Zadania z https://gitexercises.fracz.com/

Oskar Blajsz

Zadanie 1. Commit One File

W zadaniu dane są dwa pliki **A.txt** i **B.txt**, trzeba zcommitować tylko jeden z nich. Możemy to zrobić przygotowując do commita tylko jeden z plików komendą:

git add A.txt

a następnie zcommitować komendą:

git commit -m „commit one file”

Zadanie 2. Commit one file of two currently staged

W tym zadaniu mamy dwa pliki już przygotowane do commita: **A.txt** i **B.txt**. Musimy usunąć z przygotowania jeden z nich komendą:

git reset A.txt

a następnie zcommitować komendą:

git commit -m „commit one file of two currently staged”

Zadanie 3. Ignore unwanted files

Zadanie polega na stworzeniu pliku **.gitignore**, który będzie ignorował:

- Wszystkie pliki .exe
- Wszystkie pliki .o
- Wszystkie pliki .jar
- Cały folder directiores

W tym celu dodamy plik o nazwie **.gitignore** i uzupełnimy go zawartością:

```
*.exe
*.o
*.jar
 libraries/
```

Następnie dodamy ten plik do commita komendą:

git add .gitignore

i zcommitujemy komendą:

git commit -m „add .gitignore”

Zadanie 4. Chase branch that escaped

W tym zadaniu mamy dane dwa branche: **chase-branch** i **escaped**. Branch **escaped** jest dwa commity do przodu względem brancha **chase-branch**. Chcemy aby **chase-branch** wskazywał na ten sam commit, co branch **escaped**. Możemy to osiągnąć mergując branch **escaped** na **chase-branch** komendą:

git merge escaped

znajdując się na branchu **chase-branch**.

Zadanie 5. Resolve a merge conflict

W tym zadaniu mamy dwa branche: **merge-conflict** i **another-piece-of-work**. Podczas mergowania obu tych branchy komendą:

git merge another-piece-of-work

Ponieważ znajdujemy się na branchu **merge-conflict**, powstaje konflikt w pliku **equation.txt**. Aby go rozwiązać należy wejść w ten plik, którego zawartość obecnie wygląda tak:

```
<<<<<<<<<< HEAD
2 + ? = 5
=======
? + 3 = 5
>>>>>>>>> another-piece-of-work
```

I zmienić ją na taką:

```
2 + 3 = 5
```

Następnie dodajemy te zmiany do commita:

git add equation.txt

i je commitujemy komendą:

git commit

tym razem bez -m, ponieważ przy rozwiązywaniu konfliktów nie musimy podawać wiadomości commita.

Zadanie 6. Saving your work

Zadanie polega na zapisaniu aktualnie wprowadzonych zmian „na boku" i zrobieniu szybkiego bugfixa ASAP. W tym celu najpierw odkładamy aktualnie wykonaną pracę „na półkę" komendą:

git stash

Następnie naprawiamy błąd w pliku **bug.txt** usuwając z niego jedną linię. Commitujemy bugfixa komendami:

git add bug.txt oraz *git commit -m „bugfix"*

Na końcu przywracamy „z półki" wcześniej odłożone zmiany komendą:

git stash pop

kończymy pracę dodając linię „Finally, finished it!" do pliku **bug.txt** i commitujemy zmiany takimi samymi komendami:

git add bug.txt oraz *git commit -m „finish work"*

Zadanie 7. Change branch history

W tym zadaniu chcemy zmodyfikować historię brancha **change-branch-history** o zmiany, które zostały wprowadzone po utworzeniu tego brancha, w branchu **hot-bugfix**. W tym celu użyjemy komendy:

git rebase hot-bugfix

znajdując się na branchu **change-branch-history**. Spowoduje to, że zmiany z brancha **hot-bugfix** zostaną wprowadzone na branchu **change-branch-history** i efekt będzie taki jakbyśmy utworzyli ten branch z brancha **hot-bugfix**. Można powiedzieć, że **hot-bugfix**, będzie nową bazą brancha **change-branch-history**.

Zadanie 8. Remove ignored file

W tym zadaniu chcemy przestać śledzić plik **ignore.txt**, który został dodany przed dodaniem pliku **.gitignore**. W tym celu skorzystamy z komendy:

git rm ignore.txt

a następnie zcommitujemy zmiany:

git commit -m „remove the file that should have been ignored”

Zadanie 9. Change a letter case in the filename of an already tracked file

W tym zadaniu chcemy zmienić nazwę śledzonego pliku **File.txt** na **file.txt**. W tym celu skorzystamy z komendy:

mv File.txt file.txt

Dzięki temu trikowi plik dalej będzie śledzony i będzie miał zmienioną nazwę. Następnie commitujemy zmiany:

git commit -m „Lowercase file.txt”

Zadanie 10. Fix typographic mistake in the last commit

W tym zadaniu chcemy zmodyfikować ostatni commit i usunąć literówkę, którą zcommitowaliśmy przez przypadek. Zmieniamy zatem literówkę w pliku **file.txt**, przygotujemy tę zmianę komendą:

git add file.txt

Następnie modyfikujemy ostatni commit o tę zmianę. W tym celu skorzystamy z komendy:

git commit --amend

Teraz tylko zmieniamy wiadomość commita za pomocą vima. Tutaj również poprawiamy literówkę i wychodzimy zapisując zmiany.

Zadanie 11. Forge the commit's date

W tym zadaniu należy zmienić datę ostatniego commita. W tym celu możemy skorzystać z komendy:

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

Zadanie 12. Fix typographic mistake in old commit

W tym zadaniu należy zmodyfikować starszego commita niż ostatni. W tym celu wykonujemy komendę:

*git rebase -i HEAD^^*

Następnie otworzy nam się w vimie plik tekstowy, w którym wybieramy które commity chcemy zmodyfikować zmieniając słowo „pick” na „edit” przy wybranych commitach. Nas interesuje commit oznaczony komentarzem „Add Hello Wordl”. Następnie zapisujemy plik.

Wtedy nasze repozytorium wróci do stanu po zcommitowaniu zmian w commicie, który wybraliśmy i możemy zmienić to co nas interesuje. W tym wypadku będzie to plik **file.txt**. Po dokonaniu w nim zmian wykonujemy kolejno komendy:

git add file.txt

git commit --amend

i tutaj ponownie odpali się plik tekstowy w vimie, w którym modyfikujemy commit message commita, który obecnie modyfikujemy. Po zmianach zapisujemy plik.

Teraz zakończyliśmy modyfikację commita i możemy wykonać komendę:

git rebase --continue

która przejdzie do modyfikacji kolejnego commita lub zakończy rebase’a (tak będzie w naszym przypadku).

Zadanie 13. Find a commit that has been lost

To zadanie polega na powrocie do pierwotnego stanu ostatniego commita, którego właśnie zmieniliśmy korzystając z git rebase. W tym celu najpierw wykonujemy komendę:

git reflog

która pokaże logi wszystkich wydarzeń, które wydarzyły się w repozytorium. Za pomocą niej możemy właśnie znaleźć commity, które zostały zmodyfikowane np. za pomocą git rebase.

Następnie wykonujemy komendę:

git reset --hard HEAD@{1}

która cofnie repozytorium do stanu sprzed ostatniej zmiany, czyli właśnie do stanu sprzed modyfikacji ostatniego commita.

Zadanie 14. Split the last commit

W tym zadaniu chcemy podzielić ostatni commit, w którego skład wchodziły pliki: **first.txt** oraz **second.txt** na dwa commity – osobne dla każdego pliku. W tym celu najpierw musimy cofnąć ostatniego commita za pomocą komendy:

*git reset HEAD^*

Teraz nasze repozytorium wróciło do stanu sprzed wykonania komendy „git commit...", zatem mamy dwa zmodyfikowane, ale jeszcze nie zcommitowane pliki. Teraz pozostaje zrobić dwa osobne commity dla każdego z plików za pomocą komend:

git add first.txt

git commit -m „first commit"

git add secound.txt

git commit -m „secound commit"

Po tym nasz pierwotny commit został podzielony na dwa oddzielne.

Zadanie 15. Too many commits

W tym zadaniu mamy 2 commity, które chcielibyśmy połączyć w jeden. W tym celu najpierw możemy je zobaczyć komendą:

git log -2

która pokaże nam 2 ostatnie commity. Następnie używając komendy:

*git rebase -i HEAD^^*

zaczynamy modyfikowanie commitów. W vimie otworzy nam się plik, w którym możemy wybrać akcje i commity. Przy najnowszym commicie dopisujemy słowo **„fixup"** lub **„squash"**, które spowodują połączenie tego commita z poprzednim. Użycie funkcji **„fixup"** spowoduje, że zostanie zachowana jedynie wiadomość ze starszego commita, natomiast **„squash"** pozwoli nam wybrać (lub ustawić) wiadomość wynikowego commita.

Następnie kończymy rebase komendą:

git rebase --continue

Zadanie 16. Executable

W tym zadaniu chcemy zmienić uprawnienia pliku **script.sh**, ponieważ obecnie nie posiada on uprawnień do wykonywania. Aby je dodać korzystamy z komendy:

git update-index --chmod=+x script.sh

Następnie commitujemy zmiany:

git commit -m „modify permitions”

Zadanie 17. Commit parts

W tym zadaniu chcemy podzielić zmiany wprowadzone w jednym pliku na dwa commity. W tym celu musimy wybrać, które linie będą należeć do pierwszego commita, a które do drugiego. Taki efekt możemy osiągnąć komendą:

git add -p file.txt

Teraz git podzieli nasz plik na fragmenty, gdzie zostały wprowadzone zmiany, tzw. hunki. Dla każdego takiego fragmentu musimy wybrać jedną z kilku opcji:

- y - dodaj ten fragment do staging area.
- n - nie dodawaj tego fragmentu.
- q - zakończ proces.
- a - dodaj ten fragment i wszystkie następne.
- d - nie dodawaj tego fragmentu i wszystkich następnych.
- s - podziel ten hunks na mniejsze kawałki.
- e - edytuj ten hunks ręcznie.
- p - wypisz obecny hunk
- k - pozostaw ten hunk bez decyzji i przejdź do poprzedniego hunka, który został pozostawiony bez decyzji
- K - pozostaw ten hunk bez decyzji i przejdź do poprzedniego hunka
- j - pozostaw ten hunk bez decyzji i przejdź do kolejnego hunka, który został pozostawiony bez decyzji
- J - pozostaw ten hunk bez decyzji i przejdź do kolejnego hunka
- / - szukaj hunka, który pasuje do podanego regexa
- g – wybierz hunk, do którego przejdziesz
- ? – wyświetl pomoc

Korzystając z tych opcji wybieramy odpowiednie linie do dodania do pierwszego commita, a następnie po rozstrzygnięciu wszystkich hunków commitujemy wybrane linie:

git commit -m „First commit”

Następnie dodajemy pozostałe linie do kolejnego commita i je commitujemy:

git add .

git commit -m „Secound commit”

Zadanie 18. Pick your features

W tym zadaniu chcemy „nałożyć” na siebie 3 branche (**feature-a**, **feature-b**, **feature-c**) tak aby były one widoczne jako pojedyncze commity na jednym, wynikowym branchu w odpowiedniej kolejności. W tym celu możemy użyć funkcji cherry-pick wykonując po kolei komendy:

git cherry-pick feature-a

git cherry-pick feature-b

git cherry-pick feature-c

W tym momencie pojawia nam się konflikt, który musimy rozwiązać w standardowy sposób (tak jak w **zadaniu 5**). Po tym zatwierdzamy rozwiązanie konfliktu i kończymy cherry-pick komendami:

git add program.txt

git cherry-pick --continue

Zadanie 19. Rebase complex

W tym zadaniu mamy trzy branche, których strukturę przedstawia poniższy schemat:

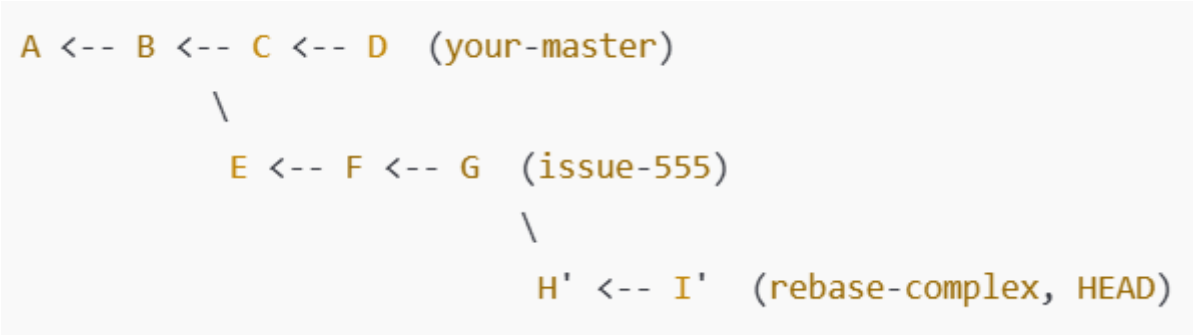
```
A <-- B <-- C <-- D  (your-master)
       \
        E <-- F <-- G  (issue-555)
         \
          H <-- I  (rebase-complex, HEAD)
```

Chcemy zrobić rebase jedynie zmian, które są na branchu **rebase-complex** (czyli commitów H oraz I) na najnowszy commit z brancha **your-master** (czyli commit D).

W tym celu rozważmy następującą sytuację. Gdybyśmy chcieli przenieść zmiany z brancha **rebase-complex** na najnowszy commit z brancha **issue-555** to znajdując się na branchu **rebase-complex** wykonalibyśmy komendę:

git rebase issue-555

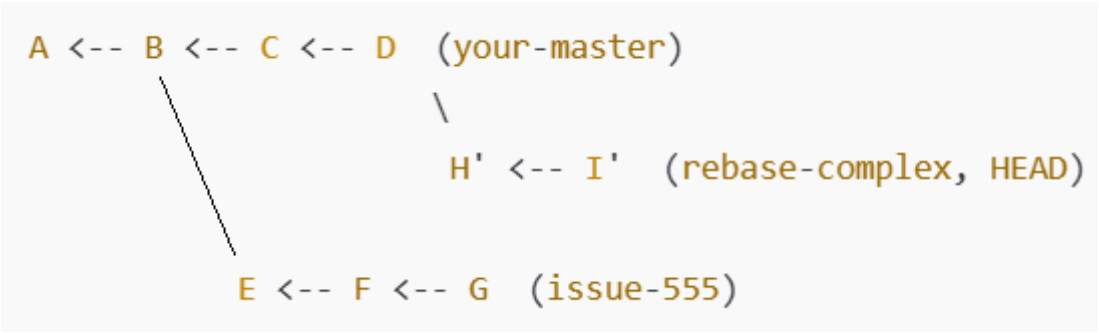
która przeniosłaby commit H oraz I na najnowszy commit z brancha **issue-555**, czyli commit G. Dokładnie taką samą sytuację mieliśmy w zadaniu 7. Po takiej komendzie nasze repozytorium wyglądałoby tak:



Teraz chcielibyśmy zmienić docelowy cel rebase'a dla tych commitów, tak aby nie przenosiły się one na górę brancha **issue-555**, tylko na górę brancha **your-master**. Możemy to zrobić dodając opcję --onto do naszej komendy:

git rebase issue-555 --onto your-master

Po jej wykonaniu repozytorium przybierze dokładnie taką formę o jaką nam chodziło:



Zadanie 20. Invalid order

W tym zadaniu chcemy zmienić miejscami w historii dwa ostatnie commity. Aby to zrobić możemy wykorzystać komendę:

*git rebase -i HEAD^^*

Następnie po prostu zamieniamy linie w pliku, który otworzy nam się w vimie. W taki sposób zamienimy kolejność commitów w historii.

Zadanie 21. Find swearwords

W tym zadaniu chcemy znaleźć wszystkie commity, które zawierają słowo **„shit”** i zamienić w nich to słowo na **„flower”**. W tym celu możemy skorzystać z komendy:

git reflog -Sshit

która pokaże nam ID commitów, zawierających słowo **„shit”**. Następnie chcemy wykorzystać rebase z opcją -i, aby dostać się do tych commitów, które znaleźliśmy i je zmodyfikować. W tym celu wywołujemy komendę:

*git rebase -i <najstarszy znaleziony commit>^*

I teraz dostaniemy listę wszystkich commitów od najnowszego do najstarszego commita, który znaleźliśmy. Następnie przy wszystkich wcześniej znalezionych commitach musimy zmienić słowo **„pick”** na **„edit”** i wyjść zapisując zmiany.

Teraz po kolei będziemy mogli modyfikować commity, które wybraliśmy. Po wprowadzeniu zmian w konkretnym commicie wywołujemy komendy:

git add .

git commit --amend --no-edit

git rebase --continue

Kiedy zmodyfikujemy wszystkie commity to rebase się zakończy i wszystkie zmiany będą wprowadzone.

Zadanie 22. Find bug

W tym zadaniu chcemy znaleźć pierwszy commit, w którym pojawiło się słowo **„jackass”** w pliku **home-screen-text.txt**. Problem w tym, że w każdym z 300 commitów zawartość tego pliku była modyfikowana oraz jest ona zaszyfrowana w Base64, więc użycie takiej komendy jak w poprzednim zadaniu nie zadziała. Z treści zadania wiemy, że ostatni commit z dobrą wersją posiada etykietę **„1.0”**

W tym celu na pewno będziemy korzystać z funkcji bisect, która za pomocą binsercha przeszuka wszystkie commity i pozwoli nam zdecydować czy obecnie sprawdzany commit jest dobry czy zły. W tym sposób w miarę szybko będziemy mogli znaleźć pierwszy zły commit.

Moglibyśmy robić to wszystko ręcznie. Zaczynając od komend:

git bisect start

git bisect bad HEAD

git bisect good 1.0

A następnie w każdej „iteracji” bisecta sprawdzać komendą:

openssl enc -base64 -A -d < home-screen-text.txt | grep jackass

czy plik **home-screen-text.txt** zawiera szukane słowo, czy nie i w zależności od tego wpisywać komendę: *git bisect good* lub *git bisect bad*.

Proces ten można natomiast zautomatyzować. Po rozpoczęciu bisectu komendami:

git bisect start

git bisect bad HEAD

git bisect good 1.0

Zamiast ręcznie weryfikować zawartość pliku **home-screen-text.txt** możemy komendą:

git bisect run sh -c "openssl enc -base64 -A -d < home-screen-text.txt | grep -v jackass"

ustawić automatyczne wywołanie sprawdzenia zawartości tego pliku w każdej „iteracji” bisecta. Jeśli zawiera on słowo **„jackass”** komenda zwróci **1** i commit zostanie zakwalifikowany jako zły, natomiast jeśli nie zawiera – komenda zwróci **0** i commit zostanie uznany za dobry. W ten sposób bardzo szybko automatycznie znajdziemy pierwszy commit, gdzie pojawiło się to słowo.

Na końcu za pomocą komendy:

git push origin COMMIT_ID:find-bug

Pushujemy znaleziony commit.