UNIVERZITET METROPOLITAN BEOGRAD

SE101 - RAZVOJ SOFTVERA I INŽENJERA SOFTVERA

Softversko inženjerstvo – teorija i praksa

Lekcija 09

PRIRUČNIK ZA STUDENTE

SE101 - RAZVOJ SOFTVERA I INŽENJERA SOFTVERA

Lekcija 09

SOFTVERSKO INŽENJERSTVO – TEORIJA I PRAKSA

UVOD

Cilj i sadržaj lekcije

Cilj lekcije predstavlja razumevanje osnovnih principa inženjerstva zahteva. U toku lekcije naučićete:

- šta podrazumeva inženjerstvo zahteva i na koji način se zahtevi prikupljaju
- koje su tehnike modelovanja procesa
- stilove arhitekture softvera i načine procene arhitekture
- na koji način se bira model koji će biti korišćen za dizajn softvera
- kroz koje faze testiranja i procene prolazi svaki softver
- osnovne principe održavanja softvera
- o ulogama i upotrebi softverske dokumentacije

Poglavlje 1

Inženjerstvo zahteva

ŠTA JE INŽENJERSTVO ZAHTEVA?

Inženjerstvo zahteva nam pomaže da što bolje prikupimo, analiziramo i dokumentujemo zahteve za softvere

Inženjerstvo zahteva je prvi veliki korak ka rešenju problema obrade podataka. Tokom ove faze poštuju se zahtevi korisnika koji se odnose kako na funkcije koje treba obezbediti, tako i na niz dodatnih karakteristika (performanse, pouzdanost, dokumentacija itd.).

Zahtev je „uslov ili sposobnost koja je potrebna korisniku da reši problem ili da postigne cilj“. Pojam „korisnik“ može označavati krajnjeg korisnika sistema ali i nekoliko klasa indirektnih korisnika, kao što su ljudi koji koriste informacije koje sistem isporučuje.

Tokom inženjerstva zahteva moguće je izdvojiti 4 različite faze:

- Prikupljanje zahteva, odnosno razumevanje problema.
- Kreiranje specifikacije zahteva - ovaj dokument opisuje proizvod koji se isporučuje.
- Validacija i verifikacija zahteva - utvrđivanje da li su navedeni tačni zahtevi (validacija) i da li su ti zahtevi ispravno realizovani (verifikacija).
- Pregovaranje o zahtevima - zbog vremenskih ograničenja ili drugih faktora, ponekad se mora pregovarati o zahtevima pri čemu se vrši prioritizacija i izbor sa liste svih navedenih zahteva.

Navedeni procesi uključuju ponavljanje i povratne informacije dobijene od strane korisnika. U teoriji, ove iteracije prethode dizajnu i implementaciji. U agilnim procesima, dizajn i implementacija su deo iteracije i povratne sprege. U svakom slučaju, tokom čitavog procesa postoji centralno skladište u kome su svi zahtevi dokumentovani.

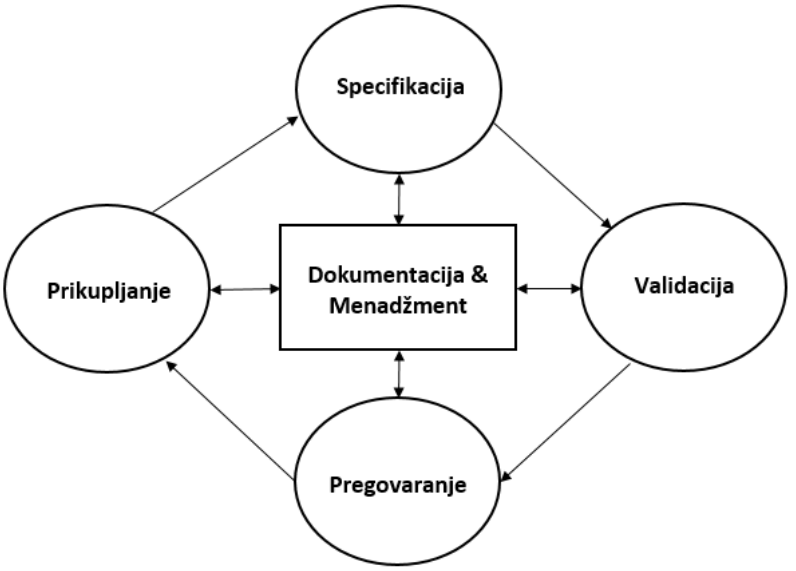
PRIKUPLJANJE ZAHTEVA

Osnovni izazov prilikom prikupljanja zahteva predstavlja razumevanje činjenica relevantnih za određeni zahtev. U cilju prevazilaženja, neophodno je koristiti neke od navedenih tehnika

Zahtev koji se kreira u procesu inženjerstva zahteva mora biti takav da ga je moguće komunicirati prema svim zainteresovanim stranama, kao što su analitičari i korisnici. Kako je neophodno da svaki zahtev sadrži samo relevantne informacije, jedan od osnovnih izazova jeste da se uzmu u obzir svi važni uticaji ali i izostave nebitni detalji. Da bi faza inženjerstva

zahteva bila uspešna, neophodno je koristiti metode i tehnike koje pokušavaju da zaobiđu navedene poteškoće. U listi ispod navedene su neke od tehnika prikupljanja zahteva:

- Pitanja
- Analiza zadataka
- Analiza zasnovana na scenariju
- Etnografija
- Opisi prirodnim jezikom
- Preuzimanje postojećih sistema
- Analiza domena
- Izrada prototipa



Slika 1.1 Prikupljanje zahteva [1,2]

SPECIFIKACIJA ZAHTEVA

Specifikacija zahteva odražava međusobno razumevanje problema koji treba da reše potencijalni korisnici i razvojna organizacija

Dokument koji se proizvodi tokom inženjerstva zahteva, *specifikacija zahteva*, služi dvema grupama ljudi. Za korisnika, specifikacija zahteva je jasan i precizan opis funkcionalnosti koju sistem može da ponudi. Za dizajnera, to je polazna tačka za dizajn. Zbog različitih očekivanja zainteresovanih strana definisanje specifikacije je veoma je kompleksna aktivnost.

Sledeće tehnike se najčešće koriste za specifikaciju zahteva: prirodni jezik, slike i formalni jezik. Prednost korišćenja prirodnog jezika je u tome da je specifikacija veoma čitljiva i razumljiva za korisnika. Prednost slika ogleda se u predstavljanju funkcionalne arhitekture sistema. Na kraju, formalni jezik omogućava korišćenje alata u analizi zahteva.

Jedna od mogućnosti za kreiranje specifikacije jeste kombinovanje formalnog i prirodnog jezika: opisivanje i analiza problema uz korišćenje neke formalne notacije razumljive dizajneru, a zatim njeno prevođenje na prirodni jezik. Ovako dobijen opis čitljiv je korisniku, ali i sa dovoljno preciznih informacija dostupnih dizajneru.

Specifikacija nefunkcionalnih zahteva

Postoje četiri tipa nefunkcionalnih zahteva:

- zahtevi za spoljni interfejs
- zahtevi za performanse
- ograničenja dizajna
- atributi softverskog sistema.

Nefunkcionalni zahtevi su ograničenja koja se postavljaju na proces razvoja ili proizvode koji će biti isporučeni. Primeri takvih ograničenja mogu biti: formati izveštaja, hardverska ograničenja i sl.

VALIDACIJA I VERIFIKACIJA ZAHTEVA

Osnovni izazov prilikom validacije zahteva jeste obezbeđivanje razumevanja od strane korisnika

Validacija zahteva podrazumeva njihovu proveru u smislu ispravnosti, potpunosti, dvosmislenosti i unutrašnje i spoljašnje doslednosti. Ovaj proces uključuje učešće korisnika koji su vlasnici problema i oni jedini odlučuju da li specifikacija zahteva adekvatno opisuje njihov problem.

Prilikom validacije zahteva neophodno je uveriti se da uključene strane imaju isto, pravilno razumevanje problema. Osnovni izazov u ovoj fazi je obezbeđivanje razumevanja sadržaja specifikacije zahteva od strane korisnika. Tehnike koje se primenjuju u ovoj fazi često podrazumevaju prevod zahteva u oblik koji je ugodan korisniku: parafraziranje na prirodnom jeziku, diskusija o mogućim scenarijima upotrebe, izradu prototipa i animacije.

Pored testiranja same specifikacije zahteva, u ovoj fazi definiše se i plan testova koji će se koristiti tokom testiranja sistema ili faze prihvatanja.

Plan testiranja je dokument koji propisuje obim, pristup, resurse i raspored aktivnosti testiranja. Ovaj dokument identifikuje stavke i karakteristike koje treba testirati, testiranja koja treba izvršiti, kao i osobe odgovorne za ove aktivnosti. U ovoj fazi može se razviti kompletan plan za fazu testiranja sistema, odnosno fazu u kojoj razvojna organizacija testira sistem u odnosu na njegove zahteve.

Testiranje u cilju prihvatanja *(verifikacija zahteva*) je slično postupku validacije, ali se vrši pod nadzorom organizacije korisnika. Postupak verifikacije podrazumeva prijemno testiranje kako bi se utvrdilo da li korisnici prihvataju sistem ili ne.

Poglavlje 2

Modelovanje procesa

TEHNIKE MODELOVANJA

U zavisnosti od primenjenih notacija, razlikuju se klasične i objektno orjentisane tehnike modelovanja

Tokom inženjeringa zahteva i dizajna primenjuju se različite notacije modelovanja. Većina njih koristi neku vrstu dijagrama kutija i linija. Današnji glavni tokovi modelovanja potiču iz Unificiranog Jezika Modelovanja - UML (Unified Modeling Language). Međutim, mnogi UML dijagrami su zasnovani ili izvedeni iz ranijih tipova dijagrama. U ovom poglavlju prikazane su najčešće korišćene klasične tehnike modelovanja, kao i osnovni tipovi UML dijagrama.

Klasične tehnike modelovanja

Četiri klasične tehnike modelovanja postoje već duže vreme:

- *Entity–relationship model (ERM)*
- *Konačni automat (FSM)*
- *Dijagram toka podataka (DFD)*
- *CRC kartice*

Svaka od navedenih tehnika detaljnije je opisana na sledećim slajdovima.

Unificirani Jezik Modelovanja - UML

UML je evoluirao iz ranijih objektno orijentisanih analiza i metoda projektovanja. Koncepti koji se koriste u UML-u, kao što su objekat, atribut, klasa, odnos, potiču iz oblasti objektne orijentacije. UML 2 nudi 13 tipova dijagrama koji spadaju u dve klase. Neki dijagrami daju statički prikaz sistema, dok drugi daju dinamički prikaz.

MODEL ENTITET–VEZA / ENTITY-RELATIONSHIP MODEL (ERM)

Model fokusiran na definisanje logičke strukture uz koji je neophodno koristiti dodatne tehnike za definisanje drugih aspekata

Entity–relationship model (ERM)

Ovaj model fokusiran je na modelovanje logičke strukture, a ne na izvršavanje zadatka u određenom sistemu. Postoji veliki broj varijanti ERM-a koji se razlikuju po svom grafičkom prikazu notacije. U nastavku su dati osnovni elementi ovog modela.

- *entitet* - prepoznatljivi objekat neke vrste
- *tip entiteta* - tip grupe objekata
- *vrednost atributa* - deo informacije koja (delimično) opisuje entitet
- *atribut* - skup vrednosti atributa
- *veza* - odnosi se na vezu između dva ili više entiteta

Entiteti se mogu jednostavno i lako identifikovati. Obično su prikazani kao pravougaonici.

Svojstva entiteta poznata su kao atributi. Atributi se obično prikazuju kao krugovi ili elipse.

I entiteti i vrednosti atributa imaju tipove. Dok modelari teže da posmatraju tip kao skup svojstava koja dele određene instance, implementatori teže da posmatraju tip kao skup vrednosti sa većim brojem pridruženih operacija.
Model odnosa entiteta može se dobiti korišćenjem bilo koje tehnike prikupljanja zahteva o kojima je bilo reči. Pošto ERM definiše samo deo priče,
pored ove moraju se koristiti dodatne tehnike za modelovanje drugih aspekata.

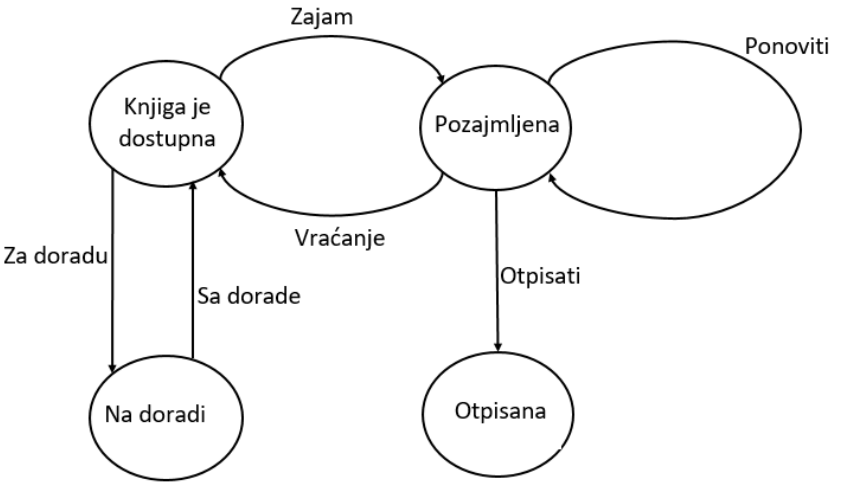
Detaljnije informacije o ERM dostupne su na linku: https://www.youtube.com/watch?v=H-MDosnw8sI&ab_channel=IvanKnezevic

KONAČNI AUTOMAT / FINITE STATE MACHINE (FSM)

FSM je slikovito predstavljen kao dijagram prelaza stanja

Tehnike specifikacije zahteva koje modeluju sistem u smislu stanja i prelaza između stanja nazivaju se *tehnike modelovanja zasnovane na stanju*. Jednostavan način za specificiranje stanja i prelaza stanja jeste *Konačni automat*, odnosno*FSM*.
FSM se sastoji od konačnog broja stanja i skupa prelaza iz jednog stanja u drugo koji se javljaju na ulaznim signalima iz konačnog skupa mogućih stimulusa. Početno stanje je posebno označeno stanje iz kojeg mašina kreće. Obično se jedno ili više stanja označavaju kao krajnja. Slikovno, FSM
su predstavljeni kao dijagrami prelaza stanja. U njima, stanja su predstavljena kao krugovi sa oznakom koja identifikuje stanje, dok su prelazi označeni kao lukovi iz jednog stanja u drugo, gde oznaka označava stimulans koji pokreće tranziciju.

Ovaj tip modelovanja ne prikazuje kompletno stanje sistema u jednom krugu niti oslikava sve moguće prelaze stanja. Modelovanje sistema u jednom ovakvom dijagramu nije preporučljivo jer ubrzo postaje nezgrapan i teško razumljiv. Mogući način za prevazilaženje je da se omogući hijerarhijska dekompozicija FSM-ova. Ovo je suština notacije poznate kao grafikoni stanja.



Slika 2.1 Primer Konačnog automata za pozajmiljivanje knjige iz biblioteke [1,2]

DIJAGRAM TOKA PODATAKA / DATA FLOW DIAGRAM (DFD)

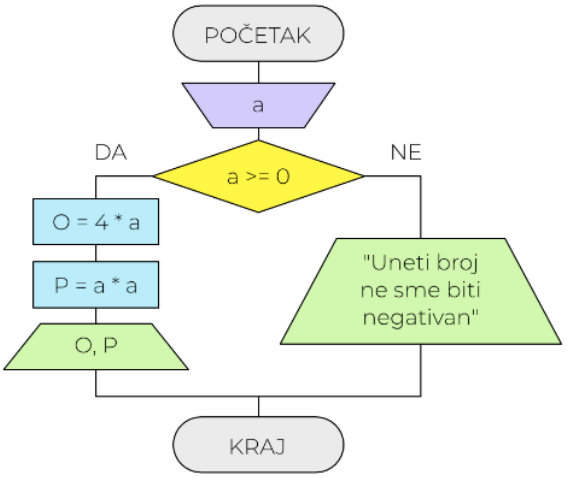
Dijagrami toka podataka su rezultat procesa funkcionalne dekompozicije sistema

Dizajn toka podataka je funkcionalna dekompozicija u odnosu na tok podataka. Komponenta (modul) je crna kutija koja transformiše neki ulazni tok u neki izlazni tok. Glavna notacija koja se koristi je *dijagram toka podataka (DFD).*

U dijagramu toka podataka razlikuju se četiri tipa entiteta podataka:

1. *Eksterni entiteti* - nalaze se van domena koji se razmatra u dijagramu toka podataka i označeni su kao kvadrati
2. *Procesi* - transformišu podatke i označeni su kružićima
3. *Tokovi podataka* - označeni su strelicama i nalaze se između procesa, eksternih entiteta i skladišta podataka. To su putevi duž kojih se kreću strukture podataka.
4. *Skladišta podataka* - definišu ih imena između dve paralelne linije i predstavljaju mesta gde se čuvaju strukture podataka do potrebe.

Dijagrami toka podataka su rezultat procesa dekompozicije odozgo na dole. Na najvišem nivou se nalazi sistem koji se dalje dekomponuje.



Slika 2.2 Dijagram toka podataka [1:autor]

CRC KARTICE

CRC kartice sastoje se od 3 polja (Klasa, Obaveza, Saradnici) i koriste se za dokumentovanje dizajnerske odluke

CRC predstavlja skraćenicu za vezu Klasa - Obaveza - Saradnici (Class - Responsibility - Collaborators). To je kartica od tri polja koja su označena kao Klasa, Obaveza i Saradnici. Ime klase se pojavljuje u gornjem levom uglu kartice, lista obaveza ispod naziva klase, a lista saradnika se pojavljuje u desnom delu kartice.

CRC kartice su razvijene kao odgovor na potrebu za dokumentovanjem zajedničke dizajnerske odluke. Posebno su korisne u ranim fazama razvoja softvera kako bi se pomoglo u identifikaciji komponenti, razgovaralo o pitanjima dizajna u multidisciplinarnim timovima i neformalno specificirale komponente.

Klasa	**Saradnici**
Rezervacija	Katalog Korisnici
Obaveza Napraviti listu rezervisanih naslova Upravljati rezervacijama	

Slika 2.3 Grafički prikaz CRC kartice [1,2]

UNIFICIRANI JEZIK MODELOVANJA - UML

Najrasprostranjenija notacija koja je zasnovana na objektno orjentisanom modelovanju

Sa aspekta modelovanja, objekat je konceptualni model nekog dela realnosti ili imaginarnog sveta. Važne karakteristike objekata su identitet (oznaka koja ga jedinstveno identifikuje) i svojstva (promenljiva stanja i operacije za modifikaciju ili inspekciju stanja). Ovako posmatrano, objekat je kolekcija tri aspekta: objekat = identitet + stanje + ponašanje

Termin atribut označava bilo koje polje u osnovnoj strukturi podataka. Tako, identitet objekta predstavlja atribut, stanje označava skup 'strukturnih' atributa, a operacije označavaju atribute „ponašanja“. Za objekte koji imaju isti skup atributa kaže se da pripadaju istoj klasi. Pojedinačni objekti klase su instance te klase.

Unificirani Jezik Modelovanja - UML

UML je najrasprostranjenije korišćena notacija za inženjering zahteva i dizajn. Trenutno se koristi verzija UML2 koja se sastoji od 13 osnovnih dijagrama:

- Aktivnost - za modelovanje poslovnih procesa i proceduralnih logika.
- Klasa - za klase, njihove karakteristike i odnose.
- *Komunikacija* - za tokove poruka između instanci klasa.
- *Komponenta* - za modelovanje skupa komponenti i njihovih međusobnih odnosa.
- *Kompozitna struktura* - za modelovanje unutrašnje dinamičke strukture klase.
- *Primena* - za modelovanje fizičkog rasporeda od sistemskih do hardverskih elemenata.
- *Pregled interakcije* - kombinuje dijagrame aktivnosti i dijagrame sekvence.
- *Objekat* ili *dijagram instance* - za modelovanje objekata i njihovih odnosa u nekom trenutku.
- *Paket* - za grupisanje elemenata u pakete.
- *Sekvenca* - za redosled poruka razmenjenih između instanci klasa.
- *Stanje mašine* - za stanja u kojima objekat može biti i prelaz između njih.
- *Vreme* - za promene stanja objekta tokom vremena.
- *Slučaj upotrebe (Use case)* - za modelovanje slučajeva upotrebe.

Poglavlje 3

Projektovanje sistema

ARHITEKTURA SOFTVERA

Dekompozicija sistema na najvišem nivou na glavne komponente zajedno sa karakterizacijom interakcije ovih komponenti

Tokom faze projektovanja, sistem se razlaže na više komponenti koje međusobno deluju. Dekompozicija sistema na najvišem nivou na glavne komponente zajedno sa karakterizacijom interakcije ovih komponenti naziva se njegova ***softverska arhitektura*** .

Arhitektura softvera ima tri glavne svrhe:

- To je sredstvo za komunikaciju među zainteresovanim stranama. Arhitektura softvera je globalni, često grafički, opis koji se može podeliti sa kupcima, krajnjim korisnicima, dizajnerima itd.
- Obuhvata rane dizajnerske odluke. U softverskoj arhitekturi, globalna struktura sistema je odlučena kroz eksplicitno dodeljivanje funkcionalnosti komponentama arhitekture. Ove rane dizajnerske odluke su važne jer se njihova poboljšanja osećaju u svim narednim fazama, te je veoma korisno da se njihov kvalitet proceni što je pre moguće.
- To jeprenosiva apstrakcija sistema i osnova za ponovnu upotrebu. Sve suštinske odluke su obuhvaćene u arhitekturi, stoga ona pruža osnovu za porodicu sličnih sistema, takozvanu liniju proizvoda.

Veoma aktivno polje istraživanja usmereno je na identifikaciju i opisivanje komponenata na višem nivou apstrakcije, odnosno iznad nivoa modula ili apstraktnog tipa podataka. Ove apstrakcije višeg nivoa poznate su kao softverski arhitektonski stilovi.

STILOVI ARHITEKTURE SOFTVERA

Različiti stilovi arhitekture mogu biti primenjeni u zavisnosti od tipa problema koji treba rešiti

Arhitektonski stil softvera opisuje određenu kodifikaciju elemenata i njihovog rasporeda, dok, sa druge strane, ograničava i elemente i njihove međusobne odnose. Odabir određenog arhitektonskog stila zavisi od problema koji treba rešiti kao i šireg konteksta u kome se problem javlja.

U nastavku je opisano 6 dobro poznatih stilova arhitekture softvera:

- Glavni program sa potprogramima - glavni zadaci sistema su dodeljeni različitim komponentama koje se pozivaju, u odgovarajućem redosledu, iz kontrolne komponente koja kontroliše taj redosled.
- Apstraktni tipovi podataka - suština ovog stila je da korisnik ne dolazi direktno u kontakt sa podacima, već se pristupanje vrši preko interfejsa, odnosno putem poziva odgovarajućih procedura ili metoda.
- Implicitno pozivanje - u ovom stilu generiše se tzv. događaj. Ostale komponente u sistemu mogu izraziti svoj interes za ovaj događaj povezujući metod sa njim. Ovaj metod se automatski poziva svaki put kada se događaj pokrene.
- Cevi i filteri - kada određena komponenta proizvodi izlaz koji se zatim čita i obrađuje komponentom i+1, istim redosledom kojim ga piše komponenta i može se koristiti režim cevi i filtera, operacije koje su dobro poznate iz UNIKS-a i direktno unose izlaz jedne transformacije u sledeću.
- Repozitorijum **-** koristi se u situacijama kada je glavno pitanje upravljanje bogato strukturiranim korpusom informacija. Pristup ovom problemu je osmišljavanje šema baza podataka za različite tipove podataka u aplikaciji i skladištenje podataka u jednoj ili više baza.
- Slojeviti stil - sastoji se od 7 slojeva: fizički, podaci, mreža, transport, sesija, prezentacija i prijava. Donji sloj pruža osnovnu funkcionalnost, dok
  viši slojevi koriste funkcionalnosti nižih slojeva. U slojevitoj šemi, po definiciji, niži nivoi ne mogu da koriste funkcionalnosti
  viših nivoa.

PROCENA ARHITEKTURE SOFTVERA

Testiranje softverske arhitekture od rane faze je izuzetno važno za uspešnost kompletnog projekta

Arhitektura softvera obuhvata rane dizajnerske odluke. Pošto ove rane odluke imaju veliki uticaj važno je započeti testiranje čak i u ranoj fazi. Testiranje softverske arhitekture se obično naziva procena arhitekture softvera.

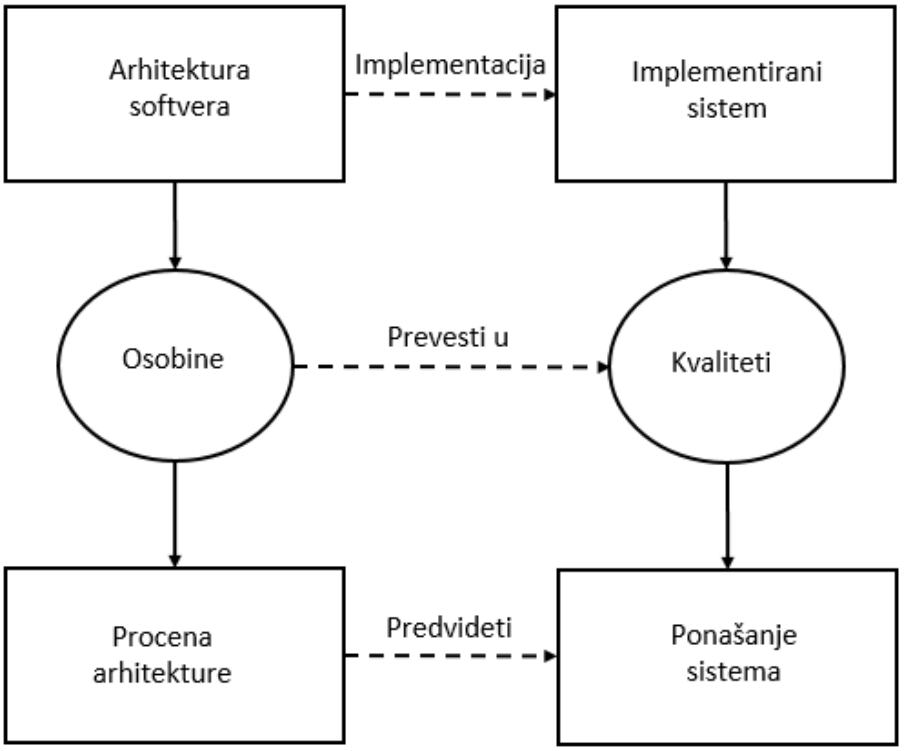
Postoje dve široke klase tehnika za procenu softverske arhitekture.

- Prva klasa obuhvata merne tehnike i oslanja se na kvantitativne informacije. Primeri uključuju metriku arhitekture i simulaciju.
- Druga klasa obuhvata tehnike ispitivanja u kojima se istražuje kako arhitektura reaguje na određene situacije. Ovo se često radi uz pomoć scenarija.

Postoje različite vrste scenarija koje se mogu koristiti u procenama arhitekture:

- Slučajevi upotrebe
- Slučajevi promena
- Stresne situacije
- Scenariji daleke budućnosti

Jedna od najpoznatijih metoda procene arhitekture je ATAM: metoda analize kompromisa arhitekture.



Slika 3.1 Procena arhitekture [1,2]

Poglavlje 4

Razvoj softvera

DIZAJN SOFTVERA

Dizajn softvera podrazumeva dekompoziciju sistema na jednostavnije komponente

Sa tehničke tačke gledišta, problem dizajna se može formulisati na sledeći način:
kako dekomponovati sistem na delove tako da svaki deo ima manju složenost od sistema u celini, dok delovi zajedno rešavaju problem korisnika.
Pošto složenost pojedinačnih komponenti treba da bude razumna, važno je da interakcija između komponenti ne bude previše komplikovana.
Dizajn ima i aspekt proizvoda i aspekt procesa. Aspekt proizvoda se odnosi na rezultat, dok se aspekt procesa odnosi na to kako se do toga dolazi.

Postoji mnogo načina da se sistem razloži na module. Međutim, nije svaka dekompozicija podjednako poželjna. Poželjne karakteristike dekompozicije se mogu koristiti kao mera kvaliteta dizajna. Neke od ovih karakteristika olakšavaju održavanje i ponovnu upotrebu: jednostavnost, jasno razdvajanje koncepata u različite module i ograničena vidljivost informacija. Sistemi koji imaju ta svojstva lakše se održavaju jer je moguće koncentrisati se na one delove koji su direktno pogođeni promenom. Ova svojstva takođe utiču na ponovnu upotrebu, jer rezultirajući moduli obično imaju dobro definisanu funkcionalnost koja odgovara konceptima iz domen aplikacije. Takvi moduli mogu biti kandidati za uključivanje u druge sisteme koji se bave problemima iz istog domena.

U nastavku je navedeno pet međusobno povezanih pitanja koja imaju jak uticaj na gore navedene karakteristike:

- odvajanje,
- modularnost,
- skrivanje informacija,
- složenost i
- struktura sistema.

Za objektno orijentisane sisteme definisan je specifičan skup heuristike kvaliteta i povezanih metrika.

ODABIR TEHNIKE ZA DIZAJN SOFTVERA

Izbor odgovarajuće tehnike zavisi od karakteristika problema, odnosno proizvoda koji su predmet dizajna

Postoji veliki broj pokušaja za klasifikaciju metoda projektovanja, kao što su proizvodi koje isporučuju, vrsta korišćenih reprezentacija ili njihov nivo formalnosti. Jednostavan, ali koristan okvir predlaže dve dimenzije: orijentacijsku dimenziju i dimenziju modela.

U dimenziji orijentacije, pravi se razlika između problemski orijentisane tehnike i tehnike orijentisane na proizvod.

- Tehnike orijentisane na problem se koncentrišu na stvaranje boljeg razumevanja problema i njegovog rešenja.
- Tehnike orijentisane na proizvod se fokusiraju na ispravnu transformaciju iz specifikacije u implementaciju.

Druga dimenzija se odnosi na proizvode, odnosno modele, koji su rezultat dizajna procesa. U ovoj dimenziji se pravi razlika između konceptualnih modela i formalnih modela. Konceptualni modeli su deskriptivni. Oni opisuju spoljašnju stvarnost i hjihova prikladnost se utvrđuje validacijom.
S druge strane, formalni modeli su preskriptivni. Oni propisuju ponašanje sistema koji treba razviti.

Koristeći ovaj okvir, sve navedene tehnike mogu se svrstati u neki od četiri kvadranta sa sledećim karakteristikama:

- Razumevanje problema
- Transformacija u implementaciju
- Predstavljanje svojstava
- Kreiranje jedinica za implementaciju

Gore navedeni argumenti se odnose na karakteristike problema koji treba rešiti. Pored njih, postoji još nekoliko drugih faktora životne sredine koji mogu uticati na izbor određene tehnike: poznavanje problematičnog domena, iskustvo dizajnera, dostupni alati, sveukupna razvojna filozofija.

Poglavlje 5

Testiranje softvera

TEHNIKE TESTIRANJA SOFTVERA

Najvažniji uslov za pravilan izbor tehnike jeste definisanje cilja testiranja

Testiranje predstavlja izvršavanje programa u cilju provere da li proizvodi tačan izlaz za dati ulaz..

Pre odlučivanja za određeni pristup testiranju, neophodno je odrediti cilj testa. Ako je cilj pronaći što više grešaka biće izabrana strategija koja ima za cilj otkrivanje grešaka. Ukoliko je cilj povećanje poverenja u pravilno funkcionisanje softvera može odlučiti za potpuno drugačiju strategiju. Dakle, cilj će imati svoj uticaj na odabrani pristup testiranju, budući da se rezultati moraju tumačiti u odnosu na postavljene ciljeve.
S obzirom da je neophodno napraviti izbor, veoma je važno izabrati mali, ali adekvatan skup test slučajeva. Tehnike ispitivanja mogu se klasifikovati
prema kriterijumu koji se koristi za merenje adekvatnosti skupa test slučajeva:

- Testiranje zasnovano na pokrivenosti - specificirani su zahtevi za testiranje u pogledu pokrivenosti proizvoda koji se testira (program, dokument sa zahtevima itd.).
- Testiranje zasnovano na greškama - Fault-based testing - fokus je na otkrivanju grešaka. Sposobnost otkrivanja grešaka testnog skupa tada određuje njegovu adekvatnost.
- Testiranje zasnovano na greškama - Error-based testing - tehnike zasnovane na greškama se fokusiraju na tačke sklone greškama, na osnovu
  poznavanje tipičnih grešaka koje ljudi prave.

Alternativno, možemo klasifikovati tehnike testiranja na osnovu izvora informacija koje se koriste za izvođenje test slučajeva:

- Testiranje crne kutije, funkcionalno testiranje ili testiranje zasnovano na specifikacijama - testni slučajevi su izvedeni iz specifikacije softvera, odnosno ne uzimaju se u obzir detalji implementacije.
- Testiranje u beloj kutiji, strukturalno testiranje ili testiranje zasnovano na programu - komplementarni pristup u kome se razmatra unutrašnja logička struktura softvera.

CILJEVI TESTIRANJA

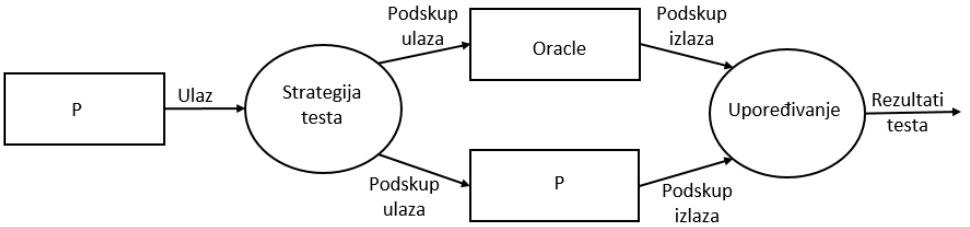
Osnovni ciljevi testiranja su verifikacija i validacija sistema

Sa aspekta ciljeva testiranja može se napraviti razlika između „verifikacije“ i „validacije“ određenog sistema.
Verifikacija predstavlja proces evaluacije sistema ili komponenta da bi se utvrdilo da li proizvodi određene razvojne faze zadovoljavaju uslove definisane na početku te faze. Verifikacija na taj način pokušava da odgovori na pitanje: Da li je sistem *ispravno* izgrađen?
Termin „ *validacija*“ je definisan kao proces evaluacije sistema ili komponente tokom ili na kraju procesa razvoja kako bi se utvrdilo da li zadovoljava definisane zahteve. Validacija se tako svodi na pitanje: Da li je izgrađen *pravi* sistem?

Čak i sa ovom suptilnom razlikom na umu, situacija nije tako jasna. Generalno, program se smatra ispravnim ako dosledno proizvodi pravi izlaz.

S obzirom na činjenicu da iscrpno testiranje nije izvodljivo, proces testiranja može biti realizovan kao što je prikazano na slici 5. Kutija sa oznakom P označava objekat (program, projektni dokument i sl.) koji treba ispitati. Strategija testiranja uključuje izbor podskupa ulaznog domena. Za svaki element ovog podskupa, P se koristi za „izračunavanje“ odgovarajućeg izlaza. Na kraju, očekivani i dobijeni rezultati aktivnosti se upoređuju.

Najvažniji korak u ovom procesu je izbor podskupa ulaznog domena koji će služiti kao test skup. Ovaj skup testova mora biti adekvatan u odnosu na neki izabrani kriterijum testa.



Slika 5.1 Globalni prikaz testiranja procesa [1,2]

PRISTUPI TESTIRANJA I PROCENE SOFTVERA

Osnovni pristupi testiranja su: testiranje odozdo na gore i odozgo na dole

Tokom faze projektovanja, sistem koji treba da se izgradi je razložen na komponente. Generalno, ovi elementi čine neku hijerarhijsku strukturu. Tokom testiranja, često se prati hijerarhijska struktura. Ne počinje se odmah sa testiranjem sistema kao celine, već se počinje testiranjem pojedinačnih komponenti (zvano testiranje jedinica). Zatim se ove komponente postepeno integrišu u sistem. Testiranje sastava komponenti naziva se *testiranje integracije*.
Prilikom testiranja može se koristiti jedan od dva pristupa. U prvom pristupu, najpre se testiraju komponente niskog nivoa koje se zatim integrišu i spajaju sa komponentama na sledećem, višem nivou. Tako dobijeni podsistem se sledeći testira, a zatim se postepeno prelazi na komponente najvišeg nivoa. Ovo je poznato kao *testiranje odozdo prema gore*.
Prilikom ovakvog testiranja često se mora simulirati okruženje u kojem komponenta koja se testira treba da bude integrisana. Ovo okruženje se zove test drajver.

Alternativni pristup je*testiranje odozgo nadole*. U testiranju odozgo nadole, komponente najvišeg nivoa se prve testiraju i postepeno se integrišu sa komponentama nižeg nivoa. U ovom slučaju simuliraju se komponente nižeg nivoa, preko takozvanih test stubova.

U praksi je često korisno kombinovati obe metode.

Postoje i drugi oblici testiranja osim testiranja jedinica i integracijskog testiranja. Jedna od mogućnosti je da se testira ceo sistem u odnosu na korisničku dokumentaciju i specifikaciju zahteva nakon što se integraciono testiranje završi. Ovo se zove sistemski test. Sličan tip testiranja se često izvodi pod nadzorom korisničke organizacije i tada se naziva validaciono testiranje. Tokom validacije, naglasak je na testiranju upotrebljivosti sistema, a ne na usklađenosti koda sa nekom specifikacijom. Validacija je glavni kriterijum na kome se zasniva odluka o prihvatanju ili odbijanju sistema.

Poglavlje 6

Održavanje softvera i dokumentacija

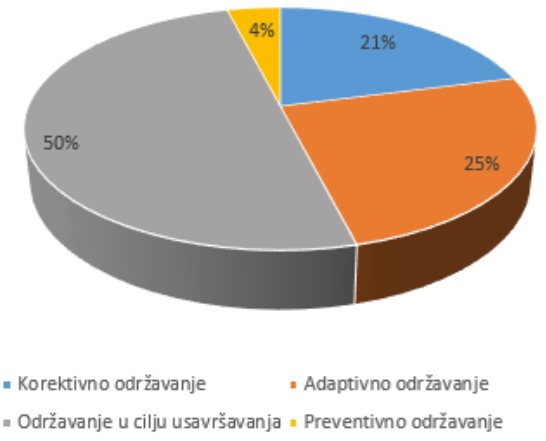
ODRŽAVANJE SOFTVERA

Najveći deo aktivnosti odnosi se na unapređenje softvera

Održavanje softvera nije ograničeno samo na ispravljanje grešaka. S obzirom da softver modeluje deo stvarnosti, a stvarnost se menja, softver takođe mora da se menja i razvija. Veliki procenat zahteva za održavanje zapravo je zahtev za evoluciju softvera. Mogu se razlikovati četiri tipa održavanja:

- Korektivno održavanje - bavi se popravkom pronađenih kvarova.
- Adaptivno održavanje - prilagođavanje softvera promenama u okruženju ali bez promena funkcionalnosti sistema.
- Održavanje u cilju usavršavanja - prilagođavanje novim zahtevima korisnika. Ono se tiče funkcionalnih poboljšanja sistema.
- Preventivno održavanje - povećanje održivosti sistema, kao što je ažuriranje dokumentacije, dodavanje komentara i sl.

Potrebno je imati u vidu da korektivno održavanje predstavlja samo 21% ukupnih aktivnosti, dok je čak 50% vezano za prilagođanje sistema novim potrebama korisnika. Preostalih 25% odnosi se na prilagođavanje softvera promenama u spoljašnjem okruženju.



Slika 6.1 Raspodela aktivnosti održavanja [1,2]

DOKUMENTACIJA SOFTVERA

Svrha dokumentacije određenog softvera jeste da prenosi sve relevantne informacije o njemu

Dokumentacija softvera je veoma važan deo projektovanja sistema. Svrha dokumentacije određenog softvera jeste da prenosi sve relevantne informacije o njemu. Takođe, informacije definisane u dokumentaciji smatraju se obavezujućim pa je neophodno zatražiti reviziju ukoliko je potrebno odstupanje od navedenih zahteva.

Kapri definiše uspešnu dokumentaciju kao onu koja čini informacije lako dostupnim, pruža ograničeni broj korisničkih ulaznih tačaka, pomaže novim korisnicima da uče brzo, pojednostavljuje proizvod i pomaže u smanjenju troškova. Po njegovoj teoriji, postoji nekoliko različitih faza u postupku kreiranja dokumentacije:

- *Analiza* - identifikuju se potencijalni korisnici kojima bi mogla biti potrebna dokumentacija za proizvod i zadaci koji će obavljati na softveru.
- *Dizajn* - kreiranje liste svih stavki identifikovanih tokom faze analize i planiranje sadržaja za svaku.
- *Razvoj* - kreiranje stvarnog dokumenta koji će biti dostavljen.
- *Validacija* - testiranje dokumentacije kako bi se osiguralo da ispunjava svoje ciljeve i potrebe ciljanih korisnika.
- *Kreiranje i proizvodnja* - proizvodnja visokokvalitetnih gotovih proizvoda (papir, audio, CD, onlajn, itd.)
- *Isporuka* - finalni proizvod (softver i dokumentacija) se isporučuje kupcu.
- *Obezbeđivanje zadovoljstva korisnika* - dokument se unapređuje na osnovu potreba korisnika.

ULOGA I UPOTREBA DOKUMENTACIJE

Dokumentacija se koristi kao komunikacioni medij između članova tima, prema klijentima, kao i prema menadžmentu

Parnas [2] je identifikovao nekoliko prednosti dokumentacije:

- lakša ponovna upotreba starih dizajna,
- bolja komunikacija o zahtevima,
- korisniji dizajn recenzije,
- lakša integracija zasebno napisanih modula,
- efikasnija inspekcija koda,
- efikasnije testiranje i
- efikasnije ispravke i poboljšanja.

Načini upotrebe dokumentacije sažeto se mogu predstaviti na sledeći način:

- Koristi se kao komunikacioni medij između članova razvojnog tima i klijenata
- U toku procesa održavanja

- Obezbeđuje informacije za menadžment koji će im pomoći da planiraju, finansiraju i organizuju proces razvoja softvera
- Objašnjava korisnicima kako da koriste i administriraju sistem.

Tipovi dokumentacije:

- *Dokumentacija procesa* - koristi za upravljanje procesom razvoja, na primer za praćenje troškova, definisanje standarda itd.
- *Dokumentacija proizvoda* - opisuje glavni proizvod (softver) i neki od ovih dokumenata su deo konačnog proizvoda koji se isporučuje korisniku. Neki od primera su: specifikacija zahteva, izvorni kod, uputstvo za upotrebu, plan i rezultati validacije i verifikacije, itd.

Poglavlje 7

Vežba - Pokazni primeri

MODELOVANJE, PROJEKTOVANJE, RAZVOJ, TESTIRANJE, ODRŽAVANJE I DOKUMENTOVANJE SOFTVERA ELEKTRONSKOG DNEVNIKA - PRVI ČAS

Zahteve za softver elektronsog dnevnika ćemo prikupiti korisćšenjem radionica i intervijua

U okviru časa vežbi, studentima treba prikazati način prikupljanja zahteva, modelovanja, razvoja, testiranja i održavanja softvera prolaskom kroz sledeće stavke:

1. Prikupljanje zahteva. S obzirom da softverski sistem elektronskog dnevnika sadrži dve aplikacije, kontrolnu i mobilnu aplikaciju, najpogodnije je da zahteve za softverom prikupimo korišćenjem radionica i intervijua. Iz radionica ćemo prikupiti najveći deo zahteva jer ćemo uključiti sve relevantne učesnike dok ćemo na intervijue pozvati sampione proizvoda da sa njima uradimo analizu svega što smo prikupili kroz radionice i upotpuniti zahteve.
2. Modelovanje. Za modelovanje softverskog sistema koristićemo UML dijagrame, klasni dijagram, dijagram veza između entitetima i dijagram toka podataka da bismo videli koji su entiteti u sistemu, koje su veze između njih, kako su organizovani podaci i koji je njihov tok.
3. Projetovanje i razvoj. Dva glavna modula se izdvajaju unutar arhitetkture, modul kontrolne aplikacije i modul mobilne aplikacije. Izdvojićemo modul za rad sa podacima jer će te podatke iste podatke koristiti obe aplikacije. Razvoj ćemo podeliti na frontend deo (timovi za razvoj web i mobilne aplikacije koji će kroz HTTP protokol pozivati serverske funkcije) i backend deo.
4. Testiranje ćemo uraditi na dva nivoa: programeri moraju da pišu unit testove kojima će testirati jedinice koda dok će testeri kasnije testirati celu integraciju.
5. Održavanje softverskog sistema radićemo korektivno (otklanjanjem nedostataka) i adaptivno (eventualne promene u okruženju).
6. Dokumentacija će sadržati informacije o prikupljenim zahtevima, njihovoj analizi, prihvatanju i specifikaciji, zatim informacije modelima u sistemu, o odlukama u arhitekturi i modelima podataka, načinu testiranja.

DRUGI ČAS

Zadatak za grupni rad studenata

Prema stavkama navedenim na prethodnom času studenti grupno analiziraju i daju predloge za način prikupljanja zahteva, modelovanja, projektovanja i izrade, testiranja, održavanja i dokuentacije pri izradi softvera za biblioteku. Sistem se sastoji iz dva dela: administratorski deo, za zaposlene u biblioteci i korisnički deo za korisnike biblioteke koji naručuju knjige.

Poglavlje 8

Vežba - zadaci

ZADACI ZA INDIVIDUALNI RAD STUDENATA

Zadaci za samostalno rešavanje na vežbi i kod kuće

Posle predavanja a pre časova vežbanja (održavaju se dva dana kasnije), poželjno je da pokušate da rešite neke od ovih zadatka radom kod kuće i da rezultate pošaljete mejlom, preko Zimbre, saradniku koji drži vežbe, najkasnije jedan sat pre održavanja vežbi. Na vežbama ćete raditi jedan deo zadataka na individualnoj bazi (svaki student sam radi svoje zadatke) uz pomoć saradnika, ako bude potrebno. Ako neki zadatak nije urađen pre ili za vreme vežbi, preporučuje se studentu da ih uradi posle vežbi, kod kuće.

Obim zadatka može da zahteva najviše 30 do 40 minuta rada studenta. Svaki student mora da ima poseban zadatak, a na temu koja se ovde definiše. Studenti koji pošalju identična ili vrlo bliska rešenja, neće im ta rešenjabiti prihvaćena, tj. dobiće 0 poena.

Tekst domaćeg zadatka:

Odaberite sopstveni primer (različit za svakog studenta) softverskog sistema za koji ćete definisati:

1. Prikažite i objasnite koji je metod prikupljanja zahteva za predloženi sistem.
2. Prikažite i objasnite UML dijagrame koji su od značaja za predloženi softverski sistem.
3. Prikažite celine koje predloženi sistem poseduje.
4. Prikažite na koji način bi se predloženi softverski sistem implementirao.
5. Prikažite na koji način bi se predloženi softverski sistem testirao.
6. Šta treba da sadrži dokumentacija o predloženom softverskom sistemu.

Poglavlje 9

Zaključak

ZAKLJUČAK

Sumiranje stečenih znanja

- U toku inženjerstva zahteva poštuju se zahtevi korisnika koji se odnose kako na funkcije koje treba obezbediti, tako i na niz dodatnih karakteristika (performanse, pouzdanost, dokumentacija itd.).
- Korišćenje odgovarajućih metoda prikupljanja zahteva omogućava da se prilikom prikupljanja zahteva uzimaju u obzir samo relevantne informacije a izostave nevažni detalji.
- Četiri klasične tehnike modelovanja prikazane u lekciji su Entity–relationship model, Konačni automat, Dijagram toka podataka i CRC kartice.
- Softverska arhitektura podrazumeva dekompoziciju sistema na najvišem nivou na glavne komponente zajedno sa karakterizacijom interakcije ovih komponenti.
- Izbor odgovarajuće tehnike za dizajn softvera zavisi od karakteristika problema, odnosno proizvoda koji su predmet dizajna.
- Testiranje predstavlja izvršavanje programa u cilju provere da li proizvodi tačan izlaz za dati ulaz. U zavisnosti od cilja testiranja vrši se odabir odgovarajuće tehnike testiranja.
- Sa aspekta ciljeva testiranja može se napraviti razlika između „verifikacije“ i „validacije“ određenog sistema.
- Četiri tipa održavanja softvera su korektivno, adaptivno, preventivno i održavanje u cilju usavršavanja.
- Kapri definiše uspešnu dokumentaciju kao onu koja čini informacije lako dostupnim, pruža ograničeni broj korisničkih ulaznih tačaka, pomaže novim korisnicima da uče brzo, pojednostavljuje proizvod i pomaže u smanjenju troškova.

LITERATURA

Pogodna literatura za učenje

1. **Obavezna literatura:**

1. Nastavni materijal za e-učenje na predmetu SE101 Razvoj softvera i inženjera softvera, Univeziitet Metropolitan, školska 2022/23. godina
Nastavni materijal je pripremljen korišćenjem reference 2.
2. Software engineering: Principles and Practice, Hans van Vliet, 2007

2. **Dopunska litertura**

1. King Graham, Wang Yingxu, Software Engineering Processes - Principles and Applications,CRC Press (2000)
2. Ian Sommerville,Software Products - An Introduction to Modern Software Engineering, Global Edition, Pearson (2021)