Unit Testing

Software Engineering I
Lecture 15

Bernd Bruegge
Applied Software Engineering
Technische Universitaet Muenchen

Final Exam

- 17 February 2007
- Location: HS 1
- Time: 10:00-12:30

Outline

This lecture

- Terminology
- Types of errors
- Approaches for dealing with errors
- Testing activities
- Unit testing
  - Equivalence testing
  - Path testing
  - Polymorphism testing

Next lecture

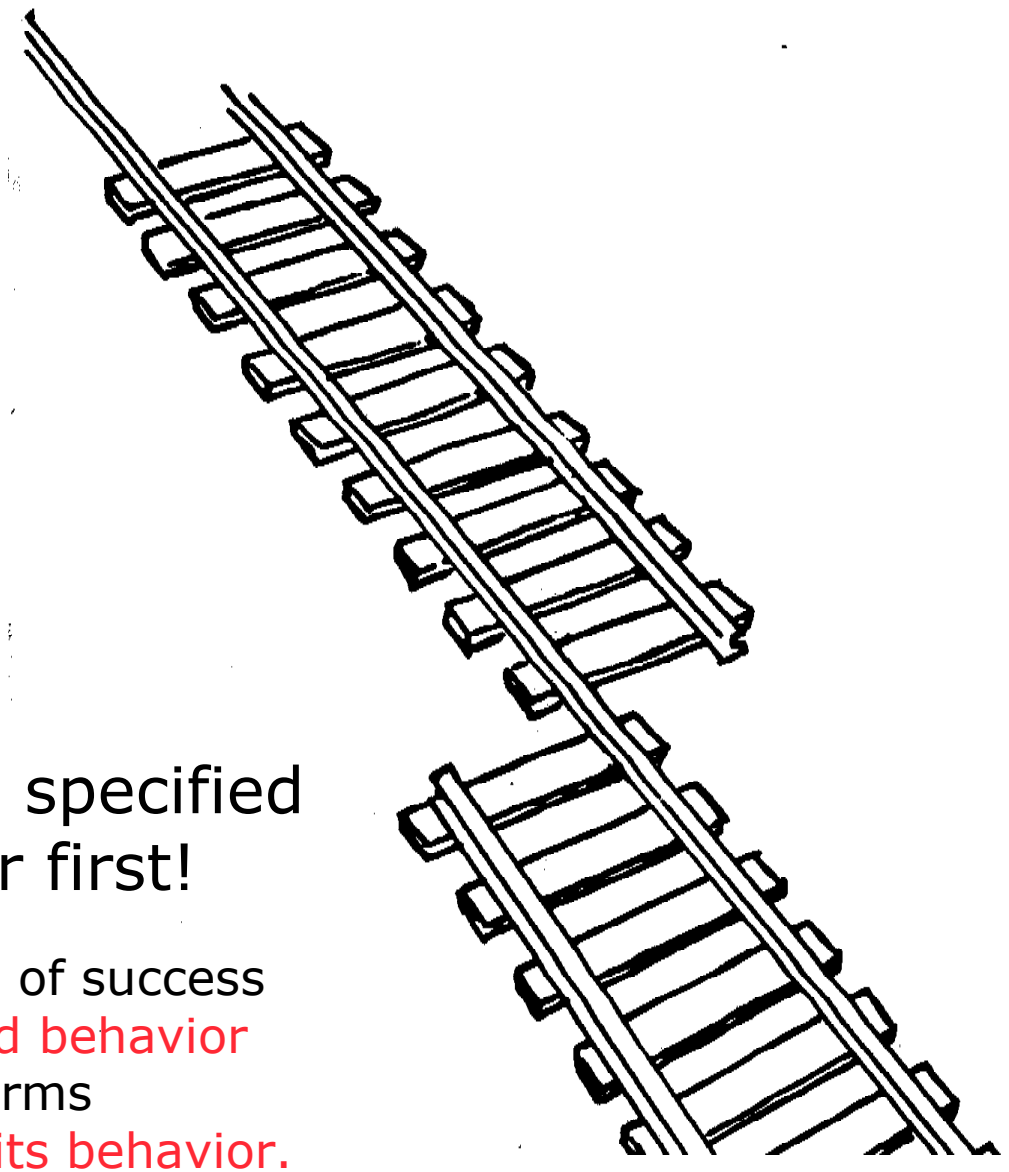
- Integration testing
  - Testing strategy
  - Design patterns & testing
- System testing
  - Function testing
  - Structure Testing
  - Performance testing
  - Acceptance testing
  - Installation testing

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
- Erroneous state (error): The system is in a state such that further processing by the system will lead to a failure
- Fault: The mechanical or algorithmic cause of an error ("bug")

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

What is this?



A failure?

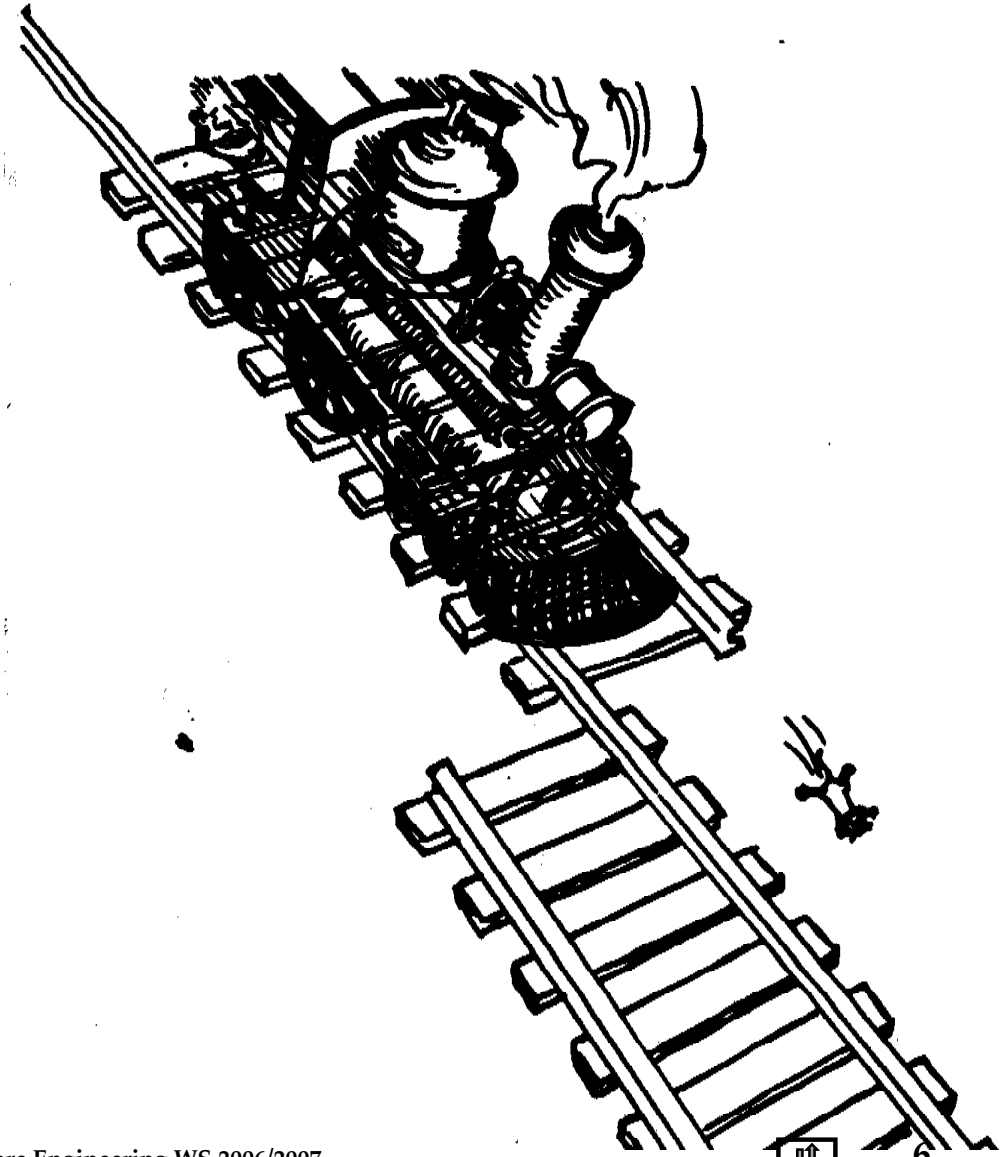
An error?

A fault?

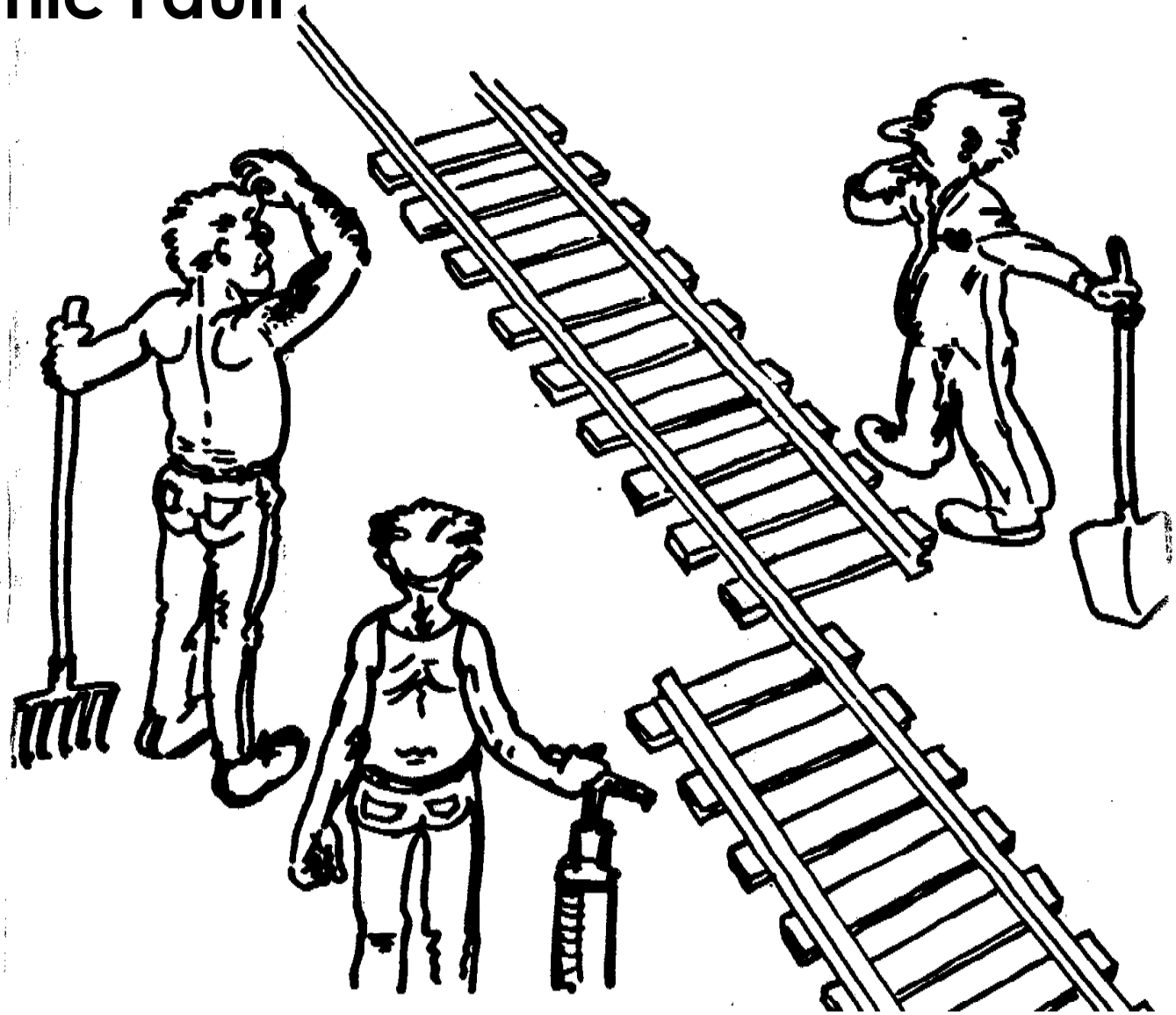
We need to describe specified and desired behavior first!

Reliability: The measure of success with which the observed behavior of a system confirms to some specification of its behavior.

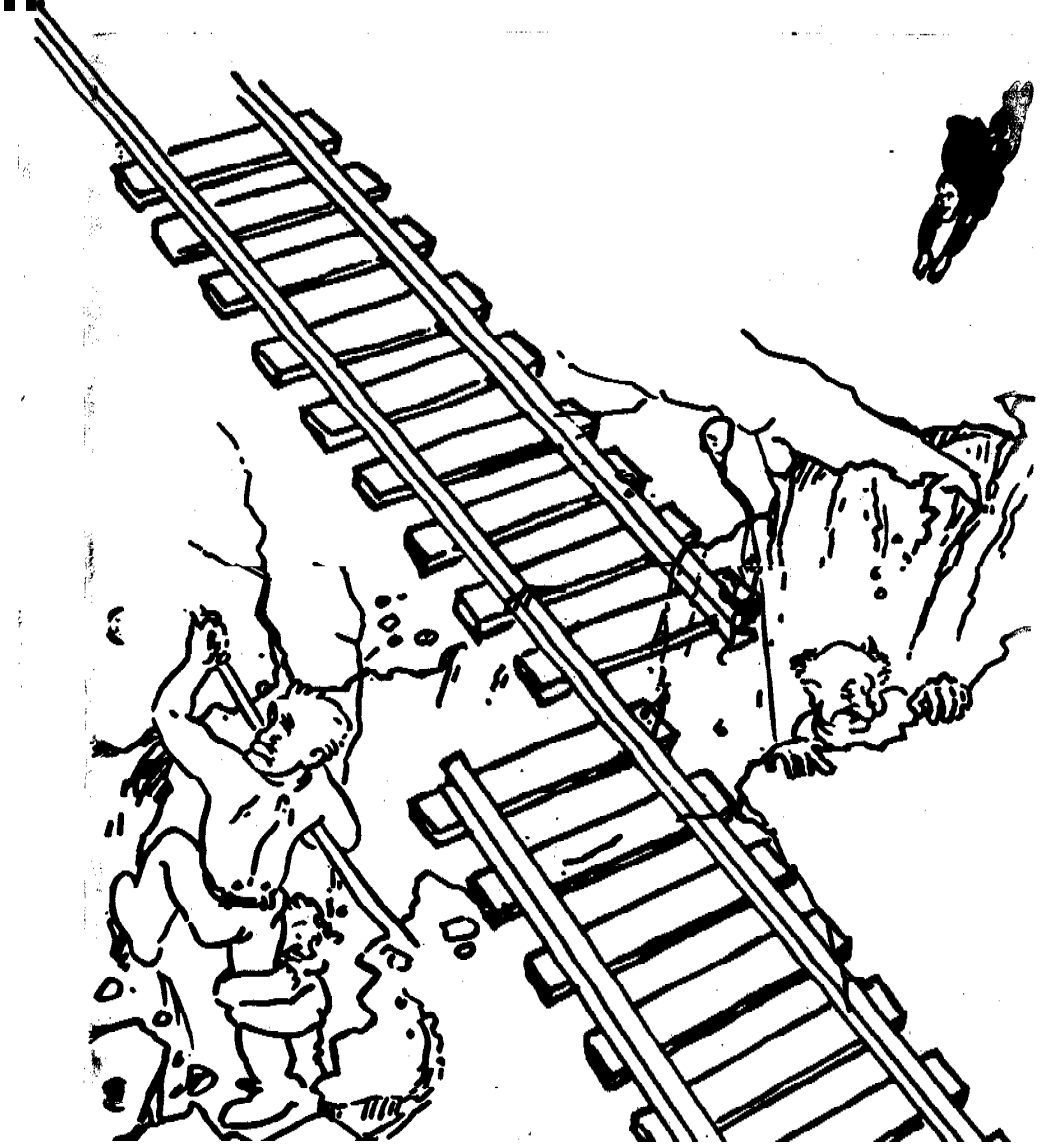
Erroneous State ("Error")



Algorithmic Fault



Mechanical Fault



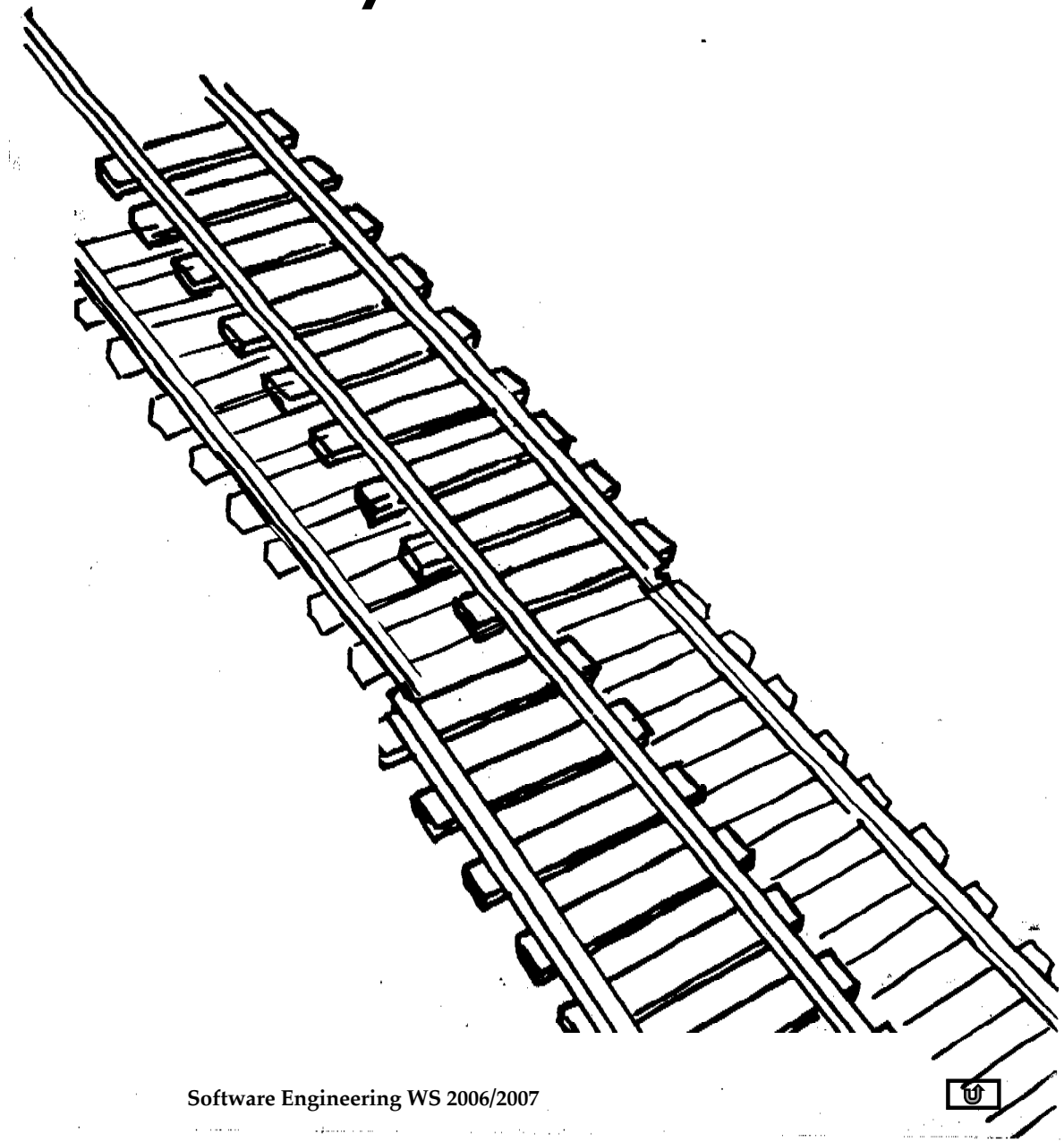
Examples of Faults and Errors

- 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
  - Incorrect branching condition
  - Missing test for null

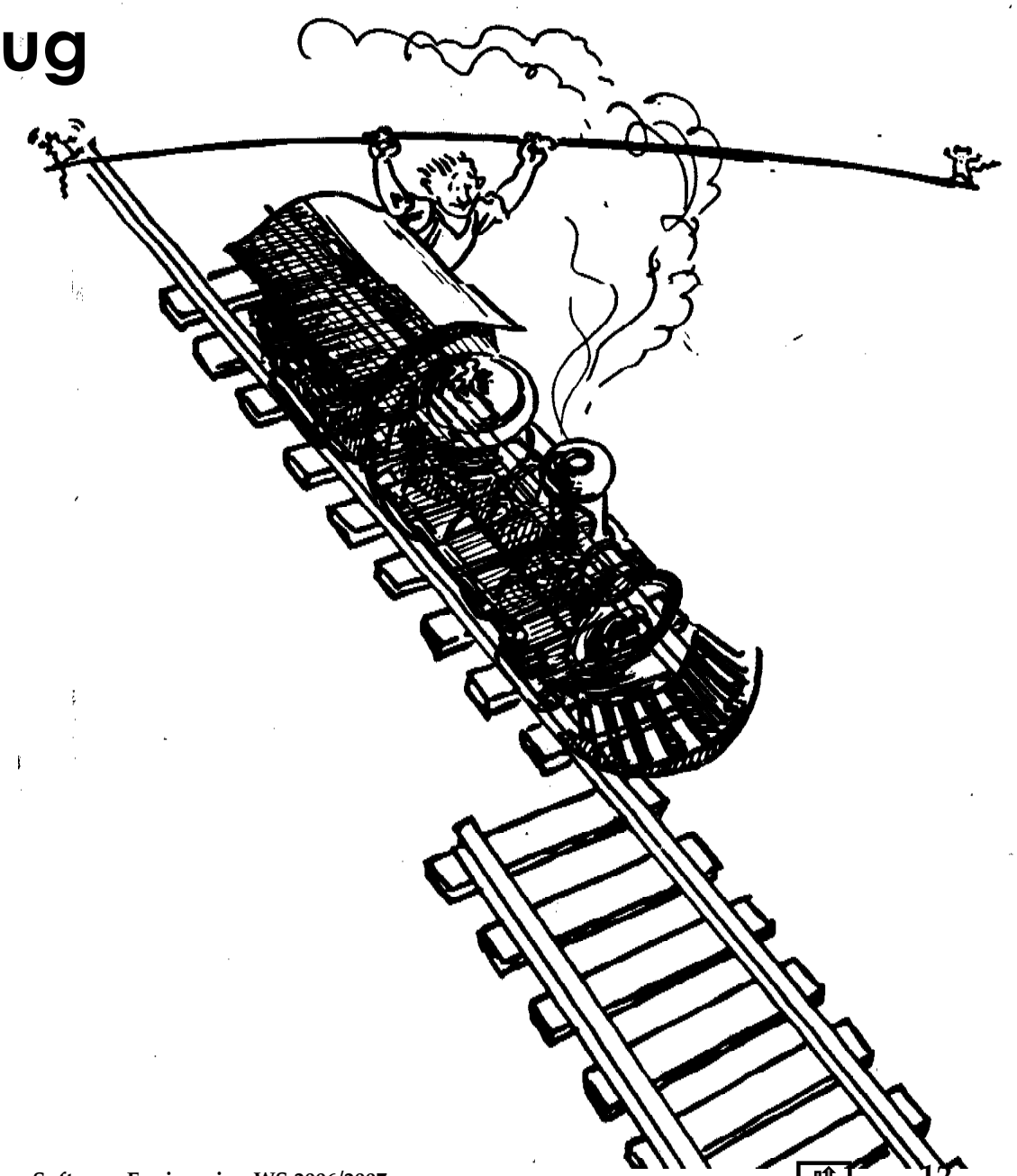
- Mechanical Faults (very hard to find)
  - Operating temperature outside of equipment specification
- Errors
  - Stress or overload errors
  - Capacity or boundary errors
  - Timing errors
  - Throughput or performance errors.

How do we deal with Errors and Faults?

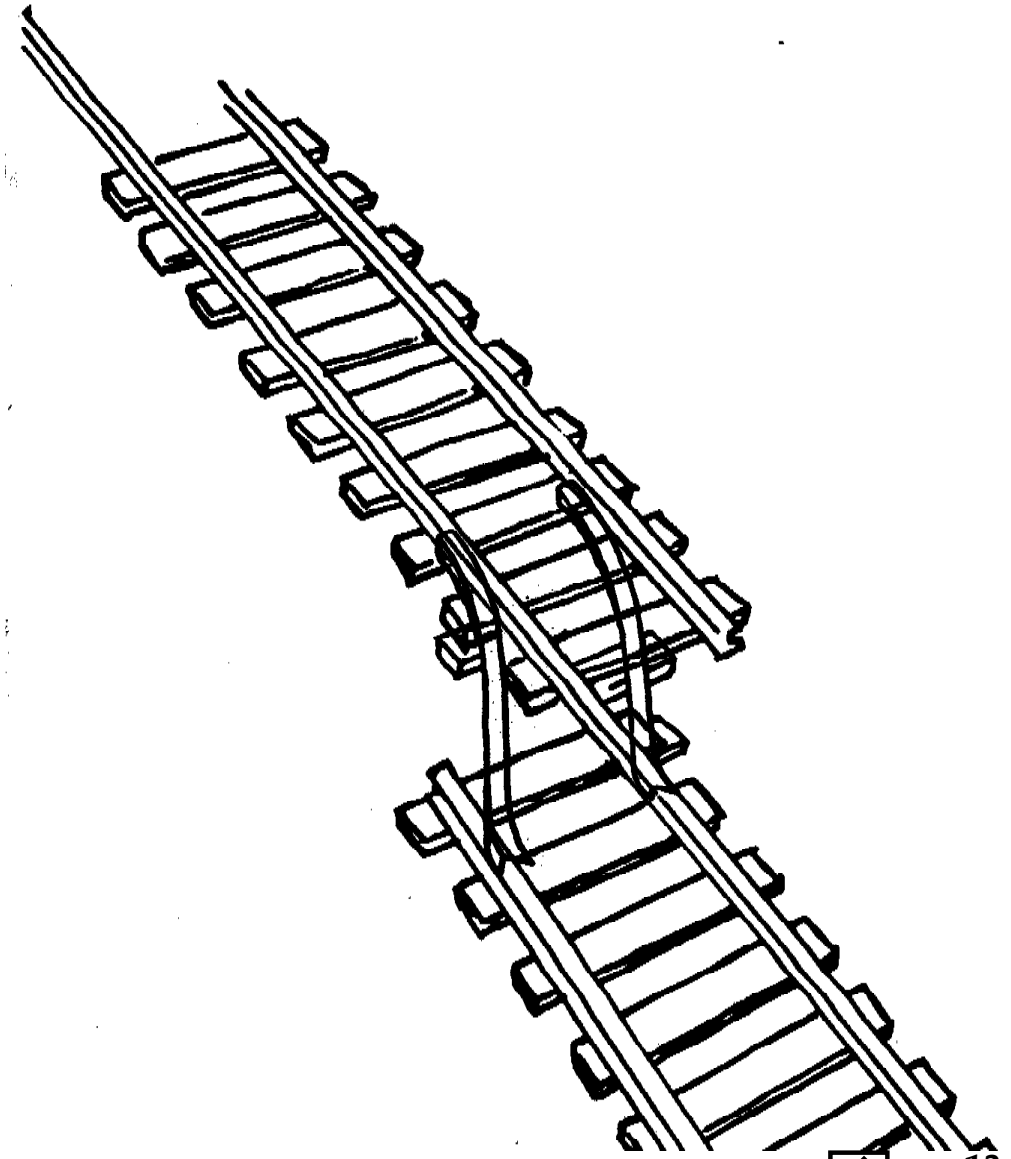
Modular Redundancy



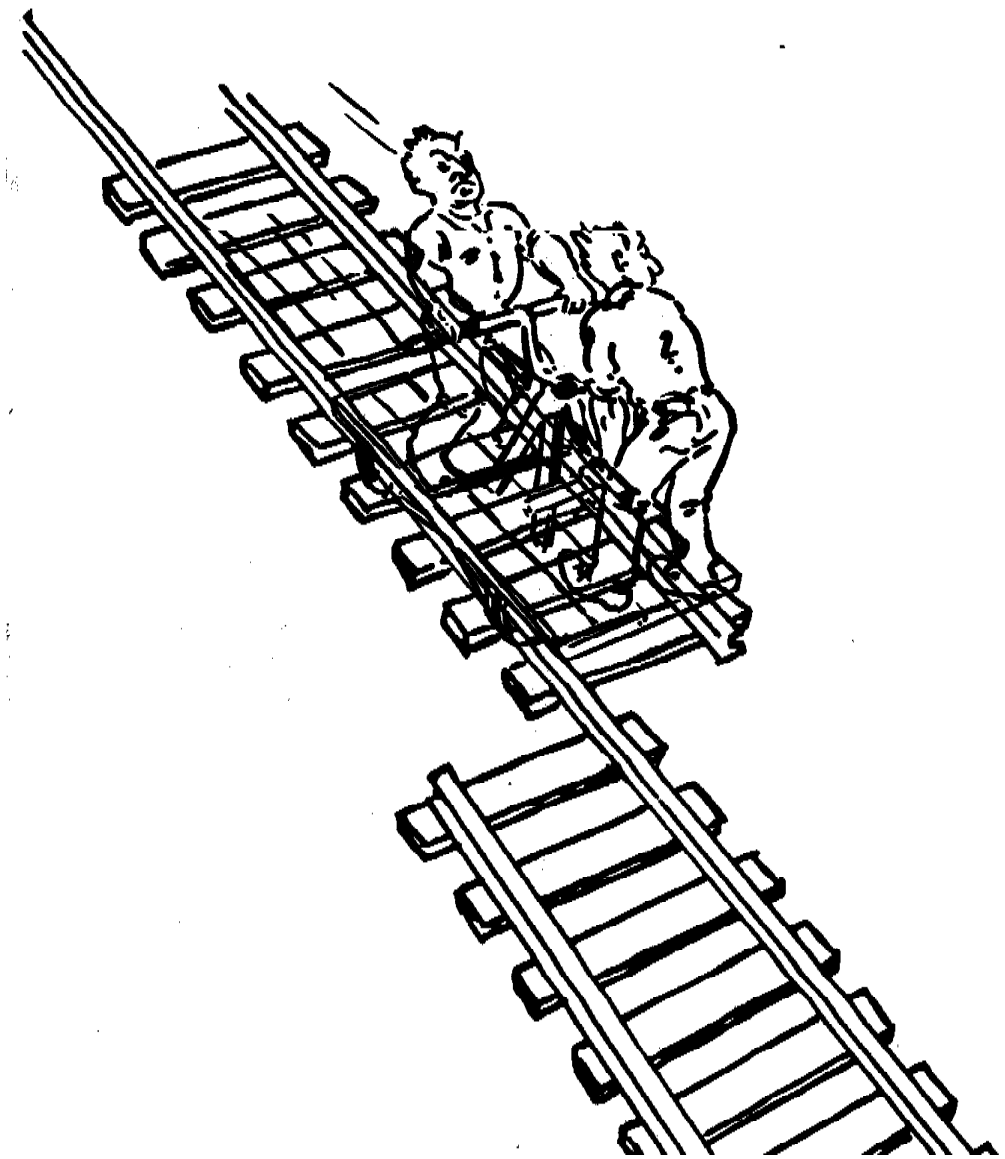
Declaring the Bug as a Feature



Patching



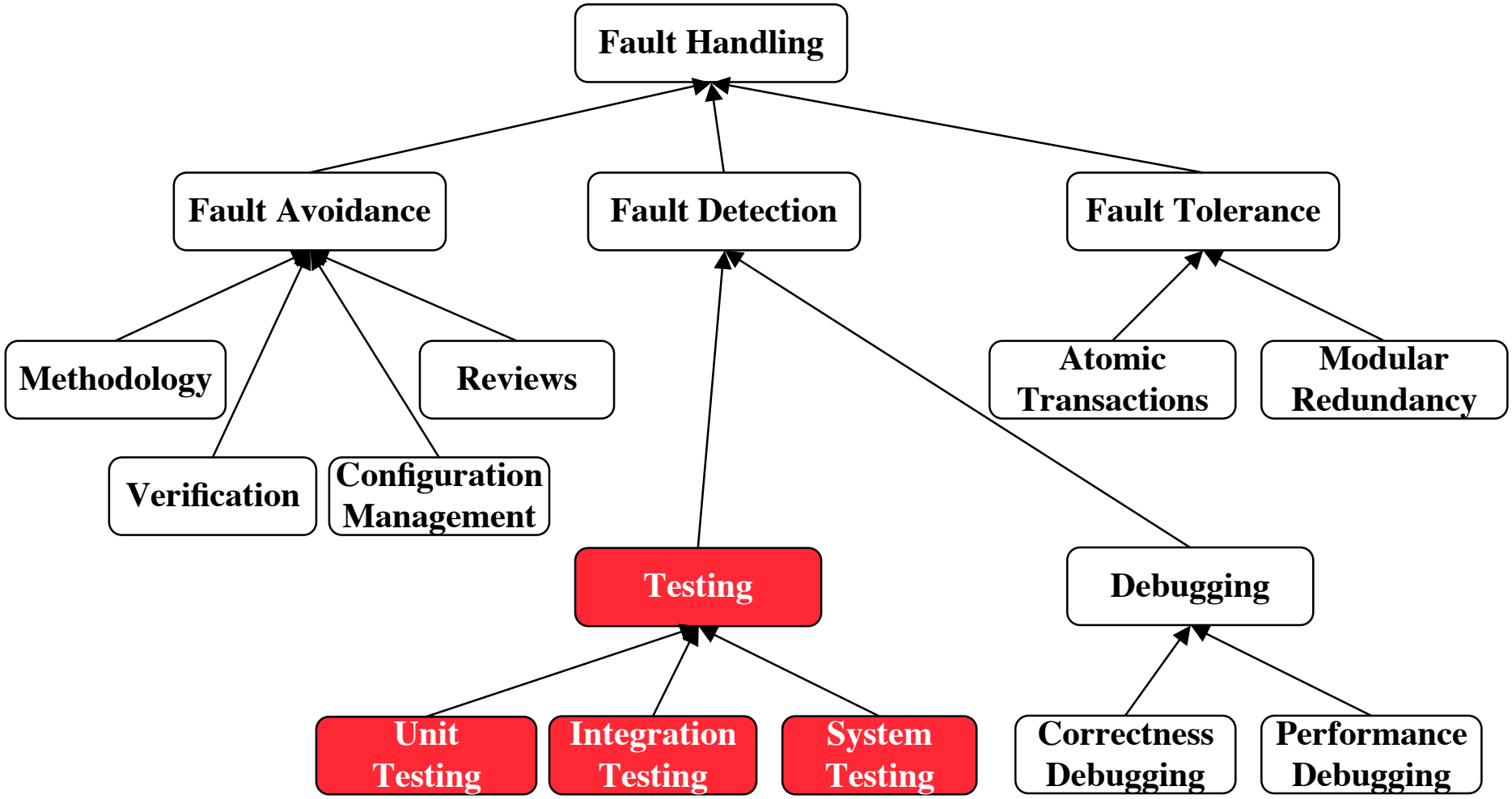
Testing



Another View on How to Deal with Faults

- Fault avoidance
  - Use methodology to reduce complexity
  - Use configuration management to prevent inconsistency
  - Apply verification to prevent algorithmic faults
  - Use Reviews
- Fault detection
  - Testing: Create failures in a planned way
  - Debugging: Start with an unplanned failures
  - Monitoring: Deliver information about state
- Fault tolerance
  - Atomic transactions
  - Modular redundancy.

Taxonomy for Fault Handling Techniques



Observations

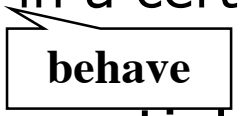
- It is impossible to completely test any nontrivial module or system
  - Practical limitations: Complete testing is prohibitive in time and cost
  - Theoretical limitations: Halting problem
- "Testing can only show the presence of bugs, not their absence" (Dijkstra).

Testing takes creativity

To develop an effective test, one must have:

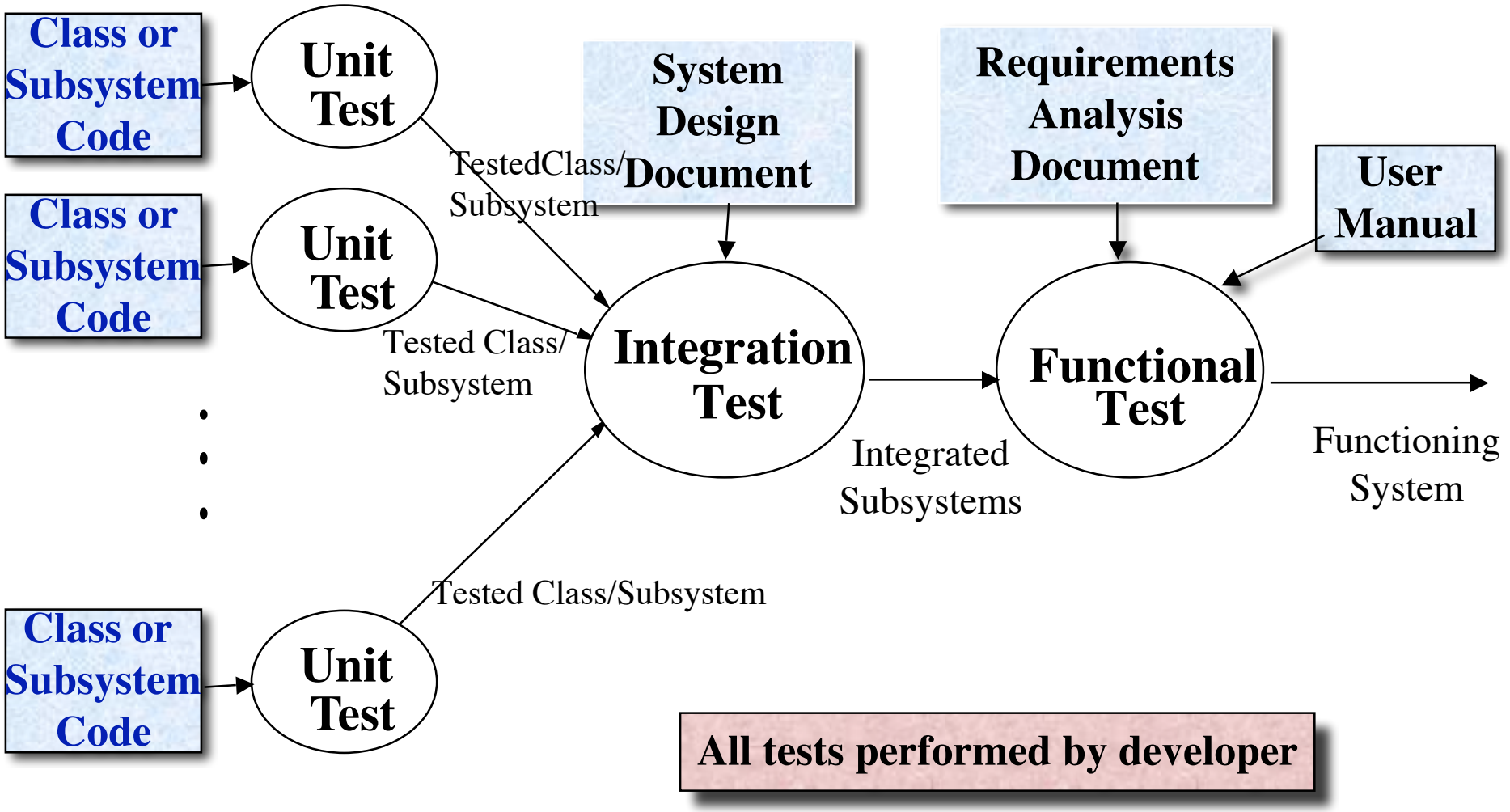
- Detailed understanding of the system
  - Application and solution domain knowledge
- Knowledge of the testing techniques
- Skill to apply these techniques

- 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

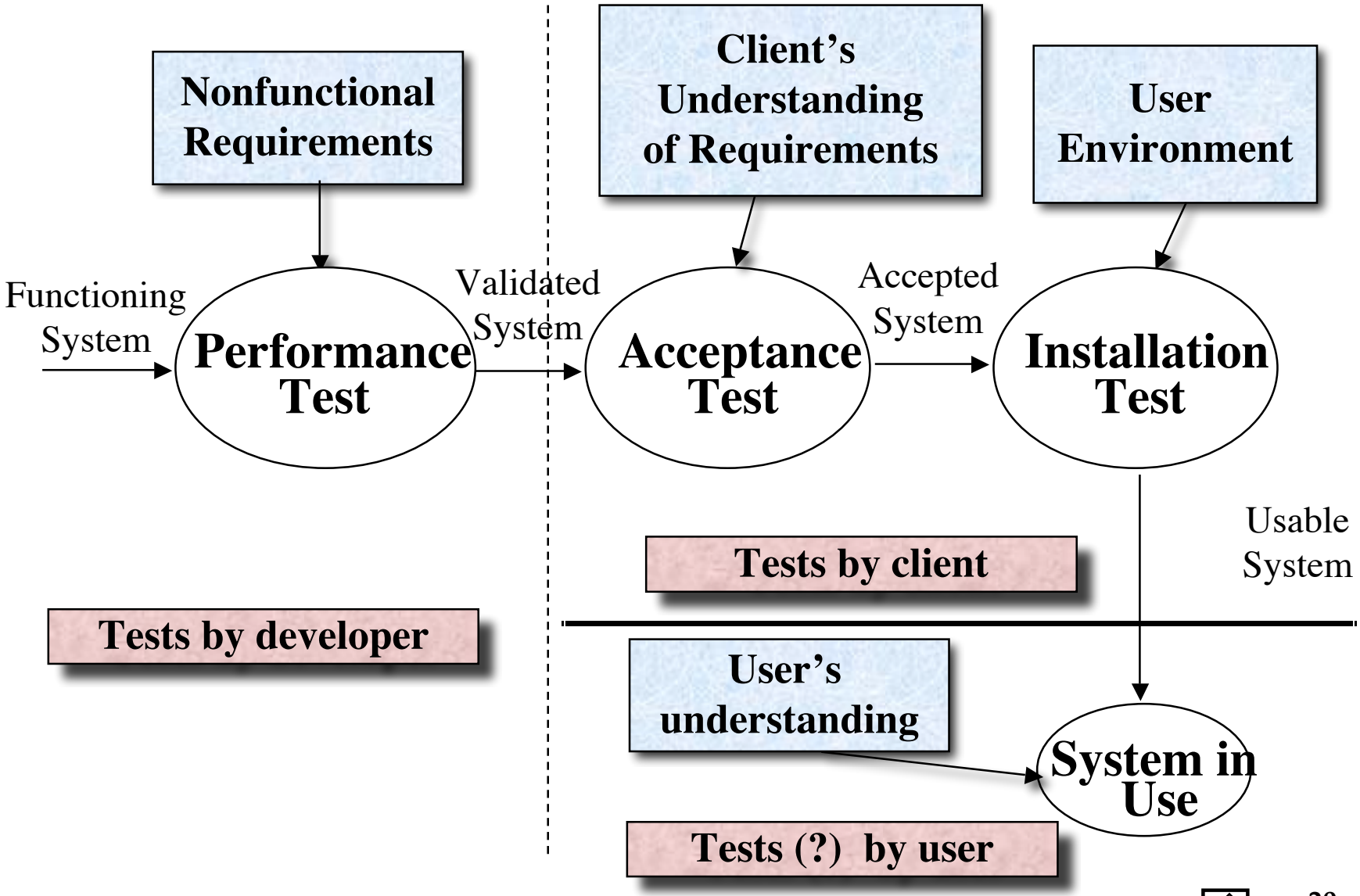


- Programmer often stick to the data set that makes the program work
- A program often does not work when tried by somebody else.

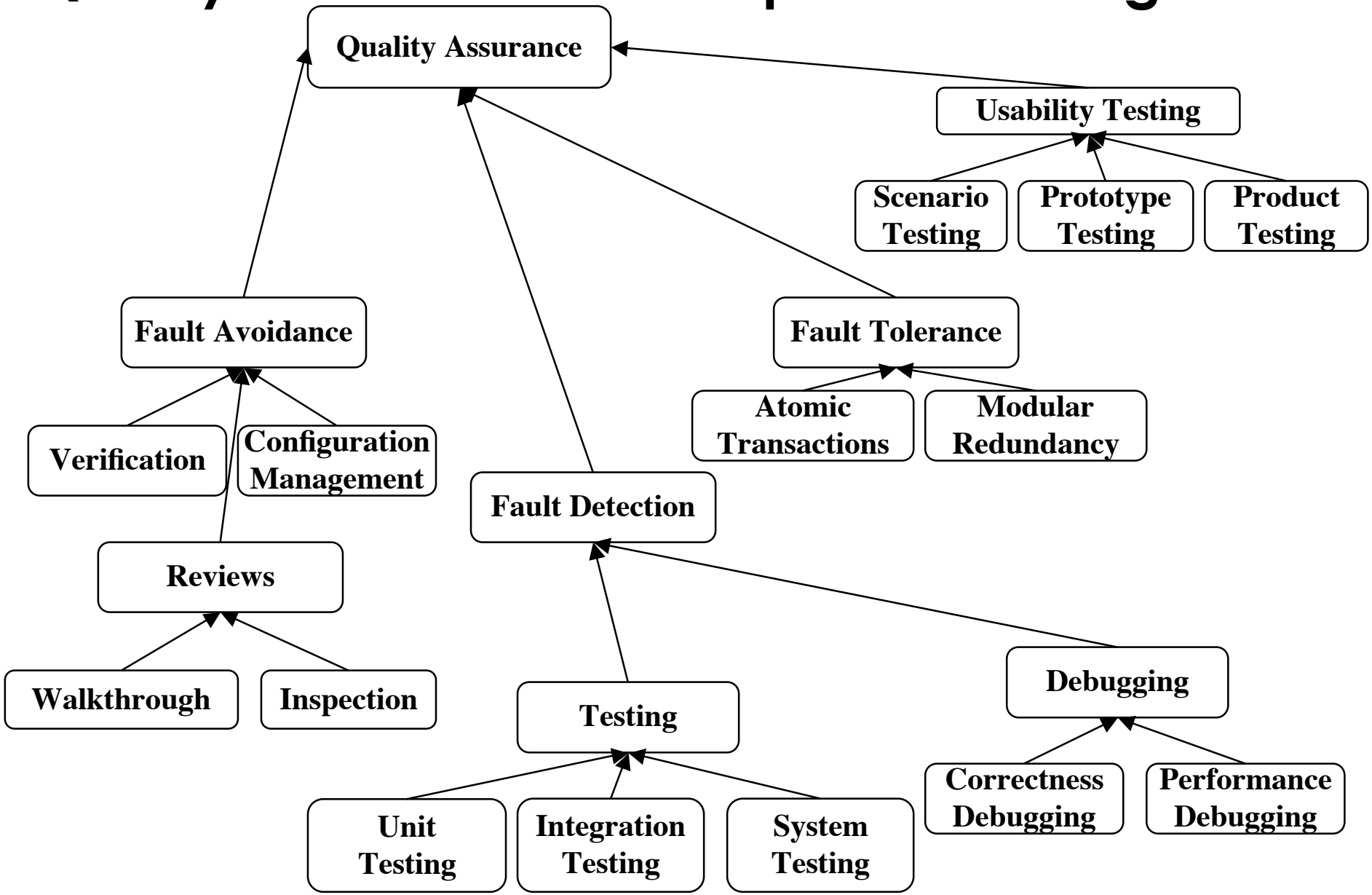
Testing Activities



Testing Activities continued



Quality Assurance encompasses Testing



Types of Testing

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

System Testing

- System Testing
  - The entire system
  - Carried out by developers
  - Goal: Determine if the system meets the requirements (functional and nonfunctional)
- 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 the requirements and is ready to use
- Implementation and testing usually go hand in hand:
  - In XP: The tests are implemented first!

Unit Testing

- Static Analysis:
  - Manual execution
  - Walk-through
  - Code inspection
- Dynamic Analysis:
  - Black-box testing
  - White-box testing
  - Data-structure based testing

Black-box testing

- Focus: I/O behavior
  - If for any given input, we can predict the output, then the component passes the test
- Goal: Reduce number of test cases by equivalence partitioning:
  - Divide input conditions into equivalence classes
  - Choose test cases for each equivalence class.

Black-box testing: Test case selection

a) Input is valid across range of values
- Developer selects test cases from 3 equivalence classes:
  - Below the range
  - Within the range
  - Above the range

b) Input is only valid, if it is a member of a discrete set
- Developer selects test cases from 2 equivalence classes:
  - Valid discrete values
  - Invalid discrete values

- No rules, only guidelines.

Black box testing: An example

```
public class MyCalendar {

  public int getNumDaysInMonth(int month, int year)
      throws InvalidMonthException
  { … }
}
```

Representation for `month`:
1: January, 2: February, …., 12: December

Representation for `year`:
1904, … 1999, 2000,…, 2006, …

How many test cases do we need for the black box testing of `getNumDaysInMonth()`?

Black box testing of `getNumDaysInMonth`

Test case	Month	Year
31 day months, non-leap year	7	2001
30 day months, non-leap year	6	2001
February, non-leap year	2	2001
31 day months, leap year	7	2004
30 day months, leap year	6	2004
February, leap year	2	2004
Non-positive invalid month		
Positive invalid month		

Valid Discrete values

Invalid Discrete values

How about:
Valid month, invalid year?

Do we have all the test cases for invalid discrete values?

Equivalence testing: Drawbacks

- Inputs are treated independently

⇒Combinations or input values are not well tested

- In our example, we also want to test if leap years are detected correctly:

Equivalence class	**Month**	**Year**
Leap years divisible by 100	2	1900
Non-leap years divisible by 400	2	2000

White-box testing

- Focus: Thoroughness (Coverage).
  - Every type of statement in the component is executed at least once
- Types of white-box testing
  - Algebraic testing
  - Loop testing
  - Path testing
  - Branch testing
  - Polymorphism testing.

White-box testing (2)

- Algebraic Testing (Statement Testing)
  - Test single statements
    - Choice of operators in polynomials, etc
- Loop Testing
  - Cause execution of the loop to be skipped completely
  - Loop to be executed exactly once
  - Loop to be executed more than once

White-box testing (3)

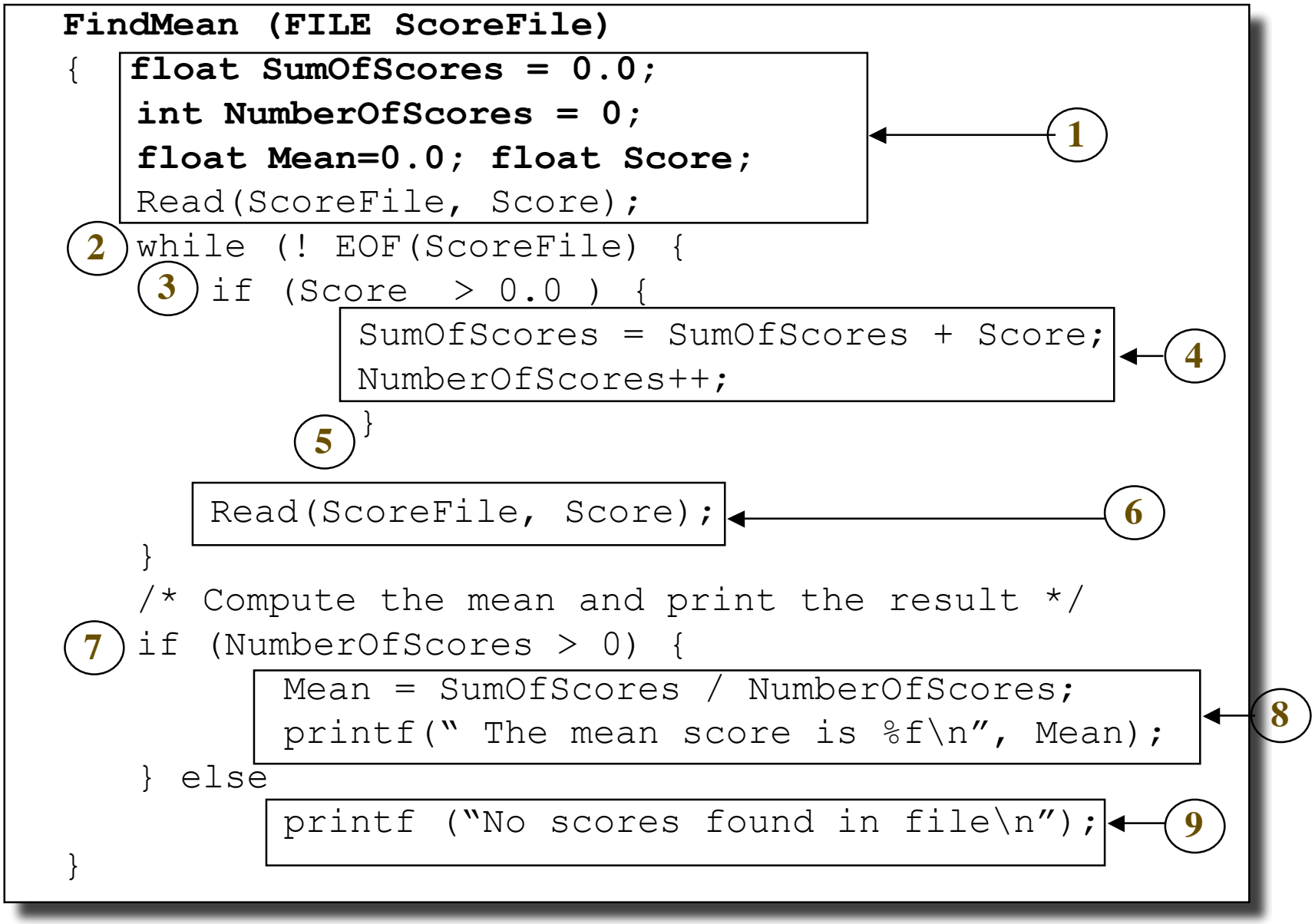
- Branch Testing (Conditional Testing)
  - Make sure that each possible outcome from a condition is tested at least once
- Path testing
  - Make sure all paths in the program are executed

```
if ( b = TRUE) {
  System.out.println("Yes");
} else {
  System.out.println("No");
}
```

Test cases:

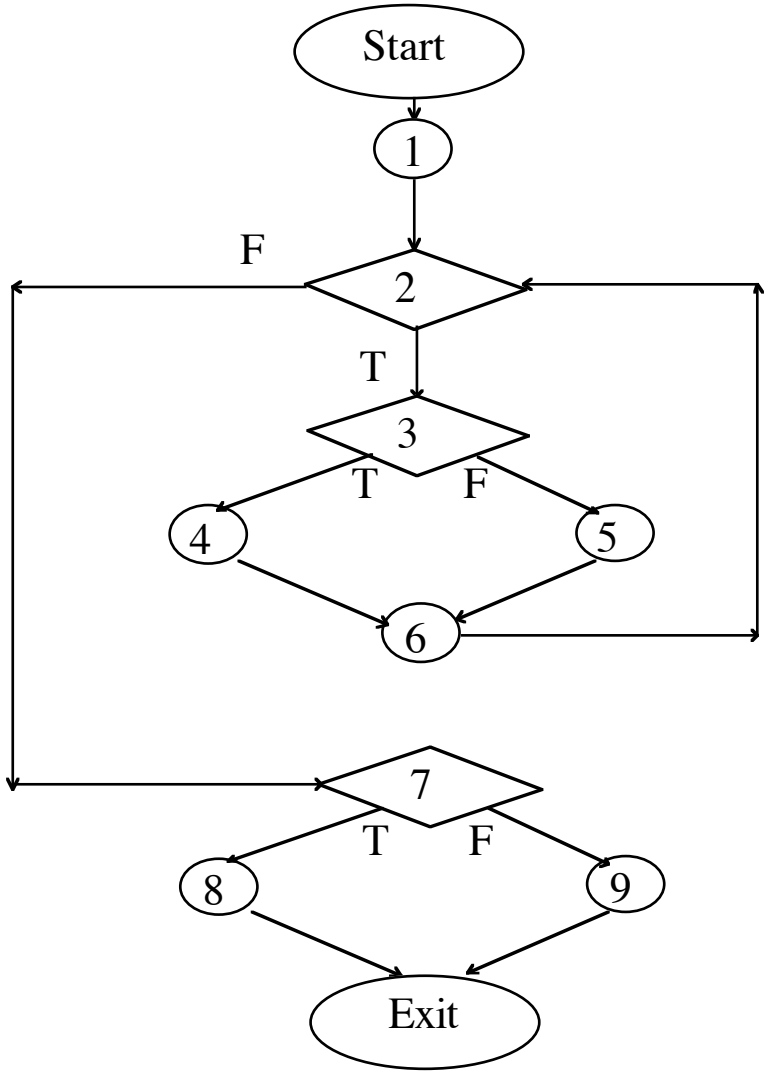
1) **b == True**
2) **b == False**

Determining Paths:
Find Decisions Points & Compound Statements

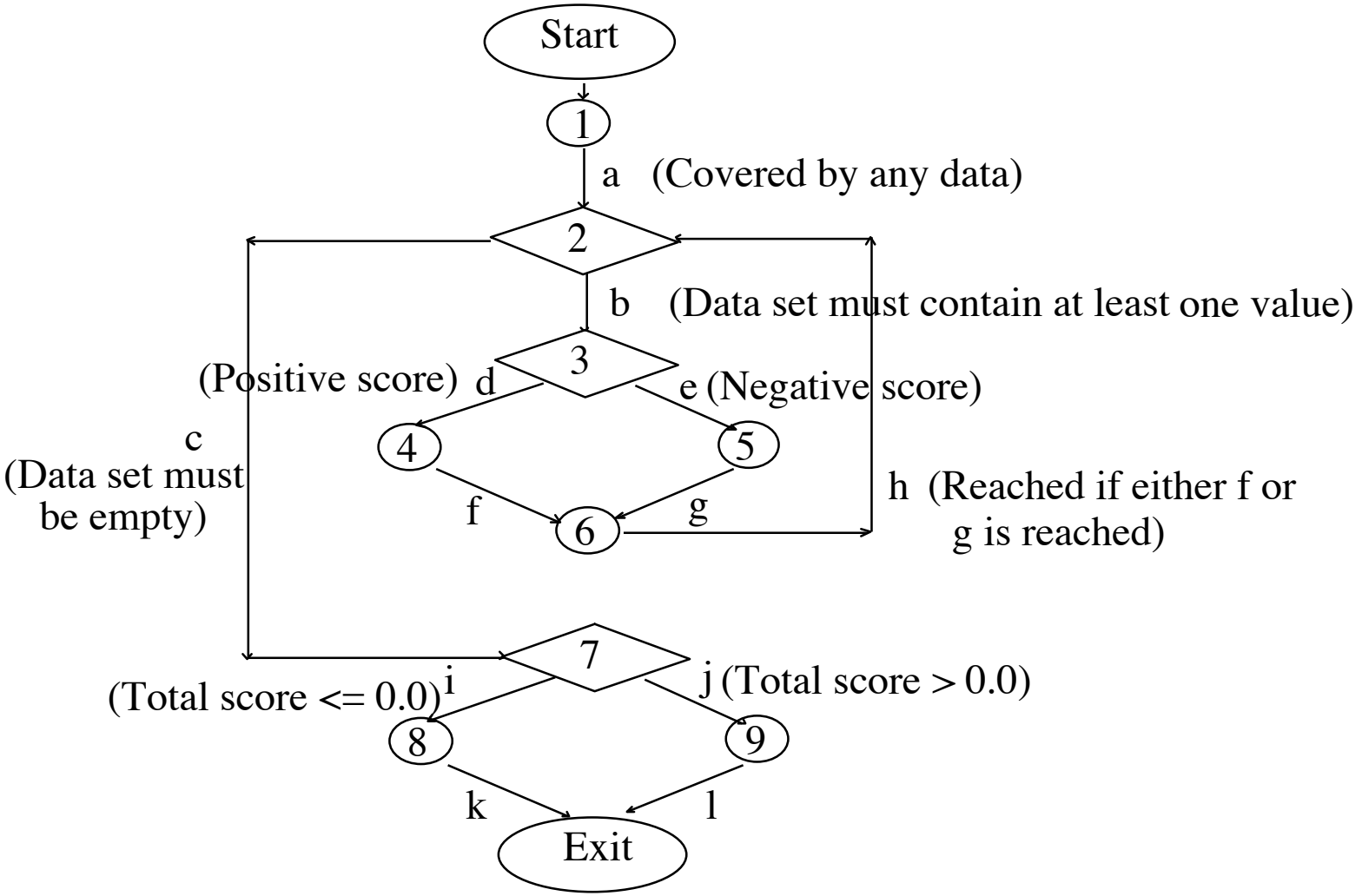


Constructing the Logic Flow Diagram

```
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");
}
```



Finding the Test Cases

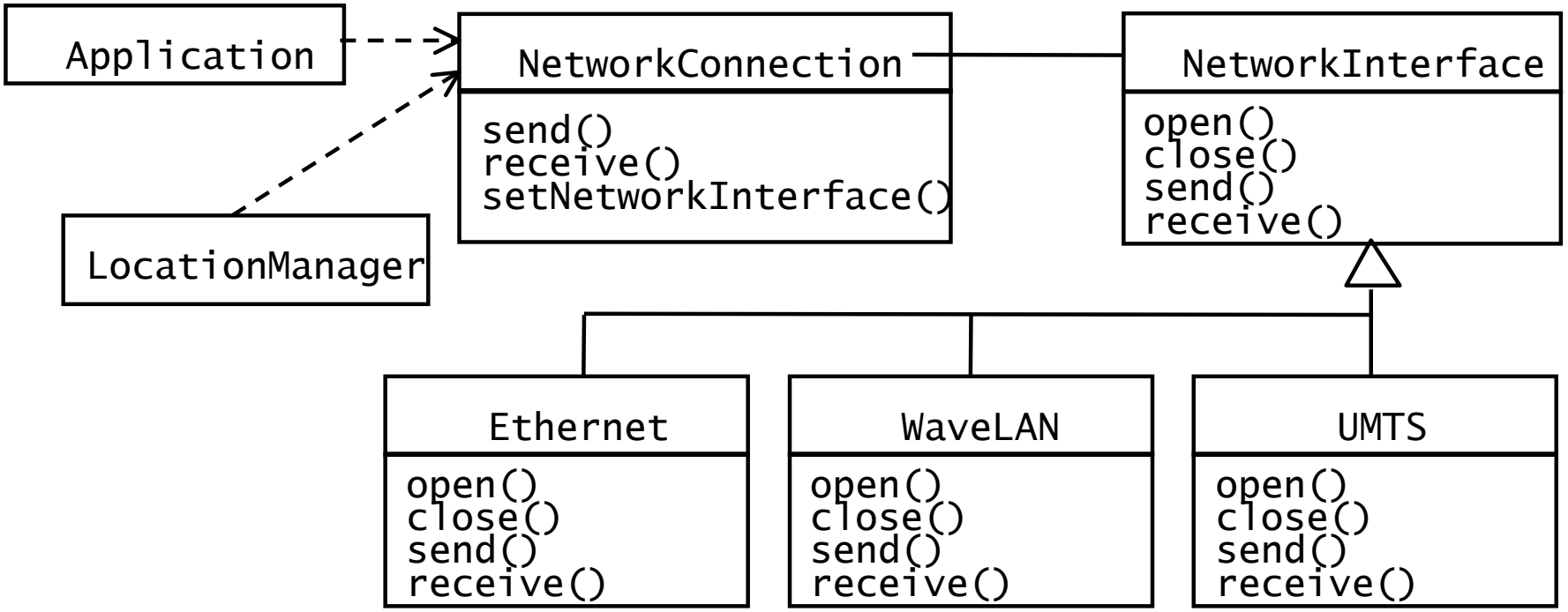


Test Cases

- Test case 1:
  - file with one value (To execute loop exactly once)
- Test case 2:
  - empty file (To skip loop body)
- Test case 3:
  - file with two values (To execute loop more than once)

These 3 test cases cover all control flow paths

Polymorphism testing



How do we test the method `NetworkInterface.send()` ?

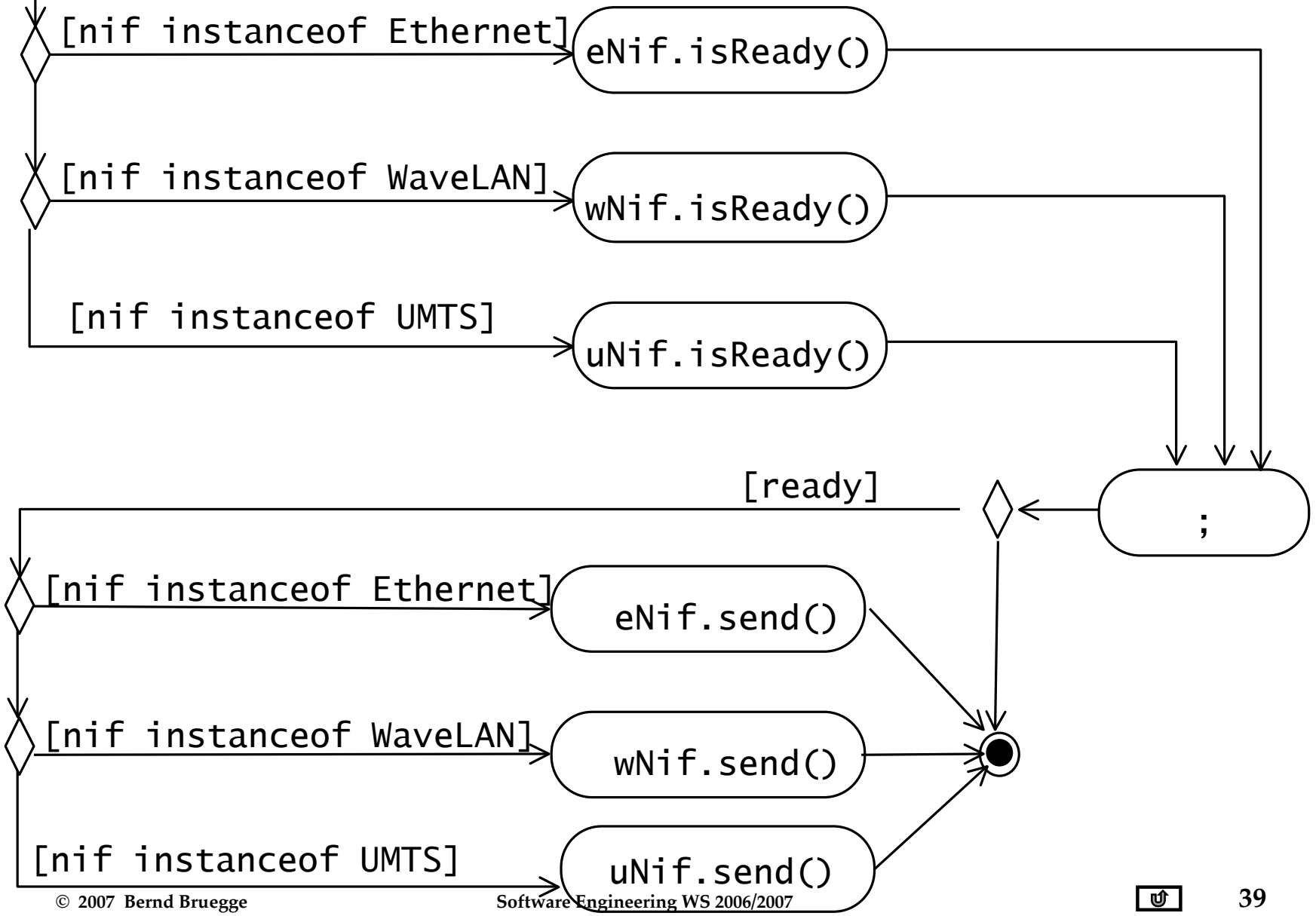
Implementation of NetworkInterface.send()

```
public class NetworkConnection {
//...
private NetworkInterface nif;
void send(byte msg[]) {
   queue.concat(msg);
   if (nif.isReady()) {
        nif.send(queue);
        queue.setLength(0);
   }
}
}
```

Polymorphism Testing of NetworkInterface.send()

```
public class NetworkConnection {
//...
private NetworkInterface nif;
void send(byte msg[]) {
   queue.concat(msg);
        boolean ready = false;
        if (nif instanceof Ethernet) {
                Ethernet eNif =
(Ethernet)nif;
                ready = eNif.isReady();
        } else if (nif instanceof
WaveLAN) {
                WaveLAN wNif =
(WaveLAN)nif;
                ready = wNif.isReady();
        } else if (nif instanceof UMTS)
{
                UMTS uNif = (UMTS)nif;
                ready = uNif.isReady();
        }
        if (ready) {
//
```

Polymorphism Testing: Activity Diagram



Comparison of White & Black-box Testing

- White-box Testing:
  - Potentially large number of paths have to be tested
  - White-box tests test 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 tested
   - Analysis: Completeness of requirements
   - Design: Cohesion
   - Implementation: Source code
2. Decide how the testing is done
   - Review or code inspection
   - Proofs (Design by Contract)
   - 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 (class, subsystem, system) being tested or about the attribute being measured
4. Create the test oracle
   - An oracle contains 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 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 and datastructures:
  - Force a division by zero
  - If the upper bound of an array is 10, then use 11 as index.

Unit Testing Heuristics

1. Create unit tests when object design is completed
   - Black-box test: Test the functional model
   - White-box test: Test the dynamic model
   - Data-structure test: Test the object model
2. Develop the test cases
   - Goal: Find minimal number of test cases to cover all paths
3. Cross-check the test cases to eliminate duplicates
   - Don't waste your time!
4. Desk check your source code
   - Sometimes 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
   - Re-execute test whenever a change is made ("regression testing")
8. Compare the results of the test with the test oracle
   - Automate this if possible.

Next Lecture

- Junit test framework
- Integration testing
- System testing