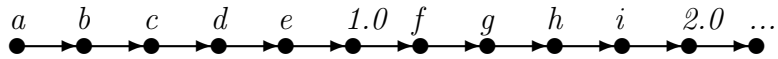
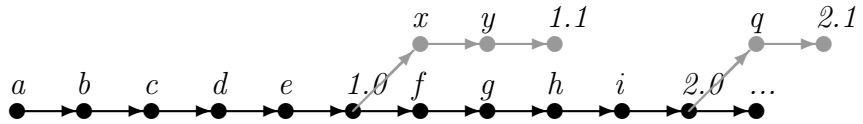
Uvod u git

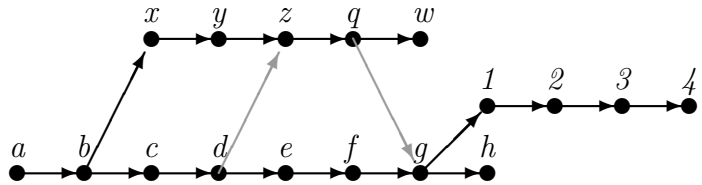
Od...



...prema...



...pa do...



...a bome i dalje od toga.

Commit `6465718576104d841db924691fef6f89b1194892`

Sadržaj

Uvod

Git je alat kojeg 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*. Dakle, drugi alati bi trebali razvijati svoje sučelje na osnovu gita. Međutim, kao s mnogim drugim projektima otvorenog koda, ljudi su ga počeli koristiti takvog kakav jest, on je organski rastao onako kako su ga ljudi koristili.

Rezultat je program koji ima drukčiju terminologiju i, nerijetko, malo neintuitivnu sintaksu. Usprkos tome, milijuni programera diljem svijeta su prihvatili git. Nastali su brojne platforme za *hosting* projekata, kao što je Github[1], a drugi su morali dodati git jednostavno zato što su to njihovi korisnici tražili (Google Code[2], Bitbucket[3], Sourceforge[4]).

Nekoliko je razloga zašto je to tako:

- Postojeći sustavi za verzioniranje su obično zahtijevali da se točno se zna tko sudjeluje u projektu (tj. tko je *comitter*). To je automatski demoraliziralo puno ljudi koji bi možda pokušali pomoći projektima kad mi imali priliku. S distribuiranim sustavima, bilo tko je mogao "forkati" repozitorij, isprogramirati izmjenu i predložiti vlasniku originalnog repozitorija da preuzme svoje izmjene. Rezultat je da je broj ljudi koji su potencijalni mogli pomoći projektu puno veći. A vlasnik projekta je ionako imao kontrolu nad time čije će izmjene prihvatiti, a čije neće.

- git je **brz**,

- vrlo je lako i brzo granati, isprobavati izmjene koje su radili drugi ljudi i preuzeti ih u svoj kod,

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

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

Format ove knjige

Ova knjiga nije zamišljena kao detaljan uvod u to kako git funkcionira i kao općeniti priručnik u kojem ćete tražiti riješenje svaki put kad negdje zapnete. Osnovna ideja mi je bila da se za svaku "radnju" s gitom opišem problem, ilustriram ga grafikonom, malo razradim teoriju, potkrijepim primjerima i onda opišem nekoliko osnovnih git naredbi koje se najčešće koriste. Uspijem li u tom naumu – nakon što pročitate knjigu, trebali biste biti sposobni git koristiti u svakodnevnom radu.

Zapnete li, a odgovora ne nađete ovdje, pravac Stackoverflow[5], Google, forumi, blogovi i, naravno, `git help`.

Pretpostavke

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

- znate programirati u nekim programskom jeziku,
- poznajete princip funkcioniranja klasičnih sustava za verzioniranje koda (CVS, SVN, ...),
- ne bojite se komandne linije,
- poznajete osnove rada s unixoidnim operativnim sustavima.

Nekoliko riječi o zadnje dvije stavke. Iako git nije nužno ograničen na unix/linux operativne sustave, njegovo komandnolinijsko sučelje je tamo nastalo i drži se istih principa. Git nije nužno komandnolinijski alat, međutim mnoge iole složenije stvari je 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. No, ako želite pomoći – zahvalan sam vam. Evo nekoliko načina kako to možete učniti:

- Pošaljite email na `tkrajina@gmail.com`,
- *Twitt*nite na `@puzz`,

[5] http://stackoverflow.com

- *Fork*ajte i pošaljite *pull request* s ispravkom. Repozitorij s tekstom je na: `https://github.com/tkrajina/uvod-u-git`

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[6].

Ukoliko i uočite grešku, možda je ona već ispravljena u zadnjoj verziji. Na `https://github.com/tkrajina/uvod-u-git` uvijek možete naći zadnju verziju ove knjižice.

Naredbe i operativni 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[7]. Dakle, u primjerima ćemo koristiti naredbe koje se koriste na UNIX, OS X i linux operativnim sustavima. No, i za korisnike Microsoft Windowsa to ne bi trebao biti problem jer se radi o relativno malo broju naredbi kao što su `mkdir` umjesto `md`, `ls` umjesto `dir`, i slično.

[6]Na primjer ono što piše *Commit* b5af8ec79a7384a5a291d15d050fc932eb474e79. E, ovaj nerazumljivi dugi string meni olakšava traženje verzije za koju prijavljujete grešku.

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

Verzioniranje koda i osnovni pojmovi

Što je to verzioniranje koda?

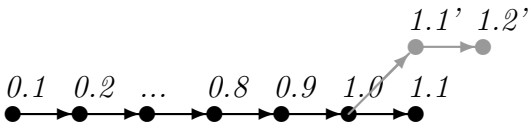
S problemom verzioniranja koda ste se sreli kad ste prvi put napisali program koji rješava neki konkretan problem. Bilo da je to neka jednostavna web aplikacija, CMS[8], komandnolinijski pomoćni programčić ili kompleksni ERP[9].

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 riješenje) 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 ju napraviti na licu mjesta. No, možda ćemo trebati otići kući, potrošiti nekoliko dana i napraviti **novu verziju**.

Desiti 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:

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

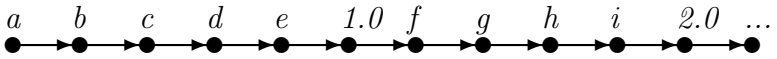


U trenutku kad je korisnik odlučio da mu trenutna verzija ne odgovara – trebamo ju "baciti u smeće", vratiti se jedan korak unazad u povijest projekta i započeti novu verziju – odnosno novu **granu projekta**. I nastaviti projekt s tom izmjenom.

I to je samo jedan od mnogih složenih scenarija kakvi se dešavaju s aplikacijama.

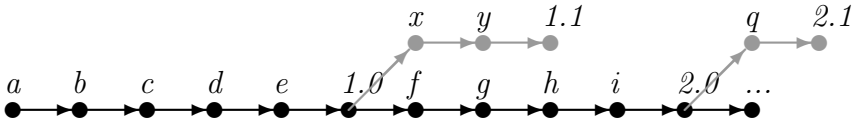
Linearno verzioniranje koda

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 nekim 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:



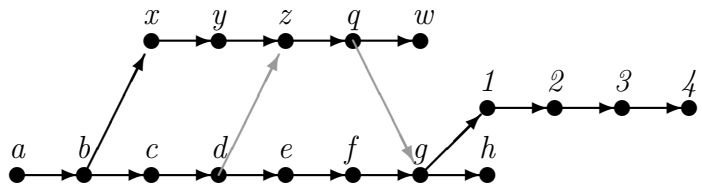
Ona je česta u situacijama kad nemate kontrolu nad time koja je točno verzija programa instalirana kod klijenta. S web aplikacijama to nije problem, jer vi jednostavno možete aplikaciju prebaciti nas server i odmah svi klijenti koriste novu verziju. Međutim, ukoliko je vaš program klijentima "spržen" na CD i takav poslan klijentu – može se desiti 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 preći 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, vratite se na zadnju verziju i tamo nastavite rad na svoj projektu.

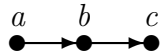
Grafovi, grananje i spajanje grana

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

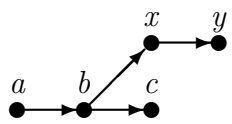


Svaka točka grafa je stanje projekta. Projekt s gornjim grafom je započeo s nekim početnim stanjem *a*. Programer je napravio nekoliko izmjena i **snimio** novo stanje *b*, zatim *c*, pa sve do *h*. Važno je napomenuti da je ovakav graf stanje povijesti projekta, ali iz njega ne možete zaključiti kojim redosljedom je nastao.

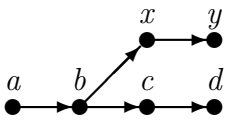
Evo, na primjer, jedan scenarij kako je navedeni graf mogao nastati:



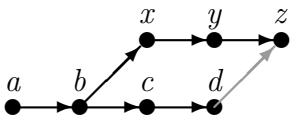
Programer je započeo aplikaciju, snimio stanje *a*, *b* i *c* i tada se sjetio da ima neki problem kojeg može riješiti na dva načina, vratio se na *b* i napravio novu granu. Tamo je napravio izmjene *x* i *y*:



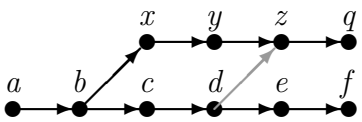
Zatim se sjetio jedne sitnice koju je mogao napraviti u *originalnoj* verziji; vratio se tamo i dodao *d*:



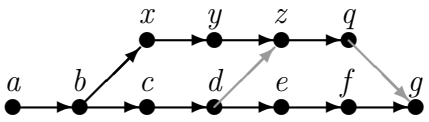
Nakon toga se vratio na svoj prvotni eksperiment, i odlučio se da bi bilo dobro tamo imati izmjene koje je napravio u *c* i *d*. Tada je *preuzeo* te izmjene u svoju granu:



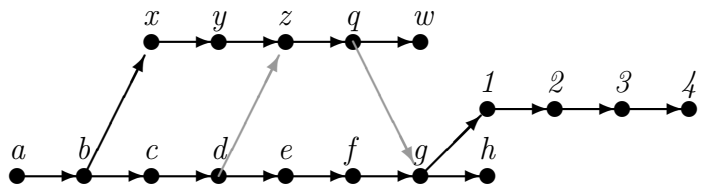
Na eksperimentalnoj grani je napravio još jednu izmjenu *q*. I tada je odlučio pustiti taj eksperiment malo sa strane i raditi malo na glavnoj grani. Vratio se na originalnu granu i tamo napredovao s *e* i *f*.



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



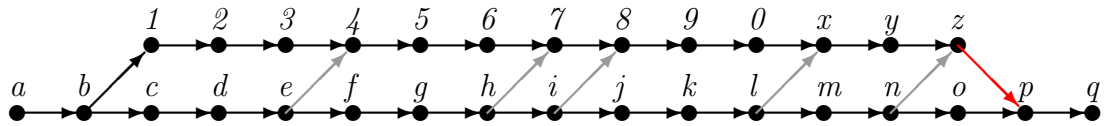
I, zatim je nastavio, stvorio još jednu eksperimentalnu granu(*1*, *2*, *3*, . . .). . . I tako dalje. . .



Uočite 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 ovako razmišljati i postupiti: *"Ovaj problem bih mogao riješiti na dva različita načina. Pokušati ću i jedan i drugi, i onda vidjeti koji mi bolje ide."*. Za svaku verziju će 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 želime 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 izmjene su na grafu označene sivim strelicama.

Kad novu funkcionalnost završimo, u glavnoj granu treba preuzeti sve izmjene iz sporedbne (crvena strelica).

Na taj način ćemo često imati ne samo dvije grane (glavnu i sporednu) nego nekoliko njih (pa i do nekoliko desetaka). 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** da vas uči kako je najbolje organizirati povijest projekta, odnosno kako granati, kad i kako preuzimati izmjene iz pojedinih grana. Osnovna ideja je da vas nauči **kako** to napraviti s gitom.

Mit o timu i sustavima za verzioniranje

Prije nego nastavimo, htio bih ovdje samo srušiti jedan mit. Taj mit glasi ovako nekako: "*Sustavi za verzioniranje koda su potrebni 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 često grananju. Ukoliko se natjerate da o projektu razmišljate kao o jednom usmjerenom grafu i tako posebne stvari radite u posebnim granama, to višestruko olakšava samo razmišljanje o projektu. Ako imate jedan direktorij sa cijelim projektom i u kodu imate paralelno izmjene od tri različite stvari koje radite istovremeno, onda imate problem. Ostatak ove knjižice bi vas trebao uvjeriti u to...

No, nemojte pročitati knjigu i reći "*Nisam baš uvjeren*". Probajte git[10] na nekoliko tjedana.

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

Instalacija, konfiguracija i prvi projekt

Instalacija

Instalacija gita je relativno jednostavna. Ukoliko ste na na nekom od linuxoidnih operativnih sustava – sigurno postoji paket za instalaciju. Za sve ostale, postoje jednostavne instalacije, a svi linkovi su dostupni na službenim web stranicama[11].

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" i omogućuju nove stvari.

Prvi git repozitorij

Ukoliko 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 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 samo odlučiti čije izmjene će 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.

No, idemo sad na prvi i najjednostavniji korak – stvoriti ćemo novi direktorij

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

`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.

Ukoliko idete pogledati kakva se to čarolija desila u 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 se čuva 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 primjeru smo u našem direktoriju inicijalizirali git repozitorij s naredbom `git init`. Općenito, git naredbe uvijek imaju sljedeći format:

```
git <komanda> <opcija1> <opcija2> ...
```

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

Za svaku git naredbu možemo dobiti *help* s:

```
git help <komanda>
```

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

Osnovna konfiguracija

Ima nekoliko osnovnih stvari koje moramo konfigurirati da bi ste 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 način će *email* adresa `ana.anic@poslodavac.hr` figurirati 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 do **konflikta** (o tome više

kasnije). Ja koristim `gvimdiff`:

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

.gitignore

Prije ili kasnije će se dogoditi 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 klase koje nastaju kompajliranjem (`.class` za javu, `.pyc` za python, `.o` za C, isl.). 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 unesimo sve ono što ne treba biti dio povijesti projekta.

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

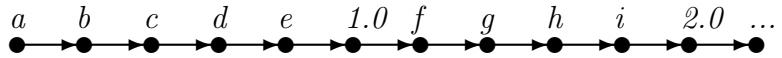
```
# Vim privremene datoteke:
*.swp
*.swo
# Java kompajlirane klase:
*.class
# Output direktorij s rezultatima kompajliranja i builda:
target/*
```

Sve one linije koje započinju sa znakom `#` su komentari i git će se ponašati kao da ne postoje.

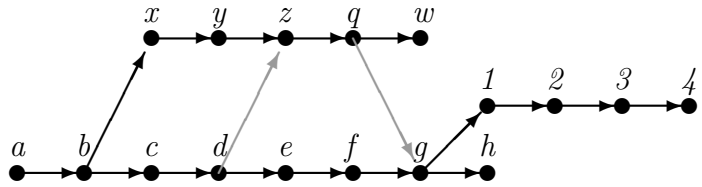
...i sad smo spremni početi nešto i raditi s našim projektom...

Spremanje izmjena

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



...i primjer s granama:



U oba slučaja, čvorovi tih grafova su stanje projekta u nekom trenutku. Na primjer, kad smo prvi put inicirali projekt s `git init`, dodali smo nekoliko datoteka i *spremili ih*. U tom trenutku je nastao čvor *a*. Nakon toga smo možda izmijenili neke od tih datoteka, možda neke obrisali, neke nove dodali i opet – spremili novo stanje i dobili stanje *b*.

To što smo radili između svaka dva stanja je naša stvar i ne tiče se gita[12]. No, 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, odnosno spremamo u **lokalni** repozitorij na našem računalu. Interakcija s udaljenim repozitorijem će biti tema poglavlja o udaljenim repozitorijima.

[12]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.

Status

Da bismo provjerili imamo li uopće nešto za spremati, koristi se naredba `git status`. Na primjer, kad na projektu koji nema lokalnih izmjena za spremanje utipkate `git status`, dobiti 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` i 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 podatak su linija u kojima piše `modified: README.tex`, jer to je datoteka koju 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 jednog 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',
```

Ako vam ovo izgleda zbunjujuće – postoji i način kako da se to ljepše prikaže, no, čisto

za vježbu, nije loše pokušati interpretirati ispis od `git diff`. Najvažniji dijelovi su linije oni koji počinju si `diff`, jer one govore o kojim datotekama se radi. Nakon njih slijedi nekoliko linija s općenitim podacima i zatim kod *oko* dijela koji je izmijenjen i onda ono najvažnije – linije obojane u crveno i one obojane u plavo.

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

Rezultat `diff` naredbe su samo dijelovi koje ste izmijenili i nekoliko linija *oko njih*. Ukoliko želimo malo veću tu "okolinu" oko naših izmjena, možemo ju izmijeniti s opcijom `-U<broj_linija>`. Na primjer, ukoliko želimo 10 linija oko izmjenjenih dijelova koda, to ćemo dobiti sa:

```
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 ne radi s datotekama nego sa stanjima (ili verzijama) datoteka. Dakle, za jednu te istu datoteku – git čuva njena različita stanja.

U gitu postoji poseban "međuprostor" u koji se "stavljaju" datoteke koje ćemo spremiti (*commit*ati). U odnosu na taj indeks i repozitorij općenito, datoteka (odnosno neko njeno stanje) može biti:

- datoteka je već spremljena negdje u git repozitoriju. U tom repozitoriju čuvamo sve prethodnde verzije te iste datoteke.
- datoteku smo tek izmijenili i nismo ju još *commit*ali u repozitorij. Ta datoteka može, ali i ne mora, imati prethodne verzije snimljene u repozitoriju.
- datoteku smo stavili u taj "međuprostor" tako da bismo se pripremili za *commit*.

Ovo zadnje stanje, odnosno, taj "međuprostor za *commit*" se zove *index* iliti indeks. U literaturi ćete često naći i naziv *staging area* ili *cache*[13]. I naredba `git status` je upravo namijenjena pregledavanju statusa indeksa. Na primjer, u trenutku pisanja ovog poglavlja,

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

`git status` je[14]:

```
$ 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:   git-commit.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.

Ukoliko 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 datoteke su izmijenjene, a na nama je da sad u indeks stavimo (samo) one datoteke koje ćemo u sljedećem koraku *commit*ati.

Spremanje u indeks

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

```
git add git-commit.tex
```

. . . i sad je status:

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

Primijetiti ćete dio u kojem piše, `Changes to be committed` – e to je spisak datoteka koje

[14] Da, i ova knjiga je pisana koristeći git. Detalji na https://github.com/tkrajina/uvod-u-git

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

ste stavili u indeks. Dakle, datoteku smo spremili u indeks i sad smo spremni 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 ju 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").

Micanje iz indeksa

Ako smo neku datoteku stavili u indeks i kasnije se predomislili. Sad tu datoteku želimo izbaciti iz indeksa, ali bez da gubimo izmjene koje smo na njoj napravili jer će one biti dio nekog sljedećeg *commit*a. Ili ih jednostavno ne želimo u povijesti projekta (i datoteku ćemo resetirati na prethodno stanje). To se može naredbom:

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

Nakon toga, izmjene treba *commit*ati.

Događati će nam se da smo promijenili neku datoteku, no kasnije se ispostavilo da ta izmjena nije bila potrebna. I sad ju ne želite spremiti nego vratiti u prethodno stanje. To se može ovako:

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

Više detalja o `git checkout` i zašto ta gornja naredba radi to što radi će biti 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:   git-commit.tex
#
```

No, ako sad tu datoteku izmijenimo direktno u projektu – novo stanje će biti ovakvo:

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

Sad tu datoteku imamo izmijenjenu i u radnoj verziji (direktorij kakav je trenutno na disku) i u indeksu. I, u tom smislu, nije ništa neobično da imamo jedno stanje datoteke u indeksu, a drugo stanje datoteke u radnoj verziji našeg projekta. Indeks (i git općenito) ne sprema datoteke nego **stanja datoteka**.

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

```
git add <datoteka>
```

Dakle, ukratko, indeks je prostor u kojeg spremamo grupu datoteka (odnosno **stanja** datoteka). Takav skup datoteka treba predstavljati neku logičku cjelinu tako da bi ih mogli 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. Sve dok nismo sigurni da indeks predstavlja točno one datoteke koje želimo u našoj izmjeni (*commit*u).

Razlika između tog, novog, čvora i njegovog prethodnika su upravo datoteke koje smo imali u indeksu u trenutku kad smo *commit* izvršili.

Prvi commit

Izmjene možemo spremiti s:

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

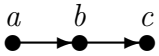
U stringu nakon `-m` **moramo** unijeti komentar uz svaku promjenu koju spremamo u repozitorij. Git ne dopušta spremanje izmjena bez komentara.

Sad je status projekta opet:

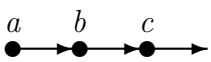
```
$ git status
# On branch master
nothing to commit (working directory clean)
```

Indeks i *commit* grafički

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

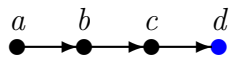


To znači da je stanje projekta u direktoriju potpuno isti kao i stanje projekta u zadnjem čvoru našeg git grafa. Nakon toga smo u direktoriju izmijenili nekoliko datoteka:



To znači, napravili smo nekoliko izmjena, no one još nisu dio repozitorija (zapamtite, samo čvorovi u grafu su ono što git čuva u repozitoriju).

Nakon toga smo izmijenjene datoteke spremili u indeks i *commit*ali, i sad je stanje projekta:



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

Datoteke koje želimo u repozitoriju

Jedan scenarij koji se često događa je sljedeći: 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 je, na primjer, situacija 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 ju 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 ju obrisati s diska, ali ne želimo ju 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 vam u indeks dodati kao da je datoteka obrisana, iako ju ostavlja netaknutu na disku. Sad tu izmjenu treba *commit*ati

Budući da smo datoteku prethodno dodali u `.gitignore`, git nam ubuduće nuditi da ju *commit*amo. Odnosno, što god radili s tom datotekom, `git status` će se ponašati kao da ne postoji.

Povijest projeka

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

...
```

Za sada je važno znati da u gitu svaki *commit* ima jedinstveni string koji ga identificira. Taj string ima 40 znamenaka i primjere možemo vidjeti u rezultatu od `git log`. Na primjer, `bf4fc495fc926050fb10260a6a9ae66c96aaf908` je jedan takav.

Više riječi o povijesti repozitorija će biti u posebnom poglavlju.

Ispravljanje zadnjeg *commit*a

Dogoditi će se da spremite neku izmjenu u repozitorij, a nakon toga shvatite da ste mogli još jednu sitnicu ispraviti. I, nekako vam se čini da bi bilo logično da ta sitnica bude dio

prethodnog *commit*a. Bilo bi lijepo, pomislit' ćete, kad biste mogli izmijeniti zadnji *commit* tako da sadrži i ovu novu, sitnu, ispravku koju biste dodali.

S gitom se to može. Prvo učinimo tu izmjenu. Zatim recimo da je ona bila na datoteci `README.md`. Dodamo tu datoteku u indeks s `git add README.md` kao da se spremamo napraviti još jedan *commit*. No, umjesto `git commit`, sad je naredba:

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

Ovaj `--amend` gitu naređuje da promijeni zadnji *commit* u povijesti tako da sadrži **i izmjene koje je već imao i izmjene koje smo upravo dodali**. Možemo provjeriti s `git log` šte se desilo i vidjeti ćemo da zadnji *commit* sad ima novi komentar. U stvari, ispravno bi bilo kazati da je promijenjen i cijeli taj *commit*.

`git commit --amend` nam omogućava da u zadnji *commit* dodamo neku datoteku ili čak i *maknemo* datoteku koju smo prethodno *commit*ali. No, 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.

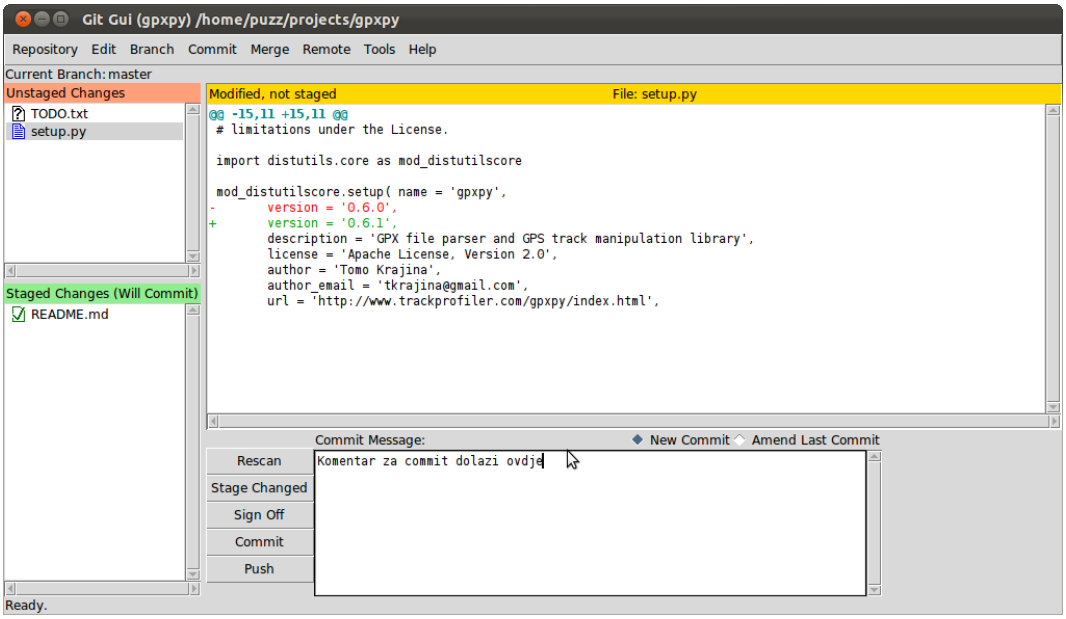
Git gui

Kad spremamo neku izmjenu koja ima 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:



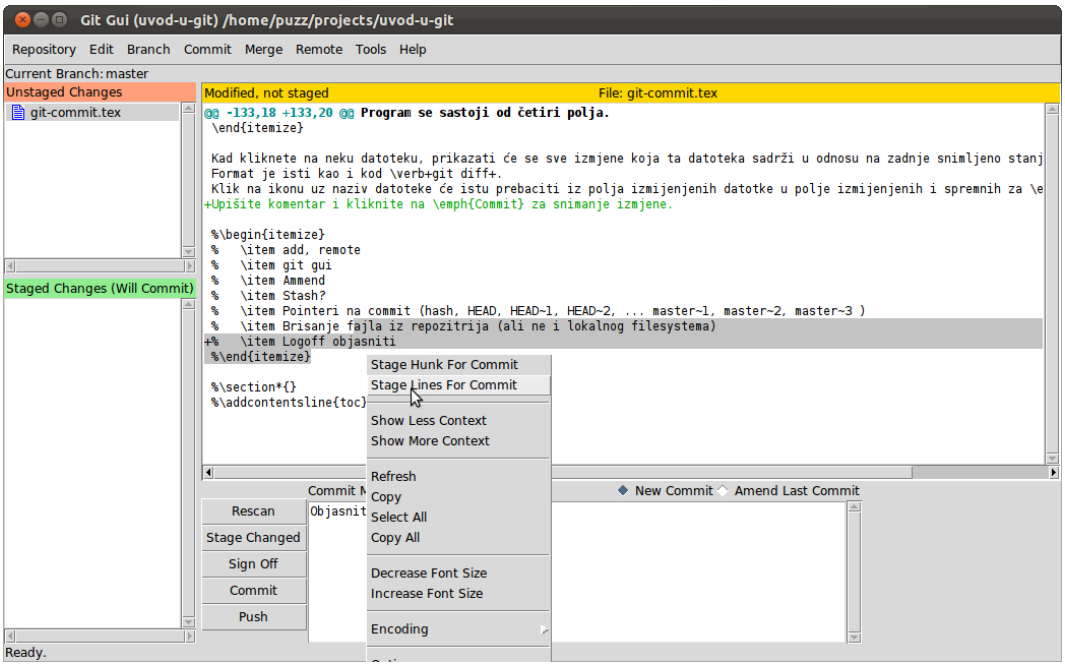
Program se sastoji od četiri polja.

- Polje za datoteke koje su izmijenjene, ali nisu još u indeksu (gore lijevo).
- Polje za prikaz izmjena u pojedinim datotekama (gore desno).
- Polje za datoteke koje su izmijenjene i stavljene su u indeks (dolje lijevo).
- Polje za *commit* (dolje lijevo).

Klik na neku datoteku će prikazati 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 će istu prebaciti iz polja izmijenjenih datotka u polje s indeksom i suprotno. Nakon što odaberemo datoteke za koje želimo da budu dio našeg *commit*a, 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 dodati nekoliko datoteka, upisati komentar, snimiti i nakon toga dodati sljedećih nekoliko datoteka, opisati novi komentar i snimiti sljedeću izmjenu. Drugim riječima, izmjene možete 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 spremite, desni klik i:

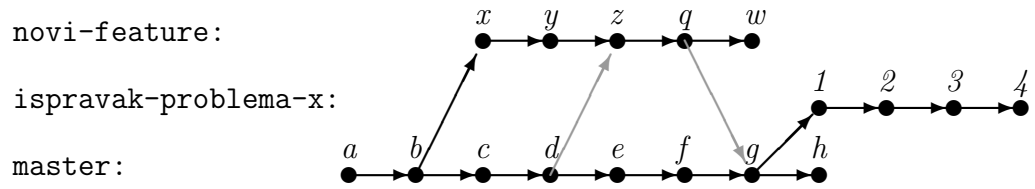


Ako smo na nekoj datoteci napravili izmjenu koju *ne* želimo snimiti – takvu datoteku možemo resetirati, odnosno vratiti u početno stanje. Jednostavno odaberemo ju u meniju `Commit` → `Revert changes`.

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 tastaturom u radu, jer to će vam značajno ubrzati daljnji rad.

Grananje

Još jednom ćemo početi s ovim, već viđenim, grafom:



Ovaj put s jednom izmjenom, svaka "grana" ima svoj naziv. U uvodnom poglavlju je opisan scenarij koji bi, otprilike, mogao dovesti do ovog grafa. Ono što je ovdje važno još jednom spomenuti je sljedeće; svaki čvor grafa je stanje projekta u nekom trenutku njegove povijesti. Svaka strelica iz jednog u drugi čvor je izmjena koju je programer napravio i snimio u nadi da će ona dovesti do željenog ponašanja aplikacije.

Spisak 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 slučajeva, rezultat te naredbe će 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`.

Ukoliko 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
```

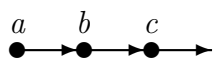
Svaki redak predstavlja jednu granu, a redak koji počinje sa zvjezdicom (*) je **grana u kojoj se trenutno nalazimo**. U toj grani tada možemo raditi sve što i na `master` – commitati, gledati njenu povijest, ...

Nova grana

Ukoliko je trenutni ispis komande `git branch` ovakav:

```
$ git branch
* master
```

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



I sad se može desiti da nakon stanja c želimo isprobati dva različita pristupa. Novu granu

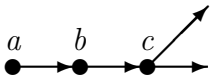
možemo stvoriti naredbom `git branch <naziv_grane>`. Na primjer:

```
git branch eksperimentalna-grana
```

Sad je novo stanje projekta:

`eksperimentalna-grana:`

`master:`



Prebacivanje s grane na granu

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

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

Naime, `git branch` će nam sam stvoriti novu granu. Prebacivanje s jedne grane na drugu granu se radi s komandom `git checkout <naziv_grane>`:

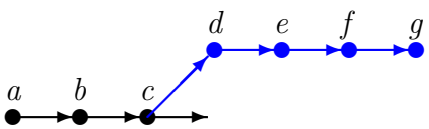
```
$ git checkout eksperimentalna-grana
Switched to branch 'eksperimentalna-grana'
```

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

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

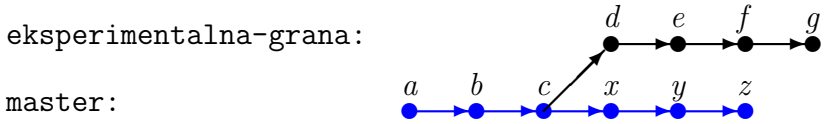
`eksperimentalna-grana:`

`master:`



I, kad god želimo, možete 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 nam neće 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 je, u većini ostalih sustava, *proces* grananja u stvari podradzumijeva **kopiranje svih datoteka** na mjesto gdje se čuva nova grana. To, em traje neko vrijeme, em zauzima više memorije na diskovima.

Kod gita to je puno jednostavnije, kad kreiramo novi 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.

Brisanje grane

Zato što je grananje memorijski nezahtjevno i brzo, pripremite se na situacije kad ćete se naći s **previše** grana. Možda smo neke grane napravili da bi isprobali nešto novo, a to se na kraju pokazalo kao loša ideja pa smo granu napustili. Ili smo neku granu kreirali da bi započeli nešto novo u aplikaciji, ali na kraju je to riješio netko drugi ili se pokazalo da to nije potrebno.

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`: s

```
$ git branch -D master
error: Cannot delete the branch 'ma
```

I to vrijedi općenito – ne možete obrisati granu na kojoj se trenutno nalazimo.

Preuzimanje datoteke iz druge grane

S puno grana, događati će se svakakvi scenariji. Jedan od relativno čestih je situacija kad bismo htjeli preuzeti samo jednu ili više datoteka iz druge grane, ali ne želite **preći** na tu drugu granu. Znamo da su neke datoteke u drugoj grani izmijenjeni, i želimo ih u trenutnoj grani. To se može ovako:

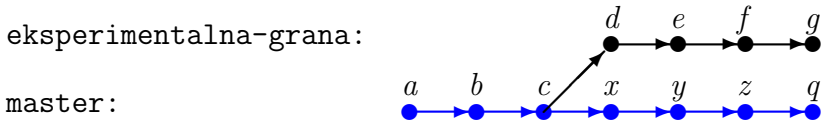
```
git checkout <naziv_grane> -- <datoteka1> <datoteka2> ...
```

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

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

Preuzimanje izmjena iz jedne grane u drugu

Vratimo se opet na jednu 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. No, 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. No, *merge*anjem dvije grane – one nastavljaju svoj život. Jedino što se sve izmjene koje su do tog trenutka rađene u jednoj granu preuzimaju u drugoj grani.

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 kako **trebamo se nalaziti u onoj grani**

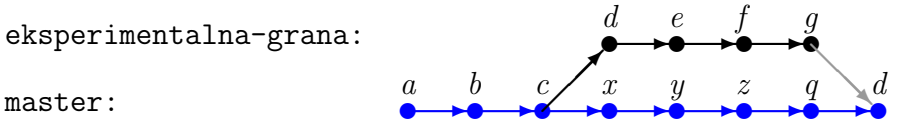
u koju želimo preuzeti izmjene (u našem slučaju `master`). Tada:

```
$ 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
```

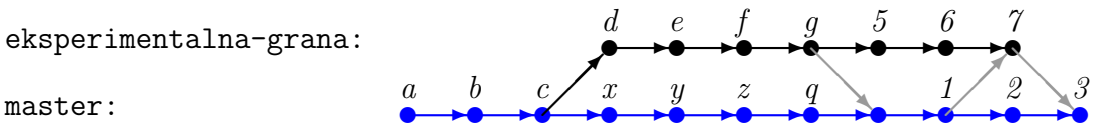
Rezultat neredbe `git merge` ukratko opisan rezultat procesa preuzimanja izmjena; koliko je linija dodano, koliko obrisano, koliko je datoteka dodano, koliko obrisano, i tako dalje...

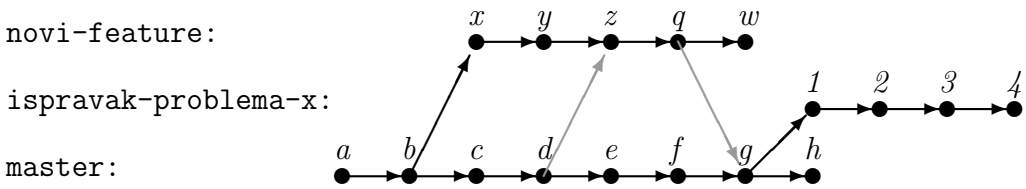
Još nešto je važno napomenuti – ako je `git merge` proveden bez grešaka, to automatski dodaje novi *commit* u grafu. Ne moramo "ručno" *commit*ati. Za razliku od svih ostalih *commit*ova, ovaj ima dva "roditelja" iliti "prethodnika" – jednog iz grane u kojoj se nalazi i drugog iz grane iz koje su izmjene preuzete.

Grafički se `git merge` može prikazati ovako:



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, isl. Kasnije možemo i ponoviti preuzimanje izmjena, odnosno *merge*anje. I, eventualno, jednog dana kad odlučimo da nam grana više ne treba, možemo ju obrisati.





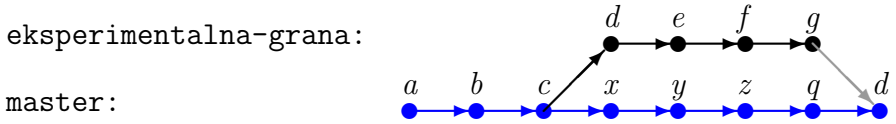
Što *merge* radi kad...

Vjerojatno se pitate što git radi u raznim mogućim situacijama. Na primjer u jednog grani smo dodali datoteku, a u drugoj editirali, ili...

U principu, vjerujte mi, git najčešće napravi točno ono što treba. 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. Svaki put sam išao pogledati je li napravio ono što bih očekivao da će napraviti i provjeravao bih da mi nije možda pregazio neku datoteku ili važan komad koda. Rezultat tog neformalnog istraživanja je: Moja (ljudska) i gitova (mašinska) intuicija o tome što treba napraviti se skoro uvijek poklapaju.

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. Pokušati ću to ilustrirati tako da opišem jedan od gornjih slučajeva; što se desi kad...

Što se desi kad...

Vjerojatno to i sami slutite – 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 jedan malo vjerojatan 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...

Scenarij kao stvoren za git, nije li?

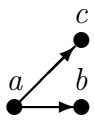
Recimo da je autor krenuo sa sljedećom verzijom pjesme:

```
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
```

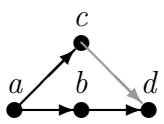
...dok je u drugoj varijanti (c) izmijenio zadnji stih:

```
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:

`varijanta:`

`master:`



... 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
```

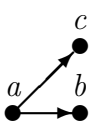
I to je jednostavno. U obje grane je mijenjao istu datoteku, ali u jednoj je dirao početak, a u drugoj kraj. I rezultat *merge*anja je bio očekivan – datoteka u kojoj je izmijenjen i početak i kraj.

Konflikti

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

`varijanta:`

`master:`



... 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* ovako:

```
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` ovakav:

```
$ 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 da *automatski* preuzme izmjene iz *b* u *a*. Idete li sad editirati

datoteku s pjesmom naći ćete ovakvo nešto:

```
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` iz druge grane.

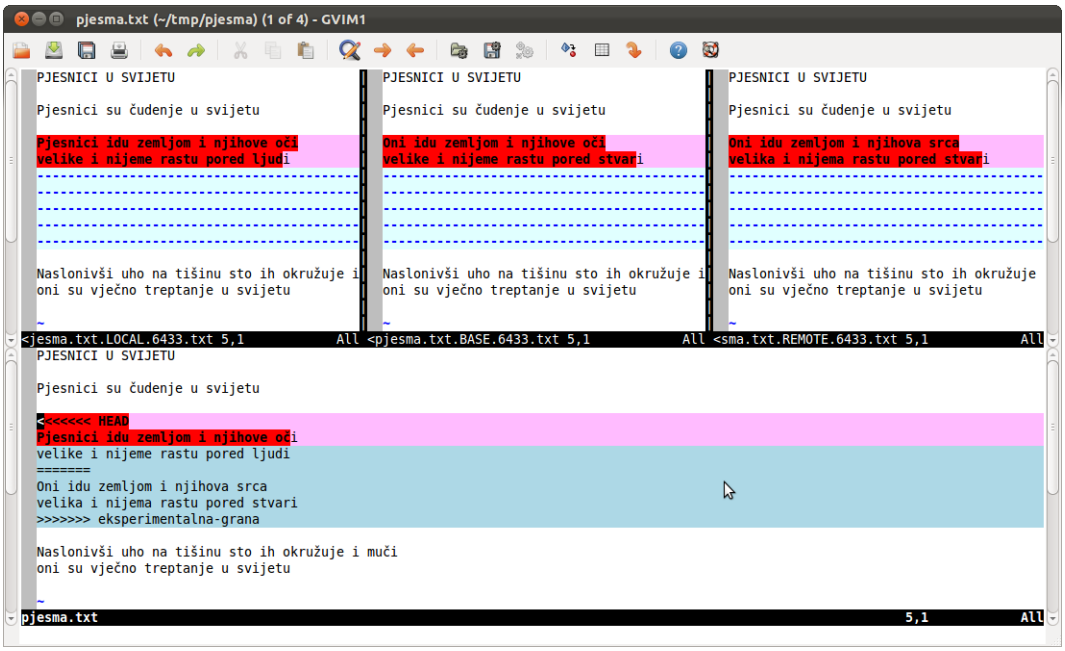
Za razliku od standardnog *merge*, ovdje niti jedna datoteka nije *commit*ana. To možete lako provjeriti sa `git status`:

```
$ 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 koristite `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. Iako to nije nužno (možete uvijek sami editirati konfliktne datoteke) – to je zgodno riješenje ako znate raditi s nekim od tih programa.

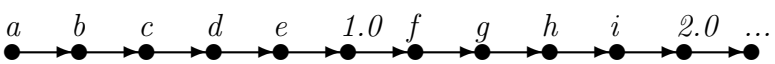
Na primjer, ako koristite vimdiff, onda editiranje konfliktnih datoteka izgleda ovako:



Merge, *branch* i povijest projekta

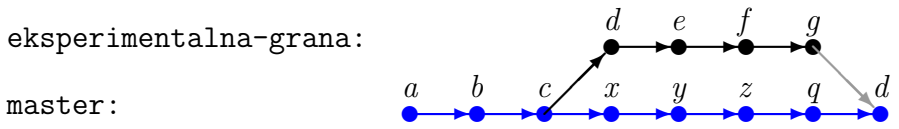
Kad radimo s nekim projektom, onda nam je uvijek važno da imamo sačuvanu povijest projekta. To nam omogućava da saznamo tko je originalni autor nekog koda kojeg možda slabije razumijemo. Ili želimo vidjeti kojim redosljedom je neka datoteka nastajala.

Sa standardnim sustavima za verzioniranje, stvar je jednostavna. Grananje i preuzimanje izmjena iz jedne u drugu granu je bilo komplicirano i rijetko se radilo. Posljedica je da su projekti *uvijek* 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 mnogi programeri grane koje više ne koriste brišu.

No, tu sad imamo mali problem. Pogledajte, na primjer, ovakav projekt:

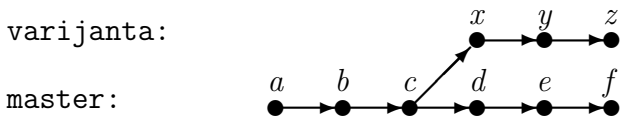


Eksperimentalna grana ima svoju povijest, a u trenutku *merge*a, sve izmjene iz te grane su se "prelile" u `master` i to u *commit d*. Kad bi obrisali `eksperimentalna-grana` – izgubili bi cijelu povijest te grane. Sve međukorake kako je ona nastajala (d, e, f i g), sve bi nestalo.

Nekome to nije problem, ali desiti će nam se situacija da to ne želimo. Drugim riječim, ne bismo htjeli da imamo previše grana (jer nam to otežava organizaciju projekta), a ne želimo gubiti povijest nastajanja svake od tih grana. 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*...

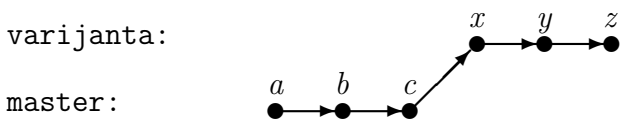
Fast forward

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



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 što 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).

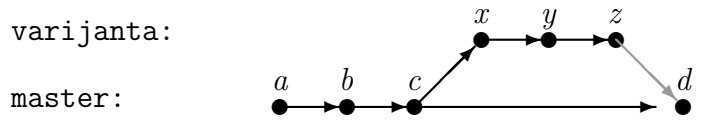
No, postoji jedan poseban slučaj koji zahtijeva pojašnjenje. Uzmimo da je povijest projekta bila slična gornjem grafu, ali s jednom malo izmjenom:



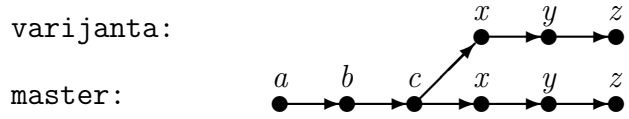
Programer je napravio novu granu `varijanta` i na njoj je nastavio rad. I svo to vrijeme nije radio nikakve izmjene na `master`. Što kad sad želi preuzeti sve izmjene u `master`?

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.

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*:



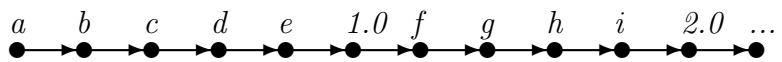
Dakle, doslovno kopira cijelu povijest (ovdje je to x, y i z) u novu granu. Čak i ako sad obrišete `varijanta`, cijela njega povijest se nalazi u `master`.

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

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

Rebase

Idemo, još jednom, pogledati linearni model:

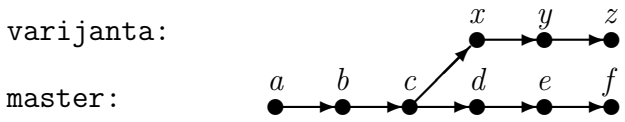


Do sada bi svima trebalo biti jasno da on ima svoje mane. No, ima i nešto korisno

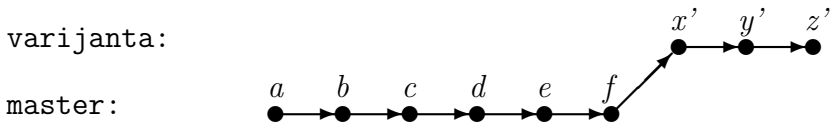
– jednostavna i pregledna povijest projekta i privid da je sve skupa teklo po nekom točno određenom rasporedu. Korak po korak, do trenutne verzije.

Git nas ne tjera da radite grane, no postupak grananja čini bezbolnim. I zbog toga povijest projekta *može* postati cirkus kojeg je teško pratiti i organizirati. Zbog toga, organizacija repozitorija zahtijeva dosta veliku pažnju.

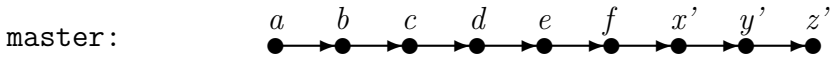
Postoji, ipak, način kako se može od puno grana stvoriti linearna povijest. Kad bi postojao trik kako da iz ovakvog stanja:



... stvorimo ovo:



To jest, da *pomaknemo mjesto od kud smo granali* neku granu. Tada bi *fast-forward* samo kopirao cijelu našu granu u `master`. Kad bi, tada, obrisali `varijanta`, povijest bi odjednom postala linerna:



Taj trik postoji i zove se *rebase*.

Radi se na sljedeći način; trebamo biti postavljeni u grani koju želite "pomaknuti". Zatim `git rebase <grana>`, gdje je `<grana>` ona grana na koju kasnije želite napraviti *fast-forward*.

Dakle, 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 ispis bi mogao izgledati ovako nekako:

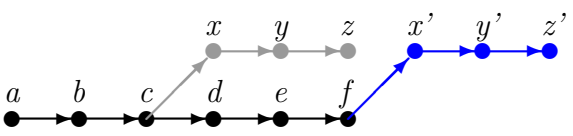
```
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: Prvi commit
Applying: Drugi commit
```

Međutim, ovdje mogu nastati problemi slični klasičnom *merge*u. Tako da će se češće 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 datoteke, vidjeti ćete da je format konfliktne datoteke isti kao kod konflikta pri *merge*u. I opet, od nas se očekuje da te konflikte riješimo. Bilo koristeći `git mergetool`, bilo tako da editiramo datoteku i ispravimo je u željeno stanje.

Promotrimo još jednom grafički prikaz onoga što ovdje pokušavamo napraviti:



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

Git će automatski napraviti koliko može. No, kad dođe konflikt (kao što je u primjeru kojeg trenutno obrađujemo), to trebamo napraviti sami.

Ako koristite `mergetool`, to ćete izvesti...

```
$ git mergetool
Merging:
pjesma.txt

Normal merge conflict for 'pjesma.txt':
  local: modified file
  remote: modified file
Hit return to start merge resolution tool (gvimdiff):
4 files to edit
```

Važno je napomenuti da, nakon što smo konflikt ispravili, **ne smijemo izmjene *commitati***, nego samo spremiti u indeks s `git add`. Nakon toga:

```
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 ovakav:

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

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

Rebase ili ne?

Rebase nije nužno raditi. Na vama je odluka želite li izmjene iz neke grane sačuvati u povijesti projekta ili prepustiti zaboravu (za trenutak kad i ako granu obrišete).

Ponekad će vam se dogoditi da sam *rebase* ima previše konflikata pa ćete odustati. U

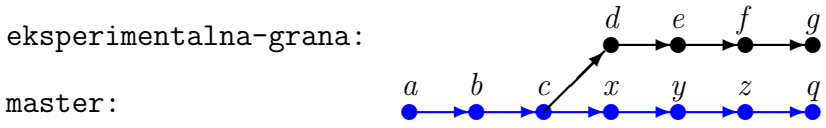
svakom slučaju, ukoliko ste ga i započeli, uvijek ga možete prekinuti s:

```
git rebase --abort
```

Cherry-pick

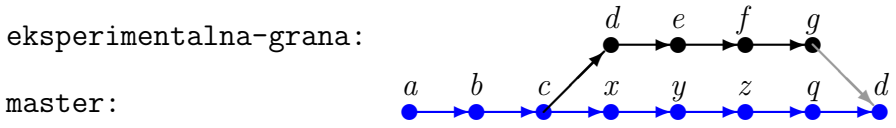
Ima još jedna posebna vrsta *merge*a, a nosi malo neobičan naziv *cherry-pick*[16].

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:



... ne dolazi u obzir, jer bi on u `master` prebacio i 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*.

[16] 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.

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 neki čudan string kao `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`.

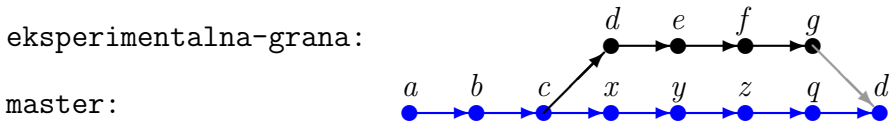
Prebacimo se u granu u koju želimo preuzeti samo izmjene it tog *commit*a i utipkajmo `git cherry-pick <commit>`. Ako nema konflikata, rezultat će biti:

```
$ 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 *commit*a

Vratimo se opet na klasični *merge*. Ukoliko je prošao bez ikakvih problema, onda ćemo nakon...



...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'
```

Vratimo se opet na cijelu onu priču oko *rebase*. Poanta cijele priče je bila u tome što, ako obrišemo granu, brišemo i cijelu njenu povijest. Ako se već odlučimo da ne želimo *rebase*, onda bi bilo lijepo u povijesti projekta umjesto `Merge branch 'eksperimentalna-grana'` imati smisleniji komentar koji bi bolje opisao što smo točno u toj grani napravili. Na taj način će, čak i ako granu obrišemo, ostati jedan commit u kojem je opisano što je taj *merge* riješio.

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 mi sami možemo spremiti sve izmjene i staviti neki svoj komentar.

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

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

Ukoliko ste zaboravili `--no-commit`, tekst poruke zadnjek *commit*a možete uvijek ispraviti

s *amend commit*om.

Tagovi

u *Tag*, "oznaka", iliti "ključna riječ" je naziv koji je populariziran s dolaskom takozvanim "web 2.0" sajtovima. Mnogi ne znaju, ali tagovi 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 redosljedu (na primjer, vremenu kako su stizale u knjižicu), a neki *software* bi za svaku pamtio:

- Ključne riječi (iliti *tag*ovi). Na primjer, za neki roman s Sherlockom Holmesom kao glavnim likom, to bi bili "ubojstvo", "krimić", "detektiv", "engleski", "sherlock_homes", "watson", ...
- Nekakav identifikator knjige (vjerojano ISBN).
- Mjesto gdje se knjiga nalazi.

Kad bi pretraživali knjige, otišli bi na računalo i utipkali "krimić" i "detektiv" i on bi nam izbacio sve knjige koje imaju oba ta *tag*a. Tu ne bi bili samo romani Artura Conana Doylea, našli bi se i romani Agathe Christie i mnogih drugih. Što više *tag*ova zadamo, to će rezultati pretraživanja biti više specifični. Uz svaku knjigu bi pisalo na kojoj točno polici se ona nalazi.

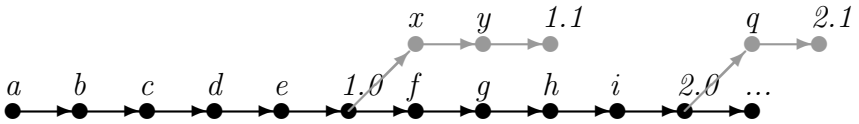
Kako mi ovdje radimo s poviješću projekata pa ćemo to i *tag*irati. Malo preciznije, *tag*irati ćemo čvorove našeg grafa povijesti projekta - *commit*ove. Postoji samo jedna razlika, *tag* ovdje mora biti jedinstven. Dakle, ako smo neki tag iskoristili za jedan *commit* onda niti jedan drugi ne smije imati taj isti *tag*.

Kao što znamo, u gitu svaki *commit* ima neki svoj identifikator. To je string od 40 znamenaka. 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 komentarom "Ispravljen bug s izračunom kamate".

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*, odnosno stanje projekta u nekom trenutku njegove povijesti.

Rad s *tag*ovima je jednostavan; s `git tag` ćete dobiti spisak 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 tesni-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.

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" projekata. Svaki direktorij predstavlja neko stanje projekta (dakle to je *commit*).

I to je nekakvo verzioniranje koda, no puno toga nedostaje.

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 nekakav graf, odnosno redosljed kako su direktorij nastajali jedan iz drugog. 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.

No, 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. To je gnjavaža.

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 jednostavno *shell* skriptu koja će to napraviti a 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 kopiranjem direktorija sa izmijenjenim datotekama. 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 hakerovu varijantu. Jeste li sigurni da bi primijetili podvaljenu datoteku?

Riješenje 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 sadržaja i koji jedinstveno određuje sadržaj. 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. Dakle, haker 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 projekta) onda mu je još teže – jer tu radnju mora ponoviti na računalima svih ljudi koji imaju kopije. S distribuiranim sustavima, nitko nikad niti ne zna tko sve ima kopije.

Nakon ovog početnog razmatranja, idemo pogledati koje od tih ideja su programeri gita uzeli kad su krenuli dizajnirati svoj sustav. Krećimo 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**[17].
- 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.

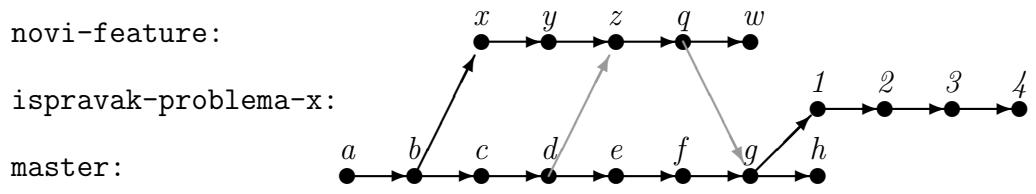
Git nije ništa drugo nego graf SHA1 stringova, od kojih svaki jedinstveno identificira neko stanje projekta **i izračunati su iz tog stanja**. Osim SHA1 identifikatora git uz svaki *commit* čuva i neke metapodatke kao, na primjer:

- Datum i vrijeme kad je nastao.
- 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. No, onda mora promijeniti i SHA1 njegovog sljedbenika, i sljedbenika njegovog sljedbenika, i...

Grane

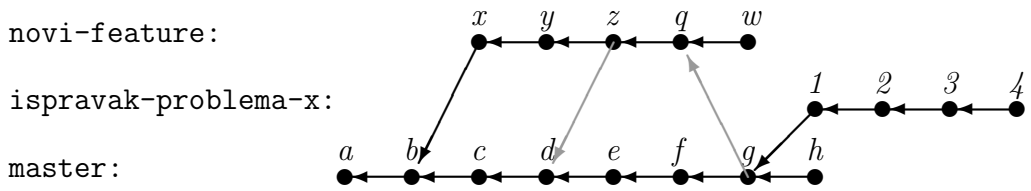
Razmislimo o još jednom detalju, uz poznati graf:



[17] Ovdje treba napomenuti kako SHA1 nije *potpuno* siguran. Ukoliko se nađe algoritam s kojime je moguće naći različite stringove za prizvoljne SHA1 stringove, onda on prestaje biti jednosmjerna funkcija. I tada cijela sigurnost potencijalno pada u vodu, jer netko može podvaliti drukčiji string u povijest za isti SHA1 string. Postoje istraživanja koja naznačuju da se to može. Moguće je da će git u budućnosti preći na SHA-256.

Graf kao ovaj gore matematičari zovu usmjereni graf jer su veze između čvorova usmjerene: $\vec{ab}$, $\vec{bc}$, itd. Znamo već da svaki čvor tov 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, gornji graf bi mogli prikazati i ovako:



Sve strelice su ovdje usmjerene suprotno negoli u grafovima kakve smo do sada imali. Naime, čvor g ima referencu na 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 kronološkog reda nastajanja) doći sve do mjesta gdje je grana nastala.

Zato gitu interno grane i nisu ništa drugo neki reference na njihove zadnje *commit*ove.

Reference

SHA1 stringovi su računalu praktični, no ljudima su mrvicu nezgodni za pamćenje :) Zbog toga git ima par zgodnih 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.

Referenca `HEAD` je uvijek referenca na zadnji *commit* u grani u kojoj se nalazimo. Ukoliko nam treba referenca na predzadnji *commit*, mogli bi pogledati `git log` i tamo naći njegov SHA1. Postoji i lakši način: `HEAD~1`. Analogno, pred-predzadnji commit je `HEAD~2`, i tako dalje.

Na primjer, 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 HEAD~10
```

Notacija kojom dodajemo `~1`, `~2`, ... vrijedi i za reference na grane i na pojedine SHA1. 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`.

Zapamtite ove trikove s referencama, jer to ćete često koristiti!

.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/
```

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

.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
```

I to je onaj 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 objekta 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 metapodatke vezane uz ono što im sam naziv kaže. *Blob* sadrži binarni sadržaj neke datoteke, dok je *tree* spisak 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 smo to učinili u tom *commit*u,
- Datum i vrijeme kad je nastao,
- Autora,

- Referencu na jedan objekt tipa *tree* koji sadrži spisak svih datoteka koje su sudjelovale u tom *commit*u.

Drugim riječima, 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 *blob* objekte može zapakirati više datoteka ili samo dijelova datoteka posebnim algoritmom za pakiranje (*pack*).

.git/refs

Bacimo pogleda 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
```

Svaki 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.

Zaključatk se sam nameće – u `.git/refs` se nalaze reference na sve grane, tagove i grane udaljenih repozitorija, a koji se nalaze u `.git/objects`.

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 nalazim 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 ekspenzijom `.sample`. Ukoliko taj sample maknete, one će se početi izvršavati na tim događajima.

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

Sigurno će vam se ponekad desiti da želite vidjeti koje se se izmjene desile između predzadnjeg i pred-predzanjeg *commit*a ili da vratite neku 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_xizse` u projektu... Čak i ako vam se neki od navedenih scenarija, vjerujte mi – trebati će vam.

U ovom poglavlju ćemo proći samo neke od često korištenih varijanti naredbi za proučavanje povijesti projekta.

Diff

Važna naredba je i `git diff`. S njome provjeravate razlike između dva *commit*a. Na primjer:

```
git diff master tesna-grana
```

...će nam ispisati razliku između dvije grane. Pripazite, jer redosljed je ovdje bitan. Ukoliko isprobate s:

```
git diff tesna-grana master
```

...dobiti ćete suprotan ispis. Ako ste u `testna-grana` jedan redak dodali – u jednom slučaju će `diff` ispisati da ste ga dodali, a u drugom oduzeli.

Želite li provjeriti koje su izmjene dogodile između predzadnjeg i pred-predzadnjeg *com-*

*mit*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 `git log <naziv_grane>` će vam ispisati povijest 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*, on će jednostavno krenuti "unazad" po grafu i dati vam povijest koju na taj način nađe. Pa tako, ako želimo povijest trenutne grane, ali bez zadnjih pet unosa, pitati ćemo jednostavno:

```
git log HEAD~5
```

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

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

Želimo li povijest sa *samo* nekoliko zadjnih unosa, koristimo `git log -n` sintaksu:

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

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

```
git log -10
```

Pretraživanje povijesti

Vrlo često će vam se dogoditi da tražite neki *commit* iz povijesti. Ovdje ćemo proći samo dva najčešća slučaja. Prvi je kad pretražujete prema tekstu *commit*a, tada se koristi `git log --grep=<regularni_izraz>`. Na primjer, tražim 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 string 'x'"? Tada se koristi `git log -S<string>`. Dakle, ne u komentaru *commit*a nego **u sadržaju datoteka**. Recimo da tražite tko je prvi napisao funkciju `get_image_width`:

```
git log -Sget_image_width
```

Treba li pretraživati za string s razmacima:

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

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

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

Blame

S `git blame <datoteka>` ćete dobiti ispis datoteke s detaljima o tome **tko**, **kad** i u **kojem *commit*u** je napravio svaku liniju u toj datoteci i **iz koje datoteke** je ta izmjena došla ovdje:

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

Whatchanged

Naredba `git whatchanged` je vrlo slična `git log`, jedino što uz svaki *commit* ispisuje i spisak 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
```

Preuzimanje datoteke iz povijesti

Ponekad nam se svima dešava da smo izmijenili datoteku i kasnije se sjetili da ta izmjena ne valja. Verzija od prije 5 *commit*ova je 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 datoteke iz te grane. Odnedavno znamo i da naziv grane nije ništa drugo nego referenca na njen zadnji *commit*. Analogno, ako umjesto grane, tamo stavimo referencu na neki drugi *commit* dobiti ćemo stanje datoteke iz tog trenutka u povijesti.

Dakle, `git checkout` se, osim za prebacivanje s grane na granu, može koristiti i za pruzi-

manje neke datoteke iz prošlosti:

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

. . . nam u trenutni direktorij vraća 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 da datoteka kakva je bila u predzadnjem *commit*u grane `test`?

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

Isto 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.

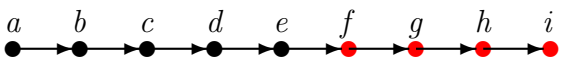
Na primjer, otkrili ste bug, ne znate gdje točno u kodu, ali znate da se taj bug nije prije manifestirao. Bilo bi zgodno kad bismo cijeli projekt mogli teleportirati na neko stanje kakvo je bilo prije n *commit*ova. Ništa lakše:

```
git checkout HEAD~10
```

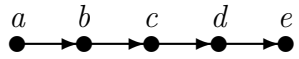
. . . i za trenutak imate stanje kakvo je bilo prije 10 *commit*ova. Sad tu možete s `git branch` kreirati novu granu ili vratiti se na najnovije stanje s `git checkout HEAD`.

Reset

Uzmimo ovakav scenarij; radite na nekoj grani. U jednom trenutku stanje je ovakvo:



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



...i od tamo nastaviti cijeli posao, ali nekako drukčije. E sad, lako je "teleportirati se" s `git checkout ...`, to 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
```

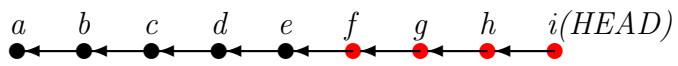
Želimo li se vratiti na `974ef0ad8351ba7b4d402b8ae3942c96d667e199` i maknuti sve *commit*ove koji su se desili nakon njega. Isto:

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

Postoji i `git reset --soft <referenca>`. S tom varijantom se isto gube *commit*ovi nakon onog kojeg ste specificirali, ali stanje datoteka u vašem 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:



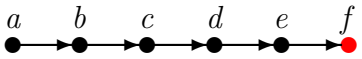
Uopće se ne trebamo truditi **brisati** čvorove/*commit*ove f, g, h i i. Dovoljno je reći "Pomakni HEAD tako da pokazuje na e. I to je upravo ono što git čini s `git reset`:

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

Revert

Treba imati na umu jednu stvari – `git reset` **mijenja povijest projekta retrogradno**. Svaki *commit* mijenja povijest projekta, no on to čini tako da **dodaje na kraju grane**. To ponekad može biti problem (posebno u situacijama kad radite s drugim ljudima, o tome više u sljedećem poglavlju).

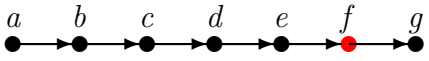
To se može riješiti s `git revert`. Uzmimo, na primjer, ovakvu situaciju:



Došli ste do zaključka da je *commit* f neispravan i treba vratiti na stanje kakvo je bilo u e. `git reset` bi f jednostavno maknuo 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*:

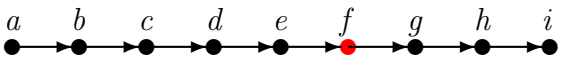
```
git revert HEAD
```

Rezultat će biti da je dosadašnji graf ostao potpuno isti, no **dodan je novi commit** koji miče sve izmjene iz 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:

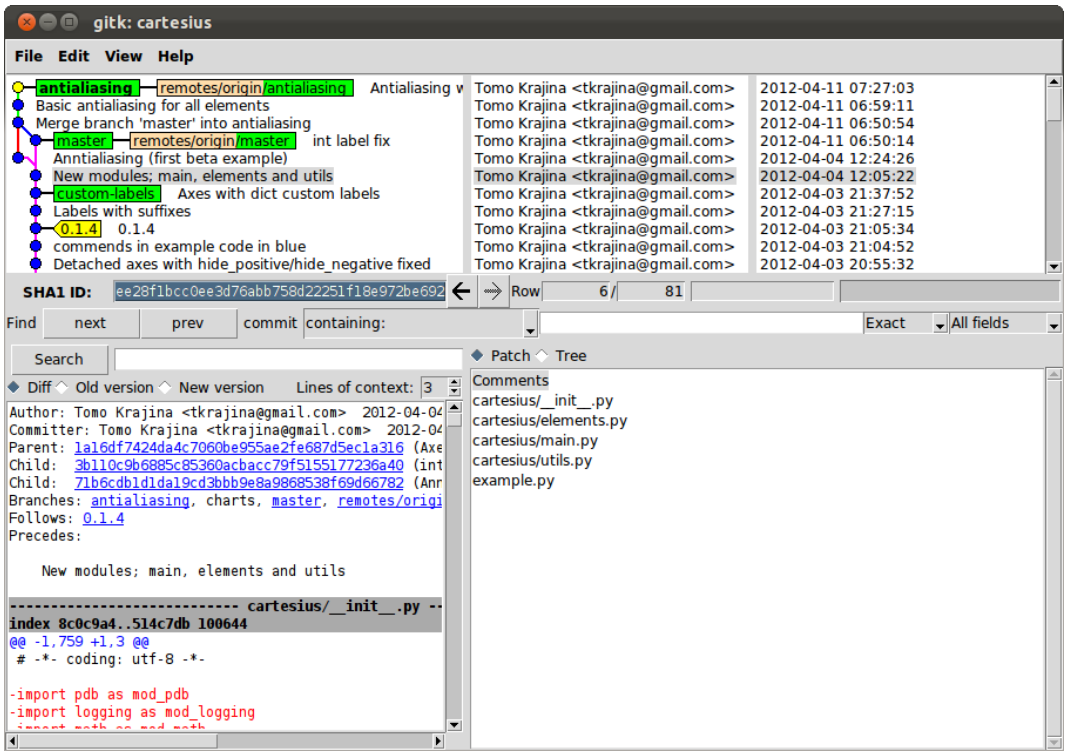
. . . gdje *commit* j ima stanje kao u i, ali bez izmjena koje su uvedene u f.

Naravno, ako malo o tome razmislite, doći ćete do zaključka 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 samo damo do znanja kako bi *revert* trebao izgledati.

Gitk

Git, standardno, sadrži i jedan pomoćni programčić koji vam grafički prikazuje povijest trenutne grane:



Prikazani dijelovi su, redom s lijeva na desno od gore prema dolje:

- grafički prikaz povijesti grane i *commit*ova iz drugih grana koji su imali veze s tom granom (bilo zbog grananja, *merge*anja ili *cherry-pick merge*),
- spisak ljudi koji su *commit*ali,
- datum i vrijeme,
- pregled svih izmjena,
- pregled datoteka koje su izmijenjene.

Kad odaberete *commit* dobiti ćete sve izmjene i datoteke koje su sudjelovale. Kad odaberete na 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
```

Kao i `git gui`, tako i `gitk` na prvi pogled možda baš i nije jako intuitivan. No, brz je i praktičan za rad kad se naučite na njegovo grafičko sučelje.

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 scenarija kako može funkcionirati ta interakcija. Koncentrirajmo se sada na sam trenutak kad repozitorij "dođe" ili "nastane" na našem lokalnom računalu. 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).

No, 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. Krenimo s prvim tipičnim scenarijem...

Naziv i adresa repozitorija

Prvu stvar koju ćemo obraditi je kloniranje udaljenog repozitorija. No, ima prije toga jedna sitnica koju 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 njega andresa. Github[18] omogućava 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).

Nadalje, svaki udaljeni repozitorij ima i svoje kratko ime. Nešto kao: `origin` ili `vanjin-repozitorij` ili `slobodan` ili `dalenov-repo`. Nazivi su vaš slobodan izbor. Tako, ako vas četvero radi na istom projektu, njihove udaljene repozitorije možete nazvati `marina`, `ivan`, `karla`. I sa svakim od njih možete imati nekakvu vrstu interakcije. Na neke ćete slati svoje izmjene (ako imate ovlasti), a s nekih ćete izmjene preuzimati u svoj repozitorij.

[18] Web servis koji omogućava da držite svoje git repozitorije

Kloniranje repozitorija

Kloniranje je postupak kojim kopiramo cijeli repozitorij na nekoj udaljenoj lokaciji na naše lokalno računalo, a da bi onda s njime dalje nastavili raditi. U stvari, kopirati repozitorij je jednostavno, dovoljno je u neki direktorij kopirati `.git` direktorij drugog repzitorija i onda u njemu napraviti `git checkout HEAD`.

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**.

Postupak je jednostavan, ako znamo adresu (a moramo znati), onda...

```
$ 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.
```

...**kopira cijeli projekt, zajedno sa cijelom poviješću** na naše računalo. I to u direktorij `uvod-u-git`. Sad tu možemo gledati povijest, granati, *commit*ati, ... 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 malo kasnije.

E, i još nešto. Sjećate se kad sam napisao da su nazivi udaljenih repozitorija vaš slobodan izbor. Nisam baš bio 100% iskren. Kloniranje je 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 na 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). No, naš dio svijeta su samo ova dva s kojima direktno imamo posla. Jedan je udaljeno kojeg smo klonirali, a drugi je lokalni koji se nalazi pred nama.

Prije negoli 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

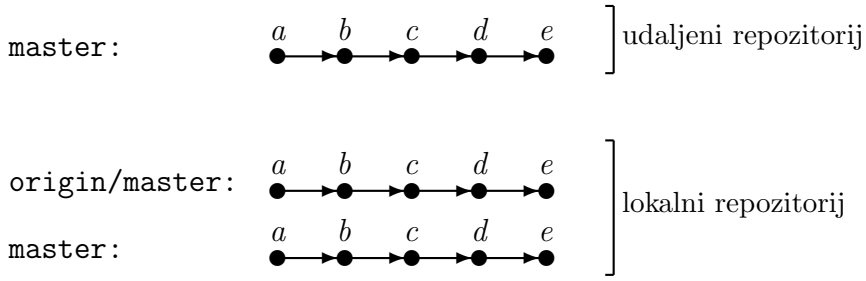
za naredbu `git branch`, koja nam ispisuje spisak svih grana na našem repozitoriju. No, sad imamo posla i sa udaljenim repozitorijem – njega smo klonirali.

S `git branch -a` ispisujemo **sve grane koje su nam trenutno dostupne u lokalnom rpeozitoriju**. 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`. Prvo, ovo `remotes/` znači da, iako toj grani imamo pristup 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 udaljenog repozitorija.

Ako vam ovo zvuči zbunjujuće, ništa čudno. No, sve će sjesti na svoje mjesto kad to počnete koristiti.

Djelomično kloniranje povijesti repozitorija

Našli ste na internetu neki zanimljiv projekt i njegov git repozitorij i htjeli bi ga skinuti i proučiti njegov 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. 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 njegovih zadnjih *commit*ova s:

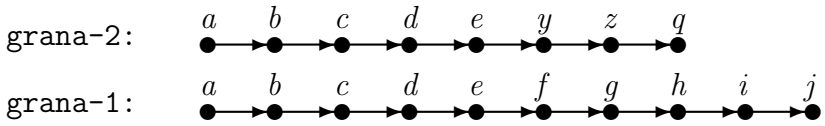
```
git clone --depth 5 --no-hardlinks git://github.com/tkrajina/uvod-u-git.git
```

To će biti puno brže, no s takvim klonom nemamo pristup cijeloj povijesti i ne bi mogli raditi sve ono što teorijski možeomo s pravim klonom. Djelomično kloniranje je više zamišljeno da bismo skinuli kod nekog projekta samo da proučimo kod, a ne da bi se na njemu nešto ozbiljno radilo.

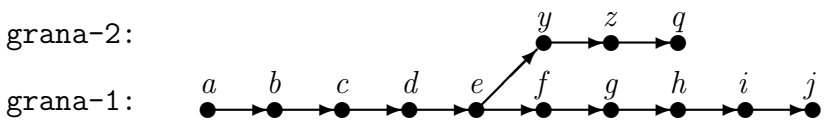
Digresija o grafovima, repozitorijima i granama

Prije nego što nastavimo, napravio bih kratku digresiju koja je važna za razumijevanje grafova projekata i udaljenih projekata koji će slijediti.

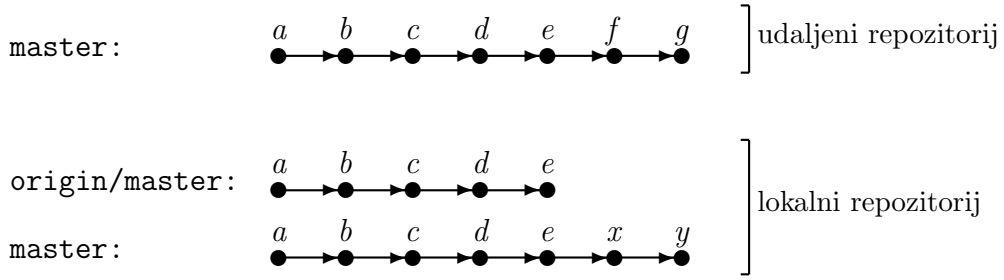
Za razumijevanje nastavka ovog poglavlja važno je shvatiti da je graf...



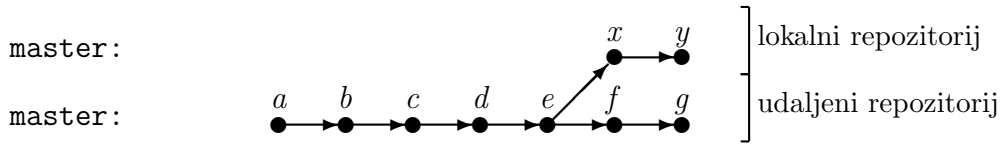
...ekvivalentan grafu...



...zato što ima zajednički početak povijesti (a, b, c, d i e). Zbog toga, na primjer u sljedećoj situaciji:



Trebamo si zamisliti da je odnos između našeg lokalnog `master` i udaljenog `master` – kao da imamo jedan graf koji se granao u e. Jedino što se svaka grana nalazi u svom repozitoriju, a ne više u istom. Zanemarimo li na trenutak `origin/master`, odnos između naša dva `master`a je:

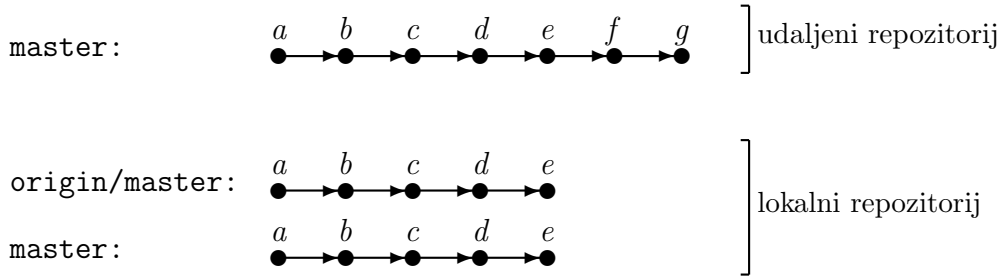


Opisani odnos vrijedi i za `master` i `origin/master`.

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

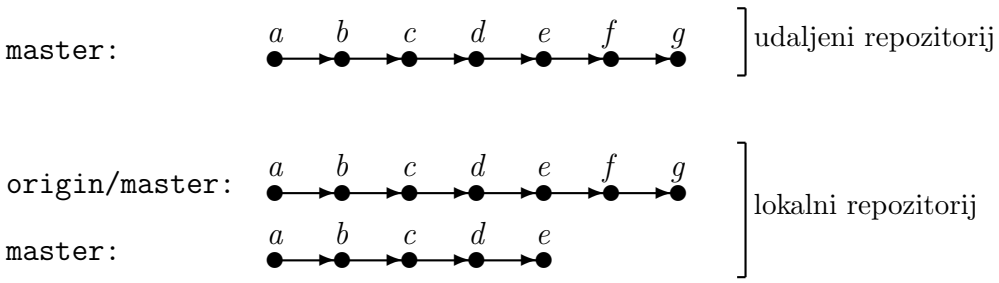
automatski. To što je vlasnik udaljenog repozitorija dodao dva *commit*a (e i f) ne znači da će naš repozitorij nekom čarolijom to odmah saznati.

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 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 da `origin/master`, ali...

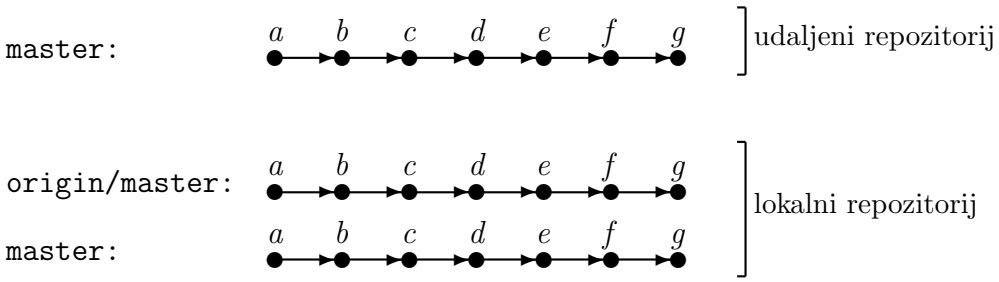
```
$ git checkout origin/master
Note: checking out 'origin/master'.

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 smijemo *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:

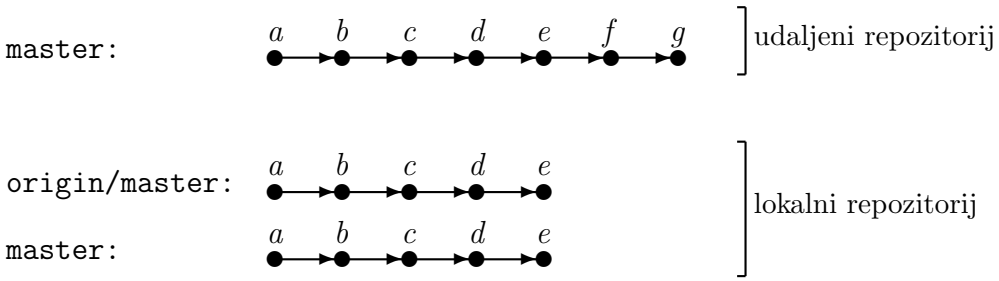
```
git fetch
```

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

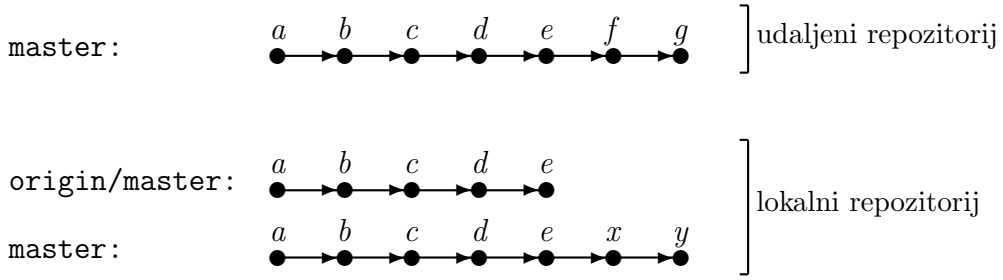
```
git merge origin/master
```

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

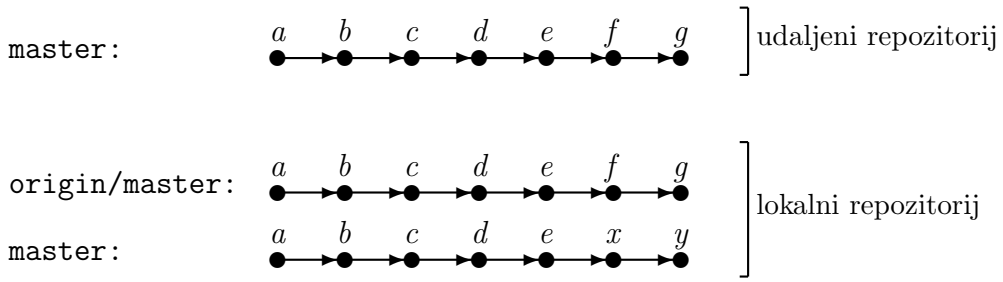
Malo složeniji scenarij je sljedeći – recimo da smo nakon...



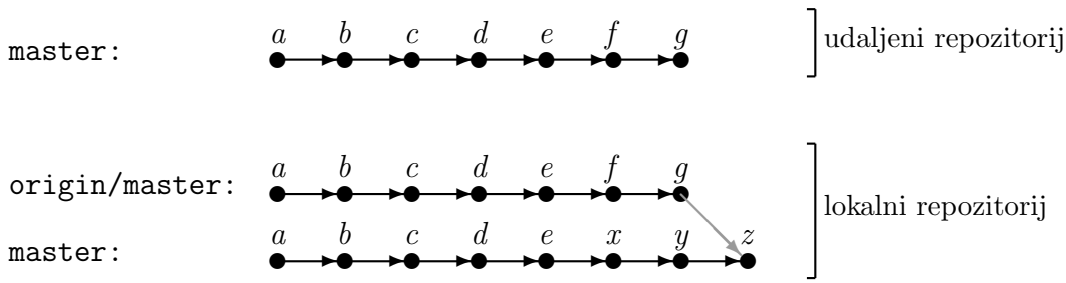
... mi *commit*ali u naš lokalni repozitorij svoje izmjene. Recimo da su to x i y:



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 imamom 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. No, to već znamo riješiti. Rezultat *merge*a je novi čvor z:



Pull

U prethodnom poglavlju smo opisali da je tipični redosljed 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 redosljedu.

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

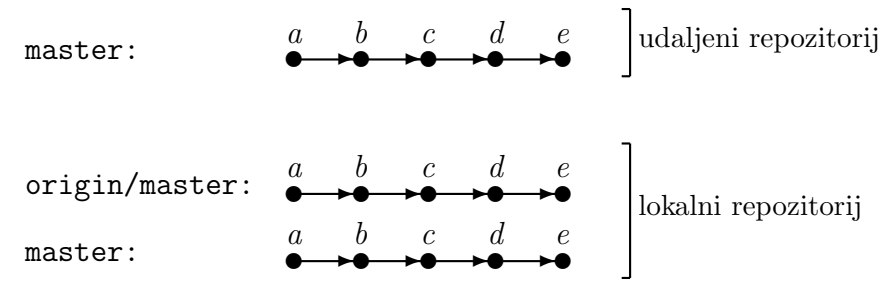
```
git pull
```

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

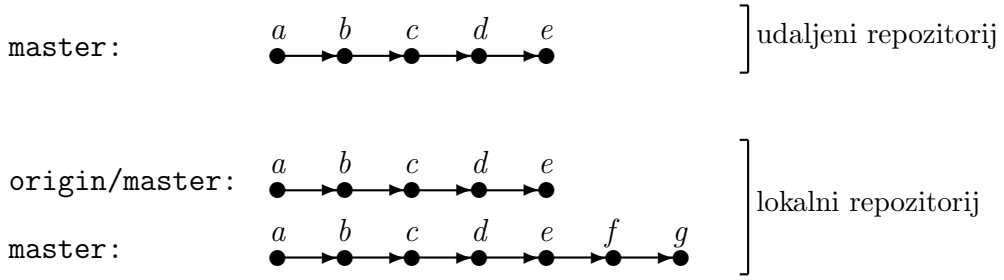
Push

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. Krećemo sad na prvu (i ovdje jedinu) radnju 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...



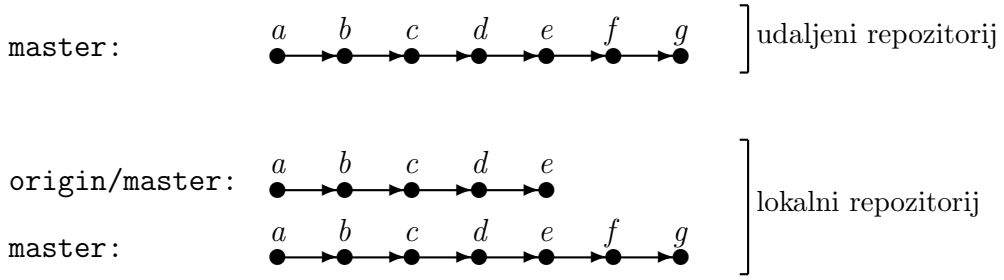
...i sad bi htjeli te izmjene "prebaciti" na udaljeni repozitorij. Prvo i osnovno što nam treba svima biti jasno – "prebacivanje" naših lokalnih izmjena na udaljeni repozitorij ovisi o tome imamo li ovlasti za to ili ne. **Udaljeni repozitorij mora biti tako konfiguriran da bismo mogli raditi** `git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

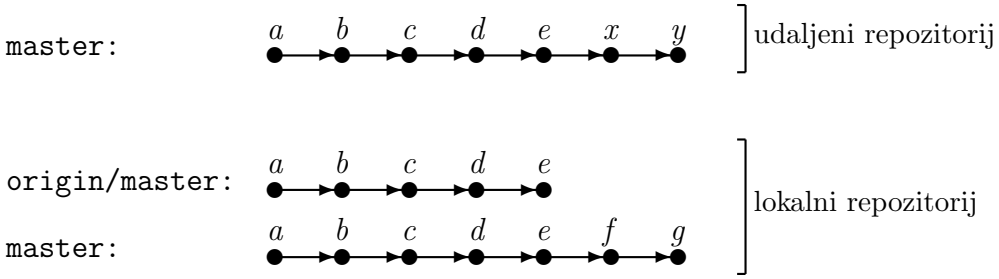
```
$ git push origin master
Counting objects: 11, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (9/9), done.
Writing objects: 100% (9/9), 1.45 KiB, done.
Total 9 (delta 6), reused 0 (delta 0)
To git@github.com:tkrajina/uvod-u-git.git
   0335d78..63ced90  master -> master
```

Stanje će sad biti:



Ovo je bio jednostavni scenarij u kojem smo u našem `master` *commit*ali, ali u udaljenom

`master` se nije ništa događalo. 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:

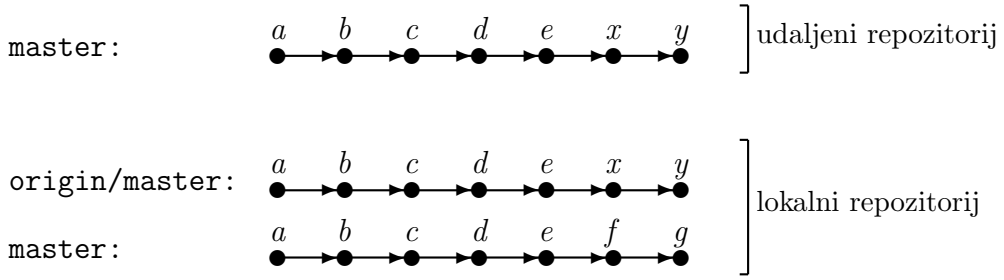
```
$ git push origin master
To git@domena.com:repozitorij
 ! [rejected]        master -> master (non-fast-forward)
error: failed to push some refs to 'git@domena.com:repozitorij'
To prevent you from losing history, non-fast-forward updates were rejected
Merge the remote changes (e.g. 'git pull') before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

Riješenje je standardno:

```
git fetch
```

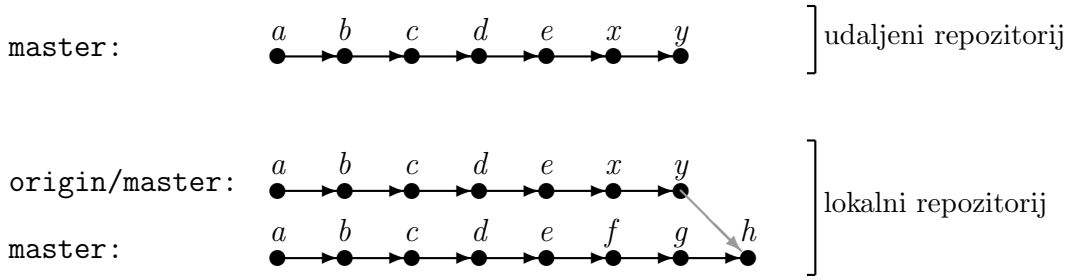
...i stanje je sad:



Sad ćemo, naravno:

```
git merge origin/master
```

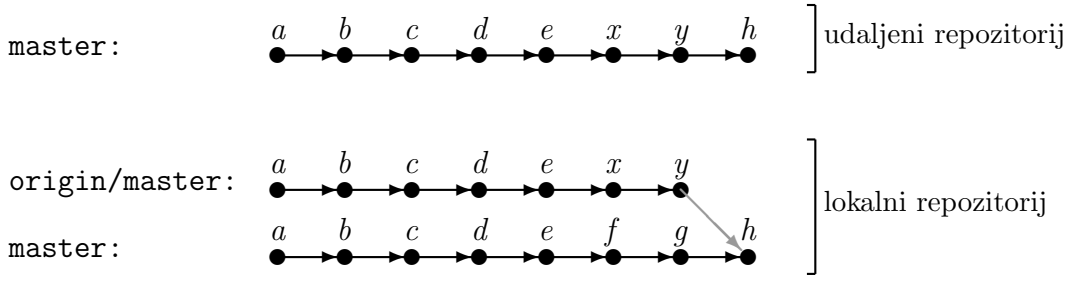
Na ovom koraku se može desiti da imate konflikata koje eventualno treba ispraviti s `git mergetool`. Nakon toga, stanje je:



Sad možemo uredno napraviti:

```
git push origin master
```

. . . a stanje naših repozitorija će biti:



Iako nam se može učiniti da udaljeni projekt nema izmjene koje smo mi uveli u f i g – to nije točno. Naime, h je rezultat preuzimanja izmjena (*merge*) iz `origin/master` i on u sebi sadrži kombinaciju i x i y i (naših) f i g.

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 lokalnom repozitorija poslati na udaljeni. To su tagovi.

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

```
git push origin --tag
```

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 komplikacija.

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

Rebase origin/master

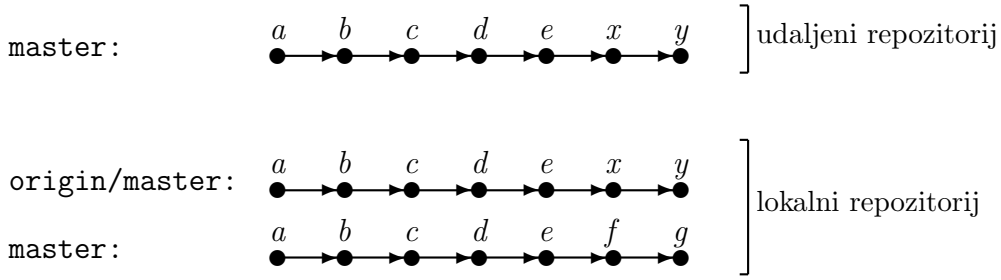
Želimo li da se naši f i g (iz prethodnog primjera) vide u povijesti udaljenog projekta – i to se može s:

```
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:



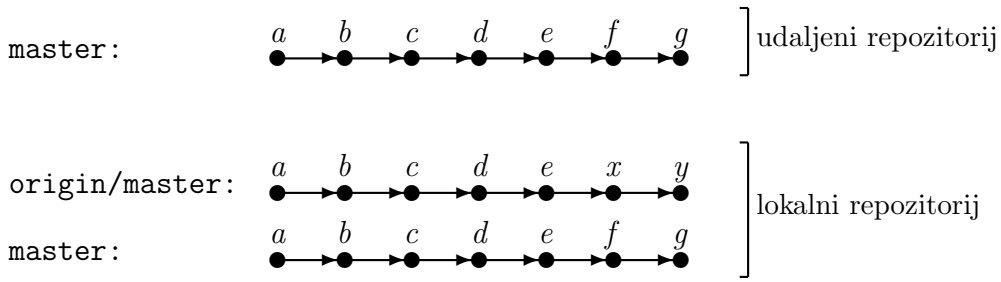
Da ponovimo još jednom. Standardni postupak je `git fetch` (što je u gornjem primjeru već učinjeno) i `git merge origin/master`.

Međutim, ovdje ima 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 jednostavno "pregazili" njegove *commit*ove u udaljenom `master`.

To se može, komanda je:

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

. . . a rezultat će biti:



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

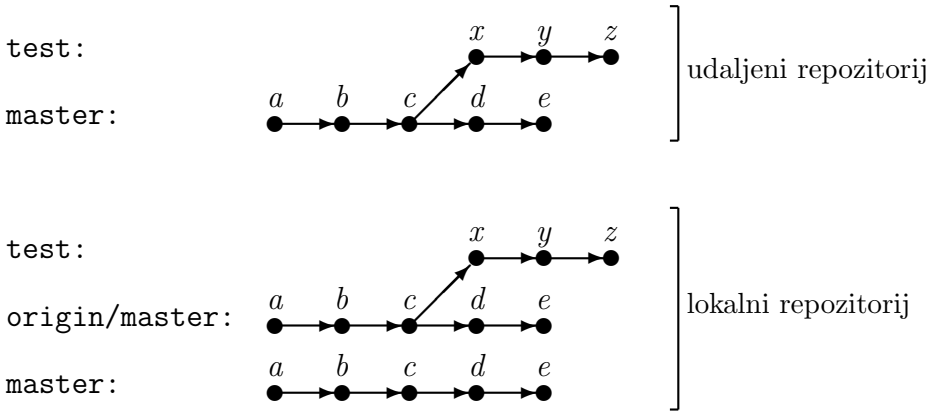
Treba, međutim, imati na umu da ovakvo ponašanje nije baš uvijek poželjno. Zbog dva razloga:

- **Mi** smo zaključili da *commit*ovi x i y ne valjaju. Možda smo jednostavno 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 riješenje 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/test`. Da je bilo više grana u repozitoriju kojeg kloniramo, imali bi više ovih `origin/` grana. To lokalno možemo vidjeti s:

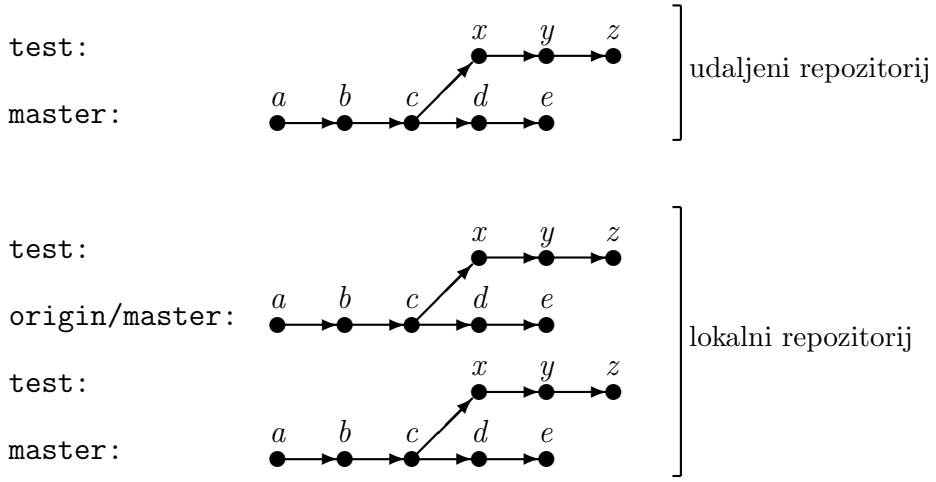
```
$ git branch -a
* master
  remotes/origin/master
  remotes/origin/test
```

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/test
git branch test
git checkout test
```

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

Sad je stanje:



... odnosno:

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

Sad u taj lokalni `master` možemo *commit*ati koliko nas je volja. Kad ćemo htjeti poslati svoje izmjene na udaljenu granu, postupak je isti kao i do sada, jedino što radim s novom granom umjesto master. Dakle,

```
git fetch
```

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

```
git merge origin/test
```

I, na kraju...

```
git push origin test
```

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

TODO: Zamijeniti naziv grane `test` u nešto smislenije

Brisanje udaljene grane

Ukoliko želimo obrisati granu na udaljenom repozitoriju – to radimo s posebnom sintaksom naredbe `git push`:

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

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

Git push i *tracking* grana

TODO:

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 dobijamo one `origin/` brancheve, koji su kopija udaljenih grana i mogućnost osvježavanja njihovo stanje s `git fetch`.

Imamo i nekoliko ograničenja, a najvažnija je to što možemo imati samo jedan `origin`. Što ako želimo imati posla s više udaljenih repozitorija. Dakle, 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[19]. Bilo kloniranjem bilo kopiranjem nečijeg projekta (tj. kopiranjem `.git` direktorijia i *checkout*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, uzmomo da radimo s dva programera od kojih je jednome repozitorij našeg zajedničkog projekta `https://github.com/korisnik/projekt.git`, a drugome `git@domena.com:projekt`.

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

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 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 njegovo vlasnika, možda ćemo htjeti i `origin` nazvati tako. Recimo da je to Karlin:

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

Spisak 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
```

I sad, kad želimo isprobati neke izmjene koje je Karla napravila (a nismo ih još preuzeli u

naš repozitorij), jednostavno:

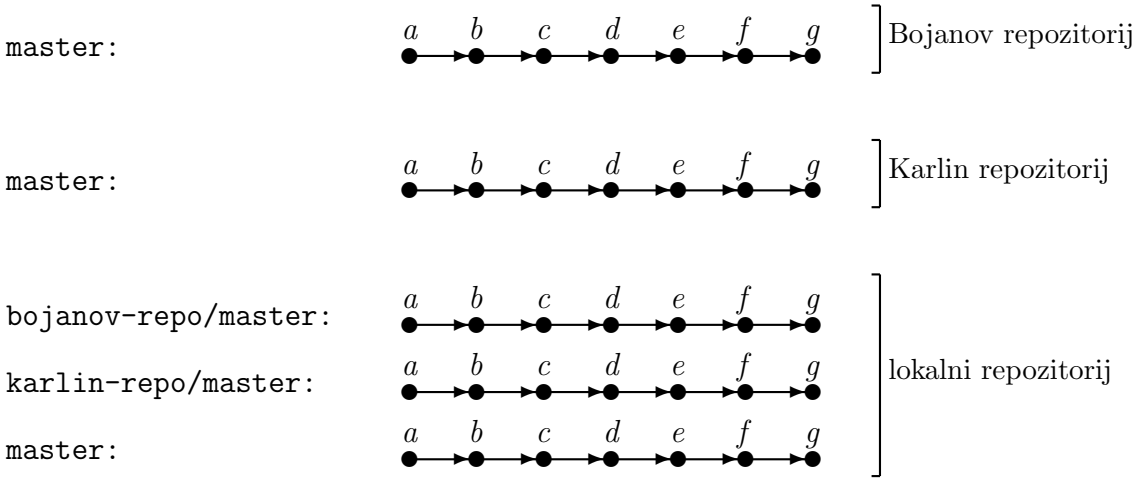
```
git checkout karlin-repo/master
```

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

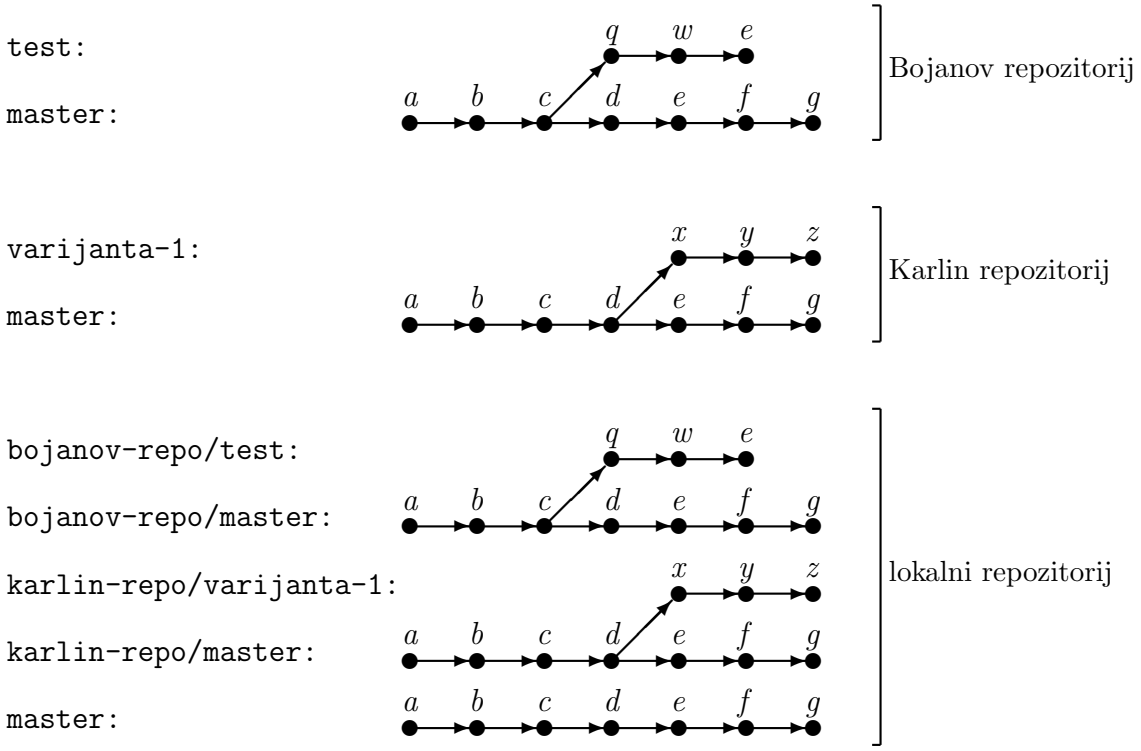
```
git merge karlin-repo/master
```

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

TODO:"Bare" repozitorij

TODO: Commit na udaljenu granu

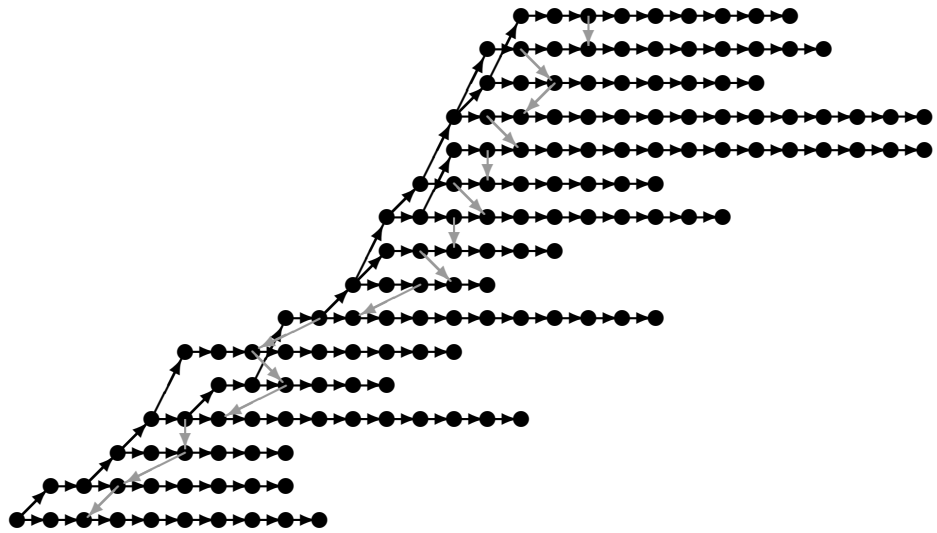
”Higijena” repozitorija

Naš repozitorij je naš životni prostor i s njime živimo dio dana. Kao što se trudimo držati stan urednim, tako bi trebalo i s našim virtualnim prostorima. Preko tjedna, kad rano ujutro odlazimo na posao i vraćamo se kasnije popodne, se ponekad desi da nam se u stanu nagomila robe za pranje. No, nekoliko puta tjedno treba uzeti pola sata vremena i počistiti nered koji je zaostao.

Tako i s repozitorijem. Ima nekoliko stvari na koje bi trebali pripaziti.

Grane

Nemojte si dopustiti da grane u vašem repozitoriju izgledaju ovako ”zbrčkano”:



Iako nam git omogućuje da imamo neograničen broj grana, ljudski um nije sposoban vizualizirati si više od[20] 5-10. Kako stvaramo nove grane – dešava se da imamo one u kojima smo počeli neki posao i kasnije odlučili da nam ne treba. 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

[20]Barem moj nije, ako je vaš izuzetak, preskočite ovo.

udaljenim granama) i provjerite one koje više ne koristite.

Ukoliko nismo sigurni je li nam u nekoj grani ostala možda još kakva izmjena koju treba vratiti u `master`, postupak je sljedeći:

```
git checkout neka-grana
git merge master
```

... da bismo preuzeli sve izmjene iz master, tako da stanje bude ažurno. I sad...

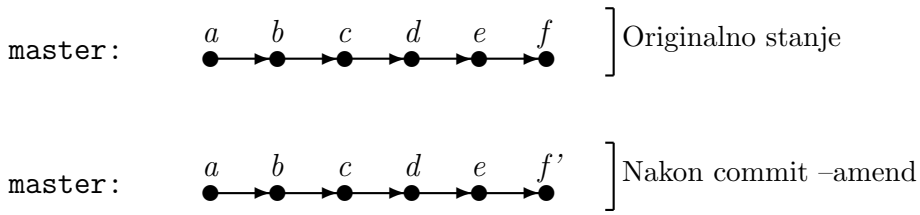
```
git diff master
```

... da bismo provjerili koje su točno izmjene ostale. Ako je prazno – nema razlika i možemo brisati. Ako nije – **ima** izmjena i sad se sami odlučite jesu li te izmjene nešto važno što ćete nastaviti razvijati ili nešto što možemo zanemariti i obrisati tu granu.

Git gc

Druga stvar koju 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 u tom direktoriju. Što se dešava sa zadnjim objektom kad napravimo `git commit --amend`.

Podsjetimo se, s tom komandom mijenjamo naš zadnji *commit*. Grafički:



Tako barem izgleda, ali nije baš doslovno tako. Ono što git interno napravi je da doda **novi** objekt (*f'*) i na njega pomiče referencu `HEAD` (koja je do tada gledala na *f*).

Međutim, objekt koji je prije bio zadnji *commit* **ostaje u** `.git/object` **direktoriju**. I one se više nikad neće koristiti. Puno tih `git commit --amend` posljedično ostavlja puno takvog "smeća" u repozitoriju.

To vrijedi i za neke druge operacije kao brisanje grana ili rebase. Git to čini da bi tekuće operacije bilo što je brže moguće. Čišćenje takvog smeća (*garbage collection* iliti *gc*) ostavlja za kasnije.

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)
```

Tu naredbu treba izvršiti 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 zanima – `git help` je uvijek tu da pomogne.

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 verzioniranje koda 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 tvoreni 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, umjesto hostinga je koristiti vlastiti server. Najjednostavniji scenarij je da jednostavno korisimo ssh protokol. Git uredno tako funkcionira. Dakle, sve što trebamo učiniti je inicirati neki udaljeni git repozitorij i koristiti ga kao remote.

Ako je naziv servera `server.com`, korisničko ime s kojim se prijavljujemo `git`, a direktorij s repozitorijem `projekt/`, onda ga možemo početi koristiti s:

```
git remote add moj-repozitorij git@server.com:projekt
```

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 listati datoteke na serveru, gledati koji servisi su startani, isl. 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 operativnom sustavu korisnici imaju definiran *shell*, odnosno nekakav program u kojem on može 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 komande 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 pluginovi

Ukoliko vam se učini da je skup naredbi koje možemo dobiti s `git <naredba>` limitiran – lako je dodati nove. Sve što trebate jer snimiti **TODO:**

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. Postoji i *plugin* koji omogućuje da naredbe iz jednog koristite u radu s drugim[21].

Mercurial ima malo konzistentnije imenovane naredbe, ali isto tako i manji broj koris-

[21] http://hg-git.github.com

nika. Međutim, 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:

```
hg init
```

Dodavanje datoteke `README.txt` u prostor predviđen za sljedeći *commit* (ono što je u gitu

index):

```
hg add README.txt
```

Micanje datoteke iz istog:

```
hg forget README.txt
```

Commit:

```
hg commit
```

Trenutni status repozitorija:

```
hg status
```

Izmjene u odnosu na repozitorij:

```
hg diff
```

Kopiranje 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:

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 strane riječi tipa *mrđanje*, *brenčanje*[22], *ekspajranje*, *eksekjutanje*. Slažem se da bi bilo poželjno bi bilo koristiti alternative koje su više u duhu jezika[23]. Međutim, besmisleno mi je bilo izmišljati nove riječi za potrebe priručnika koji bi bio uvod u git. Koristiti termine koje sam izmislio u letu i 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.

Činjenica je da većina pojmova jednostavno nemaju ustaljen hrvatski prijevod[24]. I zato sam ih koristio na točno onakav način kako se one upotrebljavaju u (domaćem) programerskom svijetu.

Problem je i u tome što prijevod često uopće 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. Jedino što se izmjene iz jedne preuzimaju i u drugu. Ispravno bi bilo "preuzimanje izmjena iz jedne grane u drugu", ali to zvuči prilično nespretno da bi se koristilo u standardnom govoru.

Dakle, svi pojmovi sam koristio u izvornom obliku, ali *ukošenim* fontom. Na primjer *fast-forward merge*, *merge*anje ili *commit*anje.

[22] Jedan kolega s posla, koji često *merge*a ima nadimak "mrđer"

[23] Naravno, ne treba ni pretjerati s raznim *vrtoletima*, *čegrtasim velepamtilima*, *nadstolnim klizalima*, isl.

[24] Jedan od glavnih krivaca za to su predavači na fakultetima koji ne misle da je verzioniranje koda tema za fakultetske kolegije. Oni su ti, a ne ljudi "na terenu" kao ja, koji su najviše zaduženi za to da budući stručnjaci koriste izraze u duhu jezika.