Alati za razvoj softvera

Git

Prvobitno su programeri pravili različite kopije programa, međutim na taj način ne možemo da znamo šta je bilo u kojoj verziji. Kasnije su u sam naziv kopija dodavani datumi kako bismo znali kada smo šta radili. Sledeći korak je bio da se u ime direktorijuma dodaje i opis izmene koju smo napravili. Ovo je u redu za lične backup-e, ali je problem što svi direktorijumi zauzimaju prostor na disku, čak iako je izmena mala. Ako želimo da podelimo drugima svoj kod, možemo da kompresujemo svoj program i pošaljemo ga, a drugi programer bi uradio isto nakon što odradi svoj deo posla što postaje previše komplikovano. Kao jedno poboljšanje koristi se komanda `diff` koja nam ispisuje razlike između dva koda. Pomoćni programer sada šalje samo izmene, a mi zatim te promene primenimo na svoju verziju:

```
# Razlike između fajlova našeg (originalnog) programa i izmenjenog upisujemo u IZMENE.diff
diff IZMENJEN_PROGRAM ORIGINALNI_PROGRAM > IZMENE.diff
# Menjamo originalni program
patch -p1 ORIGINALNI_PROGRAM < IZMENE.diff
```

Stvar se komplikuje ako imamo veći broj programera. U tom slučaju možemo uzeti centralni server gde čuvamo svaki projekat, kako trenutnu verziju tako i sve diff-ove koji su učestvovali u izgradnji projekta pomoću kojih možemo da se vraćamo unazad kroz projekat. Sada više nemamo "glavnog" programera, već se svi obraćaju serveru. Svako slanje i primena diff-ova se automatizuje uz pamćenje datuma, poruka programera i slično pa se može reći da smo došli do **sistema za kontrolu verzija**. Najpopularniji softver ovog tipa bio je **subversion (svn)** koji danas više nije popularan. Problem nastaje kada je potrebno da komuniciramo sa serverom, a nismo u mogućnosti (npr. nemamo internet konekciju). Umesto jednog centralnog servera, ideja je da svi programeri imaju celokupnu kopiju celog repozitorijuma i cele istorije, odnosno sistem za kontrolu verzija postaje **distribuiran**. Ovde postoje dva problema:

1. Kada dođe do izmene da li šaljemo svima šta smo izmenili (ovo je verovatno lakše) ili kada želimo da radimo nešto tražimo od svih programera njihove poslednje verzije?
2. Sve verzije su ravnopravne jer je sistem distribuiran, pa je pitanje čija verzija se šalje kom klijentu? Potrebno je uvesti neku hijerarhiju među programerima. Jedno rešenje je opet uvesti neki centralni server koji bi komunicirao sa svim korisnicima.

Danas najpopularniji sistem za praćenje verzija je **Git**. **GitHub** i **GitLab** su servisi koji implementiraju Git na neki način i nisu isto što i Git. Prazan **Git repozitorijum** (nije stvarno prazan, sadrži neke informacije) pravimo na sledeći način:

```
git init
```

Informacije o Git repozitorijumu:

```
git status
```

Dodavanje fajlova za praćenje:

```
git add FAJL
```

Izvršni fajlovi ili bilo koji fajlovi koji se generišu pokretanjem programa se ne dodaju u Git repozitorijum. Da bismo automatski ignorisali fajlove koje ne treba dodavati možemo napraviti **.gitignore** fajl i u njega upisati sve što Git treba da ignoriše. Fajl .gitignore treba dodati u repozitorijum. Primer .gitignore fajla:

```
program
*.o
```

Git će ignorisati fajlove koji se nazivaju "program", kao i fajlove koji imaju .o ekstenziju. **Komitovanje** promena vrši se pomoću komande:

```
git commit
git commit -m KOMENTAR
```

Komituju se samo ispravni programi. **Istorija** repozitorijuma:

```
git log
git log -- PUTANJA # istorija samo unutar određenog poddirektorijuma
git whatchanged # detaljniji prikaz šta je promenjeno
alias glol='git log --oneline --all -decorate --graph' # alias za lepši ispis istorije
```

Alternativno se može napraviti **alijas** u gitconfig fajlu u delu sa alijasima. Pozivaju se sa:

```
git IME_ALIJASA
```

Razlike između naše verzije i:

```
# Poslednje add komande
git diff
# Poslednjeg komita
git diff HEAD
# Pretposlednjeg komita
git diff HEAD~1
# Određenog komita čiji ID uzimamo iz log-a
git diff KOMIT_ID
```

Glavna grana Git repozitorijuma naziva se **master** i predstavlja glavnu razvojnu verziju projekta. Moguće je praviti više grana za svake od stvari na kojima se trenutno radi, umesto da svi rade na istoj grani. **Pravljenje grane**:

```
git branch IME_GRANE
```

Neka konvencija je da ime grane sadrži bitne informacije, na primer: "david/bag1-ne-radi-izlaz". Prelazak na drugu granu:

```
git checkout IME_GRANE
```

Spajanje grane IME_GRANE na trenutnu granu u kojoj se nalazimo:

```
git merge IME_GRANE
```

Ukoliko na više grana radimo na istom delu koda i pokušamo da spojimo grane doći će do **konflikata** koje moramo ručno ispraviti. Mesto konflikta je u kodu obeleženo strelicama i znakovima jednakosti. Za svaki konflikt sami odlučujemo šta želimo da ostavimo u programu, a šta brišemo. Na primer:

```
void function() {
<<<<<<< HEAD
    // TODO: implement this function
=======
    std:cout << "Something\n";
>>>>>>> david/implementirana-funkcija
}
```

Kloniranje repozitorijuma:

```
git clone LINK
```

Naš lokalni repozitorijum imaće pokazivač na originalni repozitorijum pod nazivom **origin**. Pravljenje novog pokazivača pod imenom "personal":

```
git remote add personal SSH_KEY
git remote -v # svi pokazivači
```

Na ovaj način sada kada radimo lokalno možemo da šaljemo izmene ili na originalni repo ili na klonirani. Svaki pokazivač ima jednu instancu za push i jednu instancu za fetch. Spisak svih grana:

```
git branch --all
```

Razvoj različitih verzija odvija se na posebnim granama. Kada želimo da objavimo neku verziju, ne pravimo novu granu već pravimo **tag** koji je read-only pokazivač. Grane služe da se nastavi razvoj, a tagovi samo daju ime određenom komitu. Komande za tagove:

```
git tag --all # svi tagovi
git checkout IME_TAGA # prelazak na tag
```

Kloniranje grana:

```
git checkout --track -b IME_LOKALNE_GRANE IME_KOPIRANE_GRANE
```

Kada kloniramo udaljenu granu mi kreiramo lokalnu granu koja je prati na neki način. Kada radimo push/pull na kloniranoj grani ustvari radimo sa udaljenom granom. Premeštanje grane sa originalnog repozitorijuma na klonirani:

```
git push personal IME_GRANE:NOVO_IME_GRANE
git push personal IME_GRANE # ako smo prethodno iskoristili --track
```

Grane za razvoj različitih verzija uglavnom zauvek imaju odvojene istorije. Ako razvijamo neku novinu to radimo na **feature** granama, a zatim kada završimo promene izbacujemo na master granu. **Slanje izmena na master** sa neke druge grane:

```
git checkout master # pređemo na master
git merge IME_GRANE # ubacujemo izmene sa grane "IME_GRANE" na trenutnu granu (master)
```

Ako je neko uporedo izbacio svoje izmene na master git objavljuje komitove oba programera, a od njih se očekuje da razreše konflikte ako ih ima. Sa komandom **rebase** se sa mastera kopiraju svi commitovi na vrh grane koju prosleđujemo:

```
git rebase IME_GRANE
```

Git prepoznaje sve komitove na našoj grani od kada smo se odvojili sa mastera, kao i sve komitove na masteru od tog trenutka izdvajanja, a zatim komitove mastera kopira na vrh naše istorije. Dakle, merge spaja dve istorije, a rebase piše novu istoriju. Rebase se koristi kada npr. komitove naše feature grane hoćemo da prebacimo na master:

```
git rebase master
```

Drugi (bolji) način je:

```
git merge master
```

čime osvežavamo našu granu i dodajemo izmene sa mastera, što treba raditi s vremena na vreme. **Kretanje po istoriji**:

```
git checkout COMMIT_ID # vraćamo se na prosleđeni commit
git reset HEAD~1 # brišemo poslednji komit, ali kod ostaje isti
git reset --hard HEAD~1 # brišemo poslednji komit i vraćamo se na prethodnu verziju programa
```

Reset ne sme da se koristi ako nismo na nekoj lokalnoj grani, već na grani koja je sinhronizovana sa ostalim korisnicima. Tada treba napraviti novi commit koji poništava prosleđeni commit:

```
git revert COMMIT_ID
```

Reset i revert ne mogu da obrišu fajlove koji nisu komitovani. Ako želimo da obrišemo fajl koji nikad nismo dodali i komitovali možemo koristiti sledeći trik:

```
git add . # dodamo sve fajlove (dodaće se oni koji do sada nisu bili dodani)
git commit -m kill # pravimo komit proizvoljnog imena
git reset --hard HEAD~1 # brišemo taj komit, a samim tim i fajlove
```

Ako želimo da **poništimo** neke izmene na fajlovima, ali da ih ipak sačuvamo koristimo:

```
git stash
```

Izmene su poništene, ali sačuvane. Ako ponovo želimo da primenimo te izmene koristimo:

```
git stash pop
```

Ako smo stashovali više stvari, pop će vratiti samo poslednju. Lista svih stvari u stashu:

```
git stash --all
```

Ako se isti bag pojavljuje u više verzija našeg koda, treba ga ispraviti u jednoj grani, a zatim koristiti merge da se izmene primene i na druge grane (verzije) u slučaju kada grane nisu previše divergirale. Ako su grane previše

divergirale, možemo koristiti komandu **cherry-pick** koja će prebaciti samo jedan komit iz jedne grane u drugu. Potrebno je da prvo pređemo na granu gde želimo da dodamo komit, a zatim iskoristimo:

```
git cherry-pick COMMIT_ID
```

čime na tu granu kopiramo komit druge grane.

Ako ne želimo svaki put da kucamo šifru pri push ili pull komandama, možemo da je sačuvamo jednom i biće zapamćena za trenutnu sesiju. To radimo preko **ssh-agenta** na sledeći način:

```
ssh-agent > /tmp/ssh-env # sačuvamo ssh-agent u neki fajl
. /tmp/ssh-env # izvršimo fajl, tj. pokrećemo ssh-agent
ssh-add NAZIV_KLJUČA # dodajemo konkretan SSH ključ
```

Nakon dodavanja ključa ponudiće nam se da ukucamo šifru, koja će onda biti zapamćena za trenutnu sesiju.

Kada dođemo u situaciju da postoji bag, a sećamo se da pre nekoliko komitova nije postojao, možemo ići ručno od ispravnog komita ka neispravnom i proveravati svaki. Možemo ići i binarnom pretragom, umesto linearnom, ali to opet zahteva vreme. Komanda **bisect** to obavlja umesto nas:

```
git bisect start
git bisect good ID_DOBROG_KOMITA
git bisect bad master
```

Nakon ovoga, git će uzeti komit između ova dva. Nakon testiranja, potrebno je da mu kažemo da li je taj komit dobar ili loš:

```
git bisect good # trenutni komit je dobar
git bisect bad # trenutni komit je loš
```

Nakon svakog koraka uzima se novi interval i traži se prvi komit gde je nastao problem. Kada smo završili izlazimo iz algoritma bisekcije:

```
git bisect reset
```

Ako često radimo na nekoliko grana i stalno prelazimo sa jedne na drugu, pri svakom prelasku kompilacija traje dugo jer se fajlovi razlikuju. Umesto toga, možemo te grane dodati u neku vrstu "klona" repozitorijuma preko **worktree** komande:

```
git worktree add NAZIV_DIREKTORIJUMA NAZIV_GRANE
```

Napraviće se novi direktorijum gde će grana biti dodata. Ne pravi se stvarni novi repozitorijum, već on deli isti .git direktorijum kao i originalni, tako da se ne zauzima više mesta. Imamo privid kao da radimo na dva repozitorijuma i ne moramo da menjamo grane. Build direktorijum se pravi za svaku granu ponaosob. Dok je grana u worktree modu, ne može da se prelazi na nju iz originalnog repozitorijuma ili iz drugih worktree direktorijuma. Nakon završetka rada, izbrišemo granu iz direktorijuma:

```
git worktree remove NAZIV_GRANE
git worktree list # prikaz svih grana u posebnim direktorijumima
```

Nekada želimo i da u naš projekat uključimo neki drugi. Git omogućava da projekat sadrži veze ka drugim projektima - **submodule (podmodule)**:

```
git submodule add LINK_DO_GIT_REPOZITORIJUMA
```

Kada kloniramo repo koji ima submodule, oni se neće automatski kopirati, već moramo pozvati:

```
git submodule update --init --recursive
```

Opcija --recursive se stavlja ako i ti submoduli imaju svoje submodule. U glavnom repou dobijamo direktorijum koji se naziva **.gitmodules** gde se nalazi spisak svih submodula.

Komande za pretraživanje (npr. grep) pretraživaće sve fajlove u našem repozitorijumu. Komanda **git grep** pretražuje samo fajlove koji su vidljivi gitu i koristi se kao i grep:

```
git grep NISKA
```

GitLab

Merge request omogućava da svi članovi tima pre pushovanja mogu da pogledaju i komentarišu izmene. Naslov bi trebao da bude kratak i jasan, a u opisu objašnjavamo šta je promenjeno, zašto i zbog čega je ovaj pristup bolji od nekih drugih. Ako je komit popravljao neki bag može se ostaviti i informacija o tome koji bag je ispravljen. Može se napisati i ako nijedan bag nije ispravljen, ali komit ima veze sa nekim bagom. Na primer:

```
BUG !42 -- Code is obsolete # bag popravljen
CCBUG !42 -- Code is obsolete # komit ima veze sa bagom
```

Polje **Assignee** označava osobu koja je zadužena za merge request. **Reviewer** su ljudi ili osoba od kojih tražimo pregled. **Milestone** označava koji cilj smo hteli da postignemo. Moguće je dodavanje i **labela** za više informacija. Klikom na dugme **Approve** govorimo da je spajanje dozvoljeno, ali ono i dalje treba da se uradi posebno. Prilikom pregleda koda, moguće je dodavanje **komentara**, na konkretne linije koda. Kada ispravimo problem, možemo zatvoriti diskusiju sa **Resolve Thread**. Stvari koje treba proveravati pri ocenjivanju koda:

- **korektnost dizajna koda**
- **imenovanje funkcija i promenljivih**
- **složenost**
- **C++ korektnost** (ako se koristi C++)
- **pokrivenost testovima**
- **dokumentacija**
- **stil i formatiranje koda**

GDB

Uprkos postojanju **debagera**, mnogi programeri i dalje programe debaguju koristeći printf-ove i analiziranje izlaza nakon završetka programa. Sam debager može da radi isto, ako koristimo samo osnovne opcije i idemo korak po korak kroz kod. Naravno, debager će to raditi brže i možemo debagovati veće programe na ovaj način - linearno. Kada tražimo bagove, uglavnom tražimo mesta gde se naše pretpostavke razlikuju od stvarnog stanja. Umesto linearnom, možemo ići binarnom pretragom. U stvarnosti retko problem tražimo zaista binarnom pretragom, već

nekom logikom tražimo delove koda gde verujemo da bi bag mogao biti. Najveći problem printf-ova je stalno brisanje i dodavanje linija i ponovna kompilacija. Postoje i uređaji koji nemaju opciju ispisa poruka na neki displej.

GDB (GNU Debugger) se može koristiti na dva načina - pokrećemo program iz gdb-a ili gdb zakačimo za aplikaciju koja je već pokrenuta (ovaj pristup često nije dozvoljen zbog sigurnosti). Program prvo pokrećemo na sledeći način:

```
gdb PUTANJA_DO_FAJLA
```

Na ovaj način otvara se gdb prompt. Ako želimo da pokrenemo program iz gdb-a kucamo `run`, a ako želimo da se zakačimo gdb-om za program kucamo `attach ID_PROCESA`. U trenutku kada se zakačimo za program, on se pauzira. Naredbom `cont` nastavljamo program. Kada se desi bag, možemo ukucati `bt` da dobijemo stack trace poziva funkcija. Ako ukucamo `bt n` vratiće poslednjih n poziva. Komanda `frame k` prikazuje jedan konkretan poziv od izlistanih, onaj čiji je indeks k. Svi izlistani pozivi imaju svoj indeks. Komandom `list` možemo da izlistamo kod, mada je preporuka da se kod gleda u editoru. Komandom `p PROMENLJIVA` ispisujemo vrednost promenljive u tom trenutku. Komandom `q` pokrećemo ponovnu kompilaciju programa, na primer kada ispravimo grešku. Komanda `break KLASA::FUNKCIJA` postavlja breakpoint na određenu funkciju. Moguće je postavljanje i breakpointa pod nekim uslovom sa `break KLASA::FUNKCIJA if USLOV`. Komanda `rbreak` radi isto kao break, ali moguće je zadavanje regularnih izraza i postavljanje breakpoint-a na svaku funkciju koja zadovoljava taj izraz. Komandama `set logging redirect on` i `set logging on` se u poseban fajl ispisuju izlazi funkcija. Ako želimo da se nakon breakpoint-a izvrši neka komanda koristimo `commands`. Nakon toga, u svakom redu unosimo komande koje želimo da se izvrše i na kraju unosimo `end`. Komanda `delete` briše sve breakpoint-e. **Core fajl** sadrži sve bitne informacije o debagovanju. Komanda `tui reg GRUPA` prikazuje stanje date grupe registara. Komandom `set PROMENLJIVA = VREDNOST` možemo da postavljamo vrednost promenljivih. Komandom `call FUNKCIJA` pozivamo datu funkciju.

Komandom `python` dobijamo mogućnost pisanja Python skripti. Svaka linija se piše u posebnom redu, a na kraju se kuca `end`, nakon čega se skripta izvršava. Ako prosledimo argumente komandi možemo i da čuvamo podatke. Na primer, prvo kucamo `python x = 5`, a zatim `python print(x)` što će ispisati broj 5. Komandom `python import gdb` učitavamo gdb biblioteku preko koje možemo koristiti gdb komande u python skriptama. Na primer, `python gdb.execute('break hello.c:7')`. Drugi način da postavimo breakpoint je da kreiramo objekat: `python bp = gdb.Breakpoint('hello.c:7')`. Sada, ako želimo da ugasimo breakpoint to možemo uraditi na sledeći način: `python bp.enabled = False`. Za više informacija o objektima koristimo `python help(bp)`. Istu komandu možemo koristiti i za module: `python help(gdb)`. Možemo definisati i korisničke komande koristeći komandu `define IME_KOMANDE`, nakon čega navodimo komande koje želimo da se izvrše i na kraju navodimo `end`. Drugi način je da komandu napišemo u python fajlu, pa da ga pokrenemo iz gdba. Primer komande:

```
class BugReport(gdb.Command):
    """ Opis komande koji izlazi kada se korist help """
    def __init__(self): # Konstruktor
        ...

    def invoke(self, arg): # Komande koje se izvršavaju
        ...

BugReport()
```

Možemo definisati i hook-ove na sledeći način: `define hook-KOMANDA`. Nakon toga, pišemo komande koje želimo da se izvrše i završavamo sa `end`. Komande će se izvršavati pre nego što se pozove komanda "KOMANDA". Ako želimo da se izvrše nakon, koristimo `define posthook-KOMANDA`. Možemo povezati python funkcije i za gdb event-ove. Na primer:

```
def foo():
    print("Ovo se ispisuje pre prompta")

gdb.events.before_prompt.connect(foo)
```

Pre svake pojave gdb prompta pokrenuće se funkcija "foo", tj. biće ispisana odgovarajuća poruka. Postoje različiti gdb eventovi, kao npr. new_thread, breakpoint_created, memory_changed i tako dalje.