13 Systemy kontroli wersji – `git`

`Git` to darmowy, otwarty, rozproszony system kontroli wersji, który w szczególności pozwala na niezależne i bezkonfliktowe tworzenie projektu np. programistycznego lub np. projektu mającego na celu przygotowanie notatek z przedmiotu `Środowisko Programowania` przez wiele osób znajdujących się w różnych częściach wszechświata. System kontrolowania wersji taki jak git pozwala na zapisywanie zmian w pliku lub zbiorze plików (repozytorium) tak, aby w razie konieczności można było skorzystać później z odpowiednich wersji tych plików, przywrócić te pliki do ich wcześniejszych wersji, porównać zmiany w różnych okresach czasu, zobaczyć, kto ostatni wprowadził modyfikację, która być może wprowadziła błędy do projektu. Znane serwisy korzystające z gita to m.in `GitHub` i `Bitbucket`.

`Git` jest `rozproszonym` systemem kontroli wersji, co oznacza m.in., że obraz całego repozytorium (również wszystkie poprzednie wersje plików) jest przechowywany lokalnie na maszynie każdego z użytkowników repozytorium. Pozwala to na kontynuacje projektu nawet w przypadku, gdy wersja, która znajduje się na głównym serwerze ulegnie awarii czy zostanie usunięta. Ta cecha odróżnia `Git`a od scentralizowanych systemów kontroli wersji takich, jak: CVS, Subversion. Inną ważna cechą Gita jest to, że większość operacji na danych wykonuje się lokalnie.

Sposób reprezentowania kolejnych wersji w Gicie

Świadomie korzystać z Gita oznacza m.in. rozumieć sposób w jaki pamięta on kolejne wersje przechowywanych danych. W Tabeli **??** zobrazowany został sposób w jaki większość innych systemów wersjonowania (CVS, Subversion, Perforce, Bazaar, itd.) pamięta zmieniające się dane tj. jako kolejne zmiany każdego z plików z osobna.

Git natomiast pamięta zmieniające się dane jako zbiór migawek (ang. snapshots). Za każdym razem kiedy użytkownik komituje lub zapisuje stan projektu Git zapamiętuje referencję do migawki, która pamięta obecny stan każdego z plików. Nie znaczy to oczywiście, że niepotrzebnie przechowuje się te same dane wielokrotnie. Tam, gdzie to możliwe zapamiętuje się link do danych zapisanych w jednej z poprzednich migawek. Sposób reprezentowania danych przez Gita został zobrazowany w Tabeli **??**

Wersja 1	Wersja 2	Wersja 3	Wersja 4	Wersja 5
Plik A	Δ_1	$\rightarrow$	Δ_2	
Plik B	$\rightarrow$	$\rightarrow$	Δ_1	Δ_2
Plik C	Δ_1	$\rightarrow$	Δ_2	Δ_3

Rysunek 1: Sposób przechowywania danych poprzez pamiętanie kolejnych zmian w każdym pliku z osobna

Wersja 1	Wersja 2	Wersja 3	Wersja 4	Wersja 5
|	|	|	|	|
Plik A	A_1	A_1	A_2	A_2
|	|	|	|	|
Plik B	B	B	B_1	B_2
|	|	|	|	|
Plik C	C_1	C_1	C_2	C_3

Rysunek 2: Sposób przechowywania danych (przez Gita) jako zbiór kolejnych migawek.

Stany w jakich mogą znajdować się pliki

Pliki w Gicie mogą znajdować się w jednym z trzech stanów mogą być: zakomitowane (ang. comitted), zmodyfikowane (ang. modified) oraz mogą być staged. Jeśli są zakomitowane tzn., że są już bezpiecznie zapisane w lokalnej bazie danych użytkownika. Jeśli są zmodyfikowane tzn. że są zmienione, ale nie zakomitowane. Plik jest staged jeśli jest oznaczony jako ten, który zostanie przesłany w kolejnym komicie.

Każdemu z tych trzech stanów odpowiada jedna z trzech sekcji, w których rezyduje Git projekt: katalog Gita (ang. the Git directory), katalog roboczy (the working directory), oraz staging area. W katalogu Gita, który domyślnie rezyduje w naszym systemie plików jako `.git`, umieszczone są metadane oraz obiektowa baza danych przechowująca projekt użytkownika. Tę właśnie, najważniejszą część przechowywanych danych kopiuje się w przypadku klonowania danych z jednego komputera na drugi. Katalog roboczy zawiera jedną z migawek, na której aktualnie pracuje użytkownik. Staging area to plik w katalogu .git, w którym przechowuje się informację, o tym, które pliki zostaną następnym razem zakomitowanie.

Sposób krążenia danych w Gicie obrazują kolejne trzy punkty

1. Modyfikacja pliku w katalogu roboczym.
2. Przesłanie pliku do staging area.
3. Commit, zapamiętanie danych ze staging area jako trwały obraz w katalogu .git.
4. Synchronizacja z serwerem zewnętrznym.

Konfiguracja Gita

Po zainstalowanie Gita, ale jeszcze przed pierwszym użyciem warto poświęcić chwilę czasu na konfigurację. Przede wszystkim należy zaznajomić Gita ze swoim imieniem i nazwiskiem oraz adresem poczty elektronicznej. Te dane będą częścią każdego naszego komitu. Wykorzystuje się do tego narzędzie `git config` pisząc np.

```
$ git config --global user.name "Michal Wrona"
$ git config --global user.email wrona@tcs.uj.edu.pl
```

Opcja `--global` pozwala na to, aby wykonać "identyfikację" tylko raz. Jeśli dla niektórych projektów chcielibyśmy użyć innych danych, wtedy nie używamy opcji `--global`.

Czasami Git potrzebuje skorzytać z edytora tekstu (każdy komit należy opatrzyć odpowiednim opisem). Jeśli chcemy skorzystać z innego edytora niż domyślny, możemy to ustawić:

```
git config --global core.editor emacs
```

Listę tak skonfigurowanych ustawień można podejrzeć pisząc:

```
git config --list
```

lub np. `git config user.name`, gdy interesuje nas jedynie nazwa użytkownika.

Ustawienia Gita przechowywane są w jednym z trzech miejsc. W pliku `/etc/gitconfig`, w pliku `~/.gitconfig` lub w pliku `~/.config/git/config` w katalogu domowym, lub w pliku konfiguracyjnym danego repozytorium `.git/config`.

Poza tym warto wiedzieć, że można uzyskać pomoc w zakresie używania gita pisząc: `man git`, albo `git help`. Git też czasami sam podpowiada co można zrobić.

Praca z Gitem

Będziemy pracować z naszym ulubionym projektem programistycznym zawierającym pliki: `compile.sh`, `main.c`, `power.c`, `power.h`, `sqr.c`.

Polecenie **`git init`** tworzy pusty katalog `.git` i pozwala na rozpoczęcie pracy z Gitem. `git init` Alternatywnie, można rozpocząć pracę z projektem klonując już istniejące repozytorium umieszczone na jednym z serwerów przechowujących repozytoria:

`$ git clone` `https://github.com/mchwrona/projekt/`

tworzymy katalog o nazwie `projekt`, inicjalizujemy katalog .git w środku katalogu projekt oraz ściągamy wszystkie dane związane z repozytorium. W katalogu pojawia się kopia ostatniej wersji (ostatniej migawki).

Nazwę katalogu roboczego możemy również podać jako część polecenia:

```
$ git clone https://github.com/mchwrona/projekt/ Project
```

W dalszym ciągu notatek będziemy pracować z zainicjalizowanym lokalnie, za pomocą `git init`, projektem. W tej chwili jeszcze żaden z plików projektu nie jest "śledzony" (ang. tracked) przez Gita.

Dodajemy pliki do śledzenia za pomocą **git add**:

```
$ git add *.c
$ git add *.h
$ git add *.sh
$ git add LICENSE
$ git add CONTRIBUTING.md
```

Po czym komitujemy dodane pliki do repozytorium (**git commit**):

```
$ git commit -m 'initial project version'
```

W pliku `LICENSE` umieszczamy treść licencji, np. `GNU General Public License v2.0`. `Github` pozwala nam na automatyczne stworzenie pliku `LICENSE`. W tworzonym przez siebie repozytorium warto jest mieć również plik `CONTRIBUTING.md`, w którym znajdują się domyślnie wytyczne tworzenia projektu/repozytorium.

Każdy komit wymaga komentarza. Można umieścić go po opcji `-m`. W przypadku ominięcia tej opcji włączy się automatycznie nasz domyślny edytor, gdzie będziemy musieli skomentować poczynione przez nas zmiany. Polecenie **git status** pozwala nam na monitorowanie bieżącego stanu. Przykładowo, wywołanie tej komendy zaraz po `git add *.c`:

```
On branch master

Initial commit

 Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   main.c
    new file:   power.c
    new file:   sqr.c

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

    compile.sh
    power.h
    LICENSE
    CONTRIBUTING.md
```

Przed wywołaniem ostatniej komendy wszystkie pliki znajdą się w "pierwszej sekcji" podobnego komunikatu. Po wykonaniu komita, `git status` zwróci:

```
On branch master
nothing to commit, working directory clean
```

Gałąź `master` to tylko jedna z możliwych gałęzi, kierunków w jakim możemy rozwijać nasz projekt. Technologię `Git branching` omówimy w dalszej części notatek.

Można również zauważyć, że w pierwszej części komunikatu po wykonaniu komita, występuje nazwa gałęzi `master`, suma kontrolna `SHA-1`: `ad5f8ec`, oraz wpisana przez nas wiadomość. Następnie umieszczane są dane komitującego (nazwa użytkownika, adres poczty).

```
$ git commit -m 'initial project version'
[master (root-commit) ad5f8ec] initial project version
 Committer: Michal Wrona <wrona@student.tcs.uj.edu.pl>
....
```

W przypadku, gdy w czasie pracy nad projektem zmienimy plik CONTRIBUTING.md, wykonamy na nim `git add` i ponownie zmienimy (bez wykonywania po drodze żadnego komita), na wywołanie `git status` otrzymamy następującą "odpowiedź":

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

    modified:   CONTRIBUTING.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   CONTRIBUTING.md
```

Istotnie, pojawia się w powyższym komunikacie nazwa pliku `CONTRIBUTING.md` dwukrotnie. Jest tak dlatego, że dwie różne wersje pliku `CONTRIBUTING.md` znajdują się w dwóch różnych miejscach przestrzeni danych Gita. Plik `CONTRIBUTING.md` po pierwszej zmianie jest `staged` i czeka na komit. Plik po drugiej zmianie nie jest nawet `staged`, a tylko `modfied`. W tej chwili wykonanie `git add` spowoduje umieszczenie w staging area pliku `CONTRIBUTING.md` po obydwóch modyfikacjach, natomiast wykonanie `git commit` wykonanie komita zawierającego ten plik tylko po pierwszej modyfikacji.

W tej chwili warto wspomnieć o tym, że wywołanie `git status -s` zamiast `git status` wiążę się z dużo oszczędniejszym komunikatem. Gdy osiągniemy większą łatwość w posługiwaniu się gitem skromnijszy, choć bardziej zakodowany komunikat pozwoli nam na szybszą pracę z systemem.

W trakcie naszej pracy nad projektem tworzone są pewne mniej ważne pliki takie, jak pliki `.log`. W zasadzie nie interesuje nas, czy takie pliki w trakcie naszej pracy zostaną

zmodyfikowane, czy nie. Nie chcemy więc, aby informacja o ich modyfikacji pojawiała się po wywołaniu `git status`. Nazwy (np. ich roszerzenia) plików ignorowanych można zdefiniować w pliku `.gitignore`. Przy pomocy `githuba`, można stworzyć taki plik automatycznie, wybierając tylko rodzaj naszego projektu, jako np. projekt w `C`.

Rozważmy teraz kolejne zmiany na pliku `CONTRIBUTING.md`. Wywołanie `git status` może dać nam wiedzę o tym, które pliki zostały zmienione i nie zostały przeniesione do `staged area` po ostatniej modyfikacji. Jeśli natomiast chcemy dowiedzieć się dokładnie jakie zmiany zostały wykonane w zmienionym pliku `CONTRIBUTING.md` możemy użyć polecenia **`git diff`**. Wywołanie tej komendy w takiej sytuacji może doprowadzić do następującego komunikatu.

```
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index e161e48..0b6d938 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -2,7 +2,14 @@ Examples of behavior that contributes to creating a positive
environment include

 - Using welcoming and inclusive language
 - Being respectful of differing viewpoints and experiences
-- Gracefully accepting constructive criticism
 - Focusing on what is best for the community
 - Showing empathy towards other community members

+Examples of unacceptable behavior by participants include:
+
+- The use of sexualized language or imagery and unwelcome sexual attention or
advances
+- Trolling, insulting/derogatory comments, and personal or political attacks
+- Public or private harassment
+- Publishing others' private information, such as a physical or electronic address,
without explicit permission
+- Other conduct which could reasonably be considered inappropriate in a professional
setting
```

Teraz już wiemy dokładnie jakie zmiany zostały wprowadzony po ostatnim `git add`. Wywołanie `git diff --staged` albo **`git diff --cached`** pozwala dowiedzieć się jakie zmiany zostały już przesłane do `staged area`.

W tej chwili, gdy wszystkie zmiany są już staged, możemy wykonać komita. Opcjonalnie, można, bez wywoływania `git add`, dodać do komita wszystkie zmienione pliki. W tym celu używamy opcji `-a` i piszemy:

```
$ git commit -a -m 'new message'
```

Do usuwanie plików zarówno z twardego dysku jak i z projektu służy polecenie **`git rm`**. Nie wystarczy wykonać zwykłego usuwania za pomocą `rm`. Wtedy Git nie przestanie śledzić pliku. Jeśli plik został zmodyfikowany, ale jego nowa wersja nie została jeszcze przesłana do staging arena należy napisać `git rm -f nazwa_pliku`. W przypadku gdy chcemy wyrzucić plik z projektu, ale chcemy zostawić go na dysku piszemy: `git rm --cached`.

Do zmiany nazwy pliku używamy polecenia: **`git mv`**.

Polecenie `git mv nazwa_pliku_1 nazwa_pliku2` jest równoważne

```
$ mv nazwa_pliku_1 nazwa_pliku_2
$ git rm nazwa_pliku_1
$ git add nazwa_pliku_2
```

Jeśli chcemy wyświetlić listę ostatnich komitów w odwróconym porządku chronologicznym piszemy **`git log`**. Przydatne opcje tego polecenia umieszczamy w następującej liście.

`git log -p`	wyświetla różnice pomiędzy kolejnymi komitami, z `git diff` po każdym komici
`git log -k`	wyświetla tylko k ostanich komitów
`git log --stat`	pod każdym komitem drukuje listę zmodyfikowanych plików, liczbę plików zmodyfikowanych, jak wiele linii zostało dodanych, i jak wiele ujętych; podobna informacja zbiorcza znajduje się na końcu raportu
`--shortstat`	wyświetla tylko zmienione, wstawiona i usunięte linie ze `--stat`
`--name-only`	pokazuje listę plików zmodyfikowanych po ostatnim komicie
`--name-status`	pokazuje listę plików dodanych, usuniętych i zmodyfikowanych
`--abbrev-commit`	pokazuje tylko część sumy kontrolnej SHA-1, a nie całe 40 znaków
`--relative-date`	wypisuje datę modyfikację względem czasu wywołania, np. “2 weeks ago”
`--graph`	wyświetla graf w kodzie ASCII dla gałęzi projektów
`--pretty`	przedstawia listę komitów w alternatywnej formie
`--since, --after`	np. `git log --since=2.weeks`
`--author`	wypisuje tylko komity odpowiedniego autora
`--committer`	wypisuje tylko komity odpowiedniego komitera
`--grep`	wypisuje komity z komentarzami zawierającymi konkretne stringi

Teraz zajmiemy się poprawianiem błędnie wykonanych akcji, które mogą zdarzyć nam się na różnych poziomach operowania danymi w Gicie.

Do poprawienie ostatniego komita możemy użyć polecenia **`git commit --amend`**. Możemy użyć tego polecenia w przypadku kiedy zapomnieliśmy umieścić w naszym ostatnim komicie jednego z plików lub też dodaliśmy o jakiś plik za dużo. Poniżej rozpatrujemy pierwszą możliwość.

```
$ git commit -m 'initial commit'
$ git add forgotten_file
$ git commit --amend
```

W przypadku, gdy mamy dwie wersje pliku (np. `README`), jedną w `staging area`, a drugą zmodyfikowaną, możemy chcieć zrobić jedną z dwóch rzeczy. Cofnąć `git add`, albo modyfikację, na której `git add` jeszcze nie został wykonany. Podpowiedź jak to zrobić, możemy przeczytać z `git status`:

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

    new file:   README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   README
```

Tak więc, aby cofnąć `git add` na pliku `README` piszemy `git reset README`. W przypadku natomiast gdy chcemy wycofać późniejsze modyfikacje piszemy `git checkout -- README`.

Praca zdalna z Gitem

Tak, jak już wcześniej o tym pisaliśmy Gita używa się najczęściej korzystając jednocześnie z jednego z popularnych serwerów np. `Githuba`. W tym celu zakłada się konto na portalu `github.com`. Zaraz po założeniu konta możemy rozpocząć nowy projekt, a później sklonować go na naszą lokalną maszynę w następujący sposób.

```
$ git clone https://github.com/mchwrona/projekt
Cloning into 'projekt'...
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
Checking connectivity... done.
```

Polecenie **`git remote`** podaje listę zdalnych miejsc, w których znajduje się nasze repozytorium, a `git remote -v` z rozróżnieniem zdalnych miejsc z których i na które można czytać. W naszym przypadku wszystkie operacje będziemy wykonywać, na domyślnym, `origin`.

```
$ git remote -v
origin    https://github.com/mchwrona/projekt (fetch)
origin    https://github.com/mchwrona/projekt (push)
```

Pliki z serwera można pobierać za pomocą poleceń `git fetch` oraz **`git pull`**. Tak więc w naszym przypadku piszemy `git fetch origin`, albo `git pull origin`. Pierwsza komenda ściąga nowe dane, ale ich nie merguje z tym co mamy na dysku. Możemy to zrobić w odpowiedniej dla nas chwili, pisząć: `git pull`. Druga od razu merguje.

Aby wysłać dane na serwer piszemy, np. **`git push`** `origin master`. W tym przypadku podaliśmy gałąź, tutaj: `master` (nie podając zadziałają domyślne powiązania).

Tagowanie

Niektóre ważne punkty w rozwoju naszego projektu możemy tagować, np. nadawać nr wersji. Aby wylistować dostępne tagi wystarczy posłużyć się poleceniem: **`git tag`**. Można zawęzić listę tagów używając opcji `-l` po której znajduje się glob, np. `git tag -l "ver3.*"`.

Git używa tagów dwojakiego rodzaju: lekki tag (ang. lightweight) oraz tag z adnotacją (ang. annotated). Tag lekki to po prostu wskaźnik na którąś z migawek. Z drugiej strony tag z adnotacją to pełnoprawny (np. o własnej sumie kontrolnej) obiekt (migawka) w bazie danych Gita. Tagi lekkie obliczone są raczej jako tagi, które po pewnym czasie zostaną usunięte. Lekki tag można utworzyć pisząc po prostu: `git tag temp-lw`, do tagu z adnotacją używamy opcji `-a`. Aby stworzyć tagu tego drugiego rodzaju piszamy np.:

```
git tag -a v1.0 -m 'version 1.0'
```

Polecenie `git show nazwa_tagu` wyświetla informacje o tagu. Łatwo zauważyć, że inne informacje zostają wypisane w przypadku tagów lekkich i inne w przypadku tagów z adnotacją.

Otagować można również dużo wcześniejsze komity. Możemy zrobić tak, jak w przykładzie poniżej.

```
$ git log --pretty=oneline
62175bc8265d5ce241506e8c1f630fe28bf574b4 README
db1c87b2203fb486f81fdc45de8ade5638fc92b9 new message
55659d3c33f7becd0e6bc7a73be5f441144230f5 new CONTRIBUTING.md
a96b22a238f6423e64df9ca406381a1ab5bec9cd CONTRIBUTING
28cb30cc3262d31d372b5e407b63901b1a6c8820 Dropbna modyfikacja
ad5f8ecd53ec684f3b54b69e82a01e359024169f initial project version
```

Teraz używamy sumy kontrolnej komita, aby utworzyć wersję 0.5 naszego projektu:

```
git tag v0.5 28cb30
```

Tagi należy przesyłać na serwer wprost. Można napisać: `git push origin nazwa_tagu` albo **`git push origin --tags`** aby przesłać wszystkie tagi na serwer.

Zmiana migawki

Możemy dostać się do dowolnego miejsca w historii używając pierwszych cyfr sumy kontrolnej pisząc np. **`git checkout`** `28cb30`.

Aby dostać się do miejsca w historii wskazywanego przez tag, wystarczy napisać
`git checkout nazwa_tagu`.

Następnie możemy komitować poczynając od wybranego miejsca w historii. Aby jednak moc później powrócić do tworzonej odnogi należy wcześniej stworzyć gałąź:
`git checkout -b [branchname] [tagname]`
lub
`git checkout -b [branchname] [suma_kontrolna]`.

Branching

Pracując nad projektem z różnych powodów możemy chcieć np. od pewnego momentu rozwijać projekt równolegle. Pisząc coś w kilka osób dobrze jest mieć co najmniej jedną aktywną gałąź projektu dla każdej osoby. Możemy również chcieć wprowadzić zmiany o dużym prawdopodobieństwie błędu. W takim calu możemy stworzyć gałąź testing.

Gałąź w Gicie to po prostu wskaźnik na pewną migawkę w naszej bazie danych. Pracę nad repozytorium zaczynamy zwyczajowo na gałęzi master. W dogodnym dla nas momencie możemy utworzyć nową gałąź `testing` pisząc:

`git branch` `testing`
lub
`git checkout -b` `testing`

Dostępne gałęzie można wypisać za pomocą: `git branch`. Do wskazywania na aktualną gałąź służy w Gicie wskaźnik HEAD. Gałąź wskazywaną przez HEAD zmieniamy pisząc:

```
$ git checkout testing
```

Jeśli po jakimś czasie zaaprobujemy "zmiany testowe" gałęzi branching możemy chcieć ją włączyć do gałęzi master. Wtedy piszemy.

```
$ git checkout master
$ git merge testing
```

W przypadku jeśli równolegle do zmian na gałęzi testing, przeprowadzaliśmy zmiany na gałęzi master powyższa komenda może prowadzić do konfliktów.

```
Auto-merging README
CONFLICT (add/add): Merge conflict in README
Automatic merge failed; fix conflicts and then commit the result.

$ git status

On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both added:      README

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

Wykonanie `git status` ujawniło nam powody konfliktu. Przed zmerdżowaniem gałęzi należy najpierw rozwiązać konflikt. W tym celu możemy posłużyć się narzędziem: `git mergetool`.

Komendą podobną do `git merge` o której trzeba wiedzieć to **`git rebase`**.

```
$ git checkout testing
$ git rebase master
```

Powyższe komendy przepiszą gałąź testing na gałąź master po ostatnim komicie na master. W przeciwieństwie do `git merge` nowa historia gałęzi master nie będzie zawierała komita o zmerdżowaniu gałęzi testing oraz merge. Poza tym po wykonaniu `git rebase` nowa historia jest liniowa, nie jst zlepkiem kilku gałęzi jak w przypadku merge. Przypominamy, że można to zaobserwować pisząc `git log --graph`. Pierwszy przykład poniżej pokazuje sytuację po `git merge`, a drugi po `git rebase`

```
$ git log --oneline --graph
*   54dd789 Merge branch 'test'
|\
| * dcb2e9b helloB2
| * d5ca4e0 helloB1
* | 751bb10 helloA2
* | dc60953 helloA1
|/
* 6872d3f hello
```

```
$ git log --oneline --graph
* c67a479 helloB2
* 7200d32 helloB1
* 751bb10 helloA2
* dc60953 helloA1
* 6872d3f hello
```