Vežbe - Objektno-orijentisano testiranje

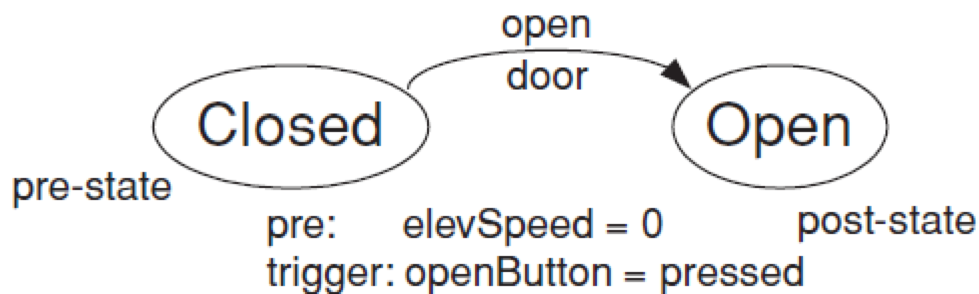
Teorija

Finale State Machine (FSM) - Mašina sa konačnim brojem stanja ili Graf stanja i prelaza

U testiranju softvera:

FSM se prikazuje kao graf čiji čvorovi reprezentuju stanja koja predstavljaju stanja izvršavanja softvera, a veze predstavljaju prelaze između stanja. Stanje predstavlja prepoznatljivu situaciju koja ostaje u toj egzistenciji tokom nekog vremenskog perioda. Stanje se definiše posebnim vrednostima za skup promenljivih, i dok promenljive imaju te vrednosti, smatra se da je softver u tom stanju.

Sledeća slika ilustruje model sa jednostavnim prelazom kojim se otvaraju vrata lifta. Ako je dugme u liftu pritisnuto (događaj koji se hvata), vrata će se otvoriti samo ako se lift ne pomera (preduslov: elevSpeed=0).

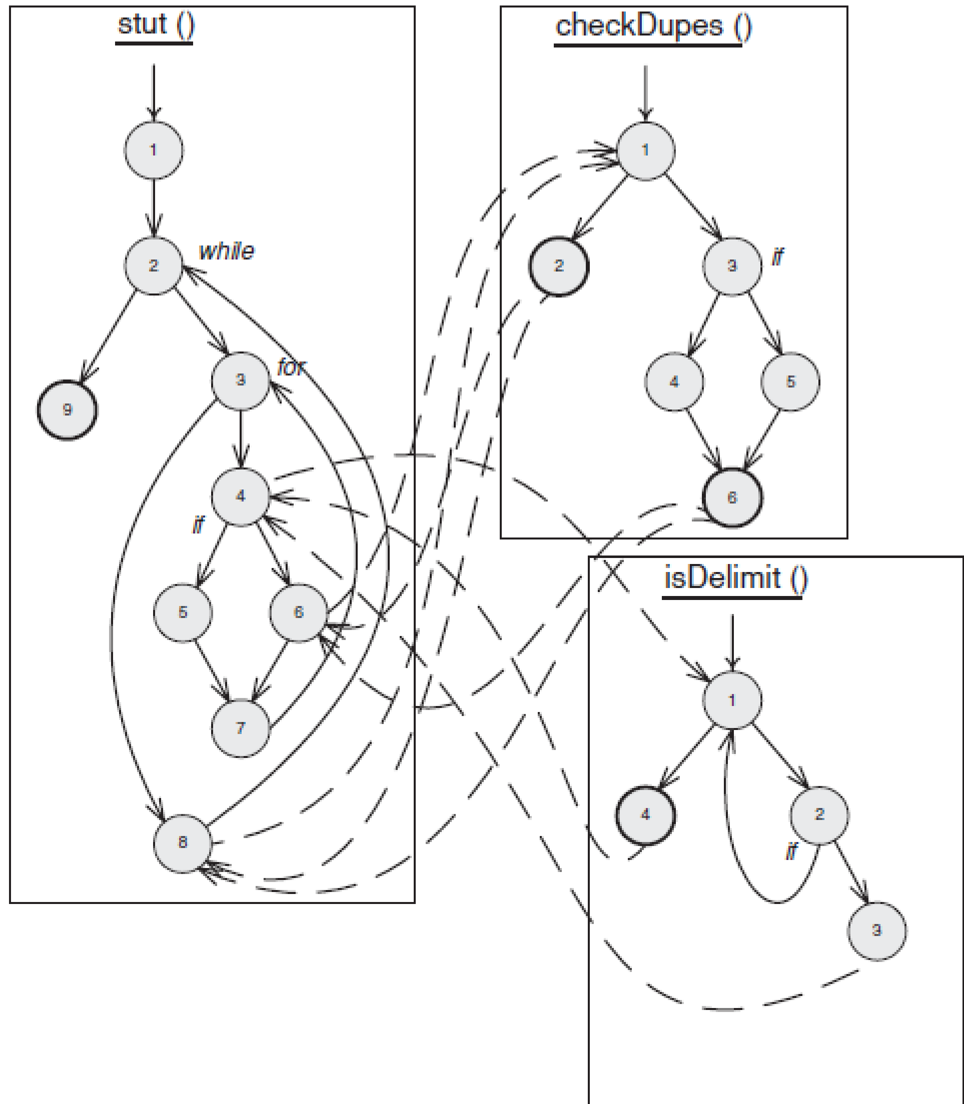


Zadatak 1

```
/** *****************************************************
// Stutter checks for repeat words in a text file.
// It prints a list of repeat words, by line number.
// Stutter will accept standard input or a list
// of file names.
// Jeff Offutt, June 1989 (in C), Java version March 2003
//******************************************************* */
class Stutter
{
// Class variables used in multiple methods.
private static boolean lastdelimit = true;
private static String curWord = "", prevWord = "";
```

```
private static char delimits [] =
{'', ' ', ',', '.', '!', '-', '+', '=', ';', ':', '?',
'&', '{', '}', '\\'}; // First char in list is a tab
//**********************************************
// main parses the arguments, decides if stdin
// or a file name, and calls Stut().
//**********************************************
public static void main (String[] args) throws IOException
{
String fileName;
FileReader myFile;
BufferedReader inFile = null;
if (args.length == 0)
{ // no file, use stdin
inFile = new BufferedReader (new InputStreamReader (System.in));
}
else
{
fileName = args [0];
if (fileName == null)
{ // no file name, use stdin
inFile = new BufferedReader (new InputStreamReader (System.in));
}
else
{ // file name, open the file.
myFile = new FileReader (fileName);
inFile = new BufferedReader (myFile);
}
}
stut (inFile);
}
//**********************************************
// Stut() reads all lines in the input stream, and
// finds words. Words are defined as being surrounded
// by delimiters as defined in the delimits[] array.
// Every time an end of word is found, checkDupes()
// is called to see if it is the same as the
// previous word.
//**********************************************
private static void stut (BufferedReader inFile) throws IOException
{
String inLine;
char c;
int linecnt = 1;
while ((inLine = inFile.readLine()) != null)
{ // For each line
      for (int i=0; i<inLine.length(); i++)
      { // for each character
            c = inLine.charAt(i);
            if (isDelimit (c))
                  { // Found an end of a word.
                        checkDupes (linecnt);
                  }
            else
                  {
```

```
                    lastdelimit = false;
                    curWord = curWord + c;
                    }
        }
        linecnt++;
        checkDupes (linecnt);
}
} // end Stut
//*************************************************
// checkDupes() checks to see if the globally defined
// curWord is the same as prevWord and prints a message
// if they are the same.
//*************************************************
private static void checkDupes (int line)
{
if (lastdelimit)
        return; // already checked, keep skipping
lastdelimit = true;
if (curWord.equals(prevWord))
{
        System.out.println ("Repeated word on line " + line + ": " +
        prevWord+""+curWord);
}
else
{
        prevWord = curWord;
}
curWord = "";
} // end checkDupes
//*************************************************
// Checks to see if a character is a delimiter.
//*************************************************
private static boolean isDelimit (char C)
{
for (int i = 0; i < delimits.length; i++)
        if (C == delimits [i])
                return (true);
        return (false);
}
} // end class Stutter
```



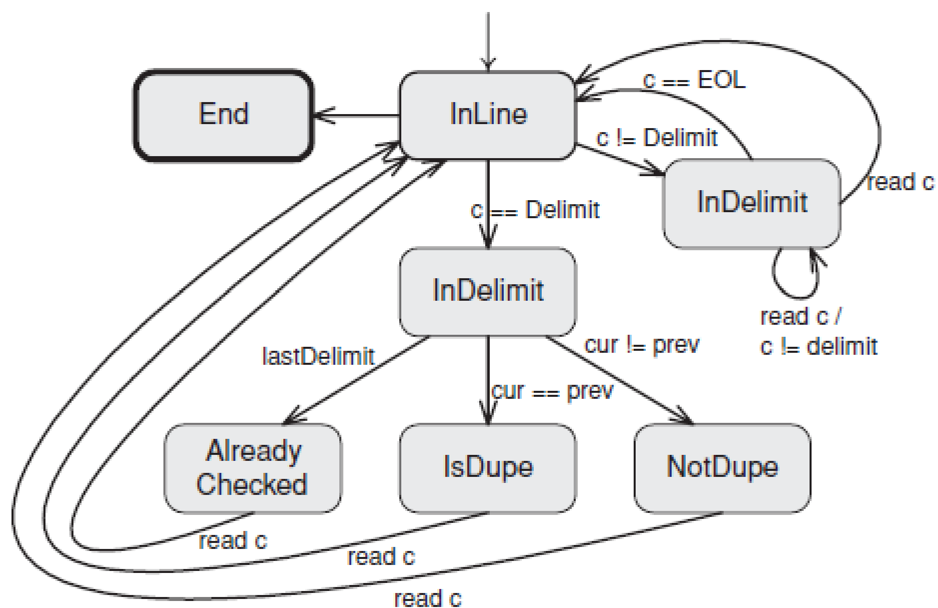
1. FSM koji reprezentuje Stutter, baziran je na grafu kontrole toka za ove 3 metode (tzv. metod kombinovanja grafova toka kontrole).

Graf prikazan na gornjoj slici nije u potpunosti FSM, i ovo nije način da se crtaju grafici iz softvera. Ova metoda ima nekoliko problema, prvi je da čvorovi nisu stanja. Metode moraju da vrate rezultat na odgovarajućim mestima poziva tih metoda, što znači da grafovi sadrže ugrađene ne-determinizme. Na primer, postoje veze između čvora 2 u metodi checkDupes() ka čvoru 6 u metodi shut() i takođe veza između čvora 2 u checkDupes() ka čvoru 8 u metodi shut(). Mesto povratka zavisi odakle smo zvali metodu checkDupes(), iz čvora 6 ili čvora 8 u stut(). Drugi problem može biti taj što implementacija samog softvera mora biti kompletno završena, pre nego što počnemo da pravimo graf. Ipak, naš cilj je uvek da pripremimo testove što je ranije moguće. Najvažniji problem je ne srazmernost grafa veličini softvera. Možemo primetiti da graf postaje vro komplikovan sa samo ove tri metode, a sa velikim programima graf postaje mnogo gori.

2. FSM baziran na softverskoj strukturi

Prednosti:
- prikazuje opšti tok operacija u programu

Nedostaci:
- različiti testeri bi prikazivali različite grafikone, što uvodi nedoslednost u testiranju
- zahteva temeljno poznavanje softvera, ne možemo testirati dok ne znamo najmanje detalje projekta, što je jako teško kod velikih programa



3. Modelovanje stanja promenljivih

Mehanički način za dobijanje FSM je da razmotrimo vrednosti promenljivih u programu. To se obično definiše prilikom dizajniranja programa. Prvi korak je da odredimo koje promenljive mogu učestvovati u stanjima, a zatim izabrati one relevantne za FSM, na primer globalne i klasne promenljive.

Na primer, klasa Shutter definiše 4 promenljive:lastdelimit, curWord, prevWord i delimits. Promenljiva delimits je definisana na nivou klase, ali ne treba da bude odabrana kao deo stanja.

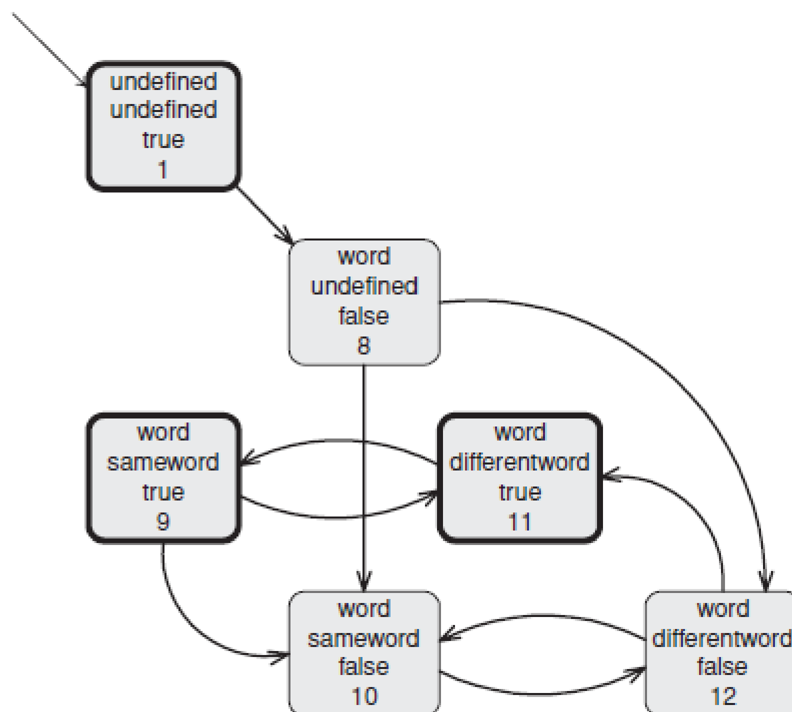
Teoretski svaka kombinacija ove 3 promenljive definiše različito stanje. U praksi međutim to može dovesti do beskonačnog broja stanja. Na primer curWord i prevWord su stringovi i imaju neograničen skup vrednosti. Dakle, uobičajno je da se identifikuju vrednosti ili opsezi vrednosti

koji će biti zastupljeni kao stanja. Za klasu Shutter, očigledno je da taj skup treba da bude zasnovan na odnosu između te dve promenljive, stvarajući sledeće moguće vrednosti:

curWord: undefined, word
prevWord: undefined, sameword, differentword
lastdelimit: true, false
gde su reči word, sameword, differentword tipa string.

Kombinacijom ovih vrednosti dolazimo do 12 mogućih kombinacija stanja:

1. (undefined, undefined, true)
2. (undefined, undefined, false)
3. (undefined, sameword, true)
4. (undefined, sameword, false)
5. (undefined, differentword, true)
6. (undefined, differentword, false)
7. (word, undefined, true)
8. (word, undefined, false)
9. (word, sameword, true)
10. (word, sameword, false)
11. (word, differentword, true)
12. (word, differentword, false)

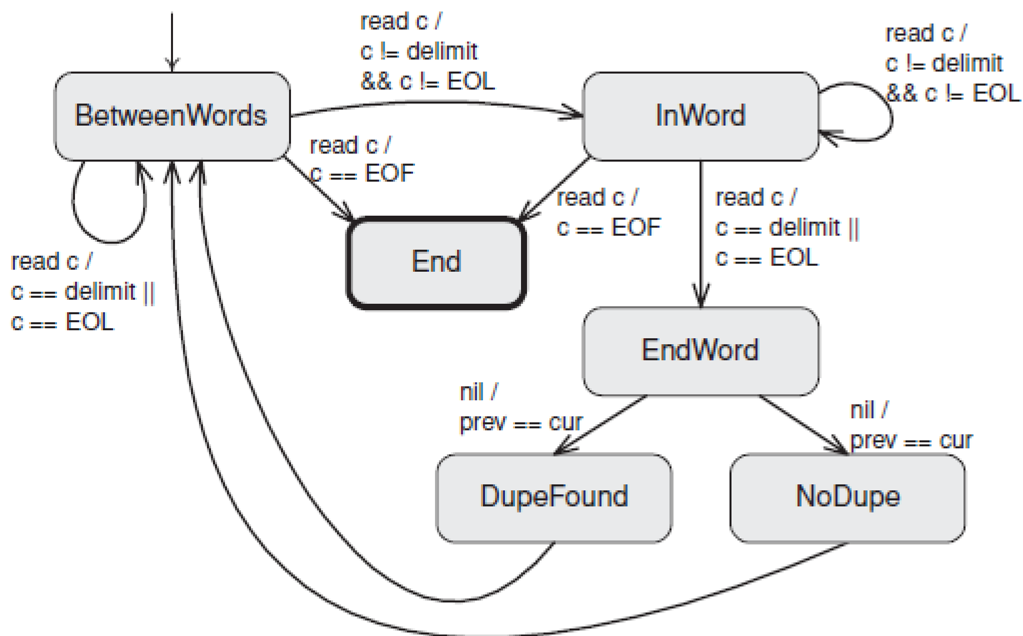


Naravno, nije svaka od ovih kombinacija moguća. Na primer curWord dobija odmah vrednost i nikad ne postaje nedefinisana nakon toga. Dakle, jedino moguće stanje u kome promenljiva curWord ima vrednost je stanje 1, koje ćemo proglasiti za početno stanje. Čim se pročita neki karakter, curWord ima vrednost neke reči, a lastdelimit je podešen na false (stanje 8). Kada je delimiter pronađen, prevWord ima vrednost iste reči ili različite reči, u odnosu na curWord (stanja 10 i 12). Svako stanje gde je lastdelimit = true, može biti konačno stanje (nacrtano podebljanom linijom).

Stanje 7 je izostavljeno, jer je nemoguće doći do njega. Takođe, treba znati da se ovaj program završava na kraju pročitanog fajla.

Mehanički proces u ovoj strategiji je privlačan zato što možemo očekivati da različiti testeri izvedu isti ili sličan FSM. Taj proces nije uvek moguće da se potpuno automatizuje, zato što postoji teškoća za utvrđivanje prelaza iz odredišta i zato što odluka o bilo kojoj varijabli zahteva presudu.

4. Poslednja metoda za dobijanje FSM oslanja se na eksplicitinim zahtevima ili formalnoj specifikaciji koja opisuje ponašanje tog softvera.



FSM na osnovu koda su dosta laki za razumevanje. Ako je softver dobro dizajniran, ovaj tip FSM treba da sadrži iste informacije koje čine UML dijagram stanja.

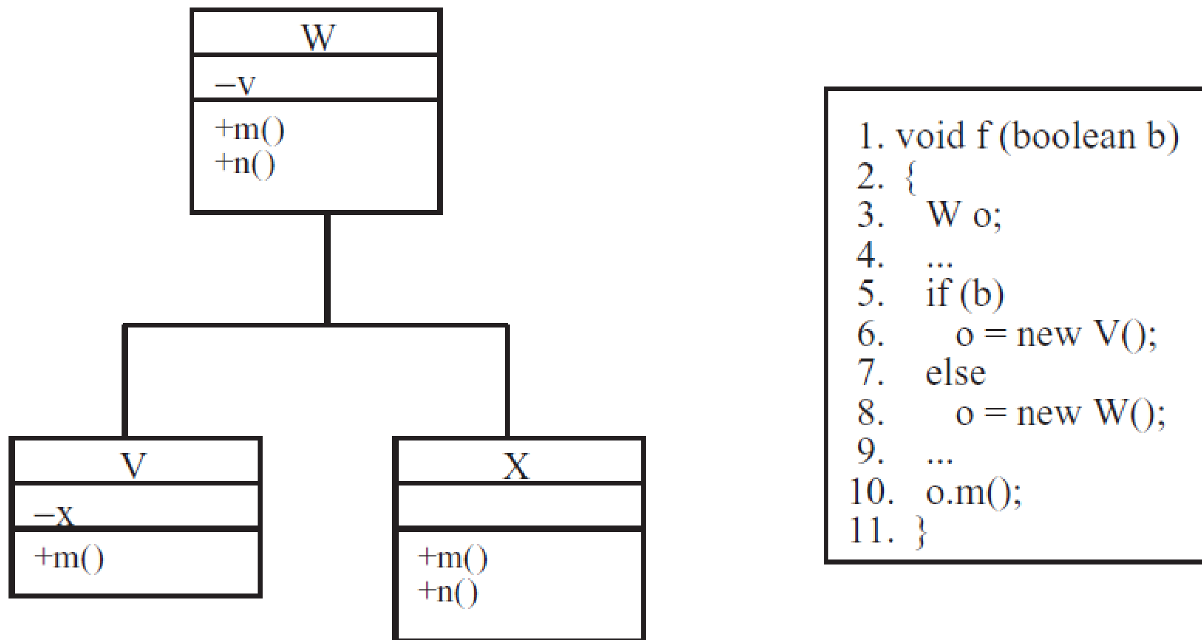
Teorija

Kada se testira objektno-orijentisani softver, klasa se obično predstavlja osnovnu jedinicu testiranja. To dovodi do 4 nivoa testiranja klasa:

1. Intra-metod testiranje: testovi su izgrađeni za pojedinačne metode (ovo je tradicionalno jedinično testiranje)
2. Inter-metod testiranje: višestruke metode u okviru klase su testirane zajedno (ovo je tradicionalno testiranje modula)
3. Intra-klasno testiranje: testovi su izgrađeni za pojedinačne klase, obično kao sekvenca poziva metoda unutar klase
4. Inter-klasno testiranje: više od jedne klase je testirano u isto vreme, obično se vidi kako su u interakciji (ovo je vrsta integracionog testiranja)

Ranija istraživanja u objektno orijentisanom testiranju bila su fokusirana na inter-metod i intra-klasnom nivou. Kasnija istraživanja usmerena su na ispitivanje interakcije između pojedinačnih klasa i njihovih korisnika i na testiranju OO softvera na sistemskom nivou. Problemi vezani za nasleđivanje i polimorfizme se ne mogu rešiti na nivou inter-metode ili intra-klase. U tom slučaju treba testirati više klasa koje se kombinuju kroz nasleđivanje i polimorfizam. Većina istraživanja u OO testiranju je bazirana na dva problema. Jedan je poredak u kome treba klase da budu integrisane i testirane, a drugi je da se razviju tehnike i kriterijumi pokrivenosti koda za odabrane testove. Jedan od najtežih zadataka za OO softverske inženjere predstavlja prikazivanje interakcije koje se javljaju uz prisustvo nasleđivanja, polimorfizma i dinamičkog. To je često veoma složeno. Ova vizuelizacija podrazumeva da klasa enkapsulira informaciju o stanju u kolekciji stanja promenljivih i ima skup osobina koje se realizuje metodama, koje koriste te promenljive.

Primer



Razmatramo sledeći UML klasni dijagram. V i X su izvedene klase iz klase W, klasa V preklapa metodu m(), a klasa X prekla metode m() i n(). Minus (-) označava atribut sa privatnim pravom pristupa, a plus (+) označava javno pravo pristupa. Deklarisani tip objekta o je W (linija koda 3), ali u liniji 10 stvarni tip može biti V ili W.

Koja preklopljena metoda m() će se izvršiti zavisi od argumenta metode f - boolean promenljive b. U sledećem zadatku biće ilustrovan problem preklapanja i polimorfizma.

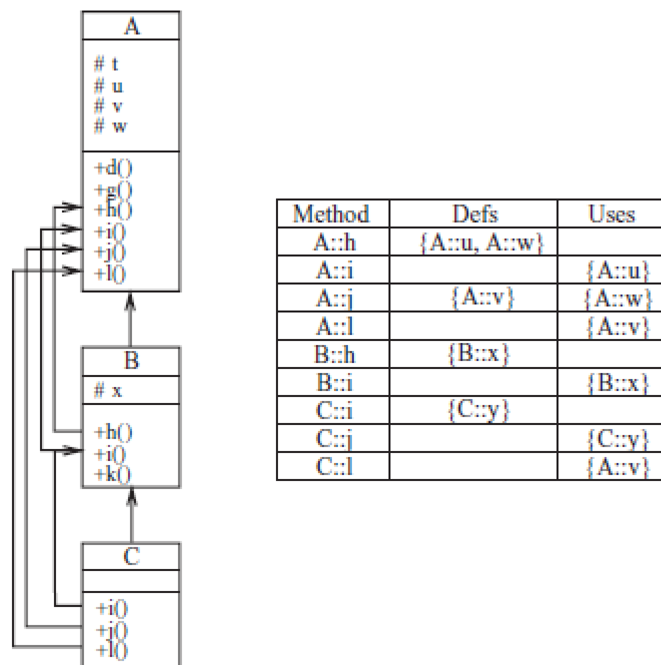
Zadatak 2

Neka je data sledeća struktura programa sa 3 klase: A, B i C. Promenljive su označene sa # i označavaju zaštićeno (*protected*) pravo pristupa, a date su i metode sa javnim (*public*, zato što su označene sa +) pravom pristupa. Korena klasa A sadrži stanja za 4 promenljive i 6 metoda. Promenljive su zaštićene što znači da su dostupne u klasama potomcima klase A, u klasama B i C. Klasa B deklariše stanje 1 promenljive i 3 metode, a klasa C samo 3 metode. Strelice na slici označavaju pokazuju važnost metoda: metoda B::h() preklapa (nadjačava) metodu A::h(), metoda B::i() preklapa metodu A::i(), a metoda C::i() preklapa metodu B::i(), C::j() preklapa A::j(), i C::l() preklapa A::l(). Tabela pored dijagrama prikazuje stanje definisanih promenljivih i u kojim metodama se koriste te promenljive.

U implementaciji klase A, pretpostavimo da metoda d() poziva metodu g(), g() poziva h(), h() poziva i() i i() poziva j(). Dalje pretpostavimo da kod implementacije B, metoda h() poziva

metodu i(), i() poziva metodu i() iz svoje roditeljske klase (A), a k() poziva l(). Konačno, kod C imamo metodu i() koja poziva roditeljsku metodu i() iz klase B, i metoda j() koja poziva k().

Nacratati yo-yo graf za dati primer.



Method	Defs	Uses
A::h	{A::u, A::w}	
A::i		{A::u}
A::j	{A::v}	{A::w}
A::l		{A::v}
B::h	{B::x}	
B::i		{B::x}
C::i	{C::y}	
C::j		{C::y}
C::l		{A::v}

Rešenje:

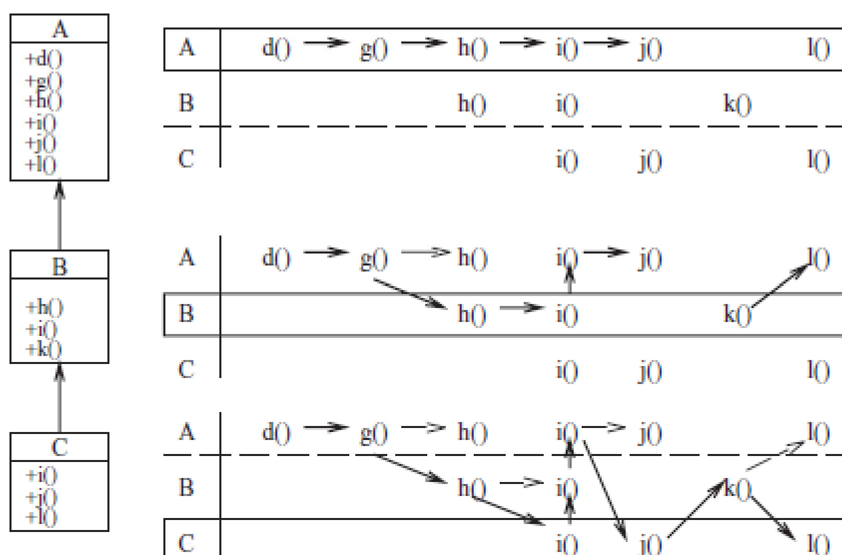
Razumevanje koja metoda će se izvršiti je vrlo teško i za programere i testere, jer moraju pratiti sve pozive i sve nivoe nasleđivanja. Izvršavanje metoda, kao što u ovom slučaju jedna poziva drugu, druga treću,… može se stalno skakati između nivoa nasleđivanja i taj efekat se naziva yo-yo efekat.

Yo-yo graf definiše hijerarhiju nasleđivanja, ima koren i potomke. Grafik prikazuje sve nove, nasleđene i poništene metode za svakog od potomaka. Pozivanje metoda predstavlja se strelicama. Strelica ide od metode iz koje se poziva ka metodi koja je pozvana. Svaka klasa ima svoj nivou u yo-yo grafu, koji prikazuju stvarne pozive koji su napravljeni ukoliko je kreiran objekat tipa tog nivoa. Podebljane strelice su stvarni pozivi, a svetle strelice su pozivi koji ne mogu biti napravljeni zbog preklapanja.

Problem počinje sa pozivom A::d(). Na slici je prikazano šta se dešava kada je tip objekta A, B i C, kada pozivamo tu metodu. U prvom slučaju, pretpostavljamo da je metoda pozvana nad objektom tipa A i to je prikazano u prvom nivou grafikona. Ovaj redosled poziva je jednostavan i jasan.

Drugi nivo pokazuje složeniju situaciju, kada je objekat tipa B. Metoda g() poziva metodu h(), ali izvršava se verzija h() koja je definisana u B (svetla linija od A::g() ka A::h() pokazuje da se A::h() NE IZVRŠAVA!). Zatim se kontrola nastavlja pozivanjem B::i(), A::i() i A::j().

Kada formiramo objekat tipa C, možemo videti odakle dolazi termin "yo-yo". Kontrola nakon pozivanja A::g() prelazi na B::h(), pa na C::i(), zatim se vraća preko B::i() i A::i() nazad pozivanjem C::j(), ponovo se ide u B nivo i poziva B::k(), a na kraju C::l().



Primer primene metoda klase ekvivalencije na intraklasno testiranje

Metod može implementirati jednu ili više funkcionalnosti, tako da svaka prezentuje različite nivoe kohezije. Čak kohezivne funkcije mogu imati složene ulazno/izlazne odnose. Potrebno je naći neku sistemsku tehniku koja bi identifikovala funkcije i testirala svaki ulano/izlazni odnos.

Uzorak podela na kategorije (eng. category partition pattern) je pogodan za bilo koju metodu koja sprovodi jednu ili više nezavisnih funkcija. Za metode i funkcije kojima nedostaje kohezija, sastavne obaveze treba da budu identifikovanje i modelovanje kao zasebnih funkcija. Ovaj pristup je kvalitetan i može se raditi ručno.

Ovaj uzorak pretpostavlja da se greške odnose na vrednosti kombinacija parametara poruke i instance promenljive, a ove greške će dovesti do nedostatka ili netačne metode za izlaz. Istraživači Offutt i Irvine su utvrdili da ovim pristupom mogu da se otkriju 23 uobičajne greške kodiranja u C++. Međutim, greške

koje se manifestuju samo pod određenim sekvencama, ili čije korumpirane instance promenljivih su sakrivene u MUT (method under test) interfejsu, ne mogu biti otkrivene ovim metodom.

Ovaj primer ilustrovaćemo jednom C++ funkcijom List::getNextElement(). Ta funkcija vraća sledeći element objekta List. Ako je tekuća pozicija u listi od m elemenata n-ti element, uzastopni pozivi getNextElement() vratići elemente na pozicijama n+1, n+2, n+3, itd. Ako nema pozicije koja se dobija pozivanjem prethodne funkcije ili prethodna pozicija više ne postoji zbog neke promene ili brisanja, baca se izuzetak NoPosition. EmptyList izuzetak se baca ako je lista prazna.

Procedura testiranja:

1) Identifikujemo funkcije koje testiramo - Dobro dizajnirane metode implementiraju samo nekoliko kohezivnih funkcija, koje se obično razlikuju u imenu i parametrima. Jedan metod može implementirati više funkcija. Druge funkcije mogu biti bočni efekti primarne funkcije. Funkcija može samostalno da se testira ako zadovoljava tri kriterijuma:
    - Ona može biti određena postavljanjem parametara na određene vrednosti
    - Izlaz se posmatra kao vrednost koju vraća metod
    - Izlaz može biti drugačiji od izlaza druge funkcije ili nekorektna funkcija

2) Identifikovati ulazne i izlazne parametre svake funkcije koja se testira
3) Identifikovati kategoriju svakog ulaznog parametra

   Kategorije za metodu getNextElement()

Parametar	Kategorija
Pozicija poslednjeg referenciranog elementa	n-ti element
	specijalni slučaj
Stanje liste	- element
	specijalni slučaj

4) Svaku kategoriju podeliti u particije prema različitim vrednostima

Parametar	Kategorija	Vrednost
Pozicija poslednjeg referenciranog elementa	n-ti element	n = 2
		n = neko x > 2, x < Max
		n = Max
	specijalni slučaj	nedefinisana
		prva
		poslednja, n < Max
Stanje liste	- element	m = neko x > 2, x < Max
	specijalni slučaj	prazna lista
		polupopunjena lista
		popunjena lista (m = Max)

5) Identifikovati ograničenja kod nekih vrednosti

6) Generisati test primere sa svim mogućim kombinacijama

7) Proračunati očekivani rezultat za svaki test primer:

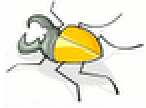
CATEGORY-PARTITION TEST DESIGN PATTERN

Test case	Function Parameters/Choices		Expected Result		
	position of the last referenced element	state of the list	Returned	Exception	Position of the last referenced element
1	undefined	empty	null	noPosition	undefined
2	undefined	singleton	null	noPosition	undefined
3	undefined	m= rand(x)	null	noPosition	undefined
4	undefined	full	null	noPosition	undefined
5	first	empty	null	listEmpty	undefined
6	first	singleton	first		first
7	first	m= rand(x)	second element		second element
8	first	full	second element		second element
9	n = 2	empty	null	listEmpty	undefined
10	n = 2	singleton	null	noPosition	undefined
11	n = 2	m= rand(x)	element $n + 1$		element $n + 1$
12	n = 2	full	element $n + 1$		element $n + 1$
13	n = rand(x)	empty	null	listEmpty	undefined
14	n = rand(x)	singleton	null	noPosition	undefined
15	n = rand(x)	m= rand(x)	element $n + 1$		element $n + 1$
16	n = rand(x)	full	element $n + 1$		element $n + 1$
17	Max	empty	null	listEmpty	undefined
18	Max	singleton	null	noPosition	undefined
19	Max	m= rand(x)	element $n + 1$		element $n + 1$
20	Max	full	element $n + 1$		element $n + 1$
21	Last	empty	null	listEmpty	undefined
22	Last	singleton	null	noPosition	undefined
23	Last	m= rand(x)	first element		first element
24	Last	full	first element		first element

Rand objektno orijentisano testiranje

Teorijske osnove

Objektno orijentisano jedinično testiranje (eng. *object-oriented unit test*) sastoji se od sekvence poziva metoda koje postavljaju stanje (kreiranje objekata i promene vrednosti) i provere rezultata finalnog poziva.



Randoop - Random test generation
http://code.google.com/p/randoop/

Randoop je generator jediničnog testiranja za Javu. On automatski kreira jedinične testove u JUnit formatu. Ovo poglavlje pokriva testiranje primenom metode slučajnog izbora (eng. *random*), pri čemu koristi informacije prikupljene prilikom kreiranja prethodnih testova (eng. *feedback-directed*). Ova tehnika slučajno, ali pametno, generiše sekvence poziva metoda i poziva konstruktora za klase koje testiramo i koristi te sekvence za kreiranje testova.

Sekvenca metoda (engl. Method Sequence), ili *sekvenca* je niz poziva metoda. Svaki poziv sastoji se od imena metode i niza argumenata, *s.i* označavaće vrednost vraćenu od i-tog metoda u sekvenci s.

Produžavanje sekvenci (engl. Extending sequences), prihvata 0 ili više sekvenci i generiše novu sekvencu. Ova operacija kreira novu sekvencu nadovezujući ulazne sekvence dodavanjem jednog metoda na kraj. Možemo zapisati *extend(m, seqs, vals)*:

- **m** je metod sa formalnim parametrima T_1, ..., T_k
- **seqs** je lista sekvenci
- **vals** je lista vrednosti v_1:T_1, ... v_k:T_k. Svaka vrednost je primitivnog tipa ili vrednost vraćena nekim prethodnim pozivom u sekvenci.

Rezultat izvršavanja extend je nova sekvenca koja sadrži nadovezane sekvence u navedenom redosledu u seqs praćene pozivom metoda m(v_1,..., v_k).

Primer:

Elementi sekvence:

- s1.1 (b1)
- s2.1 (b2)
- s3.1 i s3.2 (a1 i b3)

Generisanje testova upotrebom informacija prikupljenih ranijim generisanjem

```
 8        List successSequences;
 9        List errorSequences;
10
11        while (timeLimitReached() == false) {
12
13            Method method    = randomPublicMethod(classes);
14            List   sequences = randomSequences(successSequences);
15            List   values    = randomValues(method, successSequences);
16
17            Sequence newSequence = extend(method, sequences, values);
18
19            if (successSequences.contains(newSequence) || errorSequences.contains(newSequence)) {
20                continue;
21            }
22
23            boolean violated = newSequence.execute(contracts);
24
25            if (violated) {
26                errorSequences.add(newSequence);
27            } else {
28                successSequences.add(newSequence);
29                newSequence.setExtensibleFlags(filters);
30            }
31        }
```

Sekvence se grade postupno (eng. *incrementally*) pri čemu algoritam započinje sa praznim skupovima. Nakon što je sekvenca zagrađena, ista se izvršava kako bi se izbeglo kreiranje redudantnih sekvenci, što je sprecificirano filterima (eng. *filters*) i ugovorima (eng. *contracts*).

Četiri ulaza u algoritam: lista klasa za koje je potrebno kreirati sekvence, lista ugovora, lista filtera i vreme na raspolaganju za računanje sekvenci.

randomPublicMethod()

- slučajan izbor jedne metode (public), jedna od raspoloživih u classes

randomSequences()

- izbor slučajnih sekvenci

randomValues()

- izbor vrednosti za argumente metode. Ukoliko je argument primitivnog tipa, vrši se izbor vrednosti iz konstantnog niza vrednosti (na primer: -1, 0, 1, ’a’, true, itd.).

- Ukoliko je argument složenog tipa, random se bira jedna od sledećih mogućnosti: vrednost koja je izgenerisana nekim od prethodnih poziva u postojećoj sekvenci, izborom sekvence koja se nalazi u skupu successSequnces njenim dodavanjem u skup sekvenci i upotrebom vrednosti ili null.

Nakon poziva ranije opisane operacije extend dobijamo novu sekvencu.

Linije 19-21 koda proveravaju da li je sekvenca već ranije izgenerisana.

Nakon što je kreirana jedinstvena sekvenca, poziva se execute metoda nad sekvencom koja izvršava metode. Nakon svakog poziva metode vrši se provera ugovora. Ugovor je opis karakteristika koje treba da važe na početku i na kraju svakog poziva (true/false).

Ugovori koji se proveravaju:

Metode	
Ugovor	**Opis**
izuzeci	metod ne baca NullPointerException, pri čemu su svi prosleđeni argumenti ne null
	metod ne baca AssertionError

Objekti	
Ugovor	**Opis**
equals	o.equals(o) vraća true
	o.equals(o) ne baca izuzetak
toString	o.toString() ne baca izuzetak

Ukoliko neki od ugovora nije ispoštovan, izgenerisana sekvenca se dodaje u errorSequences, inače u successSequences. Ako izgenerisana sekvenca pripada skupu successSequences, nju je potrebno filtrirati kako bi se utvrdilo da li vrednost koju vraća metoda s.i može biti korišćena kao ulaz za neki naredni poziv.

Jednakost (engl. *Equality*) - ovaj metod koristi metod equals() za proveru da li je rezultujući objekat kreiran ranije ili ne. Filter održava listu svih kreiranih objekata. Ukoliko je ova vrednost viđena ranije novodobijena vrednost će biti filtrirana.

Null - ukoliko je rezultat izvršavanja null, njegova vrednost dalje neće biti korišćena.

Izuzeci (engl. *Exceptions*) - ograničavaju sekvencu; ukoliko se u poslednjem metodu trenutne sekvence generiše izuzetak, tu sekvencu nema smisla dalje produžavati.

Primer (nastavak):

Sekvenca s1	Sekvenca s2	Sekvenca s3
`B b1 = new B(0);`	`B b2 = new B(0);`	`A a1 = new A();` `B b3 = a1.m1(a1);`

seqs	vals	extend(m2, seqs, vals)

s_1, s_3	s_1.1, s_1.1, s_3.1	`B b1 = new B(0);`
		`A a1 = new A();`
		`B b3 = a1.m1(a1);`
		`b1.m2(b1, a1);`

s_3, s_1	s_1.1, s_1.1, s_3.1	`A a1 = new A();`
		`B b3 = a1.m1(a1);`
		`B b1 = new B(0);`
		`b1.m2(b1, a1);`

s_1, s_2	s_1.1, s_2.1, null	`B b1 = new B(0);`
		`B b2 = new B(0);`
		`b1.m2(b2, null);`

Testiranje Web aplikacija

Testiranje Web aplikacija (Web sajta) obuhvata proveru opterećenja, performansi, sigurnosti, funkcionalnosti, testiranje interfejsa, kompatibilnosti, i druga pitanja u vezi upotrebljivosti.

Testiranje Web aplikacija obuhvata tri nivoa (sloja) testiranja:

- Testiranje Web sloja
- Testiranje srednjeg sloja
- Testiranje sloja baze podataka

Kod testiranja prvog Web sloja ispituje se kompatibilnost aplikacije u Web pregledaču (engl. browser). Aplikacija treba da bude testirana u svim mogućim verzijama pregledača koje najčešće koriste krajnji korisnici - Mozilla Firefox, Google Chrome, IE (više verzija, zbog različitog ponašanja), Opera, Safari,... Kod testiranja srednjeg sloja ispituju se funkcionalnosti i sigurnosna pitanja. Kod sloja baze podataka, testiraju se i verifikuju se integritet baze podataka i sadržaj baze podataka.

Testiranje interfejsa

Testiranje interfejsa se radi da proveri pravilno funkcionisanje interfejsa aplikacije, prema datoj specifikaciji. Testiranje interfejsa se uglavnom koristi u testiranju korisničkih GUI aplikacija.

Alfa, beta i gama testiranje

Alfa testiranje je uglavnom prvo testiranje, koje prikazuje upotrebljivosti, obavlja se najčešće „in-house“, od strane programera koji su razvijali program ili od strane testera u toj kompaniji koja je razvijala. Ponekad se ovo alfa testiranje vrši od strane klijenata ili osobe sa strane, ali uz prisustvo programera koji je razvijao ili testera koji je već upoznat. Verzija nakon ovog testiranja, zove se alfa verzija programa.

Beta testiranje vrši ograničen broj krajnjih korisnika pre isporuke, a zahtevana promena će biti izmenjena od strane isporučilaca samo ukoliko korisnik daje povratne informacije o programu ili pripremi izveštaj o defektima (bagovima). Verzija nakon beta testiranja naziva se beta izdanje programa.

Gama testiranje je urađeno kada je softver spreman za puštanje u produkciju sa definisanim zahtevima. Utvrđeni defekti iz prethodnih faza moraju biti uklonjeni, ali mora biti utvrđeno i da krajnji korisnici nemaju žalbe na pojavu drugih defekata.