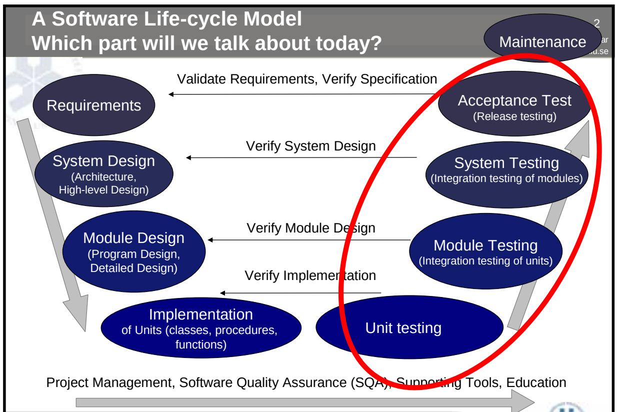
Software Testing

Software Engineering
January 2008

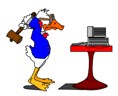
Mariam Kamkar
Department of Computer and Information Science
Linköping University, Sweden
marka@ida.liu.se

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

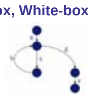


Agenda - What will you learn today?

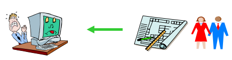
Part I
Introduction, Testing Process



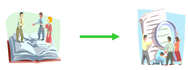
Part II
Unit Testing:
Black-box, White-box testing



Part III
Module Testing
(Integration testing)



Part IV
System Testing



Part I
Introduction, Testing Process

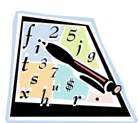
Triangle program (simple version)

- 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.

Test case	*a*	*b*	*c*	*Expected output*
1	3	3	4	isosceles (likbent)
2	?	?	?	?
…				

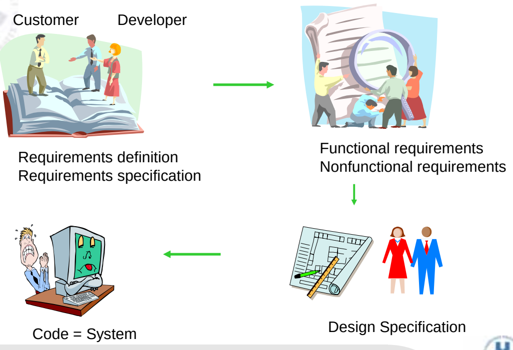
Testing a ballpoint pen

- 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?

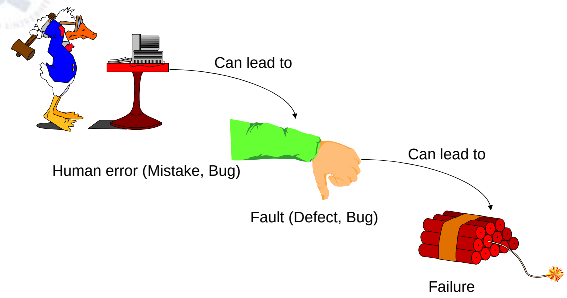


What is expected from this pen?

Intended use!!



Error, Fault, Failure

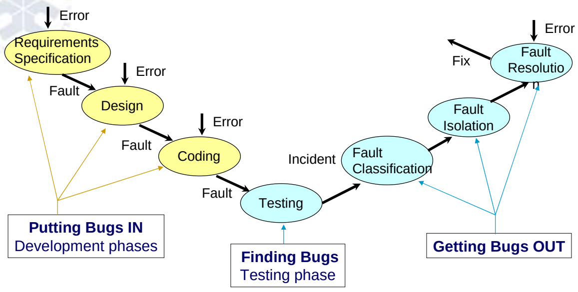


Types of Faults

(dep. on org. IBM, HP)

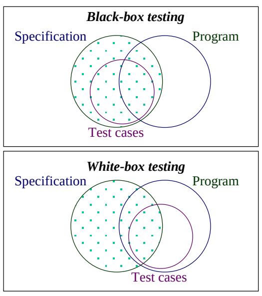
- Algorithmic: division by zero
- Computation & Precision: order of op
- Documentation: doc - code
- Stress/Overload: data-str size (dimensions of tables, size of buffers)
- Capacity/Boundary: x devices, y parallel tasks, z interrupts
- Timing/Coordination: real-time systems
- Throughput/Performance: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

A Testing Life Cycle



Types (strategy) of testing

- **Black-box**: a strategy in which testing is based on requirements and specifications.
- **White-box**: a strategy in which testing is based on internal paths, structure, and implementation.
- **Gray-box**: peek into the "box" just long enough to understand how it has been implemented.



Contents of a Test Case

"Boilerplate": author, date, purpose, test case ID

Pre-conditions (including environment)

Inputs

Expected Outputs

Observed Outputs

Pass/Fail

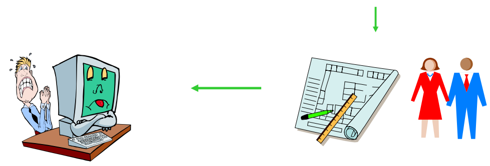
Part II

Unit Testing:

Black-box, White-box testing

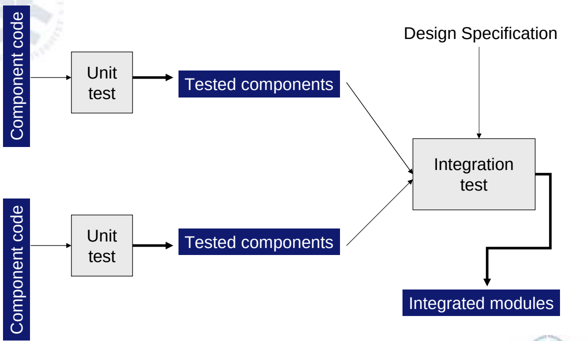
Unit & Integration Testing

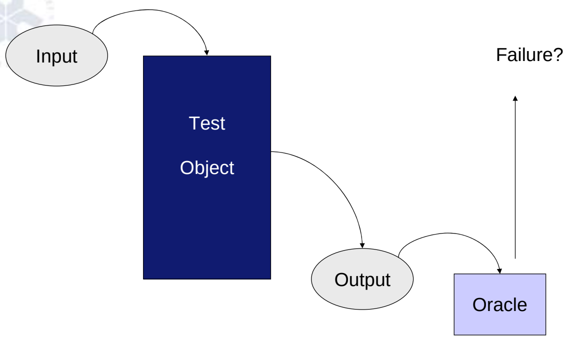
Objective: to ensure that code implemented the design properly.



Code = System

Design Specification





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.

Unit Testing

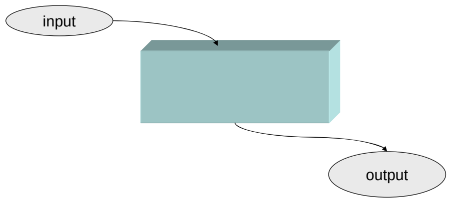
- Black-box Testing
- White-box Testing

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



Block-Box Testing Techniques

- **Definition**: a strategy in which testing is based on requirements and specifications.
- **Applicability**: all levels of system development
  - Unit
  - Integration
  - System
  - Acceptance
- **Disadvantages**: never be sure of how much of the system under test (SUT) has been tested.
- **Advantages**: directs tester to choose subsets to tests that are both efficient and effective in finding defects.

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

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
  - ? < x < ?
- Invalid equivalence classes
  - x < ?
  - x > ?

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.

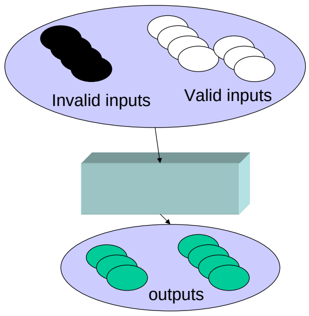
Identifying the Test Cases

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.

Applicability and Limitations

- Most suited to systems in which much of the input data takes on values within ranges or within sets.
- It makes the assumption that data in the same EC is, in fact, processed in the same way by the system. The simplest way to validate this assumption is to ask the programmer about their implementation.
- EC testing is equally applicable at the unit, integration, system, and acceptance test levels. All it requires are inputs or outputs that can be partitioned based on the system's requirements.

Equivalence partitioning



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).

Technique

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.

Boundary value analysis

Less than 10000	Between 10000 and 99999	More than 99999

Applicability and Limitations

Boundary value testing is equally applicable at the unit, integration, system, and acceptance test levels. All it requires are inputs that can be partitioned and boundaries that can be identified based on the system's requirements.

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.

Technique

The general format of a decision table:

	Rule 1	Rule 2	…	Rule P
Conditions				
Condition-1				
Condition-2				
…				
Condition-m				
Actions				
Action-1				
Action-2				
…				
Action-n				

A decision table with "don't care" entry

	Rule 1	Rule 2	Rules 3,4	Rule 5	Rule 6	Rules 7,8
C1	T	T	T	F	F	F
C2	T	T	F	T	T	F
C3	T	F	_	T	F	_
A1	X	X		X		
A2	X				X	
A3		X		X		
A4			X			X

- **_ : "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..

Technique (cont.)

A decision table converted to a test case table:

	Test Case 1	Test Case 2	…	Test Case P
Inputs				
Condition-1				
Condition-2				
…				
Condition-m				
Expected Results				
Action-1				
Action-2				
…				
Action-n				

Triangle program (new conditions)

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 integers *a*, *b*, and *c* must satisfy the following conditions:

C1: $1 <= a <= 200$
C2: $1 <= b <= 200$
C3: $1 <= c <= 200$
C4: $a < b + c$
C5: $b < a + c$
C6: $c < a + b$

Test Cases for the Triangle Problem

	1	2	3	4	5	6	7	8	9	10	11
C1: a < b + c ?											
C2: b < a + c ?											
C3: c < a + b ?											
C4: a = b?											
C5: a = c?											
C6: b = c?											
A1: Not a triangle											
A2: Scalene											
A3: Isosceles											
A4: Equilateral											
A5: **Impossible**											

Case ID	a	b	c	Expected output
DT1				
DT2				
DT3				
DT4				
DT5				
DT6				
DT7				
DT8				
DT9				
DT 10				
DT 11				

Applicability and Limitations

Decision table testing can be used whenever the system must implement complex business rules when these rules can be represented as a combination of conditions and when these conditions have discrete actions associated with them.

White-box Testing

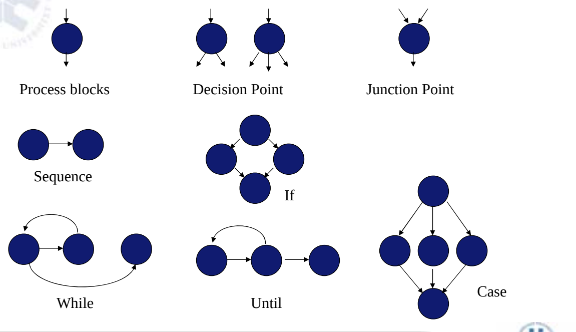
(Glass box testing, Open box testing, Clear box testing, Structural testing)

1. **Control flow testing**
2. **Data flow testing**

1. Control flow testing

- **Definition:** a strategy in which testing is based on the internal paths, structure, and implementation of the software under test (SUT)
- **Applicability:** all levels of system development (path testing!)
  - Unit
  - Integration
  - System
  - Acceptance
- **Disadvantages:** 1) number of execution paths may be so large; 2) test cases may not detect data sensitivity; 3) assumes that control flow is correct (nonexistent paths!); 4) tester must have programming skills.
- **Advantages:** tester can be sure that every path have been identified and tested.

Control Flow Graphs



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).

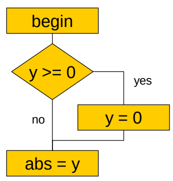
Code Coverage (test coverage metrics)

Levels of Coverage:

- Statement/Line/Basic block/Segment Coverage
- Decision (Branch) Coverage
- Condition Coverage
- Decision/Condition Coverage
- Path Coverage

Statement Coverage


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



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

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.

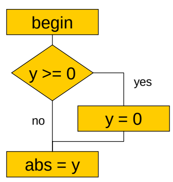
Of all the structural coverage criteria, line coverage is the weakest, indicating the fewest number of test cases.

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.

Branch Coverage


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



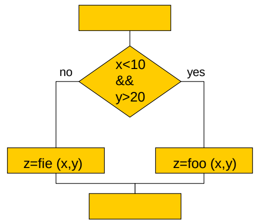
test case-1 **(yes)**:
input: y = 0
expected result: 0
actual result: 0

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

... more conditions?

Condition Coverage

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

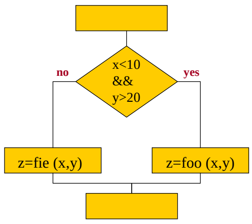


test case-1 **(T,F):**
input: x = ?, y = ?
expected result: ?
actual result: ?

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

Decision/Condition Coverage

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



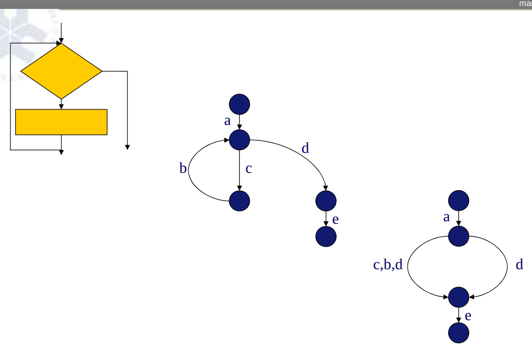
test case-1 **(T,T,yes):**
input: x = ?, y = ?
expected result: ?
actual result: ?

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



Path Coverage (cont.)

- All possible execution paths
- Question: How do we know how many paths to look for?
- Answer: The computation of cyclomatic complexity

Computation of cyclomatic complexity

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

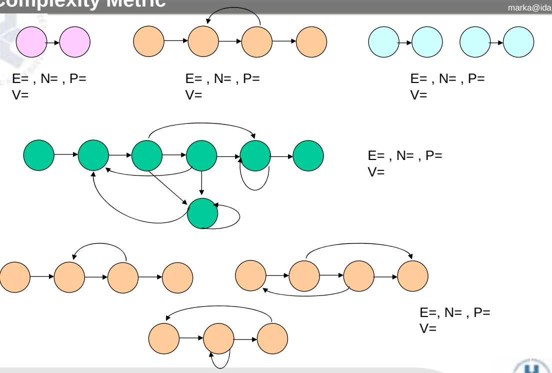
1. Cyclomatic complexity V(G), for a flow graph, G, is defined as:

   V(G) = E – N + 2P
   E: number of edges
   N: number of nodes
   P: number of disconnected parts of the graph

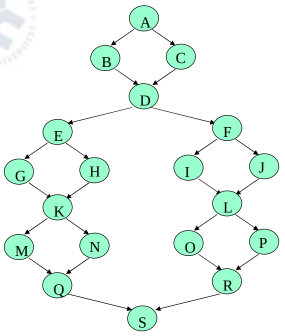
2. Cyclomatic complexity V(G), for a flow graph, G, with only binary decisions, is defined as:

   V(G) = b + 1
   b: number of binary decision

Examples of Graphs and calculation of McCabe's Complexity Metric



Part I Introduction, Testing Process | Part II Unit Testing: Black-box, White-box | Part III Module Testing (Integration testing) | Part IV System Testing | Linköpings universitet



1. V(G) = E – N + 2P

E = ?
N = ?
P =

V(G) = ?

2. V(G) = b + 1

b = ?

V(G) = ?

Part I Introduction, Testing Process | Part II Unit Testing: Black-box, White-box | Part III Module Testing (Integration testing) | Part IV System Testing | Linköpings universitet

Applicability and Limitation

- Control flow testing is the <u>cornerstone</u> of unit testing. It should be used for <u>all modules of code</u> that cannot be tested sufficiently through reviews and inspections.
- Its limitation are that the tester must have sufficient <u>programming skill</u> to understand the code and its control flow.
- Control flow testing can be very <u>time consuming</u> because of all modules and basic paths that comprise a system.

Part I Introduction, Testing Process | Part II Unit Testing: Black-box, White-box | Part III Module Testing (Integration testing) | Part IV System Testing | Linköpings universitet

2. Data Flow Testing

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

Part I Introduction, Testing Process | Part II Unit Testing: Black-box, White-box | Part III Module Testing (Integration testing) | Part IV System Testing | Linköpings universitet

Define/Reference Anomalies

- 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

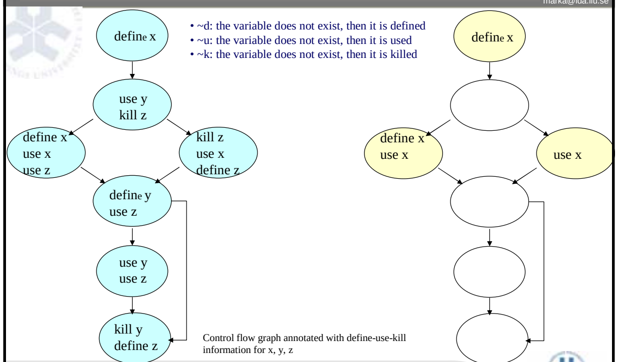
Part I Introduction, Testing Process | Part II Unit Testing: Black-box, White-box | Part III Module Testing (Integration testing) | Part IV System Testing | Linköpings universitet

- **dd**: defined and defined again – not invalid but suspicious
- **du**: defined and used – perfectly correct
- **dk**: defined and then killed – not invalid but probably a programming error
- **ud**: used and defined – acceptable
- **uu**: used and used again – acceptable
- **uk**: used and killed – acceptable
- **kd**: killed and defined – acceptable
- **ku**: killed and used – a serious defect
- **kk**: killed and killed – probably a programming error.

Definitions

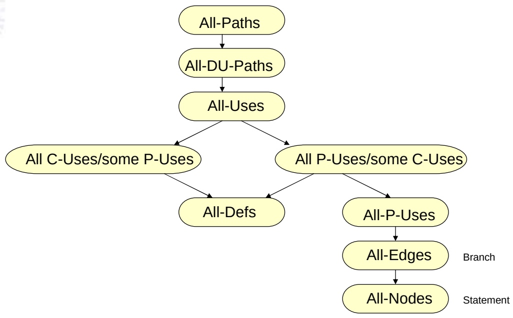
du-path: a definition-use path (du-path) with respect to variable *v* is a path in *PATHS(P)* such that, for some *v* in *V*, there are defined and usage nodes *DEF(v, m)* and *USE(v, n)* such that *m* and *n* are initial and final nodes of the path.

Data Flow Graphs



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

Hierarchy of data flow coverage metrics



Applicability and Limitations
(data-flow testing)

- It should be used for all modules of code that cannot be tested sufficiently through reviews and inspections.
- Tester must have sufficient programming skill
- Can be very time consuming

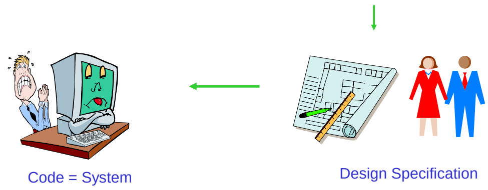
Part III
Module Testing
(Integration testing)

Integration Testing

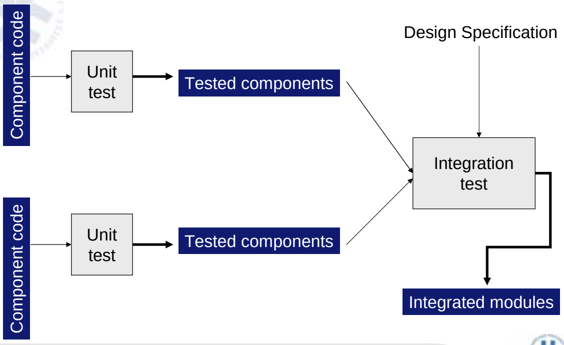
1. Top-down
2. Bottom-up
3. Big-bang
4. Sandwich

Unit & Integration Testing

Objective: to ensure that code implemented the design properly.

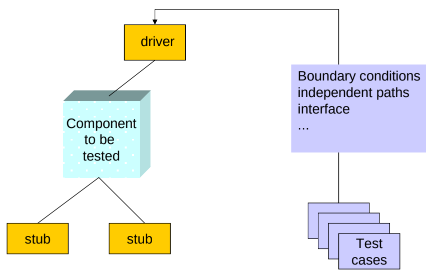


Part I Introduction, Testing Process
Part II Unit Testing: Black-box, White-box
Part III Module Testing (Integration testing)
Part IV System Testing
Linköpings universitet

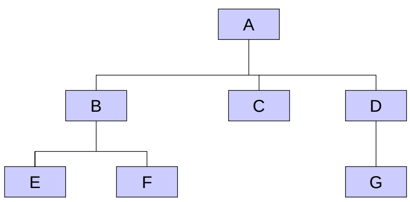


Part I Introduction, Testing Process
Part II Unit Testing: Black-box, White-box
Part III Module Testing (Integration testing)
Part IV System Testing
Linköpings universitet

Components

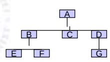


Part I Introduction, Testing Process
Part II Unit Testing: Black-box, White-box
Part III Module Testing (Integration testing)
Part IV System Testing
Linköpings universitet



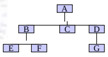
Part I Introduction, Testing Process
Part II Unit Testing: Black-box, White-box
Part III Module Testing (Integration testing)
Part IV System Testing
Linköpings universitet

1. Top-down

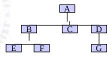


Part I Introduction, Testing Process
Part II Unit Testing: Black-box, White-box
Part III Module Testing (Integration testing)
Part IV System Testing
Linköpings universitet

2. Bottom-up

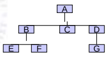


3. Big-bang



Part I Introduction, Testing Process | Part II Unit Testing: Black-box, White-box | Part III Module Testing (Integration testing) | Part IV System Testing | Linköpings universitet

4. Sandwich



Target level B,C,D

Part I Introduction, Testing Process | Part II Unit Testing: Black-box, White-box | Part III Module Testing (Integration testing) | Part IV System Testing | Linköpings universitet

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 | Linköpings universitet

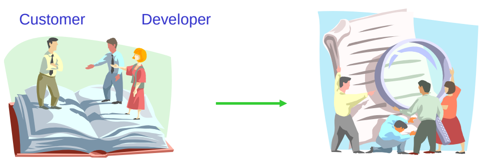
- System Testing Steps
  - Function testing / Thread testing
  - Performance testing
  - Acceptance testing
  - Installation testing
- Test Automation
- Termination Problem

Part I Introduction, Testing Process | Part II Unit Testing: Black-box, White-box | Part III Module Testing (Integration testing) | Part IV System Testing | Linköpings universitet

System Testing

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

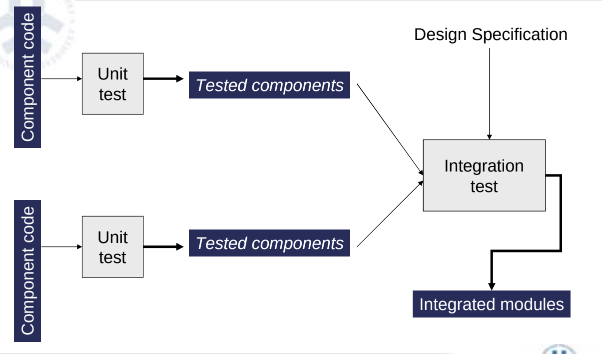
Customer Developer

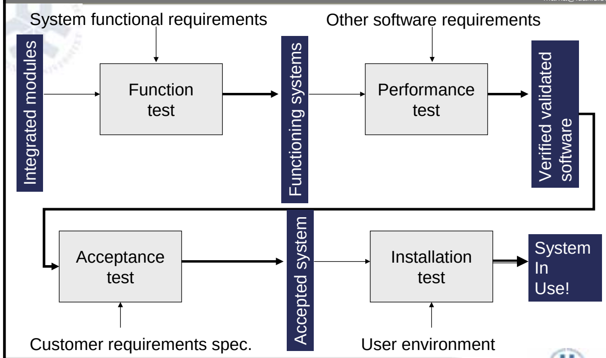


Requirements definition
Requirements specification

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 | Linköpings universitet





Function testing/Thread testing

(testing one function at a time)

functional requirements

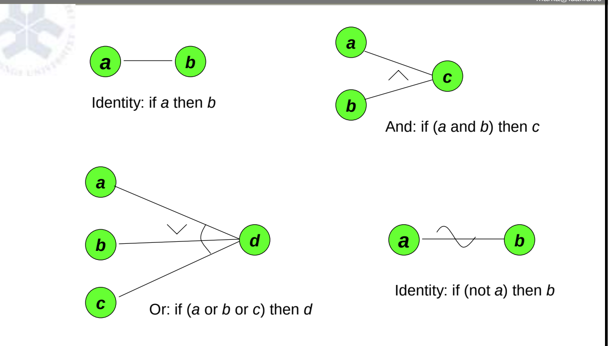
A function test checks that the integrated system performs its function as specified in the requirement

- 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

Cause-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



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 a digit

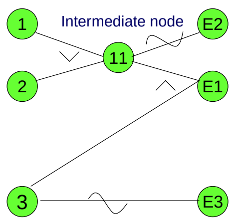
Effects

E1: update made

E2: message X12 is issued

E3:message X13 is issued

Sample cause-effect graph



Decision table for cause-and effect graph

	Test 1	Test 2	Test 3	Test 4
Cause 1	1	0	0	X
Cause 2	0	1	0	X
Cause 3	1	1	X	0
Effect E1	1	1	0	0
Effect E2	0	0	1	0
Effect E3	0	0	0	1

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!!

Test Automation

- Automating parts of the testing process can provide long-term benefits to organization, such as:
  - reducing the amount of time it takes to execute a suite of tests
  - reducing the tester's involvement in executing tests
  - facilitating regression testing
  - allowing for the simulation of hundreds of users
  - avoiding human mistakes by having tools control repetitive and tedious tasks
- Test automation refers to two key testing activities:
  - Executing the tests
  - Evaluating the output

Automated Testing Tools

- Code Analysis tools
  - Static, Dynamic
- Test execution tools
  - Capture-and-Replay
  - Stubs & Drivers
  - Comparators
- Test case generator

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

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

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? (2/2)

Part III: Module Testing (Integration Testing)

- **Top-down**
- **Bottom-up**
- **Big-bang**
- **Sandwich**

Part IV: System Testing