**Visoka Tehnička Škola Strukovnih Studija**

# Održavanje softvera

Student:
**Maja Tolj**

Profesor:
**Janoš Šimon**

**Decembar,2020., Subotica**

# Održavanje softvera

## Uopšteno o održavanju

Nakon puštanja u upotrebu, softverski sistem se i dalje mora menjati. Promene softvera neophodne su da bi se držao korak s novim korisničkim zahtevima, promenama poslovnog okruženja, napretkom hardvera itd.

Proces menjanja softvera nazivamo održavanje odnosno evolucija. Za neke autore ove dve reči su sinonimi, s time da je prva tradicionalna i pozajmljena iz tehnike, a druga je novija i tačnija. Pod održavanjem podrazumevamo planirani rutinski dio procesa koji se sastoji od manje radikalnih promena, dok evolucija označava ukupni proces s nepredvidivim elementima koji može dovesti do bitnih promena u arhitekturi softvera i korištenoj tehnologiji.

## Strategije menjanja softvera

Prema autoru Warren-u (1998), postoje sledeće tri strategije koje se međusobno dopunjuju:

- Održavanje softvera

Promene se rade zato da bi se držao korak s (promenjenim) zahtevima. Menja se funkcionalnost, no struktura softvera uglavnom ostaje ista.

- Arhitekturna transformacija

Značajna promena arhitekture softverskog sastava. Na primer, sastav se iz centralizirane mainframe arhitekture prebacuje u distribuiranu arhitekturu s klijen- tima i poslužiteljima. Funkcionalnost se može ili ne mora promeniti.

- Softversko re-inženjerstvo

Ne dodaje se nikakva nova funkcionalnost. Takođe, nema velike promene u arhitekturi. Umesto toga, sastav se transformiše u oblik koji se lakše može razumeti i dokumentovati, te obrađivati raspoloživim alatima. Na primer, programski kod se automatski prebacuje iz jednog (zastarelog) programskog jezika u drugi (savremeni).

Održavanje softvera je kontinuirani rutinski proces koji se svodi na niz uzastopnih promena. Kao rezultat održavanja nastaje velik broj različitih verzija softvera, od kojih svaka predstavlja nešto drugačiju konfiguraciju sastavnih delova. Celi proces treba pažljivo planirati i kontrolisati. Takođe je neophodno imati tačnu evidenciju svih verzija i njihovih konfiguracija. Skup potrebnih metoda, alata i organizacijskih zahvata naziva se upravljanje konfiguracijom.

Arhitekturna transformacija i softversko re-inženjerstvo su jednokratni i radikalni zahvati kojima se pribegava onda kad održavanje više ne daje rezultata ili postane preskupo. Ti zahvati obično se primenjuju na stari "baštinjeni" (engleski legacy ) softver, zato da bi se takav softver osavremenio te učinio pogodnijim za daljnje korištenje i mijenjanje. Nakon arhitekturne transformacije ili re-inženjerstva, transformirani softver dalje se podvrgava održavanju.

## Vrste održavanja softvera

Održavanje se sastoji od menjanja postojećih delova softvera, te po potrebi od dodavanja novih delova, bez bitnog narušavanja postojeće arhitekture. S obzirom na sadržaj i karakter promena razlikujemo sledeće vrste održavanja.

- Korekcijsko održavanje

Ispravljaju se uoštene greške. Može se raditi o greškama u kodiranju (jeftino za ispraviti), u oblikovanju (skuplje), odnosno u specifikaciji (najskuplje).

- Adaptacijsko održavanje

Obavlja se prilagođavanje na novu platformu, na primer drugi hardver, drugi operacijski sastav, ili drugu biblioteku potprograma. Funkcionalnost softvera nastoji se održati nepromenjenom.

- Perfekcijsko održavanje

Implementiraju se novi funkcionalni ili ne-funkcionalni zahtevi. Ti zahtevi odražavaju povećane potrebe korisnika ili promene u poslovnom okruženju.

Pojedina nova verzija softvera obično kombinuje više vrsta promena, no na management-u je da odluči kojoj vrsti održavanja će dati prednost.

## Dinamika održavanja softvera

Autori Lehman i Belady (1985) objavili su empirijska istraživanja dinamike održavanja softvera. Istraživanja sugerišu da vrede sledeći, takozvani Lehman-ovi "zakoni".

1. Nužnost menjanja

Softver koji se zaista koristi u stvarnom svetu nužno se mora menjati jer u protivnom ubrzo postaje neupotrebljiv.

2. Povećavanje složenosti

Dok se softver menja, njegova struktura teži tome da postane sve složenija. Da bi se očuvala jednostavnost strukture, potrebno je uložiti dodatni trud i resurse.

3. Ograničena brzina unapredivanja

Količina "novosti" koju pojedino izdanje softvera može doneti otprilike je konstantna i karakteristična je za taj softver.

Prvi i drugi Lehman-ov zakon dovoljno su razumljivi bez dodatnih komentara. Treći zakon znači da ako u jednom izdanju unesemo previše novosti, unećemo i mnogo novih grešaka, pa će iduće izdanje morati biti posvećeno ispravljanju tih grešaka, a prosečna količina novosti po izdanju vratiće se na normalnu vrednost.

Konstanta iz trećeg zakona određena je originalnim procesom razvoja softvera i ne može se naknadno popraviti. Znači, ako je proces razvoja bio kvalitetniji, stvoriće se softver koji je pogodniji za održavanje, te će konstanta biti veća, to jest dopuštaće veću brzinu kasnijeg unapređivanja softvera.

## Cena održavanja softvera

Ukupna cijena održavanja softvera dostiže i prelazi cenu njegovog originalnog razvoja. Cenu održavanja možemo smanjiti opet tako da povećamo kvalitet originalnog razvojnog procesa. Uz veći kvalitet cena razvoja će biti veća, ali će se stvoriti softver koji je jeftiniji za kasnije održavanje.

1. Celovitost polazne specifikacije.

Ukoliko odmah uključimo sve zahteve, kasnije će biti manje perfekcijskog održavanja.

2. Dobrota oblikovanja

Dobar design je jeftiniji za održavanje. Smatra se da su sa stanovišta održavanja najbolji objektno-oblikovani sastavi, koji se sastoje od malih modula s jakom unutrašnjom kohezijom i labavim vezama prema van.

3. Način implementacije

Kod u "strožem" programskom jeziku poput Jave lakše se održava nego kod u jeziku poput C-a. Strukturirani kod (if, while) sa smisleno imenovanim varijablama razumljiviji je od kompaktnog koda s mnogo goto naredbi.

4. Stepen verifikovanosti

Dobro verifikovan softver ima manje grešaka pa će zahtevati manje korekcijskog održavanja.

5. Stepen dokumentovanosti

Uredna, dobro strukturisana i celovita dokumentacija olakšava razumevanje softvera, te na taj način pojeftinjuje održavanje.

6. Način upravljanja konfiguracijom

Ukoliko se primjenjunju metode, alati i organizacijska pravila upravljanja konfiguracijom, tada je održavanje na dugi rok jeftinije.

7. Starost softvera

Što je softver stariji, to je skuplji za održavanje, budući da mu se grada degradirala, ovisan je o zastarelim razvojnim alatima, a dokumentacija mu je postala neažurna.

8. Svojstva aplikacijske domene

Ako je reč o stabilnom domenu gde se poslovna pravila retko menjaju, tada će se retko pojavljivati potreba za perfekcijskim održavanjem u svrhu usklađivanja s novim pravilima.

9. Stabilnost razvojnog tima

Održavanje je jeftinije ako se njime bave originalni razvijaći softvera, jer oni ne moraju trošiti vreme na upoznavanje sa softverom.

10. Stabilnost platforme

Ako smo softver implementirali na platformi koja će još dugo biti savremena, tada neće trebati adaptacijsko održavanje.Na prvih šest faktora moguće je utecati tekom originalnog razvojnog procesa

## Upravljanje konfiguracijom

Veliki softverski sustav može se promatrati kao konfiguracija svojih sastavnih delova. Za vreme svog života, sustav se menja. Nastaju različite verzije, od kojih svaka predstavlja nešto drukčiju konfiguraciju delova .

Upravljanje konfiguracijom (engleski configuration management ) je organizirani proces koji kontrolira menjanje softvera i evidentira njegove različite verzije. Proces uključuje sljedeće četiri aktivnosti:

- izrada plana upravljanja konfiguracijom,
- upravljanje promenama sustava,
- upravljanje verzijama sustava,
- gradnja sustava.

## Izrada plana upravljanja konfiguracijom

Plan upravljanja konfiguracijom opisuje standarde i postupke koji se trebaju koristiti za upravljanje konfiguracijom. Reč je o dokumentu koji sadrži sledeća poglavlja.

- Popis entiteta koji će biti obuhvaćeni upravljanjem. Šema za identifikaciju tih entiteta.
- Odluka o tome ko je odgovoran za pojedine procese unutar upravljanja.
- Opis postupaka koji će se koristiti za upravljanje promenama i upravljanje verzijama.
- Opis dokumenata koji će se generisati i pohranjivati tekom upravljanja.
- Opis alata koji će se koristiti za upravljanje.
- Opis baze podataka za pohranjivanje informacija o konfiguracijama.

Šema za identifikaciju entiteta daje jedinstveno ime svakom dokumentu, modulu, . . . , itd. koji je obuhvaćen upravljanjem konfiguracijom. Obično se koristi hijerarhijska šema. U tom prilogu reč je o paketu PCL-TOOLS koji se sastoji od četiri programa: COMPILE, BIND, EDIT i MAKE GEN. Svaki program dalje se sastoji od raznih modula, a za svaki modul na najnižem nivou pamtimo opis (OBJECT), programski kod (CODE) i skup testova (TESTS).

Problem s ovakvom šemom je da isti entitet dobiva sasvim drugo ime ako se upotrebi unutar drugog sistema. Takođe, otežano je pretraživanje po kriterijima koji su drukčiji od onih primenjenih za klasifikaciju.

Baza podataka o konfiguracijama pohranjuje informacije koje su potrebne da bi se odredile konfiguracije pojedinih vezija softvera, te takođe i druge relevantne informacije. Baza mora biti u stanju dati odgovore na primer na sledeće upite.

- Koliko verzija sistema postoji i koji su datumi njihovog nastanka?
- Od kojih verzija kojih delova se sastoji zadana vezija sistema?
- Koji kupci su instalirali zadanu verziju sistema?
- Koji hardver, operativni sistem i biblioteke su potrebni za pokretanje zadane verzije sistema?
- Koji su nerešeni zahtevi za promenom zadane verzije sistema?
- Koje su greške prijavljene za zadanu verziju sistema?

Baza služi kao podrška za upravljanje promenama, upravljanje verzijama, te građenje sistema.

## Upravljanje promenama sistema

Upravljanje promenama treba osigurati da se proces menjanja (održavanja) softvera drži pod kon- trolom, da se promene odvijaju na racionalan način, te da se sve promene evidentiraju i dokumentuju. Jedna nova verzija sistema obično sadrži veći broj promena .

Zahtev za promenom upisuje se u propisani obrazac. Analizu posledica promene obavlja imenovani analitičar. Odluku o tome koje promene će se usvojiti i uključiti u novu verziju sustava donosi "change control board". Implementiranje skupa odobrenih promena te ponovno testiranje sistema obavlja programerski tim za održavanje. Za svaki modul ili funkciju evidentira se "istorija" promena - obično preko uvodnog komentara.

## Upravljanje verzijama sistema

Upravljanje verzijama omogućuje identifikovanje i evidentiranje različitih verzija i izdanja istog sistema. Verzija je primerak sistema koji se po nečemu razlikuje od ostalih primeraka. Izdanje je verzija koja se isporučuje kupcima. Osim samih izvršivih programa, izdanje takođe sadrži konfiguracijske datoteke, datoteke s podacima, program za instalaciju, elektronički ili papirnati priručnik za korisnike, itd. Izdanje se distribuira na odgovarajućim medijima (CD, DVD, download s Interneta, . . . ).

Prvo što upravljanje verzijama mora osigurati je jednoznačna identifikacija svih verzija i njihovih sastavnih delova. Uobičajeni način identifikacije pomoću brojeva, na primer:

verzija 1.0, 1.1, 1.2, . . . , 2.0, 2.1, . . . .

Druga mogućnost je pomoću atributa, na primer:

program AC3D (jezik=Java, platforma=WinXP, datum=Jan2003).

Brojčani način identifikacije je češći, no on u sebi krije probleme jer implicira linearni redosled stvaranja verzija. Pravi redosled može biti komplikovaniji. Atributni način identifikacije je pogodniji za pretraživanje i on je obično implementiran "iznad" brojčane identifikacije, to jest baza podataka o konfiguracijama ˇcuva veze između atributa koji opisuju verzije i brojeva odgovarajućih verzija sistema i njihovih delova.

Nakon identifikacije, upravljanje verzijama treba osigurati da se različite verzije sistema mogu reprodukovati kad je to potrebno, te da se te verzije neće nepažnjom promeniti. To ne mora značiti da svaka verzija mora biti eksplicitno pohranjena, već na primer da postoji postupak kojim se izvorni kod tražene verzije može proizvesti iz neke pohranjene verzije.

Upravljanje verzijama takođe uključuje donošenje odluke koju od verzija treba pretvoriti u izdanje. Reč je o osetljivoj poslovnoj odluci. Naime, ako su izdanja prečesta, korisnici ih neće prihvatiti jer im instalacija iziskuje troškove. Ako su izdanja preretka, korisnici bi mogli odustati od našeg softvera i preći na alternativna rešenja. Lehman-ov treći zakon sugeriše da nakon izdanja s novim mogućnostima (perfekcijsko održavanje) obično treba slediti jedno ili dva izdanja kojima se poprav- ljaju grečke (korekcijsko održavanje).

## Izgradnja sistema

Reč je o postupku prevođenja i povezivanja delova sistema u sistem koji će se izvršavati na određenoj ciljnoj platformi. Stvari na koje treba obratiti pažnju kod gradnje su sledeće.

Da li su u instrukcije za gradnju uključeni svi potrebni delovi softvera (moduli, headeri, bib- lioteke, . . . )?

- Da li instrukcije za gradnju uključuju pravu verziju svakog dela softvera?
- Da li su dostupne sve potrebne datoteke s podacima?

Ukoliko modul referencira datoteku, da li je upotrebljeno ime datoteke unutar modula isto kao ime na ciljnom stroju?

- Da li je još uvek dostupna prava verzija compilera i ostalih alata?

## CASE-alati za upravljanje konfiguracijom

Prva generacija alata davala je podršku samo za pojedine aktivnosti unutar upravljanja konfiguracijom. Primeri alata prve generacije su: sccs (Rochkind, 1975) i rcs (Tichy, 1985) za upravljanje verzijama, te make (Feldman, 1979) za izgradnju sistema.

Alati druge generacije kao na primer Lifespan (Whitgift, 1991) i DSEE (Leblang and Chase, 1987) daju podršku za više aktivnosti, no još uvek ne obuhvataju celi proces upravljanja konfiguracijom. Postoje i sasvim integrisani alati (Leblang, 1994), no oni su obično složeni i skupi, tako da održavatelji softvera ipak još uvek koriste prvu i drugu generaciju. Microsoft-ov Visual Studio takođe daje prilično sveobuhvatnu podršku za upravljanje konfiguracijom, no isključivo za softver pisan za Windows platformu.

Podrška za upravljanje promenama. Alati za upravljanje promenama obično su integrisani sa sistemom za upravljanje verzijama i sadrže sledeće elemente.

- Editor formulara kojim se može menjati izgled zahteva za promenama.
- Workflow sistem koji omogućuje da se definišu ljudi koji će obraditi zahtev za promenu, te omogućuje da se definiše redosled obrade - sistem šalje formulare i obaveštenja pravim ljudima u pravo vreme elektronskom poštom;
- Baza podataka koja čuva sve zahteve za promenama i koja se može povezati sa sistemom za upravljanje verzijama.
- Podrška za upravljanje verzijama - Alati za upravljanje verzijama pohranjuju razne verzije sistema i njihovih sastavnih delova. Uobičajene mogućnosti alata su sledeće:

- Identifikacija verzija (i izdanja) - Svakoj novoj verziji automatski se dodeljuje identifikator, te eventualno dodatni atributi za pretraživanje.
- Kontrolirano menjanje - Da bi bio menjan, deo softvera mora eksplicitno biti "izvađen" iz repozitorija te premešten u neki radni direktorij. Kad se deo vrati u repozitorij, stvara se njegova nova verzija, a stara verzija i dalje postoji.
- Efikasno pohranjivanje - Budući da se verzije velikim delom podudaraju, pohranjuje se zapravo samo jedna "master" verzija, a ostale se opisuju tako da se zapišu razlike u odnosu na master.
- Pamćenje istorije promena - Sve promene za zadani sistem ili njegov deo se pamte, te se mogu izlistati.
- Nezavisan razvoj - Različite verzije sistema mogu se razvijati paralelno, te se svaka verzija može menjati nezavisno o drugim verzijama. Alat rešava konflikte koji mogu nastati kad više ljudi pokuša menjati isti deo.

Jednostavan primjer alata prve generacije za upravljanje verzijama na UNIX platformama je već spomenuti rcs (Revision Control System). Unutar rcs-a, izvorni kod zadnje verzije sprema se kao master, a ostale verzije definiraju se kao razlike (takozvane "delte") u odnosu na zadnju verziju. Podržani su samo ASCII masteri, a delte se opisuju kao skupovi komandi za editovanje na primer u vi editoru. Dopušteni su i složeni redoslijedi verzija.

Podrška za gradnju sistema. Alat za gradnju sistema automatizuje postupak gradnje da bi smanjio mogućnost ljudske greške. Takođe, alat nastoji smanjiti posao tako da, na primer, izbegne nepotrebna kompiliranja ili povezivanja. Alat može biti samostalan ili može biti integriran s alatom za upravljanje verzijama. Poželjne su sledeće mogućnosti.

Jezik za definiranje zavisnosti i pripadni interpreter. Jezik omogućuje definiranje ovisnosti izmedu različitih delova softvera, ili bolje rečeno zavisnosti izmedu različitih datoteka od kojih se gradi sistem.

Podrčka za izbor i pokretanje drugih alata. Omogućuje se specificiranje compilera, linkera i drugih alata koji se koriste za obradu datoteki. Takođe se omogućuje pokretanje izabranih compilera ili linkera uz zadavanje opcija.

Distribuirano kompiliranje. Neki alati za izgradnju sistema u stanju su istovremeno pokrenuti kompiliranje različitih delova softvera na različitim umreženim računarima. Time se drastično smanjuje vreme potrebno za izgradnju sistema.

Upravljanje s izvedenim datotekama. Izvedene datoteke su one koje su stvorene iz drugih (izvornih) datoteka primenom odgovarajućih alata. Na primer, object-datoteka se izvodi iz datoteke s izvornim kodom primenom compilera. Upravljanje izvedenim datotekama omogućuje da se izvedena datoteka ponovo stvara jedino ako je to zaista potrebno zbog promena u izvornoj datoteki.

Jednostavni primer alata prve generacije za gradnju sustava na UNIX platformama je make. On povezuje program koji se sastoji od viˇse modula, s time da prethodno re-kompilira samo one izvorne datoteke koje su bile mijenjane nakon svog zadnjeg kompiliranja. Korisnik piˇse skriptu (takozvani "makefile") koji opisuje veze izmedu datoteka. Ažurnost pojedine datoteke utvrđuje se upoređivanjem datuma i vremena zadnje promene.

Glavna manjkavost make-a je ta što on nije povezan s odgovarajućim alatom za upravljanje verzijama, na primer s rcs. Drugim rečima, make nije svestan postojanja različitih verzija istog programa, već se korisnik sam mora brinuti da mu "podmetne" pravu verziju.

## Baštinjeni softver i njegovo menjanje

U velikim organizacijama postoje takozvani baštinjeni (engleski *legacy*) sistemi. Reč je o softveru razvijenom tekom 70-tih ili 80-tih godina 20. veka koji je u tehnološkom smislu zastareo. Održavanje baštinjenih sistema izuzetno je skupo, koji put i nemoguće. S druge strane, organizacija se ne može tek tako odreći tog softvera, budući da je on neophodan za njeno svakodnevno funkcioniranje. Izlaz iz ove situacije nastoji se naći u jednokratnim radikalnim zahvatima kao što su softversko re-inženjerstvo odnosno arhitekturna transformacija. Tim zahvatima baštinjeni softver se naprije menja u oblik koji se lakše može razumeti, dokumentirati i obradivati savremenim alatima. Nakon toga, softver se nastoji osavremeniti i učiniti pogodnijim za daljnje korištenje i održavanje.

## Grada i tehnološke osobine baštinjenog softvera

Baštinjeni sustav obično je originalno bio razvijen pre 20-30 godina, te je kasnije bio menjan. U originalnom razvoju koristila se neka od funkcionalno-orijentiranih metoda za razvoj softvera. Zato imamo sledeće osobine

Sistem se sastoji od više programa i zajedničkih podataka koje koriste ti programi. Podaci se čuvaju u datotekama ili pomoću (zastarelog) sistema za upravljanje bazom podataka.

Pojedini program unutar sistema oblikovan je funkcionalnim pristupom i sastoji se od funkcija koje komuniciraju preko parametara i globalnih varijabli.

U slučaju poslovnih primena, sistem se obično svodi na “batch” obradu ili obradu transakcija. U oba slučaja, opšta organizacija je s skladu s modelom “ulaz-obrada-izlaz”.

Sistem je implementiran u zastarelom programskom jeziku kojeg današnji programeri ne razumeju, te za kojeg više nema compilera.

Dokumentacija sistema je delom izgubljena ili neažurna. U nekim slučajevima jedina dokumentacija je izvorni programski kod. Ima slučajeva kad je čak i izvorni kod izgubljen te jedino postoji izvršiva verzija programa.

Dugogodišnje održavanje pokvarilo je strukturu sistema, tako da ju je vrlo teško razumeti. Izvorni kod pisan je s namerom da se optimiziraju performanse ili smanje zahtevi za memorijom, a ne s namjerom da kod bude razumljiv. To stvara poteškoće današnjim programerima koji su učili savremene tehnike softverskog inženjerstva i nikad nisu videli takav stil programiranja.

## Softversko re-inženjerstvo

Cilj softverskog re-inženjerstva je da se poboljša struktura sistema, te da se ona dovede u oblik koji se može lakše razumeti. Očekuje se da će se na taj način smanjiti cena daljnjeg održavanja sistema. Re-inženjerstvo uključuje sledeće pod-aktivnosti:

Prevodenje izvornog koda - Automatska (uglavnom) konverzija programa pisanog u jednom (za- starelom) programskom jeziku u drugi (savremeni) jezik. Ovaj zahvat je nužan ukoliko za

zastareli jezik više nema compilera ili ako ga mladi programeri ne razumiju.

Reverzno inženjerstvo - To je proces reproduciranja designa i specifikacije sustava na osnovu njegovog programskog koda. Potrebno je ukoliko je originalna dokumentacija izgubljena. Koriste se pomoćni alati, takozvani "program-browseri" kojima se obavlja "navigacija" kroz kod, na primer pronalaženje mesta gde se koristi određena varijabla.

Poboljšanje strukture programa - Uključuje na primer zamjenu nestrukturiranih kontrolnih kon- strukcija poput gotos ekvivalentnim whilepetljama i ifnaredbama. Postupak se može autom- atizirati. Nakon ovog zahvata, kod postaje uredniji i pogodniji za daljnje održavanje.

Modularizacija programa - Svodi se na reorganizaciju izvornog koda, tako da se delovi koji su u međsobnōj vezi grupiraju zajedno. Nakon toga ti dijelovi postaju razumljiviji, pa ih je lakše optimizirati, ili je lakše ukloniti redundanciju.

Podatkovno re-inženjerstvo - Potrebno je ukoliko programi na nekonzistentan ili zastareli način upravljaju podacima. Uključuje na primer modifikaciju svih programa tako da umesto vlastitih datoteka koriste zajedničku bazu podataka, ili na primer prelazak sa zastarele (hijerarhijske) baze na savremenu (relacijsku) bazu.

Kao primer poboljšanja strukture programa - Reč je o nestrukturisanoj verziji kontrolera sistema za grejanje. Kod je gotovo nemoguće razumeti zbog brojnih gotonaredbi - takozvana špageti logika. Nakon re-inženjerskog zahvata, dobili smo znatno razumljiviju verziju. Ovakav zahvat moguće je obaviti uz pomoć automatskog alata, koji najprije konstruiše dijagram toka kontrole originalne "goto" verzije, a zatim realizuje taj dijagram uz pomoć ifi while naredbi.

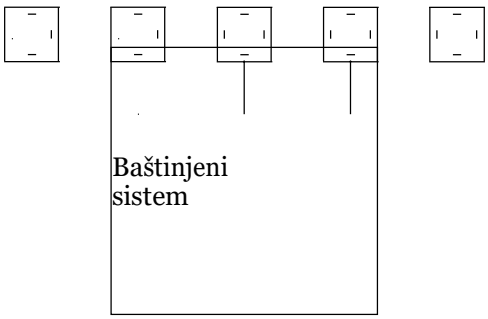
Programi u polaznom sistemu koristili su klasične datoteke u kojima je postojala redundancija i nekonzistentnost, i od kojih je svaka bila optimizovana za jednu aplikaciju a nepogodna za druge. Nakon re-inženjerskog zahvata imamo integrisanu i konzistentnu bazu podataka koja osim postojećih aplikacija može podjednako dobro podržati i razne buduće aplikacije. Za ovakav zahvat potrebno je stvoriti logički model podataka, prepraviti sve programe tako da pristupaju bazi umesto datotekama, te prebaciti podatke iz datoteki u bazu.

Softversko re-inženjerstvo ima svoja ograničenja. Na primer, sistem razvijen funkcionalnim pristupom nije moguće pretvoriti u objektni sistem. Nakon re-inženjerstva softver je pogodniji za održavanje, no ipak se ne može tako dobro održavati kao novi softver. Podatkovno re-inženjerstvo može biti izuzetno skupo. Management treba proceniti da li se re-inženjerstvo uopšte isplati, ili je bolje krenuti u razvoj sasvim novog sistema.

## Arhitektonska transformacija

Nakon re-inženjerstva, baštinjeni softver obično se podvrgava arhitekturnoj transformaciji, te se na taj način nastoji prilagoditi sužavremenim tehnologijama. Tipični baštinjeni system je centralizovani sistem za rad na mainframe računaru i tekstualnim terminalima, prikazan na slici.

(terminali)




(mainframe)

Polazna arhitektura baštinjenog sistem.

Najčešći cilj arhitekturne transformacije je prelazak na arhitekturu klijent-poslužitelj, tako da se trans- formirani sistem može instalirati na mrežu sastavljenu od osobnih i poslužiteljskih računara.

Prelazak na arhitekturu klijent-poslužitelj je poželjan iz sledećih razloga:

Cena kupovine i održavanja hardvera - Distribuisana računarska mreža znatno je jeftinija od mainframe računara ekvivalentne snage.

Očekivanja u pogledu korisnikog interfejsa - Tradicionalni mainframe sistemi daju interfejs s tekstualnim formularima. No korisnici danas očekuju grafički interfejs i laganu interakciju sa sistemom. Grafički interfejs zahteva znatno više lokalnog procesiranja i može se efikasno implementirati jedino na klijentima kao što su lični računari ili grafičke radne stanice.

Distribuirani pristup sistemu - Organizacije su sve više fizički distribuisane, pa računarski sistemi moraju biti dostupni s različitih vrsta opreme.

Kod planiranja arhitektonske transformacije, korisno je zamišljati da se naš polazni baštinjeni sistem sastoji od slojeva. Pojedini slojevi obavljaju sledeće zadatke:

- *Prezentacijski sloj* bavi se prikazom i organizacijom formulara za unos ili prikaz podataka.
- *Sloj za validaciju podataka* proverava ispravnost podataka koje korisnik unosi u formular.
- *Sloj za kontrolu interakcije* upravlja redosledom korisnikovih operacija, te redosledom formulara koji se prikazuju korisniku.
- *Sloj s aplikacijskim servisima* obavlja obradu podataka i računanje.
- *Sloj s bazom podataka* omogućuje pristup podacima koji su pohranjeni u bazi.

Postoje razni načini distribuisanja sistema, koji se razlikuju po načinu kako su slojevi raspoređeni izmedu poslužitelja i klijenata. Slojevi koji su prebačeni na klijenta u pravilu se moraju menjati jer u protivnom nećemo moći iskoristiti prednosti grafičkog interfejsa. Zato distribucije gde je sve veći broj slojeva prebačen na klijenta postaju sve skuplje. Najjeftinije rešenje prebacuje samo prezentaciju na klijenta, no time ne dobijamo mnogo jer se klijent tada ponaša kao emulator terminala, pa interfejs i dalje sledi logiku tekstualnih formulara. Najskuplje rešenje je da se sve osim baze prebaci na klijenta - tada se poslužitelj svodi na poslužitelja baze podataka i možda se može zameniti komercijalnim DBMS-om, čime bi se celi baštinjeni sistem izbacio iz upotrebe. Srednje rešenje je da na klijentu budu prezentacija,

validacija i kontrola interakcije - ta tri sloja zajedno mogu se na najbolji način zameniti s novim grafičkim interfejsom koji više neće slediti logiku tekstualnih formulara.

Opisani načini distribuisanja sistema pretpostavljali su da polazni sistem ima jasno podeljenje slojeve. U praksi to često nije slučaj, to jest slojeve nije moguće izdvojiti budući da su njihove funkcije pomešane u raznim modulima. Tada se pribegava nešto drugačijem vidu arhitekturne transformacije. Transformirana arhitektura tada se sastoji od sledeća tri dela.

- Baštinjeni sistem, "zamrznut" u svom originalnom obliku, instaliran na poslužiteljskom računaru.

- Iznova razvijeno (grafičko) korisnički interfejs, koje radi kao klijent na osobnom računalu. Posebno razvijeni "wrapper", koji prevodi zahteve klijenata u interakcije s nepromenjenim baštinjenim sistemom. Iz perspektive klijenta, wrapper izgleda kao poslužitelj. Iz perspektive baštinjenog sistema, wrapper izgleda kao skup tekstualnih teminala.

Spomenuta arhitektura s wrapperom opet dozvoljava razne varijante, koje se razlikuju po tehnologiji upotrebljenoj na strani klijenta. Neke od varijanti su sledeće.

Klijent je off-the-shelf emulator terminala. Wrapper je tada nepotreban ili vrlo jednostavan. Ovo je svakako najjeftinije no i najmanje kvalitetno rešenje. Naime, interfejs je zadržao tekstualni oblik, makar je uklopljeno u prozor unutar grafičkog interfejsa.

Klijent je standardni web browser poput Microsoft Internet Explorer. Wrapper je tada web server poput Apache, nadopunjen servletima ili CGI-bin skriptama koje komuniciraju s baštinjenim sistemom i dinamički generišu web stranice za klijenta. Ovo je prilično jeftino rešenje gde postoji velika nezavisnost o klijentskoj platformi. Ipak, u skladu s mogućnostima web browsera, grafički interfejs je relativno skroman.

Klijent je program razvijen nekim od alata za razvoj PC aplikacija, koji koristi sve mogućnosti grafičkog interfejsa na osobnim računarima. Ovo rešenje je najkvalitetnije i najskuplje, te ima dodatnu manu da je vezano uz odredenu klijentsku platformu poput Microsoft Windows

# Sadržaj