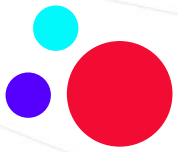


GIT – Part 2



HELLO

Damian Filipkowski





Agenda

1. Powtórka
2. Git Flow
3. Merge
4. Rebase
5. Merge vs Rebase
6. Konflikty
7. GitHub, GitLab, GitTortoise etc.
8. Pull Request i Code Review

Krótką powtórka :)

1. clone ?
2. commit ?
3. pull ?
4. push ?
5. revert ?
6. reset ?
7. add ?
8. init ?
9. log ?
10. status ?



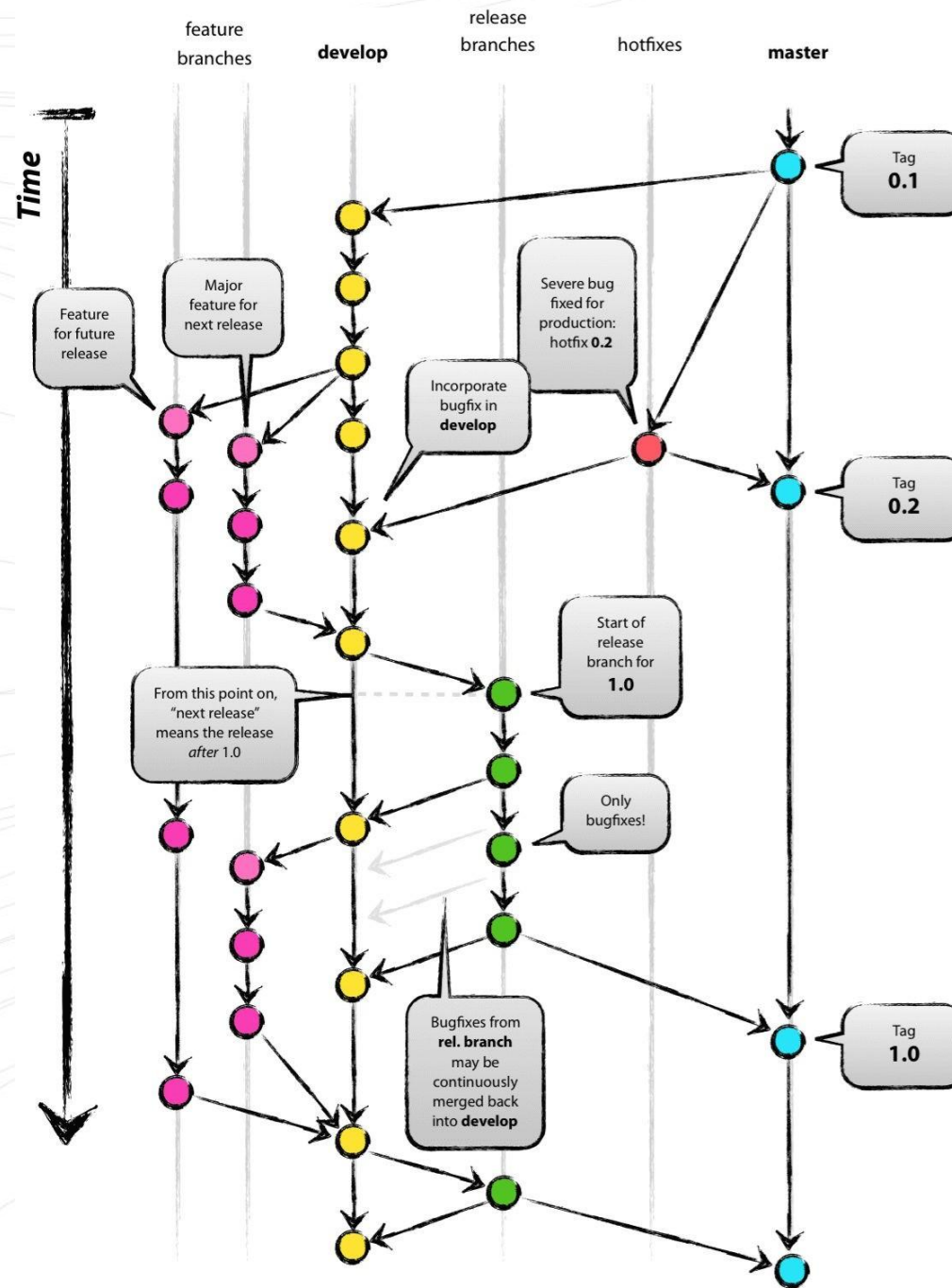
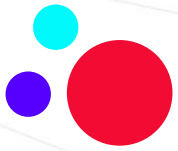
Powtórka

1. Branch ?
2. Commit ?
3. HEAD vs DETACHED HEAD ?
4. git log ?
5. .gitignore ?
6. .gitkeep ?
7. git -m commit "???" ?
8. reset
9. add
10. init

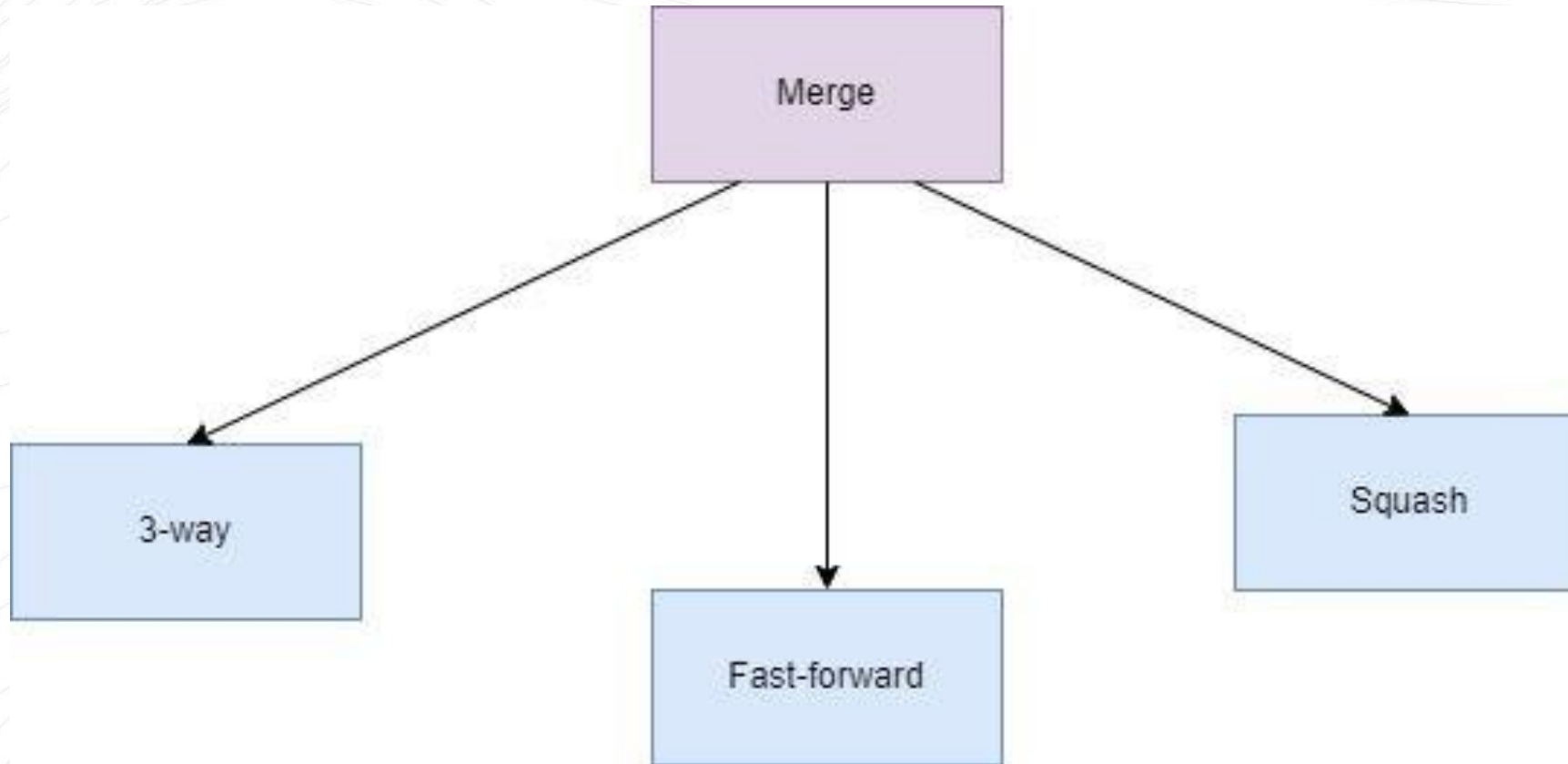




- przez flow rozumiemy ustalony i znany zestaw operacji wykonywanych w powtarzalnym procesie
- GIT nie posiada określonych zasad jego użytkowania, jest tylko narzędziem, sami możemy ustalić własne zasady
- Flow używamy w pracy z GIT-em w celu ustandaryzowania procesów tworzenia oraz scalania gałęzi
- Flow pomaga określić zastosowania tzw. długo żyjących gałęzi(gałęzi głównych)

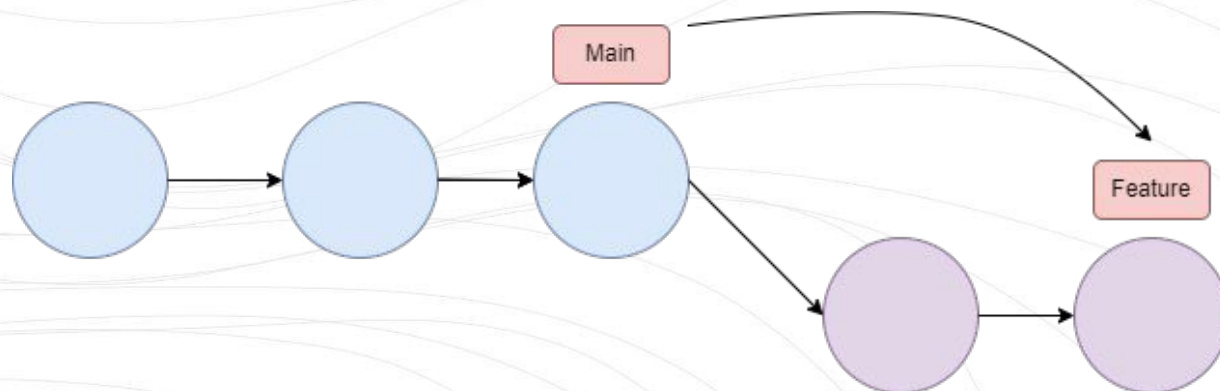






Fast-forward

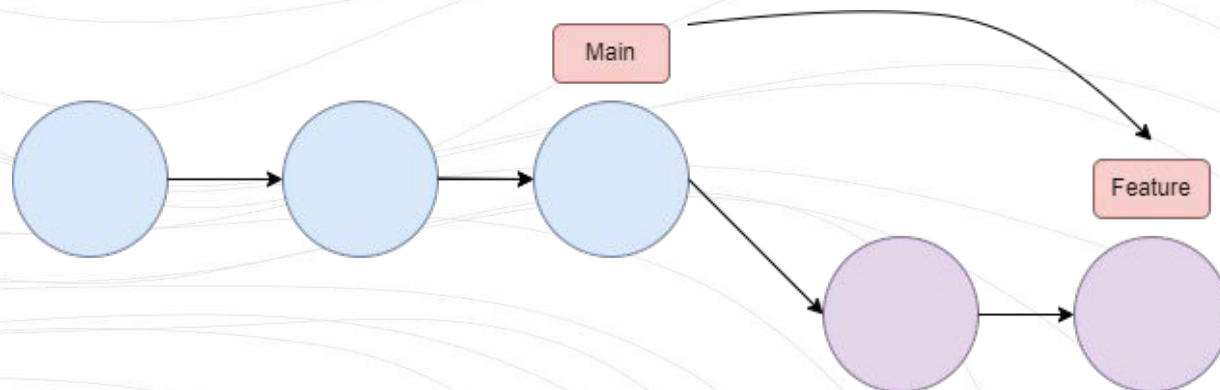
Wyobraźmy sobie sytuację, że musimy dodać do naszej aplikacji jakąś nową funkcjonalność. Pracujemy na branchu **main**, tworzymy z niego nowy branch **feature**, robimy jeden lub kilka **commitów**. W międzyczasie na branchu **main** nikt nic nie dodał. Chcemy **zmergować** (scalić) nasze zmiany z branchem **main**.





Fast-forward

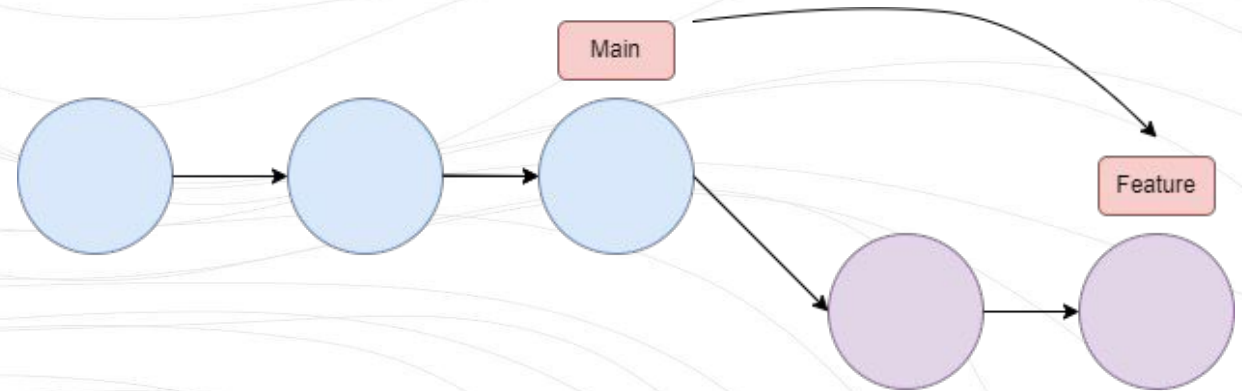
Rodzaj merge'a z którym mamy do czynienia gdy na branchu docelowym nie zostały dokonane żadne zmiany od momentu utworzenia brancha źródłowego (commit na który wskazuje head brancha docelowego jest commitem wyjściowym dla naszego brancha źródłowego). W tym przypadku nie jest tworzony merge commit tylko **HEAD** brancha docelowego jest przenoszony na head brancha źródłowego.





Fast-forward

```
git checkout main  
git checkout -b feature  
git commit -m 'commit 1'  
git commit -m 'commit 2'  
git checkout main  
git merge feature
```



Fast-forward Quest

1. Sprawdź na jakim branchu jesteś
2. Przenieś się na branch **main**
3. Stwórz branch feature z brancha main
4. Przenieś się na branch **feature**
5. Stwórz folder ze swoim imieniem
6. Dodaj plik **author.py** do repozytorium
7. **Zcommituj** zmiany
8. Wróć na branch **main**
9. Zmerguj branch **feature** do **main**
10. Wyślij na chacie jaką metodą został zmergowany
11. Sprawdź historię **commitów**



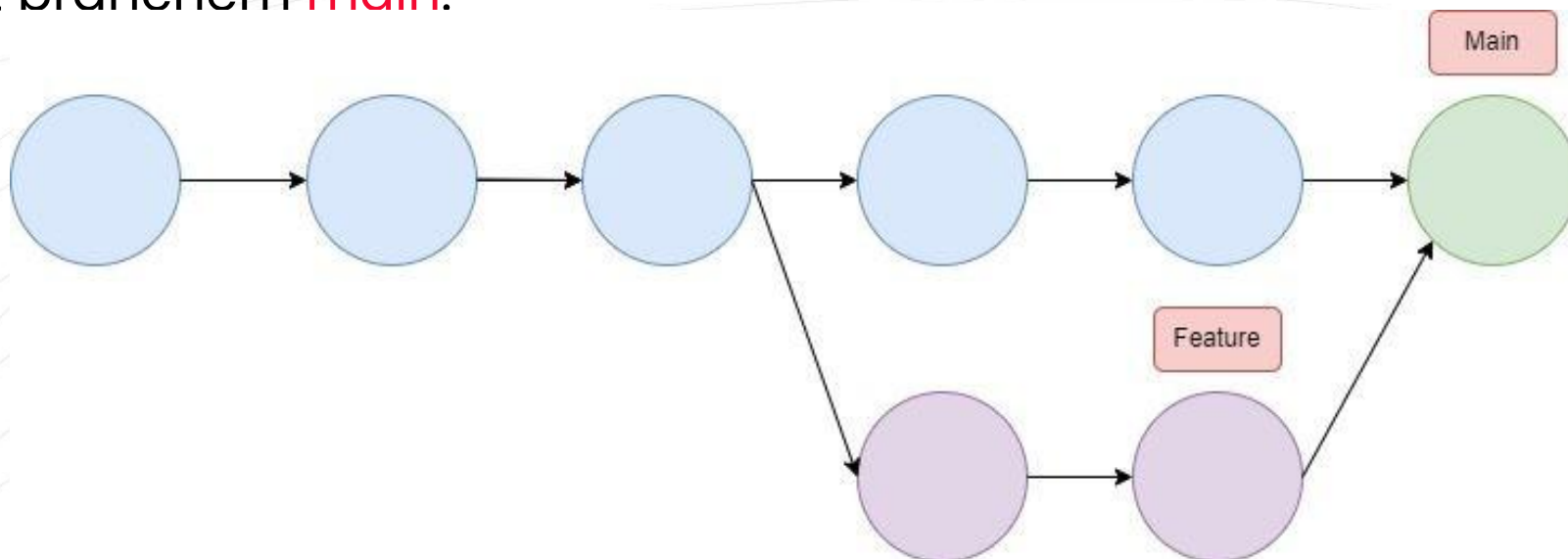


Fast-forward flagi

- **-ff** flaga domyślna przy używaniu polecenia merge określa że w przypadku gdy jest możliwe mergowanie fast-forward powinno to nastąpić
- **-no-ff** wywołanie merge z tą flagą powoduje że zawsze zostanie utworzony merge-commit nawet jeśli fast-forward było możliwe
- **-ff-only** próbuje wymusić fast-forward, gdy nie jest to możliwe zwraca komunikat błędu

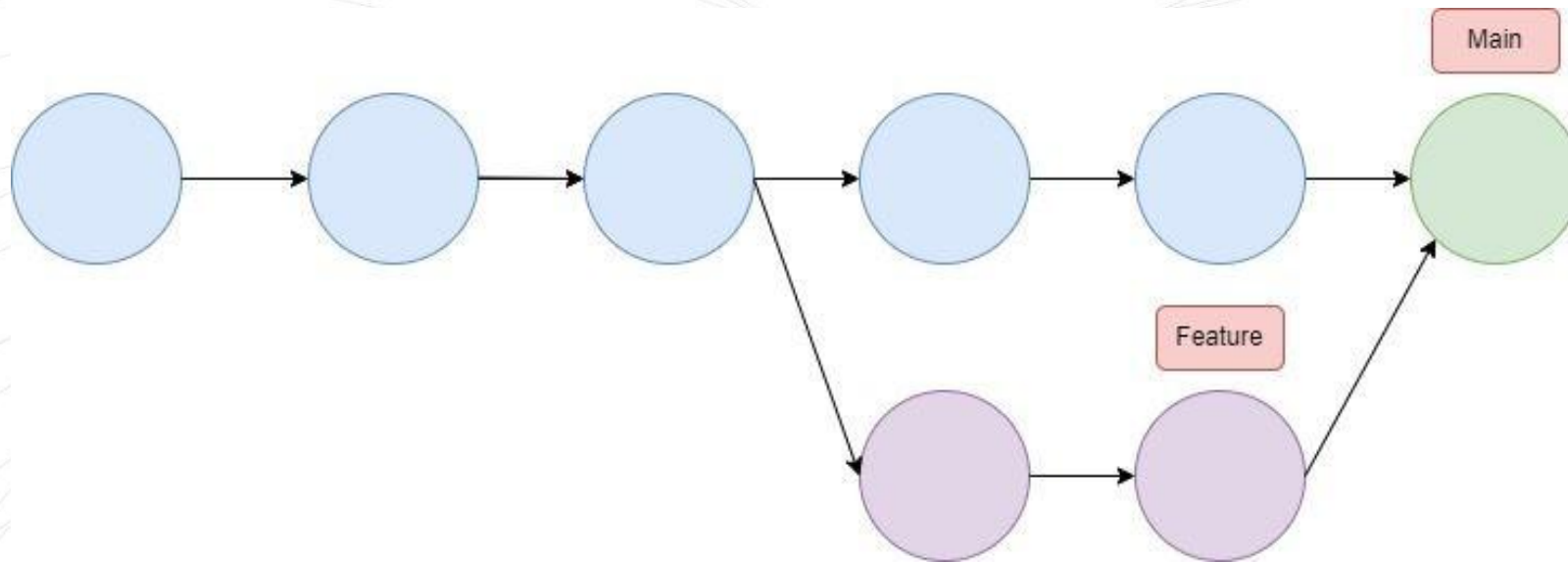
3-Way Merge

Podobnie jak ostatnio, wyobraźmy sobie sytuację, że musimy dodać do naszej aplikacji jakąś nową funkcjonalność. Pracujemy na branchu **main**, tworzymy z niego nowy branch **feature**, robimy jeden lub kilka **commitów**. Tym razem w międzyczasie na branchu **main** pojawiły się inne **commity**. Nasz kolega, albo my sami dodaliśmy tam inną funkcjonalność. Tak jak poprzednio chcemy **zmergować** nasze aktualne zmiany z z branchem **main**.



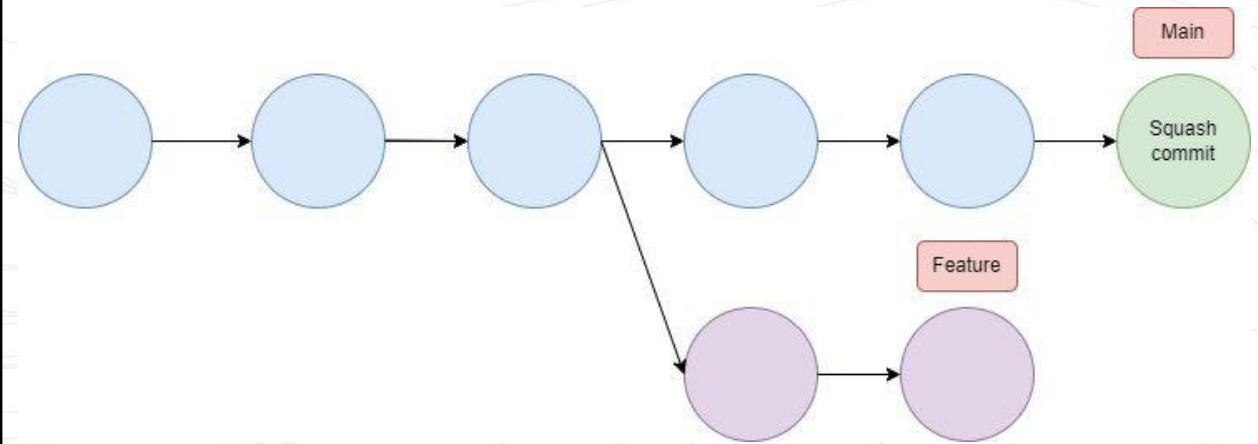
3-Way Merge

Rodzaj merge'a, z którym mamy do czynienia w sytuacji gdy łączymy zmiany pomiędzy dwoma branchami bez wspólnego przodka. Wynikiem operacji jest nowy **commit** (merge commit)



3-Way Merge

```
git checkout main
git checkout -b feature
git commit -m 'commit 3'
git commit -m 'commit 4'
git checkout main
git commit -m 'commit 5'
git commit -m 'commit 6'
git merge feature
```



3-Way Merge - VIM



<https://devhints.io/vim>

```
h j k l    arrow keys

:i          enter insert mode
Esc         quit insert mode

:q          close file
:w          write file
:wq         write and close
```


3-Way Quest

1. Sprawdź na jakim branchu jesteś
2. Przenieś się na branch **main**
3. Stwórz branch feature z brancha main
4. Przenieś się na branch **feature**
5. Dodaj plik **title.py** do repozytorium (wpisz tytuł)
6. **Zcommituj** zmiany
7. Wróć na branch **main**
8. Wpisz swoje imię w pliku **author.py**
9. Zmerguj branch **feature** do **main**
10. Wyślij na slacku jaką metodą został
zmergowany
11. Sprawdź historię **commitów**



Klasyczny Merge – wady i zalety

Klasyczny merge łączy zmiany z gałęzi **D** i **E** do mastera za pomocą merge commita (**F**).

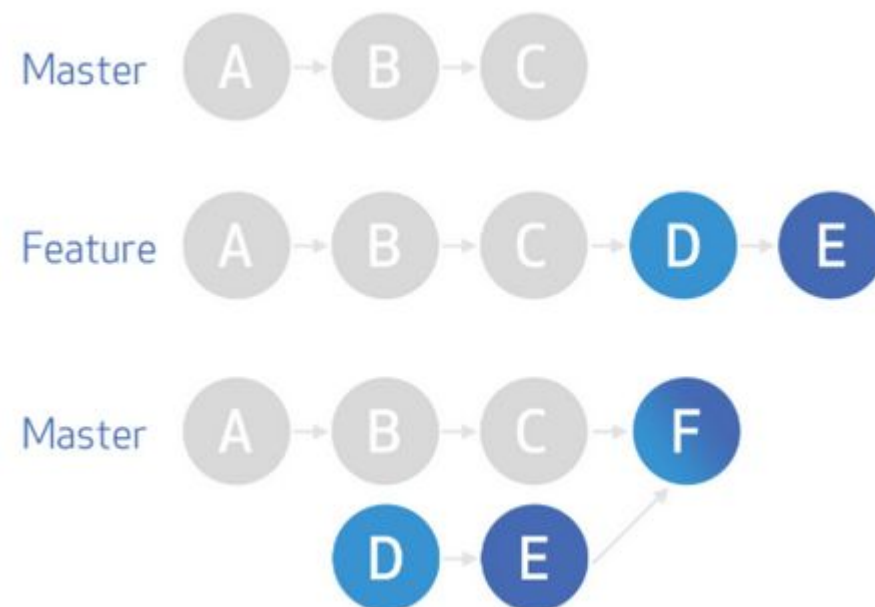
W historii gałęzi master wystąpią wszystkie commity od **A** do **F**.

Taka historia ma kilka wad:

- Do mastera dołączane są nasze commity częściowe **D** i **E** zaciemnia to historię zmian i utrudnia proces przeglądu kodu w ramach **Code Review Pull Requestów**.
- Oprócz naszych commitów częściowych pojawia się dodatkowy **merge commit F** (chyba, że jest możliwość przeprowadzenia Fast-Forward).
- Commity są poprzepłatane – zgodnie z czasem ich wykonania – w gałęzi master pojawi się więc mętlik.

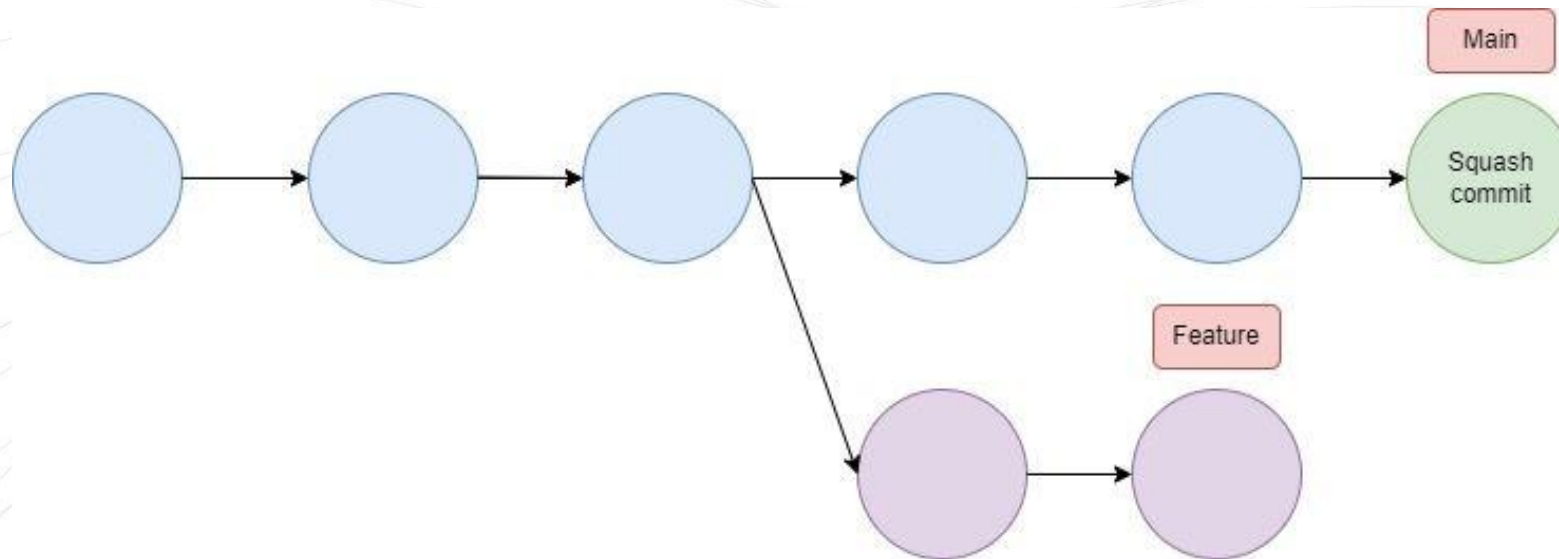
Zalety klasycznego Merge

- Klarowna historia powiązań między gałęziami. Dzięki Merge Commitom dokładnie wiemy, która gałąź weszła do której i w jakim czasie.



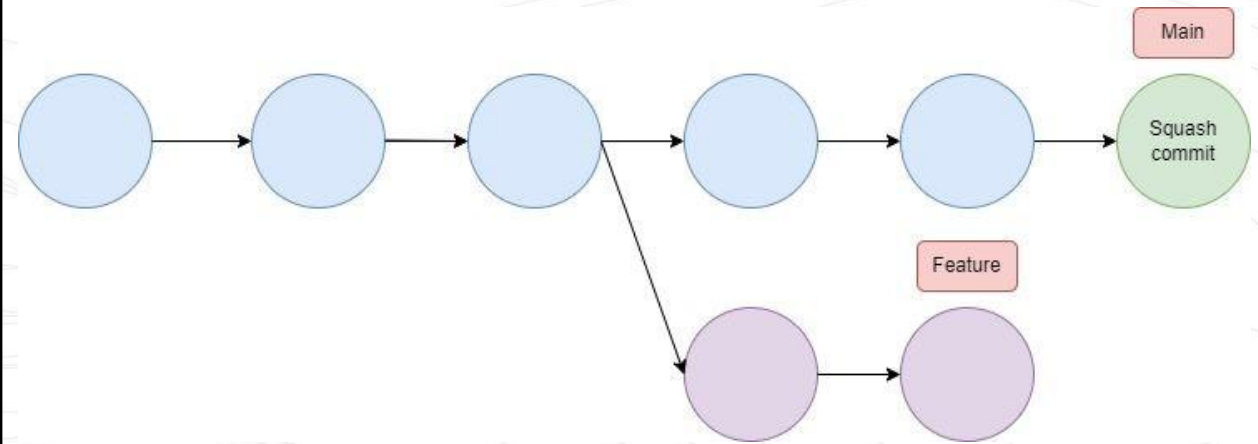
Squash Merge

Merge, który wywoływany jest flagą `--squash` polega na spłaszczeniu commitów z brancha źródłowego do pojedynczego commita, który jest dodawany do brancha docelowego.



3-Way Merge

```
git checkout main
git checkout -b feature
git commit -m 'commit 5'
git commit -m 'commit 6'
git checkout main
git commit -m 'commit 7' # optional
git commit -m 'commit 8' # optional
git merge feature --squash
```



Squash Quest

1. Sprawdź na jakim branchu jesteś
2. Przenieś się na branch **main**
3. Stwórz branch feature z brancha main
4. Przenieś się na branch **feature**
5. Dodaj plik **story.py** do repozytorium
6. Zrób kilka **commitów**
7. Wróć na branch **main**
8. Zrób zmiany w pliku **author.py** lub **title.py**
9. Zmerguj branch **feature** do **main** z flagą **squash**
10. Wyślij na slacku jaką metodą został zmergowany
11. Sprawdź historię **commitów**



Squash Merge – zalety i wady

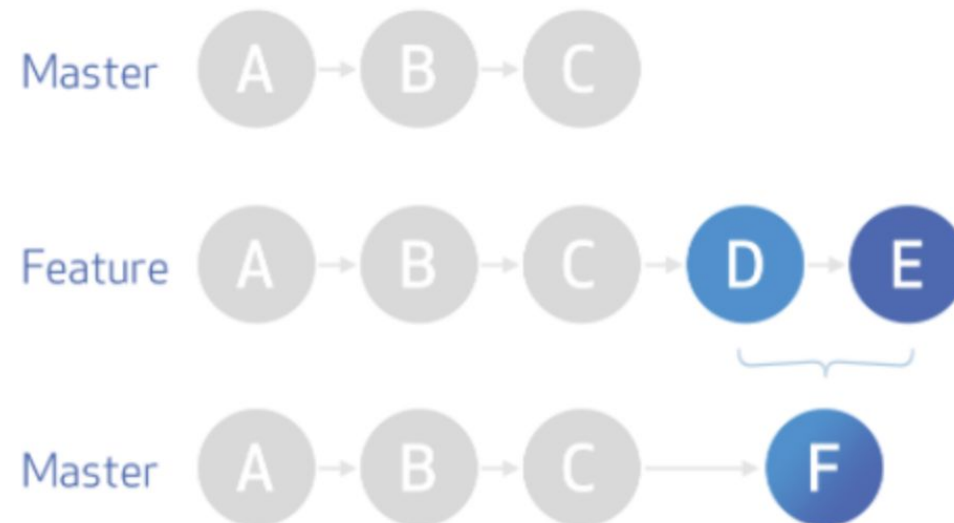
Wszystkie zmiany (**D** i **F**) zrealizowane na branchu feature zostaną spakowane jako jeden commit i dołączone do gałęzi master.

Zalety Squash Merge

- Najprostsza i najbezpieczniejsza metoda scalania
- Squash jest bezpiecznym sposobem przekazywania zmian z jednej gałęzi do drugiej – wykonując **squash**, podobnie jak merge – na pewno nie stracimy żadnych zmian.
- otrzymamy liniową historię **commitów**
- unikamy **merge commita**

Wady:

- tracimy jednak informację o pochodzeniu zmiany – nie mamy informacji, z której gałęzi ona pochodzi.



GIT Merge - linki

<https://git-scm.com/docs/git-merge>

<https://www.youtube.com/watch?v=zOnwgxiC0OA>

<https://www.arturnet.pl/index.php/2021/10/03/git-zaawansowane-funkcje-cz-1-head-i-utrata-glowy/>





infoShareAcademy.com

infoShare
ACADEMY

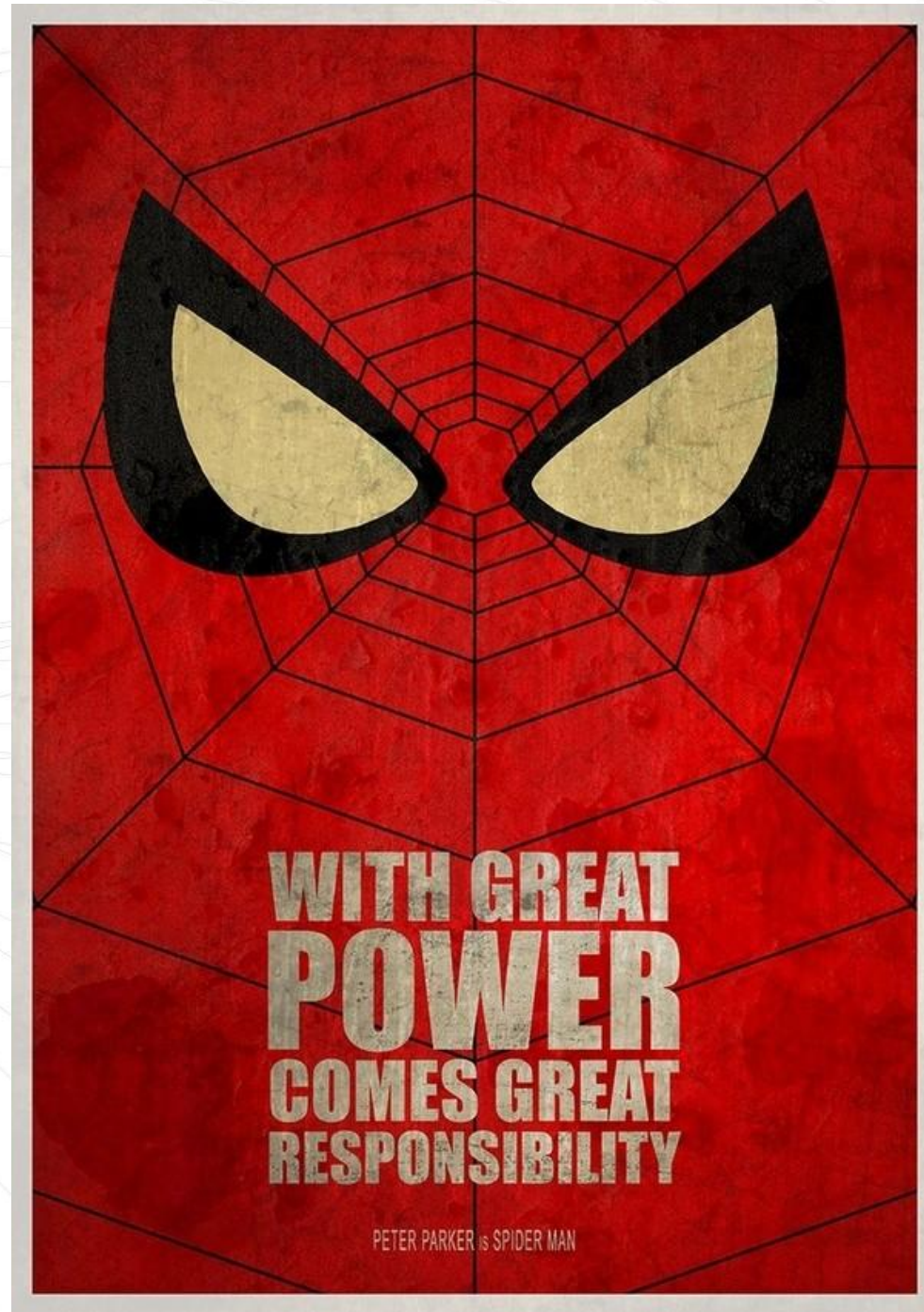


Rebase

Rebase to w najprostszym wydaniu skopiowanie szeregu commitów z jednej gałęzi do drugiej.

Commity są **kopiowane**, mają nowe **hashe** i nie są powiązane z commitami źródłowymi. **To całkowicie nowe commity.**

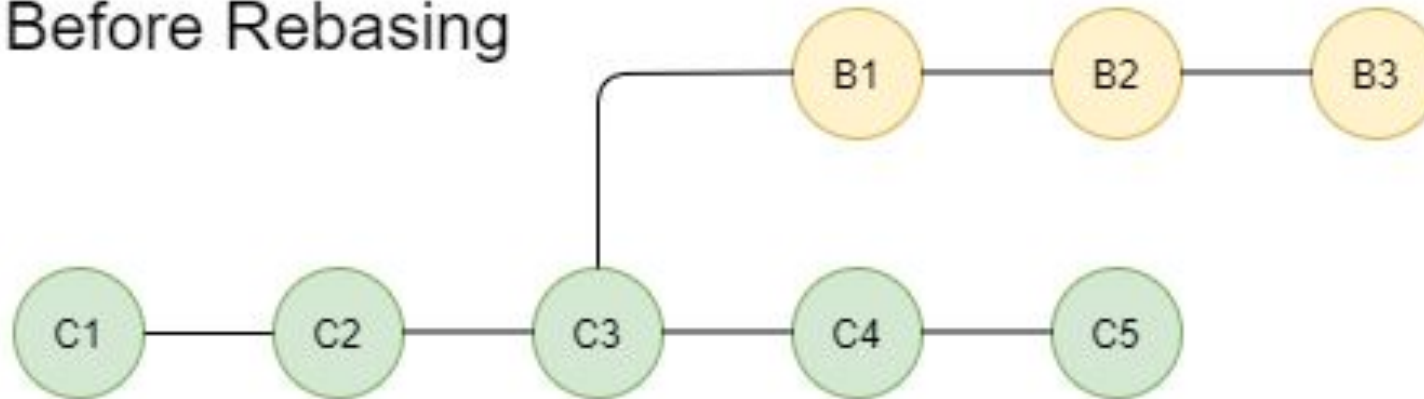
Rebase to procedura zmiany bazy naszej gałęzi z jednego commita na inny, sprawiająca, że rodzic naszej gałęzi ulega modyfikacji, finalnie w GIT mamy stan jakbyśmy wyszli od innego commitu.



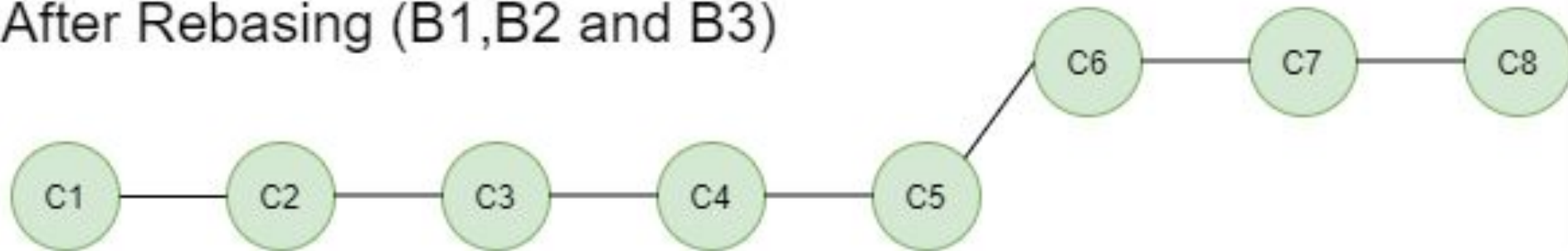


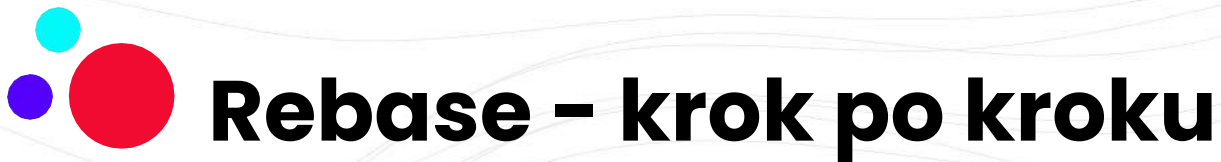
Rebase

Before Rebasing



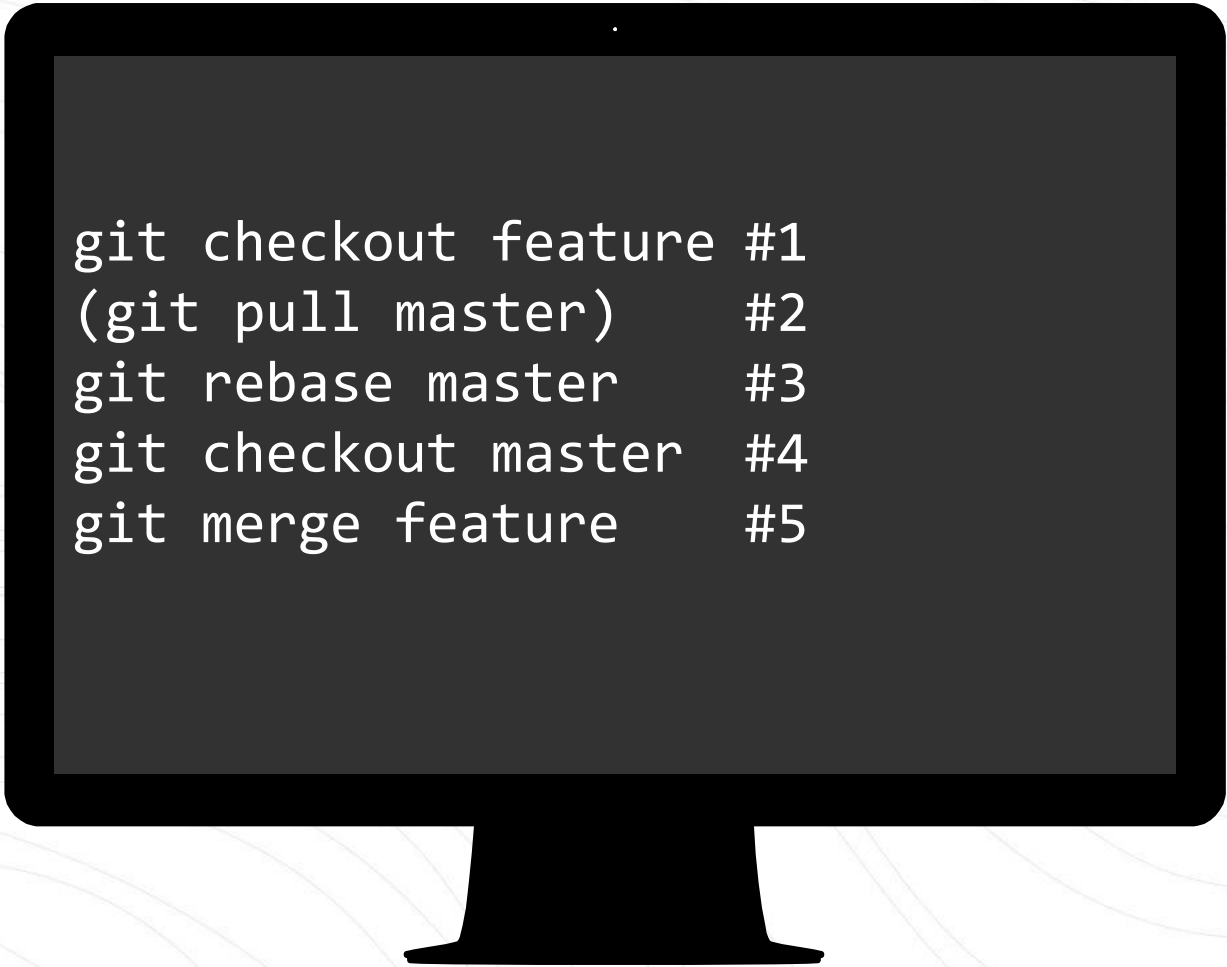
After Rebasing (B1,B2 and B3)





Rebase – krok po kroku

1. Git ustala wspólnego przodka gałęzi **feature** i **master**.
2. GIT przenosi nasze zmiany wprowadzone na gałęzi **feature** do tymczasowego schowka.
3. GIT pobiera zmiany z gałęzi **master** i dodaje je do gałęzi **feature** („ustawia wskaźnik feature na ostatnim commicie z mastera”).
4. Git **KOPIUJE** nasze zmiany z tymczasowego schowka do gałęzi Feature. Kopiowanie odbywa się commit po commicie, na bieżąco rozwiązujemy konflikty. Zmiany lądują „na końcu gałęzi.”
5. Wykonujemy MERGE master i feature. Merge przeprowadzi zwykły **Fast Forward!** Dzięki temu uzyskamy liniową historię i nie narazimy się na nadpisanie identyfikatorów commitów w gałęzi master



```
git checkout feature #1
(git pull master)    #2
git rebase master     #3
git checkout master   #4
git merge feature     #5
```

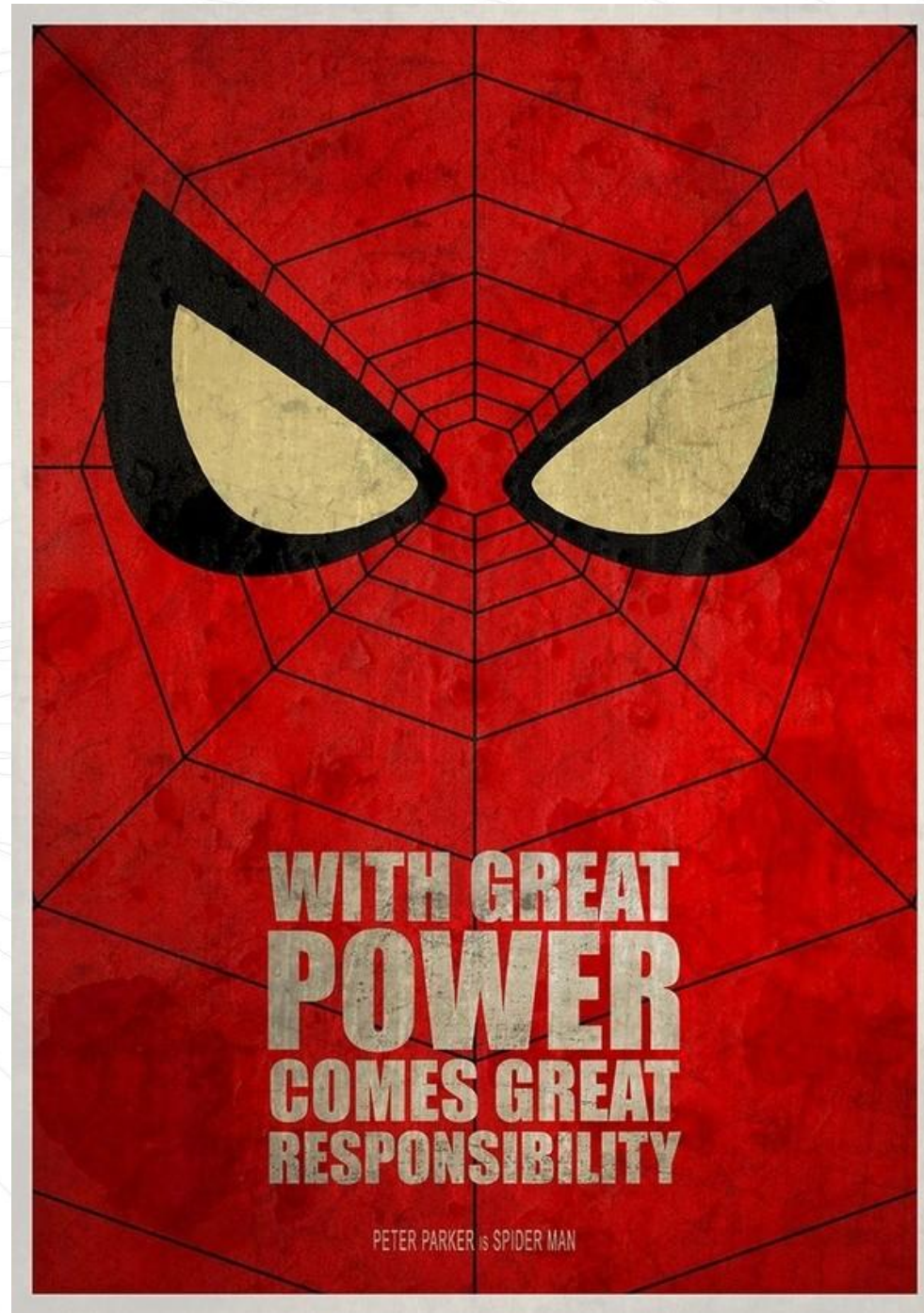



Głównym celem rebase jest utrzymanie liniowej historii zmian.

**Nie wykonuj rebase na głównej gałęzi,
ani na gałęzi na której pracują inni!**

**Chyba, że masz pewność, że nikt jeszcze nie
pobrał zmian.**

GIT kopiuje commity i nadaje im nowe hashe, jeśli zmienisz bazę gałęzi master inni programiści będą mieć **KONFLIKTY** (przez duże K), gdyż ich lokalne repozytorium zawiera inne identyfikatory commitów. Bazę gałęzi, których używasz tylko Ty możesz zmieniać bezkarnie, ponieważ nie ma ryzyka wystąpienia konfliktów.

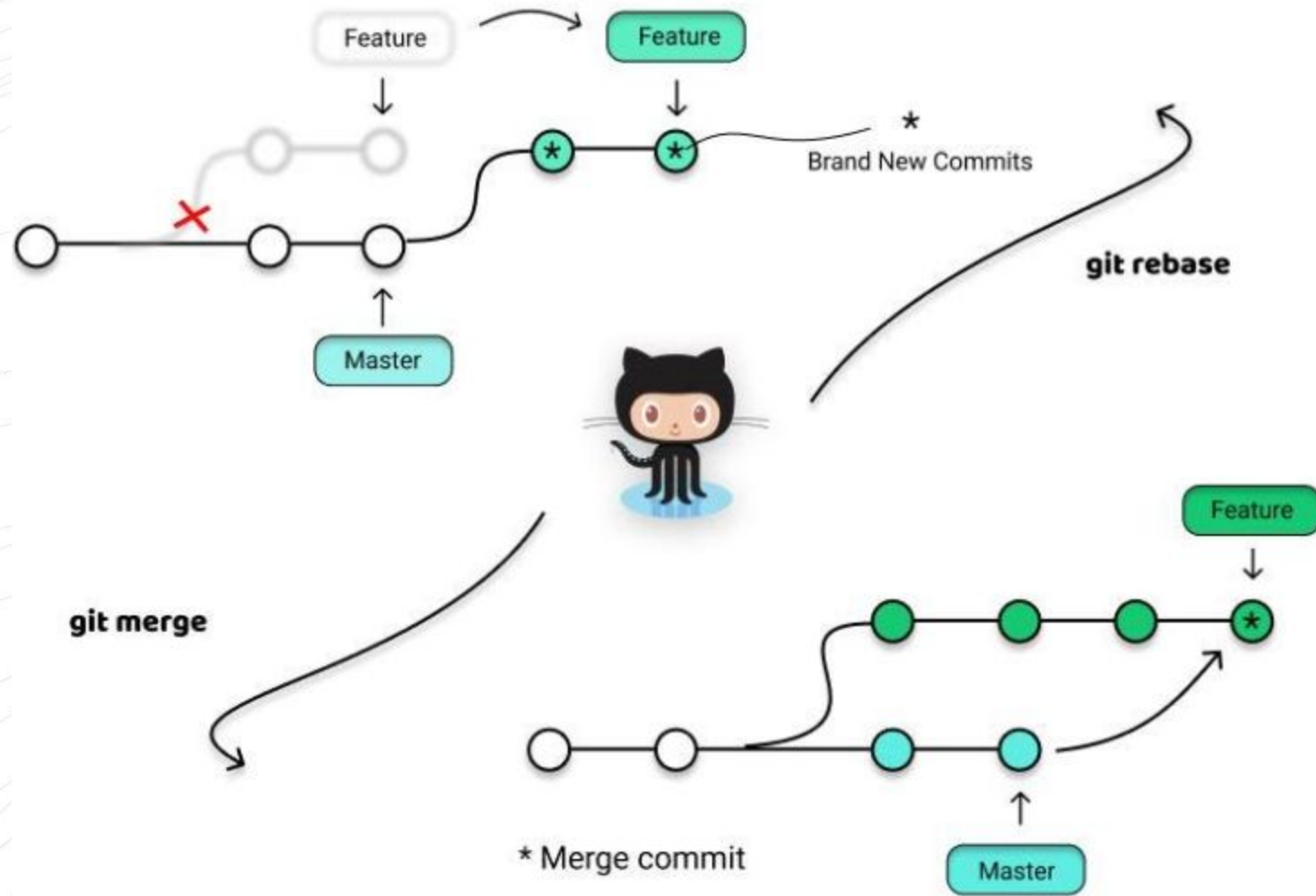


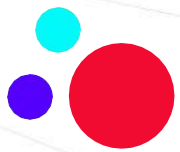
Rebase Quest

1. Sprawdź na jakim branchu jesteś
2. Przenieś się na branch **main**
3. Zrób zmiany w pliku **author.py** lub **title.py**
4. Stwórz branch feature z brancha main
5. Przenieś się na branch **feature**
6. Dodaj zmiany w pliku **story.py**
7. Zrób kilka **commitów**
8. Zrób rebase **brancha** main do **feature**
9. Zmerguj branch **feature** do **main**
10. Wyślij na slacku jaką metodą został zmergowany
11. Sprawdź historię **commitów**



Merge vs Rebase





GIT Rebase – linki

<https://www.atlassian.com/pl/git/tutorials/rewriting-history/git-rebase>

<https://www.youtube.com/watch?v=zOnwgxiC0OA>

<https://www.arturnet.pl/index.php/2021/10/15/git-zaawansowane-funkcje-cz-3-rebase/>





Konflikty

Wyobraźmy sobie sytuację, że wraz z koleżanką lub kolegą pracujecie nad jednym projektem.

Wychodzicie z brancha **main**, każdy z Was robi swój branch feature.

Dodajecie kod w tym samym miejscu. **Commitujecie**.

Teraz chcecie zmergować Wasze branche **feature**, do brancha **main**.

Co się wydarzy ?



Czym są konflikty ?

Z konfliktami mamy do czynienia gdy na dwóch osobnych gałęziach były wprowadzone w tych samych plikach zmiany a następnie chcemy połączyć tę gałęzie.

Konflikty mogą zaistnieć także gdy na jednej gałęzi plik został usunięty a na drugiej zmodyfikowany.

Konflikty są naturalną częścią pracy w zespole. Te w pracy z Gitem przeważnie są najbardziej brutalne.



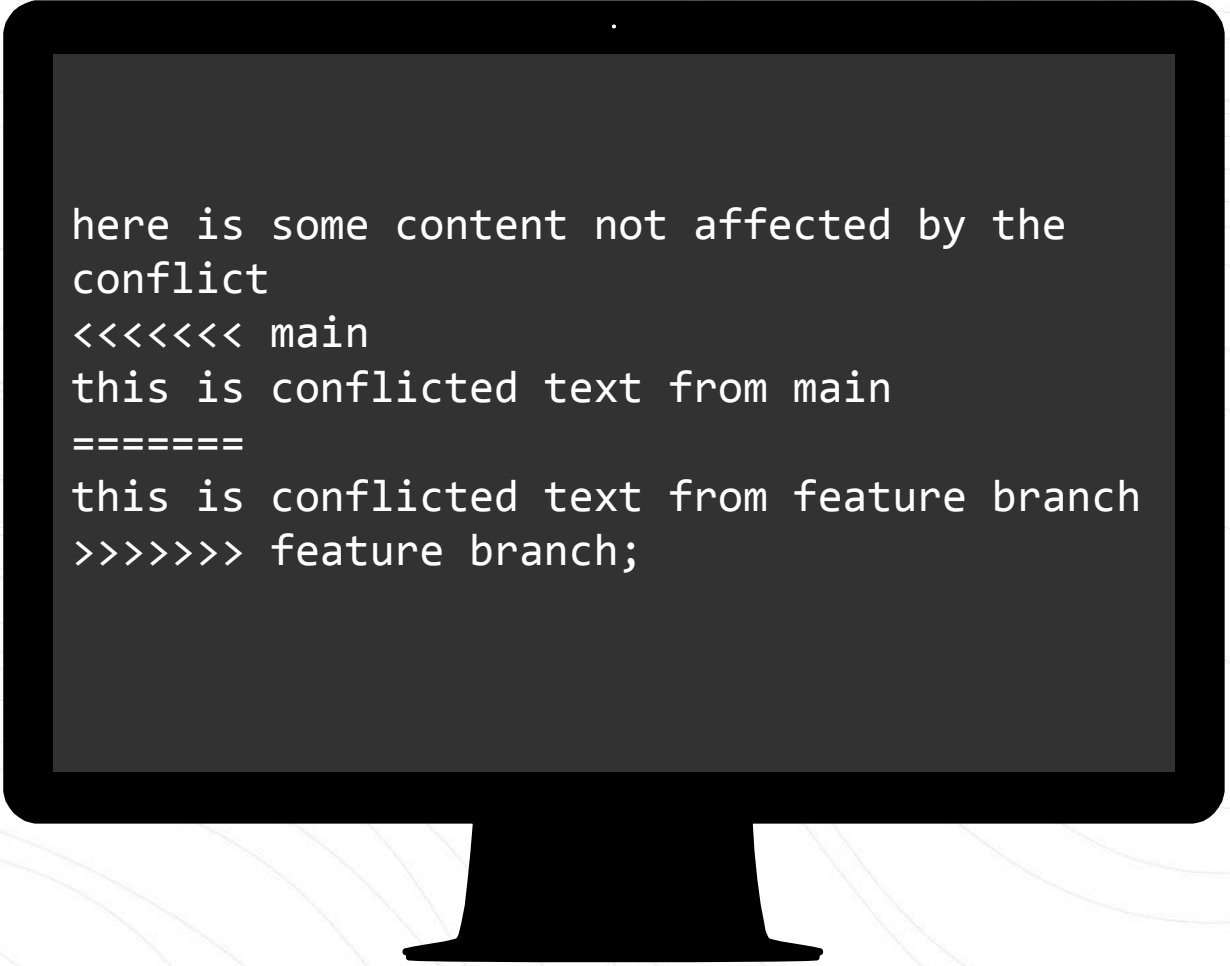
meme-arsenal.ru



Oznaczanie konfliktów

GIT oznacza miejsca w plikach gdzie pojawiły się konflikty używając poniższych znaczników:

- <<<<<< HEAD - określa początek sekcji konfliktu
- ===== - rozdziela kod znajdujący się na branchu docelowym (u góry) od kodu pobranego z brancha źródłowego (na dole)
- >>>>>>
new_branch_to_merge_later - określa koniec sekcji konfliktu oraz nazwę brancha docelowego z którego zmiany zostały pobrane



```
here is some content not affected by the
conflict
<<<<<< main
this is conflicted text from main
=====
this is conflicted text from feature branch
>>>>>> feature branch;
```


Rozwiązywanie konfliktów

Konflikt można rozwiązać poprzez ręczną modyfikację pliku w którym się znajdują.



Mergetool – jest to narzędzie, które w przejrzysty sposób pokazuje różnice w pliku między commitami oraz pozwala w szybki sposób zmodyfikować plik tak aby rozwiązać powstałe konflikty i uzyskać plik w oczekiwanym stanie.

Difftool i Mergetool

Difftool

Narzędzie które pozwala podejrzeć zmiany w plikach pomiędzy commitami. Wywołuje się je poleceniem

git difftool

Domyślnie używanego difftoola (np. na vs code) możemy zmienić używając komendy

git config --global diff.tool vscode

git config --global difftool.vscode.cmd "code --wait --diff \$LOCAL \$REMOTE"

Mergetool

Narzędzie, które pozwala rozwiązać konflikty w plikach powstałe w wyniku łączenia branchy. Wywołuje się je poleceniem.

git mergetool

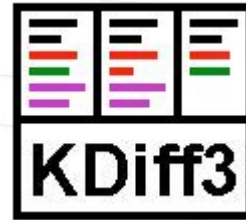
Domyślnie używanego mergetoola (np. na vs code) możemy zmienić używając komendy

git config --global merge.tool vscode

git config --global mergetool.vscode.cmd "code --wait \$MERGED"

Przykładowe difftools

- PyCharm ma wbudowany
- Kdiff3
- Difftastic
- P4Merge
- Visual Studio Code
- WinMerge
- vimdiff
- DiffMerge
- CodeCompare
- i wiele więcej...







Hostingowy serwis internetowy przeznaczony do projektów programistycznych wykorzystujących system kontroli wersji **Git**. Stworzony został przy wykorzystaniu frameworka **Ruby on Rails** i języka **Erlang**. Serwis działa od kwietnia 2008 roku. GitHub udostępnia darmowy hosting programów open source i prywatnych repozytoriów (część funkcji w ramach prywatnych repozytoriów jest płatna). W czerwcu 2018 ogłoszono, iż serwis zostanie przejęty przez przedsiębiorstwo Microsoft za kwotę 7,5 miliarda dolarów.

W maju 2019 roku GitHub informuje, że ma około **37 milionów** użytkowników i więcej niż **100 milionów** repozytoriów (w tym co najmniej 28 milionów repozytoriów publicznych).

Usługa jest jedną z najpopularniejszych tego typu na rynku, z której korzystają takie firmy, jak:

Airbnb, Netflix, Shopify, Udemy, Instacart, LaunchDarkly, Robinhood, reddit.

źródło: <https://pl.wikipedia.org/wiki/GitHub>





Hostingowy serwis internetowy i oprogramowanie przeznaczone dla projektów programistycznych. Gitlab oparty jest o system kontroli wersji Git oraz **otwarteźródłowe (opensource)** oprogramowanie wspomagające zarządzanie projektami opartymi na Git.

Oprogramowanie zostało stworzone przez **Dmitrija Zaporozhets** oraz **Sida Sijbrandij**, którzy założyli później firmę GitLab Inc.

Usługa jest jedną z najpopularniejszych tego typu na rynku, z której korzystają takie firmy, jak: **IBM, Sony, NASA, Oracle, GNOME Foundation, Nvidia, SpaceX.**



źródło: <https://pl.wikipedia.org/wiki/GitLab>



Hostingowy serwis internetowy przeznaczony dla projektów programistycznych wykorzystujących system kontroli wersji **Git** oraz **Mercurial**, którego obecnym właścicielem jest firma **Atlassian**. Serwis umożliwia bezpłatne wykorzystanie usługi wraz z dodatkowymi płatnymi planami. Jest obecnie jednym z najpopularniejszych tego typu serwisów, z którego korzystają m.in. firmy **Ford**, **PayPal**, czy **Starbucks**. W kwietniu 2019 r. Atlassian ogłosił, że Bitbucket dotarł do **10 milionów** zarejestrowanych użytkowników i ponad **28 milionów** repozytoriów.

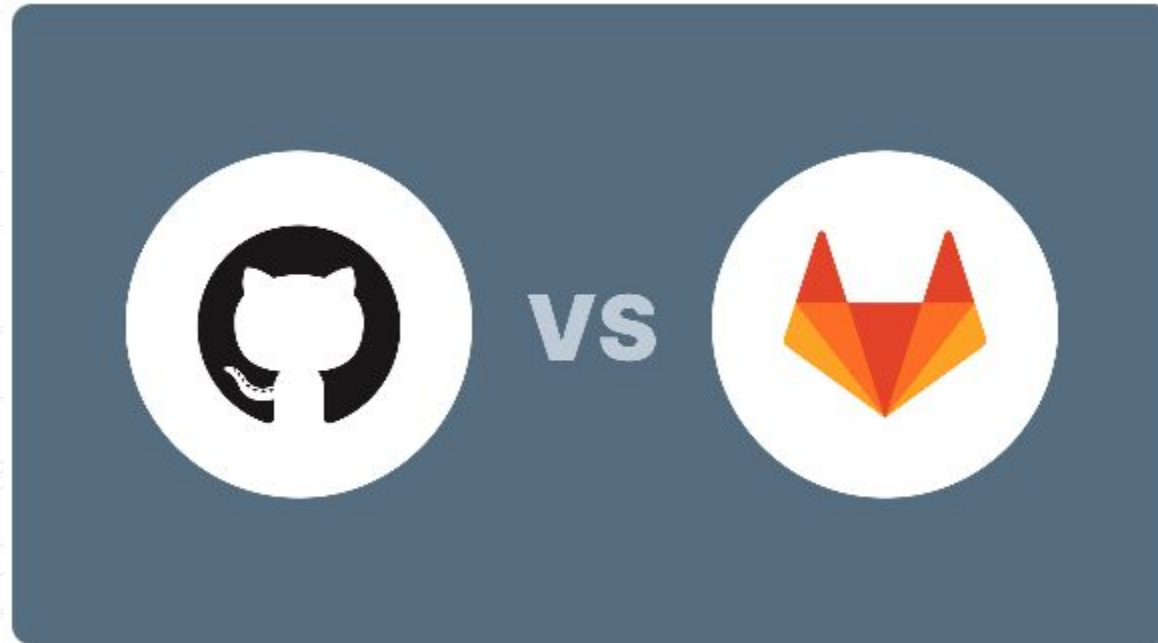
Oprogramowanie serwisu Bitbucket zostało stworzone w Django (platformie programistycznej napisanej w języku Python).



źródło: <https://pl.wikipedia.org/wiki/Bitbucket>





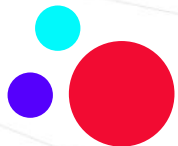
GitHub vs GitLab





GitHub vs GitLab

 Bitbucket	GitHub	 GitLab	So, what does it mean?
Pull Request	Pull Request	Merge Request	In GitLab a request to merge a feature branch into the official master is called a Merge Request.
Snippet	Gist	Snippet	Share snippets of code. Can be public, internal or private.
Repository	Repository	Project	In GitLab a Project is a container including the Git repository, discussions, attachments, project-specific settings, etc.
Teams	Organizations	Groups	In GitLab, you add projects to groups to allow for group-level management. Users can be added to groups and can manage group-wide notifications.



Choose the plan that's right for you.

How often do you want to pay?

Monthly

Yearly ♥ Get 2 months free

Free

The basics for individuals and organizations

\$0 per year
forever

Join for free

- > Unlimited public/private repositories
- > Automatic security and version updates
- > 2,000 CI/CD minutes/month
Free for public repositories
- > 500MB of Packages storage
Free for public repositories
- > New Issues & Projects (in limited beta)
- > Community support

MOST POPULAR

Team

Advanced collaboration for individuals and organizations

\$48 \$40 per user/year
for the first 12 months*

Continue with Team

- ← Everything included in Free, plus...
- > Access to GitHub Codespaces
- > Protected branches
- > Multiple reviewers in pull requests
- > Draft pull requests
- > Code owners
- > Required reviewers

Enterprise

Security, compliance, and flexible deployment

\$252 \$210 per user/year
for the first 12 months*

Start a free trial

Contact Sales

- ← Everything included in Team, plus...
- > Enterprise Managed Users
- > User provisioning through SCIM
- > Enterprise Account to centrally manage multiple organizations
- > Environment protection rules and secrets
- > Audit Log API



Funkcjonalności GitHuba

- repozytoria GIT
- dokumentacja (renderowanie README, Wiki, wsparcie dla Markdown)
- GitHub Actions – wsparcie dla procesów CI\CD – automatyczne budowanie i wdrażanie nowego kodu
- Graphs – statystyki commitów, pull requestów
- WebHooks – integracja zdarzeń z akcjami
- Gists – szybkie wklejanie i hostowanie próbek kodu
- Github Pages – hostowanie statycznych stron www na domenie github.io
- Github Codespaces – edytor kodu wraz ze środowiskiem uruchomieniowym hostowany w chmurze
- Github Copilot – sztuczna inteligencja generująca kod na podstawie sugestii



Inne serwisy hostujące GIT

GitLab

Bitbucket

SourceForge

Google Cloud Source Repositories

Amazon AWS Code Commit

Codebase

Assembla

Launchpad

i wiele więcej...



codebase





Pull Request



W uproszczonym ujęciu **Pull Request** jest mechanizmem powiadamiania członków zespołu przez programistę, że ukończył on jakąś część kodu i chce zmergować zmiany do brancha głównego. Bardzo często, zanim to się stanie jego kod przejdzie tzw. **Code Review**.

Technicznie, gdy branch na którym nowa funkcjonalność będzie gotowy, programista tworzy nowy pull request za pośrednictwem GIT GUI (GitHub, GitLab, BitBucket etc .)





Pull Request

 infoshareacademy / jpydzt4-materialy-podstawy-programowania

infoshareacademy / jpydzt4-materialy-podstawy-programowania

Type to search

[Code](#) [Issues](#) [Pull requests](#) [Actions](#) [Projects](#) [Wiki](#) [Security](#) [Insights](#) [Settings](#)

Filters Search all issues Labels 9 Milestones 0 New pull request

Clear current search query, filters, and sorts

☐ 0 Open ☒ 8 Closed

Author Label Projects Milestones Reviews Assignee Sort

☐ prezentacje day_2 and day_3 added
#8 by kodyliszek was merged 13 minutes ago

☐ Mutable vs immutable
#7 by kodyliszek was merged 4 days ago

☐ instructions
#6 by kodyliszek was merged 5 days ago

☐ operacje na stringach
#5 by kodyliszek was merged 5 days ago

☐ reference_counting.py
#4 by kodyliszek was merged 5 days ago

☐ dodana algorytmy i skrypt
#3 by kodyliszek was merged 2 weeks ago

☐ dodana algorytmy i skrypt
#2 by kodyliszek was merged 2 weeks ago

☐ dodana prezentacja
#1 by kodyliszek was merged 2 weeks ago



Code Review

Code Review polega na tym, że kod, który napiszemy, zanim trafi do głównego brancha, jest przeglądany przez drugiego programistę zwanego w tym procesie **reviewerem**.

Code Review to jedna z technik, która pozwala utrzymać jakość kodu na wysokim poziomie. Jest też formą dzielenia się wiedzą, nie tylko biznesową na temat systemu, nad którym aktualnie pracujemy, ale również dotyczącą programowania jako takiego.





Code Review

reference_counting.py #4

Merged kodyliszek merged 1 commit into `main` from `memory` 5 days ago

Conversation 0 Commits 1 Checks 0 Files changed 1

Changes from all commits File filter Conversations Jump to

39 day_2/zarzadzanie pamiecia/reference_counting.py

```
@@ -0,0 +1,39 @@
1 + import sys
2 +
3 +
4 + class PythonObject():
5 +     pass
6 +
7 + python_object = PythonObject()
8 +
9 + print(f"Liczba referencji obiekt: {sys.getrefcount(python_object)}")
10 +
11 +
12 +
13 + a = 4242
```

Write

Preview

H B I

This is a very good number |

Cancel

Add single comment

Start a review

<https://kobietydokodu.pl/pull-request-i-code-review-czyli-o-empatii-w-programowaniu/>

<https://bulldogjob.pl/readme/jak-stworzyc-idealny-pull-request-w-5-krokach>

<https://bulldogjob.pl/readme/code-review-w-pigulce-czyli-jak-zrobic-to-dobrze>



**Dziękuję za
uwagę!**