



Vállalatirányítási rendszer

Git kezelés

Felhasználói útmutató

Git verziókezelés

## Fogalomtár

Repository

Az adatbázis, amely a fájlokat és azok összes verzióját tárolja a repository (röviden repó).  
Working copy

A kódbázis helyi verziója, amin éppen dolgozunk.

Revision

A revision a verzió szinonimája: fájlok halmazának egy egyértelműen azonosítható változata, a head pedig a repositoryban található legfrisebb verzió

Baseline

Egy dokumentum vagy fájl jóváhagyott verziója, melyhez az azt követő változtatásokat viszonyítják.

Branch (ág)

Egy branch létrehozásakor másolatot készítünk az egész kódbázisról, amit aztán külön láthatunk el változtatásokkal. Ez nagyon hasznos például akkor, ha egy új funkción dolgozunk a kódban, de ez akár hosszú ideig is tarthat, és csak kész állapotában szeretnénk majd belefoglalni az éles szoftverbe.  
A verziókezelt fájlok egy részhalmaza elágazhat, így azoknak több aktuális változatuk lesz egyidejűleg, melyeket akár különböző sebességgel és különböző irányokba is fejleszthetnek.

Változások címkézése (commit)

A tárolóba való feltöltés előfeltétele, hogy a változtatásokat egy szöveges üzenettel kísérve jegyezzük fel.

Merge

A merge művelettel két változtatáslistát lehet összefésülni, s ezáltal egy közös verziót létrehozni. Erre a következő esetekben lehet szükség:

* Ha egy felhasználó módosítja a saját munkamásolatát, majd letölt a szerverről egy másik módosított változatot. Ekkor a szerveren lévő változásokat össze kell fésülni a lokális munkapéldány változásaival a kliensen.
* Ha a fejlesztésben elágazás történt, majd egy hibát kijavítottak valamely ágban, s a javítást alkalmazni kell a másik ágra is.
* Ha a fejlesztésben elágazás történt, majd az ágakat különböző irányba fejlesztettek tovább, s a különböző fejlesztéseket össze kell vonni egy közös változatba (trunk-ba).

Conflict

Konfliktusról akkor beszélünk, ha ketten akarnak megváltoztatni egy dokumentumot vagy fájlt és a rendszer nem képes összeépíteni a változásokat.   
A felhasználónak ekkor fel kell oldania a konfliktust (resolve), amit vagy úgy tehet meg, hogy a változtatásokat összekombinálja vagy úgy, hogy kiválasztja az egyik változtatást és csak azt juttatja érvényre.

Check-out

Lokális másolat készítése valamely verziókezelt fájlról. Alapértelmezésben ilyenkor a legfrissebb verziót kapja a felhasználó, de általában van lehetőség konkrét verzió kikérésére is verziószám alapján.

Fetching

Adatok lekérése a központi repository-ból.

Változtatások feltöltése (push)

A szerverre a következő módon tudjuk a változtatásokat feltölteni (a feltöltés előfeltétele, hogy commit-oljunk, tehát a változtatásokat rögzítsük. A rögzítés után nem vehetünk fel új változtatásokat.)

Projekt aktualizálása (pull)

Amennyiben több számítógépen dolgozunk ugyanazon a projekten, több helyre is leklónozhatjuk.  
Tegyük fel, hogy A gépen végrehajtott és feltöltött (push) módosításainkat a B gépen is látni szeretnénk, ahová korábban már egyszer letöltöttük a projektet.  
Előfeltétel a pull-hoz, hogy commit-oljuk a B gépen esetlegesen végrehajtott módosításokat, amennyiben azok olyan fájlokat érintenek, amelyeket az A gépen is módosítottunk.

Change

Egy változtatás (change, diff vagy delta) mindig egy verziókezelt dokumentumon vagy fájlon tett változtatást jelenti.   
Rendszerfüggő, hogy milyen mértékű módosítások számítanak change-nek.

Change list

Egy change list vagy change set egy check-in művelet során bevitt változtatások listája.

Dynamic stream

Egy olyan adatszerkezet, amely egy adott tárolón lévő elemek konfigurációját reprezentálja, és időben változik.

Export

Az export a checkout-hoz hasonlít azzal a különbséggel, hogy tiszta könyvtárat csinál a verziókezeléshez szükséges metaadatok nélkül.   
Ezt a műveletet általában közvetlenül a tartalom publikálása előtt szokták használni.

Import

Az import művelettel lehet egy lokálisan tárolt adathalmazt, amely még nem munkamásolat, felmásolni a tárolóra és verziókontroll alá helyezni.

Trunk

A fejlesztés egyik olyan vonala, amely nem branch.  
Mainline

Hasonlít a trunk-hoz, de minden ágnak lehet saját mainline-ja.

Reverse integration

Az egyes ágak összedolgozása és bedolgozása a verziókezelő fő trunk-jába.

Tag

A tag, label vagy címke egy fontos időpillanatot jelöl. Egy adott fájlcsoporthoz hozzárendelhető egy címke, amely beszédes, felhasználóbarát nevet vagy verziószámot adhat a csoportnak.

Update

Az update vagy sync a repositoryban lévő változtatásokat dolgozza bele a felhasználó munkamásolatába.

## Központosított verziókezelés

Egy repository van, amely követi a tartalom változását, és a legfrissebb (vagy bármely másik) állapot megtekinthető és szerkeszthető a working copyban.  
A módosítások utána visszavezethetők a repóba, ahol a history részévé válik.  
Ehhez a repóhoz mindenki hozzáfér, és amelyet mindenki használ, a working copy-k tipikusan másik gépen találhatók, a kommunikáció a hálózaton keresztül történik.

### Központosított verziókezelés fajtái

CVS

Az egyik legkorábbi verziókezelő rendszer a CVS (Concurrent Versions System). Az alapvető igényeket kielégíti, hiszen a fejlesztők lemásolhatják a repositoryt, és a helyi változtatásaikat vissza is írhatják abba. A CVS az ütközéseket eredetileg egy meglehetősen egyszerű, first come, first serve modellel oldotta fel, ami azt jelenti, hogy változtatást mindig csak a szoftver legfrissebb verzióján lehet végrehajtani. Emiatt gyorsan és gyakran kellett a változtatásainkat visszatölteni a repositoryba, nehogy valaki megelőzzön minket. A CVS ma már támogatja a branchinget, így valamivel rugalmasabb. Fő előnye, hogy régóta jelen van, így stabilnak mondható, azonban néhány alapvető hibája miatt (nincsenek atomi műveletek - ld. később, a branching nagyon költséges művelet) ma már kevésbé népszerű.

SVN

Az Apache Subversion (SVN) annak idején a CVS alternatívájaként jött létre, ami számos bug kijavítása mellett megpróbált maximális kompatibilitást biztosítani az eredeti rendszerrel. Az adatbázis konzisztenciáját az SVN atomi műveletekkel biztosítja: egy adott változtatásnak vagy minden összetevője bekerül a repositoryba, vagy egy sem. Míg a CVS-ben a branching lassú művelet, és nem is igazán támogatta a rendszer a hosszú életű brancheket, az SVN-t sokkal inkább erre találták ki. Fő negatívumaként azt szokták felhozni, hogy centralizált: a repository egyetlen központi helyen van tárolva. Ennek az a fő hátránya, hogy ha ez a központi szerver valamiért épp nem elérhető, a fejlesztők nem tudnak hozzáférni a kódbázishoz.

Elosztott verziókezelő rendszerek

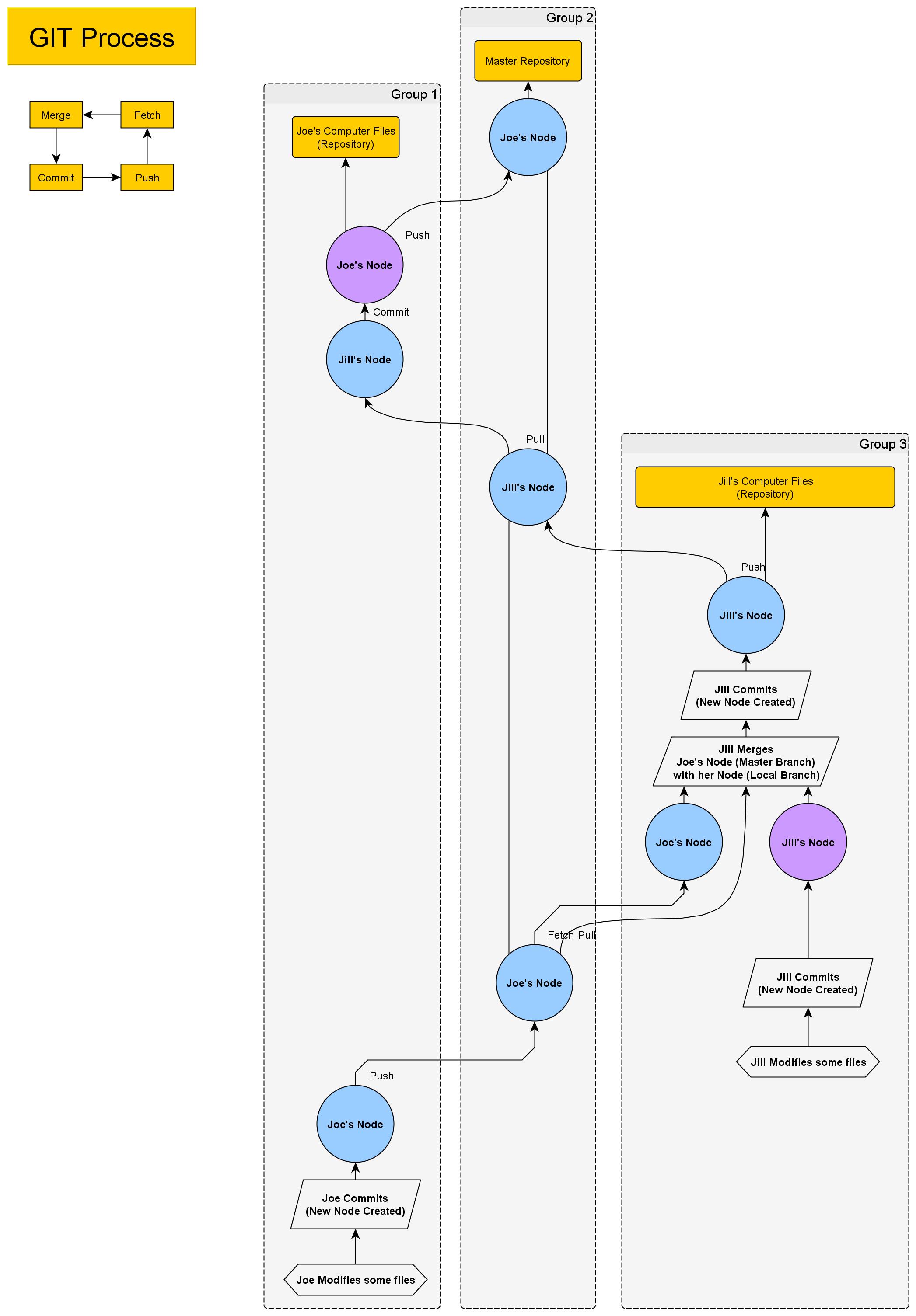
Szemben a kliens-szerver modellel, az elosztott verziókezelők decentralizált rendszerek. Itt egy központi tároló helyett minden felhasználó gépe egy-egy külön tárolóként jelenik meg. A szinkronizáció az egyes gépek között küldött patch-ek (módosításcsomagok) által valósul meg. Ez a megközelítés jelentős változásokat okoz:

Nincs nagy központi adatbázis, csak munkamásolatok vannak.

A gyakori műveletek, mint a becsekkelés, verziótörténet böngészés és a változtatások visszaállítása gyorsak, mert nem kell központi szerverrel kommunikálni.

Minden munkamásolat egy-egy backup, ami természetes védelmet ad az adatvesztés ellen.

Két fajta elosztott verziókezelő létezik, a nyitott és a zárt. A nyitott rendszereket inkább nyílt forráskódú termékeknél használják, a zártakat inkább a nem nyilvános forráskódú termékeknél.

Egyik legnépszerűbb elosztott verziókezelő rendszer a [Git](http://git-scm.com/book/en/v2).  
CVS-hez és az SVN-hez képest a Git egy radikálisan új modellt állt elő: a rendszer teljesen elosztott, így nincs szükség egyetlen központi repositoryra. A Git általában véve elődeinél sokkal gyorsabb, és a teljes verziótörténet offline is elérhető minden egyes kliensen.

Röviden összefoglalva a Git annyit csinál, hogy amikor azt mondjuk neki (commit), akkor egy directoryról csinál magának egy helyi adatbázist a .git nevű könyvtárba. Ezekkel az adatbázisokkal  
  
- nyomon tudja követni, hogy mikor hogyan változott a könyvtárunk,  
- vissza tudja állítani bármelyik korábbi (commit-olt) állapotát a könyvtárnak,  
- szinkronizálni tud egy másik gépen levő hasonló könyvtárral, közben intelligensen átvezeti a változásokat, illetve jelez, ha nem megy neki.  
A Gitnek megvan az a további előnye, hogy mindezt tudja titkosított csatornán (ssh-n keresztül).  
A szemléletbeli különbség ahhoz képest ahogy eddig éltünk az, hogy a dolgaink nem akkor vannak elmentve ha megnyomtuk az editorban a save gombot, hanem csak akkor, ha a git adatbázisunkba is bekerültek (commit). A dolgok tehát röviden a következőképp néznek ki: frissítjük a helyi adatbázisunkat a központi szerver gépről (arra az esetre gondolva, ha valaki más továbbírta azóta, hogy láttuk; pull). Ezután dolgozunk a fájljainkon.   
Ha eljutottunk egy olyan állapotba amit érdemes menteni, akkor megmondjuk melyik fájlt vegye figyelembe (add, vagy tage), majd mentjük az adatbázisba (commit). Végül pedig feltöltjük a módosított adatbázist a Szerver-re (push).

Branch-elés a fejlesztési munkafolyamatban

Branch-re több esetben is szükség lehet. A klasszikus példa, mikor kiadjuk az 1.0 verziót, elkezdjük a 2.0-ás verzió fejlesztését, és közben derül ki, hogy az 1.0-ás verzióban hibák vannak, melyre az ügyfél azonnali javítást vár, és nem várja meg a 2.0-ás verzió kifejlesztését. Ekkor elágazhatunk az 1.0-ás verziónál, és létrehozhatunk egy külön branch-et az 1.1-es verziónak, mely a hibajavítást fogja tartalmazni. A példán máris látszik, hogy később szükséges lesz a hibajavítás visszavezetése a 2.0-ás verzióba is. Ez a merge, mely során az egyik branch módosításait kell átvezetni egy másik branch-be, a mainline-ba (többnyire master branch).   
Az előbbi használati eseten ( amit nevezzünk bug-fix branch-nek) kívül is szükség lehet branch-ek használatára.

Feature branch-ek  
A feature branch esetén a fejlesztők minden egyes szoftver funkciót (feature, de nevezik story-nak is) külön ágon fejlesztenek.  
Használatával egy adott időpillanatban választhatunk, hogy mely funkciók állnak készen, melyikeket lehet kiadni. Ez a klasszikus szoftverfejlesztéssel, ahol is előre kitűzzük a verziószámokat, és hogy abban milyen funkciók lesznek elérhetőek ( maximum kicsit sakkozunk közöttük) teljesen ellentétes, hiszen itt a következő verzió azokat a fejlesztéseket fogja tartalmazni, amik készen vannak.   
A funkciók külön ágon készülnek, egymást nem zavarják, bármikor bármelyiket be lehet hozni. A probléma az lesz, hogy mi van akkor, ha két branch módosításait kell összevezetni. Ilyenkor merge-ölni kell.  
A funkciók ilyen szintű összeválogatását nevezzük cherry-picking-nek.

Cherry-picking:  
Verziókezelés terén ez gyakorlatilag azt jelenti, hogy egyenként kiválogatom, hogy milyen módosításokat szeretnék merge-ölni az egyik ágról a másik ágra.   
Ez különösen fontos akkor, ha a másik ágon folyik egy hosszabb fejlesztés, de közben egy bug-ot is kijavítottak, amit érdemes áthozni az én ágamba is.  
  
Release branch  
Abban az esetben, ha több feature branch-et szeretnénk egy release-be összevonni, akkor az integráció, a merge akár hosszabb folyamat is lehet. Ilyenkor az integrációval akár meg is akaszthatjuk a fő fejlesztést, ha azt a mainline-on végezzük. Ilyenkor szokták bevetni a release branch fogalmát.   
A release branch-en történik a merge, valamint történhet a tesztek.  
És amikor ez sikeresen befejeződik, csak ekkor kerülnek vissza a módosítások a mainline-ba. Ezzel a fősodorban mindig az aktuálisan működő, kitesztelt verziónk lesz, melyből biztonságosan le lehet ágazni, és nem kell kérdezni, hogy egy új fejlesztés (branch nyitása) most honnan történjen.

Branch-elés hátrányai:

Branch-eléssel sok olyan dolog is a felszínre kerül amelyekkel kényelmetlenebbé teszi a munkafolyamatot. Nézzük ezek közül egy párat:

* Adminisztrálni kell, hogy melyik branch-ben éppen mi történik. Valamint ezt meg kell osztani a fejlesztőgárda tagjaival is, hogy tudják, mikor melyik ágba kell fejleszteni, mikor melyik ágból lehet új branch-t nyitni.
* Merge. Azt hiszem ez önmagáért beszél. Amennyiben módosításokat akarunk ágak között átvinni, nem egyszerű feladat. Persze, az újabb eszközök már támogatják, de gondoljunk csak a szemantikus conflict-okra.
* A branch-eket meg kell szüntetni, különben elburjánzanak, elszaporodnak, így nehéz követni, hogy melyikben éppen milyen fejlesztés folyik, melyikkel mi a cél. Sajnos gyakran előfordul, hogy az ágak annyira elcsúsznak egymástól, hogy a fejlesztőcsapatnak már esélye sincs a merge-re, így párhuzamosan fejlesztenek több ágat.
* Egy fejlesztő egyszerre több branch-en dolgozik, időbe és energiába telik a váltogatás (context switching), esetleg előfordulhat, hogy össze is keveri a branch-eket. Különösen problémás lehet egy mixed working copy esetén.

Continuous integration (CI)

A continuous integration gyakorlatilag arra épít, hogy a fejlesztési folyamatban az egyik legfájóbb pont a merge. Ezt úgy próbálja kiküszöbölni, hogy kizárólag a mainline-ban történik a fejlesztés, így nincs szükség merge-ölésre. A mainline-nak mindig egy működő, futtatható, tesztelhető állapotnak kell lennie, hogy ne akassza meg a többiek munkáját. A fejlesztés során így a fejlesztők kötelesek azonnal kommunikálni, hiszen nem vonulhatnak el külön kis branch-ükbe, hanem ha olyan funkciókat fejlesztenek, melyek közös területeket érintenek, azonnal össze kell dolgozniuk. A külön branch-ek esetén erre nincs szükség, két teljesen különböző irányokba elmehetnek, és csak a conflict során derül ki, hogy esetleg már az elején közösen kellett volna dolgozniuk.  
A CI arra ösztökéli a fejlesztőket, hogy minél gyakrabban tegyék vissza a saját gépükről a változásokat a mainline-ba (minimum naponta egyszer), megelőzve ezzel az ütközéseket, valamint kis, atomi egységekben dolgozzanak. Szemben a branch-ek esetén, ahol igen nagy méretű módosításokat kellhet összevezetni. A branch-ek használata esetén megfigyelték, hogy a fejlesztők féltek a refactoring-tól, hiszen leggyakrabban azok okoznak szemantikai ütközést. CI esetén a refactoring-ot el lehet végezni, folyamatosan javítva ezzel a kódminőséget.   
  
Branch-elés vs. CI

Mindkét megközelítésnek megvan a megfelelő alkalmazási környezete. Mint sok mindenben, itt is a projekt ismeretében lehet dönteni. Kevésbé rugalmas, vagy problémásabb megrendelő, nagyobb projekt esetében kellő körültekintéssel nem lehet probléma a branch-ek használatával. Vigyázzunk, ezek ne szaporodjanak el, kontrolláljuk őket.  
Branch esetén a merge-lésre kell nagyon oda figyelni, CI esetén viszont úgy kell fejleszteni, hogy véletlenül se tegyünk a mainline-ba olyan dolgot, ami megakasztja a többi fejlesztőt.

Köztes megközelítés

Continuous integration, a feature vagy release branch-eken kívül létezik köztes megközelítés is. Javítások, kisebb módosítások trunk-ba mennek, csak a nagyobb fejlesztések, melyek több commit-ból állnak, és a commit-ok között inkonzisztensen hagyják a szoftvert, azokat szervezik külön branch-be. Ha feature branch-t indítunk nem feltétlenül érdemes feature-önként külön branch-et indítani, megfelelő lehet az is, ha a feature-öket csoportokba rendezzük, és egy csoporthoz indítunk egy branch-et. Egy ilyen csoport kap egy verziót, amit fel lehet venni az issue tracker-be, és a feauture-öket meg a verzióhoz rendelni.  
  
Összegezve  
Tehát egy jó fejlesztési munkafolyamat leírás megmondja, hogy bizonyos helyzetekben mit kell tenni. Szabályokat kell hozni a következőkre:

* Hogyan kell szervezni a branch-eket?
* Ki hozhat létre branch-et?
* Mikor kell branch-et létrehozni?
* Honnan nyitható új branch?
* Hogyan kell a branch-eket elnevezni?
* Honnan hova történik a merge?
* Mikor, milyen gyakran történik a merge?
* Ki merge-öl?
* Mikor történik a branch törlése?
* Ki törli a branch-et?

Specifikus, hogy a branch-eket hogyan érdemes szervezni. Az alap struktúra a /trunk, /branches, /tags könyvtárak használata. Csinálhatunk külön alkönyvtárakat a release és feature branch-eknek.

A branch létrehozása történhet a projektvezető által szigorúan vezetett projekteknél. De nem mindig érdemes megszorítást adni, bizonyos projekteknél bármelyik fejlesztő nyithat új ágat.

Egy feature branch egészen kis módosítástól kezdve nagy változtatásokat is tartalmazhat. Ez a granularitás projektenként, de akár projekten belül is változhat. Történhet issue-nként, és akkor lehetőség van arra is, hogy egy release esetén össze lehessen válogatni a fejlesztéseket. Ennek előnye a maximális flexibilitás, hátránya azonban az overhead, hiszen sok branch-et kell kezelnünk. Ekkor a branch-et egy issue-val kell azonosítanunk. Másik módszer, ha egy branch több fejlesztést is tartalmaz, melyet verziószámmal tudunk azonosítani.

A mergelést érdemes minél gyakrabban elvégezni úgy, hogy még nem zavarja a fejlesztési munkafolyamatot. Ebben az esetben azonban arra is kell figyelni, hogy a branch-ben gyakran, apróbb részletekben commit-oljunk, hiszen csak ekkor garantálható, hogy a merge is kis részletekben történik. Minél ritkábban van merge, annál nagyobb az esélye, hogy a két ág szétcsúszik, és nő a conflict veszélye. Előnyös, ha egy branch-nek van egy felelőse, és ő végzi a merge-lést.

Amikor a branch-ben szereplő fejlesztés kiadásra került, a branch törölhető. Ez akár a release része is lehet. Törölheti a release, vagy a branch felelőse, vagy egy fejlesztési munkafolyamatért felelős más személy.

Git alapok konzolból

Státusz, többek között kiválasztott Branch lekérdezése:

git status

Ha van módosított fájl, de még nincs 'kiválasztva', akkor az Untracked files alatt, pirossal kiemelve látható.

Fájl kiválasztása saját adatbázisba töltéséhez:

git add teszt.txt

Több fájl esetén, az összes kijelölése:

git add \*

Egy bizonyos kiterjesztésre fókuszálás:

git add ”\*.txt”

ha ismét lekérem a státuszt, akkor a Changes to committed: alatt new fájl: mellett látható a kiválasztott állomány(ok).

Kijelölés visszavonása:

git reset

Üzenet, megjegyzés hozzáfűzése a feltöltéshez:

git commit -m ”teszt fájl hozzáadása”

A tárolóba való feltöltés előfeltétele, hogy a változtatásokat egy szöveges üzenettel kísérve jegyezzük fel.

Ezek a commitek gyakorlatilag pillanatképekként szolgálnak a tárolóról.   
Így, ha meg akarjuk tekinteni a változásokat (vagy egy másik fejlesztő) egy adott repo.-ról, akkor idővonal szerűen tájékoztatást adhat a Git az összes módosításról (log).

Repository ”idővonalának” lekérdezése:

git log

Változtatások feltöltése távoli tárolóba:

git push -u source develop , ahol a

-u kapcsoló: azt mondja meg a Git-nek, hogy emlékezzen a paraméterekre, így legközelebb elég egy git push parancsot kiadni,

source : a repo neve,

develop : branch neve.

Új fejlesztési ág (branch) létrehozása:

git branch new\_branch

(Fenti parancs kiadásakor, az éppen aktuális branch-ről készítünk másolatot.)

Branch váltás:

git checkout new\_branch

Utóbbi két lépés megoldható egy lépésben is: git checkout -b new, azaz létrehozzuk, majd át is állunk a new\_branch ágra.

Frissítés távoli tárolóból:

git pull source develop

Egy fájl törlése:

git rm teszt.txt,

Több fájl törlése:

git rm -r \*,

Egy bizonyos kiterjesztésre fókuszálás:

git rm '\*.txt'

Törlés is egyfajta módosítás amit szintén végre kell hajtani a távoli szerveren, szükséges előbb commit-olni, majd push-olni!

Branch törlése:

git branch -d new\_branch

Ágak összefésülése (merge):

Adott feladat végeztével a kivezetett ágat összefésüljük a fejlesztési ággal vagy a masterral, (Célszerű előbb egy fejlesztési ágba, majd, ha minden stimmel, nincs konfliktus, a fejlesztés célját nézve is sikeres, utána mergelni a masterbe (fő branch)).

Első lépésként át kell állni (checkout) a 'szülő’ ágra. Utána lehet a 'gyermek ágat' beolvasztása a szülőbe:

git merge new\_branch

Stash:

Új feladatot kaptunk magasabb prioritással, szorosabb határidővel és félre kell tenni a folyamatban levő munkánkat, ám az még nincs olyan állapotban, hogy egy commitojam. Ilyen esetekre jó a Stash.

Elteszem, bedobozolom a módosításokat, majd amikor vissza térek a korábbi feladatomhoz, előveszem azt.

Kirakás stash-be: git stash

Stash beolvasztása: git stash apply.

SourceTree

Jira, Confluence, BitBucket, Git mind Atlassian softwarek, ahogy az egyik legjobb Git felület a [SourceTree](https://www.sourcetreeapp.com/) is.

SourceTree egy ingyenes (accountot így is létre kell hozni 30 napon belül) Git és Mercurial kliens Windows és Mac platformra.

Hsználata

Funkciók tesztje Windows 7 alatt, egy GitHub-os repóval kommunikálva történt.

Beállítások

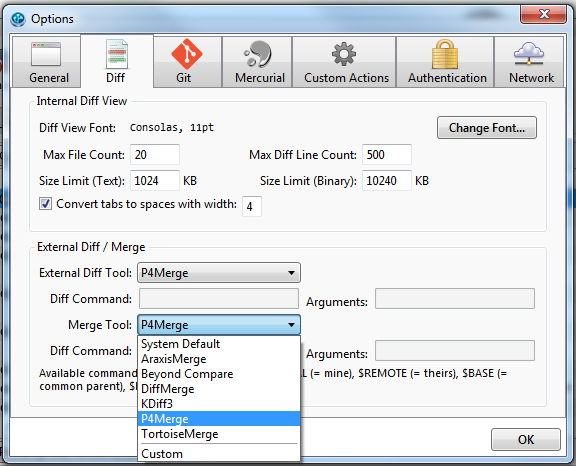
SSH kulcsok megadása: Tools → Create or Import SSH keys.

Kommitok azonosításához szükséges megadni egy Nevet és Email címet (máskülönben hibát fog jelezni a program) : Tools → Options → General fül.

Diff/Merge eszközök: fájlok összehasonlításához, mergeléséhez különböző programokat társíthatunk a SourceTree-hez: Tools → Options → Diff, External Diff Tool és Merge Tool.

Több lehetőséget is felajánl a program, bármelyiket is választjuk, használat előtte nekünk fel kell telepíteni azt.

DiffMerge, KDiff3, P4Merge eszközöket próbáltam ki,

**

[P4Merge](http://www.perforce.com/product/components/perforce-visual-merge-and-diff-tools) tűnt nekem a legszimpatikusabbnak. Átlátható és könnyen kezelhető eszköz.

### Funkciók

Clone/New

Create New Repository: új, lokális tároló létrehozása amit majd később fel lehet tölteni szerverre és szintén lehet verzió kezelni.

Add Working Copy: létező repo. hozzáadása lokális gépről.

Clone Repository: új repo. létöltése (másolat készítése) a szerverről.   
Source Path/URL mezőjébe másolom a GitHub-tól kapott urlt a projektemhez. Destination Path: ahova letölteni kívánom – üres mappának kell lennie - annak a mappának az elérési útvonalát adom meg.  
Bookmarks, azaz könyvjelzők. Amennyiben szeretném megjeleníteni a képernyő bal oldalában a példányt, abban az esetben megadhatok neki egy külön nevet a Name mezőben.

Új fejlesztési ág létrehozása (Branch)

Branches → jobb gomb → New Branch… vagy Branches menüpont a menüsorból.  
A feladat végeztével (vagy akár közben is) össze fésülhető az más ágakkal, vagy megosztható más fejlesztőkkel is - pl.: ugyanazon a feladaton többen dolgoznak -, ha feltöltjük a Remotes-ba (távoli ágak) pl.: jobb gomb a brancre, majd Push to → develop.

Ágak meg is szüntethetőek, de törlés előtt nem 'állhatunk’ rajta, branchet kell váltani.

Módosítások feltöltése (Uncommitted changes)

Új fájl jön létre, vagy változik a tartalma egy meglévőnek, esetleg törlődik, akkor az fent, a táblázat tetején Uncommitted changes címszóra kattintva, lent az Unstaged files alatt megtekinthető.   
Ha fel akarom tölteni a módosítást, akkor azt ki kell pipálni, ezután az felkerül a Staged files közé és mehet a Commit.

Előfordulhatnak majd olyan változások a könyvtárszerkezeten belül, amiket nem szeretnénk feltölteni (push-olni) – ilyenek pl.: a lokális config fileok – azokat ignorálhatjuk, ha a fájlra jobb gombbal kattintva az Ignore menüpontot választjuk.

(Látható, hogy nem csak egy fájl, de fájl típus ... stb is letiltható.)

Discart

Vissza is lehet vonni egy változtatást a Discart menüpontra kattintva, vagy a fájl(ok)ra jobb gomb->discart. Ezt követően az állományok vissza nyerik eredeti állapotukat, mintha hozzájuk se nyúltunk volna.

Commit, majd Push

Módosítások feltöltését (Push) mindig egy Commit-nak kell megelőznie.  
Miután kipipáltuk a fájlokat, fent a Clone/New menüpont mellett van a Commit, katt rá. Ezt követően megjegyzéssel láthatjuk el az aktuális feltöltést, valamint a Push changes immediately to origin/master bepipálásával commit után a program egyből fel is push-olja a fájljainkat. Egyébként külön kell a Push-ra kattintanunk.

Célszerűbb az előbbi utat választani, mert ilyenkor mi adhatjuk meg, hogy melyik Remote branch-re töltünk fel.

Merge

Egy munkafolyamat végén, a kész brancet beolvasszthatjuk (merge) a fejlesztő ágba (develop) vagy egy másik branch-be.

(Develop branch az az ág, ahova a kész módosítások bekerülnek, így, ha conflict van, nem az éles repot javítjuk.)

Ennek lépései:

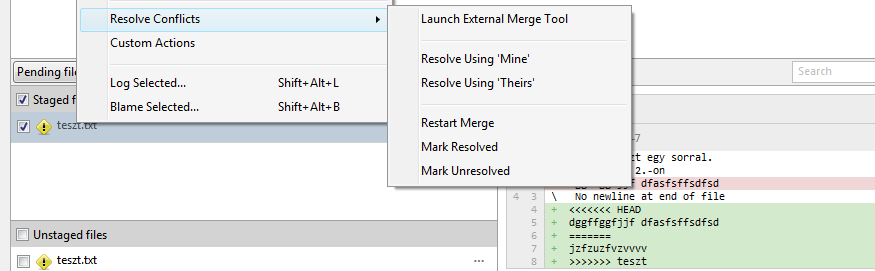
* pl.:develop branchre átállás,
* jobb gomb az ágra → Merge [branch neve] into current branch

Ütközés (conflict)

Két ág mergelése közben előfordulhatnak ütközések. Ilyenkor több lehetőség közül is választhatunk:

* Resolve Using 'Mine': az én változtatásaimat tartjuk meg,
* Resolve Using 'Theirs': adatbázisban levő tartalom marad,
* szerkesztjük a fájlt (Lunch External Merge Tool).

(Ne felejtsünk el Commitolni, majd Pusholni feloldás után!)

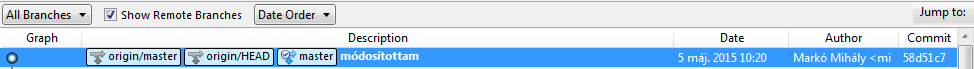


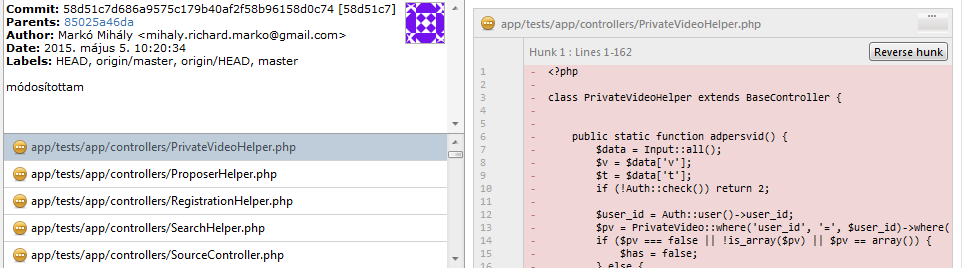
Fetch és Pull

Mielőtt folytatnánk a munkát – egy olyan Branch-en amin mások is dolgozhatnak –, célszerű egy Fetch-el indítani, hogy a módosításokat letölthessük a saját adatbázisunkba. Ha lett változtatás valamely közös fájlon, akkor

azt jelzi is a SourceTree, lehet Pullozni.

Minden változtatás az adott repón szemléletesen ábrázolva van.   
Látható, hogy ki, mikor, mit módosított, mit töltött fel.



…

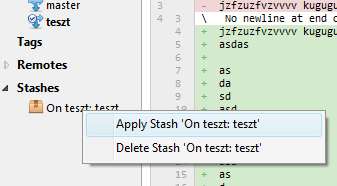
Vissza lépés egy előző verzióra

Előfordulhat olyan eset, amikor egy korábbi verzióval szeretnénk dolgozni.   
Ez úgy oldható meg, ha a gráfon rákattintunk a választott commit-ra, jobb gomb → Reset current branch to this commit.

Stash

Bedobozolás: Stash menüpont a menüsorban, adok egy nevet (cimkét) a dobozomnak és elteszem.

Kidobozolás: megfelelő branch-en kell lennem, majd a Stashes lenyílóból kiválasztom és kidobozolom az elrakott módosításokat.



GitEye

GitEye Java nyelven íródott népszerű, elosztott verziókezelő és forráskód menedzselő (SCM) rendszer.   
Teljes funkcionalitású Git kliens, grafikus felületén keresztül többek között elérhető a clon, push, fetch, pull, merge, rebase, reset stb.

Követelmények

OS: Windows, Mac, vagy Linux

32 és 64-bites verziókon egyaránt használható.

Java: Minimum követelmény Java 1.6 vagy ennél frissebb.

Javaslat: JRE Sun / Oracle vagy OpenJDK 7.

Telepítés

[GitEye](http://www.collab.net/downloads/giteye) oldaláról egy tömörített állományt tölthetünk le.  
Kicsomagolást követően használható is az alkalmazás GitEye.exe futtatásával.

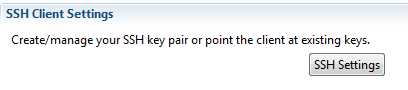
Testre szabás, Beállítások

Maga a kezelő felületet is testre szabható. Drag and drop technikával a fülek áthelyezhetőek a képernyő több pontjára is. Sőt, a fülek kiemelhetőek az alkalmazás felületéről, így több ablakos kezelést is alkalmazhatunk.

Legtöbb beállítás a Dashboard fülön keresztül elérhető kategóriákba rendezve (Git Actions, Tools) vagy a Windows menüpontról a Preferences-re kattintva.

SSH-kulcsok kezelése

Dashboard → Git Actions majd SSH Client Settingsen:

Felugró ablakon bellül a *General* fülön megadhatunk egy meglévő privát kulcsot vagy generálhatunk DSA vagy RSA algoritmussal is a *Key Management* alatt.  
  
Használata

Alkalmazás funkcionális tesztje Windows 7 rendszeren, egy GitHub-on levő teszt projekttel kommunikálva történt.

Távoli könyvtár másolása (clone)

Dashboard → Git Actions → Clone Repository

Ha URL keresztül hoznánk létre a kapcsolatot a távoli tárolóval, akkor Clone URI.  
Ezen kívül a CollabNet tovább termékeivel, GitHub-al kapcsolódhatunk össze.  
Mielőtt létre jönne a lokális Repository, több beállítást is el kell végezni:

* URL, Host, Repository path: forrás útvonalai,
* Protocol: kapcsolódás módját választhatjuk ki pl.: ssh, https,
* User, Password: távoli kapcsolathoz létrehozott felhasználó neve és jelszava,
* Destination, Directory: ahova másolni kívánjuk a távoli könyvtárszerkezetet.

Létező Git repó hozzáadása

Dashboard → Git Actions → Add Local Git Repositories

Létező Git klónokat adhatunk olvastathatunk be, kezelhetünk ezen funkción keresztül. Ehhez csak a forráskönyvtárak útvonalát kell megadni.

Új repository készítése

Dashboard → Git Actions → Create New Repository

Üres tárolót hozhatunk létre amelyet majd verzió kezelhetünk, feltölthetjük szervere, megoszthatjuk fejlesztőkkel.

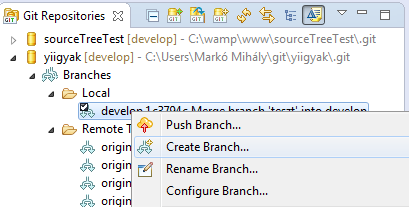
Új branch (ág)

Többféleképpen is származtathatunk új ágakat a repository-ból.

Git menüpont, Switch to .. majd a felugró ablak bal alsó sarkában a New Branch … gombra kattintva,

Git Repositories fülön belül a helyi menünk keresztül szintén szintén Switch to .. → New Branch.

Repó helyi menüjéből Create Branch:

****

Branch módosítása, törlése

Ágakat átnevezhetőek, törölhetőek ( előtte váltani kell branchet) szintén helyi menün keresztül, valamint érdemes megemlíteni, hogy módosítható a helyi remote és távoli tárolója is Configure Branch .. menüpontról.

Commit

Helyi könyvtárainkban történt változásokat a Git Files fülön belül lehet kezelni.  
Fontos, hogy minden fülön előbb frissíteni kell, majd csak ezt követően jelennek meg a módusúlt, új fájlok a Working Tree Files ablakban!

Commit előtt Drag and droppal át kell helyezni azok a Staged Changes ablakba majd a Commit Message mezőt ki kell tölteni.

Push

Commitot követően a módosításoknak még csak a saját git könyvtárunkban van nyoma. Távoli tároló tartalma pusholást követően frissül.

Pusholni lehet:

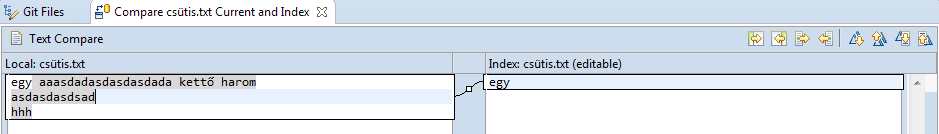
* Commit and Push: Git Files fülön belül commit és push egyszerre,
* felső menüből, ikonon lenyomásával.

Merge

Brenchek összefésülése is több módon lehetséges (nem elfelejteni, hogy annak az ágnak kell lennie aktívnak, amelybe mergelni akarunk):

* Merge lokális menün keresztül,
* felső menüből, ikonon lenyomásával.

*Merge* után *Push*-nak kell következnie!

Conflict  
Olyan ütközéseknél, melyet a program nem tud önállóan feloldani, MergeTool eszközzel nekünk kell megoldani.  
(*Working Tree Files* ablakban helyi menü a fájlon, majd *Merge Tool* menüpont.)

Jobb oldalt a kis ikonok használatával alakíthatjuk a feltöltendő fájl tartalmát:

* bal oldali verziót tartjuk meg,
* jobb oldalit,
* mind két fájl tartalmát egyaránt megtartjuk.

Fetch és Pull

Fetchel ”csak” a saját adatbázisunkba szedjük le a távoli tárolóból a változtatásokat.  
Ha egy branchbe szeretnénk letölteni, akkor még egy pull is szükséges.  
Viszont, ha egyből pullozunk, akkor az egyből elvégzi a fetchelést is.



History

Historyban idővonal szerűen láthatóak a branchen történt változások committal, szerkesztővel, módosítás idejével ... stb.

Amennyiben a szükség úgy kívánja, lehetőség van vissza lépni korábbi verzióra a Reset helyi menüpont használatával a kijelölt soron.

Stash

Bedobozolás: Stash away changes menüpont a menüsorban, adok egy nevet (cimkét) a dobozomnak és elteszem.



Kidobozolás: megfelelő Branch-en kell lennem, majd a Stashed Commits lenyílóból kiválasztom és kidobozolom az elrakott módosításokat.