Git

Git je verzovací systém, který umožňuje vzdálenou spolupráci a správu verzí upravovaného textu, lze ho používat i pouze lokálně.

První spuštění

Je nutné nastavit data o uživateli, která se použijí při zapisování revizí, a editor

```
git config --global user.name "Jan Turoň"
git config --global user.email janturon@email.cz
git config --global core.editor "C:\\Program Files\\..."
```

Volba `--global` uloží nastavení pro všechny projekty, bez něj pouze pro aktuální adresář. Nastavení lze prohlédnout příkazem `git config --list`, globální nastavení jsou ukládána do souborů `%USERPROFILE%\.gitconfig` a `MinGW64\etc\gitconfig` v adresáři programu Git.

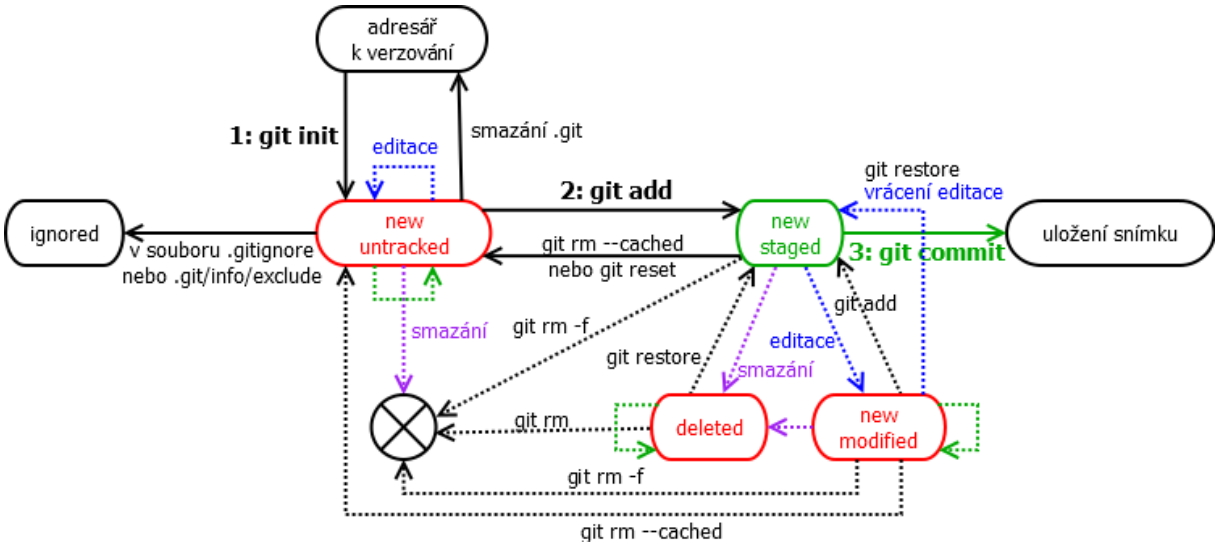
Základní pojmy

- **repozitář** je podadresář `.git`, kde jsou uchovávány informace o změnách; vytvoříme (obnovíme) jej příkazem `git init`. Od této chvíle je v terminologii gitu nazýván sledovaný adresář jako pracovní adresář (*working directory*)
- **index** je databáze sledovaných souborů; nové soubory jsou nesledované (*untracked*) a příkaz `git status` je červeně hlásí jako neznámé nové soubory. Je nutno se rozhodnout, jestli je ignorovat nebo sledovat (příkazem `git add`); index (*staging area*) uchovává informace o připravovaných změnách do dalšího snímku
- **snímek (*commit*)** se vytvoří příkazem `git commit –m "popis změn"`; obsahuje kopie změněných souborů (a odkazy na nezměněné soubory do předchozího snímku)

Vytvoření prvního snímku

1. Připravíme si adresář se soubory, které chceme sledovat
2. Vytvoříme repozitář příkazem `git init`
3. Zapneme sledování všech souborů v adresáři příkazem `git add .`; pokud některé soubory nechceme sledovat, vyjmeme je z indexu příkazem `git reset <soubor>`; Názvy těchto ignorovaných souborů (jeden na řádek, je možné používat masky) pak vložíme do souboru `.gitignore` (v případě, že jsou to dočasné soubory průběžně měněné při práci s projektem) nebo do souboru `.git/info/exclude` (v případě, že jsou to naše poznámky nesouvisející s projektem)
4. Vytvoříme první snímek příkazem `git commit –m "first commit"`; vždy uvádějte zprávu a dejte si práci s dostatečně stručným a popisným názvem změny: pokud budete chtít nějaké změny vrátit, s komentáři typu *foobar* vás čeká mravenčí práce.

Následující diagram popisuje stavy při vytváření prvního snímku. Tečkované šipky se používají zřídka, ale doporučuju je vyzkoušet k pochopení pojmu *staged*: `git status` hlásí zelený stav jako *staged* (připravený k commit) a červený jako nepotvrzenou změnu, která zůstane nevyřešená i po commit).



Pozor zejména na situaci, kdy soubor vytvoříme a naplníme, poté jej přidáme do indexu a poté jej editujeme, je ve dvou stavech: new staged (zelený) a new modified (červený). Do indexu je zapsán bez dodatečné editace a dodatečná editace zůstává jako nevyřešený problém do dalšího commit.

Kde najít pomoc

Před potvrzením změn je vždy dobré zavolat příkaz `git status`, který ukáže, co se bude zapisovat a také poradí, jakými způsoby vyřešit problémy. K tomu je dobré chápat základní pojmy.

Reference příkazů je k dispozici příkazem `git help`, případně konkrétněji např. `git add --help` (seznam parametrů), nebo podrobněji `git add --help` (kompletní manuálová stránka)

Správa lokálního adresáře – práce se změnami

Jakmile provedeme ucelenou změnu (přidání, smazání a úpravu vybraných souborů), je třeba tyto změny zapsat do indexu příkazem `git add .`, případně si před přidáním konkrétního souboru můžeme změny rekapitulovat příkazem `git add --patch <soubor>` a rozhodnout se, co dál.

Poté změny potvrdíme ji tedy příkazem `git commit -m "popis změn"`. Historii změn si lze pak prohlédnout pomocí `git log` (či kompaktním výpisem `git log --oneline`):

- hnědě je uveden SHA-1 hash změny, který slouží jako unikátní identifikátor v databázi

- zeleně je uvedena vývojová větev; při založení repozitáře se implicitně vytvoří hlavní větev master, později je možné projekt větvit a slučovat větve
- tyrkysově je uveden ukazatel HEAD, který ukazuje na aktuální verzi pracovní větve (v souboru `.git/HEAD`)

Ve většině případů probíhá vývoj posloupností editací (případně přidáním) souborů následovaných potvrzením (commit). Někdy je ale zapotřebí napravit omyly:

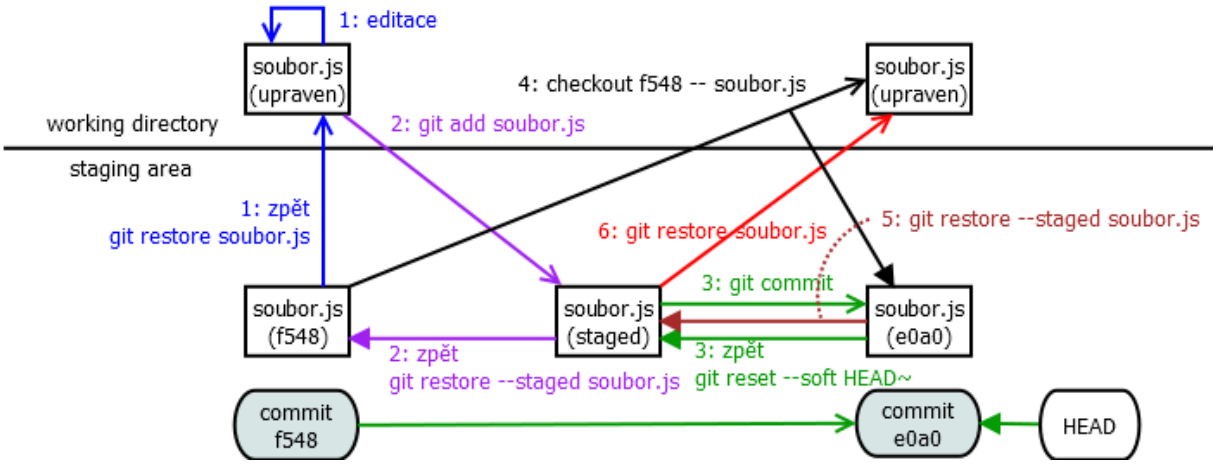
- **Právě jsme provedli commit, ale chceme to vrátit a ještě něco dodat:** commit zrušíme příkazem `git reset --soft HEAD~1`, kde `HEAD~n` znamená *verze před* `n` *kroky* (vynechání `n` dosadí `1`) a `--soft` ponechání změn v indexu i v pracovním adresáři
- **Chceme sloučit několik triviálních commitů do jednoho**: vrátíme změny v indexu, ale ponecháme změny v pracovním adresáři `git reset HEAD~n`, a změny následně zapíšeme jako jeden commit `git commit -m "souhrnný commit triviálních úprav"`.
- **Poslední commit byl slepá cesta, chceme vše vrátit**: vrátíme změny v indexu i v pracovním adresáři: `git reset --hard HEAD~`
- **Provedením** `git reset` **jsme se dostali do nechtěného bodu**: situaci zachráníme příkazem `git reflog`, který vypíše seznam všech referencí v minulosti, kde vyhledáme ID dříve odstraněné, např: `git reset --hard 651add2`

Ukažme si rozdíly na příkladě: vytvořme projekt s jediným souborem `test.txt`, zapišme do něj `one` (commit „first“) a nahraďme to obsahem `two` (commit „second“).

příkaz	obsah `test.txt`	`git status`	`git log`	`git diff --cached`
před resetem	two	nothing to commit	two first	(nic)
`git reset --soft HEAD~`	two	modified: test.txt	first	-one +two
`git reset HEAD~`	two	modified: test.txt	first	(nic)
`git reset --hard HEAD~`	one	nothing to commit	first	(nic)

Často člověk zavzpomíná ve stylu *„V minulosti jsem někde v projektu napsal takovou hezkou funkci. Pak jsem ji smazal, protože už nebyla zapotřebí, ale teď by se mi zase hodila.“* Je možné obnovit do pracovního adresáře konkrétní soubor z nějaké revize z historie. Tento krok je vhodné dělat po potvrzení změn: `git checkout <revize> -- <soubor>`.

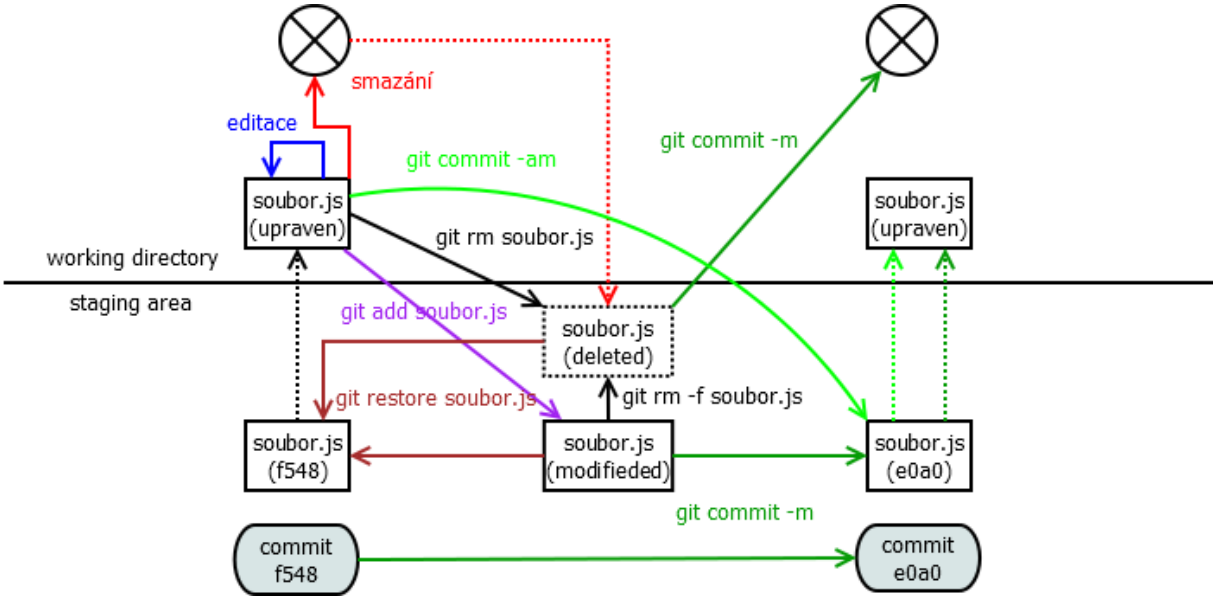
Workflow v tomto případě popisuje následující obrázek. Nad čarou ve *working directory* je to, co vidíme v pracovním adresáři, pod čarou ve *staging area* je to, co se děje v git repozitáři. Obyčejné šipky znamenají přepis, plné šipky změnu ukazatele.



1. (modré šipky) Pokud editujeme soubor, můžeme změny vrátit z aktuální revize voláním `git restore`.
2. (fialové šipky) Pokud jsme již přidali změny do následujícího volání commit, můžeme tento krok odvolat příkazem `git restore --staged`, následným voláním `git restore` se vrátíme k původnímu souboru.
3. (zelené šipky) Voláním `git commit` vytvoříme novou revizi a HEAD se přesune k ní. Commit lze odvolat voláním `git reset` s uvedenými parametry.
4. Pokud se chceme vrátit k souboru z minulé revize (nemusí být bezprostředně předcházející), zajistíme to voláním `git checkout` s uvedenými parametry. Změny se promítnou do indexu i do pracovního adresáře.
5. (hnědá a červená šipka) Pokud chceme vrácení pouze u tohoto souboru zvrátit, uděláme to jako v kroku 2: `git restore --staged` přesune ukazatel na posledně uloženou verzi v repozitáři a následné volání `git restore` tento soubor promítne do pracovního adresáře.

Přejmenovat soubory není praktické v systému, jelikož se tato informace nedostane gitu a ten tak zobrazí původní soubor jako smazaný a nový soubor jako přidaný. Změny je pak nutné zapsat také do indexu příkazy `git rm původní` a `git add nový`. Je lepší proto volat příkaz `git mv původní nový`, který též přejmenuje soubor v systému.

Ukažme si ještě situaci s mazáním souboru (tečkované šipky zde značí automatický přechod)



Lokální adresář – základní shrnutí

Při lokální práci s gitem tedy postupujeme nejčastěji:

1. `git init` v adresáři sledování změn, poté `git add .` a `git commit -m "initial"`
2. editace, `git commit -am "popis"`
3. zobrazení změn `git status`, minulé commity `git log --oneline`

Správa vzdáleného adresáře

Většina lidí pracující s gitem využívá služeb veřejného git serveru github.com. Při komunikaci s tímto serverem se používá protokol ssh a k jeho použití je nutno vygenerovat veřejný a privátní klíč (pod Windows nejlépe nástrojem puttygen). Veřejný klíč v openSSH formátu (začínající na ssh-rsa) zkopírujeme na github server (ikona profilu > Settings > SSH and GPG keys) a pro jistotu uložíme též do souboru mykey.pub. Uložíme si též privátní klíč (v puttygen Conversions > Export OpenSSH key) do souboru mykey (bez přípony). Dvojice těchto klíčů se ukládá (pod Windows i na Linuxu) do podadresáře .ssh v domovském adresáři, kde se též ukládají veřejné klíče vzdálených serverů do souboru known_hosts.

Po tomto nastavení SSH komunikace začneme nejsnáze vytvořením vzdáleného (tj. na githubu uloženého) repozitáře (ikona profilu > Your repositories > New). Bývá zvykem při inicializaci nastavit soubor README.md, který využívá Markdown značkovací formátování (používaný též k editaci wikipedie, na stackoverflow a dalších serverech, doporučuju studenty naučit) – zde můžeme zapsat stručné představení repozitáře příchozím. Po vytvoření repozitáře vám webové rozhraní ukáže SSH adresu (končící na .git) a návody, jak vzdálený repozitář propojit s lokálním.

Nejjednodušší je na lokálním počítači v nějaké složce otevřít příkazovou řádku a napsat příkaz `git clone vzdálený-soubor.git`, který vytvoří link na vzdálený repozitář pod názvem **origin**.

Před započetím práce pak zavoláme příkaz `git pull`, který z origin stáhne aktuální verzi našich sdílených souborů (zatím s repozitářem pracujeme sami, tak to není nutné, ale je dobré si na toto zvyknout, jelikož pak obvykle odpadají starosti se slučováním nekompatibilních změn). Poté provedeme práci na lokálním repozitáři, při které není třeba připojení k internetu (můžeme provést několik commitů). Poté provedeme příkaz `git push`, který změny nahraje do origin.

Pokud jsme ale začali tvorbou lokálního repozitáře a na vzdáleném serveru jsme pak vytvořili vzdálený repozitář, je možné je propojit příkazem `git remote add origin vzdálený-soubor.git`. Poté provedeme příkaz `git push –u origin master`, který nastaví směr nahrávání z vaší hlavní větve (**master**, o větvích za chvíli) do vzdáleného repozitáře **origin**. Tento směr je zapamatován, příště pak stačí volat prosté `git push`.

Vzdálená spolupráce

Nyní můžeme přizvat ke spolupráci další lidi nastavením ve webovém rozhraní repozitáře v menu Settings > Collaborators. Kdokoliv může stáhnout obsah vašeho gitu (případně si ho zkopírovat na vlastní github účet kliknutím na Fork), jeho použití můžete upravit přidáním licenčního souboru LICENSE.md (který si lze předgenerovat v githubu v menu Code > LICENSE.

Lidé, které přidáte jako spolupracovníky, mohou také příkazem `git push` provádět změny do vašeho repozitáře. Při práci se studenty je tak vhodné vytvořit si vlastní repozitář se zadáním a dát studentům SSH git adresu pro `git clone`. Poté studenti mohou založit vlastní repozitář s řešením a pozvat vás jako spolupracovníka (tj. můžete editovat a komentovat jejich práci). Vytvořte si pak adresář a do něj do podadresářů jednoho studenta po druhém naklonujte (adresáře jednotlivých studentských projektů průběžně přejmenovávejte dle jejich příjmení, abyste se v nich vyznali).

Studenti si pak ve chvíli volna stáhnou aktualizace vašeho zadání, pracují na řešení a nahrají ho do svého repozitáře. My si pak můžeme napsat skript, který stáhne všechny studenty najednou:

pulall.bat

```
@echo off

for /D %%i in (*) do (
  cd %%i
  cd
  git pull
  cd..
)
```

Dále si připravíme skript, který otevře všechny práce najednou: soubor s prací studenta a soubor README.md, kde můžeme studentovi napsat komentáře. V Případě práce s textovými soubory skript v mém případě vypadá následovně:

openall.bat

```
@echo off

for /D %%i in (*) do (
  cd %%i
  cd
  D:\ProgramsWin\PSPad\pspad.exe "%1"
  D:\ProgramsWin\PSPad\pspad.exe README.md
  cd..
)
```

Studentům připravím zadání do souboru např. `task01.cpp`, který může vypadat následovně:

task01.cpp

```
#include <cstdio>

char* toUpper(char* input) {
      // TO DO
}

int main() {
      char input[] = "Hello, World!";
      puts(toUpper(input)); // should print HELLO, WORLD!
}
```

Tento soubor si zkopírují do svého řešení, provedou změny dle komentářů a nahrají jej do svého repozitáře. Já pak zavolám `openall task01`, který v PSPadu otevře po dvojicích řešení a soubor README.md příslušného studenta. Po dvojicích pak opravím `task01`, zapíšu komentáře do `README.md` a zavřu dvojici souborů. Po zpracování všech souborů pak spustím následující skript, který mé změny nahraje studentům do jejich repozitářů:

pushall.bat

```
@echo off

for /D %%i in (*) do (
  cd %%i
  cd
  git push
  cd..
)
```

Efektivní práce se studenty

Studenti od sebe pochopitelně tímto způsobem mohou opisovat. Mohou se i pozvat jako spolupracovníci a vzdáleně si pomáhat, zdatnější si dokonce mohou napsat skript, který automaticky pravidelně kopíruje práci spolužáka a provádí kosmetické změny, aby se na opisování nepřišlo.

Toto není špatně. Opisování není neřest, je to ctnost. Pokud v praxi děláte něco, co jste mohli zkopírovat, plýtváte zdroji a nerudný manažer vás za to může oprávněně vyhodit.

Neměli bychom proto ve studentech rozvíjet škodlivý návyk, že opisování je špatné, ale naopak: opiš, co se dá (dle licence). Tento zdravý návyk je akcelerace úspěchu.

Zpočátku je vhodné dávat úkoly, kde se dá opisováním učit. Je třeba studentům vysvětlit zásadu *opisuj pouze to, čemu rozumíš*. A dále *vše opisuj ručně (a pokud možno svými slovy), aby sis to zapamatoval, nepoužívej Ctrl+C, Ctrl+V*.

Později, když si studenti osvojí základy, jim je třeba dávat kreativní úkoly, kde je třeba vymyslet postup a kde je mnoho korektních postupů. Pokud studenti nedokáží být v něčem kreativní, není třeba je torpédovat pětkami, stačí jim taktně poradit, aby dále pátrali po činnostech, ve kterých jsou přirozeně kreativní a připravit pro ně alternativní náplň zaměřenou nikoliv na aplikaci znalostí, ale na získání podvědomí, o čemže zhruba předmět je, a domluvit se na nějaké kompromisní známce.

Konflikt lokální a vzdálené verze

Snadno může dojít k následujícímu scénáři:

1. já i spolupracovník provedeme `git pull`
2. spolupracovník upraví soubor.txt provede `git commit` a `git push`
3. já upravím soubor.txt jinak, provedu `git commit`

Pokud nyní zavolám `git push`, dostanu chybu [rejected]. Příkazem bych nechtěně přepsal změny spolupracovníka. Situaci vyřeší `git pull`, který vygeneruje hlášku o tom, ve kterých souborech nastal konflikt a provede jejich automatické sloučení (merge). Situace pak vypadá následovně:

původní soubor.txt

```
first line
second line
third line
```

soubor.txt po úpravách spolupracovníka

```
first line
2nd line
third line
```

soubor.txt po mých úpravách

```
first line
second edited line
third line
```

Provádím příkaz `git push`

```
! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'git@github.com:janturon/test.git'
```

Provádím příkaz `git pull`

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

můj lokální soubor.txt nyní

```
first line
<<<<<<< HEAD
second edited line
=======
2nd line
>>>>>>> de9307b7fb27a9d52778ff79446b162f63c02d1f
third line
```

Ručně nahradím obsah mezi <<<<<<< a >>>>>>>, provedu `git commit -am "merged conflict"` a následný `git push` proběhne v pořádku.

Práce s větvemi

Občas je žádoucí pokračovat ve vývoji různými směry a pak je opět sloučit do jednoho. U studentů můžeme použít větve pro práci více studentů na jedné úloze. Při vytvoření repozitáře se automaticky vytvoří větev **master**. Větev není nic více, než pojmenovaný odkaz na revizi. Vykonáme-li commit, aktivní větev se přesune na nově vzniklou revizi. Aktivní větev je označena ukazatelem **HEAD**. Ten můžeme přesunout na libovolnou revizi v historii příkazem `git checkout <hash>`. (Hashe historických verzí získáme příkazem `git reflog`.) Tím se aktualizuje pracovní adresář, ale nemůžeme provádět žádné změny, protože HEAD neukazuje na žádnou větev (stav oddělené hlavy, *detached HEAD*). Je to užitečný příkaz k nahlédnutí do historie, zpět se vrátíme příkazem `git switch -`.

```
HEAD is now at 28c2633 start

d:\Projects\Git\test03>git log --oneline
28c2633 (HEAD) start

d:\Projects\Git\test03>git branch tmp

d:\Projects\Git\test03>git log --oneline
28c2633 (HEAD, tmp) start

d:\Projects\Git\test03>git checkout tmp
Switched to branch 'tmp'

d:\Projects\Git\test03>git log --oneline
28c2633 (HEAD -> tmp) start
```

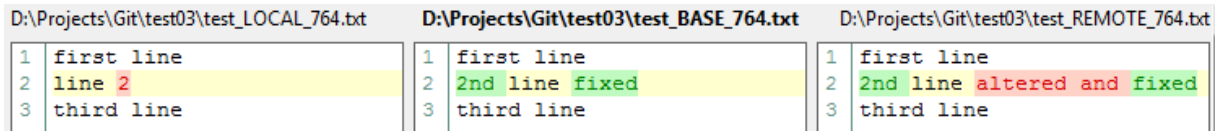
Pokud se chceme vrátit do historie a založit odtamtud novou větev, provedeme to příkazem `git checkout <hash> -b <větev>`, který je kombinace příkazů `git checkout <hash>`, `git branch <větev>` a `git checkout <větev>`. Neaktivní větev lze kdykoliv smazat příkazem `git branch -d <větev>`. Pokud v každé větvi provádíme změny v různých souborech (nebo alespoň na různých místech souborů), lze sloučení s hlavní větví provést příkazy `git checkout master` (přesunutí HEAD do hlavní větve) a `git merge <větev>` (sloučení s hlavní větví).

Pokud sloučení nelze provést jednoznačně (v každé větvi jsme stejné místo změnili jinak), příkaz pro sloučení vyhodí hlášku zmiňující soubory, ve kterých je konflikt a které nyní obsahují poznámky pro sloučení. Příkaz `git status` u takových souborů vypáše červený záznam `both modified`. Pokud je konfliktů mnoho, je vhodné použít specializovaný editor

pro slučování souborů. Multiplatformní volný DiffMerge nakonfigurujeme pro práci s gitem následujícími příkazy:

```
git config --global merge.tool diffmerge
git config --global mergetool.diffmerge.trustExitCode true
git config --global mergetool.diffmerge.cmd
"D:\\Cesta\\k\\programu\\sgm.exe -merge -result=\"$MERGED\" \"$LOCAL\"
\"$BASE\" \"$REMOTE\""
```

Pro dokončení slučování pak použijeme příkaz `git mergetool`, který by nyní měl spustit DiffMerge, v mém případě s následujícím rozhraním:



Vlevo je větev, do které slučujeme (LOCAL, v mém případě master), vpravo napojovaná větev (REMOTE, v mém případě tmp) a uprostřed editorem navrhované sloučení: červeně jsou mazané části a zeleně přidávané části. Prostřední soubor můžeme doupravit upravit ručně, po uložení je konflikt vyřešen a upravený soubor (BASE) je přidán k připravovaným změnám (zelený v `git status`), přičemž původní soubor (s poznámkami, před úpravou DiffMerge) je uložen s doplněnou příponou .orig). Tyto soubory můžeme smazat, nebo vložit do souboru .gitignore.

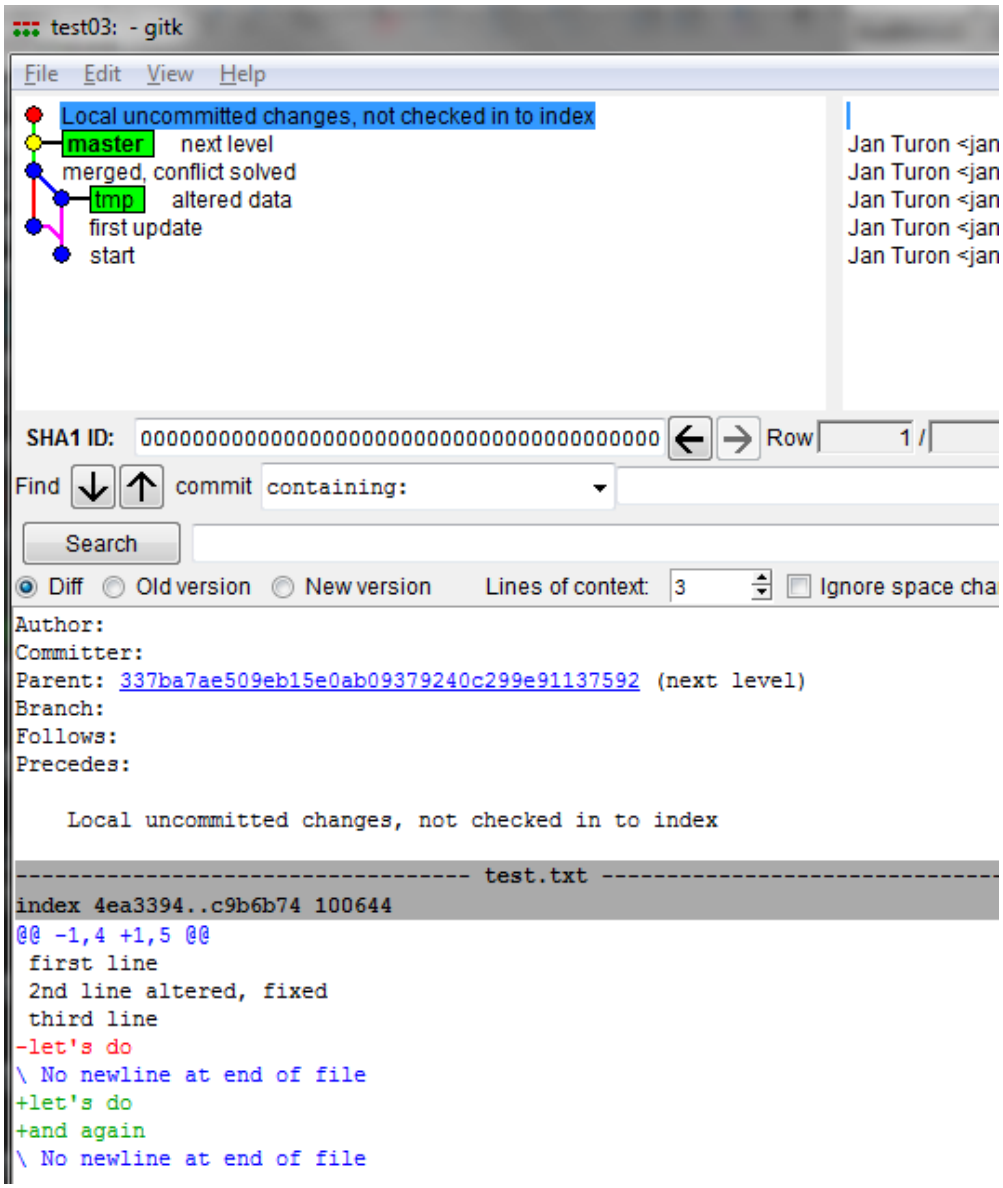
Sloučená revize má pak dva rodiče, můžeme ověřit příkazem `git log --oneline --graph`:

```
d:\Projects\Git\test03>git log --oneline --graph
*   f8feead (HEAD -> master) merged, conflict solved
|\
| * 0904ba0 (tmp) altered data
* | 40b632b first update
|/
* 28c2633 start
```

Nyní už není důvod pro větev tmp, můžeme ji smazat příkazem `git branch -d tmp`.

Kdo kdy co upravil

Prozkoumat historii souboru lze na konzoli příkazem `git log -p <soubor>`, přehlednější je však použít integrovaný gitk příkazem `gitk <soubor>`, kde si lze pohodlně překlikávat mezi revizemi:



z obrázku také plyne, jak moc mohou vhodné komentáře při odesílání změn usnadnit práci při hledání konkrétní změny.

Vymazání historie

V případě, že historie už není třeba, je možné smazat její záznamy (lokálně i vzdáleně) následující posloupností příkazů:

1. Založením sirotčí větve a přepnutím na ni `git checkout --orphan tmp`
2. Přidáním všech souborů `git add -A`
3. Potvrzení změn `git commit -m "first commit"`
4. Smazáním hlavní větve `git branch -D master`
5. Přejmenováním sirotčí větve na master: `git branch -m master`
6. Zapsáním změn na vzdálený repozitář `git push -f origin master`

Pořád zůstává viditelná historie změn příkazem `git reflog`. Tu odstraníme vymazáním obsahu adresáře .git/logs a následným voláním `git gc --prune=now`. Zaručenější způsob je pak prosté smazání celého .git adresáře a opětovné naklonování ze vzdáleného repozitáře.