Introduction to software testing

Spec-based testing

Professor Hossein Saiedian

EECS 348: Software Engineering

November 11, 2023

THE UNIVERSITY OF KANSAS

An exam question

What is the objective of testing?

(A) to show that a program works

(B) to improve the reliability of a program

(C) to find defects in the code

(D) to protect the end-user

(E) to protect the developing company

An exam question

What is the objective of testing?

(A) to show that a program works

(B) to improve the reliability of a program

(C) to find defects in the code

(D) to protect the end-user

(E) to protect the developing company

Dijkstra's view on testing

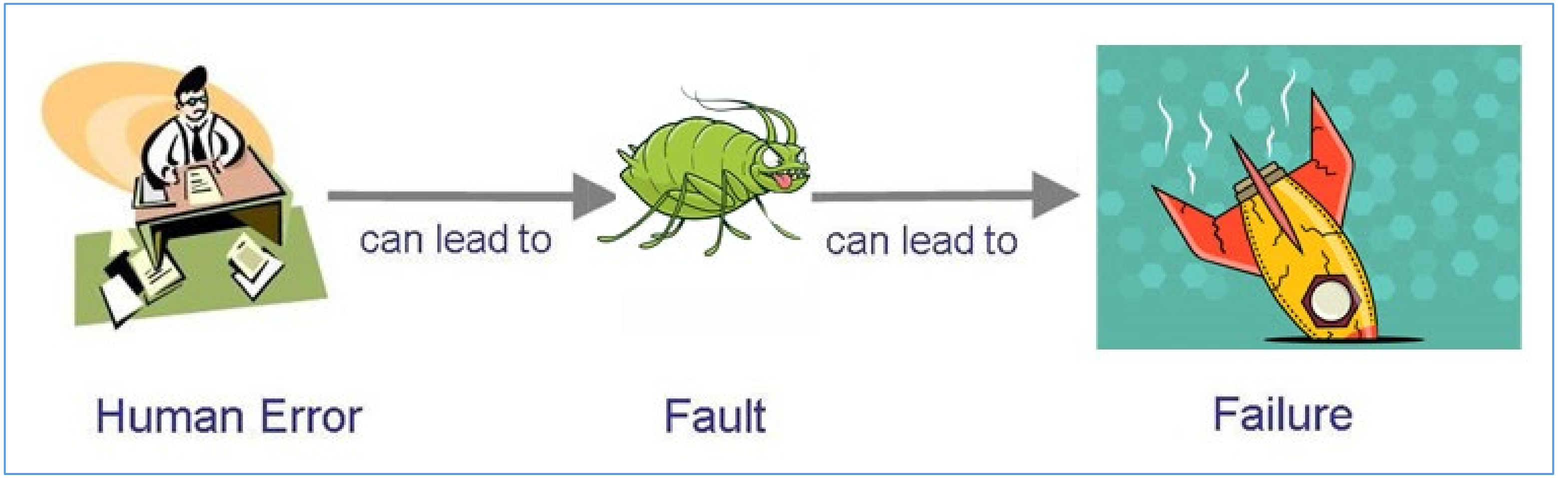
- Testing shows the presence, not the absence of defects (1969)
- In other words, a program can be proven incorrect by a test, but it cannot be proven correct

The objective of testing

- The objective of testing to find faults in a program
- A test is successful only when a fault is discovered
  - Fault identification is the process of determining what fault(s) caused the failure
  - Fault correction is the process of making changes to the system so that the faults are removed

Software defects

- A **fault** occurs when a human makes a mistake, called an **error**, in performing some software activities
- A **failure** is a departure from the system's required behavior



Program failures

- Failure occurs when the code corresponding to a fault executes
- A **test case** has an identity and is associated with a program behavior; it also has a set of inputs and expected outputs
  - A good test case can reveal a failure

A good test case

- Test case identifier (usually a short name for test management purposes)
- Name
- Purpose (e.g., a business rule)
- Pre-conditions (if any)
- Inputs
- Expected outputs
- Observed outputs
- Pass/fail?

Defects can be very bad

- Malfunctioning code in Therac-25 killed several people
- Ariane-5 crashed 37 seconds after takeoff because of malfunctioning code; cost near $500,000
- The IRS hired Sperry Corporation to build an automated federal income tax form processing process
  - An extra $90 M was needed to enhance the original $103M product
  - The IRS lost $40.2 M on interests and $22.3 M in overtime wages because refunds were not returned on time

Test-to-pass vs test-to-fail

- Objective of test-to-pass (constructive testing)
  - Software works minimally
  - Apply straightforward test cases for correct behavior
  - Capabilities are not pushed
  - The right way of initial testing
- Objective of test-to-fail (destructive testing)
  - Aggressive testing
  - Testing a feature in every conceivable way possible
  - Break the software (find faults)

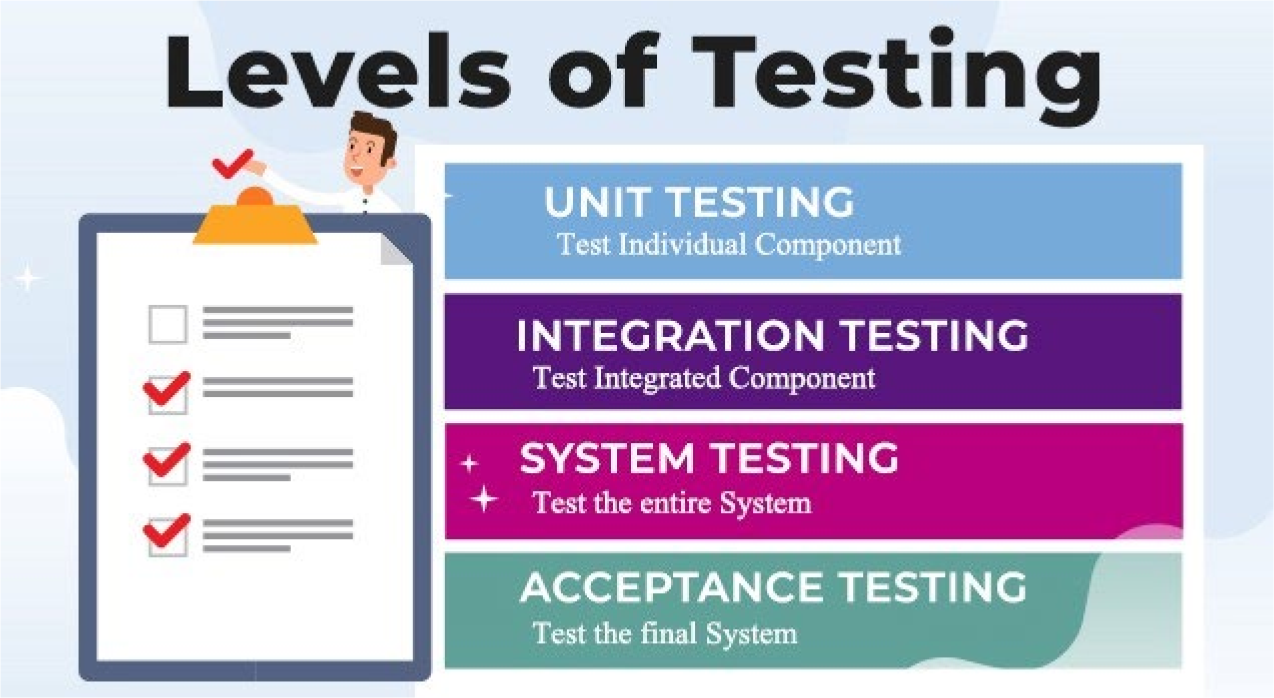
Umbrella activities for quality software

- Software testing
  - Static
  - Dynamic
- Formal specification and verification
- Technical reviews
- Software configuration management
- Software project tracking and control
- Metrics and measurement

Common testing organization

- Module testing, component testing, or **unit** testing
- Integration testing
- Function testing
- Performance testing
- Acceptance testing
- Installation testing

Common testing organization

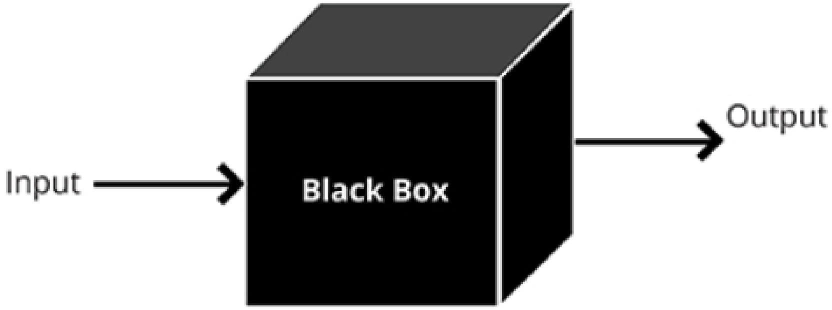


Test case generation techniques

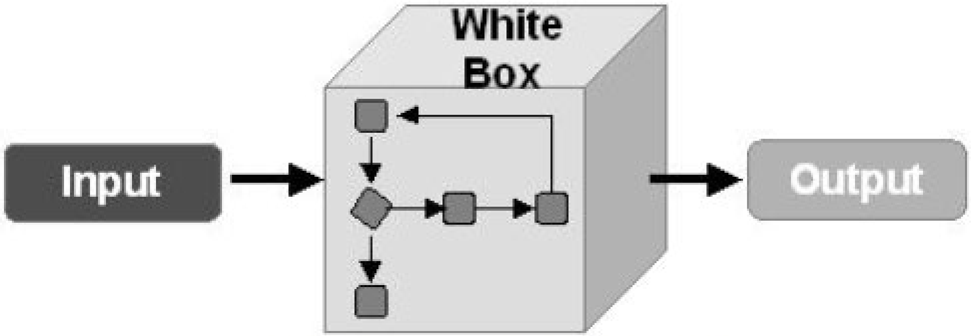
- Specification-based or black-box or closed-box or functional testing
  - Functionality of the test objects
  - No view of code or data structure – input and output only
- Code-based or white-box or clear box or glass box or structural testing
  - Structure of the test objects is explored
  - Internal view of code and data structures

Specification-based vs code-based testing

- Specification-based testing
  - Black-box testing
  - Functional testing
  - Mostly to establish confidence



- Code-based testing
  - White-box (clear-box, glass-box) testing
  - Structural testing
  - Mostly to seek faults



Code review

- Code walkthrough
  - Present code and documentation to review team
  - Team comments on correctness
  - Focus is on the code not the coder
  - No influence on developer performance
- Code inspection
  - Check code and documentation against list of concerns
  - Review correctness and efficiency of algorithms
  - Check comments for completeness
  - Formalized process

Formal proof techniques

- Write assertions to describe input/output conditions
- Generate theorems to be proven
- Locate loops and if-then statements and develop assertions for each
- Identify all paths from A_1 to A_n
- Cover each path so that each input assertion implies an output assertion
- Prove that the program terminates

Steps in choosing test cases

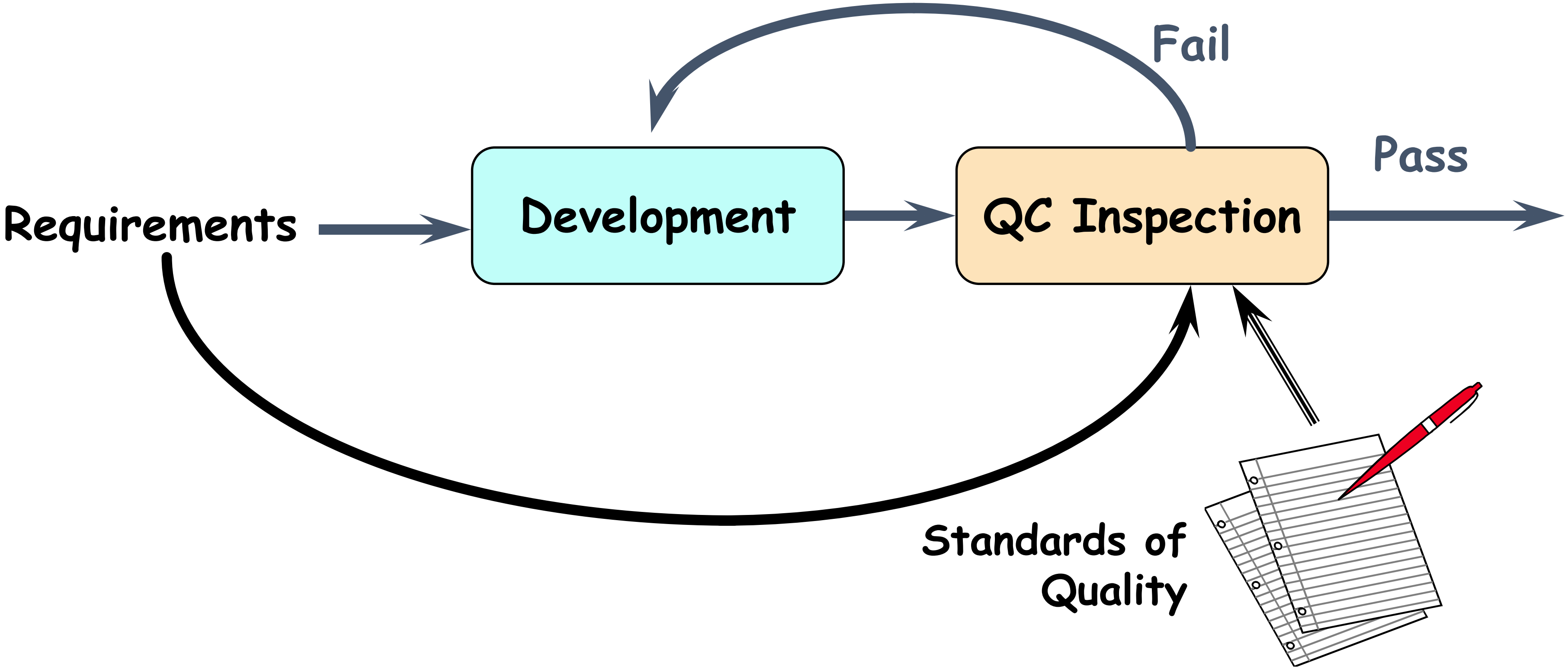
- Determining test objectives
  - Coverage criteria
- Selecting test cases
  - Inputs that demonstrate the behavior of the code
- Defining a test
  - Detailing execution instructions

When to stop testing

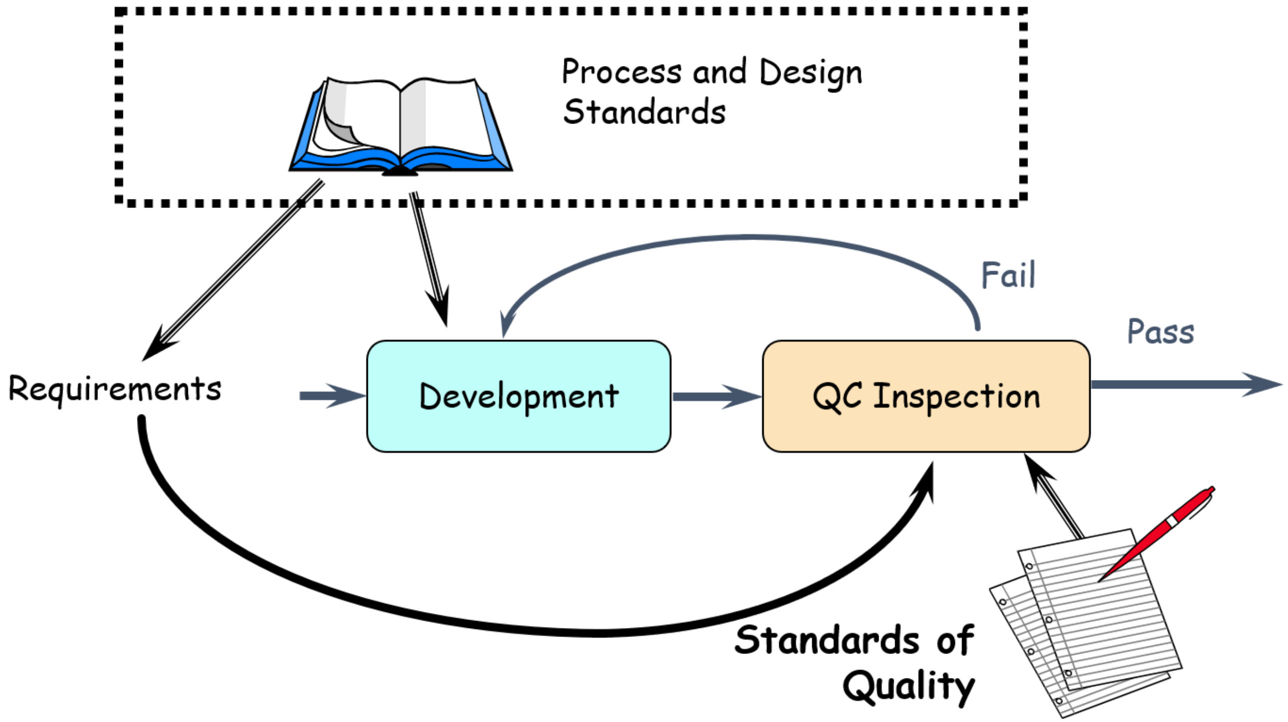
- When you run out of time
- When continued testing causes no new failures
- When continued testing reveals no new faults
- When you cannot think of any new test cases
- When you reach a point of diminishing returns
- When mandated coverage has been attained
  - Determine how many statement, branch, condition, or path, tests are required

Quality control

- Goal: Keep quality at an acceptable level by rejecting unacceptable products



Software quality assurance

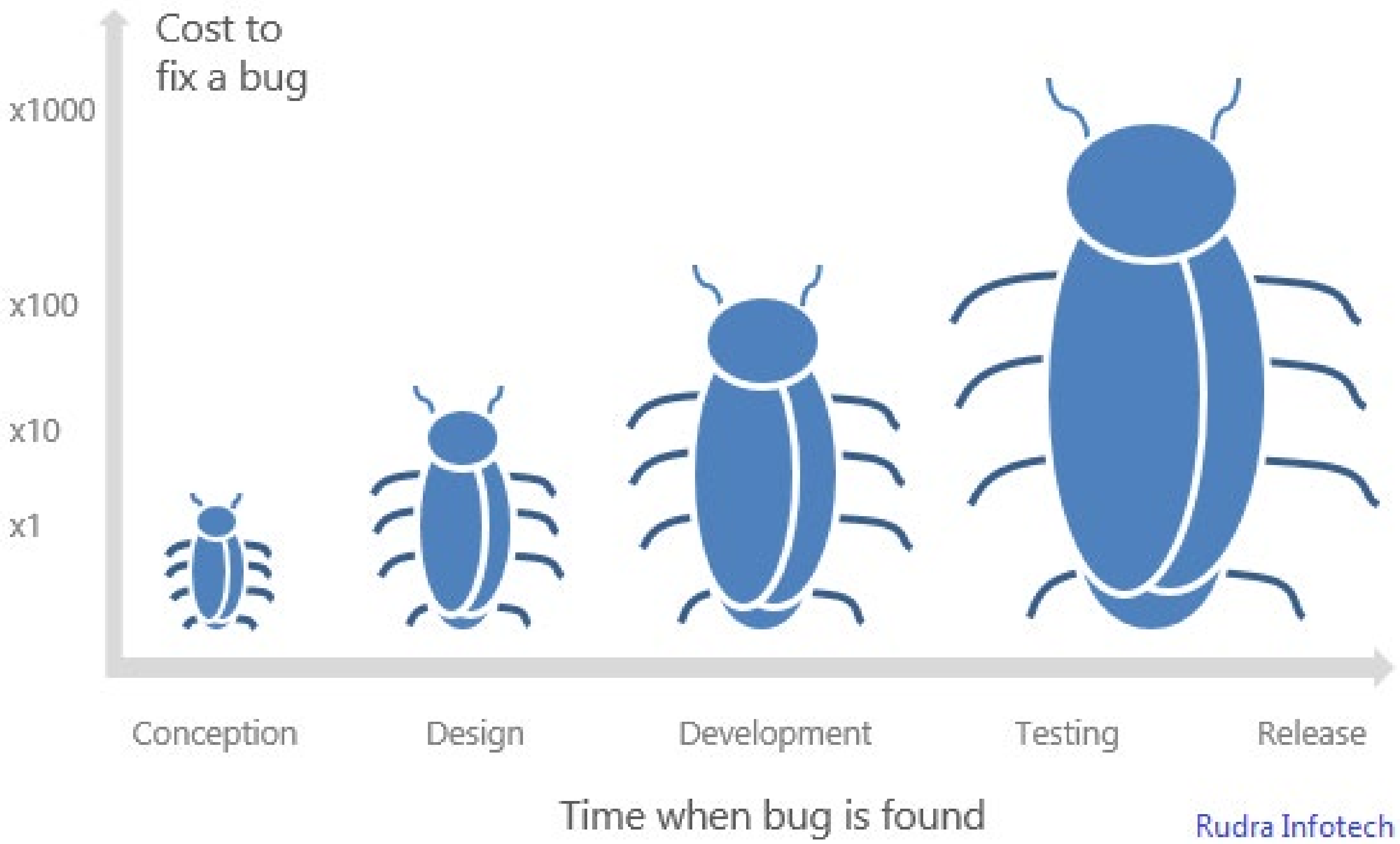


"A planned and systematic pattern of all actions necessary to provide adequate confidence that the product conforms to established technical requirements" IEEE

Quality assurance is more effective than quality control

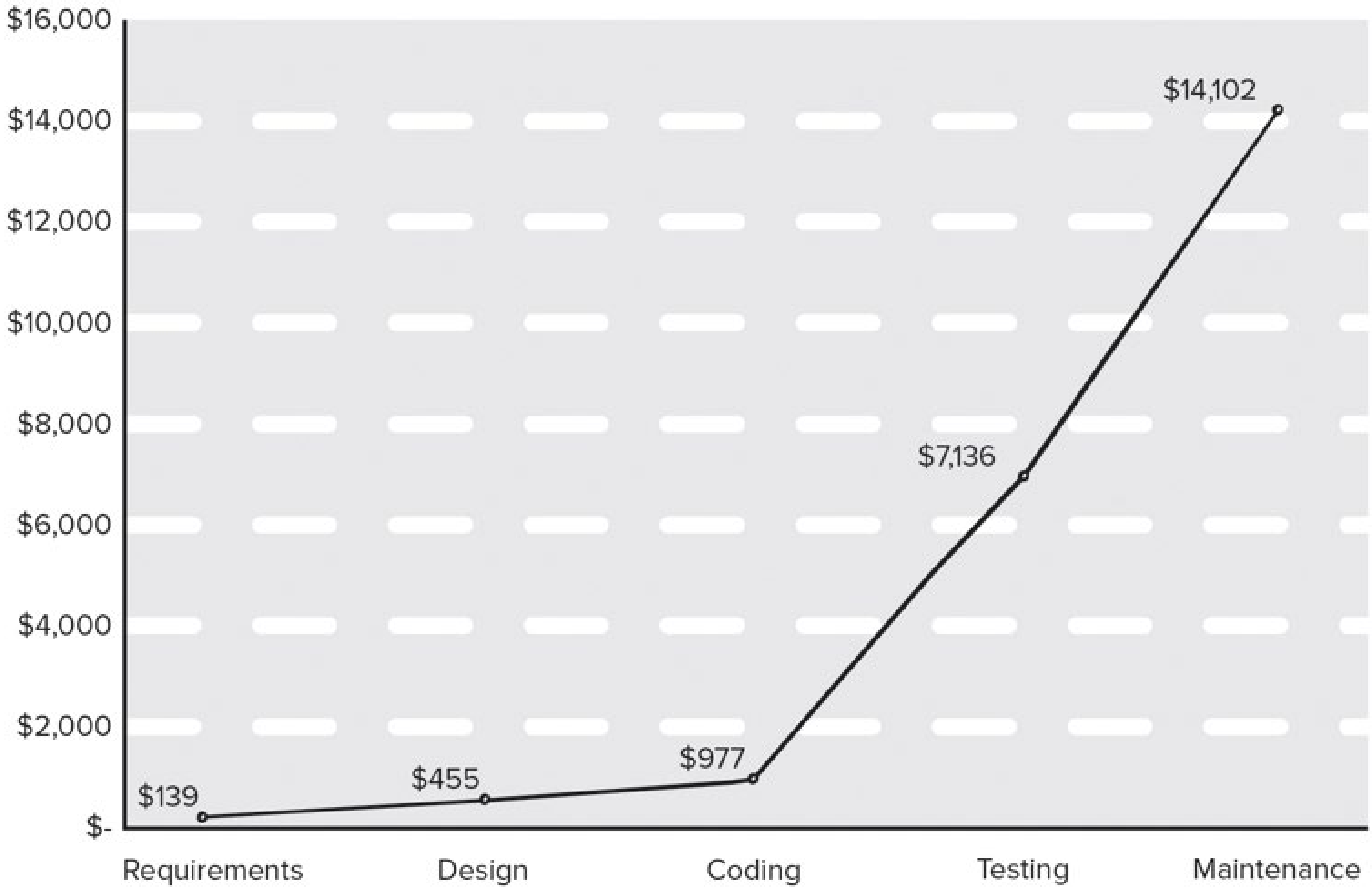
- Because the emphasis moves to the development process
- You attempt to fix problems ***before*** *and* ***during*** the development process
- You ***improve the process*** and therefore reduce the number of defects in a lasting manner

Relative costs to find and repair a defect



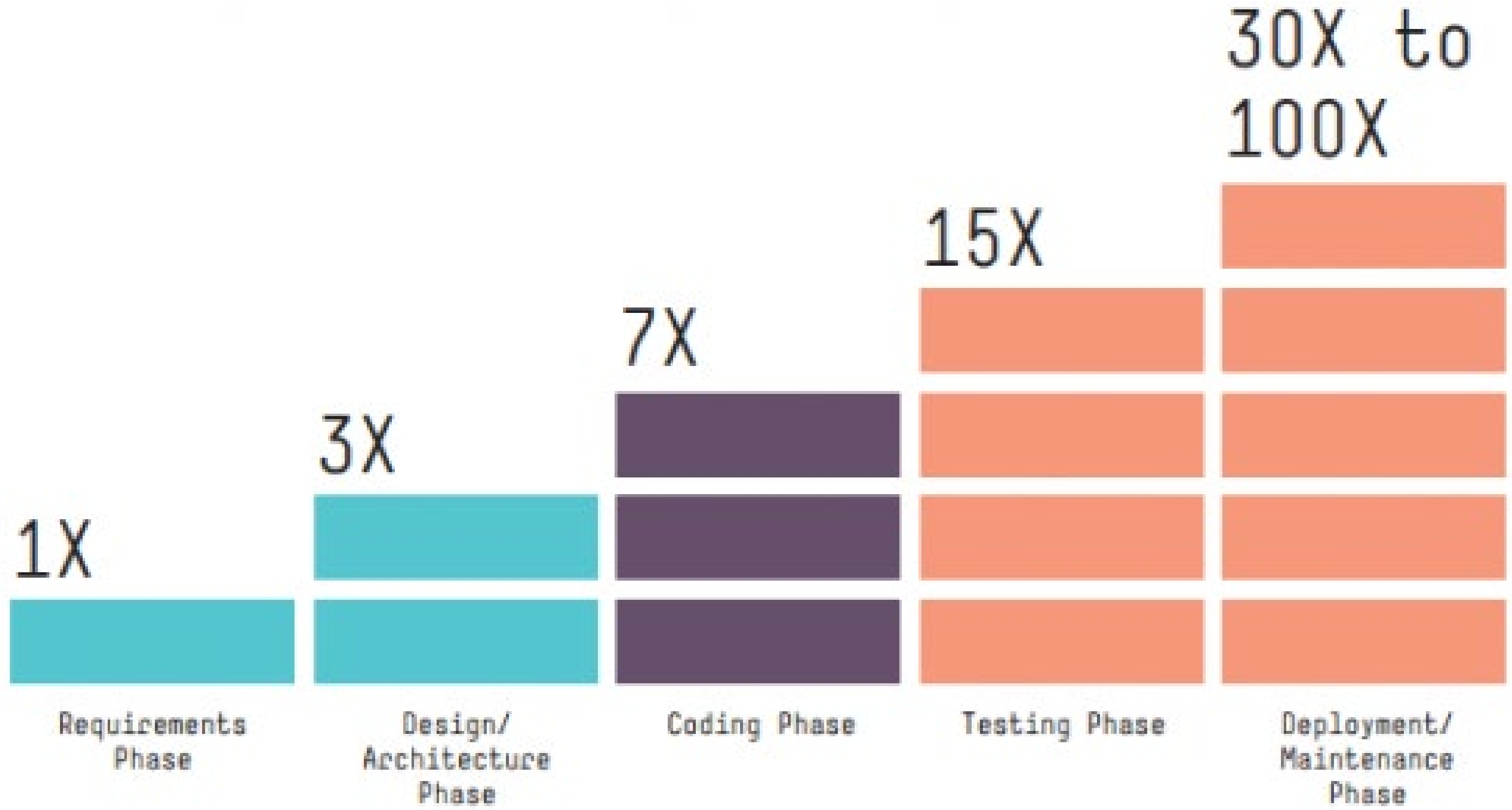
Relative costs to find and repair a defect

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.



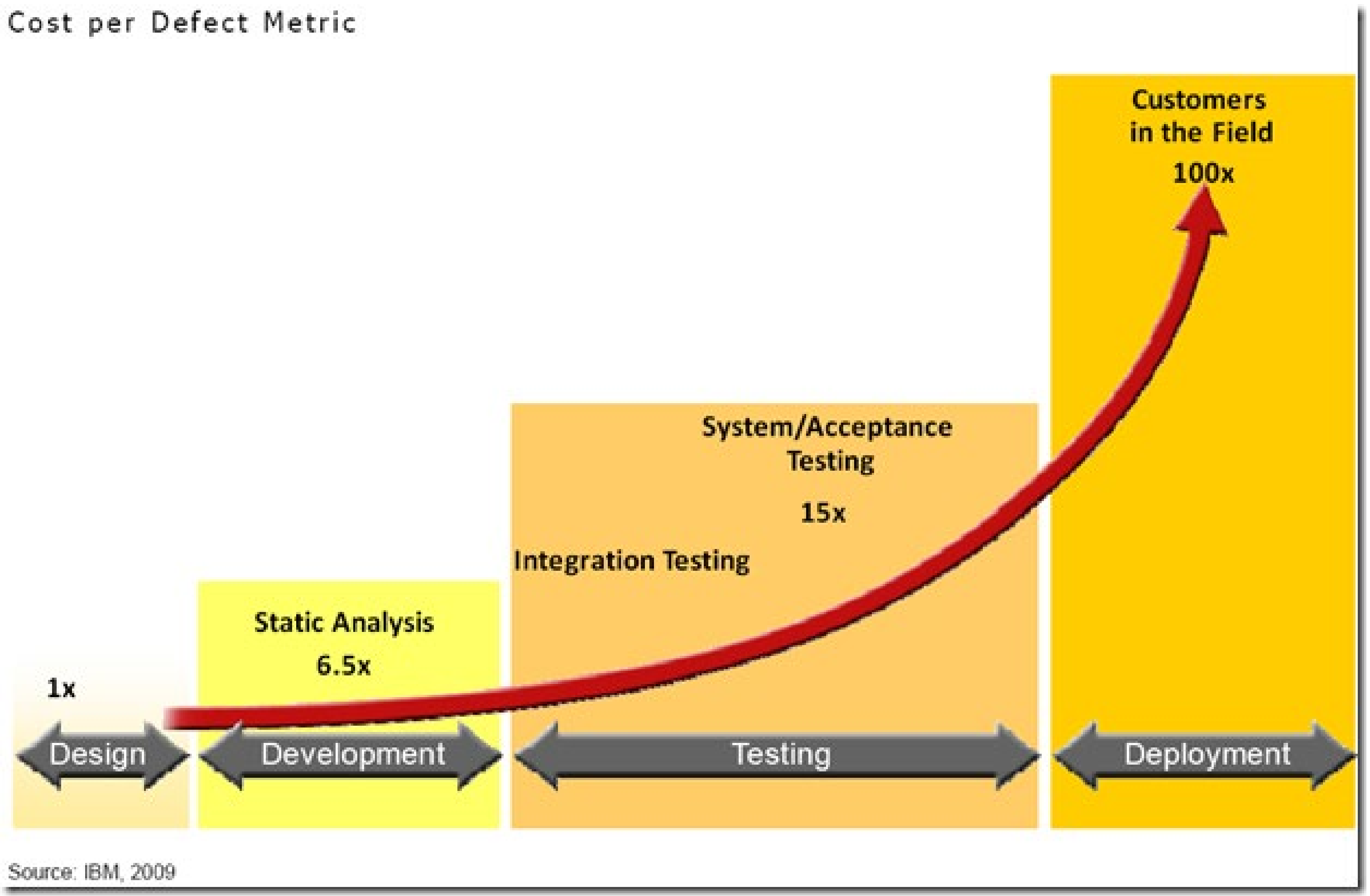
Relative costs to find and repair a defect

- Commonly accepted ratios
- Figure from: https://xbsoftware.com

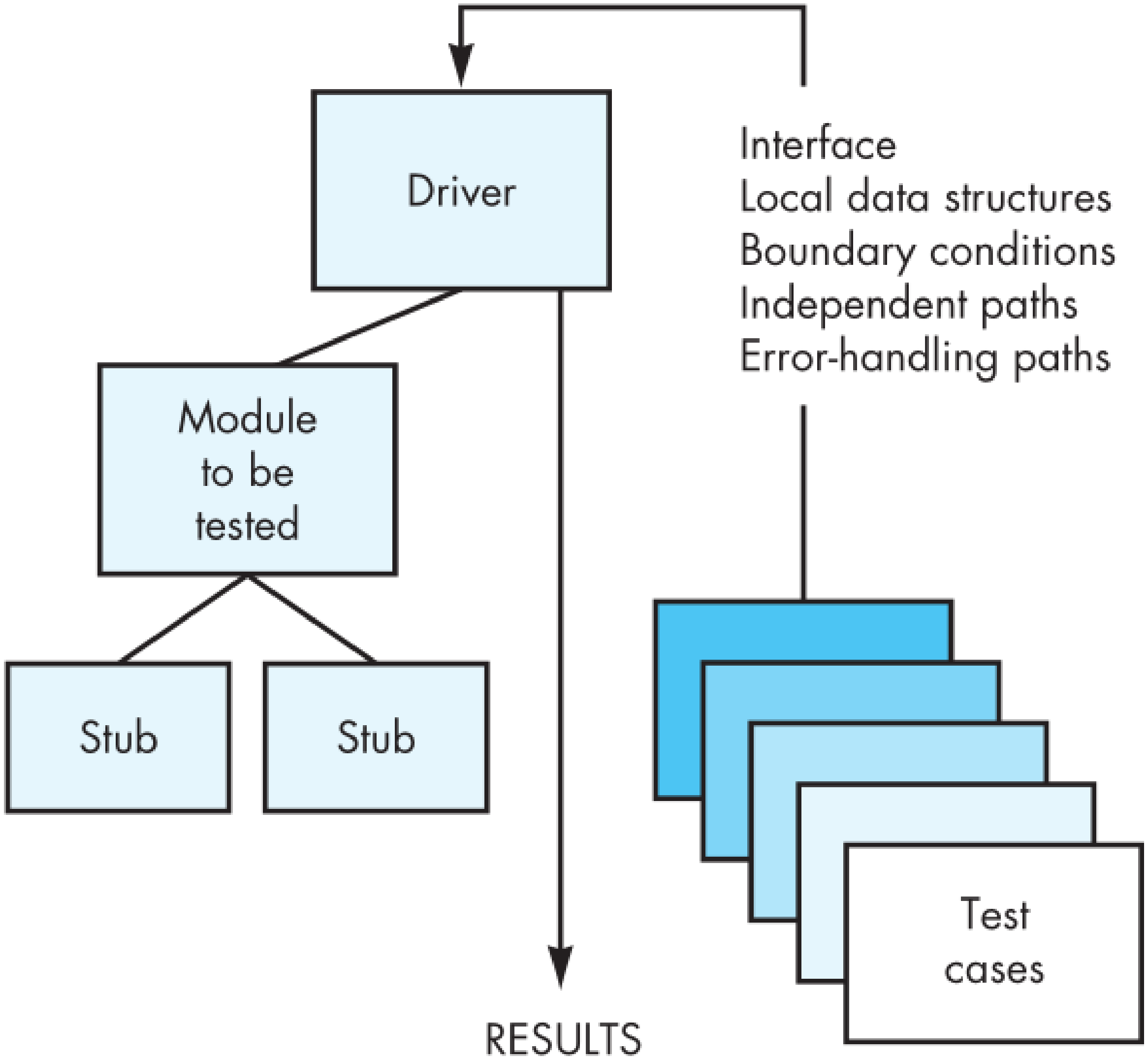


Relative costs to find and repair a defect

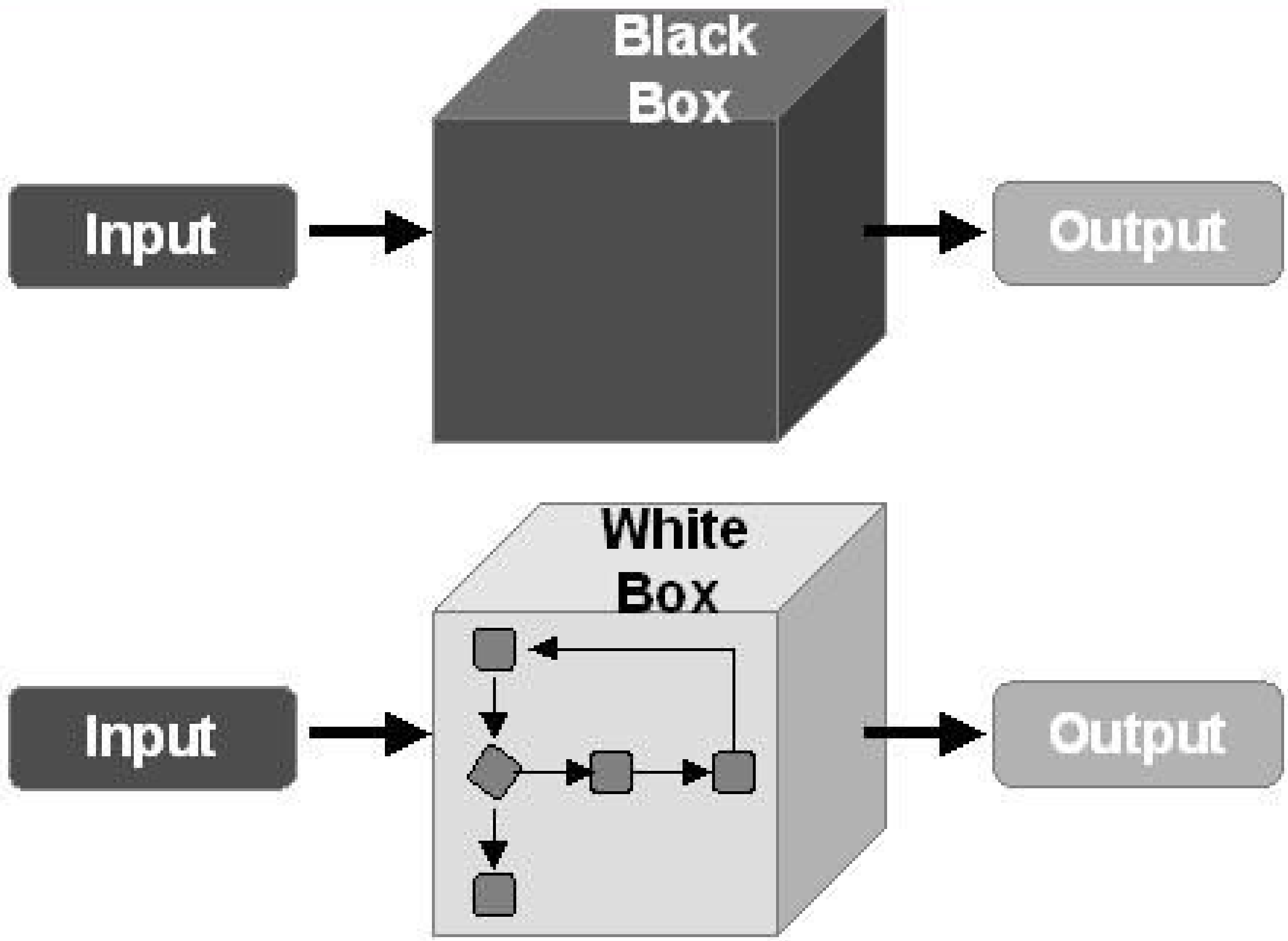
- An IBM's analysis



Unit testing

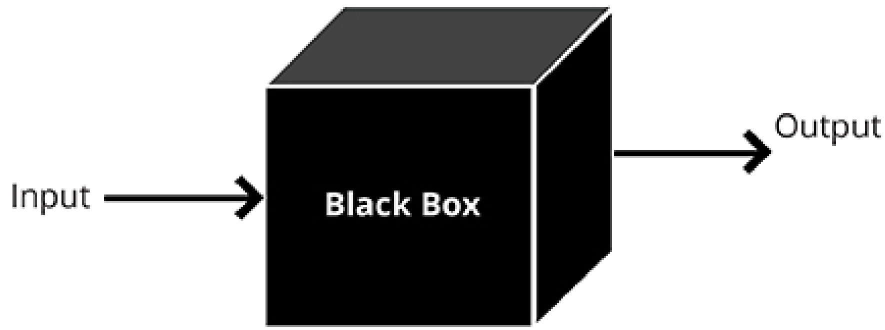


Types of testing



Blackbox testing

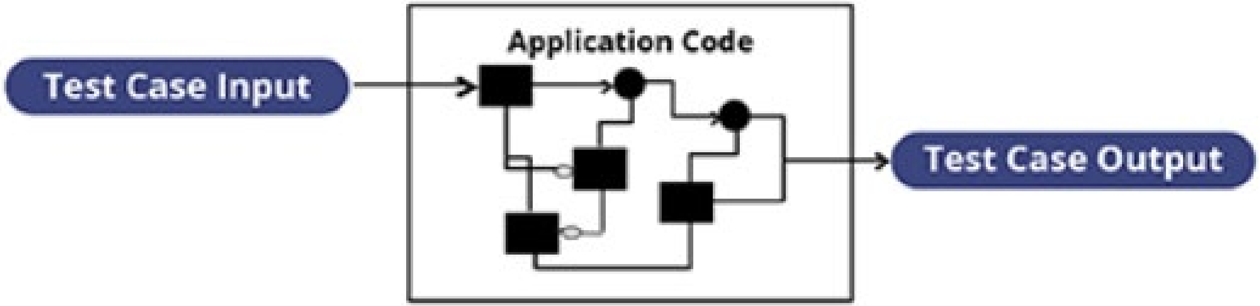
BLACK BOX TESTING APPROACH



- Equivalence partitioning (hours <0, 0<=hours<=80, hours>80)
- Boundary value analysis (hours = -1, 0, 1, 79, 80, 81)
- Decision table testing

	Conditions/ Courses of Action	Rules					
		1	2	3	4	5	6
Condition Stubs	Employee type	S	H	S	H	S	H
	Hours worked	<40	<40	40	40	>40	>40
Action Stubs	Pay base salary	X		X		X	
	Calculate hourly wage		X		X		X
	Calculate overtime						X
	Produce Absence Report		X				

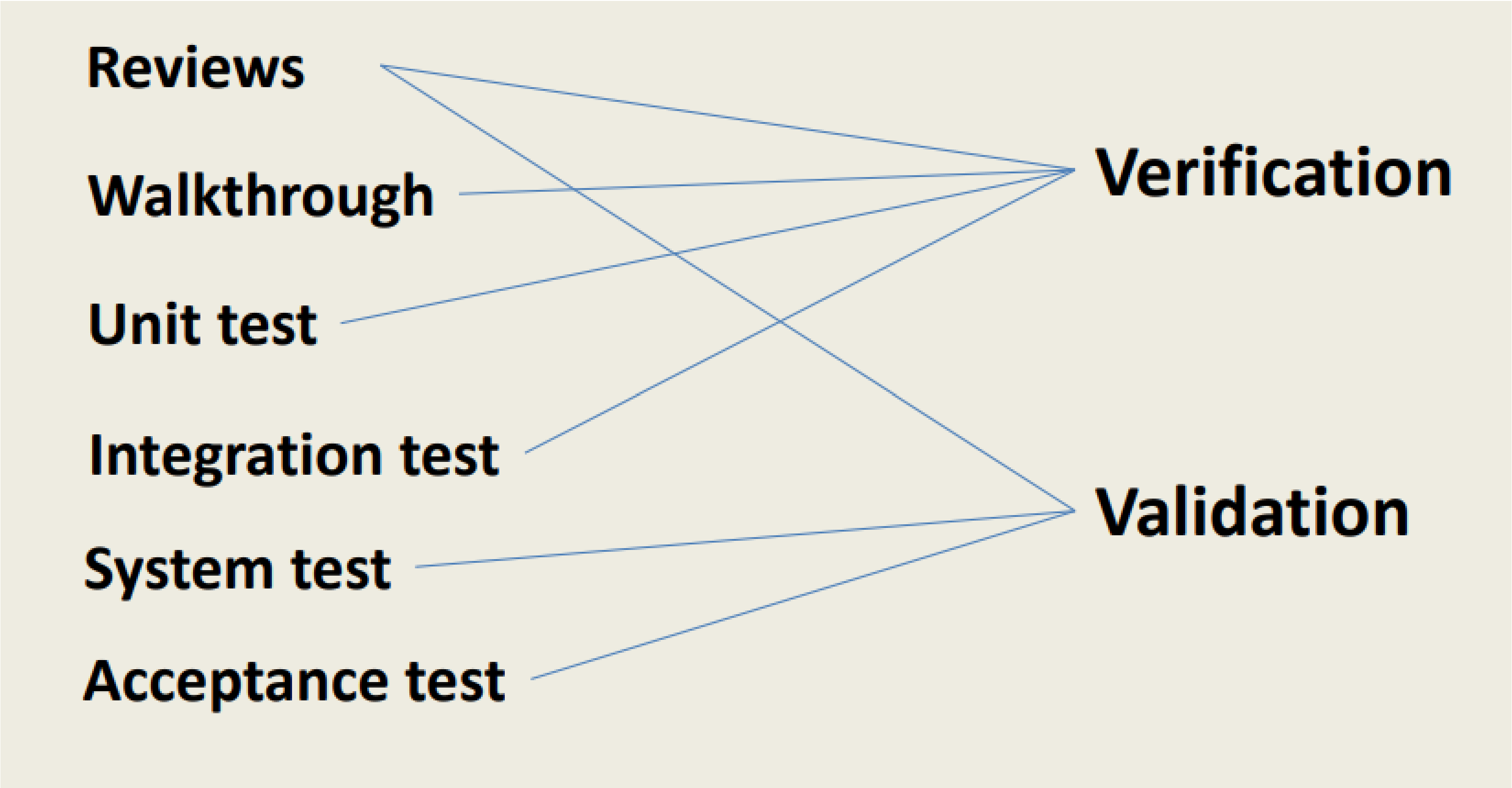
Whitebox testing



- Statement coverage
- Branch coverage
- Condition coverage
- Path coverage
- Dataflow testing

Some verification & validation techniques

- Verification: Are we building the product right?
- Validation: Are we building the right product?



Debugging

- A consequence of successful testing
- While testing is (or should be) planned and systematic, debugging is more of an art
  - Map symptoms to cause
  - Investigate suspected causes
- Fixing bugs is hard

The devil's guide to debugging

- Find the defect by guessing
  - Scatter debugging statements throughout the program
  - Try changing code until something works
  - Don't back up old versions of the code
  - Don't bother understanding what the program should do
  - Assume that most problems are trivial anyway
  - Use the most obvious fix; just fix what you see:

```
x = compute(y);
//compute() does not work for y==17, so fix it
if (y==17)
    x = 25.15
```

The debugging process

- Information collection
- Fault isolation
- Fault confirmation
- Fault correction
- Fault and correction
- Regression testing
- Documentation

Debugging: module `findLast( )`

```
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;
}
```

Short answer questions

(a) Identify the fault.

(b) If possible, identify a test case that does not execute the fault.

(c) If possible, identify a test case that executes the fault, but does not result in an error state.

(d) If possible identify a test case that results in an error, but not a failure. Hint: Don't forget about the program counter.

(e) For the given test case, identify the first error state. Be sure to describe the complete state.

(f) Fix the fault and verify that the given test no produces the expected output.

Short answer questions

(a) Identify the fault.

The for-loop should include the 0 index:

```
for (int i=x.length - 1; i >= 0; i--) {
```

findLast()

(b) If possible, identify a test case that does not execute the fault.

The null value for x will result in a NullPointerException before the loop test is evaluated, hence no execution of the fault.

Input: `x = null; y = 3`

Expected Output: NullPointerException

Actual Output: NullPointerException

findLast()

(c) If possible, identify a test case that executes the fault, but does not result in an error state.

For any input where y appears in the second or later position, there is no error.

Also, if x is empty, there is no error.

Input: x = `[2, 3, 5]; y = 3;`

Expected Output: 1

Actual Output: 1

findLast()

(d) If possible identify a test case that results in an error, but not a failure. Hint: Don't forget about the program counter.

For an input where y is not in x, the missing path (i.e., an incorrect PC on the final loop that is not taken) is an error, but there is no failure.

Input: `x = [2, 3, 5]; y = 7;`

Expected Output: -1

Actual Output: -1

findLast()

(e) For the given test case, identify the first error state. Be sure to describe the complete state.

Note that the key aspect of the error state is that the PC is outside the loop (following the false evaluation of the 0>0 test). In a correct program, the PC should be at the if-test, with index i==0.

Input: x = [2, 3, 5]; y = 2;

Expected Output: 0

Actual Output: -1

First Error State: x = [2, 3, 5]; y = 2;

i = 0;

PC = just before return -1

findLast()

(f) Fix the fault and verify that the given test no produces the expected output.

See (a)

Main spec-based testing techniques

- Boundary value testing
- Equivalence partitioning
- Decision table-based testing
- Special value, random (fuzz) testing

Example problems: triangle type

- The triangle program accepts three integers, **a**, **b**, and **c**, as input; these are taken to be the sides of a triangle
- The output of the program is the type of triangle determined by the three sides: *Equilateral*, *Isosceles*, *Scalene*, or *Not A Triangle*
- **a**, **b**, and **c** must satisfy the following conditions:

c1. $1 \leq a \leq 200$

c2. $1 \leq b \leq 200$

c3. $1 \leq c \leq 200$

c4. $a < b + c$

c5. $b < a + c$

c6. $c < a + b$

Example problems: triangle type

- If an input value fails any of conditions c1, c2, or c3, the program notes this with an output message, .e.g., "Value of b is not in the range of permitted values."
- If values of a, b, and c satisfy conditions c1, c2, and c3, one of four mutually exclusive outputs is given:
  - If any of conditions c4, c5, and c6 is not met, the program output is *Not A Triangle*
  - If all three sides are equal, the program output is *Equilateral*
  - If exactly one pair of sides is equal, the program output is *Isosceles*
  - If no pair of sides are equal, the program output is *Scalene*

Triangle type: Java solution

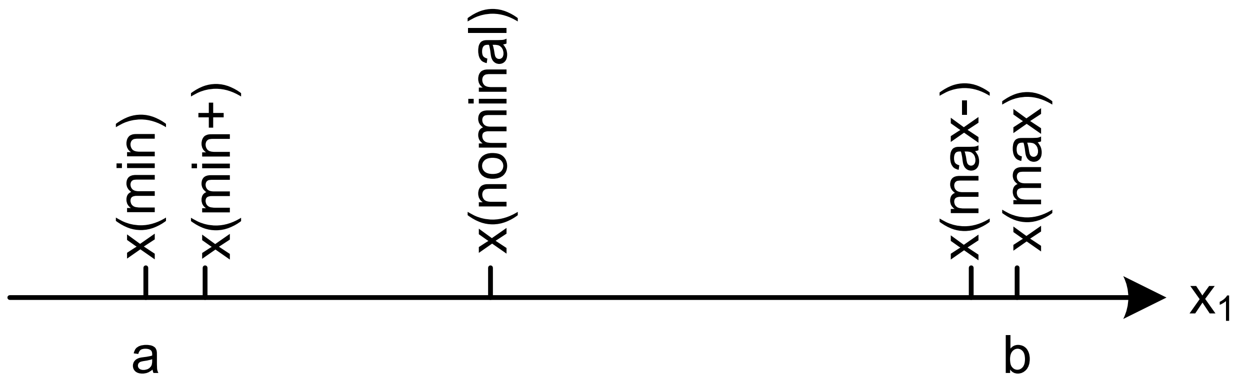
```
public static int triangle3(int a, int b, int c) {

    boolean c1, c2, c3, isATriangle;

    // Step 1: Validate Input
       c1 = (1 <= a) && (a <= 200);
       c2 = (1 <= b) && (b <= 200);
       c3 = (1 <= c) && (c <= 200);

       int triangleType = INVALID;
       if(!c1 || !c2 || !c3)
          triangleType = OUT_OF_RANGE;
       else {

       // Step 2: Is A Triangle?
          if((a < b + c) && (b < a + c) && (c < a + b))
               isATriangle = true;
            else
               isATriangle = false;

         // Step 3: Determine Triangle Type
            if(isATriangle) {
                if((a == b) && (b == c))
                  triangleType = EQUILATERAL;
                else if((a != b) && (a != c) && (b != c))
                  triangleType = SCALENE;
               else
                  triangleType = ISOSELES;
            } else
               triangleType = INVALID;
         }

         return triangleType;
      }
```

Functional testing: BV testing

- Impossible to check all input/output combinations: need to choose some
- Rationale: most errors occur at near their extremes (instead of < used <=, counters off by one)
- Two considerations apply to boundary value testing
  - are invalid values an issue?
  - can we make the "single fault assumption" of reliability theory?

How many test cases

- Assume variable x (a<= x <= b)
- Extreme values only
  - Five test cases (four boundary values, one from the middle)



- If there are n variables
  - 4n+1 test cases

Normal BV

- Test cases 3, 8, and 13 are identical and two should be deleted
- $4n + 1 = 13$

Table 5.1 Normal Boundary Value Test Cases

Case	a	b	c	*Expected Output*
1	100	100	1	Isosceles
2	100	100	2	Isosceles
3	100	100	100	Equilateral
4	100	100	199	Isosceles
5	100	100	200	Not a triangle
6	100	1	100	Isosceles
7	100	2	100	Isosceles
8	100	100	100	Equilateral
9	100	199	100	Isosceles
10	100	200	100	Not a triangle
11	1	100	100	Isosceles
12	2	100	100	Isosceles
13	100	100	100	Equilateral
14	199	100	100	Isosceles
15	200	100	100	Not a triangle

Robust BV testing

- Beyond extreme values (for robustness)
  - n variables
  - 6n + 1

Robust BV

- Six more test cases should be added (not shown in the table)

  100,100,0

  100,100,201

  100,0,100

  100,201,100

  0,100,100

  201,100,100

- Total: $6n + 1 = 19$

Table 5.1 Normal Boundary Value Test Cases

Case	a	b	c	*Expected Output*
1	100	100	1	Isosceles
2	100	100	2	Isosceles
3	100	100	100	Equilateral
4	100	100	199	Isosceles
5	100	100	200	Not a triangle
6	100	1	100	Isosceles
7	100	2	100	Isosceles
8	100	100	100	Equilateral
9	100	199	100	Isosceles
10	100	200	100	Not a triangle
11	1	100	100	Isosceles
12	2	100	100	Isosceles
13	100	100	100	Equilateral
14	199	100	100	Isosceles
15	200	100	100	Not a triangle

No single-fault assumption

- Interested in what happens when more than one variable has an extreme value
- Multiple-fault assumption
- For each variable:
  - Start with five–element set (min, min+, middle, max-1, max)
  - Consider all combinations of 5 values for all variables
    - Take the Cartesian product of these sets to generate test cases
- How many test cases?
  - *5^n*

Multiple-fault BV testing

- Also known as worst-case BV testing
- Table shows *selected* test cases
- How many tests?
  - 5^n = 5^3 = 125

Table 5.2 (Selected) Worst-Case Boundary Value Test Cases

Case	a	b	c	*Expected Output*
1	1	1	1	Equilateral
2	1	1	2	Not a triangle
3	1	1	100	Not a triangle
4	1	1	199	Not a triangle
5	1	1	200	Not a triangle
6	1	2	1	Not a triangle
7	1	2	2	Isosceles
8	1	2	100	Not a triangle
9	1	2	199	Not a triangle
10	1	2	200	Not a triangle
11	1	100	1	Not a triangle
12	1	100	2	Not a triangle
13	1	100	100	Isosceles
14	1	100	199	Not a triangle
15	1	100	200	Not a triangle
16	1	199	1	Not a triangle
17	1	199	2	Not a triangle
18	1	199	100	Not a triangle
19	1	199	199	Isosceles
20	1	199	200	Not a triangle
21	1	200	1	Not a triangle
22	1	200	2	Not a triangle
23	1	200	100	Not a triangle
24	1	200	199	Not a triangle
25	1	200	200	Isosceles

Multiple-fault BV testing

- Also known as worst-case BV testing
- Table shows *selected* test cases
- How many tests?
  - $5^n = 5^3 = 125$

Worst Case Test Cases (60 of 125)

Case	a	b	c	Expected Output	Case	a	b	c	Expected Output
1	1	1	1	Equilateral	31	2	2	1	Isosceles
2	1	1	2	Not a Triangle	32	2	2	2	Equilateral
3	1	1	100	Not a Triangle	33	2	2	100	Not a Triangle
4	1	1	199	Not a Triangle	34	2	2	199	Not a Triangle
5	1	1	200	Not a Triangle	35	2	2	200	Not a Triangle
6	1	2	1	Not a Triangle	36	2	100	1	Not a Triangle
7	1	2	2	Isosceles	37	2	100	2	Not a Triangle
8	1	2	100	Not a Triangle	38	2	100	100	Isosceles
9	1	2	199	Not a Triangle	39	2	100	199	Not a Triangle
10	1	2	200	Not a Triangle	40	2	100	200	Not a Triangle
11	1	100	1	Not a Triangle	41	2	199	1	Not a Triangle
12	1	100	2	Not a Triangle	42	2	199	2	Not a Triangle
13	1	100	100	Isosceles	43	2	199	100	Not a Triangle
14	1	100	199	Not a Triangle	44	2	199	199	Isosceles
15	1	100	200	Not a Triangle	45	2	199	200	Scalene
16	1	199	1	Not a Triangle	46	2	200	1	Not a Triangle
17	1	199	2	Not a Triangle	47	2	200	2	Not a Triangle
18	1	199	100	Not a Triangle	48	2	200	100	Not a Triangle
19	1	199	199	Isosceles	49	2	200	199	Scalene
20	1	199	200	Not a Triangle	50	2	200	200	Isosceles
21	1	200	1	Not a Triangle	51	100	1	1	Not a Triangle
22	1	200	2	Not a Triangle	52	100	1	2	Not a Triangle
23	1	200	100	Not a Triangle	53	100	1	100	Isosceles
24	1	200	199	Not a Triangle	54	100	1	199	Not a Triangle
25	1	200	200	Isosceles	55	100	1	200	Not a Triangle
26	2	1	1	Not a Triangle	56	100	2	1	Not a Triangle
27	2	1	2	Isosceles	57	100	2	2	Not a Triangle
28	2	1	100	Not a Triangle	58	100	2	100	Isosceles
29	2	1	199	Not a Triangle	59	100	2	199	Not a Triangle
30	2	1	200	Not a Triangle	60	100	2	200	Not a Triangle

Robust and multiple-fault assumption

- Combination of worst-case and robustness BV testing
  - AKA worst-case robust BV testing
- All combinations of 7 values for all variables
  - Robust worse case BV test cases: 7^n
  - For the triangle problem: 343 test cases

(0,1,2,100,199,200,201) x (0,1,2,100,199,200,201) x (0,1,2,100,199,200,201) = 343

Summary of BV testing

- Relatively straightforward test-case generation
- Easy to do/automate
- Appropriate for calculation-intensive applications with variables that represent physical quantities (e.g., have units, such as meters, degrees, kilograms)
- Potential redundancies

Equivalent class partitioning (rationale)

- All four variations of boundary value testing are vulnerable to
  - Gaps of untested functionality, and
  - Redundancy (that results in extra testing effort)
- The mathematical notion of an equivalence class can potentially help this because
  - Members of a class should be "treated the same" by a program
  - Equivalence classes form a partition of the input space
- A partition deals explicitly with
  - Redundancy (elements of a partition are disjoint)
  - Gaps (the union of all partition elements is the full input space)
- Equivalence class testing provides a strategy to resolve the above

Basics of EC testing

- Consider function $F(x_1, x_2)$:
  - valid values of x_1: $a \leq x_1 \leq b$
  - invalid values of x_1: $x_1 < a$, $b < x_1$
  - valid values of x_2: $c \leq x_2 \leq d$
  - invalid values of x_2: $x_2 < c$, $d < x_2$
- Process
  - Identify valid and invalid values of all variables
  - Test one invalid variable at a time
    - * Note this makes the single fault assumption

Basics of EC testing

- Consider function $F(x_1, x_2)$:
  - valid values of x_1: $a \le x_1 \le b$
  - invalid values of x_1: $x_1 < a$, $b < x_1$
  - valid values of x_2: $c \le x_2 \le d$
  - invalid values of x_2: $x_2 < c$, $d < x_2$

- Note: x_1 could be the number of hours work and for a paycheck we may have to define the following equivalent classes:

  0-40 (to calculate a regular pay rate)

  41-50 (to calculate 1.5x pay rate)

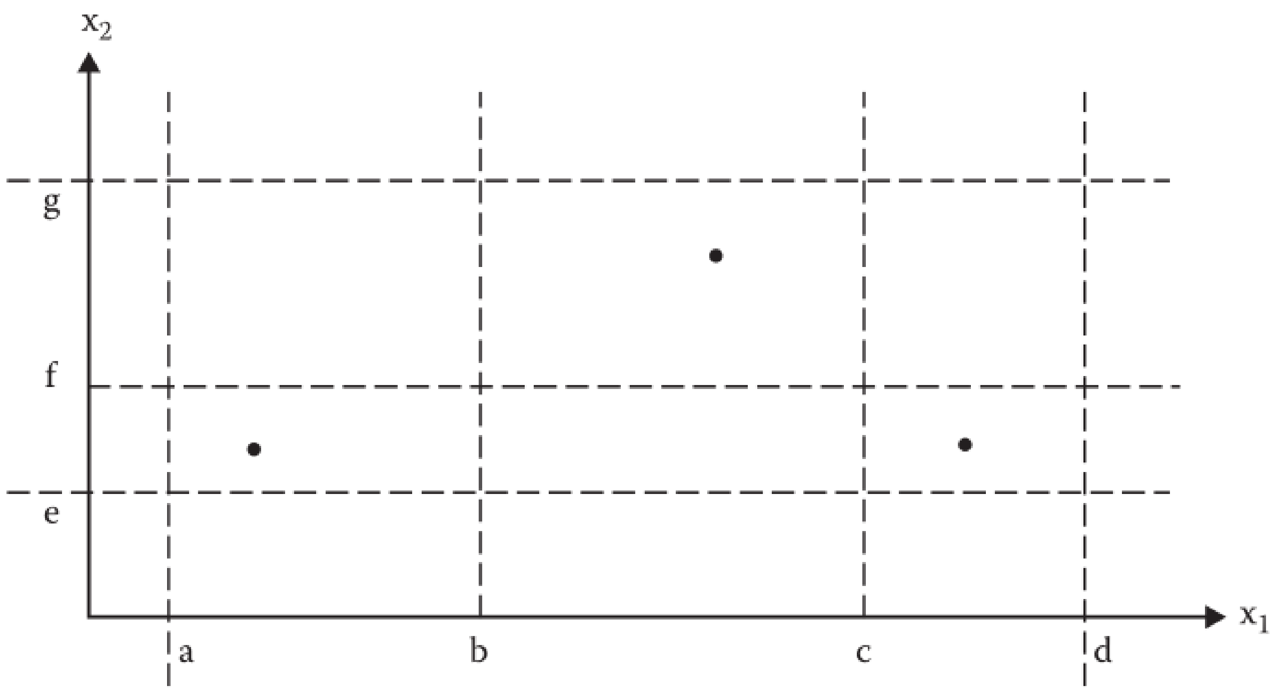
  51-60 (to calculate 2x pay rate)

Forms of EC

- Normal: classes of valid values of inputs
- Robust: classes of valid and invalid values of inputs
- Single fault assumption: one from each class
- Multiple fault assumption: one from each class in Cartesian product
- We compare these for a function of two variables, $F(x_1, x_2)$
- Extension to problems with 3 or more variables should be obvious

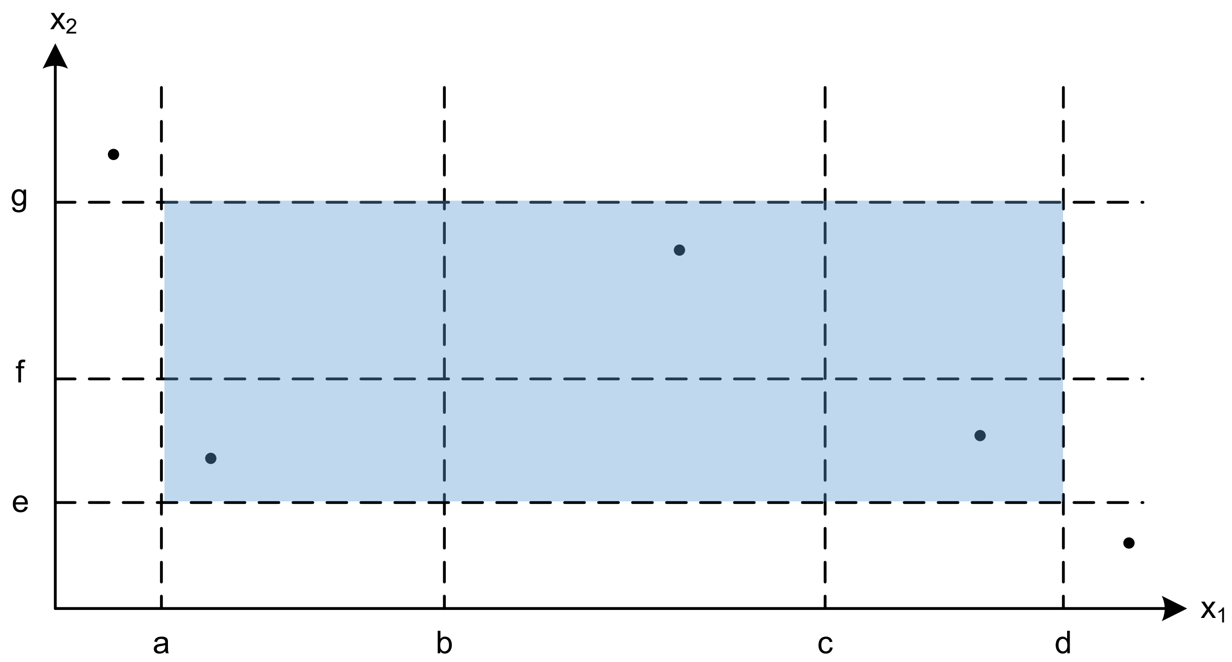
Normal EC test cases

- Valid ECs
  - We are not considering robust testing yet (invalid values)
  - For x1 and x2, we need three test cases
    - Each dot represent a test: a value for x1 and a value for x2



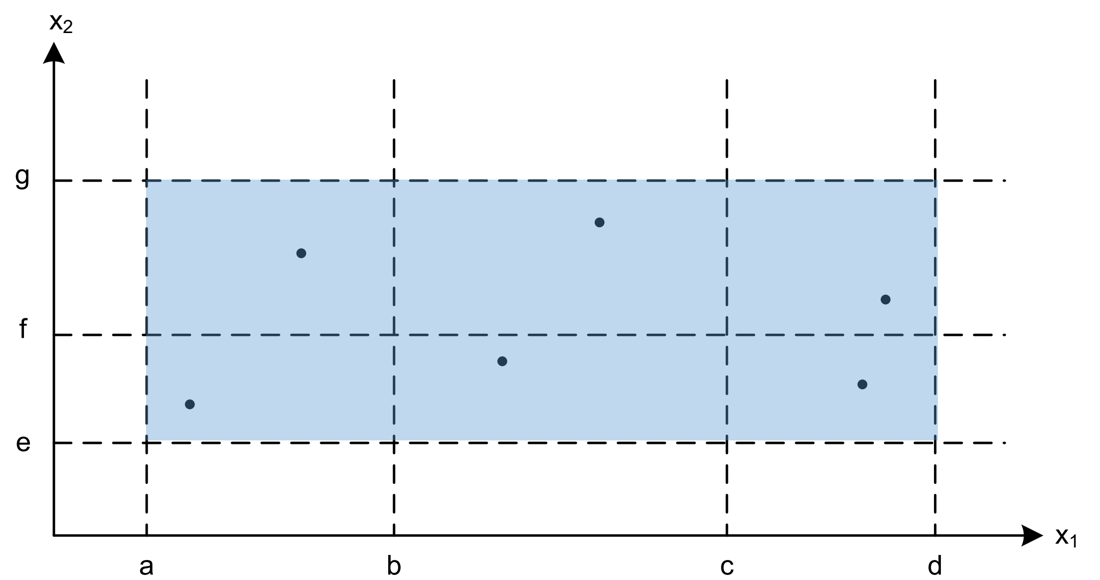
Robust EC testing process

- Identify equivalence classes of valid and invalid values
- Test cases have all valid values except one invalid value
  - Detects faults due to invalid values of a single variable



Normal EC testing but no single-fault

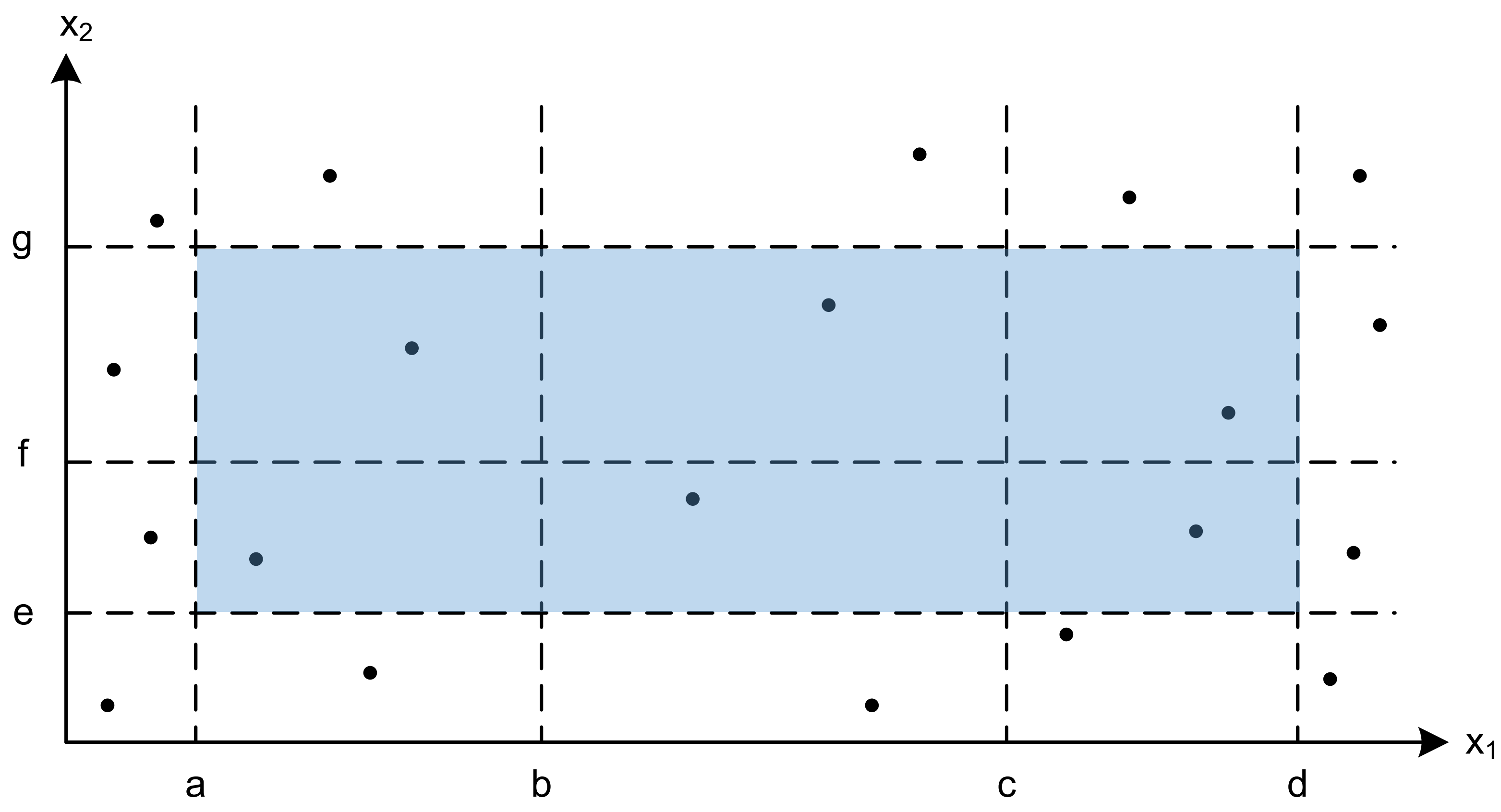
- No longer assume single-fault (AKA strong normal EC)
- Identify equivalence classes of valid values
- Test cases from Cartesian product of valid values
- Detects faults due to interactions with valid values of any number of variables



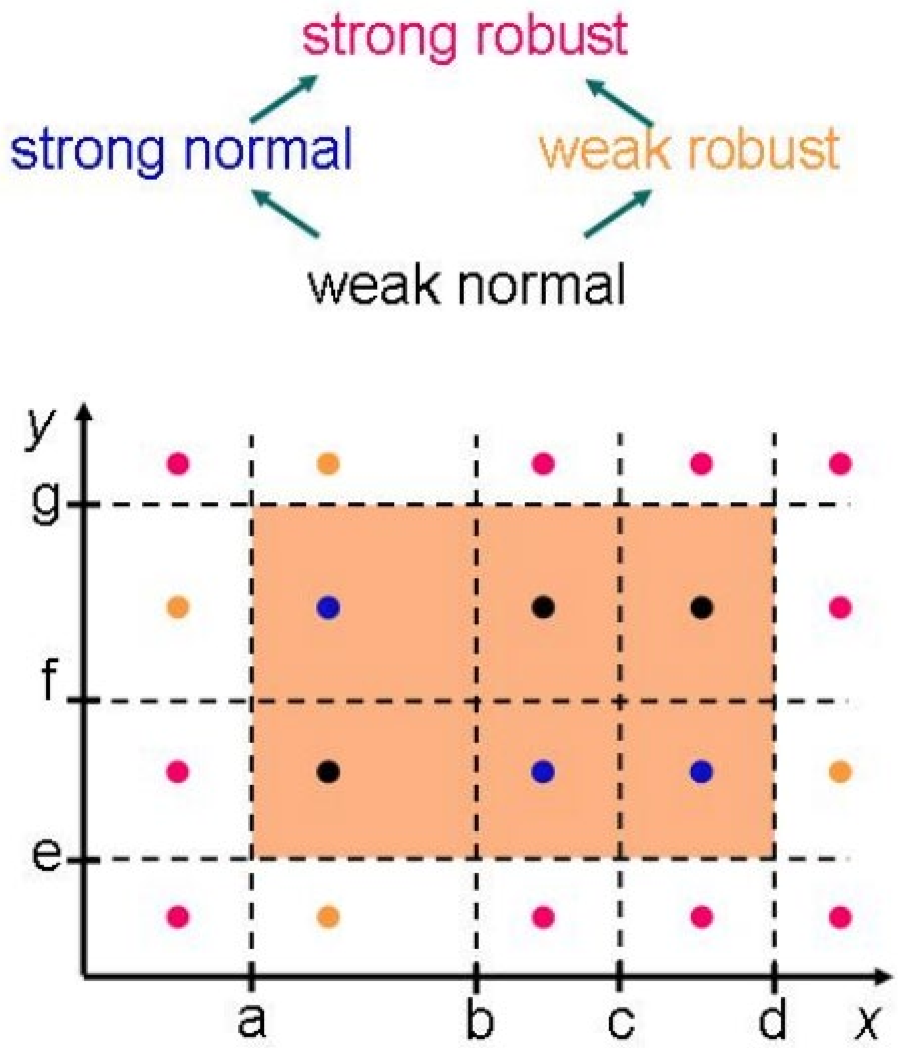
Robust EC testing, no single-fault

- Consider invalid input; do not assume single fault
  - AKA strong robust EC testing
- Identify equivalence classes of valid and invalid values
- Test cases from Cartesian product of all classes
- Detects faults due to interactions with any values of any number of variables
- Most rigorous form of EC testing

Strong robust EC testing



A brief comparison



Strong robust EC testing

- "Corner" of the cube in three-space of the additional strong robust equivalence class test cases

Test Case	a	b	c	*Expected Output*
SR1	–1	5	5	Value of a is not in the range of permitted values
SR2	5	–1	5	Value of b is not in the range of permitted values
SR3	5	5	–1	Value of c is not in the range of permitted values
SR4	–1	–1	5	Values of a, b are not in the range of permitted values
SR5	5	–1	–1	Values of b, c are not in the range of permitted values
SR6	–1	5	–1	Values of a, c are not in the range of permitted values
SR7	–1	–1	–1	Values of a, b, c are not in the range of permitted values

Mortgage example

Write a program that takes three inputs: gender (Boolean), age(18-55), salary (0-10000) and output the total mortgage for one person

Mortgage = salary * factor

where factor is given by the following table:

Category	**Male**	**Female**
Young	(18-35 years) 75	(18-30 years) 70
Middle	(36-45 years) 55	(31-40 years) 50
Old	(46-55 years) 30	(41-50 years) 35

From: P.C. Jorgensen. *Software Testing: A Craftsman's Approach*.

Code-based testing

- Continued on another slide set