

UVOD U SOFTVERSKO INZENJERSTVO SKRIPTA

[Document subtitle]



# **SDLC**

## Svrha svake od 6 praksi zivotnog ciklusa razvoja softvera

* Inzenjering Zahteva

Spada u prvu fazu razvoja softvera. Podrazumeva razgovor sa klijentom o problemu, sta se trazi, koji su korisnici Sistema, kolika kolicina podataka da se ocekuje, koliki rast podataka, zahtevi dizajna I upotrebe. Sve te informacije koje dobijemo zovemo “User Stories” I na osnovu njih dalje pravimo projekat.

* Specifikacija dizajna

Spada u drugu fazu razvoja softvera. U ovoj fazi na osnovu korisnickih prica kreiramo neke dijagrame klasa, aktivnosti, sekvence kako bi dalje bolje razumeli kako ce sistem da izgleda, I kako ce koja funkcionalnost da radi.

* Implementacija

Spada u trecu fazu razvoja softvera. Tu spada samo pisanje koda.

* Testiranje

Spada u cetvrtu fazu razvoja softvera. Nakon sto smo implementirali neke funkcionalnosti sledi testiranje tih funkcionalnosti pre nego sto se moze poslati klijentu na koriscenje.

* Postavka

Postavka predstavlja postavljanje softvera klijentu, postavljanje neke baze podataka, posebnih alata…

* Odrzavanje

Odrzavanje je poslednja faza razvoja softvera I ona podrazumeva povremena unapredjenja softvera, resavanje bagova…

## Kako je moj tim primenio svaku SDLC praksu u sklopu projekta I kakva unapredjenja ocekujem u sledecem projektu na nivou svake prakse

Na projektu smo preskocili prvu fazu razvoja, a to je inzenjering zahteva jer smo dobili vec gotovu specifikaciju projeka. Na drugu fazu nismo nesto preterano obracali paznju s tim da su funkcionalnosti bile jasne I razumljive, kao I korisnici Sistema I njihove medjusobne veze. Kod implementacije svaki clan tima je dobio po jednog korisnika Sistema I resavao njegove funkcionalnosti. Pre toga smo napisali skripte koje ce popuniti nasu bazu podataka kako ne bi drugi clanovi tima zavisili od tudjih funkcionalnosti. Testiranje softvera je isto bilo povrsno, bez konkretnih test slucajeva, nego vise usput tako sto smo licno unosili vrednosti koje ne zelimo da prodju I videli da li ce proci ili ne. Postavka je isla usput, svaki clan tima kad zavrsi neku funkcionalnost je pravio PR I onda se spajalo na glavnu granu.

Vec na sledecem projektu iz predmeta SIMS smo imali ogromno unapredjenje sto se tice primene ovih praksi. Tu smo prve dve faze u potpunosti primenili, kako inzenjering zahteva, tako I dijagrame. Opet testiranje je bilo minimalno, tako sto smo testirali program u toku rada, ali u sledecim projektima se nadam da cemo na taj deo mnogo vise paznje obratiti.

## Waterfall vs Agile – Razlike, problem

Waterfall I Agile su dve razlicite metodologije razvoja softvera. Agile je iterativna metoda koja ponavlja ceo SDLC na odredjeni vremenski period, dok waterfall za svaki od njih odvoji fiksno vreme, tj ona je sekvencijalna metodologija I taskovi se resavaju u linearnom procesu.

Agile metodologijom projekat ce se kretati kroz niz ciklusa kroz svoj razvoj. Faza razvoja, recenzije, feedbeka, I naravno odobrenja funkcionalnosti. Kod agile metode cemo odmah znati da li se klijentu ne svidja neka nasa funkcionalnost I moci cemo je promeniti u daljim ciklusima razvoja.

Kod waterfall metodologije prvo sve zavrsimo, pa tek onda saljemo klijentu na neki review. Naravno ova metodologija je jako losa, jer klijent nema ideju kako ce proizvod izgledati do samog kraja. Cesto je dolazilo do odbitaka proizvoda ili potputnog ponovnog pisanja.

Problemi:

* Agile: Teze je odrediti koliko ce trajati razvoj softvera, jer mnogo cesce dolazi do nekih znacajnih promena
* Waterfall: Svaka faza mora da se zavrsi pre nego sto sledeca pocne, pa ovo cesto zna da se oduzi. Ne mozemo znati da je doslo do problema sa nekom fazom dok ne dodjemo do sledece. Ovo rezultuje ponovnim radom na prethodnoj fazi. Moramo znati ceo projekat od pocetka do kraja.

# **IDE**

## Prednosti koje donosi editor koda savremenih integrisanih razvojnih okruzenja

Sa savremenim editorima pisanje koda postaje mnogo lakse I intuitivnije.

Imamo automatsko zavrsavanje nasih linija koda, automatsko izdvajanje funkcija, brze kretanje kroz kod, debagovanje koda, refraktorisanje, generisanje nekog koda, odrzavanje cistoce koda, ubrzano pisanje koda…

## Prednosti koje donosi debugger savremenih integ razv okruzenja

Prednosti su naravno sto se mnogo lakse pronalazi problem, mozemo staviti breakpoint na neku liniju koda, ili na vise njih I odatle kretati sa debag rezimom, takodje mozemo staviti uslovne breakpointe koji se stopiraju samo ako vazi uslov. Savremeni debaggeri nam pokazuju trenutne vrednosti nekih promenljivih, takodje mozemo mi da unesemo koje vrednosti zelimo da gledamo, kao I da modifikujemo kod u toku debag rezima, vidimo opterecenje racunara…

## Tehnike za resavanje bagove koje smo analizirali

* Reprodukcija baga – gde mi pokusavamo da ponovo proizvedemo odredjeni bag kako bi nasli nacin da ga otklonimo
* Citanje poruke greske – na osnovu poruke cemo znati na kojoj liniji nam je program puko, I imamo stack trace koji nam govori koje su funkcije uticale na to
* Upotreba debuggera – stavimo breakpoint, upalimo debagger I idemo korak po korak u potrazi za problemom
* Logovanje stanja – pamtimo promene u bazu, neke sistemske procese, greske koje sun am se javljaje tokom transakcija I izvrsenja testova
* Binary chop – na osnovu stack trace-a binarnom pretragom trazimo problem u kodu
* Rubber ducking – pozovemo kolegu nekog kome cemo mi da pricamo sta se desilo, a on da klima glavom, ovde najcesce kazemo stvari koje inace nebi rekli I otkrijemo sta je problem

# GIT

## Odnos radnog stable, pripremne zone, lokalnog git repozitorijuma I komande koje uticu na izmenu ovih prostora

U radon stablo spade cela istorija promena tokom razvoja softvera, kao I trenutne promene, pripremna zona(index) predstavlja skup planiranih izmena za commit I u nju prebacujemo izmene iz radnog stable pomocu komande git add. Lokalni git repozitorijum predstavlja trajnu evidenciju izmena I u njega pravimo izmene komando git commit.

## Interakcije koje postoje izmedju vise repozitorijuma koji rade sa istim projektom I komande koje se koriste da podrze ovu interakciju

Svako ko radi na nekom projektu sa komandom git pull puluje na svoj lokalni repozitorijum najnovije izmene sa udaljene grane, tj centralnog repozitorijuma. U svako trenutku sa komandom git push( ako ima commitova ) moze na udaljenu granu da posalje svoje izmene.

## Prednosti grananja I komande koje se koriste

Prednosti su sto ce svako imati kopiju glavne grane razvoja I sto moze da radi na njoj, a da ne utice na glavnu granu. Sa ovim odrzavamo konzistentnost ispravnosti glavne grane. Za kreiranje nove grane se koristi git branch ime\_grane, dok za promenu grane se koristi git checkout ime\_grane. Kada smo zavrsili promene na grani mozemo

Da odradimo merge te grane na main granu ili neku drugu s tim da ukoliko ima konflikata moramo prvo pullovati najnoviju verziju, I onda merge odraditi I raazresiti konflikte.

## Git workflow I ilustrirati kroz pricu razvoj softvera u vecim timovima sa aspect upotrebe gita

Git workflow je recept ili preporuka kako koristiti Git da ostvarimo rad u konzistentnom I produktivnom rezimu. Ono podstice developere da koriste git efikasno I konzistentno.

U vecim timovima bla bla

## Svojsta dobro formiranih commitova I pull requestova

* Dobro formiran commit: sadrzi povezane izmene, lake za razumevanje, lakse za rollback, pravi se cesto( frekventna integracija promena, manji gubitak posla prilikom skartiranja ), je kompletan (logicka Celina kode, ne samo neki delic, istestiran), prati dobra poruka( postuje conventional commit strukturu, odgovara zasto je napravljena izmena I sta je sustinska promena)
* Dobro formiran PR: Cesto sadrzi vise nekih commitova koji spajaju jednu celinu, ili sadrzi neko resenje za odredjeni bag, unapredjenje… Dobro formiran pr treba da zatvori neke problem, u najboljem slucaju svaki PR za jedan konkretan problem. Ima opis problema, sta je reseno, kako je reseno, takodje imamo informaciju ko ga je kreirao, ko je radio na njemu I koga zelimo da stavimo za reviewera da odobri nas PR.

# **CILJEVI I PLANIRANJE**

## Svojstva SMART ciljeva

Specific – zelim da zaradjujem odredjenu svotu novca

Measurable – zelim da budem strucan

Attainable/Ambitious – zelim da prodam 10 licenci dnevno

Relevant – zelim da implementiram feature x

Time-bound –zelim da pasivno zaradjujem 1k do 2027

## Razlika izmedju zadatka, cilja I misija pojedinca

Misije su putevi do ostvarenja cilja. Cilj je ostvariti nesto u odredjenom roku, dok su zadaci delovi tog cilja koji kad se ispune ostvare cilj.

## Proces planiranja posla kroz work breakdown strukturu

Work breakdown struktura je podela posla u manje zadatke I sa tim je posao upravljiviji I lakse mu je pristupati.

Koraci:

1. Definisemo ciljeve I njihove rokove
2. Razlozimo ciljeve na manje celine: Definisemo medjuzavisnost, Procenimo potrebno vreme
3. Odredimo vremenski tok posla

## Problematika optimizacije nedeljnog rasporeda

Treba razmisliti prvo sta nam je obavezno, sta je jako zahtevno, a sta mozemo sa pola mozga I na osnovu togu organizovati raspored tako da imamo dobar odnos zahtevnog-nezahtevnog rada kako ne bi doslo do preopterecenja I umora.

## Kako alati poput trella, calendar I toggle medjusobno interaguju za efikasnije postizanje cilja

# **KOMUNIKACIJA I TIMSKI RAD**

## Kategorije primera komunikacije koje pronalazimo u inzenjerstvu softvera

Rad u timu:

* Dailiy stand-up – konciznost
* Revizija koda – kritika
* Svakodnevna komunikacija – kolegijalnost
* Iznosenje I odbrana ideja – feedback

Rad van tima:

* Razumevanje korisnika – vrednost
* Prezentovanje inovacija – public speaking
* Dokumentacija – komunikacija
* Svakodnevna komunikacija – know your audience

## Korenski problem koji dovodi do lose komunikacije I konflikta

To je nerazumevanje. Mi zelimo nesto A da se uradi, objasnimo to kao A’, a osoba sa druge strane to razume kao B’. Posledice: See -> Do -> Get. Izazovno je znati objektivnu sliku, izazovno je znati sta je najbolje.

## Koraci komunikacionog algoritma kojim postizemo nenasilnu komunikaciju

Kako davati feedback:

* Pripremiti sagovornika
* Istaknuti konrketne problem
* Istaknuti konkretne posledice
* Pozvati sagovornika na pronalazak resenja

Razumevanje mog problema I pospesivanje njegovog uklanjanja:

* Sta percipiram
* Kako se osecam zbog toga
* Sta su moje potrebe I ocekivanja sto izazivaju ta osecanja
* Kakvu molbu mogu da srocim

## Osobine dobrog clana tima

* Nesebicni
* Skromni
* Iskreni
* Imaju visoka ocekivanja
* Inovativni
* Optimisticki
* Razumljivi

# **CIST DIZAJN KODA**

## Znacajni nazivi

Nazive pronalazimo u svim segmentima razvoja softvera – kod indentifikatora promenljive, funkcije, klase, ali I biblioteke I aplikacije. Jasan naziv funkcije nas oslobadja od citanja njenog tela, dok ce misteriozni naziv zahtevati dodatan mentalni napor da razumemo svrhu koncepta koji opisuje. U Najgorem slucaju, los naziv ce nas naterati da formiramo pogresnu pretpostavku, sto ce nam drasticno produziti vreme razvoja.

Kod znacajnih naziva moramo paziti na:

* Timske konvencije
* Izbegavanje beznacajnih reci
* Primena prikladnog tipa reci
* Terminologija iz domena problema
* Analiza sireg konteksta prilikom formiranja naziva
* Formiranje naziva na dobrom nivou apstrakcije

## Ciste funkcije

Tradicionalna funkcija predstavlja imenovani blok koda koji obuhvata smislen zadatak. U objektno-orijentisanom programiranju funkcije su često metode koje definišu ponašanje objekta nad kojim se pozivaju. Principi koje poštujemo za formiranje čistih funkcija su jednako primenjivi na metode. Čista funkcija je fokusirana na jedan zadatak. Ovaj zadatak je jasno opisan kroz nazive u zaglavlju funkcije, što uključuje naziv funkcije i nazive njenih parametra. Čista funkcija ima jednostavno telo i sastoji se od koda koji zahteva malo mentalnog napora da se razume. Kao posledica, ovakve funkcije često sadrže mali broj linija koda.

## Ciste klase

Klasa nam je šablon po kom instanciramo objekte. Takav objekat poseduje kolekciju atributa koji definišu njegovo stanje, kao i metoda koje opisuju njegovo ponašanje. Klasa je značajno složeniji koncept od funkcije, te su i prateći principi za formiranje čistih klasa složeniji. Tako će čista klasa, donekle slično kao funkcija, biti zadužena za jednu odgovornost. Određivanje odgovornosti klase je složen zadatak koji zahteva da poznajemo ulogu koju klasa igra u sistemu i da odredimo njenu internu koheziju i eksternu spregu sa ostalim klasama u aplikaciji. Kroz ovaj modul ispitujemo dobre i loše prakse za pisanje klasa.

Kod formiranja cistih klasa moramo paziti na:

* Identifikovanje uloge klase koju ima u sistemu
* Strukturalne kohezije klase – stepen upotrebe polja od strane metoda, max kad sve metode koriste sva polja
* Sematicke kohezije klase
* Spregnutost klase sa okolnim klasama

Klaze inzenjerstvu softvera donose: apstrakciju, enkapsulaciju, nasledjivanje, polimorfizam

3 vrste klasa:

* Klasa struktura podataka
* Klasa pametnih objekata
* Klasa koordinatora

Struktura podataka izlaze podatke, nema ponasanje. Objekat sakriva podatke, izlaze ponasanje. Str pod su stanja, glupave grupe podataka(DTO), objekti su ponasanje: pamet I dinamika. Koordinacija je takodje specijalna vrsta ponasanja.

## Doprinos polimorfizma cistoci koda

Objektno-orijentisani jezici uvode koncepte nasleđivanja i interfejsa koji mogu značajno da unaprede kvalitet našeg koda kada se ispravno koriste, kao i da značajno redukuju njegov kvalitet kada se primenjuju bez razmišljanja. Oba mehanizma doprinose redukciji dupliranja koda, no glavna prednost koju donose je mogućnost stvaranja proširivog softvera uz pomoć polimorfizma. Kroz ovu lekciju analiziramo kada i na koji način treba koristiti nasleđivanje, a kada interfejse kako bismo unapredili čistoću i proširivost našeg softvera.

Cist polimorfizan prati sledece principe:

- Prepoznavanje potrebe za hijerarhijom

* Refraktorisanje nepotrebne hijerarhije
* Definisanje interfejsa fokusiranih na jednu odgovornost

## Zlo kod komentara

* Mumlanje – previse informacija
* Redundantnost – za neke proste jednolinijske funkcije napisan komentar
* Laz - pogresan opis funkcije/dela koda
* Zakomentarisan kod

## **SOLID**

### SRP – Single Responsibility Principle

Klasa treba da ima samo jedan razlog da se promeni, sto znaci da klasa treba da ima samo jednu odgovornost. Sto znaci da svaka klasa u kodu treba da radi samo jedan posao. Sve u toj klasi treba da je okrenuto jednoj svrsi. SRP nam daje dobar nacin da identifikujemo klase u fazi dizajniranja aplikacije I natera te da mislis kako bi klasu dalje mogao menjati.

SRP je primenljiv na razne apstrakcije: Klase, pakete, mikroservise.

SRP donosi vise prednosti: povecana citljivost, olaksane izmene, manja sansa za runtime greske

### OCP – Open Closed Principle

OCP kaze da Softverski modul/klasa je otvorena za ekstenziju I zatvorena za modifikaciju. Otvorena za ekstenziju znaci da treba da dizajniramo nas modul/klasu tako da nove funkcionalnosti se mogu dodati samo kad dodje do novih potreba. Zatvorena za modifikaciju znaci da smo vec razvili klasu I da je prosla kroz testiranja. Nakon uspesnih testiranja ne smemo menjati tu klasu dok ne dodjemo do baga. Za prosirenja se moze naravno koristiti nasledjivanje.

OCP zahteva procenu: sta konkretizovati, sta apstraktovati

OCP donosi vise prednosti: fleksibilnost, ponovna iskoristivost, olaksano odrzavanje

### LSP – Liskov Substitution Principle

LCP kaze da mozemo koristiti svaku klasu naslednika umesto roditeljske klase I ona mora da se ponasa na isti nacin bez modifikacija. Ono osigurava da klasa naslednica ne utice na ponasanje roditeljske klase. Ovaj princip je ekstenzija OCP I znaci da moramo da osiguramo da klasa naslednica prosiri roditeljsku klasu bez da menja njeno ponasanje.

Krsenje LSP-a:

* Cast u kodu klijenta nadklase (eksplicitan ili instanceOf)
* Kada podklasa ima zahtevnije preduslove za izvrsavanje ponasanja koje nasledjuje
* Kada podklasa ima slabije postuslove kao rezultat ponasanja koje nasledjuje
* Kada podklasi nije potreban deo metoda I polja nadklase(refused bequest)

LSP zahteva brigu o apstrakciji: da li override menja pred I post uslove koje ocekuju klijenti, da li je data klasa potomak nadklase

LSP donosi vise prednosti: ispravnost polimorfizma, smislene hijerarhije

### ISP – Interface Segregation Principle

ISP kaze da klijent ne sme da bude nateran da implementira interfejse koje ne koristi. Umesto jednog velikog interfejsa, mnogo malih interfejsa su preferirani na osnovu grupe metoda. Drugim recima, interfejs bi trebao biti blize povezan kodu koji ga koristi osim kodu koji ga implementira. Tako da metode interfejsa su definisane na osnovu metoda koje klijent treba, a ne na osnovu metoda koje klasa implementira. Kao I klase I interfejsi treba da imaju SRP. Ne bi trebali implementirati interfejs ako nas objekat ne deli istu svrhu sa njim.

ISP podrzava druge principe: implementacija debelog interfejsa narusava SRP, implementacija dela debelog interfejsa narusava LSP

ISP istice dobar dizajn: implementacije, klijenta

### DIP – Dependency Inversion Principle

DIP kaze da klase/moduli visokog nivoa ne bi trebali da zavise od klasa/modula niskog nivoa. Oba treba da zavise na osnovu apstrakcija. Pored toga, apstrakcije ne bi trebalo da zavise na osnovu detalja. Detalji treba da zavise na osnovu apstrakcija. Klase visokog nivoa implementiraju biznis pravila ili logiku u sistemu. Klase niskog nivoa rade sa detaljnijim operacijama, kao sto su pisanje informacija u bazu, prosledjivanje poruka operativnom sistemu..

Klasa visokog nivoa koja zavisi od klase niskog nivoa I koja zna mnogo o toj klasi sa kojom je u interakciji je tightly coupled klasa. Ako previse zna o implementaciji druge klase povecava se rizik da ce jedna klasa da srusi drugu. Tako da ih moramo drzati loosely coupled.

DIP I DI su osnovni mehanizmi za ono sto zovemo prosirivost u OO jezicima.

## Opisi dependency inversion princip, prateci dependency injection mehanizam I prednosti koje oba pojma donose

Dependency inversion je dizajn princip.

Dependency injection je dizajn pattern koji omogucava da razdvojimo kreiranje od upotrebe. Dozvoljava nam da injectujemo potrebne objekte tokom run-time-a, a da ne brinemo o tome da ih mi konstruisemo. Radi ruku uz ruku sa dep inversion principom. Npr mozemo da zavisimo od abstraktnih klasa visokog nivoa I da injectujemo specificnu implementaciju tokom run-time-a , na osnovu potrebe.

## Svojstva I prednosti cistih paketa

LIFT princip organizacije paketa

Locate – intuitivno I jednostavno pronadji sta ti treba

Identify – razumi sadrzaj spram naziva

Flat – izbegavati duboko ugnjezdavanje

Try to be dry – izbegavati redundantnost u nazivu

Prednosti su dobra citljivost, laka snalazljivost kroz kod…

## Segmenti ciste arhitekture

* Hexagonijalna arhitektura
* Onion arhitektura
* Screaming arhitektura
* DCI
* BCE

Svi ovi segmenti su:

* Nezavisni od frejmvorka.
* Testivi
* Nezavisni od UI-ja
* Nezavisni od baze
* Nezavisni od eksternih stvari

## Razradjena SDLC praksa implementacije koja podrazumeva aktivnosti pisanja koda, refraktorisanja, revizije

Dogovorite timske konvencije:

* Cist kod pospesuje komunikaciju
* Kod koji prati timska pravila pospesuje timsku komunikacij
* Paketiranje, imenovanje, indentacija, formatiranje, sirina koda, duzina datoteka...

Vecina troska razvoja je u citanju koda. Cist dizajn koda znacajno smanjuje troskove.

Prljav kod se pise zbog nedostatka vestine I brige inzenjera.

Treba stalno da izdvajamo vreme za refraktorisanje koda.

Postupak je sledeci:

* Pristigao zahtev
* Postojeci kod treba Refraktorisati?
  + Treba – Refraktorisi/enkapsuliraj
  + Ne treba – kodiraj da proradi -> refraktorisi

Code smell-ovi su structure u kodu koji ukazuju na potrebu za refraktorisanjem. To su misteriozna imena, komentari, duge funkcije, velike klase, globalni podaci…

## 