**Seminarski rad**

**iz predmeta Razvoj softvera 2**

**tema:**

**Software evolution**

**Razvoj softvera**

**Sanja Petrović 1110/2012**

**Matematički fakultet**

**09.01.2013.**

**Razvoj softvera**

Ovo poglavlje govori zasto je razvoj softvera bitan deo softverskog inzenjerstva I objasnjava procese razvoja softvera.

Razvoj softvera ne prestaje nakon isporucivanja vec se nastavlja tokom citavog zivota softvera. Nakon sto je sistem isporucen neophodno je da se izvrsavaju promene na njemu ukoliko zelimo da ostane koristan. Poslovne promene i promene zbog korisnickih ocekivanja stvaraju nove uslove za postojeci softver. Delovi softvera mogu biti izmenjeni radi korekcije greske koje su pronadjene tokom operacija, da bi se prilagodio softverskoj I hardverskoj platform, da bi se poboljsale preformanse softvera ili neke druge nefunkcionalne karakteristike.

Razvoj softvera je veoma bitan jer firme ulazu ogromne kolicine novca u njihov softver I uglavnom kompletno zavise od tog softvera. Sistemi su kriticna poslovna imovina I oni moraju da ulazu u svoje sisteme kako bi odrzali vrednost te imovine. Shodno tome, trose mnogo vise na odrzavanje postojecih sistema nego na razvoj novih sistema.

Razvoj softvera moze biti poktenut usled izmene poslovnih zahteva, prilikom pojave gresaka u radu softvera ili promenom sistema na kojem se radi. Dakle, evolucija sistema se retko moze posmatrati posebno. Promene okruzenja koje dovode do promene sistema mogu kasnije dovesti do dodatnih promene tog istog okruzenja. Naravno, cinjenica da sistemi teze da evoluiraju do “sistema sa bogatim okruzenjem” cesto uzrokuje komplikovan I skup razvoj. Kao sto je neophodno dobro poznavanje I analiziranje uticaja na sam sistem koje mogu izvazvati predlozene promene, mozda je potrebno proceniti kako ove promene mogu uticati I na druge sisteme u operativnom okruzenju.

Kvalitetni I korisni softveri uglavnom imaju dug zivot. Na primer, sistemi za kontrolu leta imaju zivotni vek 30 godina cak I vise. Sistemi obicno kostaju dosta novca pa kompanije teze ka tome da ih sto duze koriste kako bi se ulaganja isplatila. Ocigledno, prilikom zahteva za promenama sistema dolazi kako do poslovnih tako I do promena okruzenja. Dakle, nova izdanja sistema, uvodjenje promena, azuriranja se obicno rade u redovnim intervalima.

Zbog toga treba da razmisljamo o razvoju softvera kao spiralnom procesu koji podrazumeva zahteve, dizajn, implementacije I testiranja koji se desavaju tokom zivotnog veka sistema. Pocinjemo tako sto pravimo prvo izdanje sistema. Kada je jednom isporucen, do promena isporucenog izdanja kao I razvoj drugog izdanja krecu istovremeno. Potreba za razvojem moze da se javi I pre nego sto se sistem isporuci sto moze dovesti do toga da neka kasnija izdanja softvera mogu biti u fazi razvoja I pre nego sto je tekuca verzija objavljena.

Ovaj model razvoja softvera podrazumeva da je jedna organizacija odgovorna I za pocetni razvoj softvera kao I za kasniji razvoj softvera. Mnogi paketi softverskih proizvoda su isporuceni koristeci ovaj pristup. Za prilagodjen softver se koristi drugaciji pristup. Kompanije za razvoj softvera prave softver za svoje klijente, a nakon toga zaposleni tog klijenta koji se bave razvojem preuzimaju sistem. Oni su odgovorni za razvoj softvera. Takodje, kupac softvera moze napraviti dogovor za nekom drugom kompanijom koja ce vrsiti odrzavanje I razvoj sistema.

U ovom slucaju, oni bi prekinuli taj spiralni proces. Zahtevi I dokumentacija se ne mogu preneti iz jedne kompanije u drugu. Kompanije mogu spojiti ili reorganizovati I naslediti softver od drugih kompanija, a da onda pronadju sta je potrebno izmeniti. Kada tranzicija od razvoja do evolucije nije odgovarajuca proces menjanja softvera nakon isporuke se cesto naziva “odrzavanje softvera”. Odrzavanje softvera prouzrokuje dodatne aktivnosti kao sto su potreba za razumevanjem softvera(kako funkcionise) pored uobicajenih aktivnosti prilikom razvoja softvera.

Rajlich I Bannett (2000) uvode novi pogled na odvijanje razvoja softvera. U ovom modelu, oni prave razliku izmedju razvoja I servisiranja. Razvoj je faza koja podrazumeva znacajne promene koje mogu da se obave na softveru. Servisiranje je faza u kojoj se vrse neke male, sustinske promene.

Prilikom razvoja, softver se uspesno koristi I postoji konstantan tok izvrsavanja promena predlozenih prilikom zahteva. Kako god, ako je softver menjan, njegova struktura tezi degradaciji I promene na softveru postaju sve skuplje I skuplje. Ovo ce obicno desava nakon nekoliko godina koriscenja softvera prilikom kojih je cesto dolazilo do zahteva za promenama okruzenja, kao sto su hardverske promene I promene operativnog sistema. U nekom stadijumu zivotnog ciklusa, softver dostize tacku tranzicije gde dolazi do znatnih promena I gde sprovodjenje novih zahteva postaje sve manje I manje isplativo.

U tom stadijumu softver prelazi sa razvoja na servisiranje. Tokom faze servisiranja, softver je I dalje upotrbljiv I koristan ali na njemu su moguce samo minimalne promene. Prilikom ovog stadijuma kompanije uglavnom razmatraju kako ovaj softver moze biti zamenjen. U krajnjem stadijumu, faza iskljucivanja, softver se I dalje moze koristiti ali se ne sprovode nikakve dalje promene. Korisnici moraju sve probleme koji se pojavljuju prilikom rada resavati na neki drugi nacin.

**Proces razvoja softvera**

Proces razvoja softvera dosta zavisi od vrste softvera koji se odrzava, procesa koji je koriscen za razvoj tog softvera I od vestine ljudi koji su ukljuceni u proces razvijanja. U nekim organizacijama, razvoj moze biti neformalni proces gde promene uglavnom poticu od komuniciranja izmedju korisnika sistema I programera. U drugim kompanijama, ovo je formalni proces sa obimnom dokumentacijom koja prati svaku fazu u procesu razvoja softvera.

Predlozi promena sistema su putokaz u razvoju sistema u svim organizacijama. Predlozi promena mogu doci od postojecih zahteva ali koji nisu ukljuceni u realizaciju sistema, od zahteva za novim zahtevima, prijave gresaka od strane korisnika sistema I novih ideja za poboljsanje sistema od strane razvojnog tima. Proces uocavanja promena I razvoja sistema je ciklican I nastavlja se kroz zivotni vek sistema.

Predlozi promena trebaju biti povezani sa komponentama sistema koje trebaju biti izmenjene kako bi se ukljucili ovi predlozi. Ovo omogucava odredjivanje cene I uticaja promene na sistem. Ovo je deo celokupnog procesa upravljanja promenama sto takodje treba da obezbedi da su odgovarajuce verzije komponenata ukljucene u svaku od realizacija sistema.

*Arthur (1988)* ukazuje na proces razvoja softvera. Proces ukljucuje osnovne aktivnosti analize promena, planiranja, implementacije sistema kao I isporucivanja softvera klijentima. Cena I uticaj ovih promena se koriste da bi se procenilo I videlo koliki procenat sistema ce podlegnuti promenama kao I kolika ce biti cena implementacije tih promena. Ako su predlozene promene prihvatljive planira se nova realizacija sistema. Prilikom planiranja realizacije sve predlozene promene se uzimaju u obzir. Zatim se pravi odluka koja od promena ce biti ukljucena u narednoj verziji sistema. Promene se implementiraju I proveravaju I nova verzija sistema se realizuje. Nakon toga se proces ponavlja sa novim skupom promena koje su predlozene za narednu verziju.

Mozemo razmisljati o implementaciji promena kao o iteraciji razvojnog procesa gde se izmene sistema dizajniraju, implementiraju I testiraju. Kako god, velika razlika je ta da prvi stadijum implementacije promena moze da dovede do proucavanja programa, pogotovu ako prvobitni razvijaoc sistema nije odgovoran za implementaciju promena. Tokom faze proucavanja programa bitno je razumeti strukturu programa, na koji nacin pruza funkcionalnost I kako predlozena promena moze uticati na program. Razumevanje programa je neophodno da bi bili sigurni da implementirana promena nece prouzrokovati nove probleme kada se ukljuci u sistem.

U idealnom slucaju, stadijum implementacije promene ovog procesa moze promeniti specifikaciju sistema, dizajn I implementacija treba da odrazavaju promene sistema. Novi zahtevi koji odrazavaju promene sistema su predlozeni, analizirani I provereni. Komponente sistema su redizajnirane I implementirane I sistem je testiran. Ukoliko je potrebno, prototip predlozenih promena moze biti ukljucen kao deo procesa analize promena.

Tokom procesa razvoja, zahtevi se analiziraju do detalja I pojavljuju se posledice promena koje nisu bile vidljive u ranijem procesu analiz. Ovo podrazumeva da predlozene promene mogu biti modifikovane I da je nekada potrebno razgovarati sa klijentom pre nego sto se one implementiraju. Zahtevi za promenama se nekad odnose na sistemske probleme koje je potrebno hitno resavati. Ove urgentne promene mogu nastati iz tri razloga:

1. Ukoliko ozbiljna sistemska greska javlja da mora biti popravljena kako bi se nastavilo normalno funkcionisanje sistema
2. Ukoliko promene u random okruzenju sistema prouzrokuju neocekivane efekte koji ometaju normalne operacije
3. Ukoliko postoje nepredvidjene promene u poslu rada na sistemu kao sto su pojava novih konkurenata ili uvodjenje novog zakonodavstva koje utice na sistem

U ovim slucajevima, potreba da se promene sprovedu brzo podrazumeva da mozda necemo biti u mogucnosti da pratimo proces formalne analize promena. Umesto izmena uslova I dizajna vrsimo hitnu prepravku programa kako bi resili odmah problem. Opasnost je ta sto zahtevi, dizajn softvera I kod ne ostaju u skladu. Iako planiramo da u dokumentaciju unesemo podatke o promenama u zahtevima I dizajnu tada se moze pojaviti jos dodatnih hitnih prepravki u softveru. Ovo ima prioritet nad dokumentacijom. Eventualno, originalne promene mogu biti zaboravljene I tada se dokumentacija sistema I kod nikada nece poklapati.

Hitne sistemske popravke obicno zahtevaju da budu odradjene sto pre. Uvek se bira brze I efektivnije resenje nego najbolje moguce sve dok je sistem u kriticnoj fazi. Ovo ubrazava proces starenja softvera tako da kasnije promene postaju sve komplikovanije I odrzavanje sistema postaje skuplje.

U idealnom slucaju, kada su izvrsene sve hitne popravke koda zahtev za promenama bi ostao inzvarendan I nakon popravki gresaka u kodu. Oni mogu biti reimplementirani jos pazljivije nakon kasnije analize. Naravno, kod za popravku greske moze ponovo da se koristi. Bolje resenje problema bi se moglo pronaci kada bi bilo moguce analizirati ga vise puta. Ipak, u praksi gotovo je neizbezno to da ova poboljsanja imaju nizak prioritet. Cesto se na njih zaboravlja I ukoliko se izvrse jos neke promene sistema postaje skoro nemoguce da se hitne prepravke koriguju.

Agilne metode I procesi spomenuti u odeljku 3 se mogu koristiti za razvoj programa. S ozbirom da su ove metode zasnovane na inkrementalnom razvoju, tranzicija od agilnog razvoja do razvoja nakon isporuke je dobra. Tehnike kao sto je automatsko testiranje promena su korisne kada su izvrsene promene sistema. Ukratko, razvoj podrazumeva kontinuirano koriscenje agilnih metoda.

Takodje problemi se mogu javiti u situacijama kada se vrsi predaja softvera od tima koji napravio softver timu koji ucestvuje u kasnijem razvoju tog softvera. Postoje 2 moguce problematicne situacije:

1. Kada stvaralacki tim koristi agilni pristup a razvojni tim koji preuzima softver ne koristi agilni pristup vec neki pristup zasnovan na planiranju. Tim za razvoj softvera moze ocekivati detaljnu dokumentaciju radi olaksavanja daljeg razvoja a to se retko desava u agilnom pristupu. Moze da se dogodi da nema definitivnih podataka o zahtevima sistema koji mogu biti modifikovani kao promene koje su izvrsene na sistemu.
2. Kada se za izradu softvera koristi pristup zasnovan na planiranju, a ako razvojni tim koristi agilni pristup. U ovom slucaju ce razvojni tim morati da krene od nule da razvija automatizovane testove I kod u sistemu mozda nece biti rafaktorisan I uproscen kao sto se to podrazumeva kod agilnog razvoja. U ovom slucaju, kod nekih rekonstruisanja moze biti potrebno da se poboljsa kod pre nego sto se upotrebi u agilnom razvoju.

Poole and Huisman (2001) su pisali o njihovom iskustvu u odrzavanju velikog sistema koristeci ekstremno programiranje, a koji je prvobitno napravljen koristeci pristup baziran na planiranju. Nakon rekonstruisanja sistema kako bi se poboljsala njegova struktura ekstremno programiranje se pokazalo kao koristan nacin za odrzavanje sistema.

**Dinamika razvoja programa**

Dinamika razvoja programa je studija promene sistema. 1970ih I 1980ih Lehman I Belady su sproveli nekoliko empirijski studija o promeni sistema sa ciljem da sto vise uzmu u obzir karakteristike razvoja softvera. Rad se nastavio 1990ih kada su Lehman I ostali istrazivali znacaj povratnih informacija u razvojnom procesu. Iz ovi studija, oni su predlozili “Lehmanovi zakoni” o promeni sistema.

Lehman I Belady su tvrdili da ce ovi zakoni verovatno vaziti za sve tipove velikih softverskih sistema. Ovo su sistemi u kojima se zahtevi menjaju da bi odrazavali promenljive poslovne potrebe. Nove realizacije sistema su od sustinskog znacaja za sistem da bi obezbedili poslovnu vrednost.

*Prvi zakon* kaze da je odrzavanje sistema neizbezan proces. Kako se menja okruzenje sistema pojavljuju se novi zahtevi I sistem se mora modifikovati. Kada se modifikovan sistem ponovo vrati u to okruzenje ovo podstice jos promena okruzenja I razvojni proces pocinje ponovo.

*Drugi zakon* kaze da ako je sistem promenjen, njegova struktura je degradirana. Jedini nacin da se ovo izbegne je da se ulaze u preventivno odrzavanje. Unapredjujemo strukturu softvera, ali ne radeci na njegovoj funkcionalnosti. Ocikledno, ovo prouzrokuje dodatne troskove.

*Treci zakon* je najzanimljiviji. Predlaze da veliki sistemi imaju sopstvenu dinamiku koja je osnovana u ranoj fazi razvoja. Ovo definise grube tokove u procesu odrzavanja sistema I limitira broj pojava mogucih sistemskih promena. Lehman I Belady govore da je ovaj zakon posledica strukturnih faktora koji ogranicavaju I uticu na promenu sistema I organizacionih faktora koji uticu na proces razvijanja.

Strukturni faktori koji podsticu treci zakon dolaze od kompleksnosti velikih sistema. Kada menjamo I prosirujemo program njegova struktura tezi degradaciji. Ovo vazi za sve tipove sistema (ne samo za softver) I do toga dolazi zato sto adaptiramo strukturu namenjenu za jednu svrhu za razlicite namene. Ova degradacija, ako nije kontrolisana, pravi sve vece I vece poteskoce da kasnije dodje do nekih promena u programu. Pravljenje malih promena smanjuje stepen degradacije structure I smanjuje rizike pojave ozbiljnih problema u funkcionisanju sistema. Ako pravimo velike izmene velika je verovatnoca da ce to prouzrokovati nove probleme. To onda sprecava dalje programske promene.

Organizacioni faktori koji uticu na treci zakon odrazavaju cinjenicu da se veliki sistemi uglavnom prave od strane velikih organizacija. Ove kompanije imaju unutrasnje birokratije koje odredjuju promenu budzeta za svaki sistem I koje kontrolisu proces odlucivanja. Kompanije moraju doneti odluke o rizicima I vrednostima promena kao I o troskovima koji su ukljuceni. Takve odluke zahtevaju vreme da se donesu, ponekad duze traje donosenje odluka o promenama koje traba da se izvrse, nego sama implementacija promena. Brzina procesa odlucivanja o promenama regulise stopu promene sistema.

*Lehmanov cetvrti zakon* govori o tome da vecina velikih projekata rade u “zasicenim” firmama. Tj. promena resursa ili promena osoblja ima minimalan efekat na dugorocan razvoj sistema. Ovo je u skladu sa trecim zakonom koji govori da evolucija programa u velikoj meri ne zavisi od odluka menadzmenta. Ovaj zakon potvrdjuje da su veliki timovi za razvoj softvera cesto neproduktivni zbog komunikacije medju clanovima tima koja oduzima veci deo vremena.

*Lehmanov peti zakon* se tice promena inkrementalnih koraka u svakoj realizaciji sistema. Dodavanje nove funkcionalnosti sistemu uzrokuje nove probleme I promene. Sto se vise funkcionalnosti dodaje vise je I problema koji nastaju I koji se moraju resiti. Velika promena funkcionalnosti u jednoj realizaciji sistema pozdrazumeva da ce to pratiti narednu realizaciju sistema u kojoj ce novonastali sistemski problem biti reseni. Relativno malo novih funkcija treba da bude ukljuceno u ovoj realizaciji. Ovaj zakon kaze da ne bi trebalo praviti budzet za velike funkcionalne promene u svakoj od realizacija bez uzimanja u obzir potrebe za popravljanjem gresaka.

*Sesti I sedmi zakon* su slicni I u sustini kazu da ce korisnici softvera biti jako nesrecni sve dok je softver u fazi razvoja I dok se dodaju nove funkcionalnosti. Poslednji zakon govori o radu na povratnim informacijama iako jos uvek nije jasno kako ce to da se uklopi u proces razvoja softvera.

**Odrzavanje softvera**

Odrzavanje softvera podrazumeva promene sistema nakon njegovog isporucivanja. Promene na softveru mogu biti promene koje su se pojavile u kodu prilikom kodiranja, malo veca promena je promena u dizajnu, promene radi ispravljanja gresaka u specifikaciji ili promene usled ubacivanja novih zahteva. Promene se implementiraju menjajuci postojeci sistem I gde je to potrebno, dodavanje novih komponenata u sistem.

Postoje tri razlicita tipa odrzavanja sistema:

1. *Popravke gresaka* – greske u kodiranju su obicno jeftine za korekciju, greske u dizajnu su skuplje jer one mogu podrazumevati prepravljanje nekoliko komponenti programa. Greske u definisanju zahteva su najskuplje za popravku jer obimnog redizajna koji je uglavnom neophodan.
2. *Adaptacija na okruzenje* – ovaj tip odrzavanja je potrban kada ne postoji sklad sa sistemskim okruzenjem kao sto je hardver, platforma operativnog sistema,… Aplikativni sistem mora biti izmenjen kako bi se prilagodio ovim razlikama u okruzenju.
3. *Dodavanje funkcionalnosti* – ovaj tip odrzavanja je neophodan kada dodje do promena u sistemskim zahtevima sto je najcesce pruzrokovano promenama u organizaciji ili poslovanju.u ovom slucaju je izmena softvera uglavnom mnogo veca nego u prethodna dva slucaja.

U praksi, cesto ako dodje do izmena sistema usled adaptacije na novu okolinu treba dodati I neke nove funkcionalnosti kako bi se iskoristile prednosti novog okruzenja. *Softverske greske* su cesta pojava jer korisnici cesto koriste sofver na neki neocekivani nacin. Promena sistema kako bi se olaksao rad na njima je najbolji nacin za resavanje ovih problema.

Istrazivanja se slazu sa tim da se u odrzavanje sistema ulaze mnogo vise novca nego u razvijanje novog softvera. Takodje se slazu sa tim da veci deo budzeta ide na implementiranje novih zahteva u sistem nego na popravljanje bagova. Prilagodjavanje sistema za novo okruzenje podrazumeva najvise napora.

Obicno je ispaltivo ulagati napore u osmisljavanje I sprovodjenje sistema za smanjenje cena buducih promena. Dodavanje novih funkcionalnosti nakon isporucivanja softvera je skupo zato sto je neophodno dobro prouciti I razumeti sistem I analizirati uticaj predlozenih promena. Dakle, rad tokom razvoja softvera na tome da softver ucinimo laksim za razumevanje I menjanje najverovatnije dovodi do kasnijeg smanjenja troskova prilikom razvoja. Dobre tehnike inzenjeringa, kao sto je precizna specifikacija, koriscenje objektno-orijentisanog razvoja I upravljanje konfiguracijom doprinose smanjenju troskova prilikom odrzavanja.

Ove procene su hipoteticke ali tu nema sumnje da ako prilikom razvijanja softvera pazimo na to da on kasnije bude laksi za modifikaciju cini da kasnije promene kostaju manje. To je razlog za refaktorisanje u agilnom razvoju. Bez refaktorisanja kod postaje sve vise I vise komplikovan I skup za promene. U razvoju zasnovanom na planiranju realnost je da je dodatno ulaganje u poboljsanje koda vrlo retko prilikom razvoja. To je uglavnom zbog nacina na koji vecina organizacija vodi svoje budzete. Ulaganje u odrzavanje dovodi do kratkorocnog povecanja troskova, koji su merljivi. Nazalost, dugorocni dobitci se ne mogu meriti u isto vreme tako da kompanije uglavnom ne zele da trose novac na nesto ciji se ishod ne zna tacno u buducnosti.

Obicno je mnogo skuplje dodavati funkcionalnosti kada je sistem vec isporucen I u fazi koriscenja nego dok je u fazi ravoja. Razlozi za ovo su:

1. *Stabilnost tima* – nakon sto je sistem isporucen normalno je da se razvojni tim razidje I da ljudi koji su radili na tom projektu pocnu da rade na nekom drugom projektu. Novi tim ili clanovi u timu koji su odgovorni za odrzavanje sistema ne poznaju bas najbolje sistem I zbog cega su neke odluke prilikom razvoja donate. Oni moraju provesti neko vreme u proucavanju sistema pre nego sto krenu da uvode promene.
2. *Losa praksa razvoja* – ugovor za odrzavanje sistema je uglavnom razdvojen od ugovora za razvoj tog istog sistema. Ugovor o odrzavanju sistema moze biti sklopljen sa nekom drugom kompanijom umesto sa kompanijom koja je razvila sistem. Ovaj faktor zajedno sa nedostatkom stabilnosti tima, znaci da tu nema podsticaja za razvojni tim da ucestvuje u odrzavanju sistema. Svakako bi bilo dobro kada bi razvojni tim mogao da nastavi da radi I odrzavanje softvera kako bi na taj nacin mozda smanjio troskove I tako otezao implementaciju promena softvera u buducnosti.
3. *Vestine osoblja* – odrzavanje sistema obicno ostavlja losu sliku medju inzenjerima softvera. Cini se kao da je to posao za koji je potrebno manje vestina od razvoja softvera I obicno se ti poslovi ostavljaju najmladjem osoblju. Stavise, stari sistemi mogu biti pisani na starim programskim jezicima. Osoblje za odrzavanje sistema cesto nema dovoljno iskustva u programiranju u tim jezicima I moraju da uce ove jezike kako bi odrzavali sistem.
4. *Starost programa i struktura* – kako se neka promena izvrsi na programu njihova struktura tezi degradiranju. Cinjenica je da sto su programi stariji da su tezi za razumevanje I izmenu. Neki sistemi su razvijeni bez koriscenja modernih tehnika softverskog inzinjerstva. Oni verovatno nikada nisu bili dobro strukturisani I verovatno su vise optimizovani za efikasnost nego za razumljivost. Dokumentacija moze biti izgubljena ili nedosledna.

Prva tri problema poticu od cinjenice da mnoge organizacije I dalje smatraju da razvoj I odrzavanje sistema treba da budu dve razdvojene aktivnosti. Na odrzavanje se gleda kao na sekundarnu aktivnost I tu ne postoji podsticaj za ulaganjem novca prilikom razvoja kako bi se kasnije smanjile cene uvodjenja promena u sistem.

Tehnike softverskog reinzinjeringa mogu se koristiti kako bi unapredile strukturu sistema kao I njegovu razumljivost. Arhitekturne transformacije mogu pomoci da se sistem prilagodi novom hardveru. Refaktorisanje moze unaprediti kvalitet koda sistema I olaksati njegovo menjanje.

**Prognoze odrzavanja**

Menadzeri mrze iznenadjenja pogotovu ako je rezultat povecanje troskova. Zbog toga treba da pretpostavimo koje promene na sistemu bi mogle biti predlozene I koji delovi sistema bi verovatno bili najkomplikovaniji za odrzavanje. Takodje bi trebali da procenimo ukupan trosak odrzavanja sitema za neki dati vremenski period. Predvidjanje broja zahteva za promene sistema podrazumeva razumevanje odnosa izmedju sistema I njegove okoline. Neki sistemi imaju veoma komplikovan odnos sanjihovom spoljnom sredinom I promene u okolini dovode I do obaveznih promena na sistemu. Da bi odredili odnos izmedju sistema I njegove okoline, moramo proceniti:

1. *Broj I kompleksnost interfejsa sistema* – sto je veci broj interfejsa I sto su oni slozeniji veca je verovatnoca da ce promene interfejsa biti potrebne kada se novi zahtevi predloze.
2. *Broj sustinski promenljivih zahteva za izmenu sistema* – zahtevi koji odrazavaju organizacione politike I procedure ce verovatno prouzrokovati vise promena nego zahtevi koji su zasnovani na stabilnom domenu karakteristika.
3. *Poslovni procesi u kojima se koristi sistem* – kako se poslovni procesi razvijaju tako se stvaraju novi zahtevi za promene sistema. Sto vise poslovnih procesa koristi jedan sistem vise ce biti I zahteva za promenu sistema.

Istrazivaci su posmatrali odnose izmedju slozenosti programa, koja je merena pokazateljima kao sto je uslovna slozenost I izmedju odrzavanja sistema. Nije iznenadjujuce to sto je ova studija pokazala da sto su programi kompleksniji to je skuplje kasnije odrzavanje sistema. Kompleksna merenja su posebno korisna u pronalazenju komponenti programa koje ce biti skupe za odrzavanje. Kafura I Reddy (1987) ispitivali su broj komponenti sistema I dosli su do toga da proces odrzavanja tezi da se fokusira na manji broj kompleksnih komponenti. Kako bi se smanjili troskovi odrzavanja treba se truditi da kompleksne delove programa zamenimo manje kompleksnim alternativama.

Nakon sto sistem pustimo u koriscenje, mozemo iskoristiti to da pogledamo rezultate rada kako bi predvideli proces odrzavanja sistema. Primeri metrika koje mozemo iskoristiti za ocenjivanje odrzavanja sistema:

1. *Broj zahteva za korektivno odrzavanje* – povecanje u broju bagova I broju prijavljenih gresaka moze ukazato na to da u program postoji vise gresaka nego sto je ispravljeno tokom procesa odrzavanja. Ovo moze ukazati na pad odrzivosti.
2. *Prosecno vreme potrebno za analizu efekta* – ovo ukazuje na broj komponenti programa koje su obuhvacene zahtevom za promenu sistema. Ako je ovo vreme povecano, to znaci da je sve vise I vise takvih komponenti cime odrzavanje sistema opada.
3. *Prosecno vreme potrebno za implementaciju zahtevane promene* – ovo predstavlja potrebno vreme za modifikaciju sistema I njegove dokumentacije, nakon sto procenimo koje ce komponente biti izmenjene. Povecanje vremena potrebnog za implementaciju promene moze ukazati na pad odrzivosti sistema.
4. *Broj izuzetnih zahteva za promenama* – povecanje ovog broja vremenom moze ukazati pad odrzivosti sistema

**Softverski reinzenjering**

Proces razvoja sistema ukljucuje program koji mora biti izmenjen, a kasnije tako izmenjenog ga treba ukljuciti u kompletan sistem. Mnogi sistemi, pogotovu stariji sistemi, su teski za razumevanje I menjanje. Programi mogu biti optimizovani radi poboljsanja performansi ili korigovanja kolicine memorije koja im je potrebna, ili ipak vremenom inicijalna programska struktura moze biti ostecena usled cestih promena.

Kako bi ucinili da stariji softveri budu laksi za odrzavanje, mozemo se baviti reinzenjeringom ovih sistema I na taj nacin unaprediti njihovu strukturu I razumljivost. Reinzenjering moze obuhvatiti preuredjivanje dokumentacije, refaktorisanje arhitekture sistema, prevodjenje programa na moderne programske jezike I modifikaciju I azuriranje strukture i vrednosti sistemskih podataka. Funkcionalnost softvera nije izmenjena I naravno, treba se truditi I izbegavati velike promene u arhitekturi sistema.

Prednosti reinzenjeringa u odnosu na zamenu softvera:

1. *Smanjena opasnost* – postoji veliki rizik u ponovnom razvoju softvera koji je od velike vaznosti za poslovanje. Mogu se napraviti greske u specifikaciji sistema ili se mogu pojaviti problemi u razvoju. Kasnjenja u uvodjenju novog softvera mogu dovesti do toga da se neki poslovi izgube I tada nastaju novi troskovi.
2. *Smanjeni troskovi* – trosak prilikom reinzenjeringa moze biti dosta manji nego trosak razvijanja novog softvera.

9.11

Ulaz u proces uglavnom obuhvata neki stari softver, dok izlaz iz procesa obuhvata isti taj softver samo poboljsan. Osnovni koraci u reinzenjeringu:

1. *Prevodjenje izvornog koda* – koristeci neki od alata program se konvertuje sa starog programskog jezika na neku od novijih verzija istog tog programskog jezika ili na neki novi programski jezik.
2. *Prepravljanje softvera* – program je analiziran I dobijene su sve potrebne informacije. Ovaj proces je isto kompletno automatizovan.
3. *Unapredjivanje programske strukture* – kontrolna struktura programa je analizirana I modifikovana kako bi se ucinila laksom za citanje I razumevanje. Ovaj deo moze biti delom automatizovan ali neke manualne konverzije su cesto neophodne.
4. *Rekonstruisanje podataka* – podaci programa su izmenjeni kako bi ukazali na promene u programu. Ovo moze podrazumevati redefinisanje seme baze podataka I konvertovanje postojece baze podataka u novu strukturu. Obicno treba I da brisemo neke podatke. Ovo podrazumeva pronalazenje I korekciju gresaka, skloniti duple zapise itd. Postoje razni alati koji podrzavaju softverski reinzenjering.

Programski reinzenjering ne mora sadrzati sve korake sa seme 9.11. Ne moramo koristiti prevodjenje programskog jezika ukoliko jos uvek koristimo programski jezik aplikacije sa kojom radimo. Ukoliko ceo reinzenjering odradimo automatski, onda dopunjavanje dokumentacije ne mora biti neophodno. Rekonstruisanje podataka je jedino neophodno jedino ukoliko je struktura podataka u programu izmenjena prilikom rekonstruisanja sistema.

Kako bi namestili da rekonstruisani sistem bude interoperatibilan sa novim softverom, mozda ce biti neophodno da razvijemo adapter servise. Oni kriju originalne interfejse softverskog sistema I pokazuju nove, bolje struktuirane interfejse koji mogu biti korisceni od strane drugih komponenti. Ovaj proces obradjivanja starih sistema je veoma vazna tehnika za razvijanje velikih reiskoristivih usluga.

Cene rezinzenjeringa zavise od obima rada koji je sproveden. Postoji spektar mogucih pristupa reinzenjeringu. 9.12.

Problem kod softverskog reinzenjeringa je taj sto postoje granice u tome koliko mozemo da unapredimo sistem koristeci inzenjering. Nije nemoguce prevesti program za koji je koriscen funkcionalni pristup na neki objektno-orijentisani jezik. Velike promene u arhitekturi ili radikalna reorganizacija sistema za upravljanje podacima ne moze biti sprovedena automatski pa je zbog toga jako skupa. Iako reinzenjering moze unaprediti odrzavanje, rekonstruisani sistemi verovatno nece biti odrzavani kao novi razvijeni sistemi za koje su koriscene moderne tehnologije inzenjerstva.

**Preventivno odrzavanje sistema refaktorisanjem**

Refaktorisanje je proces uvodjenja poboljsanja u sistem radi usporavanja degradacije prilikom promena. To podrazumeva modifikovanje programa kako bi se unapredila njegova struktura, kako bi se smanjila njegova kompleksnost ili kako bi ga ucinili jednostavnijim za razumevanje. Za refaktorisanje se nekada smatra da je predvidjeno samo za objektno-orijentisani razvoj, medjutim moze se vrsiti u bilo kom okruzenju. Prilikom refaktorisanja ne dodajemo nove funkcionalnosti, vec samo uvodimo poboljsanja u kod. Dakle, refaktorisanje na neki nacin smanjuje probleme u buducim izmenama.

Mada se I refaktorisanje I reinzenjering koriste kako bi uprostili softver sa razumevanje I menjanje, oni nisu ista stvar. Reinzenjering nastupa nakon toga sto je sistem vec neko vreme koriscen I odrzavan I troskovi odrzavanja ulavnom rastu. Ovde se koriste automatizovani alati I vrsimo rekonstruisanje starih sistema kako bi dobili novi sistem sto je dosta tezi vid odrzavanja. Refaktorisanje je kontinuiran proces unapredjivanja za vreme razvoja softvera. Namena je da izbegne degradaciju koda I strukture koji izazivaju troskove I probleme prilikom kasnijeg odrzavanja sistema.

Refaktorisanje je sastavni deo agilnih metoda kao sto je ekstremno programiranje jer su ove metode zasnovane na promenama. Kvalitet programa je podlozan degradiranju pa razvijaoci koji koriste agilne metode cesto podlezu refaktorisanju kako bi izbegli ovo degradiranje. Naglasak na cestom testiranju kod agilnih metoda smanjuje rizik od pojave novih gresaka prilikom refaktorisanja. Svaka greska koja se pojavi treba da bude detektovana tako sto ce test koji izvrsen pre izmene I bi uspesan sada pokazati gresku. Refaktorisanje ne zavisi od drugih agilnih metoda I moze se primeniti bilo gde.

Primeri koda “koji smrdi” a koji se moze unaprediti refaktorisanjem:

1. *Ponavljajuci kod* – veoma slican kod moze se pojaviti na vise mesta u programu. Ovo se moze izbeci tako sto ce se taj deo koda izdvojiti u poseban metod ili funkciju.
2. *Dugacki metodi* – ukoliko je metod suvise dugacak treba se rekonstruisati u vise manjih metoda.
3. *Switch-case naredba* - moze izazvati razna ponavljanja koda, zbog toga se kod objektno-orijentisanih jezika moze zameniti uz pomoc koriscenja polimorfizma.
4. *Lanci poruka (data clumping)* – lanci poruka se javljaju kada iste stavke podataka (polja u klasama, parametri u metodama) pojavljuju na nekoliko mesta u programu. Ovo se cesto moze resiti enkapsulacijom podataka.
5. *Spekulativno uopstavanje* – javlja se kada programeri koriste generalizaciju u kodu u slucaju da ce to biti potrebno u buducnosti. Ovo se uglavnom moze lako ukloniti.

Refaktorisanje se sprovodi tokom razvoja programa I efikasan je nacin da se smanje dugorocni troskovi odrzavanja programa. Ako uzmemo neki program da odrzavamo cija je struktura znacajno degradirana tada moze da dodje to toga da je refaktorisanje prakticno nemoguce izvesti. Refaktorisanje dizajna je dosta skuplji metod.

**Upravljanje zastarelim sistemom**

Za novo razvijene sisteme za koje su koriscene moderne tehnologije inzenjerstva, kao sto je inkrementalni razvoj moguce je razmsljati o tome kako integrisati razvoj sistema I evoluciju. Mnoge kompanije pocinju da razumeju da je razvoj sistema ceo proces I da razdvajanje razvoja I odrzavanja nije isplativ. Kako god, I dalje postoji mnogo zastarelih sistema koji predstavljaju kriticnu tacku u poslovanju.

Mnoge firme imaju jos uvek zastarele sisteme za koje imaju ogranicene budzete za odrzavanje I unapredjivanje sistema. Oni odlucuju kako da na najbolji nacin vrate ono sto su ulozili. Ovo ukljucuje pravljenje realisticne procene njihovih zastarelih sistema I onda smisliti najbolju strategiju za unapredjenje ovih sistema. Postoje 4 strategijske mogucnosti:

1. *Unistiti sistem kompletno* – ovu opciju treba izabrati kada sistem ne daje odgovarajuci doprinos poslovnom procesu. Ovo se obicno javlja kada se izmeni poslovni proces I kada sistem vise nije toliko upotrebljiv.
2. *Ostaviti sistem neizmenjen I krenuti sa regularnim odrzavanjem* – ova opcija se bira kada je sistem jos uvek odgovarajuc, a korisnici imaju samo nekoliko zahteva za promenama.
3. *Rekonstruisanje sistema kako bi se unapredilo njegovo odrzavanje* – bira se kada je kvalitet sistema degradiran nekom promenom, a kada je jos uvek zatrazena ta promena. Ovaj proces moze ukljuciti razvijanje novih komponenti interfejsa tako da stari sistem moze raditi sa novim sistemima.
4. *Zameniti deo ili ceo stari sistem novim sistemom* – ova opcija se bira kada npr. stari sistem ne moze da funkcionise kako treba na novom hardveru ili ako je moguce razviti novi sistem po nekoj razumnoj ceni.

Naravno, ove opcije nisu iskljucive. Kada je sistem sastavljen od vise programa, na svaki od njih se moze primeniti druga opcija.

Kada procenjujemo stari sistem moramo ga posmatrati iz poslovne I tehnicke perspektive. Iz poslovne perspektive da li je tom poslu zaista potreban sistem. Iz tehnicke perspektive moramo da procenimo kvalitet aplikacionog softvera I sistemsku podrsku softvera I hardvera. Tada kombinujemo poslovne vrednosti I kvalitet sistema kako bi lakse odlucili sta da radimo sa zastarelim sistemom.

1. *Los sistem, niska poslovna vrednost* – kod ovakvih sistema se najvise isplati unisititi ih I praviti nov
2. *Los kvalitet, visoka poslovna vrednost* – s obzirom na visoku poslovnu vrednost ne mozemo ih unistiti, ali zbog loseg kvaliteta ce odrzavanje biti skuplje. Ove sisteme treba rekonstruisati kako bi se poboljsao njihov kvalitet.
3. *Visok kvalitet, niska poslovna vrednost* – ovi sistemi ne znace mnogo u poslu, ali zbog kvaliteta nisu skupi za odrzavanje. Ukoliko nisu potrebne neke skupe promene, steta je zameniti ih, vise se isplati rekonstruisanje. Ukoliko su potrebne promene vrlo skupe, najbolje je unistiti ih.
4. *Visok kvalitet, visoka poslovna vrednost* – ove sisteme treba zadrzati. Najbolje je nastaviti odrzavanje sistema.

Kako bi procenili poslovnu vrednost sistema neophodno je odrediti aktere sistema, kao sto su krajnji korisnici I njihovi menadzeri I postaviti im seriju pitanja o sistemu. Postoje 4 osnovne stvari o kojima treba prodiskutovati:

1. *Koliko je sistem koristan* – ako se sistem koristi samo povremeno ili ako ga koristi mali broj ljudi tada verovatno ima nisku poslovnu vrednost. Stari sistemi su mozda napravljeni da zadovolje potrebe poslovanja a koje je promenjeno ili se taj softver moze vise iskoristiti u nekim drugim situacijama. Treba biti obazriv sa sistemima koji se koriste povremeno jer mozda imaju bitnu *ulogu.*
2. *Koje poslovne procese softver podrzava* – kada je sistem uveden, poslovni procesi su napravljeni tako da iskoriste mogucnosti sistema. Ako je sistem nefleksibilan, promena ovih poslovnih procesa moze biti nemoguca. Tako moze doci do toga da poslovni procesi postanu zastareli. Dakle, sistemi mogu imati nisku poslovnu vrednost jer se primorava upotreba neefikasnih poslovnih procesa.
3. *Pouzdanost sistema* – pouzdanost sistema nije samo tehnicki problem vec I poslovni. Ukoliko je sistem nepouzdan I ako problemi direktno uticu na musterije ili ako se salju zaposleni sa drugih radnih mesta kako bi resili neki problem tada sistem ima losu poslovnu vrednost.
4. *Rezultati rada sistema* – ovde je veoma bitno to da rezultati rada sistema doprinose uspesnom funkcionisanju poslova. Ako poslovi zavise od ovih rezultata onda sistem ima visoku poslovnu vrednost. Ali ako se ovi rezultati mogu postici na neki drugi laksi nacin ili ako se retko koriste onda poslovna vrednost moze postati losa.

Kako bi procenili softverski sistem iz tehnicke perspektive tada treba da razmotrimo obe stvari I aplikativni sistem posebno I okruzenje u kojem sistem radi. Okruzenje podrazumeva hardver I ukljucujuci softver, broj gresaka u radu hardvera koje se javljaju cesce nego sto je predvidjeno za neki vremenski period I ucestalost popravki gresaka koje se javljaju na softveru. Takodje treba proveriti I pouzdanost proizvodjaca hardvera I njegove softverske podrske. Ako ovi proizvodjaci vec duze vreme nisu u poslu mozda nisu dobra podrska za sistem.

Kako bi procenili tehnicki kvalitet aplikativnog sistema moramo prouciti niz faktora koji se prvenstveno odnose na pouzdanost, kompleksnost odrzavanja sistema kao I na njegovu dokumentaciju. Podaci koji mogu biti korisni u proceni kvaliteta su:

1. *Broj zahteva za izmenu sistema* – sistemske promene cesto narusavaju sistemsku strukturu I kasnije promene cine jos komplikovanijim. Sto je ovaj broj veci, to je kvalitet sistema losiji.
2. *Broj korisnickih interfejsa* – sto je vise interfejsa veca je verovatnoca da ce biti vise nedoslednosti I nepotrebnih delova u ovim interfejsima.
3. *Kolicina podataka koje koristi sistem* – sto je vise podataka veca je verovatnoca da ce biti vise nedoslednosti koje smanjuju kvalitet sistema.

Objektivna procena treba da se koristi za odlucivanje sta uraditi sa zastarelim sistemom. U dosta slucajeva, odluka cesto nije objektivna vec je cesto zasnovana na organizacijskim I politickim razmatranjima.

*Sanja Petrović 1110/2012*