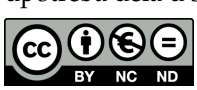
Verifikacija softvera

Milena Vujošević Janičić

21. jul 2025.

Matematički fakultet, Univerzitet u Beogradu

Verifikacija softvera



Izdavač
Matematički fakultet, Univerzitet u Beogradu. Studentski trg 16, Beograd.
Za izdavača: *prof. dr XX*, dekan
Decembar, 2025.

– Mami

Predgovor

XXX

Milena Vujošević Janičić

Sadržaj

Spisak slika

Spisak tabela

Ispravnost i neispravnost softvera

Kvalitet softvera 1

Pregled

- Koji procesi su važni za kvalitet softvera?
- Koji su standardi kvaliteta softvera najbitniji i šta oni definišu?
- Kojim se atributima opisuje kvalitet softvera?
- Koji atributi kvaliteta softvera su važni za bankarske aplikacije, koji za autonomnu vožnju, koji za klakulator, koji za onlajn prodaju karata, a koji za sisteme za razmenu poruka?

Tokom poslednjih godina, IT industrija se brzo razvija i predstavlja jednu od najdinamičnijih industrija u svetu. Softver se razvija za veoma raznovrsne uređaje i svrhe. Ove namene obuhvataju oblasti poput interneta stvari, virtuelne i proširene stvarnosti, igara i zabave, pametnih okruženja i zdravstvene zaštite. Takođe, softver ima ključnu ulogu u razvoju veštačke inteligencije, obradi velikih podataka, poslovnim, naučnim i vojnim primenama, kao i u savremenim komunikacionim tehnologijama.

Slika 1.1: Softver je svuda oko nas. Počevši od malih kućnih aparata, preko telefona, prevoznih sredstava, satelita, raketa, aprata u zdravstvu — gotovo da više ne postoji elektronski uređaj koji u sebi ne sadrži nekakav softver.

Razvoj softvera je složen proces koji obuhvata veliki broj različitih aktivnosti, od kojih se osnovne mogu grupisati u:

- analizu sistema i specifikaciju zahteva,
- projektovanje i implementaciju softvera,
- upravljanje kvalitetom softvera i
- održavanje softvera.

Sa sve većim obimom proizvodnje softvera, raste i potreba za efikasnijim upravljanjem procesima njegove izrade. U procesu razvoja softvera, ključno je da se dostupni resursi optimalno iskoriste kako bi se na vreme zadovoljili korisnički zahtevi i obezbedio visok kvalitet softverskog proizvoda.

1.1 Upravljanje kvalitetom softvera

Upravljanje kvalitetom softvera (eng. *quality management*) obuhvata principe, metode i procese koji se koriste za obezbeđivanje i unapređivanje kvaliteta softvera. Cilj upravljanja

kvalitetom je da se ispune zahtevi korisnika, optimizuju poslovni procesi i smanji broj grešaka (defekata).

Visok kvalitet softvera predstavlja ključni faktor njegove uspešnosti, bilo da se koristi u komercijalne svrhe na tržištu ili u okviru specifičnih, nekomercijalnih okruženja. Standardi kvaliteta softvera definišu smernice i propise koji omogućavaju sistematski pristup upravljanju kvalitetom. Pored toga, za postizanje visokokvalitetnog softvera neophodna je automatizovana podrška i upotreba sofisticiranih alata.

Osnovni procesi koji su vezani za kvalitet softvera uključuju

- planiranje kvaliteta softvera,
- obezbeđivanje (eng. *assurance*) kvaliteta softvera,
- kontrolu (eng. *control*) kvaliteta softvera i
- poboljšanje kvaliteta softvera.

Planiranje kvaliteta softvera je neophodno kako bi se definisao pristup razvoju softvera koji omogućava postizanje željenog nivoa kvaliteta. Na primer, različiti nivoi kvaliteta softvera očekuju se za softver aviona i za igricu na mobilnom telefonu. Obezbeđivanje kvaliteta softvera podrazumeva uključivanje aspekata kvaliteta u svakodnevni razvoj, dok kontrola treba da obezbedi da dobijeni krajnji proizvod bude željenog kvaliteta. Na kraju, poboljšanje kvaliteta podrazumeva kontinuirano praćenje i unapređenje procesa razvoja softvera kroz analizu povratnih informacija i primenu novih metodologija razvoja.

1.2 Standardi

Željeni kvalitet softvera definiše se softverskim zahtevima, ali može biti nametnut i različitim međunarodnim standardima. Serija standarda ISO/IEC 25000 sadrži okvir za procenu kvaliteta softvera. Najvažniji standard u ovoj seriji je ISO/IEC 25010.

Slika 1.2: ISO/IEC 25010.

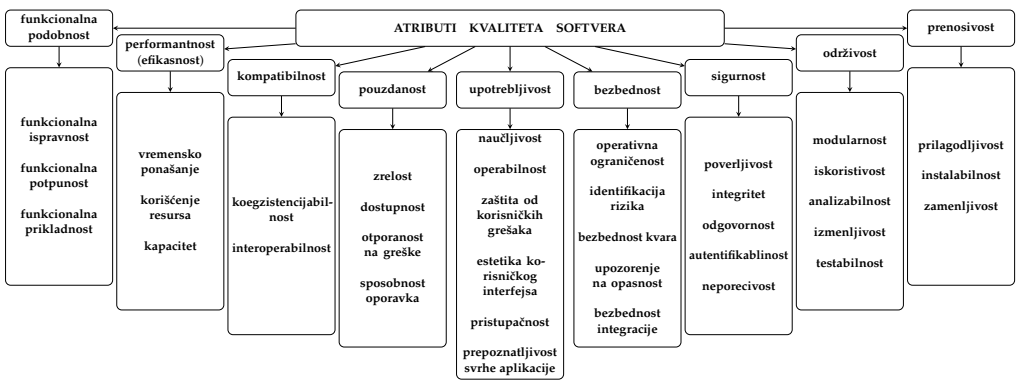
Ovaj standard definiše devet karakteristika kvaliteta softvera, koje se obično nazivaju atributima kvaliteta softvera. Standard ISO/IEC 25023 opisuje kako se ove karakteristike kvaliteta koriste za merenje ukupnog kvaliteta proizvoda.

Postoje i važni IEEE standardi koji se široko koriste. Na primer, standard IEEE 730 daje smernice za pokretanje, planiranje, kontrolu i izvršavanje procesa obezbeđenja kvaliteta softvera,

dok standard IEEE 1012-2016 definiše procese verifikacije i validacije za razvoj sistema, softvera i hardvera.

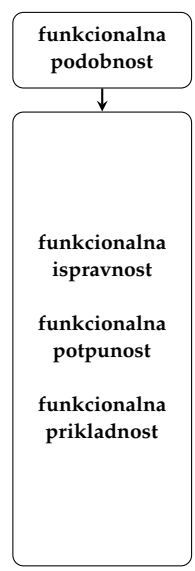
1.3 Atributi kvaliteta softvera

Standardom ISO 25010 definiše hijerarhiju devet atributa kvaliteta (slika 1.3): funkcionalna podobnost, performantnost, kompatibilnost, pouzdanost, upotrebljivost, bezbednost, sigurnost, održivost i prenosivost. Svaki atribut uključuju svoje podatribute. U zavisnosti od svrhe i ciljeva softvera, svaki atribut kvaliteta softvera može imati različit nivo važnosti.



Slika 1.3: Atributi softvera u skladu sa kategorizacijom standarda ISO/IEC 25010.

1.3.1 Funkcionalna podobnost



Funkcionalna podobnost (eng. *functional suitability*) odgovara stepenu u kojem softver ispunjava funkcionalne zahteve. Funkcionalna podobnost obuhvata naredne podatribute:

funkcionalnu ispravnost — softver treba da daje tačne rezultate,

funkcionalnu potpunost — dostupnost funkcija koje su očekivane specifikacijom i

funkcionalnu prikladnost — ispunjavanje očekivane funkcionalnosti.

Primer 1.3.1 (Kalkulator) Razmotrimo primer jednostavnog kalkulatora.

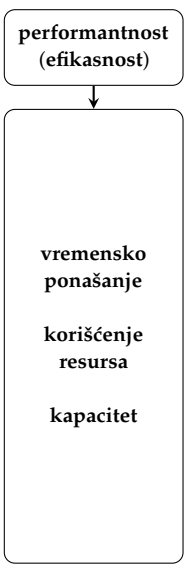
Funkcionalna ispravnost podrazumeva da kalkulator uvek računa ispravne vrednosti. Na primer, kalkulator treba da ne greši u sabiranju dva broja.

Funkcionalna potpunost podrazumeva da kalkulator ima dostupne sve funkcionalnosti predviđene specifikacijom. Na primer, željene funkcionalnosti mogu da budu sabiranje, oduzimanje, množenje i deljenje.

Funkcionalna prikladnost podrazumeva da se kalkulator ponaša na očekivani način. Na primer, kalkulator ne treba da ima neočekivane dodatne opcije kao što su skidanje sadržaja sa interneta ili puštanje filmova.

1.3.2 Performantnost

Performantnost (efikasnost) (eng. *performance*) odgovara stepenu u kojem softver zadovoljava vremenske i resursne zahteve. Performantnost obuhvata naredne podatribute:



vremensko ponašanje — vreme odgovora i obrade (koliko vremena je potrebno da aplikacija da odgovor korisniku, odnosno da obradi neke netrivijalne zahteve), stopa protoka (koliko transakcija je aplikacija u mogućnosti da obradi u jedinici vremena),

korišćenje resursa — količina i vrsta korišćenih resursa i

kapacitet — maksimalna ograničenja (u kontekstu broja korisnika ili veličine ulaza koje aplikacija može da obradi).

Primer 1.3.2 (Kupovina karata) Razmotrimo primer sistema za onlajn rezervaciju i kupovinu avionskih karata. Ponašanje ovog sistema treba razmotriti u odnosu na svakog pojedinačnog korisnika sistema, ali i u kontekstu mogućeg velikog opterećenja sistema u situacijama kada veliki broj korisnika istovremeno želi da rezerviše ili kupi kartu.

Vremensko ponašanje ovog sistema uključuje, pored ostalog:

vreme odgovora (odziva) aplikacije pri pretraživanju dostupnih karata i

vreme obrade potrebno za rezervaciju ili plaćanje karte.

Na primer, očekivanja koja se postavljaju visokoefikasnom sistemu su da:

pretraga karata treba da se izvrši za manje od 2 sekunde u 97% slučajeva,

prikaz rezultata pretrage (lista dostupnih karata) mora da se učita u roku od 3 sekunde za 95% zahteva,

plaćanje i potvrda rezervacije ne smeju trajati duže od 5 sekundi u 99% slučajeva.

U ovakvom sistemu često je prisutna i komunikacija sa spoljnim sistemima (kao što su baze podataka ili eksterni API servisi za plaćanje), te u tom kontekstu može se postaviti uslov da kašnjenje odgovora eksternih sistema ne sme biti veće od 1 sekunde. Relevantni resursi čiju upotrebu treba pratiti za ovu aplikaciju su procesor, memorija, baza podataka i mrežni resursi. Na primer, očekivanja koja mogu da se postave su:

opterećenje procesora ne sme preći 70% pri normalnom radu i ne sme preći 90% pri vršnom opterećenju duže od 10 sekundi,

potrošnja RAM memorije ne sme preći 8 GB po instanci aplikacije tokom normalnog rada,

upotreba baze podataka mora da omogući obradu do 500 upita u sekundi bez značajnog usporavanja (tj. odgovor baze uvek mora da bude ispod 200ms) i

potrošnja mrežnih resursa ne sme premašiti 1 GB po sekundi na pojedinačnim serverima, pri čemu se ne sme desiti gubitak podataka.

Kapacitet aplikacije definiše koliko istovremenih korisnika aplikacija može da podrži pre nego što performanse padnu ispod prihvatljivog nivoa. Na primer, sistem mora da podrži najmanje 10.000 istovremenih korisnika bez degradacije performansi.

1.3.3 Kompatibilnost

Kompatibilnost (eng. *compatibility*) odgovara stepenu u kojem softver može da funkcioniše na različitim platformama ili da deli podatke sa drugim proizvodima, sistemima i komponentama. Kompatiblinost obuhvata naredne podatribute:

koegzistencijabilnost — sposobnost deljenja okruženja i resursa sa drugim softverskim proizvodima i

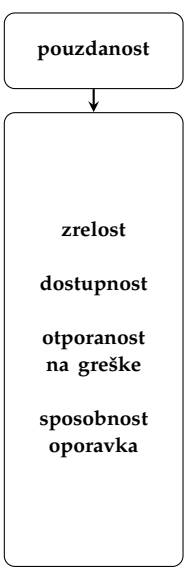
interoperabilnost — sposobnost promene i korišćenja informacija sa/iz drugih aplikacija.

Primer 1.3.3 (Razmena poruka) Razmotrimo primer aplikacije za razmenu poruka (na primer, aplikaciju nalik na *WhatsApp*, *Viber* ili *Slack*). Ova aplikacija mora da funkcioniše zajedno sa drugim aplikacijama na uređaju (koegzistencijabilnost) i treba da može da komunicira sa drugim servisima (interoperabilnost).

Na primer, korisnik na svom telefonu koristi aplikaciju za razmenu poruka dok istovremeno sluša muziku putem aplikacije *Spotify* ili gleda video preko aplikacije *YouTube*. Koegzistencijabilnost podrazumeva da aplikacija ne sme ometati druge aplikacije. Na primer, ako korisnik primi poziv, muzika treba da se automatski pauzira, ali i da nastavi nakon završetka poziva. Notifikacije koje aplikacija pruža ne smeju blokirati druge aplikacije ili ometati korisničko iskustvo sa drugim aplikacijama.

Interoperabilnost podrazumeva da aplikacija treba da, na primer, omogući korisnicima da šalju slike iz galerije telefona ili sa *Google Drive*-a. To znači da ima pristup lokalnoj memoriji ili eksternim servisima.

1.3.4 Pouzdanost



Pouzdanost (eng. *reliablity*) označava stepen u kojem je softver pouzdan. Pouzdanost obuhvata naredne podatribute:

zrelost — stabilnost tokom svakodnevne upotrebe,
dostupnost — mogućnost neprekidnog rada,
toleranciju na greške — sposobnost funkcionisanja čak i u prisustvu određenih hardverskih i softverskih kvarova i
sposobnost oporavka (povratljivost) — sposobnost vraćanja podataka i procesa u slučaju kvara sistema.

Primer 1.3.4 (Hitne službe) Razmotrimo primer sistema za hitne službe (kao što su pozivi policiji ili prvoj pomoći). Za ovakve sisteme kritično je da zadovolje sve kriterijume pouzdanosti. Pre svega, sistem mora biti dobro proveren i stabilan, a svaka nova funkcionalnost mora biti temeljno

testirana i verifikovana pre uvođenja — zrelost.

Sistem mora biti neprekidno dostupan, što znači da nikada ne sme biti van funkcije — dostupnost sistema. To podrazumeva mogućnost simultanog prijema velikog broja poziva, kao i postojanje dodatne infrastrukture u slučaju pada primarnog sistema ili njegovih delova.

Ukoliko na nekoj serverskoj lokaciji dođe do prekida napajanja, sistem mora nastaviti sa radom bez vidljivih problema. Posao tog servera treba automatski da bude preusmeren na drugi dostupni server, čime se osigurava kontinuitet rada i sprečava gubitak podataka — tolerancija na greške.

Takođe, bitno je da sistem u slučaju softverskog pada može automatski da se oporavi i nastavi sa radom — sposobnost oporavka.

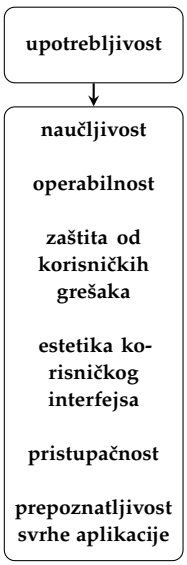
1.3.5 Upotrebljivost

Upotrebljivost (eng. *usability*) odgovara stepenu u kojem korisnici mogu koristiti softver. Upotrebljivost obuhvata naredne podatribute:

naučljivost — koliko je jednostavno naučiti kako softver funkcioniše,
operabilnost — koliko je lako raditi sa softverom i kontrolisati ga,
zaštitu od korisničkih grešaka — mere u okviru samog softvera koje sprečavaju pravljenje grešaka u korišćenju,
estetiku korisničkog interfejsa — vizuelnu prijatnost i prihvatljivost dizajna,
pristupačnost — mogućnost korišćenja softvera od strane osoba sa različitim sposobnostima i
prepoznatljivost svrhe aplikacije — jasnoću namene softvera korisnicima.

Primer 1.3.5 (Bankarska aplikacija) Razmotrimo primer bankarske aplikacije. Ova aplikacija mora biti intuitivna, jednostavna za korišćenje, sigurna i pristupačna svim korisnicima, uključujući i osobe sa posebnim potrebama. Na primer:

Naučljivost. Prilikom prvog korišćenja, korisnik bi trebalo da može samostalno da pronađe i izvrši osnovne radnje (poput slanja novca i provere stanja) za manje od 5 minuta,



bez potrebe za tutorijalom.

Operabilnost. Ključne funkcionalnosti (poput provere stanja ili prenosa sredstava između računa) treba da budu dostupne u manje od četiri klika, omogućavajući brz i lak rad sa aplikacijom.

Zaštita od korisničkih grešaka. Ako korisnik unese neispravan broj računa prilikom slanja novca, aplikacija mora automatski izvršiti validaciju i upozoriti korisnika na grešku pre potvrde transakcije.

Estetika korisničkog interfejsa. Aplikacija treba da ima profesionalan i vizuelno prijatan dizajn. Dizajn bi trebalo da bude minimalistički, moderan i intuitivan, sa jasnim kontrastom između elemenata. Treba koristiti harmonične boje i lako čitljive fontove, dok dugmad i ikonice moraju biti vizuelno jasne i prepoznatljive.

Pristupačnost. Aplikacija mora biti pristupačna osobama sa različitim potrebama, uključujući korisnike sa oštećenjem vida. To znači da treba da bude kompatibilna sa alatima za čitanje ekrana, da fontovi mogu biti uvećani po potrebi i da kontrast boja omogućava čitanje osobama sa daltonizmom.

Prepoznatljivost svrhe aplikacije. Korisnik odmah treba da razume da je aplikacija namenjena bankarskim uslugama. Početni ekran treba da sadrži jasno prepoznatljive elemente, poput salda računa i dugmeta za transfer novca. Ikonice i nazivi funkcionalnosti treba da budu intuitivni i nedvosmisleni (npr. „Plaćanje“, „Kartice“, „Računi“), a brend-boje i logo banke istaknuti.

Bezbednost i sigurnost se u srpskom jeziku mogu koristiti kao sinonimi. Zbog toga je bitno napomenuti da se u kontekstu atributa kvaliteta softvera bezbednost odnosi na **fizičku** bezbednost/sigurnost, dok se sigurnost koristi u kontekstu **informacione** bezbednosti/sigurnosti.

bezbednost (eng. *safety*)
=
fizička bezbednost

sigurnost (eng. *security*)
=
informaciona bezbednost

1.3.6 Bezbednost

Bezbednost (eng. *safety*) odgovara stepenu u kojem proizvod, pod definisanim uslovima, izbegava stanje u kojem su ugroženi ljudski život, zdravlje, imovina ili životna sredina. Bezbednost obuhvata naredne podatribute:

operativnu ograničenost — stepen do kojeg proizvod ili sistem ograničava svoje funkcionisanje unutar bezbednih parametara ili stanja prilikom susreta sa operativnim opasnostima,

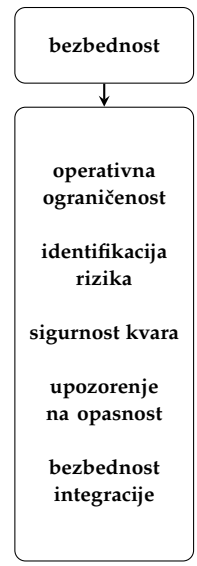
identifikacija rizika — stepen do kojeg proizvod može da identifikuje tok događaja ili operacija koji mogu izlo-

žiti život, imovinu ili životnu sredinu neprihvatljivom riziku,

bezbednost kvara — Stepen do kojeg proizvod može automatski da se postavi u bezbedan režim rada ili da se vrati u bezbedno stanje u slučaju kvara,

upozorenje na opasnost — Stepen do kojeg proizvod ili sistem pruža upozorenja o neprihvatljivim rizicima u operacijama ili internim kontrolama, kako bi se omogućila pravovremena reakcija i održavanje bezbednog rada i

bezbednost integracije — Stepen do kojeg proizvod može da održi bezbednost tokom i nakon integracije sa jednom ili više komponenti.



Primer 1.3.6 (Autonomna vožnja) U sistemu za autonomnu vožnju svi atributi sigurnosti su izuzetno važni.

Operativna ograničenost — Automobil mora ograničiti svoje manevre (npr. brzinu, skretanje) u skladu sa bezbednim parametrima, posebno pri lošim vremenskim uslovima ili oštećenjima na putu.

Identifikacija rizika — Sistem treba da ume da prepozna potencijalne opasnosti kao što su pešaci koji iznenada prelaze ulicu, druga vozila koja se kreću nepredvidivo ili prepreke na putu.

Bezbednost kvara — U slučaju otkaza senzora ili računarskog sistema, automobil mora automatski da se zaustavi na bezbedan način ili da preuzme minimalno funkcionalno stanje koje ne ugrožava putnike ili druge učesnike u saobraćaju.

Upozorenje na opasnost — Ako se pojavi situacija koju sistem ne može da reši sam, mora brzo upozoriti vozača ili centralni nadzorni sistem na opasnost.

Bezbednost integracije — Kada se novi moduli (npr. dodatni senzori ili softverske nadogradnje) integrišu u postojeći sistem, ceo sistem mora i dalje održavati visok nivo bezbednosti i ne sme uvoditi nove rizike.

1.3.7 Sigurnost

Sigurnost (eng. *security*) odgovara stepenu u kojem softver štiti informacije i podatke. Sigurnost obuhvata naredne po-

datribute:

poverljivost — dostupnost podataka samo ovlašćenim korisnicima,
integritet — sprečavanje neovlašćenog pristupa i modifikacije podataka,
odgovornost — mogućnost praćenja radnji korisnika,
autentifikabilnost — mogućnost dokazivanja identiteta korisnika i
neporecivost — nemogućnost osporavanja, tj. mogućnost prikupljanja informacija o određenim aktivnostima i događajima.

Primer 1.3.7 (Bankarska aplikacija — nastavak) Razmotrimo dalje primer 1.3.5. Bankarske aplikacije moraju biti maksimalno sigurne, jer rukovode osetljivim finansijskim podacima korisnika i omogućavaju transakcije koje ne smeju biti zloupotrebljene. Bankarska aplikacija je odličan primer aplikacije kod koje je kritična bezbednost jer je potrebno:

- Održavati poverljivost — podaci korisnika moraju biti maksimalno zaštićeni. Na primer, ukoliko korisnik može da pristupi svojoj mobilnoj banci i pregleda stanje na računu — niko drugi ne sme videti ove informacije. To uključuje da pristup aplikaciji mora biti zaštićen lozinkom, PIN-om ili biometrijom (otisak prsta, prepoznavanje lica), podaci moraju biti šifrovani tokom prenosa i skladištenja, a sama aplikacija ne sme dozvoliti snimanje ekrana kako bi se sprečilo curenje podataka.
- Garantovati integritet — transakcije ne smeju biti neovlašćeno menjane. Na primer, ukoliko korisnik izvrši uplatu putem aplikacije, transakcija mora biti tačna i niko ne sme neovlašćeno izmeniti iznos uplate. Banka mora da šifruje podatke tokom prenosa, svaka transakcija mora imati digitalni potpis koji potvrđuje da nije menjana, a sistem mora detektovati bilo neovlašćene izmene podataka i odmah ih blokirati.
- Obezbediti odgovornost — sve aktivnosti se beleže i mogu se proveriti. Na primer, ukoliko se korisnik žali da neko neovlašćeno koristi njegov račun – banka mora biti u stanju da proveri ko je koristio aplikaciju i kada.

- Osigurati autentifikabilnost — samo pravi korisnik može pristupiti svom računu. Banka mora biti sigurna da se korisnik koji pokušava da se prijavi u aplikaciju zaista identifikuje kao pravi vlasnik računa. Ovo se može ostvariti korišćenjem višefaktorske autentifikacije koja uključuje na primer lozinku, biometriju i jednokratne kodove. Dodatno, prijava preko novog uređaja treba da zahteva dodatnu verifikaciju putem elektronsk epošte ili telefonskog poziva.
- Omogućiti neporecivost — korisnik ne može kasnije tvrditi da nije izvršio transakciju. To se može ostvariti digitalnim potpisivanjem kriptografskim ključevima. Pri tome, logovi moraju biti nepromenljivi, tako da niko, pa ni administratori, ne mogu izbrisati ili izmeniti istoriju transakcija.

1.3.8 Održivost

Održivost (eng. *maintainability*) odgovara lakoći integrisanja promena u softver. Promene uključuju dodavanje novih funkcionalnosti, ispravljanje grešaka i poboljšanje performansi, kao i prilagođavanje promenjenim zahtevima korisnika ili izmenjenom okruženju. Izmena softvera zahteva:

1. razumevanje softvera;
2. pronalaženje delova softvera koje je potrebno izmeniti;
3. izvođenje željenih izmena;
4. proveru da izmene nisu poremetile postojeći kôd.

Održivost se odnosi na lakoću svih ovih koraka. Smatra se jednim od ključnih atributa kvaliteta.

Održivost je statička karakteristika softvera koja ne utiče direktno na performanse i karakterisitke softvera koje su vidljive krajnjem korisniku. Iako ne utiče direktno na ponašanje sistema u radu, ima značajan uticaj na dugoročnu kvalitetu i održavanje softvera. Primeri metrika softvera koje su važne u kontekstu održivosti su:

spregnutost (eng. *coupling*) – kvantitativna mera međuzavisnosti između različitih modula,

kohezija (eng. *cohesion*) – kvantitativna mera povezanosti funkcija ili objekata unutar istog modula,

ciklomatska složenost (eng. *cyclomatic complexity*) – kvantitativna mera broja linearno nezavisnih putanja kontrole

toka programa, izračunava se po formuli:

$$C = E - N + 2P$$

gde je:

- E broj grana (eng. *edges*) u grafu kontrole toka programa,
- N broj čvorova (eng. *nodes*),
- P broj povezanih komponenti (za većinu pojedinačnih funkcija $P = 1$);

veličina (eng. *size*) – broj linija koda.

Naime, niska spregnutost, visoka kohezija, niska ciklomatska složenost i manja veličina su karakteristike održivog softvera. Kako se dizajn softvera i njegov ukupni kvalitet vremenom pogoršavaju, neophodno je primenjivati različite tehnike za očuvanje održivosti. Refaktorisanje softvera predstavlja proces poboljšanja dizajna postojećeg koda i ključni je deo održavanja tokom evolucije softvera. Održivost uključuje naredne podatribute: modularnost, iskoristivost, analizabilnost, izmenljivost i testabilnost.

Modularnost se odnosi na stepen u kome je softver logički podeljen na nezavisne i međusobno zamenljive module. Razbijanje softvera na module (jedinice, komponente) omogućava skrivanje njegove ukupne složenosti kroz apstrakciju i definisanje interfejsa. Modularnost se obično postavlja kao jedan od ključnih ciljeva u fazi dizajna softvera.

Idealan modul treba da bude:

- relativno male dimenzije,
- niske ciklomatske složenosti,
- visoke kohezije,
- minimalno spregnut sa ostalim modulima.

Ovakvi moduli se onda mogu nezavisno odvajati i fleksibilno kombinovati na različite načine. Modularnost podrazumeva i standardizovane interfejse između modula, što je posebno naglašeno u savremenim softverskim arhitekturama poput mikroservisa.

Iskoristivost (ponovna upotrebljivost) predstavlja stepen u kome se komponente jednog sistema mogu koristiti u drugim sistemima. Postoje različiti nivoi iskoristivosti, uključujući ponovnu upotrebu:

- specifikacija

- dizajna
- koda
- podataka
- testova.

Umesto razvoja novih funkcionalnosti uvek treba razmotriti iskoristivost postojećih komponenti. Glavne prednosti ponovne upotrebe uključuju:

- povećanje produktivnosti,
- smanjenje troškova,
- poboljšanje kvaliteta,
- ubrzanje razvoja i
- smanjenje rizika u procesu razvoja.

Analizabilnost označava lakoću analize i razumevanja softvera, te direktno utiče na prva dva koraka unošenja izmena: (1) razumevanje softvera i (2) pronalaženje delova softvera koje je potrebno izmeniti. Direktno je povezana sa modularnošću (dobra modularnost smanjuje kompleksnost i time poboljšava analizabilnost) i iskoristivošću (korišćenje postojećih komponenti olakšava analizu koda). Anlizabilnost se poboljšava i kroz postajnje dobre dokumentacije i kroz dosledno poštovanje kodnih standarda. Za razumevanje neispravnog ponašanja softvera, postojanje mehanizma za praćenje rada sistema i aktivnosti korisnika može značajno da poboljša analizabilnost.

Izmenljivost predstavlja lakoću implementacije izmena bez uvođenja grešaka. Ključni pokazatelj izmenljivosti je spregnutost, jer visoka spregnutost povlači potrebu za izmenama u različitim delovima koda što dodatno povećava i verovatnoću uvođenja grešaka. Dobra modularnost utiče pozitivno na izmenljivost jer podrazumeva smanjenje kompleksnosti i ograničava uticaj izmena na lokalne delove koda.

Testabilnost označava lakoću provere da li su izmene narušile postojeće funkcionalnosti. Prednosti visoke testabilnosti uključuju bržu isporuku korisnicima i češću povratnu informaciju o ispravnosti softvera programerima. Praktični izazovi uključuju balans između potpunosti testova i vremena njihovog izvršavanja. Za testabilnost je ključna automatizacija procesa testiranja, mogućnost automatskog generisanja testova kao i dostupnost test slučajeva sa unapred poznatim rezultatom izvršavanja.

Primer 1.3.8 (Razmena poruka - nastavak) Razmotrimo dalje primer 1.3.3. Aplikacija za razmenu poruka mora da bude laka za održavanje jer je karakteriše često dodavanje novih funkcionalnosti u skladu sa potrebama i zahtevima korisnika.

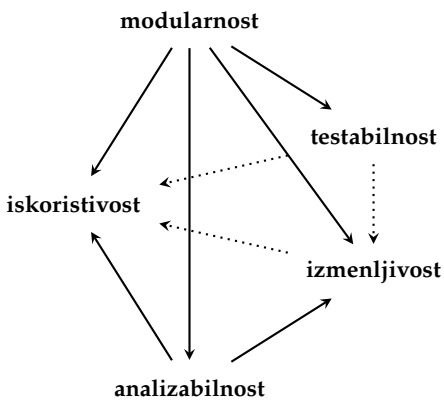
Aplikacija za razmenu poruka treba da ima različite funkcionalnosti, kao što su sistem za slanje poruka, sistem za obaveštenja, korisničke postavke, lično/grupno slanje poruka i sistem za autentifikaciju. Implementacija koju karakteriše lako održavanje je modularna, odnosno podeljena u različite module koje idgovaraju prethodno navedenim funkcionalnostima. Svaki od ovih modula mora da bude nezavisan ali poveziv sa ostatkom sistema putem jasno definisanih interfejsa za komunikaciju. Na primer, sistem za autentifikaciju mora biti odvojen od sistema za poruke i pozive, kako bi se omogućilo lakše upravljanje bez uticaja na ostatak aplikacije.

Poželjno je da se aplikacija može koristiti na različitim platformama, na primer, na mobilnim telefonima (*Android*, *iOS*) i desktop računarima (*Linux*, *Windows*). Da bi se to ostvarilo, ključna je iskoristivost odnosno da postoji mogućnost ponovne upotrebe koda. Sve komponente i funkcionalnosti, koje nisu specifične za platformu, potrebno je dizajnirati tako da ih je moguće koristiti na različitim platformama. Na primer, sistem za prijavu treba da bude identičan na svim platformama, omogućavajući korisnicima da se prijave koristeći isti način verifikacije na svakom uređaju.

Analizabilnost aplikacije omogućava brzo praćenje i analiziranje problema (na primer, ukoliko korisinici prijave da se neke poruke nisu stigle ili da su stigle dva puta). Da bi se to ostvarilo, aplikacija treba da omogući praćenje aktivnosti korisnika i sistema (logovi grešaka, statistika korišćenja, praćenje mrežnih veza).

Usled potrebe za čestim promenama u skladu sa željama korisnika, aplikacija treba da je lako izmenljiva i testabilna. Posebno važna vrsta izmene je nadgradnja. Aplikacija treba da bude dizajnirana tako da omogući da se lako i brzo mogu dodati nove funkcionalnosti, bez uticaja na postojeće. Na primer, dodavanje video poziva mora biti odvojeno od sistema za razmenu poruka, ali se mora integrisati u postojeći korisnički interfejs. Testabilnost u kontekstu

čestih promena i nadgradnja ključna je za osiguravanje da promene i novouvedene funkcionalnosti ne narušavaju postojeće funkcionalnosti. Za testabilnost je posebno značajna automatizacija procesa testiranja, na primer kroz pisanje jediničnih testova i upotrebu alata za pokretanje i automatsku proveru rezultata rada testova. Testiranjem je potrebno pokriti različite scenarije (na primer, sve vrste poruka: tekstualne, glasovne i multimedijalne) uključujući i scenarije koji obuhvataju pogrešno korišćenje aplikacije ili nestandardne slučajeve upotrebe (na primer, korišćenje aplikacije u slučaju problema sa mrežom).



Slika 1.4: Veze između atributa koji utiču na održavanje softvera: strelice sa punim linijama predstavljaju jak uticaj jednog atributa na drugi, dok isprekidane strelice odgovaraju indirektnom uticaju.

Atributi kvaliteta softvera nisu nezavisni, već su međusobno povezani i isprepleteni (slika 1.4). Na primer, razmotrite atribute održavanja. Modularnost, koja se obično postavlja kao jedan od glavnih ciljeva faze projektovanja softvera, direktno utiče na sva ostala četiri atributa, jer je dobra modularnost preduslov iskoristivost, analizabilnost, izmenljivost i testabilnost. Slično tome, analizabilnost utiče na iskoristivost i modifikaciju, dok testabilnost indirektno utiče na iskoristivost i izmenljivost.

1.3.9 Prenosivost

Prenosivost (eng. *portability*) odgovara stepenu u kome se softver može koristiti u različitim okruženjima. Prenosivost obuhvata naredne podatribute:

prilagodljivost — mogućnost korišćenja sa različitim hardverom, softverom ili okruženjem,

instalabilnost — mogućnost instaliranja/deinstaliranja softvera u različitim okruženjima i

zamenljivost — mogućnost zamene drugim softverskim proizvodom za istu svrhu.

Primer 1.3.9 (Kupovina karata — nastavak) Razmotrimo dalje primer 1.3.2.

Aplikacija za kupovinu avio karata treba da bude prilagodljiva različitim uređajima (mobilnim uređajima, tabletima, desktop računarima) i tržištima. Na primer, aplikacija treba da može da se prilagodi različitim veličinama ekrana, od malih ekrana na mobilnim telefonima do velikih ekrana na desktop računarima, pri čemu je potrebno da se automatski prilagođava vizuelni prikaz u skladu sa veličinom ekrana. Takođe, aplikacija treba da ponudi lokalizovane opcije za različite jezike i valute.

Aplikacija treba da bude jednostavna za instalaciju na različitim uređajima. Na mobilnim uređajima, korisnici treba da mogu da preuzmu aplikaciju sa *App Store*-a ili *Google Play*-a uz nekoliko klikova. Za desktop verziju, aplikacija treba da bude dostupna za platforme *Windows*, *Linux* i *macOS* u vidu jednostavnih instalacija koje se pokreću sa par klikova. Za veb verzije, aplikacija mora da omogući jednostavno korišćenje u pretraživaču bez potrebe za preuzimanjem ili instalacijom.

Prilikom ažuriranja aplikacije, potrebno je omogućiti laku zamenljivost starije verzije sa novom verzijom. Zamenljivost znači da aplikacija omogućava da nova verzija može zameniti staru verziju bez gubitka podataka, kao što su starija putovanja, preferencije korisnika ili istorija pretrage. Takođe, zamenljivost podrazumeva lako uklanjanje prethodne verzije aplikacije prilikom instalacije nove. U slučaju da korisnik želi da pređe na konkurentsku aplikaciju, zamenljivost podrazumeva da će podaci kao što su rezervacije ili istorija pretrage biti prenosivi ili lako eksportovani u druge formate.

Rezime

- Cilj upravljanja kvalitetom je da se ispune zahtevi korisnika, optimizuju poslovni procesi i smanji broj grešaka ili defekata.

- Osnovni procesi koji su vezani za kvalitet softvera uključuju planiranje, obezbeđivanje, kontrolu i poboljšanje kvaliteta softvera.
- Serija standarda ISO/IEC 25000 sadrži okvir za procenu kvaliteta softvera.
- U zavisnosti od svrhe i ciljeva softvera, svaki atribut kvaliteta softvera može imati različit nivo važnosti.
- Osnovni atributi kvaliteta softvera su funkcionalna podobnost, performantnost, kompatiblnost, upotrebljivost, pouzdanost, sigurnost, bezbednost, održivost i prenosivost.

Ispitna pitanja

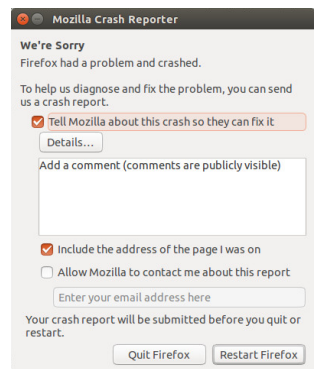
1. Značaj kvaliteta softvera. Procesi i standardi.
2. Značaj kvaliteta softvera. Atributi kvaliteta softvera: funkcionalna podobnost, performantnost, kompatibilnost, pouzdanost. Primeri.
3. Značaj kvaliteta softvera. Atributi kvaliteta softvera: održivost. Primeri.
4. Značaj kvaliteta softvera. Atributi kvaliteta softvera: upotrebljivost, sigurnost, bezbednost, prenosivost. Primeri.

2 Greške u softveru

Pregled

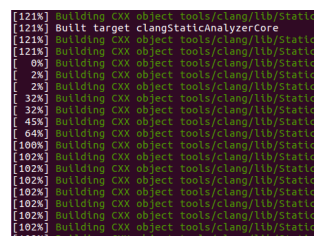
- Na koji sve način neispravan softver utiče na svet oko nas?
- Koliko koštaju softverske greške?

Greške u softveru mogu na različite načine uticati na korisničko iskustvo. Najblaži oblik posledica jeste manja neprijatnost, kao što je iznenadno gašenje internet pregledača (slika 2.1), muzičkog plejera ili mobilne igre, ili greška u sistemu za izgradnju softvera (slika 2.2). Ipak, te neprijatnosti mogu biti i ozbiljnije — na primer, ako navigacioni softver pogrešno usmeri korisnika ka nebezbednoj lokaciji ili ako, u hitnoj situaciji, softverski problem onemogući uspostavljanje poziva.



Slika 2.1: Greška u radu Internet pregledača

Drugi nivo uticaja softverskih grešaka ogleda se u materijalnim gubicima, koji mogu imati ozbiljne posledice po finansijsko poslovanje organizacija. Ove greške se najčešće javljaju u okviru poslovnog softvera i bankarskih informacionih sistema, gde mogu izazvati direktne novčane gubitke, kao i gubitak ili kompromitovanje važnih podataka. U određenim slučajevima, posledice se mogu reflektovati i na reputaciju kompanije, izazivajući dugoročne finansijske probleme.



Slika 2.2: Greška u sistemu za izgradnju softvera

Na kraju, greške u softveru mogu imati i fatalne posledice. One se javljaju u kritičnim sistemima, kao što su softver za upravljanje avionima, automobilima i vozovima, gde čak i najmanji propust može dovesti do teških saobraćajnih nesreća. Sličan rizik postoji i u medicinskim uređajima koji se koriste za dijagnostiku, praćenje vitalnih funkcija i pružanje terapije, gde softverska greška može direktno ugroziti život pacijenta. Greške u softveru svemirskih letelica mogu dovesti do neuspeha misija vrednih više stotna miliona dolara, dok softverski problemi u nuklearnim elektranama nose potencijal za katastrofe nesagledivih razmera. Zbog svega ovoga, razvoj softvera u ovim oblastima zahteva najviši stepen pažnje, rigorozno testiranje i stalnu proveru kvaliteta.

2.1 Primeri poznatih grešaka

Greška $\xrightarrow{\text{proizvodi}}$
Defekt $\xrightarrow{\text{uzrokuje}}$
Pad $\xrightarrow{\text{pravi}}$
Incident $\xrightarrow{\text{posledice}}$ ...

Greške koje naprave programeri, ukoliko se ne otkriju u procesu ispitivanja ispravnosti softvera, čine da softver nije u potpunosti ispravan, odnosno da sadrži defekte. Ti defekti u određenim situacijama uzrokuju pad sistema, odnosno softver se ponaša drugačije od očekivanog. Pad pravi incident koji sa sobom može da nosi različite posledice.

2.1.1 Neprijatnosti i materijalni gubici

Neke od najpoznatijih softverskh grešaka, osim što su izazvale ozbiljne neprijatnosti korisnicima, ostavile su i duboke finansijske posledice po kompanije koje su ih razvile. Njihov uticaj nije bio ograničen samo na korisničko iskustvo, već je često vodio ka gubicima poverenja, padu akcija i tržišne vrednosti, kao i značajnim materijalnim štetama.

Aplikacija *Apple Maps*, 2012

Kompanija *Apple* je 2012. godine odlučila da ukloni aplikaciju za mape *Google Maps* na svom operativnom sistemu *iOS* i zameni je svojom aplikacijom — *Apple Maps*. Nova aplikacija je predstavljena uz operativni sistem *iOS 6*, a trebala je da bude modernija, sa 3D prikazima, navigacijom koja prati tekuće stanje u saobraćaju i glasovnim navođenjem.

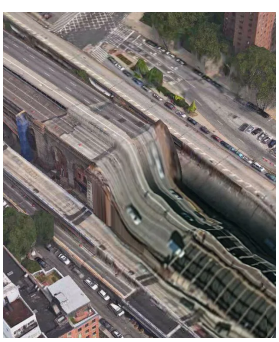
Međutim, aplikacija *Apple Maps* je odmah po lansiranju doživela veliku kritiku zbog brojnih grešaka, uključujući:

Pogrešne lokacije — Gradovi, znamenitosti i firme su bili loše označeni ili potpuno pogrešno locirani. Na primer, policija u Australiji je upozorila građane da ne koriste *Apple Maps* jer je vodič za grad Mildura ljude odvodio usred opasne pustinje.

Netačne rute — Navigacija je korisnike vodila pogrešnim ili nepostojećim putevima.

Izobličene mape — 3D prikazi zgrada i terena su izgledali iskrivljeno ili neprirodno. Primer lošeg prikaza dat je na slici 2.3.

Nedostajuće informacije — Mnoge lokacije i ulice nisu bile uopšte ucrtane.



Slika 2.3: Primer prikaza aplikacije *Apple Maps*

Izvršni direktor kompanije *Apple*, Tim Kuk (eng. *Tim Cook*), uputio je javno izvinjenje i korisnicima čak preporučio da privremeno koriste konkurentske aplikacije. Zbog skandala, *Apple* je otpustio nekoliko ključnih ljudi koji su bili na visokim pozicijama. Negativna percepcija kompanije dovela je do pada vrednosti akcija *Apple*-a za približno 4,5%, što je bilo smanjenje tržišne vrednosti za oko 30 milijardi dolara. Kako bi ispravio početne defekte i poboljšao kvalitet svoje aplikacije, *Apple* je tokom narednih godina uložio više milijardi dolara u razvoj *Apple Maps*. Ova ulaganja obuhvatala su akvizicije kompanija specijalizovanih za pravljenje mapa, prikupljanje sopstvenih podataka, kao i razvoj novih funkcionalnosti. Iako je teško precizno kvantifikovati ukupne finansijske gubitke, kombinacija pada tržišne vrednosti i višegodišnjih ulaganja u unapređenje *Apple Maps* sugeriše da je ova greška imala značajan ekonomski uticaj na kompaniju.

Knight Capital Group, 1. avgust 2012. godine

U avgustu 2012. godine, kompanija *Knight Capital Group* suočila se sa jednim od najozbiljnijih tehnoloških incidenata u istoriji savremenog finansijskog tržišta. Softverska greška u sistemu za trgovanje rezultirala je gubitkom od $440 miliona za svega 45 minuta — što predstavlja iznos gotovo četiri puta veći od neto profita kompanije ostvarenog tokom prethodne godine. Dodatno, vrednost akcija kompanije je u roku od dva dana opala za čak 75%.

Do incidenta je došlo prilikom implementacije novog softverskog modula, kada je nenamerno reaktiviran stari, nefunkcionalni deo koda. Aktivirani kôd je automatski generisao veliki broj netačnih naloga, izazivajući kupovinu i prodaju akcija po izrazito nepovoljnim cenama. Ovaj događaj ne samo da je značajno ugrozio reputaciju kompanije, već je ubrzo doveo i do njenog potpunog finansijskog kolapsa.

Knight Capital Group

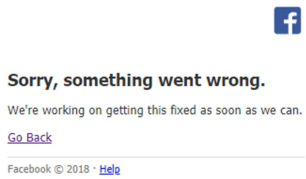
- **Greška**:
  (1) Programer nije ažurirao sve servere sa najnovijom verzijom softvera
  (2) Nepostojanje provere verzije koda prilikom obrade upita
- **Defekt**: Moguće je pokrenuti kôd koji se koristio za testiranje
- **Pad**: Pokrenute su pogrešne transakcije
- **Incident**: Veliki finansijski gubitak

Facebook Outage, 13. mart 2019. godine

U martu 2019. godine, kompanija Facebook doživela je jedan od najznačajnijih prekida u svojoj istoriji koji je uticao na milijarde korisnika širom sveta. Ovaj događaj, poznat kao *Facebook Outage 2019*, izazvao je ozbiljne poremećaje u funkcionisanju ne samo servisa *Facebook*, već i njegovih povezanih servisa kao što su *Instagram*, *WhatsApp* i *Messenger*.

Zanimljivost: Zvanično saopštenje kompanije *Facebook*:

> "Juče smo izvršili promenu konfiguracije servera koja je pokrenula niz problema. Kao rezultat toga, mnogi korisnici su imali poteškoća u pristupu našim aplikacijama i uslugama."



Slika 2.4: Poruka o nedostupnosti servisa

Slika 2.5: Deo zvaničnog izveštaja kompanije *Google* https://status.cloud.google.com/incident/zall/20013

Prema zvaničnom saopštenju kompanije, uzrok prekida bila je „promena konfiguracije servera" koja je pokrenula lanac problema u sistemu. Nakon što su servisi ponovo postali dostupni, *Facebook* se izvinio korisnicima i naveo da su sistemi u fazi oporavka. Međutim, kompanija nije pružila detaljnije informacije o tačnom uzroku problema, što je izazvalo kritike stručne javnosti i korisnika zbog nedostatka transparentnosti. Iako su detalji ostali oskudni, izveštaji sugerišu da je došlo do greške u internim sistemima za rutiranje podataka, što je dovelo do prekida komunikacije između *Facebook*-ovih centara podataka. Zbog toga su korisnici širom sveta imali ograničen ili nikakav pristup platformama.

Ovaj prekid je bio jedan od najopsežnijih u istoriji društvenih mreža. Pored korisničkih neugodnosti (slika 2.4), prekid je imao i finansijske posledice. Analitičari su procenili da je *Facebook*, sa prosečnim dnevnim prihodima od oko 189 miliona dolara, pretrpeo značajan gubitak prihoda tokom ovog perioda. Neki izvori navode da je prekid trajao 14 sati, dok drugi navode da je prekid trajao oko 24 sata. U skladu sa time se i razlikuju procene finansijskog gubitka. Takođe, ovaj incident je uticao i na pad akcija kompanije *Facebook* za oko 2%.

Google Cloud Outage, 14. decembar 2020. godine

Kompanija *Google* je imala neželjeni prekida usluga 14. decembra 2020. godine u trajanju od 47 minuta. Tokom ovog perioda, korisnici širom sveta nisu mogli da pristupe uslugama koje zahtevaju *Google* nalog, uključujući *Gmail*, *YouTube*, *Google Drive*, *Google Docs* i druge. Ovaj događaj je imao globalni uticaj na korisnike i kompanije koje se oslanjaju na *Google*-ovu infrastrukturu.

Prema zvaničnom izveštaju kompanije *Google* (slika 2.5), uzrok prekida bio je problem u sistemu za upravljanje kvotama skladišnog prostora, koji je nenamerno smanjio kapacitet centralnog sistema za upravljanje identitetima korisnika. Ova greška je dovela do toga da se zahtevi za autentifikaciju nisu mogli obraditi, što je rezultiralo greškama prilikom pokušaja pristupa uslugama koje zahtevaju prijavu.

Iako je prekid trajao manje od sat vremena, njegov uticaj je bio značajan zbog široke upotrebe *Google*-ovih servisa u svakodnevnom poslovanju i komunikaciji. *Google* se izvinio

korisnicima i naveo da su preduzete mere kako bi se sprečilo ponavljanje sličnih incidenata u budućnosti.

Teško je precizno kvantifikovati direktan finansijski gubitak izazvan ovim konkretnim prekidom. Poznato je da je *Google Cloud* u 2020. godini zabeležio operativni gubitak od 5,61 milijardu dolara. Ovaj gubitak pripisuje se raznim poslovnim odlukama i nije vezan samo za pomenuti incident.

2.1.2 Fatalne posledice

Većina softverskih grešaka ne rezultira fatalnim posledicama. Ipak, kada se propusti jave u okviru kritičnih sistema, njihovi efekti mogu biti izuzetno ozbiljni — uključujući gubitke ljudskih života, teške povrede, značajne finansijske štete i razorne sistemske havarije. Iako su ovakvi incidenti retki u apsolutnim brojkama, njihov uticaj na bezbednost, ekonomiju i poverenje u tehnologiju je nesrazmerno velik.

Uprkos kontinuiranom unapređenju programerskih praksi, uključujući primenu formalnih metoda, automatizovanog testiranja i usklađivanje sa bezbednosnim standardima, učestalost najtežih grešaka nije opala u meri u kojoj bi se očekivalo. Promenila se, međutim, njihova priroda: umesto pojedinačnih grešaka u kodu, sve češće se javljaju kompleksni, sistemski defekti koji proizilaze iz loše integracije, nejasnih zahteva ili neočekivane interakcije među podsistemima.

Therac 25, 1985–1987

Aparat za radioterapiju *Therac-25* je bio porizvod kanadske kompanije AECL (*Atomic Energy of Canada Limited*). Ovaj aparat je bio napredni naslednik serije uređaja *Therac-6* i *Therac-20*, koji su takođe koristili sličnu tehnologiju za primenu radioterapije. *Therac-25* je kombinovao softversku kontrolu i hardverske komponente u cilju povećanja preciznosti zračenja.

Između 1985. i 1987. godine, apart je uzrokovao najmanje šest slučajeva predoziranja pacijenata radijacijom, od kojih su troje umrli. Ovi nesrećni slučajevi bili su rezultat softverske greške. *Therac-25* je imao dva režima zračenja:

- Elektronski snop niskog intenziteta,

- Fotonski snop visoke energije, koji mora da se dodatno filtrira.

Ako bi tehničar najpre uneo prvi režim, a zatim ga promenio u drugi, softver nije uvek uspevao da ispravno ažurira stanje uređaja. Rezultat toga je bio da uređaj misli da koristi snop niskog intenziteta i doze koje su za to predviđene, ali bi zapravo poslao visokoenergetski snop bez zaštitnog filtera. Greška je nastajala usled loše sinhronizacije:

Therac 25

- **Greška**:
  (1) Programer je napravio grešku u promeni podataka bez adekvatne sinhronizacije,
  (2) Tehničar je promenio režim rada u *pogrešnom* trenutku
- **Defekt**: Mogućnost slanja visokoenergetskog snopa bez zaštitnog filtera
- **Pad**: Prekoračena je bezbedna doza zračenja
- **Incident**: Pacijent je umro

- Jedan proces je ažurirao režim tretmana.
- Drugi proces je proveravao spremnost uređaja za zračenje.

Ako bi se ti procesi „preklopili“ u pogrešnom trenutku, uređaj bi pogrešno zaključio da je sve spremno i poslao snop visoke energije bez zaštitnog filtera sa dozom zračenja koja je predviđena za snop niskog intenziteta.

Pored ove softverske greške, nesrećama su doprineli i naredni propusti:

- nedostatak hardverskih sigurnosnih sistema — *Therac-25* je uklonio fizičke sigurnosne blokade koje su postojale u prethodnim uređajima, oslanjajući se isključivo na softver,
- loša dijagnostika — interfejs aparata nije jasno ukazivao na probleme, tehničari su dobijali nejasne poruke o grešci) i
- zanemarivanje prethodnih incidenata — kompanija *AECL* nije preduzela adekvatne mere nakon inicijalnih pritužbi i događaja.

Kao rezultat ovih nesrećnih događaja, razvijeni su novi standardi za sigurnost medicinskih uređaja, uključujući rigoroznije zahteve za testiranje i formalnu verifikaciju softvera u medicinskim sistemima.

Dahran raketa, 25. februar 1991. godine

Tokom Zalivskog rata, 25. februara 1991. godine, iračka balistička raketa pogodila je američku vojnu bazu u Dahranu (Saudijska Arabija), usmrtivši 28 vojnika i ranivši više od 100 vojnika. Nakon detaljne istrage, utvrđeno je da je uzrok neuspeha protivraketnog sistema *Patriot* bila greška u softveru koji je upravljao internim sistemskim vremenom. Naime,

zbog akumulacije zaokruživanja decimalnih vrednosti u brojevima s pokretnim zarezom, došlo je do odstupanja od približno jedne trećine sekunde nakon 100 sati neprekidnog rada sistema. Iako se ovo odstupanje može činiti beznačajnim, u kontekstu ogromnih brzina kojima se balističke rakete kreću, ono je rezultovalo prostornim promašajem od oko 600 metara. Radar je usled toga tražio raketu na pogrešnoj lokaciji, nije uspeo da je prati, i samim tim nije aktivirao mehanizam za presretanje.

Ovaj događaj predstavlja jedan od najupečatljivijih primera fatalnih posledica softverske greške u savremenoj vojnoj istoriji. Osim ljudskih žrtava, incident je izazvao ozbiljnu zabrinutost u vezi sa pouzdanošću softverski vođenih odbrambenih sistema. Pentagon i američka vojska bili su izloženi javnoj i političkoj kritici, a celokupan događaj podstakao je temeljnu reviziju načina na koji se razvija, testira i primenjuje softver u vojnim i drugim bezbednosno osetljivim sistemima.

Dahran raketa

- **Greška**: Neprecizno zaokruživanje vrednosti brojeva
- **Defekt**: Pogrešno računanje vremena
- **Pad**: Neaktiviran mehanizam za presretanje rakete
- **Incident**: Smrt 28 vojnika i povrede više od 100 vojnika

Slika 2.6: Maketa rakete *Ariane 5*, Muzej vazduhoplovstva u Parizu, Francuska

Ariane 5, 4. jun 1996. godine

Raketa *Ariane 5* Evropske svemirske agencije (ESA) uništila se svega 37 sekundi nakon poletanja, 4. juna 1996. godine. Pri tome su izgubljena četiri satelita namenjena proučavanju Zemljine magnetosfere. Procenjena šteta iznosila je preko 370 miliona dolara, čime je ovaj incident postao jedan od najskupljih softverskih grešaka u istoriji svemirskih misija.

Uzrok nesreće bio je softverski propust. Sistem je pokušao da konvertuje 64-bitnu vrednost brzine u 16-bitnu vrednost, što je izazvalo numeričko prekoračenje i izuzetak koji nije bio adekvatno obrađen. To je dovelo do pogrešnih komandi, skretanja sa putanje i aktivacije sistema za samouništenje.

Ova katastrofa je istakla kritičnu važnost verifikacije softverskih komponenti u kompleksnim i bezbednosno osetljivim sistemima, te je podstakla reforme u ESA-inim procedurama testiranja i verifikacije softvera.

Ariane 5

- **Greška**:
  (1) Konverzija iz 64-bitne u 16-bitnu vrednost
  (2) Nepostojanje adekvatne obrade izuzetka
- **Defekt**: Sistem neadekvatno reaguje na greške u ulaznim podacima
- **Pad**: Skretanje sa putanje i aktivacija sistema za samouništenje
- **Incident**: Veliki finansijski gubitak i propuštena prilika za novim naučnim saznanjima

The Mars Climate Orbiter 23. septembar 1999. godine

Letilica *Mars Climate Orbiter (NASA)* nestala je prilikom ulaska u orbitu Marsa 23. septembra 1999. godine, nakon što je ušla u orbitu značajno niže od planirane putanje i najverovatnije izgorela u atmosferi. Istragom je utvrđeno da je

Slika 2.7: Fotografija letilice *Mars Climate Orbiter* prilikom sklapanja, NASA, januar 1998. godine

Mars Climate Orbiter

- **Greška**: Deo softvera je koristio imperijalne jedinice dok je drugi deo softvera očekivao metričke jedinice
- **Defekt**: Pogrešno računanje putanje
- **Pad**: Ulazak u orbitu Marsa na pogrešnom rastojanju
- **Incident**: Letilica je izgorela u atmosferi Marsa

uzrok nesreće bio propust u konverziji jedinica: deo softvera je koristio imperijalne jedinice dok je drugi deo softvera očekivao metričke jedinice. Ova neusaglašenost dovela je do pogrešnog proračuna putanje.

Sama letilica je koštala 125 miliona dolara, dok je ukupni trošak misije procenjen na oko 327 miliona dolara. Incident je izazvao ozbiljnu kritiku NASA-inih programerskih i upravljačkih procedura. Kao rezultat, uvedene su strože kontrole, standardizacija jedinica i unapređeni protokoli testiranja softverskih komponenti.

Greške u automobilskoj industriji

Softverske greške u automobilskoj industriji su ozbiljan problem koji može imati direktne posledice po bezbednost putnika. Na primer, kompanija *General Motors* je softversku grešku otkrila tek nakon saobraćajnih nesreća u kojima je bio jedan smrtni ishod i tri povrede. Ova greška je sprečavala aktivaciju prednjih vazdušnih jastuka i zatezača sigurnosnih pojaseva tokom sudara, povećavajući rizik od povreda putnika. Kompanija je u septembru 2016. godine, najavila opoziv 4,3 miliona vozila.

General Motors je naveo da ovaj opoziv neće imati značajan uticaj na finansijske rezultate kompanije i precizna procena troškova nije javno objavljena. Međutim, u slučaju koji je uključivao opoziv sličnog broja vozila (opoziv zbog problema sa *Takata* vazdušnim jastucima) troškovi su bili procenjeni na 550 miliona dolara. S obzirom na to, može se pretpostaviti da su troškovi ovog opoziva bili značajni, ali verovatno niži od pomenute sume, jer je rešenje problema uključivalo ažuriranje softvera, što je manje skupo od fizičke zamene delova.

Sličan problem je imala i kompanija *Fiat Chrysler* koja je u maju 2017. odlučila da opozove više od 1,25 miliona kamioneta širom sveta, s ciljem otklanjanja softverske greške koja je potencijalno povezana sa jednom saobraćajnom nesrećom sa smrtnim ishodom i dve povrede. Iako nisu postojali konačni dokazi da je softverski propust direktno uzrokovao ove nesreće, proizvođač vozila je bio odlučio da opoziv sprovodi preventivno, kao meru predostrožnosti.

Kako je navedeno u zvaničnom saopštenju, softverska anomalija je privremeno onemogućavala aktivaciju bočnih vazdušnih jastuka, kao i mehanizama koji automatski zatežu

pojas. Do takvog neispravnog ponašanja sistema je moglo da dođe u slučaju prevrtanja vozila izazvanog snažnim udarcem u donji deo automobila, na primer pri udaru u prepreke na putu ili prilikom vožnje van uređenih saobraćajnica. Verovatnoća nastanka takvog incidenta ocenjena je kao izuzetno mala, jer je za njegovu pojavu potrebna vrlo specifična i retka sekvenca događaja. U cilju otklanjanja uočene greške, bilo je neophodno izvršiti reprogramiranje računarskih modula u svim vozilima koja su obuhvaćena opozivom. Tačni troškovi ovog opoziva nisu objavljeni.

2.2 Troškovi usled grešaka u softveru

Najveći troškovi usled grešaka u softveru nastaju kada se greške otkriju nakon puštanja sistema u rad. To uključuje gubitke u prihodu, narušavanje reputacije, opozive proizvoda, regulatorne kazne, kao i gubitke podataka ili živote.

Američki nacionalni institut za standarde i tehnologiju (eng. *US National Institute of Standards and Technology, skraćeno NIST*) je 2002. godine napravio opsežnu studiju sa ciljem procene uticaja softverskih grešaka na američku ekonomiju. Po ovoj studiji, neispravan softver je koštao američku ekonomiju 59.5 milijardi dolara godišnje. Dodatno, u studiji je naglašeno da bi rano otkrivanje grešaka moglo da uštedi 22 milijarde dolara godišnje, dakle nešto više od trećine ukupnih troškova.

Ova studija značajno je uticala na ulaganje u unapređivanje procesa razvoja softvera sa ciljem poboljšanja kvaliteta softvera i smanjenja količine grešaka u softveru. Dodatno, posebna ulaganja su bila usmerena ka razvoju alata koji daju podršku i omogućavaju automatizaciju procesa verifikacije softvera.

Dvadeset godina kasnije*, Konzorcijum za kvalitet informacija i softvera (eng. *Consortium for Information and Software Quality*), skraćeno CISQ) u izveštaju o ceni lošeg kvaliteta softvera (eng. *Cost of Poor Software Quality*) u Sjedinjenim Američkim Državama za 2022. godinu navodi da su troškovi usled lošeg kvaliteta softvera porasli na 2410 milijardi američkih dolara. U izveštaju za 2020. godinu, navedna je procena troškova 2080 milijardi američkih dolara. Dakle, za kratko vreme uočen je veliki porast troškova uzrokovanih softverskim greškama.

Slika 2.8: Tehnički izveštaji konzorcijuma *CISQ*, uključujući i izveštaje *Cost of Poor Software Quality* za 2020. i 2022. godinu

* Iako najavljen, izveštaj za 2024. godinu još uvek nije dostupan. Poslednji javno dostupan izveštaj je iz 2022. godine.

CISQ u izveštaju za 2020. godin navodi da su najbitniji troškovi[†] proistekli iz softverskih grešaka:

Operativni softverski kvarovi: Procena je da su operativni softverski kvarovi (eng. *operational software failures*), uključujući greške u funkcionisanju sistema i nepredviđene zastoje, doveli do gubitaka od oko 1560 milijardi dolara. Ovaj broj predstavlja povećanje od 22% u odnosu na 2018. godinu.

Neuspešni razvojni projekti: Neuspešni ili otkazani softverski projekti, često zbog lošeg upravljanja ili nedostatka kvaliteta, doveli su do gubitaka od 260 milijardi dolara.

Problemi sa zastarelim sistemima: Održavanje i nadogradnja zastarelih sistema koštali su oko 520 milijardi dolara.

Tehnički dug: Troškovi tehničkog duga[‡] (eng. *technical debt*), koji predstavlja nepopravljene greške i neefikasnosti u kodu, procenjeni su na 1310 milijardi dolara.

Gubici zbog sajberkriminala: U izveštaju nije navedena tačna procena troškova usled sjaberkriminala, osim da su ovi troškovi značajni i u porastu.

CISQ u izveštaju za 2022. godinu navodi da su najviše porasli gubici

- prouzrokovani sajber kriminalom koji su rezultat postojećih softverskih ranjivosti. Oni su između 2020. i 2021. godine porasli za 64%, a rastući trend se nastavio i u 2022. godini. Prosečan trošak jednog incidenta narušavanja podataka u SAD-u dostigao je 9,44 miliona dolara.
- nastali usled problema u komponentama softvera otvorenog koda. Broj neuspeha zbog slabosti u komponentama otvorenog koda povećao se za alarmantnih 650% između 2020. i 2021. godine.
- koji su rezultat uticaja tehničkog duga. Ovaj trošak je porastao na približno 1520 milijardi dolara godišnje, što

[†] Ovi troškovi nisu međusobno nezavisni tj. ima preklapanja, pa zbog toga njihov zbir nije 2080 već veći. Na primer, problemi sa zastarelim sistemima i operativni softverski kvarovi su usko povezani sa tehničkim dugom.

[‡] Tehnički dug se odnosi na posledice pravljenja brzih, kratkoročnih rešenja u razvoju softvera, umesto boljih, dugoročno održivih rešenja. To je cena koju tim mora da plati u budućnosti zbog kompromisa u kvalitetu koda napravljenih da bi se nešto brže isporučilo danas. Tehnički dug nastaje usled nejasnih i nepreciznih zahteva, zaobilaženja dobrih programerskih praksi da bi se „stigao rok“, odsustva testova i dokumentacije, održavanja starog koda bez značajnih refaktorisanja i nedostatka automatizacije i sigurnosnih provera. Posledice tehničkog duga su teži i sporiji razvoj novih funkcionalnosti, više grešaka i kvarova, kao i povećani troškovi održavanja.

ukazuje na ozbiljnost problema i potrebu za njegovim rešavanjem.

Svi ovi izveštaji su vrlo konzervativni jer uzimaju u obzir samo dostupne i javno prijavljene podatke. U većini slučajeva, oni se oslanjaju na analize izveštaja o incidentima, javno objavljenim gubicima, regulatornim dokumentima i istraživanjima sprovedenim među organizacijama. Međutim, stvarna slika je često znatno ozbiljnija.

Kompanije, iz različitih razloga, često smanjuju i ublažavaju izveštaje o problemima koji su nastali usled grešaka u softveru. To rade kako bi zaštitile svoj ugled, izbegle pad poverenja korisnika i investitora, ili kako bi izbegle regulatorne i pravne posledice. Greške u softveru koje dovedu do curenja podataka, pada sistema, zastoja u radu ili finansijskih gubitaka neretko se interno rešavaju i nikada ne dospeju u javnost.

Pored toga, mnoge organizacije nemaju razvijen sistem za praćenje i precizno merenje posledica softverskih grešaka, pa čak i kada žele da prijave problem – nemaju celovite podatke. Zbog toga je procena ukupnih troškova lošeg kvaliteta softvera u izveštajima poput onih koje objavljuje CISQ gotovo sigurno donji prag stvarne štete.

Stvarna cena lošeg softvera uključuje i skrivene gubitke kao što su izgubljene prilike, pad produktivnosti i frustracije korisnika. Sve to ostaje van zvanične statistike, ali ima dugoročne posledice po poslovanje i razvoj digitalnog tržišta.

Rezime

- Uticaj neispravnog softvera: neprijatnosti, materijalni gubici i materijalno nemerljive posledice.
- Postoji veliki broj primera softverskih grešaka koje su imale značajne materijalne i nematerijalne posledice.
- Materijalni troškovi neispravnog softvera se mere hiljadama milijardi dolara na godišnjem nivou.
- Troškovi neispravnog softvera u velikoj meri su posledica tehničkog duga, odnosno pravljenja softvera bez adekvatnog kvaliteta.

Ispitna pitanja

5. Uticaj neispravnog softvera. Primeri neprijatnosti i materijalnih gubitaka.
6. Uticaj neispravnog softvera. Primeri fatalnih posledica.
7. Troškovi usled grešaka u softveru.

Verifikacija i validacija softvera 3

Pregled

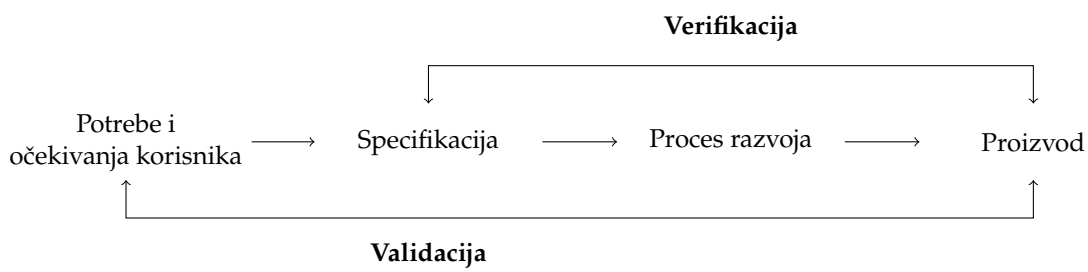
- Koji je odnos verifikacije i validacije softvera?
- Kako se definišu verifikacija i validacija softvera?
- Koji pristupi i problemi koji se javljaju u okviru verifikacije softvera?
- Koje su specifičnosti automatske statičke verifikacije softvera?

Obezbeđivanje i kontrola kvaliteta softvera obuhvataju procese validacije i verifikacije.

Verifikacija softvera obuhvata sve procese koji su potrebni da se osigura da razvijeni softver zadovoljava zadatu specifikaciju, odnosno da ne sadrži defekte i da bude ispravan. Verifikacijom se odgovara na pitanje „Da li je softver ispravno izgrađen?" i fokusira se na proveru da li sistem u potpunosti ispunjava specifikaciju.

Validacija softvera obuhvata sve procese koji su potrebni da se osigura da razvijeni softver zadovoljava korisničke potrebe. Validacijom se odgovara na pitanje „Da li je izgrađen pravi softver?" i procenjuje da li softver ispunjava potrebe i očekivanja krajnjih korisnika.

U oba slučaja, kontrola se najčešće sprovodi testiranjem, ali postoje i druge napredne tehnike koje se mogu koristiti. Kako bi softverski proizvod bio upotrebljiv i uspešan, neophodno je da zadovolji i kriterijume ispravnosti (verifikacija) i usklađenosti sa korisničkim zahtevima (validacija).



Slika 3.1: Verifikacija i validacija softvera

Funkcionalna podobnost, performantnost, kompatibilnost, pouzdanost, sigurnost, bezbednost i prenosivost predstavljaju dinamička svojstva softvera i procenjuju se primarno kroz procese verifikacije. Upotrebljivost se uglavnom procenjuje kroz validaciju, iako se pojedini aspekti, poput otpornosti na korisničke greške, mogu ispitivati i verifikacijom. Održivost je statička karakteristika koja se ocenjuje kroz verifikaciju, najčešće pregledima koda i analizom pomoću alata za statičku analizu.

3.1 Odnos verifikacije i validacije softvera

Napomena: studentski i industrijski projekti. Proces verifikacije i validacije studentskih projekata najčešće se sprovodi pri kraju procesa razvoja softvera, koristeći mali broj test primera. Ovaj proces obično čini veoma mali procenat ukupnog napora koji studenti ulože u razvoj softvera. S druge strane, proces verifikacije i validacije u industriji sprovodi se tokom celokupnog razvoja softvera i najčešće čini više od 30% ukupnog napora, pri čemu taj procenat može značajno da varira u zavisnosti od vrste projekta i nivoa kvaliteta koji je neophodno postići. Za bezbednosno kritične sisteme ovaj procenat je značajno viši.
Osnovne razlike između studentskih i industrijskih projekata su:

- dužina upotrebe softvera,
- način upotrebe softvera,
- broj korisnika,
- očekivanja i
- cena pada.

U savremenom digitalnom okruženju, korisnici pokazuju sve manju toleranciju prema softverskim greškama, naročito kada je reč o mobilnim aplikacijama. Ogromna konkurencija i visoka očekivanja doveli su do toga da i najmanji propust može biti presudan za uspeh proizvoda. Korisnici danas imaju širok izbor – za svaku potrebu postoji desetine alternativa, bilo da se radi o aplikacijama za zdravlje, zabavu, hobije ili društvene mreže. Zbog toga, loše korisničko iskustvo se ne oprašta; prelazak na konkurentski proizvod je lak i brz. Greške u softveru nisu samo tehnički izazov — one predstavljaju direktan poslovni rizik.

Upravo zato, procesi verifikacije i validacije softvera postaju ključni u razvoju aplikacija. Verifikacija se odnosi na sve vrste provera ispravnsoti rada aplikacije, uključujući i testiranje aplikacije u realnim uslovima i na različitim uređajima, kako bi se osiguralo da sistem funkcioniše bez grešaka. Ako aplikacija ima bagove, ruši se ili je spora, korisnici je percipiraju kao nepouzdanu i deinstaliraće je često i nakon samo prve greške. Dodatno, negativne recenzije odvraćaju buduće korisnike.

S druge strane, validacija procenjuje da li aplikacija zaista ispunjava očekivanja korisnika. Aplikacija može biti tehnički ispravna, ali ako ne nudi jasnu vrednost ili ako je interfejs zbunjujući, korisnik je verovatno više neće koristiti. Upravo zbog toga, elementi kao što su dizajn korisničkog iskustva i korisničkog interfejsa, uvodno vođenje korisnika kroz aplikaciju i prvi utisak igraju presudnu ulogu u njenom prihvatanju.

Primer 3.1.1 (Verifikacija i validacija) U komunikaciji sa klijentom, definisana je specifikacija sistema:

> *Aplikacija treba da prikazuje vreme.*

U skladu sa zadatom specifikacijom, moguće je napraviti dva rešenja.

Rešenje 1: Digitalni časovnik

Rešenje 2: Aplikacija za vremensku prognozu

Rešenja su fundamentalno različita jer je reč „vreme" višeznačna.

Verifikacijom se proverava da li je sistem koji je izgrađen u skladu sa zadatom specifikacijom, tj. da li sistem tačno prikazuje vreme.

- U slučaju digitalnog časovnika, verifikacija obuhvata proveru da li aplikacija ispravno prikazuje trenutno lokalno vreme, u skladu sa sistemskim vremenom uređaja ili referentnim vremenskim serverom. To podrazumeva ispitivanje usklađenosti prikazanog vremena sa vremenom operativnog sistema, kao i ispravnost prikaza prilikom promene vremenske zone. Takođe, proverava se da li se vreme automatski osvežava bez potrebe za dodatnom intervencijom korisnika, odnosno bez manuelnog osvežavanja stranice ili aplikacije.
- Kod aplikacija za prikaz vremenske prognoze, verifikacija se fokusira na tačnost i konzistentnost prikaza prognoziranih meteoroloških podataka koji se preuzimaju sa odgovarajućih vremenskih servisa, kao što su *AccuWeather* ili *OpenWeatherMap*. Neophodno je utvrditi da li se podaci, poput temperature i opisa vremenskih uslova, prikazuju u odgovarajućem i očekivanom formatu. Pored toga, proverava se i da li aplikacija ispravno prikazuje vremenske podatke za različite lokacije, bilo da su one definisane nazivima gradova ili geografskim koordinatama.

Validacija predstavlja postupak kojim se utvrđuje da li je razvijeni sistem u skladu sa stvarnim potrebama i očekivanjima korisnika. Na primer, ukoliko je korisnik zahtevao digitalni časovnik, tada je rešenje koje implementira upravo tu funkcionalnost validno. Međutim, ako je korisnik

očekivao aplikaciju za vremensku prognozu, tada digitalni časovnik ne zadovoljava njegove potrebe i smatra se potpuno neodgovarajućim rešenjem.

Jasno je da i potpuno ispravan digitalni časovnik nema vrednost u kontekstu zahteva za aplikacijom koja prikazuje vremenske uslove. S druge strane, ukoliko je korisnik zaista želeo digitalni časovnik, ali je implementacija neispravna, sistem takođe postaje neupotrebljiv.

3.2 Tehnike verifikacije softvera

Verifikacija softvera obuhvata skup sistematskih metoda i aktivnosti kojima se proverava da li softverski proizvod ispravno implementira zadatu specifikaciju. Tehnike verifikacije softvera uključuju dinamičku i statičku verifikaciju softvera. Dinamička verifikacija softvera podrazumeva ispitivanje ispravnosti softvera u toku njegovog izvršavanja. Statička verifikacija softvera podrazumeva ispitivanje ispravnosti softvera bez njegovog izvršavanja, odnosno analizu izvornog koda programa.

Napomena: Testiranje se često koristi i kao sinonim za verifikaciju i kao sinonim za validaciju softvera. Testiranje se često koristi i kao sinonim za brigu o kvalitetu softvera. Međutim, testiranje je ipak samo važan deo verifikacije softvera, važan deo validacije softvera, a verifikacija i validacija su važan deo brige o kvalitetu softvera.

Dinamička verifikacija softvera. Najčešći vid dinamičke verifikacije softvera je testiranje. Testiranje je izvršavanje programa sa ciljem da se pronađe što više mogućih defekata ili da se stekne dovoljno poverenja u sistem koji se testira. Pravilnim i sistematičnim testiranjem podižemo nivo pouzdanosti i smanjujemo verovatnoću da greške promaknu. Obim testiranja i aktivnosti koje su vezane za testiranje zavise od metodologije razvoja softvera i prilagođavaju se svakom konkretnom projektu. Moderni pristupi razvoju softvera podrazumevaju da je testiranje prisutno u svakoj fazi razvoja softvera. U okviru dinamičke verifikacije softvera, koriste se i alati za debagovanje kao i razne vrste profajlera.

Statička verifikacija softvera. Postoje različite vrste statičke verifikacije:

- Provere i pregledi koda koje rade ljudi
- Formalne metode verifikacije softvera — uslovi ispravnosti softvera iskazuju se u terminima matematičkih tvrđenja na striktno definisanom formalnom jeziku izabrane matematičke teorije.

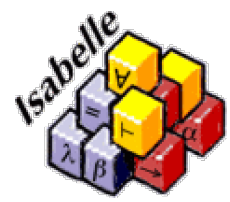
Ručne provere i pregledi koda su veoma važni i svakodnevno se primenjuju u okviru procesa razvoja softvera. Formalne metode se sve više koriste na najrazličitije načine.

Važno mesto u oblasti formalnih metoda zauzima formalna semantika programskih jezika. Formalna semantika programskog jezika koristi matematičke modele i formalne tehnike za precizno definisanje značenja programa. Cilj formalne semantike je da na rigorozan način opiše kako se izvršavaju naredbe u programskom jeziku, čime se eliminiše dvosmislenost i omogućava proverljiva interpretacija. Formalna semantika predstavlja temelj za razumevanje i analiziranje značenja softverskog koda na precizan, matematički način.

Upravo kroz formalnu semantiku postaje moguće:

- modelovati izvršavanje programa nezavisno od konkretne mašine ili kompajlera,
- dokazivati korektnost softverskih komponenti i
- razvijati alate za formalnu verifikaciju.

Slika 3.2: Logo alata *Rocq* (`https://rocq-prover.org/`)



Slika 3.3: Logo alata *Isabelle* (`https://isabelle.in.tum.de/`)

Idealno rešenje za verifikaciju softvera bi bio alat koji automatski analizira kôd i daje precizne informacije o njegovoj ispravnosti. Međutim, postoji fundamentalno ograničenje zbog kojeg tako nešto nije moguće napraviti. Naime, **halting problem je neodlučiv.*** Ne postoji opšti automatizovan način za proveravanje da li je neka naredba programa dostižna, pa sami tim ni da li je ispravna, odnosno da li je sam program ispravan.

Posledica teorijskog ograničenja je da nije moguće napraviti program koji bi potpuno automatski u konačnom vremenu, koristeći konačne resurse mogao da utvrdi ispravnost proizvoljnog programa poptuno precizno. Međutim, ukoliko se odreknemo potpune preciznosti, možemo da napravimo program koji poptuno automatski, u konačnom vremenu, koristeći konačne resurse može da dâ veoma korisne informacije o ispravnosti programa, iako ne u potpunosti precizne. Preciznost može da bude narušena na dva načina. Alat može da ima

- **lažna upozorenja** — da prijavljuje problem koji u realnom izvršavanju ne može da se desi,
- **propuštene greške** — da ne prijavljuje problem koji u realnom izvršavanju može da se desi.

Zanimljivost: *Rocq* je interaktivni dokazivač teorema, odnosno pomoćnik za dokazivanje. On omogućava pisanje matematičkih dokaza i formalnih specifikacija, tj. pisanje programa i dokaza da programi ispunjavaju svoje specifikacije.
Rocq je ranije bio poznat kao *Coq Proof Assistant*. Pod tim imenom je 2013. godine dobio *ACM Software System Award* kao i *ACM SIGPLAN Programming Languages Software Award*, dve najprestižnije nagrade koje softver može da dobije.

* Alan Turing, *On Computable Numbers With an Application to the Entscheidungsproblem*, Proceedings of the London Mathematical Society, 1936.

Automatski pristupi obično teže da imaju ili samo lažna upozorenja, ili samo propuštene greške — kombinovanje lažnih upozorenja sa propuštenim greškama čini alat praktično nepoželjnim.

Takođe, prilikom izgradnje automatskog alata za ispitivanje ispravnosti često se pravi kompromis između preciznosti i efikasnosti. Efikasni alati najčešće nisu precizni, dok precizni alati nisu efikasni.

U automatsku statičku analizu se ubrajaju:

- Simboličko izvršavanje
- Proveravanje modela
- Apstraktna interpretacija

Pored alata za automatsko otkrivanje grešaka u programima, formalne metode verifikacije softvera obuhvataju i tehnike razvoja ispravnog softvera. Generisanje koda direktno iz specifikacije i formalno dokazivanje ispravnosti softvera koji se razvija predstavljaju najviši nivo sigurnosti u ispravnost softvera. Ovo je ujedno i najskuplji razvoj softvera i zahteva visoko stručne ljude i upotrebu posebnih alata, kao što su npr *Rocq* (slika 3.2) i *Isabelle* (slika 3.3). Iako ovakav razvoj softvera vodi najpouzdanijem softveru, on je ujedno i najskuplji i najsporiji pa se u industriji najčešće koristi samo za razvoj kritičnih delova koda.

Rezime

- Obezbeđivanje i kontrola kvaliteta softvera obuhvataju validaciju i verifikaciju softvera.
- Validacija softvera obuhvata sve procese koji su potrebni da se osigura da definisana specifikacija softvera zadovoljava korisničke potrebe.
- Verifikacija softvera obuhvata sve procese koji su potrebni da se osigura da razvijeni softver zadovoljava zadatu specifikaciju.
- I verifikacija i validacija su od suštinske važnosti za kvalitet softvera.
- Verifikacija softvera može da bude dinamička i statička.
- Dinamička verifikacija softvera se oslanja pre svega na testiranje, ali testiranje ne može da garantuje ispravnost softvera.
- Statička verifikacija softvera uključuje ljudske preglede koda i formalne metode.

- Automatska statička verifikacija: neophodni su kompromisi između preciznosti i efikasnosti.
- Razvoj direktno iz specifikacije i formalno dokazivanje ispravnosti softvera koristi se za razvoj kritičnih delova aplikacija.

Ispitna pitanja

8. Odnos verifikacije i validacije softvera. Primeri.
9. Tehnike verifkacije softvera. Osnovna podela. Mogućnosti dinamičkog i statičkog pristupa verifikaciji softvera.

Dinamička verifikacija softvera

Testiranje 4

Pregled

- Koja je uloga testiranja u procesu razvoja softvera?
- Koje vrste testiranja postoje — šta se proverava?
- Koje tehnike testiranja postoje — kako napraviti dobre test primere?
- Koji načini sprovođenja testiranja postoje — kada koristiti manuelno, a kada automatsko testiranje?

Dinamička verifikacija softvera predstavlja skup metoda i tehnika kojima se proverava ispravnost softverskog sistema tokom njegovog izvršavanja. Među tim tehnikama, najčešće primenjivana — i svakako najrasprostranjenija forma verifikacije uopšte — jeste testiranje. Testiranje se ogleda u kontrolisanom pokretanju softvera uz korišćenje unapred definisanih skupova ulaznih podataka, pri čemu se pažljivo prati njegovo ponašanje i upoređuje sa očekivanim ishodima.

Kada se testiranje sprovodi na pravilan i sistematičan način, ono može u velikoj meri doprineti unapređenju kvaliteta softverskog proizvoda. Razumevanje toga kako, kada i zašto testirati omogućava ne samo efikasnije otkrivanje grešaka, već i oblikovanje procesa razvoja softvera tako da se od samog početka teži visokom kvalitetu konačnog rešenja. Iz tih razloga, od suštinske je važnosti da svi koji učestvuju u razvoju softvera poseduju znanje o metodologiji testiranja, kao i o njegovim procesima i osnovnim principima.

4.1 Testiranje i razvoj softvera

Softver nastaje kao odgovor na jasno definisane zahteve korisnika, koji precizno određuju šta softver treba da radi, na koji način treba da funkcioniše, u kojim uslovima, koje potrebe korisnika treba da zadovolji i kako treba da ostvaruje interakciju sa krajnjim korisnikom. Svako odstupanje softvera od ovih zahteva smatra se defektom. Defekti su neposredna posledica ljudskih grešaka napravljenih u različitim fazama

razvoja softvera — od analize i dizajna, preko implementacije, pa sve do testiranja i održavanja.

Testiranje softvera predstavlja sistematsku aktivnost u okviru životnog ciklusa razvoja, koja ima za cilj da otkrije defekte pre nego što softver dospe u ruke krajnjih korisnika. Osim što doprinosi identifikaciji i ispravljanju defekata, testiranje takođe pomaže u proceni pouzdanosti, performansi i ukupne stabilnosti softverskog proizvoda. Na taj način, ono igra ključnu ulogu u postizanju visokog kvaliteta i korisničkog zadovoljstva.

Slika 4.1: Edger Dijkstra (eng. *Edsger Dijkstra*)[a], dobitnik Tjuringove nagrade 1972. godine:

> "*Program testing can show the presence of bugs, never their absence*"

Testiranje može da potvrdi prisustvo grešaka u softveru — testiranje ne može da dokaže ispravnost softvera.

[a] Fotografiju je napravio *Hamilton Richards*. Fotografija je dostupna pod licencom *Creative Commons Attribution-Share Alike 3.0*

4.1.1 Cena grešake u kontekstu vremena otkrivanja

U savremenom razvoju softvera, osnovni cilj organizacija i razvojnih timova jeste postizanje maksimalnog profita kroz isporuku proizvoda visokog kvaliteta, ali istovremeno uz poštovanje vremenskih rokova i budžetskih ograničenja. Upravo iz tog razloga, metode i pristupi koji omogućavaju bržu, efikasniju i pouzdaniju isporuku softverskih rešenja postaju sve važniji.

Faze razvoja softvera su organizovane prema metodologiji koja se koristi za upravljanje razvojem softverskog projekta, a osnovne faze se javljaju u svim modelima razvoja. Osnovne faze razvoja softvera uključuju analizu zahteva, dizajn i implementaciju softvera, razne vrste testiranja i upotrebu softvera. Najzastupljenije i trenutno najpopularnije metodologije razvoja softvera, kao što su agilne metodologije (npr. *Scrum*, *Kanban*, *Extreme Programming*), zasnivaju se na iterativnom pristupu koji podrazumeva paralelno razvijanje i testiranje softverskih komponenti. Umesto da se testiranje odlaže za kraj procesa (kao što je to slučaj u tradicionalnim pristupima razvoju softvera), ono se integriše u svaku fazu razvoja — svaka funkcionalna celina se implementira i istovremeno proverava testovima, čime se omogućava brza povratna informacija i pravovremeno otkrivanje defekata.

Kako složenost softverskih sistema raste, proporcionalno raste i značaj testiranja. Greške koje promaknu u ranim fazama razvoja mogu da izazovu lančane probleme, koji ne samo da komplikuju završne faze projekta, već u krajnjem slučaju mogu dovesti i do potpunog neuspeha proizvoda.

Zbog toga je od suštinskog značaja da se sve greške otkriju što ranije u životnom ciklusu softvera. Ispravljanje grešaka u početnim fazama razvoja je znatno brže, jednostavnije i jeftinije u poređenju sa otklanjanjem grešaka u kasnijim fazama. To je povezano i sa brojem ljudi koje je potrebno uključiti u proces ispravljanja greške, a koji zavisi od faze razvoja softvera u kojoj se greška pronađe:

Faza analize zahteva. Greška zahteva reviziju definisanih zahteva. Trošak uključuje dodatno vreme analitičara.

Faza dizajna softvera. Ispravka greške podrazumeva izmenu dizajna, što zahteva dodatni rad dizajnera i ažuriranje opisa impementacije.

Faza implementacije softvera. Greške otkrivene tokom kodiranja obično programer brzo ispravi. Cena zavisi od složenosti, ali je niža ako programer sam pronađe grešku. U okviru implementacija softvera, posebno je bitna **faza integracije komponenti**, koja obuhvata sklapanje različitih softverskih komponenti u veće celine. Greške su u ovoj fazi teže za otkrivanje jer uključuju više različitih delova softverskog sistema. Ispravka zahteva saradnju više članova tima i potencijalno različitih razvojnih timova pa samim tim i više vremena za analizu uzroka, što utiče i na to da je ispravljanje greške skuplje.

Faza sistemskog testiranja. Cena greške otkrivene u ovoj fazi uključuje rad tima zaduženog za kvalitet softvera, prijavu greške, komunikaciju sa programerima, ponovna testiranja i praćenje kroz sistem za praćenje defekata. Ukoliko u procesu razvoja softvera postoji korak **testiranja prihvatljivosti softvera**, cena greške dodatno obuhvata i rad kupca odnosno korisnika koji vrši proveru da li je softver prihvatljiv. Korisnički otkrivene greške zahtevaju dodatnu koordinaciju sa timom zaduženim za kvalitet softvera i povratnu integraciju ispravki. Trošak uključuje vreme korisnika i produžava završne faze projekta.

Faza upotrebe. Greške u produkciji imaju najveću cenu. Pored tehničke ispravke, dolazi do gubitka poverenja korisnika i potencijalne reputacione štete.

4.1.2 Uloga testera u razvoju softvera

Projekti u oblasti razvoja softvera mogu se voditi na najrazličitije načine. Briga o kvalitetu softvera podrazumeva primenu

različitih pravila, procedura i praksi, koje zavise kako od vrste projekta, tako i od zrelosti i veličine same kompanije. U nekim slučajevima kvalitet softvera je odgovornost celog razvojnog tima, dok u drugim postoji posebno definisan tim zadužen za ovu oblast.

Tim za obezbeđenje kvaliteta softvera može postojati kao zasebna organizaciona jedinica unutar kompanije koja razvija softver (interni tim), ili može biti eksterno angažovan od strane specijalizovane firme koja pruža usluge testiranja i kontrole kvaliteta. U nekim kompanijama, posebno u manjim i manje formalnim okruženjima, ovakav tim uopšte ne postoji, a aktivnosti testiranja programeri obavljaju *ad-hoc*.

Tester softvera (eng. *software tester*) ili jednostavno tester je stručnjak koji se bavi planiranjem, izvođenjem i dokumentovanjem aktivnosti testiranja softverskog sistema, sa ciljem da se obezbedi očekivani kvalitet proizvoda. Njegov osnovni zadatak jeste sistematsko ispitivanje softvera kako bi se identifikovali defekti u funkcionisanju, performansama ili upotrebljivosti.

Raspodela obaveza između programera i testera u timovima može biti organizovana na različite načine:

- programeri su istovremeno i testeri i sami proveravaju svoj kôd,
- programeri i testeri rade zajedno u istom timu, blisko sarađujući,
- programeri i testeri čine potpuno odvojene timove sa jasnim razgraničenjem odgovornosti.

Slika 4.2: Osnovni razlog neslaganja testera i programera. [a]

[a] Crtež je generisao alat *OpenArt*

Iako je kvalitetan softver krajnji cilj i programera i testera, često postoji veliki broj problema na relaciji programer — tester onda kada su programeri i testeri u razdvojenim timovima. Problemi su u lošoj komunikaciji, međusobnom nerazumevanju i opštoj netrpeljivosti. Ovi problemi su najčešće posledica psihološke reakcije na pronalaženje greške u softveru. Pronalaženje greške u softveru testeru označava da je uspešno obavio svoj zadatak, dok programeru to znači da nije uspešno obavio svoj posao i da je potrebno ponovo da radi na nekom delu koda za koji je verovao da je završen (slika 4.2).

Da bi se osoba bavila testiranjem potrebno je da poseduje osnovno razumevanje programiranja i procesa razvoja softvera. Posebno, mora da detaljno poznaje procedure i proces testiranja kao i alate koji se koriste u testiranju. Za efikasnu

implementaciju i automatizaciju testiranja neophodno je i poznavanje skript jezika i skript programiranja.

Dobri testeri su kreativni i imaju potrebu za stalnim usavršavanjem, a vremenom upoznaju i česte greške i propuste kao i nesvakidašnje slučajeve upotrebe, što značajno olakšava i ubrzava proces testiranja. Za poslove testiranja još uvek je dominantna neformalna edukacija kroz kurseve, konferencije i sastanke profesionalnih udruženja. Za napredne poslove automatizacije u testiranju neophodne su napredne programerske veštine.

4.1.3 Faze testiranja softvera

Za početak procesa testiranja, postojanje koda nije neophodno. Dovoljno je imati jasno definisane zahteve korisnika jer priprema za testiranje počinje analizom tih zahteva. Dakle, da bi se započeli procesi vezani za testiranje, potrebno je da postoji specifikacija zahteva sistema (eng. *system requirements specification*) kao i specifikacija zahteva softvera (eng. *software requirements specification*).

Testiranje softvera se u opštem slučaju sastoji od narednih faza, pri čemu svaka faza obuhvata veliki broj aktivnosti.

1. Planiranje testiranja
2. Analiza, dizajn i implementacija testova
3. Izvršavanje testova
4. Evaluacija testova
5. Zatvaranje testiranja

Ove faze se usklađuju sa metodologijom razvoja softvera tako da se u okviru iterativnih modela razvoja izvršavaju iterativno.

Planiranje testiranja

Planiranje testiranja (eng. *test planning*) predstavlja pripremu za proces testiranja. Plan testiranja se prilagođava svakom konkretnom projektu.

Planiranje testiranja započinje analizom zahteva, pri čemu se detaljno razmatraju svi funkcionalni i nefunkcionalni aspekti sistema. Na osnovu te analize definišu se ciljevi testiranja, odnosno šta se želi postići samim testiranjem. Sledeći korak obuhvata određivanje opsega testiranja, uključujući odluku

Planiranje definiše:

- potrebne vrste testova,
- tehnike testiranja koje će se koristiti,
- načine sprovođenja testiranja,
- opseg testiranja,
- ulazne i izlazne kriterijume, posebno kriterijum završetka,
- potrebne resurse,
- procenu rizika,
- metodologiju praćenja defekata i
- uloge i način komunikacije između članova tima.

o tome koje će komponente softvera biti testirane i koliko. Dodatno, određuju se i načini testiranja: koji će delovi testiranja biti sprovedeni ručno, a koji automatizovano, uključujući i alate koji će biti korišćeni tokom testiranja. Planiranje takođe uključuje precizno definisanje uloga i odgovornosti članova tima.

Važan deo planiranja je i procena rizika, koja uključuje identifikaciju potencijalnih problema i njihov uticaj na kvalitet proizvoda. Vremenski okvir testiranja se takođe detaljno planira, uz jasno definisane ulazne i izlazne kriterijume koji određuju kada testiranje može da počne i pod kojim uslovima se smatra završenim. Konačno, planiranje uključuje pripremu test okruženja, odnosno obezbeđivanje potrebnog hardverskog i softverskog okruženja neophodnog za efikasno i pouzdano izvođenje testova.

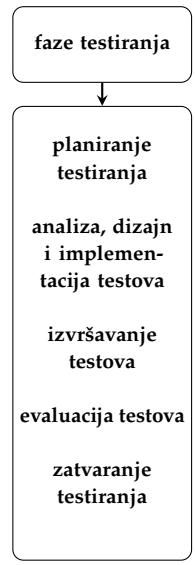
Neizostavan deo plana je i metodologija za praćenje defekata, koja precizno definiše kako se greške prijavljuju, dokumentuju, prate tokom životnog ciklusa i na kraju zatvaraju, čime se obezbeđuje kontrola kvaliteta i transparentnost celokupnog procesa testiranja.

Analiza, dizajn i implementacija testova

Analiza, dizajn i implementacija testova (eng. *test analysis, design and implementation*) predstavljaju ključne faze u okviru procesa testiranja softverskih sistema. Ove aktivnosti podrazumevaju sistematski pristup u osmišljavanju načina na koji će se sprovesti testiranje, u skladu sa prethodno definisanim test planom.

Jedan od najbitnijih rezultata ovih faza je koherentan i sveobuhvatan skup test slučajeva i procedura koji služe kao temelj za uspešno sprovođenje testiranja i verifikaciju ispravnosti softverskog rešenja. Test slučaj (eng. *test case*) predstavlja formalni dokument koji opisuje određenu situaciju testiranja kroz definisani skup ulaznih podataka i očekivane izlaze sistema. Svaki test slučaj ima za cilj da proveri konkretnu funkcionalnost softvera, njegovo ponašanje u graničnim uslovima ili otpornost na nevalidne ili neočekivane ulaze. Procedura testiranja (eng. *test procedure*) definiše tačan redosled i način primene test slučajeva u kontrolisanom okruženju. Testni okvir, testno okruženje ili infrastruktura za testiranje (eng. *test harness*) je skup skripti, alata i konfiguracija koji omogućavaju

automatsko pokretanje testova, simulaciju realnog upotrebnog okruženja i prikupljanje rezultata.



U fazi analize testova, vrši se detaljno ispitivanje mogućnosti testiranja pojedinih delova softverskog sistema i identifikaciju zahteva koje sistem mora da ispuni. Ovaj proces podrazumeva detaljno razlaganje korisničkih priča (eng. *user story*), upotrebnih scenarija (eng. *use cases*) i tehničke dokumentacije, kako bi se identifikovali svi elementi koji mogu i treba da budu predmet testiranja. Kroz identifikaciju zavisnosti, ograničenja i potencijalnih rizika, analiza testova gradi detaljnu mapu puta koji vodi ka dizajnu testova i njihovom kasnijem izvršavanju. Time se obezbeđuje sveobuhvatna pokrivenost svih relevantnih aspekata softverskog sistema.

Primer 4.1.1 (Aplikacija za elektronsko bankarstvo) Razmotrimo projekat razvoja aplikacije za elektronsko bankarstvo. Tokom faze analize testova, testeri analiziraju funkcionalne zahteve sistema. Jedan od njih je i

> Korisnici treba da mogu da prenose novac između računa.

Na osnovu ovog zahteva, identifikuju se različiti scenariji testiranja, kao što su:

- Prenos novca između računa u istoj banci
- Prenos ka računima u drugim bankama
- Testiranje granica – npr. minimalni i maksimalni iznos za prenos

Pored toga, identifikuju se i granični slučajevi (eng. *edge cases*), kao što su:

- Pokušaj prenosa novca kada korisnik nema dovoljno sredstava na računu
- Pokušaj ponovljenog prenosa usled pada mreže

Tabela 4.1: Poređenje: test slučaj, procedura testiranja i testni okvir

Pojam	**Opis**
Test slučaj (test)	Konkretna situacija koja se testira, sa definisanim ulazima i očekivanim izlazima.
Procedura testiranja	Skup jasno definisanih koraka koji opisuju kako se jedan ili više test slučajeva izvršavaju u praktičnom okruženju.
Testni okvir	Skup alata i infrastrukture koji omogućavaju automatsko izvršavanje testova, simulaciju delova sistema i prikupljanje rezultata testiranja.

- Neispravno unet broj računa primaoca

Svi ovi identifikovani scenariji biće u narednim fazama formalizovani u test slučajeve, čime se obezbeđuje da funkcionalnost prenosa novca bude testirana u celosti — sa aspekta tačnosti, pouzdanosti i bezbednosti. Na ovaj način, analiza testova ne samo da razbija kompleksne zahteve na razumljive i proverljive jedinice, već i postavlja temelj za kvalitetan dizajn i izvršavanje testova, koji direktno doprinosi uspešnosti krajnjeg softverskog proizvoda.

Analiza, dizajn i implementacija testova:

- Faza analize testova daje odgovor na pitanje „šta testirati?".
- Faza dizajna testova daje odgovor na pitanje „kako testirati?".
- Faza implementacije daje sve što je potrebno da testiranje može da počne.

Faza dizajna testova sledi nakon analize i predstavlja temelj za dalji tok testiranja. U ovoj fazi definišu se ciljevi testiranja, identifikuju funkcionalnosti koje treba proveriti i preciziraju uslovi pod kojima će testovi biti sprovedeni. Identifikovani scenariji se razrađuju u test slučajeve i prateću dokumentaciju (eng. *testware*), čime se obezbeđuje jasan i sistematičan pristup. Ključne aktivnosti u okviru dizajna testova obuhvataju:

- kreiranje i prioritizaciju test slučajeva,
- identifikaciju potrebnih test podataka,
- preciziranje testnog okruženja, uključujući potrebnu infrastrukturu i alate.

Cilj ove faze je da se obezbedi dosledna i kvalitetna osnova za kasniju implementaciju i izvršavanje testova.

Primer 4.1.2 (Aplikacija za elektronsko bankarstvo — nastavak) Razmotrimo identifikovan scenario testiranja:

> Pokušaj prenosa novca kada korisnik nema dovoljno sredstava na računu.

U fazi dizajna testova, za ovaj scenario, može se identifikovati naredni skup test slučajeva:

1. **Apsolutno nedovoljno sredstava**
   Korisnik ima 0 RSD na računu, a pokušava da prenese npr. 1000 RSD.
   Očekivanje: Transakcija odbijena; jasna poruka o nedostatku sredstava.
2. **Delimično pokrivanje iznosa**
   Korisnik ima 500 RSD na računu, a pokušava da prenese 1000 RSD.
   Očekivanje: Transakcija odbijena; jasna poruka o nedostatku sredstava, ali potrebno je testirati i prikaz

informacija o preostalom saldu i sugestiju da se izvrši prenos manjeg iznosa.

3. **Nedovoljno sredstava zbog provizije**
   Korisnik ima dovoljno novca na računu, ali se naplaćuje provizija za prenos.
   Očekivanje: Transakcija odbijena uz poruku o dodatnoj proviziji.
4. **Nedovoljno sredstava zbog rezervacije sredstava**
   Na računu piše da ima dovoljno sredstava, npr. 1000 RSD, ali je deo sredstava rezervisan, npr. 800 RSD.
   Očekivanje: Efektivno dostupno stanje je 200 RSD; prenos većeg iznosa treba da bude odbijen.
5. **Nedovoljno sredstava u drugoj valuti**
   Račun je u dinarima, a korisnik pokušava prenos u evrima. Nakon konverzije, stanje nije dovoljno.
   Očekivanje: Sistem treba da prikaže da konvertovani iznos nije dovoljan.
6. **Ponovljeni pokušaji prenosa bez sredstava**
   Višestruki pokušaji prenosa istog iznosa bez pokrića.
   Očekivanje: Sistem blokira pokušaje, potencijalno uvodi vremensko ograničenje ili detekciju zloupotrebe.

Dodatno, potrebno je precizirati sve identifikovane test slučajeve. Na primer, test slučaj za prenos novca u slučaju delimičnog pokrivanja iznosa je dat u okviru primera 4.2.3.

Ove test slučajeve može da izvršava tester, po zadatim koracima, a može se i napisati skript koji automatizuje njihovo izvršavanje i simulira sve korisničke korake. Ukoliko je predviđeno da se testovi izvršavaju automatizovano, u fazi implementacije obezbeđuje se kôd koji je potreban za njihovo izvršavanje.

Faza implementacije testova predstavlja završni korak u pripremi za izvršavanje testiranja. U ovoj fazi dolazi do konkretizacije prethodno dizajniranih test slučajeva kroz njihovo organizovanje u procedure testiranja, kao i pripreme potrebne testne dokumentacije, alata i okruženja. Cilj implementacije je da obezbedi doslednost, ponovljivost i tehničku spremnost za efikasno izvršavanje testova.

Ključne aktivnosti koje se sprovode tokom implementacije testova uključuju:

- **Razvoj i prioritizaciju procedura testiranja, uključujući i razvoj test okvira** — omogućava strukturisano

i, gde je primenljivo, automatizovano izvršavanje test slučajeva.
- **Formiranje test kompleta** (eng. *test suites*) grupisanjem povezanih procedura i skripti — radi lakše organizacije i povećanja efikasnosti tokom izvršavanja.
- **Raspoređivanje test kompleta u plan izvršavanja** — u skladu sa raspoloživim resursima i prioritetima.
- **Pripremu test podataka i proveru njihove integracije u okruženje** — kako bi se obezbedila validnost ulaza i pouzdanost rezultata testiranja.

Na ovaj način, implementacija testova zatvara krug pripremnih aktivnosti i postavlja temelj za sledeću fazu — sâmo izvršavanje testiranja.

Primer 4.1.3 (Aplikacija za elektronsko bankarstvo — nastavak) **Naziv test procedure:** Testiranje prenosa sredstava u uslovima nedovoljnog stanja na računu

Opis: Ova test procedura pokriva slučajeve kada korisnik pokušava da izvrši prenos novca, ali na računu nema dovoljno sredstava za realizaciju transakcije. Cilj je da se verifikuje ispravno ponašanje sistema u različitim varijacijama ovog problema, uključujući osnovni nedostatak sredstava, rezervacije, provizije i valute.

Preduslovi:

- Korisnički nalog je kreiran i verifikovan.
- Korisnik ima pristup elektronskom bankarstvu.
- Test račun je aktivan i koristi dinarsku valutu (RSD).
- Testno okruženje koristi realne validacione mehanizme.

Koraci:

Test slučaj 1: Apsolutno nedovoljno sredstava
Test slučaj 2: Delimično pokrivanje iznosa
Test slučaj 3: Nedovoljno sredstava zbog provizije
Test slučaj 4: Nedovoljno sredstava zbog rezevisanih sredstava
Test slučaj 5: Nedovoljno sredstava u drugoj valuti

Završni uslovi:

- Sve test transakcije treba da budu evidentirane u log fajlovima.
- Nije dozvoljena promena stvarnog salda korisnika.

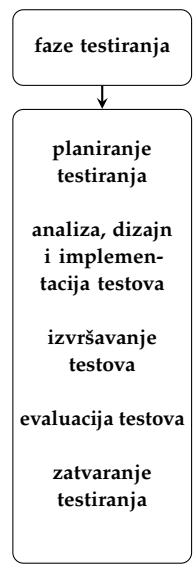
- Sistem mora prikazati odgovarajuće poruke greške.

Primetimo da test slučaj vezan za ponovne uzastopne pokušaje podizanja novca kada na računu nema dovoljno sredstava može takođe da se uvrsti u ovu proceduru testiranja ili da se grupiše u drugu proceduru testiranja, na primer, u neku koja se odnosi na sigurnost sistema.

Izvršavanje testova

Izvršavanje testova (eng. *test execution*) predstavlja fazu u kojoj se praktično primenjuju test slučajevi i test procedure razvijene tokom prethodnih faza. Testovi se mogu sprovoditi ručno – kada tester sam prati korake iz test slučaja – ili automatski, korišćenjem specijalizovanih alata koji omogućavaju bržu proveru funkcionalnosti.

Prilikom testiranja, neophodna je organizacija testiranja kako bi se testovi izvršavali u skladu sa prioritetima i efikasno, bez gubljenja resursa i vremena. To uključuje raspoređivanje testova, kontrolu test okruženja i praćenje napretka test aktivnosti.



Izvršavanje testova predstavlja ključni korak u proverenju funkcionalnosti sistema i verifikaciji da softver ispunjava postavljene zahteve. Tokom ove faze, testovi se sistematski sprovode kako bi se otkrile greške, ali i kako bi se pratio trenutni status svih prethodno prijavljenih problema. Dakle, pored samog testiranja, važan deo ove aktivnosti jeste praćenje i upravljanje greškama – to podrazumeva ne samo prijavu otkrivenih problema, već i njihovo kontinuirano praćenje, proveru ispravki i konačnu potvrdu da su greške uspešno otklonjene. Svaka korekcija zahteva ponovno izvršavanje relevantnih testova kako bi se osiguralo da popravka nije dovela do novih problema ili narušila postojeću funkcionalnost.

Ova faza testiranja zahteva intenzivnu i jasnu komunikaciju između testera i programera. Bliska saradnja i razmena informacija omogućavaju bržu identifikaciju uzroka grešaka, efikasnije rešavanje problema i osiguranje stabilnosti sistema u završnim fazama razvoja.

Evaluacija testova

Evaluacija testova (eng. *test evaluation*) obuhvata procenu kriterijuma završetka testiranja i izveštavanje. Svaka izmena

u kodu, čak i koja podrazumeva popravljanje grešaka, može da dovede do novih grešaka. Iz tog razloga se, za različite oblasti testiranja, definiše kriterijum završetka testiranja u odnosu na rezultate izvršavanja testova, procenta nerešenih bagova ili preostalog vremena za testiranje. Proces evaluacija uključuje i pregled rezultata dobijenih analizom izlaza test slučajeva.

Izlazni kriterijumi (eng. *exit criteria*) određuju da li je testiranje kompletirano i da li je aplikacija spremna za korišćenje u skladu sa korisničkim zahtevima. Da bi se odredilo da li su izlazlni kriterijumi ispunjeni, potrebno je uzeti u obzir rezultate testiranja date kroz sažeti izveštaj testiranja (eng. *test summary report*), rezultate izračunavanja raznih metrika i izveštaj o defektima (eng. *defect analysis report*).

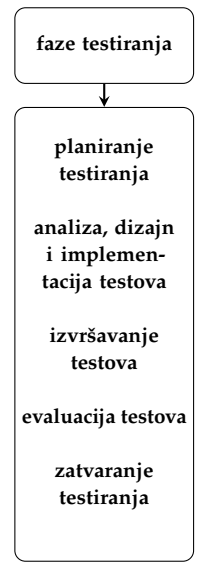
Primer 4.1.4 (Aplikacija za elektronsko bankarstvo — nastavak) Primer izlaznih kriterijuma za elektronsko bankarstvo dat je u nastavku.

1. **Pokrivenost testovima:** 95% definisanih funkcionalnih i nefunkcionalnih zahteva mora biti pokriveno odgovarajućim test slučajevima.
2. **Prolaznost test slučajeva:** 100% kritičnih test slučajeva i najmanje 98% svih ostalih test slučajeva mora biti uspešno izvršeno.
3. **Odsustvo kritičnih i visoko-prioritetnih grešaka:** Nisu dozvoljene greške koje utiču na osnovnu funkcionalnost sistema (npr. greške koje onemogućavaju prijavu korisnika, pregled stanja računa, ili izvršavanje transakcija).
4. **Potvrđena sigurnost sistema:** Sigurnosne provere moraju potvrditi da sistem ne sadrži ranjivosti koje bi mogle dovesti do neautorizovanog pristupa, curenja podataka ili drugih sigurnosnih incidenata.
5. **Stabilnost u test okruženju:** Aplikacija mora stabilno funkcionisati tokom višednevnog testiranja bez rušenja, curenja memorije ili degradacije performansi.
6. **Obezbeđena testna dokumentacija:** Sva testna dokumentacija (test planovi, zapisnici o testiranju, rezultati testova, prijave grešaka) mora biti ažurna i kompletna.
7. **Odobrenje zainteresovanih strana:** Tim za testiranje, razvoj, bezbednost i poslovni korisnici moraju formalno

potvrditi da su svi zahtevi zadovoljeni i da je sistem spreman za dalji proces.

Zatvaranje testiranja

Zatvaranje testiranja označava završnu fazu test procesa, koja nastupa kada su svi planirani testovi sprovedeni i softver je isporučen krajnjem korisniku, bez daljih obaveza održavanja. Međutim, ovakva situacija je u praksi veoma retka, jer se nakon isporuke softvera gotovo uvek podrazumeva i njegovo održavanje, u okviru kog se testiranje i dalje sprovodi — u vidu provera ispravki, unapređenja funkcionalnosti ili novih verzija softvera.



Testiranje se može zatvoriti i u drugim okolnostima — na primer, u slučaju da je projekat otkazan, ako su ciljevi testiranja ostvareni ranije nego što je planirano, ili ako nema više smisla nastavljati testiranje zbog promene poslovnih prioriteta.

Tokom faze zatvaranja testiranja, vrši se arhiviranje test slučajeva, izveštaja i prateće dokumentacije, čime se omogućava budući uvid u realizovane aktivnosti. Istovremeno, sprovodi se analiza samog procesa testiranja, uz identifikaciju onih praksi koje su se pokazale uspešnima i koje bi trebalo zadržati i primeniti u budućim projektima.

Podjednako je važno prepoznati i aspekte koji nisu funkcionisali dobro, kako bi se greške iz prošlosti izbegle i unapredila efikasnost testiranja u narednim iteracijama. Na taj način, zatvaranje testiranja ne označava samo kraj jedne faze, već i priliku za učenje i stalno poboljšanje procesa.

4.2 Vrste i nivoi testiranja

Testiranje softvera može se klasifikovati na više načina, u zavisnosti od cilja, obima i nivoa na kojem se testiranje sprovodi. Osnovna podela testiranja odnosi se na prirodu osobina koje se proveravaju, pa razlikujemo:

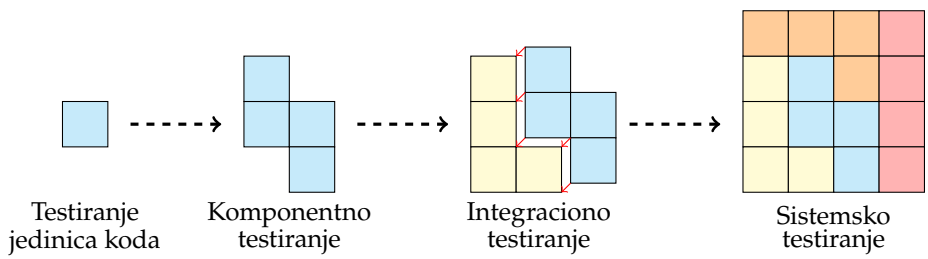
Testiranje funkcionalnih karakteristika usmereno je na proveru da li aplikacija pravilno izvršava funkcije definisane specifikacijom zahteva. Ova vrsta testiranja ispituje ponašanje sistema u odnosu na očekivane ulaze i izlaze.

Testiranje nefunkcionalnih karakteristika usmereno je na tehničke i kvalitativne aspekte sistema, kao što su performanse, sigurnost, upotrebljivost, kompatibilnost i pouzdanost.

Druga bitna podela je podela po **nivoima testiranja** (slika 4.3). Mogu se testirati

- osnovne jedinice koda,
- pojedinačni moduli,
- grupe modula (vezanih namenom, upotrebom, ponašanjem ili strukturom) ili
- ceo sistem.

U skladu sa pomenutom podelom, prema nivou testiranja, razlikujemo testove jedinice koda, komponentne, integracione i sistemske testove. Na svakom nivou mogu se testirati funkcionalne i nefunkcionalne karakteristike softvera.



Slika 4.3: Nivoi testiranja

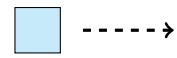
Posebna vrsta testiranja koja se može javiti u okviru svakog nivoa je regresiono testiranje. Regresiono testiranje (eng. *regression testing*) predstavlja proces ponovnog izvršavanja prethodno uspešno završenih test slučajeva s ciljem da se proveri da li novouvedene izmene u softveru nisu nenamerno dovele do *regresije*, tj. narušile postojeću funkcionalnost sistema ili dovele do pada performansi. Regresiono testiranje se sprovodi uvek nakon ispravki grešaka, dodavanja novih funkcionalnosti i prilikom refaktorisanja koda.

4.2.1 Testiranje jedinica koda

Testiranje jedinica koda (eng. *unit testing*) predstavlja proveru ispravnosti najmanjih delova softverskog sistema koji se mogu nezavisno testirati. U zavisnosti od programskog paradigme, jedinice koda mogu biti različite: u objektno orijentisanom

programiranju to su najčešće klase, u funkcionalnom programiranju funkcije, dok se u imperativnom programiranju obično testiraju procedure i pojedinačni moduli.

Ova vrsta testiranja omogućava detaljan uvid u ponašanje svake pojedinačne jedinice, čime se osigurava stabilna osnova za kasniju integraciju u veće sisteme. *1008-1987 IEEE Standard for Software Unit Testing* je standard koji definiše jedinično testiranje i koji postavlja smernice i preporuke za njegovo sprovođenje.



Testiranje jedinica koda

Jedna od ključnih prednosti testiranja jedinica koda jeste snažna podrška u alatima za automatsko izvršavanje i proveru rezultata rada testova, koji su često integrisani u savremena razvojna okruženja. Automatizacija ovih testova omogućava brzu i čestu proveru funkcionalnosti u toku razvoja.

Cilj jediničnih testova je da se potvrdi da izolovani delovi koda funkcionišu u skladu sa očekivanjima. Kada jedinica koda komunicira sa spoljnim resursima, kao što su mreža, baze podataka ili fajl sistemi, ta komunikacija se u test okruženju zamenjuje fiksiranim (eng. *mock*) vrednostima. Isto važi i za komunikaciju sa drugim klasama ili komponentama – sve se apstrahuje, kako bi se test fokusirao isključivo na ponašanje posmatrane jedinice. Dozvoljena je samo interakcija sa memorijom.

Ove testove najčešće pišu sami programeri tokom razvoja, što omogućava brzo otkrivanje i otklanjanje grešaka. Ukoliko postoji problem unutar same jedinice koda, upravo ova faza testiranja treba da ga otkrije — pre nego što dođe do složenijih i skupljih grešaka u kasnijim fazama razvoja.

Primer 4.2.1 (Biblioteka `unittest` u jeziku Python) Funkcija `saberi(a, b)` je jednostavna funkcija koja vraća zbir dva broja i definisana je u modulu `matematika.py`.

```
    def saberi(a, b):
    return a + b
```

Naredni kôd (datoteka `test_matematika.py` koja je u istom direktorijumu sa datotekom `matematika.py`) sadrži jednostavan jedinični test za ovu funkciju.

```
import unittest
from matematika import saberi

class TestSaberi(unittest.TestCase):
    def test_saberi_pozitivne(self):
```

```
        self.assertEqual(saberi(2, 3), 5)

    def test_saberi_negativne(self):
        self.assertEqual(saberi(-1, -1), -2)

if __name__ == '__main__':
    unittest.main()
```

Klasa `TestSaberi` nasleđuje klasu `unittest.TestCase` i sadrži dva testa:

- Jedan testira sabiranje pozitivnih brojeva.
- Drugi testira sabiranje negativnih brojeva.

Funkcija `assertEqual` proverava da li je rezultat funkcije jednak očekivanoj vrednosti. Funkcija `unittest.main()` pokreće ove testove.

Test se pokreće iz komandne linije narednom naredbom

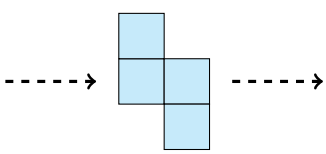
```
python test_matematika.py
```

Ako je sve ispravno, izlaz će izgledati ovako:

```
..
----------------------------------------------------
Ran 2 tests in 0.001s

OK
```

4.2.2 Komponentno i integraciono testiranje



Komponentno testiranje (eng. *component testing*) proverava ispravnost komponente. Komponenta je skup funkcionalno povezanih jedinica koda koje zajedno obavljaju jednu logičku celinu i imaju jasno definisan interfejs. Komponentno testiranje se obično sprovodi izolovano od ostatka sistema i odmah nakon razvoja komponente.

Iako se po načinu izvođenja komponentno testiranje često nadovezuje na jedinično testiranje, osnovna razlika je u nivou koji obuhvataju — dok jedinično testiranje proverava pojedinačne jedinice koda, komponentno testiranje se fokusira na veće celine koje sadrže više jedinica koda i njihovu međusobnu saradnju. Sama jedinica koda je u prethodnoj fazi testiranja izolovana u potpunosti od spoljašnjeg sistema, dok je u okviru komponentnog testiranja sada napuštena izolacija na nivou same komponente, ali i dalje ostaje izolacija u okviru povezanosti sa drugim komponentama i spoljašnjim sistemom.

S obzirom da se u komponentnom testiranju integrišu osnovne jedinice koda, ovo je vrsta integracionog testiranja, tako da se često i ne izdvaja kao posebna vrsta testiranja. Pošto se odnosni na direktno spajanje jedinica koda, može se shvatiti i kao integraciono testiranje na najnižem nivou. U praksi, komponentno testiranje mogu obavljati i programeri i testeri, u zavisnosti od organizacije projekta i kompleksnosti testiranih komponenti.

Integraciono testiranje (eng. *integration testing*) predstavlja fazu u procesu testiranja u kojoj se proverava saradnja između više softverskih komponenti koje zajedno čine funkcionalne celine sistema. Integraciono testiranje ispituje ispravnost interfejsa i komunikacije među komponentama. Cilj ovog testiranja je da se utvrdi da li su veze između komponenti pravilno definisane i implementirane, odnosno da li različiti delovi sistema međusobno komuniciraju na način koji je predviđen specifikacijom projekta. Na taj način proverava se funkcionalna ispravnost u kontekstu saradnje, a ne samo u izolaciji.

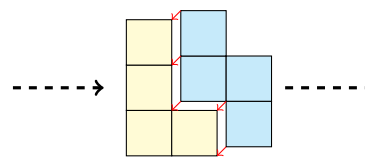
Slika 4.4: Brava

Pretpostavimo da imamo bravu kao na slici 4.4. Ukoliko takva brava treba da se integriše sa kliznim vratima (vratima koja se pomeraju levo-desno, a ne napred-nazad), bez obzira što su i brava i vrata ispravni, njihova integracija neće davati željene rezultate.

Integracioni testovi otkrivaju razne vrste problema, uključujući:

- nekompatibilnost podataka pri razmeni između modula,
- neusklađene formate, tipove ili protokole komunikacije,
- pogrešno tumačenje interfejsa ili očekivanih vrednosti,
- greške u redosledu izvršavanja aktivnosti.

Ove testove obično sprovode testeri, iako je u praksi česta saradnja sa programerima, naročito u složenim sistemima. Dobra praksa je da se integraciono testiranje sprovodi inkrementalno — kako se nove komponente razvijaju i dodaju sistemu, tako se istovremeno proverava njihova integracija.



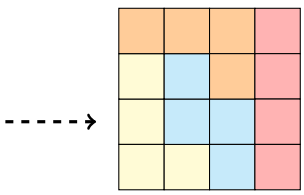
Integraciono testiranje

Primer 4.2.2 (Realni brojevi) Dve softverske komponente, *Aproksimacija* i *Optimizacija*, vrše računanja sa realnim brojevima. Međutim, zbog nedovoljno precizne specifikacije, jedna komponenta koristi realne brojeve jednostruke tačnosti (*float*), dok druga koristi realne brojeve dvostruke tačnosti (*double*). Iako svaka od komponenti zasebno funkcioniše ispravno u okviru svojih testova, njihova integracija dovodi do neočekivanih odstupanja u rezultatima. Na primer, vrednost *double d = 1234567.89;* kada se preba-

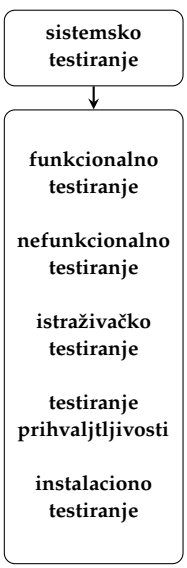
ci u *float* postaje *1234567.875* što doprinosi nepreciznosti izračunavanja.

Upravo kroz integraciono testiranje, takva neusklađenost u tipovima podataka treba da se otkrije. Testovi bi trebalo da obuhvate međusobnu razmenu podataka između komponenti i da provere konzistentnost vrednosti, preciznost proračuna i stabilnost u radu. Kada se ovakva greška otkrije, neophodno je uskladiti podatkovne tipove kako bi se osigurala kompatibilnost komponenti i sprečile greške u računanju.

4.2.3 Sistemsko testiranje



Sistemsko testiranje



Sistemsko testiranje (eng. *system testing*) obuhvata proveravanje sistema kao celine. Ispituje se da li je ponašanje sistema u skladu sa specifikacijom zadatom od strane klijenta. Ovde se zahteva i potpun pristup svim delovima sistema, uključujući bazu podataka, ukoliko se koristi, pristup mreži i svim hardverskim delovima sistema.

Sistemsko testiranje uključuje i funkcionalne i nefunkcionalne aspekte sistema. U sistemsko testiranje se ubrajaju i istraživačko testiranje, testiranje prihvatljivosti i instalaciono testiranje.

Funkcionalno sistemsko testiranje

Funkcionalno sistemsko testiranje predstavlja proveru funkcionalnosti sistema u uslovima koji odgovaraju stvarnom korišćenju. Cilj funkcionalnog sistemskog testiranja je da se potvrdi da su sve očekivane funkcionalnosti sistema implementirane (funkcionalna potpunost), da svaka funkcionalnost pojedinačno radi u skladu sa zahtevima korisnika (funkcionalna ispravnost) kao i da ne postoje neprikladne funkcionalnosti sistema (funkcionalna prikladnost). Ovo testiranje obuhvata provere ulaza, obrada i izlaza za različite funkcionalne scenarije, uključujući tipične, granične i nesvakidašnje slučajeve upotrebe. Dodatno, potrebno je evaluirati i ponašanje sistema u slučaju pogrešnih ili neočekivanih ulaza.

Primer 4.2.3 (Aplikacija za elektronsko bankarstvo — nastavak) Primer test slučaja za funkcionalno testiranje.

Slika 4.5: Vrste sistemskog testiranja

Naziv test slučaja: TC-002 — Prenos novca sa nedovoljno sredstava na računu — Delimično pokrivanje iznosa

Opis: Proverava da li aplikacija pravilno obrađuje pokušaj prenosa novca kada na korisničkom računu nema dovoljno sredstava.

Preduslovi:

- Korisnik je prijavljen u aplikaciju

- Na računu korisnika nalazi se manje sredstava od iznosa koji želi da prenese (npr. stanje: 500 RSD)
- Primaoc ima validan račun u sistemu

Test podaci:

- Iznos za prenos: **1000 RSD**
- Broj računa primaoca: **170-1234567890123-45**

Koraci:

1. Prijaviti se u aplikaciju sa validnim korisničkim nalogom
2. Otvoriti sekciju „Prenos sredstava“
3. Uneti iznos prenosa: 1000 RSD
4. Uneti broj računa primaoca
5. Kliknuti na dugme „Potvrdi prenos“

Očekivani rezultat:

- Sistem prikazuje poruku o grešci: *„Nedovoljno sredstava na računu“*
- Sistem sugeriše prenos manje količine novca
- Prenos se **ne izvršava**
- Stanje na računu ostaje nepromenjeno

Stvarni rezultat: *(popunjava se nakon izvršavanja testa)*

Status: *(PASS / FAIL – popunjava se nakon testiranja)*

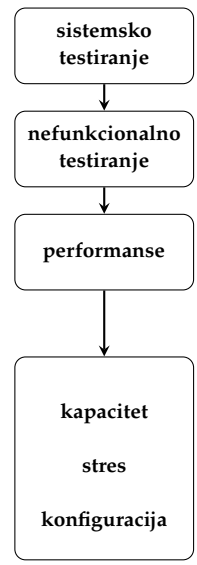
Uspešno funkcionalno testiranje potvrđuje da sistem zadovoljava svoje osnovne zahteve i da je spreman za naredne faze verifikacije, uključujući nefunkcionalna testiranja, istraživačko testiranje i konačno prihvatanje od strane krajnjih korisnika.

Nefunkcionalno sistemsko testiranje

Nefunkcionalno sistemsko testiranje obuhvata testiranje dinamičkih nefunkcionalnih aspekata sistema: performanse, kompatibilnost, pouzdanost, upotrebljivost, bezbednost, sigurnost i prenosivost. Testovi se sprovode za one aspekte kvaliteta softvera koji su za dati sistem prepoznati kao važni i koji su stoga precizirani specifikacijom. Na primer,

Testovima performansi proverava se vremensko ponašanja aplikacije i njena upotreba resursa. Ova vrsta testiranja

se fokusira na metrike kao što su vreme odgovora na zahteve, broj obrađenih zahteva u jedinici vremena, vreme obrade podataka, kao i iskorišćenost memorije i drugih sistemskih resursa. Sastavni deo testiranja performansi su i testovi kapaciteta, stresa i konfiguracije:



Testovima kapaciteta proverava se ponašanje sistema pri obradi velikih količina podataka. Posebna pažnja posvećuje se granicama sistema, tj. kako softver funkcioniše kada se približi maksimalnim kapacitetima definisanim u zahtevima.

Stres testovima se proverava kako se sistem ponaša kada ga izložimo zahtevima izvan njegovog projektovanog kapaciteta, odnosno šta se dogodi kada sistem preopteretimo i kako se sistem oporavlja kada se opterećenje smanji.

Testovima konfiguracije proverava se rad sistema u različitim hardverskim i softverskim okruženjima. Time se obezbeđuje da aplikacija funkcioniše stabilno bez obzira na konkretne kombinacije operativnih sistema, drajvera, baza podataka ili uređaja.

Primer 4.2.4 (Elektronska bankarska aplikacija — nastavak) Testovi kapaciteta su ključni za planiranje infrastrukture i predviđanje kapaciteta u rastu korisničke baze. Rezultati mogu ukazati na potrebu za horizontalnim skaliranjem ili optimizacijom baze podataka.
Naziv test slučaja: TC-C003 — Test kapaciteta

Cilj testa:
Proceniti ponašanje sistema pri obradi velikog broja korisničkih zahteva i velikih količina podataka u periodima maksimalnog opterećenja, kako bi se utvrdili granični kapaciteti sistema.

Scenario:
Simulirati rad bankarskog sistema tokom vršnog opterećenja (npr. kraj meseca, isplata plata) kada veliki broj korisnika istovremeno koristi aplikaciju za pregled stanja, prenose sredstava i uplate računa.

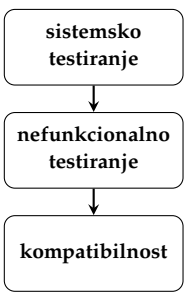
Uslovi testa:

- 10.000 simultanih korisničkih sesija.
- Svaka sesija izvršava niz standardnih operacija: prijava, pregled stanja, prenos novca, uvid u istoriju transakcija.
- Korišćenje stvarnog produkcionog okruženja ili njegove precizne simulacije.
- Baza podataka inicijalizovana sa velikim brojem korisnika (npr. 1.000.000 korisničkih naloga).

Metrike koje se prate:

- Prosečno i maksimalno vreme odziva za ključne operacije.
- Iskorišćenost resursa (CPU, memorija, baza podataka).
- Broj uspešno izvršenih zahteva u sekundi.
- Broj neuspešnih ili odbijenih zahteva (zbog grešaka, prekoračenja vremena ili kapaciteta).

Kriterijum prolaznosti:
Sistem mora podržati najmanje 8.000 istovremenih korisnika sa vremenom odziva manjim od 2 sekunde za sve ključne operacije. Maksimalni gubitak zahteva ne sme preći 0.05%.



Testovima kompatibilnosti proverava se način na koji sistem komunicira sa spoljnim komponentama i sistemima. Ovaj vid testiranja uključuje ispitivanje koegzistencije (mogućnosti rada zajedno sa drugim softverima na istom sistemu) i interoperabilnosti (sposobnosti razmene podataka i funkcionalne saradnje sa drugim sistemima). Cilj je da se osigura da sistem može uspešno da funkcioniše u širem okruženju bez konflikata ili nekompatibilnosti.

Primer 4.2.5 (Elektronska bankarska aplikacija — nastavak) Aplikacija mora da se izvršava stabilno i kada se istovremeno izvršavaju druge aplikacije.

Naziv test slučaja: TC-K009 — Koegzistencija sa antivirusom, VPN klijentom i upravljanjem lozinkama.

Opis: Proveriti da li elektronska bankarska aplikacija može da funkcioniše ispravno kada se izvršava paralelno sa drugim softverom instaliranim na korisnikovom uređaju, bez međusobnog negativnog

uticaja.

Scenario: Korisnik koristi elektronsku bankarsku aplikaciju dok su istovremeno aktivni drugi programi koji zahtevaju mrežnu komunikaciju i bezbednosne resurse, kao što su antivirus softver, VPN klijent i aplikacije za upravljanje lozinkama.

Okruženje testa:

- OS: *Windows 11*
- Aktivni softveri:
  - Antivirus: *Kaspersky Internet Security*
  - VPN: *Cisco VPN*
  - Password manager: *Bitwarden*
- Aplikacija pokrenuta u pregledaču *Chrome*

Uslovi testa:

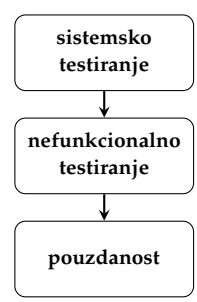
- Pokrenuti sve navedene programe istovremeno sa aplikacijom.
- Izvršiti osnovne funkcije u aplikaciji: prijava, pregled računa, slanje novca, pristup izvodima.
- Posmatrati eventualne konflikte: zamrzavanja, gubitak mrežne konekcije, bezbednosna upozorenja, itd.

Metrike koje se prate:

- Broj zabeleženih konflikata između aplikacije i drugih programa
- Smanjenje performansi aplikacije u uslovima koegzistencije
- Broj korisničkih funkcionalnosti koje su otežano dostupne ili onemogućene

Kriterijum prolaznosti:
Bankarska aplikacija mora biti funkcionalna i stabilna tokom istovremenog rada sa drugim programima. Ne sme doći do prekida funkcionalnosti, gubitka podataka, ili problema sa bezbednosnim komponentama sistema. Maksimalno dozvoljen broj manjih vizuelnih ili funkcionalnih anomalija: 1.

Testovima pouzdanosti proverava se stabilnost softvera i sposobnost da funkcioniše bez grešaka tokom određenog vremenskog perioda, u realnim ili simuliranim uslovima rada. Na primer, proverava se da li softver može da radi bez prekida i padova tokom dužeg vre-



menskog korišćenja (stabilnost u radu, zrelost i dostupnost), ispituje se sposobnost sistema da se oporavi nakon što dođe do greške, bilo automatski ili uz minimalnu intervenciju korisnika (oporavak od grešaka), softver se izvršava neprekidno tokom više sati ili dana (dugotrajno testiranje) da bi se otkrile skrivene greške (npr. curenje memorije, degradacija performansi) i proverava se otpornost na promene okruženja, tj. kako sistem reaguje na promene u okruženju (npr. prekid mreže, promena konfiguracije).

Primer 4.2.6 (Elektronska bankarska aplikacija — nastavak) Testovi pouzdanosti su važni za kritične sisteme kao što su bankarski, gde pouzdanost direktno utiče na poverenje korisnika i bezbednost podataka.

Naziv test slučaja: TC-P006 — Pouzdanost tokom 72h
Opis:
Proceniti sposobnost sistema da radi stabilno i bez prekida tokom produženog vremenskog perioda, u realnim uslovima korišćenja.

Scenario:
Izvršavanje aplikacije u trajanju od 72 sata bez prekida, tokom kojih se konstantno generišu korisnički zahtevi (prijave, pregledi računa, transferi sredstava, uplate računa), uz periodične simulacije grešaka i poremećaja u mrežnom okruženju.

Uslovi testa:

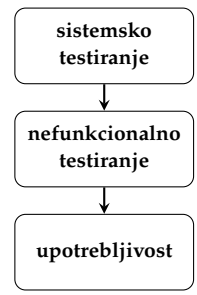
- Aplikacija se pokreće u testnom okruženju koje odgovara produkcionom.
- Koristi se alat za automatizaciju koji kontinuirano šalje realistične zahteve sa različitih virtuelnih korisničkih naloga.
- Na svakih 12 sati se simulira greška u komunikaciji sa bazom podataka, zatim oporavak.
- Prati se stabilnost sistema, ponašanje memorije i CPU-a, kao i sposobnost aplikacije da se automatski oporavi.

Metrike koje se prate:

- Broj sistemskih padova ili prekida rada.

- Vremensko trajanje oporavka posle greške.
- Prisustvo curenja memorije ili degradacije performansi.
- Očuvanost tačnosti i konzistentnosti podataka.

Kriterijum prolaznosti:
Tokom 72 sata kontinuiranog rada ne sme doći do nijednog sistemskog pada. Nakon svake simulirane greške, sistem mora da se oporavi u roku kraćem od 60 sekundi. Potrošnja memorije ne sme rasti progresivno bez oslobađanja (indikator curenja memorije). Podaci moraju biti konzistentni i odgovarati obavljenim transakcijama.



Testovima upotrebljivosti proverava se koliko je softverski sistem lak za upotrebu krajnjim korisnicima. Ova vrsta testiranja je posebno važna za aplikacije sa korisničkim interfejsom, kao što su mobilne i veb aplikacije, jer ima direktan uticaj na korisničko iskustvo i prihvatanje proizvoda. U testovima upotrebljivosti proverava se koliko brzo novi korisnik može da nauči da koristi sistem bez pomoći ili obuke (naučljivost), da li korisnik može da izvrši zadatke u prihvatljivom broju koraka i u razumnom vremenskom okviru (efikasnost korišćenja), da li sistem omogućava korisniku da lako prepozna i ispravi greške (prevencija i oporavak od grešaka), pristupačnost da li su elementi korisničkog interfejsa intuitivno raspoređeni i da li korisnik lako razume njihovu svrhu (jasnoća interfejsa), da li korisnici subjektivno ocenjuju sistem kao prijatan za korišćenje (zadovoljstvo korisnika), da li je sistem upotrebljiv za osobe sa različitim oblicima invaliditeta (pristupačnost), da li je svrha aplikacije jasna i lako razumljiva (prepoznatljivost).
Testiranje se često sprovodi kroz posmatranje korisnika dok izvršavaju tipične zadatke, uz praćenje vremena, broja grešaka i nivoa frustracije. Cilj je da se identifikuju prepreke koje bi korisniku otežale ili onemogućile korišćenje sistema. Takođe, često se koriste upitnici iintervjui nakon testiranja kako bi se saznao subjektivni utisak korisnika.

Primer 4.2.7 (Elektronska bankarska aplikacija — nastavak) Test naučljivosti je važan u ranim fazama dizajna korisničkog interfejsa i često se koristi za

usmeravanje iterativnih poboljšanja.

Naziv test slučaja: TC-N010 — Provera naučljivosti prenosa novca sa jednog na drugi tekući račun.

Opis: Proveriti koliko brzo novi korisnik, bez prethodnog iskustva sa aplikacijom, može samostalno da nauči kako da izvrši osnovnu funkcionalnost – prenos sredstava.

Scenario: Učesniku (koji ranije nije koristio aplikaciju) se daje pametni telefon sa instaliranom verzijom aplikacije i osnovnim uputstvom za početak (npr. kako da se prijavi). Zadatak je sledeći:

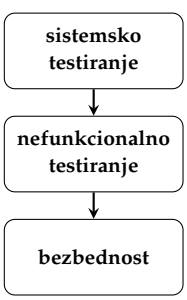
> „Prenesite 1000 RSD sa tekućeg računa na drugi račun koristeći opciju za internu transakciju.“

Parametri koji se prate:

- Vreme potrebno da korisnik pronađe opciju za prenos sredstava.
- Broj pokušaja i grešaka pre nego što se uspešno izvrši transakcija.
- Broj pitanja koje korisnik postavi (ako je dozvoljena asistencija).
- Subjektivna ocena korisnika (npr. ocena složenosti na skali od 1 do 5).

Kriterijum prolaznosti: Korisnik bi trebalo da uspešno izvrši transakciju bez pomoći u roku kraćem od 5 minuta, sa najviše jednom greškom (npr. pogrešan izbor opcije ili unos iznosa u pogrešno polje).

Napomena: Test se ponavlja sa 30 korisnika kako bi se dobila reprezentativna metrika. Ukoliko većina ispitanika ima poteškoće sa istim delovima interfejsa, to je jasan indikator za poboljšanje dizajna.



Testovima bezbednosti proverava se zaštita ljudi, opreme i okruženja od neželjenih efekata korišćenja softvera, posebno u kritičnim sistemima kao što su medicinski uređaji, automobilski sistemi, vazduhoplovstvo, industrijska automatizacija i slični domeni gde greška može imati ozbiljne posledice po bezbednost.

Primer 4.2.8 (Autonomna vožnja) Testovi bezbednosti su posebno važni u kontekstu autonomne vožnje.

Naziv test slučaja: TC-S007 — Automatsko zaustavljanje prilikom iznenadne prepreke na putu.

Opis: Proveriti kako autonomni sistem reaguje u slučaju iznenadne prepreke na putu, sa ciljem zaštite putnika i drugih učesnika u saobraćaju.

Scenario: Tokom vožnje pri brzini od 50 km/h, pešak iznenada ulazi na put na udaljenosti od 15 metara ispred vozila. Sistem za autonomnu vožnju mora da detektuje pešaka i aktivira bezbednosne mehanizme: kočenje, izbegavanje sudara i/ili upozorenje putnicima.

Uslovi testa:

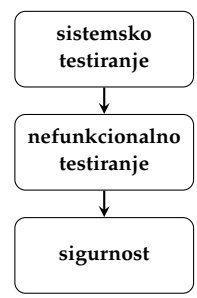
- Vozilo je u režimu potpunog autonomnog upravljanja.
- Test se izvodi u zatvorenom kontrolisanom okruženju sa korišćenjem lutke kao simulacije pešaka.
- Sistem ima pristup svim funkcionalnim senzorima.

Parametri koji se prate:

- Vreme reakcije sistema nakon detekcije prepreke.
- Efikasnost kočenja (da li je došlo do zaustavljanja pre prepreke).
- Aktivacija dodatnih sistema (upozorenja, sigurnosni pojasevi, naglo kočenje).
- Usklađenost sa bezbednosnim standardima.

Kriterijum prolaznosti:
Vozilo mora zaustaviti pre nego što dođe do kontakta sa preprekom u 98% slučajeva pri definisanoj brzini i udaljenosti, uz toleranciju za nepredviđene faktore kao što su vremenski uslovi ili osvetljenje.

Testovima sigurnosti proverava se da li su određene funkcionalnosti dostupne isključivo onim korisnicima kojima su namenjene. Proveravaju se dostupnost, integritet i poverljivost svih skupova podataka.



Primer 4.2.9 (Elektronska bankarska aplikacija — nastavak) Sigurnost bankarskih aplikacija i zaštita od neautorizovanih pristupa je izuzetno važna i zato

postoji posebna kategorija testiranja koja se naziva penetraciono testiranje (eng. *penetration testing*) koja pomaže u identifikaciji ranjivosti aplikacije na pokušaje neautorizovanog pristupa.

Naziv test slučaja: TC-SC004 — Neautorizovani pristupi grubom silom.

Opis: Proveriti da li je aplikacija otporna na pokušaj neautorizovanog pristupa putem napada grubom silom (eng. *brute force attack*) na formu za logovanje.

Scenario: Napadač pokušava da pristupi korisničkom nalogu automatskim slanjem velikog broja kombinacija korisničkog imena i lozinke putem skripte.

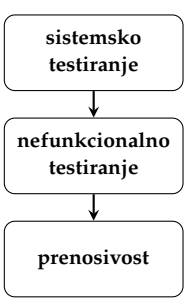
Uslovi testa:

- Test se izvodi u testnom okruženju uz nadzor administratora bez pristupa stvarnim korisničkim podacima.
- Koristi se alat za automatizovani napad (npr. alat *Hydra*).
- Postoji korisnički nalog sa poznatim korisničkim imenom i slabom lozinkom radi simulacije.

Parametri koji se prate:

- Maksimalan broj dozvoljenih pokušaja unosa lozinke pre zaključavanja naloga.
- Vreme blokade naloga i mehanizam obaveštavanja korisnika.
- Evidentiranje pokušaja napada u log fajlovima i reakcija sistema.
- Prisustvo mehanizama *CAPTCHA* nakon prvog neuspešnog pokušaja.

Kriterijum prolaznosti:
Aplikacija mora da blokira pristup korisničkom nalogu nakon najviše 3 neuspešna pokušaja i mora da onemogući dalju automatizovanu verifikaciju lozinki. Napad mora biti zabeležen u sistemskim logovima sa pripadajućim podacima (IP adresa, vreme napada, broj pokušaja).



Testovima prenosivosti proverava se sposobnost softverskog sistema da funkcioniše u različitim okruženjima. Glavni cilj ovih testova je da se potvrdi da softver može lako da se instalira, koristi i radi ispravno na više platformi, operativnih sistema, hardverskih konfiguracija ili pre-

gledača (u slučaju veb-aplikacija), kao i da može da se u potpunosti ukloni sa sistema bez ostavljanja tragova ili neželjenih podataka.

Primer 4.2.10 (Elektronska bankarska aplikacija — nastavak) Test prenosivosti je posebno značajan za bankarske aplikacije jer se koristi na velikom broju različitih uređaja, a doslednost i pouzdanost korisničkog iskustva direktno utiču na poverenje korisnika.

Naziv test slučaja: TC-P003 — Prilagodljivost različitim uređajima i operativnim sistemima.

Opis: Proveriti da li mobilna bankarska aplikacija funkcioniše dosledno na različitim operativnim sistemima i verzijama uređaja, bez potrebe za izmenom izvornog koda.

Scenario: Mobilna aplikacija razvijena je za platforme *Android* i *iOS*. Testira se njeno ponašanje na više verzija operativnih sistema i različitim uređajima:

- *Android* 10, 11 i 13 (*Samsung Galaxy*, *Xiaomi*, *Pixel*)
- *iOS* 15 i 17 (*iPhone* 11, *iPhone* 14)

Uslovi testa:

- Instalacija najnovije verzije aplikacije sa zvanične prodavnice (Google Play / App Store).
- Test uređaji resetovani na fabrička podešavanja (čisto okruženje).
- Stabilna internet konekcija (Wi-Fi).

Aktivnosti testa:

- Instalacija i pokretanje aplikacije na svakom test uređaju.
- Prijavljivanje korisnika sa validnim kredencijalima.
- Provera prikaza početnog ekrana, menija i funkcionalnosti (pregled stanja, prenos sredstava).
- Upoređivanje vizuelne konzistentnosti interfejsa.
- Provera lokalizacije (prikaz na srpskom i engleskom jeziku).

Kriterijum prolaznosti: Aplikacija mora da mogući korisnicima da nesmetano obave sve osnovne ope-

racije na svim platformama. Ne sme doći do pada sistema, vizuelnih nepravilnosti ili funkcionalnih razlika među verzijama.

Istraživačko testiranje

Jedna posebna forma sistemskog testiranja jeste istraživačko testiranje (eng. *exploratory testing*). Ova metoda testiranja oslanja se na znanje, intuiciju i kreativnost testera, pri čemu se testiranje ne zasniva isključivo na prethodno definisanim test slučajevima, već se test slučajevi osmišljavaju i izvršavaju u hodu.

Istraživačko testiranje podrazumeva otkrivanje neočekivanih pravaca korišćenja softverskog sistema, kao i prepoznavanje potencijalnih problema koji nisu bili identifikovani tokom analize i dizajna testova. Tokom ove aktivnosti tester intuitivno istražuje aplikaciju, posmatra njeno ponašanje u različitim situacijama i na osnovu uočenih obrazaca formuliše nove test slučajeve u realnom vremenu.

Ova vrsta testiranja ima najveći značaj kada je softver već funkcionalno zaokružen, odnosno kada je aplikacija dostupna u svom gotovo finalnom obliku. Tada tester ima mogućnost da uoči alternativne tokove korišćenja sistema koji nisu prethodno uzeti u obzir, čime se značajno povećava pokrivenost testiranjem.

Ukoliko se istraživačko testiranje zanemari, postoji rizik da određene funkcionalnosti ostanu neproverene, naročito one koje se ne nalaze u primarnim ili očekivanim tokovima rada. Stoga se ova vrsta testiranja često koristi kao dopuna formalnim tehnikama, sa ciljem da se dodatno osigura kvalitet konačnog proizvoda. U okviru istraživačkog testiranja mogu se proveravati i funkcionalna i nefunkcionalna svojstva sistema.

Primer 4.2.11 (Aplikacija za deljenje fotografija) Tester istražuje neočekivane tokove korišćenja u aplikaciji koja omogućava korisnicima da postavljaju, komentarišu i dele fotografije. Testiranje se sprovodi bez prethodno definisanih test slučajeva, oslanjajući se na intuiciju, iskustvo i zapažanja testera. Na primer, tester istražuje naredne scenarije upotrebe

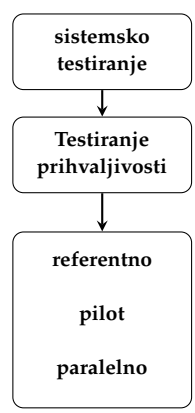
- Pokušaj postavljanja slike u formatu koji aplikacija formalno ne podržava (npr. format `.tiff`) — proverava se reakcija sistema.
- Brzo višestruko postavljanje iste fotografije — provera se da li dolazi do duplikata, grešaka ili zamrzavanja aplikacije.
- Istovremeno korišćenje više funkcionalnosti (npr. slanje komentara dok je slika u procesu postavljanja) — provera se stabilnost aplikacije i ispravnost rada.
- Pokušaj učitavanja sadržaja bez dostupnog interneta, ili sa slabo propusnim internetom — provera se prisustvo adekvatne poruke o grešci.
- Promena jezičkih podešavanja tokom aktivne sesije — provera se uticaj na postojeći sadržaj korisničkog interfejsa.

U svakom od prethodnih situacija aplikacija bi trebalo da se ponaša stabilno, obezbedi korisniku jasne poruke o greškama i izbegne pad sistema ili gubitak podataka. Cilj ovih provera je otkrivanje potencijalnih grešaka i nepredviđenih ponašanja sistema u realnim, kompleksnim i neobičnim scenarijima koje standardni test slučajevi možda ne pokrivaju.

Testiranje prihvatljivosti

Testovi prihvatljivosti (eng. *acceptance testing*) treba da omoguće klijentima i korisnicima da se sami uvere da je napravljeni softver u skladu sa njihovim potrebama i očekivanjima. Ovu vrstu testiranja izvode i procenjuju korisnici, a razvojni tim im pruža pomoć oko tehničkih pitanja, ukoliko za tim ima potrebe. Testiranje prihvatljivosti obično spada u tehnike validacije softvera.

Klijent može da proceni sistem na tri načina: **referentnim** testiranjem, **pilot** testiranjem i **paralelnim** testiranjem.



Referentno testiranje izvode korisnici kako bi procenili da li je softver implementiran u skladu sa očekivanjima. Kod referentnog testiranja, klijent generiše test slučajeve koji predstavljaju uobičajne uslove u kojima sistem treba da radi.

Pilot testiranje podrazumeva instalaciju sistema na privremenoj lokaciji i njegovu upotrebu. U ovom slučaju, testiranje se vrši simulacijom svakodnevnog rada na sistemu.

Paralelno testiranje se koristi tokom razvoja, kada jedna verzija softvera zamenjuje drugu ili kada novi sistem treba da zameni stari. Ideja je paralelno funkcionisanje oba sistema (starog i novog) čime se korisnici postepeno privikavaju i prelaze na korišćenje novog sistema.

Instalaciono testiranje

Instalaciono testiranje predstavlja specifičnu vrstu testiranja u kojoj se softver instalira u klijentskom okruženju, kako bi se proverila njegova sposobnost da se pravilno instalira i funkcioniše u realnim uslovima. Tokom ovog procesa, sistem se podešava u skladu sa tehničkim karakteristikama ciljne mašine, kao i sa specifičnostima okruženja (npr. operativni sistem, mrežne postavke, povezani uređaji). Ukoliko softver zahteva komunikaciju sa spoljnim uređajima ili servisima, testira se i sposobnost uspostavljanja te komunikacije.

Instalacioni testovi se najčešće sprovode u saradnji sa krajnjim korisnicima, kako bi se osiguralo da instalacija teče bez grešaka i da sistem nakon instalacije ispravno funkcioniše. Poseban akcenat stavlja se na detekciju eventualnih uticaja okruženja na funkcionalne i nefunkcionalne karakteristike sistema, kao što su performanse, sigurnost ili kompatibilnost. Ovo testiranje ima za cilj da potvrdi da je softver spreman za rad u stvarnom okruženju korisnika, bez potrebe za dodatnim intervencijama nakon instalacije.

4.3 Tehnike testiranja

Tehnike testiranja imaju za cilj da pruže sistematski odgovor na pitanje kako identifikovati reprezentativni skup test slučajeva (*test primera*, ili samo *testova*). Reprezentativni skup testova treba da obuhvati slučajeve koji najbolje odražavaju stvarne uslove rada softverskog sistema, ali i one koji imaju povećanu verovatnoću da otkriju potencijalne slabosti u implementaciji. Dodatno, treba da omogući efikasno balansiranje između obima testiranja i potrošnje resursa. Reprezentativni skup testova treba da ima naredne karakteristike:

- Visok potencijal otkrivanja grešaka.
- Relativno mala veličina.
- Relativno velika brzina izvršavanja.
- Pruža visok stepen poverenja u pouzdanost softvera.

Dobro definisan i pažljivo odabran skup testova predstavlja osnovu za kvalitetno, ciljno-orijentisano testiranje koje doprinosi visokom kvalitetu softverskog rešenja.

4.3.1 Pokrivenost testiranjem

Pokrivenost testiranjem (eng. *test coverage*) je metrika koja pomaže da se proceni koliko su testovi sveobuhvatni i koliko dobro proveravaju funkcionalnost, strukturu i ponašanje sistema. Koristi se za procenu kvaliteta softvera. Takođe, može se koristiti i da ukaže na delove sistema koji nisu testirani. Pomaže u donošenju odluke da li je softver spreman za isporuku.

Pokrivenost se definiše kao procenat elemenata sistema koji su obuhvaćeni testiranjem u odnosu na sve elemente te vrste. Postoje razne vrste pokrivenosti, a osnovne vrste pokrivenosti su date u nastavku.

$$\textbf{Pokrivenost} = \frac{\text{Broj testiranih elemenata}}{\text{Ukupan broj elemenata}} \times 100\%$$

$$\textbf{Pokrivenost zahteva} = \frac{\text{Broj testiranih zahteva}}{\text{Ukupan broj zahteva}} \times 100\%$$

$$\textbf{Pokrivenost funkcionalnosti} = \frac{\text{Broj testiranih funkc.}}{\text{Ukupan broj funkc.}} \times 100\%$$

$$\textbf{Pokrivenost koda} = \frac{\text{Broj izvršenih elem. koda}}{\text{Ukupan broj tih elem.}} \times 100\%$$

Pokrivenost zahteva (eng. *requirements coverage*) Mera koja pokazuje koliki je procenat zahteva proveren testiranjem u odnosu na ukupan broj zahteva korisnika koji su definisani kroz specifikaciju. Potpuna pokrivenost podrazumeva da je svaki zahtev testiran.

Pokrivenost funkcionalnosti (eng. *functional coverage*) Mera koja pokazuje koliki je procenat funkcionalnosti proveren testiranjem u odnosu na ukupan broj funkcionalnosti sistema. Potpuna pokrivenost podrazumeva da je svaki funkcionalnost testirna.

Pokrivenost koda (eng. *code coverage*) Ova mera sadrži u sebi veći broj različitih pokrivenosti. Na primer, pokrivenost naredbi se definiše kao broj izvršenih naredbi u okviru testiranja podeljen sa ukupnim brojem naredbi u projektu. Pokrivenost koda se detaljno razmatra u delu 4.3.4.

Primer 4.3.1 (Veb aplikacija za kupovinu) Visoka pokrivenost zahteva ne znači nužno i visoku pokrivenost funkcionalnosti — mogu postojati funkcionalnosti koje nisu formalno dokumentovane zeahtevima, a koje ipak treba testirati. Zato se u praksi često prate obe pokrivenosti.

Na primer, za veb aplikaciju za kupovinu, u okviru specifikacije mogu biti zadati naredni zahtevi:

- Korisnik može dodati proizvod u korpu.
- Korisnik može obrisati proizvod iz korpe.
- Korisnik može završiti kupovinu klikom na dugme *Kupi*.

U okviru implementacije, uz te zadatke, programer implementira i sledeće

- Kada je korpa prazna, dugme *Kupi* se automatski onemogućava.
- Ako korisnik pokuša da doda negativan broj proizvoda, sistem prikazuje poruku o grešci.

Pokrivenost zahteva u ovom slučaju ne obuhvata pokrivenost funkcionalnosti jer nedostaju testovi implementiranih funkcionalnosti za onemogućavanje dugmeta *Kupi* kada je korpa prazna i za prikazivanje poruke o grešci prilikom unosa negativnog broja proizvoda.

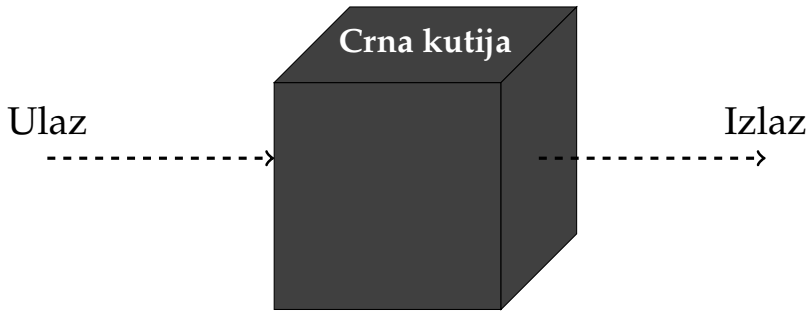
4.3.2 Podela tehnika testiranja

Većina softverskih sistema podrazumeva mogućnost relativno jasnog utvrđivanja očekivanog izlaza za zadate uslove. U takvim slučajevima, dobri test primeri se mogu naći na osnovu specifikacije programa (testiranje crne kutije), na osnovu koda programa (testiranje bele kutije) ili na osnovu kombinacije specifikacije i koda programa (testiranje sive kutije).

Odnos: Testiranje crne kutije je testiranje iz ugla korisnika dok je testiranje bele kutije testiranje iz ugla programera.

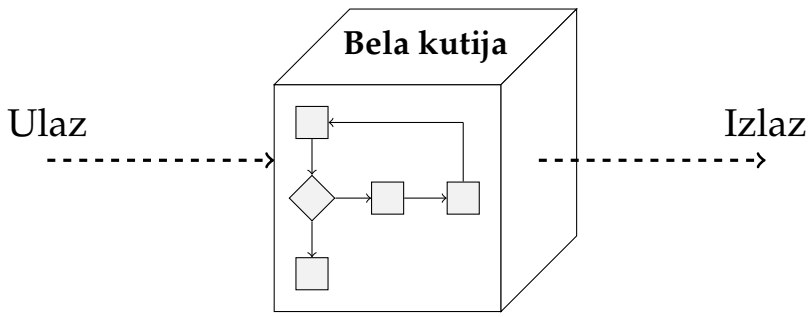
Testiranje crne kutije (eng. *black box testing*) — generisanje test primera bez razmatranja interne strukture koda već isključivo na osnovu specifikacije. Ovakav način testiranja se fokusira na ponašanje sistema, posmatrano iz korisničkog ugla. Drugi nazivi su i funkcionalno testiranje (eng. *functional testing*), testiranje ponašanja (eng. *behavioural testing*), testiranje vođeno podacima (eng. *data driven testing*). Prednost ovog pristupa je mogućnost potpunog razdvajanja programera i testera i zato ovo testiranje obično obavljaju testeri. Zadatak testera je da sistemu pruži ulaze, a zatim da proveri izlaze u odnosu na datu specifikaciju.

Testiranje bele kutije (eng. *white box testing*) — generisanje test primera na osnovu interne strukture i logike koda. Alternativni nazivi za ovaj pristup su strukturno testiranje (eng. *structural testing*) i testiranje vođeno logikom (eng. *logic driven testing*), što dodatno naglašava oslonac



Slika 4.6: Testiranje crne kutije: fokus na ulazu i izlazu

na logičku strukturu koda i neophodnost poznavanja implementacije radi pisanja testova za ovu vrstu testiranja. Zbog toga testiranje bele kutije obično obavljaju programeri tokom faze razvoja softvera. Primer testiranja bele kutije su testovi jedinica koda u kojima se proverava ispravnost najmanjih funkcionalnih delova softvera.

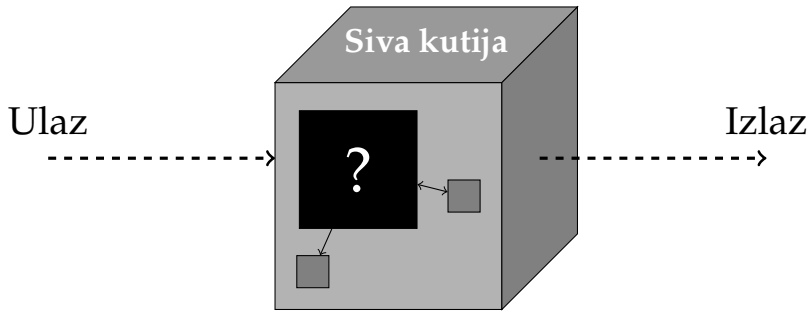


Slika 4.7: Testiranje bele kutije: fokus na strukturi koda.

Testiranje sive kutije (eng. *gray box testing*) — predstavlja prelaz između tehnika crne i bele kutije (mešovita strategija). Kod ove tehnike postoji uvid u unutrašnju strukturu sistema, ali ne u toj meri kao kod tehnika bele kutije. Koristiti se, na primer, kod komponentnog i integracionog testiranja. Ovo je tehnika koju koriste i programeri i testeri.

Problem proročišta

Prethodne tehnike testiranja podrazumevaju da je za ulazne vrednosti moguće jednostavno odrediti odgovarajuće izlazne vrednosti. Međutim, to ne mora da bude slučaj za sve



Slika 4.8: Testiranje sive kutije: mešovita strategija.

softverske sisteme. Preciznije, proročište (eng. *test oracle*) je mehanizam (ili znanje) koje omogućava testeru da odredi: „Očekivani rezultat za ulaz *U* je izlaz *I*“ i da uporedi stvarni izlaz sa tim očekivanim. Kada takav mehanizam ne postoji ili ga je teško definisati, dolazi do problema proročišta. Problem proročišta (eng. *oracle problem*) u testiranju softvera odnosi se na izazov određivanja da li je rezultat testa ispravan — tj. kako sa sigurnošću znati da je izlaz sistema za određeni ulaz tačan. Problem proročišta je izražen u oblastima kao što su računarska grafika, konstrukcija kompilatora i mašinsko učenje.

Primer 4.3.2 (C kompajler) U razvoju kompajlera za programski jezik C potrebno je da proverimo da li kompajler ispravno prevodi programe. Test bi trebalo da proveri da li je kompajler:

- ispravno generisao mašinski kôd koji odgovara očekivanom izvršavanju programa (zadržao je semantiku izvornog programa),
- ispravno optimizovao kôd (npr. eliminisao suvišne promenljive).

Međutim

- Najčešće ne postoji formalna specifikacija očekivanog izlaza za proizvoljan ulazni program.
- Ručna provera mašinskog koda ili ponašanja može biti izuzetno teška i podložna greškama.
- Automatizovana provera semantičke ekvivalencije dva programa (izvorni i prevedeni) je u opštem slučaju nerešiv problem.

Problem proročišta se može rešavati na različite načine.

- Korišćenje alternativnih implementacija kao uporednog standarda ili upotreba ranijih verzija softvera kao referentnog ponašanja (onda kada su ranije verzije softvera dostupne). Ukoliko su rezultati različiti, onda bar jedna od implementacija ima grešku.
- Metamorfno testiranje — ne proverava se konkretan izlaz, već odnos između više izlaza koji se računa na osnovu odnosa između ulaza i osobina algoritma koji se testira.

Primer 4.3.3 (C kompajler — nastavak) Jedan praktičan pristup prevazilaženju problema proročišta kod kompajlera je tzv. diferencijalno testiranje:

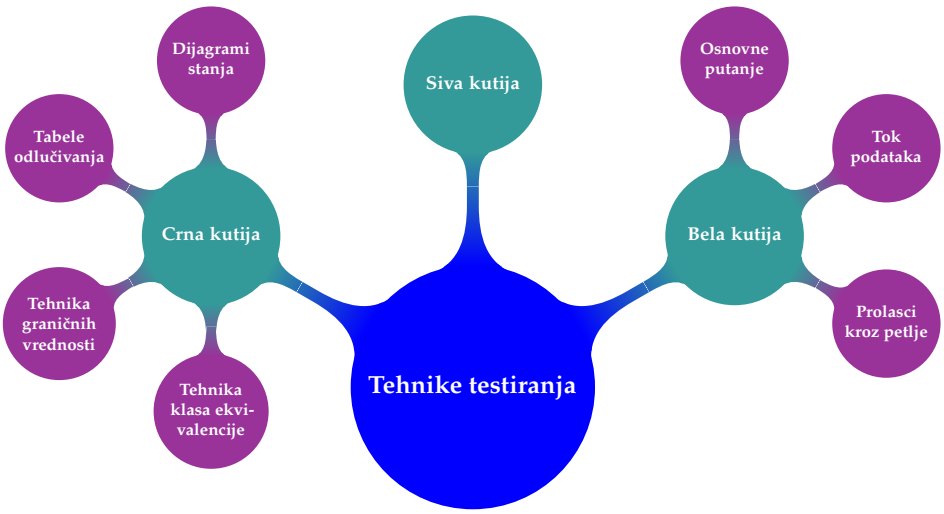
- Izvorni kôd se prevede pomoću više različitih kompajlera (ili različitih verzija istog kompajlera),
- Izvrše se sve dobijene verzije izvršivog koda u kontrolisanom okruženju,
- Ako svi rezultati izvršavanja daju isti izlaz, pretpostavlja se da su tačni.

Jasno je da se čak i tada ne može sa sigurnošću tvrditi da je rezultat ispravan, iz više razloga. Najpre, provera je urađena samo na nekim konkretnim ulazima i za te ulaze je utvrđeno da se različiti kompajleri isto ponašaju. Za neke druge ulaze možda bismo dobili razliku u ponašanju. Dodatno, moguće je i da svi kompajleri koji učestvuju u poređenju daju istu grešku.

Korišćenje različitih implementacija se ne može uvek primeniti pošto često ne postoji više implementacija istog algoritma (jer su te implementacije previše skupe i zahtevaju puno vremena). Takođe, ako se različite implementacije kreiraju od strane istih ljudi, moguće je da oni prave iste greške.

4.3.3 Testiranje crne kutije

Testiranje crne kutije teorijski se može uraditi isprobavanjem svih mogućih ulaza (eng. *exhaustive input testing*). Međutim, već za trivijalne programe nije moguće koristiti ovu tehniku.



Slika 4.9: Tehnike testiranja softvera

Primer 4.3.4 Kvadratna jednačina

$$a \cdot x^2 + b \cdot x + c = 0$$

ima rešenje

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4 \cdot a \cdot c}}{2 \cdot a}$$

Ako su koeficijenti jednačine celobrojni i tipa `int32` onda je broj različitih test primera za potpuno testiranje

$$2^{32} \cdot 2^{32} \cdot 2^{32} = 2^{96}$$

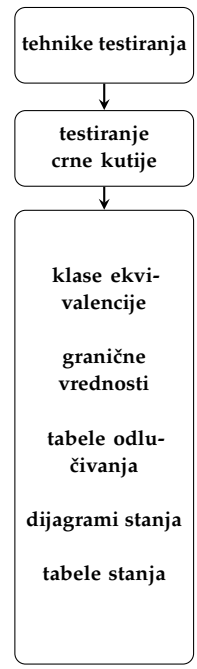
Dodatno, pored isprobavanja svih mogućih ispravnih vrednosti ulaza, potrebno je razmotriti i moguće neispravne ulaze kojih takođe može biti mnogo.

Primer 4.3.5 Neispravan ulaz za očekivanu starost osobe može da bude negativan broj ili unos neke proizvoljne reči umesto broja. Očekivano ponašanje softvera u takvim situacijama može da bude signaliziranje greške korisniku sa mogućnošću unosa nove vrednosti ili prekid rada softvera uz odgovarajuću poruku.

Cilj tehnika testiranja crne kutije je da pronađu prihvatljiv broj test slučajeva (tj. kombinacija ulaza) koji odgovara repre-

zentativnom skupu testova. Test slučajevi mogu se podeliti u dve osnovne kategorije: test slučajevi koji odgovaraju validnim ulazima i test slučajevi koji odgovaraju nevalidnim ulazima.

Primer 4.3.6 (Nastavak primera 4.3.4.) Test slučaj koji odgovara ispravnom ulazu za kvadratnu jednačinu može da sadrži ulazne vrednosti {1, -3, 2} i izlazne vrednosti {1, 2}. Test slučaj koji odgovara neispravnom ulazu može da sadržu ulazne vrednosti {"I", -3, 2} i izlaznu vrednost {"Neispravan unos prvog koeficijenta"}

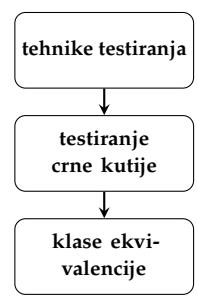


Pronalaženje reprezentativnog skupa testova metodama crne kutije se ostvaruje postavljanjem odgovarajućih pretpostavki o softveru koji treba da se testira. Najpoznatiji tehnike obuhvataju tehniku klasa ekvivalencije, tehniku graničnih vrednosti, tabele odlučivanja i dijagrame stanja.

Klase ekvivalencije

Testiranje pomoću klasa ekvivalencije (eng. *equivalence class testing*) je tehnika koja se koristi da smanji broj test slučajeva na prihvatljiv nivo, a da se pri tome zadrži zadovoljavajuća pokrivenost ulaznog prostora podataka. Klase ekvivalencije predstavljaju skupove ulaznih podataka koji se obrađuju na isti način od strane sistema. One se mogu podeliti na validne (ispravne) i nevalidne (neispravne) klase, u zavisnosti od toga da li bi sistem trebalo da prihvati te vrednosti za dalju obradu ili ih odbaci.

Primer 4.3.7 (Validne i nevalidne klase) Ukoliko se očekuje da korisnik unese broj godina između 18 i 65, validne klase bi pokrivale sve vrednosti unutar tog opsega, dok bi nevalidne klase uključivale vrednosti ispod 18 i iznad 65.



Ova tehnika se zasniva na pretpostavci da se sve vrednosti unutar jedne klase ponašaju identično u kontekstu testiranja, te je dovoljno izabrati po jedan predstavnik iz svake klase. Preciznije, pretpostavke za klase ekvivalencije su:

- Ukoliko jedan test slučaj iz određene klase ekvivalencije otkrije grešku, velika je verovatnoća da bi i svi ostali test slučajevi iz iste klase otkrili istu tu grešku.

- Ukoliko jedan test slučaj iz određene klase ekvivalencije ne otkrije grešku, pretpostavlja se da ni ostali test slučajevi iz te klase ne bi otkrili grešku.

Na osnovu ovih pretpostavki, testiranje se može racionalizovati izborom po jednog predstavnika iz svake klase. Ključni koraci primene ove tehnike su:

1. Identifikovati sve validne klase ekvivalencije za ulazne vrednosti koje bi sistem trebalo da prihvati za dalju obradu.
2. Izabrati po jedan test slučaj za svaku validnu klasu.
3. Identifikovati nevalidne klase ekvivalencije za ulazne vrednosti koje bi sistem trebalo da odbaci.
4. Izabrati po jedan test slučaj za svaku nevalidnu klasu.

Na taj način se smanjuje broj potrebnih testova, a da se pritom zadrži visok nivo pokrivenosti funkcionalnih pravila sistema i otkrivanja potencijalnih grešaka.

Primer 4.3.8 (Zapošljavanje na osnovu starosti) Pretpostavimo da sistem za zapošljavanje sprovodi pravila na osnovu starosti kandidata. Pravila su definisana narednom tabelom:

Godine	Pravilo
0–16	Ne zaposliti
16–18	Može se zaposliti samo sa pola radnog vremena
18–55	Može se zaposliti sa punim radnim vremenom
55–99	Ne zaposliti

Validne klase ekvivalencije:

- V1 — Godine u opsegu 0–16 (npr. 10)
- V2 — Godine u opsegu 16–18 (npr. 17)
- V3 — Godine u opsegu 18–55 (npr. 30)
- V4 — Godine u opsegu 55–99 (npr. 70)

Nevalidne klase ekvivalencije:

- I1 – Vrednost ispod minimalne vrednosti (npr. −5)
- I2 – Vrednost iznad maksimalne vrednosti (npr. 105)

Test slučajevi:

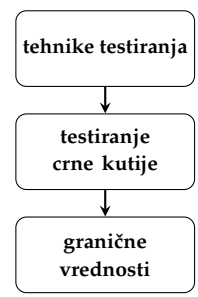
ID	Klasa	Ulaz	Očekivani rezultat
TC1	V1	10	Ne zaposliti
TC2	V2	17	Zapošljavanje sa pola radnog vremena
TC3	V3	30	Zapošljavanje sa punim radnim vremenom
TC4	V4	70	Ne zaposliti
TC5	I1	−5	Odbiti unos
TC6	I2	105	Odbiti unos

Korišćenjem tehnike klasa ekvivalencije, broj test slučajeva se smanjuje sa 100 (po jedan za svaku godinu) na svega nekoliko reprezentativnih testova, uz zadržavanje pokrivenosti svih funkcionalnih pravila sistema.

Napomena: Često je lakše identifikovati validne klase ekvivalencije od nevalidnih. Na primer, pored nevalidnih unosa -5 i 105, potrebno je testirati i nevalidne unose kao što je unos izraza koji vodi neispravnom broju (npr. 66+35), unos slova i nenumeričkih karaktera i slično.

Granične vrednosti

Testiranje pomoću klasa ekvivalencije predstavlja osnovnu tehniku za oblikovanje test slučajeva. Međutim, u praksi se pokazalo da se veliki broj grešaka javlja upravo na granicama definisanih klasa. Odatle potiče ideja o dodatnoj tehnici: testiranju graničnih vrednosti (eng. *boundary value testing*).



Ova tehnika fokusira se na vrednosti koje se nalaze na samim granicama, neposredno ispod i neposredno iznad granica klasa ekvivalencije, jer upravo na tim mestima programeri najčešće prave greške. Tipična greška je pogrešno upisivanje operatora nejednakosti, na primer korišćenje znaka $>$ umesto znaka $\geq$ (ili obratno).

Primetimo da su termini *ispod* i *iznad* relativni pojmovi koji zavise od tipa i preciznosti vrednosti koje se testiraju. Na primer, kod celobrojnih vrednosti, to su uzastopni brojevi ($n-1$ i $n+1$, ukoliko je n granica), dok kod realnih brojeva razlika može biti definisana brojem značajnih decimala. Dodatna složenost se javlja kada postoje višedimenzionalni ulazi (npr. dve povezane numeričke vrednosti) ili kada se radi sa realnim brojevima visoke preciznosti. U takvim slučajevima, identifikacija relevantnih granica i test tačaka zahteva dodatnu pažnju i detaljniju analizu.

Osnovni koraci za primenu testiranja graničnih vrednosti su sledeći:

1. Identifikovati klase ekvivalencije na osnovu ulaznih podataka.
2. Odrediti tačne granice svake identifikovane klase.
3. Za svaku granicu, definisati sledeće test tačke:
   - tačku na samoj granici,
   - tačku neposredno ispod granice,
   - tačku neposredno iznad granice.

Važno je imati u vidu da tačke koje se nalaze ispod i iznad granice mogu da pripadaju klasama ekvivalencije koje su već obuhvaćene testovima. Zbog toga treba paziti da se testovi ne dupliraju kada se kombinuje testiranje graničnih vrednosti sa testiranjem klasa ekvivalencije.

Primer 4.3.9 (Zapošljavanje na osnovu starosti — nastavak.) Prilikom analize graničnih vrednosti, uočava se problem u specifikaciji sistema na granicama klasa ekvivalencije. Naime, vrednosti *16*, *18* i *55* pojavljuju se u više klasa, što dovodi do preklapanja. Na primer, jedno pravilo navodi da osobe stare 16 godina ne treba zaposliti, dok drugo pravilo ističe da se takve osobe mogu zaposliti sa pola radnog vremena. Ovakvo preklapanje ukazuje na grešku u specifikaciji koju je neophodno ispraviti.

Ispravna verzija tabele, koja eliminiše ova preklapanja, prikazana je u nastavku:

Godine	Pravilo
0–15	Ne zaposliti
16–17	Može se zaposliti samo sa pola radnog vremena
18–54	Može se zaposliti sa punim radnim vremenom
55–99	Ne zaposliti

Na osnovu nje možemo da definišemo slične validne klase ekvivalencije i njihove predstavnike.

Validne klase ekvivalencije:

- V1 — Godine u opsegu 0–15 (npr. 10)
- V2 — Godine u opsegu 16–17 (npr. 17)

- V3 — Godine u opsegu 18–54 (npr. 30)
- V4 — Godine u opsegu 55–99 (npr. 70)

Dalje, na osnovu definisanih klasa ekvivalencije, možemo definisati odgovarajuće vrednosti na granicama:

- Granica 0: {0, 1}
- Granica 15: {14, 15, 16}
- Granica 16: {15, 16, 17}
- Granica 17: {16, 17, 18}
- Granica 18: {17, 18, 19}
- Granica 54: {53, 54, 55}
- Granica 55: {54, 55, 56}
- Granica 99: {98, 99, 100}

Unija svih ovih skupova daje sledeće test vrednosti:

{0, 1, 14, 15, 16, 17, 18, 19, 53, 54, 55, 56, 98, 99, 100}

Zajedno sa vrednostima identifikovanim za same klase ekvivalencije, to čini skup

{0, 1, 10, 14, 15, 16, 17, 18, 19, 30, 53, 54, 55, 56, 70, 98, 99, 100}

Međutim, sada imamo preklapanja unutar istih klasa ekvivalencije, i neke vrednosti mogu biti redundantne. Na primer, vrednosti 0, 1, 10 i 14 pripadaju istoj klasi (0–15), dok su 18, 19, 30 i 53 sve iz klase 18–54. Testove koji pripadaju istoj klasi ekvivalencije, a nisu granične vrednosti moguće je po potrebi ukloniti u cilju smanjivanja troškova testiranja, bez gubitka pokrivenosti.

Tabele odlučivanja

Tabela odlučivanja (eng. *decision table*) pruža pregled složenih poslovnih pravila u strukturisanom i lako čitljivom obliku. One predstavlja snažno sredstvo za analizu i dizajn test scenarija u složenim informacionim sistemima. Često se koriste u testiranju softvera za sistematsko generisanje test slučajeva, naročito u situacijama kada postoji veći broj kombinacija ulaznih uslova i očekivanih akcija.

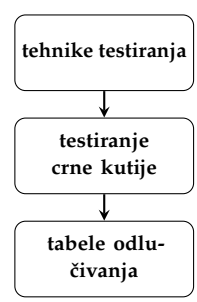


Tabela odlučivanja je organizovana u redove i kolone. Redovi se dele u dve grupe: prvu grupu čine uslovi nad ulazima, dok drugu grupu čine moguće akcije koje sistem treba da izvrši. Kolone predstavljaju pravila — svaka kolona odgovara jedinstvenoj kombinaciji vrednosti uslova i opisuje koje se akcije u tom slučaju preduzimaju.

Uslovi mogu biti binarni (na primer, *da/ne*, *istina/laž*) ili viševrednosni (na primer, *malo/srednje/veliko*). Kada su uslovi binarni, iz svakog pravila se obično izvodi jedan test slučaj. Kod viševrednosnih uslova može se izvesti više test slučajeva, u zavisnosti od željenog nivoa pokrivenosti.

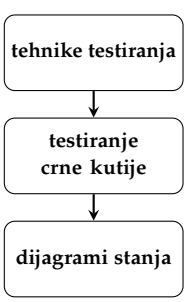
U slučajevima kada više pravila dovode do iste akcije, bez obzira na vrednost nekog uslova, moguće je izvršiti objedinjavanje pravila (eng. *table collapsing*). U takvim slučajevima, uslov koji ne utiče na ishod označava se simbolom - - (crtica) i naziva se nebitnim (eng. *don't care*).

Test slučajevi izvedeni iz tabele odlučivanja mogu se dodatno kombinovati sa drugim tehnikama testiranja, kao što su testiranje pomoću klasa ekvivalencije ili testiranje graničnih vrednosti. Ova kombinacija povećava preciznost i efektivnost testiranja.

Primer 4.3.10 (Banka) Klijent zahteva isplatu gotovine na bankomatu. Sistem treba da odluči da li će da odobri isplatu. Odlučivanje vrši pomoću podataka o sredstvima na računu i o dozvoljenom minusu. Način odlučivanja je prikazan narednom tabelom.

	Pravilo 1	Pravilo 2	Pravilo 3
Uslovi			
Dovoljno sredstava na računu	Da	Ne	Ne
Dozvoljen minus	—	Da	Ne
Akcije			
Isplata odobrena	Da	Da	Ne

Prvo pravilo je dobijeno spajanjem dva pravila: kada ima dovoljno sredstava na računu, nezavisno od toga da li je minus dozvoljen ili ne, isplata se odobrava. Iz drugog pravila može se izvesti test slučaj tako što se za ulaz uzme da korisnik nema dovoljno sredstava na računu i da mu je dozvoljen minus. Zatim se izlaz iz programa poredi sa očekivanom akcijom, a to je da je isplata odobrena. Iz trećeg pravila može se izvesti test slučaj za koji isplata nije odobrena.



Dijagrami stanja

Sistemi koji reaguju na spoljašnje događaje i čije ponašanje zavisi od prethodno izvršenih akcija mogu se efikasno modelovati pomoću konačnih automata (eng. *finite state machines*). U svakom trenutku, takav sistem se nalazi u jednom od

konačno mnogo mogućih stanja i pasivno čeka na neki ulazni događaj. Kada se dogodi odgovarajući događaj, sistem, u zavisnosti od trenutnog stanja, prelazi u novo stanje. Tokom ovog prelaza često se izvršava i određena akcija, kao što je generisanje izlaza, slanje poruke ili promena internog podatka.

Jedan od najvažnijih načina prikaza konačnog automata u inženjerskoj praksi je dijagram stanja (eng. *state-transition diagram*). Ovaj dijagram kompaktno i pregledno opisuje složene zahteve sistema i njegov način interakcije sa spoljašnjim svetom. Posebno se koristi za modelovanje sistema čije ponašanje zavisi od sekvence prethodnih događaja.

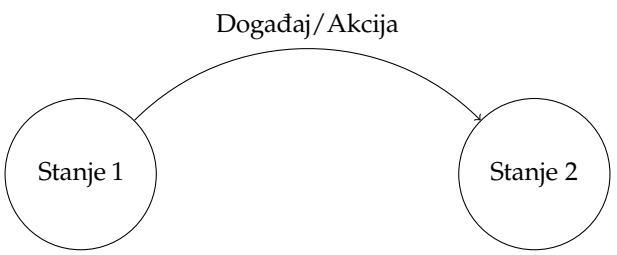
Osnovni elementi dijagrama stanja prikazani su na slici 4.10 i uključuju:

Stanje — Predstavlja određeno ponašanje ili konfiguraciju sistema; čuva informaciju o prošlim događajima i određuje reakciju na buduće.

Prelaz — Predstavlja promenu iz jednog stanja u drugo, iniciranu događajem.

Događaj — Spoljašnji ili unutrašnji signal koji izaziva prelaz između stanja.

Akcija — Operacija koju sistem izvršava kao odgovor na prelaz, kao što su promene izlaza, logovanje ili pozivi metoda.



Slika 4.10: Prikaz prelaza u okviru dijagrama stanja

Dijagram stanja omogućava vizuelizaciju svih mogućih stanja sistema, događaja koji uzrokuju promene stanja, kao i pratećih akcija. Ova tehnika je široko rasprostranjena u razvoju softverskih komponenti koje zahtevaju precizno definisano ponašanje u skladu sa spoljašnjim stimulusima, kao što su korisnički interfejsi, komunikacioni protokoli i kontrolni sistemi.

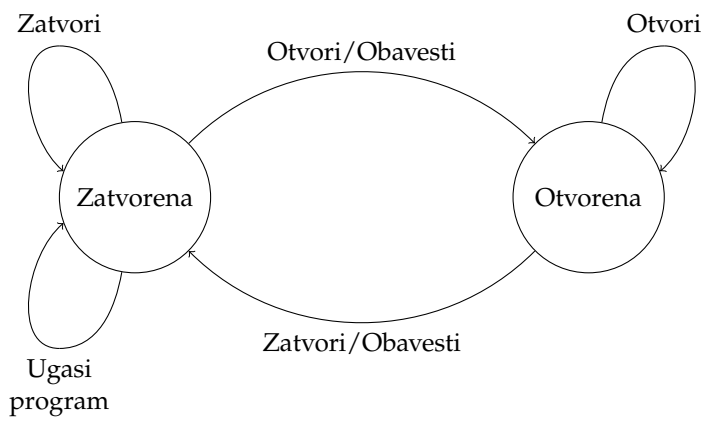
Pored modelovanja ponašanja, dijagram stanja može se koristiti i za generisanje test slučajeva. Budući da je dijagram

stanja oblik usmerenog grafa, testiranje se može zasnivati na njegovom obilasku. Na primer, možemo zahtevati da svaki prelaz u dijagramu bude ispitan bar jednom — što predstavlja dobar kompromis između pokrivenosti i obima testova. Alternativno, može se zahtevati pokrivenost svih stanja ili čak svih mogućih putanja kroz dijagram, u zavisnosti od ciljeva i kritičnosti sistema.

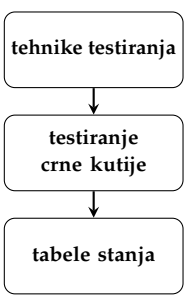
Primer 4.3.11 (Ugrađena kasa) Posmatramo softverski sistem za maloprodaju koji ima ugrađenu opciju za otvaranje i zatvaranje (fioke) kase prikazan na slici 4.11. Primeri test slučajeva

1. Otvori, zatvori, ugasi program
2. Otvori, otvori, zatvori, zatvori, ugasi program.

Drugi test slučaj aktivira svaki prelaz datog dijagrama bar jednom. Pri izvršavanju navedenih naredbi proverava se da li sistem reaguje u skladu sa zadatim zahtevima.



Slika 4.11: Dijagram stanja za ugrađenu kasu



Tabele stanja

Konačni automat koji modeluje sistem se može prikazati i tabelama stanja (eng. *state transition tables*). Osnovna prednost tabela stanja jeste njihov sistematični pristup, prikazuju sve moguće kombinacije stanja i događaja. Takvim pristupom mogu da se uoče situacije u kojima ponašanje sistema nije definisano, što može da spreči pojavu grešaka. Kod tabela stanja, iz svakog reda se može direktno izvesti jedan test

slučaj. Tabele stanja postaju nepraktične ukoliko postoji veliki broj mogućih stanja i događaja.

Primer 4.3.12 (Ugrađena kasa — nastavak.) Tabela stanja za primer 4.3.11 prikazana je u narednoj tabeli.

Trenutno stanje	Događaj	Akcija	Naredno stanje
Zatvorena	otvori	obavesti	Otvorena
Zatvorena	zatvori	-	Zatvorena
Zatvorena	ugasi program	-	Zatvorena
Otvorena	otvori	-	Otvorena
Otvorena	zatvori	obavesti	Zatvorena
Otvorena	ugasi program	-	Nedefinisano

Na osnovu tabele možemo lako da uočimo da postoji test slučaj za koje stanje nije definisano. To je test slučaj koji odgovara situaciji u kojoj korisnik sistema želi da ugasi program, a kasa nije zatvorena (poslednji red tabele). Mogućnost ostavljanja otvorene kase nije očigledna na dijagramu stanja 4.11, ali jeste u tabeli stanja. Moguća rešenja su da sistem upozori korisnika i spreči zatvaranje programa ili da automatski zatvori kasu.

4.3.4 Testiranje bele kutije

Testiranje bele kutije podrazumeva detaljno poznavanje unutrašnje strukture softverskog sistema. Test slučajevi se kreiraju na osnovu analize izvornog koda, pri čemu se posebno proučava tok izvršavanja programa. Najčešće ih izrađuju sami programeri tokom razvoja, ali ih mogu pisati i iskusni test inženjeri koji imaju pristup kodu i razumevanje implementacije.

Zbog potrebe za dubinskim uvidom u rad sistema, testiranje bele kutije je zahtevnije i skuplje u poređenju sa tehnikama crne kutije. Zato se najčešće primenjuje u sistemima za koje je potrebna visoka pouzdanost i gde su softverske greške posebno skupe.

Cilj ovog testiranja je ispitivanje različitih *putanja* kroz kôd programa. Putanja (eng. *execution path*) predstavlja konkretan niz naredbi koje se izvršavaju tokom jednog prolaska kroz program, u zavisnosti od ulaznih podataka i kontrole toka programa. Ključni elementi koji utiču na formiranje putanja su:

- uslovna grananja (na primer, naredbe `if` i `switch`),
- petlje (na primer, naredbe `while`, `for` i `do-while`),
- skokovi (na primer, `break`, `continue`, `return` i `goto`),
- pozivi funkcija,
- izuzeci i obrada izuzetaka (na primer, naredbe `try`, `catch` i `finally`).

Primer 4.3.13 (Funkcija `pozitivni`) Posmatrajmo kôd funkcije koja proverava da li je broj pozitivan.

```
1  int pozitivni(int a) {
2    if (a > 0)
3      return 1;
4    return 0;
5  }
```

Ova funkcija ima dve putanje. Prva putanja nakon provere uslova `a > 0` (linija 2) izvršava naredbu `return 1` (linija 3). Druga putanja nakon provere uslova `a > 0` (linija 2) izvršava naredbu `return 0` (linija 3).

Analogno ispitivanju svih kombinacija ulaza kod tehnika zasnovanih na modelu crne kutije, može se zahtevati ispitivanje svih putanja kroz program. Međutim, zbog eksponencijalnog rasta broja mogućih putanja sa porastom složenosti programa, potpuno testiranje svih putanja je u praksi najčešće neizvodljivo.

Primer 4.3.14 (Funkcija `pozitivni` — nastavak) Posmatrajmo kôd funkcije koja za tri broja računa koliko njih je pozitivno.

```
1  int pozitivni(int a1, int a2, int a3) {
2    int brojPozitivnih = 0;
3    if (a1 > 0)
4      brojPozitivnih++;
5    if (a2 > 0)
6      pozitivan++;
7    if (a3 > 0)
8      brojPozitivnih++;
9    return brojPozitivnih;
10 }
```

Broj putanja kroz ovaj program je 2^3, jer za svaku od tri naredbe grananja imamo opciju izvršavanja ukoliko je uslov ispunjen i ukoliko nije. Konkretno, to su putanje koje prolaze kroz naredne linije koda:

1. 1, 2, 3, 4, 5, 6, 7, 8, 9
2. 1, 2, 3, 4, 5, 6, 7, 9
3. 1, 2, 3, 4, 5, 7, 8, 9
4. 1, 2, 3, 4, 5, 7, 9
5. 1, 2, 3, 5, 6, 7, 8, 9
6. 1, 2, 3, 5, 6, 7, 9
7. 1, 2, 3, 5, 7, 8, 9
8. 1, 2, 3, 5, 7, 9

Primer 4.3.15 (Funkcija `pozitivni` — nastavak) Posmatrajmo kôd funkcije koja za niz od 100 brojeva računa koliko njih je pozitivno.

```
int pozitivni(int a[], int n) {
  int i = 0, brojPozitivnih = 0;
  while (i < n) {
    if (a[i] > 0)
      brojPozitivnih++;
  }
  return brojPozitivnih;
}
```

Broj putanja kroz ovaj program je 2^{100}, jer za svaku naredbu grananja imamo opciju izvršavanja ukoliko je uslov ispunjen i ukoliko nije. Bez pretpostavke da niz ima tačno 100 elemenata, broj putanja kroz ovu funkciju zavisi od veličine niza n i iznosi 2^n za fiksiranu vrednost n.

Primer 4.3.16 (Funkcija `pozitivni` — nastavak) Posmatrajmo kôd funkcije koja za vrednosti koje se unose sa standardnog ulaza (sve do unosa broja nula) računa koliko njih je pozitivno.

```
int pozitivni() {
  int broj, brojPozitivnih = 0;
  do {
    scanf("%d", &broj);
    if (broj > 0)
      brojPozitivnih++;
  } while (broj!=0);
  return brojPozitivnih;
}
```

Broj putanja kroz ovaj program zavisi od broja unetih elemenata sa standardnog ulaza i nije ograničen.

Zbog velikog broja mogućih putanja i kroz jednostavne pro-

grame, umesto testiranja svih mogućih putanja, koriste se metrike pokrivenosti koda kako bi se obezbedio balans između kvaliteta i troškova testiranja. Pre početka testiranja, treba odrediti odgovarajući nivo i vrstu pokrivenosti. Osnovni koraci u testiranju bele kutije su:

1. Analiza i razumevanje strukture i logike izvornog koda;
2. Kreiranje test primera na osnovu
   a) identifikovanih relevantnih putanja i
   b) odabranih metrika pokrivenosti;
3. Izvršavanje test primera.

Metrike pokrivenosti koda

Metrike pokrivenosti koda predstavljaju važno sredstvo za procenu kvaliteta testiranja softvera. One pomažu u identifikaciji delova sistema koji su testirani i onih koji su ostali neprovereni, čime omogućavaju donošenje informisanih odluka o daljem razvoju i unapređenju testne infrastrukture. Metrike pokrivenosti koda koje se najčešće koriste su pokrivenost putanja, naredbi, odluka, uslova, višestrukih uslova i funkcija.

Pokrivenost putanja = $\frac{\text{Izvršenih putanja}}{\text{Ukupno putanja}} \times 100\%$

Pokrivenost putanja (eng. *path coverage*) Mera koja pokazuje koliki je procenat putanja u programu koje su izvršene tokom testiranja. Potpuna pokrivenost znači da je svaka moguća putanja kroz program izvršena bar jednom.

Pokrivenost naredbi = $\frac{\text{Izvršenih naredbi}}{\text{Ukupno naredbi}} \times 100\%$

Pokrivenost naredbi (eng. *statement coverage*) Mera koja pokazuje koliki je procenat naredbi u programu koje su izvršene tokom testiranja. Potpuna pokrivenost se postiže kada je svaka naredba u programu izvršena bar jednom.

Pokrivenost odluka = $\frac{\text{Ispitanih odluka}}{\text{Ukupno odluka}} \times 100\%$

Pokrivenost grana/odluka (eng. *branch/decision coverage*) Mera koja pokazuje koliki je procenat ishoda odluka testirano u odnosu na ukupan broj odluka u programu. Za potpunu pokrivenost, svaka odluka u kodu mora biti evaluirana bar jednom kao tačna i bar jednom kao netačna.

Pokrivenost uslova = $\frac{\text{Ispitanih uslova}}{\text{Ukupno uslova}} \times 100\%$

Pokrivenost uslova (eng. *condition coverage*)* Mera koja pokazuje koliki je procenat pojedinačnih logičkih uslova unutar složenijih izraza (npr. `A && B`) dobio i tačnu i netačnu vrednost tokom testiranja. Za potpunu pokrivenost, svaki uslov u svakoj odluci u kodu mora

* Za lakše razumevanje razlika između pokrivenosti odluka, uslova i višestrukih uslova, pogledati primer 4.3.18.

biti evaluiran bar jednom kao tačan i bar jednom kao netačan.

Pokrivenost višestrukih uslova (eng. *multiple condition coverage*) Mera koja pokazuje koliki je procenat mogućih kombinacija vrednosti pojedinačnih uslova u okviru svake odluke izvršen tokom testiranja. Za potpunu pokrivenost, svaka moguća kombinacija vrednosti pojedinačnih uslova za svaku odluku mora biti izvršena bar jednom.

Pokrivenost višestrukih uslova = $\frac{\text{Izvršenih kombinacija viš. uslova}}{\text{Ukupno kombinacija viš. uslova}} \times 100\%$

Pokrivenost funkcija (eng. *function coverage*) Mera koja pokazuje koliki je procenat funkcija izvršen tokom testiranja. Za potpunu pokrivenost, potrebno je obezbediti da je svaka funkcija bar jednom pozvana.

Pokrivenost funkcija = $\frac{\text{Izvršenih funkcija}}{\text{Ukupno funkcija}} \times 100\%$

Pokrivenost funkcija je korisna za osnovnu proveru testne infrastrukture. Ukoliko neka funkcija nije nikada pozvana tokom testiranja, to znači da taj deo softvera nije uopšte pokriven testovima, što može ukazivati na potencijalne rizike ili nepotrebni (mrtvi) kod.

Najčešće korišćena metrika u praksi je pokrivenost naredbi. Ova metrika je jednostavna za implementaciju i brzo se izračunava prilikom pokretanja testova pa je njeno izračunavanje podržano u okviru većine modernih razvojnih okruženja. Međutim, pokrivenost naredbi ne odražava složenost logike programa i ne garantuje da su sve bitne situacije obuhvaćene testovima.

Za detaljniju analizu ponašanja softvera, neophodno je koristiti naprednije metrike, kao što je metrika pokrivenosti grana/odluka. Pokrivenost grana pruža uvid u to da li su testovi prošli kroz sve moguće logičke tokove.

Primer 4.3.17 (Maksimalna vrednost) Razmorimo naredni kôd

```
1 max = b;
2 if (a > max) {
3     max = a;
4 }
```

Test {a=6, b=3} pokriva sve naredbe (izvršavanje prolazi kroz naredbe 1, 2, 3 i 4), ali ne i sve putanje (nedostaje putanja 1, 2, 4) kao ni sve odluke (odluka u liniji 2 se razmatra samo kao tačna, ne i kao netačna).

Testovi {a=6, b=3} i {a=3, b=6} pokrivaju i sve naredbe, sve putanje i sve odluke. Dodati drugi test pokriva putanju kroz linije 1, 2, 4. Odluka u liniji 2 u ovom testu ima vrednost netačno.

Međutim, za visokokvalitetno testiranje kao i za kritične delove sistema, gde su greške neprihvatljive ili izuzetno skupe, prati se pokrivenost putanja, uslova i višestrukih uslova. Odabir odgovarajuće metrike treba da zavisi od konteksta upotrebe softvera, njegove složenosti i posledica potencijalnih grešaka.

Primer 4.3.18 (Inkrementiranje pozitivnih brojeva) Razmotrimo naredni kôd i njegovu pokrivenost testovima.

```
1  if(a > 0 && b > 0) {
2    a++;
3    b++;
4  }
```

Test
{a=1, b=2}
pokriva sve naredbe ali

- ne pokriva sve odluke — nedostaje odluka da uslov $a > 0$ && $b > 0$ nije ispunjen,
- ne pokriva sve uslove — nedostaje da su vrednosti oba podizraza netačne,
- ne pokriva sve višestruke uslove — nedostaju još tri kombinacije vrednosti podrizraza, tj. testovi za koje su vrednosti podrizraza tačno-netačno, netačno-tačno i netačno-netačno,
- ne pokriva sve putanje — nedostaje putanje koja prolazi samo kroz liniju 1.

Testovi
{a=1, b=2} i
{a=-1, b=2}
pokrivaju sve naredbe, sve odluke i sve putanje, ali

- ne pokrivaju sve uslove — nedostaje da je vrednost podizraza $b > 0$ netačna,
- ne pokrivaju sve višestruke uslove — nedostaju još dve kombinacije vrednosti podrizraza, tj. testovi za koje su vrednosti podrizraza tačno-netačno i netačno-netačno.

Testovi
{a=1, b=2},
{a=-1, b=2} i
{a=-1, b=-2}
pokrivaju sve naredbe, sve odluke, sve putanje i sve uslove, ali ne pokrivaju sve višestruke uslove, nedostaje kombinacija vrednosti podizraza koja odgovara vrednostima tačno-netačno. Testovi
{a=1, b=2},
{a=-1, b=2},
{a=-1, b=-2} i
{a=1, b=-2}
pokrivaju sve naredbe, sve odluke, sve putanje, sve uslove i sve višestruke uslove.

Odnosi između različitih metrika

Potpuna pokrivenost naredbi podrazumeva potpunu pokrivenost funkcija. Potpuna pokrivenost odluka podrazumeva potpunu pokrivenost naredbi, dok potpuna pokrivenost putanja podrazumeva potpunu pokrivenost odluka (slika 4.12).

Slika 4.12: Odnosi različitih pokrivenosti počevši od pokrivenosti putanja

Primer 4.3.19 (Apsolutne vrednosti) Potpuna pokrivenost odluka ne podrazumeva potpunu pokrivenost putanja. Na primer, razmotrimo naredni kôd.

```
1  if (a < 0) {
2     a = -a;
3  }
4  if (b < 0) {
5     b = -b;
6  }
```

Test {a=-6, b=-3} pokriva sve naredbe, ali ne pokriva sve odluke niti sve putanje.

Testovi {a=-6, b=-3} i {a=6, b=3} pokrivaju sve naredbe i sve odluke. Međutim, ne pokrivaju sve putanje.

Testovi {a=-6, b=-3}, {a=6, b=3}, {a=-6, b=3} i {a=6, b=-3} pokrivaju sve naredbe, sve odluke i sve putanje.

Takođe, potpuna pokrivenost višestrukih uslova podrazumeva potpunu pokrivenost uslova i potpunu pokrivenost odluka (slika 4.13). Međutim, potpuna pokrivenost uslova i višestrukih uslova nisu u nužnoj vezi sa potpunom pokrivenošću putanja.

Slika 4.13: Odnosi različitih pokrivenosti vezanih za uslove u granajima

Primer 4.3.20 (Inkrementiranje pozitivnih brojeva) Razmotrimo naredni kôd i njegovu pokrivenost testovima.

```
1  if(a > 0 && b > 0) {
2    a++;
3    b++;
4  }
5  if(c > 0 && d > 0) {
6    c++;
7    d++;
8  }
```

Testovi
{a=1, b=2, c=3, d=4} i
{a=-1, b=2, c=-3, d=4}
pokrivaju sve naredbe i sve odluke (ali ne i sve uslove, sve višestruke uslove ni sve putanje).

Testovi
{a=1, b=2, c=3, d=4} i
{a=-1, b=-2, c=-3, d=-4}
pokrivaju sve naredbe, sve odluke i sve uslove (ali ne i sve višestruke uslove ni sve putanje).

Testovi
{a=1, b=2, c=3, d=4},
{a=1, b=-2, c=3, d=-4},
{a=-1, b=2, c=-3, d=4} i
{a=-1, b=-2, c=-3, d=-4}
pokrivaju sve naredbe, sve odluke, sve uslove i sve višestruke uslove, ali ne i sve putanje.

Testovi
{a=1, b=2, c=3, d=4},
{a=1, b=2, c=-3, d=4},
{a=-1, b=2, c=3, d=4} i
{a=1, b=-2, c=3, d=-4}
pokrivaju sve naredbe, sve odluke, sve uslove i sve putanje, ali ne i sve višestruke uslove.

Preporučeni nivo pokrivenosti

U praksi, pitanje *„Koliki nivo pokrivenosti je dovoljno dobar?“* nema univerzalni odgovor i zavisi od prirode softverskog sistema, njegovog konteksta upotrebe, kritičnosti i ciljeva testiranja. Ipak, industrijska praksa pokazuje određene smernice koje se često koriste kao polazna tačka.

Na primer, za pokrivenost naredbi, iako se potpuna pokrivenost često postavlja kao cilj, kao preporučeni prag navodi se nivo od 80% do 90%. Ova vrednost predstavlja kompromis između efikasnosti testiranja i troškova njegove realizacije. Važno je napomenuti da čak i potpuna pokrivenost ne garantuje odsustvo grešaka. Ona samo znači da je sav kôd bio izvršen tokom testiranja, ali ne i da su obuhvaćeni svi logički putevi, uslovi ili granični slučajevi, odnosno logički propusti i dalje mogu ostati neprimećeni. Sa druge strane, nizak nivo pokrivenosti jasno ukazuje na povišen rizik da greške mogu ostati neotkrivene i sugeriše da značajni delovi koda nisu uopšte testirani.

Pokrivenost koda testovima se menja tokom razvoja softvera i dodavanja novih funkcionalnosti. Zbog toga je potrebno redovno merenje i praćenje pokrivenosti koda testovima.

Alati za merenje pokrivenosti koda

Alati za merenje pokrivenosti koda koriste se za analizu koji delovi izvornog koda su bili izvršeni tokom testiranja. Oni predstavljaju ključnu podršku u procesu testiranja softvera, jer omogućavaju jasan uvid u to koliko je testiranje temeljno.

Većina alata funkcioniše na osnovu instrumentacije — procesa u kojem se u kôd automatski umeću dodatne instrukcije koje beleže rezultate izvršavanja. Instrumentacija može da se vrši pre ili za vreme procesa kompilacije, kao i nakon kompilacije, na nivou bajtkoda (npr. za Java programe) ili izvršivog formata (npr. za kompajlirane C/C++ binarne fajlove).

Izvršavanjem instrumentovanog koda nad ulazima koje određuju testovi, prikupljaju se podaci o tome koje su linije, grane, uslovi ili funkcije izvršeni. Na osnovu tih informacija generiše se izveštaj o pokrivenosti, koji pomaže programerima da identifikuju nedovoljno testirane delove sistema. Ovi izveštaji se često vizualizuju kroz procente pokrivenosti i označene delove koda (npr. bojama), kako bi se lakše uočili segmenti koji zahtevaju dodatne testove.

Izbor relevantnih testova i putanja kroz program

Jedan od ključnih izazova u testiranju metodama bele kutije jeste izbor odgovarajućih test primera. Dobar izbor testova treba da obezbedi visok nivo pokrivenosti uz minimalan broj

test primera, čime se postiže efikasnost i izbegava redundantnost.

U praksi, preporučuje se korišćenje više jednostavnih i kratkih putanja koje se međusobno razlikuju u malim detaljima, umesto jedne veoma složene putanje. Takav pristup olakšava dijagnostikovanje grešaka i održavanje testova.

Primer 4.3.21 (Petlje) Petlje u kodu mogu potencijalno generisati beskonačan broj putanja, pa je važno odabrati reprezentativne slučajeve, kao što su:

- Petlja se preskače (nula iteracija).
- Petlja se izvršava jednom.
- Petlja se izvršava dva puta.
- Ako postoji poznat gornji limit n, izvršavanje $n-1$ i n puta.

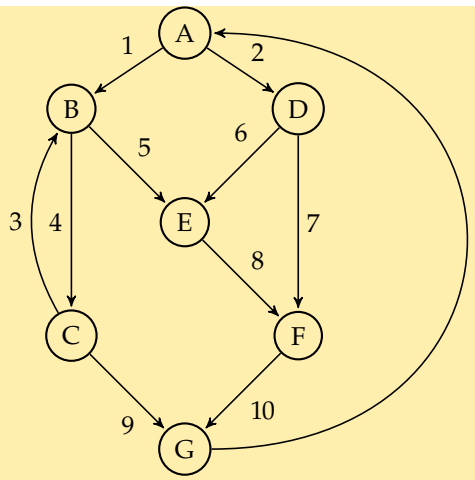
Testiranje osnovnih putanja (eng. *basis path testing*) predstavlja tehniku kojom se osigurava da su sve logičke grane u programu ispitane bar jednom. Ova tehnika se zasniva na analizi grafa toka kontrole (eng. *control flow graph*) i uključuje sledeće korake:

1. Izgradnja grafa kontrole toka programa iz posmatranog softverskog modula;
2. Izračunavanje ciklomatične kompleksnosti grafa[†], označene sa C;
3. Odabir skupa od C linearno nezavisnih osnovnih putanja;
4. Kreiranje test primera za svaku osnovnu putanju;
5. Izvršavanje testova i analiza rezultata.

Primena testiranja osnovnih putanja garantuje pokrivenost svih naredbi i svih grana u kodu, jer je skup osnovnih putanja konstruisan tako da pokriva svaki čvor i svaku logičku odluku u grafu toka upravljanja.

Primer 4.3.22 (Osnovne putanje) Naredni graf odgovara grafu kontrole toka programa P.

[†] Ciklomatična kompleksnost je broj linearno nezavisnih putanja kroz graf. Računa se kao vrednost izraza $E - N + 2P$ gde je E broj grana u grafu, N broj čvorova grafa, a P broj komponenti povezanosti grafa.



Ciklomatična kompleksnost za dati graf je 5, jer je

$$C = 10 - 7 + 2 = 5$$

Primenimo sada algoritam odabira 5 linearno nezavisnih osnovnih putanja. U svakom čvoru odluke potrebno je promeniti odluku, počevši od poslednje donete odluke. U skladu sa time, imamo naredne putanje:

1. A, B, C, G
2. A, B, C, B, C, G
3. A, B, E, F, G
4. A, D, E, F, G
5. A, D, F, G

Glavni izazov ove tehnike leži u pretpostavci da su sve osnovne putanje dostižne i izvodljive. U složenim programima, određene putanje mogu biti logički ili praktično nedostižne zbog međuzavisnosti uslova. Zbog toga je neophodna pažljiva analiza grafa kontrole toka i dodatna provera da izabrane putanje zaista odgovaraju realnim scenarijima izvršavanja.

Testiranje na osnovu grafa toka podataka (eng. *data flow testing*) ispituje ispravnost životnog ciklusa promenljivih i upotrebe promenljivih u različitim delovima programa. Tipične anomalije u upotrebi podataka uključuju:

- **Upotreba neinicijalizovane promenljive** – promenljiva se koristi pre nego što joj je dodeljena vrednost.
- **Neiskorišćena definicija** – promenljiva je definisana i inicijalizovana, ali se nikada ne koristi.

- **Ponovno definisanje** – promenljivoj se dodeljuje nova vrednost pre nego što je prethodna upotrebljena.
- **Upotreba nakon oslobađanja resursa** – promenljiva (ili referenca) se koristi nakon dealokacije memorije.
- **Neusaglašenost dodeljivanja i korišćenja u različitim granama izvršavanja**.

Cilj testiranja je osigurati da se sve promenljive koriste na ispravan način — da su inicijalizovane pre upotrebe, iskorišćene nakon dodeljivanja i oslobođene samo kada više nisu potrebne. Za to je potrebno u kodu identifikovati mesta gde se promenljive definišu i gde se koriste, i kreirati test primere koji pokrivaju tok izvršavanja programa koji prolazi kroz date definicije i korišćenja. Ova vrsta testiranja je naročito korisna za otkrivanje suptilnih grešaka koje nisu nužno vidljive kroz standardne tehnike testiranja kontrole toka i često se koristi u bezbednosno kritičnim i složenim sistemima.

4.3.5 Metamorfno testiranje

Metamorfno testiranje predstavlja tehniku testiranja softverskih sistema koja omogućava proveru ispravnosti bez potrebe za proročištem (eng. *oracle*) — komponentom koja određuje da li je rezultat testa ispravan. Umesto toga, koristi se poznavanje osobina sistema koji se testira, odnosno *metamorfna svojstava (relacije)*.

Metamorfna relacija izražava kako bi se izlaz programa trebalo da promeni kada je ulaz izmenjen na određeni način. Osnovna ideja je da se na osnovu dva ulaza koja su u nekom specifičnom odnosu odredi odnos između odgovarajućih izlaza.

Primer 4.3.23 (Standardna devijacija) Ukoliko ne možemo da proverimo ispravnost rezultata za računanje standardne devijacije za veoma dugačak niz brojeva, možemo da iskoristimo sledeće odnose (metamorfna svojstva):

- Permutacija elemenata ne utiče na standardnu devijaciju.
- Množenje svake vrednosti sa -1, ne utiče na standardnu devijaciju.
- Ako se svaki broj pomnoži sa nekom konstantom, standardna devijacija novog niza brojeva bi trebalo

da je srazmerna standardnoj devijaciji originalnog niza.

Testiranjem se proverava da li se za zadate ulaze na očekivani način menjaju izlazi. Testiranjem se detektuje greška u sistemu ukoliko se javi odstupanje u očekivanim odnosima između odgovarajućih izlaza.

Primer 4.3.24 (Standardna devijacija — nastavak) Na osnovu identifikovanih metamorfnih relacija za algoritam standardne devijacije, možemo da napravimo test primere koji su permutacija istog niza brojeva i da proverimo da li svaki put dobijemo isti rezultat. Takođe, možemo da napravimo test primere sa skaliranim vrednostima niza i da proverimo da li za dobijenu devijaciju važi da je skalirana u odnosu na devijaciju originalnog niza. Na primer, možemo da napravimo naredne testove:

- **Test 1**: Proveriti za četiri permutacije istog niza da li daju isti rezultat.
- **Test 2**: Proveriti da li se dobija isti rezltat za početni niz i za niz u kojem su sve vrednosti pomnožene sa -1.
- **Test 3**: Proveriti da li su srazmerne devijacije za početni niz, niz u kojem su svi elementi pomnoženi sa 2, i niz u kojem su svi elementi pomnoženi sa -5.

Metamorfno testiranje značajno doprinosi povećanju stepena automatizacije u testiranju i omogućava detekciju grešaka u situacijama kada tradicionalne metode nisu primenljive. Za uspešnu primenu metamorfnog testiranja neophodno je:

- dobro razumevanje domena problema,
- identifikacija relevantnih i pouzdanih metamorfnih svojstava,
- automatizacija generisanja izvedenih testova i njihove verifikacije.

Ključni korak u primeni metamorfnog testiranja jeste identifikacija odgovarajućih metamorfnih relacija. Kvalitet i snaga ovih relacija direktno utiču na efikasnost testiranja.

Jedan od najkorisnijih tipova metamorfnih relacija je relacija ekvivalencije. Ova relacija podrazumeva da transformisani ulaz mora proizvesti potpuno isti izlaz kao i original, što omogućava jednostavnu i preciznu detekciju nepravilnosti u

ponašanju sistema. Zbog svoje stroge prirode, relacija ekvivalentnosti omogućava detektovanje širokog spektra grešaka.

Primer 4.3.25 (Konstrukcija kompilatora) Veoma je teško utvrditi ekvivalenciju između izvornog koda i izvršivog koda. Jedna alternativa je da se u dati izvorni kôd dodaju određene naredbe ili delovi naredbi koje mu ne menjaju semantiku. Na primer, to može biti dodavanje nule izrazu (time se ne menja vrednost izraza) ili dodavanje uslova koji je uvek ispunjen (uslov `if (true) ...` takođe ne menja semantiku programa). Za tako izmenjene izvorne programe, trebalo bi da bude generisan izvršivi kôd koji se ponaša na isti način kao i početni kôd.

Primer 4.3.26 (Mašinsko učenje — klasifikacija slika) Razmotrimo klasifikator slika obučen da prepoznaje kategorije objekata na slikama. Jedan primer metamorfnog odnosa je:

> Ako se ulazna slika rotira za mali ugao (npr. do 10°), očekuje se da klasifikacija ostane nepromenjena.

Na osnovu ove osobine, moguće je konstruisati sledeći postupak testiranja:

1. Izabrati originalni test primer x (sliku) za koji klasifikator daje rezultat $f(x)$.
2. Generisati transformisani primer x' (npr. rotacija slike za 5°).
3. Uporediti rezultate $f(x)$ i $f(x')$; ukoliko se razlikuju, to može ukazivati na potencijalnu grešku u modelu, ili na potrebu za dodatnim podacima u obuci.

Međutim, u praksi, relacija ekvivalencije nije uvek dostupna metamorfna relacija. U tim slučajevima koriste se slabije forme, kao što su relacije očuvanja određenih karakteristika rezultata ili relacije koje predviđaju smer promene izlaza. Pravilna selekcija metamorfnih relacija zahteva duboko razumevanje testiranog sistema i domena primene.

Primer 4.3.27 (Predviđanje cene nekretnine) Sistem predviđa cenu nekretnine na osnovu kvadrature, lokacije i broja soba. U slučaju povećanje kvadrature, za isti ili veći broja

soba i za nekretninu na istoj lokaciji, očekuje se povećanje predviđene cene.

Na primer, ako imamo kao ulaz nekretninu koja ima 60m^2 i tri sobe, za koju sistem predviđa cenu 90.000 evra, onda možemo da kreiramo novi ulaz tako što samo povećamo kvadraturu na 80m^2. Za ovaj novi ulaz očekujemo da cena treba da bude veća od 90.000 evra. Izlaz se menja, ali postoji logički odnos između prvog i drugog izlaza (monotonost u predikciji), što se koristi za proveru ponašanja modela.

4.4 Načini sprovođenja testiranja

U procesu testiranja mogu se izdvojiti dve osnovne aktivnosti:

(i) definisanje test primera koji će biti predmet provere i
(ii) sprovođenje postupka izvršavanja i evaluacije tih test primera.

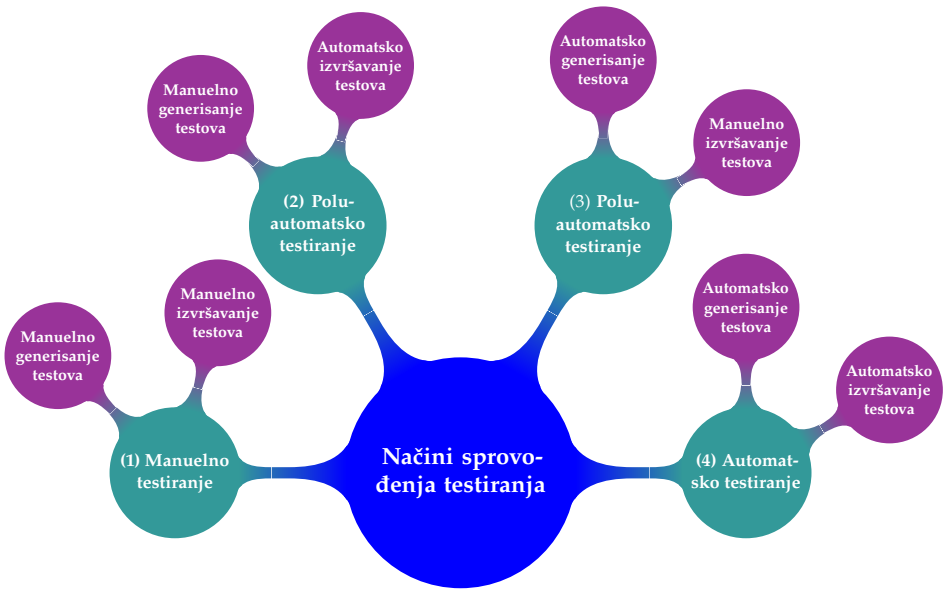
Obe ove aktivnosti, prema načinu na koji se sprovode, mogu biti obavljane manuelno ili automatski. Iako postoje četiri moguće kombinacije:

(1) manuelno generisanje testova i manuelno sprovođenje testiranja,
(2) manuelno generisanje testova i automatsko sprovođenje testiranja,
(3) automatsko generisanje testova i manuelno sprovođenje testiranja i
(4) automatsko generisanje testova i sprovođenje testiranja,

u praksi se (3) obično ne koristi.

4.4.1 Manuelno testiranje

Termin manuelno testiranje podrazumeva da se obe aktivnosti sprovode manuelno, odnosno da testeri smišljaju test slučajeve i samostalno sprovode testiranje i ocenjuju rezultate testiranja. Manuelno testiranje je nezamenjivo u delovima verifikacije koji zahtevaju ljudsku percepciju, razumevanje konteksta i subjektivnu procenu kvaliteta. Dodatno, manuelno testiranje je ključno u domenu validacije softvera, gde je cilj da se utvrdi da li softver zaista zadovoljava potrebe



Slika 4.14: Načini izvođena testiranja softvera

i očekivanja krajnjih korisnika, a ne samo da li funkcioniše prema tehničkim specifikacijama.

U kontekstu verifikacije softvera, primer upotrebe manuelnih tehnika testiranja je istraživačko testiranje, koje se oslanja na iskustvo i intuiciju testera. Umesto striktno definisanih koraka, tester istražuje softver u realnom vremenu, tražeći neočekivane greške i ponašanja. Automatizacija ne može da zameni ovu vrstu kreativnog pristupa.

Manuelnim testiranjem često se proveravaju svojstva koja se nalaze na preseku validacije i verifikacije softvera. Na primer, testiranjem prihvatljivosti, koje obično sprovodi sam korisnik ili predstavnik korisnika, procenjuje se da li je softver upotrebljiv i spreman za puštanje u produkciju. Osim što uključuje funkcionalne provere, često podrazumeva i subjektivne ocene o korisničkom iskustvu i interfejsu.

Slično tome, aplikacije u realnom vremenu i interaktivne aplikacije, poput video igara, sistema za virtuelnu realnost ili specijalizovanih simulacija, zahtevaju manuelnu evaluaciju ponašanja sistema tokom neposredne upotrebe. U takvim slučajevima, subjektivno korisničko iskustvo ima ključnu ulogu u oceni ispravnosti i kvaliteta softvera.

U kontekstu validacije softvera, provera korisničkog interfejsa često zahteva manuelni pristup. Vizuelni aspekti kao što

su poravnanje elemenata, čitljivost i jasnoća poruka najbolje se procenjuju ljudskim okom. Automatizovani alati mogu potvrditi prisustvo odgovarajućih elemenata, ali ne i njihovu funkcionalnu ili estetsku vrednost. Slično, testiranja vezana za ocenu naučivosti i intuitivnosti korišćenja fokusiraju se na iskustvo krajnjeg korisnika. Ona ocenjuju koliko je softver jednostavan za učenje i upotrebu, što zahteva direktno posmatranje korisnika, analizu ponašanja i prikupljanje povratnih informacija.

4.4.2 Automatsko izvršavanje test primera

Iako manuelno testiranje omogućava uvid u korisničko iskustvo i subjektivne aspekte sistema, ono je često podložno greškama, sporo i teško skalabilno. Zbog toga je automatizacija testiranja od ključnog značaja, naročito kod složenih sistema i testova koji se često ponavljaju. Automatizovani testovi smanjuju potrebu za ručnim intervencijama, ubrzavaju razvoj i unapređuju kvalitet softverskog proizvoda.

Najčešći oblik automatizovanog testiranja je izvršavanje testova jedinica koda. Ovakvi testovi se često automatski pokreću prilikom svake izmene u kodu, a njihova integracija sa alatima za merenje pokrivenosti koda omogućava praćenje kvaliteta testiranja. Postoji veliki broj okruženja za automatsko izvršavanje testova, poznatih po obrascu *xxxUnit*, gde *xxx* označava konkretni programski jezik. Na primer, za C++ postoji *CppUnit*, za Javu *JUnit*, a za Python *PyUnit*.

Testiranje veb aplikacija
Selenium je softver otvorenog koda koji se dominantno koristi za automatizovano testiranje veb aplikacija. *Selenium* podržava testiranje veb aplikacija u gotovo svim dostupnim pretraživačima. Test skriptovi mogu biti pisani u različitim programskim jezicima, kao što su C#, Java, Ruby, Python i Perl i pokretani na operativnim sistemima *Windows*, *macOS* ili *Linux*.

Pored alata za izvršavanje testova, postoje i alati koji automatizuju čitav proces testiranja — uključujući upravljanje test slučajevima, integraciju sa repozitorijumima koda, praćenje defekata i generisanje izveštaja. Ovi alati su često deo većih razvojnih okruženja ili sistema za kontinuiranu integraciju.

Kontinuirana integracija softvera

Automatizacija omogućava brzo, ponovljivo i pouzdano izvršavanje test primera, što je posebno važno u okruženjima kontinuirane integracije i isporuke softvera. Kontinuirana integracija (eng. *continuous integration, CI*) predstavlja ključni proces u razvoju softvera, posebno u timovima gde više članova svakodnevno unosi izmene u zajednički kodni repozitorijum. Cilj kontinuirane integracije je da se svaka izmena

odmah objedini sa trenutnim stanjem projekta, automatski izgradi i testira, kako bi se potencijalne greške otkrile što ranije.

Popularni alati za kontinuiranu integraciju:

- Jenkins
- Buildbot
- Travis CI
- GitLab CI
- CircleCI
- TeamCity (JetBrains)
- Bamboo (Atlassian)

Prilikom svakog objedinjavanja koda, sistem prolazi kroz sledeće faze:

Integracija — sve trenutne izmene se spajaju u jedinstvenu verziju projekta,

Izgradnja — kôd se kompajlira i pakuje u izvršni fajl ili instalacioni paket,

Testiranje — automatski test primeri se izvršavaju kako bi se proverila ispravnost sistema,

Arhiviranje — rezultujući artifakti se verzionišu i čuvaju za buduću upotrebu,

Primena — sistem se učitava u okruženje za testiranje ili integraciju, gde može biti pokrenut i dostupan za evaluaciju.

Ovakav pristup značajno doprinosi:

- ranom otkrivanju grešaka,
- smanjenju troškova projekta,
- kraćem vremenu razvoja,
- i nižem riziku prilikom isporuke novih verzija softvera.

4.4.3 Automatsko generisanje test primera

Generisanje test primera moguće je automatizovati samo za određene vrste testiranja, u kojima se mogu formalno definisati kriterijumi pokrivenosti i ponašanja sistema. Tipični primeri uključuju testiranje robusnosti (vidi poglavlje **??**) i testove usmerene na otkrivanje grešaka u kodu kao što su deljenje nulom, korišćenje neinicijalizovanih promenljivih ili pristupi nedozvoljenim memorijskim lokacijama (vidi poglavlje 9).

U ovim scenarijima, automatizacija ima značajnu prednost jer omogućava sistematsko generisanje velikog broja ulaznih podataka uz minimalni ljudski napor. Posebno je korisna za otkrivanje ekstremnih i graničnih slučajeva koji bi lako mogli ostati neotkriveni manuelnim testiranjem. Na primer, alati za *fuzzing* mogu nasumično, ali ciljno, generisati razne kombinacije ulaza, uključujući i one koji izlaze iz uobičajenih opsega, kako bi izazvali neželjeno ili nepredviđeno ponašanje softverskog sistema.

Slično tome, statička analiza softvera omogućavaju identifikaciju potencijalno problematičnih mesta u kodu, kao što su upotreba promenljivih pre dodeljivanja vrednosti ili dereferenciranje `null` pokazivača. Na osnovu tih nalaza, mogu se automatski generisati test primeri koji ciljano testiraju te kritične tačke i tako povećavaju verovatnoću otkrivanja grešaka u ranim fazama razvoja.

Međutim, kada se radi o testiranju složenih funkcionalnosti koje zavise od višeslojnih uslova, konteksta upotrebe ili korisničkih odluka, automatizovano generisanje test primera postaje izuzetno izazovno, a često i neizvodljivo. U takvim situacijama, uspešno testiranje zahteva duboko razumevanje poslovne logike, očekivanog ponašanja sistema u realnim scenarijima, kao i interakcija među različitim komponentama softvera. Ove aspekte je teško formalizovati na način koji bi omogućio automatsko pouzdano i smisleno generisanje kvalitetnih test primera. Zbog toga se u praksi automatizovano generisanje testova najefikasnije koristi kao dopuna manuelnom procesu.

Rezime

- Testiranje softvera ima za cilj da otkrije defekte u softveru.
- Poželjno je ispravljanje grešaka u početnim fazama razvoja softvera.
- Testiranje mogu da obavljaju i programeri i testeri.
- Proces testiranja obuhvata planiranje, analizu, dizajn, implementaciju, evaluaciju, izvršavanje i zatvaranje procesa testiranja
- Osnovna podela testiranja je na testiranje funkcionalnih i nefunkcionalnih karakteristika softvera.
- Prema nivou testiranja, razlikujemo testove jedinice koda, komponentne, integracione i sistemske testove.
- Nefunkcionalno sistemsko testiranje uključuje testiranje performansi, kompatiblinosti, pouzdanosti, upotrebljivosti, bezbednosti, sigurnosti i prenosivosti.
- U sistemsko testiranje se ubrajaju i istraživačko testiranje, testiranje prihvatljivosti i instalaciono testiranje.
- Regresiono testiranje predstavlja proces ponovnog izvršavanja prethodno uspešno završenih testova.
- Pokrivenost testiranjem je metrika koja pomaže da se proceni koliko su testovi sveobuhvatni.

- Tehnike testiranja se dele na testiranje crne kutije, testiranje bele kutije i testiranje sive kutije.
- Metamorfno testiranje je pristup testiranju pri postojanju problema proročišta.
- Testiranje može da bude manuelno i automatizovano.

Ispitna pitanja

10. Testiranje u razvoju softvera. Cena greške u kontekstu vremena otkrivanja.
11. Testiranje u razvoju softvera. Uloga testera u razvoju softvera.
12. Testiranje u razvoju softvera. Faze testiranja softvera: planiranje, analiza, dizajn i implementacija testova.
13. Testiranje u razvoju softvera. Faze testiranja softvera: izvršavanje i evaluacija testova. Zatvaranje testiranja.
14. Vrste testiranja. Testiranje jedinca koda. Primeri.
15. Vrste testiranja. Komponentno i integraciono testiranje. Primeri.
16. Vrste testiranja. Sistemsko testiranje. Funkcionalno sistemsko testiranje. Regresiono testiranje. Primeri.
17. Vrste testiranja. Sistemsko testiranje. Istraživačko testiranje. Testovi prihvatljivosti. Instalaciono testiranje. Primeri.
18. Vrste testiranja. Nefunkcionalno testiranje. Testovi preformansi. Testovi kompatibilnosti. Testovi pouzdanosti. Primeri.
19. Vrste testiranja. Nefunkcionalno testiranje. Testovi upotrebljivosti. Testovi bezbdnosti. Testovi sigurnosti. Testovi prenosivosti. Primeri.
20. Tehnike testiranja. Karakteristike dobrog skupa tesotva. Pokrivenost testiranjem. Podela na tehnike testiranja.
21. Tehnike testiranja. Testiranje metodama crne kutije. Isprobavanja svih mogućih ulaza. Metod klasa ekvivalencije. Primeri.
22. Tehnike testiranja. Testiranje metodama crne kutije. Metod klasa ekvivalencije. Metod graničnih vrednosti. Primeri.
23. Tehnike testiranja. Karakteristike dobrog skupa tesotva. Tabele odlučivanja. Primeri.
24. Tehnike testiranja. Karakteristike dobrog skupa tesotva. Dijagrami stanja. Tabele stanja. Primeri.
25. Tehnike testiranja. Testiranje metodama bele kutije. Putanja i broj putanja u programu. Pojam i vrste pokrive-

nosti. Njihovi odnosi. Primeri.

26. Tehnike testiranja. Testiranje metodama bele kutije. Pojam i vrste pokrivenosti. Preporučen nivo pokrivenosti. Alati za merenje pokrivenosti. Primeri.
27. Tehnike testiranja. Testiranje metodama bele kutije. Izbor relevantnih testova i putanja. Testiranje osnovnih putanja. Testiranje na osnovu grafa toka podataka. Primeri.
28. Tehnike testiranja. Problem proročišta. Metamorfno testiranje. Primeri.
29. Načini testiranja. Manuelno testiranje.
30. Načini testiranja. Automatsko izvršavanje i evaluacija testova. Kontinuirana integracija.
31. Načini testiranja. Automatsko generisanje test primera.

Debagovanje 5

— If Broken it is, Fix it You Should —
(Yoda, Star Wars)

Pregled

- Kada se u okviru testiranja otkrije da postoji defekt u radu softvera, kako pronaći grešku koja je uzrokovala taj defekt?
- Koja je uloga debagovanja u procesu razvoja softvera?
- Zašto je debager sistemski alat?
- Koje vrste debagovanja postoje?
- Koje alternative debagovanju postoje?

Debagovanje je proces u razvoju softvera koji ima za cilj pronalaženja greške odnosno uzroka defekta u programu. Debager (eng. *debugger*) je alat koji se koristi za praćenje izvršavanja programa radi boljeg razumevanja ponašanja programa i lakšeg pronalaženja greške u programu.

Kako bi informacije koje debager pruža programeru bile precizne i razumljive, neophodna je podrška kompajlera i linkera, koji obezbeđuju povezivanje izvršivog koda sa izvornim kodom, kao i razne druge korisne podatke. Funkcionalnost debagera takođe zavisi od podrške operativnog sistema, a u određenim slučajevima i samog hardvera.

Debageri se često integrišu u razvojna okruženja koja im obezbeđuju grafički korisnički interfejs. Iako grafički interfejs može znatno olakšati upotrebu i unaprediti korisničko iskustvo, važno je imati na umu da je ono samo sredstvo interakcije sa debagerom, a ne njegov suštinski deo. Sama funkcionalnost debagera ostaje nezavisna od načina na koji se korisniku prikazuje.

Poznati debagr GDB, koji se razvija u okviru projekta GNU, se može koristiti kroz veliki broj razvojnih okruženja, uključujući *QtCreator*, *Visual Studio Code*, *Eclipse*, *NetBeans*, *CLion* i *IntelliJ*.

5.1 Veza izvršivog koda i debagera

Tokom kompilacije softverskog projekta dostupne su brojne opcije i podešavanja koja omogućavaju precizno upravljanje načinom prevođenja izvornog koda u izvršivi program. Ova

podešavanja se često koriste za optimizaciju performansi, lakše pronalaženje grešaka, ili prilagođavanje ciljnim platformama.

Režim kompilacije je skup opcija kompilacije koje se često zajedno koriste sa određenim ciljem. Postoje:

režim prevođenja za upotrebu (eng. *release mode*), skraćeno režim *riliz* — to je svaki režim koji ima za cilj dobijanje optimizovane izvršive verzije namenjene krajnjem korisniku,

režim prevođenja za pronalaženje grešaka (eng. *debug mode*), skraćeno režim *debag* — to je režim prevođenja koji ima za cilj dobijanje izvršive verzije programa namenjene programeru radi lakšeg otkrivanja grešaka u kodu i

kombinovani režim koji odgovara kombinaciji odabranih opcija režima za prevođenje za upotrebu i režima za pronalaženje grešaka — ovaj režim se koriste u situacijama kada je potrebno naći grešku u programu, a režim prevođenja za pronalaženje grešaka ne daje željene rezultate.

Na program koji se dobija prevođenjem za upotrebu često se referiše sa *riliz verzija programa*, dok se na program koji se dobija prevođenjem za pronalaženje grešaka referiše sa *debag verzija programa*.

Nakon generisanja izvršive verzije programa, moguće je primeniti specijalizovane alate koji modifikuju izvršivi kôd sa ciljem otežavanja i ometanja debagovanja*. Ovakve tehnike se nazivaju anti-debagovanje (eng. *anti-debugging*) i koriste se radi zaštite intelektualne svojine, kako bi se sprečilo analiziranje logike programa i krađa koda, ili pak u zlonamernom softveru, kako bi se onemogućilo prepoznavanje i analiziranje malicioznog ponašanja.

Debager se može koristiti za svaku izvršivu verziju programa. Mogućnosti i informacije koje se dobijaju za debag verziju su povezane sa izvornim kodom i olakšavaju programeru da poveže stanje izvršavanja sa izvornim kodom. S druge strane, informacije koje debager može da pruži za riliz verziju su često samo uvid u asemblerski kôd, na isti način kao što ih vidi i procesor. U slučaju primene anti-debagovanja, mogućnosti debagera su dodatno ograničene.

* Iako anti-debagovanje može da se koristi na svakoj izvršivoj verziji programa, ima ga smisla koristiti samo nad programima koji su prevedeni u reliz režimu.

5.1.1 Režim prevođenja za upotrebu

Primarni cilj prevođenja u režimu za upotrebu jeste postizanje visoke efikasnosti izvršavanja kroz različite strategije optimizacija koje sprovodi kompajler. Ove optimizacije mogu uključivati uklanjanje nepotrebnog koda, preuređivanje instrukcija radi boljeg iskorišćenja procesora, zamenu složenijih konstrukcija bržim ekvivalentima, kao i umetanje funkcija. Kao rezultat, generisani izvršivi kôd je kompaktniji i brži u odnosu na verziju prevedenu u režimu za uklanjanje grešaka.

Režim prevođenja za upotrebu podrazumeva optimizacije koje kao primarni cilj imaju optimizaciju brzine izvršavanja programa. Međutim, za neke aplikacije, na primer za one koje se izvršavaju na uređajima sa malom količinom memorije, neophodne su optimizacije koje se odnose na smanjivanje veličine izvršive datoteke kao i upotrebe memorije u fazi izvršavanja. Režim prevođenja za upotrebu u ovom slučaju se naziva režim prevođenja za upotrebu sa minimalnom veličinom (eng. *minimum size release mode*).

Sprovedene optimizacije smanjuju mogućnost povezivanja izvršivog koda sa izvornim datotekama. Neki delovi koda se mogu potpuno izostaviti, promeniti redosled ili transformisati do neprepoznatljivosti, što značajno otežava proces razumevanja izvršavanja koda i traženja greške u kodu. Zbog toga se ova verzija prevođenja gotovo nikada ne koriste za analizu i dijagnostiku grešaka.

Primer 5.1.1 (Prevođenje za upotrebu) Prilikom prevođenja programa pomoću kompajlera `gcc`, ne postoji jedinstvena opcija za režim prevođenja za upotrebu, već se to postiže kombinovanjem odgovarajućih kompajlerskih opcija. Ove opcije uključuju korišćenje visokog nivoa optimizacije, na primer `-O2` ili `-O3`, dok se za režim za upotrebu sa minimalnom veličinom koristi opcija optimizacije `-Os`. Dodatno, druga bitna opcija je `-DNDEBUG`. Ova opcija definiše makro `NDEBUG`, koji onemogućava izvršavanje poziva funkcije `assert()`, čija je upotreba karakteristična u fazi otklanjanja grešaka.

S druge strane, u mnogim sistemima za izgradnju koda, izbor režima može da se uključi jedinstvenom opcijom. Na primer, u sistemu *CMake*, koristi se opcija `-DCMAKE_BUILD_-`

`TYPE` koja se za režim prevođenja za upotrebu postavlja na vrednost `Release`, a za režim prevođenja za upotrebu sa minimalnom veličinom postavlja na vrednost `MinSizeRel`.

5.1.2 Režim prevođenja za pronalaženje grešaka

Primarni cilj prevođenja u režimu za pronalaženje grešaka je pravljenje izvršive verzije koda koja za svaku instrukciju u fazi izvršavanja omogućava preciznu povezanost sa odgovarajućim linijama izvornog koda. Kompajler u ovom režimu generiše dodatne informacije koji omogućavaju debageru da poveže izvršavane instrukcije sa odgovarajućim linijama izvornog koda.

Za razliku od režima riliz, gde su uključene optimizacije radi postizanja što efikasnijeg izvršavanja, u režimu debag su te optimizacije isključene ili svedene na minimum kako bi se očuvala što preciznija veza između izvornog i izvršivog koda. Zbog odsustva optimizacija, izvršiva datoteka generisana u režimu debag najčešće je znatno veća od datoteke prevedene u režimu riliz[†]. Takođe, izvršavanje programa prevedenog u ovom režimu može biti primetno sporije i manje efikasno u pogledu iskorišćenja memorije. Međutim, ta neefikasnost je prihvatljiva u fazi razvoja, jer omogućava precizno praćenje toka izvršavanja i stanja promenljivih.

Primer 5.1.2 (Prevođenje za pronalaženje grešaka) Prilikom prevođenja programa pomoću kompajlera `gcc`, ne postoji jedinstvena opcija za režim prevođenja za pronalaženje grešaka, ali se to postiže kombinovanjem opcije koja uključuje korišćenje najnižeg nivoa optimizacije `-O0` i opcije `-g` koja uključuje upisivanje informacija za debagovanje u izvršivu datoteku.

Kao što je već pominjano, u mnogim sistemima za izgradnju koda, izbor režima može da se uključi jedinstvenom opcijom. U sistemu *CMake*, opcija `-DCMAKE_BUILD_TYPE` se za režim prevođenja za pronalaženje grešaka postavlja na vrednost `Debug`.

[†] Izvršiva datoteka može biti značajno veća i iz drugih razloga. Na primer, informacije u formatu *DWARF* se upisuju direktno u izvršivu datoteku.

Formati za predstavljanje pomoćnih informacija

Formati za predstavljanje pomoćnih informacija potrebnih za debagovanje preciziraju način na koji kompajler može da zapiše podatke kao što su nazivi funkcija, nazivi promenljivih, tipovi podataka, odgovarajuće linije iz izvorne datoteke i slično. Najpoznatiji formati za predstavljanje pomoćnih informacija potrebnih za debagovanje su format *DWARF* i format *Microsoft CodeView*. Ovi formati nisu kompatibilni.

DWARF je dizajniran kao nezavisan format ali u slično vreme sa formatom za izvršive i povezane datoteke ELF. ELF na engleskom znači vilenjak, dok DWARF znači patuljak. Nazivi DWARF i ELF su uklopljeni kao skladni delovi epske fantastike. Ipak, kasnije je predložen odgovarajući akronim i za format DWARF — *Debugging With Arbitrary Record Formats* (debagovanje sa proizvoljnim formatima zapisa).

Iako je format *DWARF* nezavisan od formata izvršive datoteke, najčešće se koristi uz format za izvršive i povezane datoteke ELF (eng. *Executable and Linkable Format*) u okviru *UNIX*-olikih operativnih sistema, kao što su *Linux* i *macOS*. Format *DWARF* se može korisiti prilikom prevođenja programa koji su napisani u različitim višim programskim jezicima — uključujući C, C++, Fortran i mnoge druge, a karakteristike formata omogućavaju i proširenja za dodatne jezike i njihove specifične konstrukte, ukoliko je to potrebno. Za format *DWARF* je karakteristično da se metapodaci koje on definiše umeću direktno u izvršivi kôd, što čini debag verziju programa dodatno većom u odnosu na riliz verziju koja te podatke ne sadrži.

Primer 5.1.3 (Format *DWARF*) Kompajleri kao što su *GCC* i *Clang* generišu informacije u formatu *DWARF* i njih mogu da koriste debageri *GDB* i *LLDB*.

Operativni sistem *Windows* koristi sopstveni format za pomoćne informacije — *Microsoft CodeView* — koji je integrisan u Majkrosoftove razvojne alate. Ovaj format koristi se prilikom prevođenja programa koji su napisani u programskim jezicima C, C++ i jezicima *.NET* platforme. Metapodaci u formatu *CodeVeiw* se distribuiraju odvojeno od izvršivog koda (u okviru datoteka sa ekstenzijom *pdb*). Oni se koriste po potrebi — ne opterećuju izvršivi kôd ali se učitavaju u debager prilikom debagovanja.

Primer 5.1.4 (Format *Microsoft CodeView*) Kompajler *MSVC* (kompajler *Microsoft Visual Studio* C++) generiše podatke u formatu *Microsoft CodeView* i njih mogu da koriste debageri *WinDbg* i *Visual Studio Debugger*.

5.1.3 Kombinovani režimi prevođenja

Moguće je da ponašanje programa dobijenih u režimima debag i riliz bude različito, što može značajno otežati proces otklanjanja grešaka. Na primer, može se desiti da se određene greške ispoljavaju isključivo u riliz verziji, dok se u debag verziji program izvršava ispravno. Ovakva situacija može imati više uzroka.

Jedan od mogućih razloga je to što debag verzija sadrži dodatne informacije, kao i inicijalizaciju memorije koju riliz verzija ne poseduje. Time se greška može privremeno „zamaskirati“, iako je prisutna u izvornom kodu. S druge strane, uzrok može biti i greška u kompajleru, koja se manifestuje tokom optimizacije prilikom generisanja riliz verzije. Iako su ovakve greške izuzetno retke, one i dalje predstavljaju ozbiljan problem jer zahtevaju pronalaženje zaobilaznog rešenja kako bi se dobila ispravna optimizovana verzija programa.

U oba opisana slučaja, debagovanje debag verzije programa ne može otkriti uzrok problema, dok je debagovanje riliz verzije otežano zbog napostojanja veze između izvornog i izvršivog koda. Zbog toga se u savremenom razvoju softvera sve više ulaže u unapređenje mogućnosti za debagovanje optimizovanog koda, a problemu se pristupa upotrebom hibridnog režima kompilacije, u kojem se optimizacije kombinuju sa zadržavanjem debag informacija (onoliko koliko to optimizacije dozvoljavaju).

Primer 5.1.5 (Kombinovani režim prevođenja) Kombinovani režim prevođenja, za kompajler *gcc*, podrazumeva viši stepen optimizacije, na primer `-O2` kao i opciju koja uključuje informacije za debagovanje `-g`. Dodatno, uključuje se i opcija `-DNDEBUG` kako bi izvršavanje bilo više nalik izvršavanju koje se dobija sa režimom prevođenja za upotrebu.

Ovakav režim prevođenja u sistemu *CMake* može se postići postavljanjem opcije `-DCMAKE_BUILD_TYPE` na vrednost `RelWithDebInfo`.

5.1.4 Anti-debagovanje

Mogućnost debagovanja izvršive verzije programa nije uvek poželjna osobina, naročito u kontekstu zaštite intelektualne

Poboljšavanju korisničkog iskustva prilikom korišćenja debagera nad optimizovanom verzijom programa aktivno doprinose **Đorđe Todorović** & **Nikola Prica**, diplomirani master studenti Matematičkog fakulteta:

- *Debug info in optimized code - how far can we go? Improving LLVM debug info with function entry values*
  `https://www.youtube.com/watch?v=1cWAmLMF1eI`
- *Improving Debug Information in LLVM to Recover Optimized-out Function Parameters*
  `https://www.youtube.com/watch?v=ih5v65K10M8`

svojine, bilo radi zaštite od neovlašćenog kopiranja (eng. *software copy protection*) ili radi sprečavanja obrnutog inženjeringa (eng. *reverse engineering*). U takvim slučajevima, primenjuju se različite tehnike poznate pod nazivom *anti-debagovanje*. Tehnike anti-debagovanja se koriste i u malicioznim programima, gde služe kao sredstvo za izbegavanje analize i prepoznavanje od strane antivirusnih alata i sistema za detekciju pretnji.

Anti-debagovanje obuhvata implementaciju jedne ili više metoda čiji je cilj da ometaju ili potpuno onemoguće pokušaje debagovanja ciljanog procesa. Ove tehnike mogu uključivati detekciju prisustva debagera, izmenu toka izvršavanja u slučaju da je debager prisutan, korišćenje specifičnih instrukcija koje se ponašaju drugačije u prisustvu nadgledanja, kao i tehnike obfuskacije[‡].

Iako tehnički sofisticirano, anti-debagovanje uvek predstavlja balans između zaštite i funkcionalnosti. Prekomerna zaštita može negativno uticati na stabilnost, efikasnost i održivost softverskog proizvoda.

5.2 Vrste debagovanja

Debagovanje možemo podeliti na interaktivno debagovanje, udaljeno debagovanje i debagovanje nakon prekida izvršavanja programa (*post-mortem* debagovanje). Debager može da započne proces i da prati njegovo izvršavanje, može da prati izvršavanje procesa koji je već započeo sa radom kao i da analizira stanje programa nakon što je program završio sa radom.

5.2.1 Interaktivno debagovanje

Interaktivno debagovanje podrazumeva dinamičku analizu programa u realnom vremenu uz aktivno učešće programera, koji koristi debager da bi pratio i kontrolisao tok izvršavanja aplikacije. Ovaj način debagovanja omogućava dvosmernu komunikaciju sa debagerom, putem komandne linije ili grafičkog korisničkog interfejsa.

[‡] Obfuskacija ne menja funkcionalnost programa, već samo način na koji je on predstavljen i uključuje preimenovanje simbola, dodavanje beskorisnog (mrtvog) koda, dinamičko generisanje koda i šifrovanje delova koda ili podataka (kôd se generiše ili dešifruje tokom izvršavanja, umesto da bude prisutan u izvršivoj datoteci, kako bi se sprečila statička analiza koda).

Osnovni mehanizam interaktivnog debagovanja zasniva se na tačkama prekida (eng. *breakpoints*), koje omogućavaju da se izvršavanje programa automatski pauzira na odabranoj liniji koda. Kada izvršavanje programa dođe do linije koda na kojoj je postavljena tačka prekida, debager omogućava inspekciju promenljivih, sadržaja steka, registara i drugih komponenti stanja programa. Nakon analize, izvršavanje može biti nastavljeno, korak po korak (naredbe koje se nazivaju na engleskom *step* i *next*) ili do sledeće tačke prekida.

Primer 5.2.1 (Funkcija *ptrace*) Debageri su sistemski zavisni alati. Za razumevanje funkcionisanja debagera potrebno je razumeti procese i sistem prekida na odgovarajućem operativnom sistemu.

Pod operativnim sistemom *Linux*, za rad debagera od suštinske važnosti je sistemska funkcija ptrace (skraćeno od (eng. *process trace*), tj. „pratilac procesa"). Korišćenjem sistemskog poziva *ptrace*, jedan proces, *upravljački proces*, najčešće debager, može da kontroliše drugi, *ciljni proces*, i da upravlja njegovim unutrašnjim stanjem. Debageri koriste funkciju *ptrace* kako bi mogli da zaustavljaju program, da posmatraju memoriju zaustavljenog programa i da je menjaju.

Pored običnih tačaka prekida, moguće je postavljati i uslovne tačke prekida koje se aktiviraju samo kada je ispunjen određeni logički uslov. Na taj način se izvršavanje ne zaustavlja pri svakom prolazu, već samo kada je stanje sistema relevantno za otkrivanje greške.

Primer 5.2.2 (Implementacija tačaka prekida) Kada se postavi prekidna tačka u programu sa željom da se na tom mestu zaustavi program, debager zameni odgovarajuću instrukciju instrukcijom prekida, pri čemu sačuva i originalnu instrukciju. Kada se prilikom izvršavanja programa naiđe na instrukciju prekida, desi se hardverski izuzetak, operativni sistem zaustavlja rad procesa i obaveštava o tome debager.

Debager najpre proverava da li je prekid u listi očekivanih prekida koje je postavio debager (tj. da li je u pitanju namerno zaustavljanje ili greška u originalnom kodu). Ukoliko je greška u originalnom kodu, onda se dopusti da

se ta greška i izvrši.

Ukoliko je u pitanju tačka prekida koju je postavio debager, onda debager na tom mestu omogući uvid u sve vrednosti fizičkih registara procesa kao i u stanje memorije. Debager prikazuje pročitane informacije o procesu povezane sa informacijama o izvornom kodu koje generisane od strane kompajlera/linkera prilikom prevođenja programa.

Kada korisnik želi da nastavi sa izvršavanjem,

1. debager zameni instrukciju prekida originalnom instrukcijom,
2. izvrši je,
3. zameni ponovo originalnu instrukciju instrukcijom prekida,
4. prepusti dalje kontrolu programu.

Ukoliko je u pitanju uslovna prekidna tačka, debager proverava uslov i u slučaju da uslov nije ispunjen, preskače se zaustavljanje i prikaz stanja memorije već se samo nastavlja dalje sa izvršavanjem procesa. Ukoliko uslov jeste ispunjen, debager zaustavlja rad procesa i prikazuje relevantne informacije, kao i za obične tačke prekida.

Savremeni debageri omogućavaju i postavljanje *tačaka posmatranja* (eng. *watchpoint*) nad određenim lokacijama u memoriji. Kada se sadržaj označene memorijske lokacije promeni, debager automatski pauzira izvršavanje programa, čime se olakšava praćenje neželjenih ili neočekivanih promena podataka u toku rada aplikacije. Ova tehnika se naročito koristi kod analize grešaka izazvanih nepredviđenim upisima u memoriju, kao što su prekoračenje bafera (eng. *buffer overflow*) ili konkurentni pristupi deljenim resursima.

Primer 5.2.3 (Hardver — specijalni registri za debagovanje) Za efikasno funkcionisanje, debager može da koristi i direktno neke funkcionalnosti hardvera, ukoliko su dostupne. Na primer, tačke posmatranja omogućavaju praćenje vrednosti neke promenljive u memoriji, tj da li se neka vrednost u memoriji menja ili se sa nje nešto čita. Ukoliko postoji podrška hardvera (specijalni registri koji pamte i proveravaju odgovarajuće adrese), biće podignut izuzetak koji će debager da obradi. Ukoliko to nije dostupno ili se zahteva praćenje većeg broja vrednosti nego što je

to dostupno podrškom hardvera, onda debager mora da izvršava instrukciju po instrukciju i da za svaku proverava šta se dešava na traženim memorijskim lokacijama, što značajno usporava izvršavanje.

Interaktivno debagovanje omogućava precizno i fleksibilno praćenje toka izvršavanja, što je od izuzetne važnosti u složenim programima gde je ponašanje teško predvideti. Kroz kombinaciju pauziranja, ispitivanja i nastavljanja rada, programer stiče detaljan uvid u ponašanje softverskog sistema i može efikasno identifikovati greške.

Primer 5.2.4 (Interaktivno debagovanje: *gdb* iz komandne linije) Analizirajmo program *maksimum_niza.c* interaktivnim debagerom *gdb*.

```
#include <stdio.h>

int maksimum_niza(int* niz, size_t velicina_niza) {
    int max = niz[0];
    for (size_t i = 1; i < velicina_niza; i++) {
        int tekuci = niz[i];
        if (tekuci > max) {
            max = tekuci;
        }
    }
    return max;
}

int main() {
    int niz[] = {8,1,16,0,256,5};
    int max = maksimum_niza(niz, sizeof(niz)/sizeof(
    int));

    printf("maksimum = %i\n", max);
    return 0;
}
```

Najpre je potrebno prevesti program (nivo optimizacije `-O0` je podrazumevan pa se može i izostaivti) i pokrenuti debager.

```
gcc -g -O0  maksimum_niza.c  -o maksimum_niza
gdb maksimum_niza
```

Nakon toga, pokreće se debager *gdb* koji štampa svoju pozdravnu poruku. Nakon toga je dostupan prompt za komunikaciju.

```
...
Reading symbols from maksimum_niza...
(gdb)
```

Osnovna komanda je komanda `run` koja će pokrenuti i izvršiti program

```
(gdb) run
Starting program: /home/matf/Desktop/maksimum_niza
maksimum = 256
[Inferior 1 (process 42904) exited normally]
(gdb)
```

Komandom

```
break
```

možemo postaviti tačku prekida na željeni broj linije koda ili na željenu funkciju. Na primer,

```
break maksimum_niza
```

će postaviti tačku prekida na početak izvršavanja funkcije `maksimum_niza`. Naredba

```
run
```

će pokrenuti izvršavanje programa i zaustaviti se na liniji 3 (jer je tu postavljena tačka prekida, tj. tu je početak funkcije `maksimum_niza`). Komandom

```
next
```

prelazimo na narednu naredbu programa, liniju 4. Komandom

```
info locals
```

možemo videti vrednost svih lokalnih promenljivih. Komandom

```
info args
```

možemo videti vrednost argumenata funkcije.

```
(gdb) break maksimum_niza
Breakpoint 1 at 0x1169: file maksimum_niza.c, line 3.
(gdb) run
Starting program: /home/matf/Desktop/maksimum_niza

Breakpoint 1, maksimum_niza (niz=0xf0, velicina_niza
    =93824992231488) at maksimum_niza.c:3
3   int maksimum_niza(int* niz, size_t velicina_niza)
     {
(gdb) next
4       int max = niz[0];
(gdb) next
5       for (size_t i = 1; i < velicina_niza; i++) {
```

```
(gdb) info locals
i = 140737488346839
max = 8
(gdb) next
6           int tekuci = niz[i];
(gdb) info locals
tekuci = 0
i = 1
max = 8
(gdb) info args
niz = 0x7fffffffded0
velicina_niza = 6
```

Uslovne tačke prekida postavljaju se dodavanjem uslova prekida. Na primer, komandom

```
break 6 if i==3
```

prekidamo izvršavanje na liniji 6 ukoliko je vrednost promenljive `i` jednaka 3. Slično, komandom

```
watch max
```

možemo da prekinemo izvršavanje svaki put kada se promeni vrednost promenljive `max`. Komandom

```
info break
```

možemo da vidimo koje su sve prekidne tačke trenutno postavljene u programu.

```
(gdb) break 6 if i==3
Breakpoint 2 at 0x55555555518c: file maksimum_niza.c,
     line 6.
(gdb) watch max
Hardware watchpoint 3: max
(gdb) info break
Num     Type           Disp Enb Address
    What
1       breakpoint     keep y   0x0000555555555169 in
     maksimum_niza at maksimum_niza.c:3
    breakpoint already hit 1 time
2       breakpoint     keep y   0x000055555555518c in
     maksimum_niza at maksimum_niza.c:6
    stop only if i==3
3       hw watchpoint  keep y
    max
```

Komandom

```
continue
```

nastavljamo izvršavanje programa do prve naredne prekidne tačke.

```
(gdb) continue
Continuing.

Hardware watchpoint 3: max

Old value = 8
New value = 16
maksimum_niza (niz=0x7fffffffded0, velicina_niza=6)
    at maksimum_niza.c:5
5       for (size_t i = 1; i < velicina_niza; i++) {
(gdb) continue
Continuing.

Breakpoint 2, maksimum_niza (niz=0x7fffffffded0,
    velicina_niza=6) at maksimum_niza.c:6
6           int tekuci = niz[i];
(gdb) info locals
tekuci = 16
i = 3
max = 16
```

Dodatno, savremeni debageri pružaju i niz naprednih mogućnosti koje prevazilaze klasično postavljanje tačaka prekida i izvršavanje koda korak po korak. Jedna od značajnih funkcionalnosti jeste mogućnost izmene koda u toku izvršavanja (eng. *hot code replacement*), što omogućava programeru da prilagodi ili ispravi deo programa bez potpunog zaustavljanja i ponovnog pokretanja izvršavanja. Ova osobina značajno ubrzava razvoj i testiranje, naročito u kompleksnim sistemima sa dugim vremenom pokretanja. Još jedna napredna tehnika jeste debagovanje unazad (eng. *reverse debugging*), koje omogućava programeru da prati izvršavanje koda unazad, analizirajući operacije koje su dovele do određenog stanja.

5.2.2 Udaljeno debagovanje

Pored interaktivnog debagovanja koje se sprovodi na lokalnoj mašini, često se koristi i udaljeno debagovanje (eng. *remote debugging*). Ova tehnika podrazumeva interaktivno debagovanje pri čemu se aplikacija izvršava na jednom sistemu — ciljnom sistemu, dok se proces debagovanja obavlja sa drugog sistema — udaljenog sistema. Povezivanje se ostvaruje preko mreže koristeći odgovarajuće protokole i servise.

Udaljeno debagovanje omogućava da debager sa udaljenog računara upravlja izvršavanjem programa na ciljnom siste-

mu, postavlja tačke prekida, ispituje vrednosti promenljivih i dobija informacije o stanju memorije i registara ciljne aplikacije. Ova tehnika je naročito korisna u scenarijima kada ciljni sistem ima ograničene resurse i ne može lokalno izvršavati debager, što je čest slučaj kod uređaja sa ugrađenim računarom (eng. *embedded systems*), kao i prilikom razvoja sistema za kojima nije lako obezbediti direktni fizički pristup, kao što su *IoT* (eng. *internet of things*) platforme ili sistemi u produkcionom okruženju.

Primer 5.2.5 (Udaljeno debagovanje korišćenjem debagera *gdb*) Korišćenjem alata *gdb* na udaljenom sistemu i servisa *gdbserver* na ciljnom sistemu, moguće je uspostaviti vezu između razvojne stanice i ciljnog sistema, čime se omogućava upravljanje izvršavanjem programa sa udaljene lokacije. Da bi udaljeno debagovanje bilo moguće, potrebno je da ciljna i udaljena arhitektura budu iste, ili da debager na udaljenom sistemu ima podršku za arhitekturu ciljnog sistema (npr. *multiarch* podrška za debager *gdb*). Dodatno, mrežna povezanost između dva sistema treba da bude stabilna i odgovarajuće konfigurisana (na primer, često se koristi protokol *TCP/IP*).

5.2.3 Debagovanje nakon prekida izvršavanja programa

Doprinos unapređenju debagovanja nakon prekida izvršavanja programa alatom *gdb* dao je **Đorđe Todorović**, diplomirani master student Matematičkog fakultata:

Podrška za naprednu analizu promenljivih lokalnih za niti pomoću alata GNU GDB

Debagovanje nakon prekida izvršavanja programa ili *post-mortem* debagovanje je tehnika analize ponašanja programa nakon što je njegovo izvršavanje već prekinuto, najčešće usled greške kao što su pristup nevažećoj memoriji ili neobrađen izuzetak. Za razliku od interaktivnog debagovanja, koje se sprovodi tokom izvršavanja programa, *post-mortem* debagovanje se oslanja na informacije koje su zabeležene u trenutku prekida, bez mogućnosti ponovnog pokretanja aplikacije u istom stanju. *Post-mortem* debagovanje je posebno važno u produkcionim okruženjima, gde se greške moraju analizirati bez direktnog ponovnog izvršavanja programa u identičnim uslovima.

Za *post-mortem* debagovanje ključnu ulogu imaju tzv. *core dump* datoteke, koje sadrže snimak memorije procesa u trenutku njegovog pada, uključujući relevantne informacije kao što su sadržaj steka, registra, segmenta podataka i koda. Da

bi se generisala *core dump* datoteka prilikom prekida rada programa usled kritične greške, potrebna su dodatna podešavanja u okviru operativnog sistema pošto je najčešće opcija generisanja ove datoteke podrazumevano isključena.

Tehnike *post-mortem* analize uključuju:

- pregled sadržaja memorije i registara,
- rekonstrukciju steka poziva funkcija (eng. *backtrace*),
- utvrđivanje poslednjih instrukcija koje su se izvršile,
- uvid u vrednosti promenljivih u trenutku prekida,
- analizu sistemskih logova i izlaznih datoteka.

Debageri kao što su *GDB*, *LLDB* i *WinDbg* podržavaju *post-mortem* debagovanje.

Primer 5.2.6 (Datoteka *core dump*) Naredni jednostavan program izaziva grešku *Segmentation fault* zbog dereferenciranja `NULL` pokazivača u liniji 6.

```
1 // crash.c
2 #include <stdio.h>
3
4 int main() {
5     int *p = NULL;
6     *p = 42;
7     return 0;
8 }
```

Ukoliko se program prevede sa

```
1  gcc -g crash.c -o crash
```

i pokrena sa

```
1 ./crash
```

dobija se poruka

```
1 Segmentation fault (core dumped)
```

Standardna podešavanja većine *Linux* distribucija podrazumevaju da se *core dump* datoteka ne generiše, ali se to može promeniti podešavanjima sistema koja zavise od distribucije i verzije same distribucije. Pored ostalog, potrebno je podesiti da veličina *core dump* datoteke ne bude ograničena. Na primer, u okviru *Linux* distribucije *Ubuntu*, to se radi komandom `ulimit -c unlimited`. Takođe, potrebno je pokrenuti odgvarajući sistemski servis za generisanje *core dump* datoteka kao i pronaći lokaciju gde se *core dump* datoteka smešta (ili namestiti da se on smesti u isti direktorijum kao i izvršiva datoteka).

Ime *core dump* datoteke može da bude u različitim formatima. Na primer, ako se generisana datoteka zove

`core.24122.crash.1750685973`
i ako se nalazi u istom direktorijumu, onda se debagerom *gdb post-mortem* debagovanje može pokrenuti na sledeći način

```
gdb ./crash core.24122.crash.1750685973
```

Debager *gdb* će, nakon standardne uvodne poruke, odštampati sledeće

```
Reading symbols from ./crash...
[New LWP 24122]
Core was generated by `./crash'.
Program terminated with signal SIGSEGV, Segmentation
    fault.
#0  0x0000562e1631313d in main () at crash.c:6
6       *p = 42;
(gdb)
```

Ova poruka govori da je do greške došlo u liniji broj 6 programa `crash` i prikazuje nam sadržaj te linije (`*p = 42;`). Takođe, saznajemo i da je program završen sa signalom `SIGSEGV` koji odgovara grešci tipa *Segmentation fault*. Dalje u okviru prompta debagera možemo da koristimo standardne komande, na primer štampanje vrednosti promenljivih (`print p`) ili tekućih registara (`info registers`) u trenutku kada je došlo do greške.

```
(gdb) print p
$1 = (int *) 0x0
(gdb) info registers
rax            0x0                 0
rbx            0x562e16313150      94755940806992
...
```

5.3 Primeri debagera

Debagovanje se najčešće dovodi u vezu sa tradicionalnim sistemskim i imperativnim programskim jezicima kao što su C i C++. Međutim, odgovarajući alati postoje i za druge programske jezike i paradigme.

Upotreba debagera u različitim jezicima ima mnogo zajedničkih osobina: uobičajene funkcionalnosti uključuju postavljanje i uklanjanje tačaka prekida, koračanje naredbu po naredbu, ispitivanje vrednosti promenljivih i stanja memorije, kao i praćenje kontrole toka izvršavanja. Iako se sintaksa i interfejs mogu razlikovati, osnovni principi debagovanja

ostaju isti i čine sastavni deo efikasnog procesa razvoja softvera.

Primer 5.3.1 (Debager *gdb*) Debager *gdb* je jedan od najpoznatijih i najmoćnijih alat za debagovanje. On omogućava interaktivno debagovanje, udaljeno debagovanje i debagovanje nakon prekida rada programa. Dodatno, *multiarch gdb* omogućava pokretanje i analizu izvršivih datoteka namenjenih procesorskim arhitekturama koje su različite od one na kojoj se debagovanje izvršava.

Gdb je napisan u programskom jeziku C i kontinuirano se razvija kao deo GNU projekta. Podržava debagovanje programa napisanih u više programskih jezika, uključujući Ada, C, C++, Objective-C, Pascal, Fortran, Go, Rust i Java. *GDB* se može integrisati u veliki broj različitih razvojnih okruženja, kao što su *Eclipse*, *Visual Studio*, *Qt Creator* i *CLion*. Takođe, *gdb* je dostupan na velikom broju operativnih sistema, uključujući razne *Unix* i *GNU/Linux* distribucije, kao i *Microsoft Windows* platforme.

Primer 5.3.2 (Debager *LLDB*) Debager *LLDB* je savremeni debager koji se razvija kao deo projekta *LLVM*, sa ciljem da ponudi efikasnu, modularnu i proširivu alternativu tradicionalnim alatima za debagovanje. *LLDB* omogućava interaktivno debagovanje. U dizajniranju ovog debagera, fokus je bio na ostvarivanje brzine, lake nadogradivosti i mogućnosti integracije sa modernim alatima.

LLDB je implementiran u programskom jeziku *C++* i kontinuirano se razvija u okviru razvoja kompajlerske infrastrukture *LLVM*, odakle i koristi veliki broj komponenti. Podržani programski jezici uključuju *C*, *C++*, *Objective-C* i *Swift*. *LLDB* se može integrisati u različita razvojna okruženja, kao što su *Xcode*, *CLion*, *Qt creator* i *Visual Studio Code*.

Debager je prvenstveno razvijan za *Unix*-olike operativne sisteme, uključujući *macOS* i *Linux*, ali je u poslednjim verzijama proširena i podrška za *Microsoft Windows*, iako još uvek u nešto manjoj meri nego kod debagera *GDB*. *LLDB* ima poseban značaj u ekosistemu *Apple* platformi, gde predstavlja podrazumevani debager u razvoju aplikacija za *macOS*, *iOS* i srodne sisteme.

Primer 5.3.3 (Debageri *Visual Studio Debugger* i *WinDbg*) Debagere *Visual Studio Debugger* i *WinDbg* razvija kompanija *Microsoft*. Oba su namenjena za rad u okviru *Windows* operativnih sistema i pored podrške za jezike C i C++, imaju podršku i za programske jezike zasnovane na *.NET* platformi (uključujući, na primer, *C#*, *Visual Basic* i *F#*). Oba alata podržavaju interaktivno debagovanje i *post-mortem* analizu.

Visual Studio Debugger je duboko integrisan u grafičko razvojno okruženje *Visual Studio* i prvenstveno je namenjen razvoju korisničkih aplikacija, pri čemu nudi intuitivni korisnički interfejs, automatsku inspekciju vrednosti, vizuelno upravljanje tačkama prekida i druge udobnosti grafičkog korisničkog interfejsa koje ga čine pogodnim za svakodnevni razvoj aplikacija.

Debager *WinDbg* je manje poznat u širem krugu programera u poređenju sa debagerom *Visual Studio Debugger*, ali predstavlja napredniji alat sa širim spektrom mogućnosti, posebno u kontekstu sistemskog debagovanja. *WinDbg* nudi precizniju kontrolu i detaljniji uvid u sistemski kontekst, što ga čini pogodnim za analizu modula kernela, drajvera, sistemskih komponenti i složenih grešaka koje se ne mogu lako otkriti putem standardnog razvojnog okruženja. Koristi se kao samostalna aplikacija, često iz komandne linije, i zahteva viši nivo tehničkog znanja.

Primer 5.3.4 (Debageri za programski jezika Java) Za programski jezik Java dostupan je debager *jdb* koji je deo standardne Java distribucije. Debager *jdb* se može koristiti iz komandne linije na sličan način kao i *gdb*, omogućavajući interaktivno debagovanje (postavljanje tačaka prekida, koračanje kroz kôd i ispitivanje vrednosti promenljivih). Takođe, može se integrisati u razvojna okruženja poput *Eclipse* i *IntelliJ IDEA*, što omogućava grafički interfejs i lakšu upotrebu. Iako je *jdb* namenski alat za Javu, moguće je koristiti za Javu i druge debagere u kombinaciji sa specifičnim podešavanjima i prevodiocima koji generišu izvršivi kôd. Na primer, debager *gdb* može da se koristi u kombinaciji sa kompajlerom *Native Image* koji je sastavni deo kompajlerske infrstrukture *GraalVM*.

Primer 5.3.5 (Debageri za programski jezik *Python*) Za skript jezike takođe postoje odgovarajući debageri. Na primer, programski jezik *Python* ima standardni modul *pdb*, koji omogućava interaktivno debagovanje direktno iz terminala ili integraciju u razvojne alate. Modul *pdb* podržava osnovne debagerske funkcije: postavljanje tačaka prekida, koračanje kroz kôd i inspekciju stanja programa.

Primer 5.3.6 (Debageri za programski jezik *Go*) Standardno okruženje za razvoj u jeziku *Go* ne dolazi sa ugrađenim interaktivnim debagerom. Programi napisani u programskom jeziku *Go* mogu se debagovati debagerom *gdb*, ali uz ograničene mogućnosti zbog modela konkurentnosti koji jezik *Go* koristi. Najznačajniji i najrasprostranjeniji alat za debagovanje *Go* programa je *Delve* koji omogućava standardne funkcionalnosti interaktivnih debagera. *Delve* se može integrisati sa mnogim popularnim razvojnim okruženjima i editorima, uključujući *Visual Studio Code* (preko ekstenzije za *Go*) i *GoLand*, koji razvija kompanija *JetBrains*.

Primer 5.3.7 (Debageri za programski jezik *JavaScript*) Debagovanje *JavaScript* koda podržano je kroz širok spektar alata, kako integrisanih u same veb pregledače, tako i kroz zasebna razvojna okruženja i pomoćne biblioteke. Najrasprostranjeniji i najčešće korišćeni debagerski alati za *JavaScript* su ugrađeni direktno u moderne veb pregledače kao što su *Google Chrome*, *FireFox*, *Safari* i *Microsoft Edge*.

Primer 5.3.8 (Debageri za programski jezik *Haskell*) Debagovanje *Haskell* programa razlikuje se od debagovanja u imperativnim jezicima zbog njegove funkcionalne prirode i podrške lenjom izračunavanju. U ovom jeziku prioritet ima razumevanje evaluacije izraza i praćenje toka podataka kroz čiste funkcije, pa su alati za debagovanje posebno prilagođeni tim konceptima.

Najpoznatiji i najčešće korišćeni alat za debagovanje *Haskell* programa jeste *GHCi Debugger*, koji dolazi kao deo standardnog kompajlera GHCi i koji omogućava standardno interaktivno debagovanje: postavljanje tačaka prekida u

funkcijama i praćenje evaluacije izraza korak po korak.

Pored *GHCi* debagera, programeri se često oslanjaju i na alate *Haskell Tracer Hat* i *Hoed* koji se najviše koriste za *post-mortem* analizu izvršavanja *Haskell* programa.

5.4 Otvoreni problemi

Iako je debagovanje ključna tehnika za analizu izvršavanja programa, primenljivost debagera je ograničena. Jedan od najizraženijih problema upotrebe debagera javlja se kod debagovanja višenitnih aplikacija. Zbog konkurentnog izvršavanja više niti, greške mogu zavisiti od redosleda izvršavanja koji se teško reprodukuje. Debageri često ne uspevaju da precizno upravljaju svim nitima istovremeno, a samo pokretanje aplikacije u debageru može promeniti ponašanje programa i redosled izvršavanja niti, prikrivajući probleme poput trke za resursima i međusobnog blokiranja. Iako savremeni debageri podržavaju praćenje pojedinačnih niti, njihova primena u kompleksnim višenitnim sistemima ostaje nepraktična, neprecizna i često nedovoljna. Zbog toga se pronalaženje grešaka u ovakvim aplikacijama uglavnom oslanja na kombinovanje više tehnika. Tehnike uključuju dinamičke pristupe: logovanje, profajliranje, upotrebu specijalizovanih sanitajzera za otkrivanje konkurentnih grešaka ali i statičku analizu (na primer, tehniku proveravanje modela).

Protivnici debagera
Iako velika većina programera koristi debagere, postoji izvesan broj značajnih programera koji ne vole debagere i glasno se izjašnjavaju protiv njihove upotrebe:

Rob Pike, jedan od autora programskog jezika *Go*, kaže sledeće
If you dive into the bug, you tend to fix the local issue in the code, but if you think about the bug first, how the bug came to be, you often find and correct a higher-level problem in the code that will improve the design and prevent further bugs.

Linus Torvalds, kreator operativnog sistema *Linux*, ne koristi debager.

Robert C. Martin, jedan od autora agilnog programiranja, kaže sledeće
Debuggers are a wasteful timesink.

Distribuirane aplikacije predstavljaju dodatni izazov, jer njihovo izvršavanje uključuje više nezavisnih komponenti koje rade na različitim fizičkim mašinama. Tradicionalni debageri nisu dizajnirani za takav kontekst i ne omogućavaju jedinstveno praćenje toka izvršavanja u svim delovima sistema.

Debagovanje je takođe problematično u sistemima sa ugrađenim računarom i aplikacijama koje rade u realnom vremenu, gde ograničenja resursa i strogi vremenski zahtevi ne dozvoljavaju pauziranje programa ili umetanje dodatnog koda za analizu. U takvim okruženjima debager može biti nepristupačan, a njegovo korišćenje može dovesti do narušavanja funkcionalnosti sistema.

Važno ograničenje odnosi se i na debagovanje aplikacija koje su prevodene u režimu za upotrebu. Optimizacije koje vrši kompajler menjaju strukturu izvršivog koda: uklanjaju se delovi koda, funkcije se spajaju ili reorganizuju, čime se

narušava mogućnost jasnog povezivanja sa izvornim kodom što čini debagovanje značajno otežanim.

Na kraju, kod veoma velikih i kompleksnih sistema, količina podataka i broj međusobno povezanih komponenti često prevazilaze mogućnosti praćenja putem debagera. U takvim slučajevima, debagovanje postaje sporo, nepregledno i nedovoljno skalabilno.

Zbog svega navedenog, debagovanje se u praksi često kombinuje sa drugim tehnikama: logovanjem, automatskim testiranjem, statičkom analizom i specijalizovanim alatima za otkrivanje specifičnih vrsta grešaka. Razumevanje ograničenja debagovanja je ključno kako bi se odabrala odgovarajuća strategija za ispitivanje i ispravljanje softverskih problema.

5.5 Štampanje umesto debagera

Štampanje vrednosti promenljivih pomoću funkcije `print` (odnosno funkcije koja vrši štampanje) predstavlja jednu od prvih tehnika debagovanja koju većina programera usvaja. Razlog za to je jednostavnost — tehnika je intuitivna, lako primenljiva i ne zahteva poznavanje dodatnih alata.

Ova metoda podrazumeva umetanje naredbe za štampanje u ključne delove programa, kako bi se:

- prikazale trenutne vrednosti promenljivih,
- pratila kontrola toka (npr. ulazak u funkciju, izvršavanje određenih grana uslova, izlazak iz petlji),
- uočili neočekivani rezultati tokom izvršavanja.

Na primer, ako program proizvodi netačan rezultat, programer može umetnuti štampanje unutar petlje ili grane uslova kako bi ispratio kako se vrednosti promenljivih menjaju. Ovo omogućava osnovni uvid u ponašanje programa bez potrebe za korišćenjem spoljašnjih alata za debagovanje.

`Print` umesto debagera

Brian W. Kernighan, koautor knjige *Programski jezik C* i pionir u razvoju operativnog sistema *Unix*, čiji su radovi oblikovali savremeno sistemsko programiranje, tvrdi:
The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.

Guido van Rossum, autor programskog jezika *Python*, koristi poziv funkcije `print` za 90% svog debagovanja.

Primer 5.5.1 (Print za debagovanje) Funkcija `suma_niza` dopunjena je sa pozivom funkcije `printf` unutar petlje koji služi za praćenje trenutnog indeksa, vrednosti elementa niza i trenutne sume. Ovo pomaže da se uoče greške, npr. ako petlja ide van granica ili ako se neka vrednost ne sabira kako treba.

```
1 int suma_niza(int niz[], int duzina) {
```

```
 int suma = 0;
 for (int i = 0; i < duzina; i++) {
  printf("niz[%d]=%d, suma=%d\n", i, niz[i], suma);
  suma += niz[i];
 }
 return suma;
}
```

Uprkos dostupnosti savremenih i veoma moćnih alata za debagovanje, upotreba štampanja za praćenje toka izvršavanja programa i dalje je široko rasprostranjena. Jedan od osnovnih razloga za to jeste jednostavnost umetanja funkcije `print` nasuprot nepoznavanja upotrebe alata za debagovanje.

Oslanjanje na štampanje kao zamenu za debager danas se smatra prevaziđenim pristupom iz više razloga:

- Svaka izmena u vidu umetanja poziva funkcije za štampanje zahteva ponovno prevođenje programa, što je kod većih projekata vremenski zahtevno.
- Umetanje poziva funkcije za štampanje može promeniti raspored memorije programa i time prikriti ili izmeniti ponašanje greške.
- Štampanjem se ne mogu lako ispratiti svi relevantni aspekti stanja programa, kao što su sadržaji registara, vrednosti pokazivača ili stanja steka.
- Štampanje ne omogućava zaustavljanje programa, niti interaktivno ispitivanje stanja promenljivih.

Uprkos tim nedostacima, štampanje ostaje važna dopunska tehnika jer mogu postojati i opravdani razlozi za upotrebu štampanja umesto debagera. To su, na primer:

- nepostojanje debagera za određenu platformu ili arhitekturu, i
- slučajevi kada i sam debager svojim prisustvom menja ponašanje programa i time otežava detekciju greške.

U takvim situacijama, štampanje može predstavljati poslednje dostupno sredstvo za ispitivanje izvršavanja programa.

Primer 5.5.2 (Štampanje u programskom jeziku *Haskell*) Iako postoje specijalizovani alati za debagovanje *Haskell* programa, programeri često pribegavaju i osnovnim tehnikama poput umetanja funkcija za ispis (`trace`) iz paketa `Debug.Trace` kako bi pratili tok evaluacije izraza prilikom izvršavanja programa.

Uporedna analiza korišćenja debagera i štampanja data je u tabeli 5.1. Iako korisna, tehnika štampanja ne može zameniti funkcionalnosti modernih debagera i trebalo bi je koristiti kao dopunu, tj. kao pragmatičnu i ponekad neizbežnu pomoćnu tehniku, ali nikako kao osnovnu metodu za analizu ponašanja programa.

Tabela 5.1: Poređenje pristupa: `print` i debager

`Print`	**Debager**
Zahteva izmene u izvornom kodu i ponovno prevođenje	Omogućava analizu bez izmena izvornog koda
Statičan — unapred definisano šta se prikazuje	Dinamičan — moguće je menjati prikazane informacije u realnom vremenu
Može prikriti greške ili uticati na izvršavanje programa	Može prikriti greške ili uticati na izvršavanje programa, ali ređe i manje
Uvek moguće koristiti	Nekada debager nije dostupan
Pruža delimičan uvid u stanje programa	Pruža kompletan uvid u stanje programa
Jednostavno za upotrebu	Kompleksno za upotrebu za početnike
Pogodno za jednostavne analize izvršavanja programa	Pogodno za kompleksne analize izvršavanja programa

Rezime

- Debager je sistemski alat koji se koristi za praćenje izvršavanja programa u kojem je potrebno identifikovati greške.
- Rad debagera se oslanja na operativni sistem, hardver i kompajler.
- Tri osnovna režima prevođenja su režim prevođenja za upotrebu, režim prevođenja za pronalaženje grešaka i kombinovani režim.
- Tehnike anti-debagovanja se koriste radi ometanja ili otežavanja procesa debagovanja.
- Interaktivno debagovanje je osnovni pristup debagovanju koji obuhvata postavljanje tačaka prekida i tačaka posmatranja, i izvršavanje programa korak po korak.
- Udaljeno debagovanje podrazumeva interaktivno debagovanje pri čemu se apliakcija koja se debaguje izvršava

na jednom, a debager na drugom sistemu.
- Debagovanje nakon prekida izvršavanja programa omogućava uvid u stanje memorije nakon prekida izvršavanja programa u situacijama kada se greške moraju analizirati bez direktnog ponovnog izvršavanja programa u identičnim uslovima.
- Najpoznatiji debageri su debageri *gdb*, *lldb*, *Visual Studio Debugger* i *WinDbg*.
- Debagovanje ima niz ograničenja i ne može se uvek primeniti.
- Štampanje umesto debagovanja je tehnika koja se smatra prevaziđenom i koju treba koristiti samo kada ne postoji alternativa

Ispitna pitanja

32. Debagovanje. Veza izvršivog koda i debagera. Režim prevođenja za upotrebu. Primeri.
33. Debagovanje. Veza izvršivog koda i debagera. Režim prevođenja za pronalaženje grešaka. Formati za predstavljanje pomoćnih informacija. Primeri.
34. Debagovanje. Veza izvršivog koda i debagera. Kombinovani režimi prevođenja. Primeri.
35. Debagovanje. Veza izvršivog koda i debagera. Antidebagovanje.
36. Debagovanje. Vrste debagovanja. Interaktivno debagovanje. Implementacija interaktivnog debagovanja, tačaka prekida i tačaka posmatranja. Primeri.
37. Debagovanje. Vrste debagovanja. Udaljeno debagovanje. Debagovanje nakon prekida izvršavanja programa. Primeri.
38. Debagovanje. Primeri debagera.
39. Debagovanje. Otvoreni problemi.
40. Debagovanje. Štampanje umesto debagera. Primeri.

Profajliranje i dinamičko detektovanje grešaka

6

Pregled

- Šta je instrumentacija i kako se ona koristi?
- Koja je uloga profajliranja, a koja sanitiziranja u procesu razvoja softvera?
- Koje vrste profajlera postoje?
- Koji su najpoznatiji sanitajzeri i šta oni omogućavaju?

U procesu razvoja softvera, naročito složenih i zahtevnih sistema, neophodno je osigurati efikasnost i pouzdanost krajnjeg proizvoda. Za postizanje ovih ciljeva koriste se različiti alati za dinamičku analizu programa i detekciju grešaka, od kojih se izdvajaju profajleri i sanitajzeri. Ovi alati omogućavaju detaljno praćenje ponašanja softvera tokom njegovog izvršavanja, ali na drugačiji način u odnosu na debagere.

Profajleri prikupljaju informacije o performansama programa — na primer, koliko često se izvršavaju određene funkcije, koliko vremena se troši u različitim delovima koda i koliko se memorije i ostalih resursa koristi. Profajleri se koriste kada postoje poroblemi sa performansama i kada je potrebno optimizovati kôd, najčešće u kasnijim fazama razvoja, kada su osnovne funkcionalnosti sistema implementirane.

Sanitajzeri, s druge strane, fokusirani su na automatsku detekciju grešaka koje mogu dovesti do nestabilnosti, kvarova ili sigurnosnih propusta. Oni identifikuju probleme kao što su pristupi neinicijalizovanoj ili nealociranoj memoriji, curenje memorije ili greške u pristupu podacima kod višenitnih aplikacija. Sanitajzeri se mogu koristiti da olakšaju pronalaženje uzroka uočenog defekta u kodu, ali i za proveru da li kôd sadrži greške čak i onda kada nema uočenih defekata. Sanitajzeri mogu da se koriste i u ranijim fazama razvoja, odnosno nije neophodno čekati na potpunu implementaciju funkcionalnosti sistema. Naziv sanitajzer vezuje se za porodicu alata koji rade u fazi kompilacije. Pored sanitajzera, postoji i veliki broj drugih alata koji imaju za cilj otkrivanje sličnih vrsta grešaka. Većina tih alata je komercijalne prirode.

U tabeli 6.1 data je uporedna analiza profajlera i alata za dinamičku detekciju grešaka. Razumevanje rada i primene profajlera i alata ze dinamičku detekciju grešaka predstavlja

Tabela 6.1: Poređenje profajlera i detektora grešaka

Karakteristika	**Profajleri**	**Dinamička detekcija grešaka**
Cilj upotrebe	Za odabir delova koda koji treba optimizovati	Za otkrivanje grešaka u kodu
Šta meri ili prati	Vreme izvršavanja, broj poziva funkcija, memorijska potrošnja, promašaji u keš memoriji...	Neispravni pristupi memoriji, curenja memorije, pristup podacima u višenitnim aplikacijama, neinicijalizovane promenljive...
Vrsta analize	Kvantitativna: koliko se šta koristi?	Kvalitativna: da li postoje greške, koje i gde?
Faza razvoja	U kasnim fazama, kada je funkcionalnost implementirana	U ranim fazama
Primeri alata	*gprof*, *perf*, *VTune*, *Callgrind*, *Cachegrind*, *Java Flight Recorder*, *Visual Studio Profiler*, *Instruments*, *dotTrace*, *cProfile*, *JProfiler*, *YourKit*	**Sanitajzeri**: *ASan*, *MSan*, *TSan*, *UBSan*. **Alati**: *Memcheck*, *Massif*, *Helgrind*, *DRD*, *BoundsChecker*, *Purifier*, *Insure++*, *Intel Inspector*, *Go race detector*

osnovu za efikasno testiranje, analizu i unapređenje softverskih sistema.

6.1 Instrumentacija

Instrumentacija (eng. *instrumentation*) je proces dodavanja instrukcija u program kako bi se, tokom njegovog izvršavanja, pratile određene pojave i prikupljali podaci o njegovom ponašanju. Instrumentacija se koristi u jednoj vrsti profajlera kao i u sanitajzerima.

Najvažnije osobine koje dobra instrumentacija treba da zadovolji su:

1. Prikuplja samo potrebne podatke.
2. Ne utiče na funkcionalnost programa.
3. Ne usporava previše rad programa.
4. Ne povećava previše upotrebu memorije.

Prvi uslov je važan jer zahtev za previše podataka dodatno usporava program i samu njihovu obradu, dok premalo informacija može biti beznačajno. Drugi uslov ističe da ukoliko

dodata instrumentacija utiče na funkcionalnost programa onda prikupljeni podaci neće oslikavati pravi način njegovog rada.

Preterano usporavanje rada programa može da vodi do praktične neupotrebljivosti programa zbog instrumentacije. Na primer, ako izvršavanje programa traje jedan sat, a usporenje koje instrumentacija nameće izvršavanju je sto puta, onda izvršavanje instrumentovanog programa traje 100 sati što je prilično nepraktično. Slično, ako program već zahteva upotrebu velike količine memorije, onda povećenje potrošnje memorije može da utiče na to da instrumentovan program ne može da stane na uređaj na kojem treba da se izvršava. Usporavanje i povećavanje upotrebe memorije zavise od tipa aplikacije i mogu se kontrolisati izborom delova programa koji se instrumentuju.

Primer 6.1.1 (Uređaji koji rade u realnom vremenu) Usporavanje rada programa koje je posledica instrumentacije je posebno važno u kontekstu sistema koji treba da rade u realnom vremenu. Ovi sistemi često imaju vrlo stroga vremenska ograničenja koja se instrumentacijom mogu poremetiti (prekršiti) i time onemogućiti relevantnu smislenu analizu rada uređaja. Dodatno, problem mogu da budu i memorijska ograničenja uređaja jer dodatni kôd za instrumentaciju i njeno rukovanje može uvećati program tako da on ne može da stane na uređaj.

Instrumentacija se može klasifikovati prema načinu na koji se nove instrukcije dodaju u program: može je vršiti programer ili se može vršiti automatski. Programer može izvršiti instrumentaciju dodavanjem i inkrementiranjem brojača na željenim mestima u kodu. Automatsku instrumentaciju može da sprovodi:

- kompajler i/ili linker tokom prevođenja i linkovanja programa,
- specijalizovani alat na već prevedenom (izvršivom) programu i
- specijalizovani alat tokom samog izvršavanja programa.

Instrumentacija se može vršiti prilikom proizvoljne vrste prevođenja kao i nad proizvoljnom izvršivom verzijom programa. Za dinamičku analizu koja ima za cilj otkrivanje

grešaka, obično je poželjno debag prevođenje (ili verzija) programa, dok je za praćenje performansi najčešće neophodno korisiti reliz prevođenje (verziju) programa, odnosno onu verziju programa koja će biti isporučena korisniku (jer je za tu verziju ključno izmeriti performanse).

Primer 6.1.2 (Manuelna instrumentacija) Dodavanjem globalne promenljive `brojac_saberi` i njenim inkrementiranjem unutar funkcije `saberi` moguće je izbrojati pozive ove funkcije tokom izvršavanja programa. Na kraju rada programa potrebno je odštampati vrednost ove promenljive kako bi se prikazao rezultat brojanja, ili sačuvati tu vrednost u nekoj datoteci.

```
1 int brojac_saberi = 0;
2 int saberi(int a, int b) {
3   brojac_saberi++;
4   return a + b;
5 }
```

Manuelna instrumentacija se u praksi gotovo nikada ne primenjuje. Ovde se koristi kao ilustracija, jer se sličan kôd automatski ubacuje, bilo putem kompajlera, bilo putem specijalizovanih alata.

Ime alata *Valgrind* (izgovara se *Velgrind*, između a i e) potiče iz nordijske mitologije, gde *Valgrind* označava glavnu kapiju koja vodi u Valhalu (eng. *Valhalla*) — Odinovu dvoranu u koju ulaze duše palih ratnika. Odin je vrhovni bog u nordijskoj mitologiji, bog mudrosti, poezije, rata i smrti. *Valgrind* je simbolično opisan kao široka, čvrsta kapija kroz koju prolaze samo oni koji su dostojni Odinove časti i poziva.

Autori alata su ovo ime odabrali kako bi dočarali ideju ulaska u „unutrašnjost" programa: kao što *Valgrind* u mitologiji otvara put u svet hrabrih ratnika, tako i softverski *Valgrind* omogućava programerima da zavire u unutrašnje stanje svojih aplikacija, otkrivajući greške i nedostatke koje inače ne bi mogli da vide. Ovo ime ističe ulogu alata kao „čuvara kapije" između uobičajenog izvršavanja programa i njegove detaljne analize na nivou memorije i performansi.

Instrumentacija u fazi izvršavanja

Valgrind je platforma koja omogućava izvršavanje programa, njegovu instrumentaciju u fazi izvršavanja i snimanje izveštaja koji nastaju prilikom analize izvršavanja programa. *Valgrind* se koristi kao osnova za pravljenje alata za dinamičku analizu programa: profajlera i alata za dinamičku detekciju grešaka u programima.

Svi *Valgrind* alati rade na istoj osnovi koja obuhvata podršku za izvršavanje i snimanje rezultata analize, kao i interfejs ka definisanju instrumentacije. Informacije koje se emituju variraju u zavisnosti od zadate instrumentacije koja je specifična za svaki pojedinačni alat:

$$\textit{Valgrind} + \textit{Instrumentacija} = \textit{Alat za dinamičku analizu}$$

Izazovi uspešnog rada alata zasnovanog na platforimi *Valgrind* sastoje se, pre svega, u sklapanju dva procesa u jedan: program koji se analizira i program alata se sklapaju u jedan proces. Mnogi resursi se dele između ova dva programa,

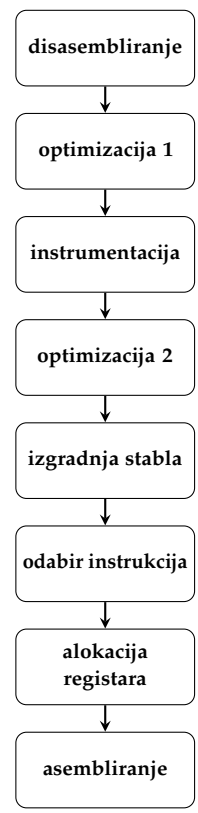
kao što su registri ili memorija, i potrebno ih je pravilno uskladiti.

Dostupnost Valgrinda
Valgrind je sistemski i arhitekturalno zavisni alat koji radi na sledećim platformama:

Linux - *x86, AMD64, ARM, ARM64, PPC32, PPC64, S390X, MIPS32, MIPS64*
Solaris - *x86, AMD64*
Android - *ARM, ARM64, x86, MIPS32*
Darwin - *x86, AMD64* (Mac OS X 10.12)

Prilikom analize izvršavanja programa nijedan deo programa koji se analizira se ne izvršava u svom izvornom obliku. *Valgrind* deli originalni kôd u sekvence koje se nazivaju osnovni blokovi. Osnovni blok je linearna sekvenca mašinskog koda, na čiji se početak dolazi nekom naredbom skoka, koja u sebi ne sadrži grananja, pozive funkcija ili skokove, i koja se završava sa skokom, pozivom funkcije ili naredbom povratka u funkciju pozivaoca. Veličina osnovnog bloka je ograničena na maksimalno šesdeset mašinskih instrukcija. *Valgrind* dopunjava mašinski kôd programa kodom koji vrši instrumentaciju. Dopunjavanje se vrši pojedinačno po osnovnim blokovima, neposredno pre prvog izvršavanja samog bloka.

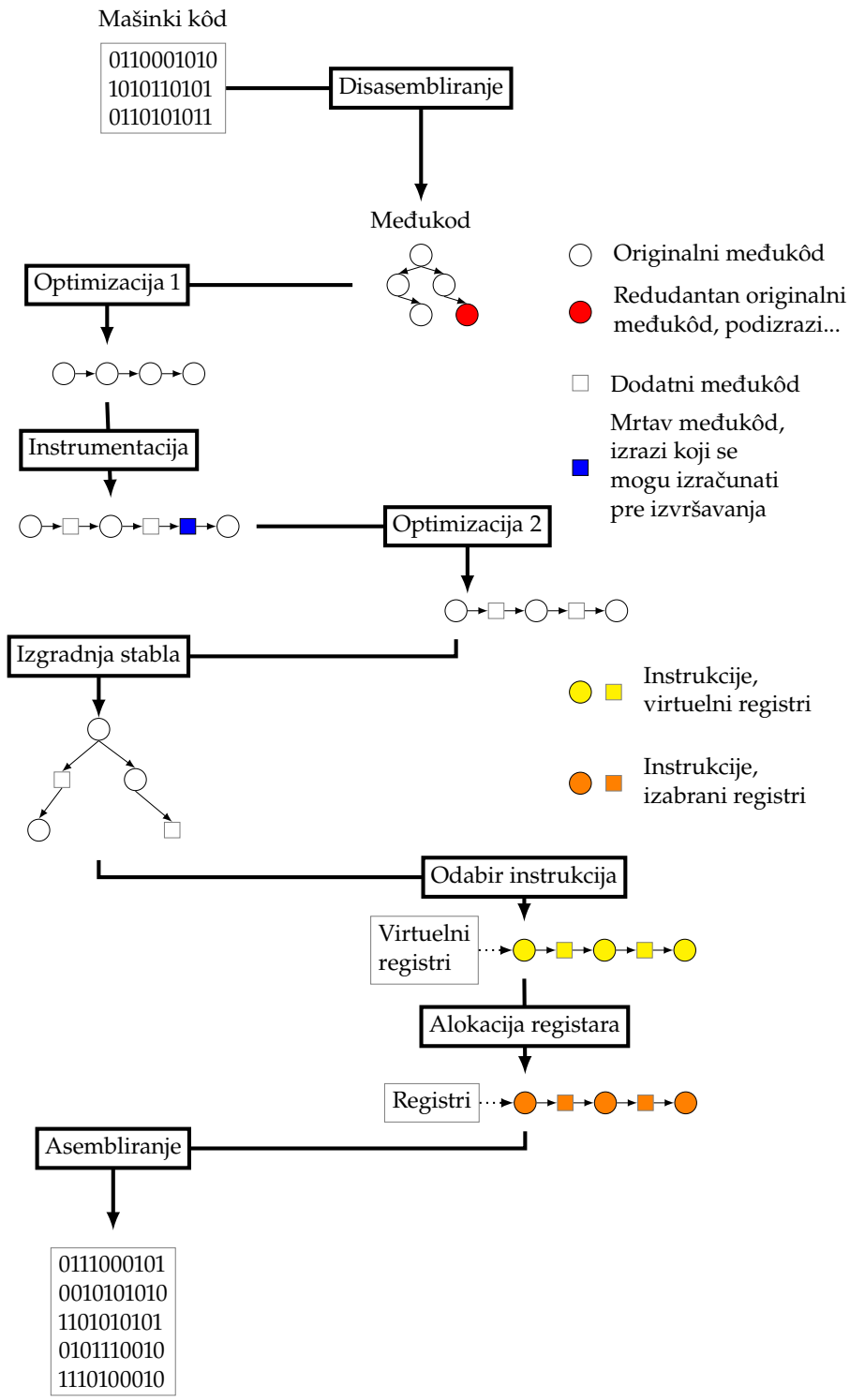
Proces transformisanja koda se sastoji iz podizanja originalnog mašinskog koda u odgovarajuću međureprezentaciju (eng. *intermediate representation*) koja se zatim instrumentuje i ponovo prevodi u novi mašinski kôd.



Izgradnja optimizovane međureprezentacije sastoji se od disasembliranja (*Valgrind* vrši podizanje mašinskog koda programa u internu međureprezentaciju) i standardnih optimizacija programskih prevodilaca (kao što su uklanjanje redudantnog koda i eliminacija zajedničkih podizraza). Disasembliranje je arhitekturalno zavisno.

Instrumentacija se vrši nad generisanom međureprezentacijom. Blok koda u međureprezentaciji se prosleđuje alatu, koji može proizvoljno da ga transformiše. Alat u zadati blok dodaje novi međukod, kojim proverava ispravnost ili prati i skuplja informacije o radu programa.

Prevođenje u izvršivi kôd obuhvata optimizacije, izgradnju stabla, odabir instrukcija, alokaciju registara i asembliranje. Druga faza optimizacije je jednostavnija od prve i uključuje samo izračunavanje vrednosti izraza koji se mogu izračunati pre faze izvršavanja i uklanjanje mrtvog koda. Zatim se od međureprezentacije kreira stablo radi lakšeg odabira instrukcija. Prilikom odabira instrukcija, koriste se virtuelni registri koji se određuju u fazi alokacije registara. Na kraju, u fazi asembliranja, kôd se prevodi u mašinski i smešta se u odgovarajući



Slika 6.1: Faze transformacije koda koje su neophodne radi instrumentacije koda u fazi izvršavanja (platforma *Valgrind*)

blok memorije. Odabir instrukcija, alokacija registara i asembliranje su arhitekturalno zavisni.

Sve faze procesa transformacije koda obavlja *Valgrind*, osim instrumentacije koju obavlja odgovarajući alat. Rezultat procesa transformacije koda se čuva u memoriji i izvršava se po potrebi. *Valgrind* troši najviše vremena na sam proces transformacije koda, kao i na pronalaženje i izvršavanje transformisanog koda. Usporenje koje *Valgrind* nameće izvršavanju programa je od 5 do 100 puta, u zavisnosti od konkretnog alata.

Najpoznatiji *Valgrind* alati su:

- ***Memcheck*** — detektor memorijskih grešaka,
- ***Massif*** — detektor memorijskih grešaka praćenjem upotrebe dinamičke memorije,
- ***Cachegrind*** — profajler keš memorije,
- ***Callgrind*** — profajler funkcija,
- ***Helgrind*** i ***DRD*** — detektori grešaka višenitnog izvršavanja.

Razvoju platforme *Valgrind* doprinela je **Aleksandra Karadžić**, diplomirani master student Matematičkog fakultata:

Alat Valgrind — Implementacija konvencije FPXX za arhitekturu MIPS

Kako *Valgrind* i njegovi alati ne analiziraju izvorni kôd već mašinski kôd programa, to znači da ih je moguće koristiti za analizu programa napisanih u bilo kom programskom jeziku. Ipak, najčešće se koriste za analizu programa napisanih u jezicima C i C++ jer su i greške koje se analiziraju i prate karakteristične za te jezike.

6.2 Profajleri

Važan deo testiranja performansi odnosi se na merenje vremenske i memorijske efikasnosti programa. Ukoliko program ne zadovoljava postavljene kriterijume performansi, neophodno je identifikovati uzroke i sprovesti odgovarajuće optimizacije. Proces optimizacije predstavlja sastavni i neizostavan deo razvoja softverskih sistema, jer obezbeđuje da implementacija ispunjava zahteve u pogledu brzine izvršavanja i potrošnje resursa.

Kako bi se precizno identifikovali delovi koda koji zahtevaju poboljšanje, u praksi se koriste specijalizovani pomoćni alati — profajleri — koji generišu detaljne informacije o ponašanju programa tokom izvršavanja. Na osnovu ovih informacija donose se odluke o prioritetima i načinima optimizacije.

Profajliranje je vid dinamičke analize programa. Profajliranjem se tokom izvršavanja programa nad unapred definisanim skupom ulaznih podataka prikupljaju detaljni podaci o ponašanju programa u realnim uslovima. Rezultat ovog procesa je skup podataka poznat kao profil programa. Profil može da obuhvata različite informacije, uključujući frekvencije poziva funkcija i izvršavanja osnovnih blokova koda,

procenat ukupnog vremena utrošenog u pojedinačnim delovima koda, informacije o korišćenju memorije — uključujući alokacije i dealokacije — zatim učestalost promašaja u kešu procesora, kao i redosled zaključavanja i otključavanja katanaca prilikom višenitnog izvršavanja.

Profajliranje može biti implementirano softverski, oslonjeno na podršku hardvera ili kao kombinacija oba pristupa. Kada se koristi hardverska podrška, profajliranje postaje efikasnije i omogućava prikupljanje šireg spektra podataka. Za određene vrste merenja, poput broja promašaja u keš memoriji ili vremena utrošenog zbog čekanja u protočnoj obradi instrukcija (eng. *pipeline stall*), hardverska podrška je neophodna.

Profajliranje se može zasnivati na instrumentaciji, tj. ubacivanju dodatnih instrukcija u kôd radi preciznog praćenja ponašanja programa tokom izvršavanja, ili na uzorkovanju (eng. *sampling*), gde se povremeno beleže karakteristični podaci o stanju sistema. Obe tehnike imaju svoje prednosti i ograničenja, a izbor tehnike, tj. odgovarajućeg alata, treba načiniti u skladu sa zahtevima za preciznošću i dozvoljenim opterećenjem sistema.

6.2.1 Upotreba profila

Profil programa se može upotrebljavati na različite načine, u zavisnosti od informacija koje on sadrži. Najčešća upotreba je identifikacija tzv. vrućih delova koda (eng. *hot spots*) — delova koda koji se najčešće izvršavaju ili koji najviše doprinose ukupnom vremenu izvršavanja — što je od suštinskog značaja za optimizaciju performansi. Takođe, profil može da posluži za procenu pokrivenosti koda datim skupom testova, čime se dobija uvid na koji način dalje proširti skup testova sa ciljem da se dobije veća pokrivenost koda. Pored toga, profil se može koristi i za detektovanje curenja memorije i raznih grešaka u kodu.

Optimizaciju na osnovu profila može da izvršiti programer, a može je automatski sprovesti i programski prevodilac. Samo programer može da suštinski izmeni algoritam koji se koristi u implementaciji ali i automatska optimizacija može da napravi važna poboljšanja u efikasnosti koda.

Automatska optimizacija može da se sprovodi u fazi kompilacije pre izvršavanja programa ili u fazi kompilacije tokom izvršavanja programa. U oba slučaja, optimizacije koje se

sprovode na osnovu profila programa nazivaju se *optimizacije vođene profilima* (eng. *profile guided optimizations*). Ove optimizacije značajno poboljšavaju performanse i prevazilaze domete standardnih kompajlerskih optimizacija.

Optimizacija u fazi kompilacije pre izvršavanja programa, da bi mogla da se sprovede, zahteva da je profil programa dostupan, što znači da toj kompilaciji prethodi jedna kompilacija i izvršavanje programa sa ciljem prikupljanja profila. Prikupljanje profila koji je potreban za optimizaciju je često veoma zahtevan posao koji značajno troši memorijske i vremenske resurse. Zbog toga se umesto prikupljanja profila, nekada koriste *statički* profajleri, odnosno *predviđanje* profila metodama mašinskog učenja na osnovu karakteristika koda. Na primer, obrada grešaka i izuzetaka odgovaraju putanjama programa koja se ređe izvršavaju u odnosu na glavne funkcionalnosti programa. Modeli mašinskog učenja mogu da nauče takve i kompleksinje zakonitosti i da na osnovu statičkih osobina koda predvide koji će se delovi programa najčešće izvršavati.

Optimizacija u fazi kompilacije tokom izvršavanja programa je karakteristična za kompilaciju u toku izvršavanja* (eng. *Just-In-Time compilation, JIT*). Kompilacija u toku izvršavanja korisiti profil koji se dobija tokom izvršavanja da bi se donele odluke o tome da se neki delovi koda kompiliraju, umesto interpretiraju, i da se dodatno optimizuju.

Razvoju statičkih profajlera modelima mašinskog učenja aktivno doprinosi **Milan Čugurović**, asistent i student doktorskih studija Matematičkog fakulteta:

GraalSP: Polyglot, efficient, and robust machine learning-based static profiler

6.2.2 Kvalitet profila

Izvršavanjem programa upravljaju konkretni ulazi i različiti ulazi daju različite rezultate profajliranja. Da bi se donosile odluke o optimizaciji na osnovu profajliranja, važno je da izvršavanje u okviru kojeg se vrši profajliranje reflektuje realnu upotrebu programa, odnosno da su skupljeni podaci na osnovu relevantnih ulaznih podataka ili da su skupljeni podaci na osnovu više različitih skupova ulaznih podataka. U suprotnom, mogu se propustiti prilike za optimizaciju programa ili se mogu doneti pogrešne odluke.

* Kompilaciju u toku izvršavanja je karakteristična za sve jezike koji se izvršavaju na Javinoj virtulenoj mašini, na primer Java, Scala i Kotlin.

Primer 6.2.1 (Propuštena prilika) Razmotrimo funkciju koja vrši obradu elemenata niza algoritmom kubne složenosti i pretpostavimo da se ta funkcija u praksi koristi nad velikim nizovima. Ukoliko se program profajlira sa ulazom koji funkciji prosleđuje niz sa malim brojem elemenata, onda se profajliranjem ne može uočiti problem sa performansama koji će nastati u praksi.

Primer 6.2.2 (Pogrešne odluke) Razmotrimo funkciju f koja ima naredbu grananja u okviru koje se u `then` grani poziva funkcija f_{then}, a u `else` grani funkcija f_{else}. Pretpostavimo da se u praksi `then` grana izvršava 20 puta češće nego `else` grana ali da su ulazni podaci za profajliranje izabrani tako da nas vode u `else` granu. Na osnovu takvog profajliranja, može da se donese pogrešan zaključak, tj. da je potrebno optimizovati funkciju f_{else}. Optimizacija te funkcije u praksi neće davati vidljive rezultate, s obzirom na to da se ona retko izvršava.

Prilikom optimizacije na osnovu dobijenih profila važno je pronaći ravnotežu između vremena utrošenog na prikupljanje podataka i koristi od dobijenih informacija. U početnim fazama optimizacije mogu se koristiti jednostavnije i manje precizne metode kako bi se identifikovali najveći problemi. Kako optimizacija napreduje, preostale prilike za unapređenje postaju suptilnije, što zahteva preciznije tehnike profajliranja.

6.2.3 Profajliranje uzimanjem uzoraka

Profajler zasnovan na uzorkovanju (eng. *sample based profiler*) periodično prekida izvršavanje programa, najčešće u fiksnim vremenskim intervalima, koristeći prekide operativnog sistema. Prilikom svakog prekida, profajler snima stek poziva (eng. *call stack*) svake niti, beležeći niz aktivnih funkcijskih poziva u tom trenutku. Prikupljeni podaci se zatim agregiraju tako da budu pogodni za analizu.

Na osnovu dovoljno velikog broja uzoraka i za dobro izabrane intervale merenja, moguće je napraviti preciznu statističku procenu koji delovi koda se najčešće izvršavaju ili gde se troši najviše vremena. Uzorkovanje pruža statističku aproksimaciju, a ne precizno merenje broja poziva ili vremena izvršavanja.

Ako se funkcija izvršava vrlo brzo i retko, moguće je da neće biti obuhvaćena uzorkovanjem.

Ovaj pristup omogućava programu da se izvršava gotovo punom brzinom, što ga čini pogodnim za velike i složene aplikacije, dugotrajne aplikacije i sisteme u realnom vremenu. Dodatno, nije potrebno ponovno prevođenje izvornog koda već se profajliranje uzimanjem uzoraka može vršiti nad proizvoljnim izvršivim datotekama (ali prisustvo debag simbola može da poboljša analizu i interpretaciju rezultata).

Najpoznatiji alati koji vrše profajliranje uzorkovanjem su *perf*, *oprofile*, *gprof* (kombinacija uzorkovanja i instrumentacije), *Visual Studio Profiler* (ima podršku i za instrumentaciono profajliranje), *Intel VTune Profiler*, *Google CPU Profiler* i *Instruments (Xcode)*.

Primer 6.2.3 (perf) Alat *perf* uveden je kao deo Linux kernela verzije 2.6.31, objavljene u septembru 2009. godine. Cilj alata bio je da zameni starije i manje fleksibilne alate za profajliranje uzorkovanjem, poput alata *oprofile*, kao i da pruži moderan interfejs za korišćenje savremenih ugrađenih hardverskih brojača za performanse (eng. *performance monitoring units*). Od tada se *perf* aktivno razvija zajedno sa *Linux* kernelom i održava ga *Linux* kernel zajednica, uz značajan doprinos od programera iz kompanija kao što su *Red Hat*, *Intel*, *Google* u *IBM*.

Perf pruža statistički pregled vremena izvršavanja i resursa koje program koristi. Zahvaljujući malim troškovima upotrebe i visokoj preciznosti, *perf* je postao standardni alat za profajliranje na *Linux* platformama. Njegova fleksibilnost omogućava primenu kako na nivou pojedinačnih procesa, tako i na nivou celog sistema, a rezultati mogu biti prikazani tekstualno ili u kombinaciji sa vizuelnim alatima za dublju analizu.

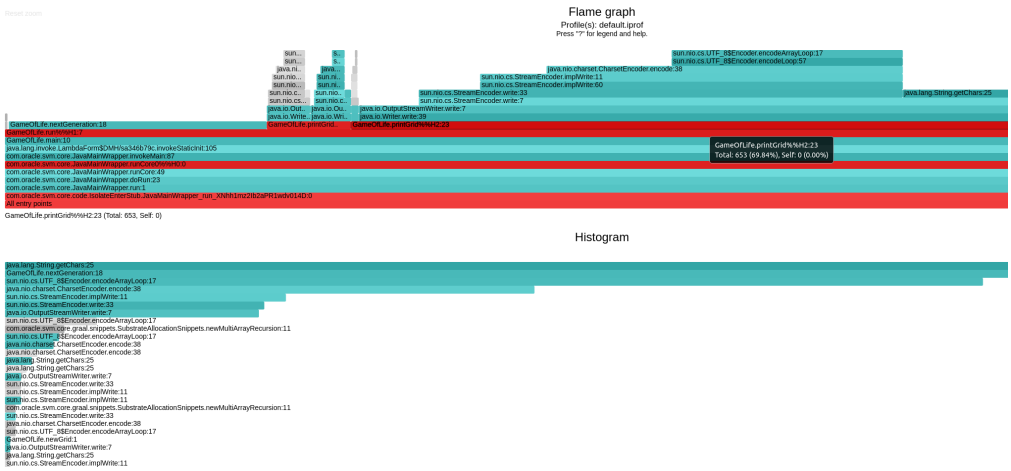
Primer 6.2.4 (Plameni graf u Javi) Plameni graf (eng. *flame graph*) je vizuelizacija koji se koristi za analizu performansi softvera, posebno za identifikovanje uskih grla i razumevanje upotrebe resursa. Plameni graf se generiše na osnovu rezultata rada profajlera.

Primer dela plamenog grafa za program *Game of Life* u Javi

čiji je deo koda prikazan u listingu 6.1 dat je na slici 6.2. Primer pokazuje da se u programu najviše vremena troši redom u metodama koje pozivaju metod *main*, koja zatim poziva metod *run* u okviru koje se najviše vremena troši u metodama *nextGeneration* i *printGrid*. Histogram u dnu slike pokazuje sortirano koliko najviše vremena se troši u pojedinačnim metodama isključujući vreme koje troše funkcije koje su iz njih pozvane. Dakle, najviše vremena se trošu u funkciji koja preuzima karaktere *getChars*, a zatim i u funkciji koja računa narednu generaciju *nextGeneration*.

Listing 6.1: Deo koda aplikacije *Game of Life*

```
1  import java.io.FileWriter;
2
3  // A simple Java program to implement Game of Life
4  // Modified from https://www.geeksforgeeks.org/
       program-for-conways-game-of-life/
5  public class GameOfLife {
6    ...
7    public static void main(String[] args) {
8        new GameOfLife().run();
9    }
10   private void run() {
11       // Initializing the grid.
12       int[][] grid = newGrid();
13       printGrid(grid);
14       for (int i = 0; i < GENERATIONS; i++) {
15           grid = nextGeneration(grid);
16           printGrid(grid);
17       }
18   }
19   ...
20   static int[][] nextGeneration(int[][] grid) {
21       int[][] future = new int[M][N];
22       // Loop through every cell
23       for (int l = 0; l < M; l++) {
24           for (int m = 0; m < N; m++) {
25               ...
26           }
27       }
28       return future;
29   }
30
31   private static void printGrid(int[][] grid) {
32     try (FileWriter myWriter = new FileWriter("f")) {
33         for (int i = 0; i < M; i++) {
34             for (int j = 0; j < N; j++) {
35                 if (grid[i][j] == 0)
```



Slika 6.2: Deo plamenog grafa za izvršavanje programa *Game of Life* u Javi

```
36                      myWriter.write(".");
37                  else
38                      myWriter.write("*");
39              }
40              myWriter.write(System.lineSeparator());
41          }
42      } catch (Exception e) {
43          throw new IllegalStateException();
44      }
45  }
46  ...
```

Plameni graf je obično interaktivan prikaz poziva metoda tokom izvršavanja programa. Svaka traka (pravougaonik) odgovara pozivu jedne metode, a svaki „plamen“ odgovara jednom nizu poziva metoda. Širina svake trake proporcionalna je vremenu koje je program proveo u tom metodu i metodama koje su iz njega pozvane.

Plameni graf na pregledan način prikazuje veliki broj uzoraka i pomaže u donošenju odluka o prioritetima u optimizaciji performansi. Vizuelni prikaz putanja poziva olakšava i otkrivanje neočekivanih redosleda izvršavanja ili neefikasnih rekurzivnih poziva.

6.2.4 Instrumentaciono profajliranje

Instrumentacijom se mogu pratiti najrazličitije karakteristike izvršavanja programa. Profili koji se koriste za određivanje

delova koda koji se najčešće izvršavaju kao i tačne udele učestalosti izvršavanja različitih delova koda, za kompajlerski zasnovane optimizacije i za utvrđivanje pokrivenosti koda testovima, dobijaju se na osnovu sledećih vrsta profajliranja:

1. profajliranje putanja,
2. profajliranje grana i
3. profajliranje blokova.

Instrumentacija se može vršiti u fazi kompilacije, kada je na raspolaganju graf kontrole toka izvršavanja programa i u tom slučaju se profajliranje sprovodi izvršavanjem instrumentovanog programa. Instrumentacije se može vršiti i u kasnijim fazama, pa čak i u fazi samog izvršavanja programa. Tada je, pored izvršive verzije programa, neophodno pokrenuti i profajler, koji dinamički ubacuje dodatne instrukcije i prati ponašanje programa u realnom vremenu.

Doprinos razvoju kompajlerski zasnovanog profajliranja u okviru kompajlerske infrastrukture LLVM dao je **Nikola Prica**, diplomirani master student Matematičkog fakultata:

Podrška za profajliranje softvera uređaja sa ugrađenim računarom

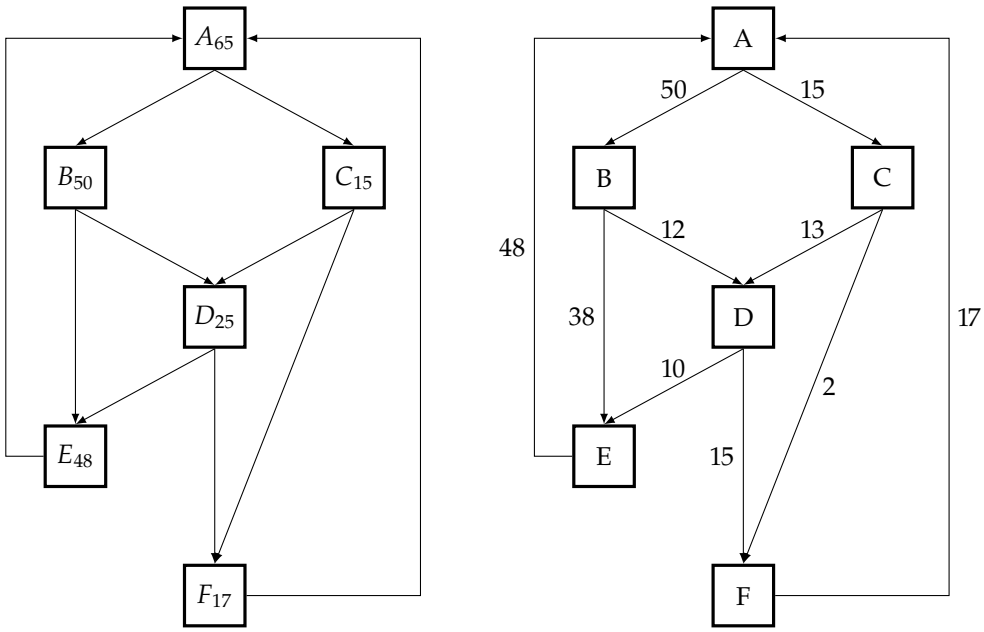
Profajliranje putanja i blokova

Profajliranje putanja (eng. *path profiling*) je složen vid profajliranja kojim se dobijaju informacije o najčešće korišćenim putanjama kroz program. Ova vrsta profila u sebi sadrži i informacije o profilima grana i blokova. Profajliranje putanja zahteva kompleksne algoritme i najviše utiče na performanse izvršavanja prilikom profajliranja.

Profajliranjem blokova (eng. *basic-block profiling*) utvrđuje se ukupan broj izvršavanja svakog bloka u programu. Blok može biti funkcija ili deo koda u kome se ne nalaze instrukcije grananja ili skokova. Naivni algoritam može se ostvariti tako što se u svaki blok umeće brojač, čime se dobijaju precizne informacije o broju izvršavanja blokova, ali se i prilično usporava sistem i opterćuje memorija. Ovaj način profajliranja ne daje informacije o tome koje su putanje kroz program najčešće kao ni koji su prelazi između blokova česti. Primer grafa kontrole toka i odgovarajućih profila blokova dat je na slici 6.3a.

Profajliranje grana

Profajliranjem grana (eng. *edge profiling*) dobijaju se podaci o prelascima koji se ostvaruju instrukcijama grananja ili skoka kojom se prebacuje tok izvršavanja programa iz jednog bloka u drugi. Profajliranjem grana mogu se dobiti i podaci koji



(a) Profili blokova su upisani uz oznaku bloka (npr. blok A je izvršen 65 puta)

(b) Profili grana su su označeni uz svaku granu (npr. grana A-B je izvršen 50 puta)

Slika 6.3: Profili dobijeni profajliranjem blokova (levo) i profajliranjem grana (desno)

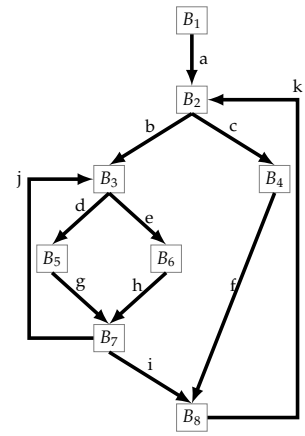
se dobijaju profajliranjem blokova. Broj izvršavanja svakog bloka može se izračunati pomoću brojača grana tako što se sumiraju sve grane koje ulaze u blok. Primer profila grana dat je na slici 6.3b.

Naivni algoritam profajliranja grana podrazumeva da se za svaku naredbu skoka umeće po jedan brojač. Međutim, takvo rešenje previše opterećuje program. Profajliranje grana, ukoliko se instrumentacija radi u fazi kompilacije, može da se implementira značajno efikasnije. Rešenje koje podrazumeva minimalni broj umetnutih brojača predložio je Donald Knut (eng. *Donald Knuth*). Osnovni oraci Knutovog algoritma profajliranja grana su:

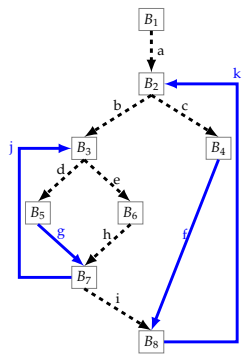
1. Konstruisati graf kontrole toka (eng. *control flow graph*) programa, u kojem svaki čvor predstavlja blok instrukcija, a grana naredbu skoka ili grananja.
2. Za dati graf kontrole toka izračunati odgovarajuće razapinjuće stablo (eng. *spanning tree*).
3. Granama koje ne pripadaju dobijenom stablu treba dodati brojač.

Knutovim algoritmom se umesto svake grane u programu

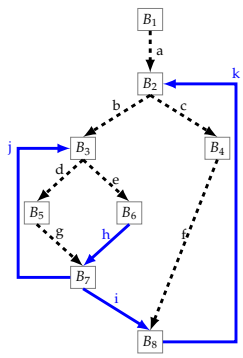
Podsetnik
Razapinjuće stablo je podskup grafa G koji sadrži sve čvorove pokrivene sa minimalnim mogućim brojem grana. Razapinjuće stablo ne sadrži cikluse i ne može biti nepovezano. Po definiciji, svaki povezan neusmeren graf G ima najmanje jedno razapinjuće stablo. Broj grana u razapinjućem stablu je $v - 1$, gde je v broj čvorova grafa.



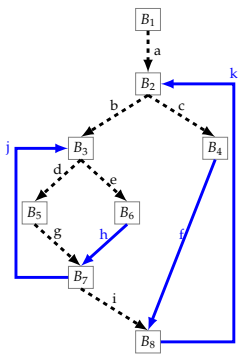
(a) Graf kontrole toka



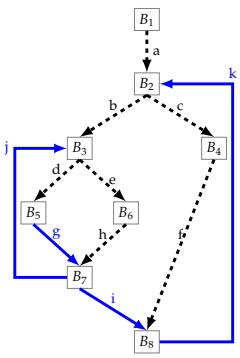
(b) Brojači: f, g, j, k



(c) Brojači: h, i, j, k



(d) Brojači: f, h, j, k



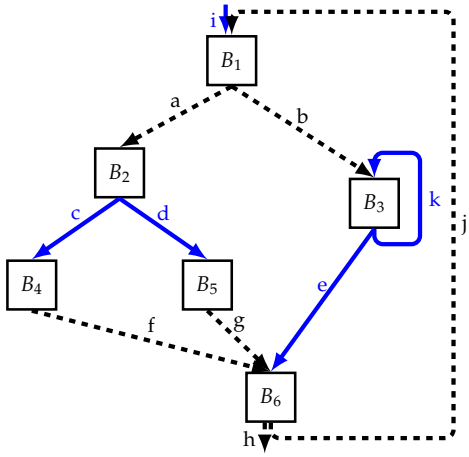
(e) Brojači: g, i, j, k

Slika 6.4: Za četiri različita razapinjuća stabla (obeležena crnom isprekidanom linijom) instrumentuju se različite grane (obeležene punom plavom linijom)

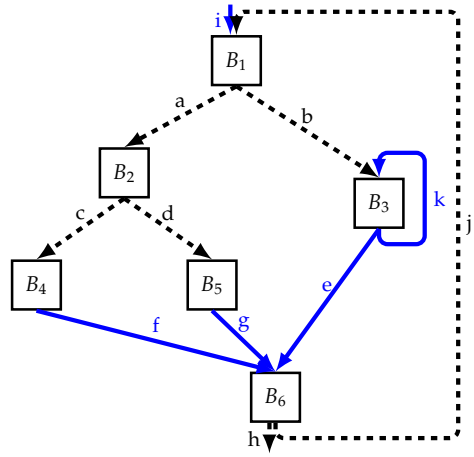
instrumentuje $e - v + 1$ grana, gde je v broj čvorova grafa, a e broj grana grafa. Knutovo rešenje može instrumentovati isti broj grana ali na različite načine, jer standardni algoritam računanja razapinjućeg stabla može vratiti različita stabla u zavisnosti od redosleda obrade grana (slika 6.4).

Broj izvršavanja grana koje ne sadrže brojač se može izračunati na osnovu profajliranih vrednosti jer je zbir izlaznih vrednosti grana iz jednog bloka jednak zbiru ulaznih vrednosti grana u taj blok. Primer izračunavanja je dat na slici 6.5.

Optimalno razapinjuće stablo grafa izvršavanja programa je ono stablo kod kojeg se grane najveći broj puta izvršavaju, jer ako se te grane ne instrumentuju, onda se izvršavanje progra-



(a) Broj izvršavanja neinstrumentovanih grana se može izračunati u narednom redosledu:
$f = c$
$g = d$
$a = c + d$
$b = e + k$
$j = a + b - i$
$h = e + f + g - j$



(b) Broj izvršavanja neinstrumentovanih grana se može izračunati u narednom redosledu:
$c = f$
$d = g$
$a = c + d$
$b = e + k$
$j = a + b - i$
$h = e + f + g - j$

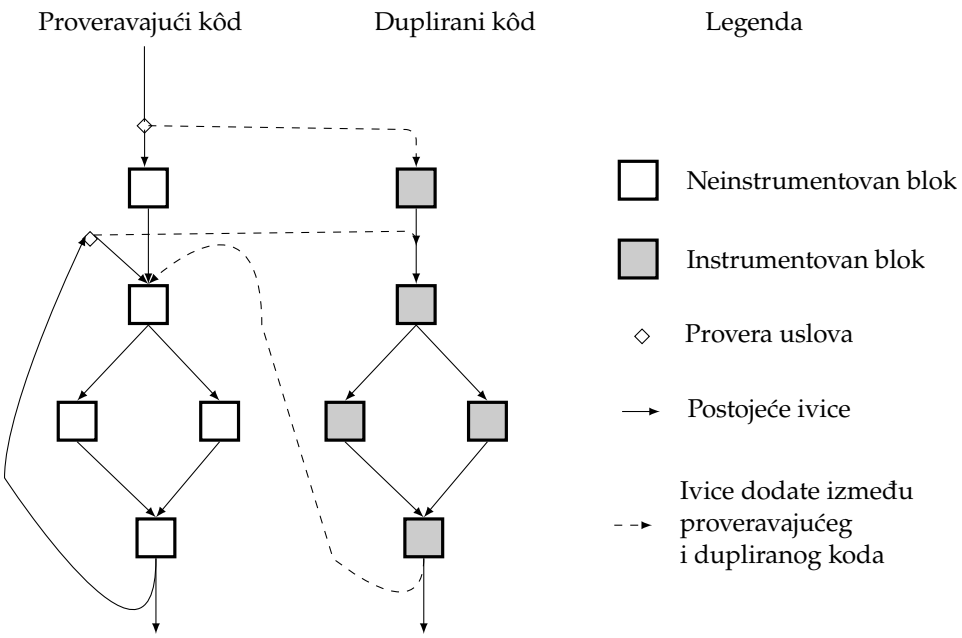
Slika 6.5: Profajliranje grana: dve mogućnosti grana za instrumentaciju. Plavom bojom i punom linijom obeležene su grane koje se instrumentuju, dok su isprekidane crne grane deo razapinjućeg stabla. Na osnovu instrumentovanih grana je moguće rekonstruisati brojeve izvršavanja preostalih grana. Razlika u izračunavanju za ova dva primera je samo u prva dva koraka.

ma najmanje opterećuje. Međutim, to je upravo informacija koja se profajliranjem i traži i zbog koje se instrumentacija i radi, tako da je veoma teško izabrati baš to stablo. Tomas Bal (eng. *Tomas Ball*) i Džejms Larus (eng. *James R. Larus*) su osmislili heuristike izbora onih grana za koje se predviđa, na osnovu statičke analize koda, da će se najmanje puta izvršiti.

Smanjivanje troškova instrumentacije

Instrumentacija omogućava dobijanje preciznog profila izvršavanja programa. Međutim, instrumentacija povlači značajno veću potrošnju vremenskih i memorijskih resursa. Zbog toga postoje razni pristupi smanjivanju troškova instrumentacije uz smanjivanje preciznosti dobijenih profila.

Jedan od načina smanjivanja vremena izvršavanja je naizmenično izvršavanje instrumentovanog i originalnog koda. U kôd se umeću provere na osnovu koji se odlučuje da li se



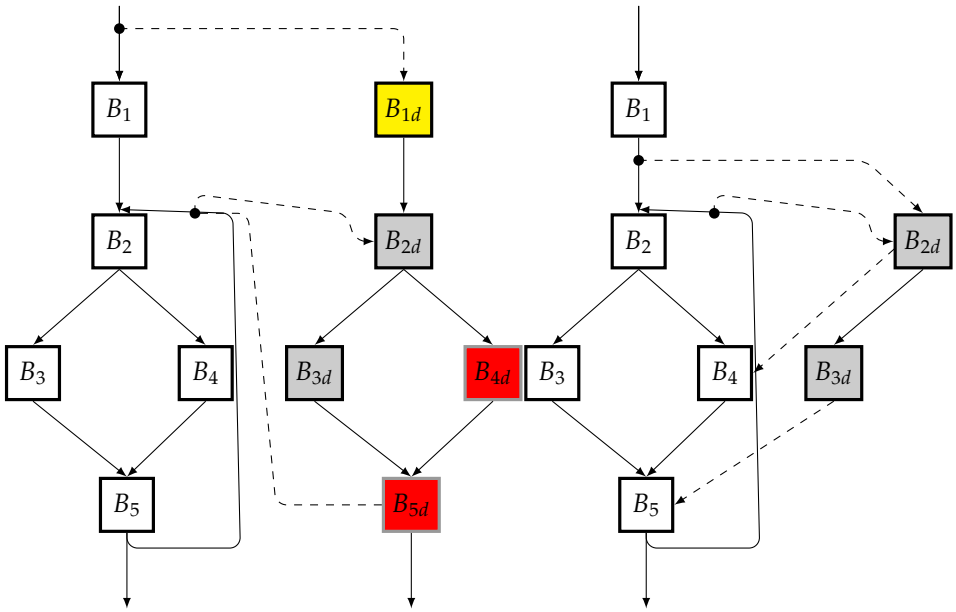
Slika 6.6: Dupliranje koda radi periodičnog izvršavanja instrumentovanog koda

izvršava instrumentovan kôd ili se ostaje u originalnom kodu. Za ovaj pristup potrebno je da postoje dve verzije koda (slika 6.6):

- instrumentovani kôd, koji se naziva još i *duplirani* kôd i
- originalni kôd, koji se naziva još *proveravajući* zato što se u njemu ispituje uslov koji, ukoliko je ispunjen, omogućava kontrolisan prelazak u duplirani kôd.

Kada izvršavanje pređe u duplirani kôd ono tu ostaje ograničeno vreme, a zatim se vraća u proveravajući kôd. Uslov prelaska iz proveravajućeg u duplirani kôd kontroliše koliko vremena će se izvršavati svaki od ova dva koda. Trenutak prelaska iz proveravajućeg u duplirani kôd se može inicirati hardverski, putem operativnog sistema ili softverski.

Prelazak iz proveravajućeg u duplirani kôd može da se vrši na osnovu fiksiranih vremenskih intervala ili na osnovu vrednosti brojača. U prvom slučaju, kada istekne odgovarajući vremenski period, sledeći prelazak u duplirajući kôd će se desiti tek kada se ponovo ispita uslov prelaska. U drugom slučaju, čuva se brojač za prelaske koji se dekrementira. Kada brojač dođe do nule, prelazi se u duplirajući kôd i resetuje se



(a) Potpuno dupliranje koda **(b)** Delimično dupliranje koda

Slika 6.7: Potpuno i delimično dupliranje koda: blokove B_1, B_4 i B_5 nije potrebno duplirati jer je za njihove duplikate moguća rekonstrukcija vrednosti na osnovu instrumentovanih blokova. Ako se izvršio dupliran blok B_{2d}, onda se tačno jednom izvršio i blok B_{1d}. Blok B_{5d} se izvršio onoliko puta koliko i blok B_{2d}, a broj izvršavanja bloka B_{4d} je jednak razlici broja izvršavanja blokova B_{2d} i B_{3d}. Slično se mogu rekonstruisati i profili grana. Zbog toga je dovoljno duplirati samo blokove B_2 i B_3.

brojač. Najčešće je profajliranje na osnovu brojača preciznije u odonosu na profajliranje na osnovu fiksiranih vremenskih intervala.

Predložen pristup smanjuje vreme izvršavanja ali povećava potrebnu memoriju kao i vreme kompilacije. Postoje proširene verzije ovog pristupa koje ne prave celu kopiju koda već samo onih delova koji su vezani za instrumentaciju, jer, za rekonstrukciju profila nije potrebno instrumentovati svaki blok ili svaku granu. Delimičnim dupliranjem može se smanjiti upotreba memorije i vreme kompilacije sa potpunim zadržavanjem preciznosti dobijenih informacija (u odnosu na potpuno dupliranje koda).

Primer 6.2.5 (Profajler keš memorije ***Cachegrind***) *Cachegrind* je alat platforme *Valgrind* namenjen softverskom profajliranju keš memorije i izvršavanja grana. On simu-

lira memorijski podsistem procesora i prati pristupe softverskom kešu tokom izvršavanja analiziranog programa. Simulacija uključuje keš prvog nivoa (*L1*) podeljen na dve nezavisne sekcije: *I1* za instrukcije i *D1* za podatke, kao i objedinjeni drugi nivo keša (*L2*), što odgovara konfiguraciji mnogih modernih procesora. Na mašinama sa tri ili više nivoa keša, *Cachegrind* modeluje prvi i poslednji nivo, odnosno sekcije *I1*, *D1* i *LL* (poslednji nivo keša).

Alat prikuplja detaljnu statistiku o pristupima memoriji i kešu, kako na nivou celog programa, tako i pojedinačno za svaku funkciju. Prate se ukupni brojevi izvršenih instrukcija, čitanja i upisa u memoriju, kao i broj promašaja prilikom čitanja i upisa u različite nivoe keš memorije — posebno za instrukcije i za podatke u prvom i poslednjem nivou keša. Na savremenim procesorima promašaj u kešu *L1* ima približnu cenu od 10 procesorskih ciklusa, dok promašaj u poslednjem nivou keša (*LL*) može koštati i do 200 ciklusa. *Cachegrind* tako omogućava identifikaciju uskih grla u performansama vezanim za efikasnost upotrebe keš memorije.

Primer 6.2.6 (Profajler funkcija ***Callgrind***) *Callgrind* je alat platforme *Valgrind* koji profajlira pozive funkcija i generiše graf poziva funkcija korisničkog programa, prikazujući odnos između pozivaoca i pozvanih funkcija, broj izvršenih instrukcija i broj poziva. U osnovnim podešavanjima prikupljeni podaci uključuju mapiranje instrukcija na linije izvornog koda, kao i statistiku poziva između funkcija. Dodatno, *Callgrind* može da analizira upotrebu keš memorije i grananja u programu na način sličan alatu *Cachegrind*.

Cachegrind meri isključivo događaje koji nastaju unutar same funkcije, takozvane *ekskluzivne* troškove. S druge strane, *Callgrind* propagira troškove funkcija kroz sve njene pozive, računajući *inkluzivne* troškove — ukupnu cenu funkcije zajedno sa svim funkcijama koje ona poziva, direktno ili indirektno. Tako je, na primer, cena funkcije `bar` uključena u cenu funkcije `foo` ako je `bar` pozvana iz `foo`.

Zahvaljujući prikazu grafa poziva, moguće je pratiti putanje od početka rada programa i identifikovati funkcije sa najvećim ukupnim troškovima. Statistika odnosa pozivaoca i pozvane funkcije posebno je korisna kada funkcija

ima više poziva iz različitih mesta u kodu, jer omogućava konteksno osetljive optimizacije.

6.3 Alati za dinamičku detekciju grešaka

Alati za dinamičku detekciju grešaka koriste instrumentaciju kako bi omogućili automatsku detekciju grešaka u kodu, najčešće u radu sa memorijom i nitima. Postali su neizostavan deo modernih razvojnih alata, jer značajno doprinose stabilnosti, bezbednosti i pouzdanosti softverskih sistema. Njihova glavna prednost je u tome što automatizuju proveru velikog broja potencijalno problematičnih situacija, čime programeru omogućavaju da se fokusira na otklanjanje grešaka umesto na njihovo traženje. Ovi alati se obično primenjuju u ranim fazama razvoja i testiranja, često i kao dodatni stepen sigurnosti da kôd ne sadrži greške.

6.3.1 Sanitajzeri

Sanitajzeri se razvijaju u okviru kompajlerskih infrastruktura. U fazi kompilacije, oni vrše automatsku instrumentaciju kao i drugačiju organizaciju memorije tako da se u fazi izvršavanja omogućava automatska detekcija grešaka. Primena sanitajzera značajno povećava vreme izvršavanja programa i potrošnju memorije, zbog čega se obično koriste samo u testnom, a ne u produkcionom okruženju.

Najpoznatiji sanitajzeri su adresni sanitajzer *AddressSanitizer*, sanitajzer memorije *MemorySanitizer*, sanitajzer niti *ThreadSanitizer* i sanitajzer nedefinisanog ponašanaj *UndefinedBehaviorSanitizer*. U tabeli 6.2 dat je uporedni pregled karakteristika ovih sanitajzera, dok su detalji za svaki sanitajzer dati u nastavku.

Adresni sanitajzer

Adresni sanitajzer *AddressSanitizer (ASan)* je jedan od najčešće korišćenih sanitajzera, namenjen dinamičkoj detekciji grešaka u upravljanju memorijom. On instrumentuje program tokom kompilacije, ubacujući dodatne provere pri svakom pristupu memoriji, i preuređuje način organizacije memorijskog

Tabela 6.2: Poređenje sanitajzera

Sanitajzer	Otkriva greške	Podržani kompajleri	Uticaj na performanse
AddressSanitizer (ASan)	Pristup memoriji izvan granica, korišćenje nakon oslobađanja, dvostruko oslobađanje, curenje memorije	Clang/LLVM, GCC, XCode, MSVC	Umeren
MemorySanitizer (MSan)	Korišćenje neinicijalizovanih vrednosti	Clang/LLVM	Visok
ThreadSanitizer (TSan)	Nesinhronizovan pristup podacima	Clang/LLVM, GCC	Veoma visok
UndefinedBehaviorSanitizer (UBSan)	Nedefinisano ponašanje prema standardu jezika	Clang/LLVM, GCC	Mali do umeren

prostora tokom izvršavanja kako bi omogućio precizno otkrivanje čitavog spektra grešaka. *AddressSantizer* je dostupan u okviru kompajlera *GCC*, *Clang/LLVM*, *XCode* i *MSVC*.

AddressSanitizer detektuje nepravilne pristupe memoriji kao što su čitanje ili pisanje izvan granica alocirane memorije, korišćenje memorije nakon njenog oslobađanja, dvostruko oslobađanje memorije i curenje memorije. Ove greške su uobičajene u programima napisanima u jezicima poput C i C++, a često su uzrok padova programa, oštećenja podataka ili sigurnosnih propusta.

Osnovni princip rada alata *ASan* zasniva se na instrumentaciji svih pristupa memoriji i uvođenju provera validnosti na svaku operaciju čitanja i pisanja. Pored toga, alokacije memorije proširuju se dodatnim tzv. „crvenim zonama" (eng. *red zones*), koje okružuju svaki blok memorije i služe za otkrivanje prekoračenja granica. Status svakog bajta memorije čuva se u posebnoj dodatnoj memoriji (eng. *shadow memory*), koja omogućava efikasnu detekciju grešaka.

Glavne prednosti upotrebe alata *ASan* su njegova pouzdanost i jednostavnost upotrebe. Program je potrebno prevesti uz dodatak odgovarajuće opcije pri kompilaciji kako bi proveravanje bilo omogućeno. Greške u pristupu memoriji otkrivaju se odmah prilikom prvog nepravilnog pristupa. Zahvaljujući tome, *ASan* je postao standardni deo alata za razvoj i testiranje modernih softverskih sistema.

Sa druge strane, njegova primena nosi i određene probleme. Programi instrumentovani ovim sanitajzerom zahtevaju značajno više memorije — često dva do tri puta više od uobičajenog — i značajno sporije se izvršavaju, takođe dva do tri puta u poređenju sa neinstrumentovanom verzijama.

Memorijski sanitajzer

Memorijski sanitajzer *MemorySanitizer (MSan)* je specijalizovan za otkrivanje korišćenja promenljivih kojima nisu inicijalizovane vrednosti[†] u programima napisanima u jezicima poput C i C++. Korišćenje neinicijalizovanih podataka može dovesti do nepredvidljivog ponašanja programa, uključujući pogrešne rezultate izračunavanja, oštećenja podataka ili sigurnosne ranjivosti. Takve greške su često teške za detekciju, jer se program može činiti ispravnim u nekim slučajevima, dok u drugim generiše neočekivane rezultate. *MSan* se razvija u okviru kompajlerske infrastrukture *LLVM* i može se koristiti kao opcija kompajlera *Clang/LLVM*.

MSan instrumentuje program tokom kompilacije, ubacujući dodatne provere koje prate inicijalizovanost svake promenljive u memoriji. On održava poseban deo memorije (eng. *shadow memory*) u kojoj pamti status svakog bajta korisničke memorije, označavajući da li je njegova vrednost inicijalizovana. Svaki pristup memoriji tokom izvršavanja se dodatno proverava korišćenjem ove memorije, i čim se otkrije čitanje neinicijalizovane vrednosti, program prekida izvršavanje i prijavljuje grešku sa tačnom lokacijom.

Prednost *MSan*-a je njegova sposobnost da pouzdano i precizno otkrije ovu klasu grešaka koja je inače izuzetno teško uočljiva običnim testiranjem ili debagovanjem. Međutim, ovaj sanitajzer ima i određena ograničenja: njegova primena značajno povećava potrošnju memorije i usporava izvršavanje programa, često još više nego *ASan*. Dodatno, *MSan* zahteva da sve biblioteke sa kojima se program linkuje budu takođe kompajlirane sa podrškom za ovaj sanitajzer, jer u suprotnom može generisati lažno pozitivne rezultate.

[†] Neke jednostavne slučajeve korišćenja neinicijalizovanih promenljivih može da prijavi kompajler u fazi prevođenja programa.

Sanitajzer niti

Sanitajzer niti *ThreadSanitizer (TSan)* je namenjen otkrivanju grešaka u višenitnim programima, odnosno detekciji nesinhronizovanog pristupa podacima (eng. *data race*). Greške izazvane nekorektnim pristupom deljenim podacima iz različitih niti bez odgovarajuće sinhronizacije spadaju među najteže za pronalaženje i reprodukovanje, jer se često ispoljavaju samo u specifičnim, teško ponovljivim scenarijima. *TSan* je dostupan u okviru kompajlera *GCC* i *Clang/LLVM*. Sastoji se od modula za instrumentaciju u fazi kompilacije i od posebne biblioteke koje se koristi u fazi izvršavanja (eng. *run-time library*).

TSan instrumentuje program tokom kompilacije i prati sve pristupe deljenim promenljivama tokom izvršavanja. Pomoću dodatnih provera u kodu vodi evidenciju o tome koje niti pristupaju kojim memorijskim lokacijama i na koji način, kao i o sinhronizacionim primitivima koje se koriste. Kada otkrije da dve ili više niti pristupaju istoj promenljivoj, a bar jedna od njih vrši upis, bez adekvatne sinhronizacije, sanitajzer prijavljuje grešku, navodeći tačne lokacije problematičnih pristupa.

Osnovna prednost ovog sanitajzera je u tome što otkriva nesinhronizovan pristup podacima, grešku koja je veoma opasna i teška za otkrivanje. Međutim, programi instrumentovani ovim sanitajzerom izvršavaju se znatno sporije, a potrošnja memorije je značajno veća u poređenju sa neinstrumentovanom verzijom. Tipično uspori izvršavanje od 5 do 15 puta, a memorijsko dodatno opterećenje je od 5 do 10 puta.

Sanitajzer nedefinisanog ponašanja

Sanitajzer nedefinisanog ponašanja *UndefinedBehaviorSanitizer (UBSan)* namenjen je otkrivanju ponašanja koja su nedozvoljena ili nedefinisana prema standardu programskog jezika. Nedefinisana ponašanja u jezicima kao što su C i C++ često su izvor suptilnih grešaka koje mogu dovesti do ozbiljnih problema uključujući padove programa i oštećenja podataka. *UndefinedBehaviorSanitizer* je dostupan u okviru kompajlera *GCC* i *Clang/LLVM*.

UBSan instrumentuje kôd tokom kompilacije i dodaje provere za razne vrste operacija koje mogu izazvati nedefinisano

ponašanje. Među najčešćim problemima koje *UBSan* detektuje su:

- prekoračenje opsega celobrojnih tipova (eng. *integer overflow*),
- konverzije između tipova koje nisu dozvoljene (koje vode prekoračenju),
- pristup nepravilno poravnatim memorijskim lokacijama (eng. *unaligned access*),
- dereferenciranje *null* pokazivača,
- operacija šiftovanja koja vodi do prekoračenja tipa.

UBSan omogućava programeru da identifikuje i otkloni greške koje često na određenim arhitekturama i u određenim uslovima prolaze bez vidljivih posledica. Na taj način, *UBSan* doprinosi pisanju prenosivijeg i pouzdanijeg koda, koji neće zavisiti od specifičnih karakteristika platforme. Sa druge strane, *UBSan* ima minimalni uticaj na performanse u poređenju sa drugim sanitajzerima. Preporučuje se njegovo korišćenje zajedno sa drugim sanitajzerima kako bi se obuhvatio što širi spektar grešaka.

6.3.2 Alati platforme *Valgrind*

Većina alata za dinamičku detekciju grešaka je komercijalne prirode i zatvorenog koda. Platforma *Valgrind* i njeni alati su besplatni i otvorenog koda.

Alati za dinamičku detekciju grešaka platforme *Valgrind* otkrivaju slične vrste grešaka kao sanitajzeri, ali, za razliku od sanitajzera, instrumentaciju rade u fazi izvršavanja. Zbog toga su performanse ovih alata lošije od performansi sanitajzera. Detektori grešaka višenitnog izvršavanja *Helgrind* i *DRD* pokrivaju značajno veći spektar grešaka u odnosu na sanitajzere niti.

Primer 6.3.1 (Detektor memorijskih grešaka ***Memcheck***) *Memcheck* je najpoznatiji i najčešće korišćeni alat platforme *Valgrind*. Njegova osnovna uloga je detekcija memorijskih grešaka u korisničkom programu. Program koji se izvršava pod kontrolom alata *Memcheck* u proseku je od dvadeset do sto puta sporiji u odnosu na samostalno izvršavanje, što je posledica dinamičke transformacije koda. Izlaz programa dopunjen je izveštajima koje generiše sam *Memcheck* i koji

se ispisuju na standardni izlaz za greške.

Za programe pisane u jezicima C i C++ *Memcheck* otkriva najčešće greške u radu sa memorijom, kao što su:

- upisivanje podataka van opsega hipa ili steka,
- pristupanje memoriji koja je već oslobođena,
- neispravno oslobađanje memorije, uključujući duplo oslobađanje ili neupareno korišćenje funkcija `malloc/new/new[]` i `free/delete/delete[]`,
- curenje memorije,
- korišćenje neinicijalizovanih vrednosti ili vrednosti izvedenih iz drugih neinicijalizovanih podataka,
- preklapanje parametara prosleđenih funkcijama, na primer preklapanje pokazivača `src` i `dst` kod funkcije `memcpy`.

Primer 6.3.2 (Detektor memorijskih grešaka ***Massif***) *Massif* je alat platforme *Valgrind* namenjen analizi korišćenja memorije u hip segmentu korisničkog programa. Postoje određeni scenariji curenja memorije koji ne spadaju u klasične slučajeve i koje *Memcheck* ne može da detektuje. Do ovakvih problema dolazi kada memorija formalno nije izgubljena — pokazivač na nju i dalje postoji — ali se više nikada ne koristi tokom izvršavanja. Programi koji imaju ovakvu vrstu „prikrivenog“ curenja memorije bespotrebno zauzimaju dodatne resurse. *Massif* pomaže upravo u identifikaciji takvih slučajeva. *Massif* pruža detaljne informacije o tome koji delovi programa su odgovorni za alokaciju memorije, olakšavajući pronalaženje uzroka neefikasnog upravljanja memorijom.

Primer 6.3.3 (Detektori grešaka višenitnog izvršavanja ***Helgrind*** i ***DRD***) *Helgrind* je alat namenjen otkrivanju grešaka u sinhronizaciji pri korišćenju *POSIX* modela niti. *DRD* je takođe alat za detekciju grešaka u višenitnim programima, namenjen programima koji koriste niti u skladu sa *POSIX* standardom ili konceptima koji su na njemu zasnovani. Iako oba alata imaju sličnu namenu i značajan broj preklapajućih detekcija, koriste različite algoritme za analizu izvršavanja i otkrivaju delimično različite tipove grešaka.

Greške u višenitnom izvršavanju koje alati detektuju uključuju mrtvo blokiranje (eng. *deadlock*) kao posledica pogrešnog redosleda zaključavanja i pristup zajedničkoj memoriji bez adekvatne sinhronizacije i zaključavanja. Alat *DRD* može da detktuje i zadržavanje katanca (eng. *lock-holding*) i lažno deljenje (eng. *false sharing*). Greške u korišćenju *POSIX* niti koje ovi alati mogu da otkriju su:

pogrešno otključavanje muteksa — kada je muteks nevažeći, nije prethodno zaključan ili je zaključan od strane druge niti,

nepravilno rukovanje zaključanim muteksom — uništavanje nevažećeg ili zaključanog muteksa, kao i dealokacija memorije koja sadrži zaključan muteks,

pogrešno korišćenje funkcije `pthread_cond_wait` — prosleđivanje nezaključanog, nevažećeg ili muteksa koji je zaključala druga nit,

greške u radu sa barijerama `pthread_barrier` — nevažeća ili dvostruka inicijalizacija, kao i čekanje na objekat koji nikada nije inicijalizovan.

Rezime

- Profajleri i sanitajzeri su alati koji vrše dinamičku analizu programa.
- Instrumentacija je proces dodavanja instrukcija u program kako bi se, tokom njegovog izvršavanja, pratile određene pojave i prikupljali podaci o njegovom ponašanju.
- Instrumentacijom se značajno utiče na performanse instrumentovanog programa i postoje različiti algoritmi koji imaju za cilj samnjivanje troškova instrumentacije.
- Profajliranje može biti implementirano softverski, oslonjeno na podršku hardvera ili kao kombinacija oba pristupa.
- Profil programa se može upotrebljavati za optimizacije koje sprovodi programer ili za automatsku optimizaciju koda koju sprovodi kompajler.
- Da bi se donosile ispravne odluke o optimizaciji na osnovu profajliranja, važno je da izvršavanje u okviru kojeg se vrši profajliranje reflektuje realnu upotrebu programa.
- Profajliranje može biti profajliranje uzimanjem uzoraka ili profajliranje zasnovano na instrumentaciji.

- Profajleri zasnovani na instrumentaciji daju preciznije informacije ali su vremenski i memorijski zahtevniji.
- *Valgrind* je platforma koja se koristi kao osnova za pravljenje alata za dinamičku analizu programa.
- Najpoznatiji sanitajzeri su adresni sanitajzer, memorijski sanitajzer sanitajzer niti i sanitajzer nedefinisanog ponašanja.

Ispitna pitanja

41. Instrumentacija.
42. Profajliranje. Upotreba profila.
43. Profajliranje. Kvalitet profila.
44. Profajliranje. Profajliranje uzimanjem uzoraka. Primeri.
45. Profajliranje. Instrumentaciono profajliranje. Profajliranje putanja, grana i blokova.
46. Profajliranje. Instrumentaciono profajliranje. Smanjivanje troškova instrumentacije.
47. Profajliranje. Instrumentaciono profajliranje. Instrumentacija u fazi izvršavanja. *Valgrind* i faze transformacije koda.
48. Profajliranje. Instrumentaciono profajliranje. *Valgrind* i najpoznatiji *Valgrind* alati. *Cachegrind* i *Callgrind*.
49. Dinamička detekcija grešaka. Sanitajzeri. Adresni i memorijski sanitajzer.
50. Dinamička detekcija grešaka. Sanitajzeri. Sanitajzer niti i nedefinisanog ponašanja.
51. Dinamička detekcija grešaka. *Valgrind* i najpoznatiji *Valgrind* alati. *Memcheck* i *Massif*.
52. Dinamička detekcija grešaka. *Valgrind* i najpoznatiji *Valgrind* alati. *Hellgrind* i *DRD*.

Statička verifikacija softvera

Semantika programskih jezika

7

Pregled

-
-
-
-
-

Pregledi koda 8

Pregled

- Ko pregleda kôd i zašto?
- Koje vrste grešaka se traže prilikom pregledanja koda?
- Kako efikasno pregledati kôd?
- Koji alati se upotrebljavaju za preglede koda?

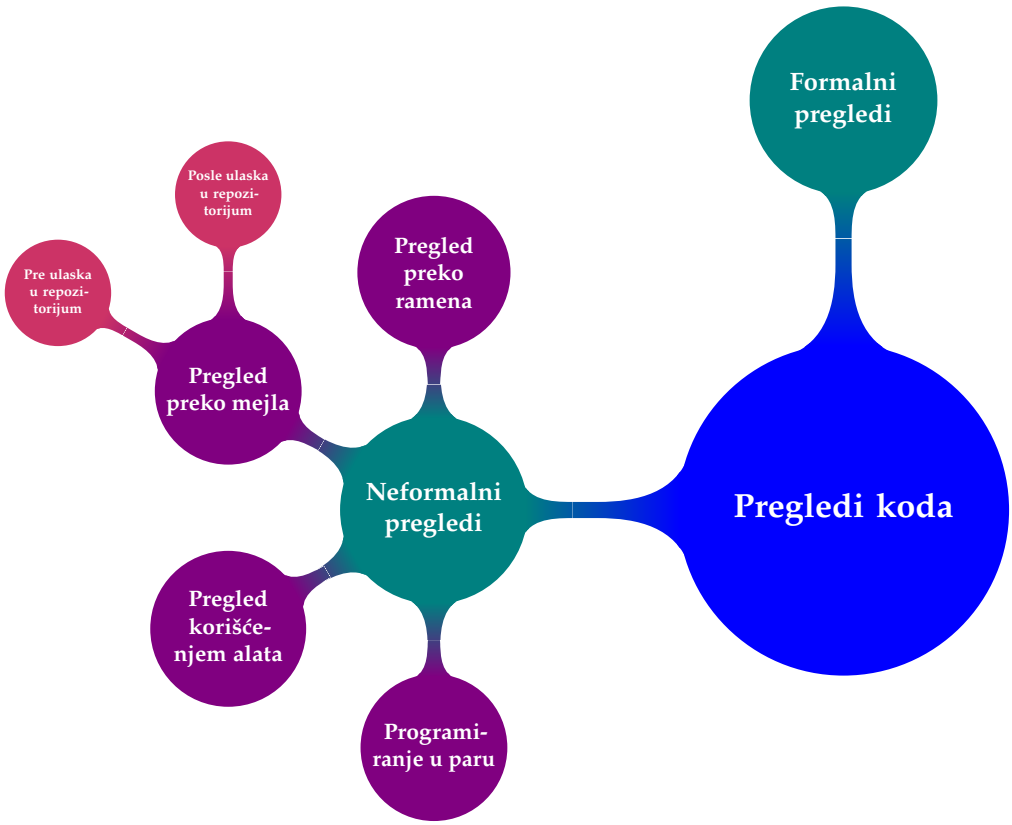
Pregledi koda (eng. *code reviews*) podrazumevaju ljudsku kontrolu koda, najčešće pre nego što se kôd uključi u glavni repozitorijum*. Pregledima se mogu uočiti različite vrste grešaka i nepravilnosti. Njihov osnovni cilj je unapređenje kvaliteta koda, kako kroz smanjenje broja grešaka, tako i kroz proveru usklađenosti sa definisanim konvencijama i pravilima pisanja programa, kao i standardima dokumentovanja. Oni omogućavaju kontrolu onih aspekata koje automatsko testiranje ne može da pokrije i sprečavaju loše odluke i neodgovarajuća rešenja koja mogu da kompromituju osnovnu liniju razvoja.

Pregledi koda doprinose i ostvarivanju drugih važnih ciljeva, poput prenosa znanja unutar tima i unapređenja komunikacije među programerima. U skladu sa nivoom formalnosti, pregledi mogu da budu više ili manje formalni. Formalni pregledi podrazumevaju anažovanje većeg broja ljudi i organizaciju niza formalnih sastanaka. Neformalni pregledi obuhvataju pregled „preko ramena", pregled putem mejla, pregled korišćenjem specijalizovanih alata i programiranje u paru. Podela pregleda po formalnosti prikazana je na slici 8.1.

8.1 Ciljevi pregleda koda

Prilikom pregleda koda procenjuju se različiti aspekti njegovog kvaliteta, u skladu sa ciljevima pregleda i pravilima organizacije. Osnovna svrha je da se utvrdi da li kôd ispravno implementira predviđenu funkcionalnost, bez očiglednih logičkih grešaka, i da li su svi zahtevi iz specifikacije pokriveni

* Podrazumeva se da softver razvija korišćenjem sistema za kontrolu verzija i da lokalne promene koje prave programeri postaju vidljive ostalim programerima tek kada se razvijeni kôd uključi u zajednički repozitorijum.



Slika 8.1: Osnovne vrste pregleda koda

i u potpunosti realizovani. Proverava se i prisustvo potencijalnih bezbednosnih ranjivosti, poput neproverenih ulaza ili neinicijalizovanih promenljivih, kao i eventualne neefikasnosti koje bi mogle negativno uticati na performanse sistema. Pažnja se posvećuje i proveri testova: da li novododati testovi adekvatno pokrivaju izmene i da li postojeće testove treba prilagoditi kako bi odražavali nove promene.

Jedan od najbitnijih aspekata pregleda koda je i vrednovanje kvaliteta rešenja sa aspekta dizajna — da li je moguće postići jednostavnije, održivije ili elegantnije rešenje, na primer primenom polimorfizma umesto niza uslovnih izraza ili korišćenjem postojećih implementacija i bibliotečkih funkcija umesto pisanja novih. Pregledi koda često uključuju i proveru usklađenosti sa arhitekturom sistema i razmatranje potencijalnih uticaja promene na ostatak projekta.

Dodatno, pregledi obuhvataju proveru usklađenosti sa konvencijama kodiranja i dogovorenim stilom pisanja, uključuju-

ći imenovanje, uvlačenje i komentarisanje. Posebno se ceni čitljivost i održivost koda, odnosno njegova jasnoća, razumljivost i stepen dokumentovanosti koji omogućava jednostavno održavanje i dalji razvoj.

Primer 8.1.1 Razmotrimo naredni kôd.

```
    int x, y, Q, p, A, b, c, D, d;
    scanf("%d%d%d%d", &x, &y, &p, &d);
    Q = 2*(x + y);
    c = x*y;
    A = 2*(p+b);
    D=p*b;
```

Pregledom su mogu dati sledeće preporuke.

- Neophodno je preimenovati promenljive, jer su nazivi nekonzistentni u pogledu upotrebe velikih i malih slova i nedovoljno deskriptivni, čime se otežava razumevanje njihove namene.
- Potrebno je obezbediti konzistentno formatiranje izraza, budući da u nekim slučajevima postoji razmak oko operatora, dok u drugim ne.
- Koristi se neinicijalizovana promenljiva `b`. Verovatno je na tom mestu trebalo koristiti promenljivu `d`.
- Pošto se to izračunavanje površine i obima pravougaonika ponavlja, preporučuje se izdvajanje tih računanja u posebne funkcije, čime bi se poboljšala čitljivost i modularnost koda. (Iz šireg konteksta, pregledač može da razume da se kôd odnosi na izračunavanje površine i obima pravougaonika, što se zbog lošeg kvaliteta izdvojenog koda ne može videti.)

U nastavku je kôd izmenjen u skladu sa datim preporukama. Pored modularnosti, čitljivosti, razumljivosti i moguće ponovne upotrebe izdvojenih funkcija narednog koda, on omogućava i produbljivanje modularizacije i uvođenje apstrakcija ukoliko su potrebne.

```
  int obim_pravougaonika(int a, int b) {
    return 2*(a+b);
  }

  int povrsina_pravougaonika(int a, int b) {
    return a*b;
  }
  ...
  int a1, b1, obim1, povrsina1;
```

```
  int a2, b2, obim2, povrsina2;
  scanf("%d%d", &a1, &b1);
  obim1 = obim_pravougaonika(a1, b1);
  povrsina1 = povrsina_pravougaonika(a1, b1);
  scanf("%d%d", &a2, &b2);
  obim2 = obim_pravougaonika(a2, b2);
  povrsina2 = povrsina_pravougaonika(a2, b2);
```

Primer 8.1.2 Razmotrimo naredni kôd.

```
int f(int a, int b, int c) {
    if(a>b) if(b>c) return a;
    else if(a>c) return a; else return c;
    else if(b>c) return b; else return c;
}
```

Pregledom su mogu dati sledeće preporuke.

- Kôd je nepregledno formatiran; preporučuje se formatiranje korišćenjem alata za automatsko formatiranje.
- Standardi kodiranja najčešće propisuju upotrebu zagrada i kada `if` uslov sadrži samo jednu naredbu. Cilj ove konvencije je sprečavanje grešaka prilikom višestrukih `if` izraza i neispravnog uparivanja `if` i `else` grana.
- Pošto kôd računa maksimum tri broja, potrebno je dati deskriptivan naziv funkciji *f*, na primer *max3*.
- U implementaciji se preporučuje upotreba standardnog algoritma za pronalaženje maksimuma, na primer:

```
int max3(int a, int b, int c) {
    max = a;
    if(b > max) {
        max = b;
    }
    if(c > max) {
        max = c;
    }
    return max;
}
```

ili upotreba bibliotečke funkcije `max`, na primer:

```
int max3(int a, int b, int c) {
    return max(a, max(b, c));
}
```

Primer 8.1.3 Razmotrimo naredni kôd.

```
1    int strCmp(char* s1, char* s2) {
2        while(*s1 && (*s1 == *s2)) {
3            s1++;
4            s2++;
5        }
6        return *s1 - *s2;
7    }
```

Predložena implementacija može se unaprediti na više načina, na primer dodavanjem kvalifikatora `const` u argumentima, proverom pokazivača pre prvog dereferenciranja kako bi se izbeglo dereferenciranje `null` pokazivača, ili kastovanjem tako da rezultat funkcije bude tipa `int`. Ipak, najprikladnije rešenje bilo bi odbacivanje ove implementacije i korišćenje standardne bibliotečke funkcije `strcmp`.

8.2 Efekti i značaj pregleda koda

Pregledi ne garantuju potpunu eliminaciju grešaka. Pravilo prema kojem kôd mora biti pregledan pre nego što se unese u repozitorijum obezbeđuje da nijedan deo koda ne bude uključen u repozitorijum bez prethodnog pregleda. Međutim, pregledi, ukoliko se sprovode na ispravan način, garantuju veliki broj koristi u razvoju projekta.

Primer 8.2.1 (Važnost pregleda koda) Autori Džejson Koen (eng. *Jason Cohen*), Stiven Tileki (eng. *Steven Teleki*) i Erik Braun (eng. *Eric Brown*) u svojoj knjizi *Best Kept Secrets of Peer Code Review* navode interesantnu studiju slučaja o uticaju pregleda koda na troškove projekta:

> Jedan od naših klijenata odlučio je da testira koliko bi njihova kompanija uštedela da su koristili međusobne preglede koda na jednom tromesečnom projektu od 10 000 linija koda, u kojem je učestvovalo 10 programera. Pratili su koliko su grešaka otkrili tim kontrole kvaliteta i krajnji korisnici u narednih šest meseci.
>
> Zatim su angažovali drugu grupu programera da izvrši međusobni pregled tog istog koda. Koristeći prikupljene metrike iz prethodnog

perioda ovog projekta, znali su prosečne troškove otklanjanja greške u svakoj fazi razvoja, pa su mogli direktno da izračunaju koliko bi novca uštedeli.

Rezultat: pregled koda bi uštedeo polovinu troškova otklanjanja grešaka, a uz to bi bilo otkriveno još 162 dodatne greške.

Pregledi koda pozitivno utiču na rad i razvoj programera. Svest programera da će njihov rad biti pregledan od strane drugih članova tima deluje motivišuće, podstičući ih da ulože dodatni trud u proveru, kvalitetan dizajn i poštovanje pravila kodiranja.

Pregledi koda značajno doprinose razmeni znanja u timu, pre svega kroz mentorisanje manje iskusnih članova. Novi članovi tima stiču uvid u projektni kôd i brže savladavaju tehnologije i tehnike zahvaljujući povratnim informacijama i razgovorima tokom pregleda. Često se tokom pregleda ispoljava i „skriveno" znanje o kodu koje nije dokumentovano, čime se unapređuje kolektivno razumevanje projekta. Izlaganjem različitim pristupima i idejama, programeri razvijaju veštinu pisanja koda.

Zašto je decentralizacija znanja važna?
Niko ne želi da bude jedina kontakt osoba za određeni deo koda, jer to, između ostalog, onemogućava odlazak na duži odmor bez ometanja tima. S druge strane, niko ne voli da preuzima rad na tuđem kodu, naročito u hitnim situacijama, kada za upoznavanje sa kodom nema dovoljno vremena. Pregledi koda, kroz deljenje znanja unutar tima, omogućavaju da svaki član može preuzeti i nastaviti bilo koji posao, čime se smanjuje zavisnost od pojedinaca i povećava otpornost tima na promene.

Iako pregledi obično podrazumevaju da iskusniji programeri proveravaju rad mlađih kolega, oni mogu biti korisni i u suprotnom smeru. Novi članovi tima, zahvaljujući svežoj perspektivi, neretko uočavaju stare propuste i nedostatke na koje su se ostali članovi tima već navikli i prestali da ih primećuju.

Pregledi donose koristi svim članovima tima, nezavisno od primenjene razvojne metodologije. U agilnim timovima dodatno podstiču decentralizaciju znanja, tako da ne postoji pojedinac koji je jedini upoznat sa specifičnostima određenog dela koda. Na taj način se širi znanje unutar tima i omogućava ravnomernije raspodeljeno razumevanje projekta.

8.3 Preporuke za efikasan pregled koda

Prilikom organizovanja pregleda koda preporučuje se pregledanje između 200 i 400 linija koda u jednoj sesiji, kako bi se održao visok kvalitet pregleda i izbegao zamor. Ovaj broj linija je preporučen za standardne objektno-orijentisane

jezike i može da varira, odnosno da bude manji za jezike kod kojih je programiranje jezgrovitije (na primer, za kôd u funkcionalnim jezicima).

Brzina pregledanja treba da bude između 300 i 500 linija po satu[†], budući da brže pregledanje povećava verovatnoću propuštanja grešaka. Neophodno je planirati dovoljno vremena za temeljno i pažljivo pregledanje, u trajanju od najviše 90 minuta, jer duže sesije vode do zamora pri čemu se gubi kvalitet pregleda. Pre početka pregleda, programer (autor koda) treba da jasno obeleži izmene u kodu kako bi fokus pregledača bio usmeren na relevantne delove.

Za efikasan pregled koda dobro je imati pripremljenju listu provera po kojoj se proveravaju aspekti koda. Lista provera može da ima opšte delove kao i delove koji su specifični za svaki konkretan projekat.

Primer 8.3.1 (Lista provera) Naredna lista provera pokriva deo uopštenih provera vezanih za logiku rešenja, stil kodiranja, dokumentaciju, testove, obradu grešaka, bezbednost, višenitno izvršavanje i performanse.

Logika rešenja:

- Novi kôd rešava predmetni problem.
- Nije pisan novi kôd za funkcionalnost koja već postoji u postojećem, testiranom API-ju.

Stil rešenja:

- Imenovanje promenljivih je odgovarajuće.
- Formatiranje i stil prate konvencije kodiranja.
- Nema dupliranja koda i ponavljanih izračunavanja.

Dokumentacija:

- Sve potprogramske jedinice su komentarisane jasnim i razumljivim jezikom.
- Opisano je ponašanje koda u graničnim slučajevima ulaza.
- Složeni algoritmi su objašnjeni i opravdani.
- Kôd koji zavisi od neočiglednog ponašanja eksternih biblioteka je dokumentovan uz upućivanje na odgovarajuću spoljašnju dokumentaciju.

[†] Važi slična napomena, odnosno da broj može da varira u zavisnosti od programskog jezika.

- Za numeričke vrednosti su jasno navedene jedinice mere.
- Nedovršen kôd je obeležen odgovarajućim oznakama (npr. „TODO" ili „FIXME").
- Dokumentacija namenjena korisnicima je ažurirana.

Provera testova:

- Dodati su jedinični testovi za nove grane koda i ponašanja.
- Jedinični testovi pokrivaju greške i slučajeve sa nevalidnim parametrima.
- Jedinični testovi demonstriraju da algoritam radi u skladu sa dokumentacijom.

Obrada grešaka:

- Nevalidne vrednosti parametara obrađene su odmah na početku potprograma.
- Vraćene greške su adekvatno obrađene.
- Obrada grešaka osigurava da se stanje i resursi oslobode bez obzira na to gde se greška dogodila.
- Memorija je oslobođena, resursi zatvoreni i brojači referenci ažurirani i u slučaju greške i u slučaju uspeha.

Bezbednost:

- Sve promenljive su inicijalizovane odmah prilikom deklaracije ili neposredno nakon deklaracije.
- Mogući pokazivači na `null` su uvek provereni pre upotrebe.
- Indeksi nizova su provereni radi sprečavanja grešaka pristupa van granica.

Bezbedno višenitno izvršavanje:

- Globalne promenljive su zaštićene pomoću zaključavanja ili odgovarajućih potprograma.
- Objekti kojima pristupaju više niti pristupaju se isključivo kroz mehanizme zaključavanja.
- Zaključavanja se stiču i oslobađaju u ispravnom redosledu kako bi se sprečili zastoji, uključujući i u kodu za obradu grešaka.

Performanse:

- Objekti se dupliraju samo kada je to neophodno.

- Upotreba memorije je prihvatljiva čak i za velike ulazne podatke.
- Upotrebljeni algoritmi su odgovarajuće složenosti.
- Optimizacije koje otežavaju čitanje koda sprovode se i pažljivo dokumentuju samo ako alati, poput profajlera, ukažu da optimizacija daje značajan dobitak.

Primena lista provera značajno doprinosi kvalitetu pregleda za obe strane, i za programera i za pregledača. Ako je lista provera dostupna programeru (autoru koda), on može da se potrudi da proveri i po potrebi ispravi stavke koje se nalaze na listi. Pregledaču lista stavki omogućava da mu nešto što je važno za pregledanje greškom ne promakne.

Važno je održavati zdravu kulturu pregleda koda u kojoj se otkrivanje grešaka posmatra kao prilika za unapređenje kvaliteta, a ne kao kritika pojedinca. Posebnu pažnju treba obratiti i na negativni efekat koji se naziva efekat „velikog brata": programeri mogu steći utisak da su pod stalnim nadzorom i da će vrednosti metrika koje se prikupljaju prilikom pregleda (na primer, broj izmena u kodu, broj dodatih novih linija, broj pronađenih grešaka) biti upotrebljene protiv njih. To može dovesti do toga da se pažnja usmeri na poboljšanje statistika umesto na unapređenje kvaliteta koda. Iz tog razloga preporučuje se pažljivo biranje tona i načina izražavanja u komentarima, koji treba da se odnose isključivo na kôd i predložene izmene, a nikada na lične osobine programera.

Čak i u situacijama kada nije moguće pregledati ceo kôd, preporučuje se pregled makar njegovog dela, jer sama činjenica da će kôd biti pregledan podstiče programera da uloži dodatni trud i napiše kvalitetniji kôd. Takođe, delimična povratna informacija može da doprinese unapređenju celokupnog koda kao i da omogući programeru da nastavi sa radom dok ne stigne kompletna povratna informacija.

Poželjno je usvojiti proces pregleda koji koristi odgovarajuće alate za organizovanje i evidentiranje pregleda, čime se proces čini efikasnijim i transparentnijim. Nakon pregleda i nakon što programer obradi dobijene komentare, potrebno je proveriti da li su uočeni propusti zaista i otklonjeni.

Za unapređenje procesa pregledanja preporučuje se definisanje jasnih, merljivih ciljeva i praćenje relevantnih metrika radi kontinuiranog unapređenja procesa. Relevantne metrike obuhvataju broj linija koda pregledanih u jedinici vremena, broj

uočenih grešaka, dužinu trajanja pojedinačnih pregledanja, broj pregleda u toku radne nedelje i slično.

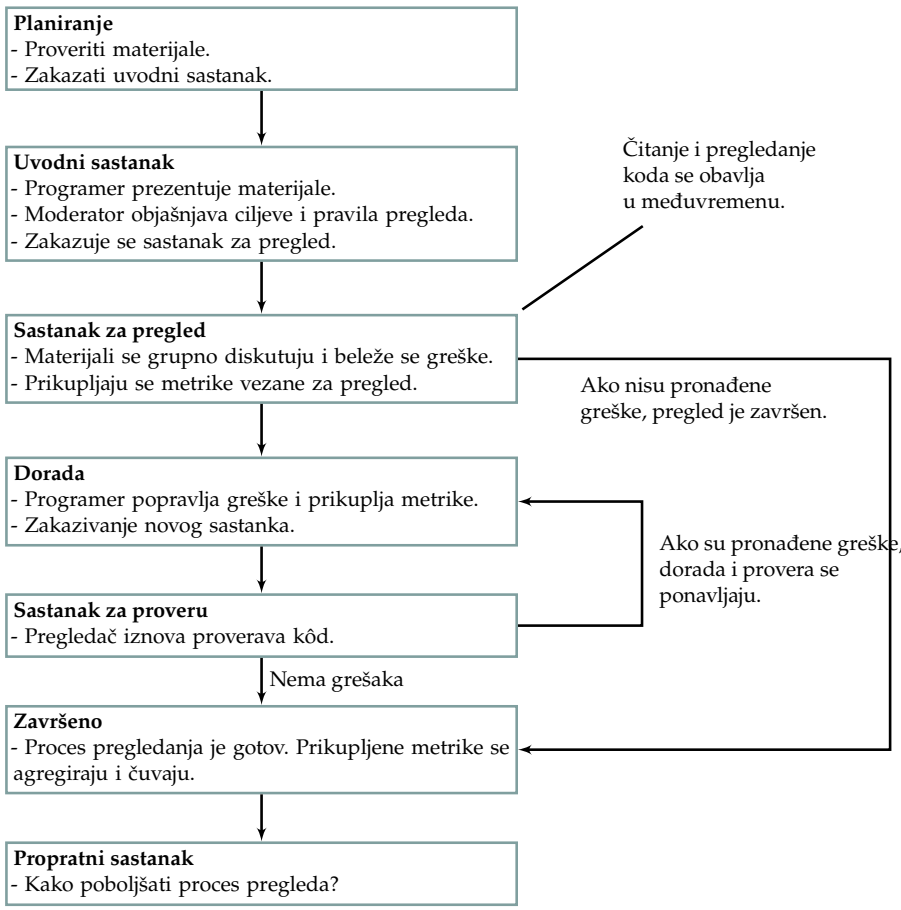
8.4 Formalni pregledi

Formalni pregledi (eng. *formal inspections*) podrazumevaju grupne sastanke, u kojima učestvuje između tri i šest osoba, na kojima se zajednički diskutuje o kodu i vrši pregled. Ovaj pristup predstavlja dosta skup i vremenski zahtevan proces, koji se u praksi sve ređe primenjuje. Iako formalni pregledi omogućavaju pronalaženje najvećeg broja grešaka, njihova realizacija zahteva značajne resurse i angažovanje, što većina organizacija ne može da priušti na svakodnevnom nivou. Formalni pregledi se zato nekada koriste samo za kritične delove koda. Proces koji je potrebno sprovesti radi realizacije formalnog pregleda koda prikazan je na slici 8.2.

Proces započinje planiranjem aktivnosti, pri čemu se najpre zakazuje uvodni sastanak. Na uvodnom sastanku programer prezentuje svoj kôd, dok moderator podseća na ciljeve i pravila pregledanja. Tom prilikom se dogovara i termin glavnog sastanka za pregled. U periodu između uvodnog i glavnog sastanka, učesnici samostalno proučavaju materijale i pripremaju se za zajednički pregled.

Na sastanku za pregled materijali se zajednički diskutuju i komentarišu, a uočene greške se beleže. Jedna osoba ima zadatak da prikuplja metrike vezane za proces pregledanja, uključujući podatke o utrošenom vremenu, broju pronađenih grešaka, broju grešaka koji zahtevaju ispravke i slično. Ukoliko greške nisu uočene, proces pregledanja se ovde završava. U suprotnom, programer koda otklanja greške, ažurira relevantne metrike i zakazuje dodatne sastanke radi provere ispravljenih delova koda. Proces se smatra završenim tek kada svi pregledači potvrde da u kodu ne primećuju greške.

Na kraju procesa, prikupljene metrike se agregiraju i arhiviraju. Po potrebi, može se organizovati i dodatni sastanak posvećen analizi i predlozima za unapređenje samog procesa pregledanja.



Slika 8.2: Tipičan proces formalnog pregleda. Artefakti dobijeni tokom pregleda nisu prikazani. To su spisak grešaka, beleške sa sastanaka i spisak prikupljenih metrika. Neki pregledi takođe imaju završnu anketu na propratnom sastanku.

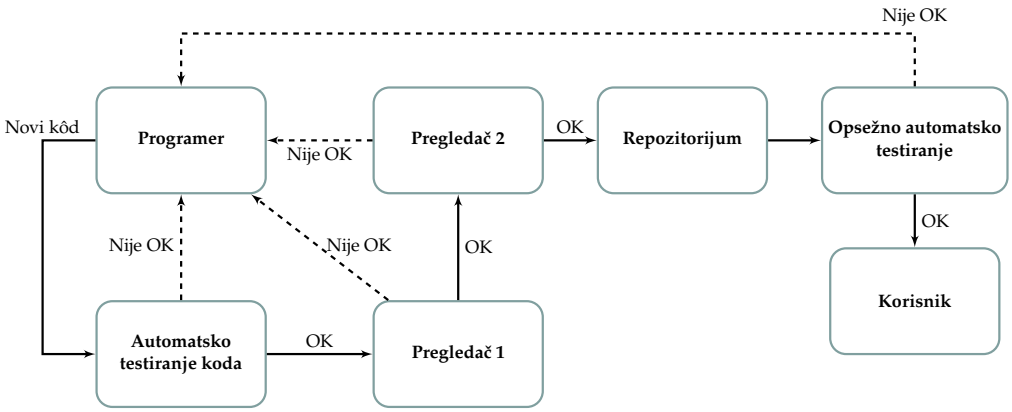
8.5 Neformalni pregledi

Neformalni pregledi mogu da budu više i manje neformalni. Osnovne vrste neformalnih pregleda su:

- pregled „preko ramena" (eng. *over-the-shoulder reviews*),
- pregled putem elektronske pošte (mejla) (eng. *e-mail pass-around*),
- pregled korišćenjem specijalizovanih alata za pregled koda (eng. *tool-assisted reviews*),
- programiranje u paru (eng. *pair-programming*).

Neformalni pregledi su veoma česti u okviru agilnih metodologija razvoja softvera i posebno u okviru kontinuirane integracije i isporuke softvera. Tok koda od lokalne izmene

do korisnika u sklopu kontinuirane integracije i isporuke softvera prikazan je na slici 8.3. Ovaj tok važi bez obzira na primenjenu vrstu neformalnog pregleda koda.



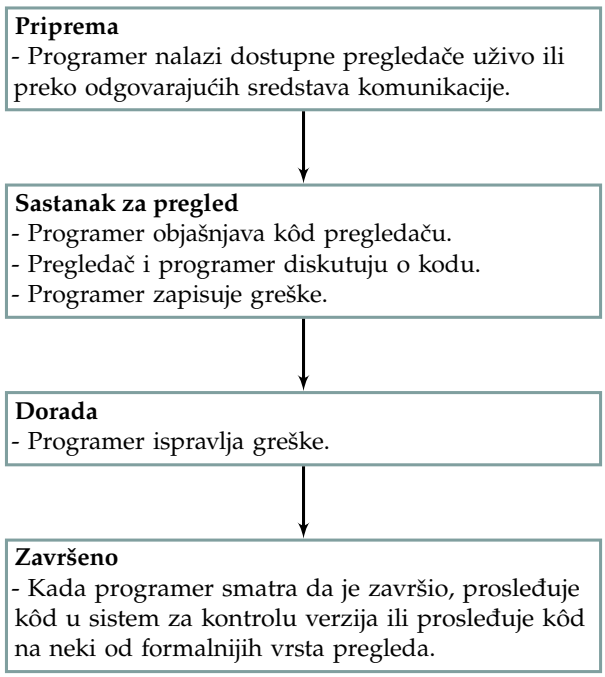
Slika 8.3: Tok koda od lokalne izmene do korisnika

Pre početka pregleda koda podrazumeva se da je programer izvršio osnovni skup testova i da su svi testovi uspešno prošli. Proveru koda obično vrše najmanje dva pregledača, a proces se nastavlja tek kada oba pregledača daju saglasnost da kôd može biti unet u repozitorijum. Nakon toga sprovode se opsežnija testiranja, a ukoliko neki od testova ne prođe, kôd se vraća programeru na doradu. Tek kada svi testovi budu uspešno izvršeni, izmena se prihvata i postaje vidljiva krajnjem korisniku.

Pregled „preko ramena"

Pregled „preko ramena" je veoma čest i predstavlja najneformalniji oblik pregleda. Proces pregleda „preko ramena" je prikazan na slici 8.4. Pregled „preko ramena" se zakazuje najčešće uživo, ali ukoliko je tim distribuiran, onda putem odgovarajućih sredstava komunikacije. Prilikom pregleda „preko ramena", programer objašnjava pregledaču šta je implementirano u kodu i iz kojih razloga. Programer beleži greške i dorade koje je potrebno da uradi nakon sastanka. Ovaj oblik pregleda može se realizovati na daljinu korišćenjem interneta i deljenjem ekrana, te ne zahteva fizičko prisustvo učesnika.

Pregled „preko ramena" predstavlja najjednostavniji oblik pregleda koda, najmanje organizaciono zahtevan, a dodatno



Slika 8.4: Proces pregleda „preko ramena"

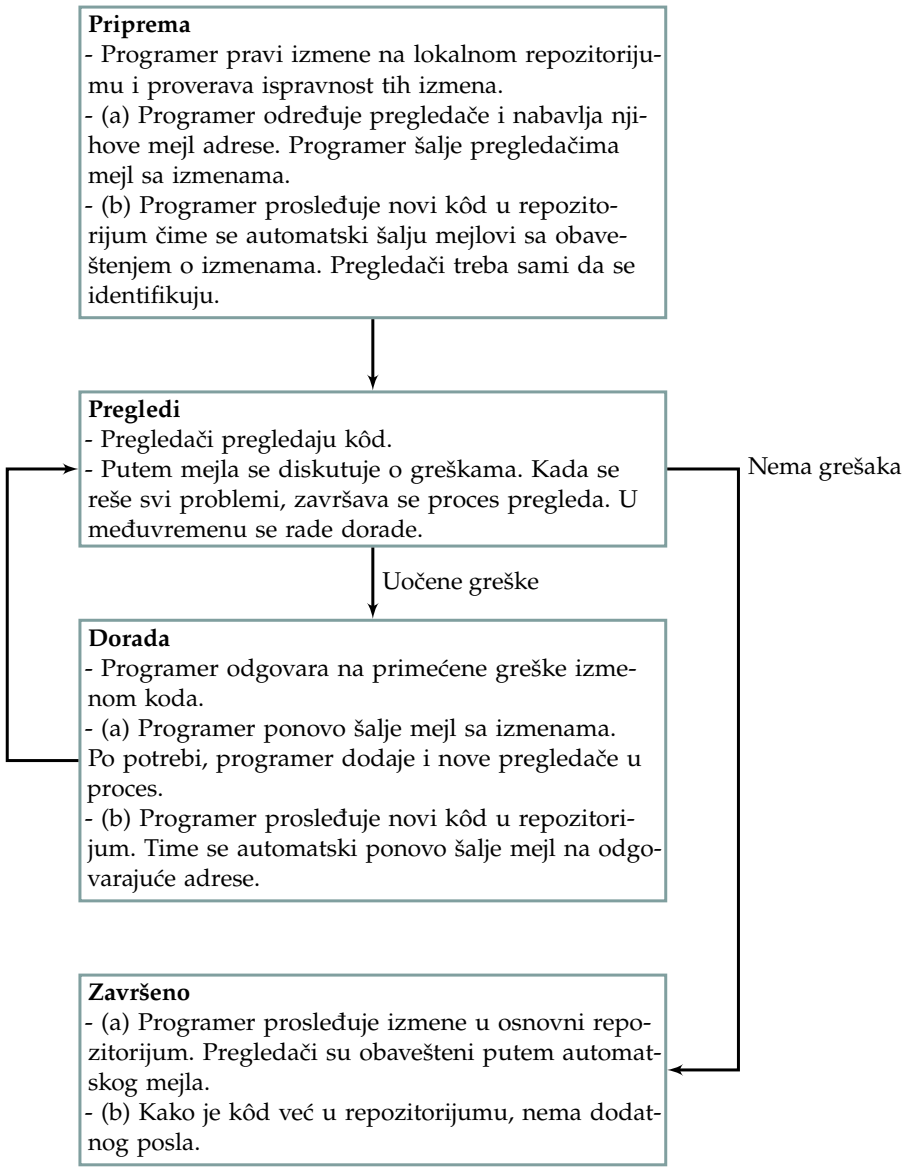
omogućava i razmenu ideja koje često ne bi bile podeljene pisanim putem. Ovakav pristup, međutim, ima i određene nedostatke. Nemoguće je precizno ispratiti šta je tačno pregledano, a postoji i rizik da neka izmena bude propuštena iako je uočeno da je potrebna. Čak i kada se identifikuju greške i sprovedu korektivne akcije, postoji mogućnost da one budu pogrešno primenjene ili da uvedu nove greške, budući da se nakon ovakvog pregleda obično ne sprovodi dodatna provera. Zbog toga se ova vrsta pregleda često kombinuje sa drugim, formalnijim vrstama pregleda.

Pregled putem elektronske pošte

Pregled putem elektronske pošte (mejla) podrazumeva slanje izmena pregledačima elektronskom poštom radi komentarisanja i ocene. Ovakav oblik pregleda lako je primenljiv i kada su pregledači locirani na različitim udaljenim lokacijama.

Pregled putem mejla može se organizovati pre nego što se kôd unese u repozitorijum, čime se sprečava unošenje grešaka u glavni tok razvoja, ili nakon što kôd već uđe u repozitorijum, kao naknadna kontrola. Obe varijante imaju svoje prednosti i ograničenja: prva omogućava proaktivno otkrivanje problema, dok druga obezbeđuje bržu integraciju

ali zahteva dodatne revizije ukoliko se uoče greške. Proces pregleda preko mejla je prikazan na slici 8.5.



Slika 8.5: Razlike u procesu pregleda preko mejla su obeležene sa (a) pre nego što kôd uđe u repozitorijum i (b) nakon što kôd uđe u repozitorijum. Zajednički delovi procesa nisu obeleženi.

Pregled putem mejla pre ulaska koda u repozitorijum podrazumeva da programer odredi pregledače i prosledi im predlog izmena u kodu mejlom. Ukoliko se pregled radi nakon ulaska koda u repozitorijum, onda se mejl o izmenama automatski šalje na kofigurisane adrese. Za manje projekte, to

mogu da budu svi programeri na projektu, dok se kod većih projekata to određuje na osnovu koncepta vlasništva odgovornosti za određene delove koda. U oba slučaja, potrebno je da se pregledači sami identifikuju i odazovu pregledu.

Pregledači zatim pregledaju dostavljene izmene, a diskusija o uočenim greškama i potrebnim doradama vodi se u daljoj prepisci putem mejla, pri čemu programer sprovodi ispravke u skladu sa predlozima iz diskusije. Nekada se u toku procesa pregleda dodaju novi pregledači. Programer ispravke koje pravi šalje ponovno mejlom ili unosi u repozitorijum što povlači ponovno automatsko slanje mejlova. Prepiska se zaustavlja kada se otklone sve uočene greške.

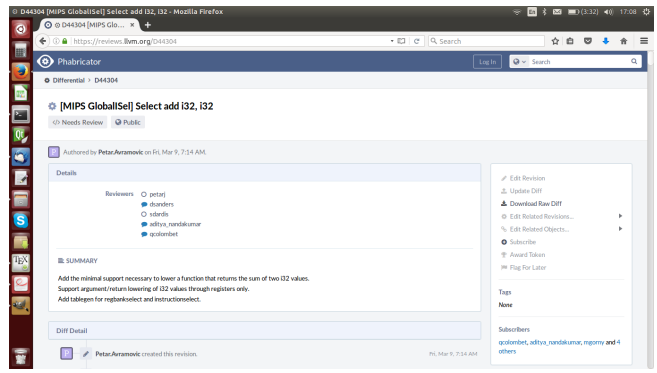
Dugi niz godina ovaj pristup bio je jedan od najčešćih oblika pregledanja koda zahvaljujući jednostavnosti i lakoj primeni. Međutim, u savremenoj praksi se sve ređe koristi, jer su ga zamenili specijalizovani alati za kolaborativno pregledanje koda, koji efikasnije rešavaju probleme specifične za pregled putem mejla, kao što su slaba preglednost, nedostatak evidencije o statusu komentara i otežano praćenje istorije izmena.

Pregled koda korišćenjem specijalizovanih alata

Pregled koda korišćenjem specijalizovanih alata može se sprovoditi u okviru različitih metodologija razvoja softvera. Ovi alati omogućavaju lakše vođenje diskusija, praćenje statusa komentara i preporuka, kao i čuvanje istorije pregleda.

Sam proces pregleda koda korišćenjem specijalizovanih alata za pregled veoma je sličan pregledima putem mejla, osim što sada na raspolaganju postoji alat koji daje podršku za razne delove procesa. Proces pregleda isto započinje nakon što programer razvije kôd i lokalno proveri njegovu ispravnost. Nakon toga, on postavlja predlog svojih izmena u alata za pregledanje koda. Uz pregled izmena obično se ostavlja i kratak tekstualni opis koji objašnjava prirodu izmena (da li je upitanju ispravljanje nekog defekta ili dodavanje nove funkcionalnosti u kôd) i kratak opis izmene i odluka koje su ukviru izmena načinjene (slika 8.6).

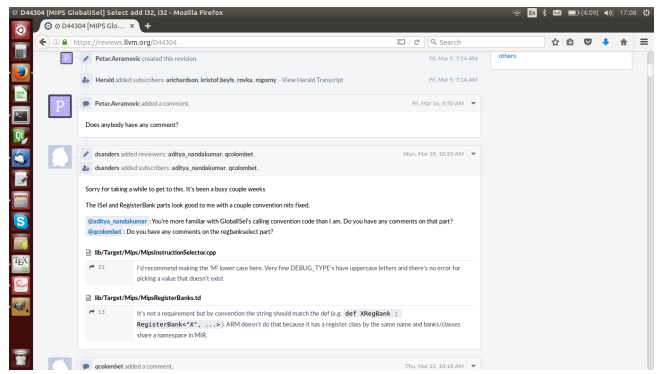
Neki alati očekuju kao ulaz datoteku koja sadrži sve izmene u odnosu na originalni kôd (na primer, datoteku koja se može generisati komandom *git diff* ukoliko se koristi alat za kontrolu verzija *git*). Alati za pregled koji su integrisani



Slika 8.6: Uvid u pregledače i u opis izmena korišćenjem alata *Phabricator*

sa sistemom za kontrolu verzija obično direktno generišu razlike koje su potrebne za pregled koda.

Sledeći korak podrazumeva određivanje pregledača. Svaki tim može da ima svoja pravila vezana za odabir pregledača, ali najčešće postoji podela odgovornosti po delovima koda u zavisnosti od koncepta vlasništa nad kodom ili količine doprinosa tom delu koda. Programer postavlja zahtev za pregledanjem koda i sva diskusija vezana za kôd se dešava korišćenjem alata za pregled (slika 8.7). Često komentari i

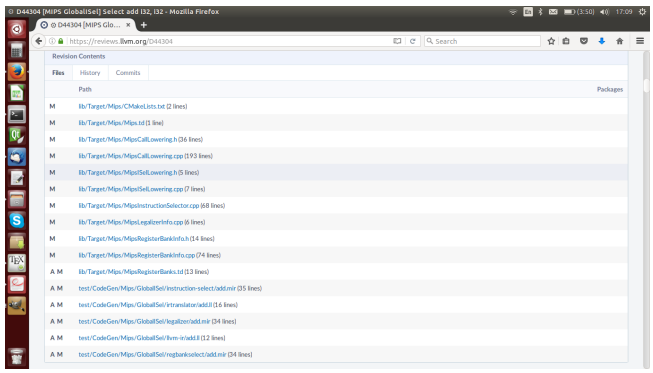


Slika 8.7: Uvid u deo diskusije korišćenjem alata *Phabricator*

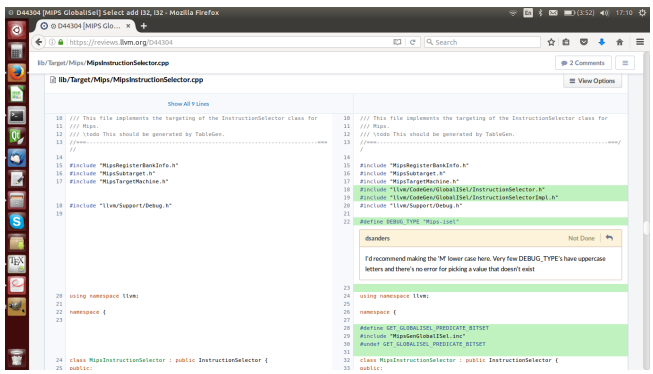
izmene koje se ostave korišćenjem alata za pregled stižu i na mejlove učesnika diskusije. Svi alati za pregled koda daju jasan uvid u sve fajlove koji su izmenjeni ili dodati (slika 8.8).

Programer radi dorade u skladu sa uočenim greškama i postavlja nove izmene u alat za pregledanje (slika 8.9).

Po potrebni, programer može da doda i nove pregledače u proces pregledanja. Proces se završava kada se sve uočene



Slika 8.8: Uvid u spisak modfikovanih i dodatih fajlova korišćenjem alata *Phabricator*



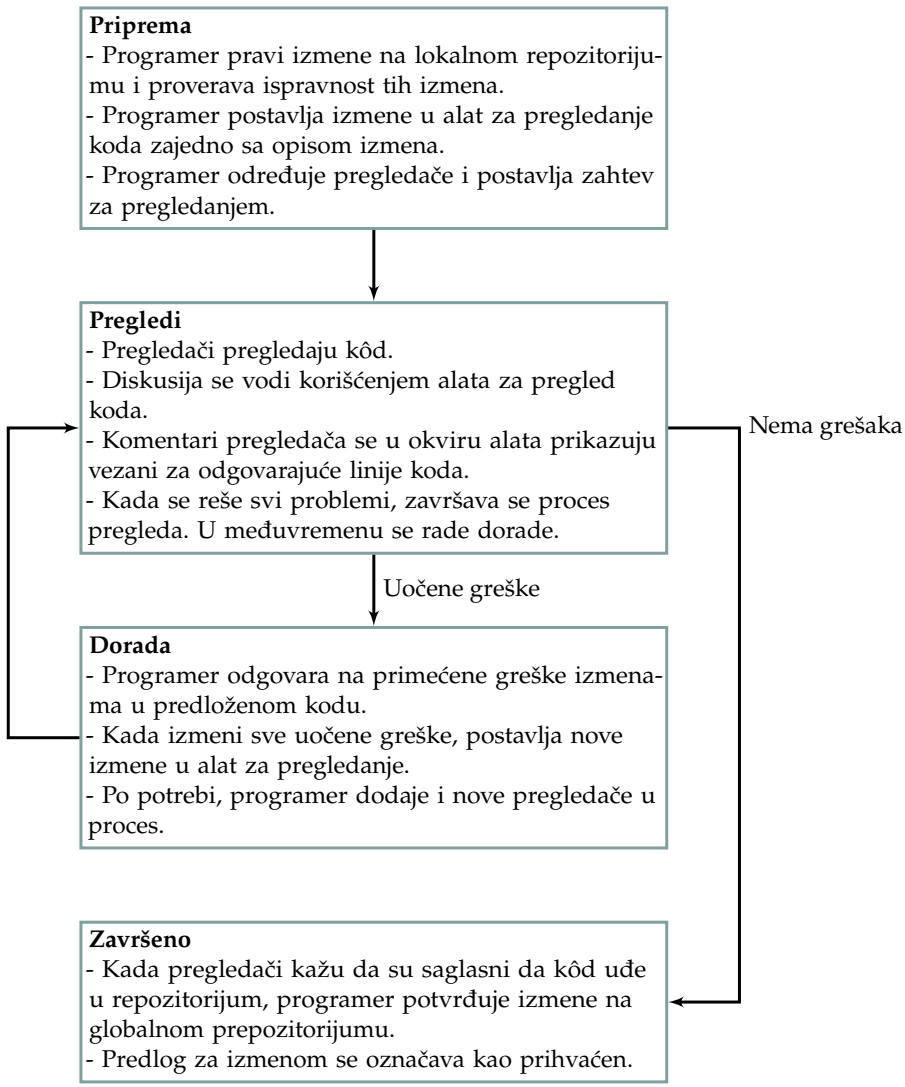
Slika 8.9: Uvid u razlike u kodu korišćenjem alata *Phabricator*: originalni kôd je dat sa leve strane, zelenom bojom su označene dodate linije sa desne strane). Komentar pregledača je umetnut u kôd.

greške uklone i kada pregledači označe da je u redu da kôd uđe u glavni repozitorijum. Proces pregleda prikazan je na slici 8.10.

Alatima za pregled koda su najčešće veb zasnovani i pristupa im se korišćenjem veb pregledača. Na raspolaganju je veliki broj alata koji podržavaju ovu aktivnost, uključujući *Phabricator*, *Gerrit*, *Collaborator*, *GitLab* i mnoge druge.

Programiranje u paru

Programiranje u paru može se posmatrati kao posebna vrsta pregleda koda, budući da druga osoba kontinuirano prati i komentariše rad programera. Kod programiranja u paru pregled koda se odvija u realnom vremenu, paralelno sa njegovim pisanjem, što omogućava brzu identifikaciju problema i razmenu ideja. Ovaj pristup podrazumeva da dva programera rade zajedno za istim radnim mestom, pri čemu jedan od njih aktivno piše kod, dok drugi posmatra, razmišlja



Slika 8.10: Proces pregleda korišćenjem specijalizovanih alata za pregled koda

o široj slici i odmah uočava potencijalne greške ili predlaže poboljšanja. Uloge se u toku rada periodično menjaju kako bi oba člana tima ravnomerno doprinosila i iz oba ugla sagledavala problem.

Iako programiranje u paru često vodi ka kvalitetnijem i čitljivijem kodu, ono zahteva dodatno vreme i resurse te nije često zastupljeno u razvojnim procesima. Dodatno, postoji rizik da programeri koji rade zajedno dele sličan način razmišljanja i zajedno previde određene greške. Zbog toga su eksterni pregledi od strane drugih članova tima i dalje

neophodni i predstavljaju dodatnu kontrolu kvaliteta, čak i kada se programiranje u paru primenjuje.

Rezime

- Prilikom pregleda koda proverava se da li kôd ispravno implementira predviđenu funkcionalnost i traže se greške u kodu koji se ne mogu otkriti testiranjem.
- Pored značajnog uticaja na poboljašenje kvaliteta koda, pregledi koda donose i niz drugih koristi uključujući razmenu znanja u timu i pozitivan uticaj na rad i razvoj programera.
- Za efikasan pregled koda dobro je razviti listu provera i pridržavati se standardnih preporuka o količini i načinu pregledanja.
- Najčešće se pregled koda obavlja pre ulaska koda u repozitorijum korišćenjem specijalizovanih alata za pregled koda.

Ispitna pitanja

61. Pregledi koda. Ciljevi pregleda koda. Primeri.
62. Pregledi koda. Efekti i značaj pregleda koda. Primeri.
63. Pregledi koda. Preporuke za efikasan pregled koda. Primeri.
64. Pregledi koda. Formalni pregledi.
65. Pregledi koda. Neformalni pregledi. Pregled „preko ramena”.
66. Pregledi koda. Neformalni pregledi. Pregled putem elektronske pošte.
67. Pregledi koda. Neformalni pregledi. Pregled korišćenjem specijalizovanih alata.
68. Pregledi koda. Neformalni pregledi. Programiranje u paru.

9 Simboličko izvršavanje

Pregled

-
-
-
-
-

Proveravanje modela 10

Pregled

-
-
-
-
-

11 Apstraktna interpretacija

Pregled

-
-
-
-