UNIVERZITET U BEOGRADU
Elektrotehnički fakultet
Katedra za računarsku tehniku i informatiku

Testiranje softvera

Vežbe - Mutaciono testiranje

Profesor:
dr Dragan Bojić

Asistent:
dipl. ing. Dražen Drašković

Beograd, 31.12.2012.

Teorijske osnove

Mutacija programa je tehnika kojom se mogu unaprediti test primeri. Termin mutacija odnosi se na promenu u programu.

Mutacija je radnja kojom pravimo male promene u programskom kodu. Ako P označimo kao originalni program koji testiramo, a M kao program koji ima neku izmenu u odnosu na P, tada M nazivamo mutant programa P, a P je roditelj programa M. S obzirom da program P mora da bude sintaksno ispravan i da se kompajlira, program M takođe mora da bude sintaksno ispravan.

Primer 1

Program P:

```
1. begin
2.  int x, y;
3.  input(x, y);
4.  if (x < y)
5.  then
6.   output(x+y);
7.  else
8.   output(x*y);
9. end
```

Postoji veliki dijapazon izmena u ovom programu, tako da novodobijeni program bude sintaksno korektan. U nastavku su data dva mutanta: mutant M_1 je dobijen izmenom operatora < operatorom ≤ (u IF izrazu), a mutant M_2 je dobijen izmenom operatora * operatorom / (operaciju množenja zamenila je operacija deljenja, na izlazu).

Mutant M_1:

```
1. begin
2.  int x, y;
3.  input(x, y);
4.  if (x ≤ y)       ← Mutiran izraz
5.  then
6.   output(x+y);
7.  else
8.   output(x*y);
9. end
```

Mutant M_2:

```
1. begin
2.  int x, y;
3.  input(x, y);
4.  if (x < y)
5.  then
6.   output(x+y);
7.  else
8.   output(x/y);    ← Mutiran izraz
9. end
```

Primetimo da su promene u originalnom kodu vrlo jednostavne. Ovde je napravljena samo po jedna promena u mutantima u odnosu na roditeljski program. Na primer, moguće je napraviti mutant dodavanjem novog koda u originalni program.

Mutanti koji imaju samo jednu promenu u odnosu na program koji testiramo, zovu se mutanti prvog reda (engl. *first-order mutants*). Mutanti drugog reda se formiraju pravljenjem dve jednostavne izmene u programu, trećeg reda pravljenjem tri jednostavne izmene u programu, i tako dalje. Može se formirati mutant drugog reda tako što se formira mutant prvog reda od drugog mutanta prvog reda. Slično tome, mutant n-tog reda može se formirati kao mutant prvog reda od mutanta (n-1)-og reda.

Sada ćemo od programa P napraviti mutant M_3, tako što ćemo formirati mutant drugog reda. Mutant M_3 će imati dve izmene: izmenićemo promenljivu y u uslovnom izrazu sa y+1 i zamenićemo operator + u izrazu x+y sa operatorom /.

Mutant M_3:

```
1. begin
2.  int x, y;
3.  input(x, y);
4.  if (x < y+1)     ← Mutiran izraz
5.  then
6.   output(x/y);    ← Mutiran izraz
7.  else
8.   output(x*y);
9. end
```

Mutanti, izuzev mutanta prvog reda, zovu se i mutanti višeg reda (engl. *higher-order mutants*). Mutanti prvog reda su oni koji se uglavnom koriste u praksi. Postoji više razloga zašto se mutanti prvog reda više koriste u praksi u odnosu na mutante višeg reda. Jedan od razloga je to što ima mnogo više mutanata višeg reda, od mutanata prvog reda.

Na primer, program FIND koji ima samo 28 linija Fortran koda, generiše 528 906 mutanata drugog reda. Tako veliki broj mutanata stvara problem skalabilnosti.

U dosadašnjim primerima, prikazane su samo jednostavne sintaksne promene. Ali mogu se vršiti i mutacije sa semantičkim promenama. Imajmo u vidu da je sintaksa nosilac semantike u programima, odnosno da se jedna semantička promena pravi koristeći jednu ili više sintaksnih promena.

U prethodnom primeru, u programu P, možemo uslovni izraz da izdvojimo u jednu funkciju f(x,y):

$$f_P(x,y) = \begin{cases} x + y, x < y \\ x * y, x \geq y \end{cases}$$

Istu funkciju za mutante M_1 i M_2 možemo napisati ovako:

$$f_{M1}(x,y) = \begin{cases} x + y, x \leq y \\ x * y, x > y \end{cases}$$

$$f_{M2}(x,y) = \begin{cases} x + y, x < y \\ x / y, x \geq y \end{cases}$$

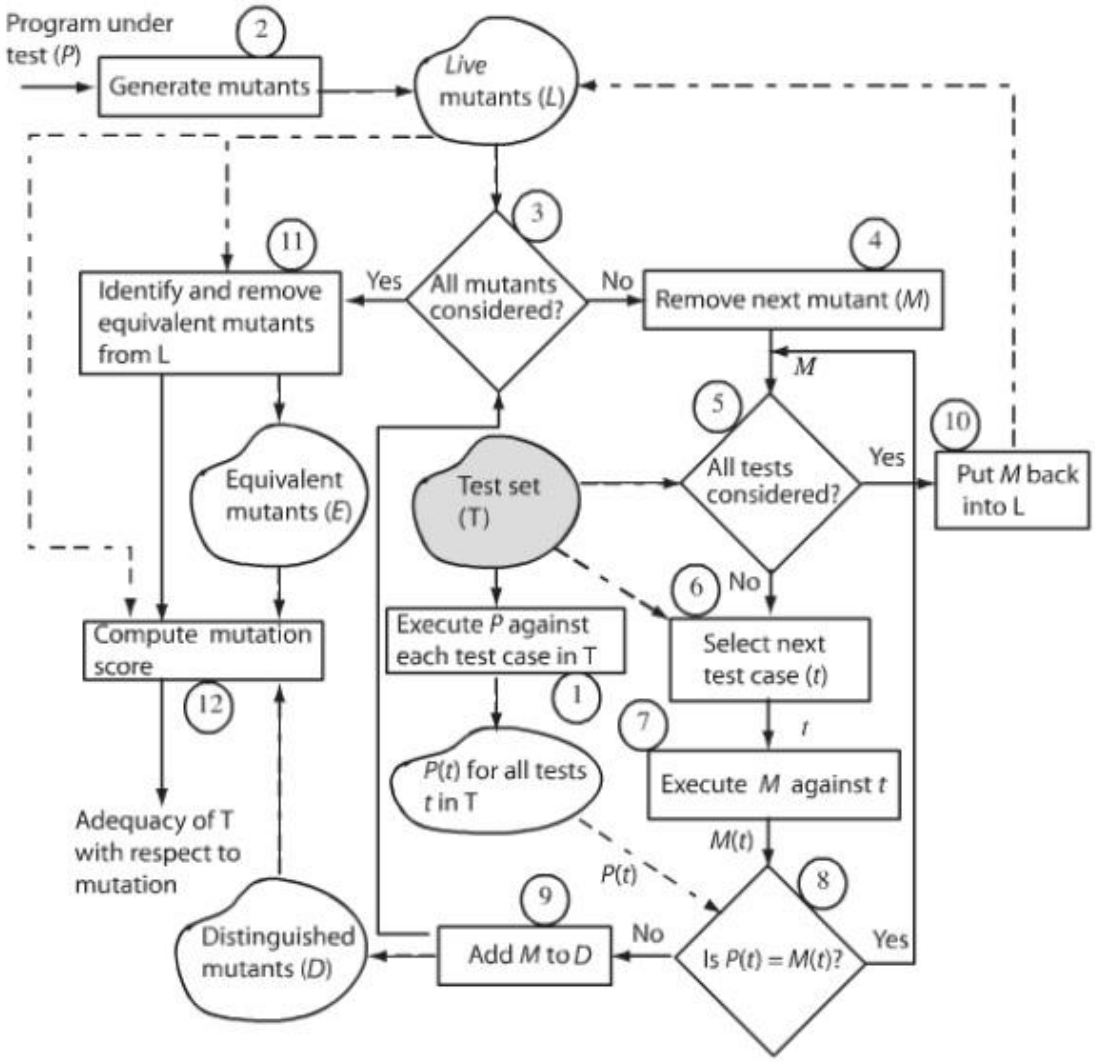
Sve tri funkcije f_P, f_{M1} i f_{M2} su različite. Tako smo promenili semantiku programa, promenom sintakse. Mutacija može da na prvi pogled izgleda kao jednostavna sintaksna promena napravljena u programu, ali jedna takva jednostavna sintaksna promena može imati drastičan uticaj na semantiku programu, ili čak suprotno može uopšte da nema efekat na semantiku program.

Test procena koristeći mutacije

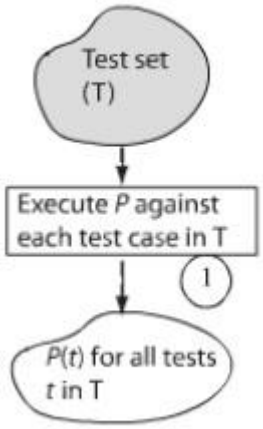
Ako je P program koji testiramo, T skup test primera za program P, a R skup uslova koje P mora da ispuni, pretpostavimo da je program P testiran svim testovima iz skupa T, uz poštovanje svih uslova iz skupa R, i da se korektno izvršava, pa želimo da znamo koliko je dobar skup test primera T.

U zavisnosti od skupa test primera koje odaberemo i programa koji testiramo, možemo izračunati mutacioni rezultat (engl. *mutation score*), koji predstavlja kvantitativnu procenu kvaliteta skupa test primera T. Mutacioni rezultat je broj između 0 i 1. Rezultat 1 označava da je skup test primera T odgovarajući u pogledu mutacija. Rezultat koji je manji od 1, označava da skup test primera T nije odgovarajući u pogledu mutacija. Neadekvatan skup testova se može poboljšati dodavanjem test primera koji bi uvećali mutacioni rezultat.

Na sledećoj slici, prikazan je mogući niz od 12 koraka za procenu kvaliteta date grupe testova korišćenjem mutacije. Iako slika deluje zbunjujuće, prilično je jednostavna sekvenca, kada se prođe korak po korak.



Slika 1: Procedura koja se koristi u proceni kvaliteta skupa testova korišćenjem mutacija. Puna linija označava sledeći korak sekvence. Isprekidana linija ukazuje na prenos podataka između skladišta podataka i trenutnog koraka. L, D i E su notacije za žive (engl. *live*), mrtve (engl. *distinguished* ili *killed*) i jednake (engl. *equivalent*) mutante, koji su definisani. P(t) i M(t) označavaju program i mutant, odnosno njihovo ponašanje prilikom izvršavanja testa t.



(Korak 1) Izvršavanje programa

Prvi korak u proceni kvaliteta skupa test primera T nad programom P i uslova R, je da se izvrši program P za svaki test primer iz skupa T. Notacija P(t) označava da program P izvršava test primer t. Ponašanje određenog programa se izražava kao skup vrednosti izlaznih promenljivih u P.

Ovaj korak nekada i nije potreban, ako je program P već izvršen sa svim test primerima iz skupa T, i P(t) je zabeležen u bazi podataka. U svakom slučaju, krajnji rezultat izvršavanja koraka 1 je baza podataka P(t) za sve $t \in T$.

U ovom trenutku, mi pretpostavljamo da je P tačno u pogledu R za sve test primere t. Ako se utvrdi da je P netačno, greška mora biti ispravljena, a korak 1 ponovo izvršen. Dakle, sekvenca počinje izvršavanje tek kada je utvrđeno da program P ispravno radi sa test primerima iz skupa T.

Razmatraćemo program iz primera 1, koji koristi funkciju $f_P(x,y) = \begin{cases} x+y, x<y \\ x*y, x \geq y \end{cases}$

Pretpostavimo da smo testirali program P koristeći sledeće test primere:

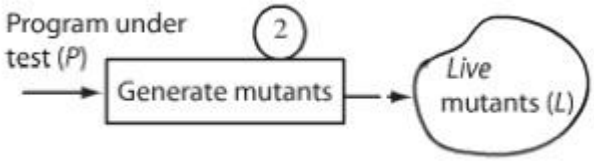
$$T_P = \begin{cases} t_1 : (x=0, y=0) \\ t_2 : (x=0, y=1) \\ t_3 : (x=1, y=0) \\ t_4 : (x=-1, y=-2) \end{cases}$$

Baza podataka P(t) za sve test primere $t \in T_P$ izgleda ovako:

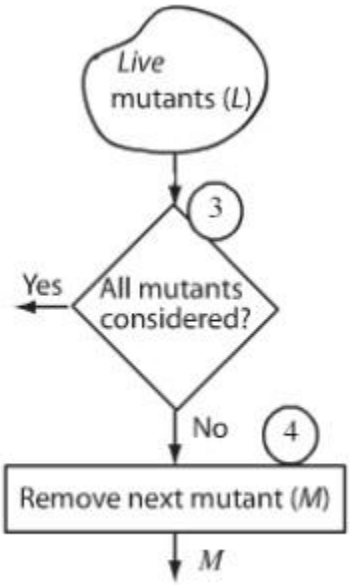
Test primer (t)	Očekivani izlaz f(x,y)	Posmatrani izlat P(t)
t_1	0	0
t_2	1	1
t_3	0	0
t_4	2	2

(Korak 2) Formiranje mutanta

U drugom koraku formiraju se mutanti, kao što je prikazano u primeru 1, i kasnije u zadacima. Svi mutanti se inicijalno stavljaju u skup živih mutanata L (engl. *live*).



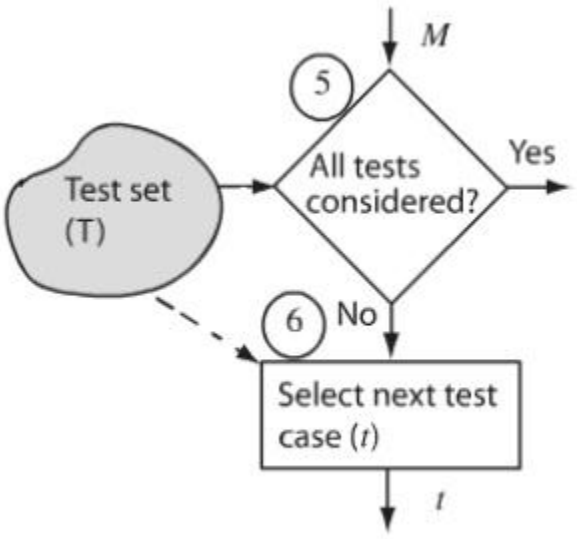
(Koraci 3 i 4) Izbor sledećeg mutanta



U koracima 3 i 4, biramo sledeći mutant koji nije do tada razmatran. Na ovom mestu počinje petlja, u kojoj ćemo proći kroz sve mutante iz skupa živih mutanata L, sve dok taj skup ne postane prazan. Kada taj skup postane prazan, petlja se prekida.

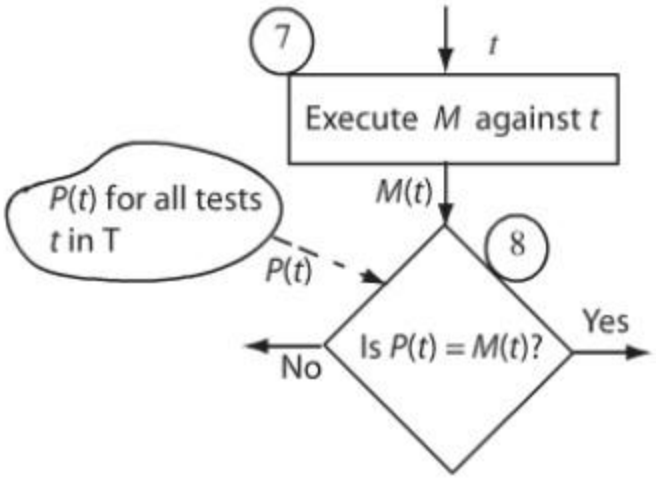
Ova provera čini korak 3. Dakle, ako postoje živi mutanti u skupu L, koji nikada nisu bili izabrani u bilo kom prethodnom koraku, biramo proizvoljni mutant i uklanjamo ga iz skupa L, što čini korak 4.

(Koraci 5 i 6) Izbor sledećeg test primera



Kada izaberemo mutant M, tada pokušamo da nađemo bar jedan test iz skupa T koji se razlikuje od svog roditelja P. Da bismo to utvrdili, moramo da izvršimo M nad testovima iz skupa T. Dakle, u ovom trenutku ulazimo u drugu petlju, koja se izvršava određeni broj puta za svaki izabrani mutant. Petlja se završava kada su svi testovi izvršeni ili se mutant M razlikuje u nekim test primerima.

(Koraci 7, 8 i 9) Izvršavanje mutanta i klasifikacija



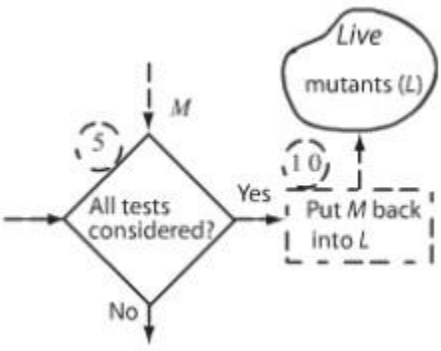
Prvo izaberemo jedan mutant M za izvršavanje nad test primerom t. U koraku 7, mi izvršavamo M nad t. U koraku 8, mi proveravamo da li je rezultat koji je generisao izvršeni mutant M nad testom t isti ili različit od rezultata koji je generisao izvršeni (originalni) program P nad testom t.

Ukoliko je rezultat koji generiše M nad test primerom t_1, isti kao i rezultat koji generiše P nad test primerom t_1, $P(t_1) = M(t_1)$, vraćamo se na korak 5. Dalje ispitujemo za test druge primere t_2, t_3,... t_n, i ako se za sve test primere $t \in T$, ispostavi da je P(t) = M(t), mutant M je živ i stavljamo ga u skup živih mutanata (označen sa L).

Ako se ispostavi da za neki test m, izvršeni mutant daje različit rezultat od izlaza koji daje program P, $P(t_m) \neq M(t_m)$, onda prestaje izvršavanje petlje koja je počela u koraku 5 i skačemo na korak 9. Tada je mutant M mrtav i stavljamo ga u skup mrtvih mutanata (označen sa K ili D).

Ako za prvi test primer t_m, koji izvršavamo nad mutantom M, dobijemo da rezultat roditelja razlikuje od rezultata mutanta, $P(t_m) \neq M(t_m)$, tada možemo da prekinemo izvršavanje ostalih testova t iz skupa T nad tim mutantom, jer ostali test primeri ne moraju da se izvrše kada utvrdimo da su roditelj i mutant različiti.

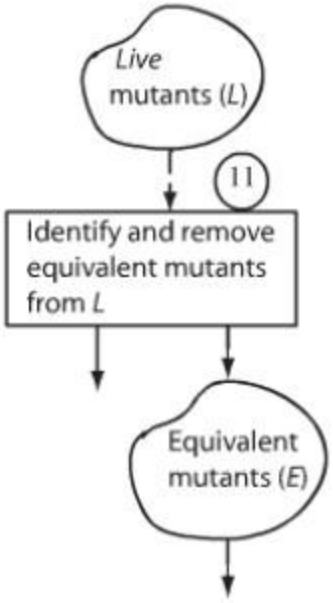
(Korak 10) Živi mutanti



Kada rezultat izvršavanja mutanta M, ne možemo da razlikujemo u odnosu na rezultat izvršavanja programa P, za sve test primere iz skupa T, stavljamo taj mutant M ponovo u skup živih mutanata L.

Ovde treba napomenuti da svaki mutant koji se vraća u skup živih mutanata L, se ne bira opet u koraku 4, ako je već jednom izabran (jer će ponovo da se utvrdi da je živ).

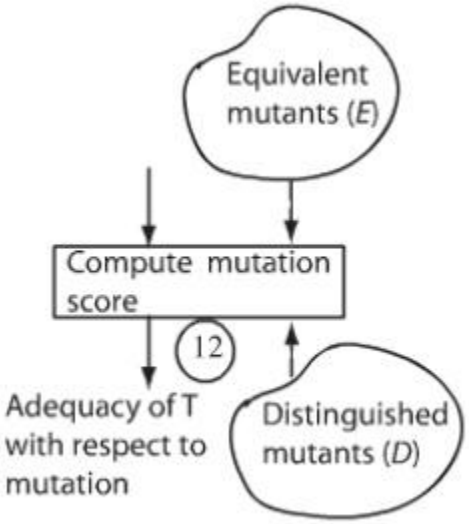
(Korak 11) Jednaki mutanti



Posle izvršavanja svih mutanata, vrši se još jedna provera da li je ostao još neki živi mutant. Preostali živi mutanti se testiraju da li su ekvivalentni sa svojim roditeljskim programom. Mutant M se smatra ekvivalentnim svom roditelju P, ako za svaki ulazni test iz ulaznog domena P, ponašanje M je identično kao i ponašanje P.

Ako nađemo makar jedan test primer, po kome se mutant M ne ponaša isto kao roditelj P, kažemo da mutant M nije ekvivalentan sa roditeljem P. Skup E, ekvivalentnih mutanata, ostaje prazan.

(Korak 12) Izračunavanje mutacionog rezultata



Ovo je poslednji korak u našoj sekvenci za procenu kvaliteta testa. Imajući u vidu skupove L, D (K) i E, mutacioni rezultat (engl. *mutation score*) testova iz skupa T se može izračunati na sledeći način:

, $$MS(T) = \frac{|D|}{|L| + |D|}$$

gde D predstavlja broj mrtvih, a L broj živih mutanata.

Značajno je da se napomene da skup L čini samo žive mutante, a ne i mutante koji su ekvivalentni svojim roditeljima. Kao što se vidi iz formule, mutacioni rezultat uvek ima vrednost između 0 i 1.

Ovaj mutacioni rezultat se može izračunati i na sledeći način:

$MS(T) = \frac{|D|}{|M| - |E|}$, gde D predstavlja broj mrtvih mutanata, M predstavlja ukupan broj mutanata (formiranih u koraku 2), a E broj ekvivalentnih mutanata

Ako grupa test primera T koja se izvršava, razlikuje ponašanje svih mutanata u odnosu na originalni program, osim onih koji su ekvivalentni, onda je L = 0, i mutacioni rezultat je jednak 1. Ako je D = 0, onda je mutacioni rezultat jednak 0.

Mutacioni operatori

Svaki mutacioni operator je označen sa jedinstvenim imenom, radi lakšeg snalaženja. Na primer, ABS je mutacioni operator koji generiše mutante tako što zameni svako pojavljivanje aritmetičkog izraza *e*, izrazom *abs(e)*. Pretpostavlja se da *abs* označava funkciju apsolutne vrednosti, koja se može pronaći u mnogim programskim jezicima.

Za primer 1, možemo generisati 8 mutanata, koristeći ABS mutacioni operator:

Lokacija	**Izraz**	**Mutant**
Linija 4	if (x < y)	if (abs(x) < y)
		if (x < abs(y))
Linija 5	output (x + y)	output (abs(x) + y)
		output (x + abs(y))
		output (abs(x + y))
Linija 7	output (x * y)	output (abs(x) * y)
		output (x * abs(y))
		output (abs(x * y))

Ulazne naredbe i deklaracije se ne mutiraju. Ulazne naredbe se ne mutiraju dodavanjem *abs* operatora, zato što bismo u tom slučaju dobili sintaksno nekorektan program.

Mutacioni operatori su napravljeni tako da modeliraju jednostavne greške u programu koje programer napravi. Greške u programima mogu biti mnogo složenije nego jednostavne greške modelirane od strane mutacionih operatora. Ipak, utvrđeno je da uprkos jednostavnosti mutacija, kompleksne greške se otkrivaju u pokušaju da se napravi razlika mutanata od svojih roditelja.

Skup mutacionih operacija je specifičan za programski jezik, ali i pored toga može biti podeljen u mali skup generičkih kategorija. Jedna kategorizacija prikazana je u sledećoj tabeli:

Kategorija	**Modelovana greška**	**Primeri**
Mutacija konstante	Nekorektna konstanta	x = x + 1; → x = x + 3; y = y * 3; → y = y * 3;
Mutacija operatora	Nekorektan operator	if (x < y) → if (x ≤ y) x++ → ++x
Mutacija naredbe	Nekorektno mesto naredbe	z = x + 1; → Delete break break → z = x + 1;
Mutacija promenljive	Nekorekna korišćena promenljiva	int x, y, z; z = x + 1; → z = y + 1; z = x + y; → z = abs(x) + y

Oznaka → označava da je leva strana mutirala u desnu stranu od strelice.

U svakoj od prikazanih kategorija u praksi ima mnogo mutacionih operatora. Broj i vrsta mutacionih operatora u svakoj od prikazanih kategorija, zavisi od od programskog jezika za koji su operatori dizajnirani. Na primer, za program pisan u ANSI C, moraju se koristiti operatori mutacije za programski jezik C. Java programi se mutiraju koristeći mutacione operatore dizajnirane za programski jezik Java.

Postoje najmanje tri razloga za zavisnost mutacionih operatora od programskog jezika:

- Ako je program, koji se mutirao, sintaksno ispravan, mutacioni operator mora proizvesti mutanta, koji je takođe sintaksno ispravan.
- Sintaksna pravila programskog jezika određuju domen mutacionog operatora (npr. u Javi, domen mutacionog operatora koji menja jedan relacioni operator drugim je skup {<, <=, >, >=, !=, ==}).
- Specifičnosti sintakse jezika imaju uticaj na vrste grešaka koje programer može da napravi.

Lista entiteta koji ne mutiraju:

- Deklaracije
- Adresni operator &
- Formatiranje stringa u U/I funkcijama
- Zaglavlja deklaracije funkcija
- Kontrolne linije
- Imena funkcija kojima se poziva funkcija

Mutacioni operatori u programskom jeziku C

Konstante:

Mutacioni operator	Domen	Opis
CGCR	Konstante	Konstanta se zamenjuje koristeći globalnu konstantu
CLSR	Konstante	Konstanta za skalarnu zamenu koristeći lokalnu konstantu
CGSR	Konstante	Konstanta za skalarnu zamenu koristeći globalnu konstantu
CRCR	Konstante	Obavezna zamena konstante
CLCR	Konstante	Konstanta se zamenjuje koristeći lokalnu konstantu

Zamena konstante kod CRCR može da se vrši u zavisnosti da li je celobrojna ili realna vrednost promenljive. Ako I označava skup {0, 1, -1, u_i}, gde u_i označava celobrojnu promenljivu koju je korisnik odredio, svaka konstanta treba da bude izmenjena elementom iz skupa I. Pokazivač se zamenjuje sa null. Levi operandi operatora dodele ++ i -- se ne mutiraju.

Primer 2

Za izraz `k = j + *p`, gde su k i j celobrojne promenljive, a p pokazivač na celobrojnu promenljivu, primeniti CRCR mutacioni operator.

```
k = 0 + *p
k = 1 + *p
k = -1 + *p
k = u_i + *p
k = j + null
```

Binarni operatori mutacije - Ocor (comparable operator replacement)

Naziv	Domen	Opseg	Primer
OAAA	Aritmetička dodela	Aritmetička dodela	a += b → a -= b
OAAN	Aritmetički	Aritmetički	a + b → a * b
OBBA	Bitska dodela	Bitska dodela	a &= b → a |= b
OBBN	Bitski	Bitski	a & b → a | b
OLLN	Logičko	Logičko	a && b → a || b
ORRN	Relacioni	Relacioni	a < b → a <= b
OSSA	Pomeraj sa dodelom	Pomeraj sa dodelom	a <<= b → a >>= b
OSSN	Pomeraj	Pomeraj	a << b → a >> b

Binarni operatori mutacije - Oior (incomparable operator replacement)

Naziv	Domen	Opseg	Primer
OABA	Aritmetička dodela	Bitska dodela	a += b → a |= b
OAEA	Aritmetička dodela	Obična dodela	a += b → a = b
OABN	Aritmetički	Bitski	a + b → a & b
OALN	Aritmetički	Logički	a + b → a && b
OARN	Aritmetički	Relacioni	a + b → a < b
OASA	Aritmetička dodela	Pomeraj sa dodelom	a += b → a <<= b
OASN	Aritmetički	Pomeraj	a + b → a << b
OBAA	Bitska dodela	Aritmetička dodela	a |= b → a += b
OBAN	Bitski	Aritmetički	a & b → a + b
OBEA	Bitska dodela	Obična dodela	a &= b → a = b
OBLN	Bitski	Logički	a & b → a && b
OBRN	Bitski	Relacioni	a & b → a < b
OBSA	Bitska dodela	Pomeraj sa dodelom	a &= b → a <<= b
OBSN	Bitski	Pomeraj	a & b → a << b
OEAA	Obična dodela	Aritmetička dodela	a = b → a += b
OEBA	Obična dodela	Bitska dodela	a = b → a &= b
OESA	Obična dodela	Pomeraj sa dodelom	a = b → a <<= b
OLAN	Logički	Aritmetički	a && b → a + b
OLBN	Logički	Bitski	a && b → a & b
OLRN	Logički	Relacioni	a && b → a < b
OLSN	Logički	Pomeraj	a && b → a << b
ORAN	Relacioni	Aritmetički	a < b → a + b
ORBN	Relacioni	Bitski	a < b → a & b
ORLN	Relacioni	Relacioni	a < b → a && b
ORSN	Relacioni	Pomeraj	a < b → a << b
OSAA	Pomeraj sa dodelom	Aritmetička dodela	a <<= b → a += b
OSAN	Pomeraj	Aritmetički	a << b → a + b
OSBA	Pomeraj sa dodelom	Bitska dodela	a << b → a |= b
OSBN	Pomeraj	Bitski	a << b → a & b
OSEA	Pomeraj sa dodelom	Obična dodela	a <<= b → a = b
OSLN	Pomeraj	Logički	a << b → a && b
OSRN	Pomeraj	Relacioni	a << b → a < b

```
if (uslov) izraz    →    if (!uslov) izraz

if (uslov) izraz else izraz   →
                          if (!uslov) izraz else izraz

while (uslov) izraz      →    while (!uslov) izraz

do izraz while (uslov)   →    do izraz while (!uslov)

for (izraz; uslov; iterator) izraz →
                    for (izraz; !uslov; iterator) izraz

uslov ? izraz1 : izraz2  →    !uslov ? izraz1 : izraz2
```

Unarni operatori mutacije

Inkrementiranje/Dekremtentiranje:

OPPO mutacioni operator generiše dva mutanta. Na primer izraz ++x mutira u: x++ i --x. Izraz x++ mutira u: ++x i x--.

OMMO mutacioni operator generiše slično. Na primer izraz --x mutira u: x-- i ++x. Izraz x-- mutira u: --x i x++.

Kada imamo FOR petlju, izraz za inkrementiranje (ili dekrementiranje) petlje, i++ (ili i--) se ne mutira, čime se izbegava stvaranje ekvivalentnog mutanta. Izraz *x++ će mutirati u *++x i *x--.

Logička negacija:

OLNG je mutacioni operator koji razmatra izraz: *x op y*, gde *op* može da bude logički operator && ili ||. OLNG generiše tri mutanta: *x op !y*, *!x op y* i *!(x op y)*.

Mutiranje izraza

Mutacioni operator	Domen	Opis
SBRC	break	break se zamenjuje sa continue
SBRn	break	break se izbacuje na n-ti nivo
SCRB	continue	continue se zamenjuje sa break
SDWD	do-while	do-while se zamenjuje sa while
SGLR	goto	goto labela se zamenjuje
SMVB	izraz	pomeranje zagrada
SRSR	return	return se zamenjuje
SSDL	izraz	brisanje izraza
SSOM	izraz	operator mutacije sekvence izraza
STRI	if izraz	zamena u if uslovu
STRP	izraz	zamena kod izvršavanja izraza
SMTC	iterativni izraz	n-puta continue
SSWM	switch israz	mutacija switch
SMTT	iterativni izraz	n-puta zamena
SWDD	while	while se zamenjuje sa do-while

STRP je mutacioni operator koji ima za cilj da otkrije nedostatak koda u programu. Svaki izraz se sistematski zamenjuje sa trap_on_statement(). Mutant završava izvršavanje, kada se izvrši trap_on_statement().

Primer 3

```
while (x != y)
{
   if (x < y)
      y -= x;
   else
      x -= y;
}
```

STRP generiše 4 mutanta u ovom primeru:

```
M1:
trap_on_statement();
```

```
M2:
while (x != y)
{
     trap_on_statement();
}

M3:
while (x != y)
{
   if (x < y)
     trap_on_statement();
   else
      x -= y;
}

M4:
while (x != y)
{
   if (x < y)
      y -= x;
   else
     trap_on_statement();
}
```

Primer 4

```
if (c) S1
   else S2
```

STRP generiše 2 mutanta u ovom primeru:

```
M1:
if (c) trap_on_statement()
   else S2

M2:
if (c) S1
   else trap_on_statement()
```

Primer 5

Mutacioni operator SSDL je dizajnitan tako da pokaže da svaki izraz u programu ima efekat na izlazne podatke. Ovaj operator briše sistematski svaki izraz u programu i na primer za primer broj 3, generiše 4 mutanta:

```
M1:
...

M2:
while (x != y)
{
      ...
}

M3:
while (x != y)
{
   if (x < y)
      ...
   else
       x -= y;
}

M4:
while (x != y)
{
   if (x < y)
       y -= x;
   else
      ...
}
```

Primer 6

```
   while (--lim>0 && (c=getchar())!=EOF && c!='\n')
     s[i++]=c
```

Primenom operatora SWDD, dobijamo mutiranu petlju:

```
do {
  s[i++]=c
}
while (--lim>0 && (c=getchar())!=EOF && c!='\n')
```

Primer 7

Ako imamo sekvencu od n izraza, i_1, i_2,... i_n, odvojenih zarezima, SSOM operator generiše (n-1) mutanata za te izraze, tako što svaki put rotira sekvencu u levo.

```
for (i=0, j=strlen(s)-1; i<j; i++, j--) {
  c=s[i], s[i]=s[j], s[j]=c;
}

M1: //jedna rotacija u levo
for (i=0, j=strlen(s)-1; i<j; i++, j--) {
  s[i]=s[j], s[j]=c, c=s[i];
}

M2: //druga rotacija u levo
for (i=0, j=strlen(s)-1; i<j; i++, j--) {
  s[j]=c, c=s[i], s[i]=s[j];
}
```

U ovom programu primenom SSOM, mogu nastati još dva dodatna mutanta. Jedan mutant nastaje zamenom `i=0, j=strlen(s)-1;` sa `j=strlen(s)-1, i=0;` a drugi mutant nastaje zamenom `i++, j--` sa `j--, i++`.

Mutacioni operatori u programskom jeziku Java

Java, kao i mnogi drugi programski jezici, je objektno-orijentisana. Svaki takav jezik pruža sintaksne konstrukcije koje enkapsuliraju podatke i procedure u objekte. Klasa predstavlja jedan šablon za objekat, a objekat jednu instancu (primerak) klase. Procedure unutar klase nazivamo metode. Metod se piše na tradicionalan način, sa dodelama vrednosti, uslovima i petljama.

Zbog postojanja klasa i mehanizma nasleđivanja u Javi, programeri vrlo često prave greške, pa zbog takvih grešaka program ne može da se izvršava. Dakle, mutacioni operatori za mutiranje Java programa se dele na dve kategorije: tradicionalne mutacione operatore i klasne mutacione operatore.

U sledećoj tabeli dati su tradicionalni mutacioni operatori, koji nastaju kada se prave greške u proceduralnom programiranju:

Mutacioni operator	Domen	Opis
ABS	Aritmetički izraz	Zamenjuje aritmetički izraz *exp* sa *abs(exp)*
AOR	Binarni aritmetički operator	Zamenjuje binarni aritmetički operator sa drugim binarnim aritmetičkim operatorom, validnim za dati sadržaj
LCR	Logička veza	Zamenjuje logičku vezu drugom
ROR	Relacioni operator	Zamenjuje relacioni operator drugim
UOI	Aritmetički ili logički izraz	Ubacuje unarni operator, kao što su unarni minus ili logičko ne

Primer 8

Ako imamo x = y + z, ABS operator generiše mutante menjajući svaku komponentu u aritmetičkom izrazu sa apsolutnom vrednošću:

```
x = abs(y) + z
x = y + abs(z)
x = abs(y+z)
```

Klasno-orijentisani mutacioni operatori, dele se u četiri kategorije:

- mut.operatori koji nastaju pojavom grešaka u nasleđivanju,
- mut.operatori koji nastaju pojavom grešaka u polimorfizmu i dinamičkoj obradi,
- mut.operatori koji nastavju prilikom preklapanja metoda,
- objektno-orijentisane i Java-specifične kategorije.

Notacija: $P \overset{Mut.op.}{\Rightarrow} Q$

ukazuje na to da mutacioni operator Mut.Op. kada se primenjuje na programu P kreira mutanta Q, kao jednog od nekoliko mogućih mutanata.

U sledećoj tabeli dati su mutacioni operatori, koji nastaju kada se prave greške u nasleđivanju:

Mutacioni operator	Domen	Opis
IHD	Varijable	Uklanja deklaraciju varijable *x* u potklasi *C*, ako je *x* deklarisano u *parent(C)*
IHI	Potklase	Dodaje deklaraciju za varijablu *x* u potklasu *C*, ako je *x* deklarisano u *parent(C)*
IOD	Metode	Uklanja deklaraciju metoda *m* iz potklase *C*, ako je *m* deklarino u *parent(C)*
IOP	Metode	Unutar potklase, pomera poziv *super.M(..)* u metod *m* - na početak *m*, na kraj *m*, jedan izraz ispod u *m*, i jedan izraz iznad u *m*
IOR	Metode	Imajući u vidu da metod f_1 poziva metod f_2 u *parent(C)*, menja ime f_2 u f_2', ako je f_2 poništeno u potklasi *C*
ISK	Pristup roditeljskoj klasi	Zamenjuje eksplicitno pozivanje promenljive *x* u *parent(C)* zato što je *x* poništeno u potklasi *C*
IPC	poziv nadklase (`super`)	Briše ključnu reč *super* iz konstruktora u potklasi *C*

Primer 9

Ako imamo IHD operator, on će ukloniti deklaracij varijable, ukoliko ta deklaracija poništava deklaraciju roditeljske klase:

```
class Planeta {
   double udaljenost;
   ...
}

class udaljenaPlaneta
extends Planeta {
   double udaljenost;
   ...
}
```

IHD ⟹

```
class Planeta {
   double udaljenost;
   ...
}

class udaljenaPlaneta
extends Planeta {
   //deklaracija uklonjena
   ...
}
```

Operator IHI, radi obrnuti proces od IHD, odnosno dodaje deklaraciju u potklasu:

```
class Planeta {
   double udaljenost;
   ...
}

class udaljenaPlaneta
extends Planeta {
   ...
}
```

IHI ⟹

```
class Planeta {
   double udaljenost;
   ...
}

class udaljenaPlaneta
extends Planeta {
   double udaljenost;
   ...
}
```

Operator IOD uklanja deklaraciju metode:

```
class Planeta {
   String ime;
   Orbita orb(...);
   ...
}

class udaljenaPlaneta
extends Planeta {
   Orbita orb(...);
   ...
}
```

IOD ⇒

```
class Planeta {
   String ime;
   Orbita orb(...);
   ...
}

class udaljenaPlaneta
extends Planeta {
   //metoda orb uklonjena
   ...
}
```

Operator IOP menja poziciju bilo kog pozivanja neke preklopljene metode, koja ima ključnu reč super. Ovde je prikazan samo jedan mutant, ali je moguće formirati dodatna tri mutanta.

```
class Planeta {
   String ime;
   Orbita orb(...);
   PType tip = velika;
   ...
}

class udaljenaPlaneta
extends Planeta {
   Orbita orb(...);
   PType tip = mala;
   super.orb();
   ...
}
```

IOP ⇒

```
class Planeta {
   String ime;
   Orbita orb(...);
   PType tip = velika;
   ...
}

class udaljenaPlaneta
extends Planeta {
   Orbit orb(...);
   ...
   super.orb();
   PType tip = mala;
   ...
}
```

Kao što je prikazano u nastavku, metoda orb() u klasi Planeta poziva metodu provera(). Operator IOR menja naziv poziva te metode u nova_provera.

```
class Planeta {
   String ime;
   Orbita orb(...);
   { ... provera();
     ... }
   void provera(...){
   ...
   }
   ...
}
```

IOR ⟹

```
class Planeta {
   String ime;
   Orbita orb(...);
   { ... nova_provera();
   ... }
   void provera(...){
   ...
   }
   ...
}
```

```
class udaljenaPlaneta
extends Planeta {
   void provera(...){
   ...
   }
   ...
}
```

```
class udaljenaPlaneta
extends Planeta {
   void provera(...){
   ...
   }
   ...
}
```

ISK operator formira mutanta tako što briše ključnu reč super u potklasi:

```
class udaljenaPlaneta
extends Planeta {
   ...
   p = super.ime
   ...
}
```

ISK ⟹

```
class udaljenaPlaneta
extends Planeta {
   ...
   p = ime
   ...
}
```

Konačno, IPC operator u ovoj kategoriji formira mutante tako što zamenjuje poziv podrazumevanog konstruktora roditeljske klase:

```
class udaljenaPlaneta                    class udaljenaPlaneta
           extends Planeta {                        extends Planeta {
   ...                                      ...
   udaljenaPlaneta(String p);     IPC       udaljenaPlaneta(String p);
   ...                            ⟹        ...
   super(p)                                 //Uklonjen poziv
   ...                                      ...
}                                        }
```

U sledećoj tabeli dati su mutacioni operatori, koji nastaju kada se prave greške u polimorfizmu i dinamičkoj obradi:

Mutacioni operator	Domen	Opis
PMC	Primerci objekta	Zamenjuje se tip t_1 sa t_2 instance u objektu koristeći *new*; t_1 je roditeljski tip, a t_2 tip njegove potklase
PMD	Deklaracije objekta	Zamenjuje tip t_1 objekta *x* tipom t_2 njegovog roditelja
PPD	Parametri	Za svaki parametar, zamenjuje tip t_1 objekta *x* sa tipom t_2 njegovog roditelja
PRV	Reference objekta	Zamenjuje referencu objekta, npr. O_1 na desnoj strani dodele po referenci u kompatibilan tip objekta reference O_2

```
udaljenaPlaneta p;              PMC  planeta p;
p = new udaljenaPlaneta();      ⟹   p = new udaljenaPlaneta();
```

```
Planeta p;                      PMD  planeta p;
p = new Planeta();              ⟹   p = new udaljenaPlaneta();
```

```
void duzinaOrbite
      (udaljenaPlaneta p){
   ...
}
```

PPD ⟹

```
void duzinaOrbite(Planeta p){
   ...
}
```

```
Element anElement;
specialElement sElement;
gas g;
...
anElement = sElement;
```

PRV ⟹

```
Element anElement;
specialElement sElement;
gas g;
...
anElement = g;
```

Mutacioni operatori u Javi koji menjaju program u skladu sa greškama prilikom preklapanja metoda:

Mutacioni operator	Domen	Opis
OMR	Preklopljene metode	Zamenjuje telo jedne preklopljene metode sa telom druge metode
OMD	Preklopljene metode	Briše preklopljenu metodu
OAO	Metode	Menja redosled parametara metode
OAN	Metode	Briše argumente u preklopljenim metodama

```
void init(int i) { ... }
void init(int i, String s) {
   ...
}
```

OMR ⟹

```
void init(int i) { ... }
void init(int i, String s) {
   this.init(i);
}
```

```
void init(int i) { ... }
void init(int i, String s) {
   ...
}
```

OMR ⟹

```
void init(int i) {
   this.init(i);
}
void init(int i, String s) {
   ...
}
```

```
void init(int i) { ... }
void init(int i, String s) {
   ...
}
```

OMD ⇒

```
void init(int i) { ... }

//Druga init metoda obrisana
```

```
Orbita.dohvatiOrbitu(p, 4);
```

OMR ⇒

```
Orbita.dohvatiOrbitu(4, p);

ili

Orbita.dohvatiOrbitu(p);
```

Nekoliko dodatnih operatora specifičnih za Javu i OOP jezike može da izazove greške. U sledećoj tabeli prikazani su neki mutacioni operatori koji prave takve mutante:

Mutacioni operator	Domen	Opis
JTD	this	Brisanje ključne reči *this*
JSC	Klasne promenljive	Promena klasne promenljive u promenljivu instance
JID	Promenljive članice	Uklanjanje inicijalizacije za promenljivu članicu
JDC	Konstruktori	Uklanjanje korisnički-definisanih konstruktora
EOA	Reference objekata	Zameniti referencu na objekat njegovim sadržajem koristeći *clone*
EOC	Izraz za upoređivanje	Zameniti == sa *equals*
EAM	Poziv pristupnih metoda	Zameniti poziv metode pozivom kompatibilne pristupne metode
EMM	Poziv modifikovanih metoda	Zameniti poziv metode pozivom kompatibilne modifikovane metode

Zadatak 1

U sledećem programu izvršiti mutaciono testiranje i formirati sledeće mutante:

(a) kod aritmetičkih operatora, sabiranje zameniti oduzimanjem, a množenje deljenjem,

(b) svaku celobrojnu promenljivu *v*, zameniti sa *v+1*,

za svaki izraz gde je to moguće.

```
1. begin
2.  int x, y;
3.  input(x, y);
4.  if (x < y)
5.  then
6.   output(x+y);
7.  else
8.   output(x*y);
9. end
```

1) Napisati četiri test primera sa različitim parovima ulaznih promenljivih x i y i pokazati da li su dati mutanti živi ili mrtvi.

2) Izračunati mutacioni skor nakon tačke a.

Rešenje

Prvo ćemo formirati mutante kod svakog izraza gde je to moguće:

Linija koda	Originalni program	Mutant ID	Izmena u Mutant programu
1	`begin`		-
2	`int x,y`		-
3	`input (x, y)`		-
4	`if (x<y)`	M_1	`if (x+1<y)`
		M_2	`if (x<y+1)`
5	`then`		-
6	`output (x+y)`	M_3	`output (x+1+y)`
		M_4	`output (x+y+1)`
		M_5	`output (x-y)`
7	`else`		-
8	`output (x*y)`	M_6	`output ((x+1)*y)`
		M_7	`output (x*(y+1))`
		M_8	`output (x/y)`
9	`end`		-

Ovde primetite da nismo mutirali deklaracije, input izraz, then i else izraze. Takođe, oznake za početak i kraj programa, begin i end nismo mutirali. Na kraju koraka 2, gore navedene sekvence (algoritma) koju smo prikazali, imaćemo formiranih 8 mutanata. Označićemo te mutante kao žive (engl. *live*) mutante. Ti mutanti su živi, zato što još uvek nismo utvrdili njihove razlike u odnosu na originalni program. To ćemo uraditi u sledećih nekoliko koraka. Dakle, na početku postoji skup živih mutanata L={M_1, M_2, M_3, M_4, M_5, M_6, M_7, M_8} i skup mrtvih (engl. *killed*) mutanata K={}, koji je prazan. Ovaj skup mrtvih mutanata se još označava i sa D (engl. *distinguished*), jer mutant predstavlja program koji se razlikuju od svog roditelja, odnosno originalnog programa.

Testiraćemo program sledećim test primerima:

$$T_P = \begin{cases} t_1 : (x = 0, y = 0) \\ t_2 : (x = 0, y = 1) \\ t_3 : (x = 1, y = 0) \\ t_4 : (x = -1, y = -2) \end{cases}$$

Program	t_1	t_2	t_3	t_4	Killed
P(t)	0	1	0	2	{}
Mutant					
M_1(t)	0	0*	NE	NE	{M_1}
M_2(t)	0	1	0	2	{M_1}
M_3(t)	0	2*	NE	NE	{M_1, M_3}

$M_4(t)$	0	2*	NE	NE	$\{M_1, M_3, M_4\}$
$M_5(t)$	0	-1*	NE	NE	$\{M_1, M_3, M_4, M_5\}$
$M_6(t)$	0	1	0	0*	$\{M_1, M_3, M_4, M_5, M_6\}$
$M_7(t)$	0	1	1*	NE	$\{M_1, M_3, M_4, M_5, M_6, M_7\}$
$M_8(t)$	ND*	NE	NE	NE	$\{M_1, M_3, M_4, M_5, M_6, M_7, M_8\}$

ND = Izlaz nije definisan (u testu t_1 kod mutanta M_8 pokušava se deljenje sa nulom)
NE = Not executed (mutant ne mora da se izvrši)
*** = Prvi test kod koga smo utvrdili da se rezultat testa nad mutantom razlikuje od testa nad programom**

2) Mutacioni rezultat:

$$|D| = 7,$$
$$|L| = 1,$$
$$|E| = 0,$$
$$MS(T_P) = \frac{7}{(7+1)} = 0.875$$

Zadatak 2

Neka je dat deo bankarskog softverskog sistema. U sledećem programskom Java kodu izvršiti mutaciono testiranje mutacionom operacijom ROR. Ulazni argument metode je prosečni iznos plate. Izračunati mutacioni rezultat, ako se na ulaz metode dovode sledeće vrednosti: 0 dinara, 15000 dinara i 35000 dinara.

```
public String kreditnoSposoban(double iznosPlate) {
  if(iznosPlate >= 30 000) {
        return "Kredit odobren!";
  }
  else {
        return "Kredit nije odobren!";
  }

}
```

Rešenje

Mutant M1:

```
public String kreditnoSposoban(double iznosPlate) {
  if(iznosPlate > 30 000) {
        return "Kredit odobren!";
  }
  else {
        return "Kredit nije odobren!";
  }
}
```

Mutant M2:

```
public String kreditnoSposoban(double iznosPlate) {
  if(iznosPlate < 30 000) {
        return "Kredit odobren!";
  }
  else {
        return "Kredit nije odobren!";
  }
}
```

Mutant M3:

```
public String kreditnoSposoban(double iznosPlate) {
  if(iznosPlate <= 30 000) {
        return "Kredit odobren!";
  }
  else {
        return "Kredit nije odobren!";
  }
}
```

Mutant M4:

```
public String kreditnoSposoban(double iznosPlate) {
  if(iznosPlate == 30 000) {
        return "Kredit odobren!";
  }
  else {
        return "Kredit nije odobren!";
  }
}
```

Mutant M5:

```
public String kreditnoSposoban(double iznosPlate) {
  if(iznosPlate != 30 000) {
        return "Kredit odobren!";
  }
  else {
        return "Kredit nije odobren!";
  }
}
```

Testiraćemo program navedenim test primerima:

$$T_P = \begin{cases} t_1 : iznosPlate = 0 \\ t_2 : iznosPlate = 15000 \\ t_3 : iznosPlate = 35000 \end{cases}$$

Program	**t_1**	**t_2**	**t_3**	**Live**	**Kill**
P(t)	KN	KN	KO	{M_1, M_2, M_3, M_4, M_5}	{}
Mutant					
M_1(t)	KN	KN	KO	{M_1, M_2, M_3, M_4, M_5}	{}
M_2(t)	KO	-	-	{M_1, M_3, M_4, M_5}	{M_2}
M_3(t)	KO	-	-	{M_1, M_4, M_5}	{M_2, M_3}
M_4(t)	KN	KN	KN	{M_1, M_5}	{M_2, M_3, M_4}
M_5(t)	KO	-	-	{M_1}	{M_2, M_3, M_4, M_5}

KN = kredit nije odobren
KO = kredit odobren

Mutacioni rezultat:

$$MS(T) = \frac{|KILL|}{|LIVE| + |KILL|} = \frac{4}{1+4} = 0.8$$

U slučaju da je za test primer t4 odabran iznosPlate = 30000, svi mutanti bi bili mrtvi, pa bi mutacioni rezultat iznosio 1.