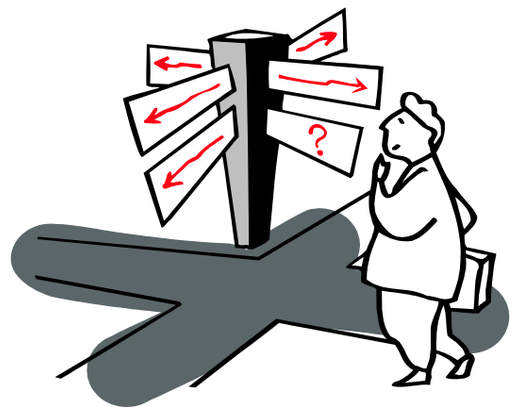
CHAPTER 5 – Testing



Introduction
- When, Why, What & Who ?
- What is “Correct”?
- Terminology

Testing Techniques
- White Box basis path, conditions, loops
- Black Box: equivalence partitioning

Testing Strategies
- Unit & Integration Testing
- Regression Testing
- Acceptance Testing
- More Testing Strategies

Conclusion
- When to Stop?
- Tool Support
- More Good Reasons

Literature

Books

- [Ghez02a] Chapter on “Software Verification” is quite good with plenty of examples of the need for complementary testing techniques. Terminology used here differs from [Pres01a] and [Somm04a]
- [Pres01a] Chapter on “Software Testing Techniques” is very good with lots of concrete examples of the different techniques.
- [Somm04a] Chapter on “Verification and Validation” places Testing in a broader context.

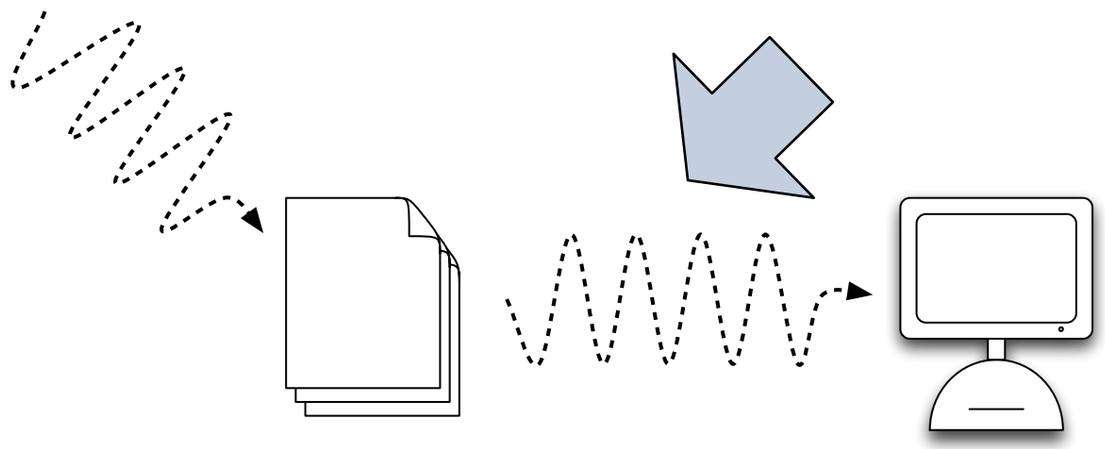
Specific Books

- [Bind00a] Testing Object-Oriented Systems — Models, Patterns, and Tools, Addison-Wesley, 2000.
  + Used as a basis for the master course “Software Testing” [industrial guest speakers — 2 yearly]

Tools

- [xUnit] Free Test Harness for Unit Tests/Regression Tests (C++, Java, Smalltalk, Perl, Python, …)
  + http://www.sourceforge.net/ >>Software Development>> Testing

When to Test ?



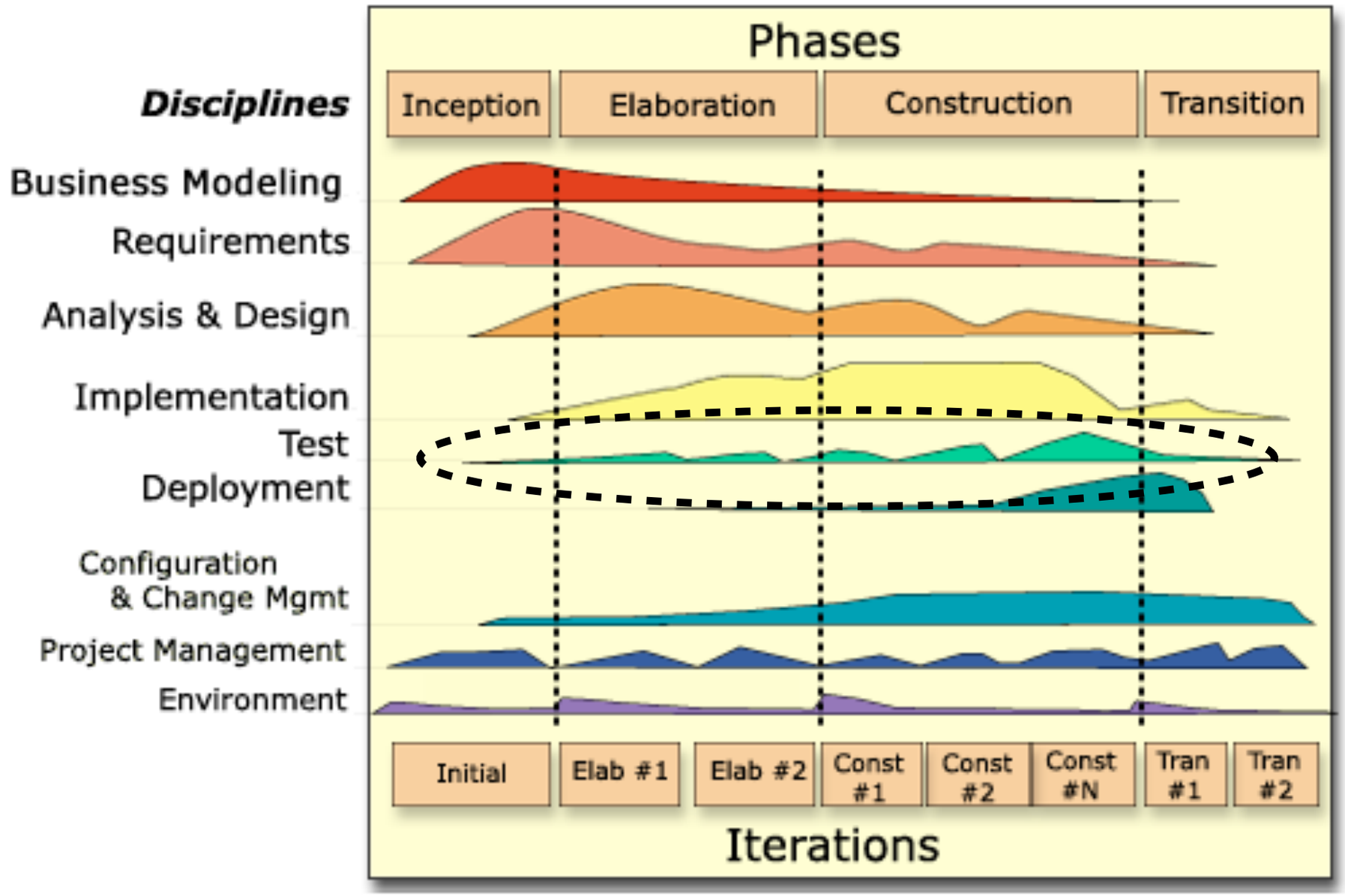
Mistakes are possible (likely !?)

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

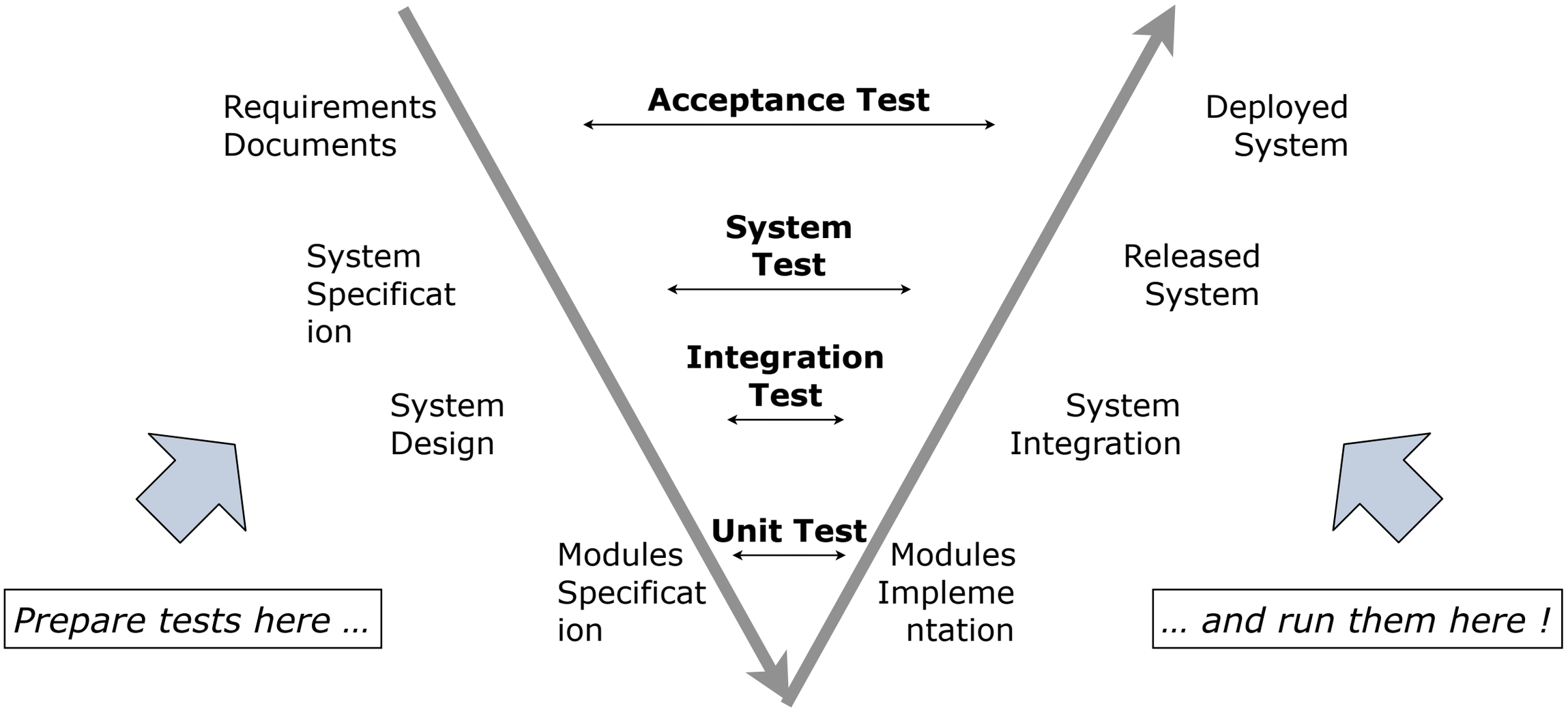
When to Test ? The Unified Process

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



When to Test ? The V-model



SPECIFY & DESIGN WITH TESTABILITY IN MIND

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

What is Testing? (i)

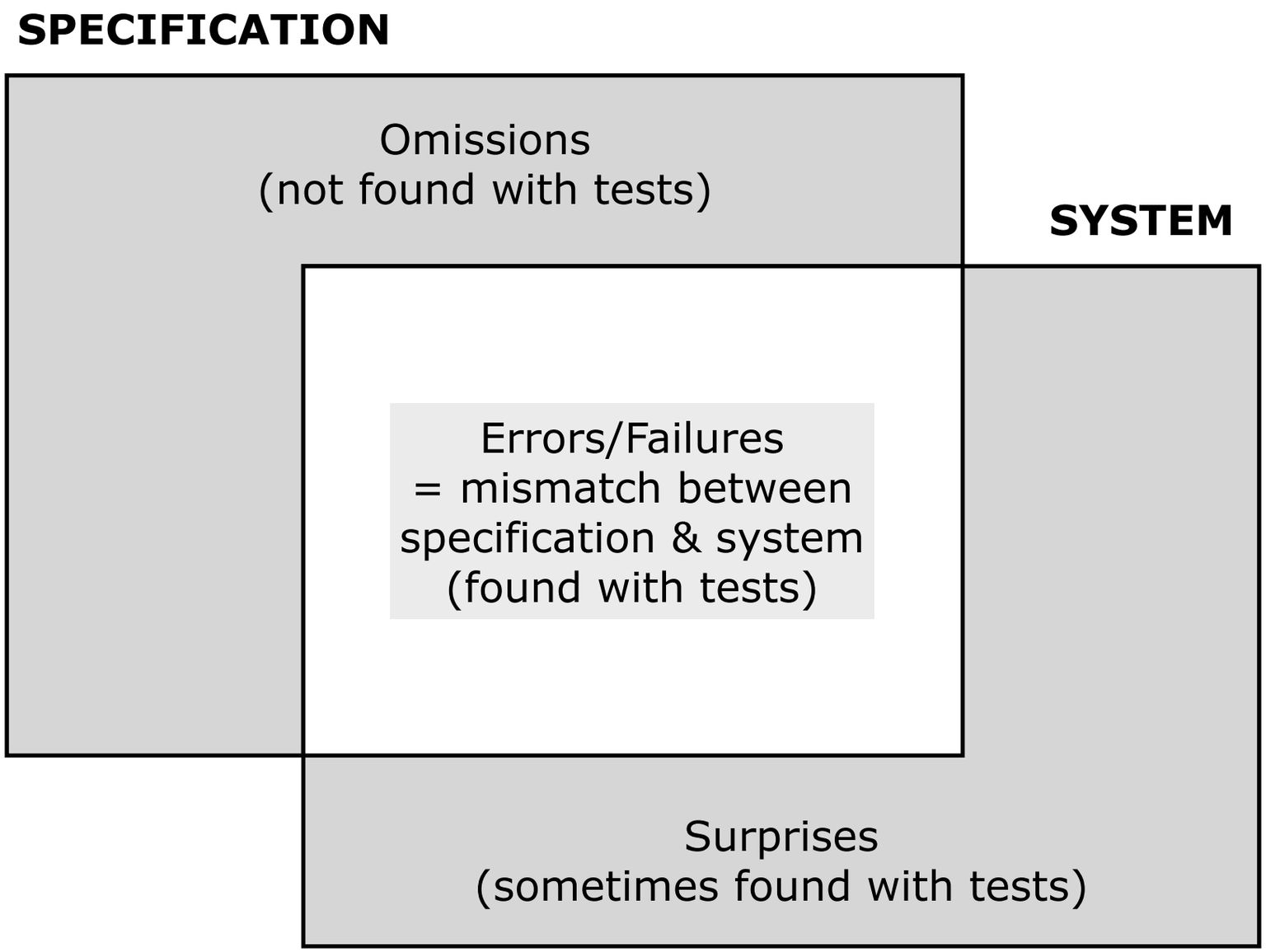
Testing should

- *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/specification/requirements/…
- Testing Strategies
  - + Tell you *when* you should perform *which* testing technique
    - Criterion: *Confidence* that you can safely proceed
    - Next activity = other testing until deployment

What is Testing? (ii)



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)

What is “Correct” ?

- See [Ghez02a] -- 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 undecideable

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 (i)

Avoid the term "Bug"

- Implies mistakes creeping into the software from the outside
- imprecise because mixes various "mistakes"

To be more precise (Terminology not standard!)

Defect / Fault (NL = DEFECT, GEBREK, NALATIGHEID)

- A design or coding mistake that may cause abnormal behaviour
  + abnormal behaviour = deviations from specification (incl. surprises !)

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

Terminology (ii)

Component

- A part of the system that can be isolated for testing
  + an object, a group of objects, one or more subsystems

Test Case

- A set of inputs and expected results that exercise a component with the purpose of causing errors and failures
  + a predicate method 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

- A partial implementation of components on which the tested component depends
  + dummy code that provides the necessary input values and behaviour to run the test cases

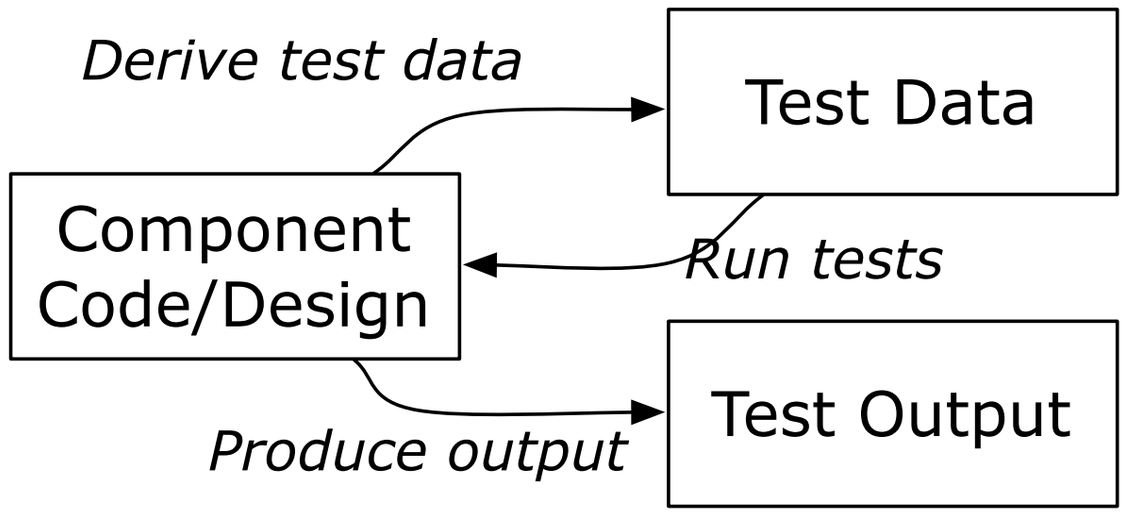
Test Driver

- A partial implementation of a component that depends on the tested component
  + a “main()” function that executes a number of test cases

White Box Testing

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

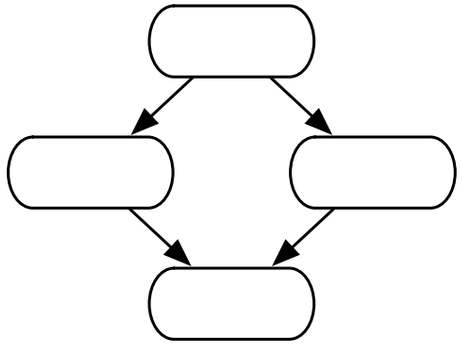
- Treat a component as a "white box", i.e. you can inspect its internal structure
- Internal structure is also design specs; e.g. sequence diagrams, state charts, …
- Derive test cases to maximize coverage of that structure, yet minimize number of test cases



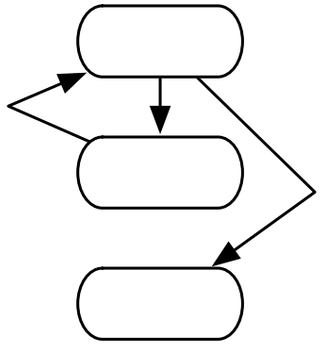
- Coverage criteria
  - + every statement at least once
  - + all portions of control flow (= branches) at least once
  - + all possible values of compound conditions at least once
  - + all portions of data flow at least once
  - + all loops, iterated at least 0, once, and N times

Basis Path Testing

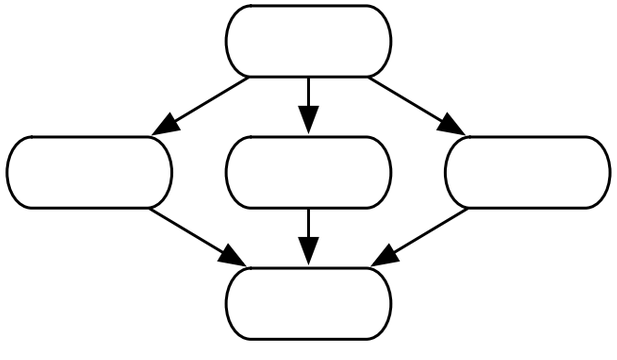
- 1. Draw a control flow graph
  + nodes = sequences of non branching statements (assignments, procedure calls)
  + edges = control flow



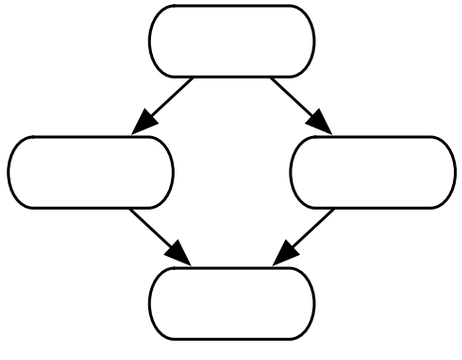
if-then-else
[cc = 2]



while
[cc = 2]



case-of
[cc = 3]



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

- 2. Compute the Cyclomatic Complexity
  = #(edges) - #(nodes) + 2
  = number of binary conditions + 1 = # regions
- 3. Determine a set of *independent* paths (= at least one new edge in every path)
  [*independent* paths form a mathematical vector *basis* for the complete graph⇒ name]
  + 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

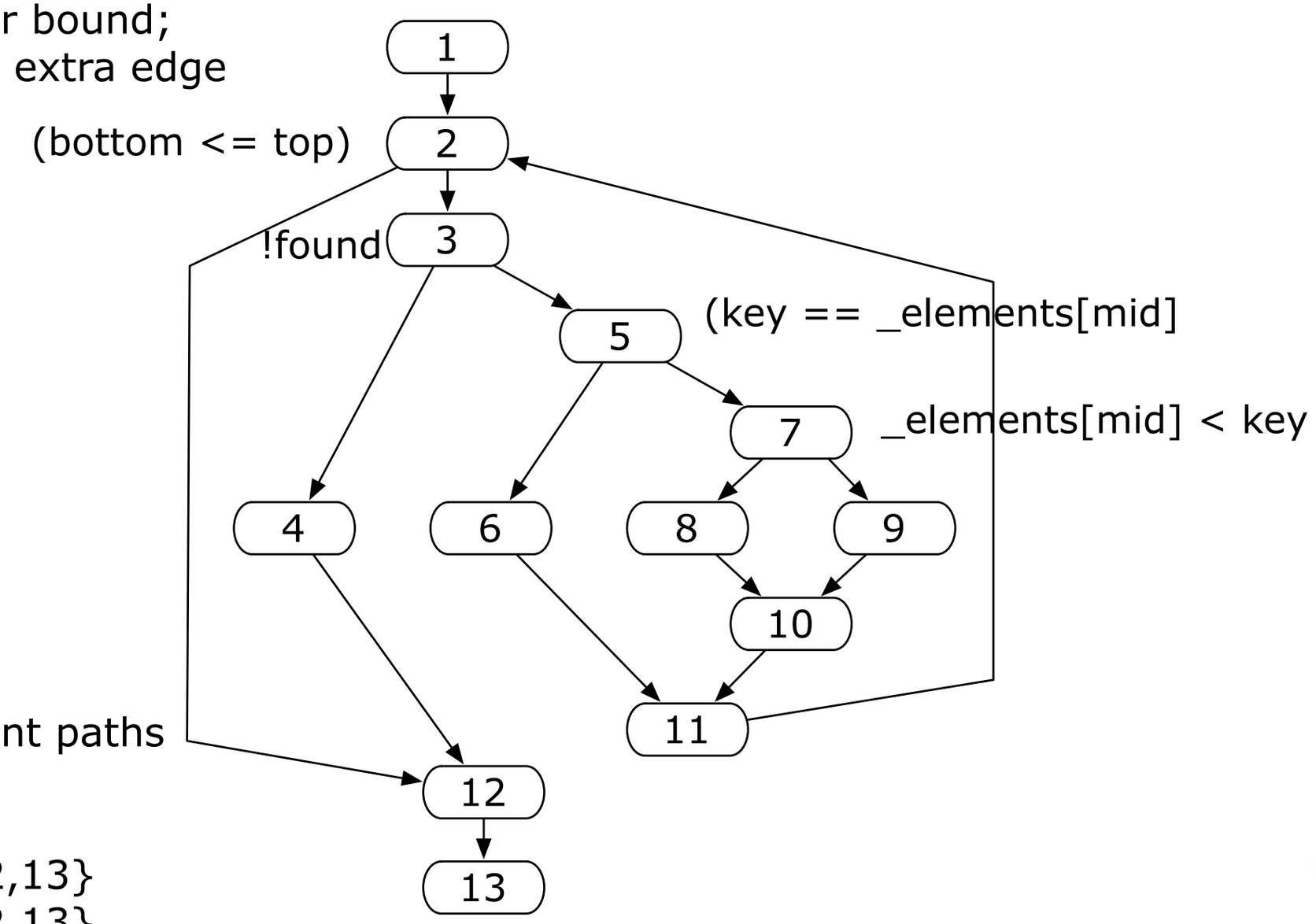
```
public boolean find(int key) {                          //Binary Search
   int bottom = 0;                                      // (1)
   int top = _elements.length-1;
   int lastIndex = (bottom+top)/2;
   int mid;
   boolean found = key == _elements[lastIndex];
   while ((bottom <= top) && !found) {                  // (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)
}
```

Example - Flow Graph

- set of independent paths of a flow graph ⇒ 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} {1,2,3,*5,7,9,10,11,2,3,* 5,6,11,2,12,13}	_elements = [1, 5, 9]; key = 1 *actual path is not intended path*	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

Basis Path Testing: Evaluation

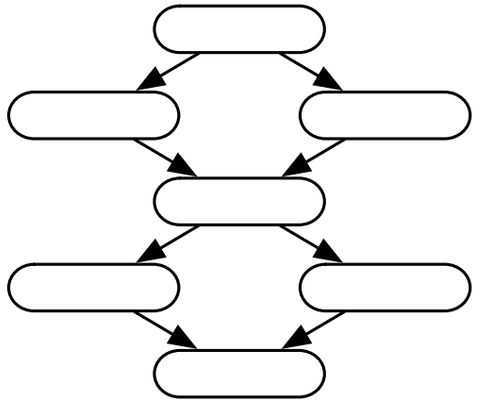
Pros

- 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

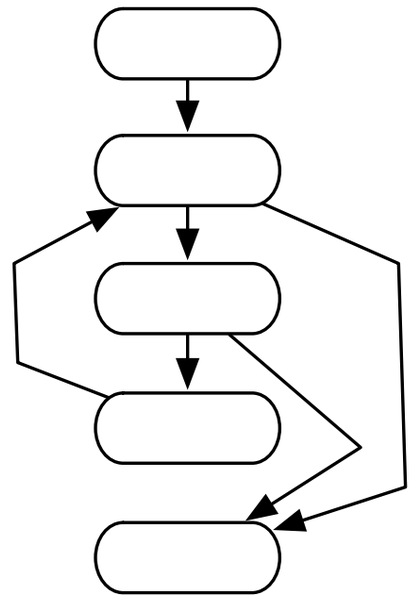
- not all cc independent paths will cover all statements and all branches (see "Summary"; Perform basis path with a nested conditional of 2 levels deep.)

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 + allmost "true/false" for each occurrence of a < b, a <= b, a == b, a <> b, a >= b, 3 tests
  - ➡ test cases {a < b; a == b; a > b}

Condition Testing - Test Cases

Condition Coverage

- line 2: (x <> y) && (x <> 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 = $\log_2$(size (_elements)) = $\log_2$(16) = 4

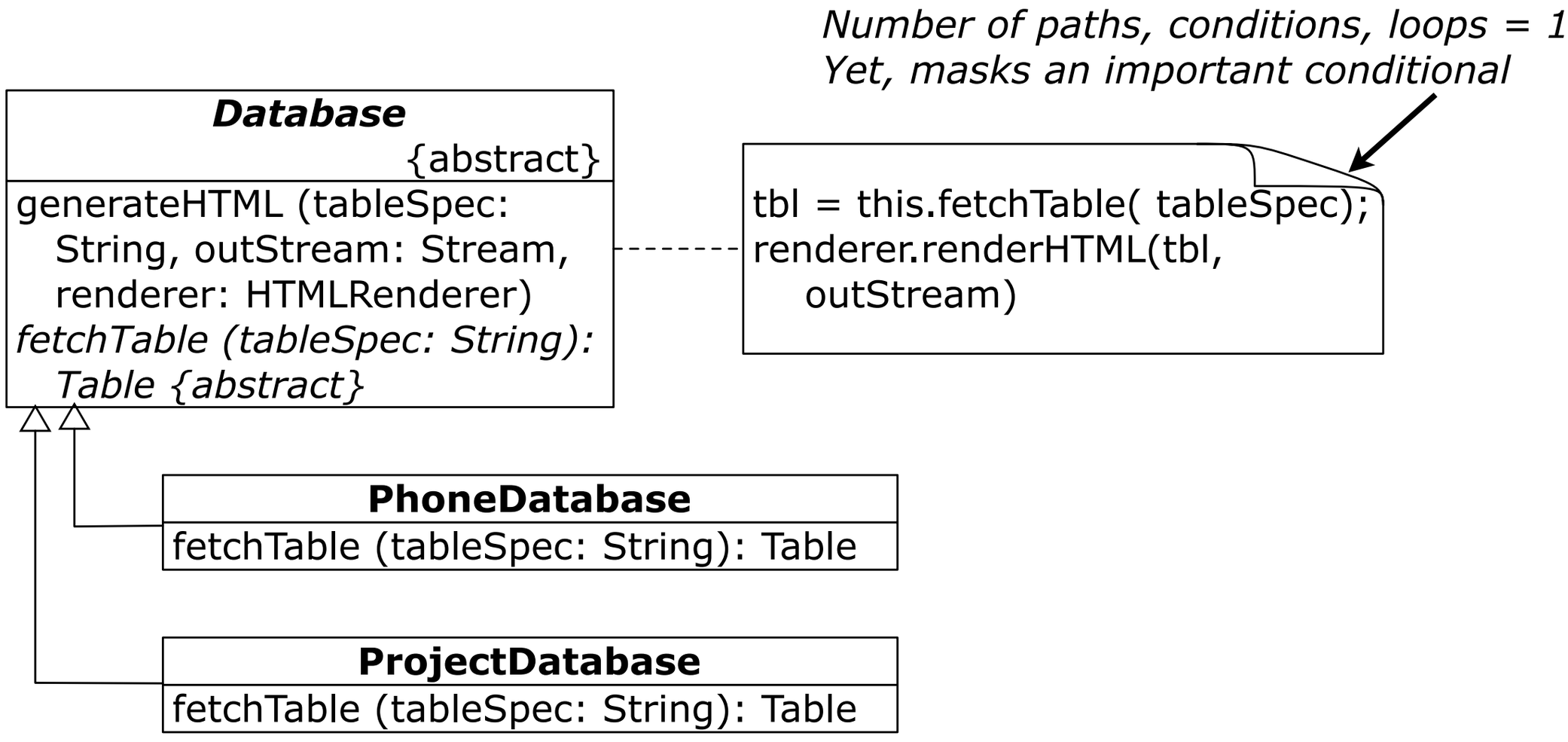
Path	Input	Output
skip the loop	_elements = [1, 3, ..., 29, 31]; key = ...	
1 pass through the loop	_elements = [1, 3, ..., 29, 31]; key = ...	
2 passes through the loop	_elements = [1, 3, ..., 29, 31]; key = ...	
m passes where 2 < m < n	_elements = [1, 3, ..., 29, 31]; key = ...	
n-1	_elements = [1, 3, ..., 29, 31]; key = ...	
n passes	_elements = [1, 3, ..., 29, 31]; key = ...	
n+1 passes	_elements = [1, 3, ..., 29, 31]; key = ...	

The actual test cases are left as an exercise

White Box Testing and Objects (i)

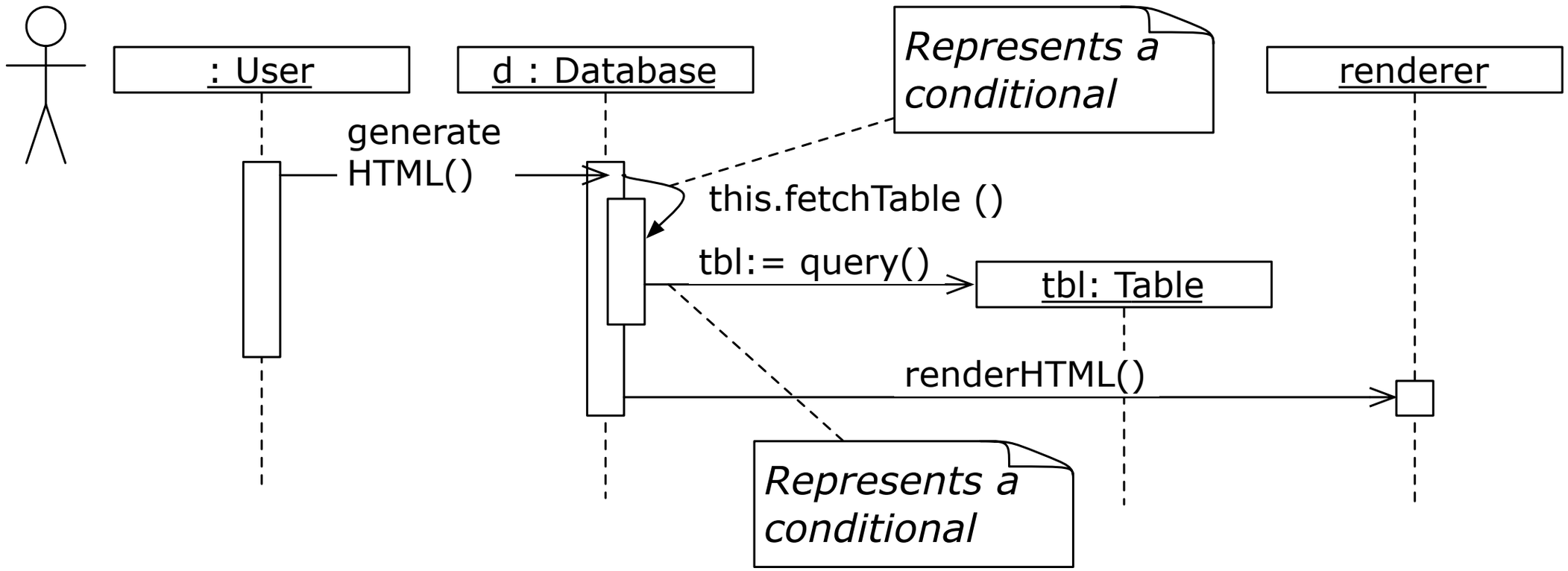
Pure white box testing is less relevant in an object-oriented context.

- Internal structure embedded in object compositions and polymorphic method invocations



White Box Testing and Objects (ii)

… but: sequence & collaboration diagrams may serve better



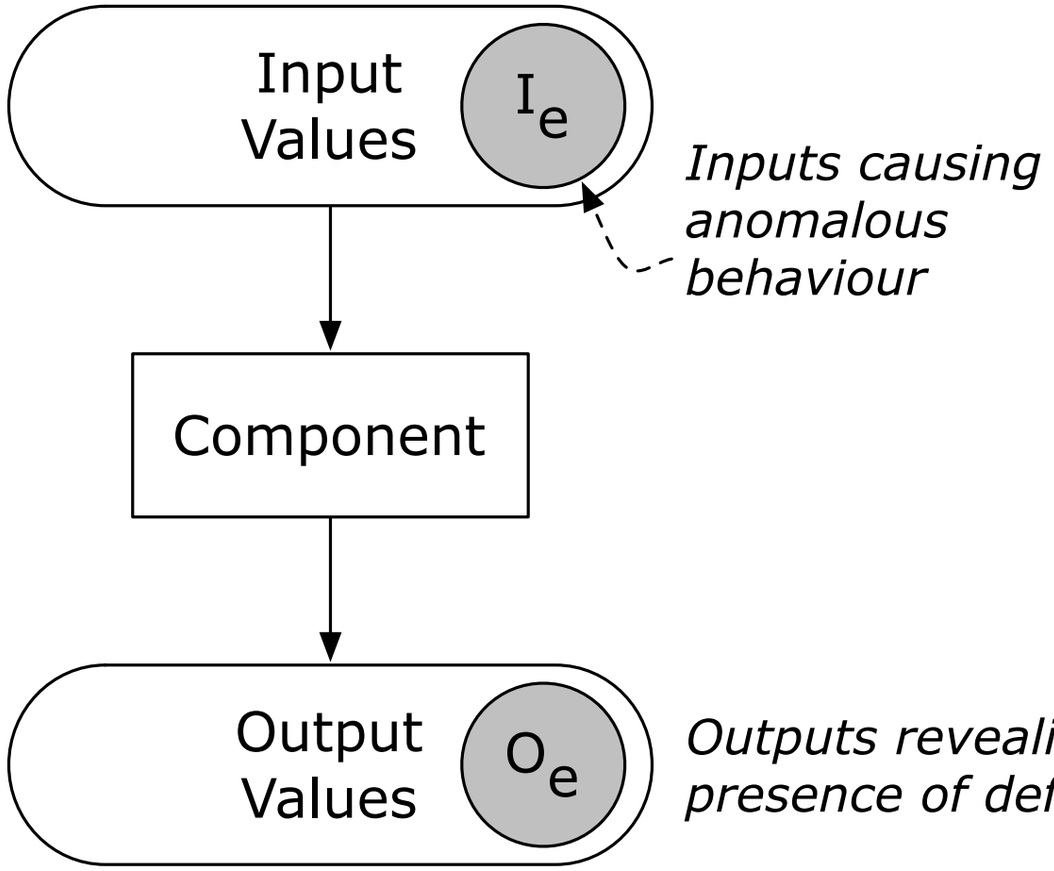
⇒ Identify polymorphic messages representing a conditional

⇒ plug-in instances of appropriate subclasses to exercise branches

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

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 => 1 valid (in the range) and 2 invalid equivalence classes
  + a value => 1 valid (= value) and 2 invalid equivalence classes
  + a set => 1 valid (in the set) and 1 invalid equivalence class
  + a boolean => 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?

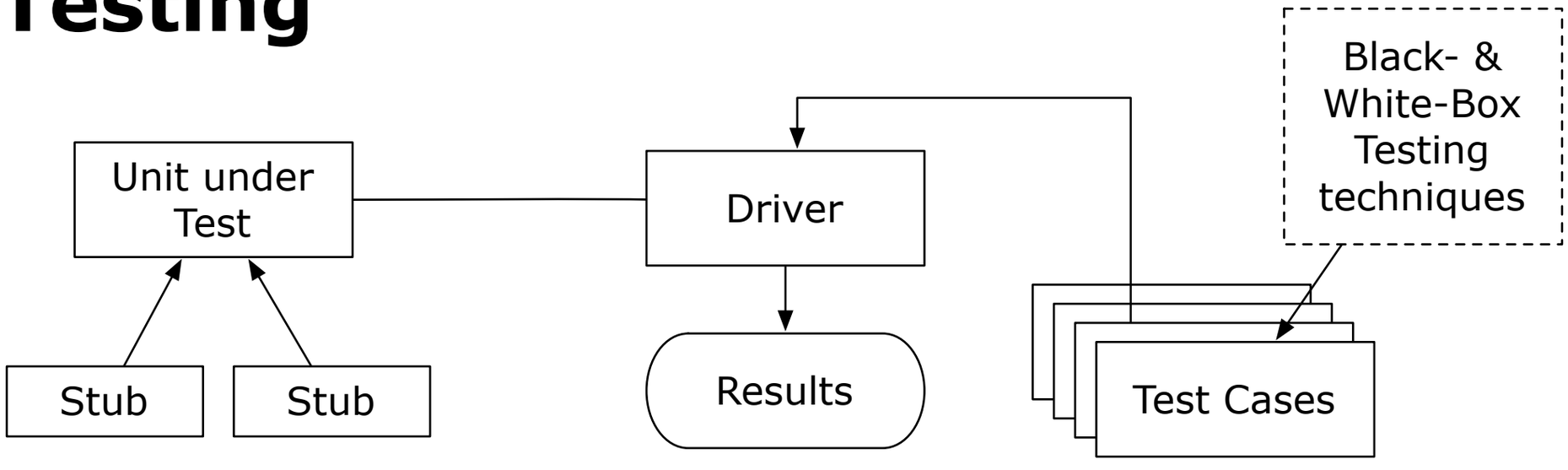
➡ leads to further subdivisions
(not all combinations make sense)

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
…	…	…

Unit Testing

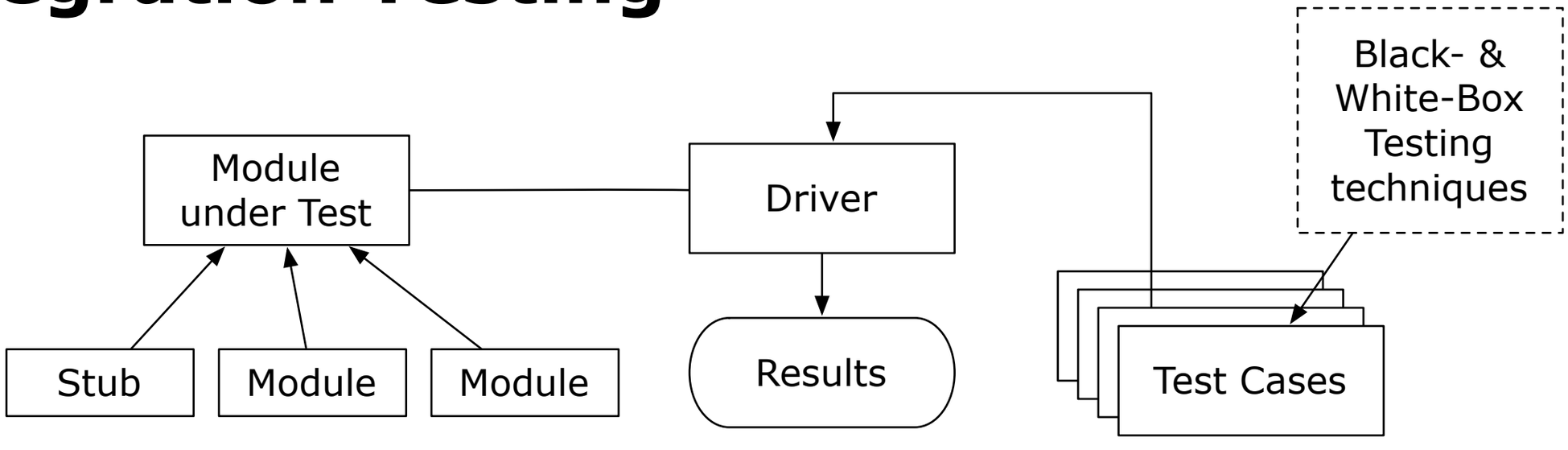


- Why?
  + Locate small errors (= within a unit) fast
- Who?
  + Person developing the unit writes the tests.
- When ? At the latest when a unit is delivered to the rest of the team
  + No test ⇒ no unit
  + Test drivers & stubs are part of the system ⇒ configuration management

*** **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 then 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

More Testing Strategies

Recovery 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

When to Stop ?

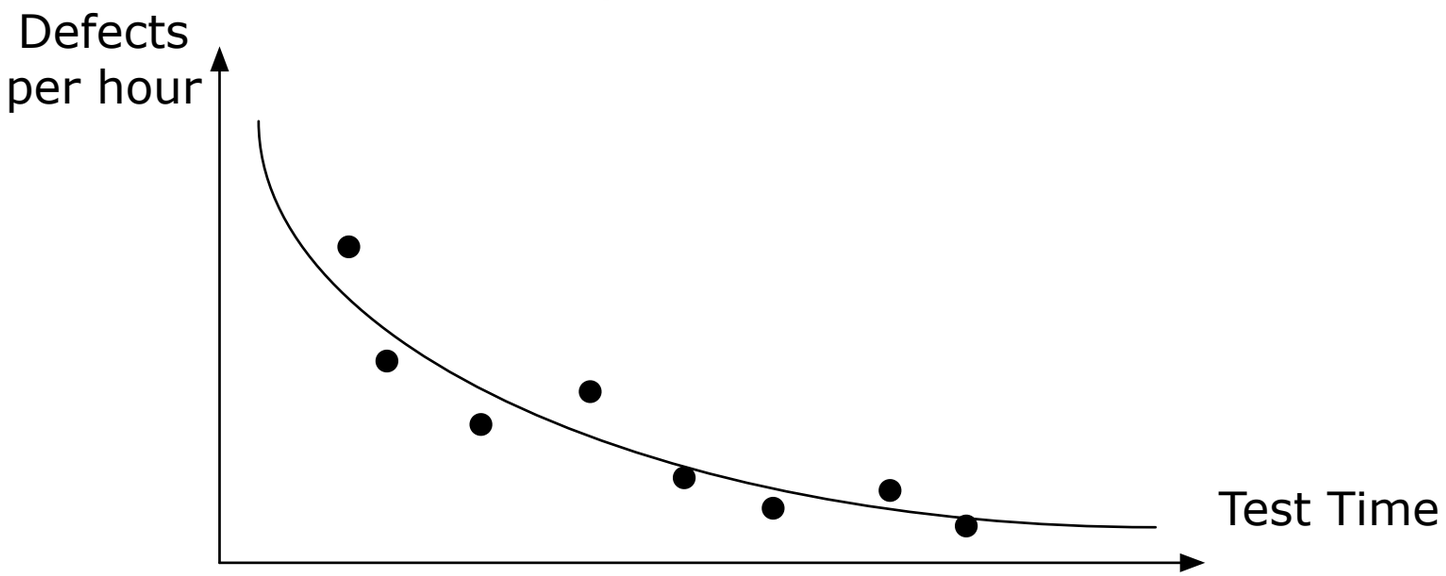
When are we done testing? When do we have enough tests?

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 regression 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

- [http://www.sourceforge.net/ >>Software Development>> Testing]
- 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 ≠ revealing more defects

Capture-playback tools

- A tool records all UI-actions and their results
- Possibility to replay recordings and verify results

  ➡ Vulnerable to modifications in UI

Conclusion: Correctness & Traceability & …

Correctness

- Obviously

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 fault?
- What is a test case? A test stub? A test driver?
- What are the differences and similarities between basis path testing, condition testing and loop testing?
- 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.

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 ?
- Explain why basis path testing, condition testing and loop testing complement each other.
- 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?