Testing Theory

Lecture 8

Software Engineering
TDDC88/TDDC93
autumn 2008

Mariam Kamkar
Department of Computer and Information Science
Linköping University, Sweden
marka@ida.liu.se

Theory Lecture Plan

L1 - Course Introduction and Overview
L2 - Project Management
L3 - Requirements
L4 - Acceptance Testing and Quality Factors
L5 – UML
L6 - Design Patterns
L7 - System Design and Architecture

L8 – Testing Theory

L9 - Testing in Practice
L10 - Inspection
L11 - Software Life Cycles and Configuration Management
L12 - Software Quality Management
L13 - Course Summary, Exam examples, Questions

Part I
Introduction,
Testing Process

Part II
Unit Testing:
Black-box, White-box

Part III
Module Testing
(Integration testing)

Part IV
System Testing
Part I
Introduction, Testing Process

Part II
Unit Testing: Black-box, White-box

Part III
Module Testing (Integration testing)

Part IV
System Testing

A Software Life-cycle Model
Which part will we talk about today?

- Requirements
  - Validate Requirements, Verify Specification
  - System Design (Architecture, High-level Design)
  - Module Design (Program Design, Detailed Design)
  - Implementation of Units (classes, procedures, functions)

- System Testing (Integration testing of modules)
- Module Testing (Integration testing of units)

- Acceptance Test (Release testing)

Project Management, Software Quality Assurance (SQA), Supporting Tools, Education

Agenda - What will you learn today?

Part I
Introduction, Testing Process

Part II
Unit Testing: Black-box, White-box

Part III
Module Testing (Integration testing)

Part IV
System Testing
Part I
Introduction, Testing Process

Triangle program (simple version)

Student Worksheet

- triangle problem is the most widely used example in software testing literature.

- The program accepts three integers, $a$, $b$, and $c$ as input. The three values are interpreted as representing the lengths of sides of a triangle. The program prints a message that states whether the triangle is scalene (oregelbunden), isosceles (likbent) or equilateral (liksidig).

- On a sheet of paper, write a set of test cases (i.e., specific sets of data) that you feel would adequately test this program.

<table>
<thead>
<tr>
<th>Test case</th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>Expected output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>isosceles (likbent)</td>
</tr>
<tr>
<td>2</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Part I
Introduction, Testing Process

Part II
Unit Testing:
Black-box, White-box

Part III
Module Testing
(Integration testing)

Part IV
System Testing

Mariam Kamkar
marka@ida.liu.se
Testing a ballpoint pen

- Does the pen write in the right color, with the right line thickness?
- Is the logo on the pen according to company standards?
- Is it safe to chew on the pen?
- Does the click-mechanism still work after 100,000 clicks?
- Does it still write after a car has run over it?

What is expected from this pen?

**Intended use!!**
Part I
Introduction, Testing Process

Part II
Unit Testing:
Black-box, White-box

Part III
Module Testing:
(Integration testing)

Part IV
System Testing

Error, Fault, Failure

Human error (Mistake, Bug)

Can lead to

Fault (Defect, Bug)

Can lead to

Failure

Types of Faults
(dep. on org. IBM, HP)

- Algorithmic: division by zero
- Computation & Precision: order of op
- Documentation: doc - code
- Stress/Overload: data-str size (dimensions of tables, size of buffers)
- Capacity/Boundary: x devices, y parallel tasks, z interrupts
- Timing/Coordination: real-time systems
- Throughout/Performance: speed in req.
- Recovery: power failure
- Hardware & System Software: modem
- Standards & Procedure: organizational standard; difficult for programmers to follow each other.
Part I
Introduction, Testing Process

Part II
Unit Testing: Black-box, White-box

Part III
Module Testing (Integration testing)

Part IV
System Testing

Part II
Unit Testing: Black-box, White-box testing
Unit & Integration Testing

Objective: to ensure that code implemented the design properly.

Code = System

Part I
Introduction, Testing Process

Part II
Unit Testing: Black-box, White-box

Part III
Module Testing (Integration testing)

Part IV
System Testing

Component code

Unit test
Tested components

Integration test
Integrated modules

Part I
Introduction, Testing Process

Part II
Unit Testing: Black-box, White-box

Part III
Module Testing (Integration testing)

Part IV
System Testing
Two Types of Oracles

- **Human**: an expert that can examine an input and its associated output and determine whether the program delivered the correct output for this particular input.

- **Automated**: a system capable of performing the above task.
Black-box Testing

1. Exhaustive testing
2. Equivalence class testing (Equivalence Partitioning)
3. Boundary value analysis
4. Decision table testing

Black-box / Closed-box Testing

- incorrect or missing functions
- interface errors
- performance error
1. Exhaustive testing

- **Definition**: testing with every member of the input value space.

- Input value space: the set of all possible input values to the program.

2. Equivalence Class Testing

- Equivalence Class (EC) testing is a technique used to reduce the number of test cases to a manageable level while still maintaining reasonable test coverage.

- Each EC consists of a set of data that is treated the same by the module or that should produce the same result. Any data value within a class is equivalent, in terms of testing, to any other value.
Identifying the Equivalence Classes

Student Worksheet

Taking each input condition (usually a sentence or phrase in the specification) and partitioning it into two or more groups:

- Input condition
  - range of values $x$: 1-50
- Valid equivalence class
- Invalid equivalence classes

Guidelines

1. If an input condition specifies a range of values; identify one valid EC and two invalid EC.
2. If an input condition specifies the number (e.g., one through 6 owners can be listed for the automobile); identify one valid EC and two invalid EC (- no owners; - more than 6 owners).
3. If an input condition specifies a set of input values and there is reason to believe that each is handled differently by the program; identify a valid EC for each and one invalid EC.
4. If an input condition specifies a "must be" situation (e.g., first character of the identifier must be a letter); identify one valid EC (it is a letter) and one invalid EC (it is not a letter).
5. If there is any reason to believe that elements in an EC are not handled in an identical manner by the program, split the equivalence class into smaller equivalence classes.
Identifying the Test Cases

1. Assign a unique number to each EC.

2. Until all valid ECs have been covered by test cases, write a new test case covering as many of the uncovered valid ECs as possible.

3. Until all invalid ECs have been covered by test cases, write a test case that cover one, and only one, of the uncovered invalid ECs.

---

Equivalence partitioning

- Invalid inputs
- Valid inputs
- Outputs
Student Worksheet

Specification: the program accepts four to eight inputs which are 5 digit integers greater than 10000.

3. Boundary Value Testing

Boundary value testing focuses on the boundaries simply because that is where so many defects hide. The defects can be in the requirements or in the code.

The most efficient way of finding such defects, either in the requirements or the code, is through inspection (Software Inspection, Gilb and Graham’s book).
Boundary value analysis

Student Worksheet

<table>
<thead>
<tr>
<th>Less than 10000</th>
<th>Between 10000 and 99999</th>
<th>More than 99999</th>
</tr>
</thead>
</table>

1. Identify the ECs.

2. Identify the boundaries of each EC.

3. Create test cases for each boundary value by choosing one point on the boundary, one point just below the boundary, and one point just above the boundary.
4. Decision Table Testing

Decision tables are an excellent tool to capture certain kinds of system requirements and to document internal system design. They are used to record complex business rules that a system must implement.

In addition, they can serve as a guide to creating test cases.

The general format of a decision table:

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Rule 1</th>
<th>Rule 2</th>
<th>...</th>
<th>Rule P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Condition-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Condition-2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Condition-m</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Actions</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Action-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Action-2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Action-n</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A decision table with "don't care" entry

<table>
<thead>
<tr>
<th>Rule 1</th>
<th>Rule 2</th>
<th>Rule 3, 4</th>
<th>Rule 5</th>
<th>Rule 6</th>
<th>Rules 7, 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>C2</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>C3</td>
<td>T</td>
<td>F</td>
<td>_</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>A1</td>
<td>X</td>
<td>X</td>
<td>_</td>
<td>_</td>
<td>X</td>
</tr>
<tr>
<td>A2</td>
<td>X</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>X</td>
</tr>
<tr>
<td>A3</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>X</td>
</tr>
<tr>
<td>A4</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>X</td>
</tr>
</tbody>
</table>

• _ : “don't care” entry. The don't care entry has two major interpretations: the condition is irrelevant, or the condition does not apply. Sometimes the “n/a” symbol for this latter interpretation.

A decision table converted to a test case table:

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Condition-1</th>
<th>Condition-2</th>
<th>...</th>
<th>Condition-m</th>
<th>Expected Results</th>
<th>Action-1</th>
<th>Action-2</th>
<th>...</th>
<th>Action-n</th>
<th>Test Case 1</th>
<th>Test Case 2</th>
<th>...</th>
<th>Test Case P</th>
</tr>
</thead>
</table>

# Test Cases for the Triangle Problem

## Student Worksheet

<table>
<thead>
<tr>
<th>Case ID</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>Expected output</th>
</tr>
</thead>
<tbody>
<tr>
<td>DT1</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>Not a Triangle</td>
</tr>
</tbody>
</table>

## Test Cases for the Triangle Problem

### Student Worksheet

<table>
<thead>
<tr>
<th>Case ID</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>Expected output</th>
</tr>
</thead>
<tbody>
<tr>
<td>DT1</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>Not a Triangle</td>
</tr>
<tr>
<td>DT2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DT3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DT4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DT5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DT6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DT7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DT8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DT9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DT10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DT11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
White-box Testing
(Glass box testing, Open box testing, Clear box testing, Structural testing)

1. Control flow testing
2. Data flow testing

Control Flow Graphs

- Process blocks
- Decision Point
- Junction Point
- Sequence
- While
- Until
- Case
**Definition**: Given a program written in an imperative programming language, its program graph is a directed graph in which nodes are statement fragments, and edges represent flow of control (a complete statement is a "default" statement fragment).

---

**Code Coverage**

(test coverage metrics)

**Levels of Coverage:**
- Statement/Line/Basic block/Segment Coverage
- Decision (Branch) Coverage
- Condition Coverage
- Decision/Condition Coverage
- Path Coverage
Part I
Introduction, Testing Process

Part II
Unit Testing: Black-box, White-box

Part III
Module Testing (Integration testing)

Part IV
System Testing

---

**Student Worksheet**

```
Begin
if ( y >= 0)
    then y = 0;
    abs = y;
end;
```

**test case-1 (yes):**

<table>
<thead>
<tr>
<th>input:</th>
<th>y = ?</th>
</tr>
</thead>
<tbody>
<tr>
<td>expected result:</td>
<td>?</td>
</tr>
<tr>
<td>actual result:</td>
<td>?</td>
</tr>
</tbody>
</table>

---

**What is Wrong with Line Coverage**

Steve Cornett (Bullseye testing technology)

Software developers and testers commonly use line coverage because of its simplicity and availability in object code instrumentation technology.

Of all the structural coverage criteria, line coverage is the weakest, indicating the fewest number of test cases.

Bugs can easily occur in the cases that line coverage cannot see.

The most significant shortcoming of line coverage is that it fails to measure whether you test simple if statements with a false decision outcome. Experts generally recommend to only use line coverage if nothing else is available. Any other measure is better.
Part I
Introduction, Testing Process

Part II
Unit Testing: Black-box, White-box

Part III
Module Testing (Integration testing)

Part IV
System Testing

Branch Coverage

Student Worksheet

Begin
if ( y >= 0 )
    then y = 0;
    abs = y;
end;

**test case-1 (yes):**
*input: y = 0*
*expected result: 0*
*actual result: 0*

**test case-2 (no):**
*input: y = ?*
*expected result: ?*
*actual result: ?*

... more conditions?
Part I
Introduction, Testing Process

Part II
Unit Testing:
Black-box, White-box

Mariam Kamkar
marka@ida.liu.se

Part III
Module Testing
(Integration testing)

Part IV
System Testing

---

**Condition Coverage**

**Student Worksheet**

Begin
if ( x < 10 && y > 20 ) {
    z = foo (x,y); else z = fie (x,y);
} end;

**test case-1 (T,F):**
input: x = ?, y = ?
expected result: ?
actual result: ?

**test case-2 (F,T):**
input: x = ?, y = ?
expected result: ?
actual result: ?

---

**Decision/Condition Coverage**

**Student Worksheet**

Begin
if ( x < 10 && y > 20 ) {
    z = foo (x,y); else z = fie (x,y);
} end;

**test case-1 (T,T, yes):**
input: x = ?, y = ?
expected result: ?
actual result: ?

**test case-2 (F,F, no):**
input: x = ?, y = ?
expected result: ?
actual result: ?
Path Coverage

- A path is a sequence of branches, or conditions.
- A path corresponds to a test case, or a set of inputs.
- In code coverage testing, branches have more importance than the blocks they connect.
- Bugs are often sensitive to branches and conditions.

Path with loops
All possible execution paths

Question: How do we know how many paths to look for?

Answer: The computation of cyclomatic complexity

Cyclomatic complexity has a foundation in graph theory and is computed in the following ways:

1. Cyclomatic complexity \( V(G) \), for a flow graph, \( G \), is defined as:

\[
V(G) = E - N + 2P
\]

- \( E \): number of edges
- \( N \): number of nodes
- \( P \): number of disconnected parts of the graph

2. Cyclomatic complexity \( V(G) \), for a flow graph, \( G \), with only binary decisions, is defined as:

\[
V(G) = b + 1
\]

- \( b \): number of binary decision
Examples of Graphs and calculation of McCabe's Complexity Metric

Student Worksheet

1. \( V(G) = E - N + 2P \)
   
   \[ E = ? \]
   \[ N = ? \]
   \[ P = ? \]
   \[ V(G) = ? \]

2. \( V(G) = b + 1 \)
   
   \[ b = ? \]
   \[ V(G) = ? \]
2. Data Flow Testing

Data flow testing focuses on the points at which variables receive values and the points at which these values are used (or referenced). It detects improper use of data values due to coding errors.

- Early data flow analyses often centered on a set of faults that are known as define/reference anomalies.
  - A variable that is defined but never used (referenced)
  - A variable that is used but never defined
  - A variable that is defined twice before it is used
Part I
Introduction, Testing Process

Part II
Unit Testing: Black-box, White-box

Mariam Kamkar
marka@ida.liu.se

Part III
Module Testing (Integration testing)

Part IV
System Testing

Definitions

du-path: a definition-use path (du-path) with respect to variable \( v \) is a path in \( PATHS(P) \) such that, for some \( v \) in \( V \), there are defined and usage nodes \( DEF(v, m) \) and \( USE(v, n) \) such that \( m \) and \( n \) are initial and final nodes of the path.
Data Flow Graphs

- ~d: the variable does not exist, then it is defined
- ~u: the variable does not exist, then it is used
- ~k: the variable does not exist, then it is killed

Student Worksheet

Variable x:

Control flow graph annotated with define-use-kill information for x, y, z

Hierarchy of data flow coverage metrics

- All-Paths
- All-DU-Paths
- All-Uses
  - All C-Uses/some P-Uses
  - All P-Uses/some C-Uses
- All-Defs
- All-P-Uses
  - All-P-Uses
  - All-Edges
  - All-Nodes

Branch
Statement
Part III
Module Testing
(Integration testing)

Integration Testing

1. Top-down
2. Bottom-up
3. Big-bang
4. Sandwich
Unit & Integration Testing

Objective: to ensure that code implemented the design properly.

Part I
Introduction, Testing Process

Part II
Unit Testing: Black-box, White-box

Part III
Module Testing (Integration testing)

Part IV
System Testing

Design Specification

Component code

Unit test

Tested components

Integration test

Integrated modules

Part I
Introduction, Testing Process

Part II
Unit Testing: Black-box, White-box

Part III
Module Testing (Integration testing)

Part IV
System Testing
Part I
Introduction, Testing Process

Part II
Unit Testing: Black-box, White-box

Part III
Module Testing (Integration testing)

Part IV
System Testing

Components

Component to be tested

driver

stub

stub

Boundary conditions
independent paths
interface ...

Test cases

Components

A

B

C

D

E

F

G

Part I
Introduction, Testing Process

Part II
Unit Testing: Black-box, White-box

Part III
Module Testing (Integration testing)

Part IV
System Testing
Part I
Introduction, Testing Process

Part II
Unit Testing: Black-box, White-box

Part III
Module Testing (Integration testing)

Part IV
System Testing

1. Top-down

2. Bottom-up

Student Worksheet
Part I
Introduction, Testing Process

Part II
Unit Testing: Black-box, White-box

Part III
Module Testing (Integration testing)

Part IV
System Testing

3. Big-bang

4. Sandwich

Student Worksheet
### Part IV
System Testing

<table>
<thead>
<tr>
<th>Part I</th>
<th>Introduction, Testing Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Part II</td>
<td>Unit Testing: Black-box, White-box</td>
</tr>
<tr>
<td>Part III</td>
<td>Module Testing (Integration testing)</td>
</tr>
<tr>
<td>Part IV</td>
<td>System Testing</td>
</tr>
</tbody>
</table>

- **System Testing Steps**
  - Function testing / Thread testing
  - Performance testing
  - Acceptance testing
  - Installation testing

- Termination Problem
System Testing

Objective: to ensure that the system does what the customer wants it to do.

Customer  Developer

Requirements definition
Requirements specification

Functional requirements
Nonfunctional requirements

Part I
Introduction, Testing Process

Part II
Unit Testing: Black-box, White-box

Part III
Module Testing (Integration testing)

Part IV
System Testing

Customer Developer

Requirements definition
Requirements specification

Functional requirements
Nonfunctional requirements

Part I
Introduction, Testing Process

Part II
Unit Testing: Black-box, White-box

Part III
Module Testing (Integration testing)

Part IV
System Testing

Unit test
Component code

Tested components

Design Specification

Integration test

Integrated modules

Part I
Introduction, Testing Process

Part II
Unit Testing: Black-box, White-box

Part III
Module Testing (Integration testing)

Part IV
System Testing

Component code

Unit test

Tested components

Integration test

Integrated modules

Part I
Introduction, Testing Process

Part II
Unit Testing: Black-box, White-box

Part III
Module Testing (Integration testing)

Part IV
System Testing
Part I
Introduction, Testing Process

Part II
Unit Testing: Black-box, White-box

Part III
Module Testing (Integration testing)

Part IV
System Testing

---

Function testing/Thread testing

A function test checks that the integrated system performs its function as specified in the requirement

- Guidelines
  - use a test team independent of the designers and programmers
  - know the expected actions and output
  - test both valid and invalid input
  - never modify the system just to make testing easier
  - have stopping criteria
Part I
Introduction, Testing Process

Part II
Unit Testing: Black-box, White-box

Part III
Module Testing (Integration testing)

Part IV
System Testing

---

**Causes-and-Effect-Graph**
(test case generation from req.)

- causes: inputs
- effects: outputs and transformations
- causes-and-effect graph:
  - boolean graph reflecting causes and effects relationships
  - is a formal language into which a natural language specification is translated

---

**Basic cause-effect graph symbols**

- Identity: if a then b

- And: if (a and b) then c

- Or: if (a or b or c) then d

- Identity: if (not a) then b
**Specification:** the character in column 1 must be an “A” or a “B”. The character in column 2 must be a digit. In this situation, the file update is made. If the first character is incorrect, message X12 is issued. If the second character is not a digit, message X13 is issued.

**Causes**
- C1: character in column 1 is “A”
- C2: character in column 1 is “B”
- C3: character in column 2 is a digit

**Effects**
- E1: update made
- E2: message X12 is issued
- E3: message X13 is issued
### Decision table for cause-and-effect graph

<table>
<thead>
<tr>
<th></th>
<th>Test 1</th>
<th>Test 2</th>
<th>Test 3</th>
<th>Test 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cause 1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>Cause 2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>Cause 3</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>Effect E1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Effect E2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Effect E3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

### Performance Testing

**nonfunctional requirements**

- Stress tests
- Volume tests
- Configuration tests
- Compatibility tests
- Regression tests
- Security tests
- Timing tests
- Environment tests
- Quality tests
- Recovery tests
- Maintenance tests
- Documentation tests
- Human factors tests / usability tests
Acceptance Testing

customers, users need

- Benchmark test: a set of special test cases
- Pilot test: everyday working
  - Alpha test: at the developer's site, controlled environment
  - Beta test: at one or more customer site.
- Parallel test: new system in parallel with previous one

Installation Testing

users site

Acceptance test at developers site
→ installation test at users site,
otherwise may not be needed!!
Termination Problem
How decide when to stop testing

- The main problem for managers!
- Termination takes place when
  - resources (time & budget) are over
  - found the seeded faults
  - some coverage is reached

Part I: Introduction, Testing process
Part II: Unit Testing:
  - Black-box Testing
    1. Exhaustive testing
    2. Equivalence class testing (Equivalence Partitioning)
    3. Boundary value analysis
    4. Decision table testing
  - White-box Testing
    - Control Flow Testing
      1. Statement/Line/Basic block/Segment Coverage
      2. Decision (Branch) Coverage
      3. Condition Coverage
      4. Decision/Condition Coverage
      5. Path Coverage
    - Data Flow Testing

Summary - What have we learned today? (1/2)
Summary - What have we learned today? (2/2)

Part III: Module Testing (Integration Testing)
- Top-down
- Bottom-up
- Big-bang
- Sandwich

Part IV: System Testing

Do you want to know more about testing?
Take the course on Software Testing (TDDC04).