Git: Distribuovaný verzovací systém

Git je nejpoužívanější distribuovaný systém správy verzí (Version Control System - VCS). Další je Mercurial.

Git znamená v angličtině hajzlík, šmejd. Linus Torvalds jednou na blogu napsal, že všechny své výtvory pojmenovává podle sebe. Git byl vyvinut pro vývoj Linuxu. Ten je komunitní, proto potřebuje distribuovaný verzovací systém. Git i Mercurial vznikly roku 2005, oba jsou dostupné pod licencí GNU GPL.

Dříve se používaly centralizované verzovací systémy, např. Subversion a SVN. Jejich nevýhodou je, že v době nedostupnosti serveru nemůže nikdo aktualizovat. A v případě havárie centrálního počítače zůstane programátorům jen verze, na které právě pracovali (pokud si nedělali zálohy).

V centralizovaném verzovacím systému je na serveru vše, programátor si odtud zkopíruje jen verzi, na které právě pracuje. U distribuovaného systému má každý programátor na svém stroji vše (tedy i staré verze), tyto stroje se pak navzájem synchronizují. Každá lokální repository je tak kompletní zálohou.

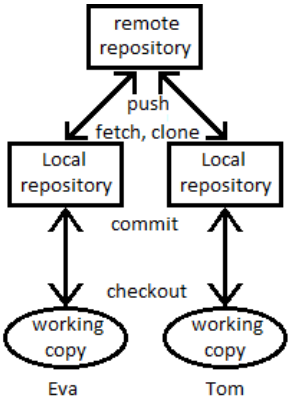
Pozn.: Komunitní vývoj Linuxu probíhal od roku 2002 na distribuovaném VCS BitKeeper. Později byl zpoplatněn, proto si Linux Torvalds musel napsat svůj.

Distribuovaný verzovací systém se rozšířil až díky dostupnosti velkých disků na stanicích. Protože zde má každý programátor u sebe vše. Může kdykoliv začít pracovat na jiné větvi programu, aniž by si tuto větev musel stáhnout z centrální repository. Není tedy závislý na připojení k internetu, nemá-li pár dní připojení, tak prostě jen odloží synchronizaci.

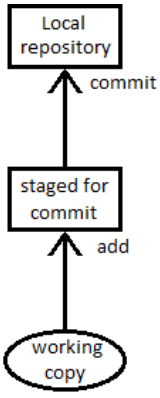
I když byla řeč o velkých discích, tak to neznamená, že by se při každém kopírování verze do repository kopíroval celý obsah souboru. Každý soubor se uloží jen poprvé, a potom se ukládají jen tzv. snapshoty.

Dřívější VCS ukládaly na poprvé celý soubor a poté jen rozdíly (tedy např. jen pár řádků). To zabralo nejméně místa, ale při velkém množství změn chvíli trvalo, než se soubor zrekonstruoval. Systém Git byl navržen pro co nejvyšší rychlost, méně již záleží na objemu na disku (ceny disků zatím klesly, kapacity se zvýšily). Proto Git ukládá tzv. snapshot, což je u změněných souborů celý soubor, u nezměněných souborů (kterých bývá většina) jen odkaz na poslední uložení.

Struktura uložišť při práci v týmu



Přesněji: commit má dvě etapy:



U distribuovaného systému má každý programátor programy uložené na čtyřech místech.

-**working copy**: verze programu, na které právě pracuje. To je tedy složka s programem, na které programátor pracoval předtím, než si vůbec nějaký git nainstaloval.

-**local repository**: toto úložiště má programátor na svém počítači. Vznikne prvním použitím příkazu git commit. Tedy prvním uložením (initial commit) první verze daného programu do GITu. Nebo klonováním cizího programu z remote repository. Od té doby tam programátor (po řádném otestování) posílá (pomocí dalších commit) všechny další verze. Už tato lokální repository programátorovi umožňuje pracovat s verzemi. Ne ještě práci v týmu, k tomu potřebuje uložit svou práci na společné remote repository.

-**remote repository**: Jakmile chce programátor svůj program zpřístupnit ostatním, tak vytvoří někde na cloudu (např. GitHub či BitBucket) prostor a zkopíruje tam příkazem push celou svou lokální repository (včetně všech verzí). A nastaví práva k této remote repository. Například všem jen čtení, pokud chce například vystavit své

výtvory v rámci svého portfolia (tedy aby umožnil personalistům firem posoudit, jak hodně je dobrý). Nebo umožní členům svého týmu i zápis. Práci v týmu budeme probírat na závěr těchto skript.

-**staging area:** Nelze ve skutečnosti provést přímo commit z working copy do Local repository. A to proto, že ne všechny soubory ve working copy se přenášejí do repository (například to ovlivňuje soubor .gitignore, viz níže). Takže pokud se objeví ve working copy nový soubor, je nutné jej příkazem **git add** přidat mezi soubory, o které se git stará. Tyto soubory jsou ve virtuální složce, jakémsi meziskladu, kde jsou soubory vystaveny pro commit (staged for commit, nebo také Staging area), tedy určené k dalšímu zpracování. Obsah složky bude v dalším commitu přenesen do lokálního repositáře. Nicméně i později nestačí pro zahrnutí do commitu to, že byl soubor od posledního commitu změněn. Znovu je nutno předtím použít pro něj příkaz **git add.**

Shrnutí: working copy je od počátku, staging area a local repository zajistí Git, remote repository získáme např. na GitHubu.

Poznámky:

Každý změněný soubor může být ve třech stavech: modified, staged (připraven pro zapsání do lokální repository), committed (zapsán).

Pozor: příkaz git add by se měl použít těsně před příkazem commit. Pokud by mezi nimi byla prodleva a nějaký soubor by se mezitím změnil, pak by se do lokální repository poslala neaktuální verze. Takže musím po skončení modifikace dát ještě jednou add.

Working copy je adresář, kde jsou čitelné verze souborů. Naopak local repository je samozřejmě komprimovaný, je ve složce .git. Staging area je jeden komprimovaný soubor, obvykle také ve složce .git.

Složku .git si můžeme vzít na USB disk a máme vše, co potřebujeme pro práci tam, kde není internet. Ale lépe i celé working copy, to pro případ, že by se nějaké soubory z working copy neposílaly na local repository.

Instalace Git

Na Google hledáme "Git client download", najdeme např. https://git-scm.com/downloads

Zvolíme stažení klienta pro Windows. Po stažení z místní nabídky staženého souboru vybereme *Run as administrator*. Pokud by se spustil pod běžným uživatelem (tedy 2xclick v Průzkumníku), tak už se neobjeví výzva k dodatečnému poskytnutí práv do složky Program files a proto by se to nainstalovalo jen pro daného uživatele (jiní uživatelé počítače by Git neměli). Ve všech krocích průvodce pak potvrdíme standardní nastavení.

Při instalaci gitu je výběr (radiovými tlačítky) mezi:*Use git from Git Bash only* a *Use git from the Windows Command prompt.* Druhá volba je zvolena. To znamená, že příkazy gitu bude možné používat i v Command promptu.

Pozn.: V okně gitu je na rozdíl od Command promptu možno písmo zvětšovat standardním způsobem pomocí CTRL+kolečko na myši. Navíc na dotykové obrazovce i roztahováním dvou prstů.

Po dokončení instalace pod Administátorem se v Program files objeví složka Git (pokud instalujeme pod běžným kontem, např. u nás a učebně pod kontem student, tak se objeví v Users\student\AppData\Local\Programs).

V Průzkumníku či v Total Commanderu se v místní nabídce složek od této chvíle navíc objeví položky *Git GUI here* a *Git Bash here*. V tuto chvíli již můžeme používat git pro uchovávání verzí. Musíme ale sdělit Gitu, soubory v které složce má sledovat, tedy která složka je working copy. To provedeme tak, že v tomto adresáři zadáme v prostředí Git bash příkaz **git init.** Tím se vytvoří podsložka **.**git, ve které bude lokální repository. A pak po změnách budeme z working copy kopírovat do lokální repository příkazem **git commit**, případně naopak stahovat do working copy nějakou verzi pomocí **git checkout**.

Pro týmovou práci se však ještě potřebujeme připojit k některému veřejnému Remote repository. Některé jsou placené, jiné zdarma. Nejpoužívanější se jmenuje Github, jeho nevýhodou ale je, že zde vše je veřejné, hodí se proto jen pro opensource vývoj. Lepší je BitBucket (doslova kbelík na bity), který je zdarma, ale k vašemu kódu se nikdo jiný nedostane. Projekt lze nasdílet pro pět lidí, pro více to už je placené. BitBucket spolupracuje s Git i Mercurial, GitHub jen s Git (jak už je patrné i z názvu).

Doplňkové nastavení identity

Ještě nastavíme identitu, následující obrázek je example z gitu v situaci, kdy jsme volbu *Git bash here* zvolili z místní nabídky kořene disku e:

```
lenovo1+student@lenovo1 MINGW64 /e
$ git config --global user.name "novak"
```

```
git config --global user.name "Cervenka"
git config --global user.email "cervenka@vsb.cz"
```

Každý commit pak bude tímto jménem a emailem podepsán. Kdybychom nezadali, tak by bylo u commitu uvedeno jméno konta, se kterým se uživatel hlásí do Windows, takže by u všech účastníků kurzu bylo konto stejné, např. Student. A email by byl (na našich učebnách na VŠB) vždy Student@dom890.vsb.cz. to by dělalo problémy v situaci, kdy by s remote repository pracovalo více studentů, nebylo by možné rozlišit, který commit kdo provedl.

V novějších verzích si již git zadání těchto parametrů vynutí. Po zadání prvního commitu by se objevilo: *** Please tell me who you are. A rada, ať spustíme ty dva výše uvedené příkazy.

Nastavení alternativního editoru

Jako standardní editor je v prostředí Git bash here nastaven linuxový editor vim. Práce s ním není moc intuitivní. Je proto nepříjemné, když se nečekaně otevře v situaci, kdy to nečekáme a my nevíme, jak dál. Editor se otevře např. v situaci, kdy na konec příkazu git commit zapomeneme udat název commitu. Editor nám má umožnit jeho dodatečné zadání. Pokud by se to někdy stalo, tak jak editor zavřít: stisknout klávesu ESC, pak dvojtečku a potom písmeno q.

Nastavíme proto místo linuxového editoru vim jako implicitní editor Windowsovský jednoduchý editor Notepad:

```
git config --global core.editor "notepad"
```

GitHub: vytvoření repository

GitHub je zdaleka nejrozšířenější server pro hosting projektů. Další jsou například: BitBucket, GitLab, SourceForge. GitHub vznikl v roce 2008, roku 2018 jej koupil Microsoft.

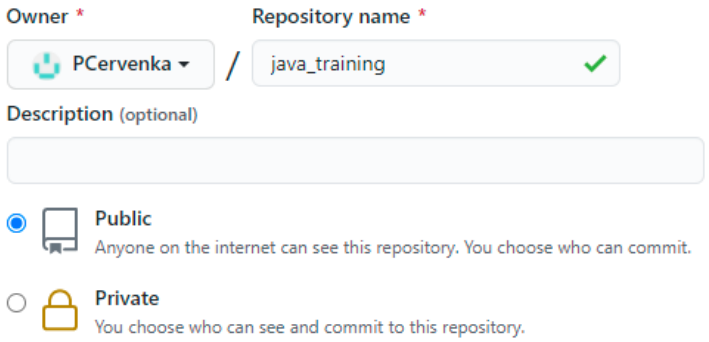
Tvorbu svého prostoru na GitHubu začneme registrací. Na github.com klepneme na tlačítko *Sign up for GitHub*.

Vyplníme *Username*, *Email address* a *Password*. Prokážeme, že nejsme robot – vybereme správný obrázek podle zadání. Poté klepneme na tlačítko *Create Account*. GitHub zašle mail na udanou adresu, aby se přesvědčil, že si nevymýšlíme. Otevřeme mail, klepneme na tlačítko *Verify email address*. A profil je hotov.

Poté si vytvoříme první repository. Přepneme se na kartu *Repositories* a klepneme na tlačítko *New*. Objeví se stránka s výběrem tří možností, vybereme *Create a repository*.

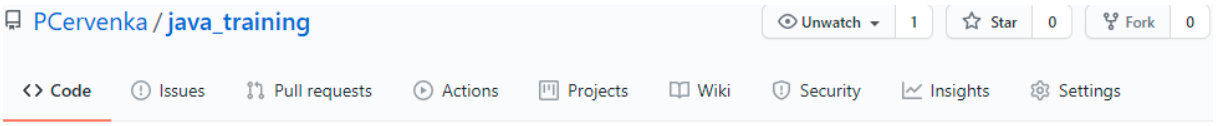
Objeví se formulář, kde vyplníme *Repository name* (např. java-training). Dále vybereme, zda přístup k repository bude *Public* nebo *Private*. Při veřejném přístupu mohou všichni obsah repository číst, můžeme však upravit, kdo má právo zápisu příkazem push (provedeme to pak pomocí *Settings/Manage access/Invite a collaborator*). Při přístupu Private můžeme nastavit, kdo má právo čtení a kdo právo zápisu.

Vybereme Public, protože budeme chtít GitHub použít i pro vystavení zdrojových kódů svého portfolia.



Ještě můžeme vybrat vložení tří textových souborů do repository, toto nevyužijeme.

Klepneme na tlačítko *Create repository*. Repository se vytvoří a přejdeme do něj. Na webové stránce je několik karet, vybrána je karta < >*Code*:

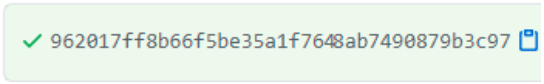


Teď bychom mohli do repository pomocí příkazu push něco nahrát.

Vytvoření tokenu

Ještě to odložíme. Náš profil je typu Public, takže kdokoliv z něj může číst. Ale při zápisu je to jiné, musíme prokázat, že jsme to my. Takže při každém příkazu push bychom museli zadat jméno (zde PCervenka) a heslo (to, které jsme zadali při tvorbě účtu). Což by bylo dost otravné. Ale jde to i jinak: necháme si vygenerovat jakýsi průvodní list, zvaný token, který se uloží na daném stroji v profilu daného uživatele. A při každém příkazu push si to náš stroj a GitHub mezi sebou vyřikají a my se nemusíme nijak prokazovat.

Token se tvoří na kartě s naším osobním nastavením. Je to karta *Settings*, zcela vpravo. V její levé části je menu, jedna z posledních položek je *Developer Settings*. Z podnabídky vybereme *Personal access tokens* a klepneme na tlačítko *Generate new token*. Objeví se formulář, kde token nějak popíšeme (protože tokenů můžeme mít více, pro různé účely) a vybereme, k čemu všemu má token poskytovat přístup. Pokud chceme používat příkaz push v režimu příkazového řádku, tak zatrhneme skupiny repo, admin:repo_hook a delete_repo. Poté klepneme na tlačítko *Generate token*. Token se zobrazí.



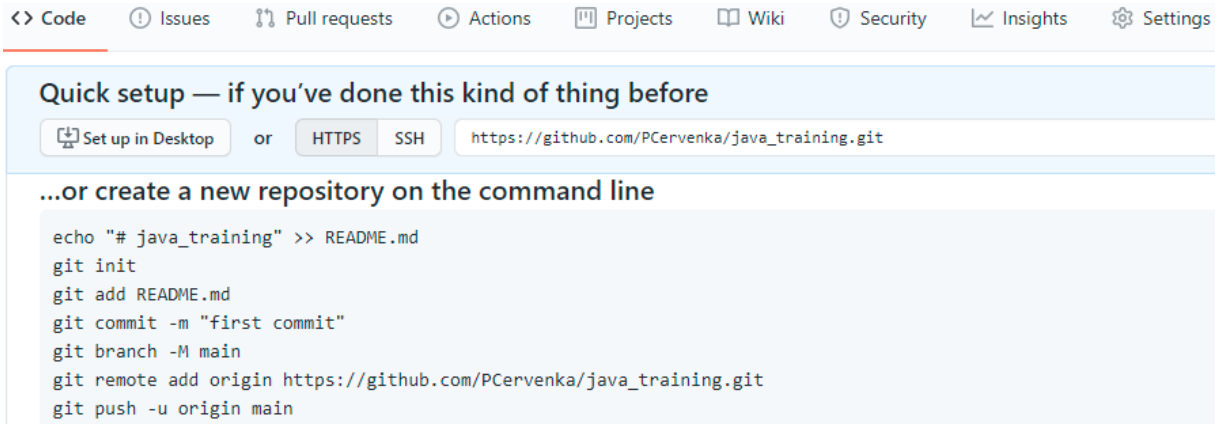
Zkopírujeme si jej do schránky (a pro jistotu i do nějakého dokumentu). Nezapomenout, protože po opuštění stránky se již token nezobrazí.

Pozn.: Token platí jen pro daný stroj. Pokud bychom někdy chtěli použít příkaz push z jiného stroje, tak nám GitHub pošle mail s šestimístným verifikačním kódem a musíme si vygenerovat nový token.

Zatím token odložíme, budeme ho potřebovat později, při prvním příkazu push.

Provázání lokální a remote repository, první comit

Nyní musíme tuto novou remote repository provázat se svou lokální repository. Na panelu <>*Code*> k tomu máme návod:



Příkazy si mírně upravím. Místo souboru Readme.md vytvořím soubor Contributors.txt. A to proto, že vytvoření tohoto souboru bylo v návodu pro začátek práce se serverem BitBucket, který jsme používali dříve. A tento název je uveden i ve všech obrázcích v dalších částech skript, tak ať je nemusím všechny předělávat. A do souboru vložím jméno uživatele.

A také není nutné provést příkaz git branch –M main. Tímto příkazem se hlavní větem přejmenovává ze standardního master na main. V duchu politické hyperkorektnosti, protože název master by se mohl někoho dotknout. Mohl by mu připomenout spojení dvou slov master – slave. Pokud přejmenování neprovedeme, tak v posledním příkazu neuvedeme slovo main ale master. Následuje upravený seznam příkazů:

```
echo "Cervenka" >> contributors.txt
git init
git add contributors.txt
git commit -m 'Initial commit with contributors'
git remote add origin https://github.com/PCervenka/java_training.git
git push -u origin master
```

Tyto příkazy vykonáme na svém stroji, a to v příkazovém řádku v adresáři, kde máme working copy, v mém případě to je složka work na disku E:. Neprovedeme to však v klasickém Command promptu, protože Git poskytuje vylepšený příkazový řádek. V Průzkumníku klepneme pravým tlačítkem myši na složku work a z místní nabídky vybereme *Git Bash Here*. Tím "Here" se myslí, že daná složka se stane přitom aktuální. Spustí se černé okno příkazového řádku, je to ale příkazový řádek Linuxu. Používá pro lepší rozlišení různé barvy a má také pár příkazů navíc. A reaguje na zvětšování poocí CTRL+kolečko myši či na dotykové obrazovce na roztahování dvěma prsty. Nevychází z syntaxe MS-DOS, ale z Linuxu. Takže prompt je místo `E:\work>` takovýto:

```
student@Lenovo2 MINGW64 /e/work
$
```

Tedy kromě disku a adresáře ukazuje i jméno uživatele a stroje a prompt je dolar místo >. A Linux nezná disky, takže disk je pro něj vlastně adresář, takže za "e" není dvojtečka ale lomítko. A lomítko je i na začátku, aby bylo jasné, že to není relativní cesta. Pozor také na to, že Linux je case sensitive. Takže pokud by nějaký název začínal na velké písmeno, pak je třeba to respektovat.

Pozn.: MINGW64 je zkratka z Minimalist GNU for Windows, což je název programu, který poskytuje toto prostředí umožňující použití minimalistické (ořezané) sady linuxových příkazů pod Windows.

Nyní tedy výše uvedené příkazy zadáme. Můžeme je napsat na klávesnici nebo zkopírovat z návodu na kartě <>*Code* a pak vložit do příkazového řádku klepnutím kolečkem myši (určitě to tak uděláme minimálně s tím nejdelším příkazem)

```
student@Lenovo2 MINGW64 /e/work
$ echo "Cervenka" >> contributors.txt

student@Lenovo2 MINGW64 /e/work
$ git init
Initialized empty Git repository in E:/work/.git/

student@Lenovo2 MINGW64 /e/work (master)
$ git add contributors.txt
warning: LF will be replaced by CRLF in contributors.txt.
The file will have its original line endings in your working directory

student@Lenovo2 MINGW64 /e/work (master)
$ git commit -m 'Initial commit with contributors'
[master (root-commit) 6e34302] Initial commit with contributors
 1 file changed, 1 insertion(+)
 create mode 100644 contributors.txt

student@Lenovo2 MINGW64 /e/work (master)
$ git remote add origin https://github.com/PCervenka/java_training.git

student@Lenovo2 MINGW64 /e/work (master)
$ git push -u origin master
```

Vysvětlení výše uvedeného postupu

-Příkaz echo zobrazí na obrazovce text. Znak > způsobí přesměrování tohoto textu z obrazovky do souboru (pokud už existoval, je přepsán). Dva znaky >> způsobí přidání textu na konec souboru. Jelikož náš soubor předtím neexistoval, tak je jedno, zda použijeme > nebo >>.

`echo "Cervenka" >> contributors.txt`

Samozřejmě je možno tento soubor vytvořit i v Notepadu.

Pro kontrolu spustíme `git status`. Uvidíme, že soubor contributors.txt je červeně v sekci Untracked files

-Příkazem `git init` jsme gitu oznámili, že zde bude naše working copy, zde budeme vytvářet zdrojové soubory. Ve složce vznikne skrytá složka s názvem .git. Ověřte si, zda složka vznikla. **Pozor**: v Průzkumníku je nutno mít zatrženu volbu *Zobrazení/Skryté položky*.

-Poté prohlásíme, že se soubor contributors má dát do seznamu souborů, které se budou commitovat (tedy budou v seznamu staged)
`git add contributors.txt`
Pro kontrolu spustíme `git status`. Soubor contributors.txt je nyní zeleně v sekci Changes to be committed.
Pozn.: pokud bychom chtěli přidat všechny tracked soubory ve složce, které se od posledního commitu změnily, zadali bychom (včetně tečky) `git add .`
Pokud bychom chtěli například přidat všechny zdrojové soubory Javy, tak `git add *.java`
Více souborů je možno zadat odděleno mezerou.

-Potom provedeme commit
`git commit -m 'Initial commit with contributors'`
Pro kontrolu spustíme `git status`. Zobrazí se: nothing to commit, working tree clean. Pokud bychom zapomněli na komentář, tak se spustí implicitní textový editor (viz přepnutí z vim na Notepad výše)

-Pak už provedeme provázání se vzdálenou repository:
`git remote add origin https://github.com/PCervenka/java_training.git`
Tedy jsme se napojili na úložiště java_training uživatele PCervenka. Na toto uložiště se příště budeme odkazovat slovem origin.
Ověření: `git remote show origin`
Zobrazí se:
```
* remote origin
  Fetch URL: https://github.com/PCervenka/java_training.git
  Push  URL: https://github.com/PCervenka/java_training.git
```

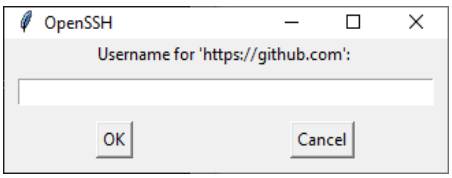
-a poté konečně obsah lokální repository zkopírujeme do remote repository
`git push –u origin master`

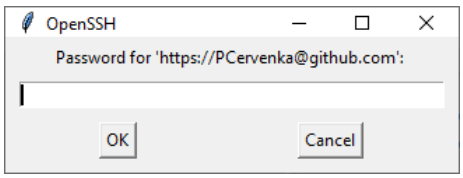
kde origin je „kam“ a master je jméno větve (je na konci promptu v závorce). Pokud větev na remote repository již je, tak ji aktualizuje, jinak ji vytvoří.

Pozor!!!: tento první příkaz push potřebuje parametr –u (neboli --set-upstream), čímž se zadá název remote repository (zde origin) a větve (zde master). Bez toho dojde k fatal error, push se neprovede! Zároveň příkaz nastaví, jaký název větve a remote repository bude pro push a fetch implicitní. Takže příště již stačí napsat `git push` (přesněji řečeno příště už –u být nesmí!!!)

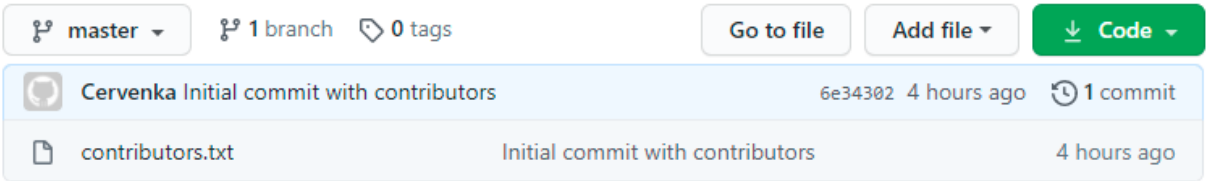
Příkaz `git push` zapisuje na GitHub, proto se objeví okno (viz obr. vlevo) s požadavkem zadání *Username or email* a *Password*. Ať zadáme cokoliv, stejně nebudeme úspěšní (v Git Bash se vypíše `Logon failed`…) a objeví se druhý pokus o přihlášení, tentokrát v protokolu OpenSSH (viz obr. uprostřed). Zadáme své uživatelské jméno (v mém případě PCervenka) a zobrazí se okno (viz obr vpravo), kde ale nezadáme heslo, ale token.

Pozn.: dialogové okno OpenSSH se někdy nezobrazí na popředí, najdeme je pak minimalizované na liště.





Povedlo se. V GitHubu na kartě <>Code nyní je:



Pozn.: jen první příkaz push musel uvést odkud kam se uploaduje. Příště už bude stačit zdat jen git push. A nebude dotaz na token.

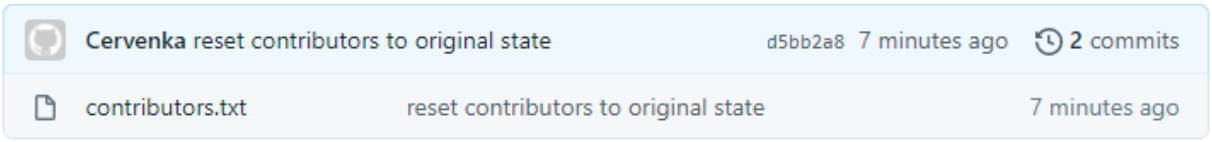
Push do remote repository, která se zatím nezměnila

1)Výchozí situace: je stejný obsah working copy a lokální i remote repository, je v nich jen soubor contributors.txt, který jsme vytvořili při tvorbě repository. Protože jsem si s ním od vytvoření trochu hrál, tak jsem vrátil původní obsah z doby vytvoření GitHubu, tedy jen slovo Cervenka. Příkazem `git status` si ověřím, zda git zaregistroval, že se soubor změnil (červeně vypíše modified). Pak jsem zadal

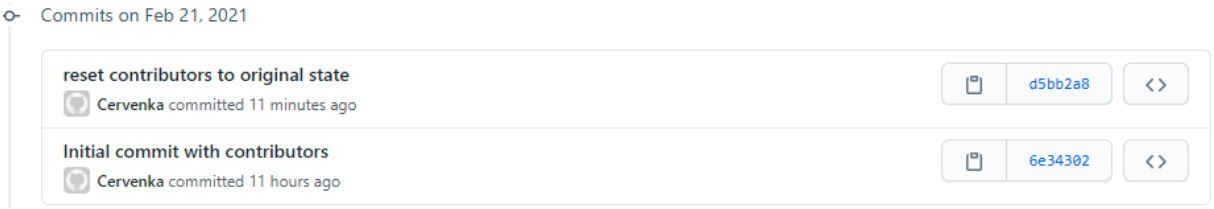
```
git add .                    (git status ukáže nyní soubor zeleně, v sekci Changes to be committed)
git commit -m 'reset contributors to original state'
                             (git status ukáže, že local repo je nyní o 1 commit popředu před GitHubem)
git push
```

```
student@Lenovo2 MINGW64 /e/work (master)
$ git add .

student@Lenovo2 MINGW64 /e/work (master)
$ git status
Changes to be committed:
        modified:   contributors.txt

student@Lenovo2 MINGW64 /e/work (master)
$ git commit -m 'reset contributors to original state'
[master d5bb2a8] reset contributors to original state

student@Lenovo2 MINGW64 /e/work (master)
$ git status
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

student@Lenovo2 MINGW64 /e/work (master)
$ git push
To https://github.com/PCervenka/java_training.git
   6e34302..d5bb2a8  master -> master
```

Toto se pak zobrazí v GitHubu:

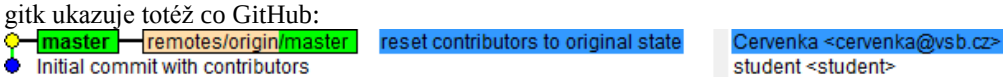


Po klepnutí na odkaz *2 commits* se oba commity zobrazí:

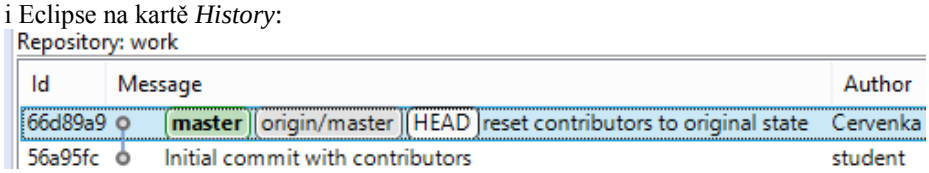


Následují obrázky z gitk a z Eclipse (v perspektivě GIT) pro srovnání. Tyto obrázky byly sejmuty při dřívější práci s BitBucketem. Proto mají jiná čísla commitů a autor tam je Cervenka místo aktuálního PCervenka. Aby se to nemísilo s popisem práce v Git Bash, tak popis práce s gitk a Eclipse bude odsazena. Práce s gitk, GIT-GUI a Eclipse je popsána v druhé části skript v kapitole GUI versus Git Bash.

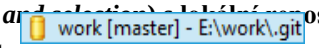
gitk ukazuje totéž co GitHub:



i Eclipse na kartě *History*:



Pozn.: Pokud se na kartě *History* v Eclipse nic neukazuje, pak je třeba její obsah provázat tlačítkem (*Link with editor and selection*) s lokálním repository, tedy nestačí jen v panelu *Git repositories* řádek work vybrat:



V panelech v Git-GUI ani v Eclipse na kartě *Git Staging* nic není a git status hlásí:

```
Your branch is up-to-date with 'origin/master'.
nothing to commit, working tree clean
```

2)Vytvoříme ve složce working copy (tedy té, která má podsložku .git) soubory text1.txt a text2.txt (nebo je odněkud zkopírujeme). A do souboru contributors.txt doplníme 2. řádek s textem např. 2nd line.

3)V Git bash spustíme git status:

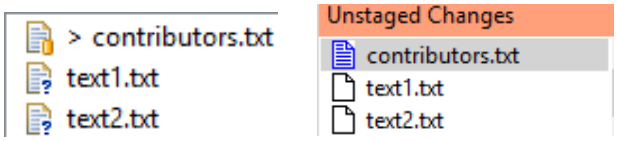
```
lenovo1+student@lenovo1 MINGW64 /e/work (master)
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working

        modified:   contributors.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        text1.txt
        text2.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

Červené texty jdou špatně vidět. Uprostřed obrázku je: `modified: contributors.txt.` a dole je pod sebou: `text1.txt` a `text2.txt`

Vidíme, že systém zaregistroval, že soubor contributors.txt byl změněn. Ale je v sekci Changes not staged for commit (a červeně). Nové soubory jsou červeně v sekci Untracked files (git tedy ještě nemá vytvořen jejich snapshot), tedy nejsou sledované gitem. V obou případech Git navrhuje použít příkaz git add (viz poslední řádek).

*Pozn.: Ještě se soubory dělí na sledované (**tracked**) a nesledované. Sledované jsou soubory, které byly již v předchozím commitu. Takže u nich systém zjistí, že byly od posledního commitu modifikovány a doporučí jejich přidání příkazem add do stagig area. Po klonování jsou tedy všechny soubory sledované. Když přidáme nový soubor, tak git status jej uvede v sekci untracked (a zmíní možnost add). Untracked soubory odlišuje od unstaged jen utilita **git status**, naopak git-gui i Eclipse zobrazuje oba typy v panelu Unstaged changes. Seznam tracked souborů se nazývá index.*

Podobné to bude na kartě *Git Staging* v Eclipse (refresh se provede automaticky, pokud spěcháte, použijte tlačítko *Refresh*), zde jsou všechny soubory v panelu Unstaged changes (untracked soubory mají otazník). Obdobně v Git-gui (untracked s prázdnou ikonkou), zde refresh provedeme klávesou F5 (automaticky se neprovede).



Kdybychom dali nyní git commit, tak by se objevila hláška No changes added to commit.

Pozn: hlášení your branch is up-to-date with ‘origin/master’ znamená, že stav naší lokální repository je totožný s větví master v remote repository origin. Je to pravda, dosavadní změny jsou jen ve working copy, do local repository se dosud nepromítly.

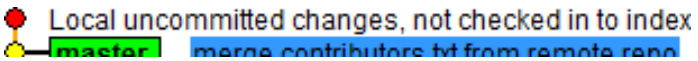
4)Výpis git status navrhuje, abychom použili git add název_souboru. Provedeme to tak se souborem contributors.txt.

Na kartě *Staged Changes* v Eclipse si po Refresh ověříme, že soubor skutečně skočil do spodního panelu. A z cvičných důvodů pro soubor text1.txt provedeme totéž v Eclipse: stáhneme jej do spodního panelu Staged changes. V Git-GUI provedeme cvičně totéž klepnutím na ikonce text2.txt. Ale dalším klepnutím vrátíme zpět nahoru.

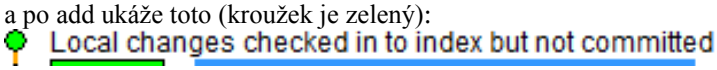
Ověříme i pomocí git status, barva souborů se změní na zelenou a jsou v sekci Changes to be committed, v ní jsou soubory přidané již do staging area, které ale ještě neprošly commitem. Soubor text2.txt zůstává červeně v sekci untracked.

```
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   contributors.txt
        new file:   text1.txt
```

Pozn.: Program gitk (na rozdíl od karty *History* v Eclipse) zobrazí kromě commitů (mají modrý kroužek) i informaci, že od posledního commitu se nějaký soubor změnil (kroužek je červený):



a po add ukáže toto (kroužek je zelený):



5)Provedeme commit, příkaz by měl obsahovat i komentář, co se odposledního commitu změnilo:

```
git commit -m '2nd line in contributors, new file text1.txt'
```

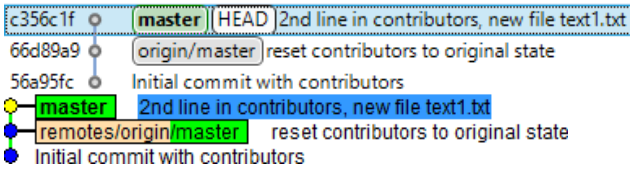
Git status ukáže, že sekce Changes to be committed zmizela. A text your branch is up-to-date je nahrazen informací, že lokální repository je proti vzdálené popředu o jeden commit:

```
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)
```

V Git-Gui i Eclipse došlo k vyprázdnění panelu Staged changes

Zůstal jen soubor, který je unstaged. Smažeme jej, ať nás ve všech dalších výpisech nerozptyluje.

Přesvědčíme se, že v Eclipse na kartě *History* přibyl další commit (podobně v gitk), v Git Bash ověřovat nebudeme, tam je to nepřehledné.

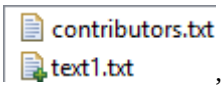


Zatímco předtím byl na nejvyšším řádku nápis "master" i „origin/master", protože lokální a remote repository byly ve stejném stavu, tak nyní je na nejvyšším řádku jen master (tedy větev master z lokální repository), a origin/master (tedy master z remote repository „origin") je o jeden commit níže, tedy pozadu.

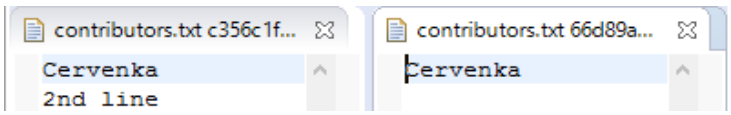
Ještě si všimněte, že u nejvyššího řádku je text „HEAD" (v gitk má tutéž funkci žlutý kroužek), což označuje, že v tomto stavu jsou soubory ve working copy.

Pozn.: příští commit provedeme pro změnu v prostředí Eclipse.

Pozn.: git commit se dá použít i bez předchozího příkazu add, v tom případě se ale musí v příkazu git commit použít navíc parametr –a. V tom případě se provede nejprve add všech tracked souborů, které se zatím změnily. Nepřidají se do commitu nové soubory, pro ně by se muselo spustit add ručně.

6)Prohlédneme si aktuální a dřívější obsah souborů v Eclipse na kartě *History* a v gitk. V panelu vpravo dole uvidíme seznam souborů v lokální repository (nejsou zde untracked soubory). Je-li vybrán nejvyšší řádek historie, tak jsou v panelu vpravo dole dva soubory: 

, vybereme-li předchozí commit, tak je zde jen soubor contributors.txt. Poklepeme-li na tomto souboru při obou commitech, dostaneme v editačním okně výpis obou souborů, pro odlišení je uvedeno číslo commitu. Dáme-li výpisy vedle sebe, vypadá to takto:



Výpis v gitk na to jde jiným způsobem: vypíše jen rozdíly proti minulému commitu:

```
--------- contributors.txt -
index 984ea8c..1b781c5 10064
+2nd line
```

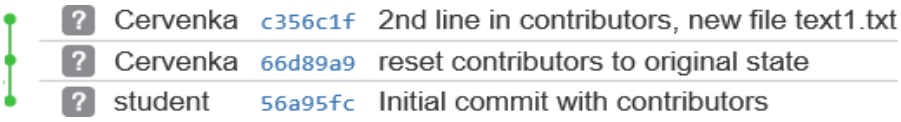
7)Pro připomenutí se podíváme na seznam commitů na remote repository na webu GitHub.com. Vidíme, že se nezměnil, poslední commit zde ještě není.

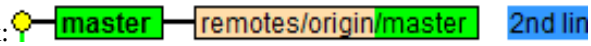
8)Provedeme upload lokální repository do remote příkazem `git push`. Nebo `git push origin master`

> Pokud bychom použili tlačítko *Push* v Git-gui, tak by se ještě objevilo dialogové okno, kde bychom vybrali, kterou větev (nyní máme jen master) a do které repository (nyní jen origin) chceme uploadovat. Nebo v Eclipse *Push to Upstream* z místní nabídky příslušné local repository v levém panelu *Git Repositories* (případně *Push Commit* z místní nabídky posledního commitu v Eclipse na kartě *History*).

V git status se opět objeví informace, že lokální větev je up-to-date se vzdálenou.

9)Načež se tento commit objeví i v remote repository (po refresh – F5) (pozn.: starý obrázek, z BitBucketu)



> V Eclipse na kartě *History* se ukáže, že opět je na nejvyšším řádku i stav remote repository, tedy je na tomto řádku master i origin/master:
>
> 
>
>
> Podobně v Gitk:
>
> 
>

Pozn.: Pomocí git help push získáme syntaxi (zkráceno):

git push [<repository> [<refspec>]]

kde refspec je název větve. Není-li uvedeno, pak je to aktuální větev (obvykle master). Ale z bezpečnostních důvodů to projde, jen pokud již taková větev v remote repository je. A není-li uvedeno jméno repository, pak to je origin.

Klonování

Stažení programového balíku z public repository GitHubu je velmi jednoduché. Uživatel si vytvoří nějakou složku, skočí do ní pomocí Git Bash here a zadá příkaz git clone následovaný adresou remote repository, v našem případě to tedy bude:

```
git clone https://github.com/PCervenka/java_training.git
```

Pozor: původně jsem měl složku e:\work, ve které byl soubor contributors.txt a skrytá složka .git. Nyní jsem si vytvořil složku např. e:\work2 a v ní jsem dal výše uvený příkaz git clone. Výsledkem však nebude soubor contributors.txt a složka .git ve složce work2. Do aktuální složky (tedy u mne work2) se příkazem git clone stáhne složka s názvem uvedeným na konci, tedy java_training. Teprve v ní je soubor contributors.txt a složka .git.

Toto ještě není práce v týmu, tedy uživatel, který si repository stáhl, nemá právo po úpravě toto repository na gitHubu aktualizovat. Toto budeme brát až v druhé části skript, tedy při práci v týmu.

Část II: práce v týmu a práce s verzemi

Stručný přehled základních příkazů

Po mém prvním zkopírování aktuálního stavu programu do mé lokální repository a pak do remote repository pokračuje postup práce (workflow) takto: kolegové z týmu si tuto mou remote repository naklonují (**clone**) – viz výše - a vytvoří si svou local repository. A z ní si pak zkopírují příkazem **checkout** poslední verzi do svého working copy.

Jakmile potom provede někdo nějakou ucelenou změnu, tak provede testy (!!!), zařadí pomocí příkazu **add** změněné soubory do seznamu souborů připravených pro commit a potom provede **commit** ze své working copy do své lokální repository. Při každém commitu je třeba napsat, co se změnilo. Ale nepsat názvy změněných souborů, ty jsou vidět v pravém dolním rohu (v Eclipse v perspektivě Git na kartě *History*, v gitk to je podobné).

Načež provede **push** z lokální do společné týmové remote repository (termín push se při práci s repository používá místo termínu upload). Než ale spustí push do remote repository, tak se musí ujistit, zda zatím někdo neposlal do remote repository něco nového, to by se totiž push nepodařil - fail. Proto zvolí nejprve **fetch** z remote repository (obdoba download), tím mu v local repository vznikne další větev. Fetch na rozdíl od clone nekopíruje celé repository ale jen rozdíly.

Pokud v mezidobí (tedy od doby předchozího mého stažení z remote repository příkazem fetch) nikdo nic v remote repository nezměnil, tak se nová větev nevytvoří, a lze hned poté použít příkaz push. Pokud mezitím byla nějaká změna, tak v lokální repository vznikne větev. Musí se nejprve spojit s větví master pomocí **merge** (nebo pomocí rebase, viz níže).

Pokud v mezidobí každý upravoval jiný soubor, pak je merge (spojení) změn daného programátora bezproblémové. Nebo když se upravují různá místa téhož souboru. Pokud však dva programátoři pracovali na stejném místě souboru, je nutné rozhodnout (ručně editovat), která z úprav se zachová, případně zda obě. K přehlednému zobrazení změn v obou souborech slouží příkaz **diff**, verze v příkazovém řádku je ale nepřehledná, tento jediný přikaz budeme trénovat jen v GUI. Eclipse v perspektivě Git k tomu nabízí okno s dvěma panely s oběma soubory, stránkují se synchronně. V případě nejasnosti je dobré podívat se v libovolném editoru ještě na společného předka, tedy stav tohoto souboru v commitu, ze kterého obě větve vycházejí.

Pozn.: Společného předka lze zobrazit i pomocí `git show :1:jméno_souboru`

Po vyřešení úprav v daném souboru znovu spustíme merge.

Druhou možností je místo merge použít **rebase**, to zahladí větvení, které nebylo jako větvení zamýšleno. Tedy linearizuje, bude to vypadat, jako by nejprve kolega provedl změny, já si je pak stáhl a já až potom jsem udělal své změny. Je to pak přehlednější.

A teprve po merge či rebase provedeme push do remote repository.

Ideální je, když si kolegové hned poté tuto spojenou verzi přenesou (fetch) z remote repository k sobě (ale mohou i kdykoliv později, budou jen řešit více konfliktů). A pak můžeme všichni začít znovu pracovat, máme stejný výchozí bod.

K vrácení se ke stavu, který byl při některém z předchozích commitů, slouží příkaz:

git **checkout** <číslo commitu, tag nebo větev>. Tím se aktualizují všechny soubory v pracovní složce do stavu, v jakém byly v té době. Můžeme zde vytvářet nové větve nebo commit upravovat. Poté se dostaneme zpátky příkazem git checkout master. Tím přepneme na poslední commit, tedy na vrchol větve master.

Novou větev vytvoříme příkazem git **branch** <Název_větve>. Někdy se větev vytváří i dočasně. Například někdo v týmu vyvíjí nějakou jednu funkcionalitu. Bylo by pracné pořád to s hlavní větví spojovat. Až to dodělá a ověří se funkčnost, tak se to spojí.

git **checkout** <Název_větve> přepne ukazovátko HEAD na tuto větev. V Git bash bude tato větev v závorce na konci řádku místo (master). V grafickém režimu (např. na kartě History) bude na daném řádku tag HEAD (dosud byl vždy na vrcholu větve master).

Syntaxi kteréhokoliv příkazu získáme pomocí git **help** název_příkazu.

Pozn.: Nemusí každý commit být hned následován příkazem push. Může být i několik commitů na jeden push. Programátor může dlouho pracovat v místě bez internetu a ukládat postupně několik verzí jen do lokální repository. A až po návratu do civilizace nahrát do remote repository. Dočasné ukládání jen do lokální repository je také ochranou proti zbrklosti. Jakoukoliv nepodařenou úpravu programu lze vrátit zpět, aniž by se to kdo dozvěděl. Po publikování v remote repository to již neutajíme (opravit kód samozřejmě lze).

Pozn.: `git diff HEAD` vypíše rozdíly mezi pracovní složkou a posledním commitem (HEAD)

Pozn.: lokální repository ví o remote repository (tedy o odlišnostech), remote o lokální neví (to by musel sledovat všechny lokální repository členů týmu).

Jenkins: integrační nástroj, který si osahává remote repo, při změně (novém commitu) vše zkompiluje a prožene testy. A pokud někdo poslal na repository projekt, který nemá všechny testy zelené (nedejbože projekt, který nelze skompilovat), tak u jeho jména svítí nějakou dobu výstražná ikona. A šéf to vidí...

Terminologie:

Head je poslední verze v dané větvi (last revision within given branch). Pro verzi se v angličtině používá častěji termín revision, ne version.

Master branch: vždy by jedna větev měla být hlavní. Název Master dostane první větev, která se v Gitu objeví při jeho založení. Lze však přejmenovat.

Tag (návěští): místo, kam se chceme často vracet. Vrátit se je sice možné ke kterékoliv verzi, ale špatně se v tom hledá (sedmimístná hexa čísla commitů). Používání tagů to zjednodušuje. A sedmimístná čísla commitů jsou ve skutečnosti jen prvními sedmi znaky 40-ti místného sha-1 hashe.

Origin: default alias for remote repository. Obvykle remote repository, ze které vznikla klonováním moje local repository.

GUI versus Git Bash

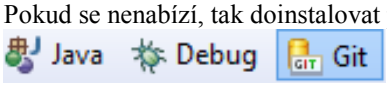
V kapitole Stručný přehled základních příkazů byly uvedeny příkazy bez vysvětlení, jak se konkrétně zadávají. Jsou tři možnosti:

-v Git Bash, tedy v příkazovém řádku Linuxu

-v prostředí Git GUI, které se spouští v Průzkumníku z místní nabídky složky volbou *Git GUI Here*. Nebo se spouští z prostředí Git Bash příkazem git-gui (není nutný na konci znak &). Zobrazuje totéž jako příkaz git status. Nahoře v pull-down menu je položka *Remote*. Rozbalí se nabídka, kde je *Fetch from, Add, Push* atd. Další nástroj je gitk, který zobrazí historii commitů včetně větvení, podobně jako karta *History* v Eclipse. Spouští se z prostředí Git Bash příkazem gitk &. Znak & v Linuxu způsobí spuštění aplikace na pozadí. Kdybychom & nepoužili, tak by okno Git Bash stále čekalo, až se okno gitk uzavře.
Pomocí gitk se díváme na local repository, pomocí git gui na staging area

Pozn.: někteří používají SourceTree – jiný nástroj obdobný gitk.

-v Git perspektivě v Eclipse. Tento způsob je nejjednodušší, není nutné mít nic dalšího otevřeného. Ale pro začátek je vhodné zkoušet si příkazy i z příkazového řádku. Protože grafické prostředí přesně tyto příkazy generuje.

Změníme uspořádání Eclipse tak, aby zobrazovalo tlačítko perspektivy Git: *Open perspective* a vybrat *git*. Pokud se nenabízí, tak doinstalovat plugin egit. Potom se mezi perspektivami dá přepínat tlačítky *Java* a *Git:* 

zcela vpravo na panelu nástrojů.

Pak bude v levém panelu seznam všech repozitory. Standardně je jen jeden. Pokud však programátor pracuje ve více týmech, tak má i více repository (nebo se jedná o lektora, který sleduje práci více týmů studentů. Nebo student který se chce podívat, jak si s domácím úkolem poradil jiný studentský tým). Pokud tedy chce lektor otevřít všechny poslané projekty, tak klepne v panelu Git repositories na ikonku *Add* . Objeví se okno, v ní v boxu *Directory* nabrouzdá složku, která je nadřazená složkám všech klonovaných repository. Zvolí *Search*. Objeví se všechny repository (tedy složky s podsložkou .Git) – samozřejmě jen ty, které ještě nejsou přidány - naráz přidá všechny.

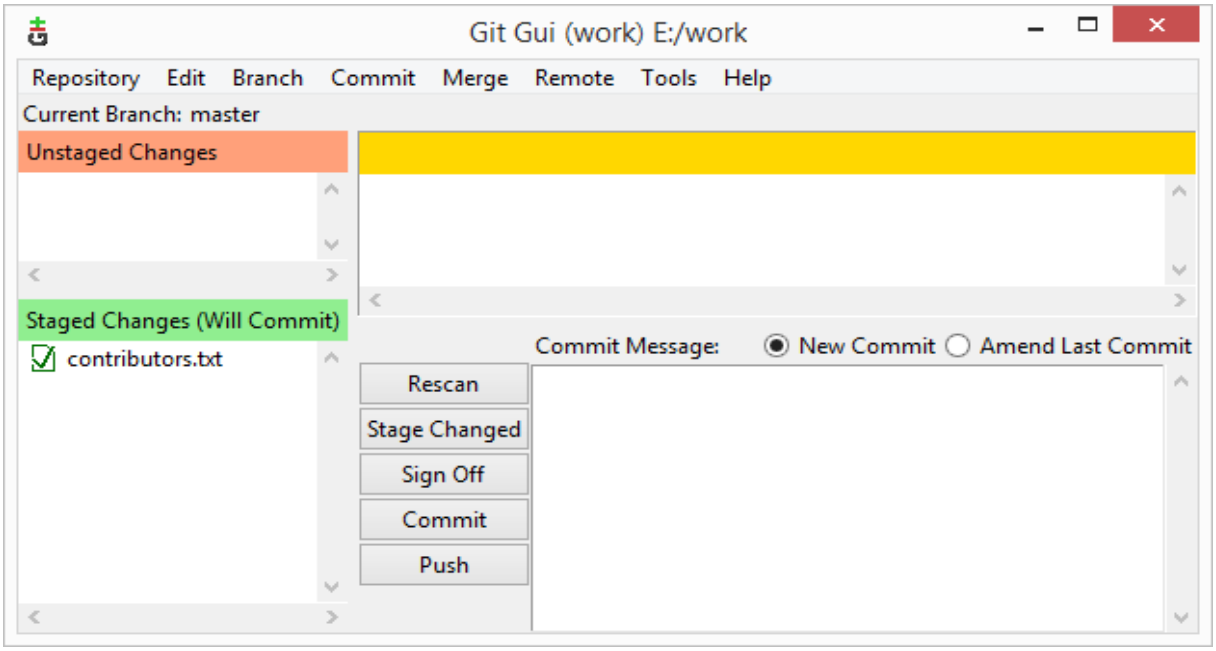
Pozn.: Je dobré umět git-gui, neomezovat se jen na využití v Eclipse. Pro případ práce na PC, kde by byl jen NetBeans či jiné IDE.

Doporučení pro tréning: příkazy nejprve provést např. v příkazovém řádku a podívat se, jak se to projevilo v GUI. A pak naopak.

Pozn.: jak je zvykem v Linuxu, git bash dokáže dokončovat příkazy pomocí tabulátoru, pokud je jednoznačné, který příkaz na daná písmena začíná. Takže git com<TAB> doplní na git commit. Pokud bychom však napsali jen git co<TAB>, tak se nedoplní nic, protože je více možností. Můžeme si nechat ty možnosti zobrazit druhým stisknutím tabulátoru, tedy git co<TAB><TAB>, nabídne se commit a config.

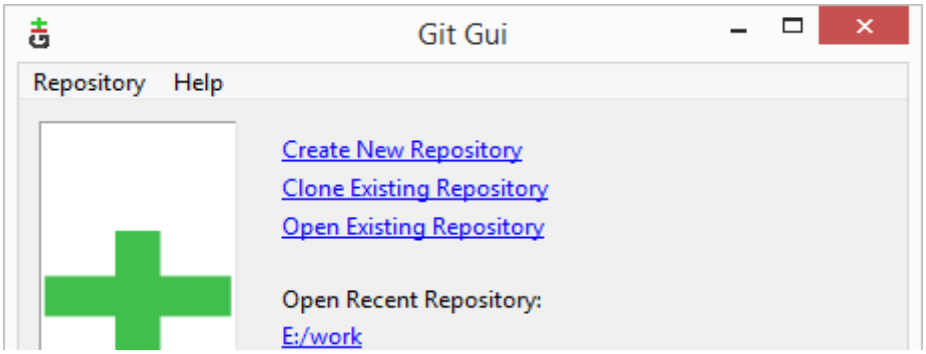
Obrázky celých aplikací:

Spustíme-li git-gui v Průzkumníku (nebo Total Commanderu) volbou *Git GUI here* z místní nabídky složky, která obsahuje podsložku .Git, pak program ukáže status dané repository:

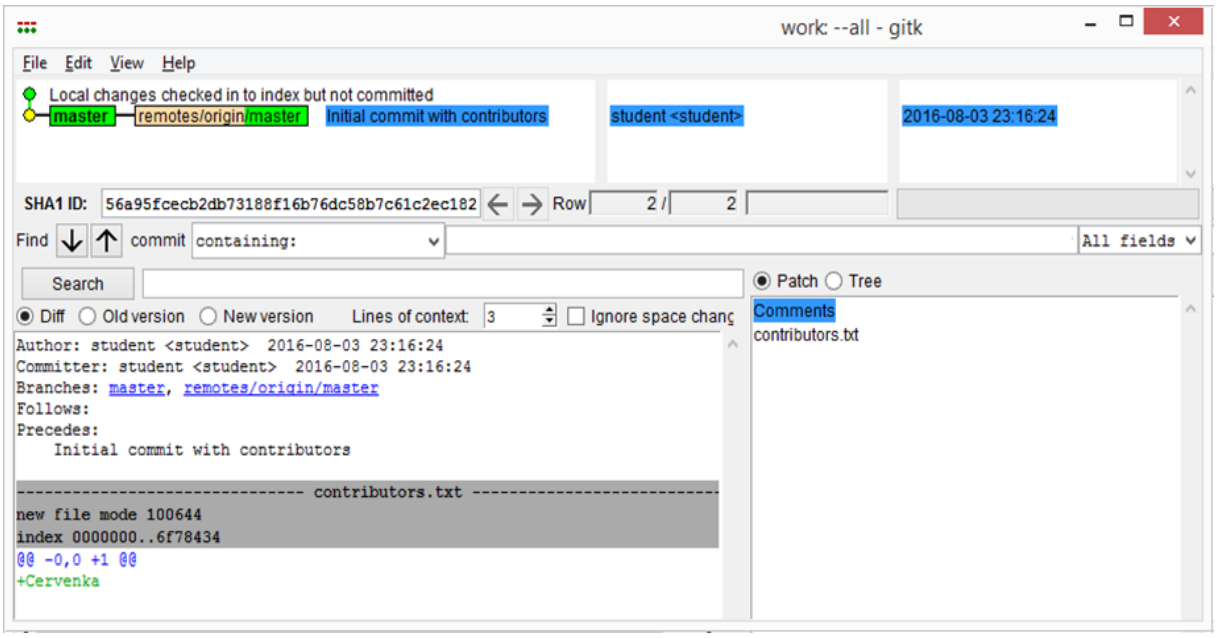


Totéž se ukáže, pokud spustíme pomocí git-gui v Git bash, pokud je aktuální daná složka.

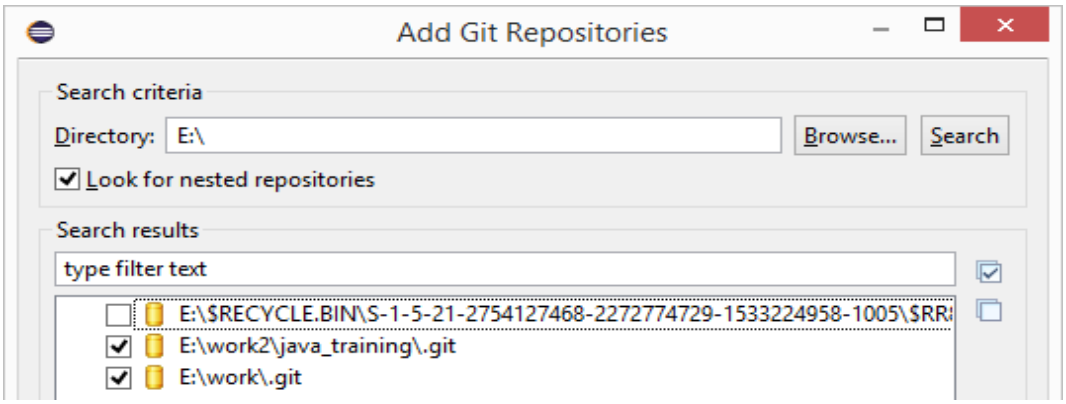
Pozn.: Pokud spustíme z jiné složky, objeví se obrázek níže a musíme ještě pomocí *Open Existing Repository* do správné složky nabrouzdat (případně využít *Open Recent Repository*):



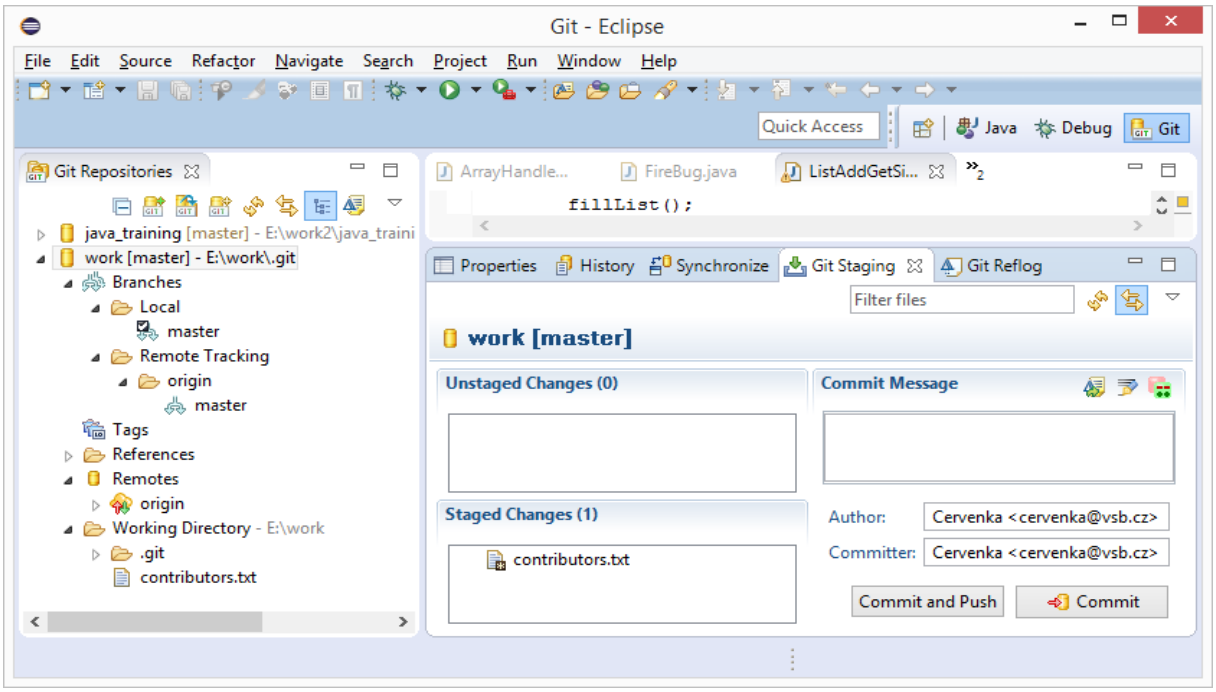
Z git-gui se můžeme přepnout do gitk, a to pomocí *Repository/Vizualize All Branch history*. Zobrazí se:



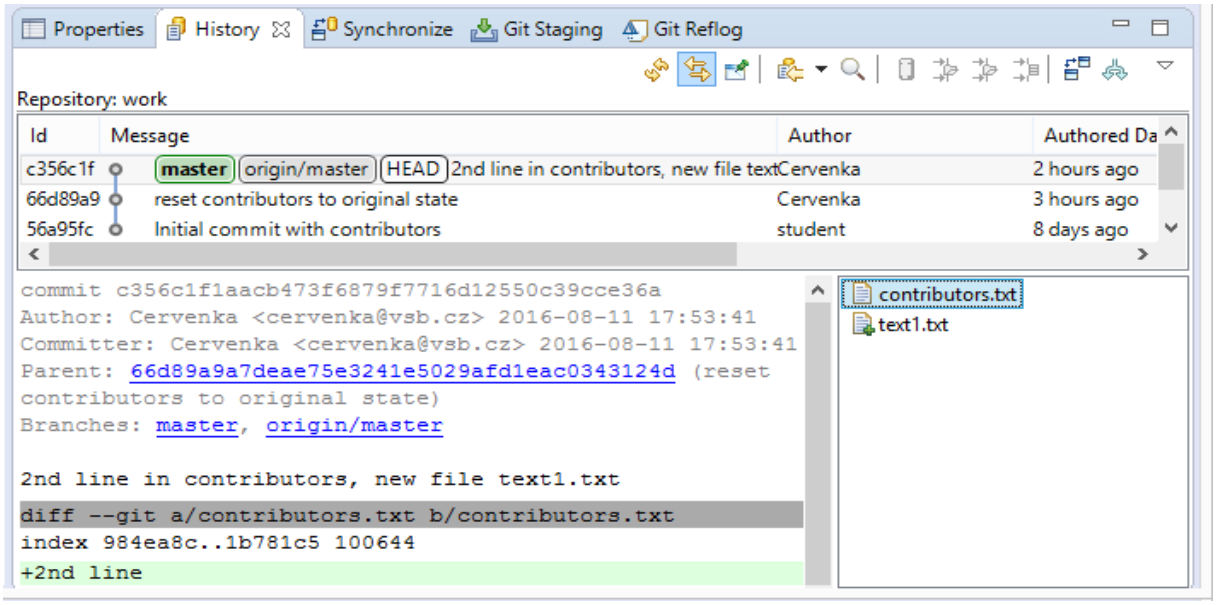
Následuje ukázka Eclipse v perspektivě Git. Nejprve ale musíme do panelu *Git Repositories* vlevo přidat ikonkou  *Add an existing local git repository to this view* všechny **lokální** repository, které chceme spravovat. Ukáže se okno *Add Git Repositories*.



Nejjednodušší je nabrouzdat na složku, která je všem local repository nadřazena (pokud taková je), potom klepneme na tlačítko *Search* a pak se v boxu *Search results* všechny ukážou. Zatrhneme ty, které chceme sledovat (viz horní obr.). Výsledek pak vidíme na obrázku dole, v levém panelu (karta *Git repositories*). Vidíme tam složku java_training i work. Vybrána je repository work, té odpovídají karty na panelu vpravo, zobrazená je právě karta *Git Staging*:



Karta *History* vypadá takto:



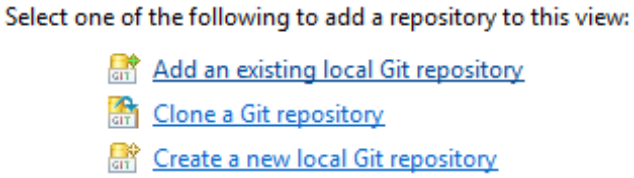
V panelu v pravém dolním rohu je vidět, které soubory se ve vybraném commitu změnily. Když pak na nějaký klepneme, pak se vlevo od toho dole objeví - uvedené nadpisem diff - odlišnosti tohoto souboru proti stavu v předchozím commitu. Přidaný text je zelený, odebraný červený. Podobně je to v Git-GUI.

Vidíme, že karta *Git staging* je prakticky totožná s programem git-gui. A karta *History* je zase totožná s programem gitk.

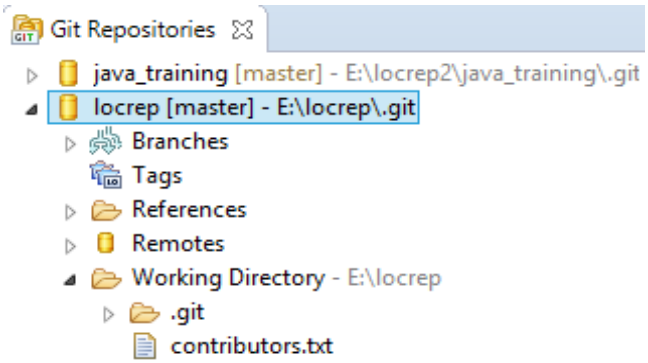
Pozn: snímek obrazovky v Eclipse je z commitu o jedno pozdějšího než snímek gitk. Ale už to opravovat nebudu.

Odbočka:

V Eclipse lze sledovat čísla verzí a případná větvení po přepnutí do perspektivy Git. Pokud v tomto režimu pracujeme poprvé, tak se po přepnutí objeví:



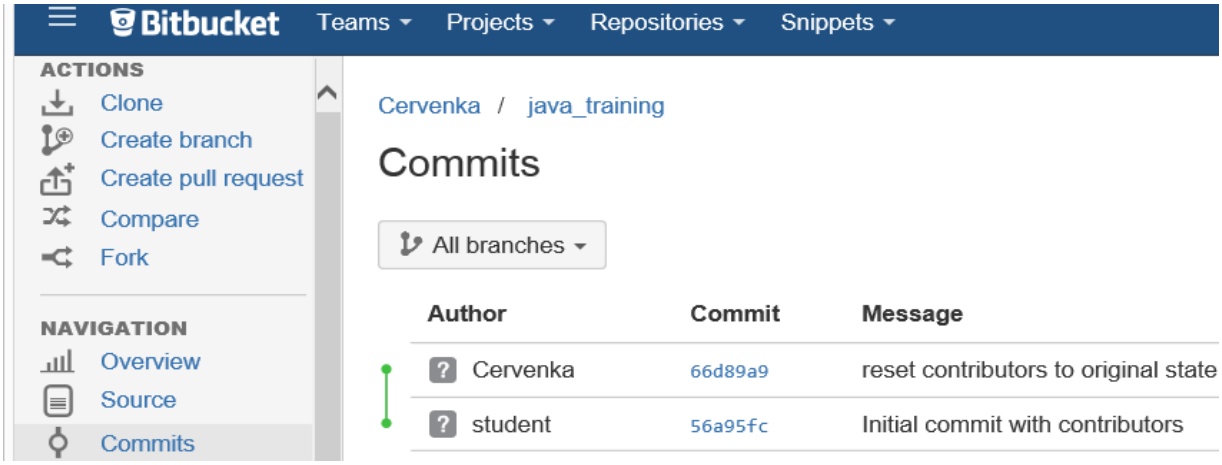
Zvolíme *Add an existing local Git repository*, nabrouzdáme cestu k E:\locrep. Případně přidat ještě další, ve složce locrep2\java_training. V panelu Git Repositories se pak objeví:



Pozn.: ikonku (*Add an existing local Git repository*) emůžeme využít i kdykoliv později. Bude v pásu ikonek nahoře v panelu Git Repositories.

Konec odbočky

To byla práce s lokálními úložišti. Jak sledovat **historii** commitů a větvení **remote** repository: na webové stránce Bitbucket, vlevo třetí ikonka zvrchu (Commits). A je třeba, aby se zobrazovaly všechny větve All branches:



Pozor: Pokud se na začátku práce po prvním commitu v gitk nic neobjeví, tak pomůže místo F5 dát Shift+F5. I v některých jiných situacích je výjimečně nutné dát Shift+F5.

Pozor: Na začátku práce s novým lokálním repository je nutné provést provázání karty *History* s danou repository. Provede se to nejprve tlačítkem *Pin this History View* a potom *Link with Editor and Selection*.

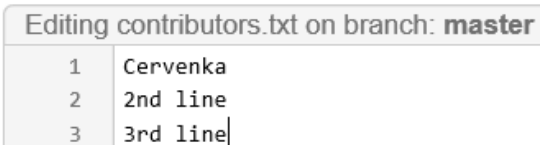
Push

Následují tři případy Push, seřazené s rostoucí obtížností:
-Push do remote repository, která se zatím nezměnila (bylo probráno v první části)
-Merge (automatické)
-Merge s ručním řešením konfliktu

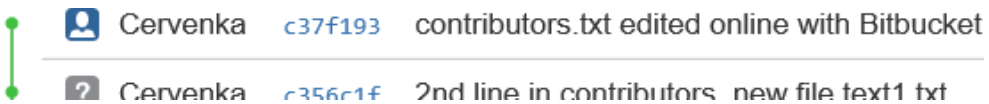
Merge (automatické)

Nyní si demonstrujme případ, kdy se v mezidobí stav remote repository změnil. Nejvěrněji to bude odpovídat skutečnosti, když někdo jiný zatím provede změnu a pak ji na remote repository uploaduje příkazem push.

1)Pokud ale nechceme být na něčí spolupráci závislí, pak tuto změnu můžeme nasimulovat tím, že nějaký soubor změníme přímo v remote repository. Použijeme k tomu webové prostředí na bitbucket.org. Na kartě *Source* klepneme na názvu souboru contributors.txt, objeví se jeho obsah. Klepneme na tlačítko *Edit* a budeme mít k dispozici jednoduchý editor. Na konec souboru dopíšeme 3rd line. Potom klepneme na tlačítko *Commit*.



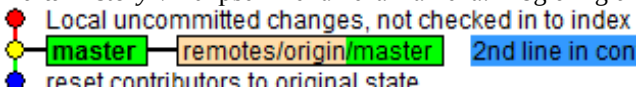
To se hned projeví na odkazu Commit, přibude nahoře nový řádek, message se vygeneruje automaticky:



Ale git status ani ostatní aplikace změnu nezaregistrují. Je tedy vidět, že lokální repository si „neosahává“ stav remote repository.

2)Dále provedeme nějakou změnu ve své working copy. Například na začátek souboru contributors.txt (tedy téhož souboru, ale na jiné místo, než byla změna v remote repository) napíšeme Line 0.

Karta *History* v Eclipse změnu nezaznamená. Program gitk ano, objeví se:



Aplikace Git-GUI a karta *Git Staging* umístí soubor do horního okna.

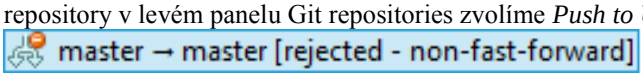
3)Provedeme git add contributors.txt, git commit –m „Line 0 added“. Podíváme se do sekce diff na kartě *History*, před přidaným řádkem je znaménko +, stejné je to i v gitk:

```
+Line 0
 Cervenka
 2nd line
```

A spustíme git push

```
lenovo1+student@lenovo1 MINGW64 /e/work (master)
$ git push
To git@bitbucket.org:cervenkapetr/java_training.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'git@bitbucket.org:cervenkapetr/java_training
.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
```

4)Tedy nepovedlo se, Git poznal, že obsah remote repository se od poslední synchronizace změnil. A protože se to stejně neprovede, můžeme si push z cvičných důvodů vyzkoušet i v prostředí Eclipse. Z místní nabídky repository v levém panelu Git repositories zvolíme *Push to Upstream*. Objeví se box s hlášením:



Do třetice tlačítko 

v Git-GUI, objeví se hlášení totožné s hlášením příkazu git push.

Z toho vyplývá, že bychom před push měli vždy pomocí fetch ověřit, zda zatím někdo něco nedal do repository.

5)Nezbyde tedy než stáhnout aktuální obsah remote repository příkazem:

git fetch origin master

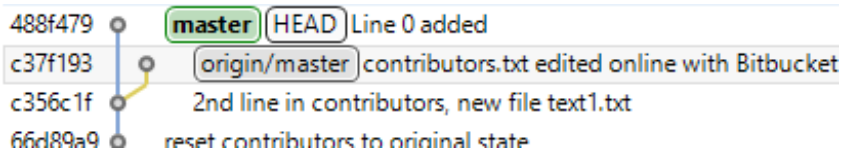
(tedy fetch odkud, jakou_větev), stačí ale jen git fetch, když spolupracujeme jen s remote repository origin a když máme jen větev master)

Druhou možností je provést to v Eclipse: Fetch vybereme z místní nabídky kořene projektu v panelu *Git repositories* vlevo, *Fetch from Upstream.* Objeví se box s např. *Fetch from work – origin*. Zde se zobrazí commity, které se tím stahují (na remote repository zatím mohlo proběhnout několik commitů).

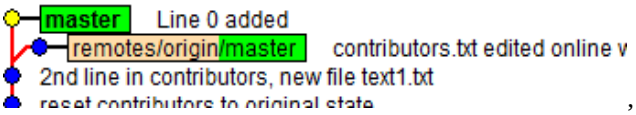
Třetí možnost: v Git-GUI, menu *Remote/Fetch from*.

V Eclipse na kartě *History* si ověříme, že se vytvořila nová větev, bude vycházet z místa, kde byl ještě obsah obou repository společný. V tomto případě větev vychází z předposledního commitu.

Pozor: Pokud se na kartě *History* v Eclipse neobjeví po fetch větev, pak je nutné klepnout na ikonku  ***Show all branches and tags*. Podobně v gitk to je *View/New view*/volba *All (local) branches*, nebo lépe: spustit gitk takto: gitk --all &**



Pozn.: Pokud by ale od poslední synchronizace byl proveden commit v lokální repository vícekrát, pak by byla odbočka níže.



Gitk: , je ale nutno v dialogovém okně *View/edit view* zatrhnout *All refs*

6)Příkaz git status oznámí, že se cesty lokální větve a větve origin/master rozdvojily (diverge je opak od converge). Vypíše se také doporučení použít merge.

```
$ git status
On branch master
Your branch and 'origin/master' have diverged,
and have 1 and 1 different commits each, respectively.
  (use "git pull" to merge the remote branch into yours)
```

7)Spustíme git merge –m “merge of contributors.txt from remote repo“

```
$ git merge -m "merge contributors.txt from remote repo"
Auto-merging contributors.txt
Merge made by the 'recursive' strategy.
 contributors.txt | 1 +
 1 file changed, 1 insertion(+)
```

V Eclipse na kartě *History* se přesvědčíme, že se větve spojily, vytvořil se tím další commit.

Po merge pokračuje ta větev, která je při merge na konci promptu v závorce, tedy aktuální větev. Pokud chceme, aby naopak pokračovala ta druhá, tak musíme předtím dát checkout do té druhé.

Pozor: pokud bychom nedali parametr –m, pak by se po spojení souborů objevilo prostředí vim editoru umožňující dodatečné zadání komentáře. S tímto linuxovým editorem ale často programátoři neumí pracovat, takže ho ukončí (příkazem **:q!**). Proto se commit neprovede, git status ukáže, že soubor je v sekci Changes to be committed, je nutné pak commit spustit ručně.

Pozn.: Do message v commitu by se měl dávat důvod. Tedy ne seznam změněných souborů, to lze vyčíst jinde.

V Eclipse je možno vybrat *Merge* z místní nabídky commitu z remote repository na kartě *History*, komentář se vygeneruje sám, toto je výsledek:

Pozn.: Samozřejmě se předpokládá, že v okamžiku merge je naše lokální repository aktuální, tedy byl čerstvě proveden commit. Naštěstí merge (ani pull, což je fetch+merge, syntaxe: git pull origin master) se nespustí, jestliže zjistí, že některý ze souborů, který chtějí spojovat, je ve stavu uncommitted changes. Zobrazilo by se:

```
$ git merge -m "merge of contributors.txt from remote repo"
error: Your local changes to the following files would be overwritten by merge:
        contributors.txt
Please commit your changes or stash them before you can merge.
Aborting
```

8)Podíváme se v Poznámkovém bloku, jak si příkaz merge poradil s rozdíly, zobrazíme si soubor contributors.txt z workspace. Jelikož odlišnosti byly na různých místech souboru, tak se do výsledné verze promítly oba, tedy nultý i 3. řádek.

```
Line 0
Cervenka
2nd line
3rd line
```

9)Nyní již můžeme zadat git push

10)Přesvědčíme se na webu bitbucket.org na odkazu *Commits* 

, že se sem změny propagovaly, je vidět i větvení.

Pozn.: výše uvedené činnosti lze provést i v git-gui (z pull-down menu), např. *Remote/Fetch from*/Origin, *Remote/Push* (a pak vybrat větev, kterou chceme pushnout, na začátku mám jen master)

Pozn.: Pokud jsme od posledního fetch na programu nepracovali, jen si např. po dovolené stahujeme nový stav programu, pak se nevytvoří větev. Vytvoří se nahoře řádek s jedním commitem s textem remote origin master, je ale v přímé linii. Tedy ne vždy odlišný stav remote a local repository způsobí přidání větve. A tedy pak ani není nutné použít merge.

Pokud je naopak stav remote repository stejný s naším lokálním nebo s některým jeho starším commitem, pak se ani žádný řádek nepřidá.

Pozn.: Ve firmě obvykle merge provádí vždy jen ten, který je za příslušnou část programu odpovědný.

Merge s ručním řešením konfliktu

Když jsme udělali v nějakém souboru změny na různých místech (první a poslední řádek), tak merge proběhl automaticky. Nyní provedeme v nějakém souboru změny ve stejném místě. Proto budeme muset pak editaci spojovaného souboru provést ručně.

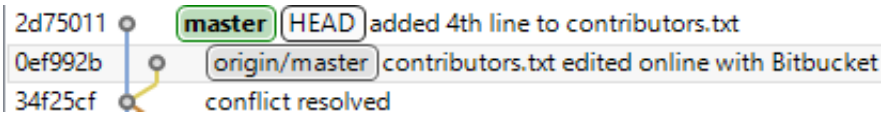
1)Editorem v remote repository doplníme na konec souboru contributors řádek s textem Last line.

2)Ve working copy napíšeme naopak na konec 4th line

3)Add, commit, fetch (stejně jako v předchozí kapitole). Git status pak hlásí:

Your branch and origin/master have diverged

V Eclipse na kartě *History* (podobně v gitk) se objeví větev:

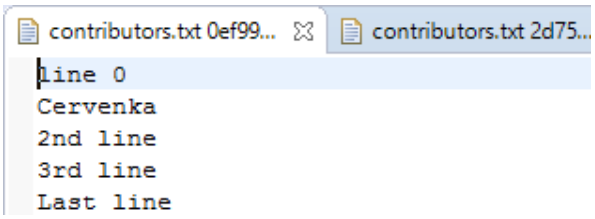


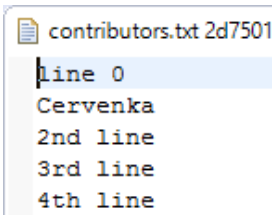
Můžeme klepnout na horní řádek a potom v pravém dolním panelu (kde jsou změněné soubory) klepnout na soubor contributors.txt. Ve spodním panelu se pak zobrazí obsah souboru v diff zobrazení. Pak totéž s commitem z remote repository. Takto vypadají konce souborů:

```
 3rd line
-
+4th line
```

```
 3rd line
-
+Last line
```

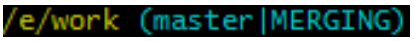
Nebo místo klepnutí můžeme provést 2xclick, pak se soubory zobrazí nahoře v sekci editovaných souborů, pro rozlišení bude u názvu souboru i číslo commitu:





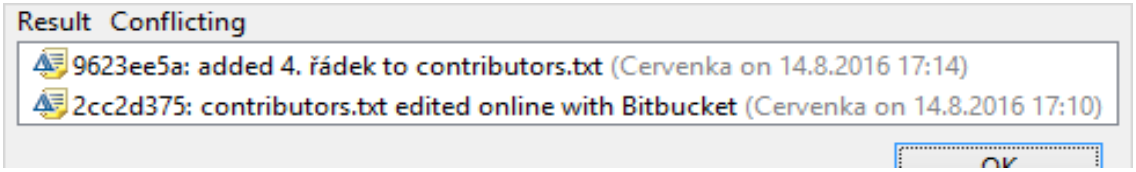
4)`git merge -m "merge conflict on 4ht/last line"`

Merge provede spojení souborů v místech, kde je možné automatické spojení. Potom se kvůli konfliktu přeruší, proto se v promptu příkazového řádku bude až do vyřešení ukazovat navíc |MERGING: (při rebase tam bude REBASE)



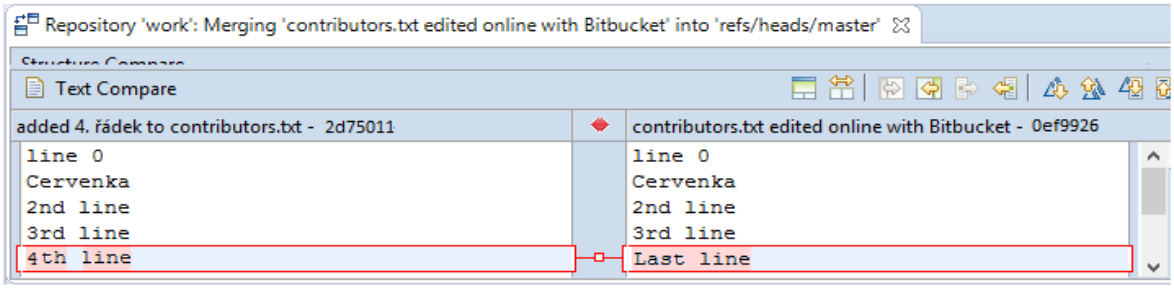
Soubor, kde došlo k nevyřešeným konfliktům, ukáže git status v sekci unmerged paths. Je červeně a u něj stav: both modified (oba modifikovány)

Postup v Eclipse: na kartě *History* z místní nabídky commitu remote repository vybereme položku *Merge*. Objeví se dialogové okno, které rekapituluje, které větve se budou slučovat, a upozorňuje na konflikt:



Žádné další upozornění, že se spojení nedokončilo, není.

5)Podíváme se postupně na všechna konfliktní místa ve všech konfliktních souborech. V Eclipse na kartě *Git Staging* v sekci *Unstaged changes* poklepeme na souboru contributors.txt. V horní editační oblasti se objeví karta s dlouhým názvem Repository ‘work‘: merging contributors.txt editing online with bitBucket with master. A v něm vedle sebe obsahy obou souborů s růžově zvýrazněnými konfliktními místy:



6)Výše uvedené zobrazení obou souborů je přehledné, není ale vhodné pro úpravy. Ty budeme provádět v souboru contributors.txt ve složce work, do kterého už merge sloučil obě verze souboru. Prohlédnout si jej můžeme pomocí git diff, ale upravovat jej budeme v notepadu (spustí se tak, že na souboru poklepeme v Průzkumníku). Nebo jej můžeme otevřít v Eclipse.

```
line 0
Cervenka
2nd line
3rd line
<<<<<<< HEAD
4th line
=======
Last line
>>>>>>> refs/remotes/origin/master
```

Totéž ukáže i git diff

Kvůli řádkům se znaky <, > a = nelze soubor zkompilovat (pokud by to byl zdrojový soubor)

Upravím, např. pokud chci zachovat nové části kódu z obou zdrojů, tak jen odmažu řádky s <, > a = a uložím.

7a)git add contributors.txt

Tím naznačím, že konflikt byl vyřešen. Protože git samozřejmě nesleduje editaci souboru v notepadu

7b)Když pak spustím příkaz git status, tak u tohoto souboru nebude napsáno „both modified“, ale jen modified.

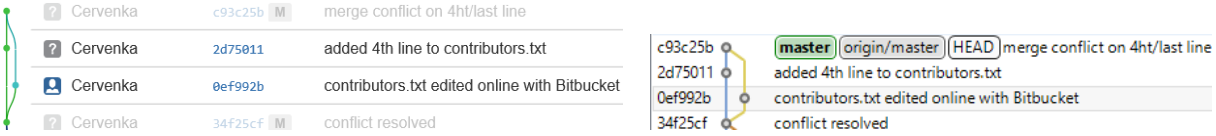
7)git commit –m "merge conflict on 4ht/last line" -a

Tím ručně spustím poslední příkaz, který původně měl provést merge (i komentář zadám stejný). Proto z promptu zmizí |MERGING, je zde zase jen /e/work (master).

Pozn.: pokud je v git commit parametr –a, pak git add není nutný a body 7a a 7b jsou zbytečné

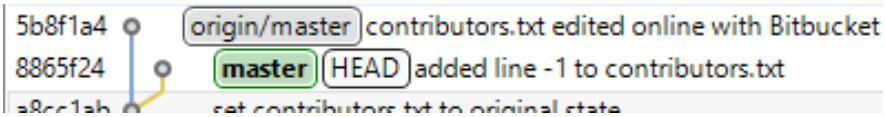
8)git push

9)Dopad Push na remote repository si ověřím na webu BitBucketu, vpravo obr. z Eclipse:

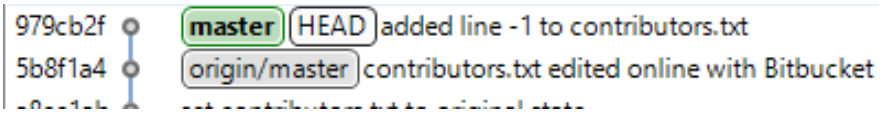


Rebase (automatické)

Syntaxe příkazu rebase je složitější než u merge. Musí se totiž ještě zadat, zda po linearizaci budou nejprve commity, které proběhly ve vzdálené repository a pak commity lokální (což je častější a budeme to tak trénovat) nebo naopak. Proto tento příkaz budeme trénovat v Eclipse, neboť tam pořadí určíme tím, že na kartě *History* zvolíme příkaz *Rebase on* z místní nabídky vrcholu větve remote (origin/master). Stav před rebase:



A po:

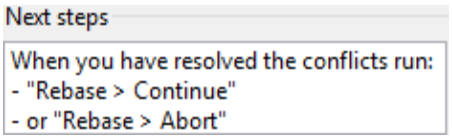


Zbytek postupu je stejný jako u merge.

Rebase s ručním řešením konfliktu

Po spuštění Rebase on se objeví dialogové okno s informací Rebase was stopped due to 1 conflicting files.

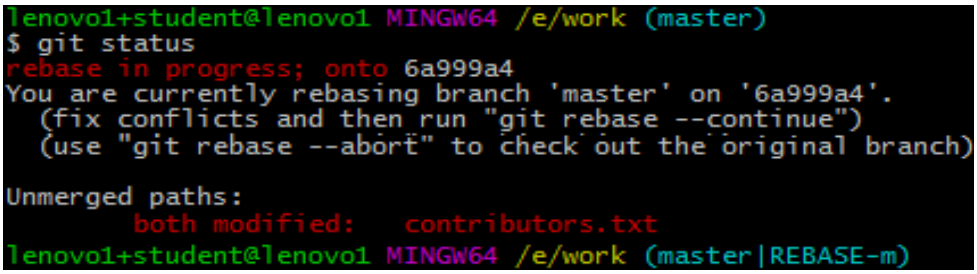
a s informací, co dělat po odstranění konfliktu:



Dialogové okno uzavřeme tlačítkem *OK*.

Podíváme se na kartu *Git staging*, tlačítko *Continue* je šedě.

Podíváme se, co hlásí git status:



(Dole je červeně: `both modified: contributors.txt`)

Otevřeme soubor contributors.txt a vyřešíme konflikt

Zadáme git add contributors.txt, tím systému git sdělíme, že jsme konflikt vyřešili. Git reaguje tím, že zpřístupní tlačítko *Continue* na kartě *Git staging*.

Pro kontrolu si vypíšeme i git status:

```
rebase in progress; onto 6a999a4
You are currently rebasing branch 'master' on '6a999a4'
  (all conflicts fixed: run "git rebase --continue")

Changes to be committed:
        modified:   contributors.txt
```

Pozn.: Tedy po git add se v tomto případě nedává git commit

Klepneme na tlačítko *Continue* nebo napíšeme git rebase --continue.

Ověříme linearizaci commitů. Na závěr git push.

Pozn.: Pozor: všichni členové týmu by měli mít v Eclipse stejně nastavené, zda se budou při ukládání tabelátory měnit na mezery. Také zda budou { na samostatném řádku. Jinak totiž bude odlišnost na každém řádku:

Nedoporučuje se použít merge v situaci, kdy víme, že některý soubor má mnoho netriviálních odlišností. Když dojde ke konfliktu, tak se merge pozastaví. Ale předchozí automatické merge zůstávají v platnosti. Pokud zjistím, že konfliktů je tolik, že je nelze ručně vyřešit (například mám jinak velké tabulátory nebo nastavenu záměnu tabelátorů za mezery při uložení), pak je nutné celý merge vzít zpět. Provede se to příkazem git merge --abort. Nebo git rebase –abort

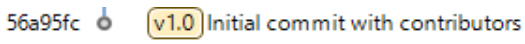
Nebo na kartě *Git staging* klepnu na tlačítko *Abort*.

Pozn.: pokud chci kvůli nevhodné záměně tabulátorů za mezery svou verzi nahradit verzí z remote repository, pak přesunu soubor zpět nahoru do panelu Unstaged changes a pak z místní nabídky tohoto souboru zvolím *Replace with HEAD revision*. Pak zmizí i z tohoto panelu

Pozn.: další možnost zjednodušení historie: lze seskupit více commitů do jednoho (pravděpodobně jen dokud se nedá push). Říká se tomu squash, realizuje se to pomocí rebase, nějak jako git rebase –i head

Tagy

V Eclipse na kartě *History* lze z místní nabídky nějakého řádku vybrat *Create tag*. Například úplně prvnímu commitu přiřadíme tag v1.0.



`git tag`: vypsání všech použitých tagů. Chceme-li vypsat i číslo commitu, musíme jednotlivě:

```
$ git tag
v1.0

lenovo1+student@ler
$ git tag -v v1.0
object 56a95fcecb2c
```

Pozn.: tagy se na remote repository nepřenášejí automaticky (tedy ostatní spolupracovníci je nevidí), musíme zadat např.:

```
git push origin v1.0
```

To platí i pro grafické nástroje, V git-gui musíme po klepnutí na tlačítko *Push* zatrhnout v dialogovém okně volbu *Include tags*. A v Eclipse zase po *Push to Upstream* musíme v dialogovém okně klepnout na tlačítko *Advanced* a v dalším dialogovém okně klepnout na tlačítko *Add All Tags Spec*.

Checkout – návrat k staré verzi

Nejčastěji se vracíme k commitům, které byly nějakou produkční verzí, neboť právě tyto verze mívají zákazníci (a ti někdy hlásí chybu). A protože se jednodušeji vrací k tagu např. v1.0 než k sedmimístnému číslu commitu, tak je vhodné ke commitu, který je produkční verzí, přiřadit tag s názvem této verze (ale obecně lze commitu přiřadit jakýkoliv řetězec).

Checkout lze vybrat z místní nabídky libovolného řádku commitu. Nebo z příkazového řádku napsat git checkout v1.0. Pokud danému commitu není přiřazen tag, tak: git checkout 56a95fc. A tím dostaneme do working copy všechny soubory v této verzi. Aktuální soubory se neztratí, později se k nim vrátím díky local repository. Podmínkou samozřejmě je, že před použitím příkazu checkout lokální repository aktualizuji (add a commit).

Pokud takto přejdeme do verze 1.0, tak se na dané řádku objeví HEAD, což znamená, že všechny soubory ve working copy jsou nyní v tomto stavu:

Podobně je to v gitk, místo HEAD je zde žlutý kroužek:  Initial commit with contributors

Ještě ověření git status:

```
$ git status
HEAD detached at 56a95fc
nothing to commit, working tree clean

lenovo1+student@lenovo1 MINGW64 /e/work ((v1.0))
```

(pod git status je červeně: `HEAD detached at`)

Pak se Průzkumníkem podíváme na složku work, je tam jen soubor contributors.txt (soubor text1.txt tam není) a v něm jen jeden řádek, tak jak byl po založení BitBucketu.

Zpět se vrátíme zase checkoutem na nejvyšším commitu.

Větvení

Větve se v projektu tvoří tím, že příkazem fetch stáhnu z remote repository stav, který je odlišný od stavu lokální repository (a není totožný ani s žádným starším commitem). Takovouto větev se obvykle snažíme hned spojit s hlavní větví (nebo ještě lépe použít rebase).

Někdy však větve vytváříme úmyslně, například větve pro různé zákazníky. Takové se s hlavní větví spojují až mnohem později nebo se nespojují vůbec.

Větve jsou také nezbytné při opravování starších chyb. Opravenou verzi nemůžeme na původním místě nějak „refreshnout“, to by bylo nutné tuto úpravu přidat do všech navazujících verzí. A to není možné už z toho důvodu, že kdyby zákazník s nějakou pozdější verzí hlásil, že má chybu na řádku 55, tak by to nesedělo. Programátor z místní nabídky příslušného commitu na kartě *History* vybere *Create branch*. Tím se současně provede checkout. Pak programátor provede opravy a zadá commit. A na konci oprav musí konec této větve spojit s koncem hlavní větve.

Přepínání mezi větvemi lze provést i v git bash:

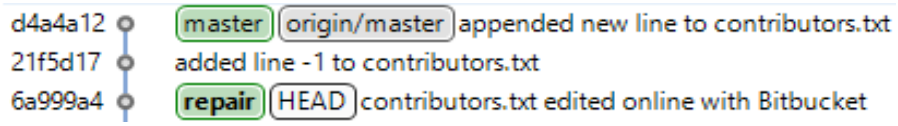
`git branch` vypíše všechny větve, zatím jen master

`git checkout newBranch`

Od té chvíle je na konci řádků (newBranch)

```
$ git branch
* master
  newBranch

lenovo1+student@lenovo1 MINGW64 /e/work (master)
$ git checkout newBranch
Switched to branch 'newBranch'

lenovo1+student@lenovo1 MINGW64 /e/work (newBranch)
$ git branch
  master
* newBranch

lenovo1+student@lenovo1 MINGW64 /e/work (newBranch)
$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.

lenovo1+student@lenovo1 MINGW64 /e/work (master)
```

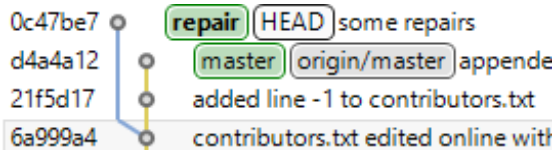
Mějme situaci, kdy na třetím commitu od vrchu chci provést nějaké opravy. Z místní nabídky daného commitu na kartě *History* vyberu *Create branch*. Pojmenuji repair. Automaticky se provede i checkout a přepnutí to této větve, ověříme na kartě *History* i v git bash:



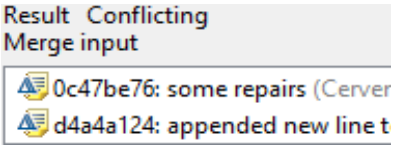

```
$ git status
On branch repair
nothing to commit, working tree clean

lenovo1+student@lenovo1 MINGW64 /e/work (repair)
```

Poté provedeme nějakou opravu v souboru contributors.txt a zadáme `git commit –m "some repairs" –a`

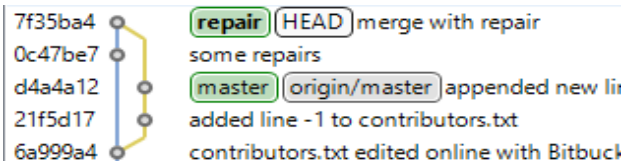
Pozn.: nicneříkajícím komentářem „some repairs“ by kolegové nebyli moc nadšeni...



Provedeme *Merge*, objeví se hlášení o konfliktu:

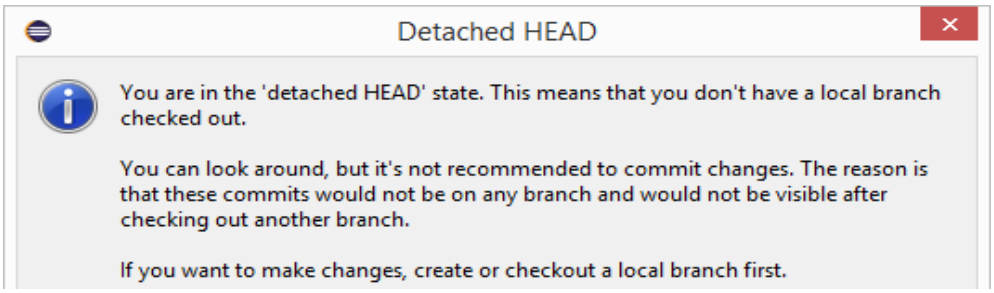


V prostředí Eclipse editujeme contributors.txt a zadáme `git commit –m "merge with repair" -a`



Vidíme, že se mi větve spojily tak, že dál pokračuje větev repair. Já bych radši chtěl, aby dál pokračovala větev master. Co jsem udělal za chybu? Měl jsem ještě před Merge provést checkout na vrchol větve master

Pozn.: pokud bych provedl checkout do daného commitu bez vytvoření větve, pak by to bylo vhodné jen pro prohlížení, ne pro úpravy. Protože úpravy by nebyly součástí žádné pojmenované větve, proto by později byly nedostupné. Proto by se po checkout objevilo následující hlášení.



Pokud chci v daném místě program rozvětvit, například založit větev, která bude řešit bug 1568 nebo request106, tak je dobré tak tuto větev nazvat (např. req106). Větev se vytvoří:

git branch req106

a přejde se na ni (pak je v závorce na konci promptu):

git checkout req106

Nebo jedním příkazem git checkout –b req106

Poznámky:

`git remote` výpis všech vzdálených repository, které používám, minimálně se vypíše origin.

Lektorovi, který si naklonoval více remote repository studentů, se zobrazí více. Aby měl u toho přehledně i cestu (tu, kterou mu student předtím poslal mailem), použije parametr –v (verbose). Díky tomu pak může rychle stáhnout nové verze od všech uživatelů, aniž by musel hledat příslušný mail.

`git remote show origin` (nebo jiné remote repository) vypíše názvy větví na daném remote repo, jedna z nich bude obvykle master. Poslední příkaz následujícího obrázku ukazuje, že konfigurace pro pull (tedy fetch) a push je taková, že když neuvedeme parametry odkud-kam, tak se předpokládá propojení lokální větve master a toutéž větví v origin.

```
lenovo1+student@lenovo1 MINGW64 /e/work (master)
$ git remote
origin
lenovo1+student@lenovo1 MINGW64 /e/work (master)
$ git remote -v
origin  git@bitbucket.org:cervenkapetr/java_training.g
lenovo1+student@lenovo1 MINGW64 /e/work (master)
$ git remote show origin
  Remote branch:
    master tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local ref configured for 'git push':
    master pushes to master (up to date)
```

Pozn.: pokud máme více větví, používá se git fetch --all nebo git push --all

Práce s reálným projektem

Do lokální repository (je v ní složka .git) nakopíruji nějaký projekt, na kterém mám spolupracovat. Bude zde kompletní projekt, tedy složky bin, src, test a soubory .classpath a .project). Git status zobrazí vše červeně. Jsou v sekci untracked, tedy Gitem nesledované. Ověřím i na kartě *Git Staging* v Eclipse. Aby Git začal soubory ve složce src sledovat, tak zadám git add src/*

Když pak dám znovu git status, tak se soubory ze složky src zobrazí žlutě, budou v sekci Changes to be comitted. Ostatní zůstanou červené.

Zkopírujeme do složky soubor .gitignore pro Javu. Ověříme si, že se soubory v .gitinit zmíněné, přestanou v sekci Untracked zobrazovat.

Vyzkoušejte si do souboru .gitignore z Javy přidat i obsah souboru . gitignore pro Eclipse.. Ověřte si, že se v sekci Untracked přestanou zobrazovat soubory .classpath a .project. Pokud ale má daný projekt kontrolovat i lektor, pak zase vraťte soubor .gitignore do stavu jen pro Javu.

Soubor .gitignore

Soubory, které jsou výsledkem kompilace, nechceme dávat do gitu, dokonce ani nechceme, aby se vypisovaly v seznamu souborů, které nejsou sledovány. Např. v jazyku C to jsou soubory .obj, v Javě soubory *.class a *.jar, v C# soubory .exe. K tomu slouží soubor .gitignore, který uložím na kořen projektu. Vhodný soubor si stáhneme z webu Githubu,

https://github.com/github/gitignore

Odtud stáhnu verzi pro svůj jazyk, a pokud pracuji v Eclipse, tak i pro Eclipse. Ten pak ignoruje smetí, které vytváří Eclipse: podsložky .metadata, .recommenders, bin a .settings a soubory .classpath a .project. Zvlášť není dobré dávat smetí z Eclipse do Gitu tehdy, pokud některý z kolegů pracuje v NetBeans. Při spolupráci v týmu je vhodné stáhnout si template java.gitignore i Global/Eclipse.gitignore. Při posílání lektorovi ale použiji jen java.gitignore. Tedy soubory speciální pro eclipse tam nechat, aby si je tam nemusel kopírovat (jinak by se mu program nerozběhl). Ale např. .class soubory nepotřebuje, ty se vytvoří kompilací.

kniha

http://git-scm.com/book/cs/v1
tutoriál:

http://www.itnetwork.cz/software/git