

SADRŽAJ

1. Uvod.....	1
1.1. Git.....	1
1.2. Uloga procesa razvoja programske potpore.....	2
1.3. Motivacija i cilj.....	3
2. Procesi timskog razvoja.....	4
2.1. Scrum.....	5
2.2. GitHub API.....	7
2.3. Struktura studentskih repozitorija.....	8
2.4. ReadMe.....	8
2.5. Licenca.....	8
2.6. Selenium.....	9
3. Analiza programskog koda.....	10
3.1. Potrebne instalacije.....	11
3.2. Povezivanje s GitHub API-jem.....	11
3.3. Obrada osnovnih podataka repozitorija.....	12
3.4. Obrada sadržaja direktorija repozitorija.....	16
3.5. Problemi s identifikacijom.....	23
4. Zaključak.....	24
5. Literatura.....	25
6. Sažetak.....	26
7. Summary.....	27

1. UVOD

Programsko inženjerstvo je disciplina koja se bavi metodama razvoja programske potpore. Ona zahtjeva vještine razumijevanja, oblikovanja i vrednovanja sustava programske potpore i njihovih programa. Vrednovanje sustava programske potpore obuhvaća razne elemente među kojima su glavni analiza strukture razvoja programske potpore projekta i analiza dinamike tima tijekom razvoja programske potpore. Analiza strukture razvoja programske potpore je glavna tema ovog završnog rada, a ona uključuje arhitekturnu analizu programske potpore, u što spadaju odabir arhitekurnog stila projekta te identifikacija glavnih modula sustava programske potpore. Programska potpora ne čine samo programi i programski sustavi, već i dokumentacija koja mora iznositi detalje o završnom produktu razvoja programske potpore, pratiti njen razvoj te zasebno opisivati komponente programske potpore. Za proces razvoja programske potpore je poželjno koristiti alate za razvoj programske potpore. Neki od važnijih i poznatijih vrsta alata su alati za pisanje dokumentacije kao što je to LaTeX koji osigurava profesionalno i uredno pisanje dokumentacije u PDF formatu, alati za testiranje same programske potpore kao što je to Selenium, te nedvojbeno najvažniji alati su sustavi za kontrolu verzija programske podrške od kojih je najpoznatiji Git.

1.1. Git

Git je sustav za kontrolu verzija programske podrške koji osigurava sigurnu implementaciju novih značajki i siguran paralelni rad različitih članova tima na istom projektu. Također osigurava pristup i uvid u cijeli programski kod i ostale datoteke spremljene na Git repozitorij. Neovisni paralelni rad je osiguran sustavom grana (branch) gdje različiti članovi tima rade u različitim granama i tamo razvijaju zasebne značajke. Nakon dovršavanja neke značajke njena grana se može spojiti (merge) s glavnom granom te se značajka tako asimilira u program. Svaki član Git repozitorija obično ima lokalnu kopiju glavne grane te grana koje razvija, a nakon njihovog potpunog razvijanja ih može izvršiti promjene nad Git repozitorijem (commit) kojima svi mogu pristupiti. Izvršene promjene se bilježe te je omogućeno vraćanje na bilo koju prijašnju verziju repozitorija. S Git repozitorija se

naredbe dohvaćaju naredbom „fetch“ i s naredbom „pull“ koja će dohvatiti sve promjene s Git repozitorija te ih spojiti s trenutnom lokalnom granom. Git se obično ne koristi izravno već preko Git platforma kao što su GitLab i GitHub. Takve Git platforme nude još više funkcionalnosti pomoću koje se poboljšava komunikacija tima. Samo neke od takvih značajki GitHub platforme su komentari poslije izvršavanja promjene, korištenje wiki načina za pisanje dokumentacije te alati za analizu Git repozitorija što je vrlo važna značajka za ovaj rad.

1.2. Uloga Procesa razvoja programske potpore

Procesi programskog inženjerstva obuhvaćaju razvoj programske potpore od specifikacije korisničkih zahtjeva sve do krajnje implementacije, isporuke i održavanja. Tradicionalni razvoj programske potpore se oslanja na domišljatost i iskustvo programera u implementaciji programskog koda i svega što čini programska potpora koju razvija. Dosta stavki razvoja programske potpore su u tradicionalnom načinu ostale nedefinirane te su oni koji razvijaju programsku potporu ostavljeni sa samostalnim odabirom načina razvoja programske potpore. Moderni razvoj programske potpore sve to mijenja i postavlja kriterije kvalitete proizvoda. Za kvalitetan razvoj programske potpore moderni tip razvoja zahtjeva uporabu strukture i apstrakcije izrade proizvoda. Apstrakcija mora biti evidentna u obliku modela na više slojeva proizvoda gdje model sa svakog sloja apstrahirajući prikazuje niže slojeve. Spomenuto je bitno zbog sve veće složenosti programskih proizvoda gdje je modularnost sustava i njegova podijeljenost na manje cjeline jednostavno uvjet za efikasan razvoj programskog proizvoda. Struktura izrade proizvoda je zahtjev modernog razvoja programske potpore koji govori o sustavnom pristupu razvoja. Razvoj programske potpore se dijeli na korake kao što su: definiranje programskog zahtjeva, specifikacija arhitekture programa, odabir potrebnih alata za razvoj programa, vođenje dokumentacije i samo pisanje programskog koda s konvencijama zadanim unutar tima. Kvaliteta Ovi postupci su oni koji olakšavaju proces razvoja te povećavaju kvalitetu završnog proizvoda od kojeg se očekuje da radi bez pogrešaka, ne koristi nepotrebne resurse, intuitivno se koristi te omogućava daljnju nadogradnju programskog sustava.[1]

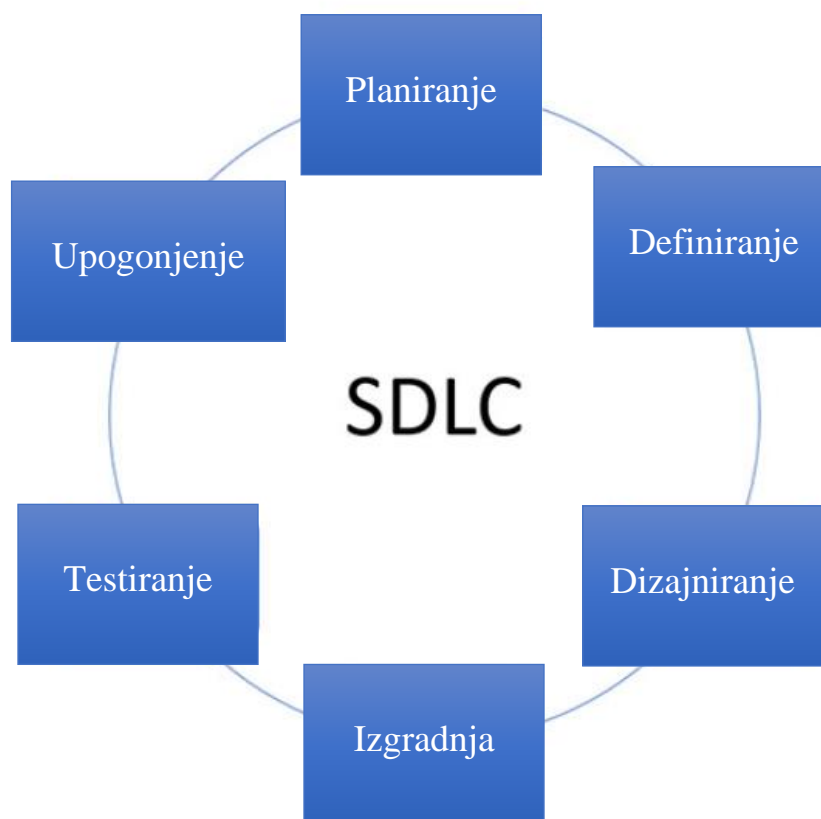
1.3. Motivacija i cilj

Glavni cilj ovog rada jest izrada programske potpore za analizu strukture GitHub repozitorija. Testni primjeri za ovaj rad su upravo projekti studenata iz predmeta Programsko inženjerstvo rađenih akademske godine 2023./2024. Program je početno predviđen kao pomoć pri ocjenjivanju daljnjih projekata iz Programskog inženjerstva narednih akademskih godina, no može se koristiti za analizu bilo kojih GitHub direktorija. Daljnja nadogradnja programa je predviđena za specijaliziranije analize repozitorija.

Programsko inženjerstvo je kolegij dizajniran za učenje studenata o efektivnom razvoju programske potpore. Timski projekt je vjerojatno najbitniji dio predmeta zato što se upravo na projektu simulira timski rad i razvoj aplikacije od početka do kraja s kakvim bi se susreli u profesionalnom okruženju.

2. PROCESI TIMSKOG RAZVOJA

Timski razvoj programske potpore iziskuje različite procese koje treba odraditi na putu do krajnjeg razvijanja programa. SDLC (*Software development life cycle*) je generički model procesa timskog razvoja programske podrške.



Slika 2.1 Prikaz SDLC ciklusa razvijanja programske potpore

Procesi timskog razvoja SDLC-a su[2]:

- **Planiranje** što uključuje definiranje projekta, skupljanje ljudskih i novčanih resursa i raspoređivanje za razvoj na razvoju programskoj potpore
- **Definiranje** što uključuje detaljnu analizu i inženjerstvo zahtjeva te postavljanje ciljeva za program koji se razvija
- **Dizajniranje** arhitekture sustava prema definiranim ciljevima

- **Izgradnja** programa i ostvarenje samog sustava i izgradnja funkcionalnosti produkta
- **Ispitivanje** funkcionalnosti produkta kako bi se osiguralo da program odgovara svim traženim zahtjevima i da se pri korištenju programa ne događaju incidenti
- **Implementacija** produkta i njegovo puštanje na tržište ili dostava klijentu. Također uključuje održavanje produkta ažuriranjima prema novim zahtjevima i eventualan popravak nedostataka

Sami razvoj programske potpore se u profesionalnom okruženju modelira pomoću raznih modela razvoja procesa programskog inženjerstva. Od tih modela su najpoznatiji vodopadni, evolucijski i komponentni model. Navedeni modeli se smatraju generičkim modelima programskog inženjerstva no u trenutno vrijeme se više naglašavaju modeli unificiranog procesa i pogotovo agilnog razvoja. Unificirani proces i agilni razvoj su zapravo podmodeli generičkih modela no puno su relevantniji u trenutnom kontekstu programskog inženjerstva.

Agilni razvoj je trenutno najrasprostranjeniji model procesa timske razvoja zbog svoje mogućnosti prilagodbe na dinamično i promjenjivo tržište. Agilni se razvoj funkcionira bez jasnog definiranja programske potpore na početku jer je u realističnim slučajevima teško sve zahtjeve definirati na početku. Preciznije je agilni razvoj pristup razvoju programske potpore kojeg obilježavaju mali inkrimenti i brze prilagodbe klijentovim zahtjevima. Takav razvoj također podrazumijeva manju ovisnost o dokumentaciji, veću komunikaciju s klijentom te reagiranje na promjene u zahtjevima. Najpoznatije metode agilnog razvoja programske podrške su ekstremno programiranje i Scrum.

2.1. Scrum

Scrum je jedan od okvira agilnog razvoja koji je strukturiran za podržavanje razvoja kompleksnije programske potpore. To radi shvaćajući da je problem kojemu je program rješenje nije razumljiv u potpunosti od početka, već se kroz razvijanje programa upoznaje s njim. Takav pristup razvijanju programskih rješenja zahtjeva inkrementalni i iterativni pristup koji će optimizirati razvijanje programa. Pri izvedbi Scrum razvoja cjelokupni tim koji radi na projektu se dijeli na manje Scrum

timove. Scrum timovi su samoorganizirajući, izvršavaju više funkcionalnosti i generalno su fleksibilni, ali imaju neke stalne uloge a to su:

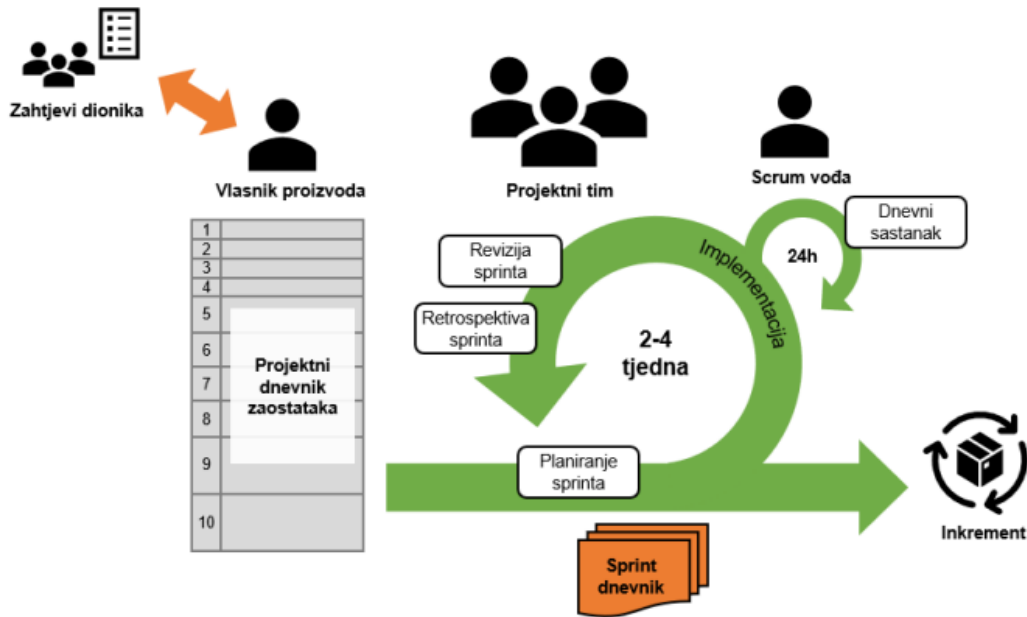
- vlasnik proizvoda – odgovoran za upravljanje dnevnikom zaostataka
- Scrum vođa – odgovoran praćenje Scrum načela unutar razvojnog tima
- Razvojni tim – grupa inženjera koje rade na inkrementu programske potpore koja se razvija

Glavni činovi razvoja programske potpore u Scrumu su događaji čiji je cilj organiziranje i izvršavanje sprintova. Sprintovi ili etape su osnovna jedinica razvoja programske potpore, obično vremenska ograničena na kraće vrijeme s točno definiranim ciljem. Događaji preko kojih se izvršava sprint su sljedeći:

1. Planiranje sprinta gdje se planira koji dio funkcionalnosti treba sprint implementirati
2. Sam sprint u kojemu se razvija program i implementira funkcionalnost, također je potrebno organiziranje dnevnog sastanka Scrum tima na kojem se vodi dnevnik o napretku sprinta
3. Revizija sprinta je proces nakon završetka sprinta gdje se analizira koliko je napravljeno od toga što se u planu zadalo za napraviti, što se nije stiglo napraviti te se dogovaraju sljedeće nadogradnje
4. Retrospektiva sprinta je općeniti retrospektivan pogled na detalje urađenog tijekom sprinta, što je bilo urađeno dobro, što loše i koje prakse se trebaju zadržati, a koje odbaciti

Scrum artefakti su zapisani dijelovi Scrum razvojnog procesa koji su neophodni za njegovo uspješno izvršavanje, a to su:

- projektni dnevnik zaostataka – sadrži listu zahtjeva za proizvod prioritetno sortirana. Odgovornost za sadržaj projektnog dnevnika zaostataka preuzima vlasnik proizvoda jer on ima najbolji uvid o prioritetu zahtjeva koje zahtjeva.
- sprint dnevnik – sadrži listu zahtjeva koji bi se trebali implementirati te plan izvršavanja posla kojim će se ti zahtjevi implementirati.
- inkrement – je napredak koji je napravljen tijekom sprinta integriran sa napretkom svih prijašnjih sprintova



Slika 2.2 Pregled događaja i artefakata Scrum-a

2.2. GitHub API

API (application programming interface) je vrsta programske potpore koja omogućuje različitim programima da pristupaju podacima i komuniciraju nekoj programskoj potporom. Pomoću GitHub API-a[3] će se u ovom projektu izvlačiti podaci o različitim GitHub repozitorijima koji će se analizirati kako bi se dobio uvid u strukturu Git repozitorija.

Programski jezik u kojem je ovaj rad pisan jest Python, a PyGithub[4] je Python knjižnica koja koristi GitHub API. Program je licenciran pomoću GNU LGPL (Lesser General Public Licence) licence koja dozvoljava korisnicima da slobodno koriste i modificiraju knjižnicu pod zahtjevom da sve modifikacije originalne knjižnice također budu objavljene pod GNU LGPL licencom. U ovom se radu koristi Python verzija 3.9 te PyGithub verzija 2.3.0, a njena dokumentacije se može pronaći na linku: <https://pygithub.readthedocs.io/en/latest/index.html>

Programska potpora koristi GitHub API te podatke i resurse s GitHub-a tako da je potrebna internetska veza za pokretanje. GitHub API nam može vratiti sve repozitorije nekog korisničkog računa tako da je GitHub token potreban za korištenje programa. Analizirat će se svi repozitoriji

kojima korisnički račun čiji je token ima pristup. Izvršavanje programa ovisi o broju repozitorija koji se analiziraju stoga je također potrebno i vremena za izvršavanje.

2.3. Struktura studentskih repozitorija

Struktura studentskih repozitorija čija je glavna motivacija ovog rada nemaju strogo zadanu strukturu, ali bi se repozitoriji trebali organizirati u skladu programskog inženjerstva. Studenti su radili većinom web aplikacije dok su neki radili mobilne aplikacije. Izbor tehnologija je ostao neograničen tako da u ovom radu analiza tokova rada (*workflow*) nije bila moguća za svaki repozitorij. No ipak postoje aspekti strukture koji su trebali biti prisutni u svim ili bar u velikoj većini studentskih repozitorija. Svaki od radova je trebao biti u cijelosti sadržan na jednom repozitoriju. Radovi su trebali unutar sebe imati različite direktorije za različite dijelove projekta kao što je prednji i zadnji kraj koda te naravno dokumentacija koja također mora biti sadržana na direktoriju. U korijenskom direktoriju također postoje i neke konvencionalne datoteke kao što su ReadMe i licenca. Također je jedan od zahtjeva ovog programa provjera korištenja Selenium alata za ispitivanje koji se koristi za ispitivanje web stranica.

2.4. ReadMe

ReadMe datoteka je uvodna datoteka u projektnim repozitorijima koja opisuje osnovne informacije o projektu. Najčešći format ReadMe datoteke je '.md' (*markdown*) te je to definitivno konvencija prisutna na Git platformama. ReadMe obično sadržava kratki opis projekta i njegovu namjenu, upute za instalaciju, upute za korištenje te sudionike koji su stvorili ili doprinijeli projektu i njihove informacije. Prisutnost ReadMe datoteke se smatra osnovnom praksom za svaki projekt te se u ovoj programskoj potpori njeno postojanje u korijenskom direktoriju provjerava.

2.5. Licenca

Licenciranje je važan dio rada na projektu i ono definira na koji način može koristiti i distribuirati određeni projekt ili u ovom slučaju programsku potporu.

Neke od glavnih podataka koje sadrži licenca uključuju:

1. Pravila koje korisnik treba poštovati pri korištenju programa kako bi korištenje programa ostalo u pravnim obvezama korisnika i autora projekta
2. Pravna zaštita od plagiranja projekta te definira kako i pod kojim uvjetima ostali zainteresirani mogu doprinosti projektu

Licence ne moraju biti unikatne za svaki projekt. Postoje vrlo popularne preddefinirane licence kao što su to MIT licenca, GNU GPL licenca i ostale.

2.6. Selenium

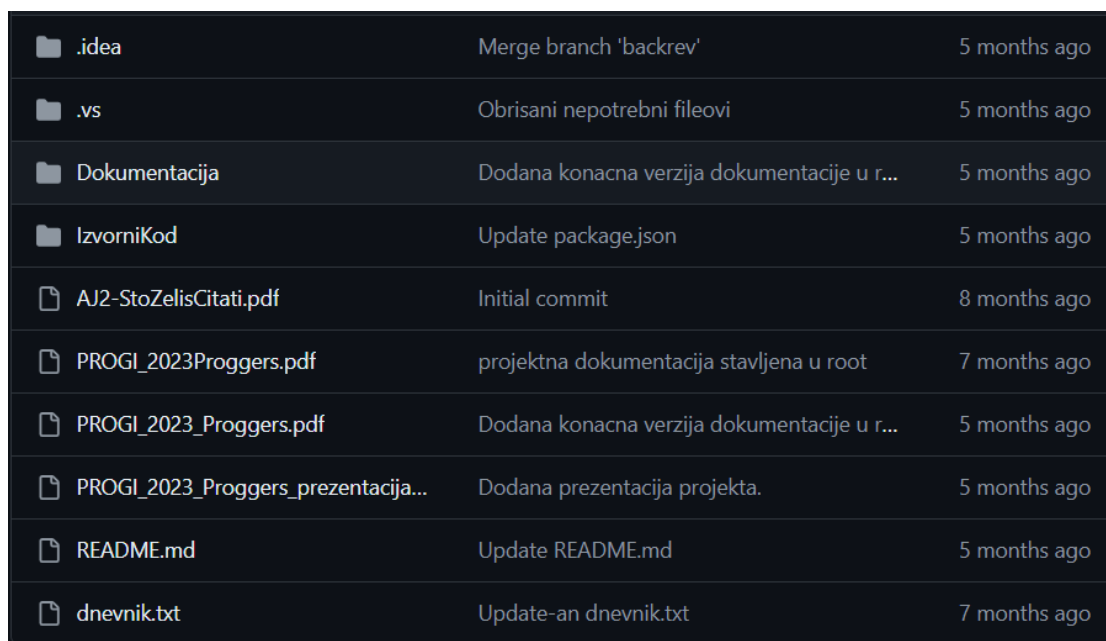
Selenium je jedan od najčešće korištenih alata za ispitivanje web aplikacija. Ima podršku za sve najpopularnije programske jezike te je besplatan. Ispituje web aplikacije tako da simulira web preglednik i korisničku interakciju s njim. Na taj način testiraju krajnje slučajeve i granice korištenja svoje web aplikacije. Ovaj završni rad uključuje programsku potporu koja provjerava je li se koristio Selenium. To se vrši tako da se traži konfiguracijska datoteka za Selenium 'pom.xml' te se gleda njen sadržaj kako bi se ustanovilo je li to zapravo konfiguracijska datoteka Seleniuma.

3. ANALIZA PROGRAMSKOG KODA

Program pri pokretanju prima Git token korisničkog računa čije direktorije želimo analizirati. Program se sastoji od 2 potprograma, svaki analizira direktorije na drugačije načine.

Prvi potprogram izvlači imena direktorija, s imenima i korisničkim imenima svih koji su sudjelovali na projektu. Nakon toga potprogram analizira programske jezike i druge tipove datoteka korištene u repozitoriju te ih navodi u kojem postotku su korištene. Zadnja funkcionalnost prvog potprograma je provjera je li se koristio alat za testiranje Selenium.

Drugi potprogram izvlači datum stvaranja repozitorija te detaljnije analizira strukturu direktorija na dva načina. Prvi od tih načina je korijenski način gdje iz najblićeg levela direktorija potprogram uzima poddirektorije u kojima se nalaze datoteke iz jedne od tri skupine: front, back i doc. Datoteke vezane uz frontend aplikacije se označavaju s front, one vezane uz backend aplikacije se označavaju s back te datoteke vezane uz dokumentaciju projekta se označavaju s doc. Korijenska analiza strukture repozitorija vraća sve direktorije najbliće razine repozitorija koje imaju datoteke iz neke od tri skupine te ispisuje koliko datoteka koje skupine imaju. Dubinska analiza strukture funkcionira na sličan način, samo što ona ispisuje sve direktorije koje unutar sebe imaju neke od triju tipova datoteka i koliko kojih tipova datoteka ima u direktoriju. Za kraj drugi potprogram provjerava jesu li prisutni ReadMe.md datoteka i licence datoteka u repozitoriju.



📁 .idea	Merge branch 'backrev'	5 months ago
📁 .vs	Obrisani nepotrební fileovi	5 months ago
📁 Dokumentacija	Dodana konacna verzija dokumentacije u r...	5 months ago
📁 IzvorniKod	Update package.json	5 months ago
📄 AJ2-StoZelisCitati.pdf	Initial commit	8 months ago
📄 PROGI_2023Proggers.pdf	projektna dokumentacija stavljena u root	7 months ago
📄 PROGI_2023_Proggers.pdf	Dodana konacna verzija dokumentacije u r...	5 months ago
📄 PROGI_2023_Proggers_prezentacija...	Dodana prezentacija projekta.	5 months ago
📄 README.md	Update README.md	5 months ago
📄 dnevnik.txt	Update-an dnevnik.txt	7 months ago

Slika 3.1 Prikaz korijenskog direktorija preko GitHuba za grupu *Proggers*

Potprogrami vraćaju rezultate u svoje zasebne tekstualne datoteke `podaciOsnovni.txt` i `podaciSadrzaj.txt` gdje se podaci spremaju. Bitna funkcionalnost programa je spremanje foldera koji su se analizirali pomoću pomoćnih datoteka `obrađeniOsnovni.txt` i `obrađeniSadrzaj.txt` tako da te datoteke nisu namijenjene za korištenje. Te datoteke služe kao lista analiziranih repozitorija tako da se analiza u bilo kojem trenutku može prekinuti i nastaviti kasnije.

3.1 Potrebne instalacije

Kako bi se programski kod mogao pokretati potrebno je imati instaliran Python3 te knjižnicu PyGithub 2.3.0. Program je napisan u Python verziji 3.9. Iako je moguće da program funkcionira na nižim verzijama Pythona i nižim verzijama PyGithuba, preporučeno je da se koristi minimalno verzija Python verzija 3.9 te minimalno PyGithub verzija 2.3.0 jer će s tim verzijama program sigurno raditi. Instalacija Pythona se može ostvariti uputama sa službene Python stranice: <https://www.python.org/downloads/>. Također PyGithub se može instalirati pomoću instalatora paketa za Python – pip pomoću naredbe:

```
pip install PyGithub
```

Ako je slučaj da pip instalator nije instaliran na računalu može se instalirati pomoću uputa sa stranice službene pip dokumentacije: <https://pip.pypa.io/en/stable/installation/>.

3.2. Povezivanje s GitHub API-jem

Na početku programa bilo je potrebno uvesti tražene pakete:

```
from github import Auth  
  
from github import Github  
  
import sys
```

Kod 3.1 Uvoženje paketa

Uvoženjem paketa se omogućava korištenje funkcionalnosti iz knjižnice PyGithub. Kako bi program imao pristup GitHub repozitorijima potrebno se autentificirati pomoću tokena:

```
githubToken=sys.argv[1]

auth=Auth.Token(githubToken)

g=Github(auth=auth)
```

Kod 3.2 Autentifikacija tokenom

Varijabla `githubToken` se prvi argument funkcije i koristi se u navedenim funkcijama za autentifikaciju korisnika čijim se repozitorijima želi pristupiti. Poslije autentifikacije GitHub daje dopuštenje za traženje i dobivanje informacija od GitHub API-ja.

3.3. Obrada osnovnih podataka repozitorija

Klasa `repozitorijOsnovniPodaci` napravljena je za parsiranje podataka uzetih preko GitHub API-ja stavljajući osnovne podatke o nekom repozitoriju u isti spremnik (container).

```
class repositoryOsnovniPodaci:

    self.ime=repo.name

    self.stvoren=repo.created_at

    self.doprinostitelji=repo.get_contributors()

    self.jezici=repo.get_languages()
```

Kod 3.3 klasa `repositoryOsnovniPodaci`

U instancu klase `repositoryOsnovniPodaci` se pohranjuje ime repozitorija (`repo.name`), vrijeme stvaranja repozitorija (`repo.created_at`), listu sudionika na repozitorija (`repo.get_contributors()`) i listu svih jezika (`repo.get_languages()`) koja će biti detaljnije obrađena u funkciji `postociJezika()`.

Sljedeća funkcija koja treba biti objašnjena je funkcija `postociJezika`:

```
def postociJezika(jezici):

    suma=0

    povratnaVrijednost=""
```

```

for jezik in jezici:
    suma+=jezici[jezik]

for jezik in jezici:
    ret+="\n"+jezik+" "+"{:.2f}"

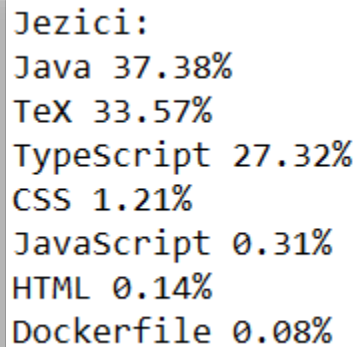
    .format(jezici[jezik]/suma*100)+"%"

return povratnaVrijednost

```

Kod 3.4 funkcija postoci jezika

Funkcija `repositoryOsnovniPodaci` prima argument `jezici` koji će zapravo biti atribut `jezici` objekta klase `repositoryOsnovniPodaci` pri pozivu funkcije. Funkcija najprije zbraja broj bajtova svih tipova funkcija u varijablu `suma`, a onda ukupnu veličinu svih podataka nekog tipa podataka u bajtovima dijeli s ukupnom veličinom svih tipova podataka u repozitoriju. Funkcija vraća vrijednost `povratnaVrijednost` koji će se pri završetku izvršavanja funkcije biti oblikovan u string koji će prikazivati u podacima postotni udio neke vrste programa u repozitoriju.



```

Jezici:
Java 37.38%
TeX 33.57%
TypeScript 27.32%
CSS 1.21%
JavaScript 0.31%
HTML 0.14%
Dockerfile 0.08%

```

Slika 3.2 Primjer ispisa funkcije `postociJezika`

Sam zapis se zapisuje u datoteke s podacima uz pomoć funkcije `upisiSveOsnovniPodaci`, a ona je zapravo članska funkcija klase `repozitorijOsnovniPodaci`:

```

def upisiSveOsnovniPodaci(self, podaci):
    podaci.write("Ime direktorija:\n"+self.ime)
    podaci.write("\n\nStvoreno:\n" + str(self.stvoren.date()))
    podaci.write("\n\nSuradnici:")
    for doprinositelj in self.doprinositelji:
        podaci.write("\nIme-"+str(doprinositelj.name)+
            " login-"+ doprinositelj.login)
    podaci.write("\n\nJezici:")
    podaci.write(postociJezika(self.jezici))
    podaci.write("\n\n\n")

```

Kod 3.5 funkcija za upis osnovnih podataka

Funkcija `upisiSveOsnovniPodaci` je namijenjena je za upis podataka u prvu datoteku `podaciOsnovni.txt`. Ime svakog atributa klase i njegova vrijednost za određeni repozitorij su odvojeni jednim znakom za novi red ("`\n`"). Takvi različiti skupovi podataka za različite attribute objekta klase repozitorij su odvojeni s dva znaka za novi red ("`\n\n`"), dok su različiti cjelokupni zapisi za pojedine repozitorije odvojeni s tri znaka za novi red ("`\n\n\n`").

Do sada su obrađene sve funkcije potrebne za generiranje i spremanje podataka o repozitorijima te je sad preostao samo izvršni dio:

```

podaci=open("podaciOsnovni.txt", "a")
obrađeniR=open("obrađeniOsnovni.txt", "r")
obrađeniA=open("obrađeniOsnovni.txt", "a")
brojac=0
obrađeniR=obrađeniR.read().split("\n")
for repo in g.get_user().get_repos():
    if(repo.name not in obrađeniR):
        repozitorij(repo).upisiSve1(podaci)
        obrađeniA.write(repo.name+"\n")

```



```
print(brojac)
```

3.6 Kod povezivanja s datotekama, stvaranja instance klase i pozivanja potrebnih funkcija

U ovom kodu se završava obrada prvog skupa podataka te se ti podaci zapisuju u datoteku `podaciOsnovni.txt` u "a" (append) načinu tako da se cijela datoteka ne briše pri ponovnom pokretanju programa već pamti već obrađene repozitorije. Ta funkcionalnost se ostvaruje čitanjem i zapisivanjem u datoteku `obrađeniOsnovni.txt` gdje se zapisuju imena repozitorija nakon što su njihovi podaci upisani u datoteku `podaciOsnovni.txt`. Prije obrade repozitorija se vrši provjera je li ime tog repozitorija u datoteci `obrađeniOsnovni.txt` te ako ima obrada podataka tog repozitorija se neće izvršiti. varijabla `g` u ovom dijelu koda je zapravo instanca klase `github` koja predstavlja našu vezu s GitHub API-jem te se preko naredbe `g.get_user().get_repos()` dobiva lista svih repozitorija kojim korisnik ima pristup. Varijabla `brojac` je tu kao dodatna pomoć i ispisivanje koliko potprograma je obrađeno od pokretanja programa (nakon obrade svakog repozitorija će se povećati za jedan i ispisati na standardni ulaz).

```
Ime:
Progers

Stvoreno:
2023-10-16

Doprinostelji:
ime-Sven Winkler login-1winks
ime-lk login-lk534
ime-None login-BozidarPucar
ime-Yu Xing Jin login-kkkkkkkkkk225
ime-Andrej Lovei login-Cyb3rSab3r
ime-None login-DamjanSarlija

Jezici:
TeX 48.05%
JavaScript 25.41%
Java 16.34%
CSS 8.64%
HTML 1.10%
Dockerfile 0.46%
```

Slika 3.3 Cjelokupni ispis grupe *Progers* iz datoteke podaciOsnovni.txt

3.4. Obrada sadržaja direktorija repozitorija

Za parsiranje drugog dijela podataka se koristi klasa `repozitorijSadrzaj`:

```
class repozitorijSadrzaj:

    def __init__(self, repo):

        self.ime = repo.name

        self.sadrzaj = repo.get_contents("")

        self.korjenskaStrukturaRepozitorija={}

        self.dubinskaStrukturaRepozitorija=[]

        ...
```

Kod 3.7 Prvi dio klase `repozitorijSadrzaj`

U objekt klase `repozitorijSadrzaj` pohranjuje se ime repozitorija (`repo.ime`), njegov sadržaj (`repo.sadrzaj`) u obliku putanje do svih datoteka i repozitorija koji se nalaze u korijenskom direktoriju `repozitorija`. Atributi `korjenskaStrukturaRepozitorija` i `dubinskaStrukturaRepozitorija` su samo inicijalizirani u konstruktoru te se dodaju u objekt kasnije. Atribut `korijenskaStrukturaRepozitorija` će sadržavati analizu direktorija koji se nalaze u korijenskom direktoriju `repozitorija`. Preciznije taj atribut će navoditi koji tipovi podataka su zastupljeni i koliko su ti tipovi podataka zastupljeni u pojedinom direktoriju u samom korijenu `repozitorija`. Atribut `dubinskaStrukturaRepozitorija` će navoditi koji tipovi podataka se nalaze u svakom direktoriju u `repozitoriju`.

Zadnji dio klase `repozitorijSadrzaj` je provjera prisutnosti alata za testiranje Selenium:

```
...
self.sadrzaj = repo.get_contents("")
self.selenium = False
sadrzajKopija=self.sadrzaj.copy()
while sadrzajKopija:
    datotekaSadrzaj = sadrzajKopija.pop(0)
    if datotekaSadrzaj.type == "dir":
        sadrzajKopija.extend(repo.
            get_contents(datotekaSadrzaj.path))
    elif datotekaSadrzaj.type == "file":
        if "pom.xml"==datotekaSadrzaj.name.lower() or
            "package.json"==datotekaSadrzaj.name.lower():
            seleniumSadrzaj =
                repo.get_contents(datotekaSadrzaj.path)
                .decoded_content.decode("utf-8")
```

```

        if("selenium" in str(seleniumSadrzaj)):

            self.selenium=True

            break

```

Kod 3.8 Dio klase `repozitorijSadrzaj` sadržaj za ispitivanje prisutnosti Selenium alata

Program na početku pretpostavlja da se u repozitoriju nije koristio Selenium te će zaključiti suprotno ako se nađe jedna od konfiguracijskih datoteka po imenu `"pom.xml"` ili `"package.json"`. Pri pronalasku tih datoteka one se otvaraju u tekst formatu te se pretražuje ključna riječ “selenium“ u njima. Ako je ključna riječ pronađena vrijednost `selenium` atributa se podešava na `True` te se proces zaustavlja naredbom `break`.

Zadnji dio konstruktora klase `repozitorijSadrzaj` su atributi `readMe` i `licenca`.

```

self.ReadMe=None

self.licenca=None

try:

    self.ReadMe = repo.get_readme()

except:

    self.ReadMe = False

try:

    self.licenca = repo.get_license()

except:

    self.licenca = False

```

Kod 3.9 Dio klase `repozitorijSadrzaj` koji provjerava prisutnost licence i `ReadMe` datoteka

Navedeni dio koda pokušava pristupiti `ReadMe` i `licenca` datoteci repozitorija. Ako one postoje spremaju se u svoje odgovarajuće attribute dok će se vrijednost njihovih atributa ako ne postoje postaviti u `False`.

Upisivanje ispisa sadržaja klase `repozitorijSadrzaj` je ostvareno funkcijom `upisiSveSadrzaj` koja upisuje podatke u datoteku `podaciSadrzaj.txt`:

```
def upisiSveSadrzaj(self, podaci):

    podaci.write("Ime direktorija:\n"+self.ime)

    podaci.write(self.ime+"\n\nKorijenska struktura
    direktorija:\n")

    for i in self.korjenskaStrukturaRepozitorija:

        podaci.write(i)

        for j in self.korjenskaStrukturaRepozitorija[i]:

            podaci.write("|" + j + ":" +

                str(self.korjenskaStrukturaRepozitorija[i][j]))

        podaci.write("\n")

    podaci.write("\nDubinska struktura direktorija:\n")

    for i in self.dubinskaStrukturaRepozitorija:

        podaci.write(i)

        for j in self.dubinskaStrukturaRepozitorija[i]:

            podaci.write("|" + j + ":" +

                str(self.dubinskaStrukturaRepozitorija[i][j]))

        podaci.write("\n")

    if(self.ReadMe==False):

        podaci.write("\nRepozitorij ne sadrzi ReadMe\n")

    else:

        podaci.write("\nRepozitorij sadrzi ReadMe\n")

    if(self.licenca==False):

        podaci.write("\nRepozitorij ne sadrzi licencu\n")

    else:

        podaci.write("\nRepozitorij sadrzi licencu\n")

    if (self.selenium == False):
```

```

        podaci.write("\nRepozitorij ne koristi Selenium alat
        za testiranje\n")
    else:
        podaci.write("\nRepozitorij koristi Selenium alat za
        testiranje\n")
    podaci.write("\n\n\n")

```

Kod 3.10 Funkcija `upisiSveSadrzaj` koja ispunjava sadržaj datoteke `podaciSadrzaj.txt`

Za analizu datoteka u repozitoriju ih je potrebno klasificirati ih u datoteke prednjeg, datoteke stražnjeg kraja te datoteke dokumentacije. Datoteke dokumentacije su trenutnom slučaju studentskih radova morale biti napisane u Latex formatu. Datoteke Latex formata se mogu vrlo lako identificirati pomoću njihovog nastavka “.tex“. Zbog otvorene prirode projekta postoje razne vrste datoteka prednjeg i stražnjeg kraja. Ako su korištene još neke tehnologije potrebno je dodati njihove nastavke u liste na pod nazivima `nastavciStraznjegKraja` i `nastavciPrednjegKraja`:

```

nastavciStraznjegKraja=["py", "java", "cpp", "c", "cs"]
nastavciPrednjegKraja=["html", "css", "js", "ts", "php", "dart",
                        "tsx", "jsx", "ejs"]

```

Koda 3.11 Lista nastavaka datoteka za klasificiranje

Podaci različitih direktorija su razmaknuti s dva prazna reda kao što su i u prijašnjoj datoteci. Zapis počinje s navođenjem imena direktorija te se zatim navodi već objašnjena korijenska i dubinska struktura cijelog repozitorija (uvod trenutnog poglavlja). Strukture repozitorija se ispisuju tako da se ispiše putanja direktorija te odvojeno znakom "|" tip datoteka u tom direktoriju i broj datoteka tog tipa.

```
Korijenska struktura direktorija
Dokumentacija|doc:9
Izvornikod|front:42|back:61

Dubinska struktura direktorija
Dokumentacija|doc:9
Izvornikod/frontend|front:42|back:61
Izvornikod/frontend/src|front:4
Izvornikod/backend/src/main|back:1
Izvornikod/backend/src/test|back:1
Izvornikod/frontend/.vite/deps|front:19
Izvornikod/frontend/src/components|front:17
Izvornikod/backend/src/main/java/com/lostpetfinder|back:1
Izvornikod/backend/src/test/java/com/lostpetfinder|back:5
Izvornikod/backend/src/main/java/com/lostpetfinder/config|back:1
Izvornikod/backend/src/main/java/com/lostpetfinder/controller|back:4
Izvornikod/backend/src/main/java/com/lostpetfinder/dao|back:12
Izvornikod/backend/src/main/java/com/lostpetfinder/dto|back:10
Izvornikod/backend/src/main/java/com/lostpetfinder/entity|back:14
Izvornikod/backend/src/main/java/com/lostpetfinder/exception|back:2
Izvornikod/backend/src/main/java/com/lostpetfinder/service|back:6
Izvornikod/backend/src/main/java/com/lostpetfinder/utils|back:2
Izvornikod/backend/src/main/java/com/lostpetfinder/entity/pkeys|back:2
```

Slika 3.4 Primjer ispisa strukture repozitorija za grupu *Progers*

Nakon ispisa struktura direktorija repozitorija se ispisuje prisutnost ReadMe-ja, licence i korištenja alata Selenium.

```
ReadMe:  
Repozitorij sadrzi ReadMe  
  
Licence:  
Repozitorij ne sadrzi licencu  
  
Selenium:  
Repozitorij ne koristi Selenium alat za testiranje
```

Slika 3.5 Primjer ispisa strukture repozitorija za grupu *Progers*

Jedini dio koda koji još nije opisan je samo pretraživanje direktorija po repozitoriju. Taj dio je nešto kompleksniji i dosta duži no princip mu je u suštini jednostavno iterativno pretraživanje po direktorijima. Nakon izvlačenja svih direktorija nekog repozitorija se za svaki repozitorij dohvaćaju datoteke koje se u njemu nalaze te se gleda koliko je datoteka prednjeg kraja, stražnjeg kraja i koliko je datoteka dokumentacije te se svi podaci spremaju u atribut klase `repozitorijSadrzaj` zvan `sadrzajDubinskihDirektorija`. Sada kada je dohvaćen sadržaj dubinskih direktorija iz njega je moguće dohvatiti sadržaje korijenskih direktorija jer putanja svakog dubiskog direktorija počinje s jednim od korijenskih direktorija. Za sve korijenske direktorije se tada iz riječnika `sadrzajDubinskihDirektorija` zbrajaju svi tipovi podataka koje direktoriji koji su njegovi podirektoriji sadrže. Ti podaci se spremaju u atribut `korijenskaStrukturaRepozitorija`.

3.5. Problemi s identifikacijom

Iz ispisa prvog podzadatka je vidljivo da ime korisnika nije prisutno svim članovima repozitorija (na mjestima gdje nije prisutno piše None). Korisnička imena za prijavu („`login`“) su obavezna no većinu sudionika na repozitorijima nije moguće identificirati s njima. Oni moraju biti unikatni, a to rezultira korisničkim imenima „lwinks“ i „kkkkkkkkkk225“ koji su vidljivi iz primjera ispisa iz kojih nikako ne možemo identificirati korisnika, a ne možemo ga identificirati pomoću punog imena i prezimena („`ime`“) zato što je ono neobavezno. Također je bitno za znati da se preko GitHub API-ja nikako ne može doći do E-mail adrese korištenu za kreiranje GitHub računa nekog sudionika repozitorija zbog zaštite privatnosti korisnika GitHuba. Preporuka kod korištenja ove programske potpore ako je uz nju potrebna identifikacija korisnika GitHub repozitorija je da se obavijesti sudionike repozitorija da u polje „`ime`“ napišu svoje ime i prezime ili alternativno da se u polje „`ime`“ napiše drugi jedinstveni identifikacijski parametar kojim korisnik programske potpore ima pristup. Na primjer u slučaju gdje profesor koristi ovu programsku potporu kod analiziranja studentskih radova gdje ima pristup JMBAG-ovima (jedinstveni matični broj akademskog građana) studenata, bilo bi najbolje u polje „`name`“ staviti JMBAG svakog studenta.

4. ZAKLJUČAK

U ovom radu su prikazane funkcionalnosti GitHub API-ja te je vidljivo da je on moćan alat za analiziranje repozitorija, no programsko inženjerstvo nije toliko jednostavno područje nad kojim bi se vršila analiza. Svaka repozitorij bi strukturno morao slijediti neke osnovne prihvaćene smjernice no teško je generički reći točno kakva je struktura dobra, a koja odlična jer ona sama ovisi korištenim tehnologijama i općenito drugačijim pristupima razvijanju programske potpore. No ovaj rad je dobra pripomoć pri ocjenjivanju radova predmeta Programskog inženjerstva što je bila i početna motivacija za pisanje rada. Pomoću rada ocjenjivači studentskih projekata mogu dobiti uvid gdje se nalaze datoteke koje bi ih mogle interesirati, jesu li prisutni ključni dijelovi repozitorija te koje su tehnologije koristili svaki od repozitorija.

5. LITERATURA

- [1] A. Jović, N. Frid, PROCESI PROGRAMSKOG INŽENJERSTVA, 5. izdanje, Zagreb, rujan 2022.
- [2] EduBridge India, SDLC Models, What is It? [Online]. Dostupno: <https://www.edubridgeindia.com/blog/sdlc-models/> [Pristupljeno: 11-6-2024]
- [3] GitHub REST API documentation [Online]. Dostupno: <https://docs.github.com/en/rest/about-the-rest-api/about-the-rest-api?apiVersion=2022-11-28> [Pristupljeno: 11-6-2024]
- [4] PyGithub — PyGithub 2.3.0 documentation [Online]. Dostupno: <https://pygithub.readthedocs.io/en/latest/index.html> [Pristupljeno: 11-6-2024]

6. SAŽETAK

Analiza procesa timskog razvoja programske potpore uporabom GitHub API-a

Zadatak ovog rada je analiza procesa timskog razvoja programske potpore uporabom GitHub API-a. Razvoj programske potpore u današnje vrijeme zahtijeva sve veći timski rad te koriste razne tehnologije kao što je Git kako bi se taj timski rad lakše ostvario. Cilj ovog rada je analiza strukture Git repozitorija preko GitHub platforme koristeći GitHub API. Kod koji koristi GitHub API je pisan u Pythonu te je kao poveznica s GitHub API-jem korišten Python paket PyGithub. Rad prolazi kroz sve Github repozitorije nekog korisnika te zapisuje podatke o njima u dvije datoteke pod imenom `podaciOsnovni.txt` i `podaciSadrzaj.txt`. U datoteku `podaciOsnovni.txt` se spremaju podaci o datumu kreiranja datoteke, svim doprinositeljima na repozitoriju gdje su navedena njihova imena (ako su prisutna) te njihova korisnička imena programskim jezicima i tehnologijama korištenim u repozitorijima. u datoteci `podaciSadrzaj.txt` će biti zapisana struktura repozitorija s ispisom prisutnosti i količina datoteka koje bi mogle zanimati ocjenjivača. U tu datoteku će također biti upisani podaci o prisutnosti datoteka licence i ReadMe-a te će se također provjeravati prisutnost korištenja alata za testiranje Selenium.

6. SUMMARY

Analysis of the Software Team Development Processes Using GitHub API

The task of this paper is analysis of the software team development processes using GitHub API. The development of software nowadays requires an increasing amount of teamwork, and various technologies such as Git are used to make teamwork easier. The goal of this paper is to analyze the structure of the Git repository on the GitHub platform using the GitHub API. The code that uses the GitHub API is written in Python, and the Python package PyGithub is used as a link to the GitHub API. The work goes through all the GitHub repositories of a user and writes data about them in two files named `podaciOsnovni.txt` and `podaciSadrzaj.txt`. Data describing the file creation date, contributors with their names (if present), and usernames as well as the programming languages and technologies used in the repositories will be listed in the file `podaciOsnovni.txt`. The file `podaciSadrzaj.txt` will contain the structure of the repository with a printout of the presence and amount of files that might be of interest to the evaluator. This file will also include data about the absence or presence of license and ReadMe files and the use of the Selenium testing tool.