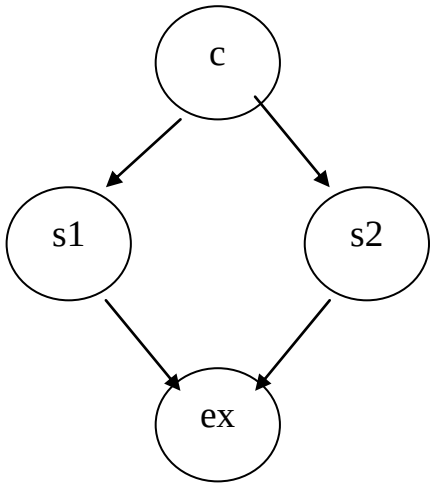
Testare structurala

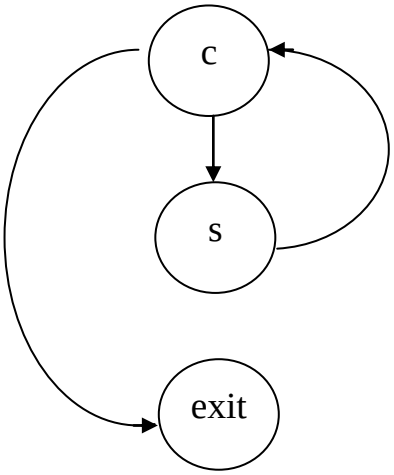
- datele de test sunt generate pe baza implementarii (programului), fara a lua in considerare specificatia (cerintele) programului
- pentru a utiliza metode structurale de testare programul poate fi reprezentat sub forma unui graf orientat
- datele de test sunt alese astfel incat sa parcurga toate elementele (instructiune, ramura sau cale) grafului macar o singura data. In functie de tipul de elemente ales, vor fi definite diferite masuri de acoperire a grafului: acoperire la nivel de instructiune, acoperire la nivel de ramura sau acoperire la nivel de cale

Transformarea programului intr-un graf orientat

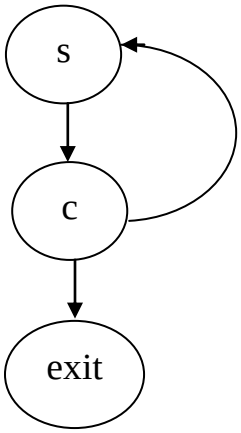
- Pentru o secventa de instructiuni se introduce un nod
- if c then s1 else s2



- while c do s

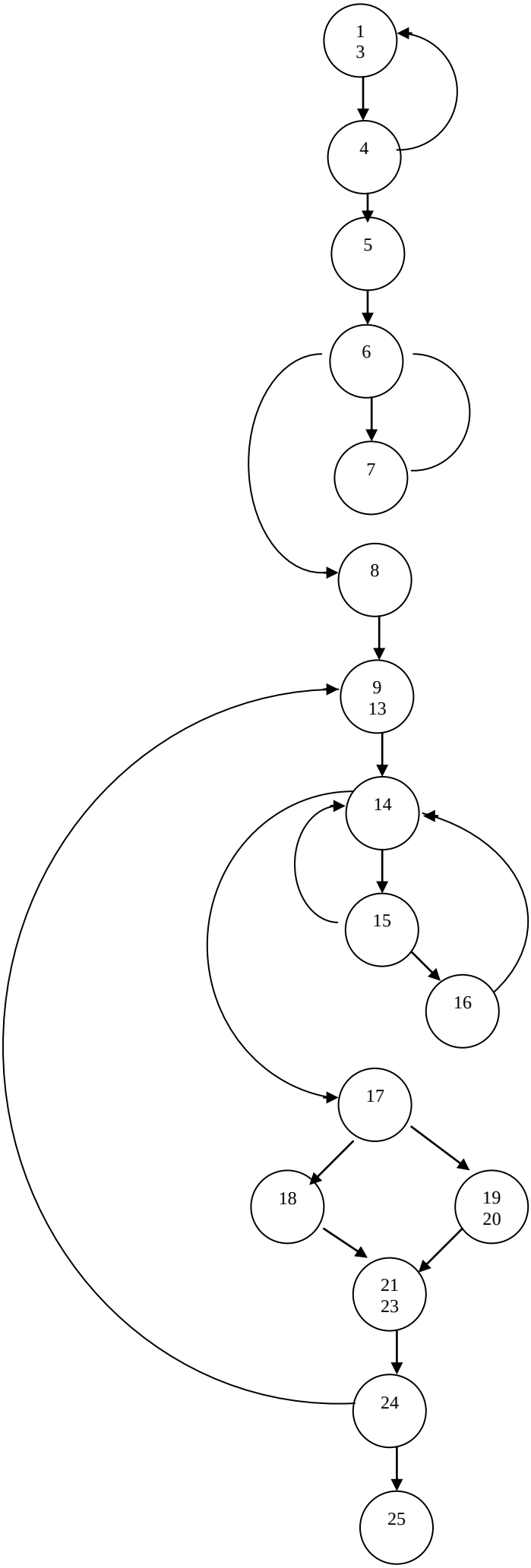


- repeat s until c



```
public class Test {
   public static void main(String[] arg) {

        KeyboardInput in = new KeyboardInput();
        char response,c, nl;
        boolean found;
        int n,i;
        char[]a=new char[20];
1               do {
2               System.out.println("Input an integer between 1 and 20: ");
3               n = in.readInteger();
4       } while (n<1||n>20);
5               System.out.println("input "+n+" character(s)");
6       for (i=0; i<n; i++)
7               a[i] = in.readCharacter();
8         nl = in.readCharacter();
9               do {
10              System.out.println("Input character to search for: ");
11              c = in.readCharacter();
12              nl = in.readCharacter();
13              found=false;
14              for(i=0; !found && i<n; i++)
15                      if(a[i]==c)
16                              found=true;
17              if(found)
18                              System.out.println("character "+c+" appears at
position "+i);
19              else
20                      System.out.println("character "+c+" does not appear
in string");
21                      System.out.println("Search for another
character?[y/n]: ");
22              response=in.readCharacter();
23              nl = in.readCharacter();
24     } while ((response=='y') ||(response=='Y'));
25   }
}
```


1
3
4
5
6
7
8
9
13
14
15
16
17
18
19
20
21
23
24
25

Pe baza grafului se pot defini diverse acoperiri:

- *Acoperire la nivel de instructiune*: fiecare instructiune (nod al grafului) este parcursa macar o data
- *Acoperire la nivel de ramura*: fiecare ramura a grafului este parcursa macar o data
- *Acoperire la nivel de cale*: fiecare cale din graf este parcursa macar o data

1. Statement coverage (acoperire la nivel de instructiune)

Pentru a obtine o acoperire la nivel de instructiune, trebuie sa ne concentram asupra acelor instructiuni care sunt controlate de conditii (acestea corespund ramificatiilor din graf)

Intrari				Rezultat afisat	Instructiuni parcurse
N	x	C	s		
1	a	a	y		1..3, 4, 5, 6 7, 6, 8, 9..13 14, 15, 16, 14, 17, 18, 21..23 24, 9..13
		b	n		14, 15, 14, 17, 19..20, 21..23 24, 25

Testarea la nivel de instructiune este privita de obicei ca nivelul minim de acoperire pe care il poate atinge testarea structurala. Acest lucru se datoreaza sentimentului ca este absurd sa dai in functionare un software fara a executa macar odata fiecare instructiune.

Totusi, destul de frecvent, acesta acoperire nu poate fi obtinuta, din urmatoarele motive:

- Existenta unei portiuni izolate de cod, care nu poate fi niciodata atinsa. Aceasta situatie indica o eroare de design si respectiva portiune de cod trebuie inlaturata.
- Existenta unor portiuni de cod sau subrutine care nu se pot executa decat in situatii speciale (subrutine de eroare, a caror executie poate fi dificila sau chiar periculoasa). In astfel de situatii, acoperirea acestor instructiuni poate fi inlocuita de o inspectie riguroasa a codului

Avantaje:

- Realizeaza executia macar o singura data a fiecarei instructiuni
- In general usor de realizat

Slabiciuni:

Nu asigura o acoperire suficienta, mai ales in ceea ce priveste conditiile:

- Nu testeaza fiecare conditie in parte in cazul conditiilor compuse (de exemplu, pentru a se atinge o acoperire la nivel de instructiune in programul folosit ca exemplu, nu este necesara introducerea unei valori mai mici ca 1 pentru *n*)
- Nu testeaza fiecare ramura
- Probleme suplimentare apar in cazul instructiunilor *if* a caror clauza *else* lipseste. In acest caz, testarea la nivel de instructiune va forta executiei ramurii corespunzatoare valorii adevarat, dar, deoarece nu exista clauza *else*, nu va fi necesara si executia celeilate ramuri. Metoda poate fi extinsa pentru a rezolva aceasta problema.

2. Decision coverage) (acoperire la nivel de decizie) sau branch coverage (acoperire la nivel de ramura)

- Este o extindere naturala a metodei precedente.
- Genereaza date de test care testeaza cazurile cand fiecare decizie este adevarta sau falsa.

Intrari				**Rezultat afisat**	**Instructiuni parcurse**
N	**x**	**C**	**s**		
25				Cere introducerea unui intreg intre 1 si 20	
1	a	A	y	Afiseaza pozitia 1; se cere introducerea unui nou caracter	
		B	n	Caracterul nu apare	

Avantaje:

- Este privita ca etapa superioara a testarii la nivel de instructiune; testeaza toate ramurile (inclusiv ramurile nule ale instructiunilor *if/else*)

Dezavantaje:

- Nu testeaza conditiile individuale ale fiecarei decizii

3. Condition coverage (acoperire la nivel de conditie)

- Genereaza date de test astfel incat fiecare conditie individuala dintr-o decizie sa ia atat valoarea adevarat cat si valoarea fals (daca acest lucru este posibil).

- De exemplu, daca o decizie ia forma c1 || c2 sau c1 && c2, atunci acoperirea la nivel de conditie se obtine astfel incat fiecare dintre conditiile individuale c1 si c2 sa ia atat valoarea adevarat cat si valoarea fals

Nota: decizie inseamna orice ramificare in graf, chiar atunci cand ea nu apare explicit in program. De exemplu, pentru constructia for i := 1 to n din Pascal conditia implicita este i <= n

Decizii	Conditii individuale
while (n<1||n>20)	n < 1, n > 20
for (i=0; i<n; i++)	i < n
for(i=0; !found && i<n; i++)	found, i< n
if(a[i]==c)	a[i] = c
if(found)	Found
while ((response=='y') ||(response=='Y'))	(response=='y') (response=='Y')

Intrari				**Rezultat afisat**	**Instructiuni parcurse**
n	**x**	**C**	**s**		
0				Cere introducerea unui intreg intre 1 si 20	
25				Cere introducerea unui intreg intre 1 si 20	
1	a	A	y	Afiseaza pozitia 1; se cere introducerea unui nou caracter	
		B	Y	Caracterul nu apare; se	

				cere introducerea unui nou caracter	

Avantaje

- Se concentreaza asupra conditiilor individuale

Dezavantaj

- Poate sa nu realizeze o acoperire la nivel de ramura. De exemplu, datele de mai sus nu realizeaza iesirea din bucla *while* ((response=='y') ||(response=='Y')) (conditia globala este in ambele cazuri adevarata). Pentru a rezolva aceasta slabiciune se poate folosi testarea la nivel de decizie / conditie.

4. Condition/decision coverage (acoperire la nivel de conditie/decizie)

- Genereaza date de test astfel incat fiecare conditie individuala dintr-o decizie sa ia atat valoarea adevarat cat si valoarea fals (daca acest lucru este posibil) si fiecare decizie sa ia atat valoarea adevarat cat si valoarea fals.

Intrari				**Rezultat afisat**	**Instructiuni parcurse**
n	**x**	**C**	**s**		
0				Cere introducerea unui intreg intre 1 si 20	
25				Cere introducerea unui intreg intre 1 si 20	
1	a	A	y	Afiseaza pozitia 1; se cere introducerea unui nou caracter	
		B	Y	Caracterul nu apare; se cere introducerea unui nou caracter	
		B	n	Caracterul nu apare	

5. Multiple condition coverage (acoperire la nivel de conditii multiple)

Genereaza date de test astfel sa fie incat sa parcurga toate combinatiile posibile de adevarat si fals ale conditiilor individuale.

6. Modified condition/decision (MC/DC) coverage

- Condition/Decision coverage poate sa nu testeze unele conditii individuale (care sunt “mascate” de alte conditii)
- Multiple condition coverage poate genera o explozie combinatorica (pentru n conditii pot fi necesare 2^n teste)

Solutie: O forma modificata a condition/decision coverage.
Un set de teste satisface MC/DC coverage atunci cand:

- Fiecare conditie individuala dintr-o decizie ia atat valoare True cat si valoare False
- Fiecare decizie ia atat valoare True cat si valoare False
- Fiecare conditie individuala influenteaza in mod independent decizia din care face parte

Avantaje:

- Acoperire mai puternica decat acoperirea conditie/decizie simpla, testand si influenta conditiilor individuale aspra deciziilor
- Produce teste mai putine – depinde liniar de numarul de conditii

AND

Test	**C1**	**C2**	**C1∧C2**
t1	True	True	True
t2	True	False	False
t3	False	True	False

t1 si t3 acopera C1
t1 si t2 acopera C2

OR

Test	**C1**	**C2**	**C1∨C2**
t1	False	True	True
t2	True	False	True
t3	False	False	False

t2 si t3 acopera C1
t1 si t3 acopera C2

XOR

Test	**C1**	**C2**	**C1xorC2**
t1	True	True	False
t2	True	False	True
t3	False	False	False

t2 si t3 acopera C1
t1 si t2 acopera C2

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
t2	False	True	False	False
t3	True	False	False	False
t4	True	False	True	True

t1 si t2 testeaza C1
t1 si t3 testeaza C2
t3 si t4 testeaza C3

Testarea circuitelor independente

Acesta este o modalitate de a identifica limita superioara pentru numarul de cai necesare pentru obtinerea unei acoperiri la nivel de ramura.

Se bazeaza pe formula lui McCabe pentru Complexitate Ciclomatica:

Dat fiind un graf complet conectat *G* cu *e* arce si *n* noduri, atunci numarul de circute linear independente este dat de:

$$V(G) = e - n + 1$$

Terminologie:

- Graf complet conectat: exista o cale intre oricare 2 noduri (exista un arc intre nodul de stop si cel de start)
- Circuit = cale care incepe si se termina in acelasi nod
- Circuite linear independente: nici unul nu poate fi obtinut ca o combinatie a celorlalte

In exemplul nostru, adaugand un arc de la 25 la 1, avem:
n = 16, e = 22, V(G) = 7

Circuite independente:

a) 1..3, 4, 5, 6, 8, 9..13, 14, 17, 18, 21..23, 24, 25, 1..3
b) 1..3, 4, 5, 6, 8, 9..13, 14, 17, 19..20, 21..23, 24, 25, 1..3
c) 1..3, 4, 1..3
d) 6, 7, 6
e) 14, 15, 14
f) 14, 15, 16, 14
g) 9..13, 14, 17, 18, 21..23, 24, 25, 1

Acesta poate numele de set de baza

Orice cale se poate forma ca o combinatie din acest set de baza.
De exemplu

1..3, 4, 1..3, 4, 1..3, 4, 5, 6, 7, 6, 8, 9..13, 14, 15, 16, 14 17, 18, 21..23, 24, 25, 1..3

este o combinatie din: a, c (de 2 ori), d, f

Avantaje:
Setul de baza poate fi generat automat si poate fi folosit pentru a realiza o acoperire la nivel de ramura

Dezavantaje:
Setul de baza nu este unic, iar uneori complexitatea acestuia poate fi redusa

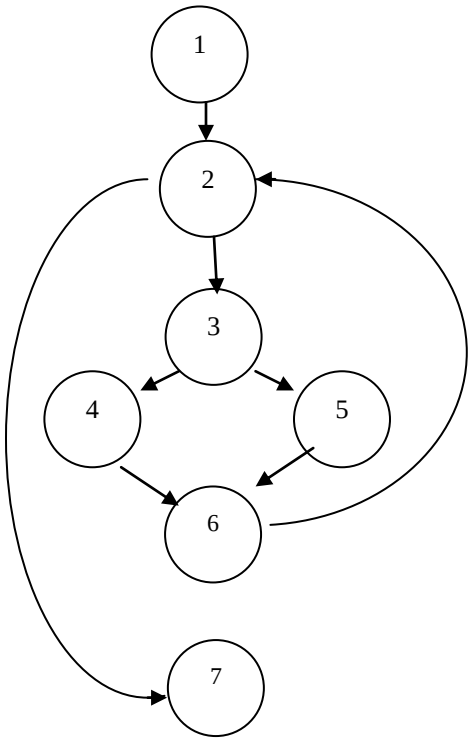
Testare la nivel de cale

- Genereaza date pentru executarea fiecarei cai macar o singura data
- Problema: in majoritatea situatiilor exista un numar infinit (foarte mare) de cai
- Solutie: Impartirea cailor in clase de echivalenta.

De exemplu: 2 clase pot fi considerate echivalente daca difera doar prin numarul de ori de care sunt traversate de acelasi circuit;
determina 2 clase de echivalenta: traversate de 0 ori si n ori, n > 1

Aplicatie:
Daca un program este structurat, atunci, folosindu-se o tehnica descrisa de Paige si Holthouse (1977), acesta poate fi caracterizat de o expresie regulata formata din nodurile grafului

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



Expresia regulata obtinuta este: 1.2.(3.(4+5).6.2)*.7

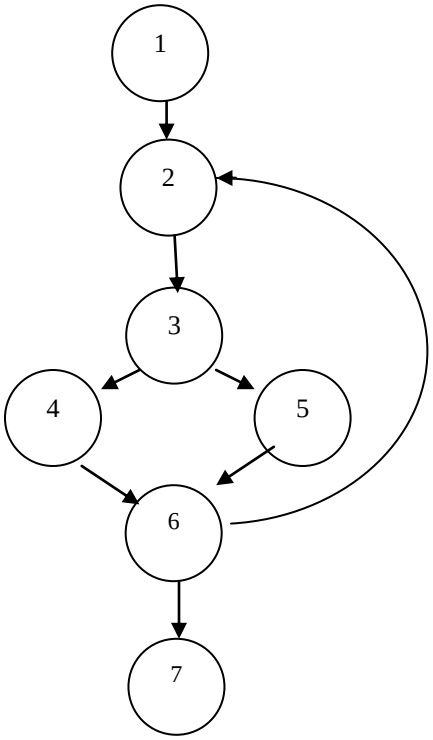
se ia n = 0 si n = 1

1.2.(3.(4+5).6.2 + null).7

1.2.7
1.2.3.4.6.2.7
1.2.3.5.6.2.7

numar de cai:
1.1.(1.(1+1).1.1 + 1).1 = 3

```
1  repeat
2              if c2 then
3
4              else ….
5  until c1
6
```



Expresia regulata: 1.2.3.(4+5).6.(2.3.(4+5).6)*.7

se ia n = 0 si n = 1

1.2.3.(4+5).6.(2.3.(4+5).6 + null).7

1.2.3.4.6.7
1.2.3.5.6.7
1.2.3.4.6.2.3.4.6.7
1.2.3.4.6.2.3.5.6.7
1.2.3.5.6.2.3.4.6.7
1.2.3.5.6.2.3.5.6.7

6 cai

Pentru exemplul nostru:

1.4.(1.4)*.5.6.(7.6)*.8.9.14.(15.(null+16).14)*17.(18+19).21.24.(
9.14.(15.(null+16).14)*17.(18+19).21.24)*.25

numar de cai

2.2.3.2.(3.2+1) = 168 cai

Observatie: multe cai, din care o mare parte sunt nefezabile (de exemplu, iesirea de la inceput din cele doua instructiuni *for*)

Avantaje:

- Sunt selectate cai pe care alte metode de testare functionala (inclusiv testare la nivel de ramura) nu le ating

Dezavantaje:

- Multe cai, din care o parte pot fi nefezabile
- Nu exerseaza conditiile individuale ale deciziilor
- Tehnica descrisa pentru generarea cailor nu este aplicabila direct programelor nestructurate