SOFTWARE ENGINEERING

Week 09
Software Testing

Dr. A. Cüneyd TANTUĞ
Dr. Tolga OVATMAN
Istanbul Technical University
Computer Engineering Department

Agenda

Implementation/Programming Guidelines

9.1

Good Programming Practice

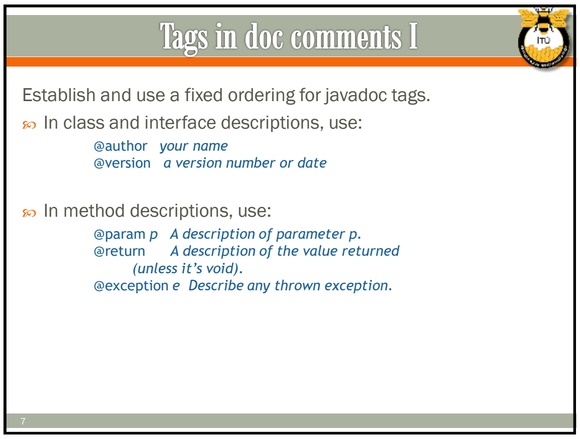
- Use of *consistent* and *meaningful* variable names
  - "Meaningful" to future maintenance programmers
  - "Consistent" to aid future maintenance programmers

Use of Consistent and Meaningful Variable Names

- A code artifact includes the variable names `freqAverage`, `frequencyMaximum`, `minFr`, `frqncyTotl`
- A maintenance programmer has to know if `freq`, `frequency`, `fr`, `frqncy` all refer to the same thing
  - If so, use the identical word, preferably `frequency`, perhaps `freq` or `frqncy`, but *not* `fr`
  - If not, use a different word (e.g., `rate`) for a different quantity

Consistent and Meaningful Variable Names

- We can use `frequencyAverage`, `frequencyMaximum`, `frequencyMinimum`, `frequencyTotal`
- We can also use `averageFrequency`, `maximumFrequency`, `minimumFrequency`, `totalFrequency`
- But all four names must come from the same set



Tags in doc comments I

Establish and use a fixed ordering for javadoc tags.

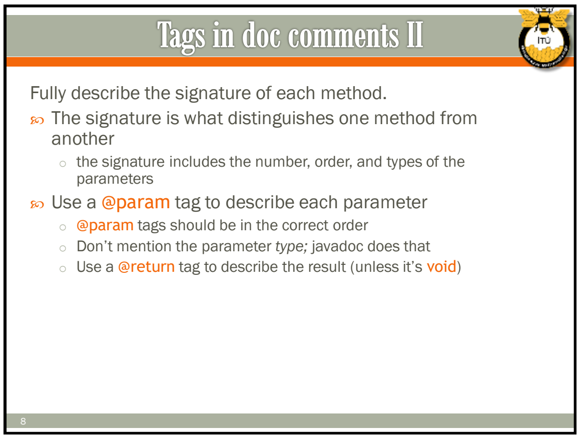
- In class and interface descriptions, use:

  @author *your name*
  @version *a version number or date*

- In method descriptions, use:

  @param p *A description of parameter p.*
  @return *A description of the value returned (unless it's void).*
  @exception e *Describe any thrown exception.*

7

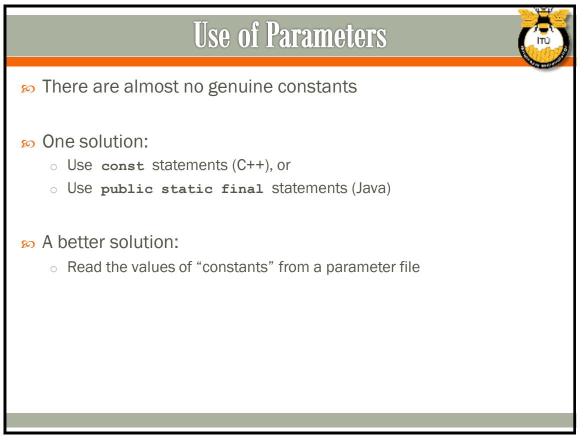


Tags in doc comments II

Fully describe the signature of each method.

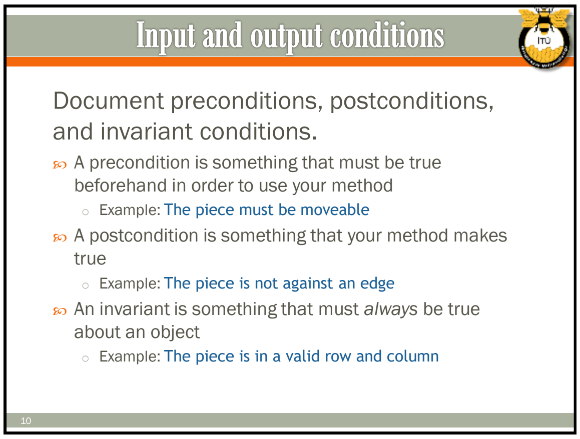
- The signature is what distinguishes one method from another
  - the signature includes the number, order, and types of the parameters
- Use a @param tag to describe each parameter
  - @param tags should be in the correct order
  - Don't mention the parameter *type;* javadoc does that
  - Use a @return tag to describe the result (unless it's void)

8



Use of Parameters

- There are almost no genuine constants
- One solution:
  - Use `const` statements (C++), or
  - Use `public static final` statements (Java)
- A better solution:
  - Read the values of "constants" from a parameter file

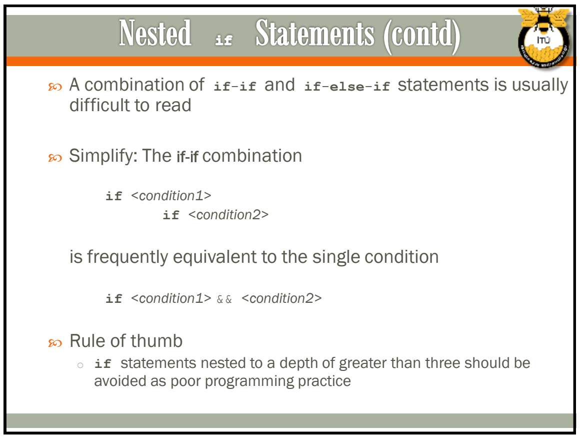


Input and output conditions

Document preconditions, postconditions, and invariant conditions.

- A precondition is something that must be true beforehand in order to use your method
  - Example: The piece must be moveable
- A postcondition is something that your method makes true
  - Example: The piece is not against an edge
- An invariant is something that must *always* be true about an object
  - Example: The piece is in a valid row and column

10



Nested `if` Statements (contd)

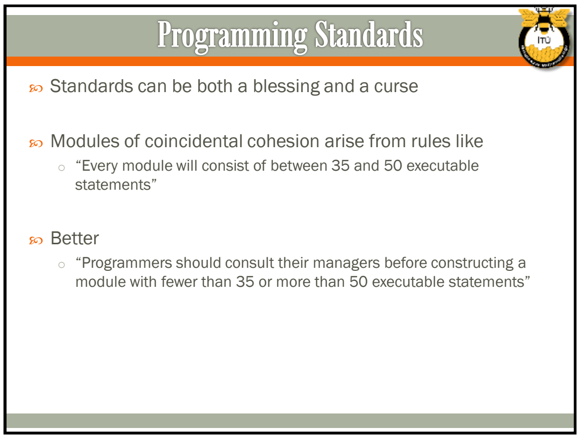
- A combination of `if-if` and `if-else-if` statements is usually difficult to read
- Simplify: The if-if combination

  ```
  if <condition1>
          if <condition2>
  ```

  is frequently equivalent to the single condition

  ```
  if <condition1> && <condition2>
  ```

- Rule of thumb
  - `if` statements nested to a depth of greater than three should be avoided as poor programming practice



Programming Standards

- Standards can be both a blessing and a curse
- Modules of coincidental cohesion arise from rules like
  - "Every module will consist of between 35 and 50 executable statements"
- Better
  - "Programmers should consult their managers before constructing a module with fewer than 35 or more than 50 executable statements"

Examples of Good Programming Standards

- “Nesting of `if` statements should not exceed a depth of 3, except with prior approval from the team leader”
- “Modules should consist of between 35 and 50 statements, except with prior approval from the team leader”
- “Use of `goto` should be avoided. However, with prior approval from the team leader, a forward `goto` may be used for error handling”

Software Testing Concepts

9.2

Software Testing

- Testing is the process of executing a program to find errors.
- Testing is planned by defining “Test Cases” in a systematic way.
- **A test case is a collection of input data and expected output.**

Phases of Testing

1. Unit Testing

Does each module do what it supposed to do?

2. Integration Testing

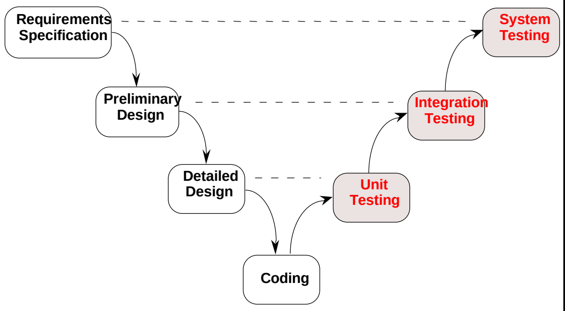
Do you get the expected results when the parts are put together?

3. System Testing

Does it work within the overall system?

Does the program satisfy the requirements ?

Levels of Testing



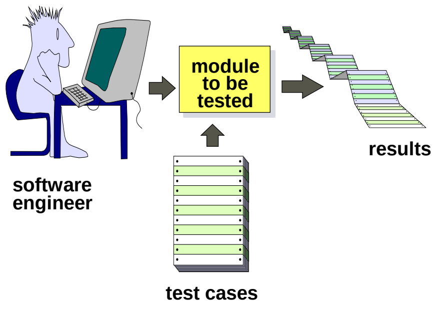
Unit Testing

9.3

Unit Testing

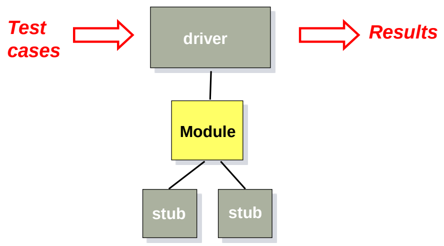
- Static Analysis:
  - Hand execution (Reading the source code)
  - Code inspection (Walk-through)
  - Automated tools checking for syntactic and semantic errors
- Dynamic Analysis:
  - Black-box testing (Test the **input/output** behavior)
  - White-box testing (Test the internal **logic**)

Unit Testing



Unit Test Environment

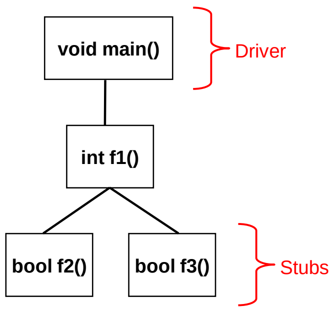
- Stubs and drivers should be written in unit testing.



Stub modules and driver modules

- A stub is an empty (dummy) module, which:
  - Just prints a message (“Module X called"), or
  - Returns pre-determined values from pre-planned test cases.
- A driver is a caller module, which calls its stubs:
  - Once or several times,
  - Each time checking the value returned.

Example: Stubs and driver (1)



Example: Stubs and driver (2)

```
#include <stdio.h>
#include <math.h>
#define TRUE 1
#define FALSE 0

// f2 is a stub
bool f2(int X)
{
 return TRUE;
}

// f3 is a stub
bool f3(int X)
{
 return TRUE;
}
```

```
// We are testing f1
int f1(int X)
{
  if (f2(X)  || f3(X))
     printf("%f",sqrt(X));
  else
     printf("Invalid value for X");
}

// This is the driver.
void main()
{
  f1(49);
}
```

Unit Testing Methods

- Black-box testing
  - Testing of the software interfaces
  - Used to demonstrate that
    - functions are operational
    - input is properly accepted and output is correctly produced
- White-box testing
  - Close examination of procedural details
  - Logical paths through the software is tested by test cases that exercise specific sets of conditions and loops

25

Verification and Validation

- Verification: Are we building the product right?
- Validation: Are we building the right product?

- White box testing is used for verification since it is done at the level of the implementation to ensure that the code performs correctly.

- Black box testing is used for validation since the tests are done at the interface level and can therefore test the requirements. It can also be used for verification in unit testing to check that the implementation satisfies the design.

26

Module Integration and Testing Strategies

9.4

A strategic approach to Testing

- Testing begins at the module level and works outward toward the integration of the entire system.

- Different testing techniques are appropriate at different points in time.

- Testing should be conducted by the developer of the software and also by an independent test group.

28

Strategies for Integration Testing

1) Big bang approach:

- All modules are fully implemented and combined as a whole, then tested as a whole. It is not practical.

2) Incremental approach: (Top-down or Bottom-up)

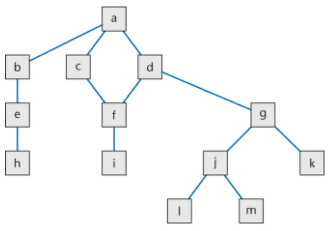
- Program is constructed and tested in small clusters.
- Errors are easier to isolate and correct.
- Interfaces between modules are more likely to be tested completely.
- After each integration step, a regression test is conducted.

29

Big bang integration testing

Integration and testing at once:

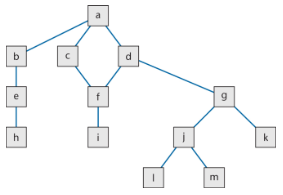
1. a, b, c, …, l, m



30

Top-down Integration

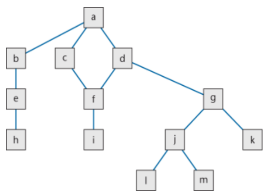
- If code artifact `mAbove` sends a message to artifact `mBelow`, then `mAbove` is implemented and integrated before `mBelow`
- One possible top-down ordering is
  - `a, b, c, d, e, f, g, h, i, j, k, l ,m`



Top-down Integration (contd)

- Another possible top-down ordering is

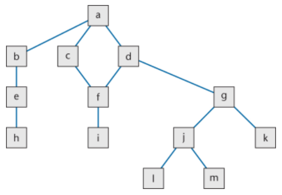
```
         a
[a]      b, e, h
[a]      c ,d, f, i
[a, d]   g, j, k, l, m
```



Bottom-up Integration

- If code artifact `mAbove` calls code artifact `mBelow`, then `mBelow` is implemented and integrated before `mAbove`
- One possible bottom-up ordering is

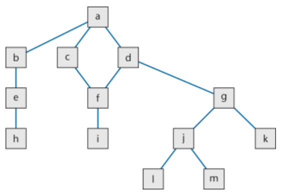
```
l, m,
h, i, j, k,
e, f, g,
b, c, d,
a
```



Bottom-up Integration

- Another possible bottom-up ordering is

```
h, e, b
i, f, c, d
l, m, j, k, g      [d]
a     [b, c, d]
```



Sandwich Integration

- Logic artifacts are integrated top-down
- Operational artifacts are integrated bottom-up
- Finally, the interfaces between the two groups are tested

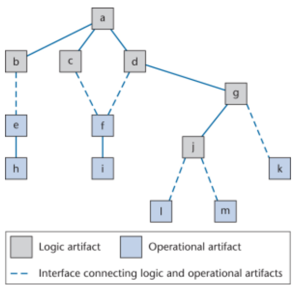


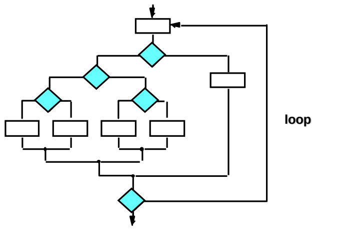
Figure 15.7

Testing Approaches

9.5

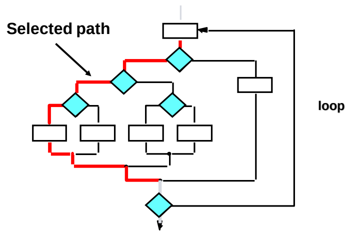
Exhaustive Testing

- There are 5 separate paths inside the following loop.
- Assuming loop <= 20, there are $5^{20} \approx 10^{14}$ possible paths.
- Exhaustive testing is not feasible.



Selective Testing

- Instead of exhausting testing, a selective testing is more feasible.



Testing to Specifications versus Testing to Code

- There are two extremes to testing
- *Test to code* (also called glass-box, logic-driven, structured, or path-oriented testing)
  - Ignore the specifications — use the code to select test cases
- *Test to specifications* (also called black-box, data-driven, functional, or input/output driven testing)
  - Ignore the code — use the specifications to select test cases

Example: Script for Unit Testing

- This is the format for a test plan to show what you're planning to do.
- It should to be filled to show what happened when you run tests.

Scenario # : 1 Tester: AAA BBB Date of Test: 01/01/2010

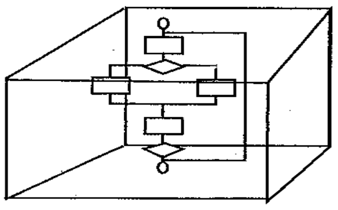
Test Number	Test Description / Input	Expected Result	Actual Result	Fix Action
1	Invalid file name	"Error: File does not exist"		
2	Valid filename, but file is binary	"Error: File is not a text file"		
3	Valid filename	"Average = 99.00"		
4				
5				

White-Box Testing

9.5.1

White-Box Testing

- Our goal is to ensure that all statements and conditions have been executed at least once.
- **Code coverage:** At a minimum, every line of code should be executed by at least one test case.

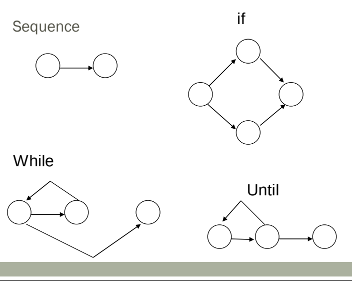


Flow graph for White box testing

- To help the programmer to systematically test the code
  - Each branch in the code (such as if and while statements) creates a node in the graph
  - The testing strategy has to reach a targeted coverage of statements and branches; the objective can be to:
    - cover all possible paths (often infeasible)
    - cover all possible nodes (simpler)
    - cover all possible edges (most efficient)

Flow Graph Notation

- Each circle represents one or more nonbranching source code statements.



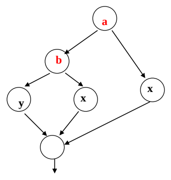
Compound Logic

If **a** OR **b**

then procedure x

else procedure y

ENDIF



Cyclomatic Complexity

- Cyclomatic Complexity V(G) is defined as the number of regions in the flow graph.

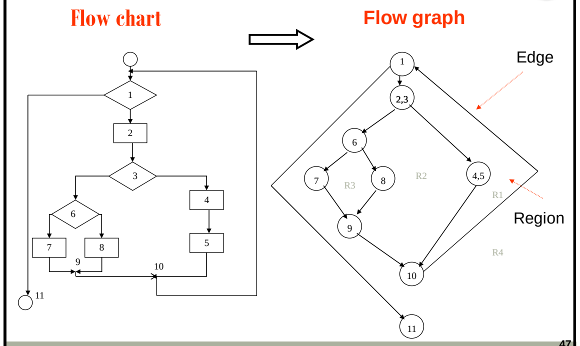
  $V(G) = E - N + 2$

  E: number of edges in flow graph
  N: number of nodes in flow graph

- Another method:
  $V(G) = P + 1$

  P: number of predicate nodes (simple decisions) in flow graph

Example-1

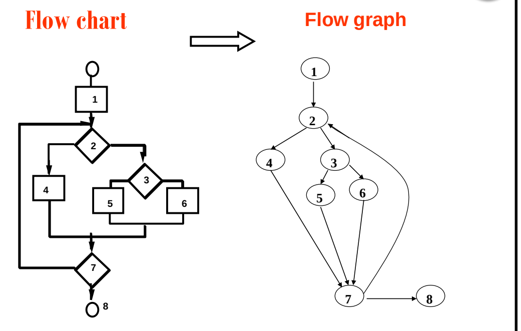


Example-1 (Cyclomatic Complexity)

- $V(G) = E - N + 2$
  $= 11 - 9 + 2 = 4$

- Other method:
  $V(G) = P + 1$
  $= 3 + 1 = 4$

Example-2



Example-2 (Cyclomatic Complexity)

First, we calculate V(G):
V(G) = E – N + 2
= 10 – 8 + 2 = 4

Now, we derive the independent paths :

Since V(G) = 4, there are four paths

Path 1 : 1,2,3,6,7,8
Path 2 : 1,2,3,5,7,8
Path 3 : 1,2,4,7,8
Path 4 : 1,2,4,7,2,4,...7,8