

Technologia Programowania 2017/2018
Wykład 10

Systemy kontroli wersji, Git

Jakub Lemiesz

Kontrola wersji

- ▷ Chcemy śledzić wszystkie zmiany w projekcie, by
 - ① móc przywołać dowolną wcześniejszą wersję (bezpieczeństwo)
 - ② móc porównać zmiany, sprawdzić, kto, kiedy i dlaczego je wprowadził (historia)
 - ③ móc łączyć zmiany równolegle wprowadzane przez wiele osób (zarządzanie zmianami)
- ▷ Alternatywa? Można robić kopie do katalogów oznaczonych datami i wysyłać pocztą...

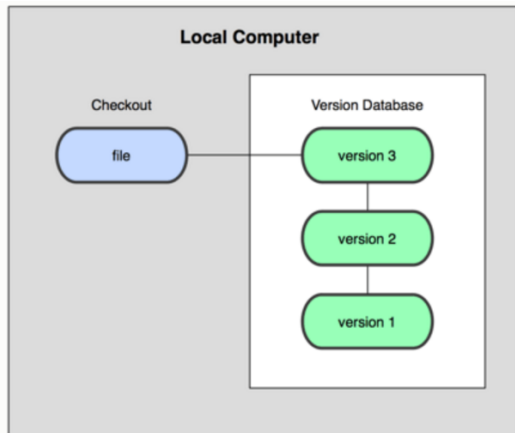
Architektura systemów kontroli wersji

▷ Systemy kontroli wersji:

- 1 **lokalne**, pozwalające na zapisanie danych jedynie na lokalnym komputerze (np. RCS)
- 2 **scentralizowane**, oparte na architekturze klient-serwer (np. CVS, Subversion)
- 3 **rozproszone**, oparte na architekturze P2P (np. Git, BitKeeper)

Lokalne systemy kontroli wersji

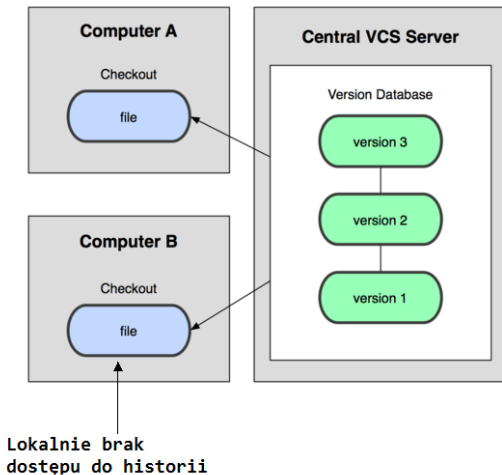
- ▷ Zapis danych różnicowych: pamiętamy różnice pomiędzy kolejnymi wersjami **jednego** pliku
- ▷ W Unix np. Revision Control System (RCS, 1982)



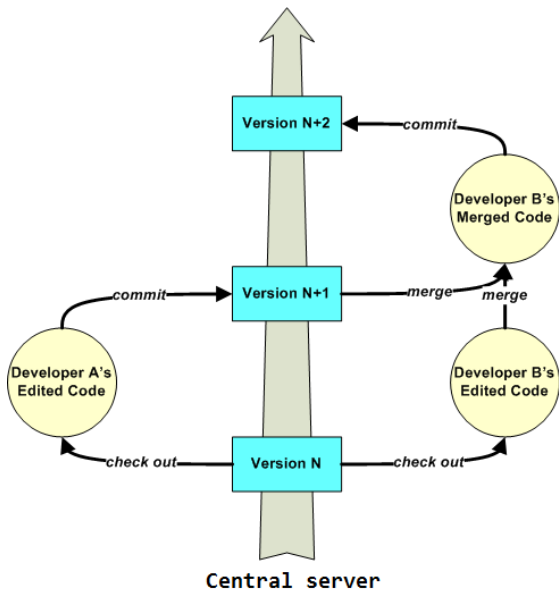
Scentralizowane systemy kontroli wersji

- ▶ Duże projekty, potrzeba współpracy dużych zespołów ⇒ **Centralized Version Control System**
- ▶ Jeden serwer centralny, który zawiera wszystkie pliki poddane kontroli wersji i z którym każdy się synchronizuje (*lokalnie brak dostępu do historii*)
- ▶ Np. **CVS** (na bazie RCS, 1990), **SVN** (2000)
(*SVN wprowadza m.in. atomowe transakcje*)
- ▶ W skrócie: **checkout**, **commit**, **update**, **merge**

Scentralizowane systemy kontroli wersji



Scentralizowane systemy kontroli wersji



Systemy scentralizowane a rozproszone

▷ Systemy scentralizowane:

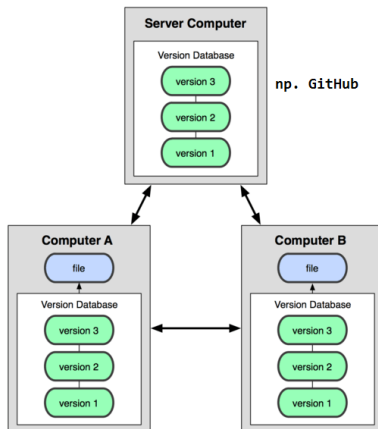
- ❶ awarii centralnego serwera, oznacza czasowy brak możliwości współpracy
- ❷ awaria dysku na centralnym serwerze może oznaczać utratę całej historii projektu
- ❸ kłopotliwa współpraca w dużych zespołach

▷ Systemy rozproszone:

każdy ma lokalny dostęp do całego repozytorium
(*to ma swoje zalety, ale też wady*)

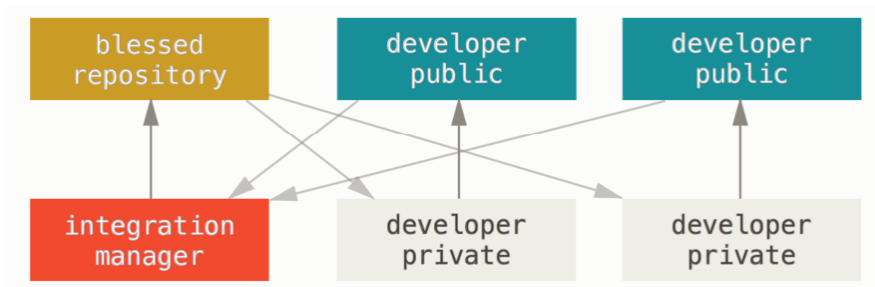
▷ Warto przejrzeć <https://svnvs-git.com/>

Rozproszone systemy kontroli wersji



- ▷ Teoretycznie żaden węzeł nie jest wyróżniony
- ▷ DVCS możemy używać jak CVCS (wyróżniamy 'serwer')
- ▷ To się sprawdza jeśli mamy mały zespół, ale możliwe są inne sposoby organizacji pracy np. "integrator", "dyktator"

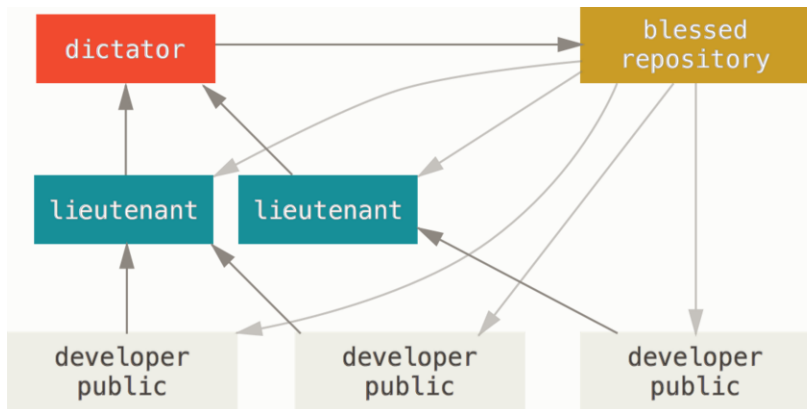
DVCS – integrator

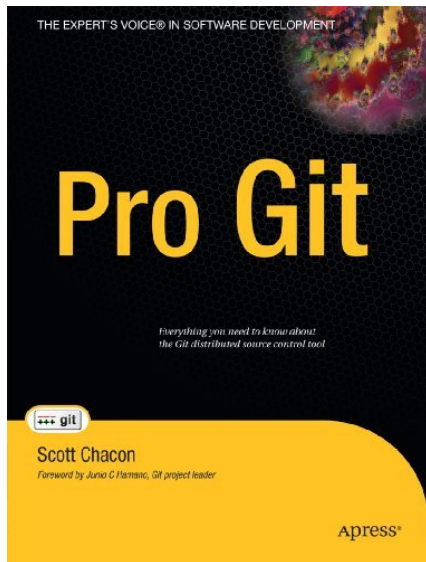


DVCS – integrator

- 1 Opiekun/integrator projektu wprowadza zmiany do głównego (publicznego) repozytorium
- 2 Programiści pobierają aktualną wersję projektu i lokalnie wprowadzają zmiany
- 3 Programista wprowadza zmiany do swojego publicznego repozytorium
- 4 Programista wysyła prośbę do integratora, aby pobrał zmiany z jego repozytorium
- 5 Integrator kiedy uzna za stosowne pobiera zmiany, integruje i wprowadza do głównego repozytorium

DVCS – dyktator, dla dużych projektów (np. Linux kernel)





<https://git-scm.com/book/en/v2>

Historia Git w dużym skrócie

- ▶ 2002: projekt 'Linux kernel' zaczął używać DVCS BitKeeper (*tysiące programistów*)
- ▶ 2005: właściciel BitKeeper cofa pozwolenie na nieodpłatne używanie
- ▶ 2005: programiści pracujący nad jądrem Linuksa tworzą Git na bazie zdobytych doświadczeń (*silne wsparcie dla nieliniowego rozwoju, wiele równoległych gałęzi*)

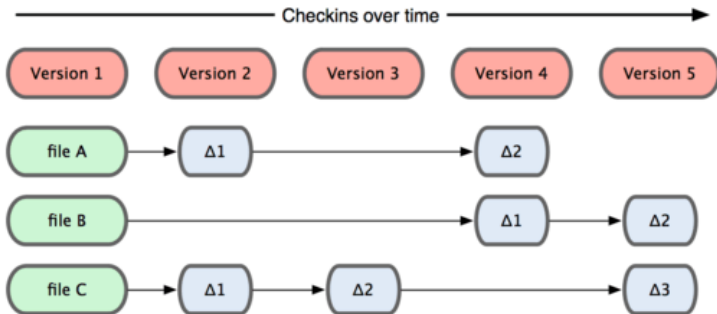
Git — instalacja

- ▷ Pobieramy i instalujemy Git
<https://git-scm.com/downloads>
- ▷ Na początek dobrze jest się posługiwać linią komend, choć istnieje wiele graficznych nakładek
- ▷ Jednorazowa konfiguracja (*home/.gitconfig*)
 - ❶ \$ git config --global **user.name** "JLemiesz"
 - ❷ \$ git config --global **user.email** me@pwr.pl
 - ❸ \$ git config --global **core.editor** 'ścieżka do edytora tekstu'
 - ❹ \$ git config --list

Tworzenie/klonowanie

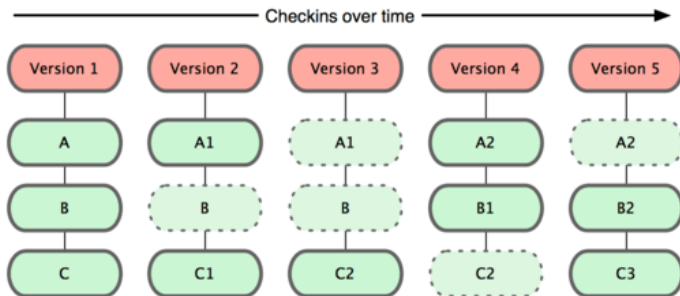
- ▷ \$ **git init** w wybranym katalogu, lub
- ▷ \$ **git clone** <https://github.com/kubal/wyklad.git>
- ▷ Patrzymy na zawartość katalogu **.git**
 - ❶ **hooks** - automatycznie wykonywane skrypty
(*np. zawsze przed commit*)
 - ❷ **objects** - zawartość bazy
 - ❸ **info/exclude** - jakich plików git ma 'nie widzieć'
 - ❹ **refs** - wskaźniki do commitów
(*o tym przy gałęziach*)
 - ❺ **HEAD** wskaźnik do aktualnej gałęzi
(*o tym przy gałęziach*)

Przechowywanie informacji w większości VCS



Większość systemów kontroli wersji (np. SVN) przechowuje informacje jako zbiór plików i zmiany dokonane na każdym z nich w okresie czasu

Git — przechowywanie informacji



- ▶ Git: commit tworzy migawkę (ang. snapshot) pamiętającą jak wyglądają wszystkie kontrolowane pliki w danym momencie i przechowuje referencję do tej migawki
- ▶ By poprawić wydajność, jeśli dany plik nie był zmieniony, nie zapisujemy go ponownie, a tylko referencję do jego poprzedniej, identycznej wersji, która jest już zapisana

Katalog **objects**, czyli jak Git pamięta zmiany

- 1 \$ git hash-object -w test.txt
(hash z zawartości pliku: 40 'cyfr szesnastkowych')
- 2 W **objects** zostanie **zapisany** plik, podkatalog 2 pierwsze znaki hashu, nazwa pliku pozostałe 38
- 3 \$ git cat-file -p 'wygenerowany hash'
(pokazuje zawartość pliku)
- 4 Zmieniamy zawartość pliku test.txt, powtarzamy krok 1 i patrzymy co się dzieje w **objects**
- 5 Standardowo Git wyłącznie dodaje nowe dane
(trudno jest zrobić nieodwracalną zmianę)

Ignorowanie plików

- ▶ Mamy już lokalne repozytorium, ale jeszcze żadnych commitów
- ▶ W katalogu roboczym polecenie `git status` pokazuje nam dostępne pliki
- ▶ Jeśli chcemy by pewne pliki nie były widoczne możemy wykorzystać plik `.git/info/exclude` (*np. pliki wynikowe lub tymczasowe*)
- ▶ Alternatywnie można też dodać do katalogu roboczego plik `.gitignore`

Git — jak ignorować określone pliki

```
.git/info/exclude
```

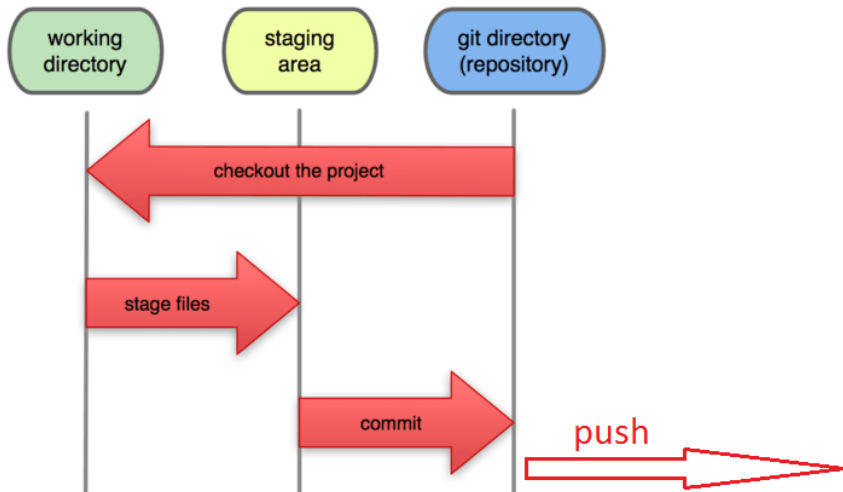
```
# no .a files
*.a
# but do track lib.a
!lib.a
# only ignore the test.txt file in the current
# directory, not subdir/test.txt
/test.txt
# ignore all files in the build/ directory
build/
# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt
# ignore all .pdf files in the doc/ directory
doc/**/*.pdf
```

Commit'owanie zmian

- 1 Mamy **lokalną bazę danych git'a** (katalog objects)
- 2 Możemy pobierać przez **checkout** z repo do **katalogu roboczego** jedną z wersji projektu lub stworzyć w katalogu roboczym nowe pliki
- 3 Dodajemy/modyfikujemy pliki i te które mają być przesłane do repo przy najbliższym commit dodajemy przez **git add** do **przechowalni** (*ang. staging area*)
- 4 Gdy odpowiednio dużo zmian **git commit**
- 5 Jeśli chcemy przesłać zmiany do zdalnego repozytorium robimy **git push**

Commit'owanie zmian

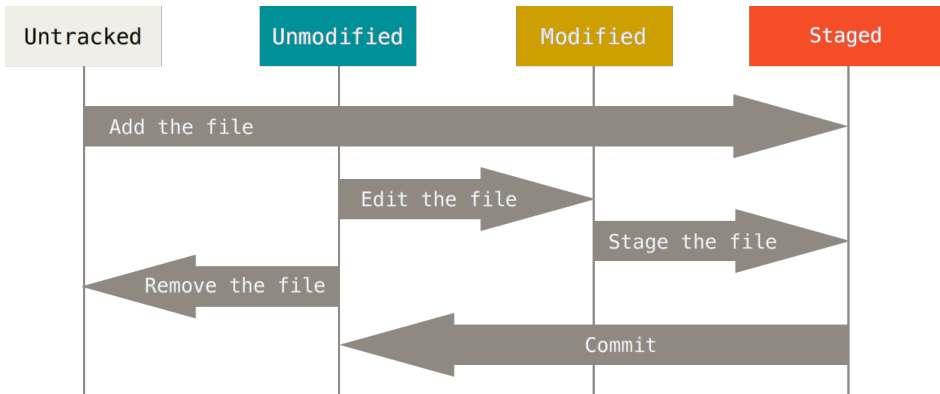
Local Operations



Operacje lokalne a stan pliku

- ▶ Plik w katalogu roboczym może być w stanie **tracked** albo **untracked** (oznaczenie ??)
- ▶ Jeśli jest w stanie **tracked** to musi być w jednym ze stanów:
 - ▶ **Unmodified** — to samo jest w lokalnej bazie
 - ▶ **Modified** — zmieniony, czyli co innego w bazie
 - ▶ **Staged** — zmieniony i zaakceptowany (do dodania do bazy w następnym commit)

Operacje lokalne a stan pliku

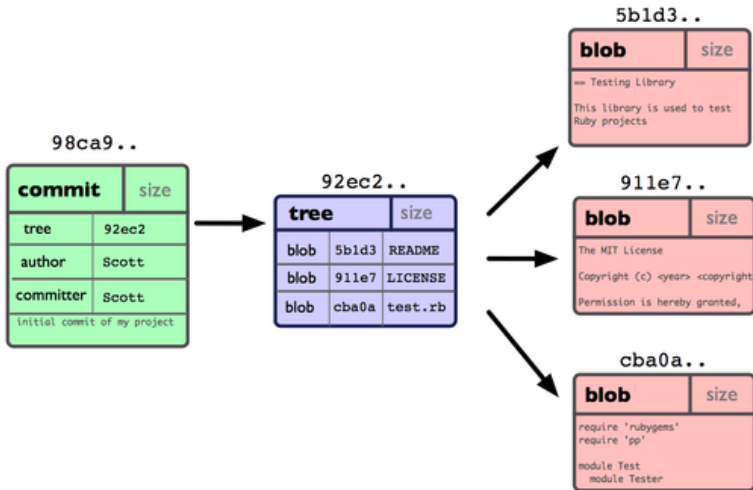


Weryfikacja stanu: `git status -s`

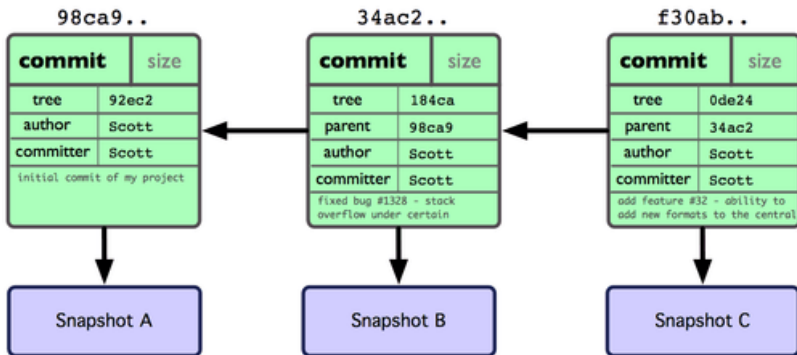
Git — typowe operacje lokalne

- ▷ `git status` `git status -s`
- ▷ `git add <filename>` `git add *.txt`
- ▷ `git commit -m 'comment'`
- ▷ `git commit -a`
- ▷ `git diff` (*zmiany w katalogu roboczym w stosunku do tego co jest przechowywane*)
- ▷ `git diff --cached`
(*przechowywanie vs. ostatni commit*)
- ▷ `git help <action_name>`

- ▶ 3 pliki, robimy **git commit**, do bazy trafia 5 obiektów
(3 x *blob dla plików, drzewo katalogu, commit z metadanmi*)
- ▶ Blob (ang. binary large object) - przechowywanie dużych ilości danych binarnych jako pojedynczy obiekt w bazie



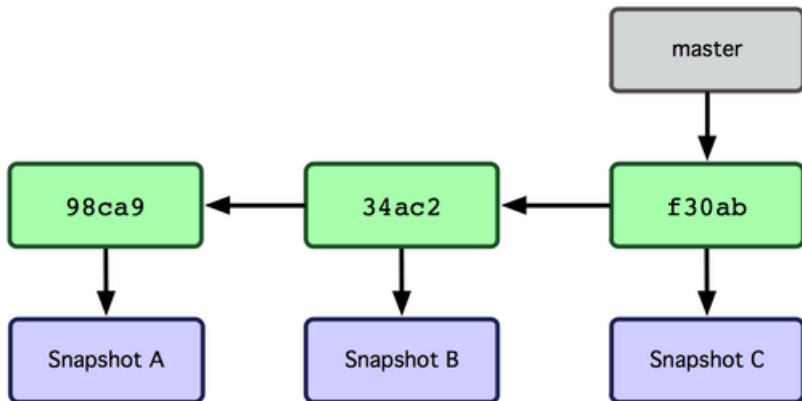
Każdy kolejny obiekt 'commit' zachowuje wskaźnik do swojego poprzednika



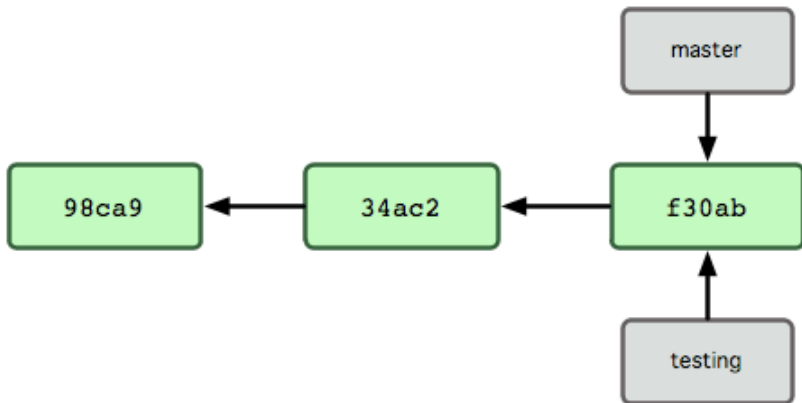
Gałęzie (ang. branches)

- 1 Utworzenie nowej gałęzi to odbicie od głównego pnia (ciągu commitów) i pracę 'na boku'
(w przyszłości będzie ją można dołączyć do pnia)
- 2 Git ma efektywny sposób zarządzania gałęziami oparty na wskaźnikach do commit'ów
(każdy nowy wskaźnik traktowany jako gałąź)
- 3 Wskaźniki te są przechowywane w **refs/heads**
- 4 W **HEAD** jest pamiętany obecnie ustawiony wskaźnik (gałąź)

Gałąź to **przesuwalny wskaźnik** na obiekt 'commit'.
Domyślna nazwa pierwszej gałęzi to **master**, wskazuje
ostatni 'commit' (*automatycznie przesuwa się do przodu*)



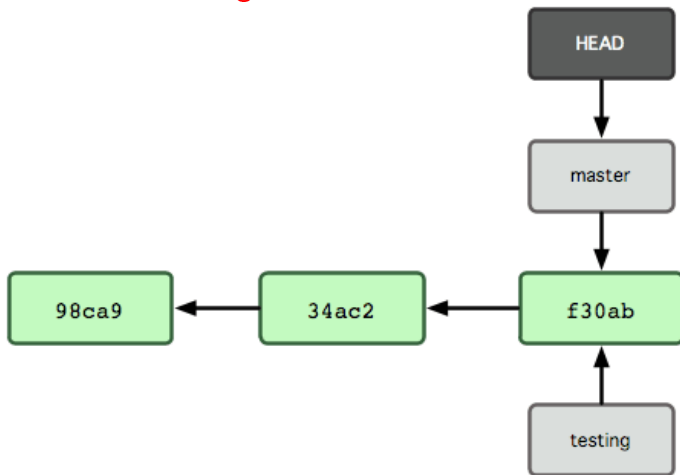
\$ git branch testing tworzy nową gałąź (wskaźnik) do aktualnego 'committa'



Dodatkowo Git ma specjalny wskaźnik o nazwie HEAD, wskazujący gałąź w której jesteś:

\$ git branch --list

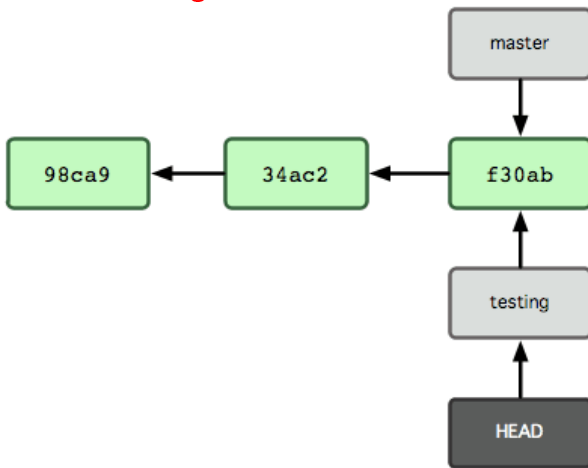
\$ git checkout testing



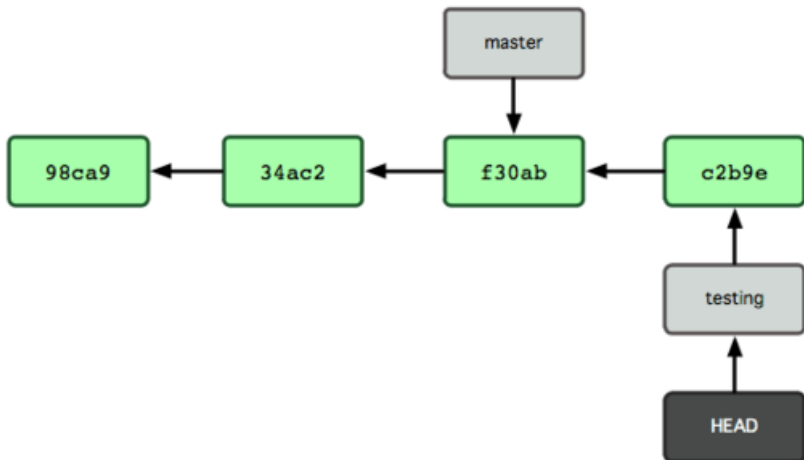
Dodatkowo Git ma specjalny wskaźnik o nazwie HEAD, wskazujący gałąź w której jesteś:

`$ git branch --list`

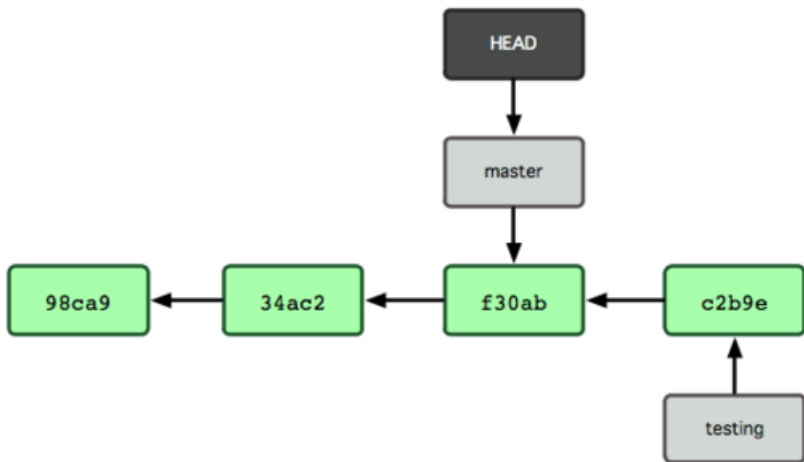
`$ git checkout testing`



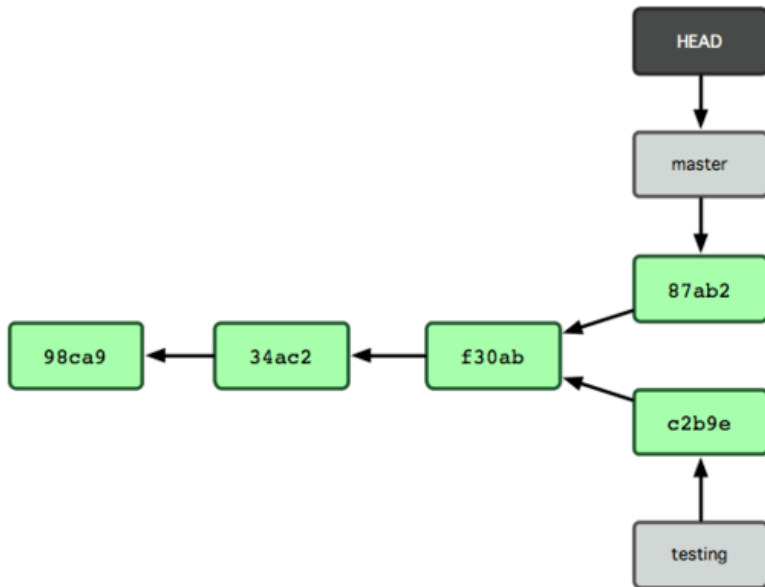
Będąc w **testing** wprowadźmy zmiany i zrobmy commit...



Wróćmy do master i zrobmy zmiany + commit ...



\$ git log --graph --all --oneline



Scalanie — fast forward

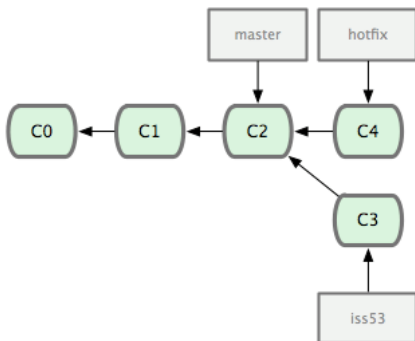
Scalanie jest łatwe jeśli nie ma rozgałęzień:

\$ git checkout -b 'hotfix' *//robimy zmiany + commit*

\$ git checkout master

\$ git merge hotfix

\$ git branch -d hotfix *//usuwanie gałęzi*



Scalanie — fast forward

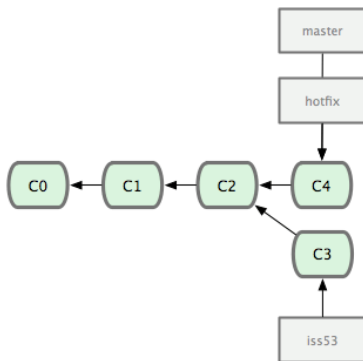
Scalanie jest łatwe jeśli nie ma rozgałęzień:

\$ git checkout -b 'hotfix' *//robimy zmiany + commit*

\$ git checkout master

\$ git merge hotfix

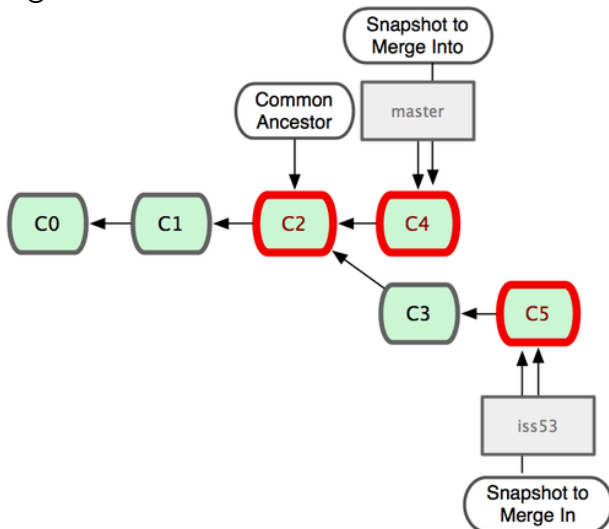
\$ git branch -d hotfix *//usuwanie gałęzi*



Scalanie – "trójstronne" (ang. three-way merge)

\$ git checkout master

\$ git merge iss53

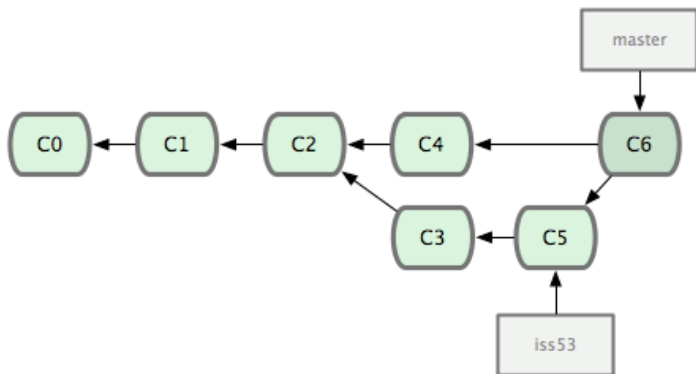


Scalanie – "trójstronne" (ang. three-way merge)

Jeśli są konflikty to musimy je rozwiązać, a następnie

```
$ git add <plik_z_konfliktami>
```

```
$ git commit
```

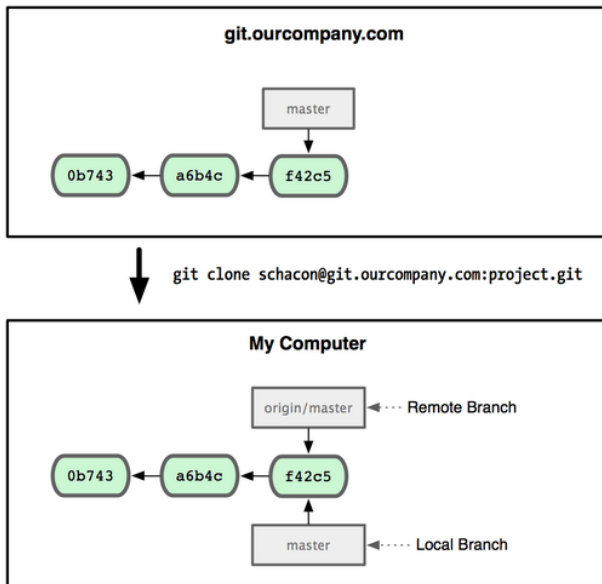


commit c6 posiada więcej niż jednego rodzica...

Gałęzie zdalne


- ▶ **Gałąź zdalna** jest odnośnikiem do stanu gałęzi w zdalnym repozytorium (przy ostatnim połączeniu)
- ▶ Nie można ich lokalnie zmienić, są modyfikowane automatycznie przy wykonywaniu operacji zdalnych
- ▶ Nazywają się:
 <nazwa zdalnego repo>/<nazwa gałęzi>
 (np. <origin>/<master>)

Gałęzie zdalne



Gałęzie zdalne - GitHub

Quick setup — if you've done this kind of thing before

 Set up in Desktop or **HTTPS** SSH `https://github.com/kubal/remote_TP2017.git`

We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

...or create a new repository on the command line

```
echo "# remote_TP2017" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/kubal/remote_TP2017.git
git push -u origin master
```

...or push an existing repository from the command line

```
git remote add origin https://github.com/kubal/remote_TP2017.git
git push -u origin master
```

Obsługa zdalnego repozytorium

- ▷ `$ git clone https://github.com/kubal/wyklad.git`
- ▷ `$ git remote show origin`
- ▷ `$ git remote -v`
- ▷ `$ git remote add <skrót> <url>`
- ▷ `$ git fetch <skrót>` (*pobiera dane, nie merguje*)
- ▷ `$ git merge <skrót>/master` (*merguje pobrane*)
- ▷ `$ git pull` (*pobiera i merguje*)
- ▷ `$ git push <zdalne-repo> <nazwa-gałęzi>`
(*wysyła zmiany do zdalnego repozytorium,
zapewne git spyta o hasło*)

Mergowanie gałęzi zdalnych

- ▶ Klonujemy zdalne repozytorium, modyfikujemy pliki, robimy commit i push i ...
- ▶ ... niestety ktoś wrzucił zmiany do głównego repozytorium przed nami :-)
- ▶ Musimy pobrać zmiany i lokalnie zmerdżować...

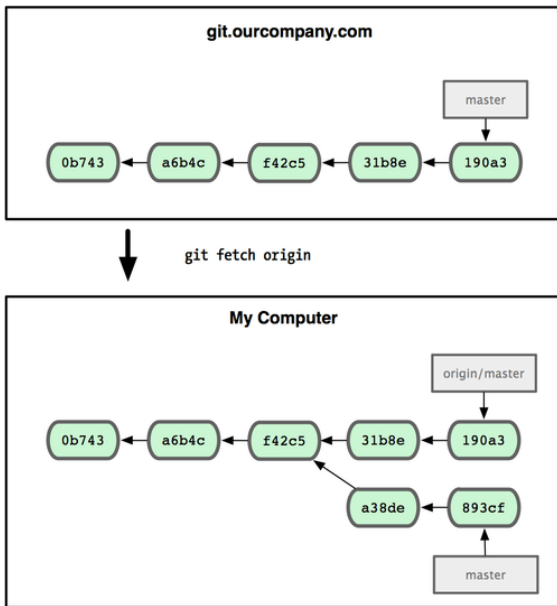
```
$ git fetch origin
```

```
$ git checkout master
```

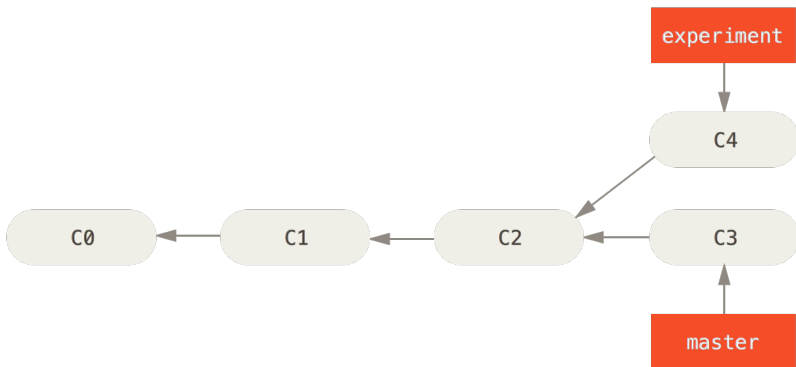
```
$ git merge origin/master
```

```
$ git push origin master
```

Mergowanie gałęzi zdalnych

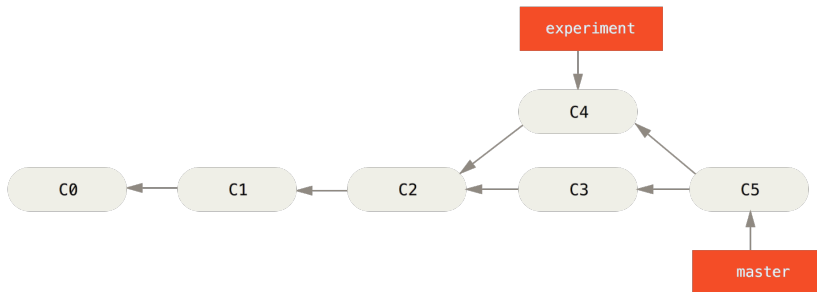


Zmiana bazy (ang. rebasing)



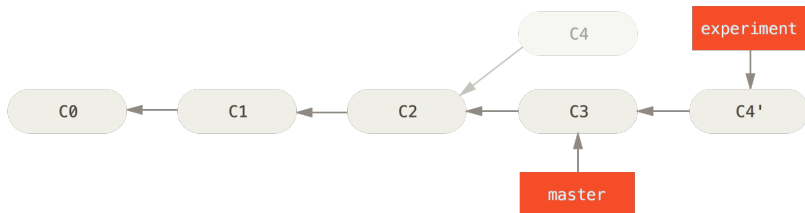
Możemy zrobić **merge...**

Zmiana bazy (ang. rebasing)



Zmiast **merge**, możemy robić (dla urody) **zmianę bazy**

Zmiana bazy (ang. rebasing)



Idziemy do wspólnego przodka gałęzi, zapisujemy tymczasowo co zmieniał każdy z kolejnych commitów aktualnej gałęzi (experiment), dodajemy commity gałęzi do której robimy 'rebase' (master), i na końcu dajemy zapisane commity z aktualnej gałęzi (experiment)

Zmiast **merge**, możemy robić (dla urody) **zmianę bazy**

\$ git checkout experiment

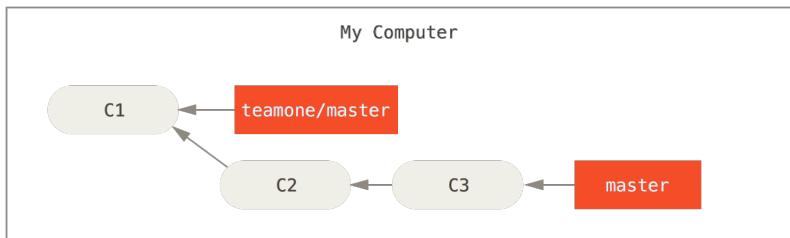
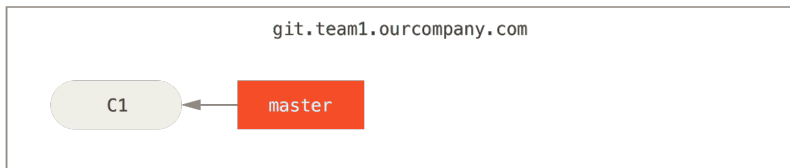
\$ git rebase master

\$ git checkout master

\$ git merge experiment (*fast forward*)

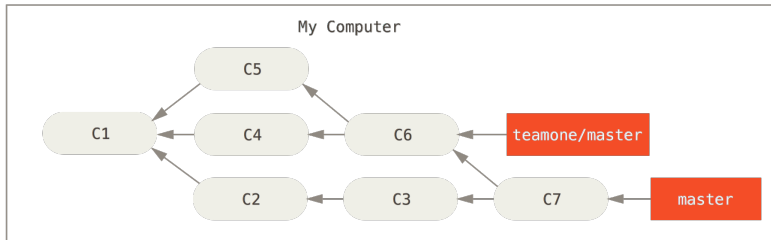
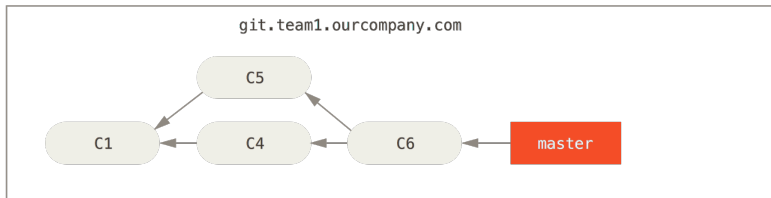
Zmiana bazy – problemy

Ktoś zrobił 'rebase' i wypchnął zmiany do publicznego repozytorium - mamy problem



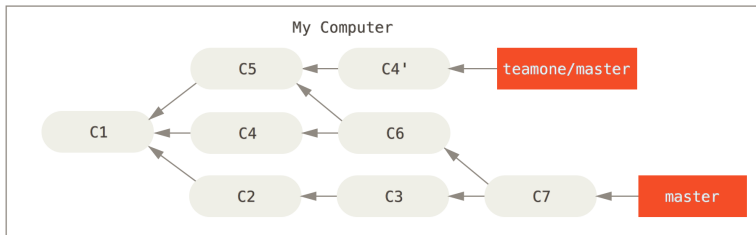
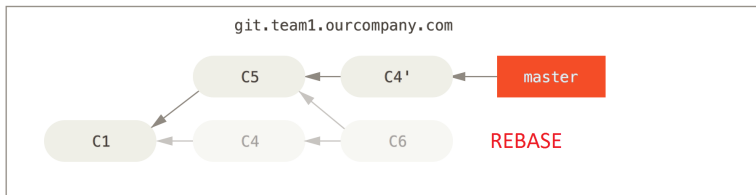
Zmiana bazy – problemy

Ktoś zrobił 'rebase' i wypchnął zmiany do publicznego repozytorium - mamy problem



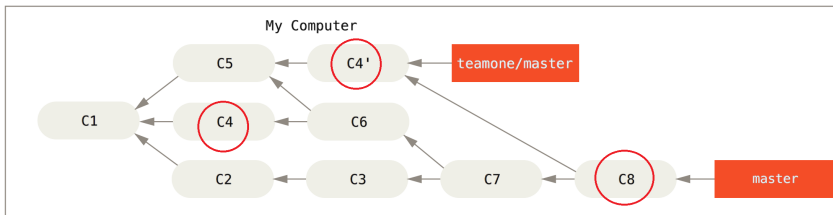
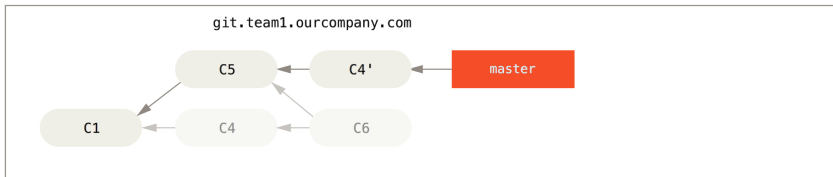
Zmiana bazy – problemy

Ktoś zrobił 'rebase' i wypchnął zmiany do publicznego repozytorium - mamy problem



Zmiana bazy – problemy

Ktoś zrobił 'rebase' i wypchnął zmiany do publicznego repozytorium - mamy problem



Zmiana bazy — problemy

Nie zmieniaj bazy commitów wypchniętych już do publicznego repozytorium!