Metode za pregled kôda i njihov značaj

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

Nikola Janković, Anđela Milićević, Katarina Savičić, Dunja Spasić
nikola_jankovic@tuta.io, milicevica32@gmail.com,
katarina.savicic@hotmail.com, spasicdunja3013@gmail.com

20. april 2020.

Sažetak

Pregled kôda (eng. *code review*) je proces u kom tim programera, zajedničkim čitanjem kôda, pokušava da uoči moguće propuste nastale u toku procesa razvoja softvera. U najvećem broju slučajeva, pregled kôda se organizuje pre nego što novoizmenjen kôd bude poslat na zajedničko spremište (eng. *repository*). U radu je prikazan kratak pregled osnovnih metoda, formalnih i neformalnih, informacije o softverskim rešenjima koja služe kao pomoć pri ovom procesu, uticaji pregleda i neki od praktičnih saveta koji mogu da povećaju produktivnost prilikom pregledanja kôda.
Ključne reči: pregled kôda, neformalni pregledi, formalni pregledi, Fagan, preko ramena, preko mejla, programiranje u paru, ad-hok.

Sadržaj

1 Uvod

Pri razvoju softverskih projekata dosta pažnje se posvećuje dizajnu i zahtevima, uzimaju se u obzir sva mišljenja, od naručioca posla i menadžera do projektanata, kako bi se jasno definisali zahtevi i ciljevi. Ovakav pristup je neophodan, jer su greške u definisanju zahteva i arhitekture skupe i mogu voditi ka lošem rešenju, ali ovo nije jedini izvor potencijalnih problema. Ozbiljne probleme može izazvati i nedovoljna posvećenost kôdu. Propusti i greške u projektovanju i izradi softvera koji nisu na vreme uočeni mogu dovesti do finansijskih i vremenskih gubitaka.

Prilikom razvoja softvera, čest je slučaj da autori rade samo svoj deo kôda, a da je komunikacija među kolegama ograničena na par skica projekta ili interfejsa. Ovo može biti uzrok propusta, promicanja očiglednih grešaka, dokumentacije koja ne odgovara kôdu i slično. Pregled kôda je proces u kom tim programera, zajedničkim čitanjem kôda, pokušava da uoči moguće propuste nastale u toku procesa razvoja softvera. Na ovaj način se vraća element saradnje u proces razvoja softvera, obezbeđuje da funkcionalan kôd ostane funkcionalan i u budućnosti, da novi saradnici ne prave greške koje bi njihovi prethodnici izbegli, kao i da se pronađu nedostaci koji bi potencijalno bili otkriveni tek od strane korisnika.

2 Vrste pregleda

Postoji više različitih metoda pregleda kôda, zavisno od samog pristupa, utrošenog vremena i resursa. One se mogu podeliti u dve vrste – formalne (zahtevaju više vremena i resursa, daju vrlo dobre rezultate) i neformalne (daju skoro približno dobre rezultate, ali zahtevaju mnogo manje resursa i vremena) [6].

2.1 Formalni pregledi

Formalni ili detaljni pregledi su nastali na bazi istraživanja Majkla Fagana (eng. *Michael Fagan*) u IBM-u o recenzijama kôda 1976. godine. Majkl Fagan je definisao proceduru za pregled i do 250 linija izvornog kôda. Nakon 800 iteracija došao je do formalizovane strategije za detaljan pregled, a njegove metode su dalje razvijali mnogi stručnjaci.

Formalan pregled se odnosi na iscrpnu proveru gde se tri do šest osoba sastaje u prostoriji sa projektorom i štampanim materijalima. Neko je „moderator" ili „kontroler" i ima ulogu organizatora, brine o tome da svi rade na svojim zadacima i održava tempo pregleda. Svi čitaju materijale unapred da bi se pripremili za sastanak. Svako dobija neku ulogu u pregledu. U Faganskom načinu pregleda kôda „čitač" ima zadatak da samo čita i razume kôd. Nakon toga, bez svojih dodatnih kritika i komentara kôd prezentuje grupi. Ovakav pristup odvaja cilj i zadatak autora od onoga što program zapravo radi.

Kada se otkriju greške, obično se detaljno zabeleže. Beleži se njihov tip (npr. algoritam, dokumentacija i slično), lokacija u kôdu, ozbiljnost greške i faza razvoja u kojoj se pojavljuje. Takvi podaci o greškama se najčešće čuvaju u bazi podataka kako bi kasnije mogli da se analiziraju zabeleženi podaci o greškama.

U formalnom pregledu se često beleže i druge informacije, kao što su individualno utrošeno vreme na čitanje pre sastanka, vreme provedeno na samom sastanku, stopa pregleda kôda i problemi vezani za sam proces. Ovi brojevi i komentari se razmatraju periodično na sastancima posvećenim

unapređivanju procesa. Faganski pregled ide jedan korak dalje i zahteva pregled performansi na kraju svakog sastanka.

Najveća prednost formalnog pregleda je i njegova najveća mana. Kada grupa ljudi provede mnogo vremena čitajući kôd i raspravljajući o njemu, naći će mu mnogo nedostataka. Iako mnoge studije pokazuju da formalan pregled pronalazi veliki broj problema u izvornom kôdu, većina organizacija ne može da obaveže veliki broj zaposlenih na toliko dug vremenski period. Pored toga treba zakazati sastanke, što je težak zadatak sam po sebi i zahteva dodatno vreme. Na kraju, za većinu formalnih pregleda je potrebno obučavanje recenzenata, kako bi pregledi bili efikasni, što je dodatan trošak i gubitak vremena koji je teško prihvatiti [17].

2.2 Neformalni pregledi

Istraživanja pokazuju da druge vrste pregleda kôda, osim formalnih metoda, mogu da daju približno dobre rezultate, sa mnogo manje uloženog novca u obučavanje i planiranje. Formalne metode sastanaka su ipak pokazale značajnu prednost u sprečavanju lažno pozitivnih problema (sprečavanje pronalaženja grešaka koje ne predstavljaju prave greške). Osim toga, 30% pronađenih grešaka na formalnim sastancima je otkriveno zajednički, kroz razgovor [19]. Uprkos tome, neformalne metode pregledanja kôda štede dosta resursa i vremena pa je korisno da se detaljnije razmotre.

2.2.1 Pregled preko ramena

Pregled kôda *preko ramena* (eng. *over-the-shoulder*) podrazumeva da recenzent stoji nad radnim stolom autora kôda, dok autor sprovodi recenzenta kroz najnovije izmene kôda. Obično autor sedi za računarom, otvara razne fajlove i pokazuje nove linije kôda koje su dodate ili one koje su izbrisane. Ako recenzent primeti sitne nepravilnosti, može odmah da ih istakne programeru i da one budu izmenjene na licu mesta.

Sa pojavom novih softvera za deljenje ekrana, pregledi kôda korišćenjem metode *preko ramena* mogu da se obavljaju i na većim razdaljinama. Ovakva komunikacija komplikuje proces pregleda kôda jer je potrebno zakazati sastanke ili telefonske razgovore.

Prednost ove metode je jednostavnost. Bilo ko može da učestvuje u pregledu preko ramena bez prethodnog obučavanja. Još jedna značajna prednost je što može da se primeni bilo kad, što je bitno kada je potrebno pregledati jako važnu izmenu kôda. Generalno, svi pregledi kôda uživo su dosta korisni jer pružaju programerima mogućnost da razmene ideje koje ne bi delili preko mejla ili poruka.

Jednostavnost i neformalnost ovog pristupa takođe sa sobom nose i neke nedostatke. Glavna mana je nemogućnost provere da li su pregledane sve izmene kôda. Ne postoje nikakvi izveštaji ili mere koje bi dokumentovale proces. Još jedan problem je što može veoma lako da se desi da autoru promakne napravljena izmena kada svoj kôd prezentuje recenzentu. Recenzent jedino vidi ono što mu autor pokaže, nema mogućnost da sam pregleda ostale datoteke na koje bi ta izmena mogla da utiče. Ako autor napravi izmenu koja nije potpuno jasna svakom programeru bez pojašnjenja autora, sledećem programeru koji pregleda ili koristi kôd neće biti jasna izmena. Pregled kôda *preko ramena* može efikasno da funkcioniše samo ako su autor i recenzent fizički blizu, u istoj prostoriji ili rade u susednim prostorijama ili zgradama. Svako prebacivanje recenzije na elek-

tronski vid komunikacije gubi poentu same ideje pregleda *preko ramena* [17].

2.2.2 Pregled preko mejla

Pregled kôda *preko mejla* (eng. *email pass-around*) podrazumeva da autor kôda zapakuje sve izmenjene fajlove, a zatim ih mejlom pošalje recenzentima. Recenzenti pregledaju fajlove, postavljaju pitanja, razgovaraju sa autorima i predlažu izmene. Ovakav način pregleda kôda je najpopularniji među programerima na projektima otvorenog kôda (eng. *open souce*). Do potencijalnih propusta može doći na obe strane, jer autor ima obavezu da sakupi sve izmenjene fajlove, gde je lako prevideti neki, a recenzent mora da pažljivo uporedi stari i novi kôd da bi pronašao sve napravljene izmene.

Sistemi za kontrolu verzija su korisni u ovakvim recenzijama. Oni obično vode računa o tome koje su datoteke bile menjane. Sistemi za kontrolu verzija mogu da pomognu i tako što pošalju mejlove automatski svima koji su uključeni u projekat. Ova mogućnost je korisna, ali često, potrebno je pregledati kôd pre nego što se on postavi u sistem za kontrolu verzija [17].

Preglede kôda *preko mejla* je jednako lake primeniti kao i pregledi *preko ramena*. Iako je velika prednost pregleda *preko ramena* što oduzima manje vremena, kod pregleda *preko mejla* komunikacija među programerima je podjednako dobra nezavisno od njihove fizičke udaljenosti (ne pravi razliku da li su na različim kontinentima ili u istoj kancelariji). Još jedna prednost pregleda *preko mejla* je što je lako dodati novog programera u sistem. Glavna mana pregleda *preko mejla* je što je veoma teško voditi računa o svim razgovorima koji se vode o korekciji grešaka u kôdu. Kod projekata koji uključuju programere u različitim vremenskim zonama pregledi mogu i da traju dugo, zbog vremena potrebnog da programeri pregledaju kodove i razmene opažanja. Pored toga, pregledi kôda *preko mejla* dele već pomenute mane pregleda *preko ramena* – nemogućnost formalnog beleženja statističkih podataka o tome koji delovi kôda su pregledani, a koji ne. Ipak, uvođenjem sistema za kontrolu verzija otklonjene su neke mane ova dva pregleda [17].

2.2.3 Programiranje u paru

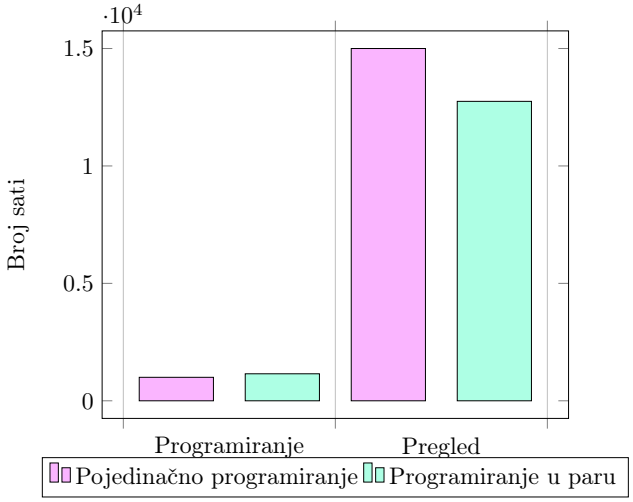
Tehnika programiranja u paru podrazumeva da dva programera pišu kôd zajedno za jednim računarom, jedan programer piše kôd dok drugi sedi pored njega i prati šta je napisano. Naravno, oni zajedno diskutuju i planiraju kako će da bude napisan program.

Istraživanje je pokazalo da programiranje u paru ima veliku prednost u odnosu na samostalno programiranje [15]. Mnoge greške se prepoznaju već za vreme pisanja kôda, pa je stoga i učestalost grešaka tokom pregleda kôda manja. Program je obično bolje osmišljen i napisan u manje linija; programerima je potrebno manje vremena da osmisle rešenje problema zajedno. Kao i kod bilo kog drugog timskog rada, programeri razmenjuju znanja i informacije, prilagođavaju se bolje kolegama i zadovoljniji su svojim poslom.

Pomenuto istraživanje je vršeno sa studentima softverskog inženjerstva [15]. Jedna trećina njih je radila samostalno, a dve trećine su radile u paru. Pokazalo se da su studenti koji su radili u paru potrošili 15% više vremena da napišu kôd u odnosu na samostalne studente, ali se ispostavilo da su

oni u paru imali 15% manje grešaka u kôdu. U većini firmi se napisan program šalje odeljenju za proveru kvaliteta kôda, kojima je obično potrebno između 4 i 16 sati da pronađu grešku. Izračunato je da bi na 50.000 linija kôda programeri u paru imali 225 grešaka manje. To znači da bi u proseku bilo potrebno 2.250 sati dodatno da se pregleda samostalno pisan kôd, u odnosu na onaj pisan u paru. To je 15 puta više nego broj dodatnih sati koje je izračunato da bi programerima u paru bilo potrebno da napišu kôd od 50.000 linija (150 dodatnih sati). Ovo znači da bi bilo potrebno 15% više sati rada programera da se napiše program, zato i više resursa da se isplate programeri, ali je mnogo značajnije umanjenje vremena provere softvera i povećanje kvaliteta kôda. Na slici 1 je dat uporedni prikaz broja sati utrošenih pri pisanju i pregledanju 50.000 linija koda, pojedinačno ili u paru. Može da se primeti značajno uvećanje sati pregleda u odnosu na dodatno vreme potrebno dvama programerima da napišu kôd. Ako se kôd ne pregleda, nego se odmah prosleđuje korisniku, korisnije je da se piše u paru zbog manjeg broja inicijalnih grešaka.

Glavna mana programiranja u paru, osim većeg broja programerskih sati je što nisu svi programeri radi da imaju partnera sa kojim će da razmenjuju svoje ideje, a i moguće je da uparene osobe nisu kompatibilne za zajednički rad. Osim toga, rad u paru se posebno komplikuje ako programeri nisu fizički na istom mestu [17].



Slika 1: Potreban broj sati za pisanje i pregledanje 50.000 linija kôda pisanih pojedinačno i u paru

3 Alati

U odeljku posvećenom pregledu kôda *preko mejla* je pomenuta komunikacija preko sistema za kontrolu verzija. Sistem za kontrolu verzija je korak ka softverskoj formalizaciji pregleda kôda *preko ramena*. U ovom delu će više pažnje biti posvećeno drugim softverskim pristupima za automatizaciju različitih delova pregleda kôda.

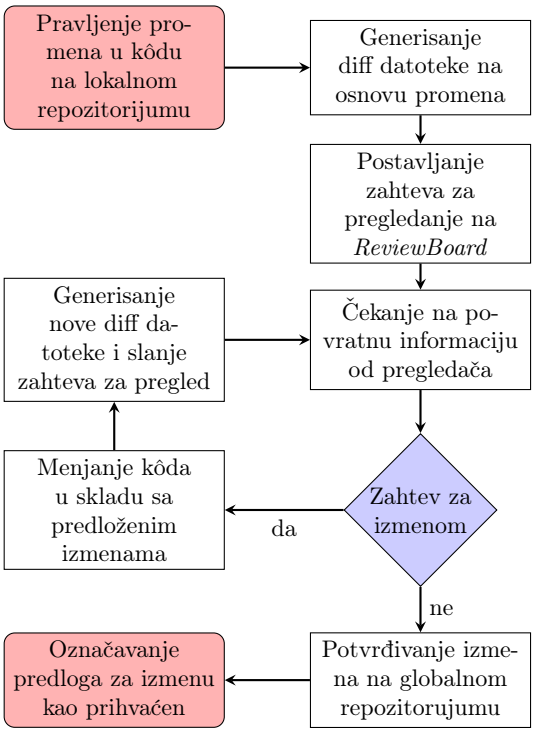
Veliki broj procesa obuhvaćenih pregledom kôda može da se automatizuje. Softveri koji rešavaju ovakve probleme mogu da se podele u dve

kategorije – **otvorenog kôda** i **vlasnički**. U ovom radu je više prostora izdvojeno za opcije iz prve grupe, jer je softver otvorenog kôda svima dostupan, a zbog te svoje osobine i često korišćen u akademskom radu.

3.1 Alati otvorenog kôda

ReviewBoard: Projekat *ReviewBoard* su započela dva programera, Kristejn Hemond (eng. *Christain Hammond*) i David Troubridž (eng. *David Trowbridge*), kompanije *VMware*. Oni su dobili zadatak da unaprede dotadašnji mehanizam pregledanja kôda u timovima te kompanije. Projekat se svodio na generisanje HTML kôda koji je prikazivao staru i novu verziju kôda i markirao delove koji se razlikuju. Članovi tima su imali mogućnost da dodaju i objašnjenja zašto su pravili te izmene i koje su sve testove sproveli nad novom verzijom kôda. Nakon toga se slao zahtev potencijalnim pregledačima. Sve to je oduzimalo previše vremena, dešavalo se i da neki zahtevi bivaju zagubljeni usled kompleksnosti i zahtevnosti samog sistema. Zato je bilo neophodno unaprediti ceo sistem.

Izvorni kôd ovog alata napisan je u jeziku *Python* uz pomoć radnog okvira *Django*. Pod licencom je *Open Source MIT license* [14]. Proces rada prikazan na slici 2 ne razlikuje se bitno ni kod ostalih alata koji će biti predstavljeni u nastavku.



Slika 2: Graf toka jednog procesa u radu sa alatom *ReviewBoard*

Gerrit: *Gerrit Code Review* je započet kao skup dodatnih mogućnosti na već ranije razvijen projekat pod nazivom *Rietveld* i prvobitna svrha

mu je bila da služi projektu AOSP[1]. Kasnije je postao zaseban projekat sa novim, značajnijim mogućnostima koje je autor sistema *Rietveld*, Gvido van Rosum (hol. *Guido van Rossum*)[2] odbijao da doda u originalni projekat kako bi zadržao jednostavnost. U tom momentu je počela i značajna promena samog kôda pa je bio potreban novi naziv. Odabrano je ime *Gerrit*, u čast holandskog arhitekte Gerita Ritvelda. Verzija softvera *Gerrit* pod oznakom 2.X je bila značajna jer je izvorni kôd napisan u programskom jeziku *Python* reimplementiran uz pomoć programskog jezika *Java* [4]. Ovaj softver je licenciran pod *Apache* licencom, ali postoje i vlasničke verzije softvera.

Codestriker: *Codestriker* je veb aplikacija koja podržava onlajn preglede kôda. Moguće je to učiniti na tradicionalan način, pregledom dokumentacije, ali takođe je podržan i pregled promena generisanih pomoću SCM[3] sistema. Prva verzija ovog softvera je nastala u decembru 2001. godine i objavljena je na *SourceForge*[4] platformi. Prvobitno, bio je implementiran kao ad-hok rešenje u vidu manjeg skript programa napisanog u programskom jeziku *Perl*. Sve mogućnosti bi se mogle svesti u jednu rečenicu: „Prosledi svim potencijalnim pregledačima na mejl adresu izlaz komande iz programa koji služi kao posrednik u sistemu CVS [5] i omogući pregledačima da mogu da ostavljaju komentare" [3].

Poslednje verzije ovog softvera omogućavaju i dalje ovakav manje formalan (eng. *light-weight*) metod pregleda kôda, ali podržava i potpuno formalan pristup. Za razliku od projekta *Gerrit*, ovaj projekat je u svim svojim verzijama pod licencom koja spada u grupu onih koje zastupaju koncept softvera otvorenog kôda, tačnije GPL[6].

Phabricator: *Phabricator* nije jedna, već skup više veb aplikacija koje olakšavaju razvoj softvera, najviše rad u timu. Implementacija je najvećim delom bazirana na internim alatima kompanije *Facebook* [2].

Glavne komponente sistema *Phabricator* su:

- *Differential*
- *Diffusion*
- *Maniphest*
- *Arcanist*

Differential je komponenta koja obavlja glavni deo posla koji se razmatra u ovom radu. Ima sličan tok rada kao *ReviewBoard*, što je prikazano na slici 2. Razlika je samo što ceo proces ne obavlja samostalno kao *ReviewBoard*, već koristi i alat *Arcanist* pomoću kog se iz terminala (eng. *command-line*) generišu diff datoteke. *Diffusion* omogućava pregled repozitorijuma, što implicitno omogućava pregled kôda nakon što izmena bude potvrđena na globalnom repozitorijumu (eng. *post-push review*). *Maniphest* je komponenta koja služi za praćanje grešaka (eng. *bug tracker*) [1].

[1] Android Open Source Project
[2] Guido van Rossum, poznatiji kao kreator programskog jezika *Python*
[3] Source Code Management
[4] Platforma na Internetu koja omogućava pristup softveru otvorenog kôda
[5] CVS je sistem za kontrolu verzija
[6] The GNU General Public License

Nabrojaćemo i nekoliko primera iz grupe koja je pod vlasničkom licencom, a čitaocu ćemo ostaviti referentne lokacije ukoliko je zainteresovan da više istraži:

- *Collaborator* [12]
- *CodeScene* [7]
- *Crucible* [10]
- *Veracode* [8]
- *Jarchitect* [9]

Razlog široke upotrebe alata za pregled kôda je potreba proizvodnih menadžera (eng. *product managers*) da podstaknu članove tima da budu ažurni i pedantni pri pregledanju. Alati mogu da obezbede timovima manje rigoroznu kontrolu pregleda, ali i strože mehanizme kontrole, u vidu servera za praćenje pregleda kôda (eng. *verision control level*) [17].

Tim kompanije *SmartBear* je proveo godine istražujući postojeće studije pregleda kôda i učeći na primerima predstavnika više od 30 različitih industrija. Ispitivao je kako različite organizacije primenjuju tehniku pregledanja kôda i sa kojim izazovima se suočavaju [5].

Tabela 1: Istraživanje kompanije *SmartBear* iz 2017. godine

metod / period	Ad-Hok	Sastanci	Alati
dnevni nivo	82	22	110
sedmični nivo	192	132	130
mesečni nivo	77	82	43
kvartalni nivo	27	43	43
godišnji nivo	11	12	14
nikad	132	230	186
ostalo	27	27	22

Sumarni rezultati istaživanja na 548 ispitanika o primeni različitih metoda pregleda kôda (ad-hok, sastanci i alati) i učestanosti njihove primene dati su u tabeli 1. Tabela pokazuje da se na dnevnom nivou najviše koriste alati, dok se na sedmičnom nivou više primenjuje ad-hok metoda odnosno metoda *preko ramena*. Sastanci su najzastupljeniji na mesečnom nivou. Ipak, veliki procenat ispitanika uopšte ne koristi ove metode što bi trebalo menjati u budućnosti, s obzirom na njihov pomenuti značaj.

4 Uticaji pregleda

Kako pregledi kôda utiču na programere koji zajedno rade u timu? Pokazalo se da dolazi do mnogih pozitivnih uticaja, ali i do nekih negativnih [17]. O ovim uticajima je korisno diskutovati – pozitivne pohvaliti, a na negativne posebno obratiti pažnju i predložiti načine da se oni reše. U ovom delu rada će biti navedeni i objašnjeni primeri obe ove vrste uticaja pregleda kôda.

4.1 Pozitivni uticaji

Programeri, znajući da će kolege iz tima pregledati njihov kôd, postaju pažljiviji. Svako želi da dobije što bolju povratnu informaciju i sigurno niko

ne želi da bude član tima koji uvek pravi neke početničke greške. Stoga će svako uložiti dodatni napor da sve proveri i poštuje pravila kodiranja. Ovaj uticaj se naziva „Ego efekat" (eng. *the "Ego Effect"*) [17].

Kod tehnike pregledanja, razmena znanja je obostrana. Iako se može pretpostaviti da u tom procesu uči samo programer čiji se kôd pregleda, nije tako. I pregledač može nešto novo naučiti gledajući kôd. Prvenstveno, iskusniji programeri pregledaju rad mlađih programera. Tada oni mogu uočiti neke stvari na koje, zbog navike, nisu obraćali pažnju, otkriti nove ideje i usvojiti nov način razmišljanja [17].

Pregledi ne podstiču samo konverzaciju o kôdu, već i lični razvoj. Programer će saznati koje su to greške koje često ponavlja, a kojih možda nije ni bio svestan. Radiće na tome da ih ispravi. Vremenom će postati produktivniji i efikasniji, bez nekog pritiska, samo posmatrajući sebe [17].

4.2 Negativni uticaji

Niko ne prihvata rado kritike i svakome bude neprijatno kada mu se ukaže greška. Ipak, ljudi reaguju na kritike na različite načine. Neki ljudi to lakše podnesu, isprave grešku koju su napravili i uz šalu prevaziđu situaciju. Drugi, posebno ako smatraju da su dali sve od sebe, kritike shvataju vrlo lično i povlače se u sebe. O tome treba voditi računa. Menadžeri moraju promovisati stav da su nedostaci pozitivni. Svaki od njih je prilika za poboljšanje kôda, a cilj postupka pregledanja je učiniti kôd što boljim. Cilj je eliminisanje što većeg broja oštećenja, bez obzira na to ko je prouzrokovao grešku. Pronalazak nedostatka ne znači „autor je napravio grešku i pregledač ju je pronašao", već znači da su autor i pregledač zajedno kao tim radili na poboljšanju proizvoda [17, 16].

Pored lošeg prihvatanja kritike, negativni uticaj je i efekat „Velikog brata" (eng. *the "Big Brother" effect*) [17]. Programer može steći utisak da ga neko stalno posmatra, pogotovo ako radi sa alatima za pregledanje. Zabeleženi i izmereni podaci su značajni za proces pregledanja, ali mogu izazvati loš efekat. Ako programer misli da će ti podaci biti iskorišćeni protiv njega, ne samo da će biti neprijateljski nastrojen prema procesu pregledanja, već će se fokusirati na poboljšanje svoje statistike umesto da zaista napiše bolji kôd. Menadžeri moraju biti svesni ovoga. Ako izmereni podaci ukazuju na potencijalni problem u kôdu, ne treba to istaći programeru kao pojedincu. Bolje je obratiti se celoj grupi koja radi na projektu, ali ne sazivati poseban sastanak u ovu svrhu, već samo taj problem uvrstiti u neki uobičajeni postupak [17, 16].

Važno je imati na umu i to da je složeniji kôd skloniji greškama. Vrše se detaljniji pregledi i očekuje se da će biti dosta nedostataka, pa se često veliki broj istih više pripisuje složenosti kôda nego sposobnostima autora [16].

4.3 Agilni timovi i pregledi kôda

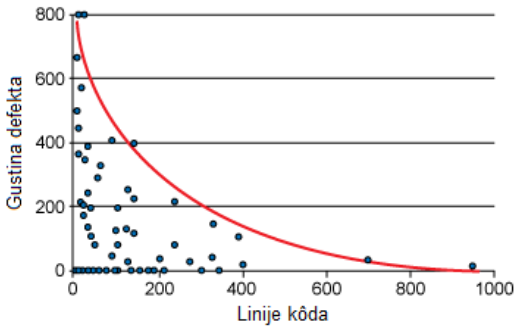
Pregledi kôda u timskom radu imaju veliki značaj bez obzira na metodologiju razvoja softvera. Ipak, agilni timovi mogu imati dodatnu prednost jer se njihov posao decentralizuje u okviru tima. Niko ne voli kada mora da preuzme rad na tuđem kôdu, posebno u hitnoj situaciji. Pregledi kôda, deljenjem znanja kroz tim, omogućavaju da svaki član tima može da preuzme i nastavi svaki posao. Suština je da ne postoji jedinstvena ličnost koja zna specifičnosti nekog dela kôda, već su svi u sve upućeni [6, 11].

5 Saveti za dobro pregledanje

Pri pregledanju kôda važno je znati na koje stvari treba ukazati autoru, a koje stvari su manje bitne. Prolazeći kroz kôd, recenzenti mogu ostaviti pohvale za neke segmente kôda ili mogu postaviti pitanja u vezi sa daljim planovima za razvoj projekta. Ovakvi komentari su značajni za komunikaciju i dobru atmosferu u timu, ali ne doprinose direktno poboljšanju kôda. Zato su ovakve komentare ispitanici u istraživanju koje je sprovela kompanija *Microsoft* 2015. godine ocenili kao nekorisne [13]. Za korisne komentare naveli su one koji otkrivaju funkcionalne greške, greške u implementaciji ili stilske greške kako bi se poboljšala čitljivost kôda.

Na osnovu rezultata opsežnog istraživanja kompanije *SmartBear* razvijena je teorija najboljih preporuka za efikasno i kvalitetno pregledanje [16]. Neke od njih će biti predstavljene u nastavku.

Pregledajte manje od 200, odnosno 400 linija kôda odjednom. Istraživanje je pokazalo da sve preko ove granice dovodi do smanjenja efikasnosti. Ovim tempom rada, koji ne prelazi vremenski raspon od 60 do 90 minuta, uspešnost bi trebalo da bude između 70 i 80 procenata. Grafik na slici 3 koji prikazuje međusobnu zavisnost broja linija kôda koji se pregleda i gustine defekta, potvrđuje ovu tezu. Gustina defekta je broj defekata pronađenih u 1000 linija kôda. Kako je broj linija kôda koji se pregleda prelazio vrednost 200, gustina defekta je značajno opadala.



Slika 3: Grafik zavisnosti broja linija kôda i gustine defekta

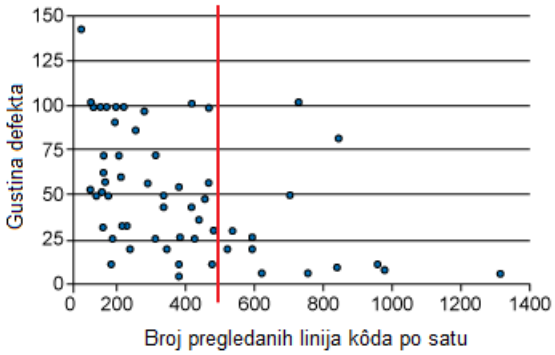
U ovom slučaju, gustina defekta je vrednosna jedinica „efektivnosti pregleda“ (eng. *review effectiveness*). Ako dva pregledača pregledaju isti kôd i jedan pronađe više grešaka, smatrali bismo da je on efikasniji.

Slika 3 pokazuje da što je više kôda pred pregledačem, njegova efikasnost kao pronalazača defekata opada. Ovaj rezultat ima smisla jer on verovatno neće imati dovoljno vremena da pregleda kôd u celini.

Težite ka tome da broj linija koje pregledate bude između 300 i 500 u toku jednog sata. Ne treba žuriti prilikom pregledanja kôda. Brže ne znači bolje. Istraživanje pokazuje da će biti postignut optimalan rezultat čak i ako je po satu pregledano manje linija od ove granice.

Upoređena je gustina defekta sa tim koliko brzo je pregledač prošao kroz kôd. Ponovo, rezultat nije iznenađujući – ako nije provedeno dovoljno vremena na pregledanju kôda, neće biti pronađeno dovoljno defekata. Ako

je pregledač opterećen prevelikim brojem linija koje mora da pregleda, on neće posvetiti dovoljno pažnje svakoj liniji. Samim tim, ako napravi i malu izmenu, on neće imati dovoljno vremena da isprati sve efekte do kojih ta izmena može dovesti.



Slika 4: Grafik zavisnosti broja pregledanih linija kôda po satu i gustine defekta

Na slici 4 se vidi da pregledanje brže od 500 linija kôda po satu rezultira drastičnim padom efikasnosti pregleda. Tempom od preko 1000 linija po satu, može se zaključiti da pregledač u suštini i ne pregleda kôd, već samo preleće preko njegovog sadržaja.

Posvetite dovoljno vremena pravilnom i detaljnom pregledu, ali ne prelazite vremenski interval od 60 do 90 minuta. Nikada ne treba pregledati kôd duže od 90 minuta bez pauze. Bilo je reči o tome da, kako bi se dostigao najbolji rezultat i najveća efikasnost, ne bi trebalo pregledati kôd prebrzo. Ali takođe ne bi trebalo pregledati kôd ni predugo bez pravljenja pauze. Nakon nekih 60 minuta, pregledači jednostavno postaju umorni i prestaju da uočavaju pojedine defekte.

6 Zaključak

Proces programiranja postaje sve kompleksniji i podložniji greškama sa povećanjem broja saradnika koji rade na istom zadatku, kao i sa porastom broja linija kôda. Zbog toga je pregledanje kôda važna komponenta razvoja softvera. Rezultati istraživanja pomenutih u radu ukazuju na značajna poboljšanja kvaliteta kôda kada se neki od pregleda primene pri izradi projekta. Iako trenutno veliki broj timova zapostavlja ovaj proces, jasno je da će pregledanje u budućnosti biti u porastu. Sve veća ekspanzija alata koji ovaj proces automatizuju, a samim tim i olakšavaju, glavni je razlog zbog kog se očekuje ovaj rast. Za dalje istraživanje ove teme preporučuje se sledeća literatura [17, 18].

Literatura

[1] Adresa sa detaljnijim uputstvima vezanim za softver *Maniphest*. https://www.phacility.com/phabricator/maniphest.

[2] Adresa sa detaljnijim uputstvima vezanim za softver *Phabricator*. https://secure.phabricator.com/book/phabricator/article/introduction/.

[3] Internet lokacija *Codestriker* projekta. http://codestriker.sourceforge.net/.

[4] Internet lokacija *Gerrit* projekta. https://www.gerritcodereview.com/about.html.

[5] Istraživanje kompanije *SmartBear*. https://smartbear.com/resources/ebooks/the-state-of-code-review-2017/.

[6] Materijali za kurs Verifikacija softvera na Matematičkom fakultetu, Univerziteta u Beogradu. http://www.verifikacijasoftvera.matf.bg.ac.rs/vs/predavanja/04_staticka_analiza_pregledi/04_staticka_analiza.pdf.

[7] Zvanična internet adresa sa detaljnim informacijama. https://codescene.io/.

[8] Zvanična internet adresa sa detaljnim informacijama. https://www.veracode.com/.

[9] Zvanična internet adresa sa detaljnim informacijama. https://www.jarchitect.com/.

[10] Zvanična internet lokacija kompanije *Atlassian*. https://www.atlassian.com/software/crucible.

[11] Zvanična internet lokacija kompanije *Atlassian*. https://www.atlassian.com/agile/software-development/code-reviews.

[12] Zvanična internet lokacija kompanije *SmartBear*. https://smartbear.com/product/collaborator/overview/.

[13] Michaela Greiler Amiangshu Bosu and Christian Bird. Characteristics of useful code reviews: An empirical study at microsoft. 2015. https://www.researchgate.net/publication/308735251_Characteristics_of_Useful_Code_Reviews_An_Empirical_Study_at_Microsoft.

[14] Bosu, A. i Carver, J. Peer Code Review in Open Source Communities Using ReviewBoard. In *ACM 4th annual workshop on Evaluation and usability of programming languages and tools*, 2012.

[15] Alistair Cockburn and Laurie Williams. The costs and benefits of pair programming. In *In eXtreme Programming and Flexible Processes in Software Engineering XP2000*, pages 223–247. Addison-Wesley, 2000.

[16] Jason Cohen. 11 proven practices for more effective, efficient peer code review. 2011. https://www.ibm.com/developerworks/rational/library/11-proven-practices-for-peer-review/.

[17] Teleki S. Cohen, J. and E. Brown. *Best Kept Secrets of Peer Code Review*. Smart Bear Inc., 2006.

[18] Gee, T. *What to Look for in Code Review*. JetBrains, 2016.

[19] Philip M. Johnson and Danu Tjahjono. Does every inspection really need a meeting. *Empirical Software Engineering*, pages 9–35, 1998.