



Funded by the
Erasmus+ Programme
of the European Union



This project has been funded with support from the European Commission. This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.



KI206 - PROCES I METODOLOGIJE RAZVOJA SOFTVERA

Evolucija softvera

Lekcija 08

PRIRUČNIK ZA STUDENTE

KI206 - PROCES I METODOLOGIJE RAZVOJA SOFTVERA

Lekcija 08

EVOLUCIJA SOFTVERA

- ▼ Evolucija softvera
- ▼ Poglavlje 1: Procesi evolucije softvera
- ▼ Poglavlje 2: Dinamika programa evolucije
- ▼ Poglavlje 3: Održavanje softvera
- ▼ Poglavlje 4: Upravljanje starim softverom
- ▼ Poglavlje 5: Radionica - Jenkins alat za stalnu integraciju SW
- ▼ Zaključak

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

✓ Uvod

UVOD

Posle puštanja u rad prve verzije softvera, nastaje dugi period njegovog evolutivnog menjanja u toku njegovog životnog veka.

Cilj ove lekcije da vam objasni zašto je evolucija softvera važan deo softverskog inženjerstva i da vam opiše proces evolucije softvera. Ova lekcija vam omogućava da:

- razumete da je promena neizbežna ako softverski sistemi treba da ostanu korisni i da su razvoj softvera i njegova evolucija integrисани u spiralnom modelu razvoja;
- razumete procese evolucije softvera i uticajne faktore koji deluju na ove procese;
- naučite o različitim tipovima održavanja softvera i faktorima koji utiču na troškove održavanja; i
- razumete kako nasleđeni sistemi se mogu procenjivati radi odlučivanja da li da se odbace, održavaju, poprave ili zamene

▼ Poglavlje 1

Procesi evolucije softvera

ŠTA JE EVOLUCIJA SOFTVERA?

Softversko inženjerstvo je jedan spiralni proces razvoja i evolucije, sa zahtevima, projektovanjem, implementacijom i testiranjem softvera tokom celog njegovog životnog veka

Isporučen softver se mora da menja iz bar dva razloga.

1. da bi se otklanjale uočeni nedostaci i greške.
2. da bi se prilagođavao novim zahtevima korisnika.

Zbog toga, jednom urađen softver, ceo svoj radni vek najčešće prolazi kroz razne vrste iymena i dopuna, tj. On je u stalnoj evoluciji. Evolucija softvera je stalna promena softvera posle njegovog inicijalnog razvojai isporuke naručiocu, ili tržištu. .

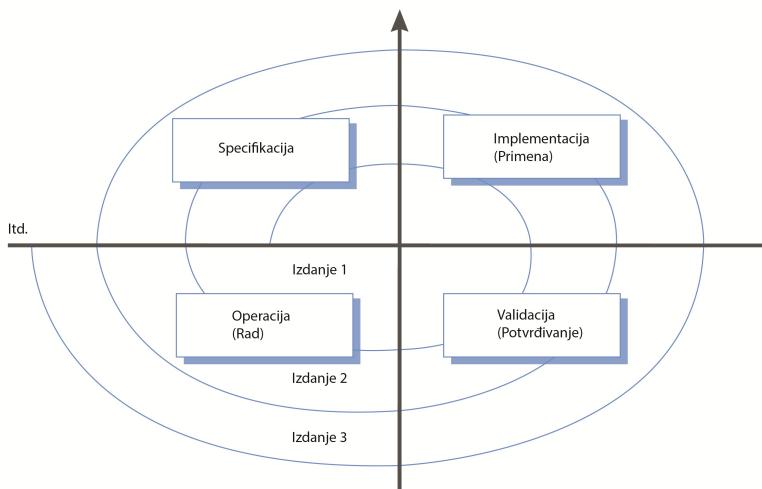
Neke analize pokazuju, da

- oko 85 do 90% troškova vezanih za softver kompanije troše za evoluciju softvera.
- da troškovi evolucije čine dve trećine ukupnih troškova za softver

Kompanija koja je razvila takav softver najčešće predaje odgovornost za dalji razvoj i modifikacije softvera kompaniji koja ga je naručila i koja mora da ima odgovarajući kadar za takav razvoj. Druga varijanta je da kompanija angažuje neku drugu organizaciju da podržava evolutivan razvoj njenog softvera.

Iz navedenih razloga, treba posmatrati softversko inženjerstvo kao jedan spiralni proces sa zahtevima, projektovanjem, implementacijom i testiranjem tokom celog njegovog životnog veka.

Posle verzije 1, ciklusni razvoj se nastavlja, sa navedenim osnovnim aktivnostima, te se dobija verzija 2, pa 3 itd. (slika 1).



Slika 1.1 Spiralni model razvoja i evolucija

Ovaj model evolucije softvera podrazumeva da je jedna organizacija odgovorna i za njegov inicijalni razvoj, a i za njegovu evoluciju. To je model koji se najčešće koristi kod softverskih paketa, koji su razvijeni za prodaju na tržištu.

EVOLUCIJA I SERVISIRANJE SOFTVERA

Evolucija je faza u kojoj se vrše značajne promene u arhitekturi softvera i njegovoj funkcionalnosti. Za vreme servisiranja, vrše se sitnije promene, ali neophodne promene

Firma koja je inicijalno razvila softver, može da sklopi ugovor sa drugom firmom koja onda preuzima posao evolucije tog softvera. To može da dovede do prekida u procesu, tj. do diskontinuiteta. Na primer, može da se desi, da dokumentacija o zahtevima i projektnom rešenju ne bude poznata organizaciji – partneru, koja vrši dalju evoluciju softvera. Kompanije, takođe, se mogu spajati ili reorganizovati, što dovodi do uvođenja novog softvera., npr. firme sa kojom se spajaju. Onda, najčešće utvrde da moraju da izvrše određene promene na softveru.

Kada prelazak iz faze razvoje softvera, u fazu evolucije posle isporuke softvera, nije neprimetan, te se ta faza često naziva "**održavanjem softvera**".

Na slici 2 prikazan je alternativan model evolucije softvera tokom njegovog životnog veka. Model razlikuje fazu evolucije od faze servisiranja. **Evolucija** je faza u kojoj se vrše značajne promene u arhitekturi softvera i njegovoj funkcionalnosti. Za vreme **servisiranja**, vrše se sitnije promene, ali neophodne promene.

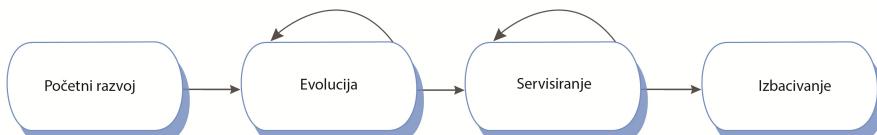
Za vreme evolucije, softver se uspešno koristi, i paralelno postoji tok stalnih promena u zahtevima. Međutim, sa promenama softvera, menja se i njegova struktura, što posetepeno dovodi do njegove degradacije, tj. opadanja performansi softvera.

Promene postaju teže i samim tim, i skuplje. To se obično dešava posle nekoliko godina korišćenja softvera, kada se vrše i promene hardverskog i softverskog okruženja u kome

softver radi. U jednom momentu, značajne promene softvera, zbog novih zahteva, postaju sve manje isplative. U tom momentu, sistem prelazi iz faze efolucije u fazu servisiranja.,.

Za vreme faze servisiranja, softver je i dalje koristan, ali se na njemu rade samo male, neophodne izmene. U to vreme, kompanija počinje da razmišlja o zameni softvera novim. I

U konačnoj fazi (phase-out), softver se i dalje koristi, ali se vrše ne vrše nikakve izmene na njemu. Korisnici moraju sami da reše probleme na koje najdu.



Slika 1.2 Evolucija i servisiranje

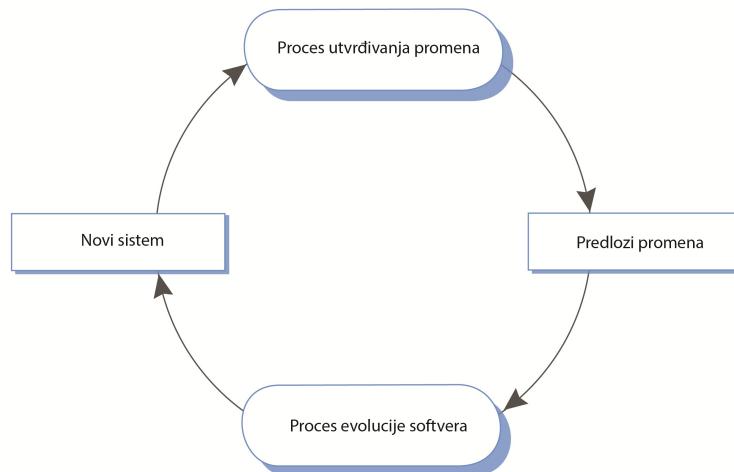
PROCESI EVOLUCIJE SOFTVERA

Glavni pokretač evolucije softvera u svim organizacijama su predlozi za promenu sistema

Procesi evolucije softvera zavise od tipa softvera, od proces razvoja softvera koji se koriste u organizaciji i od veština ljudi koji u tome učestvuju. U nekim organizacijama, evolucija softvera je neformalni proces u kome se zahtevi za promenama formulišu razgovorima između korisnika softvera i inženjera razvoja. U drugim organizacijama, to može biti jedan formalizovan proces, sa strukturisanom dokumentacijom u svakoj fazi procesa.

Glavni pokretač evolucije softvera u svim organizacijama su predlozi za promenu sistema. Oni dolaze iz potrebe da se realizuju još nerealizovani zahtevi sistema, zbog utvrđenih novih zahteva, zbog izveštaja o greškama u sistemu, a i zbog javljanja novih ideja za poboljšanje softvera od strane razvojnog tima. Procesi utvrđivanja zahteva sistema evolucije su cikličnog karaktera i ponavljaju se tokom životnog ciklusa sistema (slika 3)

Zahtevi za promenom softvera bi trebalo da ukazuju na komponentu softvera koja se treba modifikovati. Na taj način se mogu proceniti troškovi realizacije te promene kao i njen efekat.

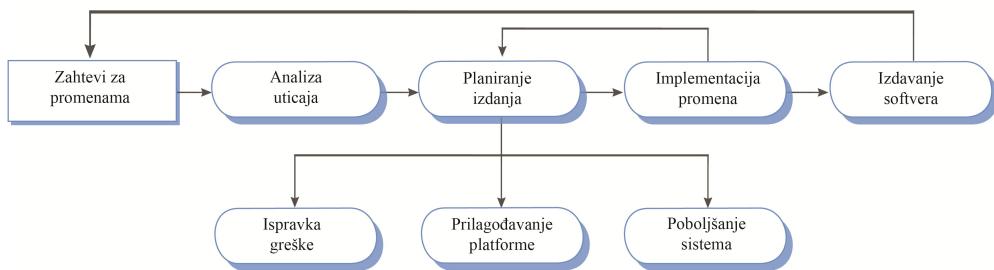


Slika 1.3 Procesi utvrđivanje zahteva za promenom i evolucije

OSNOVNE AKTIVNOSTI PROCESA EVOLUCIJE

Troškovi i efekat promena se ocenjuju da bi se donela odluka o prihvatanju

Na slici 4 prikazan je proces evolucije softvera (Arthur, 1988) koji uključuje osnovne aktivnosti procesa. Troškovi i efekat promena se ocenjuju da bi se donela odluka o prihvatanju. Ako je ona pozitivna, onda ona određuje i verziju softvera u kojoj će ta promena biti primenjena. Kada su promene promenjene i potvrđene, izdaje se novo izdanje (verzija) softvera. Ovaj ciklusni proces se stalno ponavlja.



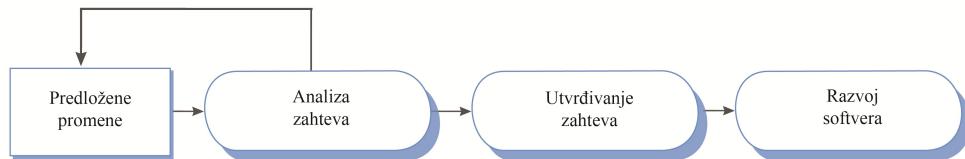
Slika 1.4 Proces evolucije softvera

IMPLEMENTACIJA PROMENE

Mora se proceniti da li promena neće izazvati neke nove probleme.

Proces evolucije procesa i uvođenja promena u softver je jedan iterativni proces razvoja, u kome se nova izdanja softvera projektuju, implementiraju i testiraju. Specifičnost je u prvoj fazi primene promena mora da postoji dobro razumevanje samog softvera, jer u ovom procesu prvo bitni razvojni tim nije više odgovaran za implementaciju promena. Mora se proceniti da li neće promena da izazove neke nove probleme.

U idealnom slučaju, u fazi implementacije promena, trebalo bi izvršiti promenu specifikacije sistema, projektnih rešenja i implementacije softvera, u skladu sa izvršenim promenama (slika 5). Predložene promene se analiziraju i potvrđuju (validacija). Komponente sisteme se ponovo projektuju i implementiraju, a sistem testira.



Slika 1.5 Implementacija promene

Zahtevi za promenom sistema ponekad se izazvani problemima u sistemu i koji traže hitnu reakciju. Ove urgentne promene su uslovljene sledećim razlozima:

1. Javljanje ozbiljne greške u sistemu koja se mora otkloniti da bi sistem mogao normalno da radi.
2. Došlo je do promenama u radnom okruženju koje imaju neočekivane efekte i sprečavaju normalan rad sistema.
3. Došlo je do promena u načinu poslovanja (novi konkurenti, novi propisi) koji zahtevaju hitne promene u sistemu.

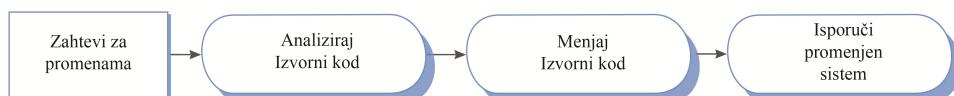
PROCES HITNE OPRAVKE I PROMENE SISTEMA

Sistemi koji imaju puno hitnih ispravki u svojoj istoriji (menja se kod, ali ne i projektna dokumentacija) brže stare, jer nove promene postaju teže izvodljive i teže se održavaju

Zbog hitnosti, ove promene se ne mogu realizovati normalnom procedurom (slika 5), već se i bez promene zahteva i projektnog rešenja, odmah pristupa promeni koda (slika 6). Opasnost je, naravno, da se time ne poremeti konsistentnost softvera, jer se naknadne dorade dokumentacije obično ne vrše, te stvarni sistem odudara od sistema u dokumentaciji.

Zbog hitnosti, hitne intervencije ne obezbeđuju i najbolji mogući način promene. Zato, sistemi koji imaju puno hitnih ispravki u svojoj istoriji, brže stare, jer nove promene postaju teže izvodljive i teže se održavaju (tzv. „špageti problem“). Normalno bi bilo, da po izvršenoj hitnoj intervenciji na sistema, da se sproveđe pažljiva analiza, i ako je potrebno, promena izvede na drugojačiji način, i dokumentuje. Ali, u praksi se to retko dešava, jer te aktivnosti obično nemaju

visok prioritet i to vodi ubrzanom pogoršanju softverske strukture i skraćivanju njegovog životnog veka.



Slika 1.6 Proces hitne opravke i promene sistema.

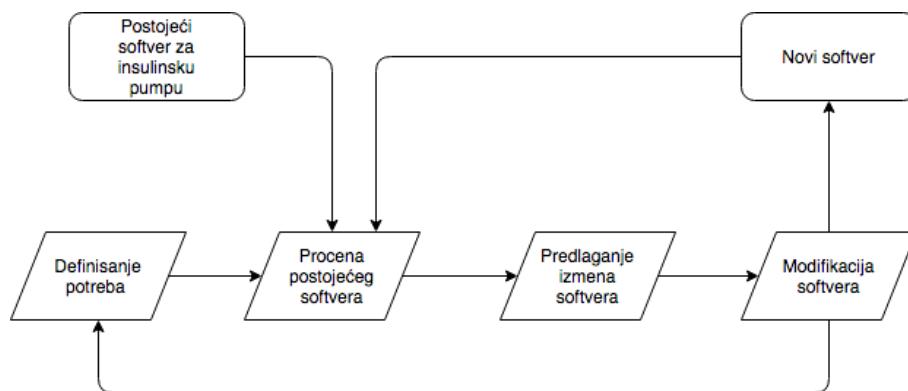
Kod primene agilnih metoda razvoja softvera, zbog inkrementalnog razvoja softvera, prelazak iz procesa razvoja u proces evolucije je neosetan. Primena tehnika automatskog regresionog testiranja je koristan kada se vrše promene u sistemu. Praktično, evolucija samo produžava proces agilnog razvoja softvera.

Mogu se javiti problemi kada razvojni tip predaje dalju nadležnost tima za evoluciju softvera:

1. Problem se javlja kada je razvojni tim primenjivao agilne metode, a tim za evoluciju primenjuje planski pristup u razvoju softvera. Timu za evoluciju nedostaje potpunija dokumentacija o softveru.
2. Problem se javlja i u suprotnom slučaju, kada je razvojni tim primenjivao planski pristup u razvoju softvera, a tim za evoluciju primenjuje agilne metode. U tom slučaju tim za evoluciju mora da počne niodčega kada definiše automatske testove te ne dolazi do uprošćavanja koda, kao što se očekuje od agilnog razvoja. U ovakovom slučaju, reinženjerинг softvera može biti rešenje problema, kako bi se on poboljšao pre nego što se primene agilne metode u fazi evolucije softvera.

PRIMER EVOLUCIJE SOFTVERA

Dat je primer evolucije softvera.



Slika 1.7 Primer evolucije softvera za upravljanje insulin pumpom

Na slici 7 se može videti evolucija softvera za upravljanje insulin pumpom. Za početak potrebno je analizirati i proceniti sve mogućnosti već postojećeg softvera. Analizirati funkcije za merenje nivoa šećera u krvi i doze insulina. Kada se taj korak završi, potrebno je predložiti izmene koje bi mogle unaprediti opisane funkcionalnosti samog softvera. Zatim se prelazi na korak modifikacije softvera a nakon tog koraka se može kreirati novi softver ili ponovo definisati potrebe korisnika

ZADACI ZA SAMOSTALNI RAD

Dati su zadaci koji se odnose na evoluciju softvera.

- 1. Zadatak:** Nacrtati dijagram evolucije softvera za rezervaciju karata za pozorište.
- 2. Zadatak:** Detaljno opisati aktivnosti u svakoj od navedenih faza evolucije softvera.

▼ Poglavlje 2

Dinamika programa evolucije

LEHMANOVI ZAKONI

Lehman-ovi zakoni iskazuju izvesne uočene zakonitosti u evoluciji softvera.

Pojedini istraživači (Lehman i Belady) uočili su izvesne zakonitosti u evoluciji softvera:

1. Održavanje sistema je jedan neizbežan proces. Kako se okruženje sistema menja, javljaju se novi zahtevi i sistem se mora menjati. Kada se novi sistem aktivira, on utiče na okruženje, te se i ono menja, te proces evolucije nastavlja svoj ciklični razvoj.
2. Kako se sistem menja, tako se njegova struktura pogoršava. Preventivno održavanje je jedini način da se ovo izbegne. To znači da je neophodno da se investira vreme i novac za promenu strukture softvera bez promene njegove funkcionalnosti.
3. Veliki sistemi imaju svoju dinamiku koja je postavljena u ranoj fazi procesa razvoja. To određuje dalji trend procesa održavanja softvera i ograničava broj mogućih promena. Zakon je posledica uticaja strukturnih faktora koji utiču i ograničavaju promene sistema, i organizacionih faktora (brzina donošenja odluka) koji utiču na proces evolucije.
4. Veliki projekti programiranja rade u stanju „zasićenja“. To znači da neka promena u resursima ili ljudima ima neprimetan uticaj na evoluciju sistema u dužem periodu. To znači, kao i u 3. Zakonu, da evolucija programa je dosta nezavisna od odluka menadžmenta. Zakon potvrđuje da su veliki timovi razvoja često neproduktivni zbog složenih komunikacija koje dominiraju nad njihovim radom.
5. *Dodavanje nove funkcionalnosti sistemu neminovno unosi nove greške sistema. Što više novih funkcija unosite, unosite i više grešaka u sistem. To traži ubrzo novo izdanje softvera sa ispravkama ovih grešaka. Znači, ne pravite inkremente sa velikim promenama funkcionalnosti ako niste spremni da ubrzo pravite novo izdanje da bi ispravili unete greške.*

Pored ovih pet zakona, Lehman je kasnije dodao nove. Šesti i sedmi zakon su slični i kažu da korisnici softvera postaju nezadovoljni ako se on ne održava i ako se ne dodaju nove funkcionalnosti. Poslednji, osmi zakon se bavi povratnim procesima.

.

SEDAM LEHMAN-OVIH ZAKONA

Lehman-ove zakone treba uzeti u obzir kada planirate proces održavanja softvera.

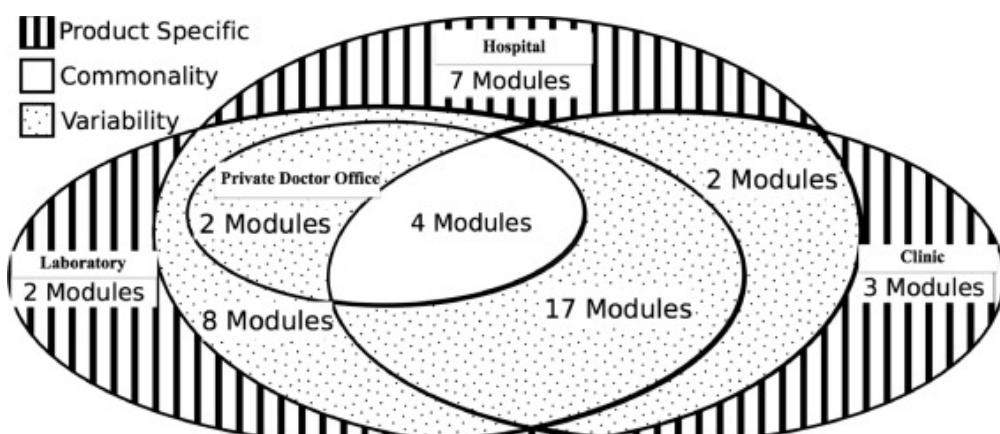
Na slici 1 prikazani su svi Lehmanovi zakoni u celosti. Ove zakone treba uzeti u obzir kada planirate proces održavanja

Zakon	Opis
Kontinualna promena	Program koji se koristi u stvarnom okruženju mora da se menja ili postaje progresivno nekoristan za to okruženje.
Povećanje složenosti	Sa promenama u softveru, njegova struktura postaje složenija. Moraju se odvojiti dodatni resursi za održavanje i uproščavanje strukture.
Velika evolucija programa	Program evolucije i samoregulišući proces. Atributi sistema, kao što su veličin, vreme između izdanja, i broj prijavljenih grešaka je približno nepromenljiv u svakom izdanju.
Organizaciona stabilnost	U toku životnog veka programa, njegova brzina menjanja je približno konstantna i nezavisna od dodeljenih resursa u njegovom razvoju.
Konzervacija familijarnosti	Za vreme životnog veka sistema, incrementalna promena u svakom izdanju je približno ista.
Stalni rast	Funkcionalnost sistema konstantno raste da bi zadržala zadovoljstvo korisnika.
Opadajući kvalitet	Kvalitet sistema opada ukoliko se ne menja u skladu sa promenama u njegovom radnom okruženju.
Povratni sistem	Procesi evolucije sadrža višestruke povratne sprege i višestruke agente i treba da ih tretirate kao povratne sisteme da bi ostvarili značajno poboljšanje proizvoda.

Slika 2.1

PRIMER PRIMENE LEHMANOVIH ZAKONA

Dat je primer primene Lehmanovih zakona unutar procesa održavanja softvera.



Slika 2.2 Primer upotrebe Lehmanovih zakona

Na slici 2 prikazan je sistem bolnice koji se sastoji od niza modula (bolnica, laboratorija, klinika, privatna doktorska ordinacija). Sistem zahteva kontinualnu promenu shodno novim zahtevima korisnika, od početka razvojnog toka povećana je složenost samog sistema a samim tim i organizaciona stabilnost u samom sistemu. U okviru ovog primera primenjeno je četiri Lehmanova zakona.

ZADACI ZA SAMOSTALNI RAD

Dati su zadaci koji se odnose na primenu Lehmanovih zakona.

Zadatak 1: Prikažite primenu Lehmanovih zakona u toku procesa održavanja softvera za elektronsko bankarstvo. Da li je moguće primeniti sve zakone u procesu održavanja softvera za elektronsko bankarstvo?

Zadatak 2: Navedite razliku između primene Lehmanovih zakona u procesu održavanja velikih sistema i unutar procesa održavanja malih sistema.

✓ Poglavlje 3

Održavanje softvera

SADRŽAJ

Održavanje softvera praćeno je značajnim troškovima jer se u dugom periodu vrše povremene manje ili veće dorade softvera.

U ovom poglavlju, izlažu se sledeći aspekti održavanja softvera:

1. Vrste i troškovi održavanja
2. Previđanje troškova održavanja softvera
3. Reinženjering softvera
4. Preventivno održavanje

✓ 3.1 Vrste i troškovi održavanja

VRSTE ODRŽAVANJA

Održavanje softvera je proces menjanja sistema posle njegove isporuke

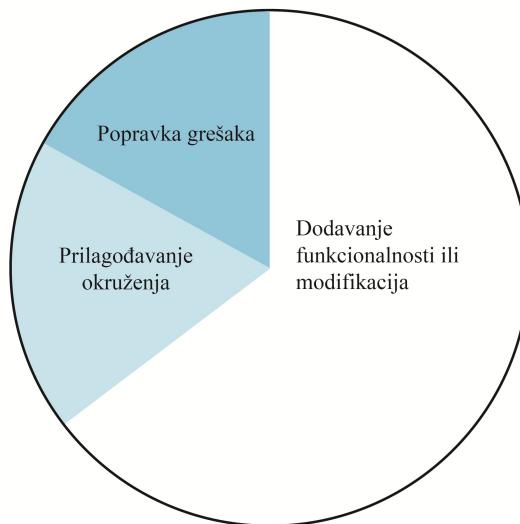
Održavanje softvera je proces menjanja sistema posle njegove isporuke. Promene obuhvataju otklanjanje grešaka u programiranju, ali i veće greške koje otklanjaju greške u projektnom rešenju ili u specifikaciji sistema, ali u zadovoljenju novih zahteva. Implementacija promene postojećeg sistema se primenjuje promenom postojećih komponenti sistema, ali i dodavanjem novih.

Postoji tri tipa održavanja softvera:

1. **Popravka greški:** Otklanjanje greški u kodu (najmanji troškovi popravke), u projektnom rešenju (veći troškovi) jer izaziva promenu nekoliko komponenti, i u zahtevima (najveći troškovi), jer može da zahteva i redizajn sistema.
2. **Prilagođenje okruženju:** Ova promena u softveru je izazvana promenama platforme i okruženja u kome radi sistem (hardver, softver).
3. **Dodavanje funkcionalnosti:** Promene u softveru su nužne zbog zadovoljenja novih organizacijskih ili poslovnih promena. Ove promene sistema su obično znatno ozbiljnije nego u prethodna dva slučaja.

U praksi se koriste različiti nazivi za ove tipove održavanje softvera, a i granice između ovih tipova održavanja nisu uvek jasno određene.

Istraživanja pokazuju da *troškovi održavanja u IT budžetu imaju veći udeo nego razvoj novog softvera* (oko 2/3 održavanje, 1/3 novi razvoj). Veći deo troškova održavanja su vezani na zadovoljenju novih zahteva ili prilagođavanju okruženju, nego otklanjanju grešaka u softveru (slika 1).



Slika 3.1.1 Distribucija troškova održavanja softvera

TROŠKOVI ODRŽAVANJA SOFTVERA

Ukupni troškovi održavanja softvera u toku njegovog životnog veka mogu da se smanje u odnosu na troškove razvoja ako se o održavanju vodi računa još u fazi razvoja softvera

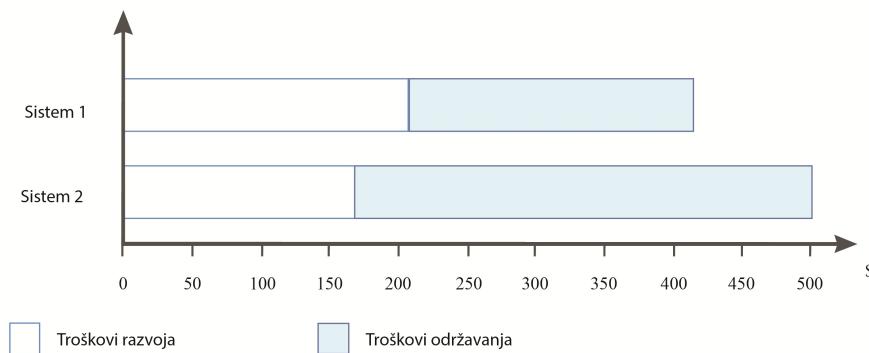
U slučaju tzv. **ugrađenih sistema** koji rade u realnom vremenu, troškovi održavanja su i četiri puta veći nego troškovi razvoja. Razlog za ovo zadovoljenja zahteva pouzdanosti i performansi koje prouzrokuju visok stepen integracije njihovih komponenata, te zbog toga, troškovi održavanja su veći. Doduše, primenom objektno-orientisanih programiranja u razvoju ugrađenih sistema dovodi do smanjivanju troškova održavanja jer olakšavaju promenu njihovih komponenti.

S ciljem da se smanje troškovi održavanja softvera, ulažu se naporci da se razviju metodi razvoja softvera koji će voditi računa o održavanju još u fazi razvoja softvera (npr. primena tehnikе dobijanje precizne specifikacije, primena objektno-orientisanog razvoja, upravljanje konfiguracijom softvera).

Slika 2 pokazuje kako ukupni troškovi održavanja softvera u toku njegovog životnog veka mogu da se smanje u odnosu na troškove razvoja ako se o održavanju vodi računa još u fazi razvoja softvera. U slučaju sistem 1 trošak razvoja je bio uvećan za 25.000 dolara, zbog vođenja računa o održavanju softvera. Ovo je dovelo do ušteda od \$100.000 u održavanju

ovog sistema u toku njegovog životnog veka. Očigledno, više ulaganja u razvoj softvera dovodi do opadanja troškova održavanja softvera kasnije.

Da bi se smanjili troškovi održavanja softvera koji se razvija **metodama agilnog razvoja**, neophodno je menjati i njegovu strukturu (engl. refactoring), kako bi sporije „stario“. Kod **planski vođenom razvoju softvera** retko se ulažu dodatni napor da se kod učini lakšim za održavanje, jer to povećava troškove razvoja. S druge strane, uštede u održavanju dolaze mnogo kasnije, što nije mnogo atraktivno za mnoge kompanije koje se bore za opstanak na tržištu ili su usmerene ka ostvarivanju što većih profita (zbog pritiska akcionara i bonusa za menadžment).



Slika 3.1.2 Troškovi razvoja i održavanja softvera

RAZLOZI ZA POVEĆANJE TROŠKOVA ODRŽAVANJA

Skuplje je dodati novu funkcionalnost već postojećem softveru nego je ugraditi pri njegovom razvoju

Skuplje je dodati novu funkcionalnost već postojećem softveru nego je ugraditi pri njegovom razvoju, iz sledećih razloga:

1. **Stabilnost tima:** Održavanje obično radi novi tim koji ne razume sistem tako dobro kao tim koji ga je razvio, te im treba više vremena za razumevanje sistema, pre nego što ga menjaju.
2. **Loša praksa razvoja:** Ugovor o održavanju je obično nezavisan od ugovora o razvoju sistema. Taj posao može biti dat i nekoj kompaniji koji i nije razvila softver. Zato, kompanija koja razvija sistem, nema motiv da pri razvoju softvera vodi računa o njegovom kasnjem održavanju. Čak suprotno, razvojni tim je motivisan da smanji što više troškove razvoja, ne vodeći računa da to može da dovede do većih troškova održavanja u budućnosti.
3. **Veštine zaposlenih:** Zaposleni koji se bave održavanjem je obično manje iskusani i ne poznaje dobro oblast primene softvera. Pored toga, stari sistemi su često pisani u programskim jezicima koji su prevaziđeni, te ljudi u održavanju nemaju dovoljno iskustva sa tim jezicima. Sve ovo povećava troškove održavanja.
4. **Starost programa i njegova struktura:** S promenama u softveru, postepeno se kvari njegova struktura, te on postaje teži za razumevanje i održavanje. Stari sistemi nisu ni razvijeni primenom modernih tehnika razvoja softvera. Pri njihovom razvoju, njihova struktura

nije optimizovana za razumevanje koda, već za njegovu efikasnost. Dokumentacija sistema je ili izgubljena ili je nekonzistentna. Stari sistemi ne koriste upravljanje konfiguracijama, te je često otežano nalaženje prave verzije njegovih komponenti.

Prva tri razloga su vezana za problem shvatanja u industriji softvera. Mnoge organizacije i dalje vide održavanje i razvoj softvera kao dve nezavisne aktivnosti. Održavanje često se smatra aktivnošću nižeg ranga, te se ne poklanja pažnja održivosti sistema u toku faze razvoja sistema. Četvrti problem je lakše rešiv, jer se primenom odgovarajućih softverskih tehniki može se uspešno održavati struktura softvera u toku njegovog životnog veka, a i razumljivost sistema.

PRIMER FAZA ODRŽAVANJA SOFTVERA

Dat je primer identifikovanih faza održavanja softvera za rezervisanje stolova i naručivanje hrane putem interneta.

Proces korektnog održavanja podeljen je u 5 faza radi lakše organizacije poslova u toku održavanja.

Aktivnosti koje treba obaviti u toku procesa korektnog održavanja:

1. Identifikacija i analiza - u ovoj fazi vrši se identifikacija i analiza grešaka koje treba biti popravljen u procesu korektnog održavanja, kao i identifikacija i analiza novih zahteva koji treba da budu implementirani u procesu održavanja
2. Dizajn - u ovoj fazi vrši se projektovanje svih modula koji treba da budu promenjeni ili dodati u toku korektnog održavanja
3. Implementacija - u ovoj fazi popravljaju se uočene greške i dodaju se nove funkcionalnosti
4. Testiranje - nakon implementacije sistem se testira kako bi se osiguralo da izvršene promene nisu izazvale nove greške
5. Isporuka - klijentu se isporučuje nova verzija softvera sa ispravljenim greškama ili dodatim funkcionalnostima

U procesu identifikacije i analize treba napraviti dokument svih promena koje treba biti implementirane u fazi korektnog održavanja. Taj dokument treba da obuhvata sve greške koje su otkrivene nakon puštanja sistema u upotrebu, kao i zahteve za poboljšanje sistema.

Zahtevi za poboljšanje sistema obuhvataju sledeća proširanja:

- Web sajt ugostiteljskog objekta preko koga će gosti objekta moći da rezervišu sto u objektu
- Treba omogućiti da gosti prilikom rezervacije stola naprave i svoju porudžbinu

U procesu dizajna potrebno je modifikovati strukturu baze podataka kako bi odgovarala zahtevima za proširenja sistema. Takođe, ukoliko je potrebno treba napraviti promene u arhitekturi sistema kako bi omogućili implementaciju planiranih dodatnih funkcionalnosti sistema. Pored toga, u procesu dizajna treba isplanirati način uklanjanja identifikovanih grešaka

U fazi implementacije ispravljaju se pronađene, dokumentovane greške i dodaju se funkcionalnosti planirane proširenjem sistema.

U fazi isporuke klijent dobija novu veziju softvera koja obuhvata dogovoren proširenja i u kojoj su ispravljenje pronađene greške. Nakon isporučivanja druge verzije sistema prelazi se u fazu adaptivnog održavanja softvera.

ZADACI ZA SAMOSTALNI RAD

Dati su zadaci za samostalni rad koji obuhvataju održavanje softvera.

1. Zadatak: Predstaviti troškove održavanja softvera za rezervisanje karata u bioskopu. Navesti faze održavanja softvera i aktivnosti koje je potrebno izvršiti u svakoj fazi.

2. Zadatak: Simulirati povećanje troškova održavanja softvera za rezervisanje karata u bioskopu. Na osnovu čega dolazi do povećanja troškova održavanja?

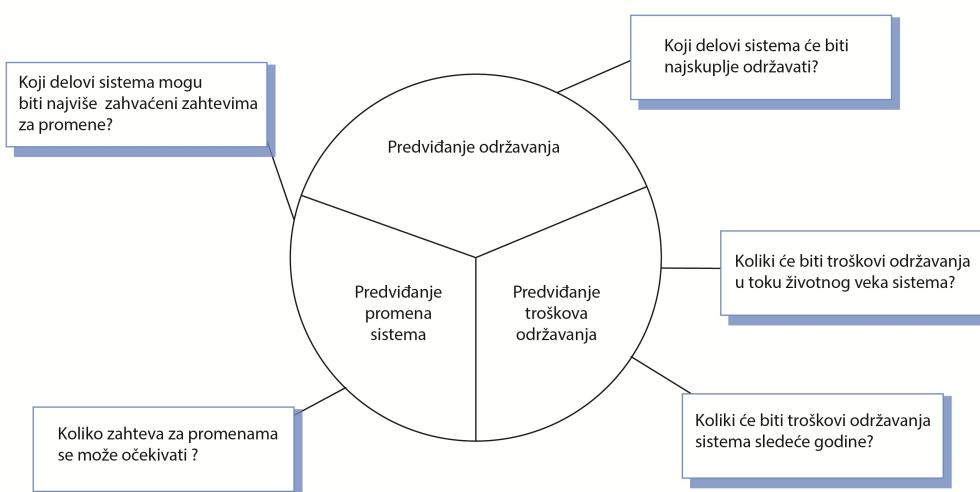
✓ 3.2 Predviđanje troškova održavanja softvera

PREDVIĐANJE TROŠKOVA ODRŽAVANJA

Poželjno je proceniti ukupne troškove održavanja sistema u datom vremenskom periodu

Poželjno je predvideti promene sistema, kao i delove sisteme koji će biti najteži za održavanje. Poželjno je proceniti ukupne troškove održavanja sistema u datom vremenskom periodu (slika 1).

Da bi se predvideo broj zahteva za promenama nekog sistema, potrebno je dobro razumeti odnos između sistema i spoljnog okruženja. Ukoliko su ti odnosi složeniji, treba očekivati veći broj zahteva za promenama.



Slika 3.2.1 Predviđanje održavanja

MERE ZA SMANJENJE TROŠKOVA ODRŽAVANJA

Da bi se smanjili troškovi održavanja, preporučljivo je da se složene komponente zamene sa jednostavnijim alternativnim rešenjima

Kako oceniti složenost odnosa između sistema i okruženja? Uzmite u obzir sledeće:

- Broj i složenost sistemskih interfejsa:** Što ih je više, i što su složeniji, veća je verovatnoća da će se menjati.
- Broj nasledno nestabilnih sistemskih zahteva:** Zahtevi koji odražavaju politiku organizacije i procedure, uvek se češće menjaju nego zahtevi koji se odnose na stabilne karakteristike u nekom području.
- Poslovni procesu u kojim se sistem koristi:** Kako se poslovni procesi razvijaju i menjaju, tako se generišu zahtevi za promenama. Što je veći broj poslovnih procesa koji ih upotrebljavaju, veći je broj zahteva za promenama.

Primenom metrike softvera, meri se njegova složenost. Na taj način se utvrđuju složenije komponente nekog sistema. Što su komponente složenije, to se teže održavaju. Da bi se smanjili troškovi održavanja, preporučljivo je da se složene komponente zamene sa jednostavnijim alternativnim rešenjima.

Kada se neki sistem pusti u rad, obradom nekih podataka, možete oceniti mogućnosti njegovog održavanja. Kakva je metrika procesa potrebna da bi se ocenila sposobnost održavanja sistema? Evo nekih primera:

- Broj zahteva za korektivno održavanje:** Ako je broj izveštaja o greškama veći od broja ispravljenih grešaka programa u toku procesa održavanja, onda je održavanje sistema lošije, tj. opada njegova sposobnost održavanja.
- Prosečno vreme potrebno za izradu analize uticaja promena:** Ovo odražava broj komponenata koje su pod uticajem neke promene. Ako vreme da se ovo utvrdi raste, to ukazuje da je sve veći broj komponenata zahvaćen promenama, te se smanjuje sposobnost održavanja sistema.
- Prosečno vreme za realizaciju zahteva za promenom:** Ako vam je potrebno više vremena da realizujete (primenite, implementirate) sistem i njegovu dokumentaciju, ukazuje da sposobnost održavanja sistema opada.
- Broj izuzetnih zahteva za promenama:** Ako se povećava ovih vanrednih zahteva, onda to ukazuje da se smanjuje sposobnost održavanja sistema.

Ove informacije obezbeđuju da se izvrše predviđanja sposobnost i održavanja sistema, i da se na osnovu toga, predvide troškovi održavanja. Naravno, intuicija i iskustvo su ovde neophodni za bolje predviđanje. Zato je vrlo važno dobro razumevanje postojećeg koda kao i znanje o razvoju novog koda.

PRIMER PREDVIĐANJA TROŠKOVA ODRŽAVANJA

Dat je primer predviđanja troškova održavanja sistema za podršku elektronskog bankarstva.

Estimacija troškova uglavnom se zasniva ne prethodnom iskustvu sa projektima sličnih obima i karakteristika. Estimacija troškova je jako bitna jer ona čini između 80% i 90% cene softvera u toku njegovog životnog ciklusa. Neki od faktora koji mogu doprineti smanjenju troškova održavanja su upotreba CASE alata kako bi se neki procesi automatizovali i pridržavanje standarda i unapred određenog plana bez previše izmena jer svako odstupanje od plana može doneti nove neplanirane troškove.

Primer estimacije biće dat na osnovu procene testiranja nekoliko funkcionalnosti koje treba dodati u budućnosti.

Naziv	O (h)	M (h)	P (h)	E (h)
Računanje kamate za štednju	4	7	12	7,3
Pregled svih kredita	2	4	8	4,3
Računanje rate kredita	3	6	10	6,1
Uzimanje kredita	6	10	16	10,3
Ukupno				28

Slika 3.2.2 Naziv funkcionalnosti koje treba da budu dodate u sistem za podršku elektronskog bankarstva

'3 point' estimacija koristi se kod upravljanja projektima i kod informacionih sistema za estimacije na osnovu iskustva ili na osnovu približne procene. Zasniva se na tri veličine:

O - optimistična estimacija

M - najverovatnija estimacija

P - pesimistična estimacija

Može se računati po dve formule:

$$E = (O + M + P) / 3$$

$$E = (O + 4M + P) / 6$$

ZADACI ZA SAMOSTALNI RAD

Dati su zadaci za samostalni rad koji obuhvataju predviđanje troškova održavanja softvera.

Zadatak 1: Predvideti troškove održavanja za jedan bibliotečki sistem. Navesti vreme po fazama izraženo u radnim satima. Definisati potrebne resurse po fazama.

Zadatak 2: Na osnovu prvog zadatka planirati i izvršiti mere za smanjenje troškova održavanja. Prikazati rezultate nakon smanjenja troškova održavanja.

✓ 3.3 Reinženjering softvera

ŠTA JE REINŽENJERING SOFTVERA?

Reinženjering softvera je promena strukture arhitekture sistema, ponovno pisanje koda primenom modernih programskih jezika, promena struktura i vrednosti podataka sistema.

Stari programi vremenom, zbog zastarele tehnologije, ili zbog menjanja programa tokom njihove evolucije, postaju spori i neefikasni. Pored toga, mnoge stare sisteme, teško je razumeti i menjati. Oni su optimizirani za performanse, ili za korišćenje memorijskog prostora, a na račun jasnoće koda. Vremenom, početna struktura se znatno promeni i uruši stalnim promenama softvera.

Da bi ovi stari sistemi postali lakši za održavanje, potrebno je izvršiti reinženjering sistema kako bi se poboljšala struktura sistema i njegova razumljivost. **Reinženjering softvera** je promena strukture arhitekture sistema, ponovno pisanje koda primenom modernih programskih jezika, promena struktura i vrednosti podataka sistema, kao i promena strukture i vrednosti sistemskih podataka::

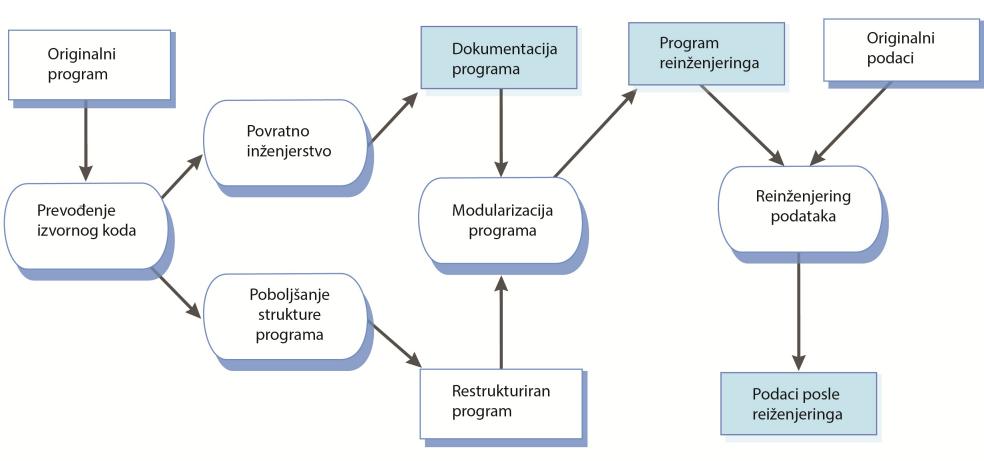
. Funkcionalnost softvera se ne menja, i treba izbeći da se vrše velike promene i arhitekturi sistema. Na taj način se mogu ostvarite dve koristi od reinženjering softvera umesto njegove zamene potpuno novim sistemom.

1. **Smanjiti rizik:** Razvoj i primena potpuno novog sistema, a naročito ako je on kritičan, je uvek rizični potez, jer se mogu javiti greške koje mogu da odlože njegovu primenu. Sa reiženjeringom starog softvera, taj rizik je niži.
2. **Smanjeni troškovi:** Troškovi reinženjeringa mogu biti višestruko niži nego troškovi razvoja novog sistema.

PROCES REINŽENJERINGA

Proces reeinženjeringa starog softvera sadrži niz aktivnosti koje poboljšavaju performanse starog softvera.

Slika 1 prikazuje opšti model procesa reinženjeringa softvera. Na ulazu je stari program a na izlazu je poboljšan i restrukturiran isti program.



Slika 3.3.1 Proces reinženjeringa starog programa

Proces reinženjeringa sadrži sledeće aktivnosti:

1. **Prevođenje izvornog koda:** Uz pomoć odgovarajućeg alata, vrši se konverzija sa nekog starog programskega jezika u noviju verziju istog jezika ili u neki potpuno drugi i savremeniji programski jezik.
2. **Povratno inženjerstvo:** Analizom izvornog koda radi se programska dokumentacija, tj. opisuje se njegova organizacija i funkcionalnost. Često se i ova aktivnost može da automatizuje.
3. **Poboljšanje strukture programa:** Analizira se struktura programa i vrši se njena promena da bi se program lakše čitao i razumeo. Ovaj proces se može samo delimično automatizovati.
4. **Modularizacija programa:** Grupišu se međusobno povezani delovi programa i eliminišu se, redundantnosti u sistemu (ponavljanja koda). U nekim slučajevima može se modifikovati i struktura programa. Na primer, umesto nekoliko, koristi se samo jedna baza podataka. Ovo je ručni proces.
5. **Reinženjering podataka:** U skladu sa promenama u programu, menjaju se i podaci. To obuhvata i promenu šeme baze podataka i konverziju postojećih baza podataka u nove strukture. Tada se i podaci „čiste“, tj. uklanjuju se pogrešni podaci, dupli slogovi podataka i dr. Postoje alati za podršku reinženjeringa podataka.

TROŠKOVI I PROBLEMI REINŽENJERINGA

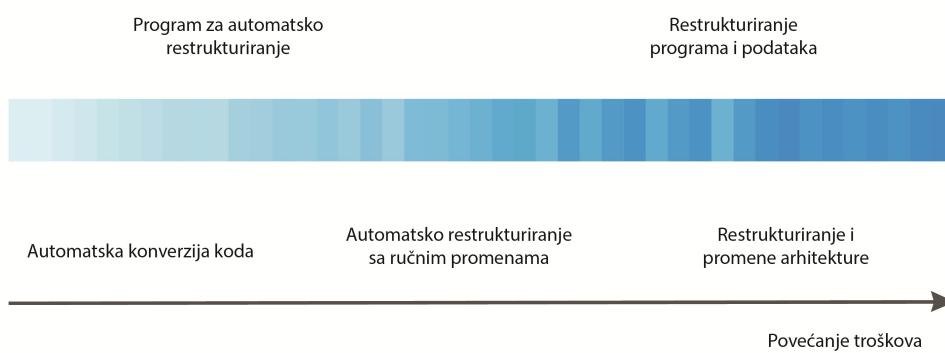
Program posle reinženjeringa ipak nije tako jednostavan za održavanje kao što je obično potpuno novi sistem koji koristi savremene metode softverskog inženjerstva.

U praksi, reinženjering programa ne mora da obuhvati sve navedene aktivnosti. Na primer, ne morate da prevodite izvorni program u novu verziju jezika ili u novi jezik. Takođe, nije vam potrebna programska dokumentacija ako možete da reinženjering sprovedete automatski.

Reinženjering podataka se vrši samo ako se promenama u programu menja struktura podataka.

Kako preći sa starog programa na promjenjen program posle reinženjeringu? Za to se često koriste adaptori koji sakrivaju originalne interfejse softverskog sistema i prikazuju nove, koji su bolje strukturisani i koji se mogu koristiti od strane drugih komponenti. Vi ste time upakovali stari softver „u novo odelo“, tj prekrili ste ga novim interfejsima, i time dali druge servisne mogućnosti sistema.

Zavisno od opsega preduzetih aktivnosti reinženjeringu, zavisi i visina troškova koji ga prate (slika 2). Najskuplji deo je promena arhitekture sistema



Slika 3.3.2 Troškovi reinženjeringu u zavisnosti od primenjenih pristupa u reinženjeringu

Šta su problemi sa reinženjeringom softvera? Postoje praktične granice poboljšanja starog programa reinženjeringom. Na primer, funkcionalno pisan program ne možete pretvoriti u objektno-orientisan program. Radikalna reorganizacija upravljanja podacima se ne može automatizovano sprovesti. Program posle reinženjeringu ipak nije tako jednostavan za održavanje kao što je obično potpuno novi sistem koji koristi savremene metode softverskog inženjerstva.

PRIMER PROCESA REINŽENJERINGA SOFTVERA

Dat je primer procesa reinženjeringu softvera.

<pre> Start: Get (Time-on, Time-off, Time, Setting, Temp, Switch) if Switch = off goto off if Switch = on goto on goto Cntrd off: if Heating-status = on goto Sw-off goto loop on: if Heating-status = off goto Sw-on goto loop Cntrd: if Time = Time-on goto on if Time = Time-off goto off if Time < Time-on goto Start if Time > Time-off goto Start if Temp > Setting then goto off if Temp < Setting then goto on Sw-off: Heating-status := off goto Switch Sw-on: Heating-status := on Switch: Switch-heating loop: goto Start </pre>	<pre> loop - The Get statement finds values for the given variables from the system's - environment. Get (Time-on, Time-off, Time, Setting, Temp, Switch); case Switch of when On => if Heating-status = off then Switch-heating ; Heating-status := on ; end if; when Off => if Heating-status = on then Switch-heating ; Heating-status := off ; end if; when Controlled => if Time >= Time-on and Time <= Time-off then if Temp > Setting and Heating-status = on then Switch-heating; Heating-status = off; elseif Temp < Setting and Heating-status = off then Switch-heating; Heating-status := on ; end if; end if; end case; end loop; </pre>
---	--

Slika 3.3.3 Primer dela programskog koda softvera na kome je izvršen proces reinženjeringa

Na slici 3 dat je primer reinženjeringa programskog koda gde je na levoj strani prikazan programski kod pre reinženjeringa a na desnoj strani slike nalazi se struktuiran programski kod nakon reinženjeringa. Metode su jasno vidljive i petlje su naglašene.

ZADACI ZA SAMOSTALNI RAD

Dati su zadaci koji se odnose na proces reinženjeringa softvera.

1. Zadatak: Kreirati program koji računa površinu i zapreminu pravougaonika. U jednom delu programa primeniti petlju if a u slučaju da stranice prelaze određeni broj program treba da izračuna zapreminu (koristeći petlju else). Nakon kreiranja programa primeniti proces reinženjeringa i prikazati dobijene rezultate.

2. Zadatak: Na osnovu prvog zadatka definisati troškove i probleme procesa reinženjeringa za program koji računa površinu i zapreminu pravougaonika.

3.4 Restrukturiranje softvera

PREVENTIVNO ODRŽAVANJA RESTRUKTURIRANJEM SOFTVERA

Restrukturiranje je poboljšanje strukture programa, smanjivanje njegove složenosti i olakšavanje njegovog razumevanja.

Restrukturiranje jer proces poboljšanja programa s ciljem da se uspori njegova degradacija (opadanje performansi, povećanje složenosti, otežano snalaženje radi uvođenja promena i dr.) usled promena kojim je vremenom izložen. Restrukturiranje je poboljšanje strukture programa, smanjivanje njegove složenosti i olakšavanje njegovog razumevanja. Restrukturiranje ne dodaje novu funkcionalnost programu, već se sam program poboljšava.

U tom smislu, restrukturiranje je jedan od oblika „preventivnog održavanja“ softvera je smanjuje probleme koji se javljaju usled promena softvera u budućnosti.

Koja je onda razlika restrukturiranja i reinženjeringa softvera?

1. **Reinženjering** se vrši posle izvesnog vremena održavanja softvera i povećanja troškova održavanja. Upotreboom automatskih alata za obradu i reinženjering starih sistema, vi kreirate novi sistem koji se lakše održava.
2. **Restrukturiranje koda** (refactoring code) je stalni proces poboljšanja za vreme razvoja i evolucije softvera. Sa njim mi izbegavamo degradaciju strukture i koda koji dovode do povećavanja troškova i do drugih teškoća održavanja sistema.

Restrukturiranje je sastavni deo agilnih metoda, kao što je ekstremno programiranje, jer se ovi metodi baziraju na promeni. Zbog čestih (malih) promena, program, teži degradaciji, te programeri često restrukturišu svoje programe kako bi izbegli degradaciju. Zbog primene regresivnog testiranja pri primeni agilnih metoda, smanjuje se rizik unošenja novih grešaka preko restrukturiranja. Sve unete greške bi trebalo da budu detektovane (otkrivene) primenom testova. Međutim, restrukturiranje nije zavisno od drugih aktivnosti primene agilnih metoda, te se može primeniti u bilo kom pristupu razvoja.

PRIMENA RESTRUKTURIRANJA SOFTVERA

Poboljšanje softvera se vrši restrukturiranjem programskog koda, ali i projektnog rešenja, što je skuplji i složeniji metod poboljšanja, ali se primenjuje kada restrukturiranje koda nije dovoljno.

Kada primeniti restrukturiranje? Odgovor: kada se pojavi neka od sledećih situacija:

1. **Dupliranje koda:** Uočavate isti ili vrlo sličan kod na različitim mestima u programu. Taj kod onda zamenjujete jednim metodom kojega onda pozivate na tim mestima.
2. **Dugački metodi:** Vidite metod koji je vrlo dugačak (puno linija koda). Odda ga zamenjujete se više manjih metoda.
3. **Iskazi SWICH:** U programu ima na dosae mesta iskaz „switch“. Kod objektno-orientisanih sistema se oni mogu zameniti upotreboom polimorfizma
4. **Cirkulacija podataka:** Cirkulacija podataka se javlja kada se ista grupa podataka (polja u klasama, parametri u metodima) javljaju na više mesta u programu. To se otklanja njihovim učaurenjem u okviru jednog objekta.
5. **Spekulativna uopštenost:** Do ovoga dolazi kada programeri uključuju opštene iskaze u program, jer očekuju da će biti potrebna u budućnosti. To se jednostavno, može ukloniti.

Primenom strukturnih transformacija mogu se navedeni problemi rešavati. Na primer, metod ekstrakcije uklanja dupliranja i kreira novi metod. Preuređenjem iskaza uslovljavanja u programu, više testova vi zamenjujete jednim testom. Možete zameniti slične metode jednim metodom u njihovoj super klasi. Okruženja za interaktivni razvoj, kao što je Eclipse, uključuje restrukturiranje u svoje editore. To olakšava nalaženje delova programa koji se treba da promene restrukturiranjem.

Restrukturiranje projektnog rešenja je skuplji i složeniji metod poboljšanja programa. On se primenjuje radi utvrđivanja odgovarajućih šablona projektnih rešenja. Ova vrsta restrukturiranja se primenjuje kada samo restrukturiranje koda nije dovoljno. Program je isuviše degradiran da se na taj način on dovoljno poboljša. Restrukturiranjem projektnog rešenja onda može da dovede i do promene projektnog rešenja (engl. design) softvera.

PRIMER RESTRUKTURIRANJA SOFTVERA

Dat je primer restrukturiranja softvera.

Na slici 1. se može videti aplikacija koja je napravljena za potrebe restorana i u kojoj se svaka klasa koju ova aplikacija koristi, nalaze u okviru jednog paketa što je loše za performanse same aplikacije. Takođe ukoliko se radi na nekoj od izmena, potrebno je dodatno vreme koje će se potrošiti na pronalaženje dela koda koji bi trebalo ispraviti ili nadograditi. Zbog trošenja vremena na pronalaženje potrebnog koda, vrši se restrukturiranje same aplikacije. Da bi se restrukturiranje obavilo uspešno, potrebno je podeliti aplikaciju na odvojene pakete koji daju informaciju programeru za šta je klasa napravljena u zavisnosti od imena paketa. Na slici 2. se može videti na koji način je sama aplikacija ubrzana i restrukturirana, pa sada umesto jednog paketa aplikacija ima 6 paketa i u svaki od njih je ubaćena klasa koja je povezana sa samim naslovom paketa. U GUI paket se smeštaju klase koje su povezane sa samim korisničkim interfejsom, u paketu Exceptions se nalaze klase povezane sa izuzecima koji se mogu javiti prilikom korišćenja aplikacije, u okviru business.baza paketa se nalaze klase koje su potrebne za povezivanje sa bazom podataka koju aplikacija koristi.

The screenshot shows the Eclipse IDE interface. On the left, the 'Projects' view displays two projects: 'Kafic' and 'Kafic-Restrukturiran'. The 'Kafic' project contains several source packages: 'projekat', 'Libraries', and 'gui'. The 'Libraries' package contains a 'data' package which includes 'Artikal.java' and 'Racun.java'. The 'gui' package contains 'Finansije.java', 'GlavnihMeni.java', 'Login.java', 'Racun.java', 'Sto.java', 'UnosArtikla.java', and 'UnosKategorija.java'. The 'main' package contains 'Main.java'. The 'Kafic-Restrukturiran' project also has a 'Libraries' package containing a 'business' package with 'Baznis.java' and 'Baznis.baza' sub-packages, and a 'data' package with 'Artikal.java' and 'Racun.java'. The 'exceptions' package contains various exception classes like 'ArtikalException.java', 'ArtikalNijeNadjenException.java', etc. On the right, the code editor window is open for 'Artikal.java' in the 'data' package. The code defines a class 'Artikal' with fields 'id', 'kategorijaId', 'cena', 'stanje', and 'naziv'. It includes a constructor that initializes these fields from parameters, and a helper constructor that initializes them from a single string parameter. The code editor also shows JavaDoc comments and imports at the top.

```

/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package data;

/**
 * Klasa koja predstavlja Artikal
 * @author nebojsa
 */
public class Artikal {
    private int id, kategorijaId, cena, stanje;
    private String naziv;

    /**
     * Prazan konstruktor
     */
    public Artikal() {
    }

    /**
     * Konstruktor za Artikal
     */
    public Artikal(int id, int kategorijaId, int cena, int stanje) {
        this.id = id;
        this.kategorijaId = kategorijaId;
        this.cena = cena;
        this.stanje = stanje;
        this.naziv = naziv;
    }

    /**
     * Pomočni konstruktor koji prima tri podatka
     */
    public Artikal(int id, String naziv, int stanje) {
        this.id = id;
        this.naziv = naziv;
        this.stanje = stanje;
    }
}

```

Slika 3.4.1 Primer restrukturiranja softvera

ZADACI ZA SAMOSTALNI RAD

Dati su zadaci koji se odnose na restrukturiranje koda.

- 1. Zadatak:** Napraviti klasu koja simulira konverziju valuta iz dolara u evro. Restrukturirati klasu i prikazati smanjenje metoda koje vrše konverziju valuta.
- 2. Zadatak:** Napraviti klasu i primeniti iskaz "switch". Zatim primenom polimorfizma zameniti iskaz "switch" i uprostiti programski kod. Prikazati dobijene rezultate.

▼ Poglavlje 4

Upravljanje starim softverom

ŠTA RADITI SA STARIM SOFTVEROM?

Da bi se izabrala pravilna strategija evolucije starog sistema, potrebno je najpre izvršiti njegovu procenu i sa poslovnog i sa tehničkog stanovišta

Organizacije razvijaju nove softverske sisteme, povećavaju pokrivenost svog poslovanja novim aplikacijama. Sve više shvataju važnost upravljanja životnim vekom ovih sistema i potrebu integracije faze razvoja i evolucije. Međutim, skoro svaka organizacija ima isti problem: Šta raditi sa starim sistemima? Oni i dalje i zadovoljavaju izvesne funkcije, na više ili manje uspešan način. Najčešće su tehnološki prevaziđeni i teško se integrišu sa novim sistemima. Koju strategiju evolucije izabrati za stare sisteme? Postoje četiri mogućnosti:

1. **Potpuno uništiti sistem:** Ovo treba primeniti kada sistem nema više efekta na poslovne procese u organizaciji. Poslovni procesi su se promenili i više ne zavise od ovog sistema.
2. **Ostaviti sistem kakav jeste i nastaviti njegovo regularno održavanje:** Primeniti u slučaju kada je sistem i dalje potreban, dosta stabilan, a malo ima zahteva za njegovu promenu.
3. **Izvršiti reinženjering sistema radi poboljšanja njegove sposobnosti održavanja:** Ovo se primenjuje kada je kvalitet sistema opao zbog izvršenih promena na sistemu i kada se i dalje predlažu nove promene (npr. razvoj novih interfejsa), tako da stari i novi sistem mogu da rade paralelno.

4. Zameniti sve i neke delove sistema sa novim sistemom: Ovu opciju treba izabrati kada zbog promene hardvera sistem ne može više da radi, ili kada se pojave novi softverski proizvodi koji se mogu relativno jednostavno i sa prihvatljivim troškovima da upotrebe za razvoj novog sistema. U mnogim slučajevima se primenjuje evolutivna zamena glavnih komponenata sa novim (nabavljenim) softverskim proizvodima, dok se ostale komponente nastavljaju da koriste

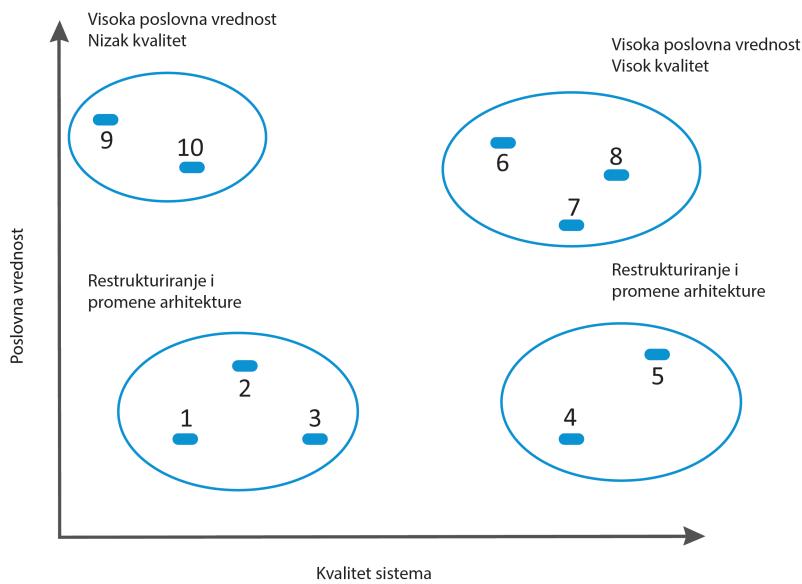
Kod složenijih sistema, može se koristiti više od jedne od navedenih opcija, jer za svaki podsistem se može koristiti i različita strategija.

Da bi se izabrala pravilna strategija evolucije starog sistema, potrebno je najpre izvršiti njegovu procenu i sa poslovnog i sa tehničkog stanovišta. Sa poslovnog stanovišta – znači da treba da ocenite da li vam je taj sistem uopšte više potreban za posao koji sada radite. Sa tehničkog stanovišta – treba da ocenite kvalitet softvera i softvera i hardvera koji ga podržavaju. Na osnovu ovakve analize zaključujete šta je najbolje da uradite sa starim sistemom

ČETIRI KLASTERA STARIH SISTEMA

Pre donošenja odluke o sudbini starih (zatečenih) sistema, vrši se analiza njihove poslovne vrednosti i njihovog kvaliteta, te se grupišu u četiri klastera.

Ako imate više starih sistema, potrebno je za svaki od njih da napravite analizu i da odredite poslovnu vrednost sistema. Dobijene rezultate možete prikazati na dijagramu koji pokazuje poslovnu vrednost u odnosu na kvalitet sistema (slika 1).



Slika 4.1 Primer ocenjivanja starih sistema

Kao što se može videti na slici, postoji četiri klastera ovih sistema:

1. **Niski kvalitet, niska poslovna vrednost:** Ove sisteme treba uništiti (napustiti) jer skup rad ovih sistema a poslovna vrednost vrlo mala.
2. **Niski kvalitet, visoka poslovna vrednost:** Ove sisteme ne možete odbaciti jer imaju veliki značaj za poslovanje. Zbog velikih troškova održavanja (niski kvalitet) ovi sistem se treba zameniti novim, i to, ako je moguće, novim gotovim („sa polica“ ili engl. „off-the-shelf“) softverskim proizvodima.
3. **Visok kvalitet, niska poslovna vrednost:** Ovi sistemi ne doprinose mnogo poslovanju, ali nije skupo njihovo održavanje (visok kvalitet). Zbog toga, ne isplati se njihova zamena i treba nastaviti njihovo korišćenje, sem u slučaju da ima zahteva za njihovu promenu koji zahtevaju visoka ulaganja ili novi hardver, kada je nužno njihovo odbacivanje.
4. **Visok kvalitet, visoka poslovna vrednost:** Ove sisteme treba zadržati i nastaviti njihovo normalno održavanje.

UTVRĐIVANJE POSLOVNE VREDNOSTI SOFTVERSKOG SISTEMA

Poslovna vrednost softvera u nekoj organizaciji se određuje analizom frekvencije njegove upotrebe, brojem poslovnih procesa koje podržava, stepenom pouzdanosti i važnosti rezultata.

Kako utvrditi poslovnu vrednost nekog softverskog sistema? Potrebno je prvo da identifikujete zainteresovane aktere tih sistema i onda da im postavite niz pitanja o sistemu. U tim razgovorima, možete diskutujete o sledećim temama:

1. **Upotreba sistema:** Ako se sistemi samo povremeno koriste, ili ih koristi mali broj ljudi, onda oni imaju nisku poslovnu vrednost. To se često dešava ako je prvobitna uloga sistema prevaziđena promenama u poslovanju ili se ona sada realizuje na neki drugi, efektniji način. Međutim, ako je i povremeno korišćenje vrlo važno za organizaciju, onda se takav sistem ne odbacuje.
2. **Poslovni procesi koje sistem podržava:** Svaki sistem podržava jedan ili više poslovnih procesa. Ako je sistem nefleksibilan, i ovi procesi se ne mogu da menjaju. Međutim, zbog promene okruženja i poslovnih zahteva, vrlo često se poslovni procesi moraju da menjaju. Sistem koji ne može da podrži takve promene onda treba izbaciti iz upotrebe.
3. **Pouzdanost sistema:** Pored tehničke, postoji i poslovna pouzdanost. Ako sistem nije pouzdan, te direktno time ometa poslovne partnere, ili zahteva ulaganje resursa u rešavanje problema pouzdanosti, onda sistem ima nisku poslovnu vrednost.
4. **Rezultati sistema:** Treba utvrditi važnost rezultata koje proizvodi sistem na uspešno funkcionisanje poslovanja. Ako posao zavisi od ovih rezultata, onda sistem ima visoku poslovnu vrednost. S druge strane, ako se rezultati sistema retko koriste, ili se mogu dobiti i na drugi način, onda je poslovna vrednost sistema niska.

OCENJIVANJE OKRUŽENJA U KOME RADI SOFTVER

Da bi ocenili okruženje, potrebno je da izvršite neka merenja sistema, njegovog okruženja i njegovog procesa održavanja

Da bi ocenili softverski sistem s tehničkog stanovišta, treba da uzmete u obzir i aplikacioni sistem i okruženje (hardver, softver) u kom on radi. Mnogi sistemi se moraju menjati jer se okruženje promenilo.

Da bi **ocenili okruženje**, potrebno je da izvršite neka merenja sistema i njegovog okruženja i njegovog procesa održavanja. Koristite podatke, kao što su troškovi održavanja hardvera i softvera za podršku, broj otkaza hardvera u nekom periodu i frekvenciju izvršenih popravki softvera za podršku. Slika 2 prikazuje faktore koje bi trebalo da uzmete u obzir prilikom ocenjivanja okruženja u kom stari sistem radi. To nisu tehničke karakteristike okruženja. Morate takođe uzeti u obzir i pouzdanost proizvođača opreme. Ako oni više nisu u poslu, onda je moguće da dalja podrška opreme nije moguća.

Faktor	Pitanja
Stabilnost proizvođača	Da li je proizvođač još postoji? Da li je finansijski stabilan i da li je verovatno da će nastaviti svoj posao? Ako proizvođač nije više u poslu, da li još neko održava njegove sisteme?
Brzina otkaza	Da li hardver ima visoku učestanost prijavljenih otkaza? Da li softver za podršku pada i prouzrokuje da i sistem pada?
Starost	Koliko je star hardver i softver? Što su hardver i softver za podršku stariji, to je manje upotrebljiv. Oni i dalje funkcioniše korektno ali su moguće značajne ekonomski i poslovne koristi od prihvatanja modernijeg sistema.
Performanse	Da li su performanse sistema odgovarajuće? Da li problemi sa performansama imaju značajan uticaj na korisnike sistema?
Zahtevi za podrškom	Koja je lokalna podrška potrebna za hardver i softver? Ako tu podršku prate visoki troškovi, onda treba razmotriti mogućnost zamene sistema.
Troškovi održavanja	Koliki su troškovi održavanja hardvera i licenci softvera za podršku? Stariji hardver može da ima visoke troškove održavanja nego moderni sistemi. Softver za podršku može imati visoke troškove godišnjih licenci.
Interoperabilnost	Da li postoje problemi sprezanja sistema sa drugim sistemima? Da li kompjajleri, na primer, mogu da koriste sadašnju verziju operativnog sistema? Da li je potrebna emulacija hardvera?

OCENJIVANJE KVALITETA APLIKACIJE

Najvažniji faktori za ocenjivanje kvaliteta aplikacije su vezani za pouzdanost sistema, poteškoće u održavanje sistema i dokumentaciju.

Da bi ocenili kvalitet aplikacije (starog softverskog sistema) treba da ocenite niz faktora (slika 3) koji su prvenstveno vezani za pouzdanost sistema, poteškoće u održavanje sistema i dokumentaciju.

Faktor	Pitanja
Razumljivost	Koliko je teško da se razume izvorni kod sadašnjeg sistema= Koliko je složena njegova upravljačka struktura? Da li promenljive imaju smisalne nazive koji odražavaju njihovu funkciju?
Dokumentacija	Koja je dokumentacija sistema dostupna? Da li je dokumentacija kompletna, konsistentna i raspoloživa?
Podaci	Da li postoji eksplizitni model podataka sistema? Do kog stepena se duplikiraju podaci po datotekama? Da li su podaci koje koristi sistem tačni i konsistentni?
Performanse	Da li su performanse aplikacije odgovarajuće? Da li problemi sa performansama imaju značajan uticaj na korisnike sistema?
Programski jezik	Da li su raspoloživi moderni kompjajleri za programski jezik koji se koristio u razvoju sistema? Da li postoji eksplizitni opis verzije komponenti koje koristi sadašnji sistem?
Upravljanje konfiguracijom	Da li se sve verzije svih delova sistema kontrolišu uz pomoć sistema za upravljanje konfiguracijom sistema? Da li postoji eksplizitni opis verzija komponenti koje upotrebljava sadašnji sistem?
Podaci za testiranje	Da li postoje podaci testiranja sadašnjeg sistema? Da li postoji zapis urađenih regresivnih testova kada su nova svojstva bila dodavana sistemu?
Lične veštine	Da li postoje ljudi koji imaju veštine neophodne za održavanje aplikacije? Da li postoji eljido koji imaju iskustvo rada sa sistemom?

Da bi izvršili ocenu rada dosadašnjeg sistema (aplikacije) potrebni su vam sledeći podaci:

- Broj zahteva za promenu sistema:** Promene sistema pogoršavaju strukturu sistema i otežavaju dalje promene. Što je veći broj akumuliranih promena, to je niži kvalitet sistema.
- Broj korisničkih interfejsa:** Kod sistema koji se baziraju na formama koje korisnici koriste, značajan faktor je njihov broj. Što ih je više, veća je verovatnost nekonsistentnosti i ponovljivosti kod ovih interfejsa u budućnosti, a prilikom daljih promena sistema.
- Količina podataka koje sistem koristi:** Što je veća količina podataka (broj datoteka, veličina baza podataka itd.), veća je verovatnoća da će doći do pojave nekonsistenih podataka koji smanjuju kvalitet softvera.

DRUGI FAKTORI ODLUČIVANJA

Često se odluke donose i na osnovu manje objektivnih ocena, jer se zasnivaju na organizacionim ili političkim faktorima

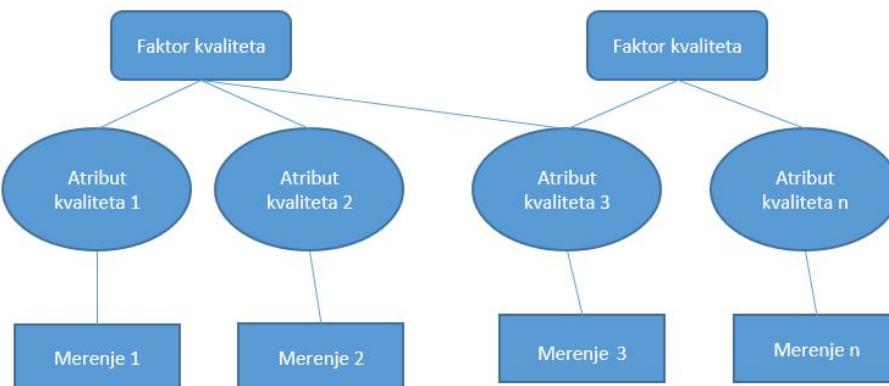
Posle ocene kvaliteta samog softvera, tj. kvaliteta aplikacije, kao i uzimanja u obzir i faktora okruženja u kome softver radi, donosi se ocena softverski sistem s tehničkog stanovišta. Konačna odluka o zadržavanju ili menjanja postojećeg softverskog sistema zavisi od ocene njegove poslovne vrednosti i njegovog kvaliteta.

U idealnom slučaju, pri donošenu odluke o sudbini postojećih sistema, trebalo bi se služiti isključivo objektivnim ocenama svih navedenih faktora. Međutim, često se odluke donose i na osnovu manje objektivnih ocena, jer se zasnivaju na organizacionim ili političkim faktorima. Navešćemo nekoliko primera:

- Pri spajanju dve kompanije, najčešće se zadržavaju sistemi jačeg partnera, a napuštaju drugi sistemi.
- Ako organizacija odluči da pređe na novu hardversku platformu, onda to zahteva promenu aplikacija.
- Ako nema potrebnog budžeta za transformaciju sistema, onda se nastavlja sa održavanjem starog sistema, iako to dovodi do, dugoročno gledano, do povećanih troškova.

PRIMER OCENJIVANJA KVALITETA APLIKACIJE

Dat je primer ocenjivanja kvaliteta aplikacije shodno određenim parametrima.



Slika 4.2 Šema ocenjivanja kvaliteta aplikacije

Na početku ocenjivanja kvaliteta aplikacije definišu se određeni faktori kvaliteta. Faktori kvaliteta se sastoje od niza atributa kvaliteta. Atribute kvaliteta definiše korisnik. Jedan faktor kvaliteta može imati više atributa. Za svaki atribut kvaliteta definisano je merenje koje utvrđuje da li je atribut kvaliteta ispunio određene zahteve kvaliteta aplikacije. Ukoliko su svi atributi u okviru faktora kvaliteta zadovoljili zahteve za kvalitet faktor kvaliteta je ocenjen pozitivno.

Na primeru sistema za upravljanje insulin pumpom faktori mogu biti:

- tačnost (sa atributima pouzdanost, dostupnost)
- održivost (sa atributima isplativost, primenljivost)
- efikasnost (sa atributima vreme odziva, operativnost)
- prenosivost (sa atributima softverska nezavisnost, dostupnost programskog jezika)

Za svaki od navedenih atributa faktora potrebno je definisani način merenja i opseg u kome je moguće dobiti rezultate kvaliteta. Merenja mogu biti organizovana kroz različite test slučajeve.

ZADACI ZA SAMOSTALNI RAD

Dati su zadaci koji se odnose na ocenjivanje kvaliteta aplikacije.

1. Zadatak: Sistem za prodaju PC komponenti. Sistem je namenjen firmama koje žele da naveliko kupuju PC komponente za svoje radnje. Napravljen je tako da olakša proces naručivanja i isporuke svih porudžbina. Podaci i informacije o svim poručivanjima čuvaće se u bazi podataka i time smanjiti broj potrebne dokumentacije za sve narudžbine.

Glavne komponente od kojih će sistem da se sastoji su komponenta za naručivanje i isporuku PC delova, komponenta za upravljanje proizvodima, komponenta za informisanje klijenata i komponenta za upravljanje naloga.

Izvršiti ocenjivanje kvaliteta sistema, definisati faktore kvaliteta i njihove atrbute kao i način merenja definisanih atrbuta.

2. Zadatak: Chatspace sistem simulira osnovne funkcionalnosti jedne chat aplikacije. Sistem omogućava korisniku da pristupi sistemu sa svojim nalogom na osnovu koga dobija sve privilegije korisnika sistema. Korisnik dobija mogućnost da razmenjuje poruke sa drugim korisnicima sistema. Poruke mogu biti u okviru teksta, priče, pesme, i ostalih tekstualnih objava. Korisnik ima mogućnost da poruke šalje drugim korisnicima. Takođe, korisnik ima mogućnost da čita poruke drugih korisnika kao i uvid u listu ostalih korisnika koji koriste sistem.

Izvršiti ocenjivanje kvaliteta sistema, definisati faktore kvaliteta i njihove attribute kao i način merenja definisanih atributa.

▼ Poglavlje 5

Radionica - Jenkins alat za stalnu integraciju SW

OPIS VEŽBE

Uvod i rad sa CI continuous integration korišćenjem Jenkins alata

Osnovni cilj ove vežbe jeste da demonstrira i uputi studente u Jenkins alat. Kako ga konfigurisati i podesiti i koristiti u realnom okruženju. Ova vežba neće pokriti sve detalje koje se odnose na Jenkins ali će prikazati veliku sliku, način rada i upotrebe Jenkins kao (Continuous integration) alata. Pored toga, kao osnovni primer moćiće da uradite build projekta uz pomoć Jenkins alata.

JENKINS

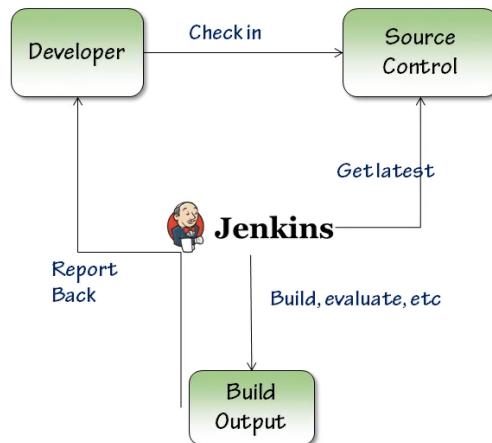
Jenkins je softver namenjen CI (continuous integration) koji se pokreće na serveru, pomaže u build procesu i aktivnostima koji se odnose u najvećoj meri na build

Jenkins Vam može omogućiti automatizaciju većine poslova kako bi dobili CI u okviru nekog projekta. Jenkins je otvorenog koda i dostupan je za besplatnu upotrebu. Jenkins u suštini predstavlja Web aplikaciju baziranu na Java tehnologijama. Bez obzira što je bazirana na Java tehnologijama, Jenkins uporedo dobro radi i sa drugim programskim jezicima (npr c#). Jenkins je dosta pogodan takođe iz razloga što poseduje veliki broj dodataka koji se mogu ugraditi u njega za specifične primene. Tako je moguće preuzeti dodatak za Build Andorid ili IOS aplikacije. Takođe postoji mogućnost kreiranja sopstvenog dodatka koji radi specifičnu stvar koja je samo Vama potrebna.

Jenkins ima mogućnost nadgledanja repozitorijuma sa kodom (Source control) kao što su SVN, GIT i drugi. Nakon toga, Jenkins se može podesiti tako da ima uvid u određenom vremenskom periodu, svaki put kada programer pošalje novu verziju koda ili na razne druge događaje u zavisnosti od potrebe. Nakon toga na Jenkinsu je da uradi šta mu je definisano. Najčešće je to puštanje build-a aplikacije koja je uzeta sa repozitorijuma, startovati testove i drugo. Nakon završavanja određenih testova dolazi do rezultata koji se šalju najčešće programerima ili određenoj osobi koja je zadužena za kontrolu nad kodom. Rezultat može biti da je sve uspešno završeno, ali i informacija da neki test nije prošao uspešno, došlo je do greške u build-u i slično. Ove informacije pomažu u otkrivanju grešaka i sigurnošću da neće biti puštena test verzija koja nije prošla proveru. Jenkins je tu da uradi automatizovanu proveru. Na slici 2 je prikazan proces rada Jenkins softvera.



Slika 5.1 Logo Jenkins softvera



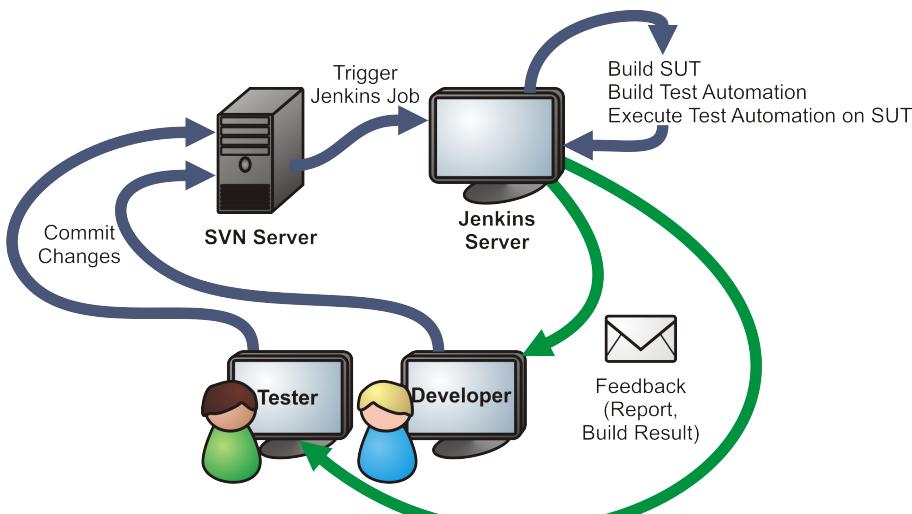
Slika 5.2 Proces rada Jenkins softvera

CONTINUOUS INTEGRATION (KONTINUALNA INTEGRACIJA KODA)

Šta predstavlja pojam Continuous Integration – stalno povezivanje koda

Continuous Integration (CI) ili u slobodnom prevodu (stalno povezivanje koda) odnosi se na potrebu za učestalim povezivanjem s kodom na kom rade ostali članovi tima. Ukoliko na istim delovima koda radi više ljudi, što duže traje razvoj (bez međusobne sinhronizacije), to su veće razlike i teže je povezati kod u jednu smislenu celinu kada napokon za to dođe vreme. Takođe, veća je šansa da nastanu oku nevidljivi bugovi. Iz ove prakse proizlazi čitav niz metoda bez kojih je danas praktično nezamisliv razvoj kvalitetnog softvera:

1. Održavanje repozitorijuma koda koji služi za čuvanje i sinhronizaciju koda u timu. Alati poput CVS, SVN ili Git danas su neizostavni i na najmanjim projektima.
2. Automatizacija kreiranja aplikacijskih artefakata – često može uključivati i instalaciju aplikacije na testnu okolinu. Za to se tipično koriste dobro poznati Ant, Maven, IBM Rational Build Forge i sl.
3. Učestalo stavljanje koda na repozitorijum. Testiranje koda trebalo bi da se izvršava na okolini koja je vrlo slična produkcijskoj.



Slika 5.3 Primer CI

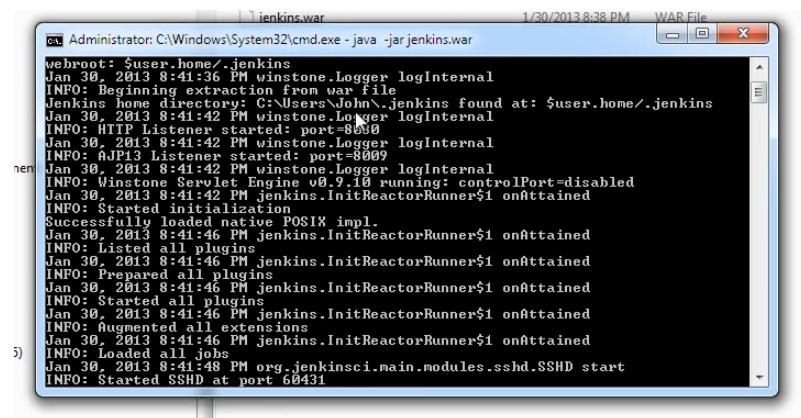
JENKINS INSTALACIJA

Preduslovi za instalaciju i proces instalacije Jenkins softvera

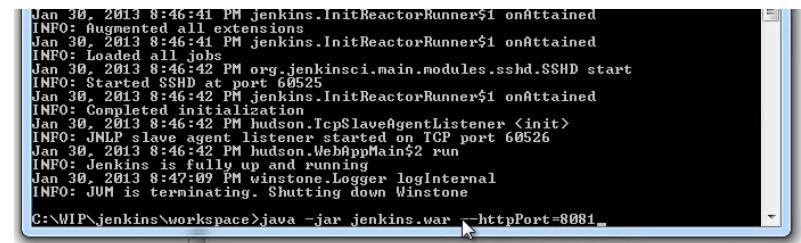
- Nepohodno je instalirati Java 1.5 verziju ili noviju, zbog toga što je Jenkins razvijan na Java platformi.
- Poželjno je konfigurisati Java Path (Ukoliko se Jenkins pokreće iz konzole kako bi se lakše pozvala java)
- Web server (samo u slučaju ukoliko zelite da hostuje aplikaciju izvan Jenkins softvera)

Nakon ispunjavanja svih preduslova moguće je preuzeti Jenkins sa zvaničnog sajta (<https://jenkins.io/download/>) se može instalirati na dva načina. Jedan način je u okviru .war extenzije kao standardne web aplikacije koju je moguće postaviti na neki od aplikacionih servera (glassfish, tomcat) i startovati. Takođe Jenkins ima "ugrađen" server tako da je ne zavisan od drugog aplikacionog servera. Ukoliko bi uneli komande (java -jar jenkins.war) u terminal (konzolu), Jenkins bi se automatski startovao sa svojim definisanim portom 8080 kao što je prikazano na slici 4 . Takođe moguće je definisati specifičan port za startovanje (slika 5).

Drugi način je sa sajta preuzeti specifičan paket instalaciju za određenu platformu na kojoj će se Jenkins nalaziti.



Slika 5.4 Prikaz startovanog Jenkins-a iz konzole



Slika 5.5 Proces startovanja Jenkinsa sa specifičnog porta

JENKINS INSTALACIJA (VIDEO)

How to install Jenkins on Windows 10

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

DASHBOARD

Prvi prikaz Jenkins softvera nakon instalacije

Kada se pokrene, Jenkins (fabrički) nalazi se na portu 8080. Obzirom da je Web aplikacija, možete joj pristupiti sa linka (<http://localhost:8080>). Gde će se otvoriti Radna površina (Dashboard) Jenkins alata. Glavni prozor za podešavanje Jenkins alata nalazi se u delu ManageJenkins.



Slika 5.6 Jenkins Dashboard i Menage Jenkins

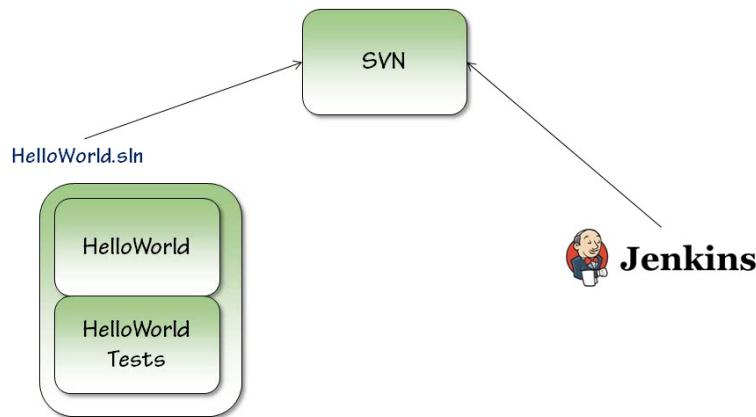
PRIMER KREIRANJA JENKINS POSLOVA

Kreiranje jednostavnog Jenkins posla i aktivnosti

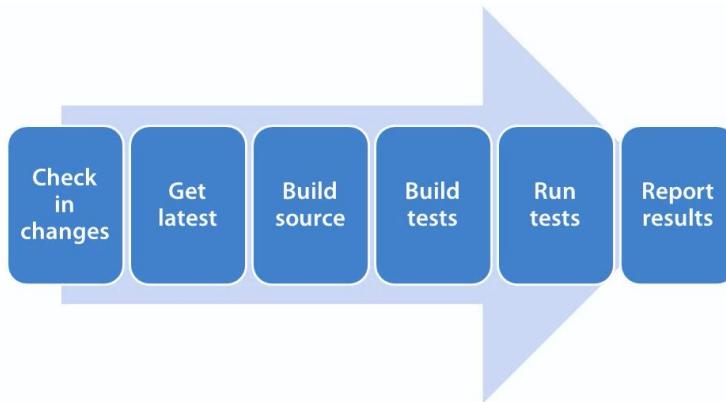
Na početku treba definisati jednostavnu strukturu projekta sistema na kome će Jenkins raditi. Na slici 7 je prikazan jednostavan projekat koji sadrži izvršnu klasu, klasu koja testira napisani kod odnosno Test klasu. Taj jednostavan projekat se prosleđuje na neki Source control u ovom primeru SVN. Dalji tok je Jenkins koji komunicira sa SVN kako bi preuzeo kod i radio na njemu.

Obzirom da će ova vežba pokazati realan primer u okviru NetBeans razvojnog okruženja napravljen je jednostavan HelloWorld primer koji će demonstrirati upotrebu Jenkins okruženja. Hello World će imati 2 jednostavna Junit testa, kao i jednu klasu. (Projekat možete preuzeti iz dodatnog materijala).

Postupak procesa postavljanja aplikacije upotrebom Jenkins-a prikazan je na slici 8 .



Slika 5.7 Prikaz arhitekture aplikacije



Slika 5.8 Proces rada Jenkins

PRIMER KRIERANJA POSLA

Proces kreiranja posla unutar Jenkins alata

Pritiskom na dugme New Items otvara se prozor za definisanje koji tip projekta se kreira. Obzirom da je demo aplikacija koja je napravljena obična Java desktop aplikacija i ne nalazi se ni na jednom SVN,GIT rezitoriju projekt koji će biti kreiran je FreeStyle project kao što je prikazano na slici 9 . Nakon odabira projekta otvara se nova stranica na kojoj se mogu defginisati posebne pogodnosti koje Jenkins omogućava za projekt koji se kreira. Postoje vremenski okidači, šta se dešava pre, a šta posle build-ovanja projekta. Okidači (Build Triggers) u Jenkinsu imaju tri mogućnosti. Prva mogućnost je (Build after other projects are built) koja označava da će ovaj projekt biti buildovan nakon uspešnog završetka nekog drugog projekta od koga možda zavisi, drugi je (Build periodically) koji omogućava periodično pokretanje Build-a npr na svakih pet minuta, jednom dnevno i dr. i treći je (Poll SCM-Poll Source control) i ovo je najkorišćeniji metod koji omogućava podesiti tako da Jenkins posmatra svaku SVN promenu i ukoliko je ima uradi build. Postoji još mogućnosti koje su pokrivene ovom metodom i mogu se pogledati na sldećem linku

(<https://wiki.jenkins-ci.org/display/JENKINS/Building+a+software+project>).

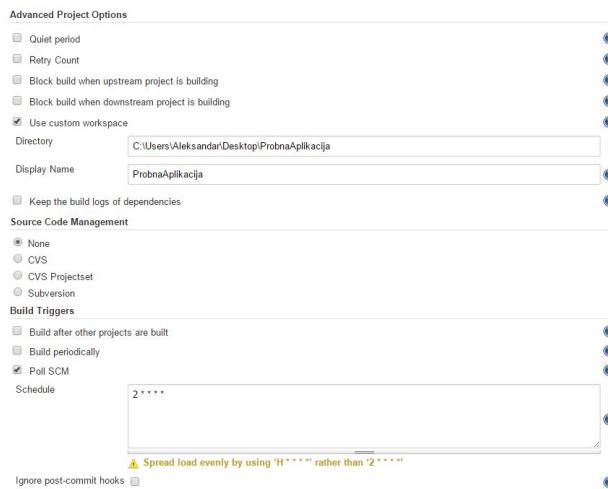
Putanja do projekta koji se Builduje u ovom primeru prikazana je na slici 10 .

The screenshot shows the Jenkins web interface with a navigation bar at the top. The left sidebar contains links for 'Home', 'All', 'Jobs', 'Builds', 'History', 'Manage Jenkins', and 'Credentials'. The main area is titled 'Create New Item' with the sub-section 'Job'. A search bar at the top right contains the text 'Traži'. The 'Item name' field is filled with 'DemoAplikacija'. Below it, there are five options with descriptions:

- Freestyle project**: This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.
- Maven project**: Build a maven project. Jenkins takes advantage of your POM files and drastically reduces the configuration.
- Build multi-configuration project**: Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.
- External Job**: This type of job allows you to recode the execution of a process run outside Jenkins, even on a remote machine. This is designed so that you can use Jenkins as a dashboard of your existing automation system. See the documentation for more details.
- Copy existing item**: Copy from [input field]

At the bottom right is an 'OK' button.

Slika 5.9 Prikaz kreiranja posla



PRIMER PREGLEDA PROJEKTA

Pregled kreiranog projekta i dobijenog workspace-a

Od naprednih podešavanja koja su bila ponuđena neophodno je podesiti putanju do Build-a, vreme izvršavanja build-a, vreme izvršavanja Builda, koje su akcije kada se Build uspešno završi ili u slučaju da Build „padne“. Nakon toga na Dashboard-u se pojavljuje projekat koji je kreiran kao što je prikazano na slici 11 .

S	W	Name	Poslednje uspešno	Poslednja greška	Poslednje trajanje
		ProbnaAplikacija	9 min 42 sec - #3	Nije dostupno	0.39 sec

ikonica: [S](#) [M](#) [L](#)

Legenda RSS za sve RSS za neuspešne RSS samo za najnovije gradnje

Slika 5.10 Prikaz projekta na Dashboard-u

Ukoliko sačekamo vreme izvršavanja Build-a ili „ručno“ pokrenemo build projekta Jenkins će ukoliko je SVN/GIT preuzeti kod ili takođe ukoliko je iz lokalnog foldera projekat kao što je u ovom slučaju. Prikaz šta je preuzeto sa Source kontrole možete pogledati pritiskom na naziv projekta na Dashboard-u i odabir Workspace. Nakon toga otvara se menadžer fajlova tog projekta. Prikaz demo projekta nalazi se na slici 12 .



Slika 5.11 Prikaz strukture foldera projekta

REZULTAT POSLA - PRIKAZ STATUSA PROJEKTA

Pregled statusa projekata

Ukoliko je projekat uspešno kreiran pored naziva projekta biće prikazan plavi krug koji označava da je build prošao uspešno. Ukoliko стоји sunce pored naziva projekta to takođe znači da su svi Build-ovi na tom projektu uspešno prošli. Ukoliko je krug crven ili nije prikazano sunce, to znači da neki od buildova nisu prošli. Prikaz uspešnog i neuspešnog build-a dat je na slici 13 .

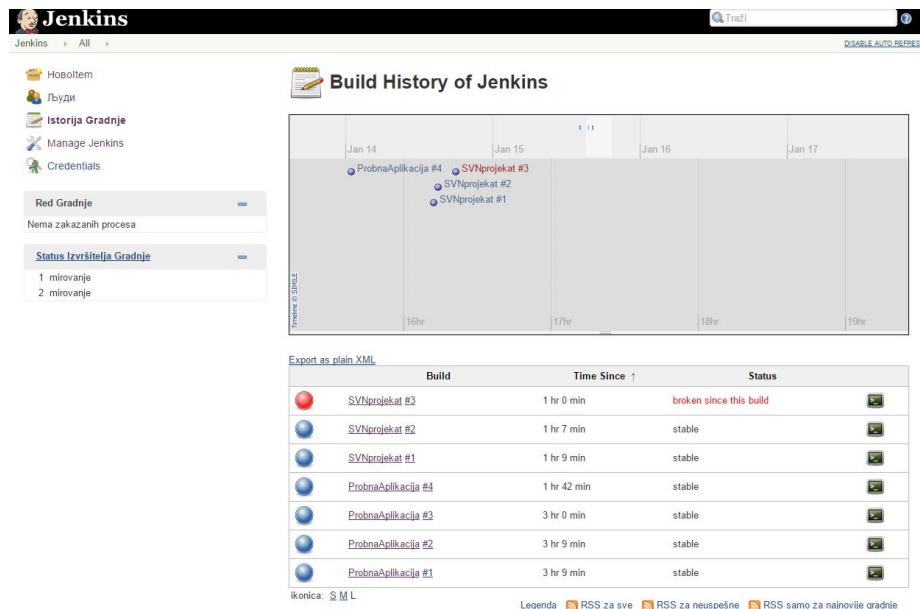
All	+	S	W	Name	Poslednje uspešno	Poslednja greška	Poslednji trajanje
		ProbnaAplikacija			1 hr 38 min - #4	Nije dostupno	0.29 s
		SVNprojekat			1 hr 3 min - #2	56 min - #3	0.84 s

ikonica: [S](#) [M](#) [L](#)

[Legenda](#) [RSS za sve](#) [RSS za neuspešne](#)

Slika 5.12 Prikaz svih statusa projekata na Dashboard-u

Postoji mogućnost detaljnijeg prikaza svih Build-ova projekata. Kao što je prikazano na slici 14 . Ukoliko se pojavi greška u build-u kada se klikne na konzolu pojavljuje se kompletan prikaz greške koja je nastala. Prikaz Greške



Build	Time Since	Status
SVNprojekat #3	1 hr 0 min	broken since this build
SVNprojekat #2	1 hr 7 min	stable
SVNprojekat #1	1 hr 9 min	stable
ProbnaAplikacija #4	1 hr 42 min	stable
ProbnaAplikacija #3	3 hr 0 min	stable
ProbnaAplikacija #2	3 hr 9 min	stable
ProbnaAplikacija #1	3 hr 9 min	stable

Slika 5.13 Grupni prikaz Build-a

REZULTAT POSLA

Pregled statusa projekata - prikaz grešaka i prikaz istorije projekta

Ukoliko je projekat uspešno kreiran pored naziva projekta biće prikazan plavi krug koji označava da je build prošao uspešno. Ukoliko стоји сунце поред назива пројекта то такође значи да су сви Build-ovi на том пројекту успешио прошли. Уколико је кружнице црвени или није приказано сунце, то значи да неки од buildova нису прошли. Prikaz uspešnog i ne uspešnog build-a dat je na slici 15 .

Postoji mogućnost detaljnijeg prikaza svih Build-ova projekata. Kao što je prikazano na slici 16 . Ukoliko se pojavi greška u build-u kada se klikne na konzolu pojavljuje se kompletan prikaz greške koja je nastala. Prikaz Greške dat je na slici 14 .

```

Started by user anonymous
Building in workspace C:\Program Files (x86)\Jenkins\jobs\SVNprojekat\workspace
Checking out a fresh workspace because there's no workspace at C:\Program Files
(86)\Jenkins\jobs\SVNprojekat\workspace
Cleaning local Directory
Checking out https://code.bmu.internal:7788/svn/se201/ProbnaAplikacija at revision '2015-01-15T15:21:21+0100'
A test
A test\demo
A test\demo\MainTest.java
A nbproject
A nbproject\build.xml
A nbproject\project.properties
A nbproject\project.xml
A nbproject\genfiles.properties
A nbproject\build-impl.xml
A manifest.mf
A manifest
A src
A src\demo
A src\demo\Main.java
A build.xml
U

At revision 3
no change for https://code.bmu.internal:7788/svn/se201/ProbnaAplikacija since the previous build
Recording test results
ERROR: hudson.tasks.junit.JUnitResultArchiver aborted due to exception
java.io.IOException: Failed to read C:\Program Files (x86)\Jenkins\jobs\SVNprojekat\workspace\manifest.mf
is this really a JUnit report file? Your configuration must be matching too many files
        at hudson.tasks.junit.TestResult.parse(TestResult.java:290)
        at hudson.tasks.junit.TestResult.parsePossiblyEmpty(TestResult.java:228)
        at hudson.tasks.junit.TestResult.parse(TestResult.java:146)
        at hudson.tasks.junit.TestResult.<init>(TestResult.java:122)
        at hudson.tasks.junit.JUnitParser$ParseResultCallable.invoke(JUnitParser.java:119)
        at hudson.tasks.junit.JUnitParser$ParseResultCallable.invoke(JUnitParser.java:93)
        at java.util.concurrent.FutureTask.run(FutureTask.java:266)
        at hudson.FilePath.act(FilePath.java:99)
        at hudson.tasks.junit.JUnitParser.parseResult(JUnitParser.java:90)
        at hudson.tasks.junit.JUnitResultArchiver.parse(JUnitResultArchiver.java:120)
        at hudson.tasks.junit.JUnitResultArchiver.perform(JUnitResultArchiver.java:137)
        at hudson.tasks.junit.JUnitResultArchiver$1.call(JUnitResultArchiver.java:74)
        at hudson.tasks.BuildStepMonitor$1$1.call(BuildStepMonitor.java:28)
        at hudson.model.AbstractBuild$AbstractBuildExecution.perform(AbstractBuildExecution.java:770)
        at hudson.model.AbstractBuild$AbstractBuildExecution.post(AbstractBuildExecution.java:183)
        at hudson.model.Run.execute(Run.java:178)
        at hudson.model.FreeStyleBuild$FreeStyleBuildExecution.post(AbstractBuild.java:683)
        at hudson.model.Run.execute(Run.java:178)
        at hudson.model.FreeStyleBuild$FreeStyleBuildExecution.post(AbstractBuild.java:43).

```

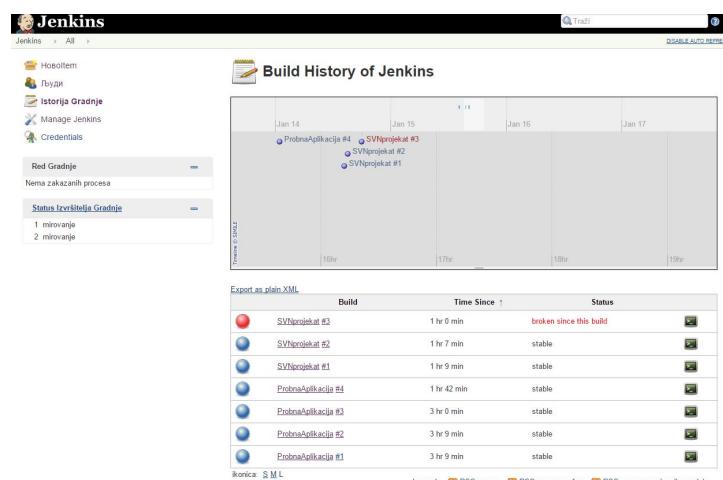
Slika 5.14 Prikaz greške tokom Build-a projekta

All	+	S	W	Name	Poslednje uspešno	Poslednja greška	Poslednji trajanj
		ProbnaAplikacija	1 hr 38 min - #4	Nije dostupno	0.29 s		
		SVNprojekat	1 hr 3 min - #2	56 min - #3	0.84 s		

ikonika: [S](#) [M](#) [L](#)

Legenda RSS za sve RSS za neuspešne

Slika 5.15 Prikaz grešaka projekta



Slika 5.16 Prikaz istorije Build-a projekta

ZADACI ZA SAMOSTALAN RAD

Upotreba Jenkins softverskog alata

1. Instalirati Jenkins korišćenjem .war fajla
2. Napraviti Java aplikaciju digitron, sa metodama za sabiranje i oduzimanje. Napraviti nekoliko Junit testova. Postaviti aplikaciju na Jenkins tako da se na svakih 5 minuta ponovo generiše (Build) aplikacija. Ukoliko Build ne prođe potrebno je da Jenkins pošalje mail na Vašu adresu.
3. Naći proizvoljan projekat na GitHub-u. Napraviti novi projekat koji će se zvati GitHubTest. Povezati projekat sa tim GitHub nalogom. Uraditi generisanje projekta svako jutro u 9AM u slučaju da je bilo promena na kodu.
4. Pronaći i instalirati bilo koji Plugin na Jenkins okruženju

▼ Poglavlje 6

Zaključak

ZAKLJUČAK

Evolucija softvera je proces izmena softvera tokom njegovog životnog veka, koje se rade u cilju uklanjanja grešaka, prilagođavanju promenama u okruženju i dodavanja novih funkcija

Pouke ove lekcije:

1. Razvoj i evolucija softvera treba da budu integrisani i iterativni procesi koji se mogu prestaviti spiralnim modelom.
2. Troškovi održavanja sistema razvijenih po posebnom zahtevu kupca, najčešće su veći od troškova nabavke (razvoja) sistema.
3. Proces evolucije softvera je vođen zahtevima za promenama, koje obuhvataju analizu uticaja promena, planirana izdanja softvera i promenu implementacije.
4. Lemanov zakoni, kao što je onaj o kontinualnim promenama, opisuju niz iskustava koji su rezultat dugotrajnih studija o evoluciji sistema.
5. Postoji tri tipa održavanja softvera: a) popravka grešaka, b) promena softvera da bi radi sa novim okruženjem, i c) implementacija ovih ili promenjenih zahteva.
6. Reinženjering softvera se bavi promenom strukture softvera i njegove dokumentacije da bi se olakšale sprovođenje i razumevanje promena softvera.
7. Restrukturiranje softvera omogućava da se malim promenama u program održava njegova funkcionalnost, ima karakter preventivnog održavanja.
8. Poslovna vrednost starih sistema i kvalitet aplikativnog softvera i njegovog okruženja bi trebalo da bude ocenjeno da bi se donela odluka da li bi trebalo da sistem bude zamenjen, transformisan ili zadržan.