

# Model-Based Testing

(DIT848 / DAT260)

Spring 2013

## Lecture 3

### White Box Testing - Coverage

**Gerardo Schneider**

Dept. of Computer Science and Engineering  
Chalmers | University of Gothenburg

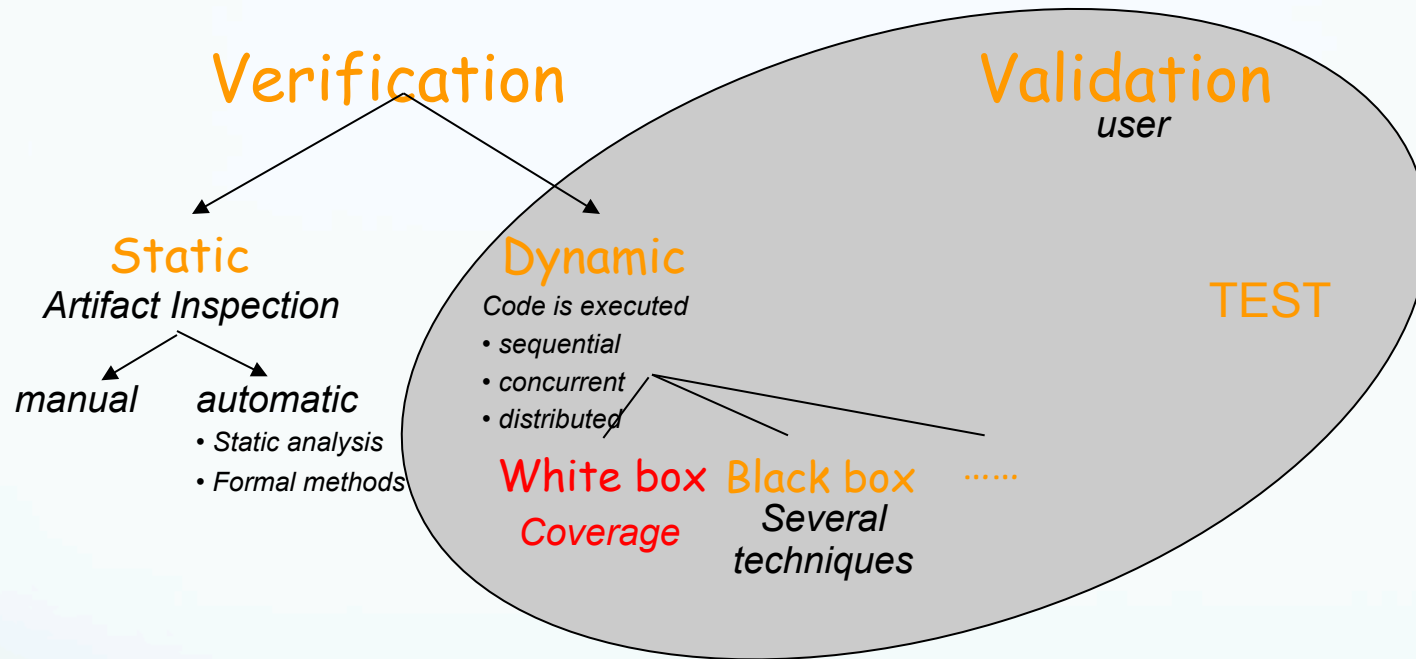
Some slides based on material by Magnus Björk, Thomas Arts and Ian Somerville

# What have we seen so far?

- V&V: Validation & Verification
- The V model
  - Test levels
- Black box testing
- (Extended) Finite State Machines

Any question?

# Today's topic



# White box testing

# Do I need more test cases?

- I think I have test cases for all aspects of the specification,
- I've added test cases for boundary values,
- ...guessed error values,
- ...and performed 10.000 random test cases.

Is that enough?

# Do I need more test cases?

- The **bad**:
  - There is no way to know for certain
- The **good**:
  - There are techniques that can help us identify some aspects that may otherwise go unnoticed!

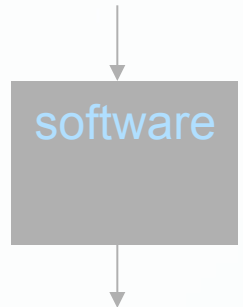


**Coverage  
techniques**

# Black box and White box testing

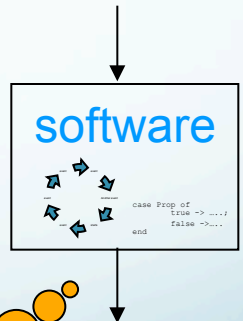
**Black box testing:** Test tactic in which the test object is addressed as a box one **cannot** open.

A test is performed by sending a **sequence of input values** and observing the output *without using any knowledge about the test object internals.*



**White box testing:** Test tactic in which the test object is addressed as a box one **can** open

A test is performed by sending a **sequence of input values** and observing the output and internals while explicitly *using knowledge about the test object internals*



Also called  
Structural testing or  
Glass box testing

# What white box testing is not

- **White box testing** is (typically) **NOT**:
  - Black box test cases that refer to internal constructs

**Id:** calc.h/pressPlus/1

**Purpose:** verifying that the correct operation is stored

**Precondition:** state is a CalcStatePtr pointing to a valid calculator state

**Action:** call pressPlus(state)

**Expected outcome:** state->op = Plus

Refers to internal representation, not interface

- Drawbacks of test cases like this:
  - Test properties not in specification
  - Fail if internal representation is changed
  - And when they fail, it may be hard to understand how to modify them
- ...but sometimes they may be necessary
  - Unit testing of internal functions



# What white box testing is

- 'Normal' **white box testing** is:
  - Black box testing, combined with tools that analyze implementation specific properties
- White box techniques covered in this lecture
  - **Code coverage analysis**
    - Are there parts of the code that are not executed by any test cases?
    - Used to find inadequacies in the test suite
  - Assignment: **EclEmma** (Java)
  - In this lecture: Some examples in C (**GCov**) and functional programming

# Coverage checking

# Coverage checking

The structure of the software is used to determine whether a set of tests is a sufficient/adequate one

Thus:

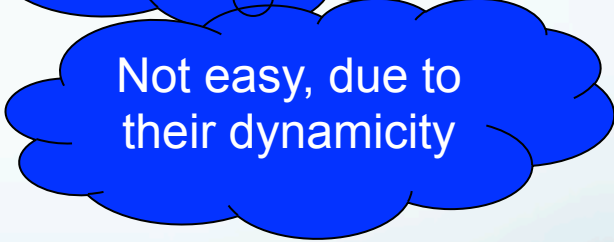
- 1) Decide which structure to consider
- 2) Decide upon **coverage criteria**
- 3) Find a set of tests such that this structure is covered according to the decided criteria

# Common structures

- **Function coverage**
  - All functions have been executed
- **Entry/exit coverage**
  - All entry and exit points of all functions have been executed
    - *Entry* points: all calls to a function
    - *Exit* points: each return statement
- **Statement coverage** (lines of code)
  - All lines of code have been executed
- **Branch coverage** (condition coverage)
  - If: both "if" and "else" part, even if no else part
  - While loop: both with true and false conditions
  - Lazy logical ops (&& and ||): first arguments both true and false
- **Path coverage**
  - All possible routes through the code (combination of branches)
  - Infinitely many if there are while loops (only feasible for small functions)
- More on later lecture...



How to cover exceptions?



Not easy, due to their dynamicity

# Example (Coverage in Functional Prog.)

- Function from (pretended) ATM system
- Representation of amount of cash in machine:
  - $[(100,23),(500,11)]$  means that machine contains:
    - 23 100kr bills
    - 11 500kr bills
  - We call it "pair-notes"
- Function to look at: *subtract*
  - subtract a number of notes - notes remaining in the ATM
  - `subtract(<list_of_pair-notes_to_withdraw>, <list_of_pair-notes_in_Bank>)`

# Example (Coverage in Functional Prog.)

```
subtract([],Notes) ->
```

```
Notes;
```

```
subtract([ {Value,Nr} | Rest ],Notes) ->
```

```
subtract(Rest,subtract2(Value,Nr,Notes)).
```

```
subtract2(Value,N,[ {Value,M} | Notes ]) when M>=N ->
```

```
[ {Value,M-N} ];
```

```
subtract2(Value,N,[ {V,M} | Notes ]) ->
```

```
[ {M,V} | subtract2(Value,N,Notes) ].
```

**Test case:** `subtract([ {500,2} ], [ {100,100}, {500,3} ])`.

**Expected output:** `[ {100,100}, {500,1} ]`

# Example (Coverage in Functional Prog.)

```
subtract([],Notes) ->
```

```
Notes;
```

```
subtract([{Value,Nr}|Rest],Notes) ->
```

```
subtract(Rest,subtract2(Value,Nr,Notes)).
```

```
subtract2(Value,N,[{Value,M}|Notes]) when M>=N ->
```

```
[{Value,M-N}];
```

```
subtract2(Value,N,[{V,M}|Notes]) ->
```

```
[{M,V}|subtract2(Value,N,Notes)].
```

**Test case:** `subtract([500,2], [{100,100}, {500,3}])`.

```
subtract([500,2], [{100,100}, {500,3}]).
```

# Example (Coverage in Functional Prog.)

```
subtract([],Notes) ->
```

```
Notes;
```

```
subtract([ {Value,Nr} | Rest ],Notes) ->
```

```
  subtract(Rest,subtract2(Value,Nr,Notes)).
```

```
subtract2(Value,N,[ {Value,M} | Notes ]) when M>=N ->
```

```
  [ {Value,M-N} ];
```

```
subtract2(Value,N,[ {V,M} | Notes ]) ->
```

```
  [ {M,V} | subtract2(Value,N,Notes) ].
```

**Test case:** `subtract([ {500,2} ], [ {100,100}, {500,3} ])`.

```
subtract([],subtract2(500,2,[ {100,100}, {500,3} ])).
```



# Example (Coverage in Functional Prog.)

```
subtract([],Notes) ->
```

```
Notes;
```

```
subtract([ {Value,Nr} | Rest ],Notes) ->
```

```
  subtract(Rest,subtract2(Value,Nr,Notes)).
```

```
subtract2(Value,N,[ {Value,M} | Notes ]) when M>=N ->
```

```
  [ {Value,M-N} ];
```

```
subtract2(Value,N,[ {V,M} | Notes ]) ->
```

```
  [ {M,V} | subtract2(Value,N,Notes) ].
```

**Test case:** `subtract([ {500,2} ], [ {100,100}, {500,3} ])`.

```
subtract([],subtract2(500,2,[ {100,100}, {500,3} ])).
```

# Example (Coverage in Functional Prog.)

```
subtract([],Notes) ->
```

```
Notes;
```

```
subtract([ {Value,Nr} | Rest],Notes) ->
```

```
subtract(Rest,subtract2(Value,Nr,Notes)).
```

```
subtract2(Value,N,[ {Value,M} | Notes]) when M>=N ->
```

```
[ {Value,M-N}];
```

```
subtract2(Value,N,[ {V,M} | Notes]) ->
```

```
[ {M,V} | subtract2(Value,N,Notes)].
```

**Test case:** `subtract([ {500,2}], [ {100,100}, {500,3}]) .`

```
subtract([],subtract2(500,2,[ {100,100}, {500,3}])) .
```

# Example (Coverage in Functional Prog.)

```
subtract([],Notes) ->
```

```
Notes;
```

```
subtract([ {Value,Nr} | Rest],Notes) ->
```

```
subtract(Rest,subtract2(Value,Nr,Notes)).
```

```
subtract2(Value,N,[ {Value,M} | Notes]) when M>=N ->
```

```
[ {Value,M-N}];
```

```
subtract2(Value,N,[ {V,M} | Notes]) ->
```

```
[ {M,V} | subtract2(Value,N,Notes)].
```

**Test case:** `subtract([ {500,2}], [ {100,100}, {500,3}]) .`

```
subtract([], [ {100,100} | subtract2(500,2, [ {500,3}])]) .
```

# Example (Coverage in Functional Prog.)

```
subtract([],Notes) ->
```

```
Notes;
```

```
subtract([ {Value,Nr} | Rest],Notes) ->
```

```
subtract(Rest,subtract2(Value,Nr,Notes)).
```

```
subtract2(Value,N,[ {Value,M} | Notes]) when M>=N ->
```

```
[ {Value,M-N} ];
```

```
subtract2(Value,N,[ {V,M} | Notes]) ->
```

```
[ {M,V} | subtract2(Value,N,Notes)].
```

**Test case:** `subtract([ {500,2} ], [ {100,100}, {500,3} ])`.

```
subtract([], [ {100,100} | subtract2(500,2, [ {500,3} ])]).
```

# Example (Coverage in Functional Prog.)

```
subtract([], Notes) ->
```

```
  Notes;
```

```
subtract([ {Value, Nr} | Rest ], Notes) ->
```

```
  subtract(Rest, subtract2(Value, Nr, Notes)) .
```

```
subtract2(Value, N, [ {Value, M} | Notes ]) when M >= N ->
```

```
  [ {Value, M-N} ] ;
```

```
subtract2(Value, N, [ {V, M} | Notes ]) ->
```

```
  [ {M, V} | subtract2(Value, N, Notes) ] .
```

**Test case:** `subtract([ {500, 2} ], [ {100, 100}, {500, 3} ]) .`

```
subtract([], [ {100, 100} | subtract2(500, 2, [ {500, 3} ]) ]) .
```

# Example (Coverage in Functional Prog.)

```
subtract([],Notes) ->
```

```
Notes;
```

```
subtract([ {Value,Nr} | Rest],Notes) ->
```

```
subtract(Rest,subtract2(Value,Nr,Notes)).
```

```
subtract2(Value,N,[ {Value,M} | Notes]) when M>=N ->
```

```
[ {Value,M-N}];
```

```
subtract2(Value,N,[ {V,M} | Notes]) ->
```

```
[ {M,V} | subtract2(Value,N,Notes)].
```

**Test case:** `subtract([ {500,2}], [ {100,100}, {500,3} ])`.

```
subtract([], [ {100,100} | [ {500,3-2} ] ]).
```

# Example (Coverage in Functional Prog.)

```
subtract([],Notes) ->
```

```
Notes;
```

```
subtract([ {Value,Nr} | Rest],Notes) ->
```

```
subtract(Rest,subtract2(Value,Nr,Notes)).
```

```
subtract2(Value,N,[ {Value,M} | Notes]) when M>=N ->
```

```
[ {Value,M-N} ];
```

```
subtract2(Value,N,[ {V,M} | Notes]) ->
```

```
[ {M,V} | subtract2(Value,N,Notes)].
```

**Test case:** `subtract([ {500,2} ], [ {100,100}, {500,3} ])`.

**Evaluates to**

```
subtract([], [ {100,100} | [ {500,1} ] ]).
```

# Example (Coverage in Functional Prog.)

```
subtract([],Notes) ->
```

```
Notes;
```

```
subtract([ {Value,Nr} | Rest],Notes) ->
```

```
subtract(Rest,subtract2(Value,Nr,Notes)).
```

```
subtract2(Value,N,[ {Value,M} | Notes]) when M>=N ->
```

```
[ {Value,M-N}];
```

```
subtract2(Value,N,[ {V,M} | Notes]) ->
```

```
[ {M,V} | subtract2(Value,N,Notes)].
```

**Test case:** `subtract([ {500,2}], [ {100,100}, {500,3}]) .`

**output:** `[ {100,100}, {500,1} ]`

All statements and all branches have been executed. Matches expected output.



# Example (Coverage in Functional Prog.)

Are we happy? Is the program correct?

What happen with the following?

**Test case:** `subtract ([{500,2}], [{100,5}, {500,3}]) .`

It will **not** work!! The case [100,100] was a particular case; we inverted values!

```
subtract ([],Notes) ->
```

```
  Notes;
```

```
subtract ([{Value,Nr}|Rest],Notes) ->
```

```
  subtract (Rest,subtract2 (Value,Nr,Notes)) .
```

```
subtract2 (Value,N, [{Value,M}|Notes])  when M>=N ->
```

```
  [{Value,M-N}];
```

```
subtract2 (Value,N, [{V,M}|Notes]) ->
```

```
  [{V,M} | subtract2 (Value,N,Notes)] .
```

# Example (Coverage in Functional Prog.)

Are we happy now? Is the program correct?

What happen with the following?

**Test case:** `subtract ([{100,2}], [{100,100},{500,3}]) .`

It will **not** work!! We are "loosing" the suffix of the list!

```
subtract([],Notes) ->
```

```
  Notes;
```

```
subtract([ {Value,Nr} | Rest],Notes) ->
```

```
  subtract(Rest,subtract2(Value,Nr,Notes)) .
```

```
subtract2(Value,N,[ {Value,M} | Notes]) when M>=N ->
```

```
  [ {Value,M-N} | Notes] ;
```

```
subtract2(Value,N,[ {V,M} | Notes]) ->
```

```
  [ {M,V} | subtract2(Value,N,Notes)] .
```

# Coverage (example in C)

```
void printPos(int n) {  
    printf("This is ");  
    if (n < 0)  
        printf("not ");  
    printf("a positive integer.\n");  
    return;  
}
```

Should be: <=

Actually:  
else { }

Code originally from Wikipedia

## Test case 3

Action: call printPos(0)

Expected outcome:

"This is not a positive integer" (printed on stdout)

Boundary value

Fails!



Are we  
happy with  
coverage?

## Test case 1

Action: call printPos(-1)

Expected outcome:

"This is not a positive integer" (printed on stdout)

**Coverage:** 100% statement,  
50% branch, 50% path

## Test case 2

Action: call printPos(1)

Expected outcome:

"This is a positive integer" (printed on stdout)

**Coverage:** 100% statement,  
branch & path (including previous)

# Group exercise

- Come up with pieces of code (in any language) and a few test cases such that conditions are met, or motivate why it is impossible:
  1. 100% branch coverage, less than 100% path coverage
  2. 100% path coverage, less than 100% statement coverage
  3. 100% function coverage, less than 100% exit point coverage

Groups 2-5 persons: 15 min

# Suggestions

1: 100% branch coverage, less than 100% path coverage

```
void foo(int n) {  
    if(n>0)  
        printf("Positive\n");  
    else  
        printf("Not positive\n");  
    if(n % 2)  
        printf("Odd\n");  
    else  
        printf("Even\n");  
}
```

**Id:** Test case 1: pos/odd

**Action:** call foo(1)

**Expected outcome:**

"Positive" and "Odd"

**Id:** Test case 2: neg/even

**Action:** call foo(-2)

**Expected outcome:**

"Not positive" and "Even"

Path pos/even **not**  
covered!

# Suggestions

2: 100% path coverage, less than 100% statement coverage

```
int main(void) {  
    printf("Hello world\n");  
    return 0;  
    printf("Unreachable code\n"); }  
}
```

**Id:** Test case 1

**Action:** run main

**Expected outcome:**

"Hello world" printed

This statement is **not** covered!

# Suggestions

3: 100% function coverage, less than 100% exit point coverage

```
int abs(int n) {  
    if(n < 0)  
        return -n;  
    return n;  
}
```

**Id:** abs/1

**Action:** call abs(-17)

**Expected outcome:**  
returns 17

Didn't cover  
this exit point

# White box test design

Strategy for using **coverage measure**:

1. Design test cases using *black box test design* techniques
2. Measure code coverage
3. Design test cases by inspecting the code to cover unexecuted code

100% coverage does **not** mean there are no errors left!

So, code coverage should be seen as complementary method -  
It cannot do the thinking for you

However, coverage analysis catches aspects that are otherwise easily forgotten



# Adding test cases after coverage analysis

- The new test cases should still be black box test cases, not referring to the code

Good test case:

*Id: abs/2*

*Purpose: Execute abs on negative integer*

*Action: call abs(-17)*

*Expected outcome: Call returns 17*

Bad test case:

*Id: abs/2*

*Purpose: Cover line 3 of abs*

*Action: call abs(-17)*

*Expected outcome: Line 3 executed*

Refers to code

# Practical coverage analysis

In order to measure coverage, most languages require a compile flag to enable keeping track of line numbers during execution

Consequences:

- Performance changes, hence **timing related faults** may be *undiscoverable*
- Memory requirements change, hence one may experience problems running in embedded devices

There are a lot of tools available for many languages

# Coverage vs Profiling

Both methods count executions of entities, but purpose is different:

- **Coverage tool**: find out which entities have been executed, to establish confidence in verification
- **Profiler**: identify bottlenecks and help programmer improve performance of software

# Example: Gcov (C)

The program **avg** (short for “average”) reads a text file, whose name is given as a command line argument, containing a number of integers, and reports the average value of all the integers. The program has been implemented in C (see below and next page), and the following small test suite has been developed by a programmer to start testing the system:

**Test case avg.1:** Normal integers

**Prerequisites:** The file avgtest1.txt contains “10 15 35”

**Action:** Run ./avg avgtest1.txt

**Expected outcome:** The program prints “The average is 20”

**Test case avg.2:** Negative numbers

**Prerequisites:** The file avgtest2.txt contains “-2 2 -6”

**Action:** Run ./avg avgtest2.txt

**Expected outcome:** The program prints “The average is -2”

Executing this test suite together with gcov reveals that there is untested code, the tool giving the message “Lines executed: 63.33% of 30”. The actual output from gcov can be seen in next slide.

NOTE: The uncovered statements are those lines preceded with #####

# Example: GCov

```
-: 1:#include <stdio.h>
-: 2:#include <stdlib.h>
-: 3:
-: 4:// readInts: read a file containing integers, and return their
-: 5://      sum and the number of integers read.
-: 6:
-: 7:#define READINTS_SUCCESS 0 // Indicates success
-: 8:#define READINTS_FILEERR 1 // the file could not be read
-: 9:#define READINTS_SYNTAXERR 2 // syntax error in file
-: 10:
2: 11:int readInts(const char* filename, int* sumRslt, int* lengthRslt){
2: 12: FILE* file = fopen(filename, "r");
2: 13: if(!file)
#####: 14:     return READINTS_FILEERR;
-: 15:
2: 16: *sumRslt=0;
2: 17: *lengthRslt=0;
-: 18: while(1) {
-: 19:     int theInt;
8: 20:     if(fscanf(file, "%d", &theInt) == 1) {
-: 21:         // Successfully read integer
6: 22:         (*sumRslt) += theInt;
6: 23:         (*lengthRslt)++;
-: 24:     } else {
-: 25:         // Could not read integer. End of file or syntax error?
2: 26:         if(!feof(file)) {
-: 27:             // End of file
2: 28:             fclose(file);
2: 29:             return READINTS_SUCCESS;
-: 30:         } else {
-: 31:             // Syntax error
#####: 32:             fclose(file);
#####: 33:             return READINTS_SYNTAXERR;
-: 34:         }
-: 35:     }
6: 36: }
-: 37:}
-: 38:
```

```
2: 39:int main(int argc, char**argv) {
-: 40:     int sum, length;
-: 41:     const char* filename;
-: 42:
2: 43:     if(argc < 2) {
#####: 44:         printf("Error: missing argument\n");
#####: 45:         exit(EXIT_FAILURE);
-: 46:     }
2: 47:     filename = argv[1];
-: 48:
2: 49:     switch(readInts(filename, &sum, &length)) {
-: 50:     case READINTS_FILEERR:
#####: 51:         printf("Error reading file %s\n", filename);
#####: 52:         exit(EXIT_FAILURE);
-: 53:
-: 54:     case READINTS_SYNTAXERR:
#####: 55:         printf("Syntax error in file %s\n", filename);
#####: 56:         exit(EXIT_FAILURE);
-: 57:
-: 58:     case READINTS_SUCCESS:
-: 59:     default:
-: 60:         break;
-: 61:     }
-: 62:
2: 63:     if(length==0) {
#####: 64:         printf("Error: no integers found in file %s\n",
filename);
#####: 65:         exit(EXIT_FAILURE);
-: 66:     }
-: 67:
2: 68:     printf("The average is %d\n", sum / length);
-: 69:
2: 70:     return EXIT_SUCCESS;
-: 71:}
```

# Group exercise

- Provide additional test cases so that all cases together yield 100% statement coverage
- Write complete test cases as shown in the test cases above, and indicate which lines each test case cover

Groups 2-5 persons: 15 min

# Exercise: Proposed solution

-To cover l.64-65 (avgtest3.txt is an empty file - Test case avg3:

Prerequisites: The file avgtest3.txt exists but is empty

Action: `./avg avgtest3.txt`

Expected outcome: An error is reported, stating that the input is empty

- To cover l.32-33 and 55-56 - Test case avg4:

Prerequisites: avgtest4.txt contains a list of non-integers

Action: `./avg avgtest4.txt`

Expected outcome: An error message is given that there is a syntax error

- To cover l.14 and 51-52 - Test case avg5:

Prerequisites: Call the function with an argument, not a file

Action: `./avg "asdfdf"` (or `./avg non_existing_file.txt`)

Expected outcome: An error reading file could be given

- To cover l.44-45 Test case avg5:

Prerequisites: None

Action: `./avg`

Expected outcome: Error missing argument is given



Any problem understanding the solution? **Try it yourself with GCoV!**

# Terminology

