Git za matematike

Martin Vuk

Git je program, ki omogoča vodenje zgodovine različic datotek v neki mapi(direktoriju). V glavnem se uporablja za upravljanje z izvorno kodo pri razvoju računalniških programov. Mnogi med nami pa ga uporabljajo tudi pri pisanju besedil v *LaTeX*-u. Poleg tega, da Git hrani zgodovino sprememb, tudi omogoča da več ljudi hkrati sodeluje pri urejanju istih datotek. Ogledali si bomo, kako Git deluje. Opisali bomo, kako Git uporabi *zgoščevalne funkcije*, *Merklejeva drevesa* in *usmerjene aciklične grafe*, da shrani zgodovino različic in omogoči hkratno urejanje vsebine. Matematični model, ki ga Git uporablja je v resnici zelo preprost in njegovo razumevanje nas lahko reši marsikatere zagate, ki nastane med njegovo uporabo.

1. Kaj je Git?

Git je kot **časovni stroj** za datoteke. Uporabniku omogoča, da vidi **pretekle različice** datotek, spreminja datoteke, **brez skrbi, da bi kaj pokvaril** in datoteke **deli z drugimi**. Poleg časovnega stroja je Git **razpršeno skladišče datotek**. Omogoča, da datoteke **hkrati ureja več uporabnikov** na različnih računalnikih in kasneje spremembe **združi**.

Git hrani vsebino mape z datotekami in celotno zgodovino različic datotek iz preteklosti. Za vsako različico hrani Git zapis o avtorju, datumu in opis sprememb, ki so nastale v primerjavi s predhodno različico. Vse te imformacije dajejo podroben pregled nad zgodovino sprememb.

Opomba
Git in GitHub nista eno in isto. Ljudje pogosto mešajo Git in GitHub. Git je program, ki si ga lahko vsakdo namesti in poganja na svojem računalniku. Program Git je ustvaril Linus Torvalds, da bi lažje upravljal z izvorno kodo za jedro operacijskega sistema Linux. GitHub je javno spletišče, ki je namenjeno skladiščenju Git repozitorijev.

Opomba
Sisteme, ki omogočajo hranjenje preteklih različic datotek, imenujemo sistemi za nadzor različic (angl. version control system (VCS)) ali sistemi za upravljanje z izvorno kodo *(angl. Source Code Management (SCM)).*

Poleg nadzora različic Git omogoča hkratno spreminjanje datotek več uporabnikov na različnih računalnikih. Zato je Git distribuiran sistem za nadzor različic (angl. Distributed Version Control System (DVCS)).

V nadaljevanju bomo obravnavali naslednje teme:
- *Podatkovno skladišče:* Kako Git uporablja zgoščevalno funkcijo in Merklejeva drevesa za hranjenje posnetkov vsebine mape.
- *Zgodovina sprememb:* Kako zgodovino predstavimo z usmerjenim acikličnim grafom, v katerem so vozlišča različice, povezave pa povežejo različice z njihovimi neposrednimi predhodniki.
- *Reference*: Kako preproste reference (kazalci) na vsebino omogočajo bliskovito preklaplanje med različicami in preprečijo popolno zmešnjavo, ko več ljudi hkrati spreminja iste datoteke.

2. Podatkovno skladišče

Ko ustvarimo nov Git repozitorij, Git ustvari podmapo z imenom `.git` z vsemi podatki, ki jih Git potrebuje. Git v mapi `.git` hrani različne stvari:

- vsebino datotek, ki smo jih dodali v repozitorij,
- drevesno strukturo korenske mape, ki jo hranimo v repozitoriju,
- posnetke stanja v različnih trenutkih s podatki o avtoju, datumu in opisu sprememb,
- kazalce na posamezne posnetke stanja.

Git repozitorij je vsaka mapa, ki vsebuje podmapo `.git` z zgoraj navedenimi podatki.

2.1. Zgoščevalna funkcija

Git ne shranjuje datotek z običajnimi imeni, ampak za ime uporabi 160 bitno število (40 mestno število v 16-tiškem zapisu), ki ga izračuna iz vsebine datoteke. Git za izračun imena uporabi *zgoščevalno funkcijo*. Naj bo B množica vseh možnih podatkovnih nizov(besedil), n-bitna zgoščevalna funkcija je funkcija

$$H : B \to \{0, 1, ..., 2^n - 1\}, \tag{1}$$

ki vsakemu besedilu b priredi n-bitno vrednost $H(b)$. Vrednosti zgoščevalne funkcije $H(b)$ pravimo *zgostitev* vsebine b. Funkcija H, je izbrana tako, da sprememba enega samega bita v besedilu $b \in B$ spremeni vrednost $H(b)$. Poleg mora dobra zgoščevalna funkcija zagotoviti, da je porazdelitev vrednosti $H(b)$ čim bližje enakomerni porazdelitvi. To pomeni, da so vse vrednosti $H(b)$ približno enako verjetne.

Git uporablja zgoščevalno funkcijo *SHA1*. Funkcija SHA1 je posebna implementacija zgoščevalne funkcije, ki se je uporabljala v kriptografiji[1].

Ko datoteko z vsebino b zabeležimo v Git repozitorij, Git izračuna zgostitev vsebine $H(b)$ in jo shrani v datoteko z imenom $H(b)$ v `git/objects`[2]. Vsebina b je tako vedno dostopna pod imenom, ki je enako njeni zgostitvi $H(b)$. Tako dobimo vsebinsko naslovljivo shrambo objektov, ki je ena od bistvenih značilnosti Gita. Ta način shranjevanja omogoča, da lahko vedno preverimo, če ima shranjenjena vsebina isto zgostitev, kot je njeno ime. Lahko tudi shranimo več različic iste datoteke, saj ima vsaka različica drugačno zgostitev. Zgostitev služi tudi kot kontrola, če je prišlo do kvaritve podatkov, ki so shranjeni v Git repozitoriju.

2.1.1. Kolizije zgostitev in rojstnodnevni paradoks

Git hrani datoteke pod imeni, ki so enaka zgostitvi vsebine. Kaj pa če imata dve različni vsebini isto zgostitev. Funkcija H ni injektivna, saj je množica nizov, bistveno večja od množice zgostitev. To pomeni, da imata lahko dve različni datoteki enako zgostitev. Če bi prišlo do tega, bi Git shranil le eno datoteko, za drugo pa bi predpostavil da je že shranjena. Na srečo je verjetnost, da bi se kaj takega zgodilo izjemno majhna. Kljub temu, da zgoščevalna funkcija H ni injektivna, je verjetnost, da bi imela dva naključno izbrana podatkovna niza isto zgostitev (vrednost H) zelo majhna. Zato Git predpostavi, da je niz b enolično določen z njegovo zgostitvijo $H(b)$. Kako bi ocenili verjetnost, da pride do kolizije zgostitev?

Koliko datotek bi morali shraniti v Git, da bi z znatno verjetnostjo prišlo do kolizije? Vprašanje je povezano z rojstnodnevnim problemom. Kako velika naj bo skupina ljudi, da bo vsaj 50% verjetnost, da imata dve osebi na isti dan rojstni dan? Velikost skupine je presenetljivo majhna(23), zato rojstnodnevnei problem včasih poimenujemo paradoks. Vprašanje lahko matematično zastavimo tako. Naključno izberemo $n < d$

[1] Leta 2017 so raziskovalci iz CWI Amsterdam in Google Research našli prvi praktični primer dveh različnih pdf datotek, ki imata isto SHA1 zgostitev[1]. Opisan napad so poimenovali *SHAttered*. Git je zato z verzijo `v2.13.0` začel uporabljati verzijo SHA1, ki je odporna proti napadu *SHAttered*. Kljub temu razvijalci Gita načrtujejo, da bodo SHA1 postopoma nadomestili z

[2] V resnici Git shrani vsebino v datoteko z imenom $h_3h_4...h_{40}$ v mapi h_1h_2, kjer je $h_1h_2h_3...h_{40}$ zapis $H(b)$ v 16-tiškem sistemu. Datoteka, katere vsebina ima zgostitev $H(b)$ enako `8dd6d4bdaeff93016bd49474b54a911131759648` bo shranjena v `.git/objects/8d/d6d4bdaeff93016bd49474b54a911131759648`

števil iz množice $\{1, 2, \ldots, h\}$, tako da je vsaka izbira enakomerno porazdeljena. Kolikšna je verjetnost $p(n, h)$, da bosta vsaj dve števili enaki? Verjetnost $p(n, h)$ izračunamo elementarno z verjetnostjo nasprotnega dogodka:

$$1 - p(n, h) = \frac{h \cdot (h-1) \cdots (h-n+1)}{h^n} = \prod_{k=1}^{n-1} \left(1 - \frac{k}{h}\right). \tag{2}$$

Če izraz logaritmiramo, dobimo

$$\log(1 - p(n, h)) = \sum_{k=1}^{n-1} \log\left(1 - \frac{k}{h}\right) < -\sum_{k=1}^{n-1} \frac{k}{h} = -\frac{n(n-1)}{2h}. \tag{3}$$

Res! Logaritem je konveksna funkcija, zato so vrednosti manjše od vrednosti na tangenti $\log\left(1 - \frac{k}{h}\right) = \log(1 - x) < x = \frac{k}{h}$.

Od tod izpeljemo oceno za $p(n, h)$

$$p(n, h) > 1 - e^{-\frac{n(n-1)}{2h}} \approx 1 - e^{-\frac{n^2}{2h}}. \tag{4}$$

Za vrednosti $1 \ll n \ll h$ je $1 - e^{-\frac{n^2}{2h}}$ tudi dobra aproksimacija za $p(n, h)$.

Da bi odgovorili kako odporna je zgoščevalna funkcija na morebitne kolizije, moramo rešiti obratno nalogo: največ koliko števil $n(p, d)$ lahko izberemo, da bosta med izbranimi števili vsaj dve enaki z verjetnostjo manjšo od $p \in [0, 1]$. Natančen odgovor na to vprašanje ni tako preprost [2]. Lahko pa uporabimo oceno (Enačba 4) in čez palec ocenimo:

$$\begin{aligned} -n^2 &\approx \log(1 - p) \Rightarrow \\ n(p, h) &\approx \sqrt{2h \log\left(\frac{1}{1-p}\right)} \approx \sqrt{2h}. \end{aligned} \tag{5}$$

Funkcija $\sqrt{\log\left(\frac{1}{1-p}\right)}$ zelo počasi narašča, ko se p približuje 1, zato jo lahko zanemarimo. Če je zgoščevalna funkcija 160 bitna, kot je primer za SHA1, je $n \approx \sqrt{2^{160}} \approx 2^{80}$. Znatna verjetnost, da pride do kolizije zgostitev, bi se pojavila, ko bi shranili 2^{80} različnih datotek v Git. Raziskovalci, ki so razvili napad *SHAttered*, pa so potrebovali zgolj približno 2^{63} primerov, da so prišli do kolizije.

2.2. Datotečna drevesa

V vsebinsko naslovljivo shrambo objektov lahko shranimo vsebino datotek in njihovih prejšnjih različic. A kako ohranimo informacijo o imenu datotek in drevesni strukturi mape? Git za to ustvari nov tip objekta *drevo* (angl. *tree*), ki hrani preprost seznam imen datotek in naslovov na vsebino datotek v mapi. Naslov na vsebino datoteke b je seveda zgostitev vsebine $H(b)$. Seznam imen datotek in zgostitev je preprosta tekstovna datoteka, za katero lahko izračunamo zgostitev. Zgostitev datotečnega drevesa natanko določa tako imena datotek, kot tudi vsebino datotek, ki so vsebovane v mapi. Če se katerakoli datoteka ali ime datoteke v mapi spremeni, se bo spremnila tudi njena zgostitev in posledično zgostitev za drevo. Poleg posameznih datotek, lahko drevo vsebuje tudi poddrevesa. Tako lahko rekurzivno ustvarimo drevesno podatkovno strukturo, ki zajema mapo z datotekami in podmapami v poljubni globini.

```
100644 blob 33476f4951afc28d5ac2dc0d42d82f17ac817de2  bla.txt
100644 blob 2ce22b4dc77442103f095503f1205937c1b0fcfc  blabla.txt
040000 tree ae247f2a35aadade5863aec2475cf13020304b06  podmapa
```

Tabela 1: Vsebina mape v Gitu je preprost seznam datotek in podmap ter zgostitev njihove vsebine

Poglejmo si primer. Denimo, da imamo v naslednjo strukturo datotek in podmap

```
├── bla.txt (vsebina: bla)
├── blabla.txt (vsebina: blabla)
└── podmapa
    └── bla.txt (vsebina: bla)
```

Git bo shranil naslednje objekte v vsebinsko naslovljivo shrambo:

- vsebino datoteke `bla.txt`:

Zgoščena vrednost: `bcc1382241e267cf790ca6b3afe9fde6dcf1072f`
`bla`

- vsebino datoteke `blabal.txt`:

Zgoščena vrednost: `2ce22b4dc77442103f095503f1205937c1b0fcfc`
`blabla`

- seznam datotek v mapi `podmapa`:

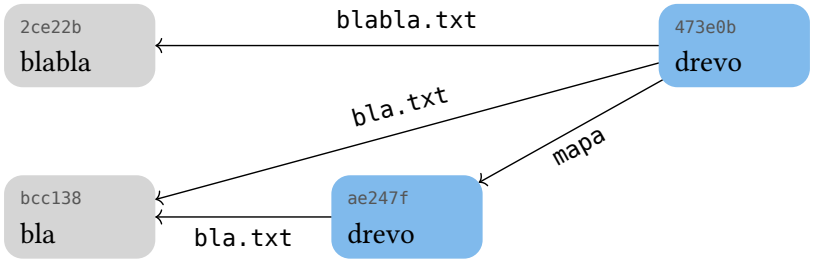
Zgoščena vrednost: `ae247f2a35aadade5863aec2475cf13020304b06`
`100644 blob bcc1382241e267cf790ca6b3afe9fde6dcf1072f  bla.txt`

- seznam datotek v korenski mapi:

Zgoščena vrednost: `473e0bbfc9de64fdca00e611e5666788ddf664ca`

`100644 blob 33476f4951afc28d5ac2dc0d42d82f17ac817de2  bla.txt`
`100644 blob 2ce22b4dc77442103f095503f1205937c1b0fcfc  blabla.txt`
`040000 tree ae247f2a35aadade5863aec2475cf13020304b06  podmapa`

Git z uporabo zgostitve kot kazalca na vsebino, vsebino mape postavi v podatkovno strukturo, ki jo matematično lahko opišemo z *usmerjenim acikličnim grafom*. Ko je vsebina datotek enaka(npr. `bla.txt` in `mapa/bla.txt`), Git shrani le eno kopijo, ki je dostopna v datoteki `.git/objects/bc/c1382241e267cf790ca6b3afe9fde6dcf1072f`. Zato datotečno drevo v Gitu ni nujno predstavljeno kot drevo, ampak kot usmerjen aciklični graf.



Slika 1: Primer datotečnega grafa povezanega z zgostitvami. Zaradi preglednosti smo izpisali le prvih 6 znakov zgostitve.

Posledično lahko vsebino celotne mape opišemo z eno samo zgostitvijo. Če spremenimo vsebino, ime ali lokacijo datoteke, bo sprememba vplivala na zgostitev spremenjene vsebine in sprememba bo splavala na površje do zgostitve za korensko mapo. Zgostitev služi tako kot identifikator vsebine, kot tudi kot kontrolna vsota, ki omogoča detekcijo sprememb.

Opomba
Podatkovna struktura objektov v Gitu je podobna Merklejevim drevesom. Razlika je v tem, da Gita hrani le eno kopijo datotek z identično vsebino, zato dobimo usmerjen aciklični graf in ne drevesa. Postopek je podoben veriženju blokov, ki se uporablja v kriptovalutah.

Opomba
*Dostop do objekta je mogoč, če poznamo **zgostitev** njegove vsebine. To pomeni, da je referenca na posamezen objekt v Gitu preprosto zgostitev(angl. hash) vsebine tega objekta. Po drugi strani je vsebina objekta določena z njegovo zgostitvijo. To pomeni, da lahko enostavno preverimo verodostojnost vsebine, ki je shranjena v Gitu. Git hrani skladišče objektov v mapi `.git/objects`.*

Podrobnosti o tem, kako Git hrani podatke, si lahko preberete v knjigi Pro Git [3, pog. 10.2].

3. Zgodovinski graf sprememb

V prejšnjem poglavju smo videli, kako Git hrani vsebino celotne mape in kako je mogoče do vsebine dostopati če poznamo zgostitvijo korenskega mape. Zgodovinsko drevo sprememb je preprosta razširitev omenjene podatkovne strukture.

3.1. Posnetki stanja

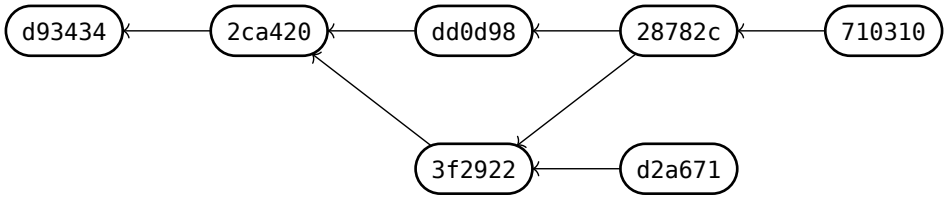
Osnovna enota v Gitu je **Vnos** (angl. **commit**). Vnos je posnetek stanja zabeleženih datotek v trenutku, ko je bil ustvarjen. Poleg vsebine datotek vsak vnos vsebuje še metapodatke o avtorju, datumu vnosa in opisom sprememb. Podobno kot objekt tipa *drevo*, je tudi vnos objekt v vsebinsko naslovljivi shrambi, ki ima določeno **zgostitev vnosa**. Zgostitev vnosa je natanko določena z vsebino shranjenih datotek in metapodatkov vnosa.

Zgoščena vrednost: `8dd6d4bdaeff93016bd49474b54a911131759648`

`tree 65c47feec7465e80492620a48206793e078702e0`
`parent 16f2994757f1213935b8edb9ae7fee3a8e9ec98d`
`author MV <mv@example.com> 1765235698 +0100`
`committer MV <mv@example.com> 1765235698 +0100`
 	
`Dodaj bla`

Tabela 2: Primer vnosa v Gitu. Vnos vsebuje zgostitev posnetka mape(`tree`), zgostitev starševskega vnosa (`parent`) in metapodatke. Tudi sam vnos je natančno določen z zgostitevjo.

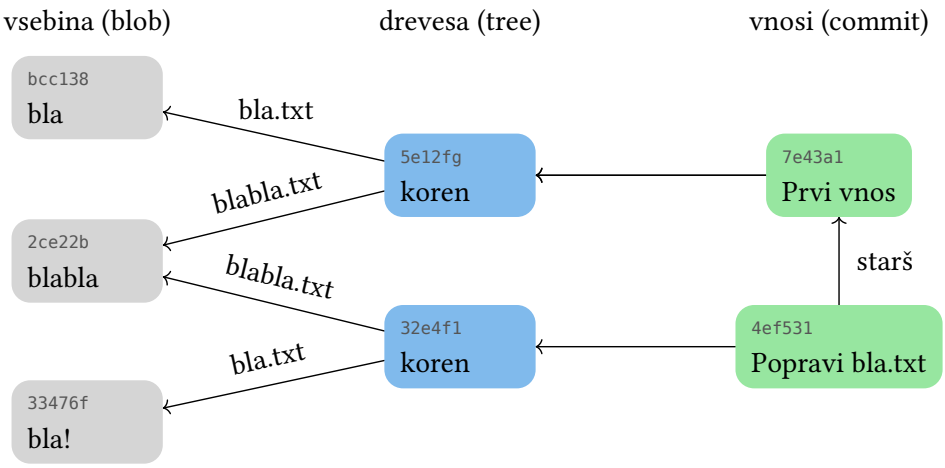
Vsak vnos je povezan s točno določenim posnetekom vsebine korenskega datotečnega drevesa, ki ga identificira zgostitev. Poleg tega so posamezni vnosi so povezani v **usmerjen acikličen graf (DAG)**, ki predstavlja zgodovino sprememb. Vsak **vnos** je **vozlišče** v grafu. Vsak vnos izhaja iz enega ali več starševskih vnosov. Izjema je prvi vnos. **Povezave** v grafu povezujejo vnose z njihovimi starši.



Slika 2: Vnosi v Gitu kot usmerjen graf. Vsak vnos(razen prvega) ima povezavo na vnose iz katerih izhaja.

Git hrani zgodovino sprememb v *vsebinsko naslovljivi shrambi objektov*, ki hrani tri vrste objektov:
- `blob`: vsebina datotek,
- `tree`: vsebina mape,
- `commit`: posnetek vsebine v določenem trenutku.

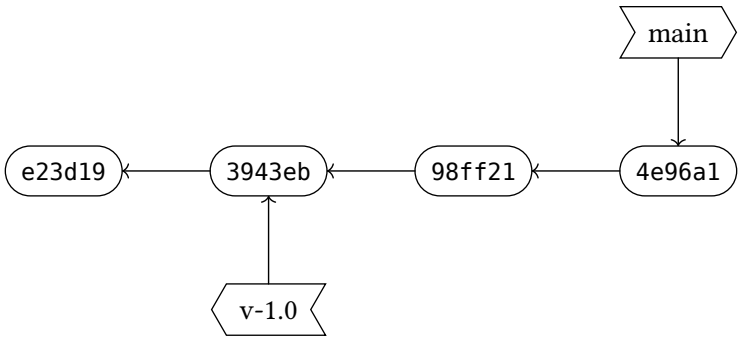
Objekti so poevazni v *usmerjen aciklični graf*. Podgraf na vnosih določa zgodovino sprememb. Naslovi objektov so *zgostitve* vsebine objekta, zato je zagotovljena verodostojnost shranjenih podatkov.



Slika 3: Vsebinsko naslovljiva shramba objektov v Gitu. Naslovi so zgostitve vsebine. Shramba vsebuje dva vnosa. V prvem vnosu smo dodali dve datoteki `bla.txt` in `blabla.txt`, v drugem vnosu pa smo spremenili le vsebino datotoeke `bla.txt`.

4. Kazalci: veje in značke

Poleg objektov kot so *vnosi*, *posnetki map* in *posnetki datotek* pozna git še reference. Reference so kazalci z določenim imenom na posamezen vnos.

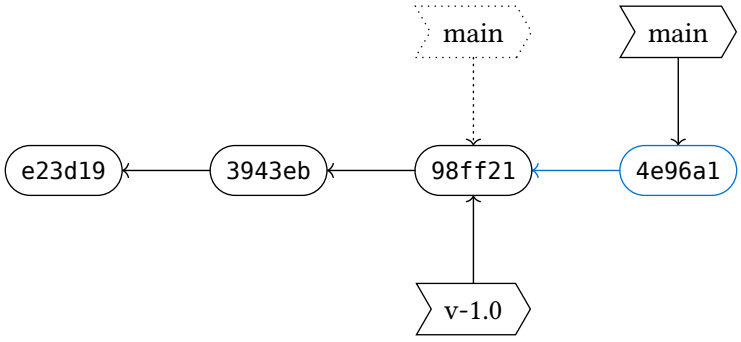


Slika 4: Veja (angl. branch) ali značka(angl. tag) je preprost kazalec na posamezen vnos(angl. commit).

Referenc git ne hrani v skladišču objektov, temveč posebej v mapi `.git/refs`. Reference vezane na posamezen repozitorij in se lahko razlikujejo med različnimi kloni določenega repozitorija.

Veja (angl. **branch**) je posebne vrste referenca, ki se premika, ko dodajamo nove vnose. Vsakič ko ustvarimo nov vnos, se trenutno aktivna veja premakne na novo ustvarjeni vnos.

Značka (angl. **tag**) je referenca, ki je statična in se ne premika več, ko jo enkrat ustvarimo.

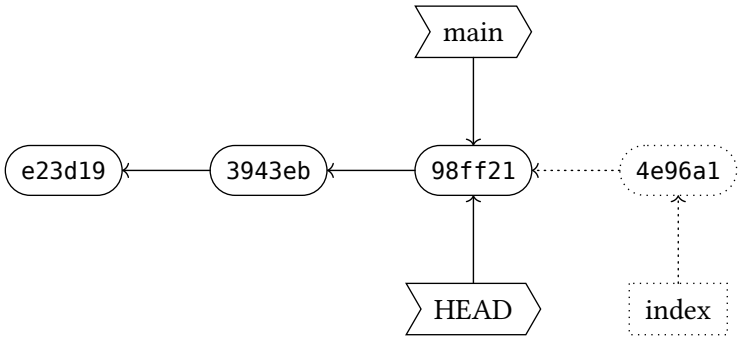


Slika 5: Ko ustvarimo nov vnos, se aktivna veja `main` premakne naprej, značka `v-1.0` pa ostane tam, kjer je bila.

Opomba
Veje in značke nimajo v Gitu nobenega posebnega pomena, razen tega, da so reference na vnose. Pomen posamenznih vej je stvar dogovora med uporabniki. Tako se pogosto uporablja različne veje za različne namene: `main` *ali* `master` *je navadno glavna veja razvoja, veje z imeni* `stable`*,* `production`*,* `development` *in podobno označujejo različne stopnje razvoja programske opreme, veje s predpono* `feature-` *označujejo razvoj novih funkcionalnosti.*

Vse te pomene damo vejam ljudje, ki sodelujemo v nekem Git repozitoriju. Za Git so vse veje in značke zgolj preprosti kazalci na določen vnos.

HEAD je posebna referenca, ki kaže na trenutno aktiven vnos. Vnos, na katerega kaže *HEAD* bo starševski vnos naslednjeg vnosa, ki ga bomo dodali.



Slika 6: **HEAD** je referenca na trenutno aktiven vnos. *Index* vsebuje spremembe, ki bodo zabeležene v naslednjem vnosu.

4.1. Povzetek

Povzemimo sedaj, kaj smo spoznali o podatkovnem modelu Gita. V vsebinsko naslovljivi shrambi hrani Git posnetke stanja celotne mape, ki ga vodimo v repozitoriju skupaj z metapodatki o spremembah. Najpomembnejša pojma sta:

- **Vnos** (angl. **commit**) je posnetek trenutnega stanja projekta, shranjen kot vozlišče v zgodovinskem grafu, ki vsebuje posnetek stanja datotek ter metapodatke (avtor, čas, sporočilo).
- **zgostitev vnosa** (angl. **commit hash**) je 40-mestna heksadecimalna vrednost, izračunana s SHA-1, ki enolično identificira vnos na podlagi vsebine posnetka in metapodatkov.

Izven shrambe objektov hrani Git še reference na posamezne vnose. Poznamo dve vrsti referenc:

- **Veja** (angl. **branch**) je premična reference, ki kaže na določen vnos v zgodovini in se samodejno premakne naprej, ko dodajamo nove vnose. Veje omogočajo vzporedne razvojne linije ki so med sabo neodvisne.
- **Oznaka** (angl. **tag**) je statična referenca, ki trajno kaže na določen vnos. Za razliko od veje se oznaka, nikoli ne premika samodejno, zato se uporablja predvsem za označevanje pomembnih točk v zgodovini, kot so izdaje ali stabilne verzije.
- **HEAD** je posebna oznaka, ki kaže na trenutno aktiven vnos v delovni kopiji.

Omenimo še dva pojma, ki jih uporabljamo pri delu z Gitom:

- **Delovna kopija** (angl. **workout copy**) je mapa v kateri urejamo datoteke, ki jih nato vnesemo v Git. V delovni kopiji imajo na začetku datoteke isto vsebino kot je vsebina trenutno aktivnega vnosa (`HEAD`). Spremembe, ki jih naaredimo na delovni kopiji lahko zabeležimo v nov vnos.
- **Oddaljen repozitorij** (angl. **remote**) je povezava(url) na isti repozitorij na drugem računalniku(ponavadi strežniku), s katerim lahko izmenjujemo vsebino.

Opomba
Gitov podatkovni model omogoča, da je večina operacij v Gitu obrnljivih. To pomeni, da lahko repozitorij povrnemo v prejšnje stanje. Običajne operacije le dodajajo nove vnose in starih ne brišejo. Prav tako se v zgodovinsko drevo le dodaja nove povezave in starih se ne briše. Zato daje delo z Gitom uporabniku samozavest, da brez strahu spreminja vsebino, saj se lahko vedno vrne v času nazaj, kot da bi imel časovni stroj.

Nekatere operacije pa tudi brišejo vnose (npr. `git rebase`). Takim operacijam rečemo, da spreminjajo zgodovino. Uporabniki morajo biti pri uporabi operacij, ki spreminjajo zgodovino posebej pazljivi, da česa trajno ne zamočijo.

5. Git ukazi kot operacije na grafu

Ko smo opremljeni z razumevanjem podatkovnega modela Gita, razložimo kaj pomenijo posamezne operacije, ki jih Git omogoča. Ukazov ne bom prevajal, ampak jih bom navedel kot jih pozna program `git`.

5.1. Checkout

Ukaz

```
git checkout referenca
```

spremeni datoteke v delovni kopiji tako, da se ujemajo z vnosom, na katerega kaže referenca. Poleg tega prestavi oznako `HEAD` na isti vnos. Če je referenca veja, jo nastavi, kot aktivno vejo. Če je referenca oznaka ali zgostitev vnosa, priedmo v stanje brez aktivne veje (angl. *deteached HEAD*).

5.2. Commit

Ukaz

```
git commit -m "Sporočilo za vnos"
```

ustvari nov vnos, ki kaže na stanje v čakalnici (angl. staging area ali index). V zgodovinskem grafu ustvari novo vozlišče, ki je povezano s prejšnjim vnosom. Poleg tega prestavi aktivno vejo in oznako `HEAD` na novo ustvarjeni vnos.

5.3. Add

Ukaz

```
git add bla.txt
```

doda vsebino spremenjene datoteke `bla.txt` v čakalnico. Ukaz ne spreminja zgodovinskega grafa, pač pa doda novo vsebino in datotečna drevesa, ki vsebujejo spremembe v shrambo objektov. Vsebina čakalnice bo zabeležena v naslednjem vnosu.

5.4. Pull

Ukaz

```
git pull
```

pobere vsebino(objekte in reference) iz oddaljenega repozitorija in uskladi lokalno vejo z oddaljeno. Shrambi objektov se preprosto doda nove objekte, ki so v oddaljeni veji. Če je lokalna veja prednik oddaljene, se lokalna veja enostavno prestavi, da kaže na isti vnos, kot oddaljena veja. V nasprotnem primeru, mora uporabnik posredovati in razrešiti morebitne konflikte.

5.5. Push

Ukaz

```
git push
```

potisne novo vsebino na oddaljeni repozitorij. Push deluje obratno kot `pull`. Ukaz je uspešno izveden le, če je oddaljena veja prednica lokalne veje in ni konflikotov.

5.6. Fetch

Ukaz

```
git fetch
```

pobere novo vsebino (vnose, veje in oznake) iz oddaljenega repozitorija. Pri tem ne more priti do konfliktov, ker git preprosto doda nove objekte v shrambo in obstoječih objektov nikakor ne spreminja. Oddaljenim vejam in oznakam preprosto doda predpono z imenom oddaljenega repozitorija.

6. Reset

Ukaz

```
git reset referenca
```

spremeni kam kaže trenutno izbrana veja. Ukaz ne spremeni zgodovinskega drevesa, ampak le to, na kateri vnos kaže trenutno izbrana veja.

6.1. Merge

Ukaz

```
git merge referenca
```

ustvari nov vnos, ki združi dve ločeni veji v eno (trenutno izbrano in referenco). Nov vnos ima dva starša: vnos na katerega kaže trenutna veja in vnos, na katerega kaže referenca. Če pride do konfliktov, jih mora uporabnik sam razrešiti, preden se ustvari nov vnos.

6.2. Rebase

Ukaz

```
git rebase referenca
```

prestavi vnose v trenutno izbrani veji tako, da so potomci vnosa, na katerega kaže referenca. Med ukazi, ki smo jih spoznali, je ta ukaz edini, ki lahko povzroči izgubo podatkov. Običajno ukazi le dodajajo nove vnose in prestavljajo reference. Zato je večina ukazov v Gitu varna, v smislu, da jih lahko kasneje prekličemo in pridemo nazaj na prejšnje stanje. Ukaz `rebase` pa spremeni zgodovino in ga ne moremo preklicati, saj trenutne vnose nadomesti z novimi in stare vnose pobriše[3].

6.3. Zaključek

Spoznali smo, kako deluje Git in s katere matematičnime pojme uporablja za model. Opis dela z Gitom presega namen tega dokumenta, zato vas raje usmerim na uradno dokumentacijo:

https://git-scm.com/cheat-sheet

[3]Obstaja enostaven način, da tudi `rebase` lahko prekličemo. Na zadnji vnos, ki ga želimo prestaviti preprosto postavimo novo referenco(vejo ali oznako). To povzroči, da se stari vnosi ne pobrišejo tudi, ko se izvede ukaz `rebase`.

Pri pisanju tega članka sem sevada uporabljal Git. V javno dostopnem repozitoriju [4] si lahko ogledate celotno zgodovino nastajanja tega članka.

Pri pripravi dokumenta sem uporabil Gemini 3. Vse odgovore sem preveril in uredil po svoje.

Literatura

[1] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, in Y. Markov, „The First Collision for Full SHA-1“, v *Advances in Cryptology – CRYPTO 2017*, J. Katz in H. Shacham, Ur., Cham: Springer International Publishing, 2017, str. 570–596.

[2] D. Brink, „A (probably) exact solution to the Birthday Problem“, *The Ramanujan Journal*, let. 28, št. 2, str. 223–238, jun. 2012, doi: 10.1007/s11139-011-9343-9.

[3] S. Chacon in B. Straub, „10.2 Git Internals - Git Objects“, *Pro Git*. Pridobljeno: 3. januar 2026. [Na spletu]. Dostopno na: https://git-scm.com/book/en/v2/Git-Internals-Git-Objects

[4] M. Vuk, „git-intro“. Pridobljeno: 3. januar 2026. [Na spletu]. Dostopno na: https://git.fri.uni-lj.si/martin.vuk/git-intro