Sveučilište Jurja Dobrile u Puli

Odjel za informacijsko-komunikacijske tehnologije

MATEJ ŽGELA

EVOLUCIJA I ODRŽAVANJE SOFTVERA

Završni rad

Pula, rujan, 2017. godine

Sveučilište Jurja Dobrile u Puli
Odjel za informacijsko-komunikacijske tehnologije

MATEJ ŽGELA

EVOLUCIJA I ODRŽAVANJE SOFTVERA

Završni rad

JMBAG: 1412993370037, redovni student
Studijski smjer: Informatika

Predmet: Programsko inžinjerstvo
Znanstveno područje: Društvene znanosti
Znanstveno polje: Informacijske i komunikacijske znanosti
Znanstvena grana: Informacijski sustavi i informatologija
Mentor: doc. dr. sc. Tihomir Orehovački

Pula, rujan, 2017. godine

IZJAVA O AKADEMSKOJ ČESTITOSTI

Ja, dolje potpisan/a Matej Žgela, kandidat za prvostupnika Informatike ovime izjavljujem da je ovaj Završni rad rezultat isključivo mojega vlastitog rada, da se temelji na mojim istraživanjima te da se oslanja na objavljenu literaturu kao što to pokazuju korištene bilješke i bibliografija. Izjavljujem da niti jedan dio Završnog rada nije napisan na nedozvoljen način, odnosno da je prepisan iz kojega necitiranog rada, te da ikoji dio rada krši bilo čija autorska prava. Izjavljujem, također, da nijedan dio rada nije iskorišten za koji drugi rad pri bilo kojoj drugoj visokoškolskoj, znanstvenoj ili radnoj ustanovi.

Student
Matej Žgela

U Puli, 15.rujan,2017.godine

IZJAVA
o korištenju autorskog djela

Ja, Matej Žgela dajem odobrenje Sveučilištu Jurja Dobrile u Puli, kao nositelju prava iskorištavanja, da moj završni rad pod nazivom Evolucija i održavanje softvera koristi na način da gore navedeno autorsko djelo, kao cjeloviti tekst trajno objavi u javnoj internetskoj bazi Sveučilišne knjižnice Sveučilišta Jurja Dobrile u Puli te kopira u javnu internetsku bazu završnih radova Nacionalne i sveučilišne knjižnice (stavljanje na raspolaganje javnosti), sve u skladu s Zakonom o autorskom pravu i drugim srodnim pravima i dobrom akademskom praksom, a radi promicanja otvorenoga, slobodnoga pristupa znanstvenim informacijama.
Za korištenje autorskog djela na gore navedeni način ne potražujem naknadu.

U Puli, 15.rujan.2017.

Potpis
Matej Žgela

SADRŽAJ

UVOD

Što je održavanje softvera (eng. *software*)? Je li to drugačije od razvoja softvera? Zašto softver nije dizajniran da bude lakši za održavanje? Što da radimo sa legacy softverom? Kako zaraditi novac od održavanja? Mnogi od naših konvencionalnih ideja temelje se na analizama provedenim 1970.-ih godina i vrijeme je da ih promislite za modernu softversku industriju.

Podrijetlo pojma *održavanje softvera* nije jasno, ali se upotrebljava posljednjih 25 godina kako bi se odnosilo na raspoređivanje (eng. *post-initial delivery)* rada. Ovaj se pogled odražava u IEEE definiciji (1990: 610-612) softvera: proces modificiranja softverskog sustava ili komponente nakon isporuke za ispravljanje pogreške, poboljšati performanse ili druge atribute ili prilagoditi se promjeni okoline.
Implicirano u ovoj definiciji je pojam životnog ciklusa softvera, koje je definirano kao: razdoblje koje počinje kada je softverski proizvod zamišljen i završava kada softver više nije dostupan za upotrebu. Životni ciklus softvera obično uključuje fazu:

1. Koncepta
2. Zahtjeva
3. Projektiranja
4. Provedbe
5. Ispitivanja
6. Instalacijsku i naplatnu fazu
7. Rada
8. Održavanja
9. Mirovine[1]

Karakteristika postojećih inženjerskih disciplina je da oni utjelovljuju strukturirani, metodički pristup razvoju i održavanju. Modeli životnog ciklusa softvera su apstraktni opisi strukturiranog metodičkog razvoja i modifikacije. Obično prikazuje glavne faze u izradi i održavanju izvršnog softvera. Ideja je započela 1960ih s modelom vodopada.
Internacionalna organizacija standarda (1995.) ističe kako model životnog ciklusa, implicitan ili eksplicitan, primarna je apstrakcija koju softver profesionalci koriste za

[1] Ove se faze mogu preklapati ili se mogu izvršiti iterativno

upravljanje i kontrolu softverskog projekta, ispunjavanje proračuna, vremenskih ciljeva i ciljeva kvalitete, s razumljivim rizicima i koristeći odgovarajuće resurse. Model opisuje proizvodnju isporuke kao što su specifikacije i korisnička dokumentacija, kao i izvršni kod. Model mora biti u skladu s bilo kojima pravnim ili ugovorenim ograničenjima unutar strategije nabave projekta. Stoga ne čudi da su modeli životnog ciklusa dobili primarnu pažnju unutar softvera inženjerske zajednice.

Vrlo koristan model je spiralni model Boehm, koji predviđa softver proizvodnju kao kontinuirani iterativni razvojni proces. Ali ključno, ovaj model se ne bavi gubitkom znanja, što prati podršku dugogodišnjih softverskih sustava i što viralno ograničava zadatke koji se mogu izvršiti.

Ovaj rad će se baviti softverskom evolucijom i održavanjem softvera sa naglaskom na procese i metode, umjesto tehnologiji, budući da su se tu dogodili glavni razvoji događaja i da je od najveće važnosti za ovaj rad. Razumijevanje programa je ključna komponenta. Ono što je zanimljivo je da, vrlo malo udžbenika ima o softverskom inženjeringu, pa čak i o održavanju softvera se samo objašnjava pojam istog. Temelj ove perspektive je da softver prolazi kroz nekoliko prepoznatljivih faza tijekom života. Bennett (2000: 35) te faze dijeli u:
Početni razvoj – prva funkcionalna verzija sustava je razvijena.
Evolucija – ako je početni razvoj uspješan, softver ulazi u stupanj evolucije, gdje inženjeri proširuju sposobnosti i funkcionalnost. Promjene su napravljane kako bi se zadovoljile nove potrebe korisnika, ili zbog toga što i sami zahtjevi nisu bili potpuno razumljivi.
Servisiranje – softver je podvrgnut manjim popravcima grešaka i vrlo jednostavnih promjena u funkciji.
Isključivanje – nema više servisiranja, a vlasnici softvera nastoje ostvariti prihod za njegovu upotrebu koliko god je moguće. Priprema migracijske rute.
Zatvaranje – softver je povučen s tržišta, a korisnici su usmjereni na zamjenski sustav, ukoliko on postoji.
Rad započinje sa pojmom imovine nakon čega se nadovezuje na vrste imovine koje su kako bi čitatelju bilo lakše prepoznati razlike, objašnjene.
U drugom poglavlju osvrnuli smo se na računovodstvene standarde, odnosno na usporedbu kako bi lakše uočili razlike između Međunarodnih standarada financijskog

izvještavanja i Hrvatskih standarada financijskih izvještavanja. Navedeno je na koga se primjenjuju, njihov pojam, način priznavanja te prestanak istog kao i njihovo objavljivanje.
U trećem poglavlju je objašnjen je računalni program – software, od značenja u dugotrajnoj nematerijalnoj imovini, preko opisa nabave sa tržišta do potrebne licence za rad istog.
U četvrtom poglavlju je svrha ovog rada, odnosno objašnjenje i primjeri o odabranom softwareu, kako i potrebnim koracima za njegovo funkcioniranje, počevši od narudžbe i nabavke istog, dostave i instalacije, njegovog knjiženja te vrijednosti softwarea nakon stavljanja u uporabu.

Korištene metode u radu su induktivna i deduktivna, metoda analize , metoda apstrakcije, konkretizacije i generalizacije te deskripcije.

Tema je razrađena na primjeru konkretnog poslovnog subjekta, postojećih licenci i stvarnog softwarea, čije nazive navodimo u smanjenom i dopuštenom obujmu.

1. POČETNI RAZVOJ

Prva faza je početni razvoj, kada je prva verzija softvera razvijena od nule. Ova faza je dobro opisana u literaturi softverskog inženjerstva i postoji vrlo mnogo metoda, alata i udžbenika koji je detaljno opisuju. Fazom se također bavi niz standarda od strane IEEE i ISO.

Tijekom proteklih 30 godina, od prepoznavanja softverskog inženjerstva kao discipline, velika je pozornost posvećena procesu inicijalnog razvoja pouzdanog softvera unutar proračuna i predvidljivih rokova. Voditelji softverskih projekata predstavili su najraniji procesni model, model slapova, jer je ponudio sredstvo da bi proces razvoja bio vidljiviji i auditivniji.

1.1 Ekspertiza softverskog tima

S gledišta budućih stadija, nekoliko važnih temelja postavljeni su tijekom početnog razvoja. Prvi temelj je stručnost inženjerskog tima za softver, a posebno sistemski arhitekti. Početni razvoj je faza tijekom kojeg tim uči o domeni i problemu. Bez obzira koliko prethodnog iskustva je bilo akumulirano prije početka projekta, nova znanja biti će stečena tijekom početnog razvoja. Ovo iskustvo je neophodna vrijednost u tome što će učiniti buduću evoluciju softvera mogućom. Dakle, ovaj aspekt, *početak timskog učenja*, karakterizira prvu fazu. Unatoč mnogim pokušajima dokumentiranja i snimanja takvih timskog učenja, većina je vjerojatno prešutna – takav je doživljaj iznimno teško snimiti formalno.

1.2 Arhitektura sustava

Drugi važan rezultat i isporuka iz inicijalnog razvoja je arhitektura sustava, tj. komponente iz kojih je sustav izgrađen, njegove interakcije i njegova svojstva. Arhitektura će ili olakšati ili ometati promjene koje će se dogoditi tijekom evolucije i hoće li izdržati te promjene ili se podvrgnuti pod njihovim utjecajem. To je svakako moguće dokumentirati arhitekturnim i standardnim pristupima arhitekture. U praksi, jedan od glavnih problema za arhitektonski integritet tijekom početnog razvoja je

ograničenost, kako se ističe u publikaciji Arhitektura Open VME (1994.). Ako zahtjevi softverskog sustava nisu jasni ili ako se mijenjaju dok se softver razvija, onda je jasno da je kontrolu arhitekture teško održati. Razvijeni su brojni pristupi koji ublažavaju ovaj problem, kao što su brži razvoj aplikacija, prototipova i raznih menadžment rješenja, kao što je glavni programerski tim i, u zadnje vrijeme, ekstremno programiranje.

1.3 Što čini arhitekturu evolucijskom?

Za softver koji se lako evoluira mora imati odgovarajuću arhitekturu, a tim inženjera mora imati potrebnu stručnosti. Na primjer, u dugotrajnim sustavima kao što je ICL VME operativni sustav, gotovo su sve pod komponente zamijenjene u nekom stadiju. Još unatoč tome, cjelokupni sustav zadržao je većinu svog arhitektonskog integriteta. Evolucija arhitekture zahtijeva pojedince vrlo visoke stručnosti, sposobnosti i vodstva. Može postojati financijski pritisak kako bi se napravili tehnički prečaci da bi se promjene vrlo brzo izvršile.[2] Bez odgovarajuće razine ljudske spretnosti i razumijevanja možda neće biti shvaćeno da te promjene znatno štete softverskoj strukturi sve dok to ne bude prekasno. Nema jednostavnog odgovora ili *recepta* da bi se izradila evolucijska arhitektura. Neizbježno je da postoji razmjena između trenutnog dobitka i dobitaka u budućnosti, a proces nije nepogrešiv. Pragmatična analiza softverskih sustava koji su izdržali test vremena obično pokazuje da je izvorni dizajn poduzela jedna, ili nekoliko, vrlo talentiranih osoba.

Unatoč brojnim pokušajima, pokazalo se vrlo teško uspostaviti ugovornim, što se podrazumijeva održivim ili *evolvable* softverom i definirati procese koji će podizvoditi softver s time karakteristikama. Na osnovnoj razini, moguće je inzistirati na usvajanju kućnog stila za programiranje, koristiti IEEE ili ISO standarde u upravljanju i tehničkoj implementaciji, koristiti suvremene alate, dokumentirati softver i slično. [3]

Tamo gdje se promjena može predvidjeti u vrijeme projektiranja, moguće je parametrirati funkcionalnost. Ove tehnike mogu biti potrebne, ali iskustvo pokazuje da one nisu dovoljne. Problem se može lako sažeti: uspješan softverski sustav će biti

[2] Zanemarujući problem kako se to ne slaže s arhitektonskim zahtjevima
[3] Softversko inženjerstvo - prikupljanje IEEE standarda op.cit.

podvrgnut promjenama tijekom svog životnog vijeka, koje izvorni dizajneri i arhitekti ne mogu ni zamisliti. Stoga nije moguće planirati takvu promjenu, i zasigurno nije moguće stvoriti dizajn koji će je prihvatiti. Tako će se neki softver razvijati, ali drugi sustaviti imati će arhitekturu koja je u križanju sa potrebnom promjenom. Kod prisilne promjene mogu se uvesti tehnički i poslovni rizici i stvoriti probleme u budućnosti.

2. EVOLUCIJA – KLJUČNI KORAK

Stadij evolucije karakterizira iterativno dodavanje, izmjenu ili brisanje netrivijalnih softverskih funkcija, programskih značajki. Ova faza predstavlja prvu veliku razliku uspoređujući sa tradicionalnim modelom. Uobičajeni prikaz je da se softver razvija i zatim prosljeđuje timu za održavanje. Međutim, u mnogim studijama slučaja opisanih kasnije, nailazimo da to nije slučaj. Umjesto toga softver se oslobađa korisnicima, a pretpostavljajući da je uspješan, počinje poticati entuzijastične korisnike.[4] Također počinje generirati prihode i tržišni udio.

Korisnici daju povratne informacije i zahtjeve za novim značajkama. Projektni tim živi u okruženju uspjeha što potiče više dizajnere da se drže oko sebe i podržavaju sustav kroz niz izdanja. U smislu timskog učenja, obično je izvorni dizajnerski tim koji vidi novi sustav kroz svoje napredne rane. Naravno, pogreške će se otkriti tijekom ove faze, ali su zakazane za ispravak u sljedećem koraku. Lehmann ističe kako tijekom evolucijskog stadija, kontinuirana dostupnost visoko kvalificiranog osoblja omogućava održavanje arhitektonskog integriteta.“ Čini se da je ovakvo osoblje bitno. Nažalost, izrada ovog oblika stručnosti nije bila uspješna usprkos brojnim projektima koji se tiču inženjeringa temeljenoga na znanju.“ (Lehman; 1995.: 78)

Povećanje veličine, složenosti i funkcionalnosti softvera djelomično je rezultat procesa učenja u softverskom timu. Cusumano i Selby (1997: 53) izvijestili su da se značajka postavljena tijekom svake iteracije može promijeniti za 30% ili više, kao izravni rezultat procesa učenja tijekom iteracije. Brooks također primjećuje kako postoji znatna *krivulja učenja* u izgradnji uspješnog novog sustava. Povećanja veličine i složenosti uzrokovana su i zahtjevima kupaca za dodatnom funkcionalnošću, a pritisci na tržište dodatno pridonose rastu jer može biti potrebno uskladiti značajke konkurentnog proizvoda.

U nekim područjima, kao što je javni sektor, zakonodavna promjena može prisiliti velike evolucijske promjene, često u kratkom vremenu, koje nikada nisu bile predviđene kada je softver prvi put proizveden. Često je kontinuirani tok takvih promjena.

[4] Ako to nije uspjelo, onda se projekt otkazuje

2.1 Izdanja softvera

Kupci obično imaju više izdanja tijekom faze razvoja softvera. Vrijeme svakog izdavanja temelji se na tehničkim i poslovnim razmatranjima. Menadžeri moraju voditi računa o različitim sukobljenim kriterijima koji uključuju vrijeme za traženje ili vrijeme isporuke, stabilnost softvera, izvješća o kvaru i slično. Osim toga, izdanje se može sastojati od nekoliko koraka, uključujući alfa i beta izdanja. Stoga je izdanje, koje je tradicionalna granica između razvoja softvera i održavanja softvera, do određenog stupnja proizvoljna prekretnica. Za softver s velikim korisničkim bazama, uobičajeno je proizvesti niz verzija. Ove verzije postoje među korisnicima i samostalno su servisirane, uglavnom za popravke programskih pogrešaka. Ovo servisiranje može biti u obliku zakrpa ili manjih izdanja ako da određena kopija softvera u rukama korisnika može imati i broj verzije i broj izdanja. Izdanja rijetko implementiraju značajnu novu funkcionalnost.

2.2 Razvoj evolucijskog softvera

Sadašnji trend u programskom inženjerstvu je smanjivanje procesa početnog razvoja, čineći ga samo u preliminarnom razvoju skeletne inačice ili prototipa aplikacije. Potpuni razvoj sastoji se od nekoliko iteracija, od kojih svaka dodaje određene funkcije ili svojstva na već postojeći softverski sustav. U takvoj situaciji, razvoj softvera uvelike zamjenjuje inicijalni razvoj, koji tada postaje ništa više nego prva od nekoliko jednakih iteracija. [5]

Svrha evolucijskog razvoja je smanjiti rizike zahtjeva. Kao što je prethodno primijećeno, zahtjevi softvera često su nepotpuni zbog poteškoća u njihovom nastanku. Korisnici su odgovorni za pružanje kompletnog skupa točnih zahtjeva, ali često pružaju manje od toga, zbog nedostatka znanja ili običnih propusta. Na kraju se zahtjevi mijenjaju tijekom razvoja, jer se situacija u koji softver operira mijenja. Tu je i proces učenja od strane korisnika i implementatora, a to opet pridonosi promjeni

[5] Informacijska tehnologija - procesi životnog ciklusa softvera.. Op.cit

zahtjevima. Zbog toga je potpuni skup uvjeta nemogući ili malo vjerojatan u mnogim situacijama tako da implementacija velikog softvera u jednom koraku nosi značajan rizik. Razvoj evolucijskog softvera koji je podijeljen u inkrementalne korake smanjuje rizik jer omogućuje korisnicima da vide i dožive nepotpuni softver nakon svake iteracije.

Jedan od poznatih i dobro opisanih procesa evolucije softvera je Unified Software Development Process opisan od strane Boehma (1993: 61-72).
Ovaj proces detaljno opisuje kako se softver treba razviti u inkrementalne iteracije. Svaka inkrementalna iteracija dodaje novu funkcionalnost ili novo svojstvo[6] već postojećem softveru. Ovo postupno povećanje zahtjeva smanjuje rizik koji je uključen, jer svaka iteracija daje svježe povratne informacije o napretku projekta. Ujedinjeni procesi razvoja softvera opisuje broj aktivnosti i specificira dokumente koji se izrađuju tijekom iteracija.

Međutim, Booch (2001: 119-121) izvještava da je glavna kritika podignuta na Unified Software Development Process i slične pristupe jer to rezultira krutim procesima koji zahtijevaju veliku dokumentaciju i mnoge korake i posljedično su previše skupi u vremenu za mnoge moderne tvrtke.

Novi alternativni pristup za sustave koji zahtijevaju brzu evoluciju jest agilna metoda, primjer čega je Extreme Programming (XP). XP gotovo ukida početnu fazu razvoja. Umjesto toga, programeri blisko surađuju s kupcima kako bi razvili skup *priča* koji opisuju željene značajke novog softvera. Zatim se provodi niz izdanja, s obično samo nekoliko tjedana između svakog. Kupac definira sljedeći oslobađajući odabir priča za implementaciju. Programeri uzimaju priče i definiraju više zadataka, uz jednog programera koji preuzima odgovornost za svaki. Testni slučajevi su definirani prije početka programa.

Zanimljiv aspekt XP je da odgovoran programer prijavi partnera za zadatak, sve se radi u paru, a oba rade na istoj radnoj stanici. Tako se znanje dijeli između najmanje dva programera a neka vrsta samokontrole je izgrađena bez potrebe za organiziranim

[6] Sigurnost, učinkovitost, ...

prohodima ili inspekcijama. Parovi su razbijeni i reformirani na različite zadatke tako da se iskustvo može distribuirati. Malo je dokumentacija o kodu ili dizajnu iako se poduzima znatna pažnja za održavanjem testova koji se mogu ponoviti u budućnosti.[7]

Činilo se da agilne metode odbacuju svo iskustvo inženjerstva softvera u posljednjih dvadeset godina i stvaraju svoje oslanjanje isključivo na zadržavanje osoblja stručnog tima sve dok se softver treba razvijati. Tako su dobili dragocjeno vrijeme, ali možda i značajan rizik. Ostaje vidjeti ako će ova vrsta metodologije biti održiva nakon kratkog roka ili ako menadžeri i dioničari umjesto toga otkriju da su njihove kritične aplikacije odjednom napravile neplanirane i skupe prijelaze na servisiranje. [8]

[7] Loc.cit
[8] Loc.cit

3. SERVISIRANJE

3.1 Propadanje softvera

Kao što je ranije spomenuta, softver ulazi u servisnu fazu kada su ljudska stručnost i/ili arhitektonski integritet izgubljeni/narušeni. Burd (1997: 322-329) ističe kako je servisiranje bilo alternativno nazvano *zasićenje, softversko zastarenje, propadni softver, spreman za održavanje i nasljedni softver*. Tijekom ove faze, teško je i skupo napraviti izmjene i stoga su promjene obično ograničene na minimum. Istodobno, softver i dalje može imati *kritički važan* status, tj. organizacije korisnika može se osloniti na softver za usluge koje su bitne za njegovo preživljavanje.

Simptomi propadanja koda uključuju:

1. Pretjerano složen kod, tj. kod koji je složeniji nego što to mora biti
2. Preostali kod koji podržava značajke koje se više ne koriste ili zahtijevaju
3. Česte promjene u kodu
4. Povijest pogrešaka u kodu
5. Često su nelokalizirane promjene, tj. promjene koje utječu na mnoge dijelove koda
6. Programeri koriste promjene izvršene na neelegantan ili neučinkovit način, na primjer klonovi ili zakrpe
7. Brojne zavisnosti u kodu; kako se broj ovisnosti povećava, sekundarni učinci promjene postaju češći i mogućnost uvođenja pogreške u softver se povećava.

3.2 Gubitak znanja i kulturne promjene

Kako bi razumjeli softverski sustav, programeri trebaju mnogo vrsta znanja. Programeri moraju razumjeti domenu detaljno. Moraju razumjeti objekte domene, njihova svojstva i odnose. Moraju razumjeti poslovni proces koji program podržava, kao i sve aktivnosti i događaje tog procesa. Oni također moraju razumjeti algoritme i strukture podataka koje implementiraju objekte, događaje i procese. Moraju razumjeti arhitekturu programa i sve njegove prednosti i slabosti, nesavršenosti kojima se program razlikuje od ideala. Ovo znanje mogu djelomično zabilježiti u programskog dokumentaciji, ali obično je takve veličine i složenosti da je potpuna snimka

nepraktična. Lagu (1997: 98) ističe kako veliki dio toga obično nije zabilježen i ima oblik pojedinačnih *iskustava* ili grupne *usmene tradicije*.

Ovo znanje stalno je u opasnosti. Promjene u kodu čine znanje koje je zastarjelo. Kako se simptomi propadanja proliferiraju, kod postaje sve više i više složeniji i dublje znanje je potrebno kako bi se mogao razumjeti. Istodobno, obično dolazi do promjene programera na projektu.

Promjena može imati različite uzroke, uključujući i prirodnu promjenu programera zbog svojih osobnih razloga, ili potrebe drugih projekata koji prisiljavaju menadžere da programeri pređu na posao. Bazirano na uspjehu projekta, članovi tima se promoviraju, prebacuju u druge projekte i uglavnom raspršuju. Timska stručnost što podržava strateške promjene i evoluciju softvera je strogo izgubljena – novi članovi osoblja pridružuju se timu koji ima više taktičke perspektive[9] softvera. Evolvivnost je izgubljena i, slučajno ili dizajnom, sustav se uklanja u servisiranje.

Međutim, uprava koja je svjesna propadanja može prepoznati eventualnu tranziciju planiranjem za nju. Tipično, trenutno je softver preseljen na servisnu poziciju, a viši dizajneri iniciraju novi projekt za objavu radikalno nove verzije.[10] Posebna instanca gubitka znanja je kulturna promjena u softverskom inženjeringu.

Softversko inženjerstvo ima gotovo pola stoljetne tradicije, a postoje programi koji su još u upotrebi stvori prije više od četrdeset godina. Ti programi stvoreni su u kontekstu potpuno različitih svojstava hardvera, jezika i operacijskih sustava. Računala su bila sporija i imali su mnogo manje memorije, često zahtijevajući razrađene tehnike za rješavanje s tim ograničenjima. Programske arhitekture u uporabi također su različite – moderne arhitekture pomoću tehnika kao što su objektno orijentirane (eng. *object-oriented*) bile su rijetke u to vrijeme. Prema Schachu (1999: 123) programeri koji su stvorili te programe vrlo su često nedostupni. Trenutni programeri koji pokušavaju mijenjati ove stare programe suočeni su s dvostrukim problemom: ne samo da moraju obnoviti znanje koje je potrebno za taj specifičan problem, ali oni također moraju oporaviti znanje kulture u kojoj su taj i slični programi stvoreni. Bez tog kulturnog

[9] Na razini koda
[10] Često s novim imenom, novim tržišnim pristupom

razumijevanja oni možda neće biti u stanju napraviti najjednostavnije promjene u programu.

3.3 Omotavanje, krpanje, kloniranje

Tijekom servisiranja teško je skupo napraviti izmjene i stoga promjene su obično ograničene na minimum. Lagu (1997: 314) objašnjava kako programeri također moraju koristiti neobične tehnike za promjene. Jedna takva tehnika je **omotavanje**. Sa omotavanjem softver se tretira kao crna kutija, a promjene se primjenjuju kao omotnice gdje je izvorna funkcionalnost promijenjena u novu sa izmjenama ulaza i izlaza starih softvera. Očito se ovako mogu provesti samo promjene ograničene vrste. Štoviše, svaka takva promjena dalje razgrađuje arhitekturu softvera i gura ga dublje u servisnu fazu. Druga vrsta promjene koja se često koristi tijekom servisiranja jest **kloniranje**. Ako programeri potpuno ne razumiju program umjesto da pronađu gdje se implementira određena funkcionalnost u sustavu programa, oni stvaraju drugu implementaciju. Tako program može završiti s nekoliko implementacija istih ili gotovo identičnih struktura podataka, nekoliko implementacija identičnih ili gotovo identičnih algoritama i slično. Ponekad programeri namjerno stvaraju klonove iz straha od sekundarnih učinaka promjene. Kao primjer, pretpostavimo da funkcija abc() zahtjeva promjenu, ali abc() se može pozvati z drugih dijelova koda tako da promjena u abc() može stvoriti sekundarne efekte u tim dijelovima. Kako je poznavanje programa u servisnoj fazi nisko, programeri odaberu *sigurnu* tehniku: copy-paste abc(), stvarajući namjerni klon abc1(). Zatim ažuriraju abc() kako bi se zadovoljili novi zahtjevi, dok stari abc() još uvijek ostaje u upotrebi u drugim dijelovima programa. Stoga nema sekundarnih učinaka na mjestima gdje je abc() pozvan. Dok programeri rješavaju svoj neposredan problem na ovaj način, oni negativno utječu na arhitekturu programa i otežavaju buduće promjene., zaključuje Johnson (1994: 120-126).

Baxter (1998: 368-377) napominje kako većem broj klonova u kodu značajna je prisutnost simptoma koda tijekom servisiranja. Nekoliko autora je predložilo metode otkrivanja klonova automatski pomoću *substring matching* ili *subtree matching*. Softver menadžeri mogu razmotriti praćenje rasta klonova kao mjeru propadanja koda i razmotriti korektivne akcije ako se čini da se sustav prebrzo raspada.

Servisne zakrpe su fragmenti koda, vrlo često u binarnom obliku, koriste se za distribuciju popravaka programskih pogrešaka u široko distribuiranom softverskom sustavu.

3.4 Reinženjering

Beck i Kent u članku Prihvaćanje promjena s ekstremnim programiranjem (1999: 70-77) ističu kako je u servisnoj fazi, teško preokrenuti situaciju i vratiti se fazi evolucije. To bi zahtijevalo ponovno stjecanje potrebnih stručnosti za evoluciju, rekonstrukciju arhitekture te restrukturiranje softvera. I restrukturiranje i povratak stručnosti su spori i skupi procesi, s mnogim rizicima koji su uključeni, a novo osoblje mora biti zaposleno s odgovarajućim i relevantnim vještinama. Korisnici naslijeđenog sustava grade svoje radne rutine i očekivanja na temelju usluga koje pruža i stoga su vrlo osjetljivi na sve poremećaje rutine. Njihova tolerancija na promjene može biti mnogo manja od tolerancije koju pokazuju korisnici potpuno novih sustava. Stoga krutost korisnika također čini reinženjering vrlo rizičnim prijedlogom. Kako bi se smanjio rizik i poremećaj rutine korisnika, zagovara se inkrementalni reinženjering, gdje je sustav promijenjen dio po dio. Novi dijelovi privremeno koegzistiraju sa starim dijelovima i stari dijelovi zamjenjuju se jedan po jedan, bez prekida usluge. Ovim pristupom reinženjeringu izbjegava se poremećaj korisničkih rutina, ali također čuva sučelja između dijelova, a time i preko svega arhitekturu sustava. Ako je arhitektura također zastarjela, proces pruža samo djelomično olakšanje. To utječe na poslovni slučaj za reinženjering, budući da prednosti koje se vraćaju u odnosu na potrebnu investiciju mogu biti teško opravdani. U najgorem slučaju, trošimo sredstva za malo ili nimalo koristi. Dodatna poteškoća s ponovnim projektiranjem široko korištenog softvera je problem distribucije. Dobivanje nove verzije za sve korisnike može biti skup ili nemoguć proces, tako da teret servisiranja stare verzije može potrajati. Jacobson (1999: 149) smatra kako će taj problem zasigurno postati još veći kada se softver sve više uvede u robu široke potrošnje kao što su mobilni telefoni. Nakon što je object level code pušten na takve uređaje, gotovo je nemoguće vratiti se na evolucijsku fazu. Kompletan reinženjering kao način, korak po korak od servisiranja do evolucije, vrlo je rijedak i skup, tako da ulazak u servisnu fazu je za se praktične svrhe nepovrata.

4. UKIDANJE

U nekoj fazi sustav je uglavnom zamrznu i nema daljnjih promjena. Ova faza, koju zovemo *phase-out (close-down)*, također se zove *pad*. Osoblje za potporu korisnika može i dalje biti na mjestu kako bi pomoglo korisnicima u pokretanju sustava, ali zahtjevi za promjenom više nisu poštovani. Korisnici moraju raditi oko bilo kakvih preostalih nedostataka umjesto da očekuju da budu fiksni. Konačno, sustav može potpuno biti povučen iz službe, pa čak i ta osnovna razina osoblja se više ne pruža. Lagu (1999: 317) kaže kako točan tijek ukidanja i zatvaranja ovisit će o specifičnim sustavima i ugovorne obveze. Ponekad sustav u ovoj fazi još uvijek generira prihod, ali u drugim slučajevima[11] korisnik je već platio za to. U ovom drugom slučaju, proizvođač softvera može biti puno manje motiviran pružiti podršku.

U anketi tvrtke Tamai i Torimitsu, provedena je istraga životnog vijeka softvera u Japanu. Istraživanje se bavilo softverom tvrtke s nekoliko područja primjene kao što su proizvodnja, financijske usluge, izgradnja, masovni mediji i slično. Otkrili su da je za softver veći od 1 milijun redaka koda, prosječni život bio 12,2 godine sa standardnim odstupanjem od 4,2 godine. Životni vijek široko se razlikovao za manji softver. Tamai i Torimitsu također su razvrstali razloge zatvaranja na sljedeći način. Promjena hardvera i/ili sustava uzrokovala je zatvaranje 18% slučajeva. Razlog je bila nova tehnologija u 23,7% slučajeva. Potrebno je zadovoljiti nove zahtjeve korisnika[12]. Konačno pogoršanje održivosti softvera bilo je krivac u 25,4% slučajeva. Možemo nagađati da je softver na kraju životnog ciklusa bio u fazi istupanja i u većini slučajeva došlo je do događaja[13] koji su gurnuli softver u zatvaranje. Samo u 25,4% slučajeva se zatvaranje dogodilo prirodno kao odluka slobodnog menadžmenta bez ikakvog događaja izvana. Postoji niz pitanja vezanih uz zatvaranje softvera. Ugovori bi trebali definirati pravne odgovornosti u ovoj fazi. U nekim slučajevima, kao što je *outsourced* softveru kojem je jedna tvrtka ugovorila s drugima razviti sustav, odnosi mogu biti prilično složeni. Konačno vlasništvo i zadržavanje sustava, njegov izvorni kod i njegovu dokumentaciju treba jasno definirati.

[11] Poput većine skupljanja wrap softvera
[12] Stari sustav nije mogao zadovoljiti – uzrok 32,8% slučajeva
[13] Hardverska promjena, nova tehnologija, novi zahtjevi

Najčešće se podaci sustava moraju arhivirati i mora mu se pristupiti. Primjeri takvih podataka su studentski prijepisi, rodni listovi i drugi dugotrajni podaci. Pitanja arhiviranja podataka i dugoročnog pristupa moraju biti riješeni prije isključivanja sustava.

5. PROMJENE SOFTVERA I RAZUMIJEVANJE

5.1 Mini proces promjene

Tijekom faze evolucije i servisiranja softverski sustav ide kroz niz promjena. Zapravo, i evoluciju i servisiranje čine ponovljene promjene, a time je razumijevanje procesa promjene softvera ključ za razumijevanje ovih faza i problema cijelog životnog ciklusa softvera. Prema tome, u ovom poglavlju proučavamo proces promjene detaljnije, rastavljajući promjenu u konstitutivne zadatke. Posebno važan zadatak je razumijevanje programa, jer se zauzima većinu vremena programera, a njegov uspjeh dominira onim što se može ili ne može postići promjenom softvera.

Zadaci koji uključuju promjenu softvera su sažeti u mini procesu promjene. Da bi naglasili zadatke koje smatramo važnima, podijelimo ih drugačije od standarda i grupiramo ih u mini proces na sljedeći način ističe Booch (1997: 119-121)

1. Zamjena zahtjeva: predloženi su novi zahtjevi za sustav
2. Promjena planiranja: analizirati predložene promjene
    a. Razumijevanje programa: razumjeti ciljani sustav
    b. Promijeniti analitiku učinka: analizirati potencijalne promjene i njihov opseg
3. Promjena provedbe: promjena je napravljena i potvrđena
    a. Restrukturiranje, refaktoriranje promjene
    b. Početne promjene
    c. Razmnožavanje promjena: izvršite sekundarne promjene kako biste zadržali cijeli sustav dosljedan
    d. Provjera valjanosti i verifikacija: kako bi se osiguralo da li sustav nakon promjena zadovoljava novi zahtjev i da li je stari zahtjev nepovoljno uticao na promjenu
    e. Ponovna dokumentacija: predložiti promjenu u svoj dokumentaciji
4. Dostava

5.2 Promjena zahtjeva i planiranje

Korisnici sustava obično potječu zahtjeve za promjenu[14]. Booch (2001 : 119-121) objašnjava kako ti zahtjevi imaju oblik izvještaja o pogrešci ili zahtjeva za poboljšanjima. Standardna je praksa da imate datoteku zahtjeva[15] koji se redovito ažurira. Postoji rok za podnošenje zahtjeva za promjene sljedećeg izdanja. Nakon toka, menadžeri odlučuju koji će posebno zahtjevi biti provedeni u tom izdanju. Svi zahtjevi koji se podnose nakon roka koji nisu ušli u izdanje će morati čekati sljedeće izdanje. Čak i ova površna obrada zahtjeva za promjenom zahtijeva neko razumijevanje postojećeg sustava, tako da potrebni napori mogu biti procijenjeni. Uobičajena pogreška je da se drastično podcjenjuje vrijeme potrebno za promjenu softvera i time vrijeme za stvaranje otpusta.

U malim promjenama, dovoljno je pronaći odgovarajuću lokaciju u kodu i zamijeniti staru funkcionalnost novom. Međutim velike inkrementalne promjene zahtijevaju implementaciju novih koncepata domene. Razmislite o maloprodajnoj *Point-of-sale* aplikaciji za rukovanje bar kodom skeniranja i *check-out* kupca. Zahtjev bi se trebao baviti s nekoliko oblika plaćanja, kao što su gotovina i kreditne kartice. Poboljšanje za obradu provjere isplate će uključivati novi koncept, povezan sa postojećim metodama plaćanja, ali dovoljno različito da zahtijevaju dodatne strukture podataka, obradu za autorizaciju i slično. Biti će dosta novog koda, i pažnja je potrebna da se održi dosljednost s postojećim kodom kako bi se izbjeglo degradiranje arhitekture sustava. Koncepti koji su međusobno ovisni moraju se provesti u redoslijedu njihove ovisnosti. Na primjer, pojam porez ovisi o konceptu stavke jer različite stavke mogu imati različite porezne stope i porez bez stavke nema smisla. Stoga, provedba stavke mora prethoditi provedbi poreza. Ako je nekoliko pojmova međusobno ovisno o tome moraju se provoditi u istoj inkrementalnoj promjeni.

Međusobno neovisni pojmovi mogu se uvesti u proizvoljnom redoslijedu, ali je poželjno ih uvesti u red važnosti za korisnika. Za primjerice, u programu Point of sale je važnije pravilno raditi s porezima nego potporom nekoliko blagajnika. Aplikacija s točnom

[14] Ili zahtjeve za održavanje
[15] backlog

podrškom za poreze je već korisna u trgovinama s jednim blagajnikom. Suprotan poredak inkrementalnih promjena odgađao bi uporabu programa.
Planiranje promjena zahtjeva uvođenje ili daljnje razvijanje koncepata domene. Također zahtjeva pronalaženje u starom kodu mjesto gdje se ovi pojmovi trebaju provoditi kako bi se ispravno komuniciralo s ostalim već prisutnim konceptima. Očito ti zadaci zahtijevaju duboko razumijevanje softvera i njezine problematične domene.

5.3 Promjena provedbe

Provedba softverske promjene zahtjeva nekoliko zadataka, često s nekoliko petlji i ponavljanjem. Ako predložena izmjena ima veliki utjecaj na arhitekturu, može doći do preliminarnog restrukturiranja programa da bi se održala čistoća dizajna. U objektno orijentiranom programu, na primjer, to može uključivati refaktoring za premještanje podataka ili funkcije iz jedne klase u drugu. Stvarna promjena može se pojaviti na jednom od nekoliko načina. Za male promjene, zastarjelu šifru zamjenjuje se novim kodom. Za velike inkrementalne promjene, napisan je novi kod, a zatim *priključen* u postojeći sustav. Nekoliko novih klasa koje implementiraju novi koncept mogu biti napisane, testirane i povezano sa starim klasama, već u kodu.

Vrlo često promjena će se širiti, tj. to će zahtijevati sekundarne promjene. Da bismo objasnili širenje promjena, moramo shvatiti da se softver sastoji od entiteta[16] i njihove ovisnosti. Ovisnost između entiteta A i B znači da entitet B pruža određene usluge koje A zahtjeva. Funkcijski poziv je primjer ovisnosti među funkcijama. Različiti programski jezici ili operativni sustavi mogu pružiti različite vrste entiteta i ovisnosti. Ovisnost A na B je konzistentna ako su ispunjeni zahtjevi A prema onome što B pruža.

Ovisnosti mogu biti suptilne razlike. Učinak može biti na razini koda; na primjer, modul u promjeni može koristiti globalnu varijablu na novi način, pa se sve upotrebe globalne varijable moraju analizirati. Ovisnost se također može dogoditi preko nefunkcionalnih zahtjeva ili poslovnih pravila. Na primjer, u sustavu u stvarnom vremenu, promjena koda može suptilno utjecati na svojstva vremena. Iz tog razloga, analiza promjena i određivanje koda koji se često mijenja ne može lako biti kompartmentiziran. Stručni

[16] Klasa, objekti, funkcije, ...

inženjeri za održavanje trebaju duboko razumijevanje cijelog sustava i načina interakcije s okolinom kako bi se utvrdilo kako se potrebna promjena treba provesti dok se nadamo izbjegavanju oštećenja arhitekture sustava. Poslovna pravila mogu biti iznimno složena[17]; u starom sustavu, bilo koja dokumentacija o takvim pravilima vjerojatno je izgubljena i određivanje pravila retrospektivno može biti dugotrajan i skup zadatak. Provođenje promjene u softveru tako počinje s promjenom u specifičnoj jedinici softvera. Nakon promjene, entitet se više ne može uklopiti s ostalim entitetima softvera, jer možda više ne pruža ono što drugi entiteti zahtijevaju ili možda zahtijevaju različite usluge entiteta o kojima ovisi. Zavisnosti koje više ne zadovoljavaju zahtijevanje-pružanje odnose se nazivaju nedosljedne ovisnosti[18], a mogu se pojaviti u svakom trenutku promjene softvera. Kako bi se ponovno uvela dosljednost u softveru, programer prati nedosljednosti i mjesta na kojima je potrebno izvršiti sekundarne promjene. Sekundarne promjene, međutim, mogu uvesti nove nedosljednosti i tako u krug. Proces u kojem se promjena širi putem softvera ponekad se zove mrežni utjecaj promjene. Programer mora jamčiti da je promjena ispravno propagirana, te da u softveru ne ostaje nikakva nekonzistentnost. Neočekivana i nekorigirana nedosljednost jedan je od najčešćih izvora pogrešaka u modificiranom softveru. Softverski sustav ne sastoji se samo od koda, već i dokumentacije. Zahtjevi, nacrti, planovi testiranja i korisnički priručnici mogu biti prilično opsežni i često su također promijenjeni. Ako će dokumentacija biti korisna u budućnosti, ona također mora biti ažurirana. Očito je da modificirani softverski sustav mora biti potvrđen i provjeren. Najčešće korištena tehnika je regresijsko ispitivanje, u kojem je skup sustavih testova očuvan i ponovno pokrenut na modificiranom sustavu. Set testova regresije mora imati prilično dobru pokrivenost postojećeg sustava ako želi biti učinkovit. Raste tijekom vremena dok se testovi dodaju za svaki novi koncept i dodanu značajku. Niz regresijskih testova također će morati biti ponavljan mnogo puta tijekom trajanja projekta. Regresijsko ispitivanje nije tako jeftino, pa je tako vrlo poželjno automatizirati vođenje testova na provjeru.

Testiranje međutim, nije dovoljno kao jamstvo da je dosljednost održavana. Inspekcije se mogu koristiti na nekoliko točaka promjena mini procesa kako bi se potvrdilo da se promjena uvodi na pravoj točki, da rezultirajući kod udovoljava standardima te da je

[17] Pravila poslovanja koja se odnose na navigaciju i sustavi letenja u sigurnosnom sustavu kontrole letenja na vozilu

[18] Nedosljednosti u kratkom roku

dokumentacija doista bila ažurirana radi dosljednosti. Jasno je da je jasno razumijevanje softverskog sustava bitno na svim točkama tijekom provedbe promjene. Refactoring zahtjeva viziju arhitekture i podjele odgovornosti između modula ili klase. Promjena propagacijske analize zahtjeva praćenje ovisnosti jednog entiteta na drugom, i može zahtijevati prepoznavanje suptilnog vremena ili ovisnosti o poslovnim pravilima. Ažuriranje dokumentacije može biti između većine znanja zahtjevnih zadataka jer zahtjeva svijest o više mjesta u svakom dokumentu gdje je opisan bilo koji pojam. Čak i testiranje zahtjeva razumijevanje test seta, njegove pokrivenosti i gdje se ispituju različiti pojmovi.

U prethodnim stavcima je opisano što treba učiniti kao dio svake promjene softvera. S obzirom na napore potrebne da *to učini*, nije iznenađujuće otkriće da se u praksi neki od tih zadataka preskaču ili zanemaruju. U svakom takvom slučaju dolazi do razmjene između neposrednih troškova ili vremena i eventualne dugoročne koristi. U sljedećoj cjelini biti će riječi kako nije nužno neracionalne odabrati neposredno nad dugoročnim, ali sve takve odluke treba poduzeti sa punom svijesti o potencijalnim posljedicama.

Kao što se ranije opisalo, razumijevanje programa ključna je odrednica životnog ciklusa bilo kojeg specifičnog softverskog proizvoda. Da bi razumjeli zašto je to moramo razumjeti što znači razumjeti program, zašto je to razumijevanje teško i kako se uklapa u ciklus promjena softvera.

5.4 Razumijevanje programa

Razumijevanje programa provodi se s ciljem razumijevanja izvornog koda, dokumentacija, testnog paketa, dizajna i slično od strane inženjera. To je obično postupni proces izgradnje razumijevanja, koji može biti dodatno korišten da se objasni izgradnja i rad programa. Tako razumijevanje programa je aktivnost razumijevanja kako je program izgrađen i njegovu temeljnu namjeru. Inženjer treba precizno poznavati podatke u programu, način na koji se te stavke stvaraju i njihove odnose ističe Beck (1999: 70-77)

Razna ispitivanja pokazala su da je glavna aktivnost u održavanju razumijevanje izvornog koda. Razumijevanje programa možemo opisati kao najskuplji i radno najintenzivniji dio softverskog održavanja, te se može reći da je ključ učinkovitog softverskog održavanja - razumijevanje programa. Tako da je to ljudsko-intenzivna aktivnost koja troši znatne troškove.
Razumijevanje se zatim koristi za:

1. Održavanje i evoluciju
2. Inverzni inženjering
3. Učenje i obuku
4. Redokumentaciju
5. Ponovno korištenje
6. Testiranje i validaciju

Polje je potaknulo nekoliko teorija izvedenih iz empirijske istrage ponašanja programera. Postoje tri temeljna gledišta:

1. Razumijevanje se vrši odozgo prema dolje, od zahtjeva do provedbe
2. Razumijevanje se poduzima u smjeru odozdo prema gore, počevši od izvornog koda i deduciranje onoga što čini i kako to radi
3. Razumijevanje se poduzima oportunistički

Sve tri se mogu koristiti u različito vrijeme, čak i od strane jednog inženjera. Ohrabrujuće je napomenuti da je velik dio rada na razumijevanju podržan empirijskim radom kako bi stekli razumijevanje onoga što inženjeri zapravo rade. Za podršku razumijevanju, skupina alata je proizvedena i neki od njih prezentiraju informacije o programu, kao što je varijabla korištenja, grafikoni poziva i sl. u dijagramu ili grafičkom obliku.

Alati se dijele u dvije vrste:

1. Alati za statičku analizu, koji pružaju informacije inženjeru temelje se samo na izvornom kodu, eventualno i dokumentaciji
2. Alati za dinamičku analizu, koji pružaju informacije kad se program izvršava.

Noviji rad koristi virtualnu stvarnost i mnogo sofisticiranije vizualizacijske metafore kako bi pomogle razumijevanju. Svi autori se slažu da je razumijevanje programa

ljudski orijentirano i vremenski intenzivan proces, koji zahtjeva stručnost na programskom jeziku i okolišu, duboko razumijevanje specifičnog koda i njegovih interakcija, kao i poznavanje problemske domene, zadaće koje softver obavlja u toj domeni, te odnosima između tih zadataka i softvera.

Kao što je ranije spomenuto, pronalaženje pojmova u kodu je zadatak programskog razumijevanja te je vrlo važno tijekom faze planiranja promjene. Zahtjevi za promjenu vrlo su često formulirani kao upiti za promjenom ili uvođenjem implementacije specifičnih pojmova u domeni i prvi zadatak je pronaći gdje se ti pojmovi nalaze u kodu. Uobičajena pretpostavka je da korisnik ne mora razumjeti cijeli program, nego samo dio koji je relevantan za koncepte uključene u promjenu.

Jedna od tehnika traženja koncepata je da se prilikom pokušaja pronalaženja koncepta u kodu, programer traži varijable, funkcije, klase i slično s imenom sličnim nazivu koncepta. Na primjer, kada pokušavati pronaći implementaciju prekidnih točaka u debuggeru, programer traži varijable s identifikatorima breakPoint, BreakPoint, Break_point, brkpt i slično. Koriste se alati za podudaranje tekstualnih uzoraka poput *grep* za ovu svrhu. Nakon što pronađe odgovarajući identifikator, programer čita i obuhvaća okolni kod kako bi se locirali svi kodovi koji se odnosi na koncept koji se potražuje.

Druga tehnika za lociranje koncepta ili značajke temelji se na analizi tragova izvršenja programa. Tehnika zahtjeva instrumentaciju programa kako bi se moglo utvrditi koje su grane programa bile izvršene za određeni skup ulaznih podataka. Tada se program izvodi za dva skupa podataka: skup podataka A s značajkom i skupom podataka za B bez značajke. Značajka se najvjerojatnije nalazi u granama za koje je izvršen skup podataka A, ali nisu izvršeni za skup podataka B.

Treća metoda za lociranje koncepta temelji se na statičkom pretraživanju koda. Pretraga obično počinje u funkciji glavna() i programer pokušava pronaći tu implementaciju koncepta. Ako se ne može smjestiti tamo mora se provesti u jednoj od pod funkcija pozvanih od glavna(), dakle programer odlučuje koja će pod funkcija najvjerojatnije implementirati koncept. Ovaj se proces rekurzivno ponavlja sve dok se koncept ne pronađe.

Kao što je već rečeno, razumijevanje programa ključni je dio kvalitete softvera i njegove mogućnosti evolucije, te da je istraživanje u razumijevanju programa jedna od ključnih granica istraživanja u softverskoj evoluciji i održavanju.

6. ODRŽAVANJE SOFTVERSKE VRIJEDNOSTI

6.1 Stavljanje izvan servisne pozornice

Jasno je iz argumenata da softver suptilno gubi veći dio svoje vrijednosti vlasnicima, kada prelazi od evolucijske do servisne faze. Softverski sustav u fazi evolucije rutinski je prilagođen promjeni organizacijske potrebe i stoga mogu dati značajan doprinos misiji organizacije i/ili prihoda. Kad sustav prijeđe u fazu servisiranja mogu se napraviti samo najjednostavnije promjene. Softver je manje vrijedna imovina i može postati ograničenje na uspjeh organizacije, tako što će većina menadžera softvera odugovlačiti servisnu fazu – što je duže moguće. Postoji niz strategija koje se mogu usvojiti za održavanje softverske vrijednosti, ali na žalost sve one proizvode svoje prednosti dugoročno, ali u kratkom vremenu zahtijevaju trošenje napora ili vremena. Softver menadžer mora tražiti odgovarajući kompromis između neposrednog proračuna i vremenskog pritiska poslovanja i potencijala dugoročne koristi od povećane vrijednosti softvera. Odgovarajući izbor strategija očito neće biti iste za sve tvrtke. E-poslovanje koje mora gotovo svakodnevno se mijenjati kako bi preživjelo, usredotočit će se na brzu promjenu, dok vlasnici ugrađenog sustava sa stabilnim zahtjevima, ali sa životnom kritičnim posljedicama neuspjeha mogu se usredotočiti na dugoročnu kvalitetu u svojoj knjizi ističe Booch (1999: 119-121).

Dakle, ovaj odjeljak nije preskriptivan, već samo pokušava identificirati neke od pitanja koja softver menadžer ili glavni arhitekt treba uzeti u obzir. Nažalost, čini se da postoji jako malo objavljene analize koja bi pomogla menadžeru softvera u procijeni troškova i koristi. Čini se da je prioritet istraživanje svake stvarne učinkovitosti.

6.2 Strategije tijekom razvoja

Ključne odluke tijekom razvoja su one koje određuju arhitekturu novog sustava i sastav tima. Te odluke su naravno, međusobno povezane. Kao što je već spomenuto, mnogi slavniji sustavi, kao što su Unix i VME bili su proizvod vrlo malo talentiranih pojedinaca. Savjeti za *unajmljivanje genija* nisu pogrešni, ali je teško pratiti ih u praksi.

U trenutnom stanju, vjerojatno se može malo učiniti da bi se dizajnirala arhitektura koja će omogućiti bilo koju zamislivu promjenu. Međutim, moguće je sustavno rješavati one potencijalne promjene koje mogu biti očekivane, barem općenito. Na primjer, dobro je poznato da su promjene iznimno česte u korisničkim sučeljima na sustavima operativnih sustava i hardvera, dok temeljni podaci i algoritmi relativno stabilniji. Tijekom početnog razvoja grubo prioritetni popis predvidljivih promjena može biti vrlo koristan vodič za arhitektonski dizajn.

Nakon identificiranja mogućih promjena, glavna arhitektonska strategija koja je korištena je skrivanje podataka o onim komponentama ili konstruktima koji se najvjerojatnije mogu promijeniti. Softverski moduli strukturirani su tako da odluke o dizajnu, kao što su izbor određene vrste korisničkog sučelja ili određene operacije sustava, su skrivene unutar jednog malog dijela ukupnog sustava. Ako očekivana promjena postane neophodna u budućnosti, trebalo bi mijenjati samo nekoliko modula.

Pojava objekta orijentiranih jezika u 90im pružila je dodatni mehanizam za projektiranje kako bi se nosili s očekivanim promjenama. Ti jezici pružaju sadržaje poput apstraktnih klasa i sučelja koji se mogu podrazrediti za pružanje novih vrsta objekata koji su zatim korišteni od strane ostatka programa bez izmjena. Dizajneri također mogu iskoristiti objektno orijentirane dizajnerske uzorke od kojih su mnogi namijenjeni omogućavanju fleksibilnosti kako bi se omogućila buduća poboljšanja softvera. Za primjer, Abstract Factory uzorak daje shemu za izradu obitelji srodnih objekata koji međusobno komuniciraju, kao u alatu za korisničko sučelje. Uzorak pokazuje kako se mogu dodati nove klase objekta, pružaju alternativni izgled i dojam, uz minimalnu promjenu postojećeg koda.
U tradicionalnim i objektno orijentiranim sustavima, arhitektonska jasnoća i dosljednost može uvelike olakšati razumijevanje programa. Slične značajke uvijek treba provoditi na isti način, a ako je moguće i slično nazivati komponente. Primjerice, test sustava praćenja pokrivenosti koji je osigurao pokrivanje osnovnih blokova, od odluka i nekoliko vrsta tokova podataka. Kod za svaku vrstu praćenja je pažljivo odvojen u izvorne datoteke, s kodom za prikaz blokova u datoteci pod nazivom bdisp.c, prikaz odluke u odisp.c i slično. Unutar svake datoteke odgovarajuće su funkcije imale isto ime, npr. svaka datoteka ima funkciju prikaz() za rukovanje određenom vrstom prikaza. Ova je dosljednost učinila vrlo lako održavanje pretpostavke gdje će se vjerojatno

nalaziti određeni pojmovi i uvelike ubrzati razumijevanje programske arhitekture, čak i u nedostatku dizajnerske dokumentacije.
S druge strane, nedosljednost u dizajnu i imenovanju može dovesti buduće održavatelje do pogrešnog razumijevanja. Polimorfni objektno orijentirani sustavi su posebno osjetljivi na ovu vrstu pogreške, budući da mnoge članske funkcije mogu imati isto ime. Tako ne mogu biti lako vidljive za održavatelja koji čita kod. Ako ne izvode upravo isti zadatak, ili ako jedna verzija ima nuspojave koje ne dijele druge, tada održavatelja može ozbiljno zavarati čitanje koda.

Najbolji način za postizanje arhitektonske dosljednosti je vjerojatno pustiti osnovni dizajn u rukama vrlo malog tima koji ima vrlo blisku suradnju. Vjerojatno bi se trebalo suzdržati od rotacije osoblja između projekata i biti oprezan u ovoj fazi. Drugi programeri kasnije mogu održavati dosljedan dizajn ako se potiču da prouče strukturu postojećih prije dodavanja vlastitih doprinosa. Konzistentnost se može potom nastaviti provođenjem inspekcije koda.

Ako projektna arhitektura ovisi o kupljenom komercijalnom *off the shelf* komponenti (COTS), kao i mnogi moderni projekti, onda je posebna pažnja potrebita. Prvo, bilo bi očito opasno jako ovisiti o komponenti koja je već u fazama servisiranja ili postupnog ukidanja. Važno je razumjeti pravi status svake komponente, što može zahtijevati informacije koje dobavljač komponente ne želi dati. Drugo, utjecaj mogućih promjena na komponentu COTS treba biti uzeta u obzir. Na primjer, ako su promjene na hardverskoj platformi očekivane da li će COTS dobavljač, po razumnoj cijeni, razviti svoj proizvod za korištenje na novom hardveru? Ako ne, opet bi moglo biti preporučljivo skrivanje informacija kako bi olakšali moguću zamjenu različitih COTS komponenta u budućnosti.

Programska okruženja i softverski alati koji generiraju kod mogu stvoriti probleme slične onima COTS-a. Mnoga takva okruženja pretpostavljaju implicitno da će se sve buduće promijene odvijati unutar okoliša. Generirani kod može biti nerazumljiv za sve praktične svrhe. Nažalost, iskustvo ukazuje da okoline i tvrtke koje ih proizvode, često imaju mnogo kraći životni vijek od sustava razvijenih pomoću istih.

Konačno, postoje dobro poznate kodne prakse koje mogu uvelike olakšati razumijevanje programa i time promjenu softvera. U cjelini 2.4 spominjalo se korištenje IEEE ili ISO standarda, izvršenje kućnog stila kodiranja kako bi se zajamčio jednoličan izgled i komentiranje i odgovarajuću razinu dokumentacije koja odgovara kritičnosti projekta. Jedna tehnika kodiranja koja bi se vjerojatno trebala koristiti češće je umetanje instrumentacija u program kako bi se pomoglo u uklanjanju pogrešaka i razumijevanju budućeg programa. Iskusni programeri koristili su takvu instrumentaciju godinama za snimanje ključnih događaja, međuspremnih poruka i slično. Nažalost, instrumentacija je rijetko propisana, a češće se uvodi samo ad-hoc nakon što je projekt pao u nevolje. Ako se sustavno koristi, može biti velika pomoć za razumijevanje dizajna složenog sustava *as-built*.

Booch (1999: 119-121) napominje da sve gore navedene tehnike uključuju kompromis između evolucije i vremena razvoja. Proučavanju potencijalnih promjena treba vremena i analiza. Dizajn za prilagodbu promjenama može zahtijevati više kompleksni kod koji utječe na vrijeme i kasnije razumijevanje programa.[19] Komponente COTS i programska okruženja mogu uvelike ubrzati početni razvoj, ali s ozbiljnim posljedicama za buduću evoluciju. Konzistentnost dizajna i standarde kodiranja teško je provesti, osim ako se gotovo svi kodovi ne pregledaju, dugotrajno i daleko od opće uporabe. U žurbi da bi dobili proizvod za tržište, menadžer mora paziti na odluke koje žrtvuju dragocjeno vrijeme protiv budućih koristi.

6.3 Strategije tijekom evolucije

Tijekom evolucijske faze, cilj mora biti odgađanje servisiranja sve dok je to moguće očuvanjem čiste arhitekture i olakšavanjem programskog razumijevanja. Kao što je već spomenuto, najsuvremeniji projekti prelaze u evolucijsku fazu s barem nekim ključnim članovima originalnog razvojnog tima. Postoje naravno, okolnosti u kojima to možda nije moguće i prijelaz na novi tim je neizbježan. Ovo je strategija koja riskira gotovo neposredno klizanje u servis zbog poteškoća programskog razumijevanja. Ako je novi tim, međutim, apsolutno neophodan, onda postoje neki koraci koji se mogu poduzeti, kao što je imati neke od razvojnih programera na licu mjesta nekoliko mjeseci

[19] Barem jedan projekt smatrao je poželjnim ukloniti uzorke dizajna koji su uvedeni kako bi se osigurala neiskorištena fleksibilnost

kao bi pomogli u prijelazu. Ako to već nije učinjeno, sustav bi trebao biti postavljen pod kontrolu upravljanja konfiguracijom. Odavde će različiti kupci imati različite verzije i bitno je imati mehanizam praćenja revizije svake datoteke koja je ušla u sastavljanje svake verzije. Bez takvog mehanizma biti će vrlo teško interpretirati izvještaje o problemima koji dolaze s terena. Promjena kontrole može biti formalna ili neformalna ovisno o postupku koji se koristi, ali promjena postupaka kontrole treba biti dobro definirana, a vrlo je poželjno imati jednu osobu koja je označena kao upravitelja konfiguracije, s odgovornošću za kontrolu promjena i verziju praćenja istaknuo je Lehman (1985: 127).

U ovoj fazi, ako ne i ranije, projekt mora imati jasnu strategiju za razumijevanje programa. Ta strategija može uključivati najmanje kombinacije tri elementa:

1. Znanje o timu koje imaju članovi tima koji se obvezuju na dugoročno sudjelovanje u projektu
2. Pisana dokumentacija specifikacija, dizajna, konfiguracija, testovi; ili njegov ekvivalent u spremištu softverskog alata
3. Obrnuti inženjerski alati za oporavak podataka o dizajnu iz samog sustava.

Kako počinje evolucija, sastavim tima može se promijeniti, često s nekim smanjenjem. Ako se voditelj projekta namjerava osloniti uglavnom na timsko znanje za razumijevanje programa, to je dobra točka u kojoj treba napraviti inventar dostupnih znanja i pokušati izbjegavati prekomjerne koncentracije na jednog ili dva člana tima.

Kao što je ranije spomenuto, agilni razvoj metode kao što je ekstremno programiranje često koriste parno programiranje, u kojem dva programera rade zajedno na svakom zadatku. Vjerojatno je nerealno očekivati da novi programeri poduzimaju velike promjene sami, ali moguće ih je staviti da rade u paru s iskusnijim programerom. Poželjno je izbjeći dobro poznati *guru* fenomen, u kojem je jedna osoba jedini stručnjak u važnom dijelu projekta. Guru često nastoji usvojiti taj dio kao svoj teritorij i čini ga teškim za bilo kojega drugoga da radi s tim kodom. Nakon što se ova situacija utvrdi, može biti vrlo teško upravljati. Problem gurua očito je opasno za budućnost projekta i također može imati loš utjecaj na timski moral.

Ako upravitelj odluči naglasiti pisanu dokumentaciju za programsko razumijevanje, onda će doći do znatne revizije i ažuriranje sve takve dokumentacije kako se sustav razvija. Vjerojatno će se dogoditi da je projektna dokumentacija dostupna na početku evolucije sustava kojeg su programeri namjeravali graditi, znatno različita od onoga što je zapravo izgrađeno. Korisna tehnika je inkrementalan redokumentacija, u kojoj se dokumentacija generira ili ažurira kako se moduli mijenjaju. Pripravnik programer može biti dodijeljen za pisanje, na temelju bilješki ili intervjua s iskusnim programerom koji je zapravo napravio modifikaciju, čime se smanjuje trošak i raspored utjecaja redokumentacije.

Menadžer treba uspostaviti zamršenu ravnotežu između programerove želje za restrukturiranjem ili prepisivanjem šifri i ekonomije projekta. Osim ako su sami napisali, programeri će se gotovo uvijek žaliti na kvalitete kodova s kojima se susreću. Neke takve pritužbe jesu zasigurno utemeljene kao kad je restrukturiranje i refaktoring potrebno za održavanje čiste arhitekture. Međutim, ako su svi takvi zahtjevi odobreni, teret kodiranja, nadzora, a osobito testiranja, postat će neodrživ.
Ključne odluke u evolucijskoj fazi odnose se na stvaranje novih verzija i prijelaz na servisiranje. Menadžment bi trebao donositi svjesne odluke o putovima koje treba slijediti, na temelju prosuđivanja o stanju stava i zahtjeva tržišta.

6.4 Strategije tijekom servisiranja

Prijelaz na servisiranje podrazumijeva da će daljnjih promjena biti relativno manje. Važno je shvatiti da je tranzicija u velikoj mjeri nepovratna jer su bitni znanje i arhitektonski integritet vjerojatno izgubljeni. Ako će proizvod biti reinženjeran, vjerojatno je najbolja strategija jednostavno pokušati reproducirati svoje ponašanje u *black-box* umjesto da se proučava i ponovno upotrijebi njegov trenutno kod ili dizajn.

Često se može dogoditi prijelaz na novi tim za održavanje kad servis počne. Očekivanja o tome što može postići takav tim bi trebala biti skromna da bi se izbjegle nemoguće obveze. Menadžment konfiguracija mora nastaviti s radom kako bi se moglo razumjeti izvješća sa terena. Strategije poput oportunističke redokumentacije i dalje mogu biti poželjne, ali popravci koji degradiraju kod mogu biti tolerirane zbog toga jer je osnovna strategija smanjivanje troškova uz zadržavanje prihoda u kratkom roku.

Burd (1997: 322-329) ističe kako je servisna faza vrijeme za izradu i provedbu plana za *phase-out*. Putevi migracija na novu verziju u razvoju trebali bi biti pruženi. Glavno pitanje je često potreba da se oporave i reformiraju vitalni organizacijski podaci kako bi se mogli koristiti u novoj verziji.

ZAKLJUČAK

U ovom radu, tvrdi se da softverski razvoj nudi perspektivu koja se razlikuje od tradicionalnih pogleda na održavanje softvera: ona upućuje na ideju bitnih promjena u okolišu, prirodno obuhvaća koncepte planiranih i neplaniranih pojava i njezina studija nudi uvid u pitanja znanosti i inženjeringa.

Istraživanje razvoja softvera i dalje je mlado polje, te mijenja svoj fokus, pa čak i temeljne koncepte. Znamo da se softver mora razvijati, ali mi još uvijek učimo kako modelirati ovu evoluciju, osobito u sve većem broju složenih okruženja u kojima je softver dizajniran i postavljen. Postoji mnogo otvorenih pitanja koja trebaju biti istražena i nekoliko izazova u istraživanju leži na putu naprijed, uključujući: probleme izgradnje modela i empirijskih studija, primjenjivost studiranja razvoja otvorenog koda, modeliranje novog dizajna, poboljšanje kolektivne memorije programera kroz bolje povezivanje artefakata , pojava softverskih eko sfera, te proučavanje i modeliranje gospodarskih kompromisa i rizika.

Karakteristika softverskog sustava za laku prilagodbu promjenama tijekom vremena naziva se softverska evolucija. Ova značajka omogućuje softverskim sustavima uključivanje novih zahtjeva kako bi se zadovoljili poslovni ciljevi dioničara. U razvoju softvera naglasak je na očuvanju i poboljšanju kvalitete softvera i smanjenju složenosti. Zakoni softverske evolucije sada su glavni igrač u području softverskog inženjeringa i računalnih znanosti. Postoji potreba za povećanjem svijesti o važnosti evolucije softvera budući da je istraživanje evolucije još uvijek mlado polje. Veću pozornost treba posvetiti područjima kao što su razvoj softvera otvorenog koda i neočekivani razvoj softvera.

Razvoj softvera i evolucija može se smatrati integriranim, iterativnim procesom. Proces evolucije softvera potaknut je zahtjevima za promjenama i uključuje analizu učinaka promjena, planiranje izdavanja i promjenu provedbe. Za većinu sustava troškovi održavanja softvera obično nadilaze troškove razvoja softvera. Veliki izazov za održavanje softvera: ugraditi nove (i neočekivane) zahtjeve u softver brzo i jednostavno. Očekuje se da će razvoj softvera biti u središtu programskog inženjeringa. Razvoj softvera treba biti adresiran kao poslovno pitanje kao i pitanje tehnologije - u osnovi interdisciplinarni. Dugoročni pogled na razvoj softvera temelji se na modelu usluge a ne modelu proizvoda.

LITERATURA

1. Arhitektura Open VME, publikacija ICL ref. 55480001, od ICL, Cavendish Rd., Stevenage, Herts, UK SG1 2DY, 1994.
2. Baxter, D .; Yahin, A .; Moura, L .; Sant "Anna, M .; Bier, L. Clone Detekcija pomoću abstraktnih stabala, IEEE međunarodna konferencija za održavanje softvera, 1998,
3. Beck, Kent, Prihvaćanje promjena s ekstremnim programiranjem, IEEE Computer, sv. 32, br. 10, listopad 1999. godine.
4. Bennett K.H., Software evolution and the staged model of the software lifecycle, Research institute for software evolution, University of Durham, UK
5. Boehm, B.W. Spiralni model razvoja i poboljšanja softvera,IEEE Computer, svibanj 1988
6. Booch, G. Razvijanje budućnosti, komunikacije ACM, sv. 44, br. 3, ožujak 2001,
7. Brooks, F. Mitski muški mjesec: Esej o softver inženjeringu, Addison Wesley Publishing Company; ISBN: 0201835959
8. Burd, E .; Munro, M. Istraživanje utjecaja održavanja replikacije koda, IEEE međunarodna konferencija o održavanju softveru, 1997
9. Cusumano, M.A .; Selby, R.W. Microsoft Secrets, HarperCollins, 1997, ISBN: 0006387780
10. IEEE Standardni rječnik terminologije softverskog inženjerstva, standard IEEE Std 610.12-1990, dostupan od IEEE, Los Alamitos.
11. IEEE Softversko inženjerstvo - prikupljanje IEEE standarda New York: IEEE, Izdanje 1994 ISBN / ISSN 155937442X.
12. Informacijska tehnologija - procesi životnog ciklusa softvera. Internacionalna organizacija standarda. ISO12207, Ženeva, Švicarska, 1995
13. Jacobson, I .; Booch, G .; Rumbaugh, J. Ujedinjeni proces razvoja softvera, Addison Wesley, 1999.
14. Johnson, J.H. Podudaranje substringova za detekciju kloniranja i promjenu praćenja, Proc. IEEE Međunarodna konferencija o održavanju softvera,Victoria, Kanada, rujan 1994.,
15. Lagu, B .; Proulx, D .; Mayrand, J .; Merlo, E.M .; Hudepohl, J. Procjenjujući prednosti uključivanja funkcije otkrivanja klonova u razvoju procesa, IEEE međunarodna konferencija o održavanju softvera, 1997

16. Lehman M. M. Evolucija programa. Akademski tisak, London. 1985
17. Schach S. R., klasično i objektno orijentirano inženjerstvo softvera s UML-om I C++, 4. izdanje, McGraw-Hill, 1999.

SAŽETAK

Softver je širok pojam za jedinstveno shvaćanje njegova korištenja, funkcija i ostalih povezanih činjenica.
Kod održavanja softvera, mora se voditi računa o pravovremenom ažuriranju, odnosno držanju koraka s korisničkim zahtjevima, promjenama poslovnog okruženja, napretka hardvera i slično. To se naziva održavanje ili evolucija softvera.
Kada se malo bolje pogleda, softver je zaista složena komponenta kojoj se mora „posvetiti“ pažnje, jer ukoliko ne funkcionira, određeni posao se ne može obaviti. Zato se mora na vrijeme uočiti mogući problemi kako rad ne bi stao, odnosno, bitno je voditi računa o arhitekturi sustava, izdanju softvera kao i njegovom razvoju, također o propadanju softvera kao i procesima koje prate zahtjeve i planiranje te razumijevanje softvera koje vodi do održavanja njegove vrijednosti tokom njegovog životnog procesa.

Ključne riječi: softver, održavanje, životni ciklus

ABSTRACT

Software is a broad term for a unique understanding of it is use, function, and other related facts.

When we speak of software maintenance, account needs to be taken of timely updating, in essence, maintaining user-friendliest steps, business environment changes, hardware progress, etc. That is called software maintenance or evolution.

When you look a bit better, software is really a complex component that needs to be „devoted to “, because if it does not work, a certain job cannot be done. Therefore, it is necessary to determine the possible problems in the time to come, that is important to keep in the mind the system architecture, the software release and it is development, as well as software degradation as well as processes that follow the requirements, planning and understanding of the software that leads to maintaining it is value for his life process.

Keywords: software, software maintenance, life process