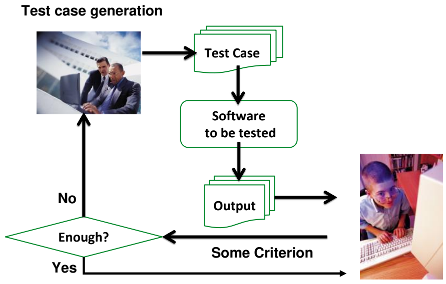
Week 1

Software testing: process of executing program/system with intent of finding errors
Fault: incorrect portions of code (can be missing as well as incorrect)
Failure: observable correct behaviour of program
Error: cause of fault, something bad programmer did (conceptual, typo, etc)
Bug: informal term for fault
Test case: set of test inputs, execution conditions, expected results developed for particular objective, such as to exercise particular program path ot verify compliance with specific requirement

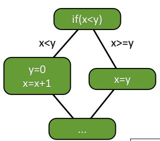
A Typical Software Testing Process



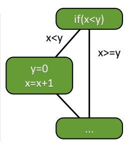
Testing: find inputs that cause failure of software, failure unknown, performed by testers
Debugging: process of finding & fixing fault given failure, failure is known, performed by devs
Black-Box/Functional Testing: identify functions & design test cases to check whether functions are correctly performed by software (formal & informal specs)
Equivalence partitioning: divide into partitions, select 1 test case from each partition, partitions must be disjointed (no input belongs to more than 1 partition) & all partitions must cover entire input domain
Equivalence partitioning examples: isEven then even & odd, password min 8 & max 12 characters then less than, valid, more than
Boundary-Value analysis: partition input domain, identify boundaries, select test data (for range $[R_1, R_2]$ less than R_1, equal to R_1, between, equal to R_2, greater than R_2, for unordered sets select in & not in, for equality select equal & not equal, for sets, lists select empty & not empty)
White box/structural testing: generate test cases based on program structure, abstract program to control flow graph (node is sequence of statements, edge is transfers of control)

CFG : The if statement

```
if (x < y)
{
  y = 0;
  x = x + 1;
}
else
{
  x = y;
}
...
```

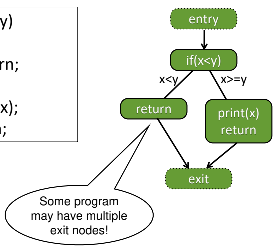



```
if (x < y)
{
  y = 0;
  x = x + 1;
}
...
```



CFG : The dummy nodes

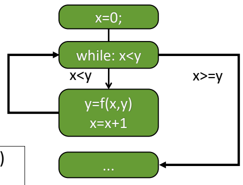
```
if (x < y)
{
  return;
}
print (x);
return;
```



CFG : while and for loops

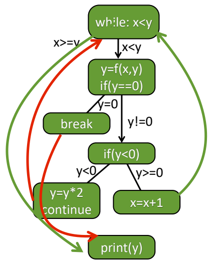
```
x=0;
while (x < y)
{
  y = f (x, y);
  x = x + 1;
}
...
```

```
for (x = 0; x < y; x++)
{
  y = f (x, y);
}
```



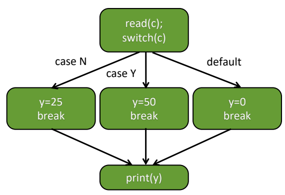
CFG: break and continue

```
while (x < y)
{
  y = f (x, y);
  if (y == 0) {
    break;
  } else if (y<0) {
    y = y*2;
    continue;
  }
  x = x + 1;
}
print (y);
```



CFG: switch

```
read ( c ) ;
switch ( c )
{
  case 'N':
    y = 25;
    break;
  case 'Y':
    y = 50;
    break;
  default:
    y = 0;
    break;
}
print (y);
```

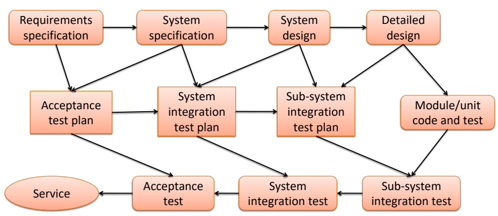


Coverage types: statement, branch, path (infinite if loop exists), strictly subsumes all beforehand

Week 2

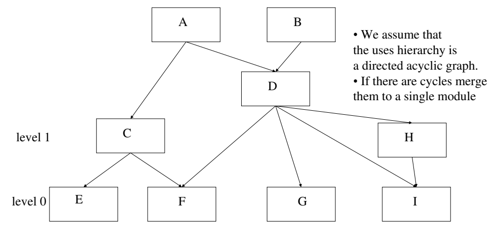
Test oracle: expected output of software for given input, part of test case
Test driver: software framework that can load collection of test cases or test suite
Test suite: collection of test cases

The V-model of development



Testing types:
Unit/Module: test single module in isolated environment, use drivers & stubs for isolation
Integration: test parts of system by combining modules, integrated collection of modules tested as group or partial system
System: test system as a whole after integration phase
Acceptance: test system as a whole to find out if it satisfies requirements specifications
Driver: program that calls interface procedures of module being tested & reports results, simulates module that calls module currently being tested, also provides access to global variables for module under test
Stub: program that has same interface as module being used by module being tested but simpler, simulates module called by module being tested
Mock objects: create object that mimics behaviour needed for testing

Module Structure



- A uses C and D; B uses D; C uses E and F; D uses F, G, H and I; H uses I
- Modules A and B are at level 3; Module D is at level 2
Modules C and H are at level 1; Modules E, F, G, I are at level 0
- level 0 components do not use any other components
- level *i* components use at least one component on level *i*-1 and no component at a level higher than *i*-1

Integration types:
Bottom-Up: only terminal modules tested in isolation, requires drivers but not stubs (since lower levels are tested already)
Top-down: modules tested in isolation are modules at highest level, requires stubs but not drivers
Sandwich: begin both bottom-up & top-down, meet at predetermined point in middle
Big bang: every module unit tested, then integrate all at once, no driver or stub needed but may be hard to isolate bugs
System/Acceptance testing: can construct test case based on requirements specifications, main purpose is to assure that system meets requirements, alpha testing performed within development organisation, beta testing performed by select group of friendly customers

Week 3

Basis Path Testing: between branch & path coverage, fulfills branch testing & tests all independent paths that could be used to construct any arbitrary path through computer program
Independent path: includes some vertices/edges not covered in other path
Cyclomatic complexity: $e - n + 2p$, e is edges, n is nodes, p is number of connected components, or $1 + d$, d is loops or decision points, upper bound on number of test cases to guarantee coverage of all statements
Decision Coverage: executing true/false of decision
Condition Coverage: executing true/false of each condition
Condition/Decision Coverage: DC & CC, better than either
Multiple Condition Coverage: whether every possible combination of boolean sub-expressions occurs, test cases are truth table, 2^n test cases for n conditions
Modified C/DC: for each basic condition C, 2 test cases, values of all evaluated conditions except C are the same, compound decision as a whle evaluates to true for 1 & false for the other, subsumed by MCC & subsumes CC, DC, C/DC, stronger than statement & branch
MC/DC coverage: each entry & exit point invoked, each decision takes every possible outcome, each condition in a decision takes every possible outcome, each condition in decision is shown to independently affect outcome of decision, independence of condition is shown by proving that only one condition changes at a time

MC/DC: linear complexity

- N+1 test cases for N basic conditions

`(((a || b) && c) || d) && e`

Test Case	a	b	c	d	e	outcome
(1)	<u>true</u>	--	<u>true</u>	--	<u>true</u>	true
(2)	false	<u>true</u>	true	--	true	true
(3)	true	--	false	<u>true</u>	true	true
(4)	true	--	true	--	<u>false</u>	false
(5)	true	--	<u>false</u>	<u>false</u>	--	false
(6)	<u>false</u>	<u>false</u>	--	false	--	false

- Underlined values independently affect the output of the decision

Table 1. Types of Structural Coverage

Coverage Criteria	Statement Coverage	Decision Coverage	Condition Coverage	Condition/ Decision Coverage	MC/DC	Multiple Condition Coverage
Every point of entry and exit in the program has been invoked at least once		•	•	•	•	•
Every statement in the program has been invoked at least once	•					
Every decision in the program has taken all possible outcomes at least once		•		•	•	•
Every condition in a decision in the program has taken all possible outcomes at least once			•	•	•	•
Every condition in a decision has been shown to independently affect that decision's outcome					•	•[8]
Every combination of condition outcomes within a decision has been invoked at least once						•

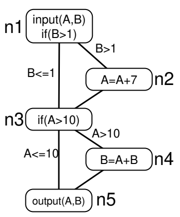
Week 4

Dataflow Coverage: considers how data gets accessed & modified in system & how it can get corrupted
Common access-related bugs: using unde-

fined/uninitializsed variable, deallocating/reinitialising variable before constructed/initialised/used, deleting collection object leaving members unaccessible
Variable definition: defined whenever value modified (LHS of assignment, input statement, call-by-reference)
Variable use: used whenever value read (RHS of assignment, call-by-value, branch statement predicate)
p-use: use in predicate of branch statement
c-use: any other use
Use & redefine in single statement: both sides of assignment, call-by-reference
du-pair: with respect to variable v is a pair (d, u) such that d is a node defining v, u is a node/edge using v (if p-use u is outgoing edge of predicate), there is a def-clear path with respect to v from d to u
Definition clear: with respect to variable v if no variable re-definition of v on path

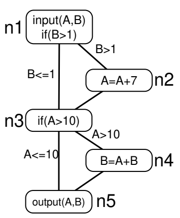
Identifying du-pairs – variable **A**

du-pair	path(s)
(1,2)	<1,2>
(1,4)	<1,3,4>
(1,5)	<1,3,4,5>
	<1,3,5>
(1,<3,4>)	<1,3,4>
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4>
(2,5)	<2,3,4,5>
	<2,3,5>
(2,<3,4>)	<2,3,4>
(2,<3,5>)	<2,3,5>



Identifying du-pairs – variable **B**

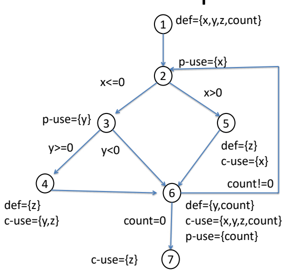
du-pair	path(s)
(1,4)	<1,2,3,4>
	<1,3,4>
(1,5)	<1,2,3,5>
	<1,3,5>
(1,<1,2>)	<1,2>
(1,<1,3>)	<1,3>
(4,5)	<4,5>



Dataflow test coverage criteria:
All-Defs: for every variable v, at least one def-clear path from every definition of v to at least one c-use or p-use of v must be covered
All-P/C-Uses: for every variable v, at least one def-clear path from every definition of v to every p/c-use of v must be covered
All-Uses: all du-pairs covered
Notations:
$d_1(x)$: definition of variable x in node i
$u_i(x)$: use of variable x in node i $dcu(d_i(x)) = dcu(x, i)$: set of c-uses with respect to $d_1(x)$
$dpu(d_i(x)) = dpu(x, i)$: set of p-uses with respect to $d_1(x)$

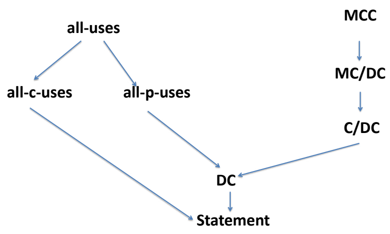
Another Example

```
begin
  float x,y,z=0.0;
  int count;
  input(x,y,count);
  do{
    if(x≤0) {
      if(y≥0)  {
        z=y*z+1;
      }
    }
    else{
     z=1/x;
    }
    y=x*y+z;
    count=count-1;}
  while(count>0)
  output(z);
end
```



All-C-Uses for above: $dcu(x, 1) + dcu(y, 1) + dcu(y, 6) + dcu(z, 1) + dcu(z, 4) + dcu(z, 5) + dcu(count, 1) + dcu(count, 6) = 2+2+2+3+3+3+1+1 = 17$
All-P-Uses for above: $dpu(x, 1) + dpu(y, 1) + dpu(y, 6) + dpu(z, 1) + dpu(z, 4) + dpu(z, 5) + dpu(count, 1) + dpu(count, 6) = 2+2+2+0+0+0+2+2 = 10$ (note this includes using the initial count definition even though it will always be redefined (-1) before the comparison)

Relationships among some of the coverage criteria



Week 5
Program mutation: create artificial bugs by injecting changes to statements of programs, simulate subtle bugs in real programs
Mutation testing: software testing technique based on program mutation, can be used to evaluate test effectiveness & enhance test suite, can be stronger than control/data-flow coverage, extremely costly since need to run whole test suite against each mutant
Mutation testing steps: applies artificial changes based on mutation operators to generate mutants (each mutant with ony one artificial bug), run test suite against each mutant (if any test fails mutant killed, else survives), compute mutation score
Symbolic execution/evaluation: analyse program to determine what inputs cause each part of program to execute, execute programs with symbols (track symbolic state rather than concrete input, when execute one path actually simulate many test inputs (since considering all inputs that can exercise same path))
Problems with symbolic execution:
Path explosion: 2^n paths for n branches, infinite paths for unbounded loops, calculate constraints for all paths is infeasible for real software
Constraint too complex: especially for large programs, also it is NP-complete
Input sub-domain: set of inputs satisfying path condition
Searching input to execute path: equivalent to solving associated path condition

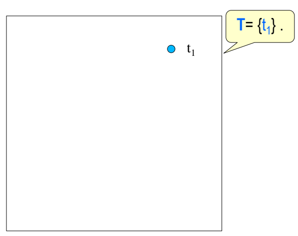
Example

```
y = read();             y=s, s is a symbolic variable for input
p = 1;                  p = 1, y = s
while(y < 10){          p = 1, y = s
     y = y + 1;          s<10, y = s + 1, p = 1
     if y >2
       p = p + 1;          2 < s + 1< 10, y = s + 1, p = 2
     else
       p = p + 2;         s + 1<=2, y = s + 1, p = 3
}
print (p);
```

Week 6
Random testing: random number generator (monkeys) to generate test cases, also called fuzz testing, monkey testing, slelect tests from entire input domain (set of all possible inputs) randomly & independently, no guide towards failure-causing inputs
Adaptive Random Testing: achieve even spread of test cases

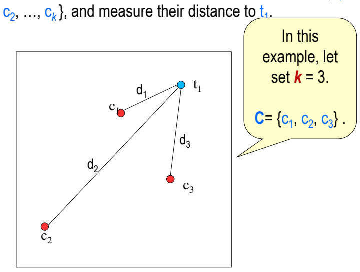
Fixed Size Candidate Set ART

Step 1. Randomly select the first input, namely t_1, and store it in a list (called T)



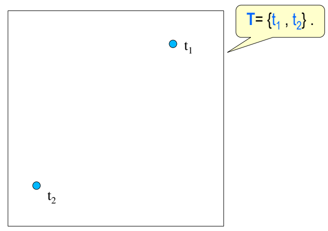
Fixed Size Candidate Set ART (cont.)

Step 2. Construct ***k*** random inputs to form a **candidate set** C = {c_1, c_2, ..., c_k}, and measure their distance to t_1.



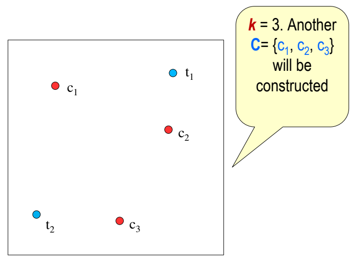
Fixed Size Candidate Set ART (cont.)

Step 3. Select the candidate which is the farthest away from t_1 to be the next test case. We name it t_2 and store it in T.



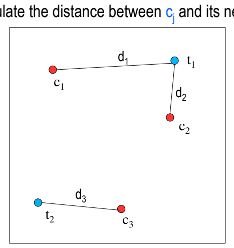
Fixed Size Candidate Set ART (cont.)

Step 4. Re-construct another **candidate set** C with ***k*** random inputs.



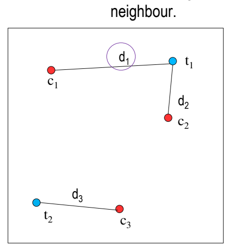
Fixed Size Candidate Set ART (cont.)

Step 5. For each candidate c_j in C, do the following
1. find which test case in T is the nearest neighbour of c_j
2. calculate the distance between c_j and its nearest neighbour.



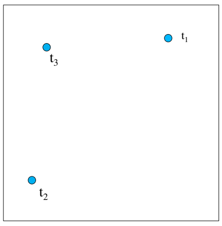
Fixed Size Candidate Set ART (cont.)

Step 6. Select the candidate with the longest distance to its nearest neighbour.



Fixed Size Candidate Set ART (cont.)

Step 7. Store the selected candidate in **T**



Distance in ART: can use Euclidean distance, if $p = (p_1, p_2, \ldots, P_n)$ & $q = (q_1, q_2, \ldots, q_n)$ are 2 points in n-dimensional space, $d(p,q) = d(q,p) = \sqrt{(q_1-p_1)^2 + (q_2-p_2)^2 + \cdots + (q_n-p_n)^2} = \sqrt{\sum_{i=1}^{n}(q_i-p_i)^2}$

ART Algorithm /2

```
Algorithm 2:
initial_test_data := randomly generate a test data from the input domain;
selected_set := { initial_test_data };
counter := 1;
total_number_of_candidates := 10;
use initial_test_data to test the program;
if (program output is incorrect) then
        reveal_failure := true;
else
        reveal_failure := false;
end_if
while (not reveal_failure) do
        candidate_set := {};
        test_data := Select_The_Best_Test_Data(selected_set, candidate_set,
                             total_number_of_candidates);
        use test_data to test the program;
        if (program output is incorrect) then
                reveal_failure := true;
        else
                selected_set := selected_set + { test_data };
                counter := counter + 1;
        end_if
end_while
output counter;
```

T. Y. Chen, H. Leung, and I. K. Mak. Adaptive random testing. In Proceedings of the 9th Asian Computing Science Conference, pages 320–329, 2004

Random white-box testing: generate random method invocations & random parameters

Fuzz testing: random testing technique that involves providing invalid, unexpected or random data as inputs to program, commonly used to discover coding errors & unknown vulnerabilities in software, OS or networks by inputting massive amounts of random data (fuzz) to system in attempt to make it crash, cost-effective alternative to more systematic testing techniques

Fuzz Testing Example /1

- Standard HTTP GET request
  - GET /index.html HTTP/1.1

Anomalous requests:

GET ///////index.html HTTP/1.1
GET %n%n%n%n%n%n.html HTTP/1.1
GET /AAAAAAAAAAAAAA.html HTTP/1.1
GET /index.html HTTTTTTTTTTTTTP/1.1
GET /index.html HTTP/1.1.1.1.1.1.1.1

Ways to generate inputs:
Mutation based/dumb fuzzing: little/no knowledge of input sctructure assumed, anomalies added to existing valid inputs, may be completely random of follow heuristics (remove NL, shift char forward), get inputs, optionally mutate them, feed it to program, record if it crached & input causing it
Generation based smart fuzzing: test cases generated from some description fo format (RFC, docs, etc), anomalies added to each possible spot in inputs, knowledge of protocol should give better results than random fuzzing

Fuzzing rules of thumb: protocol specific knowledge very helpful (generational tends to beat mutation, better specs make better fuzzers), more fuzzers better (each implementation varies different fuzzers find different bugs), the longer it runs the more bugs found, best results come from guiding process (notice where get stuck, use profiling), code coverage useful for guiding process

Fuzz testing +/-: intuitively simple, but need to figure out how to check the output, corner faults might escape detection, debugging with randomly generated input is challenging

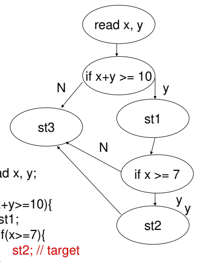
Search-based testing: deem test case generation as search problem, based on random testing & focus on input domains, use code coverage as guidance (try to generate test case that covers certain code element such as method, statement, branch)

Metaheuristic search:
Hill climbing: start from random point, try all neighboring points, go to point with highest value until all neighboring points have value lower than current point, easy to find local optimal
Annealing simulation: adaptation of hill climbing, has probability to move after reached local peak, probability drops as time goes by
Genetic algorithm: simualte process of evolution, start with random points, select number of best points, combine & mutate points until no more improvements can be made

Transform testing to search: list of random test cases as start point, each test case is point in input domain, use various metaheuristic search algorithms to find test cases, measure how well we have solved the problem (use simeple fitness function, how fat is already covered elements from target code elements, try to make it 0)

An example with hill climbing

- Target is st2
- Start from 0, 0
- f(0, 0) = 10 value gap
- Try (0,1) (1,0), (0,-1), (-1,0)
- Go to (0,1)
- Until reach (0,10)
- f(0,10) = 7 value gap
- Increase x
- Until reach (7,10)
- Done!




```
read x, y;
if(x+y>=10){
   st1;
   if(x>=7){
      st2; // target
   }
}
st3;
```

Week 7

Combinatorial Testing: instead of all possible combinations generate subset to satisfy some well-defined combination strategies, not every variable contributes to every fault, often fault caused by interactions among few variables, can dramatically reduce number of combinations to be covered but remains very effective in terms of fault detection

t-way Interaction: fault triggered by certain combination of t input values, simple fault is $t = 1$, pairwise is $t = 2$

Best size for t: 70% failures detected by $t = 2$, max for fault triggering was $t = 6$ for certain interactions (medical devices & NASA distributed database $t = 4$, medical 98% $t = 2$, web server & browser actually 6)

Each Choice Coverage

- Target at 1-way interaction: each variable value must be covered in at least one test case
- Also called: **All-values Testing**.
- Consider the previous example, a test set that satisfies each choice coverage is the following:

{(A, 1, x), (B, 2, y), (A, 3, x)}

Three variables:
P1 = (A, B), P2 = (1, 2, 3), and P3 = (x, y),

Pairwise Coverage

- Target at 2-way Interaction: Given any two variables, every combination of values of these two variables are covered in at least one test case.
- Also called 2-way coverage, pairwise testing.
- A pairwise test set of the previous example is the following:

P1	P2	P3
A	1	x
A	2	x
A	3	x
A	-	y
B	1	y
B	2	y
B	3	y
B	-	x

P1 = (A, B), P2 = (1, 2, 3), and P3 = (x, y),

3-way Coverage

- Target at 3-way interaction: Given any three variables, every combination of values of these three variables are covered in at least one test case.

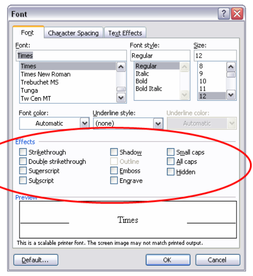
A 3-way coverage test set

(A, 1, x),
(A, 1, y),
(A, 2, x),
(A, 2, y),
(A, 3, x),
(A, 3, y),
(B, 1, x),
(B, 1, y),
(B, 2, x),
(B, 2, y),
(B, 3, x),
(B, 3, y)

P1 = (A, B), P2 = (1, 2, 3), and P3 = (x, y),

0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1
1	1	1	0	1	0	0	0	0	1
1	0	1	1	0	1	0	1	0	0
1	0	0	0	1	1	1	0	0	0
0	1	1	0	0	1	0	0	1	0
0	0	1	0	1	0	1	1	1	0
1	1	0	1	0	0	1	0	1	0
0	0	0	1	1	1	0	0	1	1
0	0	1	1	0	0	1	0	0	1
0	1	0	1	1	0	0	1	0	0
1	0	0	0	0	0	0	1	1	1
0	1	0	0	0	1	1	1	0	1

0 = effect off
1 = effect on



13 tests for **all 3-way combinations**
2^{10} = **1,024** tests for **all combinations**

Problem formulation for combinatorial testing: for fixed t, v & k costruct smallest t-way covering array

t-way covering array: for every t parameters all value combinations must appear at least once in covering array, k is number of variables, v is number of possible variables each variable can take, generating minimum is NP-complete

Mathematical approach: can yield smallest possible covering arrays (orthogonal array)

Computational approach: can be applied to any types of covering arrays, but consume more time (random, greedy, search based approaches)

In-Parameter-Order: test generation strategy for combinatorial testing, first generate pairwise set for first 2 parameters, then for first three, so on, pairwise set for first n parameters built by extending test set for first $n-1$ parameters

Horizontal growth: extend each existing test case by adding one value of new parameter

Vertical growth: adds new tests if necessary

The IPO Algorithm: Pairwise Coverage Example

- Consider a system with the following parameters and values:
  - parameter A has values A1 and A2
  - parameter B has values B1 and B2, and
  - parameter C has values C1, C2, and C3

The IPO Algorithm: Pairwise Coverage Example

A	B
A1	B1
A1	B2
A2	B1
A2	B2

A	B	C
A1	B1	C1
A1	B2	C2
A2	B1	C3
A2	B2	C1

Horizontal Growth

A	B	C
A1	B1	C1
A1	B2	C2
A2	B1	C3
A2	B2	C1
A2	B1	C2
A1	B2	C3

Vertical Growth

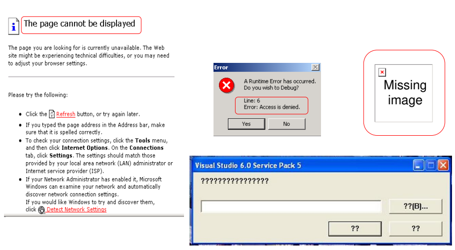
Non-functional testing:
Performance: how system performs in terms of responsiveness & stability under particular workload, evaluates system performance under normal & heavy usage, considers scalability & load (test ability to handle real-world volumes typically by generating many user access simulations)
Stress: test reliability under unexpected or rare workloads, consists of subjecting the system to varying & max loads to evaluate resulting performance, can be automated
Security: concerned mainly with security-related aspects of software, primary concern when communicating & conducting business especially sensitive & business critical transactions over Internet, regardless whether app requires user to enter password for access still must check for internet threats
Usability: concerned mainly with the use of the software, assess user friendliness & suitability by gathering info about how users interact with site, study what user actually does

Stress testing tool report: number of requests, transactions, KBps, round trip time (from user making request to receiving result), number of concurrent connection, performance degradation, types of visitors to site & number, CPU & memory use of app server

Top 2 Web App Security Risks:
Injection: SQL, OS,LDAP, occur whe untrusted data sent to interpreter as part of command/query
Cross Site Scripting: occur whenever app takes untrusted data & send to web browser without proper

validation & escaping
Injection Protection: validate input (careful with special characters, whitelist, validate length, type, syntax), avoid use of interpreter (use stored procedures), otherwise use safe APIs (strongly typed parameterised queries, such as PreparedStatement), use Object Relational Manager
XSS Protection: appropriate encoding of all output data (HTML/XML depending on output mechanism, encode all characters other than very limited subset, specify character encoding)
Usability Testing Steps: identify website purpose, identify intended users, define tests & conduct usability testing, analyze acquired info

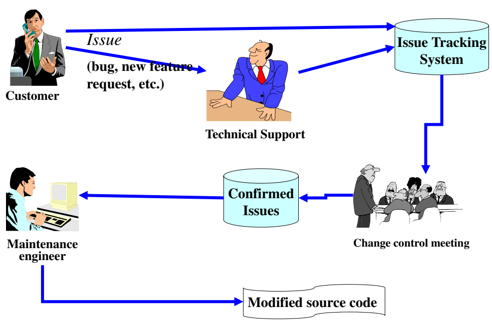
Usability Examples



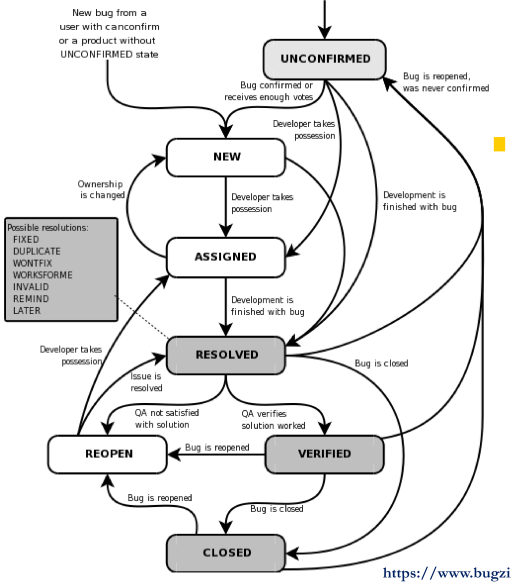
44

Compatibility Testing: ensures product functionality & reliability on supported browsers & platforms that exist on customer computer
Week 8
Test management: manage test plans & cases, track requirements & defects, execute tests, measure progress
Test Report Contents:
Test objective: identifying objectives of testing, should be planned so all requirements individually tested, state exit criteria
Test environment: description of test tools, required packages, hardware & software environment
Modules (features) under test & test cases: software modules to be tested & corresponding test cases, expected & actual test results
Testing schedule: overall test schedule & resource allocation
Test Summary: summary & analysis of test results, test coverage report

A Typical Maintenance Flow



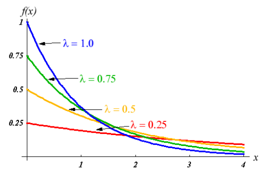
Issue Tracking System: software tool designed to help devs keep track of reported software issues (including bugs & new feature requests), extremely valuable in software development, used extensively by companies developing software products
Issue Tracking System Uses: track issues/bugs, communicate with teammates, submit & review patches, manage quality assurance
Bug Data:
Description: summary, product/module it belongs to, detailed descriptions
Role: reporter/submitter, assignee, QA contact
Status: current status, resolution, priority, severity
Time: open date, changed date, closed date, ...



Issue Tracking System Design: database, business logic, client
Useful bug reports: ones that get bugs fixed, reproducible, specific
Regression Testing: run old test cases on new version, establishing policy for regular regression testing is key for achieving successful,reliable & predictable software development projects
Regression Test Selection: speed up regression testing by only rerunning tests affected by code changes
Regression Test Prioritisation: rank all test cases (to discover bugs sooner or achieve higher coverage sooner), run test cases according to ranked sequence, stop when all resources used up
Reliability: probability that system/capability of system functions without failure for specified time in specified environment, ex: reliability of 0.92 for 8 hours means when executed for that long would operate without failure for 92 out of 100 periods
Single failure specification: what is probability of failure of a system/component
Multiple failure specification: if system/component fails at time $t_1, t_2, \ldots, t_{i-1}$m what is probability of failure at time t_i
Reliability formulas: $R(t) = P(T > t), F(t) = P(T \leq t), t \geq 0$

Probability Density Function (PDF)

- **Probability density function (PDF):** depicting changes of the probability of failure up to a given time *t*



- A common form of PDF is exponential distribution

Exponential PDF:

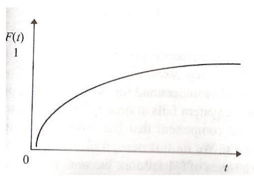
$$f(t) = \begin{cases} \lambda e^{-\lambda t} & t \geq 0 \\ 0 & t < 0 \end{cases}$$

41

Cumulative Density Function (CDF)

- **Cumulative density function (CDF):** depicting cumulative failures up to a given time *t*

$$F(t) = \int_0^t f(t)dt$$



- For exponential distribution, CDF is:

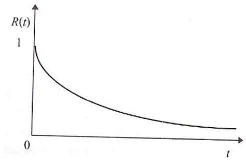
$$F(t) = \begin{cases} 1 - e^{-\lambda t} & t \geq 0 \\ 0 & t < 0 \end{cases}$$

42

Reliability Function (R)

- **Reliability function (R):** depicting a component functioning without failure until time *t*, also called (survival function)

$$R(t) = 1 - F(t)$$



- For exponential distribution, R is:

$$R(t) = e^{-\lambda t}$$

43

Reliability: Example

- Assume the failures of a computer system follows exponential distribution, and the probability of failure is 5% within 100 hours. What's the probability of the system working without failures in 1000 hours? (Hint: $ln(0.95) = -0.05$; $ln(0.61) = -0.5$).

$$P(t \leq 100) = F(100) = 0.05$$
$$1 - e^{-100\lambda} = 0.05$$
$$\lambda = 0.0005$$
$$R(1000) = e^{-1000 \times 0.0005} = 0.6065$$

44

Reliability Distributions: exponential, Poisson, lognormal, normal, Pareto, Weibull

Software Reliability Models

- Goel-Okumoto (G-O) model

$$m(t) = a(1 - \exp[-bt]), \ a > 0, b > 0,$$

- Gompertz growth curve model

$$m(t) = ak^{b^t}, \ a > 0, 0 < b < 1, 0 < k < 1,$$

- Logistic growth curve model

$$m(t) = \frac{a}{1 + k\exp[-bt]}, \ a > 0, b > 0, k > 0,$$

- Yamada delayed S-shaped model

$$m(t) = a(1 - (1 + bt)\exp[-bt]), \ a > 0, b > 0,$$

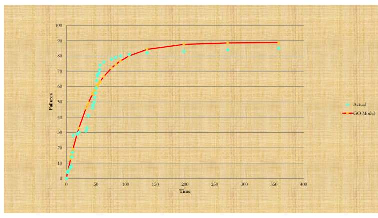
46

Goel-Okumoto (G-O) Model /1

- Proposed by Amrit Goel of Syracuse University and Kazu Okumoto in 1979.
- Based on NHPP(Non-Homogeneous Poisson Process)
- An example: the number of failures (y) of a software system at time *t* :

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$t(i)$	0	2	3	7	8	9	10	11	18	21	33	35	37	44	45	47	48
$y(i)$	0	4	5	7	8	14	17	28	29	30	31	33	41	46	48	50	53
i	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
$t(i)$	49	50	51	52	53	55	56	57	63	76	83	91	106	136	198	272	358
$y(i)$	56	59	64	67	68	69	71	74	76	78	79	80	81	82	83	84	85

Goel-Okumoto (G-O) Model /2



a = 88.733, b= 0.022

G-O model: #Failures = 88.733*(1-exp(-0.022t))

48

Mean time to failure: mean of probability density, expected value of T, average lifetime of system, $E(T) = \int_0^\infty t\, f(t)dt = \int_0^\infty R(t)dt$, for exponential is $\frac{1}{\lambda}$
Mean time between failures: $MTTF + MTTR$ (mean time to repair)
Software reliability tools tasks: collecting failure & test time info, calculating estimates of model parameters using this onfo, testing to fit model against collected info, selecting model to make predictions of remaining faults, time to test, apply model
Week 9
Software reviews: quality improvement processes for

written material, by detecting defects early & preventing leakage downstream higher cost of later detection & rework eliminated

Software products that can be reviewed: requirements specifications, design descriptions, source code (code review), release notes

Code review types: ad-hoc review, pass-round, walkthrough, group review, formal inspection

Formal Inspection: planning/overview, preparation (product docs, rules/checklist), inspection, rework

Code review steps: perform examination of software products, detect defects (bugs), violation of coding standards, code smells, other problems, look for code patterns that indicate problems based on prior xp, static analysis tools can also help

Bug patterns: infinite recursion, null pointer bugs, SQL injection, divide by 0, buffer overflow, memory leak, deadlock, infinite loop, XSS

Code smells: indications of poor coding & design choices that can cause problems during later phase of development, hint something gone wrong somewhere

A List of Code Bad Smells

- Duplicated Code
- Long Method
- Large Class (Too many responsibilities)
- Long Parameter List (Object is missing)
- Feature Envy (Method needing too much information from another object)
- Lazy Class (Do not do too much)
- Middle Man (Class with too much delegating methods)
- Temporary Field (Attributes only used partially under certain circumstances)
- Message Chains (Coupled classes, internal representation dependencies)
- Data Classes (Only accessors)
- ...

Checking Coding Conventions /1

- Your code should be readable!
- Formatting conventions ensure *consistency* and therefore, familiarity for readers
- Indentation: Indent when starting out new blocks of code

Sun's Coding Conventions for Java
–*http://www.oracle.com/technetwork/java/codeconvtoc-136057.html*

JavaScript Coding Style
–https://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml

Code Conventions for the JavaServerPages Technology
–http://java.sun.com/developer/technicalArticles/javaserverpages/code_convention/

Checking Coding Conventions /2

Checking identifier names:

- Choosing meaningful names
- Class names start upper-cased
  – e.g., BankAccount, Vehicle, VehicleApplet
- Variable and Method names *start*lower-cased
  - e.g., aliceAccount, hondaCivic, nCount, etc.
- Constants are ALL_CAPS and words in name are separated by an underscore
  – e.g., PI, MAX_WIDTH, DEFAULT_WIDTH

Checking Coding Conventions /3

Checking comments

• Block-style comments
/*
* This is a multi-line comment.
* Use when you need to write a long comment about a fragment
*/

• One-line comments
–/* C-style comments */
• use to put descriptive notes *before* a code fragment

–// C++ style comment
- use at the end of the line, to describe variables or short pieces of code
- also use for commenting-out code

• javadoc comments
– like block-style, but starts with /** instead
– use immediately before classes, methods, and fields

Code Review benefits: can find 60–100% of defects, can assess/improve quality of work product, software development process & review process itself, reduce total project cost but have non-trivial cost (15%), early defect removal is 10–100 times cheaper, reviews distribute domain knowledge, dev skills, corporate culture

Common problems in code review: insufficient preparation, moderator domination, incorrect review rate, ego involvement & personality conflict, issue resolution & meeting digression,recording difficulties & clerical overhead

Static Analysis: analyse program without executing, doesn't depend on test cases, generally doesn't know what the software is supposed to do, looks for bug patterns, no replacement for testing, many defects can't be found with static analysis

Patterns to be checked: bad practice, correctness, performance, dodgy code, vulnerability to malicious code

Pattern examples: equals method should not assume type of object argument, collection should not contain themselves ($!s.contains(s)$), should not use $String.toString()$