GLAVA

1

Proces razvoja softvera

Pod *razvojem softvera* često se ne podrazumeva samo neposredno pisanje programa, već i procesi koji mu prethode i slede. U tom, širem smislu, razvoj softvera naziva se i *životni ciklus razvoja softvera*. Razvoj softvera razlikuje se od slučaja do slučaja, ali u nekoj formi obično ima sledeće faze i podfaze:

Planiranje: Ova faza obuhvata prikupljanje i analizu zahteva od naručioca softvera, razrešavanje nepotpunih, višesmislenih ili kontradiktornih zahteva i kreiranje precizne specifikacije problema i dizajna softverskog rešenja. Podfaze ove faze, opisane u poglavlju 1.1, su:

- Analiza i specifikovanje problema;
- Modelovanje rešenja;
- Dizajn softverskog rešenja.

Realizacija: Ova faza obuhvata implementiranje dizajniranog softverskog rešenja u nekom konkretnom programskom jeziku. Implementacija treba da sledi opšte preporuke, kao i preporuke specifične za realizatora ili za konkretan projekat. Analizom efikasnosti i ispravnosti proverava se pouzdanost i upotrebljivost softverskog proizvoda, a za naručioca se priprema i dokumentacija. Podfaze ove faze su:

- Implementiranje (kodiranje, pisanje programa) (o nekim aspektima ove podfaze govori glava **??**);
- Evaluacija – analiza ispravnosti i analiza efikasnosti (o nekim aspektima ovih podfaza govore redom glava **??** i glava **??**);
- Izrada dokumentacije (obično korisničke dokumentacije – koja opisuje korišćenje programa i tehničke dokumentacije — koja opisuje izvorni kôd);

Eksploatacija: Ova faza počinje nakon što je ispravnost softvera adekvatno proverena i nakon što je softver odobren za upotrebu. Puštanje u rad uključuje instaliranje, podešavanja u skladu sa specifičnim potrebama i zahtevima korisnika, ali i testiranje u realnom okruženju i sveukupnu evaluaciju sistema u stvarnim uslovima korišćenja. Organizuje se obuka za osnovne i napredne korisnike i obezbeđuje održavanje kroz koje se ispravljaju greške ili dodaju nove manje funkcionalnosti. U održavanje se obično uloži više od tri četvrtine ukupnog rada u čitavom životnom ciklusu softvera. Podfaze ove faze su:

- Obuka i tehnička podrška;
- Puštanje u rad;
- Održavanje.

Postoje međunarodni standardi, kao što su ISO/IEC 12207 i ISO/IEC 15504, koji opisuju životni ciklus softvera kroz precizno opisane postupke izbora, implementacije i nadgledanja razvoja softvera. Kvalitet razvijenog softvera često se ocenjuje prema nivou usklađenosti sa ovim standardima.

Kontrola kvaliteta softvera (eng. software quality assurance, SQA) pokriva kompletan proces razvoja softvera i sve njegove faze i podfaze. Proces kontrole kvaliteta, takođe opisan standardom ISO/IEC 15504, treba da osigura nezavisnu potvrdu da su svi proizvodi, aktivnosti i procesi u skladu sa predefinisanim planovima i standardima.

Faze razvoja softvera i moguće probleme na šaljiv način ilustruje čuvena karikatura prikazana na slici 1.1.

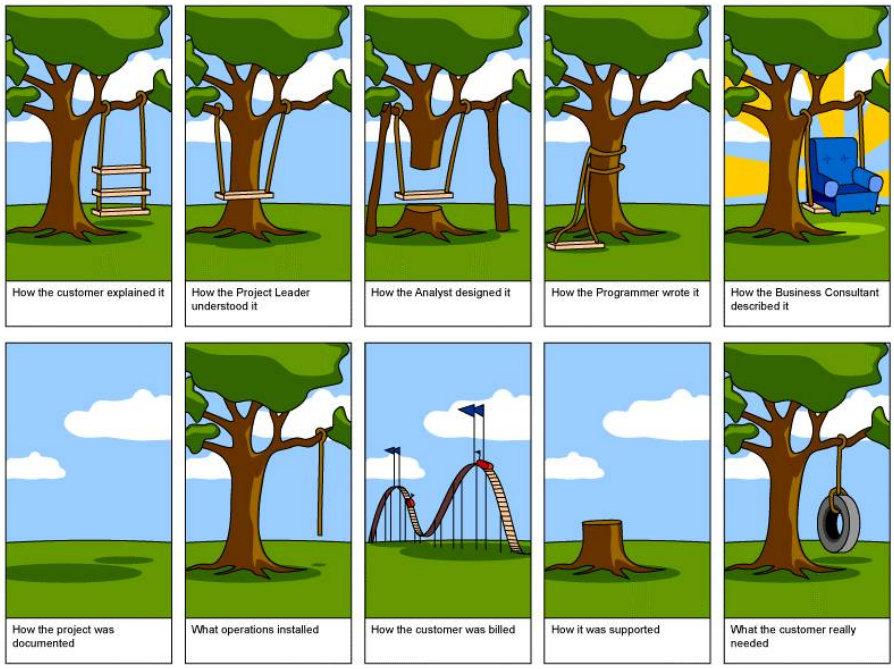
Za razvoj softvera relevantni su i procesi istraživanja tržišta, nabavke softvera, naručivanja softvera, tenderi, razmatranje ponuda i slični, ali u ovom tekstu neće biti reči o njima.

1.1 Planiranje

Poslovna analiza u fazi planiranja bavi se, pre svega, preciznom postavkom i specifikovanjem zahteva, dok se modelovanje i dizajn bave razradom projekta koji je definisan analizom zahteva. U fazi planiranja često se koriste različite dijagramske tehnike i specijalizovani alati koji podržavaju kreiranje ovakvih dijagrama (takozvani CASE alati, engl. Computer Aided Software Engineering). Procesom planiranja strateški rukovodi arhitekta čitavog sistema (engl. enterprise architect, EA). Njegov zadatak je da napravi opšti, apstraktan plan svih procesa koji treba da budu softverski podržani.

1.1.1 Analiza i specifikovanje problema

Proces analize i specifikovanja problema obično sprovodi *poslovni analitičar* (engl. business analyst, BA), koji nije nužno informatičar, ali mora da poznaje relevantne poslovne ili druge procese. Kada se softver pravi po narudžbini, za poznatog kupca, u procesu analize i specifikovanja problema vrši se intenzivna komunikacija



Slika 1.1: Faze razvoja softvera ilustrovane na šaljiv način

poslovnog analitičara sa naručiocima, krajnjim korisnicima ili njihovim predstavnicima. Kada se softver pravi za nepoznatog kupca, često u kompanijama ulogu naručioca preuzimaju radnici zaposleni u odeljenju prodaje ili marketinga (koji imaju ideju kakav proizvod bi kasnije mogli da prodaju).

U komunikaciji poslovnog analitičara sa naručiocima, često se najpre vrši analiza postojećih rešenja (na primer, postojećeg poslovnog procesa u kompaniji koja uvodi informacioni sistem) i razmatraju se mogućnosti njihovog unapređenja uvođenjem novog softvera. Naručioci često nemaju informatičko obrazovanje, pa njihovi zahtevi koje softver treba da zadovolji mogu da budu neprecizni ili čak i kontradiktorni. Zadatak poslovnog analitičara je da, u saradnji sa naručiocima, zahteve precizira i uobliči. Rezultat analize je opšta specifikacija problema koja opisuje problem (na primer, poslovne procese) i željene funkcionalnosti programa, ali i potrebnu efikasnost i druga svojstva.

Pored precizne analize zahteva, zadatak poslovne analize je i da proceni: obim posla[1] koji treba da bude urađen (potrebno je precizno definisati šta projekat treba da obuhvati, a šta ne); rizike koji postoje (i da definiše odgovarajuće reakcije u

[1] Obim posla često se izražava u terminima broja potrebnih čovek-meseci (jedan čovek-mesec podrazumeva da jedan čovek na projektu radi mesec dana).

slučaju da nešto pođe drugačije nego što je planirano); potrebne resurse (ljudske i materijalne); očekivanu cenu realizacije projekta i njegovih delova; plan rada (po fazama) koji je neophodno poštovati i slično.

Kada je problem precizno specifikovan, prelazi se na sledeće faze u kojima se modeluje i dizajnira rešenje specifikovanog problema.

1.1.2 Modelovanje rešenja

Modelovanje rešenja obično sprovodi *arhitekta rešenja* (engl. solution architect, SA), koji mora da razume specifikaciju zahteva i da je u stanju da izradi matematičke modele problema i da izabere adekvatna softverska rešenja, na primer, programski jezik, bazu podataka, relevantne biblioteke, strukture podataka, algoritamska rešenja, itd.

Model može biti matematički model (na primer, optimizacioni model, sistemski graf ili formalna specifikacija ponašanja), ali i simulacija, heuristički opis sistema, pseudokod ili vizuelna skica koja oslikava osnovne funkcionalnosti i odnose među komponentama. U određenim domenima koriste se i posebni domen-specifični jezici.

Cilj modelovanja rešenja je da se složeni problemi razlože na jasne logičke celine koje se zatim mogu dalje precizirati kroz dizajn softverskog sistema. Model mora da bude dovoljno precizan da se iz njega može izvesti dizajn, ali i dovoljno apstraktan da omogući fleksibilnost u izboru konkretnih tehnologija.

Na primer, u sistemu za elektronsku narudžbinu hrane, rešenje može biti modelovano kao relacija između korisnika, restorana i narudžbina, pri čemu tok narudžbine predstavlja automat sa stanjima: „kreirana", „potvrđena", „u pripremi", „u dostavi", „isporučena".

Modelovanje rešenja je posebno važno u kompleksnim sistemima jer omogućava bolju komunikaciju među članovima tima i olakšava procenjivanje troškova, složenosti i rizika implementacije.

Modelovanje rešenja fokusira se na razumevanje problema i njegovih ključnih komponenti, dok dizajn detaljno opisuje kako će te komponente biti tehnički realizovane na konkretnoj platformi.

1.1.3 Dizajn softverskog rešenja

U procesu dizajniranja, *arhitekta softvera* (engl. software architect) vrši preciziranje rešenja i opisuje *arhitekturu softvera* (engl. software architecture).

Arhitektura softvera

Predstavlja celokupnu strukturu softvera i načine na koje ta struktura obezbeđuje integritet sistema i željeni ishod projekta (ispravan softver, dobre performanse, poštovanje rokova i uklapanje u planirane troškove). Arhitektura softvera uključuje i komponente, njihove međusobne odnose i interakcije, kao i principe i smernice koje vode dizajn i evoluciju softverskog rešenja.

Dizajn razrađuje i pojmove koji su u ranijim fazama bili opisani nezavisno od konkretnih tehnologija, dajući opšti plan kako sistem da bude izgrađen na konkretnoj hardverskoj i softverskoj platformi. Tokom dizajna često se koriste neki unapred ponuđeni obrasci (engl. design patterns) za koje je praksa pokazala da predstavljaju kvalitetna rešenja za određenu klasu problema.

U jednostavnijim slučajevima (na primer kada softver treba da radi autonomno, bez korisnika i korisničkog interfejsa), dizajn može biti dat i u neformalnom tekstualnom obliku ili u vidu jednostavnog dijagrama toka podataka tj. tokovnika (engl. data flow diagram)[2]. U kompleksnijim slučajevima, koriste se standardizovane grafičke notacije (kaže se i *grafički jezici*), poput UML (Unified Modeling Language), koji omogućavaju modelovanje podataka, modelovanje poslovnih procesa i modelovanje softverskih komponenti.

Neke od osnovnih tema koje se razmatraju u okviru dizajna softvera su:

- **Apstrahovanje** (engl. abstraction) – apstrahovanje je proces generalizacije kojim se odbacuju nebitne informacije tokom modelovanja nekog entiteta ili procesa i zadržavaju samo one informacije koje su bitne za sâm softver. Na primer, apstrahovanjem se uočava da boja očiju studenta nema nikakvog značaja u informacionom sistemu fakulteta i ta informacija se onda odbacuje prilikom predstavljanja studenta u sistemu.

- **Profinjavanje** (engl. refinement) – profinjavanje je proces razvoja programa odozgo-naniže. Nerazrađeni koraci se tokom profinjavanja sve više preciziraju dok se na samom kraju ne dođe do sasvim preciznog opisa u obliku funkcionalnog programskog koda. U svakom koraku jedan zadatak razlaže se na sitnije zadatke. Na primer, u nekoj situaciji, zadatak koji obavlja funkcija `obradi_podatke_iz_datoteke()` razloži se na zadatke koje obavljaju funkcije `otvori_datoteku()`, `procitaj_podatke()`, `obradi_podatke()`, `zatvori_datoteku()`, itd. Apstrahovanje i profinjavanje međusobno su suprotni procesi.

- **Dekompozicija** (engl. decomposition) – cilj dekompozicije je razlaganje na komponente koje je lakše razumeti, realizovati i održavati. Njen proizvod nije

[2] Ovi dijagrami ilustruju kako podaci teku kroz sistem i kako se izlaz izvodi iz ulaza kroz niz funkcionalnih transformacija, ali ne opisuju kako ih treba implementirati. Notacija koja se koristi u tokovnicima nije standardizovana, ali različite notacije su često veoma slične i intuitivne.

implementacija, već opis arhitekture softverskog rešenja. Postoje različiti pristupi dekompoziji, obično u skladu sa programskom paradigmom koja će se koristiti (na primer, objektno-orijentisana, funkcionalna, itd). Većina pristupa teži razlaganju na komponente tako da se što više smanje njihove zavisnosti (tako što unutrašnje informacije jednog modula nisu dostupne iz drugih) i da se poveća kohezija (jaka unutrašnja povezanost) pojedinačnih komponenti. Na primer, u funkcijski-orijentisanom dizajnu, svaka funkcija odgovorna je samo za jedan zadatak i sprovodi ga sa minimalnim uticajem na druge funkcije. Rezultat dekompozicije često se prikazuje grafički, u vidu strukturnog modela sistema koji opisuje veze između komponenti i njihovu hijerarhiju (na svakom nivou hijerarhije, svakom čvoru koji nije list, odgovara nekoliko, obično između dva i sedam, podređenih čvorova).

- **Modularnost** (engl. modularity) – softver se deli na komponente koje se nazivaju moduli. Svaki modul ima precizno definisanu funkcionalnost i poželjno je da moduli što manje zavise jedni od drugih kako bi mogli da se koriste i u drugim programima.

1.1.4 Objedinjeni jezik za modelovanje, UML dijagrami

Objedinjeni jezik za modelovanje, UML (eng. Unified Modeling Language) dijagrami predstavljaju vizulenu tehniku za kreiranje dijagrama kojim se opisuju zahtevi, akcije i fizička distribucija softverskog rešenja. UML je standardizovani jezik za modelovanje softvera.

UML je pre svega grafički jezik, ali se može koristiti i u tekstualnom obliku. Samo neki elementi se po standardu opisuju tekstualno. Postoji mnogo vrsta UML dijagrama i ovde će biti prikazani samo neki od njih. Ipak, svi UML dijagrami se mogu podeliti na:

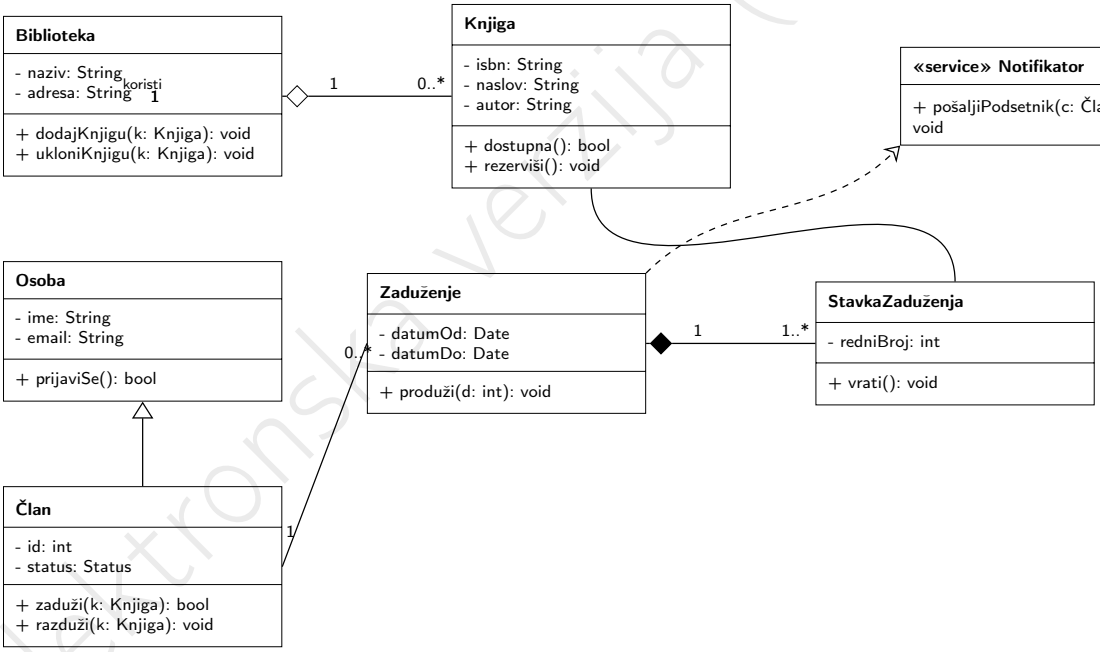
- **Strukturni dijagrami** – prikazuju strukturu sistema i odnose između njegovih komponenti. Primeri su dijagrami klasa, objekata, komponenti, rasporeda, itd.

- **Dijagrami ponašanja** – prikazuju kako se sistem ponaša tokom izvršavanja. Primeri su dijagrami aktivnosti, stanja, sekvenci, upotrebe, itd.

Strukturni dijagrami.

Strukturni UML dijagrami predstavljaju statički prikaz softverskog sistema i opisuju njegove elemente (klase, pakete, komponente, uređaje) i njihove međusobne odnose. Za razliku od dijagrama ponašanja, oni ne sadrže informaciju o vremenskom toku niti o dinamičkim promenama stanja. Koriste se u svim fazama razvoja softvera – od analize do implementacije i isporuke.

Najčešće korišćeni strukturni dijagram je dijagram klasa. *Dijagram klasa* prikazuje klase, njihove atribute i metode, kao i odnose kao što su nasleđivanje, asocijacija, kompozicija, agregacija i zavisnost. Na primer, u sistemu za upravljanje bibliotekama, klase `Knjiga`, `Član` i `Zaduženje` povezane su odnosima koji odražavaju pravila zaduživanja knjiga (Slika 1.2).



Slika 1.2: Primer strukturnog UML dijagrama klasa za sistem biblioteke (nasleđivanje, asocijacija, kompozicija, agregacija i zavisnost).

Pored dijagrama klasa, postoje i drugi strukturni dijagrami kao što su: dijagrami objekata, dijagrami komponenti, dijagrami rasporeda, dijagrami slučajeva upotrebe i drugi. Dijagram slučajeva upotrebe grafički prikazuje funkcionalnosti softverskog sistema kroz interakcije između korisnika (aktera) i osnovnih scenarija korišćenja. Na višem nivou apstrakcije, fokus je na manjem broju generalizovanih, poslovnih scenarija koji opisuju glavne uloge pojedinačnih grupa korisnika.

Dijagram sekvence.

Dijagram sekvence prikazuju ponašanje sistema u odnosu na vremenski redosled događaja, odnosno način na koji objekti međusobno komuniciraju kroz vreme u procesu izvršavanja neke aktivnosti. On jasno ilustruje koji subjekti ili objekti učestvuju u određenim koracima, kojim redosledom se aktivnosti odvijaju, kao i kako se odgovornosti prenose između njih.

Dijagram sekvence se prikazuje u dve dimenzije – vertikalna i horizontalna dimenzija (Slika 1.3). Vertikalna osa označava protok vremena (odozgo nadole), dok horizontalna osa predstavlja različite objekte uključene u interakciju. Svaki objekat je predstavljen vertikalnom linijom (životni vek objekta), na kojoj se aktivnosti objekta prikazuju uskim pravougaonicima. Aktivnosti počinju prijemom poruke i završavaju se njenom obradom. Ukoliko se objekat kreira tokom sekvence, to se označava porukom tipa *create*, a uništenje objekta prikazuje se simbolom „X" na kraju njegove linije aktivnosti.

Razmena poruka između objekata prikazana je horizontalnim strelicama. Poruke mogu biti sinhrone ili asinhrone, pri čemu sinhrone poruke impliciraju čekanje odgovora pre nastavka aktivnosti. Rezultat akcije se prikazuje povratnom strelicom, koja nije obavezna osim u situacijama gde je rezultat značajan za tok događaja.

Na primer, dijagram sekvence može detaljno ilustrovati proces autentifikacije korisnika u sistemu, uključujući slanje upita bazi podataka, validaciju kredencijala i generisanje autentifikacionog tokena.

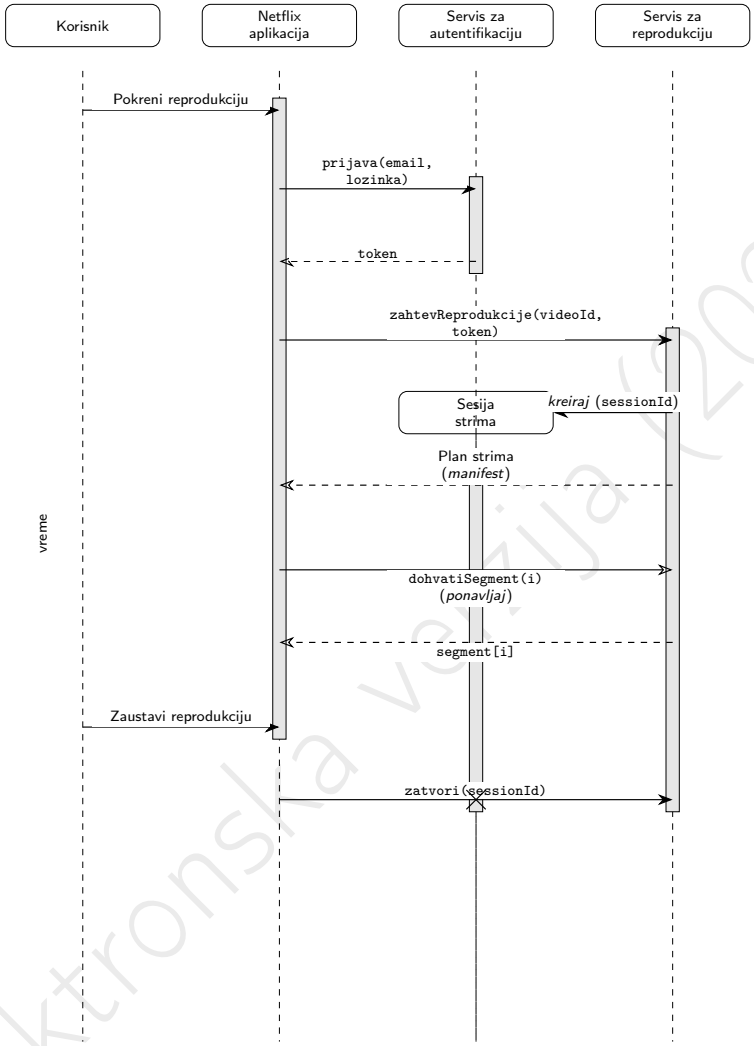
Dijagrami sekvence pružaju jasan vizualni prikaz interakcija, čime pomažu razumevanju tokova, definisanju interfejsa metoda, kao i pronalaženju eventualnih problema i optimizaciji sistema.

1.2 Metodologije razvoja softvera

I u teoriji i u praksi postoje mnoge metodologije razvoja softvera. U praksi su one često pomešane i često je teško striktno razvrstati stvarne projekte u postojeće metodologije. U nastavku je opisano nekoliko često korišćenih metodologija i ključnih ideja na kojima su zasnovane.

1.2.1 Metodologija vodopada

Metodologija vodopada je najstarija metodologija. Prvi put formalno je opisana 1970. godine, iako je praksa postojala i ranije, još od pedesetih godina prošlog veka. Vodopad metodologija je bila standard u industriji softvera tokom 70-ih i 80-ih godina posebno u velikim organizacijama kao što su vlade i velike korporacije. U strogoj varijanti ove tradicionalne metodologije na sledeću fazu u razvoju softvera prelazi se tek kada je jedna potpuno završena (slika 1.4):



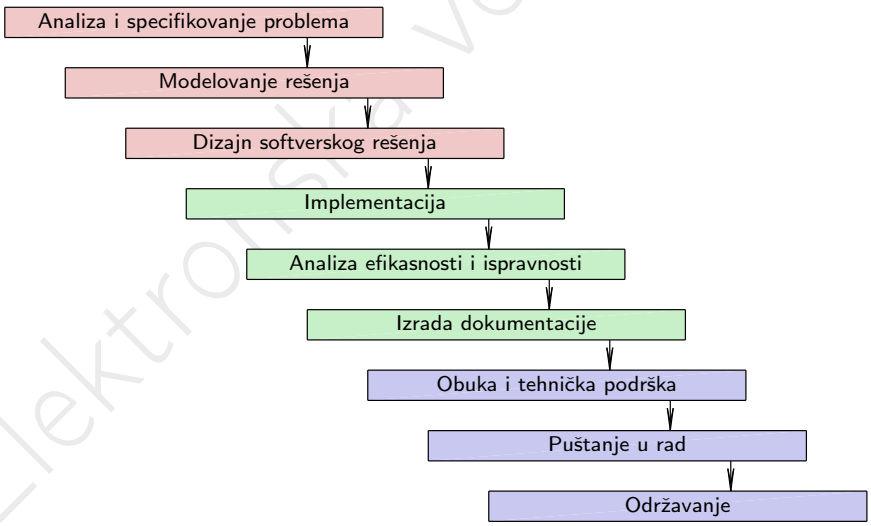
Slika 1.3: Primer dijagrama sekvence aplikacije za reprodukciju video sadrzaja.

- *zahtevi:* Skupljanje svih korisničkih zahteva, njihovo dokumentovanje i odobravanje od strane relevantnih strana;
- *dizajn:* Razrada tehničke arhitekture i specifikacija koje će se koristiti tokom implementacije;
- *implementacija:* Pisanje koda prema specificiranom dizajnu i zahtevima;
- *testiranje:* Verifikacija i validacija da sistem funkcioniše prema zahtevima;

- *integracija:* Kombinovanje različitih modula u jedinstven sistem i verifikacija njihove međusobne kompatibilnosti;
- *održavanje:* Faza nakon isporuke gde se ispravljaju greške i dodaju potrebne nadogradnje.

Ova metodologija se smatra primenljivom ako su ispunjeni sledeći uslovi:

- svi zahtevi poznati su unapred i njihova priroda ne menja se bitno u toku razvoja;
- zahtevi su u skladu sa očekivanjima svih relevantnih strana (investitori, korisnici, realizatori, itd.);
- zahtevi nemaju nerazrešene, potencijalno rizične faktore (na primer, rizike koji se odnose na cenu, tempo rada, efikasnost, bezbednost, itd);
- pogodna arhitektura rešenja može biti opisana i podrobno shvaćena;
- na raspolaganju je dovoljno vremena za rad u etapama.



Slika 1.4: Ilustracija za metodologiju vodopada

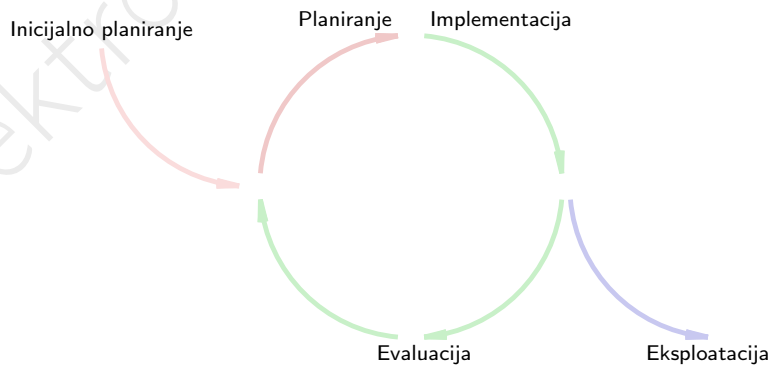
Ova metodologija koristi se obično u veoma velikim timovima. Detaljna dokumentacija za sve faze je neophodna što je korisno u velikim i kompleksnim projektima. Prednosti ove metodologije su jasna struktura jer osigurava da se svaki aspekt detaljno analizira pre nego što se krene sa sledećom fazom.

Metodologija ne predviđa modifikovanje prethodnih faza jednom kada su završene i ova osobina, krutost metodologije, predmet je najčešćih kritika. Često, klijentu je veoma teško da eksplicitno na početku projekta odredi sve zahteve. Izrada aplikacija često traje toliko dugo da se zahtevi promene u međuvremenu i završni proizvod više nije sasvim adekvatan a ponekad ni uopšte upotrebljiv. Korisnici (klijenti) se obično uključuju samo na početku (za zahteve) i na kraju (za prihvatanje), što može dovesti do neslaganja između korisničkih očekivanja i isporučenog proizvoda. Dodatno, postoje problemi i u fazi razvoja kada članovi tima moraju da čekaju izradu zadataka od kojih njihov rad zavisi. Vreme potrošeno čekajući da se zadatak odblokira nekada bude veće od produktivnog vremena.

Metodologija vodopada se danas retko koristi, mada može biti korisna kod projekta sa veoma jasnim i nepromenljivim zahtevima kao što su sistemi sa visokim nivom rigidnosti (na primer, medicinski uređaji ili avijacija). Poznato je da NASA koristi metodologiju vodopada u okviru izrade softvera za svemirski šatl, što je opravdano imajući na umu da su zahtevi i analiza za ovakav softver unapred striktno i jasno definisani. Postoje varijante metodologije vodopada (na primer, V-metodolologija) koje se češće koriste.

1.2.2 Metodologija iterativnog i inkrementalnog razvoja

U ovoj metodologiji, opisanoj prvi put šezdesetih i sedamdesetih godina prošlog veka (ilustrovanoj slikom 1.5), razvoj se sprovodi u iteracijama i projekat se gradi inkrementalno. Iteracije obično donose više detalja i funkcionalnosti, a inkrementalnost podrazumeva dodavanje jednog po jednog modula, pri čemu i oni mogu biti modifikovani ubuduće. U jednom trenutku, više različitih faza životnog ciklusa softvera može biti u toku. U ovoj metodologiji vraćanje unazad je moguće.



Slika 1.5: Ilustracija za iterativnu metodologiju

1.2.3 Metodologija rapidnog razvoja

U ovoj metodologiji (engl. rapid application development), opisanoj sedamdesetih i osamdesetih godina prošlog veka, faza planiranja svedena je na minimum zarad brzog dobijanja prototipova u iteracijama. Faza planiranja preklapa se sa fazom implementacije što olakšava izmene zahteva u hodu. Proces razvoja kreće sa razvojem preliminarnog modela podataka i algoritama, razvijaju se prototipovi na osnovu kojih se definišu, preciziraju ili potvrđuju zahtevi naručioca ili korisnika. Ovaj postupak ponavlja se iterativno, sve do završnog proizvoda. Aplikaciju prati vrlo ograničena dokumentacija.

Ova metodologija ponekad može dovesti do niza prototipova koji nikada ne dostižu do zadovoljavajuće finalne aplikacije. Čest izvor takvih problema su grafički korisnički interfejsi (engl. graphical user interface; GUI). Naime, korisnici napredak u razvoju aplikacije doživljavaju prvenstveno kroz napredak grafičkog korisničkog interfejsa. To podstiče programere, pa i vođe projekata, da se u prototipovima usredsređuju na detalje grafičkog interfejsa umesto na druge segmente aplikacije (kao što su, na primer, poslovni procesi i obrada podataka). Čak i mnogi razvojni alati privilegovano mesto u razvoju softvera daju razvoju grafičkih interfejsa. Ovakav razvoj aplikacije često dovodi do niza prototipova sa razrađenim korisničkim interfejsom, ali bez adekvatnih obrada koje stoje iza njega.

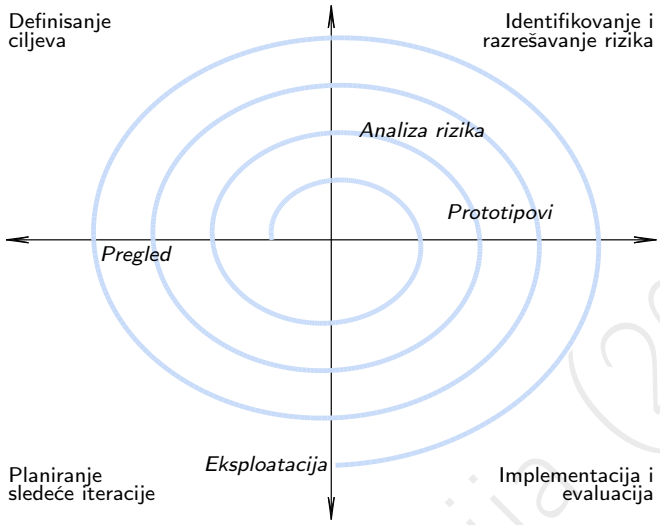
Ova metodologija pogodna je za razvoj softvera za sopstvene potrebe ili za potrebe ograničenog broja korisnika.

1.2.4 Spiralna metodologija

Ova metodologija (opisana prvi put krajem osamdesetih godina prošlog veka) kombinuje analizu rizika sa drugim metodologijama kao što su metodologija vodopada i metodologije iterativnog razvoja. Spirala koja ilustruje ovu metodologiju (prikazana na slici 1.6) prolazi više puta kroz faze kao što su planiranje, implementacija i evaluacija tekućeg verzije, kao i analiza rizika. Različite faze ne sprovode se istovremeno, već jedna za drugom. Prvi prototip pravi se na osnovu preliminarnog, pojednostavljenog dizajna i predstavlja samo aproksimaciju finalnog proizvoda. Na kraju svake iteracije, prototip se evaluira, analiziraju se njegove dobre i loše strane, profinjuje specifikacija za sledeću iteraciju i analiziraju se rizici (rizici koji se odnose na bagove, na cenu, tempo rada, efikasnost, bezbednost, itd). Na primer, planiranje dodatnog testiranja smanjuje rizik od neispravnog proizvoda, ali može da uveća cenu ili da nosi rizik zakasnelog izlaska na tržište. Ako neki rizik ne može biti eliminisan, naručilac mora da odluči da li se sa projektom nastavlja ili ne. Ukoliko se sa projektom nastavlja, ulazi se u sledeću iteraciju.

1.2.5 Agilna metodologija razvoja

Agilne metodologije su upotrebi od devedesetih godina prošlog veka a trenutno su verovatno najpopularnije.



Slika 1.6: Ilustracija za spiralnu metodologiju

Agilne metodologije

Stavlja fokus na zadovoljstvo korisnika i zato se podstiče *rana i inkremetalna isporuka softvera u vidu iteracija sa minimalnim dodavanjem funkcionalnosti u kratkim vremenskim intervalima (obično od jedne do četiri nedelje).*

Na ovaj način se teži minimizovanju rizika, kao što su bagovi, prekoračenje budžeta ili izmena zahteva. Dodatne smernice za razvoj daju prioritet isporuci naspram analize i dizajna (iako ove aktivnosti nisu obeshrabrene).

Radi se u malim, visokomotivisanim timovima koji su samoorganizovani i imaju kontrolu nad odlukama o projektu. Agilne metodologije zahtevaju permanentnu komunikaciju, poželjno uživo (zbog čega, međutim, ne ostaje mnogo pisanog traga o progresu niti pisane dokumentacije).

Agilne metode su razvijene u nastojanju da se prevaziđu uočene slabosti konvencionalnog razvoja softvera. Agilni razvoj može doneti važne prednosti, ali nije primenjiv na sve projekte, sve proizvode, sve ljude i sve situacije. U današnjem vremenu često je teško ili nemoguće predvideti kako će se softver razvijati kako vreme prolazi. Potrebe krajnjih korisnika se menjaju, a novi konkurentski proizvodi i rešenja pojavljuju se nekada iznenada, bez upozorenja. Zato, u mnogim situacijama nije moguće u potpunosti definisati specifikacije pre početka projekta i razvoj softvera mora biti dovoljno agilan da bi se prilagodio novim, promenjenim zahtevima. Sa druge strane, promene su skupe. Jedna od najprivlačnijih karakteristika agilnog pristupa je njegova sposobnost da smanji troškove promena tokom celog softverskog procesa.

Manifest agilne metodologije je osnovni dokument koji opisuje principe i vrednosti agilnog razvoja softvera. Kreiran je 2001. godine od strane grupe programera i ima dvanaest jednostavnih principa, kao što su: glavna mera napretka je upotrebljivost raspoloživog softvera, održivi razvoj, neprekidna usredsređenost na dobar dizajn, pojedinci i interakcije pre procesa i alata, funkcionalan softver pre obimne dokumentacije, odgovor na promene pre nego praćenje plana, itd.

Jedan od ciljeva ove metodologije je u ranom otkrivanju i ispravljanju propusta i neusklađenih očekivanja. Svaka iteracija odnosi se na minijaturni softverski proizvod sa svim uobičajenim fazama razvoja (koje se izvršavaju istovremeno). Svaku iteraciju potrebno je završiti na vreme i dobiti saglasnost naručioca. Za razliku od rapidne metodologije, u okviru koje se, u iteracijama, razvijaju nekompletni prototipovi, u agilnoj metodologiji, nakon nekih iteracija softver može biti isporučen naručiocu (ili na tržište) iako nema upotpunjenu funkcionalnost.

Agilna metodologija u mnogim je aspektima razvoja softvera uopštena, te postoji više vidova ove metodologije koji preciziraju neke njene aspekte, uključujući skram i ekstremno programiranje.

Skram

Skram

Skram (engl. scrum) je vid agilne metodologije u kojem se neposredna, praktična iskustva koriste u upravljanju izazovima i rizicima. Skram razvoj se sastoji od jednog ili više Skram timova, pri čemu svaki Skram tim čine tri uloge: vlasnik proizvoda, skram master i razvojni tim (Slika ??).

Softverski proizvod sve vreme se održava u stanju koje se potencijalno može isporučiti. Vreme je podeljeno u kratke intervale, „sprintove“, obično duge samo jedan mesec ili kraće i na kraju svakog sprinta svi akteri i članovi tima sastaju se da razmotre stanje projekta i planiraju dalje korake. Skram ima jednostavan skup pravila, zaduženja i sastanaka koji se, zarad jednostavnosti i predvidivosti, nikad ne menjaju. Postoje sastanci na početku i kraju svakog sprinta, ali i kratki, petnaestominutni dnevni sastanci („dnevni skram“).

SLIKA ROLA
SLIKA AKTIVNOSTI

Vlasnik proizvoda (engl. product owner) je koja poseduje celovito razumevanje projekta i autoritetom koja usmerava članove tima. Vlasnik proizvoda ima ključnu ulogu u osiguravanju sveukupnog uspeha rešenja koje se razvija ili održava. Njegova osnovna odgovornost je da odredi *šta* će biti razvijeno i *kojim redosledom*. U tom smislu, vlasnik proizvoda nadgleda, a to uključuje kreiranje, redefinisanje, procenu i prioritizaciju spiska zahteva (eng. scrum backlog). Time vlasnik proizvoda osigurava da se donose dobre finansijske

odluke na svim nivoima – od planiranja verzije proizvoda koja će biti objavljena, ali i sprinta pa do samog spiska zahteva proizvoda. Takođe, na kraju svakog sprinta, vlasnik proizvoda je odgovoran za odluku o tome da li će se finansirati sledeći sprint.

Kako bi ispunio ove odgovornosti, vlasnik proizvoda mora balansirati dve ključne uloge. S jedne strane, on predstavlja naručioce – kupce i korisnike i njegova uloga je da razume njihove potrebe i prioritete. Sa druge strane, on komunicira i sa razvojnim timom. On definiše kriterijume za prihvatanje funkcionalnosti koje se razvijaju i osigurava da se sprovode testovi kako bi se ti kriterijumi proverili. Tako da je on delom i poslovni analitičar, ali i tester.

Ključno je da vlasnik proizvoda proverava kriterijume za prihvatanje proizvoda tokom izvođenja sprinta, umesto da čeka na kraj sprinta. Funkcionalnosti se testiraju čim su završene i vlasnik proizvoda može brzo da identifikuje greške i nesporazume, i tako da omogućavajući timu da te probleme reši pre kraja sprinta.

Skram master (engl. scrum master) je osoba koja olakšava komunikaciju između vlasnika proizvoda i tima. Skram master pomaže timu i organizaciji da se pridržavaju skram vrednosti i principa. Oni vode tim kroz rešavanje problema i uklanjanje prepreka, radi poboljšanja skram procesa i štite tim od spoljašnjih ometanja. Skram master nije tradicionalan menadžer tima i ne vrši kontrolu.

Razvojni tim je samoorganizovana, multidisciplinarna grupa odgovorna za dizajn, izgradnju i testiranje proizvoda. Tim obično ima od pet do devet članova, uključujući osobe sa različitim veštinama kao što su programiranje, testiranje i dizajn korisničkog interfejsa. Ova raznolikost osigurava da tim poseduje sve potrebne veštine da isporuči visokokvalitetan, funkcionalan softver, bez potrebe da se oslanja na druge timove ili da prenosi posao, što često vodi do nesporazuma i kašnjenja.

Tokom izvršenja sprinta, tim sarađuje kako bi stavke iz liste zahteva pretvorio u potencijalno isporučive funkcionalnosti. Oni sami organizuju svoj rad, planiraju, upravljaju i realizuju zadatke, uz održavanje redovne komunikacije kroz dnevne sastanke. Ovi dnevni sastanci omogućavaju timu da pregleda napredak, prilagodi planove i osigura usklađenost sa ciljem sprinta. Na početku svakog sprinta, učestvuju u planiranju sprinta sa vlasnikom proizvoda i skram masterom kako bi odredili najvažnije stavke liste zahteva koje treba obraditi, osiguravajući usklađenost tima sa ukupnim ciljevima projekta.

Razvojni tim takođe igra ključnu ulogu u anaiziranju sprinta, gde ceo skram tim i zainteresovane strane procenjuju obavljeni posao i identifikuju oblasti za poboljšanje. Radom u kratkim, iterativnim ciklusima, tim može kontinuirano unapređivati proizvod, kao i svoje procese, osiguravajući stalno poboljšanje i prilagođavanje. Tim radi transparentno, otvoreno deli informacije i zajedno rešava probleme kako nastaju. Naglasak je na *zajedničkoj odgovornosti za obavljene zadatke*, jer rezultat rada zavisi od celog tima.

Ekstremno programiranje je vid agilne metodologije u kojem su posebno važne jednostavnost, motivacija i kvalitetni odnosi unutar tima. Programeri rade u parovima (dok jedan programer piše kôd, drugi pokušava da pronađe i ukaže na eventualne greške i nedostatke) ili u većim grupama, na kodu jednostavnog dizajna koji se temeljno testira i unapređuje tako da odgovara tekućim zahtevima. U ekstremnom programiranju, sistem je integrisan i radi sve vreme (iako svesno nema potpunu funkcionalnost). Svi članovi tima upoznati su sa čitavim projektom i pišu kôd na konzistentan način, te svako može da razume kompletan kôd i da radi na svakom delu koda. U ekstremnom programiranju, faze se sprovode u veoma malim koracima i prva iteracija može da dovede, do svesno nepotpune ali funkcionalne celine, već za jedan dan ili nedelju. Zahtevi se obično ne mogu u potpunosti utvrditi na samom početku, menjaju se tokom vremena, te naručilac treba da konstantno bude uključen u razvojni tim. Naručiocu se ne prikazuju samo planovi i dokumenti, već konstantno i (nekompletni, nesavršeni, ali funkcionalni) softver. Dokumentacija mora da postoji, ali se izbegava preobimna dokumentacija.

1.3 Ekspoloatacija

U fazi *eksploatacije* softver se *uvodi u rad* (eng. deployment) i održava u stabilnom, bezbednom i predvidljivom režimu. Uvođenje u rad obuhvata isporuku nove verzije u ciljno okruženje (npr. oblak ili lokalna infrastruktura), uz upravljanje konfiguracijom, zavisnostima i mogućnošću brzog *povratka na prethodnu verziju* ako se pojave problemi. Savremeni procesi se oslanjaju na automatizovane *cevovode izgradnje i isporuke* (eng. CI/CD): *kontinuirana integracija* automatizuje izgradnju i testiranje svake promene koda, dok *kontinuirana isporuka/puštanje u rad* omogućava bezbedno i često objavljivanje novih verzija, uz smanjenje rizika i vremena između promene i korisničke vrednosti.

Nakon puštanja u rad, neophodno je *praćenje nakon uvođenja* (eng. post-deployment monitoring): prikupljanje *podataka* , *dnevnika rada* (eng. logs) radi ranog otkrivanja degradacije performansi, regresija i incidenata, uz pragove, alarme i jasno definisane ciljeve kvaliteta usluge. Uvode se strategije oporavka, kontrola pristupa, redovno ažuriranje, kao i provera ranjivosti. Na kraju, *upravljanje krajem životnog veka* (eng. end-of-life management) obuhvata planirano gašenje ili migraciju sistema: pravovremenu komunikaciju sa korisnicima, arhiviranje i zaštitu podataka, usklađenost sa propisima i planiranje zamena, čime se smanjuju tehnički dug i operativni rizik.

1.4 Alati i tehnike korišćeni u razvoju softvera

1.4.1 Alati za upravljanje projektima

Upravljanje softverskim projektima obuhvata planiranje, organizaciju, praćenje i kontrolisanje svih aspekata procesa razvoja. To podrazumeva pažljivo upravljanje ljudima (timovima i pojedincima), procesima (metodologije, radni tokovi, standar-

di) i problemima (zahtevi, ograničenja, rizici) u cilju postizanja optimalnog balansa između kvaliteta, vremena i troškova. U okviru takvog pristupa, ključni pojmovi uključuju korišćenje softverskih metrika, adekvatnu procenu obima i složenosti projekta, identifikaciju i upravljanje rizicima, pravovremeno kreiranje rasporeda aktivnosti, kontinuirano održavanje i reinženjering postojećih softverskih rešenja.

Alati za upravljanje projektima nisu samo pasivni podsetnici, već i aktivni mehanizmi podrške za donošenje odluka i evaluaciju napretka. Pored klasičnih funkcionalnosti dodele zadataka, definisanja rokova i organizovanja timskog rada, ovi alati omogućavaju praćenje softverskih metrika (na primer, broj završenih zadataka u jedinici vremena, stepen pokrivenosti testovima, broj otvorenih bagova) koji se mogu koristiti za bolju kontrolu kvaliteta, povećanje produktivnosti i optimizaciju procesa razvoja. Na osnovu takvih metrika moguće je generisati procene troškova i trajanja projekta, kao i pratiti da li projekat napreduje u skladu sa planiranim rasporedom.

Postoje alati koji pomažu u rukovanju rizicima, pomažu da se ricici identifikuju, ali i da se izbegnu neplanirani troškovi, kašnjenja ili pad kvaliteta isporučenog softvera. Takvi alati prate istoriju razvoja i generišu projektne metrike koje se mogu koristiti za rano uočavanje trendova ili indikatora problema, što omogućava pravovremene akcije da bi se problemi spremičili ili ublažili.

Na tržištu se ističu brojni alati koji podržavaju sve ove aspekte projektnog menadžmenta. Među najpopularnijim su *Jira*, koja omogućava napredne mogućnosti izveštavanja, detaljnu integraciju sa sistemima za kontrolu verzija i metodologijama kao što su skram ili kanban, i *Trello*, intuitivna i vizuelno jednostavna platforma idealna za manje timove i agilne procese, gde je brz i lak pregled stanja zadataka od suštinskog značaja.

1.4.2 Sistemi za kontrolu verzija

Sistemi za kontrolu verzija (eng. *version control systems – VCS*) su ključni alati u razvoju softvera za praćenje promena, saradnju i upravljanje istorijom projekta. Među njima, Git je najpopularniji i najšire korišćen. Ovi sistemi su poznati i pod drugim imenima, recimo kao menadžer izvornog koda (eng. *Source Code Management – SCM*) ili kao sistem za kontrolu revizija (eng. *Revision Control System – RCS*). Bez obzira na terminologiju, cilj ostaje isti: *čuvati sadržaj, beležiti sve promene i omogućiti pristup različitim verzijama*.

Prilikom rada na projektu veoma je važno osigurati čuvanje rezervne kopije projekta jer može doći do gubitka podataka usled greške u kodu ili kvara diska. Održiva i pouzdana strategija čuvanja rezervene kopije projekta obično uključuje i kontrolu verzija omogućavajući programerima praćenje i upravljanje revizijama. **Repozitorijum** je centralizovano skladište gde se čuva sav izvorni kod projekta zajedno sa istorijom svih izmena. Repozitorijum sadrži:

Verzije datoteka – Sve datoteke i dokumenti projekta, ali i sve njihove različite verzije obeležene i datumom i vremenom kada je neka verzija napravljena.

Istorija izmena – Evidencija o svakoj promeni koja je napravljena u kodu, uključujući ko je napravio promenu, kada je promena napravljena, kao i opis promene. Ove izmene ujedno predstavljaju i formalan način komunikacije između članova tima koji rade na istom projektu.

Grane – Različite verzije koda koje se mogu razvijati paralelno i na taj način je omogućeno istovremeno razvijanje različitih funkcionalnosti ili verzija projekta.

Repozitorijum omogućava timovima da efikasno sarađuju, prate promene i vrše integraciju različitih delova koda na kontrolisan način.

U početku, upravljanje skladištem je bila ključna funkcija sistema za kontrolu verzija, smanjujući prostor na disku potreban za održavanje svih verzija. Kada se kreira nova verzija, čuva se samo razlika između nje i prethodne verzije (*delta*). U okviru repozitorijuma se čuvaju sve delte (izmene) nad projektom. Te delte, kada se primene na osnovnu verziju, ponovo kreiraju ciljnu verziju. Obično i na repozitorijumu i na lokalnom računaru na kom programer radi na projektu se čuva najnovija, poslednja verzija, a korišćenjem delti je moguće kreirati ranije verzije.

U timskom razvoju softvera, različiti članovi tima često rade na istoj komponenti istovremeno. U ovakvim okolnostima važno je izbeći sukobe između njihovih promena. Sistemi za kontrolu verzija koriste model javnog repozitorijuma i privatnog radnog prostora. Programeri preuzimaju komponente iz repozitorijuma u svoj privatni radni prostor, prave promene, a zatim ih vraćaju nazad. Ako više ljudi radi na jednoj komponenti, sistem upozorava ostale i osigurava da izmenjene komponente dobiju različite identifikatore verzija, kao i integraciju svih izmena u jedinstven kod. Dodatno, programerima je omogućeno da dodaju opise svojih izmena, tako da svi članovi tima mogu da razumeju zbog čega je neka promena nastala. Ovo je često poželjno, a zapravo u mnogim organizacijama i obavezan korak prilikom unosa izmena u kod.

Dodatno, sistemi za kontrolu verzija često imaju pridružene i dodatne alate za analizu koda. Na primer, alati koji formatiraju kod, alati za statičku analizu koda koji otkrivaju potencijalne greške, i alati za kontinualnu integraciju koji automatski prevode i testiraju kod. Kod koji ne može da se prevede ili ne prolazi sve testove obično se ne može ni postaviti na glavni ili javni repozitorijum.

Veoma često, deo sistema za kontrolu verzija je i proces revizije koda (eng. *code review*). Pre nego što se izmene unesu u glavni repozitorijum, drugi programeri, članovi tima, ili supervizori pregledaju predložene izmene. Tokom ove revizije, oni analiziraju kvalitet, efikasnost, sigurnost i održivost koda. Nakon što se utvrdi da je kod u skladu sa standardima projekta i da nema kritičnih grešaka, odobrava se integracija izmena u glavni repozitorijum.

Kao što je već rečeno, najpoznatiji i najšire korišćen sistem za kontrolu verzija je **Git**. Git je *distribuirani sistem* za kontrolu verzija, što znači da svaki korisnik ima punu kopiju istorije repozitorijuma, što omogućava rad van mreže i veću otpornost na greške. Pored Gita, postoje i drugi sistemi za kontrolu verzija, kao što su Subversion (SVN) i Mercurial. SVN je centralizovani sistem, što znači da postoji

jedan centralni repozitorijum kojem korisnici pristupaju. Ovo olakšava centralizovanu kontrolu i nadzor, ali može biti manje fleksibilno u poređenju sa distribuiranim sistemima kao što je Git. Mercurial je takođe distribuirani sistem, sličan Gitu, ali je Git poznat po jednostavnosti korišćenja i brzini.

Linux kernel je razvijan pomoću BitKeepera, komercijalnog alata za kontrolu verzija. Godine 2005. kompanija koja poseduje BitKeeper odlučila je da više ne dozvoljava besplatno korišćenje alata. Linux zajednica je morala da pronađe drugačije rešenje. Linus Torvalds je tražio besplatan alat koji bi zadovoljio sve potrebe za razvoj Linux kernela, pa je osmislio i razvio Git zajedno sa grupom programera. Git je morao da zadovolji nekoliko ključnih zahteva:

- **Distribuiran razvoj:** Omogućiti paralelan i nezavisan razvoj u privatnim repozitorijumima bez stalne potrebe za sinhronizacijom sa centralnim repozitorijumom. Programeri mogu raditi na različitim lokacijama, čak i van mreže, uz istovremeno omogućavanje hiljadama programera da rade na istom projektu. Svaki repozitorijum ima kompletnu istoriju svih promena.
- **Brzina i efikasnost:** Da bi se uštedelo na prostoru i skratilo vreme prenosa, korišćene su kompresije i đelta"tehnika. Distribuirani model umesto centralizovanog modela osigurao je da kašnjenje mreže ne ometa svakodnevni razvoj.
- **Pouzdanost:** Pošto je Git distribuirani sistem za kontrolu revizija, važno je imati apsolutnu sigurnost da je integritet podataka očuvan. Git koristi kriptografsku hash funkciju SHA1 (eng. *Secure Hash Function*) za imenovanje i identifikaciju objekata u svojoj bazi podataka, što osigurava integritet i poverenje u distribuirane repozitorijume.
- **Preuzimanje odgovornosti:** Ključni aspekt sistema za kontrolu verzija je znati ko je promenio datoteke i, ako je moguće, zašto. Git nameće vođenje evidencije o izmenama pri svakom menjanju datoteke.
- **Nepromenljivost:** Git-ova baza podataka repozitorijuma sadrži objekte koji su nepromenljivi. To znači da, kada su kreirani i smešteni u bazu podataka, ne mogu biti izmenjeni. Dizajn Git baze podataka znači da je cela istorija koja se nalazi unutar baze podataka za kontrolu verzija takođe nepromenljiva.
- **Atomske transakcije:** Niz promena koje treba da se obave se obavljaju sve zajedno ili se uopšte ne obavljaju. To znači da ako prilikom unosa izmena se desi da mrežna veza se prekine ili server prestane da radi, izmene neće biti delimično primenjene i na taj način ostaviti datoteku u neispravnom stanju.
- **Grane:** Omogućiti paralelno razvoj različitih grana u okviru kojih se mogu razvijati različite funkcionalnosti. Omogućiti i spajanje grana u jednu.
- **Slobodan za korišćenje.**

Ima mnogo različitih načina za korišćenje Gita. U ovoj knjizi Git se koristi putem komandne linije. Komandna linija je jedino mesto gde možete pokrenuti sve Git

komande – većina GUI-ja implementira samo delimičan skup Git funkcionalnosti radi jednostavnosti i izbor grafičkog klijenta je stvar ličnog ukusa.

Osnovni pojmovi i tokovi rada u Git-u

Repozitorijumi (eng. repositories). Repozitorijum je „projekat sa memorijom“: pored samih fajlova (koda, slika, dokumentacije), **on čuva i istoriju izmena** — ko je šta menjao i kada, kao i informacije o granama i verzijama. Te informacije Git smešta u poseban direktorijum `.git`, koji se nalazi unutar projekta, pa se zato repozitorijum može posmatrati kao običan folder proširen mehanizmom za praćenje promena.

Repozitorijum može postojati lokalno (na računaru) i udaljeno (na zajedničkom serveru). Lokalna kopija omogućava rad i bez interneta, dok udaljeni repozitorijum (eng. remote) služi da članovi tima razmenjuju promene i imaju zajedničko mesto na kome se čuva „zvanična“ verzija projekta. Novi repozitorijum se pravi komandom `git init`, a postojeći se preuzima komandom `git clone`.

Revizije (eng. commits). Revizija (eng. commit) je zapis jedne smislene promene u projektu: predstavlja trenutak u kome kažemo „ovo je sada novo, stabilno stanje u odnosu na malopre“. Svaka revizija ima kratku poruku koja opisuje šta je urađeno, podatke o autoru i vremenu, kao i jedinstveni identifikator (heš) pomoću koga se tačno zna na koju se reviziju mislilo. Niz revizija čini istoriju projekta, pa se u svakom trenutku može videti kako je projekat nastajao i po potrebi vratiti na neko ranije stanje.

Pravljenje revizije se obično radi u dva koraka. Najpre se izaberu fajlovi (ili delovi izmena) koje želimo da uđu u sledeću reviziju komandom `git add`, a zatim se ta promena zabeleži komandom `git commit`. Ovaj pristup pomaže da revizije budu „čiste“ i tematski jasne, umesto da se mnogo nepovezanih izmena nađe u jednom zapisu. Istorija se pregledava komandom `git log`, dok se razlike između verzija ili lokalnih izmena vide komandom `git diff`.

Grane (eng. branches). Grana je „radna verzija“ projekta u kojoj se razvija određena funkcionalnost, ispravlja greška ili radi eksperiment, bez uticaja na glavnu, stabilnu verziju (najčešće granu `main`). Ideja je jednostavna: dok je posao u toku i još nije spreman, on se drži odvojeno; kada bude završen i proveren, promene se prenose u glavnu granu postupkom spajanja (eng. merge). Na taj način više ljudi može da radi paralelno na različitim delovima projekta, a da se stabilna verzija ne „kvari“ nedovršenim izmenama.

Nova grana se obično pravi za konkretan zadatak. Kreiranje i prelazak na novu granu radi se komandom `git switch -c naziv`, dok se spisak postojećih grana može videti komandom `git branch`.

Spajanje (eng. merge). Spajanje je postupak kojim se promene iz jedne grane prenose u drugu (najčešće iz grane u kojoj se razvijala funkcionalnost u glavnu

granu `main`). U velikom broju slučajeva Git može sam da objedini izmene, jer prepoznaje koji delovi fajlova potiču iz koje grane i kako da ih spoji u jedinstvenu verziju.

Problem nastaje kada su u obe grane menjani isti redovi (ili vrlo bliski delovi) istog fajla, ali na različite načine. Tada Git ne može da pogodi šta je ispravno rešenje i prijavljuje *konflikt* (eng. conflict), koji programer mora ručno da razreši tako što izabere ili kombinuje odgovarajuće delove teksta. Spajanje se pokreće komandom `git merge`, a rezultat je ažurirana ciljna grana i zabeleženo da je do spajanja došlo. U praksi se spajanje u stabilnu granu radi oprezno, najčešće tek nakon pregleda promena i proverenog pokretanja testova.

Zahtevi za spajanje (eng. pull requests). Zahtev za spajanje (eng. pull request) je postupak kojim se na platformama kao što su GitHub ili GitLab predlaže da se promene iz jedne grane uključe u drugu (najčešće u `main`). Autor najpre objavi svoju granu na zajedničkom repozitorijumu, a zatim otvara zahtev za spajanje kako bi ostali mogli da vide šta je promenjeno, da komentarišu i da provere da li je rešenje dobro.

Suština zahteva za spajanje je kontrolisana provera pre integracije: promene se pregledaju (pregled koda), po potrebi se diskutuje pristup, a zatim se obično pokrenu automatske provere, kao što su testovi i alati koji upozoravaju na potencijalne greške. Tek kada su provere zadovoljene, promene se spajaju u ciljnu granu. Na ovaj način se smanjuje verovatnoća da se u stabilnu verziju projekta unesu greške i povećava se transparentnost timskog rada.

Reorganizacija istorije (eng. rebase). Rebase (eng. rebase) je postupak kojim se „sredi“ istorija jedne grane tako da se ona nasloni na najnovije stanje glavne grane (`main`), kao da je rad na toj grani krenuo od tog novijeg trenutka. Najjednostavnije rečeno, rebase služi da svoju granu uskladiš sa najnovijim promenama iz `main` i da istorija izgleda urednije (češće kao jedna ravna linija), bez dodatnog zapisa o spajanju.

Ova operacija se radi komandom `git rebase`. Važno je znati da rebase može da promeni „obeležja“ postojećih revizija, pa se zato uglavnom koristi dok grana još nije podeljena sa drugima (dok je lokalna ili dok na njoj ne radi više ljudi). Ako je grana već objavljena i drugi su je preuzeli, rebase može da napravi zbrku pri usklađivanju, pa se tada koristi samo uz jasan dogovor u timu.

Tokovi rada (eng. workflows) i tipični obrasci. Git je jedan alat, ali se u različitim timovima koristi na različite načine. Zato se unapred dogovara *tok rada*: koje grane postoje, kako se naziva grana za novi zadatak, kada se promene spajaju u glavnu verziju, ko i kako pregleda promene i koje provere (npr. testovi) moraju da prođu. Dobar tok rada je važan jer sprečava haos: ako svako radi „po osećaju“, brzo se dobije istorija koju je teško razumeti i još teže održavati.

Centralizovani tok rada (eng. centralized workflow) je najjednostavniji: postoji jedna glavna grana (najčešće `main`) i svi rade tako što redovno preuzimaju najnovije

stanje i objavljuju svoje izmene. Ovaj model je lak za učenje i dobar za manje timove, ali u većim timovima obično zahteva dodatna pravila, na primer da se ne sme direktno spajati u `main`, već da promene moraju prvo da prođu kroz zahtev za spajanje.

Tok sa granama funkcionalnosti (eng. feature branch workflow) danas je najčešći u praksi. Za svaki zadatak otvara se posebna grana, u njoj se radi i prave revizije, a zatim se otvara zahtev za spajanje (eng. pull request) ka `main`. Pre spajanja se promene pregledaju i proveravaju, pa tek onda ulaze u glavnu granu. Da bi se izbegli veliki konflikti, grana se povremeno usklađuje sa najnovijim stanjem `main` (bilo spajanjem *merge* (eng. merge), bilo reorganizacijom istorije *rebase* (eng. rebase)).

Gitflow je formalniji model koji uvodi više stalnih grana, najčešće `main` za stabilna izdanja i `develop` za tekući razvoj, uz posebne grane za pripremu izdanja i hitne ispravke. Koristan je kada se izdanja objavljuju u jasno planiranim ciklusima i kada je važno održavati strogu strukturu. Međutim, za timove koji objavljuju veoma često, ovaj model može biti suviše složen, pa se tada obično bira jednostavniji tok rada sa granama funkcionalnosti.

Primer 1.1. Jedan tipičan tok rada u praksi

U nastavku je prikazan objedinjeni primer rada u timu: preuzimanje repozitorijuma, rad u grani funkcionalnosti, beleženje revizija, sinhronizacija sa glavnom granom, objavljivanje grane i integracija kroz pull request. Primer koristi savremene komande `switch` i `restore`; u starijim vodičima često se sreće `checkout`.

```
# 1) Preuzimanje repozitorijuma (lokalna kopija sa celom istorijom)
git clone https://example.com/projekat.git
cd projekat

# 2) Pregled stanja i istorije
git status
git log --oneline --decorate --graph --max-count=10

# 3) Kreiranje grane za novu funkcionalnost i prelazak na nju
git switch -c feature/validacija-unosa

# (ovde izmenite fajlove u editoru, npr. dodate validaciju u
↪ src/input.cpp)
git diff                          # pregled lokalnih izmena

# 4) Priprema izmena i beleženje revizije
git add src/input.cpp include/input.h
git commit -m "Dodaj osnovnu validaciju korisnickog unosa"

# 5) Objavljivanje grane na udaljenom repozitorijumu
git push -u origin feature/validacija-unosa

# 6) U međuvremenu se main promenio: preuzimamo novosti bez
↪ automatskog spajanja
git fetch origin

# 7 Postavljanje grane na najnoviji main radi linearnije istorije
git rebase origin/main

# Ako nastane konflikt:
#   - otvorite konfliktne fajlove, razresite oznake <<<<<<< =======
↪ >>>>>>>
#   - zatim:
git add src/input.cpp
git rebase --continue

# Nakon rebase-a objavljujemo novu istoriju:
git push

# 8) Pull request (na platformi): otvaranje PR-a feature/... -> main,
↪ pregled i automatske provere
# (nije Git komanda; radi se kroz GitHub/GitLab interfejs)

# 9) Nakon sto je PR spojen, lokalno osvezavanje glavne grane
git switch main
git pull
```

1.5 Savremeni trendovi i tehnologije u razvoju softvera

1.5.1 Distribuirani softverski sistemi

Distribuirano softversko inženjerstvo bavi se razvojem, održavanjem i isporukom softverskih sistema koji se izvršavaju na više računara ili geografski distribuiranih čvorova. Za razliku od centralizovanih sistema, komponente distribuiranih sistema rade paralelno i komuniciraju putem mrežnih protokola, što omogućava skaliranje, povećanu dostupnost i otpornost na kvarove. Danas su gotovo svi veliki sistemi distribuirani, ali ovaj pristup uvodi dodatne izazove u pogledu komunikacije, sinhronizacije, bezbednosti i upravljanja odvojenim komponentama.

Distribuirani sistemi su složeniji od centralizovanih, što ih čini težim za dizajn, implementaciju i testiranje. Performanse distribuiranog sistema ne zavise samo od brzine izvršavanja pojedinačnog procesora, već i od mrežne propusnosti, opterećenja mreže i brzine svih računara uključenih u sistem.

Distribuirani sistemi imaju nekoliko osnovnih osobina:

- **Deljenje resursa:** Više računara na mreži zajednički koriste hardverske i softverske resurse (npr. diskove, štampače, baze podataka i kompajlere), što omogućava da se optimalno iskoristi dostupna infrastruktura.

- **Paralelno izvršavanje (konkurentnost):** Više procesa se izvršava istovremeno na različitim čvorovima, čime se povećava brzina obrade podataka i efikasnost sistema. Često je neophodna međusobna komunikacija procesa putem poruka ili signala. Međutim, komunikacija je ponekad *asinhrona*, što može dovesti do kašnjenja – sistem može čekati dolazak podataka ili se, u slučaju prevelike količine dolaznih informacija, može se pojaviti čekanje na svoj red za obradu.

- **Skalabilnost:** Distribuirani sistemi se mogu skalirati *dodavanjem novih resursa*. Skalabilnost je ključna karakteristika distribuiranih sistema koja omogućava proširenje kapaciteta sistema u skladu sa rastućim zahtevima korisnika. Sistem se mora projektovati tako da se njegov kapacitet može povećavati.

  Jedan način skaliranja je *povećanje kapaciteta postojećih čvorova* (eng. scaling up). U ovom pristupu zamenjuju se ili unapređuju resursi unutar postojećih čvorova (na primer, povećanjem memorije ili brzine procesora). Iako se na ovaj način postiže veća moć obrade, ova metoda može biti ograničena hardverskim kapacitetom i često je skuplja.

  Drugi način skaliranja sistema je *dodavanje novih čvorova* (eng. scaling out). Ovde se u mrežu dodaju novi čvorovi kako bi se povećala ukupna obrada i raspodelilo opterećenje. Dodavanje novih čvorova je često isplativiji i omogućava veću fleksibilnost. Zahteva pažljivo projektovanje sistema u početnoj fazi, uzimajući u obzir mogućnost budućeg proširenja. Prilikom dodavanja, neophodno je ravnomerno rasporediti opterećenje i sinhronizovati rad svih čvorova, kako novih, tako i postojećih. Međutim, ponekad, sama mrežna

arhitektura i ograničenja u propusnosti mreže mogu uticati na efikasnost skaliranja.

* **Otpornost na greške:** Zahvaljujući upotrebi dodatnih resursa i mogućnosti replikacije podataka, distribuirani sistemi mogu nastaviti da funkcionišu i u slučaju otkaza pojedinačnih čvorova. Ukoliko neki čvor prestane sa radom, često se implementiraju mehanizmi za automatsko preusmeravanje zahteva kako bi se izbegao potpuni prekid rada.

Arhitektura distribuiranih sistema. Distribuirani sistemi kojima se pristupa preko interneta najčešće se organizuju prema *klijent–server modelu*. U ovim sistemima, korisnik komunicira sa aplikacijom koja se izvršava na lokalnom uređaju (na primer, putem web pretraživača ili mobilne aplikacije), dok daljinski server pruža neophodne usluge, poput pristupa web sadržajima. Ovakva arhitektura omogućava jasno razdvajanje prezentacije informacija od same obrade podataka, što doprinosi boljoj skalabilnosti i upravljanju sistemom.

Više serverskih procesa može se izvršavati na istom procesoru, ali se često serveri implementiraju kao multiprocesorski sistemi, gde se zasebna instanca serverskog procesa pokreće na svakom računaru. Softver za balansiranje opterećenja raspoređuje zahteve klijenata ravnomerno na sve servere, čime se omogućava obrada većeg broja zahteva od pojedinačnih klijenata.

Arhitektura	Karakteristike
Master – rob arhitektura (eng. master–slave)	Ovaj model se primenjuje u sistemima gde je kritično zadovoljiti stroga vremenska ograničenja i odreagovati na zahteve u realnom vremenu. Glavni čvor (master) raspoređuje zadatke potčinjenim čvorovima (rob), što omogućava precizno planiranje i brzu obradu. Primer takvog sistema je upravljanje semaforima u saobraćaju.
Dvoslojna (eng. two-tier) *klijent–server arhitektura*	U ovom jednostavnom modelu, aplikacija se sastoji od jednog centralnog servera i brojnog skupa klijenata. Server obrađuje zahteve, dok klijenti komuniciraju direktno sa njim. Zbog centralizacije podataka, komunikacija između klijenta i servera je često enkriptovana kako bi se obezbedila sigurnost. Tipičan primer primene je bankomat sistem, gde ATM uređaji (klijenti) pristupaju centralnoj bazi podataka na računaru (server).
Višeslojna (eng. multi-tier) *klijent–server arhitektura*	Pogodna je za velike sisteme sa visokim brojem transakcija, gde se podaci integrišu iz više izvora, a često se primenjuje i dodatni integracioni server radi objedinjavanja distribuiranih podataka. Ovaj model deli aplikaciju na više logičkih slojeva – prezentacioni sloj, sloj za upravljanje podacima, aplikacioni sloj i sloj baze podataka. Svaki sloj može da se izvršava na zasebnom čvoru, što omogućava ravnomernu raspodelu opterećenja i poboljšava skalabilnost sistema. Koristi se kod aplikacija velikih razmera koje imaju nekoliko stotina ili hiljada klijenata.

Decentalizovana arhitektura (eng. peer-to-peer, P2P)	Koristi se kada klijenti razmenjuju lokalno sačuvane informacije, a uloga centralnog servera se svodi na povezivanje klijenata. Za razliku od klijent–server modela, gde postoji jasna podela između servera (pružalaca usluga) i klijenata (primaoca usluga), P2P sistemi su decentralizovani — svaki čvor može da obavlja računanje i skladišti podatke. Primeri ovakvih sistema su kriptovalute (Bitcoin), sistemi za razmenu datoteka (BitTorrent), kao i razni servisi za razmenu poruka. U principu, u P2P sistemima ne postoji striktna razlika između klijenata i servera, jer svaki čvor pokreće kopiju aplikacije koja sadrži komunikacione protokole i standarde. Primenjuje se kada je sistem računarski intenzivan i obradu je moguće podeliti na veliki broj nezavisnih operacija. Takođe i u situacijama kada primarna funkcija sistema jeste razmena informacija među pojedinačnim računarima, bez potrebe za centralizovanim upravljanjem podacima. Prednosti ove arhitekture su visoka redundantnost, otpornost na otkaze i fleksibilnost u korišćenju raspoloživih resursa. Međutim, nedostaci uključuju mogućnost dupliranja obrade istih zahteva i značajano komunikaciono opterećenje čvorova.

Komunikacioni modeli. Komunikacija između komponenti distribuiranog sistema može se ostvarivati na dva osnovna načina:

- **Proceduralna interakcija:** Podrazumeva da jedan računar poziva poznatu uslugu koju nudi neki drugi računar i (obično) čeka da ta usluga bude isporučena. Realizovana putem poziva udaljenih procedura (RPC) ili, u slučaju Jave, udaljenih metoda (RMI). U RPC, jedna komponenta poziva drugu komponentu kao da je lokalna procedura ili metoda. Ovaj model zahteva da i pozivajući i pozvani entitet budu istovremeno dostupni, što može predstavljati problem u slučaju privremene nedostupnosti neke komponente.

- **Komunikacija zasnovana na porukama:** Poruke se smeštaju u redove čekanja dok primalac ne postane dostupan, čime se omogućava asinhrona komunikacija. Međusoftver (eng. middleware), odnosno softver koji se nalazi 'između' operativnog sistema i aplikacija, igra ključnu ulogu u upravljanju

komunikacijom, transformacijom podataka i održavanju konzistentnosti između različitih komponenti.

U suštini, RPC ima iste zahteve kao i lokalni poziv procedure ili metode. Nasuprot tome, u pristupu zasnovanom na porukama, moguće je tolerisati nedostupnost, jer poruka jednostavno ostaje u redu dok primalac ne postane dostupan. Dalje, nije neophodno da pošiljalac i primalac budu svesni postojanja jedan drugog; oni jednostavno komuniciraju sa međusoftverom, koji je odgovoran za prosleđivanje poruka odgovarajućem sistemu.

Implementacione tehnologije i bezbednost. U savremenom razvoju distribuiranih sistema koriste se različite tehnologije i programski jezici koji pokrivaju sve aspekte izgradnje aplikacija – od baze podataka do korisničkog interfejsa. Na primer, SQL se široko koristi za rad sa relacionim bazama podataka, dok se JAVA često primenjuje za razvoj serverske logike. Sa druge strane, za razvoj korisničkog interfejsa popularni su ANGULAR, REACT i JAVASCRIPT, koji omogućavaju kreiranje dinamičnih i responzivnih web aplikacija.

Takođe, GO je postao vrlo cenjen jezik u razvoju distribuiranih sistema zbog svoje jednostavnosti, visokih performansi i podrške za konkurentno programiranje. Njegova ugrađena podrška za paralelno izvršavanje čini ga idealnim za izgradnju sistema koji zahtevaju visoku propusnost i efikasno korišćenje resursa.

Ključnu ulogu u održavanju doslednosti i pouzdanosti distribuiranih sistema igraju algoritmi za postizanje konsenzusa, među kojima je RAFT jedan od najpoznatijih. *Postizanje konsenzusa* znači da grupa servera, uprkos mogućim otkazima pojedinačnih čvorova, mora da se usaglasi oko zajedničkog stanja sistema – na primer, o redosledu operacija nad bazom podataka. RAFT osigurava da svi čvorovi imaju isti pregled podataka, što je od presudnog značaja za integritet i pouzdanost distribuiranih aplikacija.

Projektovanje distribuiranih sistema mora da se fokusira i na bezbednost. Sistem mora da bude otporan na različite vrste napada, uključujući presretanje podataka, uskraćivanje usluga (DoS), neovlašćene izmene i ubacivanje lažnih podataka. Implementacija enkripcije, autentifikacije i kontrola pristupa predstavlja osnovu za zaštitu podataka i komunikacionih kanala. Više tačaka pristupa i brojni komunikacioni kanali zahtevaju napredne sigurnosne mehanizme kako bi se zaštitili podaci i transakcije u sistemu.

Programiranje u oblaku (eng. cloud computing) i distribuirani sistemi. Programiranje u oblaku predstavlja prirodan nastavak distribuiranog softverskog inženjerstva, omogućavajući korišćenje udaljenih resursa bez potrebe za izgradnjom sopstvene infrastrukture. Na primer, umesto ulaganja u sopstveni prostor za skladištenje podataka, moguće je koristiti usluge poput Amazon S3, dok se algoritmi veštačke inteligencije mogu izvršavati na udaljenim serverima. Takođe, blokčejn tehnologije koriste distribuirane resurse za obradu transakcija i verifikaciju podataka, čime se omogućava sigurnost i transparentnost.

Ovakav pristup smanjuje troškove infrastrukture, jer se resursi dinamički prilagođavaju trenutnim potrebama, a timovi za razvoj distribuiranih sistema mogu brže i fleksibilnije implementirati nove funkcionalnosti. Ovo omogućava agilni razvoj softvera, gde se resursi lako povećavaju ili smanjuju u zavisnosti od zahteva projekta, bez dodatnih ulaganja u fizički hardver.

Distribuirano softversko inženjerstvo predstavlja kompleksan, ali neophodan pristup u savremenom IT svetu, omogućavajući visok nivo otpornosti, skalabilnosti i efikasnosti. Iako nudi brojne prednosti, projektovanje ovakvih sistema zahteva pažljivo balansiranje između performansi, bezbednosti i pouzdanosti. Savremene implementacije distribuiranih sistema oslanjaju se na napredne komunikacione modele i tehnologije, čime se postiže robustan temelj za buduće inovacije u oblasti softverskog inženjerstva.

1.5.2 Mikroservisno orijentisan razvoj softvera

Komponentno softversko inženjerstvo (eng. Component Based Sofware Engeeneering, CBSE), razvijeno tokom 1990-ih, formalizovalo je ideju da se složeni sistemi grade spajanjem „crnih kutija“ – komponenti sa jasno definisanim zadacima i eksplicitnim ulazno-izlaznim interfejsima. Ovakav pristup podsticao je ponovnu upotrebu koda, ali je često otežavao nezavisan razvoj pojedinačnih delova aplikacije i skaliranje samo onih komponenti kojima je to bilo potrebno. CBSE se prirodno nadovezuje na principe objektno orijentisanog programiranja, dok ih arhitektonski stil mikroservisa (engl. microservices) dodatno proširuje uvodeći autonomne, samostalno isporučive servise koji međusobno komuniciraju mrežnim protokolima, čime se postiže veća agilnost i skalabilnost sistema. *Umesto razvoja jedne velike (monolitne) aplikacije, sistem se razlaže na skup manjih, međusobno nezavisnih servisa.* Ključna razlika u odnosu na bibliotke ili klasične komponente jeste to što je svaki servis samostlna jedinica sa sopstvenim ciklusom razvoja, koja se može isporučiti nezavisno u odnosu na ostale servise.

SLIKA - PRIMER (situaational awareness)

Tri ključna principa mikroservisne arhitekture su:

- *Ograničeni kontekst* (eng. bounded context) – svaki servis ima jasno definisanu odgovornost i granice. Servis je ograničen na jednu poslovnu potrebu (funkcionalnost) i lako je razumeti njegovu svrhu, ali i pronaći i menjati željene funkcionalnosti.

- *Veličina* – fokus je na maloj veličini servisa. Čim servis postane prevelik (sa velikim brojem funkcionalnosti), treba ga podeliti na manje servise.

- *Nezavisnost* – svaki servis je nezavistan i može se razvijati, testirati, implementirati i skalirati nezavisno od drugih servisa. Ovo omogućava timovima da rade na različitim servisima istovremeno bez međusobnog ometanja. Jedina bitna zavisnost je komunikacija između servisa, koja se obično ostvaruje putem mrežnih protokola.

Obratimo pažnju da su mikroservisna arhitektura omogućava lakšu izgradnju distribuiranih sistema jer svaki servis se može naći na različitim računarima, a komunikacija između servisa se obavlja putem mrežnih protokola.

Mikroservisna arhitektura omogućava timovima da koriste različite tehnologije i programske jezike za različite servise. Na primer, ako neki servis zahteva intezivna računanja, može se implementirati u C++, dok se drugi servis koji je okrenut krajnjim korisnicima može implementirati u JavaScript jeziku. Sa druge strane, u monolitnim sistemima, korišćenje različitih tehnologija je otežano jer bi cela aplikacija morala da bude napisana u jednoj tehnologiji.

Mikroservisi se često isporučuju u kontejnerima, na primer, koristeći Docker. Kontejner sadrži sve što je potrebno za pokretanje servisa – izvršni kod servisa, ali i biblioteke i konfiguracije koje su potrebne za njegovo pokretanje. Na ovaj način, servisi se mogu pokrenuti na bilo kojem računaru koji ima instaliran Docker, bez potrebe za dodatnim konfiguracijama.

Mnoge kompanije su usvojile mikroservisnu arhitekturu, uključujući Netflix, Amazon, Google, LinkedIn i dr.

Mikroservisna arhitektura omogućava i veću robusnost sistema. Ako neki servis prestane da radi, to ne utiče na rad drugih servisa. Na primer, ako servis za autentifikaciju korisnika prestane da radi, to neće uticati na rad servisa za pretragu proizvoda. Ovo omogućava da sistem se može koristiti iako neki servisi nisu dostupni.

Mikroservisi nude i neke izazove. Na primer, komunikacija između servisa može biti spora i može doći do kašnjenja u radu sistema. Može se desiti da različiti servisi zahtevaju akciju od jednog istog servisa, što može dovesti do preopterećenja tog servisa ili nemogućnost da se zahtev ispuni. Testiranje i debagovanje mikroservisa može biti složenije nego kod monolitnih sistema, jer je potrebno testirati svaki servis posebno, ali i testirati interakcije između servisa. Ponekad mikroservisi dele istu bazu podataka (mada prema preporukama svaki mikroservis ima svoju bazu, ali nekada nije moguće realiovati takvu arhitekturu), što može dovesti do problema sa konzistentnošću podataka. U ovim situacijama najčešće se uvodi dodatni mikroservis koji se brine o upravljanju podacima.

1.5.3 Ugrađeni softver

Ugrađeni softver (eng. embedded softver) je deo integrisanog hardversko-softverskog sistema, dizajniran da upravlja uređajem i reaguje na događaje iz njegovog okruženja u realnom vremenu. Za razliku od standardnih softverskih aplikacija, ugrađeni softver radi u uslovima ograničenih resursa poput memorije, procesorske snage i energije. Često se nalazi u uređajima poput automobila, kućnih aparata, medicinskih uređaja, telekomunikacione opreme i drugih specijalizovanih sistema, gde je *pouzdanost i efikasnost ključna*.

Ključna karakteristika ugrađenih sistema jeste *rad u realnom vremenu*, što znači da sistem mora da odgovori na događaje unutar striktno definisanih vremenskih rokova. Ako reakcija sistema nije dovoljno brza, može doći do ozbiljnih posledica –

na primer, sistem za kočenje automobila mora momentalno da reaguje na komandu kočenja da bi se sprečila nesreća.

Ugrađeni softver često radi u uslovima *ograničenih resursa* kao što su memorija, procesorska snaga i energetska potrošnja, zbog čega je primena različitih *tehnika optimizacije* obavezna. Ovo uključuje pažljivo upravljanje memorijom, procesorskom snagom i energijom uređaja.

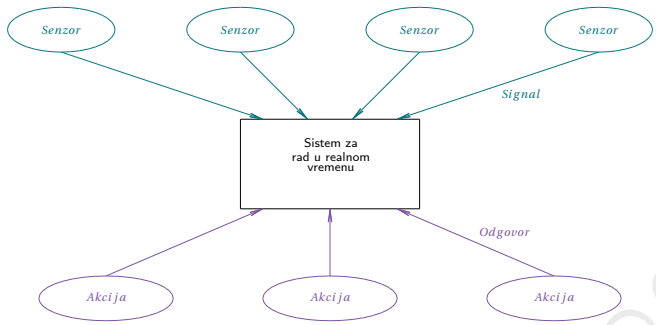
Ugrađeni sistemi imaju sledeće karakteristike:

- Direktna interakcija sa hardverom je često neophodna.
- Reakcije na okruženje mogu biti periodične (npr. redovno očitavanje senzora) ili aperodične (kao reakcija na iznenadne događaje poput aktiviranja alarma).
- Sigurnost i pouzdanost su ključne karakteristike, posebno u aplikacijama kao što su medicinski uređaji, sistemi za kontrolu saobraćaja ili automobilski sistemi.
- Postoje fizička ograničenja koja mogu uticati na dizajn sistema, kao što su ograničen prostor, energetska efikasnost, temperatura rada uređaja, vibracije, itd.

Proces razvoja embedded softvera je interdisciplinaran, zahteva saradnju između inženjera hardvera i softvera radi osiguravanja optimalne integracije i sinhronizacije. Razvoj može započeti izborom odgovarajućeg hardvera i operativnog sistema ili definisanjem softverskih zahteva prema kojima se kasnije bira odgovarajuća platforma.

Ugrađeni softver obično se realizuje kao skup konkurentnih procesa, tj. procesa koji se paralelno izvršavju i koji međusobno komuniciraju. Zbog zahteva za konkurentnim izvršavanjem, najčešće se koristi operativni sistem za rad u realnom vremenu (eng. Real-Time Operating System – RTOS), čiji raspoređivač (eng. scheduler) upravlja izvršavanjem procesa i resursima. RTOS se razlikuje od opštih operativnih sistema kao što su Windows ili Linux u standardnoj verziji, jer ovi sistemi ne mogu uvek garantovati reakciju u realnom vremenu. Međutim, postoje specijalizovane verzije Linuxa (npr. RTLinux ili PREEMPT-RT) koje su prilagođene za rad u realnom vremenu.

Za modeliranje ponašanja sistema često se koristi dijagrami stanja, koji jasno pokazuju kako sistem menja stanja kao reakciju na spoljašnje događaje, tj. ulazne signale. Signali mogu biti periodični (na primer, redovno očitavanje senzora) ili aperodični (na primer, reakcija na neočekivane događaje).



Slika 1.7: Opšti model ugrađenog sistema sa reakcijom u realnom vremenu

Pri razvoju embedded softvera, često se koriste sistemski programski jezici poput jezika C zbog efikasnosti generisanog koda. Međutim, jezici poput programskog jezika C nemaju ugrađenu podršku za konkurentnost ili upravljanje deljenim resursima. Zbog toga programeri moraju pažljivo da koriste mehanizme RTOS-a kao što su semafori ili međusobno isključivanje. Ovo može povećati rizik od grešaka, jer je potrebno dodatno znanje o specifičnim sistemskim pozivima i radu sa hardverom.

Ugrađeni sistemi često zahtevaju UML dijagrame, posebno dijagrame stanja, kako bi se jasno definisalo ponašanje sistema i njegovo reagovanje na različite signale. Time se olakšava dizajn, implementacija i testiranje sistema.

Kao zaključak – ugrađeni softver predstavlja kritičnu komponentu savremenih uređaja, omogućavajući im pouzdanost, efikasnost i pravovremeno reagovanje na zahteve okruženja.

1.5.4 Veštačka inteligencija u razvoju softvera

U današnjem svetu, veštačka inteligencija (VI) igra sve značajniju ulogu u razvoju softvera, omogućavajući brže i efikasnije procese. Dodatno, u savremenom računarstvu se mnogi problemi koji nisu se mogli rešiti tradicionalnim algoritmima sada mogu rešiti korišćenjem veštačke inteligencije.

Ipak, veštačka inteligencija nije zamena za ljudsku kreativnost i inovativnost i ne predstvlja "magični štapić" za sve probleme. Pre nego što se započne projekat zasnovan na VI, potrebno je proceniti da li je primena ovih tehnika zaista neophodna i ekonomski opravdana. Današnji VI sistemi često zahtevaju veliku količinu podataka i veliku količinu računarskih resursa (električne struje). U donjoj tabeli

prikazane su opšte smernice koje pomažu pri odlučivanju kada je VI korisna, a kada nije.

- **Postoji obrazac koji treba naučiti.** VI služi da prepozna obrasce u podacima, odnosno kompleksne relacije i veze koje su teško uočljive ljudima ili se ne mogu lako eksplicitno isprogramirati. Kompleksan obrazac znači da među podacima postoje mnogobrojne, nelinearne i često šumovite veze koje je nemoguće opisati jednostavnim pravilima (npr. raspored piksela koji otkriva lice ili kombinacija tržišnih signala koja utiče na cenu akcije). Za jednostavne – poput računa površine $A = a \cdot b$ ili bacanja poštenog novčića gde obrasca nema – dovoljan je klasičan kod, a mašinsko učenje ne donosi prednost.

- **Podaci su dostupni.** Da bi se obučio model, potreban je dovoljan broj podataka koji su relevantni za problem koji se rešava. Ako podaci nisu dostupni, onda postoji mehanizam da se podaci prikupe ili generišu. Na primer, za prepoznavanje lica potrebno je mnogo slika lica sa različitim izrazima, uglovima i osvetljenjem. Ako podaci nisu dostupni ili su nedovoljni, VI neće biti efikasna. Takođe, podaci moraju biti kvalitetni i reprezentativni za problem koji se rešava. Na primer, ako se model obučava na slikama lica, ali su slike lošeg kvaliteta ili nisu raznovrsne, model neće biti sposoban da prepozna lica u različitim uslovima.

- **Problem je prediktivan.** VI je korisna kada je cilj predvideti buduće događaje na osnovu istorijskih podataka. Na primer, predikcija vožnje automobila, vremenskih uslova ili popularnosti proizvoda na osnovu objava na društvenim mrežama.

- **Cena greške je mala.** VI sistemi nisu egzaktni i mogu napraviti greške. Ako je cena greške visoka (npr. u medicini ili autonomnim vozilima), onda je bolje koristiti klasične algoritme koji su pouzdaniji. Na primer, ako VI sistem pogrešno prepozna tumor na slici, to može dovesti do pogrešne dijagnoze i lečenja pacijenta. U takvim situacijama, u današnjem svetu, često se koristi kombinacija VI i ljudske ekspertize, gde VI pomaže lekaru da brže i efikasnije identifikuje potencijalne probleme, ali konačnu odluku donosi lekar. Slično je i u autonomnim vozilima, gde VI pomaže u prepoznavanju prepreka i donošenju odluka, ali je vozač uvek odgovoran za bezbednost vožnje. VI se danas koristi i u dokazivanju matematičkih teorema, gde se koristi za prepoznavanje obrazaca i generisanje novih dokaza, ali postoje nezavisni sistemi koji proveravaju ispravnost tih dokaza. Sa druge strane, VI je potpuno bezbedno koristiti u marketingu za sisteme preporuka i slično.

- **Obrasci se vremenom menjaju.** Ako se obrasci u podacima menjaju tokom vremena, VI može da se prilagodi tim promenama. Na primer, ako se trendovi na društvenim mrežama menjaju, VI može da se obučava na novim podacima kako bi ostala relevantna. Sa druge strane, ako su izlazi i uvek isti (npr. računanje površine kvadrata), onda VI nije potrebna.

Proces razvoja softvera zasnovan na veštačkoj inteligenciji je itertivan i često uključuje sldeće korake:

1. **Definisanje problema i metrike.** Prvi korak je jasno definisanje cilja i metrika koje će se koristiti za merenje uspeha.
2. **Prikupljanje i priprema podataka.** Prikupljanje relevantnih podataka i njihova priprema za obučavanje modela. Ovo može uključivati čišćenje podataka, anotaciju, normalizaciju, transformaciju i slično. U današnjem vremenu koristete se i tehnike generativne veštačke inteligencije za generisanje podataka koji nedostaju ili za obogaćivanje postojećih podataka. Takođe, korak pripreme podataka često može biti ključan za uspeh projekta, jer kvalitet podataka direktno utiče na performanse modela.
3. **Obučavanje modela.** Obučavanje modela na pripremljenim podacima. Ovo može uključivati izbor algoritma, podešavanje hiperparametara i evaluaciju modela na testnim podacima. Model se obično obučava na trening skupu, a postoji odvojen skup podataka na kojima se model testira.
4. **Iterativna analiza i poravka modela.** Analiza rezultata obučavanja i testiranja modela, identifikacija problema i iterativno popravka modela. Ovo može uključivati promenu arhitekture modela, dodavanje novih funkcija, promenu hiperparametara ili promene u samom skupu podataka.
5. **Kontinuirano praćenje i ažuriranje modela.** Nakon što je model implementiran, važno je kontinuirano pratiti njegovu performansu i ažurirati ga prema potrebi. Ovo može uključivati prikupljanje novih podataka, obučavanje modela na novim podacima ili prilagođavanje modela promenama u obrascima podataka.

Pored razvoja VI softvera, veštačka inteligencija se takođe koristi u različitim fazama razvoja softvera i kao pomoćni alat za programere. Sve češće se koriste "pametni" asistenti zasnovani na velikim jezičkim modelima. Najpoznatiji su GitHub Copilot i ChatGPT, ali postoje i alternative kao što su Amazon CodeWhisperer, Tabnine, Google Duet AI i JetBrains AI Assistant. Ovi alati se integrišu u razvojna okruženja (VS Code, IntelliJ, JetBrains Rider itd.) i pomažu pri automatskom dovršavanju koda, generisanju testova, refaktorisanjima i pisanju dokumentacije. Oblast se brzo razvija i gotovo svakog meseca pojavljuju se nova rešenja ili novi modeli (na primer, CodeLama, Cline, StarCoder) Očekuje se da će u budućnosti biti još više inovacija i poboljšanja u ovoj oblasti.

Sa druge strane, važno je napomenuti da ovi alati nisu savršeni i da ponekad mogu generisati netačan ili neefikasan kod. Komercijalne verzije ovih alata najčešće su pretplaćene i nisu svima finansijski dostupne, a mnoge organizacije ih ograničavaju ili potpuno zabranjuju zbog bezbednosnih i pravnih briga (slanje vlasničkog koda ka eksternim servisima za obradu, nejasna pitanja autorskih prava nad generisanim kôdom i drugo). Zbog toga se u praksi često koristi "hibridni" pristup: VI asistenti za zadatke koji nisu poverljivi, a lokalni modeli ili klasični alati za rad na osetljivom kodu.

Pitanja i zadaci za vežbu

Pitanje 1.1. *Šta su sličnosti a koje razlike između projekata u građevinarstvu i informacionim tehnologijama?*

Pitanje 1.2. *Navesti faze razvoja softvera i ko ih obično sprovodi.*

Pitanje 1.3. *Koje dve vrste dokumentacije treba da postoje?*

Pitanje 1.4. *Nabrojati najznačajnije metodologije razvoja softvera. Istraži na internetu koje su metodoloje razvoja softvera danas najpopularnije.*

Pitanje 1.5. *Koje su glavne razlike u performansama distribuiranih i centralizovanih sistema, i koji faktori najviše utiču na brzinu obrade podataka u distribuiranim sistemima?*

Pitanje 1.6. *Objasnite koncept skalabilnosti u distribuiranim sistemima i navedite razlike između povećanje kapaciteta postojećih čvorova (eng. scaling up) i dodavanja novih čvorova (eng. scaling out), uz konkretan primer kako se svaki od ovih pristupa može primeniti u praksi.*

Pitanje 1.7. *U kontekstu različitih arhitekturnih modela (gospodar–rob, dvoslojna, višeslojna i decentalizovana arhitektura), koje su prednosti i mane svakog modela, i u kojim scenarijima je jedan model pogodniji od drugog?*

Pitanje 1.8. *Koji su osnovni razlozi zašto se za ugrađeni softver često koriste operativni sistemi za rad u realnom vremenu (RTOS) umesto klasičnih operativnih sistema?*

Pitanje 1.9. *Koji osobine i zahtevi utiču na razvoj ugrađenog softvera?*

Pitanje 1.10. *Objasnite tri ključne razlike između komponentno softverskog inženjerstva (CBSE) i mikroservisne arhitekture.*

Pitanje 1.11. *Navedite i obrazložite najmanje dve prednosti i jedan izazov upotrebe* Dockera *prilikom isporuke mikroservisa, uz konkretan primer.*

Pitanje 1.12. *Definišite pojam ograničenog konteksta u domenu mikroservisa.*

Pitanje 1.13. *Kompanija želi da predvidi da li će se određeni proizvod prodavati bolje sledećeg meseca na osnovu samo tri fiksne numeričke metrike koje se ne menjaju tokom godine. Objasnite sa dva argumenta zašto bi tradicionalni algoritam mogao biti prikladniji od primene modela mašinskog učenja.*

Pitanje 1.14. *Nabrojte (tačnim redosledom) pet glavnih faza razvoja ML softverskog rešenja opisanih u tekstu i ukratko navedite šta je cilj svake faze.*

Pitanje 1.15. *(a) Navedite dve konkretne pogodnosti koje* GitHub Copilot *ili slični asistenti donose programerima.*

(b) Imenujte dva glavna razloga zbog kojih neke organizacije ipak zabranjuju upotrebu ovih alata u svojim projektima.