Mittuniversitetet

MID SWEDEN UNIVERSITY

Metoder och verktyg i mjukvaruprojekt

Lektion 2 – Versionshantering med Git

Lektionen och den tillhörande inlämningsuppgiften syftar ge studenten en fördjupad förståelse för versionshantering och dess centrala roll för utvecklingsprocessen!

Innehållsförteckning

1 Inledning

I lektion 1 introducerades konceptet av *versionshantering* och dess understödjande värde för utvecklingsprocessen. Det kan kort beskrivas som en slags **trygghetssäkring** där hela system, likväl enskilda filer, kan återställas till tidigare versioner. Dessutom blir det enkelt att jämföra olika versioners olikheter och se såväl *tidpunkt* som *tillvägagångssätt* för bidrag, samt *vem* dess författare är. Med andra ord, *vem* som gjorde *vad* och *när*!

I denna lektion skall vi fördjupa oss i ämnet och introducera verktyget **Git** som är ett modernt versionshanteringssystem. Men innan vi dyker in i detaljerna kring *Git* så skall vi översiktligt redogöra för bakgrunden av problemområdet som motiverat utvecklingen av dessa system.

Programkod lagras normalt som textfiler och förändringar genomförs genom att vi modifierar innehållet i dessa filer. Varje gång en förändring sparas så skrivs det gamla innehållet över och ersätts med det nya. Ingen människa är dock perfekt och misstag kommer att ske, förr eller senare. I dessa fall kan det vara smidigt att kunna gå tillbaka till tidigare versioner av samma fil, för att se *när*/*var*/*hur* samt *vilken* förändring som ligger till grund för problemet. I det fall detta gäller små mängder av information är det fullt möjligt att manuellt genomsöka olika versioner av aktuell fil för att finna felet, men det blir fort betydligt mer komplicerat när förändringar skett i olika delar av ett mycket stort dokument.

Några av de tidigaste hjälpmedlen för att jämföra olika filversioner är Unix-verktygen **diff** och **patch**, som utvecklades under 70-talet och står som förgrund för dagens *versionshanteringssystem*. **Diff** [1] tar två olika versioner av samma fil som *input* och genererar en *output* som beskriver vad som skiljer dem åt. **Patch** [2] kan sedan applicera denna *output* på första versionen av filen för att *rekonstruera* en korrekt version av den andra.

Output från *diff* är ren textinformation som beskriver vilket innehåll som behöver modifieras för att utföra denna transformering, och kan således vara betydligt mindre än filen den behandlar och kan sparas som egen ***patch-fil*** som sänds till andra användare för applicering.

Verktygen *diff* och *patch* effektiviserar tillvägagångssätten för hur filförändringar kan delas mellan deltagare inom utvecklingsprocessen. En förändring av en kodrad i en fil bestående av totalt **`10,000`** rader kod skulle innebära en *overhead* på hela **`9,999/10,000`** om hela filen skall sändas, medan differensen i sig inte utgör mer än **`1/10,000`** av storleken. Mjukvaruprojekt består normalt av mängder med sådana filer och denna ackumulerade *overhead* kan snabbt bli ett problem.

> *Principen bakom diff och patch är densamma som du stöter på vid uppdateringar av dina installerade programvaror, där uppdateringsfilerna ofta är relativt små i förhållande till installationsprogrammen!*

Låt oss se ett exempel på hur utbytet av en filförändring kan utföras. Två utvecklare, *Bengt* och *Kerstin*, samarbetar i ett projekt. *Bengt* stöter på problem som han inte vet hur han skall lösa, men vet att *Kerstin* besitter den nödvändiga kunskapen. *Bengt* sänder sin originalfil till *Kerstin* som genast löser problemet åt honom, men istället för att sända den *jättestora* v2.0-filen till honom så beräknar hon *skillnaden* mellan versionerna och skickar den **output** som genereras som en **patch-fil**. *Bengt* applicerar sedan denna **patch** på sin originalfil (v1.0) som då rekonstruerar den exakta lösning *Kerstin* skapat (v2.0).

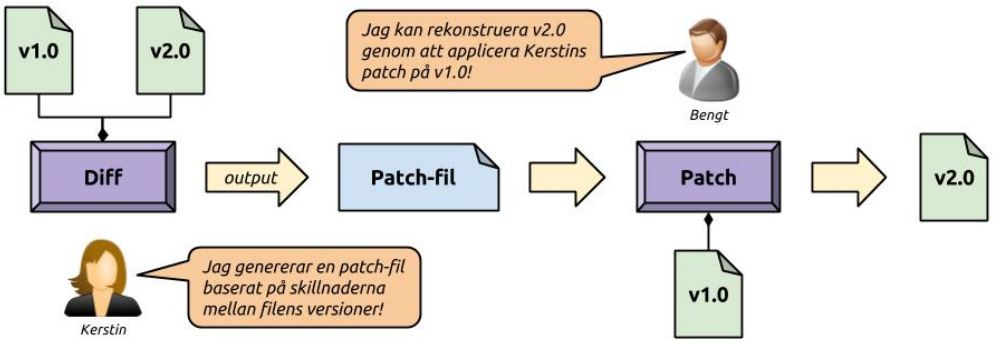


Illustration 1.1: Diff & Patch

De fördelar som *diff* och *patch* för med sig ligger dels i den enkelhet med vilken förändringar kan identifieras och dels den smidighet för hur uppdateringar kan utföras, men de har samtidigt vissa begränsningar. En stor nackdel är att verktygen enbart tar hänsyn till enskilda filer och inte beroenden dem emellan, som exempelvis relationen mellan *definitions*- och *implementationsfiler*. Ofta finns dessutom behovet att kunna se filers utveckling över en längre tidsperiod, men för att göra detta möjligt måste många versioner av filen sparas eftersom *diff* endast tillåter jämförelse mellan två versioner åt gången (den gamla gentemot den nya). Denna versionshantering kan ske manuellt genom att exempelvis *tagga* versioner med modifieringens datum;

```
data.txt
data-2016-04-08.txt
data-2016-06-06.txt
data-2016-08-10.txt
data-2016-11-04.txt
data-2017-01-23.txt
data-2017-09-21.txt
```

En sådan manuell versionshantering följs dock av en mängd svagheter; en ny version av hela filen sparas oavsett omfattningen av modifikationen och namngivningen blir mer komplicerad i det fall filen modifieras vid flera tillfällen samma datum eller då flera personer parallellt skall arbeta med den. Dessutom måste båda versionerna finnas tillgänglig för den som skall utföra jämförelsen, och om detta sker utanför den lokala miljön måste båda filerna flyttas.

Versionshanteringssystem är programvaror som utformats för att lösa dessa problem genom att automatisera versionifieringsprocessen och gruppera relaterade modifieringar i s.k. **changesets**. Det finns tre huvudsakliga förhållningssätt för versionshantering; **lokal**, **centraliserad** samt **distribuerad** och det som skiljer dem åt är deras perspektiv för projektets förvaring av data, filer men framförallt *versionshistorik*, även kallat **repository**, samt hur arbetsflödet kan se ut. I den *lokala versionshanteringen* lagras information om *versionshistoriken* i en lokal databas hos utvecklaren, oftast i relation till projektets övriga filer. Ett rent lokalt förhållningssätt lämpar sig naturligtvis inte för samarbete och är därför inte vanligt annat än som komplement till *distribuerade system*, som vi snart kommer att se.

I den *centraliserade versionshanteringen* (**CVCS**) lagras denna historik istället på en server, från vilken klienter *checkar ut* kopior av den version som skall arbetas med (**working copy**) och *checkar in* nya bidrag som på servern sedan utgör enskilda **revisioner**. Detta upplägg lägger grund för ett linjärt arbetsflöde där allt samarbete synkroniseras över den centrala datakällan. Deltagarna behöver således inte förvara olika versioner av filer på sin arbetsstation utan hämtar detta från servern istället. Detta innebär också att

versionsjämförelser antingen måste utföras direkt av servern eller genom att utvecklaren begär båda versionerna från servern för lokal jämförelse.

Både *lokala* och *centraliserade versionshanteringssystem* lider av samma gundläggande svaghet; när projektets hela historik finns lagrad på ett och samma ställe, så föreligger den övergripande risken att allt går förlorat om dess integritet sviker. **Distribuerade versionshanteringssystem** (**DVCS**) [6] löser detta problem genom att låta alla användare spegla hela repot (förkortning av *repository*) och all dess historik. Detta innebär att varje användare innehar en fullständig **backup** som kan användas som utgångspunkt för att återställa andra användares repos om så skulle behövas.

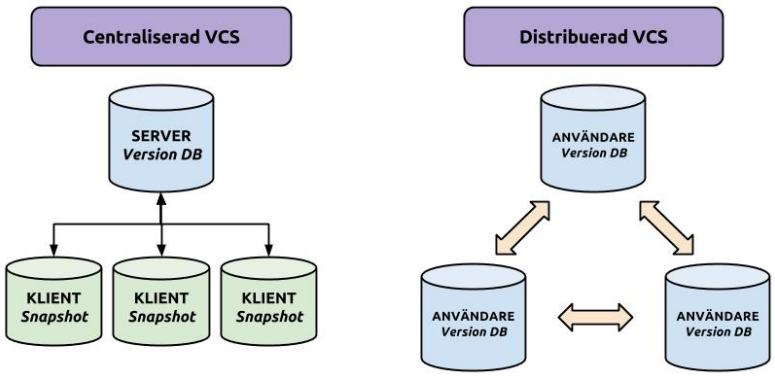


Illustration 1.2: Centraliserad vs. Distribuerad VCS

I kontrast till det *centraliserade systemet* så är relationen **server - klient** inte tydlig då alla användare tillåts agera distributör åt varandra. Denna tillgänglighet kan också lätt orsaka vissa problem, särskilt i stora projekt med hundratals deltagare, där man inte enkelt kan spåra vem som har den senaste versionen. För att råda bot på detta är det vanligt att man i *distribuerade* arbetsflöden utgår från ett auktoritärt repository (*origin*/*release*) för att såväl synkronisera samarbetet som kvalitetsgarantera nya bidrag. Normalt utgörs denna centrala datakälla av en **hosting service** för programkod såsom **GitHub** eller **Bitbucket**, men mer om det senare i lektionen.

> *Jämförelsen blir något missledande när man ställer **centraliserad VCS** gentemot **distribuerad VCS**, då det antyder att distribuerade system inte kan/bör använda en central datakälla. Detta är helt fel! En central källa är inte nödvändig för arbetsflödet, men underlättar samarbetet med många inblandade. En mer passande terminologi vore istället **distribuerad** gentemot **server/klient**!*

Arbetsflödet skiljer sig åt mellan *distribuerade* och *centraliserade* system. I *CVCS* checkas den version som skall arbetas på ut från den centrala datakällan och det fortlöpande arbetet checkas in (*commit*) direkt mot samma källa, vilket kräver ständig nätverksåtkomst. I *DVCS* dras förändringar ned (**pull**) från externa källor i den mån de finns tillgängliga och integreras i det lokala repot. Det löpande arbetet sker sedan direkt mot den lokala versionsdatabasen och eftersom alla tillägg *committas* lokalt finns det inget krav på uppkoppling. Alla *commits* kan sedan **pushas** till ett externt repository i form av **changesets**, alltså gruppering av filförändringar som skall behandlas som sammanhängande helheter.

En annan mycket viktig aspekt när det kommer till *DVCS* är att de, till skillnad från traditionella *CVCS*, erbjuder ***icke-linjära arbetsflöden*** genom sofistikerade förgreningssystem (**branching**). Med detta menas att utvecklingen kan ta olika riktningar genom att projektetversioner kan *checkas ut* till förgrenade kopior som

tillåter experimentering utan att påverka tillståndet hos orginalgrenen och dess tillhörande filer. Den nya grenen utgör då ett eget *repository* helt avskilt från övriga grenar och nya förändringar *committas* till dennes lokala versionsdatabas. Detta möjliggör ett flexibelt arbetsflöde där utvecklingen kan utföras över flera parallella och frånskilda **branches** utan att det riskerar inkräkta på varandra.

Det finns lite olika tillvägagångssätt för hur dessa förgreningar kan skapas, beroende på hur data lagras av versionssystemet. **Git** har sitt eget unika sätt för detta och som ni snart kommer att se så är just denna aspekt ansedd som en av dess främsta styrkor!

Samtidigt som *distribuerade* system kan inge frihet och minskade beroenden, så finns det även potentiella risker med systemen. Som ett exempel kan risken för läckt källkod bli stor då varje klient har en ren förkloning av projektets filer och dess historik. Denna aspekt spelar så klart mindre roll för *open source* projekt, men kan utgöra omfattande säkerhetshot för proprietär programvara eller sådan som behandlar känslig information. En mer reell utmaning för riktigt stora *open source* projekt kan istället vara *hur* nya bidrag skall integreras och *vilka* som har rätt att tillföra dem, då det sällan är en god idé att mängder av deltagare fritt gör detta utan någon form av överseende.
Det är alltså inte helt enkelt att säga att en ren *distribuerad* versionshantering alltid skulle vara mer lämplig än en *centraliserad*, det beror på projektets typ och hur ansvar skall fördelas mellan dess medlemmar. Dock finns det en tydlig trend inom branchen att allt oftare utnyttja distribuerade arbetsflöden, vilket passar bra i utvecklingsteam vars medlemmar ofta finns utspridda i världen. Vi kommer härnäst titta närmare på **Git**, som är ett av de vanligaste *DVCS* som används idag!

2 Översikt av Git

> *Den fritt tillgängliga boken **Pro Git** används under denna kurs. Den finns i två upplagor, varav vi utgår från den andra upplagan, och kan såväl läsas direkt på webbsidan [3] eller genereras till en mängd olika format via bokens källkod som finns tillgängligt på GitHub [4]. Information angående tillvägagångssätt finner du i projektets README.*

Git utvecklades ursprungligen för att fylla ett tomrum som skapats när samarbetet mellan utvecklingsteamet för **Linuxkärnan** och det distribuerade versionshanteringssystemet **BitKeeper** brutits. Linus Torvalds, skapare av Linux, var en starkt bidragande anledning till att *Git* såg dagens ljus 2005. Redan från början var kraven för systemet mycket höga och utifrån fasta målsättningar formulerades visionen för *Git*. Det skulle vara **snabbt**, innebärande att arbetsflödet huvudsakligen sker via lokala operationer som minimerar behovet av tidskrävande kommunikation över nätverk. Vidare skulle det grunda sig på simpel design som stärker **användarvänlighet**, vara fullt **distribuerat** samt **skala** med större projekt på ett effektivt och resurssnålt vis. Dessutom var det viktigt med stöd för stora mängder av parallella grenar, som underlättar **icke-linjär utveckling**.

Under åren har denna vision styrt *Git's* utveckling och är idag ett moget *VCS* som skalar väl med större projekt, erbjuder ett sofistikerat förgreningssystem som uppmuntrar till flexibla arbetsflöden samt hanterar lagring av data på ett sätt som understödjer effektivitet och snabbhet.

Utmärkande för *Git* är just hur det betraktar projektets data. Till skillnad från andra versionssystem så lagras data inte som kalkylerade skillnader (**diff's**), utan *Git* ser istället denna information som en serie bilder (**snapshots**) av systemets tillstånd över tid. Denna konceptuella skillnad innebär att perspektivet flyttas från filbaserade förändringar till mer omfattande strukturella skillnader. Varje gång nya tillägg *committas* representerar detta en *snapshot* av projektets aktuella tillstånd och för att minimera *redundancy* så återlagras inte filer vars tillstånd förblir oförändrade, utan refereras istället till sina tidigare versioner.

Allt som *committas* i Git lagras i databasen via ett **hashvärde** som är en slags sammanfattning av dess innehåll. Aktuell *snapshot* kan sedan identifieras via sitt *hashvärde*, som också används för att beräkna skillnader mellan olika *snapshots*. Detta system är mycket pålitligt och ger Git en hög grad av integritet; ***det som väl har committats är sedan omöjligt att modifiera utan Git's kännedom!***

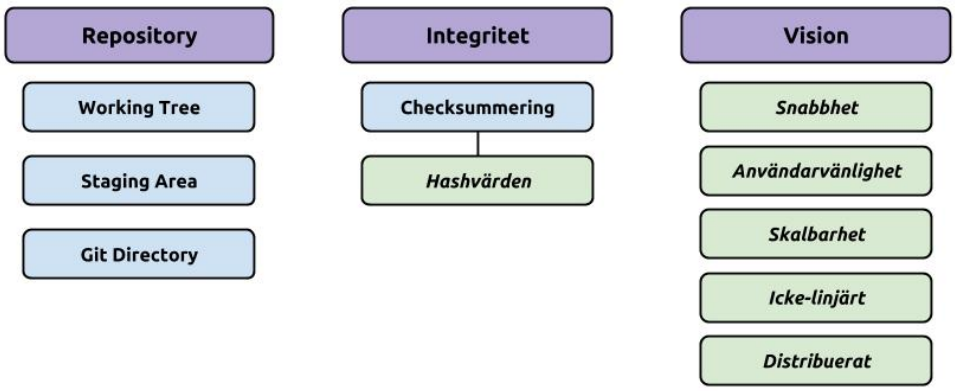


Illustration 2.1: Git översikt

Varje Git *repository* består av de tre huvudsakliga delarna **Git Directory**, **Working Tree** och **Staging Area**. Den centrala punkten för varje Git-projekt är **Git Directory**, som också är den del som kopieras när du klonar en extern källa. Här finner du metadata såsom projektets *versionshistorik*, *filregister* samt *objektdatabas*. **Working Tree** utgör en enskild utcheckning av en specifik projektversion, extraherad från *Git Directory* vari den lagrats som *snapshot*. Alla filer som relaterar till denna version har hämtats från *filregistret* och kan nu *modifieras* som vanligt. **Staging Area** är ett *index* som lagrar information om vilka filer som skall ingå i nästa *commit*. Det representerar således en *snapshot* som under arbetets gång byggs upp i takt med att filförändringar läggs till (**added**).

Projektets filer kan befinna sig i något av tre möjliga tillstånd, beroende på hur de relaterar till ovan beskrivna delar. Den fil som befinner sig i **Git Directory** är *versionshanterad* och anses vara **Committed**, om den är *förändrad* får den tillståndet **Modified** fram till dess att den blir tillagd i **Staging Area** varpå tillståndet ändras till **Staged**.

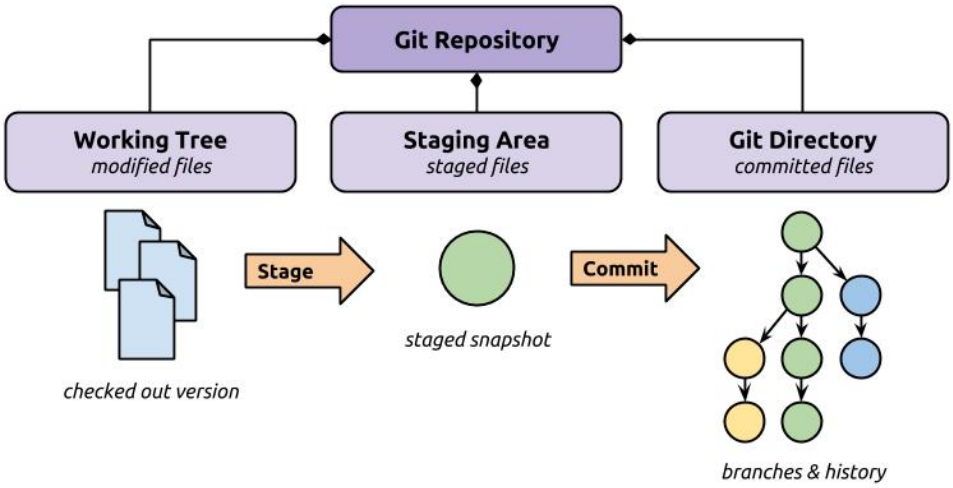


Illustration 2.2: Git repository

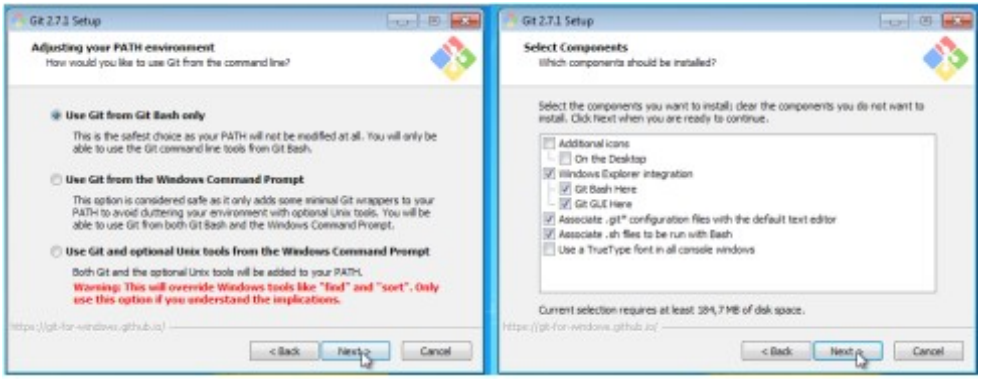
3 Konfigurera Git & Bitbucket

Ladda ned **Git** [1] för ditt operativsystem. Läs därefter **Chapter 1: Getting Started** i kursboken och följ installationsanvisningarna. De grafiska verktygen kommer inte användas under kursen, så det är viktigt att du installerar kommandoradsverktygen.

> *Om du redan har Git installerat så kan du ladda ned senaste versionen genom följande Git-kommando;* `git clone https://github.com/git/git`

3.1.1 Windows användare

Vid installation skall du välja `Git Bash` eftersom den används senare under kursen för att sätta upp **SSH**-nycklar.



`Illustration 3.1: Installation Windows`

Välj även `checkout Windows-style, commit Unix-style line endings`, eftersom du kanske kommer arbeta i grupp med någon som har en Unix-maskin!

3.2 Konfigurera Git

Lägg till användare och epost adress för Git-klienten genom att öppna en terminal (**Git Bash** på Windows) och skriva in följande kommandon. Glöm inte att skriva in **ditt namn** och **din student epost**!!

```
$ git config --global user.name "Erik Ström"
$ git config --global user.email "eristr@student.miun.se"
$ git config --global push.default simple
```

Git lägger till denna information tillsammans med dina *commits*, dels för att identifiera dig inför *file hosting services* såsom **Bitbucket** men också för att separera dina bidrag från övriga deltagares i versionshistoriken. Flaggan `--global` anger att dessa inställningar skall lagras i `~/.gitconfig`, vilket är den övergripande

1 https://git-scm.com/download

inställningsfilen för alla Git-projekt. Utelämning av denna flagga gör inställningarna istället lokala till det aktuella repot.

Sista raden anger hur Git skall hantera kommandot **`push`**, och hur det relaterar till förgreningssystemet. Genom att ange värdet **`simple`** talar vi om att det enbart är aktuell *branch* som skall *pushas*, medan alternativet **`matching`** skulle innebära att alla tillgängliga lokala förgreningar *pushas* till den externa källan. Det finns några olika alternativ för detta, som vardera lämpar sig i varierande grad till olika arbetsflöden. För nybörjare av Git rekommenderas dock att alltid behandla varje *branch* individuellt, så därför använder vi **`simple`** här.

Det finns även andra intressanta inställningar du kan göra. Som exempel kan det vara smidigt att ange vilken texteditor som skall vara standard när *commit*-meddelanden skall skrivas;

```
$ git config --global core.editor "atom –new-window --wait"
```

> *Här används GitHub's fria texteditor Atom som exempel, men du kan ange den editor du själv trivs bäst med. Namnet `'atom'` kan ersättas med exekveringsfilens sökväg och flaggan `--wait` anger att Git skall vänta på **save/close**-händelser. Du kan läsa mer om `core.editor` såväl som andra konfigurationer i kapitel **8.1 - Customizing Git** i kursboken.*

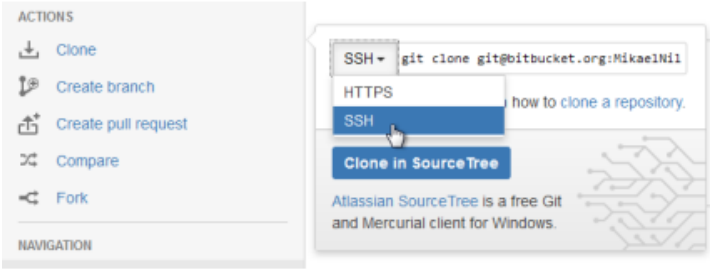
3.3 Bitbucket

I denna kurs kommer vi att använda **Bitbucket** som är en *Free source code hosting service for Git and Mercurial*. Gå till **Bitbucket's hemsida** [2] och skapa ett konto. Det är viktigt att du använder den student epost som du fått från *Mittuniversitetet*, då det ger dig tillgång till en fullversion som du behöver i denna kurs!!

Protokollet som skall användas är **SSH** och innan man kan börja använda det krävs en del konfigurationer. Detaljer för hur man sätter upp **SSH** finns specificerat i guiden **Set up SSH for Git** [5]!

3.3.1 Verifiera och använda SSH-protokollet på Bitbucket

Skapa ett nytt **Repository** och välj **`Clone`** i menyn för **`Actions`** (om det står **`HTTPS`** före adressen så byter du protokoll till **`SSH`**). Kopiera sedan kommandot och kör det i terminalen / *Git Bash*, ett tomt **repository** kommer nu att laddas hem till den katalog du befinner dig i.



`Illustration 3.2: Clone Repository`

2 http://bitbucket.org/

> *Observera att om du gjort rätt så kommer du inte behöva skriva in något lösenord. Om du har flera datorer som du planerar att arbeta med, så måste du repetera stegen ovan för varje dator!*

För att få **SSH**-länken till ett befintligt **repository** på *Bitbucket* så byter du protokoll via menyn bredvid adressen.

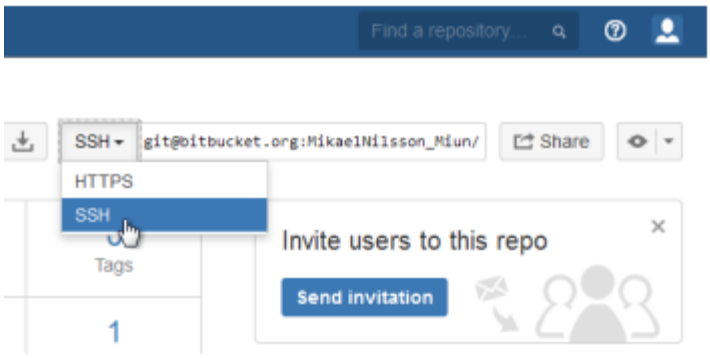


Illustration 3.3: Byta protokoll

> *På Bitbucket används **Markdown** för att på ett enkelt sätt skriva formaterad text. Information om dess syntax finner du på följande källa [8]!*

4 Grunderna i Git

> *Läs kapitel **2 – Git Basics** i kursboken och studera vanligt förekommande kommandon relaterat till initiering av git projekt, tracking / staging av filer, hur ändringar committas samt hur arbete kan ångras beroende på dess stadium i processen!*

All funktionalitet i Git styrs via olika kommandon som vardera har specifika syften, och ofta dessutom tillåter djup detaljstyrning med hjälp av tillhörande *flaggparametrar*. Lämpligast när man börjar lära sig Git är att själv skriva dessa kommandon direkt i kommandotolken/terminalen eftersom den bakomliggande mekaniken sällan framkommer ur de grafiska motsvarigheterna, som dessutom normalt bara stödjer ett urval av de kommandon som Git gör tillgängligt.

För att initiera ett nytt *repository* finns två alternativ. Antingen **importerar** du ett befintligt projekt till Git eller så **klonar** du ett fjärran repo genom att ange dess url (*remote repositories* diskuteras i avsnitt **6 Fjärran Datakällor**);

```
$ git init [name]  // import project
$ git clone [url]  // clone remote repo
```

4.1 Kommandon för arbetsflödet (*Workflow Commands*)

Det huvudsakliga arbetsflödet i Git baseras på sex kommandon som kan grupperas inom två huvudområden; de som relaterar till **Working Tree** och **Staging Area**, samt de som gäller **commit-historiken**.

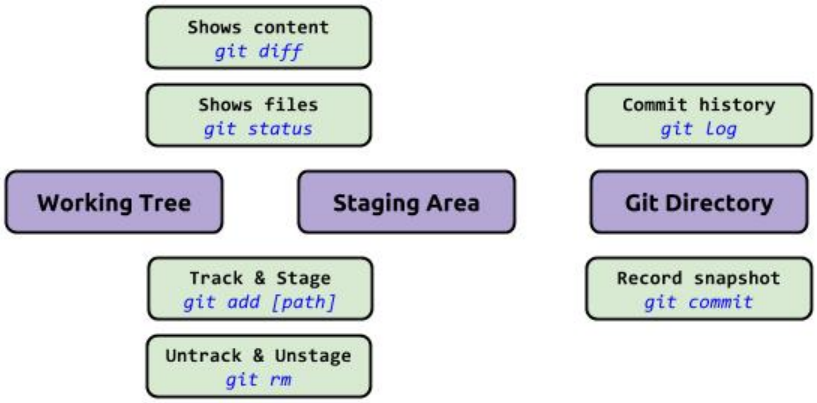


Illustration 4.1: Git kommandon

Innan filer kan versionshanteras måste Git bli instruerad att följa deras tillstånd. Detta kallas för **tracking** av filer och utförs med samma kommando som används för att registrera förändringar i den *snapshot* som skall utgöra nästa *commit;*

```
$ git add [path_to_file]
```

Ett lämpligt sätt att betrakta detta är att du lägger till *händelser*, inte filer, från *Working Tree* till *Staging Area;* att påbörja **tracking** av en fil är en *händelse*, liksom *modifieringar* av filer som redan är **tracked**. På detta sätt har utvecklaren full kontroll över innehållet i varje *commit*, där *logiska snapshots* (**pick-and-choose**) ges företräde över *kronologiska händelser*. `path_to_file` anger vilken fil som skall läggas till men kan istället utgöras av en mapp som då rekursivt lägger till allt sitt innehåll (filer och undermappar).

En viktig sak att tänka på är att `git add` lägger till filer i deras nuvarande tillstånd och tar därför inte hänsyn till senare ändringar som sker i filen. Ytterliggare modifiering av filer som redan har ingående ändringar i *snapshot* måste således registreras som nya händelser för att inkluderas.

Att ta bort en fil från projektet är även det en händelse som skall registreras. Kommandot `git rm` anger i sin ursprungsform att filen skall tas bort från såväl *Staging Area* (**unstage**) samt *Working Tree* (**untrack**), medan flaggan `--cached` begränsar detta till att enbart gälla en **unstage**.

```
$ git rm [path_to_file]         // unstage & untrack
$ git rm –cached [path_to_file] // only unstage
```

För att få en översikt över filers tillstånd i förhållande till *Working Tree* och *Staging Area* används kommandot `git status`, som ger följande *output*;

```
Changes to be committed:
   ...
Changes not staged for commit:
   ...
Untracked files:
   ...
```

Utifrån denna översikt talar Git om vilka filer vars förändringar kommer att ingå i nästa **commit**, filer vars ändringar ännu inte blivit **staged**, samt filer som finns i *Working Tree* men ännu inte är **tracked**. Medan `git status` visar vilka filer som innehåller modifieringar så avslöjar den inte vilket innehåll som är modifierat. För att istället se detaljer kring förändringarna så kan man generera en **diff** genom kommandot `git diff`, som i sin ursprungsform listar modifieringar i innehåll som ännu inte blivit **staged**. Lägger du till flaggan `--staged` så listas istället ändringar som ingår i nästa *commit*.

För att lagra en *snapshot* till versionshistoriken används kommandot `git commit`, som registrerar alla **staged changes** för den aktuella **branchen** (se avsnitt **5 Förgreningar & Sammanföring**). Vill man dock kringgå *Staging Area* kan flaggan `-a` läggas till kommandot som instruerar Git att automatiskt inkludera alla ändringar, **staged** såväl som **unstaged**, till den aktuella *committen*. När kommandot exekveras öppnas den texteditor som du tidigare angav i konfigurationen och du uppmanas att skriva ett meddelande, s.k. **commit message**, för den aktuella *committen*.

```
<Kortfattad beskrivning, max 50 tecken>

<Detaljerad beskrivning>
```

Ett **commit message** består av två delar; *summering* och *detaljerad beskrivning*. I den första raden av meddelandet skall en summering skrivas som kortfattat redogör för bidragets modifikationer. Därefter följer en tom rad och nedanför denna följer en detaljerad beskrivning av *committen*. Viktigt här är att summeringen

på ett tydligt sätt reflekterar *committens* ansvarområde, då det är denna rad som visas i versionshistorikens **log output**. Förutom **commit message** består varje *commit* även av information gällande *hashvärdet* som utgör dess **id**, vem som är **författare** samt **datum** och **tidpunkt** för bidraget;

```
commit b650e3bd831aba05fa62d6f6d064e7ca02b5ee1b
Author: Erik Ström <eristr@student.miun.se>
Date:   Wed Sep 13 00:45:10 2017 +0100

   <commmit message>
```

Genom att lägga till flaggan **`-m`** så kan en summering anges direkt med kommandot, utan att behöva blanda in en texteditor. Detta kan vara smidigt i det fall ingen detaljerad beskrivning behövs!

```
$ git commit          // base command
$ git commit --a      // commit all changes, even unstaged ones
$ git commit -m [msg] // provide message with command
```

> *Om du har svårt att kortfattat beskriva innehållet i en specifik commit så är det troligt att dess **cohesion** (sammanhållning) är låg. Kom ihåg att Git låter dig strukturera **logiska snapshots** genom att välja ut vilka ändringar som skall ingå till varje commit. För att då öka **cohesion** och underlätta beskrivningen överväg att dela upp innehållet till flera **changesets**, där alla förändringar relaterar till varandra, och committa dem separat. Ju högre grad av **cohesion** du ger dina commits desto lättare blir det att följa filernas utveckling ur ett historiskt perspektiv.*

För att inspektera historiken över utförda *commits* används kommandot **`git log`** som i sin ursprungsform listar alla *commits* som finns i historiken med dess **id**, **författare**, **tidpunkt** samt **summeringen** av *commit message*. Denna *output* kan dock snabbt bli mycket lång och för att kondensera informationen kan flaggan **`--oneline`** användas, som då begränsar utrymmet till en rad per *commit*. För att enbart se historiken för en specifik fil så kan sökvägen till filen anges och för att se vilka filer som påverkats för varje *commit* så anges flaggan –**`stat`**.

```
$ git log                // lists info about commits
$ git log --oneline      // each commit on a single line
$ git log –oneline [file] // historical changes to specified file
$ git log --stat         // list files affected by each commit
```

> *Som för de flesta gitkommandon finns det mängder av flaggor att ange tillsammans med **`git log`** för att kontrollera den output som genereras. Många tillgängliga flaggor presenteras i kapitel **2.3** i kursboken!*

De kommandon som vi hittills presenterat utgör basen för den funktionalitet som krävs för ett smidigt arbetsflöde i Git. **Illustration 4.2** visar hur dessa kommandon relaterar till arbetsflödet i stort.

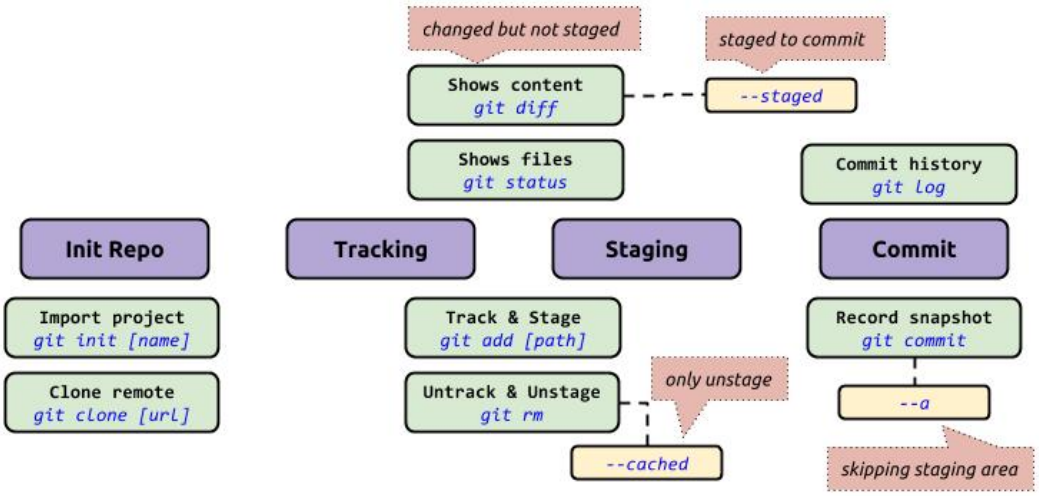


Illustration 4.2: Git kommandon

4.2 Ångra arbete (*Undoing*)

Ibland finns behovet att kunna redigera sitt arbete omgående på grund av att misstag har begåtts eller då omständigheter helt plötsligt förändras. Git erbjuder olika tillvägagångssätt för att ångra arbete alltigenom arbetsflödet och beroende på *vad* som skall ångras, vilket *skede* det är i processen samt hur övriga omständigheter ser ut så föreligger aspekter som måste beaktas.

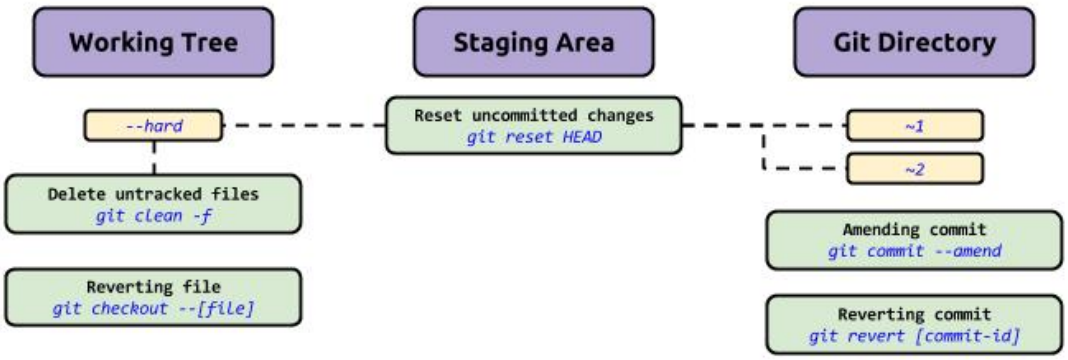


Illustration 4.3: Git undoing

Du kan relativt problemfritt ångra ännu ej versionshanterade förändringar i *Working Tree* och *Staging Area* eftersom detta arbete inte lämnat det lokala *scope* som utgör ditt privata *repository*. Om du exempelvis lagt till en oönskad fil till din *snapshot* och behöver ångra detta så kan du använda kommandot **git reset** för att utföra en **unstage** på filen;

```
$ git reset HEAD [file]
```

`HEAD` är en pekare som alltid refererar till den senast versionshanterade *snapshot*, och det som är viktigt här är att den specificerade filen enbart genomgår en **unstage**, d.v.s. att filen i *Staging Area* blir likvärdig med den motsvarighet som `HEAD` pekar till. *Working Tree* lämnas dock orörd där förändringarna på filen finns bevarade. För att även ångra filens modifikationer i *Working Tree* så kan flaggan `--hard` läggas till kommandot, men ha i åtanke att detta tar bort alla spår av dessa förändringar. Använder du kommandot utan att specificera en fil så inkluderas istället alla aktuella förändringar.

Om du inte är redo att förlora ändringar som du gjort i filen men vill testa att arbeta med en annan version så finns alternativet att *checka ut* denna version, arbeta med den och sedan återgå till samma fil senare;

```
$ git checkout HEAD [file]          // check out file from head
$ git checkout other_branch [file]  // check out file from branch
$ git checkout commit_id [file]     // check out file from commit
```

De verkliga riskerna med att ångra arbete föreligger när det relaterar till bidrag som finns i *versionshistoriken*. Det som är viktigt att komma ihåg är att projekt med flera deltagare normalt baserar samarbetet på gemensamma datakällor, vilket gör projektet mycket känsligt för omskrivning av historiken. Som ett exempel är den information som finns i *Git Directory* versionshanterade *snapshots* av projektets tillstånd över tid och där samma *revision* kan föreligga som grund för flera medlemmars pågående arbete. Om någon av deltagarna skulle modifiera en sådan *revision* kan det leda till katastrofala konsekvenser som på förhand är svåra att förutspå. På grund av detta säger en viktig tumregel att ***aldrig utföra modifieringar direkt på publika commits, utan istället genomföra dessa förändringar i form av nya bidrag!*** Vad detta innebär skall vi nu titta närmare på!

För att belysa dessa risker tittar vi på kommandot `git reset` som även kan användas för att ångra *commits* i **Git Directory**. Detta sker genom att flytta positionen på `HEAD` till att peka på en tidigare *commit*;

```
$ git reset HEAD~1  // reset to previous commit
$ git reset HEAD~2  // reset to second last commit
```

Ett alternativt tillvägagångssätt är att ange flaggan `--amend` till kommandot `git commit` som lägger till modifikationer till senaste *commit*. Detta kan vara användbart i det fall du behöver uppdatera **commit message** eller glömde lägga till någon fil.

Problemet med båda dessa alternativ är dock att de skriver om *versionshistoriken* och därför absolut inte bör användas i *publika repositories*. Ni avråds från att använda dessa metoder, men om ni ändå gör det se då till att isolera dem till era *privata repo's* och enbart ångra *commits* som ännu inte *pushats* till en central datakälla.

Ett tryggare tillvägagångssätt är att istället genom kommandot `git revert` återgå till en tidigare *revision* som specificeras genom sitt **commit id** (*hashvärde*). Git skapar då en ny *commit* baserat på differensen mellan versionerna och genererar en **patch** som appliceras på filernas nuvarande tillstånd och **transformerar** dem till den tidigare versionen. Att generera en ny *commit* för att utföra ändringar i en tidigare *commit* kan kännas som onödigt krångligt, men ha i åtanke att det följer de grundläggande principerna bakom Git som säger att ***information skall läggas till, aldrig tas bort***. Ni rekommenderas att ha som rutin att alltid använda denna metod för att ångra versionshanterad information!

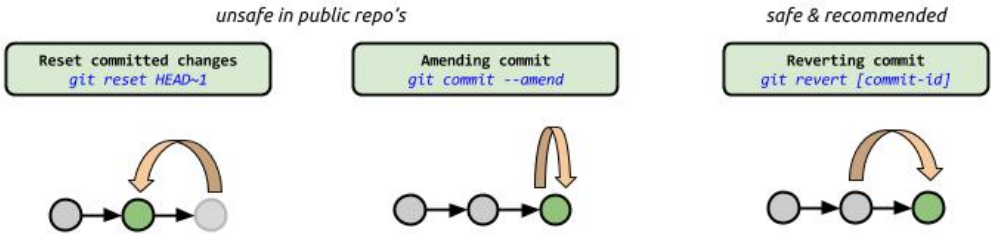


Illustration 4.4: Git Directory undoing

5 Förgreningar & Sammanföring (*Branches/Merging*)

> *Läs kapitel **3 – Git Branching** i kursboken och studera principerna bakom Git's förgreningssystem, hur olika grenar kan sammanföras samt hur eventuella konflikter bör hanteras!*

Som tidigare har nämnts möjliggör förgreningssystem *icke-linjära* arbetsflöden, vilket innebär att flera orelaterade riktningar kan utvecklas parallellt med varandra. Att skapa en ny gren (*branch*) kan likställas med att sätta upp en helt ny utvecklingsmiljö komplett med egen **Working Directory** och **Staging Area**. Även *versionshistoriken* hålls isolerad men ärver sitt ursprungliga tillstånd från den *branch* förgreningen baseras på. Denna förgrening av historiken kallas **fork** och likt en gaffels tänder ligger alla grenarna parallellt mot varandra samtidigt som de delar samma bas.

Till skillnad från många andra VCS så är Git's förgreningssystem mycket effektivt. Som ett exempel så blir *branching* i **SVN** möjlig genom att det kopierar hela projektet och dess data till en helt ny mapp, på samma sätt du skulle göra manuellt utan VCS. Detta gör dock systemet långsamt, felbenäget samt onödigt redundant. Som kontrast till detta så innebär en *branch* i Git inget mer än en pekare till en specifik *commit* och där en avskild versionshistorik håller reda på alla relaterade *commits*. Detta gör förgreningssystemet otroligt smidigt och effektivt, och då skapandet av nya *branches* inte utgör någon nämnbar kostnad finns inga egentliga hinder för experimentering och flexibla arbetsflöden.

Arbetet med *branches* utförs i Git genom särskilda kommandon. Grundkommandot **git branch** används för att ***lista***, ***skapa*** samt ***ta bort*** *branches*, medan **git checkout** används för att ***växla*** mellan existerande *branches*.

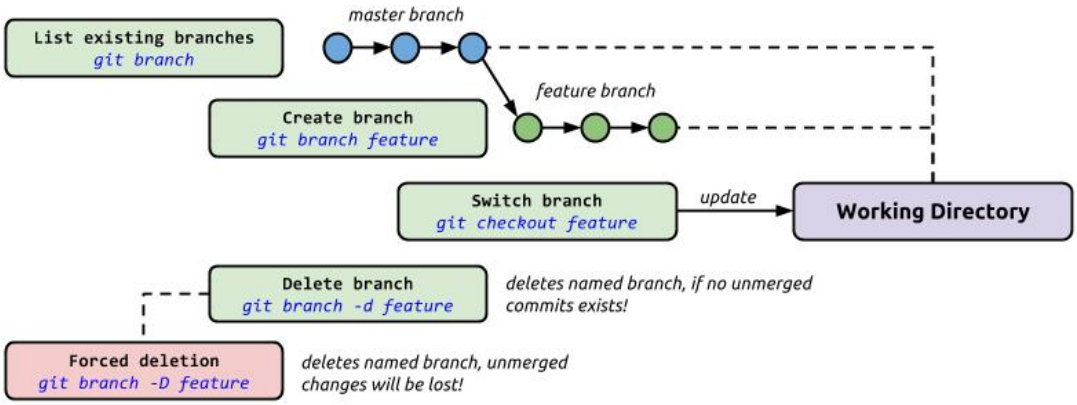


Illustration 5.1: Git branches

Master-branchen är en förvald (*default*) *branch* som Git automatiskt skapar i och med den första *commit* som görs i ett repository. Många utvecklare väljer denna gren som sin ***main branch***, en permanent gren som alltid utgör projektets stabila tillstånd. Själva utvecklingen sker istället i särskilda utvecklingsgrenar som sammanförs (**merge**) med **master-branchen** i takt med att de färdigställs.

> *Värt att notera är att kommandot* `git checkout` *ersätter den nuvarande inlästa branchen i* ***Working Directory*** *med angiven branch. Med det i åtanke är det viktigt att ha ett s.k.* ***clean directory*** *inför bytet, dvs att det inte finns arbete som är* ***uncommitted****. Annars finns risken att arbete går förlorat!*

Merging kallas den process som sammanför *commits* från en *branch* in i en annan, vilket resulterar i en kombination av bådas *versionshistorik*. Det spelar en central roll för arbetsflödet och är det som i verkligheten möjliggör flexibla och icke-linjära arbetssätt. Smidiga och effektiva förgreningar i all ära, men de förlorar betydelse om det inte också finns ett enkelt sätt att återförena divergerade utvecklingsgrenar. Lyckligtvis gör Git även detta enkelt!

Det finns i huvudsak två metoder för att genomföra en *merge;* **Fast-forward** och **3-way**, och vilken som används bestäms automatiskt av Git baserat på den tillgängliga *versionshistoriken*. Båda metoderna initieras utifrån samma kommando, `git merge`, och gemensamt är att den *branch* som skall stå som mottagare för sammanföringen *checkas ut* innan kommandot körs. Om vi som exempel har implementerat en **feature** i sin egen **feature_branch** och nu vill sammanföra denna med **master-branchen** så måste vi först *checka ut* **master**, som då utgör vår *Working Directory*, innan vi utför sammanföringen;

```
$ git checkout master
$ git merge feature_branch
```

Efter sammanföringen förblir **feature_branch** oförändrad medan **master** nu innehåller historiken från båda grenarna. Detta innebär att varje *commit* från **feature_branch** nu finns tillgänglig och kan ses av alla deltagare som har access till **master**.

> *Ett alternativ till* ***merge*** *är* ***rebase****, vilket gör det möjligt att* ***städa rent*** *i historiken innan* ***feature*** *sammanförs med* ***master****. Tanken här är att* ***master*** *skall utgöra en relevant och kompakt tidslinje över tillförda bidrag och att varje enskild commit i otaliga utvecklingsgrenar inte behöver finnas tillgänglig, då det istället gör historiken onödigt grötig.* ***Rebase*** *är ett kraftfullt verktyg men tillför samtidigt mer komplexitet, där nya problem riskerar framkomma. Läs gärna mer i kursboken, men* ***rebase*** *är inget ni behöver använda under denna kurs.*

Det som skiljer en **Fast-forward** *merge* från en **3-way** ligger i omfattningen av differensen mellan sändande och mottagande förgreningars historik. I det fall skillnaden enbart utgörs av *commits* i sändaren så blir en **Fast-forward** av mottagaren möjlig. Som namnet antyder *snabbspolas* mottagande *branch* till att överrensstämma med historiken av sändande *branch*. I det fall även mottagaren innehåller förändringar som inte finns registrerade i sändaren så kan en sådan snabbspolning inte ske, då hänsyn även måste tas till dessa skillnader. Här används istället **3-way** *merge* som skapar en helt ny *commit* hos mottagaren baserat på båda grenarnas differerande historik.

Som ett exempel på **Fast-forward** tar vi två *branches*, **master** som är projektets huvudgren samt **feature** som är en förgrening av **master**. I sin ursprungsform är alltså **feature** en direkt avspegling av **master** men det fortsatta arbetet i **feature** kommer att utgöra differensen dem emellan. Efter två *commits* skall så arbetet sammanföras och för att göra det så flyttar vi helt enkelt pekaren för **master** till att peka på samma *commit*

som **feature**. Principen bakom **Fast-forward** är alltså inte svårare än så, men i praktiken är det relativt ovanligt med sådana sammanföringar då *icke-linjära* arbetsflöden och bidrag från andra deltagare orsakar förändrar i mottagande *branch* som omöjliggör snabbspolning.

Som exempel på **3-way** *merge* kan vi utgå från scenariot ovan; en **master** *branch* som förgrenas till en **feature**. Men innan arbetet i **feature** är färdigt finner vi en bugg i koden som måste åtgärdas innan **feature** blir färdigställd. Vi skapar således en ny *branch* från **master**, fixar buggen och sammanför detta i **master**. När arbetet med **feature** sedan blir färdig och skall sammanföras med **master** går det inte att som tidigare snabbspola pekaren för **master** till att peka på samma *commit* som **feature**, eftersom det finns ytterliggare en *commit* att ta hänsyn till; *bugg-fixen*. För att genomföra *mergen* skapas istället en ny *commit* i **master** som sammanför *bugg-fixen* med historiken från **feature branch**.

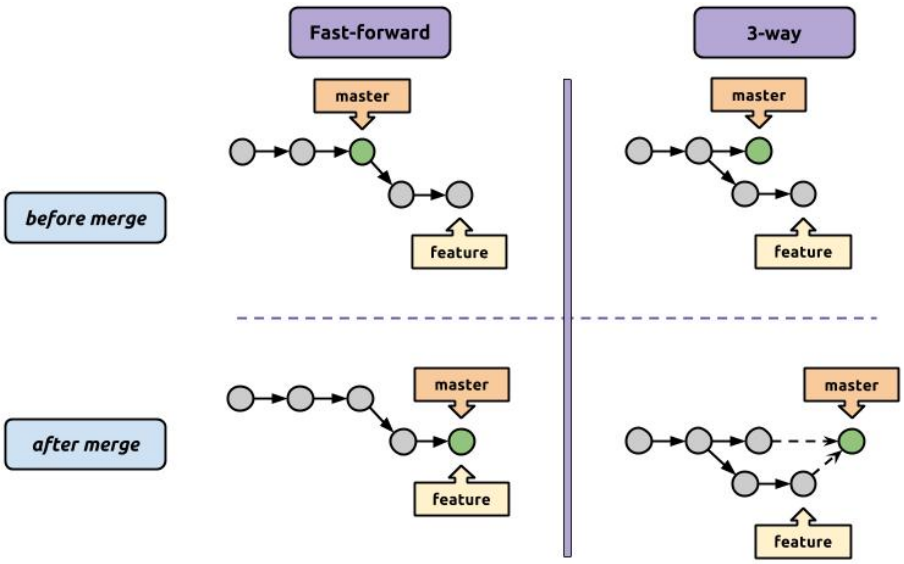


Illustration 5.2: Git merge

Val av den metod som används för sammanföring sker som tidigare nämnts automatiskt av Git. Men det är viktigt att uppfatta de principiella skillnaderna som föreligger mellan **Fast-forward** och **3-way merge** för att också förstå förekomsten av eventuella *versionskonflikter*, s.k. **merge conflicts**. En sådan konflikt uppstår när sammanföring skall ske mellan två förgreningar som båda innehåller ändringar i samma delar (t.ex. samma kodstycken). Detta kan naturligtvis aldrig inträffa under en *Fast-forward merge* där alla ändringar finns i sändande *branch*, men är ett ofta förekommande problem i *3-way* metoden.

Grunden till förekomsten av *versionskonflikter* är att Git helt enkelt inte vet hur den skall förhålla sig till konkurrerande ändringar från separata källor. Om **bugg-fixen** i exemplet ovan innebar ändringar till samma fil som används för implementeringen av **feature**, så tillkommer den ändringen efter skapandet av **feature-branchen**. Om Git vid sammanföringen av **feature** skulle godta dess ändring utan hänsyn till **bugg-fixen**, så skulle det innebära att den sistnämndas innehåll helt ersätts med **feature** och i praktiken resultera i att **bugg-fixen** helt enkelt raderas. Lyckligtvis så gör Git inte så, utan vägrar istället utföra sammanföringen till dess att denna konflikt har åtgärdats. En *output* från Git avslöjar problemet;

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

Om du kör **`git status`** i detta läge så genereras följande *output*;

```
# On branch master
# Unmerged paths:
#
#    both modified:  data.txt
<<<<<<< HEAD
   This content is from the current branch.
=======
   This is a conflicting change from another branch.
>>>>>>> feature
```

Alla filer som uppvisar *versionskonflikter* listas under sektionen **`Unmerged paths`** och särskilda annoteringar visar det innehåll som utgör problemet i relation till dess *branch*. Delen före **`=======`** gäller den mottagande *branchen* (**master**) medan det efter gäller sändaren (**feature**). För att åtgärda konflikten så tar vi bort annoteringarna **`<<<<<<<`**, **`=======`** samt **`>>>>>>>`** och ändrar innehållet till den text som skall användas. För att sedan tala om för Git att vi är färdig så lägger vi till (**stage**) filen igen följt av en *commit*, som färdigställer den **3-way merge** som vi tidigare påbörjat;

```
$ git add data.txt
$ git commit
```

6 Fjärran Datakällor (*Remote Repositories*)

> *Läs kapitel **2.5 – Working with Remotes** i kursboken och studera grunderna för hur kommunikation med fjärran datakällor kan utföras!*

En fjärran datakälla är helt enkelt allt utanför din *privata repositiory* och kan utgöras av exempelvis en *central server* eller någon annan medlems repo som denne gjort tillgänglig åt dig. Du ansluter till en extern källa över ett nätverksprotokoll och synkroniserar andras bidrag med ditt privata repo genom att dra ned (**pull**) förändringar från källan. Har du dessutom skrivrättigheter så kan du även **pusha** dina egna bidrag till källan och på så vis göra det tillgängligt för andra deltagare.

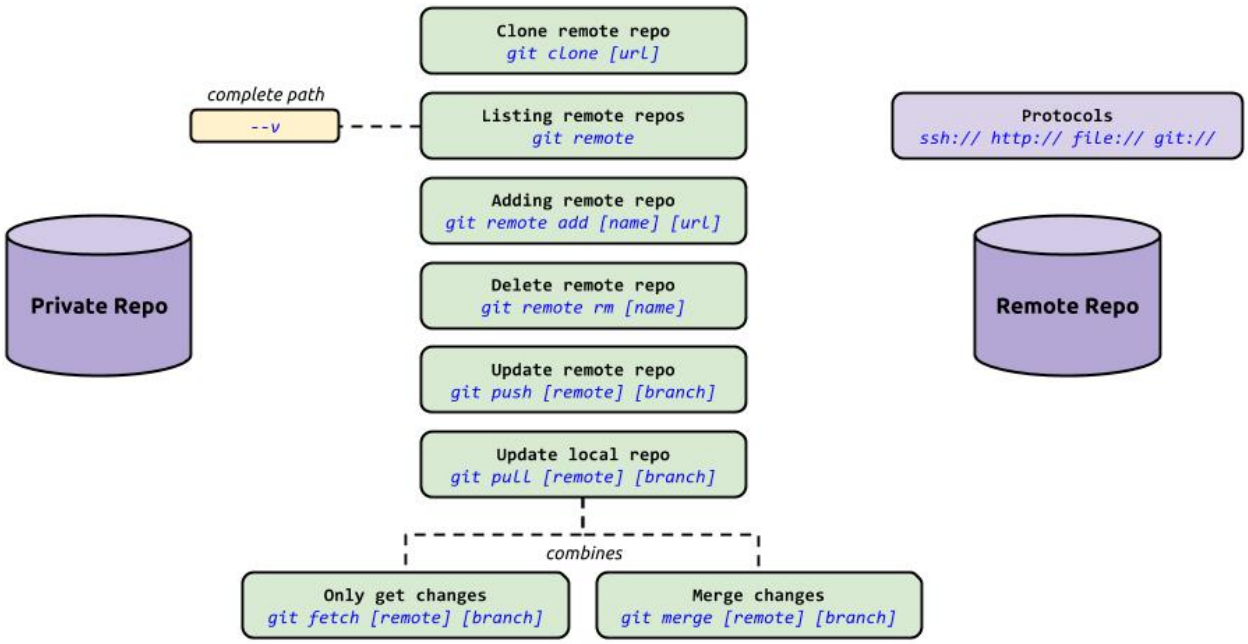


Illustration 6.1: Remote commands

För att skapa en ny kontakt till en fjärran källa används kommandot `git remote add` tillsammans med ett namn för repot samt dess url;

```
$ git remote add [name] [url]
```

Du kommer sedan åt detta fjärran repo genom att referera till det namn du angivit, som nu fungerar som dess **alias**. För att ta bort kontakten för ett repo använder du kommandot `git remote rm` och specificerar just detta namn;

```
$ git remote rm [name]
```

För att lista alla fjärran repos som har registrerats så använder du kommandot **`git remote`**, och lägger du till flaggan **`-v`** så visas varje repos fullständiga url;

```
$ git remote     // list all remote repos
$ git remote -v  // list all remote repo's full path
```

Om du initierade ditt *repository* genom kommandot **`git clone`** så kommer **`git remote`** visa en **`origin`** till det *remote repository* som var utgångspunkten för kloningen. Git namnger alla fjärran repos med prefixet **`origin`** för att särskilja dessa från ditt lokala repo;

```
$ git clone git@bitbucket.org:some-user/some-repo.git
$ git remote
origin/master
origin/some-branch
origin/another-branch
```

Du hämtar *branches* från en fjärran datakälla genom kommandot **`git fetch`** där du specificerar såväl *datakällan* som den *branch* du vill hämta. Alternativt så utelämnar du namnet på *branchen*, vilket istället hämtar alla tillgängliga förgreningar. Vill du sedan lista alla hämtade *branches* kan du göra så via kommandot **`git branch`** tillsammans med flaggan **`-r`**;

```
$ git fetch [remote] [branch] // get specified branch
$ git fetch [remote]          // get all available branches
$ git branch -r               // list all downloaded branches
```

För att sedan sammanföra en hämtad *branch* med din lokala så använder du **`git merge`** på samma sätt som du gör för dina lokala förgreningar;

```
$ git merge [remote] [branch]
```

Aktiviteterna **fetch**/**merge** är så vanligt förekommande fenomen i *distribuerade arbetsflöden* att ett kortkommando för dessa har skapats; **`git pull`**, som automatiskt hämtar och sammanför förändringarna.

```
$ git pull [remote] [branch] // fetches & merges
```

7 Arbetsflöden (*Git Workflows*)

> *Kursbokens kapitel **5.2 – Distributed Workflows** tar upp några grundläggande arbetsflöden för Git. Om ni är intresserade så kan ytterligare information läsas i artikeln Comparing Workflows [7].*

Det finns mängder av alternativa arbetsflöden för hur Git kan integreras med utvecklingsprocessen och de allra flesta faller inom spektrumet för två motparter; **centraliserat** jämte **integrerat** arbetsflöde. Oavsett vilket Gitflöde som används så sker den primära utvecklingen inom deltagarnas privata och från varandra isolerade *repositories* och skillnaderna består istället huvudsakligen av förhållningssättet gentemot publika datakällor och hur medlemmarnas bidrag kan göras tillgänglig.

Ett **centraliserat flöde** utgår, precis som det låter, från ett centralt *repository* som normalt utgörs av en server, exempelvis någon *file hosting service*. Alla deltagare har läs- och skrivrättigheter (över *SSH-protokollet*) till detta centrala repo och synkroniserar sitt arbete genom **pull** och **push** kommandon. Detta arbetsflöde lämpar sig kanske bäst för mindre team där fördelarna med öppen tillgång till **origin** blir mer påtaglig. Dock så kan det samtidigt innebära vissa problem, särskilt i större team. Dels är det inte alltid önskvärt att varje medlem har skrivrättigheter till *origin*, och även om det är enkelt att återgå till tidigare versioner så innebär projektåterställningar alltid någon form av avbrott i teamets generella arbetsflöde. Sedan tillämpas i dessa flöden något som kallas ***first come, first served*** i det att *fast-forward merges* normalt endast blir möjlig för den som tidigast **pushar** sitt bidrag till servern, vilket ger efterkommande en avvikande historik som måste synkroniseras innan deras bidrag kan **pushas**. Detta kanske inte låter som ett stort problem, att det bara skulle krävas en **fetch + merge** (eller **pull**) innan bidraget **pushas**, men eftersom varje deltagare har fulla skrivrättigheter finns heller inga garantier att denna lokala *merge* inte innebär konflikter med ditt eget bidrag.

Integrerade arbetsflöden söker lösa de problem gällande *säkerhet* och *skalbarhet* som ofta föreligger i den **centraliserade** motsvarigheten. Även här används oftast någon form av *central datakälla* som utgör **origin** för samtliga medlemmar, med skillnaden att skrivrättigheter enbart ges till enstaka individer som hanterar själva **integreringen** av alla bidrag. Varje deltagare har, förutom sitt privata repo, även ett *publikt repo* till vilket de **pushar** sina bidrag i takt att de blir färdigställda. En **integrator** hämtar (**pull**) sedan det aktuella bidraget och utför *kvalitetsbedömningar* innan det slutligen integreras med den *centrala datakällan*. Detta ger hög grad av *säkerhet* eftersom endast läsrättigheter (normalt *HTTP-protokollet*) behövs i samspelet mellan projektets aktörer och dessutom tillämpas *kvalitetssäkring* i form av *kodgranskning* som utförs av projektets nyckelindivider, vilket alltid bör understödja *skalbarheten*.

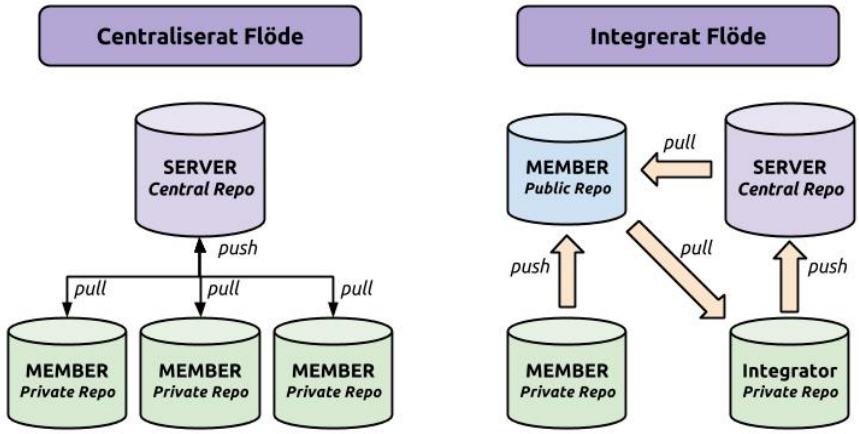


Illustration 7.1: Arbetsflöden, Centraliserat vs. Integrerat

Vilket Gitflöde ni kommer att använda beror helt på omständigheterna kring projektet. För att snabbt och enkelt komma igång med ett samarbete i ett litet team kan det **centraliserade** flödet passa bra, så länge man förstår dess risker och planerar för dem, medan ett **integrerat** flöde passar dem som kräver hög grad av säkerhet samt *kontroll* över hur bidrag skall *integreras*.

Flödet behöver heller inte uteslutande baseras på endera än det andra, utan istället vara en kombination av dem. Ett exempel på flöde som lånar aspekter från båda dessa extremer kan ses i **Illustration 7.2**. Här utgår medlemmarna *Kerstin* och *Bengt* från en central datakälla som utgörs av **origin** förvarad i **Bitbucket**. Likt ett **integrerat** flöde så **pushas** bidrag inte direkt till detta centrala repot utan istället utformas s.k. ***pull requests*** som gör övriga deltagare medvetna om att det finns ett nytt bidrag tillgängligt. Det finns ingen designerad **integrator** utan i detta team står medlemmarna som varandras godkännare. När exempelvis *Bengt* implementerat sin **feature** och sammanfört detta med sin utvecklingsgren så skapar han en **pull request** via teamets konto på **Bitbucket** och anger *Kerstin* som dess ***reviewer***. *Kerstin* drar ned *Bengts* bidrag för att bedöma dess kvalitet och först när hon godkänner bidraget kan det sammanföras med **origin**. På samma sätt figurerar *Bengt* **reviewer** för *Kerstins* bidrag.

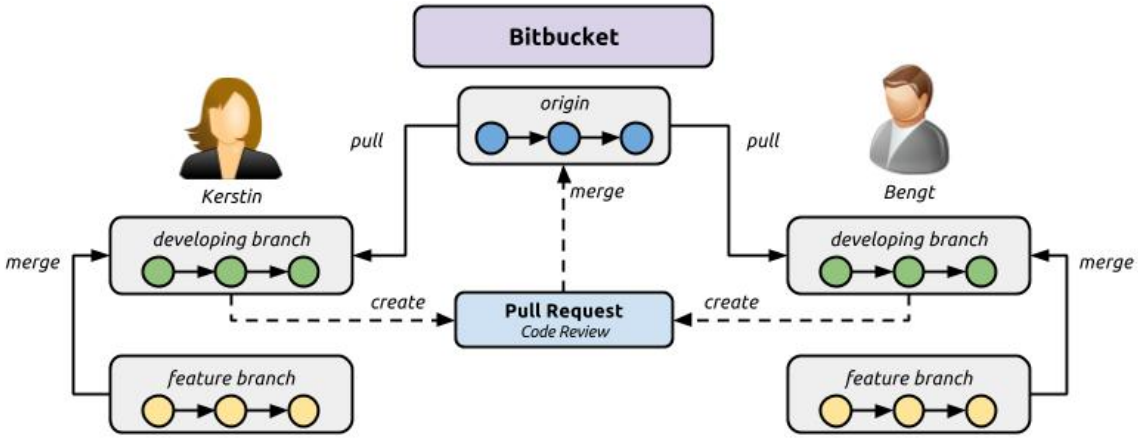


Illustration 7.2: Git arbetsflöde

I laboration 1 får du arbeta vidare med Git som tema!

Referenser

[1] Wikipedia, "Diff utility", 2017. [Online]
Tillgänglig: https://en.wikipedia.org/wiki/Diff_utility
[Åtkomst: 27 Juli 2017]

[2] Wikipedia, "Patch (Unix)", 2017. [Online]
Tillgänglig: https://en.wikipedia.org/wiki/Patch_%28Unix%29
[Åtkomst: 27 Juli 2017]

[3] S. Chacon, B. Straub, "Pro Git", 2nd Ed. 2014. [Online]
Tillgänglig: https://git-scm.com/book/en/v2
[Åtkomst: 17 Juli 2017]

[4] GitHub, "Pro Git 2nd Edition", 2017. [Online]
Tillgänglig: https://github.com/progit/progit2
[Åtkomst: 17 Juli 2017]

[5] Atlassian, "Set up SSH for Git", 2016. [Online]
Tillgänglig: https://confluence.atlassian.com/bitbucket/set-up-ssh-for-git-728138079.html
[Åtkomst: 17 Juli 2017]

[6] Wikipedia, "Distributed version control", 2017. [Online]
Tillgänglig: https://en.wikipedia.org/wiki/Distributed_version_control
[Åtkomst: 27 Juli 2017]

[7] Atlassian, "Comparing Workflows", 2017. [Online]
Tillgänglig: https://www.atlassian.com/git/tutorials/comparing-workflows
[Åtkomst: 10 Augusti 2017]

[8] Daring Fireball, "Markdown: Syntax", 2017. [Online]
Tillgänglig: https://daringfireball.net/projects/markdown/syntax
[Åtkomst: 20 Augusti 2017]

Bilaga A: Vanliga Git-kommandon

SKAPA REPOS
`$ git init [project_name]`
Skapar ett nytt lokalt repo med det angivna namnet
`$ git clone [url]`
Laddar ned angivet projekt och all dess versionshistorik

ÄNDRINGAR
`$ git status`
Listar alla nya/modifierade filer som skall 'commitas'
`$ git diff`
Visar skillnader i filer som ännu inte är 'staged'
`$ git add [file]`
Lägger till 'snapshot' för angiven fil för kommande 'commit'
`$ git diff --staged`
Visar skillnader mellan filer som är 'staged' och deras senaste versioner
`$ git reset [file]`
Tar bort angiven fil från 'staging area', men bevarar innehåll
`$ git commit -m "[descriptive_message]"`
Registrerar allt innehåll i 'staging area' till versionshistoriken

REFAKTORERING
`$ git rm [file]`
Tar bort angiven fil från 'working directory'
`$ git rm --cached [file]`
Tar bort filen från versionshantering, men bevarar den lokalt
`$ git mv [file_original] [file_renamed]`
Ändrar filens namn och förbereder den för 'commit'

SYNKRONISERA ÄNDRINGAR
`$ git fetch [bookmark]`
Laddar ned all historik från repot 'bookmark'
`$ git merge [bookmark]/[branch]`
Kombinerar bookmark's branch med aktuell lokal branch
`$ git push [alias] [branch]`
Laddar upp alla 'commits' för den lokala branchen till 'remote repo'
`$ git pull`
Laddar ned all historik och infogar förändringar

KONFIGURATION
`$ git config --global user.name "[name]"`
Anger det namn som skall visas för dina 'commit' transaktioner
`$ git config --global user.email "[email_address]"`
Anger den epost som skall visas för dina 'commit' transaktioner
`$ git config --global color.ui auto`
Färgsättning av output som visas i terminalen/kommandotolken

FÖRGRENINGAR
`$ git branch`
Listar alla lokala branches i det aktuella repot
`$ git branch [branch_name]`
Skapar en ny branch med det angivna namnet
`$ git checkout [branch_name]`
Byter till den angivna branchen och uppdaterar working directory
`$ git merge [branch_name]`
Kombinerar den angivna branchens historik med den aktuella
`$ git branch -d [branch_name]`
Tar bort den angivna branchen

HISTORIK
`$ git log`
Listar versionshistorik för den aktuella branchen
`$ git log --follow [file]`
Listar versionshistorik för angiven fil, inklusive namnändringar
`$ git diff [first_branch] [second_branch]`
Visar skillnader i innehåll mellan två 'branches'
`$ git show [commit]`
Visar metadata och förändringar för angiven 'commit'

ÅNGRA BIDRAG
`$ git reset [commit]`
Ångrar alla bidrag efter angiven 'commit', men bevarar ändringar lokalt
`$ git reset --hard [commit]`
Kasserar all versionshistorik och bidrag efter angiven 'commit'