Testiranje petlji i druge tehnike zasnovane na putanjama

Kako sistematski biramo testove za petlju?

Pored grana, moramo razmišljati o tome koliko puta se telo izvršava.

1. Jednostruka petlja

Izabrati testove za 0, 1, tipičan i granične brojeve iteracija.

2. Granice i uslovi

Povezati broj iteracija sa preduslovom, gornjom granicom i rezultatom.

3. Više petlji

Razlikovati ugneždene, nadovezane i nestrukturirane petlje.

4. Smanjiti broj testova

Testirati ciljano, bez nekritičnog množenja svih slučajeva.

Ista grana — različit broj iteracija

0 iteracija

Preduslov je odmah netačan; telo se ne izvršava.

1 iteracija

Proverava prvi prolaz i prelaz ka izlasku.

Veći broj iteracija

Može otkriti greške brojača, ažuriranja stanja ili završetka.

Tipične greške: < umesto <=; pogrešan početak brojača; izostavljeno ažuriranje; pogrešna akumulacija; terminacija tek posle pogrešnog prolaza.

Pitanje nije samo „da li je petlja izvršena?“, već „da li je izvršena nula, jedan, tipičan i graničan broj puta?“.

Pokrivenost grana ne iscrpljuje testiranje petlje

Dve test-situacije mogu pokriti ulaznu i izlaznu granu, a da telo prođe samo jednom.

```
int sumTo(int limit) {
    int sum = 0;
    int i = 1;
    while (i <= limit) {
        sum += i;
        i++;
    }
    return sum;
}
```

limit	Broj iteracija	Očekivan rezultat
0	0	0
1	1	1
4	4	10
5	5	15

Za pouzdano testiranje proveravamo i broj prolaza i rezultat (oracle/aserciju), posebno pre i posle granice.

Klasičan izbor broja iteracija

Prema kriterijumu iz izvornog predavanja (naveden kao Beizer): n označava maksimum mogućih iteracija.

Broj prolaza	Test koji treba pokušati	Šta može da otkrije
0	Preskočiti telo ako je to moguće.	Preduslov odmah netačan.
1	Izvršiti telo jednom.	Prvi prolaz i izlazak.
m, gde je 1 < m < n	Izabrati tipičan broj iteracija.	Akumulacija i stanje kroz više prolaza.
n − 1	Neposredno ispod gornje granice.	Greška u blizini maksimuma.
n	Tačno na gornjoj granici.	Tačna obrada maksimalnog opsega.
n + 1	Neposredno iznad granice, ako se može zadati/izvesti.	Odbijanje nevažećeg ulaza ili provera prelivanja granice.

Ako nema definisanog maksimuma n, koristimo domenske granice i rizike; n + 1 ne znači da treba nasilno produžiti petlju.

VAŽNA NIJANSA

„n + 1 iteracija“ nije uvek validan ulaz

Ako specifikacija ograničava ulaz, n + 1 može da se odbije pre ulaska u petlju.

Ako je n granica petlje

Proveri n − 1, n i ponašanje za n + 1; proveri da li se petlja završava očekivano.

Ako je n ograničenje ulaza

Za n + 1 očekuj jasno definisano odbacivanje ili grešku — ne nužno n + 1 prolaza.

Za for petlju koja ide od i = 1 do i <= N, pri N = 20, ulaz 21 može biti nevažeći po specifikaciji. Testiramo validaciju ulaza odvojeno od ponašanja petlje.

Uvek zapiši očekivano ponašanje na granici: rezultat, izuzetak, odbijanje ulaza ili druga specifikovana reakcija.

Faktorijel: petlja sa jasnom granicom

Definisani opseg: 0 ≤ n ≤ 20; za n izvan opsega metod prijavljuje nevažeći argument.

```
static long factorial(int n) {
    if (n < 0 || n > 20) {
        throw new IllegalArgumentException("n mora biti 0..20");
    }
    long f = 1;
    for (int i = 1; i <= n; i++) {
        f *= i;
    }
    return f;
}
```

0! = 1

Telo petlje se ne izvršava.

1! = 1

Telo se izvršava jednom.

20!

Maksimum validnog ulaza; staje u long.

Iz testova se vidi broj iteracija i oracle

Testovi proveravaju tipične slučajeve, granicu petlje i validaciju ulaza.

```
@Test void factorialBoundaries() {
    assertEquals(1L, factorial(0));   // 0 iteracija
    assertEquals(1L, factorial(1));   // 1 iteracija
    assertEquals(3628800L, factorial(10));
    assertEquals(121645100408832000L, factorial(19));
    assertEquals(2432902008176640000L, factorial(20));
    assertThrows(IllegalArgumentException.class,
                 () -> factorial(-1));
    assertThrows(IllegalArgumentException.class,
                 () -> factorial(21));
}
```

Šta pokrivamo?

0, 1, tipičan, N−1, N prolaza.

Šta radi 21?

Proverava da li se maksimum validnog ulaza zaista poštuje.

Oracle za nevažeći ulaz

Izuzetak se proverava i za n < 0 i za n > 20.

JUnit 5: assertEquals proverava rezultat, a assertThrows očekivani izuzetak.

Počinjemo od najugneždenije petlje

Potpuno kombinovanje svih brojeva iteracija brzo postaje preskupo.

```
for (int row = 0; row < rows; row++) {
    for (int col = 0; col < cols; col++) {
        matrix[row][col] = row + col;
    }
}
```

Korak	Strategija	Primer za rows/cols
1	Testirati najugneždeniju petlju.	rows = 1; cols = 0, 1, tipično, max−1, max, max+1*
2	Držati spoljne petlje na minimumu.	Mala vrednost rows, da nema mnogo kombinacija.
3	Pomerati se ka spolja.	Testirati rows granice, uz cols tipičan/minimalan.

Osnovna ideja: najpre iscrpno testirati najugneždeniju petlju uz minimalan broj prolaza spoljašnjih petlji; zatim preći na sledeću petlju.

Posebno proveriti kombinacije u kojima jedna dimenzija ima 0 ili 1 element.

Zašto ne testiramo svaku kombinaciju?

Ako za svaku od k petlji probamo 6 iteracija, broj kombinacija može dostići 6^k.

Broj petlji k	Naivnih kombinacija 6^k
1	6
2	36
3	216
4	1 296
5	7 776

Broj kombinacija raste eksponencijalno.

Smanjenje prostora

Držati okolne petlje na minimalnoj ili tipičnoj vrednosti dok se testira izabrana petlja.

Testirati interakcije

Dodati ciljane kombinacije kada jedna petlja menja podatke ili granicu druge.

Birati prema riziku

Posebno obuhvatiti nule, jedinice, maksimalne dimenzije i izlazne rezultate.

Nezavisne petlje mogu se testirati odvojeno

Ako jedna petlja menja stanje koje druga koristi, one nisu potpuno nezavisne.

```
int count = 0;
for (int x : customers) {
    count++;
}

long total = 0;
for (int amount : payments) {
    total += amount;
}
```

Nezavisne

Različiti skupovi podataka i nezavisni rezultati: može se primeniti strategija za jednostruku petlju na svaku.

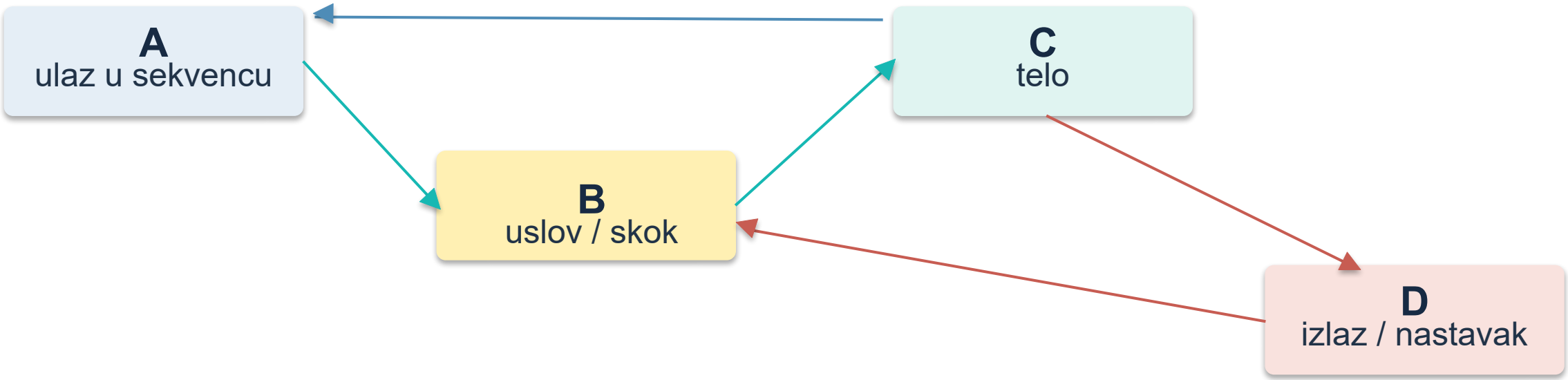
Zavisne

Ako druga petlja čita stanje ili podatke koje je prva promenila, testirati i njihovu međuzavisnost.

Praktično pravilo: petlje koje slede jedna za drugom tretiraju se kao jednostruke ako su nezavisne; u suprotnom pristup sličan ugnježdenim petljama.

Kada kontrolni tok „seče“ petlje

Ovakav oblik je indikator lošeg stila; preporuka je restrukturiranje, a ne beskonačno dodavanje testova.



Problem: granice petlji i izlazi nisu jasni; putanje se ukrštaju, pa je teško odrediti jednostavan skup slučajeva.

Preporuka

Preoblikovati u strukturirane while/for petlje.

Java napomena

Java nema proizvoljnu goto naredbu; dijagram prikazuje legacy tok.

LCSAJ i tehnike testiranja putanja

Šta dodaje LCSAJ u odnosu na pokrivanje grana?

Cilj je sistematski pronaći linearne sekvence koje se završavaju prenosom kontrole i odabrati testove koji ih izvode.

1. Definisati LCSAJ

Predstaviti sekvencu trojkom: početna linija, završna linija, cilj skoka.

2. Izvesti sekvence

Krenuti od početka programa i ciljeva netrivijalnih skokova; završiti na granici sekvence.

3. Konstruisati testove

Veza između uslova u kodu, ulaznih podataka i pokrivenih sekvenci.

4. Proceniti izvodljivost

Razlikovati sekvencu koja nije pokrivena od sekvence koju nijedan ulaz ne može izvesti.

LCSAJ je trojka, a ne samo jedna grana

Naziv potiče od Linear Code Sequence And Jump - linearna sekvenca koda i skok na drugu lokaciju.

START LINE

Početak programa ili linija na koju je kontrola preneta skokom.

+

FINISH LINE

Poslednja naredba linearne sekvence, neposredno pre prenosa kontrole.

+

JUMP TO LINE

Odredište na koje se prelazi skokom, umesto običnim sekvencijalnim izvršavanjem.

LCSAJ = (start, finish, jump-to)

Srodna ideja: dd-path testing posmatra putanju između kontrolnih odluka bez uključivanja samih uslovnih čvorova.

Program za klasifikaciju prostih i složenih brojeva

```
1  Integer num = sc.hasNextInt() ? sc.nextInt() : null;
2  while (num != null) {
3      boolean prime = true;
4      for (int factor = 2; factor <= num / 2; factor++) {
5          if (num % factor == 0) {
6              System.out.println(factor + " divides " + num);
7              prime = false;
8          }
9      }
10     if (prime) {
11         System.out.println(num + " is prime");
12     }
13     num = sc.hasNextInt() ? sc.nextInt() : null;
14 }
15 System.out.println("End of program");
```

Struktura

Ulazna petlja, unutrašnja FOR petlja i dve odluke.

Testni ulazi

Izvorni skup: prost broj 5, složen broj 6 i poseban slučaj 2.

Pretpostavka

Radi jednostavnosti, kao u izvoru, ne proverava se unos; razmatramo brojeve >= 2.

Prvo popišemo grane i mesta prenosa kontrole

LCSAJ se izvodi iz mogućih nastavaka toka kontrole, a ne proizvoljnim kombinovanjem brojeva redova.

```
1  Integer num = sc.hasNextInt() ? sc.nextInt() : null;
2  while (num != null) {
3      boolean prime = true;
4      for (int factor = 2; factor <= num / 2; factor++) {
5          if (num % factor == 0) {
6              System.out.println(factor + " divides " + num);
7              prime = false;
8          }
9      }
10      if (prime) {
11          System.out.println(num + " is prime");
12     }
13     num = sc.hasNextInt() ? sc.nextInt() : null;
14 }
15 System.out.println("End of program");
```

Grana ili skok	Značenje u primeru
2 -> 3 / 2 -> 15	Petlja se nastavlja / EOF vodi ka izlazu
4 -> 5 / 4 -> 10	FOR ulazi u telo / nema više iteracija
5 -> 6 / 5 -> 9	Broj je deljiv faktorom / nije deljiv
9 -> 5 / 9 -> 10	Sledeća iteracija FOR / kraj FOR petlje
10 -> 11 / 10 -> 13	Prime je TRUE / Prime je FALSE
14 -> 2	Povratak na uslov ulazne petlje
15 -> exit	Završetak programa

Kako sistematski nabrajamo LCSAJ sekvence?

Počinjemo od poznatih ulaznih tačaka i pratimo linearnu sekvencu sve do narednog prenosa kontrole.

1 **Odredi početne tačke** — Početak programa i odredišta skokova do kojih se može doći.

2 **Prati sekvencijalni tok** — Nastavljaj kroz naredbe sve dok se ne dođe do granice sekvence.

3 **Zabeleži trojku** — Upiši (početak, kraj, odredište skoka).

4 **Proveri uslove** — Utvrdi da li postoji ulaz koji zadovoljava sve odluke duž sekvence.

PRIMER TROJKE

`(1, 5, 9)`

Sekvenca počinje na 1, prolazi do odluke 5, a uslov koji nije ispunjen prenosi kontrolu na 9.

`(1, 10, 13)`

Sekvenca stiže do odluke 10, a grana Prime = FALSE vodi na naredno čitanje broja.

Iste početne tačke daju više LCSAJ sekvenci

Posmatramo početak programa, a zatim druge ulazne tačke - naročito 2, 5, 9, 10, 13 i 15.

```
1  Integer num = sc.hasNextInt() ? sc.nextInt() : null;
2  while (num != null) {
3      boolean prime = true;
4      for (int factor = 2; factor <= num / 2; factor++) {
5          if (num % factor == 0) {
6              System.out.println(factor + " divides " + num);
7              prime = false;
8          }
9      }
10      if (prime) {
11          System.out.println(num + " is prime");
12      }
13      num = sc.hasNextInt() ? sc.nextInt() : null;
14 }
15 System.out.println("End of program");
```

LCSAJ	Primer značenja / uslova
(1, 2, 15)	Ulazni tok je već na EOF; petlja se ne izvršava.
(1,4,10)	Prvi broj je manji od 4; FOR nema iteracija.
(1, 5, 9)	Prvi test deljivosti je FALSE; tok skače ka nastavku FOR.
(1, 9, 5)	Telo FOR je izvršeno; kontrola se vraća na proveru sledećeg faktora.
(1,10,13)	FOR se završio, a Prime = FALSE vodi na čitanje narednog broja.
(2, 4, 10)	Za sledeći broj FOR odmah završava, jer je gornja granica manja od 2.

Ovo su neke od ukupno 23 LCSAJ sekvence za posmatranu funkciju.

Tri broja pokrivaju deo - ali ne sve sekvence

Izvorni početni test ulaz: 5 (prost), 6 (složen) i 2 (poseban slučaj bez prolaska kroz telo FOR).

5 - prost

Faktor 2 ne deli 5; posle FOR važi Prime = TRUE.

6 - složen

Faktori 2 i 3 dele 6; Prime se postavlja na FALSE.

2 - poseban slučaj

Uslov FOR odmah je netačan; broj se klasifikuje kao prost.

9 / 23 LCSAJ sekvenci pokrivenih početnim test-skupom



Pokrivenost = 9/23 = 39% prema tabeli iz izvornog predavanja.

Jedan skup testova može pokriti sve važne grane, a i dalje propustiti konkretne linearne sekvence i njihove ciljeve skoka.

Nove testove biramo prema sekvencama koje nedostaju

Ulazi se biraju da bi zadovoljili uslove LCSAJ-a; više sekvenci može biti pokriveno istim ulaznim tokom.

```
1  Integer num = sc.hasNextInt() ? sc.nextInt() : null;
2  while (num != null) {
3      boolean prime = true;
4      for (int factor = 2; factor <= num / 2; factor++) {
5          if (num % factor == 0) {
6              System.out.println(factor + " divides " + num);
7              prime = false;
8          }
9      }
10      if (prime) {
11          System.out.println(num + " is prime");
12      }
13      num = sc.hasNextInt() ? sc.nextInt() : null;
14 }
15 System.out.println("End of program");
```

Ciljna LCSAJ	Uslov	Primer ulaznog toka
(1, 2, 15)	EOF pre prvog broja	Prazan ulaz
(1, 4, 10)	Prvi broj < 4	2 ili 3
(1, 10, 13)	Prvi broj = 4	4
(1, 9, 5)	Prvi broj > 5; prvi faktor nije delilac	7
(2, 10, 13)	Broj 4 nije prvi na listi	5, 4
(5, 9, 5)	Dodatni složen broj > 7	npr. 5, 9

Preporuka: beležiti ciljnu sekvencu, preduslov ulaza i stvarni ishod;

Nije svaka LCSAJ sekvenca moguća za neki ulaz

Neizvršena sekvenca može biti samo neotkrivena testovima - ili zaista neizvodljiva zbog kontradiktornih uslova.

SEKVENCA KOJA JE NEIZVODLJIVA

```
(1, 14, 2)
(2, 14, 2)
(5, 14, 2)
```

Uslovi bi tražili da se posle linije 7 (Prime = FALSE) ipak izvrši grana 10 -> 11 (Prime = TRUE). To je kontradikcija.

JOŠ JEDNA NEIZVODLJIVA SEKVENCA

```
(10, 10, 13)
```

Da bi se stiglo do početka na liniji 10, FOR je preskočen i Prime ostaje TRUE. Zato ne može uslediti skok 10 -> 13 koji zahteva Prime = FALSE.

Razlikovati: nepokrivena sekvenca (potreban bolji test) ≠ neizvodljiva sekvenca (kontradiktorni uslovi).

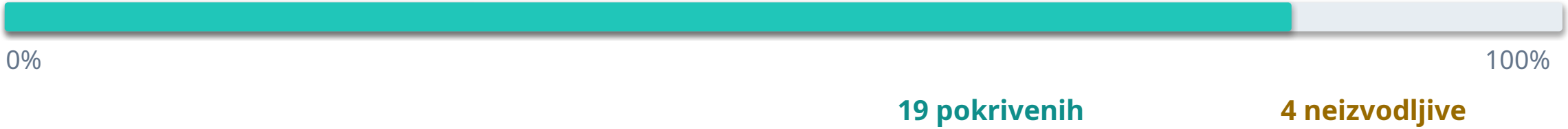
Pokrivenost raste, ali se ne može dostići 100%

Dodatni testovi pokrivaju 19 od 23 LCSAJ sekvence.

Početni skup
9 / 23 = 39% pokrivenosti.

Dodatni testovi
19 / 23 = 82,6%, približno 83%.

Preostale 4
Označene su kao neizvodljive.



Pokrivenost LCSAJ je mera strukturalnog obilaska sekvenci - nije dokaz da je program ispravan niti da svi mogući ulazi/izlazi imaju očekivano ponašanje.

LCSAJ popunjava prostor između odluka i putanja

Različiti kriterijumi opisuju različitu granularnost izvršavanja; izbor zavisi od cilja i složenosti programa.

Kriterijum	Šta traži	Važno ograničenje
Pokrivenost odluka	Svaka grana odluke se izvrši.	Ne vodi računa o broju iteracija petlje.
Bazične putanje	Skup linearno nezavisnih putanja u CFG-u.	Ne znači pokrivanje svih kombinacija i iteracija.
LCSAJ	Linearna sekvenca + cilj prenosa kontrole.	Zahteva sistematsko nabrajanje i proveru izvodljivosti.
Sve putanje	Sve putanje izvršavanja.	Petlje često čine skup beskonačnim.

LCSAJ je koristan za analizu sekvenci odluka/skokova; ne zamenjuje kriterijume zasnovane na uslovima, DU lancima ili proveri rezultata.