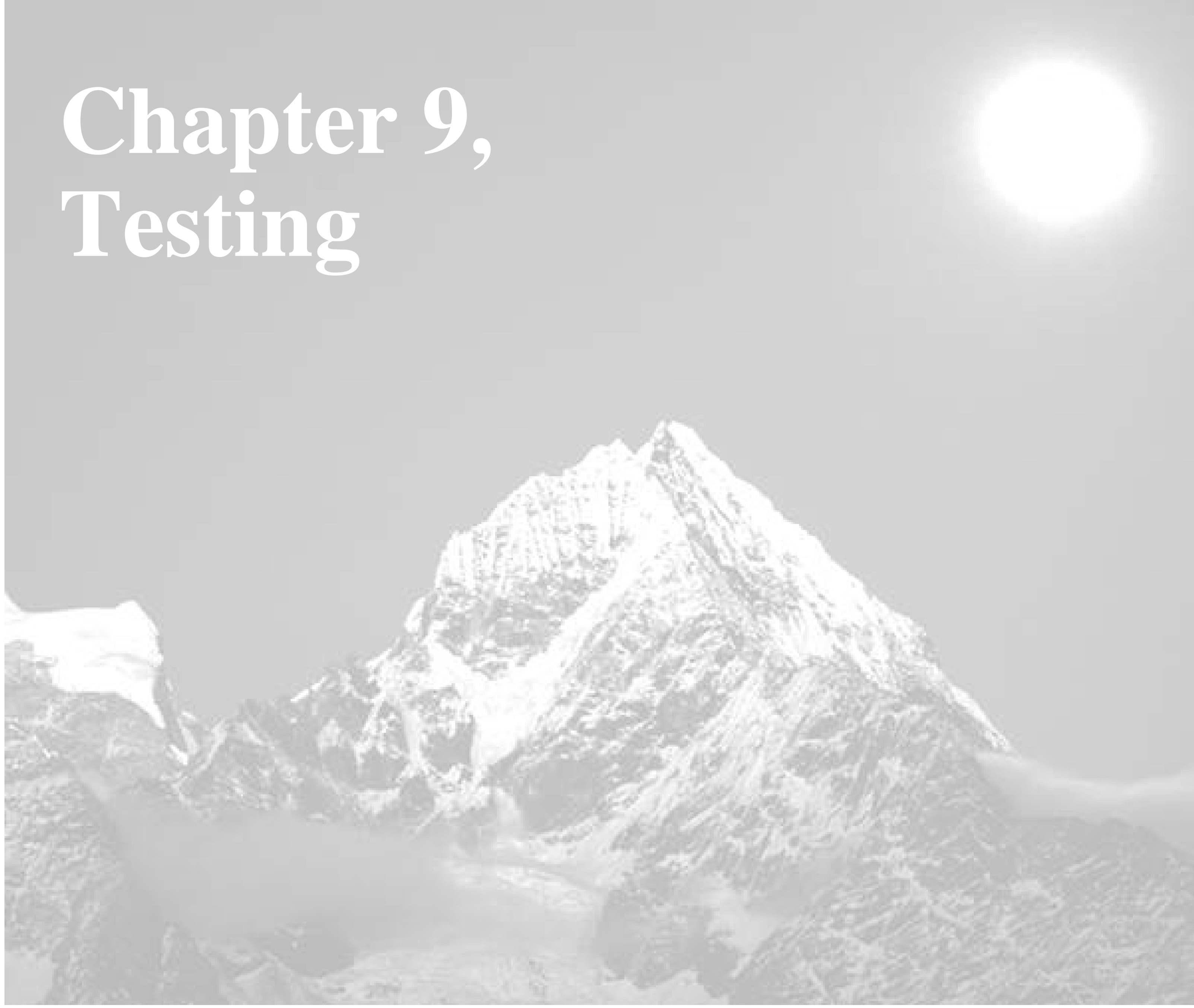
Object-Oriented Software Engineering
Conquering Complex and Changing Systems



Chapter 9, Testing

Odds and Ends: Abstract Factory & Builder Patterns revisited

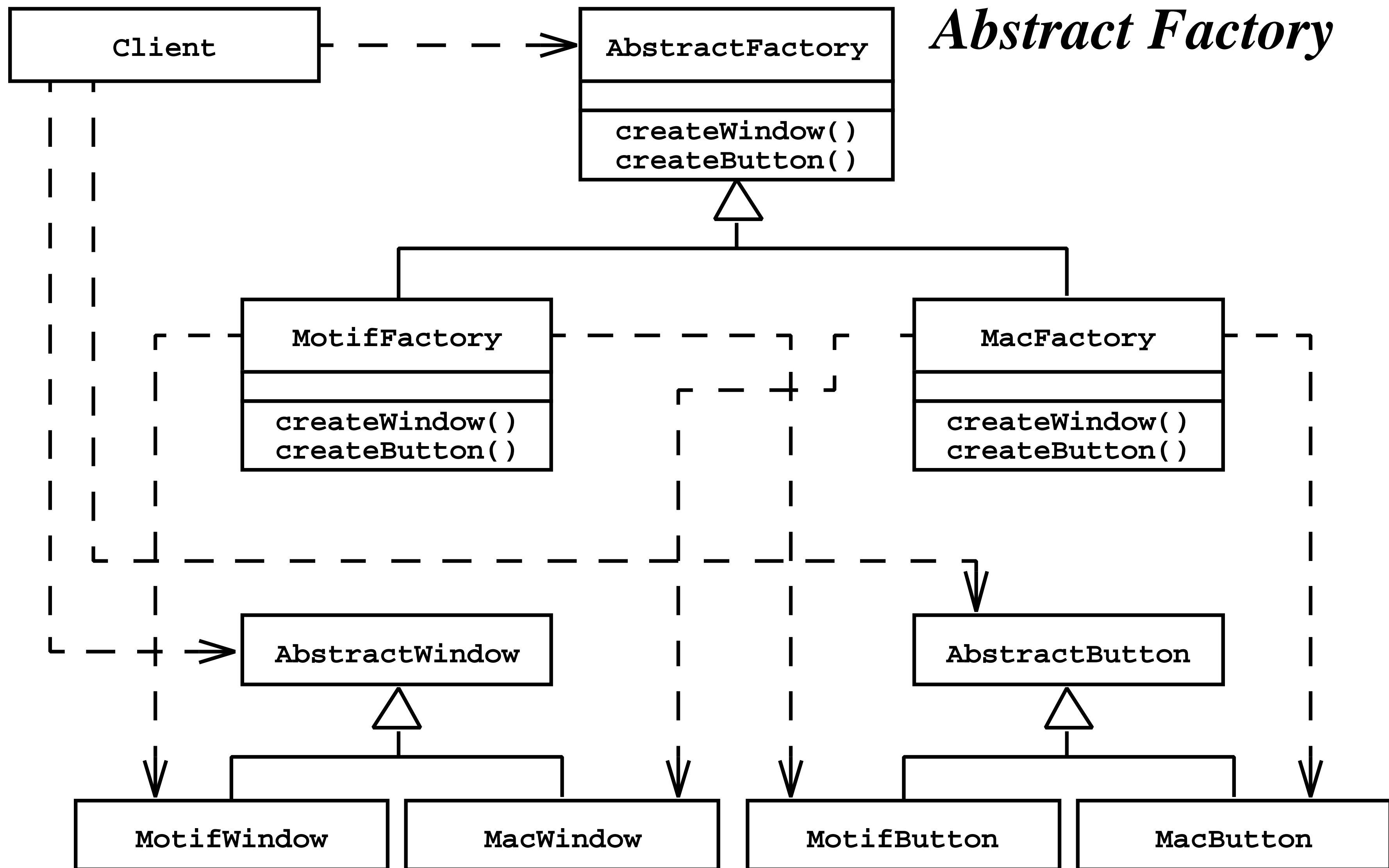
Abstract Factory and Builder are creational patterns that enable a client to build complex objects.

Abstract Factory:

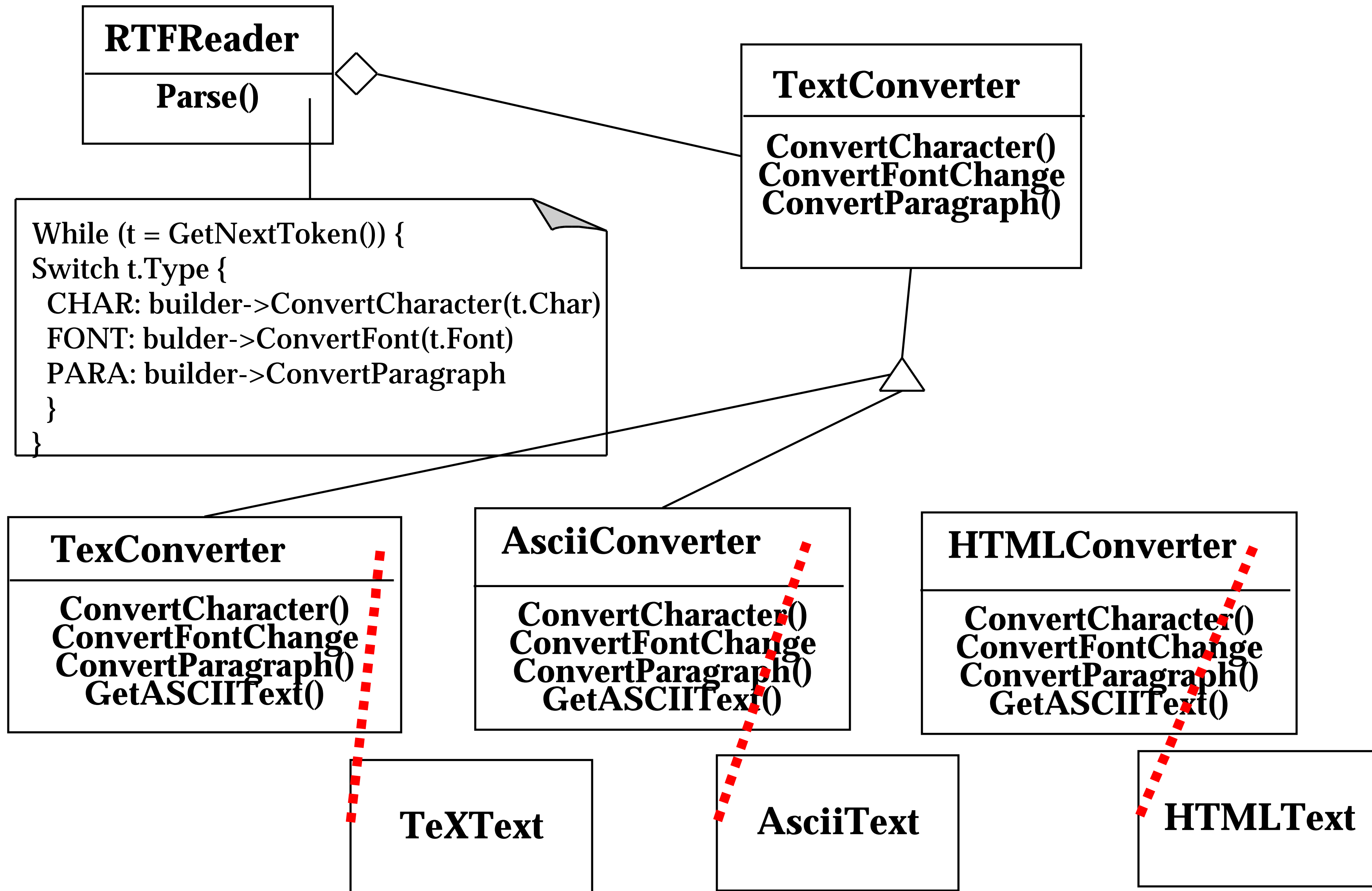
- **Provides a single interface to create families of dependent products**
- **Example: Graphical objects that are represented on different windowing platforms transparently**

Builder:

- **Separates the construction process of a complex object from its representation**
- **Example: Conversion of RTF to a variety of formats.**


Abstract Factory
Client
AbstractFactory
createWindow()
createButton()
MotifFactory
createWindow()
createButton()
MacFactory
createWindow()
createButton()
AbstractWindow
AbstractButton
MotifWindow
MacWindow
MotifButton
MacButton

Builder example: convert from RTF to external formats



Builder Creation Sequence

The client creates a directory and configures it for a specific builder:

```
director = new Directory(new HTMLBuilder());
director->construct()
```

The directory invokes the builder to create the parts

```
while (t = GetNextToken()) {
 switch t.Type {
  CHAR: builder->ConvertCharacter(t.char)
  FONT: bulder->ConvertFont(t.font)
  PARA: builder->ConvertParagraph(t.para)
 }
}
```

The client requests the result

- **htmlResult = director-> getResult();**

Abstract Factor vs. Builder

Abstract Factory

- **Shields the client from the representation**
- **Returns individual products immediately**

Builder

- **Separates the creation process from the representation**
- **The client is exposed to the representation**
- **Incrementally builds the product and returns the aggregate product when requested by the client**

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**

Terminology

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

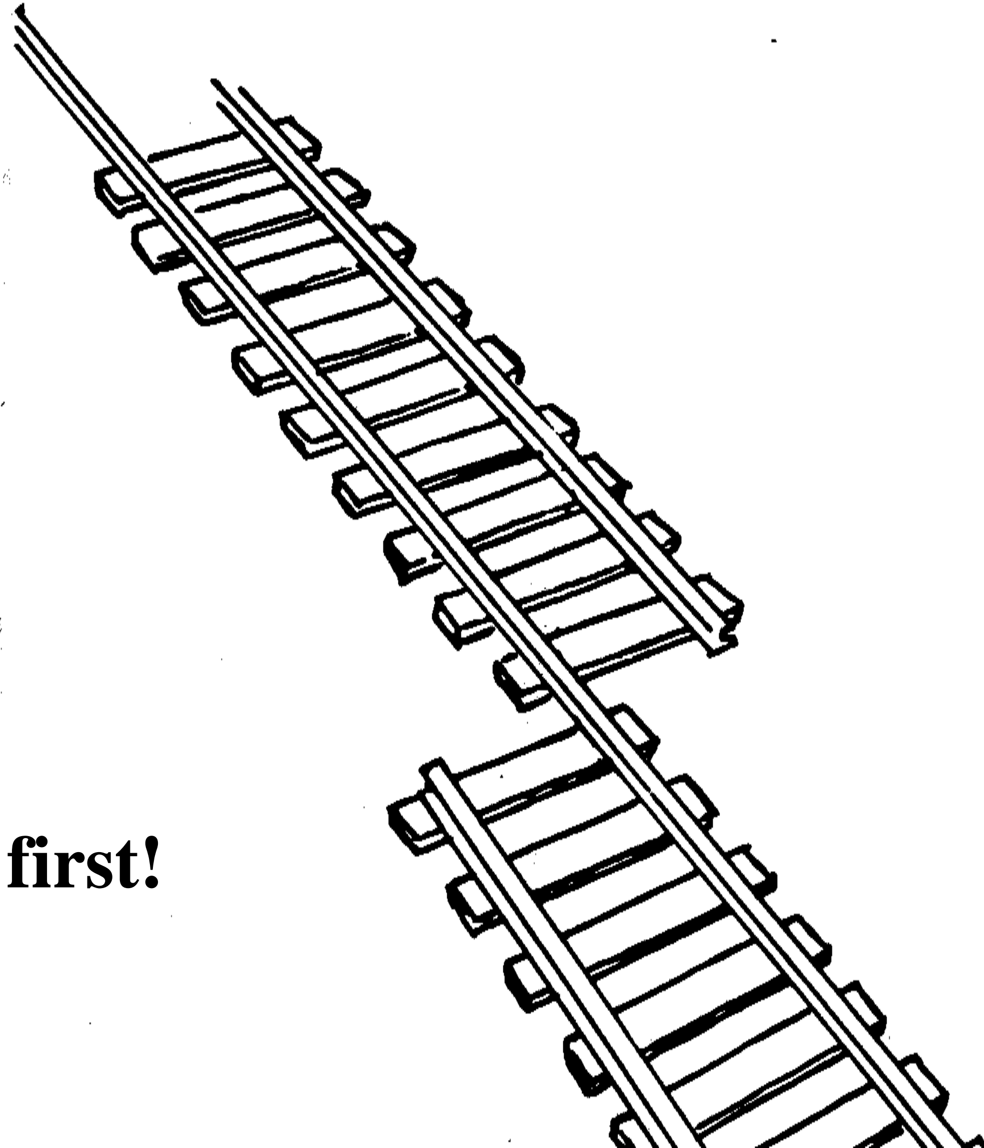
Failure: Any deviation of the observed behavior from the specified behavior.

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.

What is this?



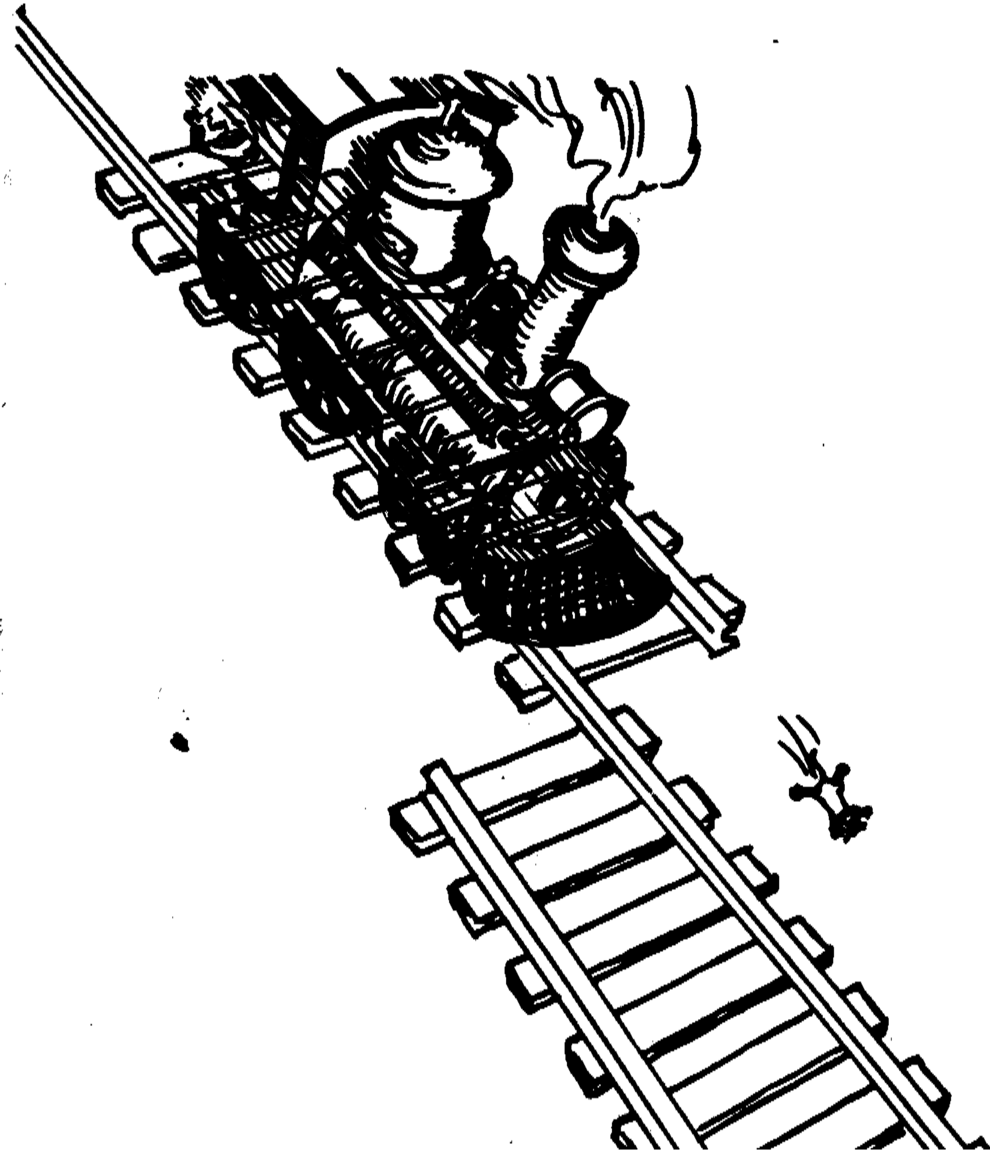
A failure?

An error?

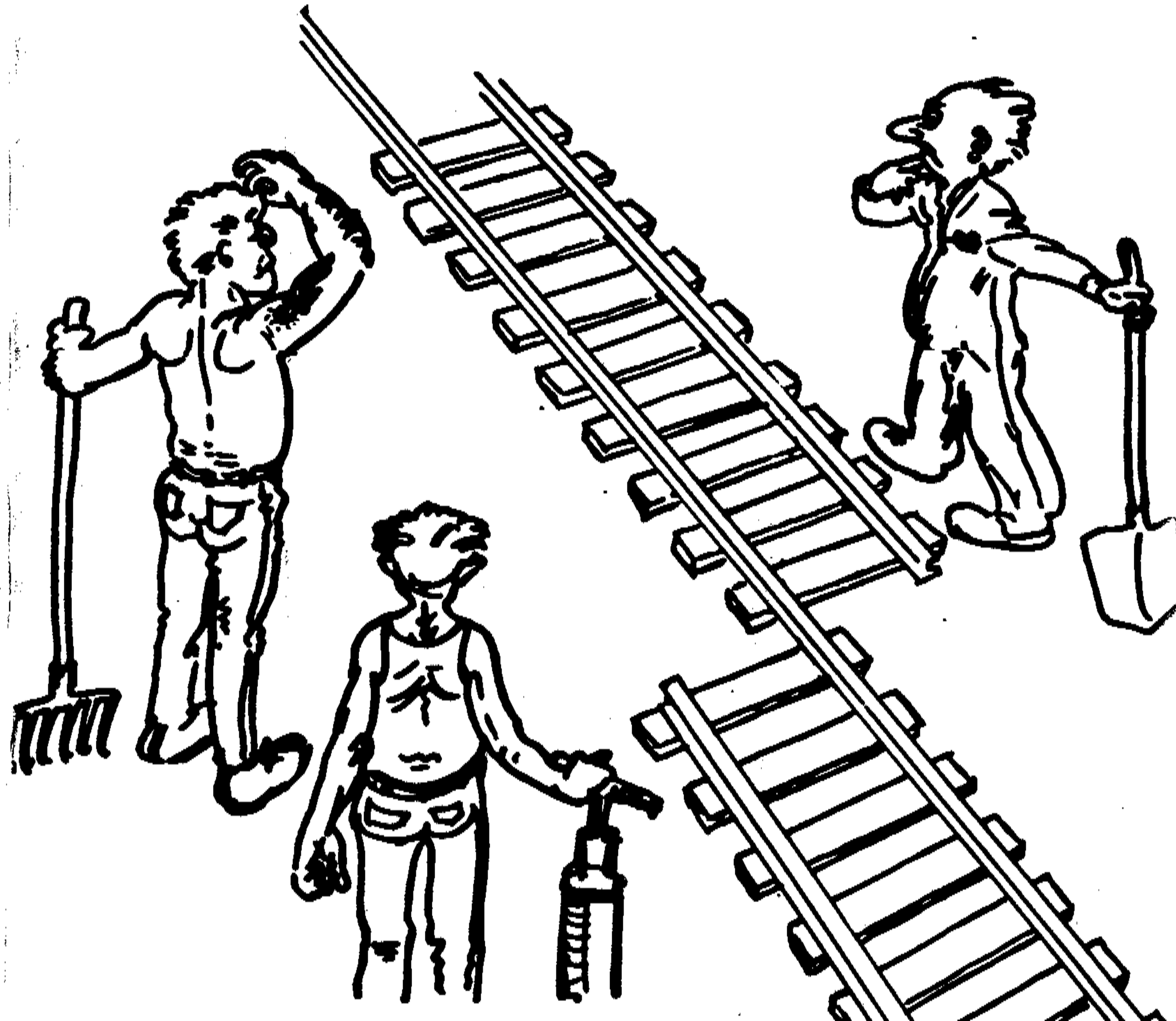
A fault?

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

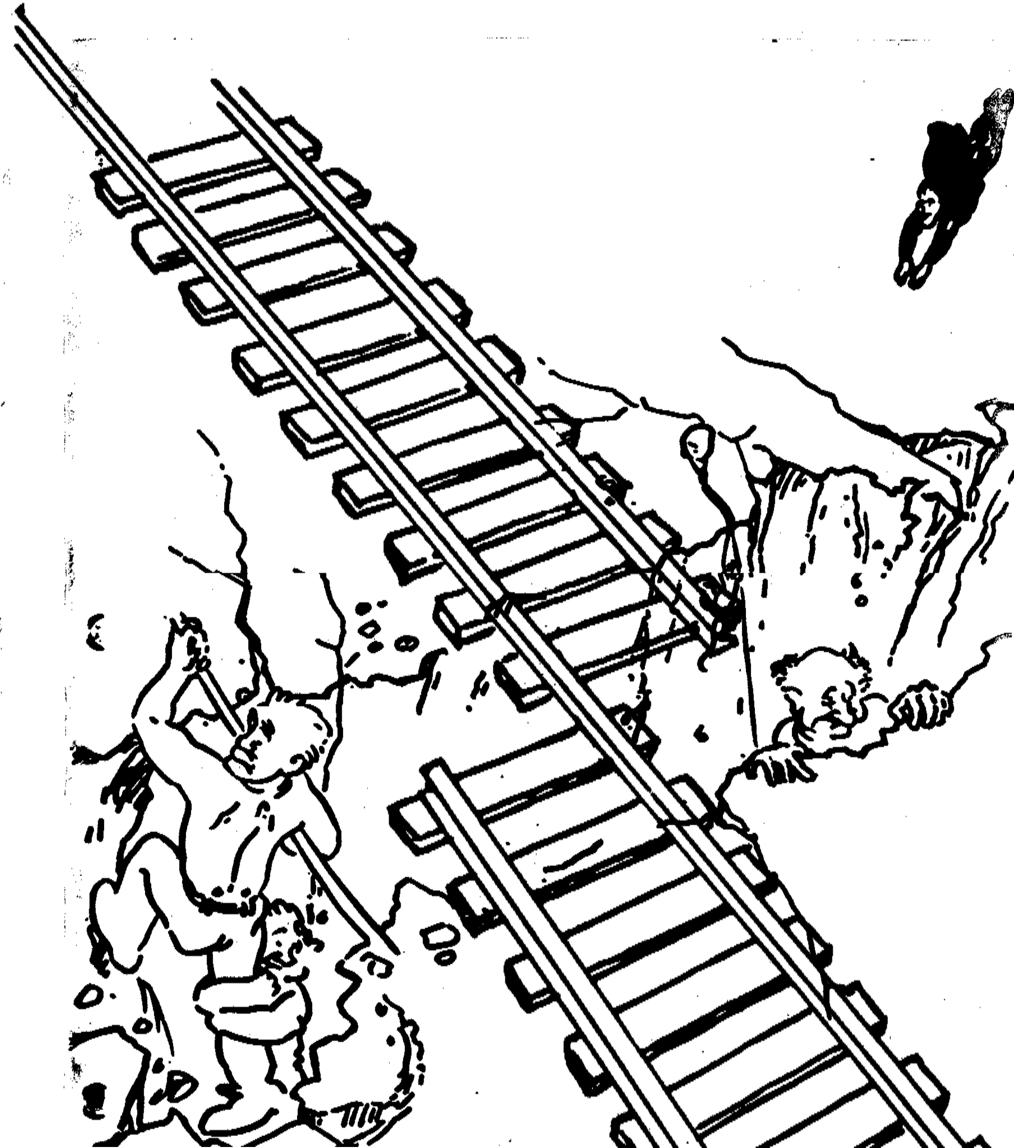
Erroneous State ("Error")



Algorithmic Fault

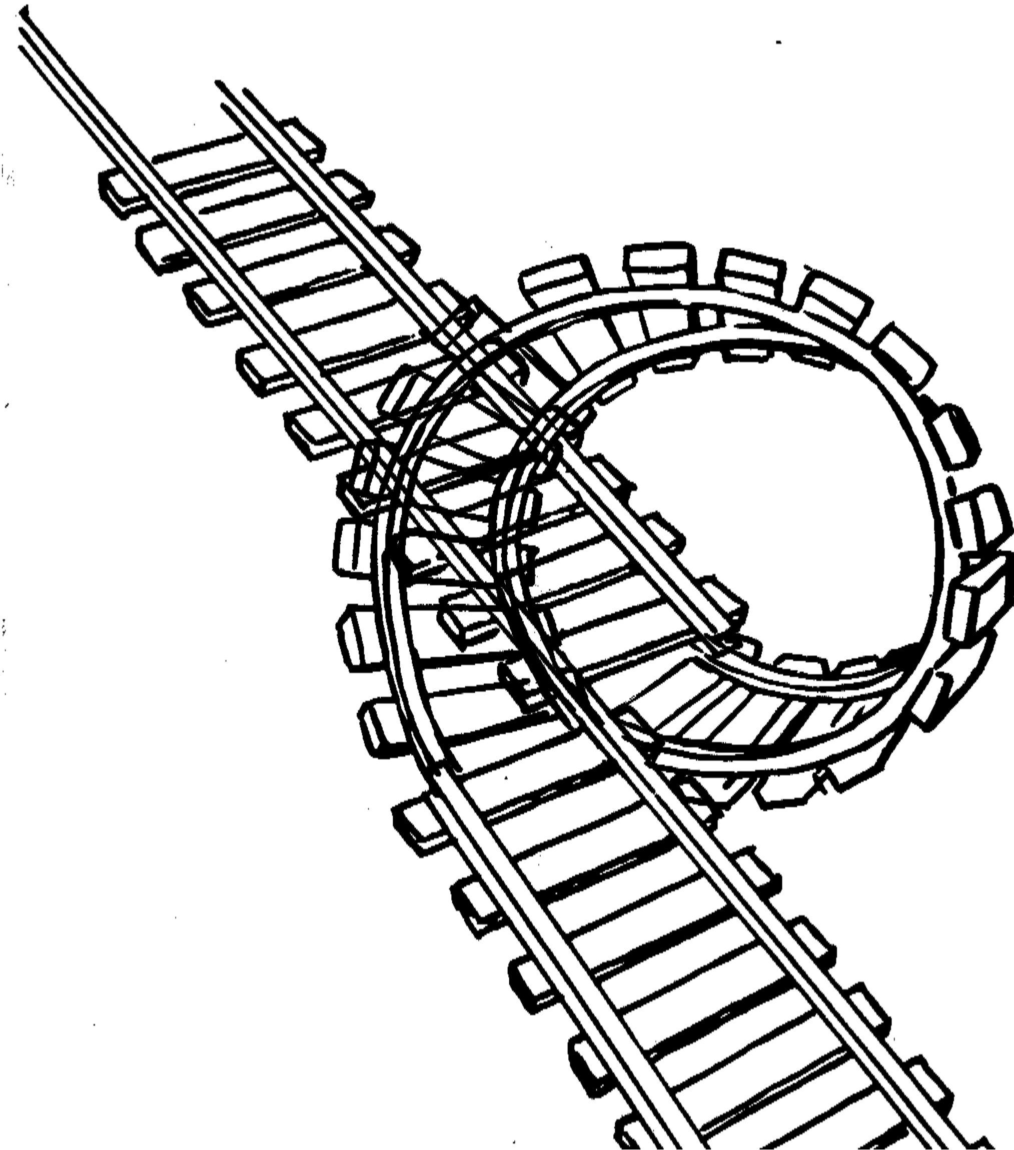


Mechanical Fault

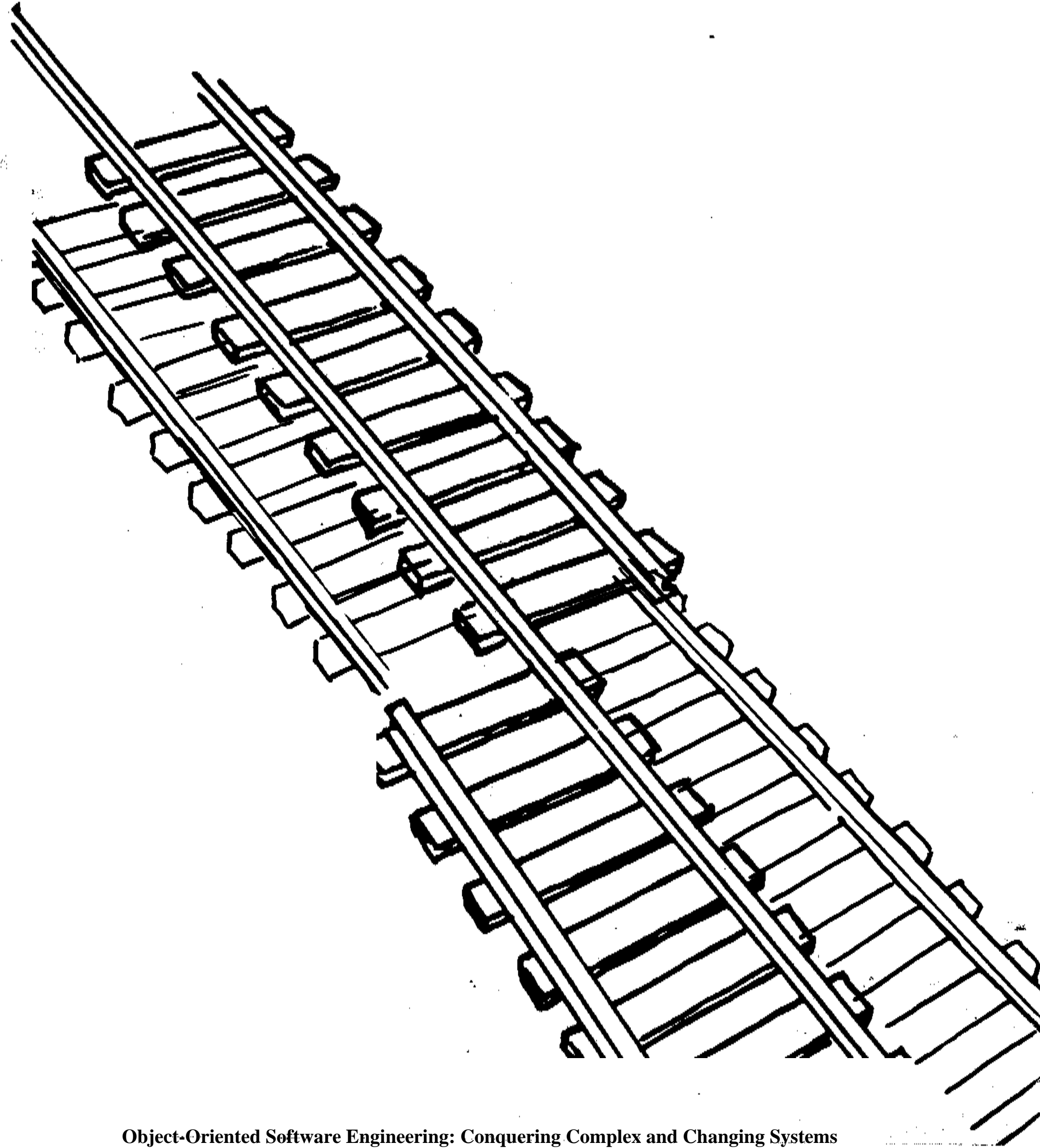


How do we deal with Errors and Faults?

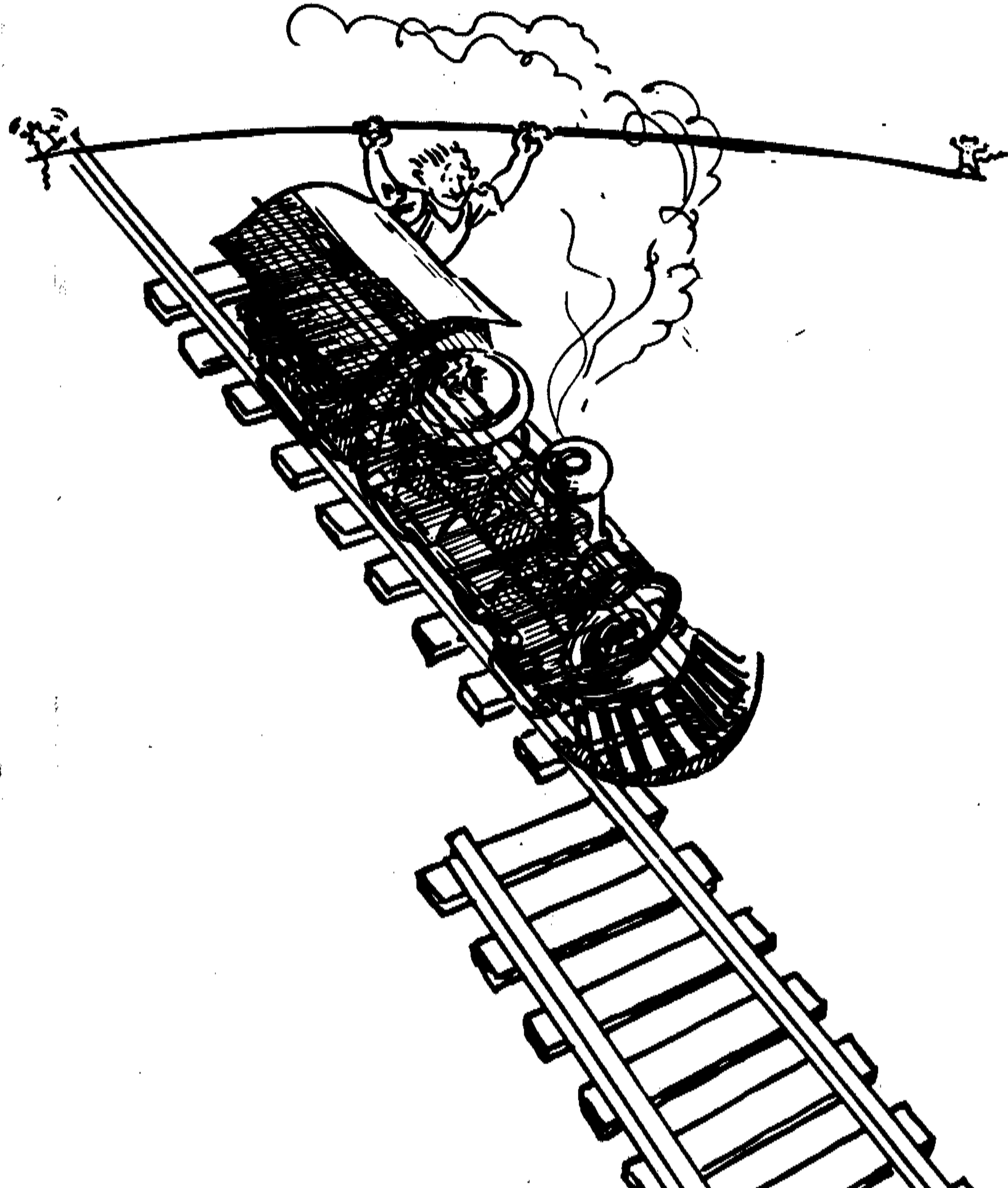
Verification?



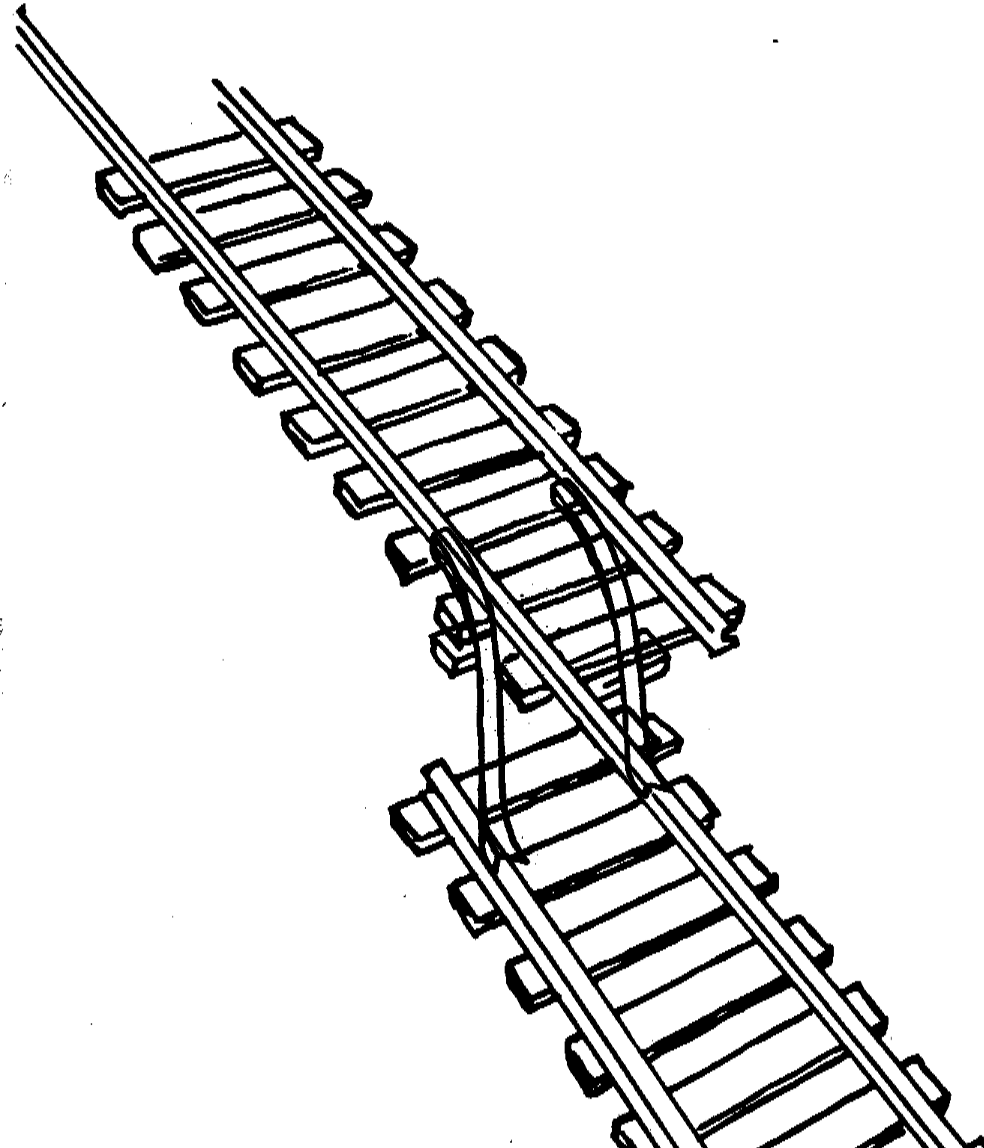
Modular Redundancy?



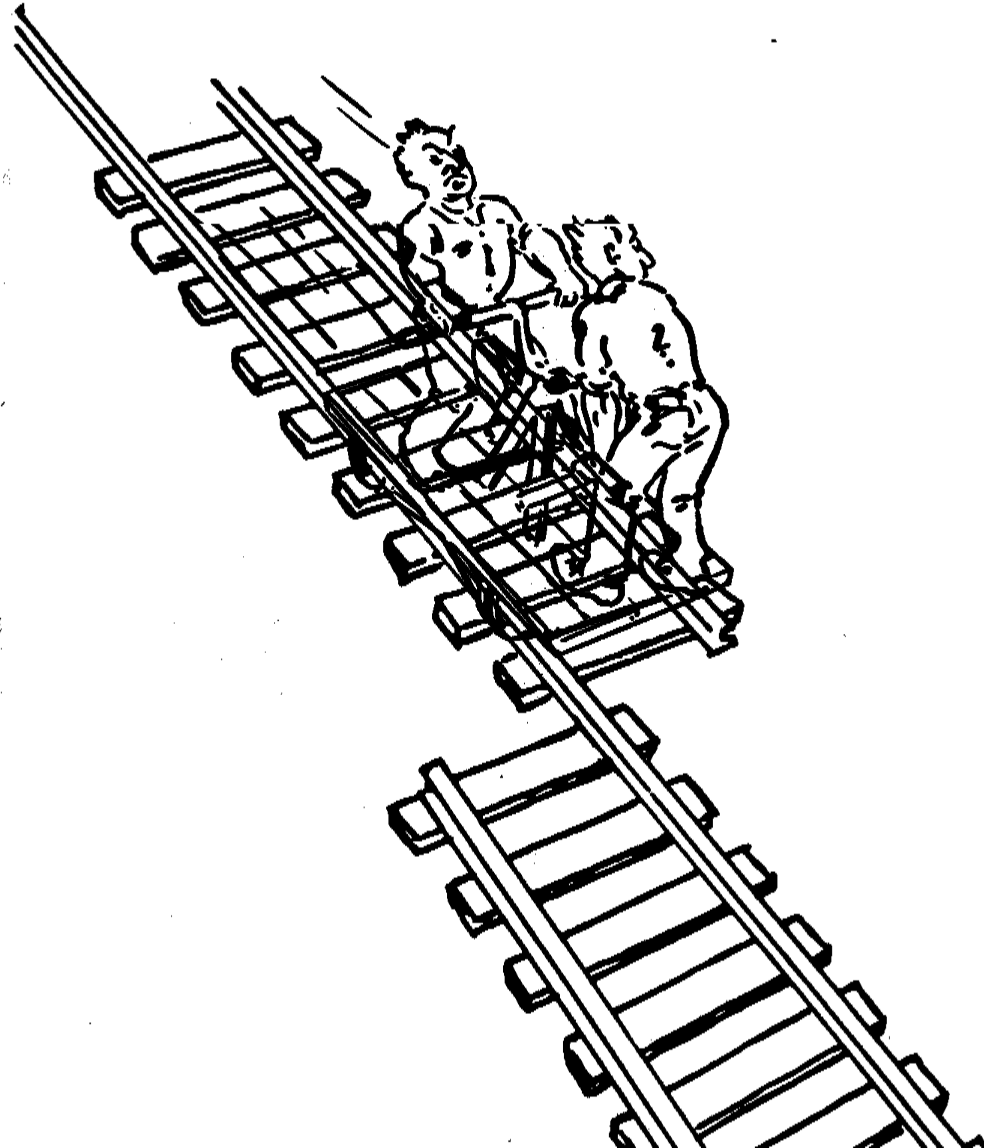
Declaring the Bug as a Feature?



Patching?



Testing?



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**

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)

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.**

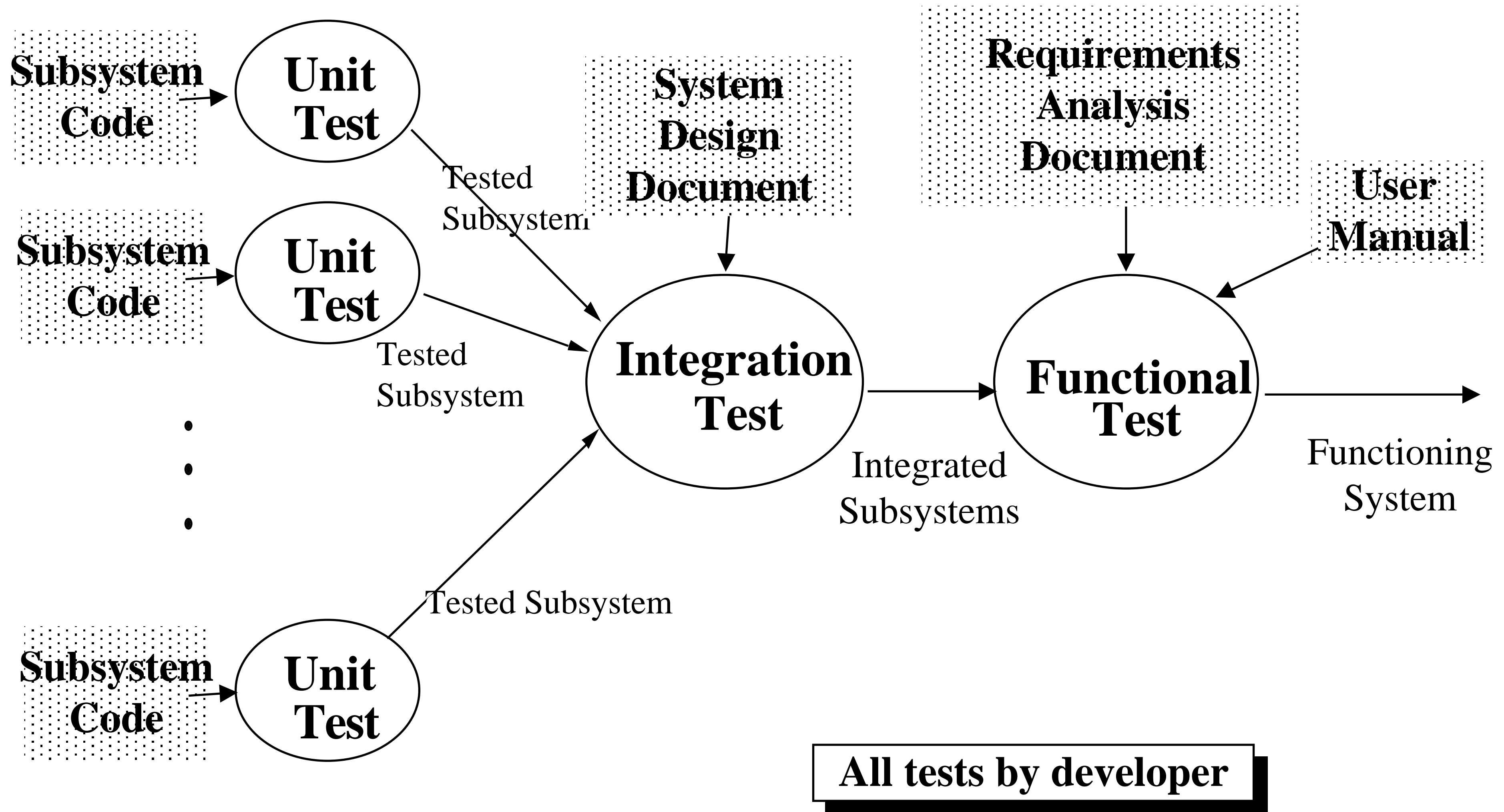
Programmer often stick to the data set that makes the program work

- **"Don’t mess up my code!"**

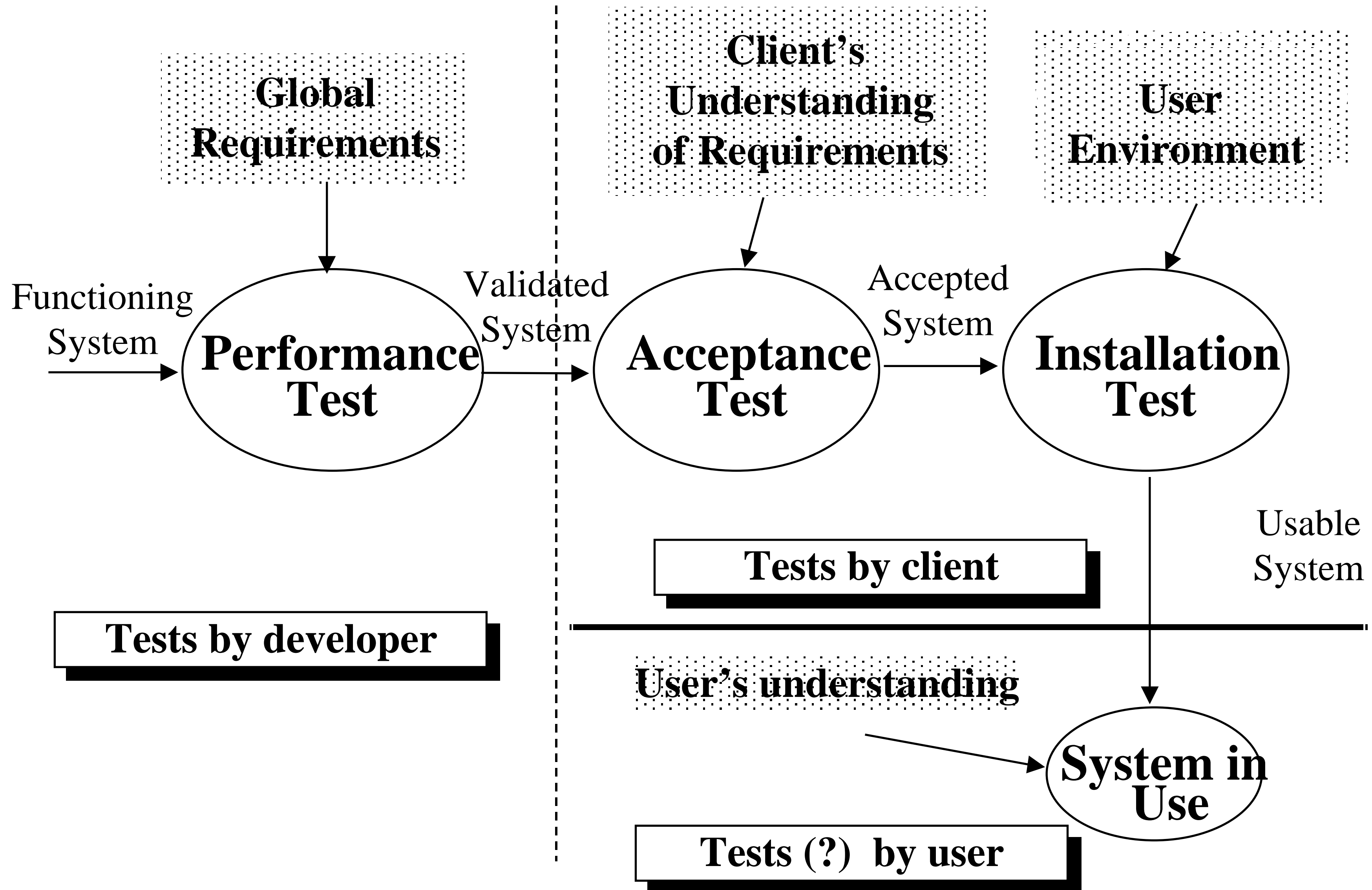
A program often does not work when tried by somebody else.

- **Don't let this be the end-user.**

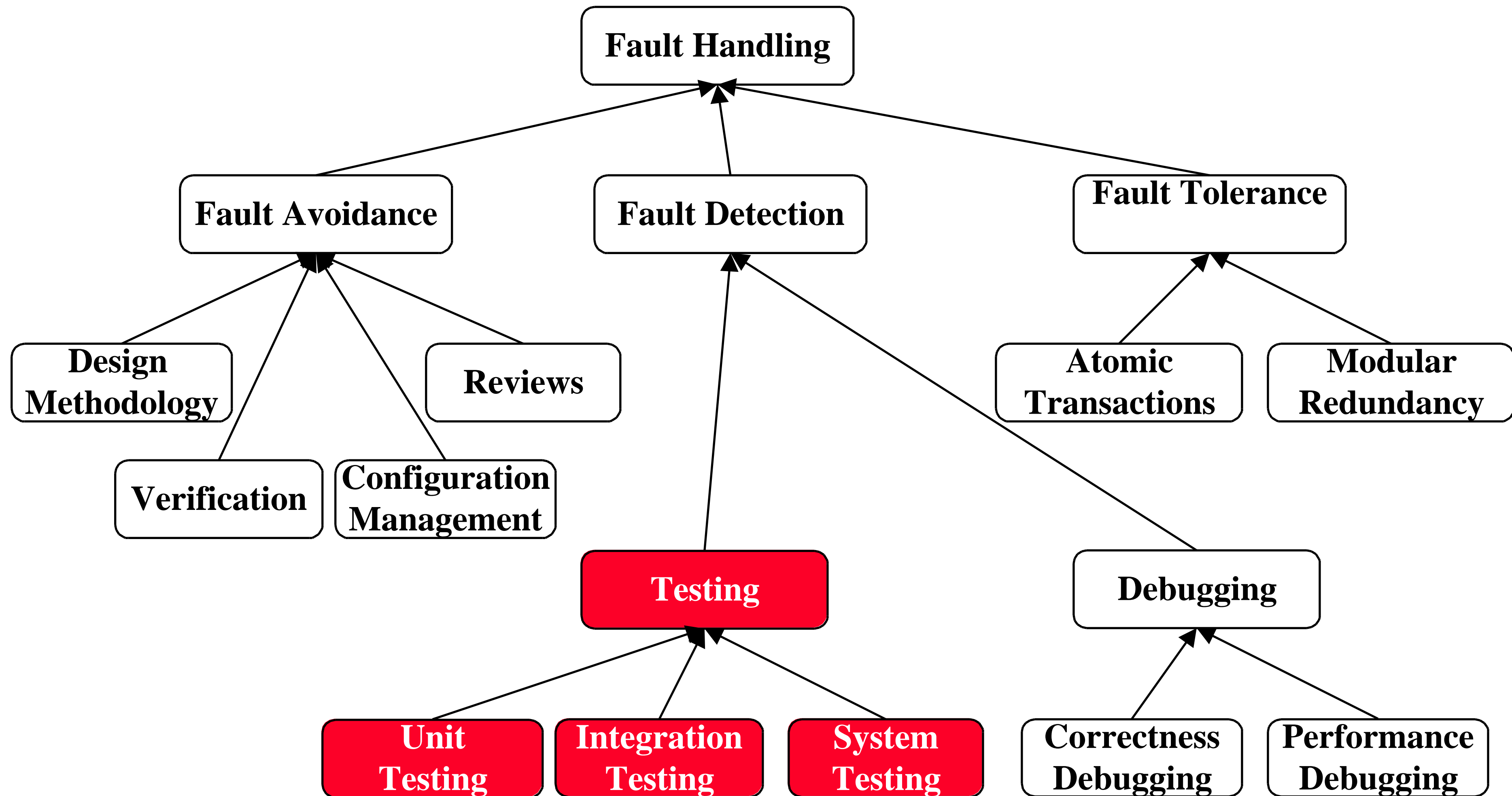
Testing Activities



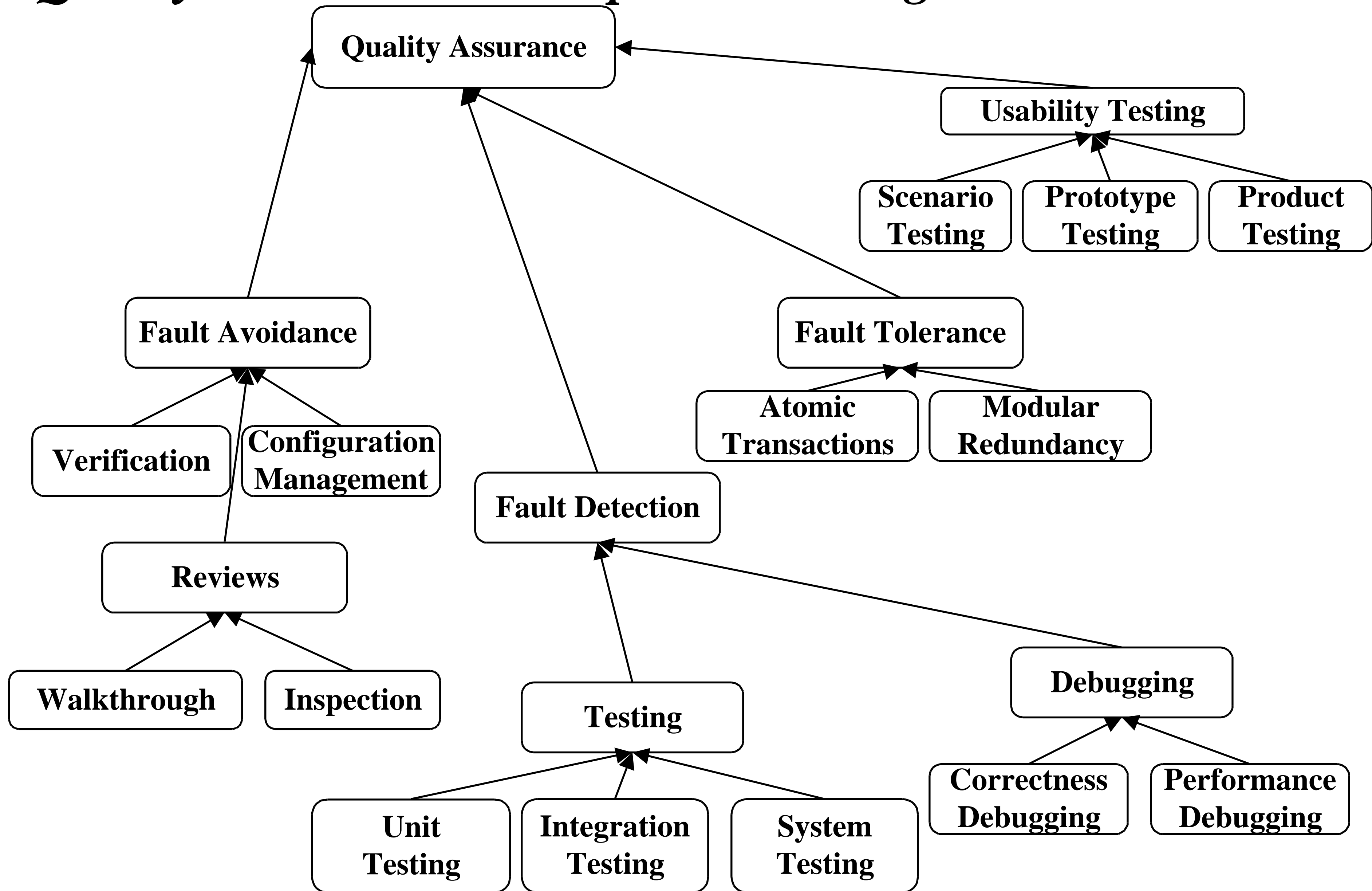
Testing Activities continued



Fault Handling Techniques



Quality Assurance encompasses Testing



Types of Testing

Unit Testing:

- **Individual subsystem**
- **Carried out by developers**
- **<u>Goal:</u> 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**
- **<u>Goal:</u> 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**

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)**

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**

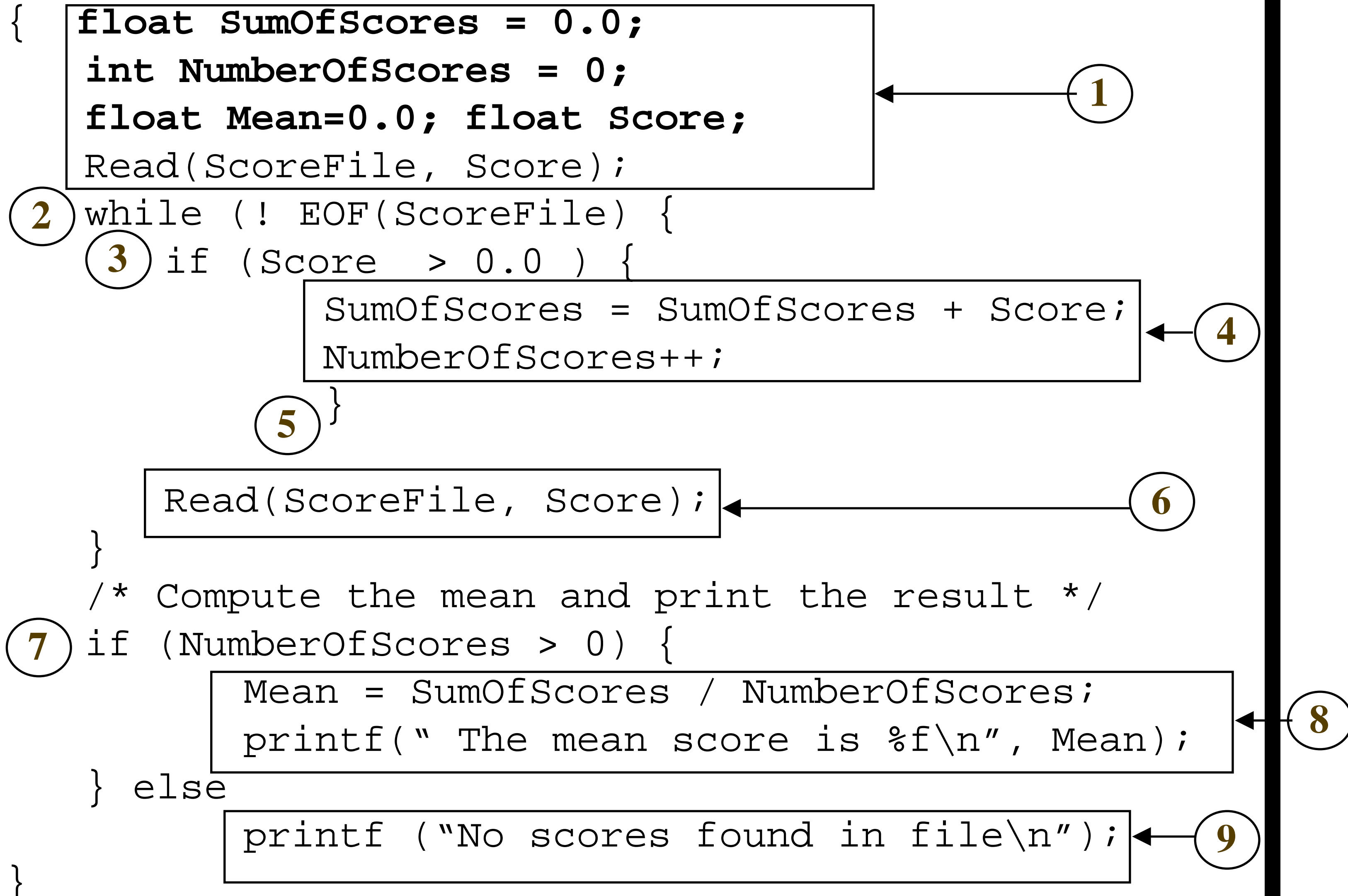
Branch Testing (Conditional Testing): Make sure that each possible outcome from a condition is tested at least once

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

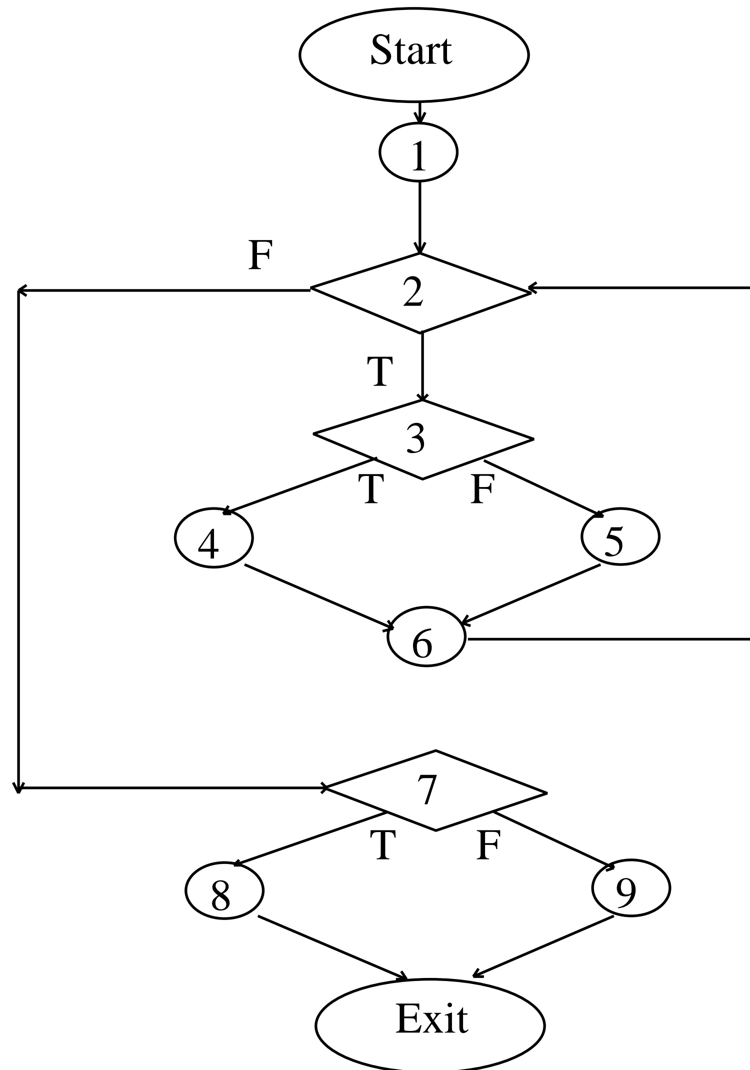
White-box Testing Example

```
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");
}
```

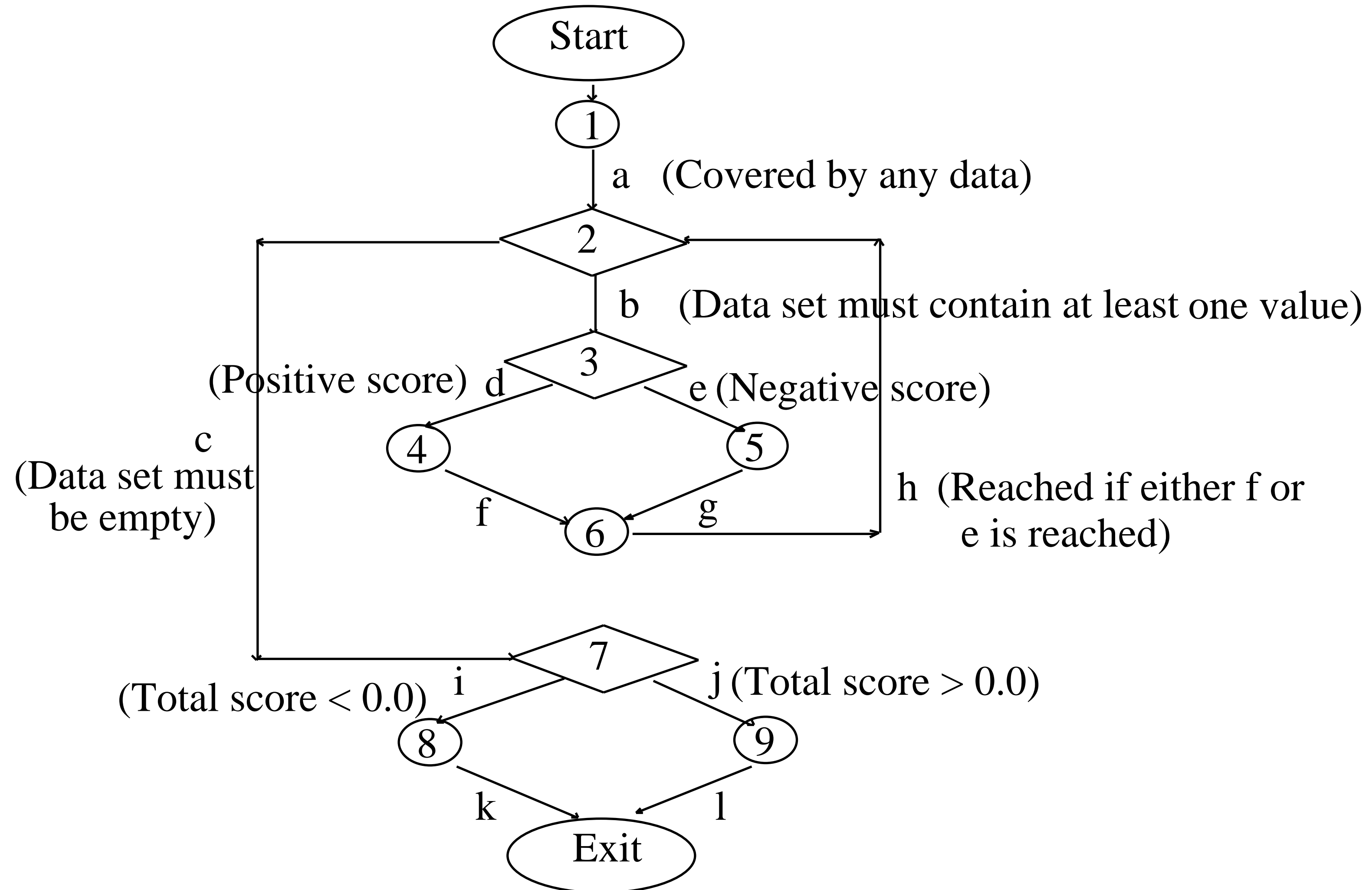
White-box Testing Example: Determining the Paths



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 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 what has to be measured
   - **Analysis: Completeness of requirements**
   - **Design: tested for cohesion**
   - **Implementation: Code tests**
2. Decide how 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)**
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 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.**
8. Compare the results of the test with the test oracle
   - **Automate as much as possible**

Integration 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

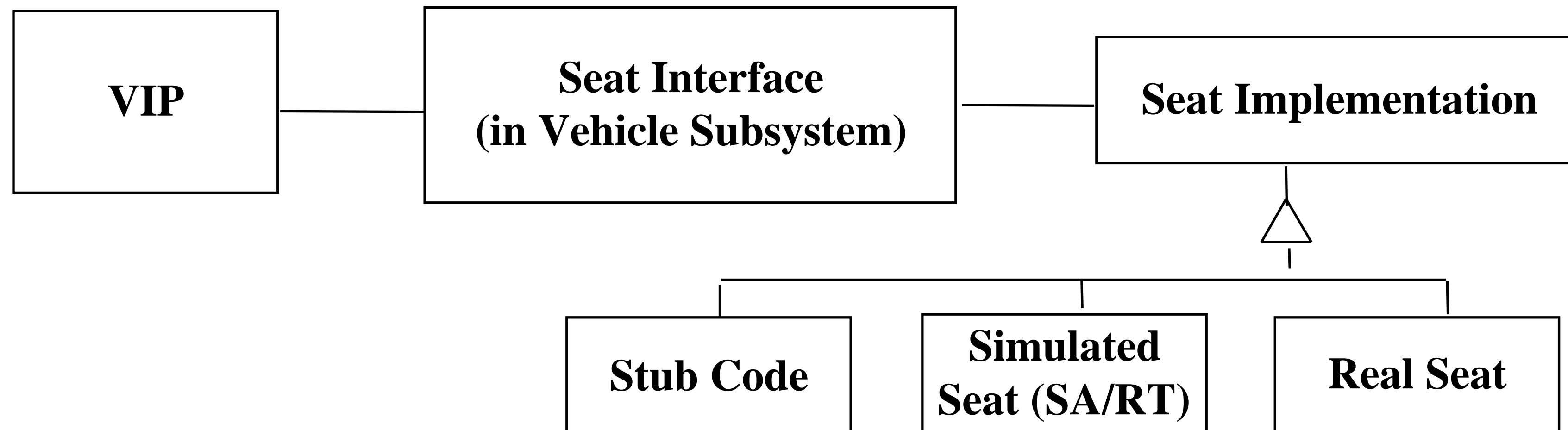
- **Big bang integration (Nonincremental)**
- **Bottom up integration**
- **Top down integration**
- **Sandwich testing**
- **Variations of the above**

For the selection use the system decomposition from the System Design

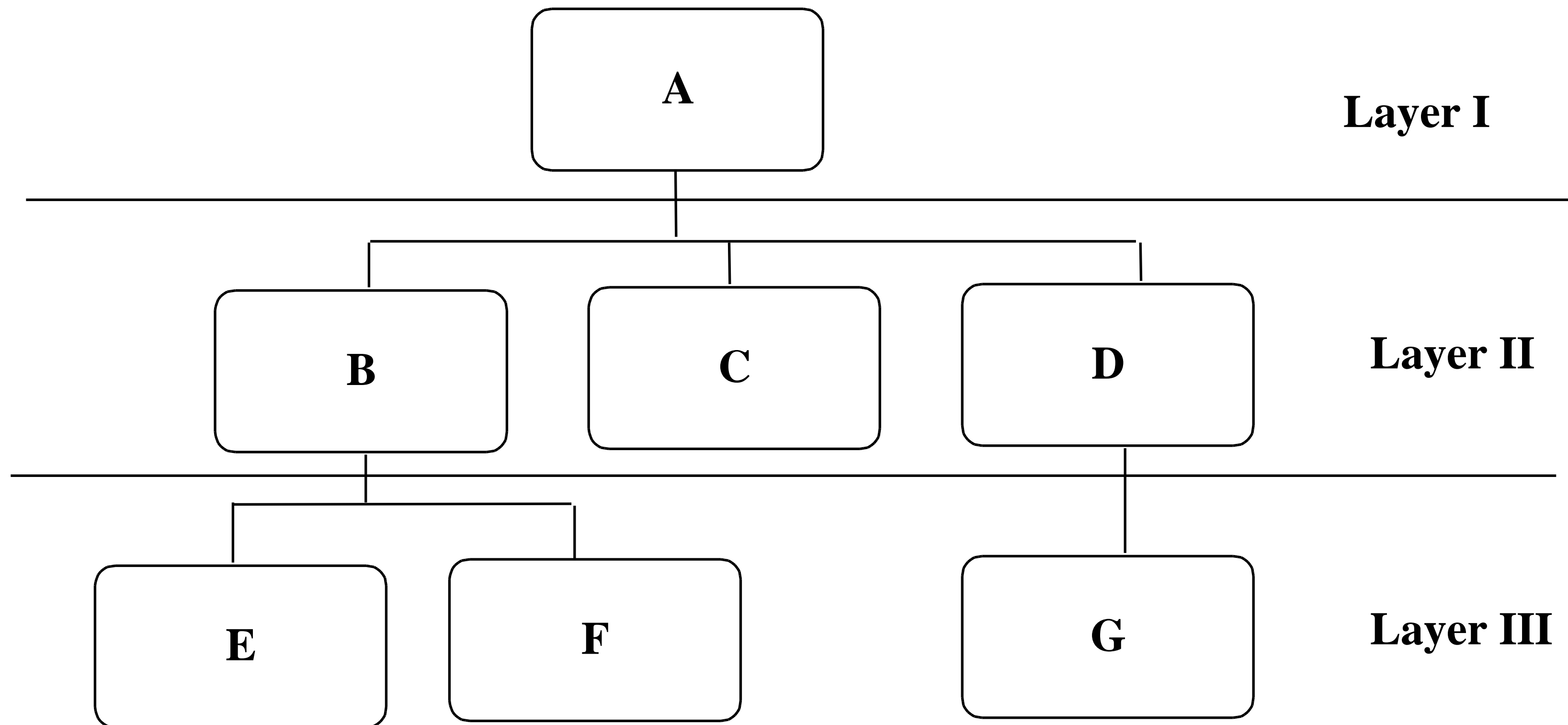
Using the Bridge Pattern to enable early Integration Testing

Use the bridge pattern to provide multiple implementations under the same interface.

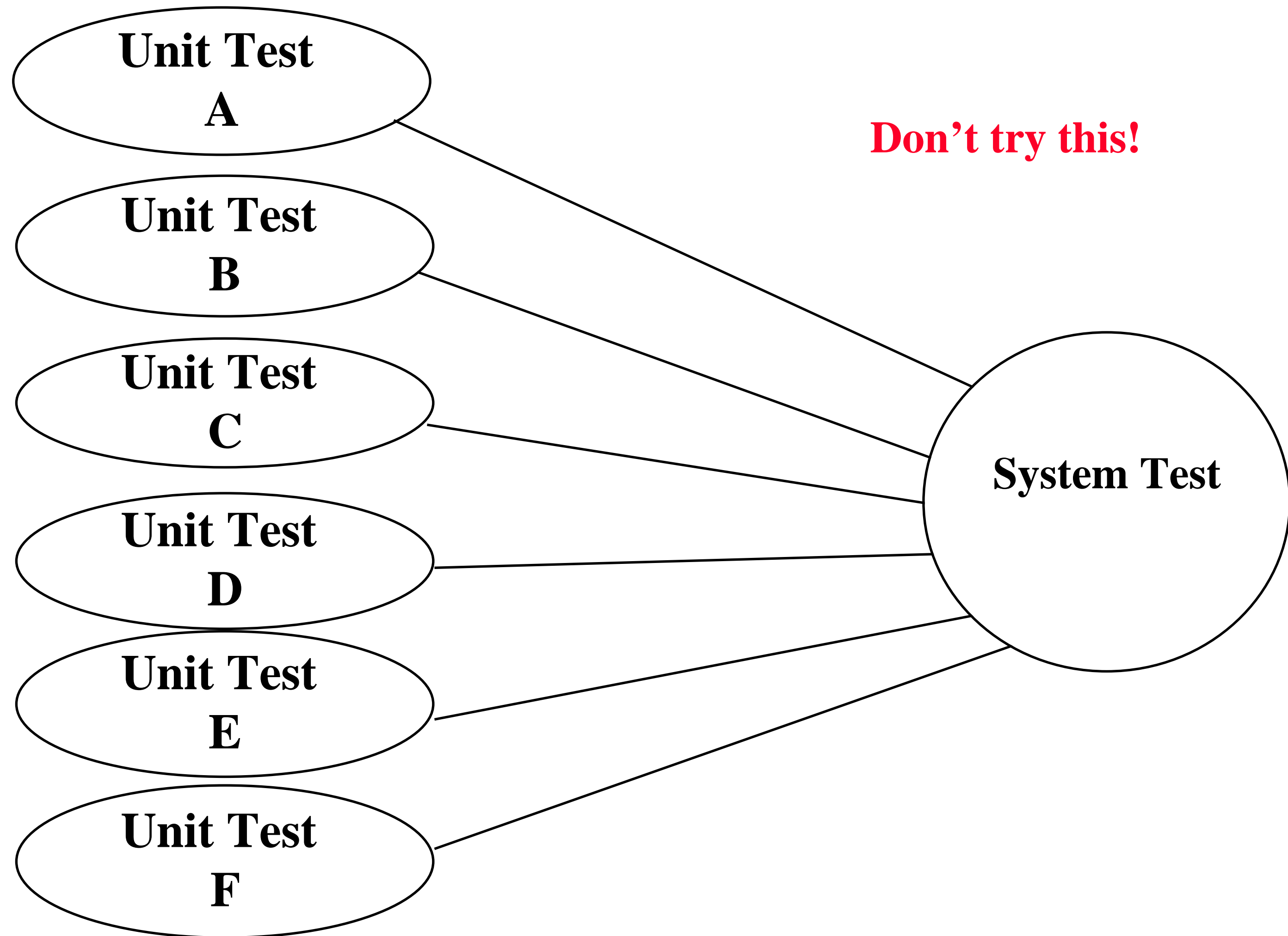
Interface to a component that is incomplete, not yet known or unavailable during testing



Example: Three Layer Call Hierarchy



Integration Testing: Big-Bang Approach



Bottom-up Testing Strategy

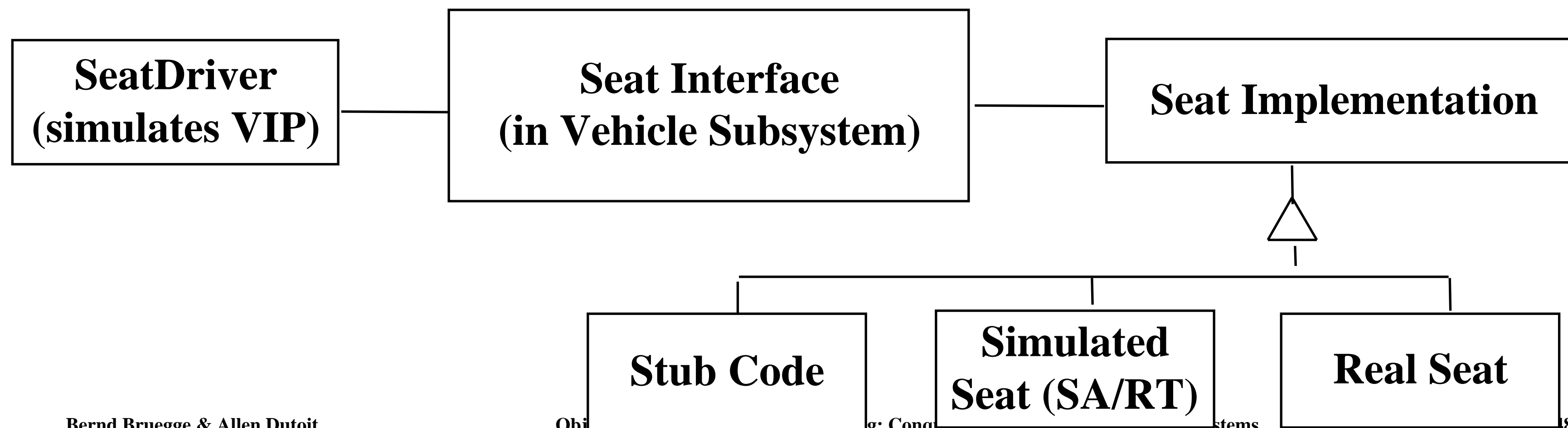
The subsystem in the lowest layer of the call hierarchy are tested individually

Then the next subsystems are tested that call the previously tested subsystems

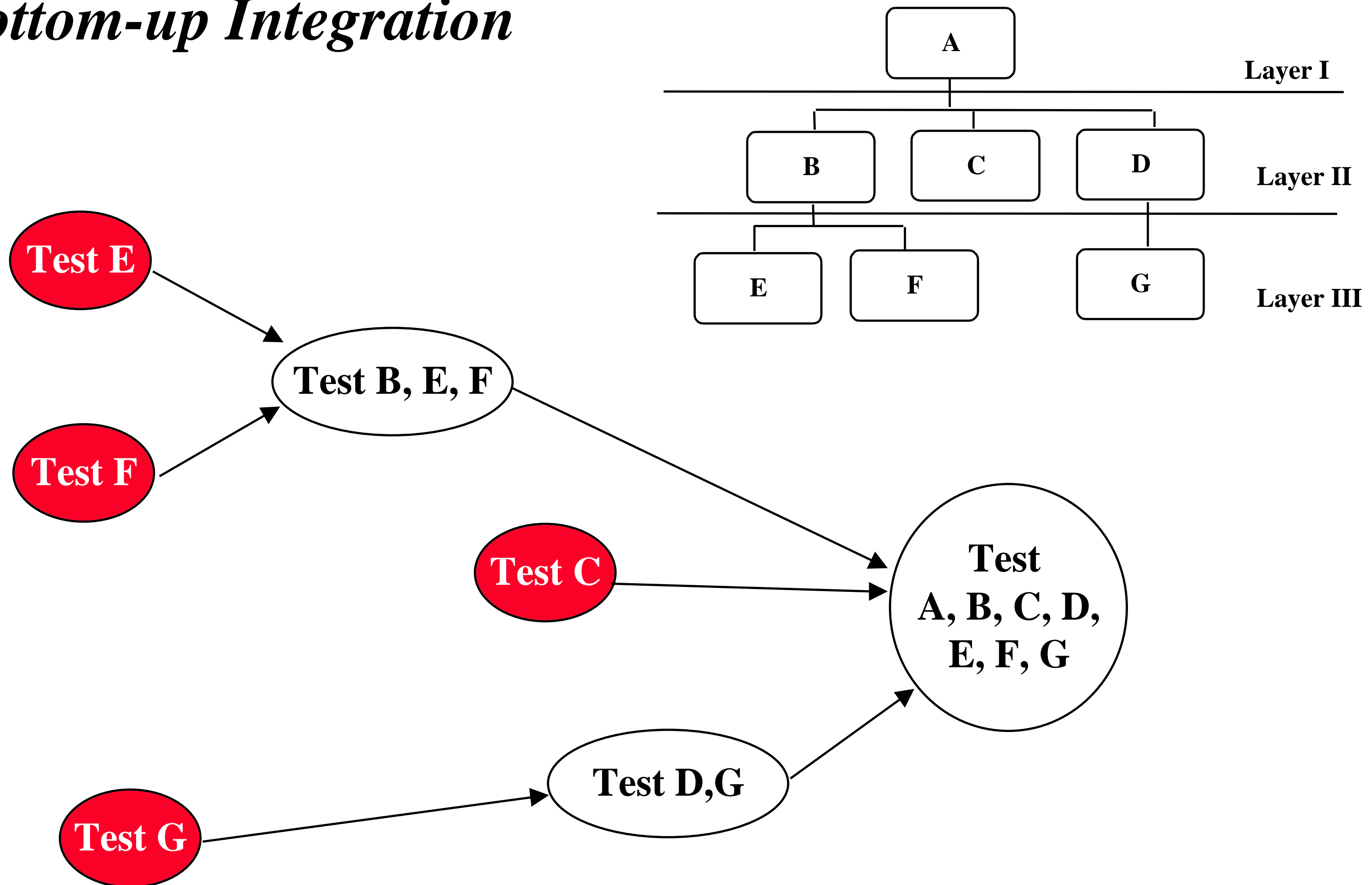
This is done repeatedly until all subsystems are included in the testing

Special program needed to do the testing, Test Driver:

- **A routine that calls a subsystem and passes a test case to it**



Bottom-up Integration



Pros and Cons of bottom up integration testing

Bad for functionally decomposed systems:

- **Tests the most important subsystem (UI) 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

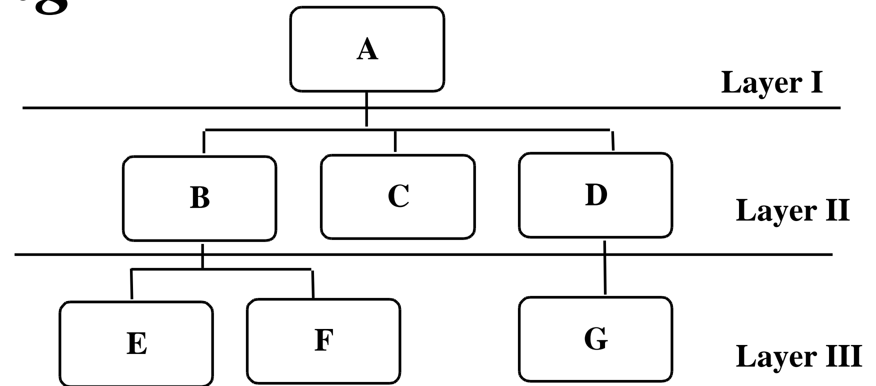
Then combine all the subsystems that are called by the tested subsystems and test the resulting collection of subsystems

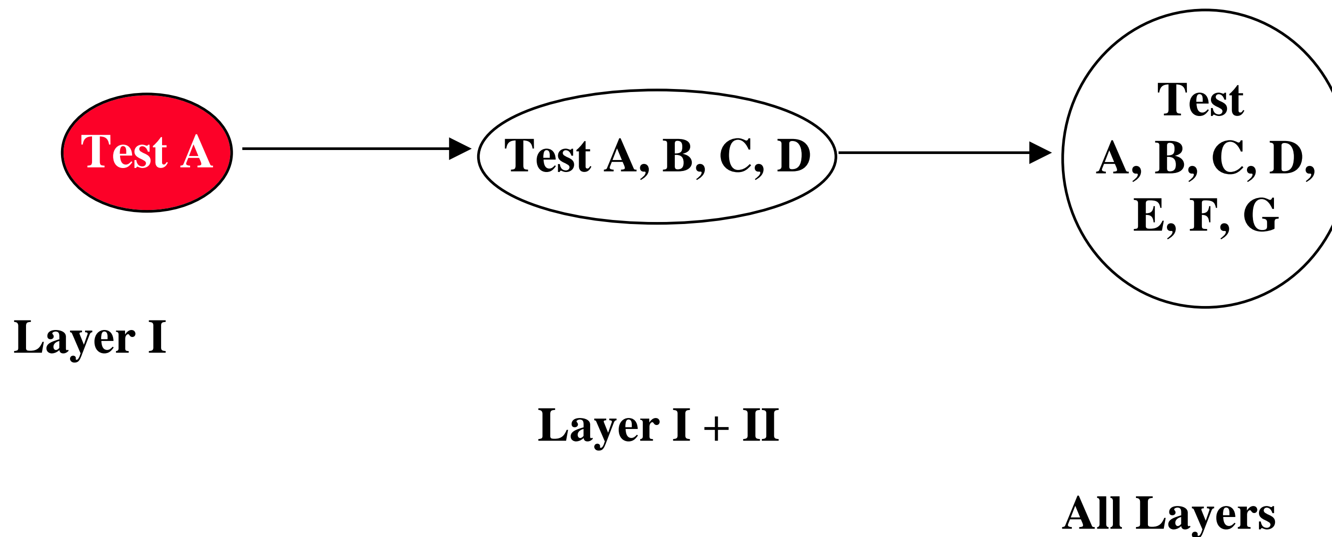
Do this until all subsystems are incorporated into the test

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.**

Top-down Integration Testing





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.

One solution to avoid too many stubs: *Modified top-down testing strategy*

- **Test each layer of the system decomposition individually before merging the layers**
- **Disadvantage of modified top-down testing: Both, stubs and drivers are needed**

Sandwich Testing Strategy

Combines top-down strategy with bottom-up strategy

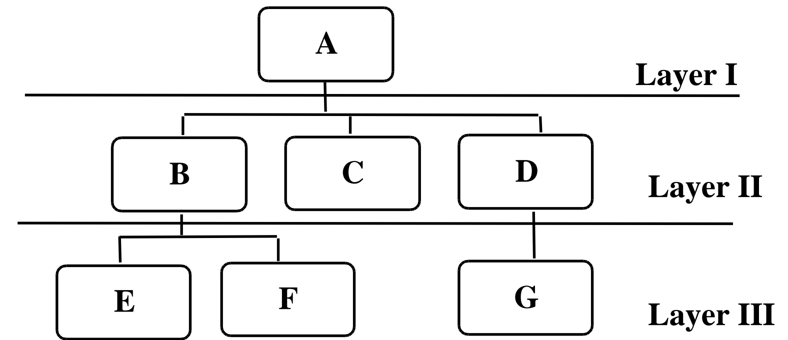
The system is view as having three layers

- **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



Test E

Test F

Test G

Bottom Layer Tests

Test B, E, F

Test D,G

Test A,B,C, D

Test A

Top Layer Tests

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 individual subsystems thoroughly before integration
- Solution: Modified sandwich testing strategy

Modified Sandwich Testing Strategy

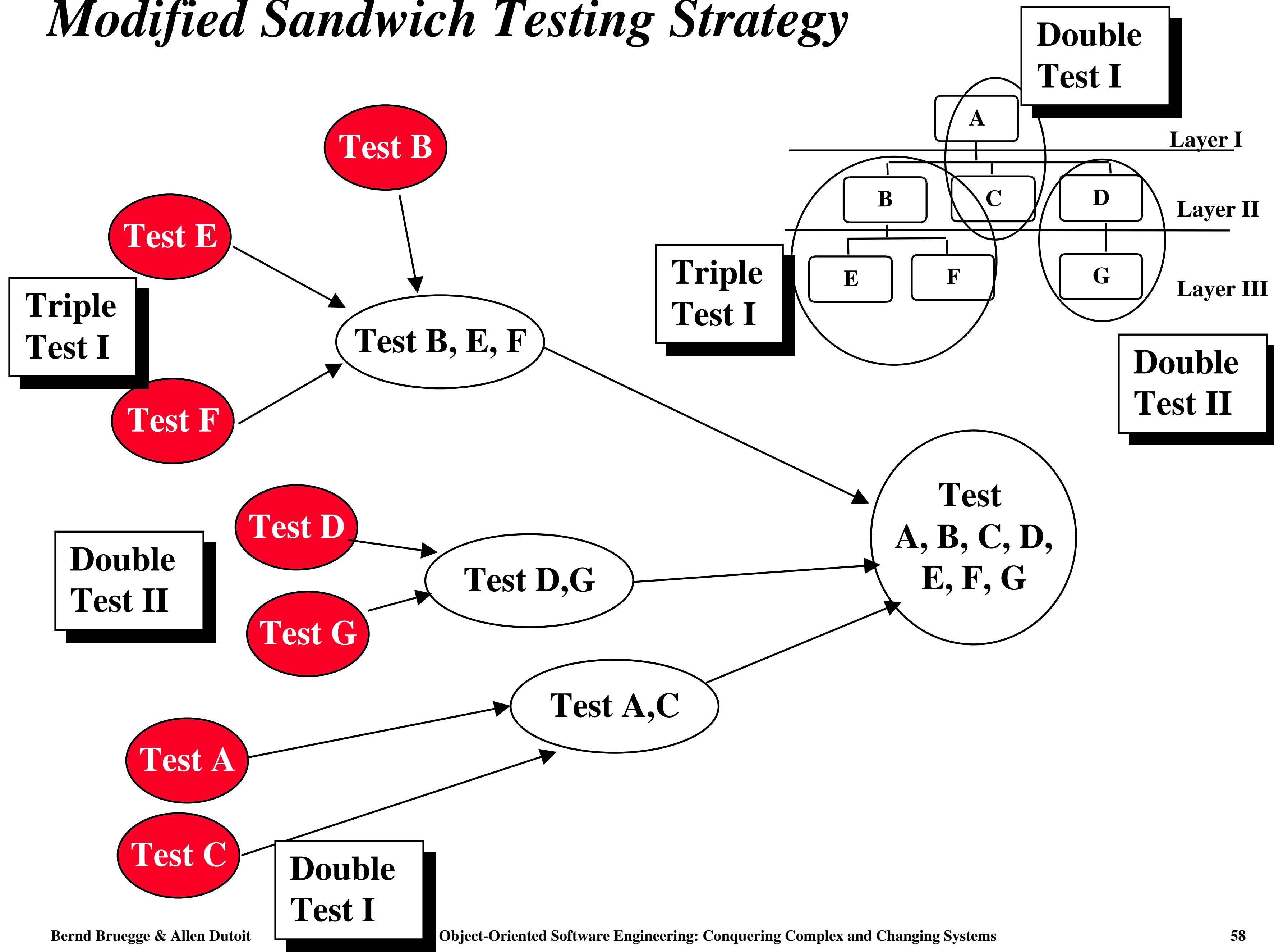
Test in parallel:

- **Middle layer with drivers and stubs**
- **Top layer with stubs**
- **Bottom layer with drivers**

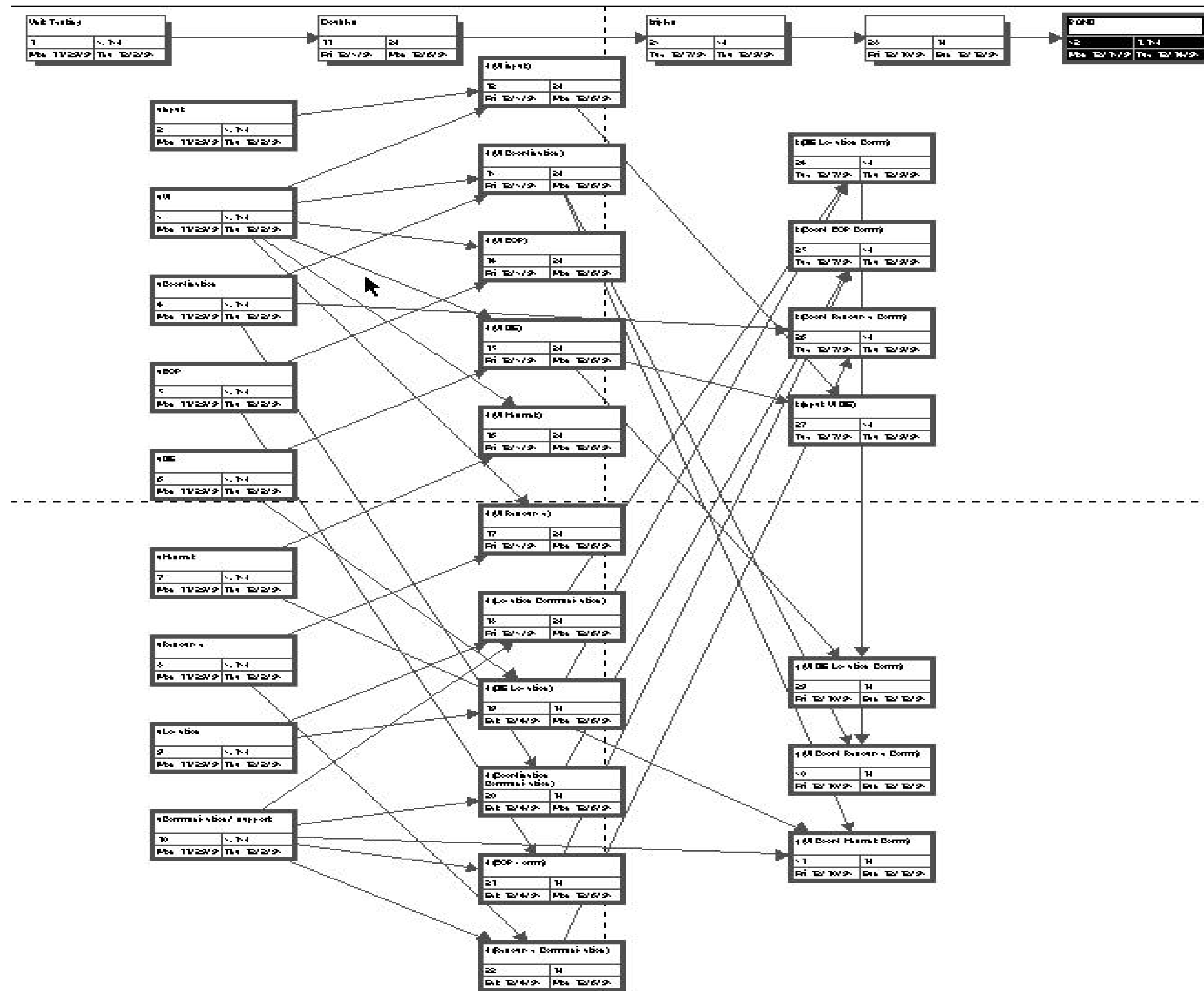
Test in parallel:

- **Top layer accessing middle layer (top layer replaces drivers)**
- **Bottom accessed by middle layer (bottom layer replaces stubs)**

Modified Sandwich Testing Strategy



Scheduling Sandwich Tests: Example of a Dependency Chart



Unit Tests **Double Tests** **Triple Tests** **SystemTests**

Steps in Integration-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**

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:**

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 component.**
- **Exercise all components and all calls (each component is called at least once and every component is called by all possible callers.)**
- **Use conditional and iteration testing as in unit testing.**

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.

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, maintain- ability & 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 the 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.**

Majority of all bugs in software is typically found by the client after the system is in use, not by the developers or testers. Therefore two kinds of additional tests:

Alpha test:

- **Sponsor 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 environ-ment**
- **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

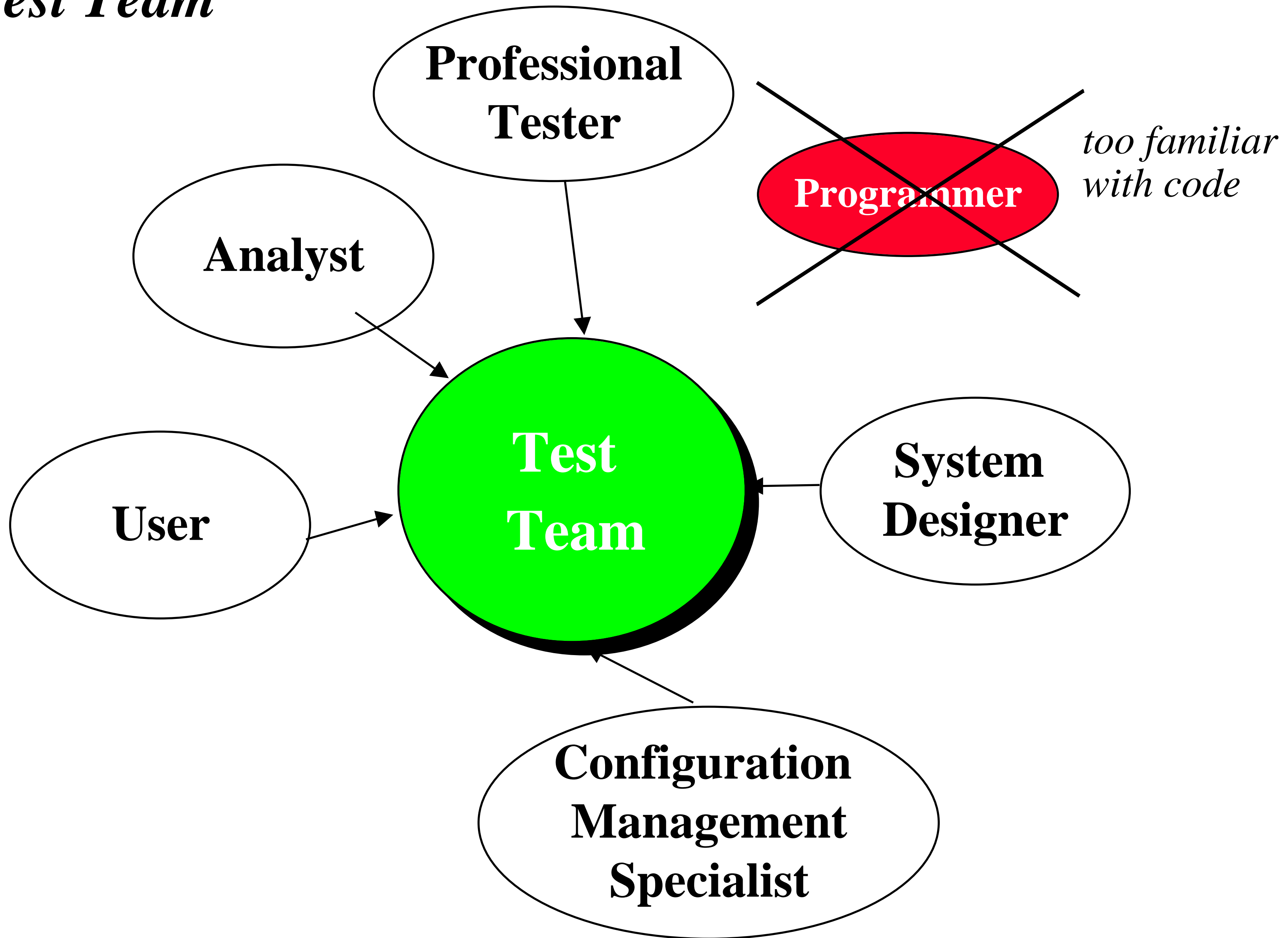
Execute the tests

Evaluate the test results

Change the system

Do regression testing

Test Team



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 integration testing

Testing has its own lifecycle