SE Lec 16:

"Testing Phase"

- Software Fault: A static defect in the software.
- Software Failure: external, incorrect behaviour with respect to the requirements
- Software error: an incorrect internal state that is the manifestation of the fault.

cause = error
effect = fault / failure.

"Testing approaches"

- Testing
- Verification
- Inspection
- Formal proof of correctness.

→ we are only testing on values from the input domain and creating an output domain.
Testing is not that efficient because;
1. no false positives
2. incomplete

→ we test on all possible input values for e.g. in the double wali e.g. we would check on not only int but also other types e.g. float. But this method can result in:

1. False positives
2. Compromised accuracy due to output of unintended inputs.

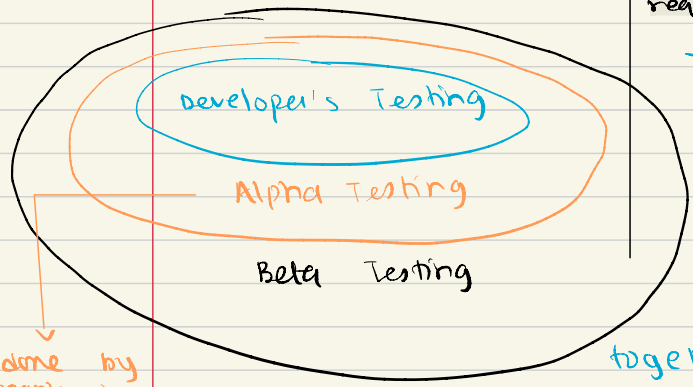
Inspection

→ reviews of the code → walkthroughs
→ code inspected manually → group activity.
→ static. (dry-running)
→ systematic, thorough ⇒ pro
→ informal, subjective. ⇒ con.

Formal Proof

→ sound guarantees ⇒ pro
→ complex, expensive ⇒ con.
→ formal proof of correction

Software Testing



→ done by external users (mainly focuses on non-functional reqs.)

→ unit testing, each developer checks their own function individually
→ integration testing, combine multiple func and test them together to see if they work well together.
→ system testing, entire system with all components checked together. (done for both functional and non-functional reqs)

done by people in the org mainly focuses on functional reqs)

→ Acceptance testing, done by the client to check if it performs as expected and fullfills all reqs.

Q. what does carry exactly mean here

→ regression testing, testing done after refactoring the program after deployment. Easier for maintainable code.

Two main types of software testing:

1. dynamic technique — whitebox
2. optimistic approximation — blackbox

does not catch error on missing path

whitebox: only the code is provided and test cases are made solely on the basis of that without knowing client reqs.

does not catch implementation errors

blackboxing: only client specification provided and test cases built on that basis. (functional testing)

~Test Oracle~

is the set of test cases representing the desired behaviour of the program on the given input.

Sr. No	Test case	Expected Result	Test Result
1			
2			
⋮			

~Test log~

similar to traceability matrix, and keeps track of each test case and their result.

(similar to test oracle, except it also has a column for steps, and pass/fail).

SE lec 17.

why?

→ user screen tested using blackbox
→ complex problems e.g. system or application software tested using whiteboxing

~ Blackbox testing ~

→ Types of errors regarding functional reqs of software

- incorrect or missing funcs
- interface errors.
- initialization and termination error.

Different types of blackbox testing:

1. equivalence class testing; divide input domain into different classes to cover all situations; The classes should be disjoint, and ensure completeness.

valid equivalence class

invalid equivalence class.

- objective is to identify those classes of data, which causes the module to respond in a different manner from other classes.

1. input condition is a range, one valid classes and 2 invalid classes are defined.

invalid class [1, 2, 3] invalid class

input range valid class

should cause module to output an error.

no. of valid and invalid classes depend on input range.

2. input condition requires a specific value; two invalid classes (<, >) specific val, and one valid class (=) the specific val.

3. input condition is boolean; one valid and one invalid class.

→ weak / strong equivalence classes

considering we have three categories with |A| is the no of equivalence class for A.

weak: choosing one variable from each equivalence classes so that all classes are covered. No. of test cases = max (|A|, |B|, |C|)

strong: testing all interactions of all equivalence classes. No. of test cases: |A| x |B| x |C|

SE lec 18:

~identifying boundaries~

- → minimum
- → just above the minimum
- → nominal
- → just below the maximum
- → maximum

?? • boundary value analysis, checks for edge vals of equivalence classes rather than focusing attention on the input condition (input space), test cases are also derived by considering result space.

~decision table testing~

pros:
1. better for complex business conditions.
2. better coverage
3. easy representation
4. if system's behaviour is different for input, not the same for range inputs, then DT works better than ECP & BVA.

better test coverage.

~when you have cause and effect relation b/w entities.

format (T/F)	T	T	T	T	F	F	F	F
size (T/F)	T	F	T	F	T	F	T	F
resolution (T/F)	T	T	F	F	T	T	F	F
output (T/F)	T	F	F	F	F	F	F	F

cause-effect table.

~state transition testing~

example nhi samajh aayi.

exposes defects triggered by moving from one program state to another. (points out bugs in flow of information). Elements include:

1. state: that the software may occupy
2. Transitions: from one state to another
3. Events: that cause a transition
4. Actions: result of a transition

~Testing Strategies~

1. Negative (fuzzy testing): testing w invalid input, to monitor for exceptions such as crashing or failing.
2. Null case testing: testing w null input, ie no data or missing data. often triggers defects.

3. Unit testing: testing each function individually.

done before system testing.

4. Integration testing: combine the unit tested functions to test together, → outputs the aggregated system ready for system testing.

Types of integration testing

1. Big Bang Integration.
2. Top-down "
3. Bottom-up "
4. sandwich.

} Incremental.

~Big Bang integration~

→ all or most of the developed modules are coupled together to form a complete system then used for testing.

→ saves time

→ no need to write stub or drivers.

→ difficult to locate error.

→ wait for all modules to develop.

~Top-down Integration~

- top integrated modules are tested and the branch of the module is tested step-by-step, until end of related module.
- could be depth first or breadth first

depth first: when there is a singular high priority component divided among developers

breadth first: when there are numerous components one assigned to each developer.

Stubs and drivers

→ Stubs are modules that act as temporary replacement for a called module and give the same output as the actual product

→ Drivers are simple programs designed specifically for testing that make calls to these lower layers.

Bottom-up Integration

- lowest level components are tested first, which are then used to facilitate the testing of higher-level components. (repeated until top of heirarchy reached)
- all low-level modules are integrated and tested before moving to the level above.

Steps involved

1. combine low-level components into clusters.
2. a driver is written (if main module is not written) to co-ordinate test case input & output
3. cluster is tested.
4. drivers are removed and clusters are combined moving upward in the program structure.

Sandwich Testing

- combines top-down and bottom-up.

- effective for large projects having no of sub-projects.
- system viewed in layers.

~Load testing~ [large amount of users] → are we testing for data races?

- process of putting demand on a system or device and measuring its response.
- It determine system behavior under normal and peak load.
- identifies max operating capacity of an app, and elements that cause degradation
- better for multi-user systems.

~volume testing~ [large amount of data] → does this test performance or db?

- test software application w large amount of data.

~stress testing~ [alot of users and data at less time]

- to test the stability of a given system or entity.
  - involves testing beyond normal operational capacity, upto the breaking point to observe the result

~Regression testing~

- performed when previously tested code has been (Bug fixed (to specific part of the code))

2. ports to a new env; i.e. operating system.

- ?? exposes defects in code that should not have changed.
- re-executes some or all existing test cases to exercise code that was tested in a previous release or previous test cycle.

"Progressive testing"

- testing new code.
- testing to determine if a change has introduced error is regression testing
- ?? eventually all progressive tests become regressin tests.

• blacbox was supposed to be functional, but volume, load or stress too non-functional Jmi. ??

SE lec 19:

static analysis; detects certain patterns in your code, does not require unit tests.

• to find potential errors; — 1. indentation
2. smells
3. dead code

dynamic analysis; test cases are written down

coverage based test cases.

Mutation Testing:
introduce bugs into your program and run the prog on standard input to see if it is caught.

if bug found kill it.

if bug not caught, code has error.

coverage refers to all lines of code being executes, it may not execute correctly ie. throw error. Any linear code w no if-else conditions requires one test for coverage.

Whitebox testing — structural testing

- static Testing:

• whitebox testing doesn't involve unit tests.
• code walkthroughs, inspections, code reviews.

- Complexity Analysis.

• also tells about heirarchy of classes and

what their depth should be

the more depth our program has, the more complex it is

"code coverage"

always come up w min. no of tests.

1. statement coverage : tries to reach every line (impractical)
2. condition coverage : tries to reach individual conditions
3. Branch coverage : check all branches, check for both T/F case. (stronger than statement coverage)
4. path coverage : checks all possible paths.

2^n test cases

Path Coverage > Condition Coverage > Branch Coverage > statement coverage.

SE lec 20:

"white box testing"

→ Multiple condition decision coverage:

- all outcomes of decisions as well as conditions are tested.

needed because branch coverage doesn't cover all conditions, and condition coverage does not cover all branc

→ Path coverage

- all linearly independant paths in the program are executed atleast once

independant paths: introducing a new node in each independent path which is already not in any other independant path.

path coverage is the most expensive as it subsumes all other types of coverage

- defined in terms of control flow graph.

→ Mc Cabe's cyclomatic matrix:

- an upper bound for the no. of linearly independant paths of program.

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

V(G) → CFG; E → edges in G; N → nodes in G.

→ Cyclomatic complexity:

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

CFG — edges in G — nodes in G.

2. $V(G)$ = number of bounded areas + 1

bounded area; region enclosed by nodes and edge sequence.

3. $V(G)$ = no. of predicates + 1

→ data flow testing:

- explore CFG to find data-flow anamolies.
- paths explored based on data objects:
  1. test paths to ensure all data objects are initialized before use
  2. all defined data objects are used atleast once
- data object categories:
  1. defined, created, initialized (d) — assignment (left), input, output
  2. killed, undefined, released (k)
  3. used (u)
     - computational / calculation
     - predicate.

     assignment (right), parameter to func, predicate.

→ Error / no error:

1. dd : harmless, but sus.

2. dk: probably a bug
3. du: normal
4. kd: normal
5. kk: harmlen, prolly a bug
6. ku: definitely a bug
7. ud: normal (reassignment)
8. uk: normal
9. uu: normal

→ a variable can be used and redefined in a single statement (y = y + x)

→ if a variable is not part of a predicate all other uses are c-use, else p-use.

SE lec 21

→ x :- (d, u)
(1,2), (1,4), (1,7)

• print computational use hota hai.

→ y :- (d, u)
(1,5), (1,6), (1,7), (6,7)

pro: easier.

con: doesn't check all paths.

• A path is (definition-clear) with respect to a variable v if there is no re-definition of v within the path.

pro: all possible paths explored.

• A complete path is a path whose initial node is the start node and whose final node is the exit node.

con: waste due to irrelevant and unnecessary test cases.

• a definition-use pair (du-pair) w.r.t var 'v', is a double (d, u), such that d is a node in the program's flow graph at which v is defined, u is a node or edge at which v is used, and there is a def-clear path w.r.t v from d to u.

PLS look at the slide wali example sahi se

low coverage tests

• all-defs: for every program variable v, atleast one def-clear path from every definition of v to atleast one c-use or p-use of v must be covered.

how?? → higher cyclomatic complexity good for maintainability. can be improved via modularity.

• control flow mein, node mein condition aur edge pe T/F, but in data flow condition on edge and d, u on node.

- all-uses: for every program variable v, atleast one def-clear path from every definition of v to every c-use and every p-use of v must be covered

All-uses > All-defs.

~Mutation testing~

- modifying code segments, to evaluate goodness of our test cases.