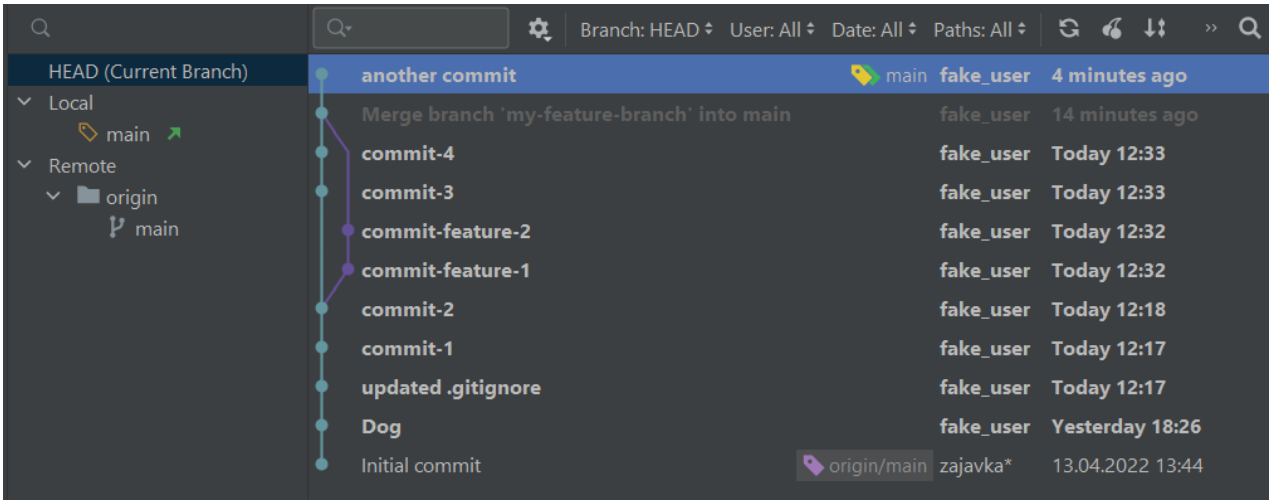
Git - Rebase

Spis treści

Rebase

Git posiada wbudowane dwa mechanizmy służące do łączenia zmian na różnych branchach. Pierwszy poznaliśmy wcześniej, był to **merge**. Drugim sposobem na złączenie zmian na branchach jest **rebase**.

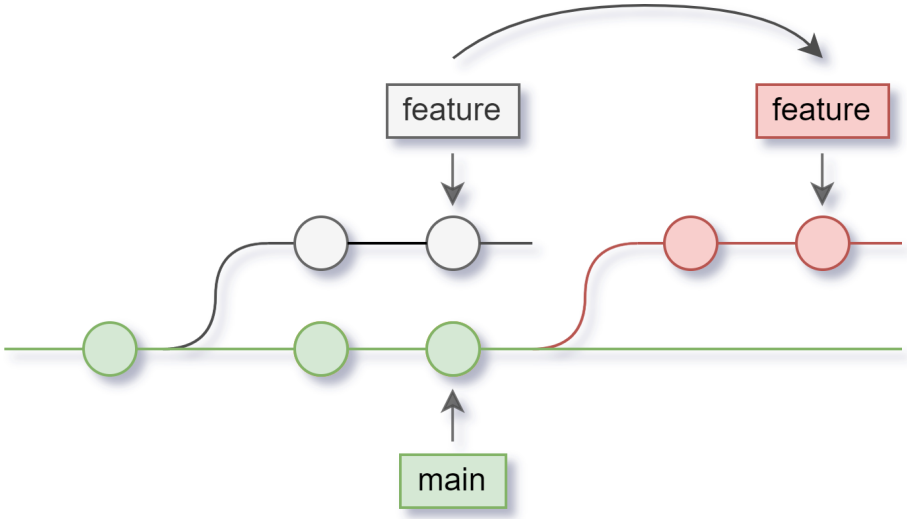
Wiemy już, kiedy **merge** tworzy nowy **merge commit**, który spina ze sobą dwa branche. Taki **merge commit** ma w sobie zaszytą informację mówiącą, które dwa commity są ze sobą spięte. Powoduje to, że nawet jak usuniemy branch, który został zmergowany, to historia zmian w projekcie nadal będzie wyglądała w ten sposób:



Obraz 1. Git merge

Merge jest operacją, która nie zmienia zawartości istniejącego brancha. Jeżeli dodawany jest **merge commit**, to spina on ze sobą historię obu branchy, co jednocześnie wpływa na strukturę przedstawioną wyżej. Prawdopodobne jest, że za każdym razem, gdy będziemy stosowali merge, będziemy dodawać **merge commit** (przypominam, że jest przypadek *fast-forward*). Jeżeli w projekcie, w którym pracujemy, będzie tworzone wiele branchy i często będą one mergowane ze sobą, może to doprowadzić do sytuacji, gdzie historia zmian będzie kompletnie nieczytelna.

Rebase różni się od **merge** swoim sposobem działania. Można powiedzieć, że **rebase** jest procesem przeniesienia commitów w nowe miejsce, przeniesienia ich na nową bazę. Najłatwiej będzie to wytłumaczyć, gdy odniesiemy się do poniższej grafiki:

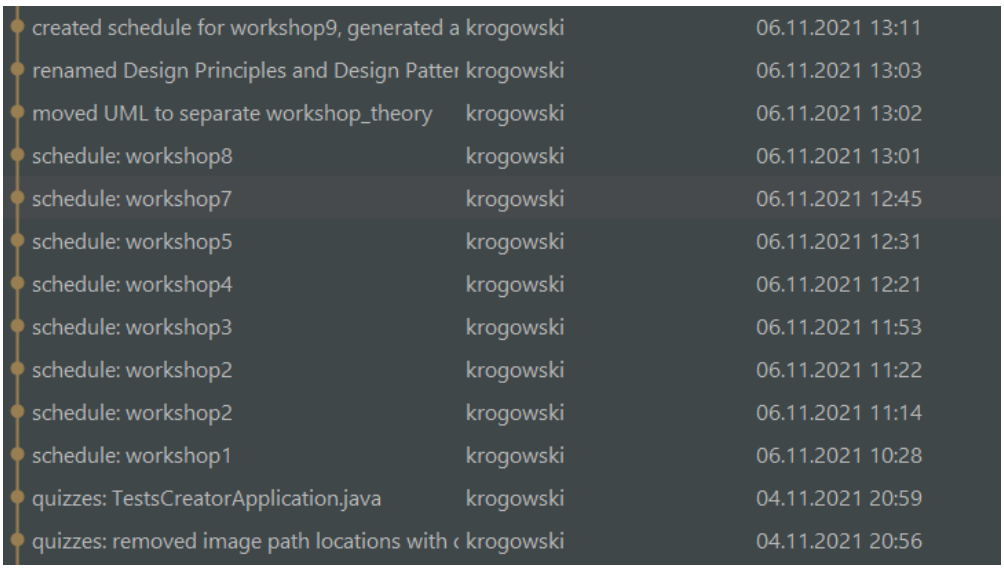


Obraz 2. Git rebase visualisation

Posługując się powyższą grafiką - **rebase** jest zmianą bazy brancha z jednego commita na inny. Wygląda to w taki sposób, jakbyśmy branch **feature** rozpoczęli z innego miejsca - dlatego właśnie mówi się o zmianie bazy. Gdy stusujemy **rebase**, **Git** zrobi to pod spodem, przeniesie bazę naszego brancha w inne miejsce. Ważne jest tutaj, żeby zrozumieć, że chociaż gałąź **feature** nadal będzie wyglądała tak samo, to będzie się ona składała z **całkiem nowych** commitów, o **treści takiej jak wcześniej**.

Rebase stosowany jest, żeby obejść problem nieczytelnej historii. Wróćmy do sytuacji, którą stworzyliśmy w poprzednich przykładach. Tworzymy branch **feature**, dodajemy do niego kilka commitów. W tym samym czasie nowe commity zostają dodane do brancha **main**. Oznacza to, że nasz branch nie opiera się już o aktualne zmiany. Gdy przyjdzie nam wykonać **merge**, możemy go wykonać poprawnie, ale stosowanie tego podejścia może doprowadzić do nieczytelnej historii zmian projektu. Możemy natomiast w takiej sytuacji najpierw wykonać **rebase**, czyli przenieść się ze swoimi zmianami na **feature** branch na nową bazę i dopiero wtedy spróbować zrobić **merge**. W ten sposób można osiągnąć czystą historię w projekcie i do takiego przykładu przejdziemy.

A co w ogóle oznacza czysta historia?



Obraz 3. Przykład mojego prywatnego repozytorium z projektami Zajavkowymi

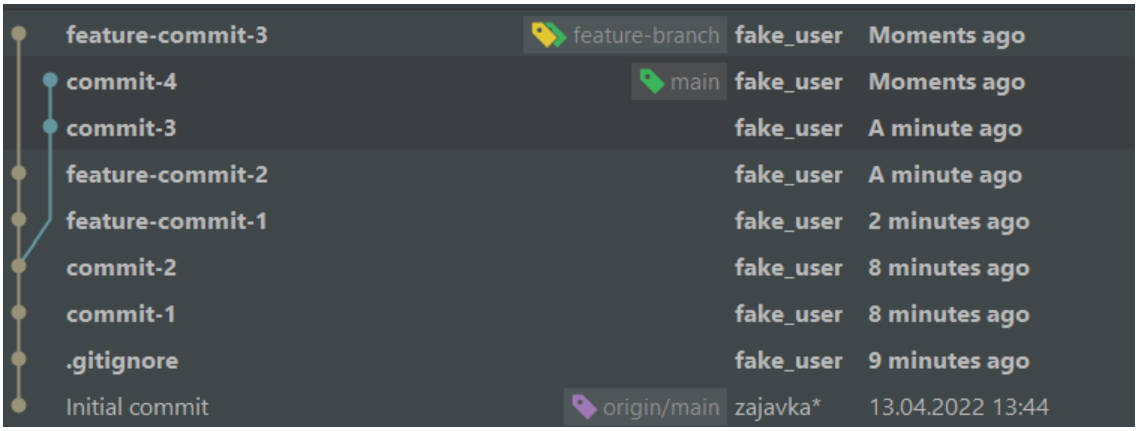
O wiele łatwiej jest się odnaleźć w zmianach, które wyglądają w ten sposób, niż w zmianach, które

wyglądają jak Guitar Hero. Nie mówiąc już o tym, że w praktyce będą występowały takie sytuacje, że będziemy potrzebowali znaleźć zmianę, która wprowadziła błąd w jakiejś funkcjonalności i w tym celu będziemy musieli przeszukiwać historię zmian.

Z tego, co zostało napisane wcześniej, można wywnioskować, że developer ma dwie możliwości, żeby finalnie jego/jej zmiany trafiły z **feature-branch** do **master**. Można albo od razu mergować zmiany wprowadzone na feature branchu, albo najpierw próbować zachować czystą historię stosując rebase, a potem mergować zmiany "na czysto". Druga opcja daje nam możliwość zachowania czystej historii i taki przypadek omówimy w dalszej części.

Zastosowanie praktyczne

Wróćmy do przykładu, który przerabialiśmy przy **merge**. Doprowadź repozytorium do takiego stanu:



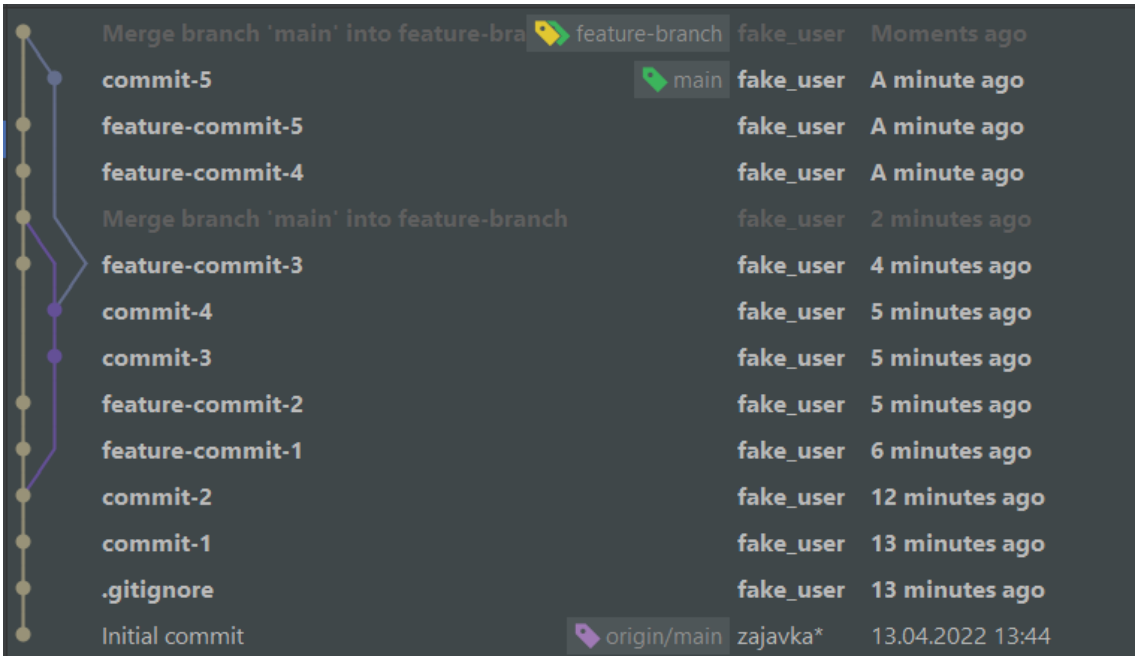
Obraz 4. Git rebase example

Czyli na etapie **commit-2** został stworzony branch **feature-branch**, do którego zostały dodane 3 commity. Jednocześnie do brancha **main** również zostały dwa commity, które są zmianami konfliktującymi ze zmianami z brancha **feature-branch**. W przykładzie, jaki widzieliśmy wcześniej (w analogicznej sytuacji), pracowalibyśmy aż do skończenia jakiegoś zadania na branchu **feature-branch** i zrobili merge do brancha **main**. Pojawia się zatem pytanie, czy można to zrobić inaczej?

Opcja 1: Możemy co jakiś czas próbować aktualizować branch **feature-branch** zmianami, które są dodawane do brancha **main**. W takim przypadku co jakiś czas musielibyśmy mergować **main** → **feature-branch** i dopiero jak skończylibyśmy pracę, to wykonalibyśmy merge **feature-branch** → **main**. Podejście takie jest o tyle rozsądne, że jeżeli do **main** cały czas dodawane są zmiany (bo pracują z nami ludzie w zespole), to staramy się nad takimi zmianami nadążyć, żeby nie zrobić sobie potem dużo merge conflictów do rozwiązania. Czyli co jakiś czas wykonywalibyśmy komendę:

```
git merge main
# znajdując się na branchu feature-branch
```

Załóżmy, że przez to, że co jakiś czas, pracując na branchu **feature-branch**, staraliśmy się nadążyć ze zmianami, które są dodawane do **main**, doprowadziliśmy historię do takiego momentu:



Obraz 5. Git rebase example

Commity były nazywane specjalnie w ten sposób, żeby łatwiej Ci było zrozumieć, który w kolejności był dany commit i na którym branchu był dodawany.

Doszliśmy do momentu, gdzie skończyliśmy pracę na branchu **feature-branch** i chcemy ją scalić z główną gałęzią - **main**. Zatem będąc na gałęzi **main**, wykonujemy polecenie:

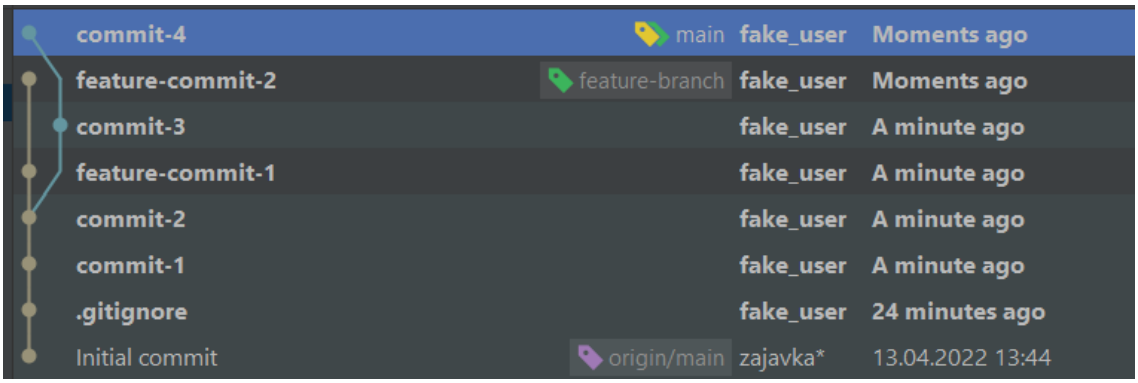
```
git merge feature-branch
```

Tym razem operacja ta przebiegła bez konfliktów, bo co chwila aktualizowaliśmy stan brancha **feature-branch** o dane z brancha **main**. Dzięki temu gdy doszliśmy do końca pracy na branchu **feature-branch** okazało się, że nie mamy konfliktów, bo rozwiązywane one były na bieżąco jeżeli się jakieś pojawiły. A jak wygląda historia?

Obraz 6. Git rebase example

Można powiedzieć, że nie jest ona zbyt czytelna. Tutaj jest o tyle prosto, że commity były nazywane specjalnie w ten sposób, żeby łatwo można było zrozumieć kolejność ich dodawania, ale gdy będą to wiadomości, które mają opisywać zmiany, zrozumienie tego będzie trudne. Jak zatem można podejść do tematu wykorzystując **rebase**?

Opcja 2: Nadal możemy co jakiś czas próbować aktualizować branch **feature-branch** zmianami, które są dodawane do brancha **main**, tyle że tym razem będziemy do tego używali **rebase**. W takim przypadku co jakiś czas będziemy rebasować **feature-branch NA main** i dopiero jak skończymy pracę, to wykonamy merge **feature-branch → main**. Podejście takie jest o tyle rozsądne, że jeżeli do **main** cały czas dodawane są zmiany (bo pracują z nami ludzie w zespole), to staramy się nad takimi zmianami nadążyć, żeby nie zrobić sobie potem dużo merge conflictów do rozwiązania. W tym przypadku jednak, to nadążanie, będzie wiązało się z wykorzystaniem **rebase**, a nie **merge**. Zanim jednak to zrobimy, doprowadź ponownie repozytorium do stanu widocznego na poniższym obrazku:



Obraz 7. Git rebase example

Przełącz się teraz na branch **feature-branch** i wykonaj komendę:

```
git rebase main
```

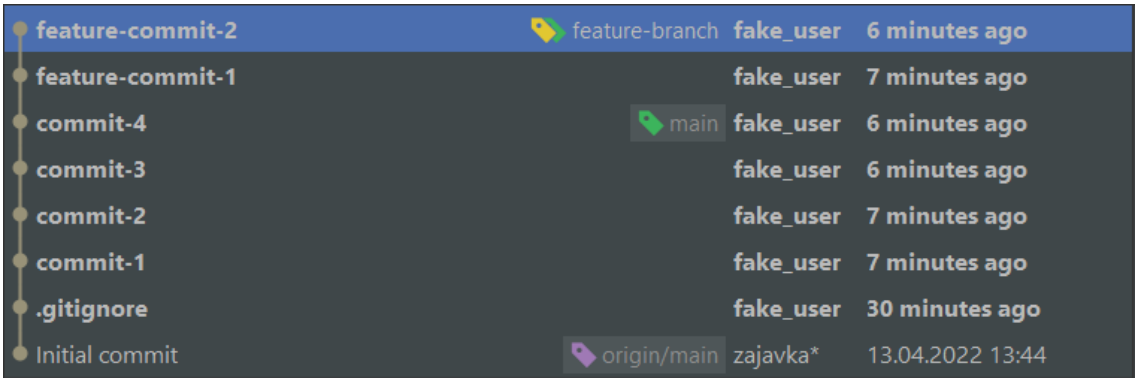
Możesz w takiej sytuacji otrzymać konflikt - trzeba go będzie wtedy rozwiązać. Jeżeli nie jesteś w stanie rozwiązać konfliktu, możesz ratować się poniższym poleceniem. Spowoduje ono porzucenie **rebase**.

```
git rebase --abort
```

Jeżeli natomiast uda Ci się poprawnie rozwiązać konflikty, musisz wtedy wykonać dwa polecenia:

```
git add Dog.java
git rebase --continue
```

Jak wtedy będzie wyglądała historia zmian w repozytorium?



Obraz 8. Git rebase example

Czyli "początek" brancha **feature-branch** został przeniesiony na ostatni commit na branchu **main**. Baza brancha **feature-branch** uległa zmianie - stąd to polecenie nazywa się **rebase**. Widać też jednocześnie, że historia jest o wiele czystsza, niż gdy robiliśmy co chwila merge **main** → **feature-branch**. Możemy teraz powtarzać tę czynność wielokrotnie, czyli jeżeli do brancha **main** zostaną dodane jakieś nowe zmiany, możemy je "wciągnąć" do brancha **feature-branch** właśnie robiąc **rebase**.

Co zrobić gdy skończymy już pracę na branchu feature-branch?

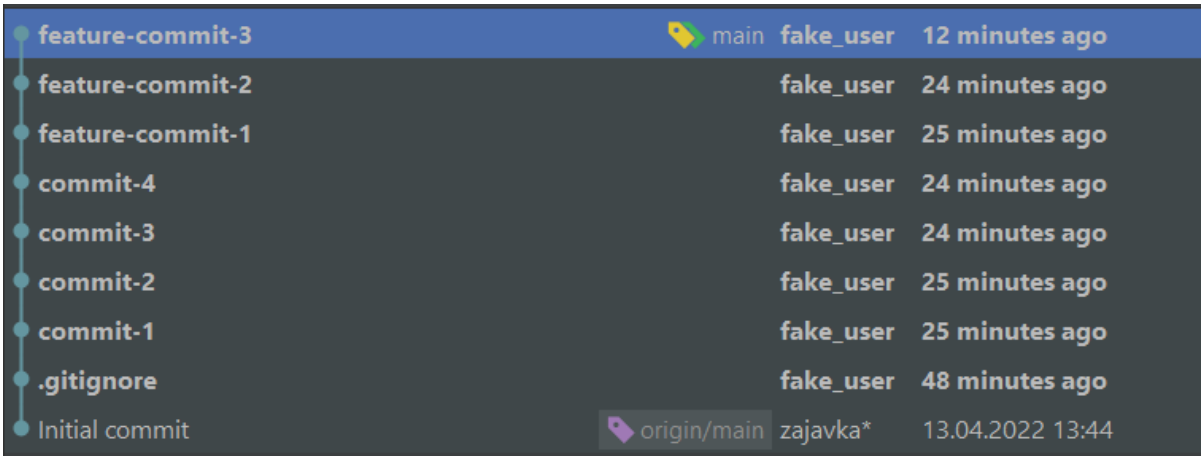
Nadal musimy zrobić **merge**, tylko że **Git** potraktuje teraz taki merge inaczej. Jeżeli przełączysz się na branch **main** i wykonasz polecenie:

```
git merge feature-branch
```

To na ekranie zostanie wydrukowana taka wiadomość:

```
Updating b075638..fa58be3
Fast-forward
 Dog.java | 1 +
 1 file changed, 1 insertion(+)
```

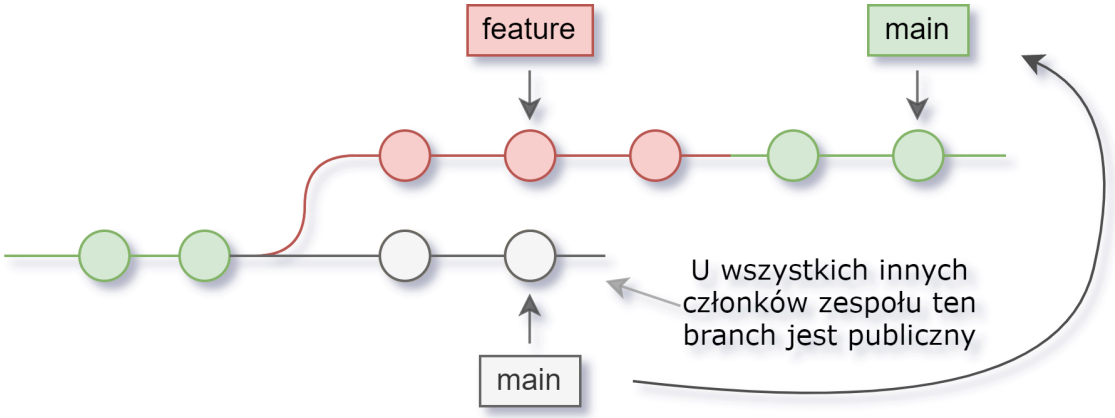
Czyli nie został dodany **merge commit**, bo udało nam się wykonać merge **fast-forward**. A to dzięki temu, że wcześniej na bieżąco robiliśmy **rebase**. Dzięki takiemu podejściu historia zmian w projekcie wygląda teraz tak:



Obraz 9. Git rebase example

Kiedy tego nie robić

Gdy już zrozumiesz, czym jest **rebase**, najważniejsze jest to, żeby pamiętać, kiedy go **nie** używać. Najważniejsze co musisz zapamiętać, to żeby nie używać go na **głównych gałęziach**. Wyobraź sobie teraz, co by się stało, gdybyśmy zrobili rebase **main NA feature-branch**.



Obraz 10. Git rebase wrong example

Czyli w powyższym przykładzie, przenosimy commity z brancha **main** na koniec brancha **feature**. Problem z tym jest taki, że możemy to zrobić tylko w naszym lokalnym repozytorium. Gdybyśmy chcieli teraz wypchać nasze lokalne zmiany do zdalnego repozytorium, to okaże się, że nie ma takiej możliwości. Wynika to z tego, że każdy z członków zespołu pracuje już z główną gałęzią w postaci takiej, jaka była, zanim zrobiliśmy ten "zły" rebase. Czyli musimy zapamiętać, że nie wolno nam robić **rebase** gałęzi, która jest współdzielona przez innych członków zespołu.

Widać jednocześnie, że **rebase** jest komendą, która ingeruje w historię brancha wstecz. Tzn., jeżeli pracujemy tylko na naszym branchu, możemy modyfikować jego historię wstecz, bo tylko my na nim pracujemy i finalnie i tak zmergujemy go do gałęzi głównej. Gałąź główna jest współdzielona przez innych programistów - nie możemy modyfikować wstecz historii, z której korzystają inni. Inaczej można to ująć w ten sposób, że **rebase** jest destrukcyjny - przepisuje historię. Dlatego przed jego użyciem trzeba się zastanowić czy robimy to w poprawnym kontekście.

Merge vs Rebase

Poruszyliśmy wcześniej cechy każdej z operacji i zaznaczyliśmy, kiedy każda z nich powinna być używana. Zaznaczam tutaj, że dosyć istotne jest umieć porównać te dwie operacje ze sobą. Dlatego podsumujmy to w skrócie:

Rebase:

- Otrzymujemy czystszą historię, bo dzięki stosowaniu **rebase** możemy pozbyć się **merge commitów**.
- Bardziej skomplikowany w użyciu.
- Trzeba uważać, żeby przez przypadek nie zrebasować publicznego brancha.

Merge:

- Łatwiejszy w użyciu.
- Zachowuje kolejność historyczną.
- Przy często zmieniających się repozytoriach może doprowadzić do bardzo skomplikowanej w odczycie i zrozumieniu historii zmian.

Wrócimy jeszcze do tego tematu, gdy zaczniemy pracować z repozytorium zdalnym. Będziesz mieć wtedy dobrą okazję, żeby przećwiczyć te polecenia jeszcze raz.