



Software Testing



Testing Concepts



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, software under test (SUT) executed with set of test cases
- Failure during testing => defects are present
- No failure => confidence grows, but can not say “defects are absent”
- 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



Test case and test suite

- Test case – a set of test inputs and execution conditions designed to exercise SUT in a particular manner
 - Test case should also specify the expected output – oracle uses this to detect failure
- Test suite - group of related test cases generally executed together



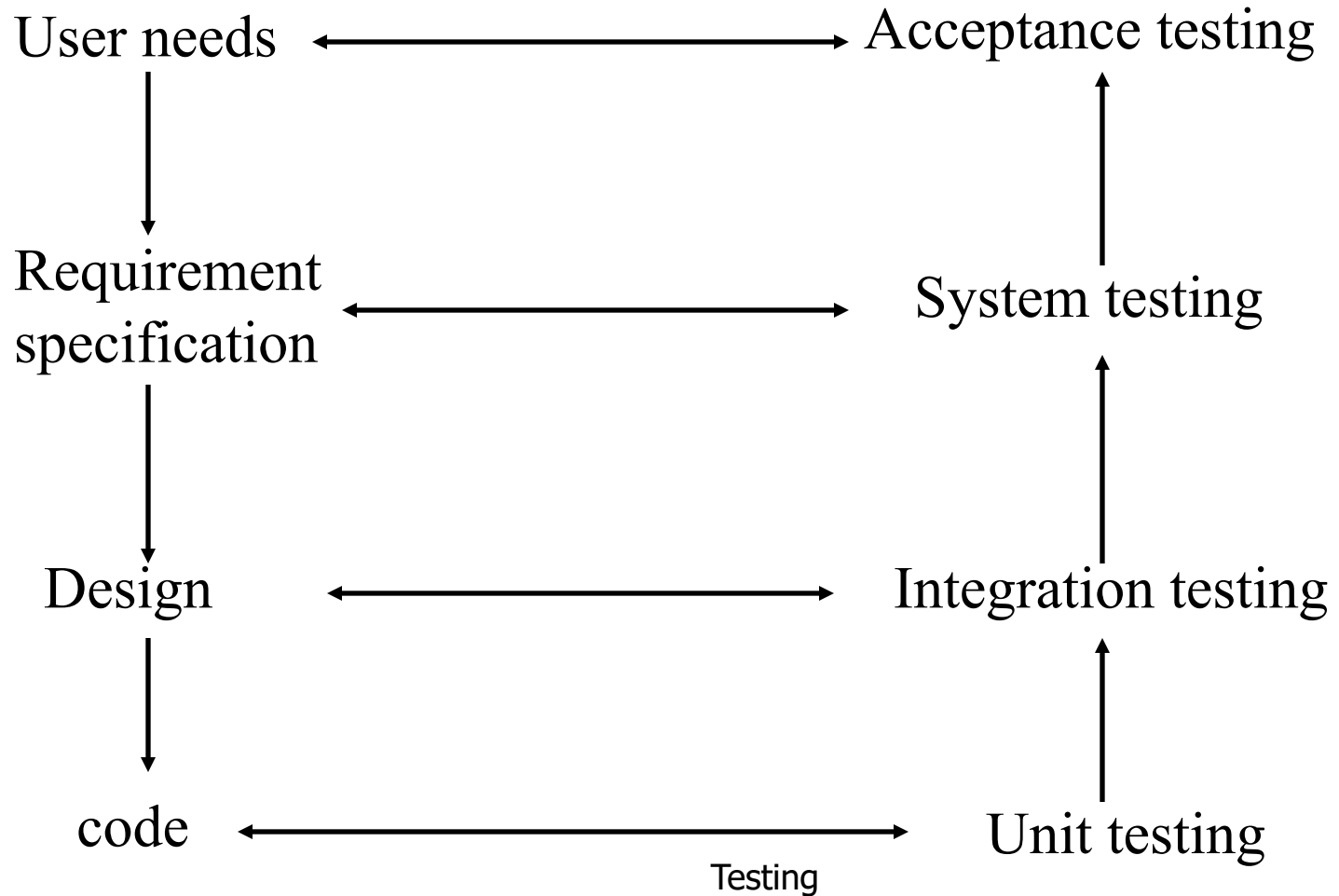
Test harness

- During testing, for each test case in a test suite, conditions have to be set, SUT called with inputs, output checked against expected to declare fail/pass
- Many test frameworks (or test harness) exist that automate the testing process
 - Each test case is often a function/method
 - A test case sets up the conditions, calls the SUT with the required inputs
 - Tests the results through assert statements
 - If any assert fails – declares failure



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



Testing Process



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



Testing...

- Multiple levels of testing are done in a project
- At each level, for each SUT, test cases have to be designed and then executed
- Overall, testing is very complex in a project and has to be done well
- Testing process at a high level has: test planning, test case design, and test execution



Test Plan

- Testing usually starts with test plan and ends with acceptance testing
- Test plan is a general document that defines the scope and approach for testing for the whole project
- Inputs are SRS, project plan, design
- Test plan identifies what levels of testing will be done, what units will be tested, etc in the project



Test Plan...

- Test plan usually contains
 - Test unit specs: what units need to be tested separately
 - Features to be tested: these may include functionality, performance, usability,...
 - Approach: criteria to be used, when to stop, how to evaluate, etc
 - Test deliverables
 - Schedule and task allocation



Test case Design

- Test plan focuses on testing a project; does not focus on details of testing a SUT
- Test case design has to be done separately for each SUT
- Based on the plan (approach, features,..) test cases are determined for a unit
- Expected outcome also needs to be specified for each test case



Test case design...

- Together the set of test cases should detect most of the defects
- Would like the set of test cases to detect any defects, if it exists
- Would also like set of test cases to be small - each test case consumes effort
- Determining a reasonable set of test case is the most challenging task of testing



Test case design

- The effectiveness and cost of testing depends on the set of test cases
- Q: How to determine if a set of test cases is good? I.e. the set will detect most of the defects, and a smaller set cannot catch these defects
- No easy way to determine goodness; usually the set of test cases is reviewed by experts
- This requires test cases be specified before testing – a key reason for having test case specs
- Test case specs are essentially a table



Test case specifications

Seq.No	Condition to be tested	Test Data	Expected result	successful



Test case specifications...

- So for each testing, test case specs are developed, reviewed, and executed
- Preparing test case specifications is challenging and time consuming
 - Test case criteria can be used
 - Special cases and scenarios may be used
- Once specified, the execution and checking of outputs may be automated through scripts
 - Desired if repeated testing is needed
 - Regularly done in large projects



Test case execution

- Executing test cases may require drivers or stubs to be written; some tests can be auto, others manual
 - A separate test procedure document may be prepared
- Test summary report is often an output – gives a summary of test cases executed, effort, defects found, etc
- Monitoring of testing effort is important to ensure that sufficient time is spent
- Computer time also is an indicator of how testing is proceeding



Defect logging and tracking

- A large software may have thousands of defects, found by many different people
- Often person who fixes (usually the coder) is different from who finds
- Due to large scope, reporting and fixing of defects cannot be done informally
- Defects found are usually logged in a defect tracking system and then tracked to closure
- Defect logging and tracking is one of the best practices in industry

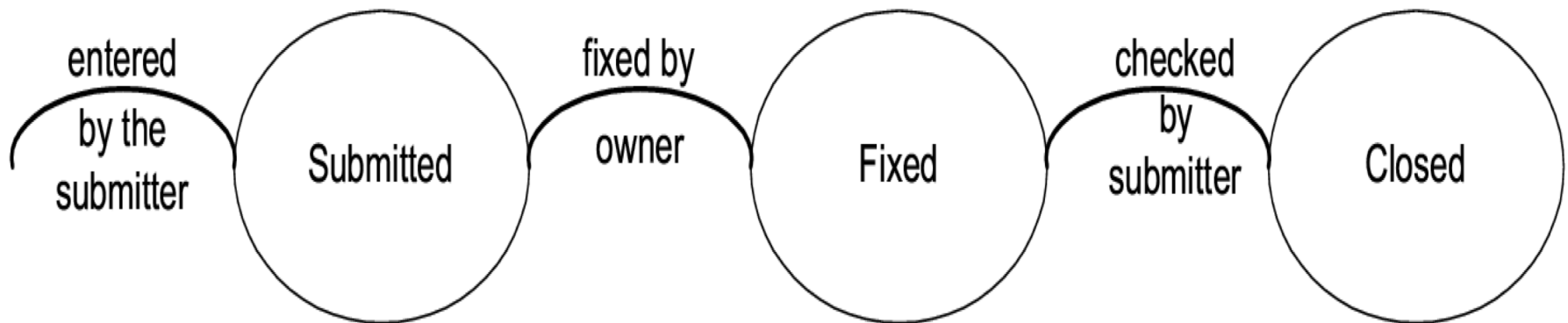


Defect logging...

- A defect in a software project has a life cycle of its own, like
 - Found by someone, sometime and logged along with info about it (submitted)
 - Job of fixing is assigned; person debugs and then fixes (fixed)
 - The manager or the submitter verifies that the defect is indeed fixed (closed)
- More elaborate life cycles possible



Defect logging...





Defect logging...

- During the life cycle, info about defect is logged at diff stages to help debug as well as analysis
- Defects generally categorized into a few types, and type of defects is recorded
 - ODC is one classification
 - Some std categories: Logic, standards, UI, interface, performance, documentation,...



Defect logging...

- Severity of defects in terms of its impact on sw is also recorded
- Severity useful for prioritization of fixing
- One categorization
 - Critical: Show stopper
 - Major: Has a large impact
 - Minor: An isolated defect
 - Cosmetic: No impact on functionality



Defect logging...

- Ideally, all defects should be closed
- Sometimes, organizations release software with known defects (hopefully of lower severity only)
- Organizations have standards for when a product may be released
- Defect log may be used to track the trend of how defect arrival and fixing is happening



Black Box Testing



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
- 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



Black box Testing...

- Premise: Expected behavior is specified.
- Hence just test for specified expected behavior
- How it is implemented is not an issue.
- For modules, specification produced in design specify expected behavior
- For system testing, SRS specifies expected behavior



Black Box Testing...

- Most thorough functional testing - exhaustive testing
 - Software is designed to work for an input space
 - Test the software with all elements in the input space
- Infeasible - too high a cost
- Need better method for selecting test cases
- Different approaches have been proposed



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 impossible
- Approximate it by identifying classes for which different behavior is specified



Equivalence class partitioning...

- Rationale: specification requires same behavior for elements in a class
- Software likely to be constructed such that it either fails for all or for none.
- E.g. if a function was not designed for negative numbers then it will fail for all the negative numbers
- For robustness, should form equivalent classes for invalid inputs also



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



Equivalent class partitioning..

- Should consider eq. classes in outputs also and then give test cases for different classes
- E.g.: Compute rate of interest given loan amount, monthly installment, and number of months
 - Equivalent classes in output: + rate, rate = 0 , -ve rate
 - Have test cases to get these outputs



Equivalence class...

- Once eq classes selected for each of the inputs, test cases have to be selected
 - Select each test case covering as many valid eq classes as possible
 - Or, have a test case that covers at most one valid class for each input
 - Plus a separate test case for each invalid class



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



Example...

- Test cases (i.e. s , n) with first method
 - s : str of $\text{len} < N$ with lower case, upper case, numbers, and special chars, and $n=5$
 - Plus test cases for each of the invalid eq classes
 - Total test cases: $1+3= 4$
- With the second approach
 - A separate str for each type of char (i.e. a str of numbers, one of lower case, ...) + invalid cases
 - Total test cases will be $5 + 2 = 7$



Boundary value analysis

- Programs often fail on special values
- These values often lie on boundary of equivalence classes
- Test cases that have boundary values have *high yield*
- These are also called *extreme cases*
- A BV test case is a set of input data that lies on the edge of a eq class of input/output



BVA...

- For each equivalence class
 - choose values on the edges of the class
 - choose values just outside the edges
- E.g. if $0 \leq x \leq 1.0$
 - 0.0 , 1.0 are edges inside
 - -0.1,1.1 are just outside
- E.g. a bounded list - have a null list , a maximum value list
- Consider outputs also and have test cases generate outputs on the boundary

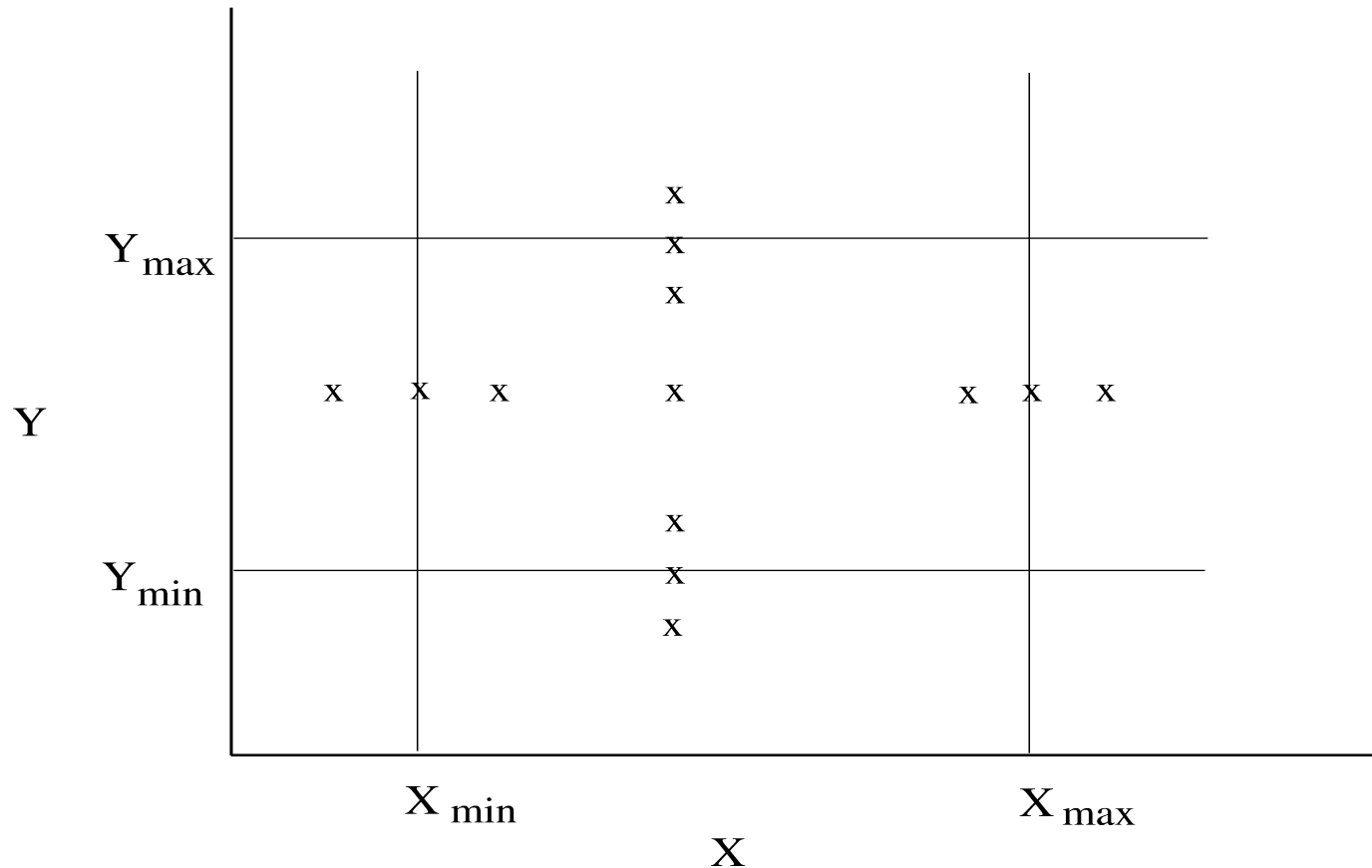


BVA...

- In BVA we determine the value of vars that should be used
- If input is a defined range, then there are 6 boundary values plus 1 normal value (tot: 7)
- If multiple inputs, how to combine them into test cases; two strategies possible
 - Try all possible combination of BV of diff variables, with n vars this will have 7^n test cases!
 - Select BV for one var; have other vars at normal values + 1 of all normal values



BVA.. (test cases for two vars – x and y)





Pair-wise testing

- Often many parameters determine the behavior of a software system
- The parameters may be inputs or settings, and take different values (or different value ranges)
- Many defects involve one condition (single-mode fault), eg. software not being able to print on some type of printer
 - Single mode faults can be detected by testing for different values of different parameters
 - If n parameters and each can take m values, we can test for one different value for each parameter in each test case
 - Total test cases: m



Pair-wise testing...

- All faults are not single-mode and sw may fail at some combinations
 - Eg tel billing sw does not compute correct bill for night time calling (one parm) to a particular country (another parm)
 - Eg ticketing system fails to book a biz class ticket (a parm) for a child (a parm)
- Multi-modal faults can be revealed by testing diff combination of parm values
- This is called combinatorial testing



Pair-wise testing...

- Full combinatorial testing not feasible
 - For n parms each with m values, total combinations are n^m
 - For 5 parms, 5 values each (tot: 3125), if one test is 5 mts, tot time > 1 month!
- Research suggests that most such faults are revealed by interaction of a pair of values
- I.e. most faults tend to be double-mode
- For double mode, we need to exercise each pair – called pair-wise testing



Pair-wise testing...

- In pair-wise, all pairs of values have to be exercised in testing
- If n parms with m values each, between any 2 parms we have $m*m$ pairs
 - 1st parm will have $m*m$ with $n-1$ others
 - 2nd parm will have $m*m$ pairs with $n-2$
 - 3rd parm will have $m*m$ pairs with $n-3$, etc.
 - Total no of pairs are $m*m*n*(n-1)/2$



Pair-wise testing...

- A test case consists of some setting of the n parameters
- Smallest set of test cases when each pair is covered once only
- A test case can cover a maximum of $(n-1)+(n-2)+\dots=n(n-1)/2$ pairs
- In the best case when each pair is covered exactly once, we will have m^2 different test cases providing the full pair-wise coverage



Pair-wise testing...

- Generating the smallest set of test cases that will provide pair-wise coverage is non-trivial
- Efficient algos exist; efficiently generating these test cases can reduce testing effort considerably
 - In an example with 13 parms each with 3 values pair-wise coverage can be done with 15 testcases
- Pair-wise testing is a practical approach that is widely used in industry



Pair-wise testing, Example

- A sw product for multiple platforms and uses browser as the interface, and is to work with diff OSs
- We have these parms and values
 - OS (parm A): Windows, Solaris, Linux
 - Mem size (B): 128M, 256M, 512M
 - Browser (C): IE, Netscape, Mozilla
- Total no of pair wise combinations: 27
- No of cases can be less



Pair-wise testing...

Test case	Pairs covered
a1, b1, c1	(a1,b1) (a1, c1) (b1,c1)
a1, b2, c2	(a1,b2) (a1,c2) (b2,c2)
a1, b3, c3	(a1,b3) (a1,c3) (b3,c3)
a2, b1, c2	(a2,b1) (a2,c2) (b1,c2)
a2, b2, c3	(a2,b2) (a2,c3) (b2,c3)
a2, b3, c1	(a2,b3) (a2,c1) (b3,c1)
a3, b1, c3	(a3,b1) (a3,c3) (b1,c3)
a3, b2, c1	(a3,b2) (a3,c1) (b2,c1)
a3, b3, c2	(a3,b3) (a3,c2) (b3,c2)



Special cases

- Programs often fail on special cases
- These depend on nature of inputs, types of data structures, etc.
- No good rules to identify them
- One way is to guess when the software might fail and create those test cases
- Also called error guessing
- Play the sadist & hit where it might hurt



Error Guessing

- Use experience and judgement to guess situations where a programmer might make mistakes
- Special cases can arise due to assumptions about inputs, user, operating environment, business, etc.
- E.g. A program to count frequency of words
 - file empty, file non existent, file only has blanks, contains only one word, all words are same, multiple consecutive blank lines, multiple blanks between words, blanks at the start, words in sorted order, blanks at end of file, etc.
- Perhaps the most widely used in practice



State-based Testing

- Some systems are state-less: for same inputs, same behavior is exhibited
- Many systems' behavior depends on the state of the system i.e. for the same input the behavior could be different
- I.e. behavior and output depend on the input as well as the system state
- System state – represents the cumulative impact of all past inputs
- State-based testing is for such systems



State-based Testing...

- A system can be modeled as a state machine
- The state space may be too large (is a cross product of all domains of vars)
- The state space can be partitioned in a few states, each representing a logical state of interest of the system
- State model is generally built from such states



State-based Testing...

- A state model has four components
 - States: Logical states representing cumulative impact of past inputs to system
 - Transitions: How state changes in response to some events
 - Events: Inputs to the system
 - Actions: The outputs for the events



State-based Testing...

- State model shows what transitions occur and what actions are performed
- Often state model is built from the specifications or requirements
- The key challenge is to identify states from the specs/requirements which capture the key properties but is small enough for modeling



State-based Testing, example...

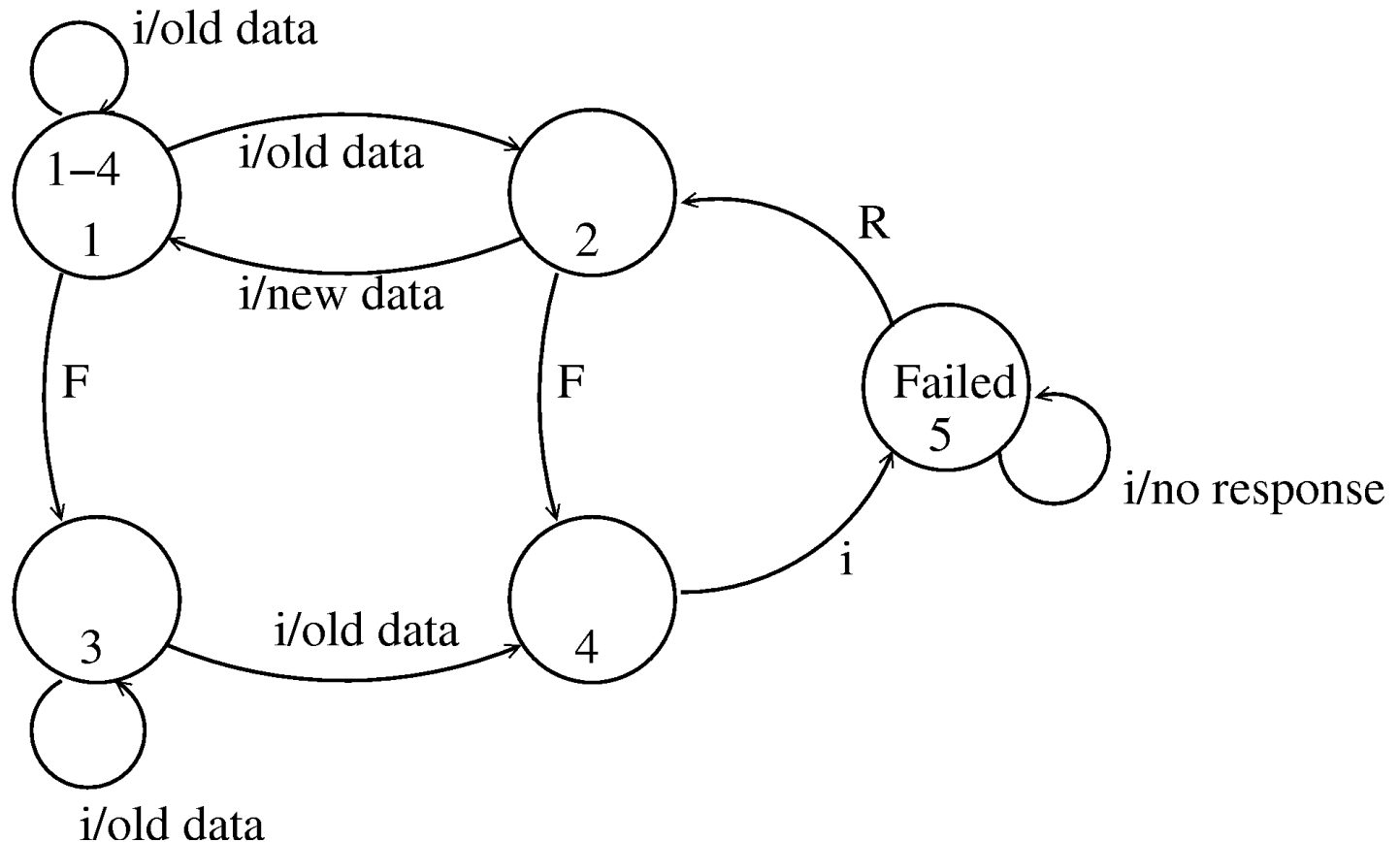
- Consider the student survey example (discussed in Chap 4)
 - A system to take survey of students
 - Student submits survey and is returned results of the survey so far
 - The result may be from the cache (if the database is down) and can be up to 5 surveys old



State-based Testing, example...

- In a series of requests, first 5 may be treated differently
- Hence, we have two states: one for req no 1-4 (state 1), and other for 5 (2)
- The db can be up or down, and it can go down in any of the two states (3-4)
- Once db is down, the system may get into failed state (5), from where it may recover

State-based Testing, example...





State-based Testing...

- State model can be created from the specs or the design
- For objects, state models are often built during the design process
- Test cases can be selected from the state model and later used to test an implementation
- Many criteria possible for test cases



State-based Testing criteria

- All transaction coverage (AT): test case set T must ensure that every transition is exercised
- All transitions pair coverage (ATP). T must execute all pairs of adjacent transitions (incoming and outgoing transition in a state)
- Transition tree coverage (TT). T must execute all simple paths (i.e. a path from start to end or a state it has visited)



Example, test cases for AT criteria

SNo	Transition	Test case
1	1 -> 2	Req()
2	1 -> 2	Req(); req(); req(); req();req(); req()
3	2 -> 1	Seq for 2; req()
4	1 -> 3	Req(); fail()
5	3 -> 3	Req(); fail(); req()
6	3 -> 4	Req(); fail(); req(); req(); req();req(); req()
7	4 -> 5	Seq for 6; req()
8	5 -> 2	Seq for 6; req(); recover()



State-based testing...

- SB testing focuses on testing the states and transitions to/from them
- Different system scenarios get tested; some easy to overlook otherwise
- State model is often done after design information is available
- Hence it is sometimes called *grey box testing* (as it not pure black box)



White Box Testing



White box testing

- Black box testing focuses only on functionality
 - What the program does; not how it is implemented
- White box testing focuses on implementation
 - Aim is to exercise different program structures with the intent of uncovering errors
- Is also called *structural testing*
- Various criteria exist for test case design
- Test cases have to be selected to satisfy coverage criteria



Types of structural testing

- Control flow based criteria
 - looks at the coverage of the control flow graph
- Data flow based testing
 - looks at the coverage in the definition-use graph
- Mutation testing
 - looks at various mutants of the program
- We will discuss only control flow based criteria – these are most commonly used



Control flow based criteria

- Considers the program as control flow graph
 - Nodes represent code blocks – i.e. set of statements always executed together
 - An edge (i,j) represents a possible transfer of control from i to j
- Assume a start node and an end node
- A path is a sequence of nodes from start to end



Statement Coverage Criterion

- Criterion: Each statement is executed at least once during testing
- I.e. set of paths executed during testing should include all nodes
- Limitation: does not require a decision to evaluate to false if no else clause
- E.g. : `abs (x) : if (x >= 0) x = -x; return(x)`
 - The set of test cases $\{x = 0\}$ achieves 100% statement coverage, but error not detected
- Guaranteeing 100% coverage not always possible due to possibility of unreachable nodes



Branch coverage

- Criterion: Each edge should be traversed at least once during testing
- i.e. each decision must evaluate to both true and false during testing
- Branch coverage implies stmt coverage
- If multiple conditions in a decision, then all conditions need not be evaluated to T and F



Control flow based...

- There are other criteria too - path coverage, predicate coverage, cyclomatic complexity based, ...
- None is sufficient to detect all types of defects (e.g. a program missing some paths cannot be detected)
- They provide some quantitative handle on the breadth of testing
- More used to evaluate the level of testing rather than selecting test cases



Tool support and test case selection

- Two major issues for using these criteria
 - How to determine the coverage
 - How to select test cases to ensure coverage
- For determining coverage - tools are essential
- Tools also tell which branches and statements are not executed
- Test case selection is mostly manual - test plan is to be augmented based on coverage data



In a Project

- Both functional and structural should be used
- Test plans are usually determined using functional methods; during testing, for further rounds, based on the coverage, more test cases can be added
- Structural testing is useful at lower levels only; at higher levels ensuring coverage is difficult
- Hence, a combination of functional and structural at unit testing
- Functional testing (but monitoring of coverage) at higher levels



Comparison

	Code Review	Structural Testing	Functional Testing
Computational	M	H	M
Logic	M	H	M
I/O	H	M	H
Data handling	H	L	H
Interface	H	H	M
Data defn.	M	L	M
Database	H	M	M



Metrics



Data

- Defects found are generally logged
- The log forms the basic data source for metrics and analysis during testing
- Main questions of interest for which metrics can be used
 - How good is the testing that has been done so far?
 - What is the quality or reliability of software after testing is completed?



Coverage Analysis

- Coverage is very commonly used to evaluate the thoroughness of testing
- This is not white box testing, but evaluating the overall testing through coverage
- Organization sometimes have guidelines for coverage, particularly at unit level (say 90% before checking code in)
- Coverage of requirements also checked – often by evaluating the test suites against requirements



Reliability Estimation

- High reliability is an important goal to be achieved by testing
- Reliability is usually quantified as a probability or a failure rate or mean time to failure
 - $R(t) = P(X > t)$
 - MTTF = mean time to failure
 - Failure rate
- For a system reliability can be measured by counting failures over a period of time
- Measurement often not possible for software as due to fixes reliability changes, and with one-off, not possible to measure



Reliability Estimation...

- Sw reliability estimation models are used to model the failure followed by fix model of software
- Data about failures and their times during the last stages of testing is used by these model
- These models then use this data and some statistical techniques to predict the reliability of the software
- Software reliability growth models are quite complex and sophisticated



Reliability Estimation

- Simple method of measuring reliability achieved during testing
 - Failure rate, measured by no of failures in some duration
- For using this for prediction, assumed that during this testing software is used as it will be by users
- Execution time is often used for failure rate, it can be converted to calendar time



Defect removal efficiency

- Basic objective of testing is to identify defects present in the programs
- Testing is good only if it succeeds in this goal
- Defect removal efficiency of a QC activity = % of present defects detected by that QC activity
- High DRE of a quality control activity means most defects present at the time will be removed



Defect removal efficiency ...

- DRE for a project can be evaluated only when all defects are known, including delivered defects
- Delivered defects are approximated as the number of defects found in some duration after delivery
- The *injection stage* of a defect is the stage in which it was introduced in the software, and *detection stage* is when it was detected
 - These stages are typically logged for defects
- With injection and detection stages of all defects, DRE for a QC activity can be computed



Defect Removal Efficiency ...

- DREs of different QC activities are a process property - determined from past data
- Past DRE can be used as expected value for this project
- Process followed by the project must be improved for better DRE



Summary

- Testing plays a critical role in removing defects, and in generating confidence
- Testing should be such that it catches most defects present, i.e. a high DRE
- Multiple levels of testing needed for this
- Incremental testing also helps
- At each testing, test cases should be specified, reviewed, and then executed



Summary ...

- Deciding test cases during planning is the most important aspect of testing
- Two approaches – black box and white box
- Black box testing - test cases derived from specifications.
 - Equivalence class partitioning, boundary value, cause effect graphing, error guessing
- White box - aim is to cover code structures
 - statement coverage, branch coverage



Summary...

- In a project both used at lower levels
 - Test cases initially driven by functional
 - Coverage measured, test cases enhanced using coverage data
- At higher levels, mostly functional testing done; coverage monitored to evaluate the quality of testing
- Defect data is logged, and defects are tracked to closure
- The defect data can be used to estimate reliability, DRE