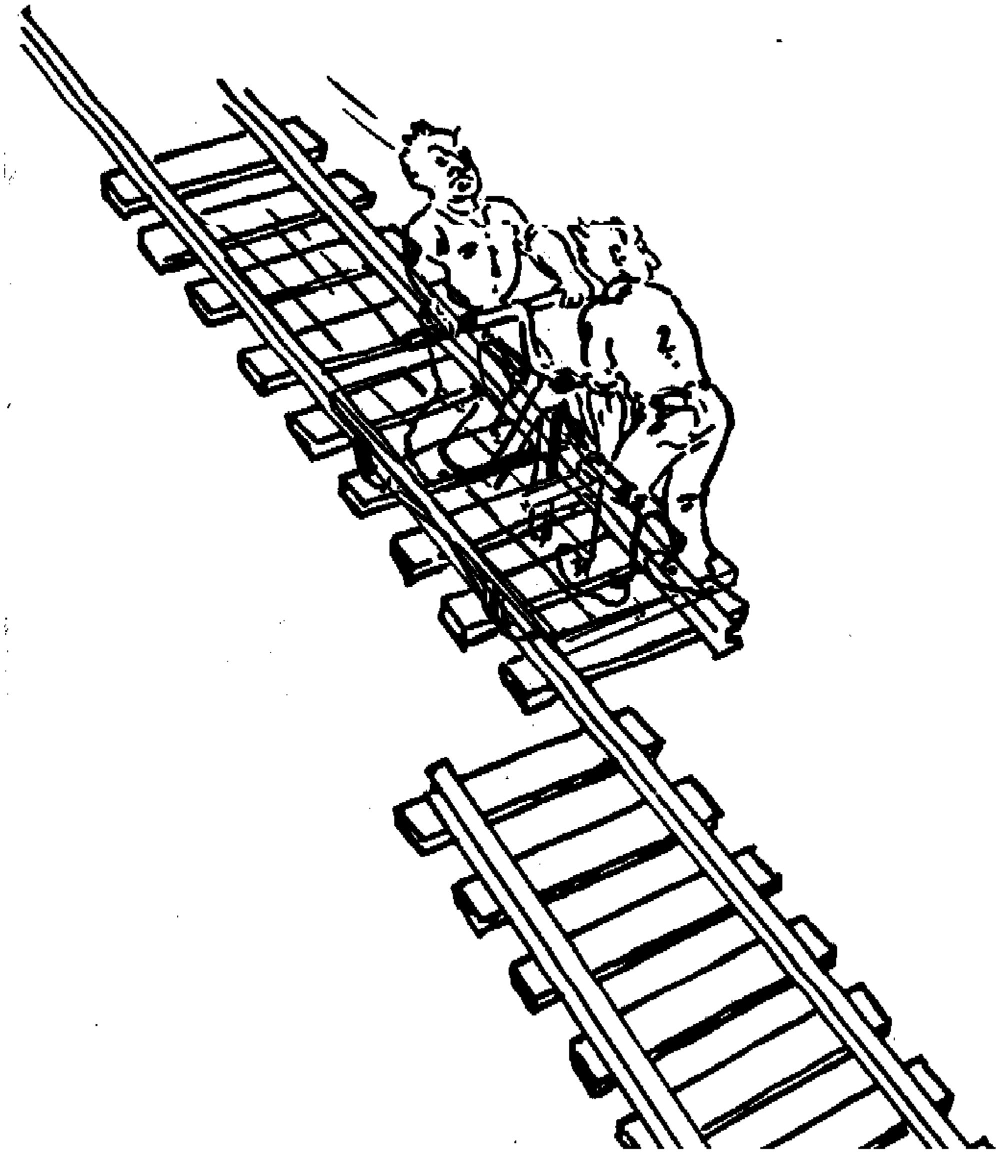
# *Declaring the Bug as a Feature?*



# *Patching?*



# *Testing?*



# *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.**

# *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