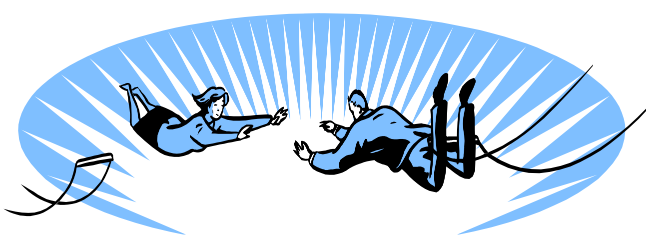
What is testing?

Why we do tests?

To avoid failures in software

Because software systems provide the infrastructure in virtually all industries today:

- air traffic control
- automotive systems
- water level management
- energy production and distribution



We want programs to be reliable!!

Do You Trust Your System?

The real wonder is that the system works as well as it does

(Peterson, 1996)

Airbus 319 Safety Critical Software Control

October 2005, BA Flight London – Bucarest

More than 2 minutes flight in degraded conditions

- Loss of autopilot
- Loss of most navigation displays
- Loss of radio power
- No MAYDAY possible !

How do we build trust in sw systems?

Any engineering process consists of

- construction activities
- techniques to check intermediate and final products

Testing is one technique to increase our confidence in the correctness of a software system.

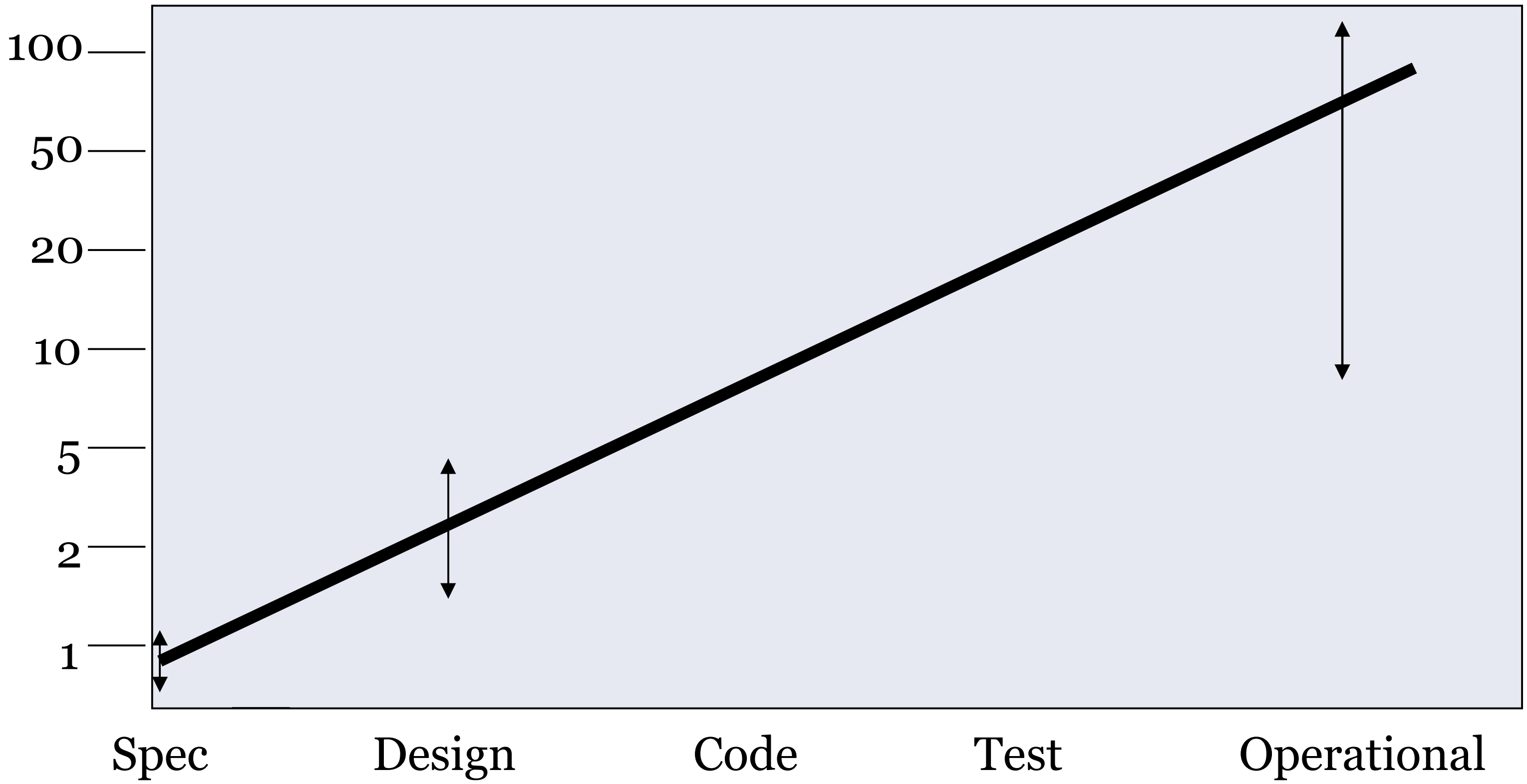
Whenever we use software, we incur some risk

- Risk may be small and consequences unimportant
- Risk may be great and the consequences catastrophic

Some statistics

- 30-85 errors are made per 1000 lines of source code
- Extensively tested software contains 0.5-3 errors per 1000 lines of source code
- The later an error is discovered, the more it costs to fix it.
- Error distribution: 60% design, 40% implementation.
- 66% of the design errors are not discovered until the software has become operational.

Relative cost of error correction



Validation and Verification

Validation : The process of evaluating software at the end of software development to ensure compliance with intended usage

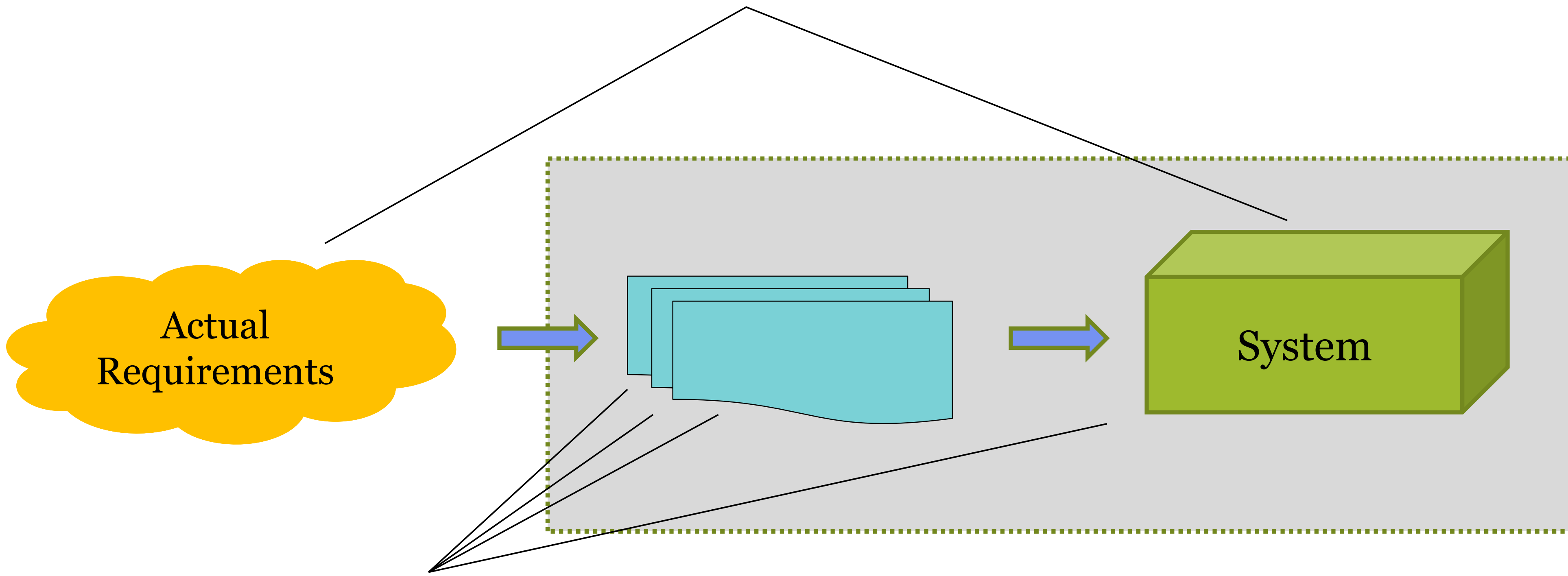
- *Are we building the right system?*

Verification : The process of determining whether the products of a given phase of the software development process fulfill the requirements established during the previous phase

- *Are we building the system right?*

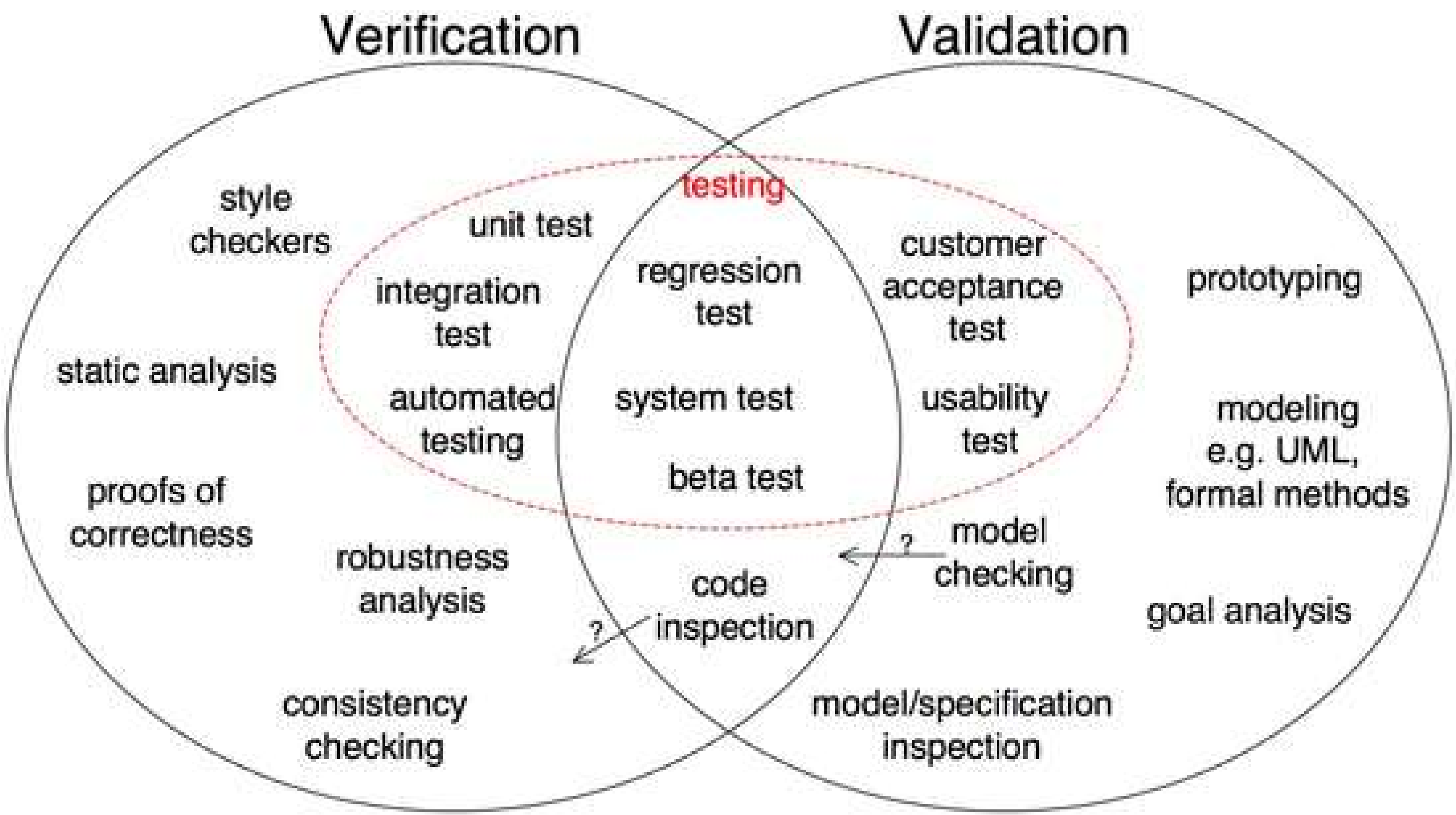
Validation and Verification

Are we building the right system?



Are we building the system right?

Testing, verification, and validation



Software verification

Software has some characteristics that make verification difficult:

- Many different quality requirements
- Evolving structure
- Uneven distribution of faults

If an elevator can safely carry a load of 1000 kg, it can also safely carry any smaller load

If a procedure correctly sorts a set of 256 elements, it may fail on a set of 255 or 53 or 12 elements, as well as on 257 or 1023

Software Faults, Errors & Failures

Patient describes symptoms

Software Faults, Errors & Failures

Doctor must find cause of symptoms

Patient describes symptoms

Software Faults, Errors & Failures

Doctor must find cause of symptoms

Patient describes symptoms

It looks for anomalous conditions,
eg. high blood pressure, or high cholesterol

Software Faults, Errors & Failures

Doctor must find cause of symptoms (fault)

Patient describes symptoms (failures)

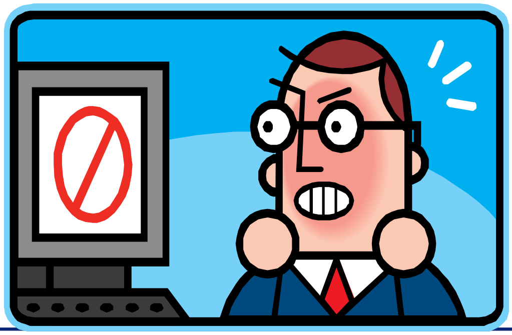
It looks for anomalous conditions, eg. high blood pressure, or high cholesterol (errors)

Software Faults, Errors & Failures

Software fault: A static defect in the software, the root of the failures. Faults are design mistakes.

Software error: the execution of an incorrect internal state (runtime) resulting from a fault

Software failure: External observable incorrect behavior with respect to the requirements or other description of the expected behavior



An example

```
Public static int numZero (int[ ] x) {
// return the number of occurrences of 0 in array x
int count = 0
for (int i=1; i < x.length; i++) {
      if (x[i] == 0) {
             count ++
       }
  }
return count
}
```

Input	x = [2,7,0]
Actual output	count =1
Expected output	count = 1

An example

```
Public static int numZero (int[ ] x) {
// return the number of occurrences of 0 in array x
int count = 0
for (int i=1; i < x.length; i++) {
      if (x[i] == 0) {
            count ++
       }
  }
return count
}
```

Input	x = [2,7,0]
Actual output	count =1
Expected output	count = 1

Input	x = [0,2,7]
Actual output	count = 0
Expected output	count = 1

An example

```
Public static int numZero (int[ ] x) {
// return the number of occurrences of 0 in array x
int count = 0
for (int i=1; i < x.length; i++) {
    if (x[i] == 0) {
        count ++
    }
  }
return count
}
```

fault

state

count = 0
i =1
x = [0,2,7]
PC =

Input	x = [2,7,0]
Output	count =1
Expected	count = 1

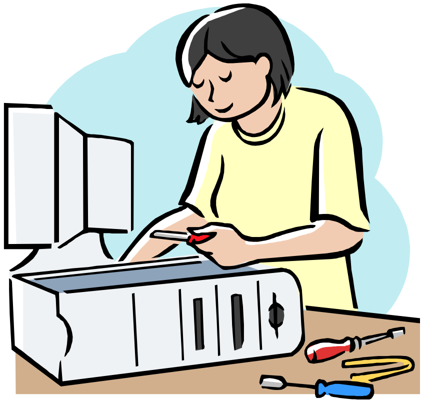
Input	x = [0,2,7]
Output	count = 0
Expected	count = 1

Both executions result in an error (the fault is executed) but only the second is a failure

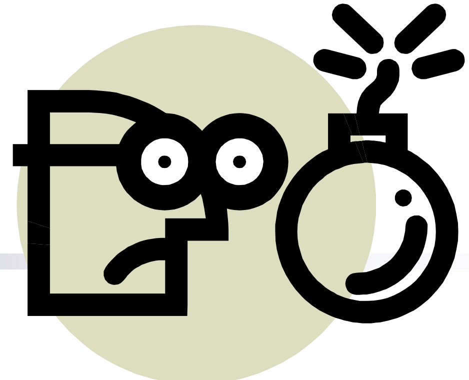
Testing Central Issue

Given a fault, not all inputs will trigger the fault into a failure.

Problem: finding inputs that cause a software failure



Fault and Failure Model



Three conditions necessary for a failure to be observed

1. Reachability: The location in the program that contain the fault must be reachable
2. Infection: When executed the states of the program must be incorrect
3. Propagation: The infected state must propagate to cause some output of the program to be incorrect

Exercise 1

The following program is faulty, as shown by the given test case.

```
public int findLast (int[] x, int y)
{
//Effects: If x==null throw NullPointerException
//   else return the index of the last element
//   in x that equals y.
//   If no such element exists, return -1
   for (int i=x.length-1; i > 0; i--)
   {
      if (x[i] == y)
      {
         return i;
      }
   }
   return -1;
}
   // test:  x=[2, 3, 5]; y = 2
   //        Expected = 0
```

1. Identify the fault
2. For the given test case, give the error state
3. Give a test case that does not execute the fault
4. Give a test case that execute the fault but does not result in an error state
5. Give a test case that results in an error but not in a failure

Exercise 1 - solution

The following program is faulty, as shown by the given test case.

```
public int findLast (int[] x, int y)
{
//Effects: If x==null throw NullPointerException
//   else return the index of the last element
//   in x that equals y.
//   If no such element exists, return -1
   for (int i=x.length-1; i > 0; i--)
   {                       i >= 0
      if (x[i] == y)
      {
         return i;
      }
   }
   return -1;
}
   // test:  x=[2, 3, 5]; y = 2
   //        Expected = 0
```

Identify the fault

Actual output = -1

The for-loop should include the 0 index.

For the given test case, give the error state

x = [2, 3, 5]

y = 2

i = 0

PC = just after the evaluation of the condition i>0 and before return -1

Exercise 1 - solution

The following program is faulty, as shown by the given test case.

```
public int findLast (int[] x, int y)
{
//Effects: If x==null throw NullPointerException
//   else return the index of the last element
//   in x that equals y.
//   If no such element exists, return -1
   for (int i=x.length-1; i > 0; i--)
                          i >= 0
   {
      if (x[i] == y)
      {
         return i;
      }
   }
   return -1;
}
   // test:  x=[2, 3, 5]; y = 2
   //        Expected = 0
```

Give a test case that does not execute the fault

Test: x = null, y = 2
Expected output = NullPointerException
Actual output = NullPointerException

Give a test case that execute the fault but does not result in an error state

Test: x = [2, 3, 5] , y = 3
Expected output = 1
Actual output = 1

Give a test case that results in an error but not in a failure

Test: x = [2, 3, 5] , y = 7
Expected output = -1
Actual output = -1

What is testing?

Testing is not the same as debugging or troubleshooting

Debugging: the diagnostic process where, given a failure, an attempt is made to find the associated fault.

Troubleshooting: The attempt to solving a fault given a software failure

What is testing?

Testing: the process of systematic evaluation of software by observing its execution possibly identifying some of the failures

- A failure is any variance between actual and expected observable results.

Testing is one way to increase quality of software.

It is part of verification and validation process in which testers and developers work together to reduce the software risk.

What is testing?

Testing can only show the presence of failures, and not the correctness of a program

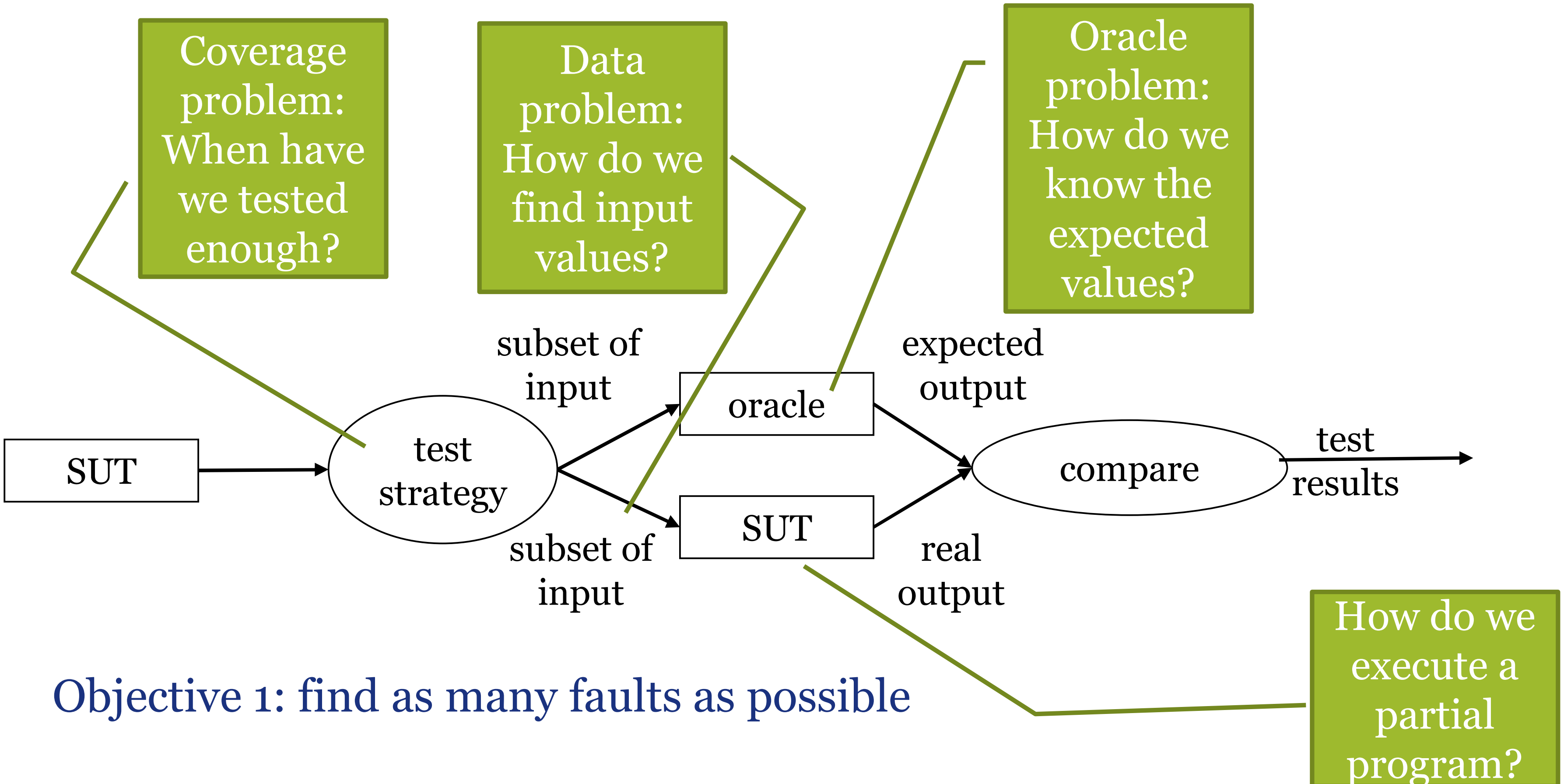
What if there are no failures?

- Good software or bad tests

But correctness via testing is impossible to achieve

- Edsger Dijkstra: *Program testing can be used to show the presence of bugs, but never to show their absence!*

Testing process



Objective 1: find as many faults as possible

Objective 2: increase confident that the software works

Test Cases

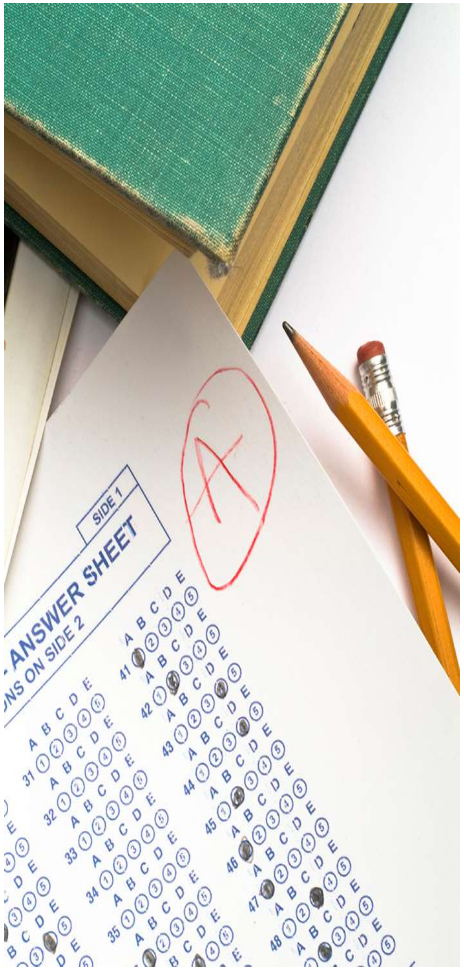
Test case values: the input necessary to complete an execution of a program

Expected results: The result that will be produced when executing the test if the program satisfies its intended behavior

Test case: The test case values, expected results, any other inputs necessary to start and conclude the execution

Test suite: a set of test cases

Test cases are used to determine if a program satisfies a test requirement



Adequacy of a test suite

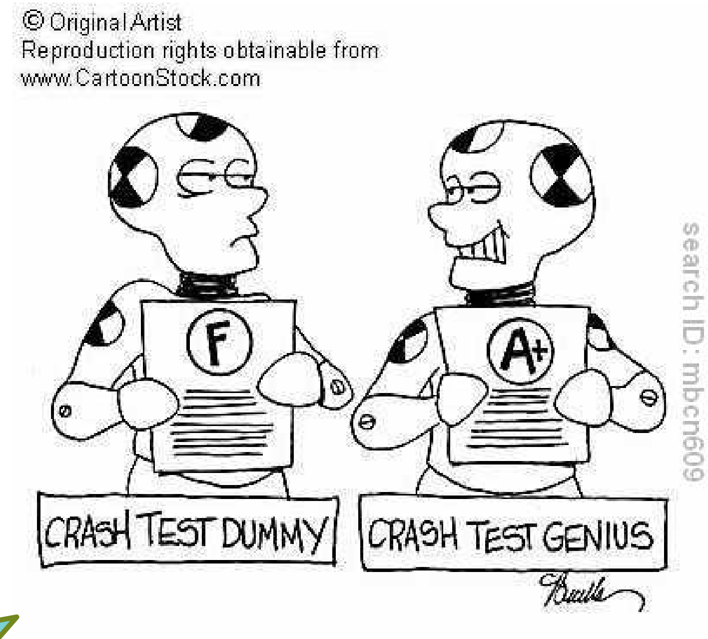
- Ideally – exhaustively test everything
- Practically impossible
- Test coverage criteria are measures of adequacy to increase the confidence that we have tested *enough*

Testing activities

Test Activities

Pre-test activities

- Analysis
- Design
- Coding



Software testing

- Unit
- Integration
- System
- Acceptance

Post-test activities

- Release
- Maintenance

Test Analysis and Design

Test design (criteria based):

design of test cases to satisfy some engineering criteria (e.g. coverage)

- Intellectually challenging
- Maths, logics, programming, …

Test design (human based): design of test cases based on domain knowledge of the program

- Empirical knowledge needed
- Psycology, law, …

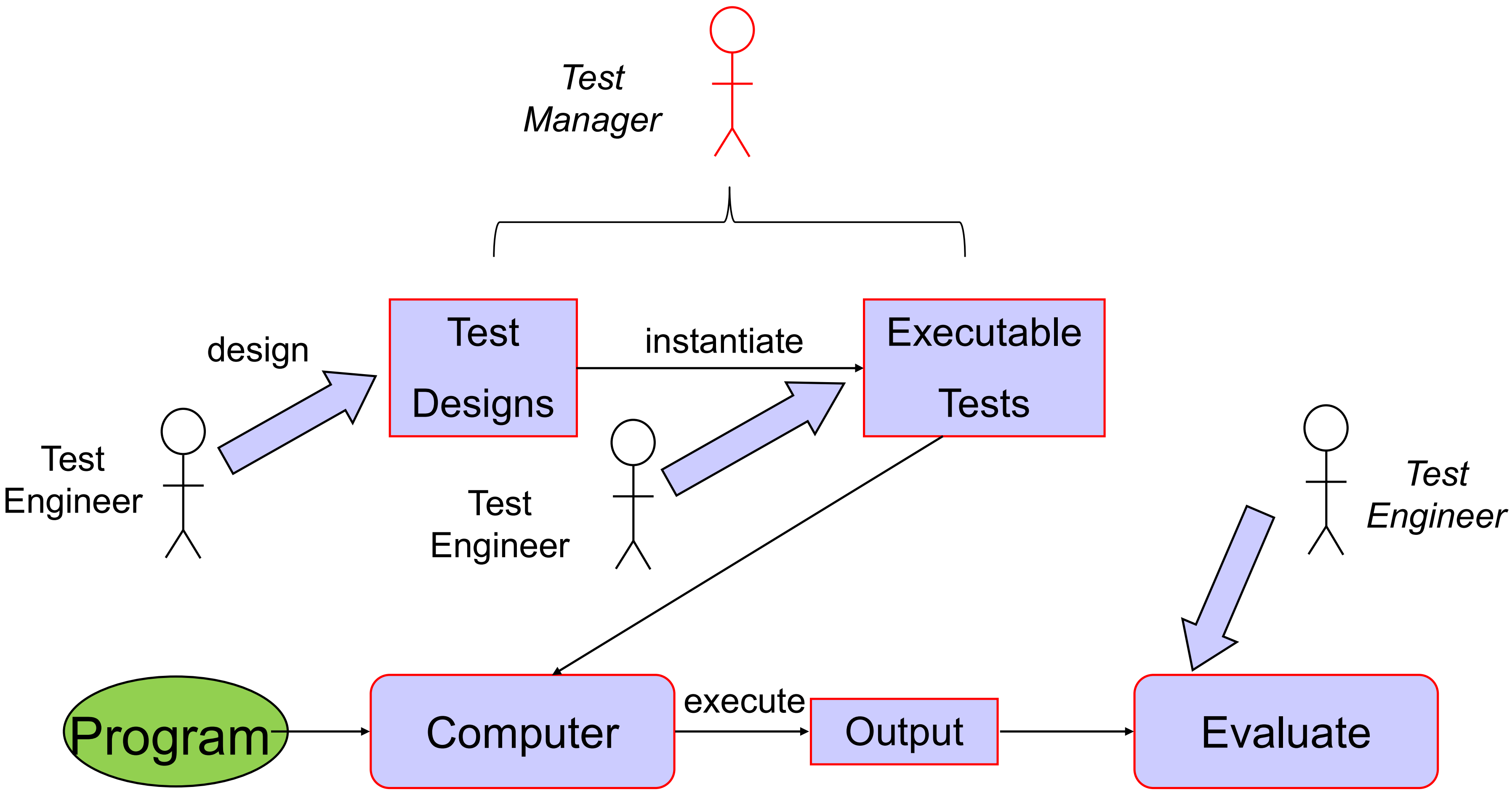
Test coding

- Test automation: embed test cases into executable script
  - Rather technical
  - Require knowledge of programming

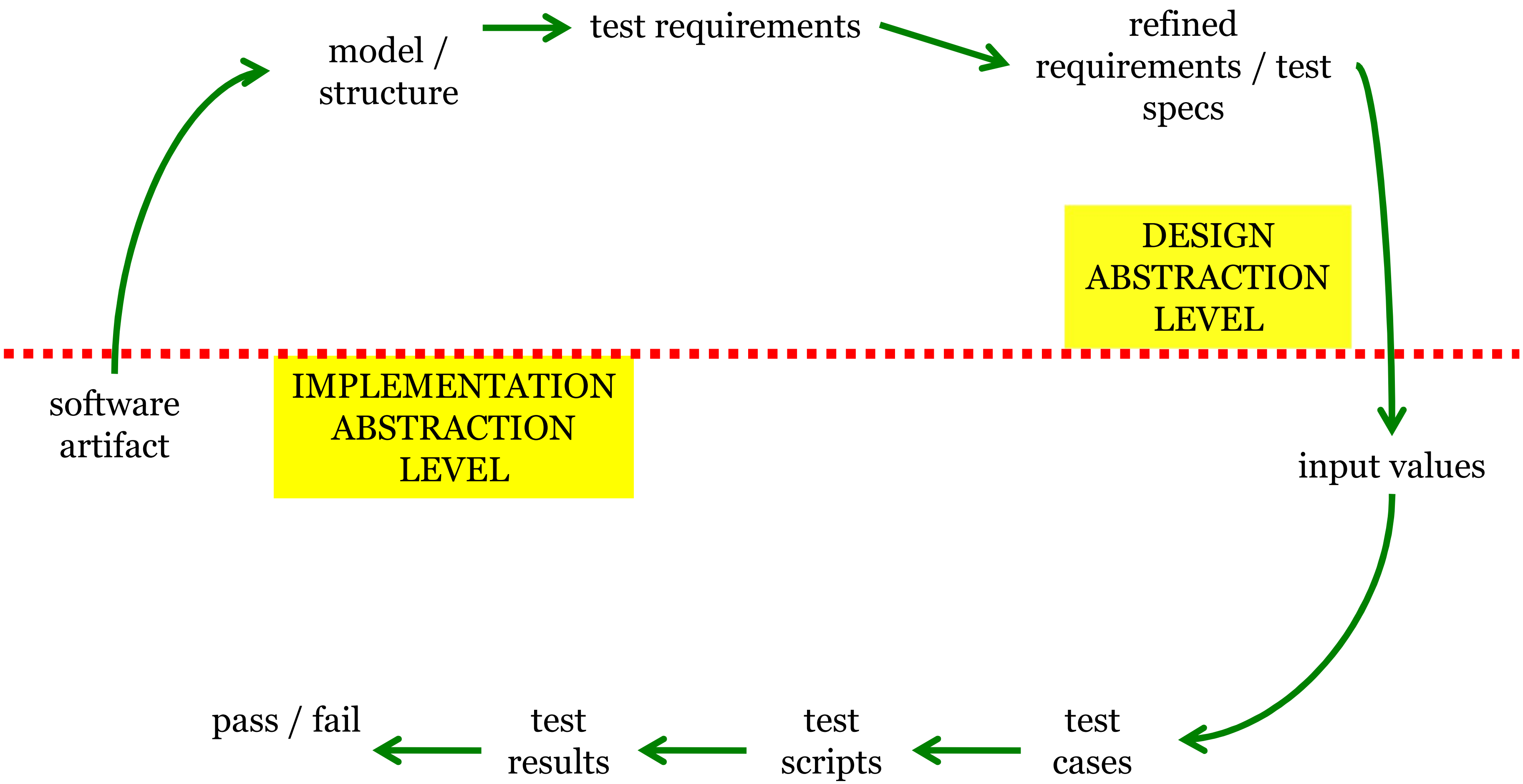
- Test execution: Run tests of the software and record the results

- Test evaluation: evaluate results and report to developer

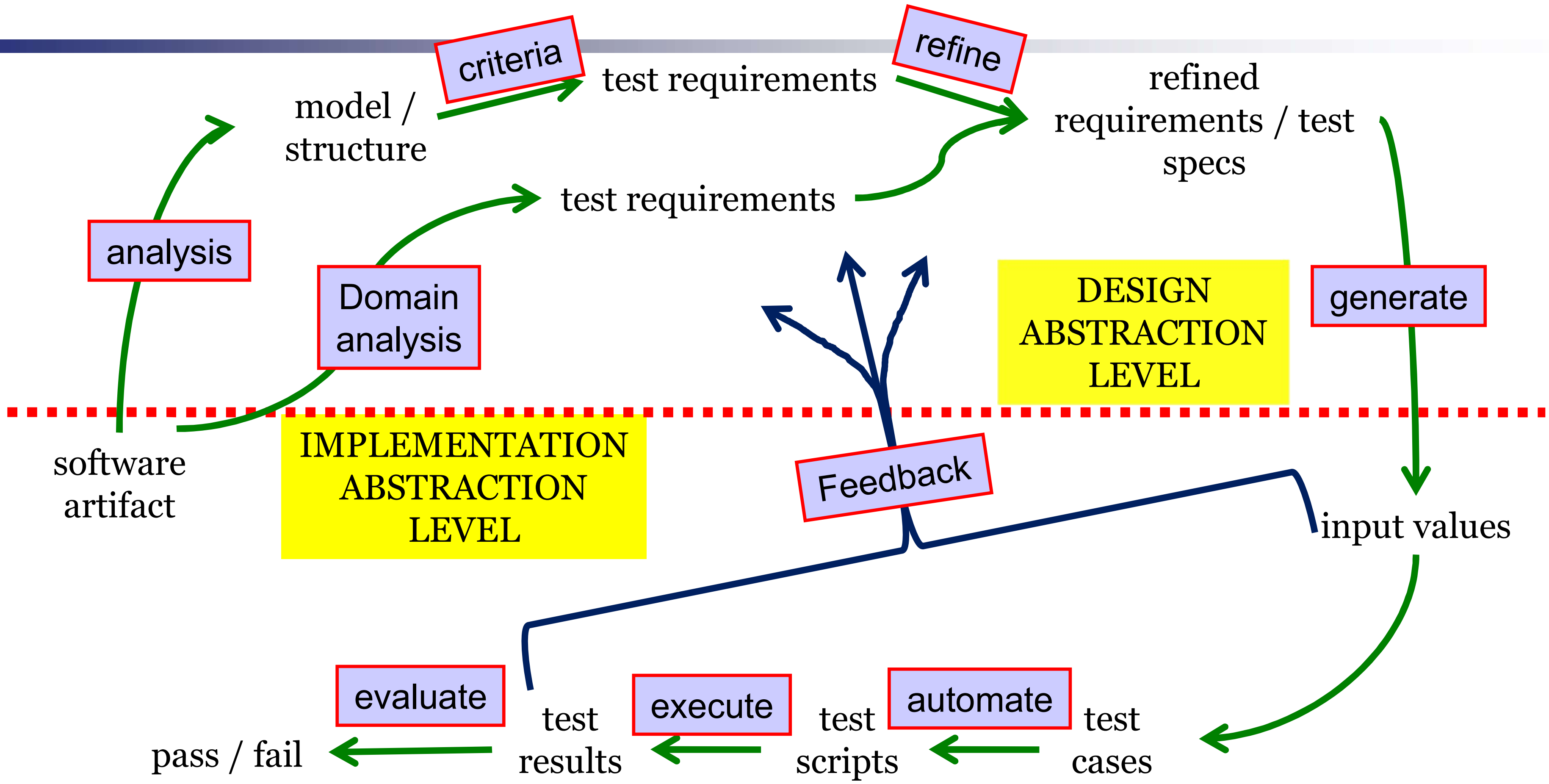
Tester Activities



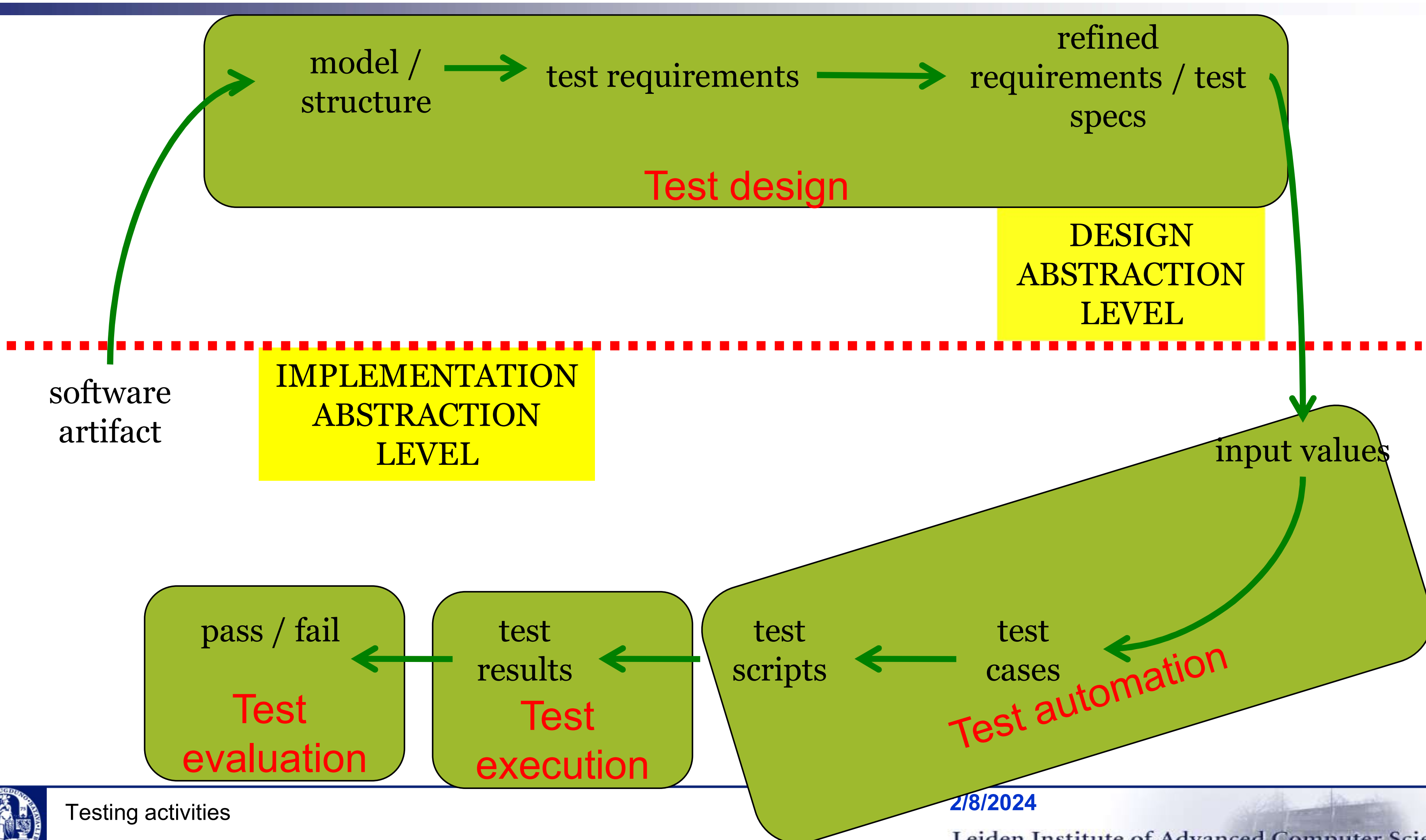
Model Driven Test Design



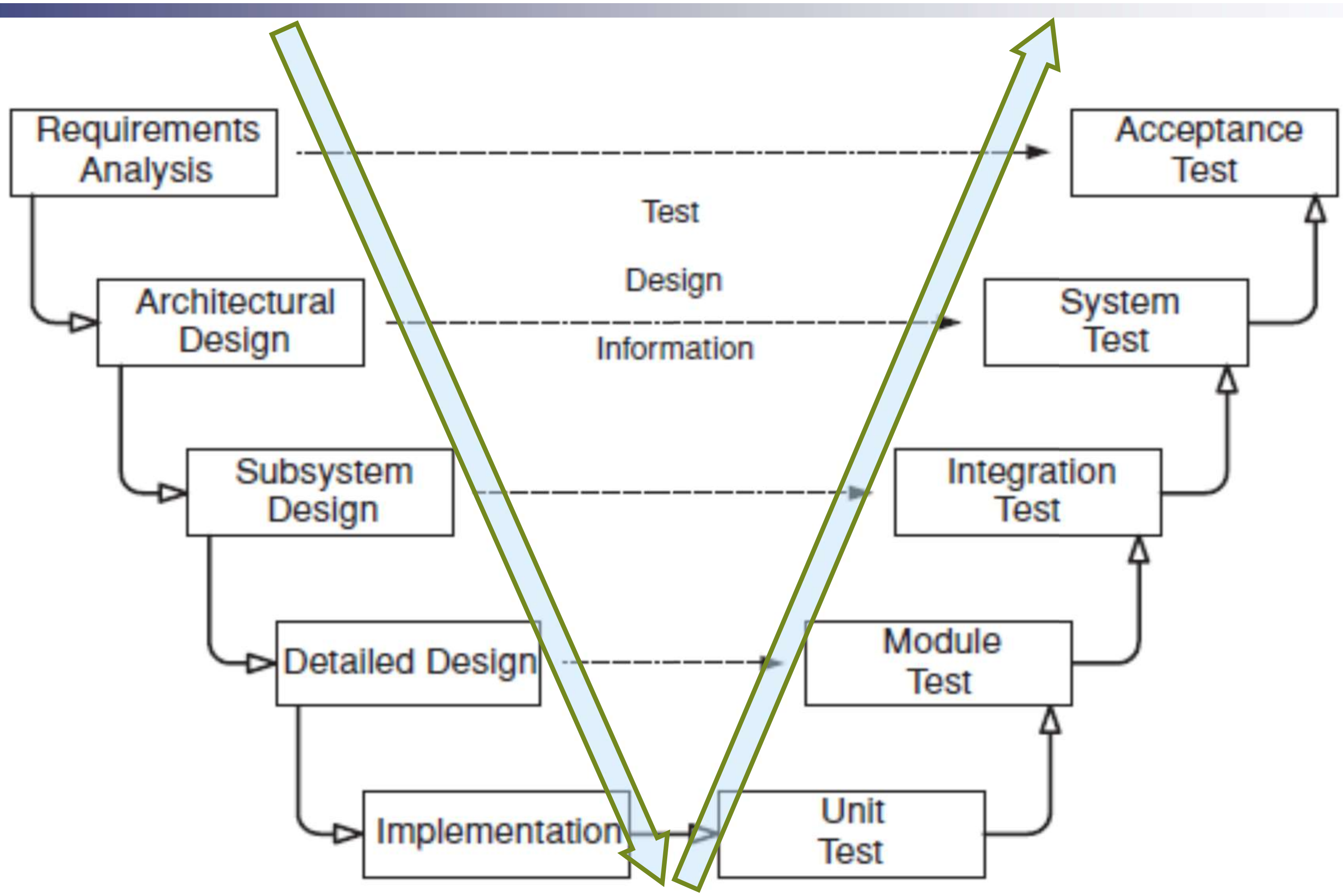
Model Driven Test Design - steps



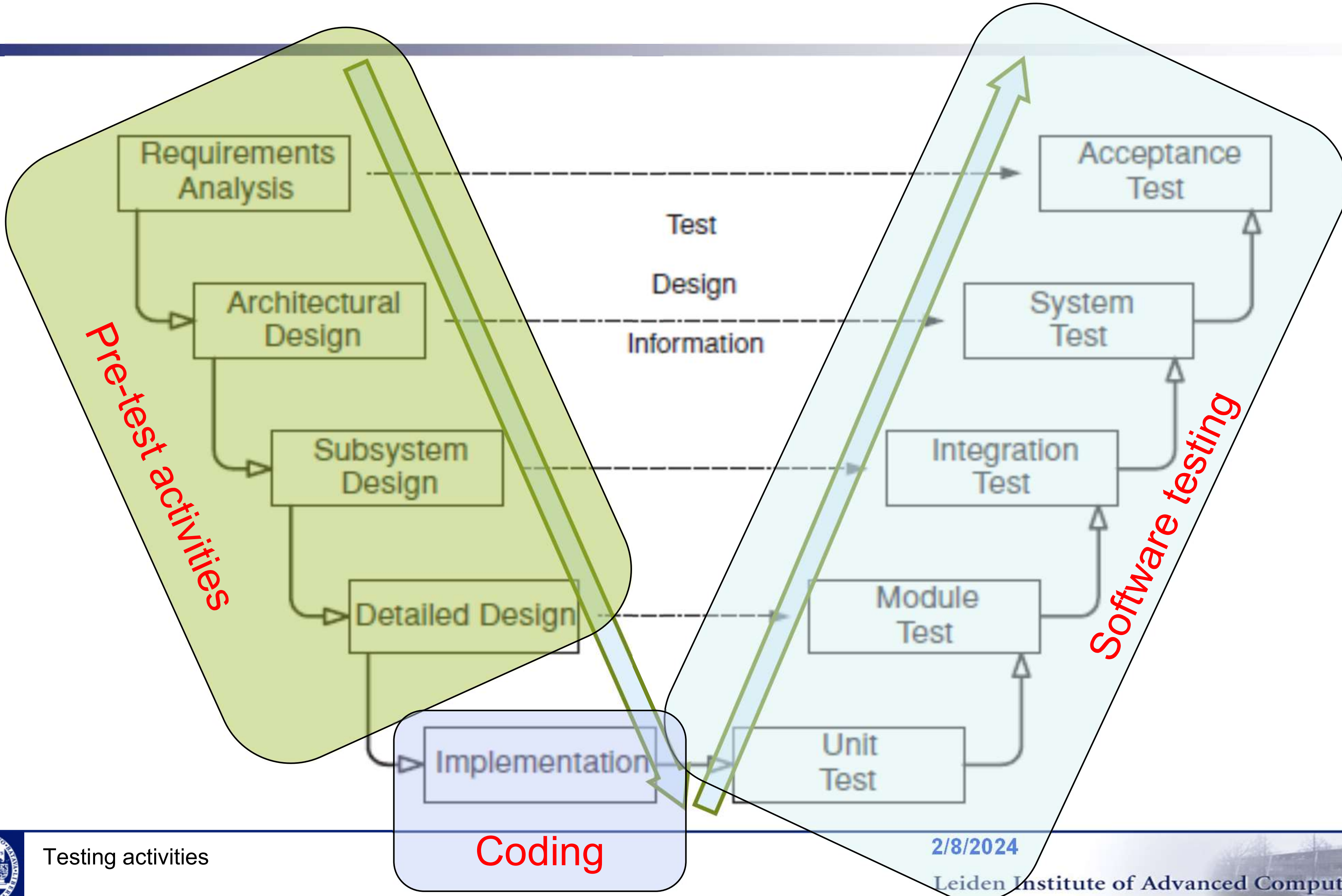
Model Driven Test Design - activities



Software Development Activities



Software Development Activities



White and Black Box Testing

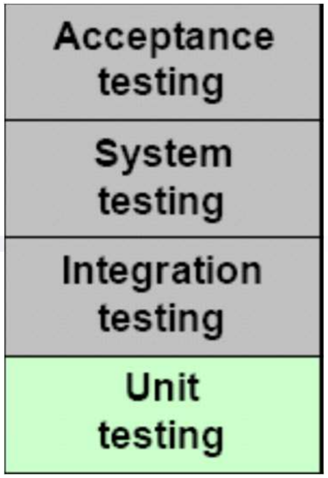
Black-box testing: Deriving tests from external descriptions of the software

- Requirements, specification, design

White-box testing: Deriving test case from the source code internals

- Conditional, statements, internal state

Unit Testing



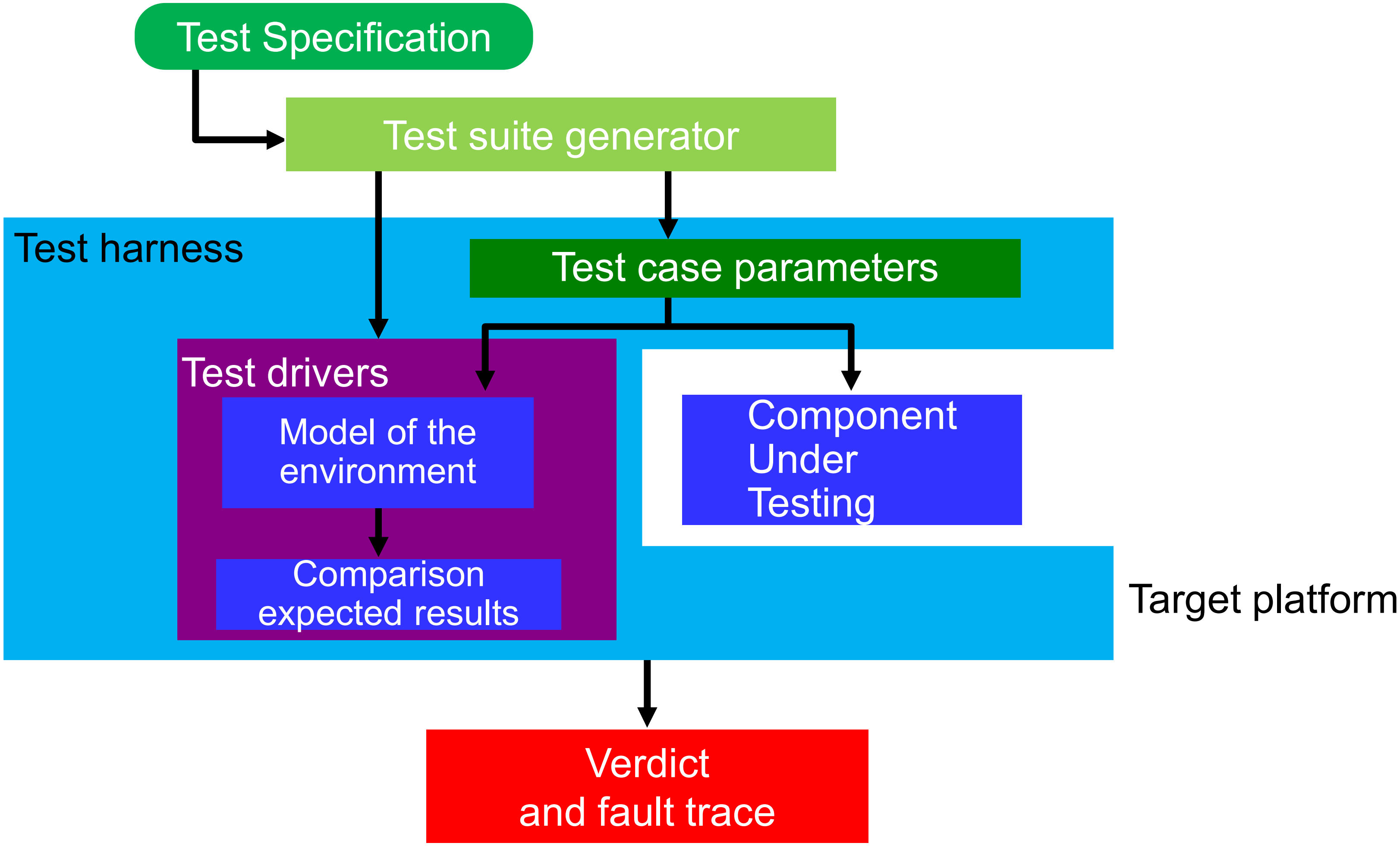
A unit is a small testable software component

- Procedure, method
- Class

Assess software with respect to implementation or detailed design

Units are tested in isolation

Unit Testing Process



Integration Testing

Acceptance testing

System testing

Integration testing

Unit testing

Testing of more than one (tested) unit together to determine if they function correctly.

Assess software with respect to subsystem design

Focus on interfaces and communication between units

System Testing

Acceptance testing
System testing
Integration testing
Unit testing

Testing the system as a whole.

Assess software with respect to architectural design

Verify that specifications are met

Acceptance Testing

Acceptance testing

System testing

Integration testing

Unit testing

Similar to system testing in that the whole system is checked, but the important difference is the change in focus

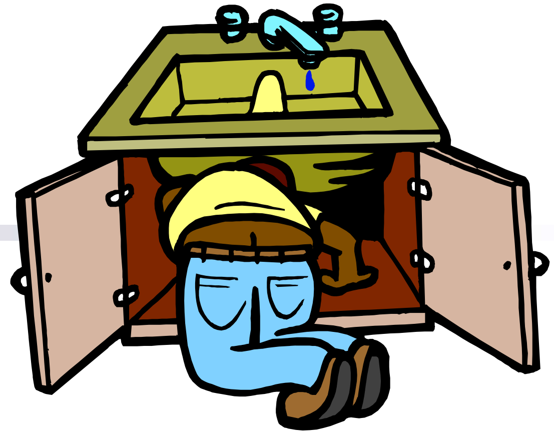
Assess software with respect to requirements

Validate that the system can be used for the intended purpose

- Done by real business users
- It enables the customer to determine whether to accept the system or not

Also called beta-testing

Regression Testing



Testing during maintenance

Assess software with respect to new and old requirements

- Re-run test cases only if they include changed elements, or touch modified control/data flow nodes and edges.

To automate selection:

- Tools record elements touched by each test case
- Tools note changes in program
- Tools check test-case database for overlap