



Miodrag Živković

TESTIRANJE SOFTVERA

UNIVERZITET SINGIDUNUM

Miodrag Živković

TESTIRANJE SOFTVERA

Prvo izdanje

Beograd, 2018.

TESTIRANJE SOFTVERA

Autor:

dr Miodrag Živković

Recenzenti:

dr Mladen Veinović

dr Nebojša Bačanin Džakula

dr Marko Beko

Izdavač:

UNIVERZITET SINGIDUNUM

Beograd, Danijelova 32

www.singidunum.ac.rs

Za izdavača:

dr Milovan Stanišić

Priprema za štampu:

Jelena Petrović

Dizajn korica:

Aleksandar Mihajlović

Tiraž:

300 primeraka

Štampa:

Mobid, Loznica

ISBN: 978-86-7912-680-1

Copyright:

© 2018. Univerzitet Singidunum

Izdavač zadržava sva prava.

Reprodukcija pojedinih delova ili celine ove publikacije nije dozvoljena.

SADRŽAJ

Lista slika	VII
Predgovor	XIII
1. Uvod.....	1
1.1. Istorijat – čuveni primeri softverskih otkaza	1
1.2. Komercijalni softver.....	3
1.3. Osnovni pojmovi	5
1.4. Osnovna klasifikacija testiranja.....	7
1.5. Pojam softver testera	8
1.6. Osnovni proces testiranja	9
1.7. Osnovni principi testiranja	10
Pitanja.....	11
2. JUnit framework	13
2.1. Osnove JUnit alata.....	13
2.2. Klasa Assert.....	19
2.3. @ Anotacije.....	21
2.4. Parametrizovani testovi	28
2.5. Kolekcija testova	31
2.6. TestRunner klasa	32
Pitanja.....	33
3. Testiranje metodama crne kutije	35
3.1. Klase ekvivalencije.....	36
3.1.1. Višedimenzionalno particionisanje	38
3.2. Analiza graničnih vrednosti.....	40
3.3. Tabela odlučivanja i uzročno-posledični graf	43
3.4. Model stanja	50
3.5. Nagađanje grešaka.....	55
3.6. Zadaci za vežbu	56
Pitanja.....	60

4.	Testiranje metodama bele kutije.....	61
4.1.	Graf toka kontrole	62
4.2.	Pokrivanje iskaza.....	64
4.3.	Pokrivanje odluka.....	66
4.4.	Pokrivanje uslova	68
4.4.1.	Pokrivanje uslova i odluka	68
4.4.2.	Pokrivanje višestrukih uslova.....	69
4.4.3.	Minimalno pokrivanje višestrukih uslova	69
4.5.	Pokrivanje putanja.....	70
4.6.	Testiranje petlji.....	73
4.7.	Testiranje metodama toka podataka	75
4.8.	Zadaci za vežbu	83
	Pitanja.....	98
5.	Integraciono testiranje	99
5.1.	Stabovi i drajveri	100
5.2.	Big bang integracija.....	103
5.3.	Inkrementalna integracija	104
5.3.1.	Integracija od vrha ka dnu	105
5.3.2.	Integracija od dna ka vrhu	109
5.3.3.	Sendvič integracija	111
5.4.	Integracija po grafu poziva.....	113
5.4.1.	Integracija prema parovima	114
5.4.2.	Integracija prema susedstvu	114
5.5.	Zadaci za vežbu	115
	Pitanja.....	133
6.	Viši nivoi testiranja.....	135
6.1.	Regresiono testiranje	135
6.1.1.	Tehnike regresionog testiranja	136
6.1.2.	Odabir testova za regresiono testiranje.....	137
6.1.3.	Automatizacija regresionog testiranja	137
6.1.4.	Razlika između regresionog testiranja i retesta	138
6.1.5.	Prednosti i mane regresionog testiranja.....	138
6.2.	Smoke test	139
6.3.	Sanity test	140
	Pitanja.....	142
7.	Sistemsko testiranje.....	143
7.1.	Testiranje performansi sistema.....	145

7.1.1. Najčešći problemi	146
7.1.2. Proces testiranja performansi	146
7.1.3. Praćenje parametara	147
7.2. Testiranje opterećenja sistema.....	148
7.3. Stres test	150
7.4. Regulatorno testiranje.....	151
7.5. Test prihvatanja od strane korisnika (UAT)	152
7.5.1. Alfa testiranje	153
7.5.2. Beta testiranje	154
Pitanja.....	155
8. Statičko testiranje.....	157
8.1. Neformalni pregled.....	158
8.2. Formalni pregled	158
8.2.1. Planiranje.....	159
8.2.2. Kick-off	159
8.2.3. Priprema	159
8.2.4. Formalni sastanak.....	160
8.2.5. Ispravka i follow up.....	161
8.2.6. Uloge i odgovornosti učesnika u formalnom pregledu	161
8.3. Walkthrough.....	162
8.4. Tehnički pregled.....	162
8.5. Inspekcija.....	163
8.6. Alati za statičku analizu	163
Pitanja.....	165
9. Modeli procesa razvoja softvera	167
9.1. Faze u modelu razvoja softvera.....	167
9.1.1. Prikupljanje zahteva	168
9.1.2. Dizajn	168
9.1.3. Implementacija	168
9.1.4. Testiranje	169
9.1.5. Isporuka softvera klijentu.....	169
9.1.6. Održavanje.....	169
9.2. Tradicionalni modeli razvoja softvera.....	169
9.2.1. Waterfall model	170
9.2.2. V model	171
9.2.3. Inkrementalni model.....	173
9.2.4. RAD model.....	175
9.2.5. Iterativni model	176
9.2.6. Spiralni model	177

9.2.7. Prototip model	179
9.3. Agilni model.....	180
9.3.1. Prednosti agilnog modela	184
9.3.2. Agilni manifest	185
9.3.3. Najpopularniji agilni pristupi	186
Pitanja.....	188
10. Testiranje korisničkih interfejsa	189
10.1. Proces testiranja.....	190
10.1.1. Manuelno testiranje	191
10.1.2. Automatsko testiranje.....	194
10.1.3. Model stanja	196
10.2. Izazovi u testiranju grafičkog interfejsa veb aplikacija	197
10.3. Selenium alat za testiranje korisničkog interfejsa	201
10.3.1. Selenium IDE	202
10.3.2. Selenium WebDriver	208
10.4. Zadaci za vežbu	215
Pitanja.....	221
11. Testiranje objektno orijentisanog softvera	223
11.1. Jedinstveni problemi u testiranju OOP	223
11.2. Jo-jo graf.....	226
11.3. Problemi u nasleđivanju i polimorfizmu	228
11.4. Kompleksnost objektno orijentisanog softvera	231
Pitanja.....	233
12. Menadžment defekata	235
12.1. Izveštaj o defektu.....	235
12.2. Klasifikacija defekata	237
12.3. Izveštaj o testiranju.....	239
12.4. Životni ciklus defekta	242
Pitanja.....	244
Literatura.....	245

LISTA SLIKA

Slika 1.1. Eksplozija Ariane 5	2
Slika 1.2. Veličina koda nekih karakterističnih softverskih sistema	4
Slika 1.3. Prvi bug - moljac prikačen u dnevnik	5
Slika 2.1. Hijerarhija klasa programa u NetBeans projektu	14
Slika 2.2. Kreiranje testova u NetBeans projektu.....	14
Slika 2.3. Dijalog sa dodatnim opcijama za kreiranje test klase	15
Slika 2.4. Test klasa unutar sekcije Test Packages.....	15
Slika 2.5. Automatski generisana test klasa	16
Slika 2.6. Pokretanje izvršavanja testova	18
Slika 2.7. Test Results prozor okruženja NetBeans.....	18
Slika 2.8. Test Results prozor u slučaju kada postoje greške	19
Slika 2.9. Primer upotrebe parametra timeout.....	22
Slika 2.10. Rezultat izvršavanja testova za klasu MatematickeOperacije.....	28
Slika 2.11. Primer izvršavanja parametrizovanih testova.....	31
Slika 3.1. Princip testiranja metodama crne kutije	35
Slika 3.2. Određivanje klasa ekvivalencije za slučaj dozvoljenog opsega vrednosti	37
Slika 3.3. Određivanje klasa ekvivalencije za primer računanja kamatne stope	37
Slika 3.4. Grafički prikaz jednodimenzionalnog particionisanja	39
Slika 3.5. Grafički prikaz višedimenzionalnog particionisanja.....	39
Slika 3.6. Određivanje graničnih vrednosti	40
Slika 3.7. Granične vrednosti za alfanumeričku lozinku dozvoljene dužine 6-10 karaktera	42
Slika 3.8. Granične vrednosti za pojedinačne delove datuma	43
Slika 3.9. Osnovne logičke relacije u uzročno-posledičnom grafu	45
Slika 3.10. Ograničenja u uzročno-posledičnom grafu	46
Slika 3.11. Uzroci i posledice.....	46
Slika 3.12. Uzročno-posledični graf.....	47
Slika 3.13. Tabela odlučivanja	49
Slika 3.14. Konkretni testovi izvedeni na osnovu tabele odlučivanja	49
Slika 3.15. Tranzicija između dva stanja.....	50

Slika 3.16. Dijagram stanja za unos PIN koda na bankomatu.....	51
Slika 3.17. Dijagram stanja za elektronsku kupovinu iz perspektive klijenta	52
Slika 3.18. Dijagram stanja za elektronsku kupovinu sa označenim stanjima i prelazima koje pokriva test 1.....	54
Slika 3.19. Klase ekvivalencije za broj karata koje jedan korisnik može da rezerviše	56
Slika 3.20. Uzroci i posledice za bazu podataka banke.....	58
Slika 3.21. Uzročno-posledični graf za bazu podataka banke	59
Slika 3.22. Tabela odlučivanja za bazu podataka banke	59
Slika 4.1. Osnovne kontrolne strukture u programu.....	63
Slika 4.2. Primer grafa toka kontrole.....	64
Slika 4.3. Graf toka kontrole u slučaju kada se ne pokriva jedna grana odluke	67
Slika 4.4. Pokrivanje višestrukih uslova, primer nemoguće kombinacije uslova	69
Slika 4.5. Minimalno pokrivanje višestrukih uslova	70
Slika 4.6. Graf toka kontrole i sve putanje kroz program.....	71
Slika 4.7. Tabelarni prikaz matrice incidencije za graf toka kontrole sa slike 4.6 ...	72
Slika 4.8. Graf toka kontrole proste petlje.....	73
Slika 4.9. Nadovezana i ugnježdjena petlja	74
Slika 4.10. Graf toka kontrole	76
Slika 4.11. Definicije i upotrebe podataka za dati primer	77
Slika 4.12. Putanje u grafu toka kontrole	77
Slika 4.13. Primer određivanja različitih kriterijuma pokrivenosti	78
Slika 4.14. Definicije i upotrebe promenljivih iz metode Reši_Kvadratnu	80
Slika 4.15. DU parovi za metodu Reši_Kvadratnu	81
Slika 4.16. Pokrivanje svih definicija za metodu Reši_Kvadratnu	81
Slika 4.17. Pokrivanje svih DU parova za metodu Reši_Kvadratnu.....	81
Slika 4.18. Pokrivanje svih p-upotreba za metodu Reši_Kvadratnu	82
Slika 4.19. Pokrivanje svih c-upotreba za metodu Reši_Kvadratnu	82
Slika 4.20. Graf toka kontrole za primer 1	84
Slika 4.21. Graf toka kontrole za metodu izracunavanjeRegistracije.....	86
Slika 4.22. Testovi koji obezbeđuju pokrivenost iskaza metode izracunavanjeRegistracije.....	86
Slika 4.23. Testovi koji obezbeđuju pokrivenost odluka metode izracunavanjeRegistracije.....	87

Slika 4.24. Testovi koji obezbeđuju pokrivanje uslova metode izracunavanjeRegistracije.....	88
Slika 4.25. Testovi koji obezbeđuju pokrivanje uslova i odluka metode izracunavanjeRegistracije.....	89
Slika 4.26. Testovi koji obezbeđuju pokrivanje putanja metode izracunavanjeRegistracije.....	90
Slika 4.27. Formiranje cene paketa na osnovu datih parametara	91
Slika 4.28. Graf toka kontrole za metodu izracunajCenuPaketa	93
Slika 5.1. Primer hijerarhije softverskog sistema	100
Slika 5.2. Modul u stvarnom okruženju i u test okruženju.....	101
Slika 5.3. Big bang integracija	103
Slika 5.4. Problemi koji obično nastaju prilikom big bang integracije	103
Slika 5.5. Primer softverskog sistema	104
Slika 5.6. Inkrementalna integracija.....	105
Slika 5.7. Smer integracije od vrha ka dnu.....	106
Slika 5.8. Nivoi hijerarhije softverskog sistema i redosled testiranja od vrha ka dnu	106
Slika 5.9. Integracija po dubini	107
Slika 5.10. Integracija po širini.....	108
Slika 5.11. Smer integracije od dna ka vrhu.....	109
Slika 5.12. Nivoi hijerarhije softverskog sistema i redosled testiranja od dna ka vrhu	110
Slika 5.13. Primer integracije od dna ka vrhu	111
Slika 5.14. Nivoi hijerarhije softverskog sistema i redosled testiranja u sendvič integraciji.....	112
Slika 5.15. Graf poziva sa primerima parova	114
Slika 5.16. Graf poziva sa obeleženim susedstvima za komponente 1, 2 i 3	115
Slika 5.17. Pregled komponenti kalendara	115
Slika 5.18. Funkcionalna dekompozicija programa calendar	116
Slika 5.19. Integracija od vrha ka dnu, prvi korak	116
Slika 5.20. Integracija od vrha ka dnu, pristup po širini.....	117
Slika 5.21. Integracija od dna ka vrhu, integracija počinje od komponenti najnižih nivoa	117
Slika 5.22. Integracija od dna ka vrhu, integrisanje jedne komponente	118

Slika 5.23. Sendvič integracija	118
Slika 5.24. Big beng integracija, sve komponente se spajaju odjedanput	119
Slika 5.25. Graf poziva za program calendar	119
Slika 5.26. Integracija prema parovima	120
Slika 5.27. Integracija prema susedstvu	120
Slika 5.28. Primer arhitekture softverskog sistema	121
Slika 5.29. Prvi korak integracije po širini	121
Slika 5.30. Drugi korak integracije po širini	122
Slika 5.31. Treći korak integracije po širini	122
Slika 5.32. Četvrti korak integracije po širini	122
Slika 5.33. Prvi korak integracije po dubini	123
Slika 5.34. Drugi korak integracije po dubini	123
Slika 5.35. Treći korak integracije po dubini	124
Slika 5.36. Četvrti korak integracije po dubini	124
Slika 5.37. Klasa InvoiceService u izvornom obliku	126
Slika 5.38. Klasa koja zavisi od resursa	126
Slika 5.39. Izvlačenje osnovne funkcionalnosti objekta u interfejs	127
Slika 5.40. Kreiranje stab klase koja implementira interfejs	127
Slika 6.1. Tehnike regresionog testiranja	136
Slika 6.2. Razlika između sanity i smoke testiranja	140
Slika 7.1. Hronološki redosled faza testiranja	143
Slika 7.2. Fokus testiranja u svakoj fazi	143
Slika 7.3. Različiti tipovi funkcionalnog testiranja	144
Slika 7.4. Metodologija procesa testiranja performansi	147
Slika 7.5. Poruka o nedostupnom servisu za vreme Amazon Prime Day 2018	149
Slika 7.6. Lokacije na kojima servis nije bio dostupan za vreme Amazon Prime Day 2018	149
Slika 7.7. Stres test Notepad aplikacije	150
Slika 7.8. Acceptance testiranje u V modelu	152
Slika 7.9. Alfa i beta testiranje kao deo UAT	153
Slika 8.1. Tipovi statičkog testiranja prema nivou formalnosti	157
Slika 8.2. Upotreba alata za statičku analizu koda od strane programera	163
Slika 9.1. Waterfall model razvoja softvera	170

Slika 9.2. V model razvoja softvera	172
Slika 9.3. Inkrementalno dodavanje modula	173
Slika 9.4. Inkrementalni model razvoja softvera.....	174
Slika 9.5. RAD model razvoja softvera.....	175
Slika 9.6. Iterativni model razvoja softvera.....	176
Slika 9.7. Iterativno poboljšavanje proizvoda	177
Slika 9.8. Spiralni model razvoja softvera.....	178
Slika 9.9. Prototip model razvoja softvera	179
Slika 9.10. Agilni model razvoja softvera	180
Slika 9.11. Primer projekta sa primenom Waterfall modela	181
Slika 9.12. Primer projekta sa primenom agilnom modela	182
Slika 9.13. Detaljna struktura agilnog modela	183
Slika 9.14. Projekat nakon prve tri iteracije	183
Slika 9.15. Scrum metodologija	186
Slika 9.16. Kanban tabla sa karticama.....	187
Slika 10.1. Elementi grafičkog korisničkog interfejsa	189
Slika 10.2. Primer korisničkog interfejsa veb aplikacije	190
Slika 10.3. Manuelno testiranje korisničkog interfejsa	192
Slika 10.4. Korisnički interfejs Eclipse okruženja	193
Slika 10.5. Automatsko testiranje korisničkog interfejsa	195
Slika 10.6. Testiranje korisničkog interfejsa modelom stanja.....	196
Slika 10.7. HTML 5 element kalendar prikazan u Chrome veb čitaču	198
Slika 10.8. HTML 5 element kalendar prikazan u Internet Explorer v11	198
Slika 10.9. Implementacija fallback metode za <video> element.....	199
Slika 10.10. Uslovni komentari za rukovanje različitim verzijama čitača	199
Slika 10.11. Provera graničnih vrednosti za korisnički unos	200
Slika 10.12. Struktura Selenium alata	201
Slika 10.13. Selenium IDE nakon instalacije u Firefox	203
Slika 10.14. Osnovni ekran Selenium IDE alata	203
Slika 10.15. Započinjanje snimanja testa u Selenium IDE alatu.....	204
Slika 10.16. Odlazak na veb stranicu koju je potrebno testirati	204
Slika 10.17. Dodavanje komandi iz konteksnog menija	205
Slika 10.18. Testiranje unosa neispravnog korisničkog imena i lozinke.....	205

Slika 10.19. Snimljene komande	206
Slika 10.20. Izvorni kod snimljene test skripte	206
Slika 10.21. Čuvanje generisanog testa	207
Slika 10.22. Sačuvani testovi u okviru Selenium IDE	207
Slika 10.23. Pokretanje snimljenog testa.....	208
Slika 10.24. Dodavanje novog Selenium testa u NetBeans okruženju.....	209
Slika 10.25. Šablon Selenium testa	210
Slika 10.26. Zadatak - forma koju je potrebno testirati	218
Slika 11.1. Primer klasnog dijagrama sa nasleđivanjem	224
Slika 11.2. Primer klasnog dijagrama sa anomalijama u toku podataka	225
Slika 11.3. Grafički prikaz nadjačanih metoda i redosleda pozivanja.....	226
Slika 11.4. Jo-jo graf	227
Slika 11.5. Klasa Stack izvedena iz klase Vector.....	228
Slika 11.6. Primer proširenog grafa toka kontrole.....	231
Slika 12.1. Novi izveštaj o bugu.....	236
Slika 12.2. Primer naslovne strane izveštaja o testiranju	239
Slika 12.3. Uvodni deo izveštaja, sa informacijama o testiranju.....	240
Slika 12.4. Primer obrasca za listu novih, aktivnih i poznatih defekata	241
Slika 12.5. Kompletan obrazac za izveštaj o testiranju	241
Slika 12.6. Životni ciklus defekta u alatu Bugzilla.....	242

PREDGOVOR

Testiranje softvera je integralni deo procesa razvoja softvera, sa ciljem osiguravanja kvaliteta finalnog proizvoda. U ljudskoj prirodi je da pravimo greške. Neke greške mogu biti sitne i nebitne, dok druge mogu biti veoma ozbiljne i skupe, pa čak i dovesti do ljudskih žrtava. Testiranje softvera je neophodno kako bi se otkrile greške načinjene u svim fazama razvoja softvera, koje se nakon toga mogu ispraviti. Otkrivanjem i ispravljanjem grešaka se obezbeđuje viši nivo kvaliteta softvera, čime se stiče poverenje i zadovoljstvo krajnjih korisnika softvera. U ovoj knjizi fokus je na tehnikama otkrivanja grešaka u softveru.

Obim i sadržaj ove knjige prilagođeni su nastavnom programu istoimenog predmeta na Fakultetu za informatiku i računarstvo Univerziteta Singidunum u Beogradu. Podrazumeva se da čitaocima knjige ovo nije prvi susret sa programiranjem u Javi, tj. očekuje se osnovno poznavanje programskog jezika Java pošto su svi praktični primeri implementirani u alatu JUnit.

Tekst je propraćen velikim brojem slika i detaljno objašnjenih primera pomoću kojih su ilustrovani novouvedeni pojmovi. Primeri su odabrani na takav način da budu što jednostavniji za razumevanje, ali sa druge strane i što realističniji i interesantniji, sa krajnjim ciljem da podstaknu čitaoce na dublje samostalno učenje i budu početna tačka za dalje eksperimentisanje. Iako je knjiga pisana kao udžbenik, ambicija autora jeste da ona posluži svima koji se prvi put susreću sa testiranjem softvera. Na kraju svakog poglavlja data su pitanja koja mogu pomoći u utvrđivanju gradiva.

Sve primedbe, komentari, preporuke, pohvale i eventualno uočene greške se mogu poslati na adresu mzivkovic@singidunum.ac.rs.

Miodrag Živković

Beograd, Srbija, 2018.

1. UVOD

Pojam testiranja softvera je star koliko i samo programiranje. Još od prvih napisanih programa je bilo neophodno proveriti da li se program ponaša ispravno i na takav način kako je definisano. U današnje vreme, iako možda to na prvi pogled ne izgleda tako, naš način života u potpunosti zavisi od softvera koji je ključni faktor u velikom boju sistema koje svakodnevno koristimo. Softver definiše ponašanje mrežnih rutera, bankarskih mreža, telefonskih sistema i samog Interneta. Dalje, softver je osnovna komponenta mnogih ugrađenih aplikacija koje kontrolišu rad veoma složenih sistema poput aviona, kontrole letenja, svemirskih brodova, a može se pronaći i u nešto prostijim uređajima (uslovno rečeno) poput automobila, mobilnih telefona, satova, DVD plejera, mikrotalasnih peći i druge bele tehnike.

Prema procenama, u modernom domaćinstvu postoji preko 50 procesora. Ako se posmatraju noviji modeli automobila, u svakom postoji preko 100 procesora. Na svakom od ovih procesora se izvršava softver, a optimistični krajnji korisnici smatraju da će sav taj softver da se izvršava bez grešaka. Postoje mnogi faktori koji mogu da utiču na kvalitet softvera i njegovu pouzdanost. Pažljiv dizajn i projektovanje softvera svakako mogu pomoći da softver bude pouzdan, ali osnovna metoda koja se u industriji koristi za evaluaciju softvera koji se razvija jeste testiranje softvera. Testiranje softvera, kao integralni deo razvoja softvera, ima za cilj da osigura da je kvalitet softvera na odgovarajućem nivou (engl. *Quality Assurance*). Na sreću, većina osnovnih koncepata testiranja softvera se može primeniti na veliki broj potpuno različitih tipova softvera različitih namena. Cilj ove knjige je da prikaže ove koncepte na takav način da čitaoci mogu lako da ih primene u bilo kojoj situaciji koja se može javiti u testiranju softvera.

U modernom razvoju softvera, zanimanje softver tester je veoma cenjeno i traženo. Međutim, neke ideje i tehnike testiranja softvera su postale neophodan skup veština koje mora da poseduje i dobar programer. Od svakog programera se očekuje da testira svoj kod, i to je standard u industriji, pa je ova knjiga namenjena i njima.

1.1. Istorijat – čuveni primeri softverskih otkaza

Značaj testiranja softvera je možda najlakše prikazati na konkretnim primerima softverskih otkaza, kada testiranje softvera nije izvršeno na adekvatan način i kada greške nisu otkrivene na vreme. U slučaju industrijskog softvera, ove greške često dovode do velikih finansijskih gubitaka, a ponekad, na žalost, i do ljudskih žrtava. Čak i u najpovoljnijem slučaju, greške koje su prisutne u finalnom isporučenom softveru dovode do smanjivanja zadovoljstva krajnjih korisnika, što kao posledicu može imati gubitak poverenja u dati sistem i okretanje alternativnim rešenjima.

U januaru 1990. godine, nijedan korisnik američke telekomunikacione kompanije AT&T nije mogao da uspostavi pozive na velikim rastojanjima, zbog softverske greške na relejnim svičevima. Neposredno pre nastanka problema, softver koji kontrolise relejne svičeve je bio ažuriran novom verzijom, bez adekvatnog testiranja da li nova verzija radi ispravno. Samo tog dana kompanija AT&T je izgubila preko 60 miliona \$ na troškove, penale i sudske procese.

NASA - Mars Climate Orbiter misija je doživela ogroman neuspeh 1999. godine. U okviru svoje misije na Marsu, zbog greške u softveru letelica je nepovratno izgubljena u svemiru. Nakon duže istrage koja je usledila, otkriveno je da je napravljena greška pri konverziji imperijalnih jedinica u metrički sistem. Jedan razvojni tim koji je radio na mlaznim motorima je računao potisak u imperijalnim jedinicama (funte), dok je drugi razvojni tim radio u metričkim jedinicama pri proračunima, a kao rezultat mlazni motori su imali 4.45 puta jači potisak od potrebnog. Pri ulasku u Marsovu orbitu, letelica od 125 miliona \$ je prišla previše blizu površini planete zbog prejakog potiska nakon pokušaja stabilizacije orbite, i usled dejstva Marsove atmosfere oštećeni su joj komunikacioni uređaji. Svaka dalja komunikacija sa letelicom je onemogućena, i sada je to veoma skupi i potpuno beskorisni komad metala u orbiti oko Sunca.

Evropska Svemirska Agencija je, poput NASA, takođe imala svoje trenutke, od kojih se izdvaja misija Ariane 5 let 501. Prethodni model Ariane 4 je imao više od 100 uspešnih poletanja. Šta bi uopšte moglo da pođe po zlu? Samo 40 sekundi nakon lansiranja 1996. godine, letelica vredna 370 miliona \$ se raspala zbog softverske greške. Problem je izazvalo prekoračenje u konverziji float -> int. Inercijalni referentni sistem je radio sa 64-bitnim podacima tipa float, koje je konvertovao u 16-bitni integer, pri čemu je izazvan aritmetički overflow. U razvoj Ariane 5 pre toga je uloženo 10 godina razvoja, oko 8 milijardi \$, a u trenutku poletanja je dodatno nosila 500 miliona \$ vredne satelite (među kojima veoma vredne satelite za ispitivanje solarnog zračenja i njegove interakcije sa magnetnim poljem Zemlje).



Slika 1.1. Eksplozija Ariane 5

EDS Child Support sistem je uveden 2004. godine u britanskoj Agenciji za podršku deci. U pitanju je bio izuzetno kompleksni IT sistem, a istovremeno je odeljenje za rad i penzije odlučilo da restrukturira softverski sistem cele agencije. Ova dva dela softvera su bila potpuno nekompatibilna, i doveli su do nepovratnih grešaka. Sistem je nekako uspeo da preplati 1.9 miliona ljudi, potplati dodatnih 700 000, dovede do 7 milijardi \$ nepodignutih dečijih dodataka i 36 000 slučajeva bespovratno zaglavljenih u sistem, i do danas je britanske poreske obveznike koštao više milijardi \$.

Greška u softveru je zamalo dovela do nuklearnog rata 1983. godine. To je bilo vreme Hladnog rata i veoma zategnutih odnosa između SAD i Sovjetskog Saveza. Nuklearni rat bi gotovo sigurno ugrozio opstanak celog čovečanstva. Sovjetski sistem za rano upozoravanje je greškom detektovao da su SAD lansirale pet balističkih projektila. Srećom, sovjetski oficir na dužnosti je na osnovu zdrave logike pretpostavio da bi Amerikanci u slučaju pravog napada lansirali mnogo više od 5 projektila – prijavio je lažni alarm umesto da inicira sovjetski odgovor. Sovjetski odgovor je podrazumevao protivudar i lansiranje projektila na ciljeve u Americi. Uzrok je bila greška u softveru, koji nije uspeo da isfiltrira lažne detekcije projektila izazvane refleksijom sunčanih zraka sa vrhova oblaka.

Na osnovu navedenih primera se vidi da postoji jasna potreba za kvalitetom softvera. Kako bi se obezbedio zadovoljavajući nivo kvaliteta softvera, postoje zahtevi za planiranim razvojem po fazama, dokumentacijom i pridržavanjem različitih standarda. Potrebno je ispuniti i nefunkcionalne zahteve, na primer sigurnost i performanse. U realnom razvoju softvera, čak do 50% uloženog vremena i troškova ide na testiranje softvera.

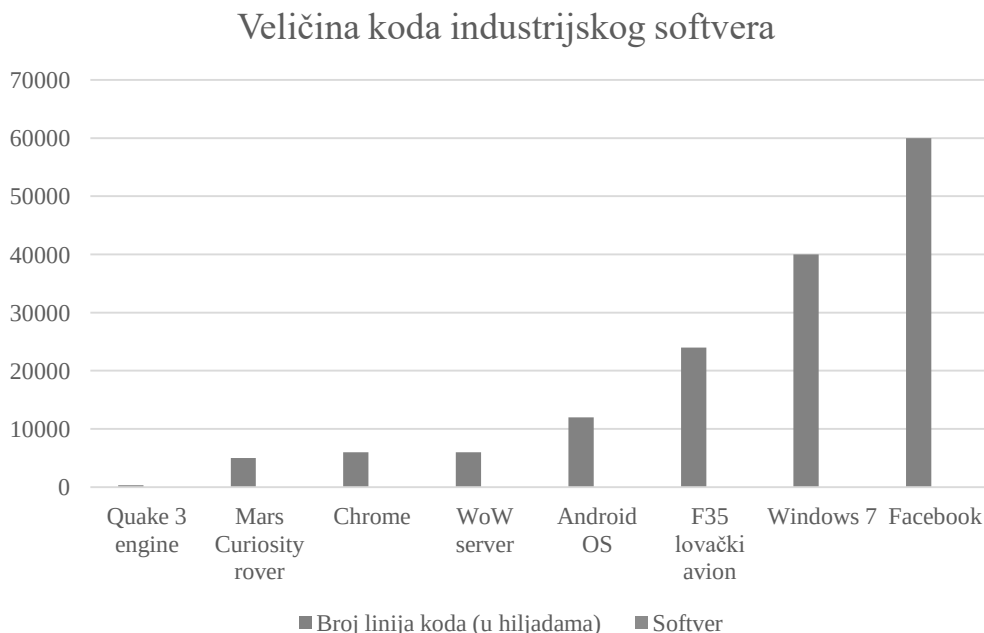
1.2. Komercijalni softver

Da bi se bolje razumela potrebu za testiranjem softvera, neophodno je pojasniti pojam komercijalnog softvera i razlike u odnosu na programe sa kojima studenti dolaze u kontakt za vreme studija. Programi koji se razvijaju u okviru nastave na fakultetima se mogu definisati kao softver za edukaciju. Takav softver se programira sa ciljem demonstracije na predavanjima i vežbama, zatim ispitnim zadacima, kao i u okviru samostalnog učenja. Ono što ga karakteriše jeste da ne rešava neki konkretan problem, već se osmišljava sa ciljem da se demonstrira ili vežba neka tehnika programiranja, a samo prisustvo grešaka (iako naravno nije poželjno) nije previše zabrinjavajuće. Za ovakve programe dokumentacija često nije od velike važnosti, pa se ni ne piše.

Komercijalni softver, sa druge strane, se razvija sa tačno definisanim ciljem, kako bi rešio neki konkretni problem krajnjih korisnika. Neophodno je da se program korektno izvršava – u suprotnom nije od koristi, pošto krajnji korisnici neće moći da ga koriste na predviđen način. Greške u radu programa mogu izazvati nezadovoljstvo korisnika, finansijske gubitke, gubitke značajnih podataka, čak i ljudske žrtve. Kako bi se obezbedilo korektno izvršavanje softvera, softver se razvija po fazama, sa jasnim zahtevima i dokumentacijom, i adekvatno testira.

Komercijalni softver može biti veoma različit – počev od desktop aplikacija, preko aplikacija za mobilne uređaje i veb aplikacija, pa sve do ugrađenih (engl. *embedded*) aplikacija koje kontrolišu rad različitih hardverskih sistema. Srećom, većina tehnika testiranja softvera su uopštene i mogu se primeniti na bilo koji tip softvera.

Veličina komercijalnog softvera može da varira drastično, a najčešće se meri u broju linija koda. Mali projekti su reda veličine 10-ak hiljada linija koda, dok veliki projekti mogu imati i više miliona linija koda. Na primer, proste aplikacije za mobilne telefone mogu biti veličine nekoliko hiljada linija koda, dok su prosečne aplikacije za iPhone reda veličine 50 hiljada linija koda. Veličina kompletnog koda koji se nalazi na Curiosity roveru na Marsu je reda veličine 5 miliona linija koda. Serverski kod popularne kompjuterske igre World of Warcraft je veličine 6.5 miliona linija koda, a najnovija verzija Google Chrome je oko 7 miliona linija koda. Kompletan operativni sistem Android je veličine oko 12 miliona linija koda. Prosečna veličina ukupnog softvera koji se nalazi u modernim automobilima iznosi preko 100 miliona linija koda. Karakteristične vrednosti veličine softvera su prikazane na slici 1.2.



Slika 1.2. Veličina koda nekih karakterističnih softverskih sistema

Testiranje komercijalnog softvera je neophodno. Prema procenama američkog nacionalnog instituta za standarde (NIST), softverski bugovi izazivaju gubitke u vrednosti od više desetina milijardi dolara godišnje samo u Sjedinjenim Američkim Državama. Uočeno je takođe da se većina otkaza hardvera i različite opreme može

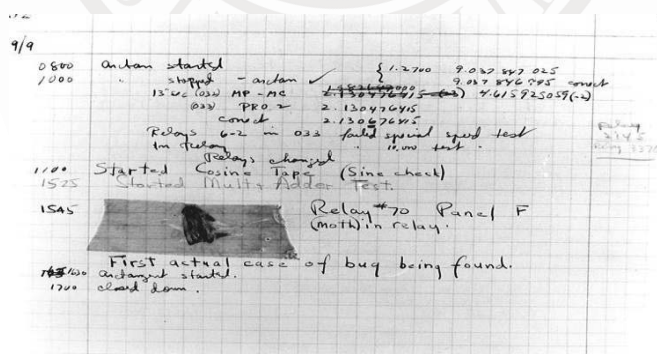
povezati sa problematičnim softverom, koji je slaba tačka. Ovde je bitno naglasiti da, za razliku od hardvera, softver ne stari – ukoliko se pojave greške, to ne znači da su se one pojavile u međuvremenu, već su one od početka prisutne, a mogu se ispoljiti i nakon dužeg pravilnog funkcionisanja.

1.3. Osnovni pojmovi

Veoma često se u praksi dešava da se određeni pojmovi pogrešno koriste i tumače. Koja je razlika između greške, defekta i buga? Šta je incident? Da li razlika između ovih termina uopšte postoji ili se odnose na istu stvar? Kako bi se izbegle nedoumice, neophodno je na samom početku definisati osnovne pojmove.

Greška (engl. *error*) jeste rezultat ljudske aktivnosti, bilo za vreme specifikacije zahteva ili tokom pisanja programa. Na primer, neka funkcionalnost može biti pogrešno specificirana, što će kasnije dovesti do pogrešne implementacije, ili se može napraviti neka greška u kodiranju, koja će dovesti do neispravnog rada programa. Posledica greške se naziva **mana**, **defekt** ili **bug** – programu nešto nedostaje, ili ima funkciju koja se ne ponaša ispravno. **Otkaz** (engl. *failure*) nastaje kada sistem nije u mogućnosti da obavi funkciju koju korisnik od njega zahteva, jer se aktivira i izvršava defektni kod.

Odakle dolazi termin bug (buba)? U septembru 1945. godine, u relejima računara Harvard Mark II je pronađen moljac, i zatim prikazan u dnevnik (slika 1.3). Računari su u to vreme zauzimali ogromne prostorije veličine više desetina kvadratnih metara, a toplota unutrašnjih komponenti je privlačila moljce, muve i ostale leteće insekte. Kada bi insekt ušao u komponente, izazvao bi kratak spoj i kvar računara. Izraz bug se koristio i ranije, ali je tek tada postao popularan i koristi se i dan danas.



Slika 1.3. Prvi bug - moljac prikazan u dnevnik

Kada postoji defekt u softveru, javlja se simptom kojim korisnik postaje svestan otkaza u sistemu. Ovaj simptom se naziva **incident (ili poremećaj)**.

Testiranje je postupak izvršavanja softvera sa testovima. Testiranje može imati jedan od dva osnovna cilja: da se pronađu otkazi ili da se demonstrira ispravno ponašanje sistema. Testiranjem se mogu otkriti otkazi u sistemu, koje je kasnije potrebno ispraviti.

Test je skup ulaznih vrednosti, preduslova izvršavanja (engl. *preconditions*), očekivanih rezultata i stanja u kome sistem treba da ostane nakon završetka testa (engl. *postconditions*). Test se razvija sa ciljem da ispita određeno ponašanje programa, na primer da izvrši određenu putanju kroz program ili da verifikuje da li je određeni zahtev ispravno implementiran. Svaki test poseduje svoj jedinstveni identitet i može se pojednostavljeno posmatrati kao trojka (Ulaz, Stanje, Izlaz):

- Ulaz označava ulazne podatke
- Stanje označava stanje sistema u trenutku unosa ulaznih podataka
- Izlaz je očekivani izlaz sistema, zajedno sa stanjem sistema u trenutku završavanja testa

Kao analogija se može posmatrati doktor koji treba da postavi dijagnozu za pacijenta. Pacijent dolazi kod doktora sa listom uočenih poremećaja (simptoma) na osnovu kojih je postao svestan da postoje otkazi u njegovom telu. Doktor mora da otkrije defekt, odnosno uzrok simptoma. Da bi olakšao dijagnostiku, doktor može naručiti testove koji pokazuju anomaliju, poput visokog krvnog pritiska, povišenog nivoa glukoze ili holesterola. Ove anomalije odgovaraju greškama. Ipak, poseta doktoru i testiranje softvera se razlikuju u jednoj ključnoj stvari. Otkazi u softveru su greške u dizajnu i kodu programa, ne nastaju spontano, već postoje kao rezultat neke pogrešne ljudske odluke. Softver ne stari – ukoliko se pojave greške, to znači da se one nalaze u njemu od početka. Sa druge strane, medicinski problemi, poput otkaza u sistemskom hardveru, su često rezultat fizičke degradacije u toku vremena. Ova razlika je bitna jer objašnjava granicu do koje je (ne)moguće kontrolisati i predvideti softverske greške (tzv. granica ljudske gluposti). Drugim rečima, ne postoji način da se uhvate sve nasumične greške koje je čovek u stanju da napravi, a samim tim, nije moguće eliminisati sve moguće greške u softveru. Dodatno, korisnik softvera je takođe čovek, čije nasumično ponašanje takođe može izazvati nepredviđen rad softvera (ne možemo isključiti da korisnik neće na primer gurnuti prste u utikač).

Testiranje softvera se može definisati kao proces izvršavanja programa sa ciljem da se pronađu greške. Ukoliko posmatramo širi smisao, testiranje je proces koji se sastoji od statičkih i dinamičkih aktivnosti, sa ciljem da se odredi da li softver zadovoljava specificirane zahteve i otkriju defekti ili da se demonstrira ispravan rad. Testiranje softvera je deo procesa osiguravanja kvaliteta (engl. *Quality Assurance*), čiji je zadatak da se obezbedi isporuka kvalitetnog softvera u zahtevanom vremenskom roku.

Test set se odnosi na skup svih testova koje je potrebno izvršiti nad softverom. Testovi se ne biraju nasumično, već je neophodno uložiti trud u pažljivo planiranje i dizajn. Čitava jedna faza u procesu testiranja softvera je odvojena u ovu svrhu. Nasumično odabrani test nema nikakvog značaja ukoliko detektuje grešku koja je već pronađena nekim drugim testom. Testiranje svih mogućih kombinacija ulaznih podataka

ili takozvano iscrpno testiranje (engl. *exhaustive testing*) je nemoguće ili u najboljem slučaju nepraktično, pošto svaki netrivialni sistem ima veoma veliki domen ulaznih podataka.

Ukupan broj testova u test setu ne garantuje da će i testiranje biti uspešno. Ako se posmatra sledeći prost kod koji treba da izračuna minimum dva broja:

```
if(x>y)
    min = y;
else
    min = x; //greška, ovde treba da stoji min = x;
```

Da bi se pokrile sve moguće putanje i otkrila greška, dovoljna su dva testa sa ulaznim podacima (x=2, y=1) i (x=1, y=2). Nasuprot tome, tri testa, sa ulaznim podacima (x=3, y=1), (x=2, y=1) i (x=5, y=2) ne zadovoljavaju kriterijume, pošto ne detektuju grešku. Ukoliko se kod pogleda detaljnije, poslednja tri testa uopšte ne izvršavaju else granu u kojoj se nalazi greška (sva tri testa idu na true granu, pa su dva zapravo višak).

Kako se zna da li je rezultat izvršavanja testa ispravan ili ne? **Predviđanje rezultata testa** (engl. *test oracle*) je sredstvo, potpuno nezavisno od softvera koji se testira, koje se koristi za proveru ispravnosti rada softvera, odnosno za definisanje šta je očekivani rezultat izvršavanja svakog pojedinačnog testa. Najčešće je to čovek (ponekad može biti i automatizovan proces), koji će na osnovu specifikacije softvera odrediti šta je ispravno ponašanje programa, odnosno tačan očekivani rezultat svakog testa. Međutim, specifikaciju softvera takođe pišu ljudi, i za specifikaciju važi takođe da može biti puna grešaka, dvosmislena ili nekompletna.

1.4. Osnovna klasifikacija testiranja

Testiranje se na najosnovniji način može klasifikovati na dva načina – prema pristupu i prema nivou testiranja. Prema pristupu, testiranje se deli na:

- Funkcionalno – testiranje bazirano na specifikaciji
- Strukturno – testiranje bazirano na samom kodu

Funkcionalno testiranje posmatra program kao **crnu kutiju**, i implementacija u ovom slučaju nije poznata. Naziva se još **metode crne kutije**. Softver se prosto posmatra kao funkcija koja mapira vrednosti sa ulaza na izlaz sistema. Testovi se određuju isključivo na osnovu specifikacije softvera. Testovi su nezavisni od konkretne implementacije, tako da su upotrebljivi i ukoliko dođe do promene implementacije, dok razvoj testova može teći u paraleli sa razvojem softvera. Sa druge strane, mana je što je moguće da se deo implementiranih funkcionalnosti ne pokrije testovima, ukoliko nisu navedeni u specifikaciji softvera.

Strukturno testiranje se fokusira na samoj implementaciji programa i raspoloživom dostupnom kodu. Naziva se još **metode bele kutije**. Fokus je na izvršavanju svih programskih struktura i struktura podataka u softveru koji se testira, i na osnovu toga se određuju testovi. U ovom tipu testiranja, ne proverava se specifikacija, a samim tim nije moguće ni otkriti da li su sve specificirane funkcionalnosti zaista implementirane u programu.

Testiranje se prema nivou deli na:

- Jedinično
- Integraciono
- Sistemsko

Jedinično testiranje (engl. *unit testing*) se odnosi na testiranje pojedinačnih jedinica izvornog koda ili delova klase. Najmanja funkcionalna jedinica izvornog koda je najčešće jedna metoda unutar klase. Jedinično testiranje najčešće koriste programeri, kako bi testirali svoj napisani kod. Najčešće se koristi određeno okruženje za pisanje jediničnih testova (u slučaju programskog jezika Java to je alat JUnit). Jedinični test je zapravo komad koda koji testira drugi deo koda. Izvršavanje ovih testova je automatizovano – programer piše ove testove jednom, a oni se izvršavaju onoliko puta i onoliko često koliko je to potrebno.

Nakon završetka jediničnog testiranja, jedinice se integrišu u celinu. Na nivou integracionog testiranja, glavni fokus je na verifikaciji funkcionalnosti i interfejsa između integrisanih modula. Na kraju, kada se sve jedinice integrišu u kompletan sistem, sistemsko testiranje proverava ponašanje tog sistema kao celine u odnosu na specifikaciju sistema. Pošto je većina funkcionalnih zahteva proverena na nižim nivoima testiranja, ovde se akcenat stavlja na nefunkcionalne zahteve, poput brzine, sigurnosti, pouzdanosti, robusnosti itd.

1.5. Pojam softver testera

U ovom udžbeniku, termin softver tester se odnosi na profesionalaca informacionih tehnologija koji je zadužen za izvršavanje jedne ili više aktivnosti testiranja, uključujući dizajn i pisanje testova, izvršavanje testova (manuelnih ili automatskih) i davanje izveštaja programerima i menadžerima. Test menadžer je nadležan nad nekoliko softver testera, i definiše proces testiranja, sarađuje sa drugim menadžerima na projektu, i na svaki način pomaže softver testerima u radu. Softver testeri dizajniraju testove, pripremaju test skripte za izvršavanje i zatim ih izvršavaju nad softverom koji se testira (engl. *software under test*). Na kraju evaluiraju rezultate testova kako bi se odredilo da li testovi otkrivaju neku grešku u softveru, zavode greške ukoliko postoje, i pišu izveštaj o izvršenom testiranju.

1.6. Osnovni proces testiranja

Testiranje se često posmatra samo kao izvršavanje softvera sa ciljem pronalaženja grešaka. To je deo testiranja, ali ne predstavlja sve aktivnosti u procesu testiranja softvera. Aktivnosti postoje i pre i posle izvršavanja testova, i u njih, između ostalog, spadaju: planiranje, odabir uslova testiranja (engl. *conditions*), dizajn i izvršavanje testova, provera rezultata, evaluiranje izlaznog kriterijuma, izveštaj o procesu testiranja i završne aktivnosti.

Testiranje može kao cilj imati jednu od sledećih stavki:

- Pronalaženje defekata
- Podizanje nivoa poverenja u nivo kvaliteta
- Pružanje informacija za dalje odluke na nivou menadžmenta
- Prevenciju defekata

U zavisnosti od nivoa testiranja, takođe se mogu uočiti različiti ciljevi, koji u različitim situacijama mogu biti potpuno različiti:

- U testiranju za vreme razvoja softvera (jedinično, integraciono i testiranje sistema) cilj je da se pronađe što više defekata kako bi se mogli ispraviti.
- U *user acceptance* testiranju, glavni cilj je da se potvrdi da sistem radi kako se očekuje, i da se stekne poverenje da je sistem ispunio sve zahteve.

Najvidljiviji deo testiranja je samo izvršavanje testova. Međutim, to je samo deo aktivnosti u procesu testiranja. Osnovni proces testiranja se sastoji i od drugih bitnih aktivnosti:

- Planiranje i kontrola testiranja
- Analiza i dizajn testova
- Implementacija i izvršavanje testova
- Evaluacija izlaznog kriterijuma i izveštavanje
- Završne aktivnosti

Planiranje se odnosi na definisanje ciljeva testiranja i specifikaciju aktivnosti potrebnih da se ti ciljevi ispune. Kontrola se odnosi na neprekidnu aktivnost poređenja progressa sa planom, i izveštavanja o statusu. Uključuje i akcije potrebne da se ciljevi ispune.

Analiza i dizajn testova su aktivnosti u kojima se uopšteni ciljevi transformišu u opipljive testove, dok se implementacija i izvršavanje testova najuopštenije gledano odnose na izvršavanje testova. Evaluacija izlaznog kriterijuma se odnosi na aktivnost gde se rezultati izvršavanja testova upoređuju sa zadatim ciljevima (izvršava se za svaki nivo testiranja ponaosob) i odlučuje se da li je potrebno još testiranja ili ne, nakon čega se piše sumarni izveštaj.

Na kraju testiranja, u završne aktivnosti spadaju:

- Provera da li je sve što je planirano isporučeno
- Dokumentovanje o prihvatanju (engl. *acceptance*) sistema
- Arhiviranje *testware*-a (testovi, okruženje, infrastruktura) za kasniju ponovnu upotrebu
- *Lessons learned* za buduće verzije sistema i sledeće projekte.

1.7. Osnovni principi testiranja

Postoji sedam osnovnih principa testiranja, koji su se izdvojili kroz nekoliko decenija prakse testiranja softvera, i koji važe za bilo koji tip softvera. Posmatraju se kao smernice koje bi trebalo ispoštovati u svakom projektu, jer su se kroz istoriju pokazali kao tačni. Principi glase:

1. Testiranje pokazuje prisustvo defekata – testiranje može da pokaže da su defekti prisutni, ali ne može da dokaže da u sistemu nema nijednog defekta. Smanjuje šansu da postoje neotkriveni defekti u softveru.
2. Iscrpno testiranje nije moguće – testiranje svih mogućih kombinacija ulaza i preduslova u praksi nije moguće za bilo koji netrivialni sistem. Radi se analiza rizika, i testovi se prioritiziraju.
3. Rano testiranje – kako bi se što pre otkrili defekti, testiranje treba da počne što je pre moguće.
4. Grupisanje defekata (engl. *defect clustering*) – testiranje treba fokusirati proporcionalno broju očekivanih i kasnije pronađenih defekata (gustini defekata) po modulima. Manji broj modula obično sadrži veći broj defekata. Ti moduli su obično najkritičniji i sadrže implementaciju bitnih funkcionalnosti sistema.
5. Paradoks pesticida – ukoliko se isti testovi ponavljaju u svakoj iteraciji testiranja, na kraju taj isti skup testova više neće moći da pronađe nijedan novi defekt. Testove treba redovno revidirati, i dodati nove (različite) testove kako bi se izbegao paradoks.
6. Testiranje zavisi od konteksta. Testiranje se različito izvršava u različitim kontekstima – softver koji je kritičan po pitanju bezbednosti se testira drugačije od komercijalnog veb sajta.
7. Odsustvo grešaka ne garantuje da sistem radi kako treba – pronalaženje i ispravljanje defekata ne pomaže u slučaju da je sistem neupotrebljiv, ili da ne ispunjava želje i očekivanja korisnika.

Pitanja

1. Šta je komercijalni softver?
2. Zbog čega je neophodno adekvatno testirati komercijalni softver?
3. Šta je greška u softveru?
4. Šta je defekt?
5. Koja je razlika između pojmova greška, defekt i otkaz sistema?
6. Kako se definiše pojam testiranja?
7. Kako se definiše pojam testa?
8. Šta jedan test mora imati?
9. Kako se definiše proces testiranja softvera?
10. Pod kojim pojmom je poznat skup svih testova koje je potrebno izvršiti?
11. Na osnovu čega se odlučuje da li je rezultat izvršavanja testa ispravan ili ne?
12. Kako se sve može klasifikovati testiranje?
13. Koja je razlika između funkcionalnog i strukturnog testiranja?
14. Koji su nivoi testiranja?
15. Šta je jedinično testiranje?
16. Koja je razlika između integracionog i sistemskog testiranja?
17. Šta sve može biti cilj testiranja softvera?
18. Koje su sve aktivnosti u osnovnom procesu testiranja?
19. Da li testiranje može da dokaže da u sistemu nema defekata?
20. Šta je iscrpno testiranje, i da li je moguće sprovesti ga?
21. Šta znači pojam grupisanje defekata?
22. Šta znači pojam paradoks pesticida u testiranju?
23. Kada treba početi sa testiranjem softvera u projektu?
24. Da li odsustvo defekata garantuje da sistem radi kako treba?

2. JUNIT FRAMEWORK

JUnit je mali, ali moćan Java *framework* za kreiranje i izvršavanje automatskih jediničnih testova u programskom jeziku Java. Jedinično testiranje podrazumeva da se testira mali deo programa – metoda, klasa ili nekoliko manjih klasa koje čine jednu komponentu i slično, u izolaciji od drugih delova koda. Na ovaj način se brzo može uočiti ukoliko nešto nije u redu sa kodom.

JUnit se često koristi u klasičnom testiranju, kao sredstvo za automatizaciju testiranja. Sa druge strane, JUnit ima ključnu ulogu u *test-driven* razvoju programa, i promovise ideju *prvo testiraj pa onda kodiraj*, gde se podrazumeva da programer paralelno sa pisanjem koda piše i jedinične testove. Ovaj pristup se zasniva na ideji da se pre programiranja određene jedinice napiše jedinični test koji će verifikovati da li je ta jedinica ispravno programirana. Na ovaj način se programiranje svodi na naizmenične cikluse pisanja jediničnih testova i programiranja samih jedinica, što kao rezultat obično ima povećanje produktivnosti programera i stabilnosti programskog koda, i smanjenje vremena koje se kasnije troši na debugovanje programa. Efekti promena u kodu su odmah vidljivi, a pošto se po pravilu testiraju mali delovi koda, mnogo lakše se može razumeti problem, uočiti i ispraviti bug.

Jedna od glavnih prednosti upotrebe JUnit alata za testiranje napisanog koda jeste brza povratna informacija o problemu. Programeri koji izbegavaju da pišu JUnit testove često koriste **System.out.println()** u svom kodu, gde je neophodno ručno porediti aktuelni rezultat sa očekivanim rezultatom. Ovaj pristup je veoma spor i podložan daljim greškama, i svakako se ne preporučuje u bilo kom ozbiljnijem projektu. Pisanje JUnit testova ne zahteva previše vremena – jednostavno se definišu šta su ulazni podaci u metodi i proverava se očekivani rezultat. Jednim klikom na dugme testovi se automatski izvršavaju, a JUnit jednostavno pokaže crvenu statusnu liniju ukoliko problemi postoje, odnosno zelenu ukoliko je sve u redu.

JUnit je open source framework, i dostupan je na sajtu <http://www.junit.org>. Na ovom sajtu se može pronaći JUnit biblioteka (junit.jar), dodatne informacije i primeri, kao i sve dostupne verzije samog alata. U ovoj knjizi primeri su urađeni sa JUnit 4, verzijom 4.12, i okruženjem NetBeans verzija 8.2. NetBeans okruženje automatski sadrži integrisan JUnit, a integracija je moguća i sa svim ostalim popularnim okruženjima za Java programiranje poput Eclipse. Potrebno je samo skinuti JAR fajl sa gore pomenute adrese i dodati ga u putanju projekta.

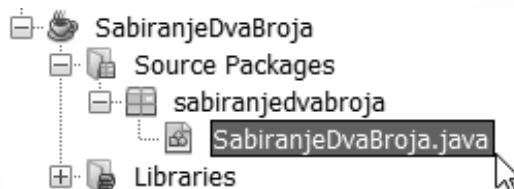
2.1. Osnove JUnit alata

Osnove JUnit alata najlakše je pokazati na konkretnim primerima. Podrazumeva se da čitalac poznaje osnove programskog jezika Java. Posmatramo klasu SabiranjeDvaBroja, koja ima implementiranu metodu saberi u sledećem obliku:

```
package sabiranjedvabroja;

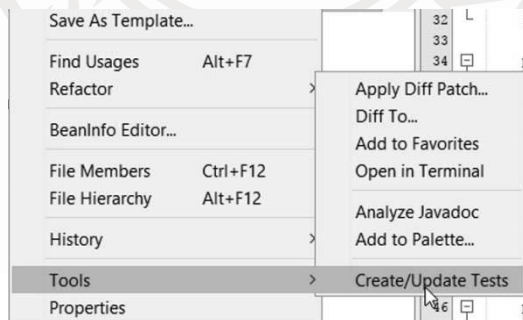
public class SabiranjeDvaBroja {
    public static int saberi (int prvi, int drugi) {
        return prvi + drugi;
    }
}
```

Hijerarhija klasa ovog programa u okviru NetBeans projekta je prikazana na slici 2.1.



Slika 2.1. Hijerarhija klasa programa u NetBeans projektu

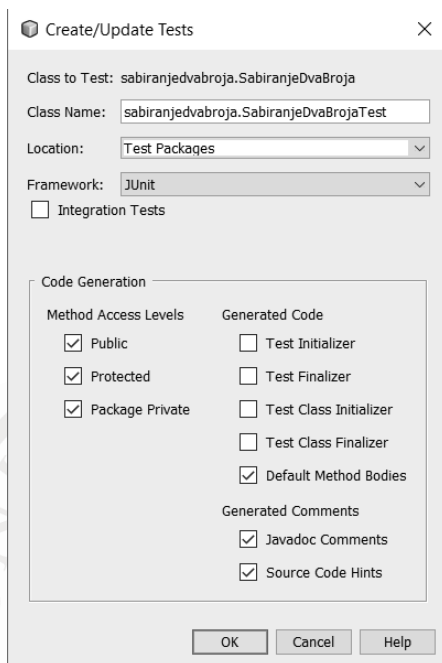
Uočava se klasa za koju je potrebno napisati JUnit test, u ovom slučaju je to klasa `SabiranjeDvaBroja.java`, koja se nalazi u okviru projekta pod imenom `SabiranjeDvaBroja`, u *Source Packages*, unutar paketa `sabiranjedvabroja`. Desnim klikom na ovaj java fajl se otvara meni sa dodatnim opcijama, od kojih je potrebno odabrati opciju `Tools`, a zatim opciju `Create/Update Tests`, kao što je prikazano na slici 2.2.



Slika 2.2. Kreiranje testova u NetBeans projektu

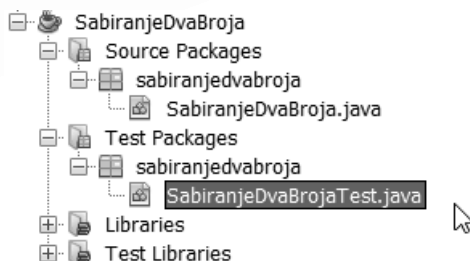
Klikom na `Create/Update Tests` opciju dobija se dijalog sa dodatnim podešavanjima, prikazan na slici 2.3. Selekcijom odgovarajućih opcija moguće je automatski dodati inicijalizaciju i finalizaciju testa ili test klase, o čemu će biti reči kasnije, kao i dodati *default* telo test metoda. Moguće je automatski generisati Javadoc komentare za kasnije

kreiranje Javadoc dokumentacije, kao i hintove za izvorni kod. U ovom početnom primeru selektovana je opcija *Default Method Bodies*, koja će automatski za svaku metodu klase koja se testira dodati jedan test u test klasi, sa *default* telom.



Slika 2.3. Dijalog sa dodatnim opcijama za kreiranje test klase

Klikom na taster OK, generiše se odgovarajuća test klasa, čije podrazumevano ime je ime klase koja se testira sa dodatim Test na kraju, u ovom slučaju SabiranjeDvaBrojaTest.java. Ova klasa se nalazi u automatski generisanom folderu Test Packages, unutar paketa pod istim imenom kao i paket u kome se osnovna klasa nalazi u Source Packages, prikazano na slici 2.4.



Slika 2.4. Test klasa unutar sekcije Test Packages

Automatski generisana test klasa je prikazana na slici 2.5. Za svaku metodu klase `SaberiDvaBroja` su automatski generisani testovi – pošto imamo samo jednu metodu `saberi`, generisana je jedna metoda `testSaberi` u test klasi. Automatski kreirano telo metode svakog generisanog testa (ukoliko je pri kreiranju selektovana opcija *Default Method Bodies*) je samo predviđeno da bude vodilja, i mora biti modifikovano kako bi to zaista bio test. Ukoliko nije potrebno generisati kod, opciju *Default Method Bodies* je potrebno ostaviti neselektovano.

```
package sabiranjeDvaBroja;

import org.junit.Test;
import static org.junit.Assert.*;

/**
 *
 * @author mzivkovic
 */
public class SabiranjeDvaBrojaTest {

    public SabiranjeDvaBrojaTest() {
    }

    /**
     * Test of saberi method, of class SabiranjeDvaBroja.
     */
    @Test
    public void testSaberi() {
        System.out.println("saberi");
        int prvi = 0;
        int drugi = 0;
        int expectedResult = 0;
        int result = SabiranjeDvaBroja.saberi(prvi, drugi);
        assertEquals(expResult, result);
        // TODO review the generated test code and remove the default call to fail.
        fail("The test case is a prototype.");
    }

}
```

Slika 2.5. Automatski generisana test klasa

Na početku test klase može se uočiti import dva paketa:

```
import org.junit.Test;
import static org.junit.Assert.*;
```

Prvi import služi da obezbedi dostupnost klase JUnit okruženja, poput klase `Test`. Drugi import služi da se sve provere (engl. *asserts*) pišu bez prefiksa klase, u obliku **`assertEquals()`** umesto **`Assert.assertEquals()`**. Ove metode su definisane kao statičke u klasi `Assert`. Sama test klasa koja se piše ne treba da nasleđuje ništa, pošto se koristi Java mehanizam refleksije.

Svaki poseban test u test klasi se piše kao public void metod bez ulaznih parametara, koji obavezno mora imati prefiks `@Test`. Ova oznaka govori okruženju da je u pitanju test, i samo u tom slučaju će metoda biti i tretirana kao test, odnosno izvršena pri pokretanju testova. Kako bi test klasa iz primera mogla da se koristi, neophodno je izmeniti telo metode `testSaber`i. Podrazumevano telo na samom kraju ima poziv metode `fail()` koji će automatski postaviti status celog testa na *Failed*.

Rezultat dobijen izvršavanjem dela koda koji je predmet testa (u ovom slučaju poziv `saber`i()) metode iz klase `SabiranjeDvaBroja`) se unutar test metode `testSaber`i() proverava pozivom jednog od `assert` metoda definisanih u okruženju JUnit. U ovom primeru, koristi se `assertEquals` metoda, koja je u JUnit-u definisana na sledeći način:

```
assertEquals( expected, actual)
```

Ukoliko se vrednost `expected` (očekivana vrednost) ne poklapa sa `actual` (vrednost koja se dobija izračunavanjem u kodu koji se testira), izbacuje se izuzetak tipa **java.lang.AssertionError**. Ovaj izuzetak dovodi do prekidanja izvršavanja trenutne test metode, status testa se automatski postavlja na *Failed*, i nastavlja se sa izvršavanjem ostalih test metoda.

Nakon neophodnih izmena tela metode `testSaber`i(), ona ima sledeći oblik:

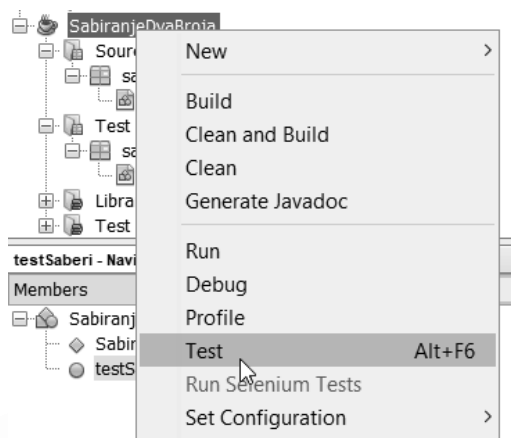
```
@Test
public void testSaber() {
    int prvi = 12;
    int drugi = 15;
    int expectedResult = 27;
    int result = SabiranjeDvaBroja.saber(prvi, drugi);
    assertEquals(expResult, result);
}
```

Definišu se dve integer vrednosti za sabirke, 12 i 15, i kao očekivani rezultat se stavlja vrednost 27 (određeno specifikacijom programa, u ovom slučaju pravilima sabiranja), pošto se ta vrednost očekuje kao aktuelni rezultat ukoliko je sabiranje ispravno implementirano. Aktuelni rezultat se dobija pozivom metode koja se testira – metoda `saber`i iz klase `SabiranjeDvaBroja` sa definisanim vrednostima za sabirke. Na kraju se, uz pomoć `assertEquals` metode, proverava da li se očekivani rezultat (prvi parametar metode `assertEquals`) poklapa sa aktuelnim rezultatom dobijenim pozivom metode koja se testira i izračunavanjem povratne vrednosti (drugi parametar metode `assertEquals`). Ukoliko se ove dve vrednosti poklapaju, rezultat testa će biti *Passed*, u suprotnom će imati vrednost *Failed*.

Moguće je pisati test i u skraćenom obliku, bez kreiranja privremenih promenljivih `prvi`, `drugi`, `expResult` i `result`, u obliku:

```
@Test
public void testSaber() {
    assertEquals(27, SabiranjeDvaBroja.saber(12, 15));
}
```

Kada su testovi napisani, možemo ih izvršiti desnim klikom na projekat, pa klikom na test (ili uz pomoć skraćenice Alt + F6), kao što je prikazano na slici 2.6.



Slika 2.6. Pokretanje izvršavanja testova

Nakon pokretanja izvršavanja testova, svi testovi koji su označeni notacijom `@Test` će biti izvršeni, a rezultati će biti prikazani u JUnit Test Results prozoru u okviru okruženja NetBeans, kao što je prikazano na slici 2.7.

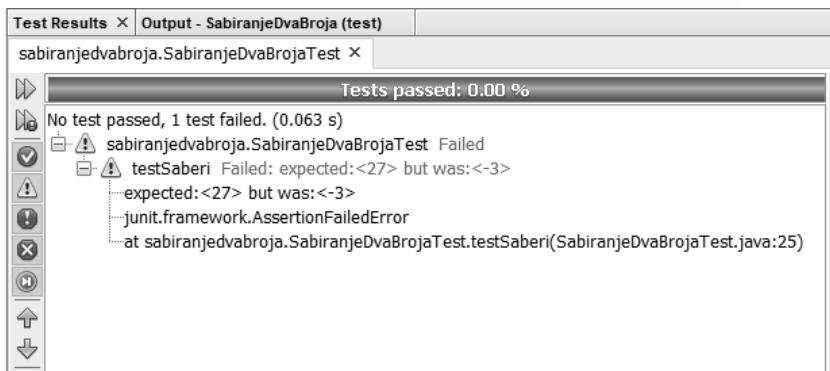


Slika 2.7. Test Results prozor okruženja NetBeans

U ovom slučaju, pošto u metodi koja se testira ne postoji greška, rezultat testa je *Passed*. Zelena statusna linija označava da nijedan test nije pao (*Failed*), a procenat označava broj testova koji su prošli (status *Passed*). Neka se sada ponovo posmatra izmenjena metoda `saberer`, ovaj put sa greškom pošto je umesto znaka `+` za sabiranje pogrešno stavljen znak `-`.

```
public static int saberi (int prvi, int drugi) {
    return prvi - drugi;
}
```

Nakon pokretanja izvršavanja testova, Test Results sada prikazuje da test nije prošao (status *Failed*), sa jasnim vizuelnim prikazom crvene boje, kao i opisom zbog čega test nije prošao. U ovom slučaju, jasno je naznačeno da je greška u testu testSaber, koji testira metodu saberi, pošto je očekivana vrednost 27, a aktuelna vrednost dobijena izvršavanjem koda metode saberi je -3 (zbog toga što metoda greškom oduzima dva broja umesto da ih sabira).



Slika 2.8. Test Results prozor u slučaju kada postoje greške

2.2. Klasa Assert

Rezultat koji se dobija izvršavanjem koda koji se testira se proverava pozivom jednog od assert metoda definisanih u klasi Assert (org.junit.Assert). Na osnovu assert metoda se odlučuje da li je određeni test prošao ili nije. Postoji veći broj assert metoda u okviru klase Assert koje se mogu koristiti u pisanju testova. U nastavku je dat pregled najčešće upotrebljivanih metoda provere.

Ukoliko je potrebno testirati da li neki boolean uslov ima vrednost true ili false, može se koristiti jedna od sledeće dve metode:

assertTrue(uslov) – proverava je uspešno prošla ukoliko je vrednost uslova true, u suprotnom je test pao.

assertFalse(uslov) – proverava je uspešno prošla ukoliko je vrednost uslova false, u suprotnom je test pao.

Ponekad je potrebno proveriti da li je neki objekat uspešno inicijalizovan (odnosno da li neka promenljiva klasnog tipa pokazuje na konkretan postojeći objekat ili je nedefinisana – null). Na raspolaganju su sledeće dve metode:

assertNull(object) – provera je uspešno prošla ukoliko promenljiva object ima vrednost null (nedefinisana je, ne pokazuje na konkretan objekat), u suprotnom je test pao.

assertNotNull(object) – provera je uspešno prošla ukoliko promenljiva object pokazuje na neki konkretan objekat (odnosno nije null), u suprotnom je test pao.

Moguće je proveriti i da li su dva objekta identična (na primer, proveriti da li dve reference pokazuju na isti objekat). Na raspolaganju su sledeće dve metode:

assertSame(prvi, drugi) – provera je uspešno prošla ukoliko je **prvi == drugi**.

assertNotSame(prvi, drugi) – provera je uspešno prošla ukoliko je **prvi != drugi**.

Provera da li su dva objekta jednaka se radi pomoću assertEquals metode, u obliku:

assertEquals(expected, actual), gde će provera uspešno proći u slučaju da je vrednost poziva **expected.equals(actual)** jednaka true. U ovom slučaju koristi se implementacija equals relacije u klasi koja se testira. Ukoliko klasa koja se testira ne nadjačava **equals()** metodu nasleđenu iz klase **Object**, korišće se podrazumevano ponašanje **equals()** metode kako je definisano u klasi **Object**, odnosno provera identičnosti dva objekta.

Za primitivne tipove (osim realnih) se assertEquals metoda koristi u istom obliku:

assertEquals(expected, actual)

Ukoliko su vrednosti expected i actual identične, provera je uspešno prošla, u suprotnom je test pao. Moguće je definisati i poruku koja će se staviti u AssertionError u slučaju da provera nije uspešno prošla, na sledeći način:

assertEquals(message, expected, actual) – gde je prvi argument poziva metode String sa porukom koju želimo da prosledimo u AssertionError.

Za realne tipove postupak je malo drugačiji. Ukoliko se porede realni tipovi (float ili double), zahteva se još jedan parametar u pozivu metode assertEquals – prihvatljivo odstupanje (delta) kako bi se izbegle greške u zaokruživanju pri poređenju realnih brojeva.

assertEquals(expected, actual, delta)

Metoda assert će u ovom slučaju proveru izvršiti na sledeći način:

Math.abs(expected – actual) <= delta

Primer koji demonstrira upotrebu različitih prethodno opisanih assert naredbi je dat u nastavku.

```

@Test
public void testAssert(){

    //Deklaracija promenljivih
    String string1 = "JUnit";
    String string2 = "JUnit";
    String string3 = "Testiranje softvera";
    String string4 = "Testiranje softvera";
    String string5 = null;

    int promenljiva1 = 1;
    int promenljiva2 = 2;
    double korenDvojke = 1.4142;

    //Assert naredbe
    assertEquals(string1,string2);
    assertEquals(string3, string4);
    assertEquals(string1, string3);

    assertEquals(string1);
    assertEquals(string5);
    assertEquals(promenljiva1<promenljiva2);
    assertEquals(korenDvojke, Math.sqrt(2), 0.001);
    //0.001 se odnosi na dozvoljeno odstupanje
}

```

Ukoliko je potrebno, moguće je ručno postaviti rezultat testa na *Failed*, pozivom **fail()** ili **fail(message)** metode, koja prekida izvršavanje testa i obara mu status na pao uz odgovarajuću poruku ako se koristi drugi oblik poziva (message).

2.3. @ Anotacije

Anotacije su uvedene u JUnit verziji 4, čime je kod značajno pojednostavljen i čitkiji u odnosu na prethodne verzije JUnit okruženja. Anotacije počinju znakom @, i mogu se posmatrati kao meta-tagovi koji se mogu primeniti u kodu. Uz pomoć anotacija JUnit zna koje metode da posmatra kao testove, koje metode da izvrši pre ili posle pokretanja testova, koje metode da ignoriše i slično. U nastavku ovog poglavlja dat je pregled svih anotacija, zajedno sa primerima upotrebe.

@Test – ova anotacija označava da se public void metoda ispred koje je navedena može pokrenuti kao test. U praksi to jednostavno znači da ispred svakog testa mora postojati oznaka @Test kako bi JUnit tu metodu tumačio kao test koji može da se izvrši. Pre izvršavanja metode, JUnit prvo konstruiše svežu instancu test klase i zatim poziva metodu. Ukoliko metoda baci bilo kakav izuzetak (java.lang.AssertionError ako je došlo do pada nekog assert uslova, ili bilo koji drugi izuzetak usled neke greške u izvršavanju

metode), JUnit prijavljuje grešku u testu postavljanjem statusa na Failed. Ukoliko nije izbačen nijedan izuzetak, test se posmatra kao prošao sa statusom Passed.

```
@Test
public void testMetoda() {
    assertTrue( new ArrayList().isEmpty() );
}
```

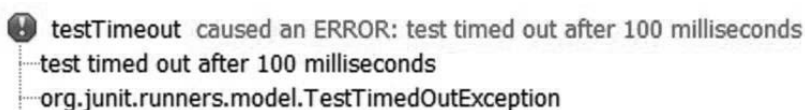
@Test (expected = Exception.class) – Ponekad je neophodno da testiramo situaciju kada se u testu prilikom izvršavanja desi izuzetak. Upotrebom parametra expected deklarise se da se očekuje da će test metoda baciti tačno određen tip izuzetka (deklarisan kao Exception.class). Ukoliko test metoda ne baci izuzetak, ili baci izuzetak drugačijeg tipa od onog koji je deklarisan, test će pasti. Na primer, u slučaju pokušaja deljenja sa nulom, očekujemo da će biti izbačen ArithmeticException. U ovom slučaju nije neophodno pisati assert naredbu.

```
@Test (expected = ArithmeticException.class)
public void deljenjeSaNulom() {
    int a = 5;
    int b = 0;
    double c = a / b;
}
```

@Test(timeout=100) – Ukoliko je potrebno zadovoljiti određene performanse po pitanju vremena izvršavanja, koristi se dodatni parametar timeout, koji će oboriti test u slučaju da je njegovo vreme izvršavanja duže od zadatog vremena u milisekundama. Najlakše se može demonstrirati postavljanjem beskonačne petlje unutar testa.

```
@Test (timeout = 100)
public void testTimeout () {
    while (true){
    }
}
```

Nakon pokretanja testa, po isteku 100 milisekundi, izvršavanje tog testa će biti prekinuto i rezultat postavljen sa failed, kao što je prikazano na slici 2.9.

The image shows a screenshot of a test failure message in a console or IDE. It starts with a red circle containing a white exclamation mark, followed by the text "testTimeout caused an ERROR: test timed out after 100 milliseconds". Below this, it says "test timed out after 100 milliseconds" and then "org.junit.runners.model.TestTimedOutException".

testTimeout caused an ERROR: test timed out after 100 milliseconds
test timed out after 100 milliseconds
org.junit.runners.model.TestTimedOutException

Slika 2.9. Primer upotrebe parametra timeout

Prilikom pisanja testova, često se dešava da su većem broju testova u istoj klasi potrebni slični objekti kako bi mogli da se izvrše, ili su im potrebni isti resursi koje je potrebno. U tom slučaju mogu se koristiti anotacije `@Before` i `@After` za inicijalizaciju i finalizaciju testova.

@Before – anotacija za metodu koja inicijalizuje preduslove za test (metoda se podrazumevano naziva `setUp`, ali moguće je naravno dati joj bilo koje ime). Ova metoda predstavlja inicijalizaciju svakog testa, i pokreće se pre svakog pojedinačnog testa u klasi. Nije neophodna, ali se mora koristiti ukoliko je potrebno inicijalizovati neke objekte pre početka izvršavanja svakog testa. U tom slučaju neophodno je dodati još jedan import u zaglavlje fajla: **`import org.junit.Before;`**

U sledećem primeru se u oba testa koristi lista, koja je izdvojena kao član podatak klase. Kako bi u svakom testu moglo da se radi sa tom listom, neophodno je inicijalizovati je u `@Before` sekciji, koja će biti pozvana i izvršena pre svakog pojedinačnog testa.

```
List<String> testList;

@Before
public void initialize() {
    testList = new ArrayList<String>();
}

@Test
public void testLista1() {
    //provera da li je lista prazna
    assertTrue( testList.isEmpty() );
}

@Test
public void testLista2() {
    //provera da li dodavanje elemenata ispravno radi
    testList.add("testiranje softvera");
    assertEquals(1, testList.size());
}
```

Ukoliko se unutar `@Before` metode alociraju neki eksterni resursi, neophodno ih je ispravno osloboditi nakon izvršavanja testa. Slično, ako testovi koji rade sa bazom, i za vreme izvršavanja unesu nove test zapise u bazu – nakon završetka svakog testa potrebno je počistiti bazu i vratiti je u prvobitno stanje kako bi i sledeći testovi imali odgovarajuće preduslove. Anotacija **@After** ispred `public void` metode obezbeđuje da takva metoda izvrši nakon svakog Testa. Sve metode označene sa `@After` će se zagarantovano izvršiti čak i u slučaju da unutar testa dođe do bacanja izuzetka. Potrebno je dodati još jedan import u zaglavlje fajla: **`import org.junit.After;`**

Primer upotrebe `@Before` i `@After` je dat u nastavku.

```

import org.junit.Before;
import org.junit.After;
import org.junit.Test;
public class TestKlasa {

    OutputStream stream;
    @Before
    public void initialize() {
        /**
         * Otvaramo OutputStream, i koristimo ga za testove. Poziva se
         pre svakog zasebnog testa
         */
        stream = new FileOutputStream(...);
    }
    @Test
    public void testMetoda() {
        /**
         *Koristimo OutputStream objekat za izvršavanje testova
         */
        ...
    }
    @After
    public void closeOutputStream() {
        /**
         * Ovde zatvaramo output stream, biće pozvano nakon svakog
         pojedinačnog testa
         */
        try{
            if(stream != null) stream.close();
        } catch(Exception ex){
        }
    }
}

```

Često više testova imaju zajedničku inicijalizaciju koja je zahtevna po pitanju resursa računara, poput logovanja na bazu podataka. To je neophodna optimizacija iako narušava nezavisnost testova. U tom slučaju moguće je koristiti **@BeforeClass** anotaciju. Metoda koja je public static void, i koja ima ovu anotaciju će se izvršiti tačno jedanput pre izvršavanja testova u toj klasi. Potrebno je dodati još jedan import u zaglavlje fajla: **import org.junit.BeforeClass;**

Na sličan način, ukoliko je potrebno uraditi oslobađanje zauzetih resursa nakon izvršavanja svih testova u klasi, koristi se metoda sa anotacijom **@AfterClass**. Ova public static void metoda će se izvršiti tačno jedanput, nakon izvršavanja svih testova u klasi. Potrebno je dodati još jedan import u zaglavlje fajla: **import org.junit.AfterClass;**

Primer koji demonstrira razliku između `@Before` i `@BeforeClass`, kao i razliku između `@After` i `@AfterClass` je dat u nastavku.

```
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
public class TestKlasa {

    @BeforeClass
    public static void setUpClass() {
        System.out.println("Izvršavanje BeforeClass metode");
    }
    @AfterClass
    public static void tearDownClass() {
        System.out.println("Izvršavanje AfterClass metode");
    }
    @Before
    public void setUp() {
        System.out.println("Izvršavanje Before metode");
    }
    @After
    public void tearDown() {
        System.out.println("Izvršavanje After metode");
    }
    @Test
    public void testMetoda1(){
        System.out.println("Test1");
    }
    @Test
    public void testMetoda2(){
        System.out.println("Test2");
    }
}
```

Rezultat izvršavanja programa jasno pokazuje redosled poziva metoda ispisom odgovarajućeg teksta na standardnom izlazu pri pozivu svake odgovarajuće metode. `BeforeClass` metoda je izvršena tačno jedanput, na samom početku, pre početka izvršavanja testova. `AfterClass` metoda je izvršena tačno jedanput, na samom kraju nakon izvršavanja svih testova. `Before` metoda se izvršava pre svakog testa, a `After` metoda nakon svakog testa.

Rezultat izvršavanja programa:

```
Izvršavanje BeforeClass metode
Izvršavanje Before metode
Test1
Izvršavanje After metode
Izvršavanje Before metode
Test2
Izvršavanje After metode
Izvršavanje AfterClass metode
```

@Ignore anotacija se koristi u slučaju da je potrebno da se privremeno preskoči izvršavanje nekog testa, i da bude navedeno u finalnom izveštaju kao ignorisani test. U opcionom String parametru se navodi razlog zašto je test deaktiviran. Ova opcija pomaže u *test-driven* razvoju softvera, pošto omogućava da se test napiše čak i pre implementacije određene funkcionalnosti. Takav test se označava sa **@Ignore** sve dok funkcionalnost ne bude implementirana, kada se anotacija jednostavno ukloni. **@Ignore** anotacija se može koristiti i na nivou cele test klase, kada će svi testovi koji se nalaze u toj klasi biti ignorisani. Pri upotrebi ove anotacije mora se dodati sledeći import u zaglavlje test klase - **import org.junit.Ignore;**

Kao primer posmatra se klasa **MatematickeOperacije**, koja treba da implementira osnovne 4 matematičke operacije. U ovom primeru metoda **pomnozi ()** nije još uvek implementirana, a kako bi primer bio što realniji u metodi **oduzmi ()** postoji greška. Još je potrebno uočiti da u metodi **podeli ()** može doći do pokušaja deljenja sa nulom, što će dovesti do izbacivanja izuzetka tipa **ArithmeticException**. Implementacija klase **MatematickeOperacije** je data u nastavku.

```
public class MatematickeOperacije {
    public static int saberi (int prvi, int drugi) {
        return prvi + drugi;
    }
    public static int oduzmi (int prvi, int drugi){
        return prvi - 1;
    }
    public static int pomnozi (int prvi, int drugi){
        //nije jos uvek implementirano
        return 0;
    }
    public static double podeli (int prvi, int drugi){
        return prvi/drugi;
    }
}
```

Odgovarajuća test klasa treba da ima testove za svaku od metoda klase koja se testira. Iako metoda **pomnozi ()** još uvek nije implementirana, moguće je napisati test unapred, pošto znamo očekivani rezultat operacije množenje, tj. da ukoliko metodi kao parametre prosledimo vrednosti 5 i 4, kao rezultat očekujemo 20. Nakon pisanja tog

testa, potrebno je označiti ga anotacijom `@Ignore`, koju će biti uklonjena onog trenutka kada metoda pomnozi () bude implementirana. U slučaju metode podeli (), pošto može doći do izbacivanja izuzetka u slučaju deljenja sa nulom, neophodno je proveriti kako se implementirana metoda ponaša tako što će se namerno pokušati deljenje sa nulom, a kao rezultat tog testa očekivati izbacivanje izuzetka tipa `ArithmeticException`. Kompletna implementacija test klase je data u nastavku. Zbog jasnoće primera, izdvojene su deklaracije lokalnih promenljivih prvi, drugi, expectedResult i result ispred assert naredbi, kako bi se jasno uočili i razlikovali očekivani rezultat (expectedResult) i stvarni rezultat izvršavanja (result). U ovom slučaju, očekivani rezultat se dobija na osnovu specifikacije funkcionalnosti koja treba da se implementira. Pošto su u pitanju osnovne matematičke operacije, na osnovu njihove definicije su izvedeni očekivani rezultati za svaki pojedinačni test (u slučaju sabiranja, ukoliko se sabiraju brojevi 1 i 3 na osnovu zakona sabiranja znamo da očekivani rezultat treba da bude 4).

```
import org.junit.Test;
import static org.junit.Assert.*;
import org.junit.Ignore;
public class MatematickeOperacijeTest {

    @Test
    public void testSaberi() {
        int prvi = 1;
        int drugi = 3;
        int expectedResult = 4;
        int result = MatematickeOperacije.saberi(prvi, drugi);
        assertEquals(expectedResult, result);
    }

    @Test
    public void testOduzmi() {
        System.out.println("oduzmi");
        int prvi = 5;
        int drugi = 2;
        int expectedResult = 3;
        int result = MatematickeOperacije.oduzmi(prvi, drugi);
        assertEquals(expectedResult, result);
    }

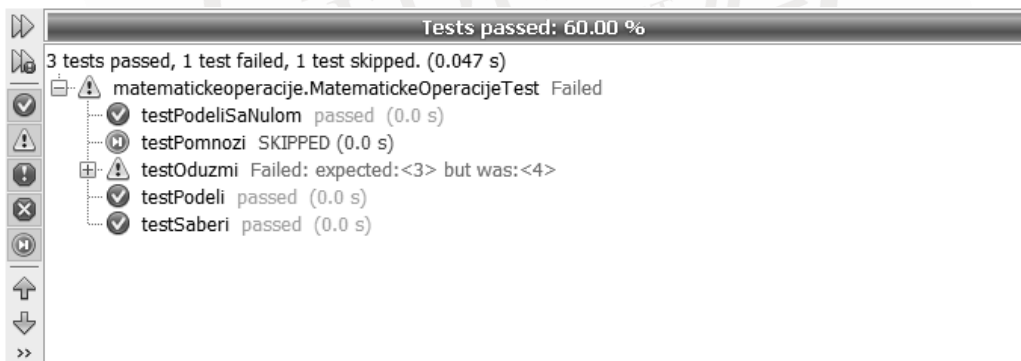
    @Test
    @Ignore ("Nije jos uvek implementirano")
    public void testPomnozi() {
        int prvi = 5;
        int drugi = 4;
        int expectedResult = 20;
        int result = MatematickeOperacije.pomnozi(prvi, drugi);
        assertEquals(expectedResult, result);
    }
}
```

```

@Test
public void testPodeli() {
    int prvi = 4;
    int drugi = 2;
    double expResult = 2;
    double result = MatematickeOperacije.podeli(prvi, drugi);
    assertEquals(expResult, result, 0.0);
}
@Test (expected = ArithmeticException.class)
public void testPodeliSaNulom () {
    int prvi = 4;
    int drugi = 0;
    double result = MatematickeOperacije.podeli(prvi, drugi);
}
}

```

Nakon izvršavanja testova u test klasi, dobija se izveštaj prikazan na slici 2.10. Može se primetiti da je test metode pomnozi () koji je označen sa anotacijom Ignore preskočen (status SKIPPED), i ne utiče na konačan status testiranja. Konačan status u ovom slučaju je *Failed*, zbog toga što je test metode oduzmi () pao zbog greške u implementaciji metode (umesto da vraća prvi – drugi, ova metoda greškom vraća prvi – 1, pa za prosledjene parametre prvi = 5 i drugi = 2 umesto rezultata 3, vraća rezultat 4, što je jasno naznačeno u statusu testa).



Slika 2.10. Rezultat izvršavanja testova za klasu MatematickeOperacije

2.4. Parametrizovani testovi

Svrha parametrizovanih testova jeste da se programeru dozvoli da izvršava isti test više puta korišćenjem različitih vrednosti. Ova funkcionalnost je relativno nova u JUnit alatu, pošto je uvedena u verziji JUnit4. Posmatra se klasa ProveraProstihBrojeva, koja

ima jednu metodu validiraj. Ova metoda proverava da li je ceo broj koji je prosleđen kao parametar prost ili ne. Postoje različiti algoritmi provere da li je broj prost ili ne – u ovom primeru se radi jednostavnosti samo proverava da li je prosleđeni broj deljiv sa bilo kojim brojem od 2 do broj/2.

```
public class ProveraProstihBrojeva {  
  
    public Boolean validiraj(Integer broj) {  
        for (int i = 2; i < (broj / 2); i++) {  
            if (broj % i == 0) {  
                return false;  
            }  
        }  
        return true;  
    }  
}
```

Jedan način da se ova metoda testira sa više različitih celobrojnih vrednosti jeste da se napiše više identičnih testova za metodu validiraj, ili da se u okviru jednog testa napiše više assert naredbi jedna za drugom. Za mali broj vrednosti koje je potrebno testirati to nije preveliki problem, ali je u svakom slučaju ponavljanje i kopiranje koda loša praksa jer je podložno greškama. U slučaju većeg broja vrednosti koje je neophodno testirati veoma je nepraktično kopirati isti kod više puta.

Drugi način jeste da se koristi parametrizacija. Definiše se skup test podataka, a zatim se test poziva sa definisanim parametrima iz skupa test podataka. Ukoliko je potrebno koristiti parametrizaciju, klasa se mora označiti anotacijom **@RunWith (Parameterized.class)**. Dalje, potrebno je napisati public static metod koji je označen anotacijom **@Parameters** I vraća kolekciju Objects u obliku Array, koji će biti skup podataka. Nakon toga je potrebno kreirati objektnu promenljivu (član podatak) za svaku kolonu test podataka (u ovom slučaju par ulazni broj i očekivani rezultat za taj broj), i napisati javni konstruktor koji uzima ekvivalent jednom redu test podataka i smešta uzete vrednosti u odgovarajuće objektno polje. Test se kreira upotrebom objektnih promenljivih kao izvora podataka, a prilikom izvršavanja će biti pozvan za svaki red podataka. Test klasa za klasu ProveraProstihBrojeva sa upotrebom parametara je data u nastavku.

```
import java.util.Arrays;  
import java.util.Collection;  
import org.junit.Test;  
import org.junit.Before;  
import org.junit.runners.Parameterized;  
import org.junit.runner.RunWith;  
import static org.junit.Assert.assertEquals;
```

```

@RunWith(Parameterized.class)
public class ProveraProstihBrojevaTest {

    private Integer ulazniBroj;
    private Boolean ocekivaniRezultat;
    //instanca klase koju testiramo
    private ProveraProstihBrojeva proveraProstihBrojeva;

    @Before
    public void initialize() {
        proveraProstihBrojeva = new ProveraProstihBrojeva();
    }

    //konstruktor koji uzima jedan red podataka
    //u ovom slucaju par ulazni celi broj i ocekivani rezultat
    //i smesta u odgovarajuće objektno polje
    public ProveraProstihBrojevaTest(Integer ulazniBroj,
        Boolean ocekivaniRezultat) {
        this.ulazniBroj = ulazniBroj;
        this.ocekivaniRezultat = ocekivaniRezultat;
    }

    @Parameterized.Parameters
    public static Collection prostiBrojevi() {
        return Arrays.asList(new Object[][]{
            {2, true},
            {6, false},
            {19, true},
            {22, false},
            {23, true}
        });
    }

    // Test ce biti izvršen 5 puta jer imamo 5 parametara

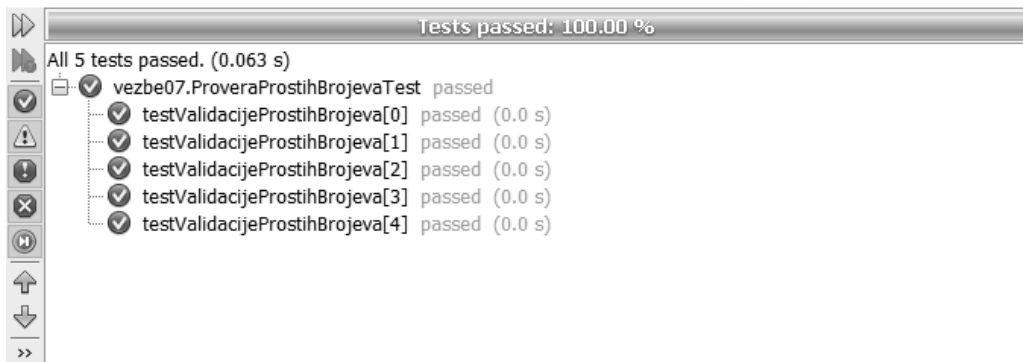
    @Test
    public void testValidacijeProstihBrojeva() {
        System.out.println("Test je pozvan sa : " + ulazniBroj);
        assertEquals(ocekivaniRezultat,
            proveraProstihBrojeva.validiraj(ulazniBroj));
    }
}

```

Test klasa prilikom izvršavanja za svako izvršavanje testa ispisuje na standardnom izlazu sa kojim parametrom je test pozvan:

```
Test je pozvan sa : 2
Test je pozvan sa : 6
Test je pozvan sa : 19
Test je pozvan sa : 22
Test je pozvan sa : 23
```

Izveštaj izvršavanja test klase je prikazan na slici 2.11.



Slika 2.11. Primer izvršavanja parametrizovanih testova

2.5. Kolekcija testova

Često se dešava da se testovi nalaze u više test klasa. Kao što je već navedeno, testovi se pokreću desnim klikom na NetBeans projekat i zatim odabirom opcije Test ili prečicom Alt + F6. Ova opcija će pokrenuti automatski sve testove koji se nalaze u Test Packages folderu. Za veću kontrolu pokretanja testova iz više klasa koristi se kolekcija testova – Test Suite.

Koristi se anotacija `@RunWith` koja definiše da se za izvršavanje test klasa koristi `runner.org.junit.runners.Suite`. Dodatna anotacija `@Suite.SuiteClasses` definiše koje test klase `runner` treba da uključi u ovu kolekciju i u kom redosledu. Na ovaj način moguće je pokrenuti samo deo test klasa (pokreću se samo one koje su navedene), a može se uticati i na redosled izvršavanja klasa, pošto se test klase izvršavaju po redosledu navođenja. Ukoliko na primer u projektu postoje 3 test klase, sa imenima `TestJunit1`, `TestJunit2` i `TestJunit3`, i želi se da se prilikom izvršavanja testova pokrenu samo klase `TestJunit2` i `TestJunit1` u tom redosledu, može se napisati kolekcija testova na sledeći način.

```

import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({
    TestJUnit2.class, //prvo navodimo TestJUnit2 klasu jer želimo da se
//prvo izvrši
    TestJUnit1.class
})

public class JunitTestSuite {
}

```

Definicija klase JunitTestSuite ostaje prazna, pošto ova klasa služi samo kao nosilac gore navedenih anotacija.

2.6. TestRunner klasa

Testovi se mogu izvršiti i uz pomoć TestRunner klase. Za pokretanje klasa se koristi runClasses metod JUnitCore klase, koja će vratiti rezultat testova u obliku Result objekta. Pomoću metode getFailures iz Result objekta se mogu dohvatiti sve greške. TestRunner klasa koja bi pokretala JunitTestSuite iz prethodnog primera je data u nastavku.

```

import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
    public static void main(String[] args) {
        Result result= JUnitCore.runClasses(JunitTestSuite.class);

        for (Failure failure : result.getFailures()) {
            System.out.println(failure.toString());
        }

        System.out.println(result.wasSuccessful());
    }
}

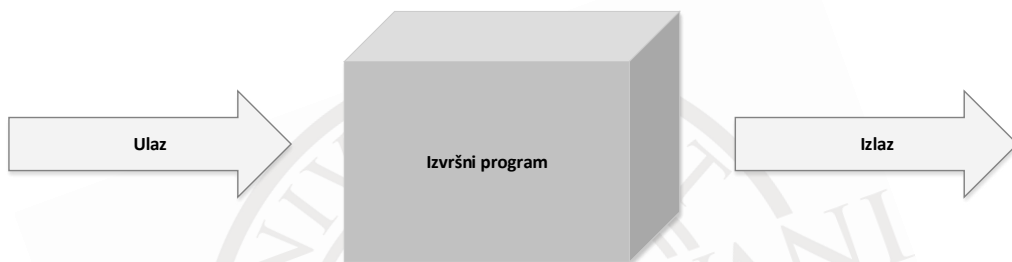
```

Pitanja

1. Šta je JUnit?
2. Koje su glavne funkcionalnosti koje JUnit pruža?
3. Zašto nije dobro koristiti `System.out.println()` za testiranje?
4. Šta je kolekcija testova u JUnit alatu?
5. Čemu služi `TestRunner` klasa?
6. Šta su anotacije i čemu služe?
7. Šta je svrha `@Test` anotacije?
8. Šta je razlika između `@Before` i `@BeforeClass` anotacije?
9. Šta je razlika između `@After` i `@AfterClass` anotacije?
10. Čemu služi `@Ignore` anotacija?
11. Koji parametar se koristi ukoliko je očekivano da test izbací izuzetak?
12. Kako se simulira timeout u testovima?
13. Šta su i kako se koriste parametrizovani testovi?
14. Pomoću koje metode se radi provera da li je stvarni rezultat jednak očekivanom?
15. Ukoliko test nije prošao, koji je tip izuzetka koji se izbacuje?

3. TESTIRANJE METODAMA CRNE KUTIJE

Funkcionalno testiranje, poznato takođe pod imenom testiranje metodama crne kutije, jeste oblik testiranja softvera gde unutrašnja struktura, dizajn i implementacija softvera ili softverske jedinice koja se testira nisu poznati testeru. Softver se posmatra kao funkcija koja mapira vrednosti iz ulaznog domena sistema u izlazni (slika 3.1). Pošto implementacija i struktura programa nisu poznati, program se posmatra kao crna kutija (odatle potiče i ime). Jedini izvor informacija koja se koristi za određivanje i dizajn testova jeste specifikacija zahteva programa.



Slika 3.1. Princip testiranja metodama crne kutije

Metodama crne kutije se mogu uočiti pogrešno implementirane ili nedostajuće funkcionalnosti, greške u ponašanju programa, kao i problemi sa performansama sistema. Na primer, tester može da testira veb sajt pomoću veb čitača, bez poznavanja interne strukture samog veb sajta, pružajući ulazne podatke u vidu klikova mišem ili kucanjem na tastaturi i poredeći ih sa očekivanim izlazom sistema uz praćenje vremena odziva i performansi sistema.

Funkcionalno testiranje se može primeniti na svim nivoima testiranja, počev od jediničnog, preko integracionog do sistemskog testiranja. Razlika u primeni metoda crne kutije u zavisnosti od nivoa testiranja je samo u tome da što je viši nivo testiranja, crna kutija je veća i kompleksnija.

Jedna od glavnih prednosti testiranja metodama crne kutije jeste da su razvijeni testovi nezavisni od konkretne implementacije programa pošto su bazirani isključivo na specifikaciji sistema. Samim tim testovi su upotrebljivi čak i u slučaju promene implementacije. Sa druge strane, pošto se testovi pišu na osnovu specifikacije sistema, oni se mogu razvijati paralelno sa razvojem programa, čime se generalno štedi vreme. Zapravo, testovi se mogu dizajnirati odmah nakon završetka specifikacije sistema. Na primer, ukoliko se implementira kalkulator, nije potrebno da se čeka završetak implementacije funkcije sabiranja. Na osnovu specifikacije (u ovom slučaju zakon sabiranja brojeva u matematici), poznato je kako treba da funkcioniše metoda saberi u kalkulatoru koji se razvija, i može se razviti test (na primer, jedan test bi bio da kada se

metoda saberi pozove sa dva celobrojna parametra 2 i 6, očekivani rezultat treba da bude 8). Još jedna prednost je da se na ovaj način testovi izvršavaju sa korisničke tačke gledišta, i pomažu u otkrivanju odstupanja implementacije od specifikacije sistema.

Postoje i određene mane ovog pristupa, koje se prvenstveno odnose na to da postoji mogućnost da se deo implementacije i neke putanje kroz program uopšte ne pokriju testovima – na primer, implementirane su i neke funkcionalnosti koje se ne nalaze u specifikaciji sistema. Najveći problem može predstavljati nejasna ili nekompletna specifikacija softvera (što se u praksi veoma često dešava, pošto krajnji korisnik često ne ume dobro da formuliše šta mu tačno treba), čime se značajno otežava dizajn testova. Dodatno, može se desiti da postoji redundantnost testova.

Najbitnije tehnike testiranja crne kutije su:

- Klase ekvivalencije
- Analiza graničnih vrednosti
- Tabela odlučivanja i uzročno-posledični graf
- Testiranje zasnovano na modelu stanja

Ove tehnike su detaljno objašnjene u nastavku ovog poglavlja.

3.1. Klase ekvivalencije

Podela ulaznih podataka na klase ekvivalencije je jedna od osnovnih metoda crne kutije. Može se primeniti na bilo kom nivou testiranja, i najčešće je dobar izbor odakle početi testiranje. Glavna ideja kod ove tehnike je da se ulazni domen podataka podeli na takav način da se program ponaša na isti način za sve ulazne vrednosti koje pripadaju istoj klasi ekvivalencije. Drugim rečima, sistem na isti način obrađuje sve vrednosti koje pripadaju istoj klasi ekvivalencije.

Osnovna pretpostavka od koje se polazi jeste da se skup koji predstavlja domen ulaznih podataka može podeliti u disjunktne podskupove, koji se nazivaju klase ekvivalencije. Da bi se obezbedila kompletnost, ceo skup ulaznih podataka se može predstaviti unijom svih podskupova. Pošto se pretpostavlja da će softver na isti način tretirati sve vrednosti iz iste klase ekvivalencije, dovoljno je iz svake klase testirati jednu reprezentativnu vrednost. Ukoliko se softver ponaša korektno pri testiranju jedne reprezentativne vrednosti, moguće je pretpostaviti da će i sve druge vrednosti iz te klase ekvivalencije softver obrađivati bez grešaka, pa nema previše smisla dodatno ih testirati, čime se izbegava redundansa.

Prilikom određivanja klasa ekvivalencije, posmatraju se svi uslovi vezani za ulazne podatke programa koji su definisani u specifikaciji programa, poput tipova promenljivih, dozvoljenog opsega vrednosti i slično. Za svaki uslov se uočavaju dve grupe klasa ekvivalencije – legalne i nelegalne. Legalne klase obuhvataju dozvoljene vrednosti, dok nelegalne klase obuhvataju sve ostale vrednosti ulaznih podataka. Posmatra se ulazni

uslov programa koji je definisan opsegom dozvoljenih vrednosti, na primer, broj između 1 i 10. Definišu se jedna legalna ($1 \leq \text{broj} \leq 10$) i dve nelegalne klase ekvivalencije, ($\text{broj} < 1$) i ($\text{broj} > 10$), kao što je prikazano na slici 3.2.



Slika 3.2. Određivanje klasa ekvivalencije za slučaj dozvoljenog opsega vrednosti

Za testiranje bi bilo dovoljno uzeti jednu reprezentativnu vrednost iz svake od klasa ekvivalencije. Na primer, bilo bi dovoljno testirati ponašanje programa za ulazne vrednosti -5 i 20 (pokrivaju nelegalne klase ekvivalencije 1 i 3, respektivno) i za ulaznu vrednost 6 (pokriva legalnu klasu ekvivalencije 2).

Posmatrajmo kao primer račun u banci koji ima različite stope kamate u zavisnosti od stanja na računu. Kako bi se testiralo da li softver ispravno izračunava kamatu, mogu se identifikovati rasponi stanja na računu i posmatrati kako se kamata izračunava unutar tih raspona. Na primer, u specifikaciji je definisano da za stanje na računu između 0 i 100\$ kamatna stopa iznosi 3%. Ukoliko je stanje na računu između 100\$ i 1000\$, kamatna stopa iznosi 5%, a ukoliko je stanje na računu preko 1000\$, kamatna stopa iznosi 7%. Mogu se uočiti tri legalne klase ekvivalencije i jedna nelegalna klasa, kao što je prikazano na slici 3.3.

Nelegalna klasa	Legalna klasa 1 (3% kamate)	Legalna klasa 2 (5% kamate)	Legalna klasa 3 (7% kamate)
-0.01	0.00	100.00	1000.00

Slika 3.3. Određivanje klasa ekvivalencije za primer računanja kamatne stope

Na ovom primeru se vidi da su identifikovane 4 klase ekvivalencije, iako specifikacija pominje samo tri. Tester ima važan zadatak da, pored testiranja stvari koje su definisane u specifikaciji, uoči i identifikuje stvari koje nisu specificirane, u ovom slučaju situaciju kada je stanje na računu manje od 0. Dodatno, nije identifikovana nelegalna klasa sa desne strane, ali bi trebalo razmotriti da li postoji, odnosno gde se klasa sa 7% završava. Trebalo bi znati koliki je maksimalni iznos na računu (ukoliko postoji). Takođe, svaki nenumerički ulazni podatak bi bio nelegalna klasa ekvivalencije (na primer slovo 'a'). Za sada se posmatraju samo numeričke particije. Prilikom definisanja klasa ekvivalencije je takođe napravljena pretpostavka o najmanjoj mogućoj razlici između dve vrednosti – pretpostavljeno je da su u pitanju 2 decimalna mesta

(99.99 i 100.00 na primer). Isto tako, moglo je da se pretpostavi 0 decimalnih mesta (100), ili 4 decimalna mesta (100.0000). Ove pretpostavke je obavezno zabeležiti kako bi se obezbedila transparentnost i kako bi mogle biti proverene i ispravljene u slučaju da su pogrešne.

Dizajniranje konkretnih testova za ovaj softver se svodi na to da se sve tri legalne klase ekvivalencije moraju pokriti sa jednom reprezentativnom vrednošću. Nelegalna klasa ekvivalencije se takođe mora pokriti bar jednim testom. Na ovom primeru, moglo bi da se testira izračunavanje kamate nad balansima -10.00, 20.00, 250.00 i 1400.00, koji pokrivaju sve klase. Da nisu inicijalno identifikovane sve klase ekvivalencije, veoma je moguće da bi testiranje neke od ovih klasa bilo propušteno, a da bi neka druga bila testirana više puta. Neiskusni tester bi mogao da pomisli da bi dobar skup testova bio da se testira na svakih 50\$. Testovi bi bili 50.00, 100.00, 150.00, 200.00, 250.00 ... recimo sve do 800.00, gde bi se tester umorio i pomislio da je sproveo dovoljno testova. Ovaj naivni pristup bi pokrio samo dve od četiri particije. Ukoliko sistem pogrešno tretira negativnu vrednost ili balans preko 1000\$, greške ne bi bile pronađene, pa je ovakav pristup manje efikasan od podele na klase ekvivalencije, pogotovo što je izvršeno četiri puta više testova (16 naspram četiri testa upotrebom klasa ekvivalencije).

Ukoliko je neki ulazni podatak definisan fiksnim brojem karaktera, na primer broj lične karte građana koji ima tačnu dužinu od 9 karaktera, moguće je uočiti 3 klase ekvivalencije. Legalna klasa u ovom slučaju su brojevi dužine tačno 9 karaktera ($\text{length} = 9$), dok dve nelegalne klase predstavljaju brojeve koji su kraći ($\text{length} < 9$) ili duži ($\text{length} > 9$) od dozvoljene dužine unosa.

Ukoliko postoji situacija da ulazni podatak treba da uzme vrednost iz nekog konačnog nabrojivog skupa, na primer četiri vrednosti $\{x, y, z, w\}$, pri čemu svaku vrednost program različito tretira, potrebno je napraviti po jednu klasu za svaku dozvoljenu vrednost ulaza (u našem slučaju četiri klase). Kao nelegalna klasa ekvivalencije pravi se jedna klasa za sve ulazne vrednosti van dozvoljenog skupa (na primer a).

Ukoliko se posmatraju nizovi (po pitanju njihove dužine), mogu se uočiti tri klase ekvivalencije. Jedna legalna sadrži sve legalne nizove, i dve nelegalne, jedna koja sadrži samo prazan niz i druga koja sadrži niz veći od očekivane dužine.

3.1.1. Višedimenzionalno particionisanje

Neka se kao primer posmatra program koji uzima kao ulaz dva cela broja, uz uslove koji su zadati u specifikaciji: $3 \leq x \leq 7$ i $5 \leq y \leq 9$. Ako se svaki uslov posmatra pojedinačno i tako definišu klase ekvivalencije, onda je u pitanju jednodimenzionalno particionisanje. Uočavaju se sledeće klase ekvivalencije:

E1: $x < 3$

E2: $3 \leq x \leq 7$

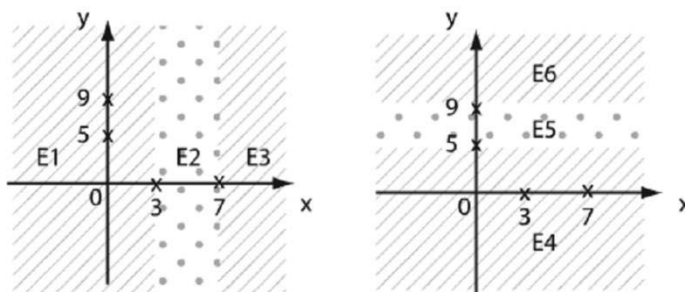
E3: $x > 7$

E4: $y < 5$

E5: $5 \leq y \leq 9$

E6: $y > 9$

Grafički prikaz ovih klasa ekvivalencije je dat na slici 3.4. Legalne klase ekvivalencije su E2 za ulaznu promenljivu x, i E5 za ulaznu promenljivu y.



Slika 3.4. Grafički prikaz jednodimenzionalnog particionisanja

Ukoliko se sada posmatraju kombinacije klasa ekvivalencije ova dva uslova, dolazi se do pojma višedimenzionalnog particionisanja. Sada postoji 9 klasa ekvivalencije, od kojih je samo jedna legalna, E5. Na slici 3.5 je dat grafički prikaz višedimenzionalnog particionisanja.

$$E1: x < 3, y < 5$$

$$E2: x < 3, 5 \leq y \leq 9$$

$$E3: x < 3, y > 9$$

$$E4: 3 \leq x \leq 7, y < 5$$

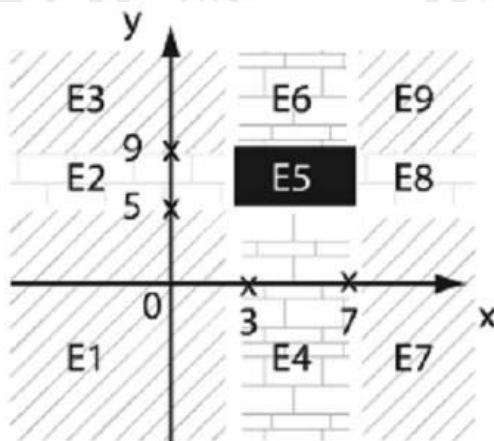
$$E5: 3 \leq x \leq 7, 5 \leq y \leq 9$$

$$E6: 3 \leq x \leq 7, y > 9$$

$$E7: x > 7, y < 5$$

$$E8: x > 7, 5 \leq y \leq 9$$

$$E9: x > 7, y > 9$$



Slika 3.5. Grafički prikaz višedimenzionalnog particionisanja

3.2. Analiza graničnih vrednosti

Programeri veoma često greše u obradi graničnih slučajeva. Statistički gledano, vrlo su česte greške tipa kada se umesto $\geq$ stavi $>$. Testiranje samo metodom podele na klase ekvivalencije ne garantuje da će greške u obradi graničnih slučajeva biti otkrivene. Neka se posmatra najprostiji primer – program koji kao ulaz treba da prihvata samo pozitivne celobrojne vrednosti (računajući nulu). Podela na klase ekvivalencije daje dve klase, legalnu klasu pozitivnih brojeva (računajući i nulu) i nelegalnu klasu negativnih brojeva. U izvornom kodu u tom slučaju postoji uslov oblika:

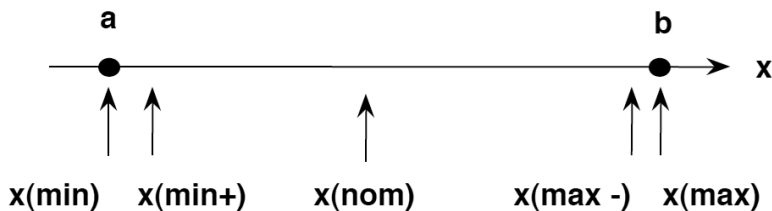
```
if (x < 0 ) throw new IllegalArgumentException( );
```

U slučaju negativnih vrednosti biće izbačen izuzetak i takve vrednosti neće biti obrađene. U slučaju 0 ili pozitivne vrednosti, izuzetak neće biti bačen i takve vrednosti će biti obrađene. Za uočene dve klase ekvivalencije, testovi bi mogli biti -5 za nelegalnu klasu i 10 za legalnu. Međutim, šta se dešava ukoliko programer napravi grešku baš u pisanju uslova, preciznije u proveru da li je ulazni celobrojni podatak manji od 0? Na primer:

```
if (x <= 0 ) throw new IllegalArgumentException( );
```

Sada program ne tretira ispravno ulaznu vrednost 0, u tom slučaju se pogrešno izbacuje izuzetak, a vrednost se ne obrađuje (iako bi trebalo). Testovi za klase ekvivalencije (-5 i 10) će uspešno proći, testiranje će biti završeno sa pozitivnim rezultatom, a greška neće biti uočena. Iz ovog primera se može zaključiti da je neophodna detaljna provera na samim granicama klasa ekvivalencije. Testiranje metodom klasa ekvivalencije se zbog toga najčešće kombinuje sa testiranjem graničnih vrednosti. U najširem smislu, testovi se biraju tako što se prvo odrede granice različitih klasa ekvivalencije, i zatim uzme po jedan test za svaku graničnu vrednost.

Testiranje graničnih vrednosti se može na osnovu gore navedenog definisati kao testiranje granica između klasa ekvivalencije. Posmatraju se kako validne legalne vrednosti (u legalnim klasama ekvivalencije), tako i nelegalne granične vrednosti (u nelegalnim klasama ekvivalencije). U opštem slučaju, ukoliko ulazna vrednost može da bude u intervalu $a - b$, potrebno je testirati minimum, prvu vrednost iznad minimuma, nominalnu vrednost, prvu vrednost ispod maksimuma i maksimalnu vrednost, kao što je prikazano na slici 3.6.



Slika 3.6. Određivanje graničnih vrednosti

Kao primer neka se posmatra štampač koji ima opciju da se definiše broj kopija koje je potrebno napraviti, a dozvoljen interval je od 1 do 99. Postoje tri klase ekvivalencije, jedna legalna i dve nelegalne, koje daju 3 testa. Ako se primeni analiza graničnih vrednosti, uočavaju se minimalna i maksimalna vrednost (granične vrednosti) legalne klase ekvivalencije (brojevi između 1 i 99), zajedno sa prvim susednim vrednostima iz obe nelegalne klase ekvivalencije koje se nalaze sa obe strane legalne klase. U ovom slučaju to su vrednosti 1 i 99 iz legalne klase, i vrednosti 0 i 100 kao prve vrednosti iz susednih nelegalnih klasa. Time se dobija još 4 testa uz 3 testa za klase ekvivalencije. Po još strožijem kriterijumu, mogu se posmatrati granice između klasa, kao i prve susedne vrednosti sa obe strane same granice. U ovom primeru, za granicu 1 trebalo bi osim same granične vrednosti 1 testirati i prvi susedni broj sa svake strane, odnosno 0 i 2. Za granicu 99 trebalo bi dodatno testirati i vrednosti 98 i 100. Ukoliko se primeni strožiji kriterijum, imamo 6 testova za granične vrednosti (po tri za svaku graničnu vrednost) i 3 testa za klase ekvivalencije.

Neka se sada posmatra primer računanja kamate na osnovu stanja na računu, opisan u odeljku 3.1. prilikom opisivanja klasa ekvivalencije. U specifikaciji je definisano da za stanje na računu između 0 i 100\$ kamatna stopa iznosi 3%. Ukoliko je stanje na računu između 100\$ i 1000\$, kamatna stopa iznosi 5%, a ukoliko je stanje na računu preko 1000\$, kamatna stopa iznosi 7%. Klase ekvivalencije su prikazane na slici 17. Mogu se uočiti sledeće granične vrednosti po strožijem kriterijumu: -0.01 (nelegalna granična vrednost, jer se nalazi u nelegalnoj klasi ekvivalencije), 0.00, 0.01, 99.99, 100.00, 100.01, 999.99, 1000.00 i 1000.01 (sve legalne granične vrednosti). Uz četiri testa za klase ekvivalencije, dobija se devet testova za granične vrednosti.

U ovom primeru nije definisana gornja granica za kamatnu stopu od 7%. Situacija kada jedna strana klase ekvivalencije ostane otvorena (odnosno nedefinisana) naziva se otvorena granica. U praksi se i ova granica mora testirati, uz odgovarajući pristup. Na primer, može se pogledati da li je u specifikaciji sistema negde definisana maksimalna vrednost stanja na računu – ukoliko jeste, to je gornja granica. Ukoliko nije definisana, postoje i drugi pristupi, poput provere veličine polja koje drži vrednost stanja u sistemu. Ukoliko ovo polje sadrži samo šest cifara plus dve decimale, maksimalno stanje na računu bi bilo 999 999.99, pa bi se to moglo iskoristiti kao gornja granica.

Granične vrednosti se mogu primeniti i u drugim slučajevima, na primer na podatku tipa String, koji može da označava ime ili adresu maksimalne dozvoljene dužine 30 karaktera. Broj karaktera u stringu se može posmatrati kao klasa ekvivalencije. Legalna klasa ekvivalencije su stringovi dužine 1-30 karaktera, sa legalnim graničnim vrednostima 1 i 30. Nelegalne granične vrednosti bi bile 0 i 31 karakter, i obe ove vrednosti bi trebalo da izazovu poruku o grešci. Ako posmatramo sistem koji prihvata lozinke (podatak tipa string) dužine od 6 do 10 alfanumeričkih karaktera, granične vrednosti su prikazane na slici 3.7.



Slika 3.7. Granične vrednosti za alfanumeričku lozinku dozvoljene dužine 6-10 karaktera

Sa slike se vidi da su granične vrednosti za dužinu lozinke 6 i 10 (legalne), i 5 i 11 (nelegalne – prekratka i predugačka lozinka). Mogu se uočiti i granične vrednosti za svaki pojedinačni karakter. U programskom jeziku Java karakteri su označeni po ASCII formatu. Pošto su dozvoljeni samo alfanumerički karakteri, donja legalna granica za kod karaktera je broj 0 (decimalni kod 48 u ASCII formatu), a gornja legalna granica je karakter 'z' (decimalni kod 122 u ASCII formatu). Nelegalne granične vrednosti su karakter '/' (decimalni kod 47, prenisoko) i karakter '{' (decimalni kod 123, previsoko).

Veoma često se u programu koriste podaci koji označavaju datum. Svaki deo datuma ima posebna ograničenja, i neophodno ih je testirati na odgovarajući način. Neka se posmatra datum u formatu MM/DD/YY. Na slici 3.8 su prikazane granične vrednosti za pojedinačne delove datuma.

MM/DD/YY

Mesec



Dan



Maksimalni broj dana zavisi od meseca u 11 slučajeva, i od godine u jednom slučaju

Godina



Minimum i maksimum za godinu zavise od sistema, aplikacije itd.

Slika 3.8. Granične vrednosti za pojedinačne delove datuma

Testiranje metodom analize graničnih vrednosti se skoro uvek kombinuje sa klasama ekvivalencije. Prvo se određuju klase ekvivalencije, i za svaku klasu se uzima jedna reprezentativna vrednost kao test. Pošto je neophodno posvetiti posebnu pažnju granicama između klasa, kao granični testovi se uzimaju sama granica, kao i prva susedna vrednost iz obe klase ekvivalencije koje ta granica deli.

3.3. Tabela odlučivanja i uzročno-posledični graf

Do sada opisane tehnike podele na klase ekvivalencije i analize graničnih vrednosti su fokusirane na korisnički interfejs i primenjuju se u specifičnim situacijama kada je odgovor sistema jasno definisan na osnovu ulaznih vrednosti. Ponekad, međutim, različite kombinacije ulaznih podataka kao rezultat imaju različite akcije koje sistem preduzima, a to se teško može prikazati pomoću podele na klase ekvivalencije i analize graničnih vrednosti. Testiranje metodom tabele odlučivanja i uzročno-posledičnog grafa se, umesto na korisnički interfejs, fokusira na poslovnu logiku i pravila definisana u specifikaciji, i predstavlja veoma dobar način da se pristupi testiranju kombinacije različitih ulaza.

Tabela odlučivanja je usko povezana sa pojmom uzročno-posledičnog grafa. Ovaj graf služi za vizuelno predstavljanje veze između ulaza programa (uzroka) i izlaznih podataka (posledica) pomoću logičkih relacija predstavljenih izrazima iz Bulove algebre. Na osnovu grafa se biraju različite kombinacije ulaznih vrednosti za testiranje, uz upotrebu različitih heuristika kako bi se sprečila kombinatorna eksplozija pri generisanju testova.

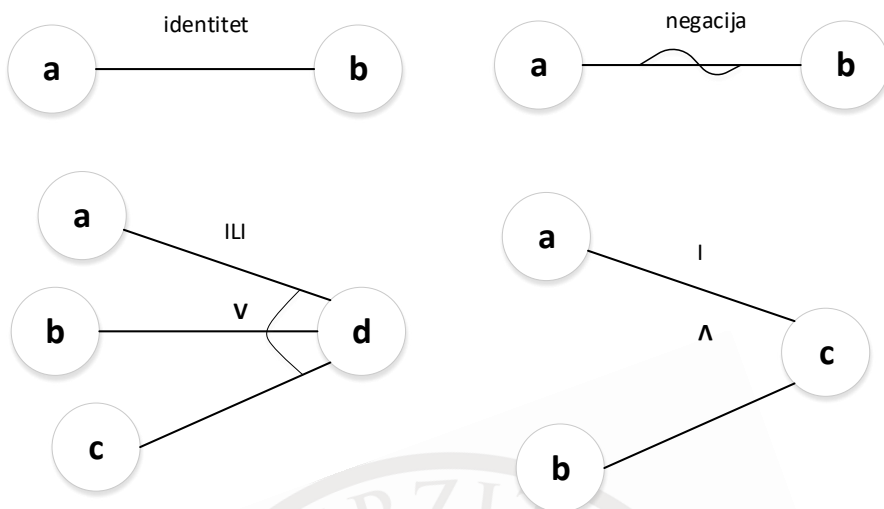
Pomoću tabele odlučivanja se na sistematičan način mogu izraziti kompleksna poslovna pravila, što značajno olakšava posao testerima. Testiranje kombinacija ulaznih parametara je veliki izazov, pošto je broj mogućih kombinacija često ogroman. Testiranje svih mogućih kombinacija je često nemoguće, a u najmanju ruku nepraktično. U praksi se mora zadovoljiti testiranjem malog podskupa kombinacija, zbog čega je veoma važno napraviti dobar izbor kombinacija koje će biti testirane, a koje kombinacije će biti izostavljene. Neophodno je postojanje sistematičnog načina odabira kombinacija, pošto nasumični podskup kombinacija ulaznih parametara nije ni efikasan ni delotvoran.

Prvi zadatak u ovoj tehnici je da se identifikuje funkcija sistema (ili podsistema) koja reaguje na različite kombinacije ulaza ili događaja. Podsystem koji se posmatra ne bi trebalo da sadrži previše ulaznih podataka, jer bi u suprotnom broj mogućih kombinacija bio nepraktičan. Često je bolje podeliti specifikaciju programa u jedinice koje se nezavisno testiraju, jer bi u slučaju svakog složenijeg programa uzročno-posledični graf bio previše složen i nepraktičan. Nakon toga se identifikuju uzroci i posledice na osnovu specifikacije dela sistema koja se posmatra. Posledice su zapravo efekti ulaznih parametara na ponašanje sistema.

Kada se završi sa analizom specifikacije podsistema, pristupa se formiranju uzročno-posledičnog grafa. Uočeni uzroci i posledice predstavljaju početne odnosno završne čvorove grafa, a veze između njih se ostvaruju preko logičkih operatora koji predstavljaju grane ovog grafa. Ukoliko je potrebno, moguće je ubaciti i međučvorove koji će olakšati formiranje logičkih kombinacija ulaznih uslova. Logičke operacije (grane grafa) mogu biti:

- Identitet - ukoliko čvor a ima vrednost 1, onda i čvor b ima vrednost 1, u suprotnom ima vrednost 0
- Negacija – ukoliko čvor a ima vrednost 1, čvor b ima vrednost 0 i obrnuto
- Ili – ukoliko bar jedan od ulaznih čvorova ima vrednost 1, izlazni čvor će takođe imati vrednost 1, u suprotnom će imati vrednost 0
- I – ukoliko svi ulazni čvorovi imaju vrednost 1, onda će i izlazni čvor imati vrednost 1, u suprotnom će imati vrednost 0.

Grafički prikaz ovih osnovnih logičkih relacija je dat na slici 3.9.

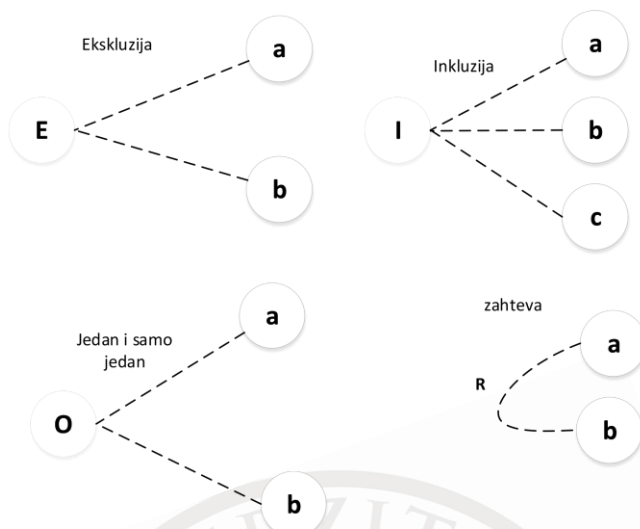


Slika 3.9. Osnovne logičke relacije u uzročno-posledičnom grafu

Nakon kreiranja grafa, pristupa se analizi ograničenja i dopunjavanju grafa. Ograničenja definišu one kombinacije između uzroka i između posledica koje su nemoguće. Moguća ograničenja među uzrocima su:

- Ekskluzija (E – engl. *exclusive*) – najviše jedan od čvorova može da bude 1
- Inkluzija (I – engl. *inclusive*) – bar jedan od čvorova mora da ima vrednost 1
- Jedan i samo jedan (O – engl. *one and only one*) – jedan i samo jedan čvor može imati vrednost 1
- Zahteva (R – engl. *requires*) – da bi čvor a imao vrednost 1, i čvor b mora biti 1

Grafičke oznake ovih ograničenja su prikazane na slici 3.10. Ograničenje takođe može važiti i za posledice, na primer da ukoliko završni čvor a ima vrednost 1, onda čvor b mora imati vrednost 0 (oznaka M – a maskira b).



Slika 3.10. Ograničenja u uzročno-posledičnom grafu

Posmatra se sledeći primer. Neka je dat deo specifikacije softvera koji služi za izračunavanje premije osiguranja kola. U specifikaciji su data sledeća pravila:

R00101 Za žene mlađe od 65 godina, premija je \$500.

R00102 Za muškarce mlađe od 25 godina, premija je \$3000.

R00103 Za muškarce između 25 i 64 godine, premija je \$1000.

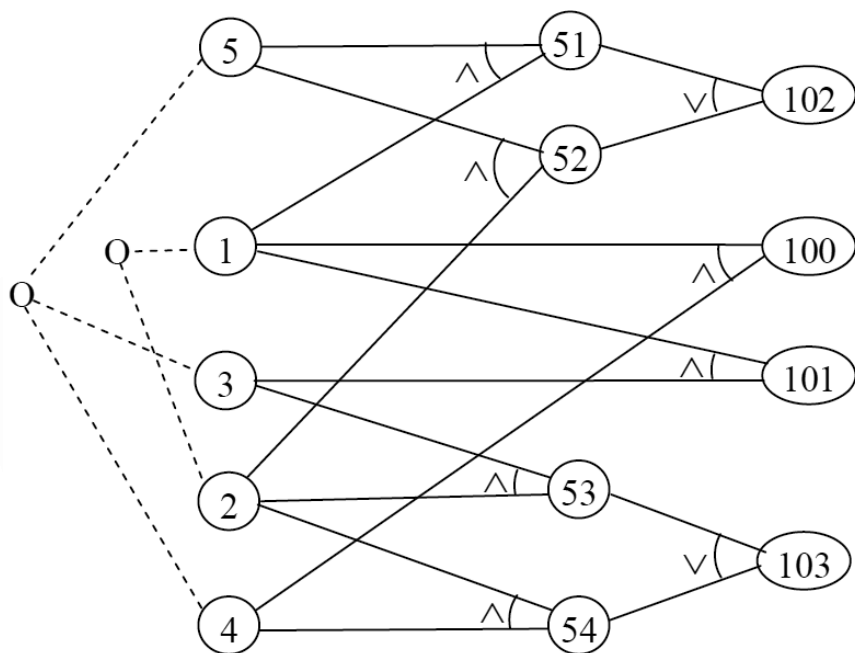
R00104 Za sve starosti 65 godina i više, premija je \$1500.

Kada se analizira ova specifikacija, uočavaju se dve promenljive koje određuju koliko će premija biti: pol i godine korisnika. Postoji pet jasno određenih uzroka: pol je muški, pol je ženski, godine su manje od 25, godine su između 25 i 64, i godine su 65 i preko. Postoje i četiri jasno određene posledice: premija može biti 500, 1000, 1500 ili 3000. Na slici 3.11 se nalazi tabelarni prikaz ovih uzroka i posledica. Svakom uzroku i posledici je dodeljen proizvoljan broj radi lakšeg crtanja uzročno-posledičnog grafa.

Uzroci	Posledice
1. Pol je muški	100. Premija je 1000
2. Pol je ženski	101. Premija je 3000
3. Godine su < 25	102. Premija je 1500
4. Godine su >= 25 i <65	103. Premija je 500
5. Godine su >= 65	

Slika 3.11. Uzroci i posledice

Na osnovu tabele se mogu uočiti i neka ograničenja između uzroka. Postoje jedan i samo jedan ograničenja između uzroka 1 i 2, kao i između uzroka 3, 4 i 5. Sada se pristupa crtanju uzročno-posledičnog grafa, na osnovu gore opisanog postupka. Prvo se crtaju uzroci i posledice kao početni i završni čvorovi grafa, a zatim i veze između njih preko logičkih operatora. Uzročno-posledični graf je prikazan na slici 3.12. Uzroci su nacrtani sa leve strane, posledice sa desne strane, i označeni su brojevima iz tabele sa slike 3.11. Između uzroka 1 i 2, kao i uzroka 3, 4 i 5 se nalazi O ograničenje (jedan i samo jedan).



Slika 3.12. Uzročno-posledični graf

Radi lakšeg prikazivanja relacija uvedeni su međučvorovi 51, 52, 53 i 54, koji imaju sledeće značenje:

- Čvor 51 predstavlja I relaciju između uzroka 1 i 5, odnosno pol je muški i godine su veće ili jednako 65.
- Čvor 52 predstavlja I relaciju između uzroka 2 i 5, odnosno pol je ženski i godine su veće ili jednako 65.
- Čvor 53 predstavlja I relaciju između uzroka 2 i 3, odnosno pol je ženski i godine su manje od 25.
- Čvor 54 predstavlja I relaciju između uzroka 2 i 4, odnosno pol je ženski i godine su između 25 i 64.

Čvorovi koji predstavljaju posledice su čvorovi 100, 101, 102 i 103, i imaju sledeće značenje:

- Čvor 100 koji označava premiju u vrednosti 1000 se dobija kao I relacija uzroka 1 i 4, odnosno označava muškarce koji imaju između 25 i 64 godina.
- Čvor 101 koji označava premiju u vrednosti 3000 se dobija kao I relacija uzroka 1 i 3, odnosno označava muškarce koji imaju manje od 25 godina.
- Čvor 102 koji označava premiju u vrednosti 1500 se dobija kao ILI relacija međučvorova 51 i 52, odnosno pol je muški i više od 65 godina, ILI pol je ženski i više od 65 godina. U stvari, označava sve koji imaju više od 65 godina (muško ili žensko).
- Čvor 103 koji označava premiju u vrednosti 500 se dobija kao ILI relacija međučvorova 53 i 54, odnosno pol je ženski i godine su manje od 25, ILI pol je ženski i godine su između 25 i 64. U stvari, označava žene mlađe od 65 godina.

Na osnovu uzročno-posledičnog grafa može se izvesti tabela odlučivanja, koja predstavlja dvodimenzionalno mapiranje uzroka na posledice. Tabela odlučivanja ima po jedan red za svaki uzrok i za svaku posledicu, a kolone odgovaraju konkretnim testovima. Tabela se popunjava upotrebom sledećih pravila:

- Završni čvorovi se, jedan po jedan, postavljaju na vrednost 1.
- Kreće se unazad od završnog čvora kroz uzročno-posledični graf i uočavaju sve kombinacije ulaznih čvorova koje ga postavljaju na 1, pri čemu se sve vreme vodi računa o postojećim ograničenjima.
- Na osnovu svake kombinacije ulaza koja postavlja posmatrani završni čvor na 1 se formira jedna kolona u tabeli odlučivanja. Ukoliko vrednost nekog od ulaznih čvorova nije bitna ili se podrazumeva, taj ulaz ostaje prazan u tabeli odlučivanja.
- Za svaku od gornjih kombinacija se određuje i efekat koji ima na stanje ostalih završnih čvorova – korisno za fazu testiranja programa pomoću generisanih testova, jer se unapred određuju efekti koji se očekuju pri izvršavanju programa.
- U slučaju da postoje kolone u tabeli odlučivanja koje se podudaraju, one se redukuju. Podudaranje se u ovom slučaju odnosi na kolone kojima su vrednosti po svim čvorovima jednake ili ukoliko je na određenim pozicijama vrednost u jednoj koloni nije bitna, a u drugoj jasno određena – tada se uzima kolona sa fiksiranom vrednosti.

Kako bi se broj kombinacija koje se podudaraju smanjio, moguće je primeniti neke olakšice. Kada se ide unazad kroz ILI čvor, čiji izlaz treba da bude 1, nikada se ne postavlja više od jednog ulaza na 1 (da bi se izbegao problem da se ne otkrije greška ukoliko jedan ulaz maskira drugi). Kada se ide unazad kroz I čvor, čiji izlaz treba da bude 0, treba proveriti sve kombinacije ulaza koje dovode taj izlaz na 0. Međutim, za

one kombinacije u kojima su neki od ulaza na vrednosti 1, nije potrebno ispitati sve kombinacije koje taj ulaz dovode na vrednost 1. Kada se ide unazad kroz I čvor, čiji izlaz treba da bude 0, za slučaj kada svi ulazi imaju vrednost 0, nije potrebno ispitati sve kombinacije koje dovode svaki od ulaza na vrednost 0, već je dovoljna samo po jedna kombinacija. Za više detalja o redukovanju i generalno heuristikama koje se primenjuju prilikom prevođenja uzročno-posledičnog grafa u tabelu odlučivanja, čitalac se upućuje na literaturu (2. Myers G., The Art of Software Testing). Tabela odlučivanja za dati primer je prikazana na slici 3.13.

Uzroci/posledice	1	2	3	4	5	6
Uzroci						
1. Pol je muški	1	1	1	0	0	0
2. Pol je ženski	0	0	0	1	1	1
3. Godine su < 25	1	0	0	0	1	0
4. Godine su >= 25 i <65	0	1	0	0	0	1
5. Godine su >= 65	0	0	1	1	0	0
Posledice						
100. Premija je 1000	0	1	0	0	0	0
101. Premija je 3000	1	0	0	0	0	0
102. Premija je 1500	0	0	1	1	0	0
103. Premija je 500	0	0	0	0	1	1

Slika 3.13. Tabela odlučivanja

Na osnovu tabele odlučivanja se pristupa kreiranju konkretnih testova. Svaka kolona u tabeli odlučivanja odgovara jednom testu. Na slici 3.14 prikazani su konkretni testovi za tabelu odlučivanja sa slike 3.13.

Testovi	Uzroci		Posledice (očekivani rezultat)
	Pol	Godine	Premija
Test 1	Muški	<25	3000
Test 2	Muški	>= 25 i < 65	1000
Test 3	Muški	>=65	1500
Test 4	Ženski	>=65	1500
Test 5	Ženski	<25	500
Test 6	Ženski	>= 25 i < 65	500

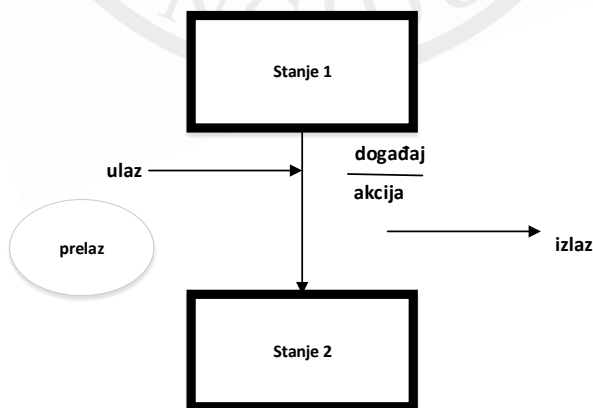
Slika 3.14. Konkretni testovi izvedeni na osnovu tabele odlučivanja

Ukoliko se ova tehnika primenjuje dosledno, za svaku kolonu tabele odlučivanja postojaće jedan test. Prednost ove tehnike je u tome što će kombinacije ulaznih podataka biti pokrivene testovima. Ponekad samo tačno određena kombinacija ulaza može da otkrije grešku u implementaciji, koja ne bi bila otkrivena drugih tehnikama. Međutim, u slučaju postojanja velikog broja ulaznih parametara, nije moguće iscrpno testirati svaku moguću kombinaciju. U tom slučaju je potrebno napraviti prioritete i testirati najvažnije kombinacije. Postojanje potpune tabele odlučivanja pomaže u određivanju koje kombinacije treba testirati, a koje ne.

3.4. Model stanja

Testiranje metodom modela stanja je tehnika crne kutije koja se bazira na specifikaciji ponašanja softvera u obliku konačnog automata. Naravno, tehniku je moguće primeniti samo za onaj softver čije se ponašanje može predstaviti konačnim automatom. Pod konačnim automatom se podrazumeva da se posmatrani sistem može nalaziti u konačnom broju različitih stanja, a prelazi iz jednog stanja u drugo su određeni pravilima automata. U opštem slučaju, svaki sistem gde se, u zavisnosti od toga šta se prethodno dešavalo, dobija različit izlaz sistema za isti ulaz, je konačni automat. Konačni automati se najčešće predstavljaju dijagramom promene stanja.

Sistem može biti u nekom stanju proizvoljno dugo, sve dok se ne desi neki spoljašnji događaj koji će izazvati promenu stanja. Prelaz između stanja je određen trenutnim stanjem u kojem se sistem nalazi i ulaznim događajem. Događaj je spoljašnja pojava koja pokreće prelazak u novo stanje, i može biti izazvan na različite načine, na primer unos preko tastature, dodir na ekran ili klik miša i slično. Akcija je odgovor sistema tokom prelaska između stanja. Na slici 3.15 se nalazi grafički prikaz prelaska između dva stanja.



Slika 3.15. Tranzicija između dva stanja

Neka se posmatra kao primer podizanje novca sa bankomata. Zahtev za podizanje 100 dolara se može izvršiti uspešno, i kao rezultat se dobija novac. Međutim, isti takav zahtev kasnije može biti odbijen jer na računu nema dovoljno novca. Zahtev je odbijen zato što se stanje računa promenilo iz stanja gde postoji dovoljno novca za podizanje, u stanje da nema dovoljno sredstava. Ova promena stanja je prouzrokovana prvom transakcijom koja je prošla uspešno.

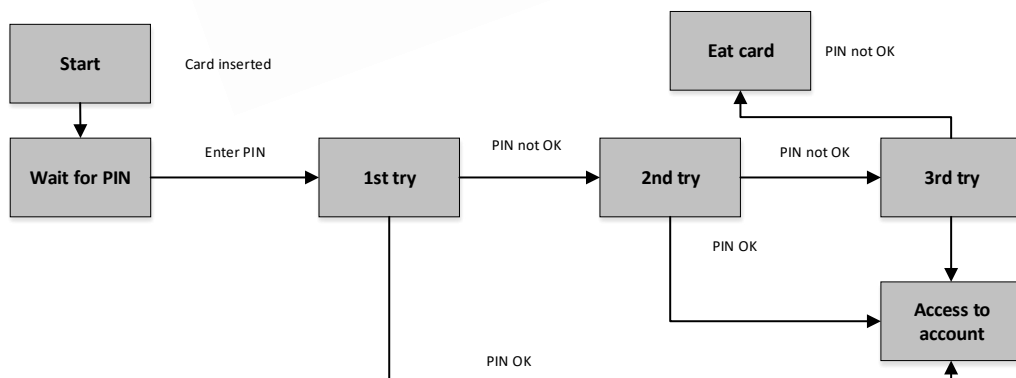
Drugi jednostavan primer je status dokumenta u bilo kom editoru teksta. Ukoliko je dokument otvoren, moguće je zatvoriti ga. Ukoliko nijedan dokument nije otvoren, opcija za zatvaranje dokumenta nije dostupna. Ukoliko se izvrši opcija zatvaranja određenog dokumenta jednom, nije moguće ponovo zatvoriti taj isti dokument osim ako se ponovo ne otvori. Samim tim, dokument ima dva stanja: otvoren i zatvoren.

Model stanja se sastoji od četiri osnovna elementa:

- Stanja u kojima se softver može nalaziti (otvoren/zatvoren dokument, dovoljno/nedovoljno sredstava)
- Prelasci (tranzicije) iz stanja u stanje (nisu svi prelasci dozvoljeni)
- Događaji koji okidaju prelasci (zatvaranje dokumenta, podizanje novca)
- Akcije koje su rezultat prelaska iz stanja u stanje (poruka o grešci, isplata novca)

Može se uočiti da kada se sistem nalazi u bilo kom datom stanju, jedan događaj može izazvati tačno jednu akciju. Događaj podizanja novca, u slučaju da je račun u stanju gde ima dovoljno sredstava za isplatu, će uvek kao akciju imati isplatu novca. Isti taj događaj, u nekom drugom stanju sistema, može izazvati drugačiju akciju i dovesti do drugačijeg krajnjeg stanja. Događaj podizanja novca, u slučaju da je račun u stanju gde nema dovoljno sredstava, će kao akciju imati odbijanje transakcije.

Posmatra se sada primer unosa PIN koda (engl. *Personal Identity Number*) na bankomatu. Na slici 3.16 prikazan je dijagram stanja. Stanja su prikazana kao pravougaonici, dok su tranzicije označene linijama sa strelicama (u pravcu prelaska) i tekstom koji opisuje događaj koji izaziva tranziciju pored linije.



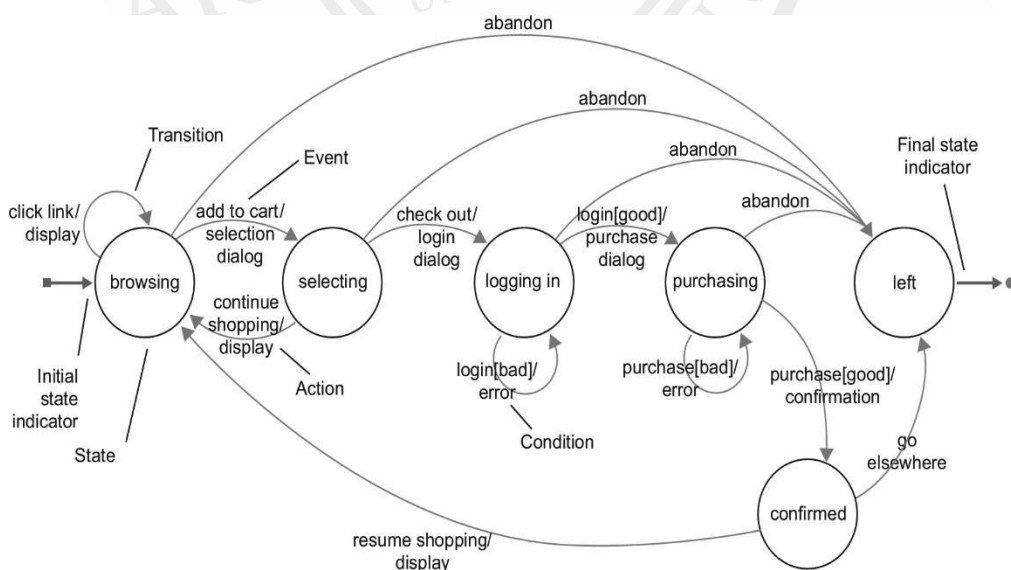
Slika 3.16. Dijagram stanja za unos PIN koda na bankomatu

Dijagram stanja prikazuje sedam stanja, a postoje samo 4 moguća događaja: ubačena kartica, unošenje PIN koda, PIN ispravan i PIN neispravan. Radi jednostavnosti u ovom primeru nisu specificirane sve moguće tranzicije – na primer, izostavljen je događaj tajmaut za čekanje unosa PIN koda u stanju „čekanja PIN koda“, kao i u stanjima za sva tri pokušaja unosa, koji bi nakon isteka definisanog vremena izazivao prelazak u početno stanje i izbacio karticu iz bankomata. Izostavljena je i tranzicija iz stanja „pojedi karticu“. Takođe su, radi jednostavnosti, izostavljeni događaji poput opcije „odustani“ u stanju „čekanja PIN koda“, kao i u stanjima za sva tri pokušaja unosa, koji bi nakon odabira te opcije izazivao prelazak u početno stanje i izbacio karticu iz bankomata.

U procesu definisanja konkretnih testova se najčešće polazi od tipičnog scenarija. U primeru unosa PIN koda, testovi bi bili sledeći:

- Prvi test bi predstavljao tipičnu situaciju, gde korisnik unosi tačan PIN iz prvog pokušaja.
- Drugi test (koji bi pokrio sva moguća stanja) bi bio da korisnik unese netačan PIN tri puta, i da sistem pojede karticu.
- Dalji testovi bi se odnosili na unošenje netačnog PIN koda prvi put i unošenje tačnog PIN koda u drugom pokušaju, kao i unošenje netačnog PIN koda u prva dva pokušaja i unošenje tačnog PIN koda u trećem pokušaju. Ova dva testa su očigledno manje bitna od prva dva testa.

Nešto složeniji primer dijagrama stanja je prikazan na slici 3.17. Posmatra se elektronska kupovina iz perspektive klijenta. Pored svake grane napisan je par događaj/akcija.



Slika 3.17. Dijagram stanja za elektronsku kupovinu iz perspektive klijenta

Sa slike 3.17 se vidi da je moguće da jedno stanje ima prelazak u samog sebe. Na primer, u stanju “logging in”, ako korisnik unese pogrešno korisničko ime i lozinku, akcija sistema će biti da se korisniku prikaže greška i da se vrati ponovo u isto stanje.

Prilikom pisanja konkretnih testova, neophodno je odrediti pokrivenost skupa testova u smislu pokrivenih stanja i pokrivenih tranzicija između stanja. Moguće je takođe posmatrati tranzicije u parovima, u trojkama itd. Kriterijumi za pokrivenost testiranja modelom stanja se definišu na sledeći način:

- Pokrivanje stanja (state cover engl.) – neophodno je da skup testova bude takav da se njegovim izvršavanjem poseti svako stanje u dijagramu stanja bar jedanput.
- Pokrivanje prelaza (transition cover ili 0-switch cover engl.) – skup testova je takav da se svaki prelaz u dijagramu stanja izvršava bar jedanput.
- Pokrivanje promena odnosno parova prelaza (switch cover ili 1-switch cover engl.) – skup testova je takav da se svaki mogući par prelaza (sekvence od dve uzastopne tranzicije) u dijagramu stanja izvršava bar jedanput.
- Pokrivanje dve promene odnosno trojki prelaza (2-switch cover engl.) – skup testova je takav da se svaka moguća trojka prelaza (sekvence od tri uzastopne tranzicije) u dijagramu stanja izvršava bar jedanput.
- Pokrivanje N promena odnosno N+1 prelaza (N-switch cover engl.) – skup testova je takav da se svaka moguća sekvenca od N+1 prelaza u dijagramu stanja izvršava bar jedanput.

Nakon određivanja pokrivenosti koja se želi postići, pristupa se samom pisanju testova. Potrebno je pridržavati se sledećih koraka:

- Usvaja se pravilo gde test mora da počne, kao i gde mora ili može da se završi. Na primer, test mora početi u početnom (inicijalnom) stanju, i završiti u konačnom stanju.
- Od početnog stanja se definiše sekvenca prelaza u obliku događaj/stanje koja dovodi do dozvoljenog završnog stanja. Za svaku tranziciju se beleži očekivana akcija sistema, i to je očekivani rezultat testa.
- Na test dijagramu označiti svako stanje i prelaz iz upravo definisanog testa kao pokrivene.
- Koraci se ponavljaju dok se ne pokriju sva stanja i svi prelazi po odabranom kriterijumu pokrivenosti.

Za dijagram stanja sa slike 3.17 se na osnovu gore navedenih pravila, po kriterijumu pokrivanje prelaza (0-switch) mogu uočiti sledeći testovi.

Test 1: browsing, click link, display, add to cart, selection dialog, continue shopping, display, add to cart, selection dialog, checkout, login dialog, login[bad], error,

login[good], purchase dialog, purchase[bad], error, purchase[good], confirmation, resume shopping, display, abandon, left.

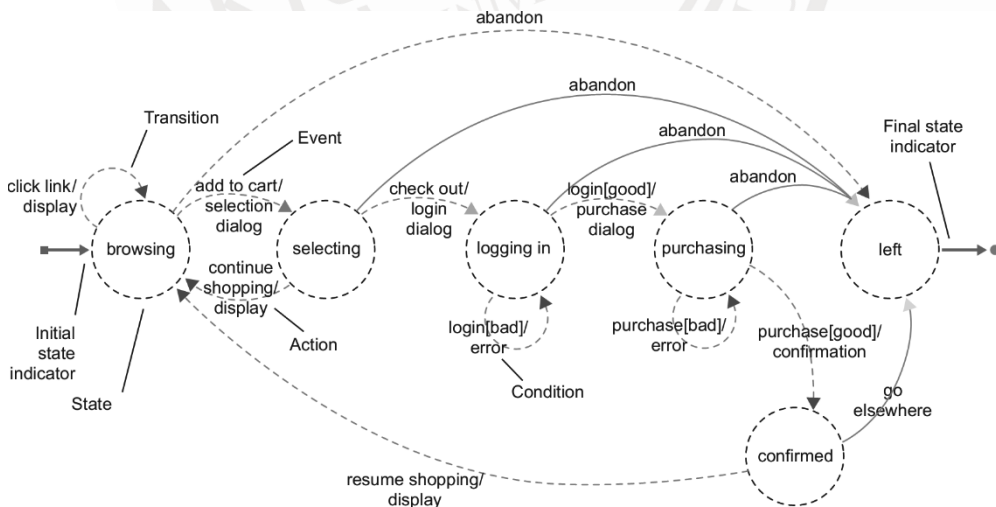
Nakon definisanja testa 1, na test dijagramu se označavaju stanja i prelazi koji su ovim testom pokriveni. Na slici 3.18 pokrivena stanja i prelazi su označeni isprekidanom linijom. Može se uočiti da je već sa ovim jednim testom postignut kriterijum pokrivanje stanja, pošto su sva stanja na dijagramu posećena bar jedanput. Kako bi se postigao željeni kriterijum pokrivanje prelaza (0-switch), moraju se testovima pokriti i svi prelazi koji nisu pokriveni testom 1, a koji su na slici 3.18 iscrtani punom linijom. Sa slike se vidi da nisu pokriveni prelazi iz stanja “selecting”, “logging in” i “purchasing” u stanje “left” odabirom opcije “abandon”, kao i prelaz iz stanja “confirmed” u stanje “left”. Stoga je neophodno napisati još 4 testa kako bi se postigla željena pokrivenost. Uopšteno gledano, potreban je onoliki broj testova koliko ima ulaza u završno stanje, odnosno u našem slučaju je potrebno ukupno 5 testova. Dodatni testovi bi mogli da izgledaju ovako:

Test 2: browsing, add to cart, selection dialog, abandon, <no action>, left.

Test 3: browsing, add to cart, selection dialog, checkout, login dialog, abandon, <no action>, left.

Test 4: browsing, add to cart, selection dialog, checkout, login dialog, login[good], purchase dialog, abandon, <no action>, left.

Test 5. browsing, add to cart, selection dialog, continue shopping, display, add to cart, selection dialog, checkout, login dialog, login[good], purchase dialog, purchase [good], confirmation, go elsewhere, <no action>, left.



Slika 3.18. Dijagram stanja za elektronsku kupovinu sa označenim stanjima i prelazima koje pokriva test 1

Na kraju je potrebno napomenuti da uvedeni pojam pokrivenosti, odnosno merenje koliko je testirana (pokrivena) određena softverska komponenta, spada u pristup testiranja metodom bele kutije. Ipak, testiranje metodom modela stanja se posmatra kao metoda crne kutije. O pokrivenosti će više reći biti u metodama bele kutije.

3.5. Nagađanje grešaka

Tehnike crne kutije opisane do sada su formalne, u smislu da postoje jasna pravila kako se testiranje sprovodi i kako se vrši odabir testova. Svima im je zajedničko da je specifikacija softvera jedini izvor informacija za formiranje testova, odnosno da ne postoji uvid u kod. Postoji još jedna, neformalna tehnika testiranja koju je neophodno pomenuti zbog česte primene u praksi, a to je metoda nagađanja grešaka.

Nagađanje grešaka (engl. *Error guessing*) takođe spada u metode crne kutije. Obično su za nju zaduženi iskusni i dobri testeri, koji se ohrabruju da razmisle o situacijama koje bi mogle da dovedu do otkaza softvera. Neki ljudi su prosto prirodno talentovani za testiranje, dok drugi imaju bogato testersko iskustvo ili dugo rade sa određenim sistemom pa lakše mogu da uoče mane tog sistema. Upravo zbog toga tehnika nagađanja grešaka, kada se primeni nakon testiranja formalnim metodama, može da bude veoma efikasna.

Treba jasno naglasiti da testiranje metodom nagađanja grešaka nikako nije nasumično testiranje, niti da je mogu raditi testeri bez većeg iskustva. Nasumično testiranje nije efikasno, i najčešće dovodi do gubljenja vremena bez nekih značajnijih rezultata. Zbog toga se za primenu ove metode obično podrazumevaju iskusni testeri, jer je u praksi pokazano da njihove pretpostavke pri nagađanju mogu uštedeti dosta vremena i pronaći defekte koji nisu uočeni primenama formalnih metoda. Uspešnost ove tehnike u potpunosti zavisi od sposobnosti testera, jer dobri testeri mogu da pretpostave gde će najverovatnije u softveru biti greške.

Nagađanje grešaka se primenjuje nakon formalnih tehnika iz prostog razloga što će, dok testira softver formalnim tehnikama, tester steći bolje razumevanje sistema, odnosno šta i kako softver radi. Tester će onda moći da bolje pretpostavi gde su potencijalne greške ili situacije u kojima se sistem neće ponašati na očekivan način. Tipične situacije su pokušaj deljenja sa nulom, prazno polje za unos, prazni fajlovi i različite vrste pogrešnih podataka – na primer, alfabetski karakteri u polju za unos na mestu gde se očekuju numerički i slično. Ukoliko postoje prethodne verzije softvera koji se testira, veoma razumna pretpostavka je da se isprobaju situacije sa kojima su prethodne verzije imale probleme. Ukoliko postoji sličan softver – razumno je isprobati i situacije sa kojima su slični programi imali problema. Često se dešava i da se prilikom razvoja softvera kaže da do neke situacije neće nikada doći, da je nemoguće i slično. Takve situacije je neophodno proveriti, jer upravo takve pretpostavke da su neke situacije (na primer kombinacije ulaznih parametara) nemoguće su često i pogrešne, pa dovode do grešaka u radu i otkaza sistema nakon puštanja u produkciju.

3.6. Zadaci za vežbu

Primer 1: Posmatra se ponašanje aplikacije za rezervaciju avionskih karata, za vreme bukiranja novog leta. Fokusira se na broj karata koje korisnik sistema može u jednom trenutku da naruči. U specifikaciji sistema je definisano da su validne vrednosti za broj karata koje jedan korisnik može istovremeno da rezerviše između 1 i 10. Vrednosti veće od 10 se smatraju pogrešnom rezervacijom, i po specifikaciji treba da se pojavi odgovarajuća poruka o grešci. Pretpostavlja se da je cena jedne karte fiksna i iznosi 100 dolara. Potrebno je napisati JUnit testove primenom metoda podele na klase ekvivalencije i analize graničnih vrednosti.

Rešenje:

Prilikom odabira testova, kao jedini izvor informacija koristi se specifikacija sistema. Uočavaju se tri klase ekvivalencije, u zavisnosti od broja karata, prikazane na slici 3.19.

Nelegalna	Legalna	Nelegalna
0	1	10 11
Klasa ekvivalencije 1	Klasa ekvivalencije 2	Klasa ekvivalencije 3

Slika 3.19. Klase ekvivalencije za broj karata koje jedan korisnik može da rezerviše

Neka je data stvarna implementacija metode rezervisiKarte, koja u slučaju da je broj rezervacija u redu vraća izračunatu ukupnu cenu (broj rezervacija pomnožen sa cenom karte), a u suprotnom se izbacuje poruka o grešci. Treba ponovo napomenuti da prilikom odabira testova se ne posmatra implementacija, već samo specifikacija.

```
public class FlightReservation {
    private static int cenaKarte = 100;

    public static int rezervisiKarte (int n) throws
    IllegalArgumentException{
        if (n>10 || n<1) throw new IllegalArgumentException
        ("Nedozvoljen broj karata");
        return n * cenaKarte;
    }
}
```

Testovi za klase ekvivalencije podrazumevaju uzimanje jedne karakteristične vrednosti iz svake uočene klase. Postoje tri testa, jedan za legalnu klasu ekvivalencije (na primer rezervisano 8 karata, očekivana vrednost vraćene izračunate cene je 800), i

dva testa za nelegalne klase (na primer -1 i 55, koji će kao očekivani rezultat dovesti do bacanja izuzetka `IllegalArgumentException`). Za granične vrednosti, po strožijem kriterijumu postoji šest testova. Potrebno je proveriti vrednosti 0, 1, 2, 9, 10 i 11. Vrednosti 0 i 11 treba da dovedu do izbacivanja izuzetka pri pozivu metode sa tim argumentom, dok za vrednosti 1, 2, 9 i 10 se očekuje uspešno izvršavanje metode i vraćanje vrednosti 100, 200, 900 i 1000 respektivno. JUnit testovi su dati u nastavku.

```
import org.junit.Test;
import static org.junit.Assert.*;
public class FlightReservationTest {

    /**
     * Testiranje metodom partitionisanja ulaznog skupa podataka na
     * klase ekvivalencije
     * Dve nelegalne klase, kada je broj karata manji od 1 i veci od 10
     * Jedna legalna, broj karata u intervalu 1 do 10
     */
    @Test
    public void testRezervisiKarteLegalnaKlasa () {
        assertEquals(800, FlightReservation.rezervisiKarte(8));
    }

    @Test (expected = IllegalArgumentException.class)
    public void testRezervisiKarteNelegalnaKlasa1 () {
        FlightReservation.rezervisiKarte(-1);
    }

    @Test (expected = IllegalArgumentException.class)
    public void testRezervisiKarteNelegalnaKlasa2 () {
        FlightReservation.rezervisiKarte(55);
    }

    /**
     * Testiranje metodom granicnih uslova, jaci kriterijum kaze da
     * treba da proverimo 0, 1, 2, 9, 10, 11
     */
    @Test (expected = IllegalArgumentException.class)
    public void testRezervisiKarteG0 (){
        FlightReservation.rezervisiKarte(0);
    }

    @Test
    public void testRezervisiKarteG1 (){
        assertEquals(100, FlightReservation.rezervisiKarte(1));
    }

    @Test
```

```

public void testRezervisiKarteG2 (){
    assertEquals(200, FlightReservation.rezervisiKarte(2));
}

@Test
public void testRezervisiKarteG9 (){
    assertEquals(900, FlightReservation.rezervisiKarte(9));
}

@Test
public void testRezervisiKarteG10 (){
    assertEquals(1000, FlightReservation.rezervisiKarte(10));
}

@Test (expected = IllegalArgumentException.class)
public void testRezervisiKarteG11 (){
    FlightReservation.rezervisiKarte(11);
}
}

```

Primer 2: Posmatra se baza podataka banke, koja ima samo dve dozvoljene komande.

credit	acct_number	transaction_amount
debit	acct_number	transaction_amount

Zahtevi u specifikaciji sistema su definisani na sledeći način. Ukoliko je komanda credit i acct_number je validan, dozvoljava se podizanje novca na kredit. Ukoliko je komanda debit, i acct_number je validan, i transaction_amount je manji od trenutnog balansa, dozvoljava se podizanje novca (debit). Ukoliko komanda, broj računa ili količina novca za debitno podizanje nisu validni, generiše se odgovarajuća poruka o grešci. Potrebno je nacrtati uzročno-posledični graf i iz njega izvesti tabelu odlučivanja.

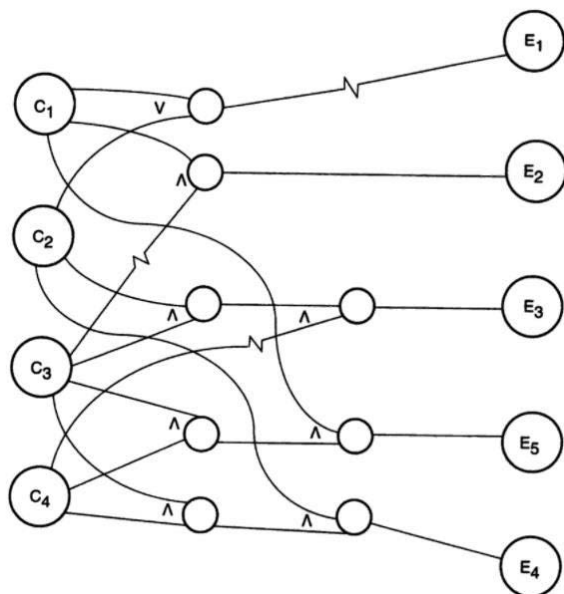
Rešenje:

Prvo se pristupa analizi specifikacije, kako bi se uočili uzroci i posledice, koji su prikazani na slici 3.20.

Uzroci	Posledice
C1: komanda je kredit	E1: Poruka korisniku o grešci
C2: komanda je debit	E2: Poruka da acct_number nije validan
C3: acct_number je validan	E3: Poruka da debit transaction_amount nije validan
C4: transaction_amount je validan	E4: Debit - podizanje novca sa računa
	E5: Kredit - podizanje novca sa računa

Slika 3.20. Uzroci i posledice za bazu podataka banke

Na osnovu uočenih uzroka i posledica, pristupa se crtanju uzročno-posledičnog grafa, na slici 3.21. Za sve posledice se relativno lako može uočiti na koji način zavise od uzroka. Na primer, za posledicu E5 se lako uočava da zavisi od uzroka C2, C3 i C4 i to na takav način da E5 ima vrednost 1 samo u slučaju da su sve tri vrednosti C2, C3 i C4 true (što znači da su povezani sa I uslovom). Na sličan način, posledica E2 direktno zavisi od C3, i to na takav način da E2 ima vrednost 1 samo ukoliko je vrednost C3 false (negacija).



Slika 3.21. Uzročno-posledični graf za bazu podataka banke

Na osnovu uzročno-posledičnog grafa, crta se tabela odlučivanja. Tabela odlučivanja ima po jedan red za svaki uzrok i za svaku posledicu, a kolone odgovaraju konkretnim testovima. Tabela odlučivanja za dati uzročno-posledični graf sa slike 3.21 je data na slici 3.22.

Uzroci/posledice	1	2	3	4	5
Uzroci					
C1: komanda je kredit	0	1	x	x	1
C2: komanda je debit	0	x	1	1	x
C3: acct_number je validan	x	0	1	1	1
C4: transaction_amount je validan	x	x	0	1	1
Posledice					
E1: Poruka korisniku o grešci	1	0	0	0	0
E2: Poruka da acct_number nije validan	0	1	0	0	0
E3: Poruka da debit transaction_amount nije validan	0	0	1	0	0
E4: Debit - podizanje novca sa računa	0	0	0	1	0
E5: Kredit - podizanje novca sa računa	0	0	0	0	1

Slika 3.22. Tabela odlučivanja za bazu podataka banke

Pitanja

1. Šta znači pojam testiranja tehnikama crne kutije?
2. Koje metode crne kutije postoje?
3. Na čemu se bazira testiranje analizom graničnih vrednosti?
4. Kako se biraju klase ekvivalencije?
5. Šta je particionisanje?
6. Ukoliko je ulazni uslov neke vrednosti definisan opsegom dozvoljenih vrednosti između 20 i 99, koje su ispravne klase ekvivalencije?
7. Ukoliko ulaz programa mora da bude u intervalu 1-100, koji su ispravni testovi graničnih vrednosti?
8. Testiranje klasama ekvivalencije se najčešće koristi zajedno sa kojom drugom tehnikom?
9. Na čemu se bazira testiranje uzročno posledičnim grafovima?
10. Koje su moguće logičke funkcije u uzročno posledičnom grafu?
11. Na osnovu čega se formira tabela odlučivanja?
12. Kako se definiše tabela odlučivanja?
13. Šta je konačni automat?
14. Šta je dijagram stanja?
15. Kako se definiše prelaz između stanja?
16. Kako se postiže pokrivanje stanja u modelu stanja?
17. Šta je 0-switch pokrivanje stanja?
18. Na čemu se zasniva testiranje metodom nagađanja grešaka?

4. TESTIRANJE METODAMA BELE KUTIJE

Strukturno testiranje, još poznato i kao testiranje metodama bele ili staklene kutije, je tehnika testiranja u kojoj je stvarna implementacija softvera poznata testerima. Dok se u tehnikama crne kutije tester fokusira na to šta sistem radi, u metodama bele kutije fokus je na tome kako sistem radi. Cilj testiranja je da se izvrše sve programske strukture, kao i strukture podataka u programu. Specifikacija se ne proverava prilikom primene metoda bele kutije, već se posmatra samo kod. Strukturno testiranje se može primeniti na svim nivoima testiranja (jedinično, integraciono, sistemsko), i najčešće se primenjuje nakon metoda crne kutije, koje su bazirane na specifikaciji i verifikuju funkciju sistema. Sami programeri često koriste strukturno testiranje na jediničnom i integracionom nivou, pošto se očekuje da su komponente koje su razvili temeljno testirane pre nego što se obavi dalja integracija tih komponenti u sistem i isporuči testerima na testiranje. Metode bele kutije su značajne i zbog toga što pomažu u određivanju mere temeljnosti testiranja, kroz procenu pokrivenosti programskog koda.

Kako se zna da li je softver testiran dovoljno? Kako se može odrediti da li je skup testova adekvatan? Pojam pokrivenosti testiranja (engl. *coverage*) predstavlja meru do koje je određeni softver ili softverska komponenta istestirana određenim skupom testova, u smislu procenta pokrivenih stavki (na primer naredbi). Ukoliko pokrivenost nije 100%, potrebno je proširiti skup testova novim testovima sa ciljem da se poveća procenat pokrivenosti. Pokrivenost se povećava tako što se sistematski razvijaju novi testovi koji će ciljati tačno one stavke (delove programa) koje su propuštene, kako bi se dostigla pokrivenost od 100%, odnosno da su svi delovi posmatranog programa izvršeni bar jedanput.

Kako se određuju pojmovi na osnovu kojih se meri pokrivenost testiranja? Generalno pravilo je da, ukoliko mogu da se izbroje neke stavke unutar programa koji se testira, i da se odredi da li su te stavke pokrivene (izvršene) nekim testom iz posmatranog skupa testova ili ne, onda se može izmeriti pokrivenost i kvantitativno prikazati u vidu procenta pokrivenosti tih stavki. Pokrivenost se dalje računa po formuli:

$$\text{Pokrivenost} = \frac{\text{Broj stavki koje su izvršene}}{\text{Ukupan broj stavki}} \times 100\%$$

Stavke mogu biti pojedinačne naredbe, odluke, uslovi, putanje i slično. Mogu se takođe posmatrati i stavke iz metoda crne kutije, pa je tako moguće imati procenat pokrivenih klasa ekvivalencije, procenat pokrivenih graničnih vrednosti, procenat posećenih stanja ili pokrivenih tranzicija i slično.

Treba takođe biti oprezan kada se koristi mera pokrivenosti testiranja, jer 100% pokrivenost ne znači da je softver 100% testiran. 100% pokrivenost samo znači da su svi delovi koda pokriveni, i predstavlja jednu dimenziju višedimenzionalnog koncepta. Dva različita testa mogu da aktiviraju identične stavke u programu i da postignu istu pokrivenost, ali na osnovu različitih ulaznih podataka jedan može pronaći grešku, dok drugi ne.

Upotreba pokrivenosti testiranja donosi veliki broj prednosti, od kojih su najznačajnije:

- Pomaže u kreiranju dodatnih testova kako bi se povećala pokrivenost.
- Pomaže u pronalaženju delova programa koji uopšte nisu pokriveni testovima.
- Određuje kvantitativnu meru pokrivenosti koda, koja indirektno predstavlja i meru kvaliteta softvera.

Kao mane ovog pristupa mogu se pomenuti sledeći nedostaci:

- Može se samo izmeriti pokrivenost koda koji je napisan – ne zna se ništa o delu softvera koji još uvek nije napisan.
- Strukturne tehnike nisu efikasne u slučaju da specificirana funkcionalnost nije implementirana, ili ukoliko je funkcija izostavljena iz specifikacije. Ceo strukturni pristup se i zasniva na činjenici da se posmatra samo kod, a ne specifikacija. Ovakvi tipovi grešaka se uočavaju upotrebom tehnika crne kutije.

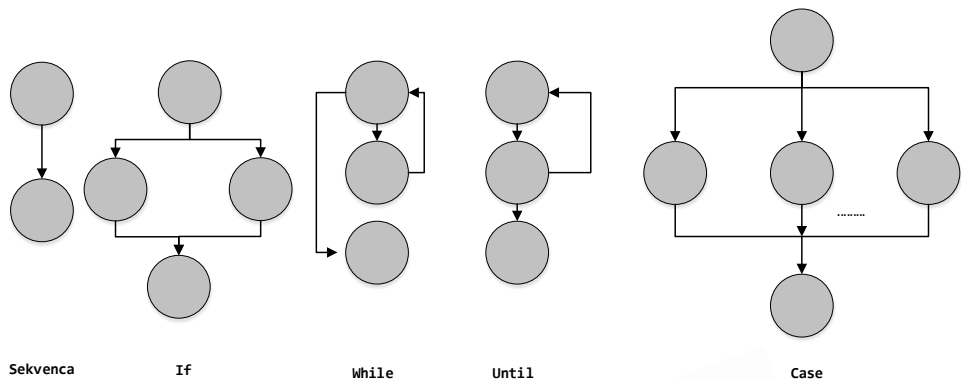
Najznačajnije metode bele kutije, koje su i obrađene u nastavku ovog poglavlja, se mogu podeliti na:

- Metode zasnovane na toku kontrole
 - Pokrivanje iskaza (engl. *Statement coverage*)
 - Pokrivanje odluka (engl. *Decision/branch coverage*)
 - Pokrivanje uslova (engl. *Condition coverage*)
 - Pokrivanje putanja (engl. *Path coverage*)
- Metode zasnovane na toku podataka

Pre nego što se počne sa konkretnim metodama bele kutije, neophodno je prvo uvesti pojam grafa toka kontrole, koji se koristi u navedenim metodama.

4.1. Graf toka kontrole

Graf toka kontrole (engl. *Control Flow Graph – CFG*) predstavlja reprezentaciju programa u obliku grafa, i implicitno prikazuje putanje izvršavanja programa. Čvorovi ovog grafa su instrukcije (ili bazični blokovi koda), a grane grafa predstavljaju sekvenciranje instrukcija, na primer ako postoji neki uslov, onda postoji grananje u grafu. Instrukcije u nizu se najčešće grupišu zajedno u bazični blok, koji predstavlja sekvencu instrukcija sa samo jednom ulaznom i jednom izlaznom tačkom. Bazični blok je deo koda se izvršava bez ikakvih skokova, može da počne mestom doskoka, a završava se skokom. Skokovi u toku kontrole se prikazuju usmerenim granama u grafu. Osnovne kontrolne strukture u programu i njihova reprezentacija u grafu toka kontrole prikazani su na slici 4.1.



Slika 4.1. Osnovne kontrolne strukture u programu

U svakom grafu toka postoji inicijalni čvor. Ovom čvoru odgovara bazični blok sa prvom instrukcijom. Između dva čvora grafa $n1$ i $n2$ postoji grana ako se bazični blok B2 koda kome odgovara $n2$ može izvršiti odmah nakon bazičnog bloka B1 kome odgovara čvor $n1$. U slučaju proceduralnih jezika, to se može desiti u dva slučaja:

- Postoji uslovni ili безусловni skok iz B1 na prvu instrukciju B2 ili,
- B2 direktno sledi iza B1 u originalnom redosledu programa, a B1 se ne završava безусловnim skokom.

Definicija: Graf toka kontrole je trojka $G = (V, E, s)$, gde je (V, E) konačni usmereni graf (V je skup čvorova grafa, E je skup grana grafa), i gde postoji putanja od inicijalnog čvora s (koji je jedan od čvorova iz V) do svih ostalih čvorova grafa. Ukoliko postoji bilo koji čvor do kog se ne može doći iz s , on se može ukloniti bez gubitka funkcionalnosti. Drugim rečima, taj deo koda se nikada neće izvršavati i samim tim je višak.

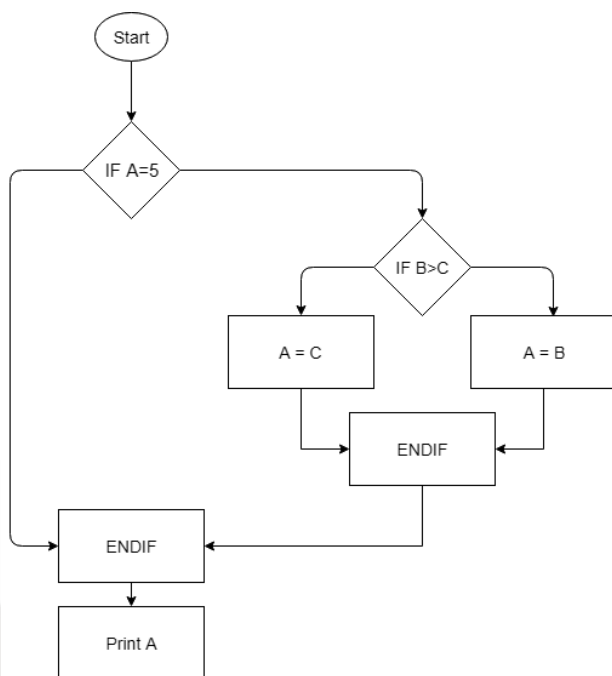
Primer: posmatra se jednostavan program (dat u pseudokodu):

```

IF A=5
    THEN IF B>C
        THEN A=B
        ELSE A=C
    ENDIF
ENDIF
Print A

```

Graf toka kontrole za ovaj program je dat na slici 4.2. Po ustaljenoj praksi se uslovne naredbe crtaju kao rombovi, dok se obične naredbe crtaju kao pravougaonici.



Slika 4.2. Primer grafa toka kontrole

Graf toka kontrole se koristi za lakše razumevanje koda prilikom pravljenja izmena ili prilikom testiranja, pošto se na njemu jasno vide sekvence u kojima se instrukcije izvršavaju. Koristi se i za izračunavanje ciklomatskog broja, koji označava kompleksnost koda i o kome će biti reči na kraju ovog poglavlja.

4.2. Pokrivanje iskaza

Pokrivanje iskaza je najosnovnija metoda bele kutije. Glavna ideja koja stoji iza ove tehnike je da nije moguće znati da li je neki iskaz ispravan ili u njemu postoji greška ukoliko se on ne izvrši. Prema tome, cilj je napraviti testove na takav način da se svaki iskaz (naredba) programa izvrši bar jedanput. Nakon testiranja se mogu identifikovati svi izvršeni iskazi, kao i oni koji nisu izvršeni zbog nekog problema, poput mrtvog koda, nekorisćenih grana i slično.

Glavne prednosti ove metode su:

- Verifikacija šta se očekuje da napisani kod radi, a šta ne.
- Može se videti kvalitet koda.
- Proverava da li postoji nedostupan kod (mrtav kod).

Među mane ovog pristupa se mogu svrstati:

- Ne mogu se testirati netačni uslovi.
- Ne može se detektovati da li je petlja došla do uslova završetka.
- Ne razume logičke operatore
- Ako se iskaz pravilno izvršava za jednu ulaznu vrednost, ne postoje garancije da će se ispravno izvršavati i za sve ostale moguće ulazne vrednosti.

Pokrivenost iskaza se predstavlja procentom izvršenih iskaza u odnosu na ukupan broj iskaza u softverskoj komponenti koja se testira. Drugim rečima, pokrivenost iskaza predstavlja meru do koje je određeni skup testova pokrio sve iskaze u kodu. Izračunava se na osnovu sledeće jednačine:

$$\text{Pokrivenost iskaza} = \frac{\text{Broj iskaza koji su izvršeni}}{\text{Ukupan broj iskaza}} \times 100\%$$

Pokrivenost iskaza je najlakše demonstrirati na konkretnom primeru. Posmatra se prosta Java metoda, koja štampa zbir dva broja, s tim što ukoliko je zbir negativan, dodaje reč negativan ispred rezultata, a ukoliko je zbir pozitivan, dodaje reč pozitivan ispred rezultata.

```
public void stampajSumu (int a, int b) {  
    int rezultat = a + b;  
    if (rezultat > 0)  
        System.out.println ("Pozitivan " + rezultat);  
    else  
        System.out.println ("Negativan " + rezultat);  
}
```

Posmatra se prvi scenario, gde su a i b pozitivni brojevi, i neka je a = 3 i b = 9. Iskazi koji su pokriveni prilikom izvršavanja ovog konkretnog testa su označeni boldovano u kodu.

```
public void stampajSumu (int a, int b) {  
    int rezultat = a + b;  
    if (rezultat > 0)  
        System.out.println ("Pozitivan " + rezultat);  
    else  
        System.out.println ("Negativan " + rezultat);  
}
```

Ukupan broj iskaza u kodu je 7, a testom je izvršeno 5, pa je prema opisanoj jednačini pokrivenost iskaza $5/7 = 0.71$, odnosno 71%. Očigledno je da sa jednim testom nisu pokriveni svi iskazi, pošto else grana nije izvršena nijedanput. Kako bi se povećala

pokrivenost iskaza, sistematično se dodaju testovi koji ciljaju naredbe koje nisu pokrivene do sada. U ovom primeru, potrebno je dodati test koji će aktivirati else granu u metodi.

Neka su sada $a = -3$ i $b = -9$. Iskazi koji su pokriveni ovim drugim testom su označeni boldovano u kodu.

```
public void stampaJSumu (int a, int b) {  
    int rezultat = a + b;  
    if (rezultat > 0)  
        System.out.println ("Pozitivan " + rezultat);  
    else  
        System.out.println ("Negativan " + rezultat);  
}
```

Drugi test izvršava 6 iskaza, pa je njegova pokrivenost $6/7 = 0.85$, odnosno 85%. Međutim, zajedno, ova dva testa pokrivaju sve iskaze, pa je pokrivenost iskaza za metodu iz primera 100%.

4.3. Pokrivanje odluka

Pokrivanje odluka (engl. *branch coverage* ili *decision coverage*) je metoda bele kutije u kojoj se testovi biraju tako da se svaka od različitih grana uslovnih iskaza izvrši bar jedanput. Grana (engl. *branch*) predstavlja ishod odluke, pa se pokrivanje odluka svodi na merenje koliko je izlaznih grana uslovnih naredbi pokriveno testovima. Pod uslovnim iskazom (odnosno odlukom) se podrazumevaju IF naredbe, kontrolne naredbe u petljama (na primer uslov u WHILE petlji) ili SWITCH-CASE naredbe, gde postoji dva ili više ishoda na osnovu vrednosti uslovnog izraza. U slučaju proste IF naredbe, postoje dve moguće izlazne grane, jedna ukoliko je logički izraz true, a druga ukoliko je logički izraz false.

Pokrivanje odluka služi za validaciju da su sve grane u kodu dostupne, i da se osigura da nijedna grana neće dovesti do grešaka u izvršavanju programa. Pokrivenost se izračunava na osnovu sledeće jednačine:

Razliku pokrivanja odluka u odnosu na pokrivanje iskaza najlakše je pokazati na primeru. Posmatra se prost pseudokod koji, ukoliko je vrednost promenljive C manja od 0 štampa na izlazu da je C negativan.

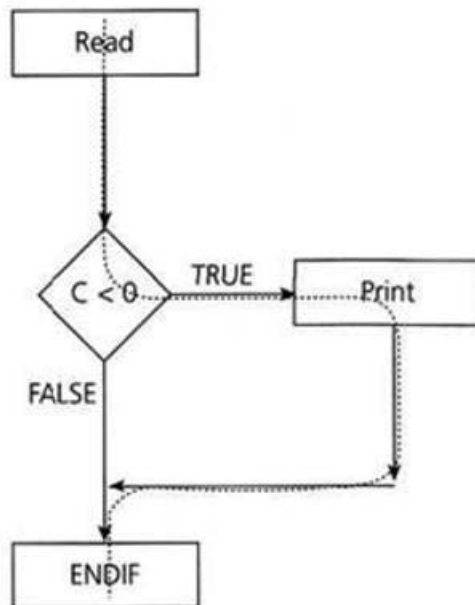
```
1 READ A
2 READ B
3 C = A - 2 * B
4 IF C < 0 THEN
5 PRINT "C je negativan"
6 ENDIF
```

Pretpostavlja se takođe da već postoji test koji obezbeđuje 100% pokrivenost iskaza, i neka je definisan sa:

Test 1: A = 20, B = 15

Lako se može uočiti da je vrednost $C = -10$, pa je i uslov $C < 0$ ispunjen. Na izlazu će biti odštampano "C je negativan", a svi iskazi u programu su pokriveni. Međutim, ako se detaljnije pogleda dati kod, izvršena je samo true grana IF uslova, a false grana nije izvršena nijedanput. Kako bi se uvidela struktura koda, naročito u slučaju da postoji puno odluka u kodu (IF uslovi), često je lakše da se kod predstavi u obliku grafa toka kontrole kao na slici 4.3. Linija sa strelicama na slici pokazuje tok izvršavanja programa za test 1, i sada se još lakše uočava da false grana IF uslova nije uopšte testirana. Ukoliko se potrebno postići 100% pokrivanje odluka, mora se dodati još jedan test koji će ciljano pokriti false granu IF uslova:

Test 2: A = 10, B = 2



Slika 4.3. Graf toka kontrole u slučaju kada se ne pokriva jedna grana odluke

Iz navedenog primera se mogu izvući bitni zaključci. Samo jedan test je bio dovoljan da se postigne 100% pokrivenost iskaza. Međutim, pokrivenost odluka zahteva da se za svaki IF uslov pokriju i true i false grane. Neophodno je dodati još jedan test kako bi se postigla 100% pokrivenost odluka, odnosno da se pokrije i false grana koja prvim testom nije aktivirana. Veoma je važno uočiti da 100% pokrivenost odluka garantuje i 100% pokrivenost iskaza, dok obrnuto ne važi. Stoga se pokrivanje odluka smatra jačim metodom od pokrivanja iskaza.

4.4. Pokrivanje uslova

Pokrivanje uslova (engl. *condition coverage*) je metoda bele kutije koja je bliska pokrivanju odluka, sa većom osetljivošću na tok kontrole. Međutim, treba biti oprezan, pošto iako je ova metoda kompleksnija od pokrivanja odluka, 100% pokrivenost uslova ne garantuje 100% pokrivenost iskaza. Kod pokrivanja uslova se svaki elementarni pojedinačni uslov u uslovnim iskazima testira i sa true i sa false vrednostima. Pri tome, elementarni uslovi se posmatraju nezavisno jedan od drugog. Neka se posmatra sledeći uslov:

```
if (x > 5 && y == 3) { ... }
```

Prema pokrivanju uslova, oba elementarna uslova u IF uslovu, $x > 5$ i $y == 3$, treba evaluirati sa true i false vrednostima nezavisno jedan od drugog. Dva testa, $x = 6, y = 2$ i $x = 4, y = 3$, generišu sve moguće vrednosti svih elementarnih uslova (true i false, false i true), ali oba idu na istu izlaznu granu IF uslova (false grana). Iako je postignuta 100% pokrivenost uslova, nije postignuta ni 100% pokrivenost odluka, ni 100% pokrivenost iskaza. Zbog toga postoji nekoliko varijacija pokrivanja uslova koje daju jaču pokrivenost, i koje su opisane u nastavku.

4.4.1. Pokrivanje uslova i odluka

Pokrivanje uslova i odluka podrazumeva simultano pokrivanje oba kriterijuma. Svaki elementarni uslov mora uzeti i true i false vrednosti, uz dodatak da se svaka odluka kao celina mora evaluirati i sa true i sa false, kako bi se obezbedilo da se sve moguće izlazne grane izvrše. Ukoliko se posmatra isti uslov:

```
if (x > 5 && y == 3) { ... }
```

Prema pokrivanju uslova i odluka, oba elementarna uslova u IF uslovu, $x > 5$ i $y == 3$, treba evaluirati sa true i false vrednostima nezavisno jedan od drugog, uz dodatak da se i kompletan uslov ($x > 5 \text{ \&\& } y == 3$) kao celina mora evaluirati i sa true i sa false, kako bi se obezbedilo da se obe izlazne grane IF uslova izvrše. Dva testa, $x = 6, y = 3$ i $x = 4, y = 2$, generišu sve moguće vrednosti svih elementarnih uslova (true i true, false i false), ali oba sada ne idu na istu granu, već prvi test pokriva true izlaznu granu kompletne IF naredbe, dok drugi test pokriva false izlaznu granu.

4.4.2. Pokrivanje višestrukih uslova

U ovoj varijanti pokrivanja uslova, testovi se dizajniraju na takav način da se pokriju sve moguće kombinacije elementarnih uslova u odlukama. Problem sa ovim pristupom je u činjenici da broj testova raste eksponencijalno sa brojem uslova – ukoliko postoji 3 uslova, biće potrebno 8 testova, odnosno u opštem slučaju za n uslova, broj potrebnih testova iznosi 2^n . Zbog kombinatorne eksplozije, često je ovaj pristup preskup za implementaciju, i praktičan je samo u slučaju da je n malo. Kao olakšavajuća okolnost, u praksi se pokazuje da su često neke od kombinacija nemoguće, što donekle smanjuje broj neophodnih testova. Na primer, ukoliko se posmatra sledeći uslov:

```
if ( a == 1 || a == 2 ) { ... }
```

Ako se posmatra broj mogućih kombinacija, pošto postoji 2 uslova, broj potrebnih testova je 2^2 , odnosno 4 testa. Samo tri kombinacije su zaista moguće, pošto kombinacija kada su oba uslova true nije izvodljiva, jer a ne može istovremeno imati i vrednost 1 i vrednost 2. Skup svih kombinacija je prikazan na slici 4.4.

Elementarni uslovi	Testovi			
	$a = 1$	$a = 2$	$a = 0$	nemoguće
$a == 1$	TRUE	FALSE	FALSE	TRUE
$a == 2$	FALSE	TRUE	FALSE	TRUE

Slika 4.4. Pokrivanje višestrukih uslova, primer nemoguće kombinacije uslova

4.4.3. Minimalno pokrivanje višestrukih uslova

U ovoj tehnici testovi se dizajniraju na takav način da svaki elementarni i neelementarni uslov složenog uslova uzmu i true i false vrednost. Ako se posmatra isti primer uslova:

```
if ( a == 1 || a == 2 ) { ... }
```

Elementarni uslovi koje je potrebno pokriti su $a == 1$ i $a == 2$, a neelementarni uslov koji je potrebno pokriti je $(a == 1 || a == 2)$. Kombinacije koje je potrebno testirati su prikazane na slici 4.5.

Uslovi	Testovi		
	a = 1	a = 2	a = 0
a == 1	TRUE	FALSE	FALSE
a == 2	FALSE	TRUE	FALSE
(a == 1 a == 2)	TRUE	TRUE	FALSE

Slika 4.5. Minimalno pokrivanje višestrukih uslova

4.5. Pokrivanje putanja

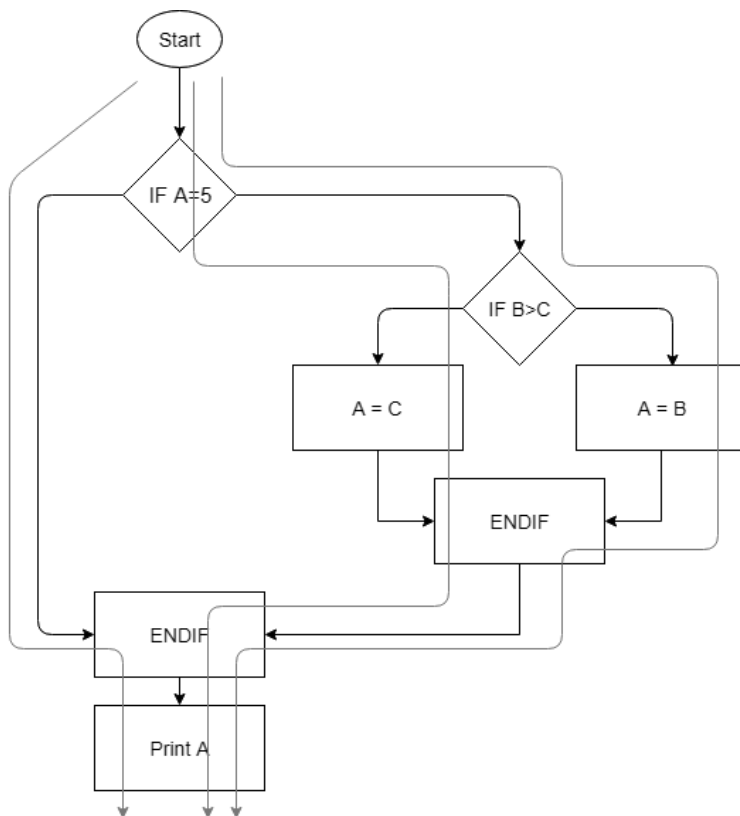
Pokrivanje putanja je metoda bele kutije koja se u potpunosti zasniva na grafu toka kontrole. Cilj ove metode je da se pokriju i bar jednom izvrše sve putanje u datom programu, čime se pokriva kompletna kontrolna struktura programa. Posmatra se dati pseudokod:

```

1 IF A=5
2 THEN IF B>C
3 THEN A=B
4 ELSE A=C
5 ENDIF
6 ENDIF
7 Print A

```

Zbog lakšeg označavanja putanja svakoj naredbi je pridružen redni broj. Kada se nacrtaj graf toka kontrole (slika 4.6), jasno se može uočiti da postoje tri moguće putanje kroz program.



Slika 4.6. Graf toka kontrole i sve putanje kroz program

Putanje koje je potrebno testirati su redom (redosled izvršavanja naredbi označen njihovim odgovarajućim pridruženim brojevima):

Putanja 1: 1, 2, 3, 5, 6, 7

Putanja 2: 1, 2, 4, 5, 6, 7

Putanja 3: 1, 6, 7

Svaki pojedinačni test je jedna putanja od početnog do završnog čvora u grafu toka kontrole. Kriterijum pokrivenosti zavisi od mere u kojoj je skup testova pokrio program, i računa se prema jednačini:

$$\text{Pokrivenost putanja} = \frac{\text{Broj putanja koje su izvršene}}{\text{Ukupan broj putanja}} \times 100\%$$

Kako bi se obezbedila 100% pokrivenost putanja, neophodno je testovima pokriti bazični skup putanja. Bazični skup putanja je skup svih linearno nezavisnih putanja (putanja je nezavisna ukoliko se razlikuje od svih drugih putanja u bar jednom svom delu, odnosno ne može se predstaviti kao linearna kombinacija ostalih putanja).

Linearne kombinacije putanja se najlakše mogu predstaviti preko matrice incidencije, koja u slučaju datog primera izgleda kao na slici 4.7.

Putanje	Grane u grafu (grana 1-2 označava granu koja se nalazi između čvorova 1 i 2)							
	1-2	1-6	2-3	2-4	3-5	4-5	5-6	6-7
P1 - 1, 2, 3, 5, 6, 7	1	0	1	0	1	0	1	1
P2 - 1, 2, 4, 5, 6, 7	1	0	0	1	0	1	1	1
P3 - 1, 6, 7	0	1	0	0	0	0	0	0

Slika 4.7. Tabelarni prikaz matrice incidencije za graf toka kontrole sa slike 4.6

Bez dubljeg zalaženja u polje matematike, jednostavno se može uočiti da su date tri putanje linearno nezavisne, jer se nijedna od putanja ne može predstaviti kao linearna kombinacija ostalih putanja iz ovog skupa. Konkretno, putanja 1 je nezavisna od ostalih putanja zbog grana 2-3 i 3-5. Putanja 2 je nezavisna zbog grana 2-4 i 4-5, a putanja 3 zbog grane 1-6.

Bazični skup putanja označava takav skup putanja u kome su sve putanje međusobno linearno nezavisne, a bilo koja druga putanja u grafu toka kontrole se može predstaviti kao linearna kombinacija putanja iz bazičnog skupa. Za jedan graf toka kontrole u opštem slučaju može postojati više različitih bazičnih skupova, ali važi da svi oni imaju isti broj elemenata.

Broj linearno nezavisnih putanja kroz program ima još jednu bitnu primenu. Direktno je povezan sa pojmom ciklomatske kompleksnosti. Ciklomatska kompleksnost je mera koja opisuje kompleksnost programa, koju je uveo Thomas McCabe (za više detalja čitalac se upućuje na dodatnu literaturu - McCabe T.J., *A Complexity Measure*). Ciklomatsku kompleksnost možemo izračunati na osnovu grafa toka kontrole programa. U slučaju da u programu nema IF naredbi ili bilo kojih petlji, kompleksnost koda je 1. Ako program sadrži tačno jednu IF naredbu, u njemu postoje dve putanje – jedna kroz true izlaznu granu, a druga kroz false izlaznu granu. U opštem slučaju, ukoliko se graf toka kontrole posmatra kao trojka $G = (V, E, s)$, gde je V skup čvorova, a E skup grana, ciklomatska kompleksnost se računa kao:

$$CC = E - V + 2$$

CC određuje broj testova koji su neophodni za postizanje 100% pokrivenosti putanja, jer predstavlja minimalni broj putanja kroz dati graf toka kontrole. Ukoliko se posmatra graf toka kontrole sa slike 42, prostom analizom grafa zaključuje se da postoji 9 grana i 8 čvorova, odnosno imamo da je $E = 9$, a $V = 8$. Ubacivanjem ovih vrednosti u jednačinu dobija se broj linearno nezavisnih putanja kroz dati kod.

$$CC = E - V + 2 = 9 - 8 + 2 = 3$$

Dobijamo da postoje 3 linearno nezavisne putanje kroz dati graf, što se slaže sa prvobitnom analizom. Ciklomatsku kompleksnost svakog modula (metode) bi trebalo ograničiti na neku nižu vrednost (u praksi u nekim firmama čak ograničeno na 10), kako bi kod bio lak za razumevanje i održavanje.

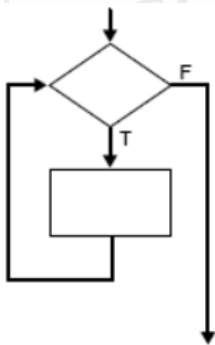
Pokrivanje putanja je najjača metoda u poređenju sa pokrivanjem odluka i pokrivanjem iskaza. 100% pokrivanje putanja garantuje i 100% pokrivanje odluka i 100% pokrivanje iskaza. 100% pokrivanje odluka garantuje 100% pokrivanje iskaza. Obrnuto ne važi.

4.6. Testiranje petlji

Testiranje petlji je metoda bele kutije koja takođe spada u tehnike koje su zasnovane na toku kontrole. Ova metoda se kompletno fokusira na ispravnost konstrukcija petlji. Zašto se petlje uopšte posmatraju odvojeno? Samo uz pomoć pažljivo pripremljenih testova se mogu otkriti:

- Problemi u ponavljanju iteracija
- Problemi sa performansama
- Neinicijalizovane promenljive unutar tela petlje
- Loša inicijalizacija petlje

Najosnovniji tip petlje jeste prosta petlja, čiji je graf toka kontrole dat na slici 4.8.

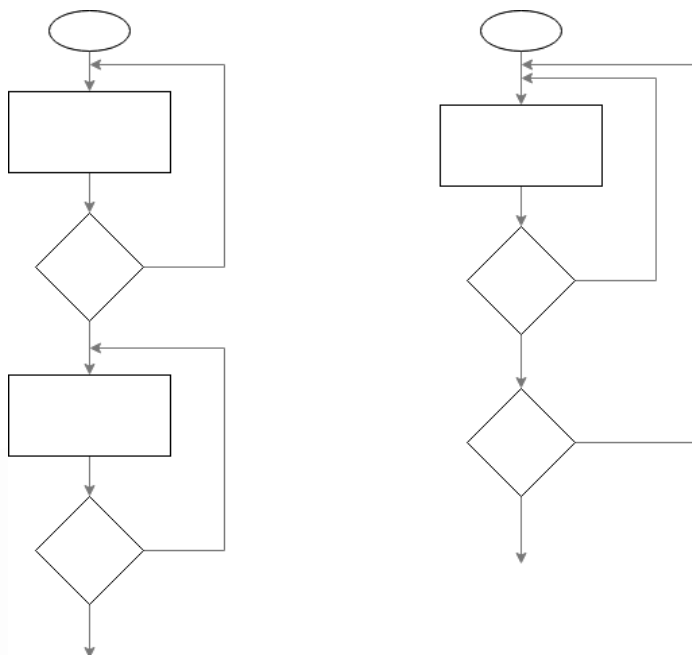


Slika 4.8. Graf toka kontrole proste petlje

Za temeljno testiranje proste petlje potrebno je pokriti sledeće slučajeve (pri čemu je M maksimalni broj mogućih iteracija petlje):

- Potpuno preskakanje tela petlje
- Jedna iteracija petlje
- Dve iteracije petlje
- Napraviti N iteracija petlje, pri čemu važi da je $N < M$
- Testirati i M-1, M i M+1 iteracija

U slučaju višestrukih petlji, postoje dva slučaja od interesa – nadovezane i ugnježdene petlje, čiji su odgovarajući grafovi toka kontrole prikazani na slici 4.9.



Slika 4.9. Nadovezana i ugnježdene petlje

Kod testiranja ugnjeđenih petlji, nije pogodno primeniti metod testiranja prostih petlji, jer bi to veoma brzo dovelo do ogromnog broja potrebnih testova. Procedura testiranja ugnjeđenih petlji podrazumeva sledeće korake:

- Počinje se od najugnjeđenije petlje
- Za najugnjeđeniju petlju se sprovode testovi za prostu petlju, dok se sve ostale petlje drže na minimumu broja ponavljanja
- Zatim se prelazi na okružujuću petlju, i sprovode testove za prostu petlju, dok se sve ostale petlje drže na minimumu broja ponavljanja
- Prelazi se sa petlje na petlju i ponavlja ista procedura sve dok se ne završi i testiranje spoljašnje petlje

Kod ugnjeđenih petlji potrebno je naglasiti da se kompleksnost koda drastično povećava sa dodavanjem svakog novog nivoa ugnježdavanja. U praksi se smatra da je jedan nivo ugnježdavanja (petlja u petlji) prihvatljiv. U retkim slučajevima se dozvoljava dva nivoa ugnježdavanja (petlja unutar petlje unutar petlje). Sve preko toga se najčešće ne toleriše, već je potrebno ponovo analizirati problem koji se rešava i pristupiti mu na drugi način.

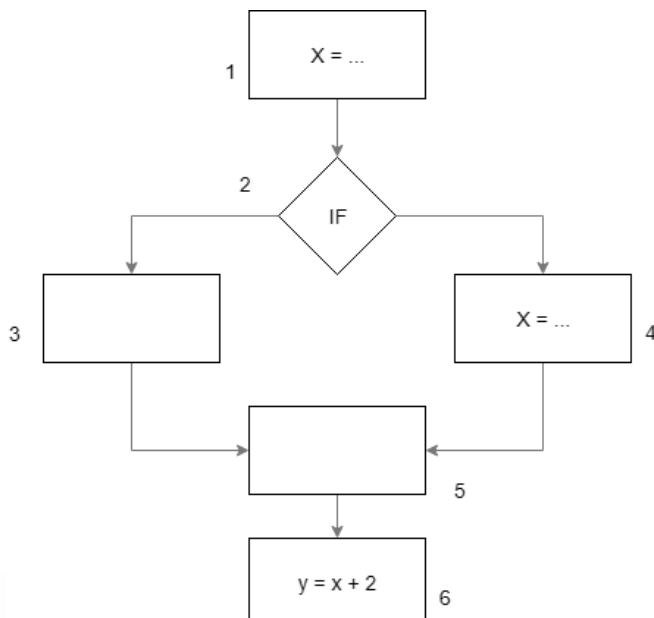
Nadovezane petlje, pošto su praktično u pitanju dve ili više petlji koje slede jedna za drugom, mogu se tretirati kao proste ukoliko su nezavisne jedna od druge.

4.7. Testiranje metodama toka podataka

Osim metoda koje su bazirane na toku kontrole, postoji još jedna grupa metoda koje spadaju u metode bele kutije. To su metode toka podataka, koje se baziraju na međusobnoj interakciji naredbi kroz zajedničke podatke koje te naredbe koriste. Motivacija za razvijanje ovih tehnika leži u činjenici da se prostim pokrivanjem iskaza i odluka u kodu mogu prevideti neke greške. Taj problem se može rešiti primenom metode pokrivanja putanja, međutim, često broj putanja kroz kod može dovesti do prevelikog broja testova. Pri tome, često nisu od interesa sve putanje, već samo neke od njih koje mogu otkriti dodatne greške. Kako uočiti i izdvojiti važne putanje?

Prostom intuicijom može se zaključiti da problemi mogu nastati u toku podataka, gde se vrednost nekog podatka izračunava u jednoj naredbi, a koristi u drugoj. Ukoliko je izračunata vrednost u jednoj naredbi pogrešna, greška se može otkriti tek kada neka druga naredba pokuša da iskoristi tu vrednost. Metoda toka podataka se zasniva na analizi grafa toka kontrole, kako bi se uočili potencijalni problemi koji mogu da utiču na podatke, i na osnovu njih odabrale putanje kroz kod koje je neophodno testirati. Prilikom odabira putanja se posmatraju lokacije dodele vrednosti promenljivama i lokacije njihove upotrebe. Ova strategija testiranja popunjava prazninu između potpunog pokrivanja putanja (koje je najčešće nepraktično i previše kompleksno) i pokrivanja iskaza i odluka (ne otkrivaju uvek sve greške). Posmatrajmo graf toka kontrole sa slike 4.10.

Radi lakšeg razumevanja, sve naredbe su označene sa brojevima. Naredbe u kojima se dodeljuje vrednost promenljivoj x su označene brojevima 1 i 4. Vrednost promenljive x se koristi u naredbi 6. Ukoliko je vrednost x loše izračunata, to se može otkriti tek u naredbi 6.



Slika 4.10. Graf toka kontrole

Mogu se uočiti dve kategorije koje opisuju stanje nekog podatka:

1. D – podatak je definisan, kreiran odnosno inicijalizovan
2. U – podatak je upotrebljen, u proračunu ili kao predikat uslova

Podatak se smatra definisanim ukoliko se pojavio u deklaraciji, dodeljena mu je nova vrednost ili je dinamički alociran. Upotreba podatka označava da je upotrebljen kao:

- deo proračuna (*c-use*), sa desne strane izraza dodele;
- deo predikata (*p-use*), ako se pojavljuje u predikatskom izrazu neke naredbe kontrole toka (*if*, *while* itd.).

Kako bi pojasnili definicije i upotrebe podataka, posmatra se sledeći pseudokod (radi lakšeg referenciranja pojedinačnih naredbi one su označene rednim brojevima):

```

1.  read (x,y);
2.  z = x + 2;
3.  if (z < y)
4.      w = x + 1;
   else
5.      y = y + 1;
6.  print (x,y,w,z);
  
```

Na slici 4.11 tabelarno su prikazane sve definicije i sve upotrebe (*c-use* i *p-use*) za sve podatke iz primera, zajedno sa odgovarajućim rednim brojem naredbe.

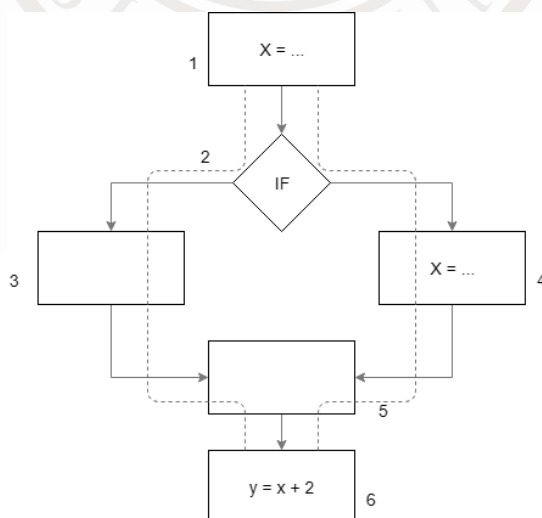
Naredba	Definicija	c - upotreba	p - upotreba
1	x , y		
2	z	x	
3			z , y
4	w	x	
5	y	y	
6		x , y , w , z	

Slika 4.11. Definicije i upotrebe podataka za dati primer

Nakon uvođenja osnovnih pojmova, sada se mogu definisati konkretne stavke koje se koriste u testiranju metodom toka podataka:

- DU par – par definicije i upotrebe neke promenljive, takav da između njih postoji bar jedna DU putanja.
- DU putanja – putanja u grafu toka kontrole koja počinje na mestu definicije i završava na mestu upotrebe jedne promenljive, bez dodatnih definicija (odnosno promena vrednosti) te promenljive.

Neka se posmatra graf toka kontrole prikazan na slici 4.12. Postoje dve putanje kroz dati graf toka kontrole, putanja 1-2-3-5-6 i putanja 1-2-4-5-6, označene prema rednim brojevima naredbi koje pripadaju tim putanjama, i prikazane isprekidanim linijama na grafu toka kontrole.



Slika 4.12. Putanje u grafu toka kontrole

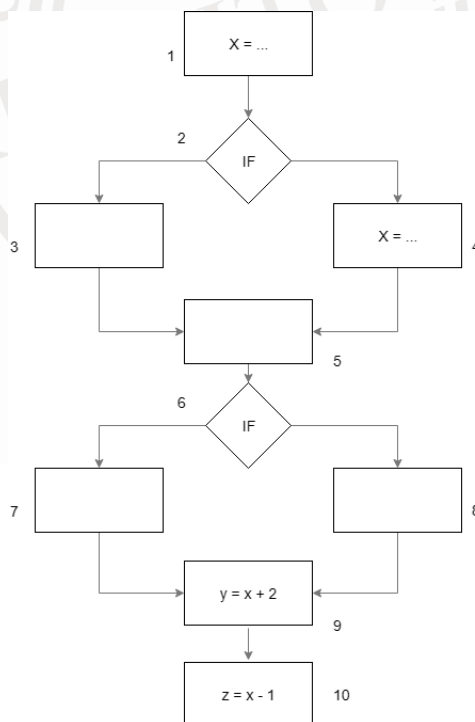
Definicije promenljive x se nalaze u naredbama 1 i 4, a upotreba u naredbi sa rednim brojem 6. Parovi naredbi (1,6) i (4,6) su prema definiciji DU parovi. Ukoliko se posmatraju putanje, 1-2-3-5-6 je putanja bez dodatnih definicija između naredbi 1 i 6, pošto ne postoji nijedna dodatna promena vrednosti promenljive x . Putanja 1-2-4-5-6 nije putanja bez dodatnih definicija, pošto se vrednost promenljive x menja u naredbi 4. Sekcija 4-5-6 jeste putanja bez dodatnih definicija, pošto je x definisana u naredbi 4, a upotrebljena u naredbi 6. Prema tome, u ovom primeru važe sledeća zapažanja:

- DU parovi su (1,6) i (4,6).
- DU putanje su 1-2-3-5-6 i 4-5-6.

Moguće je formulisati nekoliko kriterijuma pokrivenosti za metode toka podataka, od kojih su najbitniji:

- Svi DU parovi (drugi naziv sve upotrebe) – svaki DU par je pokriven bar jednim testom.
- Sve DU putanje – svaka prosta DU putanja (bez petlje) je pokrivena bar jednim testom.
- Sve definicije – za svaku definiciju, postoji bar jedan test koji pokriva DU par koji sadrži tu definiciju.

Neka se posmatra sada graf toka kontrole sa slike 4.13. Za dati graf je potrebno odrediti testove koji pokrivaju sve DU parove, sve DU putanje i sve definicije.



Slika 4.13. Primer određivanja različitih kriterijuma pokrivenosti

Pokrivanje svih DU parova podrazumeva u ovom slučaju pokrivanje parova:

- Definicija 1, upotreba 9
- Definicija 1, upotreba 10
- Definicija 4, upotreba 9
- Definicija 4, upotreba 10

Pokrivanje svih DU putanja podrazumeva pokrivanje:

- Definicija 1, upotreba 9 kroz naredbu 7: 1-2-3-5-6-7-9
- Definicija 1, upotreba 9 kroz naredbu 8: 1-2-3-5-6-8-9
- Definicija 4, upotreba 9 kroz naredbu 7: 4-5-6-7-9
- Definicija 4, upotreba 9 kroz naredbu 8: 4-5-6-8-9
- Definicija 1, upotreba 10 kroz naredbu 7: 1-2-3-5-6-7-9-10
- Definicija 1, upotreba 10 kroz naredbu 8: 1-2-3-5-6-8-9-10
- Definicija 4, upotreba 10 kroz naredbu 7: 4-5-6-7-9-10
- Definicija 4, upotreba 10 kroz naredbu 8: 4-5-6-8-9-10

Pokrivanje svih definicija podrazumeva pokrivanje:

- Definicija 1, upotreba 9 ILI definicija 1, upotreba 10
- Definicija 4, upotreba 9 ILI definicija 4, upotreba 10

Kao što je već ranije naglašeno, postoje dva tipa upotrebe promenljivih, predikatska (p-upotreba) i računska (c-upotreba). Može se uočiti da je svaki od ovih tipova zapravo podskup DU parova (odnosno svih upotreba), pošto važi da je unija svih p-upotreba i svih c-upotreba u nekom kodu jednaka svim DU parovima (svim upotrebama).

Postoje još neke strategije osim tri najopštije koje su već opisane (pokrivanje definicija, DU parova, DU putanja), koje su zasnovane na tipu upotreba promenljivih (p-use i c-use):

- Pokrivanje svih p-upotreba
- Pokrivanje svih c-upotreba
- Sve c-upotrebe, neke p-upotrebe
- Sve p-upotrebe, neke c-upotrebe

Primer - posmatra se metoda koja rešava kvadratnu jednačinu, i čiji je kod u programskom jeziku Java dat u nastavku:

```
void Reši_Kvadratnu (double A, double B, double C) { //0
    double Diskriminanta = B * B - 4 * A * C; //1
    bool Kompleksna; //2
    double R1, R2; //3
    if (Diskriminanta < 0) { //4
```

```

        Kompleksna = true; //5
    } else { //6
        Kompleksna = false; //7
    } //8
    If (! Kompleksna) { //9
        R1 = (-B + sqrt(Diskriminanta)) / (2.0 * A); //10
        R2 = (-B - sqrt(Diskriminanta)) / (2.0 * A); //11
    } //12
} //13

```

Radi lakšeg referisanja naredbi, svakoj je dodeljen redni broj. Ukoliko promenljiva Kompleksna ima vrednost true, koreni kvadratne jednačine nisu realni. Ukoliko su koreni kvadratne jednačine realni, izračunavaju se i smeštaju u promenljive R1 i R2. Definicije i upotrebe svih promenljivih (podataka) su prikazane na slici 4.14.

Naredba	Definicija	c - upotreba	p - upotreba
0	A,B,C		
1	Diskriminanta	A,B,C	
2			
3			
4			Diskriminanta
5	Kompleksna		
6			
7	Kompleksna		
8			
9			Kompleksna
10	R1	A, B, Diskriminanta	
11	R2	A, B, Diskriminanta	
12			
13			

Slika 4.14. Definicije i upotrebe promenljivih iz metode Reši_Kvadratnu

DU parovi za dati kod su prikazani na slici 4.15. Na primer, promenljiva A je definisana u liniji 0 (kada se prihvati stvarni parametar koji je prosleđen u pozivu metode), a njene upotrebe su u linijama 1, 10 i 11. Za promenljivu A stoga imamo DU parove (0, 1), (0, 10) i (0, 11). Za promenljivu B važi identična priča, i DU parovi (0, 1), (0, 10) i (0, 11). Promenljiva C je definisana takođe u liniji 0, a jedina njena upotreba je u liniji 1, pa za nju imamo DU par (0, 1). Promenljiva Diskriminanta je definisana u

liniji 1, a njene upotrebe su u linijama 4, 10 i 11, pa imamo DU parove (1, 4), (1, 10) i (1, 11). Promenljiva Kompleksna je definisana ili u naredbi 5 ili u naredbi 7, a njena upotreba je u liniji 9, pa imamo DU parove (5, 9) i (7, 9).

DU par (početna naredba - krajnja naredba)	Promenljive	Tip upotrebe
0 -> 1	A, B, C	c
0 -> 10	A, B	c
0 -> 11	A, B	c
1 -> 4	Diskriminanta	p
1 -> 10	Diskriminanta	c
1 -> 11	Diskriminanta	c
5 -> 9	Kompleksna	p
7 -> 9	Kompleksna	p

Slika 4.15. DU parovi za metodu Reši_Kvadratnu

100% pokrivanje definicija je prikazano na slici 4.16. Da se podsetimo, pokrivanje definicija se postiže na takav način da za svaku definiciju, postoji bar jedan test koji pokriva DU par koji sadrži tu definiciju.

Sve definicije				Ulazni podaci			Očekivani rezultat		
Test	Promenljiva	DU par	Putanja	A	B	C	Kompleksna	R1	R2
1	A, B, C	0 -> 1	0-1	1	1	1	true	Nedefinisano	Nedefinisano
2	Diskriminanta	1 -> 4	1-4	1	1	1	true	Nedefinisano	Nedefinisano
3	Kompleksna	5 -> 9	5-9	1	1	1	true	Nedefinisano	Nedefinisano
4	Kompleksna	7 -> 9	7-9	1	2	1	false	-1	-1

Slika 4.16. Pokrivanje svih definicija za metodu Reši_Kvadratnu

Pokrivanje svih DU parova je prikazano na slici 4.17. 100% pokrivenost se postiže tako što je svaki DU par pokriven bar jednom.

Svi DU parovi				Ulazni podaci			Očekivani rezultat		
Test	Promenljiva	DU par	Putanja	A	B	C	Kompleksna	R1	R2
1	A, B, C	0 -> 1	0-1	1	1	1	true	Nedefinisano	Nedefinisano
2	A, B	0 -> 10	0-1-4-7-9-10	1	2	1	false	-1	-1
3	A, B	0 -> 11	0-1-4-7-9-10-11	1	2	1	false	-1	-1
4	Diskriminanta	1 -> 4	1-4	1	1	1	true	Nedefinisano	Nedefinisano
5	Diskriminanta	1 -> 10	1-4-7-9-10	1	2	1	false	-1	-1
6	Diskriminanta	1 -> 11	1-4-7-9-10-11	1	2	1	false	-1	-1
7	Kompleksna	5 -> 9	5-9	1	1	1	true	Nedefinisano	Nedefinisano
8	Kompleksna	7 -> 9	7-9	1	2	1	false	-1	-1

Slika 4.17. Pokrivanje svih DU parova za metodu Reši_Kvadratnu

Pokrivanje svih DU putanja se zasniva na pokrivanju svake proste putanje od svake definicije do svih upotreba te definicije. Pod prostom putanjom u grafu toka kontrole podrazumeva se putanja čiji se svaki deo izvršava minimalan broj puta (na primer jedna iteracija kroz petlju). U metodi Reši_Kvadratnu, osim putanja koje pokrivaju navedeni testovi za sve DU parove, postoje još dve putanje: 0-1-4-5-9-10 i 1-4-5-9-10. Međutim, zbog programske logike u datom primeru, nije ih moguće pokriti testovima.

Kao što smo već ranije naveli, osim tri najopštije metode (pokrivanje definicija, DU parova i DU putanja) postoje još neke strategije koje su zasnovane na tipu potreba promenljivih (p-upotrebe i c-upotrebe). Pokrivanje p-upotreba i pokrivanje c-upotreba su praktično podskupovi pokrivanja DU parova, jer pokrivanje DU parova zapravo znači pokrivanje svih p i c upotreba. Pokrivanje svih p-upotreba i pokrivanje svih c-upotreba za dati primer Reši_Kvadratnu su dati na slikama 4.18 i 4.19, respektivno.

Test	Sve p-upotrebe			Ulazni podaci			Očekivani rezultat		
	Promenljiva	DU par	Putanja	A	B	C	Kompleksna	R1	R2
1	Diskriminanta	1 -> 4	1-4	1	1	1	true	Nedefinisano	Nedefinisano
2	Kompleksna	5 -> 9	5-9	1	1	1	true	Nedefinisano	Nedefinisano
3	Kompleksna	7 -> 9	7-9	1	2	1	false	-1	-1

Slika 4.18. Pokrivanje svih p-upotreba za metodu Reši_Kvadratnu

Test	Sve c-upotrebe			Ulazni podaci			Očekivani rezultat		
	Promenljiva	DU par	Putanja	A	B	C	Kompleksna	R1	R2
1	A, B, C	0 -> 1	0-1	1	1	1	true	Nedefinisano	Nedefinisano
2	A, B	0 -> 10	0-1-4-7-9-10	1	2	1	false	-1	-1
3	A, B	0 -> 11	0-1-4-7-9-10-11	1	2	1	false	-1	-1
4	Diskriminanta	1 -> 10	1-4-7-9-10	1	2	1	false	-1	-1
5	Diskriminanta	1 -> 11	1-4-7-9-10-11	1	2	1	false	-1	-1

Slika 4.19. Pokrivanje svih c-upotreba za metodu Reši_Kvadratnu

Naprednije teme iz testiranja metodom toka podataka, poput dekompozicije programa na bazične blokove, zatim graf toka kontrole na nivou bazičnih blokova, dalja globalna analiza toka podataka i iterativni algoritmi izlaze van obima ove knjige, pa se čitalac upućuje na dodatnu literaturu (Nielson F., Nielson H.R., Hankin C., *Principles of Program Analysis*).

Testiranje svih upotreba (odnosno DU parova) u praksi ima najbolje rezultate i najisplativije je (pošto pokrivanje putanja često nije praktično). U praksi je pokazano da ima prosečno dva puta više potrebnih testova od pokrivanja odluka, ali su rezultati mnogo bolji. Potrebno je uočiti da su sami podaci podjednako važni kao i kod, i da se na njih mora obratiti posebna pažnja. Testiranje metodama toka podataka ispunjava prazninu između pokrivanja svih odluka (često ne otkriva sve greške) i pokrivanja svih putanja (često je nepraktično).

4.8. Zadaci za vežbu

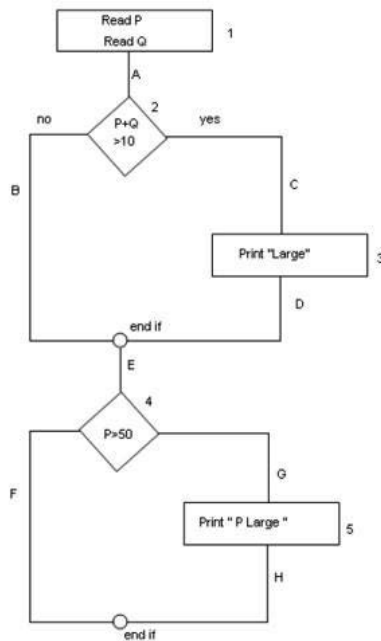
Primer 1: Za dati pseudokod nacrtati graf toka kontrole i odrediti koliko je testova potrebno kako bi se postigla pokrivenost iskaza, pokrivenost odluka i pokrivenost putanja.

Kod:

```
Read P
Read Q
IF P+Q > 10 THEN
Print "Large"
ENDIF
If P > 50 THEN
Print "P Large"
ENDIF
```

Rešenje: Graf toka kontrole za dati kod je prikazan na slici 4.20. Čvorovi grafa su označeni brojevima, dok su grane grafa označene slovima abecede. Na početku, potrebno je odrediti broj testova potrebnih da se pokriju svi iskazi. Sa slike se jasno može uočiti da su u oba IF uslova false grane prazne (bez naredbi), pa je za pokrivenost iskaza dovoljno pokriti samo true grane. Dovoljan je jedan test, koji će pokriti sledeću putanju:

1A-2C-3D-E-4G-5H



Slika 4.20. Graf toka kontrole za primer 1

Kada se posmatra pokrivenost odluka, potrebno je pokriti sve izlazne grane svih IF uslova bar jedanput. Izlazne false grane oba IF uslova iz primera nisu pokrivene testom koji pokriva iskaze, pa je neophodno dodati još testova. Konkretno, dovoljno je dodati jedan test, koji bi u oba IF uslova pokrio false grane. Za pokrivenost odluka su, dakle, potrebna dva testa koji pokrivaju putanje:

1A-2C-3D-E-4G-5H

1A-2B-E-4F

U slučaju pokrivanja putanja, potrebno je pokriti sve putanje kroz kod. Na konkretnom primeru, postoje 4 putanje koje je potrebno pokriti testovima:

1A-2C-3D-E-4G-5H

1A-2B-E-4F

1A-2B-E-4G-5H

1A-2C-3D-E-4F

Ukoliko sa PI označimo pokrivanje iskaza, sa PO pokrivanje odluka, a sa PP pokrivanje putanja, kao rešenje zadatka dobijamo:

PI = 1, PO = 2 i PP = 4

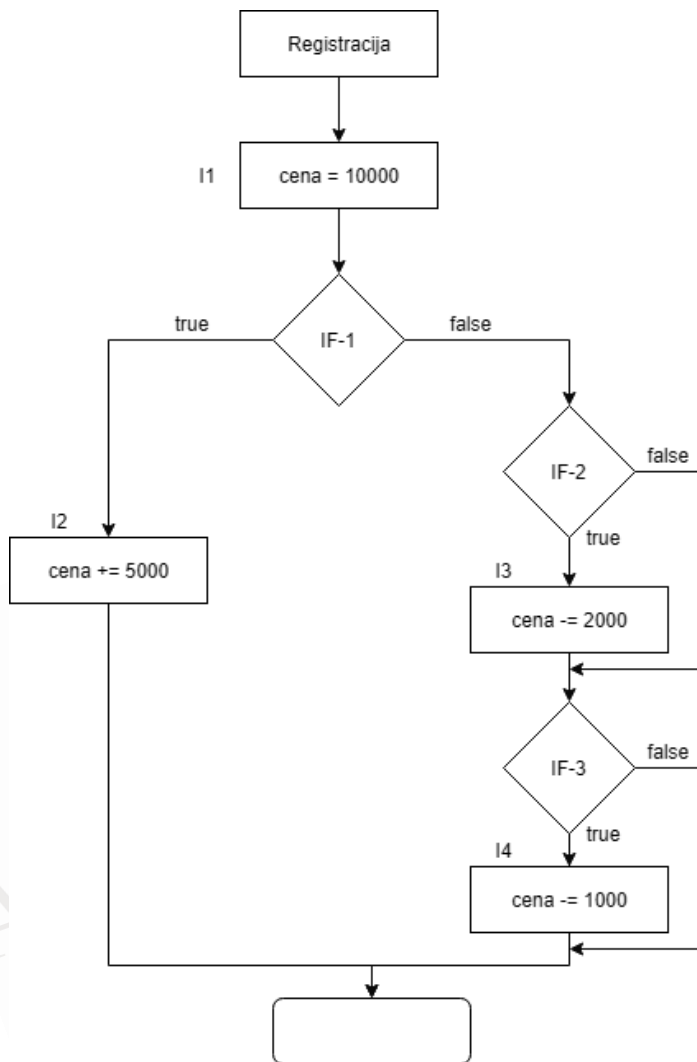
Primer 2: Za datu metodu izracunavanjeRegistracije u programskom jeziku Java odrediti skupove testova koji pokrivaju:

- Iskaze
- Odluke
- Uslove
- Uslove i odluke
- Putanje

Nakon određivanja skupova testova, realizovati odgovarajuće testove pomoću alata JUnit.

```
public class Registracija {  
    public enum Gorivo {BENZIN, DIZEL}  
    public static final int OSNOVA = 10000;  
    public int izracunavanjeRegistracije (int starost, Gorivo gorivo,  
double kubikaza){  
        int cena = OSNOVA;  
        if ((starost < 5) && (gorivo == Gorivo.BENZIN) && kubikaza >  
2000){  
            cena += 5000;  
        } else {  
            if (kubikaza < 1800 || (gorivo == Gorivo.DIZEL))  
                cena -= 2000;  
            if (starost > 10)  
                cena -= 1000;  
        }  
        return cena;  
    }  
}
```

Rešenje: Graf toka kontrole za datu metodu je prikazan na slici 4.21. Za pokrivanje iskaza, neophodno je testovima pokriti true granu IF-1 uslova gde se nalazi naredba I2, kao i false granu IF-1, true granu IF-2 i true granu IF-3 uslova, kako bi se pokrili iskazi I3 i I4. Za to su dovoljna dva testa. Na slici 4.22 prikazani su testovi koji obezbeđuju 100% pokrivenost iskaza.



Slika 4.21. Graf toka kontrole za metodu izracunavanjeRegistracije

Iskazi	starost	gorivo	kubikaza	Test	Očekivani izlaz
Iskazi1	<5	Benzin	>2000	2, Benzin, 2500	15000
Iskazi2	>10	Dizel	<1800	12, Dizel, 1400	7000

Slika 4.22. Testovi koji obezbeđuju pokrivenost iskaza metode izracunavanjeRegistracije

Na osnovu vrednosti parametara u tabeli možemo kreirati odgovarajuće JUnit testove:

```
/**
 * Pokrivanje iskaza
 */
@Test
public void testRegistracijaIskaz1() {
    Registracija r = new Registracija ();
    assertEquals (15000, r.izracunavanjeRegistracije(2,
Registracija.Gorivo.BENZIN, 2500));
}

/**
 * Pokrivanje iskaza
 */
@Test
public void testRegistracijaIskaz2() {
    Registracija r = new Registracija ();
    assertEquals (7000, r.izracunavanjeRegistracije(12,
Registracija.Gorivo.DIZEL, 1400));
}
```

Kada je u pitanju pokrivanje odluka, svaka odluka se mora izvršiti i kao true i kao false, kako bi se sve izlazne grane uslova izvršile bar jedanput. U primeru, neophodno je dodatno pokriti false grane IF-2 i IF-3 uslova, kao što je prikazano na slici 4.23. Za pokrivanje odluka neophodno je napisati 3 testa.

Odluke	starost	gorivo	kubikaza	Test	Očekivani izlaz
Odluke1	<5	Benzin	>2000	2, Benzin, 2500	15000
Odluke2	>10	Dizel	<1800	12, Dizel, 1400	7000
Odluke3	>5, <10	Benzin	>1800	8, Benzin, 2000	10000

Slika 4.23. Testovi koji obezbeđuju pokrivenost odluka metode izracunavanjeRegistracije

Na osnovu vrednosti parametara iz tabele, moguće je napisati odgovarajuće JUnit testove. Može se primetiti da su prva dva testa iz tabele identični kao testovi za pokrivanje iskaza. Jedini dodatak je test koji pokriva false grane IF2 i IF3. Testovi koji pokrivaju odluke su dati u nastavku:

```

/**
 * Pokrivanje odluka
 */
@Test
public void testRegistracijaOdluke1() {
    Registracija r = new Registracija ();
    assertEquals (15000, r.izracunavanjeRegistracije(2,
Registracija.Gorivo.BENZIN, 2500));
}
/**
 * Pokrivanje odluka
 */
@Test
public void testRegistracijaOdluke2() {
    Registracija r = new Registracija ();
    assertEquals (7000, r.izracunavanjeRegistracije(12,
Registracija.Gorivo.DIZEL, 1400));
}

/**
 * Pokrivanje odluka
 */
@Test
public void testRegistracijaOdluke3() {
    Registracija r = new Registracija ();
    assertEquals (10000, r.izracunavanjeRegistracije(8,
Registracija.Gorivo.BENZIN, 2000));
}

```

Kod pokrivanja uslova, svi uslovi (i njihove negacije) moraju biti ispunjeni bar jednom. U slučaju složenih uslova, nezavisno se posmatraju elementarni uslovi. Neophodni testovi za pokrivanje uslova su prikazani na slici 4.24. Za pokrivanje uslova dovoljna su 3 testa.

Uslovi	starost	gorivo	kubikaza	Test	Očekivani izlaz
Uslovi1	>10	Dizel	>2000	12, Dizel, 2500	7000
Uslovi2	<5	Benzin	<1800	2, Benzin, 1400	8000
Uslovi3	>5, <10	Dizel	>1800	8, Dizel, 2000	8000

Slika 4.24. Testovi koji obezbeđuju pokrivanje uslova metode izracunavanjeRegistracije

Pokrivanje uslova ne garantuje ni pokrivanje odluka, ni pokrivanje iskaza. Pokrivanje uslova i pokrivanje odluka biće isti isključivo kada se sve odluke sastoje od jednostavnih elementarnih izraza. Može se lako uočiti da iako testovi sa slike 4.24 ispunjavaju pokrivanje uslova, ne izvršavaju se true klauzula IF-1, kao ni prazna ELSE klauzula IF-2. Na osnovu vrednosti parametara sa slike 4.24, mogu se napisati odgovarajući JUnit testovi, koji su dati u nastavku.

```

/**
 * Pokrivanje uslova
 */
@Test
public void testRegistracijaUslov1() {
    Registracija r = new Registracija ();
    assertEquals (7000, r.izracunavanjeRegistracije(12,
Registracija.Gorivo.DIZEL, 2500));
}

/**
 * Pokrivanje uslova
 */
@Test
public void testRegistracijaUslov2() {
    Registracija r = new Registracija ();
    assertEquals (8000, r.izracunavanjeRegistracije(2,
Registracija.Gorivo.BENZIN, 1400));
}

/**
 * Pokrivanje uslova
 */
@Test
public void testRegistracijaUslov3() {
    Registracija r = new Registracija ();
    assertEquals (8000, r.izracunavanjeRegistracije(8,
Registracija.Gorivo.DIZEL, 2000));
}

```

Pokrivanje uslova i odluka podrazumeva da svaka izlazna grana uslovnih iskaza mora biti bar jednom izvršena, kao i da svaki uslov i njegova negacija moraju biti zadovoljeni bar jedanput. Tabela sa slike 4.25 se dobija spajanjem tabela za pokrivanje odluka i pokrivanje uslova, i onda se minimizuje broj testova. Dovoljna su tri testa, i oni su identični kao testovi prikazani u pokrivanju odluka.

Uslovi i odluke	starost	gorivo	kubikaza	Test	Očekivani izlaz
UsloviOdluke1	<5	Benzin	>2000	2, Benzin, 2500	15000
UsloviOdluke2	>10	Dizel	<1800	12, Dizel, 1400	7000
UsloviOdluke3	>5, <10	Benzin	>1800	8, Benzin, 2000	10000

Slika 4.25. Testovi koji obezbeđuju pokrivanje uslova i odluka metode izracunavanjeRegistracije

Pokrivanje putanja podrazumeva pokrivanje svih bazičnih putanja kroz kod. Za datu metodu, postoji 5 nezavisnih putanja, kao što je prikazano na slici 4.26.

Putanje	starost	gorivo	kubikaza	Test	Očekivani izlaz
Putanje1	<5	Benzin	>2000	2, Benzin, 2500	15000
Putanje2	>10	Dizel	<1800	12, Dizel, 1400	7000
Putanje3	>5, <10	Benzin	>1800	8, Benzin, 2000	10000
Putanje4	<5	Benzin	<1800	2, Benzin, 1400	8000
Putanje5	>10	Benzin	>1800	12, Benzin, 2000	9000

Slika 4.26. Testovi koji obezbeđuju pokrivanje putanja metode izracunavanjeRegistracije

Na osnovu vrednosti parametara, mogu se generisati JUnit testovi.

```

/**
 * Pokrivanje putanja
 */
@Test
public void testRegistracijaPutanja1() {
    Registracija r = new Registracija ();
    assertEquals (15000, r.izracunavanjeRegistracije(2,
Registracija.Gorivo.BENZIN, 2500));
}

/**
 * Pokrivanje putanja
 */
@Test
public void testRegistracijaPutanja2() {
    Registracija r = new Registracija ();
    assertEquals (7000, r.izracunavanjeRegistracije(12,
Registracija.Gorivo.DIZEL, 1400));
}

/**
 * Pokrivanje putanja
 */
@Test
public void testRegistracijaPutanja3() {
    Registracija r = new Registracija ();
    assertEquals (10000, r.izracunavanjeRegistracije(8,
Registracija.Gorivo.BENZIN, 2000));
}
}

```

```

    * Pokrivanje putanja
    */
    @Test
    public void testRegistracijaPutanja4() {
        Registracija r = new Registracija ();
        assertEquals (8000, r.izracunavanjeRegistracije(2,
Registracija.Gorivo.BENZIN, 1400));
    }

    /**
    * Pokrivanje putanja
    */
    @Test
    public void testRegistracijaPutanja5() {
        Registracija r = new Registracija ();
        assertEquals (9000, r.izracunavanjeRegistracije(12,
Registracija.Gorivo.BENZIN, 2000));
    }

```

Primer 3: U klasi Postarina.java, posmatra se metoda izracunajCenuPaketa. Ova metoda kao ulazne parametre prima masu paketa, tip slanja (preporučena ili obična pošiljka) i indikator da li je paket lomljiv ili ne. Cena paketa se formira na osnovu mase paketa i druga dva parametra na način definisan u tabeli na slici 4.27. U slučaju da je paket preporučen, na izračunatu cenu se dodaje još 50%. Potrebno je napisati JUnit testove koji:

- Pokrivaju sve iskaze (100% pokrivenost iskaza)
- Pokrivaju sve odluke (100% pokrivenost odluka)
- Pokrivaju sve putanje (100% pokrivenost putanja)

Stopa mase u kg	Cena	Lomljivi paket (dodaje se na cenu)
Do 3kg	206,00	286,00
Od 3 do 5kg	216,00	286,00
Preko 5kg	+10,00 za svaki dodatni kg	286,00 + 15,00 za svaki dodatni kg

Slika 4.27. Formiranje cene paketa na osnovu datih parametara

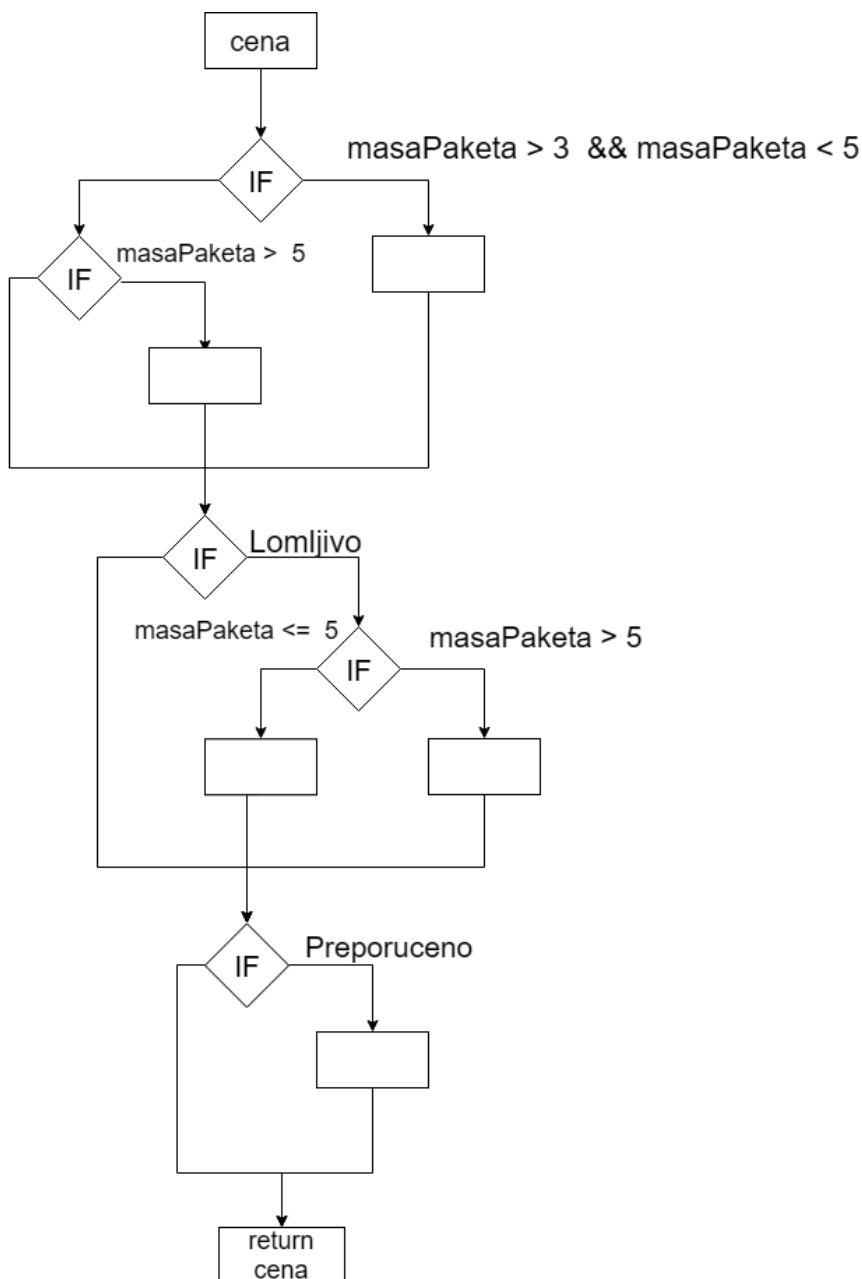
Kod metode izracunajCenuPaketa je dat sa:

```

public class Postarina {
    public enum TipSlanja {PREPORUCENO, OBICNA_POSILJKA}
    //slanje paketa
    //Stope mase (u kg) cena lomljivi ili glomazni paket (dodatno na
cenu)
    //do 3                206,00                286,00
    //preko 3 do 5        216,00                286,00
    //preko 5            +10,00 za svaki dodatni kg +15,00 za svaki
dodatni kg
    //ukoliko je paket preporucen, dodaje se jos 50% na cenu
    public double izracunajCenuPaketa (double masaPaketa, TipSlanja
tip, boolean lomljivo){
        double cena = 206;
        if (masaPaketa >=3 && masaPaketa < 5){
            cena += 10;
        } else if (masaPaketa >= 5){
            cena += Math.ceil(masaPaketa - 5) * 10;
        }
        if (lomljivo){
            if (masaPaketa < 5)
                cena += 286;
            else
                //ovde fali + 286, greska u kodu
                cena += Math.ceil(masaPaketa - 5) * 15;
        }
        if (tip == TipSlanja.PREPORUCENO)
            cena *= 1.5;
        return cena;
    }
}

```

Rešenje: graf toka kontrole za datu metodu je prikazan na slici 4.28.



Slika 4.28. Graf toka kontrole za metodu izracunajCenuPaketa

Na osnovu grafa toka kontrole, formiraju se JUnit testovi. Sa grafa toka kontrole se može uočiti da su potrebna dva testa kako bi se postigla pokrivenost iskaza, 3 testa za pokrivanje odluka, a ukupno 12 testova za pokrivanje putanja.

```

package posta;

import org.junit.Test;
import static org.junit.Assert.*;

public class PostarinaTest {

    public PostarinaTest() {
    }

    // pokrivanje iskaza

    @Test
    public void testIzracunajCenuPaketaPI1() {
        double masaPaketa = 4.0;
        Postarina.TipSlanja tip = Postarina.TipSlanja.PREPORUCENO;
        boolean lomljivo = true;
        Postarina instance = new Postarina();
        double expectedResult = 753.0;
        double result = instance.izracunajCenuPaketa(masaPaketa, tip,
lomljivo);
        assertEquals(expectedResult, result, 0.0);
    }

    @Test
    public void testIzracunajCenuPaketaPI2() {
        double masaPaketa = 6.0;
        Postarina.TipSlanja tip = Postarina.TipSlanja.PREPORUCENO;
        boolean lomljivo = true;
        Postarina instance = new Postarina();
        double expectedResult = 775.5;
        double result = instance.izracunajCenuPaketa(masaPaketa, tip,
lomljivo);
        assertEquals(expectedResult, result, 0.0);
    }

    //pokrivanje odluka
    //Prethodno (pokrivanje iskaza) plus ovo
    @Test
    public void testIzracunajCenuPaketaPO1() {
        double masaPaketa = 2.0;
        Postarina.TipSlanja tip = Postarina.TipSlanja.OBICNA_POSILJKA;
        boolean lomljivo = false;
        Postarina instance = new Postarina();
        double expectedResult = 206.0;
        double result = instance.izracunajCenuPaketa(masaPaketa, tip,
lomljivo);
        assertEquals(expectedResult, result, 0.0);
    }
}

```

```

//Pokrivanje putanja
//sve prethodno plus
@Test
public void testIzracunajCenuPaketaPP1() {
    double masaPaketa = 4.0;
    Postarina.TipSlanja tip = Postarina.TipSlanja.OBICNA_POSILJKA;
    boolean lomljivo = false;
    Postarina instance = new Postarina();
    double expectedResult = 216.0;
    double result = instance.izracunajCenuPaketa(masaPaketa, tip,
lomljivo);
    assertEquals(expectedResult, result, 0.0);
}
@Test
public void testIzracunajCenuPaketaPP2() {
    double masaPaketa = 4.0;
    Postarina.TipSlanja tip = Postarina.TipSlanja.PREPORUCENO;
    boolean lomljivo = false;
    Postarina instance = new Postarina();
    double expectedResult = 324.0;
    double result = instance.izracunajCenuPaketa(masaPaketa, tip,
lomljivo);
    assertEquals(expectedResult, result, 0.0);
}
@Test
public void testIzracunajCenuPaketaPP3() {
    double masaPaketa = 4.0;
    Postarina.TipSlanja tip = Postarina.TipSlanja.OBICNA_POSILJKA;
    boolean lomljivo = true;
    Postarina instance = new Postarina();
    double expectedResult = 502.0;
    double result = instance.izracunajCenuPaketa(masaPaketa, tip,
lomljivo);
    assertEquals(expectedResult, result, 0.0);
}
@Test
public void testIzracunajCenuPaketaPP4() {
    double masaPaketa = 6.0;
    Postarina.TipSlanja tip = Postarina.TipSlanja.OBICNA_POSILJKA;
    boolean lomljivo = false;
    Postarina instance = new Postarina();
    double expectedResult = 216.0;
    double result = instance.izracunajCenuPaketa(masaPaketa, tip,
lomljivo);
    assertEquals(expectedResult, result, 0.0);
}

```

```

@Test
public void testIzracunajCenuPaketaPP5() {
    double masaPaketa = 6.0;
    Postarina.TipSlanja tip = Postarina.TipSlanja.PREPORUCENO;
    boolean lomljivo = false;
    Postarina instance = new Postarina();
    double expectedResult = 324.0;
    double result = instance.izracunajCenuPaketa(masaPaketa, tip,
lomljivo);
    assertEquals(expectedResult, result, 0.0);
}

@Test
public void testIzracunajCenuPaketaPP6() {
    double masaPaketa = 6.0;
    Postarina.TipSlanja tip = Postarina.TipSlanja.OBICNA_POSILJKA;
    boolean lomljivo = true;
    Postarina instance = new Postarina();
    double expectedResult = 499.0;
    double result = instance.izracunajCenuPaketa(masaPaketa, tip,
lomljivo);
    assertEquals(expectedResult, result, 0.0);
}

@Test
public void testIzracunajCenuPaketaPP7() {
    double masaPaketa = 2.0;
    Postarina.TipSlanja tip = Postarina.TipSlanja.PREPORUCENO;
    boolean lomljivo = false;
    Postarina instance = new Postarina();
    double expectedResult = 309.0;
    double result = instance.izracunajCenuPaketa(masaPaketa, tip,
lomljivo);
    assertEquals(expectedResult, result, 0.0);
}

@Test
public void testIzracunajCenuPaketaPP8() {
    double masaPaketa = 2.0;
    Postarina.TipSlanja tip = Postarina.TipSlanja.PREPORUCENO;
    boolean lomljivo = true;
    Postarina instance = new Postarina();
    double expectedResult = 738.0;
    double result = instance.izracunajCenuPaketa(masaPaketa, tip,
lomljivo);
    assertEquals(expectedResult, result, 0.0);
}

```

```
@Test
public void testIzracunajCenuPaketaPP9() {
    double masaPaketa = 2.0;
    Postarina.TipSlanja tip = Postarina.TipSlanja.OBICNA_POSILJKA;
    boolean lomljivo = true;
    Postarina instance = new Postarina();
    double expectedResult = 492.0;
    double result = instance.izracunajCenuPaketa(masaPaketa, tip,
lomljivo);
    assertEquals(expectedResult, result, 0.0);
}
}
```



Pitanja

1. Šta je strukturno testiranje?
2. Koje metode bele kutije postoje?
3. Šta je graf toka kontrole?
4. Kako se definiše pojam pokrivenosti?
5. Šta se koristi kao osnova za pisanje testova metodama bele kutije?
6. Da li se proveravaju zahtevi sistema pri testiranju metodama bele kutije?
7. Šta je cilj kod pokrivanja iskaza?
8. Na čemu se zasniva pokrivanje odluka?
9. Šta obezbeđuje pokrivanje putanja?
10. Koji je odnos između pokrivanja iskaza, odluka i putanja?
11. Da li pokrivanje uslova garantuje pokrivanje odluka?
12. Zbog čega se koristi pokrivanje uslova i odluka?
13. Ukoliko je graf toka kontrole dat sa $G = (V, E)$, gde je V skup čvorova, a E skup grana, ciklomatska kompleksnost za dati graf se računa na koji način?
14. Kod metoda toka podataka, o čemu se najviše vodi računa?
15. U metodama toka podataka, u kojim slučajevima se smatra da je podatak upotrebljen?
16. Šta je DU putanja?
17. Šta znači pokrivanje svih DU parova?
18. Šta znači pokrivanje svih definicija?

5. INTEGRACIONO TESTIRANJE

Testiranje softvera se sastoji od tri jasno određena nivoa testiranja – jediničnog, integracionog i sistemskog testiranja. Svaki od ovih nivoa testiranja ima svoje jedinstvene probleme i ciljeve.

Jedinično testiranje podrazumeva individualno testiranje jedinice programskog izvornog koda. U ovom slučaju, pod jedinicom se smatra najmanji deo programa koji se može testirati. U objektno orijentisanom programiranju, jedinica je najčešće jedna metoda ili jedna klasa, a jedinično testiranje najčešće izvode sami programeri koji su tu jedinicu i napisali. Nakon završenog jediničnog testiranja, sledi faza integracionog testiranja. Na kraju integracionog testiranja, testira se kompletan sistem kao celina – faza koja se naziva sistemsko testiranje i o kojoj će biti reči u narednim poglavljima. Integraciono testiranje je, na žalost, faza na koju se najčešće ne obraća dovoljno pažnje i koja se, u praksi, često zanemari i ne uradi na adekvatan način, što može dovesti do čitavog niza problema.

Integraciono testiranje je faza u testiranju softvera, gde se pojedinačni moduli i jedinice spajaju i testiraju zajedno kao celina. Pošto faza integracionog testiranja sledi iza faze jediničnog testiranja, komponente koje ulaze u integraciono testiranje su već uspešno prošle jedinično testiranje. One se grupišu u veće celine nad kojima se izvršavaju posebno pripremljeni testovi. Logično, može se postaviti pitanje zašto je integraciono testiranje uopšte neophodno, kada su sve jedinice i komponente već detaljno testirane na jediničnom nivou. Odgovor leži u činjenici da sada više komponenti rade zajedno prvi put, i neizbežno je da će se pojaviti neki novi problemi.

Kao što je u uvodnom delu ove knjige pomenuto, u septembru 1999, misija Mars Climate Orbiter je doživela krah nakon 41 nedelje leta ka Marsu i pređenih 669 miliona kilometara. Letelica je nestala u trenutku kada je trebalo da počne da kruži oko Marsa. Problem koji je doveo do neuspeha misije je morao biti otkriven integracionim testiranjem, koje je urađeno trajavo. Na razvoju mlaznih motora radila su dva potpuno odvojena tima. Prvi tim (Lockheed Martin Astronautics) je računao potisak u imperijalnim jedinicama (funte), dok je drugi razvojni tim (Jet Propulsion Laboratory) radio u metričkim jedinicama pri proračunima, a kao rezultat mlazni motori su imali 4.45 puta jači potisak od potrebnog. Letelica je prebrzo ušla u orbitu, nakon čega je oštećena i gubi joj se svaki trag.

Iz prethodnog primera se lako može uočiti da je najveći problem prilikom integracije komponenti u njihovom povezivanju. Interfejs između komponenti može dovesti do čitavog niza novih problema, koji nisu mogli biti otkriveni na jediničnom nivou, ma koliko detaljno je jedinično testiranje izvršeno. Neki od ovih problema uključuju:

- Podaci se mogu izgubiti prilikom prolaska kroz interfejs između komponenti
- Jedna komponenta može imati negativan uticaj na drugu komponentu
- Određene kombinacije podfunkcija mogu imati nepovoljan uticaj na glavnu funkciju programa
- Određene nepreciznosti, koje su prihvatljive na nivou pojedinačne komponente, nakon međusobnog povezivanja mogu da narastu do neprihvatljivih vrednosti

Integraciono testiranje, dakle, ima za cilj detaljno testiranje integracije i interfejsa između komponenti, a na najvišem nivou uključuje i interakciju sa drugim komponentama celokupnog sistema, poput operativnog sistema, sistema fajlova, kao i hardverskih i softverskih interfejsa između različitih sistema.

Postoji više tehnika integracionog testiranja, koje će biti obrađene u nastavku ovog poglavlja:

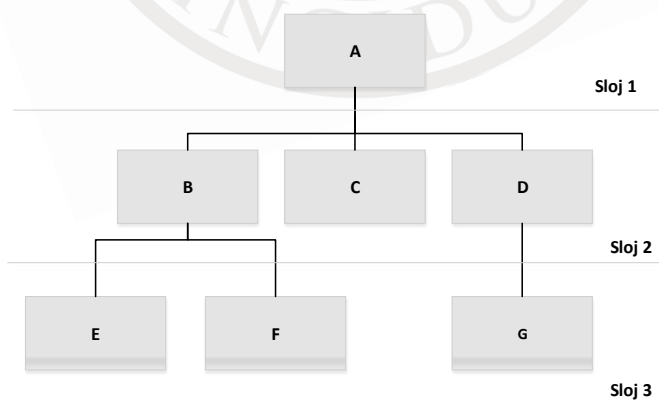
- Veliki prasak integracija (engl. *Big bang* integracija)
- Integracija od vrha ka dnu (engl. *Top-down* integracija)
- Integracija od dna ka vrhu (engl. *Bottom-up* integracija)
- Sendvič integracija (engl. *Sandwich* integracija)
- Integracija po grafu poziva

Pre nego što se pređe na same tehnike integracije, potrebno je objasniti pojmove drajvera (engl. driver) i stabova (engl. stub).

5.1. Stabovi i drajveri

Tester i često imaju zadatak da testiraju softversku komponentu koja je zavisna od drugih modula koji su još uvek u fazi razvoja. U ovakvim slučajevima, kako bi komponenta bila adekvatno testirana, neophodno je napraviti kompjuterski program koji će glumiti nepostojeći modul (odnosno modul koji je još uvek u razvoju i nije dostupan za testiranje). Ovakav kompjuterski program simulira funkcionalnosti nedovršenog modula i omogućava uspešno testiranje.

Arhitektura softverskog sistema se može ilustrovati primerom sa slike 5.1. Na ovom uopštenom primeru se može identifikovati hijerarhija, kao i slojevi sistema.

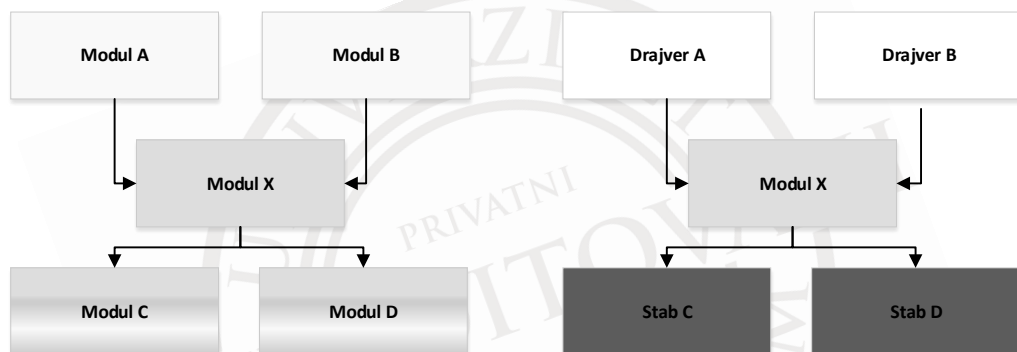


Slika 5.1. Primer hijerarhije softverskog sistema

Neke komponente u hijerarhiji mogu biti nedovršene i nedostupne za testiranje. Umesto sa stvarnim komponentama, komponenta u test okruženju komunicira sa simuliranim komponentama koje je ili pozivaju, ili joj odgovaraju na pozive na isti način kao realne komponente. Prema tome da li pozivaju ili su pozivane, ove komponente se dele na dve vrste:

- drajver – komponenta koja simulira rad modula višeg nivoa, koja poziva druge komponente i očekuju neki odgovor
- stab – modul koji simulira ponašanje modula nižeg nivoa. Simulira rad realnih komponenti koje primaju pozive i vraćaju iste tipove rezultata kao i realne komponente

Modul koji je potrebno testirati se postavlja u test okruženje, kao što je prikazano na slici 5.2.



Slika 5.2. Modul u stvarnom okruženju i u test okruženju

Modul X koji se testira je u stvarnom okruženju pozivan iz modula A i B, koji su u test okruženju zamenjeni drajverima A i B. Sa druge strane, modul X u stvarnom okruženju poziva module C i D, koji su u test okruženju zamenjeni stabovima C i D.

Drajver je kod kroz koji je moguće pozivati druge delove koda ili module koji se testiraju. Glavni program aplikacije se može posmatrati kao drajver, jer pokreće i poziva sve druge komponente programa. Test klase u okviru JUnit okruženja su takođe drajveri, jer pozivaju module programa koji se testiraju. Bez takvog programa, ne bi bilo moguće u potpunosti testirati nijedan modul. Drajveri se intenzivno koriste u integraciji od dna ka vrhu, pošto se u okviru ove tehnike prvo implementiraju i integrišu moduli na dnu hijerarhije, dok moduli višeg nivoa (koji pozivaju module sa dna hijerarhije) još uvek nisu implementirani.

Stabovi se tipično koriste u integraciji od vrha ka dnu, gde se prvo implementiraju i integrišu moduli viših nivoa u hijerarhiji. Kako bi se omogućilo njihovo testiranje, moduli nižeg nivoa koji još uvek nisu implementirani se zamenjuju stabovima. Stabovi mogu biti različitih nivoa kompleksnosti prema logici koju pružaju ka komponenti višeg nivoa:

- Najjednostavniji oblik staba omogućava prosto povezivanje programa, i vraća konstantnu povratnu vrednost na mestima poziva.
- Stab može imati pojednostavljenu logiku stvarne komponente, uz pružanje nekoliko različitih povratnih rezultata.
- Stab može da eksponira parametre i informacije o unutrašnjem stanju komponente koja se testira kako bi se omogućila detaljna analiza (tzv. *spy*).
- Stab može biti u obliku *mock* komponente, koja vrši proveru vrednosti parametara i redosleda pozivanja metoda komponente koja se testira.
- Simulacija kao najkompleksniji oblik, gde se pruža približno ista funkcionalnost konačne komponente. Simulacija se opravdano koristi samo u retkim slučajevima, pošto je najčešće neophodan veliki napor za razvijanje ovakve komponente.

Kao primer drajvera i stabova može se posmatrati aplikacija u obliku socijalne mreže, koja se sastoji od više modula, na primer moduli za prijavu na sistem, uspešna prijava na sistem, neuspešna prijava na sistem, prikaz najnovijih vesti (engl. *news feed*), dodavanje novog prijatelja, uklanjanje prijatelja i slično. Ukoliko je potrebno testirati modul za učitavanje i prikaz najnovijih vesti, može se uočiti da se on ne može izvršavati samostalno – potrebno je prvo uspešno prijaviti korisnika na sistem, pa tek onda izvršiti modul za učitavanje vesti. Ukoliko modul za prijavu korisnika na sistem još uvek nije razvijen od strane programera, potrebno je napraviti drajver koji će se ponašati kao modul za uspešnu prijavu na sistem i na ispravan način pozvati modul za učitavanje vesti, kako bi se testirale njegove funkcionalnosti.

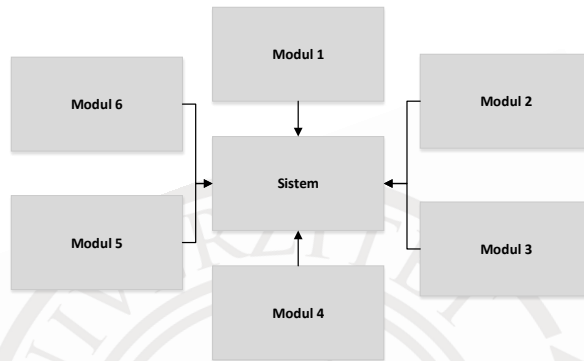
Na primeru iste aplikacije se može prikazati i suprotan slučaj, gde je potrebno testirati modul za prijavu korisnika u situaciji gde moduli za uspešnu i neuspešnu prijavu nisu spremni. Umesto ovih modula se dodaju stabovi, koji će omogućiti uspešno testiranje modula za prijavu korisnika. Ovi stabovi primaju instrukcije od modula za prijavu korisnika i prosto prikazuju poruke o uspešnoj ili neuspešnoj prijavi.

Drajveri i stabovi se u najvećem broju slučajeva programiraju, izuzev situacije gde je drajver zapravo tester koji manuelno pokreće komponente grafičkog interfejsa. U zavisnosti od programskog jezika na kome je napisana komponenta, za implementaciju drajvera se mogu koristiti JUnit (za komponente pisane u Javi), NUnit (za komponente pisane u C#), DBUnit (SQL skriptovi) itd. Sa druge strane, stabovi se implementiraju tako što se programiraju komponente koje simuliraju rad stvarnih komponenti, prilikom čega se definiše šta će biti povratna vrednost za određene ulazne podatke. Postoji veliki broj biblioteka koje značajno olakšavaju proces definisanja stabova, među kojima su najpopularnije za programski jezik Java biblioteke JMock, Mockito itd.

Postoji još jedan aspekt pisanja drajvera i stabova koji je neophodno napomenuti, a to je da i jedni i drugi predstavljaju dodatni trošak na projektu. U pitanju je softver koji se treba implementirati na uobičajen način, ali se ne isporučuju klijentu sa finalnom verzijom programa.

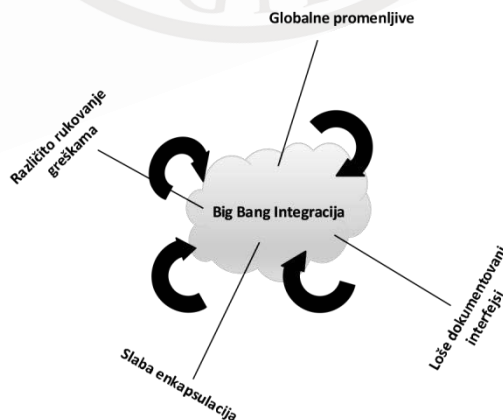
5.2. Big bang integracija

U slučaju big bang interakcije, pojedinačne komponente sistema se razvijaju odvojeno i testiraju na jediničnom nivou. Nakon završetka razvoja i jediničnog testiranja, sve komponente se spajaju i integrišu simultano, a sistem se testira kao celina. Na primeru sa slike 5.3, svi moduli od broja 1 do broja 6 se integrišu istovremeno i zatim testiraju zajedno.



Slika 5.3. Big bang integracija

Na žalost, ovakav pristup integracionom testiranju obično rezultira haosom, kako je prikazano na slici 5.4. U najvećem broju slučajeva, nakon spajanja većeg broja komponenti odjedanput, istovremeno će se pojaviti veliki broj grešaka i problema. Eventualni problemi koji postoje na interfejsima između komponenti se uočavaju jako kasno, na samom kraju razvoja softverskog sistema. Kada se problemi pronađu, vrlo je teško izolovati ih zbog velikog broja komponenti koje su istovremeno integrisane, a postoji i velika šansa da se ne uoče neki kritični defekti, koji će po pravilu isplivati na površinu u produkciji kada ih uoče sami klijenti.

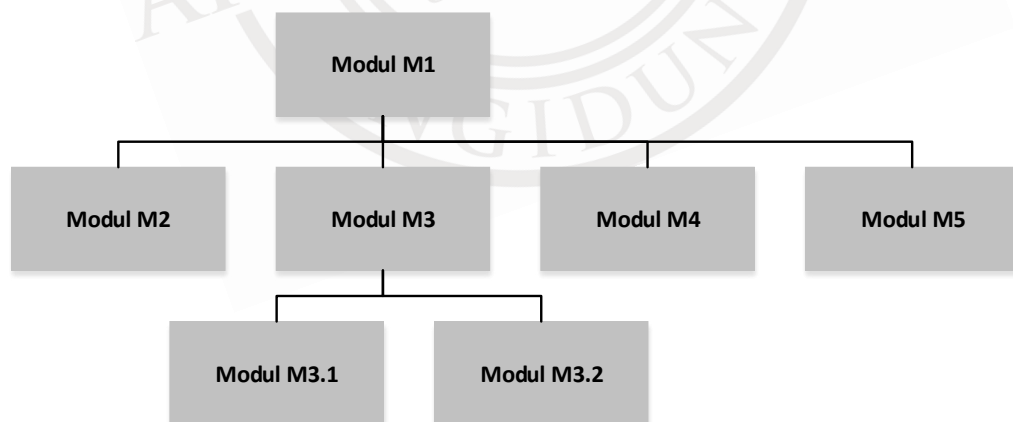


Slika 5.4. Problemi koji obično nastaju prilikom big bang integracije

Iz svega navedenog, može se zaključiti da big bang nije pravi način na koji treba integrisati i testirati softver. Interacija se u ovom slučaju izvršava maltene na slepo, uz najbolju nadu da će sve komponente raditi savršeno kada se spoje sa drugima, što uglavnom nije slučaj. Ovakav pristup integracionom testiranju je primenjiv samo na jako male sisteme, koji se sastoje od svega nekoliko komponenti, dok se ne koristi u slučaju ozbiljnijih i kompleksnijih softverskih sistema. U slučaju kompleksnih aplikacija se koristi inkrementalna integracija. Jedna od retkih prednosti big bang integracije je da su sve komponente implementirane i završene pre nego što počne integracija. Sa druge strane, glavni problem ovakvog pristupa leži u tome da je jako teško izolovati greške u ovako kasnoj fazi razvoja, a njihovo ispravljanje po pravilu oduzima jako mnogo vremena.

5.3. Inkrementalna integracija

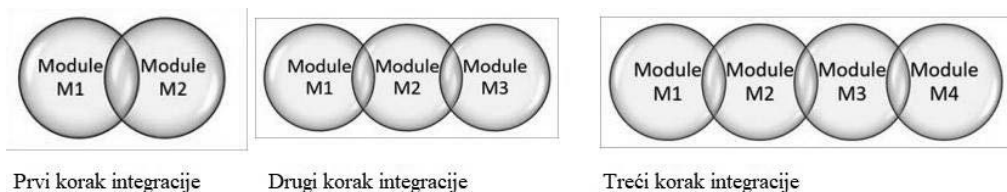
Kod inkrementalne integracije, svaka komponenta se testira individualno u okviru faze jediničnog testiranja, a nakon toga se komponente integrišu inkrementalno i detaljno testiraju kako bi se osigurala ispravnost interfejsa i interakcije između komponenti. Komponente se dodaju inkrementalno, jedna po jedna. Nakon dodavanja komponente, vrši se detaljno testiranje pre dodavanja sledeće komponente. Integrisane komponente se testiraju kao grupa, kako bi se osigurala uspešna integracija i ispravan protok podataka između komponenti. Fokus testiranja su provere interfejsa, integrisanih linkova, i toka informacija između komponenti. Proces se ponavlja do integrisanja i uspešnog testiranja poslednje komponente.



Slika 5.5. Primer softverskog sistema

Za lakše razumevanje koncepta inkrementalne integracije posmatra se softverski sistem sa slike 5.5, koji se sastoji od većeg broja modula. Svaki od modula M1, M2, M3 itd. se testira individualno u okviru jediničnog testiranja. Moduli se integrišu jedan po

jedan i testira se njihova međusobna interakcija. Prvo se integrišu i testiraju moduli M1 i M2, kao što je prikazano na slici 5.6. Zatim se dodaje modul M3 i detaljno testira. Sledeći se dodaje modul 4 i ponovo se izvršava testiranje kako bi se osiguralo da svi moduli zajedno rade ispravno. Svi preostali moduli se takođe inkrementalno dodaju jedan po jedan, i u svakom koraku se vrši testiranje kako bi se osigurala uspešna integracija.



Slika 5.6. Inkrementalna integracija

Glavni ciljevi inkrementalne integracije su osiguravanje da različite komponente uspešno rade zajedno nakon integracije, kao i identifikacija grešaka vrlo rano u fazi razvoja softvera. Rana identifikacija grešaka omogućava lakše lociranje problema i njihovu ispravku. Na primeru sa slike 5.6, ukoliko je integraciono testiranje prošlo uspešno prilikom integracije modula M1 i M2, a nakon integracije M3 testovi se ne izvrše uspešno, lako se može uočiti da je problem najverovatnije u modulu M3. Samim tim, problemi se mogu ispraviti u ranoj fazi projekta, uz značajno manji trošak u smislu potrebnog rada i utrošenog vremena u poređenju sa big bang integracijom. Dodatna prednost inkrementalne integracije je da je sistem uvek u relativno funkcionalnom stanju, i može se pokazati klijentu, što odgovara inkrementalnom modelu razvoja softvera, o kojem će biti reči u kasnijim poglavljima.

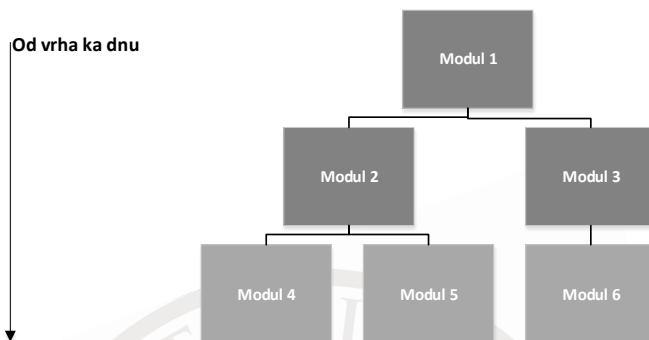
Kao glavna mana inkrementalne integracije se može izdvojiti činjenica da su često potrebni stabovi i drajveri, koji zamenjuju komponente koje još uvek nisu implementirane ili integrisane.

U nastavku ovog poglavlja će biti opisani tipovi inkrementalne integracije koji su bazirani na hijerarhijskoj strukturi programa – integracija od vrha ka dnu, integracija od dna ka vrhu i sendvič integracija.

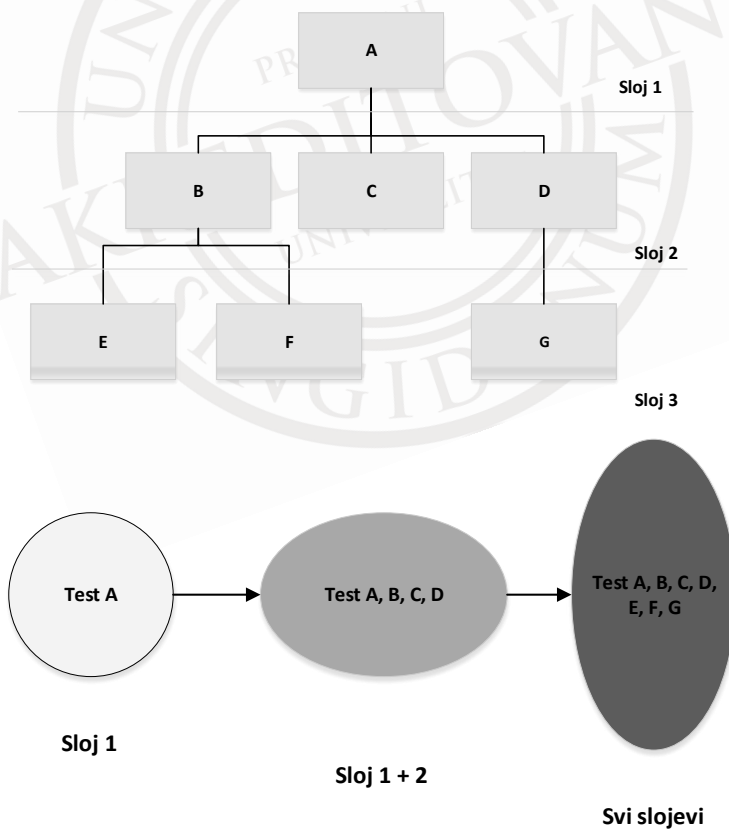
5.3.1. Integracija od vrha ka dnu

Integracija od vrha ka dnu (engl. *top-down integration*) se, kao što joj i ime nagoveštava, zasniva na postupku inkrementalne integracije koja počinje od vrha hijerarhije softverskog sistema, odnosno od glavnog kontrolnog modula, kao što je prikazano na slici 5.7. Nakon testiranja glavnog modula dodaju se niži moduli jedan po jedan. U svakom softverskom sistemu se mogu uočiti nivoi hijerarhije, kao što je

prikazano na slici 5.8. Na istoj slici je prikazan i načelni redosled testiranja modula u integraciji od vrha ka dnu. Prilikom testiranja modula viših nivoa koji se prvi integrišu, niži nivoi se uobičajeno simuliraju stabovima. Prilikom dodavanja svakog modula, stvarna komponenta se postavlja na mesto staba u hijerarhiji.



Slika 5.7. Smer integracije od vrha ka dnu

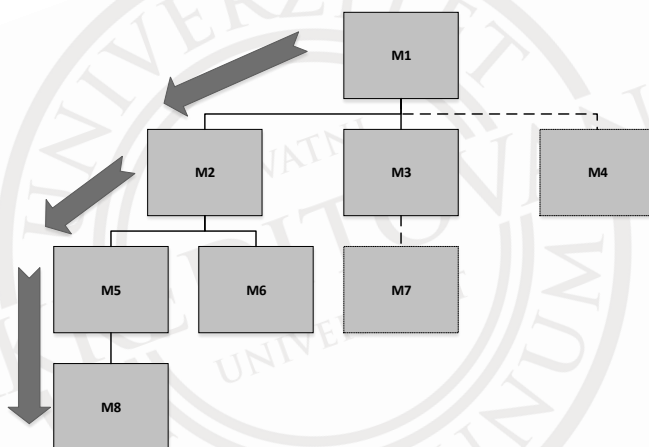


Slika 5.8. Nivoi hijerarhije softverskog sistema i redosled testiranja od vrha ka dnu

Nakon dodavanja i testiranja glavnog modula, postoje dva moguća pristupa integraciji nižih modula:

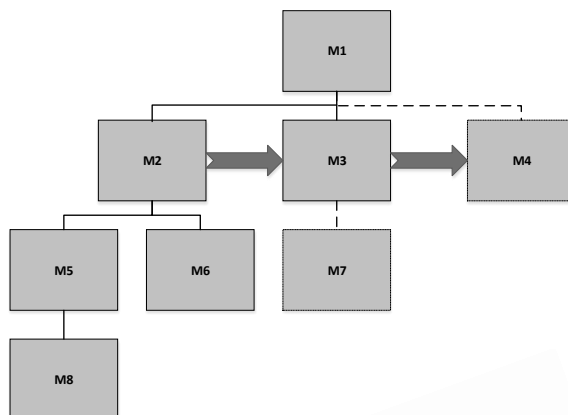
- po dubini,
- po širini.

Pristup po dubini se bazira na integraciji svih komponenti na glavnoj kontrolnoj putanji kroz arhitekturu programa. Sam izbor glavne putanje se bira u zavisnosti od karakteristika specifikacije softverskog sistema - obično je u pitanju najbitnija funkcionalnost sistema, odnosno kritični moduli. Neka je data hijerarhija sistema kao na slici 5.9. Prvo se integriše glavni kontrolni modul M1. Nakon integracije M1 i izbora leve kontrolne putanje, redom će biti integrisani moduli M2 i M5. Nakon toga se integriše M8 (ili M6 pre njega ukoliko je neophodan za ispravan rad modula M2). Nakon toga sledi implementacija i integracija srednje, odnosno desne kontrolne putanje, redosled opet zavisi od specifikacije softverskog sistema.



Slika 5.9. Integracija po dubini

Pristup po širini se zasniva na integraciji svih neposredno podređenih komponenti jednog nivoa, jedne po jedne. Kada se završi sa tim nivoom, prelazi se na sledeći niži nivo. Na primeru hijerarhije sa slike 5.10, nakon integracije modula M1, redom se integrišu moduli sledećeg sloja, M2, M3 i M4. Nakon toga se prelazi na sledeći niži sloj i integrišu redom M5, M6 itd.



Slika 5.10. Integracija po širini

Formalno gledano, proces integracije se može opisati sledećim koracima:

1. Prvo se integriše glavni modul, koji se koristi kao drajver, a umesto svih modula koji su neposredno ispod glavnog modula se postavljaju stabovi.
2. Podređeni stabovi se, jedan po jedan, zamenjuju stvarnim modulima prema odabranom pristupu (po dubini ili po širini).
3. Nakon svake dodate komponente se vrši integraciono testiranje. Po uspešnom završetku testiranja, sledeći stab se zamenjuje stvarnim modulom.
4. Može se sprovesti i regresiono testiranje, kako bi se osiguralo da se nakon ubacivanja novog modula nisu pojavile nove greške u prethodno integrisanim i testiranim modulima.

Algoritam se nastavlja od drugog koraka, sve dok se svi moduli uspešno ne integrišu i ne izgradi kompletna struktura softverskog sistema.

Glavne prednosti integracije od vrha ka dnu su:

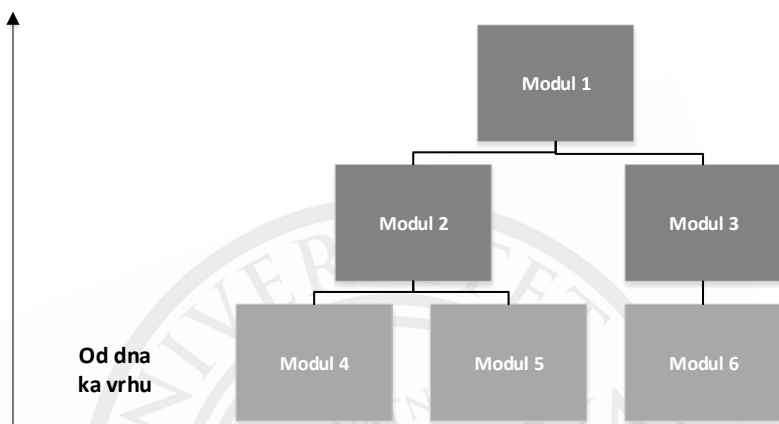
- Testiranje i integracija komponenti počinju veoma rano u razvoju softvera.
- Rano pronalaženje defekata u arhitekturi i dizajnu sistema.
- Glavne kontrolne tačke u softveru se testiraju rano.
- Nije potrebno razvijati drajvere, koji su generalno gledano kompleksniji za razvoj od stabova.
- Paralelni razvoj modula.

Kao mane integracije od vrha ka dnu, mogu se izdvojiti:

- Potreban je veliki broj stabova. Iako su jednostavniji od za pisanje od drajvera, potrebno je uložiti vreme u njihov razvoj.
- Treba držati na umu da stab nije savršena implementacija komponente koju menja, već samo simulira protok podataka između komponenti.
- Stabovi moraju biti dobro napisani, pošto od njih zavisi da li će testiranje biti uspešno.
- Često je vrlo teško definisati uslove testa.

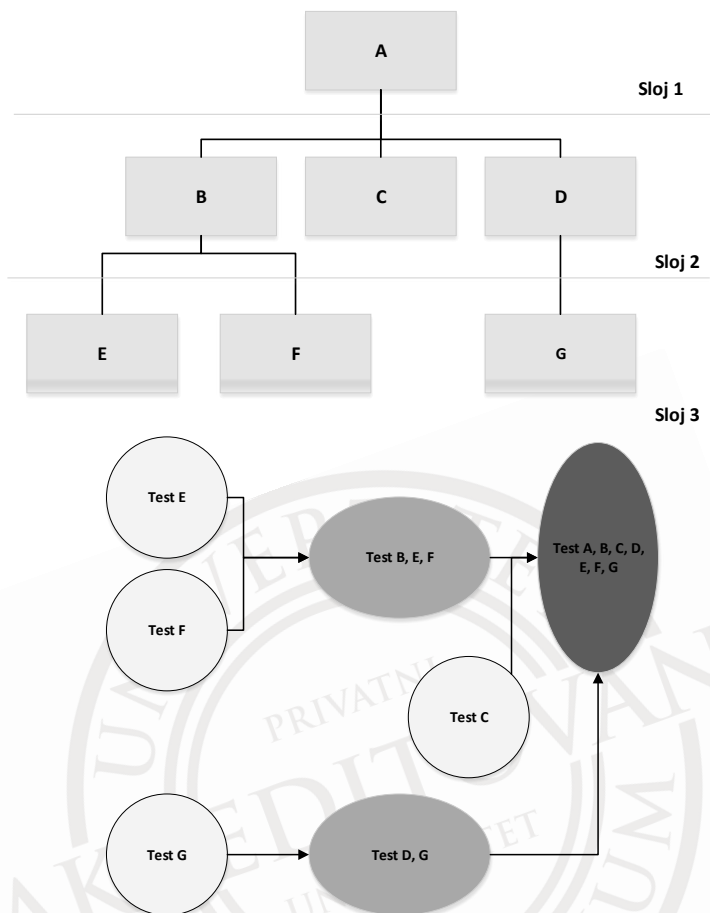
5.3.2. Integracija od dna ka vrhu

Integracija od dna ka vrhu (engl. *bottom-up integration*) se zasniva na postupku inkrementalne integracije koji počinje od dna hijerarhije softverskog sistema, odnosno od najnižih modula u hijerarhiji. Moduli se integrišu i ide se ka gore, sve do glavnog kontrolnog modula, koji se integriše na kraju, kao što je prikazano na slici 5.11.



Slika 5.11. Smer integracije od dna ka vrhu

Na slici 5.12 je prikazan primer arhitekture softverskog sistema, kao i redosled testiranja modula.



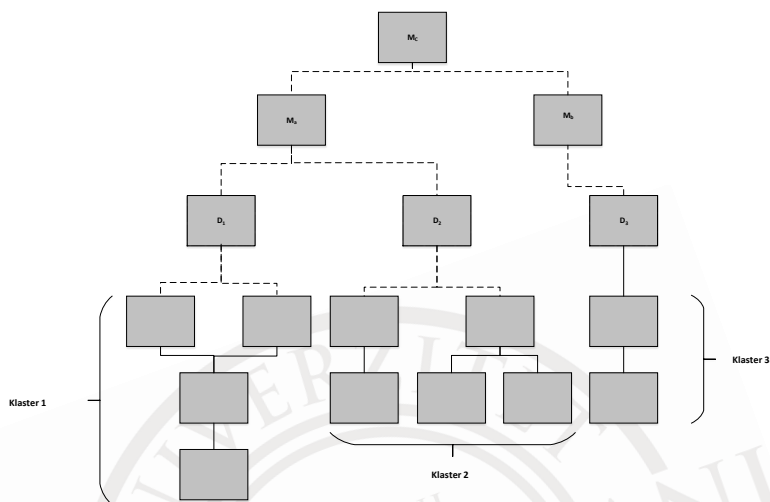
Slika 5.12. Nivoi hijerarhije softverskog sistema i redosled testiranja od dna ka vrhu

U uobičajenom softverskom sistemu, na nižim nivoima hijerarhije je najčešće implementirana obrada podataka, pristup bazi i slično. Pošto se implementacija modula i integracija vrše od dna, ova obrada će biti dostupna rano u razvoju softvera. Može se uočiti i da su u slučaju integracije od dna ka vrhu stabovi nepotrebni. Sa druge strane, neophodni su drajveri, koji će kontrolisati i pozivati module nižih nivoa.

Integracija od dna ka vrhu počinje tako što se uoče podfunkcije sistema. Najčešće je za jednu podfunkciju zaduženo više modula nižih nivoa. Da bi se određena podfunkcija sistema kompletirala, odgovarajući moduli najnižih nivoa se povezuju u klaster. Pošto moduli viših nivoa još uvek nisu spremni, mora se napisati drajver koji će koordinirati rad klastera. Nakon toga se testira klaster, a kada modul višeg nivoa bude implementiran i spreman, drajver se uklanja i umesto njega se postavlja stvarni modul.

Na primeru hijerarhije sa slike 5.12, uočavaju se tri podfunkcije sistema, a komponente koje su odgovorne za te podfunkcije se grupišu u klaster 1, 2 i 3. Svaki od navedenih klastera se testira pojedinačno, upotrebom drajvera D1, D2 i D3 respektivno.

Moduli u klasterima 1 i 2 su podređeni modulu Ma. Po završetku testiranja klastera, drajveri D1 i D2 se uklanjaju, a klasteri 1 i 2 se direktno povezuju na modul Ma kada bude implementiran.



Slika 5.13. Primer integracije od dna ka vrhu

Prednosti integracije od dna ka vrhu su:

- Testiranje počinje rano u razvoju softvera.
- Čim je prvi modul najnižeg nivoa spreman, poželjno je krenuti sa testiranjem.
- Značajno se smanjuje potreba za stabovima (mada i dalje ponekad mogu biti potrebni, na primer simulacija izuzetaka).
- Paralelni razvoj modula.

Kao mane integracije od dna ka vrhu, mogu se izdvojiti:

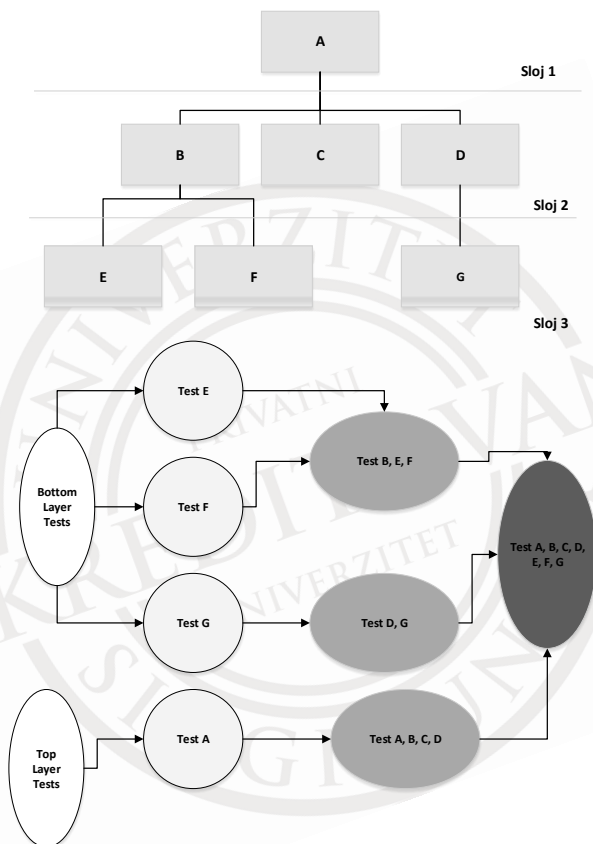
- Veliki trošak u smislu potrebnog vremena za razvoj drajvera, koji su kompleksniji od stabova.
- Drajver ne testira direktno interfejsa ka modulima
- Provera glavnih kontrolnih struktura se odlaže do samog kraja, pošto se glavni kontrolni modul integriše poslednji.
- Defekti u ključnim interfejsima i arhitekturi sistema se otkrivaju kasno.

5.3.3. Sendvič integracija

Sendvič integracija (engl. *sandwich integration*) je hibridni oblik integracije, koji kombinuje metodologije od vrha ka dnu i od dna ka vrhu. Sistem se u ovom slučaju posmatra kao da ima tri sloja:

- Ciljni sloj u sredini.
- Gornji sloj iznad ciljnog, na koji se primenjuje integracija od dna ka vrhu.
- Donji sloj ispod ciljnog, na koji se primenjuje pristup od vrha ka dnu.

Ciljni sloj u sredini se identifikuje primenom heuristike da se što je više moguće smanji broj potrebnih stabova i drajvera. Testiranje konvergira ka sloju u sredini. Na slici 5.14 prikazan je primer arhitekture sistema, kao i redosled testiranja modula.



Slika 5.14. Nivoi hijerarhije softverskog sistema i redosled testiranja u sendvič integraciji

Prednosti sendvič integracije su sledeće:

- Ovaj pristup je veoma pogodan za velike projekte koji se sastoje od različitih podprojekata.
- Pristupi od vrha ka dnu i od dna ka vrhu se mogu izvršavati paralelno.
- Drajeri i stabovi nisu potrebni za gornji i donji sloj, tako da se ukupan broj stabova i drajera smanjuje.
- Integracija modula je moguća odmah nakon implementacije.

Mane sendvič integracije su:

- Još uvek je potreban veliki broj drajvera i stabova.
- Pojedinačni moduli cilnog sloja se ne testiraju detaljno pre integracije.
- Ovaj pristup podseća na integraciono testiranje metodom velikog praska prilikom konvergencije na ciljni sloj u sredini.
- Izolacija defekata je teža.

Modifikovana sendvič integracija se može koristiti kako bi se umanjile negativne strane obične sendvič integracije. Upotrebom modifikovane sendvič integracije, sva tri sloja se individualno testiraju, nakon čega se integrišu zajedno. Testovi za svaki pojedinačni sloj su različiti:

- ciljni sloj se testira sa drajverima i stabovima,
- gornji sloj se testira samo sa stabovima,
- donji sloj se testira samo sa drajverima.

Prilikom integracije slojeva, testovi se sastoje od dve celine:

- Integracija gornjeg sa ciljnim slojem, pri čemu gornji sloj menja drajvere
- Integracija donjeg sa ciljnim slojem, pri čemu donji sloj menja stabove.

5.4. Integracija po grafu poziva

Najveća mana integracije od vrha ka dnu i od dna ka vrhu je veliki broj iteracija testiranja, kao i veličina dodatnog koda koji je neophodan za testiranje, a ne ulazi u isporuku finalnog proizvoda (drajveri i stabovi). Integracija po grafu poziva se zasniva na ideji da se broj iteracija testiranja i dodatni kod redukuju, bez narušavanja testiranja i lakog lociranja defekata.

Kod integracije po grafu poziva, sistem se predstavlja u vidu usmerenog grafa sa sledećim osobinama:

- Komponente softverskog sistema su čvorovi grafa
- Grane grafa su međusobni pozivi komponenti (putem poziva funkcija).

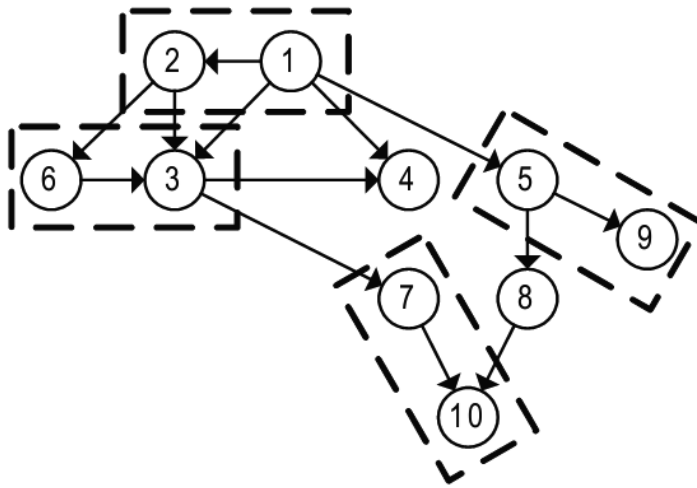
Postoje dva moguća pristupa:

- Integracija prema parovima (engl. *Pair-wise Integration*)
- Integracija prema susedstvu (engl. *Neighborhood Integration*)

U nastavku poglavlja je dat opis obe tehnike.

5.4.1. Integracija prema parovima

Prilikom testiranja integracije se koristi kompletan sistem sa stvarnim kodom, čime se eliminiše potreba za dodatnim kodom poput drajvera i stabova. Svaka iteracija testiranja je ograničena na testiranje ciljanog para komponenti koje su povezane u grafu (jedna komponenta poziva drugu), kao što je prikazano na slici 5.15. Time se redukuje problem izolacije otkrivenih defekata. Ukupan broj potrebnih iteracija integracionog testiranja je jednak broju grana u grafu poziva. U primeru sa slike 5.15 potrebno je ukupno 13 iteracija testiranja.

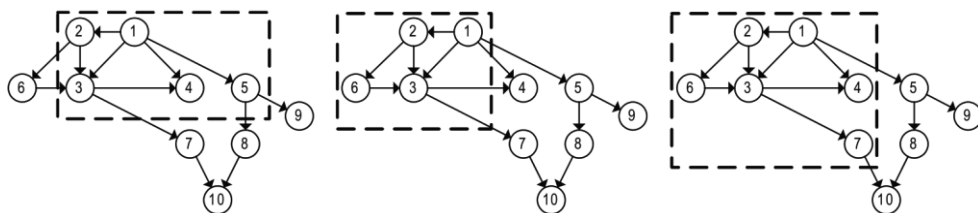


Slika 5.15. Graf poziva sa primerima parova

Glavne prednosti integracije prema parovima su upotreba realnog koda i eliminacija drajvera i stabova. Kao mane se izdvajaju činjenice da može postojati veliki broj test iteracija i da se testiranje vrši kasnije jer je potreban kompletno implementiran sistem.

5.4.2. Integracija prema susedstvu

Integracija prema susedstvu se izvršava na sličan način kao i kod parova, sa tom razlikom da se umesto dve komponente ovde grupišu ciljni čvor i svi susedni čvorovi u grafu. Susedstvo jednog čvora su svi čvorovi koji su u vezi sa ciljnim čvorom, bilo kao prethodnici ili sledbenici, odnosno koji su tačno jednu granu udaljeni od ciljnog čvora. U svakoj iteraciji se testira susedstvo jednog čvora. Na slici 5.16 prikazan je primer grafa poziva, sa označenim susedstvima za komponente pod rednim brojevima 1, 2 i 3.



Slika 5.16. Graf poziva sa obeleženim susedstvima za komponente 1, 2 i 3

Prednosti integracije prema susedstvu su slične kao u slučaju integracije prema parovima. Pošto se i ovde koristi realan kod, eliminiše se potreba za razvojem drajvera i stabova. Dodatno, smanjuje se broj potrebnih iteracija u poređenju sa integracijom po parovima. Kao mane se mogu izdvojiti teža izolacija defekata (pošto se radi tzv. *medium bang integration*, a susedstva mogu biti velika) i kasniji početak testiranja jer je potreban kompletno implementiran sistem.

5.5. Zadaci za vežbu

Primer 1: Posmatra se program koji pruža funkcionalnost kalendara, gde se dan čuva u formatu mm, dd, gggg. Funkcionalnosti sistema su:

- Datum sledećeg dana (getDate)
- Dan u nedelji koji odgovara nekom datumu (weekDay)
- Horoskopski znak datuma (Zodiac)
- Izračunavanje najskorijeg petka 13. (friday13) itd.

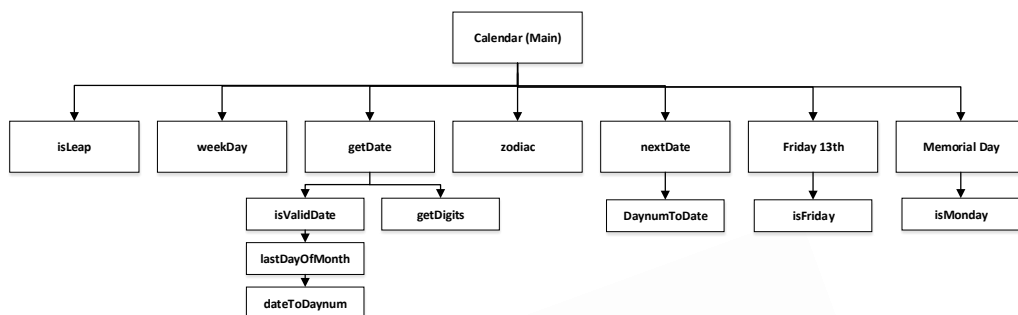
Pregled svih funkcionalnosti sistema je dat na slici 5.16. Potrebno je prikazati postupak integracije svim tehnikama opisanim u prethodnim poglavljima.

```

Main  Calendar
      Function isLeap
      Procedure weekDay
      Procedure getDate
          Function isValidDate
              Function lastDayOfMonth
          Procedure getDigits
      Procedure memorialDay
          Function isMonday
      Procedure friday13th
          Function isFriday
      Procedure nextDate
          Procedure dayNumToDate
      Procedure zodiac
  
```

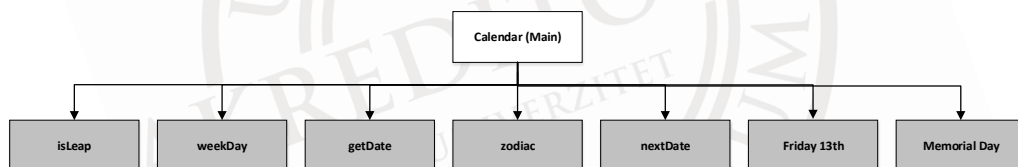
Slika 5.17. Pregled komponenti kalendara

Rešenje: Kako bi se uradila inkrementalna integracija, prvo se radi funkcionalna dekompozicija softverskog sistema koji se testira, kao što je prikazano na slici 5.18.



Slika 5.18. Funkcionalna dekompozicija programa calendar

Nakon funkcionalne dekompozicije, može se pristupiti integraciji komponenti. U slučaju da se koristi integracija od vrha ka dnu, u prvom koraku se integriše glavni modul, dok se svi direktno podređeni moduli menjaju stabovima, kao na slici 5.19 (stabovi i dražveri su označeni sivom bojom).



Slika 5.19. Integracija od vrha ka dnu, prvi korak

Primer jednog stabla, koji menja weekDay komponentu je dat sledećim kodom:

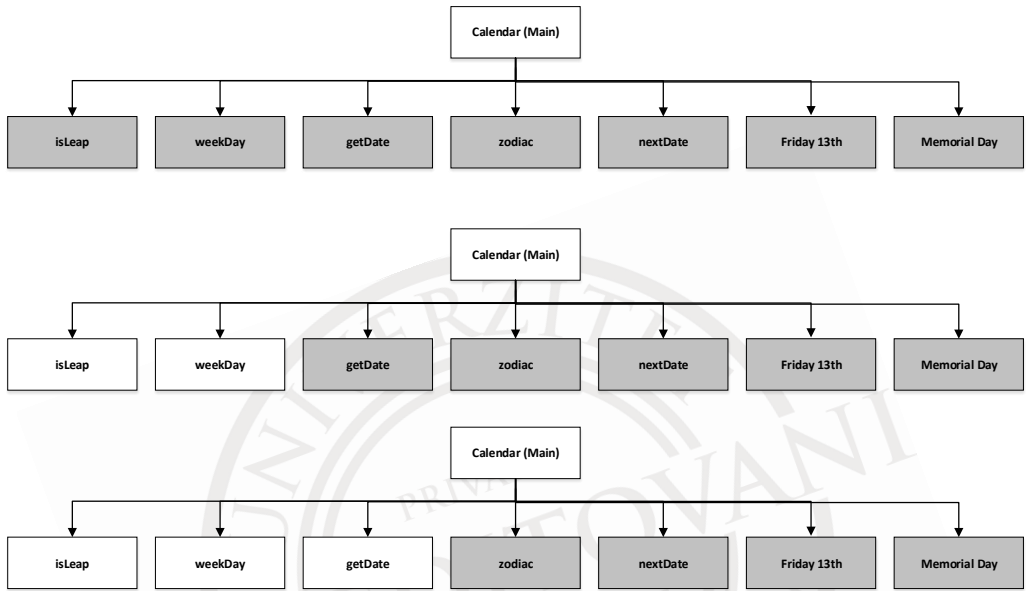
```

Procedure weekDayStub(mm, dd, yyyy, dayName)
  If ((mm = 10) AND (dd = 28) AND (yyyy = 2013))
    Then dayName = "Monday"
  EndIf
  .
  .
  .
  If ((mm = 10) AND (dd = 30) AND (yyyy = 2013))
    Then dayName = "Wednesday"
  EndIf

```

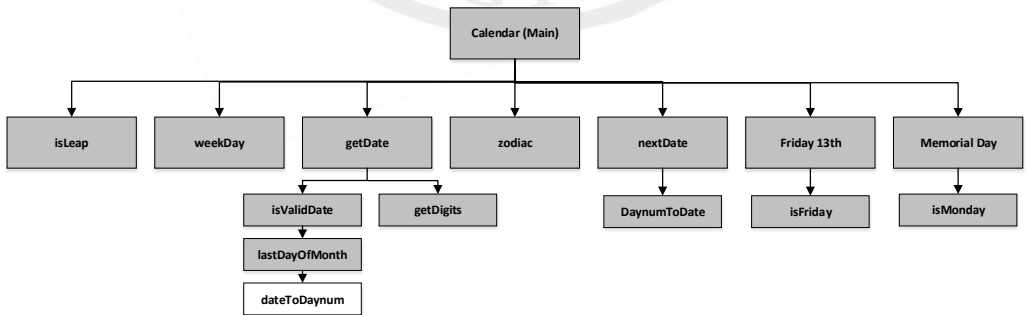
Ovaj stab, u zavisnosti od ulaznih parametara, vraća nekoliko predefinisanih konstantnih vrednosti.

Ukoliko se primenjuje integracija od vrha ka dnu sa pristupom po širini, sledeća tri koraka integracije su prikazana na slici 5.20.



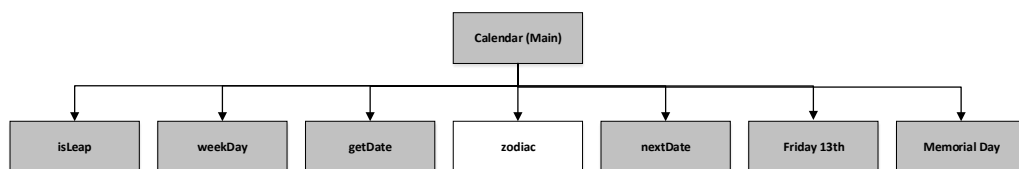
Slika 5.20. Integracija od vrha ka dnu, pristup po širini

U slučaju da se koristi integracija od dna ka vrhu, počinje se od listova u stablu funkcionalne dekompozicije, kao što je prikazno na slici 5.21.



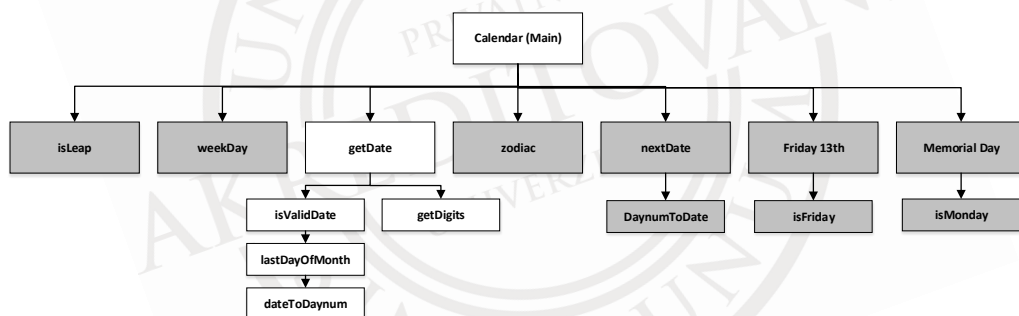
Slika 5.21. Integracija od dna ka vrhu, integracija počinje od komponenti najnižih nivoa

Integracija komponente zodiac tehnikom od dna ka vrhu je prikazana na slici 5.22. Komponenta koja poziva komponentu zodiac je Calendar (glavna komponenta), i ona se zamenjuje drajverom.



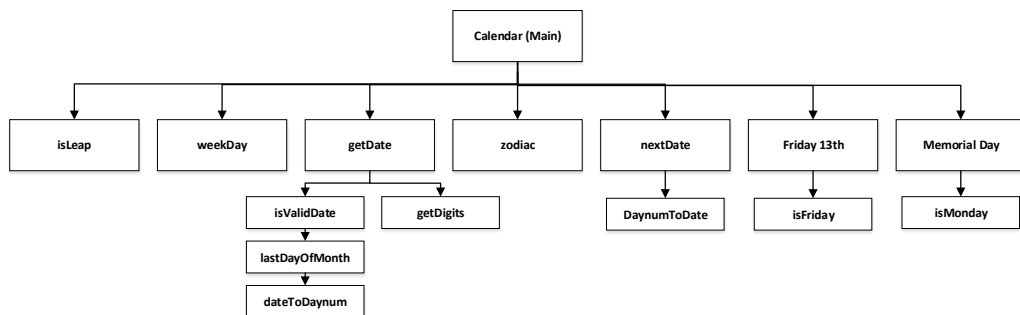
Slika 5.22. Integracija od dna ka vrhu, integrisanje jedne komponente

Ukoliko se koristi sendvič integracija, ciljani sloj u sredini se ponaša kao drajver za komponente donjeg sloja, a kao stab za komponente gornjeg sloja, sa bitnom razlikom da je u pitanju stvarna komponenta i realni kod. Na primeru sa slike 5.23, komponenta getDate se ponaša kao drajver za komponente koje se nalaze ispod nje u hijerarhiji, koje se integrišu od vrha ka dnu. Sa druge strane, komponenta getDate se ponaša kao stab za komponentu Calendar, i integracija se vrši od dna ka vrhu.



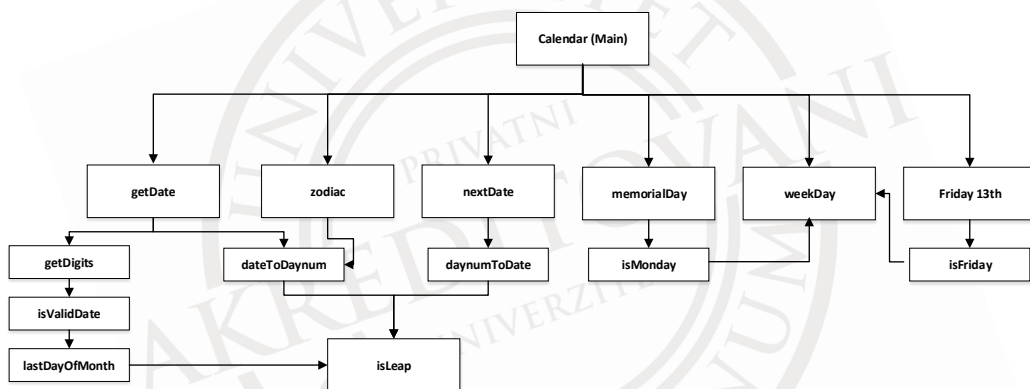
Slika 5.23. Sendvič integracija

U slučaju korišćenja big beng integracije, sve komponente bi se spojile u istom trenutku, kao što je prikazano na slici 5.24. Naravno, ovaj pristup treba izbegavati u slučaju ozbiljnih projekata, kao što je opisano u prethodnim poglavljima, i koristiti neki oblik inkrementalne integracije.



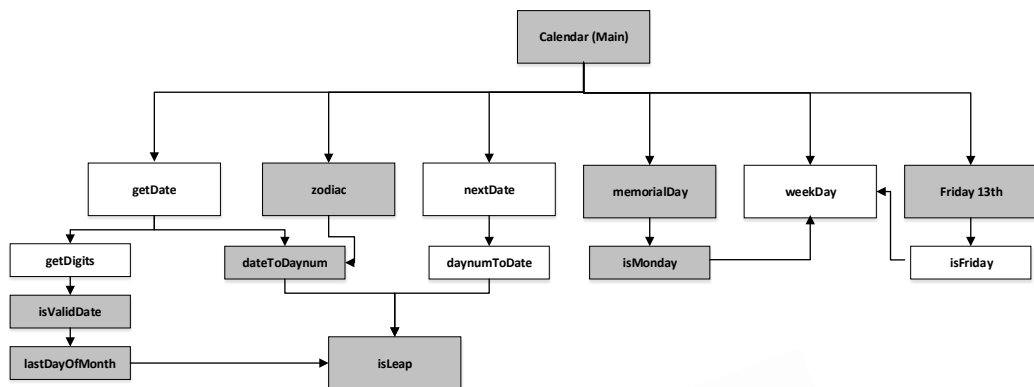
Slika 5.24. Big beng integracija, sve komponente se spajaju odjedanput

Ukoliko se vrši integracija prema grafu poziva, prvo je neophodno napraviti graf poziva. Graf poziva za program u ovom primeru je dat na slici 5.25.



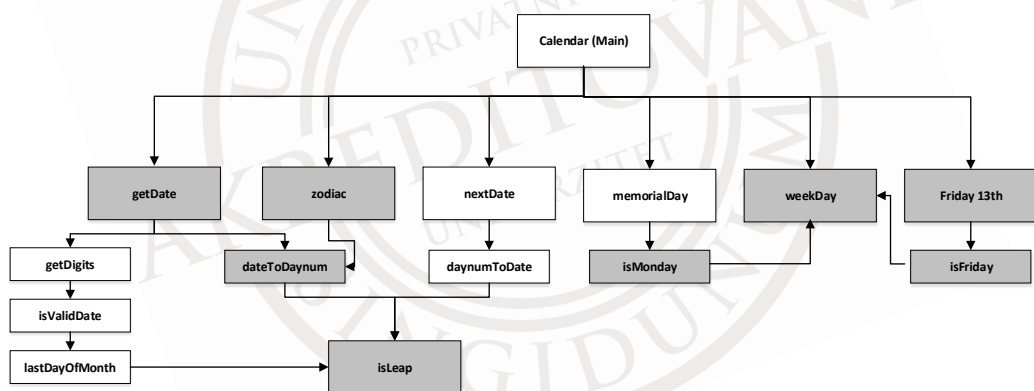
Slika 5.25. Graf poziva za program calendar

Ukoliko se koristi integracija prema parovima, na slici 5.26 prikazan je primer tri para (označeni belom bojom).



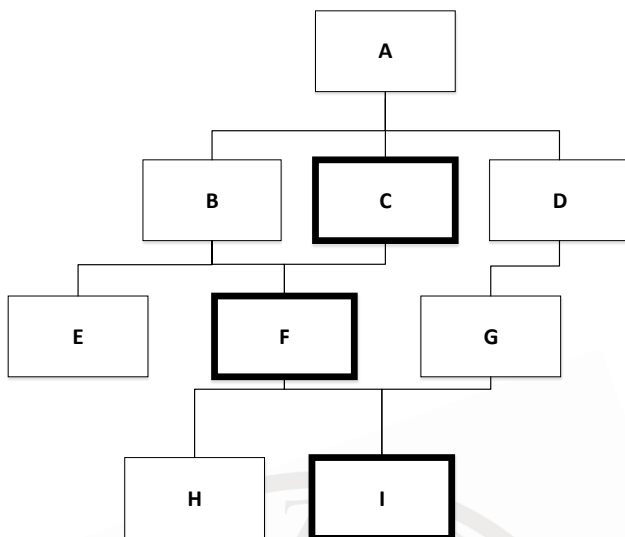
Slika 5.26. Integracija prema parovima

Ukoliko se koristi integracija prema susedstvu, primer tri susedstva je prikazan na slici 5.27. Ciljni čvorovi ovih susedstva su isValidDate, nextDate i memorialDay, respektivno.



Slika 5.27. Integracija prema susedstvu

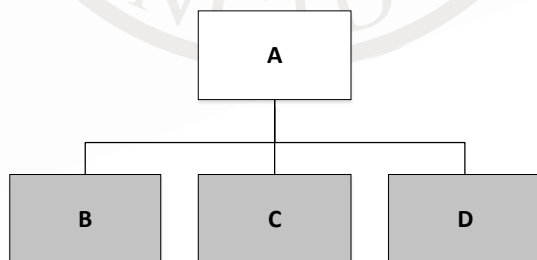
Primer 2: Za softverski sistem sa slike 5.28 napisati redosled integracije modula ukoliko se koristi tehnika od vrha ka dnu. Dati samo prva četiri koraka integracije. Kritični moduli su označeni podebljanim crnim okvirom.



Slika 5.28. Primer arhitekture softverskog sistema

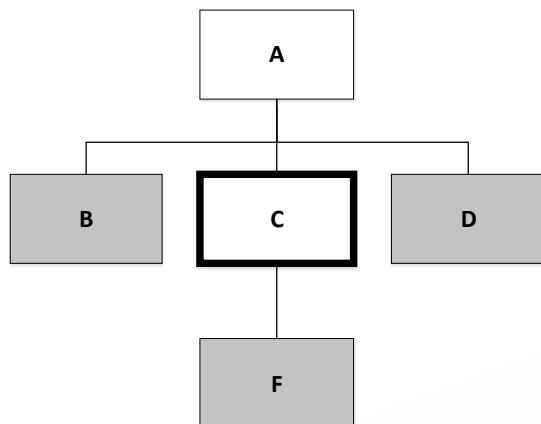
Pod kritičnim modulom se smatra komponenta koja se mora testirati i integrisati što je ranije moguće. To može biti komponenta koja je od velikog značaja za sistem (na primer, više različitih funkcionalnosti zavisi od nje), koja se nalazi visoko u kontrolnoj hijerarhiji, ili je izuzetno kompleksna i samim tim podložna greškama.

Rešenje: Tehnika od vrha ka dnu ima dva moguća pristupa – po širini i po dubini. Ukoliko se radi integracija po širini, integracija se vrši po nivoima. U nastavku primera, stabovi su označeni sivom bojom. U prvom koraku integracije, integriše se glavni modul, a svi direktno podređeni moduli se zamenjuju stabovima, kao što je prikazano na slici 5.29.



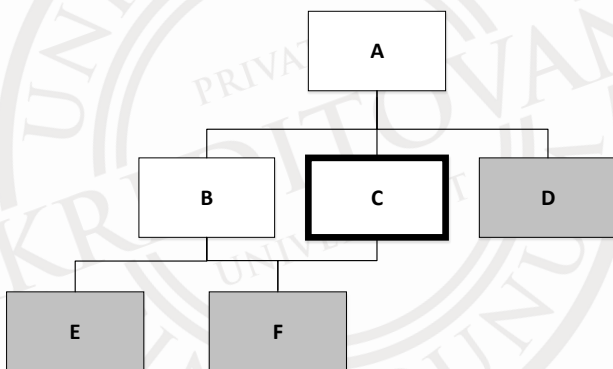
Slika 5.29. Prvi korak integracije po širini

U drugom koraku, pre integracije modula B ili D se dodaje kritični modul C, kao što je prikazano na slici 5.30.

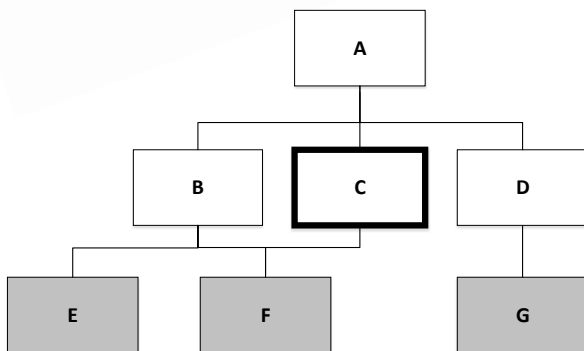


Slika 5.30. Drugi korak integracije po širini

U trećem i četvrtom koraku, dodaju se i komponente B i D, kao što je prikazano na slikama 5.31 i 5.32, respektivno.

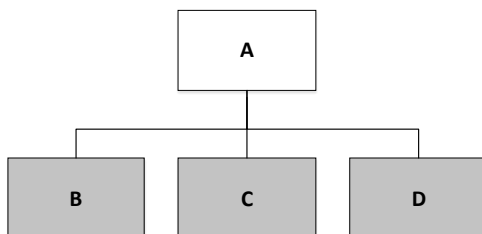


Slika 5.31. Treći korak integracije po širini



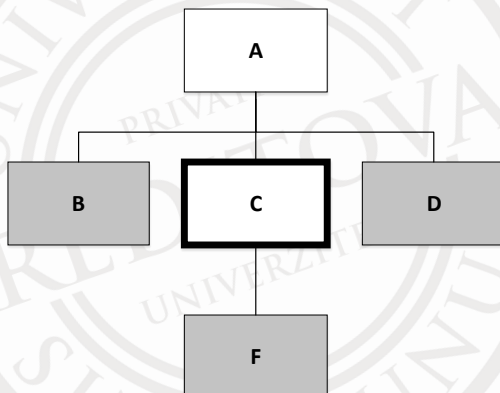
Slika 5.32. Četvrti korak integracije po širini

Ukoliko se primenjuje integracija po dubini, redosled integracije modula u ovom slučaju se obavlja po liniji kritičnih modula. Kao i u slučaju integracije po širini, prvo se integriše glavni modul (slika 5.33).



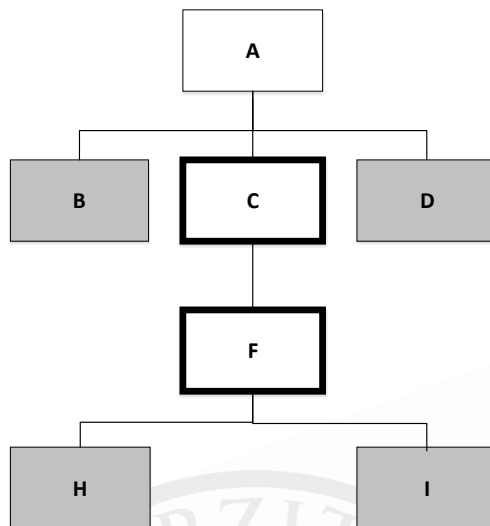
Slika 5.33. Prvi korak integracije po dubini

U drugom koraku, integriše se kritični modul C, kao na slici 5.34.



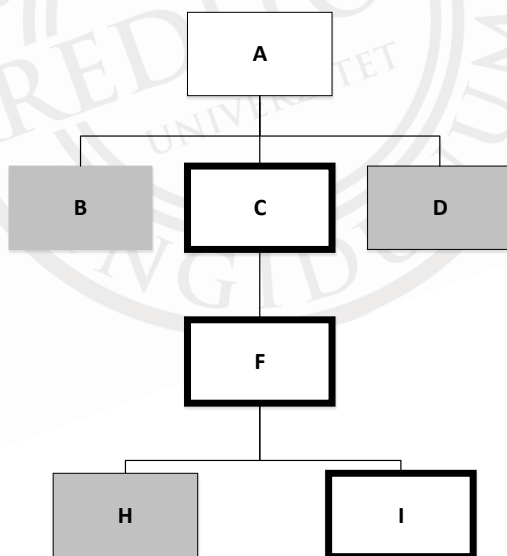
Slika 5.34. Drugi korak integracije po dubini

U trećem koraku se nastavlja put po glavnoj liniji (sa kritičnim modulima), i integriše se sledeći kritični modul F, kao što je prikazano na slici 5.36.



Slika 5.35. Treći korak integracije po dubini

Na kraju, u četvrtom koraku integracije po dubini, od dva moguća modula za integraciju (H i I) bira se kritični modul I, kao što je prikazano na slici 5.36.



Slika 5.36. Četvrti korak integracije po dubini

Nakon toga, integrisao bi se modul H, da se obezbedi ispravan rad komponente F, a integracija po dubini bi mogla da se nastavi integracijom leve, pa desne grane.

Primer 3: Posmatra se klasa koja radi fakturisanje. Postoje dve opcije, faktura se može slati mejlom, ili ukoliko mušterija više voli papirnu verziju, faktura se šalje servisu štampača na štampu. Kod ove klase je dat u InvoiceService.java:

```
public class InvoiceService {

    private PrinterService printerService = null;
    private EmailService emailService = null;

    public InvoiceService(PrinterService printerService, EmailService
emailService) {
        this.printerService = printerService;
        this.emailService = emailService;
    }

    public void handleInvoice(Invoice invoice, Customer customer) {
        if (customer.prefersEmails()) {
            emailService.sendInvoice(invoice, customer.getEmail());
        } else {
            printerService.printInvoice(invoice);
        }
    }

}
```

Potrebno je testirati klasu InvoiceService i integrisati je u već postojeće klase. Problem predstavlja činjenica da sve ostale klase koje su neophodne za rad klase InvoiceService još uvek nisu implementirane.

Rešenje: U ovakvom obliku nije moguće napisati ni najprostiji JUnit test, pošto klase PrinterService, EmailService, Invoice i Customer nisu još uvek napisane. U pitanju su klase od kojih u potpunosti zavisi rad klase InvoiceService, a čiji nedostatak sprečava osnovnu funkcionalnost ovog modula. Kod nije moguće ni kompajlirati, pošto navedene klase ne postoje i na slici 5.37 su podvučene crveno.

```

public class InvoiceService {

    private PrinterService printerService = null;
    private EmailService emailService = null;

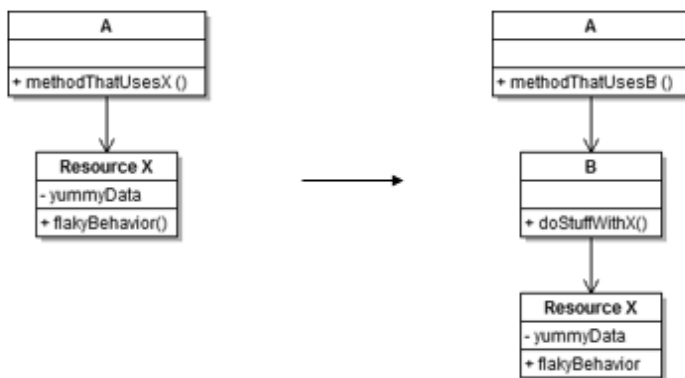
    public InvoiceService(PrinterService printerService, EmailService emailService) {
        this.printerService = printerService;
        this.emailService = emailService;
    }

    public void handleInvoice(Invoice invoice, Customer customer) {
        if (customer.prefersEmails()) {
            emailService.sendInvoice(invoice, customer.getEmail());
        } else {
            printerService.printInvoice(invoice);
        }
    }
}

```

Slika 5.37. Klasa InvoiceService u izvornom obliku

U ovakvim slučajevima se pristupa implementaciji stabova za sve klase koje nedostaju. Kako se kreira stab? Potrebno je da se identifikuje zavisnost od neke druge klase ili resursa. Ukoliko zavisnost nije objekat, već resurs, potrebno je upakovati resurs u objekat ukoliko je moguće, kao što je prikazano na slici 5.38.

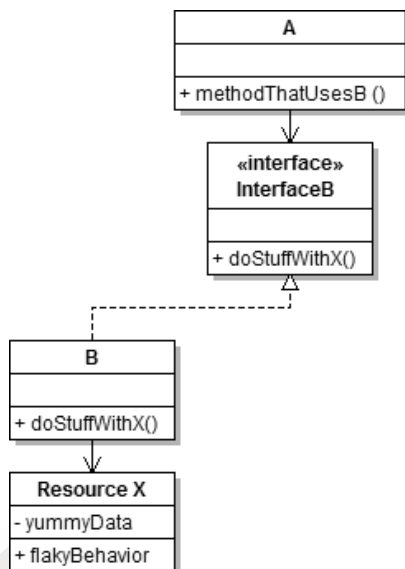


Slika 5.38. Klasa koja zavisi od resursa

U ovom slučaju, klasa A koristi resurs X direktno i zavisi od njega. Potrebno je upakovati ovaj resurs preko objekta klase B, i pristupati mu iz klase A indirektno, preko klase B.

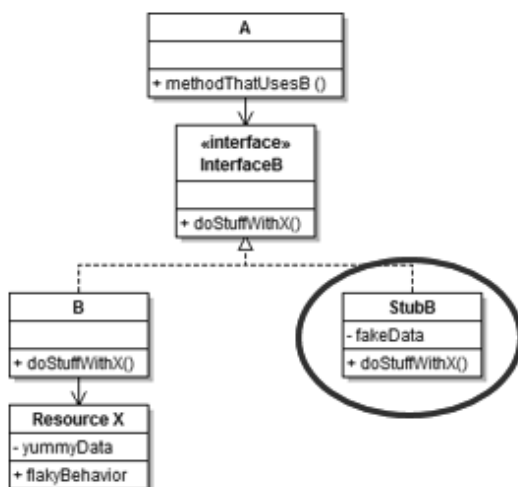
Nakon toga se izvršava izvlačenje osnovne funkcionalnosti objekta u interfejs (slika 5.39) u sledeća dva koraka:

- Kreira se InterfejsB na osnovu B
- U klasi A se kompletan kod zameni da radi sa InterfejsomB, umesto sa B



Slika 5.39. Izvlačenje osnovne funkcionalnosti objekta u interfejs

Sada se kreira stub klasa, koja takođe implementira dati interfejs, ali vraća predefinisane test podatke (slika 5.40). Na ovaj način je zavisnost A od B rešena, i može se lako testirati pomoću staba B.



Slika 5.40. Kreiranje stub klase koja implementira interfejs

Kada je najbolje injektovati stab sa kojim će raditi klasa A? Moguće su tri varijante:

- U konstruktoru
`A objekataA = new A(new StubB());`
- Kroz getter i setter metode
`A objekataA = new A(...);`
`objekataA.setResource(new StubB());`
- Neposredno pre upotrebe, kao parametar u pozivu metode
`objekataA.metodaKojaKoristiB(new StubB());`

U zadatom primeru, za sve klase koje fale se funkcionalnost mora izvući u interfejs. Potrebno je napraviti interfejs za klase `PrinterService`, `EmailService`, `Invoice` i `Customer`, i zatim napraviti stab klase koji implementiraju te interfejse. Svugde gde se u kodu očekuju stvarni objekti nedostajućih klasa, potrebno je staviti da se očekuje interfejs, i proslediti objekte stab klasa.

Za klasu `PrinterService`, sva funkcionalnost se izvlači u interfejs `PrinterServiceInterface`.

```
public interface PrinterServiceInterface {  
    public boolean isPrinterConfigured();  
    public void printInvoice(Invoice invoice);  
    public boolean anInvoiceWasPrinted();  
}
```

Kao što je ranije navedeno, za svaku klasu se kreira stab klasa koja implementira novokreirani interfejs te klase. Za slučaj klase `PrinterService`, potrebno je kreirati klasu `PrinterServiceStub`, koja će implementirati interfejs `PrinterServiceInterface`.

```
public class PrinterServiceStub implements PrinterServiceInterface {  
    boolean anInvoiceWasPrinted = false;  
  
    @Override  
    public boolean isPrinterConfigured() {  
        return true;  
    }  
  
    @Override  
    public void printInvoice(Invoice invoice) {  
        anInvoiceWasPrinted = true;  
    }  
}
```

```

    }

    public boolean anInvoiceWasPrinted() {
        return anInvoiceWasPrinted;
    }
}

```

Svugde u kodu klase InvoiceService gde se očekuje PrinterService, stavlja se PrinterServiceInterface, koji će moći da primi i stub, a kasnije i stvarni modul kada bude implementiran.

```

public class InvoiceService {

    private PrinterServiceInterface printerService = null;
    private EmailService emailService = null;

    public InvoiceService(PrinterServiceInterface printerService,
        EmailService emailService) {
        this.printerService = printerService;
        this.emailService = emailService;
    }

    public void handleInvoice(Invoice invoice, Customer customer) {
        if (customer.prefersEmails()) {
            emailService.sendInvoice(invoice, customer.getEmail());
        } else {
            printerService.printInvoice(invoice);
        }
    }
}

```

Isto je potrebno uraditi i za preostale klase koje nedostaju. Koristi se identičan princip – kreira se interfejs, a zatim stab klasa koja implementira taj interfejs. Svugde u kodu se zatim umesto nedostajućih klasa stavi interfejs, i zatim poziva sa objektima stab klasa.

```

public interface CustomerInterface {

    public void wantsEmail (boolean wantsEmail);

    public boolean prefersEmails();

    public String getEmail();
}

```

```

public class CustomerStub implements CustomerInterface{

    private boolean prefersEmails;

    public CustomerStub (){}

    public void wantsEmail (boolean prefersEmails){
        this.prefersEmails = prefersEmails;
    }

    @Override
    public boolean prefersEmails() {
        return prefersEmails;
    }

    @Override
    public String getEmail() {
        return "email@test.com";
    }

}

```

```

public interface EmailServiceInterface {

    public boolean isEmailConfigured();

    public void sendInvoice(Invoice invoice, String email);

    public boolean anInvoiceWasEmailed();

}

```

```

public class EmailServiceStub implements EmailServiceInterface {

    boolean anInvoiceWasSent = false;

    @Override
    public boolean isEmailConfigured() {
        return true;
    }

    @Override
    public void sendInvoice(Invoice invoice, String email) {
        anInvoiceWasSent = true;
    }

    @Override

```

```

    public boolean anInvoiceWasEmailed() {
        return anInvoiceWasSent;
    }
}

```

Za Invoice je dovoljno napraviti praznu klasu Invoice.

```

public class Invoice {
}

```

Sada je potrebno svugde u kodu klase InvoiceService postaviti da se umesto nedostajućih klasa koristi.

```

public class InvoiceService {
    private PrinterServiceInterface printerService = null;
    private EmailServiceInterface emailService = null;

    public InvoiceService(PrinterServiceInterface printerService,
        EmailServiceInterface emailService) {
        this.printerService = printerService;
        this.emailService = emailService;
    }

    public void handleInvoice(Invoice invoice, CustomerInterface
customer) {
        if (customer.prefersEmails()) {
            emailService.sendInvoice(invoice, customer.getEmail());
        } else {
            printerService.printInvoice(invoice);
        }
    }
}

```

Sada se klasa može uspešno kompajlirati, i može se napisati i JUnit test:

```
import org.junit.Test;
import static org.junit.Assert.*;
import org.junit.Before;

public class InvoiceTest {

    private InvoiceService finalInvoice = null;
    private CustomerInterface customer = null;
    private Invoice invoice = null;

    @Before
    public void beforeEachTest() {
        customer = new CustomerStub();
        finalInvoice = new InvoiceService(new PrinterServiceStub (),
new EmailServiceStub ());
        invoice = new Invoice();
    }

    @Test
    public void normalCustomer() {
        customer.wantsEmail(true);
        finalInvoice.handleInvoice(invoice, customer);
    }

    @Test
    public void customerWithPrintedInvoice() {
        customer.wantsEmail(false);
        finalInvoice.handleInvoice(invoice, customer);
    }

}
```

Najveća prednost izdvajanja funkcionalnosti u interfejs i pisanja staba izvedenog iz tog interfejsa leži u činjenici da je, nakon završetka implementacije stvarne komponente, koja takođe implementira taj interfejs, izuzetno lako zameniti stab stvarnom komponentom. Jedino što treba da se promeni u kodu je da se pri injekciji ne koristi objekat klase stab već stvarne klase. Na primer, pošto i klasa `CustomerService` i klasa `CustomerServiceStub` implementiraju interfejs `CustomerServiceInterface`, na svakom mestu gde se očekuje tip `CustomerServiceInterface` moguće je proslediti i objekat klase `CustomerService`, i objekat klase `CustomerServiceStub` (konkretno u metodi `handleInvoice` u klasi `InvoiceService`).

Pitanja

1. Šta je integraciono testiranje?
2. Koja faza testiranja se vrši pre integracionog testiranja?
3. Objasniti razliku između jediničnog i integracionog testiranja.
4. Zbog čega je neophodno raditi integraciono testiranje ako su komponente već testirane na jediničnom nivou?
5. Koji se problemi tipično pronalaze prilikom integracionog testiranja?
6. Zbog čega postoji potreba za stabovima i drajverima?
7. Koje sve vrste stabova postoje?
8. Koji tipovi integracionog testiranja postoje?
9. Koji problemi mogu nastati za vreme Big Bang integracije?
10. U kom tipu integracionog testiranja su tipično potrebni stabovi?
11. U kom tipu integracionog testiranja su tipično potrebni drajveri?
12. Koje dve tehnike postoje u slučaju integracije od vrha ka dnu?
13. Šta su klasteri u slučaju integracije od dna ka vrhu?
14. Šta su to kritični moduli?
15. Zbog čega se radi izdvajanje funkcionalnosti u interfejs prilikom programiranja stabova
16. Šta je sendvič integracija?
17. Šta je graf poziva i kako se pravi?
18. Koja je razlika između integracije po parovima i integracije po susedstvu?

6. VIŠI NIVOI TESTIRANJA

Viši nivoi testiranja se odnose na posebne vrste testiranja koje se sprovode za vreme razvoja softvera, za vreme ili posle integracionog testiranja, a pre funkcionalnog testiranja. Dve najbitnije tehnike iz ove oblasti su regresiono testiranje, *smoke test* i *sanity test*.

6.1. Regresiono testiranje

Iterativni razvoj se bazira na činjenici da se razvoj softvera deli u iteracije. Za vreme svake iteracije implementira se deo funkcionalnosti sistema, one se integrišu i testiranje se fokusira na tim novim funkcionalnostima koje su dodate. Međutim, u praksi se javlja problem da se prilikom integracije novih komponenti često dešava da komponente koje su implementirane, testirane i integrisane u prethodnim iteracijama prestanu da rade. Razlog zbog čega se to dešava je da tokom trenutne iteracije može doći do promene nekog dela softverskog sistema koji ima negativan efekat na ranije implementirane i testirane komponente. U opštem slučaju, komponente softverskog sistema nisu nezavisne, već deo celine, i kao takve mogu deliti različite podatke, servise i slično.

Na prvi pogled logično je da fokus testiranja bude na novim, upravo implementiranim funkcionalnostima. Ta logika je ispravna, jer su nove funkcionalnosti u svakoj iteraciji svakako najvažnija stavka. Međutim, ukoliko se testiranje bazira isključivo na ovoj logici, testovi će pokriti samo nove funkcionalnosti, dok komponente koje su implementirane u prethodnim iteracijama mogu prestati da rade ispravno. Najgore od svega se može desiti da niko ne primeti greške u radu tih komponenti, jer ih zapravo niko nije ni proveravao pri testiranju.

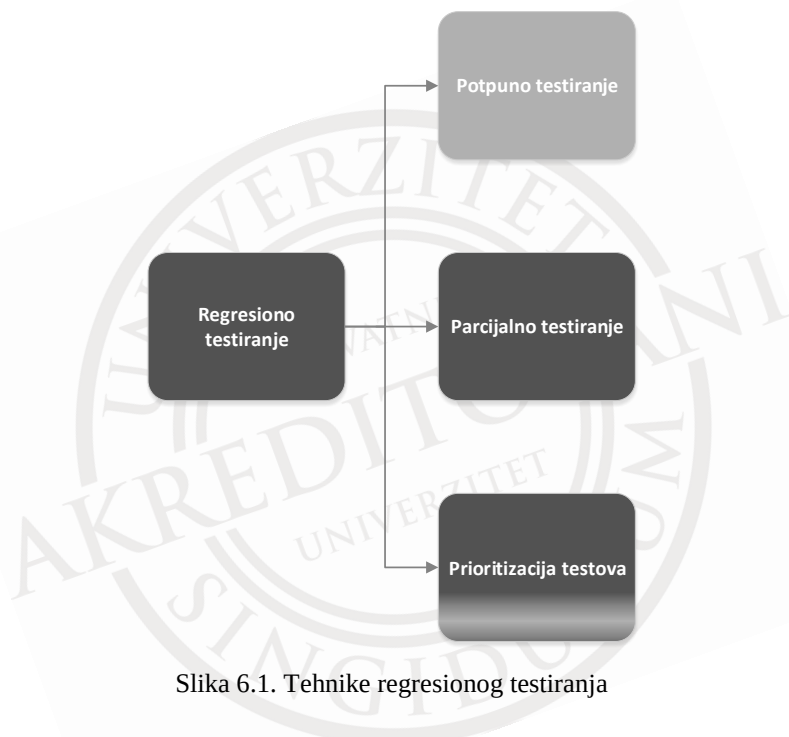
Regresiono testiranje se definiše kao tip testiranja softvera koji za cilj ima potvrdu da nove promene u kodu programa nisu uvele nove greške u radu prethodno testiranih komponenti i imale loš uticaj na već postojeće funkcionalnosti sistema. Prilikom regresionog testiranja se ponavljaju testovi koji su već izvršeni u prethodnim iteracijama, kako bi se osiguralo da postojeće funkcionalnosti i dalje rade ispravno nakon implementacije novih komponenti. Na taj način se osigurava da nove promene u kodu nisu imale neočekivane bočne efekte na već postojeće komponente, i da stari kod i dalje radi nakon uvođenja novog koda.

U opštem slučaju, bilo koja promena u kodu koji koristi neka funkcionalnost može da utiče na prethodno implementirane funkcionalnosti. Zbog toga, regresiono testiranje je potrebno u svakom od navedenih slučajeva:

- Novi zahtev u specifikaciji softvera, i odgovarajuća promena koda prema tom zahtevu.
- Dodavanje nove funkcionalnosti
- Uklanjanje postojeće funkcionalnosti
- Ispravka bilo kog defekta
- Izvršena optimizacija radi poboljšanja performansi sistema

6.1.1. Tehnike regresionog testiranja

Regresiono testiranje spada u aktivnost održavanja softvera, koja uključuje poboljšanja funkcionalnosti i sistema kao celine, ispravke grešaka, optimizaciju kao i uklanjanje nekih postojećih funkcionalnosti sistema. Svaka od ovih modifikacija može izazvati nepravilan rad softverskog sistema, pa je regresiono testiranje obavezna aktivnost. Regresiono testiranje se može izvršiti na nekoliko načina, kao što je prikazano na slici 6.1.



Slika 6.1. Tehnike regresionog testiranja

Potpuno regresiono testiranje podrazumeva da se uz nove testove koji testiraju novu funkcionalnost izvrše i svi postojeći testovi iz skupa testova iz prethodnih iteracija. Ovakav pristup je obično vrlo skup po pitanju potrebnog vremena i resursa, pa je za veće softverske sisteme najčešće neophodno da se uradi smanjivanje broja testova.

Parcijalno regresiono testiranje se svodi na odabir i izvršavanje dela testova iz skupa testova, umesto izvršavanja svih testova iz skupa. Odabir testova se fokusira na testiranju prethodno implementiranih funkcionalnosti koje imaju bilo kakvih dodirnih tačaka sa funkcionalnostima koje se uvode u trenutnoj iteraciji. Uz novi kod, testiraju se sve funkcionalnosti na koje bi nove promene mogle da imaju uticaj.

U slučaju prioritizacije testova, prioritet se određuje na osnovu poslovnog uticaja (engl. *business impact*), kritičnih i često korišćenih funkcionalnosti. Odabir testova prema prioritetu značajnu redukuje veličinu skupa testova za regresiono testiranje.

6.1.2. Odabir testova za regresiono testiranje

U softverskoj industriji je široko poznata činjenica da značajan deo defekata koji procure u produkciju i koje prijave krajnji korisnici sistema potiče od ispravke defekata u poslednji trenutak i nedovoljnog ili neadekvatnog regresionog testiranja pre isporuke sistema. Razvoj softvera je aktivnost koja je vrlo često stresna, sa izuzetno kratkim vremenskim rokovima i ograničenim budžetom, gde ponekad nema dovoljno vremena za obimno testiranje, a ispravka defekata u poslednji trenutak je nažalost svakodnevnica. Kako bi se napravio efikasan skup regresionih testova, u praksi je potrebno pridržavati se određenih pravila prilikom odabira testova. U skup regresionih testova se uključuju:

- Testovi koji su često otkrivali defekte u prethodnim iteracijama
- Funkcionalnosti sistema koje su najviše vidljive krajnjem korisniku, i koje će biti najviše u potrebi
- Testovi koji verifikuju osnovnu funkcionalnost programa
- Testovi funkcionalnosti koje su imale najskorije izmene
- Svi testovi iz skupa integracionih testova
- Svi kompleksni testovi
- Testovi koji proveravaju granične vrednosti

6.1.3. Automatizacija regresionog testiranja

Ukoliko je softver koji se razvija podložan čestim izmenama, troškovi testiranja će značajno eskalirati. Manuelno testiranje u ovom slučaju značajno produžava troškove i vreme izvršavanja testova. Još jedna loša strana manuelnog regresionog testiranja je činjenica da se stalno ponavljaju isti testovi, što može biti veoma naporno i dosadno za testera. Zbog toga, može se desiti da tester nema dovoljno koncentracije ili čak da otalja neke testove i propusti da otkrije neke od defekata. Regresiono testiranje je najbolji kandidat za automatizaciju, i uvek ukoliko je ikako moguće se teži primeni nekog od alata za automatizaciju testiranja.

Obim automatizacije zavisi od ukupnog broja testova koji su ponovo primenljivi (engl. *reusable*) u višestrukim iteracijama i ciklusima regresionog testiranja. Jedan primer alata koji se koristi za automatizaciju veb aplikacija je Selenium, koji se često primenjuje u regresionom testiranju.

Postoje i određena ograničenja automatizacije regresionog testiranja. U praksi nije dovoljno da se pokrene program i postavi na autopilot, već je neophodno svesno nadgledanje i određeni ulazni podaci od strane testera kako bi se osiguralo da svi automatski testovi zaista uhvate sve bagove koji su prisutni u kodu. Ukoliko se svakodnevno pokreće isti skup automatskih testova, proces testiranja može postati statičan. Programeri mogu pronaći način da fiksiran skup testova prođe uspešno, što kao posledicu ima nemogućnost otkrivanja novih grešaka. Kao analogija se može posmatrati prolazak kroz minsko polje. Kako bi se uspešno prošlo kroz neprijateljsku teritoriju, nije

potrebno raščistiti kompletno minsko polje. Dovoljno je raščistiti jednu putanju, označiti granice i čuvati tu putanju. Sve dok vojnici čuvaju ovu putanju (vojnici su u ovoj analogiji regresioni testovi), malo je verovatno da će bilo ko postaviti novu minu na tu putanju. Međutim, ova neprekidno raščišćena putanja ne kaže ništa o drugim minama koje postoje van nje, bilo da su nove ili stare, na koje neko može da nagazi (u ovoj analogiji to su krajnji korisnici).

Na isti način, ukoliko su regresioni testovi previše automatizovani i ne ažuriraju se vremenom, cela poenta automatizacije može imati negativan povratni efekat. Hodanje jednom raščišćenom putanjom je naravno biranje puta sa najmanje otpora, što je lakše nego raščišćavanje kompletne aplikacije kontinualnom adaptacijom skupa regresionih testova.

6.1.4. Razlika između regresionog testiranja i retesta

U praksi često dolazi do mešanja ova dva pojma. Po definiciji, regresiono testiranje je provera da li je izmena koda imala negativan uticaj na postojeće funkcionalnosti sistema. Sa druge strane, retest je tip testiranja koji se izvršava sa ciljem da se nakon ispravke defekta ponovo izvrši test koji je otkrio taj defekt, kako bi se verifikovala njegova ispravka.

Druga razlika je da se regresiono testiranje vrši nakon svake promene koda, dok se retest izvršava prema listi ispravljenih defekata. U opštem slučaju, kada nema dovoljno vremena za detaljno testiranje, prioritet retesta je veći u odnosu na regresiono testiranje.

6.1.5. Prednosti i mane regresionog testiranja

Prednosti regresionog testiranja su očigledne, i mogu se sumirati u sledećim stavkama:

- Osigurava se da bilo koja promena poput ispravke nekog defekta ili dodavanja / brisanja neke funkcionalnosti nije uticala na postojeću funkcionalnost aplikacije
- Verifikuje se da greške iz ranijih iteracija nisu ponovo uvedene
- U velikom broju slučajeva se može automatizovati u nekoj meri
- U opštem slučaju poboljšava se ukupan kvalitet proizvoda

Kao mane se mogu izdvojiti:

- Ukoliko se regresiono testiranje izvršava manuelno, može biti veoma skupo po pitanju uloženog vremena, kao i dosadno i naporno
- Svaka, i najmanja izmena koda zahteva izvršavanje regresionog testiranja, pošto u teoriji može imati negativni uticaj na druge komponente.
- Ukoliko se skup regresionih testova ne ažurira vremenom, proces postaje statički i može prevideti nove defekte.

6.2. Smoke test

Smoke test je oblik testiranja softvera koji proverava da li osnovne funkcionalnosti aplikacije rade na adekvatan način. Testiranje nije detaljno, niti se ulazi u dubinu funkcionalnosti, već se izvršava veoma mali broj testova kako bi se prosto proverilo da li osnovne funkcije programa uopšte rade. Ova tehnika se koristi kao osnovna provera da li je aplikacija dovoljno stabilna i spremna da se nastavi sa detaljnim testiranjem.

Sam termin *smoke test* (engl. testiranje da li ima dima) vuče svoje korene iz testiranja hardverskih komponenti. Prilikom prvog uključivanja hardvera, prvo se proverava vizuelno da li negde ima dima ili varnica među komponentama (princip – gde ima dima ima i vatre). Tako se prosto utvrđuje da li hardver ima neke kritične mane u spojevima, pinovima i slično.

Ista logika se primenjuje i u softveru, gde smoke test vrlo rano može da pokaže da postoje neki katastrofalni problemi u softveru (engl. *showstopper*). Ukoliko postoje, nema nikakve svrhe davati build na detaljno testiranje. Ovo testiranje je najčešće u domenu programera, koji pre davanja build-a QA sektoru na testiranje proveravaju da li je aplikacija dovoljno stabilna. Testiranje može izvršiti i QA sektor prilikom prijema novog build-a, kako bi proverili brzo osnovne funkcionalnosti pre početka detaljnog testiranja. Pošto smoke test nije detaljno testiranje, obično se radi samo testiranje *happy path* (engl. srećna putanja), odnosno prolazak kroz osnovnu funkcionalnost i provera osnovnog korisničkog scenarija, sa validnim ulaznim podacima.

Kao primer može da se posmatra aplikacija u obliku socijalne mreže, koja ima 15 modula. Od tih 15 modula, 4 modula su kritične komponente, i to:

- Login stranica
- Dodavanje detalja korisnika
- Ažuriranje detalja korisnika
- Brisanje korisnika

U okviru smoke testa, testira se login stranica sa validnim ulaznim podacima (ispravno korisničko ime i šifra). Nakon uspešnog logovanja na sistem, testira se dodavanje, ažuriranje i brisanje korisnika. Ukoliko sve 4 kritične komponente rade ispravno u kombinaciji sa validnim ulaznim podacima, smatra se da je build dovoljno stabilan kako bi se nastavilo sa testiranjem. Ukoliko se uoče problemi sa bilo kojom od ove 4 komponente u kombinaciji sa validnim ulaznim podacima, nema ni svrhe počinjati detaljno testiranje i gubiti vreme, već se build vraća programerima na doradu.

Prednosti smoke testa:

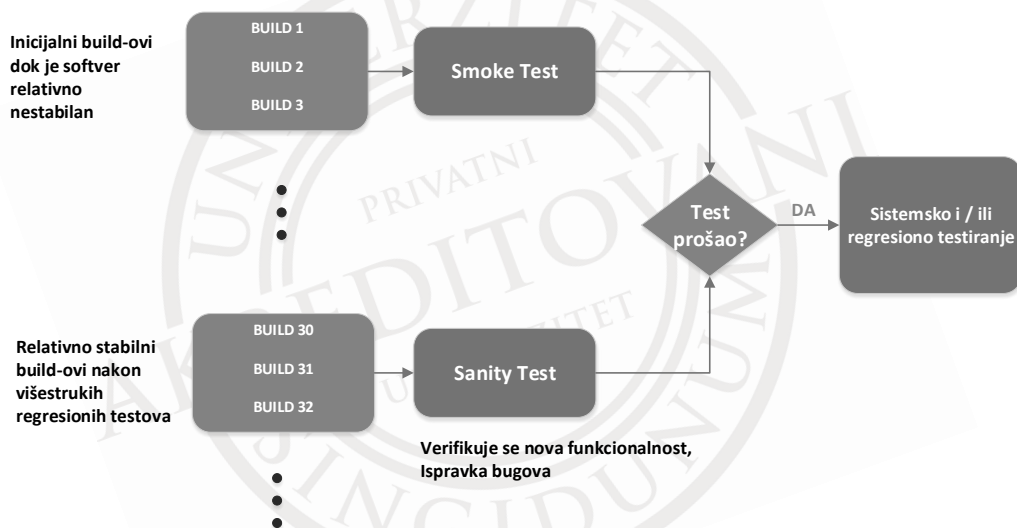
- Kritični defekti se mogu uočiti u veoma ranoj fazi testiranja
- Utvrđuje se da ispravljeni defekti iz prethodne iteracije nisu uticali na glavne funkcionalnosti sistema
- Mali broj testova
- Kratko vreme testiranja

Kao mane se mogu izdvojiti sledeće činjenice:

- Nije detaljno testiranje
- Pošto se radi mali broj testova, nije moguće otkriti sve kritične bagove
- Ne radi se sa negativnim scenarijima i sa invalidnim podacima

6.3. Sanity test

Sanity test (engl. test zdravog razuma) je vrlo sličan smoke testu. U pitanju su dva termina koja se jako često mešaju u testiranju softvera. Veliki broj testera nije siguran u razlike između ova dva tipa testiranja, koje nisu velike, ali ipak postoje. Osnovne razlike se mogu prikazati u obliku dijagrama sa slike 6.2.



Slika 6.2. Razlika između sanity i smoke testiranja

Smoke test je, kao što je opisano u prethodnom poglavlju, tehnika testiranja koja se izvršava nakon build-a softvera kako bi se verifikovao uspavan rad osnovnih kritičnih funkcionalnosti. Izvršava se pre bilo kog detaljnog testiranja, kako QA tim ne bi gubio vreme u slučaju da aplikacija ima kritične probleme. Tipični primeri smoke testa su verifikacija da li se aplikacija uspešno pokreće, da li korisnički interfejs ima odgovarajući odziv i slično. Vrlo često ovaj tip testiranja rade programeri.

Sa druge strane, sanity test se izvršava nad relativno stabilnom aplikacijom. Ovaj tip testiranja izvršava QA tim nakon prijema softverskog build-a u kome postoje manje izmene u kodu ili je dodata nova funkcionalnost, sa ciljem da utvrdi da li predložena funkcionalnost okvirno radi kako treba i da osnovne funkcionalnosti aplikacije nisu ugrožene. Ukoliko sanity test ne prođe, build se odbija i ne započinje se detaljno

testiranje, kako bi se uštedelo vreme i smanjili troškovi koje detaljno testiranje sa sobom nosi. Cilj nije da se nova funkcionalnost detaljno istestira, već samo da se utvrdi da li je programer bio pri “zdravom razumu” dok je pisao program. Odatle i potiče ime sanity. Na primer, ukoliko se razvija naučni kalkulator, i za unetu formulu $2 + 2$ vraća vrednost 5, nema svrhe testirati napredne funkcije poput $\sin(30) + \cos(45)$.

Oba tipa testiranja služe da se utvrdi da li je aplikacija dovoljno dobra za detaljno testiranje, odnosno da se uštedi vreme ukoliko postoje kritične greške. Ukoliko je aplikacija puna grešaka, build se odbija i vraća programerima na doradu. Vrlo često se ova dva tipa testiranja obavljaju zajedno nad istim softverskim build-om. Prvo se izvršava smoke test, nakon čega se nastavlja sa sanity testiranjem. Dalje, sanity testovi se često kombinuju i izvršavaju zajedno sa smoke testovima u industriji kako bi se ubrzalo izvršavanje testova. Zbog toga i dolazi često do mešanja ova dva pojma.

Za oba tipa testiranja važi da se mogu izvršiti ili manuelno, ili upotrebom nekog alata za automatizaciju. Kada se koristi alat za automatizaciju, najčešća praksa je da se testovi iniciraju od strane istog procesa koji pravi sam build.



Pitanja

1. Šta je regresiono testiranje?
2. U kojim slučajevima se obavezno radi regresiono testiranje?
3. Koja je razlika između regresionog testiranja i retesta?
4. Koje tehnike regresionog testiranja postoje?
5. Kako se biraju testovi za regresiono testiranje?
6. Da li se regresiono testiranje može automatizovati i zašto?
7. Koje su loše strane automatizacije regresionog testiranja?
8. Navedi prednosti i mane regresionog testiranja.
9. Šta je smoke test?
10. Šta je sanity testiranje?
11. Objasniti razliku između smoke i sanity testiranja.
12. Ko izvršava sanity testiranje?
13. Koje su prednosti smoke testa?
14. Koje su mane smoke testa?

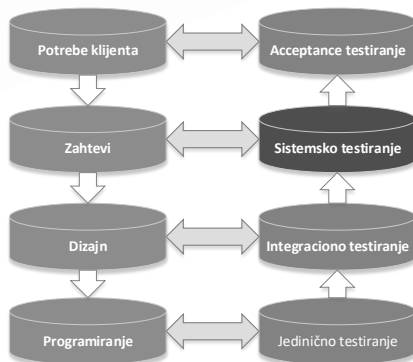
7. SISTEMSKO TESTIRANJE

Sistemsko testiranje se bazira na testiranju sistema kao celine, nakon uspešno završene kompletne integracije. Hronološki sistemsko testiranje sledi nakon integracionog testiranja, a pre *acceptance* testiranja, kao što je prikazano na slici 7.1.



Slika 7.1. Hronološki redosled faza testiranja

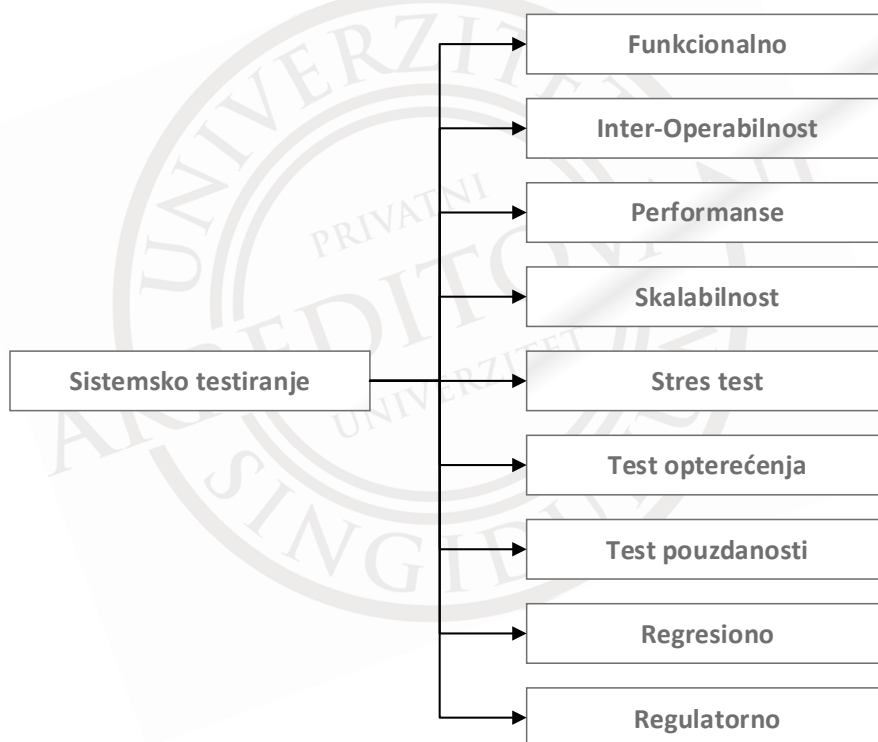
Svrha sistemskog testiranja je verifikacija da sistem kao celina u potpunosti ispunjava zahteve definisane u specifikaciji zahteva sistema (slika 7.2). Sistemsko testiranje spada u kategoriju testiranja metodama crne kutije, pošto se testovi definišu na osnovu specifikacije zahteva sistema. Za razliku od jediničnog testiranja koje najčešće obavljaju programeri i integracionog testiranja koje spada u domen testera, sistemsko testiranje najčešće sprovode najiskusniji testeri – specijalisti. Ponekad se za ovaj zadatak angažuje nezavisni tim testera, kako bi se obezbedila potpuna objektivnost pri sistemskom testiranju.



Slika 7.2. Fokus testiranja u svakoj fazi

Sistemsko testiranje se fokusira na kompletno integrisane aplikacije, uključujući tu i eksterne periferije i druge softverske sisteme, kako bi se proverilo da li sve komponente rade ispravno kao celina zajedno sa aplikacijom (engl. *end-to-end*, odnosno testiranje sa kraja na kraj). To u praksi znači da se aplikacija testira u realnom scenariju upotrebe, sa uspostavljenom komunikacijom sa bazom podataka, mrežom, svim hardverom i drugim softverskim sistemima. Testiranje se obavlja iz perspektive korisnika, uz detaljnu proveru svakog ulaznog podatka i odgovarajućeg izlaza sistema.

Sistemsko testiranje može biti i drugačijeg tipa od već opisanog funkcionalnog. Potrebno je proveriti i nefunkcionalne zahteve, među kojima su performanse sistema, interoperabilnost sa drugim sistemima, zatim skalabilnost, pouzdanost, test opterećenja i slično. Postoji preko 50 različitih podvrsta sistemskog testiranja, od kojih su najznačajnije prikazane na slici 7.3, i ukratko opisane u nastavku ovog poglavlja.



Slika 7.3. Različiti tipovi funkcionalnog testiranja

Pod sistemskim testiranjem se može podvesti i testiranje prihvatanja sistema od strane klijenta (engl. *User acceptance testing*, ili kraće *acceptance testing*). To je testiranje koje sprovodi klijent koji je naručio softverski sistem, kako bi verifikovao da je sve što je zahtevano u specifikaciji zaista i implementirano na odgovarajući način.

7.1. Testiranje performansi sistema

Funkcionalnost koju pruža softverski sistem nije jedina briga u testiranju. Performanse poput vremena odziva, pouzdanosti, korišćenja resursa i skalabilnosti su takođe od velikog značaja. Testiranje performansi je tip testiranja koji verifikuje da se softverski sistem ponaša adekvatno pod očekivanim opterećenjem. Cilj testiranja performansi sistema nije da se pronađu novi defekti, već da se uoče i eliminišu potencijalna uska grla koja utiču na performanse sistema.

Prilikom testiranja performansi sistema, pažnja se obraća na sledeće aspekte:

- Brzina – proverava se brzina odziva aplikacije
- Skalabilnost – određuje se maksimalno korisničko opterećenje koje softver može da podrži
- Stabilnost – proverava se da li je aplikacija stabilna pod različitim opterećenjem.

Testiranje performansi se izvršava sa još jednim ciljem – da se svim zainteresovanim stranama (engl. *stakeholders*) pruži informacija o performansama aplikacije po pitanju brzine, stabilnosti i skalabilnosti, na osnovu kojih se mogu doneti nove odluke o daljem razvoju aplikacije i potrebnim izmenama. Najvažnije od svega, testiranje performansi može da otkrije greške koje funkcionalno testiranje ne može da otkrije, i omogućava da se aplikacija popravi i poboljša pre nego što izađe na tržište. Bez testiranja performansi, nakon izlaska na tržište softver se često susreće sa problemima poput sporog rada kada više korisnika simultano koristi sistem, nekonzistentnosti prilikom pristupa sistemu i loše upotrebljivosti sistema.

Aplikacije sa lošim performansama (usled loše izvedenog testiranja performansi ili potpunog izostanka istog) koje se pošalju na tržište će skoro sigurno steći lošu reputaciju i neće dostići očekivan nivo prodaje. Sa druge strane, kritični softverski sistemi poput softvera medicinske opreme ili softvera u avionima se moraju dodatno testirati kako bi se osiguralo da mogu da rade duži vremenski period bez ikakvih odstupanja u radu.

Pod testiranjem performansi se može grupisati više različitih tipova testiranja, od kojih su najbitniji:

- Testiranje opterećenja (engl. *load test*)
- Stres test (engl. *stress test*)
- Test izdržljivosti (engl. *endurance test*)
- Testiranje skalabilnosti (engl. *scalability testing*).

7.1.1. Najčešći problemi

Najčešći problemi koje otkriva testiranje performansi, očekivano, jesu brzina rada aplikacije, vreme odziva, vreme učitavanja aplikacije i loša skalabilnost. Brzina je jedan od najznačajnijih atributa aplikacije. Spora aplikacija skoro sigurno dovodi do gubitka potencijalnih korisnika. Testiranje performansi osigurava da se aplikacija izvršava dovoljno brzo kako bi privukla pažnju i interesovanje korisnika. Sledeća lista prikazuje najčešće probleme po pitanju performansi (može se uočiti da je brzina zajednički faktor za većinu):

- Predugo vreme učitavanja aplikacije – posmatra se vreme koje je potrebno za pokretanje aplikacije. Potrebno je da bude što je moguće manje. Iako je za neke kompleksne aplikacije potrebno i više od minuta za učitavanje, uopšteno gledano trebalo bi da se ovo vreme drži u okviru od nekoliko sekundi (naravno, ukoliko je moguće).
- Loše vreme odziva – posmatra se vreme koje protekne od trenutka kada korisnik izvrši unos u aplikaciju, do trenutka kada aplikacija vrati odgovor na taj korisnički zahtev. Uopšteno gledano, ovo vreme bi trebalo da bude jako kratko. Ukoliko korisnik predugo čeka na odgovor aplikacije, može vrlo brzo izgubiti interesovanje. Ovo vreme generalno ne bi trebalo da traje duže od nekoliko sekundi.
- Loša skalabilnost – aplikacija ne može da podrži očekivani broj korisnika, ili ne može da podrži dovoljno širok opseg korisnika. Test opterećenja se koristi kako bi se zagarantovalo da aplikacija zaista može da podrži predviđeni broj korisnika.
- Uska grla – ograničenja sistema koja degradiraju opšte performanse celog softverskog sistema. Ponekad su uzrok greške u programiranju koje smanjuju propusnu moć sistema pod određenim opterećenjem. Uzrok može biti i hardverske prirode, usled nedovoljne procesorske moći, nedostatka memorije, loše mrežne infrastrukture, prostora na diskovima i slično.

7.1.2. Proces testiranja performansi

Metodologija testiranja performansi može vrlo često da varira, ali ciljevi testiranja ostaju isti. Na osnovu testiranja performansi, može se pokazati da aplikacija ispunjava predefinisane zahteve po pitanju performansi, ili može identifikovati komponente sistema koji degradiraju performanse. Takođe, može poslužiti za poređenje dva softverska sistema. U najvećem broju slučajeva, metodologija izgleda kako je prikazano na slici 7.4.



Slika 7.4. Metodologija procesa testiranja performansi

Prvo je neophodno identifikovati test okruženje, kao i produkciono okruženje. Potrebno je poznavati detalje hardvera, softvera i mrežne konfiguracije pre početka testiranja. Na taj način se mogu kreirati efikasni testovi. U poznavanje test okruženja, potrebno je precizno definisati kriterijum prihvatljivosti (engl. *acceptance criteria*) za performanse, uključujući poznata ograničenja za propusnu moć, vreme odziva i alokaciju resursa. Često se dobar način za definisanje kriterijuma prihvatljivosti svodi na pronalaženje slične uspešne aplikacije koja se kasnije koristi kao referenca za poređenje.

Prilikom planiranja i dizajna testova se mora odrediti koliko će način upotrebe aplikacije varirati među krajnjim korisnicima, i identifikovati svi ključni scenariji i načini upotrebe. Testovi treba da simuliraju ove različite korisnike i različite scenarije. Neposredno pre početka testiranja potrebno je pripremiti test okruženje, računajući tu i sve alate koji se možda koriste.

Nakon toga se testovi kreiraju i izvršavaju, uz detaljno praćenje rezultata. Kada su rezultati raspoloživi, pristupa se detaljnoj analizi i uočavanju problema. Uočeni problemi se zatim koriguju, izvršava se dodatno optimizovanje sistema i test se pokreće ponovo, kako bi se uočilo eventualno poboljšanje ili pogoršanje performansi. Prilikom finog podešavanja sistema, uočena poboljšanja će nakon svake ponovne iteracije biti sve manja. Obično se prestaje sa finim podešavanjem u onom trenutku kada procesor postane usko grlo sistema, kada se eventualno može razmatrati pojačanje procesorske moći ukoliko je potrebno još više poboljšati performanse.

7.1.3. Praćenje parametara

Postoji veliki broj parametara koji se mogu pratiti za vreme testiranja performansi sistema. Lista nekih najčešće korišćenih parametara je data u nastavku:

- Korišćenje procesora – vreme koje procesor provede prilikom izvršavanja niti koje nisu *idle*.
- Upotreba memorije – količina fizičke memorije koja je dostupna procesorima
- Upotreba diskova – vreme za koje je disk zauzet zahtevima za čitanje/upis.
- Korišćenje mreže (engl. *bandwidth*) – broj bitova po sekundi koje koristi mrežni interfejs.
- Privatni bajtovi – broj bajtova koje proces alocira i koji se ne mogu deliti sa drugim procesima. Često se koristi za indicaciju curenja memorije.

- Broj procesorskih prekida po sekundi – prosečan broj hardverskih prekida koje procesor prima i obrađuje svake sekunde.
- Dužina reda za čekanje na mrežnom izlazu – broj paketa koji čekaju u redu paketa na mrežnom izlazu. Sve više od dva obično znači kašnjenje i usko grlo koje se mora ispraviti.
- Vreme odziva – vreme od trenutka kada korisnik pošalje zahtev do trenutka kada se primi odgovor sistema.
- Propusnost sistema (engl. *throughput*) – broj korisničkih zahteva po sekundi.
- Maksimalni broj aktivnih sesija – maksimalni broj sesija koje mogu biti aktivne u istom trenutku.
- Broj pogodaka po sekundi (engl. *hits per second*) – broj pogodaka na veb server u svakoj sekundi testa opterećenja.
- Broj katanaca na bazi – zaključavanje tabela se mora pratiti i pažljivo podesiti kako performanse ne bi bile loše.
- Sakupljanje đubreta (engl. *garbage collection*) – sistemu se mora vratiti memorija koja više nije potrebna aplikaciji. Sakupljanje đubreta se mora pratiti kako bi se poboljšala efikasnost.
- Procenat pogodaka (engl. *hit ratios*) – broj SQL naredbi koje se obrade putem keširanih podataka umesto skupih ulazno/izlaznih operacija. Ova stavka se mora detaljno pratiti pošto je često uzrok uskog grla.

7.2. Testiranje opterećenja sistema

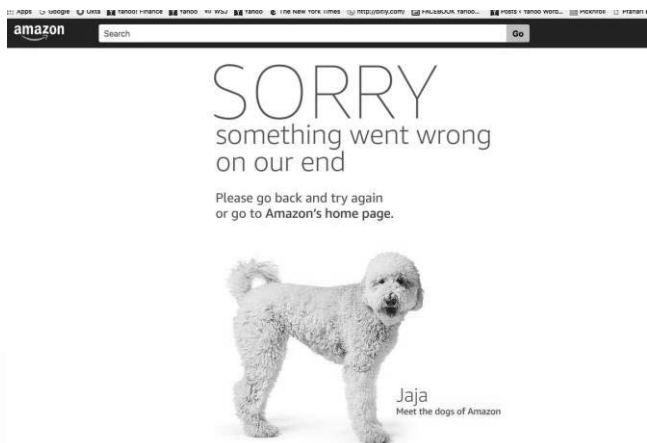
Testiranje opterećenja je vrsta testiranja performansi, koja određuje performanse sistema u okviru realnih uslova upotrebe. Spada u testiranje nefunkcionalnih zahteva. Najprostije rečeno, sistem se dodavanjem dodatnih zahteva stavlja pod opterećenje i meri se vreme odgovora. Određuje se ponašanje sistema kako pod normalnim opterećenjem, tako i pod očekivanim najvećim opterećenjem. Identifikuje se maksimalni operativni kapacitet, postojanje uskih grla i određuje koja komponenta izaziva degradaciju performansi. Kada se opterećenje podigne iznad razumnog nivoa, test opterećenja postaje stres test sistema. Najčešće se primenjuje u slučaju klijent/server veb aplikacija.

Formalno gledano, test opterećenja određuje:

- Maksimalni operativni kapacitet aplikacije
- Da li je trenutna infrastruktura dovoljna za izvršavanje aplikacije
- Održivost aplikacije u slučaju vršnog korisničkog opterećenja (engl. *peak load*)
- Broj konkurentnih korisnika koje aplikacija može da izdrži, kao i skalabilnost koja je potrebna kako bi se omogućio veći broj korisnika.

Koliko je testiranje opterećenja bitno, najlakše se može videti iz primera nekih veoma popularnih sajtova koji su u pojedinim situacijama bili oboreni duži vremenski period zbog ogromnog obima saobraćaja. Najčešći razlog zbog kojeg su veb sajtovi bili oboreni su različite promocije koje su nudili, a nisu mogli da izdrže neočekivano visok

broj korisničkih zahteva. Najbolji primer je Amazon, koji svake godine organizuje *Prime Day*, globalni dan za kupovinu za svoje *prime* korisnike, gde se unutar 36 sati daje ogroman broj popusta. Međutim, ovaj događaj je pošao naopako u julu 2018. godine, kada je zbog prevelikog broja zahteva servis bio nedostupan više sati u pojedinim delovima Sjedinjenih Američkih Država (slika 7.5).



Slika 7.5. Poruka o nedostupnom servisu za vreme Amazon Prime Day 2018

Pogled na mapu sa slike 7.6 na kojoj su prikazane lokacije gde je servis bio nedostupan pokazuje da su u pitanju najbogatije oblasti, odakle je očekivano najviše kupovina. Pošto je u tim oblastima servis bio nedostupan praktično četvrtinu ukupnog vremena trajanja događaja, uzevši u obzir da je procenjena ukupna vrednost prodaje za vreme događaja bila više milijardi dolara (stotine miliona dolara po satu), ukupni gubici za Amazon su bili reda veličine desetine miliona dolara.



Slika 7.6. Lokacije na kojima servis nije bio dostupan za vreme Amazon Prime Day 2018

Amazon naravno nije jedini primer. Veliki broj avio kompanija je imao slučajeve kada njihove veb aplikacije nisu bile dostupne po više sati, prilikom objave različitih prazničnih popusta koji obično traju ograničeno vreme. Još jedan primer je *Encyclopedia Britannica*, koja je imala probleme sa pristupom aplikaciji u trajanju od više nedelja, zbog prevelikog saobraćaja nastalog nakon promotivne ponude za besplatan pristup njihovoj online bazi.

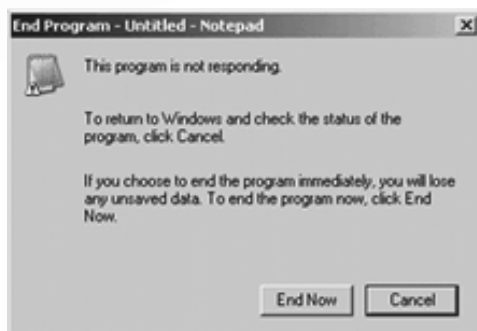
Veb sajt čak ne mora biti u potpunosti nedostupan kako bi se pojavili problemi. Svakom veb sajtu se usporava vreme odziva prilikom prevelikog broja zahteva. Sa druge strane, veliki broj korisnika će prosto odustati od čekanja ukoliko se stranica ne učita unutar 10 sekundi.

Testiranje opterećenja pruža poverenje u pouzdanost sistema prilikom maksimalnog očekivanog opterećenja. Kada se pri testiranju opterećenje podigne na nivo značajno viši od maksimalnog očekivanog, počinje stres test.

7.3. Stres test

Stres test podrazumeva stavljanje aplikacije pod ekstremno opterećenje, kako bi se posmatralo njeno ponašanje za vreme ogromnog broja zahteva ili obrade podataka. Cilj stres testa je da se aplikacija izbací izvan normalnog maksimalnog operativnog kapaciteta i dovede do tačke pucanja. Prilikom određivanja tačke pucanja, određuju se i bezbednosna ograničenja upotrebe aplikacije, koja je potrebno uporediti da li odgovaraju zahtevima u specifikaciji sistema. Na taj način se potvrđuje stabilnost i pouzdanost sistema, kao i ispunjenje zahteva zadatih u specifikaciji. Od izuzetnog značaja je i utvrditi kako tačno sistem puca, odnosno utvrditi oblik otkaza sistema.

Čak i u slučaju ekstremnog opterećenja, sistem ne bi trebalo da u potpunosti otkaže, već bi trebalo da prikaže odgovarajuću poruku o grešci. Problem je najlakše opisati na primeru relativno jednostavne aplikacije poput Notepad aplikacije. Stres test bi bio kopiranje ogromne količine teksta (na primer 5GB teksta) u Notepad. Aplikacija je u tom slučaju pod stresom i daje očekivanu *Not responding* poruku o grešci, kao što je prikazano na slici 7.7.



Slika 7.7. Stres test Notepad aplikacije

Kao što je navedeno u prethodnom poglavlju gde je opisan test opterećenja, kritične situacije za svaku aplikaciju su praznični popusti, rasprodaje i slične prilike koje mogu izazvati nagli porast korisnika. Kako bi se na adekvatan način pripremili za takve događaje, svaka aplikacija bi uz test opterećenja morala da prođe i stres test. Imperativ je verifikacija da aplikacija na adekvatan način može da se izbori sa ekstremnim brojem zahteva, makar kroz adekvatnu poruku o grešci. Ukoliko aplikacija samo pukne prilikom prevelikog saobraćaja, obično je rezultat gubitak reputacije, korisnika, i na kraju krajeva, novca.

Svaki dobro osmišljeni stres test se mora fokusirati na sledeće stavke:

- Provera da li sistem radi dobro pod ekstremnim opterećenjem.
- Provera da li sistem prikazuje adekvatnu poruku o grešci u situaciji kada je opterećenje toliko da ne može da odgovori na zahteve korisnika.
- Ukoliko sistem potpuno otkáže pod ekstremnim opterećenjem, može doći do gubitka podataka, novca i poverenja korisnika.

Prilikom stres testa, analizira se i ponašanje sistema nakon otkaza. Kako bi stres test bio smatran uspešnim, sistem mora prikazati adekvatnu poruku o grešci, bez gubitka kritičnih podataka. Tester i obično koriste masivne setove podataka kako bi izazvali ekstremno opterećenje. Nakon otkaza sistema, neophodno je i verifikovati da se sistem uspešno oporavlja nakon otkaza.

7.4. Regulatorno testiranje

Regulatorno testiranje, takođe poznato kao testiranje pridržavanja standarda (engl. *compliance*, odnosno *conformance testing*), je tip testiranja koji određuje da li je softverski sistem u skladu sa internim i eksternim standardima. Interni standardi su standardi koje je postavila sama kompanija. Uključuju odgovarajući stil pisanja koda, odgovarajuće standarde pisanja veb stranica (na primer da sve veb stranice moraju biti responsivne), da je kod adekvatno dokumentovan i slično. Pod eksternim standardima se podrazumevaju standardi koji su postavljeni od strane neke spoljašnje organizacije, koja nije kompanija koja je razvila softver. Mogu biti zakoni neke države, kojih aplikacija mora da se pridržava ukoliko želi da gađa tržište te države. Sa druge strane, mogu biti i standardi različitih regulatornih organizacija, kao na primer HIPAA (*Health Insurance Portability and Accountability Act*), koja je postavila odgovarajuće zahteve u okviru zdravstvene industrije, a svi softverski sistemi u tom domenu moraju da ih ispune.

Regulatorno testiranje može izvršiti i neka eksterna organizacija, što kao rezultat može imati odgovarajući sertifikat o pridržavanju standarda (engl. *compliance certification*). Metode i dubina testiranja koji se koriste u regulatornom testiranju veoma zavise od specifičnog standarda koji se proverava.

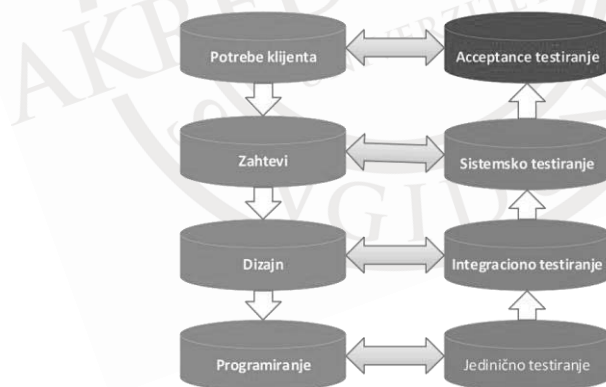
7.5. Test prihvatanja od strane korisnika (UAT)

Test prihvatanja od strane korisnika (engl. UAT - *User acceptance testing*, ili skraćeno *Acceptance testing*) je oblik testiranja koji se inicira od strane klijenta u trenutku kada je sistem spreman za isporuku, nakon što se u toku sistemskog testiranja uoči i ispravi većina defekata. Glavni cilj ovog testiranja je da klijent stekne poverenje u implementirani sistem. Proverava se funkcionalnost prema specifikaciji, i utvrđuje da li je sistem adekvatan za krajnje korisnike.

Ovaj tip testiranja izvršavaju klijenti, odnosno krajnji korisnici aplikacije, kako bi se osiguralo da sistem zaista ispunjava sve što je dogovoreno u specifikaciji zahteva (slika 7.8). Testiranje se izvršava u finalnoj fazi projekta, neposredno pre puštanja aplikacije na tržište ili u produkciono okruženje.

Nakon što je softver prošao jedinično, integraciono i sistemsko testiranje, može se postaviti pitanje zašto je uopšte potrebno testiranje prihvatanja od strane korisnika. Na prvi pogled, deluje redundantno. Međutim, ovo testiranje je obavezno zbog dve kritične pojave koje se mogu desiti pri razvoju softvera:

- Programeri mogu pogrešno da protumače zahteve i da implementiraju funkcionalnosti onako kako su to oni razumeli (što ne mora biti ispravno).
- Ponekad naknadne promene u zahtevima ne budu uspešno iskomunicirane programerima, samim tim i ne budu implementirane.
-



Slika 7.8. Acceptance testiranje u V modelu

Programeri pišu kod na osnovu zahteva koji se nalaze u dokumentu specifikacije zahteva sistema. Pošto su ljudi podložni greškama, vrlo često se dešava da se zahtevi pogrešno protumače od strane programera. Sa druge strane, klijent takođe možda nije bio dovoljno precizan prilikom definisanja zahteva. Kao rezultat, implementirani softver možda ne ispunjava potrebe klijenta, odnosno nije ono što je klijent zamislio.

Pre započinjanja testiranja, neophodno je da budu ispunjeni sledeći preduslovi:

- Dokument sa korisničkim zahtevima je dostupan.
- Aplikacija je kompletno implementirana.
- Kompletirane su faze jediničnog, integracionog i sistemskog testiranja.
- Ne postoje otvoreni kritični defekti (engl. *showstopper*), kao ni defekti visokog i srednjeg nivoa.
- U opštem slučaju dozvoljeni su samo defekti niskog nivoa, poput kozmetičkih promena i slično.
- Svi ostali prijavljeni defekti višeg nivoa su ispravljani i verifikovani.
- Potvrda od strane QA tima da je softver spreman za predaju klijentu.

U praksi se UAT najčešće deli na dve faze – alfa i beta testiranje, kao što je prikazano na slici 7.9.



Slika 7.9. Alfa i beta testiranje kao deo UAT

7.5.1. Alfa testiranje

Alfa testiranje je deo UAT, i spada u jednu od najčešće korišćenih strategija u razvoju softvera. Glavni cilj je da se identifikuju svi potencijalni problemi pre isporuke softvera klijentima na beta testiranje. Testiranje se vrši u laboratorijskim uslovima na lokaciji firme koja proizvodi softver, gde programeri posmatraju testere koji simuliraju krajnje korisnike i beleže sve uočene probleme. Korisnici se simuliraju upotrebom različitih tehnika crne i bele kutije, sa ciljem da se izvrše i provere sve akcije koje bi tipični korisnik mogao da napravi. Alfa testiranje tipično izvodi tim nezavisnih testera u okviru iste kompanije koja razvija softver. Nezavisan tim se uzima zbog objektivnosti pri testiranju, pošto nije učestvovao u ranijim fazama testiranja aplikacije.

Najprostije rečeno, ovo testiranje se vrši na samom kraju razvoja softvera sa ciljem da se sve pronađene greške isprave pre isporuke klijentu, kako bi što je moguće manje problema isplivalo u beta testiranju sa strane klijenta i kasnije nakon puštanja softvera na tržište.

7.5.2. Beta testiranje

Beta testiranje (još poznato pod nazivom testiranje na terenu – engl. *Field Test*) je testiranje koje se izvodi na lokaciji klijenta. Beta testiranje izvršavaju stvarni korisnici aplikacije u stvarnim uslovima, i često se posmatra kao eksterni UAT. Beta verzija softvera se isporučuje ograničenom broju stvarnih korisnika koji ga instaliraju i koriste u realnim uslovima upotrebe. Cilj beta testiranja je da stvarni korisnici upotrebom softvera u realnim uslovima otkriju potencijalne greške ili propuste iz korisničke perspektive, koja ne treba da postoji u finalnoj verziji aplikacije. Kao rezultat, korisnici daju povratnu informaciju o kvalitetu proizvoda.

Beta testiranje smanjuje rizik od neuspeha softvera i u opštem slučaju značajno poboljšava kvalitet softvera i zadovoljstvo korisnika kroz validaciju od strane klijenta. To je finalno testiranje pre isporuke softvera svim krajnjim korisnicima, a povratna informacija od klijenata je glavna prednost beta testiranja. Mnoge velike firme, uključujući Microsoft, koriste ovu prednost i daju beta verzije svojih proizvoda ograničenom broju krajnjih korisnika na testiranje.

Postoje dve varijante beta testiranja:

- Zatvorena beta – softver se isporučuje odabranoj grupi pojedinaca, koji učestvuju i testiraju aplikaciju isključivo po pozivu.
- Otvorena beta – softver je dostupan većoj grupi ili čak celoj javnosti, a mogu da učestvuju svi zainteresovani. Testerij prijavljuju sve greške koje uoče, a vrlo često sugerišu i dodatne funkcionalnosti za koje misle da bi trebalo da budu dostupne u finalnoj verziji.

Glavne prednosti beta testiranja su očigledne, pošto se omogućava direktna povratna informacija od krajnjih klijenata. Pošto testiraju iz korisničke perspektive, beta testerij često pronadu greške koje do tada nisu primećene, i koje mogu biti ispravljene na vreme.

Pitanja

1. Šta je sistemsko testiranje?
2. Da li se sistemsko testiranje radi metodama crne ili bele kutije? Obrazložiti odgovor.
3. Da li se u okviru sistemskog testiranja pokrivaju nefunkcionalni zahtevi?
4. Koji tipovi sistemskog testiranja se najčešće sprovode?
5. Šta je testiranje performansi sistema?
6. Koji su najčešći problemi u performansama sistema?
7. Koji se parametri najčešće prate u testiranju performansi sistema?
8. Šta je najčešće usko grlo?
9. U kom slučaju se kaže da sistem ima lošu skalabilnost?
10. Šta je to test opterećenja?
11. Zašto je neophodno sprovesti test opterećenja?
12. Koje su sličnosti, a koje razlike između testiranja opterećenja i stres testa?
13. Šta je to regulatorno testiranje?
14. Šta je testiranje prihvatanja od strane korisnika?
15. Da li se testiranje prihvatanja od strane korisnika radi metodama crne ili bele kutije? Obrazložiti odgovor.
16. Zbog čega je testiranje od strane korisnika neophodno?
17. Šta je to alfa testiranje?
18. Zašto je neophodno da alfa testiranje izvode nezavisni testeri?
19. Šta je to beta testiranje?
20. Koja je razlika između zatvorene i otvorene bete?

8. STATIČKO TESTIRANJE

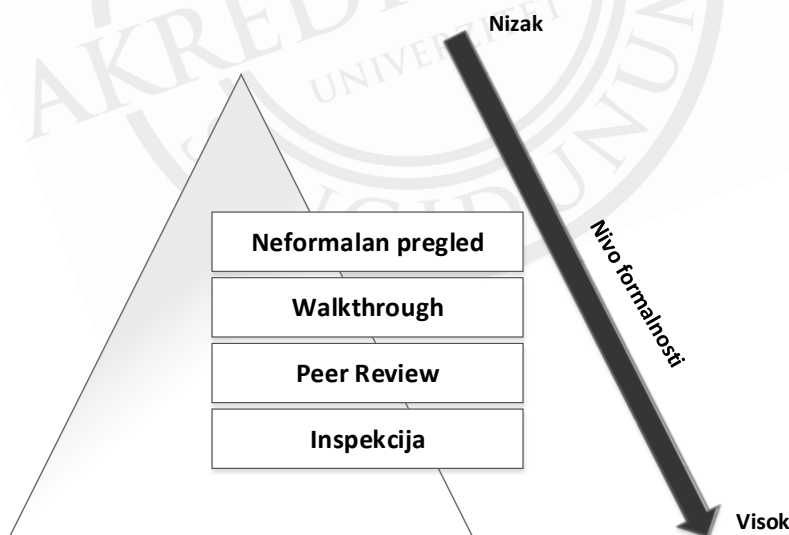
Statičko testiranje predstavlja skup tehnika koje omogućavaju poboljšanje kako kvaliteta softvera, tako i efikasnosti i produktivnosti samog procesa razvoja softvera. Statičko testiranje se bazira na procesu statičkog pregleda (engl. *review*). Glavni cilj ovog tipa testiranja jeste da se u što ranijoj fazi razvoja pronadu defekti.

Uopšteno gledano, statičko testiranje se svodi na testiranje softvera bez njegovog izvršavanja i pokretanja, bilo ručno bilo putem alata. Moguće je izvršiti vrlo rano u razvoju softvera. Nije potreban računar, jer se testiranje programa radi bez izvršavanja samog programa. Statičko testiranje se može sprovesti na izvornom kodu, dizajnu, modelima i arhitekturi aplikacije, funkcionalnim zahtevima, kao i specifikaciji zahteva.

Najčešći tipovi statičkog testiranja uključuju:

- Neformalni pregled
- Formalni pregled
- *Walkthrough*
- Tehnički pregled
- *Peer review*
- Inspekciju.

Ovi tipovi su prikazani na slici 8.1, prema nivou formalnosti koji zahtevaju.



Slika 8.1. Tipovi statičkog testiranja prema nivou formalnosti

Statičko testiranje se može započeti vrlo rano u procesu razvoja, samim tim se i potencijalne greške mogu uočiti veoma rano. U razvoju softvera važi pravilo da što se kasnije pronađe defekt, teže i skuplje ga je ispraviti. U slučaju statičkog testiranja, pronađeni defekti se mogu ispraviti veoma jeftino po pitanju uloženog vremena i troškova. Na taj način se direktno poboljšava i produktivnost, jer u kasnijim fazama razvoja ima manje posla oko ispravljanja grešaka.

Najčešći tipovi defekata koji se mogu pronaći tokom statičkog testiranja uključuju:

- Nepotpuni i nejasni zahtevi u specifikaciji zahteva
- Greške u arhitekturi sistema
- Greške u dizajnu sistema
- Nepotpune specifikacije interfejsa između komponenti sistema
- Neispunjavanje određenih zahtevanih standarda (internih i eksternih)
- Previše komplikovan kod

8.1. Neformalni pregled

Neformalni pregled (engl. *informal review*) se odnosi na najmanje formalni pregled, koji se vrlo često koristi u ranim fazama razvoja različitih dokumenata koji su deo projekta. Pod dokumentom u ovom smislu se može posmatrati specifikacija zahteva, dizajn ili arhitektura sistema, lista testova, kod ili slično. Cilj neformalnog pregleda je da autor dokumenta prezentuje dokument široj publici, koja u ovom slučaju može biti menadžment kompanije, poslovni klijenti, tim programera, tim testera ili kolege. Pošto je u pitanju neformalni pregled, obično nije potrebno da se dokumentuje.

U ranijim fazama razvoja dokumenta dovoljne su dve osobe za ovakav tip pregleda. Kasnije često uključuje više ljudi kroz neki oblik sastanka. Za vreme i nakon sastanka svi prisutni mogu dati svoje mišljenje i podeliti svoje iskustvo, sa ciljem da se kroz zajedničku analizu poboljša kvalitet samog dokumenta i uoče eventualni defekti. Ovakav oblik statičkog testiranja je veoma koristan pošto pomaže u ranoj identifikaciji defekata i sprečava da se defekti propagiraju u fazu razvoja softvera ili fazu testiranja, kada ih je mnogo teže i skuplje ukloniti, i kada mogu uticati na kašnjenje celog projekta.

8.2. Formalni pregled

Formalni pregled, za razliku od neformalnog, zahteva formalni proces. Ovaj proces je strukturiran i regulisan, i sastoji se od šest osnovnih faza:

- Planiranje
- *Kick-off*
- Priprema
- Formalni sastanak

- Ispravka
- *Follow-up*

Sve faze imaju svoju ulogu u procesu, i održavaju se u zadatom redosledu. Ovaj tip pregleda se obavezno dokumentuje. Svi oblici formalnog pregleda prate ovu strukturu procesa. Svi učesnici u procesu imaju definisane uloge i zaduženja.

8.2.1. Planiranje

Planiranje je prva faza formalnog pregleda, koja počinje zahtevom za pregled koji autor upućuje moderatoru. Moderator određuje datum, vreme i mesto sastanka i poziva sve druge članove čije je prisustvo potrebno na sastanku. Moderator vrši proveru da li je dokument uopšte spreman za pregled (ulazni kriterijum, engl. *entry criteria*) i definiše formalni izlazni kriterijum (engl. *exit criteria*). Ulazna provera je potrebna, da se ne bi gubilo vreme na dokument koji još uvek nije dovoljno završen. Ukoliko dokument ne ispunjava ulazne zahteve, vraća se autoru na doradu. Sa druge strane, ako dokument nakon ulazne provere ima malo defekata, može se nastaviti sa pregledom.

Ulazni kriterijum se najčešće definiše na sledeći način:

- Dokument nema previše kritičnih defekata
- Dokument je pročišćen primenom odgovarajućih automatskih provera koje se mogu primeniti na taj tip dokumenta (na primer *spell check*)
- Autor dokumenta je siguran u kvalitet dokumenta i može da se pridruži timu za pregled sa tim dokumentom.

Nakon što dokument prođe ulaznu proveru, moderator i autor odlučuju koji deo dokumenta se pregleda. Pošto ljudski um može da razume jako ograničen broj stranica u jednoj iteraciji, najčešće se posmatra maksimalno 10-20 stranica.

8.2.2. Kick-off

Kick-off sastanak je opcioni korak u procesu. Ovaj sastanak ima za cilj davanje kratkog uvoda o ciljevima pregleda i dokumentima svim uključenim u sastanak. Objašnjavaju se i veze sa drugim dokumentima ukoliko postoje, naročito u slučaju da su ove veze brojne i kompleksne. Prema nekim statistikama, primena kick-off sastanka uvećava broj uočenih kritičnih grešaka po strani dokumenta za 70%. Upravo zbog ove činjenice se ovaj korak skoro uvek primenjuje, iako je opcioni.

8.2.3. Priprema

U ovoj fazi, svaki član tima pregledača samostalno pregleda dokument. Koriste se odgovarajuće procedure, pravila, kao i kontrolne liste gde su navedene sve stavke koje

je potrebno verifikovati u dokumentu. Svaki član tima pregledača individualno beleži sve uočene defekte, pitanja i eventualne komentare. Svi uočeni problemi se beleže i čuvaju u nekom obliku log forme.

Ključni faktor je temeljna priprema svih članova tima pregledača. Za temeljnu pripremu je obično potreban tempo provere od 5 do 10 strana po satu. Ukoliko je tempo brži, dokument se ne pregleda dovoljno detaljno. Ukoliko je sporiji, gubi se previše vremena.

8.2.4. Formalni sastanak

Sam formalni sastanak se sastoji iz tri faze:

- Faza logovanja
- Faza diskusije
- Faza odlučivanja

U fazi logovanja, svi problemi koji su uočeni u fazi pripreme članova tima pregledača se beleže stranu po stranu, najčešće od strane samog autora ili zapisničara. Zapisničar je poseban član tima koji je često zadužen samo za beleženje, i veoma je koristan u najformalnijim oblicima pregleda poput inspekcije.

Svaki uočeni defekt i njegova ozbiljnost se zavode u jednu od tri ponuđene klase defekata:

- *Critical* – kritični defekti koji će sigurno ugroziti ceo projekat
- *Major* – ozbiljni defekti, koji mogu da ugroze ceo projekat
- *Minor* – manji defekti koji najverovatnije neće imati veći uticaj na tok projekta

Za vreme faze logovanja moderator se fokusira na beleženju što je moguće većeg broja defekata unutar određenog vremenskog limita. Uobičajeno se smatra da je sastanak efikasan ukoliko se beleži jedan do dva defekta po minutu.

Ukoliko bilo koji problem zahteva diskusiju, on se razmatra u fazi diskusije. Diskusija vrlo često može da postane burna, pošto autor brani svoj dokument, dok članovi tima pregledača ukazuju na propuste i sugerišu druge opcije. Moderator je zadužen da prati da ne dođe do previše lične diskusije i svađa, i ukoliko je potrebno može da zahteva pauzu da se smire strasti. Ishod diskusije se beleži za kasnije praćenje.

U fazi odlučivanja, na kraju sastanka, donosi se odluka o dokumentu koji se pregleda. Odluka se bazira na formalnom izlaznom kriterijumu, koji je najčešće broj kritičnih i ozbiljnih defekata pronađenih po stranici dokumenta. Ukoliko izlazni kriterijum nije ispunjen, dokument se vraća autoru i ponovo se pregleda nakon ispravki.

8.2.5. Ispravka i follow up

U fazi ispravke, ukoliko je pronađeni broj defekata po stranici veći od dozvoljenog nivoa, dokument mora biti ispravljen. Nije uvek neophodno ispraviti sve defekte, već ne je na autoru da proceni da li određeni defekt treba da bude ispravljen ili ne. Ukoliko ništa ne može da se preduzme povodom nekog defekta, potrebno je makar zabeležiti da je autor razmotrio problem.

U follow up fazi, moderator proverava da li je autor izvršio odgovarajuće akcije nad prijavljenim defektima. Ukoliko se odluči da je neophodno da svi učesnici provere i ažurirane dokumente, moderator vrši distribuciju i prikuplja povratne informacije. Na moderatoru je takođe da osigura da su sve informacije tačne, i sačuvane za dalju analizu.

8.2.6. Uloge i odgovornosti učesnika u formalnom pregledu

U formalnom pregledu postoji četiri tipa učesnika: moderator, autor, zapisničar i pregledač. Svaki od ovih učesnika ima precizno definisanu ulogu i odgovornosti. Broj članova tima zavisi od konkretnog problema, a članovi se angažuju prema stručnosti iz određene oblasti. U svakom dobrom timu najčešće postoji nekoliko pregledača koji imaju različitu oblast ekspertize, koji svojim znanjem iz te oblasti mogu ukazati na potencijalne propuste i defekte.

Moderator, odnosno vođa pregleda (engl. *review leader*), je najbitnija uloga u formalnom pregledu. Moderator vodi proces pregleda i učestvuje u svim fazama. Najčešće je to jedan od najiskusnijih članova tima ili vođa projekta, pošto uloga zahteva određen nivo veština u vođenju tima, kao i adekvatno tehničko znanje u oblasti softvera koji se pregleda. Moderator izvršava ulaznu proveru dokumenta, zakazuje i vodi sastanke, i direktno nadgleda ceo proces. Dozvoljeno je da moderator istovremeno ima i ulogu jednog od pregledača. Moderator je takođe zadužen da organizuje i follow up, i da napravi i distribuirati izveštaj pregleda svim učesnicima i zainteresovanim stranama.

Autor je osoba koja je kreirala dokument koji se pregleda. Autor ne učestvuje u samom procesu pregleda, već mora da obezbedi odgovore na pitanja koje postavljaju članovi tima pregledača. Obaveza autora je da posle pregleda izvrši odgovarajuće ispravke u dokumentu na osnovu liste pronađenih defekata.

Zapisničar je član tima čija je dužnost da beleži sve defekte koji su pronađeni u toku pregleda, kao i sve odluke i predloge koje iznesu članovi tima. Osim toga, zapisničar vodi detaljan zapisnik za vreme sastanka. Može aktivno učestvovati u procesu pregleda, i vrlo često u praksi tu ulogu preuzima jedan od članova tima pregledača.

Pregledači, poznati i kao inspektori, su odgovorni za uspešan proces pregleda. Njihova dužnost je da provere da li materijal sadrži defekte pre sastanka, u okviru faze pripreme. Svi članovi tima pregledača zajedno su odgovorni za donošenje zaključaka i preporuka vezanih za dokument koji je pod pregledom. Menadžer takođe može biti uključen u pregled, u zavisnosti od tehničke kompetencije.

Na osnovu preporuka sa sastanka, menadžment odlučuje o daljim koracima. Ukoliko je potrebno, alokira vreme u rasporedu projekta za ispravke, i odlučuje da li je proces pregleda ispunio ciljeve ili ne.

8.3. Walkthrough

Walkthrough je oblik pregleda koji spada u neformalne procese. Obično sam autor vodi sastanak. Autor vodi sve učesnike kroz dokument, kako bi se postiglo zajedničko razumevanje teme i prikupile povratne informacije. Ovaj tip pregleda je veoma pogodan za ljude koji nisu stručni u razvoju softvera, kao što je često slučaj sa klijentima za koje se aplikacija pravi. Upravo zbog toga se često koristi za dokumente višeg nivoa poput specifikacije zahteva sistema.

Walkthrough za cilj ima:

- Presentaciju dokumenata u svrhu dobijanja povratne informacije o temi dokumentacije.
- Objašnjenje ili prenošenje znanja.
- Evaluaciju sadržaja dokumenta.
- Ispitivanje i diskusiju o validnosti predloženih rešenja.

8.4. Tehnički pregled

Tehnički pregled spada u formalne procese, u manjoj meri u poređenju sa formalnim pregledom. Uobičajeno je da ga vodi trenirani moderator ili tehnički ekspert. Vrlo često se izvršava u formi *peer review*, bez prisustva menadžmenta. Peer review je tip pregleda gde se dokument pregleda od strane autora i jednog ili više kolega, kako bi se evaluirao tehnički sadržaj i kvalitet dokumenta. U praksi, tehnički pregled može da varira od veoma neformalnog do veoma formalnog procesa.

Ciljevi tehničkog pregleda su:

- Verifikacija u ranoj fazi razvoja dokumenta da se tehnički koncepti pravilno upotrebljavaju.
- Procena vrednosti tehničkih koncepata i alternativa.
- Provera konzistentnosti upotrebe tehničkih koncepata.
- Upoznavanje učesnika sa detaljnim tehničkim sadržajem dokumenta koji se pregleda.

8.5. Inspekcija

Inspekcija je najformalniji tip pregleda. Ovakav tip pregleda obavezno vode trenirani moderatori. Za vreme inspekcije, dokumenti se pripremaju i detaljno pregledaju samostalno od strane inspektora pre sastanka, a pronađeni defekti se obavezno dokumentuju u listu. Primenjuje se formalni follow up, uz precizno definisanu upotrebu izlaznog kriterijuma.

Ciljevi inspekcije su:

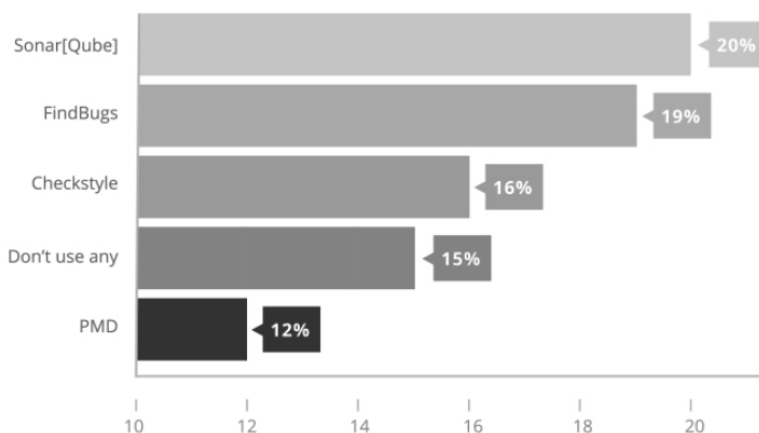
- Rano uočavanje defekata, i njihovo efikasno uklanjanje.
- Autor poboljšava kvalitet dokument pod inspekcijom.
- Poboljšava se kvalitet proizvoda.
- Uči se iz pronađenih defekata, kako bi se sprečilo pojavljivanje sličnih problema u budućnosti.

8.6. Alati za statičku analizu

Postoji veliki broj dostupnih alata za statičku analizu, koji pomažu u procesu pregleda dokumenata. Ove alate najčešće koriste programeri u okviru razvoja i testiranja komponenti. Funkcionišu na sličan način kao kompajleri. Većina modernih kompajlera i sama nudi mogućnost nekog vida statičke analize. Najčešće korišćeni alati su prikazani na slici 8.2.

STATIC CODE ANALYSIS TOOL USAGE BY DEVELOPERS

(SAMPLE SIZE: 2119)



Slika 8.2. Upotreba alata za statičku analizu koda od strane programera

Alati za statičku analizu se koriste za:

- Izračunavanje metrika poput ciklomatske kompleksnosti koda ili nivoa ugnježdavanja, koji mogu pomoći u identifikaciji delova koda koji se moraju detaljno testirati zbog povećanog rizika.
- Osiguravanje primene standarda u programiranju.
- Analizu struktura i zavisnosti u kodu.
- Identifikaciju anomalija i defekata u kodu, poput mrtvog koda, neinicijalizovanih promenljivih i slično.



Pitanja

1. Šta je statičko testiranje?
2. Koji su oblici statičkog testiranja?
3. Koja je razlika između statičkog i dinamičkog testiranja?
4. Koji je najformalniji oblik statičkog testiranja?
5. Koje su faze formalnog pregleda?
6. Šta sve može biti predmet formalnog pregleda?
7. Koje su uloge u formalnom pregledu?
8. Koje su prednosti formalnih pregleda?
9. Koje su mane formalnih pregleda?
10. Koji se alati koriste u statičkoj analizi?
11. Koje su tipične greške koje alati za statičku analizu mogu da otkriju?



9. MODELI PROCESA RAZVOJA SOFTVERA

Modeli procesa razvoja softvera su različiti procesi i metodologije, koji se biraju i koriste u razvoju projekta, u zavisnosti od ciljeva i zahteva projekta. Postoji više modela koji su razvijeni tokom godina, svaki sa drugačijim ciljevima. Uopšteno gledano, modeli specificiraju različite faze u procesu razvoja, kao i redosled kojim se te faze izvršavaju. Odabir modela koji se koristi u razvoju određenog softvera ima ogroman uticaj na testiranje koje će se sprovesti. Odabrani model definiše šta, kada i kako će se testirati, utiče na regresiono testiranje i na odabir tehnika koje će se koristiti

Neki od najčešće korišćenih modela su:

- *Waterfall*
- V model
- Inkrementalni model
- RAD model
- Iterativni model
- Spiralni model
- Prototip
- Agilni model

Ovi modeli su u literaturi poznati pod zajedničkim imenom modeli procesa razvoja softvera, (engl. *Software Development Process Models*). Svaki od ovih modela prati određeni životni ciklus projekta kako bi osigurao uspešan završetak projekta. Svi modeli opisuju faze u razvoju i njihov redosled. Tim za testiranje prati model životnog ciklusa testiranja softvera (engl. *Software Testing Life Cycle – STLC*). Ovaj model je sličan modelu razvoja softvera.

9.1. Faze u modelu razvoja softvera

Svaka od faza u modelu razvoja ima svoj određeni rezultat, koji predstavlja ulaz u sledeću fazu. Na primer, zahtevi klijenta se prevode u dizajn sistema. Na osnovu dizajna sistema, u fazi razvoja se piše kod. Nakon završenog kodiranja, testiranje verifikuje implementaciju softvera i utvrđuje da li je u skladu sa zahtevima.

Uopšteno gledano, u svakom od modela razvoja softvera postoji šest faza:

- Prikupljanje zahteva i njihova analiza
- Dizajn
- Implementacija (kodiranje)
- Testiranje
- Isporuka projekta klijentu (engl. *deployment*)
- Održavanje

U nastavku ovog poglavlja detaljno je objašnjena svaka od ovih faza.

9.1.1. Prikupljanje zahteva

U ovoj fazi se sakupljaju zahtevi klijenta. To je najvažnija faza za menadžere projekata i stejkholdere (engl. *stakeholders*, najpribližniji prevod je interesent). Stejkholderi su osobe koje su uložili nešto u neki biznis, i sa pravom imaju interes da taj biznis bude uspešan. U ovoj fazi se održavaju intenzivni sastanci između menadžera, stejkholdera i klijenta, sa ciljem da se prikupe osnovni zahtevi.

Osnovni zahtevi su tipično:

- Ko će koristiti sistem
- Kako će koristiti sistem
- Koji podaci su ulaz u sistem
- Koji podaci su izlaz iz sistema

Na ova uopštena pitanja potrebno je dobiti što je moguće preciznije odgovore u fazi prikupljanja zahteva. Nakon prikupljanja, zahtevi se analiziraju, i razmatra se kako će se inkorporirati u sistem koji treba da se razvija. Na kraju ove faze kreira se formalni dokument pod nazivom specifikacija zahteva, koji predstavlja osnovu za sledeću fazu modela koji se koristi. Tim za testiranje, koji prati STLC, počinje sa fazom planiranja testiranja čim je završena analiza zahteva.

9.1.2. Dizajn

U ovoj fazi se priprema dizajn sistema i softvera na osnovu specifikacije zahteva koja je dobijena iz prve faze. Potrebno je specificirati kako systemske tako i hardverske zahteve, i definisati uopštenu arhitekturu sistema. Definišu se komponente sistema, korisnički interfejs, interakcija korisnika sa sistemom, način perzistencije podataka, šabloni koji će biti primenjeni, upravljanje greškama i slično. Na kraju ove faze dobija se formalna specifikacija dizajna sistema. Ova specifikacija dizajna sistema predstavlja ulaz u sledeću fazu modela. U ovoj fazi, tester definišu opštu strategiju testiranja, odnosno odlučuje se šta se testira i na koji način se testira.

9.1.3. Implementacija

Faza implementacije je najduža faza u modelu razvoja softvera. Nakon dobijanja specifikacije dizajna sistema iz prethodne faze, posao se deli na module - jedinice i počinje programiranje. Pošto se kod proizvodi u ovoj fazi, ona je najbitnija za programere. Kako je fokus ove knjige na testiranju softvera, dovoljno je samo reći da se na osnovu specifikacije komponenti iste implementiraju, prateći odabrane šablone i odabrane tehnologije. Izlaz ove faze predstavlja implementirani kod, koji je potrebno testirati.

9.1.4. Testiranje

Kod koji je razvijen u fazi implementacije predstavlja ulaz u fazu testiranja. Ova faza je najbitnija za testere. Testeri izvršavaju testiranje na osnovu zahteva klijenta, kako bi se osiguralo da implementirani softver zaista rešava potrebe klijenta koje su definisane u fazi prikupljanja zahteva. U ovoj fazi se izvršavaju svi tipovi funkcionalnog testiranja, jedinično testiranje, integraciono testiranje, sistemsko testiranje, kao i testiranje prihvatanja softvera od strane korisnika.

Pored funkcionalnog, izvršava se i nefunkcionalno testiranje, kako bi se osiguralo da implementirani softver ispunjava i zahteve po pitanju performansi sistema, sigurnosti, maksimalnog korisničkog opterećenja i slično.

9.1.5. Isporuka softvera klijentu

Nakon što je testiranje uspešno završeno, softverski sistem se isporučuje klijentu na upotrebu. Nakon prve isporuke, klijent će najčešće uraditi beta testiranje. U okviru beta testiranja, ukoliko se uoči potreba za nekom dodatnom promenom, ili ukoliko se pronade neki defekt koji nije otkriven u prethodnim fazama, klijent to prijavljuje razvojnom timom. Kada se sve ove poslednje izmene implementiraju i isprave defekti, sledi finalna isporuka softvera.

9.1.6. Održavanje

Glavni cilj ove faze je održavanje i izmena softvera nakon isporuke klijentu. Kada klijent počne da koristi razvijeni sistem, često se pojave stvarni problemi koje je neophodno rešiti sa vremena na vreme. Potrebno je ispraviti bagove i poboljšati performanse sistema. Ponekad je potrebno ukloniti funkcionalnosti koje više nisu potrebne, ili poboljšati postojeće funkcionalnosti. Ove promene su neophodne, pošto se potrebe klijenta vremenom mogu menjati. Faza održavanja daje precizni plan za estimaciju, kontrolu i implementaciju ovih izmena. Tipično, održavanje softvera može nositi između 40% i 80% troškova celog projekta, sa vrlo čestim slučajevima da je stvarni trošak bliži gornjoj proceni. Zbog toga, precizna kontrola faze održavanja može pomoći u smanjivanju ukupnih troškova.

9.2. Tradicionalni modeli razvoja softvera

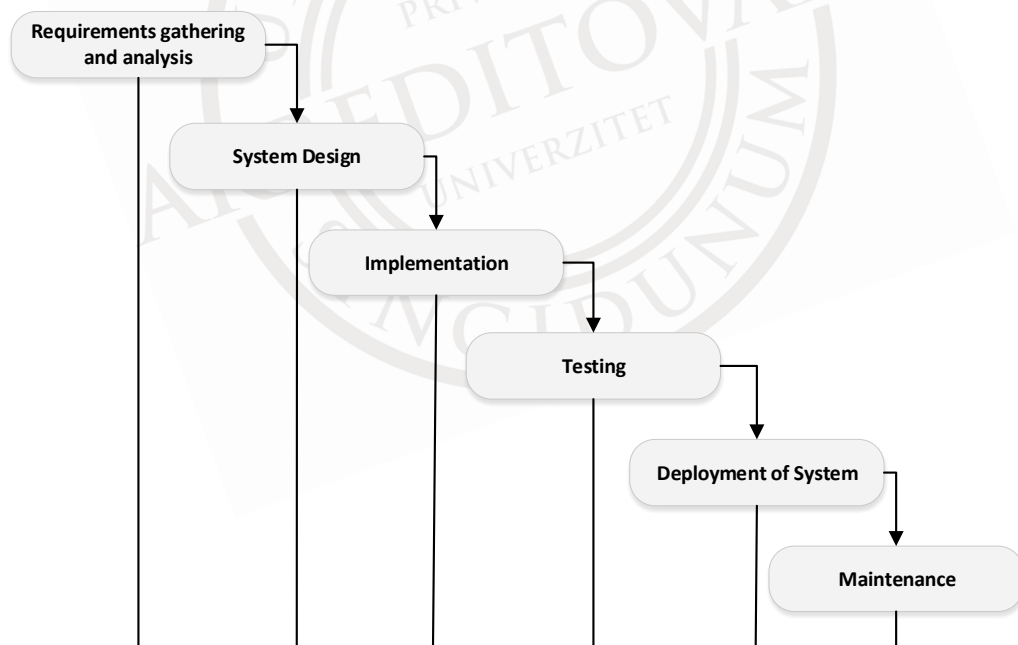
Odabir odgovarajućeg modela za razvoj konkretne aplikacije je veoma važan. Model ima ključni uticaj i na način testiranja aplikacije. Na osnovu odabranog modela se dalje odvijaju procesi razvoja i testiranja softvera. Različite kompanije biraju različite modele

koji najviše odgovaraju tipu aplikacija koje se prave. Danas, na tržištu, najkorišćenija metodologija je agilni model, o kome će biti najviše reči u posebnom delu ovog poglavlja.

Osim agilnog modela, postoji još nekoliko tradicionalnih modela koji se intenzivno koriste. *Waterfall* je veoma stari model, gde se testiranje obavlja nakon završetka implementacije. Pošto se na kraju razvoja prijavi veliki broj grešaka, cena njihove ispravke bude veoma velika, pa kompanije danas preferiraju agilni model. Veliki broj kompanija još uvek intenzivno koristi V model, kao i inkrementalne modele. U nastavku ovog poglavlja su ukratko opisani tradicionalni modeli razvoja softvera, sa posebnim aspektom na to kako utiču na proces testiranja softvera.

9.2.1. Waterfall model

Waterfall model je prvi model koji je uveden u razvoj softvera. Često se naziva još linearni sekvencijalni model. Veoma je lak za razumevanje i upotrebu. Prema Waterfall modelu, svaka faza mora biti u potpunosti kompletirana pre nego što počne sledeća faza, odnosno faze se ne mogu preklapati, kao što je prikazano na slici 9.1.



Slika 9.1. Waterfall model razvoja softvera

Nakon završetka jedne faze, počinje sledeća. Faza testiranja počinje tek kada je razvoj kompletno završen. Na kraju svake faze, održava se formalna revizija kako bi se odredilo da li je projekat na dobrom putu ili ne.

Prednosti modela:

- Lak za razumevanje i upotrebu
- Lak za menadžment zbog rigidnosti modela, svaka faza ima specifične rezultate i reviziju
- Faze se procesiraju i završavaju jedna po jedna, nema preklapanja faza
- Veoma pogodan za manje projekte, gde su zahtevi jasni

Mane modela:

- Kada je aplikacija u fazi testiranja, teško je vratiti se nazad i promeniti nešto što nije dobro promišljeno u ranijim fazama
- Operativni softver se dobija kasno u modelu
- Veliki rizik i neizvesnost
- Nije dobar model za kompleksne i objektno orijentisane projekte
- Loš za duge projekte
- Loš u slučaju rizika od čestih promena zahteva

Ovaj model se koristi u sledećim slučajevima:

- Zahtevi su poznati, jasni i fiksirani (nema naknadnih izmena)
- Tehnologija u kojoj se razvija je poznata
- Nema dvosmislenih zahteva
- Projekat je kratak

U ovom modelu ima jako malo interakcije sa klijentom za vreme razvoja proizvoda. Tek kada je projekat potpuno završen može se pokazati krajnjim korisnicima. U slučaju da tada dođe do ozbiljnih grešaka, cena njihove ispravke je obično veoma velika.

9.2.2. V model

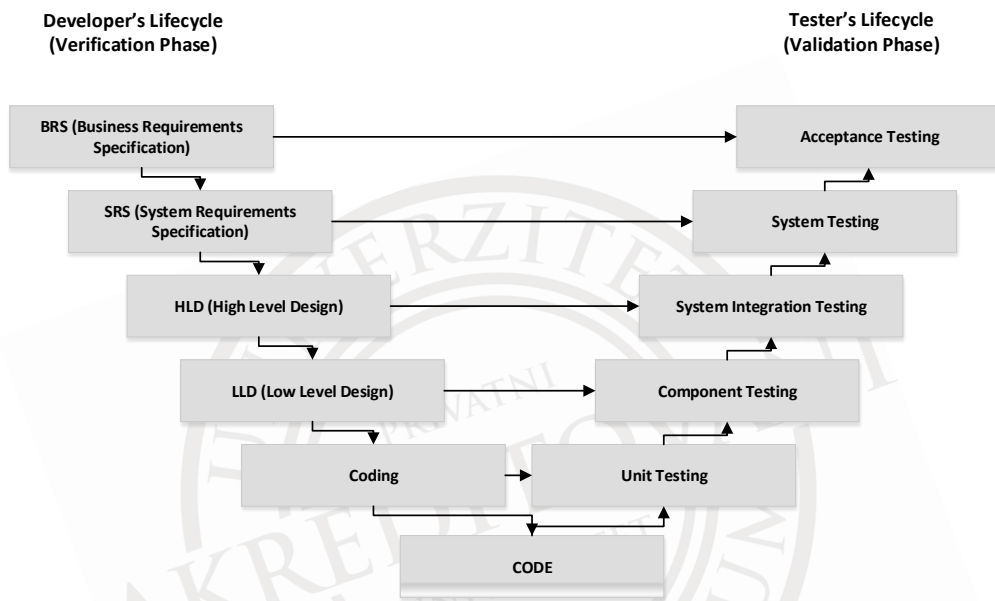
Ovaj model je poznat i pod nazivom verifikacija i validacija. Kao i u slučaju Waterfall modela, u pitanju je sekvencijalno izvršavanje procesa. Svaka faza mora biti kompletirana pre nego što počne sledeća faza u modelu. Testiranje proizvoda se planira paralelno sa odgovarajućom fazom razvoja u V modelu, kao što je prikazano na slici 9.2. Sam model se prikazuje u obliku karakterističnog slova V.

Zahtevi se analiziraju na početku modela. Pre nego što se krene sa razvojem, kreira se plan sistemskog testiranja. Ovaj plan testiranja se fokusira na specificirane funkcionalnosti koje su definisane u fazi prikupljanja zahteva.

Faza dizajna visokog nivoa (engl. *High level design* – HDL) se fokusira na arhitekturu i dizajn sistema. Pruža pregled rešenja, platformi, sistema, proizvoda i

servisa koje je potrebno implementirati. Paralelno se kreira plan integracionog testiranja, kako bi se testiralo kako delovi softvera rade zajedno.

Faza dizajna niskog nivoa (engl. *Low level design* – LLD) se fokusira na dizajn softverskih komponenti. Definiše logiku za svaku komponentu sistema, i kreira se klasni dijagram sa svim metodama i relacijama između klasa. Paralelno se kreira plan testiranja komponenti.



Slika 9.2. V model razvoja softvera

Faza implementacije je faza gde se odigrava kompletno pisanje koda. Kada je kodiranje završeno, putanja izvršavanja kreće desnom stranom V modela, gde se kreirani test planovi stavljaju u upotrebu. Kodiranje se nalazi na dnu V modela, gde se dizajn komponenti pretvara u kod. Jedinično testiranje izvršavaju programeri koji rade na kodu, paralelno sa pisanjem koda.

Prednosti modela:

- Jednostavan i lak za upotrebu
- Određene aktivnosti testiranja se dešavaju pre kodiranja, na primer planiranje i dizajn testova, što štedi dosta vremena
- Defekti se mogu pronaći rano
- Dobar za male projekte, sa jasnim zahtevima

Mane modela:

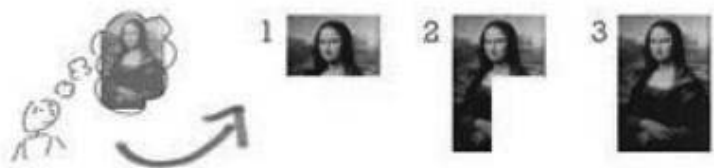
- Veoma rigidan i nefleksibilan
- Softver se razvija unutar faze implementacije, pa ne postoje rani prototipovi koji se mogu pokazati klijentu
- Ukoliko se desi promena zahteva na pola puta u modelu, svi test dokumenti se moraju ažurirati, zajedno sa zahtevima

Ovaj model se koristi:

- U slučaju malih i srednjih projekata, sa jasno definisanim i fiksiranim zahtevima
- Potreban je visok nivo poverenja od strane klijenta, jer se ne proizvode prototipovi dovoljno rano, pa postoji rizik da li će biti ispunjena očekivanja klijenta

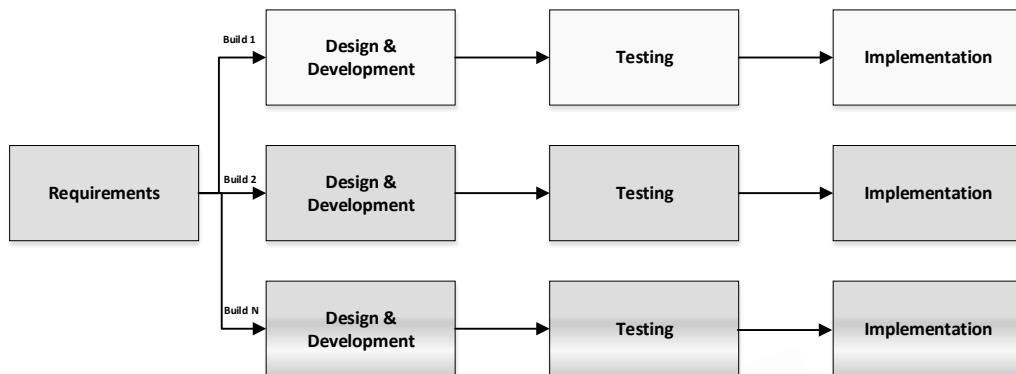
9.2.3. Inkrementalni model

Kod inkrementalnog modela, ceo razvoj je podeljen u više build-ova. Postoji više ciklusa razvoja, pa je ovo zapravo višestruki Waterfall model. Ciklusi se dele u manje module kojima se lakše rukovodi. Svaki modul prolazi kroz faze definisanja zahteva, dizajn, implementaciju i testiranje. Radna, operativna verzija je proizvedena u prvom modulu, tako da operativni softver postoji rano u modelu. Svaki novi *release* modula dodaje novu funkcionalnost na prethodni *release*, kao što je prikazano na slici 9.3. Proces se ponavlja dok se ne implementira ceo sistem.



Slika 9.3. Inkrementalno dodavanje modula

U svakoj fazi se inkrementalno dodaju delovi, pri čemu je svaki deo potpuno završen. Delovi se dodaju dok projekat nije kompletno završen. U prvoj iteraciji, prvi modul je potpuno spreman i može se pokazati klijentu. Zatim, u drugoj iteraciji, drugi modul je kompletno završen i integriše se sa prvim i tako dalje. Pošto je u pitanju višestruki Waterfall model, inkrementalni model se obično tako i prikazuje vizuelno, kao što je prikazano na slici 9.4.



Slika 9.4. Inkrementalni model razvoja softvera

Prednosti modela:

- Operativan softver postoji vrlo brzo, i rano
- Fleksibilnost – manje košta naknadna promena zahteva
- Lakše za testiranje
- Klijent može da učestvuje i odgovori na svaki build
- Smanjuje troškove inicijalne isporuke klijentu (pošto je već bio u susretu sa prethodnim buildovima, kada dobije prvu zvaničnu verziju već ima poprilično dobru sliku o softveru)
- Lakše za upravljanje rizicima, pošto se rizični moduli mogu identifikovati i na njih obratiti posebna pažnja

Mane modela:

- Potrebno je dobro planiranje i dizajn
- Potrebna jasna i kompletna definicija sistema pre nego što se izdela na delove i razvija inkrementalno
- Ukupni trošak je veći nego u običnom Waterfall modelu

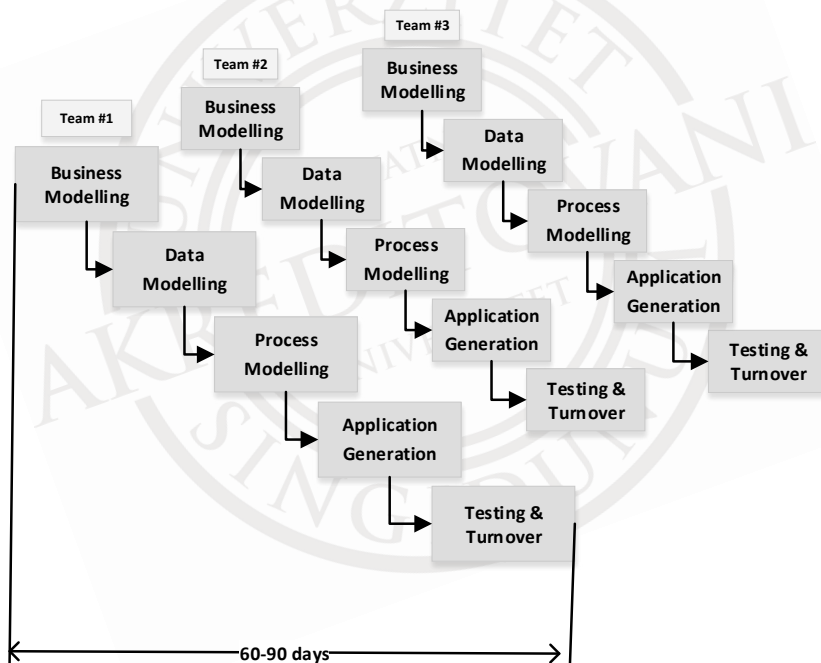
Kada se koristi:

- Postoje jasni zahtevi za ceo sistem
- Glavne funkcionalnosti su definisane, neki detalji mogu da se razvijaju sa vremenom
- Postoji potreba da se proizvod izbací rano na tržište
- Koristi se nova tehnologija
- Postoje rizične funkcionalnosti i rizični ciljevi

9.2.4. RAD model

RAD model (engl. *Rapid Application Development* – RAD) je tip inkrementalnog modela. Komponente ili funkcionalnosti se razvijaju u paraleli kao mini projekti. Razvoj je vremenski ograničen, komponente se brzo isporučuju i sastavljaju u funkcionalni prototip. Na taj način, klijent brzo može da vidi prve rezultate projekta, i da pruži brzu povratnu informaciju.

Definišu se biznis modeli, modeli podataka i modeli procesa, kako bi se što je moguće brže postigao određeni cilj. Na osnovu opisa modela se kreiraju CRUD za podatke (Create, Read, Update, Delete, 4 osnovne operacije za trajno čuvanje podataka). U ovom modelu se intenzivno koriste automatski alati kako bi date modele procesa preveli u kod i stvarni sistem. Nove komponente se testiraju i zatim integrišu zajedno. RAD model je prikazan na slici 9.5.



Slika 9.5. RAD model razvoja softvera

Prednosti modela:

- Skraćeno vreme razvoja
- Brza inicijalna revizija
- Ohrabruje se brza povratna informacija od klijenta
- Integracija od samog početka rešava mnoge integracione probleme

Mane modela:

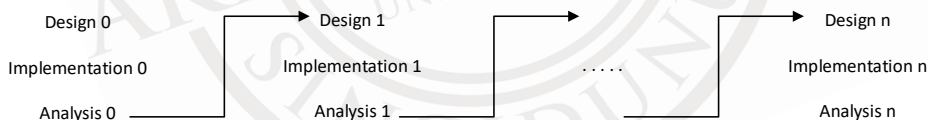
- Zavisí od jakog tima i individualnih performansi
- Samo sistem koji se može jasno podeliti na module se može ovako razvijati
- Potrebni su veoma iskusni programeri
- Veoma zavisi od modelovanja
- Nije primenljivo na jeftinije projekte, jer troškovi modelovanja i automatskog generisanja koda na osnovu modela su veoma veliki

Kada se koristi:

- Potreba za kreiranjem sistema koji se može modularizovati
- Ukoliko je budžet projekta dovoljno veliki kako bi se privukli dizajneri za modelovanje i platili troškovi alata za automatsko generisanje koda
- Ukoliko su svi resursi dostupni, a postoji potreba da se sistem kreira u vrlo kratkom periodu (2-3 meseca)

9.2.5. Iterativni model

Kod iterativnog modela se ne počinje sa punom specifikacijom zahteva odmah od početka. Umesto toga, počinje se sa specifikacijom i implementacijom samo dela softvera, koji se zatim analizira kako bi se identifikovali novi zahtevi. Ovaj proces se ponavlja, i proizvodi nova verzija softvera za svaki ciklus modela, kao što je prikazano na slici 9.6.



Slika 9.6. Iterativni model razvoja softvera

Iako model pomalo liči, na inkrementalni, u suštini se razlikuje. Iterativno se kreira prvo gruba skica kompletnog proizvoda, zatim analizira i poboljšava u sledećim iteracijama sve dok se ne završi. Ceo proizvod se razvija korak po korak, kao što je dato na slici 9.7.



Slika 9.7. Iterativno poboljšavanje proizvoda

Prednosti modela:

- Može se krenuti od visokog nivoa dizajna, a kasnije se može dalje razvijati
- Softver se razvija korak po korak, a defekti se mogu pratiti od rane faze
- Moguće je dobiti povratnu informaciju od klijenta brzo, pokazivanjem skica i šablona
- Manje vremena se troši na dokumentaciju, a više na dizajn

Mane modela:

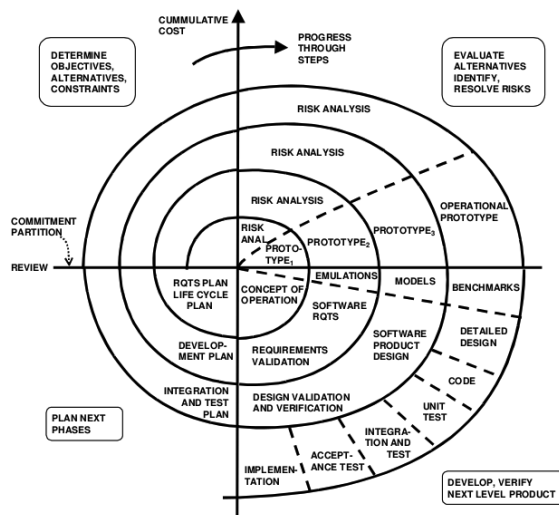
- Faze su rigidne, bez preklapanja
- Moguće su skupe greške u arhitekturi ili dizajnu sistema, jer nisu svi zahtevi bili sakupljeni pre početka razvoja

Kada se koristi:

- Zahtevi za ceo sistem su jasni i razumljivi
- Projekat je veliki
- Glavni zahtevi i funkcionalnosti moraju biti definisani, neki detalji mogu da evoluiraju tokom vremena

9.2.6. Spiralni model

Spiralni model je sličan inkrementalnom modelu, sa većim akcentom na analizu rizika. U ovom modelu postoje 4 faze: planiranje, analiza rizika, razvoj i evaluacija. Softver više puta prolazi kroz ove faze u iteracijama (spirale). Početak spirale startuje od faze planiranja, gde se prikupljaju zahtevi i analizira rizik. Dalje spirale se nadovezuju na početnu, kao što je prikazano na slici 9.8.



Slika 9.8. Spiralni model razvoja softvera

Svaka od faza ima svoju ulogu. U fazi planiranja sakupljaju se zahtevi, poput specifikacije poslovnih zahteva i sistemskih zahteva. U fazi analize rizika identifikuju se rizici, kao i alternativna rešenja. Na kraju ove faze se proizvodi prototip. Ukoliko se pronade rizik, predlažu se i implementiraju alternativna rešenja. U fazi razvoja se razvija softver, zajedno sa testiranjem na kraju faze. U fazi evaluacije klijent evaluira projekat, pre nego što se nastavi sa novom spiralom,

Prednosti modela:

- Često se radi analiza rizika, pa se i poboljšava verovatnoća izbegavanja rizika
- Dobar za velike i kritične projekte
- Jaka kontrola dokumentacije i proces odobravanja od strane klijenta
- Dodatna funkcionalnost se može dodati u kasnijim spiralama
- Softver se proizvodi rano

Mane modela:

- Može biti veoma skup
- Adekvatna analiza rizika zahteva veliko i skupo iskustvo eksperata
- Uspešnost projekta u mnogome zavisi od faze analize rizika
- Nije isplativo za manje projekte

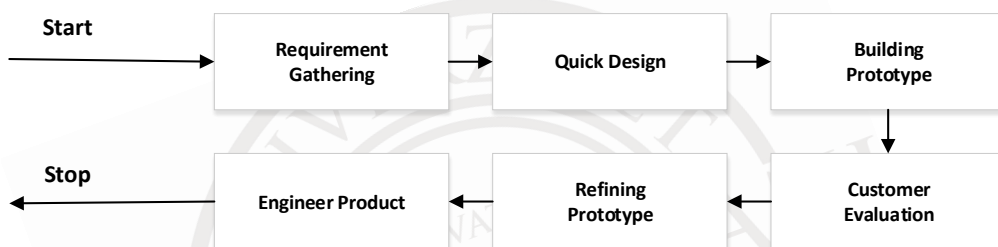
Kada se koristi:

- Ukupni troškovi i evaluacija rizika su važna stavka
- Za srednje i visoko rizične projekte
- Klijenti nisu sigurni u svoje potrebe
- Zahtevi su izuzetno kompleksni
- Očekuju se značajne promene zahteva u toku projekta

9.2.7. Prototip model

Osnovna ideja ovog modela je da se, umesto zamrzavanja zahteva pre početka dizajniranja i kodiranja, pravi jednokratni prototip kako bi se lakše razumeli postavljeni zahtevi. Prototip se pravi na osnovu trenutno poznatih zahteva. Upotrebom prototipa, klijent može steći pravi osećaj sistema, jer interakcije sa prototipom mogu pomoći klijentu da bolje shvati svoje potrebe i zahteve za željeni konačni sistem.

Ovaj model je atraktivna ideja za veoma velike i kompleksne sisteme, naročito gde ne postoji neko već postojeće rešenje, jer pomaže u određivanju i formulisanju zahteva. Prototip obično nije kompletan, već mnogi detalji nedostaju. Postoji samo bazična funkcionalnost. Prototip model je prikazan na slici 9.9.



Slika 9.9. Prototip model razvoja softvera

Prednosti modela:

- Korisnici (klijenti) su aktivno uključeni u razvoj
- Korisnici mogu lakše da shvate sistem koji se razvija
- Greške se mogu uočiti rano
- Brza povratna informacija od klijenta vodi do bolje formulisanih zahteva i na kraju boljeg rešenja
- Nedostajuća funkcionalnost se lako otkriva
- Mogu se identifikovati konfuzne ili kompleksne funkcionalnosti

Mane modela:

- Dovodi do implementacije, a zatim prepravljanja ili popravljanja rešenja
- Povećava kompleksnost sistema, pošto se skala sistema može drastično povećati izvan originalnih planova
- Postoji rizik od pogrešne implementacije sistema, jer prototip nije dobro shvaćen

Kada se koristi:

- Kada potrebni sistem mora da ima mnogo interakcije sa krajnjim korisnicima
- Tipično, sistemi dostupni preko Interneta, odnosno veb interfejsi sa velikom količinom interakcije sa korisnikom

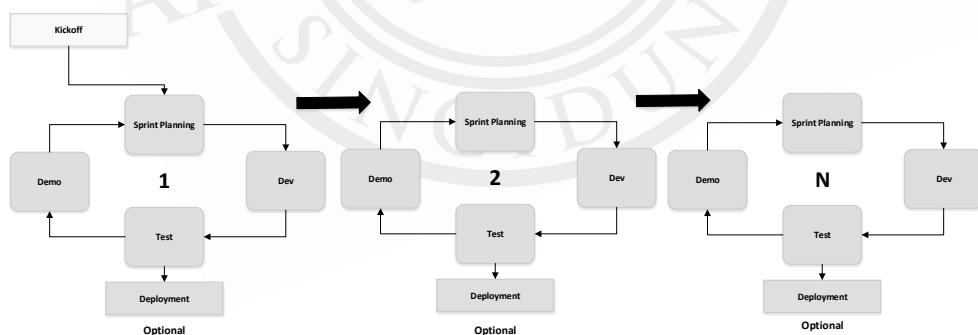
- Prototip se može izraditi brzo, dok razvoj celog sistema može da potraje, a korisnici se brzo navikavaju na prototip, što kasnije olakšava njihovo korišćenje finalnog sistema
- Odličan je za razvoj interfejsa čovek - računar

9.3. Agilni model

Agilna metodologija razvoja je proces razvoja softvera koji se značajno razlikuje od drugih metodologija. *Agile* – na engleskom, znači sposobnost da se kreće brzo i lako, i da se lako odgovori na bilo kakvu promenu. To je i ključni aspekt agilnog razvoja. U slučaju tradicionalnih modela poput Waterfall, projektu je potrebno nekoliko meseci pa i godina da se završi, a klijent najčešće nema uvid u proizvod sve do završetka projekta. Značajno vreme se troši na sakupljanje zahteva, dizajn, razvoj, testiranje aplikacije i testiranje prihvatanja od strane korisnika pre isporuke projekta.

Nasuprot tome, agilni projekti imaju sprintove (iteracije) koji su kratkog trajanja (obično između 1-2 nedelje i mesec dana), u okviru kojih se prethodno određene funkcionalnosti razvijaju i isporučuju klijentu. Može postojati više iteracija, a isporuka kompletnog projekta se vrši na kraju finalne iteracije.

Agilni model je tip inkrementalnog modela razvoja softvera. Softver se razvija u inkrementalnim, kratkim ciklusima. Rezultat su mali inkrementalni release-ovi, gde svaki nešto dodaje na prethodnu funkcionalnost. Svaki release se detaljno testira kako bi se održao kvalitet softvera. Agilni model je prikazan na slici 9.10. Ovaj model se koristi za vremenski kritične aplikacije.



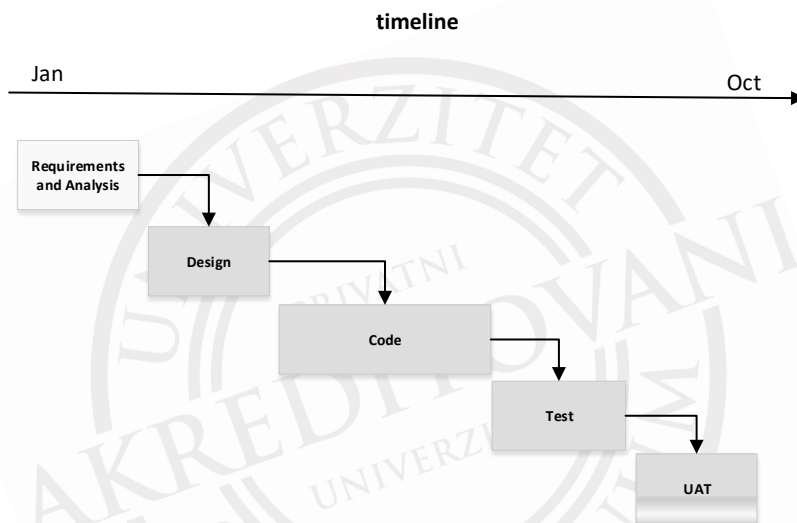
Slika 9.10. Agilni model razvoja softvera

Agilni model je najlakše shvatiti na konkretnom primeru. Razmatra se izrada konkurentnog proizvoda MS Wordu, koji će imati sve funkcionalnosti MS Worda, i bilo koje druge funkcionalnosti koje zahteva marketing tim. Finalni proizvod mora biti spreman za 10 meseci. Kako bi ovaj projekat bio izveden u tradicionalnim metodologijama, u poređenju sa agilnom?

U tradicionalnom Waterfall modelu, razvoj bi bio izveden na sledeći način:

- Oko 15% vremena bi projektni timovi potrošili na sakupljanje i analizu zahteva (1.5 meseci)
- 20% bi otišlo na dizajn (2 meseca)
- 40% na kodiranje i jedinično testiranje (4 meseca)
- 20% na sistemsko i integraciono testiranje (2 meseca)

Na kraju ciklusa, 2 nedelje bi otišlo na test prihvatanja od strane marketing tima. Ceo proces prikazan je na slici 9.11. Klijent ne bi mogao da vidi krajnji proizvod do kraja projekta, kada je prekasno napraviti značajnije izmene.

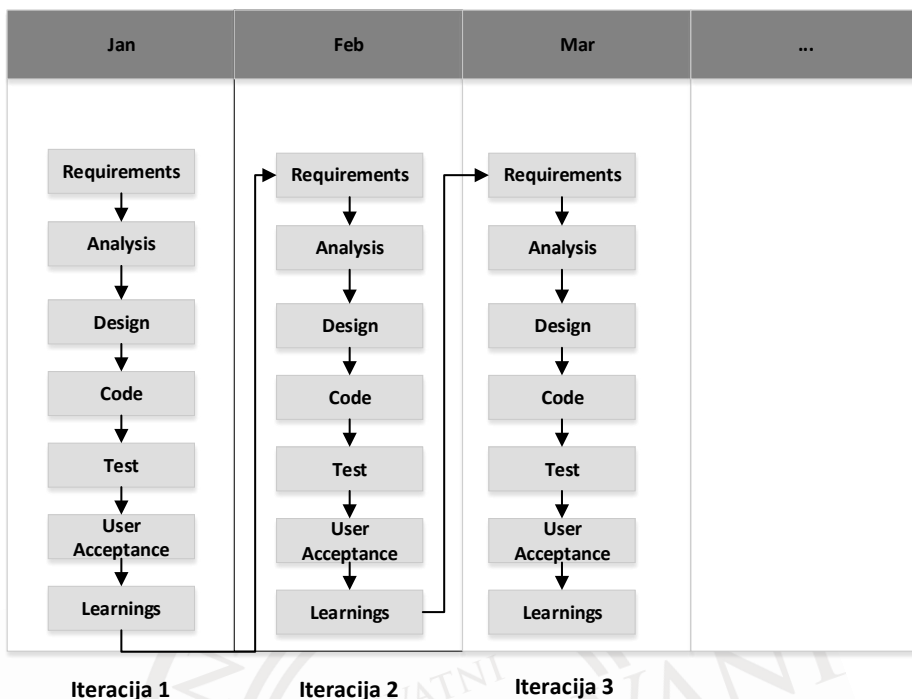


Slika 9.11. Primer projekta sa primenom Waterfall modela

Ukoliko se primenjuje agilna metodologija:

- Projekat se razbija u iteracije (sprintove)
- Svi sprintovi su iste dužine (između 2 i 8 nedelja)
- Na kraju svake iteracije isporučuje se funkcionalni proizvod

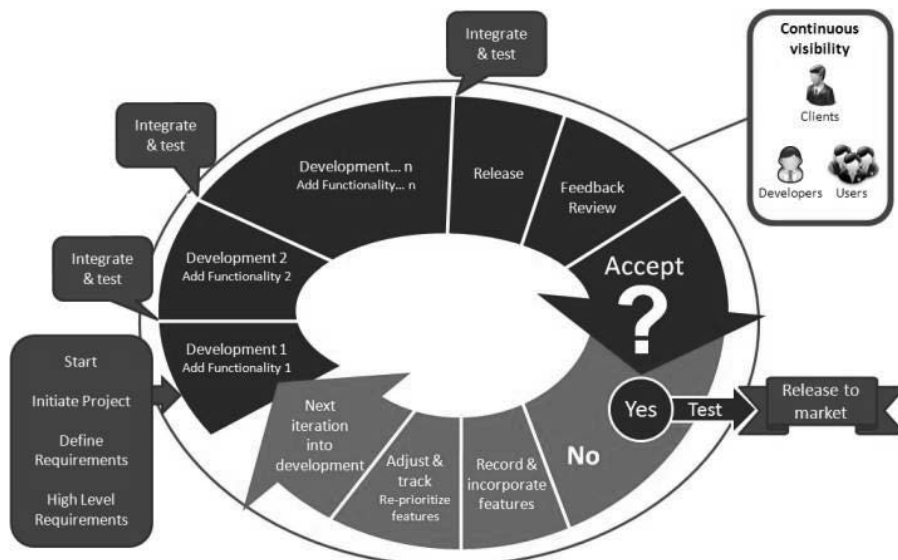
U prostom slučaju, projekat bi bio podeljen u 10 sprintova, sa 10 release-ova pod pretpostavkom da svaki sprint traje 4 nedelje (slika 9.12).



Slika 9.12. Primer projekta sa primenom agilnom modela

U agilnoj metodologiji, umesto da se troši 1.5 meseci na sakupljanje zahteva, tim se odlučuje o osnovnim funkcionalnostima koje su potrebne za proizvod i razvija ih u prvoj iteraciji. Preostale funkcionalnosti koje se ne mogu isporučiti u prvoj iteraciji, se planiraju za sledeće iteracije na osnovu njihovog prioriteta. Na kraju prve iteracije, tim isporučuje operativan softver sa funkcionalnostima implementiranim u prvom sprintu. Biće 10 sprintova, a na kraju svakog se klijentu isporučuje operativni softver koji se inkrementalno unapređuje sa novim funkcionalnostima koje su planirane za odgovarajuću iteraciju, kao što je prikazano na slici 9.13.

Koristeći ovaj pristup, klijentu se omogućava interakcija i rad sa funkcionalnim softverom na kraju svakog sprinta i mogućnost da pruži povratnu informaciju. Tim može da se adaptira promenama veoma lako i da izvrši prepravke ukoliko je potrebno.



Slika 9.13. Detaljna struktura agilnog modela

Slika 9.14 pokazuje izgled softvera koji bi bio isporučen klijentu nakon tri iteracije agilnog modela. U prvoj iteraciji bi se implementirala osnovna funkcionalnost aplikacije, u ovom slučaju prost unos teksta. Aplikacija bi izgledala slično kao Notepad. Nakon druge iteracije, implementirale bi se i osnovne funkcionalnosti za formatiranje teksta, pa bi aplikacija u tom trenutku ličila na Wordpad. U trećoj iteraciji bi bile dodate i naprednije opcije za formatiranje teksta, tako da bi aplikacija nakon treće iteracije već ličila po funkcionalnosti na MS Word.



Slika 9.14. Projekat nakon prve tri iteracije

U agilnoj metodologiji, više se značaja daje kolaboraciji unutar tima i saradnji sa klijentom, kako bi se lako odgovorilo na promene i isporučio funkcionalan softver, u poređenju sa drugim modelima razvoja. Agilni razvoj je trenutno najpopularnija metodologija u IT industriji. Po nekim statistikama, preko 50% kompanija koristi neki oblik agilnog razvoja.

9.3.1. Prednosti agilnog modela

U tradicionalnom razvoju softvera, svaka faza mora biti kompletirana pre prelaska na sledeću. Na primer, kada je gotovo prikupljanje zahteva, prelazi se na fazu dizajna, zatim nakon kompletiranja te faze se prelazi na razvoj, pa na kraju na testiranje. Glavna osobina tradicionalnih modela je da nema preklapanja faza, a greške otkrivene u kasnijim fazama su skupe za ispravku. U slučaju agilnog razvoja, svaka funkcionalnost se kompletira u smislu dizajna, razvoja, kodiranja, testiranja i prepravki, pre nego što se ta funkcionalnost proglasi za gotovu (implementiranu). Ne postoje razdvojene faze, sve se radi u jednoj fazi (iteraciji).

Pristup projektu u agilnoj metodologiji je veoma fleksibilan i adaptibilan, i može se menjati u skladu sa potrebama projekta. Sam proces se bazira na činjenici da se promene očekuju i da su dobrodošle. Nasuprot tome, tradicionalne metode su veoma rigidne i nije lako implementirati promene u projektu. Kako bi ceo proces ostao agiln i lagan, obično se agilnom metodologijom implementiraju manji projekti sa malim i kreativnim timovima. Menadžment projekta u agilnoj metodologiji takođe nije rigidan – vođenje projekta nije centralizovano, već je distribuirano na sve članove tima, nasuprot tradicionalnim modelima gde postoji jedna osoba koja donosi odluke, a svi ostali članovi tima prate.

Uopšteno gledano, u agilnim projektima se zahteva mnogo manje dokumentacije u poređenju sa tradicionalnim modelima. To je neophodno kako bi proces ostao što je laganiji moguć. Međutim, to ne znači da dokumentacija nije potrebna. Još jedna bitna razlika između agilnog i tradicionalnog modela je u planiranju projekta unapred. U slučaju agilne metodologije, planiranje unapred koje je neophodno da se izvrši pre početka projekta je minimalno. Dizajn i arhitektura agilnog projekta se prave što je fleksibilnije moguće na osnovu trenutnih zahteva klijenta (očekuje se da se mogu promeniti). Samim tim, i refaktorisanje koda nije previše skupo u slučaju da je potrebno da se uradi.

Prednosti metodologije se na osnovu svega navedenog mogu svesti na sledeće:

- Isporuka softvera je neprekidna
- Klijenti su zadovoljni jer nakon svakog sprinta imaju funkcionalni softver
- Klijenti mogu da daju povratnu informaciju ili da zatraže izmenu zahteva koja se može uključiti u prvu sledeću isporuku softvera
- Fokus je na interakciji između klijenata, programera, testera
- Fokus na dobrom dizajnu proizvoda
- Bilo koja promena u zahtevima je dobro došla, čak i u kasnijim fazama razvoja

Naravno, kao i svaki model, i agilni model ima određene mane, od kojih se izdvajaju sledeće:

- Obično je dokumentacija slaba, pošto fokus nije na njoj
- Zahtevi mogu da budu nejasni pa je teško predvideti očekivani rezultat

- Često je teško predvideti i proceniti neophodno vreme razvoja da se neka funkcionalnost implementira (engl. *effort estimation*)
- Projekti lako skrenu sa ispravnog puta ukoliko predstavnik klijenta nije dovoljno jasan o finalnim ciljevima koji su traženi
- Samo senior programeri su sposobni da donose neophodne odluke za vreme razvoja, pa nije odgovarajuće mesto za mlade programere (osim ukoliko nisu upareni sa veoma iskusnim resursima).

9.3.2. Agilni manifest

Agilni Manifest (engl. *Agile Manifesto*) je napisan u februaru 2001. godine, od strane 17 nezavisnih autora, i predstavlja skup osnovnih pravila agilne metodologije. Učesnici ovog skupa nisu uspjeli da se slože oko mnogo stvari, ali su uspjeli da pronađu konsenzus za četiri osnovna aspekta koji i danas važe:

- Pojedinci i interakcije pre procesa i alata
- Funkcionalni softver pre jasne dokumentacije
- Saradnja sa klijentom pre pregovora o ugovoru
- Odgovor na promene pre praćenja plana

Stavke sa leve strane svakog od ovih pravila imaju veću vrednost nego stavka sa desne. To ne znači da stavke sa desne treba zanemariti (mada se često u praksi upravo to dešava).

U agilnoj metodologiji, pojedinci i njihove međusobne interakcije se stavljaju na prvo mesto. Tester se u agilnom timu posmatraju kao članovi tima, a ne identifikuju se prema specijalizaciji ili veštinama kao u tradicionalnim modelima. Zapravo, i tester i programer se nazivaju razvojni tim (engl. *development team*). Dakle, razvojni tim sadrži kako programere, tako i testere koji su aktivno uključeni u razvoj proizvoda. Svi članovi agilnog tima često komuniciraju sa ciljem razvoja proizvoda visokog kvaliteta. Ohrabruje se i bliska i česta komunikacija sa klijentom.

Najbitnija stavka se odnosi na funkcionalni softver. U tradicionalnim modelima, poput Waterfall, praksa je da se prvo završavaju dizajn, dokumenti o arhitekturi sistema i test planovi pre nego što se isporuči stvarni komad operativnog softvera klijentu. Softver treba da se isporuči rano i što češće klijentu tokom razvoja. U agilnoj metodologiji, ova praksa se menja isporukom funkcionalnog inkrementa softvera u svakoj iteraciji. To ne znači da dokumentacija treba da se zanemari u potpunosti, već da se proizvodi samo neophodna dokumentacija. Na žalost, ovo pravilo se u većini slučajeva u praksi zloupotrebljava i tumači na pogrešan način, pa se prateća dokumentacija zanemaruje. Kao rezultat, u praksi agilni projekti često pate od vrlo loše ili čak nepostojeće dokumentacije.

Saradnja sa klijentom je neophodna posebno zbog činjenice da klijent najčešće ne zna šta mu stvarno treba dok ne vidi nešto kako radi. Često se dešava da nije moguće imati jasne zahteve pre početka razvoja. Intenzivna saradnja sa klijentom može pomoći

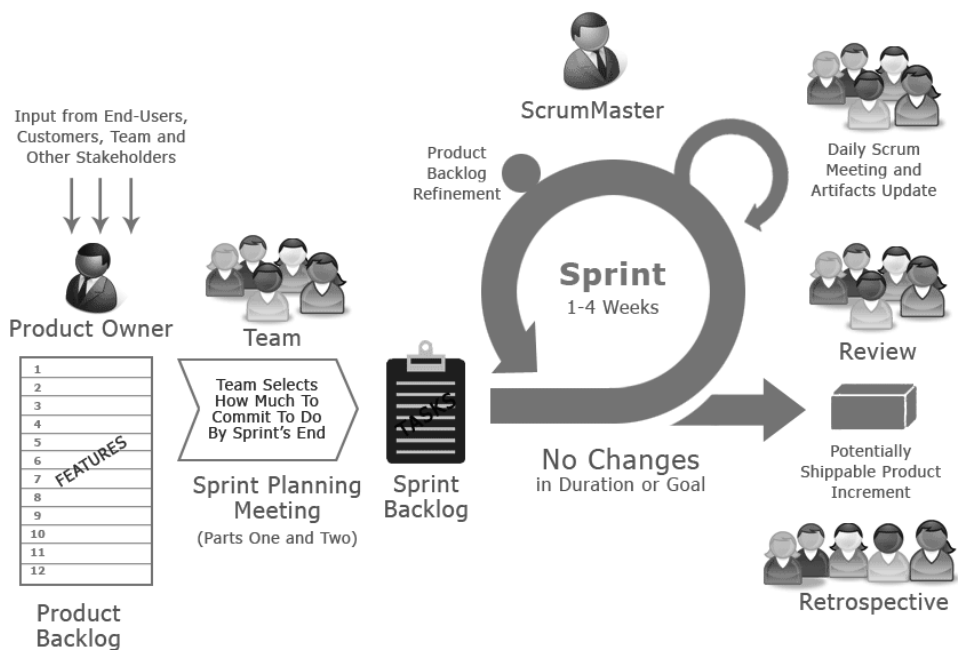
razvojnem timu da shvati šta klijentu zapravo treba. U tradicionalnim modelima, pregovori o ugovoru su obično na prvom mestu, a nakon potpisivanja ugovora blizak kontakt sa klijentom često izostaje, što može dovesti do toga da razvijena aplikacija nije u potpunosti odgovarajuća klijentu. U agilnoj metodologiji pisanje ugovora sa klijentom se smatra kao motiv profita, a intenzivna saradnja kao motiv svrhe.

Odgovor na promene se odnosi na konstantne i neizbežne izmene koje se dešavaju tokom projekta. U poslovnim digitalnom svetu, poslovni prioriteti se menjaju jako brzo, pa i razvoj softvera treba da usvoji potrebu za čestim promenama kako bi se pratili poslovni prioriteti. Bitnija je fleksibilnost da se plan prilagodi promenama, nego da se samo napiše fiksiran i rigidan plan i slepo prati.

9.3.3. Najpopularniji agilni pristupi

Postoji nekoliko oblika agilnog modela koji se često koriste u praksi u kompanijama. Svi oni se pridržavaju agilnog manifesta opisanog u prethodnom poglavlju. Dva najčešća oblika su Scrum i Kanban.

Scrum metodologija je prikazana na slici 9.15. Razvoj softvera se deli na iteracije – sprintove, obično fiksirane na dužinu 1-4 nedelje. U svakoj iteraciji cilj je da se pokuša da se napravi potencijalno isporučiv (i testiran) inkrement proizvoda. Dužina sprinta se obično određuje na osnovu tima, zahteva i kapaciteta. Kad se jednom odredi, dužinu sprinta ne bi trebalo menjati do završetka projekta.



Slika 9.15. Scrum metodologija

Na kraju svakog sprinta tim isporučuje softver koji je testiran, radi i može se pokazati klijentu. Pošto su sprintovi kratki, samo najvažnije funkcionalnosti treba razvijati na početku. Klijent tako može videti osnovne funkcionalnosti veoma brzo, može ih testirati i dati povratnu informaciju.

Skup svih zahteva se deli na male radne delove i prioritizuje se u listu, koja se naziva *Product backlog*. Skup svih zahteva koji se nalaze u trenutnom sprintu se nalazi u listi pod nazivom *Sprint backlog*. Sprint backlog se popunjava zahtevima u okviru *Sprint Planning* sastanka, u kome učestvuje kompletan razvojni tim. Zahtev sa najvećim prioritetom je na prvom mestu, a ostali zahtevi su izlistani po opadajućim prioritetima. Sve ove zahteve je potrebno završiti unutar trenutnog sprinta. Članovi tima uzimaju zadatke iz Sprint backlog-a i rade na njima.

Svi razvojni timovi održavaju dnevni sastanak pod nazivom *Daily Stand Up Meeting* (još poznat i kao *Scrum Meeting*). U okviru sastanka, svaki član tima daje kratki izveštaj i odgovore na 3 pitanja ostatku tima. Ova pitanja su:

- Šta sam završio juče,
- Šta radim danas, i
- Da li me nešto blokira da nastavim.

Na sastancima se može prijaviti nešto što može blokirati tim da ne završi sve ciljeve sprinta. Sastanci su kratki, traju do 15 minuta, održavaju se uvek u isto vreme i na istom mestu, svakog dana.

U okviru sprinta se završavaju svi zahtevi predviđeni za taj sprint, testiraju i ukoliko je moguće isporučuju klijentu. Na ovaj način, klijent je od veoma rane faze razvoja projekta upoznat sa proizvodom, i može dati komentare i primedbe, čime se efikasno izbegava da projekat odluta u pogrešnom smeru.

Kanban, sa druge strane, se fokusira na vizuelizaciju i optimizaciju zadataka, pomoću signalnih kartica. Kanban tabla prikazuje različita stanja i aktivnosti, na primer da li je neka funkcionalnost trenutno u razvoju ili fazi testiranja. Broj zadataka koji su u jednom stanju je ograničen i striktno se kontroliše. Primer Kanban table je na slici 9.16.

New	Analysis		Design		Development		QA	
	In Process	Done	In Process	Done	In Process	Done	In Process	Done
Feature	Feature	Feature	Feature	Feature	Feature	Feature	Feature	
Feature	Feature	Feature	Feature	Feature	Feature	Feature	Feature	
Feature	Feature	Feature	Feature	Feature	Feature	Feature	Feature	
	Feature		Feature		Feature	Feature	Feature	
					Feature	Feature		

Slika 9.16. Kanban tabla sa karticama

Pitanja

1. Šta je model razvoja softvera?
2. Koja je svrha modela razvoja softvera?
3. Koje su faze u modelima razvoja softvera?
4. Koji sve različiti modeli postoje?
5. Na šta se odnosi faza sakupljanja zahteva?
6. Zbog čega je bitna faza održavanja softvera?
7. Šta je Waterfall model?
8. Koje su razlike između Waterfall i V modela?
9. Šta je inkrementalni model?
10. Šta je spiralni model?
11. Šta je agilni model?
12. Koje su osnovne prednosti agilnog modela?
13. Šta su mane agilnog modela?
14. Šta je agilni manifest?
15. Koje su razlike između agilnog modela i tradicionalnog Waterfall modela?
16. Koja je pozicija QA u agilnom timu?
17. Šta je Scrum?
18. Koja je važnost dnevnog scrum sastanka?
19. Šta svaki član tima treba da obrazloži na dnevnom sastanku?
20. Kako se agilni model nosi sa čestim promenama zahteva sistema?

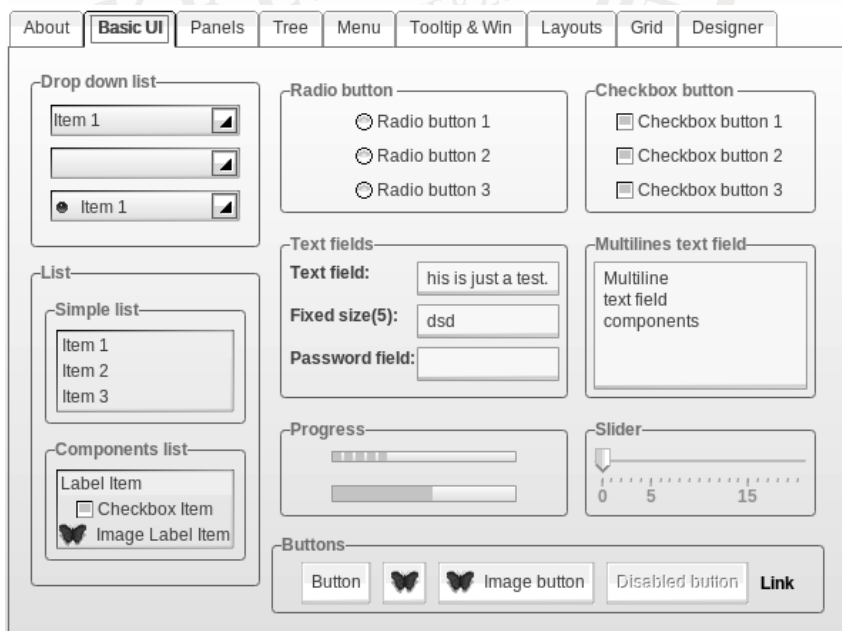
10. TESTIRANJE KORISNIČKIH INTERFEJSA

Računarske aplikacije mogu imati različite funkcionalnosti i primene, dok postoje samo dva osnovna tipa interfejsa:

- CLI - engl. *Command Line Interface*, komande se u obliku tekstualnih naredbi unose u komandnoj liniji, a računar odgovara na njih
- GUI – engl. *Graphical User Interface*, odnosi se na grafički korisnički interfejs koji predstavlja vizuelnu interakciju sa sistemom.

Sve moderne aplikacije imaju grafički korisnički interfejs, pa u nastavku ovog poglavlja CLI neće biti razmatran. Aplikacije mogu imati veoma raznoliku primenu, od aplikacija za zabavu i informativnih aplikacija, sve do sistema koji su kritični po pitanju sigurnosti i od kojih zavise ljudski životi. Zajedničko za sve ove različite tipove aplikacija je interakcija sa korisnikom, za koju je zadužen GUI. Dodatni nivo kompleksnosti dolazi od raznolikosti uređaja na kojima se aplikacija izvršava, od klasičnih desktop aplikacija, preko mobilnih uređaja, do veb aplikacija. U svim ovim slučajevima, korisnik vrši interakciju sa GUI, koji dalje komunicira sa ostatkom aplikacije.

GUI je sistem koji je baziran na događajima (engl. *event-based*). To znači da reaguje na odgovarajuće događaje koje generiše korisnik, kao na primer dodir prstom na ekran osjetljiv na dodir, klik mišem, kucanje na tastaturi i slično. GUI sadrži elemente preko kojih korisnik može da vrši interakciju sa sistemom, poput radio dugmadi, *checkbox* elemenata, tekstualnih polja, ikonica, menija itd. Neki od ovih elemenata su prikazani na slici 10.1.



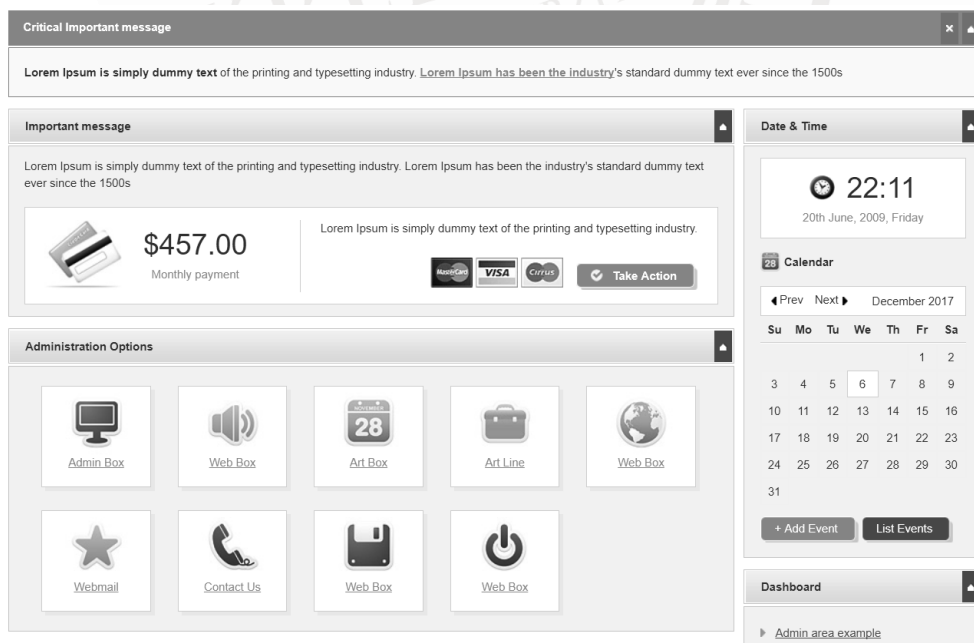
Slika 10.1. Elementi grafičkog korisničkog interfejsa

GUI je vidljivi deo aplikacije, pošto korisnik ne vidi izvorni kod. Sve funkcionalnosti sistema se mogu pokrenuti pomoću njega. Testiranje grafičkog interfejsa je od izuzetnog značaja, pošto predstavlja ulaznu tačku u aplikaciju. Neophodno je da bude otporan i robustan, i da ne prihvata ulaze koji nisu legalni, pošto u opštem slučaju korisniku može svašta da padne na pamet i da unese potpuno neprihvatljiv unos u neko polje. Osim toga, maliciozni korisnik može pokušati da napadne sistem preko korisničkog interfejsa, unošenjem specifične sekvence karaktera koja može izazvati nepredviđeno izvršavanje koda. Jedan od ovakvih primera je SQL injekcija.

U opštem slučaju, procenjuje se da je preko 50% koda aplikacije, kao i 50% vremena u razvoju sistema posvećeno korisničkom interfejsu. Otprilike 50% testiranja takođe odlazi na ovu aktivnost.

10.1. Proces testiranja

GUI testiranje je proces testiranja grafičkog interfejsa aplikacije koja se testira, i uključuje proveru svih ekrana aplikacije, zajedno sa kontrolama poput menija, dugmadi, ikonica, svih tipova barova (toolbar, meni bar i slično), svih dijaloga aplikacije itd. Naročito je potrebno obratiti pažnju da je GUI jedini deo aplikacije koji korisnik vidi. On ne vidi izvorni kod, implementaciju objekata, interfejs ka bazi podataka i slično. Neophodno je obratiti pažnju i na dizajn, strukturu, slike i ostale elemente interfejsa, jer ružan dizajn može odbiti korisnike.



Slika 10.2. Primer korisničkog interfejsa veb aplikacije

Slika 10.2 prikazuje jedan korisnički interfejs veb aplikacije. Jedna od prvih stvari koje bi trebalo proveriti je da li su slike vidljive u svim veb čitačima, a zatim da li su svi linkovi dostupni, da li dugmići rade kad se klikne na njih i slično. Dalje, ukoliko korisnik promeni veličinu ekrana veb čitača, slike ili sadržaj ne bi trebalo da budu isečeni ili da se preklape.

Često se postavlja pitanje, ukoliko se funkcionalnost i logika aplikacije već detaljno testiraju, da li je zaista neophodno gubiti toliko vreme na GUI (kao što je već navedeno, oko 50 posto vremena testiranja ide na GUI)? Potrebno je razmišljati kao korisnik aplikacije, a ne kao tester. Korisnik nema dubinsko znanje o aplikaciji (kao tester na primer), već najčešće na osnovu korisničkog interfejsa odlučuje da li će koristiti aplikaciju ili ne. Normalni korisnik prvo posmatra dizajn i izgled aplikacije (engl. *look and feel*), i koliko je korisnički interfejs jednostavan za razumevanje. Ukoliko interfejs nije dovoljno prilagođen korisniku, ili korisnik smatra da je aplikacija teška za razumevanje / korišćenje, on sigurno neće odabrati tu aplikaciju za dalje korišćenje. GUI je dakle jedna od ključnih stavki, i mora biti adekvatno testirana kako bi se obezbedilo da nema ozbiljnih defekata.

Najčešće je potrebno proveriti:

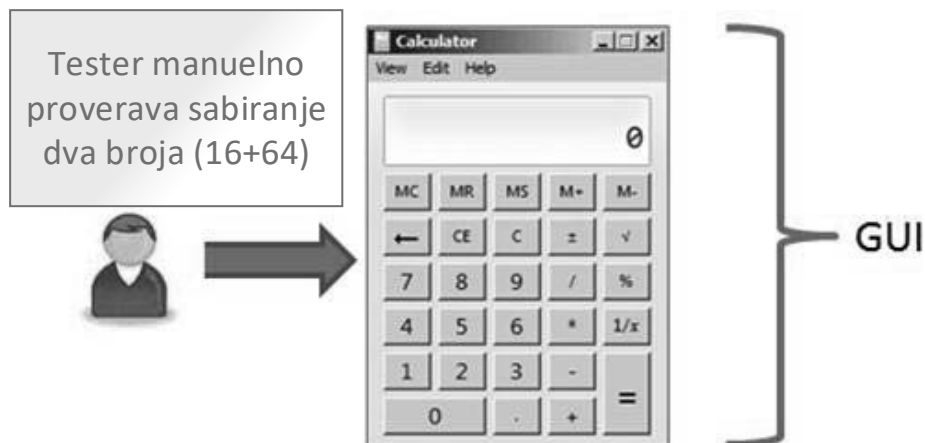
- Da li su svi elementi odgovarajuće veličine, pozicije, za unos proveriti prihvatanje karaktera i brojeva
- Da li se predviđena funkcionalnost može postići korišćenjem GUI-a
- Da li se poruke o grešci ispravno prikazuju
- Da li su sekcije ekrana jasno podeljene
- Da li je korišćen odgovarajući font, odgovarajuće formatiranje teksta, korišćene boje, ukupan estetski utisak
- Da li su slike odgovarajućeg kvaliteta
- Kako se GUI ponaša pri promeni rezolucije

Testiranje se može sprovesti na jedan od tri načina:

- Manuelno testiranje
- Automatsko testiranje
- Model stanja

10.1.1. Manuelno testiranje

Kod ovog pristupa, grafički ekrani aplikacije se proveravaju manuelno od strane testera, i porede sa zahtevima iz specifikacije. Proveravaju se kako elementi interfejsa, tako i tok između ekrana – da li su odgovarajući prelasci između ekrana implementirani kako je specificirano u zahtevima. Proveravaju se i najčešći korisnički scenariji. Na slici 10.3 prikazan je manuelni način testiranja korisničke aplikacije koja ima funkcionalnost digitrona. Tester unosi odgovarajuće ulazne podatke, i proverava da li je izlaz sistema ispravan.



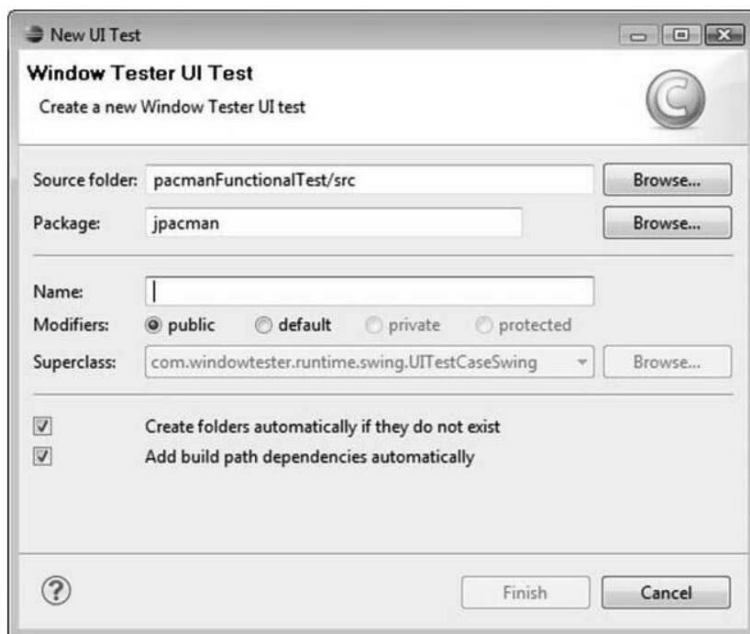
Slika 10.3. Manuelno testiranje korisničkog interfejsa

Manuelno testiranje grafičkog interfejsa je najčešći tip ovog oblika testiranja. Postoje naravno i nedostaci – u velikoj meri zavisi od sposobnosti i iskustva testera, i od njegovog poznavanja same aplikacije. Može biti veoma monotono, dosadno i frustrirajuće, a samim tim i podložno ljudskim greškama. Često je potrebno previše napora za izvršavanje i analizu rezultata testa, naročito u slučaju regresionog testiranja nakon izmene koda.

Manuelno testiranje tipično uključuje sledeće scenarije:

- Testiranje veličine, pozicije, širine i visine elemenata korisničkog interfejsa
- Testiranje da li se odgovarajuće poruke o greškama ispravno prikazuju
- Testiranje različitih sekcija ekrana
- Testiranje da li je upotrebljeni font čitak ili ne
- Testiranje da li je boja fonta odgovarajuća
- Testiranje ekrana u različitim rezolucijama i zumiranje
- Testiranje da li su elementi odgovarajuće pozicionirani u odnosu na druge elemente, poput ikonice, teksta, slika i slično
- Testiranje da li su slike odgovarajućeg kvaliteta
- Testiranje da li se korisnik oseća prijatno prilikom upotrebe korisničkog interfejsa, odnosno da li je interfejs intuitivan
- Testiranje da li je interfejs dovoljno atraktivan ili ne
- Testiranje scrollbar elemenata u odnosu na veličinu stranice
- Testiranje elemenata koja treba da budu onemogućeni u nekim scenarijima
- Testiranje korisničkog unosa u sve elemente interfejsa
- Testiranje da li svi linkovi rade

Kao primer se posmatra korisnički interfejs Eclipse okruženja na slici 10.4. Potrebno je napisati testove za korisnički ekran.



Slika 10.4. Korisnički interfejs Eclipse okruženja

Testovi koji bi pokrili ovaj korisnički interfejs i testirali njegovu upotrebljivost su dati u nastavku.

Test 01: Verifikacija da je polje za unos teksta sa labelom Source folder poravnato na odgovarajući način.

Test 02: Verifikacija da je polje za unos teksta sa labelom Package poravnato na odgovarajući način.

Test 03: Verifikacija da postoji dugme sa labelom Browse pored polja za unos teksta Source folder.

Test 04: Verifikacija da se pritiskom na dugme Browse otvara odgovarajući prozor sa fajl sistemom za odabir foldera.

Test 05: Verifikacija da postoji dugme sa labelom Browse pored polja za unos teksta Package.

Test 06: Verifikacija da se pritiskom na dugme Browse otvara odgovarajući prozor sa fajl sistemom za odabir paketa.

Test 07: Verifikacija da je polje za unos teksta sa labelom Name poravnato na odgovarajući način.

Test 08: Verifikacija da pod labelom Modifiers postoji tačno četiri radio dugmeta, sa imenima public, default, private i protected, koji su poravnati na odgovarajući način.

Test 09: Verifikacija da se pod labelom Superclass koja se nalazi ispod labele Modifiers nalazi dropdown lista koja sadrži odgovarajuće elemente i koji se mogu selektovati na ispravan način.

Test 10: Verifikacija da se pod labelom Superclass nalazi dugme Browse, koje je poravnato na odgovarajući način.

Test 11: Verifikacija da se pritiskom na dugme Browse pod labelom Superclass otvara odgovarajući prozor sa fajl sistemom za odabir klase.

Test 12: Verifikacija da korisnik ne može da kuca karaktere u polje dropdown pod labelom Superclass.

Test 13: Verifikacija da se odgovarajuća poruka o grešci prikazuje u slučaju da je nešto pogrešno odabrano.

Test 14: Verifikacija da se greška generiše u crvenoj boji.

Test 15: Verifikacija da se može odabrati tačno jedno radio dugme u svakom trenutku.

Test 16: Verifikacija da TAB dugme radi na ispravan način, i da se pritiskom na to dugme selektuje sledeći element grafičkog interfejsa.

Test 17: Verifikacija da su checkbox elementi ispravno poravnati.

Test 18: Verifikacija da se može selektovati više od jednog checkbox elementa u isto vreme.

Test 19: Verifikacija da su svi elementi na celom prozoru ispravno poravnati.

Test 20: Verifikacija da se nakon podešavanja svih opcija i pritiska na dugme Finish prikazuje odgovarajuća potvrdna poruka korisniku.

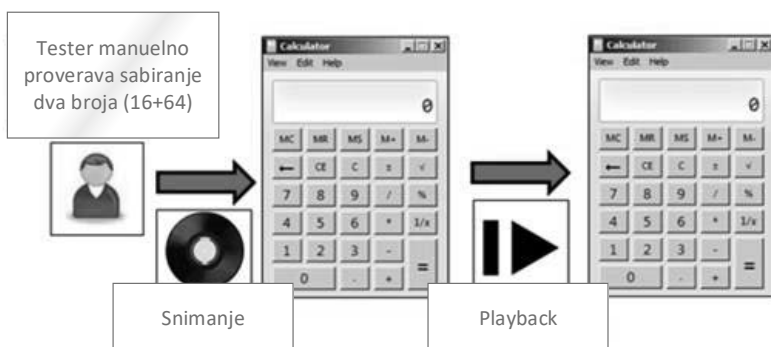
Moguće je naravno generisati još testova za ovaj korisnički interfejs. Zapravo, potpun skup testova bi uključivao i različite kombinacije selektovanih elemenata, različite tekstualne unose (računajući i nelegalne) i slično.

Pod manuelnim testiranjem se vodi i slučajno (slobodno) testiranje od strane krajnjih korisnika sistema. Na primer, beta verzija aplikacije se često ponudi odabranoj grupi korisnika na testiranje, najčešće nekoliko nedelja pre zvanične isporuke softvera.

10.1.2. Automatsko testiranje

Testiranje GUI se može izvršiti i pomoću alata za automatizaciju, metodom koja je poznata pod engleskim nazivom *Record and Playback*. Kao što i ime metode govori, postoje dve faze, faza snimanja (engl. *record*) i faza ponovnog puštanja (engl. *playback*). Za vreme record faze, testovi se snimaju pomoću alata, dok se za vreme playback faze snimljeni koraci izvršavaju nad aplikacijom koja se testira (slika 10.5).

Svi alati imaju režim rada snimanja, u kojem se snimaju sve akcije koje tester izvršava nad aplikacijom, i čuvaju u odgovarajućoj test skripti. Test skripta se konstruiše tako što se snima kompletna interakcija testera sa aplikacijom, svaki događaj koji tester generiše nad GUI se beleži radi kasnije automatske reprodukcije. Beleže se pokreti i klikovi miša, unos sa tastature, interakcija sa ekranom osjetljivim na dodir i slično.



Slika 10.5. Automatsko testiranje korisničkog interfejsa

Veliki broj alata za automatizaciju snima test u nekog od jezika za skriptovanje, kako bi se testovi mogli ručno modifikovati, optimizovati i održavati. Na primer, može se snimiti osnovni scenario, a kasnije manuelno dodati parametre za različite podatke, ubaciti dodatne verifikacije i slično.

Prednosti alata za automatsko testiranje su brojne, od kojih su najvažnije:

- Upotrebom oblika assert naredbi, može se verifikovati da li je neki element grafičkog interfejsa prisutan ili ne, da li je određen element selektovan ili ima određenu vrednost, čak i utvrditi da li se na primer stranica dovoljno brzo učitava.
- Često postoje dobre mogućnosti posmatranja kompleksnijih izlaza, poput prepoznavanja znakova (OCR - engl. *Optical Character Recognition*) i tehnike obrade slike.
- Veoma su korisni za regresiono testiranje, pošto je ovaj vid testiranja najmonotoniji i najdosadniji, i stoga podložan greškama.
- Ovi alati su upotrebljivi i u drugim različitim kontekstima, kao što su: demonstracije, daljinska podrška, analiza ponašanja korisnika i edukacija korisnika.
- Playback se izvršava znatno brže od manuelnog rada sa aplikacijom.

Alati imaju i svoje nedostatke:

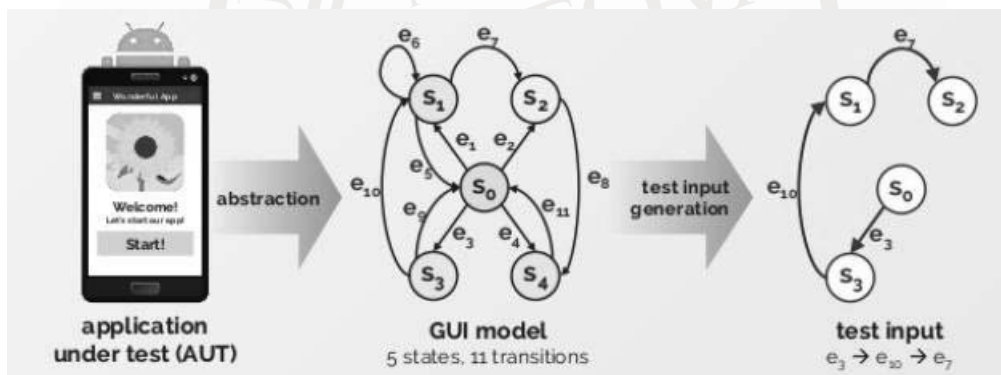
- Testiranje se obavlja relativno kasno u procesu razvoja jer se alati mogu upotrebiti jedino kada je korisnički interfejs kompletno ili barem delimično implementiran.

- Alati mogu da testiraju samo ono što je već implementirani i što je manuelno verifikovano da radi.
- Svaka promena u implementaciji često može dovesti do toga da se sve skripte koje se odnose na izmenjeni deo koda moraju snimiti iz početka ili makar manuelno ažurirati.
- Svaka promena u korisničkom interfejsu može poremetiti rad skripti, poput dodavanja novih elemenata u interfejs ili uklanjanja postojećih, promene rezolucije, promene rasporeda elemenata i slično.

Najpopularniji alat za automatizaciju testiranja korisničkog interfejsa je Selenium, koji je prikazan u nastavku.

10.1.3. Model stanja

Korisnički interfejs se može testirati i na osnovu modela stanja, kao što je detaljno opisano u poglavlju 3.4. Preko modela stanja se može grafički opisati ponašanje sistema, a samim tim i grafičkog korisničkog interfejsa, koji je prednji kraj softverskog sistema. Model stanja je model konačkog automata, u kome su slike ekrana stanja, a prelazi između stanja su događaji koji menjaju stanje, kao što je prikazano na slici 10.6.



Slika 10.6. Testiranje korisničkog interfejsa modelom stanja

Model stanja se kreira na osnovu specifikacije softverskog sistema. Testiranje se zasniva na kreiranju svih legalnih sekvenci prelaza iz stanja u stanje, koji se pokrivaju testovima. Neophodno je verifikovati i nelegalne prelaze između stanja, odnosno prelazak iz jednog ekrana u drugi koji ne bi trebalo da bude dozvoljen na osnovu specifikacije sistema.

10.2. Izazovi u testiranju grafičkog interfejsa veb aplikacija

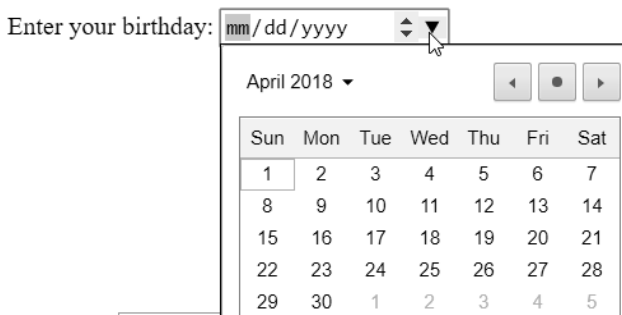
Grafički korisnički interfejs veb aplikacija se mora testirati sa izuzetnom pažnjom. Veb aplikacije, zajedno sa aplikacijama za mobilne uređaje, su najbrže rastuće softversko tržište danas. Sa druge strane, veb aplikacije nose veliki broj izazova za proces testiranja, pošto se sastoje od većeg broja različitih tehnologija koje se koriste za implementaciju *backend* i *frontend* strana. Frontend se obično bazira na skupu tehnologija poput HTML, CSS i JavaScript, koji su potrebni kako bi se napravio interaktivni korisnički interfejs. Backend se takođe može implementirati na veliki broj različitih načina, na primer PHP zajedno sa MS SQL bazom, ili ASP .NET C# aplikacija, ili Java aplikacija u obliku JSP. Potrebno je da proces testiranja bude dovoljno fleksibilan kako bi se prilagodio dinamičkoj prirodi veb aplikacija.

Dva ključna aspekta testiranja su mogućnost da se posmatra i kontroliše ponašanje aplikacije. Pod posmatranjem se podrazumeva vidljivost rezultata testa na ekranu. U opštem slučaju, vidljivost veb aplikacija nije na visokom nivou. Dodatno, veb aplikacije imaju i nizak nivo mogućnosti za kontrolu, zbog većeg broja različitih serverskih komponenti koje su pisane u različitim programskim jezicima, i koje zajedno rade kako bi proizvele korisnički interfejs u obliku HTML.

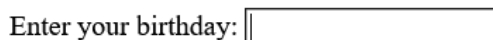
Generisanje HTML na serveru koji treba da se prikaže korisniku nije kraj problema. Veb čitač mora da interpretira dobijeni HTML kod i da prikaže stranicu korisniku. Međutim, različiti veb čitači mogu u opštem slučaju da različito prikažu isti HTML kod. Kao primer, može da se posmatra ponašanje komponenti veb stranice u različitim veb čitačima, pogotovo u slučaju da je stanica pisana u HTML 5. HTML 5 je najnovija verzija HTML standarda, koja je dostupna već godinama, ali je još uvek poznato da ima probleme i ograničenja na različitim veb čitačima. Najveći broj problema se povezuje sa Microsoft Internet Explorer ili Edge veb čitačima, međutim problemi takođe postoje i sa starijim verzijama drugih veb čitača. Neka se posmatra standardni HTML 5 kod za element forme u obliku kalendara. U velikom broju veb aplikacija je potrebno dobiti informaciju o datumu rođenja korisnika, koji se u HTML 5 može implementirati na sledeći način:

```
<form action="/action_page.php" method="post">
Enter your birthday: <input type="date" name="bd1">
</form>
```

Ovaj element se prikazuje na odgovarajući način u Google Chrome, Mozilla Firefox i Microsoft Edge veb čitačima (slika 10.7). Sa druge strane, čak i najnovija verzija Internet Explorer veb čitača ne može na ispravan način da prikaže ovaj element, već ga generiše u obliku prostog polja za unos teksta (slika 10.8).



Slika 10.7. HTML 5 element kalendar prikazan u Chrome veb čitaču



Slika 10.8. HTML 5 element kalendar prikazan u Internet Explorer v11

U ovom slučaju, ukoliko je bilo potrebno obezbediti isto ponašanje strane na svim veb čitačima, morao bi se koristiti neki drugi pristup. Što je još gore, CSS i JavaScript se takođe ne ponašaju identično na različitim veb čitačima. Uopšteno gledano, najveći broj problema sa različitim čitačima potiče od modernijih funkcionalnosti koje su uvedene u najnovijim verzijama HTML i CSS, a koje neki veb čitači ne mogu da prikažu na odgovarajući način.

Ovo je zapravo veoma čest problem, naročito u slučajevima kada veb aplikacija ima zahtev da mora da podrži i starije verzije veb čitača (opet, najviše problema postoji sa starijim verzijama Internet Explorer). Ovde se mora napomenuti da najveći deo osnovnih HTML i CSS funkcionalnosti, poput osnovnih HTML elemenata, osnovnih CSS boja i stilova teksta generalno radi dobro sa svim veb čitačima koji su dostupni na tržištu. Problemi nastaju kada se koriste novije HTML 5 i CSS 3 funkcionalnosti, poput Flexbox, video / audio podrške ili CSS grid elemenata. Iako za elemente poput `<canvas>`, `<video>` ili `<audio>` postoje prirodni mehanizmi za oporavak (engl. *fallback*), ništa ne garantuje da je programer zaista implementirao ove mehanizme. Primer implementirane fallback metode za HTML video element je dat na slici 10.9. U slučaju da veb čitač (na primer starija verzija Internet Explorer) ne podržava tu funkcionalnost korisniku se nudi Flash plejer, a ukoliko čak ni to ne radi, nudi se običan download, tako da korisnik ipak može da vidi sadržaj. Ovakve fallback metode su moguće i u CSS – ukoliko veb čitač naiđe na pravilo koje ne može da razume, samo će ga preskočiti i neće ga primeniti. Moguće je dodati uslovne komentare unutar HTML sintakse, pomoću kojih se selektivno izvršavaju različita pravila na različite čitače (slika 10.10, definiše se specifično CSS pravilo u slučaju da je veb čitač internet Explorer 8 ili stariji). Zbog toga je, u opštem slučaju, neophodno sprovesti detaljno testiranje za sve veb čitače koje aplikacija treba da podrži prema zahtevima, sa naročitim akcentom na Internet Explorer, i verifikovati da se sve stranice i njihove komponente zaista prikazuju na odgovarajući način.

```

<video id="video" controls preload="metadata"
poster="img/poster.jpg">
  <source src="video/demo_clip.mp4" type="video/mp4">
  <source src="video/demo_clip.webm" type="video/webm">
  <source src="video/demo_clip.ogv" type="video/ogg">
  <!-- Flash fallback -->
  <object type="application/x-shockwave-flash"
    data="flash-player.swf?videoUrl=video/demo_clip.mp4"
    width="1024" height="576">
    <param name="movie"
      value="flash-player.swf?videoUrl=video/demo_clip.mp4" />
    <param name="allowfullscreen" value="true" />
    <param name="wmode" value="transparent" />
    <param name="flashvars" value="controlbar=over&
      image=img/poster.jpg&
      file=flash-player.swf?videoUrl=video/demo_clip.mp4" />
    
  </object>
  <!-- Offer download -->
  <a href="video/demo_clip.mp4">Download MP4</a>
</video>

```

Slika 10.9. Implementacija fallback metode za <video> element

```

<!--[if lte IE 8]>
<link rel="stylesheet" type="text/css" href="ie-css.css" />
<![endif]>

```

Slika 10.10. Uslovni komentari za rukovanje različitim verzijama čitača

Sigurnost i robusnost veb aplikacije su još jedan aspekt od izuzetnog značaja za testiranje. Svaki korisnički interfejs mora obavezno imati implementiranu validaciju korisničkog unosa. Validacija korisničkog unosa se odnosi na proveru da li su vrednosti koje je korisnik uneo u formama validne i bezbedne. Na primer, maliciozni korisnik može da pokuša da ubaci SQL injekciju unutar forme, ciljajući bazu podataka koja se nalazi na serveru. SQL injekcija pokušava da iskoristi ranjivosti unutar validacije korisničkog unosa, poput lošeg filtriranja *escape* karaktera, i izaziva neočekivano izvršavanje koda. Ukoliko se posmatra SQL upit koja dohvata rekord sa datim korisničkim imenom iz tabele korisnika:

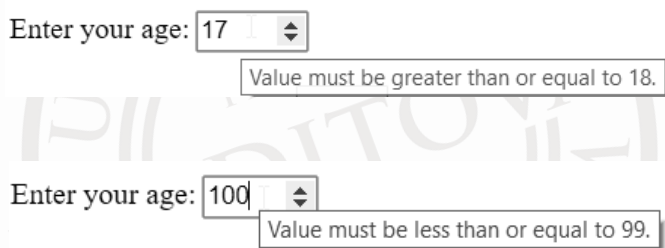
```
statement = "SELECT * FROM users WHERE name = " + userName + "';"
```

Ukoliko se sa klijentske strane aplikacije korisničko ime unese u posebnom obliku u formu za unos, maliciozni korisnik može da pokuša da izvrši SQL kod na način na koji autor koda nije planirao. U slučaju da se promenljiva `userName` postavi na vrednost `' or '1'='1'`, upit koji će doći do baze će biti u obliku:

```
SELECT * FROM users WHERE name = '' OR '1'='1';
```

Pošto je deo izraza 'l'='l' uvek tačan, ovaj upit će zapravo selektovati sva validna korisnička imena iz baze podataka. Zbog toga je validacija korisničkog unosa neophodna, i mora se raditi na JavaScript nivou kao deo sanitizacije unosa, zajedno sa dodatnim proverama na backend strani, pre okidanja upita nad bazom podataka.

Osim navedenih specifičnosti, za testiranje korisničkog interfejsa veb aplikacija se često koriste tradicionalne metode crne kutije, poput podele na klase ekvivalencije i analize graničnih vrednosti. Ove dve tehnike se često koriste u proveru korisničkog unosa. Neka se posmatra polje za unos gde se očekuje da korisnik unese svoje godine, i neka po specifikaciji zahteva ova vrednost mora biti između 18 i 99 godina. Mogu se uočiti tri klase ekvivalencije, jedna legalna (godine između 18 i 99), kao i dve nelegalne (godine < 18 i godine > 99). Pošto se obično klase ekvivalencije testiraju zajedno sa graničnim vrednostima (mesto gde će programeri najčešće napraviti grešku), tester bi morao da proveru ponašanje aplikacije za sledeće unete vrednosti: 17, 18, 19, 98, 99, 100. Tester mora da verifikuje da aplikacija prihvata ispravne vrednosti, a da odbija da prihvati vrednosti van dozvoljenog opsega (17 i 100) uz odgovarajuću poruku o grešci, kao što je prikazano na slici 10.11.



Slika 10.11. Provera graničnih vrednosti za korisnički unos

Dodatno, neophodno je proveriti kako se aplikacija ponaša ukoliko se u polje za unos unese neka potpuno neočekivana vrednost, poput unosa nasumičnog teksta u polje koje prema specifikaciji treba da prihvati samo brojeve. Na taj način se proverava da li je validacija unosa urađena prema specifikaciji, jer ukoliko u ovom slučaju validacija omane i propusti da se uneta vrednost propagira do servera, vrlo verovatno će doći do greške na serveru, a u najgorem slučaju uneta vrednost će se propagirati do baze gde može izazvati ozbiljniju grešku. Generalno, naročito kada su u pitanju tekstualna polja za unos, potrebno je proveriti ponašanje aplikacije za što je moguće veći broj nevalidnih unosa:

- Očekuju se brojevi, a korisnik unese tekst
- Predugačak tekstualni unos (na primer kopiranje celog teksta Hamleta u tekstualno polje za unos)
- Specijalni karakteri, poput karaktera koji čine neki smajli ;)
- Simboli iz nekog drugog jezika, na primer kineski karakteri

Ova validacija je delimično podržana od samog HTML 5, ali svaka ozbiljnija validacija i sanitizacija unosa podrazumeva upotrebu JavaScript i regularnih izraza, uz dodatne provere na serveru pre stvarnog slanja upita ka bazi. Svako polje za unos se mora testirati i proveriti da li je moguće ubaciti neki maliciozni unos poput SQL injekcije, koji mogu dovesti do ozbiljnih problema sa bazom. To naročito važi za aplikacije koje rade sa osetljivim podacima, pošto bi takva greška mogla dovesti do curenja privatnih podataka ili finansijskih gubitaka.

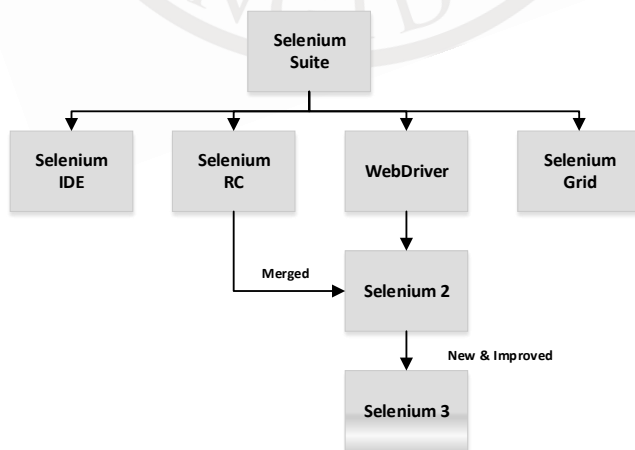
10.3. Selenium alat za testiranje korisničkog interfejsa

Selenium je besplatni, open source alat za automatsko testiranje veb aplikacija, sa podrškom za različite veb čitače i platforme. Kao i kod drugih alata za automatizaciju testiranja, Selenium skripte koje se pišu imaju za cilj da obavljaju sve akcije koje bi korisnik obavljao ručno. Selenium može obavljati razne vrste automatske interakcije, ali primarna namena i razlog zbog kog je nastao je automatsko testiranje veb aplikacija. Testiranje upotrebom alata Selenium se često i naziva Selenium testiranje.

Selenium nije samo jedan alat, već zapravo kolekcija alata, gde svaki ima svoj cilj i način primene (slika 10.12). Postoje četiri osnovne komponente;

- Selenium Integrated Development Environment (IDE)
- Selenium Remote Control (RC)
- WebDriver
- Selenium Grid

Za ime Selenium je vezana interesantna priča. U toku razvoja, postojalo je drugo okruženje za automatsko testiranje, pod okriljem firme Mercury Interactive (posle preuzeta od strane HP). Pošto je selen poznati protivotrov za trovanje živom (engl. *mercury*), novo okruženje je dobilo naziv Selenium.



Slika 10.12. Struktura Selenium alata

Selenium klijentski API pruža mogućnost pisanja test skripti u drugim programskim jezicima. Testovi pisani u drugim programskim jezicima mogu komunicirati sa Selenium-om tako što će pozivati metode Selenium API-ja. Izlaskom Selenium 2, predstavljen je i novi klijent API sa WebDriver modulom kao centralnom komponentom. Prvobitni API koji koristi klasu Selenium je i dalje podržan.

Selenium Remote Control je alat koji omogućava pisanje automatizovanih test skripti u bilo kom programskom jeziku. Funkcioniše po principu ubacivanja (injekcije) JavaScript koda u veb čitaču. Nedostatak ovakvog pristupa je to što se JavaScript ne ponaša isto na svim veb čitačima, kao i činjenica da mnogi pretraživači imaju restriktivnu politiku za injekciju JavaScript koda (smatra se sa sigurnosni propust). Izlaskom Selenium 2 prestala je podrška za Selenium RC. Nova komponenta koja treba da zameni Selenium RC je Selenium WebDriver.

Selenium Grid je server koji omogućava pokretanje testova na instancama browsera na udaljenim računarima. U ovom slučaju jedan server se ponaša kao čvor (engl. *hub*) kome se testovi obraćaju da bi dobili pristup instancama veb čitača. Najveća prednost Selenium Grid alata je u tome što omogućava paralelno izvršavanje testova na više različitih veb čitača i više različitih verzija istih veb čitača, na različitim platformama i operativnim sistemima. Na ovaj način se pokriva više uslova testiranja, uz jednostavnije centralizovano upravljanje. U kombinaciji sa Selenium WebDriver-om, Selenium Grid drastično smanjuje vreme izvršavanja testova

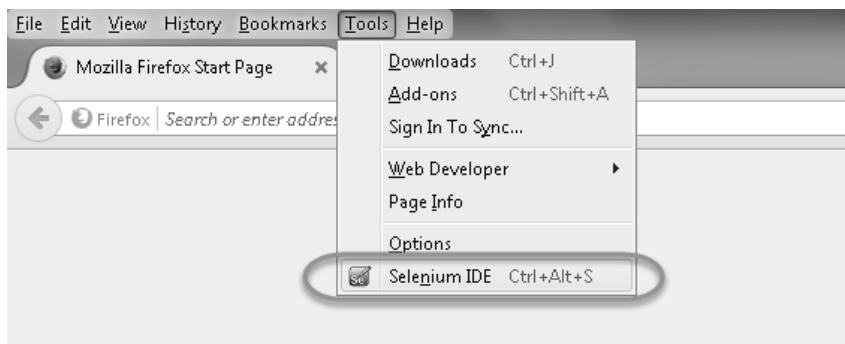
Dve najbitnije komponente, Selenium IDE i WebDriver, će biti opisane detaljno u nastavku ovog poglavlja. One su i najviše zastupljene u testiranju, sa napomenom da u trenutku pisanja ove knjige Selenium IDE nije više bio podržan na Firefox verzijama posle verzije 55. WebDriver, sa druge strane, je već duže vreme osnova za pisanje automatskih testova za veb aplikacije, i poznavanje ove tehnologije je veoma cenjena osobina za moderne testere.

10.3.1. Selenium IDE

Selenium IDE predstavlja softversko okruženje koje omogućava generisanje automatizovanih test skripti. Implementira se kao dodatak (engl. *plug-in*) za veb čitač Mozilla Firefox i radi po principu snimanja i reprodukcije. U početku, ova komponenta je bila poznata i pod nazivom Selenium Recorder.

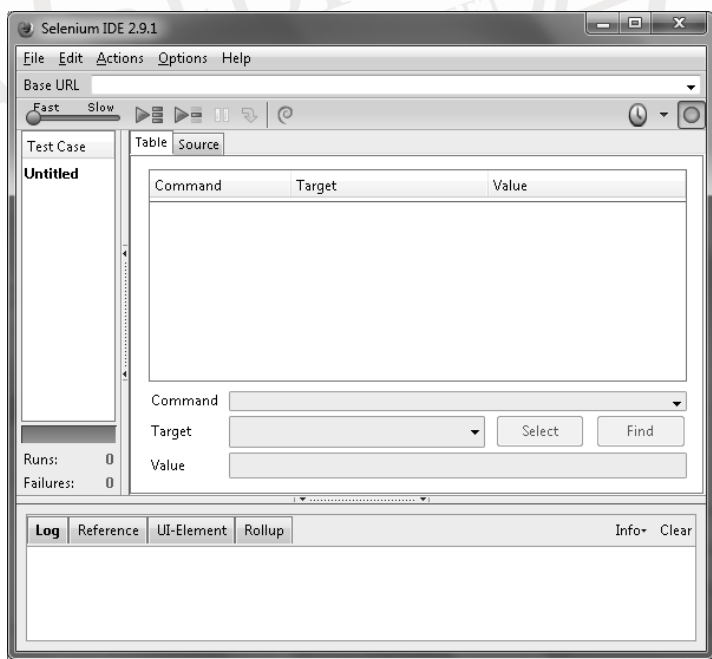
Test skripte koje se generišu čuvaju se u Selense obliku, koji predstavlja poseban skript jezik za Selenium. Selense pruža komande za izvršavanje akcija u veb čitaču, kao i za preuzimanje podataka sa stranica koje su dobijene kao rezultat akcija. Skripte se mogu automatski snimiti, a naknadno i manuelno modifikovati. Selenium IDE je odličan za početno upoznavanje sa Selenium alatima, uz primedbu da je podrška za ovaj alat trenutno obustavljena zaključno sa verzijom 55 Mozilla Firefox veb čitača. U praksi se više koristi Selenium WebDriver koji nudi znatno više mogućnosti.

Zbog svoje jednostavnosti, Selenium IDE se koristi samo kao alat za pravljenje prototipa. Ukoliko je potrebno napraviti ozbiljnije testove, mora se koristiti Selenium WebDriver. Selenium ide je dostupan sa adrese <https://addons.mozilla.org/en-us/firefox/addon/selenium-ide/>. Nakon instalacije, može se pronaći unutar *Tools* sekcije Firefox pretraživača, kao što je prikazano na slici 10.13.



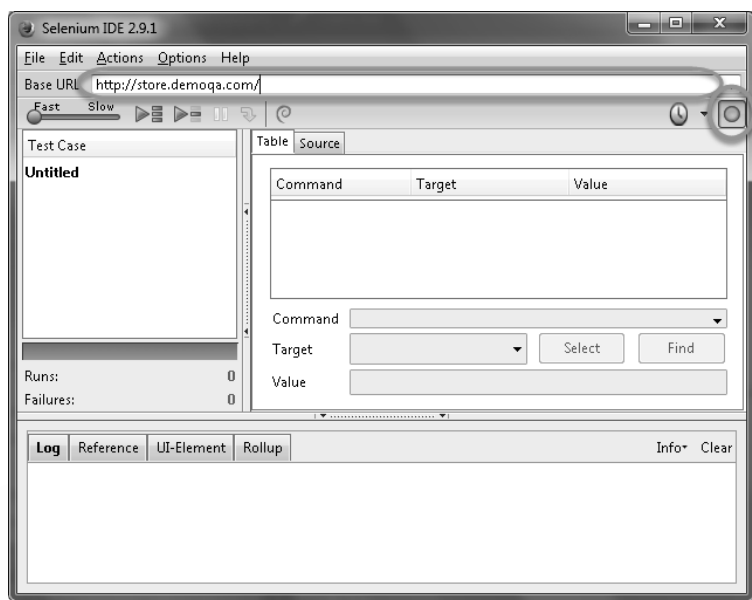
Slika 10.13. Selenium IDE nakon instalacije u Firefox

Nakon pokretanja, dobija se osnovni ekran Selenium IDE alata, kao što je prikazano na slici 10.14. Naravno, Firefox mora sve vreme biti uključen u pozadini, pošto alat funkcioniše kao plug-in.



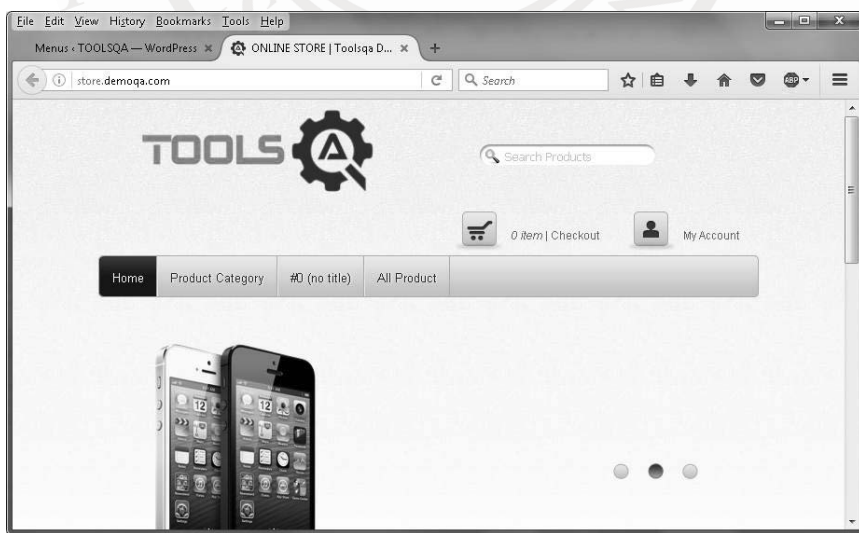
Slika 10.14. Osnovni ekran Selenium IDE alata

Snimanje testa se započinje tako što se unese adresa veb stranice koju je potrebno testirati i klikne na dugme Record, kao na slici 10.15.



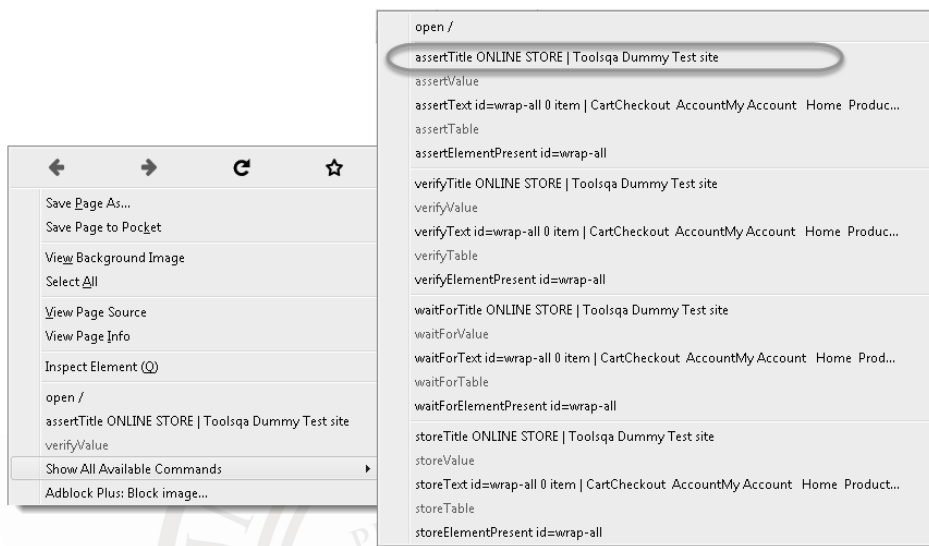
Slika 10.15. Zapocinjanie snimanja testa u Selenium IDE alatu

Nakon klika na dugme record, sve dalje akcije korisnika u okviru Firefox okruzenja se snimaju u test skriptu. Sledeci korak jeste da se u Firefox citacu ode na veb stranicu koju je potrebno testirati, kao što je prikazano na slici 10.16.



Slika 10.16. Odlazak na veb stranicu koju je potrebno testirati

Desni klik na praznom prostoru na ekranu će pozvati Selenium kontekst meni. Klikom na *Show All Available Commands* opciju dobija se spisak dostupnih komandi. Na primeru sa slike 10.16, ukoliko se odabere **assertTitle ONLINE STORE | Toolsqa Dummy Test site** kao na slici 10.17, u skriptu će biti ubačena provera da li naslov stranice tačan.

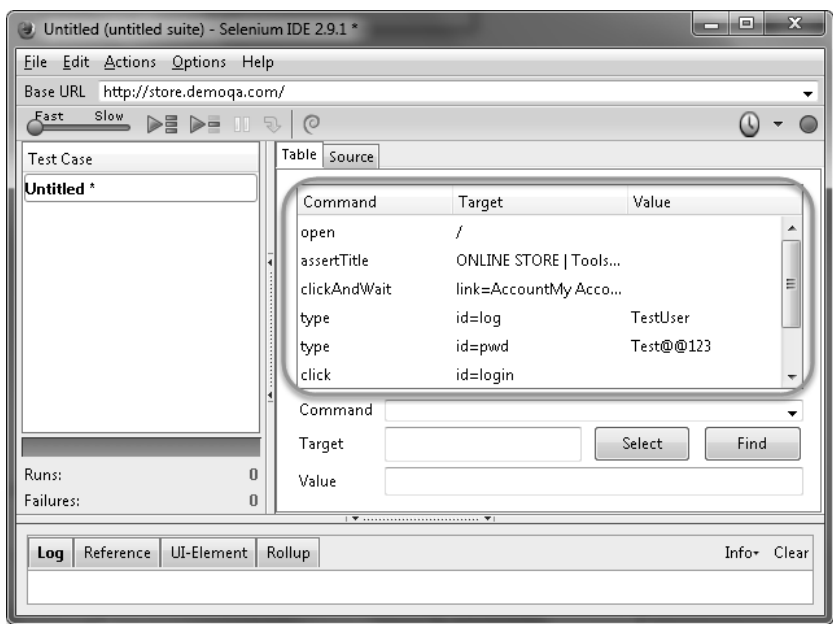


Slika 10.17. Dodavanje komandi iz konteksnog menija

Ukoliko se odabere My Account link, moguće je testirati unos korisničkog imena i lozinke. Moguće je na primer uneti *TestUser* kao korisničko ime, i *Test123* kao lozinku i kliknuti na login. Ovi kredencijali nisu ispravni, i aplikacija će vratiti poruku o grešci, kao na slici 10.18.

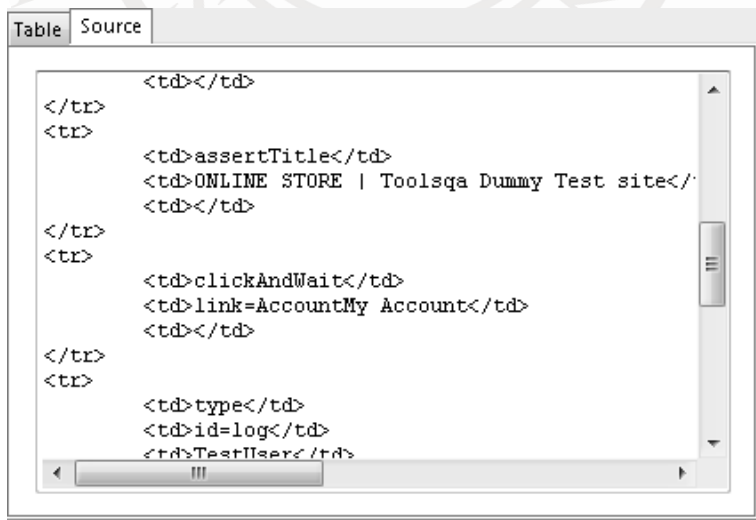
Slika 10.18. Testiranje unosa neispravnog korisničkog imena i lozinke

Snimanje se prekida ponovnim klikom na Record dugme. U tabu pod nazivom Table mogu se videti sve komande koje su snimljene, kao što je prikazano na slici 10.19.

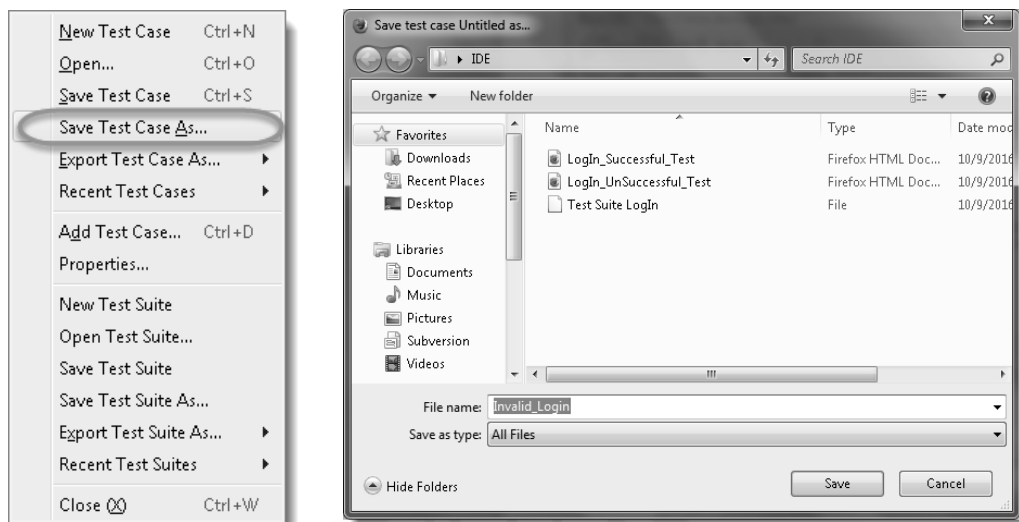


Slika 10.19. Snimljene komande

Moguće je videti i generisani izvorni kod, klikom na tab *Source*, kao što je prikazano na slici 10.20.



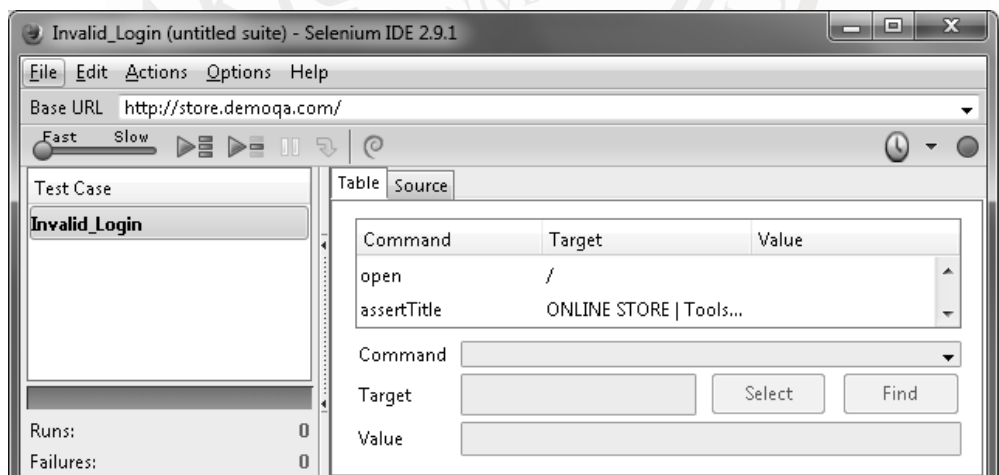
Slika 10.20. Izvorni kod snimljene test skripte



Slika 10.21. Čuvanje generisanog testa

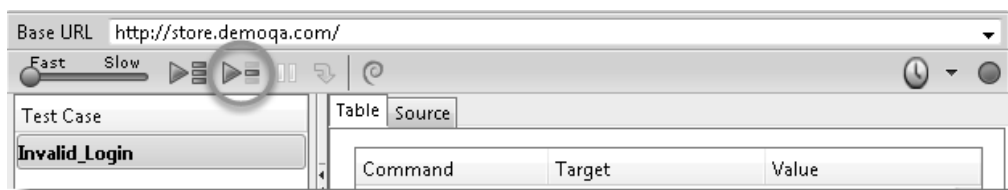
Snimljenu test skriptu je moguće sačuvati za kasniju ponovnu reprodukciju, kroz standardan File meni i opciju *Save Test Case As*, koja će otvoriti Firefox fajl sistem za čuvanje fajlova, kao što je prikazano na slici 10.21.

Nakon snimanja i čuvanja testa, on će biti dostupan u glavnom ekranu sa leve strane pod stavkom *Test Case*, kao što je prikazano na slici 10.22.



Slika 10.22. Sačuvani testovi u okviru Selenium IDE

Kako bi se ponovo pustio snimljeni test, potrebno je otvoriti novi tab u Firefox čitaču i kliknuti na Play dugme, kao na slici 10.23. Selenium će nakon toga izvršiti sve snimljene korake. Na kraju, Selenium će označiti testove koji nisu uspešno izvršeni.



Slika 10.23. Pokretanje snimljenog testa

Snimanje, modifikovanje i ponovno pokretanje testova je lako u okviru Selenium IDE okruženja. Snimljeni testovi se mogu eksportovati u veliki broj jezika, među kojima su HTML, Java, .net, perl, ruby itd. Naravno, za naprednije stvari, potrebno je koristiti drugu komponentu, Selenium WebDriver.

10.3.2. Selenium WebDriver

Najmoćniji alat u paketu Selenium je WebDriver. On se ponaša kao API koji omogućava direktno manipulisanje sa prirodnim komandama pretraživača. WebDriver prihvata komande test skripti (u Selense obliku ili preko klijentskog API-ja) i prosleđuje ih pretraživaču preko drajvera, koji je specifičan za svaki veb čitač.

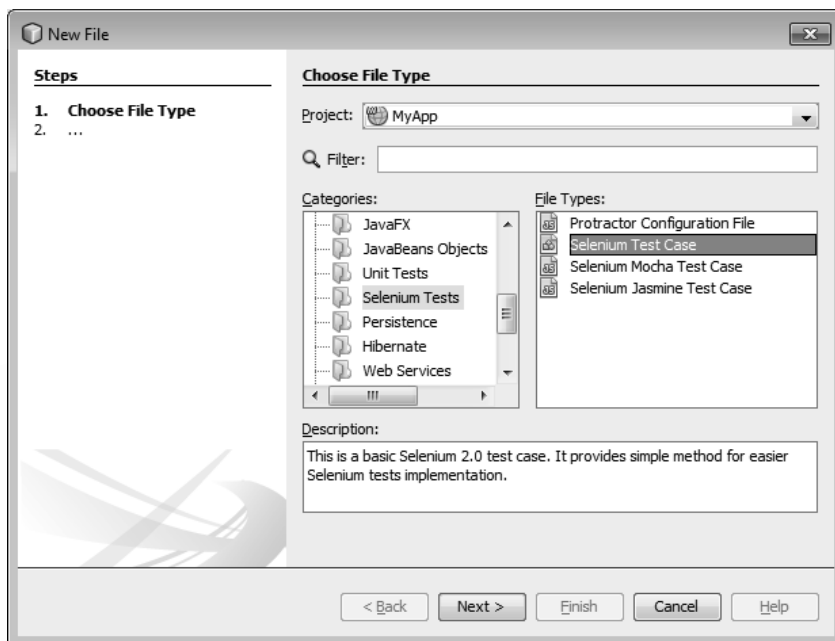
Za razliku od ostalih komponenti, Selenium WebDriver ne zahteva poseban server za pokretanje testova, već direktno pokreće veb čitač i kontroliše ga. Ovakav pristup je prevazišao sve probleme sa kojima se suočavao Selenium RC, a uz to je više okrenut korisniku, jer je API objektno orijentisan i znatno jednostavniji za učenje.

Bitna osobina WebDriver alata je da se ne zasniva na JavaScript jeziku za automatizaciju. Podržani su sledeći programski jezici:

- Java
- C#
- PHP
- Python
- Perl
- Ruby

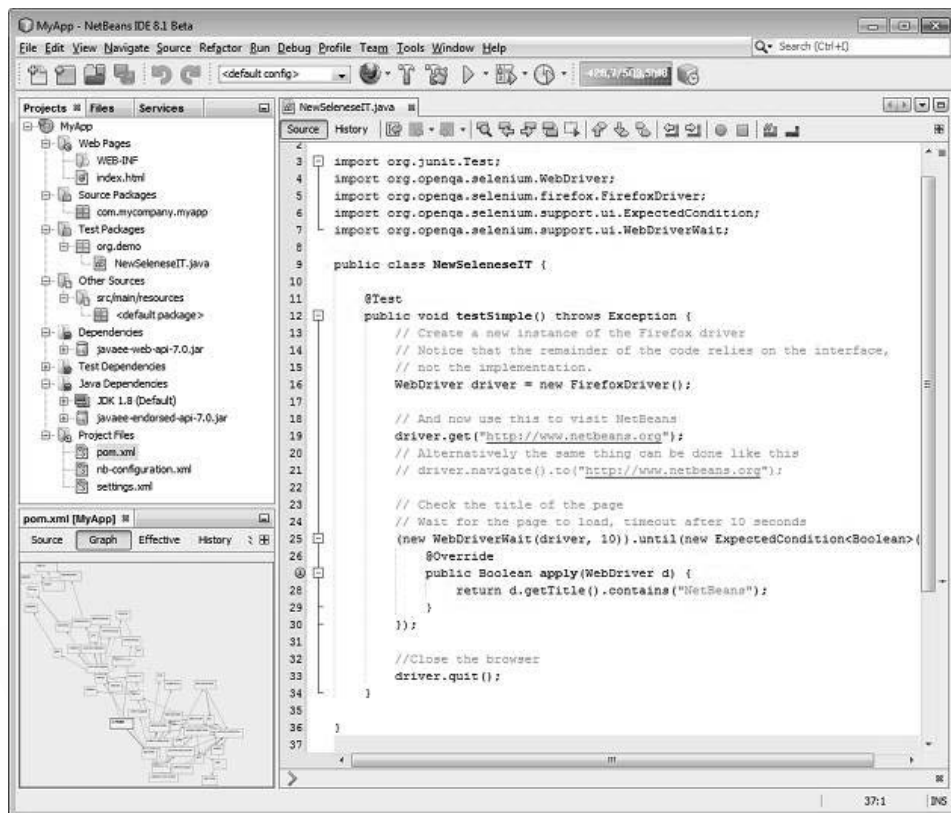
Kao jedna od mana se može izdvojiti neophodno poznavanje jednog od podržanih programskih jezika koji se želi koristiti za pisanje testova. Pošto je Selenium veoma popularan, postoji i podrška u velikom broju okruženja. Za programski jezik Java podržani su i Eclipse i NetBeans, kao dva najpopularnija okruženja za razvoj Java

aplikacija. U NetBeans okruženju, konkretno, od verzije 8.1, nije neophodno instalirati nikakav dodatni plug-in. Dovoljno je da se kreira Maven projekat i zatim doda Selenium test. Nakon kreiranja projekta, potrebno je dodati novi fajl u projekat, nakon čega se otvara ekran kao na slici 10.24. Kao kategoriju potrebno je odabrati *Selenium Tests*, a kao tip fajla *Selenium Test Case*.



Slika 10.24. Dodavanje novog Selenium testa u NetBeans okruženju

Nakon odabira Selenium testa i kompletiranja čarobnjaka, kreiran je novi test fajl, zajedno sa svim zavisnostima koje su neophodne, sa šablonom testa, kao što je prikazano na slici 10.25. U zavisnosti od željenog veb čitača, potrebno je dodati odgovarajući veb drajver (Firefox je default). Novi test je automatski konfigurisan da nakon pokretanja pokrene Firefox veb čitač i poseti stranicu NetBeans.



Slika 10.25. Šablon Selenium testa

Ukoliko je potrebno koristiti drugi veb čitač, potrebno je instalirati odgovarajući veb drajver. Za Google Chrome veb čitač, potrebno je instalirati Chrome veb drajver sa lokacije: <https://sites.google.com/a/chromium.org/chromedriver/downloads>. Zatim je neophodno u kodu specificirati lokaciju gde se on nalazi, i napraviti novu instancu drajvera.

```
System.setProperty("webdriver.chrome.driver",
    "C://chromedriver_win32/chromedriver.exe");
WebDriver driver = new ChromeDriver();
```

Osnovni test bi glasio da se uz pomoć Google Chrome čitača prvo poseti Google sajt, i da se uz pomoć polja za pretragu pronađe Univerzitet Singidunum. Implementacija ovog testa je data u nastavku:

```

import org.junit.Assert;
import org.junit.Test;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;

public class NewSeleneseIT {

    @Test
    public void testSimple() throws Exception {
        System.setProperty("webdriver.chrome.driver",
            "C://chromedriver_win32/chromedriver.exe");

        WebDriver driver = new ChromeDriver();
        driver.get("http://www.google.com/xhtml");
        Thread.sleep(5000); // pauza da korisnik vidi ekran
        WebElement searchBox = driver.findElement(By.name("q"));
//dohvata se search box
        searchBox.sendKeys("Univerzitet Singidunum");
        Thread.sleep(3000); //pauza da korisnik vidi ekran
        searchBox.submit();
        Thread.sleep(5000); // pauza da korisnik vidi ekran
        driver.quit(); //zatvaranje drajvera
    }
}

```

U pozivu metode `driver.get()` specificira se kao parametar adresa veb stranice koju je potrebno posetiti. `Thread.sleep()` metode služe samo da pauziraju test, kako bi korisnik imao vremena da vidi sve međukorake. Svaki element veb stranice je tipa `WebElement`,

i može se dohvatiti preko instance dražvera pozivom metode `findElement()`, kojoj se kao parametar daje ime ili identifikator elementa koji se želi dohvatiti. U ovom slučaju, element koji je potrebno dohvatiti je tekstualno polje u koje se unosi pojam za pretragu, čije je ime Google definisao kao "q". Nakon toga, poziva se `sendKeys()` metoda kojoj se kao parametar daje String koji je potrebno pretražiti. Metoda `sendKeys()` zapravo simulira unos korisnika sa tastature. Na kraju testa je potrebno zatvoriti dražver.

Kod pisanja automatskih testova najbitniji aspekt je interakcija sa HTML elementima veb stranice. Da bi se izvršile akcije nad elementima stranice potrebno ih je prvo locirati. Postoje različite metode za lociranje HTML elemenata, ali u osnovi se koriste lokator izrazi koji predstavljaju par vrednosti: tip lokatora i vrednost lokatora. Ove vrednosti se upoređuju sa elementima na HTML stranici da bi se locirao traženi element. Neka se kao primer posmatra login forma neke veb stranice. HTML kod za ovu formu je dat sa:

```
<html>
  <body>
    ...
    <form action="loginAction" id="loginForm">
      <label>User name:</label>
      <input type="text" name="username"><br>
      <label>Password:</label>
      <input type="text" name="password"><br>
      <button type="submit" id="loginButton">Log In</button>
      <button type="reset" id="reset">Clear</button>
    </form>
    ...
  </body>
</html>
```

Ukoliko su korisničko ime i šifra ispravni, korisnik bi bio preusmeren na HTML kod za uspešni login:

```
<html>
  <body>
    ...
    <p class="message" id="loginResponse">Welcome to our web site.
    You logged in successfully.</p>
```

```
...
</body>
</html>
```

Potrebno je testirati zadatu login formu. Mogu se uočiti dva dela:

- Korisnik unosi korisničko ime i lozinku i klikće na dugme *submit*
- Kao odgovor na uspešan login, veb sajt prikazuje odgovarajuću poruku.

Da bi Selenium test mogao da funkcioniše potrebno je identifikovati elemente HTML stranice sa kojima test treba da komunicira. U ovom primeru su to sledeći elementi:

- Polja za unos teksta User Name i Password se identifikuju preko svojih “name” atributa “username” i “password”.
- Login forma se identifikuje preko svog “id” atributa “loginForm”.
- Login odgovor paragraf koji se identifikuje preko svog “id” atributa “loginResponse”

Kod kojim se dohvataju polja za unos teksta je dat sa:

```
WebElement usernameElement = driver.findElement(By.name("username"));
WebElement passwordElement = driver.findElement(By.name("password"));
```

WebElement interfejs sadrži sledeće osnovne metode interakcije:

- sendKeys metoda - za unos teksta
- clear metoda - za brisanje unetog teksta
- submit metoda - za slanje forme

Kod kojim se unose korisničko ime u polja koja su dohvaćena u prethodnom koraku je dat sa:

```
usernameElement.sendKeys("TestTest");
passwordElement.sendKeys("Test123!");
passwordElement.submit();
```

Nakon što se forma pošalje, klikom na dugme submit, potrebno je nekoliko sekundi da server pošalje odgovor, što je očekivano ponašanje. Potrebno je u test uvesti određeno čekanje na odgovor jer bi u suprotnom test uvek davao negativan rezultat. Eksplicitno čekanje podrazumeva čekanje do momenta kada je očekivani uslov ispunjen ili dok ne istekne maksimalno vreme za čekanje. Implementira se korišćenjem WebDriverWait klase, sa maksimalnim vremenom čekanja, uz until metodu sa očekivanim stanjem.

```
WebDriverWait wait = new WebDriverWait(driver, 10);  
WebElement messageElement = wait.until(  
ExpectedConditions.presenceOfElementLocated(By.id("loginResponse"))) );
```

Kada stigne odgovor od veb sajta, potrebno je potvrditi da je login uspešan. U ovom jednostavnom primeru, potrebno je uz pomoć odgovarajuće Assert metode proveriti da li je prikazani tekst očekivan.

```
WebElement messageElement =  
driver.findElement(By.id("loginResponse"));  
String message = messageElement.getText();  
String successMsg = "Welcome to our web site. You logged in  
successfully.";  
assertEquals (message, successMsg);
```

Metoda assertEquals je standardna metoda JUnit alata, i radi na ranije opisan način.

Završavanje testova se postiže pozivanjem quit metode nad instancom WebDriver interfejsa. Ova metoda završava test tako što otpušta resurse:

- Zatvara veb stranice i ceo veb čitač
- Zatvara WebDriver server
- Otpušta sve varijable koja se odnose na jedinstvene instance WebDriver interfejsa

Ovakvo završavanje testa je neophodno jer postoji potreba da se naredni testovi izvršavaju nezavisno od prethodnih testova. Kompletan kod testa koji proverava uspešno logovanje korisnika je dat u nastavku:

```
import org.openqa.selenium.WebDriver;  
import org.openqa.selenium.firefox.FirefoxDriver;  
import org.openqa.selenium.By;  
import org.openqa.selenium.WebElement;  
import org.openqa.selenium.support.ui.ExpectedConditions;  
import org.openqa.selenium.support.ui.WebDriverWait;  
import org.junit.Assert;  
  
public class PrimerTesta {  
    public static void main(String[] args) {  
  
        // Kreiranje WebDriver instance  
        WebDriver driver = new FirefoxDriver();  
  
        // Pristupanje web stranici (kao parametar se daje stvarna adresa  
        stranice
```

```

driver.get("http://www.adresa.stranice.com");

// Izvršavanje akcija unošenja teksta i slanja forme
WebElement usernameElement =
driver.findElement(By.name("username"));
WebElement passwordElement =
driver.findElement(By.name("password"));
WebElement formElement =
driver.findElement(By.id("loginForm"));

usernameElement.sendKeys("TestTest");
passwordElement.sendKeys("Test123!");

formElement.submit();           // slanje preko elementa forme

// Eksplicitno čekanje na odgovor browsera
WebDriverWait wait = new WebDriverWait(driver, 10);

WebElement messageElement = wait.until(
ExpectedConditions.presenceOfElementLocated(By.id("loginResponse"))
);
// Provera da li je odgovor ispravan
String message = messageElement.getText();
String successMsg = "Welcome to our web site. You logged in
successfully.";
Assert.assertEquals (message, successMsg);
// Zavrsetak testa
driver.quit();
}
}

```

10.4. Zadaci za vežbu

Zadatak 1: Testira se veb stranica Univerziteta Singidunum. Koristi se alat Selenium WebDriver. Potrebno je napisati test u kome se dohvata i štampa na standardnom izlazu:

- Naslov stranice
- Dužinu naslova stranice
- Proveru da li je otvorena stranica zaista ona koja je zahtevana
- URL zahtevane veb stranice
- URL otvorene veb stranice
- Ukupan broj karaktera izvornog koda stranice

Rešenje: Kompletan kod rešenja je dat sa:

```
import org.junit.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

public class NewSeleneseIT {

    @Test
    public void testSimple() throws Exception {
        System.setProperty("webdriver.chrome.driver",
            "C://chromedriver_win32/chromedriver.exe");

        WebDriver driver = new ChromeDriver();

        // Storing the Application Url in the String variable
        String url = "https://www.singidunum.ac.rs/";

        //Launch the Singidunum page
        driver.get(url);

        // Storing Title name in the String variable
        String title = driver.getTitle();

        // Storing Title length in the Int variable
        int titleLength = driver.getTitle().length();

        // Printing Title & Title length in the Console window
        System.out.println("Title of the page is : " + title);
        System.out.println("Length of the title is : " + titleLength);

        // Storing URL in String variable
        String actualUrl = driver.getCurrentUrl();

        if (actualUrl.equals(url)) {
            System.out.println("Verification Successful - The correct
Url is opened.");
        } else {
            System.out.println("Verification Failed - An incorrect Url
is opened.");
            //In case of Fail, you like to print the actual and
expected URL for the record purpose
            System.out.println("Actual URL is : " + actualUrl);
            System.out.println("Expected URL is : " + url);
        }

        // Storing Page Source in String variable
        String pageSource = driver.getPageSource();
    }
}
```

```

// Storing Page Source length in Int variable
int pageSourceLength = pageSource.length();

// Printing length of the Page Source on console
System.out.println("Total length of the Page Source is : " +
pageSourceLength);

//Closing browser
driver.close();
driver.quit(); //closing driver
}
}

```

Izlaz programa je dat sa:

T E S T S

```

Running NewSeleneseIT
Starting ChromeDriver 2.34.522940
(1a76f96f66e3ca7b8e57d503b4dd3bccfba87af1) on port 27188
Only local connections are allowed.
Title of the page is : Univerzitet Singidunum
Length of the title is : 22
Verification Failed - An incorrect Url is opened.
Actual URL is : https://singidunum.ac.rs/
Expected URL is : https://www.singidunum.ac.rs/
Total length of the Page Source is : 81260
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 13.411
sec

Results :
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

```

Zadatak 2: Potrebno je testirati formu koja se nalazi na adresi <http://toolsqa.com/automation-practice-form/>. Na ovoj stranici se nalazi forma čiji je izgled prikazan na slici 10.26.

PERSONAL INFORMATION

Partial Link Test

Link Test

First name:

Last name:

Sex ☐ Male ☐ Female

Years of Experience ☒ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6 ☐ 7

Date:

Profession ☐ Manual Tester ☐ Automation Tester

Download Framework

[Selenium Automation Hybrid Framework](#)

[Test File to Download](#)

Automation Tool ☐ QTP ☐ Selenium IDE ☐ Selenium Webdriver

Continents

Asia ▼

Slika 10.26. Zadatak - forma koju je potrebno testirati

Potrebno je napisati Selenium WebDriver test koji je zadat sledećom specifikacijom:

- Launch new Browser
- Open “http://toolsqa.com/automation-practice-form/“
- Task One – Select the deselected Radio button (female) for category Sex (Use IsSelected method)
- Task Two – Select the Third radio button for category ‘Years of Exp’ (Use Id attribute to select Radio button)
- Task Three – Check the Check Box ‘Automation Tester’ for category ‘Profession’(Use Value attribute to match the selection)
- Task Four – Check the Check Box ‘Selenium IDE’ for category ‘Automation Tool’ (Use cssSelector)

Rešenje: Kompletan kod rešenja je dat sa:

```
import java.util.List;
import java.util.concurrent.TimeUnit;
import org.junit.Test;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;

public class NewSeleneseIT1 {

    @Test
    public void testSimple() throws Exception {
        System.setProperty("webdriver.chrome.driver",
            "C://chromedriver_win32/chromedriver.exe");

        WebDriver driver = new ChromeDriver();

        // Put an Implicit wait,
        driver.manage().timeouts().implicitlyWait(10,
            TimeUnit.SECONDS);

        // Launch the URL
        driver.get("http://toolsqa.wpengine.com/automation-practice-
            form");

        // Task 1 : Select the deselected Radio button (female) for
        category Sex (Use isSelected method)
        // Storing all the elements under category 'Sex' in the list
        of WebElements
        List<WebElement> rdBtn_Sex =
            driver.findElements(By.name("sex"));

        // Create a boolean variable which will hold the value
        (True/False)
        boolean bValue = false;

        // This statement will return True, in case of first Radio
        button is selected
        bValue = rdBtn_Sex.get(0).isSelected();

        // This will check that if the bValue is True means if the
        first radio button is selected
        if (bValue == true) {
            // This will select Second radio button, if the first
            radio button is selected by default
        }
    }
}
```

```

        rdBtn_Sex.get(1).click();
    } else {
        // If the first radio button is not selected by default,
the first will be selected
        rdBtn_Sex.get(0).click();
    }

    //Task 2: Select the Third radio button for category 'Years of
Exp' (Use Id attribute to select Radio button)
    WebElement rdBtn_Exp = driver.findElement(By.id("exp-2"));
    rdBtn_Exp.click();

    // Task 3: Check the Check Box 'Automation Tester' for
category 'Profession' ( Use Value attribute to match the selection)
    // Find the Check Box or radio button element by Name
    List<WebElement> chkBx_Profession =
driver.findElements(By.name("profession"));

    // This will tell you the number of Check Boxes are present
    int iSize = chkBx_Profession.size();

    // Start the loop from first Check Box to last Check Boxe
    for (int i = 0; i < iSize; i++) {
        // Store the Check Box name to the string variable, using
'Value' attribute
        String sValue =
chkBx_Profession.get(i).getAttribute("value");

        // Select the Check Box if the value of the Check Box is
same what you are looking for
        if (sValue.equalsIgnoreCase("Automation Tester")) {
            chkBx_Profession.get(i).click();
            // This will take the execution out of for loop
            break;
        }
    }

    // Task 4: Check the Check Box 'Selenium IDE' for category
'Automation Tool' (Use cssSelector)
    WebElement oCheckBox =
driver.findElement(By.cssSelector("input[value='Selenium IDE']"));
    oCheckBox.click();
    // Kill the browser
    driver.quit();
}
}

```

Pitanja

1. Zbog čega je važno testirati grafički korisnički interfejs?
2. Na čemu je zasnovan grafički korisnički interfejs?
3. Šta je potrebno testirati u grafičkom korisničkom interfejsu?
4. Koji su mogući načini testiranja korisničkog interfejsa?
5. Zbog čega je bitan izgled i dizajn aplikacije?
6. Kako se radi testiranje po modelu stanja?
7. Kako se radi automatsko testiranje?
8. Na šta je potrebno obratiti pažnju prilikom testiranja korisničkih interfejsa veb aplikacija?
9. Koji problemi nastaju prilikom korišćenja različitih veb čitača?
10. Šta je SQL injekcija?
11. Šta je Selenium?
12. Od kojih se komponenti sastoji Selenium?
13. Koja je najbitnija komponenta Seleniuma?
14. Kako se u Selenium alatu identifikuju elementi grafičkog interfejsa veb aplikacije sa kojima treba izvršiti interakciju?
15. Kako se pomoću Selenium alata šalje tekst u neko polje?
16. Šta je WebDriver objekat u Selenium testu i čemu služi?

11. TESTIRANJE OBJEKTNO ORIJENTISANOG SOFTVERA

U testiranju objektno orijentisanog softvera mogu se primeniti sve do sada pokrivene metode. Međutim, za potpuno testiranje objektno orijentisanog softvera, potrebno je obratiti pažnju na specifičnosti koje ovakav tip softvera nosi. Objektno orijentisani jezici, među kojima i Java, imaju akcenat na apstrakciji u softveru. Apstrakcije se implementiraju klasama koje predstavljaju korisnički definisane tipove, koji imaju i stanje i ponašanje. Ovaj pristup programiranju ima mnoge korisne strane, ali takođe utiče i na testiranje. Kompleksnost softvera se pomera sa algoritama implementiranim u jedinicama i metodama na to kako se komponente povezuju. Samim tim, umanjuje se fokus na jedinično, a povećava značaj integracionog testiranja.

Drugi značajan faktor su relacije između komponenti koje mogu biti veoma kompleksne, i zajedno sa nasleđivanjem i polimorfizmom, uvode nove tipove grešaka i zahtevaju dodatne metode za testiranje. Svi objektno orijentisani jezici koriste klase, nasleđivanje, polimorfizam i dinamičko vezivanje kako bi se podržala apstrakcija. Na primer – ukoliko klasa B proširuje klasu A, i obe klase A i B definišu metod `m()`, `m()` je u tom slučaju polimorfni metod. Ukoliko je objekat deklarisan kao tip A, odnosno u Javi:

```
A x;
```

za vreme izvršavanja programa `x` može pokazivati na objekat tipa A (`x = new A();`) ili tipa B (`x = new B();`). Ukoliko se sada posmatra poziv polimorfnog metoda:

```
x.m();
```

verzija metode `m()` koja će biti pozvana (iz A ili iz B) zavisi od trenutnog konkretnog tipa objekta `x`. Kolekcija metoda koje mogu biti pozvane se zove PCS (engl. *Polymorphic Call Set*). U ovom slučaju PCS za `x.m()` je:

```
A : m()
```

```
B : m()
```

Ovakvi primeri dinamičkog vezivanja, gde se do trenutka izvršavanja ne zna koja će metoda iz PCS zaista biti pozvana, imaju veliki uticaj na testiranje.

11.1. Jedinstveni problemi u testiranju OOP

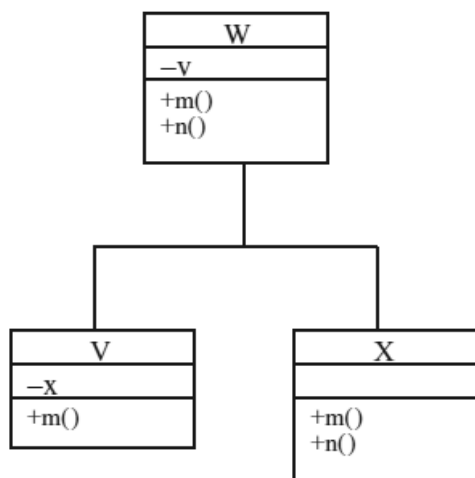
Određeni problemi su jedinstveni za objektno orijentisane programe. Prema nekim istraživanjima, tradicionalne tehnike nisu u potpunosti efikasne za objektno orijentisani softver, i ponekad ne pokrivaju sve moguće probleme. Konkretno, u objektno orijentisanom softveru metode su najčešće manje veličine i ukoliko su dobro napisane nisu previše kompleksne, pa testiranje putanja može da bude manje efikasno. Sa druge strane, nasleđivanje, polimorfizam i dinamičko vezivanje uvode nove probleme koji se moraju adekvatno testirati.

Kada se testira objektno orijentisani softver, klasa se najčešće posmatra kao osnovna jedinica testiranja. Postoje četiri osnovna nivoa testiranja klasa:

- Intra-metodsko - Testovi se pišu za individualne metode (tradicionalno jedinično testiranje)
- Inter-metodsko - Više metoda unutar klase se testira istovremeno (tradicionalno testiranje modula)
- Intra-klasno - Testovi se pišu za jednu klasu, u obliku sekvence poziva metodama te klase
- Inter-klasno - Testira se više od jedne klase istovremeno i posmatra interakcija (integraciono testiranje).

Problemi povezani sa nasleđivanjem, dinamičkim vezivanjem i polimorfizmom se ne mogu otkriti testiranjem individualnih metoda ili klasa. Ovi problemi zahtevaju istovremeno testiranje više klasa koje su povezane putem nasleđivanja i polimorfizma, odnosno mora se koristiti inter-klasno testiranje. Kao osnova za početak inter-klasnog testiranja najčešće se koristi klasni dijagram.

Jedan od najvećih problema u razvoju objektno orijentisanog softvera jeste vizuelizacija interakcija između klasa koje se mogu desiti u prisustvu nasleđivanja, polimorfizma i dinamičkog vezivanja. Ove interakcije mogu biti veoma kompleksne, naročito kada se odnose na veći broj klasa. Vizuelizacija obično podrazumeva da klasa enkapsulira informacije o stanju u svojim poljima (podacima članovima), i da ima skup ponašanja koji je implementiran metodama koje koriste polja klase. Neka se posmatra klasni dijagram sa slike 11.1.



Slika 11.1. Primer klasnog dijagrama sa nasleđivanjem

Sa klasnog dijagrama se vidi da klase V i X proširuju klasu W. Klasa V nadjačava metod m() iz klase W. Klasa X nadjačava metode m() i n() iz klase W. Na klasnom dijagramu znak – označava da je član privatan, a znak + označava da je član javan. Za ilustraciju problema sa dinamičkim vezivanjem dat je sledeći kod:

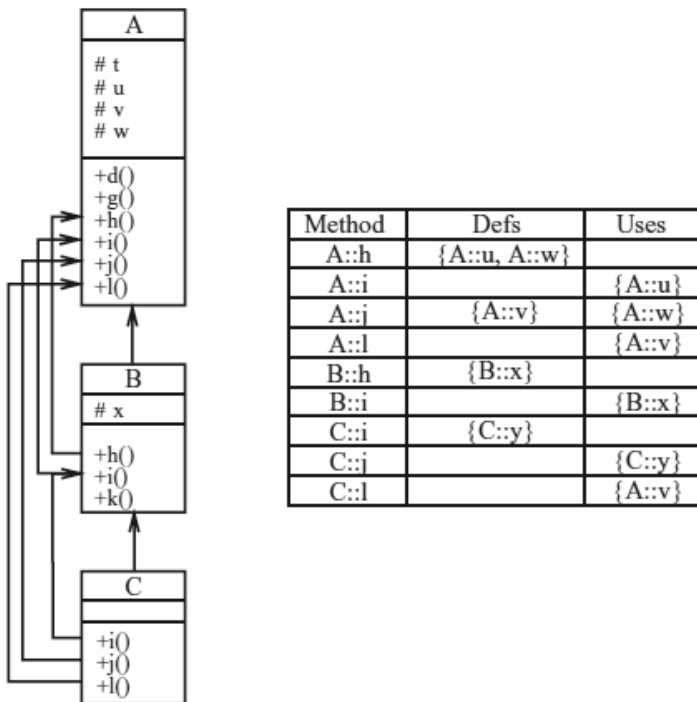
```

void f (boolean b) {
    W o;
    ...
    if(b)
        o = new V();
    else
        o = new W();
    ...
    o.m();
}

```

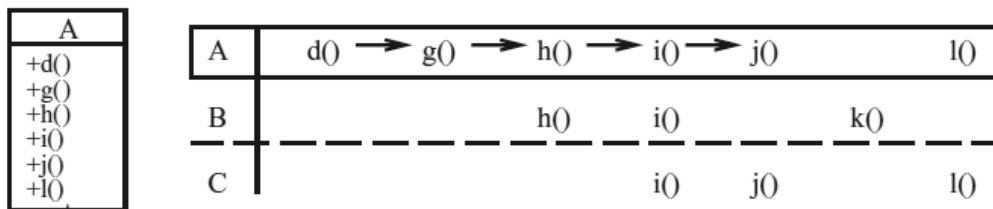
Deklarisan tip promenljive *o* je *W*. Na liniji gde se poziva *o.m()*, stvarni tip može biti ili *V* ili *W*. Pošto *V* nadjačava metod *m()*, koja verzija metode *m()* će biti pozvana zavisi od vrednosti parametra *b*, i ta informacija nije poznata za vreme prevođenja, već se odluka odlaže za vreme izvršavanja.

Kako bi se dalje demonstrirali problemi nadjačavanja metoda i polimorfizma, neka se posmatra nasleđivanje na klasnom dijagramu sa slike 11.2. Klasa *A* ima 4 polja i 6 metoda. Polja su deklarisan kao *protected* (na dijagramu označeno sa znakom #), odnosno može im se pristupiti iz izvedenih klasa *B* i *C*. Klasa *B* deklarise još jedno polje i tri metode, a klasa *C* deklarise još tri metode. Strelice na slici prikazuju nadjačavanje metoda. Na slici su takođe prikazane i definicije i upotrebe polja.



Slika 11.2. Primer klasnog dijagrama sa anomalijama u toku podataka

Na slici 11.3 prikazane su nadjačane metode. Sa slike se vidi da je metoda `h()` iz klase A nadjačana u klasi B, dok je na primer metoda `i()` nadjačana u klasi B, i zatim ponovo nadjačana u klasi C.



Slika 11.3. Grafički prikaz nadjačanih metoda i redosleda pozivanja

Moguće anomalije nastaju sa pozivom `A::d()`, koji može napraviti veoma kompleksne probleme. Neka se promenljiva `o` veže za objekat A na sledeći način:

```
o = new A ();
```

Zatim se napravi poziv `o.d()`. Ovaj poziv poziva metodu `A::d()`, koja zatim poziva metodu `A::g()`, zatim `A::h()`, pa `A::i()`, koja na kraju poziva metodu `A::j()`, kao što je prikazano na slici 11.3 (strelice u gornjem redu definišu redosled pozivanja metoda u A). U ovom slučaju, promenljive `A::u` i `A::w` su prvo definisane, i zatim korišćene u `A::i()` i `A::j()`, bez ikakvih anomalija (slika 11.2).

Problemi nastaju ukoliko se u promenljivu `o` stavi instanca klase B:

```
o = new B();
```

Poziv `o.d()` dovodi do pozivanja metoda `h()` i `i()` iz B, pošto su ove metode iz A nadjačane u klasi B. U ovom slučaju, promenljive `A::u` i `A::w` nisu prvo definisane, a poziv `A::j()` dovodi do problema u toku podataka (pokušava se upotreba promenljive koja nije definisana).

11.2. Jo-jo graf

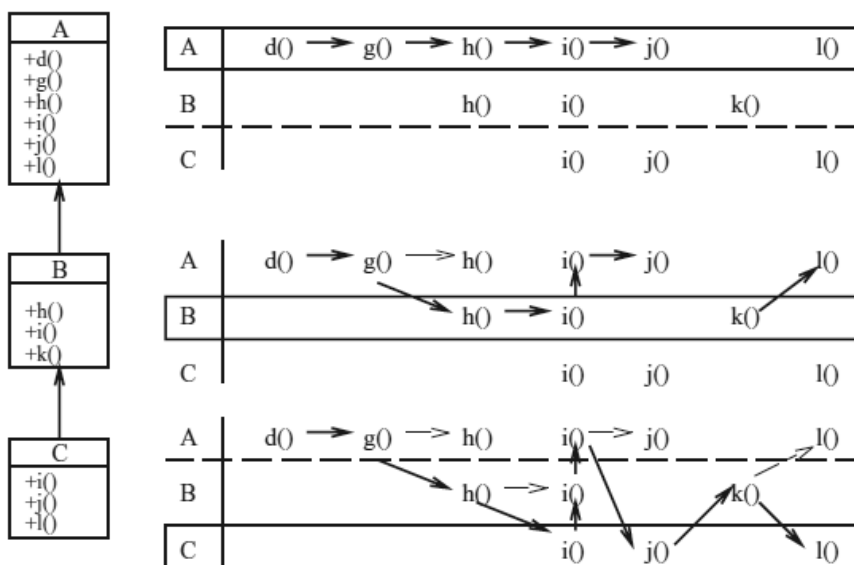
Jo-jo graf služi za vizuelizaciju polimorfizma i razumevanje koja verzija metode će biti zaista izvršena, a takođe i koje sve verzije te metode mogu biti izvršene. Izvršavanje može ići gore dole kroz nivoe nasleđivanja, i javlja se takozvani jo-jo efekat. Jo-jo graf se definiše nad hijerarhijom nasleđivanja. Ovaj graf prikazuje sve, nove, nasleđene i nadjačane metode za svaku izvedenu klasu, a pozivi se prikazuju strelicama.

Neka se ponovo razmatra hijerarhija klasnog dijagrama sa slike 11.2 i redosleda pozivanja metoda sa slike 11.3. Dodatno, neka važe određene izmene za nadjačane metode. Pretpostavlja se da u klasi A, metoda `d()` poziva metodu `g()`, `g()` poziva `h()`, `h()` poziva `i()`, a `i()` poziva `j()`. Dalje, pretpostavlja se da u klasi B, metoda `h()` poziva `i()`, `i()` poziva metodu `i()` iz bazne klase A, i `k()` poziva `l()`. Na kraju, pretpostavlja se da u klasi C, `i()` poziva metodu `i()` iz svoje bazne klase (sada klasa B), a `j()` poziva `k()`.

Ova situacija se može vizuelno prikazati na način koji je prikazan na slici 11.4. Ova slika prikazuje aktuelne sekvence poziva metoda ukoliko se napravi poziv metode d() kroz instance klase A, B i C respektivno. Gornji deo slike 11.4 prikazuje poziv metode d() kroz instancu klase A. Ovaj poziv inicira sekvencu koja je vrlo jednostavna i jednosmerna.

Ukoliko se posmatra srednji deo slike 11.4, odnosno poziv metode d() kroz instancu klase B, već se vidi da je situacija kompleksnija. Nakon poziva metode g(), sledeća metoda koja se poziva je metoda h() iz klase B, dakle poziva se nadjačana verzija B::h(), dok se verzija iz A, odnosno A::h() ne poziva. Kontrola se dalje prosleđuje sledećim redom – B::i(), A::i() i na kraju A::j().

Ukoliko se posmatra donji deo slike 11.4, odnosno poziv metode d() kroz instancu klase C, može se videti jo-jo efekat po kome je graf i dobio ime. Kontrola se redom prosleđuje od A::g() na B::h(), zatim C::i(), nakon toga se ide gore na B::i(), zatim A::i(), pa se ide dole na C::j(), pa ponovo gore na B::k(), i najzad na kraju dole na C::l().



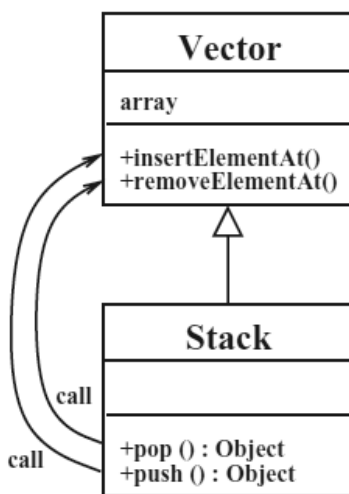
Slika 11.4. Jo-jo graf

Ovaj primer ilustruje kompleksnost koja može nastati u objektno orijentisanim programima usled polimorfizma i nadjačavanja metoda. Ova kompleksnost otežava proces testiranja pošto se mora tretirati na adekvatan način.

11.3. Problemi u nasleđivanju i polimorfizmu

Nasleđivanje omogućava programerima da budu kreativniji, efikasniji, i da ponovo upotrebe već postojeće komponente. Sa druge strane, novi tipovi grešaka koji se primenom nasleđivanja uvode su često veoma teški za detekciju, dijagnozu i ispravku. Većina tipova grešaka se može uočiti u svim objektno orijentisanim programskim jezicima. U nastavku ovog poglavlja je dato nekoliko primera koji ilustruju novu dimenziju problema koji su uvedeni nasleđivanjem.

Nekonzistentna upotreba tipova - ITU (engl. *Inconsistent Type Use*) je slučaj kada izvedena klasa ne nadjačava nijednu nasleđenu metodu, pa nema polimorfnog ponašanja. Svaka instanca izvedene klase C kada se koristi tamo gde je očekivana instanca bazne klase T, se može ponašati samo kao instanca klase T. Samo metode definisane u T se mogu koristiti a sve dodatne metode definisane u C su sakrivene. Problemi mogu nastati kada se instanca klase C koristi u više konteksta, prvo kao T, zatim kao C, i opet kao T. Kao primer se može posmatrati klasa Stack koja koristi metode nasleđene iz klase Vector za implementaciju steka. Klasni dijagram je dat na slici 11.5.



Slika 11.5. Klasa Stack izvedena iz klase Vector

Neka je implementacija klase Stack takva da metoda pop() iz klase Stack poziva metodu removeElementAt() iz klase Vector. Dalje, metoda push() iz klase Stack poziva metodu insertElementAt() iz klase Vector. Ove dve klase imaju očigledno različito značenje. Sve dok se instanca Stack koristi samo kao Stack, neće biti problema. Neće biti problema ni ako se instanca Stack koristi isključivo kao Vector. Problem nastaje kada se objekat ponekad koristi kao Stack, a ponekad kao Vector. Neka se sada posmatra sledeći fragment koda:

```

1    public void f (Stack s)
2    {
3        String s1 = "s1";
4        String s2 = "s2";
5        String s3 = "s3";
6        . . .
7        s.push (s1);
8        s.push (s2);
9        s.push (s3);
10
11       g (s);
12
13       s.pop();
14       s.pop();
15       // Ovde je stack prazan!
16       s.pop();
17       . . .
18   }
19   public void g (Vector v)
20   {
21       // Uklanjanje poslednjeg elementa
22       v.removeElementAt (v.size()-1);
23   }

```

Tri elementa se stavljaju na vrh steka s, pozivom s.push() metode. Nakon toga, poziva se metoda g() koja kao parametar prima Vector, što znači da se steku s unutar metode g() pristupa kroz promenljivu tipa Vector. Na žalost, metoda g() uklanja element iz sredine steka, što predstavlja kršenje semantike steka. Još gore, tri poziva metode s.pop() nakon poziva metode g() neće raditi ispravno. Poziv metode s.pop() na liniji 14 neće ukloniti poslednji element koji je bio dodat (taj element je nasilno uklonjen u metodi g()), čime je integritet steka prekršen. Treći poziv metode s.pop() na liniji 16 će izazvati grešku, pošto je u tom trenutku stek prazan.

Kao drugi primer posmatra se slučaj poznat kao greška nepotpune konstrukcije objekta - IC (engl. *Incomplete Construction Fault*). Odnosi se na situaciju gde inicijalno stanje objekta nije definisano na ispravan način. Konstruktor treba da uspostavi inicijalno stanje za novu instancu klase, i zbog toga uopšteno gledano konstruktori imaju instrukcije koje definišu svaki član podatak. Kada se u konstrukciju objekata ubaci nasleđivanje, neki članovi mogu da se previde i preskoče. Neka se posmatra sledeći fragment koda kao primer:

```

1    Class abstract AbstractFile
2    {
3        FileHandle fd;
4
5        abstract public open();

```

```

6
7         public read() {fd.read ( . . . ); }
8
9         public write() {fd.write ( . . . ); }
10
11        abstract public close();
12    }
13    ...
14    Class SocketFile extends AbstractFile
15    {
16        public open()
17        {
18            fd = new Socket ( . . . );
19        }
20
21        public close()
22        {
23
24            fd.flush();
25            fd.close();
26        }

```

Namera programera koji je pisao AbstractFile je bila da izvedena klasa da definiciju fd pre upotrebe, što se radi u metodi open() izvedene klase SocketFile. Ukoliko nijedna metoda ne koristi fd pre definicije, nema problema. Greška nastaje ukoliko neka metoda pokuša da koristi fd pre poziva open (). Na primer, pozivi read() i write() bi izazvali grešku ukoliko bi bili pozvani neposredno posle konstrukcije.

Još jedan primer grešaka specifičnih za objektno orijentisane programe je nekonzistentno stanje usled sakrivanja polja – SDIH (engl. *State Definition Inconsistency due to State Variable Hiding fault*), gde uvođenje lokalne promenljive može da uzrokuje anomaliju u toku podataka. Ukoliko postoji lokalna promenljiva v koja je definisana u izvedenoj klasi, i ima isto ime kao nasleđena promenljiva v, onda je nasleđena promenljiva sakrivena u opsegu izvedene klase (osim ukoliko joj se eksplicitno ne pristupa sa super.v). U izvedenoj klasi referenca v se uvek odnosi na v iz izvedene klase. To nije problem ukoliko su sve nasleđene metode nadjačane, jer nijedna druga metoda onda ne bi mogla implicitno da referiše nasleđeno v. Međutim, neke metode se ponekad ne nadjačavaju u izvedenoj klasi. Anomalija u toku podataka nastaje u slučaju da je metoda koja definiše nasleđeno v takođe nadjačana u izvedenoj klasi, dok je i sama promenljiva v sakrivena u izvedenoj klasi lokalnom definicijom promenljive v.

Prethodno opisani primeri ilustruju nove tipove grešaka koje su uvedene sa objektno orijentisanim programiranjem. Postoji još grešaka koje su specifične za objektno orijentisano programiranje, ali su van okvira ove knjige i čitalac se upućuje na dodatnu literaturu.

11.4. Kompleksnost objektno orijentisanog softvera

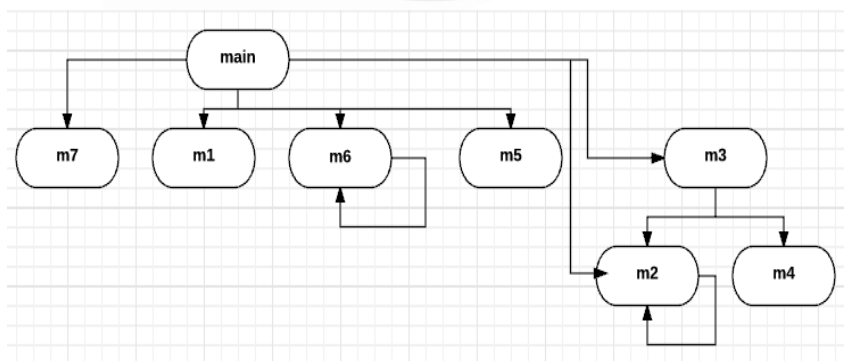
Izračunavanje ciklomatske kompleksnosti svake pojedinačne metode u nekom objektu odgovara upotrebi metrike na svakom modulu strukturnog koda. Međutim, ciklomatska kompleksnost objektno orijentisanih sistema se ne može izračunati prostim sabiranjem kompleksnosti pojedinačnih metoda, pošto ne utiču sve metode sa istim faktorom na ukupnu kompleksnost. Neke metode koje se pozivaju unutar određene metode mogu biti nasleđene iz baznih klasa, druge metode se koriste unutar samog objekta, dok se treće mogu pozivati iz drugih delova koda.

Dodatno, metode imaju samo jednu tačku ulaska, ali neke mogu imati više tačaka izlaska. Takve metode uvećavaju ukupnu ciklomatsku kompleksnost koda. Zbog toga slični sistemi mogu imati različite nivoe kompleksnosti. Svaka metoda definisana u sistemu je jedinstvena u izračunavanju ciklomatske kompleksnosti. Sa druge strane, kod može da izvršava istu semantičku funkciju u različitim metodama različitih klasa, ali zbog drugačije implementacije kompleksnost može biti različita.

Jedan od koncepata koji se može koristiti u objektno orijentisanim sistemima je koncept proširenog grafa toka kontrole (engl. *Extended Control Flow Graph* – ECFG). Softverski sistem se predstavlja preko višeslojnog grafičkog modela koji sadrži kolekciju grafova toka kontrole individualnih metoda softvera. Svaki metod se posmatra kao proceduralni program i ima svoj graf toka kontrole koji označava kontrolu između instrukcija unutar tela metode. Sam model proširenog grafa toka kontrole ima dva sloja:

- Sloj na vrhu predstavlja metode individualnih klasa
- Sledeći sloj predstavlja CFG ovih metoda

Primer modela proširenog grafa toka kontrole je dat na slici 11.6. Metode koje se pozivaju iz main metode su prikazane na grafu i označene od m1 do m7. Međusobni pozivi metoda su označeni strelicama, tako da na primer metoda m3 poziva metode m2 i m4, dok metode m2 i m6 pozivaju same sebe (rekurzija). Svaka od ovih metoda ima svoj graf toka kontrole, koji nije prikazan na slici.



Slika 11.6. Primer proširenog grafa toka kontrole

Prošireni graf toka kontrole ima sledeća svojstva:

- Graf je sličan CFG u smislu čvorova i grana, izuzev gornjeg sloja gde neki čvorovi ne moraju biti povezani. U pitanju je serija grafova organizovanih u ovakve slojeve.
- Čvorovi u CFG se odnose na instrukcije, dok se u ECFG odnose na metode.
- Svaka metoda ima svoj CFG i svoju vrednost ciklomatske kompleksnosti. Ukoliko metod nije pronađen u klasi, moguće da je nasleđen iz bazne klase.
- Deklaracija objekata je slična deklaraciji promenljivih u proceduralnim jezicima, ali nije sekvenca instrukcija već se odnosi na konstruktorsku metodu.
- Grane između čvorova se formiraju svaki put kada neki metod zove drugi metod.

U ECFG, metode (odnosno njihovi CFG) mogu biti povezani na više načina. ECC, odnosno zajednička kompleksnost dva ili više grafova je jedinstvena, uz očuvanje CC (odnosno broja nezavisnih putanja). Različiti slučajevi i vrednosti ECC, kao i odgovarajući dokazi su izvedeni u [ref na ECFG], i prevazilaze okvire ove knjige. U ovom udžbeniku su samo prezentovane neke karakteristične dobijene vrednosti.

Slučaj 1: dva ili više grafova su povezani u seriji, na primer metode se izvršavaju u sekvenci jedna iza druge. Ako su $V(G_1)$, $V(G_2)$... $V(G_n)$ kompleksnosti individualnih grafova onda ECC ima vrednost najveće od njih.

$$ECC = V(G_x), \text{ ako važi } V(G_x) \geq V(G_1), V(G_2) \dots V(G_n), \text{ i } 1 \leq x \leq n$$

Slučaj 2: dva ili više grafova su ugnježdjeni unutar jednog grafa, na primer metoda zove drugu metodu, koja zatim zove treću i tako dalje. Ako su $V(G_1)$, $V(G_2)$... $V(G_n)$ kompleksnosti individualnih grafova, a graf G_2 je ugnježđen unutar G_1 , G_3 unutar G_2 i tako dalje, onda je ECC data na sledeći način:

$$ECC = V(G_1) + V(G_2) + \dots + V(G_n) + (n-1)$$

Slučaj 3: jedan graf se rekurzivno ponavlja, na primer jedna metoda se rekurzivno poziva. Ako je $V(G_1)$ kompleksnost metode koja se rekurzivno poziva, onda je ECC data sa:

$$ECC = V(G_1) + 2$$

Pitanja

1. Zbog čega je potrebno tretirati objektno orijentisane programe na drugačiji način od standardnih programa?
2. Šta je osnovna jedinica testiranja objektno orijentisanih programa?
3. Šta je PCS?
4. Koja su osnovna četiri načina testiranja klasa?
5. Koji problemi mogu nastati usled dinamičkog vezivanja?
6. Šta je jo-jo graf i čemu služi?
7. U kojim situacijama dolazi do jo-jo efekta?
8. Koji su primeri problema koji mogu da nastanu primenom polimorfizma.
9. Kako se računa kompleksnost objektno orijentisanih programa?
10. Objasniti koncept proširenog grafa toka kontrole?



12. MENADŽMENT DEFEKATA

Svrha testiranja softvera se svodi na pronalaženje defekata u aplikaciji koja se testira, sa ciljem da ti defekti budu ispravljani pre isporuke aplikacije klijentu. U prethodnim poglavljima opisane su različite tehnike testiranja. Sve ove tehnike mogu dati rezultat u obliku pronađenih defekata u programu. Nakon što su defekti pronađeni, potrebno ih je zvesti na adekvatan i koncizan način, kako bi se što lakše mogli pratiti, i na kraju krajeva, ispraviti. Proces zavođenja i praćenja defekata se zove menadžment defekata, i predstavlja jedno od bitnih zaduženja softver testera. Za menadžment defekata se obično koristi softverski sistem za praćenje defekata. Postoji veliki broj ovakvih alata dostupnih na tržištu, a neki od najpopularnijih su:

- Jira
- Bugzilla
- Asana
- HP ALM
- Visual Studio TFS

Poređenjem stvarnog i očekivanog rezultata, tester procenjuje da li se sistem ispravno ponaša. Incident (anomalija) je neočekivan događaj koji zahteva dalju istragu. Incident može biti otkaz sistema uzrokovan defektom u sistemu. Incident može i ne mora biti zaveden u obliku izveštaja o bugu. Defekt je stvarni problem koji treba da bude rešen. Generalno važi opšte pravilo testiranja: što se ranije bug otkrije i ispravi, niži su ukupni troškovi obezbeđivanja kvaliteta sistema.

Sistem za praćenje defekata treba da sadrži informaciju o tome u kojoj fazi razvoja softvera je defekt stvoren i u kojoj fazi je detektovan. Ako su te dve faze iste, onda to znači da defekt nije propagirao. Ako netačan zahtev „promakne“ prilikom statičkog testiranja zahteva i kao takav bude implementiran od strane programera, testiran od strane testera i prijavljen tek prilikom korisničkog testiranja, sav rad na tom zahtevu je protraćeno vreme (a može dovesti i do gubitka poverenja korisnika).

12.1. Izveštaj o defektu

Nakon što tester otkrije bug, u najvećem broju slučajeva je potrebno da ga na ispravan način zavede u odabrani alat za menadžment defekata. Parametri koji se beleže u izveštaju o bugu treba da obezbede dovoljno informacija tako da je izveštaj upotrebljiv. Upotrebljiv izveštaj je:

- Kompletan – sve neophodne informacije su unete u izveštaj;
- Koncizan – izveštaj ne sadrži suvišne informacije;
- Tačan – informacije date u izveštaju su tačne i jasno opisuju stvarne i očekivane rezultate kao i tačne korake pomoću kojih se bug može reprodukovati;
- Objektivn – izveštaj predstavlja profesionalno napisano iznošenje činjenica.

Izveštaji o bugovima se pišu za greške otkrivene tokom funkcionalnog i nefunkcionalnog testiranja. Izveštaj o bugu treba da sadrži različita polja. Što su polja bolje definisana, to je lakše da se naprave izveštaji o defektu kao i da se što lakše naprave test izveštaji. Dobro definisani bugovi se lako prate i olakšavaju otklanjanje greške. Podaci u poljima treba da se validiraju kako bi se izbegle greške pri unosu i kako bi se obezbedilo efikasno izveštavanje.

Informacije u ovim izveštajima uvek treba da budu orijentisane ka jasnom prikazu scenarija tokom kog je defekt otkriven, uključujući i korake i podatke za reprodukovanje, kao i očekivane i stvarne rezultate. Ako je moguće u izveštaj treba uključiti logove i snimke scenarija u kom se defekt desio. Neophodno je i navesti podatke o sistemu:

- okruženje,
- operativni sistem,
- verziju build-a,
- verziju hardvera, itd.

Iako je primarni cilj kreiranja izveštaja o bugu popravka problema, informacije u izveštaju takođe potpomažu odgovarajuću klasifikaciju, procenu rizika i ukoliko je neophodno, poboljšanje procesa testiranja. Jedan primer izveštaja o bugu sa najčešće korišćenim poljima je dat na slici 12.1.

Slika 12.1. Novi izveštaj o bugu

U zavisnosti od izabranog alata za menadžment defekata, mogu biti dostupna još neka polja. Svi alati dodatno imaju i mogućnost modifikacije, pa je moguće dodati i sopstvena polja u zavisnosti od potreba projekta.

12.2. Klasifikacija defekata

Odgovarajuća klasifikacija defekata je suština izveštavanja u procesu testiranja. Klasifikovanjem se defekti grupišu na osnovu čega se procenjuje efikasnost testiranja i razvoja softvera. Defekti se najčešće klasifikuju prema:

- Ozbiljnosti defekta (engl. *severity*)
- Prioritetu

Ozbiljnost defekta označava do koje mreže određeni defekt ugrožava softver, odnosno njegov uticaj (engl. *impact*) na sistem. Postoje različite podele, dok se najčešća zasniva na sledećim tipovima:

- *Critical*
- *Major*
- *Moderate*
- *Minor*
- *Cosmetic*

Critical (poznat i pod drugim engleskim nazivom *showstopper*) je defekt koji rezultira prekidom rada kompletnog sistema, ili neke od njegovih komponenti, i izaziva značajnu korupciju podataka. Specificirana funkcionalnost je neupotrebljiva, i nije moguće alternativnom metodom postići zahtevane rezultate. Ugrožena je osnovna funkcionalnost programa, neupotrebljiv je za krajnje korisnike.

Major defekt rezultira prekidom rada kompletnog sistema, ili neke od njegovih komponenti, i izaziva značajnu korupciju podataka. Specificirana funkcionalnost je neupotrebljiva, ali postoji mogućnost da se alternativnom metodom postignu zahtevani rezultati.

Moderate defekt ne izaziva prekid rada sistema, ali izaziva nekorektno i netačno ponašanje, odnosno ima nekompletne ili nekonzistentne rezultate. *Minor* defekt ne izaziva prekid rada sistema i ne utiče na upotrebljivost sistema. *Cosmetic* defekt je zapravo poboljšanje sistema u vidu promena u izgledu aplikacije (engl. *look and feel*).

Prioritet defekata označava redosled u kojem se oni popravljaju. Prioritet postavlja tester, i označava vremenski okvir u kome programer treba da popravi defekt. Ukoliko je prioritet visok (engl. *high*) – programer mora da ga ispravi što je pre moguće. Prioritet se postavlja na osnovu korisničkih zahteva iz specifikacije.

Najčešći tipovi prioriteta:

- *Low* – defekt je iritantan i treba biti popravljen, ali popravka može biti odložena dok se ne isprave ozbiljniji defekti.
- *Medium* – defekt treba biti popravljen u normalnom toku aktivnosti razvoja softvera. Može sačekati sledeći build (verziju).
- *High* – defekt mora biti rešen što je pre moguće, jer ozbiljno utiče na upotrebljivost proizvoda. Sistem se ne može koristiti ukoliko se ispravka ne izvrši.

Važni primeri scenarija vezanih za upotrebu ozbiljnosti i prioriteta defekta:

- Visok prioritet i visoka ozbiljnost – greška koja ugrožava osnovnu funkcionalnost programa, i korisnici ne mogu da koriste sistem. Primer, sajt studentskih servisa, prilikom ažuriranja podataka i ocena, ne dozvoljava snimanje promene.
- Visok prioritet i niska ozbiljnost – slovna greška na naslovnoj strani, ili u naslovu aplikacije.
- Nizak prioritet i visoka ozbiljnost – greška koja ugrožava funkcionalnost aplikacije i ne dozvoljava određenu upotrebu sistema, ali koju krajnji korisnik retko koristi (klik na link koji se ne upotrebljava često).
- Nizak prioritet i niska ozbiljnost – Kozmetičke greške, ili slovne greške u okviru paragrafa, ali ne na naslovnoj strani ili nazivu aplikacije.

Uobičajena klasifikacija defekata podrazumeva i sledeće parametre:

- Projektna aktivnost pri kojoj je defekt primećen (inspekcija, kodiranje, testiranje, itd.)
- Faza projekta u kom je defekt kreiran (ukoliko je to poznato) – kreiranje zahteva, dizajn, detaljni dizajn, kodiranje, itd.
- Faza projekta u kom je defekt primećen
- Mogući uzrok defekta – greška u zahtevima, dizajnu, interfejsu, kodu, podacima, itd.
- Mogućnost reprodukcije (ponavljanja) – desio se samo jednom, dešava se povremeno, potpuno reproducibilan, itd.
- Simptom – pucanje aplikacije, beskonačno dugo učitavanje ekrana, greška na korisničkom interfejsu, sistemska greška, problemi u performansama, itd.

Kada se defekt detaljnije istraži, moguća je dalja klasifikacija:

- Uzrok – greška koja je prouzrokovala defekt, recimo, greška u procesu, u kodu, korisnička greška, greška u testiranju, problem u konfiguraciji sistema, greška u podacima, problem u dokumentaciji, itd.
- Izvor – produkt u kom se desila greška, recimo, zahtevi, dizajn, arhitektura, dizajn baze, korisnička dokumentacija, dokumentacija vezana za testiranje, itd.
- Tip – logički problem, računarski problem, vremenski problem, rukovođenje podacima, poboljšanja, itd.

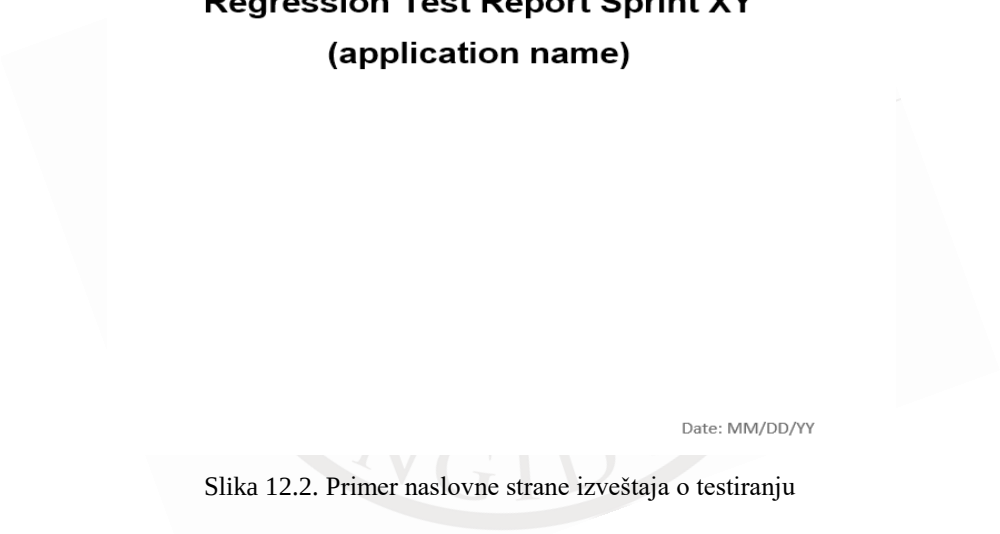
Kada je defekt popravljen (ili je njegova popravka odložena ili ga nije moguće reprodukovati), još informacija za dodatnu klasifikaciju je dostupno, kao što su:

- Rešenje – izmena u kodu, izmena u dokumentaciji, odlaganje, nije defekt, duplikat, itd.
- Akcija kojom je defekt ispravljen – pregled zahteva, pregled koda, jedinično testiranje, priprema podataka, dokumentacija o konfiguraciji, nema izmena, itd.
- Defekti se često sortiraju prema ozbiljnosti i prioritetu.

12.3. Izveštaj o testiranju

Kada se izvršavanje procesa testiranja završi, rezultati treba da se zabeleže, proslede zainteresovanim stranama (vođama projekata, klijentima, itd.) ili da se arhiviraju. Poznati defekti koji su prihvaćeni ili je njihova popravka odložena treba da budu navedeni u izveštaju kako bi ih korisnici imali u vidu prilikom korišćenja sistema. Izveštaj o testiranju treba kreirati nakon svakog izvršavanja određenog skupa testova i rezultati treba da budu jasno zavedeni.

Izveštaj obično ima predefinisanu formu. Naslovna strana izveštaja je najčešće veoma jednostavna, i sadrži tip testiranja, sprint u kome je testiranje izvedeno, ime aplikacije i datum izveštaja, kao što je prikazano na slici 12.3.



Regression Test Report Sprint XY **(application name)**

Date: MM/DD/YY

Slika 12.2. Primer naslovne strane izveštaja o testiranju

Uvodni deo izveštaja treba da sadrži sve relevantne podatke tog izvršavanja (slika 12.3):

- ime aplikacije koja se testira,
- iteraciju testiranja,
- relevantne informacije o softveru i hardveru,
- tip uređaja,
- datume testiranja,
- vrstu testiranja,
- podatke o okruženju,
- linkove ka skupu testova i rezultatima,
- imena lica odgovornih za testiranje, itd.

Regression Test Report

Iteration:	
Main Device Build Version:	
Main Device Environment:	
Referent Device Build Version:	
Referent Device Environment:	
Date:	
Testing Type:	
Test suite URL:	
Responsible tester(s):	

I. Execution Information

II. Results

[Graphic representation of Results]: Test Result Summary, Test Result Details (number of bugs found per test suite; only test suites which had bugs found in them are shown in the chart)

[Results analysis]

Slika 12.3. Uvodni deo izveštaja, sa informacijama o testiranju

Izveštaj obično sadrži grafički prikaz rezultata i trendova javljanja defekata. Ponekad može da sadrži poređenje sa rezultatima dobijenih tokom prethodnih izvršavanja istog skupa testova što može biti korisno za određivanje uticaja izmena u kvalitetu softvera tokom određenog vremenskog perioda.

Odeljak izveštaja sa informacijama o samom izvršavanju treba da sadrži podatke koji nisu navedeni u uvodu, ali su relevantni za to izvršavanje:

- broj izvršenih testova,
- broj neizvršenih testova sa obrazloženjima,
- bilo kakve promene u okruženjima,
- bilo kakve promene koje su mogle da utiču na izvršavanje testova, itd.

Odeljak sa rezultatima obično sadrži kratak sažetak rezultata i/ili link ka testovima u alatu za upravljanje testiranjem.

Odeljak gde su navedeni novi i aktivni defekti detektovani prilikom izvršavanja testova sadrži detaljne informacije o defektima:

- id defekta,
- naslov,
- stanje defekta
- reproducibilnost,
- ozbiljnost,
- verovatnoća pojavljivanja itd.

Primer odeljka sa listom defekata je prikazan na slici 12.4.

III. New bugs (bugs found in the latest build)

ID	Title	Work Item Type	Assigned To	State	Reproducibility	Probability	Severity

IV. Active bugs

- Bugs found in Sprint XYZ, build XYZ

ID	Title	Work Item Type	Assigned To	State	Reproducibility	Probability	Severity

V. Known Bugs (bugs already present in PROD)

- Bugs found in build XYZ

ID	Title	Work Item Type	Assigned To	State	Reproducibility	Probability	Severity

VI. Conclusion

Slika 12.4. Primer obrasca za listu novih, aktivnih i poznatih defekata

Zaključak sadrži mišljenje testera/test menadžera o kvalitetu softvera, doneto na osnovu ukupnih test rezultata i uticaja pronađenih defekata na ponašanje softvera. Ponekad izveštaj može da sadrži informacije o tome kada je zakazano sledeće izvršavanje testova. Nakon završetka, izveštaj se šalje svim zainteresovanim stranama kako bi bila omogućena transparentnost procesa i kako bi bilo omogućeno planiranje narednih akcija. Izveštaj o testiranju je stoga jedan od najvažnijih projektnih dokumenata. Mogući izgled kompletnog obrasca za izveštaj o testiranju je prikazan na slici 12.5.

Regression Test Report Sprint XY

(application name)

Date: MM/DD/YY

Regression Test Report

Iteration:

Main Device Build Version:

Main Device Environment:

Referent Device Build Version:

Referent Device Environment:

Date:

Testing Type:

Test suite URL:

Responsible tester(s):

I. Execution Information

II. Results

(Graphic representation of Results): Test Result Summary, Test Result Details (number of bugs found per test suite, only test suites which had bugs found in them are shown in the chart)

(Results analysis)

III. New Bugs (bugs found in the latest build)

ID	Title	Work Item Type	Assigned To	State	Reproducibility	Probability	Severity

IV. Active bugs

- Bugs found in Sprint XYZ, build XYZ

ID	Title	Work Item Type	Assigned To	State	Reproducibility	Probability	Severity

V. Known Bugs (bugs already present in PROD)

- Bugs found in build XYZ

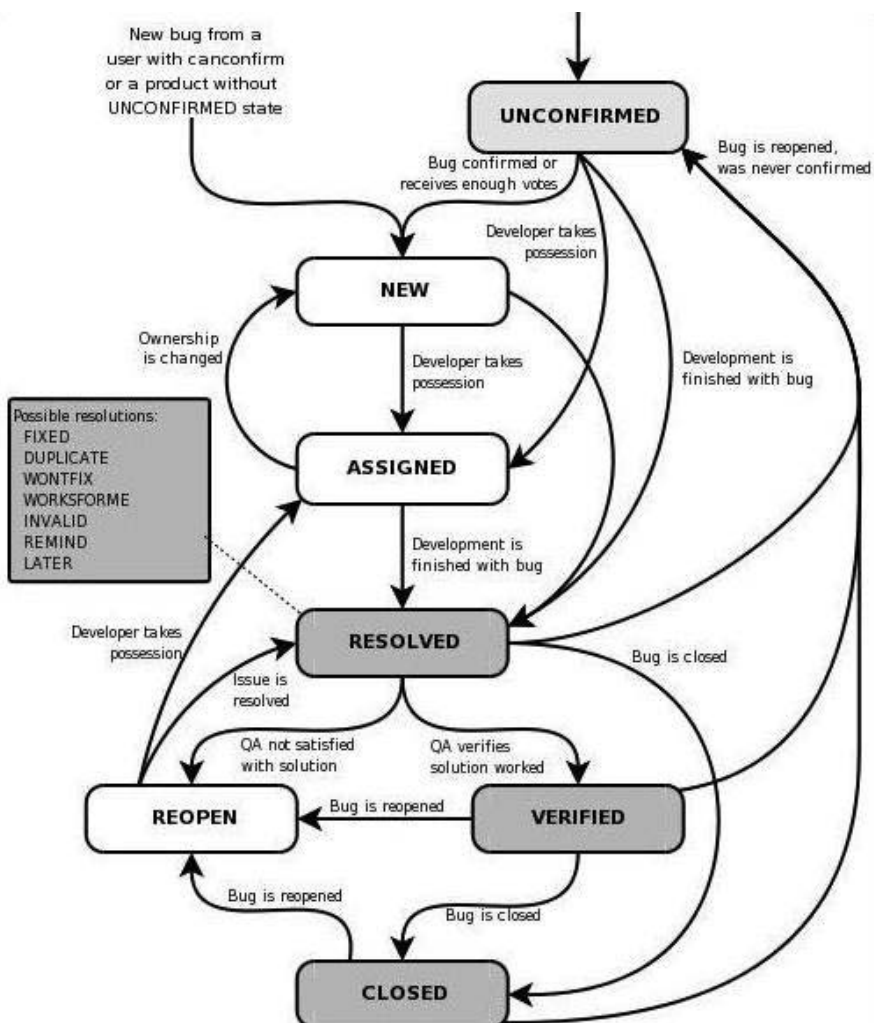
ID	Title	Work Item Type	Assigned To	State	Reproducibility	Probability	Severity

VI. Conclusion

Slika 12.5. Kompletan obrazac za izveštaj o testiranju

12.4. Životni ciklus defekta

Kada se bilo koji defekt uoči za vreme testiranja, on će proći kroz životni ciklus defekta koji se sastoji od konačnog broja stanja u kojima se defekt može nalaziti. Na slici 12.6 prikazan je životni ciklus defekta u alatu Bugzilla.



Slika 12.6. Životni ciklus defekta u alatu Bugzilla

Ceo proces počinje onog trenutka kada se defekt uoči. Prvo je potrebno zavesti bug u alatu za praćenje defekta. Nakon zavođenja, bug se dodeljuje programeru, a status se postavlja na *Open*. Programer tada može da pregleda bug, da ga reprodukuje i da počne da radi na popravci.

Kada popravi bug, programer menja status na *Fixed*, ili u drugim slučajevima može da postavi status na neku od opcija alata poput *Need more information*, *Cannot reproduce* i slično. Nakon što je bug popravljen, QA će izvršiti regresiono testiranje i verifikovati da li je bug zaista popravljen ili ne. Ukoliko je bug popravljen, QA postavlja status na *Verified/Closed*, dok u suprotnom postavlja status na *Reopen*, odnosno ponovo otvara bug i dodeljuje ga programeru.

Kompletna lista statusa u kojima se defekt može pronaći je data u nastavku:

- *New* – kada se defekt prijavi prvi put.
- *Assigned* – nakon što tester prijavi bug, vođa QA tima ga pregleda i dodeljuje odgovarajućem programerskom timu.
- *Open* – bug ostaje u ovom stanju dok programer ne izvrši odgovarajući task nad tim bugom.
- *Resolved/Fixed* – kada programer popravi bug, postavlja ovaj status.
- *Verified/Closed* – tester proverava da li je bug zaista ispravljen, i ukoliko jeste postavlja status *Verified/Closed*.
- *Reopen* – ukoliko tester utvrdi da bug nije zaista popravljen, bug se ponovo otvara.
- *Not a Bug/Invalid* – programer može postaviti ovaj status ukoliko prijavljeni bug zapravo nije greška, već očekivan rad aplikacije.
- *Deffered* – ukoliko je bug niskog prioriteta, a nema dovoljno vremena za ispravku, on se odlaže za sledeći *release*.
- *Cannot Reproduce* – ukoliko programer ne može da reprodukuje bug sa svoje strane prateći korake koje je tester zaveo u opisu buga.

Pitanja

1. Šta je menadžment defekta?
2. Zbog čega se defekti zavode u odgovarajući sistem za praćenje defekata?
3. Navedi neke popularne sisteme za praćenje defekata.
4. Šta treba da sadrži izveštaj o defektu?
5. Šta znači da izveštaj o defektu mora da bude upotrebljiv?
6. Kako se defekti mogu klasifikovati?
7. Koja je razlika između pojmova prioritet i ozbiljnost defekta?
8. Šta sadrži izveštaj o testiranju?
9. Šta je životni ciklus defekta?
10. U kojim stanjima se može pronaći defekt u toku svog životnog ciklusa?



LITERATURA

1. Jorgensen P. C., *Software Testing: A Craftsman's Approach*, 4th Edition, AUERBACH, 2013.
2. Myers G. J., Sandler C., Badgett, T., *The Art of Software Testing*, 3rd Edition, Wiley Publishing, 2011.
3. Amman P., Offutt J., *Introduction to Software Testing*, 1st Edition, Cambridge University Press, New York, NY, USA, 2008.
4. Graham D., Van Veenendaal E., Evans I., Black R., *Foundations of Software Testing: ISTQB Certification*, Intl Thomson Business Pr., 2008.
5. Patton R., *Software Testing*, 2nd Edition, Sams, Indianapolis, IN, USA 2005.
6. Crispin L., Gregory J., *Agile Testing: A Practical Guide for Testers and Agile Teams*, 1st Edition, Addison-Wesley Professional, 2009.
7. Jalote P., *An Integrated Approach to Software Engineering*, 3rd Edition, Springer Publishing Company, 2010.
8. Young M., Pezze M., *Software Testing and Analysis: Process, Principles and Techniques*, John Wiley & Sons, Inc., USA, 2005.
9. Nielson F., Nielson H.R., Hankin C., *Principles of Program Analysis*, Springer, 2nd edition, 2005
10. Tahchiev P., Leme F., Massol V., Gregory G., *JUnit in Action*, 2nd Edition, Manning Publications Co., Greenwich, CT, USA, 2010.
11. Cohn M., *Agile Estimating and Planning*, Prentice Hall PTR, 2007.
12. Schwaber K., *Agile Project Management With Scrum*, Microsoft Press, 2004.
13. Garg N., *Test Automation Using Selenium Webdriver with Java: Step by Step Guide* 1st Edition, AdactIn Group Pty Ltd., 2014.
14. IEEE 729-1983 - IEEE Standard Glossary of Software Engineering Terminology: <https://standards.ieee.org/standard/729-1983.html>
15. ISTQB Glossary: <http://glossary.istqb.org/>
16. Bugzilla 5.1.2. Guide zvanični sajt: <https://bugzilla.readthedocs.io/en/latest/using/> 2018.
17. JUnit zvanični sajt: <https://junit.org/>
18. ISTQB Foundation Level and Agile Tester Certification guide sajt: <http://tryqa.com/>
19. Agile Alliance zvanični sajt: <https://www.agilealliance.org/>
20. International Software Testing Qualifications Board zvanični sajt: <https://www.istqb.org/>
21. WestFall Team zvanični sajt: <http://www.westfallteam.com/>
22. Selenium zvanični sajt: <https://www.seleniumhq.org/>
23. Scrum zvanični sajt: <https://www.scrum.org/resources/what-is-scrum>
24. Kanban: <https://kanbanize.com/kanban-resources/getting-started/what-is-kanban/>
25. Software Testing Fundamentals sajt: <http://softwaretestingfundamentals.com/>
26. British Computer Society Specialist Interest Group in Software Testing, BS7925-2 Standard for Software Component Testing: <http://www.ruleworks.co.uk/testguide/BS7925-2.htm>

27. McCabe T.J., *A Complexity Measure*, IEEE Transactions on Software Engineering, 2(4), December 1976, 308-320
28. Bhattacharya S., Kanjilal A., *Static Analysis of Object Oriented Systems Using Extended Control Flow Graph*, IEEE, proceedings of TENCON 2004, 2004, 310-313
29. Bhattacharya S., Kanjilal A., *Code Based Analysis For Object Oriented Systems*, J. Comput. Sci. Technology, 21(6), 2006, 965-972.
30. Guru99 sajt: <https://www.guru99.com/software-testing.html>



CIP - Каталогизација у публикацији - Народна библиотека Србије, Београд

004.415.53(075.8)

ЖИВКОВИЋ, Миодраг, 1982-

Testiranje softvera / Miodrag Živković. - 1. izd. - Beograd :

Univerzitet Singidunum, 2018 (Loznica : Mobid). - XIII, 246 str. : ilustr.

; 24 cm

Tiraž 300. - Bibliografija: str. 245-246.

ISBN 978-86-7912-680-1

а) Софтвер - Тестирање

COBISS.SR-ID 267717132

© 2018.

Sva prava zadržana. Nijedan deo ove publikacije ne može biti reprodukovан u bilo kom vidu i putem bilo kog medija, u delovima ili celini bez prethodne pismene saglasnosti izdavača.



Miodrag Živković

TESTIRANJE SOFTVERA

Pojam testiranja softvera je star koliko i samo programiranje. Još od prvih napisanih programa je bilo neophodno proveriti da li se program ponaša ispravno i na takav način kako je definisano. U današnje vreme, iako možda to na prvi pogled ne izgleda tako, naš način života u potpunosti zavisi od softvera koji je ključni faktor u velikom boju sistema koje svakodnevno koristimo.

Testiranje softvera je integralni deo procesa razvoja softvera, sa ciljem osiguravanja kvaliteta finalnog proizvoda. U ljudskoj prirodi je da se prave greške. Neke greške mogu biti sitne i nebitne, dok druge mogu biti veoma ozbiljne i skupe, pa čak i dovesti do ljudskih žrtava. Testiranje softvera je neophodno kako bi se otkrile greške načinjene u svim fazama razvoja softvera, koje se nakon toga mogu ispraviti. Otkrivanjem i ispravljanjem grešaka se obezbeđuje viši nivo kvaliteta softvera, čime se stiče poverenje i zadovoljstvo krajnjih korisnika softvera. Većina osnovnih koncepata testiranja softvera se može primeniti na veliki broj potpuno različitih tipova softvera različitih namena. Cilj ove knjige je da prikaže ove koncepte na takav način da čitaoci mogu lako da ih primene u bilo kojoj situaciji koja se može javiti u testiranju softvera.

U modernom razvoju softvera, zanimanje softver tester je veoma cenjeno i traženo. Neke ideje i tehnike testiranja softvera su postale neophodan skup veština koje mora da poseduje i dobar programer. Od svakog programera se očekuje da testira svoj kod, i to je standard u industriji, pa je ova knjiga namenjena i njima. Podrazumeva se da čitaocima knjige ovo nije prvi susret sa programiranjem u Javi, tj. očekuje se osnovno poznavanje programskog jezika Java pošto su svi praktični primeri implementirani u alatu JUnit.