Git aneb správa verzí trochu jinak

Jan Kasprzak

E-mail: kas@fi.muni.cz

Klíčová slova: Version control sytems, distributed source code management, Git, Linux

Abstrakt

V oblasti systémů pro údržbu a správu verzí zdrojových kódů (SCM) došlo v posledních letech k bouřlivému vývoji, který je charakterizován přechodem od centralizovaných systémů typu CVS (a později Subversion) k plně distribu- ovaným systémům. Systém Git, představený v tomto příspěvku, nepatří mezi nejstarší mezi podobnými systémy, nicméně v mnohém se těmi staršími nechává inspirovat, a naopak v jiných případech si jde svou vlastní cestou. V příspěvku se zaměříme hlavně na vnitřní fungování Gitu, a z toho pak vyplynou možné způsoby použití.

Abstract

In the last few years, there has been an intense development in the area of source code management systems (SCM). This can be characterized mainly by the movement away from the centralized systems like CVS (and later also Subversion) to fully distributed systems. The Git system, introduced in this paper, is not the oldest amongst the similar systems, but it is inspired by the older systems in many ways, but it also uses its own way of doing things in several other cases. In the paper we will focus on the internals of Git, and this will lead us to possible use cases.

1 Úvod

S pojmem *verzovací systém*[1] se setkal asi každý, kdo někdy pracoval poblíž vývoje softwaru. Pod tímto pojmem rozumíme software, do kterého lze ukládat verze zdrojových textů tak, aby bylo možno se případně v budoucnu mohli vrátit k některé starší verzi. Nejprve základní pojmy z oblasti verzovacích systémů:

repozitář (repository): místo, kam si verzovací systém ukládá informace o tom, jak vypadaly starší verze, a další svoje metadata.

checkout: zkopírování některého z uložených stavů z repozitáře do lokálního souborového systému.

pracovní kopie: data vyexportovaná z repozitáře akcí checkout.

commit: akce uložení změn v pracovní kopii do repozitáře. Někdy se však tímto pojmem označuje i verze dat tímto procesem vytvořená.

Ukládání verzí je základní funkce SCM systému. Dále verzovací systémy obvykle poskytují následující vlastnosti:

větvení: z jednoho stavu v čase může vycházet více paralelních větví vývoje (např. stabilní a vývojová větev). Je nutno říct, že podpora slučování větví (merge) je ve většině systémů velmi slabá.

komentáře: ke každé změně (*commit, changeset*) je možno připojit popis dané změny a případně odkaz na systém pro správu chyb.

paralelní přístup: většina systémů se nějakým způsobem snaží řešit přístup více vývojářů ke zdrojovým textům a případné zamykání.

háčky: skriptovatelné akce, vykonané v různých fázích práce s verzovacím systémem (například automatizované spuštění regresních testů při *commit*u do hlavní větve).

štítky: textové názvy verzí. Jednotlivé verze pak můžeme adresovat podle času vzniku, interního čísla verze ve verzovacím systému a se štítkem i podle štítku dané verze. Štítek může například odrážet fakt, že „tato verze byla zveřejněna jako 1.0.1“.

Verzovací systémy se mohou lišit v některých vlastnostech, jako například jestli podporují ukládání binárních nebo jen textových dat, jestli je možno ukládat metadata typu UNIXová přístupová práva, MIME typy souborů a podobně, jestli pracují nad jednotlivýcmi soubory (RCS, SCCS, CVS) nebo nad celým stromem adresářů (Subversion), detekcí porušení repozitáře, atd.

[1] alternativní pojmy: systém pro správu verzí, systém správy zdrojových textů, source code management system (SCM system), version control system (VCS)

2 Distribuované SCM systémy

2.1 Centralizované systémy

Až do poměrně nedávné doby měly všechny verzovací systémy *centralizovanou architekturu*. To znamená, že repozitář pro daný projekt byl na jednom serveru, ke kterému se jednotliví vývojáři připojovali, ať už s právem čtení (stahování verzí, výpis historie, atd.) nebo i s právem měnit data (právo *commit*). Vývojář si pak vždy stáhnul aktuální verzi na svůj stroj, provedl změny, a tyto pak promítnul do repozitáře.

2.2 Problémy centralizovaných systémů

Centralizovaný přístup s sebou nese několik problémů. Ten nejméně závažný je, že pro využití výhod verzovacího systému je třeba mít síťový přístup k repozitáři. Není tedy možno vzít si práci na chalupu a přitom kromě psaní programu dělat operace typu „zjisti kdo psal tenhle kus kódu“ nebo „oprav překlep v dokumentaci ve větvi, jejíž *checkout* jsem si neudělal předem“.

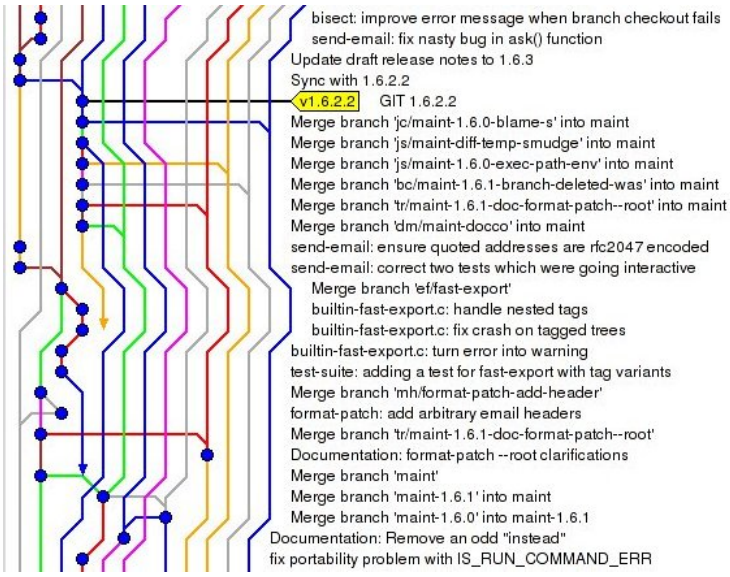
Tím se dostáváme k dalšímu problému: centralizované verzovací systémy nutí vývojáře, aby commit nedělali příliš často. Obvyklá praxe je, že vývojář udělá commit jen na konci pracovního dne. Jednotlivé commity ve verzovacím systému jsou pak čitelné jen jako „zkompilovatelné stavy projektu“, ale už ne jako logické k sobě patřící změny. V centralizovaném systému je poměrně těžké „odložit si“ právě rozpracovanou rozsáhlejší činnost a opravit triviální chybu nebo překlep ve formě samostatného commitu.

Asi největším problémem centralizovaných verzovacích systémů je překvapivě jejich centralizovanost. Centralizovanost znemožňuje například několika „okrajovým“ vývojářům bez práva zápisu do hlavního repozitáře dohodnout se a ad-hoc spolupracovat na nějaké části vývoje, včetně výměny svých částí kódu za pomoci verzovacího systému. Centralizované systémy tak přinášejí do projektů značnou míru politikaření a rozdělování světa na „core team“ s právem commit a „ty ostatní“, kteří mohou výhod verzovacího systému používat jen velmi omezeně.

2.3 Distribuované systémy

Distribuované SCM systémy představují kvalitativně novou úroveň práce s verzovacím systémem. Základní vlastností je neexistence jednoho centrálního repozitáře, do kterého mají přístup (někteří) vývojáři[2]. V distribuovaném verzovacím

[2]U většiny projektů samozřejmě „něco jako“ centrální repozitář existuje i nadále. Jeho status ale není dán funkčně (že by každý vývojář potřeboval mít k němu právo zápisu), ale deklarativně (je to *ten konkrétní* z mnoha možných a funkčně ekvivalentních).



Obrázek 1 Graf verzí v projektu Git

systému může existovat (a reálně také existuje) mnoho různých repozitářů. Na rozdíl od operace *checkout* u centralizovaných VCS je zde operace *clone*, která znamená vytvoření lokální kopie repozitáře, včetně veškeré historie.

Obvyklá metoda práce pak je vytvoření klonu nějakého repozitáře, práce uvnitř lokální kopie (s případným sdílením dat s jinými repozitáři) a zpřístupnění vlastních změn ve formě svého vlastního repozitáře, odkud si správce projektu (anebo kdokoli jiný) může dle svého uvážení změny vzít a použít.

Další rozdíl je pak ve způsobu nahlížení na data. Zatímco centralizované systémy považují jednotlivé verze za snímky dat v určitém čase[3], distribuovaný systém uvažuje v intencích posloupností *změn v datech* (angl. *changesets*). Celý vývoj pak je zobrazitelný nikoli ve formě posloupnosti, ale orientovaného acyklického grafu. Jednotlivé verze jsou pak charakterizovány nikoliv svým obsahem, ale seznamem změn (obvykle oproti společnému předkovi), které byly do dané verze zahrnuty. Příklad grafu změn v projektu s distribuovaným SCM je na obrázku 1. Komunikace mezi jednotlivými repozitáři může být jak nativní – prostředky konkrétního verzovacího systému – tak i obecná, například zasíláním souborů ve formátu programů *diff(1)* a *patch(1)*.

Významnou vlastností distribuovaných SCM systémů je, že typicky poskytují

[3] V Subversion dokonce lineárně uspořádané do posloupnosti.

širokou škálu metod čtecího přístupu k repozitáři. Repozitářem pak může být i libovolný statický adresář, zpřístupněný protokolem HTTP nebo FTP. Tímto je umožněno, že svoji vlastní práci může i s výhodami verzovacího systému zveřejnit opravdu každý, kdo má aspoň nějaký HTTP server. Není třeba ani právo zápisu do centrálního repozitáře, ani provozování vlastního serveru pro verzovací systém, ani vytvoření „oficiálního“ projektu například na SourceForge. Kdokoli takto může snadno zveřejnit svoje úpravy libovolného jinde dostupného projektu se všemi výhodami použití verzovacího systému.

3 Systém Git

3.1 Historie

Operační systém Linux byl poměrně dlouho vyvíjen bez použití verzovacího systému. Hlavní správce vývoje – Linus Torvalds – dlouho nechtěl žádný verzovací systém, protože pole něj by tehdejší systémy jej spíše brzdily než by přinášely zlepšení. Až v roce 2002 Larry McVoy, autor verzovacího systému pro potřeby Sun Microsystems TeamWare, přišel s projektem verzovacího systému BitKeeper, který byl jedním z prvních plně distribuovaných systémů. Linus Torvalds pak začal BitKeeper používat pro vývoj jádra Linuxu. BitKeeper ovšem nebyl Open Source, což vzbudilo nevoli některých dalších vývojářů.

Přibližně ve stejné době započal vývoj nástroje GNU Arch, který byl prvním Open Source distribuovaným verzovacím systémem. Arch ovšem měl několik praktických omezení (zejména výkon, z uživatelského hlediska pak poměrně složitý způsob práce a složité pojmenovávání verzí a větví), takže jeho nasazení v reálných projektech bylo spíše sporadické.

Dalším významným distribuovaným verzovacím systémem byl Monotone, jehož autoři přišli s myšlenkou, která do značné míry řešila problematiku celosvětově jednoznačného pojmenovávání sad změn (changesets) v distribuovaném prostředí. Nápad spočíval ve využití kryptograficky silné hashovací funkce pro označení verze souboru nebo i stromu souborů. Pouhým spočtením SHA1 součtu souboru můžeme celosvětově jednoznačně adresovat jeho obsah.

V dubnu 2005 Larry McVoy ukončil licenci BitKeeperu pro vývoj jádra Linuxu (důvodem bylo to, že autor Samby Andrew Tridgell zveřejnil výsledky svých pokusů o reverzní inženýrství síťového protokolu BitKeeperu) a Linux se ocitl zpět v bodě nula. Linus Torvalds se rozhodl na několik týdnů přerušit vývoj Linuxu a věnovat se vytvoření verzovacího systému. Torvalds na základě zkušeností s BitKeeperem a na základě informací o dalších verzovacích systémech navrhl systém Git[1]. I když z původní implementace toho v současném Gitu mnoho nezbylo, základní architektura, zajišťující extrémně rychlou odzevu, se používá dodnes.

4 Architektura Gitu

V následující kapitole si ukážeme principy, na jakých Git staví. Výklad je volně inspirován článkem [2]. Není to uživatelská příručka ani rychlokurz. Pro tyto typy dokumentace odkazujeme na [3], resp. [4].

4.1 Konfigurace

Pro práci s Gitem je vhodné, aby u změn byl uveden smysluplný autor. Toho dosáhneme příkazem `git-config(1)`, například:

```
$ git config --global user.name 'Jan "Yenya" Kasprzak'
$ git config --global user.email kas@fi.muni.cz
```

Git ukládá globální nastavení v souboru `.gitconfig` v domovském adresáři. Bez přepínače `--global` lze upravovat nastavení jen pro konkrétní repozitář.

4.2 Repozitář

Repozitář je v Gitu běžný adresář. Pokud vedle repozitáře mají být též vykopírovány zdrojové texty v nějaké verzi, říká se jim *pracovní kopie*. Repozitář pak má svoje data v podadresáři `.git` v kořeni pracovní kopie. Pokud pracovní kopii nepotřebujeme (například při zveřejnění repozitáře přes HTTP), jedná se o tzv. *holý repozitář* (*bare repository*), který je adresářem (obvykle) s příponou `.git`. Repozitář lze vytvořit buďto jako úplně nový (pro svůj vlastní projekt) nebo jako klon jiného repozitáře (pro úpravy existujícího projektu):

```
$ mkdir mujprojekt; cd mujprojekt; git init [--bare]
$ git clone [--bare] http://server/projekt.git; cd projekt
```

4.3 Blob, tree, commit

Git je v podstatě obsahově adresovatelná databáze souborů. Každý soubor je identifikován SHA1 hashem svého obsahu a je označen pojmem *blob*. Hodnotu SHA1 hashe není třeba uvádět celou, stačí libovolný prefix, který hodnotu jednoznačně identifikuje v rámci repozitáře:

```
$ echo ahoj > pozdrav.txt
$ git hash-object pozdrav.txt
36a0e7cf17ffe584221529d45113d821e70764a9
$ git add pozdrav.txt
$ git cat-file -p 36a0e7cf17
ahoj
```

Jak vypadal daný projekt v určité verzi je popsáno v objektu typu *tree*. Tento objekt obsahuje jména souborů které v daném čase do projektu patřily a SHA1 hashe jejich obsahů v dané verzi.

```
$ git write-tree
f1d1e2642787c59d61f01075e44b780e60ede900
$ git cat-file -p f1d1
100644 blob 36a0e7cf17ffe584221529d45113d821e70764a9 pozdrav.txt
```

S objekty typu tree obvykle uživatel nepřichází do styku přímo. Jednotlivé verze jsou identifikovány objekty typu *commit*. V nich se nachází odkaz na verze souborů (objekt typu tree), odkaz na rodičovský commit (nebo commity) a další metadata.

```
$ git commit -m 'Prvni verze'
Created initial commit f2ebc71: Prvni verze
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 pozdrav.txt
$ git cat-file -p f2eb
tree f1d1e2642787c59d61f01075e44b780e60ede900
author Jan "Yenya" Kasprzak <kas@fi.muni.cz> 1240789610 +0200
committer Jan "Yenya" Kasprzak <kas@fi.muni.cz> 1240789610 +0200

Prvni verze
```

Objekt typu commit v sobě již zahrnuje mimo jiné čas vytvoření, takže jeho SHA1 hash se na rozdíl od výše uvedených bude na vašem počítači lišit, i když budete výše uvedené příkazy zadávat doslova. První commit nemá žádného rodiče. V dalších je ale SHA1 hash rodiče nebo rodičů zmíněn:

```
$ echo nazdar >> pozdrav.txt
$ git add pozdrav.txt
$ git commit -m 'Pridan pozdrav "nazdar".'
Created commit b400b96: Pridan pozdrav "nazdar".
 1 files changed, 1 insertions(+), 0 deletions(-)
$ git cat-file -p b400b96
tree f3424e549969ff3293ddb04970bde3128c836887
parent f2ebc71c3ca437c668c99731f6ae13efe1cb2f18
author Jan "Yenya" Kasprzak <kas@fi.muni.cz> 1240790025 +0200
committer Jan "Yenya" Kasprzak <kas@fi.muni.cz> 1240790025 +0200

Pridan pozdrav "nazdar".
```

Tímto SHA1 hash commitu jednoznačně identifikuje nejen commit samotný coby sadu změn proti rodiči, ale (protože commit zahrnuje i SHA1 hashe rodičů) celou historii zpětně až ke vzniku projektu. Toto je významná bezpečnostní vlastnost Gitu – není možno bez povšimnutí modifikovat zpětně historii. Navíc takto lze detekovat i porušení integrity repozitáře.

4.4 Větve, štítky

Jednou z nejsilnějších vlastností Gitu je práce s větvemi. Uživatel si může snadno vytvářet větve (i dočasné), kopírovat mezi nimi změny a spojovat větve mezi sebou. Větev není nic jiného než pojmenovaná *hlava* (*head*), tedy commit který nemá potomky. Při commitu dalších změn do větve se odkaz přepíše na nově vzniklý commit. *Štítek* (*tag*) pak je textový odkaz na jeden pevný commit. Větve jsou uchovávány uvnitř repozitáře v podadresáři `refs/heads`, štítky v `refs/tags`. Implicitní větev se jmenuje *master*.

```
$ git checkout -b japonsky
Switched to a new branch "japonsky"
$ git branch
  master
* japonsky
$ echo konnichiwa >> pozdrav.txt
$ git add pozdrav.txt
$ git commit -m 'japonsky pozdrav'
Created commit ce1bbf5: japonsky pozdrav
 1 files changed, 1 insertions(+), 0 deletions(-)
$ git checkout master
Switched to branch "master"
$ cat pozdrav.txt
ahoj
nazdar
$ git merge japonsky
Updating b400b96..ce1bbf5
Fast forward
 pozdrav.txt |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
$ cat pozdrav.txt
ahoj
nazdar
konnichiwa
$ ls .git/refs/heads/
master japonsky
```

```
$ git branch -d japonsky
```

5 Další vlastnosti Gitu

Na výše popsané architektuře Git staví poměrně silné uživatelské rozhraní. Není cílem v tomto článku popsat typické použití Gitu. V této kapitole zmíníme některé vybrané vlastnosti a principy, kterými se Git odlišuje od jiných podobných systémů:

5.1 Produktem je záplata

Jednou ze základních myšlenek Gitu je, že produktem práce programátora nemá být pouhý nový nebo opravený kus programu, ale že tímto produktem má být „záplata“, tedy rozumně vytvořený, zdokumentovaný a logicky izolovaný popis změny oproti předchozí verzi. S těmito záplatami je pak možno zacházet různě, například použít jen některé, měnit jejich pořadí a podobně. Tomuto cíli Git podřizuje i styl práce. Například v Gitu není historie cosi neměnného. Koneckonců jde stále o lokální soubory na programátorově disku. Čili dokud programátor nepublikuje svoji práci veřejně, měl by mít možnost upravovat svoje commity dle libosti.

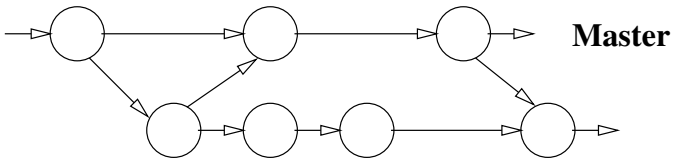
Častá je například situace, kdy programátor commituje novou funkci, ale zapomene ji popsat v manuálu. Git umožňuje pomoci příkazu `git-commit --amend` editovat poslední commit: přidávat do něho další změny, upravovat popis, a podobně. Nová verze commitu má pochopitelně nový SHA1 identifikátor, ale u dosud nepublikovaných commitů tohle nijak nevadí. Git umožňuje podobným způsobem editovat i starší commity.

Git má podporu pro export například celé větve jako sady záplat, poslání této sady e-mailem a na druhé straně aplikování několika mailů jako záplat do zadané větve.

5.2 Bisect

Máme-li jednotlivé úpravy jako logicky oddělené sady změn (a ne například sady změn typu „Moje práce za ten a ten den“), je i snadnější odhalování chyb na straně uživatele. Nástrojem `git-bisect(1)` je možno označit si která verze je funkční a která nefunkční, a Git pomocí půlení intervalu[4] umožní vytvářet mezilehlé verze, které pak uživatel otestuje a označí, jestli tato verze funguje

[4] Pojem „půlení intervalu“ je zde použit jen velmi volně. Připomínám že distribuovaný verzovací systém nemá změny jako striktně uspořádanou posloupnost, ale jako acyklický orientovaný graf.



Obrázek 2 Spojování větví – merge

nebo ne. S jistou dávkou štěstí pak může uživatel dospět k jedinému commitu, který daný problém způsobuje.

5.3 Staging area

Pozorný čtenář znající i jiné verzovací systémy si jistě povšiml, že ve výše uvedených příkladech bylo použito příkazu `git-add` častěji, než by očekával. Je to kvůli vlastnosti Gitu, kterou se tento systém odlišuje od jiných verzovacích systémů: *staging area*, někdy též méně výstižně nazývaná *index*.

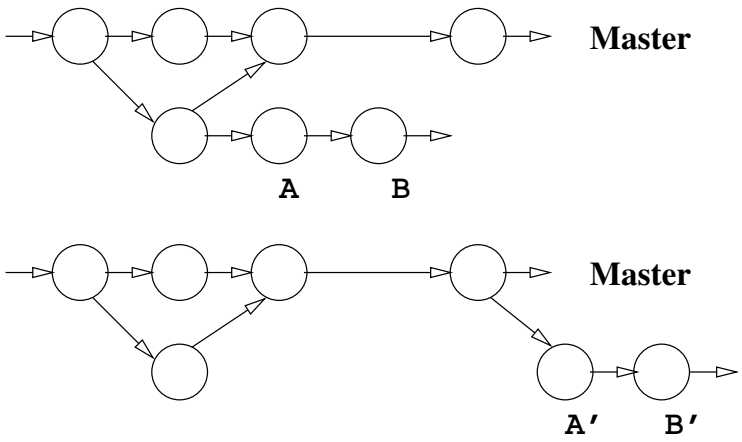
Příkaz `git-add(1)` zde má odlišný význam od příkazů `add` z jiných verzovacích systémů. Tam tato akce v podstatě znamená „začni sledovat tento soubor verzovacím systémem“. V Gitu tato akce znamená „do příštího commitu zahrň tento soubor s právě tímto obsahem“. Git kromě pracovních (vyexportovaných) dat a repozitáře již potvrzených (commit) dat má ještě třetí typ dat, a to staging area – data naplánovaná pro příští commit. Takto si programátor může commit konstruovat postupně po jednotlivých souborech. Do commitu tedy nejsou automaticky zahrnuty všechny modifikované soubory z pracovní oblasti, ale jen ty explicitně označené.

Pro uživatele přecházející z jiných verzovacích systémů má nicméně Git příkaz `git commit -a`, který dělá totéž co `commit` v jiných verzovacích systémech, tedy potvrzuje všechny změněné soubory.

5.4 Merge a rebase

Většina verzovacích systémů podporuje v nějaké formě slučování větví (merge). V Gitu je slučování větví poměrně jednoduché – hledají se všechny commity které jsou potomky společného rodiče větví a jsou předky těch hlav větví, které se právě slučují. Tyto se pak zahrnou do výsledných dat, jak je naznačeno na obrázku 2.

Mnohdy je však žádoucí svoje změny vystavit v podobě pro uživatele čitelnější, a to jako změny proti aktuální oficiální verzi (nazvěme ji třeba *master*). Git toto řeší pomocí akce *rebase*. Při rebase Git vezme změny z dané větve (které ještě



Obrázek 3 Spojování větví – rebase

nejsou obsaženy v master větvi) a vytvoří z nich nové commity se stejným obsahem, které ale naváže jako potomky hlavy větve master. Nové commity mají pochopitelně jiný SHA1 hash, protože se zejména liší jejich rodič. Situace je naznačena na obrázku 3.

5.5 Stash

Chce-li vývojář mít nejen čitelný kód, ale i smysluplné a logicky oddělené commity, narazí občas na situaci, kdy by měl opravit triviální chybu v situaci, kdy má rozdělanou nějakou větší práci. Git pro tyto účely poskytuje nástroj *stash*:

```
$ vi pozdrav.txt # rozpracovana vetsi vec
$ git stash
Saved working directory and index state
"WIP on master: ce1bbf5... japonsky pozdrav"
HEAD is now at ce1bbf5 japonsky pozdrav
(To restore them type "git stash apply")
$ ... oprav trivialni chybu ...
$ git commit ...
$ git stash apply # vracim se k rozpracovane veci
```

Stash umožňuje mít i více takto rozdělaných a odložených prací. Nicméně u více rozpracovaných déletrvajících činností je spiš doporučeno dávat každou takovou činnost do samostatné větve.

6 Závěr

Mezi distribuovanými verzovacími systémy zaujímá Git významnou pozici. I když historicky je mu vyčítána slabší úroveň dokumentace, v posledních letech se tento problém definitivně odsunul do říše *urban legends*. Některé vlastnosti jako je staging area mohou uživatelům jiných systému připadat neintuitivní, ale při častějším používání se stanou neocenitelnými pomocníky. Nedávné oznámení migrace několika velkých projektů (Perl, GNOME, Wine, X.org, ...) na Git činí z tohoto systému daleko nejrozšířenější distribuovaný systém správy verzí na světě. Budete-li začínat s novým projektem nebo uvažovat o změně verzovacího systému stávajícího projektu, zahrňte Git do svých úvah.

Reference

[1] *Domovská stránka projektu Git*
`http://git-scm.com/`

[2] *Wiegley, John: Git from the Bottom Up*
`http://www.newartisans.com/blog_assets/git.from.bottom.up.pdf`

[3] *Git User's Manual*
`http://www.kernel.org/pub/software/scm/git/docs/user-manual.html`

[4] *Git Tutorial*
`http://www.kernel.org/pub/software/scm/git/docs/gittutorial.html`