



Software Testing



Background

- Main objectives of a project: High Quality & High Productivity (Q&P)
- Quality has many dimensions
 - reliability, maintainability, interoperability etc.
- Reliability is perhaps the most important
- Reliability: The chances of software failing
- More defects => more chances of failure => lesser reliability
- Hence Q goal: Have as few defects as possible in the delivered software



Faults & Failure

- Failure: A software failure occurs if the behavior of the s/w is different from expected/specified.
- Fault: cause of software failure
- Fault = bug = defect
- Failure implies presence of defects
- A defect has the potential to cause failure.
- Definition of a defect is environment, project specific



Role of Testing

- Reviews are human processes - can not catch all defects
- Hence there will be requirement defects, design defects and coding defects in code
- These defects have to be identified by testing
- Therefore testing plays a critical role in ensuring quality.
- All defects remaining from before as well as new ones introduced have to be identified by testing.



Detecting defects in Testing

- During testing, a program is executed with a set of test cases
- Failure during testing => defects are present
- No failure => confidence grows, but can not say “defects are absent”
- Defects detected through failures
- To detect defects, must cause failures during testing



Test Oracle

- To check if a failure has occurred when executed with a test case, we need to know the correct behavior
- I.e. need a test oracle, which is often a human
- Human oracle makes each test case expensive as someone has to check the correctness of its output



Role of Test cases

- Ideally would like the following for test cases
 - No failure implies “no defects” or “high quality”
 - If defects present, then some test case causes a failure
- Psychology of testing is important
 - should be to ‘reveal’ defects(not to show that it works!)
 - test cases should be “destructive
- Role of test cases is clearly very critical
- Only if test cases are “good”, the confidence increases after testing



Test case design

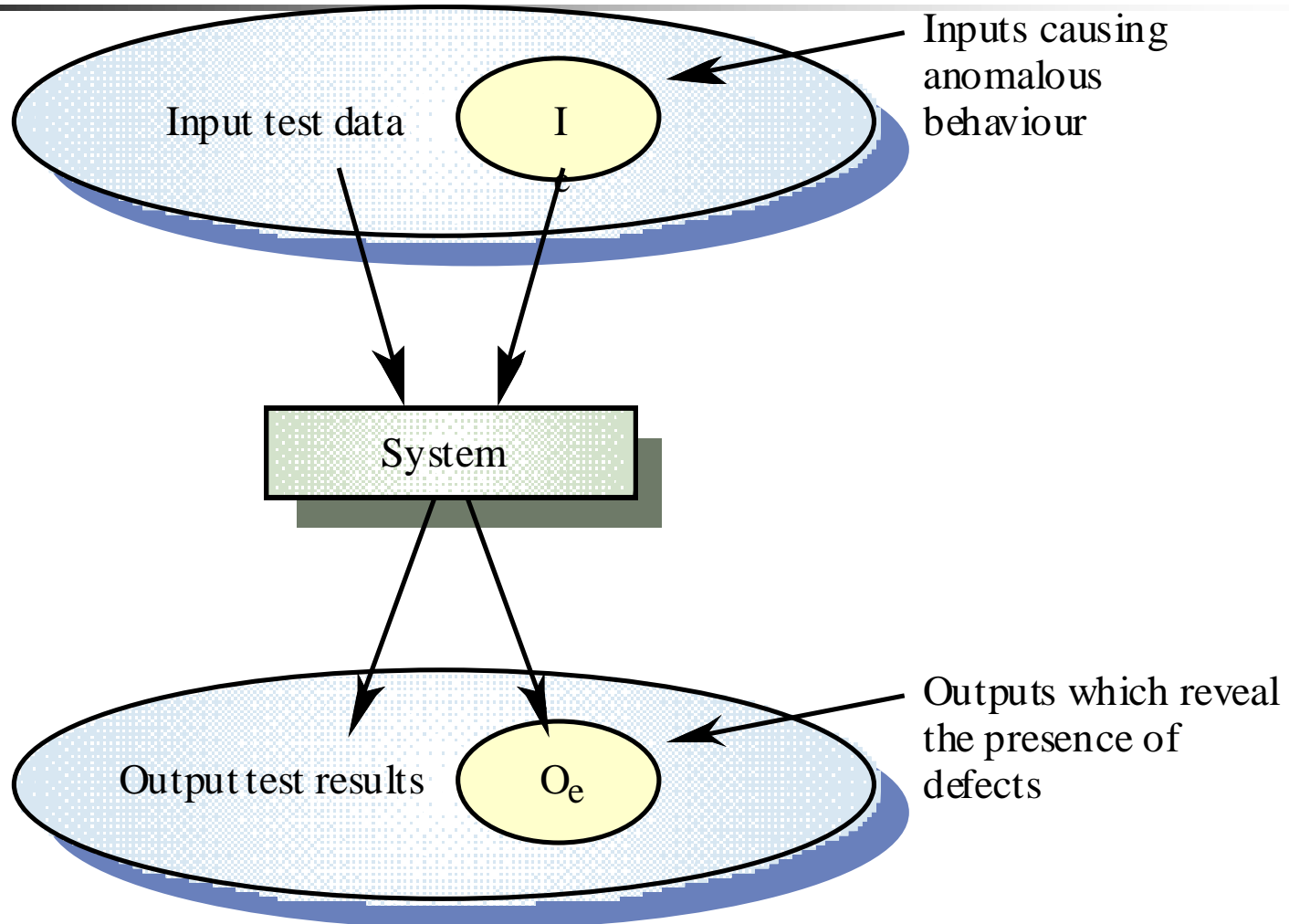
- During test planning, have to design a set of test cases that will detect defects present
- Some criteria needed to guide test case selection
- Two approaches to design test cases
 - functional or black box
 - structural or white box
- Both are complimentary; we discuss a few approaches/criteria for both



Black Box testing

- Software tested to be treated as a block box
- Specification for the black box is given
- The expected behavior of the system is used to design test cases
- i.e test cases are determined solely from specification.
- Internal structure of code not used for test case design.
- Expected behavior is specified.
- Hence just test for specified expected behavior
- How it is implemented is not an issue.

Black-box testing





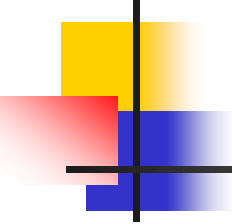
Black box testing

- Also called *behavioral testing*, focuses on the functional requirements of the software.
- It enables the software engineer to derive sets of input conditions that will fully exercise all functional requirements for a program.
- Black-box testing is not an alternative to white-box techniques but it is complementary approach.
- Black-box testing attempts to find errors in the following categories:
 - Incorrect or missing functions,
 - Interface errors,
 - Behavior or performance errors,
 - Initialization and termination errors.



Equivalence Class partitioning

- Divide the input space into equivalent classes
- If the software works for a test case from a class then it is likely to work for all
- Can reduce the set of test cases if such equivalent classes can be identified
- Getting ideal equivalent classes is almost impossible.



To define equivalence classes follow the guideline

1. If an input condition specifies a *range*, one valid and two invalid equivalence classes are defined.
2. If an input condition requires a specific *value*, one valid and two invalid equivalence classes are defined.
3. If an input condition specifies a member of a *set*, one valid and one invalid equivalence class are defined.
4. If an input condition is *Boolean*, one valid and one invalid class are defined.



Equivalent class partitioning..

- Every condition specified as input is an equivalent class
- Define invalid equivalent classes also
- E.g. range $0 < \text{value} < \text{Max}$ specified
 - one range is the valid class
 - $\text{input} < 0$ is an invalid class
 - $\text{input} > \text{max}$ is an invalid class
- Whenever that entire range may not be treated uniformly - split into classes



Search routine specification

procedure Search (Key : INTEGER ; T: array 1..N of INTEGER;
Found : BOOLEAN; L: 1..N) ;

Pre-condition

-- the array has at least one element
 $1 \leq N$

Post-condition

-- the element is found and is referenced by L
(Found and $T(L) = \text{Key}$)
or
-- the element is not in the array
(**not** Found **and**
not (**exists** $i, 1 \leq i \leq N, T(i) = \text{Key}$))



Search routine input partitions

- Inputs which conform to the pre-conditions.
- Inputs where a pre-condition does not hold.
- Inputs where the key element is a member of the array.
- Inputs where the key element is not a member of the array.



Search routine input partitions

Array	Element
Single value	In array
Single value	Not in array
More than 1 value	First element in array
More than 1 value	Last element in array
More than 1 value	Middle element in array
More than 1 value	Not in array



Search routine test cases

Input array (T)	Key (Key)	Output (Found, L)
17	17	true, 1
17	0	false, ??
17, 29, 21, 23	17	true, 1
41, 18, 9, 31, 30, 16, 45	45	true, 6
17, 18, 21, 23, 29, 41, 38	23	true, 4
21, 23, 29, 33, 38	25	false, ??



Example

- Consider a program that takes 2 inputs
 - a string *s* and an integer *n*
- Program determines *n* most frequent characters
- Tester believes that programmer may deal with diff types of chars separately
- A set of valid and invalid equivalence classes is given



Example..

Input	Valid Eq Class	Invalid Eq class
S	1: Contains numbers 2: Lower case letters 3: upper case letters 4: special chars 5: str len between 0-N(max)	1: non-ascii char 2: str len > N
N	6: Int in valid range	3: Int out of range



Boundary value analysis

- Boundary value analysis is a test case design technique that complements equivalence partitioning.
- Rather than selecting any element of an equivalence class, BVA leads to the selection of test cases at the "edges" of the class.
- In other word, Rather than focusing solely on input conditions, BVA derives test cases from the output domain as well.



Guidelines for BVA

1. If an input condition specifies a range bounded by values a and b , test cases should be designed with values a and b and just above and just below a and b .
2. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.
3. Apply guidelines 1 and 2 to output conditions.
4. If internal program data structures have prescribed boundaries be certain to design a test case to exercise the data structure at its boundary



Boundary Value Analysis

- Some typical programming errors occur:
 - at boundaries of equivalence classes
 - might be purely due to psychological factors.
- Programmers often fail to see:
 - special processing required at the boundaries of equivalence classes.
- Programmers may improperly use `<` instead of `<=`
- Boundary value analysis:
 - select test cases at the boundaries of different equivalence classes.




Example

- For a function that computes the square root of an integer in the range of 1 and 5000:
 - test cases must include the values: $\{0, 1, 5000, 5001\}$.




White Box Testing

- 
- Called glass-box testing also.
 - Uses the control structure of the procedural design to derive test cases.
 - Using white-box testing we can derive test cases that
 - guarantee that all independent paths within a module have been exercised at least once,
 - exercise all logical decisions on their true and false sides,
 - execute all loops at their boundaries and within their operational bounds,
 - and exercise internal data structures to ensure their validity.



Statement coverage

- Programs constructs can be usually classified as:
 1. Sequential control flow
 2. Two way decision statements (if then else)
 3. Multi way decision statements (switch)
 4. Loops (while, do, repeat until, for)



Statement coverage refers to writing test cases that execute each of the program statements. One can start with the assumption that more the code covered, the better is the testing of the functionality, as the code realizes the functionality. Based on this assumption, code coverage can be achieved by providing coverage to each of the above types of statements.

$$\text{Statement Coverage} = \left(\frac{\text{Total statements exercised}}{\text{Total number of executable statements in program}} \right) * 100$$



- A good percentage of defects comes because of loops that do not function properly.

- Most often loops fail in what are called as boundary conditions.

- One of the most common looping error is that termination condition of loop is not properly stated.

- For better statement coverage for statements within loop there should be test cases that:

1. Skip the loop completely.

2. Exercise the loop between once and maximum no. of times.

3. Try covering loop around the boundary.

Exhaustive coverage of all statements in a program will be practically impossible.

Even if we were to achieve a very high level of statement coverage, it does not mean that the program is defect-free. First, consider a hypothetical case when we achieved 100 percent code coverage. If the program implements wrong requirements and this wrongly implemented code is “fully tested,” with 100 percent code coverage, it still is a wrong program and hence the 100 percent code coverage does not mean anything.

Next, consider the following program.

```
Total = 0; /* set total to zero */
if (code == "M") {
    stmt1;
    stmt2;
    Stmt3;
    stmt4;
    Stmt5;
    stmt6;
    Stmt7;
}


else percent = value/Total*100; /* divide by zero */
```



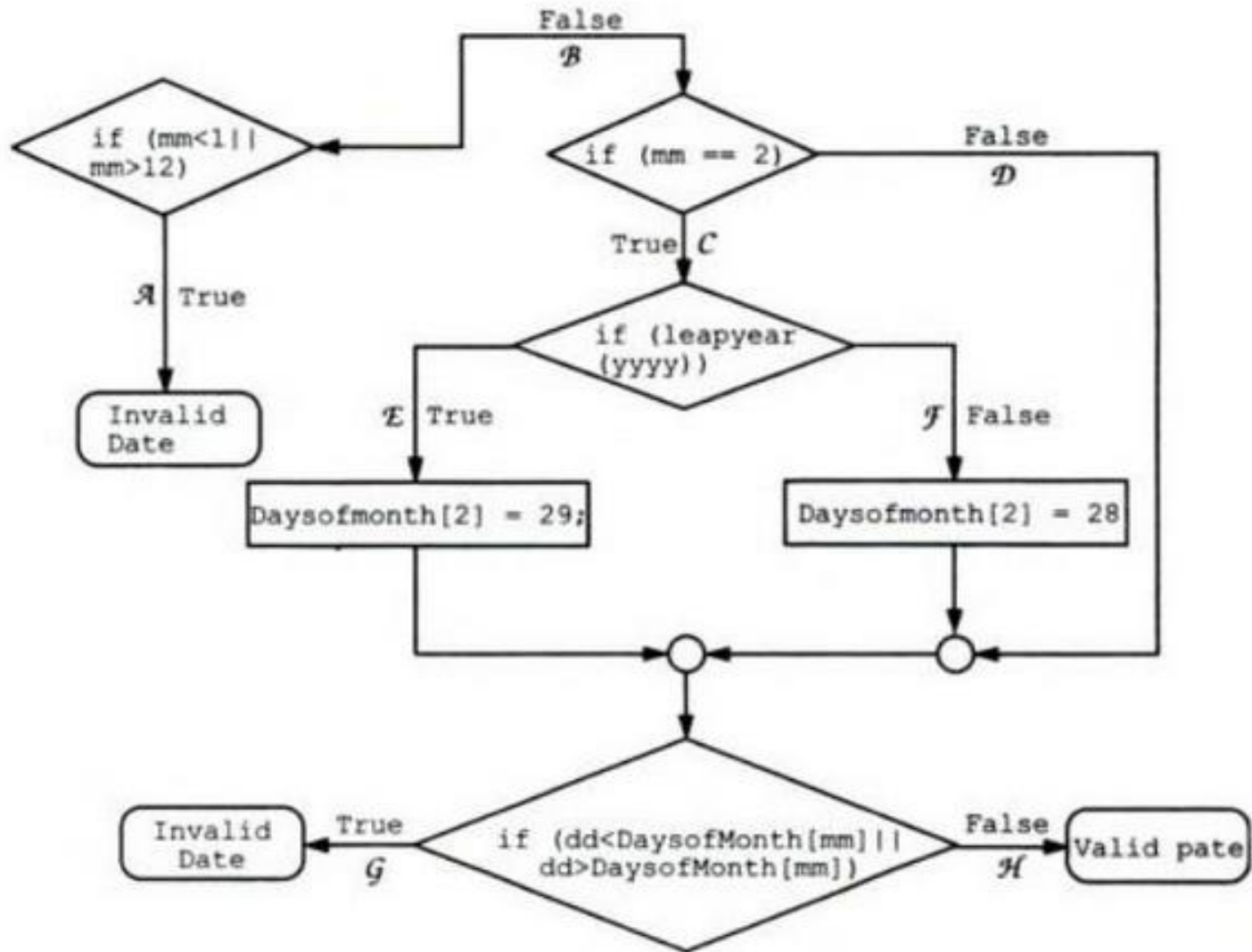
Path Coverage


In path coverage, we split a program into a number of distinct paths. A program (or a part of a program) can start from the beginning and take any of the paths to its completion.

$$\text{Path Coverage} = \frac{\text{Total paths exercised}}{\text{Total number of paths in program}} * 100$$



Let us take an example of a date validation routine. The date is accepted as three fields `mm`, `dd` and `yyyy`. We have assumed that prior to entering this routine, the values are checked to be numeric. To simplify the discussion, we have assumed the existence of a function called `leapyear` which will return `TRUE` if the given year is a leap year. There is an array called `DayofMonth` which contains the number of days in each month. A simplified flow chart





Regardless of the number of statements in each of these paths, if we can execute these paths, then we would have covered most of the typical scenarios.


Path coverage provides a stronger condition of coverage than statement coverage as it relates to the various logical paths in the program rather than just program statements.



Condition Coverage

In the above example, even if we have covered all the paths possible, it would not mean that the program is fully tested. For example, we can make the program take the path A by giving a value less than 1 (for example, 0) to `mm` and find that we have covered the path A and the program has detected that the month is invalid. But, the program may still not be correctly testing for the other condition namely `mm > 12`. Furthermore,

most compilers perform optimizations to minimize the number of Boolean operations and all the conditions may not get evaluated, even though the right path is chosen. For example, when there is an OR condition (as in the first IF statement above), once the first part of the IF (for example, `mm < 1`) is found to be true, the second part will not be evaluated at all as the overall value of the Boolean is TRUE. Similarly, when there is an AND condition in a Boolean expression, when the first condition evaluates to FALSE, the rest of the expression need not be evaluated at all.



For all these reasons, path testing may not be sufficient. It is necessary to have test cases that exercise each Boolean expression and have test cases test produce the TRUE as well as FALSE paths. Obviously, this will mean more test cases and the number of test cases will rise exponentially with the number of conditions and Boolean expressions. However, in reality, the situation may not be very bad as these conditions usually have some dependencies on one another.

$$\begin{aligned} &\text{Condition Coverage} \\ &= (\text{Total decisions} \\ &\quad \text{exercised} / \text{Total} \\ &\quad \text{number of decisions} \\ &\quad \text{in program}) * 100 \end{aligned}$$



Testing

- Testing only reveals the presence of defects
- Does not identify nature and location of defects
- Identifying & removing the defect => role of debugging and rework
- Preparing test cases, performing testing, defects identification & removal all consume effort
- Overall testing becomes very expensive : 30-50% development cost



Incremental Testing

- Goals of testing: detect as many defects as possible, and keep the cost low
- Both frequently conflict - increasing testing can catch more defects, but cost also goes up
- Incremental testing - add untested parts incrementally to tested portion
- For achieving goals, incremental testing essential
 - helps catch more defects
 - helps in identification and removal
- Testing of large systems is always incremental



Integration and Testing

- Incremental testing requires incremental 'building' I.e. incrementally integrate parts to form system
- Integration & testing are related
- During coding, different modules are coded separately
- Integration - the order in which they should be tested and combined
- Integration is driven mostly by testing needs



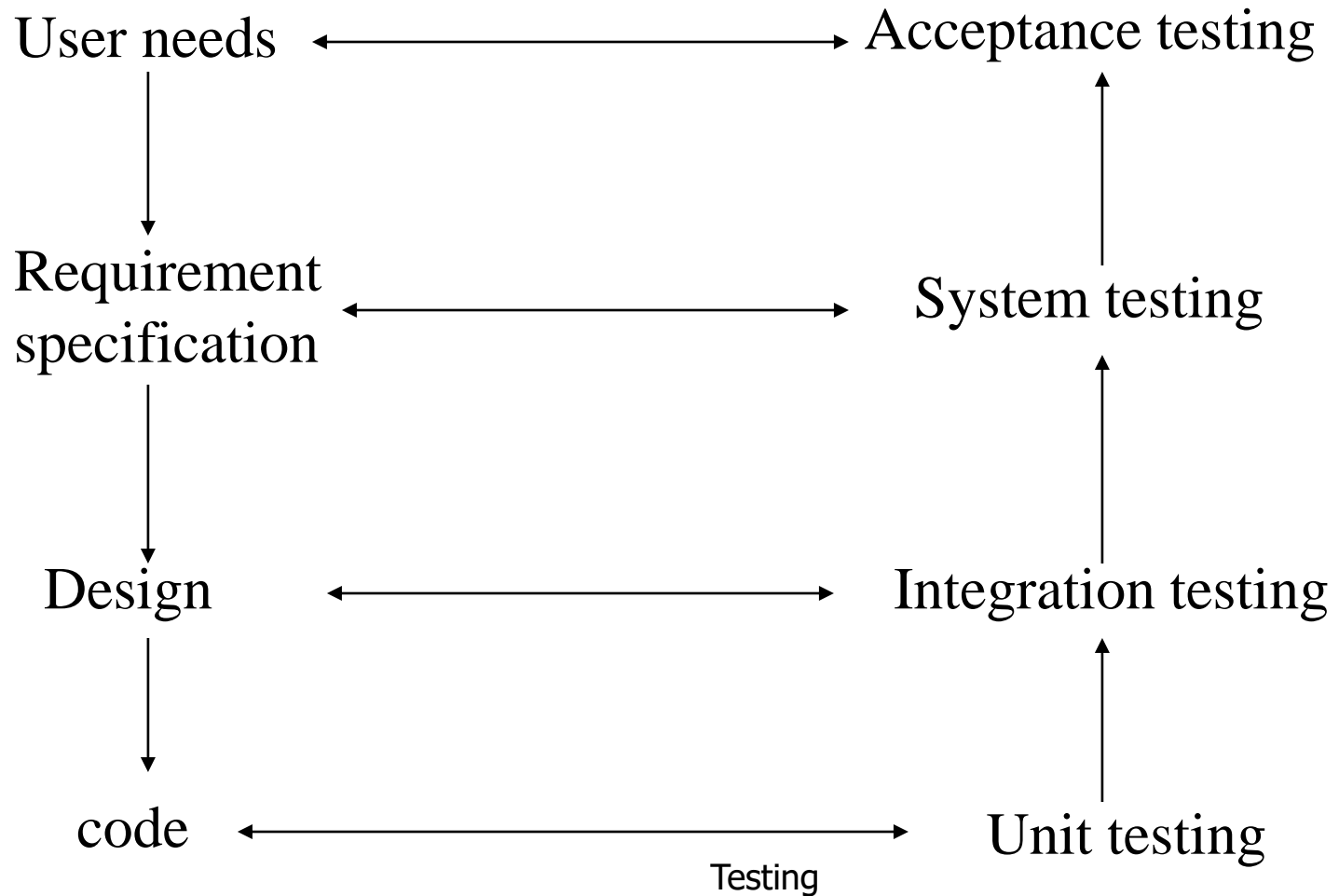
Top-down and Bottom-up

- System : Hierarchy of modules
- Modules coded separately
- Integration can start from bottom or top
- Bottom-up requires test drivers
- Top-down requires stubs
- Both may be used, e.g. for user interfaces top-down; for services bottom-up
- Drivers and stubs are code pieces written only for testing



Levels of Testing

- The code contains requirement defects, design defects, and coding defects
- Nature of defects is different for different injection stages
- One type of testing will be unable to detect the different types of defects
- Different levels of testing are used to uncover these defects





Unit Testing

- Different modules tested separately
- Focus: defects injected during coding
- Essentially a code verification technique, covered in previous chapter
- UT is closely associated with coding
- Frequently the programmer does UT; coding phase sometimes called “coding and unit testing”



Integration Testing

- Focuses on interaction of modules in a subsystem
- Unit tested modules combined to form subsystems
- Test cases to “exercise” the interaction of modules in different ways
- May be skipped if the system is not too large



System Testing

- Entire software system is tested
- Focus: does the software implement the requirements?
- Validation exercise for the system with respect to the requirements
- Generally the final testing stage before the software is delivered
- May be done by independent people
- Defects removed by developers
- Most time consuming test phase



Acceptance Testing

- Focus: Does the software satisfy user needs?
- Generally done by end users/customer in customer environment, with real data
- Only after successful AT software is deployed
- Any defects found, are removed by developers
- Acceptance test plan is based on the acceptance test criteria in the SRS



Other forms of testing

- Performance testing
 - tools needed to “measure” performance
- Stress testing
 - load the system to peak, load generation tools needed
- Regression testing
 - test that previous functionality works alright
 - important when changes are made
 - Previous test records are needed for comparisons
 - Prioritization of testcases needed when complete test suite cannot be executed for a change