# Testing, Debugging, and Verification

## Testing, Part III

Srinivas Pinisetty[1]

12 November 2018

[1]Slides based on material from Wolfgang Aherndt,..

# Overview of this Lecture

This lecture is all about unit testing

Specific topics:

- Recap: JUnit- a framework for rapid unit testing
- Recap: Integrating test units
- Principles of test set construction
- Quality criteria for test sets

# JUnit (Recap.)

- Java testing framework to write and run automated tests
- JUnit features include:
  - Assertions for testing expected results
  - Annotations to designate test cases
  - Sharing of common test data
  - Graphical and textual test runners
- JUnit is widely used in industry
- JUnit used from command line or within an IDE (e.g., Eclipse)

# Recap: JUnit and Extreme Testing

- Extreme testing
  - Test-cases first: Clear idea of what program should do before coding.
  - Understanding of specification and requirements.
  - Write test-cases first then implementation.

# Recap: JUnit and Extreme Testing

- Extreme testing
  - Test-cases first: Clear idea of what program should do before coding.
  - Understanding of specification and requirements.
  - Write test-cases first then implementation.
- Regression testing
  - re-run after changes to code.

# Recap: Integrating Test Units

Testing a unit may require:

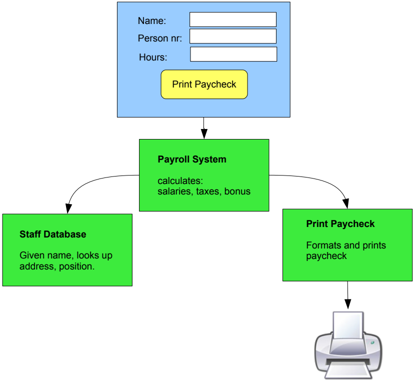
**Stubs** (top-down) to replace called procedures

- Simulate behaviour of component not yet developed.
- E.g. test code that calls a method not yet implemented.

**Drivers** (bottom-up) to replace calling procedures

- Simulate environment from where procedure is called.
- E.g. test harness.

# Recap: Top-down vs Bottom-up Testing




- Bottom-up?
  start from e.g. Print Paycheck, Staff Database
  - Driver replacing the caller, here Payroll System, Interface.
- Top-down?
  start from e.g. Interface, then Payroll System
  - Stubs replacing called procedures, i.e. Payroll System, Staff Database, Print Paycheck.

# Today

**Terminology recap**

- Specification
- Test case
- Test suite

**Today: Principles of test suite construction**

How do we pick test cases? When do we know we have enough test cases?

# Test Suites

**Test Suite**

A test suite is a set of test cases.

Most central activity of testing is the creation of test suites

How do we know if we have

- Enough test cases?
- The right test cases?
- Quality of test suites defines quality of overall testing effort

# When do we have enough tests

### Example

```
public static boolean and(boolean a, boolean b)
```

The output is true if and only if both the inputs are true and false otherwise.

### Tests

- and(false,false) == false
- and(false,true) == false
- and(true,false) == false
- and(true,true) == true

# When do we have enough tests

Example

```
public static Integer[] sort(Integer[] a) {
        ...
        }
```

- Complete/exhaustive testing is impossible in general.
- When do we have enough tests? (an answer: coverage criteria)
- Coverage criteria: How much of the code is covered by the set of tests?
- Different ways to define covered.

# Coverage Criteria

Most metrics used as quality criteria for test suites describe the degree of some kind of coverage.

These metrics are called coverage criteria.

Crucial for testing safety critical software.
Requirements of testing to certain coverage criteria.

# Categories of Coverage Criteria

Following the categorisation of [AmmannOffutt] (simplified), we group coverage criteria as follows:

**Coverage Criteria Grouping**

- Control flow graph coverage
- Logic coverage
- Input space partitioning

# Control Flow Graph

### Control Flow Graph

Represent implementation under test as graph:

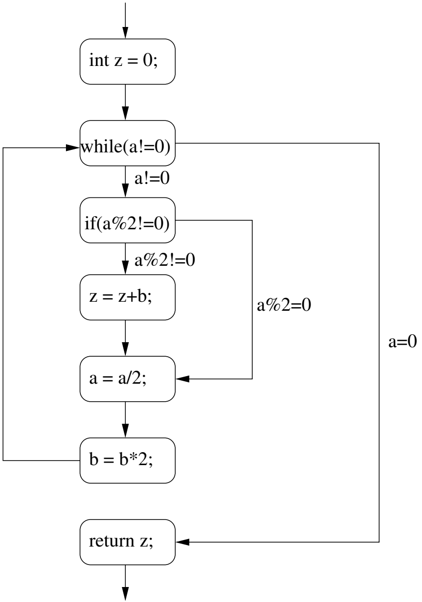
- Every statement represented by a node
- Edges describe control flow between statements
- Edges can be constrained by conditions

# Example

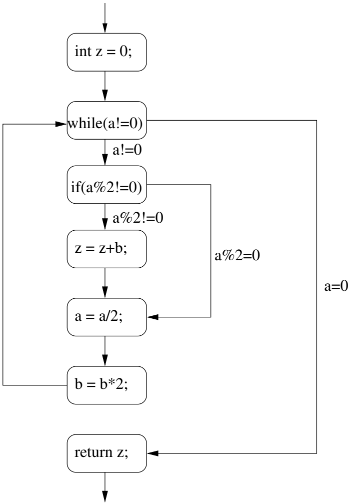
```
int russianMultiplication(int a, int b){
    int z = 0;
    while(a != 0){
        if(a%2 != 0){
            z = z+b;
        }
        a = a/2;
        b = b*2;
    }
    return z;
}
```

[example and graph by Christian Engel]

# Control Flow Graph of `russianMultiplication()`



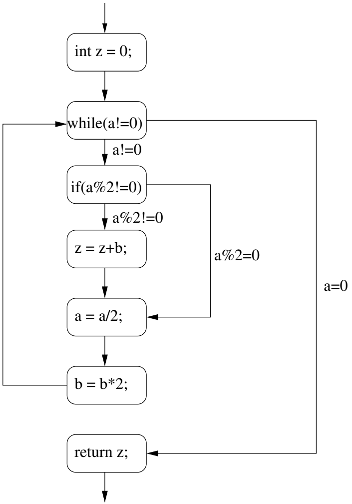
# Control Flow Graph Notions




## Execution Path

A path through a control flow graph, that starts at the entry point and is either infinite or ends at one of the exit points.

# Statement Coverage



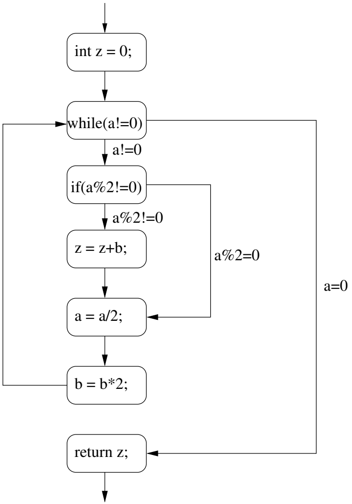

**Statement Coverage (SC)**

SC is satisfied by a test suite *TS*, iff for every node *n* in the control flow graph there is at least one test in *TS* causing an execution path via *n*.

For `russianMultiplication()`:

- $TS = \{(\mathtt{a} = 1, \mathtt{b} = 0)\}$ satisfies statement coverage

# Branch Coverage




**Branch Coverage (BC)**

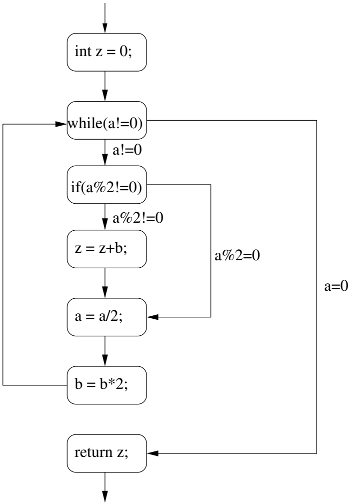
BC is satisfied by a test suite *TS*, iff for every edge *e* in the control flow graph there is at least one test in *TS* causing an execution path via *e*.

BC subsumes SC.

For `russianMultiplication()`:

- $TS = \{(\mathtt{a} = 2, \mathtt{b} = 0)\}$ satisfies branch coverage

# Path Coverage



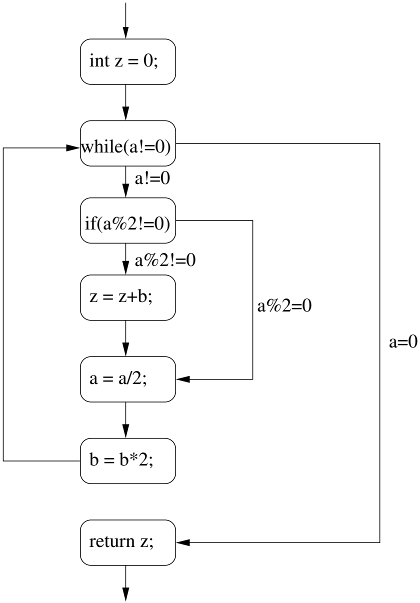

## Path Coverage (PC)

PC is satisfied by a test suite *TS*, iff for every execution path *ep* of the control flow graph there is at least one test in *TS* causing *ep*.

PC subsumes BC.

- PC cannot be achieved in practice
- For `russianMultiplication()`, number of execution paths is $>> 2^{31}$

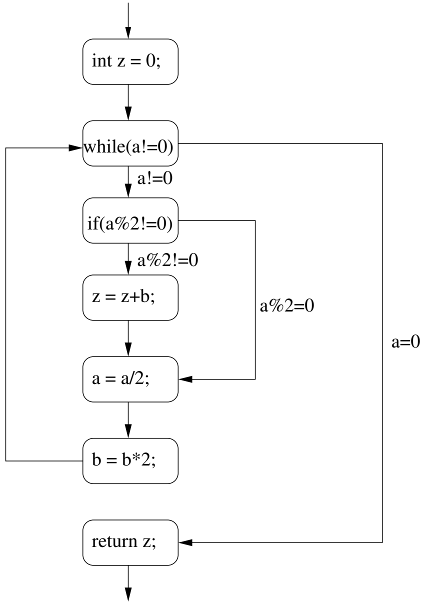
# Mini Quiz: Graph Coverage




Does the following test cases satisfy Statement Coverage, Branch Coverage and/or Path Coverage?

- `[a=3, b=3]`
- `[a=0, b=2]`
- `[a=4, b=1]`

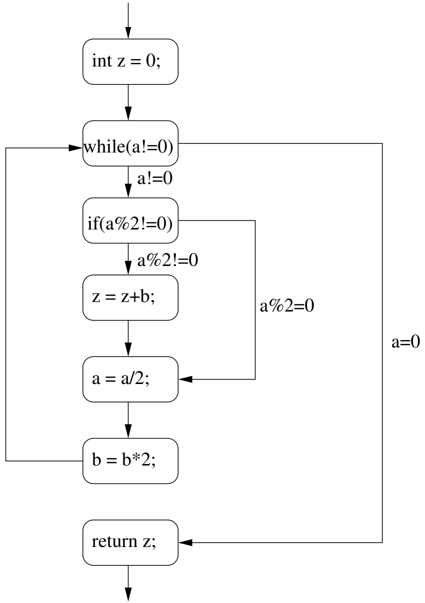
# Mini Quiz: Graph Coverage




Does the following test cases satisfy Statement Coverage, Branch Coverage and/or Path Coverage?

- `[a=3, b=3]` SC
- `[a=0, b=2]`
- `[a=4, b=1]`

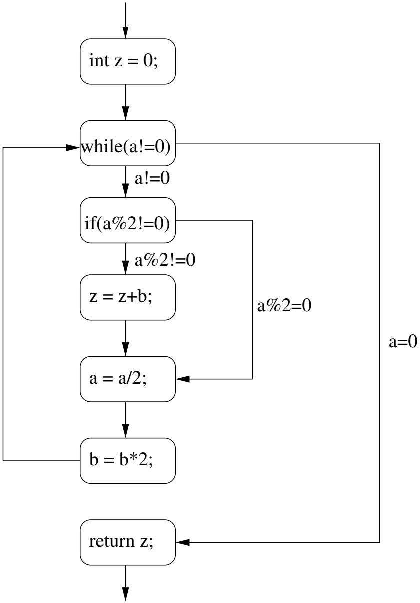
# Mini Quiz: Graph Coverage




Does the following test cases satisfy Statement Coverage, Branch Coverage and/or Path Coverage?

- `[a=3, b=3]` SC
- `[a=0, b=2]` neither
- `[a=4, b=1]`

# Mini Quiz: Graph Coverage




Does the following test cases satisfy Statement Coverage, Branch Coverage and/or Path Coverage?

- [a=3, b=3] SC
- [a=0, b=2] neither
- [a=4, b=1] SC and BC

# Logic Coverage

Logical (`boolean`) expressions can come from many sources:

1. Decisions in source code (e.g., `if`, `while`)
2. Decisions in software models (FSMs and statecharts)
3. Case distinctions in requirements

We focus on 1.

# Decision Coverage

Let the decisions of a program $p$, $D(p)$, be the set of all boolean expressions which $p$ branches on.

# Decision Coverage

Let the decisions of a program $p$, $D(p)$, be the set of all boolean expressions which $p$ branches on.

## Decision Coverage (DC)

For a given decision $d$, DC is satisfied by a test suite $TS$ if it

- Contains at least two tests
- one where $d$ evaluates to *false*
- one where $d$ evaluates to *true*

For a given program $p$, DC is satisfied by $TS$ if it satisfies DC for all decisions $d \in D(p)$.

# Decision Coverage

**Example**

For decision $((a < b) \mid\mid D)$ && $(m \geq n * o))$,
DC is satisfied for instance if $TS$ triggers executions with:

# Decision Coverage

**Example**

For decision $((a < b) \mid\mid D)\ \&\&\ (m \geq n * o))$,
DC is satisfied for instance if *TS* triggers executions with:

$a = 5, b = 10, D = true, m = 1, n = 1, o = 1$
and
$a = 10, b = 5, D = false, m = 1, n = 1, o = 1$

# Decision Coverage

### Example

For decision $((a < b) \,||\, D) \,\&\&\, (m \geq n * o))$,
DC is satisfied for instance if $TS$ triggers executions with:

$a = 5, b = 10, D = true, m = 1, n = 1, o = 1$
and
$a = 10, b = 5, D = false, m = 1, n = 1, o = 1$

### Inner Value Problem

- the above values are not test case inputs, but values at the time of executing the decision
- finding corresponding input values

# Condition Coverage

Let the conditions of a program $p$, $C(p)$, be the set of all boolean sub-expressions $c$ of decisions in $D(p)$, such that $c$ does not contain other boolean sub-expressions.

Given the decision $((a < b) \mid\mid D)$ && $(m \geq n * o))$,
the conditions are: $(a < b)$, $D$, and $(m \geq n * o)$.

# Condition Coverage

Let the conditions of a program $p$, $C(p)$, be the set of all boolean sub-expressions $c$ of decisions in $D(p)$, such that $c$ does not contain other boolean sub-expressions.

Given the decision $((a < b) \mid\mid D)$ && $(m \geq n * o))$,
the conditions are: $(a < b)$, $D$, and $(m \geq n * o)$.

## Condition Coverage (CC)

For a given condition $c$, CC is satisfied by a test suite $TS$ if it

- contains at least two tests
- one where $c$ evaluates to *false*
- one where $c$ evaluates to *true*

# Condition Coverage

Let the conditions of a program $p$, $C(p)$, be the set of all boolean sub-expressions $c$ of decisions in $D(p)$, such that $c$ does not contain other boolean sub-expressions.

Given the decision $((a < b) \mid\mid D)$ && $(m \geq n * o))$,
the conditions are: $(a < b)$, $D$, and $(m \geq n * o)$.

## Condition Coverage (CC)

For a given condition $c$, CC is satisfied by a test suite $TS$ if it

- contains at least two tests
- one where $c$ evaluates to *false*
- one where $c$ evaluates to *true*

For a given program $p$, CC is satisfied by $TS$ if it satisfies CC for all conditions $c \in C(p)$.

# Condition Coverage

### Example

For *each condition* in $((a < b) \mid\mid D)$ && $(m \geq n * o))$,
CC is satisfied for instance if *TS* triggers executions with:

$a = 5, b = 10, D = true, m = 1, n = 1, o = 1$
and
$a = 10, b = 5, D = false, m = 1, n = 2, o = 2$

# Condition Coverage

## Example

For *each condition* in $((a < b) \,||\, D) \,\&\&\, (m \geq n * o))$,
CC is satisfied for instance if $TS$ triggers executions with:

$a = 5, b = 10, D = true, m = 1, n = 1, o = 1$
and
$a = 10, b = 5, D = false, m = 1, n = 2, o = 2$

## No subsumption

- Consider $p \,||\, q$, tests $= \{(true, false), (false, true)\}$
- Note that condition coverage does not imply decision coverage or vice versa, above example has no decision coverage.

# Modified Condition Decision Coverage, MCDC

## Modified Condition Decision Coverage, MCDC

For a given condition $c$ in decision $d$, MCDC is satisfied by a test suite $TS$ if it

- contains at least two tests,
- one where $c$ evaluates to *false*,
- one where $c$ evaluates to *true*,
- $d$ evaluates differently in both, and
- other conditions in $d$ evaluate identically in both

For a given program $p$, MCDC is satisfied by $TS$ if it satisfies MCDC for all $c \in C(p)$.

# Modified Condition Decision Coverage, MCDC

## Example

For condition $a < b$ in decision $((a < b) \mid\mid D)$ && $(m \geq n * o))$,
MCDC is satisfied for instance if $TS$ triggers executions with:

$a = 5, b = 10, D = false, m = 1, n = 1, o = 1$
and
$a = 10, b = 5, D = false, m = 8, n = 2, o = 3$

# Modified Condition Decision Coverage, MCDC

**Example**

For condition $a < b$ in decision $((a < b) \mid\mid D)$ && $(m \geq n * o))$,
MCDC is satisfied for instance if *TS* triggers executions with:

$a = 5, b = 10$, $D = \mathit{false}, m = 1, n = 1, o = 1$
and
$a = 10, b = 5$, $D = \mathit{false}, m = 8, n = 2, o = 3$

**Note:** To have MCDC for whole decision also need test-cases for conditions $D$ and $(m \geq n * o)$

(Note that the examples on slides 23 and 25 do *not* guarantee MCDC.)

# Modified Condition Decision Coverage, MCDC

## MCDC in industrial certification standard

MCDC is required in the avionics certification standard DO-178 as the criterion to test adequately Level A software (failure of which is classified as 'Catastrophic').

# Mini Quiz: Logical Coverage

Suppose a program contains the decision `if(x < 1 || y > z)`
Does the following test sets satisfy Decision Coverage, Condition Coverage and/or MCDC?

- `[x=0, y=0, z=1]` and `[x=2, y=2, z=1]`
- `[x=2, y=2, z=1]` and `[x=2, y=0, z=1]`
- `[x=2, y=2, z=2]`, `[x=0, y=0, z=1]`, `[x=2, y=0, z=0]`, `[x=2, y=2, z=1]`

# Mini Quiz: Logical Coverage

Suppose a program contains the decision `if(x < 1 || y > z)`
Does the following test sets satisfy Decision Coverage, Condition Coverage and/or MCDC?

- `[x=0, y=0, z=1]` and `[x=2, y=2, z=1]`
  CC
- `[x=2, y=2, z=1]` and `[x=2, y=0, z=1]`
- `[x=2, y=2, z=2]`, `[x=0, y=0, z=1]`, `[x=2, y=0, z=0]`, `[x=2, y=2, z=1]`

# Mini Quiz: Logical Coverage

Suppose a program contains the decision `if(x < 1 || y > z)`
Does the following test sets satisfy Decision Coverage, Condition Coverage and/or MCDC?

- `[x=0, y=0, z=1]` and `[x=2, y=2, z=1]`
  CC
- `[x=2, y=2, z=1]` and `[x=2, y=0, z=1]`
  DC
- `[x=2, y=2, z=2]`, `[x=0, y=0, z=1]`,
  `[x=2, y=0, z=0]`, `[x=2, y=2, z=1]`

# Mini Quiz: Logical Coverage

Suppose a program contains the decision `if(x < 1 || y > z)`
Does the following test sets satisfy Decision Coverage, Condition Coverage and/or MCDC?

- `[x=0, y=0, z=1]` and `[x=2, y=2, z=1]`
  CC
- `[x=2, y=2, z=1]` and `[x=2, y=0, z=1]`
  DC
- `[x=2, y=2, z=2]`, `[x=0, y=0, z=1]`, `[x=2, y=0, z=0]`, `[x=2, y=2, z=1]`
  DC, CC, MCDC

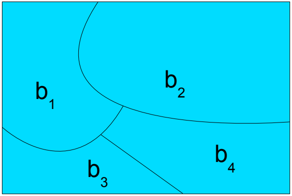
# Input Space Partitioning

- Ultimately all testing is about choosing elements from input space
- Input space partitioning takes that view in a more direct way
- Input space partitioned into regions that are assumed to contain 'equally useful values'
- Test cases contain values from each region

# Partitioning Domains

A partitioning $q$ of a domain $D$ defines a set of blocks, $B_q = \{b_1, \ldots, b_n\}$, such that:

- the blocks $b_i$ are pairwise disjoint (no overlap)
- together the blocks cover the domain D (complete)




Normally, different partitionings are combined (example later)

# Examples

Consider the domain of integer arrays.

Are the following blocks a partitioning?

- $b_1 =$ sorted in ascending order
- $b_2 =$ sorted in descending order
- $b_3 =$ arbitrary order

# Examples

Consider the domain of integer arrays.

Are the following blocks a partitioning?

- $b_1$ = sorted in ascending order
- $b_2$ = sorted in descending order
- $b_3$ = arbitrary order

Answer: no!

- The array [1] belongs to all blocks

# Combining Partitionings

When creating test cases for `findElement (int[] arr, int elem)`

partitioning $q$: `arr` is `null` ($b_{q1}$) or not ($b_{q2}$)

partitioning $r$: `arr` is empty ($b_{r1}$) or not ($b_{r2}$)

partitioning $s$: number of `elem` in `arr` is 0 ($b_{s1}$), 1 ($b_{s2}$), or $>1$ ($b_{s3}$)

# Combining Partitionings

When creating test cases for `findElement (int[] arr, int elem)`

partitioning $q$: `arr` is `null` ($b_{q1}$) or not ($b_{q2}$)

partitioning $r$: `arr` is empty ($b_{r1}$) or not ($b_{r2}$)

partitioning $s$: number of `elem` in `arr` is 0 ($b_{s1}$), 1 ($b_{s2}$), or $>1$ ($b_{s3}$)

Note:

- $r$ is a sub-partitioning of $b_{q2}$
- $b_{s2}$ and $b_{s3}$ are sub-blocks of $b_{r2}$
- $b_{s1}$ overlaps with $b_{r1}$ and $b_{q2}$ (fine, as $r$ and $s$ are different partitionings)

# Strategies for Identifying Values

After partitioning, one still has to choose values from the blocks.

**Strategies**

- Include valid, invalid and special values
- Sub-partition some blocks
- Explore boundaries of domains

# Discussion: Input Space Partitionings

Recall the method `russianMultiplication(int a, int b)`.

Suggest some input space partitionings.

# Discussion: Input Space Partitionings

Recall the method `russianMultiplication(int a, int b)`.

Suggest some input space partitionings.

E.g.

- $a \geq 0$ or $a < 0$
- $b \geq 0$ or $b < 0$
- $a \geq b$ or $a < b$

# Example

## Example

```
public static int binarySearch(int[] arr, int
 elem)
```

### Specification

*requires:* *arr !=null, arr is sorted ascendingly*
*ensures:* *if there exists an element of arr that is equal to elem then arr[result] == elem, otherwise result == -1*

- Partition 1: elem in array, elem not in array
  - Sub-partition of not in array: elem < min(array), min(array) <= elem <= max(array), max(array) < elem
  - Sub-partition of in array: left of middle, middle, right of middle

# Example (cont.)

- Partition 1: elem in array, elem not in array
  - Sub-partition of not in array: elem $<$ min(array), min(array) $<=$ elem $<=$ max(array), max(array) $<$ elem
  - Sub-partition of in array: left of middle, middle, right of middle

- Not in array:
  - binarySearch($\{1, 3, 5, 8\}, 0$); $<$
  - binarySearch($\{2, 4, 76, 40\}, 3$); in
  - binarySearch($\{34, 64, 75, 100\}, 101$); $>$
- In array:
  - binarySearch($\{0, 2, 4, 6, 44\}, 0$); left border
  - binarySearch($\{4, 5, 34, 55, 66\}, 5$); left/middle
  - binarySearch($\{4, 5, 34, 55, 66\}, 34$); middle
  - binarySearch($\{4, 5, 34, 55, 66\}, 55$); right/middle
  - binarySearch($\{3, 4, 5, 6, 432, 1000\}, 1000$); rightborder

# Input Space Partitioning

- Look at specification
- Divide input space into regions with for which the program acts similar
- Take some inputs from regions, especially from borders

Use multiple partitions, subdivide partitions when sensible

This is a guideline, not a formal procedure: use common sense to define similar, border and sensible

# Black-box Vs. White-box testing

### Black-box testing

Deriving test suites from external descriptions of the software, e.g.

- Specifications
- Requirements / Design
- Input space knowledge

Does not require knowledge of internals (i.e., source code)

# Black-box Vs. White-box testing

## Black-box testing

Deriving test suites from external descriptions of the software, e.g.

- Specifications
- Requirements / Design
- Input space knowledge

Does not require knowledge of internals (i.e., source code)

## White-box testing

Deriving test suites from the source code internals of the software, e.g.

- Conditions
- Branches in execution
- Statements

# Summary

- Control Flow Graph Coverage
  - Statement coverage: every node visited.
  - Branch coverage: every edge traversed.
  - Path coverage: every excecution path (usually too many!)
- Logic Based Coverage
  - Decision coverage: test for each decision true/false.
  - Condition coverage: each sub-expression true/false.
  - MCDC: sub-expression true/false AND affecting decision.
- Input Space Partitioning
  - Input space split into disjoint regions.

# Literature related to this Lecture

- Introduction to Software Testing - by Paul Ammann, Jeff Offutt
  - Graph coverage (Chapter 2),
  - Logic coverage (Chapter 3), and
  - Input space partitioning (Chapter 4).