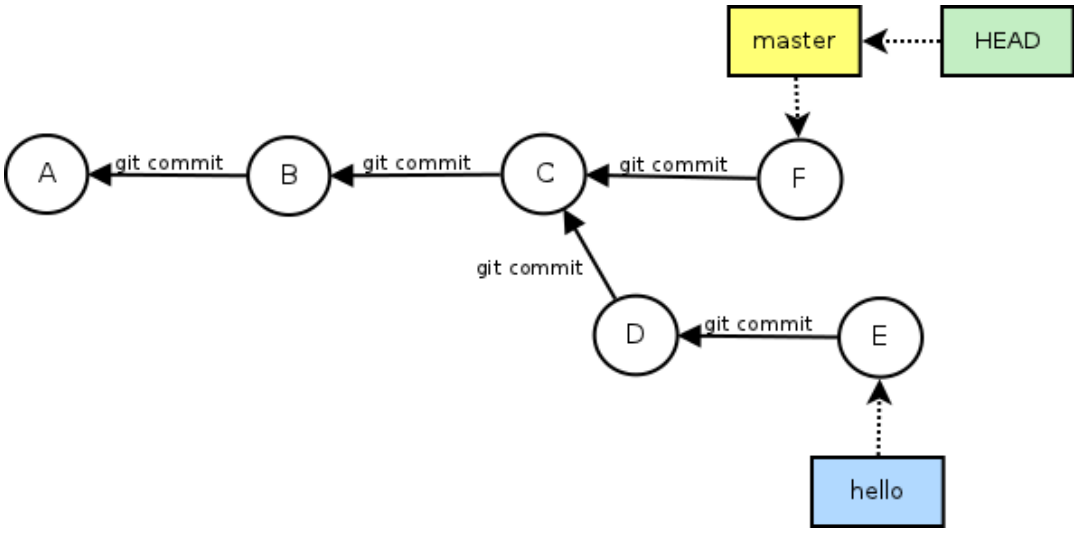
Zarządzanie projektami informatycznymi

Zarządzanie gałęziami, scalanie zmian z różnych gałęzi, przeglądanie historii zatwierdzeń

Wykorzystywanie gałęzi zapewnia niemal równoczesną pracę z różnymi wersjami tego samego projektu. Między wieloma gałęziami możemy się bardzo łatwo i szybko przełączać.



git branch, checkout, log, merge, (tag)

git branch zarządza gałęziami dostępnymi w repozytorium, można: utworzyć gałąź, zmienić jej nazwę, usunąć gałąź bądź wyświetlić ich listę.

git checkout przełącza nas do innej gałęzi, takie jest jego podstawowe zastosowanie.
Bardziej ogólnie można powiedzieć, że **git checkout** odtwarza zawartość:

- całego katalogu roboczego,
- wybranej jego ścieżki,
- konkretnych plików

zawartością zapamiętanego punktu w naszym repozytorium (konkretne zatwierdzenie). Polecenie bardzo często używane, warto sobie zrobić krótszy alias, powszechnie stosowanym jest alias **co**.

Przykłady:

`git branch`
Wyświetla listę gałęzi znajdujących się w repozytorium.

`git branch hello`
Tworzy nową gałąź o nazwie `hello`.

`git branch -d hello`
Usuwa gałąź o nazwie `hello`.

`git branch -m hello witaj`
Zmienia nazwę gałęzi z `hello` na `witaj`.

`git checkout hello`
Przełącza nas do gałęzi `hello`.

`git checkout -b hello`
Tworzy nową gałąź `hello` i od razu **przełącza nas do niej** (taki skrót dwóch pokazanych wyżej).

git log pokazuje historię zatwierdzeń **gałęzi, na której pracujemy**.

Przykłady:

`git log --oneline`
Pokazuje historię zatwierdzeń, przeznaczając na każde z nich jedną linię.

`git log --decorate`
Wyświetla historię zatwierdzeń pokazując dodatkowo: etykiety utworzone poleceniem tag, nazwy gałęzi, etykietę HEAD.

`git log --graph`
Wyświetla historię zatwierdzeń w formie grafu (rysuje linie łączące odpowiednie zatwierdzenia).

`git log --oneline --decorate --graph`
Wszystko razem, bardzo fajne polecenie :) może warto sobie zrobić alias?

git merge scala/łączy podaną gałąź, z tą, na której pracujemy. Mówiąc dokładniej: **git merge** wscala/włącza (ang. *merge in*) podaną gałąź do tej, na której pracujemy (ang. *merge into*).

Przykład:

`git merge inna`
Włącza zmiany wykonane w gałęzi **`inna`** do gałęzi, w której pracujemy.

Polecenia: **`git branch, git checkout i git log`**

Ćwiczenia:

1. Tworzymy sobie puste repozytorium o nazwie `lab02`.
2. Sprawdzamy listę dostępnych gałęzi (**branch**).
3. Tworzymy plik **`main.cpp`**, dodajemy do repozytorium z zatwierdzamy zmiany.
4. Sprawdzamy listę dostępnych gałęzi (**branch**) i historię zatwierdzeń z pokazywaniem etykiet (**log --decorate**).
5. Tworzymy nową gałąź o nazwie `hello` (**branch**), przełączamy się do niej (**checkout**). Można użyć jednego polecenia zamiast dwóch innych (**checkout -b**).
6. Sprawdzamy listę dostępnych gałęzi i historię zatwierdzeń z pokazywaniem etykiet.
7. W pliku **`main.cpp`** piszemy program „Hello World”. Tworzymy plik **`README`** z naszym imieniem i nazwiskiem. Wszystkie zmiany dodajemy i zatwierdzamy.
8. Sprawdzamy historię zatwierdzeń z pokazywaniem etykiet.
9. Przełączamy się na gałąź `master`, sprawdzamy zawartość katalogu roboczego i historię zatwierdzeń z pokazywaniem etykiet.

Polecenia: **`git merge`**, scalanie bez konfliktów

Ćwiczenia:

10. Scalamy zmiany wykonane w gałęzi `hello` do gałęzi `master`.
    Najprostszy typ scalania – **Fast-forward**. Następuje tylko przeniesienie wskaźnika gałęzi.
11. Oglądamy zawartość katalogu roboczego i historię zmian z pokazywaniem etykiet.
12. Usuwamy gałąź `hello` (**branch -d**). Tworzymy nową gałąź `witaj`. Prościej wystarczy zmienić nazwę.
13. Będąc w gałęzi `master` dodajemy w pierwszej linijce pliku **`main.cpp`** informacje o prawach autorskich: „Copyright by ...”. Zmiany zatwierdzamy.
14. Przełączamy się na gałąź `witaj`.
15. Zmieniamy treść wyświetlanego tekstu w pliku **`main.cpp`** na „Witaj WIOSNO :D”. Dodatkowo usuwamy plik README (git rm), bo jest nam już niepotrzebny. Wszystkie zmiany zatwierdzamy (tym razem w gałęzi `witaj`).

16. Oglądamy historię zmian z pokazywaniem etykiet.
17. Zmieniamy gałąź na `master` i znowu oglądamy historię zmian.
18. Scalamy gałęzie.
    Jeśli nie zrobiliśmy błędu powinno się wykonać scalanie trójstronne. Jeśli nie ma zmian w tym samym miejscu tego samego pliku to git sam automatycznie wprowadzi zmiany do plików i połączy odpowiednie gałęzie (Auto-merging main.cpp). Powstanie nowe zatwierdzenie będące połączeniem obu gałęzi – zmiana scalająca (ang. *merge commit*).
19. Sprawdzamy co się usunęło, co się zmieniło, jak wygląda historia. Proponuję użyć dodatkowo opcji **`--graph`**. Ładnie wygląda i pomaga zrozumieć co się z czym połączyło.

Polecenia: **`git merge`**, scalanie z konfliktem

20. Tworzymy sobie nową gałąź dotyczącą dokumentacji (`doc`).
21. W gałęzi `master` w pliku `main.cpp`, nad funkcją `main()` dodajemy linijkę komentarza opisującą działanie programu.
22. Zatwierdzamy zmiany w gałęzi `master`.
23. Przechodzimy do gałęzi `doc`. Nad funkcją main() piszemy dokumentację funkcji dla programu doxygen np.:
    ```
    /// @brief Główna funkcja programu, wyświetla fajny komunikat ...
    ///
    /// @return zawsze zwraca wartość 0.
    ```
24. Zmiany zapisujemy i zatwierdzamy.
25. Przechodzimy z powrotem do gałęzi `master`. Scalamy `master z doc`.
    Mamy konflikt!! Bardzo dobrze, tak miało być :) Trzeba go ręcznie rozwiązać.
    Pomocny może być dłuższy opis poniżej.
26. Polecenie **git status** pokazuje w jakich plikach jest konflikt:
    `UU main.cpp`
27. Ręcznie rozwiązujemy konflikt – wybieramy to co jest właściwe.
28. Poleceniem **git add** oznaczamy, że w pliku rozwiązano konflikt.
    Patrzymy jeszcze raz na status.
29. Zatwierdzamy zmiany z domyślną (bez parametru `-m`) lub naszą własną wiadomością.
30. Oglądamy historię zmian w gałęzi `master`, najlepiej z opcją **`–graph`**.

Rozwiązywanie konfliktu związanego ze scalaniem

Zawartość pliku `main.cpp` z zaznaczoną częścią wymagającą ręcznego rozwiązania konfliktu:

```
Copyright by Robert P.

#include <iostream>

<<<<<<< HEAD
// Cały program, który niewiele co robi ...
=======
/// @brief Główna funkcja programu, wyświetla fajny komunikat ...
///
/// @return zawsze zwraca wartość 0.
>>>>>>> doc
int main() {
    std::cout<<"Witaj wiosno!!! Dobrze, że jesteś :)"<<std::endl;
    return 0;
}
```

Mamy do wyboru dwie wersje kodu źródłowego. Jedna z gałęzi `master` oznaczona znacznikiem HEAD (powyżej `=======`). Druga z gałęzi `doc`. Wybieramy jedną albo tworzymy jakąś inną, np.:

```
/// @brief Cały program, który niewiele co robi ...
/// @return zawsze zwraca wartość 0.
```
Wszystkie znaczniki pomocnicze konfliktu usuwamy.

git tag oznacza (etykietuje) na stałe wybrany punkt w historii zatwierdzeń. Dodaje, usuwa i wyświetla listę utworzonych etykiet.

Przykłady:

`git tag`
Wyświetla listę utworzonych etykiet.

`git tag v1.0.3`
W gałęzi, na której pracujemy, tworzy etykietę `v1.0.3` wskazującą na aktualne zatwierdzenie, to wskazywane przez etykietę HEAD.

`git tag v0.03 de06`
Tworzy etykietę bez komentarza o nazwie `v0.03` do zatwierdzenia rozpoczynającego się od znaków `de06` ... (wystarczy podać cztery znaki).

`git tag v1.24 -a -m "Stabilna wersja działająca na Windows 7"`
Tworzy etykietę z komentarzem o podanej treści.

`git tag -d v1.0.3`
Usuwa etykietę `v1.0.3`, niezależnie na której gałęzi pracujemy.

Ćwiczenia:

1. Utworzyć dwie etykiety:
   Pierwsza etykieta `v0.01` ma być bez komentarza i ma się odnosić do zatwierdzenia będącego scaleniem gałęzi witaj do gałęzi master.
2. Druga etykieta `v0.02` ma zawierać komentarz: "Pierwsza stabilna i udokumentowana wersja" i ma być utworzona w aktualnym punkcie rozwoju gałęzi `master`.

Po powyższych ćwiczeniach polecenie **`git log --oneline --graph --decorate`** wywołane dla gałęzi `master` powinno wyświetlić coś takiego:

```
*   fc2d581 (HEAD, tag: v0.02, master) Merge branch 'doc'
|\
| * 427dc9f (doc) Dokładna dokumentacja dla doxygen'a
* | b30e5c2 Opis działania programu w komentarzu
|/
*   ac737ff (tag: v0.01) Merge branch 'witaj'
|\
| * b893dab (witaj) Witamy po Polsku i wiosennie, usunięty plik README
* | f3b4767 Info o prawach autorskich
|/
* de066f7 Program hello world, plik README
* 3442527 dodany plik main.cpp
```