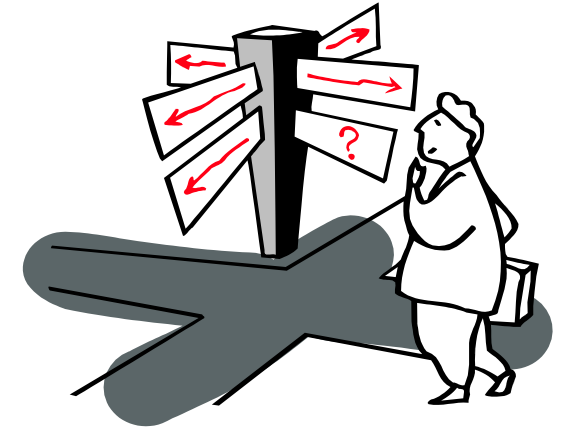


# CHAPTER 5 – Testing

- Introduction
  - + When, Why, What & Who?
    - The V-Model
  - + What is "Correct"?
  - + Terminology
- Testing Techniques
  - + White Box
    - basis path, conditions, loops
  - + Coverage
    - Code Coverage
    - MC/DC Coverage
    - Mutation Coverage
  - + Black Box
    - equivalence partitioning
  - + *Fuzz Testing*



- Testing Strategies
  - + Unit & Integration Testing
  - + Regression Testing
  - + Acceptance Testing
  - + More Testing Strategies
- Miscellaneous
  - + When to Stop?
  - + Tool Support
- Agile Testing (DevOps)
  - + Flipping the V
  - + 4-Quadrants
  - + FIT Tables
- Conclusion
  - + More Good Reasons

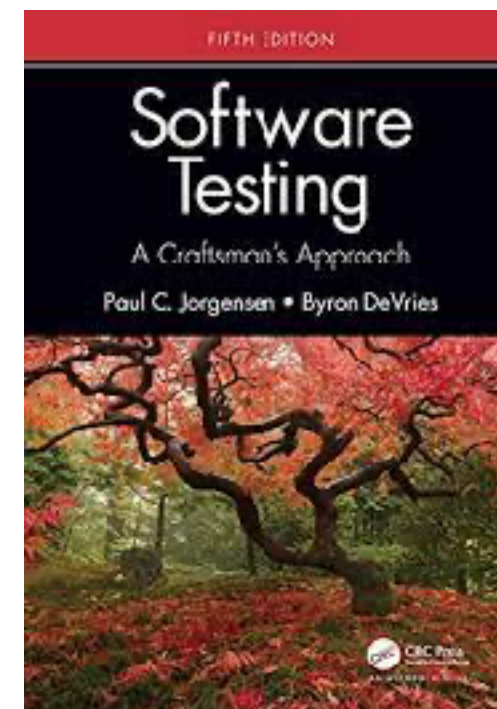
# Literature

- Books

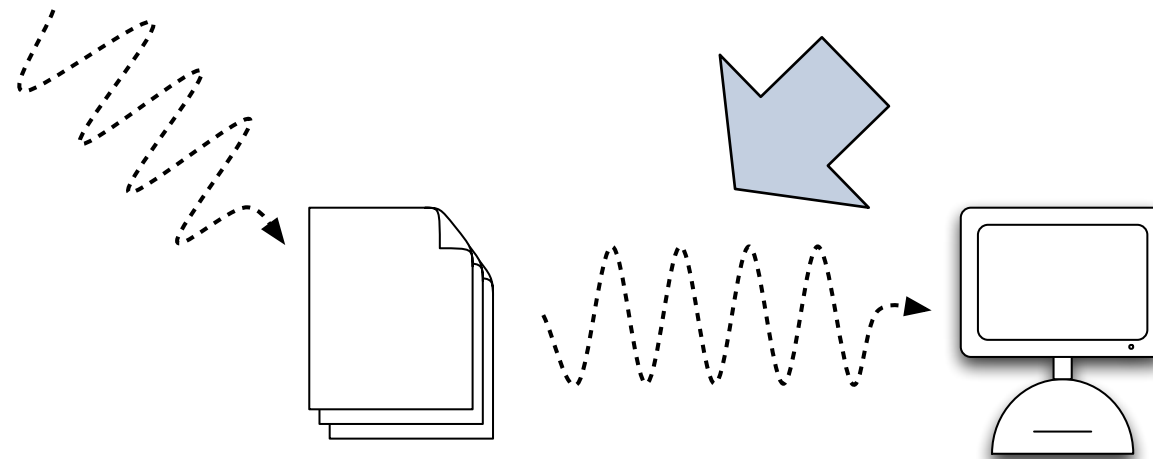
- + [Ghez02] Chapter on “Software Verification” is quite good with plenty of examples of the need for complementary testing techniques.  
Terminology used here differs from [Pres00] and [Somm05]
- + [Pres00] Chapter on “Software Testing Techniques” is very good with lots of concrete examples of the different techniques.
- + [Somm05] Chapter on “Verification and Validation” places Testing in a broader context.

- Specific Books

- + [Jorg21] Software Testing: A Craftsman’s Approach (5th edition)
  - Master course on Software Testing



# When to Test?



**Mistakes** are possible (likely!?)

- while transforming requirements into a system
- while system is changed during maintenance

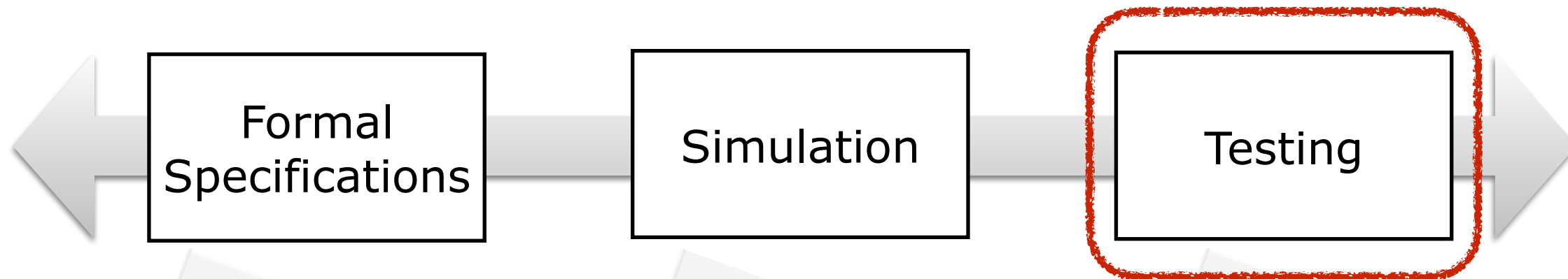
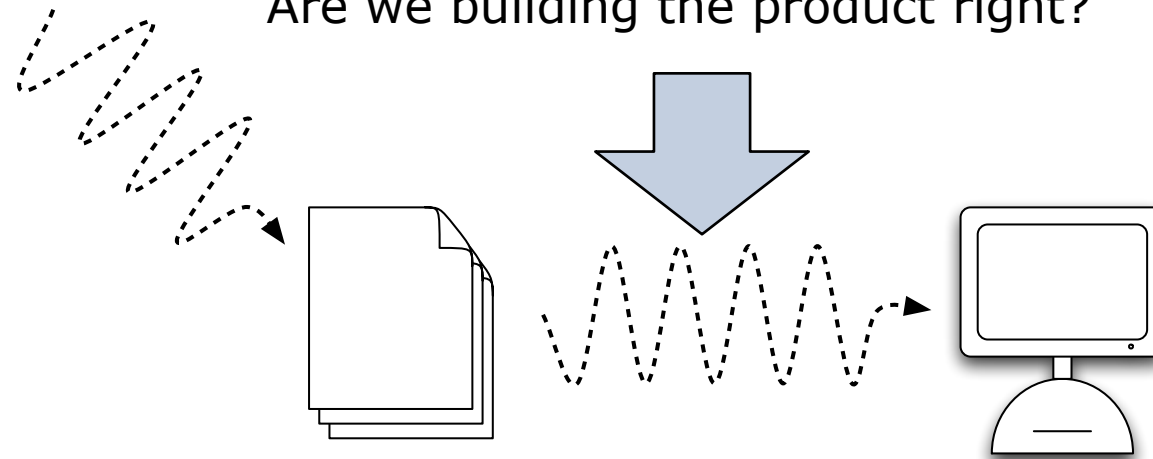
Correctness

- Are we building the right product? = VALIDATION
- Are we building the product right? = VERIFICATION

# The Verification Landscape

**\*\*New slide\*\***

Are we building the product right?



Specification and Verification  
6 ECTS-credits 1E SEM  
Lecturer(s): Guillermo Alberto Perez

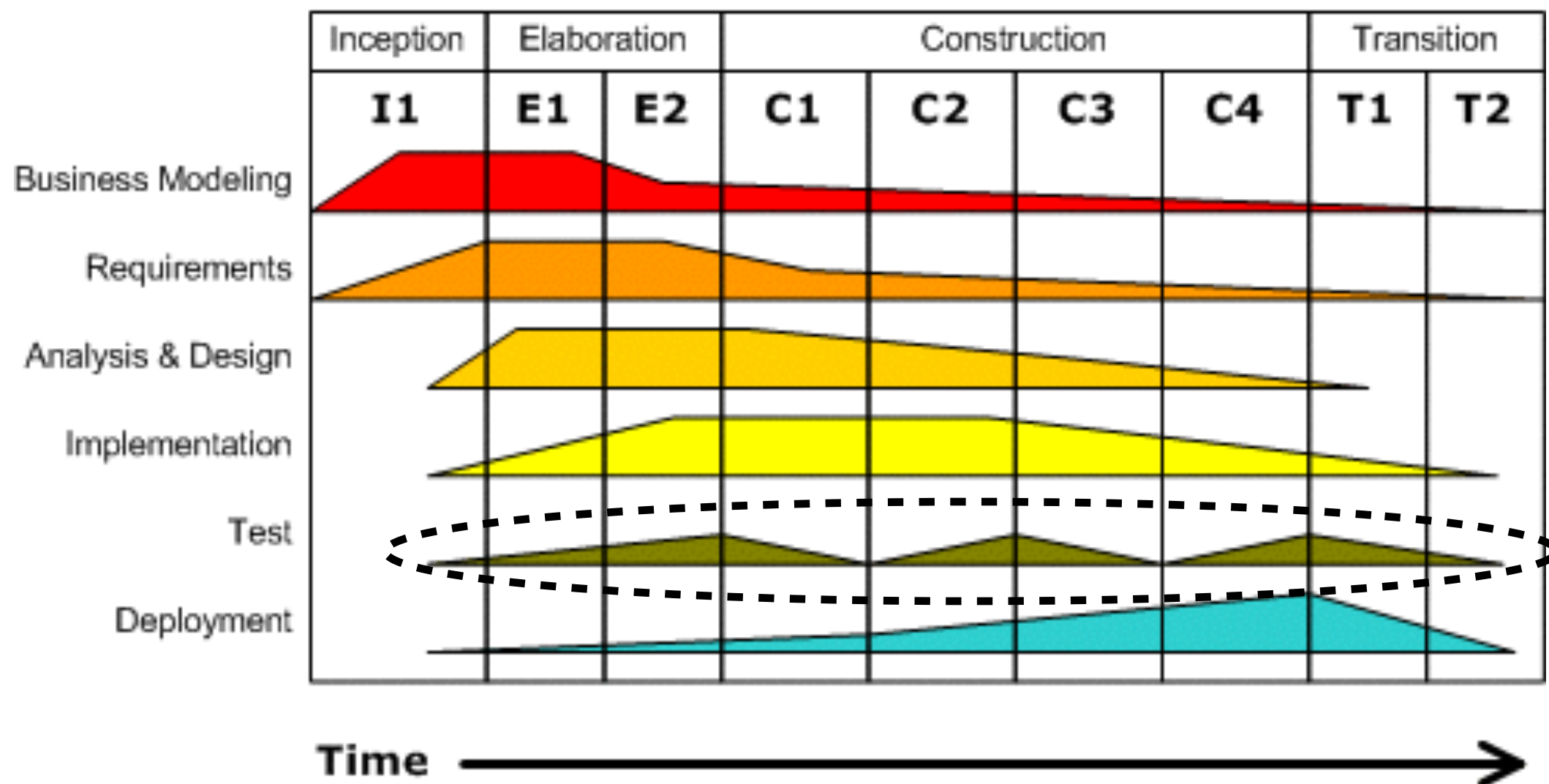
Modelling of Software-intensive Systems  
6 ECTS-credits 1E SEM  
Lecturer(s): Hans Vangheluwe

Software Testing  
6 ECTS-credits 2E SEM  
Lecturer(s): Serge Demeyer

# When to Test? The Unified Process

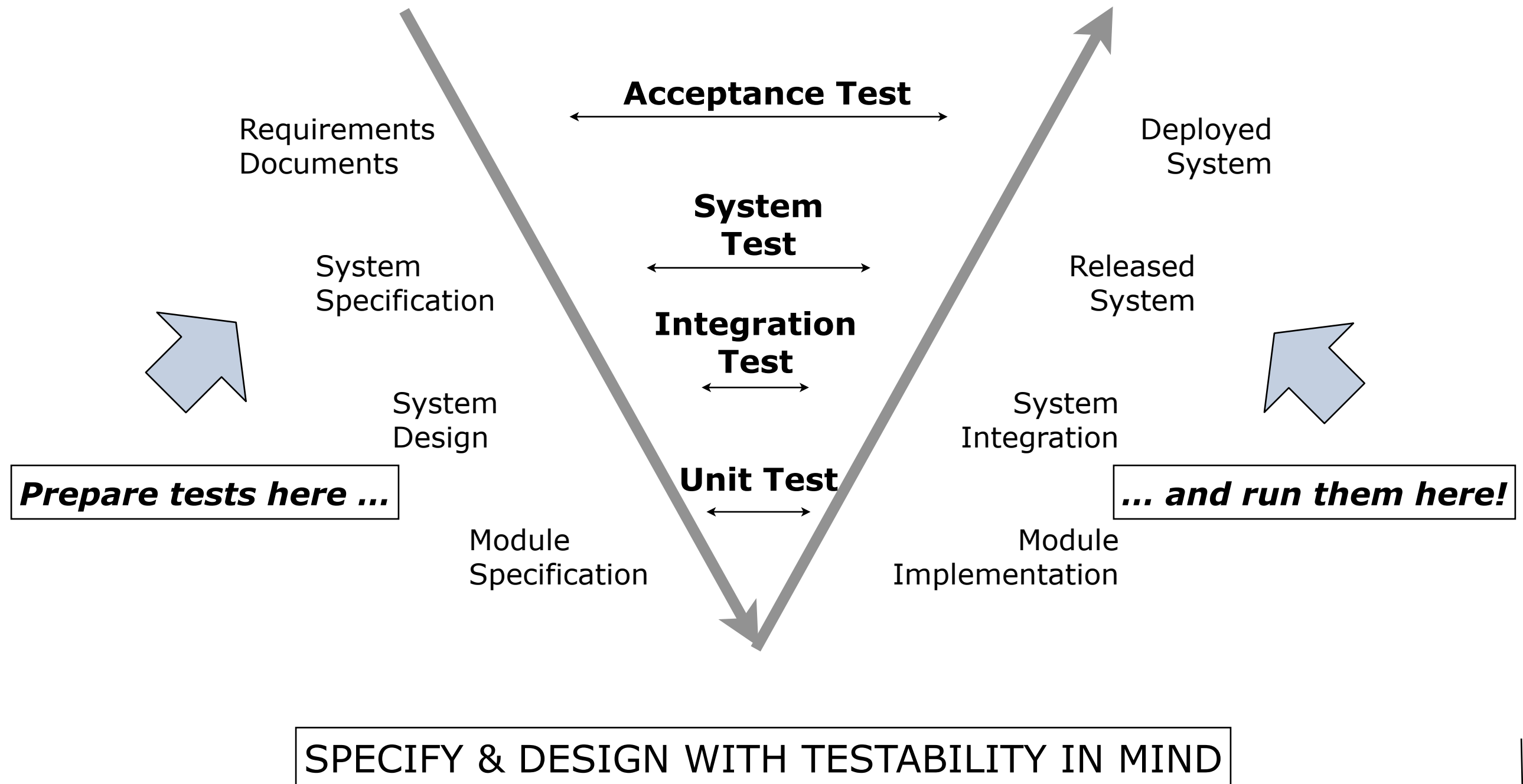
Testing is a *risk reduction* activity

- start as early as possible to assess & reduce risk towards the schedule
- repeat towards the end to assess & reduce risk towards reliability



© DutchGuilder Wikipedia

# When to Test? The V-model



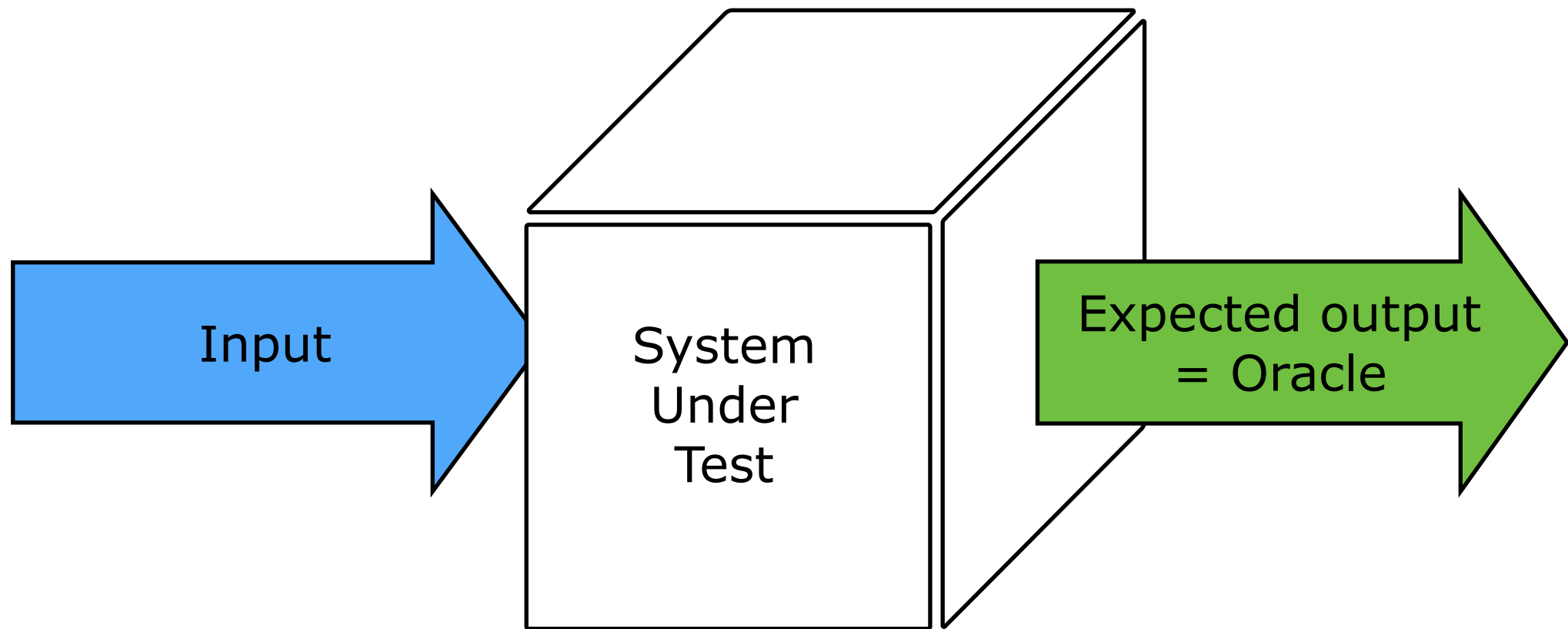
# Why to Test?

Program testing can be used to show the presence of defects, but never their absence.  
(E. W. Dijkstra)

- Perfect Excuse
  - + We should not invest in testing: our system will contain defects anyway
- Counter Arguments
  - + The more you test, the less likely such defects will cause harm
  - + The more you test, the more *confidence* you will have in the system
- Testing = Risk Management
  - + Testing is a risk *reduction* activity!
    - Result of testing is a risk report to project management  
(Can we ship this product in good confidence?)
      - \* Go / no-go decision

# What is Testing? (1/3)

**\*\*New slide\*\***



Software Testing is the process of executing a program or system with the *intent* of finding errors.

(Myers, Glenford J., The art of software testing. Wiley, 1979)

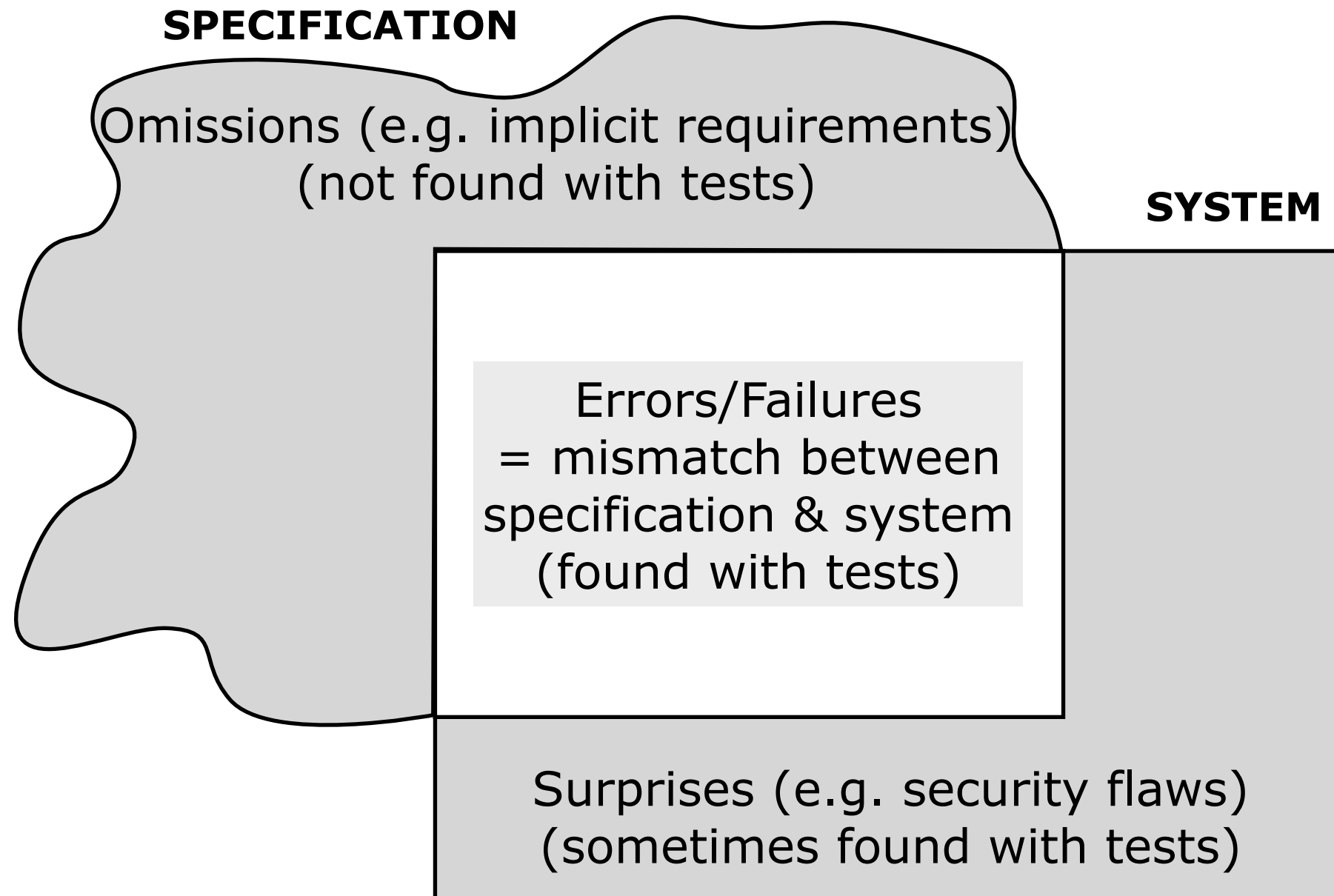


# What is Testing? (2/3)

- Testing should
  - + *verify* the requirements (Are we building the product right?)
  - + NOT *validate* the requirements (Are we building the right product?)
- Definitions
  - + Testing
    - Testing is the activity of executing a program with the intent of finding a defect
      - > A successful test is one that finds defects!
  - + Testing Techniques
    - Techniques with a high probability of finding an as yet undiscovered mistake
      - > Criterion: *Coverage* of the code/requirements/model/risks/...
  - + Testing Strategies
    - Tell you *when* you should perform *which* testing technique
      - > Criterion: *Confidence* that you can safely proceed
      - > Next activity = other testing until deployment

**REMEMBER: Testing is a risk *reduction* activity!**

# What is Testing? (3/3)



# Who should Test?

- + Programming is a constructive activity:
  - try to make things *work*
- + Testing is a destructive activity:
  - try to make things *fail*

Programmers are not necessarily the best testers!

- In practice
  - + Testing is part of quality assurance
    - done by developers when finishing a component (unit tests)
    - done by a specialized test team when finishing a subsystem (integration tests / system tests / acceptance tests)

# Unit tests ... *not* sufficient

- Interesting Tweet:  
All unit tests are passing

<https://twitter.com/olafurw/status/1578704185809244160?s=11&t=mdYnxnMXgxYBEhH7anVCgQ>



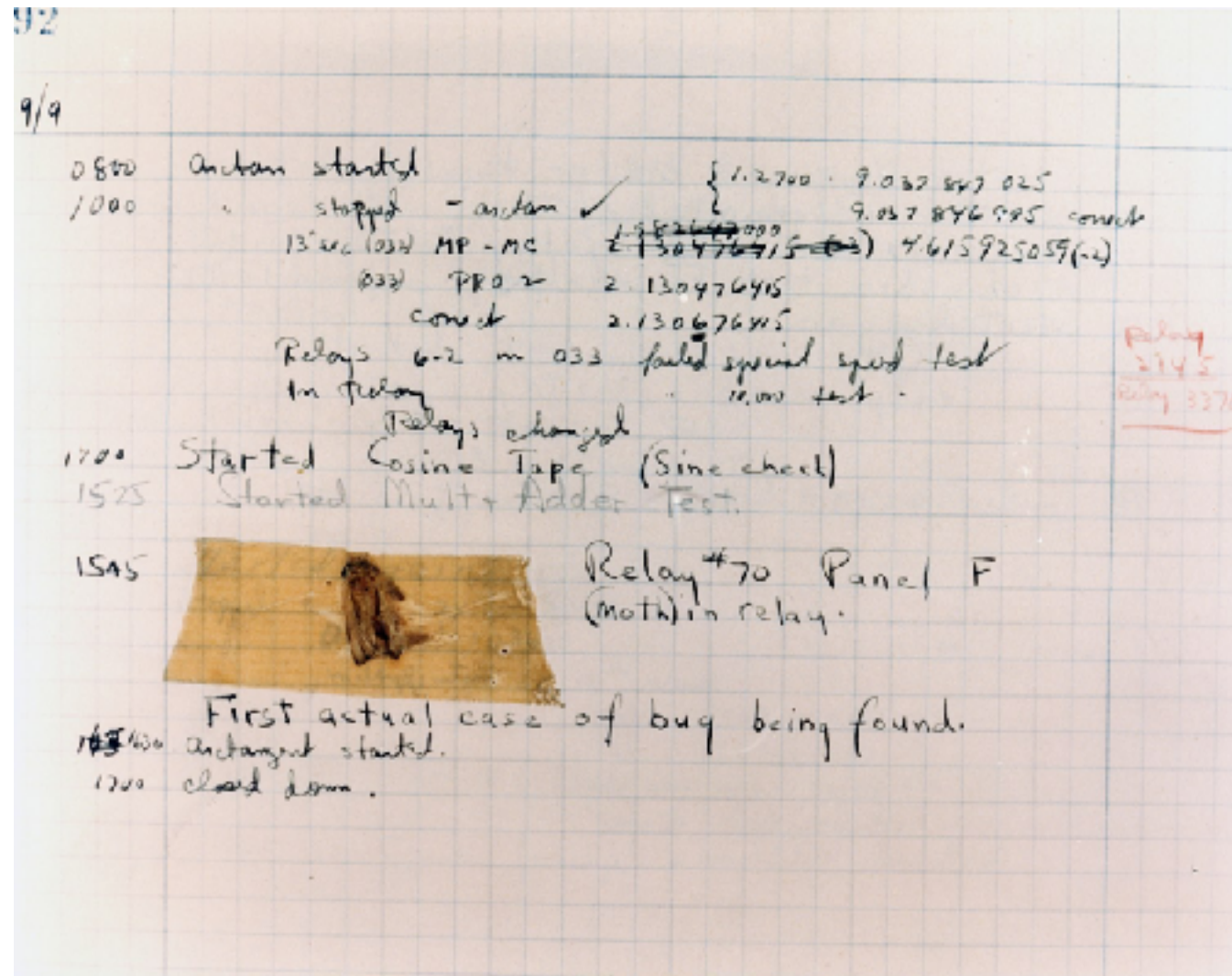
# What is “Correct”?

See [Ghez02] — Representative Qualities

- Correctness
  - + A system is correct if it behaves according to its specification
    - > An *absolute* property  
(i.e., a system cannot be “almost correct”)
    - > ... in theory and practice undecidable
- Reliability
  - + The user may rely on the system behaving properly
  - + The probability that the system will operate as expected over a specified interval
    - > A *relative* property  
(a system has a mean time between failure of 3 weeks)
- Robustness
  - + A system is robust if it behaves reasonably even in circumstances that were not specified
    - > A *vague* property (once you specify the abnormal circumstances they become part of the requirements)

# Terminology (1/3)

- Avoid the term "Bug" (\*)
  - + Implies mistakes creeping into the software from the outside
  - + imprecise because mixes various "mistakes"



📄 [https://commons.wikimedia.org/wiki/File:First\\_Computer\\_Bug,\\_1945.jpg](https://commons.wikimedia.org/wiki/File:First_Computer_Bug,_1945.jpg)

# Terminology (2/3)

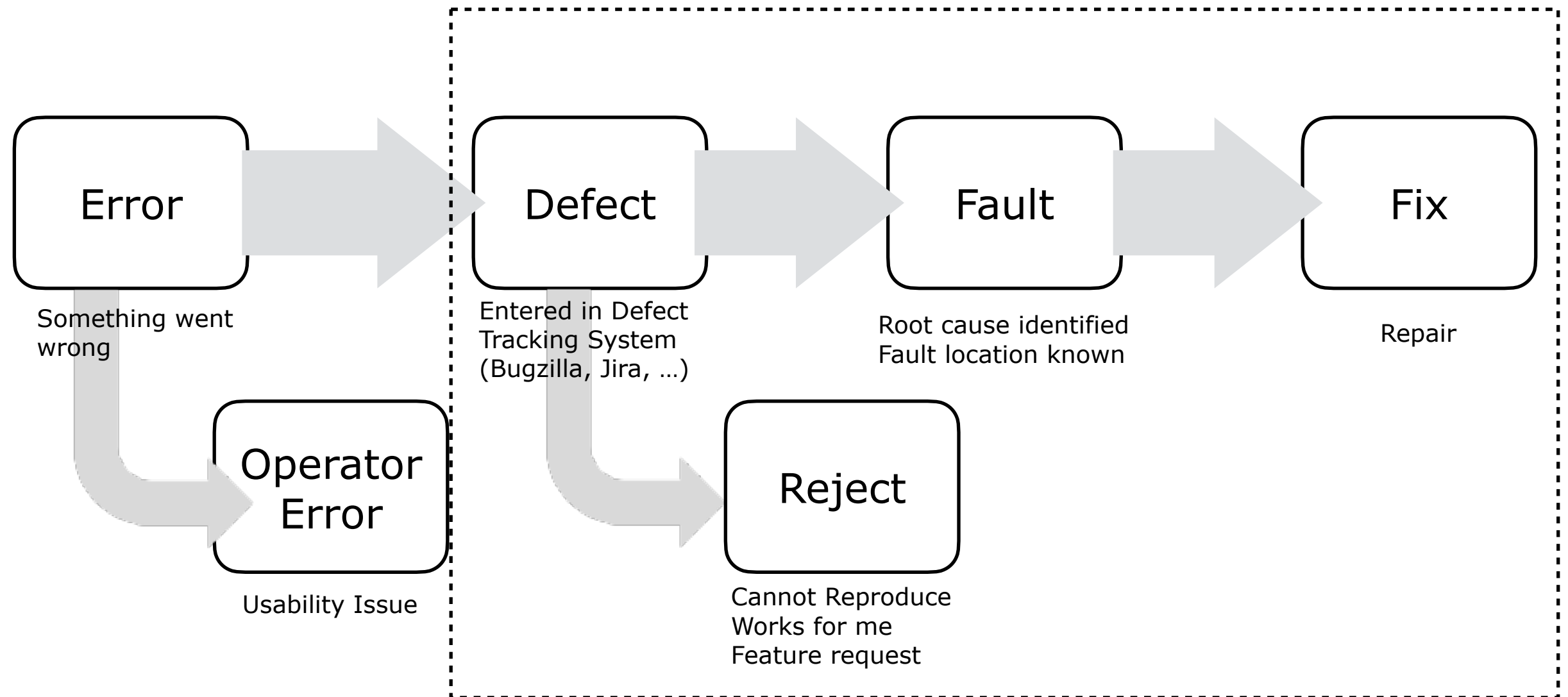
To be more precise (Terminology not standard!) : IEEE Glossary / ISTQB

- Defect / Fault (NL = DEFECT, GEBREK, NALATIGHEID)
  - + A design or coding mistake that may cause abnormal behaviour
    - abnormal behaviour = deviations from specification (incl. surprises!)
  - + Faults by *omission*: something is missing in the design, model, code, ...
  - + Faults by *commission*: incorrect entry in design, model, code, ...
- Failure (NL = MISLUKKING, FALING)
  - + A deviation between the specification and the running system
  - + A manifestation of a defect during system execution
  - + Inability to perform required function within specified limits
- Error (NL = FOUT)
  - + The input that causes a failure
    - Transient occurs only with certain input combination
    - Permanent occurs with all inputs of a given class



# Bug Tracking Workflow

Subtle deviations of terminology





# Terminology (3/3)

- Component (Component under Test)
  - + part of the system that can be isolated for testing
    - an object, a group of objects, one or more subsystems
- Test Case
  - + set of inputs and expected results that exercise a component with the purpose of causing failures
    - predicate that answers "true" when the component answers with the expected results for the given input and "false" otherwise
      - > "expected results" includes exceptions, error codes,...
- Test Stub
  - + partial implementation of components on which the tested component depends
    - dummy code providing necessary input values and behaviour
- Test Driver
  - + partial implementation of a component that depends on the tested component
    - a "main()" function that executes a number of test cases
- Test Fixture
  - + fixed state of software under test, baseline for running test
    - all that is needed to set-up the appropriate test context

# gTest Example: findLast

## Find.cpp

```
#include <vector>

int findLast(std::vector<int> x, int y) {
    if (x.size() == 0)
        return -1;
    for (int i = x.size() - 1; i >= 0; i--)
        if (x[i] == y)
            return i;
    return -1;
}
```

## Tests.cpp

```
#include <vector>
#include <gtest/gtest.h>

#include "find.cpp"

TEST(FindLastTests, noOccurrence) {
    EXPECT_EQ(-1, findLast({1, 2, 42, 42, 63}, 99));
}

TEST(FindLastTests, doubleOccurrence) {
    EXPECT_EQ(3, findLast({1, 2, 42, 42, 63}, 42));
}

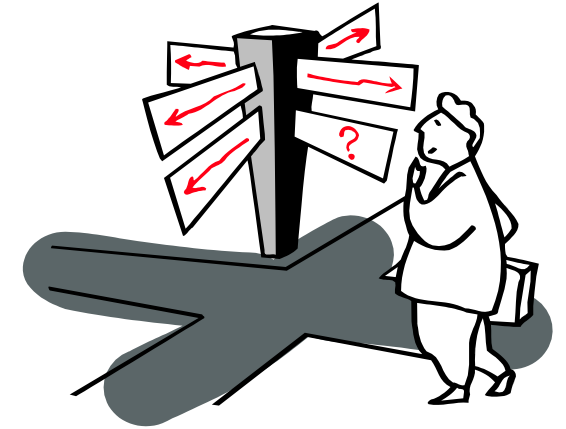
TEST(FindLastTests, emptyVector) {
    EXPECT_EQ(-1, findLast({}, 3));
}
```



Are these tests sufficiently strong?  
(Discuss with your neighbour)

# CHAPTER 5 – Testing

- Introduction
  - + When, Why, What & Who?
    - The V-Model
  - + What is "Correct"?
  - + Terminology
- Testing Techniques
  - + White Box
    - basis path, conditions, loops
  - + Coverage
    - Code Coverage
    - MC/DC Coverage
    - Mutation Coverage
  - + Black Box
    - equivalence partitioning
  - + Fuzz Testing

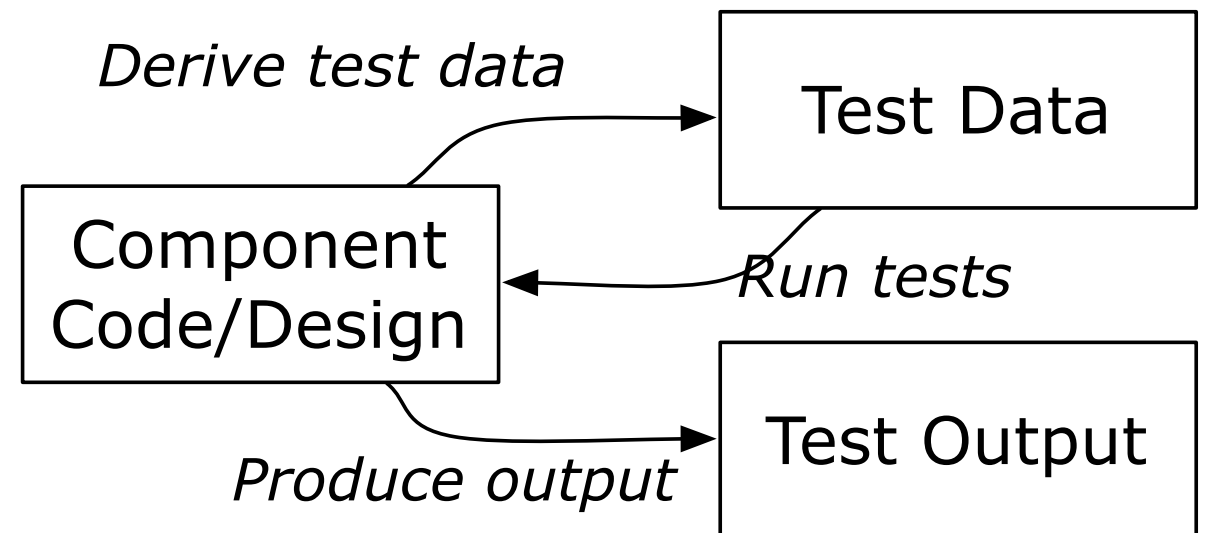


- Testing Strategies
  - + Unit & Integration Testing
  - + Regression Testing
  - + Acceptance Testing
  - + More Testing Strategies
- Miscellaneous
  - + When to Stop?
  - + Tool Support
- Agile Testing (DevOps)
  - + Flipping the V
  - + 4-Quadrants
  - + FIT Tables
- Conclusion
  - + More Good Reasons

# White Box Testing

- a.k.a. Structural testing, Testing in the small

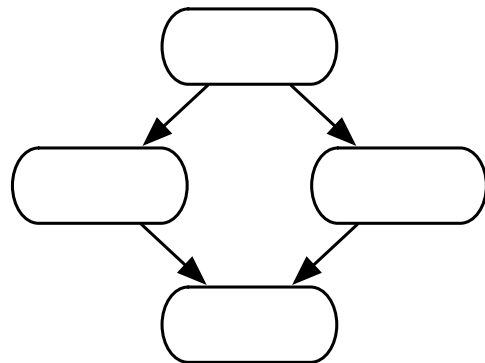
- + Treat a component as a “white box”, i.e. you can inspect its internal structure
- + Internal structure is also design specs; e.g. sequence diagrams, state charts, ...
- + Derive test cases to maximize coverage of that structure, yet minimize number of test cases



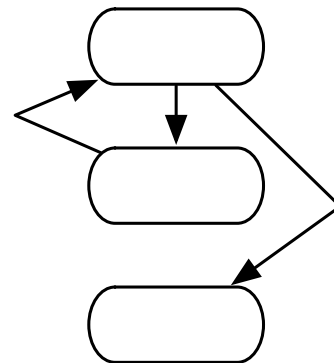
- + Coverage criteria
  - every statement at least once
  - all portions of control flow (= branches) at least once
  - all possible values of compound conditions at least once
  - all portions of data flow at least once
  - all loops, iterated at least 0, once, and N times

# Basis Path Testing (1/2)

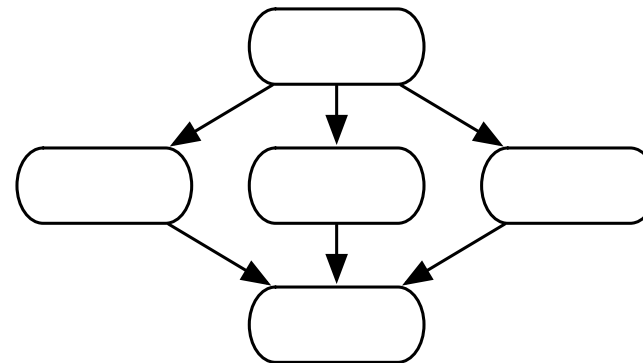
- + 1. Draw a control flow graph
  - nodes = sequences of non branching statements (assignments, procedure calls)
  - edges = control flow



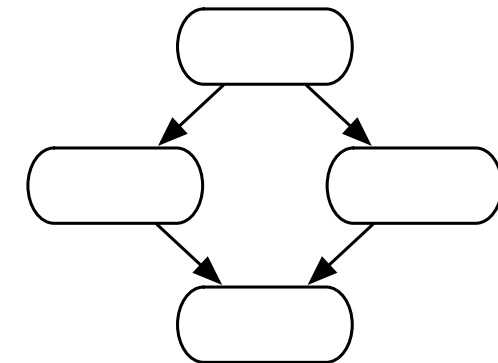
if-then-else  
[cc = 2]



while  
[cc = 2]



case-of  
[cc = 3]

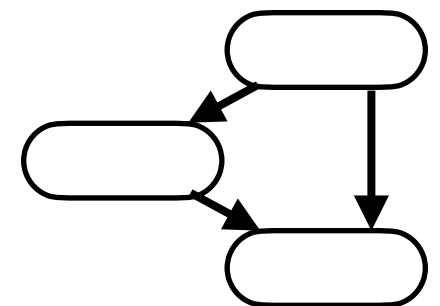


and/or  
= if-then-else  
[cc = 2]

**Guiding principle:** make sure that the control flow graphs stays as close as possible to the actual source code. This allows for better traceability when demonstrating that the test suite is well designed.

## Clarification

- + Empty nodes (= an empty sequence of non-branching statement)
  - Removing them graph does not affect the cyclomatic complexity
  - But it hinders traceability
- + What with an if-then (without an else branch)
  - Then you can remove the empty else branch



# Basis Path Testing (2/2)

- + ...
- + 2. Compute the Cyclomatic Complexity
  - =  $\#(\text{edges}) - \#(\text{nodes}) + 2$
  - = number of binary conditions + 1
  - = # regions
- + 3. Determine a set of *independent* paths (= at least one new edge in every path)  
[name *independent* stems from a mathematical vector *basis* for the complete graph]
  - Several possibilities: upper bound = Cyclomatic Complexity
- + 4. Prepare test cases that force each of these paths
  - Choose values for all variables that control the branches.
  - Predict the result in terms of values and/or exceptions raised
- + 5. Write test driver for each test case

# Example - Code

```
public boolean find(int key) {                                     //Binary Search
    int bottom = 0;                                              // (1)
    int top = _elements.length-1;
    int lastIndex = (bottom+top)/2;
    int mid;
    boolean found = key == _elements[lastIndex];
    while ((bottom <= top) && !found) {                            // (2) (3)
        mid = (bottom + top) / 2;
        found = key == _elements[mid];
        if (found) {                                            // (5)
            lastIndex = mid;                                    // (6)
        } else {
            if (_elements[mid] < key) {                          // (7)
                bottom = mid + 1;                                // (8)
            } else {
                top = mid - 1; }                                // (9)
        }                                                       // (10) (11)
    }                                                            // (4) (12)
    return found;                                              // (13)
}
```

(\*) (4) and (12) are needed to close the control flow path because the condition in (2) and (3) must be split up in two primitive conditions: (2) (bottom <= top) and (3) !found.

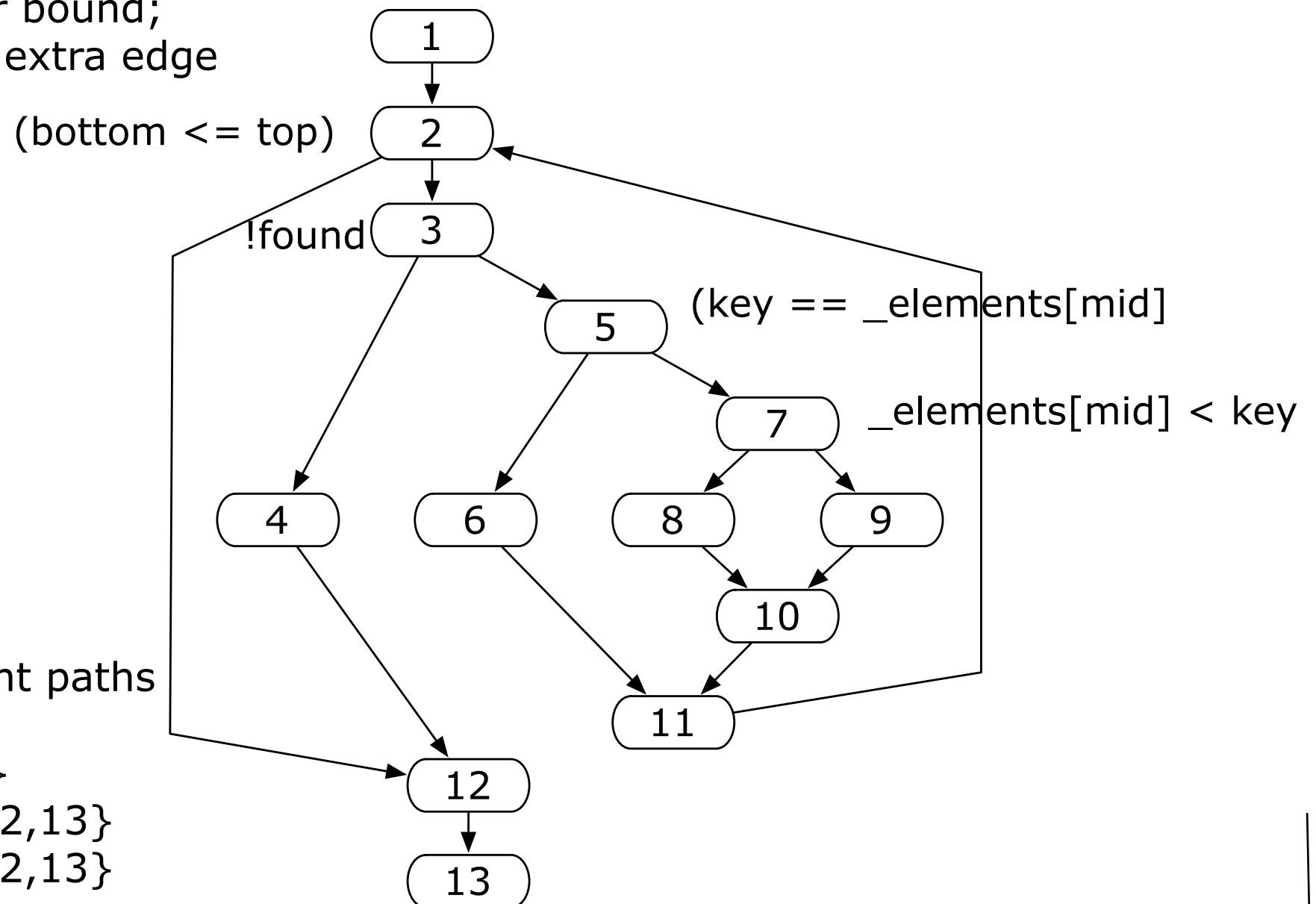
Remember: A boolean expression involving an "and" or "or" is equivalent to an if statement.

# Example - Flow Graph

set of independent paths of a flow graph  $\Rightarrow$  try to cover all the edges in the graph.

Heuristic for constructing such a set

- upper bound for size =  $16 - 13 + 2 = 4 + 1 = 5$
- pick most simple entry/exit path:  $\{1, 2, 12, 13\}$
- add new paths until upper bound;  
each addition includes an extra edge



- possible set of independent paths
  - +  $\{1, 2, 3, 4, 12, 13\}$
  - +  $\{1, 2, 3, 5, 6, 11, 2, 12, 13\}$
  - +  $\{1, 2, 3, 5, 7, 8, 10, 11, 2, 12, 13\}$
  - +  $\{1, 2, 3, 5, 7, 9, 10, 11, 2, 12, 13\}$



# Example - Test Cases

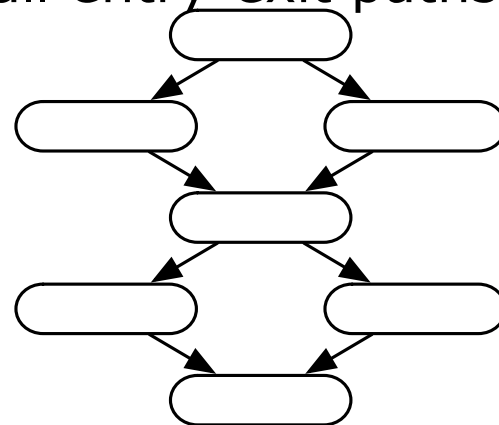
| Path   | Input  | Output                      |
|--|--|-----------------------------|
| {1,2,12,13}  | _elements = []; key = 5  | false / index out of bounds |
| {1,2,3,4,12,13}  | _elements = [1, 5, 9]; key = 5   | TRUE                        |
| {1,2,3,5,6,11,2,12,13}<br>{1,2,3,5,7,9,10,11,2,3,<br>5,6,11,2,12,13} | _elements = [1, 5, 9]; key = 1<br><i>actual path is not intended path(*)</i> | TRUE                        |
| {1,2,3,5,7,8,10,11,2,12,13}  | _elements = [5]; key = 9   | FALSE                       |
| {1,2,3,5,7,9,10,11,2,12,13}  | _elements = [5]; key = 1   | FALSE                       |

(\*) The *intended* path resulting from the heuristic is {1,2,3,5,6,11,2,12,13}. However, this path can never be forced by any input value. Therefore the *actual* path is a little different and takes an extra cycle.

# Basis Path Testing: Evaluation

- Pros
  - + coverage = (most of the times) every statement + all portions of control flow (branches)
    - \* reasonable coverage for reasonable effort
  - + tool support exists (computing cyclomatic complexity + drawing flow graph)
    - \* possibility to estimate testing complexity
- Cons
  - + construction is a heuristic: does not necessarily result in set of independent paths
  - + it is possible to get the same coverage with less paths
  - + it is sometimes not feasible to exercise all required paths
  - + it does not necessarily cover all entry-exit paths

```
if (x + y < 3)
  {x := 3} else {x := 5};
if (x + y < 3)
  {y := 3} else {y := 5};
```



- cc = 3 but 4 different entry-exit paths !
- Situation gets worse with nested conditionals

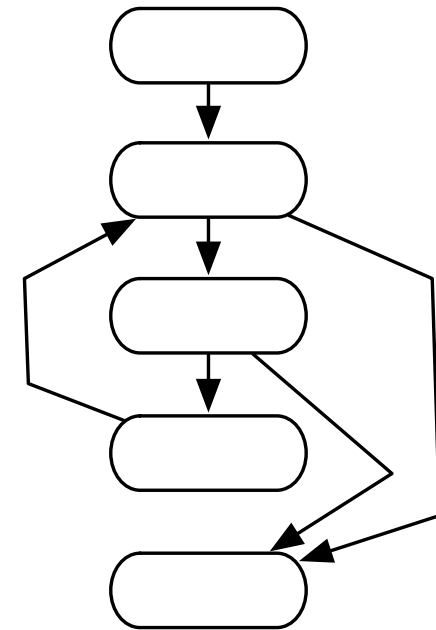
+ not all cc independent paths will cover all statements and all branches  
(see "Summary"; Perform basis path with a nested conditional of 2 levels deep.)

**For crucial code, complement basis path with condition and loop testing**

# Condition Testing

- For complex boolean expressions, Basis Path Testing is not enough!

```
1. public void helloWorld (int x, y, z) {  
2.   assert((x <> y) && (x <> z));  
3.   while (x > y) && (x > z) {  
4.     printf('Hello World');  
5.     x = x - 1;  
6.   };  
7.   assert((x == y) || (x == z));  
8. }
```



- Input
  - + {x = 3, y=4, z = 4}, {x = 4, y=3, z = 3}, {x = 4, y=4, z = 3}
  - + exercises all paths ...
    - \* but several important conditions (assertions) are not triggered (e.g. {x = 3, y=3, z=3})
- Condition Testing
  - + *Condition coverage*: all true/false combinations for whole condition expressions
  - + *Multiple condition coverage*: all true/false combinations for all simple conditions
  - + *Domain Testing*: all combinations of true/false + almost "true/false"  
for each occurrence of  $a < b$ ,  $a \leq b$ ,  $a == b$ ,  $a <> b$ ,  $a \geq b$ , 3 tests
    - \* test cases {a < b; a == b; a > b}

# Condition Testing - Test Cases

## Condition Coverage

line 2:  $(x \neq y) \ \&\& \ (x \neq z)$ :  $\{x = 3, y=3, z = 3\}$  and  $\{x = 4, y=3, z = 3\}$

line 3:  $(x > y) \ \&\& \ (x > z)$  and line 7:  $(x == y) \ || \ (x == z)$  are exercised by same values

## Multiple Condition Coverage

line 2:  $\{x = 3, y=3, z = 3\}$ ,  $\{x = 4, y=3, z = 3\}$ ,  $\{x = 4, y=4, z = 3\}$ ,  
 $\{x = 2, y = 3, z = 4\}$

line 3 and line 7: are exercised by same values

## Domain Testing

|         | $x = z$               | $x < z$                         | $x > z$                         |
|---------|-----------------------|---------------------------------|---------------------------------|
| $x = y$ | $x = 3, y = 3, z = 3$ | $x = 2, y = 2, z = 3$           | $x = 4, y = 4, z = 3$           |
| $x < y$ | $x = 2, y = 3, z = 2$ | $y = z$ : $x = 2, y = 3, z = 3$ | $y = z$ : --- not possible      |
|         |                       | $y < z$ : $x = 2, y = 3, z = 4$ | $y < z$ : --- not possible      |
|         |                       | $y > z$ : $x = 2, y = 4, z = 3$ | $y > z$ : $x = 3, y = 4, z = 2$ |
| $x > y$ | $x = 4, y = 3, z = 4$ | $y = z$ : --- not possible      | $y = z$ : $x = 4, y = 3, z = 3$ |
|         |                       | $y < z$ : $x = 3, y = 2, z = 4$ | $y < z$ : $x = 4, y = 2, z = 3$ |
|         |                       | $y > z$ : --- not possible      | $y > z$ : $x = 4, y = 3, z = 2$ |

# Loop Testing

for all loops L, with n allowable passes:

- (i) skip the loop;
- (ii) 1 pass through the loop;
- (iii) 2 passes through the loop;
- (iv) m passes where  $2 < m < n$ ;
- (v) n-1, n, n+1 passes

Test cases for binary search:  $n = \log_2(\text{size}(\_elements)) = \log_2(16) = 4$

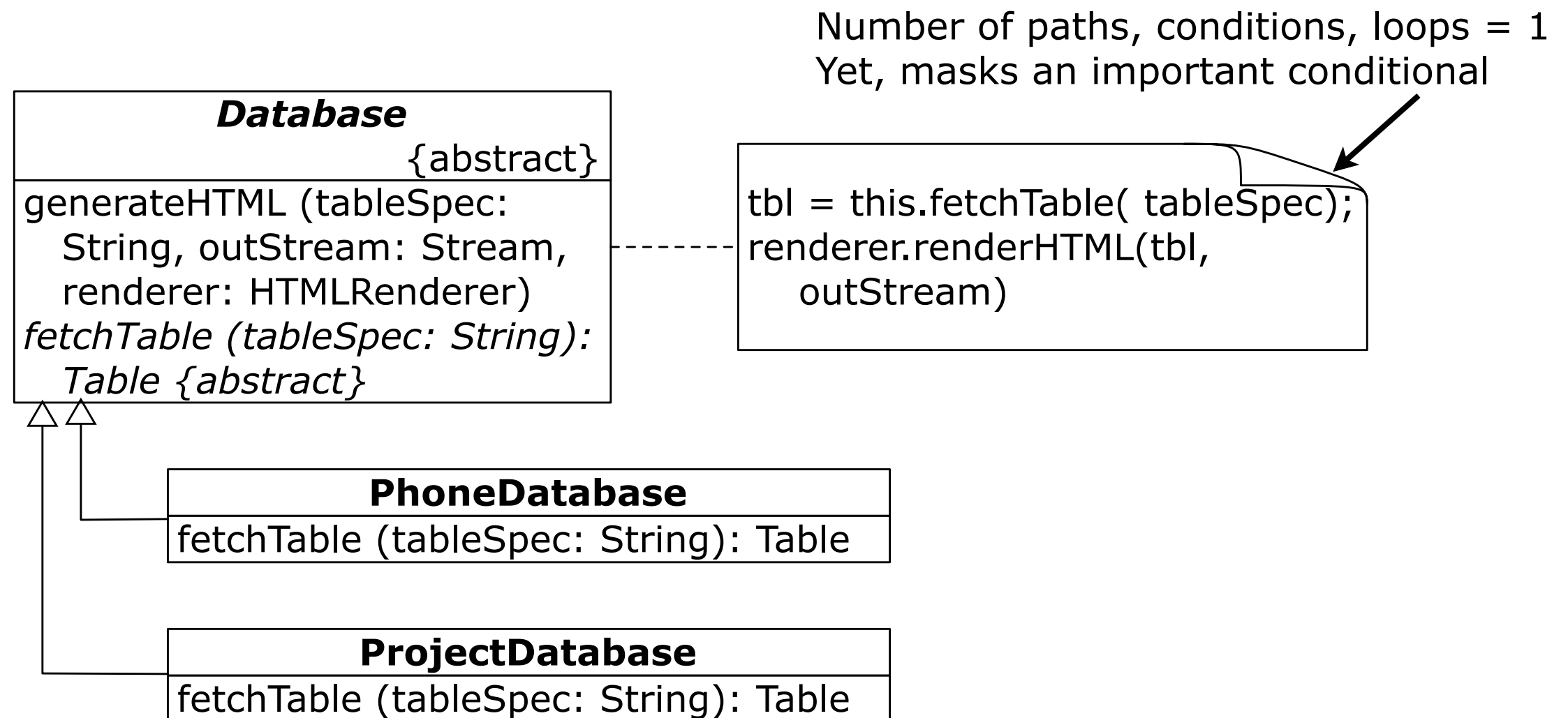
| Path                       | Input   | Output (*) |
|----------------------------|---|------------|
| skip the loop              | <code>_elements = [1, 3, ..., 29, 31]; key = ...</code> |            |
| 1 pass through the loop    | <code>_elements = [1, 3, ..., 29, 31]; key = ...</code> |            |
| 2 passes through the loop  | <code>_elements = [1, 3, ..., 29, 31]; key = ...</code> |            |
| m passes where $2 < m < n$ | <code>_elements = [1, 3, ..., 29, 31]; key = ...</code> |            |
| n-1                        | <code>_elements = [1, 3, ..., 29, 31]; key = ...</code> |            |
| n passes                   | <code>_elements = [1, 3, ..., 29, 31]; key = ...</code> |            |
| n+1 passes                 | <code>_elements = [1, 3, ..., 29, 31]; key = ...</code> |            |

(\*) The actual test cases are left as an exercise

# White Box Testing and Objects (1/2)

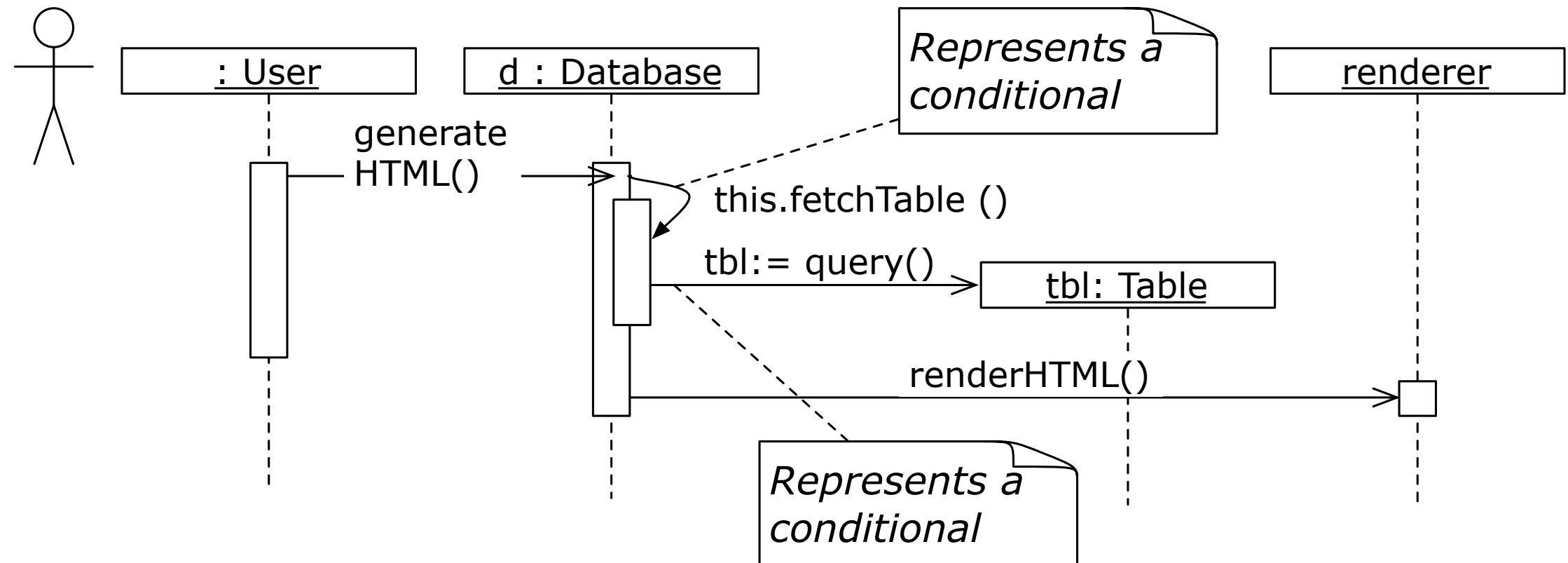
Pure white box testing is less relevant in an object-oriented context.

- Internal structure embedded in object compositions and polymorphic method invocations



# White Box Testing and Objects (2/2)

... but: sequence & collaboration diagrams may serve better



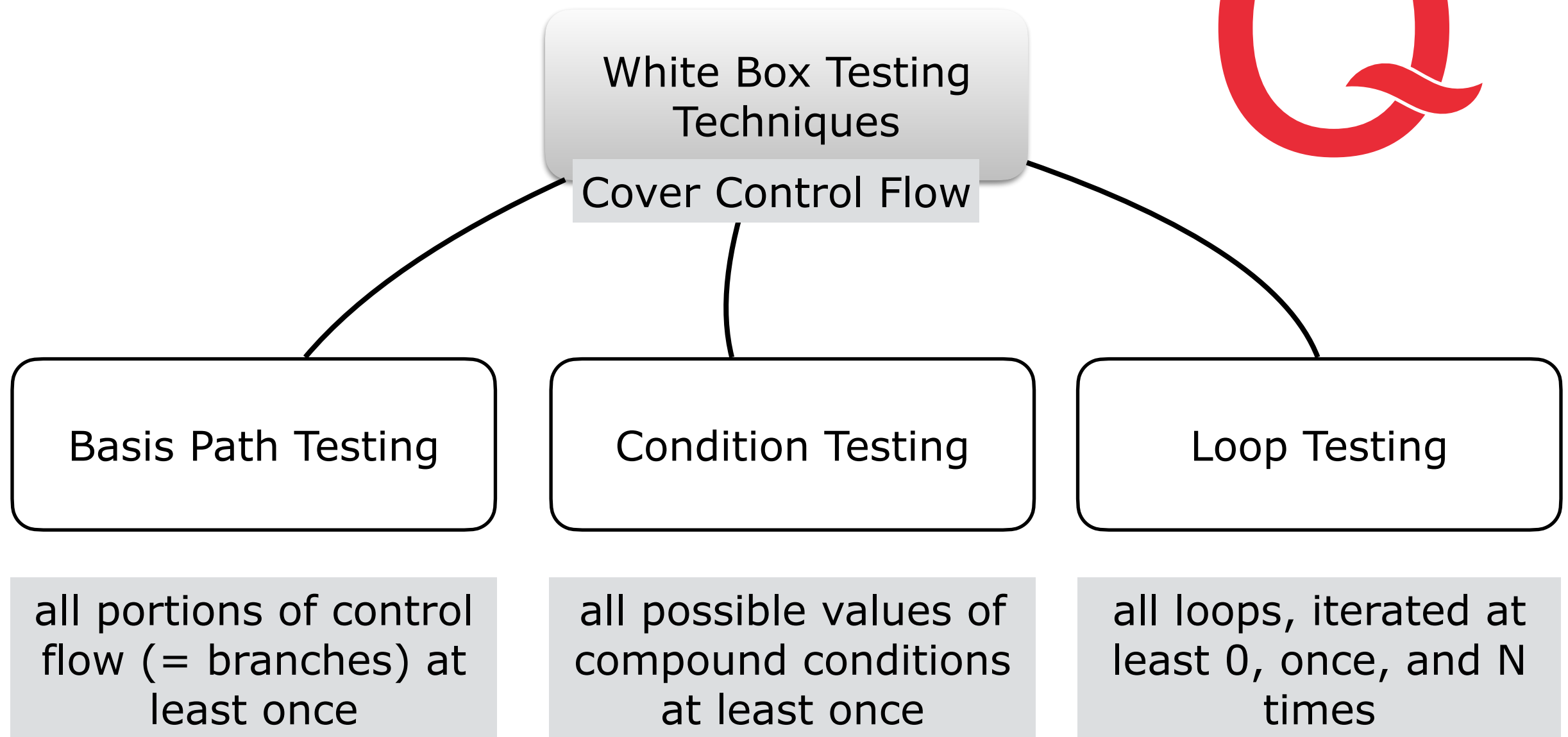
⇒ Identify polymorphic messages representing a conditional

⇒ plug-in instances of appropriate subclasses to exercise branches

The distinction between white-box and black-box testing is not that sharp.

# Question

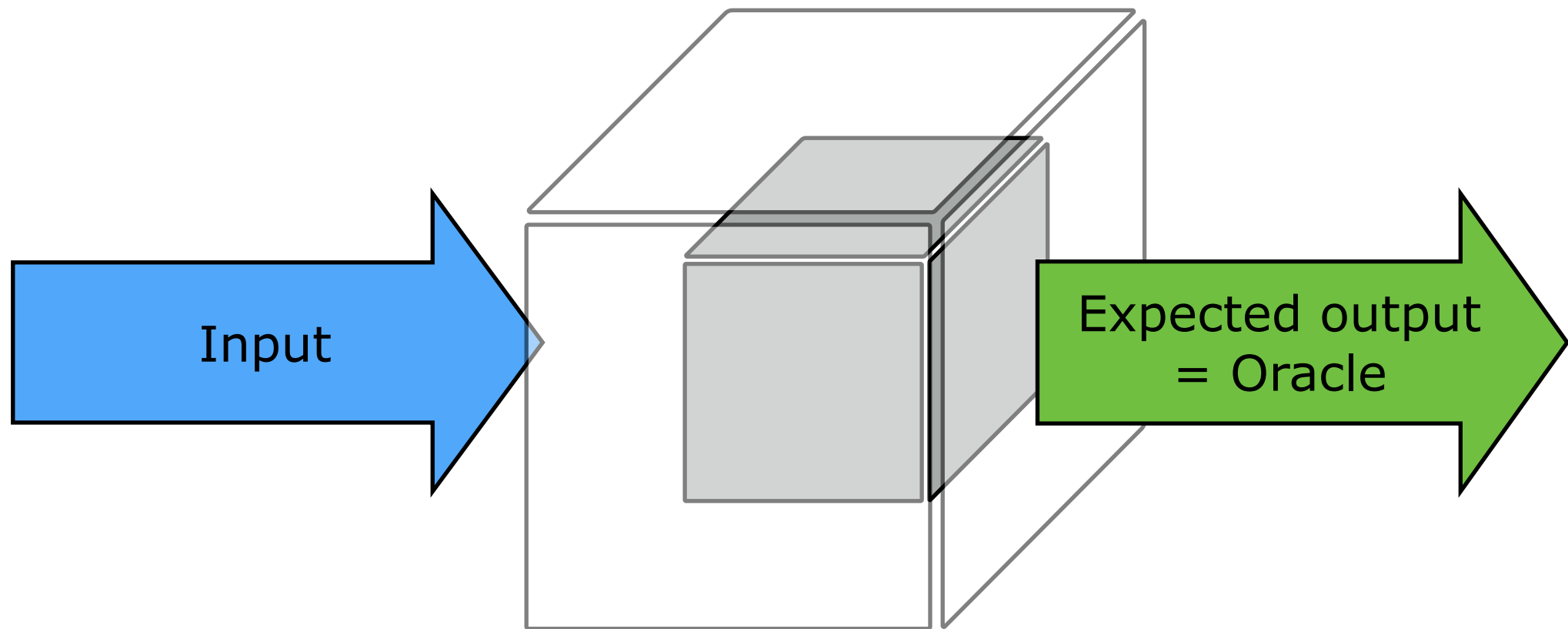
What are the differences and similarities between basis path testing, condition testing and loop testing?





# Code Coverage: Strength of a Test Suite

**\*\*New slide\*\***



**Code Coverage:**

The degree to which code is exercised by a test suite, expressed as a percentage.

# Code Coverage

## LCOV - code coverage report

|                             |       |       |
|-----------------------------|-------|-------|
| Current view: top level     |       |       |
| Test: libbash test coverage |       |       |
| Date: 2011-05-26            |       |       |
|                             | Hit   | Total |
| Lines:                      | 20840 | 34749 |
| Functions:                  | 1184  | 1287  |
| Branches:                   | 15689 | 37086 |

c++

| Directory          | Line Coverage     | Functions         |
|--------------------|-------------------|-------------------|
| src/core           | 95.7 % 314 / 328  | 98.2 % 55 / 56    |
| test               | 97.0 % 98 / 101   | 100.0 % 72 / 72   |
| src/buildins/tests | 98.6 % 144 / 146  | 100.0 % 203 / 203 |
| src/buildins       | 98.6 % 214 / 217  | 100.0 % 45 / 45   |
| src/core/tests     | 98.9 % 351 / 355  | 99.3 % 133 / 134  |
| ./src/buildins     | 100.0 % 8 / 9     | 93.3 % 14 / 15    |
| src                | 100.0 % 35 / 35   | 91.7 % 11 / 12    |
| ./src/core         | 100.0 % 190 / 190 | 98.0 % 99 / 101   |

Generated by: [LCOV version 1.9](#)

java

| Element                        | Coverage | Covered Lines | Total Lines |
|--------------------------------|----------|---------------|-------------|
| java - commons-collections     | 79.5 %   | 10927         | 13738       |
| org.apache.commons.collections | 74.1 %   | 3842          | 5183        |
| ArrayStack.java                | 86.5 %   | 32            | 37          |
| BagUtils.java                  | 86.7 %   | 13            | 15          |
| BinaryMap.java                 | 72.4 %   | 155           | 214         |
| BinaryHeap.java                | 87.5 %   | 127           | 145         |
| BoundedFifoBuffer.java         | 93.2 %   | 62            | 66          |
| BufferOverflowException.java   | 55.5 %   | 5             | 9           |
| BufferUnderflowException.java  | 88.5 %   | 8             | 9           |
| BufferUtils.java               | 30.5 %   | 4             | 13          |
| ClosureUtils.java              | 93.5 %   | 31            | 33          |
| CollectionUtils.java           | 92.4 %   | 293           | 317         |
| ComparatorUtils.java           | 8.5 %    | 3             | 35          |
| CursorableLinkedList.java      | 85.4 %   | 44            | 520         |

Tools to measure line coverage, statement coverage, function coverage, branch coverage readily exist



CAPSTONE PROJECT

# Modified Condition/Decision Coverage (MC/DC)

- Condition  $\approx$  Condition on Input to the function/component under test
- Decision  $\approx$  Output of the function/component under test

MC/DC is required by most software standards for safety critical software.

(DO-178C: Avionics Safety Standard; ISO 26262: Road vehicles – Functional safety; ISO/IEC 62304: medical device software)

MC/DC requires all of the below during testing:

- Each entry and exit point is invoked.
- Each decision takes every possible outcome.
- Each condition in a decision takes every possible outcome.
- Each condition in a decision is shown to *independently affect* the outcome of the decision.
  - + Independence of a condition is shown by proving that only one condition changes at a time.

# MC/DC Example

```
int isReadyToTakeOff(int a, int b, int c, int d) {  
    if(((a == 1) || (b == 1)) && ((c == 1) || (d == 1))) return 1; else return 0;  
}
```

2 decisions: "return 1" or "return 0"  
4 inputs: a, b, c, d  
4 conditions: (a == 1) / (b == 1) / (c == 1) / (d == 1)

Decision Coverage

- 2 test cases, one for each decision

Condition Coverage

- 2 test cases, one for all conditions to be true, one for all conditions to be false

Condition/Decision Coverage

- 3 test cases, all decisions at least once + all conditions once true, once false

Modified Condition/Decision Coverage

- n + 1 test cases (for a decision with n conditions)

Multiple Condition Coverage

- 2<sup>n</sup> test cases (for a decision with n conditions)  
+ Usually too large to handle

| Condition/Decision Coverage |       |       |       |          |
|-----------------------------|-------|-------|-------|----------|
| a==1                        | b==1  | c==1  | d==1  | decision |
| FALSE                       | TRUE  | TRUE  | TRUE  | return 1 |
| FALSE                       | FALSE | FALSE | TRUE  | return 0 |
| TRUE                        | FALSE | FALSE | FALSE | return 0 |

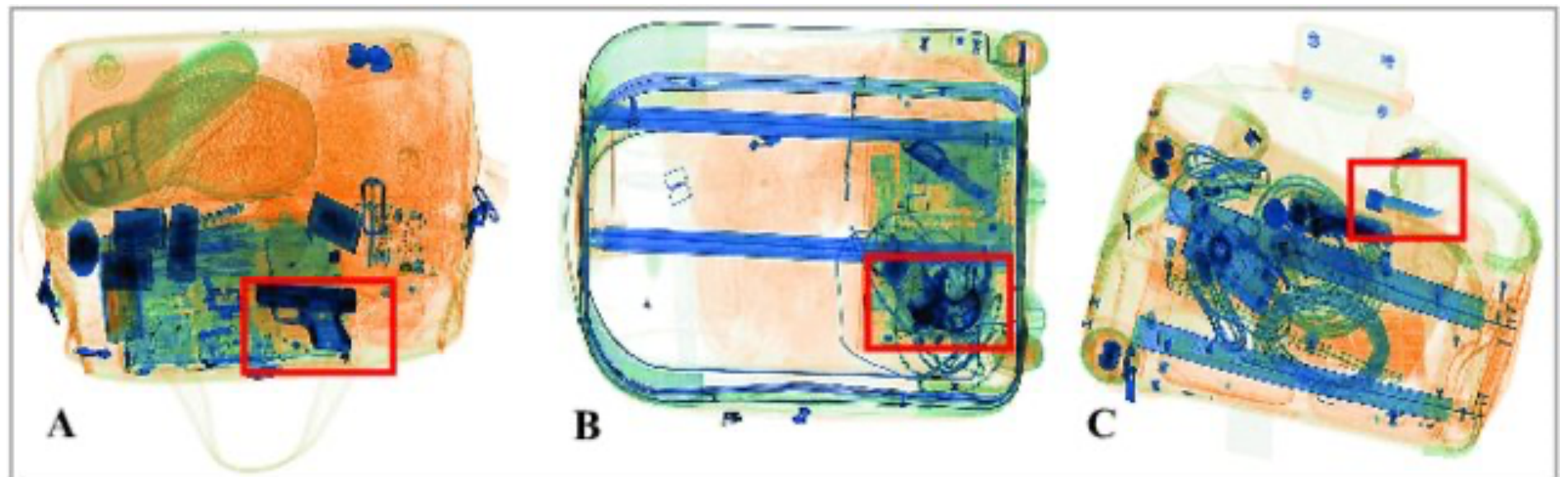
| Modified Condition/Decision Coverage |       |       |       |          |                           |
|--------------------------------------|-------|-------|-------|----------|---------------------------|
| a==1                                 | b==1  | c==1  | d==1  | decision |                           |
| TRUE                                 | FALSE | TRUE  | FALSE | return 1 | + row 4 shows effect of c |
| TRUE                                 | FALSE | FALSE | TRUE  | return 1 | + row 5 shows effect of a |
| FALSE                                | TRUE  | FALSE | TRUE  | return 1 | + row 5 shows effect of b |
| TRUE                                 | FALSE | FALSE | FALSE | return 0 | + row 2 shows effect of d |
| FALSE                                | FALSE | FALSE | TRUE  | return 0 |                           |

# Mutation Testing: Metaphor



© Brussels Airlines

**How to test the quality assurance?  
Inject synthetic problematic items.**



© "The Good, the Bad and the Ugly: Evaluating Convolutional Neural Networks for Prohibited Item Detection Using Real and Synthetically Composite X-ray Imagery" Neelanjan Bhowmik, Qian Wang, Yona Falinie A. Gaus, Marcin Szarek, Toby P. Breckon

# Code Coverage

## gTest Example: findLast

The screenshot shows an IDE window titled "MutationTestExample - tests.cpp". The left sidebar displays a project tree for "MutationTestExample" with 62% files and 23% lines covered. A red circle highlights the "find.cpp" file, which is marked as "100% lines covered". The main editor shows the source code of "find.cpp", which includes headers for `<vector>`, `<gtest/gtest.h>`, and `"find.cpp"`. It contains three test functions: `TEST(FindLastTests, noOccurrence)`, `TEST(FindLastTests, doubleOccurrence)`, and `TEST(FindLastTests, emptyVector)`. The bottom panel shows the output of the test run, with a red circle highlighting the summary line: `[ PASSED ] 5 tests.`

100% line coverage  
100% statement coverage  
100% branch coverage  
100% MC/DC coverage

... all tests passed



# Inject Mutant (Survived - Live)

```
01  int findLast(std::vector<int> x, int y) {  
02      if (x.size() == 0)  
03          return -1;  
04      for (int i = x.size() - 1; i > 0; i--)  
05          if (x[i] == y)  
06              return i;  
07      return -1;  
08  }
```

Relational Operator Replacement (ROR)  
"i >= 0" becomes "i > 0"

```
[=====] 3 tests from 1 test suite ran. (0 ms total)  
[ PASSED ] 3 tests.
```

⇒ One of these tests should fail!

But all of them pass: the mutant survives.

# Extra test *kills* the mutant

## Find.cpp

```
#include <vector>

int findLast(std::vector<int> x, int y) {
    if (x.size() == 0)
        return -1;
    for (int i = x.size() - 1; i >= 0; i--)
        if (x[i] == y)
            return i;
    return -1;
}
```

## Tests.cpp

```
#include <vector>
#include <gtest/gtest.h>

#include "find.cpp"
```

```
TEST(FindLastTests, occurrenceOnBoundary) {
    EXPECT_EQ(0, findLast({1, 2, 42, 42, 63}, 1));
}
```

```
TEST(FindLastTests, noOccurrence) {
    EXPECT_EQ(-1, findLast({1, 2, 42, 42, 63}, 99));
}
```

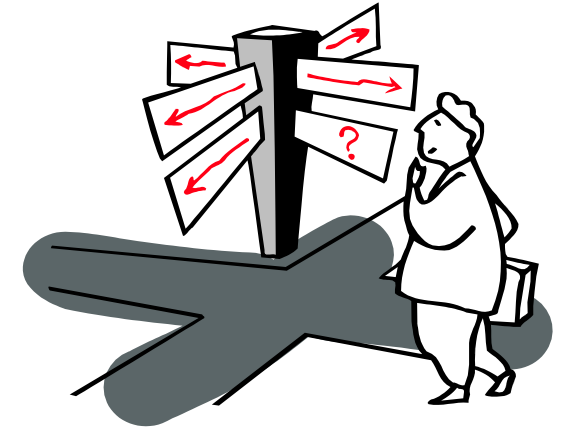
```
TEST(FindLastTests, doubleOccurrence) {
    EXPECT_EQ(3, findLast({1, 2, 42, 42, 63}, 42));
}
```

```
TEST(FindLastTests, emptyVector) {
    EXPECT_EQ(-1, findLast({}, 3));
}
```



# CHAPTER 5 – Testing

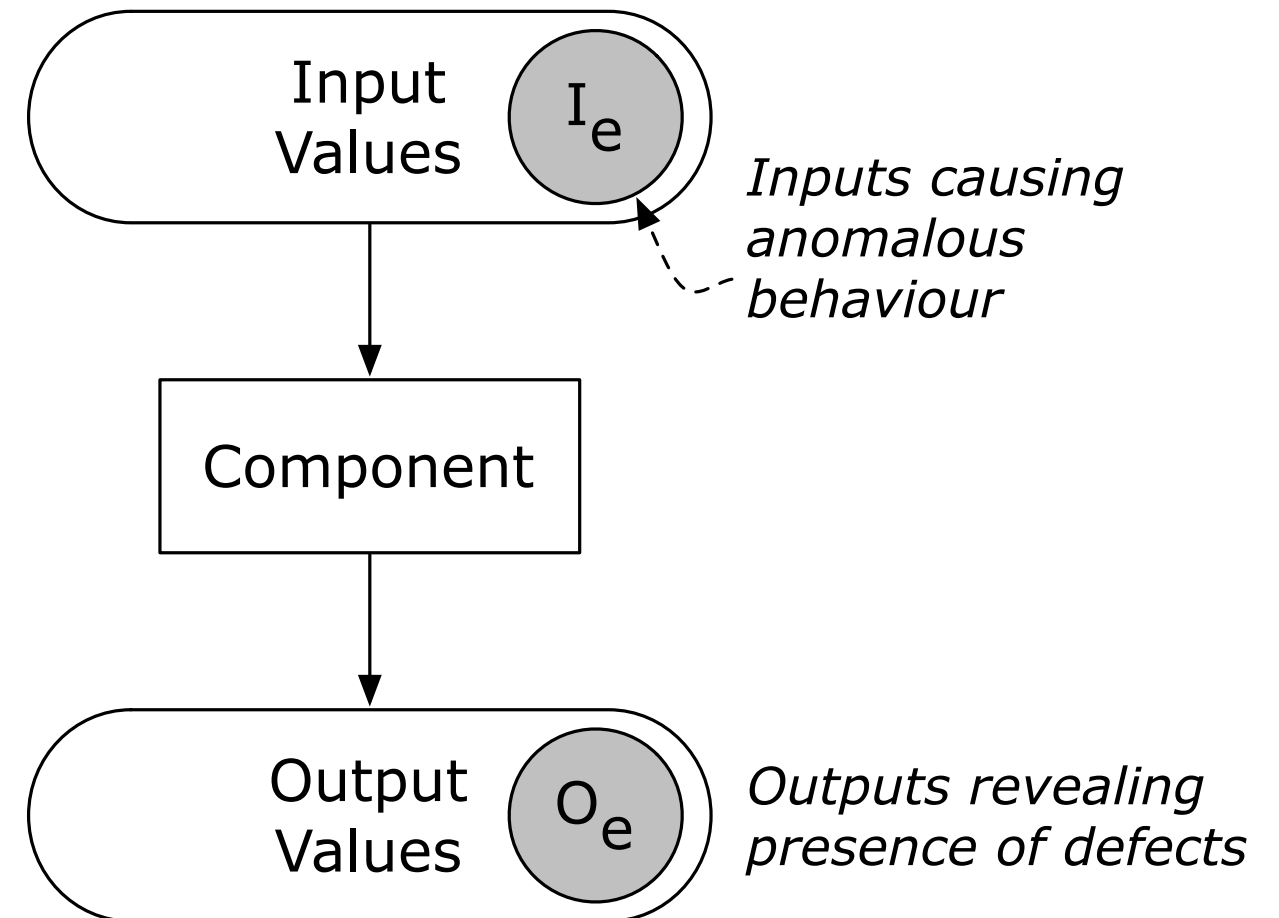
- Introduction
  - + When, Why, What & Who?
    - The V-Model
  - + What is "Correct"?
  - + Terminology
- Testing Techniques
  - + White Box
    - basis path, conditions, loops
  - + Coverage
    - Code Coverage
    - MC/DC Coverage
    - Mutation Coverage
  - + Black Box
    - equivalence partitioning
  - + Fuzz testing



- Testing Strategies
  - + Unit & Integration Testing
  - + Regression Testing
  - + Acceptance Testing
  - + More Testing Strategies
- Miscellaneous
  - + When to Stop?
  - + Tool Support
- Agile Testing (DevOps)
  - + Flipping the V
  - + 4-Quadrants
  - + FIT Tables
- Conclusion
  - + More Good Reasons

# Black Box Testing

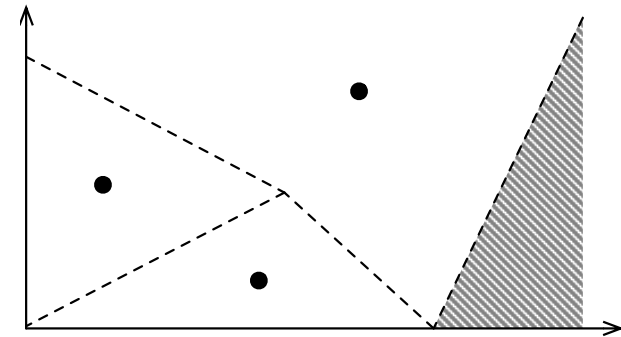
- a.k.a. Functional testing, Testing in the large
  - + Treat a component as a a “black box” whose behaviour can be determined only by studying its inputs and outputs.
  - + Test cases are derived from the external specification of the component
  - + Derive test cases to maximize coverage of elements in the spec, yet minimize number of test cases
  - + Coverage criteria
    - ⇒ all exceptions



# Equivalence Partitioning

## & Boundary Value Analysis

- 1. Divide input domain in classes of data, according to input condition.  
Input condition may require:
  - + a range  $\Rightarrow$  1 valid (in the range) and 2 invalid equivalence classes
  - + a value  $\Rightarrow$  1 valid (= value) and 2 invalid equivalence classes
  - + a set  $\Rightarrow$  1 valid (in the set) and 1 invalid equivalence class
  - + a boolean  $\Rightarrow$  1 valid and 1 invalid equivalence class
- 2. Choose test data corresponding to each equivalence class
  - + Normal equivalence partitioning chooses test data at random
  - + Boundary Value Analysis choose values at the "edge" of the class, e.g., just above and just below the minimum and maximum of a range
- 3. Predict the corresponding output and derive test case
- 4. Write test driver



You can partition the output domain as well and apply the same technique

# Equivalence Partitioning : Example

- Example: Binary search

```
private int[] _elements;  
public boolean find(int key) { ... }
```

- pre-condition(s)

- Array has at least one element
- Array is sorted

- post-condition(s)

- (The element is in \_elements and the result is true)  
or (The element is not in \_elements and the result is false)

- Check input partitions:

- + Do the inputs satisfy the pre-conditions?
- + Is the key in the array?
  - \* leads to (at least) 2x2 equivalence classes

- Check boundary conditions

- + Is the array of length 1?
- + Is the key at the start or end of the array?
  - \* leads to further subdivisions  
(not all combinations make sense)

# Equivalence Partitioning: Test Data

Generate test data that cover all meaningful equivalence partitions.

| Test Cases                           | Input                                   | Output    |
|--------------------------------------|---|-----------|
| Array length 0                       | key = 17, elements = { }                | FALSE     |
| Array not sorted                     | key = 17, elements = { 33, 20, 17, 18 } | exception |
| Array size 1, key in array           | key = 17, elements = { 17 }             | TRUE      |
| Array size 1, key not in array       | key = 0, elements = { 17 }              | FALSE     |
| Array size > 1, key is first element | key = 17, elements = { 17, 18, 20, 33 } | TRUE      |
| Array size > 1, key is last element  | key = 33, elements = { 17, 18, 20, 33 } | TRUE      |
| Array size > 1, key is in middle     | key = 20, elements = { 17, 18, 20, 33 } | TRUE      |
| Array size > 1, key not in array     | key = 50, elements = { 17, 18, 20, 33 } | FALSE     |
| ...                                  | ...                                     | ...       |

# Design by Contract — Tests

- + Pre- and post-conditions are part of the interface of a component.
  - Part of black-box testing, not white-box testing
    - > Do *not* include assertions in basis-path testing
    - > Borderline case: include assertions in condition testing

## + Example

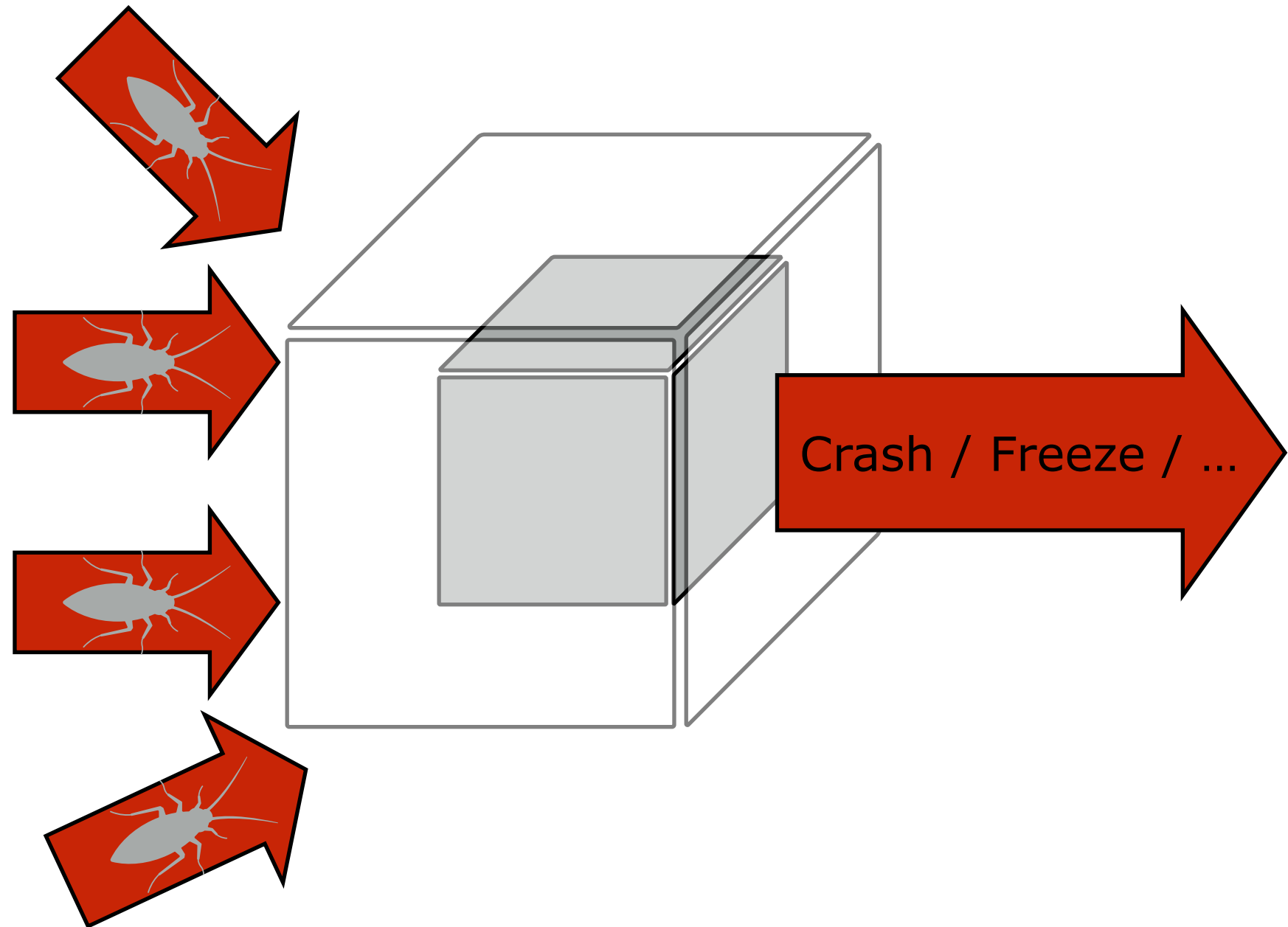
```
public char pop() throws AssertionError {  
    assert(!this.isEmpty());  
    return _store[_size--];  
}
```

- + Equivalence partition with boolean
  - = condition testing: 2 inputs cover all conditions
  - (test case 1 = non-empty stack / value on the top
  - (test case 2 = empty stack / assertion exception)

Repeat from  
(Chapter 5. Design by  
Contract)

# Fuzz Testing

**\*\*New slide\*\***



## Fuzz Testing:

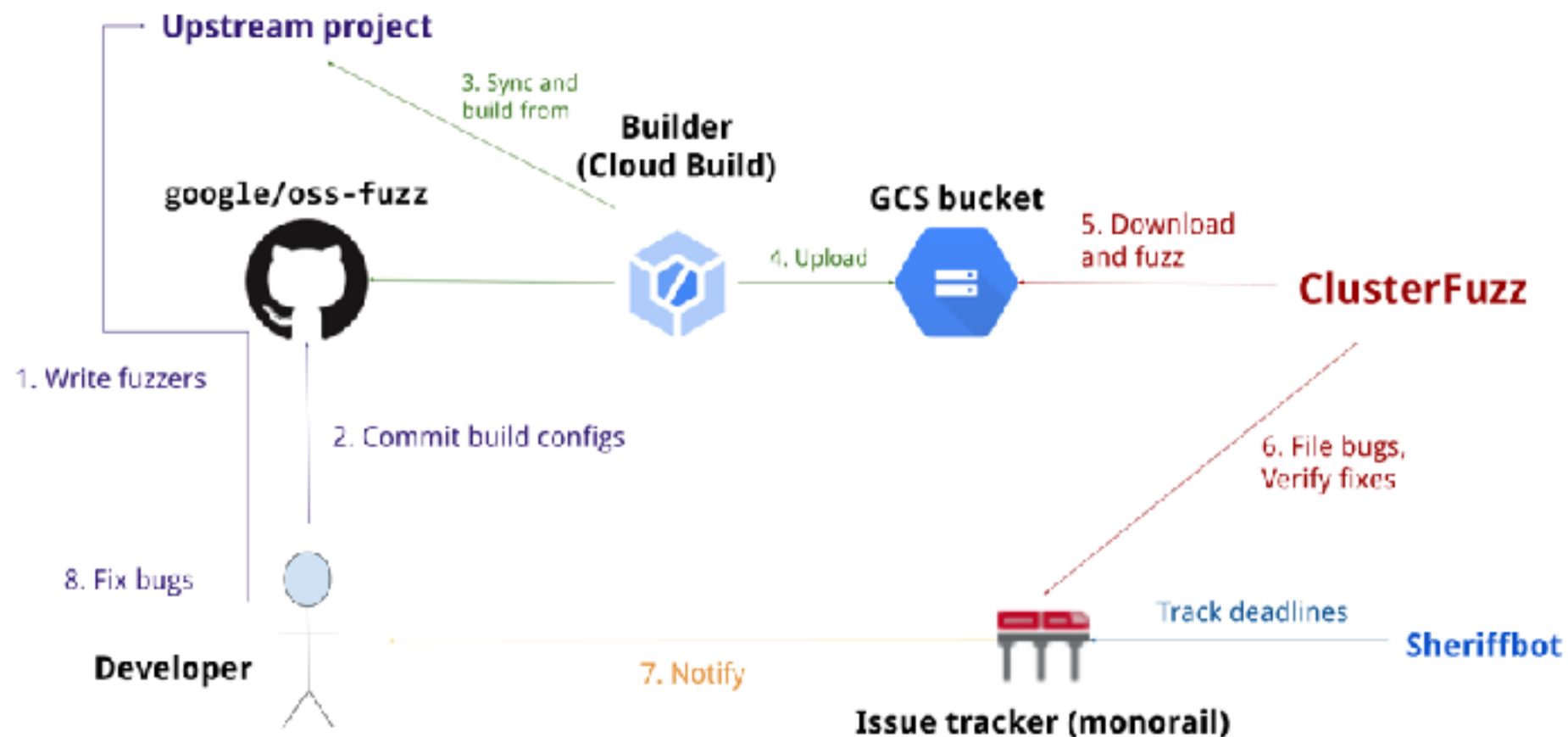
A software testing technique used to discover security vulnerabilities by inputting massive amounts of random data, called fuzz, to the component or system.

# Fuzz-Testing: Open Source Libraries

**\*\*New slide\*\***

## OSS-Fuzz: Continuous Fuzzing for Open Source Software

<https://github.com/google/oss-fuzz>



### Trophies

As of August 2023, OSS-Fuzz has helped identify and fix over 10,000 vulnerabilities and 36,000 bugs across 1,000 projects.



# Fuzz-Testing: REST-API

**\*\*New slide\*\***

RESTler: first stateful REST API fuzzing tool

<https://github.com/microsoft/restler-fu>



CAPSTONE PROJECT

- **Use-after-free rule.** A resource that has been deleted must no longer be accessible.
- **Resource-hierarchy rule.** A child resource of a parent resource must not be accessible from another parent resource.
- ...

In an Azure service, we found the following use-after-free violation.

- 1) Create a new resource R (with a PUT request).
- 2) Delete resource R (with a DELETE request).
- 3) Create a new child resource of the deleted resource R

and of a specific type (with another PUT request).

This sequence of requests results in a "500 Internal Server Error".

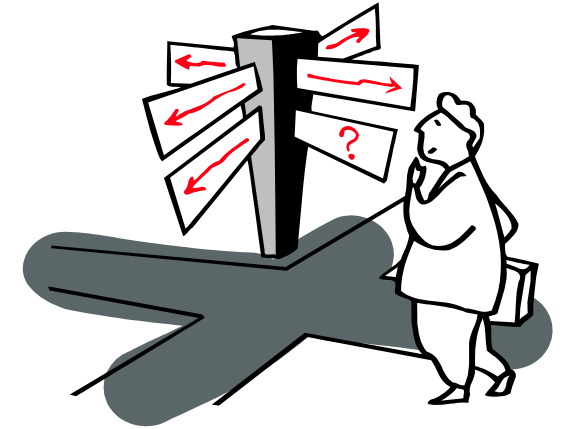
In an Office365 messaging service where users can post messages and then reply and edit these, the resource-hierarchy checker detected the following bug.

- 1) Create a first message msg-1 (with a request POST /api/posts/msg-1).
- 2) Create a second message msg-2 (with a request POST /api/posts/msg-2).
- 3) Create a reply reply-1 to the first message  
(with a request POST /api/posts/msg-1/replies/reply-1).
- 4) Edit the reply reply-1 with a PUT request using msg-2 as message identifier  
(with a request PUT /api/posts/msg-2/replies/reply-1).

Surprisingly, the last request in Step 4 returns a "200 Allowed" response while it must have returned a "404 Not Found" response.

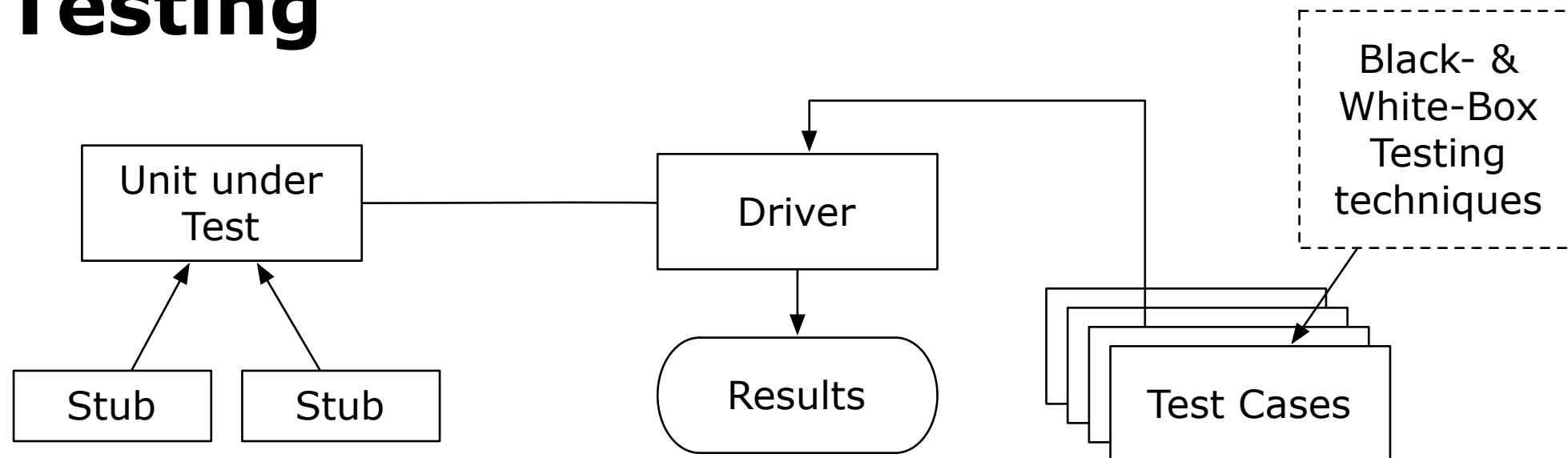
# CHAPTER 5 – Testing

- Introduction
  - + When, Why, What & Who?
    - The V-Model
  - + What is "Correct"?
  - + Terminology
- Testing Techniques
  - + White Box
    - basis path, conditions, loops
  - + Coverage
    - Code Coverage
    - MC/DC Coverage
    - Mutation Coverage
  - + Black Box
    - equivalence partitioning
  - + Fuzz Testing



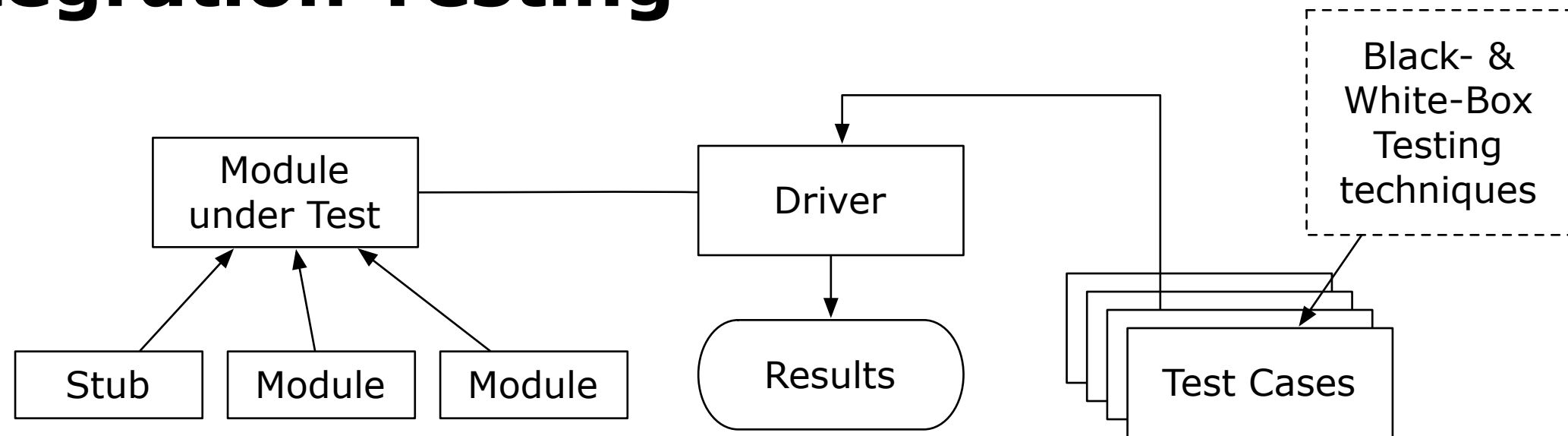
- Testing Strategies
  - + Unit & Integration Testing
  - + Regression Testing
  - + Acceptance Testing
  - + More Testing Strategies
- Miscellaneous
  - + When to Stop?
  - + Tool Support
- Agile Testing (DevOps)
  - + Flipping the V
  - + 4-Quadrants
  - + FIT Tables
- Conclusion
  - + More Good Reasons

# Unit Testing



- + Why?
  - Identify local defects (= within a unit) fast
- + Who?
  - Person developing the unit writes the tests.
- + When? At the latest when a unit is delivered to the rest of the team
  - No test  $\Rightarrow$  no unit
  - Test drivers & stubs are part of the system  $\Rightarrow$  configuration management
  - Today fully automated
- \*\*\* Write the test first,
  - + i.e. before writing the unit.
  - + It will encourage you to design the component interface right

# Integration Testing



+ Why?

- The sum is more than its parts,  
i.e. interfaces (and calls to them) may contain defects too.

+ Who?

- Person developing the module writes the tests.

+ When?

- Top-down: main module before constituting modules
- Bottom-up: constituting modules before integrated module
- In practice: a little bit of both

- ## The distinction between unit testing and integration testing is not that sharp!

# Regression Testing

Regression Testing ensures that all things that used to work still work after changes.

- Regression Test
  - + = re-execution of some subset of tests to ensure that changes have not caused unintended side effects
  - + tests must avoid regression (= degradation of results)
  - + Regression tests must be repeated often (after every change, every night, with each new unit, with each fix,...)
  - + Regression tests may be conducted manually
    - Execution of crucial scenarios with verification of results
    - Manual test process is slow and cumbersome
      - \* preferably completely automated
- Advantages
  - + Helps during iterative and incremental development
  - + during maintenance
- Disadvantage
  - + Up front investment in maintainability is difficult to sell to the customer

# Acceptance Testing

- Acceptance Tests
  - + conducted by the end-user (representatives)
  - + check whether requirements are correctly implemented
    - borderline between verification ("Are we building the system right?") and validation ("Are we building the right system?")
- Alpha- & Beta Tests
  - + acceptance tests for "off-the-shelves" software (many unidentified users)
    - Alpha Testing
      - > end-users are invited at the developer's site
      - > testing is done in a controlled environment
    - Beta Testing
      - > software is released to selected customers
      - > testing is done in "real world" setting, without developers present

# Question

What are the differences and similarities between unit testing and regression testing?

**\*\*New slide\*\***



Test Strategies

**Optimal Fault Localisation  
Automate as much as possible**

Unit Testing

**Exercise small component  
("unit under test")**

Regression Testing

**Exercise complete system  
("no regressions")**

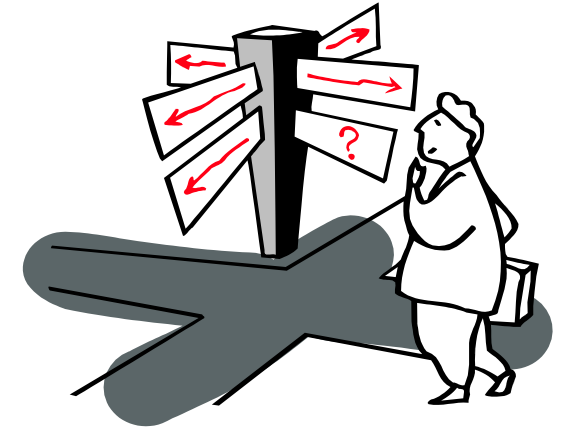
# More Testing Strategies

- Recovery Testing / Resilience Testing
  - + Test forces system to fail and checks whether it recovers properly
    - For fault tolerant systems
- Stress Testing (Overload Testing)
  - + Tests extreme conditions
    - e.g., supply input data twice as fast and check whether system fails
- Performance Testing
  - + Tests run-time performance of system
    - e.g., time consumption, memory consumption
      - > first do it, then do it right, then do it fast
- Back-to-Back Testing
  - + Compare test results from two different versions of the system
    - requires N-version programming or prototypes



# CHAPTER 5 – Testing

- Introduction
  - + When, Why, What & Who?
    - The V-Model
  - + What is "Correct"?
  - + Terminology
- Testing Techniques
  - + White Box
    - basis path, conditions, loops
  - + Coverage
    - Code Coverage
    - MC/DC Coverage
    - Mutation Coverage
  - + Black Box
    - equivalence partitioning
  - + Fuzz Testing

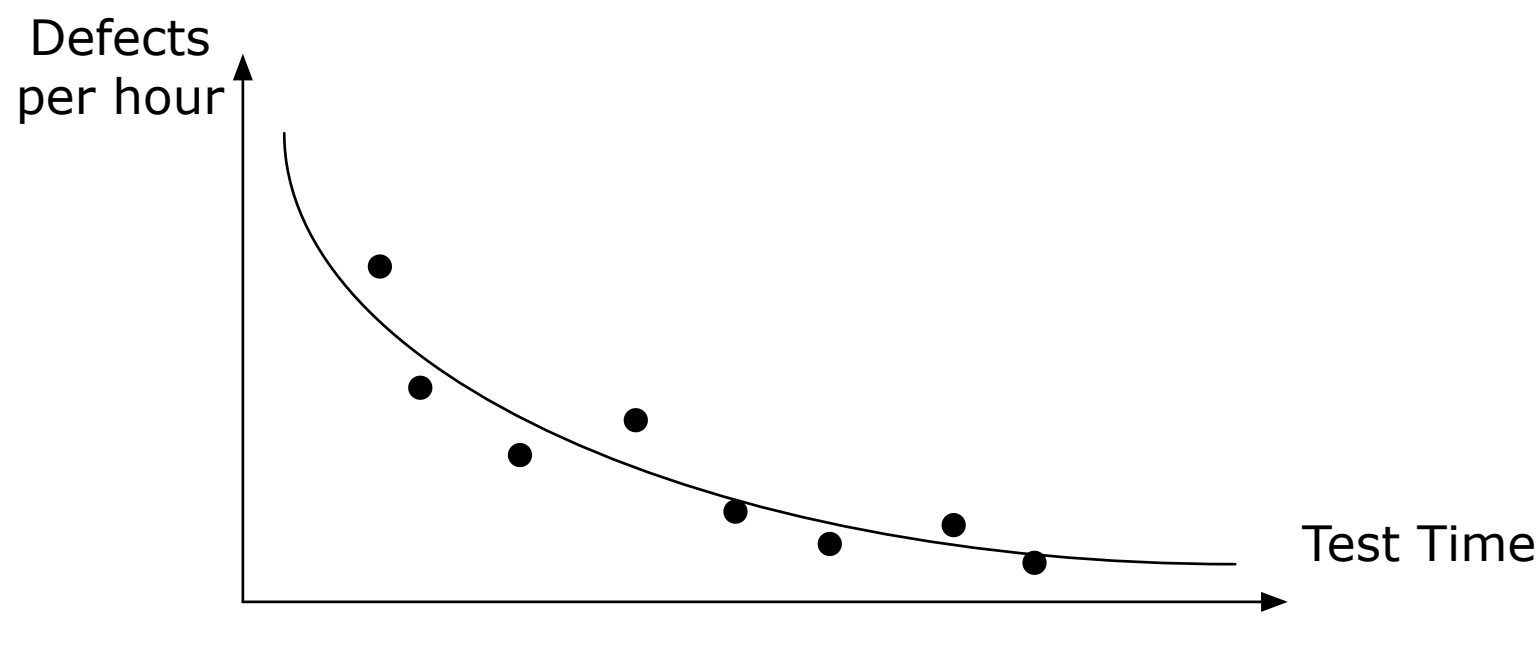


- Testing Strategies
  - + Unit & Integration Testing
  - + Regression Testing
  - + Acceptance Testing
  - + More Testing Strategies
- Miscellaneous
  - + When to Stop?
  - + Tool Support
- Agile Testing (DevOps)
  - + Flipping the V
  - + 4-Quadrants
  - + FIT Tables
- Conclusion
  - + More Good Reasons

# When to Stop?

When are we done testing? When do we have enough tests?

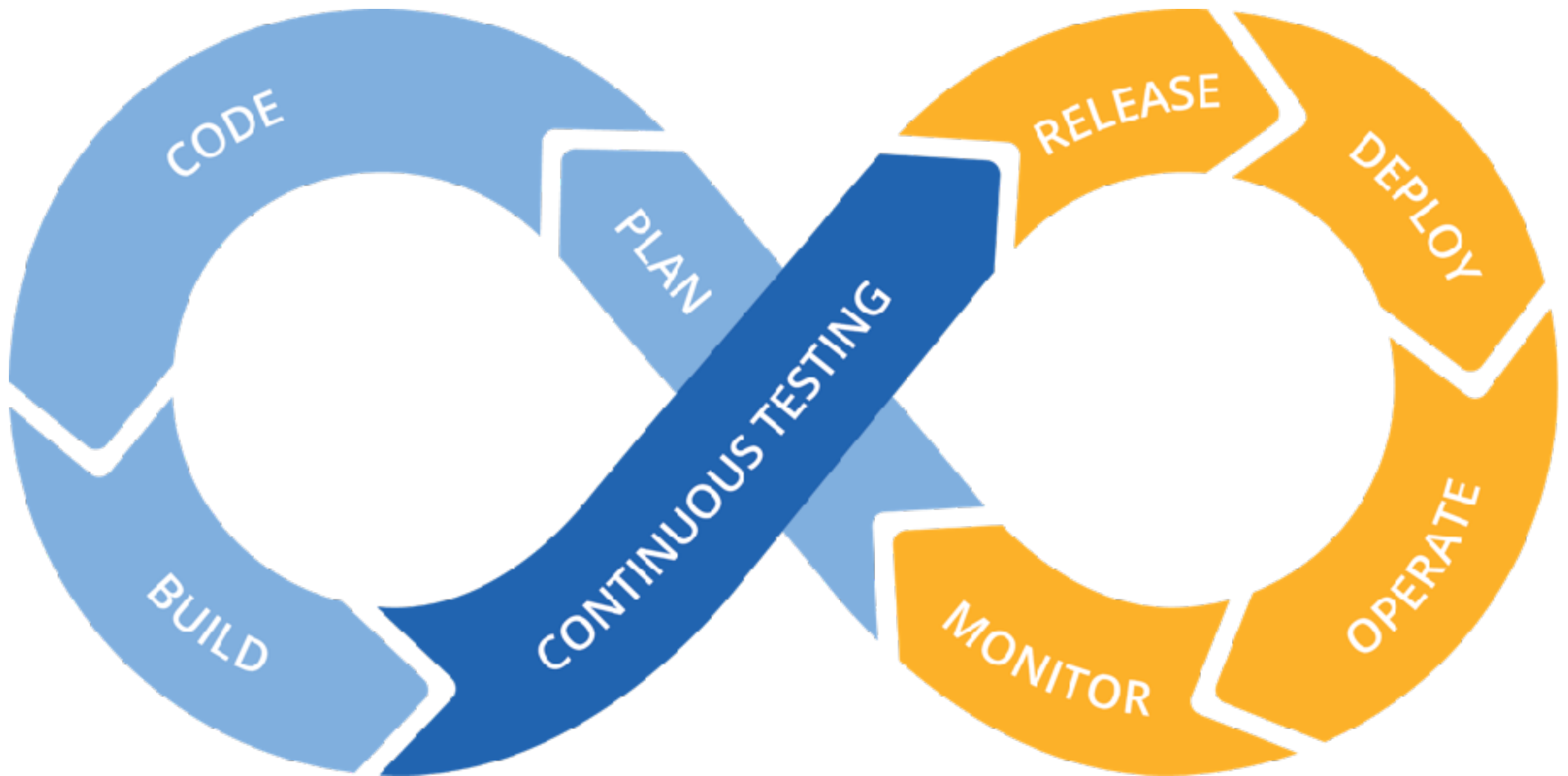
- Cynical Answers (sad but true)
  - + You're never done: each run of the system is a new test
    - > Each bug-fix should be accompanied by a new test
  - + You're done when you are out of time/money
    - > Include test in project plan
    - AND DO NOT GIVE IN TO PRESSURE
    - > ... in the long run, tests SAVE time
- Statistical Testing
  - + Test until you've reduced failure rate under risk threshold
    - \* Testing is like an insurance company calculating risks



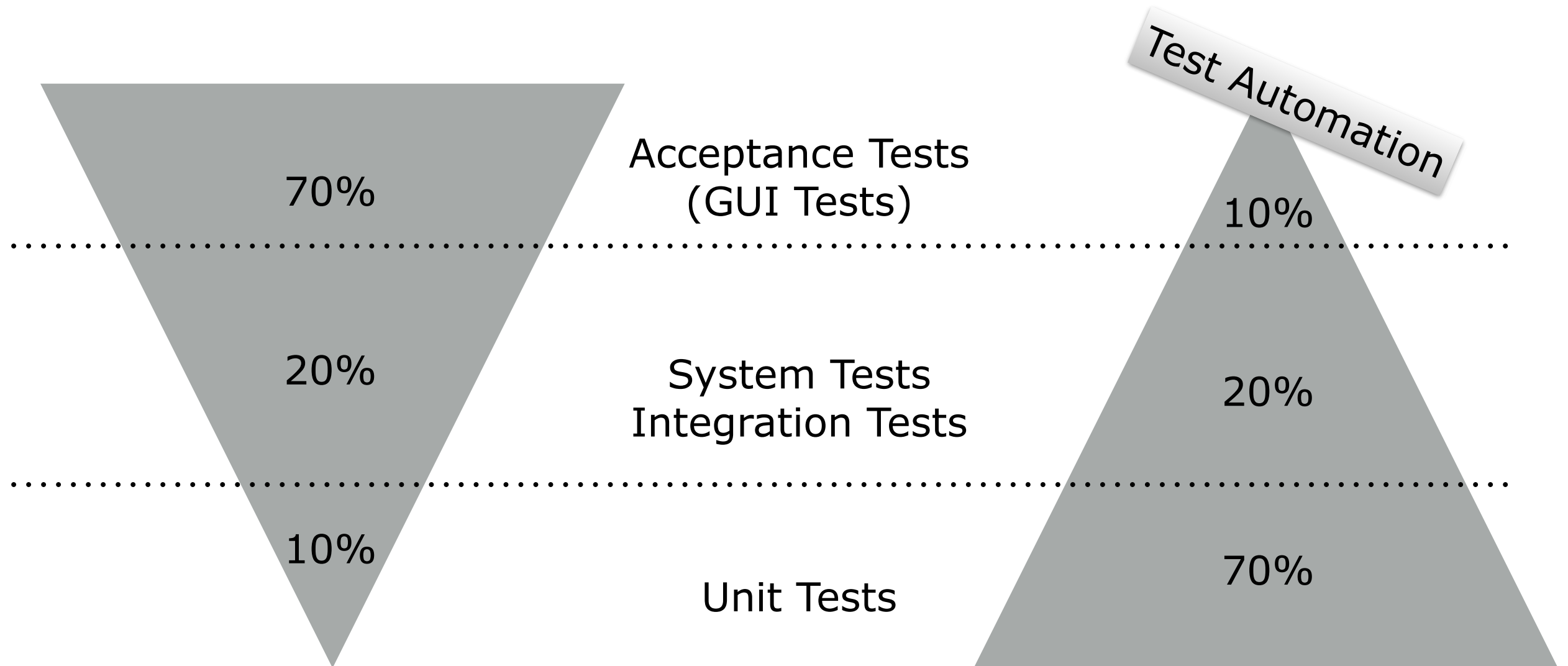
# Tool Support for Testing

- Test Harness
  - + Deterministic tests without any user intervention
    - all input is generated by stubs/all output is absorbed by stubs
    - input/output behaviour is entirely predictable
  - + A test-case is a predicate taking one parameter; an output stream
    - Answers true (component passed test successfully) or false (component did not pass the test + report on the output stream)
    - For each change in requirements, for each bug report
      - > Adapt test cases
        - \* Takes a lot of work: more test code than production code
- Code coverage tools
  - + Instrument code to see which parts are (not) executed by a test suite
    - More coverage  $\neq$  revealing more defects
  - + Mutation coverage
    - Systematically inject faults and execute test suite
- Capture-playback tools
  - + A tool records all UI-actions and their results
  - + Possibility to replay recordings and verify results
    - \* Vulnerable to modifications in UI

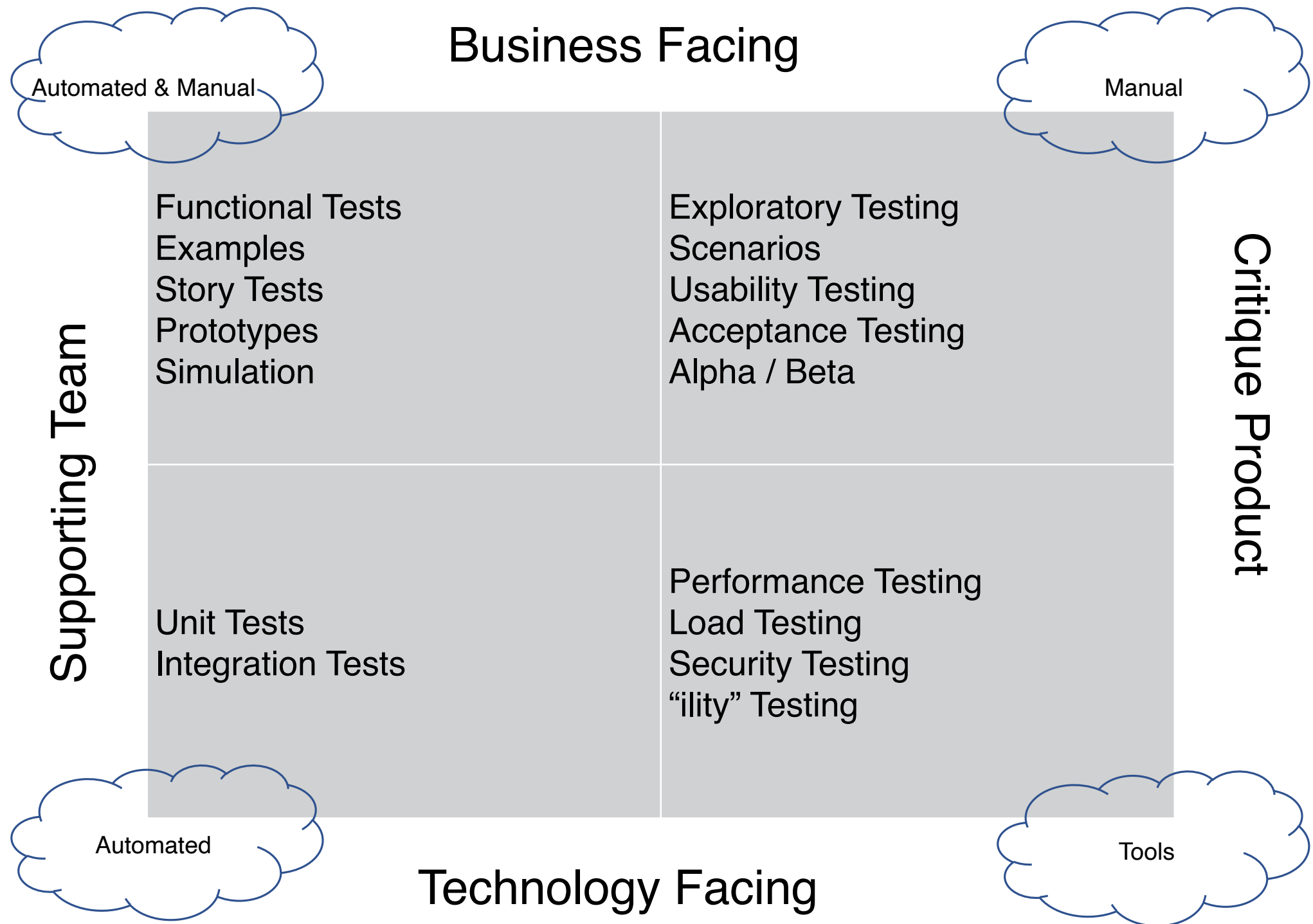
# Agile Testing (DevOps)



# Flipping the V



# 4 Quadrants

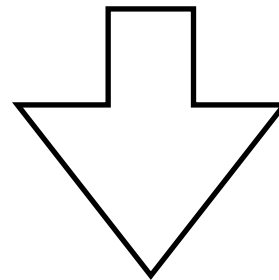


# Definition of Done

As a <user role>  
I want to <goal>  
so that <benefit>.

- ...
- ...
- ...

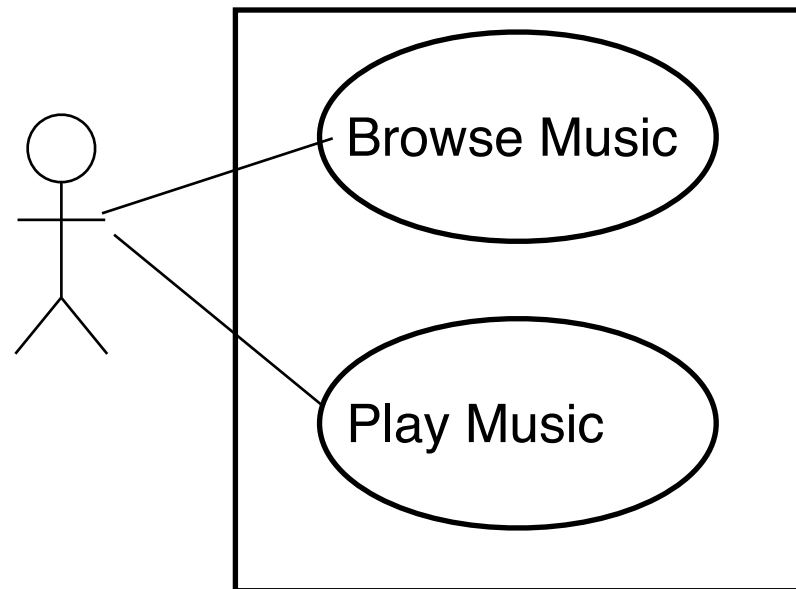
*Conditions of Satisfaction*



|   |                   |
|---|-------------------|
| ✓ | Tested            |
| ✓ | ...               |
| ✓ | Acceptance tested |

Acceptance Test  
Scenarios via FIT tables

# FIT(\*) Tables



## Example: Acceptance Test Cases

<http://fit.c2.com>

| Browse Music |                  |    |
|--------------|------------------|----|
| start        | eg.music.browser |    |
| enter        | library          |    |
| check        | total songs      | 37 |

| Browse Music |        |                      |
|--------------|--------|----------------------|
| enter        | select | 1                    |
| check        | title  | Akila                |
| check        | artist | Toure Kunda          |
| enter        | select | 2                    |
| check        | title  | American Tango       |
| check        | artist | Weather Report       |
| check        | album  | Mysterious Traveller |
| check        | year   | 1974                 |

| Play Music |                   |         |
|------------|-------------------|---------|
| start      | eg.music.Realtime |         |
| press      | play              |         |
| check      | status            | loading |
| pause      | 2                 |         |
| check      | status            | playing |

(\*) FIT = Framework for Integrated Testing



# Tool Support



I WANT YOU  
CAPSTONE PROJECT

FitNesse User Guide

Robert C. Martin,  
Micah D. Martin,  
Patrick Wilson-Welsh &  
FitNesse contributors

Table of Contents

The screenshot shows the FitNesse User Guide website. It has a navigation bar with links for Features, Download, Plug-ins, and User Guide. The main heading is 'FitNesse User Guide' by Robert C. Martin, Micah D. Martin, Patrick Wilson-Welsh, and FitNesse contributors. A 'Table of Contents' link is visible at the bottom left.

JBehave

ABOUT

- What is JBehave?
- Introduction to BDD
- License
- Download
- How to Contribute
- Tutorials
- Users' Experiences

What is JBehave?

JBehave is a framework for **Behaviour-Driven Development** (BDD). BDD is an evolution of test-driven development (TDD) and acceptance-test driven design, and is intended to make these practices more accessible and intuitive to newcomers and experts alike. It shifts the vocabulary from being test-based to behaviour-based, and positions itself as a design philosophy.

The screenshot shows the JBehave website. It features a yellow header with the 'jbehave' logo. A green sidebar contains a table of contents with links to 'What is JBehave?', 'Introduction to BDD', 'License', 'Download', 'How to Contribute', 'Tutorials', and 'Users' Experiences'. The main content area has a yellow background and a section titled 'What is JBehave?' which describes BDD as an evolution of TDD and acceptance-test driven design.

ROBOT  
FRAME  
WORK

The screenshot shows the Robot Framework logo. It consists of the words 'ROBOT', 'FRAME', and 'WORK' stacked vertically in a bold, sans-serif font. The 'ROBOT' and 'FRAME' are in white, while 'WORK' is in black. The logo is set against a teal background.

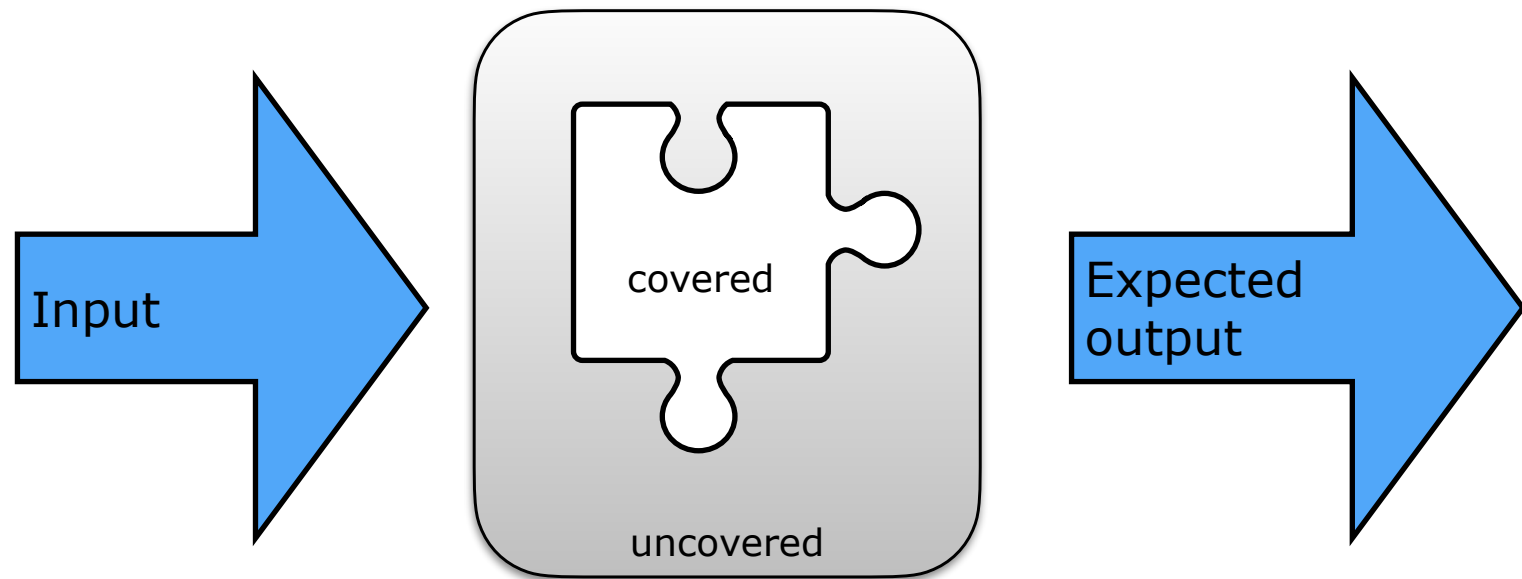
cucumber

Tools & techniques that elevate teams to greatness

The screenshot shows the Cucumber website. It features a white header with the 'cucumber' logo and a hamburger menu icon. The main content area has a background image of a group of people smiling and working together. Overlaid on this image is the text 'Tools & techniques that elevate teams to greatness' in a white, sans-serif font.

# Test Coverage $\neq$ Code Coverage

**Code Coverage**  
line, statement, ...,  
MC/DC, mutation



**Requirement Coverage**  
FIT-tables, FMEA-tables



**Test Coverage**  
Test Plan

- How many of the planned test cases did we specify?
- How many of the specified test cases did we execute?

# Conclusion: Correctness & Traceability & ...

- Correctness
  - + Obviously (are we building the product right)



Besides verifying that the implementation corresponds with the specification, there are other good reasons to test

- Traceability
  - + Naming conventions between tests and requirements specification is a way to trace back from components to the requirements that caused their presence
- Maintainability
  - + Regression tests verify that post-delivery changes do not break anything
- Understandability
  - + If you are a newcomer to the system, reading the test code is a good place to see what it actually does
  - + *Write the tests first*, and you'll be the first user of your component interface, encouraging you to make it very readable



# Summary (i)

You should know the answers to these questions

- What is (a) Testing, (b) a Testing Technique, (c) a Testing Strategy
- What is the difference between an error, a failure and a defect?
- What is a test case? A test stub? A test driver? A test fixture?
- What are the differences and similarities between basis path testing, condition testing and loop testing?
- How many tests should you write to achieve MC/DC coverage? And multiple condition coverage?
- Where do you situate alpha/beta testing in the four quadrants model?
- What are the differences and similarities between unit testing and regression testing?
- How do you know when you tested enough?
- What is Alpha-testing and Beta-Testing? When is it used?
- What is the difference between stress-testing and performance testing?

You should be able to complete the following tasks

- Complete test cases for the Loop Testing example (Loop Testing on page 19).
- Rewrite the binary search so that basis path testing and loop testing becomes easier.
- Write a piece of code implementing a quicksort. Apply all testing techniques (basis path testing, conditional testing [3 variants], loop testing, equivalence partitioning) to derive appropriate test cases.
- Write FIT test cases for the user stories in you Bachelor Capstone Project
- *Apply fuzz testing to the REST-API of your project*



I WANT YOU

CAPSTONE PROJECT

# Summary (ii)

Can you answer the following questions?

- You're responsible for setting up a test program. To whom will you assign the responsibility to write tests? Why?
- Why do we distinguish between several levels of testing in the V-model?
- Explain why basis path testing, condition testing and loop testing complement each other.
- Why is mutation coverage a better criterion for assessing the strength of a test suite?
- *Explain fuzzing (fuzz testing) in your own words.*
- Explain what FIT tables are.
- When would you combine top-down testing with bottom-up testing? Why?
- When would you combine black-box testing with white-box testing? Why?
- Is it worthwhile to apply white-box testing in an OO context?
- What makes regression testing important?
- Is it acceptable to deliver a system that is not 100% reliable? Why (not)?
- Explain the subtle difference between code coverage and test coverage.