Održavanje softvera: kvalitetna implementacija izmjena u softveru

Kristina Blašković
Sveučilište u Rijeci, Odjel Informatike, Rijeka, Hrvatska
kblaskovic@student.uniri.hr

Sažetak – Tijekom razvoja softvera prate se metodologije razvoja koje propisuju niz razvojnih aktivnosti i faza. Razvoj završava isporukom softvera korisniku, nakon čega započinje produkcijski rad, a softver ulazi u fazu održavanja. Okolina u kojoj se softver koristi s vremenom se mijenja, a javljaju se i nove potrebe korisnika. Kako bi pratio potrebe korisnika, softver se i tijekom održavanja, mora prilagoditi promjenjivoj okolini i u njega je potrebno ugraditi tražene izmjene, bilo da se radi o ispravku pogreške ili uvođenju nove funkcionalnosti. Podršku opisanim aktivnostima daje disciplina upravljanje konfiguracijama softvera.

I. UVOD

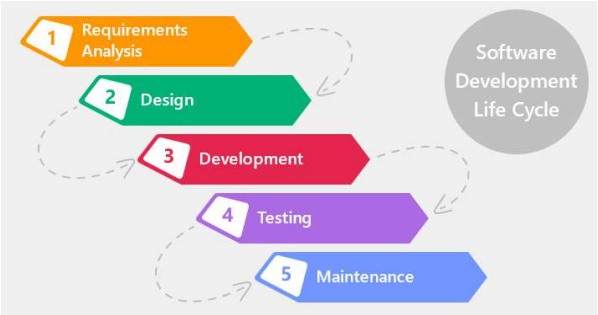
Softver danas zauzima vrlo važnu ulogu u poslovanju, znanosti i inženjerstvu [1]. Dio je različitih sustava koji podižu kvalitetu naše svakodnevice. Pojam softvera podrazumijeva programski kod, podatke te svu popratnu dokumentaciju. Razvijaju ga stručnjaci koje zovemo *programski inženjeri (engl. software engineers)*, a o kvaliteti razvoja i održavanja brine se disiplina *programsko inženjerstvo (engl. software engineering)*. Potreba za disciplineranim pristupom razvoju i održavanju softvera pokazala se nužnom već 70-ih godina prošlog stoljeća kada su primijećeni problemi vezani uz razvoj softvera kao što su: kašnjenje odnosno probijanje rokova isporuke softvera, neplanirano povećanje troškova, oskudna dokumentacija i dizajn te odstupanje krajnjeg produkta od očekivanja korisnika [3].

Korisnici softvera u svom radu često dolaze u situacije koje zahtijevaju uvođenje izmjena i poboljšanja [8]. Tada je potrebno pokrenuti aktivnosti koje će osigurati očuvanje integriteta softvera. Navedene aktivnosti spadaju u fazu održavanja, koja je najdulja u životnom ciklusu softvera te iziskuje mnogo vremena i dodatnih troškova.

Ovaj rad će se osvrnuti na mehanizme koji osiguravaju kvalitetan razvoj i održavanje modernih, kompleksnih softverskih proizvoda.

II. FAZE ŽIVOTNOG CIKLUSA SOFTVERA

Podizanjem svijesti o potrebi efikasnog i kvalitetnog razvoja softvera počinju se koristiti tehnike bazirane na teorijskim osnovama postavljenim u programskom inženjerstvu [5].

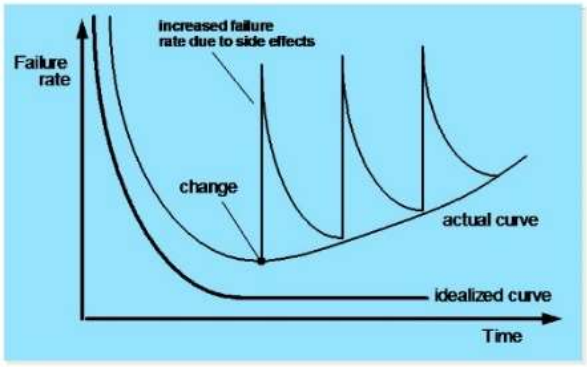


Slika 1 Životni ciklus softvera (Waterfall model) [15]

Prema tradicionalnom Waterfall modelu razvoja, koji je prvi predstavio Royce 1970. godine, životni ciklus softvera sastoji se od 5 faza (slika 1) [1]. Sve započinje postavljanjem i definiranjem korisničkih zahtjeva, koje je važno dobro dokumentirati. Na temelju korisničkih zahtjeva dizajnira se model sustava te se može započeti s programiranjem. U toj fazi razvoja bitno je izraditi čitljiv kod i dobro ga dokumentirati kako bi bio jednostavan za nadogradnju i održavanje. U suprotnom dobivamo nekvalitetan softver koji je vrlo teško održavati i nadograđivati. Po završetku programiranja te nakon što je razvojni tim proveo inicijalna testiranja, softver se predaje korisniku na testiranje i sada korisnik dobiva uvid u razvijeni proizvod. Nakon testiranja dobivamo pozitivnu ili negativnu povratnu informaciju, a kada je korisnik zadovoljan i smatra da su svi zahtjevi zadovoljeni softver započinje s produkcijskim radom. Od tog trenutka do kraja životnog ciklusa softvera programski inženjeri zaduženi su za održavanje. Brinu se o otklanjanju novonastalih ili naknadno primijećenih pogrešaka te o implementaciji izmjena.

Održavanje fizičkih sustava (npr. hardvera) zahtijeva zamjenu istrošenih ili neispravnih dijelova sa svrhom otklanjanja kvarova [6]. Softver pak smatramo logičkom komponentom koja se ne može istrošiti. Prilikom inicijalnih testiranja javljaju se pogreške u radu (slika 2). Prelaskom u fazu održavanja, sustav bi teoretski mogao beskonačno ispravno raditi, što je prikazano idealnom krivuljom (*engl. idealized curve*). Međutim, korisnički zahtjevi za izmjenama ponovno pokreću sve razvojne aktivnosti. Svaka izmjena prolazi kroz sve faze životnog

ciklusa i po isporuci ponovno dolazi do grešaka u radu. Kao rezultat dobiva se stvarna krivulja *(engl. actual curve).*



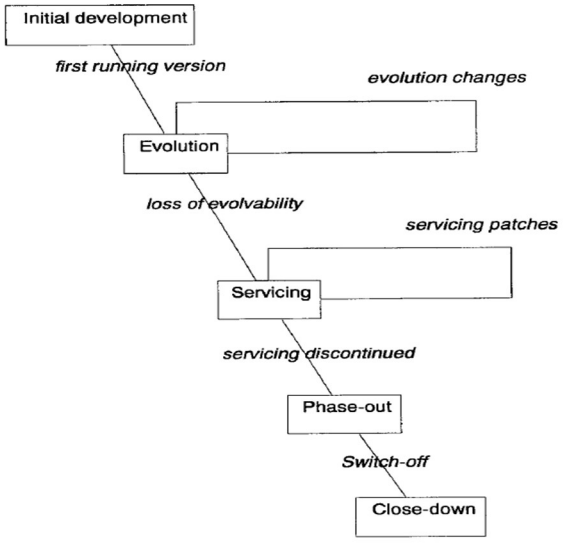
Slika 2 Učestalost grešaka u radu softvarea [1]

III. EVOLUCIJA I ODRŽAVANJE SOFTVERA

U klasičnom Waterfall modelu razvoja održavanje je zadnja faza u životnom ciklusu softvera [5], [6]. Prema SWEBOOK-u (engl. Software Engineering Body and Knowledge) ono je definirano kao skup svih aktivnosti koje su potpora softveru po najnižoj cijeni [24]. Neke se aktivnosti pokreću prilikom inicijalnog razvoja ali većina slijedi nakon isporuke. Kako bi opisali „rast" softvera znanstvenici uvode novi koncept evolucije, koji se razlikuje od održavanja i označava kontinuiranu promjenu stanja softvera iz lošijeg, jednostavnijeg u bolje [6]. Softver koji se primjenjuje na stvarnim problemima podložan je stalnim promjenama i nadogradnjama kako bi se prilagodio okolini ili postaje neprimjenjiv [25]. Vodeći istraživač u području evolucije softvera, Lehman, ovakav tip softvera svrstava u E-tip softvera (*engl. E-type software)* [16].

Zbog nefleksibilnosti klasičnih modela i činjenice da se korisnički zahtjevi mijenjaju tijekom cijelog životnog ciklusa, počinju se razvijati novi evolucijski modeli [6]. Za razliku od tradicionalnih uvjerenja prema kojima je održavanje samo faza u životnom ciklusu softvera, nova ideja je da održavanje ima vlastiti životni ciklus sa sljedećim fazama: razumijevanje koda, izmjena koda te ponovna provjera.

Staged model je jedan primjer evolucijskog modela koji uzima u obzir starenje softvera (slika 3) [8]. *Inicijalna faza* je razvoj prve produkcijske verzije softvera. Razvojem softvera tim developera dobiva potrebna znanja za nastavak evolucije softvera. Ukoliko je poznavanje sustava narušeno odlaskom ključnih stručnjaka, narušava se i integritet softvera. Uspješnom implementacijom produkcijske verzije prelazimo u *fazu evolucije*. Tijekom nje se u softver ugrađuju izmjene softvera sa svrhom prilagodbe konstantnim novim zahtjevima korisnika. Dok postoji razvojni tim koji se može nositi s velikim izmjenama i doradama softvera, traje faza evolucije. Međutim, kako softver stari, tako se mijenja sastav tima za održavanje. Članovi tima koji jako dobro poznaju sustav odlaze i umjesto njih dolaze novi manje iskusni. Softver tada ulazi u *fazu servisiranja* tijekom koje se samo otklanjaju uočene greške, bez ikakvih nadogradnji. Cilj je implementirati izmjene uz minimalne troškove i raspoloživ tim developera. Kada niti minimalno održavanje softvera nije više isplativo, sustav ulazi u posljednju *fazu napuštanja*. Iako se nikakve izmjene više ne implementiraju, softver može i dalje biti u produkciji, ali je krajnji cilj zamijeniti ga novim.



Slika 3 Evolucijski Staged model procesa životnog cilusa softvera [8]

A. Implementacija izmjena

Izmjena softvera je osnovna aktivnost kako faze evolucije tako i faze održavanja [8]. Ona uvodi ili nove izmjene u postojeći sustav, ili mijenja sustav kako bi odgovarao zahtjevima, ili migrira sustav u novu radnu okolinu. U kasnim 70-tim godinama predstavljen je c*hange mini-cycle* model procesa uvođenja izmjena, koji naglašava da izmjena jednog objekta utječe na druge objekte [23], [6]. Kada stigne zahtjev za izmjenu potrebno ju je analizirati i planirati, a potom slijedi implementacija, verifikacija i validacija te na koncu dokumentiranje.

Dobro razumijevanje sustava je ključno za implementaciju izmjena [6]. U ovom procesu programer stvara mentalne modele sustava na temelju hipoteza te formira određenu hipotezu koju verificira čitanjem koda. Međutim, ta hipoteza ne mora biti točna ukoliko programer nije u potpunosti razumio program. Kontinuiranim formuliranjem novih hipoteza razumijevanje se povećava.

Identifikacija dijelova softvera na koje izmjena može djelovati vrlo je važan zadatak u planiranju, implementaciji i praćenju izmjena [6]. U ovom procesu programer vrednuje utjecaj izmjene, njen trošak i isplativost. Za analizu se koriste sljedeće tehnike:

a) *Sljedivost (engl. traceability)*
   Identifikacija artefakata involviranih u izmjenu i generiranje modela u kojem su ti artefakti međusobno povezani. Na ovaj način jednostavno se izoliraju artefakti u koje je potrebno implementirati izmjenu.
b) *Zavisnost (engl. dependency)*
   Ocjenjivanje utjecaja izmjene na temelju semantičke zavisnosti. Ova tehnika poznata je i kao analiza izvornog koda *(engl. source code analisys)*.

Kako bismo dobili informaciju koje su izmjene učinjene i gdje, korisno je izmjeriti tzv. efekt mreškanja *(engl. ripple effect)* [22], [26] odnosno pratiti posljedice u kodu nakon izmjene te tako dobiti informaciju o utjecaj izmjene na povećanje ili smanjenje kompleksnosti određenog modula ili cijelog sustava.

Kada se jedan objekt izmijeni mogu nastati dodatne greške u interakciji s drugim objektima [6]. Tada je potrebno izmjene provesti i na susjednim objektima, onima na koje navedena izmjena utječe, a one mogu izazvati daljnje pogreške. Ovaj proces zove se širenje izmjene *(engl. change propagation)*.

Mnogi sustavi postaju sve kompleksniji i skuplji za održavanje dodavanjem novih funkcionalnosti [6]. *Refaktorizacija* ili *restrukturacija* označava izmjenu strukture softvera bez utjecaja na njegovo ponašanje zbog lakšeg održavanja. To postižemo uklanjanjem duplog koda, pojednostavljenjem koda,... Važno je naglasiti da ovakve izmjene ne rezultiraju evolucijom softvera.

B. Podrška procesu implementacije izmjena

Kao podrška procesu implementacije izmjena mogu se koristiti sustavi za praćenje grešaka *(engl. bug tracking systems)* [10]. Oni služe za prijavu i proslježivanje zahtjeva za izmjenu. Kada se primijeti pogreška ona se prijavljuje i prosljeđuje odgovornoj osobi. Zaduženi developer, u timu, prihvaća zahtjev, rješava ga i zatvara. Zahtjev *(engl. change request, CR)* može biti ili zahtjev za uklanjanje pogreške ili zahtjev za poboljšanjem. Svaki se zahtjev sprema u sustav zajedno s atributima koji ga opisuju (jedinstvenim identifikatorom, datumom kreiranja, kratkim i dugim opisom, datumom zadnje izmjene statusa, imenom proizvoda, imenom komponente, informacijom o tome tko je kreirao zahtjev i kada te kome je zahtjev proslijeđen). Kod open source projekata, s obzirom na specifični sustav proslježivanja zahtjeva odgovornim osobama i proces odobrenja predloženih zahtjeva, bilježe se i dodatne informacije o zahtjevu. Povezivanjem ovih informacija s informacijama iz sustava za kontrolu verzija *(engl. version control system)*, koji spremaju razlike između trenutne i prethodnih verzija datoteke u repozitorij, dobivaju se vrijedni podaci za analizu. Svaka datoteka evoluira kroz niz revizija koje se spremaju zajedno s jedinstvenim identifikatorom developera zaduženog za izmjenu, datumom izmjene te kratkim opisom ili komentarom. Budući da ova dva sustava obično nisu u korelaciji, uobičajena praksa je da se kod spremanja u repozitorij u komentaru doda jedinstveni identifikator zahtjeva iz sustava za praćenje grešaka. Na taj način dobivamo vezu između izmjene i zahtjeva koji ju je pokrenuo. Prilikom kreiranja novog zahtjeva obično se popunjava kratki i/ili dugi opis problema gdje developer mora naznačiti datoteke koje je potrebno izmijeniti. Ukoliko se developer susretao već s ovakvim zahtjevima, on će jednostavno locirati potrebne datoteke i implementirati izmjene. Međutim, ukoliko je zahtjev novi ili se radi o neiskusnom developeru, on će morati bolje i dublje analizirati problem ili potražiti pomoć, što se također može zabilježiti.

Canfora i Cerulo svoj rad [10] temelje na hipotezi da se iz komentara revizija datoteka i zahtjeva koji su utjecali na stvaranje istih može predvidjeti utjecaj budućih izmjena. Svoje istraživanje proveli su na open source projektima. Smatraju da se praćenjem sličnih

zahtjeva, koji su uspješno riješeni, može dobiti rang listu datoteka na koje će najvjerojatnije utjecati ta izmjena. Za dobivanje rang liste novi zahtjev prolazi kroz niz koraka algoritma. U prvom koraku ekstrahira se opis, odnosno iz zahtjeva se izvlače tekstualne informacije. Nadalje se ekstrahirani opis podijeli na riječi. Pritom niz alfanumeričkih znakova odvojenih nealfanumeričkim znakom predstavlja jednu riječ. Kako bi različite oblike iste riječi (npr. jednina i množina) svrstali pod isti pojam potrebno je svaku riječ transformirati u njen izvorni oblik. U posljednjem koraku od dobivenih riječi u prethodnim koracima postavlja se upit. Kroz isti set koraka prolaze i stari riješeni zahtjevi i kao rezultat dobivamo XML datoteku. Kako bi dobili rangiranu listu upotrijebili su probabilistički pristup. Neka imamo skup pojmova $\{t_1, t_2, \dots, t_n\}$, a upit q i XML datoteka d imaju vektorsku reprezentaciju $(x_1, x_2, \dots, x_n)$ gdje je $x_i = 1$ ukoliko se t_i nalazi u upitu/datoteci, a u suprotnom $x_i = 0$. Na temelju dobivenih vjerojatnosti vrednujemo da li je XML dokument d relevantan za upit q i tako dobivamo rang listu.

Moderni veliki softverski sustavi zahtijevaju stručnjake iz različitih područja (npr. programiranje, baze podataka, IT stručnjaci,...), koji su često na različitim lokacijama [9]. Uvođenjem inteligentnog aktivnog agenta, koji bi automatski klasificirao i prosljeđivao dospjele zahtjeve, povećala bi se efikasnost održavanja softvera i smanjili bi se troškovi. U tom procesu klasifikacije zahtjevi se kategoriziraju te se tim stručnjaka zadužuje za odgovarajuću kategoriju. Proces teče tako da korisnici, kada primijete nepravilnost u radu softvera, generiraju zahtjev s opisom problema. Na tako kreirani zahtjev primjenjuju se metode strojnog učenja za prosljeđivanje odgovarajućoj grupi stručnjaka bez ljudske intervencije. Opis zahtjeva treba biti pročišćen da se dobiju samo ključne riječi u njihovom izvornom obliku. Uspoređivanjem dobivenih podataka s podacima iz baze znanja, stvorene temeljem prethodnih zahtjeva, novi zahtjev se klasificira. Di Luca, Di Penta i Gradara u svom radu [9], osim ranije spomenutog probabilističkog modela za klasifikaciju predlažu još i sljedeće pristupe:

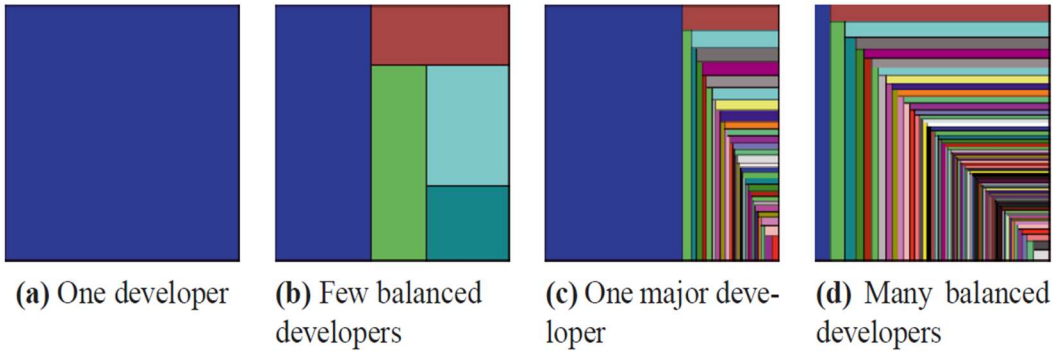
a) Model vektorskog prostora (engl. Vector Space model)

U ovom pristupu novi zahtjevi su dokumenti koji su klasificirani kao nadolazeći zahtjevi za održavanje, a upiti su skupovi pojmova povezani s određenom kategorijom koji tvore bazu znanja [21]. Dokumenti i upiti imaju vektorsku reprezentaciju. Komponente vektora određuju se tako da ako imamo veliki skup svih mogućih pojmova te se na i-tom mjestu pojavljuje traženi pojam, u najjednostavnijem obliku označavanja, i-ta pozicija odgovara 1, a u suprotnom 0. U procesu klasifikacije nadolazećeg zahtjeva dokument se uspoređuje sa svakim upitom iz baze znanja putem formule za najmanju udaljenost dvaju vektora. Kada pronađemo upit s najmanjoj udaljenosti dokument klasificiramo u danu kategoriju

b) Stroj s potpornim vektorima (engl. Support Vector Machine, SVM), Klasifikacijska i regresijska stabla (engl. Classification and Regression Trees, CART) i k-najbližih susjeda (k-nearest neighbor, KNN)

Ovo su algoritmi za klasifikaciju koji se koriste kod strojnog učenja [27]. Imamo ponovno rječnik odnosno skup svih mogućih pojmova. Zahtjevi su reprezentirani u obliku matrice gdje redovi predstavljaju zahtjeve, a stupci riječi iz rječnika. Kod KNN algoritma imamo testni skup u kojem svaki zahtjev ima oznaku da li se pojam pojavljuje u zahtjevu i klasu kojoj pripada. Kada pristigne novi zahtjev ukoliko se riječ iz rječnika pojavljuje u opisu zahtjeva na tom mjestu imamo vrijednost 1, a u suprotnom 0. Nakon toga se neklasificirani skup uspoređuje sa svakim klasificiranim zahtjevom iz testnog skupa na način da se odredi najmanja udaljenost putem formule za određivanje najmanje udaljenosti točaka u Euklidskom prostoru. Dobivene udaljenosti se poredaju silazno te za određenu vrijednost *k* algoritam odabire *k* kandidata. Klasa koju ima većina kandidata pridodijeliti će se dolaznom zahtjevu. CART algoritam je najčešće korišten algoritam za klasifikaciju. Od rječnika pojmova tvorimo stablo tako da su pojmovi čvorovi odluke, a krajnji čvorovi odnosno listovi su klase. Dolazni skup koji se sastoji od pojmova uspoređujemo s čvorovima oduke. Ukoliko se traženi pojam nalazi u opisu dolaznog zahtjeva idemo u granu s vrijednosti 1, a u suprotnom u granu s vrijednosti 0. Kada stignemo do listova stabla dobivamo klasu pripadnosti za zahtjev. SVM je novija metoda klasifikacije. Početni skup se sastoji od *n* elemenata. Elementi su uređeni parovi pojma i oznake pojavljivanja 1 ili -1. Kako imamo *n*-dimenzionalni prostor u kojem su prikazani zahtjevi polaznog skupa algoritam konstruira hiperravninu koja razdvaja zahtjeve. Novi nadolazeći se klasificiraju na način da se odredi njihov položaj u odnosu na hiperravninu. Ukoliko klase ne možemo linearno razdvojiti uvodimo jezgrenu funkciju *(engl. kernel)*.

Pojam reinženjering podrazumijeva proces generiranja novog sustava na temelju postojećeg [6], [11]. U tom procesu se analizira postojeći softverski sustav u svrhu podrške održavanju i evoluciji softvera kroz poboljšano razumijevanje. U tradicionalnom smislu znanje o softveru prikupljalo se kroz analizu programskog koda i dokumentacije, što zahtijeva puno vremena te je u nekim slučajevima problem nemoguće riješiti bez komunikacije s originalnim developerom. Ove važne informacije pohranjene su u repozitoriju sustava za kontrolu verzija bez kojih je danas razvoj i održavanje nezamislivo. Uz mogućnost upravljanja softverskim artefaktima (kroz identifikaciju dijelova softvera) i izmjenama (kroz praćenje nastalih izmjena na artefaktima), oni podupiru timski rad tako da omogućavaju konkurentan razvoj. Sustav obično sadrži centraliziran repozitorij softverskih artefakata, a korisnici mogu pristupati repozitoriju za prikupljanje



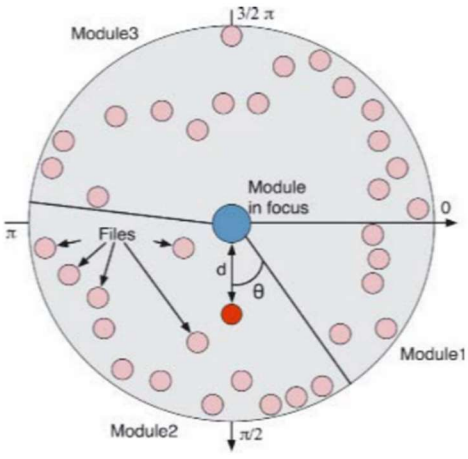
Slika 4 Razvojni uzorci temeljeni na fraktalnim figurama [18]

potrebnih informacija. Svaki korisnik može povući artefakt iz repozitorija u svoj osobni radni prostor *(engl. workspace) check-out* operacijom, napraviti izmjene te spremiti izmjene *(engl. commit)* postavljanjem artefakta u repozitorij *check-in* operacijom. Nakon svake spremljene izmjene sustav kreira novu verziju artefakta zajedno sa svim informacijama vezanim za izmjenu. Korisnici mogu pristupati bilo kojoj verziji artefakta, usporediti dvije odabrane verzije te generirati različite izvještaje za pregled povijesti. Znanstvenici su uočili da se u repozitorijima obično nalazi velika količina podataka. Uvođenjem vizualizacijskih tehnika mogla bi se poboljšati preglednost te olakšati pristup potrebnim verzijama. Vizualizacija softvera bavi se grafičkom reprezentacijom softverskih artefakata [13]. Ove tehnike prikaza prigodne su za analizu i razumijevanje velikog kompleksnog skupa podataka.

Vizualizacijske tehnike mogu biti korisne kod višejezičnih sustava *(engl. multi-language, ML)* [12]. Naime, zbog mnogih prednosti, kao što su olakšani pristup bazama podataka te olakšano ponovno korištenje *(engl. software reuse),* raste popularnost sustava gdje programi pisani u jednom programskom jeziku pozivaju funkciju ili pristupaju dijelovima koda pisanim u drugom programskom jeziku. Održavanje ovakvih sustava je otežano jer podrazumijeva detekciju opisanih međuzavisnosti te poznavanje svih zastupljenih programskih jezika. Uvođenje vizualizacijskih tehnika doprinijelo bi olakšanom održavanju.

C. Praćenje povijesti izmjena

Analizom evolucije softvera pratimo izmjene te njihov uzrok i utjecaj [7]. Softverski kod zajedno s informacijama o izmjenama, prijavom grešaka i izlaznim podacima čine povijest isporučenih verzija (*engl. release history*). Povezivanjem informacija o izmjenama, koje obično dobivamo kao *log* datoteke sustava za verzioniranje, i zahtjevima za izmjenu ili ispravak grešaka stvara se baza povijesti isporučenih verzija *(engl. Release History Data Base, RHDB)*. Pritom jedna verzija može imati jedan ili više zahtjeva, ali se i jedan zahtjev može odnositi na jednu ili više verzija. Na dobivenoj bazi podataka možemo raditi različite analize. Iz sustava za verzioniranje znamo koji developer je radio, u kojem udjelu i na kojoj verziji softvera. Primjenom fraktalnih figura (engl. Fractal figure) vizualizacije opisanih u [18], [20] kumulativni podaci mogu se pregledno prikazati tako da se rad svakog developera prikaže pomoću pravokutnika različite boje, čija je veličina proporcionalna udjelu kreiranih verzija u ukupnom broju verzija softvera. Tako možemo odmah vidjeti da li je za razvoj bio zadužen uglavnom jedan developer ili je više developera doprinijelo u različitim udjelima (slika 5).



Slika 5 Prikaz interaktivne vizualizacijske tehnike Evolution Radar

Pomoću interaktivne vizualizacijske tehnike evolucijskog radara *(engl. evolution Radar)* [17], [19] za analizu povezanih izmjena, moguće je analizirati i kako izmjena jednog artefakta utječe na izmjenu drugog. Tehnika podrazumijeva da jedan modul postavimo u sredinu dok su ostali moduli, koji se mijenjaju zajedno s njime, raspoređeni ravnomjerno po površini kruga. Udaljenost pojedinog modula od središnjeg modula je proporcionalna mjeri u kojoj izmjena središnjeg modula utječe na izmjenu pojedinog modula (slika 6).

IV. UPRAVLJANJE KONFIGURACIJOM SOFTVERA

Kod malih sustava upravljanje izmjenama nije problem jer je za njihov razvoj dovoljan jedan programski inženjer ili mali tim [1]. Kako sustavi dobivaju na kompleksnosti, povećava se broj programskih inženjera i javlja se potreba za uspostavljanjem mehanizama kontrole timskog razvoja softvera.

Upravljanje konfiguracijom softvera *(engl. Software Configuration Management, SCM)* je disciplina koja se brine o evoluciji kompleksnih softvera [14]. Kvalitetna implementacija izmjena, u ovakvim sustavima, osigurana je kroz slijed aktivnosti: *identifikacija* objekata sustava koji se mogu mijenjati, *kontrola* izmjena tih objekata kroz čitav životni ciklus, *praćenje statusa* objekata te zahtjeva za izmjenu i *vrednovanje* stupnja završenosti i točnosti izmijenjenih objekata. Softverski objekti su artefakti i dokumenti kreirani u tijeku životnog ciklusa softvera. Evoluciju softvera pratimo putem *verzija* objekata, koje označavaju određeno stanje u evoluciji. Softverski objekti se spajaju u konfiguraciju softvera (*engl. software configuration*) tako da se povezuju različite verzije različitih softverskih objekata, za što SCM mora biti podrška.

S obzirom na tip SCM sustava softverske konfiguracije koje evoluiraju mogu biti reprezentirane pomoću skupa verzioniranih dokumenata *(engl. file-based system)* ili pohranjeni u bazi podataka *(engl. database)* tako da korisnik može odabrati tip i veze softverskih objekata [14].

Slijeđenjem propisanih aktivnosti postižemo kvalitetu razvijenog softvera [1], [2], [3]. Upravo zbog toga je potrebno aktivnosti pokrenuti paralelno s pokretanjem razvoja softvera te ih provoditi do završetka njegovog životnog ciklusa. Kroz sustav se prate sve izmjene na svim softverskim objektima te je svakom developeru osigurana zasebna razvojna okolina, gdje može nesmetano raditi na svim verzijama objekata. Spoj odgovarajućih softverskih objekata tvori izgradnju *(engl. build)*. Kvalitetnim dokumentiranjem možemo ponovno kreirati okolinu u kojoj je izgradnja stvorena. Kontinuiranim praćenjem, obavještavanjem i izvještavanjem kroz sustav postavljamo kontrolu nad cijelim razvojnim procesom.

Midha je još 1997. svom radu [4] istaknuo kako bi SCM sustav dodatno morao imati prihvatljivo sučelje na platformi koja se jednostavno može integrirati s razvojnom okolinom, biti jednostavan za administraciju te biti podrška razvoju s udaljenih fizičkih lokacija kroz centraliziran repozitorij softverskih objekata kojem se može pristupiti putem WLAN-a.

V. ZAKLJUČAK

Moderni softveri su kompleksni i podložni stalnim izmjenama. Razvoj se odvija u velikim timovima stručnjaka s različitim vještinama. Pritom razvijeni softver mora zadovoljavati stroge kriterije kvalitete, što se postiže uspostavljanjem mehanizama kontrole u svakoj fazi životnog ciklusa softvera.

Nakon isporuke, softver prelazi u fazu održavanja i evolucije, kada se implementiraju izmjene i poboljšanja na temelju zahtjeva korisnika. Sada je posao stručnjaka da održavaju kvalitetu softvera tako da se prije provođenja izmjena one razumiju, analiziraju, planiraju i dokumentiraju.

Podršku kvalitetnom razvoju i održavanju softvera daju sustavi za praćenje pogrešaka, koji služe za prijavu, distribuciju i praćenje zahtjeva za izmjene, i sustavi za kontrolu verzija, koji vode brigu o verzijama svih softverskih objekata.

Analizom traženih izmjena u softveru, njihove složenosti i zahtjevnosti u smislu vremenskih i ljudskih resursa, te promatranjem podatkovne i procesne složenosti softvera prije i poslije provedenih izmjena, može se evaluirati njihov utjecaj na verziju softvera. To može biti korisno za planiranje faze održavanja te upravljanje ljudima i vremenom prilikom budućih zahtjeva za izmjenama

VI. LITERATURA

[1] R.S. Pressman: „Software engineering: a practitioner's approach“, The McGraww-Hill Companies Inc.,ISBN 73375977, 2010.

[2] D.B. Leblang, P.H. Levine: „Software Configuration Management: Why is it needed and what shod it do?“,ICSE SCM-4 and SCM-5 Workshops, Springer(str.53-60),1995.

[3] E.H. Bersoff, V.D. Henderson, S.G. Siegel: „Software Configuration Management“, ACM SIGMETRICS Performance Evaluation Review ,Vol. 7 Issue 3-4 (str. 9-17) 1978..

[4] A. K. Midha: „Software Configuration Management for 21st Century“, Bell Lab Technical Journal, Volume 2 Issue 1 (str. 154-165) 1997.

[5] T. Mens: „Introduction and Roadmap: History and Chalenges of Software Evolution“, Springer Berlin Heidelberg (str. 1-11), 2008.

[6] P. Tripathy, K. Naik:“Basic Concepts and preliminaries“, Software Evolution and Maintenance: A Practitioner's Approach, First Edition (str. 1-22), 2015.

[7] M. D'Ambros, H.Gall, M. Lanza, M. Pnzger: „Analysing software repositories to understand software evolution“, InSoftware Evolution, Springer Berlin Heidelberg (str. 37-67), 2008.

[8] K.H. Bennett, V.T. Rajlich: „Software Maintenance and Evolution: a Roadmap“, ICSE '00 Precedings of the Conference on the Future of Software Engineering (str. 73-87), 2000.

[9] G.A. Di Lucca, M. Di Penta, S. Gradara: „An Approach to Classify Software Maintenance Requests“, ICSM'02 Proceedings of the International Conference on Software Maintenance (str. 93-102), 2002.

[10] G. Canfora, L. Cerulo: „Impact Analysis by Mining Software and Change Request Repositories“, METRICS 2005 11th IEEE International Software Metrics Symposium (str. 9-17), 2005.

[11] X.Wu, A. Murray, M-A. Storey, R. Lintern, „A Reverse Engineering Approach to Support Software Maintenance: Version Control Knowledge Extraction“, WCRE'04 Preceedings of the 11th Working Conference on Reverse Engeneering (str. 90-99), 2004.

[12] K. Kontogiannis,P. Linos, K. Wong: „Comprehension and Maintenance of Large-Scale Multi-Language Software Applications“, ICSM'06 22dn IEEE International Conference on Software Maintenance (str. 497-500), 2006.

[13] R. Koschke: „Software visualisation in software maintenance, reverse engineering and reengineering: A research survey“, Journal of Software Maintenance and Evolution Research and Practice (str. 87-109), 2003.

[14] B. Westfechtel, R. Conradi: „Software Architecture and Software Configuration Management“, B. Westfechtel, A. Van der Hoek (Eds.): SCM 2001/2003, LNCS 2649, Springer-Verlag Berlin Heidelberg (str. 24-39), 2003.

[15] Gordiyenko, S. Software Development Life Cycle (SDLC). Waterfall Model. XB Software. Objavljeno: 3.11.2014., zadnji pristup: 23.10. 2016.

[16] I. Herraiz, D. Rodriguez, G. Robles, JM.Gonzalez-Barahona: „The evolution of the laws of software evolution: A discussion based on a systematic literature review“, ACM Computing Surveys (CSUR) (str. 28). 2013

[17] M. D'Ambros, M. Lanza: „Reverse engineering with logical coupling“, Working Conf. Reverse Engineering (WCRE), IEEE Computer Society Press (str. 189-198), 2006.

[18] M. D'Ambros, M. Lanza, H. Gall: „Fractal figures: Visualizing development effort for CVS entities“, Proc. Int'l Workshop on Visualizing Software for Understanding (Vissoft), IEEE Computer Society Press (str. 46-51), 2005.

[19] M. D'Ambros, M. Lanza, M. Lungu: „The evolution radar: Intergating fine grained and coarse-grained logical coupling information“, Proc. Int'l Workshop on Minning Software Repositories (MSR) (str. 26-32), 2006.

[20] M. Lanza, S. Ducasse: „Polymetric views – a lightweight visual approach to reverse engineering“, IEEE Trans. Software Engineering (str. 782-795), 2003.

[21] D. Harman: „Ranking algorithms“ In Information Retrieval: Data Structures and Algorithms, Prentice-Hall, Englewood Cliffs (str. 363-392), 1992

[22] S. Black: “Computing ripple effect for sopftware maintanance”, Journal of Software Maintanance and evolution: Research and Practice 13 (str. 263-279), 2001.

[23] S-S. Yau, J. Collofello, T. MacGregor. "RIPPLE EFFECT ANALYSIS OF SOFTWARE MAINTENANCE." COMPSAC '78, IEEE Comput Soc Int Comput Software & Appl Conf, 2nd, Proc (str. 60 – 65), 1978.

[24] A.April, A. Abran: “Software Maintenance Management: Evaluation and Continuous Improvement”, John Waley and Sons, 2012.

[25] J. P. S. Alcocer, A. Bergel, S. Ducasse, M. Denker: ”Performance Evolution Blueprint: Understanding the Impact of Software Evolution on Performance”, 1st IEEE Working Conference on Software Visualization, Eindhoven, Netherlands (str. 1-9), 2013.. IEEE, pp.1-9, 2013

[26] B. Li, X. Sun, H. Leung, S. Zhang: “A survey of code-based change impact analisystechniques”, Software testing, verification and reliability, John Waley and Sons (str. 613-646), 2012.

[27] P. Harrington: “Machine learning in action”, Vol. 5. Greenwich, CT: Manning, 2012.