Održavanje softvera

Seminarski rad u okviru kursa
Metodologija stručnog i naučnog rada
Matematički fakultet

Ana Marković, Milan Radišić, Nemanja Živanović
mi16127@alas.math.rs, mi16192@alas.math.rs, mi16089@alas.math.rs

Maj 2021

Sažetak

Održavanje softvera obuhvata sve modifikacije na softveru nakon njegove isporuke u javnost, sve do trenutka njegovog povlačenja iz upotrebe. Pored namere da se detaljnije predstavi sam termin održavanja softvera, cilj rada jeste i da čitaoci spoznaju značaj ovog dela razvoja softvera, da dobiju uvid u to koji alati im mogu pomoći da olakšaju sam proces, ali i da se upoznaju sa nekim merama koje određuju kvalitet softvera i omogućavaju da se čitav posao olakša.

Sadržaj

1 Uvod 2

2 Koncept održavanja softvera 2

3 Uloga i značaj 2
- 3.1 Zašto održavanje softvera, a ne novi proizvod? 3
- 3.2 Greške u sistemu i održavanje softvera 4

4 Održavanje softvera u razvojnom ciklusu 5
- 4.1 Održavanje softvera u različitim modelima 5

5 Alati koji olakšavaju održavanje softvera 6
- 5.1 Alati za razumevanje programa i obrnuti inženjering 7
- 5.2 Alati za testiranje 7
- 5.3 Alati za konfiguracioni menadžment 8
- 5.4 Alati za održavanje dokumentacije i metriku 8

6 Atributi kvaliteta softvera koji utiču na olakšavanje održavanja softvera 8

7 Zaključak 11

Literatura 11

A Dodatak 13

1 Uvod

Adaptabilnost je sposobnost prilagođavanja različitim uslovima ili okolnostima [7]. Upravo je adaptibilnost srž i osnova ideje održavanja softvera. Kao što su se ljudi kroz istoriju prilagođavali vremenskim dobima i gradili kuće od boljeg materijala i sa kvalitetnijom izolacijom ili pravili topliju odeću zbog teških zima, tako i softver, da bi preživeo i bivao aktuelan, takođe mora da bude sposoban da bude modifikovan. Potrebe naručioca mogu da se menjaju s vremenom, čak i iz dana u dan, i ako neki softver nije sposoban da to isprati, pitanje je da li će ostati konkurentan na tržištu. Stoga je ova tema vrlo aktuelna u današnjem dobu eksplozije softvera.

2 Koncept održavanja softvera

Održavanje softvera se može definisati kao disciplina koja se bavi promenama nekog softverskog sistema nakon njegovog puštanja u rad [12]. Intuitivno isprva možemo pomisliti da se tu radi prevashodno o promenama korektivne prirode (ispravljanje bagova). Međutim, okolnosti u kojima se koristi konkretan softver neprestano evoluiraju, razvijaju se postojeće softverske i hardverske tehnologije i pojavljuju nove. Vremenom se menja poslovna dinamika, pa i trendovi u samom vizuelnom dizajnu grafičkih interfejsa. Sve ove promene nalažu da postojeći softverski sistem ostaje u koraku sa svetom, zadovoljavajući zahteve za boljim operativnim performansama i novim funkcionalnim mogućnostima koje postavljaju korisnici.

Međunarodni standard ISO/IEC 14764 svaki zahtev za modifikacijom softvera nakon njegove isporuke klasifikuje ili u korektivnu kategoriju, ili u kategoriju unapređenja (eng. *enhancement*). Zahtev se na dalje identifikuje kao korektivan, preventivan, adaptivan ili perfektivan [13].

Korekcije su ispravke trenutno prisutnih bagova, dok se **preventivne** modifikacije tiču promena na mestima u dizajnu koja bi kasnije mogla postati izvor operativnih greški pri korišćenju [13].

Adaptivne promene se bave održavanjem softvera usled promena okruženja u kojima se koristi, bilo gledano u poslovnom ili u softversko-hardverskom kontekstu [13].

Perfektivne modifikacije u jednom smislu mogu podsećati na preventivne, s tim da svrha promena nije izbegavanje bagova (kao npr. pad sistema), već se žele izbeći neuspesi poslovne prirode. Dodatno, u ovu grupu spadaju zahtevi za poboljšanje radnih performansi softvera, unapređenje dokumentacije i dodavanje novih funkcionalnosti. Poboljšavanje održivosti softvera je takođe tematika ovih promena, gde se održivost (eng. *maintainability*) opisuje kao fleksibilnost sa kojom se postojeći kod može održavati [13].

3 Uloga i značaj

Sistem mora konstantno da se prilagođava i menja jer u suprotnom postaje sve manje prihvatljiv za korišćenje [16]. Sve ove izmene nakon isporučivanja softvera, prema definiciji, možemo svrstati u održavanje softvera. Tako da odgovor na pitanje ,,Zašto nam je potrebno održavanje softvera?" je isti kao odgovor na pitanje ,,Zašto unosimo izmene u isporučen softver?". Razloge za održavanje softvera smo podelili u sledeće grupe [12]:

Obezbeđivanje kontinuiteta usluge

Greške u softveru su neminovno prisutne. Cena neuspeha softvera može biti od uznemirenog korisnika do velikih troškova ili čak gubitka ljudskih života. Kako bi platili što manju cenu greške, koja skoro sigurno postoji, želimo da sistemi nastave da rade. Dok greška u softveru daljinskog upravljača za televizor može biti tek nelagodna, softver koji kontroliše avione u letu ne sme da stane u slučaju neočekivane greške, jer je cena prevelika. Sistem održavamo operativnim popravljanjem bagova, oporavkom od neuspeha i prilagođavanjem na promene operativnog sistema i hardvera.

Podrška obaveznih ažuriranja

Do promena, iz ovog razloga, ne dolazi zbog faličnosti samog softvera već zbog spoljnih faktora, kao što su promene u zakonu ili potreba za takmičenjem sa konkurentskim proizvodom. Na primer, u Australiji je donesen novi zakon kojim su tehnološke firme primorane da plate izdavačima vesti koje koriste, te je Facebook privremeno onemogućio pregled vesti u Australiji [8].

Podrška korisničkih zahteva za unapređenja

Što je sistem bolji to je korišćeniji, a kako korisnici zahtevaju dalja poboljšanja, izmenama dobijamo još bolji softver te je sasvim jasno zašto je ovakav vid održavanja softvera neophodan i poželjan [16]. U izmene ubrajamo dodavanje novih funkcionalnosti, promene koje unapređuju sistem kroz poboljšanje performansi i personalizovanje sistema prema specifičnim potrebama korisnika.

Olakšavanje budućih radova održavanja

Makdermidova definicija softvera naznačuje da softver nije samo izvorni i objektni kod već i opsežna dokumentacija [17], i tada održavanje softvera podrazumeva i održavanje dokumentacije. U razvoju softvera često dolazimo u iskušenje da izbegavamo rad na softveru koji ne donosi promene u funkcionalnosti ili korisničkom interfejsu, ali ovakve prečice dugoročno donose velike gubitke i vremena i novca. Tako da je održavanje baza podataka, dokumentacije i refaktorisanje neophodno.

3.1 Zašto održavanje softvera, a ne novi proizvod?

Svi sistemi, i prirodni i veštački, evoluiraju – ono što razlikuje posmatranje njihovih evolucija jeste vreme potrebno za opažanje značajne promene. U prirodnim sistemima značajne promene se mogu opaziti tek posmatranjem više generacija. Sistemi koje je napravio čovek, odnosno veštački sistemi, se nešto brže razvijaju, ali i među njima ima razlike. Uređaji ili inženjerski konstrukti su veštački sistemi koji se razvijaju mnogo brže od prirodnih sistema. Na primer, Intel je od 2006. do 2021. plasirao 11, međusobno značajno različitih, generacija Intel Core procesora [5]. Konačno možemo posmatrati programe, koji su ,,voćna mušica” veštačkih sistema. Softver, sam po sebi, ne propada i kao takav ne zahteva ,,održavanje” u klasičnom inženjerskom smislu. Održavanje u slučaju softvera podrazumeva promenu karakteristika ili atributa, što možemo

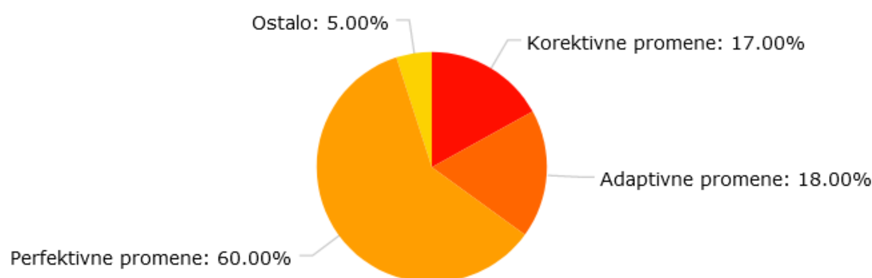
smatrati evolucionarnim promenama, čak je Lejman predlagao da se u kontekstu softvera koristi termin evolucija umesto održavanja.

Učestalost i brzina kojom se programi pokreću daju povratne informacije o manama i neophodnim proširenjima u realnom vremenu. Kako je mali fizički napor potreban za implementaciju nadolazećih zahteva, oni se u fazi održavanja prihvataju, implementiraju, a kasnije isporučuju kroz nova ažuriranja i verzije softvera. Napor koji razvoj softvera zahteva je intelektualan, nove verzije softvera se razvijaju kao nadogradnja na prethodnu implementaciju, a ne kao potpuno novi proizvod, što je slučaj kod uređaja [16].

Sledeći primer pokazuje kako je održavanje, umesto razvoja novog proizvoda, moglo da uštedi novac i vreme. Početkom devedesetih godina prošlog veka u Ujedinjenom Kraljevstvu sav avio saobraćaj bio je kontrolisan pomoću dva centra. Već tada je uočeno da se avio saobraćaj duplira svakih 15 godina, a pomenuti centri su već radili iznad svojih kapaciteta i broj situacija koje su mogle da dovedu do nesreće se značajno povećao. Tada je odlučeno da se napravi novi centar sa novom infrastrukturom i potpuno novim softverom. Završetak radova je planiran za 1996. godinu sa cenom od 339 miliona funti. Zbog grešaka u softveru otvaranje centra je odloženo za kraj 2001. sa cenom od 623 miliona funti. Na kraju, centar je otvoren tek 2002, ljudi koji su radili na softveru su priznali da je verovatno najveća greška bila upravo to što su softver pisali od početka, umesto kao nadogradnju na već postojeći sistem. Da bi rezultati bili bolji možemo videti iz primera Francuske koja je svake godine ažurirala postojeći softver, što se ispostavilo jeftinije i bezbednije jer su imali mnogo manje problema [12].

3.2 Greške u sistemu i održavanje softvera

Iz uloge i definicije održavanja softvera možemo videti da, pored novih funkcionalnosti, održavanjem se popravljaju i greške u softveru. Od ukupno uloženog novca za održavanje softvera svega 17% se troši na popravku grešaka. Zastupljenost svih kategorija u ukupnim troškovima možemo videti na slici 1.



Slika 1: Podela troškova održavanja prema kategorijama

Iznenađujuće je da je popravka grešaka na nivou šuma u ukupnoj ceni životnog ciklusa, jer ova vrsta ažuriranja može sačuvati softver od neuspeha, a samim tim materijalna sredstva i živote [11]. Na primer, Volvo je povukao sve modele vozila napravljene u 2019. i 2020. godini, jer je postojala bojazan da automatska kočnica za slučaj opasnosti neće raditi onda

kada je potrebno. Iako nisu zabeležene nesreće svi vozači ovih modela su pozvani da obavezno ažuriraju softver radi otklanjanja ove greške [18, 23].

4 Održavanje softvera u razvojnom ciklusu

Softver, ili preciznije rečeno razvoj softvera ima svoj životni vek. Time se podrazumevaju sve faze počevši od inicijalnog planiranja pa do trenutka kada se prestaje sa razvojem jer je održavanje postalo neisplativo ili su potrebe korisnika toliko povećane da je profitabilnije razviti nov softver [12].

Taj razvojni ciklus, poznat pod skraćenicom SDLC (eng. Software Development Life Cycle) je konceptualni proces koji opšte rečeno čine faze planiranja, kreiranja, testiranja i isporučivanja softverskog sistema. Vremenom su se pojavili mnogi modeli i metodologije razvoja softvera, poput kaskadnog, spiralnog, iterativnog, agilnog razvoja i drugih, pri čemu modeli opisuju same korake unutar opštog konteksta razvojnog ciklusa dok metodologije nalažu kako te ciljeve treba ostvariti [21].

4.1 Održavanje softvera u različitim modelima

S obzirom na to da se u praksi koriste različiti modeli razvojnog ciklusa [21], fazu održavanja softvera ne možemo definisati na jedinstven način. Zbog toga ćemo napraviti kratak pregled nekih razvojnih modela i pogledati kako se održavanje uklapa u njih.

Kaskadni model

Kaskadni model, poznat i kao model vodopada (eng. *waterfall*) navodi striktan niz koraka počevši od evaluacije zahteva. Svaki od koraka se vrši samo jednom, tek nakon njegove verifikacije se prelazi na sledeći, bez povratka na prethodne korake. Poslednji korak u modelu vodopada je održavanje i zapravo predstavlja ciklus sam za sebe. Ta faza traje do kraja životnog veka softvera. Opšti koncept SDLC definisan standardom ISO/IEC 12207 u velikoj meri se podudara sa modelom vodopada [20].

B-Model

B-Model je doneo modifikaciju modela vodopada time što se korak održavanja predstavlja ciklusom više koraka koji oponašaju originalni razvojni ciklus, sa ciljem da kontinualna unapređenja softvera budu zapravo deo razvojnog procesa [21].

Iterativni model

Iterativni model se može smatrati iterativnim modelom vodopada. Razvoj se obavlja u više nizova koraka modela vodopada na kraju kog se vrši isporuka, ne obavezno u produkciju. U takvom kontekstu nižu se dostupne verzije sistema, a svaka produkciona verzija ulazi u fazu održavanja dok se u isto vreme vrši razvoj u sledećoj iteraciji [20].

Agilni razvoj

Agilne metodologije u svojim modelima koriste prethodni koncept jer se oslanjaju na rad u iteracijama, međutim pristupi po kojima slovi agilni

razvoj poput konstantne komunikacije sa klijentima i reagovanje na česte promene nisu tako lako primenjivi konkretno na održavanje softvera [24].

Timovi za održavanje postojećeg koda u produkciji su praktično uvek u kontaktu sa korisnicima za razliku od razvojnih timova pred koje se postavljaju zahtevi u predvidljivijim trenucima. Na taj način, osim planiranih zadataka u okviru održavanja, u svakom trenutku rad na njima može biti prekinut zbog nekog novog zahteva koji se klasifikuje kao hitan [24].

Još jedan od razloga za značajno otežano održavanje je manja prisutnost dokumentacije nego što je to slučaj kod tradicionalnih razvojnih metodologija. Problem dodatno dobija na težini kada se uzmu u obzir slučajevi kada održavanje i razvoj vrše različiti timovi [24].

5 Alati koji olakšavaju održavanje softvera

Kada je održavanje softvera u pitanju, ne postoji magični alat koji bi rešio probleme, što je i očekivano iz obima problema koje održavanje softvera rešava, ali postoje alati koji automatski odrađuju neki zadatak koji je vezan za željenu promenu. Neke generalne kategorije zadataka koji mogu da se automatizuju i time povećaju produktivnost i olakšaju obiman posao održavanja su: razumevanje programa, obrnuti inženjering, debagovanje, testiranje, održavanje dokumentacije i konfiguracioni menadžment. Konkretne alate i preglednu podelu alata možemo videti u tabeli 1.

Zadatak	Vrsta programa	Primer programa
Razumevanje programa i obrnuti inžinjering	Program slajseri	Indus, Wisconsin Project
	Dinamički analizator	Valgrind, Daikon
	Statički analizator	Clang, Coverity, Coccinelle
	Kros-referencer	LXR Cross Referencer
Testiranje	Simulator	PureLoad, StressTester
	Generator test slučajeva	Katalon Studio
	Generator test putanji	GraphWalker
Konfiguracioni menadžment [1]	Kontrola verzija	SolarWinds, Puppet
	Izgradnja	CHEF, Ansible, SALTSTACK
	Kontrola promena	JUJU, TeamCity
Dokumentacija i metrika	Dokumentacija	Doxygen
	Metrika	GCov

Tabela 1: Pregled konkretnih alata za date kategorije zadatka

[1] U okviru konfiguracionog menadžmenta iako izdvajamo grupe programa vrlo retko se nailazi na softver koji obavlja samo poslove iz jedne od navedenih grupa, tako da su u primere programa navedeni programi koji se uopšteno koriste za konfiguracioni menadžment.

5.1 Alati za razumevanje programa i obrnuti inženjering

Pionir održavanja softvera, Ned Čejpin, naglasio je da je razumevanje najbitniji faktor tokom održavanja, koji oduzima 30% vremena utrošenog u održavanje i time je dominantna aktivnost. Razumevanje programa uključuje znanje o tome šta program radi, gde treba izvršiti promene i kako izmenjene komponente rade [12]. Obrnuto inženjerstvo daje još dublji uvid te identifikuje komponente sistema, njihove zavisnosti, daje drugačiju reprezentaciju sistema i informacije o dizajnu [3].

Poznat je fenomen programera koji je originalno napisao rešenje, a mesecima kasnije ima problem da ga modifikuje, u tom slučaju je nekom ko se priključio održavanju još teže. Ovaj problem delimično možemo rešiti pisanjem dokumentacije, ali postoje i alati koji nam olakšavaju razumevanje programa. Neki od alata su slajseri, alati za statičku i dinamičku analizu i kros-referenceri [11, 12].

Program slajser (eng. *program slicer*) izdvaja delove koda koji u nekom trenutku mogu da utiču na promenu vrednosti određene promenljive. Ovakve celine u kontekstu održavanja softvera zovemo dekompozicione celine (eng. *decomposition slicing*) i njihova definicija se proširuje na sve delove koda koji sadrže izračunavanja sa datom promenljivom. Posmatranje delova koda umesto čitavog izvornog koda nam olakšava razumevanje, a pregled na šta sve utiče promena određene promenljive nam omogućava da lokalizujemo tu promenu tako da ona ne utiče na druge nemodifikovane komponente [10].

Statički analizator daje informacije o programu ispitivanjem izvornog ili objektnog koda, bez pokretanja programa. Alat za statičku analizu traži ustaljene šablone ili pravila u kodu na osnovu kojih daje izlaz koji zahteva ljudsku evaluaciju [2]. Izlaz koji dobijamo sadrži informacije o različitim aspekatima programa kao što su moduli, procedure, promenljive, objekti, klase i hijerarhija klasa.

Dinamički analizator koristimo kada nije dovoljno samo ispitati program statičkom analizom, jer određeni aspekti se mogu sagledati tek izvršavanjem programa. Dinamičkom analizom možemo pratiti izvršavanje programa, i tako dobiti tražene informacije, kao što su putanje kroz koje treba da unesemo promene ili kako će promena koju želimo da unesemo uticati na tok izvršavanja.

Kros-referencer (eng. *cross-referencer*) generiše oznake za upotrebu datog entiteta programa. Njegovim korišćenjem možemo dobiti informacije o deklaraciji određene promenljive i svim delovima koda gde je njena vrednost menjana i korišćena. Neke primene bi bile da uočimo promenljive kojima nije dodeljena početna vrednost ili funkcije koje nikada nisu pozvane, dok nam tokom unošenja izmena ovaj alat može pomoći da sagledamo delove programa na koje će promena uticati [22].

5.2 Alati za testiranje

> ,,Testiranje, u okviru održavanja se radi na postojećem sistemu, koji je operativan, i izazvano je modifikacijama, migracijom ili povlačenjem softvera ili sistema [9]."

Testiranje je zahtevan zadatak održavanja i oduzima dosta vremena, tako da bi automatizacija značajno doprinela produktivnosti. Alate koje smo izdvojili za testiranje u fazi održavanja su simulatori, generatori test slučajeva i generatori test putanja.

Simulator omogućava da u kontrolisanom okruženju, uz podešena ograničenja, testiramo rad sistema. Tokom održavanja, simulator nam daje prednost da testiramo efekte promene u kontrolisanom okruženju pre nego što ih zaista unesemo u sistem.

Generator test slučajeva (eng. *test case generator*) generiše test slučajeve, njima obuhvatamo jedan ili više jediničnih testova koji testiraju po jednu funkcionalnost sistema kojeg modifikujemo. Za pisanje test slučajeva su nam neophodni ulazni podaci i očekivani rezultati, alat koji pomaže u generisanju ovih podataka se naziva **generator test podataka** (eng. *test data generator*).

Generator test putanja (eng. *test paths generator*) nam pomaže da generišemo test putanje. Nakon unetih izmena, poznavanjem toka podataka i kontole toka, se može izabrati odgovarajući skup testova da bi se osigurali da će uneta promena dati željene efekte – generator test putanja se može koristiti u ovu svrhu.

5.3 Alati za konfiguracioni menadžment

Sistematično praćenje i identifikovanje konfiguracija, tokom razvoja softvera, nije moguće bez automatizovane podrške alata. Jedan takav alat je Source Code Control System, ovaj alat je praktično sistem za kontrolu verzija koji prati promene u izvornom kodu i pratećim fajlovima; dodatno svaki fajl sadrži sccsid – string koji sadrži informacije o imenu fajla, datumu i verziji softvera [1, 6].

5.4 Alati za održavanje dokumentacije i metriku

Već smo naglasili da je dokumentacija sastavni deo softvera, a njen nedostatak je jedan od najvećih problema u održavanju. Alati koji olakšavaju održavanje dokumentacije uključuju alate koje prate tok podataka i generatore kontrolnih grafika, trejsere zahteva (eng. *requirements tracer*), CASE alate i alate zasnovane na hipertekstu.

Iako se, prema istraživanju, alati i tehnike vezani za metriku retko koriste u praksi, postoji dovoljno podataka da se istaknu neki od aspekata koji se najčešće mere. Na toj listi su broj defekata nađenih nakon izdanja softvera, broj promena ili broj promena zahteva, zadovoljstvo korisnika, broj defekata tokom razvoja, ažurnost i potpunost dokumentacije, vreme potrebno za identifikaciju/ispravljanje defekata, distribucija defekta kroz tip/klasu, a pri dnu se nalazi i složenost modela/dizajna.

U održavanju nam mogu pomoći i alati za merenje složenosti programa. Njihov rezultat rada se najčešće zasniva na merenju kompleksnosti algoritama i strukturama programa. Mogu nam pomoći da odredimo kada je softver previše složen da bi bio pouzdan [12].

6 Atributi kvaliteta softvera koji utiču na olakšavanje održavanja softvera

Imajući u vidu da je održavanje softvera neizostavan element u životnom veku softvera, kao i najskuplji [15], postoji opravdana potreba da troškove rada, ali i generalni posao, umanjimo i olakšamo. To se može postići ukoliko, na određeni način, uspemo da dodelimo vrednosnu ocenu kvaliteta našem softveru. Zvuči razumno zaključiti da, što bi softver bivao kvalitetniji (po nekoj meri), prostor za izbijanje bagova ili nezadovoljstva

korisnika bi bivao manji. Standard ISO/IEC 25023 iz 2016. godine nam nudi dve pogodne definicije:

Eksterna mera – mera kojom se određuje u kom stepenu se sistem ili softverski proizvod ponaša tako da zadovoljava zatražene i implicitne potrebe.

Prostim rečima, tražimo da softver uspešno izvrši zadatak koji mu je namenjen. Međutim, čak i ako uspešno izvršava zadatak, to ga ne čini nužno kvalitetnim. Razlog je, pre svega, što se mi ovde ne bavimo njegovom strukturom i implementacijom, već isključivo njegovom funkcionalnošću. Ako, recimo, imamo program koji vraća željenu vrednost, ali to čini izuzetno sporo, to opet može, gledano iz ugla korisnika, da bude nepogodnost i nešto zbog čega bi trebalo menjati program. Stoga, eksterna mera nije dovoljna, ali je svakako neophodna. Zbog toga, posmatramo drugu definiciju:

Interna mera - mera kojom se određuje u kom stepenu skup statičkih atributa softverskog proizvoda zadovoljava zatražene i implicitne potrebe.

Dakle, bilo koja merljiva osobina koja će opisati naš softver se ovde može uzeti u obzir, a kaže se da je interna pošto krajnjeg korisnika ne treba ni da interesuje, već je zanimljiva kreatorima softvera radi ocenjivanja kvaliteta softvera. Upravo su **atributi kvaliteta softvera** oni koji imaju tu ulogu, sa glavnim akcentom na one koji su najznačajniji pri procesu održavanja softvera.

Primer nečega što bi bilo teško za održavanje u programiranju je tzv. *špageti kod* (eng. *spaghetti code*). To je „sleng koji se koristi za opisivanje izvornog koda koji je težak za čitanje i praćenje zbog načina na koji ga je originalni programer napisao"[4]. Može uključivati loše raspoređene odgovornosti (klase ili funkcije koje rade preširok i preobiman posao), korišćenje naredbe goto, ponavljanje koda i sl. Jasno je da je cilj kreiranje softvera koji će biti sposoban za laku modifikaciju po potrebi. Standard ISO/IEC 25023 uvodi definicije raznih atributa kvaliteta softvera, poput mera za performanse ili sigurnost softvera. Ovde će, međutim, biti opisane tzv. **mere održavanja** (eng. *maintainability measures*)[14].

Mere modularnosti (eng. *modularity measures*) – određuju u kom stepenu je sistem ili računarski program sastavljen od komponenti tako da promena na jednoj komponenti ima minimalan uticaj na druge. Tu se ubrajaju:

- ***Spregnutost komponenti*** (eng. *coupling of components*) – meri koliko su komponente međusobno nezavisne i slobodne od izmena drugih komponenti u sistemu; računa se kao količnik broja komponenti čija izmena nema uticaj na druge i broja potrebnih komponenti koje treba da budu nezavisne; želeli bismo da taj broj bude što bliži 1, a najčešće se postavi neka granica koju treba preći.
- ***Adekvatnost ciklomatične složenosti*** (eng. *cyclomatic complexity adequacy*) – ciklomatična složenost je metrika koja određuje kompleksnost programa i ona računa broj mogućih izbora iliti puteva u izvornom kodu. Računa se po formuli E - N + 2P, gde je P broj odvojenih delova grafa toka programa (glavna funkcija predstavlja jedan deo, neka funkcija koja se poziva drugi i sl), E broj grana, a N broj čvorova[19]. Softver ima prihvatljivu ciklomatičnu složenost ako je broj softverskih modula, koji imaju vrednost ciklomatične složenosti iznad neke zadate granice, što bliži 0, a i tu se obično postavlja neka granica. U dodatku se može pogledati kod 1 i

graf toka programa za taj kod 2.

Mere mogućnosti ponovne upotrebe (eng. *reusability measures*) – određuju u kom stepenu neki deo sistema može biti korišćen u drugim sistemima ili za građenje drugih delova istog sistema; idealno je ako komponente mogu da se koriste bez izmena ili sa vrlo malim izmenama – ne želimo da radimo neki posao koji je već odrađen. Dele se na:

- ***Mogućnost ponovne upotrebe komponenti*** (eng. *reusability of assets*) – meri koliko komponenti sistema se može iznova koristiti. Računa se kao količnik komponenti koje su dizajnirane da se mogu ponovo koristiti i svih komponenti sistema. Naravno, težimo ka tome da se što više komponenti može ponovo upotrebljavati.
- ***Povinovanje pravilima kodiranja*** (eng. *coding rules conformity*) – meri koliko modula poštuje zadata pravila kodiranja. Želimo da što više modula ispunjava zadata pravila kako bi softver bio sigurniji i pouzdaniji.

Mere analiziranosti (eng. *analysability measures*) – služe za procenu uticaja nameravane promene na sistem na jedan ili više njegovih delova, kao i dijagonostiku nedostataka u softveru ili uzroka kvara, a služe i za identifikaciju delova koje treba modifikovati u softveru. Najznačajnije su:

- ***Kompletnost log sistema*** (eng. *system log completeness*) – obim beleženja sistemske evidencije (log fajlovi).
- ***Brojnost dijagnostičkih funkcija*** (eng. *diagnosis function sufficiency*) – meri koliko dijagnostičkih funkcija je implementirano u odnosu na traženi broj. Pre svega su to funkcije koje dijagnostikuju neke nedostatke ili uzroke kvara, kao i one koje identifikuju delove koje treba modifikovati.

Mere modifikabilnosti (eng. *modifiability measures*) – određuju u kojoj meri sistem može biti efikasno modifikovan bez pojavljivanja nekih nedostataka ili degradiranja kvaliteta postojećeg proizvoda. Tu nailazimo na:

- ***Efikasnost promene*** (eng. *modification efficiency*) – meri koliko efikasno, posmatrano vremenski, su modifikacije bile izvršene u poređenju sa očekivanim vremenom.

  Formula koja se koristi je $\sum_{i=1}^{n}(A_i/B_i)/n$, gde je A_i potrošeno vreme za neku izmenu, B_i očekivano vreme izmene i n broj posmatranih izmena.

  Vrednost manja od 1 reprezentuje veoma efikasne izmene, a vrednost veća od 1 neefikasne izmene.
- ***Ispravnost promene*** (eng. *modification correctness*) – proporcija izmena koje su korektno implementirane. Meri se kao količnik loših modifikacija, tj. modifikacija koje su proizvele neki problem, i broja svih modifikacija.

Mere testiranja (eng. *testability measures*) – određuju u kojoj meri možemo izvesti testove na sistemu i utvrditi da li su ispunjeni traženi kriterijumi. Ovde beležimo:

- ***Kompletnost test funkcija*** (eng. *test function completeness*) – broji implementirane test funkcije. Meri se kao broj implementiranih u odnosu na broj traženih test funkcija. Što je više test funkcija i što su one kvalitetnije, to je i sam proizvod kvalitetniji.

- ***Test restartovanja*** (eng. *test restartability*) – količnik testova u kojima je moguće pauzirati i restartovati izvršavanje testa i broja svih testova kod kojih je moguće pauzirati izvršavanje.

Pored ovih, valjalo bi dodati i *skalabilnost* – meru sposobnosti sistema da se prilagodi radu sa većim brojem korisnika.

7 Zaključak

Ovim seminarskim radom detaljno je objašnjen pojam održavanja softvera i argumentovan njegov značaj u celokupnom razvoju softvera. Ako imamo proizvod koji nije moguđe nadograđivati i unapređivati, nameće se pitanje koliko je on zaista i pogodan za korišćenje. Budući da se održavanjem softvera vrši nadogradnja nečega već postojećeg, često je mnogo bolje, praktičnije i jeftinije raditi na modifikaciji stare verzije softvera nego kretati od nule. Premda bi se u prvi mah možda dalo zaključiti, ispostavlja se da popravka grešaka zapravo ne čini dominantan udeo u sveukupnom poslu održavanja softvera. Glavne promene su perfektivne promene, one koje dovode do unapređenja sistema i dodavanja novih funkcionalnosti. Imajući u vidu da je održavanje softvera neizbežno, u radu su obuhvaćeni i opisi alata koji olakšavaju održavanje softvera, ali i opisi atributa kvaliteta softvera koji imaju istu ulogu.

Literatura

[1] Edward H Bersoff. Elements of software configuration management. *IEEE Transactions on Software Engineering*, 1984.

[2] Brian Chess and Gary McGraw. Static analysis for security. *IEEE security & privacy*, 2(6), 2004.

[3] Elliot J. Chikofsky and James H Cross. Reverse engineering and design recovery: A taxonomy. *IEEE software*, 7(1), 1990.

[4] Computer Hope. Spaghetti code. https://www.computerhope.com/jargon/s/spaghett.htm. Accessed: 15/04/2021.

[5] Intel Corporation. Intel Core Processors, 2021. on-line at: https://ark.intel.com/content/www/us/en/ark.html#@PanelLabel122139.

[6] Oracle Corporation. Source Code Control System, 2010. on-line at: https://docs.oracle.com/cd/E19504-01/802-5880/6i9k05dhp/index.html.

[7] Dictionary.com. Adaptability. https://www.dictionary.com/browse/adaptability. Accessed: 17/04/2021.

[8] Facebook. An Update About Changes to Facebook's Services in Australia, 2020. on-line at: https://about.fb.com/news/2020/08/changes-to-facebooks-services-in-australia/.

[9] ISTQB Foundation. Maintenance Testing, 2017. on-line at: https://istqbfoundation.wordpress.com/2017/09/18/maintenance-testing/.

[10] Keith Brian Gallagher and James R Lyle. Using program slicing in software maintenance. *IEEE transactions on software engineering*, 17(8), 1991.

[11] Robert L Glass. *Facts and Fallacies of Software Engineering: FREQ FORGOT FUND FACTS _p1*. Addison-Wesley Professional, 2002.

[12] Penny Grubb and Armstrong A Takang. *Software maintenance: concepts and practice*. World Scientific, 2003.

[13] ISO. ISO/IEC 14764, 2006. on-line at: https://www.iso.org/obp/ui/#iso:std:iso-iec:14764:ed-2:v1:en.

[14] ISO. ISO/IEC 25023 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Measurement of system and software product quality, pages 25-28, June 2016.

[15] Jussi Koskinen. Software Maintenance Costs, April 2015. on-line at: https://wiki.uef.fi/download/attachments/38669960/SMCOSTS.pdf.

[16] Manny M Lehman and Laszlo A Belady. *Program evolution: processes of software change*. Academic Press Professional, Inc., 1985.

[17] John A McDermid. *Software engineer's reference book*. Elsevier, 2013.

[18] MotorTrend. Volvo, 2020. on-line at: https://www.motortrend.com/news/volvo-recall-2019-2020-vehicles-automatic-emergency-braking-issue/.

[19] Perforce Software, Inc. What Is Cyclomatic Complexity? https://www.perforce.com/blog/qac/what-cyclomatic-complexity. Accessed: 15/04/2021.

[20] Thomas M Pigoski. *Practical software maintenance: best practices for managing your software investment*. Wiley Publishing, 1996.

[21] Nayan B Ruparelia. Software development lifecycle models. *ACM SIGSOFT Software Engineering Notes*, 35(3):8–13, 2010.

[22] SWI-Prolog. SWI-Prolog XREF, 2006. on-line at: https://www.swi-prolog.org/gxref.html.

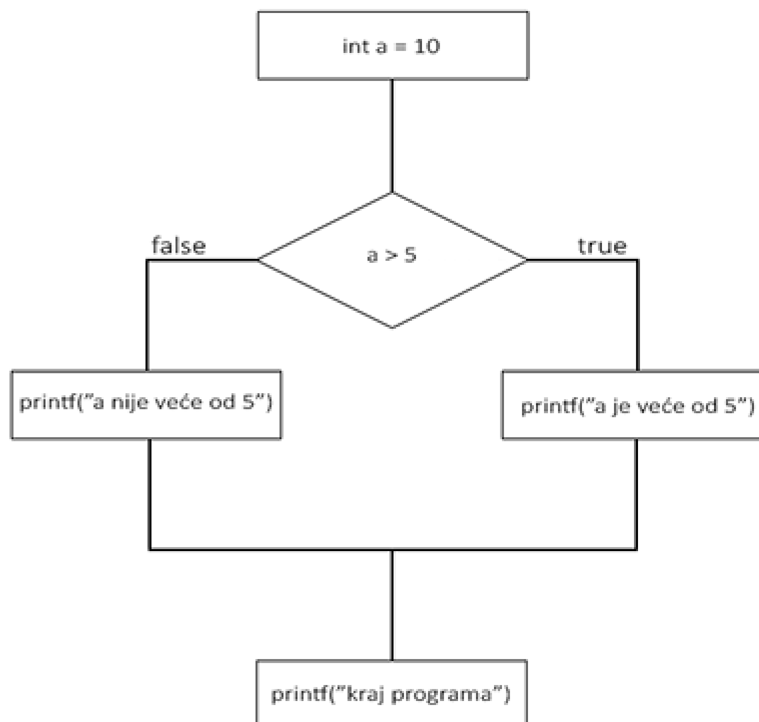
[23] Volvo Car UK. Recall Information, 2021. on-line at: https://www.volvocars.com/uk/own/maintenance/volvo-warranty/recalls.

[24] Fateh ur Rehman, Bilal Maqbool, Muhammad Qasim Riaz, Usman Qamar, and Muhammad Abbas. Scrum software maintenance model: Efficient software maintenance in agile methodology. In *2018 21st Saudi Computer Society National Computer Conference (NCC)*, pages 1–5. IEEE, 2018.

A Dodatak

```
int a = 10;
if (a > 5)
  printf("a je vece od 10");
else
  printf("a nije vece od 10");
printf("kraj programa");
```

Listing 1: Fragment koda u C-u



Slika 2: Graf toka programa 1. Pošto je E = 5, N = 5 i P = 1, dobijamo da je ciklomatična složenost 2.