AGH AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

Programowanie obiektowe - raport

# Dziady są Git

Git exercises

31 października 2024

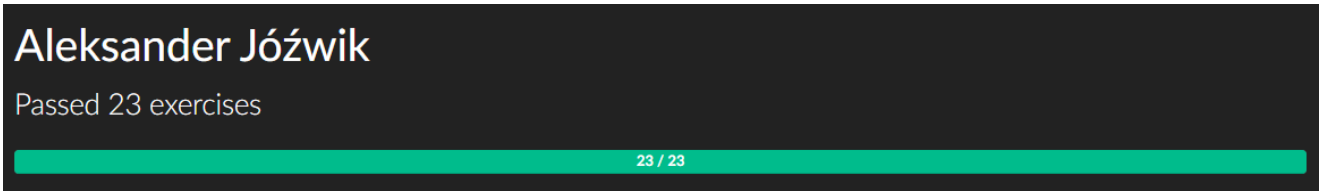
Aleksander Jóźwik

# 1. Wprowadzenie

Poniższy raport zawiera przebieg wykonywanych ćwiczeń dotyczących systemu kontroli wersji Git ze strony https://gitexercises.fracz.com/. Do każdego z zadań zamieszczono odpowiedź wraz z objaśnieniami poszczególnych komend.

Dowód wykonania 23 zadań można znaleźć na tej stronie:
https://gitexercises.fracz.com/committer/pzvc

oraz na poniższym zrzucie ekranu:




# 2. Rozwiązania

## 2.1. Zadanie 1 (master)

```
git verify
```

Objaśnienie: *git verify* działa w tym przypadku (na potrzeby tego zadania) jak *git push*, czyli służy do wysyłania zmian z lokalnego repozytorium do zdalnego repozytorium („wypycha" zacommitowane zmiany).

## 2.2. Zadanie 2 (commit-one-file)

```
git add A.txt
git commit -m "Add A.txt"
```

Objaśnienie: *git add* dodaje pliki z przestrzeni roboczej do staging area. Tylko pliki ze staging area zostaną zawarte w commicie po użyciu *git commit*. *Git commit* natomiast służy do zatwierdzania zmian w lokalnym repozytorium. Użycie flagi *-m* pozwala na dodanie wiadomości do commitu.

## 2.3. Zadanie 3 (commit-one-file-staged)

```
git reset A.txt
git commit -m "Add B.txt"
```

Objaśnienie: *git reset <plik>* usuwa wskazany plik ze staging area. Niewskazanie pliku spowoduje usunięcie wszystkich plików ze staging area. Domyślnie działa on w trybie `--mixed` (cofa wskaźnik HEAD do wybranego commita i usuwa zmiany z staging area, ale zachowuje je w katalogu roboczym).

## 2.4. Zadanie 4 (ignore-them)

```
vi .gitignore

> *.o
> *.exe
> *.jar
> libraries/

git add .gitignore
git commit -m "Add gitignore"
```

Objaśnienie: Plik *.gitignore* określa celowo nieśledzone pliki, które Git powinien ignorować. Aby zignorować wszystkie pliki z określonym ciągiem znaków w nazwie pliku, wystarczy go wpisać. Aby zignorować wszystkie pliki z określonym rozszerzeniem, należy użyć symbolu „gwiazdki", np. **.exe*. Aby zignorować całe katalogi, należy umieścić ukośnik na końcu reguły, np. *libraries/* Aby określić pełną ścieżkę z lokalizacji *.gitignore*, należy rozpocząć regułę od ukośnika, np. */libraries*

Należy zauważyć, że istnieje różnica między regułą *libraries/* i */libraries/*. Pierwsza z nich zignoruje wszystkie katalogi o nazwie *libraries* w całym projekcie, podczas gdy druga zignoruje tylko katalog *libraries* w tej samej lokalizacji co plik *.gitignore*.

Polecenia *vi* użyto do wywołania edytora *VIM* i zapisania w ten sposób treści pliku *.gitignore*.

## 2.5. Zadanie 5 (chase-branch)

```
git merge escaped
```

Objaśnienie: *git merge* służy do łączenia gałęzi w repozytorium. Ponieważ gałąź chase-branch była bezpośrednim przodkiem gałęzi escaped, wskaźnik można po prostu przenieść i nie jest konieczne zatwierdzenie scalenia (również konflikty są niemożliwe w takich sytuacjach). Jest to scalanie Fast-Forward (występuje, gdy nie było nowych commitów w gałęzi docelowej).

## 2.6. Zadanie 6 (merge-conflict)

```
git merge another-piece-of-work

vi equation.txt

> 2+3=5

git add equation.txt
git commit --no-edit
```

Objaśnienie: Ponieważ dwie gałęzie wprowadziły zmiany w tym samym pliku i w pobliżu tej samej linii, wystąpił konflikt scalania, który należało rozwiązać manualnie. W pliku *equation.txt* poprawiono równanie, a następnie dodano go do staging area. *Git commit –no-edit* to komenda, która tworzy nowy commit bez otwierania edytora tekstu do edycji wiadomości commita.

Podczas merge'a akceptuje domyślną wiadomość merge'a (gdy nie chcemy modyfikować automatycznie wygenerowanej wiadomości).

## 2.7. Zadanie 7 (save-your-work)

```
git stash

vi bug.txt
# usunięcie linii z bugiem

git add .
git commit -m "bugfix"

git stash pop

vi bug.txt
# dodanie nowej linii
> Finally, finished it!

git add .
git commit -m "Finish work"
```

Objaśnienie: *git stash* służy do tymczasowego przechowywania zmian, które nie są gotowe do commita.

*git stash* - zapisuje zmiany i czyści working directory

*git stash pop* - przywraca ostatnie zmiany i usuwa je ze stasha

*git add .* - dodaje wszystkie pliki z katalogu bieżącego do staging area

## 2.8. Zadanie 8 (change-branch-history)

```
git rebase hot-bugfix
```

Objaśnienie: *git rebase* służy do zmiany bazy gałęzi, czyli przenoszenia lub łączenia serii commitów. W tym zadaniu służy do przeniesienia bugfix (commit C) przed nasze zmiany (commit B).

## 2.9. Zadanie 9 (remove-ignored)

```
git rm ignored.txt
git commit -am "Remove ignored file"
```

Objaśnienie: *git rm <plik>* służy do usuwania plików z systemu plików oraz z repozytorium Git. Jeżeli chcemy tylko aby Git przestał śledzić dany plik (ale nie usuwał go z dysku) to należy wywołać komendę `git rm --cached <plik>`.

Metoda *-am* w *git commit* dodaje wszystkie zmiany i commit w jednej komendzie.

## 2.10. Zadanie 10 (case-sensitive-filename)

```
git mv File.txt file.txt
git commit -am "fix lettercase"
```

Objaśnienie: *git mv* służy do przenoszenia plików w repozytorium Git. W trakcie przenoszenia pliku można zmienić jego nazwę, co jest wykorzystywane w tym ćwiczeniu.

## 2.11. Zadanie 11 (fix-typo)

```
vi file.txt
# naprawa literówki w wyrazie world

git commit -a --amend
# naprawa literówki w wiadomości commitu
```

Objaśnienie: *git commit --amend* pozwala na zmianę ostatniego commitu (wskazywanego przez HEAD). Flaga *-a* pozwala na dodanie wszystkich plików do commitu bez użycia *git add*.

## 2.12. Zadanie 12 (forge-date)

```
git commit --amend --no-edit --date="1987-01-01"
```

Objaśnienie: Stosujemy poprzednią komendę z dodatkowymi flagami `--no-edit`, która pozwala na nieedytowanie oryginalnej wiadomości commitu oraz `--date`, która pozwala na zmianę daty commitu.

## 2.13. Zadanie 13 (fix-old-typo)

```
git rebase -i HEAD~2
# zmiana "pick" na "edit" przy commicie "Add Hello wordl"

vi file.txt
# zmiana "wordl" na "world"

git add file.txt
git rebase --continue
# naprawa literówki w wyrazie "world" w wiadomości commita

# naprawa konfliktów
vi file.txt
# pozostawienie "Hello world"
                "Hello world is an excellent program"

git add file.txt
git rebase --continue
# pozostawienie wiadomości commitu bez zmian
```

Objaśnienie: *git rebase -i HEAD~2* otwiera interaktywny rebase dla ostatnich 2 commitów. Wykorzystane w nim komendy to: pick (użyj commita) oraz edit (zatrzymaj się na tym commicie - w celu dokonania zmian). Nastąpiła także konieczność rozwiązania konfliktu.

## 2.14. Zadanie 14 (commit-lost)

```
git reflog
git reset --hard HEAD@{1}
```

Objaśnienie: *git reflog* pokazuje historię zmian referencji (wskaźników) w repozytorium. Jest to bardzo przydatne narzędzie do odzyskiwania „utraconych" zmian. Dzięki niemu można uzyskać m.in. referencję i numer operacji.

Użyty tutaj `git reset --hard <commit>` cofa wskaźnik HEAD, staging area oraz katalog roboczy do stanu wybranego commita, usuwając lokalne zmiany bez możliwości ich przywrócenia.

Poza opisanym wcześniej trybem `--mixed` istnieje także tryb `--soft`, który przesuwa wskaźnik HEAD do określonego commita bez modyfikowania staging area i katalogu roboczego. Zmiany, które były wprowadzane po wybranym commitcie, zostają w staging area. Można je od razu zatwierdzić (commit), ponieważ są przygotowane.

## 2.15. Zadanie 15 (split-commit)

```
git reset HEAD~1
git add first.txt
git commit -m "add first file"
git add second.txt
git commit -m "add second file"
```

Objaśnienie: *git reset* został wykorzystany w trybie *mixed* (domyśnym) by cofnąć HEAD do poprzedniego commita oraz usunąć pliki ze staging area, ale pozostawiając je na dysku. Pozwoliło to na ich późniejsze dodanie w osobnych commitach.

## 2.16. Zadanie 16 (too-many-commits)

```
git rebase -i HEAD~2
# zmiana "pick" na "squash" przy drugim commicie
# usunięcie zbędnej wiadomości drugiego commitu
```

Objaśnienie: Opcja squash, używana w trybie interaktywnym *git rebase -i*, pozwala na połączenie (squashowanie) kilku commitów w jeden. Pozostawiamy *pick* przy commicie, do którego chcemy dołączyć inne commity - przy nich ustawiamy opcję *squash*.

## 2.17. Zadanie 17 (executable)

```
git update-index --chmod=+x script.sh
git commit -m "change permissions"
```

Objaśnienie: *git update-index* pozwala na bezpośrednie manipulowanie staging area, czyli indeksem. Zapewnia zaawansowaną kontrolę nad stanem plików w indeksie. W tym ćwiczeniu pozwala na zmianę flagi wykonywalności poprzez *–chmod=+x <plik>*. Poza tym umożliwia ignorowanie lokalnych zmian, zablokowanie plików przed śledzeniem zmian oraz ręczne zarządzanie indeksem.

## 2.18. Zadanie 18 (commit-parts)

```
git add -p file.txt
# należy wybrać opcję split "s"
# nowe linie, które zawierają "task 1" należy zatwierdzić "y",
# a te, które nie zawierają odrzucić "n"

git commit -m "first"
git commit -am "rest"
```

Objaśnienie: *git add -p* pozwala na wybiórcze dodawanie fragmentów (hunków) zmian z plików do staging area. Jest szczególnie przydatne, do podziału pracy na kilka commitów, mimo że zmiany dotyczą tego samego pliku.

## 2.19. Zadanie 19 (pick-your-features)

```
git cherry-pick feature-a
git cherry-pick feature-b
git cherry-pick feature-c
# rozwiązanie konfliktu scalania poprzez połączenie wszystkich linii kodu

git add .
git cherry-pick --continue
```

Objaśnienie: *git cherry-pick* pozwala na skopiowanie (przeniesienie) konkretnego commita lub commitów z jednej gałęzi do innej, bez konieczności łączenia całych gałęzi. Jest to przydatne do przeniesienia zmiany z jednej gałęzi do drugiej bez robienia pełnego merge'a. W przeciwieństwie do rebase, cherry-pick przesuwa bieżącą gałąź do przodu.

## 2.20. Zadanie 20 (rebase-complex)

```
git rebase issue-555 --onto your-master
```

Objaśnienie `git rebase --onto` pozwala na przeniesienie gałęzi w inne miejsce oraz daje bardzo precyzyjną kontrolę nad historią commitów. Przykładowo komenda będąca rozwiązaniem tego zadania oznacza: weź wszystkie commity, które nie są w issue-555 i umieść je w gałęzi your--master.

## 2.21. Zadanie 21 (invalid-order)

```
git rebase -i HEAD~2
# przenieść drugi commit nad pierwszy
```

Objaśnienie: *git rebase* w trybie interaktywnym pozwala na zmianę kolejności commitów w historii poprzez zmianę ich pozycji na edytowanej liście.

## 2.22. Zadanie 22 (find-swearwords)

```
git log -Sshit
git rebase -i
# zmiana "pick" na "edit" przy commitach wypisanych wcześniej

# zamiana słów w poszczególnych plikach dla kolejnych commitów
# dodawanie poprawionych plików przy pomocy git add
# kontynuowanie rebase przy pomocy git rebase --continue
```

Objaśnienie: *git log -S<fraza>* działa podobnie jak *grep* i pozwala na wyszukanie commitów, które wprowadziły jakieś słowo albo kod, którym jesteśmy zainteresowani.

## 2.23. Zadanie 23 (find-bug)

```
git bisect start
git bisect bad HEAD
git bisect good 1.0
git bisect run sh -c "openssl enc -base64 -A -d < home-screen-text.txt | grep -
v jackass"

git push origin ZNALEZIONE_COMMIT_ID:find-bug
```

Objaśnienie: *git bisect* to narzędzie, które pomaga znaleźć commit wprowadzający błąd poprzez systematyczne dzielenie historii commitów na pół (wyszukiwanie binarne).

*git bisect start* - rozpoczęcie sesji bisect

*git bisect bad* - oznaczenie aktualnego commita jako wadliwego

*git bisect good <nazwa_commita>* - oznaczenie znanego działającego commita

Git automatycznie wybierze commit w połowie między good i bad. Następnie można ręcznie sprawdzić czy błąd występuje i wtedy oznaczyć *git bisect bad* lub jeżeli nie występuje *git bisect good*. Proces się powtarza do momentu aż Git znajdzie wadliwy commit.

Można też zautomatyzować proces używając skryptu testowego:
*git bisect run sh -c „twój_skrypt”*

Skrypt powinien zwracać 0 dla dobrych commitów i liczbę różną od 0 dla złych commitów.