Software Engineering VU [194.020]

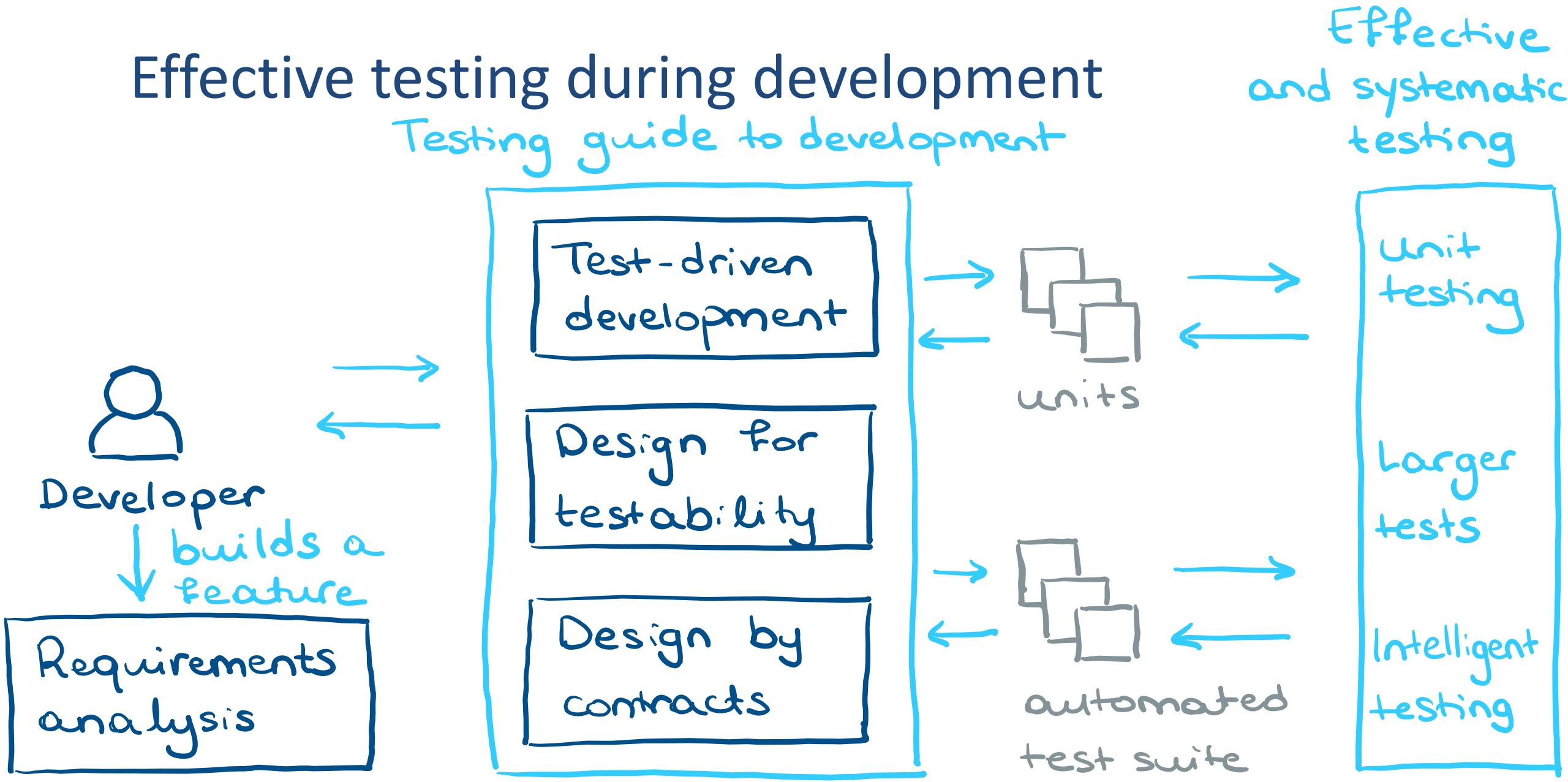
Maria Christakis

TU Wien

https://mariachris.github.io

Summary

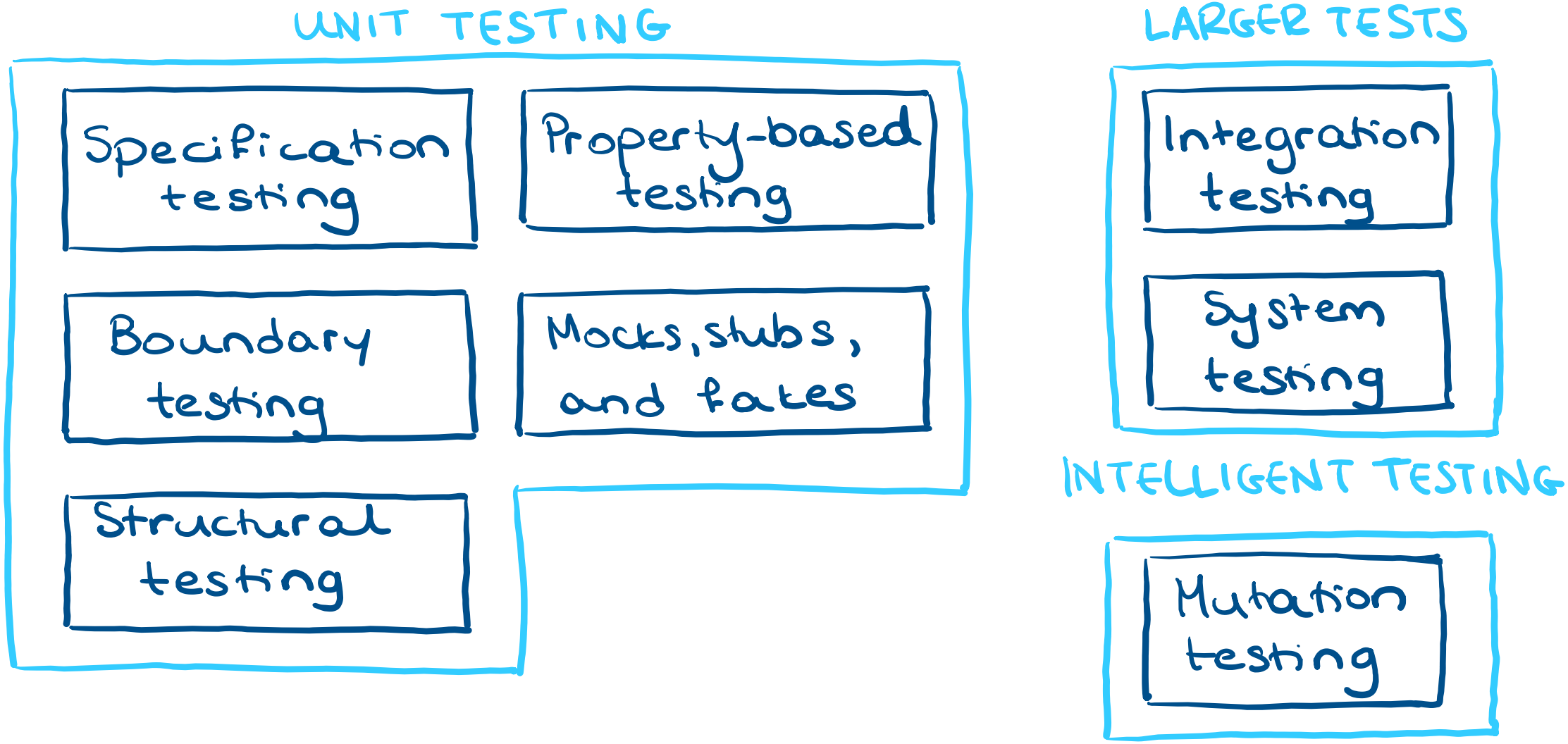
Effective testing during development



Testing

- Specification-based and boundary testing
- Structural testing
  - Control flow coverage

Effective and systematic testing



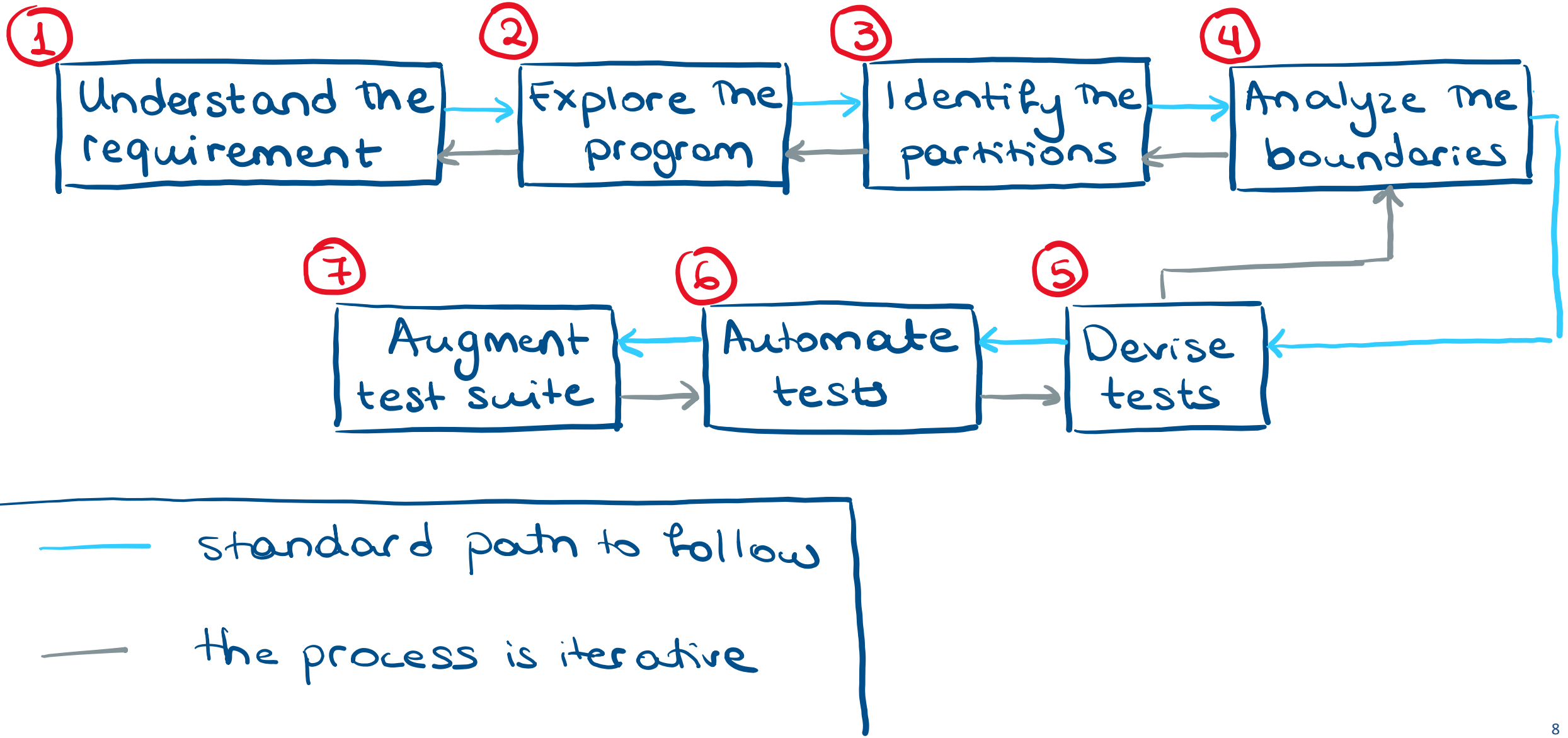
Specification-based testing: Definition

Specification-based testing uses the program requirements as testing input

Once in testing mode, it is recommended to use this technique first

It is a 7-step approach to systematically derive tests based on a specification

Specification-based testing: Approach



Example

Method: `substringsBetween`

Searches a string for substrings delimited by a start and end tag, returning all matching substrings in an array

- `str` – The string containing the substrings; null/empty returns null/empty
- `open` – The string identifying the start of the substring; empty returns null
- `close` – The string identifying the end of the substring; empty returns null

The program returns a string array of substrings, or null if there is no match

Example

Method: `substringsBetween`

Searches a string for substrings delimited by a start and end tag, returning all matching substrings in an array

If `str = "axcaycazc"`, `open = "a"`, and `close = "c"`, the output should be an array containing `["x", "y", "z"]`

Understand the requirement

- What should the program do?
  - What should it not do?
  - Does it handle specific corner cases?
- What are the inputs?
  - Their types and their domain
- What are the outputs?
  - Their type and their domain

Explore the program

- Play with the program to increase your understanding (if you didn’t write it)
  - Call the program with different inputs and see what it produces as output
  - Stop when you have a clear mental model of how the program works
  - This is not yet testing

Explore the program

- Play with the program to increase your understanding (if you didn't write it)
  - Call the program with different inputs and see what it produces as output
  - Stop when you have a clear mental model of how the program works
  - This is not yet testing

- `str = "abcd", open = "a", close = "d"`
  - `Return ["bc"]`

- `str = "abcdabcdab", open = "a", close = "d"`
  - `Return ["bc", "bc"]`

- `str = "aabcddaabfddaab", open = "aa", close = "dd"`
  - `Return ["bc", "bf"]`

3 Identify the partitions

- The input “abcd” with open tag “a” and close tag “d”, which makes the program return “bc”, makes the program behave in the same way as the input “xyzw” with open tag “x” and close tag “w”, which makes the program return “yz”
  - We changed the letters but expect the program to do the same thing for both inputs
  - Each of these cases represents the same class or partition of inputs
  - We say that these two inputs are equivalent

Identify the partitions

- A systematic way to identify the partitions is the following:
  - Look at each input variable individually, and explore its type and the range of values it can receive
  - Look at how each variable may interact with another as variables often have dependencies or put constraints on each other
  - Look at all possible types of outputs

Identify the partitions

Looking at each input variable individually:

- `str`
  - null string
  - empty string
  - String of length 1
  - String of length > 1
- Same for open
- Same for `close`

Identify the partitions

Looking at combinations of input variables:

- `str` contains neither `open` nor `close`
- `str` contains `open` but not `close`
- `str` contains `close` but not `open`
- `str` contains both `open` and `close`
- `str` contains both `open` and `close` multiple times

Identify the partitions

Looking at possible outputs:

- Array of strings
  - Null array
  - Empty array
  - Single item
  - Multiple items
- Each individual string
  - Empty string
  - Single character
  - Multiple characters

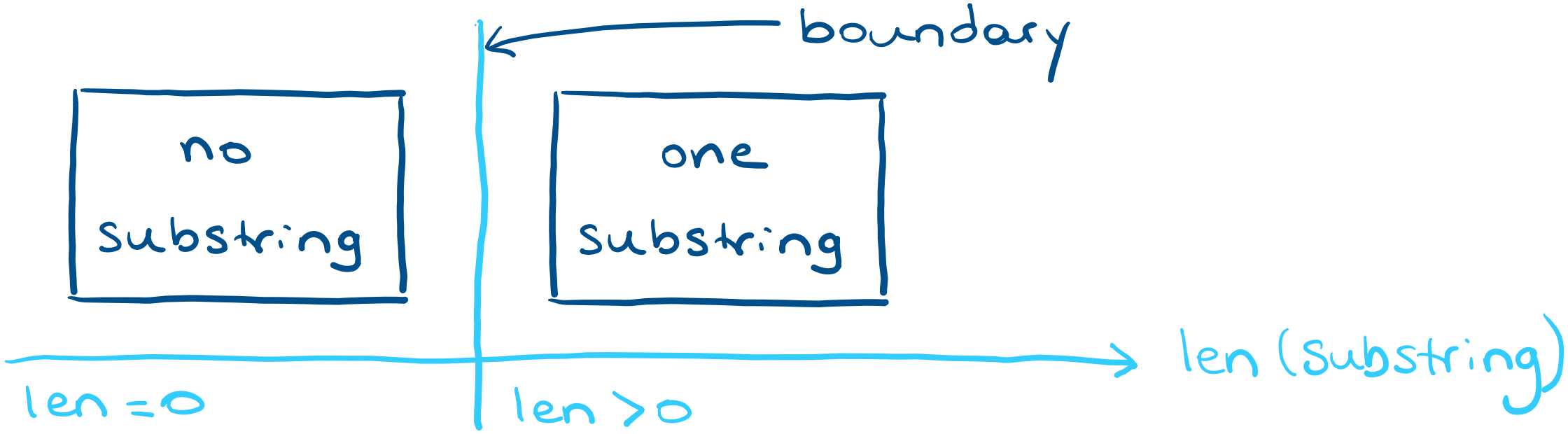
Analyze the boundaries

- Boundary testing
  - Making the program behave correctly when inputs are near a boundary
  - Think of mistakes with using > instead of >=

- Identify boundaries and test what happens to the program when inputs go from one boundary to the other
  - For each boundary, test the following points: the on point, which is on the boundary, and one or more off points, which are points closest to the boundary and belong to partitions the on point does not belong to

4 Analyze the boundaries

Consider the case where open and `close` are in `str`:



- `str` contains both open and `close` with no characters between them

- `str` contains both open and `close` with characters between them

Devise tests

- The idea is to combine all partitions we devised for each of the inputs
  - 4 x 4 x 4 x 5 = 320 tests
- Pragmatically decide which partitions should be combined
- A common strategy is to test exceptional behavior only once and not combine it with other partitions
  - E.g., consider the null string partition – what would we gain from combining the null string with open being null, empty, length = 1 and length > 1 and so on?

Automate tests

- Write automated tests for all test cases that we just devised
  - Identify concrete input values (sometimes there are values we do not care about)
  - Have a clear expectation of what the program should do (the output)
  - Ensure the tests are easily identifiable in case one fails

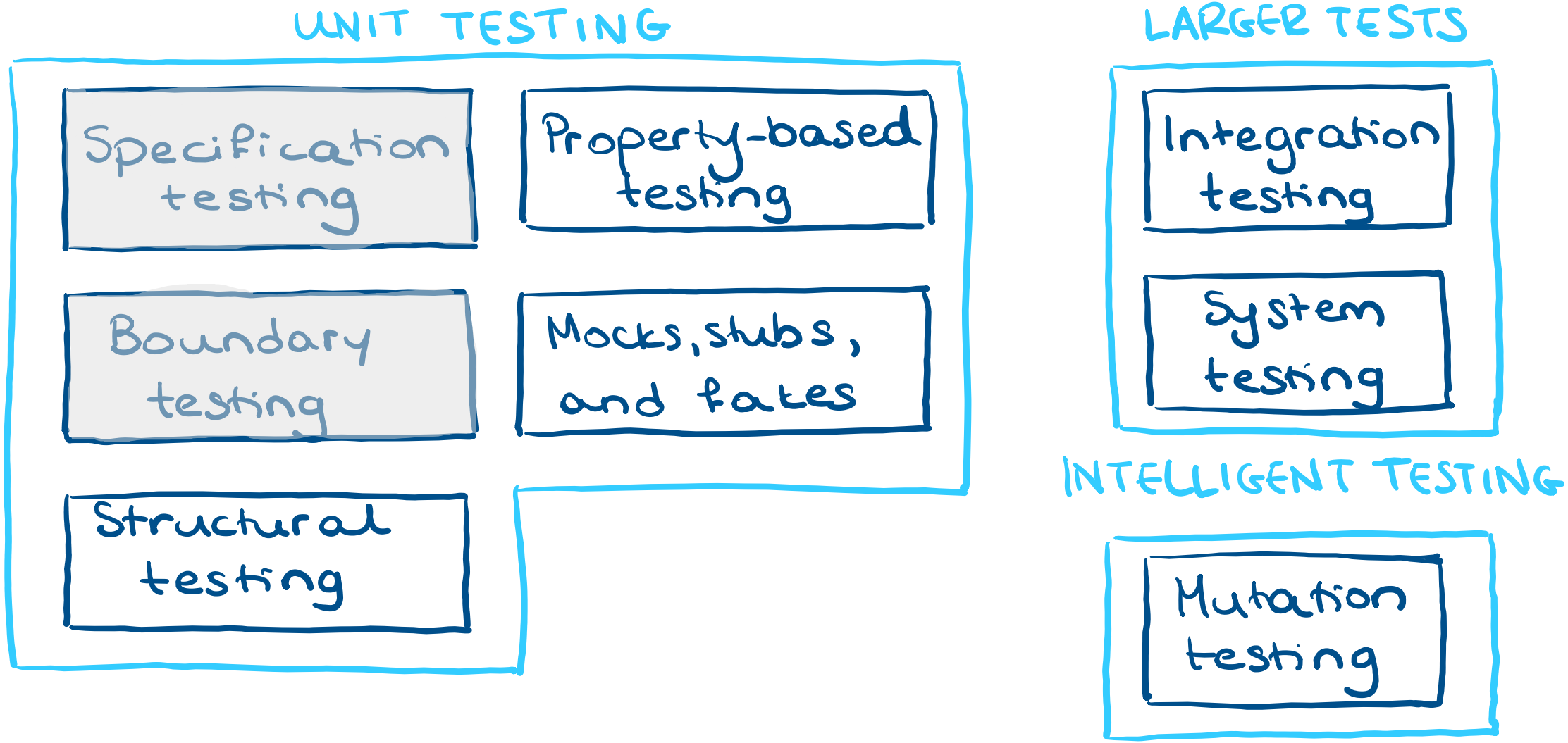
Augment test suite

- Develop interesting variations, if necessary
  - Try strings with spaces?
  - Try open and close tags with spaces?

Testing

- Specification-based and boundary testing
- Structural testing
  - Control flow coverage

Effective and systematic testing



Structural testing: Definition

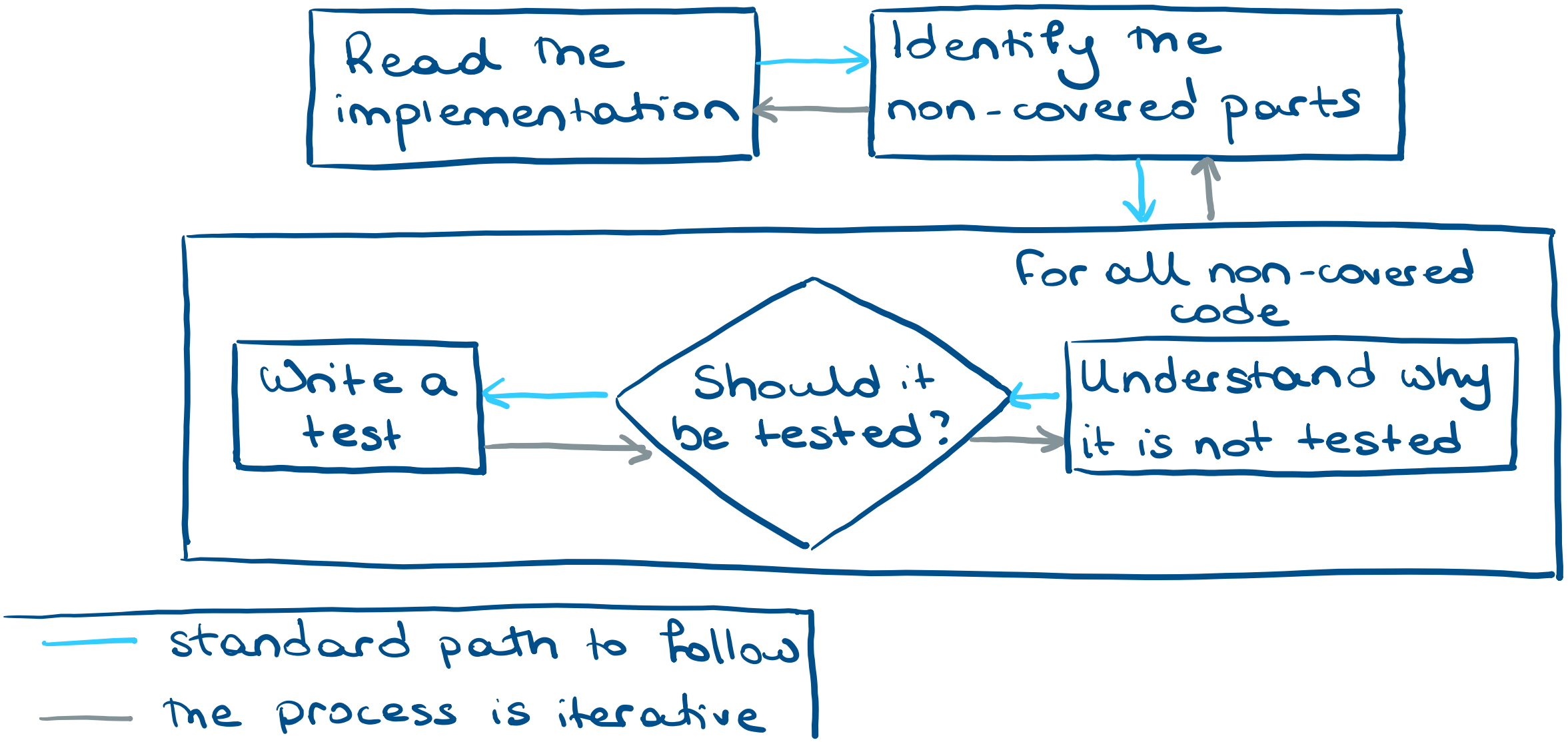
Using the structure of the source code to guide testing is called structural testing

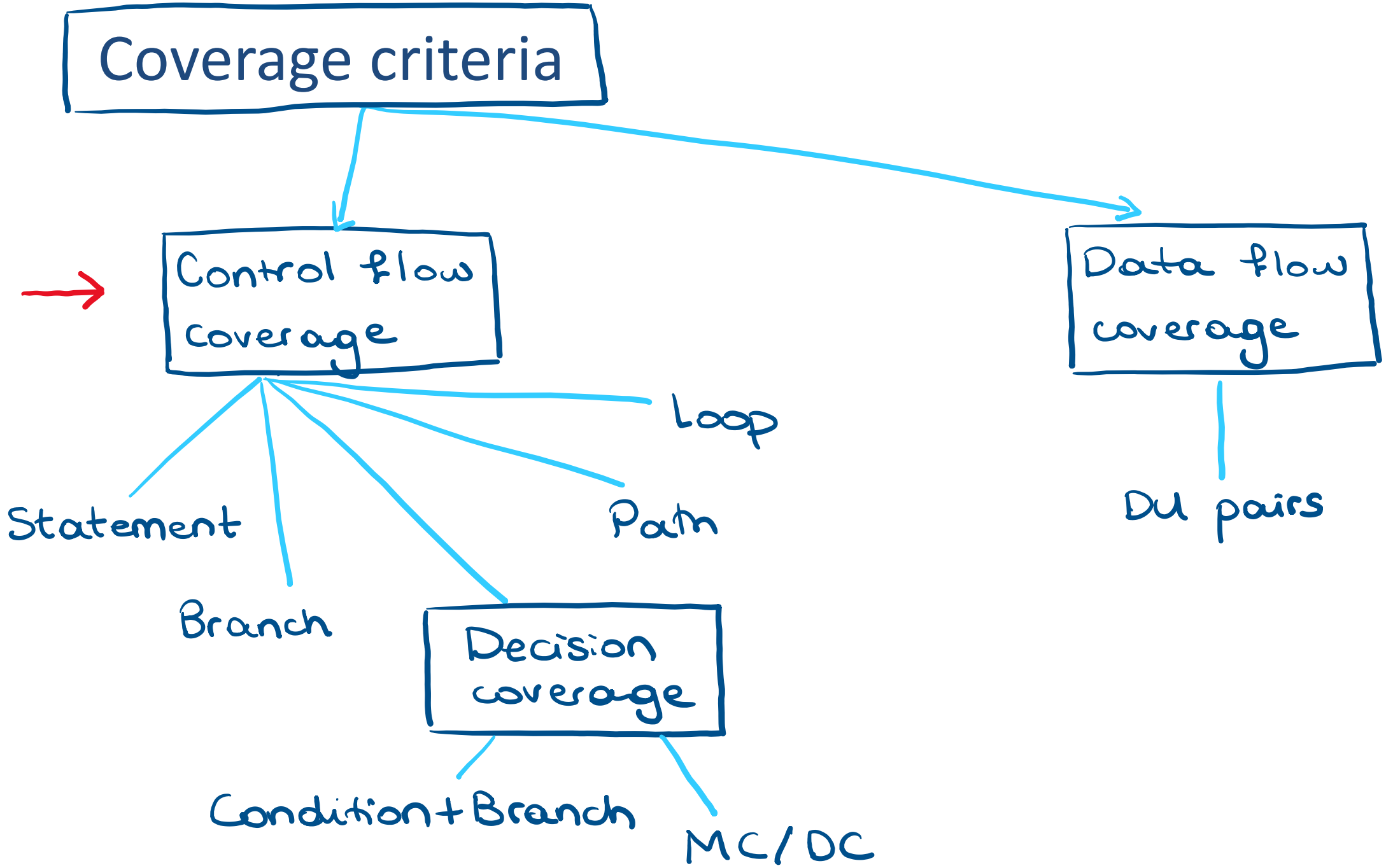
Structural testing complements the test suite devised with specification-based and boundary testing

With code coverage tools, structural testing identifies the parts of the code that are not already covered by the existing test suite (example coverage report)

Understanding structural testing means understanding coverage criteria

Structural testing: Approach




Coverage criteria
Control flow coverage
Data flow coverage
Loop
Statement
Path
DU pairs
Branch
Decision coverage
Condition+Branch
MC/DC

Basic blocks

- A basic block is a sequence of instructions such that the code in a basic block:
  - Has one entry point – no code within the basic block is the destination of a jump instruction in the program
  - Has one exit point – only the last instruction may cause the program to execute code in a different basic block
- When the first instruction in a basic block is run, the rest of the instructions necessarily run once

Basic blocks – Example

```
public void SortAscending(int[] a) {
  if (a == null || a.length < 2)
    return;
  int i;
  for (i = 0; i < a.length – 1; i++) {
    if (a[i] < a[i + 1])
      break;
  }
  if (i >= a.length – 1)
    return;
  QSort(a, 0, a.length);
}
```

Is this method correct?

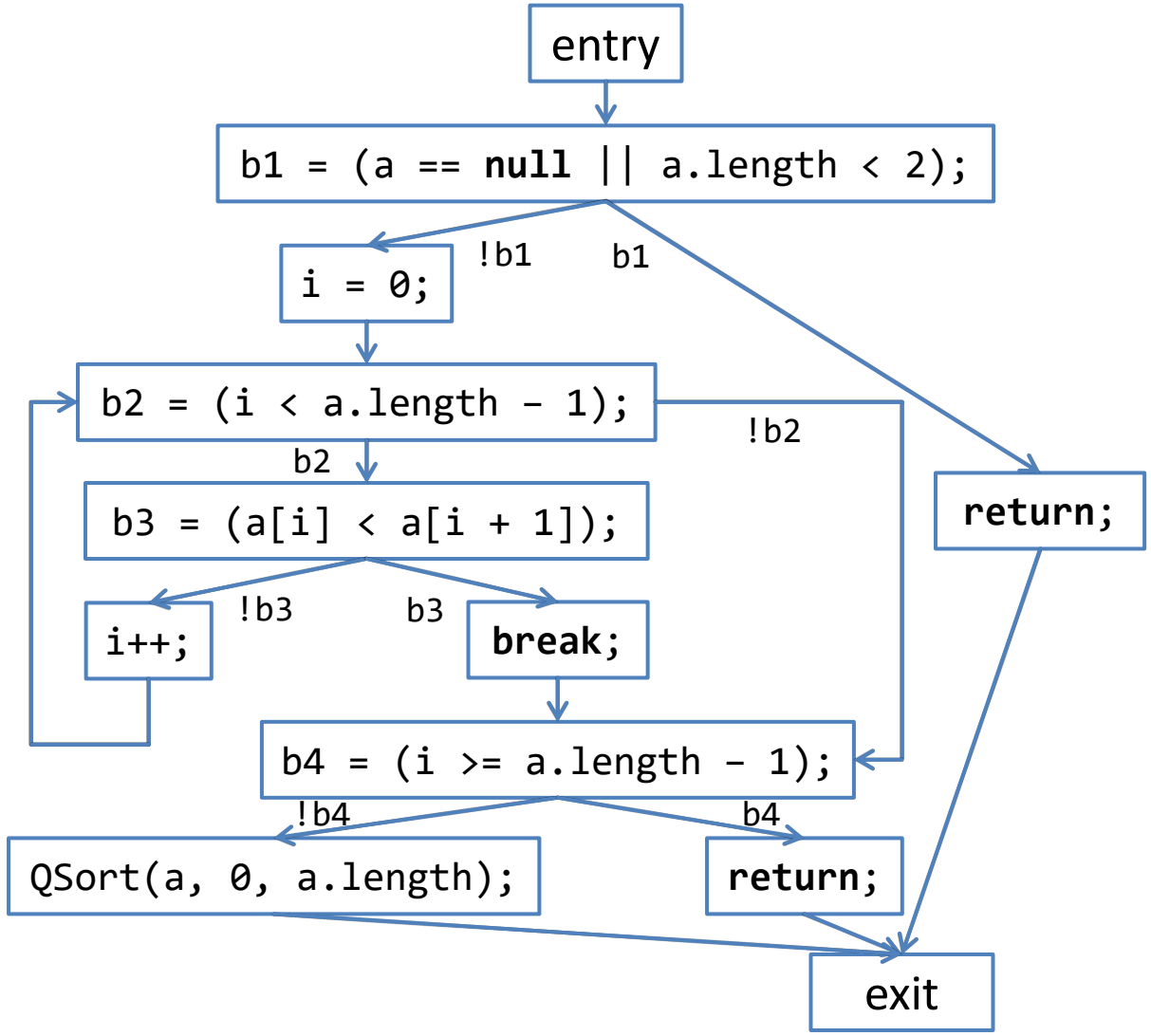
Basic blocks – Example

```
public void SortAscending(int[] a) {
  if (a == null || a.length < 2)
    return;
  int i;
  for (i = 0; i < a.length – 1; i++) {
    if (a[i] < a[i + 1])
      break;
  }
  if (i >= a.length – 1)
    return;
  QSort(a, 0, a.length);
}
```

Intraprocedural control flow graph

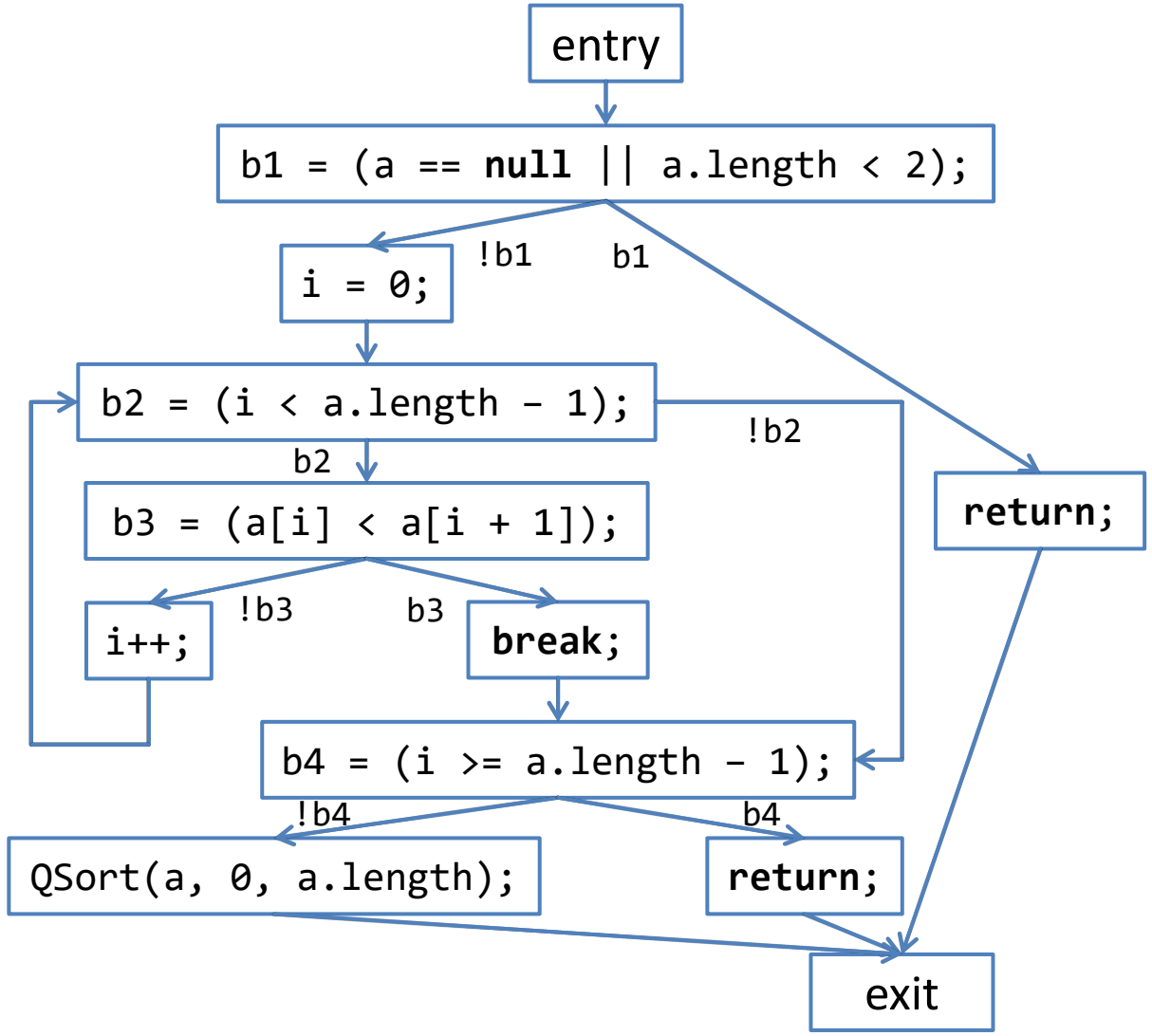
- An intraprocedural control flow graph of a procedure p is a graph (N, E) where:
- N is the set of basic blocks in p plus entry / exit blocks
- E contains
  - An edge from a to b with condition c if the execution of basic block a is succeeded by block b when c holds
  - An edge (entry, a, true) if a is the first basic block of p
  - Edges (b, exit, true) for each basic block b that ends with a possibly implicit return statement

Control flow graph (CFG) – Example



Test coverage

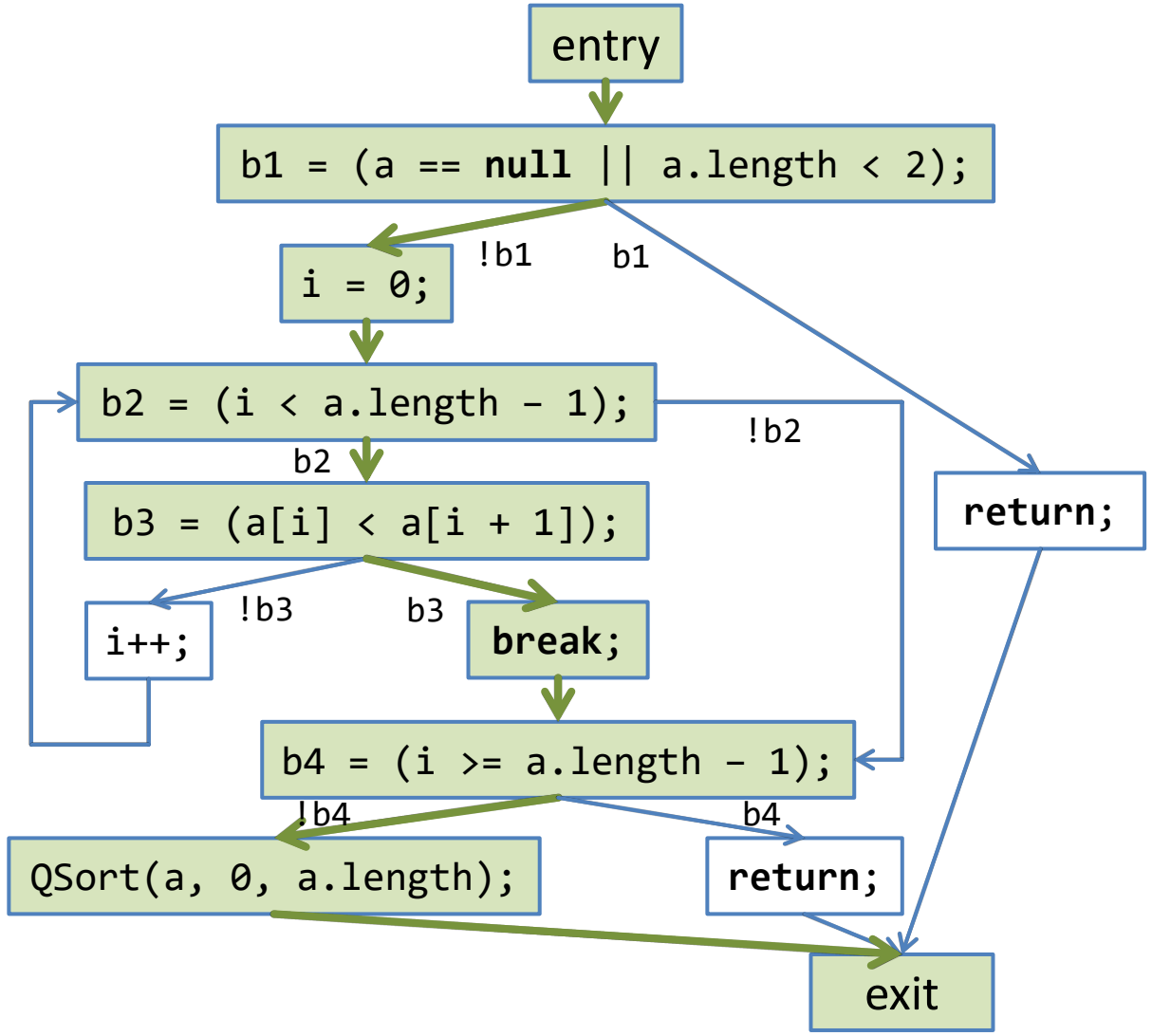
- The CFG can serve as a coverage (or adequacy) criterion for test cases
- The more executed parts, the higher the chance to uncover a bug
- “Parts” can be nodes, edges, paths, etc.



Test coverage – Example

- Consider the input

```
a = {3, 7, 5}
```



Statement coverage

$$\text{Statement coverage} = \frac{\text{Number of executed statements}}{\text{Total number of statements}} \times 100\%$$

- Can also be defined in basic blocks or lines

Basic-block coverage – Example

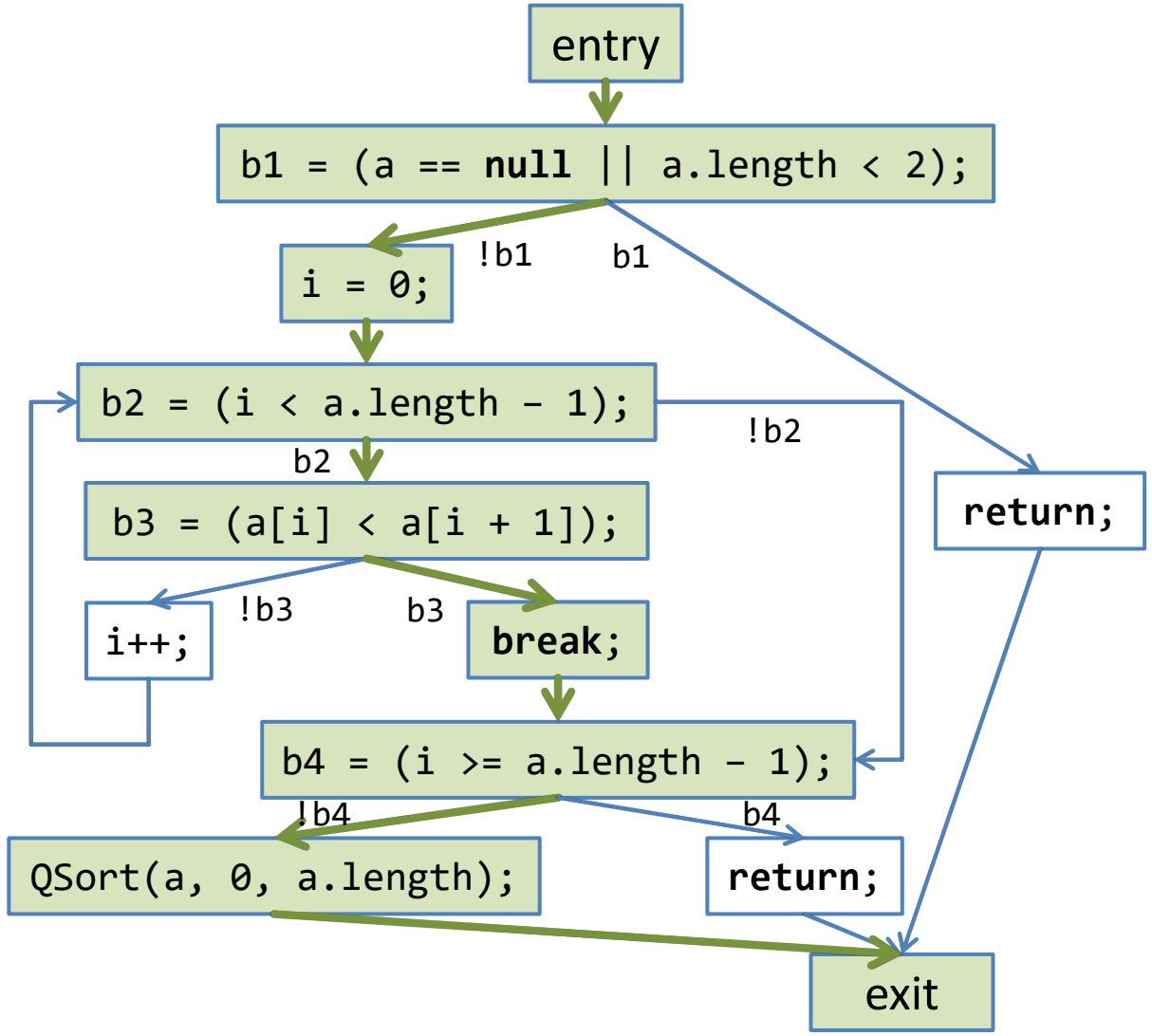
- Consider the input

`a = {3, 7, 5}`

- What is the basic-block coverage?

a) 30% b) 90% c) 70%

- A single test covers 7 / 10 basic blocks



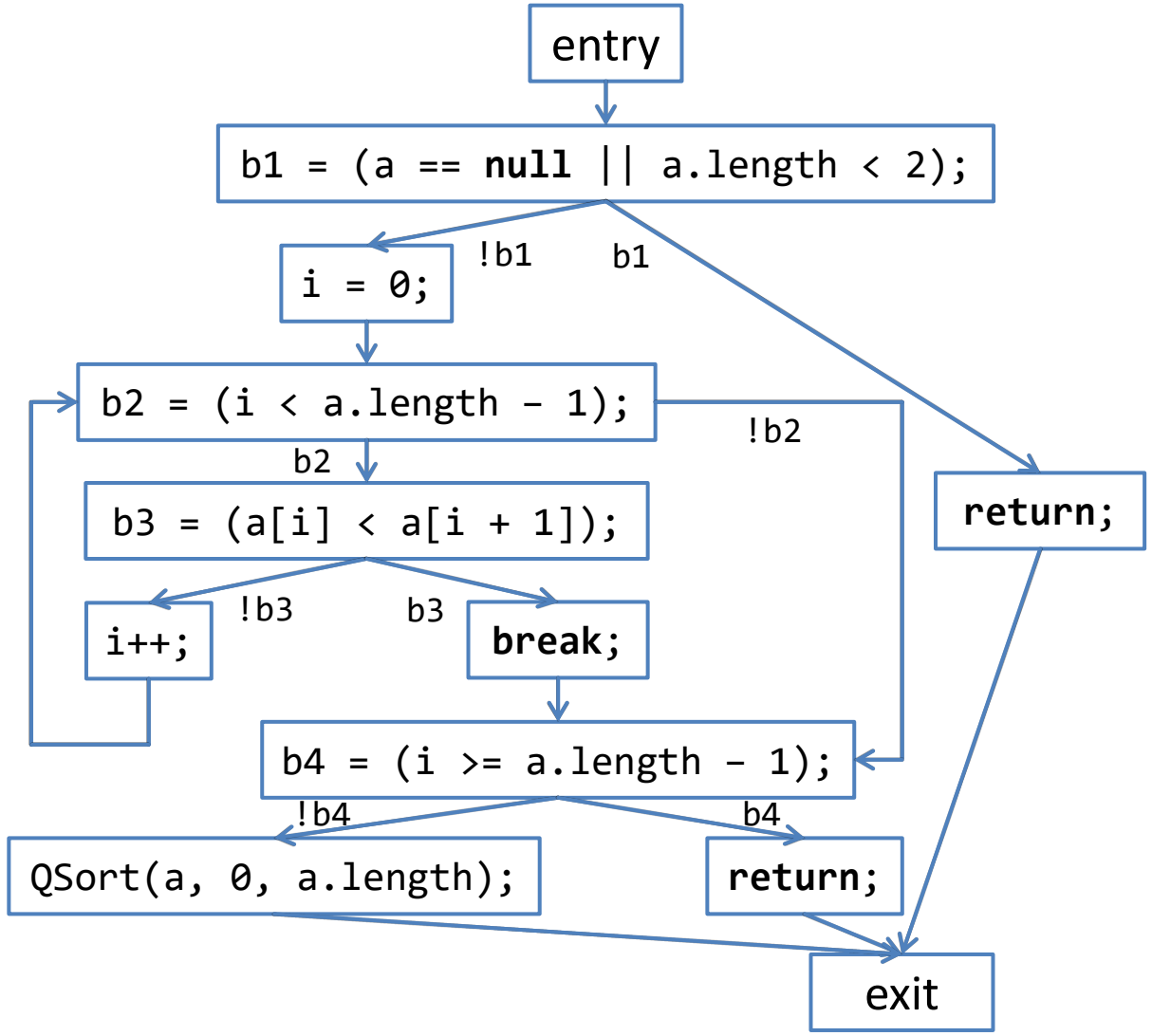
Basic-block coverage – Example

- How many test cases are needed to achieve 100% basic-block coverage?

a) 2 b) 3 c) 4

- Three tests achieve 100% basic-block coverage

```
a = {1}
a = {5, 7}
a = {7, 5}
```



Basic-block coverage – Discussion

```
boolean Contains(int[] a, int x) {
  if (a == null) return false;
  boolean found = false;
  for (int i = 0; i <= a.length; i++) {
    if (a[i] == x) {
      found = true;
      break;
    }
  }
  return found;
}
```

Is this method correct?

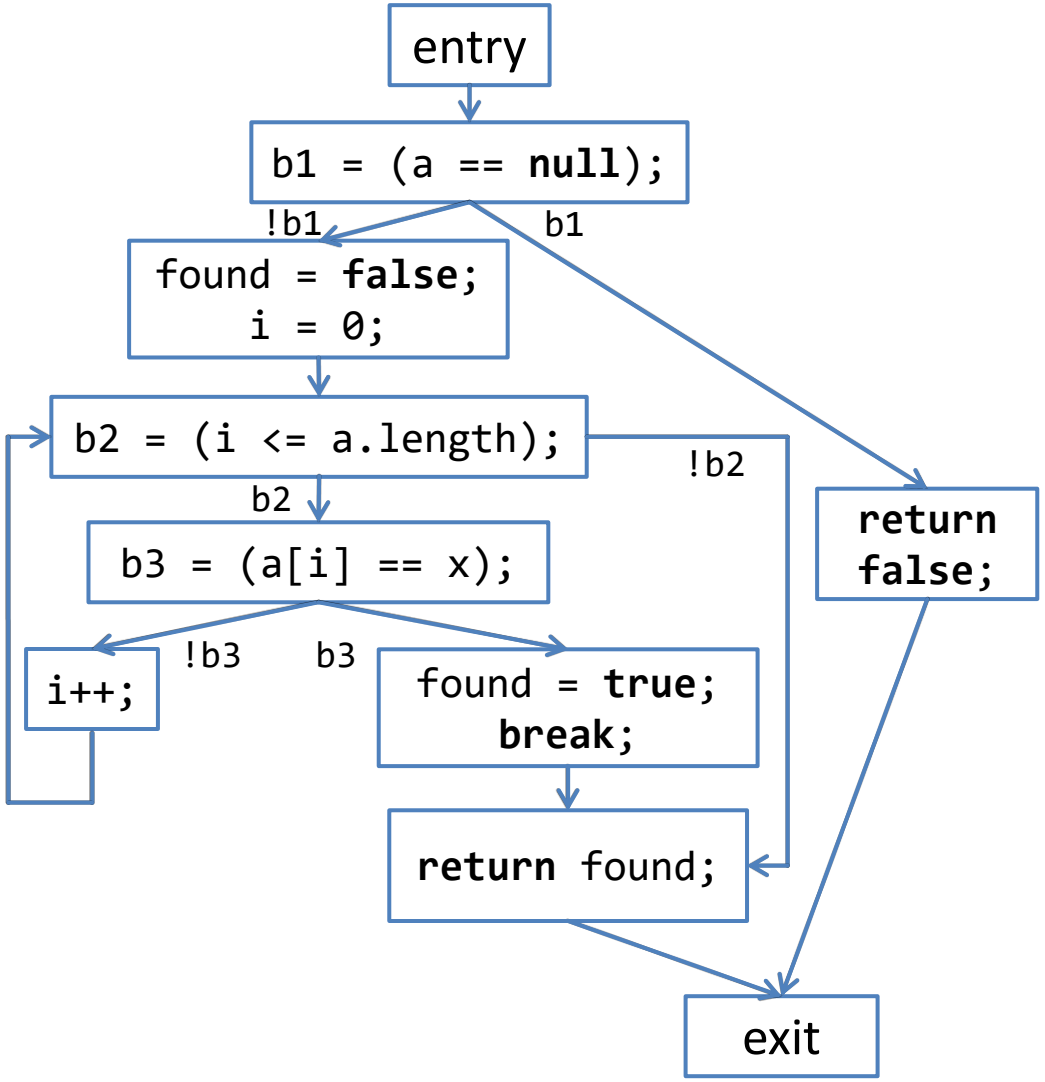
Basic-block coverage – Discussion

- Two tests achieve 100% basic-block coverage

```
a = null
a = {1, 2}, x = 2
```

- The test cases do not detect the bug

- More thorough testing is necessary



Branch coverage

$$\text{Branch coverage} = \frac{\text{Number of executed branches}}{\text{Total number of branches}} \times 100\%$$

- An edge (m, n, c) in a CFG is a branch when there is another edge (m, n', c') in the CFG with n ≠ n'
- Coverage is defined to be 100% if there are no branches

Branch coverage – Example

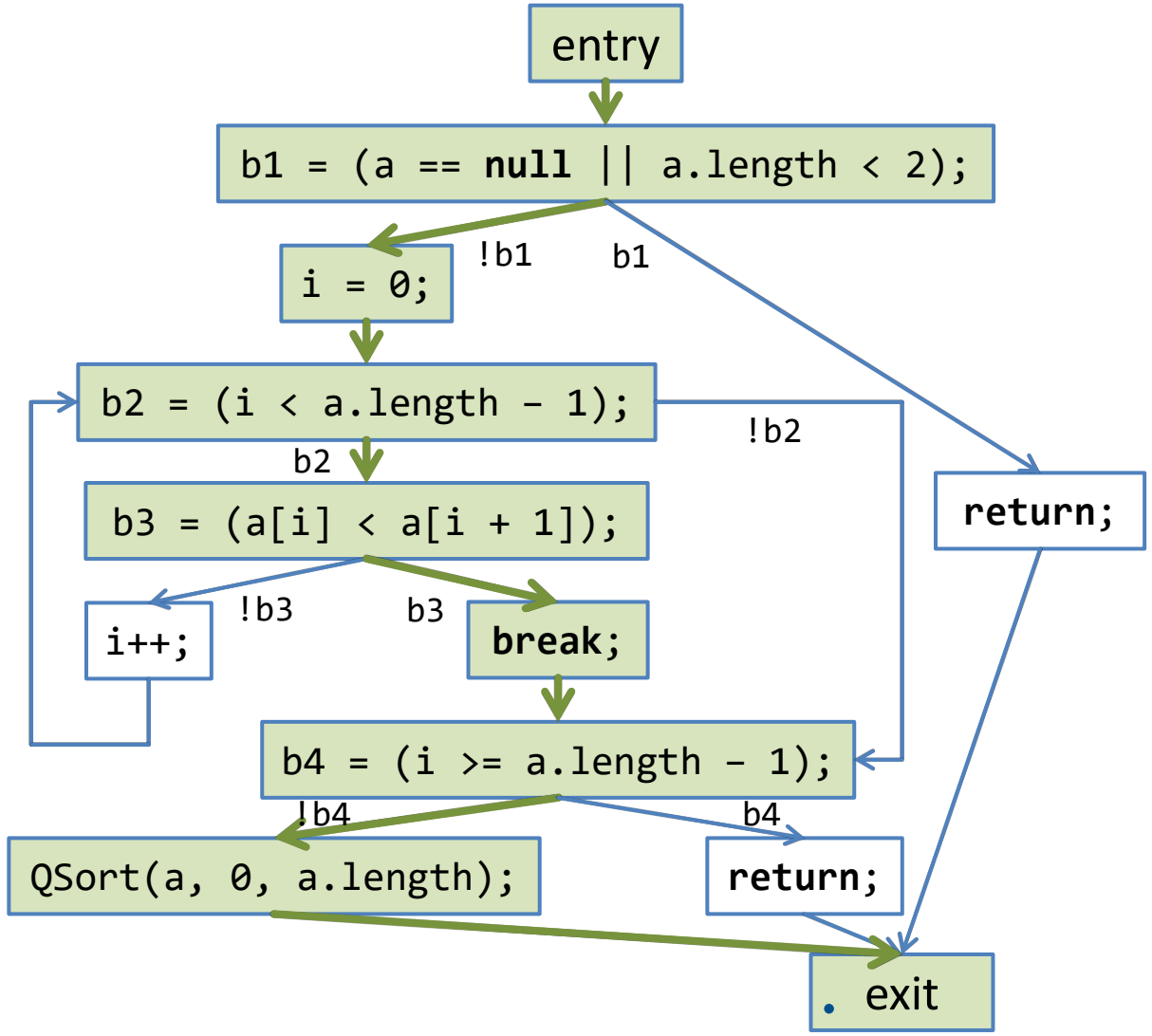
- Consider the input

```
a = {3, 7, 5}
```

- What is the branch coverage?

a) 30% b) 50% c) 70%

- The test covers 4 / 8 branches



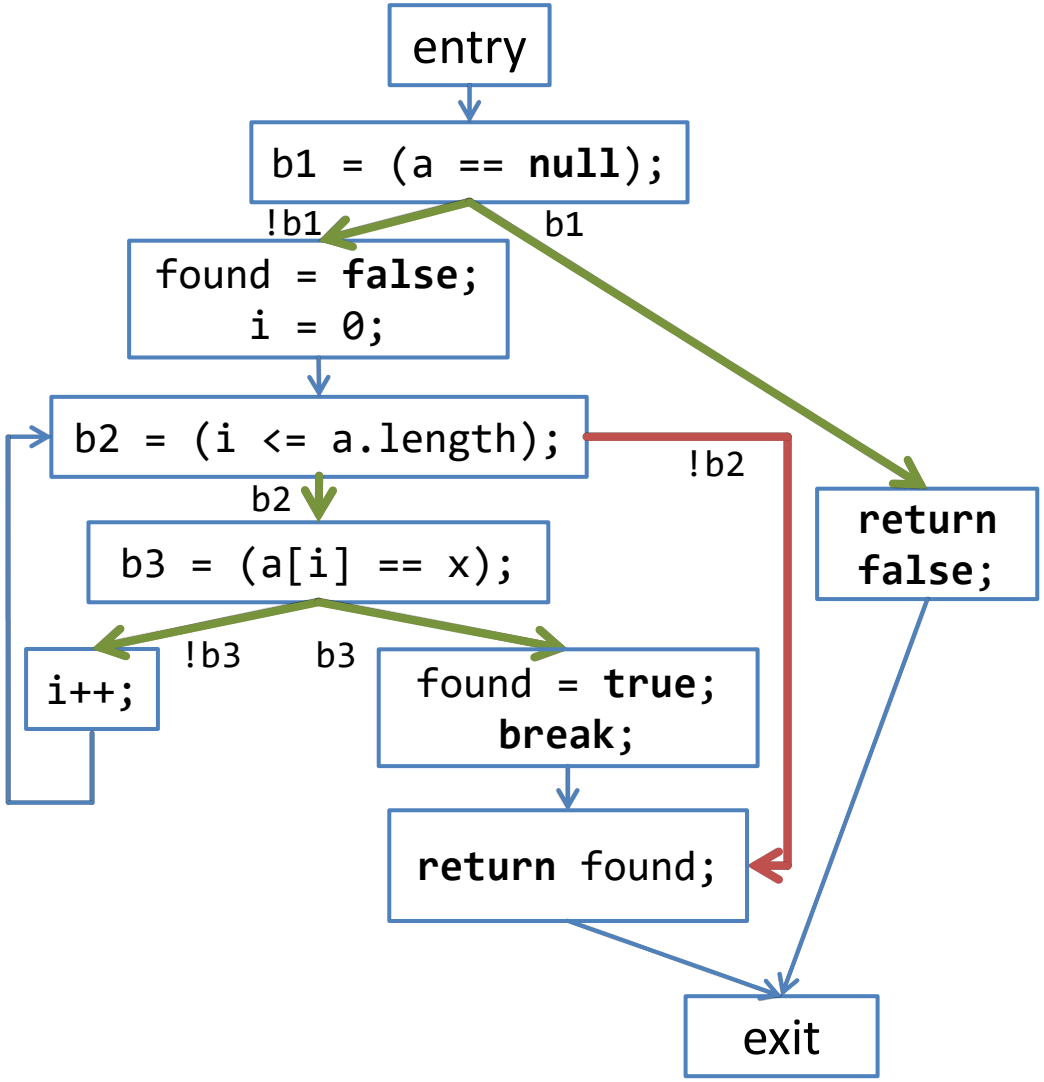
Branch coverage – Example

- Two tests achieve 83% branch coverage

```
a = null
a = {1, 2}, x = 2
```

- The test cases execute 5 / 6 branches

- The bug is not detected

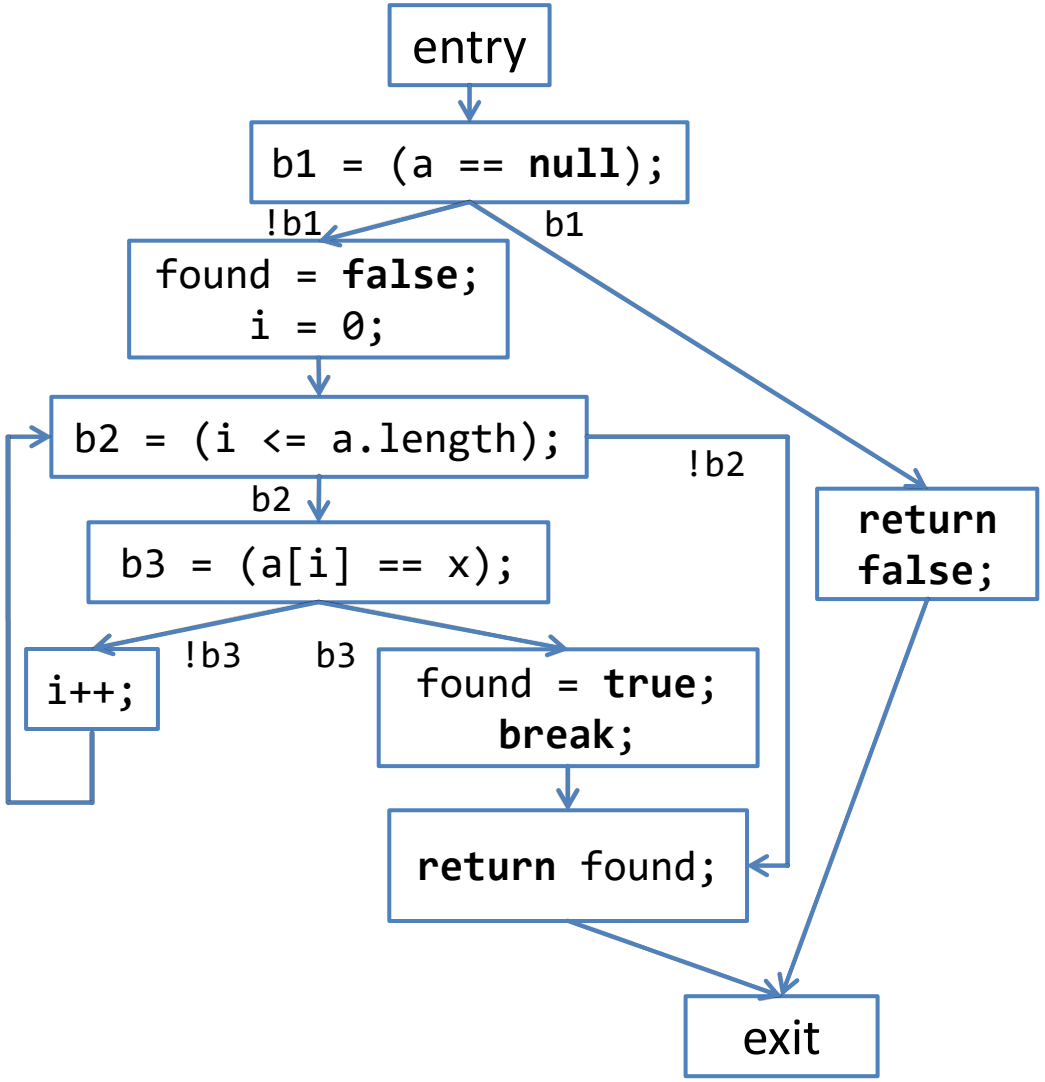


Branch coverage – Example

- Achieving 100% coverage requires a test that runs the loop to the end

```
a = null
a = {1}, x = 1
a = {1}, x = 3
```

- The last test case detects the bug



Branch coverage – Discussion

- Branch coverage leads to more thorough testing than statement coverage
  - Complete branch coverage implies complete statement coverage
  - But “at least n% branch coverage” does not generally imply “at least n% statement coverage”
- Most widely used coverage criterion in industry

Branch coverage – Discussion

```
int[] Reverse(int[] a) {
  int j = a.length – 1;
  int[] res = new int[a.length];
  for (int i = 0; i < a.length; i++) {
    res[j] = a[i];
  }
  return res;
}
```

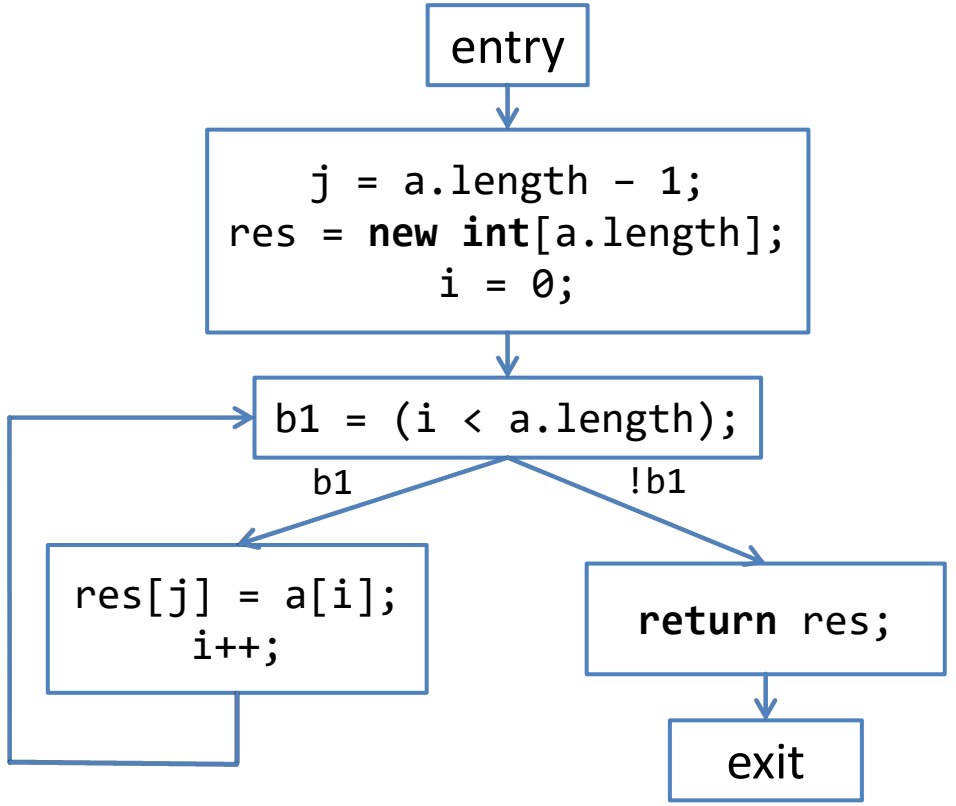
Is this method correct?

Branch coverage – Discussion

- One test achieves 100% branch coverage

```
a = {1}
```

- The test case does not detect the bug
- More thorough testing is necessary



Branch coverage – Discussion

```
int Foo(boolean a, boolean b) {
  int x = 1;
  int y = 1;
  if (a)
    x = 0;
  else
    y = 0;
  if (b)
    return 5 / x;
  else
    return 5 / y;
}
```

Is this method correct?

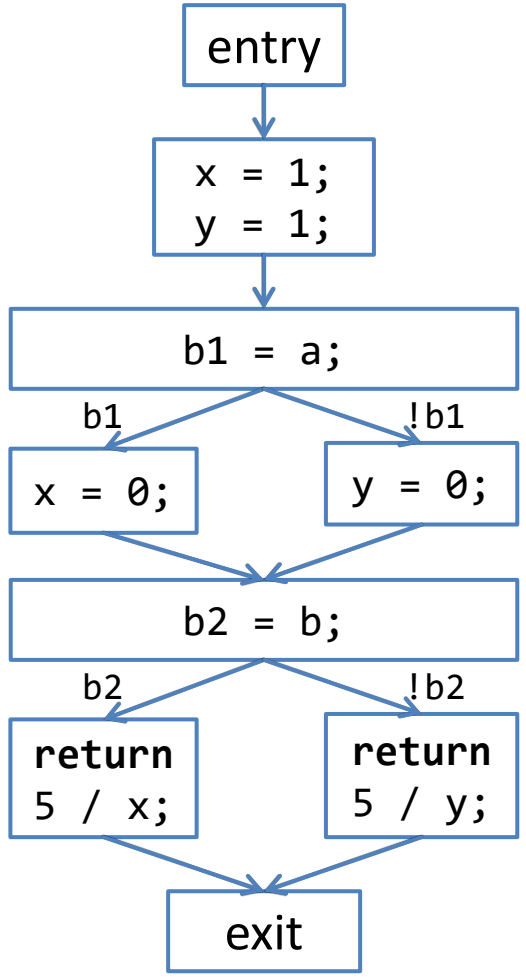
Branch coverage – Discussion

- Two tests achieve 100% branch coverage

```
a = true, b = false
a = false, b = true
```

- The test cases do not detect the bug

- More thorough testing is necessary



Path coverage

$$\text{Path coverage} = \frac{\text{Number of executed paths}}{\text{Total number of paths}} \times 100\%$$

- A path is a sequence of nodes n_1, …, n_k such that
  - n_1 = entry
  - n_k = exit
  - There is an edge (n_i, n_{i+1}, c) in the CFG

Path coverage – Example

- Consider the inputs

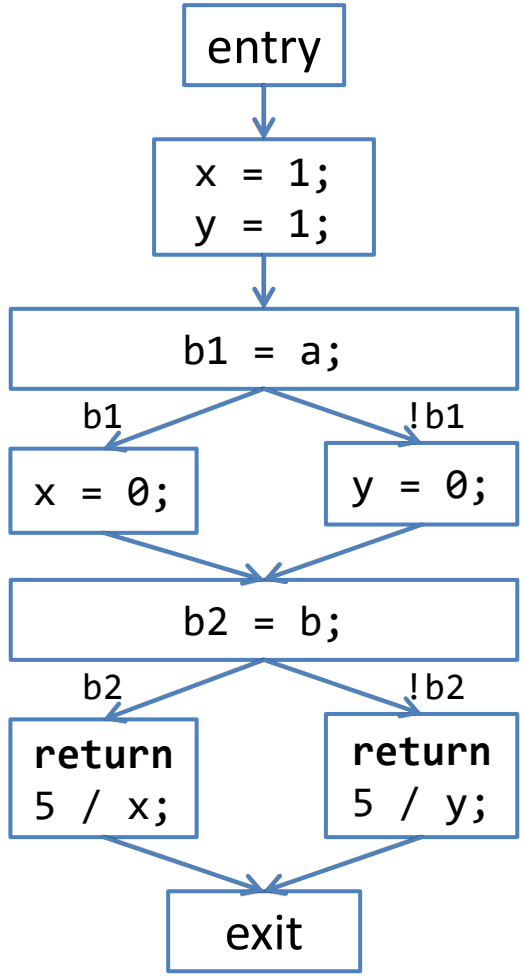
`a = true, b = false`

`a = false, b = true`

- What is the path coverage?

a) 50% b) 70% c) 100%

- The tests cover 2 / 4 paths

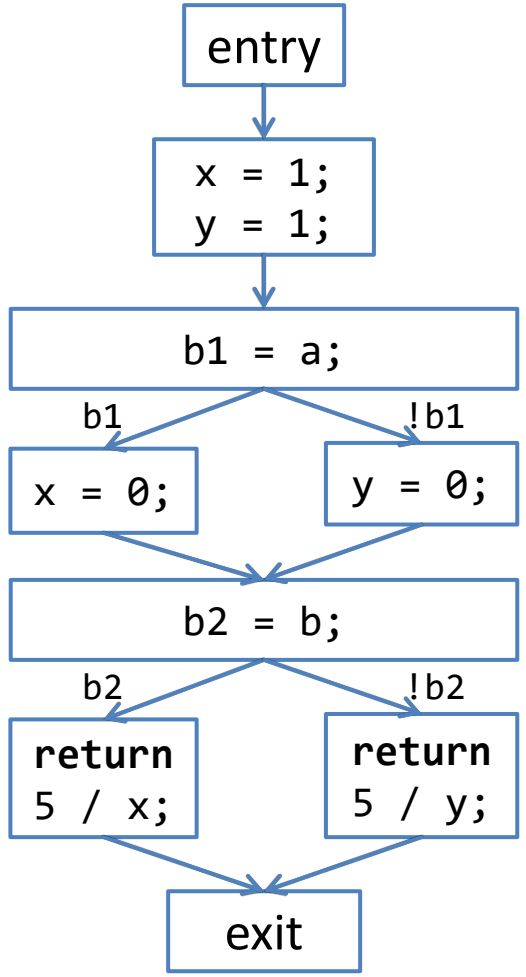


Path coverage – Example

- Achieving 100% coverage requires four test cases

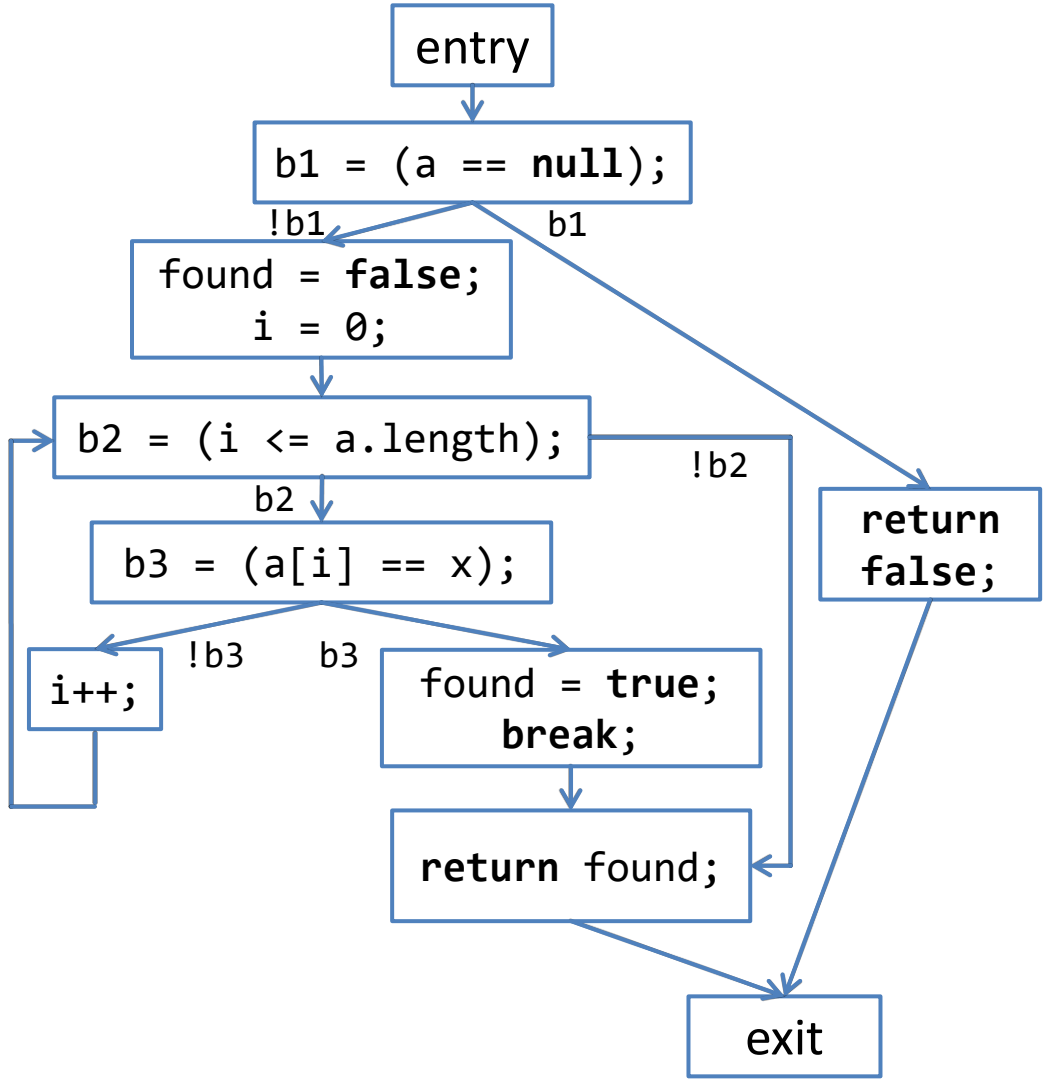
```
a = true, b = false
a = false, b = true
a = true, b = true
a = false, b = false
```

- The last two tests detect the bugs



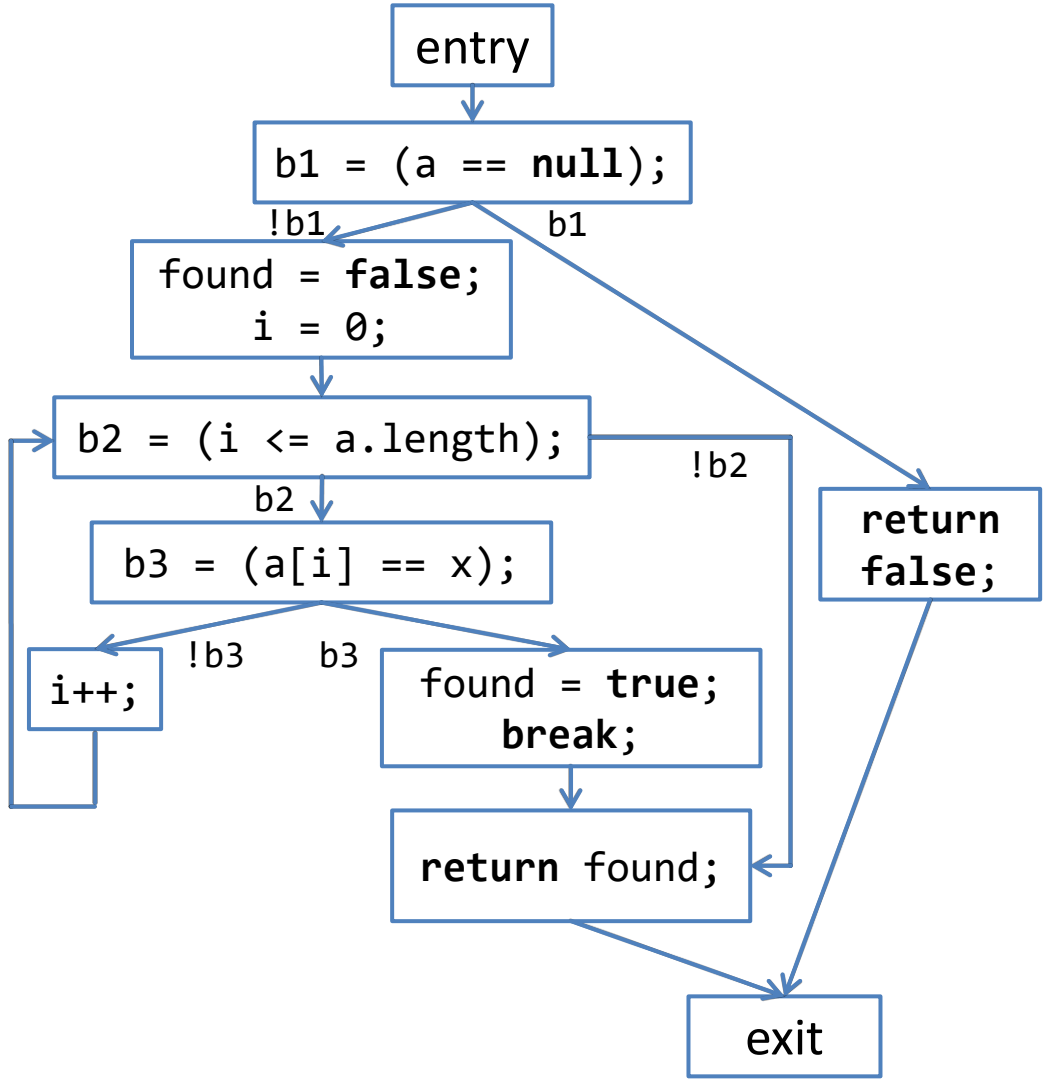
Path coverage – Example

```
boolean Contains(int[] a, int x) {
  if (a == null) return false;
  boolean found = false;
  for (int i = 0; i <= a.length; i++) {
    if (a[i] == x) {
      found = true;
      break;
    }
  }
  return found;
}
```



Path coverage – Example

- The number of loop iterations is not known statically (depends on input)
- An arbitrarily large number of test cases is needed for complete path coverage



Path coverage – Discussion

- Path coverage leads to more thorough testing than both statement and branch coverage
  - Complete path coverage implies complete statement and branch coverage
  - But “at least n% path coverage” does not typically imply “at least n% statement coverage” or “at least n% branch coverage”
- Complete path coverage is not feasible for input-dependent loops (unbounded number of paths)

Branch coverage – Discussion

```
int[] Reverse(int[] a) {
  int j = a.length – 1;
  int[] res = new int[a.length];
  for (int i = 0; i < a.length; i++) {
    res[j] = a[i];
  }
  return res;
}
```

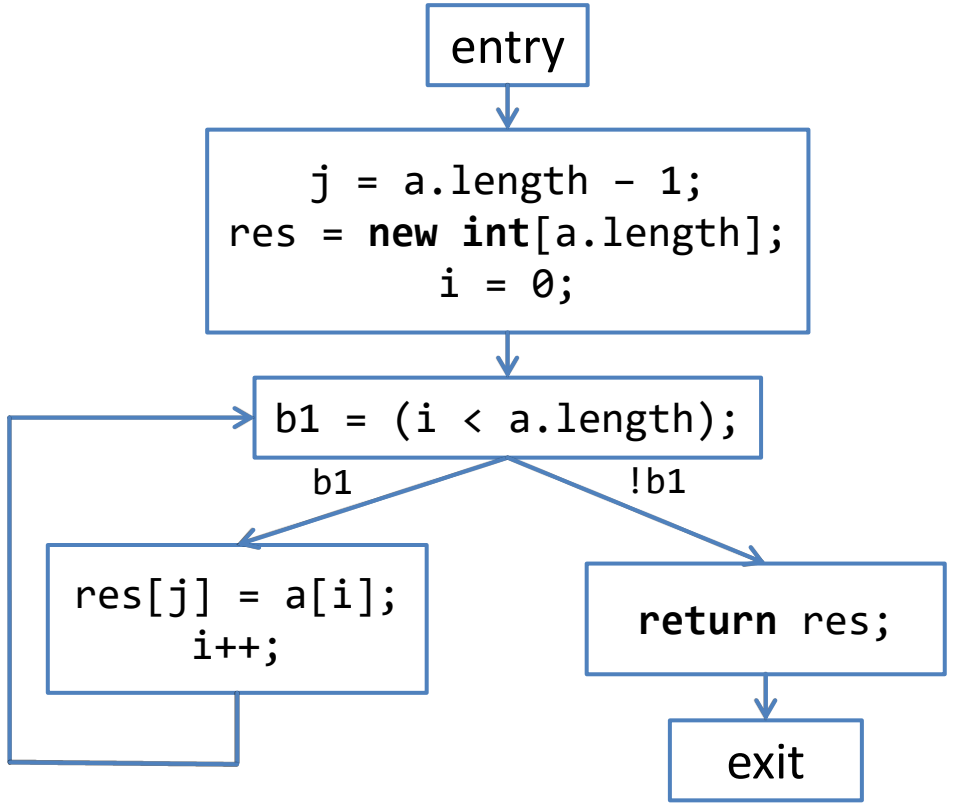
Is this method correct?

Branch coverage – Discussion

- One test achieves 100% branch coverage

```
a = {1}
```

- The test case does not detect the bug
- More thorough testing is necessary



Loop coverage

$$\text{Loop coverage} = \frac{\text{Number of executed loops with } 0, 1, \text{and more than } 1 \text{ iterations}}{\text{Total number of loops} \times 3} \times 100\%$$

- Loop coverage is typically combined with other coverage criteria such as statement or branch coverage

Loop coverage – Example

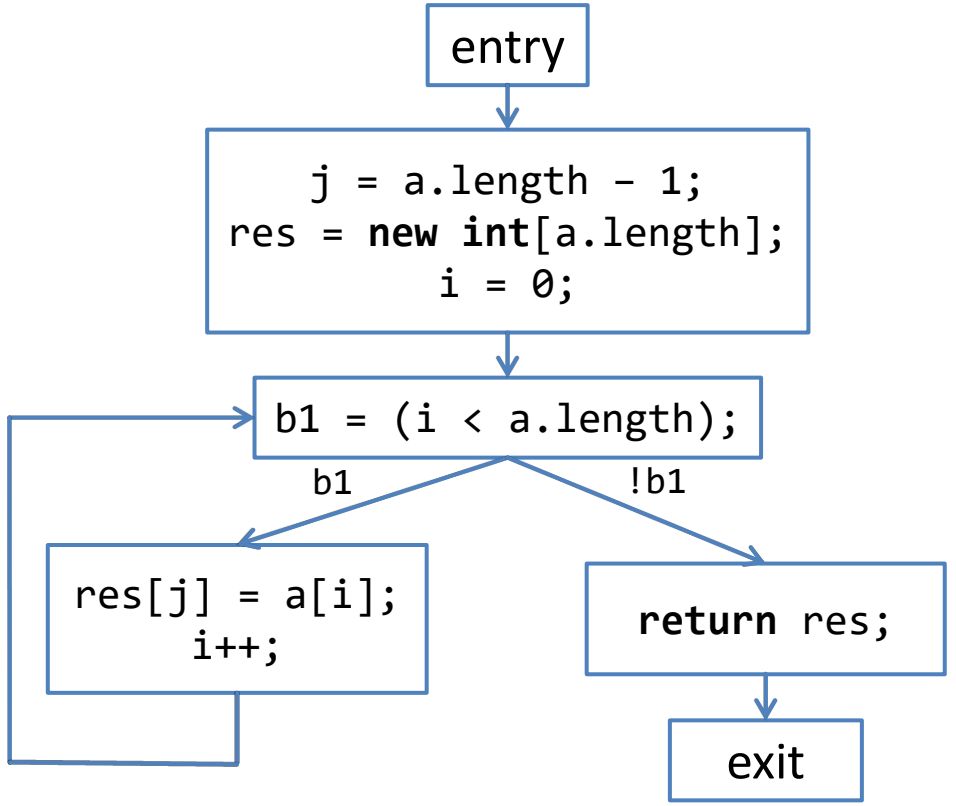
- Consider the input

`a = {1}`

- What is the loop coverage?

a) 33% b) 66% c) 99%

- The test executes 1 / 3 possible cases for the loop

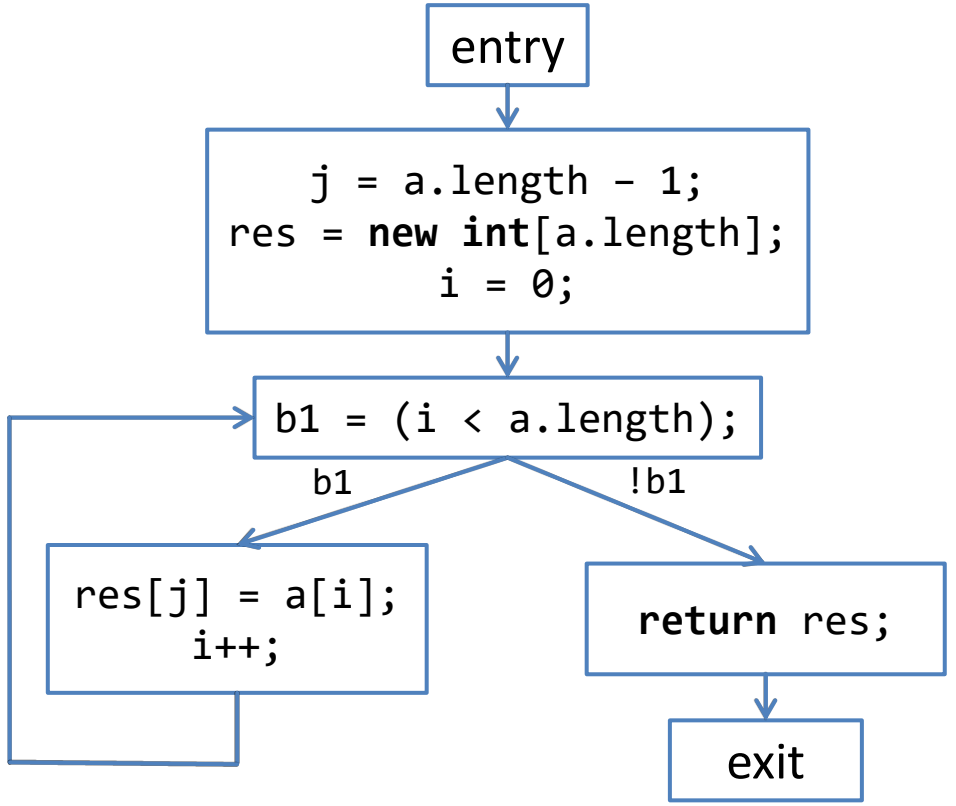


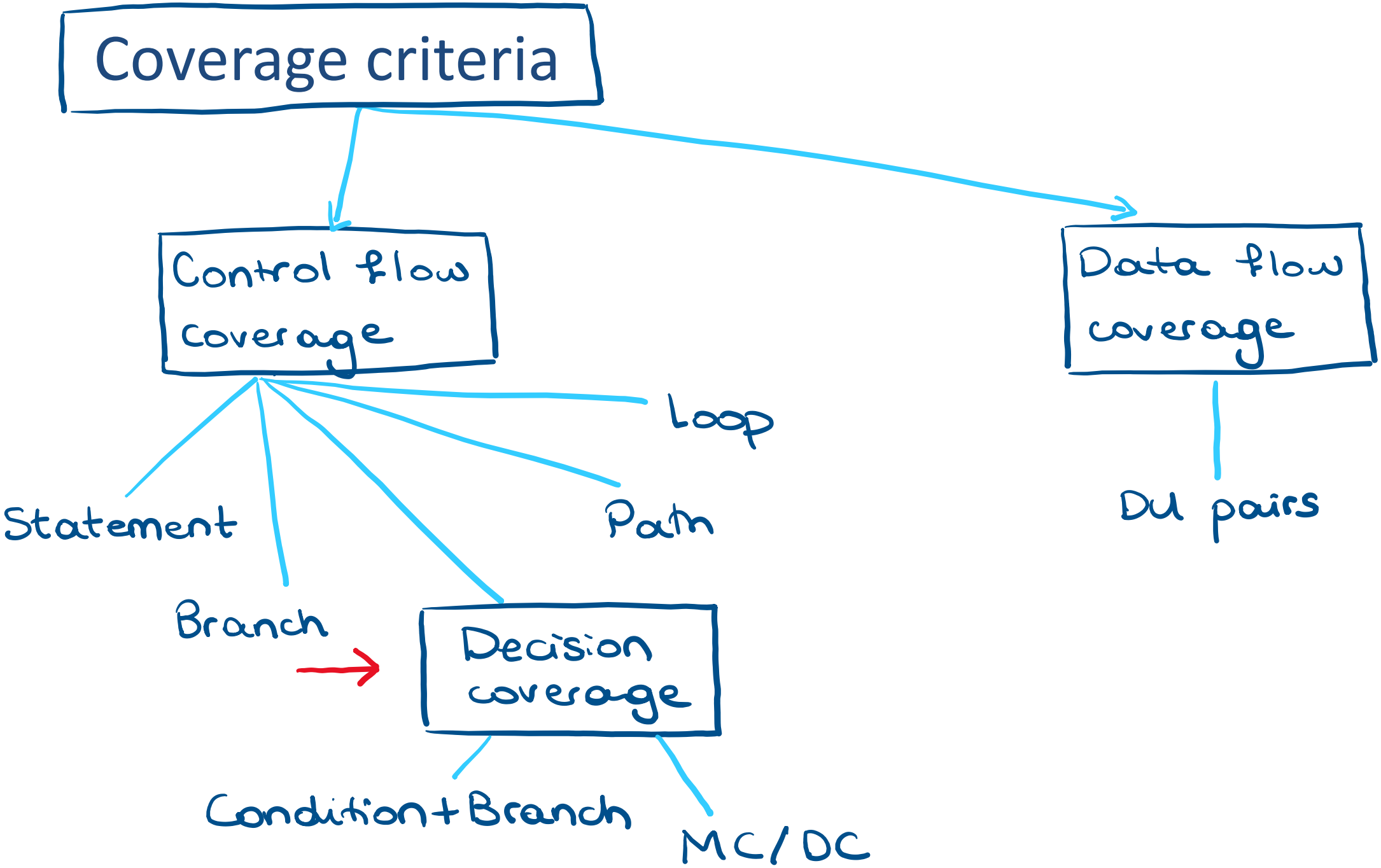
Loop coverage – Example

- Three tests achieve 100% loop coverage

```
a = {}
a = {1}
a = {1, 2}
```

- The last test case detects the bug




Coverage criteria
Control flow coverage
Data flow coverage
Loop
Statement
Path
DU pairs
Branch
Decision coverage
Condition+Branch
MC/DC

Condition + Branch coverage

$$\text{C + B coverage} = \frac{\text{Number of executed branches + condition values}}{\text{Total number of branches + condition values}} \times 100\%$$

- Coverage is 100% when each individual condition evaluates to true and false at least once and each corresponding branch statement also evaluates to true and false at least one

Condition + Branch coverage – Example

```
if (x || y) {
    do A
} else {
    do B
}
```

there are 2 condition values for x and 2 for y

Imagine a test where x = true

- Branch coverage:

$$\frac{1}{2} \times 100\% = 50\%$$

- Condition + Branch coverage:

$$\frac{1+2}{2+4} \times 100\% = 50\%$$

MC/DC

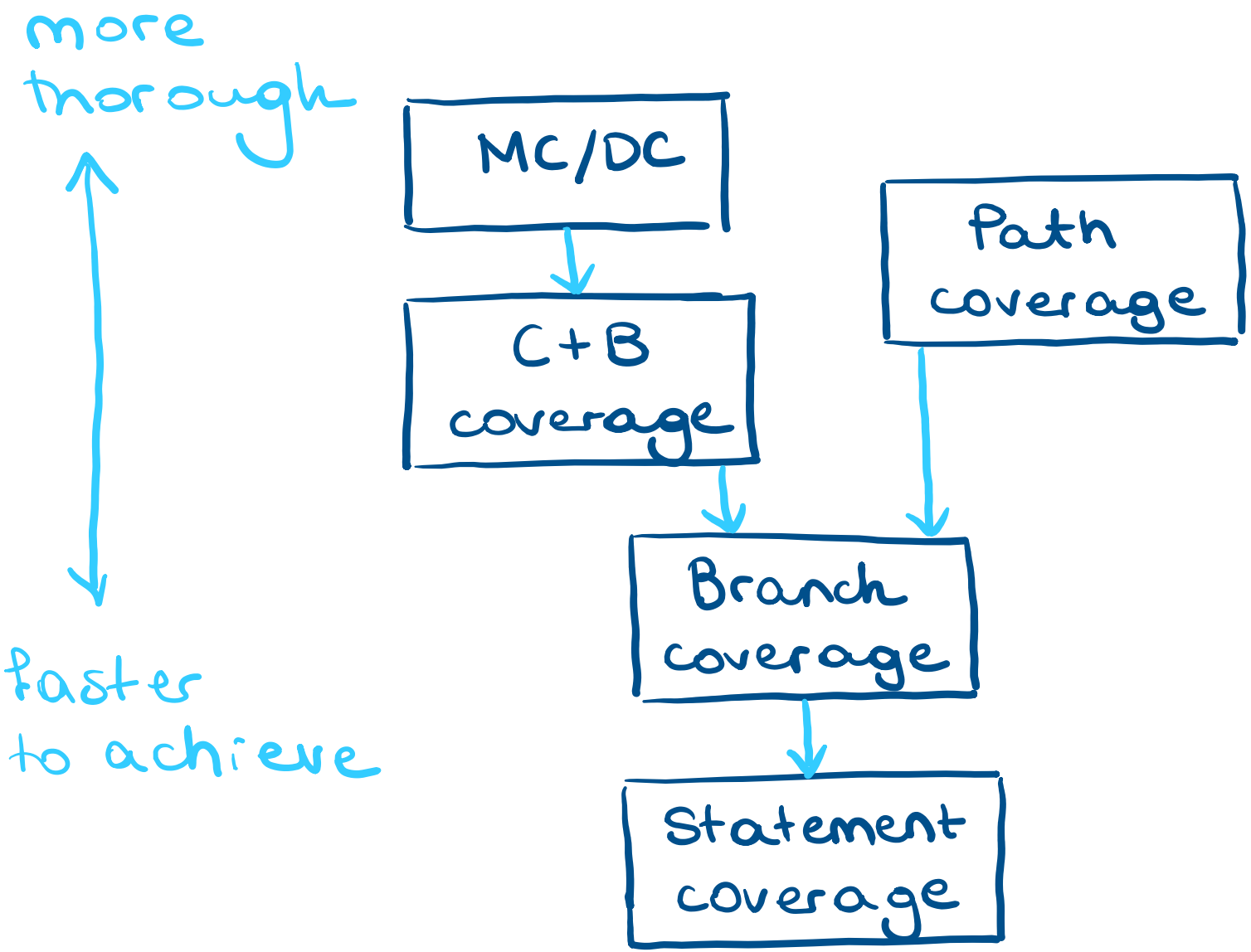
- MC/DC stands for modified condition / decision coverage
- It is used to test complex conditions more efficiently than testing all possible condition combinations
  - It exercises each condition so that it can, independently of the other conditions, affect the outcome of the entire decision

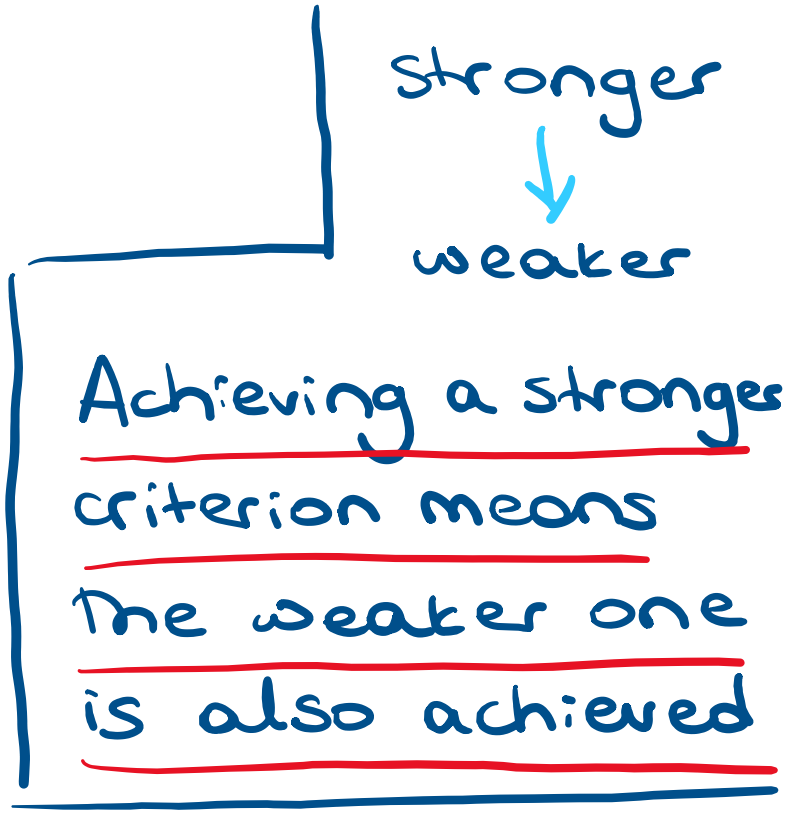
MC/DC – Example

Assume `if (A && (B || C))`, where `A`, `B`, and `C` evaluate to Booleans

- For condition `A`
  - There must be one test case where `A = true` (say, T1)
  - There must be one test case where `A = false` (say, T2)
  - T1 and T2 (which we call independence pairs) must have different outcomes (e.g., T1 makes the entire decision true and T2 makes it false)
  - `B` and `C` in T1 must have the same truth values in T1 and T2
- Entire example

Criteria subsumption





Measuring control flow coverage

- Coverage information is collected while the tests execute
  - Typically using code instrumentation to count executed basic blocks, branches, etc.

```
int Foo(boolean a, boolean b) {
  int x = 1;
  int y = 1;
  if (a)
    branchCovered[0] = true;
    x = 0;
  else
    branchCovered[1] = true;
    y = 0;
  if (b)
    branchCovered[2] = true;
    return 5 / x;
  else
    branchCovered[3] = true;
    return 5 / y;
}
```

Suggested reading

From Book: "Effective Software Testing A Developer's Guide" by M. Aniche

- Section 2.1
- Section 2.2

Reading about structural coverage from the book might be confusing