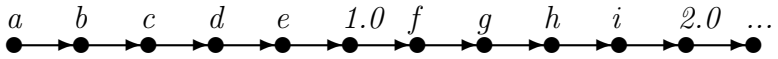
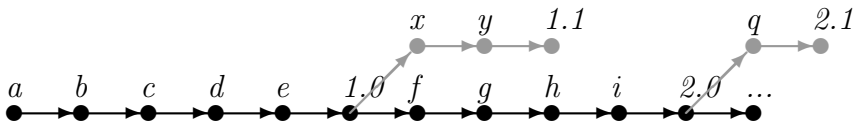
U V O D U G I T

Tomo Krajina

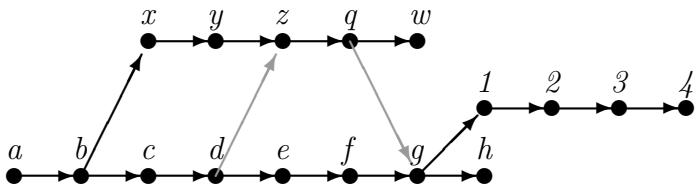
Od...



...preko...



...pa do...



...a i dalje.

Commit `ce0132889d552f51dd6c35e18f4ee9eced3d22de`

- Izdano pod "Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0)" licencom. Detalji na:
  `http://creativecommons.org/licenses/by-sa/3.0/deed.en_US`
- Ovu knjigu **možete kopirati**, fotokopirati, dijeliti prijateljima i kolegama i koristiti za učenje,
- Ovu knjigu **smijete mijenjati**, ali mora biti **jasno naznačeno koje dijelove knjige je tko napisao**,
- Izmijenjenu knjigu i dalje možete kopirati i dijeliti, ali **izmijenjena verzija mora zadržati istu ili kompatibilnu licencu**.
- Izmijenjena knjiga **mora sadržavati link na originalnu lokaciju njenog LaTeX koda**:
  `https://github.com/tkrajina/uvod-u-git`

Sadržaj

Uvod **9**

O težini, teškoćama i problemima 10

Pretpostavke 10

Našla/našao sam grešku 11

Naredbe i operacijski sustavi 12

Verzioniranje koda i osnovni pojmovi **13**

Što je verzioniranje koda? 13

Linearno verzioniranje koda 14

Grafovi, grananje i spajanje grana 15

Mit o timu i sustavima za verzioniranje 18

Instalacija, konfiguracija i prvi projekt **19**

Instalacija 19

Prvi git repozitorij 19

Git naredbe 20

Osnovna konfiguracija 21

.gitignore 22

Spremanje izmjena **24**

Status 25
Indeks 27
Spremanje u indeks 28
Micanje iz indeksa 29
O indeksu i stanju datoteka 30
Prvi commit 32
Indeks i *commit* grafički 32
Datoteke koje ne želimo u repozitoriju 33
Povijest projekta 34
Ispravljanje zadnjeg *commit*a 35
Git gui 35
Clean 37

Grananje **39**
Popis grana projekta 39
Nova grana 40
Prebacivanje s grane na granu 41
Prebacivanje na granu i tekuće izmjene 43
Brisanje grane 43
Preuzimanje datoteke iz druge grane 44

Preuzimanje izmjena iz jedne grane u drugu **45**
Git *merge* 46
Što *merge* radi kada... 47
Što se dogodi kad... 48
Konflikti 50

Merge, *branch* i povijest projekta 53
Fast forward 54
Rebase 56
Rebase ili ne *rebase*? 60
Rebase i rad sa standardnim sustavima za verzioniranje 60
Cherry-pick 61
Merge bez *commita* 63
Squash merge 64

Tagovi **65**

Ispod haube **68**

Kako biste vi... 68
SHA1 70
Grane 71
Reference 72
HEAD 73
.git direktorij 74
.git/config 74
.git/objects 75
.git/refs 76
HEAD 77
.git/hooks 78

Povijest **79**

Diff 79
Log 80

Whatchanged . . . 81
Pretraživanje povijesti . . . 82
Gitk . . . 83
Blame . . . 86
Digresija o premještanju datoteka . . . 86
Preuzimanje datoteke iz povijesti . . . 88
"Teleportiranje" u povijest . . . 88
Reset . . . 89
Revert . . . 91
Izrazi s referencama . . . 92
Reflog . . . 94

Udaljeni repozitoriji **96**
Naziv i adresa repozitorija . . . 96
Kloniranje repozitorija . . . 97
Struktura kloniranog repozitorija . . . 98
Djelomično kloniranje povijesti repozitorija . . . 99
Digresija o grafovima, repozitorijima i granama . . . 100
Fetch . . . 101
Pull . . . 106
Push . . . 106
Push tagova . . . 110
Rebase origin/master . . . 111
Prisilan *push* . . . 111
Rad s granama . . . 112
Brisanje udaljene grane . . . 115

Udaljeni repozitoriji 115
Dodavanje i brisanje udaljenih repozitorija 116
Fetch, *merge*, *pull* i *push* s udaljenim repozitorijima 119
Pull request 120
Bare repozitorij 121

"Higijena" repozitorija **124**
Grane 124
Git gc 125
Povijest i brisanje grana 126
Digresija o brisanju grana 128
Squash merge i brisanje grana 129

Bisect **131**
Automatski *bisect* 134
Digresija o atomarnim *commit*ovima 135

Prikaz grana u git alatima **136**
Prikaz lokalnih grana 136
Prikaz grana udaljenog repozitorija 139

Česta pitanja **141**
Jesmo li *push*ali svoje izmjene na udaljeni repozitorij? 141
*Commit*ali smo u krivu granu 142
*Commit*ali smo u granu X, ali te commitove želimo prebaciti u novu granu . 143
Imamo ne*commitane* izmjene i git nam ne da prebacivanje na drugu granu . 144
Zadnjih *n commit*ova treba "stisnuti" u jedan *commit* 144

*Push*ali smo u remote repozitorij izmjenu koju nismo htjeli 145

*Merge*ali smo, a nismo htjeli 145

Ne znamo gdje smo *commit*ali 146

Manje korištene naredbe **147**

Filter-branch 147

Shortlog 147

Format-patch 148

Am 148

Fsck 148

Instaweb 148

Name-rev 148

Stash 149

Submodule 149

Rev-list 149

Dodaci **150**

Git hosting 150

Vlastiti server 151

Git *shell* 151

Certifikati 152

Git *plugin* 153

Git i Mercurial 153

Terminologija **157**

Popis korištenih termina 158

Zahvale **160**

Uvod

Git je alat koji je razvio Linus Torvalds da bi mu olakšao vođenje jednog velikog i kompleksnog projekta – Linux kernela. U početku to **nije** bio program s današnjom namjenom; Linus je zamislio da git bude osnova **drugim sustavima za razvijanje koda**. Drugi alati su trebali razvijati svoje sučelje na osnovu gita. Tako je, barem, bilo zamišljeno. Međutim, kao s mnogim drugim projektima otvorenog koda, ljudi su ga počeli koristiti takvog kakav jest, a on je organski rastao sa zahtjevima korisnika.

Rezultat je program koji ima drukčiju terminologiju u odnosu na druge slične, ali milijuni programera diljem svijeta su ga prihvatili. Nastale su brojne platforme za *hosting* projekata, kao što je Github[1], a već postojeći su morali dodati git jednostavno zato što su to njihovi korisnici tražili (Google Code[2], Bitbucket[3], Sourceforge[4], pa čak i Microsoftov CodePlex[5]).

Nekoliko je razloga zašto je to tako:

- Postojeći sustavi za verzioniranje su zahtijevali da se točno zna tko sudjeluje u projektu (tj. tko je *comitter*). To je demotiviralo ljude koji bi možda pokušali pomoći projektima kad bi imali priliku. S distribuiranim sustavima bilo tko može "*fork*ati" repozitorij i raditi na njemu. Ukoliko misli da je napravio nešto korisno – vlasniku originalnog repozitorija bi predložio da preuzme njegove izmjene. Broj ljudi koji se mogu okušati u radu na nekom projektu je tako puno veći, a vlasnik i dalje zadržava pravo odlučivanja čije će izmjene uzeti, a čije neće.
- git je **brz**,
- vrlo je lako i brzo granati, isprobavati izmjene koje su radili drugi ljudi i preuzeti

[1] http://github.com
[2] http://code.google.com
[3] http://bitbucket.com
[4] http://sourceforge.net
[5] http://www.codeplex.com

ih u svoj kod,

- Linux kernel se razvijao na gitu, tako da je u svijetu otvorenog koga (*open source*) git stekao nekakvu auru važnosti.

U nastavku ove knjige pozabavit ćemo se osnovnim pojmovima verzioniranja koda općenito i načinom kako je sve to implementirano u gitu.

O težini, teškoćama i problemima

Ova knjiga nije zamišljena kao općeniti priručnik u kojem ćete tražiti rješenje svaki put kad negdje zapnete. Osnovna ideja mi je bila da za svaku "radnju" s gitom opišem problem, ilustriram ga grafikonom, malo razradim teoriju, potkrijepim primjerima i onda opišem nekoliko osnovnih git naredbi. Nakon što pročitate knjigu, trebali biste biti sposobni git koristiti u svakodnevnom radu.

Tekst koji slijedi zahtijeva koncentraciju i vježbanje, posebno ako niste nikad radili s nekim od distribuiranih sustava za verzioniranje. Trebate naučiti terminologiju, naredbe i osnovne koncepte, ali – isplati se.

Zapnete li, a odgovora ne nađete ovdje, pravac Stackoverflow[6], Google, forumi, blogovi i, naravno, `git help`.

Postoji i jednostavan način kako da postignete da čitanje ove knjige postane trivijalno jednostavno – čitanje napreskokce. Git, naime, **možete** koristiti analogno klasičnim sustavima za verzioniranje. U tom slučaju vam ne trebaju detalji o tome kako se grana ili što je *rebase*. U principu – svi smo git tako i koristili u početku.

Želite li takav "ekspresni" uvod u git – dovoljno je da pročitate poglavlja o verzioniranju, *commit*anju i prvi dio poglavlja o udaljenim repozitorijima. Pojmovi koje biste trebali savladati su *commit*, *push*, *fetch*, konflikt i `origin` repozitorij.

Izgubite li se u šumi novih pojmova – na kraju knjige imate pregled svih git-specifičnih termina s kratkim objašnjenjem.

Pretpostavke

Da biste uredno "probavili" ovaj knjižuljak, pretpostavljam da:

[6] http://stackoverflow.com

- znate programirati u nekom programskom jeziku ili barem imate dobru predodžbu o tome kako teče proces nastajanja i razvoja aplikacija,
- ne bojite se komandne linije,
- poznajete osnove rada s unixoidnim operacijskim sustavima.

Poznavanje rada s klasičnim sustavima za verzioniranje koda (CVS, SVN, TFS, ...) nije nužno.

Nekoliko riječi o zadnje dvije stavke. Iako git nije nužno ograničen na Unix/Linux operacijske sustave, njegovo komandnolinijsko sučelje je tamo nastalo i drži se istih principa. Problem je što je mnoge složenije stvari teško uopće implementirati u nekom grafičkom sučelju. Moj prijedlog je da git naučite koristiti u komandnoj liniji, a tek onda krenete s nekim grafičkim alatom – tek tako ćete ga zaista savladati.

Našla/našao sam grešku

Svjestan sam toga da ova knjižica vrvi greškama. Ja nisam profesionalan pisac, a ova knjiga nije prošla kroz ruke profesionalnog lektora.

Grešaka ima i pomalo ih ispravljam. Ako želite pomoći – unaprijed sam zahvalan! Evo nekoliko načina kako to možete učniti:

- Pošaljite email na `tkrajina@gmail.com`,
- *Twitt*nite mi na `@puzz`,
- *Fork*ajte i pošaljite *pull request* s ispravkom.

Ukoliko odaberete bilo koju varijantu osim zadnje (*fork*) – dovoljan je kratak opis s greškom (stranica, rečenica, redak) i šifra koja se nalazi na dnu naslovnice[7].

Repozitorij s izvornim LaTeX kodom knjige možete naći na adresi `http://github.com/tkrajina/uvod-u-git` a na istoj adresi se nalazi i najnovija verzija PDF-a.

[7] Na primjer ono što piše *Commit* b5af8ec79a7384a5a291d15d050fc932eb474e79. Ovaj nerazumljivi dugi string mi značajno olakšava traženje verzije za koju prijavljujete grešku.

Naredbe i operacijski sustavi

Sve naredbe koje nisu specifične za git, kao na primjer "stvaranje novog direktorija", "ispis datoteka u direktoriju", i sl. će biti prema POSIX standardu[8]. Dakle, u primjerima ćemo koristiti naredbe koje se koriste na UNIX, OS X i Linux operacijskim sustavima. Za korisnike Microsoft Windowsa to ne bi trebao biti problem jer se radi o relativno malom broju njih kao što su `mkdir` umjesto `md`, `ls` umjesto `dir`, i slično.

[8] http://en.wikipedia.org/wiki/POSIX

Verzioniranje koda i osnovni pojmovi

Što je verzioniranje koda?

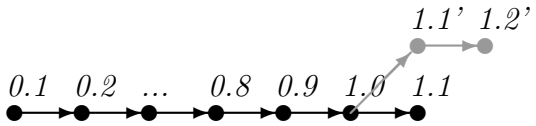
S problemom verzioniranja koda sreli ste se kad ste prvi put napisali program koji rješava neki konkretan problem. Bilo da je to neka jednostavna web aplikacija, CMS[9], komandnolinijski pomoćni programčić ili kompleksni ERP[10].

Svaka aplikacija koja ima **stvarnog** korisnika kojemu rješava neki **stvarni** problem ima i **korisničke zahtjeve**. Taj korisnik možemo biti mi sami, može biti neko hipotetsko tržište (kojemu planiramo prodati rješenje) ili može biti naručitelj. Korisničke zahtjeve ne možemo nikad točno predvidjeti u trenutku kad krenemo pisati program. Možemo satima, danima i mjesecima sjediti s budućim korisnicima i planirati što će sve naša aplikacija imati, ali kad korisnik sjedne pred prvu verziju aplikacije, čak i ako je pisana točno prema njegovim specifikacijama, on će naći nešto što ne valja. Radi li se o nekoj maloj izmjeni, možda ćemo je napraviti na licu mjesta. Možda ćemo trebati otići kući, potrošiti nekoliko dana i napraviti **novu verziju**.

Desit će se, na primjer, da korisniku damo na testiranje verziju `1.0`. On će istestirati i, naravno, naći nekoliko sitnih stvari koje treba ispraviti. Otići ćemo kući, ispraviti ih, napraviti verziju `1.1` s kojom će klijent biti zadovoljan. Nekoliko dana kasnije, s malo više iskustva u radu s aplikacijom, on zaključuje kako sad ima **bolju** ideju kako je trebalo ispraviti verziju **1.0**. Sad, dakle, treba "baciti u smeće" posao koji smo radili za `1.1`, vratiti se na `1.0` i od nje napraviti npr. `1.1b`.

Grafički bi to izgledalo ovako nekako:

[9] Content Management System
[10] Enterprise Resource Planning

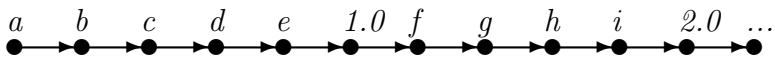


U trenutku kad je korisnik odlučio da mu trenutna verzija ne odgovara trebamo se vratiti korak unazad (u povijest projekta) i započeti novu verziju, odnosno novu **granu projekta** te nastaviti projekt s tom izmjenom.

I to je samo jedan od mnogih mogućih scenarija kakvi se događaju u programerskom životu.

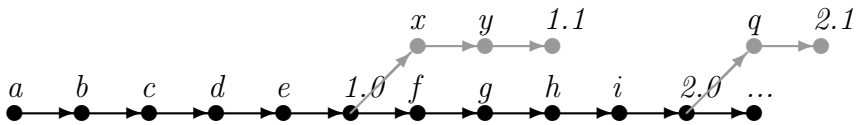
Linearno verzioniranje koda

Linearni pristup verzioniranju koda se najbolje može opisati sljedećom ilustracijom:



To je idealna situacija u kojoj točno unaprijed znamo kako aplikacija treba izgledati. Započnemo projekt s početnim stanjem *a*, pa napravimo izmjene *b*, *c*, ... sve dok ne zaključimo da smo spremni izdati prvu verziju za javnost i proglasimo to verzijom `1.0`.

Postoje mnoge varijacije ovog linearnog modela, jedna česta je:

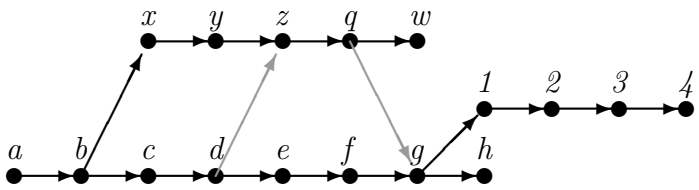


Ona je česta u situacijama kad nemamo kontrolu nad time koja je točno verzija programa instalirana kod klijenta. S web aplikacijama to nije problem jer vi jednostavno možete aplikaciju prebaciti na server i odmah svi klijenti koriste novu verziju. Međutim, ako je vaš program klijentima "spržen" na CD i takav poslan klijentu može se dogoditi da jedan ima instaliranu verziju `1.0`, a drugi `2.0`.

I sad, što ako klijent koji je zadovoljan sa starijom verzijom programa otkrije **bug** i zbog nekog razloga ne želi prijeći na novu verziju? U tom slučaju morate imati neki mehanizam kako da se privremeno vratite na staru verziju, ispravite problem, izdate "novu verziju stare verzije", pošaljete je klijentu i nakon toga se vratite na prijašnje stanje te tamo nastavite gdje ste stali.

Grafovi, grananje i spajanje grana

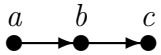
Prije nego nastavimo s gitom, nekoliko riječi o grafovima. U ovoj ćete knjižici vidjeti puno grafova kao što su u primjerima s linearnim verzioniranjem koda. Zato ćemo se na trenutak zadržati na jednom takvom grafu:



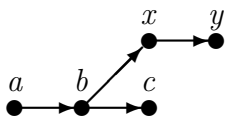
Svaka točka grafa je stanje projekta. Projekt s gornjim grafom započeo je s nekim početnim stanjem *a*. Programer je napravio nekoliko izmjena i **snimio** novo stanje *b*, zatim *c*, ...i tako sve do *h*, *w* i 4.

Primijetite da je ovakav graf stanje povijesti projekta, ali iz njega ne možemo zaključiti kojim su redom čvorovi nastajali. Neke stvari možemo zaključiti: vidi se, na primjer, da je *d* nastao nakon *c*, *e* nakon *d* ili *z* nakon *y*. Ne znamo je li prije nastao *c* ili *x*. Ili, čvor *1* je sigurno nastao nakon *g*, no iz grafa se ne vidi je li nastao prije *x* ili nakon *x*.

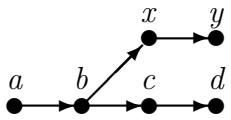
Evo jedan način kako je navedeni graf mogao nastati:



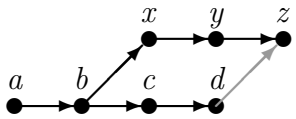
Programer je započeo aplikaciju, snimio stanje *a*, *b* i *c* i tada se sjetio da ima neki problem kojeg može riješiti na dva načina, vratio se na *b* i napravio novu granu. Tamo je napravio izmjene *x* i *y*:



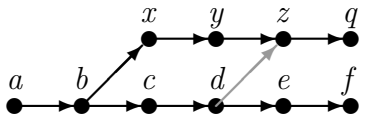
Zatim se sjetio izmjene koju je mogao napraviti u *originalnoj* verziji, vratio se tamo i dodao d:



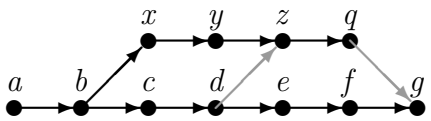
Nakon toga se vratio na svoj prvotni eksperiment i odlučio da bi bilo dobro tamo imati izmjene koje je napravio u c i d. Tada je *preuzeo* te izmjene u svoju granu:



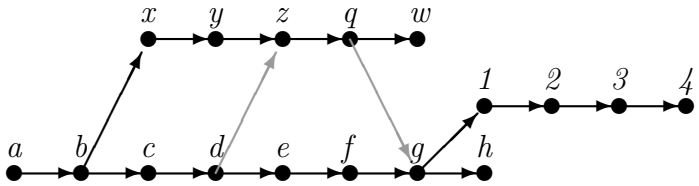
Na eksperimentalnoj je grani napravio još jednu izmjenu q. Tada je odlučio privremeno napustiti taj eksperiment i posvetiti se izmjenama koje mora napraviti u glavnoj grani. Vratio se na originalnu granu i tamo napredovao s e i f.



Sjetio se da bi mu sve izmjene iz eksperimentalne grane odgovarale u originalnoj, *preuzeo* ih u početnu granu:



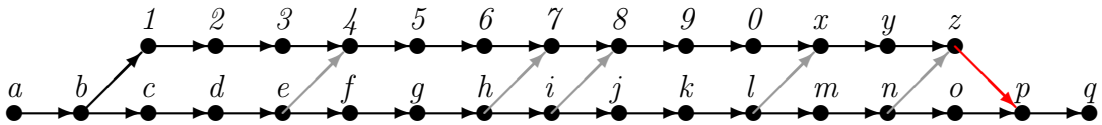
. . . zatim je nastavio i napravio još jednu eksperimentalnu granu (*1*, *2*, *3*, . . .). Na onoj je prvoj eksperimentalnoj grani dodao još i w i tako dalje. . .



Na svim će grafovima glavna grana biti ona najdonja. Uočite, na primjer, da izmjena w nije nikad završila u glavnoj grani.

Jedna od velikih prednosti gita je lakoća stvaranja novih grana i preuzimanja izmjena iz jedne u drugu granu. Tako je programerima jednostavno u nekom trenutku razmišljati i postupiti na sljedeći način: *"Ovaj problem bih mogao riješiti na dva različita načina. Pokušat ću i jedan i drugi, i onda vidjeti koji mi bolje ide."*. Za svaku će verziju napraviti posebnu granu i napredovati prema osjećaju.

Druga velika prednost čestog grananja u programima je kad se dodaje neka nova funkcionalnost koja zahtijeva puno izmjena, a ne želimo te izmjene odmah stavljati u glavnu granu programa:



Trebamo pripaziti da redovito izmjene iz glavne grane programa preuzimamo u sporednu tako da razlike u kodu ne budu prevelike. Te su izmjene na grafu označene sivim strelicama.

Kad završimo novu funkcionalnost, u glavnu granu treba preuzeti sve izmjene iz sporedne (crvena strelica). Na taj ćemo način često imati ne samo dvije grane (glavnu i sporednu) nego nekoliko njih. Imati ćemo posebne grane za različite nove funkcionalnosti, posebne grane za eksperimente, posebne grane u kojima ćemo isprobavati izmjene koje su napravili drugi programeri, posebne grane za ispravljanje pojedinih *bug*ova, ...

Osnovna ideja ove knjižice **nije** učiti vas kako je najbolje organizirati povijest projekta, odnosno kako granati te kad i kako preuzimati izmjene iz pojedinih grana. Osnovna ideja je naučiti vas **kako** da to napraviti s gitom. Nakon savladanog **kako** prirodno dolazi i intuicija o tome **kako ispravno**.

Mit o timu i sustavima za verzioniranje

Prije nego nastavimo, htio bih srušiti jedan mit. Taj mit glasi ovako nekako: "*Sustavi za verzioniranje koda potrebni su kad na nekom projektu radi više ljudi*".

Vjerujte mi, ovo nije istina.

Posebno to nije istina za git i druge distribuirane sustave koji su namijenjeni čestom grananju. Kad o projektu počnete razmišljati kao o jednom usmjerenom grafu i posebne stvari radite u posebnim granama to značajno olakšava samo razmišljanje o razvoju. Ako imate jedan direktorij s cijelim projektom i u kodu imate paralelno izmjene od tri različite stvari koje radite istovremeno, onda imate problem.

Nemojte pročitati knjigu i reći "*Nisam baš uvjeren*". Probajte git[11] na nekoliko tjedana.

[11] Ako vam se git i ne svidi, probajte barem mercurial.

Instalacija, konfiguracija i prvi projekt

Instalacija

Instalacija gita je relativno jednostavna. Ako ste na nekom od linuxoidnih operacijskih sustava sigurno postoji paket za instalaciju. Za sve ostale, postoje jednostavne instalacije, a svi su linkovi dostupni na službenim web stranicama[12].

Važno je napomenuti da su to samo **osnovni paketi**. Oni će biti dovoljni za primjere koji slijede, no za mnoge specifične scenarije postoje dodaci s kojima se git naredbe "obogaćuju" novima.

Prvi git repozitorij

Ako ste naviknuti na TFS, subversion ili CVS onda si vjerojatno zamišljate da je za ovaj korak potrebno neko računalo na kojem je instaliran poseban servis (*daemon*) i kojemu je potrebno dati do znanja da želite imati novi repozitorij na njemu. Vjerojatno mislite i to da je sljedeći korak preuzeti taj projekt s tog udaljenog računala/servisa. Neki sustavi taj korak nazivaju *checkout*, neki *import*, a u gitu je to *clone* iliti kloniranje projekta.

S gitom je jednostavnije. **Apsolutno svaki direktorij može postati git repozitorij.** Ne mora *uopće* postojati udaljeni server i neki centralni repozitorij kojeg koriste (i) ostali koji rade na projektu. Ako vam je to neobično, onda se spremite, jer stvar je još čudnija: ako već postoji udaljeni repozitorij s kojeg preuzimate izmjene od drugih programera on ne mora biti jedan jedini. **Mogu postojati deseci takvih udaljenih**

[12] http://git-scm.com/download

repozitorija, sami ćete odlučiti na koje ćete "slati" svoje izmjene i s kojih preuzimati izmjene. I vlasnici tih udaljenih repozitorija imaju istu slobodu kao i vi, mogu sami odlučiti čije će izmjene preuzimati kod sebe i kome slati svoje izmjene.

Pomisliti ćete da je rezultat anarhija u kojoj se ne zna tko pije, tko plače, a tko plaća. Nije tako. Stvari, uglavnom, funkcioniraju bez većih problema.

Idemo sad na prvi i najjednostavniji korak: stvoriti ćemo novi direktorij `moj-prvi-projekt` i stvoriti novi repozitorij u njemu:

```
$ mkdir moj-prvi-projekt
$ cd moj-prvi-projekt
$ git init
Initialized empty Git repository in /home/user/moj-prvi-projekt/.git/
$
```

I to je to.

Ako idete pogledati kakva se to čarolija desila s tim `git init`, otkriti ćete da je stvoren direktorij `.git`. U principu cijela povijest, sve grane, čvorovi i komentari, apsolutno sve vezano uz repozitorij čuva se u tom direktoriju. Zatreba li nam ikad sigurnosna kopija cijelog repozitorija, sve što treba napraviti je da sve lokalne promjene spremimo (*commit*amo) u git i spremimo negdje arhivu (`.zip`, `.bz2`, ...) s tim `.git` direktorijem.

Git naredbe

U prethodnom smo primjeru u našem direktoriju inicijalizirali git repozitorij naredbom `git init`. Općenito, git naredbe uvijek imaju sljedeći format:

```
git <naredba> <opcija1> <opcija2> ...
```

Izuzetak je pomoćni grafički program kojim se može pregledavati povijest projekta, a koji dolazi u instalaciji s gitom, `gitk`.

Za svaku git naredbu možemo dobiti *help* s:

```
git help <naredba>
```

Na primjer, `git help init` ili `git help config`.

Osnovna konfiguracija

Nekoliko postavki je poželjno konfigurirati da bismo nastavili normalan rad. Sva git konfiguracija se postavlja pomoću naredbe `git config`. Postavke mogu biti **lokalne** (odnosno vezane uz jedan jedini projekt) ili **globalne** (vezane uz korisnika na računalu).

Globalne postavke se postavljaju s:

```
git config --global <naziv> <vrijednost>
```

...i one se spremaju u datoteku `.gitconfig` u vašem *home* direktoriju.

Lokalne postavke se spremaju u `.git` direktorij u direktoriju koji sadrži vaš repozitorij, a tada je format naredbe `git config`:

```
git config <naziv> <vrijednost>
```

Za normalan rad na nekom projektu, drugi korisnici trebaju znati tko je točno radio koje izmjene na kodu (*commit*ove). Zato trebamo postaviti ime i email adresu koja će u povijesti projekta biti "zapamćena" uz svaku našu spremljenu izmjenu:

```
$ git config --global user.name "Ana Anić"
$ git config --global user.email "ana.anic@privatna.domena.com"
```

Imamo li neki repozitorij koji je vezan za posao i možda se ne želimo identificirati sa svojom privatnom domenom, tada *u tom direktoriju* trebamo postaviti drukčije postavke:

```
$ git config user.name "Ana Anić"
$ git config user.email "ana.anic@poslodavac.hr"
```

Na taj će se način *email* adresa `ana.anic@poslodavac.hr` spominjati samo u povijesti tog projekta.

Postoje mnoge druge konfiguracijske postavke, no ja vam preporučam da za početak postavite barem dvije `color.ui` i `merge.tool`.

S `color.ui` možete postaviti da ispis git naredbi bude obojan:

```
$ git config --global color.ui auto
```

`merge.tool` određuje koji će se program koristiti u slučaju **konflikta** (o tome više kasnije). Ja koristim `gvimdiff`:

```
$ git config --global merge.tool gvimdiff
```

.gitignore

Prije ili kasnije dogodit će se situacija da u direktoriju s repozitorijem imamo datoteke koje ne želimo spremati u povijest projekta. To su, na primjer, konfiguracijske datoteke za različite editore ili datoteke koje nastaju kompajliranjem (`.class` za javu, `.pyc` za python, `.o` za C, i sl.). U tom slučaju trebamo nekako gitu dati do znanja da takve datoteke ne treba nikad snimati. Otvorite novu datoteku naziva `.gitignore` u glavnom direktoriju projekta (ne nekom od poddirektorija) i jednostavno unesite sve ono što ne treba biti dio povijesti projekta.

Ako ne želimo `.class`, `.swo`, `.swp` datoteke i sve ono što se nalazi u direktoriju `target/` , naša će `.gitignore` datoteka izgledati ovako:

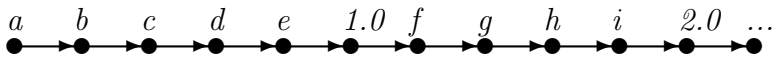
```
# Vim privremene datoteke:
*.swp
*.swo
# Java kompajlirane klase:
*.class
# Output direktorij s rezultatima kompajliranja i builda:
target/*
```

Linije koje započinju znakom # su komentari i git će se ponašati kao da ne postoje.

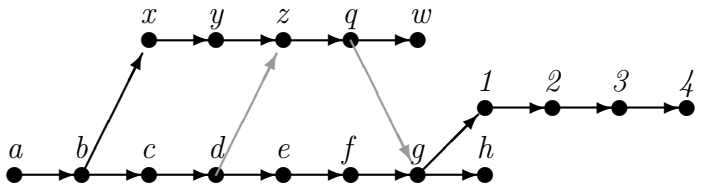
Sad smo spremni početi raditi s našim projektom...

Spremanje izmjena

Vratimo se na trenutak na naša dva primjera, linearni model verzioniranja koda:



...i primjer s granama:



Svaki čvor grafa predstavlja stanje projekta u nekom trenutku, a sam graf je redoslijed kako je naš projekt "evoluirao". Na primjer, kad smo prvi put inicirali projekt s `git init`, dodali smo nekoliko datoteka i **spremili ih**. U tom je trenutku nastao čvor *a*. Nakon toga smo možda izmijenili neke od tih datoteka, možda neke obrisali, neke nove dodali te opet spremili novo stanje i dobili stanje *b*.

To što smo radili između svaka dva stanja (tj. čvora) naša je stvar i ne tiče se gita[13]. Trenutak kad se odlučimo spremiti novo stanje projekta u naš repozitorij, to je gitu jedino važno i to se zove *commit*.

Važno je ovdje napomenuti da u gitu, za razliku od subversiona, CVS-a ili TFS-a **nikad ne commitamo u udaljeni repozitorij**. Svoje lokalne promjene *commit*amo,

[13]Neki sustavi za verzioniranje, kao na primjer TFS, zahtijevaju stalnu vezu na internet i serveru dojavljuju svaki put kad krenete editirati neku datoteku. Git nije takav.

odnosno spremamo, u **lokalni** repozitorij na našem računalu. Interakcija s udaljenim repozitorijem bit će tema poglavlja o udaljenim repozitorijima.

Status

Da bismo provjerili imamo li uopće nešto za spremiti, koristi se naredba `git status`. Na primjer, kad na projektu koji nema lokalnih izmjena za spremanje utipkamo `git status`, dobit ćemo:

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

Recimo da smo napravili tri izmjene na projektu: Izmijenili smo datoteke `README.md` i `setup.py` te obrisali `TODO.txt`: Sad će rezultat od `git status` izgledati ovako:

```
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add/rm <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working
directory)
#
#        modified:   README.md
#        deleted:    TODO.txt
#        modified:   setup.py
#
```

Najbitniji su podatak linije u kojima piše `modified:` i `deleted:` jer to su datoteke koje smo *mijenjali*, ali ne još *commit*ali.

Želimo li pogledati **koje su točne razlike** u tim datotekama u odnosu na stanje kakvo je snimljeno u repozitoriju, odnosno u **zadnjoj verziji** repozitorija, to možemo dobiti s `git diff`. Primjer ispisa te naredbe je:

```
diff --git a/README.md b/README.md
index 80b2f4b..faaac11 100644
--- a/README.md
+++ b/README.md
@@ -32,8 +32,7 @@ Usage
             for point in route:
                     print 'Point at (0,1) -> 2'.format(
point.latitude, point.longitude, point.elevation )

-    # There are more utility methods and functions...
-
+    # There are many more utility methods and functions:
     # You can manipulate/add/remove tracks, segments, points,
waypoints and routes and
     # get the GPX XML file from the resulting object:

diff --git a/TODO.txt b/TODO.txt
deleted file mode 100644
index d528b19..0000000
--- a/TODO.txt
+++ /dev/null
@@ -1 +0,0 @@
-- remove extreemes (options in smooth() remove_elevation_extreemes,
remove_latlon_extreemes, default False for both)
diff --git a/setup.py b/setup.py
index c9bbb18..01a08a9 100755
--- a/setup.py
+++ b/setup.py
@@ -17,7 +17,7 @@
 import distutils.core as mod_distutilscore

 mod_distutilscore.setup( name = 'gpxpy',
-        version = '0.6.0',
+        version = '0.6.1',
         description = 'GPX file parser and GPS track manipulation
library',
         license = 'Apache License, Version 2.0',
         author = 'Tomo Krajina',
```

Linije koje počinju s `diff` govore o kojim datotekama se radi. Nakon njih slijedi nekoliko linija s općenitim podacima i zatim kod **oko** dijela datoteke koji je izmijenjen i onda ono najvažnije: linije obojene u crveno i one obojene u plavo.

Linije koje započinju s "-" (crvene) su linije koje su obrisane, a one koje počinju s "+" (u plavom) su one koje su dodane. Primijetite da git ne zna da smo neku liniju izmijenili. Ako jesmo, on se ponaša kao da smo staru obrisali, a novu dodali.

Rezultat `diff` naredbe su samo linije koda koje smo izmijenili i nekoliko linija **oko njih**. Ako želimo malo veću "okolinu" oko naših izmjena, možemo je izmijeniti opcijom `-U<broj_linija>`. Na primjer, ako želimo 10 linija oko izmjenjenih dijelova koda, to ćemo dobiti s:

```
git diff -U10
```

Indeks

Iako često govorimo o tome kako ćemo "*commit*ati datoteku" ili "staviti datoteku u indeks" ili..., treba imati na umu da git ne čuva "datoteke" (kao nekakav apstraktni pojam) nego stanja, odnosno **verzije datoteka**. Dakle, za jednu te istu datoteku git čuva njena različita stanja kako se mijenjala kroz povijest. Mi datoteke u našem projektu mijenjamo, a sami odlučujemo u kojem su trenutku one takve da bismo ih snimili.

U gitu postoji poseban "međuprostor" u koji se "stavljaju" datoteke koje ćemo spremiti (*commit*ati). Dakle, sad imamo tri različita "mjesta" u kojima se čuvaju datoteke odnosno konkretna stanja pojedinih datoteka:

Git repozitorij čuva različita stanja iste datoteke (**povijest** datoteke).

Radna verzija repozitorija je stanje datoteka u našem direktoriju. Ono može biti isto ili različito u odnosu na stanje datoteka u repozitoriju.

Poseban "međuprostor" za *commit* gdje privremeno spremamo trenutno stanje datoteka prije nego što ih *commit*amo.

Ovo zadnje stanje, odnosno taj "međuprostor za *commit*" zove se *index* iliti indeks.

U literaturi ćete često naći i naziv *staging area* ili *cache*[14]. Naredba `git status` je namijenjena pregledavanju statusa indeksa i radne verzije projekta. Na primjer, u trenutku pisanja ovog poglavlja, `git status` je:

```
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working
directory)
#
#        modified:   uvod.tex
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Ovaj ispis govori kako je jedna datoteka izmijenjena, ali nije još *commit*ana niti stavljena u indeks.

Ako je stanje na radnoj verziji našeg projekta potpuno isto kao i u zadnjoj verziji git repozitorija, onda će nas `git status` obavijestiti da nemamo ništa za *commit*ati. U suprotnom, reći će koje su datoteke izmijenjene, a na nama je da sad u indeks stavimo (samo) one datoteke koje ćemo u sljedećem koraku *commit*ati.

Trenutno ćemo stanje direktorija s projektom u nastavku referencirati kao **radna verzija projekta**. Radna verzija projekta može, ali i ne mora, biti jednaka stanju projekta u repozitoriju ili indeksu.

Spremanje u indeks

Recimo da smo promijenili datoteku `uvod.tex`[15]. Nju možemo staviti u indeks s:

```
git add uvod.tex
```

...i sad je status:

[14] Nažalost, git ovdje nije konzistentan pa i u svojoj dokumentaciji ponekad koristi *stage*, a ponekad *cache*.

[15] To je upravo datoteka u kojoj se nalazi poglavlje koje trenutno čitate.

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   uvod.tex
#
```

Primijetite dio u kojem piše: `Changes to be committed`. E to je popis datoteka koje smo stavili u indeks.

Nakon što smo datoteku spremili u indeks, spremni smo za *commit* ili možemo nastaviti dodavati druge datoteke s `git add` sve dok se ne odlučimo za snimanje.

Čak i ako smo datoteku *obrisali*, moramo je dodati u indeks naredbom `git add`. Ako vas to zbunjuje, podsjetimo se da **u indeks ne stavljamo u stvari datoteku nego neko njeno (izmijenjeno) stanje**. Kad smo datoteku obrisali, u indeks treba spremiti novo stanje te datoteke, "izbrisano stanje".

`git add` ne moramo nužno koristiti s jednom datotekom. Ako spremamo cijeli direktorij datoteka, možemo ga dodati s:

```
git add naziv_direktorija/*
```

Ili ako želimo dodati apsolutno sve što se nalazi u našoj radnoj verziji:

```
git add .
```

Micanje iz indeksa

Recimo da smo datoteku stavili u indeks i kasnije se predomislili, lako je iz indeksa maknemo naredbom:

```
git reset HEAD -- <datoteka1> <datoteka2> ...
```

Događat će nam se situacija da smo promijenili neku datoteku, no kasnije zaključimo da ta izmjena nije bila potrebna. I sad je ne želimo spremiti nego vratiti u prethodno stanje, odnosno točno onakvo stanje kakvo je u zadnjoj verziji repozitorija. To se može ovako:

```
git checkout HEAD -- <datoteka1> <datoteka2> ...
```

Više detalja o `git checkout` i zašto ta gornja naredba radi to što radi bit će kasnije.

O indeksu i stanju datoteka

Ima još jedan detalj koji bi vas mogao zbuniti. Uzmimo situaciju da smo samo jednu datoteku izmijenili i spremili u indeks:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   uvod.tex
#
```

Izmijenimo li sad tu datoteku još jednom, novo će stanje biti ovakvo:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   uvod.tex
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working
directory)
#
#       modified:   uvod.tex
#
```

Dotična datoteka je sad u radnoj verziji označena kao izmijenjena, ali nije još stavljena u indeks. Istovremeno ona je i u indeksu! Iz stvarnog svijeta smo naviknuti da jedna stvar ne može biti na dva mjesta, međutim iz dosadašnjeg razmatranja znamo da gitu nije toliko bitna datoteka (kao apstraktan pojam) nego **konkretne verzije (ili stanja) datoteke**. I u tom smislu nije ništa neobično da imamo jedno stanje datoteke u indeksu, a drugo stanje datoteke u radnoj verziji našeg projekta.

Ako sad želimo osvježiti indeks sa zadnjom verzijom datoteke (onom koja je, *de facto* spremljena u direktoriju), onda ćemo jednostavno:

```
git add <datoteka>
```

Ukratko, indeks je prostor u kojeg spremamo grupu datoteka (**stanja** datoteka!). Takav skup datoteka treba predstavljati neku logičku cjelinu koju ćemo spremiti u repozitorij. To spremanje je jedan *commit*, a tim postupkom smo grafu našeg repozitorija dodali još jedan čvor.

Prije *commit*a datoteke možemo stavljati u indeks ili izbacivati iz indeksa. To činimo sve dok nismo sigurni da indeks predstavlja točno one datoteke koje želimo u našoj sljedećoj izmjeni (*commit*u).

Razlika između tog novog čvora i njegovog prethodnika upravo su datoteke koje smo

imali u indeksu u trenutku kad smo *commit* izvršili.

Prvi commit

Izmjene možemo spremiti s:

```
git commit -m "Nova verzija"
```

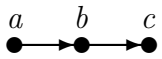
U stringu nakon `-m` **moramo** unijeti komentar uz svaku promjenu koju spremamo u repozitorij. Git ne dopušta spremanje izmjena bez komentara[16].

Sad je status projekta opet:

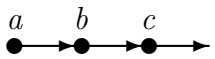
```
$ git status
# On branch master
nothing to commit (working directory clean)
```

Indeks i *commit* grafički

Cijela ova priča s indeksom i *commit*anjem bi se grafički mogla prikazati ovako: U nekom je trenutku stanje projekta ovakvo:



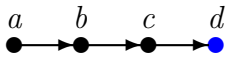
To znači da je stanje projekta u direktoriju potpuno isto kao i stanje projekta u zadnjem čvoru našeg git grafa. Recimo da smo nakon toga u direktoriju izmijenili nekoliko datoteka:



[16] U stvari dopušta ako dodate `--allow-empty-message`, ali bolje da to ne radite.

To znači da smo napravili izmjene, no one još nisu dio repozitorija. Zapamtite, samo čvorovi u grafu su ono što git čuva u repozitoriju. Strelice su samo "postupak" ili "proces" koji je doveo od jednog čvora/*commit*a do drugog. Zato u prethodnom grafu imamo strelicu koja **ne vodi do čvora**.

Nakon toga odlučujemo koje ćemo datoteke spremiti u indeks s `git add`. Kad smo to učinili, s `git commit` *commit*amo ih u repozitorij i tek je sad stanje projekta:



Dakle, nakon *commit*a smo dobili novi čvor d.

Datoteke koje ne želimo u repozitoriju

Situacija koja se često događa je sljedeća: Greškom smo u repozitorij spremili datoteku koja tamo ne treba biti. Međutim, tu datoteku ne želimo obrisati s našeg diska nego samo ne želimo njenu povijest imati u repozitoriju.

To se dešava, na primjer, kad nam editor ili IDE spremi konfiguracijske datoteke koje su njemu važne, ali nisu bitne za projekt. Eclipse tako zna snimiti `.project`, a Vim sprema radne datoteke s ekstenzijama `.swp` ili `.swo`. Ako smo takvu datoteku jednom dodali u repozitorij, a naknadno smo zaključili da ju više ne želimo, onda je prvo trebamo dodati u `.gitignore`. Nakon toga git zna da **ubuduće** neće biti potrebno snimati izmjene na njoj.

No ona je i dalje u repozitoriju! Ne želimo je obrisati s diska, ali ne želimo je ni u povijesti projekta (od sad pa na dalje). Neka je to, na primjer, `test.pyc`. Postupak je:

```
git rm --cached test.pyc
```

To će nam u indeks dodati stanje kao da je datoteka obrisana iako je ostavljena netaknuta na disku. Drugim riječima `git rm --cached` sprema "obrisano stanje" datoteke u indeks. Sad tu izmjenu treba *commit*ati da bi git znao da od ovog trenutka nadalje datoteku može obrisati iz svoje povijesti.

Budući da smo datoteku prethodno dodali u `.gitignore`, git nam ubuduće neće

nuditi da ju *commit*amo. Odnosno, što god radili s tom datotekom, `git status` će se ponašati kao da ne postoji.

Povijest projekta

Sve prethodne *commit*ove možemo pogledati s `git log`:

```
$ git log
commit bf4fc495fc926050fb10260a6a9ae66c96aaf908
Author: Tomo Krajina <tkrajina@gmail.com>
Date:   Sat Feb 25 14:23:57 2012 +0100

    Version 0.6.0

commit 82256c42f05419963e5eb13e25061ec9022bf525
Author: Tomo Krajina <tkrajina@gmail.com>
Date:   Sat Feb 25 14:15:13 2012 +0100

    Named tuples test

commit a53b22ed7225d7a16d0521509a2f6faf4b1c4c2e
Author: Tomo Krajina <tkrajina@gmail.com>
Date:   Sun Feb 19 21:20:11 2012 +0100

    Named tuples for nearest locations

commit e6d5f910c47ed58035644e57b852dc0fc0354bbf
Author: Tomo Krajina <tkrajina@gmail.com>
Date:   Wed Feb 15 21:36:33 2012 +0100

    Named tuples as return values

...
```

Više riječi o povijesti repozitorija bit će u posebnom poglavlju. Za sada je važno znati da u gitu svaki *commit* ima jedinstveni string koji ga identificira. Taj string

ima 40 alfanumeričkih znakova, a primjere takvih stringova možemo vidjeti s naredbom `git log`. Na primjer,
`bf4fc495fc926050fb10260a6a9ae66c96aaf908` je jedan takav.

Ispravljanje zadnjeg *commit*a

Meni se, prije gita, događalo da *commit*am neku izmjenu u repozitorij, a nakon toga shvatim da sam trebao još nešto promijeniti. I činilo mi se logično da ta izmjena bude dio prethodnog *commit*a, ali *commit* sam već napravio. Nisam ga mogao naknadno promijeniti, tako da je na kraju jedna logička izmjena bila snimljena u dva *commit*a.

S gitom se to može riješiti elegantnije: novu izmjenu možete **dodati** u već postojeći *commit*.

Prvo učinimo tu izmjenu u radnoj verziji projekta. Recimo da je to bilo na datoteci `README.md`. Dodamo tu datoteku u indeks s `git add README.md` kao da se spremamo napraviti još jedan *commit*. Umjesto `git commit`, sad je naredba:

```
git commit --amend -m "Nova verzija, promijenjen README.md"
```

Ovaj `--amend` gitu govori da izmijeni zadnji *commit* u povijesti tako da sadrži **i izmjene koje je već imao i izmjene koje smo upravo dodali**. Možemo provjeriti s `git log` što se dogodilo i vidjet ćemo da zadnji *commit* sad ima novi komentar.

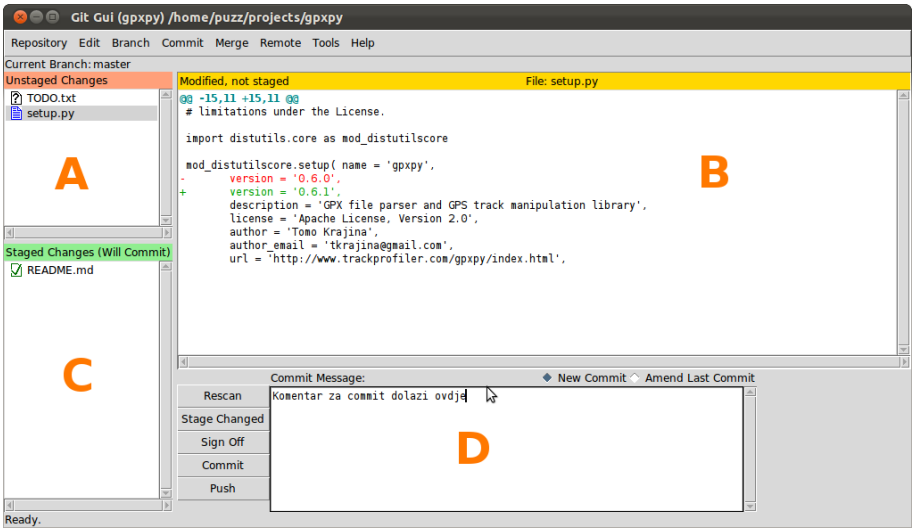
`git commit --amend` nam omogućava da u zadnji *commit* dodamo neku datoteku ili čak i *maknemo* datoteku koju smo prethodno *commit*ali. Treba samo pripaziti da se taj commit nalazi samo na našem lokalnom repozitoriju, a ne i na nekom od udaljenih. Više o tome malo kasnije.

Git gui

Kad spremamo *commit* s puno datoteka, onda može postati naporno non-stop tipkati `git add`. Zbog toga postoji poseban grafički program kojemu je glavna namjena upravo to. U komandnoj liniji:

```
git gui
```

Otvoriti će se sljedeće:



Program se sastoji od četiri pravokutna polja:

- Polje za datoteke koje su izmijenjene, ali nisu još u indeksu (označeno s **A**).
- Polje za prikaz izmjena u pojedinim datotekama (**B**).
- Polje za datoteke koje su izmijenjene i stavljene su u indeks (**C**).
- Polje za *commit* (**D**).

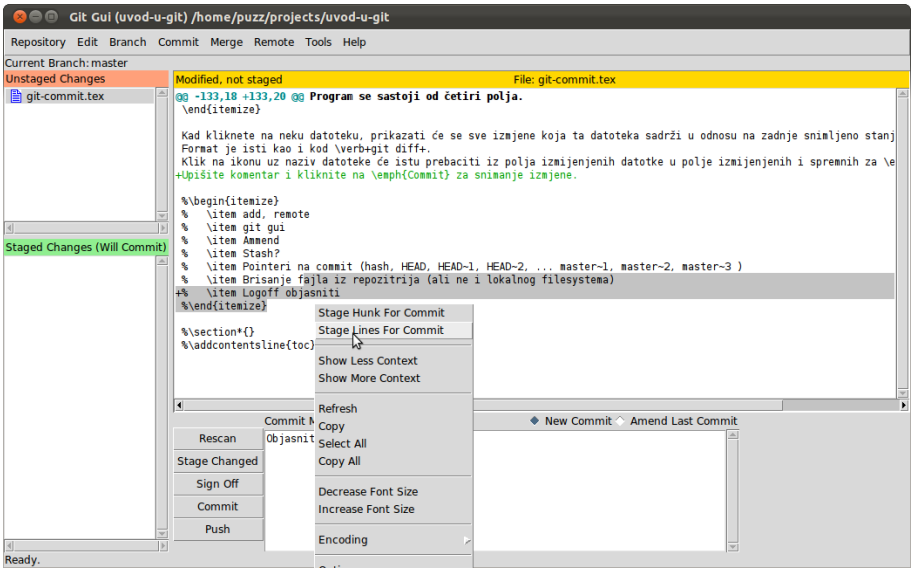
Klik na neku datoteku prikazat će sve izmjene koja ta datoteka sadrži u odnosu na zadnje snimljeno stanje u repozitoriju. Format je isti kao i kod `git diff`. Klik na ikonu uz naziv datoteke istu će prebaciti iz polja izmijenjenih datotka u polje s indeksom i suprotno. Nakon što odaberemo datoteke za koje želimo da budu dio našeg *commit*a[17], trebamo unijeti komentar i kliknuti na "Commit" za snimanje izmjene.

Ovdje, kao i u radu s komandnom linijom, ne moramo sve izmijenjene datoteke snimiti u jednom *commit*u. Možemo prebaciti u indeks samo dio datoteka, upisati

[17] To jest, nakon što te datoteke spremimo u *index* iliti nakon što ih prebacimo iz gornjeg lijevog polja u donje lijevo polje.

komentar, snimiti i nakon toga dodati sljedećih nekoliko datoteka, opisati novi komentar i snimiti sljedeću izmjenu. Drugim riječima, izmjene možemo snimiti u nekoliko posebnih *commit*ova, tako da svaki od njih čini zasebnu logičku cjelinu.

S `git gui` imamo još jednu korisnu opciju, možemo u indeks dodati **ne cijelu datoteku, nego samo nekoliko izmijenjenih linija** datoteke. Za tu datoteku, u polju s izmijenjenim linijama odaberimo samo linije koje želimo spremiti, desni klik i odaberite "`Stage lines to commit`":



Ako smo na nekoj datoteci napravili izmjenu koju *ne* želimo snimiti, takvu datoteku možemo resetirati, odnosno vratiti u početno stanje. Jednostavno odaberemo tu datoteku i u meniju kliknemo na `Commit` → `Revert changes`.

Osim ovoga, `git gui` ima puno korisnih mogućnosti koje nisu predmet ovog poglavlja. Preporučam vam da nađete vremena i proučite sve menije i kratice s tipkovnicom u radu jer to će vam značajno ubrzati daljnji rad.

Clean

Naredba `git clean` služi da bi iz radnog direktorija obrisali sve one datoteke koje nisu dio trenutne verzije repozitorija. To je korisno kad želimo obrisati privremene datoteke koje su rezultat kompajliranja ili privremene direktorije. Nismo li sigurni što će točno

izbrisati s `git clean -n` ćemo dobiti samo spisak. Ako dodamo `-x` naredba će obrisati i sve datoteke koje su popisane u `.gitignore`.

Na primjer, LaTeXje metajezik za pisanje dokumenata u kojem je pisana i ova knjiga. Dokument se kompajlira u PDF, ali pri tome generira privremene datoteke s ekstenzijama `.aux` i `.log`. Te datoteke nam nisu potrebne u repozitoriju i možemo ih dodati u `.gitignore` ili jednostavno počistiti s `git clean`.

Prvo pogledamo koje točno datoteke će `git clean` "očistiti":

```
$ git clean -n
Would remove dodaci.aux
Would remove git-branch.aux
Would remove git-commit.aux
Would remove git-gc.aux
Would remove git-init.aux
Would remove git-log.aux
Would not remove graphs/
Would remove ispod-haube.aux
Would remove verzioniranje-koda.aux
```

Ako nam je to u redu, onda s:

```
git clean -f
```

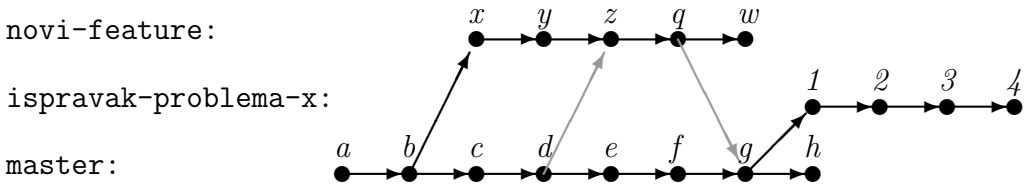
... brišemo te datoteke.

Možemo obrisati i samo određene privremene datoteke ili samo jedan direktorij s:

```
git clean -f -- <direktorij>
```

Grananje

Početi ćemo s otprije poznatim grafom:



Ovaj put s jednom izmjenom, svaka "grana" ima svoj naziv: `novi-feature`, `ispravak-problema-x` i `master`. U uvodnom je poglavlju opisan jedan od mogućih scenarija koji je mogao dovesti do ovog grafa. Ono što je ovdje važno još jednom spomenuti je sljedeće: svaki je čvor grafa stanje projekta u nekom trenutku njegove povijesti. Svaka strelica iz jednog u drugi čvor izmjena je koju je programer napravio i snimio u nadi da će dovesti do željenog ponašanja aplikacije.

Popis grana projekta

Jedna od velikih prednosti gita je što omogućuje jednostavan i brz rad s višestrukim granama. Želimo li vidjeti koje točno grane našeg projekta trenutno postoje naredba je `git branch`. U većini će slučajeva rezultat te naredbe biti:

```
$ git branch
* master
```

To znači da naš projekt trenutno ima samo jednu granu. **Svaki git repozitorij u**

početku ima jednu jedinu granu i ona se uvijek zove `master`.

Ako smo naslijedili projekt kojeg je netko prethodno već granao, dobiti ćemo nešto kao:

```
$ git branch
  api
  development
  editable-text-pages
  less-compile
* master
```

Ili na primjer ovako:

```
$ git branch
  api
* development
  editable-text-pages
  less-compile
  master
```

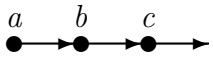
Svaki redak predstavlja jednu granu, a redak koji počinje zvjezdicom (*) **grana je u kojoj se trenutno nalazimo**. U toj grani tada možemo raditi sve što i na `master` – commitati, gledati njenu povijest, ...

Nova grana

Ako je trenutni ispis naredbe `git branch` ovakav:

```
$ git branch
* master
```

... to znači da je graf našeg projekta ovakav:



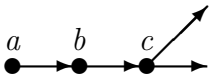
Sad se može desiti da nakon stanja *c* želimo isprobati dva različita pristupa. Novu granu možemo stvoriti naredbom `git branch <naziv_grane>`. Na primjer:

```
git branch eksperimentalna-grana
```

Sad je novo stanje projekta:

`eksperimentalna-grana:`

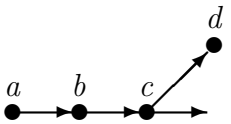
`master:`



Imamo granu `eksperimentalna-grana`, ali u njoj nemamo još ni jednog čvora (*commit*a). U toj grani sad možemo raditi točno onako kako smo se do sada naučili raditi s `master`: mijenjati (dodavati, brisati) datoteke, spremati ih u indeks s `git add` i *commit*aati s `git commit`. Sad bismo dobili da naša nova grana ima svoj prvi čvor:

`eksperimentalna-grana:`

`master:`



Prebacivanje s grane na granu

Primijetimo da se i dalje "nalazimo" na `master` grani:

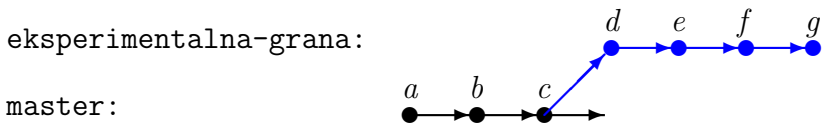
```
$ git branch
  eksperimentalna-grana
* master
```

Naime, `git branch` stvorit će nam samo novu granu. Prebacivanje s jedne grane na drugu granu radi se naredbom `git checkout <naziv_grane>`:

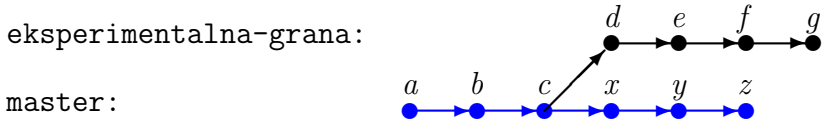
```
$ git checkout eksperimentalna-grana
Switched to branch 'eksperimentalna-grana'
```

Analogno, na glavnu se granu vraćamo s `git checkout master`.

Sada kad smo se prebacili na novu granu, možemo tu uredno *commit*ati svoje izmjene. Sve što tu radimo neće biti vidljivo u `master` grani.



Kad god želimo, možemo se prebaciti na `master` i tamo nastaviti rad koji nije nužno vezan uz izmjene u drugoj grani:



Nakon prebacivanja na `master`, izmjene koje smo napravili u *commit*ovima d, e, f i g neće nam biti vidljive. Kad se prebacimo na `eksperimentalna-grana`, neće nam biti vidljive izmjene iz x, y i z.

Ako ste ikad radili grane na nekom drugom, klasičnom, sustavu za verzioniranje koda, onda ste vjerojatno naviknuti da to grananje potraje malo duže (od nekoliko sekundi do nekoliko minuta). Stvar je u tome što u većini ostalih sustava *proces* grananja u stvari

podrazumijeva **kopiranje svih datoteka** na mjesto gdje se čuva nova grana. To em traje neko vrijeme, em zauzima više memorije na diskovima.

Kod gita je to puno jednostavnije. Nakon što kreiramo novu granu nema nikakvog kopiranja na disku. Čuva se samo informacija da smo kreirali novu granu i **posebne verzije datoteka koje su specifične za tu granu** (o tome više u posebnom poglavlju). Svaki put kad spremite izmjenu čuva se samo ta izmjena. Zahvaljujući tome postupak grananja je izuzetno brz i zauzima malo mjesta na disku.

Prebacivanje na granu i tekuće izmjene

Kod prebacivanja s grane na granu s `git checkout` može nastati manji problem ako imamo ne*commit*anih izmjena. Postoji nekoliko situacija u kojima nam git neće dopustiti prebacivanje. Najčešća je kada u dvije grane imamo dvije različite verzije iste datoteke, a tu smo datoteku u tekućoj grani izmijenili i ostavili ne*commit*anu.

Zapamtite, **najbolje je prebacivati se s grane na granu tek nakon što smo *commit*ali sve izmjene**. Tako će nas u novoj grani dočekati čista situacija, a ne datoteke koje smo izmijenili dok smo radili na prethodnoj grani.

Brisanje grane

Zato što je grananje memorijski i procesorski nezahtjevno i brzo, pripremite se na situacije kad ćete se naći s **previše** grana. Možda smo neke grane napravili da bismo isprobali nešto novo, a to se na kraju pokazalo kao loša ideja pa smo granu napustili. Ili smo je započeli da bismo riješili neki problem, ali taj problem je prije nas riješio netko drugi.

U tom slučaju granu možemo obrisati s `git branch -D <naziv_grane>`. Dakle, ako je stanje grana na našem projektu:

```
$ git branch
  eksperimentalna-grana
* master
```

. . . nakon:

```
$ git branch -D eksperimentalna-grana
Deleted branch eksperimentalna-grana (was 1658442).
```

... novo stanje će biti:

```
$ git branch
* master
```

Primijetimo samo da sad ne možemo obrisati `master`:

```
$ git branch -D master
error: Cannot delete the branch 'master' which you are currently on.
```

To vrijedi i općenito, ne možemo obrisati granu na kojoj se trenutnačno nalazimo.

Treba znati i to da brisanjem grane ne brišemo njene *commit*ove. Oni ostaju dio povijesti do daljnjega. Više detalja o tome koji točno *commit*ovi i u kojim se uvijetima brišu iz povijesti će biti u poglavlju o "higijeni" projekta.

Preuzimanje datoteke iz druge grane

S puno grana, događati će se svakakve situacije. Relativno česta situacija je da želimo preuzeti samo jednu ili više datoteka iz druge grane, ali ne želimo **preći** na tu drugu granu. Znamo da su neke datoteke u drugoj grani izmijenjene i želimo ih preuzeti u trenutnu granu. To se može ovako:

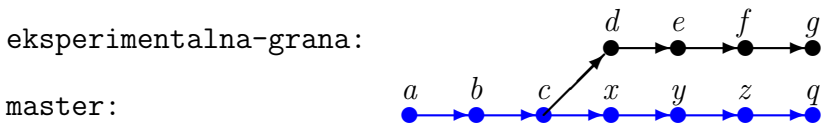
```
git checkout <naziv_grane> -- <datoteka1> <datoteka2> ...
```

Na primjer, ako smo u `master`, a treba nam datoteka `.classpath` koju smo izmijenili u `eksperiment`, onda ćemo je dobiti s:

```
git checkout eksperiment -- .classpath
```

Preuzimanje izmjena iz jedne grane u drugu

Vratimo se opet na već viđenu ilustraciju:



Prebacivanjem na granu `master`, izmjene koje smo napravili u *commit*ovima d, e, f i g nam neće više biti dostupne. Slično, prebacivanjem na `eksperimentalna-grana` – neće nam biti dostupne izmjene iz x, y i z.

To je u redu dok svoje izmjene želimo raditi u izolaciji od ostatka koda. Što ako smo u `eksperimentalna-grana` ispravili veliki bug i htjeli bismo sad tu ispravku preuzeti u `master`?

Ili, što ako zaključimo kako je rezultat eksperimenta kojeg smo isprobali u `eksperimentalna-grana` uspješan i želimo to sad imati u `master`? Ono što nam sad treba je da nekako **izmjene iz jedne grane preuzmemo u drugu granu**. U gitu, to se naziva *merge*. Iako bi merge mogli doslovno prevesti kao "spajanje", to nije ispravna riječ. Rezultat spajanja bi bila samo jedna grana. Nakon *merge*a dvije grane – one nastavljaju svoj život. Jedino što se sve izmjene koje su do tog trenutka rađene u jednoj grani preuzimaju u drugu granu.

Git *merge*

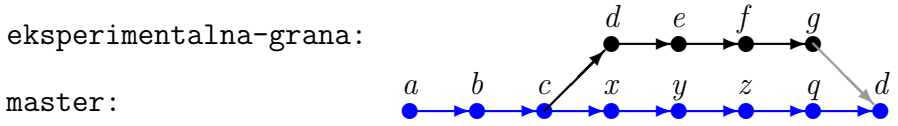
Pretpostavimo, na primjer, da sve izmjene iz `eksperimentalna-grana` želimo u `master`. To se radi s naredbom `git merge`, a da bi to napravili **trebamo se nalaziti u onoj grani u koju želimo preuzeti izmjene** (u našem slučaju `master`). Tada:

```
$ git merge eksperimentalna-grana
Updating 372c561..de69267
Fast-forward
 fig1.tex      |   23 ----------------------
 git-merge.tex |    1 +
 uvod.tex      |   13 ++++++++-----
 3 files changed, 9 insertions(+), 28 deletions(-)
 delete mode 100644 fig1.tex
```

Rezultat naredbe `git merge` je rekapitulacija procesa preuzimanja izmjena: koliko je linija dodano, koliko obrisano, koliko je datoteka dodano, koliko obrisano, itd... Sve tu piše.

Još nešto je važno napomenuti – ako je `git merge` proveden bez grešaka, to automatski dodaje novi *commit* u grafu. Ne moramo "ručno" *commit*ati.

Grafički se `git merge` može prikazati ovako:



Za razliku od svih ostalih *commit*ova, ovaj ima dva "roditelja" iliti "prethodnika"[18] – jednog iz grane u kojoj se nalazi i drugog iz grane iz koje su izmjene preuzete. Odnosno, na grafu čvor d jedini ima dvije strelice koje idu "u njega".

Kad smo preuzeli izmjene iz jednog grafa u drugi – obje grane mogu uredno nastaviti svoj dosadašnji "život". U obje možemo uredno *commit*ati, preuzimati izmjene iz jedne grane u drugu, i sl. Kasnije možemo i ponoviti preuzimanje izmjena, odnosno *merge*anje.

[18] Ostali *commit*ovi imaju ili jednog prethodnika ili nijednog – ako se radi o prvom *commit*u u repozitoriju.

I, eventualno, jednog dana kad odlučimo da nam grana više ne treba, možemo ju obrisati.

Što *merge* radi kada...

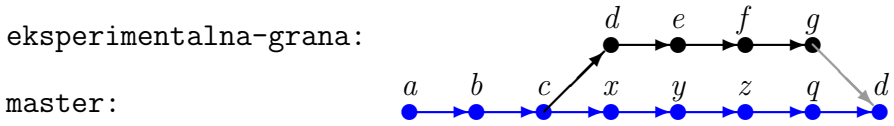
Vjerojatno se pitate što je rezultat *merge*anja u nekim situacijama. Na primjer u jednoj grani smo dodali datoteku, a u drugoj editirali ili u jednoj editirali početak datoteke, a u drugoj kraj iste ili...

Kad sam ga ja počeo koristiti imao sam malu tremu prije svakog `git merge`. Jednostavno, iskustvo CVS-om, SVN-om i TFS-om mi nije baš ulijevalo povjerenja u to da ijedan sustav za verzioniranje koda zna ovu radnju izvršiti kako treba. Nakon svakog *merge*a sam išao proučavati je li rezultat ispravan i provjeravao bih da mi nije možda pregazio neku datoteku ili važan komad koda. Rezultat tog neformalnog istraživanja je: Moja (ljudska) i gitova (strojna) intuicija o tome što treba napraviti se skoro uvijek poklapaju.

Ne brinite se, git će napravi ono što treba.

Umjesto beskonačnih hipotetskih situacija tipa "Što će git napraviti ako sam u grani A izmijenio x, a u grani B izmijenio y..." – najbolje je da to jednostavno isprobate. U ostatku ovog poglavlja ćemo samo proći nekoliko posebnih situacija.

Uzmimo poznati slučaj:



Dakle, što će biti rezultat *merge*anja, ako...

- ...u eksperimentalnoj grani smo izmijenili datoteku, a u `master` nismo – izmjene iz eksperimentalne će se dodati u `master`.
- ...u eksperimentalnoj grani smo dodali datoteku – ta datoteka će biti dodana i u `master`.
- ...u eksperimentalnoj grani smo izbrisali datoteku – datoteka će biti obrisana u glavnoj.

- ... u eksperimentalnoj grani smo **izmijenili i preimenovali** datoteku, a u `master` ste samo izmijenili datoteku – ako izmjene na kodu nisu bile **konfliktne**, onda će se u `master` datoteka preimenovati i sadržavati će izmjene iz obje grane.
- ... u eksperimentaloj grani smo obrisali datoteku, a u glavnoj ju izmijenili – **konflikt**.
- itd...

Vjerojatno slutite što znači ova riječ koja je ispisana masnim slovima: **konflikt**. Postoje slučajevi u kojima git ne zna što napraviti. I tada se očekuje od korisnika da sam riješi problem.

Što se dogodi kad...

Stvar nije uvijek tako jednostavna. Dogoditi će se da u jednoj grani napravite izmjenu u jednoj datoteci, a u drugoj grani napravite izmjenu na *istoj* datoteci. I što onda?

Pokušat ću to ilustrirati na jednom jednostavnom primjeru... Uzmimo hipotetski scenarij – neka je Antun Branko Šimić još živ i piše pjesme. Napiše pjesmu, pa s njome nije baš zadovoljan, pa malo križa po papiru, pa izmijeni prvi stih, pa izmijeni zadnji stih. Ponekad mu se rezultat sviđa, ponekad ne. Ponekad krene iznova. Ponekad ima ideju, napiše nešto nabrzinu, i onda kasnije napravi dvije verzije iste pjesme. Ponekad... Kao stvoreno za git, nije li?

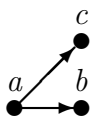
Recimo da je autor krenuo sa sljedećom verzijom pjesme:

```
PJESNICI U SVIJETU

Pjesnici su čuđenje u svijetu

Oni idu zemljom i njihove oči
velike i nijeme rastu pored stvari

Naslonivši uho
na tišinu sto ih okružuje i muči
oni su vječno treptanje u svijetu
```

I sad ovdje nije baš bio zadovoljan sa cjelinom i htio bi isprobati dvije varijante. Budući da ga je netko naučio git, iz početnog stanja (a) napravio je dvije verzije.

`varijanta:`

`master:`



U prvoj varijanti (b), izmijenio je naslov, tako da je sad pjesma glasila:

```
PJESNICI

Pjesnici su čuđenje u svijetu

Oni idu zemljom i njihove oči
velike i nijeme rastu pored stvari

Naslonivši uho
na tišinu sto ih okružuje i muči
oni su vječno treptanje u svijetu
```

...dok je u drugoj varijanti (c) izmijenio zadnji stih:

```
PJESNICI U SVIJETU

Pjesnici su čuđenje u svijetu

Oni idu zemljom i njihove oči
velike i nijeme rastu pored stvari

Naslonivši uho
na ćutanje sto ih okružuje i muči
pjesnici su vječno treptanje u svijetu
```

S obzirom da je bio zadovoljan s oba riješenja, odlučio je izmjene iz varijante `varijanta`

preuzeti u `master`. Nakon `git checkout master` i `git merge varijanta`, rezultat je bio:

... odnosno, pjesnikovim riječima:

```
PJESNICI

Pjesnici su čuđenje u svijetu

Oni idu zemljom i njihove oči
velike i nijeme rastu pored stvari

Naslonivši uho
na ćutanje sto ih okružuje i muči
pjesnici su vječno treptanje u svijetu
```

I to je jednostavno. U obje grane je mijenjao istu datoteku, ali u jednoj je dirao početak, a u drugoj kraj. I rezultat *merge*anja je bio očekivan – datoteka u kojoj je izmijenjen i početak i kraj.

Konflikti

Što da je u obje grane dirao isti dio pjesme? Što da je nakon:

... stanje bilo ovakvo: U verziji *a* je pjesma sad glasila:

```
PJESNICI

Pjesnici su čuđenje u svijetu

Pjesnici idu zemljom i njihove oči
velike i nijeme rastu pored ljudi

Naslonivši uho
na ćutanje sto ih okružuje i muči
pjesnici su vječno treptanje u svijetu
```

...a u verziji *b*:

```
PJESNICI

Pjesnici su čuđenje u svijetu

Oni idu zemljom i njihova srca
velika i nijema rastu pored stvari

Naslonivši uho
na ćutanje sto ih okružuje i muči
pjesnici su vječno treptanje u svijetu
```

Sad je rezultat naredbe `git merge varijanta`:

```
$ git merge varijanta
Auto-merging pjesma.txt
CONFLICT (content): Merge conflict in pjesma.txt
Automatic merge failed; fix conflicts and then commit the result.
```

To znači da git nije znao kako automatski preuzeti izmjene iz *b* u *a*. Idete li sad editirati datoteku s pjesmom naći ćete:

```
PJESNICI U SVIJETU

Pjesnici su čudenje u svijetu

<<<<<<< HEAD
Oni idu zemljom i njihova srca
velika i nijema rastu pored stvari
=======
Pjesnici idu zemljom i njihove oči
velike i nijeme rastu pored ljudi
>>>>>>> eksperimentalna-grana

Naslonivši uho na tišinu sto ih okružuje i muči
oni su vječno treptanje u svijetu
```

Dakle, dogodio se **konflikt**. U crveno je obojan dio za kojeg git ne zna kako ga *merge*ati. S `HEAD` je označeno stanje iz trenutne grane, a s `eksperimentalna-grana` stanje iz druge grane.

Za razliku od standardnog *merge*, ovdje niti jedna datoteka nije *commit*ana. To možete lako provjeriti sa `git status`:

```
$ git status
# On branch master
# Unmerged paths:
#   (use "git add/rm <file>..." as appropriate to mark resolution)
#
#       both modified:      pjesma.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Sad se od autora očekuje da sam odluči kako će izgledati datoteka nakon *merge*a. Jednostavan način je da editira tu datoteku i sam ju izmijeni kako želi. Nakon toga treba ju *commit*ati na standardan način.

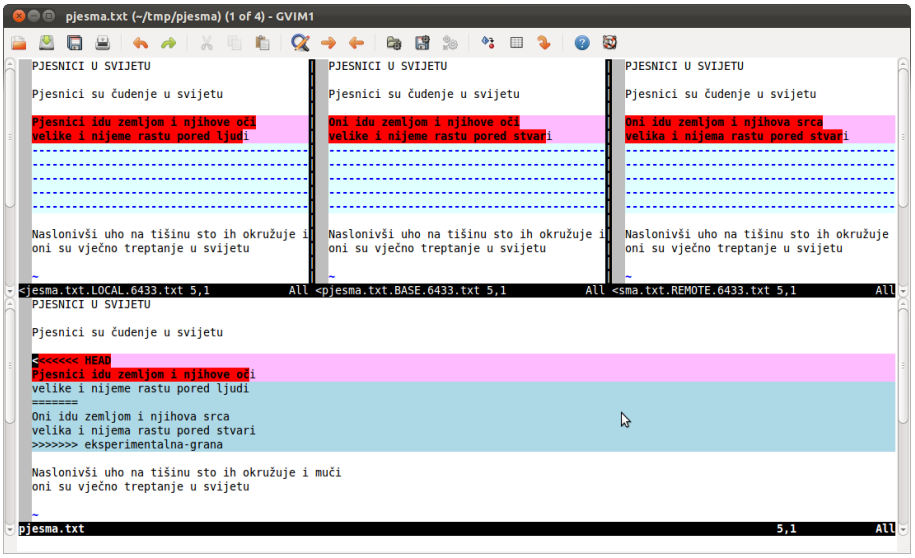
Drugi način je da koristi `git mergetool`. Ako se sjećate početka ove knjige, govorilo se o standardnoj konfiguraciji. Jedna od opcija je tamo bila i "mergetool", a to je

program s kojim lakše rješavate ovakve konflikte.

Git, sam po sebi, nema ugrađenih alata za vizualni prikaz i riješavanje konflikata, ali postoje zasebni programi u tu svrhu. Kad nađete jedan takav koji vam odgovara, postavite ga kao zadani `merge.tool` alat:

```
git config --global merge.tool /putanja/do/programa
```

Sad, kad dođe do konflikta i pokrenete `git mergetool`, git će upravo tu aplikaciju pokrenuti i olakšati vam snalaženje u konfliktnim datotekama. Na primjer, ako koristite `vimdiff`, onda editiranje konfliktnih datoteka izgleda ovako:



Iako to nije nužno (možete uvijek sami editirati konfliktne datoteke) – to je praktično rješenje za sve koji su naviknuti na slične alate.

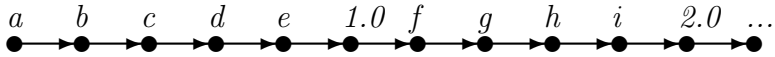
Nakon što konflikt u datoteci (ili datotekama) riješite – izmijenjene datoteke treba *commit*ati.

Merge, *branch* i povijest projekta

Kad radimo s nekim projektom, onda nam je važno da imamo sačuvanu cijelu njegovu povijest. To nam omogućava da saznamo tko je originalni autor koda kojeg možda

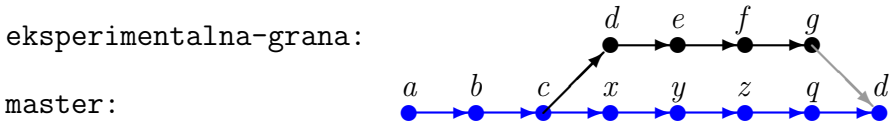
slabije razumijemo. Ili želimo vidjeti kojim redoslijedom je neka datoteka nastajala.

Sa standardnim sustavima za verzioniranje, stvar je jednostavna. Grananje i preuzimanje izmjena iz jedne u drugu granu je bilo kompliciranо i rijetko se radilo. Posljedica je da su projekti najčešće imali linearnu povijest:



S gitom često imamo po nekoliko grana. Svaka od tih grana ima svoju povijest, a kako se povećava broj grana tako organizacija projekta postaje veći izazov. I zato programeri grane koje više ne koriste brišu[19].

Tu sad imamo mali problem. Pogledajte, na primjer, ovakav projekt:



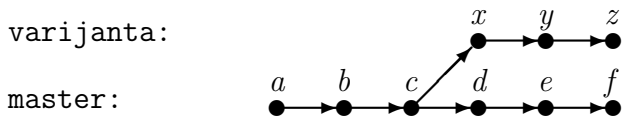
Eksperimentalna grana ima svoju povijest, a u trenutku *merge*a, sve izmjene iz te grane su se "prelile" u `master` i to u *commit d*. Postoje situacije u kojima moramo gitovu "razgranatu" povijest svesti na linearnu. U drugim situacijama, jednostavno želimo imati ljepši (linearni!) pregled povijesti projekta.

To se može riješiti nečime što se zove *rebase*. Da bi to mogli malo bolje objasniti, potrebno je napraviti digresiju u jednu posebnu vrstu *merge*a – *fast forward*...

Fast forward

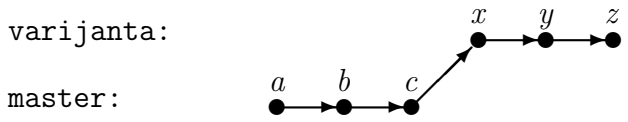
Nakon objašnjenja s prethodnih nekoliko stranica, trebalo bi biti jasno što će se dogoditi ako želimo preuzeti izmjene iz `varijanta` u `master` u projektu koji ima ovakvu povijest:

[19] Treba biti jasno da se brisanjem grane ne brišu nužno i *commit*ovi od kojih se sastojala. Koji se točno *commit*ovi gube brisanjem će biti opisano u poglavlju o "higijeni" repozitorija.



To je najobičniji *merge* dvije grane. Razmislimo samo, na trenutak, o jednom očitom detalju; osnovna pretpostavka i smisao preuzimanja izmjena iz jedne grane u drugu je to da uopće imamo dvije grane. To su ove dvije crte u gornjem grafu. Dakle, sve to ima smisla u projektu koji ima nelinearnu povijest (više grana).

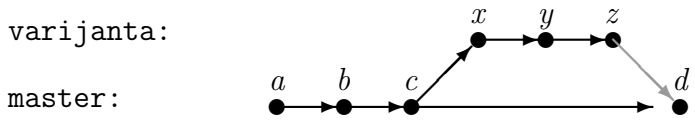
Postoji jedan slučaj koji zahtijeva pojašnjenje. Uzmimo da je povijest projekta bila slična gornjem grafu, ali s jednom malom izmjenom:



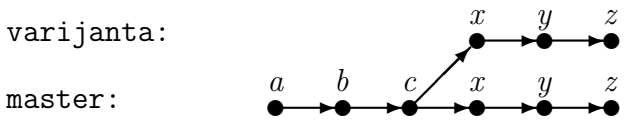
Programer je napravio novu granu `varijanta` i na njoj je nastavio rad. I svo to vrijeme nije radio nikakve izmjene na `master`. Što kad sad želi preuzeti sve izmjene u `master`?

Uočavate li što je ovdje neobično? Smisao *merge*anja je u tome da neke izmjene iz jedne grane preuzmemo u drugu. Međutim, iako ovdje imamo dvije grane, **te dvije grane čine jednu crtu**. One imaju jednu povijest. I to linearnu povijest. Jedino što se ta linearna povijest proteže kroz obje grane.

Tako su razmišljali i originalni autori gita. U git su ugradili automatsko prepoznavanje ovakve situacije i zato, umjesto standardnog *merge*a, koji bi izgledao ovako:



. . . git izvršava takozvani *fast-forward merge*:



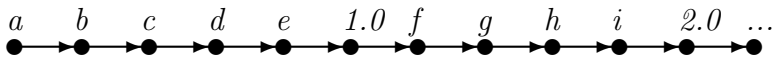
Dakle, kopira cijelu povijest (ovdje je to x, y i z) u novu granu[20]. Čak i ako sad obrišete `varijanta`, cijela njegova povijest se nalazi u `master`.

Git sam odlučuje je li potrebno izvršiti *fast-forward merge* i izvršava ga. Želimo li ga izbjeći – to se radi tako da dodamo opciju `--no-ff` naredbi `git merge`:

```
git merge --no-ff varijanta
```

Rebase

Idemo, još jednom, pogledati linearni model:

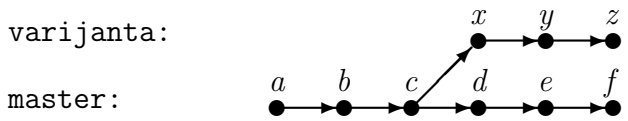


Do sada bi svima trebalo biti jasno da on ima svoje nedostatke, ali ima i pozitivnih strana – jednostavna i pregledna povijest projekta i privid da je sve skupa teklo po nekom točno određenom rasporedu. Korak po korak, do trenutne verzije.

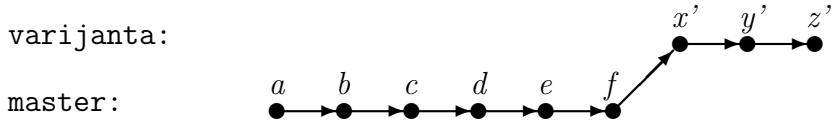
Git nas ne tjera da radimo grane, no postupak grananja čini bezbolnim. I zbog toga povijest projekta **može** postati cirkus kojeg je teško pratiti i organizirati. Organizacija repozitorija zahtijeva posebnu pažnju, posebno ako radite s više ljudi na istom projektu.

Kad bi samo postojao trik kako da iz ovakvog stanja:

[20]Preciznije, čvorovi x, y i z se sad nalaze u dvije grane. U gitu jedan *commit* može biti dio više različitih grana.

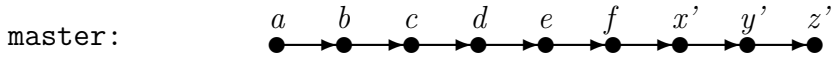


...stvorimo ovo...



...tada bi *fast-forward* samo kopirao cijelu našu granu u `master`.

Drugim riječima – treba nam način da *pomaknemo mjesto od kud smo granali* neku granu. Nakon toga bi brisanjem grane `varijanta` dobili da su se svi *commit*ovi našeg projekta događali u `master`:



Taj trik postoji i zove se *rebase*.

Radi se na sljedeći način; trebamo biti postavljeni u grani koju želimo "pomaknuti". Zatim `git rebase <grana>`, gdje je `<grana>` ona grana na kojoj kasnije treba izvršiti *fast-forward*. Želimo li granu `test` "pomaknuti" na kraj grane `master`, (to jest, izvršiti *rebase*):

```
git rebase master
```

U idealnoj situaciji, git će znati riješiti sve probleme i rezultat naredbe će izgledati ovako:

```
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: Prvi commit
Applying: Drugi commit
```

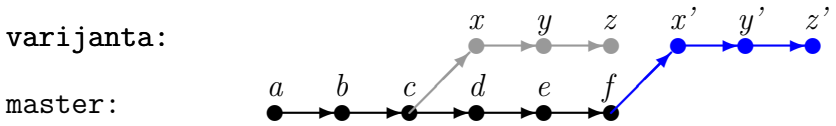
Međutim, ovdje mogu nastati problemi slični klasičnom *merge*u. Tako da će se ponekad dogoditi:

```
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: test
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
Auto-merging pjesma.txt
CONFLICT (content): Merge conflict in pjesma.txt
Failed to merge in the changes.
Patch failed at 0001 test

When you have resolved this problem run "git rebase --continue".
If you would prefer to skip this patch, instead run "git rebase
--skip".
To restore the original branch and stop rebasing run "git rebase
--abort".
```

Pogledate li konfliktne datoteke, vidjeti ćete da je njihov format isti kao kod konflikta pri *merge*u. Na isti način se od nas očekuje da konflikte riješimo. Bilo koristeći `git mergetool`, bilo tako da editiramo datoteku i ispravimo ju tako da dobijemo željeno stanje.

Kod grananja su konflikti vjerojatno malo jasniji – točno znamo da jedna izmjena dolazi iz jedne grane, a druga iz druge. Ovdje će nam se možda dogoditi[21] da nismo točno sigurni koju konfliktnu izmjenu uzeti. Ili ostaviti obje. Promotrimo, zato, još jednom grafički prikaz onoga što pokušavamo napraviti:

[21] Meni jest, puno puta



Naš cilj je preseliti sivi dio tako da postane plavi. Drugim riječim, izmjene koje smo napravili u d, e i f treba nekako "ugurati" prije x, y i z. Tako ćemo stvoriti novu granu koja se sastoji x', y' i z'.

Konflikt se može dogoditi u trenutku kad git "seli" x u x', ili y u y' ili z u z'. Recimo da je konflikt nastao u prvom slučaju (x u x'). Negdje u čvorovima d, e i f je mijenjan isti kod koji smo mi mijenjali u x[22]. Mi ovdje želimo zadržati povijest iz svih *commit*ova: d, e, f, x, y i z. To treba imati na umu dok odlučujemo kako ćemo ispraviti konflikt.

U slučaju konflikta, često ćemo htjeti zadržati obje verzije konfliktnog koda, treba samo pripaziti na njihov redoslijed i potencijalne sitne bugove koji mogu nastati.

Nakon što smo konflikt ispravili, **ne smijemo izmjene *commit*ati, nego samo spremiti u indeks s `git add`**. Nastavljamo s:

```
git rebase --continue
```

Ponekad će git znati izvršiti ostatak procesa automatski, a može se dogoditi i da ne zna i opet od nas traži da ispravimo sljedeći konflikt. U boljem slučaju (sve automatski), rezultat će biti:

```
$ git rebase --continue
Applying: test
```

...i sad smo slobodni izvesti *merge*, koji će u ovom slučaju sigurno biti *fast-forward*, a to je upravo ono što smo htjeli.

Ako *rebase* ima previše konflikata – možda se odlučimo odustati od nastavka. Prekid *rebase*a i vraćanje repozitorija u stanje prije nego što smo ga pokrenuli možemo obaviti s:

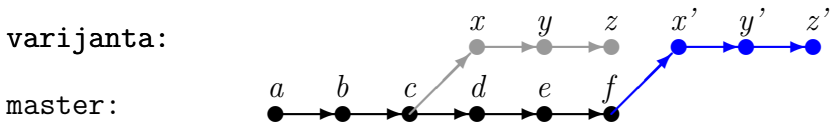
[22] Primijetite da bi do ovog konflikta došli i s klasičnim *merge*om.

```
git rebase --abort
```

Rebase ili ne *rebase*?

Na vama je odluka želite li jednostavniju (linearnu) povijest projekta ili ne. Ukoliko želite – raditi ćete *rebase*.

Treba imati na umu jednu važnu stvar: *rebase* **mijenja povijest projekta**:



Kad ste prvi put *commit*ali x, u vašem kodu nije bilo izmjena koje su nastale u d, e i f. Ali, kad netko kasnije bude gledao *commit* x', izgledati će mu da su izmjene iz d, e i f bile tamo **prije** njega. Drugim riječima, kad ste x "preselili" u x' vi ste promijenili **kontekst** tog *commit*a.

Zbog toga postoji par nepisanih pravila kad se rebase može raditi:

- *Rebase* **radite ako su svi** *commit***ovi koji će biti mijenjani vaši vlasititi** *commit***ovi**. Dakle, ako ste vi autor od x, y i z onda se možete odlučiti raditi *rebase*. Ako je neka druga osoba autor npr. od y onda nemojte raditi *rebase*.
- Ako je ikako moguće, *rebase* radite na branchevima koje još niste *push*ali na udaljene repozitorije[23].

Rebase i rad sa standardnim sustavima za verzioniranje

Postoji jedan način korištenja gita u kojem je *rebase* nužan. To je **ako koristite git kao CVS, subversion ili TFS klijent**. Nećemo ovdje u detalje, ali ukratko: moguće je koristiti git kao klijent za druge (standardne) sustave za verzioniranje[24].

[23] Više o udaljenim repozitorijima kasnije

[24] Uz instalaciju posebnih *plugin*ova, naravno.

*Commit*ovi koji su slika udaljenog repozitorija se, u takvim slučajevima, uvijek čuvaju u posebnoj grani. Dakle, jedna grana je doslovna kopija udaljenog repozitorija (CVS, subversion ili TFS). Tu granu možemo osvježavati s novim *commit*ovima iz centralnog repozitorija.

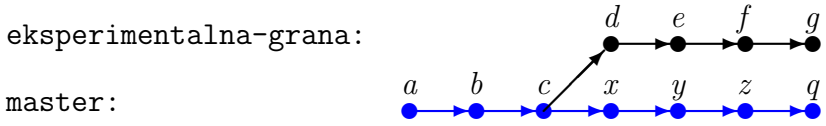
Našu gitovsku čudnovatu i razgranatu povijest treba svesti na povijest kakvu ti repozitoriji najbolje razumiju, a to je linearna. Zato, kad treba *commit*ati naše izmjene – sve što smo radili u drugim granama treba "preseliti" na kraj grane koja je kopija centralnog repozitorija. Taj postupak nije ništa drugo nego *rebase*. Tek tada možemo *commit*ati.

S obzirom da centralizirani sustavi za verzioniranje više vole linearnu od razgranate povijesti – moramo igrati po njihovim pravilima, a *rebase* je jedini način da to učinimo.

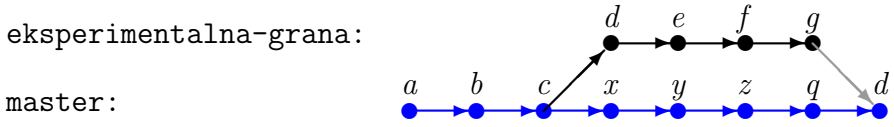
Cherry-pick

Ima još jedna posebna vrsta *merge*a, a nosi malo neobičan naziv *cherry-pick*[25].

Pretpostavimo da imamo dvije grane:



U `eksperimentalna-grana` smo napravili nekoliko izmjena i sve te izmjene *nisu* još spremne da bismo ih prebacili u `master`. Međutim, ima jedan *commit* (recimo da je to *g*) s kojim smo ispravili mali bug koji se manifestirao i u `master` grani. Klasični *merge* oblika:



[25]Engleski: branje trešanja. U prenesenom značenju: ispričati priču samo sa onim detaljima koji su pripovjedaču važni. Iliti, izbjegavanje onih dijelova priče koje pripovjedač ne želi.

... ne dolazi u obzir, jer bi on u `master` prebacio sve izmjene koje smo napravili u *d*, *e*, *f* i *g*. Mi želimo samo i isključivo *g*. To se, naravno, može i to je (naravno) taj *cherry-pick*.

Postupak je sljedeći: prvo promotrimo povijest grane `eksperimentalna-grana`:

```
$ git log eksperimentalna-grana
commit 5c843fbfb09382c272ae88315eea9d77ed699083
Author: Tomo Krajina <tkrajina@gmail.com>
Date:   Tue Apr 3 21:57:08 2012 +0200

    Komentar uz zadnji commit

commit 33e88c88dcad48992670ff7e06cebb0e469baa60
Author: Tomo Krajina <tkrajina@gmail.com>
Date:   Tue Apr 3 13:38:24 2012 +0200

    Komentar uz predzadnji commit

commit 2b9ef6d02e51a890d508413e83495c11184b37fb
Author: Tomo Krajina <tkrajina@gmail.com>
Date:   Sun Apr 1 14:02:14 2012 +0200

...
```

Kao što ste vjerojatno već primijetili, svaki *commit* u povijesti ima svoj string kao npr. `5c843fbfb09382c272ae88315eea9d77ed699083`. Taj string jedinstveno određuje svaki *commit* (više riječi o tome u posebnom poglavlju).

Sad trebamo naći takav identifikator za onaj commit kojeg želite prebaciti u `master`. Recimo da je to `2b9ef6d02e51a890d508413e83495c11184b37fb`.

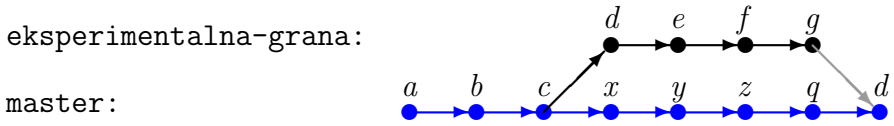
Prebacimo se u granu u koju želimo preuzeti samo izmjene iz tog *commit*a i utipkajmo `git cherry-pick <commit>`. Ako nema konflikata, rezultat će biti:

```
$ git cherry-pick 2b9ef6d02e51a890d508413e83495c11184b37fb
[master aab1445] Commit...
 2 files changed, 26 insertions(+), 0 deletions(-)
 create mode 100644 datoteka.txt
```

. . . a, ako imamo konflikata onda. . . To već znamo riješiti, nadam se.

Merge bez *commita*[26]

Vratimo se opet na klasični *merge*. Ukoliko je prošao bez ikakvih problema, onda ćemo nakon `git merge eksperimentalna-grana`:



. . . u povijesti projekta vidjeti nešto kao:

```
$ git log master
commit d013601dabdccc083b4d62f0f46b30b147c932c1
Merge: aab1445 8be54a9
Author: Tomo Krajina <tkrajina@gmail.com>
Date:   Wed Apr 4 13:12:01 2012 +0200

    Merge branch 'eksperimentalna-grana'
```

Ukoliko koristite noviju verziju gita, onda će on nakon *merge*a otvoriti editor i ponuditi vam da sami upišete poruku za taj *commit*.

Ako se već odlučimo da ne želimo *rebase* – u povijesti ćemo imati puno grana i čvorova u kojima se one spajaju. Bilo bi lijepo kad bi umjesto `Merge branch 'eksperimentalna-grana'` imali smisleniji komentar koji bi bolje opisao što smo točno u toj grani napravili.

[26] Ovaj naslov se odnosi samo na starije git klijente.

To se može tako da, umjesto `git merge eksperimentalna-grana`, *merge* izvršimo s:

```
git merge eksperimentalna-grana --no-commit
```

Na taj način će se *merge* izvršiti, ali neće se sve *commit*ati. Sad možemo *commit*ati sa svojim komentarom ili eventualno napraviti još koju izmjenu prije nego se odlučimo snimiti novo stanje.

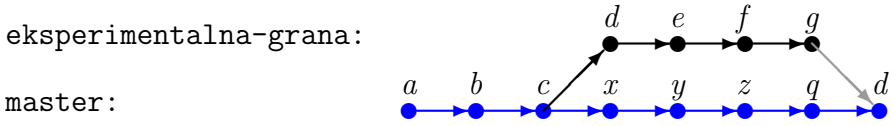
Jedini detalj na kojeg treba pripaziti je što, ako je došlo do *fast-forward merge*anja, onda `--no-commit` nema efekta. Zato je, za svaki slučaj, bolje koristiti sljedeću sintaksu:

```
git merge eksperimentalna-grana --no-ff --no-commit
```

Ukoliko ste zaboravili `--no-commit`, tekst poruke zadnjeg *commit*a možete ispraviti i s *amend commit*om.

Squash merge

Još jedna ponekad korisna opcija kod *merge*a je, takozvani *squash merge*. Radi se o sljedećem, klasični *merge* stvara commit kao d u grafu:



Čvor d ima dva prethodnika: q i g. Ukoliko želimo da d **ima** izmjene iz grane `eksperimentalna-grana`, ali **ne želimo** da d ima referencu na tu granu, to se dobije s[27]:

```
git merge --squash eksperimentalna-grana
```

[27] Ne brinite se ako vam ne pada na pamet scenarij u kojem bi to moglo trebati. Ni meni nije do jutros :)

Tagovi

Tag, "oznaka" iliti "ključna riječ" je naziv koji je populariziran s dolaskom takozvanih "web 2.0" sajtova. Mnogi ne znaju, ali *tag*ovi su postojali i prije toga. *Tag* je jedan od načina klasifikacije dokumenata.

Standardni način je hijerarhijsko klasifikaciranje. Po njemu, sve ono što kategoriziramo mora biti u nekoj kategoriji. Svaka kategorija može imati podkategorije i svaka kategorija može imati najviše jednu nadkategoriju. Tako klasificiramo životinje, biljke, knjige u knjižnici. Dakle, postoji "stablo" kategorija i samo jedan čvor može biti "korijen" tog stabla.

Za razliku od toga *tag*iranje je slobodnije. *Tag*irati možete bilo što i stvari koje *tag*iramo mogu imati proizvoljan broj *tag*ova. Kad bi u knjižnicama tako označavali knjige, onda one na policama ne bi bilo podijeljene po kategorijama. Sve bi bile poredane po nekom proizvoljnom redoslijedu (na primjer, vremenu kako su stizale u knjižicu), a neki *software* bi za svaku pamtio oznake iliti *tag*ove.

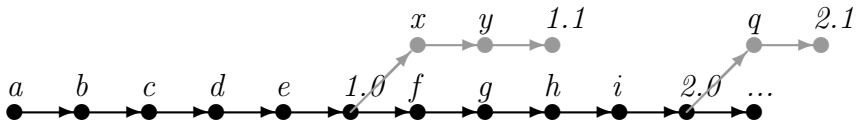
Mi ovdje radimo s poviješću projekata pa ćemo tu povijest i *tag*irati. Preciznije, *tag*irati ćemo čvorove našeg grafa povijesti projekta – *commit*ove. Za razliku od klasične klasifikacije s *tag*ovima – *tag* ovdje mora biti jedinstven. Dakle, jednom iskorišteni *tag* se ne može više koristiti.

Kao što znamo, u gitu svaki *commit* ima svoj identifikator[28]. Međutim, taj string je nama ljudima besmislen.

Nama su smisleni komentari uz kod, međutim, ovi **komentari nisu jedinstveni**. Projekt možemo pretraživati po riječima iz komentara, ali nema smisla od gita tražiti "Daj mi stanje projekta kakvo je bilo u trenutku kad je komentar *commit*a bio 'Ispravljen bug s izračunom kamate'". Jer, moglo se desiti da smo imali dva (ili više) *commit*a s istim komentarom.

[28] To je onaj string od 40 alfanumeričkih znakova.

Sjećate li se priče o verzioniranju koda? Bilo je riječi o primjeru programera koji radi na programu i izdaje verzije `1.0`, `1.1`, `2.0`, `2.1`... svoje aplikacije:



... e pa, *tag*ovi bi ovdje bili `1.0`, `1.1`, `2.0` i `2.1`.

Dakle, *tag* nije ništa drugo nego neki kratki naziv za određeni *commit*. Tag je **alias** za neki *commit*.

Rad s *tag*ovima je jednostavan; s `git tag` ćete dobiti popis svih tretnuno definiranih:

```
$ git tag
pocetak-projekta
1.0
1.1
test
2.0
2.1
```

S `git tag <naziv_taga>` dodajete novi tag:

```
git tag testni-tag
```

... dok s `git tag -d <naziv_taga>` brišete neki od postojećih tagova:

```
git tag -d testni-tag
```

Rad s tagovima je jednostavan, a ima samo jedna komplikacija koja se može dogoditi u radu s udaljenim projektima, no o tome ćemo malo kasnije.

Želimo li projekt privremeno vratiti na stanje kakvo je bilo u trenutku kad smo definirali *tag* `1.0`:

```
git checkout 1.0
```

I sad tu možete stvoriti novu granu s `git branch <grana>` ili vratiti projekt u zadnje stanje s `git checkout HEAD`.

Ispod haube

Kako biste vi...

Da kojim slučajem danas morate dizajnirati i implementirati sustav za verzioniranje koda, kako biste to napravili? Kako biste čuvali povijest svake datoteke?

Prije nego što ste počeli koristiti takve sustave, vjerojatno ste radili sljedeće: kad bi zaključili da ste došli do nekog važnog stanja u projektu, kopirali bi cijeli projekt u direktorij naziva `projekt_backup` ili `projekt_2012_04_05` ili neko drugo slično ime. Rezultat je da ste imali gomilu sličnih "backup" direktorija. Svaki direktorij predstavlja neko stanje projekta (dakle to je *commit*).

I to je nekakvo verzioniranje koda, ali s puno nedostataka.

Na primjer, nemate komentare uz *commit*ove, ali to bi se moglo srediti tako da u svaki direktorij spremite datoteku naziva `komentar.txt`. Nemate niti graf, odnosno redoslijed nastajanja *commit*ova. I to bi se moglo riješit tako da u svakom direktoriju u nekoj posebnoj datoteci, npr. `parents` nabrojite nazive direktorija koji su "roditelji" trenutnom direktoriju.

Sve je to prilično neefikasno što se tiče diskovnog prostora. Imate li u repozitoriju jednu datoteku veličine 100 kilobajta koju **nikad** ne mijenjate, njena kopija će opet zauzimati 100 kilobajta u svakoj kopiji projekta. Čak i ako nije nikad mijenjana.

Zato bi možda bilo bolje da umjesto **kopije direktorija** za *commit* napravimo novi u kojeg ćemo staviti **samo one datoteke koje su izmijenjene**. Zahtijevalo bi malo više posla jer morate točno znati koje su datoteke izmijenjene, ali i to se može riješiti. Mogli bi napraviti neku jednostavnu *shell* skriptu koja bi to napravila za nas.

S time bi se problem diskovnog prostora drastično smanjio. Rezultat bi mogli još malo poboljšati tako da datoteke kompresiramo.

Još jedna varijanta bi bila da ne čuvate izmijenjene datoteke, nego samo izmijenjene linije koda. Tako bi vaše datoteke, umjesto cijelog sadržaja, imale nešto tipa "Peta linija izmijenjena iz 'def suma_brojeva()' u 'def zbroj_brojeva()'". To su takozvane "delte". Još jedna varijanta bi bila da ne radite kopije direktorija, nego sve snimate u jednom tako da za svaku datoteku čuvate originalnu verziju i nakon toga (u istoj datoteci) dodajete delte. Onda će vam trebati nekakav pomoćni programčić kako iz povijesti izvući zadnju verziju bilo koje datoteke, jer on mora izračunati sve delte od početne verzije.

Sve te varijante imaju jedan suptilni, ali neugodan, problem. Problem konzistencije.

Vratimo se na trenutak na ideju s direktorijima sa izmijenjenim datotekama (deltama). Dakle, svaki novi direktorij sadrži samo datoteke koje su izmijenjene u odnosu na prethodni.

Ako svaki direktorij sadrži samo izmijenjene datoteke, onda prvi direktorij mora sadržavati **sve** datoteke. Pretpostavite da imate jednu datoteku koja nije nikad izmijenjena od prve do zadnje verzije. Ona će se nalaziti samo u prvom (originalnom) direktoriju.

Što ako neki zlonamjernik upadne u vaš sustav i izmijeni takvu datoteku? Razmislimo malo, on je upravo izmijenio ne samo početno stanje takve datoteke nego je **promijenio cijelu njenu povijest**! Kad bi vi svoj sustav pitali "daj mi zadnju verziju projekta", on bi protrčao kroz sva stanja projekta i dao bi vam "zlonamjernikovu" varijantu. Jeste li sigurni da bi primijetili podvaljenu datoteku?

Možda biste kad bi se vaš projekt sastojao od dvije-tri datoteke, ali što ako se radi o stotinama ili tisućama?

Rješenje je da je vaš sustav nekako dizajniran tako da sam prepoznaje takve izmijenjene datoteke. Odnosno, da je tako dizajniran da, ako bilo tko promijeni nešto u povijesti – sam sustav prepozna da nešto s njime ne valja. To se može na sljedeći način: neka jedinstveni identifikator svakog *commit*a bude neki podatak koji je **izračunat** iz trenutnog sadržaja svih datoteka u projektu. Takav jedinstveni identifikator će se nalaziti u grafu projekta, i sljedeći *commit*ovi će znati da im je on prethodnik.

Ukoliko bilo tko promijeni sadržaj nekog *commit*a, onda on više neće odgovarati tom identifikatoru. Promijeni li i identifikator, onda graf više neće biti konzistentan – sljedeći *commit* će sadržavati identifikator koji više ne postoji. Zlonamjernik bi trebao promijeniti sve *commit*ove do zadnjeg. U biti, trebao bi promijeniti previše stvari da bi mu cijeli poduhvat mogao proći nezapaženo.

Još ako je naš sustav distribuiran (dakle i drugi korisnici imaju povijest cijelog pro-

jekta) onda mu je još teže – jer tu radnju mora ponoviti na računalima svih ljudi koji imaju kopije. S distribuiranim sustavima, nitko sa sigurnošću ne zna tko sve ima kopije. Svaka kopija repozitorija sadrži povijest projekta. Ukoliko netko zlonamjerno manipulira poviješću projekta na jednom repozitoriju – to će se primijetiti kad se taj repozitorij "sinkronizira" s ostalima.

Nakon ovog početnog razmatranja, idemo pogledati koje od tih ideja su programeri gita uzeli kad su krenuli dizajnirati svoj sustav. Krenimo s problemom konzistentnosti.

SHA1

Znate li malo matematike čuli ste za jednosmerne funkcije. Ako i niste, nije bitno. To su funkcije koje je lako izračunati, ali je izuzetno teško iz rezultata zaključiti kakav je mogao biti početni argument. Takve su, na primjer, *hash* funkcije, a jedna od njih je SHA1.

SHA1 kao argument uzima string i iz njega izračunava drugi string duljine 40 znakova. Primjer takvog stringa je `974ef0ad8351ba7b4d402b8ae3942c96d667e199`.

Izgleda poznato?

SHA1 ima sljedeća svojstva:

- *Nije* jedinstvena. Dakle, sigurno postoje različiti ulazni stringovi koji daju isti rezultat, no **praktički ih je nemoguće naći**[29].
- Kad dobijete rezultat funkcije (npr. `974ef0ad8351ba7b4d402b8ae3942c96d667e199`) iz njega je **praktički nemoguće izračunati string iz kojeg je nastala**.

Takvih 40−znamenkastih stringova ćete vidjeti cijelu gomilu u `.git` direktoriju.

Git nije ništa drugo nego graf SHA1 stringova, od kojih svaki jedinstveno identificira neko stanje projekta **i izračunati su iz tog stanja**. Osim SHA1 identifikatora git uz svaki *commit* čuva i neke metapodatke kao, na primjer:

- Datum i vrijeme kad je nastao.

[29] Ovdje treba napomenuti kako SHA1 nije *potpuno* siguran. Ukoliko se nađe algoritam s kojime je moguće naći različite stringove kojima je rezultat funkcije SHA1 isti – tada cijela sigurnost potencijalno pada u vodu. Netko može podvaliti drukčiji string u povijest za isti SHA1. Postoje istraživanja koja naznačuju da se to može i zato je moguće da će git u budućnosti preći na SHA-256.

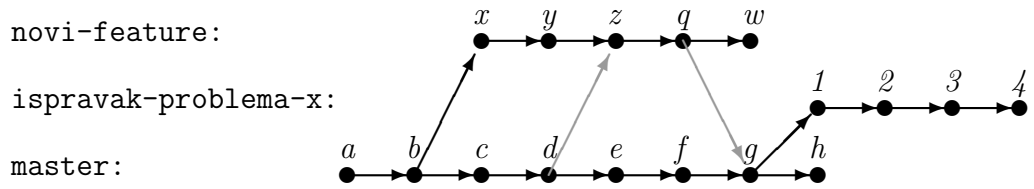
- Komentar
- SHA1 *commit*a koji mu je prethodio
- SHA1 *commit*a iz kojeg su preuzete izmjene za *merge* (**ako** je taj *commit* rezultat *merge*a).
- ...

Budući da je svaki *commit* SHA1 sadržaja projekta u nekom trenutku, kad bi netko htio neopaženo promijeniti povijest projekta, morao bi promijeniti i njegov SHA1 identifikator. Onda mora promijeniti i SHA1 njegovog sljedbenika, i sljedbenika njegovog sljedbenika, i...

Sve da je to i napravio na jednom repozitoriju – tu radnju mora ponoviti na svim ostalim **distribuiranim** repozitorijima istog projekta.

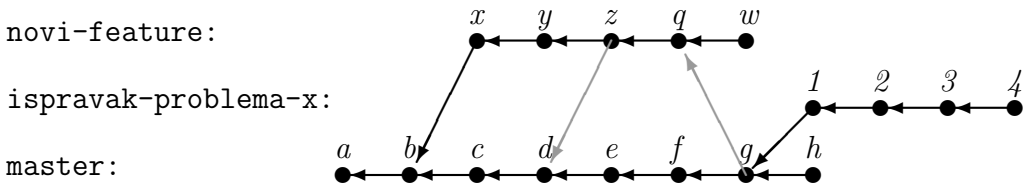
Grane

Razmislimo o još jednom detalju, uz poznati graf:



Takve grafove matematičari zovu "usmjereni grafovi" jer veze između čvorova imaju svoj smjer. To jest, veze između čvorova nisu obične relacije (crte) nego **usmjerene** relacije, odnosno strelice u jednom smjeru: $\vec{ab}$, $\vec{bc}$, itd. Znamo već da svaki čvor tog grafa predstavlja stanje nekog projekta, a svaka strelica neku izmjenu u novo stanje.

Sad kad znamo ovo malo pozadine oko toga kako git interno pamti podatke, idemo korak dalje. Prethodni graf ćemo ovaj put prikazati malo drukčije:



Sve strelice su ovdje usmjerene suprotno nego što su bile u grafovima kakve smo do sada imali. Radi se o tome da git upravo tako i "pamti" veze između čvorova. Naime, čvor h ima referencu na g, g ima reference na f i na q, itd. Uočite da nam uopće nije potrebno znati da se grana `novi-feature` sastoji od x, y, z, q i w. Dovoljan nam je w. Iz njega možemo, prateći reference "unazad" (suprotno od redoslijeda njihovog nastajanja) doći sve do mjesta gdje je grana nastala. Tako, na osnovu samo jednog čvora (*commit*a) možemo saznati cijelu povijest neke grane.

Dakle, dovoljno nam je imati reference na zadnje *commit*ove svih grana u repozitoriju, da bi mogli saznati povijest cijelog projekta. Zato **gitu grane i nisu ništa drugo nego reference na njihove zadnje *commit*ove**.

Reference

SHA1 stringovi su računalu praktični, no ljudima su nezgodni za pamćenje. Zbog toga git ima par korisnih sitnica vezanih uz reference.

Pogledajmo SHA1 string `974ef0ad8351ba7b4d402b8ae3942c96d667e199`. Takav string je teško namjerno ponoviti. I vjerojatno je mala vjerojatnost da postoji neki drugi string koji započinje s `974ef0a` ili `974e`. Zbog toga se u gitu može slobodno koristiti i samo prvih nekoliko znakova SHA1 referenci umjesto cijelog 40-znamenkastog stringa.

Dakle,

```
git cherry-pick 974ef0ad8351ba7b4d402b8ae3942c96d667e199
```

... je isto što i:

```
git cherry-pick 974ef0
```

Dogodi li se, kojim slučajem, da postoje dvije reference koje počinju s `974ef0a`, git će vam javiti grešku da ne zna na koju od njih se naredba odnosi. Tada samo dodajte jedan ili dva znaka više (`974ef0ad` ili `974ef0ad8`), sve dok nova skraćenica reference ne postane jedinstvena.

HEAD

`HEAD` je referenca na trenutni *commit*. Obično je to zadnji *commit* u grani u kojoj se nalazimo, ali može biti i bilo koji drugi. Ukoliko pokazuje na *commit* koji **nije zadnji u grani** – onda se kaže da je repozitorij u `detached HEAD` stanju.

Ukoliko nam treba referenca na predzadnji *commit*, mogli bi pogledati `git log` i tamo naći njegov SHA1. Postoji i lakši način: `HEAD~1`. Pred-predzadnji commit je `HEAD~2`, i tako dalje...

Na primjer, ako se prebacimo na stanje u predzadnjem *commit*u, to možemo napraviti s:

```
git checkout HEAD~1
```

...i za git smo sada u `detached HEAD` stanju. Na spisku grana – dobiti ćemo:

```
$ git branch
* (no branch)
  master
  grana_1
  grana_2
```

U ovakvoj situaciji smijemo čak i *commit*ati, ali te izmjene neće biti dio ni jedne grane. Ukoliko ne pripazimo – lako ćemo te izmjene izgubiti. Trebamo li ih sačuvati, jednostavno kreiramo novu granu s `git branch` i repozitorij više neće biti u `detached HEAD` stanju, a izmjene koje smo napraviti su sačuvane u novoj grani.

Želimo li pogledati koje su se izmjene dogodile između sadašnjeg stanja grana i stanja od prije 10 *commit*ova, to će ići ovako:

```
git diff HEAD~10 HEAD
```

Notacija kojom dodajemo `~1`, `~2`, ... vrijedi i za reference na grane i na SHA1 identifikatore *commit*ova. Imate li granu `test` – već znamo da je to referenca samo na njen zadnji *commit*, a referenca na predzadnji je `test~1`. Analogno, `974ef0a~11` je referenca na 11-ti *commit* prije `974ef0ad8351ba7b4d402b8ae3942c96d667e199`.

.git direktorij

Pogledajmo na trenutak `.git` direktorij. Vjerojatno ste to već učinili, i vjerojatno ste otkrili da je njegov sadržaj otprilike ovakav:

```
$ ls .git
branches/
COMMIT_EDITMSG
config
description
HEAD
hooks/
index
info/
logs/
objects/
refs/
```

Ukratko ćemo ovdje opisati neke važne dijelove: `.git/config`, `.git/objects`, `.git/refs`, `HEAD` i `.git/hooks`

.git/config

U datoteci `.git/config` se spremaju sve lokalne postavke. To jest, tu su sve one postavke koje smo snimili s `git config <naziv> <vrijednost>`, i to su konfiguracije

koje se odnose samo za tekući repozitorij. Za razliku od toga, **globalne** postavke (one koje se snime s `git config --global`) se snimaju u korisnikov direktorij u datoteci `~/.gitconfig`.

.git/objects

Sadržaj direktorija `.git/objects` izgleda ovako nekako:

```
$ find .git/objects
.git/objects/
.git/objects/d7
.git/objects/d7/3aabba2b969b2ff2cbff18f1dc4e254d2a2af3
.git/objects/fc
.git/objects/fc/b8e595cf8e2ff6007f0780774542b79044ed4d
.git/objects/33
.git/objects/33/39354cbbe6be2bc79d8a5b0c2a8b179febfd7d
.git/objects/9c
.git/objects/9c/55a9bbd5463627d9e05621e9d655e66f2acb98
.git/objects/2c
```

To je direktorij koji sadrži sve verzije svih datoteka i svih *commit*ova našeg projekta. Dakle, to git koristi umjesto onih višestrukih direktorija koje smo spomenuli u našem hipotetskom sustavu za verzioniranje na početku ovog poglavlja. Apsolutno sve se ovdje čuva.

Uočite, na primjer, datoteku `d7/3aabba2b969b2ff2cbff18f1dc4e254d2a2af`. Ona se odnosi na git objekt s referencom `d73aabba2b969b2ff2cbff18f1dc4e254d2a2af`. Sadržaj tog objekta se može pogledati koristeći `git cat-file <referenca>`. Na primjer, tip objekta se može pogledati s:

```
$ git cat-file -t d73aab
commit
```

...što znači da je to objekt tipa *commit*. Zamijenite li `-t` s `-p` dobiti ćete točan sadržaj te datoteke.

Postoje četiri vrste objekata: *commit*, *tag*, *tree* i *blob*. *Commit* i *tag* sadrže metapo-

datke vezane uz ono što im sam naziv kaže. *Blob* sadrži binarni sadržaj neke datoteke, dok je *tree* popis datoteka.

Poigrate li se malo s `git cat-file -p <referenca>` otkriti ćete da *commit* objekti sadrže:

- Referencu na prethodni *commit*,
- Referencu na *commit* grane koju smo *merge*ali (ako je dotični *commit* rezultat *merge*a),
- Datum i vrijeme kad je nastao,
- Autora,
- Referencu na jedan objekt tipa *tree* koji sadrži popis svih datoteka koje su sudjelovale u tom *commit*u.

Drugim riječima, tu imamo sve potrebno da bi znali od čega se *commit* sastojao i da bi znali gdje je njegovo mjesto na grafu povijesti projekta.

Stvar se može zakomplicirati kad broj objekata poraste. Tada se u jedan *blob* objekt može zapakirati više datoteka ili samo dijelova datoteka posebnim algoritmom za pakiranje (*pack*).

.git/refs

Bacimo pogled kako izgleda direktorij `.git/refs`:

```
$ find .git/refs/
.git/refs/
.git/refs/heads
.git/refs/heads/master
.git/refs/heads/test
.git/refs/tags
.git/refs/tags/test
.git/refs/remotes
.git/refs/remotes/puzz
.git/refs/remotes/puzz/master
.git/refs/remotes/github
.git/refs/remotes/github/master
```

Pogledajmo i sadržaj tih datoteka:

```
$ cat .git/refs/tags/test
d100b59e6e4a0fd3c3720fdfbdcc0bd4a6ead307
```

Svaka od tih datoteka sarži referencu na jedan od objekata iz `.git/objects`. Poigrajte se s `git cat-file` i otkriti ćete da su to uvijek *commit* objekti.

Zaključak se sam nameće – u `.git/refs` se nalaze reference na sve grane, tagove i grane udaljenih repozitorija koji se nalaze u `.git/objects`. To je implementacija one priče da je gitu grana samo referenca na njen zadnji *commit*.

HEAD

Datoteka `.git/HEAD` u stvari nije obična datoteka nego samo simbolički link na neku od datoteka unutar `git/refs`. I to na onu od tih datoteka koja sadrži referencu na granu u kojoj se trenutno nalazimo. Na primjer, u trenutku dok pišem ove retke `HEAD` je kod mene `refs/heads/master`, što znači da se moj repozitorij nalazi na `master` grani.

.git/hooks

Ovaj direktorij sadrži *shell* skripte koje želimo izvršiti u trenutku kad se dogode neki važni događaji na našem repozitoriju. Svaki git repozitorij već sadrži primjere takvih skripti s ekstenzijom `.sample`. Ukoliko taj sample maknete, one će se početi izvršavati na tim događajima (*event*ima).

Na primjer, želite li da se prije svakog *commit*a izvrše *unit* testovi i pošalje mejl s rezultatima, napraviti ćete skriptu `pre-commit` koja to radi.

Povijest

Već smo se upoznali s naredbom `git log` s kojom se može vidjeti povijest *commit*ova grane u kojoj se trenutno nalazimo, no ona nije dovoljna za proučavanje povijesti projekta. Posebno s git projektima, čija povijest zna biti dosta kompleksna (puno grana, *merge*anja, i sl.).

Sigurno će vam se desiti da želite vidjeti koje su se izmjene desile između predzadnjeg i pred-predzanjeg *commit*a ili da vratite neku datoteku u stanje kakvo je bilo prije mjesec dana ili da proučite tko je zadnji napravio izmjenu na trinaestoj liniji nekog programa ili tko je prvi uveo funkciju koja se naziva `get_image_x_size` u projektu... Čak i ako vam se neki od navedenih scenarija čine malo vjerojatnim, vjerujte mi, trebati će vam.

U ovom poglavlju ćemo proći neke često korištene naredbe za proučavanje povijesti projekta.

Diff

S `git diff` smo se već sreli. Ukoliko ju pozovemo bez ikakvog dodatnog argumenta, ona će nam ispisati razlike između radne verzije repozitorija (tj. stanja projekta kakvo je trenutno na našem računalu) i zadnje verzije u repozitoriju. Drugi način korištenja naredbe je da provjeravamo razlike između dva *commit*a ili dvije grane (podsjetimo se da su grane u biti samo reference na zadnje *commit*ove). Na primjer:

```
git diff master testna-grana
```

...će nam ispisati razliku između te dvije grane. Treba paziti na redoslijed jer je ovdje bitan. Ukoliko isprobamo s:

```
git diff testna-grana master
```

... dobiti ćemo suprotan ispis. Ako smo u `testna-grana` jedan redak dodali – u jednom slučaju će `diff` ispisati kao da je **dodan**, a u drugom kao da je **obrisan**.

Treba imati na umu da `diff` ne prikazuje ono što ćemo dobiti *merge*anjem dvije grane – on samo prikazuje razlike. Iz tih razlika ne možemo vidjeti kako su nastale. Ne vidimo, na primjer, koja je grana imala više *commit*ova ili kako je došlo do razlika koje nam `diff` prikazuje. Za to je bolje koristiti naredbu `gitk`, koja će biti detaljno objašnjena kasnije.

Želimo li provjeriti koje su izmjene dogodile između predzadnjeg i pred-predzadnjeg *commit*a:

```
git diff HEAD~2 HEAD~1
```

... ili između pred-predzadnjeg i sadašnjeg:

```
git diff HEAD~2
```

... ili izmjene između `974ef0ad8351ba7b4d402b8ae3942c96d667e199` i `testna-grana`:

```
git diff 974ef testna-grana
```

Log

Standardno s naredbom `git log <naziv_grane>` dobiti ćemo kratak ispis povijesti te grane. Sad kad znamo da je grana u biti samo referenca na zadnji *commit*, znamo i da bi bilo preciznije kazati da je ispravna sintaksa `git log <referenca_na_commit>`. Za git nije previše bitno jeste li mu dali naziv grane ili referencu na *commit* – naziv grane je ionako samo alias za zadnji *commit* u toj grani. Ukoliko mu damo neki proizvoljan *commit*, on će jednostavno krenuti "unazad" po grafu i dati vam povijest koju na taj način nađe.

Dakle, ako želimo povijest trenutne grane, ali bez zadnjih pet unosa – treba nam referenca na peti *commit* unazad:

```
git log HEAD~5
```

Ili, ako želimo povijest grane `testna-grana` bez zadnjih 10 unosa:

```
git log testna-grana~10
```

Želimo li povijest *samo* nekoliko zadnjih unosa, koristimo `git log -<broj_ispisa>` sintaksu. Na primjer, ako nam treba samo 10 rezultata iz povijesti:

```
git log -10 testna-grana
```

. . . ili, ako to želimo za trenutnu granu, jednostavno:

```
git log -10
```

Whatchanged

Naredba `git whatchanged` je vrlo slična `git log`, jedino što uz svaki *commit* ispisuje i popis svih datoteka koje su se tada promijenile:

```
$ git whatchanged
commit 64fe180debf933d6023f8256cc72ac663a99fada
Author: Tomo Krajina <tkrajina@gmail.com>
Date:   Sun Apr 8 07:14:50 2012 +0200

    .git direktorij

:000000 100644 0000000... 7d07a9e... A        git_output/cat_git_refs_tag.txt
:100644 100644 522e7f3... 2a06aa2... M        git_output/find_dot_git_refs.txt
:100755 100755 eeb7fca... d66be27... M        ispod-haube.tex

commit bb1ed3e8a47c2d34644efa7d36ea5aa55efc40ea
Author: Tomo Krajina <tkrajina@gmail.com>
Date:   Sat Apr 7 22:05:55 2012 +0200

    git objects, nastavak

:000000 100644 0000000... fabdc91... A        git_output/git_cat_file_type.txt
:100755 100755 573cffc... eeb7fca... M        ispod-haube.tex
```

Pretraživanje povijesti

Vrlo često će vam se dogoditi da tražite određeni *commit* iz povijesti po nekom kriteriju. U nastavku ćemo obraditi dva najčešća načina pretraživanja. Prvi je kad pretražujemo prema tekstu komentara uz *commit*ove, tada se koristi `git log --grep=<regularni_izraz>`. Na primjer, tražimo li sve *commit*ove koji **u *commit* komentarima sadrže riječ** `graph`:

```
git log --grep=graph
```

Drugi česti scenarij je odgovor na pitanje "Kad se **u kodu** prvi put spomenuo neki string?". Tada se koristi `git log -S<string>`. Dakle, ne u komentaru *commit*a nego **u sadržaju datoteka**. Recimo da tražimo tko je prvi napisao funkciju `get_image_x_size`:

```
git log -Sget_image_x_size
```

Treba li pretraživati za string s razmacima:

```
git log -S"get image width"
```

Zapamtite, ovo će vam samo naći *commit*ove. Kad ih nađemo, htjeti ćemo vjerojatno pogledati koje su točno bile izmjene. Ako nam pretraživanje nađe da je *commit* `76cf802d23834bc74473370ca81993c5b07c2e35`, detalji izmjena koje su se njime dogodile su:

```
git diff 76cf8~1 76cf8
```

Podsjetimo se `76cf8` je kratica za *commit*, a `76cf8~1` je referenca na njegovog **prethodnika** (zbog `~1`).

Drugi način kako pogledati što se točno desilo u određenom *commit*u je:

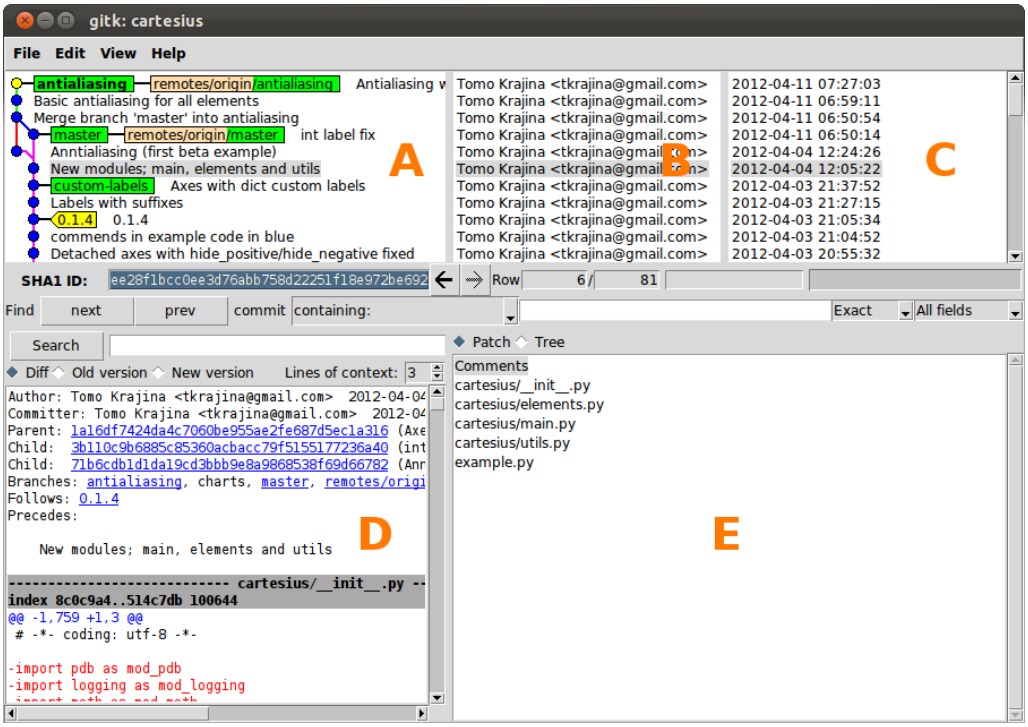
```
gitk 76cf8
```

Gitk

U standardnoj instalaciji gita dolazi i pomoćni programčić koji nam grafički prikazuje povijest trenutne grane. Pokreće se s:

```
gitk
```

. . . a izgleda ovako:



Sučelje ima pet osnovnih dijelova:

- grafički prikaz povijesti grane i *commit*ova iz drugih grana koji su imali veze s tom granom, bilo zbog grananja ili *merge*anja (označeno s A),
- popis ljudi koji su to *commit*ali (B),
- datum i vrijeme *commit*anja (C),
- pregled svih izmjena u odabranom *commit*u (D),
- pregled datoteka koje su tada izmijenjene (E).

Kad odaberete *commit* dobiti ćete sve izmjene i datoteke koje su sudjelovale u izmjenama. Kad odaberete datoteku, `gitk` će u dijelu sa izmjenama "skočiti" na izmjene koje su se dogodile na toj datoteci.

Gitk vam može i pregledati povijest samo do nekog *commit*a u povijesti:

```
gitk HEAD~10
```

...ili:

```
gitk 974ef0ad8351ba7b4d402b8ae3942c96d667e199
```

...ili određene grane:

```
gitk testna-grana
```

Gitk prikazuje povijest samo trenutne grane, odnosno samo onih *commit*ova koji su na neki način sudjelovali u njoj. Želimo li povijest svih grana – treba ga pokrenuti s:

```
gitk --all
```

Možemo dobiti i prikaz graf sa samo određenim granama:

```
gitk grana1 grana2
```

...i to je dobro koristiti u kombinanciji s `git diff grana1 grana2`. Jer, kao što smo već spomenuli, `diff` nam daje samo informaciju o razlikama između dvije grane, ali ne i njihove povijesti.

`gitk` ima puno korisnih opcija. Jedna je mogućnost da $diff$ (tj. razlike između *commit*ova) ne prikazuje linijama nego dijelovima linija. Na taj način možete točno vidjeti koju ste riječ izmijenili umjesto standardnog prikaza u kojem se vidi da je izmijenjena cijela linija. To ćete dobiti tako da odaberete `Markup words` ili `Color words` u izborniku iznad D (kad tek otvorite, postavljeno je na `Line diff`).

Kao `git gui` vjerojatno će vam se i `gitk` na prvi pogled učiniti neintuitivan, ali brz je i praktičan za rad kad se naučite na njegovo sučelje. S njime možete vrlo brzo pogledati tko je, što i kada *commit*ao u vašem projektu.

Blame

S `git blame <datoteka>` ćemo dobiti ispis neke datoteke s detaljima o tome **tko**, **kad** i u **kojem *commitu*** je napravio (ili izmijenio) pojedinu liniju u toj datoteci[30]:

```
$ git blame __init__.py
^96f3f2d (Tomo Krajina 2012-03-01 21:55:16 +0100   1)
#!/usr/bin/python
^96f3f2d (Tomo Krajina 2012-03-01 21:55:16 +0100   2) # -*- coding:
utf-8 -*-
^96f3f2d (Tomo Krajina 2012-03-01 21:55:16 +0100   3)
^96f3f2d (Tomo Krajina 2012-03-01 21:55:16 +0100   4) import logging
as mod_logging
^96f3f2d (Tomo Krajina 2012-03-01 21:55:16 +0100   5)
356f62d9 (Tomo Krajina 2012-03-03 06:13:44 +0100   6) ROW_COLUMN_SIZE
= 400
356f62d9 (Tomo Krajina 2012-03-03 06:13:44 +0100   7) NODE_RADIUS =
100
...
```

Nađete li liniju koja započinje znakom `^` – to znači da je ta linija bila tu i u prvoj verziji datoteke.

U svakom projektu datoteke mijenjaju imena. Tako da kod koji je pisan u jednoj datoteci ponekad završi u datoteci nekog trećeg imena. Ukoliko želimo znati i u kojoj su datoteci linije naše trenutke datoteke prvi put pojavile, to se može s:

```
git blame -C <datoteka>
```

Digresija o premještanju datoteka

Vezano uz `git blame -C`, napraviti ćemo jednu malu digresiju. Pretpostavimo da Mujo i Haso zajedno pišu zbirku pjesama. Svaku pjesmu spreme u posebnu datoteku, a da bi lakše pratili tko je napisao koju pjesmu – koriste git. Mujo je napisao pjesmu

[30] Nažalost, linije su preduge da bi se ovdje ispravno vidjelo.

`proljeće.txt`. Nakon toga je Haso tu datoteku preimenovao u `proljeće-u-mom-gradu.txt`. U trenutku kad je Haso *commit*ao svoju izmjenu – sustav za verzioniranje je te izmjene vidio kao da je `proljeće.txt` obrisana, a nova pjesma `proljeće-u-mom-gradu.txt` dodana.

Problem je što je novu datoteku `proljeće-u-mom-gradu.txt` dodao Haso. Ispada kao da je on **napisao** tu pjesmu, iako je samo preimenovao datoteku.

Zbog takvih situacija neki sustavi za verzioniranje (kao subversion ili mercurial) zahtijevaju od korisnika da **preko njihovih naredbi** radi preimenovanje datoteke ili njeno micanje u neki drugi repozitorij[31]. Tako oni znaju da je Haso **preimenovao** postojeću datoteku, ali sadržaj je napisao Mujo. Ukoliko to ne učinite preko njegovih naredbi – nego datoteke preimenujete standardnim putem (naredbe `mv` ili `move`) – sustav za verzioniranje neće imati informaciju o tome da je preimenovana.

To je problem, jer ljudi to često zaborave, a kad se to desi – gubi se povijest **sadržaja** datoteke.

Kad to zaboravite – problem je i kod *merge*a. Ako je u jednoj grani datoteka **preimenovana** a u drugoj samo **izmijenjena** – sustav neće znati da se u stvari radi o istoj datoteci koja je promijenjena, on će to percipirati kao dvije različite datoteke. Rezultat *merge*a može biti neočekivan.

Spomenuto nije problem i u gitu. Git ima ugrađenu heuristiku koja prati je li datoteka u nekom *commit*u preimenovana. Ukoliko u novom *commit*u on nađe da je jedna datoteka **obrisana** – proučiti će koje datoteke su u istom *commit*u **nastale**. Ako se sadržaj poklapa u dovoljno velikom postotku linija, git će sam zaključiti da se radi o preimenovanju datoteke – a ne o datoteci koja se prvi put pojavljuje u repozitoriju. Isto vrijedi ako datoteku nismo preimenovali nego premjestili (i, eventualno, preimenovali) na novu lokaciju u repozitorij.

To je princip na osnovu kojeg `git blame -C` "zna" u kojoj datoteci se neka linija prvi put pojavila. Zato bez straha možemo koristiti naredbe `mv`, `move` ili manipulirati ih preko nekog IDE-a.

Nemojte da vas zbuni to što git **ima** naredbu:

```
git mv <stara_datoteka> <nova_datoteka>
```

[31] Na primjer, u mercuriralu morate upisati `hg mv početna_datoteka krajnja_datoteka`, a ni slučajno `mv početna_datoteka krajnja_datoteka` ili `move početna_datoteka krajnja_datoteka`

... za preimenovanje/micanje datoteka. Ta naredba ne radi ništa posebno **u gitu**, ona je samo *alias* za standardnu naredbu operacijskog sustava (`mv` ili `move`).

Zato git ispravno *merge*a čak i ako u jednoj grani datoteku preimenujemo i izmijenimo, a u drugoj samo izmijenimo – git će sam zaključiti da se radi o istoj datoteci različitog imena.

Preuzimanje datoteke iz povijesti

Svima nam se dogodilo da smo izmijenili datoteku i kasnije shvatili da ta izmjena ne valja. Na primjer, zaključili smo da je verzija od prije 5 *commit*ova bila bolja nego ova trenutno. Kako da ju vratimo iz povijesti i *commit*amo u novo stanje projekta?

Znamo već da s `git checkout <naziv_grane> -- <datoteka1> <datoteka2>...` možemo lokalno dobiti stanje datoteka iz te grane. Odnedavno znamo i da naziv grane nije ništa drugo nego referenca na njen zadnji *commit*. Ako umjesto grane, tamo stavimo referencu na neki drugi *commit* dobiti ćemo stanje datoteka iz tog trenutka u povijesti.

Dakle, `git checkout` se, osim za prebacivanje s grane na granu, može koristiti i za preuzimanje neke datoteke iz prošlosti:

```
git checkout HEAD~5 -- pjesma.txt
```

To će nam u trenutni direktorij vratiti točno stanje datoteke `pjesma.txt` od prije 5 *commit*ova. I sad smo slobodni tu datoteku opet *commit*ati ili ju promijeniti i *commit*ati.

Treba li nam ta datoteka kakva je bila u predzadnjem *commit*u grane `test`?

```
git checkout test~1 -- pjesma.txt
```

Isto je tako i s bilo kojom drugom referencom na neki *commit* iz povijesti.

"Teleportiranje" u povijest

Isto razmatranje kao u prethodnom odjeljku vrijedi i za vraćanje stanja cijelog repozitorija u neki trenutak iz prošlosti.

Na primjer, otkrili smo bug, ne znamo gdje točno u kodu, ali znamo da se taj bug nije prije manifestirao. Međutim, ne znamo točno **kada** je bug zepočeo.

Bilo bi zgodno kad bismo cijeli projekt mogli teleportirati na neko stanje kakvo je bilo prije *n commit*ova. Ako se tamo bug i dalje manifestira – vratiti ćemo se još malo dalje u povijest[32]. Ako se tamo manifestirao – prebaciti ćemo se u malo bližu povijest. I tako – mic po mic po povijesti, sve dok ne nađemo trenutak (*commit*) u povijesti repozitorija u kojem je bug stvoren. Ništa lakše:

```
git checkout HEAD~10
```

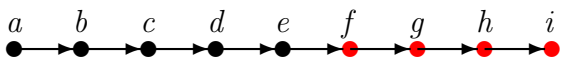
...i za čas imamo stanje kakvo je bilo prije 10 *commit*ova. Sad tu možemo s `git branch` kreirati novu granu ili vratiti se na najnovije stanje s `git checkout HEAD`. Ili možemo provjeriti je li bug i dalje tu. Ako jest, idemo probati s...

```
git checkout HEAD~20
```

...ako buga sad nemamo, znamo da se pojavio negdje između **dvadesetog** i **desetog** zadnjeg *commit*a. I tako, mic po mic, sve dok ne nađemo onaj trenutak u povijesti kad se greška pojavila.

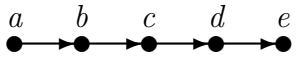
Reset

Uzmimo ovakvu hipotetsku situaciju: Radimo na nekoj grani, a u jednom trenutku stanje je:



I sad zaključimo kako tu nešto ne valja – svi ovi *commit*ovi od *f* pa na dalje su krenuli krivim smjerom. Htjeli bi se nekako vratiti na:

[32]Postoji i posebna naredba koja automatizira upravo taj proces traženja greške, a zove se *bisect*.



...i od tamo nastaviti cijeli posao, ali ovaj put drukčije. E sad, lako je "teleportirati se" s `git checkout ...`, to ne mijenja stanje grafa – f, g, h i i bi i dalje ostali u grafu. Mi bi ovdje htjeli baš obrisati dio grafa, odnosno zadnjih nekoliko *commit*ova.

Naravno da se i to može, a naredba je `git reset --hard <referenca_na_commit>`. Na primjer, ako se želimo vratiti na predzadnje stanje i potpuno obrisati zadnji *commit*:

```
git reset --hard HEAD~1
```

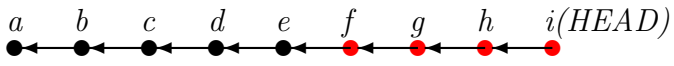
Želimo li se vratiti na `974ef0ad8351ba7b4d402b8ae3942c96d667e199` i maknuti sve *commit*ove koji su se desili nakon njega. Isto:

```
git reset --hard 974ef0a
```

Postoji i `git reset --soft <referenca>`. S tom varijantom se isto brišu *commit*ovi iz povijesti repozitorija, ali stanje datoteka u našem radnom direktoriju ostaje kakvo jest.

Cijela poanta naredbe `git reset` je da **pomiče** `HEAD`. Kao što znamo, `HEAD` je referenca na zadnji *commit* u trenutnoj grani. Za razliku od nje, kad se "vratimo u povijest" s `git checkout HEAD~2` – mi **nismo** dirali `HEAD`. Git i dalje zna koji mu je *commit* `HEAD` i, posljedično, i dalje zna kako izgleda cijela grana.

U našem primjeru od početka, mi želimo maknuti crvene *commit*ove. Ako prikažemo graf prema načinu kako git čuva podatke – svaki *commit* ima referencu na prethodnika, dakle strelice su suprotne od smjera nastajanja:



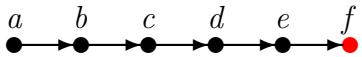
Uopće se ne trebamo truditi **brisati** čvorove/*commit*ove f, g, h i i. Dovoljno je reći "Pomakni `HEAD` tako da pokazuje na e. I to je upravo ono što git čini s `git reset`:

Iako su ovi čvorovi ovdje prikazani na grafu, git do njih više ne može doći. Da bi rekonstruirao svoj graf, on uvijek kreće od `HEAD`, dakle f, g, h i i su izgubljeni.

Revert

Svaki *commit* mijenja povijest projekta tako da izmjene **dodaje na kraju grane**. Treba imati na umu da `git reset` **mijenja postojeću povijest projekta**. Ne dodaje čvor na kraj grane, on mijenja postojeću granu tako da obriše nekoliko njegovih zadnjih čvorova. To može biti problem, posebno u situacijama kad radite s drugim ljudima, o tome više u sljedećem poglavlju.

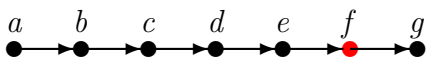
To se može riješiti s `git revert`. Uzmimo, na primjer, ovakvu situaciju:



Došli smo do zaključka da je *commit* f neispravan i treba vratiti na stanje kakvo je bilo u e. Znamo već da bi `git reset` jednostavno maknuo f s grafa, ali recimo da to ne želimo. Tada se može napraviti sljedeće `git revert <commit>`. Na primjer, ako želimo *revert*ati samo zadnji *commit*:

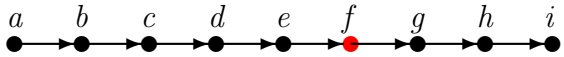
```
git revert HEAD
```

Rezultat će biti da je dosadašnji graf ostao potpuno isti, no **dodan je novi commit** koji miče sve izmjene koje su uvedene u f:



Dakle, stanje u g će biti isto kao i u e.

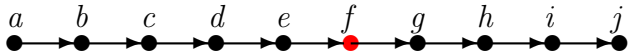
To se može i sa *commit*om koji nije zadnji u grani:



Ako je SHA1 referenca *commit*a f `402b8ae3942c96d667e199974ef0ad8351ba7b4d`, onda ćemo s:

```
git revert 402b8ae39
```

... dobiti:

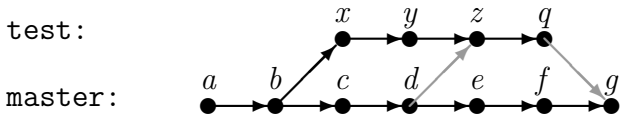


... gdje *commit* j ima stanje kao u i, ali bez izmjena koje su uvedene u f.

Naravno, ako malo o tome razmislite, složiti ćete se da sve to radi idealno samo ako g, h i i **nisu dirali isti kod kao i** f. `git revert` uredno radi ako revertamo *commit*ove koji su bliži kraju grane. Teško će, ako ikako, *revert*ati stvari koje smo mijenjali prije dvije godine u kodu koji je, nakon toga, puno puta bio mijenjan. Ukoliko to i pokušamo, git će javiti grešku i tražiti nas da mu sami damo do znanja kako bi *revert* trebao izgledati.

Izrazi s referencama

Imamo li ovakvu povijest:



Već znamo da je `HEAD` referenca na trenutni *commit*. Dakle, `HEAD` je ovdje isto što i `master`.

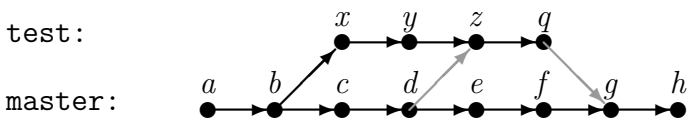
Znamo i da je `HEAD~1` referenca na predzadnji *commit* (f iz primjera), a `HEAD~2` referenca na pred-pred-zadnji (e).

Dakle, ovaj znak `~` je u principu operacija između reference na *commit* i nekog broja (n), a rezultat je *commit* koji se desio n koraka u povijesti. Te operacije možemo i konkatenirati, dakle `HEAD~2~3` je ekvivalentno `HEAD~5`.

Još jedna korisna operacija nad *commit*ovima je `^`. Za razliku od `~` koja ide na n-ti korak **prije** trenutnog, `^` nam daje n-tog **roditelja**.

U primjeru, *commit* g ima dva roditelja i zbog toga bi nam `HEAD^1` dalo referencu na f, a `HEAD^2` na q.

I ovdje, operacije možemo konkatenirati. Na primjer, u repozitoriju s poviješću...



...će biti...

- `master~1` je g,
- `master~1^1` je isto što i `master~2`, a to je f,
- `master~1^2` je isto što i `HEAD~1^2`, a to je q,
- itd.

I gdje god neka git naredba traži referencu na *commit* možete koristiti ovakve izraze umjesti SHA1 stringa. Npr:

```
gitk master~1^2
```

```
git cherry-pick master~3
```

```
git checkout ebab9a46829b8d19ebe1a826571e7803ae723c3b~1^2
```

```
git log HEAD^2
```

Često nam treba odgovor na pitanje "Što smo *commit*ali prije mjesec dana?" ili "Koji je bio zadnji *commit* prije tjedan dana?". Za to se koristi `@` s opisnom oznakom vremena, na primjer:

```
git log master@{"1 day ago"}
```

```
git log ebab9a46829b8d19ebe1a826571e7803ae723c3b@{"5 moths ago"}
```

```
git log branch@{2010-05-05}
```

U novijim verzijama gita i `HEAD` ima kraticu `@`, pa na primjer, umjesto `HEAD~10` možete pisati `@~10`.

Reflog

Reflog je povijest svih *commit*ova na koje je pokazivao `HEAD`. S naredbom:

```
git reflog
```

... ćete vidjete SHA1 identifikatore svih commitova na kojima je vaš repozitorij bio do sada.

Dakle, svaki put kad se prebacite na novu granu ili neki *commit*, git pamti gdje ste točno bili. Ukoliko ste neki *branch* obrisali, a kasnije shvatili da to niste htjeli – njegovi *commit*ovi su (vjerojatno![33]) još uvijek u lokalnom repozitoriju. S ovom naredbom ih

[33] Detalji u poglavlju o "higijeni" repozitorija.

možete naći i iz njih ponovno napraviti novu lokalnu granu.

Udaljeni repozitoriji

Sve ono što smo do sada proučavali smo radili isključivo na lokalnom repozitoriju. Samo smo spomenuli da je git distribuirani sustav za verzioniranje koda. Složiti ćete se da je već krajnje vrijeme da krenemo pomalo obrađivati interakciju s udaljenim repozitorijima.

Postoji puno situacija kako može funkcionirati ta interakcija. Moguće je da smo ga stvorili od nule s `git init`, na način kako smo to radili do sada i onda, na neki način, "poslali" na neku udaljenu lokaciju. Ili smo takav repozitorij ponudili drugim ljudima da ga preuzmu (kloniraju).

Moguće je i da smo saznali za neki već postojeći repozitorij, negdje na internetu, i sad želimo i mi preuzeti taj kod. Bilo da je to zato što želimo pomoći u razvoju ili samo proučiti nečiji kod.

Naziv i adresa repozitorija

Prvu stvar koju ćemo obraditi je kloniranje udaljenog repozitorija. Ima prije toga nešto što trebamo znati. Svaki udaljeni repozitorij s kojime će git "komunicirati" mora imati svoju adresu.

Na primjer, ova knjiga "postoji" na `git://github.com/tkrajina/uvod-u-git.git`, i to je jedna njena adresa. Github[34] omogućuje da se istim repozitoriju pristupi i preko `https://tkrajina@github.com/tkrajina/uvod-u-git.git`. Osim na Githubu, ona živi i na mom lokalnom računalu i u tom slučaju je njena adresa `/home/puzz/projects/uvod-u-git` (direktorij u kojemu se nalazi).

Svaki udaljeni repozitorij s kojime ćemo imati posla mora imati i svoje kratko ime (*alias*). Nešto kao: `origin` ili `vanjin-repozitorij` ili `slobodan` ili `dalenov-repo`. Nazivi su naš slobodan izbor. Tako, ako nas četvero radi na istom projektu, njihove

[34] Jedan od mnogih web servisa koji vam nude uslugu čuvanja (hostanja) git repozitorija.

udaljene repozitorije možemo nazvati `marina`, `ivan`, `karla`. I sa svakim od njih možemo imati nekakvu interakciju. Na neke ćemo slati svoje izmjene (ako imamo ovlasti), a s nekih ćemo izmjene preuzimati u svoj repozitorij.

Kloniranje repozitorija

Kloniranje je postupak kojim kopiramo cijeli repozitorij s udaljene lokacije na naše lokalno računalo. S tako kloniranim repozitorijem možemo nastaviti rad kao s repozitorijem kojeg smo inicirali lokalno.

Kopirati repozitorij je jednostavno, dovoljno je u neki direktorij kopirati `.git` direktorij drugog repozitorija. I onda na novoj (kopiranoj) lokaciji izvršiti `git checkout HEAD`.

Pravo kloniranje je za nijansu drukčije. Recimo to ovako, **kloniranje je kopiranje udaljenog repozitorija, ali tako da novi (lokalni) repozitorij ostaje "svjestan" da je on kopija nekog udaljenog repozitorija**. Klonirani repozitorij čuva informaciju o repozitoriju is kojeg je kloniran. Ta informacija će mu kasnije olakšati da na udaljeni repozitorij šalje svoje izmjene i da od njega preuzima izmjene.

Postupak je jednostavan. Moramo znati adresu udaljenog repozitorija, i tada će nam git s naredbom...

```
$ git clone git://github.com/tkrajina/uvod-u-git.git
Cloning into uvod-u-git...
remote: Counting objects: 643, done.
remote: Compressing objects: 100% (346/346), done.
remote: Total 643 (delta 384), reused 530 (delta 271)
Receiving objects: 100% (643/643), 337.00 KiB | 56 KiB/s, done.
Resolving deltas: 100% (384/384), done.
```

...**kopirati projekt, zajedno sa cijelom poviješću** na naše računalo. I to u direktorij `uvod-u-git`. Sad u njemu možemo gledati povijest, granati, *commit*ati, ... Ukratko, raditi što god nas je volja s tim projektom.

Jasno, ne može bilo tko kasnije svoje izmjene poslati nazad na originalnu lokaciju. Za to moramo imati ovlasti, ili moramo vlasnika tog repozitorija pitati je li voljan naše izmjene preuzeti kod sebe. O tome kasnije.

Sjećate se kad sam napisao da su nazivi udaljenih repozitorija vaš slobodan izbor?

Kloniranje je ipak izuzetak. Ukoliko kloniramo udaljeni repozitorij, on se za nas zove `origin`. Ostali repozitoriji koje ćemo dodavati mogu imati nazive kakve god želimo.

Struktura kloniranog repozitorija

Od trenutka kad smo klonirali svoj repozitorij pa dalje – za nas postoje **dva repozitorija**. Možda negdje na svijetu postoji još netko tko je klonirao taj isti repozitorij i na njemu nešto radi (a da mi o tome ništa ne znamo). Naš dio svijeta su samo ova dva s kojima direktno imamo posla. Jedan je udaljeni kojeg smo klonirali, a drugi je lokalni koji se nalazi pred nama.

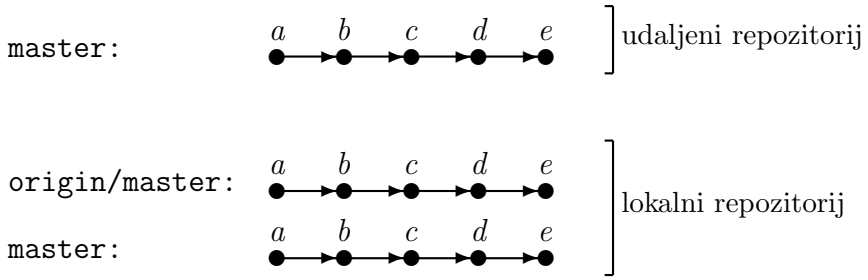
Prije nego li počnemo s pričom o tome kako slati ili primati izmjene iz jednog repozitorija u drugi, trebamo nakratko spomenuti kakva je točno struktura lokalnog repozitorija. Već znamo za naredbu `git branch`, koja nam ispisuje popis svih grana na našem repozitoriju. Sad imamo posla i s udaljenim repozitorijem – njega smo klonirali.

S `git branch -a` ispisujemo **sve grane koje su nam trenutno dostupne u lokalnom repozitoriju**. Naime, kad smo klonirali repozitorij – postale su nam dostupne i grane udaljenog repozitorija:

```
$ git branch -a
* master
  remotes/origin/master
```

Novost ovdje je `remotes/origin/master`. Ovo `remotes/` znači da, iako imamo pristup toj grani na lokalnom repozitoriju, ona je **samo kopija grane `master` u repozitoriju `origin`**. Takve kopije udaljenih repozitorija ćemo uvijek označavati s `<naziv_repozitorija>/<naziv_grane>`. Konkretno, ovdje je to `origin/master`.

Dakle, grafički bi to mogli prikazati ovako:



Imamo dva repozitorija – lokalni i udaljeni. Udaljeni ima samo granu `master`, a lokalni ima dvije kopije te grane. U lokalnom `master` ćemo mi *commit*ati naše izmjene, a u `origin/master` se nalazi kopija udaljenog `origin/master` u koju **nećemo** *commit*ati. Ovaj `origin/master` ćemo, s vremenena na vrijeme, osvježavati tako da imamo ažurno stanje (sliku) udaljenog repozitorija.

Ako vam ovo zvuči zbunjujuće – nemojte se zabrinuti. Sve će sjesti na svoje mjesto kad to počnete koristiti.

Djelomično kloniranje povijesti repozitorija

Recimo da ste na internetu našli zanimljiv *opensource* projekt i sad želite klonirati njegov git repozitorij da bi proučiti kod. Ništa lakše; `git clone ...`.

E, ali... Tu imamo mali problem. Git repozitorij sadrži cijelu povijest projekta. To znači da sadrži sve *commit*ove koje su radili programeri i koji mogu sezati i preko deset godina unazad[35]. I zato `git clone` ponekad može potrajati dosta dugo. Posebno ako imate sporu vezu.

No, postoji trik. Želimo li skinuti projekt samo zato da bi pogledali njegov kod, a ne zanima nas cijela povijest, moguće je klonirati samo nekoliko zadnjih *commit*ova s:

```
git clone --depth 5 --no-hardlinks git://github.com/tkrajina/uvod-u-git.git
```

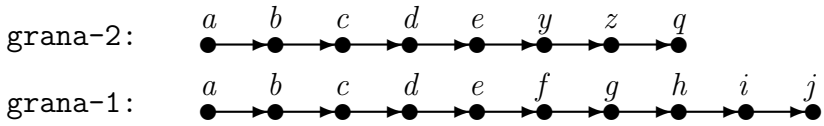
To će biti puno brže, no s takvim klonom nemamo pristup cijeloj povijesti i ne bi

[35] Ako ste zbunjeni kako je moguće da *commit*ovi u git repozitoriju sežu dulje u prošlost negoli uopće postoji git – radi se repozitorijima koji su prije bili na subversionu ili CVS-u, a koji su kasnije konvertirani u git tako da su svi *commit*ovi sačuvani.

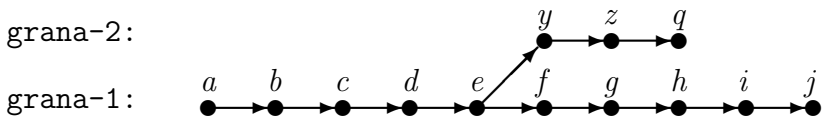
mogli raditi sve ono što teorijski možeomo s pravim klonom. Djelomično kloniranje je zamišljeno da bismo skinuli kod nekog projekta samo da ga proučimo, a ne da bi se na njemu nešto ozbiljno radilo.

Digresija o grafovima, repozitorijima i granama

Nastaviti ćemo još jednu digresiju važnu za razumijevanje grafova projekata i udaljenih projekata koji će slijediti. Radi se o tome da je ovaj graf...

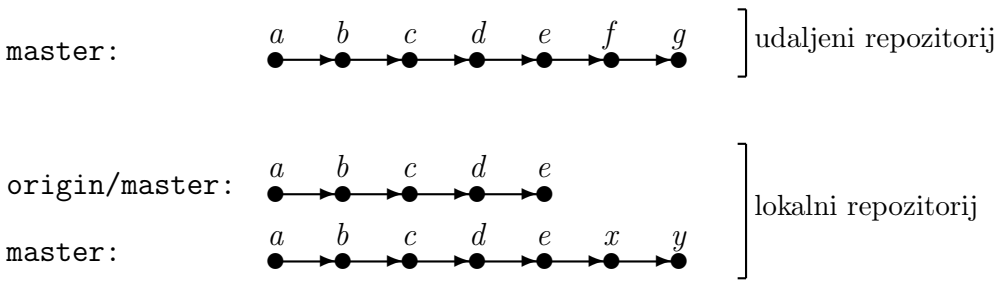


...ekvivalentan grafu...



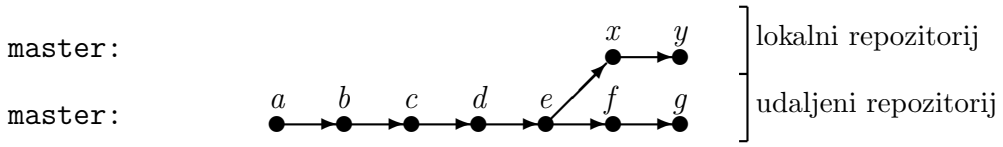
...zato što ima zajednički početak povijesti (a, b, c, d i e).

Slično, u sljedećoj situaciji:



...trebamo zamisliti da je odnos između našeg lokalnog `master` i udaljenog `master` – kao da imamo jedan graf koji se granao u čvoru e. Jedino što se svaka grana nalazi u

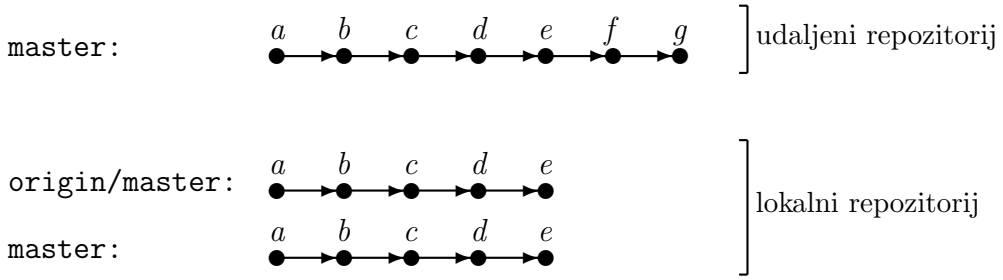
zasebnom repozitoriju, a ne više u istom. Zanemarimo li na trenutak `origin/master`, odnos između naša dva `master`a je:



U ilustracijama koje slijede, grafovi su prikazani kao zasebne "crte" samo zato što logički pripadaju posebnim entitetima (repozitorijima). Međutim, oni **su samo posebne grane istog projekta** iako se nalaze na različitim lokacijama/računalima.

Fetch

Što ako je vlasnik udaljenog repozitorija *commit*ao u svoj `master`? Stanje bi bilo ovakvo:



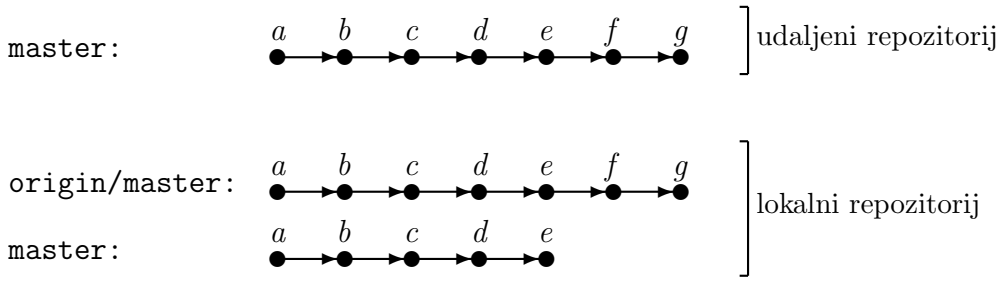
Naš, lokalni, `master` je radna verzija s kojom ćemo mi raditi tj. na koju ćemo *commit*ati. `origin/master` bi trebao biti lokalna kopija udaljenog `master`, međutim – ona se ne ažurira automatski. To što je vlasnik udaljenog repozitorija dodao dva *commit*a (e i f) ne znači da će naš repozitorij nekom čarolijom to odmah saznati.

Git je zamišljen kao sustav koji ne zahtijeva stalni pristup internetu. U većini operacija – **od nas se očekuje da iniciramo interakciju s drugim repozitorijima**. Bez da mi pokrenemo neku radnju, git neće nikad kontaktirati udaljene repozitorije. Slično, drugi repozitorij ne može našeg natjerati da osvježi svoju sliku (odnosno `origin/` grane). Najviše što vlasnik udaljenog repozitorija može napraviti je da nas **zamoli** da to učinimo.

Kao što smo mi inicirali kloniranje, tako i mi moramo inicirati ažuriranje grane `origin/master`. To se radi s `git fetch`:

```
$ git fetch
remote: Counting objects: 5678, done.
remote: Compressing objects: 100% (1967/1967), done.
remote: Total 5434 (delta 3883), reused 4967 (delta 3465)
Receiving objects: 100% (5434/5434), 1.86 MiB | 561 KiB/s, done.
Resolving deltas: 100% (3883/3883), completed with 120 local objects.
From git://github.com/twitter/bootstrap
```

Nakon toga, stanje naših repozitorija je:



Dakle, `origin/master` je osvježen tako da mu je stanje isto kao i `master` udaljenog repozitorija.

S `origin/master` možemo raditi skoro sve kao i s ostalim lokalnim granama. Možemo, na primjer, pogledati njegovu povijest s:

```
git log origin/master
```

Možemo pogledati razlike između njega i naše trenutne grane:

```
git diff origin/master
```

Možemo se prebaciti na `origin/master`, ali...

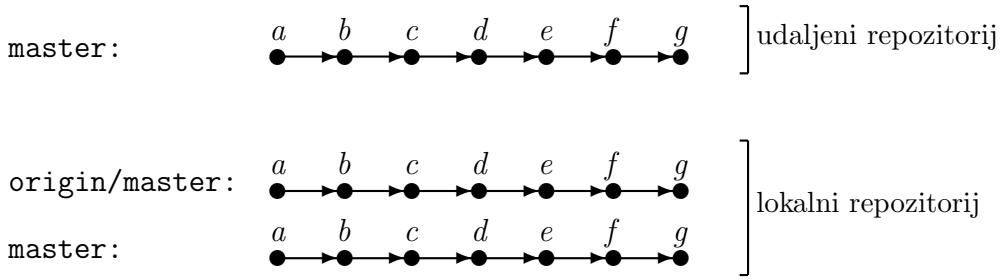
```
$ git checkout origin/master
Note: checking out 'origin/master'.

You are in 'detached HEAD' state. You can look around, make
experimental
changes and commit them, and you can discard any commits you make in
this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you
may
do so (now or later) by using -b with the checkout command again.
Example:

  git checkout -b new_branch_name

HEAD is now at 167546e... Testni commit
```

Git nam ovdje dopušta prebacivanje na `origin/master`, ali nam jasno daje do znanja da je ta grana ipak po nečemu posebna. Kao što već znamo, ona nije zamišljena da s njome radimo direktno. Nju možemo samo osvježavati stanjem iz udaljenog repozitorija. U `origin/master` ne bi smjeli *commit*ati.

Ima, ipak, jedna radnja koju trebamo raditi s `origin/master`, a to je da izmjene iz nje preuzimamo u naš lokalni `master`. Prebacimo se na njega s `git checkout master` i...

```
git merge origin/master
```

...i sad je stanje:



I sad smo tek u `master` dobili stanje udaljenog `master`. Općenito, to je postupak kojeg ćemo često ponavljati:

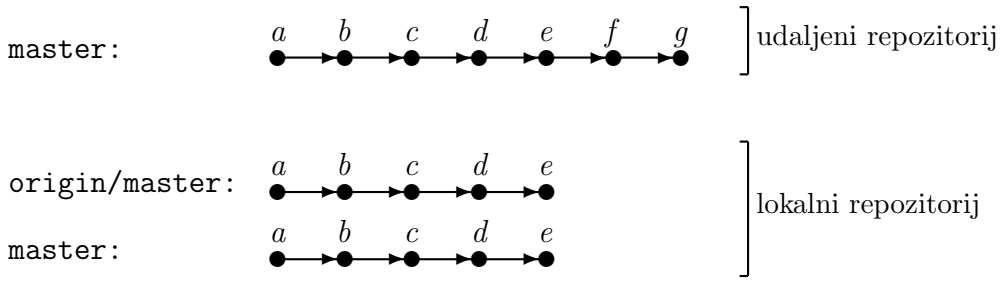
```
git fetch
```

... da bismo osvježili svoj lokalni `origin/master`. Sad tu možemo malo proučiti njegovu povijest i izmjene koje uvodi u povijest. I onda...

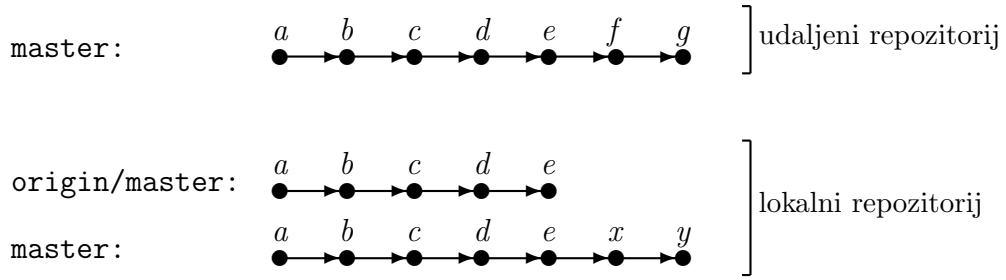
```
git merge origin/master
```

... da bi te izmjene unijeli u naš lokalni repozitorij.

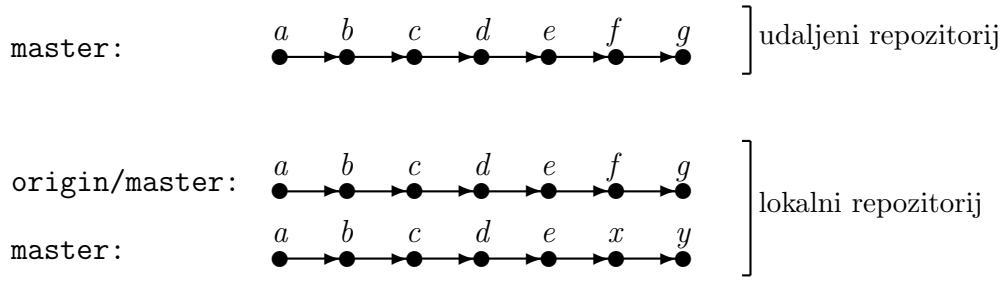
Malo složenija situacija je sljedeća: recimo da smo nakon...



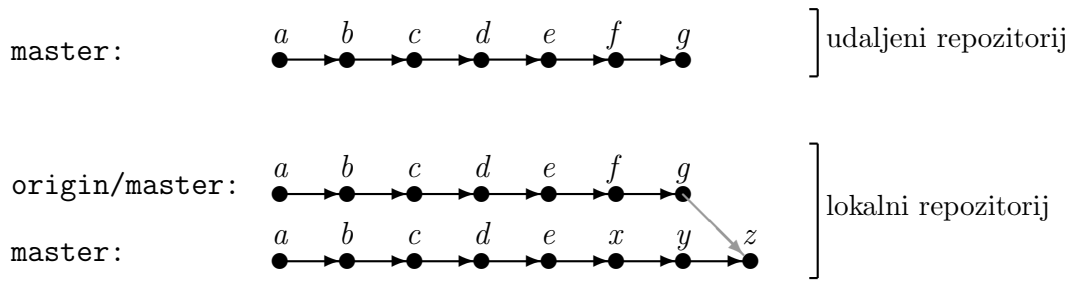
... mi *commit*ali u naš lokalni repozitorij svoje izmjene. Recimo da su to x i y:



Nakon `git fetch`, stanje je:



Sad `origin/master` nakon e ima f i g, a `master` nakon e ima x i y. U biti je to kao da imamo dvije grane koje su nastale nakon e. Jedna ima f i g, a druga x i y. Ovo je, u biti, najobičniji *merge* koji će, eventualno, imati i neke konflikte koje znamo riješiti. Rezultat *merge*a je novi čvor z:



Pull

U prethodnom poglavlju smo opisali tipični redoslijed naredbi koje ćemo izvršiti svaki put kad želimo preuzeti izmjene iz udaljenog repozitorija:

```
git fetch
git merge origin/master
```

Obično ćemo nakon `git fetch` malo pogledati koje izmjene su došle s udaljenog repozitorija, no u konačnici ćemo ove dvije naredbe skoro uvijek izvršiti u tom redoslijedu.

Zbog toga postoji "kratica" koja je ekvivalentna toj kombinaciji:

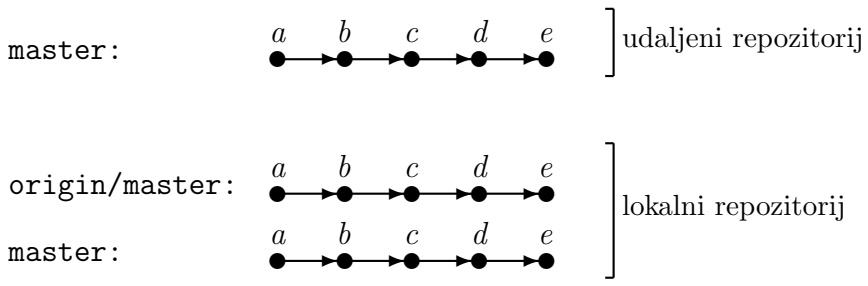
```
git pull
```

`git pull` je upravo kombinacija `git fetch` i `git merge origin/master`.

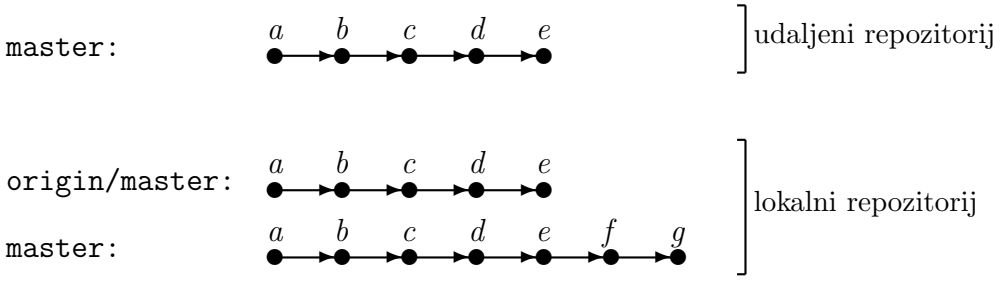
Push

Sve što smo do sada radili s gitom su bile radnje koje smo mi radili na našem lokalnom repozitoriju. Čak i kloniranje je nešto što mi iniciramo i ničim nismo promijenili udaljeni repozitorij. Krenimo sad s prvom radnjom s kojom aktivno mijenjamo neki udaljeni repozitorij.

Uzmimo, kao prvo, najjednostavniji mogući scenarij. Klonirali smo repozitorij i stanje je, naravno:



Nakon toga smo *commit*ali par izmjena...



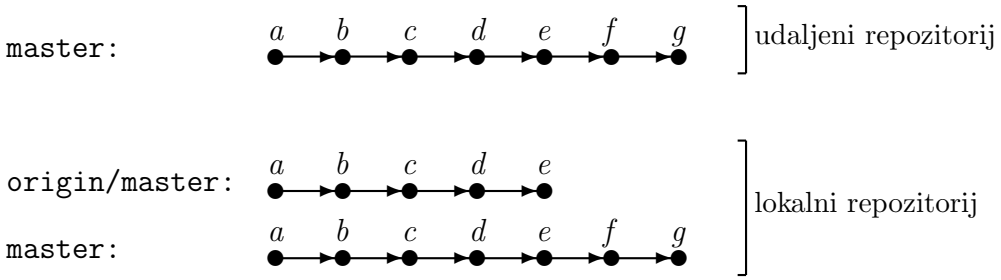
...i sad bismo htjeli te izmjene "prebaciti" na udaljeni repozitorij. Prvo i osnovno što nam treba svima biti jasno – "prebacivanje" naših lokalnih izmjena na udaljeni repozitorij ovisi o tome imamo li ovlasti za to ili ne. **Udaljeni repozitorij mora biti tako konfiguriran da bismo mogli raditi** `git push`.

Ukoliko nemamo ovlasti, sve što možemo napraviti je zamoliti njegovog vlasnika da pogleda naše izmjene (f i g) i da ih preuzme kod sebe, ako mu odgovaraju. Taj process se zove ***pull request*** iliti zahtjev za *pull* s njegove strane.

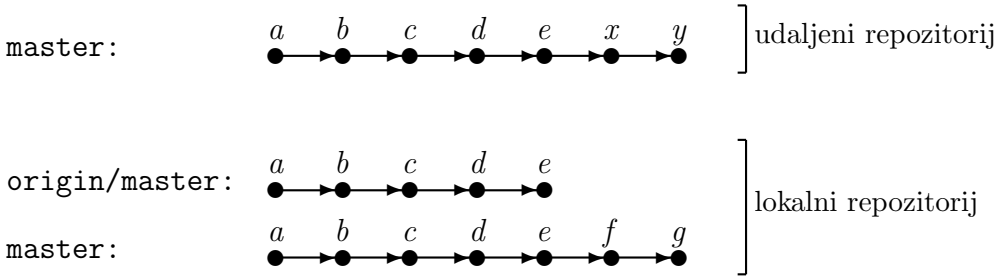
Ukoliko imamo ovlasti onda je ono što treba napraviti:

```
$ git push origin master
Counting objects: 11, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (9/9), done.
Writing objects: 100% (9/9), 1.45 KiB, done.
Total 9 (delta 6), reused 0 (delta 0)
To git@github.com:tkrajina/uvod-u-git.git
   0335d78..63ced90  master -> master
```

Stanje će sad biti:



To je bila situacija u kojoj smo u našem `master` *commit*ali, ali na udaljeni `master` nije nitko drugi *commit*ao. Što da nije tako? Dakle, dok smo mi radili na e i f, vlasnik udaljenog repozitorija je *commit*ao svoje x i y:



Kad pokušamo `git push origin master`, dogoditi će se ovakvo nešto:

```
$ git push origin master
To git@domena.com:repozitorij
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to 'git@domena.com:repozitorij'
To prevent you from losing history, non-fast-forward updates were
rejected
Merge the remote changes (e.g. 'git pull') before pushing again.  See
the
'Note about fast-forwards' section of 'git push --help' for details.
```

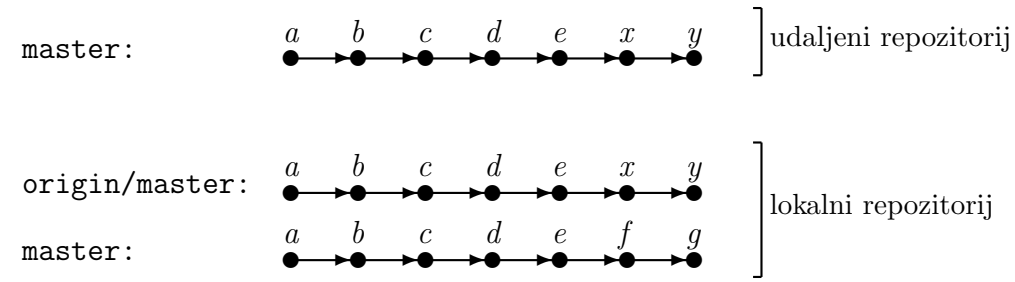
Kod nas lokalno e slijede f i g, a na udaljenom repozitoriju e slijede x i y. I git ovdje ne zna što točno napraviti, i traži da mu netko pomogne. Kao i do sada, pomoć se očekuje od nas, a tu radnju trebamo izvršiti na lokalnom repozitoriju. Tek onda ćemo

moći *push*ati na udaljeni.

Rješenje je standardno:

```
git fetch
```

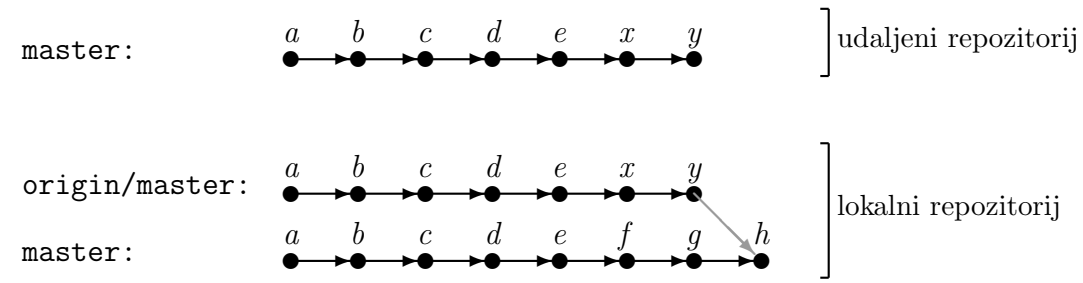
...i stanje je sad:



Sad ćemo, naravno:

```
git merge origin/master
```

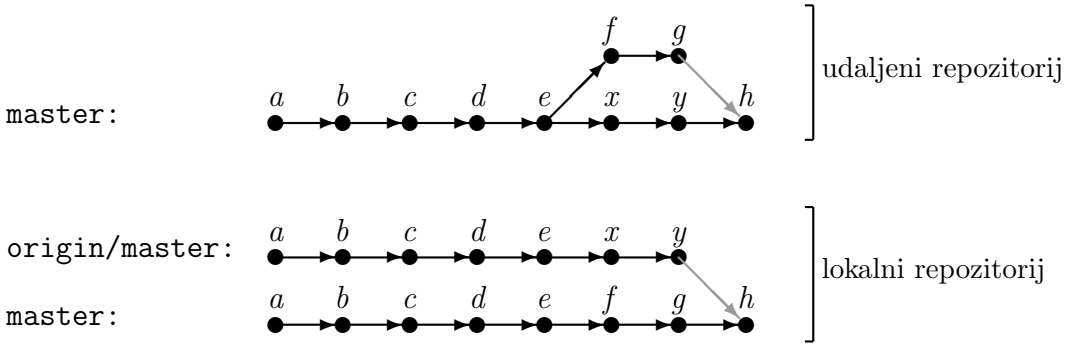
Na ovom koraku se može desiti da imate konflikata koje eventualno treba ispraviti s `git mergetool`. Nakon toga, stanje je:



Sad možemo uredno napraviti:

```
git push origin master
```

. . . a stanje naših repozitorija će biti:



Naše izmjene se sad nalaze na udaljenom repozitoriju.

Push tagova

Naredba `git push origin master` šalje na udaljeni (`origin`) repozitorij samo izmjene u grani `master`. Slično bi napravili s bilo kojom drugom granom, no ima još nešto što ponekad želimo s lokalnog repozitorija poslati na udaljeni. To su tagovi.

Ukoliko imamo lokalni tag kojeg treba *push*ati, to se radi s:

```
git push origin --tags
```

To će na udaljeni repozitorij poslati sve tagove. Želimo li tamo obrisati neki tag:

```
git push origin :refs/tags/moj-tag
```

Treba samo imati na umu da je moguće da su drugi korisnici istog udaljenog repozitorija možda već *fetch*ali naš tag u svoje repozitorije. Ukoliko smo ga mi tada obrisali, nastati će komplikacije.

Treba zato pripaziti da se *push*aju samo tagovi koji su sigurno ispravni.

Rebase origin/master

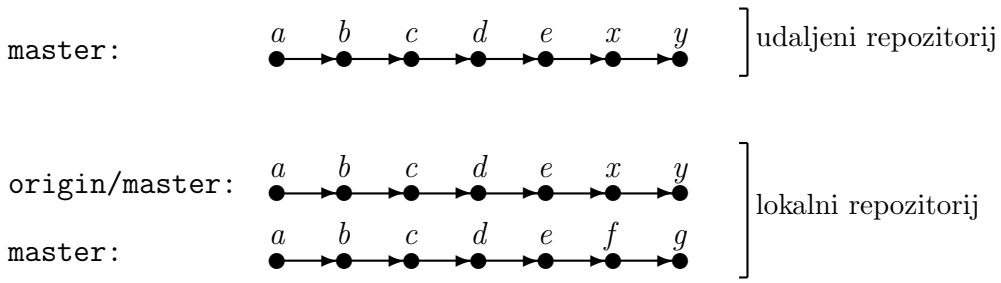
Želimo li da se naši f i g (iz prethodnog primjera) vide u povijesti udaljenog projekta kao zasebni čvorovi – i to se može s:

```
git checkout master
git rebase origin/master
git push origin master
```

Ukoliko vam se ovo čini zbunjujuće – još jednom dobro proučite "Digresiju o grafovima" koja se nalazi par stranica unazad.

Prisilan *push*

Vratimo se na ovu situaciju:

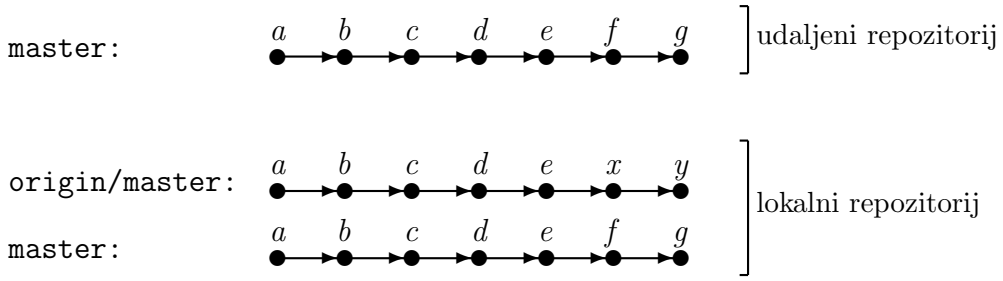


Standardni postupak je `git fetch` (što je u gornjem primjeru već učinjeno) i `git merge origin/master`. Međutim, postoji još jedna mogućnost. Nakon što smo proučili ono što je vlasnik udaljenog repozitorija napravio u *commit*ovima x i y, ponekad ćemo zaključiti da to jednostavno ne valja. I najradije bi sada "pregazili" njegove *commit*ove s našim.

To se može, naredba je:

```
git push -f origin master
```

... a rezultat će biti:



I sad s `git fetch` možemo još i osvježiti stanje `origin/master`.

Treba, međutim, imati na umu da ovakvo ponašanje nije baš uvijek poželjno. Zbog dva razloga:

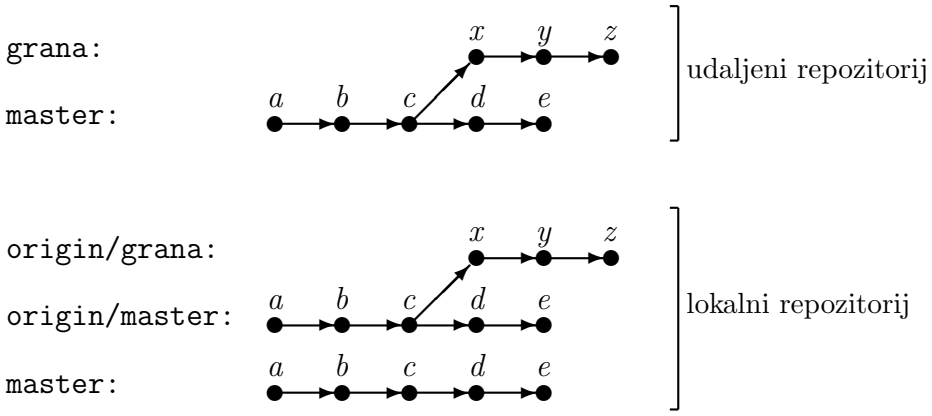
- **Mi** smo zaključili da *commit*ovi x i y ne valjaju. Možda smo pogriješili. Koliko god nam to bilo teško priznati, sasvim moguće je da jednostavno nismo dobro shvatili tuđi kod.

- Nitko ne voli da mu se pregaze njegove izmjene kao što smo to mi ovdje napravili vlasniku ovog udaljenog repozitorija. Bilo bi bolje javiti se njemu, objasniti mu što ne valja, predložiti bolje rješenje i dogovoriti da **on** *revert*a, *reset*ira ili ispravi svoje izmjene.

Rad s granama

Svi primjeri do sada su bili relativno jednostavni utoliko što su oba repozitorija (udaljeni i naš lokalni) imali samo jednu granu – `master`. Idemo to sad (još) malo zakomplicirati i pogledati što se dešava ako kloniramo udaljeni repozitorij koji ima više grana:

Nakon `git clone` rezultat će biti:



Kloniranjem dobijamo samo kopiju lokalnog `master`, dok se sve grane čuvaju pod `origin/`. Dakle imamo `origin/master` i `origin/grana`. Da je bilo više grana u repozitoriju kojeg kloniramo, imali bi više ovih `origin/` grana. To lokalno možemo vidjeti s:

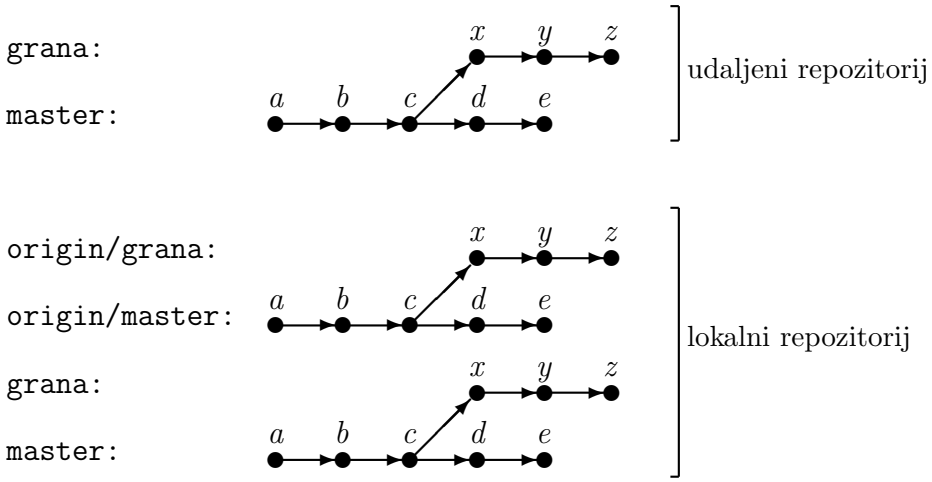
```
$ git branch -a
* master
  remotes/origin/master
  remotes/origin/grana
```

Već znamo da se možemo "prebaciti" s `git checkout origin/master`, ali se ne očekuje da tamo stvari i *commit*amo. Trebamo tu "`remote`" granu granati u naš lokalni repozitorij i tek onda s njom početi raditi. Dakle, u našem testnom slučaju, napravili bi:

```
git checkout origin/grana
git branch grana
git checkout grana
```

Zadnje dvije naredbe smo mogli skratiti u `git checkout -b grana`.

Sad je stanje:



... odnosno:

```
$ git branch -a
  master
* grana
  remotes/origin/master
  remotes/origin/grana
```

Sad u tu lokalnu `grana` možemo *commit*ati koliko nas je volja. Kad se odlučimo poslati svoje izmjene na udaljenu granu, postupak je isti kao i do sada, jedino što radimo s novom granom umjesto master. Dakle,

```
git fetch
```

... za osvježavanje **i** `origin/master` i `origin/grana`. Zatim...

```
git merge origin/grana
```

I, na kraju...

```
git push origin grana
```

...da bi svoje izmjene "poslali" u granu `grana` udaljenong repozitorija.

Brisanje udaljene grane

Želimo li obrisati granu na udaljenom repozitoriju – to radimo posebnom varijantom naredbe `git push`:

```
git push origin :grana-koju-zelimo-obrisati
```

Isto kao i kad *push*amo izmjene na tu granu, jedino što dodajemo dvotočku izpred naziva grane.

Ukoliko imamo granu `test` i *push*amo ju na udaljeni repozitorij – naši kolege će tu granu nakon *fetch*anja vidjeti kao `origin/test`. Međutim, ukoliko **obrišemo** granu na udaljenom repozitoriju, kolegama se neće automatski obrisati `origin/test`. To je ponekad malo zbunjujuće, ali s druge strane, barem nam je osiguranje da možemo skoro uvijek spasiti granu koju smo greškom obrisali. Ukoliko je to bilo slučaj – referenca na granu se i dalje nalazi u drugim kloniram repozitorijima.

Ukoliko uz *fetch* želite da vam se i obrišu sve grane koje je netko drugi obrisao na udaljenom repozitoriju, naredba je:

```
git fetch --prune
```

Udaljeni repozitoriji

Kloniranjem na našem lokalnom računalu dobijamo kopiju udaljenog repozitorija s jednim dodatkom – ta kopija je pupčanom vrpcom vezana za originalni repozitorij. Dobijamo referencu na `origin` repozitorij (onaj kojeg smo klonirali). Dobijamo i one `origin/` brancheve, koji su kopija udaljenih grana i mogućnost osvježavanja njihovog stanja s `git fetch`.

Imamo i neka ograničenja, a najvažnije je to što možemo imati samo jedan `origin`. Što ako želimo imati posla s više udaljenih repozitorija? Odnosno, što ako imamo više programera s kojima želimo surađivati od kojih svatko ima **svoj** repozitorij?

Drugim riječima, sad pomalo ulazimo u onu priču o gitu kao distribuiranom sustavu za verzioniranje.

Dodavanje i brisanje udaljenih repozitorija

Svi udaljeni repozitoriji bi trebali biti repozitorij istog projekta[36]. Bilo da su nastali kloniranjem ili kopiranjem nečijeg projekta (tj. kopiranjem `.git` direktorijia i *check-out*anjem projekta). Dakle, svi udaljeni repozitoriji s kojima ćemo imati posla su u nekom trenutku svoje povijesti nastali iz jednog jedinog projekta.

Sve naredbe s administracijom udaljenih (*remote*) repozitorija se rade s naredbom `git remote`.

Novi udaljeni repozitorij možemo dodati s `git remote add <naziv> <adresa>`. Na primjer, uzmimo da radimo s dva programera od kojih je jednome repozitorij na `https://github.com/korisnik/projekt.git`, a drugome `git@domena.com:projekt`. Ukoliko tek krećemo u suradnju s njima, prvi korak koji možemo (ali i ne moramo) napraviti je klonirati jedan od njihovih repozitorija:

```
git clone https://github.com/korisnik/projekt.git
```

...i time smo dobili udaljeni repozitorij `origin` s tom adresom. Međutim, mi želimo imati posla i sa repozitorijem drugog korisnika, za to ćemo i njega dodati kao *remote*:

```
git remote add bojanov-repo git@domena.com:projekt
```

...i sad imamo dva udaljena repozitorija `origin` i `bojanov-repo`. S obzirom da smo drugi nazvali prema imenu njegovog vlasnika, možda ćemo htjeti i `origin` nazvati tako. Recimo da je to Karlin repozitoriji, pa ćemo ga i nazvati tako:

[36] Git dopušta čak i da udaljeni repozitorij bude repozitorij nekog drugog projekta, ali rezultati *merge*anja će biti čudni.

```
git remote rename origin karlin-repo
```

Popis svih repozitorija s kojima imamo posla dobijemo s:

```
$ git remote show
bojanov-repo
karlin-repo
```

Kao i s `origin`, gdje smo kloniranjem dobili lokalne kopije udaljenih grana (one `origin/master`, ...). I ovdje ćemo ih imati, ali ovaj put će lokacije biti `bojanov-repo/master` i `karlin-repo/master`. Njih ćemo isto tako morati osvježavati da bi bile ažurne. Naredba je ista:

```
git fetch karlin-repo
git fetch bojanov-repo
```

Sad kad želimo isprobati neke izmjene koje je Karla napravila (a nismo ih još preuzeli u naš repozitorij), jednostavno:

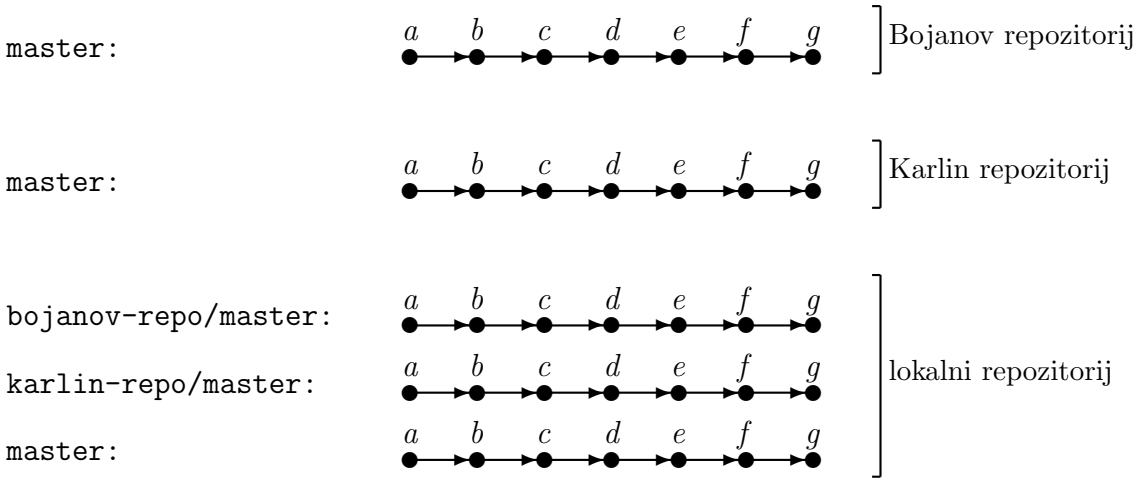
```
git checkout karlin-repo/master
```

...i isprobamo kako radi njena verzija aplikacije. Želimo li preuzeti njene izmjene:

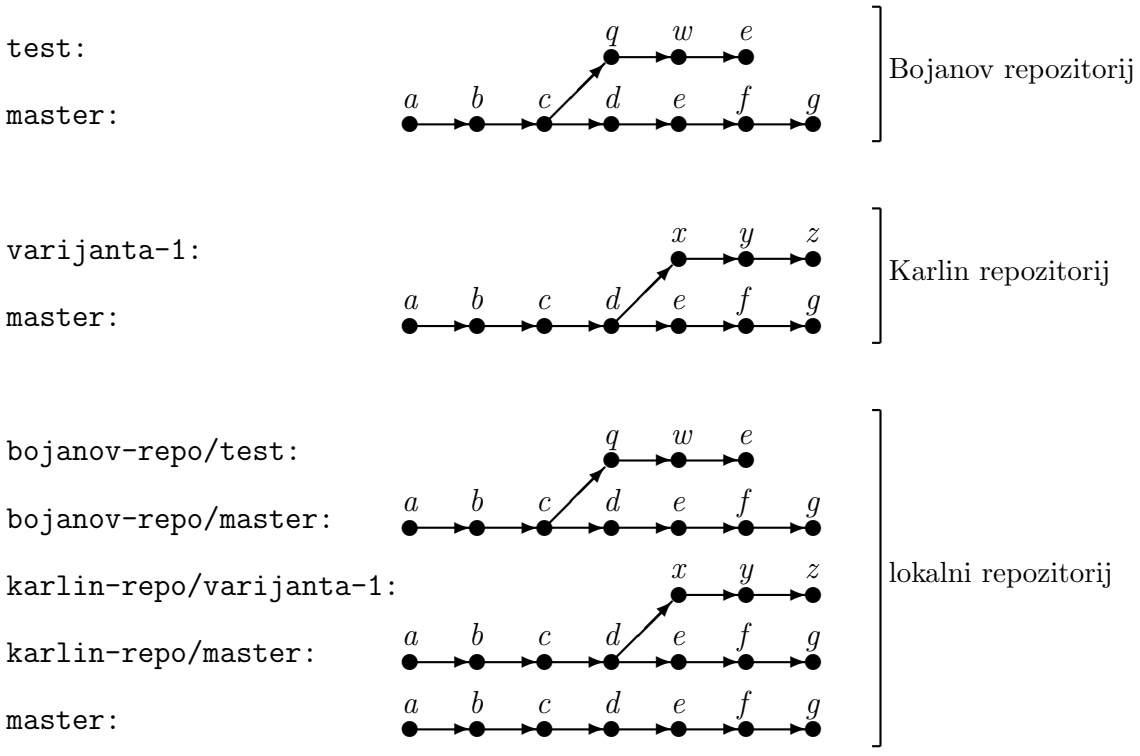
```
git checkout master
git merge karlin-repo/master
```

I, općenito, sve ono što smo govorili za *fetch*, *push*, *pull* i *merge* kad smo govorili o kloniranju vrijedi i ovdje.

Sve do sada se odnosilo na jednostavan scenarij u kojemu svi repozitoriji imaju samo jednu granu:



Naravno, to ne mora biti tako – Puno češća situacija će biti da svaki udaljeni repozitorij ima neke svoje grane:



. . . a rezultat od `git branch -a` je:

```
$ git branch -a
master
bojanov-repo/master
bojanov-repo/test
karlin-repo/master
karlin-repo/varijanta1
```

Fetch, *merge*, *pull* i *push* s udaljenim repozitorijima

Fetch, *merge*, *pull* i *push* s udaljenim repozitorijima je potpuno isto kao i s `origin` repozitorijem. U stvari, rad s njime i nije ništa drugo nego rad s udaljenim repozitorijem. Specifičnost je što u kloniranom repozitoriju možemo uvijek računati da imamo referencu na `origin` (dobijemo ju implicitno pri kloniranju), a druge udaljene repozitorije moramo "ručno" dodati s `git remote add`.

Recimo da nam je udaljeni repozitorij `ivanov-repo`. *Fetch* se radi ovako:

```
git fetch ivanov-repo
```

Nakon što smo osvježili lokalne kopije grana u `ivanov-repo`, *merge* radimo:

```
git merge ivanov-repo/master
```

...ili...

```
git merge ivanov-repo/grana
```

Pull:

```
git pull ivanov-repo master
```

...ili...

```
git pull ivanov-repo grana
```

... a *push*:

```
git push ivanov-repo master
```

... ili...

```
git push ivanov-repo grana
```

Naravno, na Ivanovom repozitoriju moramo imati postavljene ovlasti da bi mogli te operacije raditi.

Ukoliko s njegovog repozitorija možemo *fetch*ati, a ne možemo na njega *push*ati (dakle, pristup nam je *read-only*) – može se dogoditi ovakva situacija: Napravili smo izmjenu za koju mislimo da bi poboljšala aplikaciju koja je u Ivanovom repozitoriju. No, nemamo ovlasti *push*ati. Htjeli bi Ivanu nekako dati do znanja da mi imamo nešto što bi njega zanimalo.

Najbolje bi bilo da mu jednostavno pošaljemo adresu našeg git repozitorija i kratku poruku s objašnjenjem što smo točno izmijenili i prijedlogom da on te izmjene *pull*a (ili *fetch*a i *merge*a) u svoj repozitorij. Napravimo li tako, upravo smo poslali *pull request*.

Pull request

Pull request nije ništa drugo nego kratka poruka vlasniku nekog udaljenog repozitorija koja sadržava adresu **našeg** repozitorija, opis izmjena koje smo napravili i prijedlog da on te izmjene preuzme u svoj repozitorij.

Ukoliko koristite Github za svoje projekte – tamo postoji vrlo jednostavan i potpuno automatizirani proces za *pull requeste*[37]. U suprotnom, trebate doslovno poslati email vlasniku repozitorija s porukom. Postoji i naredba (`git request-pull`) koja priprema sve detalje izmjena koje ste napravili za tu poruku.

[37] Treba ovdje napomenuti da Linus Torvalds ima neke prigovore na Githubov *pull request* proces.

Bare repozitorij

U koje i kakve repozitorije smijemo *push*ati? Jedino što znamo je da udaljeni repozitorij treba tako biti konfiguriran da imamo **ovlast *push*ati**, ali ima još jedan detalj kojeg treba spomenuti.

Pretpostavimo da postoji Ivanov i naš repozitorij. Ivan ima svoju radnu verziju projekta koja je ista kao i zadnja verzija u njegovom repozitoriju. Isto je i s našim repozitorijem (dakle, i nama i njemu `git status` pokazuje da nemamo lokalno nikakvih ne*commit*anih izmjena).

Sad recimo mi napravimo izmjenu, *push*amo u Ivanov repozitorij i tako, a da on toga uopće nije svjestan, mijenjamo status radne verzije njegovog projekta. Njemu će se činiti kao da mu, u jednom trenutku `git status` kaže da nema nikakvih izmjena, a sekundu kasnije (bez da je išta editirao) – `git status` kaže da se njegova radna verzija razlikuje od zadnjeg stanja u repozitoriju. `git status`, naime pokazuje razlike između radne verzije repozitorija i **zadnjeg *commit*a** u repozitoriju. S našim *push*em – mi smo upravo promijenili taj zadnji *commit* u njegovom repozitoriju.

Git nam to neće dopustiti, a odgovoriti će dugom i kriptičnom porukom koja počinje ovako:

```
remote:  error:  refusing to update checked out branch:
refs/heads/master
remote:  error:  By default, updating the current branch in a
non-bare repository
remote:  error:  is denied, because it will make the index and work
tree inconsistent
remote:  error:  with what you pushed, and will require 'git reset
--hard' to match
remote:  error:  the work tree to HEAD.
```

Ako malo razmislite, to izgleda kao kontradikcija sa cijelom pričom o udaljenim repozitorijima. Kako to da nam git ne da *push*ati u nečiji repozitorij, čak i ako imamo ovlasti? Čemu onda uopće *push*?

Rješenje je sljedeće: postoji posebna vrsta repozitorija koji **nemaju radnu verziju projekta**. To jest, oni nisu nikad *checkout*ani, a sadrže samo `.git` direktorij. I to je

bare[38] repozitorij. Kod takvih – čak i kad *push*amo – nećemo nikad promijeniti status radne verzije. Radna verzija je tu irelevantna.

Kako se to uklapa u priču da ponekad moramo *push*ati u nečiji repozitorij?

Jednostavno, svatko bi trebao imati svoj lokalni repozitorij na svom lokalnom računalu na kojeg **nitko ne smije *push*ati**. Trebao bi imati i **svoj udaljeni repozitorij** na kojeg će onda *push*ati svoje izmjene. Na tom udaljenom repozitoriju bi on mogao/trebao postaviti ovlasti drugim programerima u koje ima povjerenje da imaju ovlasti *push*ati.

Na taj način nitko nikada ne može promijeniti status lokalnog (radnog) repozitorija bez da je njegov vlasnik toga svjestan. To može na udaljenom repozitoriju, a onda vlasnik iz njega može *fetch*ati, *pull*ati i *push*ati.

***Bare* repozitorij je repozitorij koji je zamišljen da bude na nekom serveru, a ne da se na njemu direktno *commit*a**. Drugim riječima, nisu svi direktoriji jednaki: postoje oni lokalni na kojima programiramo i radimo stvari i postoje oni udaljeni (*bare*) na koje *push*amo i s kojih *fetch*amo i *pull*amo.

Konvertirati neki repozitorij u *bare* je jednostavno:

```
git config --bool core.bare true
```

Još jedan scenarij u kojem će nam ova naredba biti korisna je sljedeći: recimo da smo krenuli pisati aplikaciju na lokalnom git repozitoriju. I nismo imali nikakvih drugih udaljenih repozitorija. U jednom trenutku se odlučimo da smo već dosta toga napisali i želimo imati sigurnosnu kopiju na nekom udaljenom računalu. Kreiramo tamo doslovnu kopiju našeg direktorija. Na lokalnom računalu napravimo:

```
git remote add origin login@server:staza/do/repozitorija
```

Ovdje smo išli suprotnim putem – nismo klonirali repozitorij i tako dobili referencu na `origin`, nego smo lokalni repozitorij kopirali na udaljeno računalo i tek sada postavili da nam je `origin` taj udaljeni repozitorij. Pokušamo li sada *push*ati svoj `master`:

```
git push origin master
```

[38] Engleski: gol. Npr. *barefoot* – bosonog.

Dobiti ćemo **onu** grešku:

```
remote: error: refusing to update checked out branch:
refs/heads/master
...
```

Rješenje je da se još jednom spojimo na udaljeno računalo[39] i tamo izvršimo:

```
git config --bool core.bare true
```

I to će gitu na udaljenom računalu reći: "ovom repozitoriju je namijena da ljudi na njega *push*aju, s njega *pull*aju i *fetch*aju – dopusti im to!".

[39] Telnetom ili (još bolje) koristeći ssh.

"Higijena" repozitorija

Za programere je repozitorij životni prostor i s njime se živi dio dana. Kao što se trudimo držati stan urednim, tako bi trebali i s našim virtualnim prostorima. Preko tjedna, kad rano ujutro odlazimo na posao i vraćamo se kasno popodne, ponekad se desi da nam se u stanu nagomila robe za pranje. Zato nekoliko puta tjedno treba odvojiti pola sata i počistiti nered koji je zaostao, inače će entropija zavladati, a to nikako ne želim(o). Nadam se.

Tako i s repozitorijem; treba ga redovito održavati i čistiti nered koji ostavljamo za sobom.

Grane

Iako nam git omogućuje da imamo neograničen broj grana, ljudski um nije sposoban vizualizirati si više od 5 do 10 njih[40]. Kako stvaramo nove grane, događa se da imamo one u kojima smo počeli neki posao i kasnije odlučili da nam to neće trebati. Ili smo napravili posao, *merge*ali u `master`, ali nismo obrisali granu. Nađemo li se s više od 10-15 grana **sigurno** je dio njih tu samo zato što smo ih zaboravili obrisati.

U svakom slučaju, predlažem vam da svakih par dana pogledate malo po lokalnim (a i udaljenim granama) i provjerite one koje više ne koristite.

Ako nismo sigurni je li nam u nekoj grani ostala možda još kakva izmjena koju treba vratiti u `master`, možemo koristiti naredbu:

```
git branch --merged master
```

[40] Barem moj nije, ako je vaš izuzetak, preskočite sljedećih nekoliko rečenica. Ili jednostavno zamislite da je umjesto "5-10" pisalo "500-1000".

To će nam ispisati popis svih grana čije izmjene **su u potpunosti** *merge*ane u master. Analogno, postoji i naredba s kojom dobijamo popis svih onih grana koje **nisu** u potpunosti *merge*ane u neku drugu granu:

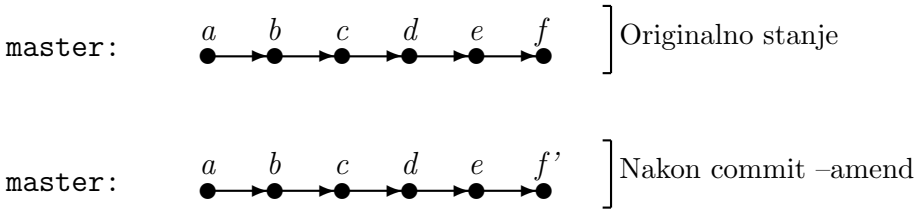
```
git branch --no-merged <naziv_grane>
```

Ako baš morate imati puno grana, onda dogovorite s drugim ljudima u projektu neki logičan način kako ćete grane imenovati. Na primjer, ako koristite neki sustav za prijavu i praćenje grešaka, onda svaka greška ima neki svoj broj. Možete se odlučiti svaku grešku ispravljati u posebnoj grani. Imate li veliku i kompleksnu aplikaciju, biti će i puno prijavljenih grešaka, a posljedično i puno grana. Tada grane možete imenovati prema broju prijavljene greške zajedno s kratkim opisom. Na primjer, `123-unicode-problem` bi bila grana u kojoj ispravljate problem prijavljen pod brojim 123, a radi se o (tko bi rekao?) nekom problemu s *unicode* enkodiranjem. Sad, kad dobijete spisak svih grana, odmah ćete znati koja grana čemu služi.

Git gc

Druga stvar koja se preporuča ima veze s onim našim `.git/objects` direktorijem kojeg smo spominjali u "Ispod haube" poglavlju. Kao što znamo, svaki *commit* ima svoju referencu i svoj objekt (datoteku) u tom direktoriju. Kad napravimo `git commit --amend` – git stvara **novi** *commit*. Nije da on samo izmijeni stari[41].

Grafički:



Dakle, git interno dodaje **novi** objekt (*f'*) i na njega pomiče referencu `HEAD` (koja je do tada gledala na *f*). On samo "kaže": Od sada na dalje, zadnji *commit* u ovoj grani više nije *f*, nego *f'*.

[41] Ne bi ni mogao izmijeniti stari jer ima drukčiji sadržaj i SHA1 bi mu se nužno morao promijeniti.

Sad u git repozitoriju imamo i *commit f*, a i *f'*, ali samo jedan od njih se koristi (*f'*). Commit *f* je **i dalje snimljen u `.git/object` direktoriju**, ali on se više neće koristiti. Puno tih `git commit --amend` posljedično ostavlja puno "smeća" u repozitoriju.

To vrijedi i za neke druge operacije kao brisanje grana ili rebase. Git to čini da bi tekuće operacije bile što je moguće brže. Čišćenje takvog "smeća" (*garbage collection* iliti *gc*) ostavlja za kasnije, a ta radnja nije automatizirana nego se od nas očekuje da ju pokrenemo[42].

Naredba je `git gc`:

```
$ git gc
Counting objects: 1493, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (541/541), done.
Writing objects: 100% (1493/1493), done.
Total 1493 (delta 910), reused 1485 (delta 906)
```

...i nju treba izvršavati s vremena na vrijeme.

Osim `gc`, postoji još nekoliko sličnih naredbi kao `git repack`, `git prune`, no one su manje važne za početnika. Ako vas zanimaju – `git help` je uvijek na dohvat ruke.

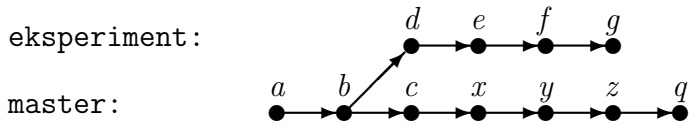
Povijest i brisanje grana

Spomenuti ćemo još nešto što bi logički pripadalo u poglavlja o granama i povijesti, ali tada za to nismo imali dovoljno znanja.

Što se događa s *commit*ovima iz neke grane nakon što je obrišemo? Uzmimo tri primjera. U sva tri imamo dvije grane: `master` i `eksperiment`.

Prvi primjer:

[42] Nije automatizirana, ali možemo uvijek sami napraviti neki task koji se izvršava na dnevnoj ili tjednoj bazi, a koji "čisti" sve naše git repozitorije.



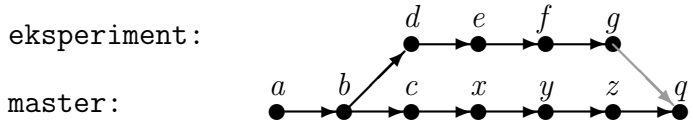
Pravilo po kojem git razlikuje čvorove koje će ostaviti u povijesti od onih koje će obrisati jednostavno je: **Svi čvorovi koji su dio povijesti projekta ostat će u repozitoriju i neće biti obrisani s** `git gc`. Kako znamo koji čvorovi su dio **povijesti projekta**? Po tome što postoji nešto (grana, čvor ili *tag*) što ima referencu na njih.

Krenimo sad primijeniti to pravilo na naš primjer...

Podsjetimo se da su strelice redoslijed nastajanja, ali reference idu suprotnim smjerom, sljedbenik ima referencu na prethodnika. Dakle, g ima referencu na f, f na e, itd. Što je sa čvorom g? Izgleda kao da nitko nema referencu na njega, ali to nije točno; grana `eksperiment` je referenca na njega.

Ako obrišemo granu `eksperiment` – g više nema nikoga da se njega referencira. `git gc` će ga obrisati, ali onda mora obrisati i f, e i d (b ne možemo, jer c ima referencu na njega). Dakle, kad obrišemo granu koja nije *merge*ana u neku drugu granu, onda se svi njeni čvorovi gube iz povijesti projekta.

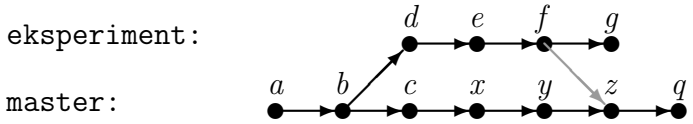
Drugi primjer:



Ovaj primjer je isti kao i prvi s jednom razlikom. Došlo je do *merge*a.

Znamo da grana nije ništa drugo nego referenca na zadnji čvor/*commit*. Obrišemo li granu `eksperiment`, obrisali smo referencu na zadnji čvor g, ali i dalje imamo q koji pokazuje na g. Zbog toga će svi čvorovi grane `eksperiment` nakon njenog brisanja ostati u repozitoriju.

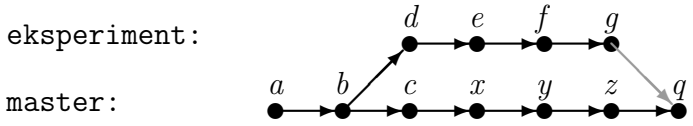
Treći primjer:



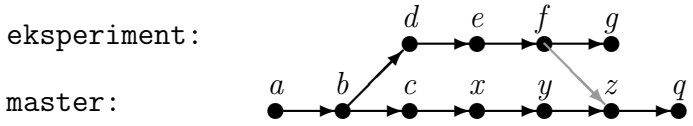
Ako u ovom primjeru obrišemo `eksperiment`, postoji samo jedan čvor koji će biti izgubljen, a to je g. Bez reference na granu, niti jedan čvor niti *tag* ne pokazuje na g. Dakle, on prestaje biti dio povijesti našeg projekta. Za razliku od njega, z ima referencu na f, a s f nam garantira da i e i d ostaju dio povijesti projekta.

Digresija o brisanju grana

Uzmimo opet iste dvije situacije, onu u kojoj će se svi čvorovi grane sačuvati:



. . . i onu u koju gubimo samo neke čvorove:



Prvu situaciju zovemo **potpuno *merge*ana grana**, a drugu **djelomično *merge*ana grana**.

Znamo sad da postoje situacije u kojima čak i nakon brisanja grane – njeni *commit*ovi ostaju u povijesti projekta. Mogli bi si postaviti pitanje: "Zašto uopće brisati grane?" Odgovor je jednostavan: brisanjem grane nećemo više tu granu imati u listi koji dobijemo s `git branch`. Kad bi tamo imali sve grane koje smo ikad imali u povijesti projekta (a njih može biti jako puno) bilo bi se teško snaći u podužem ispisu.

Druga digresija koju ćemo ovdje napraviti tiče se brisanja grane. Postoje dva načina. Prvi kojeg smo već spomenuli:

```
git branch -D grana
```

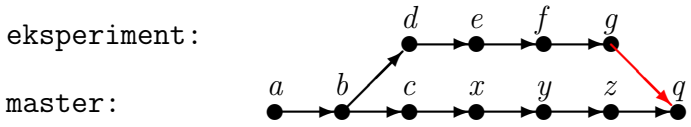
. . . koji bezuvjetno briše granu `grana`, a drugi je:

```
git branch -d grana
```

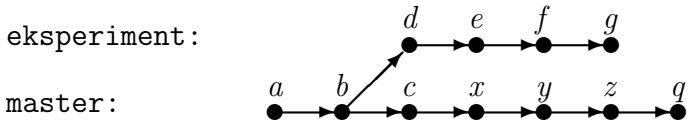
. . . koji će obrisati granu samo ako **jest** potpuno *merge*ana. Ako nije, odbiti će obrisati. Dakle, ako vas je strah da biste slučajno obrisali granu čije izmjene još niste preuzeli u neku drugu granu – koristite `-d` umjesto `-D` kod brisanja.

Squash merge i brisanje grana

Uzmimo opet:



Želimo li u povijesti projekta sačuvati izmjene iz neke grane, ali ne i njenu povijest, to se može s `git merge --squash`. Podsjetimo se, tom operacijom git **hoće** preuzeti izmjene iz grane, ali **neće** u čvoru q napraviti referencu na g. Dakle, rezultat je kao kod klasičnog *merge*a, ali bez reference (u prethodnom grafu crvenom bojom):



Sad smo preuzeli izmjene iz `grana` u `master`, ali `git gc` će prije ili kasnije obrisati d, e, f i g.

S `git merge --squash` cijelu granu svodimo na jedan *commit* i kasnije gubimo njenu povijest[43].

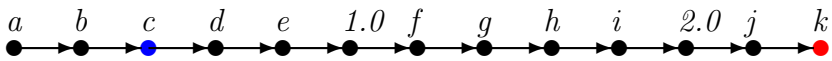
[43] ... barem ako je kasnije ne *merge*amo klasičnim putem.

Bisect

Bisect je git naredba koja se koristi kad tražimo izmjenu u kodu koja je uzrokovala grešku (*bug*) u programu. Da bi mogli koristiti *bisect* važno je da:

- Imamo način kako utvrditi da li se bug manifestira u kodu kojeg trenutno imamo. Na primjer, *unit* test ili shell skriptu koja provjerava postojanje buga.
- Znamo da je *bug* nastao u nekom trenutku u povijesti projekta (tj. sigurni smo da se nije manifestirao u ranijim verzijama aplikacije).

Ukoliko su oba kriterija zadovoljena, moramo znati koji je točno raspon commitova u kojima tražimo *bug*. Uzmimo na primjer da naš projekt ima povijest:



Znamo li da je *c* commit u kojemu se *bug* nije manifestirao, a zadnje stanje u našem *branch*u *k* je pozicija gdje se bug manifestirao, onda znamo i da je bug nastao negdje u *commit*ovima između ta dva.

Bisect nije ništa drugo nego binarno pretraživanje po povijesti projekta. U prvom koraku moramo gitu dati do znanja koji je *commit* dobar (tj. u kojemu se *bug* **nije manifestirao**), a koji je loš (tj. u kojemu se *bug* **manifestira**). I nakon toga slijedi niz iteracija pri čemu nas git prebacuje na neki *commit* između zadnjeg dobrog i lošeg. U svakom koraku se interval suzuje sve dok ne dođemo do mjesta kad je problem nastao.

Uzmimo, na primjer, da se trenutno nalazimo u grani u kojoj imamo bug. Gitu dajemo do znanja da želimo započeti *bisect* i da je trenutni *commit* "**loš**":

```
$ git bisect start
$ git bisect bad
```

Prebacujemo se na stanje za koje smo sigurni da se *bug* nije manifestiralo:

```
$ git checkout b9cee8abaf1c6ffc8b7d9bbb63cafb2c0cbdbdd0
Note: checking out 'b9cee8abaf1c6ffc8b7d9bbb63cafb2c0cbdbdd0'.
```

Dajemo mu do znanja da je to "**dobar** *commit*"[44]:

```
$ git bisect good
Bisecting: 6 revisions left to test after this (roughly 3 steps)
[a63ad54907b5247a7f507fc769df2c5794d93d7c] Started implementing
add_elevations [tmp]
```

Git nas sad prebacuje na neki *commit* između ta dva. Nama je irelevantno koji je točno. Jedino što trebamo je isprobati pojavljuje li se *bug* u kodu koji je trenutno *checkout*an.

Na primjer, kod mene se *bug* manifestirao, dakle pišem:

```
$ git bisect bad
Bisecting: 3 revisions left to test after this (roughly 2 steps)
[6385fc100431166688e1424ea877444c74aee8b2] min_points fixed
```

U sljedećem koraku sam na *commit*u gdje *bug*a nema:

```
$ git bisect good
Bisecting: 1 revision left to test after this (roughly 1 step)
[5f217903832edec04d9dd267e5bf78b0f7275b49] + add_missing_*() methods
[tmp]
```

[44]Naravno, ako niste sigurni probajte otići još malo dalje u povijest sve dok ne dođete do *commit*a gdje se *bug* sigurno ne manifestira

. . . i dalje. . .

```
$ git bisect good
Bisecting: 0 revisions left to test after this (roughly 0 steps)
[b87db36d71038074a1c478c9f9a329d5c1685a02] add_missing_points() fixed
+ tests
```

. . . sve dok u nekom trenutku ne dođemo do krivca:

```
$ git bisect bad
b87db36d71038074a1c478c9f9a329d5c1685a02 is the first bad commit
commit b87db36d71038074a1c478c9f9a329d5c1685a02
Author: Tomo Krajina <tkrajina@gmail.com>
Date:   Fri Aug 2 06:50:39 2013 +0200

    add_missing_points() fixed + tests

:040000 040000 444ac64d4e0052562fce0cbe367dbb98b471680b
13b1bbffce34bb025d8eb2e9b946295457b03977 M        gpxpy
:100644 100644 ca8d95ca4c070fc1885680dea305a7ffdf3e594d
8776678abe3d8c9faef0b9e8b8395a4328cbb28c M        test.py
```

Dakle, krivac je *commit* `b87db36d71038074a1c478c9f9a329d5c1685a02`. Da bi točno pogledali što je tada promijenjeno, možemo:

```
git diff b87db36d7~1 b87db36d7
```

Ili:

```
gitk b87db36d7
```

Ukoliko u bilo kojem trenutku *bisect* želimo prekinuti i vratiti se na mjesto (*commit*) gdje smo bili kad smo započeli, naredba je:

```
git bisect reset
```

Automatski *bisect*

Bisect se radi u koracima, a u svakom koraku morate provjeriti je li *bug* prisutan ili nije. Ponekad će ta provjera zahtijevati da restartate web server i provjerite u pregledniku (*browser*u), a u drugim slučajevima ćete jednostavno izvrtiti test. Ukoliko imate spremne integracijske ili *unit* testove ili pak neku naredbu koju možete pokrenuti u komandnoj liniji onda se postupak traženja *bug*a može automatizirati naredbom `git bisect run`.

Sintaksa je sljedeća:

```
git bisect run <naredba>
```

Pretpostavka je da ste prethodno već odredili početni `bad` i `good` *commit*.

Važno je da naredba koju pokrećete završava sa statusom `0` ukoliko je stanje ispravno ili brojem od 1 do 127 ukoliko nije.

Vrtite li testove s `make test`, možete koristiti:

```
git bisect run make test
```

... a koristite li javu i `maven` možete:

```
git bisect run mvn test
```

... ili se bug manifestira na samo jednom testu:

```
git bisect run mvn test -Dtest=MojUnitTest
```

Nakon toga će git samostalno izvrtiti sve *bisect* korake i naći prvi *commit* u kojemu

je skripta završila sa statusom između 1 i 127[45].

Ukoliko se bug nije manifestirao u postojećim testovima, nego ste test napisali naknadno tada vam `make test` neće pomoći. Tada možete sami napisati novu skriptu koja će u svakom koraku *bisect*a dodati taj test i izvrtiti ga[46].

Digresija o atomarnim *commit*ovima

Bisect vam neće otkriti točan uzrok problema, samo će vamo točno reći koje su se izmijene dogodile kad je problem nastao, ali i to je često dovoljno kod traženja pravog uzroka.

Nađe li vam *bisect* da ste u tom *commit*u promijenili 5 linija koda tada barem jedna od tih linija koda nužno mora biti krivac. Ako ste u tom *commit*u promijenili 100 linija koda među njima je krivac, ali lakše je bug tražiti u 5 nego u 100 izmijenjenih linija.

Radite li *commit*ove koji imaju tisuće promijenjenih linija, onda je puno teže naći uzrok. No, to je ionako kriv pristup. Ne bi nikad smjeli snimati više od jedne izmjene u koraku. Svaki *commit* bi uvijek trebao predstavljati jednu jedinu logičku cjelinu.

Na primjer, ako ste izmijenili dokumentaciju, ispravili nevezani bug i preformatirali kod u nekoj trećoj klasi – to ne bi nikako smio biti **jedan** *commit* nego **tri**.

Drugim riječima, treba raditi **atomarne** *commit***ove**.

Na taj način će vam izmjene u kodu uvijek biti male, a i *bisect* će nam s puno većom preciznošću moći pomoći kod traženja uzroka problema.

[45] To su standardni statusi s kojima programi operativnom sustavu daju do znanja da su završili s nekom greškom

[46] Možete na primjer napraviti posebnu granu s novim testom i onda u svakom koraju napraviti $cherry - pick$ tog testa prije nego izvrtite testove.

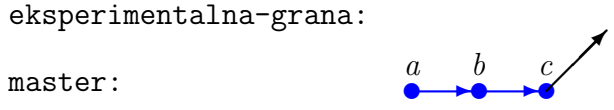
Prikaz grana u git alatima

Način kako su grafovi repozitorija prikazivani u ovoj knjizi nije isti kako ih prikazuju grafički alati za rad s gitom. Odlučio sam se na ovakav prikaz jer mi se činilo intuitivnije za razumijevanje, ali da bi lakše radili s alatima kao što je `gitk` napraviti ću ovdje kratak pregled kako ti alati prikazuju povijest, *commit*ove i grane.

Osnovna razlika je u tome što grafički alati obično prikazuju povijest od dolje (starije) prema gore (novije) i to što *commit*ovi iz iste grane nisu prikazani u posebnom retku (ili stupcu).

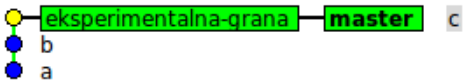
Prikaz lokalnih grana

Situacija kojoj imamo samo `master` i onda stvorimo ovu granu u kojoj još nismo ništa *commit*ali:



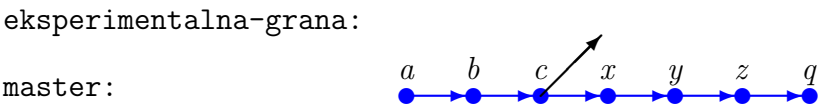
Ovdje je strelica prikazano samo zato da se vidi da smo `eksperimentalna-grana` kreirali iz *commit*a *c*. No, ta grana je trenutno ista kao i `master`.

U `gitk`u će ta ista situacija biti prikazana kao:

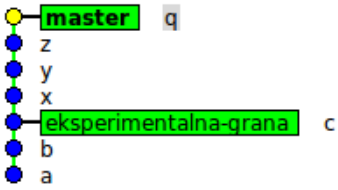


Drugim riječime, `master` i `eksperimentalna-grana` pokazuju na isti *commit*.

Slična situacija:



. . . će biti prikazana ovako nekako:

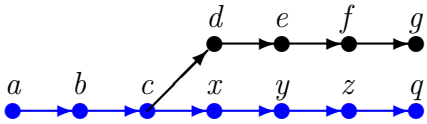


Kao što vidite, u istom stupcu su prikazane obje grane, samo je označen zadnji *commit* za svaku granu. No, to već znamo – *commit* i nije ništa drugo nego pokazivač na jedan *commit*.

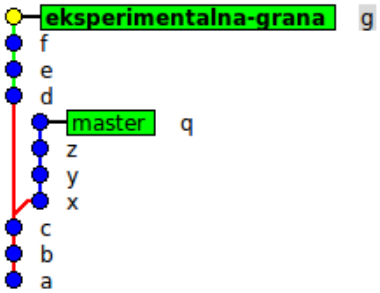
Ukoliko imate dvije grane u kojima se paralelno razvijao kod:

`eksperimentalna-grana:`

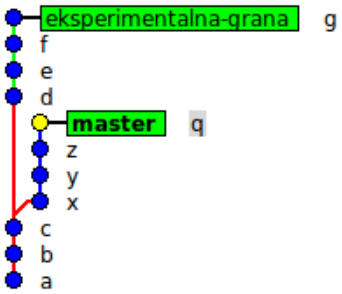
`master:`



. . . `gitk` će prikazati:



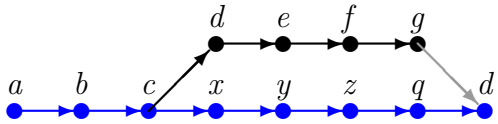
. . . u principu slično, jedino što se alat trudi da pojedine *commit*ove prikazuje svakog u posebnom redu. Ukoliko se s `master` prebacite na `eksperimentalna-grana`, graf će biti isti jedino će se prikazati na kojoj ste točno grani:



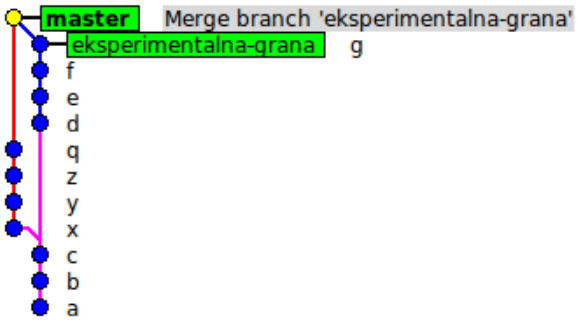
Nakon ovakvog *merge*a:

`eksperimentalna-grana:`

`master:`



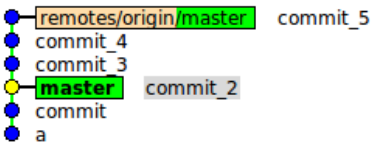
. . . graf je:



Prikaz grana udaljenog repozitorija

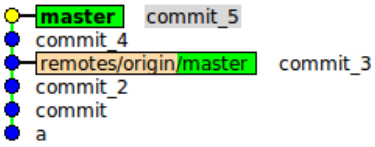
Ukoliko imamo posla i s udaljenim repozitorijima, onda će njihove grane biti prikazane na istom grafu kao npr. `remotes/origin/master` ili `origin/grana`.

Recimo da nemamo ništa za *push*ati na udaljeni repozitorij (nego čak imamo nešto za **preuzeti** iz njega):

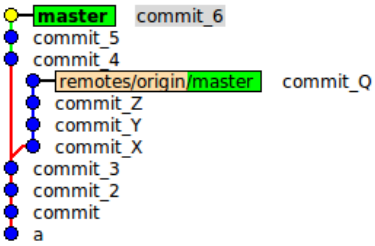


Iz ovoga je jasno da za naš `master` "zaostaje" za četiri *commit*a u odnosu na udaljeni `master`.

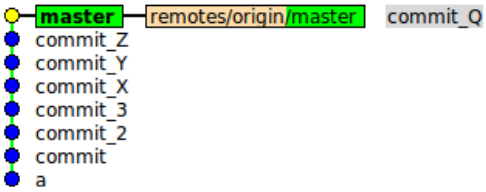
Primjer gdje imamo dva *commit*a koje nismo *push*ali, a mogli bismo:



Primjer gdje imamo tri *commit*a za *push*anje, ali trebamo prije toga preuzeti četiri *commit*a iz `origin/master` i *merge*ati ih u našu granu:



I, situacija u kojoj je lokalni `master` potpuno isti kao udaljeni `origin/master`:



Česta pitanja

Jedno je razumijeti naredbe i terminologiju gita, a potpuno drugo je imati iskustvo u radu s gitom. Da bi nekako došli do iskustva, trebamo imati osjećaj o tome koji su problemi koji se pojavljuju u radu i trebamo automatizirati postupak njihovog riješavanja. U ovom poglavlju ćemo proći nekoliko takvih "situacija".

Jesmo li *push*ali svoje izmjene na udaljeni repozitorij?

S klasičnim sustavima za verzioniranje, kod kojeg smo izmijenili može biti ili lokalno ne*commit*an ili *commit*an na centralnom repozitoriju.

Kao što sad već znamo, s gitom je stvar za nijansu složenija. Naš kod može biti lokalno ne*commit*an, može biti *commit*an na našem lokalnom repozitoriju, a može biti i *push*an na udaljeni repozitorij. Više puta mi se desilo da netko od kolega (tko tek uči git) pita "Kako to da moje izmjene nisu završile na produkciji[47], **iako sam ih** *commit***ao**?". Odgovor je jednostavan – *commit*ao ih je lokalno, ali nije *push*ao na naš glavni repozitorij.

Problem kojeg on ima je u tome što nije nigdje jasno vidljivo jesu li izmjene iz njegove `master` grane *push*ane na udaljeni repozitorij.

Jednostavan način da to provjerimo je da provjerimo odnos između `master` i `origin/master`. Za svaki slučaj, prvo ćemo osvježiti stanje udaljenog repozitorija s:

```
git fetch
```

...i sad idemo **vizualno** proučiti odnos između naše dvije grane:

[47] ...ili produkcijskom *build*u.

```
gitk master origin/master
```

Sad pogledajte na grafu je li:

- `master` **ispred** `origin/master`, u tom slučaju vi imate više *commit*ova od udaljenog repozitorija i možete ih *push*ati,
- `master` **iza** `origin/master`, u tom slučaju vi imate manje *commit*ova od udaljenog repozitorija i trebate ih pokupiti s udaljenog repozitorija (*pull* ili *rebase*),
- `master` i `origin/master` se nalaze na dvije grane koje su međusobno divergirale (u tom slučaju vi imate izmjene koje niste još *push*ali, ali trebate prije toga napraviti *pull*).
- `master` i `origin/master` pokazuju na isti čvor, tada je lokalno stanje potpuno isto ko i stanje udaljenog repozitorija.

Za primjere grafova, pogledajte poglavlje o prikazu grafova.

*Commit*ali smo u krivu granu

Na primjer, slučajno smo commitali u `master`, a trebali smo u `unicode-fix`. Pretpostavimo da su zadnja dva *commit*a iz `master` ona koja želimo prebaciti u ovu drugu granu.

Rješenje je jednostavno, prvo ćemo se prebaciti u tu drugu granu:

```
git checkout unicode-fix
```

Zatim ćemo preuzeti jedan po jedan ta dva *commit*a u trenutnu granu:

```
git cherry-pick master~1
git cherry-pick master
```

Podsjetimo se da je `master` naziv grane, ali i pokazivač na njegov zadnji *commit*, tako da `git cherry-pick master` preuzima samo taj zadnji *commit*. Commit `master~1` se

odnosi na pretposljednji u toj grani.

Umjesto `master` i `master~1` smo mogli koristiti i SHA1 identifikatore *commit*ova, koje možemo dobiti s `git log master`.

Sad, kad smo te *commit*ove prebacili (i) u željenu granu, trebamo ih maknuti iz one u kojoj su neželjeni. Idemo se prvo prešaltati na nju:

```
git checkout master
```

I, idemo ih obrisati:

```
git reset --hard master~2
```

...što tu granu resetira na stanje u `master~2` (a to je pred-predzadnji *commit*).

*Commit*ali smo u granu X, ali te commitove želimo prebaciti u novu granu

*Commit*ali smo u `master`, ali u jednom trenutku smo zaključili da te izmjene ne želimo tu. Želimo stvoriti novu granu koja će nam sačuvati te *commit*ove, a `master` resetirati na isto stanje kao i u udaljenom repozitoriju. Pa, idemo redom, s...

```
git branch nova-grana
```

...ćemo kreirati novu granu iz `master`. Te dvije grane su trenutno potpuno iste, dakle, upravo smo riješili prvi dio zadatka – sačuvali smo *commit*ove iz `master` u drugoj grani.

S obzirom da nam stanje u `master` treba biti isto kao u `origin/master`, prvo ćemo se potruditi da lokalno imamo ažurno stanje udaljenog repozitorija:

```
git fetch
```

...i sad idemo izjednačiti `master` i `origin/master`:

```
git reset --hard origin/master
```

Imamo ne*commitane* izmjene i git nam ne da prebacivanje na drugu granu

Imamo li ne*commit*anih izmjena, git ponekad neće dopustiti prebacivanje (*checkout*anje) s grane na granu. Ukoliko te izmjene predstavljaju neku logičnu cjelinu – onda ćemo ih jednostavno *commit*ati i to nije problem. No, ako se nalazimo na pola posla i to ne želimo...

To se može zaobići na dva načina. Jedan je da koristimo naredbu `git stash`, a drugi je da ipak – *commit*ate. Problem s ovim drugim pristupom je što ćemo imati djelomični *commit* s poluzavršenim kodom. Međutim, kad se naknadno vratimo na ovu granu (nakon što obavimo posao na nekoj drugoj) – možemo posao završiti i commitati ga s:

```
git commit --amend -m "Novi komentar...
```

I, osvježili smo prethodni polovični *commit*. Ukoliko to činimo, treba samo pripaziti da svoj "privremeni" *commit* ne *push*ate na udaljeni repozitorij dok nije gotov[48].

Zadnjih *n commit*ova treba "stisnuti" u jedan *commit*

S `git log` ili `gitk` nađimo SHA1 identifikator zadnjeg *commit*a kojeg **želimo ostaviti netaknutog** (tj. sve *commitove* **nakon** njega želimo "stisnuti" u jedan *commit*). Neka je to, na primjer, `15694d32935f07cc66dbc98fdd7b3b248d885492`.

Treba pripaziti se da lokalno u repozitoriju nemamo nikakvih ne*commit*anih izmjena i da se nalazimo u pravoj grani, a onda:

```
git reset --soft 15694d32935f07cc66dbc98fdd7b3b248d885492
```

[48] To općenito vrijedi za *commit*ove, nemojte koristiti "commit –amend" ukoliko ste već *push*ali na udaljeni repozitorij.

Git će nas sad vratiti u povijest, ali datoteke će ostaviti u istom stanju u kakvom su bile snimljene. Sad ih možemo *commit*ati iznova i dobiti ćemo ono što smo tražili.

*Push*ali smo u remote repozitorij izmjenu koju nismo htjeli

Kad smo **lokalno** napravili izmjenu koju nismo htjeli – možemo koristiti `git reset --hard ...`.

Međutim, ako smo našu izmjenu *push*ali, onda je najbolje napraviti:

```
git revert <nas_commit>
```

Podsjetimo se, `revert` stvara **novi** *commit* koji **miče** izmjene koje smo prethodno napravili. Nakon toga *push*ajmo još jednom na udaljeni repozitorij, i to je to.

Alternativa je da napravimo:

```
git reset --hard <commit>
git push -f origin <grana>
```

...međutim, to može biti problem ako je naše neželjene izmjene na udaljenom repozitoriju netko već preuzeo (*fetch*ao) kod sebe lokalno.

*Merge*ali smo, a nismo htjeli

Ukoliko svoje izmjene niste *push*ali na udaljeni repozitorij, jednostavno nađite *commit* prije vašeg *merge*a i napravite *reset* na to mjesto. Najčešće će biti dovoljno:

```
git reset --hard HEAD^1
```

Naravno, provjerite za svaki slučaj je li `HEAD^1` upravo onaj *commit* na kojeg želite vratiti vaš *branch*.

Ne znamo gdje smo *commit*ali

Napravili smo *commit*, prebacili se na neku drugu granu i malo se izgubili tako da sad više ne znamo gdje su te izmjene završile.

Jednu stvar koju možemo napraviti je pokrenuti `gitk --all` i pogledati možemo li naći taj *commit* u nekoj od postojećih grana. Ukoliko ga nađemo – možemo ga *cherry – pick*ati u našu granu i revertati u grani u kojoj je greškom završio.

No, ima jedan poseban slučaj na kojeg treba pripaziti. Ukoliko smo se *checkout*ali na neku *remote* granu (npr. `origin/master`) git nam neće javiti grešku ukoliko tamo i *commit*amo iako takve grane nisu zamišljene da na njima radimo, nego samo da preuzimamo izmjene iz njih u naše lokalne grane.

Na sreću, novije verzije gita će vas upozoriti kad se s takve grane želite vratiti na neku postojeću lokalnu i odmah vam sugerira što da učinite. Konkretno on vam predlaže da taj *commit* sačuvate kao novu granu koju ćete napraviti iz trenutnog stanja:

```
$ git checkout master
Warning: you are leaving 1 commit behind, not connected to
any of your branches:

  cec17e8 Tekst commita

If you want to keep them by creating a new branch, this may be a good
time
to do so with:

 git branch new_branch_name cec17e8044b15092e7e85daf4f25240a418eb54b

Switched to branch 'master'
Your branch is ahead of 'origin/master' by 3 commits.
```

No, ako ste zanemarili tu poruku (ili koristite IDE koji vam neće dati tu istu poruku) prisjetite se naredbe $reflog$. S njome možete naći SHA1 "izgubljenog *commit*a" i napraviti `cherry-pick` u tekuću granu ili novi branch.

Manje korištene naredbe

U ovom poglavlju ćemo proći neke rijeđe korištene naredbe gita. Neke od njih ćete koristiti jako rijetko, a neke možda i nikad. Zato nije ni potrebno da ih detaljno razumijete, važno je samo da znate da one postoje. Ovdje ćemo ih samo nabrojati i generalno opisati čemu služe, a ako zatrebaju – lako ćete saznati kako se koriste s `git help`.

Filter-branch

Naredba s kojom možemo promijeniti cijelu povijest projekta. Na primjer, *commit*ali smo u projekt s našom privatnom email adresom, i sad bismo htjeli promijeniti sve naše *commit*ove tako da sadrže službeni email. Slično, možemo mijenjati datume *commit*ova, dodati datoteke ili obrisati datoteke iz *commit*ova, isl.

Trebamo imati na umu da tako promijenjeni repozitorij ima različite SHA1 stringove *commit*ova. To znači da, ako naredbu primijenimo na jednom repozitoriju, drugi distribuirani repozitorij istog projekta **više neće imati zajedničku povijest s našim**.

Najbolje je to učiniti na našem privatnom repozitoriju kad smo sigurni da nitko drugi nema klon repozitorija ili ako na projektu radimo s točno određenim krugom ljudi. U ovom drugom slučaju – dogovorimo se s njima da svi iz*commit*aju svoje grane u naš repozitorij, izvršiti ćemo `git filter-branch` i nakon toga zamolimo ih da sad obrišu i iznova kloniraju repozitorij.

Shortlog

`git shortlog` ispisuje rekapitulaciju *commit*ova prema autoru.

Format-patch

Koristi se kad šaljemo *patch* emailom[49]. Na primjer, napravili smo lokalno nekoliko *commit*ova i sad ih želimo poslati emailom vlasniku udaljenog projekta. S `git format-patch` ćemo pripremiti emailove sa svim potrebnim detaljima o *commit*ovima (odnosno naše *patch*eve).

Am

Radnja suprotna onome što radimo s `git format-patch`. U ovom slučaju smo **mi** oni koji smo primili *patch*eve emailom, i sad ih treba "pretvoriti" u *commit*ove. To se radi s `git am`.

Fsck

`git fsck` provjerava ispravnost nekog objekta ili cijelog repozitorija. Ukoliko nešto ne valja s SHA1 čvorovima (*commit*ovima) ili je repozitorij "koruptiran"[50] – ova naredba će naći sve nekonzistentnosti.

Instaweb

`git instaweb` pokreće jednostavno web sučelje za pregled povijesti repozitorija.

Name-rev

Pretpostavimo da je `e0d22c0608ca0867b501f4890b4155486e8896b8` *commit* u našem repozitoriju. Gitu je to dovoljno, ali svima nama bi puno više značilo da nam netko kaže "peti *commit* prije verzije `1.0`" ili "drugi *commit* nakon što smo *branch*ali granu `test`". Za to postoji `git name-rev`.

Na primjer, meni `git name-rev e0d22` ispisuje `manje-koristene-naredbe~6`, što znači da je to šesti *commit* prije kraja grane `manje-koristene-naredbe`.

[49] Čini mi se da je to danas jako rijedak običaj.

[50] Može se dogoditi, na primjer, ako je nestalo struje dok ste s repozitorijem radili neku radnju koja zahtijeva puno snimanja po disku.

Stash

Želite li se prebaciti u drugu granu, a imate tekućih izmjena, git vam to ponekad neće dopustiti. S `git stash` možete privremeno spremiti izmjene koje ste radili u nekoj grani. Kad se kasnije vratite na prvotnu granu, prethodno spremljene izmjene možete vratiti nazad.

Submodule

Sa `git submodule` možemo u svoj repozitorij dodati neki drugi repozitorij. Jednostavno, u neki direktorij će se klonirati cijeli taj "drugi" repozitorij, a naš repozitorij će točno upamtiti SHA1 od željenog podrepozitorija.

Treba napomenuti da je s najnovijom verzijom gita uvedena jedna slična (za neke scenarije i bolja) naredba: `git subtree`, ali to još nije ušlo u široku upotrebu[51].

Rev-list

Rev list za zadane *commit* objekte daje spisak svih *commit*ova koji su dostupni. Ovu naredbu ćete vjerojatno koristiti tek ako radite neki skriptu (ili git *plugin*), a vrlo rijetko direktno u komandnoj liniji.

[51] U jednoj od sljedećih verzija ove knjige će nova naredba vjerojatno dobiti cijelo poglavlje.

Dodaci

Git hosting

Projekt na kojem radi samo jedna osoba je jednostavno organizirati. Ne trebaju udaljeni repozitoriji. Dovoljno je jedno računalo i neki mehanizam snimanja sigurnosnih kopija `.git` direktorija. Radimo li s drugim programerima ili možda imamo ambiciju kod našeg projekta pokazati svijetu – tada nam treba repozitorij na nekom vidljivijem mjestu.

Prvo što se moramo odlučiti je – hoće li taj repozitorij biti na našem serveru ili ćemo ga *host*ati na nekom od postojećih javnih servisa. Ukoliko je u pitanju ovo drugo, to je jednostavno. Većina ozbiljnih servisa za za hostanje projekata podržava git. Ovdje ću samo nabrojati neke od najpopularnijih:

- GitHub (`http://github.com`) – besplatan za projekte otvorenog koda, košta za privatne projekte (cijena ovisna o broju repozitorija i programera). Najpopularniji, brz i pregledan.
- BitBucket (`http://bitbucket.org`) – besplatan čak i za privatne repozitorije, malo manje popularan. U početku je bio zamišljen samo za projekte na mercurialu, ali sad nudi mercurial i git.
- Google Code (`http://code.google.com`) – također ima mogućnost hostanja na gitu. Samo za projekte otvorenog koda.
- Sourceforge (`http://sourceforge.net`) – jedan od najstarijih takvih servisa. Isključivo za projekte otvorenog koda.
- Codeplex (`http://www.codeplex.com`) – Microsoftova platforma za projekte otvorenog koda. Iako oni "guraju" TFS – vjerojatno im je postalo očito da je git danas *de facto* standard za otvoreni kod.

Za privatne repozitorije s više članova, moja preporuka je da platite tih par dolara Githubu ili BitBucketu. Osim što dobijete vrhunsku uslugu – tim novcem implicitno subvencionirate hosting svim ostalim projektima otvorenog koda koji su hostani tamo.

Vlastiti server

Druga varijanta je koristiti vlastiti server. Najjednostavniji scenarij je da jednostavno koristimo ssh protokol i postojeći korisnički račun na tom serveru. Treba samo u neki direktorij na rom računalu postaviti git repozitorij.

Ako je naziv servera `server.com`, korisničko ime s kojim se prijavljujemo `git`, a direktorij s repozitorijem `projekti/abc/`, onda ga možemo početi koristiti s:

```
git remote add moj-repozitorij git@server.com:projekti/abc
```

Naš projekt se u tom slučaju vjerojatno nalazi unutar korisnikovog "home" direktorija. Dakle, vjerojatno je putanja do našeg repozitorija na tom računalu: `/home/korisnik/projekti/abc`.

To će vjerojatno biti dovoljno za jednog korisnika, no ima nekih nedostataka.

Ukoliko želimo još nekome dati mogućnost da *push*a ili *fetch*a na/s našeg repozitorija. Moramo mu dati i sve potrebne podatke da bi ostvario ssh konekciju na naš server. Ukoliko to učinimo, on se može povezati *ssh*om i raditi sve što i mi, a to ponekad ne želimo.

Drugi problem je što ne možemo jednostavno nekome dati mogućnost da *fetch*a i *push*, a nekome drugome da samo *fetch*a. Ako smo dali ssh pristup – onda je on punopravan korisnik na tom serveru i ima iste ovlasti kao i bilo tko drugi tko se može prijaviti kao taj korisnik.

Git *shell*

Git shell riješava prvi od dva prethodno spomenuta problema. Kao što (pretpostavljam) znamo, na svakom UNIXoidnom operacijskom sustavu korisnici imaju definiran *shell*, odnosno program u kojem mogu izvršavati naredbe.

Git shell je posebna vrsta takvog *shell*a koja korisniku **omogućuje ssh pristup,**

ali i korištenje samo određenom broja naredbi. Postupak je jednostavan, treba kreirati novog korisnika (u primjeru koji slijedi, to je korisnik `git`). Naredbom:

```
chsh -s /usr/bin/git-shell git
```

... mu se početni *shell* mijenja u `git-shell`. I sad u njegovom *home* direktoriju treba kreirati direktorij `git-shell-commands` koji sadrži samo one naredbe koje će se ssh-om moći izvršavati. Neke distribucije Linuxa će već imati predložak takvog direktorija kojeg treba samo kopirati i dati prava za izvršavanje datotekama. Na primjer:

```
cp -R /usr/share/doc/git/contrib/git-shell-commands /home/git/
chmod +x /home/git/git-shell-commands/help
chmod +x /home/git/git-shell-commands/list
```

Sad, ako se netko (tko ima ovlasti) pokuša spojiti s *ssh*om, moći će izvršavati samo `help` i `list` naredbe.

Ovakav pristup ne riješava problem ovlasti čitanja/pisanja nad repozitorijima, on vam samo omogućuje da ne dajete prava klasičnog korisnika na sustavu.

Certifikati

S obzirom da je najjednostavniji način da se git koristi preko ssh, praktično je podesiti certifikate na lokalnom/udaljenom računalu tako da ne moramo svaki put tipkati lozinku. To se može tako da naš javni ssh certifikat kopiramo na udaljeno računalo.

U svojem *home* direktoriju bi trebali imati `.ssh` direktorij. Ukoliko nije tamo, naredba:

```
ssh-keygen -t dsa
```

... će ga kreirati zajedno s javnim certifikatom `id_rsa.pub`. Kopirajte sadržaj te datoteke u `~/.ssh/authorized_keys` na udaljenom računalu.

Ako je sve prošlo bez problema, korištenje gita preko ssh će od sad na dalje ići bez upita za lozinku za svaki *push*, *fetch* i *pull*.

Git *plugin*

Ukoliko vam se učini da je skup naredbi koje možemo dobiti s `git <naredba>` limitiran – lako je dodati nove. Recimo da trebamo naredbu `git gladan-sam`[52]. Sve što treba je snimiti negdje izvršivu datoteku `git-gladan-sam` i potruditi se da je dostupna u komandnoj liniji.

Na unixoidnim računalima, to bi izgledalo ovako nekako:

```
mkdir moj-git-plugin
cd moj-git-plugin
touch git-gladan-sam
# Tu bi sad trebalo editirati skriptu git-gladan-sam...
chmod +x git-gladan-sam
export PATH=$PATH:~/moj-git-plugin
```

Ovu zadnju liniju ćete, vjerojatno, dodati u neku inicijalizacijsku skriptu (`.bashrc`, isl.) tako da bude dostupna i nakon restarta računala.

Git i Mercurial

Mercurial je distribuirani sustav za verzioniranje sličan gitu. S obzirom da su nastali u isto vrijeme i bili pod utjecajem jedan drugog – imaju slične funkcionalnosti i terminologiju. Postoji i *plugin* koji omogućuje da naredbe iz jednog koristite u radu s drugim[53].

Mercurial ima malo konzistentnije imenovane naredbe, ali i značajno manji broj korisnika. Ukoliko vam je git neintuitivan, mercurial bi trebao biti prirodna alternativa. Naravno, ukoliko uopće želite distribuirani sustav.

Ovdje ćemo proći samo nekoliko osnovnih naredbi u mercurialu, tek toliko da steknete osjećaj o tome kako je s njime raditi:

Inicijalizacija repozitorija:

[52] Irelevantno što bi ta naredba radila :)
[53] http://hg-git.github.com

```
hg init
```

Dodavanje datoteke `README.txt` u prostor predviđen za sljedeći *commit* (ono što je u gitu indeks):

```
hg add README.txt
```

Micanje datoteke iz indeksa:

```
hg forget README.txt
```

Commit:

```
hg commit
```

Trenutni status repozitorija:

```
hg status
```

Izmjene u odnosu na repozitorij:

```
hg diff
```

Premještanje i izmjenu datoteka je poželjno raditi direktno iz mercuriala:

```
hg mv datoteka1 datoteka2
hg cp datoteka3 datoteka 4
```

Povijest repozitorija:

```
hg log
```

"Vraćanje" na neku reviziju (*commit*) u povijesti (za reviziju "1"):

```
hg update 1
```

Vraćanje na zadnju reviziju:

```
hg update tip
```

Pregled svih trenutnih grana:

```
hg branches
```

Kreiranje nove grane:

```
hg branch nova_grana
```

Grana će biti stvarno i stvorena tek nakon prvog *commit*a.

Jedna razlika između grana u mercurialu i gitu je što su u prvome grane permanentne. Grane mogu biti aktivne i neaktivne, ali u principu one ostaju u repozitoriju.

Glavna grana (ono što je u gitu `master`) je ovdje `default`.

Prebacivanje s grane na granu:

```
hg checkout naziv_grane
```

*Merge*anje grana:

```
hg merge naziv_grane
```

Pomoć:

```
hg help
```

Za objašnjenje mercurialove terminologije:

```
hg help glossary
```

Terminologija

Mi (informatičari, programeri, IT stručnjaci i "stručnjaci", ...) se redovito služimo stranim pojmovima i nisu nam neobične posuđenice kao *mrđanje*, *brenčanje*, *ekspajranje*, *eksekjutanje*. Naravno, poželjno bi bilo koristiti alternative koje su više u duhu jezika, a da opet ne petjeramo s raznim *vrtoletima*[54], *čegrtastim velepamtilima*[55], *nadstolnim klizalima*[56], *razbubnicima*[57], *uključnicima*[58] i sl.[59] Izuzmemo li ove besmislice – izrazi u duhu jezika za neke termine i ne postoje. Mogao sam ih izmisliti za potrebe ove git početnice, ali...

Besmisleno je izmišljati nove riječi za potrebe priručnika koji bi bio uvod u git. Koristiti termine koje bih sam izmislio i paralelno učiti git bi, za potencijalnog čitatelja, predstavljao dvostruki problem – em bi morao učiti nešto novo, em bi morao učiti **moju** terminologiju drukčiju od one kojom se služe stručnjaci. A stručnjaci su odlučili – oni govore *fetch*anje (iliti *fečanje*) i *commit*anje (iliti *komitanje*).

Dodatni problem je i to što prijevod često **nije** ono što se na prvi pogled čini ispravno. OK, *branch*anje bi bilo "grananje", no *merge*anje **nije** "spajanje grana". Spajanjem grana bi rezultat bio jedna jedina grana, ali *merge*anjem – obje grane nastavljaju svoj život. I kasnije se mogu opet *merge*ati, ali ne bi se mogle još jednom "spajati". Jedino što se izmjene iz jedne preuzimaju i u drugu. Ispravno bi bilo "preuzimanje izmjena iz jedne grane u drugu", ali to zvuči nespretno da bi se koristilo u svakodnevnom govoru.

[54] Helikopter
[55] *Hard disk*
[56] Miš
[57] *Debugger*
[58] *Plugin*
[59] Pretpostavljam da će ovo čitati i govornici drugih varijanti ovih naših južnoslavenskih jezika. Pa čisto da znate, kod nas je devedesetih godina vladala opsesija nad time da bi svim stranim stručnim riječima trebali naći prijevode "u duhu jezika". Pa su tako nastale neke od navedenih riječi. Neke od njih su zaista pokušali progurati kao "službene", a druge su samo sprdnja javnosti nad cijelim tim "projektom".

Činjenica je da većina pojmova jednostavno nema ustaljen hrvatski prijevod[60]. Zato sam ih koristio na točno onakav način kako se one upotrebljavaju u (domaćem) programerskom svijetu.

Nakon malo eksperimentiranja odlučio sam sve pojmove koristiti u izvornom obliku, ali *ukošenim* fontom. Na primjer, *fast-forward merge*, *merge*anje, *merge*ati, *fetch*ati, *fetch*anje ili *commit*anje. Tako su jednostavno prepoznatljivi svima (i čistuncima koji traže strane riječi i onima koji traže ustaljene IT pojmove).

Ako se ne slažete sa ovakvim pristupom – slobodni ste napisati svoju knjigu sa *razbubnicima* i *čegrtastim velepamtilima*.

Popis korištenih termina

Svi termini su objašnjeni u knjizi, ali ako se izgubite u šumi *push*eva, *merge*va i *squash*eva – evo kratak pregled:

***Bare* repozitorij** je repozitorij koji nije predviđen da ima radnu verziju projekta. Njegov smisao je da bude na nekom serveru i da se na njega može *push*ati i s njega *pull*ati i *fetch*ati.

Bisect je binarno pretraživanje povijesti u potrazi za izmjenama koje su izazvale neku grešku.

Branch je grana.

Cherry-pick je preuzimanje izmjena iz samo jednog *commit*a druge grane.

Commit je spremanje izmjena na projektu u sustav za verzioniranje.

Čvor je *commit*, ali koristi se kad se povijest projekta prikazuje grafom.

Diff je pregled izmjena između dva *commit*a (ili dvije grane ili dva stanja iste grane).

Fast-forward je proces koji se događa kad vršimo *merge* dva grafa, pri čemu je zadnji čvor ciljne grana ujedno i točka grananja dva grafa.

Fetch je preuzimanje izmjena (*commit*ova) s udaljenog repozitorija na lokalni.

Log je pregled izmjena koje su se desile između *commit*ova u nekoj grani. Ili pregled izmjena između radne verzije i stanja u repozitoriju.

[60] Jedan od glavnih krivaca za to su predavači na fakultetima koji **ne** misle da je verzioniranje koda tema za fakultetske kolegije.

Indeks je "međuprostor" u kojeg spremamo izmjene prije nego što ih *commit*amo.

Pull je kombinacija *fetch*a i *merge*a. S njime se izmjene s udaljenog repozitorija preuzimaju u lokalnu granu.

Pull request je zahtjev vlasniku udaljenog repozitorija (na kojeg nemamo ovlasti *push*ati) da preuzme izmjene koje smo mi napravili.

Push je "slanje" lokalnih *commit*ova na udaljeni repozitorij.

Radna verzija repozitorija je stanje direktorija našeg projekta. Ono može i ne mora biti jednako zadnjem snimljenom stanju u grani repozitorija u kojoj se trenutno nalazimo.

Rebase je proces kojim točku grananja jednog grafa pomičemo na kraj drugog grafa.

Referenca je informacija na osnovu koje možemo jedinstveno odrediti neki *commit* ili granu ili *tag*.

Reset je vraćanje stanja repozitorija na neko stanje. I to **ne** privremeno vraćanje nego baš izmjenu povijesti repozitorija pri čemu se briše zadnjih nekoliko *commit*ova iz povijesti.

Revert je spremanje izmjene koja poništava izmjene snimljene u nekom prethodnom *commit*u.

Repozitorij je projekt koji je snimljen u nekom sustavu za verzioniranje koda. Repozitorij sadržava cijelu povijest projekta.

Staging area je sinonim za **indeks**.

Squash merge je *merge*, ali na način da novostvoreni čvor nema referencu na granu iz koje su izmjene preuzete.

Tag je oznaka iliti imenovana referenca na neki *commit*.

Zahvale

U pisanju ovog knjižuljka je sudjelovalo puno ljudi. Neki direktno – tako što su čitali radne verzije knjige, ispravljali greške i predlagali teme, a drugi indirektno – tako što su mi postavljali **teška** pitanja :) koja su me navela da knjigu proširujem s odgovorima na ista.

Abecednim redom:

Dalen Bernaca, Aleksandar Branković, Matija Brunčić, Damir Bulić, Petar Dučić, Aldina Duraković, Mario Đanić, Vedran Ilić-Dreven, Vladimir Klemo, Katarina Majetić, Marina Maresti Krajina, Damir Milotić, Milan Mimica, Namik Nuhanović, Davor Poldrugo, Vanja Radovanović, Ante Sabo, Marko Stipanov, Krešimir Šimatović, Karlo Šmid, Mario Žagar, ...

Ispričavam se ako sam nekog zaboravio.