

# **Pro Git**

**Scott Chacon\***

2011-08-31

\*This is the PDF file for the Pro Git book contents. It is licensed under the Creative Commons Attribution-Non Commercial-Share Alike 3.0 license. I hope you enjoy it, I hope it helps you learn Git, and I hope you'll support Apress and me by purchasing a print copy of the book at Amazon: <http://tinyurl.com/amazonprogit>



# Contents

<b>1</b>	<b>Alkusanat</b>	<b>1</b>
1.1	Versionhallinnasta . . . . .	1
1.1.1	Paikalliset versionhallinta järjestelmät . . . . .	1
1.1.2	Keskittetyt versionhallinta järjestelmät . . . . .	2
1.1.3	Hajautetut versionhallinta järjestelmät . . . . .	3
1.2	Gitin lyhyt historia . . . . .	3
1.3	Git perusteet . . . . .	4
1.3.1	Tilannekuvia, ei eroavaisuuksia . . . . .	5
1.3.2	Lähes jokainen operaatio on paikallinen . . . . .	6
1.3.3	Git on eheä . . . . .	6
1.3.4	Git yleensä vain lisää dataa . . . . .	7
1.3.5	Kolme tilaa . . . . .	7
1.4	Gitin asennus . . . . .	8
1.4.1	Asennus suoraan lähdekoodista . . . . .	8
1.4.2	Asennus Linuxissa . . . . .	9
1.4.3	Asennus Macilla . . . . .	9
1.4.4	Asennus Windowsilla . . . . .	10
1.5	Ensikerran Git asetukset . . . . .	10
1.5.1	Identiteettisi . . . . .	11
1.5.2	Editorisi . . . . .	11
1.5.3	Diff työkalusi . . . . .	12
1.5.4	Tarkista asetukseksi . . . . .	12
1.6	Avun saanti . . . . .	12
1.7	Yhteenveto . . . . .	13
<b>2</b>	<b>Gitin perusteet</b>	<b>15</b>
2.1	Git tietolähteen hankinta . . . . .	15
2.1.1	Tietolähteen alustaminen jo olemassa olevalle hakemistolle . . . . .	15
2.1.2	Olemassa olevan tietolähteen kloonauk . . . . .	16
2.2	Muutosten tallennus tietolähteeseen . . . . .	17
2.2.1	Tiedostojesi tilan tarkistaminen . . . . .	17
2.2.2	Uusien tiedostojen jäljitys . . . . .	18
2.2.3	Muutettujen tiedostojen lavastus . . . . .	19
2.2.4	Tiedostojen sivuuttaminen . . . . .	21
2.2.5	Lavastettujen ja lavastattomien muutosten tarkastelu . . . . .	22
2.2.6	Pysyvien muutoksien tekeminen . . . . .	24

2.2.7	Lavastusalueen ohittaminen . . . . .	26
2.2.8	Tiedostojen poistaminen . . . . .	26
2.2.9	Moving Files . . . . .	27
2.3	Viewing the Commit History . . . . .	28
2.3.1	Limiting Log Output . . . . .	32
2.3.2	Using a GUI to Visualize History . . . . .	34
2.4	Undoing Things . . . . .	34
2.4.1	Changing Your Last Commit . . . . .	34
2.4.2	Unstaging a Staged File . . . . .	35
2.4.3	Unmodifying a Modified File . . . . .	36
2.5	Working with Remotes . . . . .	37
2.5.1	Showing Your Remotes . . . . .	37
2.5.2	Adding Remote Repositories . . . . .	38
2.5.3	Fetching and Pulling from Your Remotes . . . . .	39
2.5.4	Pushing to Your Remotes . . . . .	39
2.5.5	Inspecting a Remote . . . . .	40
2.5.6	Removing and Renaming Remotes . . . . .	41
2.6	Tagging . . . . .	41
2.6.1	Listing Your Tags . . . . .	41
2.6.2	Creating Tags . . . . .	42
2.6.3	Annotated Tags . . . . .	42
2.6.4	Signed Tags . . . . .	43
2.6.5	Lightweight Tags . . . . .	43
2.6.6	Verifying Tags . . . . .	44
2.6.7	Tagging Later . . . . .	45
2.6.8	Sharing Tags . . . . .	46
2.7	Tips and Tricks . . . . .	46
2.7.1	Auto-Completion . . . . .	46
2.7.2	Git Aliases . . . . .	47
2.8	Summary . . . . .	48

# Chapter 1

## Alkusanat

Tämä luku auttaa sinut pääsemään alkuun Gitin kanssa. Me aloitamme aluksi selittämällä vähän versionhallinta työkalujen taustaa, jonka jälkeen siirrymme siihen, kuinka saat Gitin järjestelmäsi ja lopulta, kuinka se asennetaan työskentely valmiiksi. Tämän luvun lopussa sinun tulisi ymmärtää miksi Git on olemassa, miksi sinun tulisi sitä käyttää ja kaikki pitäisi olla valmista, jotta voisit sitä käyttää.

### 1.1 Versionhallinnasta

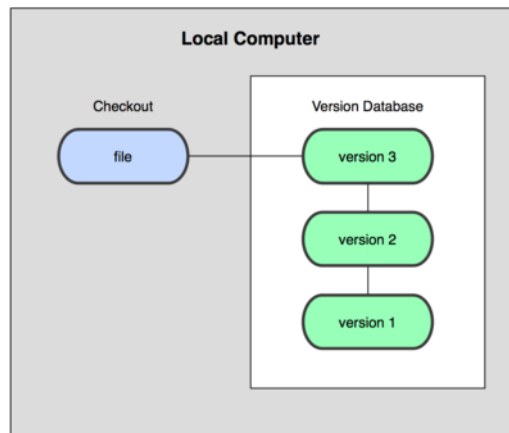
Mitä on versionhallinta ja miksi sinun pitäisi siitä välittää? Versionhallinta on menetelmä, joka ajan kuluessa tallentaa muutoksia tiedostoon tai joukkoon tiedostoja, jotta sinä voit palata tiettyihin versioihin myöhemmin. Esimerkkeinä tässä kirjassa, sinä käytät ohjelmiston lähdekoodia tiedostoina, jotka ovat versionhallittavana, vaikkakin todellisuudessa sinä voit tehdä tämän melkein millä tahansa tiedostolla tietokoneellasi.

Jos sinä olet graafinen- tai web suunnittelija ja haluat säilyttää jokaisen version kuvasta tai leiskasta (minkä sinä mitä varmimmin haluat), versionhallintamenetelmä (VCS) on erittäin viisas asia käytettäväksi. Se mahdollistaa, että voit palauttaa tiedoston takaisin edelliseen tilaan, palauttaa koko projektin takaisin edelliseen tilaan, vertailla muutoksia ajan kuluessa, nähdä kuka viimeksi muokkasi jotain mikä voi olla ongelman aiheuttaja, kuka esitteli ongelman ja milloin, ja muuta. VCS:n käyttö pääasiassa tarkoittaa sitä, että jos sinä sotket jotain tai menetät tiedostoja, voit helposti palautua edelliseen toimivaan tilaan. Lisäksi, saat kaiken tämän erittäin vähällä ylläpidolla.

#### 1.1.1 Paikalliset versionhallinta järjestelmät

Monen ihmisen versionhallintaratkaisu on kopioida tiedostoja toiseen kansioon (ehkäpä aikaleimattu kansio, jos he ovat fiksuja). Tämä lähestymistapa on erittäin yleinen, koska se on niin yksinkertainen, mutta se on myös erittäin virhealtis. On helppo unohtaa missä hakemistossa olet ja epähuomiossa kirjoittaa väärään tiedostoon tai kopioida sellaisten tiedostojen päälle, joihin et tarkoittanut koskea.

Tämän ongelman ratkaisemiksi, ohjelmoijat kauan sitten kehittivät paikallisen VCS:n, jolla oli yksinkertainen tietokanta, joka piti kaikki tiedostojen muutokset muutostenhallinnan alla (katso Kuva 1-1).



**Figure 1.1:** Paikallinen versionhallinta diagrammi.

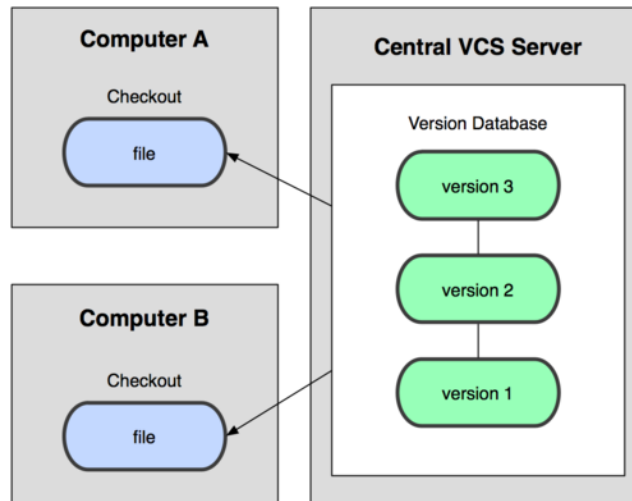
Yksi suosituimmista VCS työkaluista oli rcs:ksi kutsuttu järjestelmä, joka on yhä tänä päivänä toimitettu monen tietokoneen mukana. Jopa suosittu Mac OS X käyttöjärjestelmä sisältää rcs komennon, Developer Tools paketin asennuksen jälkeen. Tämä työkalu periaatteessa toimii pitämällä pätsi kokoelmia (muutoksia tiedostojen välillä) yhdestä muutoksesta toiseen, erikoisformaattissa kiintolevyllä; se voi täten uudelleen luoda sen, miltä mikä tahansa tiedosto näytti, millä tahansa ajan hetkellä, lisäämällä kaikki tarvittavat pätsit.

### 1.1.2 Keskitetyt versionhallinta järjestelmät

Seuraava suuri ongelma mihin ihmiset törmäivät on, että heillä on tarve tehdä yhteistyötä muissa järjestelmissä olevien kehittäjien kanssa. Tämän ongelman ratkaisemiseksi luotiin keskitetyt versionhallinta järjestelmät (CVCS). Nämä järjestelmät, kuten CVS, Subversion, ja Perforce, omaavat yksittäisen palvelimen joka sisältää kaikki versioidut tiedostot, ja asiakkaita jotka hakevat tiedostot tästä keskitetystä paikasta. Monet vuodet, tämä on ollut versionhallinnan standardi (katso Kuva 1-2).

Tämä asetelma tarjoaa monta etua, erityisesti paikalliseen VCS:n verrattuna. Esimerkiksi, jokainen tietää, jossain määrin, mitä kukin projektissa oleva tekee. Järjestelmänvalvojilla on hienosäädetty kontrolli siihen, mitä kukin voi tehdä; ja on paljon helpompi valvoa CVCS:ä, kuin toimia joka asiakkaan paikallisen tietokannan kanssa.

Kuitenkin, tässä asetelmassa on myös muutama vakava haittapuoli. Kaikkein selvin on keskitetty vikapiste, jota keskitetty palvelin edustaa. Jos kyseessä oleva palvelin ajetaan alas tunniksi, niin tämän tunnin aikana kukaan ei pysty tekemään yhteistyötä keskenään tai tallentamaan versioituja muutoksia mihinkään mitä he työskentelevät. Jos kiintolevy - jolla keskitetty tietokanta sijaitsee - korruptoituu, ja kunnollisia varmuuskopioita ei ole hallussa, menetät täysin kaiken - koko projektin historian, paitsi ne yksittäiset tilannekuvat joita ihmisillä sat-



**Figure 1.2:** Keskitetyn versionhallinnan diagrammi.

tuu olemaan heidän paikallisilla koneillaan. Paikalliset VCS järjestelmät kärsivät tästä samasta ongelmasta - milloin tahansa sinulla on koko projektin historia yhdessä paikassa, sinulla on riski menettää se kaikki.

### 1.1.3 Hajautetut versionhallinta järjestelmät

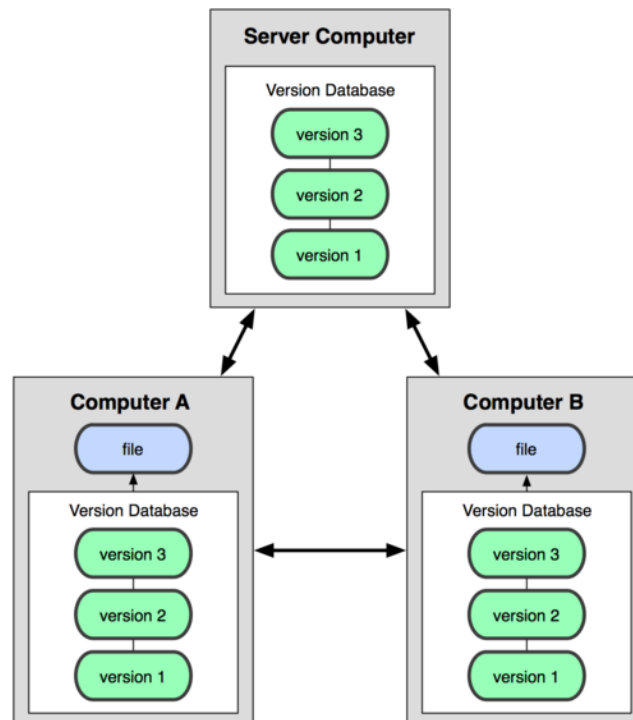
Tämä on missä hajautetut versionhallinta järjestelmät (DVCS) astuvat mukaan. DVCS:ssä (kuten Git, Mercurial, Bazaar tai Darcs), asiakkaat eivät vain hae viimeisintä tilannekuvaa tiedostoista: he täysin peilaavat koko tietolähteen. Täten, jos mikä tahansa palvelin kuolee, ja nämä järjestelmät tekivät yhteistyötä sen läpi, mikä tahansa asiakas tietolähde pystytään kopioimaan takaisin palvelimelle tiedon palauttamiseksi. Jokainen tiedonhaku on tosiasiasa täysi varmuuskopio kaikesta datasta (katso Kuva 1-3).

Lisäksi, monet näistä järjestelmistä selviytyvät melko hyvin siitä, että niillä on monia etätietolähteitä, joiden kanssa ne voivat työskennellä, joten sinä voit tehdä monenlaista yhteistyötä monenlaisien ihmisryhmien kanssa samaan aikaan, samassa projektissa. Tämä mahdollistaa sen että voit aloittaa monelaisia työkulkuja, jotka eivät ole mahdollisia keskitetyissä järjestelmissä, kuten hierarkiset mallit.

## 1.2 Gitin lyhyt historia

Kuten moni muu suuri asia elämässä, Git alkoi hippusella luovaa tuhoamista ja liekehtivää erimielisyyttä. Linux kernel on kohtalaisen suuren mittakaavan avoimen lähdekoodin projekti. Suurimman osan Linux kernelin ylläpidon elinkaaresta (1991-2002), muutokset ohjelmistoon olivat siirretty ympäriinsä pätseinä ja pakattuina tiedostoina. Vuonna 2002, Linux kernel projekti alkoi käyttämään BitKeeperiksi kutsuttua yksityistä DVCS järjestelmää.

Vuonna 2005, suhde Linux kerneliä kehittävän yhteisön ja kaupallisen BitKeeperiä kehittävän yhtiön välillä katkesi, ja ilmaisen statuksen työkalut olivat



**Figure 1.3:** Hajautettu versionhallinta diagrammi.

kumottu. Tämä johdatti Linuxin kehittäjä yhteisön (ja erityisesti Linus Torvaldsin, Linuxin luoja) kehittämään heidän omaa työkalua, perustuen oppeihin, joita he oppivat BitKeeperin käyttö-aikanaan. Muutamat uuden järjestelmän tavoitteista olivat seuraavanlaiset:

- Nopeus
- Yksinkertainen design
- Vahva tuki epälineaarille kehitykselle (tuhansia rinnakkaisia haaroja)
- Täysin hajautettu
- Pystyy tehokkaasti selviytymään suurista projekteista kuten Linuxin kernel (nopeus ja tiedon koko)

Syntymästään lähtien 2005, Git on kehittynyt ja aikuistunut helpoksi käyttää ja silti säilyttämään nämä alkuperäiset ominaisuudet. Se on uskomattoman nopea, se on erittäin tehokas suurien projektien kanssa, ja se on uskomattoman haarautuva järjestelmä epälineaarille kehitykselle (Katso Luku 3).

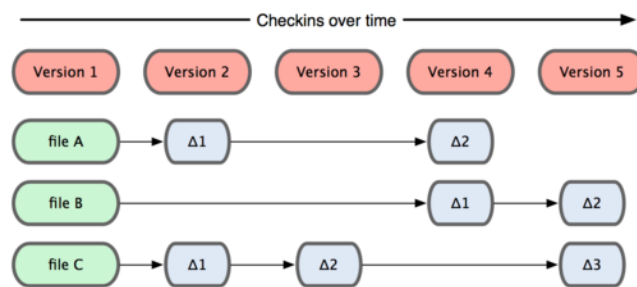
## 1.3 Git perusteet

Joten, mitä on Git pähkinänkuoressa? Tämä on tärkeä osa-alue omaksua, koska jos ymmärrät mitä Git on ja periaatteet kuinka se toimii, Gitin tehokas käyttö tulee mahdollisesti olemaan paljon helpompaa. Kun opettelet Gitin käyttöä, yritä tyhjentää mielesi asioista, joita mahdollisesti tiedät muista VCS:stä, kuten Subversionista ja Perforcesta; tämän tekeminen auttaa sinua välttämään

hienoisien sekaantumisen kun käytät työkalua. Git säilyttää ja ajattelee informaatiota huomattavasti erilailla kuin nämä muut järjestelät, vaikkakin käyttöliittymä on melko samanlainen; näiden eroavaisuuksien ymmärtäminen auttaa sinua välttämään sekaantumisia käyttäessäsi Gittiä.

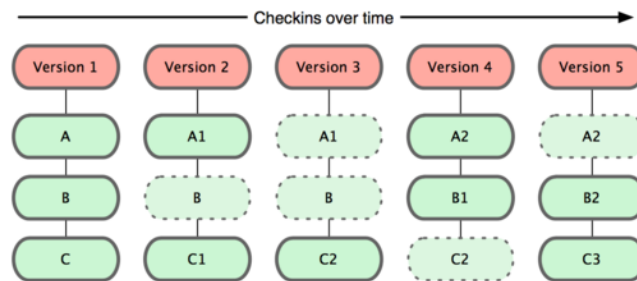
### 1.3.1 Tilannekuvia, ei eroavaisuuksia

Suurin eroavaisuus Gitin ja minkä tahansa muun VCS:n (Subversion ja ystävät mukaanlukien) on tapa jolla Git ajattelee dataansa. Käsitteellisesti, moni muu järjestelmä varastoi informaatiotonsa listana tiedostopohjaisista muutoksista. Nämä järjestelmät (CVS, Subversion, Perforce, Bazaar, ja niin edelleen) ajattelevat informaatiota jota ne varastoivat, kokoelmana tiedostoja ja ajan kuluessa jokaiseen tiedostoon tehtyinä muutoksina, kuten kuvattu Kuvassa 1-4.



**Figure 1.4:** Muut järjestelmät tapaavat varastoida dataa muutoksina joka tiedoston alkuperäiseen versioon.

Git ei ajattele tai varastoi dataansa tällä tavalla. Sen sijaan, Git ajattelee dataansa enemmän kokoelmana tilannekuvia pikkuruisesta tiedostojärjestelmästä. Joka kerta kun sinä teet pysyvän muutoksen (commit), tai tallennat projektisi tilan Gitissä, Git periaatteessa ottaa kuvan siitä miltä sinun tiedostosi näyttävät kyseisellä hetkellä ja varastoi viitteen tähän tilannekuvaan. Ollakseen tehokas, jos tiedostoja ei ole muutettu, Git ei varastoi tiedostoa uudestaan - vaan linkittää sen edelliseen identtiseen tiedostoon jonka se on jo varastoinut. Git ajattelee dataansa enemmän kuten Kuva 1-5 osoittaa.



**Figure 1.5:** Git varastoi dataa projektin tilannekuvina ajan kuluessa.

Tämä on tärkeä ero Gitin ja melkein minkä tahansa muun VCS:n välillä. Se pistää Gitin harkitsemaan uudelleen melkein joka versionhallinnan aspektia, jotka monet muut järjestelmät kopioivat edeltäneestä sukupolvesta. Tämä tekee Gitistä kuin pikkuruisen tiedostojärjestelmän, jolla on muutamia uskomattoman

tehokkaita työkaluja päälle rakennettuna, enemmän kuin simppelein VCS:n. Me tutkimme joitain hyötyjä, joita saavutat ajattelemalla datastasi tällätavoin, kun käsittelemme Gitin haarautumista Luvussa 3.

### 1.3.2 Lähes jokainen operaatio on paikallinen

Monet operaatiot Gitissä tarvitsevat ainoastaa paikallisia tiedostoja ja resursseja operoidakseen - yleensä mitään informaatiota toiselta koneelta tietoverkostasi ei tarvita. Jos olet tottunut CVCS:n, joissa suurin osa operaatioista sisältää tietoverkon viiveen, tämä Gitin aspekti laittaa sinut ajattelemaan, jotta nopeuden jumalat ovat siunanneet Gitin sanoinkuvaamattomilla voimilla. Koska sinulla on projektisi koko historia paikallisella levylläsi, suurinosa operaatioista näyttää melkein välittömiltä.

Esimerkiksi, selataksesi projektisi historiaa, Gitin ei tarvitse mennä ulkoiselle palvelimelle ottaakseen historian ja näyttääkseen sen sinulle - se yksinkertaisesti lukee sen suoraan sinun paikallisesta tietokannastasi. Tämä tarkoittaa, että näet projektin historian melkein välittömästi. Jos haluat nähdä muutokset tiedoston nykyisen version ja kuukausi sitten olleen version välillä, Git voi katsoa tiedoston tilannekuvan kuukausi sitten ja tehdä paikallisen muutoslaskelman, sen sijaan, jotta sen pitäisi joko kysyä etäpalvelimelta sitä tai jotta sen tarvitsisi vetää vanhempi versio etäpalvelimelta, jotta se voisi tehdä sen paikallisesti.

Tämä myös tarkoittaa sitä, että on hyvin vähän asioita joita et voi tehdä, jos olet yhteydetön tai poissa VPN:stä. Jos nouset lentokoneeseen tai junaan ja haluat tehdä vähän töitä, voit iloisesti tehdä pysyviä muutoksia kunnes saat tietoverkon takaisin ja voit lähettää muutoksesi. Jos menet kotiin ja et saa VPN asiakasohjelmaasi toimimaan oikein, voit yhä työskennellä. Monissa muissa järjestelmissä, tämän tekeminen on joko mahdotonta tai kivuliasta. Perforcessa, esimerkiksi, et voi tehdä paljoa mitään, silloin kun et ole yhteydessä palvelimeen; Subversiossa ja CVS:ssä voit editoida tiedostojasi, mutta et voi tehdä pysyviä muutoksia tietokantaasi (koska tietokantasi on yhteydetön). Tämä kaikki saattaa vaikuttaa, ettei se ole nyt niin suuri juttu, mutta saatat yllättyä kuin suuren muutoksen se voi tehdä.

### 1.3.3 Git on eheä

Kaikki Gitissä on tarkistussummattu ennen kuin se on varastoitu ja on tämän jälkeen viitattu tällä tarkistussummalla. Tämä tarkoittaa, että on mahdotonta muuttaa minkään tiedoston sisältöä tai kansiota ilman, ettei Git tietäisi siitä. Tämä toiminnallisuus on rakennettu Gittiin alimmalla tasolla ja se on kiinteä osa sen filosofiaa. Et voi menettää informaatiota tiedonsiirrossa tai saada tiedostoihisi korruptiota, ilman ettei Git pystyisi sitä huomaamaan.

Mekanismi jota Git käyttää tarkistussummaan on kutsuttu SHA-1 tarkisteeksi. Tämä on 40-merkkinen merkkijono, joka koostuu hexadesimaali merkeistä (0-9 ja a-f) ja joka on Gitissä laskettu tiedoston sisältöön tai hakemisto rakenteeseen pohjautuen. SHA-1 tarkiste voi näyttää tällaiselta:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Voit nähdä nämä tarkiste arvot jokapuolella Gitissä, koska se käyttää niitä niin paljon. Itseasiassa, Git varastoi kaiken, ei pohjautuen tiedoston nimeen, vaan Gitin tietokantaan osoitteistavaan sisällön tarkiste arvoon.

### 1.3.4 Git yleensä vain lisää dataa

Kun teet toimintoja Gitissä, melkein kaikki niistä ainoastaan lisää dataa Gitin tietokantaan. On erittäin vaikea saada järjestelmä tekemään mitään, mikä olisi kumoamaton tai saada se poistamaan dataa millään tavoin. Kuten missä tahansa VCS:ssä, voit menettää tai sotkea muutoksia, joita ei ole vielä tehty pysyviksi; mutta sen jälkeen kun teet pysyvän tilannekuvan muutoksen Gittiin, se on erittäin vaikeaa hävittää, etenkin jos sinä säännöllisesti työnnät tietokantasi toiseen tietolähteeseen.

Tämä tekee Gitin käyttämisestä hauskaa, koska me tiedämme, että voimme kokeilla erillaisia asioita ilman vaaraa, että vakavasti sotkisimme versionhallintamme. Syväluotaavamman tarkastelun siihen miten Git varastoi dataansa ja kuinka voit palautua datan joka näyttää hävinneeltä, katso "Kuurien alla" Kappaleesta 9.

### 1.3.5 Kolme tilaa

Nyt, lue huolellisesti. Tämä on pääasia muistaa Gitistä, jos sinä haluat lopun opiskelu prosessistasi menevän sulavasti. Gitillä on kolme pääasiallista tilaa, joissa tiedostosi voivat olla: pysyvästi muutettu, muutettu, ja lavastettu. Pysyvästi muutettu tarkoittaa, että data on turvallisesti varastoitu sinun paikalliseen tietokantaasi. Muutettu tarkoittaa, että olet muuttanut tiedostoa, mutta et ole tehnyt vielä pysyvää muutosta tietokantaasi. Lavastettu tarkoittaa, että olet merkannut muutetun tiedoston nykyisessä versiossaan menemään seuraavaan pysyvään tilannekuvaan.

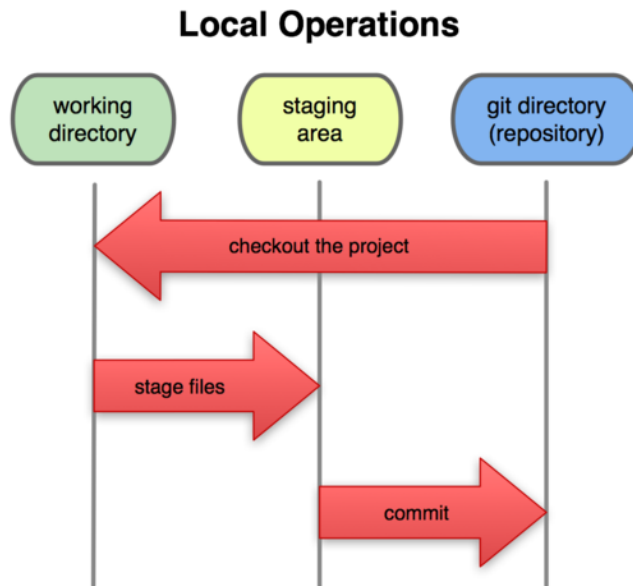
Tämä johdattaa meidät kolmeen seuraavaan osaan Git projektia: Git hakemisto, työskentely hakemisto, ja lavastus alue.

Git hakemisto on paikka, mihin Git varastoi metadatan ja olio tietokannan projektillesi. Tämä on kaikkein tärkein osa Gittiä, ja se sisältää sen, mitä kopioidaan, kun kloonat tietovaraston toiselta tietokoneelta.

Työskentely hakemisto on yksittäinen tiedonhaku yhdestä projektin versiosta. Nämä tiedostot ovat vedetty ulos pakatusta tietokannasta Git hakemistosta ja si-joitettu levylle sinun käytettäväksesi tai muokattavaksesi.

Lavastus alue on yksinkertainen tiedosto, yleensä se sisältyy Git hakemistoosi, joka varastoi informaatiota siitä, mitä menee seuraavaan pysyvään muutokseen. Sitä on joskus viitattu indeksiksi, mutta on tulossa standardiksi viitata sitä lavastus alueeksi.

Normaali Git työnkulku menee jokseenkin näin:



**Figure 1.6:** Työskentely hakemisto, lavastus alue, ja git hakemisto.

1. Muokkaat tiedostoja työskentely hakemistossasi.
2. Lavastat tiedostosi, lisäten niistä tilannekuvia lavastus alueellesi.
3. Teet pysyvän muutoksen, joka ottaa tiedostot kuin ne ovat lavastus alueella ja varastoi tämän tilannekuvan pysyvästi sinun Git tietolähteeseesi.

Jos tietty versio tiedostosta on git hakemistossa, se on yhtä kuin pysyvä muutos. Jos sitä on muokattu, mutta se on lisätty lavastus alueelle, se on lavastettu. Ja jos se on muuttunut siitä kun se on haettu, mutta sitä ei ole lavastettu, se on muutettu. Luvussa 2, opit enemmän näistä tiloista ja kuinka voit hyödyntää niitä tai skipata lavastus osan kokonaan.

## 1.4 Gitin asennus

Ryhdytäänpä käyttämään Gittiä. Ensimmäiset asiat ensin - sinun täytyy asentaa se. Voit saada sen monella tavalla; kaksi yleisintä tapaa on asentaa se suoraan lähdekoodista tai asentaa jo olemassa oleva paketti sovellusalustallesi.

### 1.4.1 Asennus suoraan lähdekoodista

Jos voit, on yleensä hyödyllistä asentaa Git suoraan lähdekoodista, koska näin saat kaikkein viimeisimmän version. Jokainen Gitin versio tapaa sisältää hyödyllisiä käyttöliittymä parannuksia, joten uusimman version hakeminen on yleensä paras reitti, jos tunnet olosi turvalliseksi kääntäessäsi ohjelmistoa sen lähdekoodista. On yleinen tilanne, että moni Linux jakelu sisältää erittäin vanhoja paketteja; joten, jollet käytä erittäin päivitettyä jakelua tai ellet käytä back-portteja, lähdekoodista asennus voi olla paras ratkaisu.

Asentaaksesi Gitin, tarvittavat seuraavat kirjastot joista Git on riippuvainen: curl, zlib, openssl, expat, ja libiconv. Esimerkiksi, jos olet järjestelmässä, jossa

on yum (kuten Fedora) tai apt-get (kuten Debian pohjaiset järjestelmät), voit käyttää yhtä näistä komennoista asentaaksesi kaikki riippuvaisuudet:

```
$ yum install curl-devel expat-devel gettext-devel \
  openssl-devel zlib-devel

$ apt-get install libcurl4-gnutls-dev libexpat1-dev gettext \
  libz-dev
```

Kun sinulla on kaikki tarvittavat riippuvuudet, voit mennä eteenpäin ja kiskaista uusimman tilannekuvan Gitin verkkosivuilta:

<http://git-scm.com/download>

Tämän jälkeen, käännä ja asenna:

```
$ tar -zxf git-1.6.0.5.tar.gz
$ cd git-1.6.0.5
$ make prefix=/usr/local all
$ sudo make prefix=/usr/local install
```

Kun tämä on tehty, voit myös ottaa Gitin päivitykset Gitin itsensä kautta:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
```

## 1.4.2 Asennus Linuxissa

Jos haluat asentaa Gitin Linuxille binääri asennusohjelman kautta, voit yleensä tehdä näin perus paketinhallinnointi ohjelmalla, joka tulee julkaisusi mukana. Jos olet Fedorassa, voit käyttää yumia:

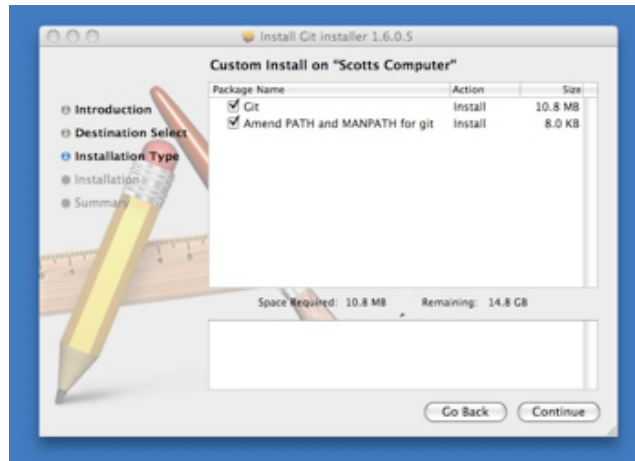
```
$ yum install git-core
```

Tai jos ole Debian alustaisessa julkaisussa kuten Ubuntu, kokeile apt-gettiä:

```
$ yum install git-core
```

## 1.4.3 Asennus Macilla

On olemassa kaksi helppoa tapaa asentaa Git Macille. Helpoin on käyttää graafista Git asennusohjelmaa, jonka voit ladata Googlen Code verkkosivuilta (katso Kuva 1-7):



**Figure 1.7:** Git OS X asennusohjelma.

<http://code.google.com/p/git-osx-installer>

Toinen pääasiallinen tapa on asentaa Git MacPortsien kautta (<http://www.macports.org>). Jos sinulla on MacPorts asennettuna, asenna Git näin:

```
$ sudo port install git-core +svn +doc +bash_completion +gitweb
```

Sinun ei tarvitse asentaa kaikkia ekstroista, mutta varmaankin haluat sisältää +svn ekstran tapauksessa, että sinun koskaan tarviiä käyttää Gittiä Subversion tietolähteiden kanssa (katso Luku 8).

#### 1.4.4 Asennus Windowsilla

Gitin asennus Windowsilla on erittäin helppoa. msysGit projektilla on yksi helpoimmista asennusmenetelmistä. Yksinkertaisesti lataa asennus exe tiedosto Googlen Code verkkosivuilta ja suorita se:

<http://code.google.com/p/msysgit>

Asennuksen jälkeen, sinulla on kummatkin, komentorivi versio (sisältäen SSH-asiakasohjelman, joka osoittautuu hyödylliseksi myöhemmin) ja standardi graafinen käyttöliittymä.

## 1.5 Ensikerran Git asetukset

Nyt kun sinulla on Git järjestelmässäsi, haluat tehdä muutamia asioita kustomoidaksesi Git ympäristöäsi. Sinun tulisi tarvita tehdä nämä asiat vain kerran; ne säilyvät Gitin päivitysten välissä. Voit myös muuttaa niitä minä tahansa hetkenä ajamalla komennot läpi uudestaan.

Git tulee mukanaan työkalu, jota git configiksi kutsutaan, tämä työkalu antaa sinun hakea ja asettaa konfiguraatio muuttujia, jotka kontrolloivat kaikkia aspekteja siitä, miltä Git näyttää ja miten se operoi. Nämä muuttujat voidaan varastoida kolmeen erilliseen paikkaan:

- `/etc/gitconfig` tiedosto: Sisältää arvot jokaiselle käyttäjälle järjestelmässä ja heidän kaikki tietolähteensä. Jos annat option `--system git config:`lle, se lukee ja kirjoittaa erityisesti tästä tiedostosta.
- `~/.gitconfig` tiedosto: Tämä on erityisesti käyttäjällesi. Voit tehdä Gitin lukemaan ja kirjoittamaan erityisesti tähän tiedostoon antamalla option `--global`.
- `config` tiedosto on Git hakemistossa (tämä on, `.git/config`) missä tahansa tietolähteistä jota juuri nyt käytät: Tämä on erityisesti tälle kyseessä olevalle tietolähteelle. Jokainen taso ylikirjoittaa arvoja aikaisemmilta tasoilta, joten arvot `.git/config`:ssa päihittää arvot `/etc/gitconfig`:ssa.

Windows järjestelmissä, Git etsii `.gitconfig` tiedostoa `$HOME` hakemistosta (`C:\Documents and Settings\%USER` suurimmalle osalle ihmisistä). Se myös yhä etsii `'/etc/gitconfig'`:a, vaikkakin se on suhteellinen `MSys` juureen, joka on missä tahansa minne päätät asentaa Gitin sinun Windows järjestelmässäsi, kun suoritat asennusohjelman.

### 1.5.1 Identiteettisi

Ensimmäinen asia joka sinun tulisi tehdä Gittiä asentaessasi, on asettaa käyttäjänimesi ja sähköposti osoitteesi. Tämä on tärkeää, koska jokainen pysyvä muutos jonka Gitillä teet, käyttää tätä informaatiota, ja se on muuttumattomasti leivottu pysyviin muutoksiisi, joita liikuttelet ympäriinsä:

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Tälläkin kertaa, sinun täytyy tehdä tämä ainoastaan kerran, jos annat `--global` option, koska silloin Git käyttää aina tätä informaation mitä tahansa teetkään järjestelmässäsi. Jos haluat yliajaa tämän toisella nimellä tai sähköposti osoitteella tietyille projekteille, voit aina ajaa komennon ilman `--global` optiota, kun olet projektissasi.

### 1.5.2 Editorisi

Nyt kun identiteettisi on asetettu, voit konfiguroida oletus teksti-editorisi, jota käytetään kun Git tarvitsee sinua kirjoittamaan viestin. Oletusarvoisesti, Git käyttää järjestelmäsi oletus editoria, joka yleensä on `Vi` tai `Vim`. Jos haluat käyttää erillistä teksti editoria, kuten `Emacs`, voit tehdä seuraavanlaisesti:

```
$ git config --global core.editor emacs
```

### 1.5.3 Diff työkalusi

Seuraava hyödyllinen optio jota saatat haluta konfiguroida, on oletus diff työkalu, jota käytetään yhdentämiskonfliktien selvittämiseen. Sanotaan vaikka, että haluat käyttää vimdiffiä:

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

Git hyväksyy seuraavat yhdentämistyökalut kdiff3, tkdiff, meld, xxdiff, emerge, vimdiff, gvimdiff, ecmerge, ja opendiff. Voit myös käyttää kustomoitua työkalua; katso Luku 7 saadaksesi lisäinformaatiota tähän.

### 1.5.4 Tarkista asetuksesi

Jos haluat tarkistaa asetuksesi, sinä voit käyttää `git config --list` komentoa listataksesi kaikki asetukset, jotka Git löytää tällä hetkellä:

```
$ git config --list
user.name=Scott Chacon
user.email=schacon@gmail.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

Voit nähdä avaimia useammin kuin kerran, koska Git lukee saman avaimen monesta eri tiedostosta (/etc/gitconfig:sta ja ~/.gitconfig:sta, esimerkkinä). Tässä tapauksessa, Git käyttää viimeistä arvoa jokaisella yksittäiselle avaimelle jonka se näkee.

Voit myös tarkistaa mitä Git ajattelee tietyin avaimen arvosta, kirjoittamalla `git config key`:

```
$ git config user.name
Scott Chacon
```

## 1.6 Avun saanti

Jos koskaan tarvitset apua Gittiä käytäessäsi, on olemassa kolme tapaa päästä minkä tahansa git komennon manuaalisivulle (manpage):

```
$ git help <verb>
$ git <verb> --help
$ man git-<verb>
```

Esimerkiksi, saat manuaalisivun config komennolle suorittamalla:

```
$ git help config
```

Nämä komennot ovat mukavia, koska pääset niihin käsiksi jokapaikasta, jopa yhteydettömässä tilassa. Jos manuaalisivut ja tämä kirja eivät ole tarpeeksi ja tarvitset henkilökohtaista apua, voit yrittää `#git` tai `#github` kanavia Freenoden IRC palvelimella ([irc.freenode.net](http://irc.freenode.net)). Nämä kanavat ovat säännöllisesti täynnä satoja ihmisiä, jotka ovat erittäin osaavia Gitin käyttäjiä ja ovat usein halukkaita auttamaan.

## 1.7 Yhteenveto

Sinulla tulisi olla perus ymmärtämys siitä mikä Git on ja kuinka se eroaa muista CVCS järjestelmistä, joita mahdollisesti käytät. Sinulla myös tulisi olla toimiva versio Gitistä järjestelmässäsi, joka on konfiguroitu sinun henkilökohtaisella identiteetilläsi. Joten, nyt on aika opetella Gitin perusteita.



## Chapter 2

# Gitin perusteet

Jos voit lukea vain yhden kappaleen päästäksesi vauhtiin Gitin kanssa, se on tämä. Tämä kappale sisältää jokaisen peruskomennon jonka tarvitset tehdäksesi valtavan määrän asioita, joiden kanssa viimein tulet käyttämään aikaasi Gitillä työskennellessäsi. Tämän kappaleen lopussa, sinun tulisi pystyä konfiguroimaan ja alustamaan tietolähde, aloittamaan ja lopettamaan tiedostojen jäljitys, ja lavastaa ja tehdä pysyviä muutoksia. Me myös näytämme sinulle kuinka asettaa Git niin, että se jättää tietyt tiedostot ja tiedostomallit huomioimatta, kuinka kumota virheet nopeasti ja helposti, kuinka selata projektisi historiaa ja tarkastella muutoksia pysyvien muutosten välillä, ja kuinka työntää ja vetää etätietolähteistä.

## 2.1 Git tietolähteen hankinta

Voit hankkia itsellesi Git projektin käyttäen kahta yleistä lähestymistapaa. Ensimmäinen ottaa jo olemassa olevan projektin tai hakemiston ja tuo sen Gittiin. Toinen kloonaa jo olemassa olevan Git tietolähteen toiselta palvelimelta.

### 2.1.1 Tietolähteen alustaminen jo olemassa olevalle hakemistolle

Jos aloitat jo olemassa olevan projektin jäljittämisen Gitillä, sinun täytyy mennä projektisi hakemistoon ja kirjoittaa

```
$ git init
```

Tämä luo uuden alihakemiston nimeltään `.git`, joka sisältää kaikki tarvittavat tietolähde tiedostot - luurangon Git tietolähteelle. Tällä hetkellä, mitään projektissasi ei vielä jäljitetä. (Katso Luku 9 saadaksesi enemmän tietoa siitä, mitä tiedostoja tarkalleen ottaen juuri luomasi `.git` hakemisto sisältää.)

Jos haluat aloittaa versionhallinan jo olemassa oleville tiedostoille (tyhjän kansion sijaan), sinun täytyy mitä luultavammin aloittaa näiden tiedostojen jäljit-

täminen ja tehdä alustava pysyvä muutos. Sinä saavutat tämän muutamalla git komennolla, joista ensimmäiset määrittävät mitä tiedostoja haluat jäljittää ja joita seuraa pysyvän muutoksen luonti:

```
$ git add *.c
$ git add README
$ git commit -m 'initial project version'
```

Me käymme pian läpi mitä nämä komennot tekevät. Tällä hetkellä, sinulla on Git tietolähde, joka jäljittää tiedostoja sekä alustava pysyvä muutos.

### 2.1.2 Olemassa olevan tietolähteen kloonaus

Jos haluat kopion olemassa olevasta tietolähteestä - esimerkiksi, projektista johon haluat olla osallisena - komento jonka tarviat on git clone. Jos muut VCS järjestelmät, kuten Subversion ovat sinulle tuttuja, niin huomaat, että komento on clone, eikä checkout. Tämä on tärkeä ero - Git saa kopion melkein kaikesta datasta mitä palvelimella on. Jokainen versio jokaisesta tiedostosta projektin historiasta tulee vedetyksi, kun suoritat git clone-komennon. Itseasiassa, jos palvelimesi levy korruptoituu, voit käyttää mitä tahansa klooneista, miltä tahansa asiakas sovellukselta, asettaaksesi palvelimen takaisin tilaan, jossa se oli kun se kloonattiin (voit menettää jotain palvelinpuolen sovelluskoukkuja ja muuta, mutta kaikki versioitu data on tallessa - katso Luku 4 tarkempia yksityiskohtia varten).

Kloonaat tietolähteen git clone [url]-komennolla. Esimerkiksi, jos haluat kloonata Gritiksi kutsutun Ruby Git kirjaston, voit tehdä sen näin:

```
$ git clone git://github.com/schacon/grit.git
```

Tämä luo hakemiston nimeltään "grit", alustaa .git hakemiston sen sisään, vetää kaiken datan tietolähteestä, ja hakee viimeisimmän version työkopion. Jos menet uuteen grit hakemistoon, näet projektin tiedostot, valmiina työtä varten tai käytettäväksi. Jos haluat kloonata tietolähteen hakemistoon, joka on nimetty joksikin muuksi kuin grit, voit antaa nimen seuraavanlaisella komentorivi optiolla:

```
$ git clone git://github.com/schacon/grit.git mygrit
```

Tämä komenta tekee saman asian kuin edellinen, mutta kohde hakemisto on nimeltään mygrit.

Git has a number of different transfer protocols you can use. The previous example uses the git:// protocol, but you may also see http(s):// or user@server:/path.git, which uses the SSH transfer protocol. Chapter 4 will

introduce all of the available options the server can set up to access your Git repository and the pros and cons of each.

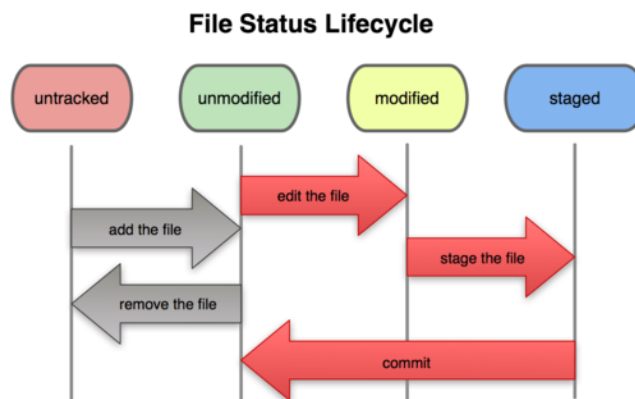
Gitissä on monta erilaista siirto protokollaa jota voit käyttää. Edellinen esimerkki käyttää `git://` protokollaa, mutta voit myös nähdä `http(s)://` tai `user@server:/path.git`, joka käyttää SSH siirto protokollaa. Luku 4 esittelee kaikki saatavilla olevat optiot, miten palvelin voi asettaa Git tietolähteen, sekä jokaisin hyvät ja huonot puolet.

## 2.2 Muutosten tallennus tietolähteeseen

Sinulla on vilpitön Git tietolähde ja haettu- tai työkopio projektin tiedostoista. Sinun täytyy tehdä joitain muutoksia ja pysyviä tilannekuvia näistä muutoksista sinun tietolähteeseesi joka kerta, kun projekti saavuttaa tilan jonka haluat tallentaa.

Muista, että jokainen tiedosto työhakemistossasi, voi olla yhdessä kahdesta tilasta: jäljitetty tai jäljittämätön. Jäljitetyt tiedostot ovat tiedostoja, jotka olivat viimeisimmässä tilannekuvassa; ne voivat olla muokkaamattomia, muokattuja, tai lavastettuja. Jäljittämättömät tiedostot ovat kaikkea muuta - mitkä tahansa tiedostot työhakemistossasi, jotka eivät olleet viimeisimmässä tilannekuvassa ja jotka eivät ole lavastusalueella. Kun ensimmäisen kerran kloonat tietolähten, kaikki tiedostoistasi tulevat olemaan jäljitettyjä ja muokkaamattomia, koska sinä juuri hait ne ja et ole muokannut vielä mitään.

Editoidessasi tiedostoja, Git näkee ne muokattuina, koska olet muuttanut niitä viimeisimmän pysyvän muutoksen jälkeen. Lavastat nämä muutetut tiedostot, jonka jälkeen muutat kaikki lavastetut muutokset pysyvästi, ja sykli toistuu. Tämä elämänsykli on kuvattu Kuvassa 2-1.



**Figure 2.1:** Tiedostojesi tilan elämänsykli.

### 2.2.1 Tiedostojesi tilan tarkistaminen

Päätyökalu tiedostojesi eri tilojen selvittämiseen on `git status` komento. Jos ajat tämän komennon suoraan kloonauksen jälkeen, sinun tulisi nähdä jotain vastaavaa:

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

Tämä tarkoittaa, että sinulla on puhdas työhakemisto - toisin sanoen, se ei sisällä jäljitettyjä tai muutettuja tiedostoja. Git ei myöskään näe yhtään jäljittämätöntä tiedostoa, muuten ne olisi listattu näkymään. Viimein, komento kertoo sinulle missä haarassa olet. Tällä hetkellä se on aina master haara, joka on oletusarvo; sinun ei tarvitse huolehtia siitä nyt. Seuraavan kappale käy läpi haarautumiset ja viittaukset yksityiskohtaisesti.

Sanotaan vaikka, että lisäät uuden tiedoston projektiin, vaikka yksinkertaisen README-tiedoston. Jos tiedosto ei ollut olemassa ennen, ja ajat `git status`-komennon, näet jäljittämättömän tiedoston tällä tavoin:

```
$ vim README
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
# README
nothing added to commit but untracked files present (use "git add" to track)
```

Voit nähdä, että juuri luomasi README tiedosto on jäljittämätön, koska se on otsikon "Untracked files" alla tilatulosteessa. Jäljittämätön periaatteessa tarkoittaa, että Git näkee tiedoston, jota ei ollut edellisessä tilannekuvassa (pysyvässä muutoksessa); Git ei aloita sisällyttämään sitä sinun pysyviin muutostilannekuvii, ennen kuin sinä vartavasten kerrot sen tehdä niin. Se tekee tämän, että et vahingossa alkaisi lisäämään generoituja binaaritiedostoja tai muita tiedostoja, joita et tarkoittanut lisätä. Haluat lisätä README:n, joten aloitetaan jäljittämään tiedostoa.

## 2.2.2 Uusien tiedostojen jäljitys

Jotta voisit jäljittää uusia tiedostoja, sinun täytyy käyttää `git add`-komentoa. Aloittaaksesi README tiedoston jäljittämisen, voit ajaa tämän:

```
$ git add README
```

Jos ajat tila komennon uudestaan, näet että README tiedostosi on nyt jäljitetty ja lavastettu:

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

Voit nähdä, että se on lavastettu, koska se on otsikon “Changes to be committed” alla. Jos teet pysyvän muutoksen tässä kohtaa, versio tiedostosta sillä hetkellä kun ajoit ‘git add’-komennon on se, joka tulee olemaan historian tilannekuvassa. Voit palauttaa mieleen hetken, jolloin ajoit ‘git init’-komennon aikaisemmin, ajoit sen jälkeen ‘git add’-komennon - tämä komento aloitti tiedostojen jäljittämisen hakemistossa. Git add komento ottaa polunnimen joko tiedostolle tai hakemistolle; jos se on hakemisto, niin komento lisää kaikki tiedostot hakemiston alta rekursiivisesti.

### 2.2.3 Muutettujen tiedostojen lavastus

Muutetaanpa tiedostoa, joka on jo jäljitetty. Jos muutat aikaisemmin jäljitettyä benchmarks.rb tiedostoa ja sen jälkeen ajat status komennon uudestaan, saat suunnilleen tämänäköisen tulosteen:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# new file:   README
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
# modified:   benchmarks.rb
#
```

Benchmarks.rb tiedosto näkyy kohdan “Changed but not updated” alla - mikä tarkoittaa, että tiedostoa jota jäljitetään on muokattu työskentely hakemistossa, mutta sitä ei vielä ole lavastettu. Lavastaaksesi sen, ajat git add komennon (se on monitoimikomando - käytät sitä aloittaaksesi uusien tiedostojen jäljittämisen, lavastaaksesi tiedostoja, ja tehdäksesi muita asioita, kuten merkataksesi liitos-konflikti tiedostot ratkaistuksi). Ajetaanpa nyt git add-komento lavastaaksemme benchmarks.rb tiedoston, ja sitten git status-komento uudestaan:

```
$ git add benchmarks.rb
```

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

Kummatkin tiedostot ovat lavastettuja ja tulevat menevät seuraavaan pysyvään muutokset. Tällä kyseisellä hetkellä, oletetaan että muistat pienen muutoksen, jonka haluat tehdä benchmarks.rb tiedostoon, ennen kuin teet pysyvää muutosta. Avaat tiedoston uudestaan ja muutat sitä, jonka jälkeen olet valmis tekemään pysyvän muutoksen. Mutta ajetaan silti `git status`-komento vielä kerran:

```
$ vim benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# new file:   README
# modified:  benchmarks.rb
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
# modified:  benchmarks.rb
#
```

Mitä ihmettä? Nyt benchmarks.rb on listattu sekä lavastettuna että lavastamattomana. Miten se on mahdollista? Tapahtuu niin, että Git lavastaa tiedoston juuri sellaisena kuin se on, kun ajat `'git add'`-komennon. Jos teet pysyvän muutoksen nyt, niin benchmark.rb tiedoston versio sillä hetkellä, kun ajoit `'git add'`-komennon, on se, joka menee tähän pysyvään muutokseen, eikä se tiedoston versio, joka on työskentely hakemistossasi sillä hetkellä, kun ajat `'git commit'`-komennon. Jos muutat tiedostoa sen jälkeen, kun olet ajanut `git add`-komennon, täytyy sinun ajaa `git add` uudestaan lavastaaksesi uusimman version tiedosta:

```
$ git add benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# new file:   README
# modified:  benchmarks.rb
```

#

## 2.2.4 Tiedostojen sivuuttaminen

Usein, sinulla on luokka tiedostoja, joita et halua Gitin automaattisesti lisäävän tai edes näyttävän, että ne ovat jäljittämättömiä. Näitä ovat yleensä automaattisesti generoidut tiedostot, kuten logi-tiedostot tai tiedostot, jotka sinun rakennearjestelmä on luonut. Tällaisissa tapauksissa, voit luoda tiedoston listaus malleja löytääksesi ne, .gitignore tiedostoon. Tässä on esimerkki .gitignore tiedostosta:

```
$ cat .gitignore
*. [oa]
*~
```

Ensimmäinen rivi kertoo Gitille, että jokainen tiedosto joka loppuu .o tai .a päätteeseen, sivuutetaan - näitä ovat mm. olio ja arkisto tiedostot, jotka voivat olla ohjelmakoodisi rakennuksen tulos. Toinen rivi kertoo Gitille, että kaikki tiedostot, jotka loppuvat tildeen (~), mitkä yleensä ovat monen teksti-editorin, kuten Emacsin tapa merkata väliaikaisia tiedostoja, sivuutetaan. Voit myös sisällyttää log, tmp, tai pid hakemiston; automaattisesti generoidun dokumentaation; ja niin edelleen. Välttääksesi sellaisten tiedostojen joutumisten Git tietolähteeseen, joita et sinne alunperinkään halua menevän, on .gitignore tiedoston asettaminen ennen varsinaisen työskentelyn aloittamista yleensä hyvä idea.

Säännöt malleille joita voit pistää .gitignore tiedostoon ovat seuraavanlaiset:

- Tyhjät rivit ja rivit jotka alkaa # merkillä sivuutetaan.
- Yleiset keräysmallit toimivat.
- Voit päättää malleja kenoviivalla (/) määrittääksesi hakemiston.
- Voit kieltää mallin aloittamalla sen huutomerkillä (!).

Keräysmallit ovat kuin yksinkertaistettuja 'säännöllisiä ilmaisuja' (regular expressions), joita komentorivit käyttävät. Asteriski (\*) löytää nolla tai enemmän merkkiä; [abc] löytää jokaisen merkin, joka on hakasulkujen sisällä (tässä tapauksessa a:n, b:n tai c:n); kysymysmerkki (?) löytää yksittäisen merkin; hakasulut, jotka ovat väliviivalla erotettujen merkkien ympärillä ([0-9]) löytävät jokaisen merkin, joka on merkkien välissä, tai on itse merkki (tässä tapauksessa merkit 0:sta 9:n).

Tässä toinen esimerkki .gitignore tiedostosta:

```
# kommentti - tämä sivuutetaan
*.a      # ei .a tiedostoja
!lib.a   # mutta jäljitä lib.a, vaikka sivuutatkin .a tiedostot yllä
```

```
/TODO      # sivuttaa vain juuren TODO tiedosto, ei subdir/TODO hakemistoa
build/     # sivuttaa kaikki tiedostot build/ hakemistosta
doc/*.txt  # sivuttaa doc/notes.txt, mutta ei doc/server/arch.txt
```

## 2.2.5 Lavastettujen ja lavastattomien muutosten tarkastelu

Jos `git status`-komento on liian epämääräinen sinulle - haluat tietää tarkalleen mitä on muutettu, et ainoastaan sitä, mitkä tiedostot ovat muuttuneet - voit käyttää `git diff`-komentoa. Me käsittelemme `git diff`-komennon yksityiskohtaisesti myöhemmin; mutta sinä tulet mahdollisesti käyttämään sitä useasti, vastataksesi näihin kahteen kysymykseen: Mitä olet muuttanut, mutta et ole vielä lavastanut? Ja mitä sellaista olet lavastanut, josta olet tekemässä pysyvän muutoksen? Vaikkakin `git status` vastaa näihin kysymyksiin yleisesti, `git diff` näyttää sinulle tarkalleen ne rivit, jotka on lisätty ja poistettu - vähän niin kuin pätsi.

Sanotaan vaikka, että muokkaat ja lavastat README tiedostoa uudestaan, jonka jälkeen muokkaat benchmarks.rb tiedostoa, ilman että lavastat sitä. Jos ajat `status`-komennon, näet jälleen kerran jotain tällaista:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# new file:   README
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
# modified:   benchmarks.rb
#
```

Nähdäksesi mitä olet muuttanut, mutta et vielä lavastanut, kirjoita `git diff` ilman mitään muita argumentteja:

```
$ git diff
diff --git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..da65585 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main
     @commit.parents[0].parents[0].parents[0]
   end

+   run_code(x, 'commits 1') do
+     git.commits.size
+   end
```

```
+
  run_code(x, 'commits 2') do
    log = git.commits('master', 15)
    log.size
```

Tämä komento vertailee sitä, mitä sinun työskentely hakemistossa on verrattuna siihen, mitä sinun lavastus alueellasi on. Tulos kertoo tekemäsi muutokset, joita et ole vielä lavastanut.

Jos haluat nähdä mitä sellaista olet lavastanut, joka menee seuraavaan pysyvään muutokseen, voit käyttää `git diff --cached`-komentoa. (Git versiossa 1.6.1:stä lähtien, voit käyttää myös `git diff --staged`-komentoa, joka on helpompi muistaa.) Tämä komento vertailee lavastettuja muutoksia viimeisimpään pysyvään muutokseen.

```
$ git diff --cached
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README2
@@ -0,0 +1,5 @@
+grit
+ by Tom Preston-Werner, Chris Wanstrath
+ http://github.com/mojombo/grit
+
+Grit is a Ruby library for extracting information from a Git repository
```

On tärkeää ottaa huomioon, että `git diff` itsessään ei näytä kaikkia muutoksia viimeisimmästä pysyvästä muutoksesta lähtien - vain muutokset jotka ovat yhä lavastamattomia. Tämä voi olla sekavaa, koska kun olet lavastanut kaikki muutoksesi, `git diff` ei anna ollenkaan tulostetta.

Toisena esimerkkinä, jos lavastat `benchmarks.rb` tiedoston ja sitten muokkaat sitä, voit käyttää `git diff`-komentoa nähdäksesi tiedoston lavastetut muutokset ja lavastamattomat muutokset:

```
$ git add benchmarks.rb
$ echo '# test line' >> benchmarks.rb
$ git status
# On branch master
#
# Changes to be committed:
#
#   modified:   benchmarks.rb
#
# Changed but not updated:
#
#   modified:   benchmarks.rb
#
```

Nyt voit käyttää `git diff`-komentoa nähdäksesi mitä on yhä lavastamatta:

```
$ git diff
diff --git a/benchmarks.rb b/benchmarks.rb
index e445e28..86b2f7c 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -127,3 +127,4 @@ end
  main()

##pp Grit::GitRuby.cache_client.stats
+# test line
```

ja `git diff --cached`-komentoa nähdäksesi mitä olet lavastanut tähän mennessä:

```
$ git diff --cached
diff --git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..e445e28 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main
  @commit.parents[0].parents[0].parents[0]
  end

+   run_code(x, 'commits 1') do
+     git.commits.size
+   end
+
  run_code(x, 'commits 2') do
    log = git.commits('master', 15)
    log.size
```

## 2.2.6 Pysyvien muutoksien tekeminen

Nyt kun lavastusalueesi on asetettu niin kuin sen haluat, voit tehdä muutoksiasi pysyviä. Muista, että kaikki, mikä vielä on lavastamatta - mitkä tahansa tiedostot, jotka olet luonut tai joita olet muokannut, joihin et ole ajanut `git add`-komentoa editoinnin jälkeen - ei mene pysyvään muutokseen. Ne pysyvät muokattuina tiedostoina levylläsi. Tässä tapauksessa oletamme, että viime kerran kun ajoit `git status`-komennon, näit, jotta kaikki oli lavastattu, joten olet valmis tekemään pysyvän muutoksen. Helpoin tapa pysyvän muutoksen tekoon on kirjoittaa `git commit`:

```
$ git commit
```

Tämän suorittaminen aukaisee editorisi. (Tämä on asetettu komentorivisi `$EDITOR` ympäristömuuttujalla - yleensä `vim` tai `emacs`, kuitenkin voit konfiguroida sen käyttämään mitä tahansa haluat, käyttäen `git config --global core.editor`-komentoa, kuten Kappeleessa 1 näit).

Editori näyttää seuraavanlaisen tekstin (tämä esimerkki on Vimistä):

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   README
#       modified:  benchmarks.rb
~
~
~
".git/COMMIT_EDITMSG" 10L, 283C
```

Voit nähdä, että oletuksena pysyvän muutoksen viesti sisältää viimeisimmän `git status`-komennon syötteen kommentoituna ja yhden tyhjän rivin ylhäällä. Voit poistaa nämä kommentit ja kirjoittaa pysyvän muutoksen viestisi, tai voit jättää kommentit viestiin auttamaan sinua muistamaan mihin olet pysyvää muutosta tekemässä. (Saadaksesi vieläkin tarkemman muistutukseen muutoksesi, voit antaa `-v`-option `git commit`-komennolle. Tämä option laittaa myös diff muutostulosten editoriin, jotta näet tarkalleen mitä teit.) Kun poistut editorista, Git luo pysyvän muutoksesi viestilläsi (kommentit ja diff pois lukien).

Vaihtoehtoisesti, voit kirjoittaa pysyvän muutoksen viestin suoraan `commit`-komentolla antamalla sen `-m` lipun jälkeen, näin:

```
$ git commit -m "Story 182: Fix benchmarks for speed"
[master]: created 463dc4f: "Fix benchmarks for speed"
 2 files changed, 3 insertions(+), 0 deletions(-)
 create mode 100644 README
```

Nyt olet luonut ensimmäisen pysyvän muutoksen! Voit nähdä, että pysyvä muutos on antanut sinulle tulosten itsestään: kertoen mihin haaraan teit pysyvän muutoksen (master), mikä SHA-1 tarkistussumma pysyvällä muutoksella on (463dc4f), kuinka monta tiedostoa muutettiin ja tilastoja pysyvän muutoksen rivien lisäyksistä ja poistoista.

Muista, että pysyvä muutos tallentaa tilannekuvan lavastusalueestasi. Kaikki mitä et lavastanut on yhä istumassa projektissasi muokattuna; voit tehdä toisen pysyvän muutoksen lisätäksesi ne historiaasi. Joka kerta kun teet pysyvän muutoksen, olet tallentamassa tilannekuvaa projektistasi. Tilannekuvaa johon voit palata tai jota voit vertailla myöhemmin.

## 2.2.7 Lavastusalueen ohittaminen

Vaikka se voi olla uskomattoman hyödyllinen pysyvien muutoksien tekoon tarkalleen niin kuin ne haluat, on lavastusalue joskus pikkaisen liian monimutkainen, kuin mitä työkulussasi tarvitsisit. Jos haluat ohittaa lavastusalueen, Git tarjoaa siihen helpon oikoreitin. Antamalla `-a` option `git commit`-komentoon, asettaa Gitin automaattisesti lavastamaan jokaisen jo jäljitetyn tiedoston ennen pysyvää muutosta, antaen sinun ohittaa `git add` osan:

```
$ git status
# On branch master
#
# Changed but not updated:
#
#   modified:   benchmarks.rb
#
$ git commit -a -m 'added new benchmarks'
[master 83e38c7] added new benchmarks
1 files changed, 5 insertions(+), 0 deletions(-)
```

Huomaa miten sinun ei tarvitse ajaa `git add`-komentoa `benchmarks.rb` tiedostolle tässä tapauksessa pysyvää muutosta tehdessäsi.

## 2.2.8 Tiedostojen poistaminen

Poistaaksesi tiedoston Gitistä, täytyy sinun poistaa se sinun jäljitetyistä tiedostoistasi (tarkemmin sanoen, poistaa se lavastusalueeltasi) ja sitten tehdä pysyvä muutos. Komento `git rm` tekee tämän myös ja myös poistaa tiedoston työskentely hakemistostasi, joten et näe sitä enää jäljittämättömänä tiedostona.

Jos yksinkertaisesti poistat tiedoston työskentely hakemistostasi, näkyy se "Changed but not updated" otsikon alla (se on, *lavastamaton*) `git status` tulosteessasi:

```
$ rm grit.gemspec
$ git status
# On branch master
#
# Changed but not updated:
#   (use "git add/rm <file>..." to update what will be committed)
#
#       deleted:   grit.gemspec
#
```

Tämän jälkeen, jos ajat `git rm`-komennon, se lavastaa tiedostot poistoon:

```
$ git rm grit.gemspec
```

```
rm 'grit.gemspec'  
$ git status  
# On branch master  
#  
# Changes to be committed:  
#   (use "git reset HEAD <file>..." to unstage)  
#  
#       deleted:    grit.gemspec  
#
```

Seuraavan kerran kun teet pysyvän muutoksen, tiedosto katoaa ja sitä ei jäljitetä enää. Jos muokkasit tiedostoa ja lisäsit sen jo indeksiin, täytyy sinun pakottaa poisto -f optiolla. Tämä on turvallisuus ominaisuus, joka estää vahingossa tapahtuvan datan poistamisen, datan, jota ei ole vielä tallennettu tilanekuvaksi ja jota ei voida palauttaa Gitistä.

Toinen hyödyllinen asia, jonka saatat haluta tehdä, on tiedoston pitäminen työskentely puussa, mutta samalla sen poistaminen lavastusalueelta. Toisin sanoen, voit haluta pitää tiedoston kovalevylläsi, mutta et halua, että Git jäljittelee sitä enää. Tämä on erityisesti hyödyllinen, jos unohdit lisätä jotain `.gitignore`-tiedostoosi ja vahingossa asetit sellaisen Gittin, kuten suuri logi tiedosto tai joukko `.a`-muotoon käännettyjä tiedostoja. Tehdäksesi tämän, käytä `--cached` optiota:

```
$ git rm --cached readme.txt
```

Voit antaa tiedostoja, hakemistoja, tai tiedosto-keräys malleja `git rm`-komennolle. Tämä tarkoittaa, että voit tehdä asioita kuten:

```
$ git rm log/\*.log
```

Huomaa kenoviiva (`\`) `*`-merkin edessä. Tämä on tarpeellinen, koska Git tekee oman tiedostonimi laajennuksensa komentorivisi tiedostonimilaajennuksen lisänä. Tämä komento poistaa kaikki tiedostot, joilla on `.log` liite, `log/`-hakemistosta. Tai, voit tehdä näin:

```
$ git rm \*~
```

Tämä komento poistaa kaikki tiedostot, jotka loppuvat `~`-merkkiin.

### 2.2.9 Moving Files

Unlike many other VCS systems, Git doesn't explicitly track file movement. If you rename a file in Git, no metadata is stored in Git that tells it you renamed

the file. However, Git is pretty smart about figuring that out after the fact — we'll deal with detecting file movement a bit later.

Thus it's a bit confusing that Git has a `mv` command. If you want to rename a file in Git, you can run something like

```
$ git mv file_from file_to
```

and it works fine. In fact, if you run something like this and look at the status, you'll see that Git considers it a renamed file:

```
$ git mv README.txt README
$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       renamed:    README.txt -> README
#
```

However, this is equivalent to running something like this:

```
$ mv README.txt README
$ git rm README.txt
$ git add README
```

Git figures out that it's a rename implicitly, so it doesn't matter if you rename a file that way or with the `mv` command. The only real difference is that `mv` is one command instead of three — it's a convenience function. More important, you can use any tool you like to rename a file, and address the `add/rm` later, before you commit.

## 2.3 Viewing the Commit History

After you have created several commits, or if you have cloned a repository with an existing commit history, you'll probably want to look back to see what has happened. The most basic and powerful tool to do this is the `git log` command.

These examples use a very simple project called `simplegit` that I often use for demonstrations. To get the project, run

```
git clone git://github.com/schacon/simplegit-progit.git
```

When you run `git log` in this project, you should get output that looks something like this:

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test code

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit
```

By default, with no arguments, `git log` lists the commits made in that repository in reverse chronological order. That is, the most recent commits show up first. As you can see, this command lists each commit with its SHA-1 checksum, the author's name and e-mail, the date written, and the commit message.

A huge number and variety of options to the `git log` command are available to show you exactly what you're looking for. Here, we'll show you some of the most-used options.

One of the more helpful options is `-p`, which shows the diff introduced in each commit. You can also use `-2`, which limits the output to only the last two entries:

```
$ git log -p -2
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

diff --git a/Rakefile b/Rakefile
index a874b73..8f94139 100644
--- a/Rakefile
+++ b/Rakefile
@@ -5,7 +5,7 @@ require 'rake/gempackagetask'
  spec = Gem::Specification.new do |s|
-   s.version   = "0.1.0"
+   s.version   = "0.1.1"
    s.author    = "Scott Chacon"

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
```

```

Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test code

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index a0a60ae..47c6340 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -18,8 +18,3 @@ class SimpleGit
     end

   end
-
-  -if $0 == __FILE__
-    - git = SimpleGit.new
-    - puts git.show
-  -end
\ No newline at end of file

```

This option displays the same information but with a diff directly following each entry. This is very helpful for code review or to quickly browse what happened during a series of commits that a collaborator has added. You can also use a series of summarizing options with `git log`. For example, if you want to see some abbreviated stats for each commit, you can use the `--stat` option:

```

$ git log --stat
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

Rakefile |    2 +-
1 files changed, 1 insertions(+), 1 deletions(-)

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test code

lib/simplegit.rb |    5 -----
1 files changed, 0 insertions(+), 5 deletions(-)

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit

README          |    6 ++++++
Rakefile        |   23 ++++++
lib/simplegit.rb |   25 ++++++

```

```
3 files changed, 54 insertions(+), 0 deletions(-)
```

As you can see, the `--stat` option prints below each commit entry a list of modified files, how many files were changed, and how many lines in those files were added and removed. It also puts a summary of the information at the end. Another really useful option is `--pretty`. This option changes the log output to formats other than the default. A few prebuilt options are available for you to use. The `oneline` option prints each commit on a single line, which is useful if you're looking at a lot of commits. In addition, the `short`, `full`, and `fuller` options show the output in roughly the same format but with less or more information, respectively:

```
$ git log --pretty=oneline
ca82a6dff817ec66f44342007202690a93763949 changed the version number
085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 removed unnecessary test code
a11bef06a3f659402fe7563abf99ad00de2209e6 first commit
```

The most interesting option is `format`, which allows you to specify your own log output format. This is especially useful when you're generating output for machine parsing — because you specify the format explicitly, you know it won't change with updates to Git:

```
$ git log --pretty=format:"%h - %an, %ar : %s"
ca82a6d - Scott Chacon, 11 months ago : changed the version number
085bb3b - Scott Chacon, 11 months ago : removed unnecessary test code
a11bef0 - Scott Chacon, 11 months ago : first commit
```

Table 2.1 lists some of the more useful options that `format` takes.

Option	Description of Output
<code>%H</code>	Commit hash
<code>%h</code>	Abbreviated commit hash
<code>%T</code>	Tree hash
<code>%t</code>	Abbreviated tree hash
<code>%P</code>	Parent hashes
<code>%p</code>	Abbreviated parent hashes
<code>%an</code>	Author name
<code>%ae</code>	Author e-mail
<code>%ad</code>	Author date (format respects the <code>--date=</code> option)
<code>%ar</code>	Author date, relative
<code>%cn</code>	Committer name
<code>%ce</code>	Committer email
<code>%cd</code>	Committer date
<code>%cr</code>	Committer date, relative
<code>%s</code>	Subject

You may be wondering what the difference is between *author* and *committer*. The author is the person who originally wrote the work, whereas the committer is the person who last applied the work. So, if you send in a patch to a project and one of the core members applies the patch, both of you get credit — you as the author and the core member as the committer. We'll cover this distinction a bit more in Chapter 5.

The oneline and format options are particularly useful with another log option called `--graph`. This option adds a nice little ASCII graph showing your branch and merge history, which we can see our copy of the Grit project repository:

```
$ git log --pretty=format:"%h %s" --graph
* 2d3acf9 ignore errors from SIGCHLD on trap
* 5e3ee11 Merge branch 'master' of git://github.com/dustin/grit
|\
| * 420eac9 Added a method for getting the current branch.
* | 30e367c timeout code and tests
* | 5a09431 add timeout protection to grit
* | e1193f8 support for heads with slashes in them
|/
* d6016bc require time for xmlschema
* 11d191e Merge branch 'defunkt' into local
```

Those are only some simple output-formatting options to `git log` — there are many more. Table 2.2 lists the options we've covered so far and some other common formatting options that may be useful, along with how they change the output of the log command.

#### Option Description

```
-p Show the patch introduced with each commit.
--stat Show statistics for files modified in each commit.
--shortstat Display only the changed/insertions/deletions line from the --stat command.
--name-only Show the list of files modified after the commit information.
--name-status Show the list of files affected with added/modified/deleted information as well.
--abbrev-commit Show only the first few characters of the SHA-1 checksum instead of all 40.
--relative-date Display the date in a relative format (for example, "2 weeks ago") instead of using the full date.
--graph Display an ASCII graph of the branch and merge history beside the log output.
--pretty Show commits in an alternate format. Options include oneline, short, full, fuller, and format (where
```

### 2.3.1 Limiting Log Output

In addition to output-formatting options, `git log` takes a number of useful limiting options — that is, options that let you show only a subset of commits. You've seen one such option already — the `-2` option, which show only the last two commits. In fact, you can do `-<n>`, where `n` is any integer to show the last

n commits. In reality, you're unlikely to use that often, because Git by default pipes all output through a pager so you see only one page of log output at a time.

However, the time-limiting options such as `--since` and `--until` are very useful. For example, this command gets the list of commits made in the last two weeks:

```
$ git log --since=2.weeks
```

This command works with lots of formats — you can specify a specific date (“2008-01-15”) or a relative date such as “2 years 1 day 3 minutes ago”.

You can also filter the list to commits that match some search criteria. The `--author` option allows you to filter on a specific author, and the `--grep` option lets you search for keywords in the commit messages. (Note that if you want to specify both author and grep options, you have to add `--all-match` or the command will match commits with either.)

The last really useful option to pass to `git log` as a filter is a path. If you specify a directory or file name, you can limit the log output to commits that introduced a change to those files. This is always the last option and is generally preceded by double dashes (`--`) to separate the paths from the options.

In Table 2.3 we'll list these and a few other common options for your reference.

#### Option Description

```
-(n) Show only the last n commits
--since, --after Limit the commits to those made after the specified date.
--until, --before Limit the commits to those made before the specified date.
--author Only show commits in which the author entry matches the specified string.
--committer Only show commits in which the committer entry matches the specified string.
```

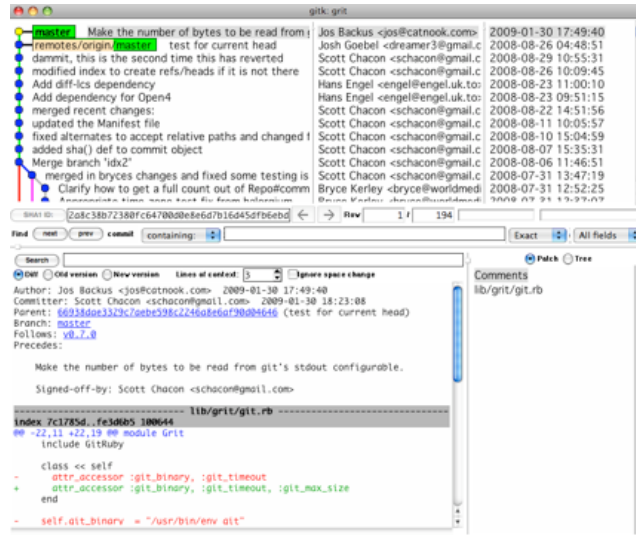
For example, if you want to see which commits modifying test files in the Git source code history were committed by Junio Hamano and were not merges in the month of October 2008, you can run something like this:

```
$ git log --pretty="%h - %s" --author=gitster --since="2008-10-01" \
  --before="2008-11-01" --no-merges -- t/
5610e3b - Fix testcase failure when extended attribute
acd3b9e - Enhance hold_lock_file_for_{update,append}()
f563754 - demonstrate breakage of detached checkout wi
d1a43f2 - reset --hard/read-tree --reset -u: remove un
51a94af - Fix "checkout --track -b newbranch" on detac
b0ad11e - pull: allow "git pull origin $something:$cur
```

Of the nearly 20,000 commits in the Git source code history, this command shows the 6 that match those criteria.

## 2.3.2 Using a GUI to Visualize History

If you like to use a more graphical tool to visualize your commit history, you may want to take a look at a Tcl/Tk program called `gitk` that is distributed with Git. `Gitk` is basically a visual `git log` tool, and it accepts nearly all the filtering options that `git log` does. If you type `gitk` on the command line in your project, you should see something like Figure 2.2.



**Figure 2.2:** *The `gitk` history visualizer.*

You can see the commit history in the top half of the window along with a nice ancestry graph. The diff viewer in the bottom half of the window shows you the changes introduced at any commit you click.

## 2.4 Undoing Things

At any stage, you may want to undo something. Here, we'll review a few basic tools for undoing changes that you've made. Be careful, because you can't always undo some of these undos. This is one of the few areas in Git where you may lose some work if you do it wrong.

### 2.4.1 Changing Your Last Commit

One of the common undos takes place when you commit too early and possibly forget to add some files, or you mess up your commit message. If you want to try that commit again, you can run commit with the `--amend` option:

```
$ git commit --amend
```

This command takes your staging area and uses it for the commit. If you've have made no changes since your last commit (for instance, you run this com-

mand immediately after your previous commit), then your snapshot will look exactly the same and all you'll change is your commit message.

The same commit-message editor fires up, but it already contains the message of your previous commit. You can edit the message the same as always, but it overwrites your previous commit.

As an example, if you commit and then realize you forgot to stage the changes in a file you wanted to add to this commit, you can do something like this:

```
$ git commit -m 'initial commit'
$ git add forgotten_file
$ git commit --amend
```

All three of these commands end up with a single commit — the second commit replaces the results of the first.

## 2.4.2 Unstaging a Staged File

The next two sections demonstrate how to wrangle your staging area and working directory changes. The nice part is that the command you use to determine the state of those two areas also reminds you how to undo changes to them. For example, let's say you've changed two files and want to commit them as two separate changes, but you accidentally type `git add *` and stage them both. How can you unstage one of the two? The `git status` command reminds you:

```
$ git add .
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   README.txt
#       modified:   benchmarks.rb
#
```

Right below the “Changes to be committed” text, it says use `git reset HEAD <file>...` to unstage. So, let's use that advice to unstage the `benchmarks.rb` file:

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

```
#
#   modified:   README.txt
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#   modified:   benchmarks.rb
#
```

The command is a bit strange, but it works. The benchmarks.rb file is modified but once again unstaged.

### 2.4.3 Unmodifying a Modified File

What if you realize that you don't want to keep your changes to the benchmarks.rb file? How can you easily unmodify it — revert it back to what it looked like when you last committed (or initially cloned, or however you got it into your working directory)? Luckily, `git status` tells you how to do that, too. In the last example output, the unstaged area looks like this:

```
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#   modified:   benchmarks.rb
#
```

It tells you pretty explicitly how to discard the changes you've made (at least, the newer versions of Git, 1.6.1 and later, do this — if you have an older version, we highly recommend upgrading it to get some of these nicer usability features). Let's do what it says:

```
$ git checkout -- benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   modified:   README.txt
#
```

You can see that the changes have been reverted. You should also realize that this is a dangerous command: any changes you made to that file are gone — you just copied another file over it. Don't ever use this command unless you absolutely know that you don't want the file. If you just need to get it out of

the way, we'll go over stashing and branching in the next chapter; these are generally better ways to go.

Remember, anything that is committed in Git can almost always be recovered. Even commits that were on branches that were deleted or commits that were overwritten with an `--amend` commit can be recovered (see Chapter 9 for data recovery). However, anything you lose that was never committed is likely never to be seen again.

## 2.5 Working with Remotes

To be able to collaborate on any Git project, you need to know how to manage your remote repositories. Remote repositories are versions of your project that are hosted on the Internet or network somewhere. You can have several of them, each of which generally is either read-only or read/write for you. Collaborating with others involves managing these remote repositories and pushing and pulling data to and from them when you need to share work. Managing remote repositories includes knowing how to add remote repositories, remove remotes that are no longer valid, manage various remote branches and define them as being tracked or not, and more. In this section, we'll cover these remote-management skills.

### 2.5.1 Showing Your Remotes

To see which remote servers you have configured, you can run the `git remote` command. It lists the shortnames of each remote handle you've specified. If you've cloned your repository, you should at least see `origin` — that is the default name Git gives to the server you cloned from:

```
$ git clone git://github.com/schacon/ticgit.git
Initialized empty Git repository in /private/tmp/ticgit/.git/
remote: Counting objects: 595, done.
remote: Compressing objects: 100% (269/269), done.
remote: Total 595 (delta 255), reused 589 (delta 253)
Receiving objects: 100% (595/595), 73.31 KiB | 1 KiB/s, done.
Resolving deltas: 100% (255/255), done.
$ cd ticgit
$ git remote
origin
```

You can also specify `-v`, which shows you the URL that Git has stored for the shortname to be expanded to:

```
$ git remote -v
origin git://github.com/schacon/ticgit.git
```

If you have more than one remote, the command lists them all. For example, my Grit repository looks something like this.

```
$ cd grit
$ git remote -v
bakkdoor  git://github.com/bakkdoor/grit.git
cho45     git://github.com/cho45/grit.git
defunkt   git://github.com/defunkt/grit.git
koke      git://github.com/koke/grit.git
origin    git@github.com:mojombo/grit.git
```

This means we can pull contributions from any of these users pretty easily. But notice that only the origin remote is an SSH URL, so it's the only one I can push to (we'll cover why this is in Chapter 4).

## 2.5.2 Adding Remote Repositories

I've mentioned and given some demonstrations of adding remote repositories in previous sections, but here is how to do it explicitly. To add a new remote Git repository as a shortname you can reference easily, run `git remote add [shortname] [url]`:

```
$ git remote
origin
$ git remote add pb git://github.com/paulboone/ticgit.git
$ git remote -v
origin git://github.com/schacon/ticgit.git
pb git://github.com/paulboone/ticgit.git
```

Now you can use the string `pb` on the command line in lieu of the whole URL. For example, if you want to fetch all the information that Paul has but that you don't yet have in your repository, you can run `git fetch pb`:

```
$ git fetch pb
remote: Counting objects: 58, done.
remote: Compressing objects: 100% (41/41), done.
remote: Total 44 (delta 24), reused 1 (delta 0)
Unpacking objects: 100% (44/44), done.
From git://github.com/paulboone/ticgit
 * [new branch]      master      -> pb/master
 * [new branch]      ticgit      -> pb/ticgit
```

Paul's master branch is accessible locally as `pb/master` — you can merge it into one of your branches, or you can check out a local branch at that point if you want to inspect it.

### 2.5.3 Fetching and Pulling from Your Remotes

As you just saw, to get data from your remote projects, you can run:

```
$ git fetch [remote-name]
```

The command goes out to that remote project and pulls down all the data from that remote project that you don't have yet. After you do this, you should have references to all the branches from that remote, which you can merge in or inspect at any time. (We'll go over what branches are and how to use them in much more detail in Chapter 3.)

If you clone a repository, the command automatically adds that remote repository under the name `origin`. So, `git fetch origin` fetches any new work that has been pushed to that server since you cloned (or last fetched from) it. It's important to note that the `fetch` command pulls the data to your local repository — it doesn't automatically merge it with any of your work or modify what you're currently working on. You have to merge it manually into your work when you're ready.

If you have a branch set up to track a remote branch (see the next section and Chapter 3 for more information), you can use the `git pull` command to automatically fetch and then merge a remote branch into your current branch. This may be an easier or more comfortable workflow for you; and by default, the `git clone` command automatically sets up your local master branch to track the remote master branch on the server you cloned from (assuming the remote has a master branch). Running `git pull` generally fetches data from the server you originally cloned from and automatically tries to merge it into the code you're currently working on.

### 2.5.4 Pushing to Your Remotes

When you have your project at a point that you want to share, you have to push it upstream. The command for this is simple: `git push [remote-name] [branch-name]`. If you want to push your master branch to your `origin` server (again, cloning generally sets up both of those names for you automatically), then you can run this to push your work back up to the server:

```
$ git push origin master
```

This command works only if you cloned from a server to which you have write access and if nobody has pushed in the meantime. If you and someone else clone at the same time and they push upstream and then you push upstream, your push will rightly be rejected. You'll have to pull down their work first and incorporate it into yours before you'll be allowed to push. See Chapter 3 for more detailed information on how to push to remote servers.

## 2.5.5 Inspecting a Remote

If you want to see more information about a particular remote, you can use the `git remote show [remote-name]` command. If you run this command with a particular shortname, such as `origin`, you get something like this:

```
$ git remote show origin
* remote origin
  URL: git://github.com/schacon/ticgit.git
  Remote branch merged with 'git pull' while on branch master
    master
  Tracked remote branches
    master
    ticgit
```

It lists the URL for the remote repository as well as the tracking branch information. The command helpfully tells you that if you're on the master branch and you run `git pull`, it will automatically merge in the master branch on the remote after it fetches all the remote references. It also lists all the remote references it has pulled down.

That is a simple example you're likely to encounter. When you're using Git more heavily, however, you may see much more information from `git remote show`:

```
$ git remote show origin
* remote origin
  URL: git@github.com:defunkt/github.git
  Remote branch merged with 'git pull' while on branch issues
    issues
  Remote branch merged with 'git pull' while on branch master
    master
  New remote branches (next fetch will store in remotes/origin)
    caching
  Stale tracking branches (use 'git remote prune')
    libwalker
    walker2
  Tracked remote branches
    acl
    apiv2
    dashboard2
    issues
    master
    postgres
  Local branch pushed with 'git push'
    master:master
```

This command shows which branch is automatically pushed when you run `git push` on certain branches. It also shows you which remote branches on the server you don't yet have, which remote branches you have that have been

removed from the server, and multiple branches that are automatically merged when you run `git pull`.

### 2.5.6 Removing and Renaming Remotes

If you want to rename a reference, in newer versions of Git you can run `git remote rename` to change a remote's shortname. For instance, if you want to rename `pb` to `paul`, you can do so with `git remote rename`:

```
$ git remote rename pb paul
$ git remote
origin
paul
```

It's worth mentioning that this changes your remote branch names, too. What used to be referenced at `pb/master` is now at `paul/master`.

If you want to remove a reference for some reason — you've moved the server or are no longer using a particular mirror, or perhaps a contributor isn't contributing anymore — you can use `git remote rm`:

```
$ git remote rm paul
$ git remote
origin
```

## 2.6 Tagging

Like most VCSs, Git has the ability to tag specific points in history as being important. Generally, people use this functionality to mark release points (v1.0, and so on). In this section, you'll learn how to list the available tags, how to create new tags, and what the different types of tags are.

### 2.6.1 Listing Your Tags

Listing the available tags in Git is straightforward. Just type `git tag`:

```
$ git tag
v0.1
v1.3
```

This command lists the tags in alphabetical order; the order in which they appear has no real importance.

You can also search for tags with a particular pattern. The Git source repo, for instance, contains more than 240 tags. If you're only interested in looking at the 1.4.2 series, you can run this:

```
$ git tag -l 'v1.4.2.*'  
v1.4.2.1  
v1.4.2.2  
v1.4.2.3  
v1.4.2.4
```

## 2.6.2 Creating Tags

Git uses two main types of tags: lightweight and annotated. A lightweight tag is very much like a branch that doesn't change — it's just a pointer to a specific commit. Annotated tags, however, are stored as full objects in the Git database. They're checksummed; contain the tagger name, e-mail, and date; have a tagging message; and can be signed and verified with GNU Privacy Guard (GPG). It's generally recommended that you create annotated tags so you can have all this information; but if you want a temporary tag or for some reason don't want to keep the other information, lightweight tags are available too.

## 2.6.3 Annotated Tags

Creating an annotated tag in Git is simple. The easiest way is to specify `-a` when you run the tag command:

```
$ git tag -a v1.4 -m 'my version 1.4'  
$ git tag  
v0.1  
v1.3  
v1.4
```

The `-m` specifies a tagging message, which is stored with the tag. If you don't specify a message for an annotated tag, Git launches your editor so you can type it in.

You can see the tag data along with the commit that was tagged by using the `git show` command:

```
$ git show v1.4  
tag v1.4  
Tagger: Scott Chacon <schacon@gee-mail.com>  
Date: Mon Feb 9 14:45:11 2009 -0800  
  
my version 1.4  
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
```

```
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date: Sun Feb 8 19:02:46 2009 -0800
```

```
Merge branch 'experiment'
```

That shows the tagger information, the date the commit was tagged, and the annotation message before showing the commit information.

### 2.6.4 Signed Tags

You can also sign your tags with GPG, assuming you have a private key. All you have to do is use `-s` instead of `-a`:

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'
You need a passphrase to unlock the secret key for
user: "Scott Chacon <schacon@gee-mail.com>"
1024-bit DSA key, ID F721C45A, created 2009-02-09
```

If you run `git show` on that tag, you can see your GPG signature attached to it:

```
$ git show v1.5
tag v1.5
Tagger: Scott Chacon <schacon@gee-mail.com>
Date: Mon Feb 9 15:22:20 2009 -0800

my signed 1.5 tag
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.8 (Darwin)

iEYEA BECAAYFAkmQurIACgkQ0N3DxfchxFr5cACeIMN+ZxLKggJQf0QYiQBwgySN
Ki0An2JeAVUCAiJ70x6ZEtK+NvZAJ82/
=WryJ
-----END PGP SIGNATURE-----
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date: Sun Feb 8 19:02:46 2009 -0800

Merge branch 'experiment'
```

A bit later, you'll learn how to verify signed tags.

### 2.6.5 Lightweight Tags

Another way to tag commits is with a lightweight tag. This is basically the commit checksum stored in a file — no other information is kept. To create a

lightweight tag, don't supply the `-a`, `-s`, or `-m` option:

```
$ git tag v1.4-lw
$ git tag
v0.1
v1.3
v1.4
v1.4-lw
v1.5
```

This time, if you run `git show` on the tag, you don't see the extra tag information. The command just shows the commit:

```
$ git show v1.4-lw
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date: Sun Feb 8 19:02:46 2009 -0800

Merge branch 'experiment'
```

## 2.6.6 Verifying Tags

To verify a signed tag, you use `git tag -v [tag-name]`. This command uses GPG to verify the signature. You need the signer's public key in your keyring for this to work properly:

```
$ git tag -v v1.4.2.1
object 883653babd8ee7ea23e6a5c392bb739348b1eb61
type commit
tag v1.4.2.1
tagger Junio C Hamano <junkio@cox.net> 1158138501 -0700

GIT 1.4.2.1

Minor fixes since 1.4.2, including git-mv and git-http with alternates.
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Good signature from "Junio C Hamano <junkio@cox.net>"
gpg: aka "[jpeg image of size 1513]"
Primary key fingerprint: 3565 2A26 2040 E066 C9A7 4A7D C0C6 D9A4 F311 9B9A
```

If you don't have the signer's public key, you get something like this instead:

```
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Can't check signature: public key not found
error: could not verify the tag 'v1.4.2.1'
```

## 2.6.7 Tagging Later

You can also tag commits after you've moved past them. Suppose your commit history looks like this:

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support
0d52aaab4479697da7686c15f77a3d64d9165190 one more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc added a commit function
4682c3261057305bdd616e23b64b0857d832627b added a todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a started write support
9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme
```

Now, suppose you forgot to tag the project at v1.2, which was at the “updated rakefile” commit. You can add it after the fact. To tag that commit, you specify the commit checksum (or part of it) at the end of the command:

```
$ git tag -a v1.2 9fceb02
```

You can see that you've tagged the commit:

```
$ git tag
v0.1
v1.2
v1.3
v1.4
v1.4-lw
v1.5

$ git show v1.2
tag v1.2
Tagger: Scott Chacon <schacon@gee-mail.com>
Date: Mon Feb 9 15:32:16 2009 -0800

version 1.2
commit 9fceb02d0ae598e95dc970b74767f19372d61af8
Author: Magnus Chacon <mchacon@gee-mail.com>
Date: Sun Apr 27 20:43:35 2008 -0700

    updated rakefile
...
```

## 2.6.8 Sharing Tags

By default, the `git push` command doesn't transfer tags to remote servers. You will have to explicitly push tags to a shared server after you have created them. This process is just like sharing remote branches – you can run `git push origin [tagname]`.

```
$ git push origin v1.5
Counting objects: 50, done.
Compressing objects: 100% (38/38), done.
Writing objects: 100% (44/44), 4.56 KiB, done.
Total 44 (delta 18), reused 8 (delta 1)
To git@github.com:schacon/simplegit.git
* [new tag]          v1.5 -> v1.5
```

If you have a lot of tags that you want to push up at once, you can also use the `--tags` option to the `git push` command. This will transfer all of your tags to the remote server that are not already there.

```
$ git push origin --tags
Counting objects: 50, done.
Compressing objects: 100% (38/38), done.
Writing objects: 100% (44/44), 4.56 KiB, done.
Total 44 (delta 18), reused 8 (delta 1)
To git@github.com:schacon/simplegit.git
* [new tag]          v0.1 -> v0.1
* [new tag]          v1.2 -> v1.2
* [new tag]          v1.4 -> v1.4
* [new tag]          v1.4-lw -> v1.4-lw
* [new tag]          v1.5 -> v1.5
```

Now, when someone else clones or pulls from your repository, they will get all your tags as well.

## 2.7 Tips and Tricks

Before we finish this chapter on basic Git, a few little tips and tricks may make your Git experience a bit simpler, easier, or more familiar. Many people use Git without using any of these tips, and we won't refer to them or assume you've used them later in the book; but you should probably know how to do them.

### 2.7.1 Auto-Completion

If you use the Bash shell, Git comes with a nice auto-completion script you can enable. Download the Git source code, and look in the `contrib/completion`

directory; there should be a file called `git-completion.bash`. Copy this file to your home directory, and add this to your `.bashrc` file:

```
source ~/.git-completion.bash
```

If you want to set up Git to automatically have Bash shell completion for all users, copy this script to the `/opt/local/etc/bash_completion.d` directory on Mac systems or to the `/etc/bash_completion.d/` directory on Linux systems. This is a directory of scripts that Bash will automatically load to provide shell completions.

If you're using Windows with Git Bash, which is the default when installing Git on Windows with `msysGit`, auto-completion should be preconfigured.

Press the Tab key when you're writing a Git command, and it should return a set of suggestions for you to pick from:

```
$ git co<tab><tab>
commit config
```

In this case, typing `git co` and then pressing the Tab key twice suggests `commit` and `config`. Adding `m<tab>` completes `git commit` automatically.

This also works with options, which is probably more useful. For instance, if you're running a `git log` command and can't remember one of the options, you can start typing it and press Tab to see what matches:

```
$ git log --s<tab>
--shortstat --since= --src-prefix= --stat --summary
```

That's a pretty nice trick and may save you some time and documentation reading.

## 2.7.2 Git Aliases

Git doesn't infer your command if you type it in partially. If you don't want to type the entire text of each of the Git commands, you can easily set up an alias for each command using `git config`. Here are a couple of examples you may want to set up:

```
$ git config --global alias.co checkout
$ git config --global alias.br branch
$ git config --global alias.ci commit
$ git config --global alias.st status
```

This means that, for example, instead of typing `git commit`, you just need to type `git ci`. As you go on using Git, you'll probably use other commands frequently as well; in this case, don't hesitate to create new aliases.

This technique can also be very useful in creating commands that you think should exist. For example, to correct the usability problem you encountered with unstaging a file, you can add your own `unstage` alias to Git:

```
$ git config --global alias.unstage 'reset HEAD --'
```

This makes the following two commands equivalent:

```
$ git unstage fileA
$ git reset HEAD fileA
```

This seems a bit clearer. It's also common to add a `last` command, like this:

```
$ git config --global alias.last 'log -1 HEAD'
```

This way, you can see the last commit easily:

```
$ git last
commit 66938dae3329c7aeb598c2246a8e6af90d04646
Author: Josh Goebel <dreamer3@example.com>
Date: Tue Aug 26 19:48:51 2008 +0800

    test for current head

Signed-off-by: Scott Chacon <schacon@example.com>
```

As you can tell, Git simply replaces the new command with whatever you alias it for. However, maybe you want to run an external command, rather than a Git subcommand. In that case, you start the command with a `!` character. This is useful if you write your own tools that work with a Git repository. We can demonstrate by aliasing `git visual` to run `gitk`:

```
$ git config --global alias.visual '!gitk'
```

## 2.8 Summary

At this point, you can do all the basic local Git operations — creating or cloning a repository, making changes, staging and committing those changes,

and viewing the history of all the changes the repository has been through. Next, we'll cover Git's killer feature: its branching model.