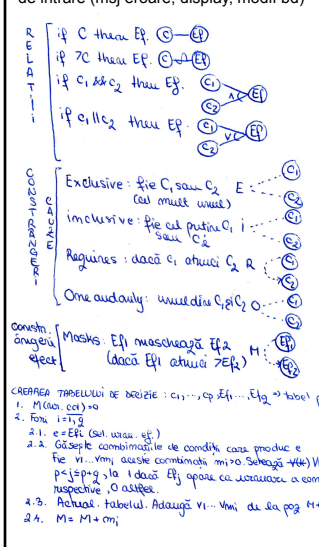
Testare funcțională = tip de testare software care validează în raport cu cerințele funcționale. **Input** gen. pe baza cerințelor programului alcătuit din pre+post-condiții. Structura codului nu influențează. Se bazează pe **partiționare a intrărilor**⇒clase de elem. cu prop. similare.
Parț. echivalență-elem. tratate identic⇒alege reprezentant arbitrar. Clasele nu se suprapun. **+** reduce nr. date, bun când input și output ușor de identificat, valori distincte. **-** nu poți identifica riguros clasele, uneori nu pot fi procesate identic, puțin aplicabile când parcurgerea este complexă.
Algo: • stab. domeniu de intrări + restricții • fă clasele de echivalență • stab. domeniul de ieșiri⇒clase de echiv • clasele globale=combină clasele de input/output
IN: n întreg pozitiv, x șir char, c char, s=y/n, 1<n<20⇒N1={1,20} N2={n<1} N3={n>20} S1=[y] S2=[n]
OUT: poz dacă c in x, msj⇒C1={c in x} C2={c nu în x}⇒combinări, 6 clase tabel intrări+expected out
Val. frontieră-exam. capetele de interval, sursă importantă de erori.
pt n 0, 1, 20, 21, pt c pe prima/ultima poz⇒N1={1,20} N2={0} N3={21} C1, C2⇒alege testele în funcție de valorile astea
Parț. categorii-gen. date de test care acoperă funcționalitate+maxim. pos. găsire erori. **+** produce teste mai cuprinzătoare **-** bazează pe experiența testerului, pot rezulta date de mare dimens.
Algo: (1)Descompune funcționalitatea în unități testate separat (2)Pt. fiecare unitate identifică param+condiții de mediu (starea sistemului, de ce depinde comportamentul)(3)Găsește categoriile (pp. sau caract. import) fiecărui param/condiț.(4)Parț în alternative = o mulț. de valori similare (5)Scrie specificația de testare = listă categorii și lista alternativelor pt categ (6).Creează cazuri de testare prin alegerea unei combinații de alternative din specif. de test. (fiecare categ. contribuie cu 0 sau 1 altern)
(7)Creează date de test alegând câte o singură valoare pt fiecare alt.
1. - avem o singură unitate., 2 - param: n,x,c,s
3. categorii: n dacă în [1,20], x dacă lung. minimă, maximă, intermediară, c dacă ocupă prima sau ultima sau poz din interior lui x sau dacă nu apare x , s dacă e y/n
4. Alternativele: - n: <0, 0, 1, 2..19, 20, 21, >21 (restul ca sus)
5. Specificația de testare:
n: - n < 0; 0; 1 [ok, lung1], 2…19[ok, lung_medie], 20, [ok, lungime20], 21, n>21
x: - {x| |x| =1} [if ok and lungime1]
- {x| 1<|x|<20} [if ok and lung_medie]
- {x| |x|=20} [if ok and lung20]
c - {c| c pe 0 în x} [if ok]
- {c| c în interior x} [if ok and not lung1]
- {c|c ultimu în x} [if ok and not lung1]
- {c|c nu în x} [if ok]
s - y [if ok] sau - n [if ok]
⇒ 7 * 3 * 4 * 2 = 168 cazuri, unele nu au sens și pot fi eliminate adăugând constrângeri⇒ reducem la 24
Metoda Grafului Cauză-Efect(modelarea dependențelor)[Black box]Cauză = orice condiție în specif(cerințe) care poate afecta output. Efect = răspunsul programului la o combinație de condiții de intrare (msj eroare, display, modif bd)

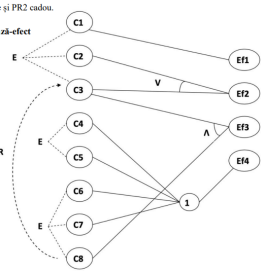


Crearea grafului cauză-efect (exemplu)
după Aditya P. Mathur, Foundations of Software Testing, Pearson Education 2008.

Cerințe: O companie vinde pe web calculatoare (CPU1, CPU2, CPU3), imprimante (PR1, PR2), monitoare (M20, M23, M30) și memorie adițională (RAM256, RAM512, RAM1G). O comandă cuprinde între 1 și 4 articole, cel mult câte unul dintre cele 4 categorii amintite. Interfața grafică constă în 4 ferestre (pentru cele 4 categorii de produse) și o fereastră în care sunt afișate articolele primite cadou.

Monitoarele M20 și M23 pot fi cumpărate cu oricare CPU sau singure. M30 poate fi cumpărat doar împreună cu CPU3. PR1 este oferită cadou la cumpărarea lui CPU2 sau CPU3. Monitoarele și imprimantele, în afară de M30, pot fi cumpărate separat, fără a cumpăra și CPU. La cumpărarea unui CPU1 se primește RAM256 upgrade, iar la cumpărarea unui CPU2 sau CPU3 se primește RAM512 upgrade. La cumpărarea unui CPU3 și a unui M30 se primește RAM1G upgrade și PR2 cadou.

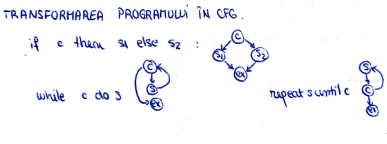
Construirea grafului cauză-efect
C1: Cumpărare CPU1
C2: Cumpărare CPU2
C3: Cumpărare CPU3
C4: Cumpărare PR1
C5: Cumpărare PR2
C6: Cumpărare M20
7: Cumpărare M23
8: Cumpărare M30
1: RAM256
2: RAM512 și PR1
3: RAM1G și PR2
Ef4: nici un cadou



Generarea cazurilor de testare.Fiecare coloană din tabelul de decizie generează cel puțin un caz de testare, corespunzător combinației C1, ..., Cp respective.(Obs) C1, ..., Cp sunt în general expresii care folosesc variabile deci pentru o combinație pot fi selectate mai multe cazuri de testare.**Problemă**: Explozie a stărilor datorită combinației de cauze. **Soluție**: Limitarea numărului de cazuri de testare folosind euristici.

Udacity CS258. The ultimate goal is to have software that works reliably by finding and fixing failures (bugs). The key to addressing the testing problem is breaking it down into smaller subproblems and applying known techniques and previous experiences. selecting a good set of test inputs, and designing good acceptability test end up being actually really hard. Testing involves selecting good test inputs and designing acceptable tests, which can be challenging. The goal of testing is to find bugs as early as possible, as early bug detection is cheaper to fix. Mars climate Orbiter⇒basic unit error. SUT should have clean code, be refactored, avoid unnecessary threads, global vars, excessive pointers, modules should have unit tests, software should support fault injection, make assertions to validate expected behavior.
Fixed size Queue constructor-param. size, enqueue returns A/F, dequeue returns elem/None, when overflow pointer tail/head reset to end/first elem. *Ex.* test: init q, check empty, nq elem, check a/f, dq elem, check value, dq empty, check None…
Assertion: Executable check for a property that must be true for your code. they are not for error handling, have no side effects, check nontrivial property that could be wrong. It's not something that could be wrong if the user did something wrong; make code self-checking, effective testing, assign blame, code fail early, document assumptions, preconditions, postconditions, invariants.
checkRep checks the variables in the program for self-consistency, terminate the program if some invariant. Often when we are testing software, we are not so much just looking for bugs in the software, but we are hoping to refine the specification for the SUT. **Crashme** works by allocating a block of memory and filling it with random garbage. The goal is to explore and test operating system calls with unusual and unexpected values. **defensive coding** i.e, error checking for its own sake to detect internal inconsistencies. *golden rules of testing is we shouldn't ever just hope that it does something; we need to actually check this.* **Faults injected** into SUT should be faults that we want our code to be robust to. Timing of inputs is important for: SUT interacts with hardware, SUT interfaces with other machines on the web, multi-thread.
Differential Testing=taking the same test input delivering it to 2 different implementations of the SUT and comparing them for equality.***Stress Testing***=test beyond usage limits. for sqrt test large/tiny nr, OS make huge nr sys calls, mems allocation, large files, web server many requests. When simulating a noisy wire for a SUT for cars. When testing features for a web service⇒system/validation testing.

Testare structurală **Input/test** gen. pe baza implementării programului fără a lua în considerare cerințele; sunt alese aî să parcurgă toate elem. grafului măcar o dată (Control Flow Graph, nodurile sunt linii de cod).



Metrici/acoperiri: • **nivel instrucțiune**=fiecare instr.(nod) parcurs măcar o dată. Pt a obține trb să ne concentrăm asupra ramificațiilor/instr. controlate. Privită ca nivelul minim de acoperire. Există porțiuni de cod care se execută doar în situații speciale (rutină eroare)⇒inspecție cod. **+** ușor de făcut **-** nu testează fiecare condiție la cele compuse, nu testează fiecare ramură, când lipsește else forțează execuția cu adev.
• **nivel decizie sau ramură**=fiecare ramură a grafului este parcursă măcar o dată, extinde metoda precedentă; gen date de teste pt fiecare valoare de adevăr a decizilor. **+** testează ramuri nule a if/else **-** nu testează individual condițiile. • **nivel condiție**= fiecare condiție individuală dintr-o decizie să ia atât valoarea A și F **+** acoperă condițiile individuale **-** poate să nu realizeze acop. nivel ramură • **nivel condiție/decizie**=gen. date aî fiecare condiție individuală dintr-o decizie să ia atât A și F (dacă posibil) și fiecare decizie să ia atât A și F. (unele condiții individuale "mascate" pot să nu fie testate) • **nivel condiții multiple**= gen. date aî să parcurgă toate combinațiile a condițiilor individuale. Pt n condiții poate genera 2^n teste. **•Modified condition/decision (MC/DC)** =fiecare condiție individuală dintr-o decizie ia A și F; fiecare decizie ia A și F; fiecare condiție individuală influențează în mod independent decizia din care face parte **+** acoperire mai puternică, testează influența condițiilor, produce teste mai puține, depinde liniar. 2 teste acoperă o condiție⇒în ele condiția ia ambele valori de adevăr. ex la și: t1 și t2 acoperă C2

Test	C_1	C_2	$C_1 \wedge C_2$
t_1	A	A	A
t_2	A	F	F
t_3	F	F	F

Test	C_1	C_2	$C_1 \vee C_2$
t_1	F	T	T
t_2	T	F	T
t_3	F	F	F

Test	C_1	C_2	$C_1 \times OR C_2$
t_1	A	A	F
t_2	A	F	A
t_3	F	F	F

Exemplu: $C = C1 \wedge C2 \vee C3$

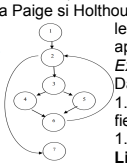
Test	C1	C2	C3	C	Efect demonstrat pentru
1	True	True	False	True	C1
2	False	True	False	False	
3	True	True	False	True	C2
4	True	False	False	False	
5	True	False	True	True	C3
6	True	False	False	False	

Set de teste minimal

Test	C1	C2	C3	C	
t1	True	True	False	True	t1 și t2 testeaza C1
t2	False	True	False	False	t1 și t3 testeaza C2
t3	True	False	False	False	t3 și t4 testeaza C3
t4	True	False	True	True	

• **nivel circuite independente**=identifica limita superioară pt. numărul de căi necesare pentru obținerea unei acoperiri la nivel de ramură. **+** setul de bază generat automat și folosit pt acoperire ramură. **-** set de bază unic, complexitate poate fi redusă. G graf complet, e arce n noduri, nr. circuite linear independente(=nici unul nu poate fi obținut ca o combinație a celorlalte) V(G)=e-n+1, V(G)=e-n+2p(nr. comp. conectate). V(G)=e-n+p(complet conectat).V(G)=e-n+2(program pt o singură subrutină p=1). Circuitele l.i.=set de bază.
• **nivel cale**=fiecare cale din graf este parcursă măcar o dată. **Problemă**: în majoritatea situațiilor există un număr infinit (foarte mare) de căi. **Soluție**: Împărțirea căilor în clase de echivalență. Se poate forma o expresie regulată (tehnica Paige si Holthouse) din noduri. **+** selectate căi pe care alte metode de testare structurală nu le ating **-** căi nefezabile, nu exersează condițiile individuale, tehnica nu este aplicabilă direct pe programe nestructurate

```
1  …
2  while c1 do begin
3  if c2 then
4                …
5            else …
6  end
7  …
```



Expresia regulată obținută este: 1.2.(3.(4+5).6.2)*.7
Dacă secv.* se repetă 1 sau 0 ori⇒1.2.(3.(4+5).6.2 + null).7⇒căile 1.2.7, 1.2.3.4.6.2.7, 1.2.3.5.6.2.7. Nr de căi se obține înlocuind în expresia regulată fiecare nod cu (inclusiv pt. null), iar operația de concatenare devine înmulțire: 1.1.(1.(1+1).1.1 + 1).1 = 3 căi
Linear Code Sequence and Jump Coverage=este o cale(execuție) a unei secv. de cod urmată de un salt al controlului=(X=*start secv liniară*,Y=*sf. secv, Z=linie salt*)

```
1.  begin
2.    int x,y,z;
3.    read(x, y);
4.    z = 1;
5.    while (y>0) {
6.      z=z*x;
7.      y=y-1;
8.    }
9.    write(z);
10. end
```

LCSAJ	Start	End	Jump to
1	1	8	5
2	5	8	5
3	5	5	9
4	1	5	9
5	9	9	Exit

T={t1=(x=3,y=0), t2=(x=3,y=2)} t1:(1,5,9)→(9,9,ex); t2:(1,8,5)→(5,8,5)→(5,5,9)→(9,9,ex) ⇒ T acoperă cele 5 LCSAJ.
Un set de teste care realizează o acoperire la nivel de decizie nu realizează în mod necesar o acoperire la nivel de LCSAJ

```
1.  begin
2.    int x,y,z;
3.    read(x, y);
4.    p=0;
5.    if(x<0)
6.      p=x;
7.    if(y<0)
8.      p=p+1;
9.    else
10.     p=p+2;
11. end
```

LCSAJ	Start	End	Jump to
1	1	9	exit
2	1	5	7
3	7	9	exit
4	7	7	10
5	1	7	10
6	10	10	Exit

T = {t1, t2}, unde t1 = (x = -1, y = -1), t2 = (x = 0, y = 0). Pentru t1 ambele condiții sunt satisfăcute, t1: (1, 9, exit). Pentru t2 ambele condiții sunt false, t2 : (1, 5, 7) → (7, 7, 10) → (10, 10, exit).Cele două LCSAJ rămase pot fi parcurse de t3 = (x = -1, y = 0) si t4 = (x = 0, y = -1),
t3: (1, 7, 10) → (10, 10, exit) t4: (1, 5, 7) → (7, 9, exit)

Mutation Testing=tehnică de ev. a unui set de teste pt program. Mutation=modificare f. mică corect dpdv sintactic a unui program.

```
P:                M1:               M2:
begin             begin             begin
  int x, y;         int x, y;         int x, y;
  read(x, y);       read(x, y);       read(x, y);
  if (x > 0)        if (x >= 0)       if (x > 0)
    write(x+y)        write(x+y)        write(x-y)
  else              else              else
    write(x*y)        write(x*y)        write(x*y)
end               end               end
```

Tehnica: - generare mutanți (cu operatori de mutatie), rularea setului de date pe P și M, dacă un test distinge P de m ⇒ P omoară M
Mutanți first-order/higher-order. n-order = mutanți obținuți din n modificări la P; first-oder mutant of a (n-1)order mutant, higher-order mutants când n>1
Sunt fol. doar mutanții de ord 1 ⇒ nr. mare de mutanți de ordin mare, coupling effect

Principiile de bază: **Competent Programmer Hypothesis**=programatorul va scrie un P care se află în vecinătatea unui P care rezolvă problema (mutanți ordin 1)
Coupling effect - datele de test care disting orice P care diferă cu puțin de P corect sunt suficient de puternice pt. a distinge erori mai complexe.
Strong/weak mutation - un t omoară M (distinge M de P) dacă cele 2 se comportă diferit pt t ⇒ (1) le aduce în stări diferite (2)se propagă până la sfârșitul programului. Doar 1 ⇒ weak, ambele ⇒ strong
Mutanți echivalenți. Un M al lui P sn echiv. dacă se comportă identic cu P pt orice date de intrare. Este o problemă nedecidabilă ~> halting problem. În practică prin analiza codului.**Utilitatea**: evaluare set de date existent (+construire), detectare erori în cod. Programul nou obținut trebuie să fie valid din punct de vedere sintactic

```
P                  Mutanți
begin
  int x, y;
  read(x, y);
  if (x > 0)       M1 if (x+1 > 0)
                   M2 if (x> 0+1)
    write(x+y)     M3 write(x+1+y)
                   M4 write(x+y+1)
                   M5 write(x-y)
  else
    write(x*y)     M6 write((x+1)*y)
                   M7 write(x*(y+1))
                   M8 write(x/y)
end
```

P	t1	t2	t3	t4	Mutant distins
P(t)	0	0	1	1	
M1(t)	0	**1**	NE	NE	Y
M2(t)	0	0	**0**	NE	Y
M3(t)	0	0	**2**	NE	Y
M4(t)	0	0	**2**	NE	Y
M5(t)	0	0	1	1	**N**
M6(t)	0	1	NE	NE	Y
M7(t)	0	0	1	**0**	Y
M8(t)	ND	NE	NE	NE	Y

Operatori de mutație:+⇒-|*⇒/|x⇒x+1|
t1=(0, 0) t2=(0, 1) t3=(1, 0) t4=(-1, -1)
Mutanți nedistinși = M5. Nu este echivalent pt că (1,1) distinge P și M5.
Mutantion Score: MS(T)=D/(L+D) D=nr. mut. destinși, L=nedestinși neechivalenți Pt. ca t să distingă între P și M:
Reachability: Instr. mutantă să fie excecutată la aplicarea lui t.
State infection: Instr. mutantă să afecteze starea progr.
State propagation: Schimbarea de stare să se propage în exterior.

Traditional mutation operators (method-level operators) – operatori aplicabili oricăruilimbaj procedural. Class mutation operator – operatori specifici paradigmei orientate pe obiect și sintaxei Java • Încapsulare• Moștenire• Polimorfism și dynamic binding• Suprascrierea metodelor• Java specific
Blackbox Testing= examine while not knowing the internal structure of program. Testers do not require technical knowledge. Difficult to automate. Requires prioritization, typically infeasible to test all user paths. Tests can be executed by crowdsourced or outsourced testers. Difficult to calculate test coverage. If a test fails, it can be difficult to understand the root cause of the issue. refers to the fact that we are rather testing the system based on what we know about **how it's supposed to respond** to our test cases.
Whitebox Testing= refers to is the fact that the tester is using detailed knowledge about the internals of the system in order to construct better test cases
Unit Testing=looking at some small software module at a time and testing it in an isolated fashion. find defects in the internal logic of the SUT as early as possible
Integration Testing=taking multiple software modules that have already been unit tested and testing them in combination with each other
System Testing=does the system as a whole meet its goals, often blackbox.

```
def correct_test():
    # Queue1 silently holds only 2 byte unsigned integers, than wraps around
    q = Queue1(2)
    succeeded = q.enqueue(100000)  # value greater than 2^16
    assert succeeded
    value = q.dequeue()
    assert value == 100000  # test1 failed

    # Queue2 silently fails to hold more than 15 elements
    q = Queue2(30)
    # try to enqueue more than 15 elements
    for i in range(20):
        succeeded = q.enqueue(i)
        assert succeeded  # test2 failed

    # Queue3 implements empty() by checking if dequeue() succeeds.
    # This changes the state of the queue unintentionally.
    q = Queue3(2)
    succeeded = q.enqueue(10)
    assert succeeded
    assert not q.empty()  # the function checks by trying to dequeue
    value = q.dequeue()
    assert value == 10  # test3 failed

    # Queue4 dequeue() of an empty queue returns False instead of None
    q = Queue4(2)
    value = q.dequeue()
    assert value is None  # test4 failed

    # Queue5 holds one less item than intended
    q = Queue5(2)
    for i in range(2):
        succeeded = q.enqueue(i)
        assert succeeded  # test4 failed
```

Fuzzer Charlie Miller

```
# List of files to use as initial seed
file_list = [
    "blackbox_testing.pdf",
    "code_coverage.pdf",
    "pluginuri_utile_java.pdf",
    "examen.pdf"
]

# List of applications to test
apps = [
    "/Applications/Adobe Acrobat Reader.app/Contents/MacOS/AdobeReader",
    "/Applications/Microsoft Word.app/Contents/MacOS/Microsoft Word"
]

fuzz_output = "fuzz.pdf"

FuzzFactor = 250
num_tests = 10000
```

```
for i in range(num_tests):
    file_choice = random.choice(file_list)
    app = random.choice(apps)

    buf = bytearray(open(file_choice, 'rb').read())

    # start Charlie Miller code
    numwrites = random.randrange(math.ceil((float(len(buf)) / FuzzFactor))) +
1

    for j in range(numwrites):
        rbyte = random.randrange(256)
        rn = random.randrange(len(buf))
        buf[rn] = "%c" % rbyte
    # end Charlie Miller code

    open(fuzz_output, 'wb').write(buf)

    process = subprocess.Popen([app, fuzz_output])

    time.sleep(1)
    crashed = process.poll()
    if not crashed:
        process.terminate()
```

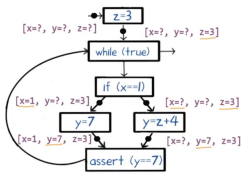
Test full coverage pt queue.

```
def test():
    q = Queue(2)
    assert q
    q.checkRep()

    empty = q.empty()
    assert empty
    q.checkRep()

    full = q.full()
    assert not full
    q.checkRep()

    result = q.dequeue()
    assert result == None
    q.checkRep()

    result = q.enqueue(10)
    assert result == True
    q.checkRep()

    result = q.enqueue(20)
    assert result == True
    q.checkRep()

    empty = q.empty()
    assert not empty
    q.checkRep()

    full = q.full()
    assert full
    q.checkRep()

    result = q.enqueue(30)
    assert result == False
    q.checkRep()

    result = q.dequeue()
    assert result == 10
    q.checkRep()

    result = q.dequeue()
    assert result == 20
    q.checkRep()
```

Test Coverage=automatic way of partitioning the input domain with some observed features of the source code
Function Coverage=all functions are executed. Test coverage este o masura a proportion in care se executa programul, dacă e 100 nu înseamnă ca am găsit toate bug-urile.
Code Coverage = cat % din de cod se executa, relativ la input Line coverage=cat % din linii, relativ la cum e scris codul(dacă avem condiții pe mai multe linii e posibil să nu ia în considerare liniile)
Branch(Decision) Coverage is a metric where a branch in a code is covered if it executes both ways, probleme dacă nu există else| **Loop coverage** specifies that we execute each loop 0 times, once, and more than once|**MC/DC coverage** starts off with a branch coverage, it additionally states that every condition involved in a decision takes on every possible outcome|**Path coverage**. A path through a program is a sequence of decisions made by operators in the program. achieving path coverage is going to be impossible for all real code.|**Boundary value coverage**. test cases are designed to cover the minimum, maximum, and boundary values within each input range|**Synchronization coverage** is a metric used in software testing to measure the effectiveness of testing the synchronization mechanisms within concurrent or multi-threaded software systems. It focuses on ensuring that all relevant synchronization points and scenarios are adequately tested. | Cod care nu poate fi urmărit. Cod imposibil (cod care se executa pt un input prost de ex). Cod care nu merită (e greu de triggeruit), suita de teste nu e adecvată. We strongly believe that if we have a good test suite, and we measure its coverage, the coverage will be good. We do not believe, on the other hand, that if we have a test suite which gets good coverage, it must be a good test suite.Used in the right way, coverage can be a relatively low cost way to improve the testing that we do for a piece of software. Used incorrectly, it can waste our time, and perhaps worst, lead to a false sense of security
Random Testing in Practice=construct test cases to test things that we don't actually understand or know very well. some of our random testers become useless as the software development timeline advances. Our focus should be on external interfaces provided, things like file I/O and the graphical user interface, and so we're fuzzing exactly these sorts of things. **Tuning rules and probabilities**: start simple, examine test cases, look at coverage results, adjust rules, tweak probabilities, add functionality repeat. **Filesystem testing**. If we start with a simple file system tester, we'll make a list of all the API calls that we'd like to test then call them randomly with random arguments.After observing that this doesn't work very well we consider special test cases in order to remove limitations of our random tester and, over time, this process ends up with a random tester that will be extremely strong.**we need to think about what we're testing, how the code is structured, and how we're going to execute all the way through it, and this is a fundamental limitation of random testing**. Fuzzing implicit inputs techniques - perturbing the scheduler (generate load, network activity, thread stress testing), inserting delays near synchronization+access to shared vars, "unfriendly emulators". **+** less tester bias, weaker hypotheses about where bugs are, once testing is automated human cost is almost 0, often surprises, every fuzzer finds different bugs, **-** input validity can be hard, oracles are hard, no stopping criteria, may find unimportant bugs/spend time on, can find the same one multiple times, can be hard to debug, every fuzzer finds different bugs
The purpose of testing is to maximize the number of bugs found per amount of effort spent testing. One issue that can come up if we write a really good random test-case generator is that we can be overwhelmed by the success of our own bug-finding effort. If a lot of bugs: Pick a bug and report it to the developers. As soon as we get a new version of the system that fixes a bug we've reported, we can just do another one.***Bug Triage*** is the process by which the severity of different bugs is determined, and we start to disambiguate between different bugs in order to basically try to get a handle on which bugs we can report in parallel. **-** core dump or stack trace, examine bug-triggering test cases→ test case reduction, search over version history git_bisect.***Delta Debugging*** The framework takes our script and the test input and automates this process in a loop. The loop terminates when the delta debugger can't reduce the input anymore, using built-in heuristics.
Reporting Bugs dont report duplicates, respect conventions, include small stand-alone test case, only report valid test cases, expected+actual output, instructions for reproduction (platform/system, version of SUT)
Test Suite is a collection of tests that can often be run automatically, run periodically (ex on every commit), goal is to show that SUT has desired properties. Add small feature-specific tests, large realistic inputs, regression tests(any input that caused some previous vs to fail), usually not a random tester.
When appropriate, all 3 kinds of input should be used as a basis for testing: APIs that are provided by the SUT, can be tested directly, APIs used by the SUT can be tested using fault injection, Non-functional inputs, Failed coverage items do not provide a mandate to cover the failed items, but rather give clues to ways in which the tests are inadequate.

Program analysis =body of work to automatically discover useful facts about programs.⇒***Dynamic(run-time) analysis*** → infers facts of program by monitoring its runs tools: array bound checking Purify, memory leak detection Valgrind, data race detection Eraser, Finding likely invariants Daikon, ⇒***Static analysis*** →infers facts of program by inspecting its code, tools for Suspicious error patterns, checking API usage rules, memory leak detection, verifying invariants. Since **dynamic analysis** discovers information by running the program a finite number of times, it cannot in general discover information that requires observing an unbounded number of paths. ***Program invariants*** refer to properties or conditions that r**emain true throughout the execution of a program**, serve as useful guidelines for understanding and reasoning about the correctness and behavior of a program, prove the correctness of algorithms, detect bugs, and aid in program verification
Static analysis creates Control Flow Graph representation, node=statement. **Abstract state**: tracks the constant values of the 2 variables in the above program. **Concrete state**: tracks the actual values in a particular run. Static analysis operates on abstract states, each of which summarizes a set of concrete states. **static analysis sacrifices completeness,** may fail to accurately represent
the value of a variable in an abstract state, may miss variables with constant value

```
void main() {
  z = 3;
  while (true) {
    if (x == 1)
      y = 7;
    else
      y = z + 4;
    assert (y == 7);
  }
}
```



Static analysis discovers that the variable y has the constant value 7 at the exit of this
program. We will use a common static analysis method called **iterative approximation.**
Static analysis discovers that the variable y has the constant value 7 at the exit of this
program. We will use a common static analysis method called iterative approximation.
At this point, the analysis has concluded that at each immediate predecessor of the assertion, the value of y is 7. It thereby concludes that the value of y in the assertion
must be 7, and therefore that this assertion is valid
the analysis might need to visit the same program point multiple times.

Dynamic vs static analysis

	Dynamic	Static
Cost	Proportional to program's execution time	Proportional to program's size
Effectiveness	Unsound (may miss errors)	Incomplete (may report spurious errors)

We say that a **dynamic analysis is unsound**. In other words, **it may produce false negatives**
We say that **static analysis is incomplete**. In other words it **may produce false positives**.

Undecidability of program properties Designing a program analysis is an art that involves striking a suitable tradeoff between domination soundness and completeness. This tradeoff is typically dictated by the consumer of the program analysis. **Who needs program analysis?** ***compilers*** ⇒ bridge between high-level languages and architectures. Use program analysis to generate efficient code. ***Software quality tools*** ⇒Finding programming errors, Proving program invariants, Generating test cases, Localizing causes of errors, ***IDEs*** ⇒Understand programs, Refactor programs, Restructuring a program without changing its external behavior.