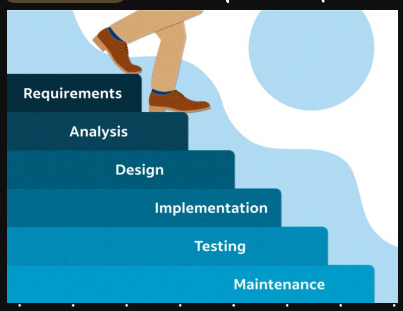
INTRO TO TESTING

waterfall development process:



- failure: observable incorrect behaviour of program
  - ↳ related to behaviour, not code
- fault (bug): necessary condition for occurrence of failure
  - ↳ related to code
- error: mistake usually made by ppl

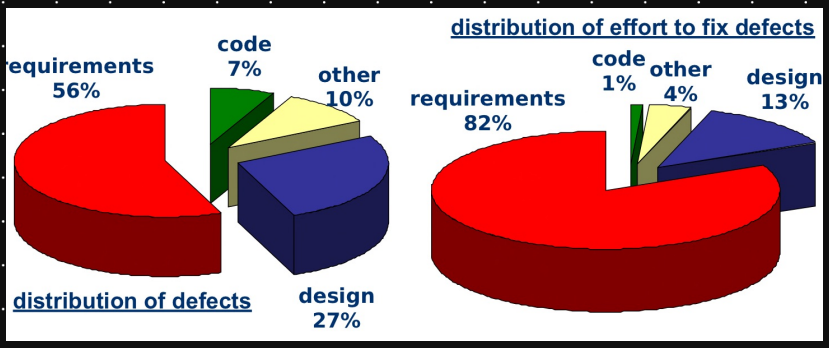
error —(can lead to)→ fault —(can lead to)→ failure

e.g.

```
1. int double (int param) {
2.     int result;
3.     result = param * param;
4.     return result;
5. }
```

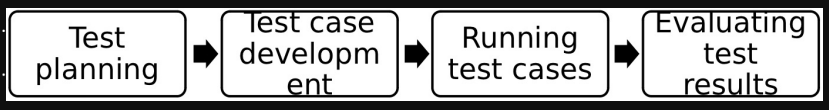
A call to double(3) returns 9
- Result 9 represents a **failure**
- Such failure is due to the **fault** at line 3
- The **error** can be a typo (hopefully)

- 9 causes of software errors:
  1) faulty requirements defn
     - ↳ e.g. incorrect, missing, incomplete, unnecessary
  2) client-developer communication failures
  3) deliberate deviations from software requirements
     - ↳ e.g. improper reuse, gold plating (i.e. adding unnecessary features)
  4) logical design errors
     - ↳ e.g. problems w/ algos, missing states, handling illegal inpur
  5) coding errors
  6) non-compliance w/ documentation r coding ins
  7) shortcomings of testing process
  8) procedure errors
     - ↳ e.g. workflow + UI errors
  9) documentation errors
- majority of defects are introed in earlier phases
  - ↳ requirements are top reasons for proj success/failure



TESTING TYPES

- **test** is act of exercising software w/ test cases to find failures or demo correct execution
- **test case** specifies set of inputs + expected outputs for program behaviour
- **test process**:



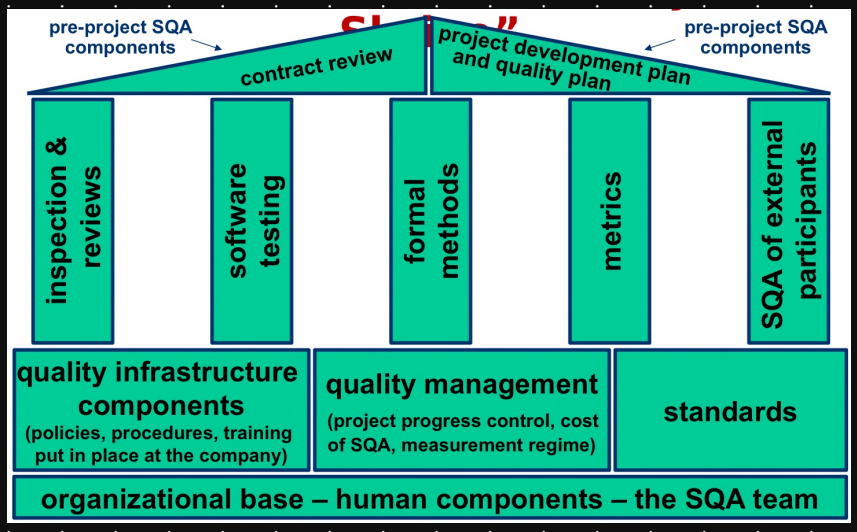
- lvls of software testing:
  - ↳ **unit testing**: individual units tested in isolation
    - determine if each unit fcns as designed
  - ↳ **integration testing**: test group of related units tgt
    - find interface issues btwn units
    - e.g. db access
  - ↳ **system testing**: test complete software system + eval system's compliance w/ specifications
- **black-box testing**: based on specs + covers as much specified behaviour as possible
  - ↳ can't reveal errors from implementation details
  - ↳ e.g. system
- **white-box testing**: based on code + covers as much coded behaviour as possible
  - ↳ can't reveal errors due to missing paths
  - ↳ e.g. unit, integration

SOFTWARE QUALITY

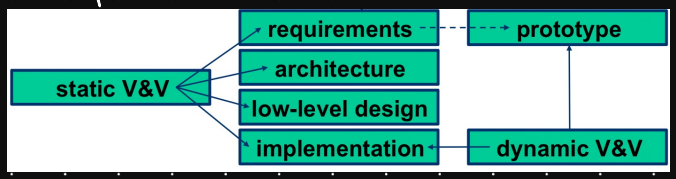
- **software quality** is conformance to requirements
  - ↳ lack of bugs
    - low **defect rate** ($\frac{\#\ \text{defects}}{\text{size unit}}$)
  - ↳ high **reliability** (# failures / n hrs of op)
    - measured as **mean time to failure (MTTF)**, which is probability of failure-free op in specified time
- software **quality assurance** (QA) comprises of verification (building it right) + validation (building right thing)
- **software quality factors**:

- ↳ correctness: accuracy + completeness of required output, up-to-dateness + availability of info
- ↳ reliability: max failure rate
- ↳ efficiency: hardware resources needed to perform software fcns
  - e.g. hardware processing capabilities, data storage, bandwidth, power usage
- ↳ integrity: system security + access rights
- ↳ usability: required training + ability to perform tasks
- ↳ maintainability: effort needed to find + fix failures
- ↳ flexibility: deg of adaptability
  - e.g. if 1 version of software has bug, how efficient is it to fix it for all versions
- ↳ testability: testing support
  - e.g. log files + auto diagnostics
- ↳ portability: adaption to other envs
- ↳ reusability: use of components for other projs
- ↳ interoperability: ability to interface w/ other components/systems

- software QA is quality control (i.e. set of activities designed to evaluate quality of product) + quality assurance (i.e. aims to minimize cost of guaranteeing quality)
- objectives of SQA:
  - ↳ software must conform to fcnal technical requirements
  - ↳ software must conform to scheduling + budgetary requirements
  - ↳ initiation + management of activities for improvement + greater efficiency of software dev, maintenance, + QA
- 3 general principles of SQA:
  - ↳ know what we are doing
    - understand what's being built, how it's built, + current state
  - ↳ know what we should be doing
    - explicit requirements + specs
  - ↳ know how to measure diff by having explicit measures for comparison
    - 4 methods:
      1) formal methods: verify mathematical properties
      2) testing: explicit input + check for expected output
      3) inspections: human examination of requirements, design, + code
      4) metrics: measure known set of properties related to quality
- SQA includes V+V
  - ↳ verification: performed at end of phase to ensure requirements of prev phase have been met
  - ↳ validation: performed at end of dev process to ensure compliance w/ user expectations
- software quality shrine:



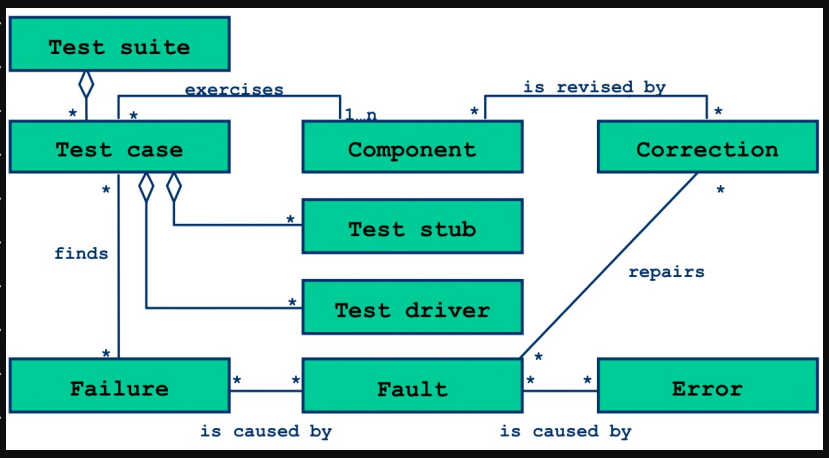
- SQA includes defect prevention, detection, + removal
- SQA process:



- key SQA capabilities:
  - ↳ inspections: systematic, structured reviews of software docs
  - ↳ monitor + control quality across all proj phases + activities
  - ↳ derive effective test cases to find faults
  - ↳ automate testing + inspection
- use CI/CD practice
  - ↳ continuous integration: continuous rebuilds of source code every time code is committed
    - often includes deployment, installation, + testing of apps in prod env
  - ↳ continuous deployment: changes are auto deployed to prod w/o manual intervention
  - ↳ continuous delivery: software can be released to prod at any time w/ as much automation as possible

TESTING BASICS

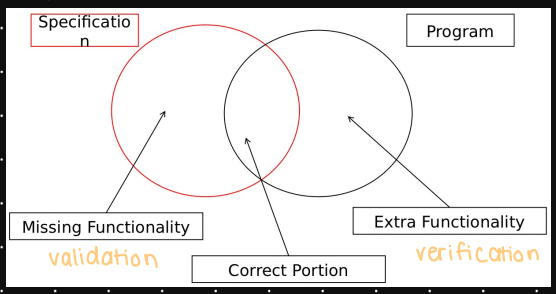
- software testing: exercise software w/ test cases to gain / reduce confidence in system
- test cases: set of inputs + list of expected outputs
- test stub: partial implementation of component that tested component depends on
- test driver: partial implementation of component that exercises + depends on test component
- summary of defns:



test case contains:
- ↳ boilerplate: author, date, summary, test case ID, ref to spec, version
- ↳ pre-conditions (including env)
- ↳ inputs
- ↳ expected outputs
- ↳ observed outputs
- ↳ pass/fail

3 conditions for failure to be observed (RIP model)
- ↳ reachability: location(s) that contain fault must be reached
- ↳ infection: state of program must be incorrect after executing
- ↳ propagation: infected state must propagate to cause some output to be incorrect

program behaviour



total correctness is impossible to demo b/c test can nvr reveal absence of fault
- ↳ better viewpoint is testing if program P is correct wrt spec S

TESTING TECHNIQUES

black-box:
- \+ checks conformance w/ specs
- \+ scales up (i.e. diff techniques at diff granularity lvls)
- \- depends on spec notation + deg of detail
- \- doesn't tell us how much of system is being tested
- \- software might perform additional undesired task

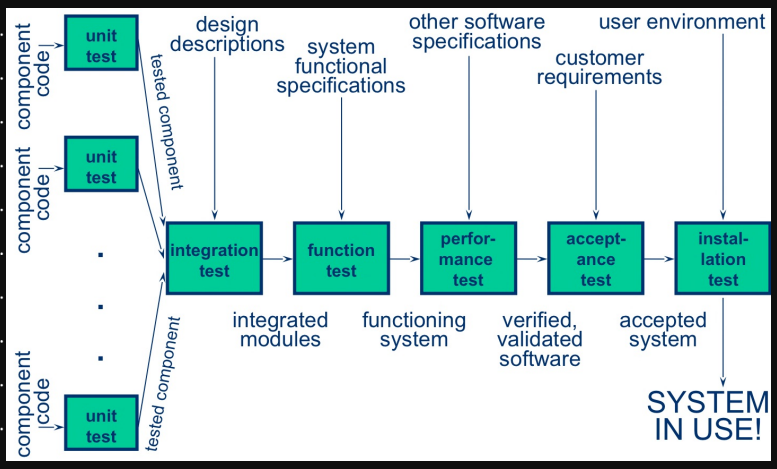
white-box:

+ confidence abt test coverage
+ based on control / data flow coverage
- doesn't scale up
  ↳ mostly applicable for unit + integration
- can't reveal missing fcnalities

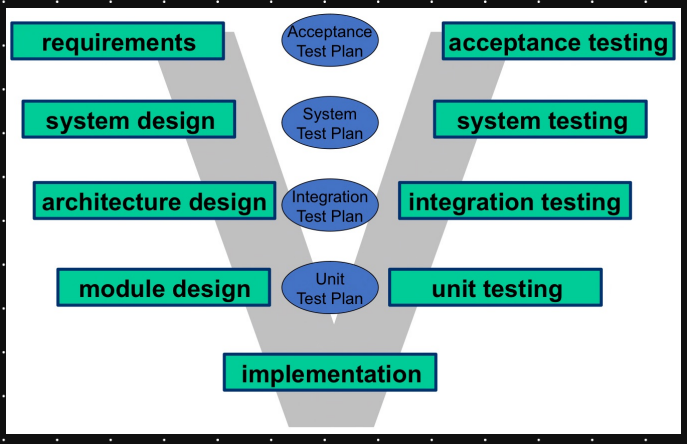
diffs among testing activities:

Unit testing	Integration testing	System testing
↳ from module specs	↳ from interface specs	↳ from requirements specs
↳ visibility of code details	↳ visibility of integral structure	↳ no visibility of code
↳ complex scaffolding	↳ some scaffolding	↳ no drivers / stubs
↳ behaviour of single module	↳ interactions among modules	↳ system fcnalities

types of testing:



V-model of dev:



integration of well-tested components may lead to failure b/c:
↳ bad use of interfaces
↳ wrong hypothesis on behaviour / state of related modules
↳ use of poor (too simple) drivers / stubs

according to Feather, unit test runs fast + helps localize problems

TESTING MATURITY LEVELS

- 4 lvls according to Beizer
- lvl 0: no diff btwn testing + debugging
  - ↳ no distinguishing btwn incorrect program behaviour + mistakes
- lvl 1: purpose of testing is to show software works
  - ↳ correctness is impossible to achieve / demo
  - ↳ test engineers have no strict goal, stopping rule, + formal technique
  - ↳ no way to quantitively express / eval work
- lvl 2: purpose is to show software doesn't work
  - ↳ adversarial rltnship btwn testers + developers
  - ↳ unsure conclusion when no failures found
- lvl 3: purpose is to reduce risk of using software
  - ↳ testing can show presence, but not absence, of failures
  - ↳ united team goal
- lvl 4: testing is mental discipline that helps all IT professionals develop higher quality software
  - ↳ testers are technical leaders
  - ↳ primary responsibility is measuring + improving software quality
  - ↳ testers train developers

DIFFICULTY OF TESTING

- impossible to test program under all operating conds
- to choose test cases (b/c not all are significant):
  - ↳ use only 1 from those of the same class (i.e. same result from test cases)
  - ↳ to choose class reps, pick those most likely to fail
- e.g. triangle

■ Input: Three integers, a, b, c, the lengths of the side of a triangle
■ Output: Scalene, isosceles, equilateral, invalid

  - ↳ test case classes:
    - Valid scalene, isosceles, equilateral triangle
    - All permutations of two equal sides
    - Zero or negative lengths
    - All permutations of a + b < c
    - All permutations of a + b = c
    - All permutations of a = b and a + b = c
    - MAXINT values
    - Non-integer inputs
  - ↳ example implementation:

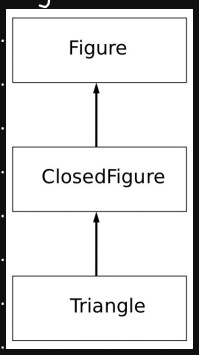
```
class Triangle{
  public Triangle(LineSegment a, LineSegment b,
                                  LineSegment c)
  public boolean is_isosceles()
  public boolean is_scalene()
  public boolean is_equilateral()
  public void draw()
  public void erase()
}
class LineSegment {
  public LineSegment(int x1, int y1,
                     int x2, int y2)
}
```

↳ extra tests based on implementation:

- Is the constructor correct?
- Is only one of the **is_*** methods true in every case?
- Do results repeat, e.g., when running **is_scalene** twice or more, do they have the same results?
- Results change after **draw** or **erase**?
- Segments that do not intersect?

↳ if Triangle has superclasses, we must also consider **inheritance tests**

- e.g. tests that apply to all Figure + ClosedFigure objs must still work for Triangle objs



complete testing means that we know there's no remaining unknown bugs

coverage: extent of testing of certain attrs of program (e.g. stmt/branch/condition coverage) or extent of testing completed compared to population of possible tests

↳ complete coverage is impossible b/c domain of possible inputs too large + too many possible paths thru program

↳ good measurement tool to show how far we're from complete testing, but not good to show how close

enormous #s of possible tests:

↳ **valid** inputs

↳ **invalid** inputs

- error handling must be correctly triggered

↳ **edited** input

- any char can be changed into any other

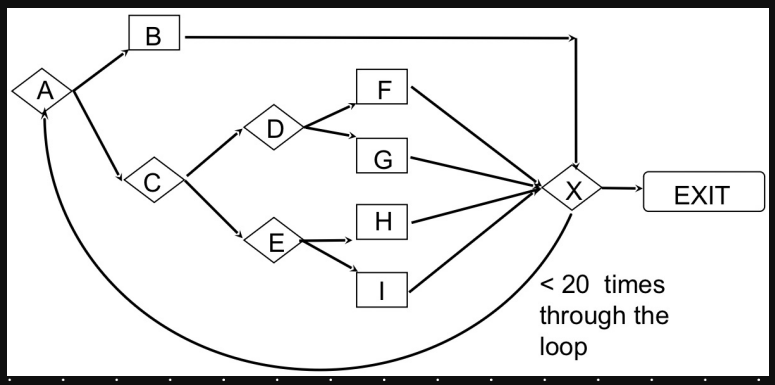
- repeated editing

↳ input timing variations

- race conds btwn events often lead to bugs

↳ combo testing

e.g. too many paths in simple program



↳ 5 ways to get to X, then to EXIT in 1 pass

- e.g. ABX-EXIT

↳ 5 ways to get to X 1st time + 5 ways to get to X 2nd time so there's $5(5)=25$ cases

- e.g. ABXACDFX-EXIT

↳ $5+5^2+\ldots+5^{20} = 10^{14} \approx 100$ trillion paths

testing can't verify requirements

bugs in test design / drivers are hard to find

expected output may be difficult to determine

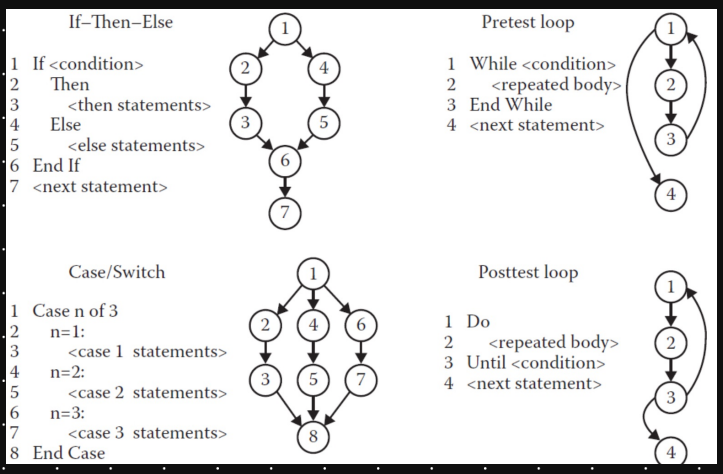
CONTROL FLOW AND COVERAGE

CONTROL FLOW GRAPHS

program graph: directed graph where nodes are stmt fragments + edges rep flow of control

↳ complete stmt is also stmt fragment

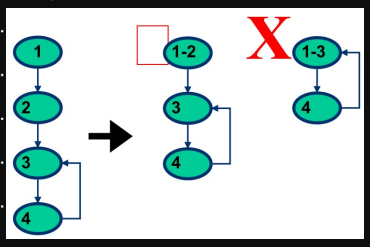
program graphs for 4 structured programming constructs:



control flow graph (CFG) models all executions of method by describing control structs

↳ **nodes**: stmts or seqs of stmts

- **basic blocks**: seq of stmts st if 1st is executed, all of them will be (i.e. no branches)
  - → intermediate stmts aren't shown as long as there's only 1 exiting + 1 entering edge
  - → e.g.

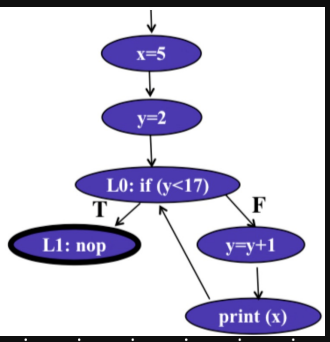


↳ **edges**: transfers of control

- e.g. (s1, s2) means s1 may be followed by s2 in execution

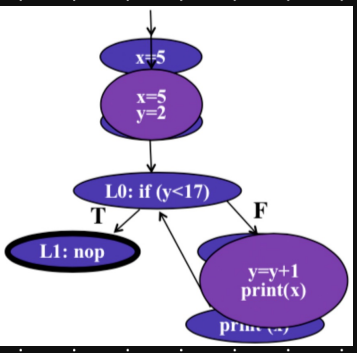
e.g.

```
    x=5
    y=2
L0: if (y< 17) goto L1
    y=y+1
    print (x)
    goto L0
L1: nop
```



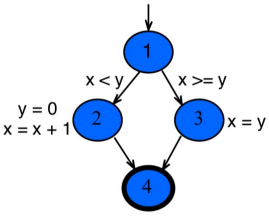
↳ w/ basic blocks:

```
    x=5
    y=2
L0: if (y< 17) goto L1
    y=y+1
    print (x)
    goto L0
L1: nop
```

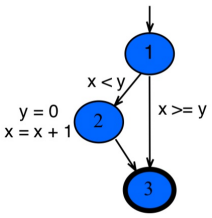


e.g. if stmt

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



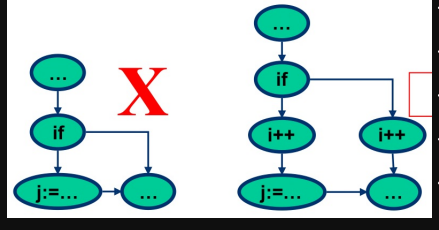

```
if (x < y)
{
  y = 0;
  x = x + 1;
}
```



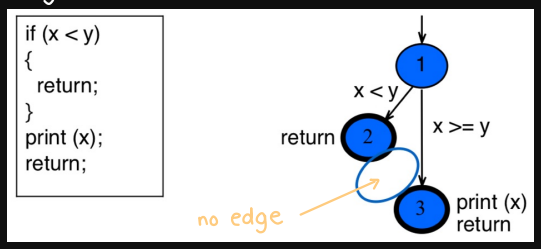
in CFG, nodes corresponding to branching can't contain asgnts b/c we need to identify data flow info

↳ e.g.

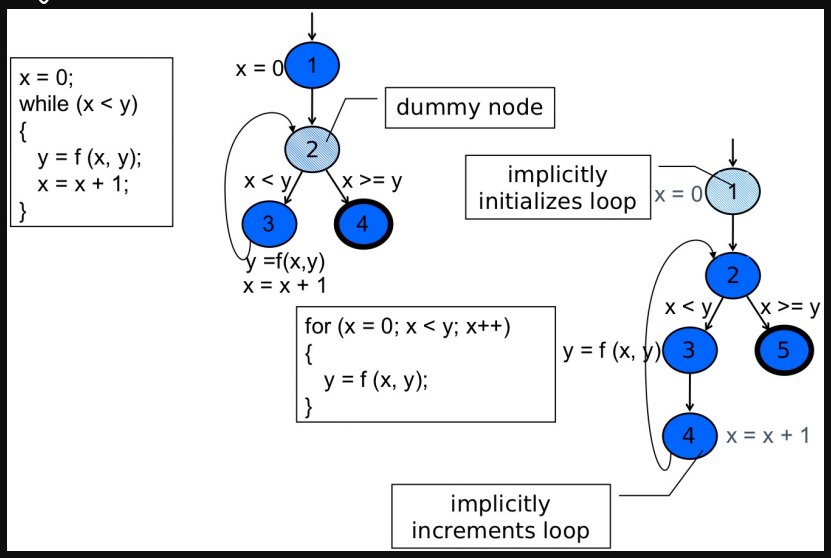
```
...
if (i++==1) {
   j := ...
}
...
```



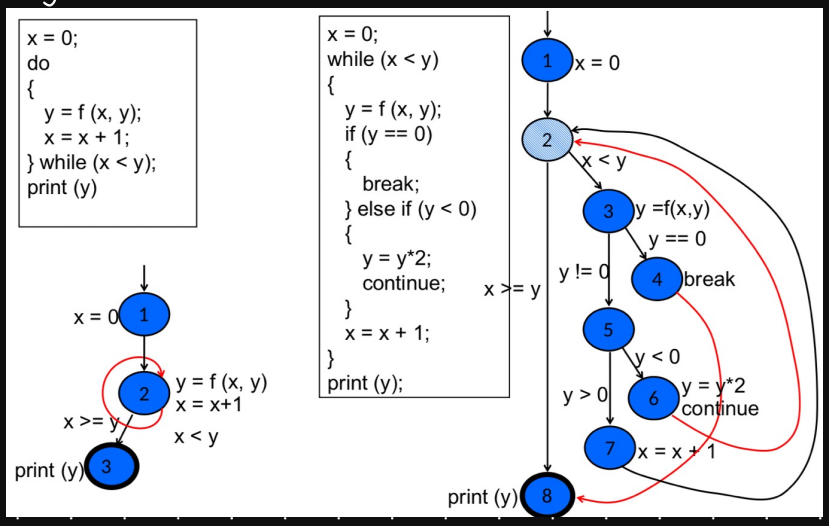
e.g. if - return stmt



e.g. while & for loops

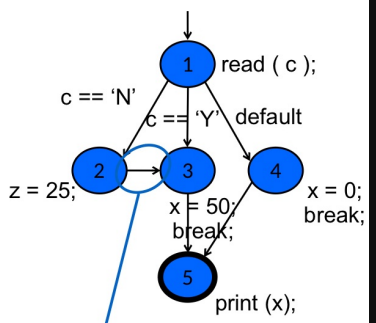


e.g. do loop, break & continue



e.g. case /switch

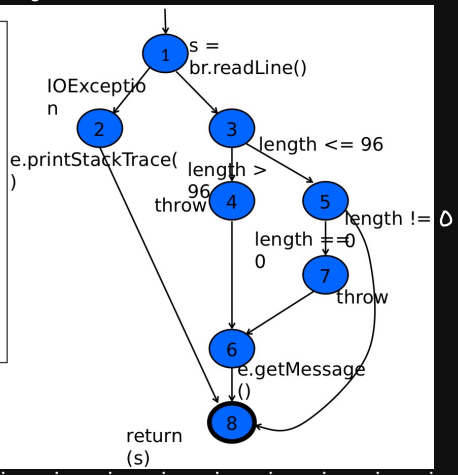
```
read ( c) ;
switch ( c )
{
   case 'N':
      z = 25;
   case 'Y':
      x = 50;
      break;
   default:
      x = 0;
      break;
}
print (x);
```



Cases without breaks fall through to the next case

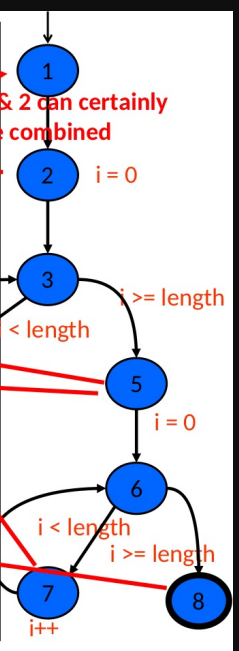
e.g. exceptions (try-catch)

```
try
{
   s = br.readLine();
   if (s.length() > 96)
      throw new Exception
         ("too long");
   if (s.length() == 0)
      throw new Exception
         ("too short");
} (catch IOException e) {
   e.printStackTrace();
} (catch Exception e) {
   e.getMessage();
}
return (s);
```



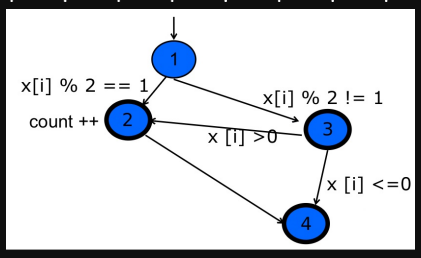
e.g.

```
public static void computeStats (int [ ] numbers)
{
   int length = numbers.length;
   double med, var, sd, mean, sum, varsum;

   sum = 0;
   for (int i = 0; i < length; i++)
   {
      sum += numbers [ i ];
   }
   med   = numbers [ length / 2];
   mean = sum / (double) length;

   varsum = 0;
   for (int i = 0; i < length; i++)
   {
      varsum = varsum  + ((numbers [ I ] - mean) * (numbers [ I ] - mean));
   }
   var = varsum / ( length - 1.0 );
   sd  = Math.sqrt ( var );

   System.out.println ("length:                " + length);
   System.out.println ("mean:                  " + mean);
   System.out.println ("median:                " + med);
   System.out.println ("variance:              " + var);
   System.out.println ("standard deviation:    " + sd);
}
```

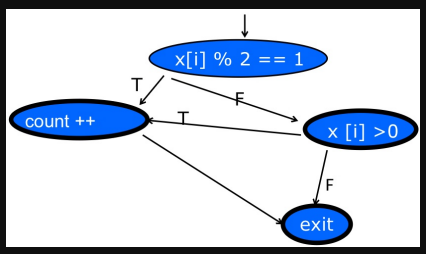


e.g. short circuit

```
if (x[i] % 2 == 1 || x [i] >0 ) {
count++; }
```

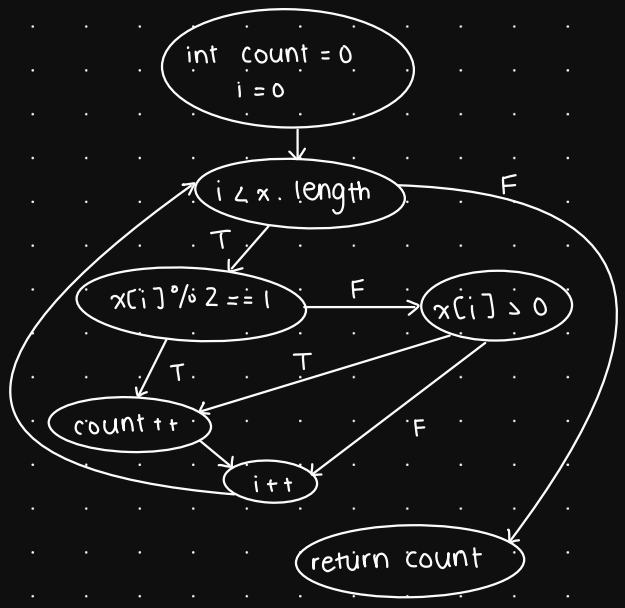


or



e.g.

```
public static int oddOrPos(int[] x)
{
//Effects: if x==null throw NullPointerException
// else return the number of elements in x that
//      are either odd or positive (or both)
   int count = 0;
   for (int i = 0; i < x.length; i++)
   {
      if (x[i]%2 == 1 || x[i] > 0)
      {
         count++;
      }
   }
   return count;
}
   // test:  x=[-3, -2, 0, 1, 4]
   //        Expected = 3
```



CODE COVERAGE

code coverage models:

- ↳ stmt
- ↳ segment (basic block)
- ↳ branch
- ↳ condition
- ↳ condition / decision
- ↳ modified condition / decision

stmt coverage is achieved when all stmts in method have been executed at least once

- ↳ equiv to covering all nodes in CFG
- ↳ executing stmt is weak guarantee of correctness

$$\text{stmt cov} = \frac{\#\text{ executed stmts}}{\#\text{ total stmts}}$$

- ↳ e.g.

```
1  public void printSum(int a, int b) {
2    int result = a + b;
3    if (result > 0)
4        System.out.println("red", result);
5    else if (result < 0)
6        System.out.println("blue", result);
7  }
```

TC1	TC2
• a = 3 • b = 9 Coverage (TC1) = 5/7 = 71%	• a = -5 • b = -8 Coverage (TC1+TC2)= 100%

- stmt cov is most used in industry
  - ↳ aim for 80% - 90%
- problems w/ stmt cov:
  - ↳ predicate may be tested for only 1 val
  - ↳ loop bodies may be iterated only once
  - ↳ can be achieved w/o branch cov + important cases missed
  - ↳ e.g.

```
1  public void printSum(int a, int b) {
2    int result = a + b;
3    if (result > 0)
4        System.out.println("red", result);
5    else if (result < 0)
6        System.out.println("blue", result);
     [else do nothing]
7  }
```

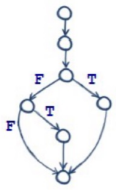
TC1	TC2
• a = 3 • b = 9	• a = -5 • b = -8

Never in here for the above two test cases!

- segment (basic block) coverage counts segments instead of stmts
  - ↳ may prod diff. #s than stmt
    - e.g. if segment P has 1 stmt + Q has 9 stmts, exercising 1 of the segments will give 10% or 90% stmt cov but 50% segment cov in both cases

- branch coverage is achieved when every branch from a node is executed at least once
  - ↳ at least 1 T + F eval for each pred
  - ↳ achieved w/ D + 1 paths in CFG w/ D 2-way branching nodes + no loops
- branch cov = $\frac{\#\text{ executed branches}}{\#\text{ total branches}}$
  - ↳ e.g.

```
public void printSum(int a, int b) {
  int result = a + b;
  if (result > 0)
      System.out.println("red", result);
  else if (result < 0) {
      System.out.println("blue", result);
  [else do nothing]
}
```



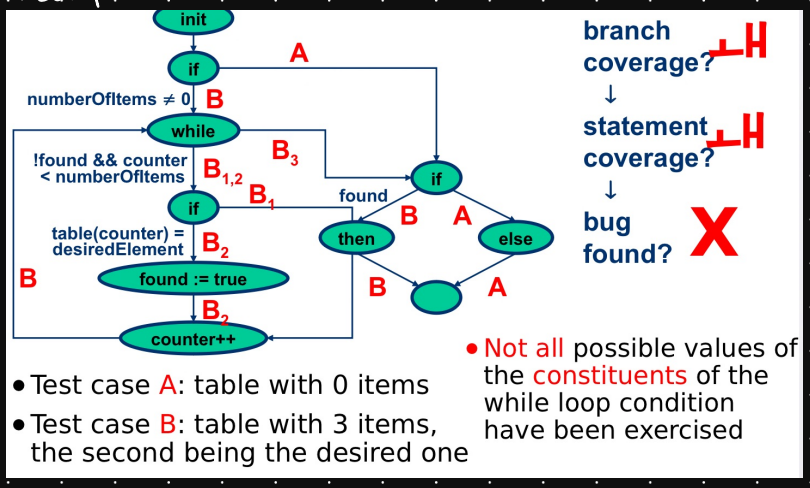
TC1	TC2	TC3
• a = 3 • b = 9 Coverage = 1/4 = 25%	• a = -5 • b = -8 Coverage = 3/4 = 75%	• a = -5 • b = 5 Coverage = 100%

- problems w/ branch cov:
  - ↳ short-circuit eval means that many predicates might not be evaluated
  - ↳ compound pred is treated as single stmt
    - 2^n combos for n clauses, but only 2 tested
  - ↳ only subset of all entry-exit paths is tested
  - ↳ e g.

```
counter := 1;
found := false;
if numberOfItems ≠ 0 then
      while (not found) and counter < numberOfItems loop
            if table(counter) = desiredElement then
                  found := true;
            end if;
            counter := counter + 1;
      end loop;
end if;
if found then write ("desired element exists in table");
else write ("desired element does not exist in table");
end if;
```

≤

- bug: if desired elmt is last elmt, exit loop before found
- incomplete cov:



- Test case A: table with 0 items
- Test case B: table with 3 items, the second being the desired one
- Not all possible values of the constituents of the while loop condition have been exercised

condition coverage reports T/F outcome of each cond
↳ measures conds independently of each other

cond cov = $\frac{\#\text{ conds that are both T \& F}}{\#\text{ total conds}}$

↳ e.g.

```
public void printResults(int a, int b) {
  if ((a == 0) || (b > 0))
      System.out.println("red", b/a;
  else
      System.out.println("blue", b + 2);
  System.out.println("end");
}
```

TC1: (a = 0, b = -5)
TC2: (a = 5, b = 5)
Branch coverage = 50%
Condition coverage = 100%

condition/decision coverage is computed by considering both branch + cond cov measures
↳ aka branch + cond cov
↳ e.g. 100% cov

```
public void printResults(int a, int b)
  if ((a == 0) || (b > 0))
      System.out.println("red", b/a);
  else
      System.out.println("blue", b + 2);
  System.out.println("end");
}
```

TC1: (a = 0, b = -5)
TC2: (a = 5, b = 5)
TC3: (a = 3, b = -2)

DATA FLOW

MODIFIED CONDITION / DECISION COVERAGE

modified condition / decision cov (MC/DC) : test important combos of conds to limit testing costs

↳ each cond should eval to T once, F once, & affect decision's outcome

↳ e.g.

a && b && c

Test Case	a	b	c	Outcome
1	T	T	T	T
2	T	T	F	F
3	T	F	T	F
~~4~~	~~T~~	~~F~~	~~F~~	~~F~~
5	F	T	T	F
~~6~~	~~F~~	~~T~~	~~F~~	~~F~~
~~7~~	~~F~~	~~F~~	~~T~~	~~F~~
~~8~~	~~F~~	~~F~~	~~F~~	~~F~~

T T T
a b c
F
c
F
b

F
a

↳ e.g. (a || b) && c

	a	b	c	outcome
1	T	T	T	T
2	T	T	F	F
3	T	F	T	T
4	T	F	F	F
5	F	T	T	T
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

T
a

T T
b c
F
c

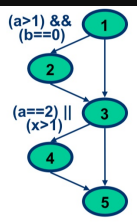
F F
a b

path cov is having a test case for each possible path

↳ in practice, #paths is too large / infinite & some paths are infeasible

↳ e.g.

```
int proc(int a,int b,int x) {
1  if ((a>1) && (b==0))
2     x = x/a;
3  if ((a==2) || (x>1))
4     x = x+1;
5  return x;
}
```



Path	a	b	x	Node 1	Node 3
1-2-3-4-5	2	0	-	TT→T	T-→T
1-2-3-5	3	0	1	TT→T	FF→F
1-3-4-5	1	1	2	FF→F	FT→T
1-3-5	1	1	1	FF→F	FF→F

e.g.

Show that path coverage does not imply condition coverage by changing the if statement below

```
...
1  if (a>0)
2     x = 1;
3  else
4     x = 2;
5  end if;
...
```

```
...
if (a > 0 || a<-1)
    x = 1;
else
    x = 2;
end if;
```

Path coverage:
T1 = {<a=0>, <a=1>}
→ condition coverage . a<-1 is F in both cases

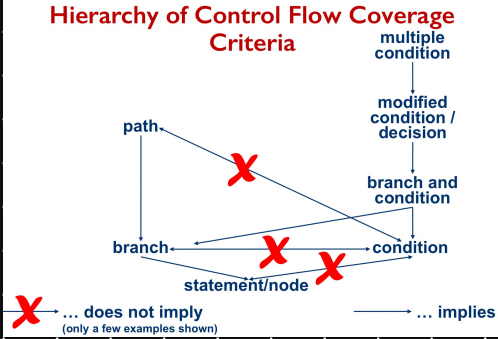
↳ e.g.

Show that condition coverage does not imply path coverage by changing the if statement below

```
...
1  if (a>0)
2     x = 1;
3  else
4     x = 2;
5  end if;
...
```

```
if (a>0 && b>0)
    x = 1;
else
    x = 2;
end if;
```

- cond cov: T1 = {<a=0, b=1>, <a=1, b=0>}
- path cov: only else branch covered



LOOP COVERAGE

- in loops:
  - ↳ decide to traverse it or not
  - ↳ analyze boundary val on index var
  - ↳ nested loops have to be tested separately, starting w/ innermost
- min cov should execute loop body 0 times (don't enter), once (don't repeat), + 2+ times (repeat)
- treat loop as blackbox + carry out testing based on blackbox technique
- in single loop, set loop control var to:
  - ↳ min -1
  - ↳ min
  - ↳ min +1
  - ↳ typical
  - ↳ max -1

↳ max
↳ max + 1

in nested loops, start from innermost loop:
↳ set all outer loops to min vals
↳ set all other loops to typical vals
↳ use single loop test cases for innermost loop
↳ move up in loop lvl
↳ finally, do single loop test cases for all loops in nest simultaneously

DATA FLOW AND COVERAGE

focus on CFG paths that are significant for data flow

analyze occurrences of vars (i.e. where data is defined & used)
↳ defn occurrence: val is bound to var
↳ use occurrence: val of var is referred
- predicate use (p-use): var used to decide whether predicate is T
- computational use (c-use): compute val for defining other vars or output val

↳ e.g.

- Statement `x = y + z` defines variable `x` and uses variables `y` and `z`
- Statement `scanf("%d %d", &x, &y)` defines variables `x` and `y`
- Statement `printf("Output: %d \n", x + y)` uses variables `x` and `y`

e.g. ptrs

```
z = &x; y = z+1; *z = 25; y = *z + 1;
```

- 1st statement: defines a pointer variable z but does not use x
- 2nd statement: defines `y` and uses `z`
- 3rd statement: defines `x` accessed through the pointer variable `z`
- 4th statement: defines `y` and uses `x` accessed through the pointer variable `z`

e.g. arrs

```
int A[10]; A[i] = x + y;
```

- 2nd statement: defines `A` and uses `i`, `x`, and `y`
- Alternate view for 2nd statement: defines `A[i]` – not the entire array `A` (the choice of whether to consider the entire array `A` as defined or the specific element depends upon how stringent the requirement is for coverage analysis)

↳ same reasoning applies to fields of class

e.g.

```
if (A[x+1]>0) {output(x)};
```

↳ A is p-use
↳ x is 1st a c-use, then a p-use in A[x+1]

var defn written as d(v,n): val is assigned to v at node n

var use / ref written as:
- ↳ c-use(v,n): var v is used in computation at node n
- ↳ p-use(v,m,n): var v used in predicate from node m to n

var kill written as k(v,n): var v dealloced at node n

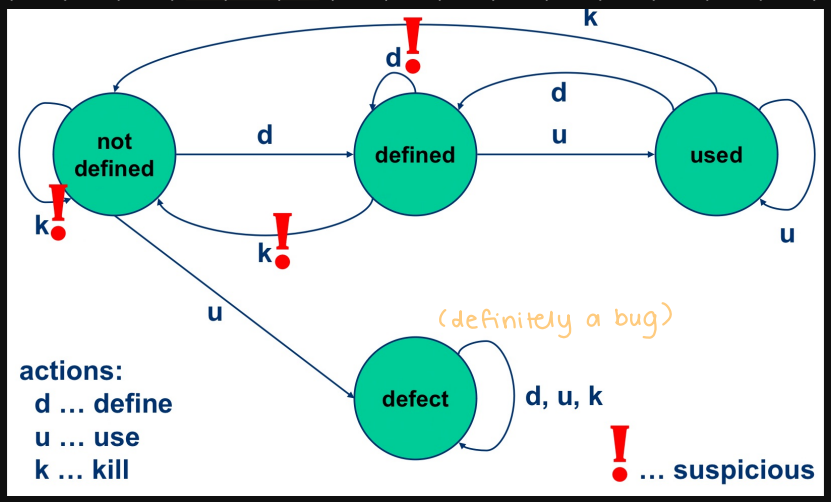
e.g.

```
int main(void) {
 1  char *line;
 2  int x = 0, y;              d(x,2)
 3  line = malloc(256 * sizeof(*line));
 4  fgets (line, 256, stdin);  d(line,4)
 5  scanf ("%d", &y);          d(y,5)
 6  if (y > x)                 p-use(y,6,7/9) p-use(x,6,7/9)
 7    y = y - x;               c-use(x,7) c-use(y,7) d(y,7)
 8  else {
 9    x = getvalue();          d(x,9)
10    y = y - x;               c-use(y,10) c-use(x,10) d(y,10)
11  }
12  printf("%s%d", line, y);   c-use(line,12) c-use(y,12)
13  free(line);                k(line,13)
}
```

a p-use per branch (6-7 and 6-9)

order is important! definition after uses!

data flow anomalies:



(definitely a bug)

(might be a bug)

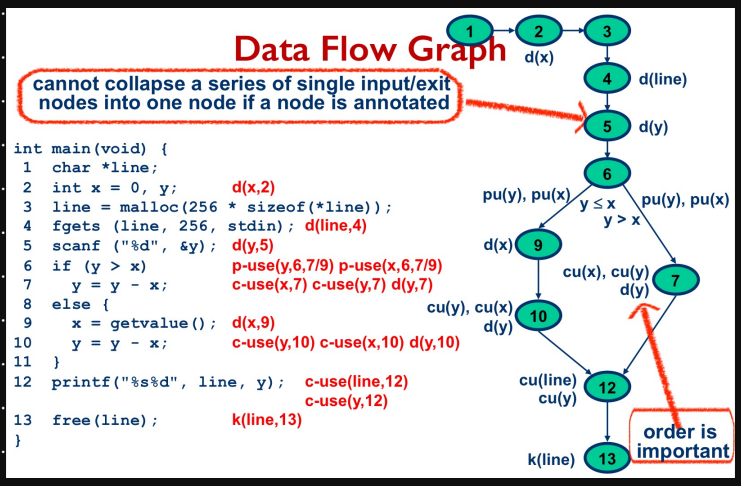
data flow actions
- ↳ legend:
  - S: suspicious
  - D: defect
  - PD: probably defect
  - N: normal use
  - O: ok
- ↳ for each successive pair of actions:

1st \ 2nd	d	k	u
d	S	PD	N
k	O	PD	D
u	O	O	O

↳ 1st occurrence of action on var:
- k → S
- d → 0
- u → S (may be global)

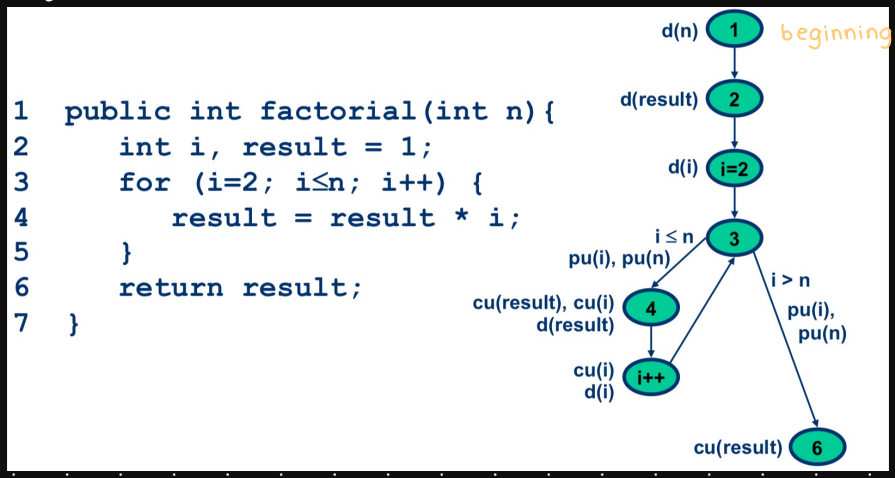
↳ last occurrence of action on var:
- k → 0
- d → S
- u → 0 (maybe forgot dealloc)

data flow graph (DFG) captures flow of defns + uses across basic blocks in program
↳ aka def-use graph
↳ annotate each node w/ def + c-used + each edge w/ p-use
↳ label each edge w/ cond that when T causes edge to be taken
↳ e.g.



node i	def(i)	c-use(i)	edge(i,j)	p-use(i,j)
2	x		(2,3)	
4	line		(4,5)	
5	y		(5,6)	
6			(6,7)	y,x
6			(6,9)	y,x
7	y	x,y	(7,12)	
9	x		(9,10)	
10	y	y,x	(10,12)	
12		line,y	(12,13)	

e.g. DFG for factorial



complete path: init node is start node + final node is exit node
simple path: all nodes (except possibly 1st + last) are distinct
loop-free path: all nodes are distinct

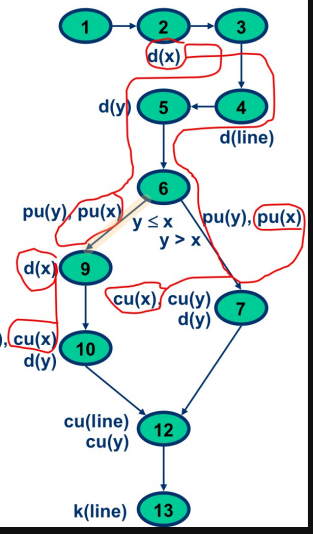
- **def-clear path** wrt v: any path starting from node where var v is defined & ending at node where v is used, w/o redefining v anywhere else in path
- **du-pair** wrt v written as (d, u)
  - ↳ d is node where v is defined
  - ↳ u is node where v is used
  - ↳ def-clear path wrt v from d to u
  - ↳ aka **reach**, as in def of v at d reaches use at u
- **du-path** wrt v is path $P = \langle n_1, \ldots, n_j, n_k \rangle$ st $d(v, n_i)$ + either 1 of 2 cases:
  - ↳ c-use of v at node n_k + P is def-clear simple path wrt v
    - i.e. at most, single loop traversal
  - ↳ p-use of v on edge n_j to n_k + $\langle n_1, n_2, \ldots, n_j \rangle$ is def-clear, loop-free path

e.g.

du-paths for variable x:

2 – 3 – 4 – 5 – 6 – 7 (p,c)
2 – 3 – 4 – 5 – 6 – 9 (p)
9 – 10 (c)

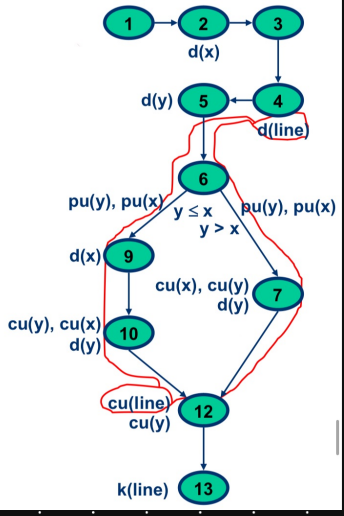
must specify which node p-use of x leads to (doesn't include redefn of x at 9)



e.g.

- du-paths for variable line:

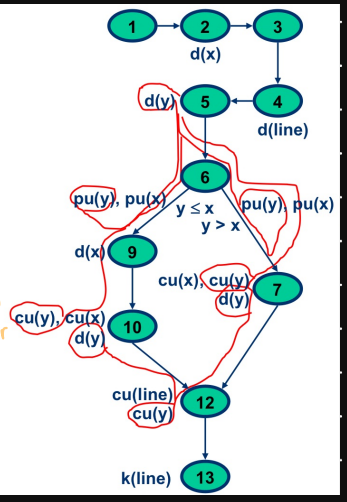
4 – 5 – 6 – 7 – 12 (c)
4 – 5 – 6 – 9 – 10 – 12 (c)



e.g.

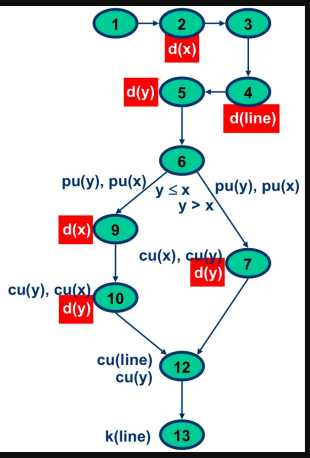
- du-paths for variable y:

5 – 6 – 7 (p,c)
5 – 6 – 9 (p)
5 – 6 – 9 – 10 (c) ← don't include p-use of y at 6 b/c it's path for 5-10 du-pair
7 – 12 (c)
10 – 12 (c)



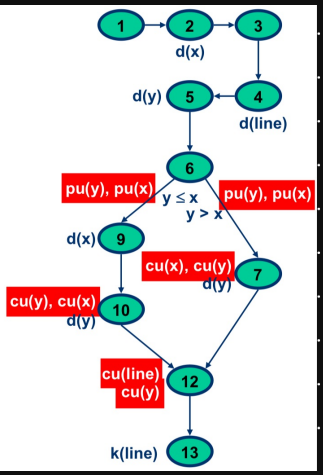
all-defns cov: at least 1 def-clear path from every defining node of v to at least 1 use of v (p-use / c-use)
↳ aka all-def
↳ e.g.

x:	2 – 3 – 4 – 5 – 6 – 7	(p)
	9 – 10	(c)
line:	4 – 5 – 6 – 7 – 12	(c)
y:	5 – 6 – 7	(p)
	7 – 12	(c)
	10 – 12	(c)

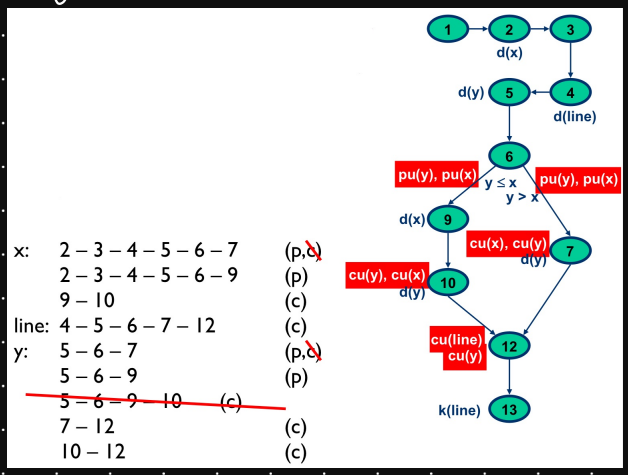


all-uses cov: at least 1 def-clear path from every defining node of v to every reachable use of v
↳ e.g.

x:	2 – 3 – 4 – 5 – 6 – 7	(p,c)
	2 – 3 – 4 – 5 – 6 – 9	(p)
	9 – 10	(c)
line:	4 – 5 – 6 – 7 – 12	(c)
y:	5 – 6 – 7	(p,c)
	5 – 6 – 9	(p)
	5 – 6 – 9 – 10 (c)	
	7 – 12	(c)
	10 – 12	(c)

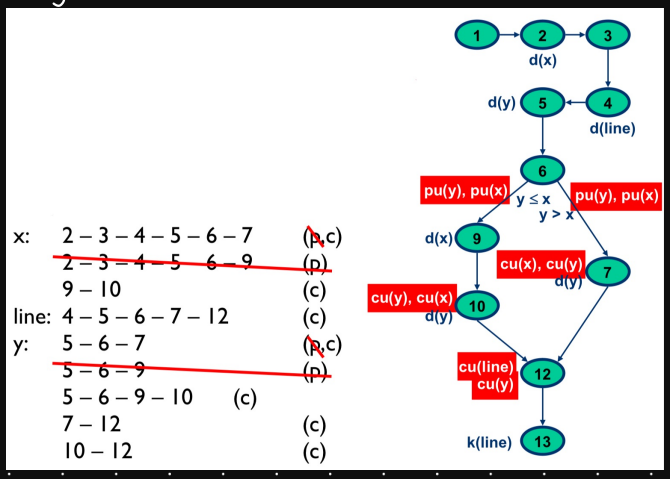


all p-uses / some c-uses cov: at least 1 def-clear path from every defining node of v to every reachable p-use of v
↳ if no p-uses, make path to c-use
↳ e.g.



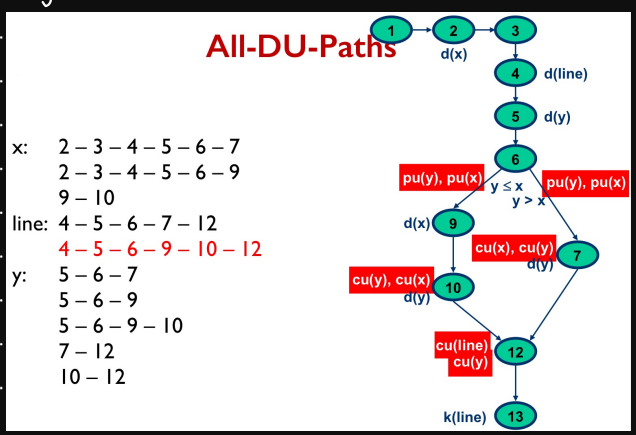
x: 2 – 3 – 4 – 5 – 6 – 7 (p,c)
2 – 3 – 4 – 5 – 6 – 9 (p)
9 – 10 (c)
line: 4 – 5 – 6 – 7 – 12 (c)
y: 5 – 6 – 7 (p,c)
5 – 6 – 9 (p)
~~5 – 6 – 9 – 10 (c)~~
7 – 12 (c)
10 – 12 (c)

all c-uses / some p-uses cov: at least 1 def-clear path from every defining node of v to every reachable c-use of v.
↳ if no c-uses, make path to p-use
↳ e.g.



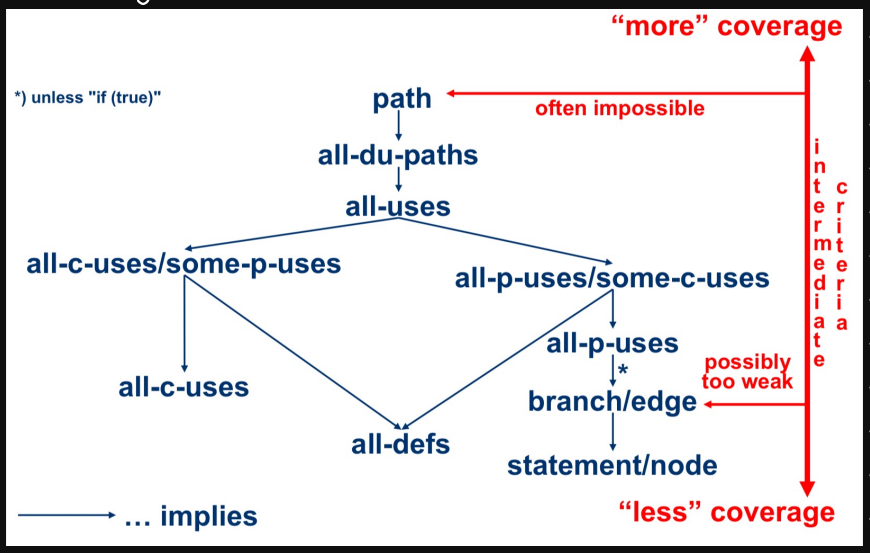
x: 2 – 3 – 4 – 5 – 6 – 7 (p,c)
~~2 – 3 – 4 – 5 – 6 – 9 (p)~~
9 – 10 (c)
line: 4 – 5 – 6 – 7 – 12 (c)
y: 5 – 6 – 7 (p,c)
~~5 – 6 – 9 (p)~~
5 – 6 – 9 – 10 (c)
7 – 12 (c)
10 – 12 (c)

all-du-paths cov: all du-paths for every defn is included
↳ e.g.



x: 2 – 3 – 4 – 5 – 6 – 7
2 – 3 – 4 – 5 – 6 – 9
9 – 10
line: 4 – 5 – 6 – 7 – 12
4 – 5 – 6 – 9 – 10 – 12
y: 5 – 6 – 7
5 – 6 – 9
5 – 6 – 9 – 10
7 – 12
10 – 12

hierarchy of data flow cov criteria:



MUTATION TESTING

- **mutation testing**: modify / mutate some stmts in code so we have diff, buggy versions of code
  - ↳ used to measure test suite effectiveness b/c we run the tests on mutated versions + see if they fail (i.e. find the bug)
  - ↳ **fault-based** testing (i.e. directed twd typical faults)
  - ↳ should be used w/ traditional testing techniques + can't replace them
- **steps** in mutation testing:
  1) modify stmts in code + create **mutants**
  2) test cases run thru mutants
  3) compare og results vs mutants'
  4) mutant is **killed/dead** if diff results are found + test set is adequate b/c it detected change
     - ↳ mutant is **alive** if results are same + test case is ineffective
- mutant is alive b/c:
  - ↳ it's equiv to og program (i.e. syntactically diff but fcnally equiv)
  - ↳ test set inadequate to kill it
- **stillborn mutant**: syntactically wrong + killed by compiler
- **trivial mutant**: killed by almost any test case
- **equiv mutant**: always prods same output as og program
- **mutation operators**:

1. ABS — Absolute Value Insertion:

Each arithmetic expression (and subexpression) is modified by the functions *abs()*, *negAbs()*, and *failOnZero()*.

Examples:

```
      a = m * (o + p);
Δ1    a = abs (m * (o + p));
Δ2    a = m * abs ((o + p));
Δ3    a = failOnZero (m * (o + p));
```

2. AOR — Arithmetic Operator Replacement:

Each occurrence of one of the arithmetic operators +, −, *, /, and % is replaced by each of the other operators. In addition, each is replaced by the special mutation operators leftOp, and rightOp.

Examples:

```
      a = m * (o + p);
Δ1    a = m + (o + p);
Δ2    a = m * (o * p);
Δ3    a = m leftOp (o + p);
```

3. ROR — Relational Operator Replacement:

Each occurrence of one of the relational operators (<, ≤, >, ≥, =, ≠) is replaced by each of the other operators and by falseOp and trueOp.

Examples:

```
      if (X <= Y)
Δ1    if (X > Y)
Δ2    if (X < Y)
Δ3    if (X falseOp Y)  // always returns false
```

4. COR — Conditional Operator Replacement:

Each occurrence of one of the logical operators (and - &&, or - || , and with no conditional evaluation - &, or with no conditional evaluation - |, not equivalent - ^) is replaced by each of the other operators; in addition, each is replaced by falseOp, trueOp, leftOp, and rightOp.

Examples:

```
      if (X <= Y && a > 0)
Δ1    if (X <= Y || a > 0)
Δ2    if (X <= Y leftOp a > 0) // returns result of left
clause
```

5. SOR — Shift Operator Replacement:

Each occurrence of one of the shift operators <<, >>, and >>> is replaced by each of the other operators. In addition, each is replaced by the special mutation operator leftOp.

Examples:

```
      byte b = (byte) 16;
      b = b >> 2;
Δ1    b = b << 2;
Δ2    b = b leftOp 2; // result is b
```

6. LOR — Logical Operator Replacement:

Each occurrence of one of the logical operators (bitwise and - &, bitwise or - |, exclusive or - ^) is replaced by each of the other operators; in addition, each is replaced by leftOp and rightOp.

Examples:

```
      int a = 60;    int b = 13;
      int c = a & b;
Δ1    int c = a | b;
Δ2    int c = a rightOp b; // result is b
```

11. BSR — Bomb Statement Replacement:

Each statement is replaced by a special Bomb() function.

Example:

```
      a = m * (o + p);
Δ1    Bomb() // Raises exception when
reached
```

mutation operators specific to OOP langs:

- ↳ replacing type w/compatible subtype (i.e. inheritance)
- ↳ changing access modifier of attr / method
- ↳ changing instance creation expr
- ↳ changing order of params in method defn
- ↳ changing order of params in call
- ↳ removing overloading method

↳ reducing # params
↳ removing overriding method

we make 2 assumptions:

↳ **competent programmer**: program developed is correct or differs from correct program by combo of simple errors
- i.e. algo + general strategy is right

↳ **coupling effect**: test data is sensitive enough that if it detects simple fault, it implicitly detects complex ones
- i.e. combo of simple errors allow for high probability of detecting complex ones

MUTATION COVERAGE

mutants is large b/c they must capture all possible syntactic variations in program

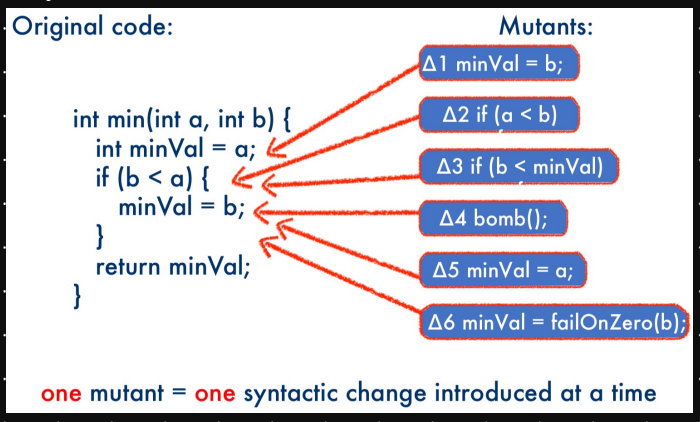
complete cov is when we kill all non-equiv mutants

amt of cov is called **mutation score**

↳ **mutation score** $= 100 * \frac{D}{N-E}$
- D: # dead mutants
- N: # total mutants
- E: # equiv mutants

↳ test set is mutation adequate if score is 100%

e.g.



↳ Δ3 is equiv b/c minVal = a at that time

to have good test case, must have **RIP (reachability, infection, propagation)**

strong mutation: fault must be reachable, infect the state, + propagate to output

↳ given mutant m for program P + test case t, t strongly kills m iff output of t on P is diff from output of t on m

↳ **strong mutation cov (SMC)**: for each mutant m in M, test suite TR contains test

weak mutation: fault that kills mutant only infects state but doesn't propagate to output

↳ given mutant m that modifies source loc L in program P, & test case t, t weakly kills m iff state of execution of P on t is diff from state of execution of m on t, immediately after execution of L

↳ **weak mutation cov (WMC)**: for each mutant m in M, TR contains test case which weakly kills m

e.g.

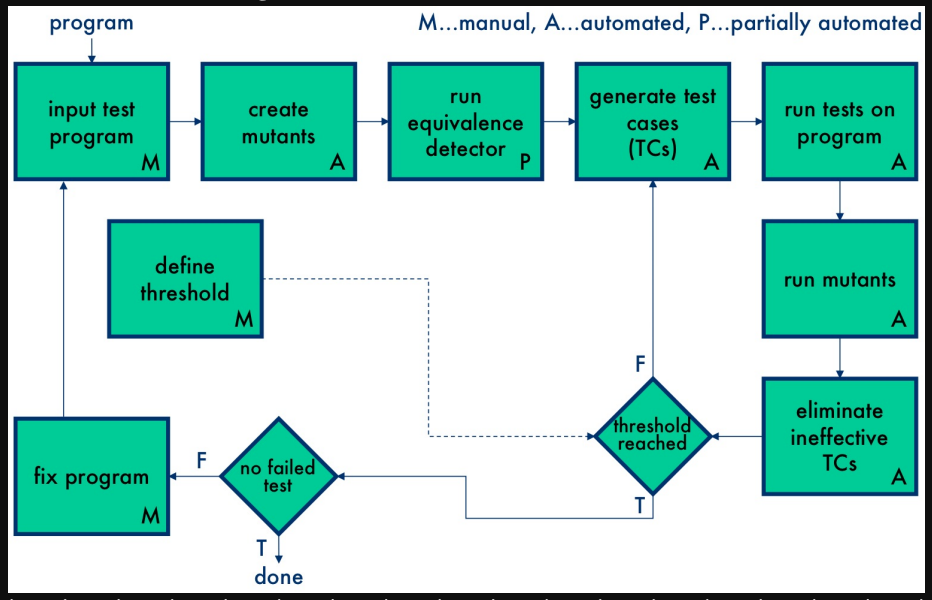
```
1. int min(int A, int B)
2. { // original
3.    int minVal;
4.    minVal = A;
5.    if (B < A) {
6.     minVal = B;
7.   }
8.   return minVal;
9. }
```

```
1. int min(int A, int B)
2. { // mutant
3.    int minVal;
4.1   minVal = B;
3.    if (B < A) {
4.     minVal = B;
5.   }
6.   return minVal;
7. }
```

Replace one varirable with another

- Reachability: unavoidable
- Infection: need B!= A
- Propagation: wrong minVal needs to return to the caller; that is we cannot execute the body of the if statement, so need B > A
- Condition for strongly killing mutation B > A
  - TC: (A=5, B=7), return 7 but expected 5
- Conditions for weakly killing mutation B!=A
  - TC: (A=8, B=2), return 2 and expected 2

mutation testing process:



e.g.

```
1:    public static double getMinValue(double[] array, int from) {
2:      int min = from;
3:      int temp = array[min];
4:      for (int i = from + 1; i < array.length; i++) {
5:       //mutant: for (int i = from + 2; i < array.length; i++) {
6:         if (array[min] > array[i]) {
7:            min = i;
8:         }
9:       }
10:      return array[min];
11:    }
```

↳ If possible, find test inputs that do **not** reach the mutant. If it is impossible, explain why.

- if from + 1 ≥ array.length, we nvr enter for loop anyways in both cases

↳ If possible, find test inputs that satisfy **reachability** but not infection for the mutant. If it is not possible, explain why.

- this isn't possible b/c i will always be a diff val in the for loop compared to og

↳ If possible, find test inputs that satisfy **reachability and infection**, but not propagation for the mutant. If it is not possible, explain why.

- if array[min] > array[i] is always F so we nvr enter if body (i.e array[from] is minimum elmt)

↳ Find a mutant of line 4 that is equivalent to the original statement.

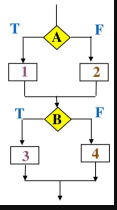
- for (int i = min + 2 ; i < array.length ; i++)

CREATING TESTS

TEST CASES FOR PATHS

- to create test for path, generate input data that satisfies all conds on path that makes it execute (if possible)
- key items for path test case:
  - ↳ input vector
  - ↳ predicate
  - ↳ path predicate
  - ↳ predicate interpretation
  - ↳ path predicate expr
  - ↳ test input from path predicate expr
- **input vector**: collection of all data entities whose vals must be fixed before entering + being read by routine
  - ↳ input args
  - ↳ global vars + constants
  - ↳ files
  - ↳ network connections
  - ↳ timers
- **predicate**: logical fcn evaled at decision pt
  - ↳ e.g.

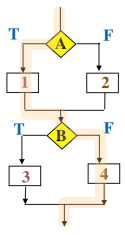
```
if (a < b)
        { c = a + b ; d = a * b }
else
        { c = a * b ; d = a + b }
if (c < d)
        { x = a + c ; y = b + d }
else
        { x = a * c ; y = b * d }
```



    - predicates
      - → a < b = T
      - → a < b = F
      - → c < d = T
      - → c < d = F
- **path predicate**: set of predicates associated w/path
  - ↳ e.g.

• In the following a < b = true and c < d = false is a path predicate for path A 1 B 4

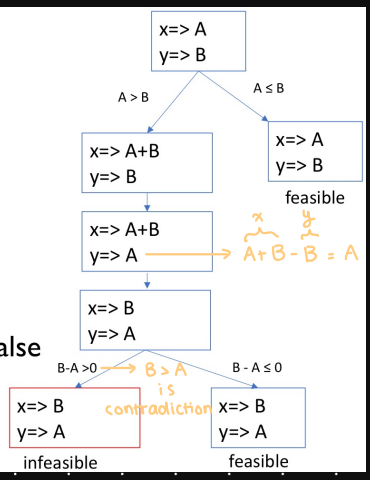
```
if (a < b)
        { c = a + b ; d = a * b }
else
        { c = a * b ; d = a + b }
if (c < d)
        { x = a + c ; y = b + d }
else
        { x = a * c ; y = b * d }
```



predicate interpretation: since local vars can't be selected independently of input vars + play no role in selecting inputs, we assign vals to local vars to eliminate them
↳ **symbolic execution**: using symbolic substitutes along path so we can express pred solely in terms of input + constant vectors
- use automated thm prover to check if there's concrete input vals that make program fail

↳ pred may have diff interpretations depending on how control reaches it
↳ e.g.

```
def f (x, y):

    if (x > y):
        x = x + y
        y = x - y
        x = x - y
        if (x - y > 0):
            assert false
    return (x, y)
```



- all paths in program form **execution tree** w/ feasible + infeasible paths

path pred expr is interpreted path pred
↳ no local vars
↳ set of constraints in terms of input vector + maybe constants
↳ inputs that force each path are gen by solving constraints
↳ if it has no soln, path is infeasible

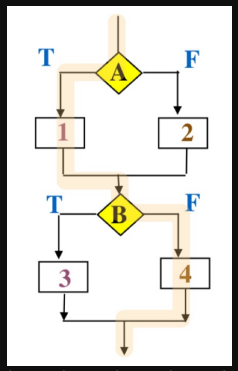
e.g. gen input vals

```
if (a < b) { c = a + b ; d = a * b }
else { c = a * b ; d = a + b }
if (c < d) { x = a + c ; y = b + d }
else { x = a * c ; y = b * d }
```

- Path predicate $a < b = \text{true} \wedge c < d = \text{false}$
- Substitute for c and d $c = a + b \quad d = a * b$

$$a < b = \text{true} \wedge a + b < a * b = \text{false}$$

$$a < b \wedge a + b \geq a * b$$



↳ solve for a and b (solns aren't unique):
- a = 0
- b = 1

organize set of path preds using **decision table**
↳ e.g.

	A1B3	A1B4	A2B3	A2B4
A < B	T	T	F	F
C < D	T	F	T	F
A value	2	0	1	5
B value	5	1	0	2

- A1B3 + A2B4 or A1B4 + A2B3 both give stmt cov

PRINCIPLES OF MAINTAINABLE TEST CODE

tests should be:

↳ **fast**
- if facing slow test:
  - → use mocks/stubs to replace slower components
  - → redesign prod code so slower pieces of code can be tested separately from faster ones
  - → move slower tests to diff suite that we run less often

↳ **cohesive, independent, + isolated**
- single test should test 1 fonality / behaviour
- test shouldn't depend on others to succeed
- set up data + clean up own mess

↳ having **reason to exist**
- find bugs or document behaviour
- not for inc code cov

↳ **repeatable + not flaky**
- same results no matter how many times it's executed

↳ having **strong assertions**

↳ **breaking** if behaviour changes
- test driven development (TDD) is useful for this

↳ having **single + clear reason** to fail
- descriptive name
- input vals are understandable
- clear assertions + explain why val is expected

↳ **easy to write**
- e.g. if test require db, provide API that can set up db w/ a few method calls

↳ **easy to read**

↳ easy to **change + evolve**

e.g.

```
public class Invoice {

  private final double value;
  private final String country;
  private final CustomerType customerType;

  public Invoice(double value, String country, CustomerType customerType) {
    this.value = value;
    this.country = country;
    this.customerType = customerType;
  }

  public double calculate() {
    double ratio = 0.1;

    // some business rule here to calculate the ratio
    // depending on the value, company/person, country ...

    return value * ratio;
  }
}
```

The method we will soon test. Imagine business rule here.

An invoice class may have a lot of attribute

```
@Test
void test1() {
  Invoice invoice = new Invoice(new BigDecimal("2500"), "NL",
    CustomerType.COMPANY);
  double v = invoice.calculate();
  assertThat(v).isEqualTo(250);
}
```

A not-very-clear test

```
@Test
void taxesForCompanies() {
  Invoice invoice = new InvoiceBuilder()
      .asCompany()
      .withCountry("NL")
      .withAValueOf(2500.0)
      .build();

  double calculatedValue = invoice.calculate();

  assertThat(calculatedValue)
    .isEqualTo(250.0); // 2500 * 0.1 = 250
}
```

The Invoice object is now built through a fluent builder.

The variable that holds the result has a better name.

The assertion has a comment to explain where the 250 comes from.

```
@Test
void taxesForCompanyAreTaxRateMultipliedByAmount() {
  double invoiceValue = 2500.0;
  double tax = 0.1;

  Invoice invoice = new InvoiceBuilder()
      .asCompany()
      .withCountry("NL")
      .withAValueOf(invoiceValue)
      .build();

  double calculatedValue = invoice.calculate();

  assertThat(calculatedValue)
    .isEqualTo(invoiceValue * tax);
}
```

Declares the invoiceValue and tax variables

Uses the variable instead of the hard-coded value

The assertion uses the explanatory variables instead of hard-coded numbers.

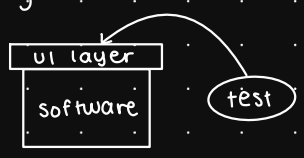
TEST SMELLS

test smells (aka anti-patterns) are symptoms of potential deeper issues in testing

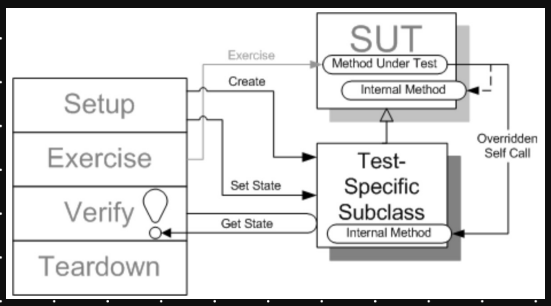
- eager test: several methods of obj are tested
  - ↳ soln is to separate into tests that only test 1 method each
- lazy test: several tests check same method using same fixture
  - ↳ soln is to join them tgt
- mystery guest: when test uses external resources (e.g. file containing test data) so it's not self-contained
  - ↳ intros hidden dependencies
  - ↳ solns:

- inline resource: incorporate resource in test code
- setup external resource: test explicitly creates/allocs resources before testing + releases them when done

- **resource optimism**: test code makes assumptions abt existence/absence + state of external resources
  - ↳ soln is to setup external resource
- **test run war**: tests fail when multiple programmers running them
  - ↳ most likely cause is resource interference (e.g. temp files used by others)
  - ↳ soln is to make resource unique
- **general fixture**: setup fixture is too general + diff tests only access part of fixture
  - ↳ may make tests run slower
  - ↳ soln is to use setup only for part of fixture used by all methods + put rest in specific method using it inline
- **assertion roulette**: assertions in test method have no explanation
  - ↳ if 1 fails, don't know which one
  - ↳ soln is to add assertion explanation
- **indirect testing**: test class contains methods that perform tests on other objs
  - ↳ e.g.



  - ↳ soln is to apply extract then move methods on that part of test
  - ↳ may be problems w/ data hiding in prod code
- **for testers only**: when prod class has methods only used by test methods
  - ↳ not needed or only needed to set up testing fixture
  - ↳ soln is to extract them as subclass so others can't use them in prod code
    -



- **sensitive equality**: fast + easy way to test for equality is to map result to string, but this may depend on irrelevant details (e.g. commas, spaces)
  - ↳ soln is to replace toString equality checks w/ intro equality method
    - i.e. implement equals method in obj's class
- **test code duplication**
  - ↳ if there's duplication of test code in same class, soln is to use extract method

↳ special case is test implication (i.e. A + B cover same prod code + A fails iff B fails)
  - can occur when code is refactored

more test smells:

- Redundant assertion

```
@Test
public void testTrue()
{
    assertEquals(true, true);
}
```

- Unknown Test (missing assertion)

```
@Test
public void hitGetPOICategoriesApi() throws Exception
{
    POICategories poiCategories = apiClient.getPOICategories(16);
    for (POICategory category : poiCategories) {
System.out.println(category.name() + ": " + category);
    }
}
```

- Empty test

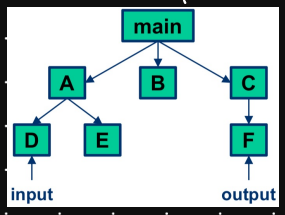
```
public void testCredGetFullSampleV1() throws Throwable{
//      ScrapedCredentials credentials = innerCredTest(FULL_SAMPLE_v1);
//      assertEquals("p4ssw0rd", credentials.pass);
//      assertEquals("user@example.com",credentials.user);
}
```

- Duplicate Assert

```
public void testXmlSanitizer()
{
    boolean valid = XmlSanitizer.isValid("Fritzbox");
    valid = XmlSanitizer.isValid("Fritz-box");
    assertEquals("Minus is valid", true, valid)
    valid = XmlSanitizer.isValid("Fritz-box");
    assertEquals("Minus is valid", true, valid)
    System.out.println("Minus test - passed");
}
```

INTEGRATION TESTING

- integration strategy is to assemble individual components to form larger program entities
  - ↳ test component interaction
  - ↳ determine optimal order of integration
- objective of integration testing is to ensure components work in isolation + when assembled
  - ↳ need component dependency struct



- assuming all components work in isolation, mostly interface-related faults come from integration testing
  - ↳ e.g. wrong method call, params, version use
- stubs replace called modules
  - ↳ can replace whole components (e.g. db)
  - ↳ declared / invoked as real module
  - ↳ e.g.

```
public static int someFunc(float fThat)
```

•Example of stub:

```
public static int someFunc(float fThat) {
  int iDummy = 42;
  System.out.println("In method someFunc " +
    "Input float =" + fThat);
  System.out.println("Returning " + iDummy +
    ".");
  return iDummy;
}
```

- common fcns of stub:
  - ↳ display / log trace msg + passed params
  - ↳ return val according to test objective
    - e.g.

```
void main() {
1 int x, y;
2 x = A();
3 if (x > 0) {
4   y = B(x);
5   C(y);
  } else {
6   C(x)
  }
7 exit(0);
}
```

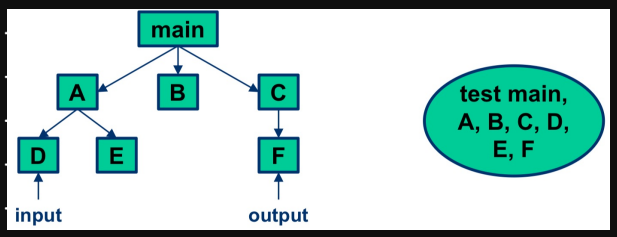
- Test for path 1 – 2 – 3 - 4 – 5 – 7:
  - Stub for A() such that x > 0 returned
- Test for path 1 – 2 – 3 – 6 – 7:
  - Stub for A() such that x <=0 returned
- Problem:
  - Stubs can become too complex

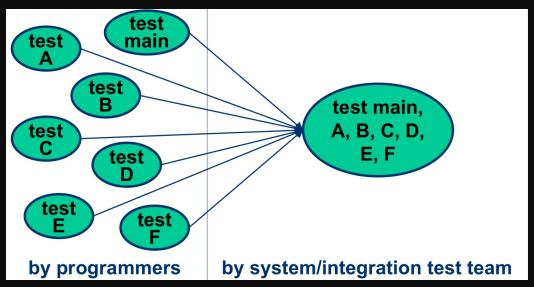
drivers are used to call test modules + controls execution of tests
↳ responsible for param passing + handling return vals
↳ usually simpler than stubs
↳ e.g.

```
public class TestSomething {
  public static void main (String[] args) {
    float iSend = 98.6f;
    Whatever what = new Whatever();
    System.out.println("Sending someFunc: " +
      iSend);
    int iReturn = what.someFunc(iSend);
    System.out.println ("SomeFunc returned: " +
        iReturn);
  }
}
```

INTEGRATION TESTING STRATEGIES

big bang: non-incremental strategy that integrates all components as whole
↳ assumes components are alr tested in isolation

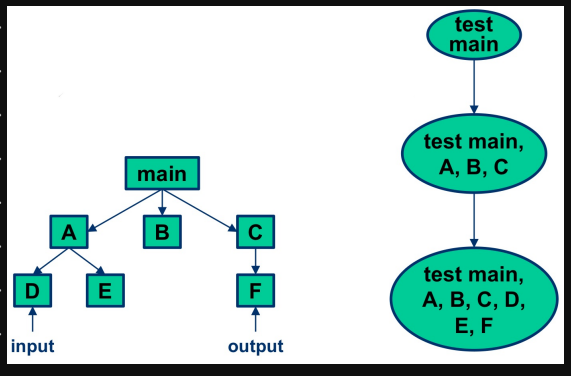




↳ advantages:
- convenient for small + stable systems

↳ disadvantages:
- doesn't allow parallel dev (i.e. testing while implementing)
- difficult to localize faults
- easy to miss interface faults

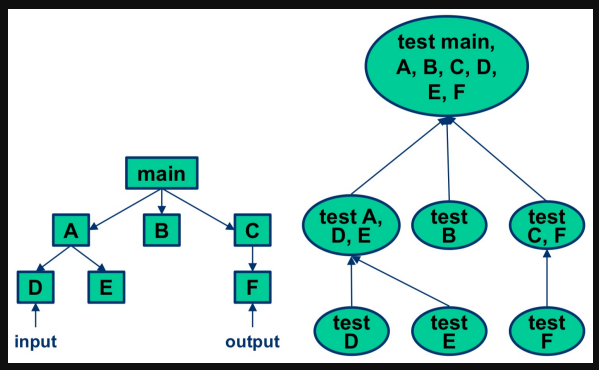
top-down: incremental strategy where we test HL components then keep calling lower lvl components



↳ can alter order so critical + I/O components are tested ASAP
↳ advantages:
- fault localization easier
- few/no drivers needed
- can get early prototype
- parallel w/ implementation
- major design flaws in HL logic components found first

↳ disadvantages:
- need lots of stubs
- potentially reusable components (at bottom of hierarchy) may be inadequately tested

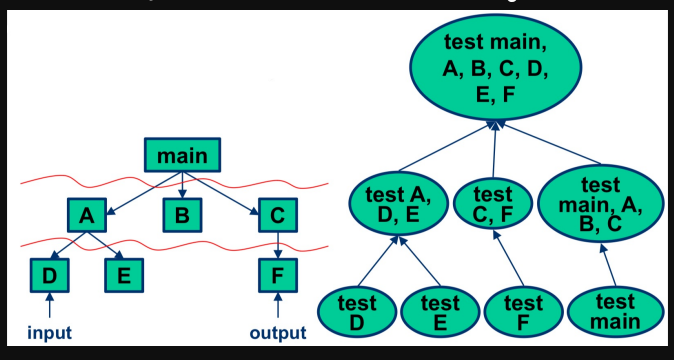
bottom-up: incremental strategy where we test LL components then keep calling higher lvl components



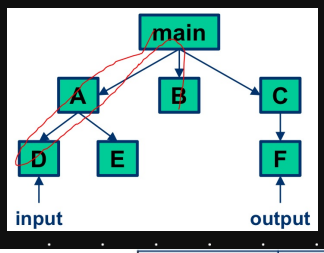
↳ advantages:
- fault localization easier
- no stubs needed
- reusable components tested thoroughly
- parallel w/ implementation

↳ disadvantages:
- needs drivers
- HL components, which are related to soln logic, tested last
- no concept of early skeletal system

sandwich: combines top-down + bottom-up approaches
↳ distinguishes 3 layers: logic, middle, + operational



- risk-driven: integrate based on criticality (i.e. most critical / complex components first)
- fcn / thread-based: integrate according to threads / fcns components belong to



	Big Bang	Top Down	Bottom Up	Sandwich	Risk Driven	Thread
Fault Localization	Difficult	Easier	Easier	Easier	Easier	Medium
Effort Required	Convenient for small systems	Few drivers, many stubs needed	Many drivers, few stubs needed	Both drivers and stubs needed	Both drivers and stubs needed	Few drivers, many stubs needed
Degree of Testing	Equal	Focus on major design flaws, not reusable modules	Focus on reusable modules, not major design flaws	Focus on both as needed	Focus on critical module	Focus on major design flaws, not reusable modules
Parallel Development & Testing	No	Yes (prototype / vary test order)	Yes (vary test order but no prototype)	Yes (prototype / vary test order)	Yes (but possibly no prototype)	Yes (prototype / vary test order)

TESTING OBJECT-ORIENTED SYSTEMS

- unit + integration testing are white-box testing so they're driven by software struct
  - ↳ need specific techniques for OO systems
- in procedural programming:
  - ↳ basic component is fcn
  - ↳ testing method based on I/O rltn
- in OO programming:
  - ↳ basic component is class
  - ↳ objs are tested
  - ↳ correctness must include I/O rltn + obj state
- in OO systems, test single methods using traditional techniques
  - ↳ most methods only contain a few lines of code (LOC)
  - ↳ method behaviour must be analyzed in rltn to other ops + joint effects on shared state
- new problems w/ testing classes:

- ↳ how to combine I/O rtns w/ state info to define test cases + **oracles** (i.e. std that is ref for correct/expected outcome)
- ↳ manipulate obj state w/o violating encapsulation principles
- ↳ due to polymorphism + dynamic binding, test 1-to-many possible invocations of same interface
- ↳ every exception must be tested
  - e.g. null ptr

e.g.

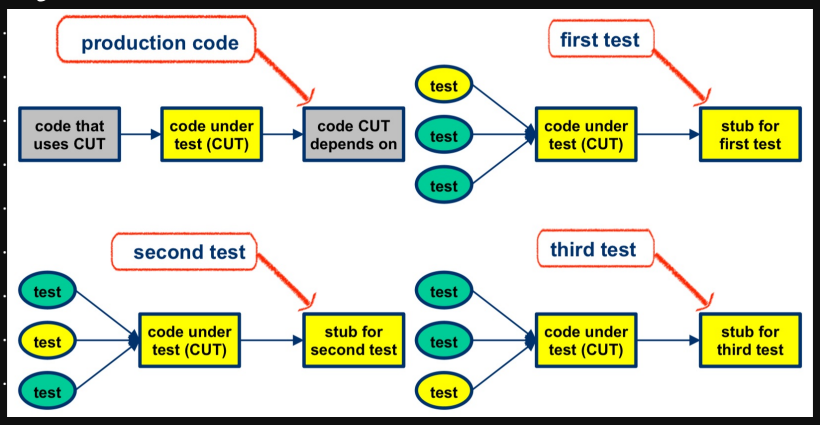
- Testing method checkPressure() in isolation is **meaningless**:
  - Generating test data
  - Measuring coverage
- Creating oracles is **more difficult**:
  - The value produced by checkPressure() depends on the state of class Watcher's instances (variable status)

```
class Watcher {
  private:
    …
    int status;
    …
  public:
    void checkPressure() {
      …
      if (status == 1)
        …
      else if (status …)
        …
    }
  …
}
```

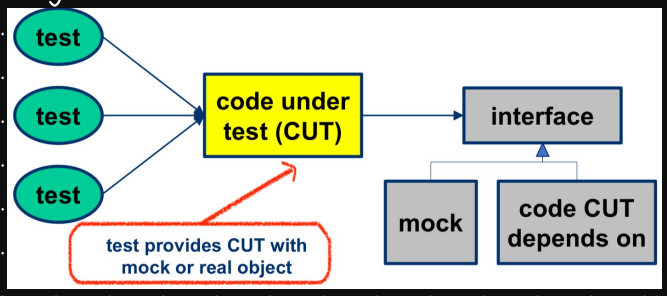
- new **fault models** must be created targeting OO specific faults
  - ↳ e.g. wrong instance of method inherited b/c of multiple inheritance
  - ↳ e.g. wrong redefn of attr
- OO **integration lvls**:
  - ↳ basic unit testing: single method of class
    - aka **intra-method** testing
  - ↳ unit testing: integrate methods within class
    - aka **intra-class** testing
    - black + white box approaches w/ data flow across several methods
    - each exception raised at least once
    - each attr set/get at least once
    - state-based testing
    - big bang method for tightly coupled methods
    - **alpha-omega cycle**
      - → ctors, get, Boolean, set, iterators, dtors
    - use stubs/mocks for complex methods
  - ↳ diff ways to do **cluster integration**:
    - inheritance
    - containment
      - → e.g. 1 class has instance of another as attr
    - classes form subsystem/component
    - use big bang, bottom-up, top-down, + case-based techniques
  - ↳ integration of components into single app
    - aka **subsystem/system integration**
    - e.g. client/server integration

MOCK OBJECTS

- testing of OO systems requires drivers + stubs
  - ↳ difficult to flexibly stub dependent code w/o changing code under test (CUT) + maintaining lib of stub objs
    - e.g.



- mock obj is a form of stubs
  - ↳ based on interfaces
  - ↳ easier to setup + control
  - ↳ isolates code from details that may be filled in later
  - ↳ refined by incrementally replacing w/ acc code
  - ↳ based on dependency inversion principle (i.e. HL modules use interfaces as abstract layer instead of LL modules directly)
    - e.g.



- e.g.:

- Mocks replace HttpServletRequest and HttpServletResponse when testing doGet:

```
1  public void test_boil() throws Exception {
2    TemperatureServlet s = new TemperatureServlet();
3    MockHttpServletRequest request = new MockHttpServletRequest();
4    MockHttpServletResponse response = new MockHttpServletResponse();
5
6    request.setupAddParameter("Fahrenheit", "212");
7    response.setExpectedContentType("text/html");
8    s.doGet(request, response);
9    response.verify();
10   assertEquals("Fahrenheit: 212, Celsius: 100.0\r\n",
11        response.getOutputStreamContents());
12 }
```

- manually creating mocks allow for more control, but may need several

classes + clash w/ existing infrastructure classes

- we'll use **Mockito**, which is popular Java testing framework for creating + using mock objs
- when **creating** mock, it has same method calls as normal obj
  - ↳ records how other objs interact w/ it
  - ↳ no real obj instance
  - ↳ e.g. List

```
//Let's import Mockito statically so that the code looks clearer
import static org.mockito.Mockito.*;
//mock creation
List mockedList = mock(List.class);
Or @Mock List mockedList;
```

- mock will remember all interactions so we can selectively **verify** them
  - ↳ e.g.

```
mockList.add("one");
verify(mockedList).add("one");
verify(mockedList).add("two"); //will fail because we never called with this value
```

  - ↳ e.g. verifying #invocations

```
verify(mockedList, times(2)).add("one");//will fail since called once

verify(mockedList, atLeast(2)).add("one");//will fail since called once
```

  - ↳ e.g. checking if mocks have unverified interactions

```
"//using mocks

mockedList.add("one");
mockedList.add("two");


verify(mockedList).add("one");
//following verification will fail
verifyNoMoreInteractions(mockedList);
```

  - ↳ verifying determines what was passed to mocked method by method under test
    - **asserts** only check returned vals, but verify checks when method was called
    - still use asserts when testing vals of class/method under test
- **stubbing** returns whatever val passed in or default val
  - ↳ default vals include null, empty collection, + primitive vals
  - ↳ e.g. using "when" to stub method

```
when(mockStorage.barcode("1A")).thenReturn("Milk, 3.99");
```

  - ↳ last stubbing is 1 that's used

- e.g.
  - //All mock.someMethod("some arg") calls will return "two"
  - when(mock.someMethod("some arg")) .thenReturn("one")
  - when(mock.someMethod("some arg")) .thenReturn("two")

e.g. iterator-style stubbing (i.e. consecutive calls)

```
when(mockedStack.pop()).thenReturn(3,2,1);
```

ensure interactions happened in particular order w/ "inOrder"

↳ e.g.

```
List singleMock = mock(List.class);

//using a single mock
singleMock.add("was added first");
singleMock.add("was added second");

//create an inOrder verifier for a single mock
InOrder inOrder = inOrder(singleMock);

//following will make sure that add is first called with "was added first", then
with "was added second"
inOrder.verify(singleMock).add("was added first");
inOrder.verify(singleMock).add("was added second");
```

↳ e.g.

```
mock.foo(); //1st
mock.bar(); //2nd
mock.baz(); //3rd

InOrder inOrder = inOrder(mock);
inOrder.verify(mock).bar(); //2n
inOrder.verify(mock).baz(); //3rd (last method)

//passes because there are no more interactions after last
method:
inOrder.verifyNoMoreInteractions();

//however this fails because 1st method was not verified:
Mockito.verifyNoMoreInteractions(mock);
```

arg captor captures val that's passed in

↳ e.g.

- Set it up for barcode(String barcode)
  - ArgumentCaptor<String> argCaptor = ArgumentCaptor.forClass(String.class);
- Capture what was passed to barcode
  - verify(mockStorage).barcode(argCaptor.capture());
    - //this line verifies the barcode function is called and remembers what is the input for the bar code function
- Use that arg value later to verify what was displayed
  - verify(mockDisplay).showLine(argCaptor.getValue());
    - // to ensure the same values are called in both barcode and showLine

a spy of a real obj uses real methods when called

↳ e.g.

```
List list = new LinkedList();
List spy = spy(list);
//optionally, you can stub out some methods:
when(spy.size()).thenReturn(100);
 //using the spy calls *real* methods
spy.add("one");
```

↳ use only when there's legacy code we can't completely mock out

OO INTEGRATION

INHERITANCE TESTING

- classes intro diff scopes of integration testing
  - ↳ e.g. inter-class testing, rltns thru dependencies
- additional things to look out for when testing inherited classes:
  - ↳ instance vars are like global vars wrt individual methods
  - ↳ test suites almost nvr adequate for **overriding methods**
  - ↳ test **inherited method** in context of subclass
    - no issues if subclass is **pure extension** meaning there's no interaction btwn new instance vars/methods + inherited vars/methods
  - ↳ accidental reuse
    - e.g. inherited method not built for reuse
  - ↳ **multiple inheritance**
  - ↳ abstract classes (need instantiation to test)
  - ↳ generic classes
- e.g. inherited method

```
class A {
  int x; // invariant: x > 100
  void m() { // correctness depends on
             // the invariant } }
```

```
class B extends A {
   void m2() { x = 1; }  }
```

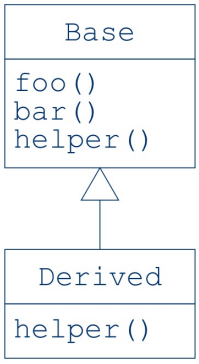
  - ↳ m2 might break invariant so m is incorrect in context of B
    - must test m again in B
- **overriding subclass method** must be tested w/ diff test sets
  - ↳ struct (i.e. data + control flow) of overriding method may differ
  - ↳ behaviour (i.e. I/O rltn) is also probably diff
  - ↳ e.g.

```
class A {
void m() { …; m2(); … }
void m2() { … }    }
```

```
class B extends A {
       void m2() { … } }
```

    - m calls B.m2 not A.m2 in B so both m + m2 must be retested
    - test cases for m in A may be insufficient for B, but all superclass tests must still be run
  - ↳ e.g

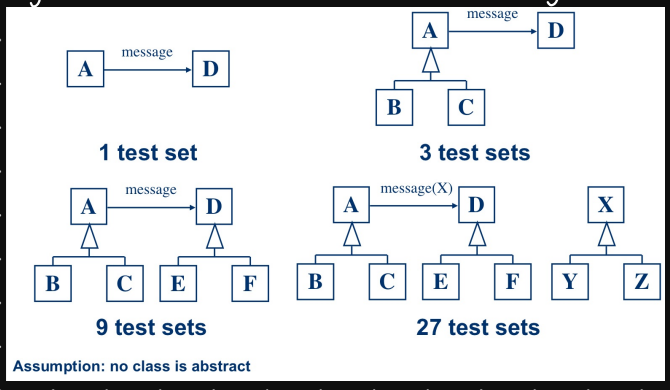
```
class Base  {
  public:
    void foo()     { … helper(); …}
    void bar()     { … helper(); …}
  private:
    virtual void helper()      {…}
};
class Derived : public Base {
  private:
    virtual void helper()      {…}
};

void test_driver()       {
  Base base;
  Derived derived;
  base.foo();      // test case 1
  derived.bar();   // test case 2
}
```



- test cases fully exercise code in both classes according to traditional control flow cov (i.e. 100% line, block, etc. cov)
- haven't acc fully tested all interactions btwn Base + Derived
- new test driver is needed:

```
void better_test_driver(){
  Base base;
  Derived derived;
  base.foo();        // test case 1
  derived.foo();     // test case 2
  base.bar();        // test case 3
  derived.bar();     // test case 4
}
```

e.g. # test sets needed when integration testing w/ polymorphism



cov isn't enough to ensure complete inheritance testing
- ↳ 100% inheritance context cov requires code to be fully exercised in each context
- ↳ must test method sequences (i.e. interactions btwn methods)

ABSTRACT CLASSES

can't instantiate abstract classes, but must test them for fcnal compliance (i.e. module's compliance w/ documented fcnal spec)

abstract test pattern provides way to build test suite that can be reused across descendants
- ↳ must write abstract test class for every interface + abstract class

- abstract test must have test cases that can't be overridden + abstract factory method for creating instances of class to be tested
  - ↳ must write concrete test class for every implementation
    - concrete test class should extend abstract test class + implement factory method
  - ↳ e.g. stats app

Interface:

```
public interface StatPak
  {
    public void reset();
    public void addValue(double x);
    public double getN();
    public double getMean();
    public double getStdDev();
  }
```

Abstract test class:

```
public abstract TestStatPak {
  private StatPak statPak;

  @Before
  public final setUp() throws Exception {
    statPak = createStatPak();
    assertNotNull(statPak);
  }

  // Factory Method. Every test class of a
  // concrete subclass K must override this
  // to return an instance of K
  public abstract StatPak createStatPak();
  //Continued in next slide…
```

```
@Test
public final void testMean() {
  statPak.addValue(2.0);
  statPak.addValue(3.0);
  statPak.addValue(4.0);
  statPak.addValue(2.0);
  statPak.addValue(4.0);
  assertEquals("Mean value of test data should be 3.0",
               3.0,statPak.getMean());
}

@Test
public final void testStdDev() { ... }}
```

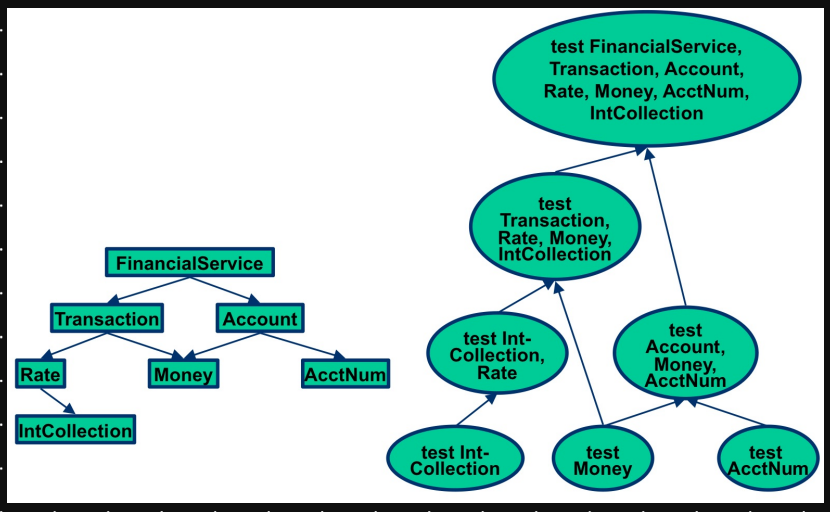
Concrete test class:

```
public class TestSuperSlowStatPak
  extends TestStatPak {

  public StatPak createStatPak()
  {
    return new SuperSlowStatPak();
  }
}
```

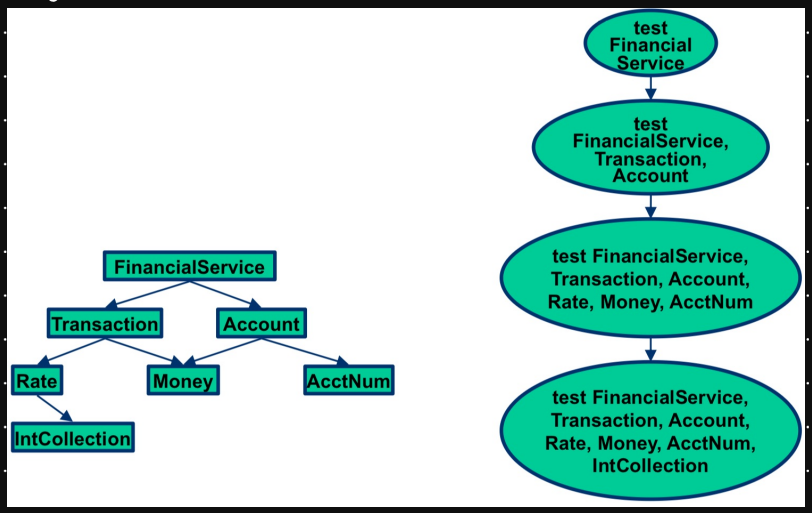
- tests that define fcnality of interface belong in abstract test class
- tests specific to implementation belong in concrete test class

CLUSTER INTEGRATION

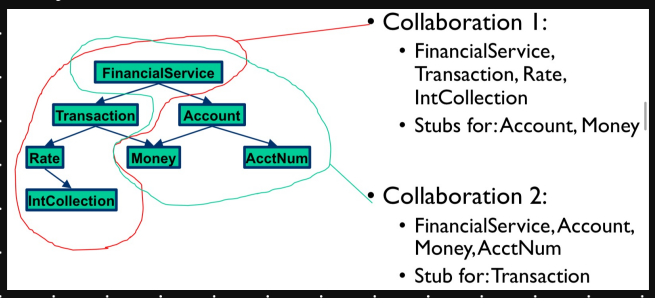
- cluster integration (i.e. inter-class) testing needs class dependency tree
- big bang method only suitable when cluster is stable, meaning only a few elmts are added/changed
  - ↳ small cluster
  - ↳ components are tightly coupled
- bottom-up is most widely used technique
  - ↳ integrate classes starting from leaves + moving up in dependency tree
  - ↳ e.g.



- top-down is also widely used
  - ↳ integrate classes starting from top + moving down dependency tree to leaves
  - ↳ e.g.



- scenario-based cluster integration is describing interaction / collab of classes
  1) map collabs onto dependency tree
  2) choose seq to apply collabs
     - e.g. simplest 1st, w/ fewest stubs 1st
  3) develop + run test exercising collab
  - ↳ e.g.



INTEGRATION ORDER

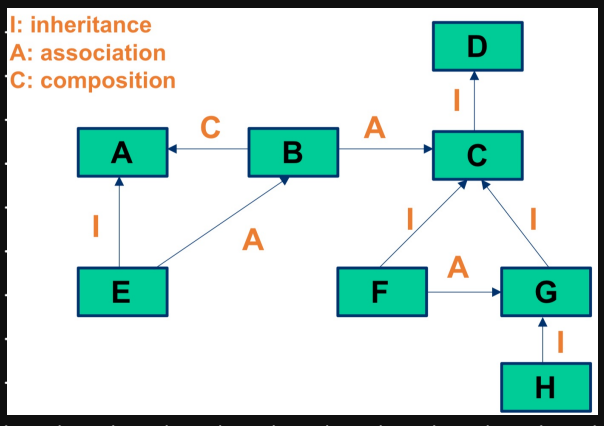
- integration must be done in stepwise manner, but that leads to stubs

- can't always create stub that's simpler than acc code
- can't automate stub gen b/c it requires understanding of simulated fcns' semantics
- fault potential for stubs may be ≥ acc fcn
- minimizing #stubs saves costs

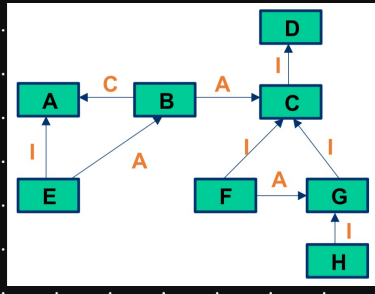
class dependency graphs usually form complex networks
- strong connectivity
- cyclic interdependencies

Kung et al. strategy prods partial ordering of testing lvls based on class diagrams
- classes under test at curr lvl should only depend on classes prev tested (i.e. lower lvl)
- no stub needed b/c only prev tested classes are needed
- topological sorting

obj rltn diagram (ORD) for program P is edge-labelled graph where nodes rep classes in P & edges rep inheritance, composition, & association rltns
- uses static info only
- for any 2 classes C1 & C2:
  - C1 $\xrightarrow{I}$ C2 : C1 is subclass of C2
  - C1 $\xrightarrow{C}$ C2 : C1 is composed of C2 (i.e. C1 contains ≥ instances of C2)
  - C1 $\xrightarrow{A}$ C2 : C1 associates w/ C2
- e.g.



class firewall (CFW) identifies effect of class change at class lvl
- CFW(X) : set of classes that could be affected by change to X
  - i.e. classes that should be retested if class X changes
- assuming ORD isn't modified, CFW(X) must includes subclasses of, classes composed w/, & classes associated w/ X
- e.g.



Class X	CFW(X)
A	B, E
B	E
C	B, E, F, G, H
D	B, C, E, F, G, H
E	
F	
G	F, H
H	

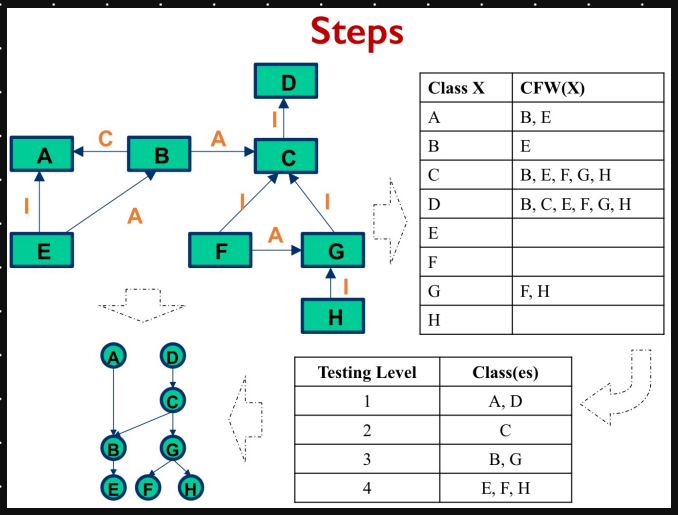
CFW derivation:
- ↳ $R = \{ \langle C1, C2 \rangle \mid$ directly labelled edge from C1 to C2 in ORD$\}$
- ↳ $CFW(X) = \{ C_k \mid \langle C_k, X \rangle \in R^+ \}$
  - R^+ is irreflexive (i.e. can't related to itself), transitive closure of R
  - contains all classes C_k st there's directed path from C_k to X in ORD
- ↳ changed class set is $S = \{X_1, \ldots, X_q\}$
  - $CFW(S) = CFW(X_1) \cup \ldots \cup CFW(X_q)$

desirable test order minimizes # stubs
- ↳ test independent classes 1st, then test dependent ones based on rltnships

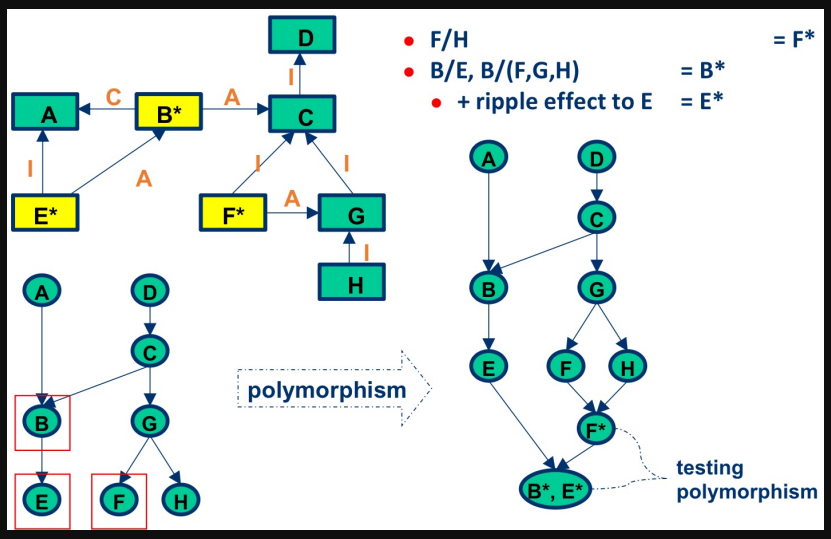
acyclic ORDs can gen test order that ensures X is tested before all classes of CFW(X)
- ↳ no stubs needed
- ↳ partial order is when several classes are at same lvl
- ↳ e.g.

Class X	CFW(X)
A	B, E
B	E
C	B, E, F, G, H
D	B, C, E, F, G, H
E	
F	
G	F, H
H	

Testing Level	Class(es)
1	A, D
2	C
3	B, G
4	E, F, H



dynamic (i.e. execution time) rltnships not considered by Kung et al
- ↳ e.g.



° F may dynamically associate w/ H due to polymorphism so F* should be tested after H

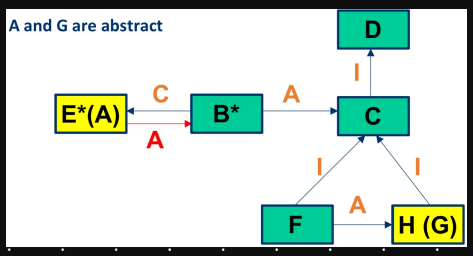
some testing lvls become partly infeasible b/c of abstract classes

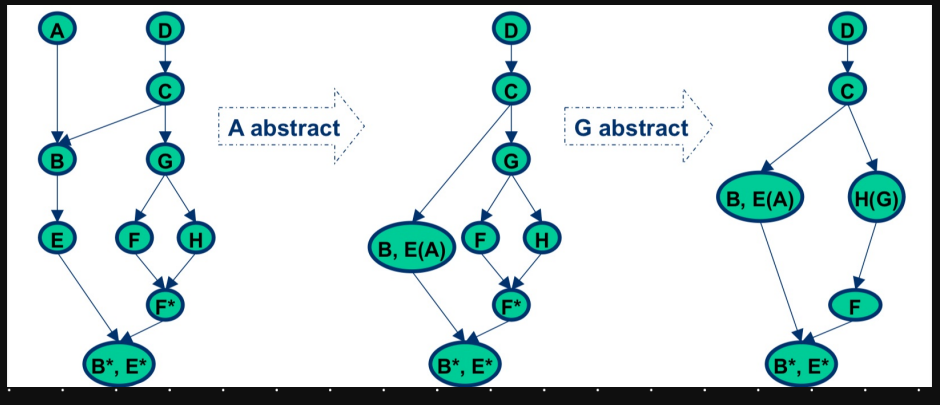
↳ e.g.

- If A is an abstract class, testing it at level 1 is unworkable
- Testing B at level 3 now requires subclass E to be instantiated (instead of A)
- E cannot be tested after B

Testing Level	Class(es)
1	A, D
2	C
3	B, G
4	E, F, H

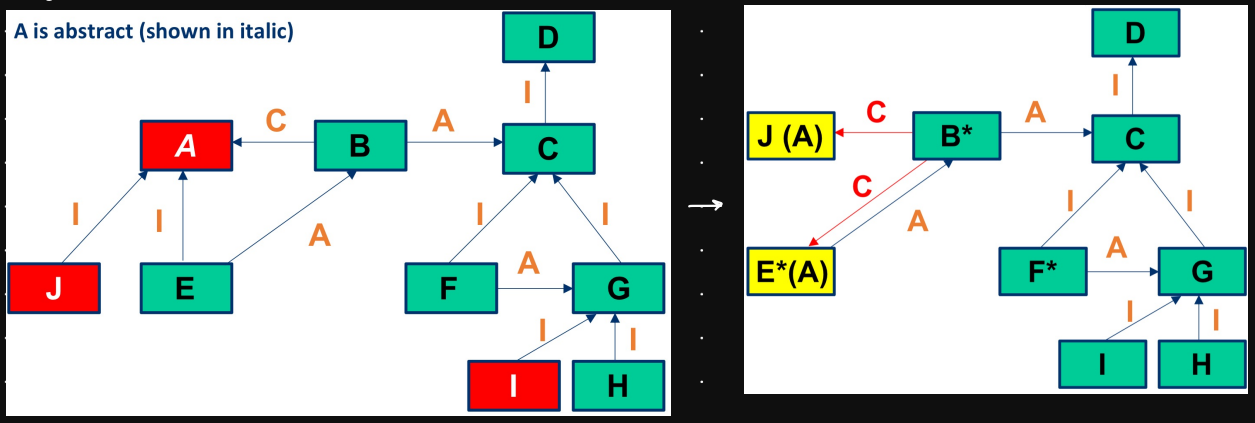
↳ e.g.

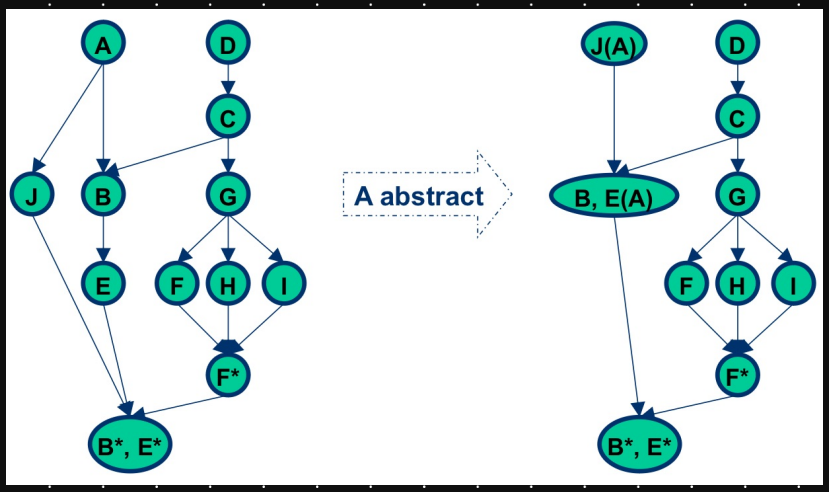




- if A's abstract, A's source code is tested w/ its subclass E so B + E are now interdependent + tested tgt
- if G's abstract, G's source code is tested w/ its subclass H so F doesn't need to test for polymorphism anymore

↳ e.g.





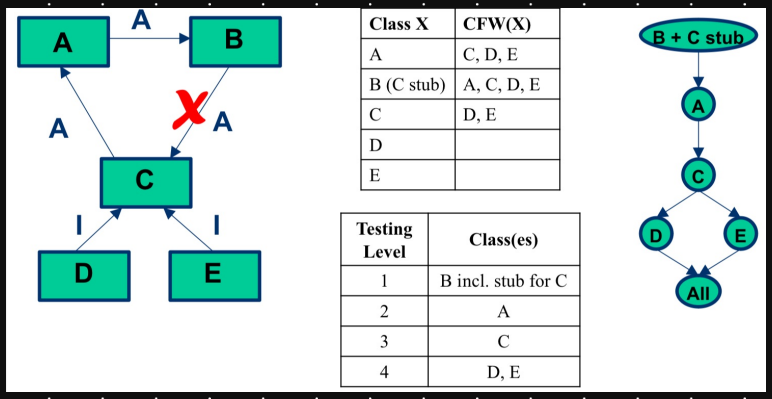
ORDs are often cyclic in practice so topological sorting can't be applied

↳ cluster: maximal set of nodes that are mutually reachable thru R^+

- i.e. strongly connected component

↳ cycle breaking: identify + remove edge from cluster + repeat until graph in cluster is acyclic

- best to remove association rltn edge
  - ↳ thm: every directed cycle contains at least 1 association edge
  - ↳ select 1 that leads to simplest stub
- don't remove inheritance edge b/c many inherited, non-overridden methods are re-tested in subclasses
  - ↳ can't stub parent class
- composition edges usually involve intense interactions
  - ↳ not good for stubbing
- e.g. cycle breaking w/ polymorphism



Class X	CFW(X)
A	C, D, E
B (C stub)	A, C, D, E
C	D, E
D	
E	

Testing Level	Class(es)
1	B incl. stub for C
2	A
3	C
4	D, E

class dependency info in ORD can come from reverse engineering or design documentation (e.g. UML)

↳ straightfwd for UML

↳ source code can be more complicated

- dependencies that aren't I, C, or A must still be labelled w/ A edge in ORD
  - → e.g. passing in obj as param
  - → e.g. creating obj as local var in method
  - → e.g. factory creates obj + returns it to caller

DATA SLICE TESTING

DATA SLICES

testing class aims at finding **seq of ops** for which class is in state / prods output that's incorrect
- ↳ state-based testing across classes + data flow-based testing across methods
- ↳ usually impossible to test for all seqs

e.g. CCoinBox

```
class CCoinBox {
  unsigned totalQtrs;
  unsigned curQtrs;
  unsigned allowVend;

public:
  CCoinbox() {Reset();}
  void AddQtrs();
  void ReturnQtrs() {curQtrs = 0;}
  unsigned isAllowedVend() {
    return allowVend;
  }
  void Reset() {
    totalQtrs = 0;
    allowVend = 0;
    curQtrs = 0;
  }
  void Vend();
};
```

```
void CCoinBox::AddQtr() {
  curQtrs = curQtrs + 1;
  if (curQtrs > 1)
    allowVend = 1;
}

void CCoinBox::Vend() {
  if(isAllowedVend()) {
    totalQtrs = totalQtrs + 2;
    if (curQtrs > 1)
      curQtrs = curQtrs - 2;
    if (curQtrs < 2)
      allowVend=0;
  }
}
```

fault: omission, should be void ReturnQtrs() {curQtrs = 0; allowVend = 0;}

- ↳ {AddQtrs(), AddQtrs(), ReturnQtrs(), Vend();} → free drinks
- ↳ **state dependent fault**, meaning only some op seqs lead to unexpected behaviour
- ↳ method testing can't detect

concrete state of obj at single instant of time is equiv to aggregated state of all of its data members at that instant

correctness of class depends on:
- ↳ data members correctly repping intended state of obj
- ↳ member fcns correctly manipulating rep of obj

data slice: portion of class w/ single data member + set of member fcns that can manipulate vals associated w/ this member
- ↳ class is composition of data slices
- ↳ test 1 slice at a time
  - for each, test possible seqs of methods in that slice
    - → i.e. class partial correctness

data slice testing is code-based approach so there's many potential issues
- ↳ faulty code where method should access data member but doesn't
- ↳ identifying legal methods seqs
- ↳ large #method seqs

MADUM

generate **MaDUM (Minimal Data Member Usage Matrix)** to minimize redundancy when doing data slicing testing

↳ n x m matrix where n is #data members & m is #member methods
↳ reports usage of data members by methods
↳ diff usage categories:
- ctors
- reporters / read-only
- transformers
- others

↳ account for indirect use thru other member fcns

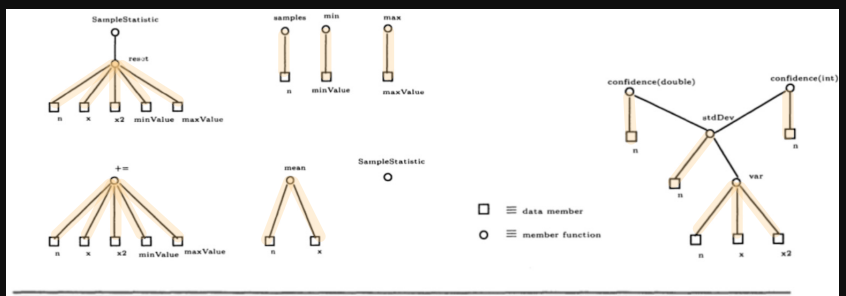
e.g. SampleStatistic

```
class SampleStatistic {
  friend TestSS;

protected :
  int n;
  double x;
  double x2;
  double minValue;
  double maxValue;

public :
  sampleStatistic();
  virtual ~SampleStatistic();
  virtual void reset();
  virtual void operator += (double);
  int samples();
  double mean();
  double stdDev();
  double var();
  double min();
  double max();
  double confidence(int p_percentage);
  double confidence (double p_value);
  void error(const char * msg);
};
```

test driver

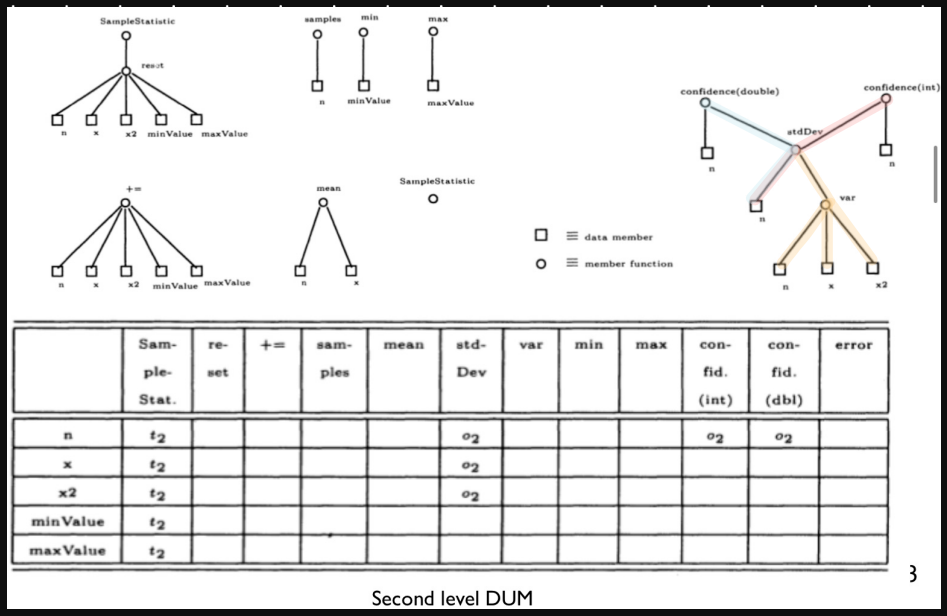
```
SampleStatistic::mean() {
 if (n>0)
   return (x/n);
 else
   return (0.0);
}

double SampleStatistics::stdDev() {
  if (n<=0 || this->var()<=0)
    return(0);
  else
    return (double)sqrt(var());
 }

SampleStatistic::operator+=(double value) {
 n += 1;
 x += value;
 x2 += (value * value);
 if(minValue > value)
   minValue = value;
 if(maxValue < value)
   maxValue = value;
}

double SampleStatistics::var() {
  if (n>1)
    return ((x2 - ((x*x)/2))/(n-1));
 }
```

↳ creating call graph:



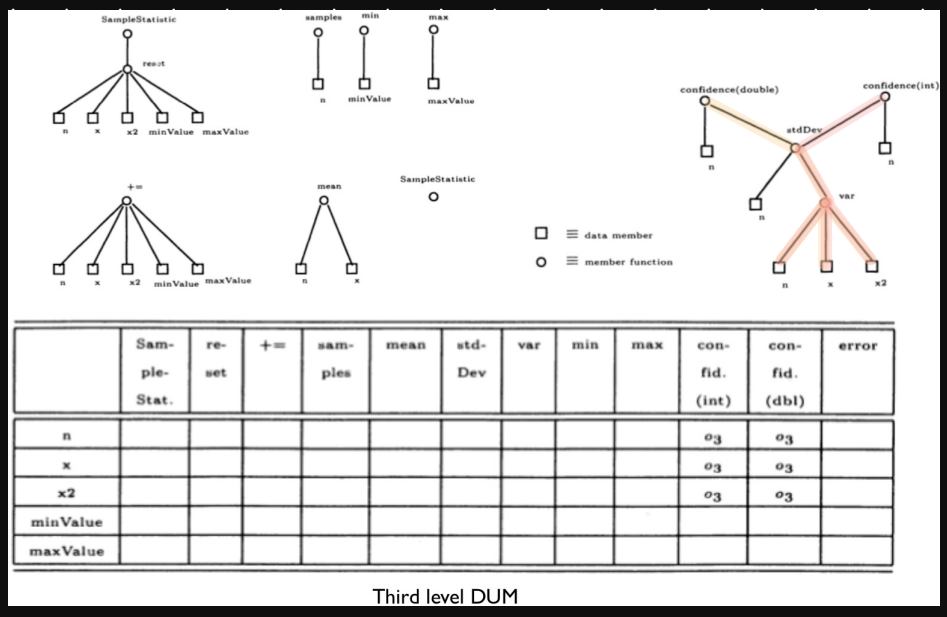
	Sample-Stat.	reset	+=	samples	mean	stdDev	var	min	max	confid. (int)	confid. (dbl)	error
n		t_1	t_1	r_1	o_1	o_1	o_1			o_1	o_1	
x		t_1	t_1		o_1		o_1					
x2		t_1	t_1				o_1					
minValue		t_1	t_1					r_1				
maxValue		t_1	t_1						r_1			

First level DUM



	SampleStat.	reset	+=	samples	mean	stdDev	var	min	max	confid. (int)	confid. (dbl)	error
n	t_2					o_2				o_2	o_2	
x	t_2					o_2						
x2	t_2					o_2						
minValue	t_2											
maxValue	t_2											

Second level DUM



	SampleStat.	reset	+=	samples	mean	stdDev	var	min	max	confid. (int)	confid. (dbl)	error
n										o_3	o_3	
x										o_3	o_3	
x2										o_3	o_3	
minValue												
maxValue												

Third level DUM

↳ DUM:

	Sample Statistic	reset	+=	samples	mean	stdDev	var	min	max	confidence (int)	confidence (dbl)	error
n	t R	t	t	r	o	o	o			o	o	
x	t R	t	t		o	o R	o			o R	o R	
x2	t R	t	t			o R	o			o R	o R	
minValue	t R	t	t					r				
maxValue	t R	t	t						r			

R = redundant

- SampleStatistic() just calls reset(), which alr tests n, x, x2, minValue, + maxValue.
- stdDev() only accesses x + x2 thru var()
- confidence(int) + confidence(double) only access x + x2 thru stdDev() → var()

↳ MaDUM:

	Sample-Stat.	reset	+=	samples	mean	std-Dev	var	min	max	confidence (int)	confidence (dbl)	error
n		t	t	r	o	o	o			o	o	
x		t	t		o		o					
x2		t	t				o					
minValue		t	t					r				
maxValue		t	t						r			
	C	T	T	R	O	O	O	R	R	O	O	O

- **test procedure** using classification of methods:
  1). reporters
  2). ctors
  3). transformers
  4). others
- to **test reporters**:
  ↳ state of data member that method reports on isn't altered
  ↳ create random seq of ops, except for reporters, of data slice + append it w/ reporter method
- to **test ctors**:
  ↳ test all data members are correctly initialized in correct order
  ↳ run ctor + append reporter methods for each data member
  ↳ verify if correct initial state (i.e. state invariant)
- to **test transformers**:
  ↳ test interactions btwn methods
  ↳ for each data slice:
    1). instantiate obj under test w/ctor
    2). create seqs (e.g. all legal permutations of transformer + other methods)
    3). append reporter methods
  ↳ for each member fcn, execute all control flow paths where data slice is manipulated
- to **test others**:
  ↳ treated as any other fcn so only fonality as stand-alone entities needs to be tested
  ↳ apply any std test technique

- **derived class** may add facilities or modify ones provided by base class
  ↳ 2 extreme testing options:
    - flatten derived class + retest all slices of base class in new context
    - only test new/redefined slices
- **Bashir + Goel's strat** is to extend MaDUM of base class to gen $\text{MaDUM}_{\text{derived}}$.
  ↳ take MaDUM of base class
  ↳ add row for each newly/re-defined data member of derived class
  ↳ add col for each newly/re-defined member fcn
- e.g. SampleHistogram

```
class SampleHistogram : public SampleStatistic {
protected:
  short howmanybuckets;
  int *bucketCount;
  double *bucketLimit;
public:
  SampleHistogram(double low, double hi, double bucketWidth =
                  -1.0);
  ~SampleHistogram();
  virtual void reset();
  virtual void operator+=(double);
  int similarSamples(double);
  int buckets();
  double bucketThreshold(int i);
  int inBucket(int i);
  void printBuckets(ostream&);
}
```

↳ MaDUM derived :

	Samp-Stat	reset	+=	sam-ples	mean	stdDev	var	min	max	conf. (int)	conf. (dbl)	error	Samp-Histo	reset	+=	similar-samples	buckets	bucket-Thres-hold	inBucket	print-buckets
n		t	t	r	o	o	o			o	o			t	t					
x		t	t		o		o							t	t					
x2		t	t				o							t	t					
minValue		t	t					r						t	t					
maxValue		t	t						r					t	t					
howMany-Buckets													t	o	o	o	r	o	o	o
bucket Count													t	t	t	r			r	o
bucket Limit													t		o	o		r		o
	C	T	T	R	O	O	O	R	R	O	O	O	C	T	T	O	R	O	O	O

- 3 new rows for local data members
- 8 new cols for local member fcns
  → 2 redefined ones (reset, +=)

↳ quadrants :

	Samp-Stat	reset	+=	sam-ples	mean	stdDev	var	min	max	conf. (int)	conf. (dbl)	error	Samp-Histo	reset	+=	similar-samples	buckets	bucket-Thres-hold	inBucket	print-buckets
n	t	t	t	r	o	o	o			o	o			t	t					
x	t	t	t		Quadrant two									t	Quadrant one					
x2	t	t	t				o							t	t					
minValue	t	t	t					r						t	t					
maxValue	t	t	t						r					t	t					
howMany-Buckets					Quadrant three								t	o	o	o	r	o	o	o
bucket Count													t	t	Quadrant four					o
bucket Limit													t		o	o		r		o
	C	T	T	R	O	O	O	R	R	O	O	O	C	T	T	O	R	O	O	O

- Q1 : rltnship btwn member fcns of derived class + inherited data members
- Q2 : rltnship btwn inherited attrs from base class
  → use DUM w/ redundant entries to cover cases where member fcn of derived class uses some member of base class indirectly

- Q3: rltnship btwn data members of derived class + inherited member fcns
- Q4: rltnship btwn local attrs of derived class only

order to fill quadrants in:

1) Q2 should alr be filled w/ test history of base class
2) Q4
3) Q3
4) Q1

test procedure for derived class:

↳ test local attrs similarily to base class testing
↳ retest inherited attrs (i.e. data slice for each data member) if #entries in its MaDUM row has increased btwn MaDUM + MaDUM derived
- e.g. SampleHistogram must retest {n, x, x2, minValue, maxValue} b/c reset() and t=() affect each inherited data member

BLACKBOX TESTING

TEST CASE DERIVATION

- black-box testing: testing w/o details of underlying code
  - ↳ based on system's specs instead of struct
  - ↳ cov can also be used for fcnal black-box testing
- rigorous specs help test case gen. + test oracles
  - ↳ categorize inputs
  - ↳ derive expected outputs
- advantages of black-box testing:
  - ↳ no need for source code
  - ↳ wide applicability (e.g. used for unit + system-lvl testing)
- disadvantages of black-box testing:
  - ↳ doesn't test hidden fcns
- error guessing is ad-hoc approach based on exp
  1) make list of possible errors
  2) this yields error model
  3) design test cases to cover error model
  - ↳ e.g. sorting arr fcn

    - Error model:
      - Empty array
      - Already sorted array
      - Reverse-sorted array
      - Large unsorted array
      - ...
- test case derivation from fcnal requirements involves restatement of requirements to make them testable (i.e. a form of error guessing)
  - ↳ put into bullet pts
    - enumerate single requirements
    - group related requirements
  - ↳ for each requirement, provide:
    - 1 test case showing it fcning
    - 1 test case trying to falsify smth
    - test boundaries + constraints when possible
  - ↳ e.g. movie rental system

1.1 The system shall allow a movie to be rented if at least one of its copies has not been rented.

1.2 At the time of a movie rental, the system shall set the due date of the movie copy to two days from the rental date.

Test situations:

- **Attempt to rent an available movie when at least one copy is available (check for correct due date)** → legal
- **Attempt to rent the last available movie when only one copy is available (check for correct due date)** → legal
- **Attempt to rent an unavailable movie when no copy is available** → illegal

1.3 At the time of a movie rental, the system shall make the movie copy unavailable for another rental.

Test situations:

- **Attempt to rent the same copy again immediately after it was already rented** → illegal
- **Attempt to rent a different copy of the same movie immediately after another copy of the movie was rented (i.e., initially exactly two copies are available)** → legal

1.4 Upon a status request for a movie, the system shall show the current borrowers of all copies of the movie.

Test situations:

- **Attempt to display the status of a movie where all copies are currently rented**
- **Attempt to display the status of a movie where all copies are currently not rented**
- **Attempt to display the status of a movie which was just returned**

test case derivation from use cases requires that for each use case:

1) develop scenario graph
2) determine all possible scenarios
3) analyze & rank scenarios
4) gen test cases from scenarios to meet cov goal
5) execute test cases

scenario graph is gen from use case & is a visual rep of possible seqs of actions, events, & conds in program

↳ node: pt where we wait for event to occur
- single starting node
- end of use case is finish node

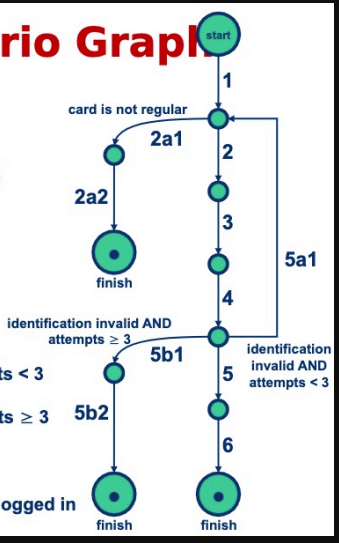
↳ edge: event occurrence
- may include conds

↳ scenario: path from starting to finish node

↳ e.g.

Example: Scenario Graph

Title: Authenticate User
Actors: User
Precondition: System is ON, User is not logged in
Main Scenario:
1: User inserts a card
2: System asks for personal identification number (PIN)
3: User types PIN
4: System validates user identification
5: System displays a welcome message to user
6: System ejects card
Alternatives:
2a: Card is not regular
2a1: System emits alarm
2a2: System ejects card
5a: User identification is invalid AND number of attempts < 3
5a1: Go back to step 2
5b: User identification is invalid AND number of attempts ≥ 3
5b1: System emits alarm
5b2: System ejects Card
Post-condition (for main): User is logged in
Post-condition (for alternatives 2a and 5b): User is not logged in



↳ cond of an alt Na is evaled at node & if it's true, then step N

isn't executed + instead go to Nal
↳ #times loop can be taken needs to be restricted for finite #scenarios

rank scenarios if too many
↳ can base on fcnality + criticality
↳ always include main scenario

e.g. test case

Test Case: TC1
Goal: Test the main course of events for Authenticate User
Scenario Reference: S1
Setup: Create a card #2411 with PIN #5555 as valid user identification, System is ON
Course of test case:

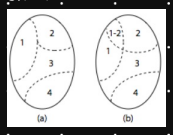
	External Event	Reaction	Comment
1	User inserts card #2411	System asks for personal identification number (PIN)	
2	User types PIN #5555	System validates user identification; System displays a welcome message to user; System ejects card	

Pass criteria: User is logged in

EQUIVALENCE PARTITIONING

equivalence classes (ECs): partition of input set according to spec
↳ classes behave same way, are mapped to similar output, test same thing, + reveal same bugs
↳ appropriate when input data is in terms of ranges + sets of discrete vals.

aim for completeness (i.e. entire input set covered), but avoid redundancy
↳ aim for a + avoid b



weak EC testing (WECT): choose 1 var val from each EC
strong EC testing (SECT): test all class interactions by using Cartesian product of partition subsets.

e.g.

ECT Test Cases

Test Case	A	B	C
WE1	a1	b1	c1
WE2	a2	b2	c2
WE3	a3	b3	c1
WE4	a1	b4	c2

WECT (max number of ECs of all variables, i.e., 4)

SECT (3*4*2 = 24)

Test Case	A	B	C
SE1	a1	b1	c1
SE2	a1	b1	c2
SE3	a1	b2	c1
SE4	a1	b2	c2
SE5	a1	b3	c1
SE6	a1	b3	c2
SE7	a1	b4	c1
SE8	a1	b4	c2
SE9	a2	b1	c1
SE10	a2	b1	c2
SE11	a2	b2	c1
SE12	a2	b2	c2
SE13	a2	b3	c1
SE14	a2	b3	c2
SE15	a2	b4	c1
SE16	a2	b4	c2
SE17	a3	b1	c1
SE18	a3	b1	c2
SE19	a3	b2	c1
SE20	a3	b2	c2
SE21	a3	b3	c1
SE22	a3	b3	c2
SE23	a3	b4	c1
SE24	a3	b4	c2

↳ 3 input vars of domains A, B, + C

- $A = A_1 \cup A_2 \cup A_3$ where $a_n \in A_n$
- $B = B_1 \cup B_2 \cup B_3$ where $b_n \in B_n$
- $C = C_1 \cup C_2 \cup C_3$ where $c_n \in C_n$

e g. NextDate

- NextDate is a function with three variables – month, day, year – and returns the date of the day after the input date; limitation: 1812 ≤ year ≤ 2012
- Treatment summary:
  - If it is not the last day of the month, the next date function will simply increment the day value
  - At the end of a month, the next day is 1 and the month is incremented
  - At the end of the year, both the day and the month are reset to 1, and the year incremented
  - Finally, the problem of leap year makes determining the last day of a month interesting (leap year if divisible by 4 and not a century year; century year is leap year if multiple of 400)

Input Condition	Valid ECs	Decomposed Valid ECs
month	1 ≤ month ≤ 12	Months with 30 days (1)
		Months with 31 days (2)
		February (3)
day	1 ≤ day ≤ 31	1 ≤ day ≤ 28 (4)
		day = 29 (5)
		day = 30 (6)
		day = 31 (7)
year	1812 ≤ year ≤ 2012	year = 1900 (8)
		1812 ≤ year ≤ 2012 and year ≠ 1900 and year mod 4 = 0 (9)
		1812 ≤ year ≤ 2012 and year mod 4 ≠ 0 (10)

Decomposition/refinement can be done based on how input is treated:

- **Day incremented by 1 unless it's the last day of a month ⊔ depends on month (30, 31, February)**
- **Problem of leap year**

↳ WECT w/4 test cases, which is max partition:

Test Case	ECs	Month	Day	Year	Output
WE1	1, 4, 8	6	14	1900	6/15/1900
WE2	2, 5, 9	7	29	1912	7/30/1912
WE3	3, 6, 10	2	30	1913	Invalid Input
WE4	1, 7, 8	6	31	1900	Invalid Input

↳ SECT w/ 3(4)(3) = 36 test cases:

Case ID	Month	Day	Year	Expected Output
SE1	6	14	1900	6/15/1900
SE2	6	14	1912	6/15/1912
SE3	6	14	1913	6/15/1913
SE4	6	29	1900	6/30/1900
SE5	6	29	1912	6/30/1912
SE6	6	29	1913	6/30/1913
SE7	6	30	1900	7/1/1900
SE8	6	30	1912	7/1/1912
SE9	6	30	1913	7/1/1913
SE10	6	31	1900	ERROR
SE11	6	31	1912	ERROR
SE12	6	31	1913	ERROR
SE13	7	14	1900	7/15/1900
SE14	7	14	1912	7/15/1912
SE15	7	14	1913	7/15/1913
SE16	7	29	1900	7/30/1900
SE17	7	29	1912	7/30/1912
SE18	7	29	1913	7/30/1913
SE19	7	30	1900	7/31/1900
SE20	7	30	1912	7/31/1912
SE21	7	30	1913	7/31/1913
SE22	7	31	1900	8/1/1900
SE23	7	31	1912	8/1/1912
SE24	7	31	1913	8/1/1913
SE25	2	14	1900	2/15/1900
SE26	2	14	1912	2/15/1912
SE27	2	14	1913	2/15/1913
SE28	2	29	1900	ERROR
SE29	2	29	1912	3/1/1912
SE30	2	29	1913	ERROR
SE31	2	30	1900	ERROR
SE32	2	30	1912	ERROR
SE33	2	30	1913	ERROR
SE34	2	31	1900	ERROR
SE35	2	31	1912	ERROR
SE36	2	31	1913	ERROR

must extend SECT to include both valid (E_i) + invalid (U_i) inputs if error conds are high priority
↳ SECT assumes vars are indep so we gen error test cases thru dependencies
↳ e.g. NextDate

Input Condition	Valid ECs	Invalid ECs
month	1 ≤month ≤12	month < 1 (11)
		month > 12 (12)
day	1 ≤day ≤31	day < 1 (13)
		day > 31 (14)
year	1812 ≤year ≤2012	year < 1812 (15)
		year > 2012 (16)

- **Invalid ECs lead to additional test cases (one per invalid EC)**

Test Case	ECs	Month	Day	Year	Output
IEC1	11	0	14	1900	Invalid Input
IEC2	12	13	14	1900	Invalid Input
IEC3	13	6	0	1900	Invalid Input
IEC4	14	6	32	1900	Invalid Input
IEC5	15	6	14	1811	Invalid Input
IEC6	16	6	14	2013	Invalid Input

heuristics for identifying EC:
↳ if input is range of valid vals, define:
- 1 valid EC (within range)
- 2 invalid ECs (1 outside each end of range)

↳ if input is #N of valid vals, define:
- 1 valid EC
- 2 invalid ECs (none + >N)

↳ if input is elmt from set of valid vals, define:
- 1 valid EC (within set)
- 1 invalid EC (outside set)

↳ if input is necessary cond, define:
- 1 valid EC (satisfies cond)
- 1 invalid EC (doesn't satisfy cond)

↳ if elmts in EC aren't handled in identical manner, subdivide EC
↳ create equivalence partitions for default, empty, blank, null, 0, + none conds

until all valid ECs covered, write new test case that covers as many uncovered ECs as possible
until all invalid ECs covered, write new test case that covers only 1 uncovered EC

advantages of equivalence partitioning:
↳ small # test cases needed
↳ probability of uncovering defects higher than random test suite of same size

disadvantages of equivalence partitioning:
↳ strongly typed langs don't need test cases for some invalid inputs
↳ specs don't always define expected output for invalid test cases
↳ doesn't work well when input vars aren't indep
↳ impractical when # inputs is large b/c SECT tests every combo of input ECs

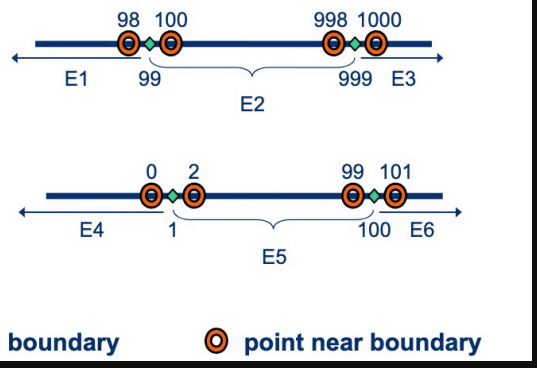
boundary val analysis (BVA): focuses on edge of planned operational limits
↳ e.g.

- First / last, start / finish
- Min / max, under / over, lowest / highest
- Empty / full
- Slowest / fastest, smallest / largest, shortest / longest
- Next-to / farthest-from
- Soonest / latest

e.g. findPrice

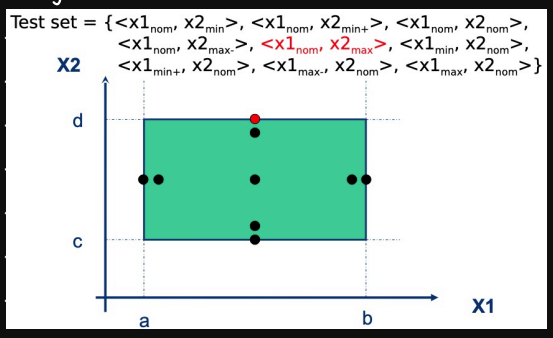
- Function findPrice() has two parameters:
  - An item code must be in the range 99..999
  - A quantity in the range 1..100
- Equivalence classes for item code:
  - E1: values less than 99
  - E2: values in the range
  - E3: values greater than 999
- Equivalence classes for quantity:
  - E4: values less than 1
  - E5: values in the range
  - E6: values greater than 100

- Equivalence classes and boundaries for findPrice()



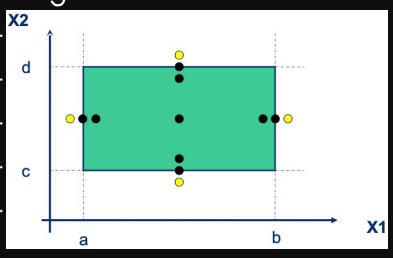
for BVA, use input vars at:
- ↳ min
- ↳ min+ (just above)
- ↳ nom (i.e. nominal)
- ↳ max- (just below)
- ↳ max

hold vals of all but 1 var at nom, then test that 1 var at extremes
- ↳ for each boundary cond, include boundary val in at least 1 valid test case
- ↳ e.g.



- ↳ for w/ n vars needs (4n + 1) test cases

robustness testing: add val just beyond boundary in at least 1 invalid test case
- ↳ i.e. add min-, max+
- ↳ e.g.

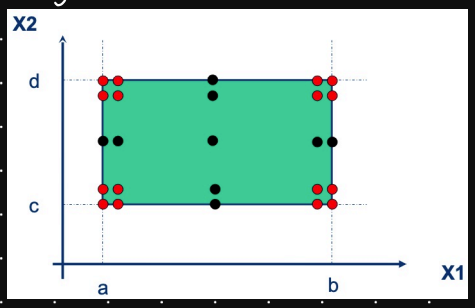


↳ $6n+1$ test cases

worst case testing: Cartesian product of {min, min+, nom, max-, max} so that n vars have extreme vals

↳ good strat when physical vars have numerous interactions + failure is costly

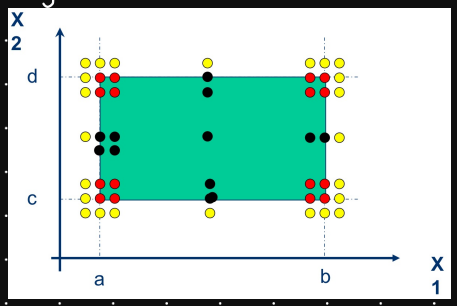
↳ e.g.



↳ 5^n test cases

robust worst case testing: factor in min-, max+

↳ e.g.



↳ 7^n test cases

CATEGORY PARTITIONING

category partitioning: systematic, spec-based methodology that prods formal test spec from informal fcnal spec

↳ integrates EC testing, BVA, + adds **constraints** btwn classes

steps:

1) decompose fcnal spec into **fcnal units** that can be independently tested
2) examine each fcnal unit
   - identify **params** + **environmental conds** (i.e. characteristics of system's state)
3) find **categories** for each param + env cond
4) subdivide categories into **choices**, similar to equivalence partitioning
   - i.e. vals of domains
5) determine constraints among choices
   - e.g. OS must be Windows if browser is Edge
6) create **test frames** (i.e. set of choices, 1 from each category)
7) create test cases using test frames w/ explicit val for each choice

e.g. find cmd

- Syntax: find <pattern> <file>
- Function: The find command is used to locate one or more instances of a given pattern in a text file. All lines in the file that contain the pattern are written to standard output. A line containing the pattern is written only once, regardless of the number of times the pattern occurs in it.
- The pattern is any sequence of characters whose length does not exceed the maximum length of a line in the file. To include a blank in the pattern, the entire pattern must be enclosed in quotes ("..."). To include a quotation mark in the pattern, two quotes in a row ("") must be used.

↳ params:

- **Pattern size:**
  1) Empty
  2) Single character
  3) Many characters
  4) Longer than any line in the file
- **Quoting:**
  1) Pattern is quoted
  2) Pattern is not quoted
  3) Pattern is improperly quoted
- **Embedded blanks:** ← category
  1) No embedded blank
  2) One embedded blank ← choices
  3) Several embedded blanks
- **Embedded quotes:**
  1) No embedded quote
  2) One embedded quote
  3) Several embedded quotes
- **File name:**
  1) Good file name
  2) No file with this name

↳ env conds:

- **Number of occurrence of pattern in file:**
  1) None
  2) One
  3) More than one
- **Pattern occurrences on target line:** ← category
  1) None
  2) One ← choices
  3) More than one

↳ total test frames = 4(3)(3)(3)(2)(3)(3) = 1944

↳ impossible test frame:

- Pattern size: empty
- Quoting: pattern is quoted
- Embedded blanks: several embedded blanks → can't embed blanks in empty pattern
- Embedded quotes: no embedded quote
- File name: good file name
- Number of occurrence of pattern in file: none
- Pattern occurrence on target line: one → contradiction

use constraints like properties + selectors w/ choices to elim contradictory frames

↳ property is assigned to choice w/ format [property <name>, ...]

↳ selector makes use of properties w/ format [if <name>, ...]

↳ e.g.

E.g.: ChoiceA1 [property X]; ChoiceA2; ChoiceB1 [if X] (␣ {ChoiceA1, ChoiceB1} but not {ChoiceA2, ChoiceB1})

some annotated choices aren't combined w/ other choices to create test frame + instead, test frame contains only 1 choice/error

↳ use [error] to limit redundant frames when param/env cond causes choice to error regardless of other choices

↳ use [single] to mark choice that can be adequately tested w/ only 1 test case

- judgment by tester

e.g. find cmd w/ constraints

- **Pattern size:**
1) Empty [property = Empty]
2) Single character [property = NonEmpty]
3) Many characters [property = NonEmpty]
4) Longer than any line in the file [error]

- **Quoting:**
1) Pattern is quoted [property = Quoted]
2) Pattern is not quoted [if NonEmpty]
3) Pattern is improperly quoted [error]

- **Embedded blanks:**
1) No embedded blank [if NonEmpty]
2) One embedded blank [if NonEmpty and Quoted]
3) Several embedded blanks [if NonEmpty and Quoted]

- **Embedded quotes:**
1) No embedded quote [if NonEmpty]
2) One embedded quote [if NonEmpty]
3) Several embedded quotes [if NonEmpty] [single]

- **File name:**
1) Good file name
2) No file with this name [error]

- **Pattern occurrences on target line:**
1) None
2) One [if Match]
3) More than one [if Match] [single]

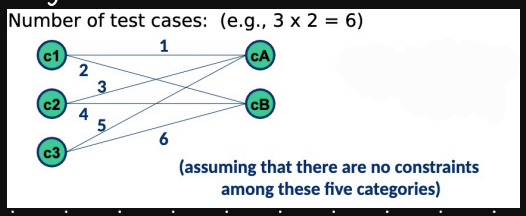
- **Number of occurrence of pattern in file:**
1) None [if NonEmpty] [single]
2) One [if NonEmpty] [property Match]
3) More than one [if NonEmpty] [property Match]

↳ 40 test frames

diff criteria for using choices:

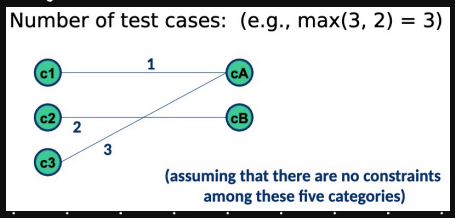
↳ all combos (AC): 1 val for every choice in category must be used w/ 1 val of every possible choice of every other category

- e.g.



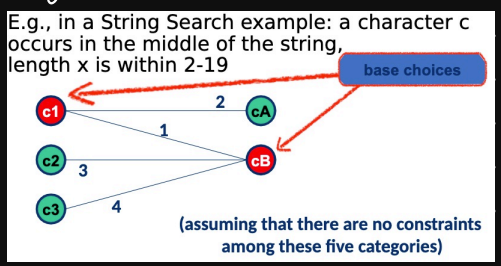
↳ each choice (EC): 1 val from each choice for each category must be used in at least 1 test case

- weaker criterion
- e.g.:

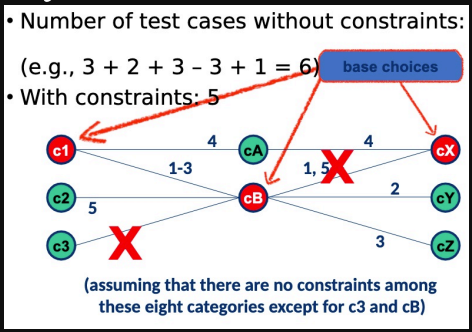


↳ base choice (BC): base choice is chosen for each category

- 1st test is base test w/ all BCs
- subsequent tests are chosen by varying 1 category at a time while other categories are held constant at BC
- BC can be simplest, smallest, 1st in ordering, most likely from user's POV, etc.
- e.g.



- e.g.



DECISION TABLES

- decision tables (DTs) are precise + compact way to model logic
  - ↳ associate conds w/ actions to perform
  - ↳ format:

	1	2	3	4	5	6
condition c1	T			F		
c2	T		F	T		F
c3	T	F	-	T	F	-
action a1	X	X		X		
a2	X				X	
a3		X		X	X	
a4			X			X

    - T: true
    - F: false
    - - : don't care (i.e. can be T/F)
    - rule is col
      - → e.g. $c1 = T \wedge c2 = T \wedge c3 = F \Rightarrow a1, a3$
- limited entry table: cond entries are bin vals
- extended entry table: cond entries have 2+ vals
- don't care vals in DTs reduce # variants
  - ↳ can be b/c:
    - inputs are necessary but have no effect
    - inputs may be omitted
    - mutually exclusive cases
      - → e.g. conds are ECs

Conditions	*R1*	*R2*	*R3*
c1: month in M1?	T	–	–
c2: month in M2?	–	T	–
c3: month in M3?	–	–	T
a1			
a2			
a3			

  - ↳ don't confuse w/ don't know cond (i.e. undefined)
    - reflects incomplete model
    - can indicate spec bugs

e.g. triangle DT

C1: a < b+c?	F	T	T	T	T	T	T	T	T	T
C2: b < a+c?	-	F	T	T	T	T	T	T	T	T
C3: c < a+b?	-	-	F	T	T	T	T	T	T	T
C4: a = b?	-	-	-	T	T	T	T	F	F	F
C5: a = c?	-	-	-	T	T	F	F	T	T	F
C6: b = c?	-	-	-	T	F	T	F	T	F	T
A1: Not a Triangle	X	X	X							
A2: Scalene										X
A3: Isosceles							X		X	X
A4: Equilateral				X						
A5: Impossible					X	X		X		

- to use DT in software testing:
  - ↳ cond entries are input / ECs of inputs
  - ↳ action entries are output / major fcnal processing portions
  - ↳ rules are test cases
  - ↳ e.g. triangle test cases

Case ID	a	b	c	Expected Output
DT1	4	1	2	Not a Triangle
DT2	1	4	2	Not a Triangle
DT3	1	2	4	Not a Triangle
DT4	5	5	5	Equilateral
DT5	?	?	?	Impossible
DT6	?	?	?	Impossible
DT7	2	2	3	Isosceles
DT8	?	?	?	Impossible
DT9	2	3	2	Isosceles
DT10	3	2	2	Isosceles
DT11	3	4	5	Scalene

- limited entry tables w/ N conds have 2^N rules (i.e. combo rule count)
  - ↳ to do rule count:
    - don't care entry doubles count for that rule
    - determine rule count for each rule / col in DT
    - sum up rule counts
  - ↳ if rule count < 2^N, there's missing rules
  - ↳ if rule count > 2^N, there's redundant or inconsistent rules
    - e.g. rule 9 is redundant w/ rule 1-4

Conditions	1-4	5	6	7	8	9
C1	T	F	F	F	F	T
C2	-	T	T	F	F	F
C3	-	T	F	T	F	F
A1	X	X	X	-	-	X
A2	-	X	X	X	-	-
A3	X	-	X	X	X	X

    - e.g. rule 9 is inconsistent w/ rule 1-4

Conditions	1-4	5	6	7	8	9
C1	T	F	F	F	F	T
C2	-	T	T	F	F	F
C3	-	T	F	T	F	F
A1	X	X	X	-	-	-
A2	-	X	X	X	-	X
A3	X	-	X	X	X	-

- e.g. NextDate problem shows dependencies in input domain
  - ↳ impossible dates can be marked as separate action
  - ↳ try 1:

C1: month in	M1	M1	M1	M1	M2	M2	M2	M2
C2: day in	D1	D2	D3	D4	D1	D2	D3	D4
C3: year in	-	-	-	-	-	-	-	-
A1: Impossible				X				
A2: Increment day	X	X			X	X	X	
A3: Reset day			X					X
A4: Increment month			X					?
A5: Reset month								?
A6: Increment								?

C1: month in	M3	M3	M3	M3	M3	M3	M3	M3
C2: day in	D1	D1	D1	D2	D2	D2	D3	D3
C3: year in	Y1	Y2	Y3	Y1	Y2	Y3	-	-
A1: Impossible				X		X	X	X
A2: Increment day		X						
A3: Reset day	X		X		X			
A4: Increment month	X		X		X			
A5: Reset month								
A6: Increment year								

M1= {month | month has 30 days}
M2= {month | month has 31 days}
M3= {month | month is February}
D1= {day | 1 ≤ day ≤ 28}
D2= {day | day = 29}
D3= {day | day = 30}
D4= {day | day = 31}
Y1= {year | year = 1900 or 2100}
Y2= {year | year is a leap year}
Y3= {year | year is a common year}

  - ↳ try 2 w/ new ECs:

M1= {month | month has 30 days}
M2= {month | month has 31 days, but not Dec.}
M3= {month | month is December}
M4= {month | month is February}
D1= {day | 1 ≤ day ≤ 27}
D2= {day | day = 28}
D3= {day | day = 29}
D4= {day | day = 30}
D5= {day | day = 31}
Y1= {year | year is a leap year}
Y2= {year | year is a common year}

C1: month in	M1	M1	M1	M1	M1	M2	M2	M2	M2	M2
C2: day in	D1	D2	D3	D4	D5	D1	D2	D3	D4	D5
C3: year in	-	-	-	-	-	-	-	-	-	-
A1: Impossible					X					
A2: Increment day	X	X	X			X	X	X	X	
A3: Reset day				X						X
A4: Increment month				X						X
A5: Reset month										
A6: Increment year										

C1: month in	M3	M3	M3	M3	M3	M4	M4	M4	M4	M4	M4	M4
C2: day in	D1	D2	D3	D4	D5	D1	D2	D2	D3	D3	D4	D5
C3: year in	-	-	-	-	-	-	Y1	Y2	Y1	Y2	-	-
A1: Impossible										X	X	X
A2: Increment day	X	X	X	X		X	X					
A3: Reset day					X			X	X			
A4: Increment month								X	X			
A5: Reset month					X							
A6: Increment year					X							

- can apply DTs when:
  - ↳ spec is given as or can be converted to DT
  - ↳ order of predicate eval doesn't affect rule interpretation or resulting action
  - ↳ order of rule eval doesn't affect resulting action
  - ↳ once rule is satisfied + action selected, no need to examine other rules
  - ↳ order of executing actions in rule doesn't matter
- before deriving test cases, ensure rules are complete (i.e. every combo of predicate vals is explicit) + consistent (i.e. every combo of predicate vals results in only 1 / 1 set of actions)
- DTs are most appropriate for programs w/:
  - ↳ lots of decision making
  - ↳ important logical rltnships among input vars

↳ calcs w/ subsets of input vars
↳ cause + effect rltnship btwn input + output
↳ complex computation logic

black-box testing techniques summary

Technique	Input	Output
Error guessing	Ad hoc	Ad hoc
Scenario graph	Each use case ⊟ scenarios	Not considered
Equivalence partitioning	Each input ⊟ classes	Each output ⊟ classes
Decision table	Each input ⊟ conditions	Each output ⊟ actions
Category partitioning	Each parameter ⊟ categories ⊟ choices + constraints	Each environmental condition ⊟ categories ⊟ choices + constraints

same
same
same
same
characteristics of input/output

FUZZING

- **fuzzing**: set of automated testing that tries to identify abnormal program behaviour by evaling how tested program responds to various inputs
  - ↳ i.e. auto test case generation
  - ↳ goals:
    - find real bugs
    - reduce # false +ves
      - → gen reasonable input (e.g. if expecting str, passing in file as input will result in rejection before even making it to code)
- fuzzer can be categorized as:
  - ↳ **generation-based**: inputs gen from scratch
    - can't always simply gen random inputs b/c it might take very long to reach valid input
      - → e.g. URL parser that only accepts URL in valid format
  - ↳ **mutation-based**: modifying existing inputs
    - e.g. set of img files to be randomly mutated
    - e.g. set of logged input to be randomly modified
      - → "Milk, 3.99" → "Gilk, 3.99"
  - ↳ **dumb/smart** depending on if it's aware of input struct
  - ↳ **white/grey/black-box** depending on if it's aware of program struct
- **grammars** are 1 of most popular + best understood formalisms to specify input langs
  - ↳ grammar gen inputs can be seeds in mutation-based fuzzing
- simple grammar fuzzer:
  - ↳ starts w/start symbol <start> + keeps expanding it
  - ↳ to avoid expansion to infinite inputs, limit # nonterminals
  - ↳ to avoid situation where we can't reduce # symbols further, limit total # expansion steps
  - ↳ e.g.

```
"<start>": ["<expr>"],
"<expr>": ["<term> + <expr>", "<term> - <expr>", "<term>"],
"<term>": ["<factor> * <term>", "<factor> / <term>", "<factor>"],
"<factor>": ["+<factor>", "-<factor>", "(<expr>)", "<integer>.<integer>", "<integer>"],
"<integer>": ["<digit><integer>", "<digit>"],
"<digit>": ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"]
```

<start> -> <expr>	<expr>
<expr> -> <term> + <expr>	<term> + <expr>
<term> -> <factor>	<factor> + <expr>
<factor> -> <integer>	<integer> + <expr>
<integer> -> <digit>	<digit> + <expr>
<digit> -> 6	6 + <expr>
...	

- use **coverage** to guide fuzzer into testing new paths

↳ evolve successful inputs (i.e. new path found during test execution)
↳ e.g.

```
corpus ← initSeedCorpus()
queue ← ∅
observations ← ∅
while ¬isDone(observations,queue) do
    candidate ← choose(queue, observations)
    mutated ← mutate(candidate,observations)
    observation ← eval(mutated)
    if isInteresting(observation,observations) then
        queue ← queue ∪ mutated
        observations ← observations ∪ observation
    end if
end while
```

- **initSeedCorpus**: Initialize a new seed corpus.
- **isDone**: Determine if the fuzzing should stop or not based on progress toward a goal, or a timeout.
- **choose**: Choose at least one candidate seed from the queue for mutation.
- **mutate**: From at least one seed and any observations made about the program so far, produce a new candidate seed.
- **eval**: Evaluate a seed on the program to produce an observation.
- **isInteresting**: Determine if the observations produced from an evaluation on a mutated seed indicate that the input should be preserved or not.

Can be: New path is executed

power schedule: distributes fuzzing time among seeds in pop

greybox fuzzing: collect runtime info (e.g. cov), then guiding gen of inputs twd unexplored paths to uncover new bugs

↳ goal is to maximize time spent fuzzing most progressive seeds so there's higher cov inc in shorter time

↳ seed's energy is likelihood it's chosen from its pop

- assign more energy to seeds that are shorter, faster, or yielding cov that inc more often

↳ directed greybox fuzzing is when we know spot in code is more dangerous so we implement power schedule that assigns more energy to seeds that are closer to target fcn

- build call graph among fcns
- for each fcn in a test, calc its shortest path dist to target fcn + calc avg dist
- using avg dists for all tests, calc power schedule

search-based fuzzing: deriving specific test inputs to achieve some objective (e.g. reach specific stmts)

↳ applying classic search algos (e.g. BFS) is unrealistic b/c we may need to look at all possible inputs

↳ domain knowledge can help us estimate which of several program inputs is closest to target

↳ e.g.

```
def test_me(x, y):
    if x == 2 * (y + 1):          To cover this
        return True
    else:
        return False
```

Randomly test_me(4, 2) return false

⬇

Search the 8 neighbors of (4, 2)

⬇

Recursively until covering this spot

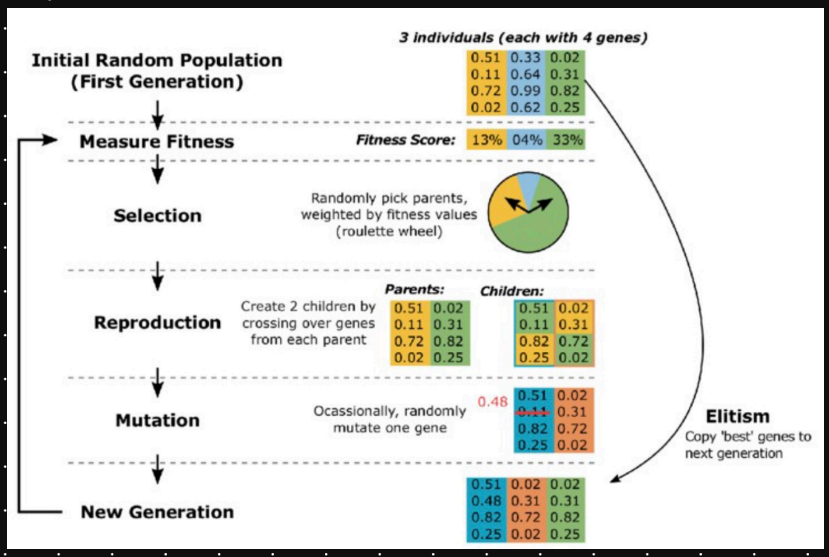
- to estimate how good input is, check **fitness** of individual
  - → e.g.:

```
def calculate_distance(x, y):
        return abs(x - 2 * (y + 1))
```

- to do the recursion, apply **hill climbing algo**:
  1) take random starting pt
  2) determine fitness of neighbours
  3) move to neighbour w/ best fitness val
  4) if soln not found, continue w/ step 2

- hill climbing only works well if neighbourhood relatively small
  - ↳ e.g. if input is UTF-16 chars, range of 65536 possible neighbours
  - ↳ instead of looking at immediate neighbours, we can allow larger modifications (i.e. mutation)
    - still too costly

- **genetic algo (GA)**: based on natural selection process that mimics biological evolution
  - ↳ **chromosome** consists of seq of genes + each gene encodes 1 trait of individual
  - ↳ process:
    1) create init pop of random chromosomes
    2) select fit individuals for reproduction
    3) gen new pop thru reproduction of selected parents + creating mutations
  - ↳ e.g.



FUZZING CHALLENGES

- **feeding in inputs**: fuzz engine will execute target many times w/ diff inputs in same process + it must:
  - ↳ tolerate any kind of input
    - e.g. empty, huge, malformed
  - ↳ not exit on any input
  - ↳ ideally join all threads by end of fcn if using them

↳ be as deterministic as possible
- non-determinism (e.g. random decisions not based on input bytes) makes fuzzing inefficient

↳ be fast
- avoid ≥ cubic complexity, logging, + mem consumption

↳ ideally not modify any global state

↳ ideally be narrower
- e.g. if target can parse several data formats, split into several targets w/ 1 per format

detecting abnormal behaviour

↳ need oracle to provide us w/ defn of abnormal

↳ some common ones:
- crashes
- triggers user-provided assertion failure
- hangs (i.e. execution takes longer than expected)
- allocs too much mem

↳ use **sanitizers** for early crash detection
- e.g. address, thread, mem, undefined behaviour, leak

ensuring progress - use cov to ensure program is exploring more program states over time.

↳ evolves test cases that find new path thru program execution

↳ e.g. greybox + search-based approaches

coming up w/ interesting inputs

↳ options:

1) random input where we don't necessarily control input type
   → e.g. "Milk, 3.99" → 9620
2) understand input type
   → e.g. "Milk, 3.99" → (is str) → "%ab$"
3) understand format type
   → e.g. "Milk, 3.99" → '\w+, \d\.\d\d' → "HELLO, 8.43"
4) formal approaches
   → e.g. model, grammar, protocol based fuzzing
   → useful when problem well-struct but impractical for large real-world problems

speed

↳ avoid paying penalty for setup time by starting / copy + cloning when init is done

↳ replace costly resources w/ cheaper ones
- e.g. local db instead of connecting to remote db

↳ run many inputs on single process

↳ minimize # **test corpuses** (i.e. collection of test cases) + their size

- e.g. when 2 result in same cov, discard bigger 1
- e.g. try to remove parts of existing corpus st cov stays same

- overwhelming # false +ves
- focus on code cov rather than finding reasonable inputs
- requires manual cleaning
  - ↳ make random input more reasonable
  - ↳ minimization: elim redundant test failures thru diffing
  - ↳ triage: find similar outputs + group in same bug report

METAMORPHIC TESTING

- software testing techniques generally assume there's oracle, which may not hold in practice
  - ↳ e.g. shortest path in graph, translation software, compiler
- metamorphic testing: gen new test cases based on I/O pairs of prev test cases, esp. successful ones
  - ↳ i.e. always assume there's errors inside software despite correct output from curr input
- to gen new test cases:
  - ↳ given program P & test case x, corresponding output is P(x)
  - ↳ let x0 be test case w/ output P(x0)
  - ↳ construct new test cases x1, x2, ..., xk based on I/O pair (x0, P(x0)) + error usually associated w/ P
    - designed to reveal errors undetected by x0
- e.g. bin search on sorted arr

Consider a program which locates the position of a given key in a sorted array with distinct elements using binary search. The program should return the position of the key if it exists or a value of -1otherwise.

Let us assume that the input is (x,A[i..j]) and the output is k. Depending on the input-output pair, we have the following cases:

Case 1: A [k] != x. Obviously there is an error.

Case 2: k= -1

Either x does not exist in A, or

there is a bug.

If A is known, this can be verified easily. What if A is a huge database in production and impractical to scan the entire database to confirm?

Testing another randomly selected value from A[p] and verify it with test case (A[p], A[i..j]) would increase the confidence.

Case 3: A[k]=x. There still may be bugs in the program.

Possibility one:

Program bug that overwrites the content of A[k] with x.

Check A[k-1]<x<A[k+1].

If correct:

Choose y from A[k-1]<y<A[k+1] where y !=x

Test (y, A[i..j]), output should be -1.

This is the newly generated test case by metamorphic testing


```
1.  int BinSearch(x,A,i,j)
2.  {
3.    if (i > j)
4.       return -1
5.    else
6.       {
7.         mid = floor((i+j)/2);
8.         if (A[mid]==x)
9.           return mid;
10.        if (A[mid] > x)
11.          BinSearch(x,A,i,mid-1);
12.        else
13.          BinSearch(x,A,mid+2,j); /* correct statement is:   */
14.      }                           /* BinSearch(x,A,mid+1,j); */
15. }
```

Let A be [4, 6,10,15,18,25, 40]
and x be 25
Call BinSearch(x, A, 1, 7)
The return value would be 6

Case 3: A[k]=x
Possibility one:
Check A[k-1]<x<A[k+1] A[5]=18, x=25, A[6]=40
Pick a value from 18 to 40, run BinSearch, always return -1

Case 3: A[k]=x. There still may be bugs in the program.

Possibility two: splitting error which is not revealed by this test case.

Rerun the program with

A[k-1], A [i..j] with expected output k-1

A[k+1], A [i..j] with expected output k+1

These are newly generated test cases by metamorphic testing


```
1.  int BinSearch(x,A,i,j)
2.  {
3.    if (i > j)
4.       return -1
5.    else
6.       {
7.         mid = floor((i+j)/2);
8.         if (A[mid]==x)
9.           return mid;
10.        if (A[mid] > x)
11.          BinSearch(x,A,i,mid-1);
12.        else
13.          BinSearch(x,A,mid+2,j); /* correct statement is:   */
14.      }                           /* BinSearch(x,A,mid+1,j); */
15. }
```

Let A be [4, 6,10,15,18,25, 40]
and x be 25
Call BinSearch(x, A, 1, 7)
The return value would be 6

Case 3: A[k]=x
Possibility two:
Run A[k-1], A [i..j] with expected output k-1
A[k+1], A [i..j] with expected output k+1

BinSearch(18, A, 1, 7) => expected return 5, actual return -1
Binsearch (40, A, 1, 7)=> expected return 7, actual return -1

e.g. shortest path in undirected graph

Given a weighted graph G, a source node X and the destination node y in G. The problem is to output the shortest path and shortest distance.

Assume the output path is x, v1, v2,....vk, y and the distance is p.
What are the new test cases?

(y,x,G) expected distance is p

(x, vi, G)and (vi, y, G)for any 1<=i <=k, the sum of the two distances = p

e.g. translation software

- I live on campus with smart people
  - Je vis sur le campus avec des gens intelligents

Replace words by synonyms and translation results should be similar.

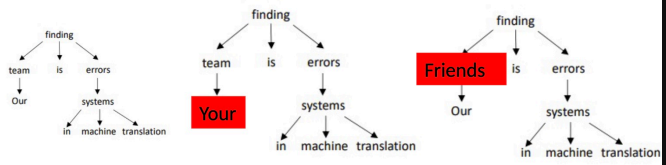
- I live on campus with **clever** people
- I live on campus with **intelligent** people
  - Je vis sur le campus avec des gens intelligents

Even synonyms may not have the exactly same translation.

- I live on campus with bright people
  - Je vis sur le campus avec des gens brillants

- Notre équipe détecte des erreurs dans les systèmes de traduction automatique
  - Our team is finding errors in machine translation systems

Structure of the sentence should not be changed



- Votre équipe détecte des erreurs dans les systèmes de traduction automatique
  - Your team is finding errors in machine translation systems
- Notre ami détecte des erreurs dans les systèmes de traduction automatique
  - Our friend is finding errors in machine translation systems

TEST DRIVEN DEVELOPMENT

test driven development (TDD) algo

1) write failing test case
2) get it to compile
3) make it pass
4) refactor code + remove duplication
5) repeat

benefits:

↳ look at requirements 1st
↳ full control over pace of writing prod code
↳ quick feedback
↳ testable code
↳ feedback abt design

use TDD when:

↳ unclear idea of how to design, architect, or implement requirement
↳ dealing w/ complex or unfamiliar problem

don't use TDD when we alr know problem + comfortable w/ coding soln directly

↳ nothing to learn in process

disadvantages:

↳ distracts from writing code blc we must always think abt getting code to fail
↳ needs lots of discipline to write tests 1st
↳ must write lots of simple + useless tests
↳ must maintain tests
  • often need to rewrite as system evolves
↳ architecture is designed around making it easier to test
↳ needs lots of mock objs to decouple from db + UI
↳ maintenance tests aren't same as dev ones
  • sometimes better to remove low val unit tests

compromise is to ensure every commit has test

↳ don't need to write test 1st, but must write before adding commit to git repo

TDD legacy code algo:

1) get class we want to change under test
2) write failing test case
3) make it pass
  • try to not change existing code + instead add new code
4) refactor code + remove duplication
5) repeat

benefits of TDD w/ legacy code:

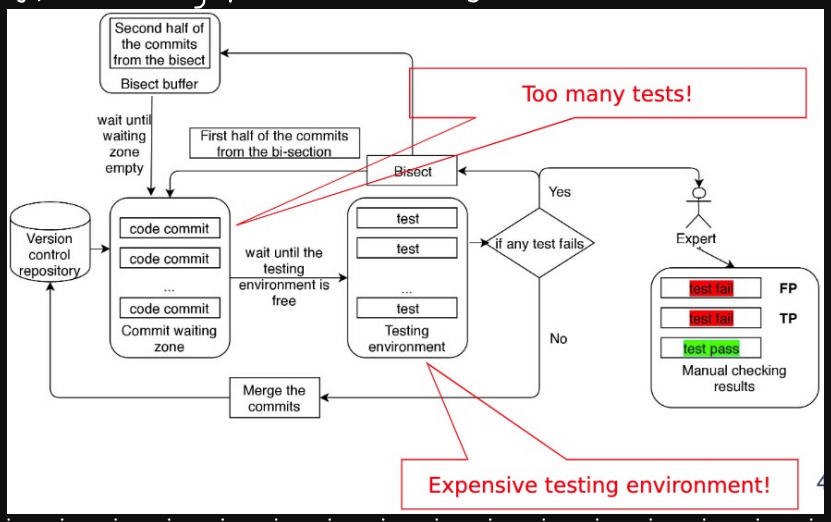
↳ concentrate on either writing code or refactoring + nvr doing both at once

- can write new code independently of old code
  - ↳ after writing new code, refactor to remove duplication btwn it + old code

TEST MINIMIZATION, SELECTION, AND PRIORITIZATION

TEST SUITE MINIMIZATION

- software testing is expensive + most tests don't fail
- typical testing process in industry:



- **test suite minimization** aims to reduce # test cases while maintaining adequate cov
  - ↳ input: test suite T, set of requirements $\{r_1, \ldots, r_n\}$ that must be satisfied, + subsets $T_1, \ldots, T_n$ of T (each is associated w/ r_i st any test case t_j in T_i can acheive r_i)
  - ↳ output: **minimal hitting set** of t_j's st all r_i's are satisfied
- **essential test case**: r_i can only be satisfied by that 1 test case
- **redundant test case**: satisfies only subset of what another test case satisfies
- **greedy essential (GE)** heuristic: select all essential test cases 1st then use additional greedy algo on remaining reqs (i.e. select test case that satisfies max. #unsatisfied reqs)
- **greedy redundant essential (GRE)** heuristic: remove all redundant test cases + do GE heuristic on reduced test suite
- **min hitting set process**:
  1) all test cases in single elmt T_i's are included + all T_i's containing those elmts are crossed off
  2) all unmarked T_i's w/ 2 elmts are considered
     - ↳ test case w/ max # occurrences in T_i's is chosen to be in hitting set
     - ↳ if there's tie, no decision
  3) all unmarked T_i's w/ 3 elmts are considered
     - ↳ consider only test cases involved in tie before

4) all unmarked T_i's w/max 2 elmts are considered
5) all unmarked T_i's w/max 3 elmts are considered
6) repeat + inc max #elmts until all reqs are covered

e.g.

Test Case	Testing Requirements					
	r_1	r_2	r_3	r_4	r_5	r_6
t_1	x	x	x			
t_2	x			x		
t_3		x			x	
t_4			x			x
t_5					x	

i	r_i	T_i
1	REQ1	{2,5}
2	REQ2	{5}
3	REQ3	{1,2,3}
4	REQ4	{3,6}
5	REQ5	{1,4}
6	REQ6	{1,6}
7	REQ7	{3,4,7}
8	REQ8	{2,3,4,7}

hit test : {}

→ test 5 is essential

i	r_i	T_i
~~1~~	~~REQ1~~	~~{2,5}~~
~~2~~	~~REQ2~~	~~{5}~~
3	REQ3	{1,2,3}
4	REQ4	{3,6}
5	REQ5	{1,4}
6	REQ6	{1,6}
7	REQ7	{3,4,7}
8	REQ8	{2,3,4,7}

hit test : {5}

→

i	r_i	T_i
~~1~~	~~REQ1~~	~~{2,5}~~
~~2~~	~~REQ2~~	~~{5}~~
3	REQ3	{1,2,3}
4	REQ4	{3,6}
5	REQ5	{1,4}
6	REQ6	{1,6}
7	REQ7	{3,4,7}
8	REQ8	{2,3,4,7}

hit test : {5}

→ tests 1 + 6 have tie

i	r_i	T_i
~~1~~	~~REQ1~~	~~{2,5}~~
~~2~~	~~REQ2~~	~~{5}~~
~~3~~	~~REQ3~~	~~{1,2,3}~~
4	REQ4	{3,6}
~~5~~	~~REQ5~~	~~{1,4}~~
~~6~~	~~REQ6~~	~~{1,6}~~
7	REQ7	{3,4,7}
8	REQ8	{2,3,4,7}

hit test : {5, 1}

→ only test 1 appears

i	r_i	T_i
~~1~~	~~REQ1~~	~~{2,5}~~
~~2~~	~~REQ2~~	~~{5}~~
~~3~~	~~REQ3~~	~~{1,2,3}~~
4	REQ4	{3,6}
~~5~~	~~REQ5~~	~~{1,4}~~
~~6~~	~~REQ6~~	~~{1,6}~~
7	REQ7	{3,4,7}
8	REQ8	{2,3,4,7}

hit test : {5, 1}

→ tests 3+6 have tie

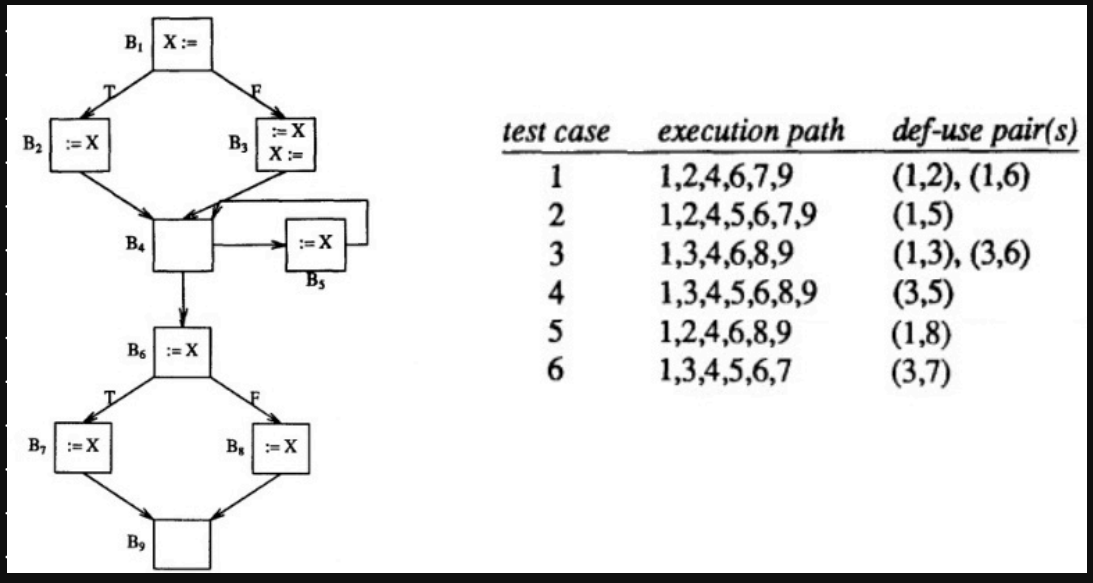
i	r_i	T_i
~~1~~	~~REQ1~~	~~{2,5}~~
~~2~~	~~REQ2~~	~~{5}~~
~~3~~	~~REQ3~~	~~{1,2,3}~~
~~4~~	~~REQ4~~	~~{3,6}~~
~~5~~	~~REQ5~~	~~{1,4}~~
~~6~~	~~REQ6~~	~~{1,6}~~
~~7~~	~~REQ7~~	~~{3,4,7}~~
~~8~~	~~REQ8~~	~~{2,3,4,7}~~

test 3 in both + all reqs satisfied

hit test : {5, 1, 3}

↳ min hitting set is {5, 1, 3}

e.g. data flow graph



test case	execution path	def-use pair(s)
1	1,2,4,6,7,9	(1,2), (1,6)
2	1,2,4,5,6,7,9	(1,5)
3	1,3,4,6,8,9	(1,3), (3,6)
4	1,3,4,5,6,8,9	(3,5)
5	1,2,4,6,8,9	(1,8)
6	1,3,4,5,6,7	(3,7)

def-use pair	associated testing set
(1,2)	{1,2,5}
(1,5)	{2}
(1,6)	{1,2,5,6}
(1,3)	{3,4,6}
(3,5)	{4,6}
(3,6)	{3,4,6}
(1,7)	{1,2}
(1,8)	{5}
(3,7)	{6}
(3,8)	{3,4}

→ tests 2, 5, & 6 are essential.

def-use pair	associated testing set
~~(1,2)~~	~~{1,2,5}~~
~~(1,5)~~	~~{2}~~
~~(1,6)~~	~~{1,2,5,6}~~
~~(1,3)~~	~~{3,4,6}~~
~~(3,5)~~	~~{4,6}~~
~~(3,6)~~	~~{3,4,6}~~
~~(1,7)~~	~~{1,2}~~
~~(1,8)~~	~~{5}~~
~~(3,7)~~	~~{6}~~
(3,8)	{3,4}

hit test 1, 2, 5, 6 ✓

→ tests 3 & 4 to satisfy remaining 1 req.

def-use pair	associated testing set
~~(1,2)~~	~~{1,2,5}~~
~~(1,5)~~	~~{2}~~
~~(1,6)~~	~~{1,2,5,6}~~
~~(1,3)~~	~~{3,4,6}~~
~~(3,5)~~	~~{4,6}~~
~~(3,6)~~	~~{3,4,6}~~
~~(1,7)~~	~~{1,2}~~
~~(1,8)~~	~~{5}~~
~~(3,7)~~	~~{6}~~
~~(3,8)~~	~~{3,4}~~

hit test 1, 2, 5, 6, 3, 4 ✓

TEST CASE SELECTION

- **test case selection** techniques also reduce size of test suite, but are **modification-aware**
  - ↳ selection isn't just temporary (i.e. specific to curr version)
  - ↳ test cases selected b/c they're relevant to changed parts of SUT
    - typically uses white-box static analysis of code
- **regression testing**: given program P, its modified version P', & test set T used prev to test P, find a way to use T in gaining sufficient confidence in correctness of P'
  - ↳ steps:
    1) identify modifications made to P
    2) select $T' \subseteq T$ to re-execute on P'
       → test selection problem
    3) retest P' w/ T' & establish correctness of P' wrt T'
    4) create new tests for P' if needed
    5) create new complete test set for P', including tests from 2) & 4) & old non-selected tests that are still valid
- test $T_i \in T$ is **modification-revealing (MR)** if it prods diff outputs in P & P'
  - ↳ can't decide if test is MR
- test $T_i \in T$ is **modification-traversing (MT)** if it executes new/modified stmt in P' or misses stmt in P' that's executed in P
  - ↳ test is MR ⇒ test is MT
    - implication doesn't hold in rev dir
- if T contains n MR tests & S selects m of those tests, **inclusiveness** of S is $\frac{m}{n}$
- if T contains n non-MR tests & S omits m of those tests, **precision** of S is $\frac{m}{n}$

given following 2 assumptions, MR test case is also fault-revealing (FR) b/c they all terminate & prods correct output for P, so they're supposed to prod same output for P'

↳ P-Correct-for-T assumption: ∀ test cases t∈T, when P is tested w/ t, P halts & prods correct output

↳ Obsolete-Test-Identification assumption: ∀t∈T, there exists effective procedure for determining whether t is obsolete for P'

Controlled-Regression-Testing assumption: ∀t∈T, when P' is tested w/ t, all factors that might influence output of P' (except for code in P') are kept constant wrt states when they were tested w/ P

↳ given this, non-obsolete test case T is MR only if it's MT for P & P'

given above 3 assumptions, $T_{FR} = T_{MR} \subseteq T_{MT} \subseteq T$

↳ key is to find MT / MR

not every stmt that's executed under test case has effect on output

program slice consists of all stmts (including conds) that might affect val of var v at pt p

bwd slices S(v,n) are stmt frags that contribute to val of v at stmt n

↳ stmt n is use node of var v

↳ e.g.

```
1 void foo() {
2  x = 1;
3  y = 2;
4  z = y - 2;
5  r = x;
6  z = x + y;
7 }

// backward slicing on z
// at line 7
```

```
2  x = 1;
3  y = 2;

6  z = x + y;

// resulting backward slicing
// on z at line 7
```

```
1 n = readInt();
2 s=0;
3 p=0;
4 while (n>0) {
5   s=s+n;
6   p=p*n;
7   n=n-1;
8 }
9 System.out.println("p:"+p+"s:"+s);
// backward slicing on p at line 9
```

```
1 n = readInt();

3 p=0;
4 while (n>0) {

6   p=p*n;
7   n=n-1;
8 }

// resulting backward slicing
// on p at line 9
```

↳ e.g. mark program

```
Pass = 0 ;
Fail = 0 ;
Count = 0 ;
while (!eof()) {
TotalMarks=0;
scanf("%d",Marks);
if (Marks >= 40)
Pass = Pass + 1;
if (Marks < 40)
Fail = Fail + 1;
Count = Count + 1;
TotalMarks = TotalMarks+Marks ;
}
printf("Out of %d, %d passed and %d failed\n",Count,Pass,Fail) ;
average = TotalMarks/Count;
/* This is the point of interest */
printf("The average was %d\n",average) ;
PassRate = Pass/Count*100 ;
printf("This is a pass rate of %d\n",PassRate) ;
```

Slice of "average"

```
while (!eof()) {
TotalMarks=0;
scanf("%d",Marks);
Count = Count + 1;
TotalMarks = TotalMarks+Marks;
}
average = TotalMarks/Count;
printf("The average was %d\
n",average) ;
```

fwd slices S(v,n) are stmts that are affected by val of v & stmt n

↳ refers to p-uses & c-uses of v

↳ e.g.

```
x = 1; /* considering changing
this line */
y = 3;
p = x + y ;
z = y -2 ;
if(p==0)
r++ ;
```

```
/* Change to first line will affect */
p = x + y ;
if(p==0)
r++ ;
```

Affected by value

Affected by predicate

↳ e.g.

```
n = 0;
product = 1;
sum = 1;
scanf("%d",&x) ;
while (x >= 0){
sum = sum + x;
product = product * x ;
n = n + 1;
scanf("%d",&x);
}
average = (sum - 1) / n ;
printf("The total is %d\n",sum) ;
printf("The product is %d\n",product) ;
printf("The average is %d\n",average) ;
```

Backward slice of **sum**

```
sum = 1;
scanf("%d",&x) ;
while (x >= 0) {
sum = sum + x;
scanf("%d",&x);
}
printf("The total is
%d\n",sum) ;
```

Bug shows here

↓ fix bug

```
n = 0;
product = 1;
sum = 0;
scanf("%d",&x) ;
while (x >= 0){
sum = sum + x;
product = product * x ;
n = n + 1;
scanf("%d",&x);
}
average = (sum - 1) / n ;
printf("The total is %d\n",sum) ;
printf("The product is %d\n",product) ;
printf("The average is %d\n",average) ;
```

We change this line to fix the bug.

This line also needs to be changed

bwd + fwd slicing are forms of static slicing

↳ can be used to simplify program, but slices can be large

dynamic slice is constructed wrt static slicing criterion + dynamic info like input

↳ criterion:

- vars to be sliced
- pt of interest within program
- seq of input vals for which program is executed

} same as static slicing

↳ e.g.

```
1 n = readInt();
2 s=0;
3 p=0;
4 while (n>0) {
5   s=s+n;
6   p=p*n;
7   n=n-1;
8 }
9 System.out.println("p:"+p+"s:"+s);
// dynamic slicing on p at line 9
// for input <0>
```

```
3 p=0;
```

```
// resulting dynamic slicing on p
// at line 9 for input <0>
```

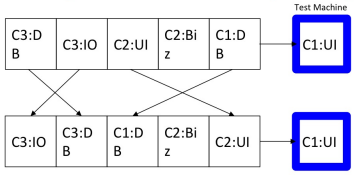
- 2 main props desirable in slicing algos:
  - ↳ soundness: algo must nvr delete stmt from og program which could have effect on slicing criterion
    - allows us to analyse slice in total confidence that it contains all stmts relevant to criterion
    - not enough b/c trivial algo could delete no stmts + still achieve soundness
  - ↳ completeness: algo must remove all stmts which can't affect slicing criterion
    - generally unachievable
- goal of slicing algos is to delete as many stmts as possible while still being sound
  - ↳ closer algo apx completeness, more precise slices are
- dynamic slice for some vars at certain pt will always be at least as thin as corresponding static slice
  - ↳ in practice, dynamic slices are usually simpler

TEST CASE PRIORITIZATION

- test case prioritization: order test cases for early maximisation of desirable props (e.g. rate of fault detection)
  - ↳ find optimal permutation
  - ↳ doesn't involve test selection + assumes all test cases may be executed in chosen order
  - ↳ input: test suite T, set of permutations PT for T, + fcn from PT to real #s
    $f: PT \to \mathbb{R}$
  - ↳ output: $T' \in PT$ st $\forall T''. (T'' \in PT) \wedge (T'' != T') \wedge (f(T') \geq f(T''))$
- while goal of test case prioritization is achieving higher fault detection rate, techniques aim to maximise cov
  - ↳ early max cov will inc chance of early max of fault detection
  - ↳ algos can be random, hill climbing, genetic, greedy, etc.
- can use test history to prioritise tests
  - ↳ historical probability failure method
  - ↳ adjust failure probability of queued tests after each run
  - ↳ e.g.

Test Name	Historical Failure rate
UI_{test}	10%
Biz_{test}	5%
DB_{test}	2%
IO_{test}	1%

Historical Failure Probability (S. Elbaum, G. Rothermel, and J. Penix)



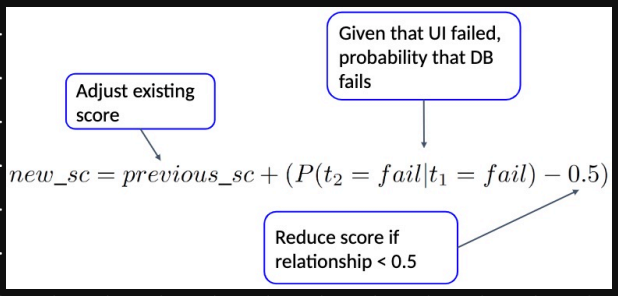
Test	Test	Co-failure
UI_{test}	DB_{test}	60%
UI_{test}	Biz_{test}	50%
UI_{test}	IO_{test}	10%

Test Machine

C1:UI

Given that the UI failed, adjust failure probability of queued tests

  - ↳ use conditional probability to re-order tests for single change
    - scoring fcn is used to globally order tests:



- some tests may **starve** due to prioritization

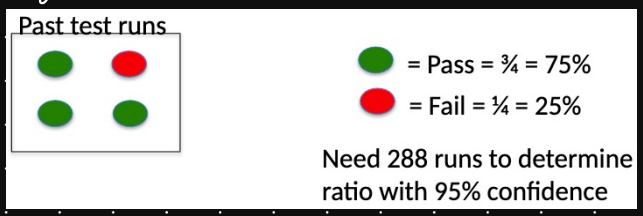
- **flaky test** passes & fails on same build
  - ↳ reasons to keep:
    - some tests may have inherent non-determinism
      - → hardware / env noise
      - → async props
      - → decision isn't rule-based, but on AI model of data
    - broken tests that are expensive to fix
- to quantify a flaky test:
  1) measure deg of test flakiness
     - **FlakeRate = flakes / runs**
  2) establish FlakeRate baseline on stable build / prod release
  3) determine how flaky tests are
     - calc #runs to have statistically confident StableFlakeRate
- classification of **test outcomes**:
  - ↳ good:
    - **TP**: test fails & finds fault (true +ve)
    - **TN**: test passes & no fault (true -ve)
  - ↳ bad:
    - **FP**: test fails & no fault (false +ve)
    - **FN**: test passes & fault slips thru to prod (false -ve)
- **accuracy = (TP + TN) / runs**
- **FlakeRate = (FP + FN) / runs = 1 - accuracy**

stable build is 1 that's been successfully running in prod or prev stable release
- ↳ need this so that any fail is FP & any pass is TN
- ↳ **StableAccuracy = TN / runs**
- ↳ **StableFlakeRate = FP / runs = 1 - StableAccuracy**

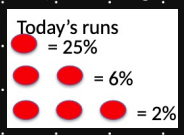
use binomial estimation to determine #runs for **statistical confidence** in FlakeRate
- ↳ e.g. 99% confidence = 666 runs, 99.9% confidence = 1083 runs
- ↳ highly flaky tests require many re-runs

use **failure rate** to prioritize tests
- ↳ assess likelihood of system going from stable to unstable state using binomial distribution
- ↳ prioritize re-runs of tests that are more likely to uncover instabilities

- tests are **no longer binary** b/c they become unstable relative to baseline StableFlakeRate
- use Bernoulli trial to calc probability of exactly k successes in n trials
  - ↳ $P(k) = \binom{n}{k} p^k q^{n-k}$
    - p is prob of success
    - q is prob of failure
  - ↳ e.g.



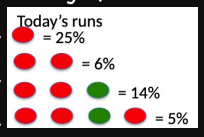
    - likelihood of all fails:



$\binom{2}{0} 0.25^2$
$\binom{3}{0} 0.25^3$

→ cost 3 runs

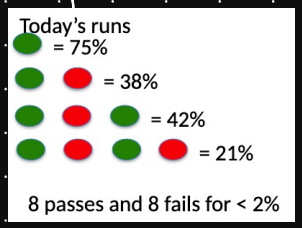
    - noisy pass:



$\binom{3}{1} 0.25^2 \cdot 0.75$
$\binom{4}{1} 0.25^3 \cdot 0.75$

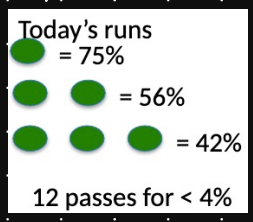
→ cost 4 runs

    - alt pass/fail:



→ cost 16 runs

    - all passes:



→ cost 12 runs

- **signal**: test starts failing much more than expected
  - ↳ someone likely changed code + there's fault
- **noise**: test fails at expected lvls
  - ↳ likely just normal interference
- algo to **prioritize re-runs**:
  1) run each test once

2) calc P_t
   - $P_t(f, r, \text{FlakeRate}) = \binom{r}{f} \cdot \text{FlakeRate}^f \cdot (1 - \text{FlakeRate})^{r-f}$
     - → P_t tells prob of getting set of test runs w/same outcomes
     - → FlakeRate is estimated by test failure rate
     - → r is # runs
     - → f is # failures
3) order tests by P_t
4) run test w/ lowest P_t
5) investigate tests w/ highly unlikely FailRate (i.e. unstable)
6) go to 4) until no more test runs budgeted or high statistical confidence reached
7) compare pass/fail distribution of each test w/ history & decide if there's significant deviance

- comparison btwn w/ & w/o considering flaky tests:

	Without flaky tests	With flaky tests
Indicator of smth bad (e.g. bug)	↳ test fail	↳ ratio btwn pass/fail changes
Confidence of test results	↳ N/A b/c run once	↳ run multiple times until budget reached or statistical confidence achieved
Prioritization	↳ which test most likely to fail	↳ which test w/ most unlikely results

- as long system behaviour is stable, might not have to fix flaky tests
  - ↳ can still get signal by re-running & examing P_t
- making test less flaky results in less noise & require fewer re-runs
  - ↳ cost to fix test vs cost to re-run & calc P_t

AUTOMATED DEBUGGING

STATISTICAL FAULT LOCALIZATION

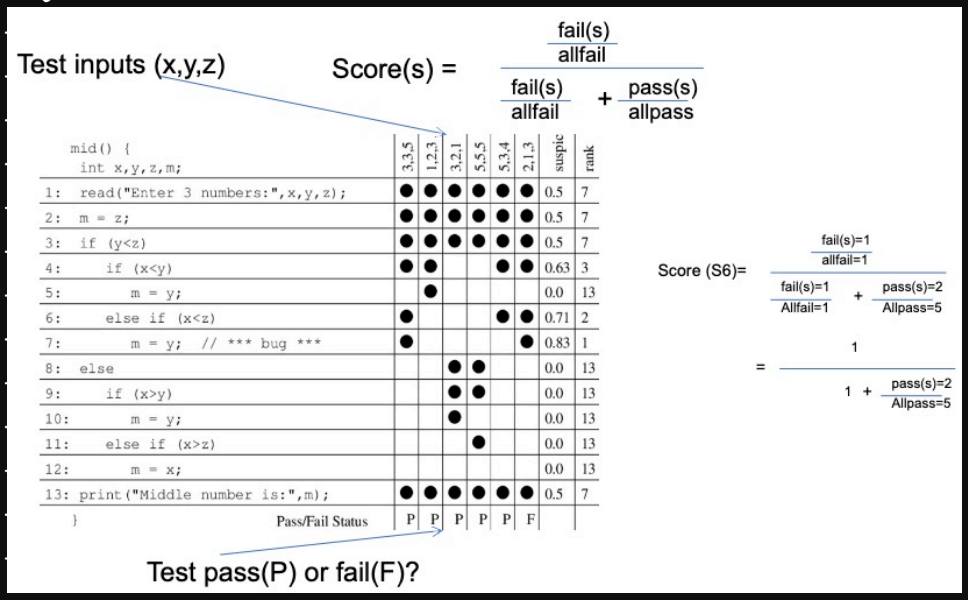
assign scores to programs based on their occurrence in passing / failing tests

↳ correlation ≠ causation

↳ $score(s) = \frac{fail(s)/allfails}{fail(s)/allfails + pass(s)/allpasses}$

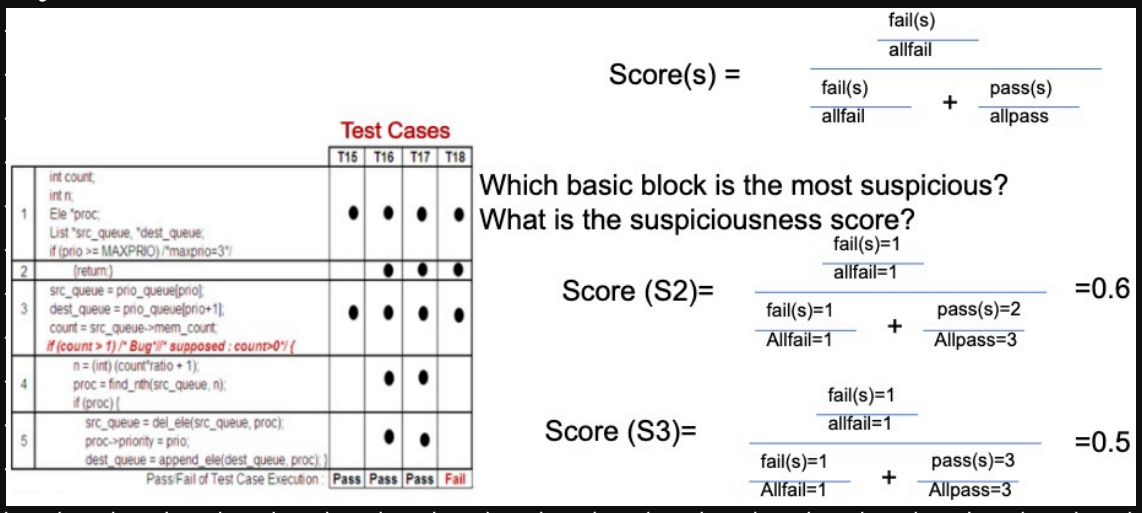
- s is stmt
- Tarantula scoring scheme
- fail(s) = # failing executions where s occurs
- pass(s) = # passing executions where s occurs
- allfails = total # failing executions
- allpasses = total # passing executions

↳ e.g.



	mid() { int x,y,z,m;	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3	suspic	rank
1:	read("Enter 3 numbers:",x,y,z);	●	●	●	●	●	●	0.5	7
2:	m = z;	●	●	●	●	●	●	0.5	7
3:	if (y<z)	●	●	●	●	●	●	0.5	7
4:	if (x<y)	●	●			●	●	0.63	3
5:	m = y;		●					0.0	13
6:	else if (x<z)	●				●	●	0.71	2
7:	m = y; // *** bug ***	●					●	0.83	1
8:	else			●	●			0.0	13
9:	if (x>y)			●	●			0.0	13
10:	m = y;			●				0.0	13
11:	else if (x>z)				●			0.0	13
12:	m = x;							0.0	13
13:	print("Middle number is:",m);	●	●	●	●	●	●	0.5	7
}	Pass/Fail Status	P	P	P	P	P	F		

↳ e.g.



DELTA DEBUGGING

- once program failure is reproduced, simplify test case for:
  - ↳ ease of communication
  - ↳ easier debugging
  - ↳ identifying duplicates
- use bin search + automate it to simplify test case
  - ↳ throw away half of input + see if output's still wrong
  - ↳ if output's correct, go back to prev state + try throwing away other half of input instead
  - ↳ e.g.

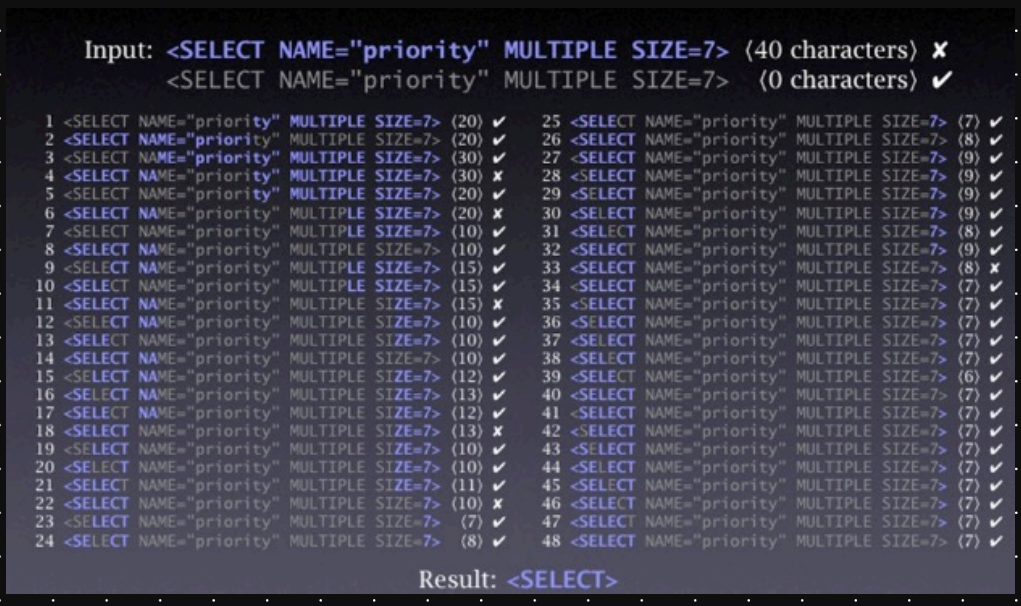
```
<SELECT NAME="priority" MULTIPLE SIZE=7>      ✘
<SELECT NAME="priority" MULTIPLE SIZE=7>      ✔
<SELECT NAME="priority" MULTIPLE SIZE=7>      ✔
```

    - both halves pass so we must shrink input size
- impact of input granularity:

	Finer granularity	Coarser granularity
Chance of finding failing input subset	Higher	Lower
Progress of search	Slower	Faster

- e.g.

```
<SELECT NAME="priority" MULTIPLE SIZE=7>      ✘
<SELECT NAME="priority" MULTIPLE SIZE=7>      ✔
<SELECT NAME="priority" MULTIPLE SIZE=7>      ✔
<SELECT NAME="priority" MULTIPLE SIZE=7>      ✔
<SELECT NAME="priority" MULTIPLE SIZE=7>      ✘
<SELECT NAME="priority" MULTIPLE SIZE=7>      ✘
<SELECT NAME="priority" MULTIPLE SIZE=7>      ✔
```



- R is set of all possible inputs
  - ↳ $r_p \in R$ is input that passes

↳ $r_F \in R$ is input that fails

change δ is mapping $R \to R$ which takes 1 input + changes it to another

↳ e.g.

Example: δ' = insert ME="priori at input position 10

```
r1 = <SELECT NAty" MULTIPLE SIZE=7>
δ'(r1) = <SELECT NAME="priority" MULTIPLE SIZE=7>
```

↳ δ can be decomposed to elementary changes st $\delta = \delta_1 \circ \delta_2 \circ \cdots \circ \delta_n$

$\circ$ $(\delta_i \circ \delta_j)(r) = \delta_j(\delta_i(r))$

↳ e.g. deleting part of input can be decomposed to deleting 1 char at a time

↳ e.g.

Example: δ' = insert ME="priori at input position 10
can be decomposed as $\delta' = \delta_1$ o δ_2 o ... o δ_{10}

where δ_1 = insert M at position 10
δ_2 = insert E at position 11
...

$C_F = \{\delta_1, \delta_2, \ldots, \delta_n\}$ is a set of changes st $r_F = (\delta_1 \circ \delta_2 \circ \cdots \circ \delta_n)(r_P)$

↳ each subset $C \subseteq C_F$ is a test case

given test case C, we want to know if input generated by applying changes in C to r_P will cause same failure as r_F.

↳ define test: Powerset $(C_F) \to \{P, F, ?\}$ st given $C = \{\delta_1, \delta_2, \ldots, \delta_n\} \subseteq C_F$, test(C) = F iff $(\delta_1 \circ \delta_2 \circ \cdots \circ \delta_n)(r_P)$ is input that fails

when minimizing test cases, goal is to find smallest test case C st test(C) = F

↳ failing test case $C \subseteq C_F$ is global min if $\forall C' \subseteq C_F$, $|C'| < |C| \Rightarrow$ test(C') $\neq$ F

- i.e. global min is smallest set of changes that'll make test fail
- finding global min may require exponential # tests

↳ failing test case $C \subseteq C_F$ is local min if $\forall c' \subset C$, test(C') $\neq$ F

failing test case $C \subseteq C_F$ is n-minimal if $\forall C' \subset C$, $|C| - |C'| \leq n \Rightarrow$ test(C') $\neq$ F

↳ 1-minimal if $\forall \delta_i \in C$, test$(C - \{\delta_i\}) \neq$ F

- i.e find set of changes that cause failure, but removing any 1 of them causes failure to go away

e.g.

A program takes a string of **a**'s and **b**'s as input. It crashes on inputs with an odd number of b's AND an even number of **a**'s. Write a crashing test case (or **NONE** if none exists) that is a sub-sequence of input **babab** and is:

- Smallest: b
- 1-minimal, of size 3: aab, aba, baa, bbb
- Local minimum but not smallest: NONE
- 2-minimal, of size 3: NONE

naive algo to find 1-minimal subset of C:

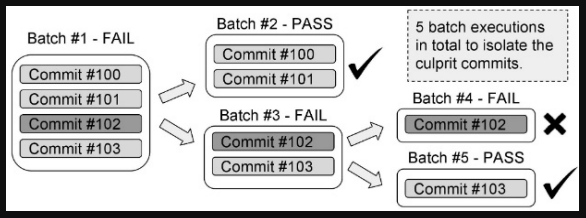
if $\forall \delta_i \in c$, test$(c - \{\delta_i\}) \neq F$
  c is 1-minimal
else
  recurse on $c - \{\delta\}$ for some $\delta \in c$ st test$(c - \{\delta\}) = F$
↳ in worst case, we remove 1 elmt from set per iteration after trying every other elmt
↳ runtime is $N + (N-1) + (N-2) + \ldots \in O(N^2)$

delta debugging algo for finding a 1-minimal test case is better
↳ steps:
  1) start w/ $n = 2$ + $\Delta(c_F)$ as test set
  2) test each $\Delta_1, \Delta_2, \ldots, \Delta_n$ + each $\nabla_1, \nabla_2, \ldots, \nabla_n$
    → Δ_i is partition of c_F st c_F is divided into n partitions
    → ∇_i is complement of Δ_i st $\nabla_i = c_F - \Delta_i$
  3) there's 3 possible outcomes:
    → some Δ_i causes failure → go to 1 w/ $\Delta = \Delta_i$ + $n = 2$
    → some ∇_i causes failure → go to 1 w/ $\Delta = \nabla_i$ + $n = n - 1$
    → no test causes failure
      - if granularity can be refined, go to 1 w/ $\Delta = \Delta$ + $n = 2n$
      - if not, done + found 1-minimal subset
↳ worst case still quadratic
  • subdivide until each set is of size 1 so it's reduced to naive algo
↳ for single failure, converges in $O(\log N)$
↳ e.g.

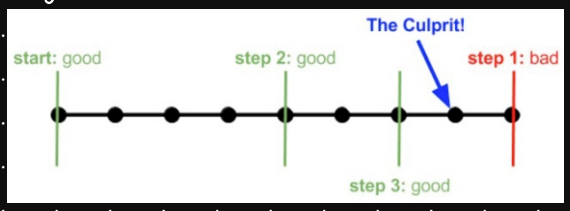
A program crashes when its input contains 42. Fill in the data in each iteration of the minimization algorithm assuming character granularity.

Iteration	n	Δ	$\Delta_1, \Delta_2, \ldots, \Delta_n$, $\nabla_1, \nabla_2, \ldots, \nabla_n$
1	2	2424	24
2	4	2424	2, 4, 242, 424, 224, 244
3	3	242	2, 4, 24, 42, 22
4	2	42	4, 2

delta debugging is technique, not tool
↳ probably have to re-implement for each system to exploit knowledge changes
↳ relatively simple algo w/ big payoff
e.g. batch testing + bisect



- `git bisect` performs bin search to find 1st faulty commit that fails test
  - ↳ steps:
    1) start pt is good commit + ending pt is bad commit
    2) split suspect commits in half
    3) if test fails, bug is in left half
    4) if test passes, bug is in right half
    5) repeat
  - ↳ e.g.



  - ↳ uses bin search + regression test to find intro of bug
  - ↳ runtime is $O(\log n)$
  - ↳ `cmds`:
    - git bisect start: starts tool
    - git bisect bad: loc of 1st commit that we're sure is bad
      - → default is head
      - → can specify older commit w/ hash #
    - git bisect good: loc of commit where we're sure code worked
    - git bisect reset: get out of bisect
    - git bisect run <cmd>: automate git bisect to run <cmd> as test iteratively until 1st bad commit is found

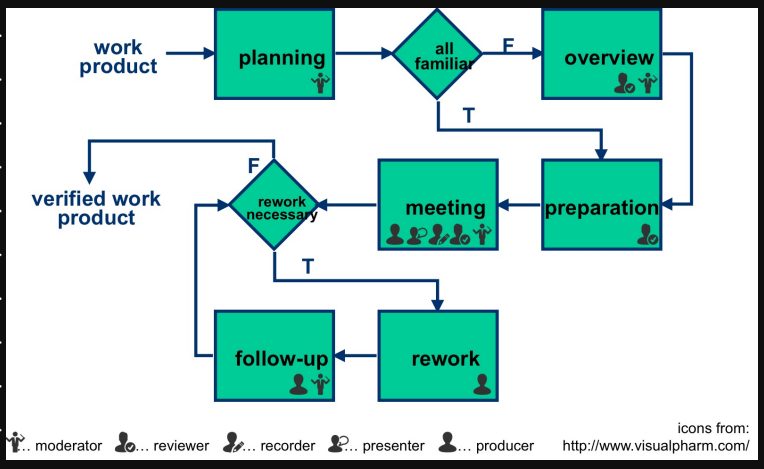
CODE REVIEW METRICS

SOFTWARE INSPECTIONS AND REVIEWS

- verification + validation (V+V) techniques can be of 2 types:
  - ↳ static: analyze code
    - advantages:
      - → can catch cascading errors that might mutate test results
      - → incomplete / non-executable versions can be inspected
      - → good inspections don't just catch bugs, but also check for inefficiencies + style issues while also sharing knowledge
  - ↳ dynamic: execute code
- review process: work product presented to proj personnel, managers, users, clients, + other interested parties for approval
  - ↳ identify defects
  - ↳ informally exchange knowledge
  - ↳ collect data to learn from mistakes
- software review: human exam of work product
  - ↳ static, white-box approach
  - ↳ diff types:
    - inspection: structured + formal
    - walkthru: less formal + presentation format
    - buddy check: informal check
    - personal review: check by author
    - formal design review (FDR): similar to inspections, but decides whether to give proj green light
    - modern, web-based reviews
- review reports rate severity of defect + determine stats abt findings + invested resources (i.e. quality / efficiency metrics)
  - ↳ required for formal inspections
  - ↳ e.g. inspection / FDR meeting notice report launches review process
- checklists are most important tools for review
  - ↳ generic types:
    - requirements
    - design
    - generic code
    - specific lang code
    - generic doc
- inspections are strict + v formal form of review to obtain defects, collect data, + communicate dev docs
  - ↳ roles:

- moderator
  - → ensures procs are followed
  - → verifies work product's readiness
  - → verifies entry + exit criteria are met
  - → assembles effective team
  - → keeps meeting on track
  - → outside proj team, but still peer
- recorder / scribe
  - → documents all defects found in meeting
- reviewer
  - → analyzes + detects bugs
  - → all participants play this role
  - → rep for user, test, design, + implementation teams
- reader / presenter
  - → leads team by reading aloud small logical units
- producer / author
  - → work product author
  - → responsible for correcting defects

↳ inspection process:



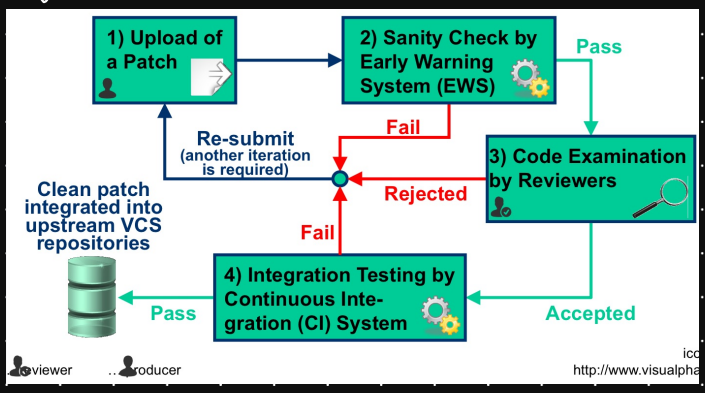
- planning: identify work prod, ensure entry criteria met, select team, assign roles, prep + distribute materials, set schedule, + determine if overview needed
- overview: optional phase where members unfamiliar w/ work product get orientation
- prep: members individually look for defects
  - → most defects found in this stage
- inspection meeting: no resolution of defects, but action items assigned
- third hr: optional additional time used for possible soln discussion + closure of issues
- rework: author revises work prod
- follow-up: ensure defects are corrected + final inspection data is

collected + summarized
- cov of 5%-15% of doc pgs rep significant contribution to quality
- when choosing **inspection target**:
  - ↳ include:
    - sections w/ complicated logic
    - critical sections
    - sections dealing w/ new envs
  - ↳ omit:
    - straightfwd sections
    - sections similar to alr reviewed ones
    - sections not expected to affect fonality if faulty
    - reused sections
- **Pressman's inspection guidelines**:
  - ↳ review product not producer
  - ↳ set agenda + maintain it
  - ↳ limit debate + rebuttal
    - 90 min - 120 min for meeting
  - ↳ identify problem areas, but don't attempt to solve
  - ↳ write notes
  - ↳ limit # participants + prep in advance
  - ↳ develop checklist for each prod
  - ↳ alloc resources + schedule time
  - ↳ train all reviewers
  - ↳ prep report + follow-up proc
  - ↳ review entire process
- **walkthru**: find defects + become familiar w/ dev docs
  - ↳ material distributed in advance
    - only presenter preps
  - ↳ each participant lists potential defects
  - ↳ directed by anyone
- **buddy check**: code walkthru by 1+ reviewers where they read code + report back to author
  - ↳ e.g. pair programming
- **comparison** of review techniques:

inspection	walkthrough	buddy check
presenter is not the author	anyone can be presenter	no presenter
three to six participants	two to seven participants	two or three participants
participants are peers	participants are peers	participants are peers

preparation essential	only presenter needs to prepare	preparation not required
follow-up on corrective actions	no follow-up on corrective actions	no follow-up on corrective actions
data is collected	data collection not required	data not collected
effective but short-term costs	less effort / finds fewer defects	inexpensive / finds fewer defects

- roles in **modern reviews**:
  - ↳ author: corrects problems
  - ↳ reviewer: analyzes + detects bugs
    - often at least 2 reviewers must agree w/ artifact before it's valid
- e.g. Qt code review process



SOFTWARE QUALITY METRICS

- **quality metric**: quantitative measure of deg that software possesses given quality attr
- **measure**: quantitative indication of size of product / proj / process (i.e. single data pt)
  - ↳ e.g. # defects found by 1 test
- **measurement**: act of determining measure
- metric may relate individual measures
  - ↳ e.g. total defects found over a yr
- **indicator**: metric that provides insight in software product / proj / process compared against threshold / std
- objectives of SQ metrics:
  - ↳ assist management in:
    - control of software dev projs + maintenance
    - decision making
    - preventive / corrective action
  - ↳ facilitate managerial control based on calc of metrics + how much they deviate from goal
    - final (i.e. quality) performance
    - timetable + budget performance
- **software size / volume** measures are basis of many SQ metrics
  - ↳ **KLOC**: classic measure by thousands of lines of code
    - lang dependent
    - programming style dependent

- difficult to predict before writing code
- ↳ fcn pts (FP): measures human resources required to dev program based on specified fcnality
  - pre-proj estimates are in terms of required dev resources
    - → assessment based on requirements doc
    - → refined during analysis phase
    - → not dependent on dev tools or langs
  - disadvantages:
    - → detailed specs may be unavail early on
    - → subjective results
    - → domain-dependent (e.g. most successful for data processing systems)

types of metrics:

- ↳ process: improve software process
  - quality metrics
    - → error density metrics
    - → error severity metrics
  - timetable metrics
  - error removal effectiveness
  - productivity
- ↳ product: characteristics of product
  - failure metrics
  - help desk / maintenance services
- ↳ proj: characteristics + execution of proj
  - staffing pattern over life cycle
  - productivity

diff types of process metrics:

Errors Counted Measures

- Number of code errors (NCE) vs. weighted number of code errors (WCE)

	Calculation of NCE	Calculation of WCE	
Error Severity Class	Number of Errors (n)	Relative Weight (w)	Weighted Errors (w*n)
low severity	42	1	42
medium severity	17	3	51
high severity	11	9	99
Total	NCE = 70	---	WCE = 192

↕ The higher the metrics, the lower the quality

Error Density Metrics

Code	Name	Calculation Formula
CED	Code Error Density	CED = NCE / KLOC
DED	Development Error Density	DED = NDE / KLOC
WCED	Weighted Code Error Density	WCDE = WCE / KLOC
WDED	Weighted Development Error Density	WDED = WDE / KLOC
WCEF	Weighted Code Errors per Function Point	WCEF = WCE / NFP
WDEF	Weighted Development Errors per Function Point	WDEF = WDE / NFP

↕ The higher the metrics, the lower the quality

- NCE = number of code errors detected by code inspections and testing
- NDE = total number of development (design and code) errors detected during development process
- WCE = same as NCE but weighted errors
- WDE = same as NDE but weighted errors

Error Severity Metrics

Code	Name	Calculation Formula
ASCE	Average Severity of Code Errors	ASCE = WCE / NCE
ASDE	Average Severity of Development Errors	ASDE = WDE / NDE

- Used when error density metrics are decreasing to detect adverse situations of increasing numbers of severe errors

↕ The higher the metrics, the more severe errors, the lower the quality

- NCE = number of code errors detected by code inspections and testing
- NDE = total number of development (design and code) errors detected during development process
- WCE / WDE = same as NCE / NDE but weighted errors

Software Process Timetable Metrics

Code	Name	Calculation Formula
TTO	Time Table Observance	TTO = MSOT / MS
ADMC	Average Delay of Milestone Completion	ADMC = TCDAM / MS

↕ The higher the TTO metric, the greater the quality

↕ The higher the ADMC metric, the lower the quality

- MSOT = milestones completed on time
- MS = total number of milestones
- TCDAM = total completion delays (days, weeks, etc.) for all milestones (milestones completed before the scheduled date may count either as 0 or as "minus" delays, as long as it is done consistently)

Error Removal Effectiveness Metrics

Code	Name	Calculation Formula
DERE	Development Errors Removal Effectiveness	DERE = NDE / (NDE + NYF)
DWERE	Development Weighted Errors Removal Effectiveness	DWERE = WDE / (WDE + WYF)

↕ The higher the metrics, the more effective, the greater the quality

- NDE = total number of development (design and code) errors detected during development process
- WDE = same as NDE but weighted
- NYF = number software failures detected during a year of maintenance service (or any defined period)
- WYF = same as NYF but weighted failures

Software Process Productivity Metrics

Code	Name	Calculation Formula
DevP	Development Productivity	DevP = DevH / KLOC
FDevP	Function Point Development Productivity	FDevP = DevH / NFP
CRe	Code Reuse	CRe = ReKLOC / KLOC
DocRe	Documentation Reuse	DocRe = ReDoc / NDoc

↕ The higher the DevP / FDevP metrics, the less productive, the lower the quality

↕ The higher the CRe / DocRe metrics, the more reuse, the higher the quality

- DevH = total working hours invested in development of software system
- ReKLOC = number of thousands of reused LOC
- ReDoc = number of reused pages of documentation
- NDoc = number of pages of documentation

limitations of software metrics:

↳ universal:
- enough resources for developing + collecting metrics
- opposition from employees
- uncertainty in data's validity due to maybe biased / partial reporting

↳ software-specific:
- low validity + limited comprehensiveness of metrics
- KLOC isn't good estimate for dev time
- defects detected depend on thoroughness of review + reporting style

REFACTORING

SOFTWARE REFACTORING

- many obvious bugs exist in real code
  - ↳ due to complexity of modern OOP langs, high potential for misuse of features + APIs
  - ↳ simple static code analysis can find many bugs
  - ↳ e.g. bug patterns

Code	Description
Eq	Bad Covariant Definition of Equals
HE	Equal Objects Must Have Equal Hashcodes
IS2	Inconsistent Synchronization
MS	Static Field Modifiable By Untrusted Code
NP	Null Pointer Dereference
OS	Open Stream
RR	Read Return Should Be Checked
RV	Return Value Should Be Checked
UR	Uninitialized Read In Constructor
UW	Unconditional Wait
Wa	Wait Not In Loop

- **preventive software maintenance**: changes aimed to improve curr / future maintainability + reliability of software system
- **refactoring**: process of changing software system st it doesn't alter external behaviour of code, but improves internal struct
  - ↳ cleans up code to minimize chances of introing bugs
  - ↳ **process**:
    1) identify where software should be refactored
       → aka **code smells**
    2) determine which refactorings to apply
    3) guarantee applied refactorings preserve behaviour
    4) apply them
    5) assess effect on design quality
    6) maintain consistency btwn refactored code + other artifacts (e.g. documentation, design docs, tests)
  - ↳ 2 types:
    - **floss refactoring**: frequent + interleaved w/ other kinds of program changes
      → maintain healthy code
    - **root canal refactoring**: infrequent + protracted periods, where few/no other kinds of program changes made
      → correct unhealthy code
- when refactoring OO frameworks, preserve program behaviour while restructuring ops
  - ↳ use **preconds** to examine whether refactoring can be safely applied
    - e.g.

C. *create_member_function.*

Add a locally defined member function to a class that either is unreferenced or is identical to an already inherited function.

Arguments: function F, class C.

Preconditions:

1. ∀ memberFunction ∈ C.locallyDefinedMemberFunctions,
   memberFunction.name ≠ F.name.
   (the function is not already defined locally)

there is no function having the same name with F in class C

2. ∀ F2 ∈ inheritedMemberFunctions(C),
   (F2.name = F.name) ⇒
   matchingSignatureP (F, F2).
   (the signature matches that of any inherited function with the same name)
3. ∀ F3 ∈ functionsThatOverride(F),
   matchingSignatureP(F, F3).
   (the signatures of corresponding functions in subclasses match it)
4. ∀ F2 ∈ inheritedMemberFunctions(C),
   (F2.name = F.name) ⇒
   (∀ class ∈ C ∪ subclassesOf(C),
   unrefdOnInstancesP(F2, class)) ∨
   (semanticallyEquivalantP F, F2).
   (if there is an inherited function with the same name, either the inherited function is unreferenced on instances of C (and its subclasses), or the new function is semantically equivalent to the function it replaces)

history of refactoring:

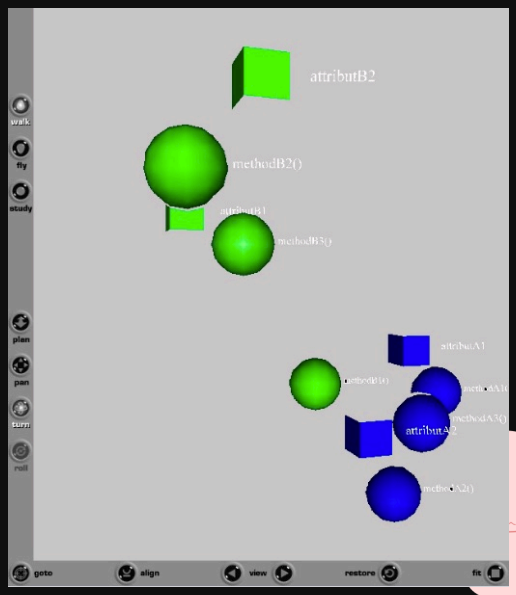
↳ 1999: remove duplicated code

- if same code struct in >1 place, unify them

↳ 2001: metrics based refactoring

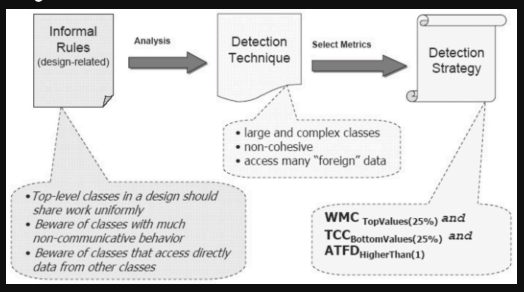
- quantify cohesion btwn attrs & methods:
  $dist(A,B) = 1 - \frac{|A \cap B|}{|A \cup B|} = 1 - similarity(A,B)$
- defn of entity set:
  → for method f is method itself, all directly used methods, & all directly used attrs
  → for attr f is attr itself & all methods using it
- calced dists can be visualized in 3D using VRML
  → manually identify refactoring opportunities
- e.g.

	mA1()	mA2()	mA3()	mB1()	mB2()	mB3()	aA1	aA2	aB1	aB2
mA1()	0									
mA2()	0.5	0								
mA3()	0.4	0	0							
mB1()	0.6	0.6	0.5	0						
mB2()	1	1	1	1	0					
mB3()	1	1	1	0.86	0.6	0				
aA1	0.5	0.67	0.33	0.5	1	0.88	0			
aA2	0.83	0.6	0.5	0.67	1	0.86	0.5	0		
aB1	1	1	1	1	0.5	0.25	1	1	0	
aB2	1	1	1	1	0.33	0.8	1	1	0.75	0



↳ 2000: form detection strategies, which are compositions of various metric rules combined w/ AND + OR ops into single rule

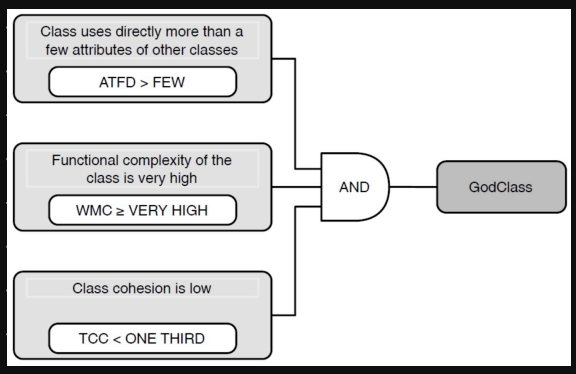
- metric rules: metrics that should comply w/ proper threshold vals
- composition of metric rules rep violation of design heuristic
- e.g. informal design rule → detection strat



- diff types of data filters/metric rules:

Type of Data Filter	Limit Specifiers		Filter Example
Marginal	Semantical	*Relative*	▪ TopValues(10), ▪ BottomValues(5%)
		Absolute	▪ HigherThan(20) ▪ LowerThan(6)
	Statistical		▪ Box-Plot
Type of Data Filter	**Specification**		**Filter Example**
Interval	Composition of two marginal filters, with semantical limit specifiers of opposite polarities		Between(20,30) := HigherThan(20) ∧ LowerThan(30)

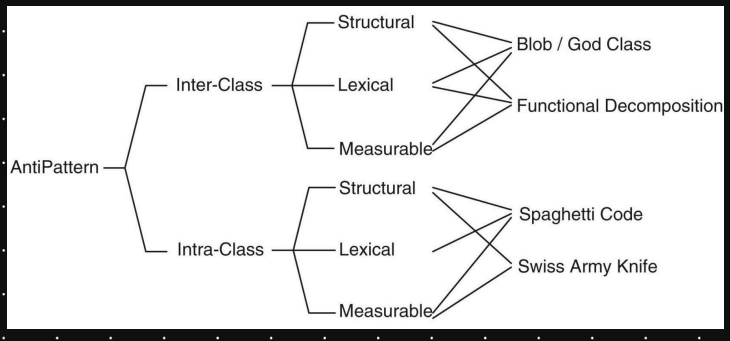
- e.g. god class detection strat



↳ 2009: DECOR is a method for spec + detection of code smells based on domain-specific lang (DSL)

- DETEX is detection technique that instantiates DECOR

classification of anti-patterns:



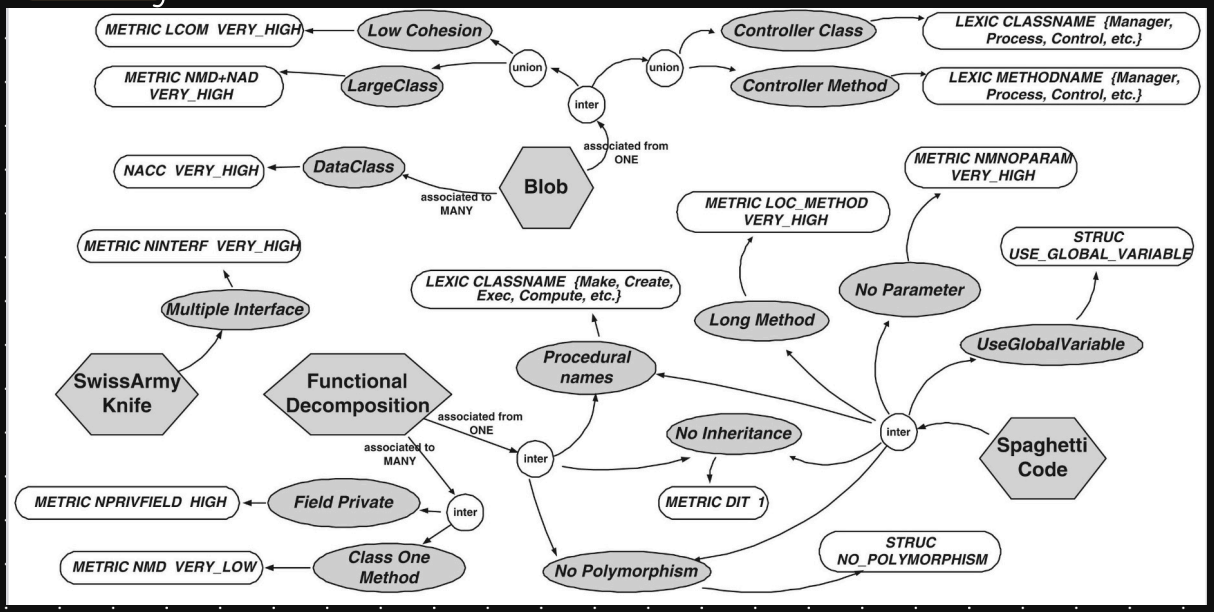
↳ **blob / god class**: knows / does too much + violates single responsibility principle
- large #methods + attrs

↳ **fcnal decomp**: code organized in purely procedural manner w/o proper abstraction + encapsulation
- large fcns that perform unrelated tasks

↳ **spaghetti code**: tangled + unstructured program flow
- excessive use of control stmts
- difficult to follow + maintain

↳ **swiss army knife**: class that tries to do too many unrelated things, resulting in poor cohesion

↳ **taxonomy** of smells:



↳ e.g. detection algo auto generated from **Rule Card**

The *Spaghetti Code* is an antipattern that is characteristic of procedural thinking in object-oriented programming. Spaghetti Code is revealed by classes with no structure, declaring ***long methods*** with ***no parameters***, and utilising ***global variables***. *Names* of classes and methods may suggest ***procedural*** programming. Spaghetti Code does not exploit and prevents the use of object-orientation mechanisms, ***polymorphism*** and ***inheritance***.

```
1  RULE_CARD:SpaghettiCode  {
2    RULE:SpaghettiCode
           { INTER LongMethod NoParamete NoInheritance
           NoPolymorphism ProceduralName UseGlobalVariable };
3    RULE:LongMethod       { METRIC LOC_METHOD VERY_HIGH 10.0 };
4    RULE:NoParameter      { METRIC NMNOPARAM VERY_HIGH 5.0 };
5    RULE:NoInheritance    { METRIC DIT 1 0.0 };
6    RULE:NoPolymorphism   { STRUCT NO_POLYMORPHISM };
7    RULE:ProceduralName   { LEXIC CLASS_NAME
                                (Make, Create, Exec...) };
8    RULE:UseGlobalVariable { STRUCT USE_GLOBAL_VARIABLE };
9  };
```

long methods contain excessive #lines, making them difficult to understand

↳ larger modules (i.e. method / fcn) are associated w/ higher error + change proneness

↳ soln is extract code fragments having distinct fcnality into new methods

- use slice-based metrics to measure lvl of cohesion within module
  - ↳ computed based on slice profiles, which are reps of dependencies btwn parts of module
  - ↳ tightness = $\frac{SL_{int}}{SL_{all}}$
    - SL_{int} = intersection of all SL_i (slice for var $v_i \in V_o$)
    - ratio of # stmts common to all slices
  - ↳ overlap = $\frac{1}{\#slices} \left(\sum_{SL_i} \frac{SL_{int}\ stmts}{all\ SL_i\ stmts} \right)$
    - avg ratio of # stmts that are common to all slices to # stmts in each slice
  - ↳ e.g.

drawWidth	scaleX	scale
		|
|	|	|
	|	
|	|	|
	|	
|		
		|
|	|	|
	|	
|		
		|
|		

↑ slice

```
1  public void paintComponent(Graphics g) {
2    boolean scale = false;
3    drawWidth = available.getWidth();
4    drawHeight = available.getHeight();
5    this.scaleX = 1.0;
6    this.scaleY = 1.0;
7    if (drawWidth < this.minimumDrawWidth) {
8      this.scaleX = drawWidth / this.minimumDrawWidth;
9      drawWidth = this.minimumDrawWidth;
10     scale = true;
     }
11   else if (drawWidth > this.maximumDrawWidth) {
12     this.scaleX = drawWidth / this.maximumDrawWidth;
13     drawWidth = this.maximumDrawWidth;
14     scale = true;
     }
15   Rectangle2D chartArea = new Rectangle2D.Double(
                 0.0, 0.0, drawWidth, drawHeight);
}
```

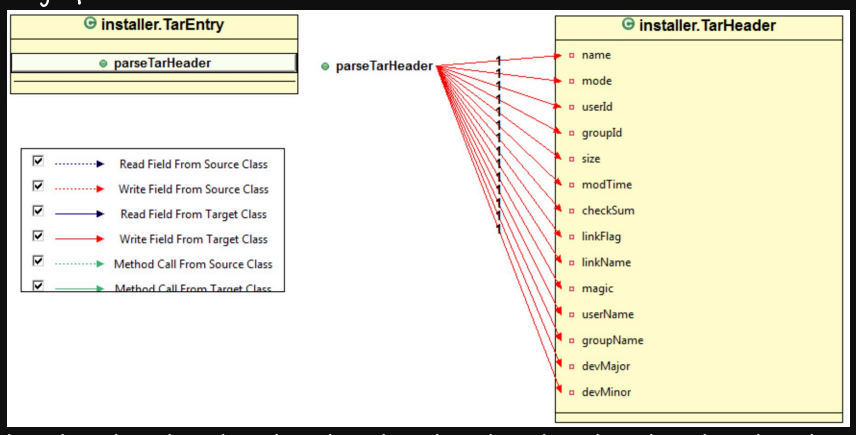
The "|" symbol in position (*x*, *y*) of the slice profile table indicates that the statement in row *x* is required for the computation of the variable in column *y*.

	drawWidth	scaleX	scale
1			
2			|
3	|	|	|
4			
5		|	
6			
7	|	|	|
8		|	
9	|		
10			|
11	|	|	|
12		|	
13	|		
14			|
15	|		

  - tightness = $\frac{3}{15} = 0.2$
  - overlap = $\frac{1}{3}\left(\frac{3}{6} + \frac{3}{6} + \frac{3}{6}\right) = 0.5$
  - ↳ range from [0, 1]
  - ↳ higher tightness + overlap = more cohesive module

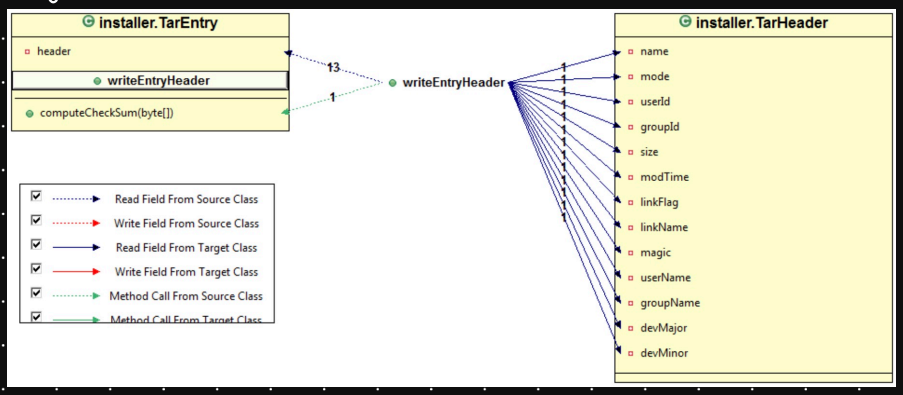
CODE SMELLS

- feature envy: method is more interested in another class that it doesn't belong to
  - ↳ violates design principle that behaviour should be alloced along w/ data accessed
  - ↳ envied data accessed thru param of method or field of class
  - ↳ e.g. parseTarHeader more interested in TarHeader class than its own (TarEntry)



- data class: class containing mostly attrs + very few/no methods accessing them

↳ violates design principle that behaviour should be alloced along w/ data accessed
↳ look for classes w/ instance vars are manipulated by other classes + several public fields
↳ e.g. TarHeader is data class



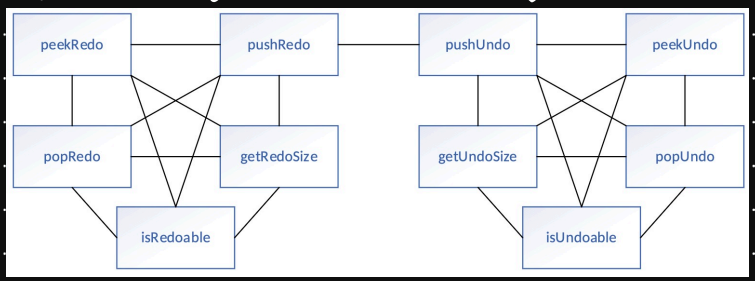
god class : class has multiple responsibilities
↳ violates single responsibility principle
↳ look for huge classes treating other classes like data classes, names containing System, Subsystem, Manager, Driver, or Controller, + has unrelated sets of methods working on separate instance vars (i.e. low cohesion)

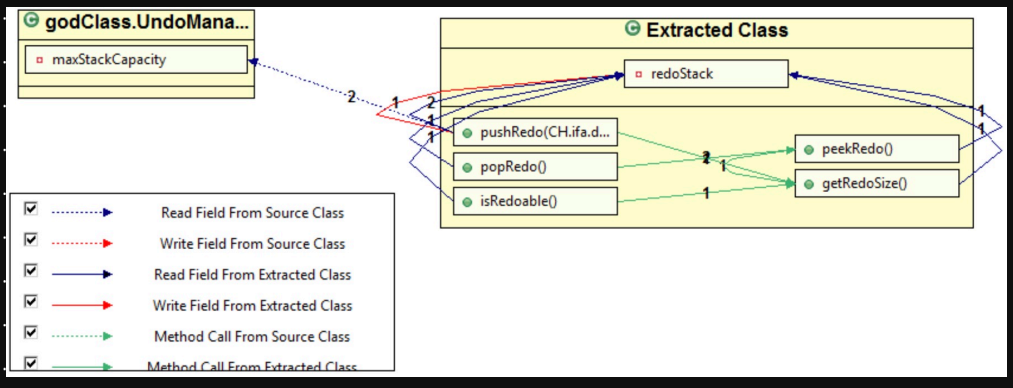
to construct cohesion graph of class:
↳ every method (excluding ctors) is rep as node
↳ connect 2 methods w/ undirected edge if:
- they access common instance var
- 1 calls other

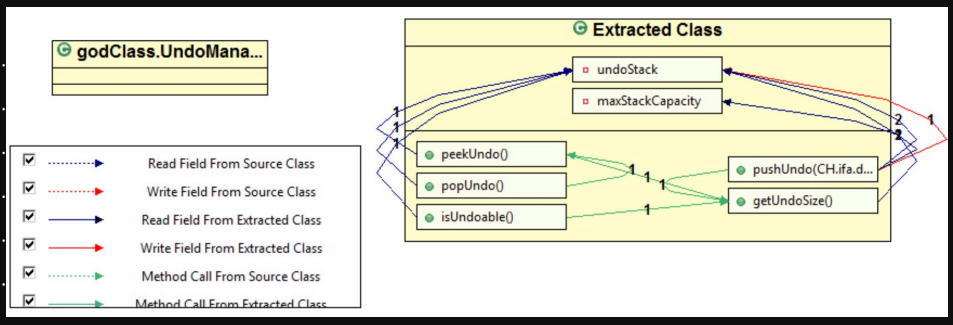
↳ edges due to common field accesses are weightier than those due to calls

to decompose class, find disconnected components or cut graph using max-flow min-cut thm

e.g. cohesion graph for UndoManager







- duplicated code (aka software clones) is believed to be worst code smell
  - ↳ in case of bug / new feature, all duplicates must be found + updated
  - ↳ inconsistent updates lead to errors
- in clone detection, there's 4 types of clones
  - ↳ type 1: code frags are identical except for variations in white space, layout, + comments
    - some clones aren't refactorable b/c they:
      - → return >1 var
      - → return vars w/ diff types
      - → contain conditional return stmts
      - → contain branching stmts (e.g. break, continue) w/o corresponding loop
  - ↳ type 2: in addition to type 1, code frags are structurally + syntactically identical except variations in identifiers, literals, + types
    - e.g.

Diference between mapped statements Data dependence

```
public void testGetFirstMillisecond() {
   Locale saved = Locale.getDefault();
   Locale.setDefault(Locale.UK);
   TimeZone savedZone = TimeZone.getDefault();
   TimeZone.setDefault(TimeZone.getTimeZone("Europe/London"));
   Day d = new Day(1, 3, 1970);
   assertEquals(5094000000L, d.getFirstMillisecond());
   Locale.setDefault(saved);
   TimeZone.setDefault(savedZone);
}
```

```
public void testGetFirstMillisecond() {
   Locale saved = Locale.getDefault();
   Locale.setDefault(Locale.UK);
   TimeZone savedZone = TimeZone.getDefault();
   TimeZone.setDefault(TimeZone.getTimeZone("Europe/London"));
   Hour h = new Hour(15, 1, 4, 2006);
   assertEquals(1143900000000L, h.getFirstMillisecond());
   Locale.setDefault(saved);
   TimeZone.setDefault(savedZone);
}
```

    - solns to refactor:
      - → generalize type
      - → parametrize obj creation
      - → parametrize # literal
  - ↳ type 3: in addition to type 2, code frags w/ further modifications like changed, added, + removed stmts
    - e.g.

```
52  if (state.getInfo() != null)
53    EntityCollection entities = state.getEntityCollection();
54      if (entities != null)
55        String tip = null;
56          if (getToolTipGenerator(row, column) != null)
57            tip = getToolTipGenerator(row, column).generateToolTip(dataset, row, column);
58        String url = null;
59          if (getItemURLGenerator(row, column) != null)
60            url = getItemURLGenerator(row, column).generateURL(dataset, row, column);
61        CategoryItemEntity entity = new CategoryItemEntity(bar, tip, url, dataset, dataset.
          getRowKey(row), dataset.getColumnKey(column));
62        entities.add(entity);
```

```
58  if (state.getInfo() != null)
59    EntityCollection entities = state.getEntityCollection();
60      if (entities != null)
61        String tip = null;
62        CategoryToolTipGenerator tipster = getToolTipGenerator(row, column);
63          if (tipster != null)
64            tip = tipster.generateToolTip(dataset, row, column);
65        String url = null;
66          if (getItemURLGenerator(row, column) != null)
67            url = getItemURLGenerator(row, column).generateURL(dataset, row, column);
68        CategoryItemEntity entity = new CategoryItemEntity(bar, tip, url, dataset, dataset.
          getRowKey(row), dataset.getColumnKey(column));
69        entities.add(entity);
```

- to refactor:
  - → move stmts w/o changing behaviour
  - → abstract behaviour

↳ type 4: 2+ frags perform same behaviour, but implemented thru diff syntactic variants

- e.g.

```
void bubbleSort(int[] arr) {
  int n = arr.length;
  int temp = 0;
  for(int i=0; i < n; i++){
    for(int j=1; j < (n-i); j++){
      if(arr[j-1] > arr[j]){
        //swap elements
        temp = arr[j-1];
        arr[j-1] = arr[j];
        arr[j] = temp;
      }
    }
  }
```

```
void insertionSort(int arr[]) {
  int n = arr.length;
  for (int i = 1; i < n; ++i) {
    int key = arr[i];
    int j = i - 1;
    while (j >= 0 && arr[j] >
key) {
      arr[j + 1] = arr[j];
      j = j - 1;
    }
    arr[j + 1] = key;
  }
}
```

to refactor duplicated code:

↳ if in same class, apply extract method refactoring (i.e. extract duplicated code into new method)

↳ if in diff subclasses of same superclass, apply extract + pull up method (i.e. extract method into superclass)

↳ if in diff + unrelated classes:

- if there's several duplicated instance methods, move into new common superclass
- if methods are static, move into Utility class

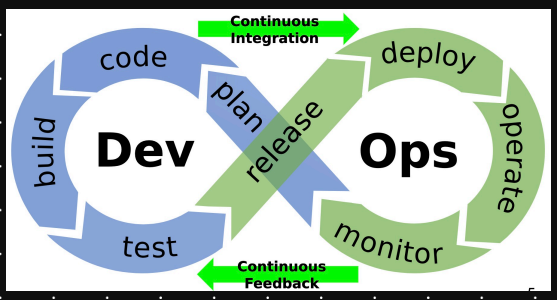
LOGGING

DEVOPS

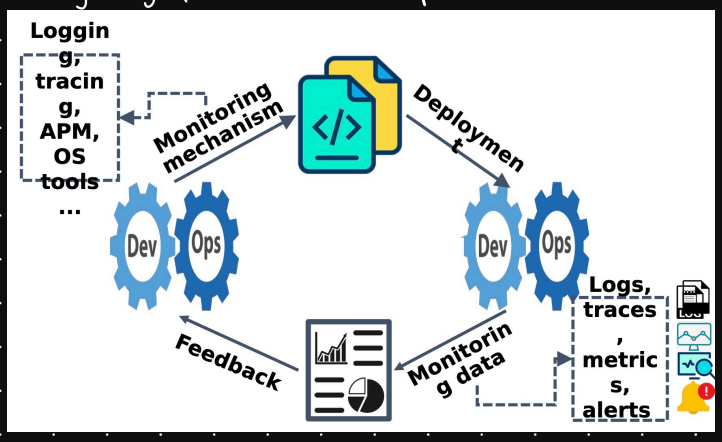
software ops: IT service management to deliver right set of services at right quality + competitive costs for customers
- ↳ manual: traditional system admin
- ↳ automated: repeatedly used scripts (i.e. manually coded rules)
- ↳ **devops**: combine software dev + ops to shorten delivery cycle

traditional dev + ops teams have diff goals
- ↳ dev wants change
- ↳ ops wants stability

devops is culture that aims to break dev-ops barrier + improve collab
- ↳



monitor software systems to understand state + runtime behaviour
- ↳ critical for ensuring quality of delivered services
- ↳ main parts:
  - logs
  - metrics
  - traces
  - alerts
- ↳ bridges gap btwn dev + ops



- ↳ challenges:
  - observing system perturbs it
    - → e.g. monitoring software systems can lead to more performance overhead

- humans can't process as much + complex data as computers can produce

LOGGING

- inserting **log stmts** into code can help w/ debugging, anomaly detection, system comprehension, auditing, etc.
  - ↳ may be only way b/c debuggers aren't always avail (e.g. prod envs)
  - ↳ stay in program while debugging sessions are transient
  - ↳ more efficient than using debugger to trace
- what to log:
  - ↳ performance metrics
  - ↳ timestamp, lvl / severity, thread name, method name, + msg
  - ↳ log msg should ans 5 Ws + how
- we log b/c:
  - ↳ can't diagnose problem w/o data
  - ↳ must for distributed systems
  - ↳ legal compliance
- logging everything equiv to logging nothing
- basic guidelines of **log types**:

Table 18.3 Basic Guidelines of Log Types

What Should be Logged	Description
AAA (Authentication, Authorization, Access)	Successful and failed authentication or authorization decisions; System access, data access, and application component access; and Remote access, including from one application component to another in a distributed environment
Change Events	System or application changes (especially privilege changes); Data changes (including creation and destruction); and Application and component installation and changes
"Badness"/Threats	Invalid inputs and other likely application abuses; and Other security issues known to affect the application
Resource Issues	Exhausted resources, exceeded capacities, and so on; Connectivity issues and problems; and Reached limits
Mixed Availability Issues	Startups and shutdowns of systems, applications, and application modules or components; Faults and errors, especially errors affecting the application's availability; and Backup successes and failures that affect availability

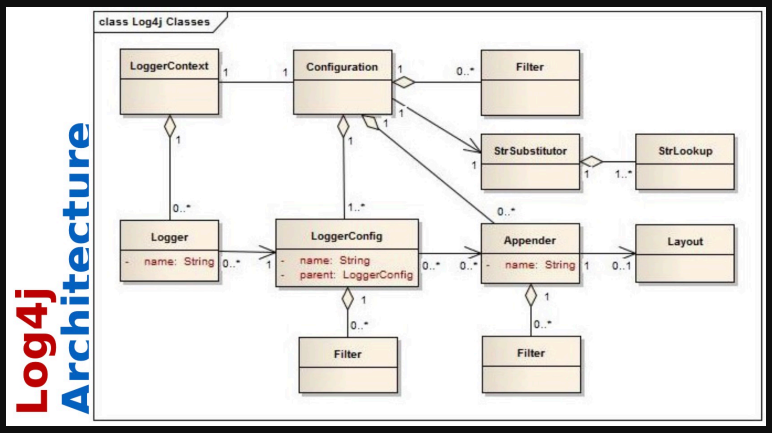
- must consider **log struct**:
  - ↳ unstructured logs require later processing
    - difficult to use grep, sed, awk, + perl (CLI tools for txt manipulation)
  - ↳ logs contain real data from context instead of just msgs
    - e.g. file sizes, latencies
    - easily look for similarily structured data
  - ↳ typically use JSON or csv format
    - can auto transform unstructured to structured
- old-fashioned **file logs** use robust filesystem, but it doesn't scale
  - ↳ if machine doesn't last, logs lost
  - ↳ must manage rotation strat + log expiry (or machine will be full)
  - ↳ difficult to parse + locate important info
- **syslog** fwds logs to aggregator

↳ single central place so won't have to worry abt individual machines running out of storage
↳ still using files + primitive Unix tools

most **modern log architecture** aggregates into search engine
↳ single interface for all log data
↳ search lang is more expressive than grep / regexp
↳ provide **analytics** on top of finding things
- search logs for exceptions
- instant analysis of fields
- behaviour over time
- traffic patterns

LOGGING LIBRARY

use **Log4j** as logging utility / lib
↳ architecture:



each logger has **LoggerConfig** that specifies its config settings
hierarchy of loggers maintained w/ dot struct
↳ e.g. root . parent . child

e.g. getting specific loggers

- To get the root logger
  - Logger logger = LogManager.getLogger();
- To get any other logger
  - Logger analytics = LogManager.getLogger("analytics");

basic code setup for Log4j:

- private static Logger logger = LogManager.getLogger();
- logger.debug("Debug log message");
- logger.info("Info log message");
- logger.error("Error log message");

8 diff **lvls** in Log4j:
↳ if LogLvl isn't defined in config, use parent's lvl

Event Level	LoggerConfig Level						
	TRACE	DEBUG	INFO	WARN	ERROR	FATAL	OFF
ALL	YES	YES	YES	YES	YES	YES	NO
TRACE	YES	NO	NO	NO	NO	NO	NO
DEBUG	YES	YES	NO	NO	NO	NO	NO
INFO	YES	YES	YES	NO	NO	NO	NO
WARN	YES	YES	YES	YES	NO	NO	NO
ERROR	YES	YES	YES	YES	YES	NO	NO
FATAL	YES	YES	YES	YES	YES	YES	NO
OFF	NO	NO	NO	NO	NO	NO	NO

- appenders allow Log4j to print logs to multiple destinations (e.g. raw file, csv, console)
  - ↳ inherited additively from LoggerConfig hierarchy
    - requests are fwded down by default
  - ↳ to stop fwding, use additivity = "false"
  - ↳ e.g.

Logger Name	Specific Appenders	Additivity Flag	Active Appenders
root	A1	na	A1
x	A-x1, A-x2	true	A1, **A-x1, A-x2**
security	S1	false	**S1**
security.access	S1-access	true	S1, **S1-access**

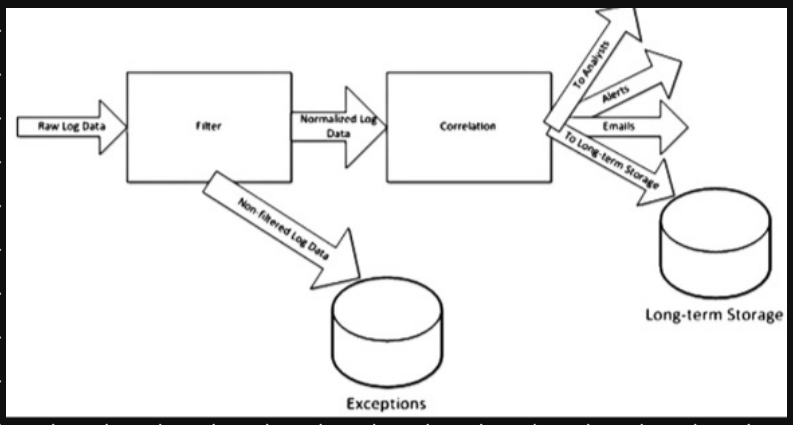
- diff layouts include:
  - ↳ csv
  - ↳ HTML
  - ↳ JSON
  - ↳ text pattern
  - ↳ syslog
  - ↳ datastores (e.g. Mongo, CouchDB, Cassandra)
  - ↳ Kibana
  - ↳ elasticsearch
- ideally, logs should be put in formatted datastore
  - ↳ however, most need regexp to do analysis
  - ↳ e.g.

```
<Pattern>%d %p %c{1.} [%t]
%m%n</Pattern>
```

- %d = date
- %p = level (eg. ERROR)
- %c = name of logger
  - %c{1.} = org.apache.commons.Foo -> o.a.c.Foo
- %t = name of the thread
- %m = the message
- %n = new line
- %n = new line

LOG ANALYSIS

process of **log analysis**:



↳ **raw log data**: 1st input

↳ **filter**: keep data + discard those we don't care abt

↳ **normalization**: parse raw log data into common format for easier analysis

- e.g.

```
- 1132230755 2005.11.17 R36-M1-N0-C:J16-U11 2005-11-17-04.32.35.362512 R36-M1-N0-C:J16-U11 RAS KERNEL FATAL memory and bus summary.....0
- 1132230755 2005.11.17 R36-M1-N0-C:J16-U11 2005-11-17-04.32.35.522718 R36-M1-N0-C:J16-U11 RAS KERNEL FATAL command manager unit summary.....0
- 1132230755 2005.11.17 R36-M1-N0-C:J16-U11 2005-11-17-04.32.35.686057 R36-M1-N0-C:J16-U11 RAS KERNEL FATAL memory manager uncorrectable error.....1
- 1132230755 2005.11.17 R36-M1-N0-C:J16-U11 2005-11-17-04.32.35.853765 R36-M1-N0-C:J16-U11 RAS KERNEL FATAL memory manager strobe gate.....0
- 1132230756 2005.11.17 R36-M1-N0-C:J16-U11 2005-11-17-04.32.36.019773 R36-M1-N0-C:J16-U11 RAS KERNEL FATAL memory manager address parity error.....0
- 1132230756 2005.11.17 R36-M1-N0-C:J16-U11 2005-11-17-04.32.36.182225 R36-M1-N0-C:J16-U11 RAS KERNEL FATAL memory manager miscompare.....0
- 1132230756 2005.11.17 R36-M1-N0-C:J16-U11 2005-11-17-04.32.36.343925 R36-M1-N0-C:J16-U11 RAS KERNEL FATAL memory manager address error.....0
- 1132230756 2005.11.17 R36-M1-N0-C:J16-U11 2005-11-17-04.32.36.498770 R36-M1-N0-C:J16-U11 RAS KERNEL FATAL memory manager store buffer parity.....0
- 1132230756 2005.11.17 R36-M1-N0-C:J16-U11 2005-11-17-04.32.36.654251 R36-M1-N0-C:J16-U11 RAS KERNEL FATAL memory manager RMW buffer parity.....0
- 1132230756 2005.11.17 R36-M1-N0-C:J16-U11 2005-11-17-04.32.36.811117 R36-M1-N0-C:J16-U11 RAS KERNEL FATAL memory manager refresh.....0
- 1132230756 2005.11.17 R36-M1-N0-C:J16-U11 2005-11-17-04.32.36.967017 R36-M1-N0-C:J16-U11 RAS KERNEL FATAL memory manager refresh counter timeout.....0
- 1132230757 2005.11.17 R36-M1-N0-C:J16-U11 2005-11-17-04.32.37.091161 R36-M1-N0-C:J16-U11 RAS KERNEL FATAL memory manager refresh contention.....0
- 1132230757 2005.11.17 R36-M1-N0-C:J16-U11 2005-11-17-04.32.37.256152 R36-M1-N0-C:J16-U11 RAS KERNEL FATAL state machine.....0
- 1132230757 2005.11.17 R36-M1-N0-C:J16-U11 2005-11-17-04.32.37.429270 R36-M1-N0-C:J16-U11 RAS KERNEL FATAL memory manager / command manager address
parity..0
- 1132230757 2005.11.17 R36-M1-N0-C:J16-U11 2005-11-17-04.32.37.588217 R36-M1-N0-C:J16-U11 RAS KERNEL FATAL regctl scancom interface.....0
```

Log Description	RAS KERNEL FATAL memory and bus summary	RAS KERNEL FATAL command manager unit summary	RAS KERNEL FATAL memory manager uncorrectable error	RAS KERNEL FATAL memory manager strobe gate	RAS KERNEL FATAL memory manager address parity error	RAS KERNEL FATAL memory manager miscompare	RAS KERNEL FATAL memory manager address error	RAS KERNEL FATAL memory manager store buffer parity	RAS KERNEL FATAL memory manager RMW buffer parity	RAS KERNEL FATAL memory manager refresh	RAS KERNEL FATAL memory manager refresh counter timeout	RAS KERNEL FATAL memory manager refresh contention	RAS KERNEL FATAL state machine	RAS KERNEL FATAL memory manager / command manager address parity	RAS KERNEL FATAL regctl scancom interface
Log Value	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0

↳ **correlation**: grouping events tgt using a common prop so we get more complete view

- e.g.

#	Log lines	Group
1.	accountId(*Tom*) User purchase, item=*100*	Tom
2.	accountId(*Jerry*) User purchase, item=*100*	Jerry
3.	accountId(*Tom*) Update shopping cart, item=*100*	Tom
4.	accountId(*John*) User purchase, item=*103*	John
5.	accountId(*Tom*) User checkout	Tom
6.	accountId(*Jerry*) Update shopping cart, item=*100*	Jerry
7.	accountId(*John*) User purchase, item=*105*	John

Event ID	Event Template
E_1	User purchase, item=$v
E_2	Update shopping cart, item=$v
E_3	User checkout

Parsing and normalizing logs

Group	Event ID	Log line #
Tom	E_1	1
	E_2	3
	E_3	5
Jerry	E_1	2
	E_2	6
John	E_1	4
	E_1	7

Correlating events by user na

- **rule correlation**: crafting rule that models certain behaviour
  - → e.g.:

Event ID	Event Template
E_1	User purchase, item=$v
E_2	Update shopping cart, item=$v
E_3	User checkout

E1 => E2 thousands of times
E1 => E1 1 time

E1 => E1 becomes a rule to detect anomalies

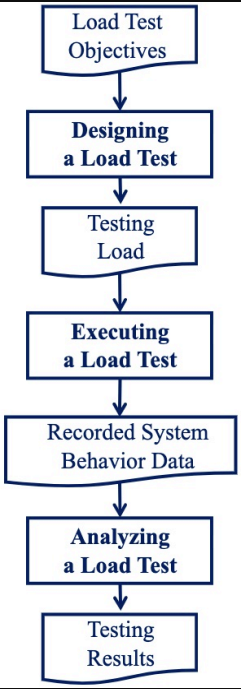
↳ **action**: what to do after correlation has occurred
  - e.g.:

For example, if we receive log data from firewalls and IDSes, we can capture reconnaissance attempts followed by a firewall policy violation with the following rule:
If the system sees an event E1 where E1.eventType=portscan followed by
an event E2 where E2.srcip=E1.srcip and E2.dstip=E1.dstip and E2.eventType=fw.reject then
doSomething

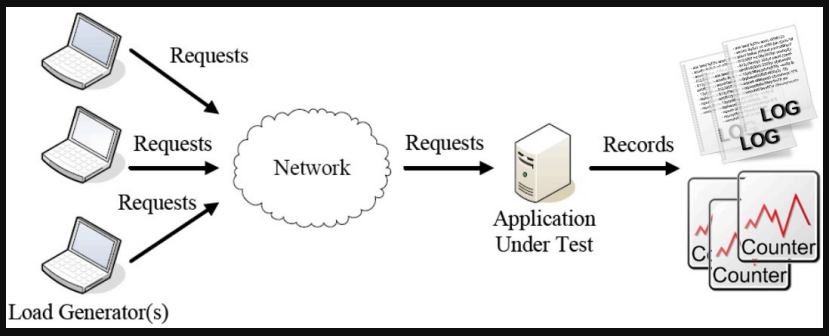
using AIOps, filtering, normalization, & correlation can be done by AI

LOAD TESTING

roadmap to load testing consists of 3 main steps: designing, executing, + analyzing load test



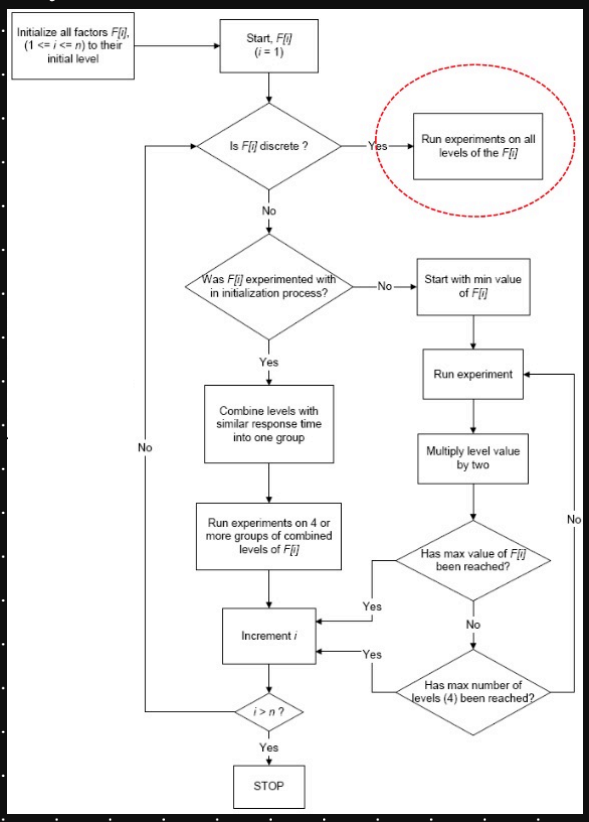
load testing mimics multiple users repeatedly performing same tasks
- ↳ can take hrs / days



experimental design aims to obtain max info w/ min # experiments
- ↳ response var: outcome of experiment
  - e.g. thruput + response time
- ↳ factor: var that affects response var + has several alts
  - e.g. to measure workstation performance, factors include CPU type, mem size, #disk drives, + workload
- ↳ lvl: val that factor can have
  - e.g. lvls of mem size could be 2 GB, 6 GB, + 12 GB
- ↳ replication: repetition of all / some experiments
- ↳ interaction effects: 2 factors A + B interact if effect of 1 depends on other

ad-hoc approach is to iterate thru all (discrete + continuous) factors + identify which

affect performance
↳ e.g.



- t-way covering arr : set of configs where each valid combo of factor-vals for every combo of t factors appear at least once
  ↳ e.g.

  Suppose a system has 5 user configuration parameters. Three out of five parameters have 2 possible values (0, 1) and the other two parameters have 3 possible values (0, 1, 2). There are total $2^3 \times 3^2 = 72$ possible configurations to test.

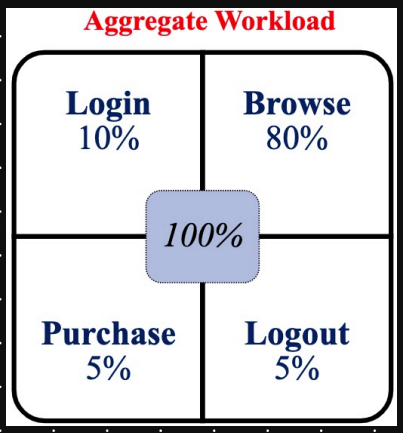
- other kinds of covering arrs include var-strength + test-case-aware
- combinatorial interaction testing (CIT) models system under test as set of factors, then gens sample that meets specific cov criteria

DESIGNING A LOAD TEST

- main parts:

Designing a Load Test	
Designing Realistic Loads	Designing Fault-Inducing Loads
Load Design Optimizations and Reductions	

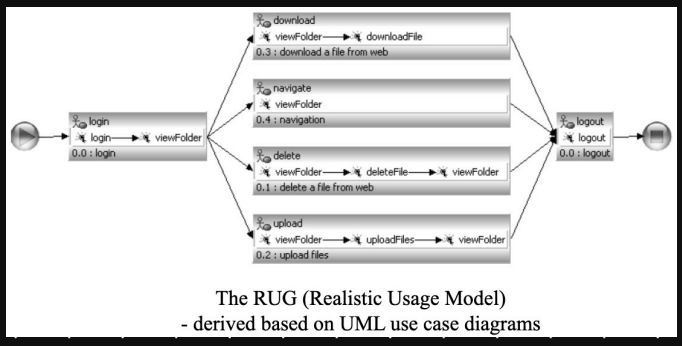
- characterize aggregate workload (1 way to design realistic load) by using:
  ↳ workload mix
    - e.g. 30% browsing, 10% purchasing, + 60% searching
  ↳ workload intensity : rate of requests
  ↳ e.g. e-commerce system



↳ 2 types of loads:
- steady load : same workload mix + intensity
  - → ease of measurement
  - → might result in mem leaks
- step-wise load : same workload mix + step-wise inc in intensity

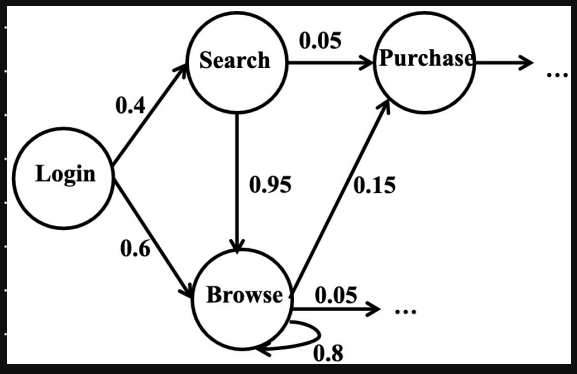
↳ derive testing loads from historic data
- if missing past usage data, extrapolate from:
  - → beta-usage data
  - → interviews w/ domain experts
  - → competitors' data

another way to design realistic load is w/ use-case
↳ can use UML diagrams
- e.g.



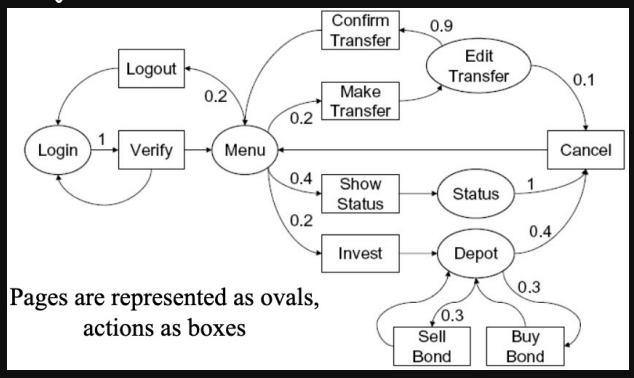
The RUG (Realistic Usage Model)
- derived based on UML use case diagrams

↳ can use Markov chains
- e.g.



↳ can use Stochastic form-oriented model to show more details than Markov chain (e.g. can rep failure + success of login)

- e.g.



2 ways to design **fault-inducing** loads:

↳ **src code analysis**

- **data flow analysis** identifies potential modules + regions for load sensitive faults (e.g. mem leaks + incorrect dynamic mem alloc)
  - → annotate CFGs of malloc / free calls w/ size
  - → **load sensitivity index (LSI)**: net inc/dec of heap space after every iteration
  - → write test cases that exercise regions w/ highest LSI vals
- **symbolic execution** explores paths w/ symbolic inputs instead of concrete vals
  - → e.g.

```
y = x;
if (y > 0) then y++;
return y;
```

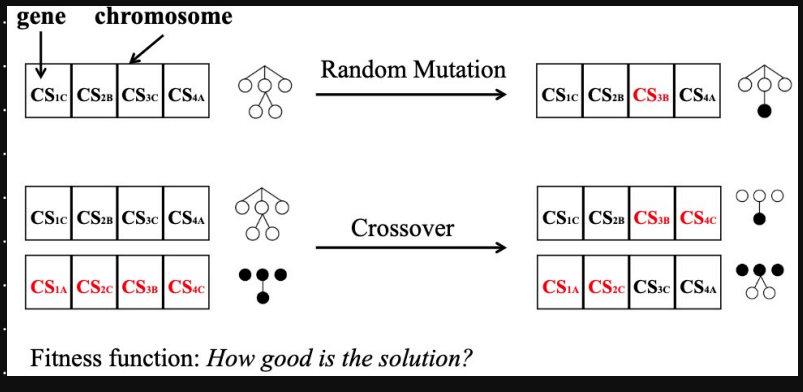
Two path conditions:
- $x > 0$
- $x \leq 0$

Path Performance Estimation (Response Time)
- Weight of 10 for invoking bytecode
- Weight of 1 for all other methods

Memory Analysis
- Uses Java PathFinder's built-in object lifecycle listener mechanism to track the heap size of each path

↳ **system models** rep behaviour + interactions of components

- **genetic algos**



→ e.g. service-oriented architecture (SOA)
- each gene rep particular type of web service
- chromosome is resulting workflow
- fitness fcn states that risky workflow has high response time

- diff ways to **reduce load designs**:
  - ↳ extrapolation for step-wise testing
    - only examine a few load lvls
    - extrapolate system performance for other lvls
  - ↳ use probability mass cov to assess how well test cases cover #inputs/states
    - e.g.

Probability	# of States
0.3	34
0.4	51
0.5	72
0.6	99
0.7	137
0.8	206
0.9	347
0.99	721
0.999	843
1.0	857
Test Coverage	

- realistic load tests are based on historical **field workloads**, but those change over time
  - ↳ must periodically update + maintain load profiles

EXECUTING A LOAD TEST

- main parts:

Executing a Load Test	
Live-user Based Execution	Driver Based Execution
Setup	
Load Generation and Termination	
Test Monitoring and Data Collection	

- **live-user-based** test execution coordinates live users to test tgt
  - ↳ users selected based on diff criteria (e.g. loc, browser type)
  - ↳ pros:
    - reflects realistic user behaviour
    - obtain real user feedback
  - ↳ cons:
    - hard to scale
    - limited complexity due to manual coordination
- **driver-based** test execution
  - ↳ can use:
    - specialized benchmarking tools
    - centralized load drivers

→ easy to control load, but hard to scale
- peer-to-peer load drivers
→ easy to scale, but hard to control load

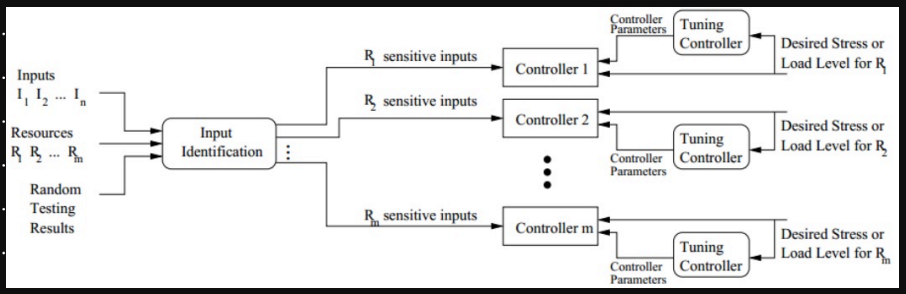
↳ pros:
- easy automation
- scale to large # requests

↳ cons:
- load driver configs
- hard to track some system behaviour (e.g. audio / img quality)

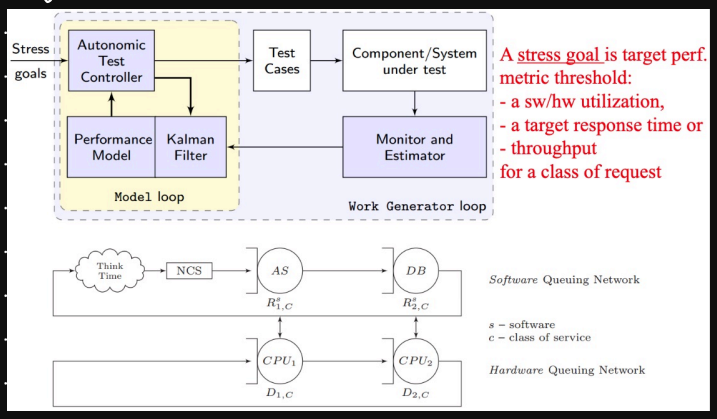
3 general aspects while executing load test:

↳ test setup
- things to consider when doing system deployment:
→ field load testing
- costly but realistic
→ selection of hardware
- dedicated hardware or cloud-based testing
→ creating realistic dbs
- importing realistic raw data
- sanitizing field db
→ mimicking realistic network traffic
- network latency + spoofing
→ don't deploy drivers on same machines w/ SUT
- test execution setup can differ
→ for live-user-based, must do test recruitment, setup, + training
→ for driver-based, must do programming, store + replay config, + model config

↳ load gen + termination can be separated into 2 categories:
- static config
→ timer-driven
→ counter-driven
→ statistic-driven
- dynamic feedback
→ steer testing loads based on system feedback
→ start w/ random testing to identify performance sensitive input
- e.g.



→ can use 2-layered queuing model
- e.g.



- live-user-based only works w/ static config
- driver-based works w/ both static & dynamic

↳ test monitoring & data collection
- test monitoring tools can be agentless (e.g. Task Manager, JConsole) or agent-based (e.g. App Dynamics, CA Willy)
- measurement bias is hard to avoid & unpredictable, but we can lessen it by repeating measurements & randomizing experiment setup
  → e.g.

> **Example 1**: How come the same application today runs faster compared with yesterday?
>
> **Example 2**: Why the response time is very different when running the same binary under different user accounts?
>
> **Example 3**: Why the code optimization only works on my computer?

ANALYZING A LOAD TEST

main parts:

Analyzing a Load Test		
Verifying Against Threshold Values	Detecting Known Problems	Automatically Detecting Anomalous Behavior

- methods to verify against threshold vals:
  ↳ straightfwd comparison w/ target
  ↳ compare against processed data
    - e.g. max, min, 90th percentile
  ↳ compare against derived data
- to detect known problems, use patterns
  ↳ mem leak detection w/ patterns in mem utils
    - e.g.

↳ error keywords w/ patterns in logs

- e.g.

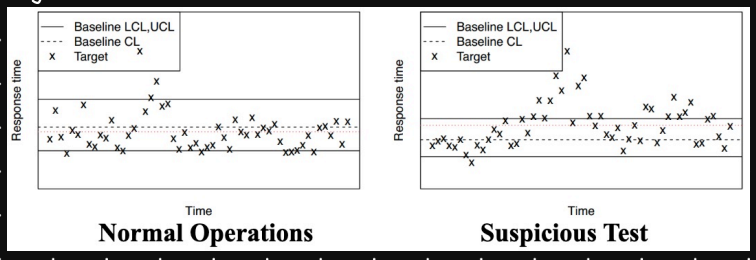
- A large-scale enterprise system can generate 1.6 million log lines in an 8-hour load test
  - 23,000 lines contain "fail" or "failure"
  - How many types of failures are there in this test?

Events	Frequency
Error occurred during purchasing, item=$v	500
Error! Cannot retrieve catalogs for user=$v	300
Authentication error for user=$v	100

automated detection of anomalous behaviour : auto derive normal behaviour + flag anomalies

↳ use control charts to derive performance ranges from past good tests

- e.g.



↳ derive performance rules + flag tests when rule is broken

- e.g.

- **From past tests, we can extract rules such as:**

DB read/sec **Medium** + Throughput **Medium** → Request Queue size **Low**

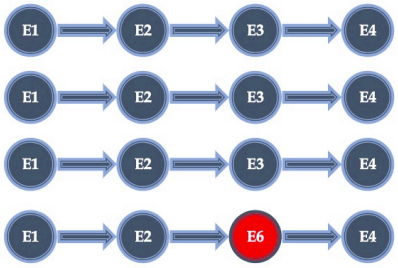
- **Flags tests where the rule does not hold**

DB read/sec **Medium** + Throughput **Medium** → Request Queue size **High**

↳ derive final behaviour + flag anomalies

- e.g.

- (E2, E3) are always together:
  - (acquire_lock, release_lock)
  - (open_inbox, close_inbox)
- If we see (E2, E6), this might be a problem



CHAOS, AB, AND TOGGLES

CHAOS

- chaos is expensive IRL, but can be almost free + automated in cloud
  - ↳ added in prod in middle of business day w/ careful monitoring + engineers on standby so weaknesses can be identified + auto-recovery mechanisms can be built
- **chaos engineering** process:
  - ↳ define steady state of system to set baseline for normal behaviour
  - ↳ hypothesize steady state will cont in both control + experimental group
  - ↳ intro vars that reflect real world events
    - e.g. crashing servers, malfcning hard drives, + severed network connections
  - ↳ try to disprove hypothesis by looking for diff btwn 2 groups
- **chaos monkey**: randomly disables prod instances to ensure system is robust enough to survive this common type of failure w/o customer impact
- **latency monkey**: induces artificial delays in communication layer to simulate service degradation
  - ↳ measures if upstream services respond appropriately
  - ↳ large delays simulate node / service being down w/o acc bringing it down
  - ↳ tests fault tolerance of new service
- **doctor monkey**: runs health checks + monitors other external health signs on each instance
  - ↳ check for + remove unhealthy instances
  - ↳ keep unhealthy service up only for devs so problem can be found
  - ↳ once root cause found, terminate service
- **janitor monkey**: ensures env is running free of clutter + waste
  - ↳ searches for unused resources + disposes them
- **conformity monkey**: checks for + shuts down instances that don't adhere to best practices
  - ↳ e.g. service that isn't part of auto-scaling will fail when requests surge so shut it down now instead of letting it fail during peak period
  - ↳ **security monkey**: special case that finds + terminates services w/ security violations / vulnerabilities
    - ensures all SSL + DRM certificates are valid + not expiring soon
- **10-18 localization-internationalization monkey**: detects config + runtime problems in instances serving customers in diff geographic regions
- **chaos gorilla**: simulates outage of entire availability zone
  - ↳ verify services re-balance to fcnal availability zones w/o user impact + manual intervention
  - ↳ e.g. remove NA Amazon instances + run from Europe

A/B AND HYPOTHESIS TESTING

- process of **A/B testing**:
  - ↳ use toggles + field trials
    - some users see ver A
    - some users see ver B
  - ↳ logging
    - measure clicks, conversions, sales, etc.
  - ↳ analytics
    - statistically compare A + B to determine if there's significant diff
- process of **statistical hypothesis testing**:
  - ↳ null hypo = no diff btwn A + B
  - ↳ run statistical test
  - ↳ reject null hypo w/ particular confidence
    - e.g. 95% → $p \leq 0.05$
  - ↳ determine direction (i.e. $A > B$ or $A < B$) + magnitude of diff
- always run **randomized trials** to avoid systematic bias
  - ↳ don't have to send same # users to use statistical test
  - ↳ need statistically significant sample
  - ↳ if testing multiple hypos, may need correction (e.g. Bonferroni)
- start introing new feature gradually
  - ↳ e.g. only 1% of users can see it
- once sample size for new feature is large enough, run a test
  - ↳ e.g.

```
• If countB >= 100
  ◦ //compare a similar sized sample of A and B results
  ◦ Wilcoxon(bResults(), rand(countB, AResults())
• if p > 0.05
  ◦ Wait for more B results
  ◦ If countB >= 400 and still p > 0.05 likely that there is no difference
```

  - ↳ w/ v large #s, small diffs can still be statistically significant
    - depend on effect sizes
- e.g.

```
• "Offer ends this Saturday!"
  ◦ 50/1000 = 5%
• "Offer ends soon!"
  ◦ 30/1000 = 3%
• clicks <- c(50, 30)
• trials <- c(1000, 1000)
• prop.test(clicks, trials)
• p-value = 0.03015
• Magnitude
  ◦ 20 additional sales or 1.4 times more sales
```

- to choose **statistical test**:

Assumed Distribution	Data type	Example Case	Test
Gaussian/Normal	Continuous	Average Revenue per Paying User	Welch's t-test
Binomial	Ratio (Percentage of successes)	Click Through Rate	Fisher's exact (small) Chi-squared (large)
Poisson	Count	Num transactions per paying users	E-test
Multinomial	Count per category	Num of each product purchased	Chi-squared test
No assumption	Continuous	Time between requests for same file	Wilcoxon or Mann-Whitney test

- e.g.

Scroll prediction

Predicts the finger's future position during scrolls allowing time to render the frame before the finger is there. , – Mac Windows, Linux, Chrome OS

#enable-scroll-prediction

- Speed to render frame
- Number of bytes
- Hypothesis: With scroll prediction rendering will be 2 times faster with an increase of 1.25 more bytes.

- e.g.

Single-click autofill

Make autofill suggestions on initial mouse click on a form element. – Mac, Windows, Linux, Chrome OS, Android

#enable-single-click-autofill

- Average number of autofill fields that are later corrected
- Average speed to fill a field
- Threshold: With autofill, on average, users will correct less than 25% of the fields
- Hypothesis: With autofill, on average, users will complete a field in 50% less time.

- e.g.

Enable picture in picture.

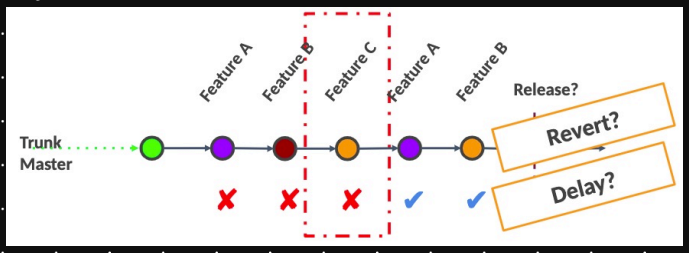
Enable the picture in picture feature for videos. – Mac, Windows, Linux, Chrome OS

#enable-picture-in-picture

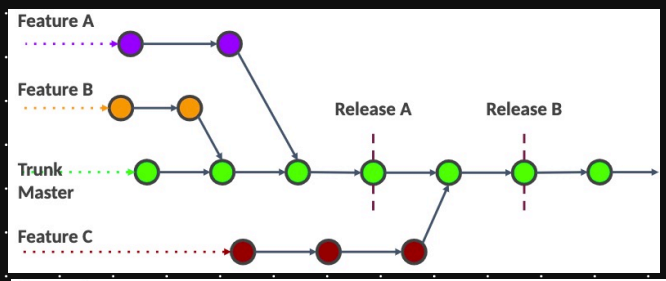
- Raw count of number users, if below threshold, may be able to eliminate the feature
- Threshold: With p-in-p enabled, 2% of users will use p-in-p feature

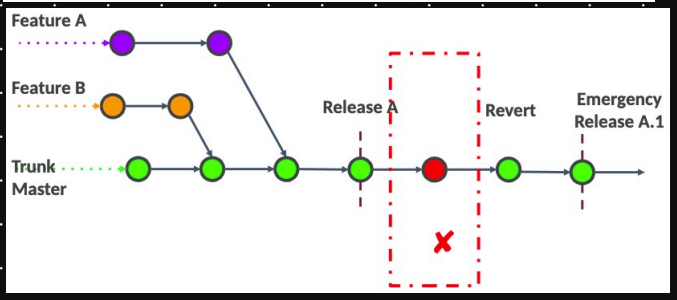
FEATURE TOGGLES

- trunk-based development: all developers work on single main shared branch, committing changes frequently to it
  - ↳ issue on how to deal w/ feature that's integrated but not rdy for release
    - e.g.

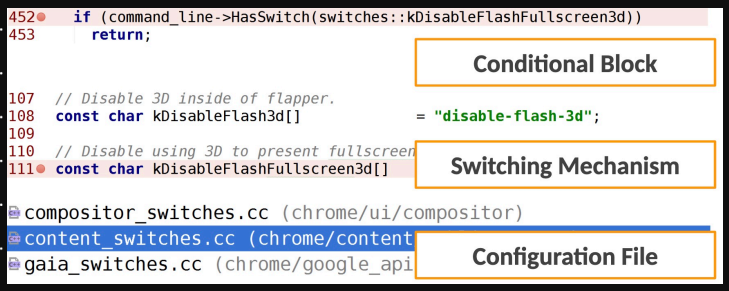


- if feature management is done w/ branching, can lead to merge hell + hard to revert
  - ↳ e.g.

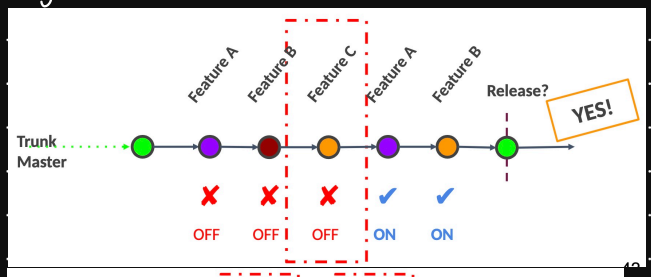


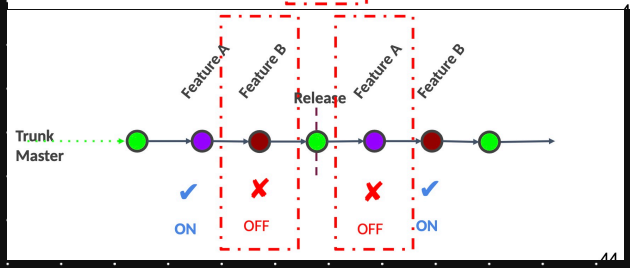


feature toggles allow us to hide certain features for release
- ↳ e.g.



- ↳ trunk dev w/ feature toggles is much easier
  - e.g.





if there's error after release, toggle off new feature
- ↳ don't need to make new build
- ↳ fix problem + gradually roll out again

feature toggles change dev mentality
- ↳ old + new feature code compiled into same build

- ↳ must be able to toggle on old ver
- ↳ features are more isolated

- advantages of toggles:
  - ↳ reconciling rapid release + long-term feature dev
  - ↳ flexible feature rollout
  - ↳ fast context switches
- toggle inertia / debt: can't remove toggles until we're sure new feature is working
  - ↳ i.e. old code is live, but commented out
  - ↳ delay can result in forgetting what to remove or not doing it b/c we've moved on to other tasks
- when combining features, test + gradually rollout any feature combo to be deployed to prod