

Jan Orliński

# System kontroli wersji git

## Usługa GitHub

**Narzędzia Wspierające Programowanie**  
Semestr letni 2024/25



UNIVERSITY  
OF WARSAW

**FACULTY OF  
PHYSICS**

UNIVERSITY  
OF WARSAW

# Co omówimy w tym module?

1. Co to są systemy kontroli wersji i dlaczego stały się tak istotne w rozwijaniu oprogramowania?
2. Dlaczego to akurat git stał się standardem przemysłowym?
3. Jak tworzyć lokalne repozytoria git oraz jak z nich korzystać?

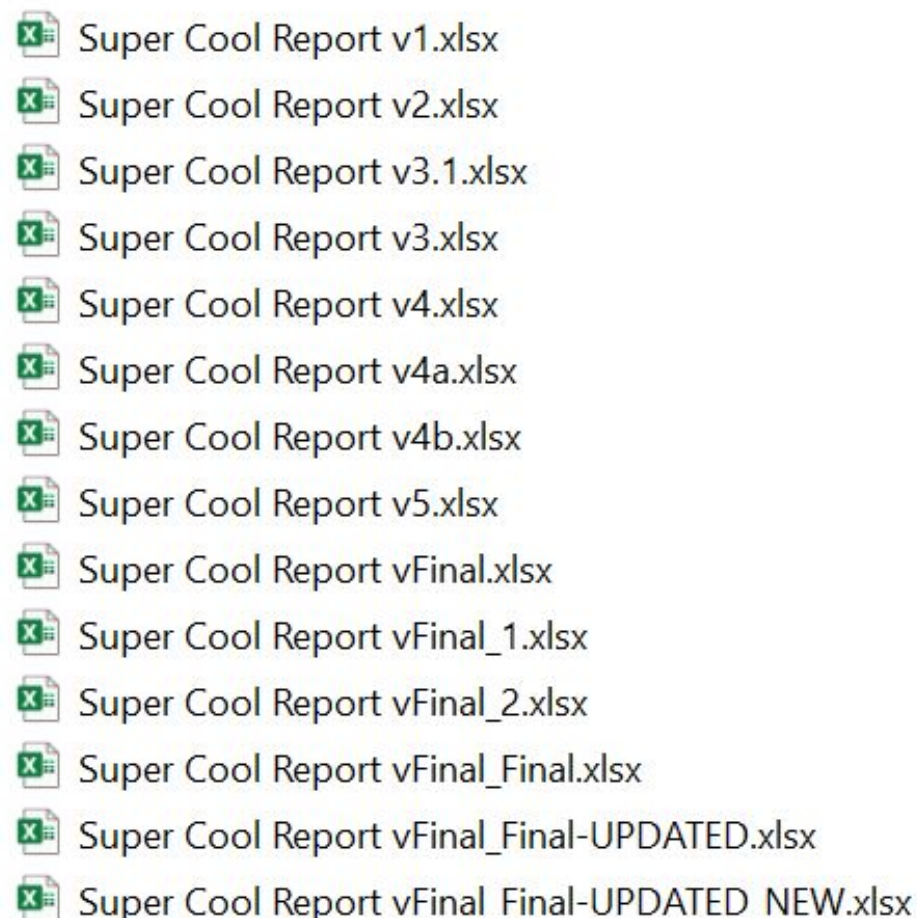
---

4. Jak używać portalu GitHub?
5. Jak połączyć lokalne repozytorium z serwerem?
6. Co może pójść nie tak?

# Systemy kontroli wersji

- Wszyscy (mniej lub bardziej świadomie) wersjonujemy pliki
- Dlaczego to ważne?
  - a. dostęp do wcześniej sprawdzonych rozwiązań ("backup")
  - b. łatwiejsza współpraca (ustalenie wspólnego punktu odniesienia)
  - c. precyzyjne ustalenie wersji obowiązującej (np. zmiany w aktach prawnych)
- Jak to rozwiązać technicznie?

## Name



- Super Cool Report v1.xlsx
- Super Cool Report v2.xlsx
- Super Cool Report v3.1.xlsx
- Super Cool Report v3.xlsx
- Super Cool Report v4.xlsx
- Super Cool Report v4a.xlsx
- Super Cool Report v4b.xlsx
- Super Cool Report v5.xlsx
- Super Cool Report vFinal.xlsx
- Super Cool Report vFinal\_1.xlsx
- Super Cool Report vFinal\_2.xlsx
- Super Cool Report vFinal\_Final.xlsx
- Super Cool Report vFinal\_Final-UPDATED.xlsx
- Super Cool Report vFinal\_Final-UPDATED\_NEW.xlsx

# Proste systemy kontroli wersji

## Pomysł 1: Lokalne wersjonowanie plików (Local Version Control System)

Regularnie tworzymy kopie plików roboczych (wersjowanych i/lub datowanych), ewentualnie śledzimy same zmiany (patche). Historycznie pierwszy i intuicyjnie najprostszy system kontroli wersji.

Przykładowe oprogramowanie: często zbędne, ale do patchy wspierany jest RCS (lata '80).

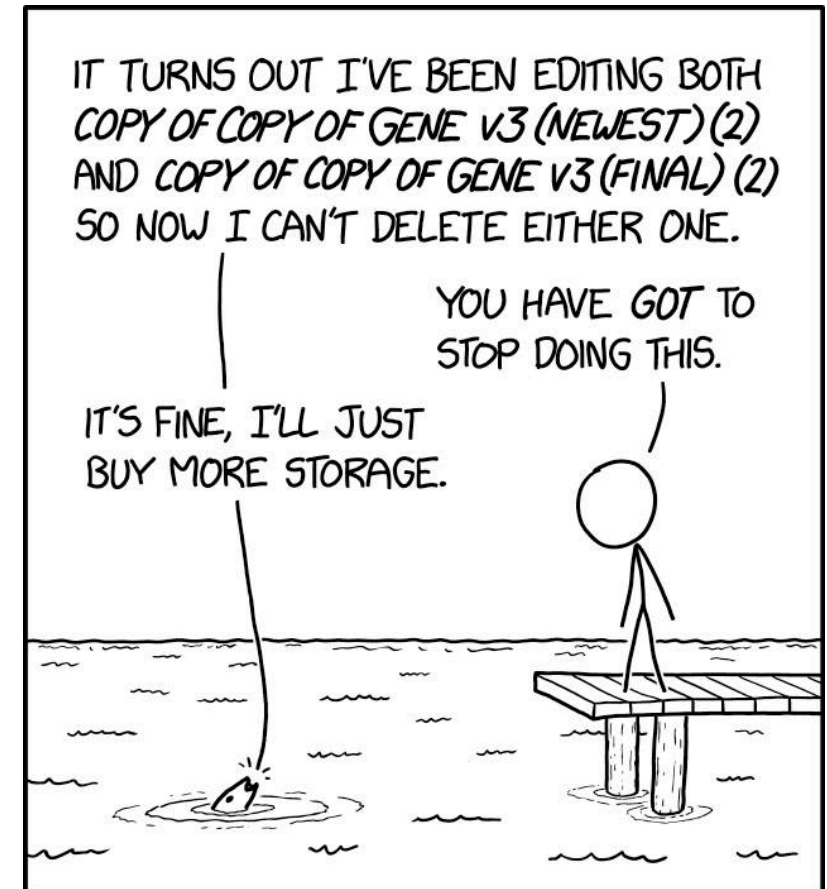
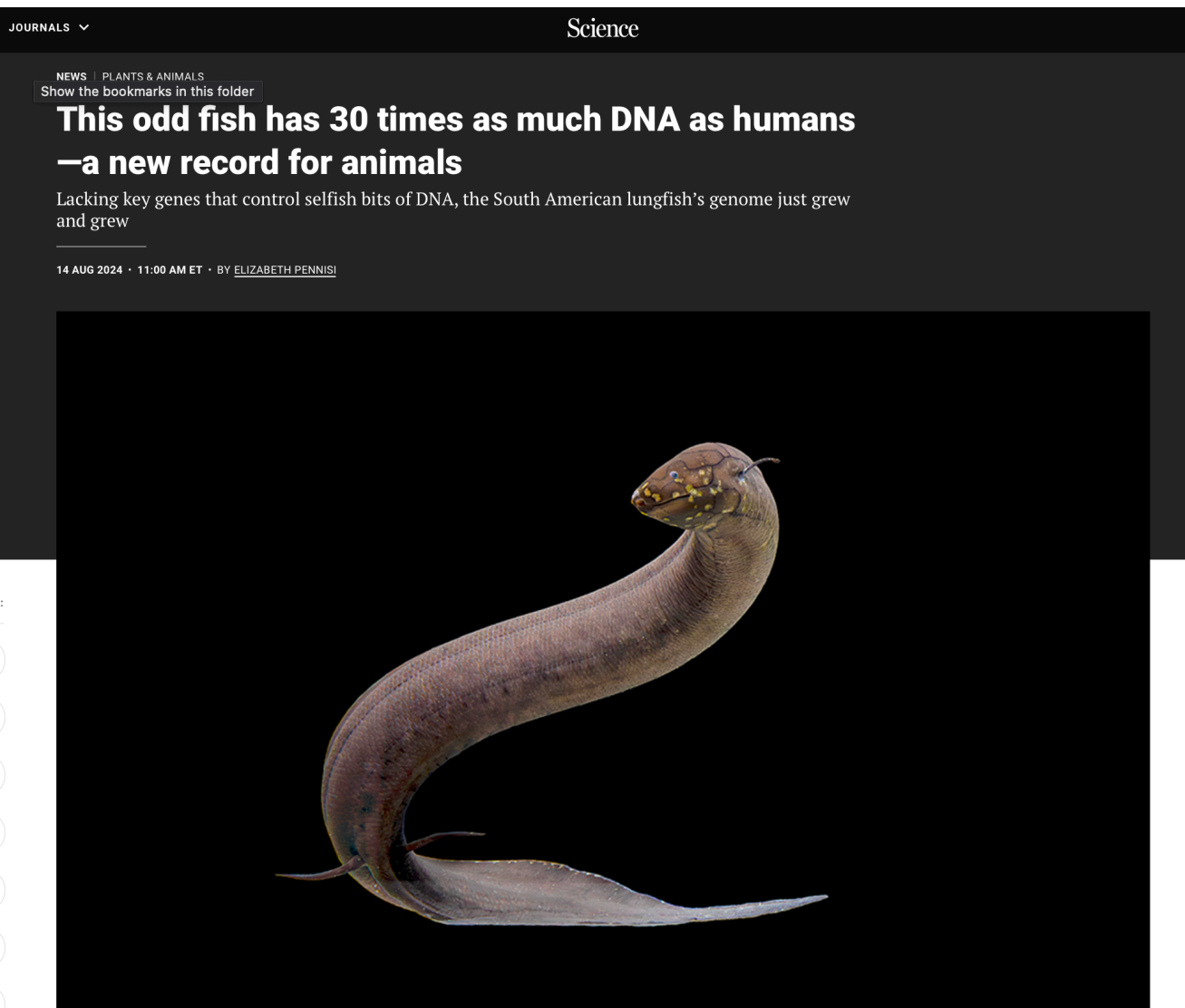
### Zalety:

- prosta implementacja.

### Wady:

- centralizacja przechowywania ("single point of failure"),
- wysoki koszt przestrzeni dyskowej (o ile przechowujemy całe pliki),
- brak możliwości współpracy nad kodem

# Dlaczego to nie wystarcza...



WHY LUNGFISH HAVE SUCH ENORMOUS GENOMES

# Scentralizowane systemy kontroli wersji

Lokalna kontrola wersji jest problematyczna:

- **bardzo podatna na błędy**
- **nie ma możliwości zdalnej współpracy**

Rozwiązaniem tych problemów jest stworzenie scentralizowanego systemu.

**Pomysł 2: Wersjonowanie plików + scentralizowana przestrzeń do przechowywania**

Przykładowa implementacja: na koniec każdego dnia roboczego, tworzymy archiwum przestrzeni roboczej (np. zip) i wgrywamy z odpowiednią datą na ustaloną przestrzeń dyskową (np. w chmurze), którą zarządza koordynator.

**Zalety:**

- **możliwość zdalnej współpracy,**
- **większy rygor (trudniej o pomyłkę)**

**Wady:**

- **centralizacja wersjonowania (konieczny koordynator),**
- **praca zespołowa offline bardzo trudna**

# Niebezpieczeństwo centralizacji





# Rozproszone systemy kontroli wersji

W jaki sposób można obejść dwie największe wady scentralizowanej kontroli wersji?

- **single point of failure**
- **konieczność stałego połączenia z serwerem**

**Odpowiedzią jest rozproszona kontrola wersji** (Distributed Version Control Systems, znane od lat '90 ale spopularyzowane po 2005 gdy Linus Torvalds stworzył Git).

## **Pomysł 3: Wersjonowanie plików + rozproszona przestrzeń do przechowywania**

Przykładowa implementacja: na koniec każdego dnia roboczego, tworzymy archiwum całego projektu (wraz z jego historią) i przekazujemy na zasadzie peer-to-peer. Każdy użytkownik posiada lokalną kopię całego repozytorium.

### **Zalety:**

- **możliwość zdalnej współpracy (nawet bez stałego połączenia sieciowego),**
- **cały projekt jest przechowywany w wielu rozproszonych kopiach**

### **Wady:**

- **większy rozmiar plików do przechowywania i przesyłania w porównaniu do scentralizowanych systemów (ale rzadziej trzeba to robić)**



# I przede wszystkim...



# Ćwiczenie 1

## Tworzenie lokalnego repozytorium, podstawowe komendy

# Jak utworzyć repozytorium

```
> mkdir repo
> cd repo
> touch plik1.txt
> nano plik1.txt
> touch plik2.txt
> nano plik2.txt
> ls -a
. .. plik1.txt plik2.txt

> git init
Initialized empty Git repository in /{...}/repo/.git/

> ls -a
. .. .git plik1.txt plik2.txt

> git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be
  committed)
    plik1.txt
    plik2.txt

nothing added to commit but untracked files present (use
"git add" to track)

> git add plik1.txt plik2.txt
```

Na początek utworzymy nowy, pusty folder i dodajemy do niego parę plików. Na razie nie są one częścią żadnego repozytorium, musimy je dopiero utworzyć.

Będąc w docelowym folderze, tworzymy repozytorium.

Co się zmieniło? Utworzyliśmy repozytorium, którego całość mieści się w ukrytym katalogu "git"

Komendą "status" wyświetlamy stan naszego repozytorium. Mamy w nim dwa pliki, ale na razie są **nieśledzone**. Git nie będzie ich uwzględniał w kolejnych wersjach (commitach).

Komendą "add" dodajemy pliki do śledzenia.

# Śledzenie zmian

```
> git status -s
A plik.txt
A plik2.txt
> git rm --cached *
rm 'plik.txt'
rm 'plik2.txt'
> git status -s
?? plik.txt
?? plik2.txt
> git add -A
> git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   plik.txt
        new file:   plik2.txt

> git commit

*** Please tell me who you are.

Run

  git config --global user.email "you@example.com"
  git config --global user.name "Your Name"

to set your account's default identity.
Omit --global to set the identity only in this repository.

fatal: unable to auto-detect email address (got '{...}')
```

Ponownie wyświetlmy status repozytorium, tym razem w formie skróconej.

Przećwiczmy wyłączanie śledzenia plików.

Często chcemy po prostu dodać do śledzenia całe repozytorium. Robimy to opcją -A.

Mamy już repozytorium; Git wie, które pliki ma śledzić. Jak zapisać zmiany (w tym przypadku "utworzenie" plików, skoro pojawiają się po raz pierwszy).

Zapiszmy je w repozytorium.

Commit jest momentem, w którym potwierdzamy dodanie konkretnej wersji do repozytorium. Musimy się pod tymi zmianami podpisać...

# Pierwszy commit

```
> git commit
2 files changed, 2 insertions(+)
create mode 100644 plik.txt
create mode 100644 plik2.txt

> git status
On branch master
nothing to commit, working tree clean

> git log
commit 03cc5a49e715fbdc7d44730d43f5b51f357fe178 (HEAD -> master)
Author: jan kowalski <mail@email.com>
Date: Mon Apr 7 15:12:12 2025 +0200

    Pierwszy commit

> nano plik.txt

> git commit -a
1 file changed, 1 insertion(+), 1 deletion(-)
```

Po uzupełnieniu naszych danych osobowych możemy zrobić pierwszy commit w repozytorium.

Status pokazuje teraz, że wszystkie wprowadzone zmiany są zarejestrowane ("working tree clean")

Wszystkie commity możemy sprawdzić przez "log" Zobaczemy hash SHA-1, autora i datę zmian oraz commit message.

Wprowadźmy zmiany w jednym z plików.

Możemy ręcznie kreować commity metodą "add", ale na ogół chcemy po prostu zarejestrować wszystkie zmiany w dotychczas śledzonych plikach. Służy do tego opcja "-a" w commicie.

# Sprawdzanie historii repozytorium

```
> git log  
commit 2c73445f887b69bf903a2b102001e3a4c42af00c (HEAD -> master)  
Author: jan kowalski <mail@email.com>  
Date:   Fri Apr 11 09:54:52 2025 +0200
```

Dodano nowa wartosc

```
commit 03cc5a49e715fbdc7d44730d43f5b51f357fe178  
Author: jan kowalski <mail@email.com>  
Date:   Mon Apr 7 15:12:12 2025 +0200
```

Pierwszy commit

Do sprawdzenia historii repozytorium służy komenda "log".

Możemy chcieć na różne sposoby wizualizować tę historię!

# Sprawdzanie historii repozytorium c.d.

```
> git log -p
commit 2c73445f887b69bf903a2b102001e3a4c42af00c (HEAD -> master)
Author: jan kowalski <mail@email.com>
Date:   Fri Apr 11 09:54:52 2025 +0200

    Dodano nowa wartosc

diff --git a/plik.txt b/plik.txt
index 7ab1351..a1f014f 100644
--- a/plik.txt
+++ b/plik.txt
@@ -1,1 @@
-raz dwa trzy
+raz dwa trzy cztery

commit 03cc5a49e715fbdc7d44730d43f5b51f357fe178
Author: jan kowalski <mail@email.com>
Date:   Mon Apr 7 15:12:12 2025 +0200

    Pierwszy commit

diff --git a/plik.txt b/plik.txt
new file mode 100644
index 0000000..7ab1351

> git log --pretty=oneline
2c73445f887b69bf903a2b102001e3a4c42af00c (HEAD -> master) Dodano nowa wartosc
03cc5a49e715fbdc7d44730d43f5b51f357fe178 Pierwszy commit

> git log --pretty=format:"%h - %an, %ar : %s"
2c73445 - jan kowalski, 3 days ago : Dodano nowa wartosc
03cc5a4 - jan kowalski, 7 days ago : Pierwszy commit
```

Do sprawdzenia historii repozytorium służy komenda "log".

Możemy chcieć na różne sposoby wizualizować tę historię!

Np. opcja "-p" (lub "-patch") pokaże nam zmiany wprowadzone w każdym commicie.

Opcję "pretty" możemy określać wiele innych metod wypisu, jak np. oneline, short, full, fuller, itd.

Możemy nawet sami zaprojektować własny sposób zapisu loga, poprzez podanie argumentu "format" dla opcji "pretty".



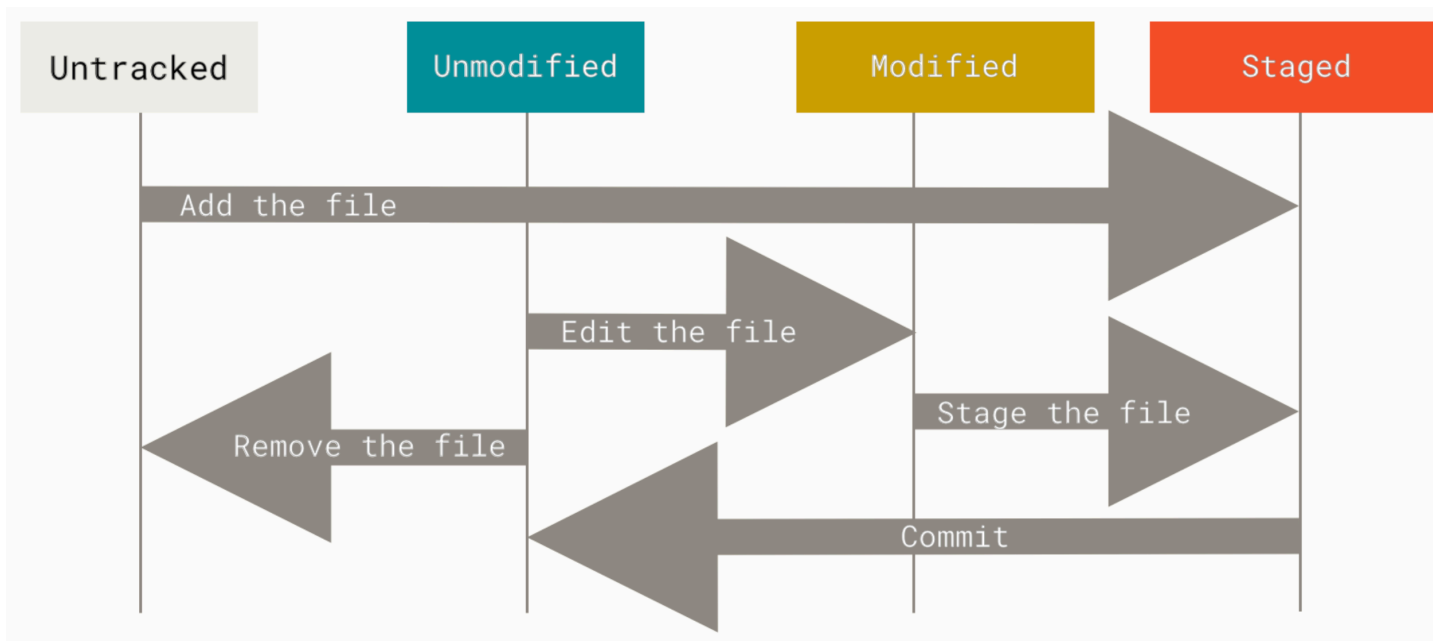
# Opcje formatu:

## Specifier Description of Output

%H	Commit hash
%h	Abbreviated commit hash
%T	Tree hash
%t	Abbreviated tree hash
%P	Parent hashes
%p	Abbreviated parent hashes
%an	Author name
%ae	Author email
%ad	Author date (format respects the --date=option)
%ar	Author date, relative
%cn	Committer name
%ce	Committer email
%cd	Committer date
%cr	Committer date, relative
%s	Subject

# Podsumowanie komend

Komenda	Użyteczne opcje	Opis działania
git init		Rozpoczyna repozytorium w bieżącym katalogu
git add	-A (równoważne git add *)	Dodaje pliki do śledzenia
git rm	-cache (usuwa śledzenie zmian, zostawia plik)	Usuwa pliki
git commit	-a (automatycznie dodaje śledzone pliki)	Zapisuje zmiany w repozytorium
git status	-a (skrótowy zapis)	Ostatnie zmiany i ich śledzenie
git log	-patch, --pretty=oneline, --pretty=format " "	Wypisuje historię repozytorium



## Ćwiczenie 2

# Klonowanie zdalnego repozytorium

# Jak pobrać repozytorium

```
> git clone https://github.com/schacon/ticgit
Cloning into 'ticgit'...
remote: Reusing existing pack: 1857, done.
remote: Total 1857 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (1857/1857), 374.35 KiB | 268.00 KiB/s, done.
Resolving deltas: 100% (772/772), done.
Checking connectivity... done.

> git clone https://github.com/akalinow/Narzedzia_Wspierajace_Programowanie
Cloning into „Narzedzia_Wspierajace_Programowanie”...
remote: Enumerating objects: 326, done.
remote: Counting objects: 100% (326/326), done.
remote: Compressing objects: 100% (235/235), done.
remote: Total 326 (delta 118), reused 243 (delta 73), pack-reused 0 (from 0)
Resolving objects: 100% (326/326), 2.74 MiB | 5.08 MiB/s, done.
Resolving deltas: 100% (118/118), done.

> cd Narzedzia_Wspierajace_Programowanie
> git log --pretty=oneline

4aa075322598d4d51897781c237bb7c3797257e3 (HEAD -> main, origin/main, origin/HEAD) CMake -
przykłady.
1d4eab3e61bf27a02974c37a9a89fba53ff9368d complex.C typo correction
58c228d4169fcd07e0fa80b410a09cf5b8322191 Pliki obiektowe - typo correction
7c110d43d3b5545c809269a30566fda850bd90aa 03-04 minor corrections
ae87fc0486be7b22fdada208c27b8853ffbfbd1c Bash_2 minor corrections
476ee3deb52a49a4909e779eefcaa61938178e0b Bash_2 minor styling corrections
1eb57a9b2a37658aa9982a9735bb543eb9a541e7 Merge branch 'main' of https://github.com/akalinow/
Narzedzia_Wspierajace_Programowanie into main
d66e8a44b605f9137427080b2b47c0e17219d6d2 Podział materiałów na dwa zajęcia
d54067e5bcd4aab2c4c44c3cc12ce304c8e7ce1b Merge branch 'main' of https://github.com/akalinow/
Narzedzia_Wspierajace_Programowanie into main
991a86ca29bb9048caadf62425e4246412d83fc8 02_Python first commit
210e71d6f8f7b6a4f8bcfb8590a413aa510df691 Bash_1 correcting typo
```

Komendą "clone" możemy w łatwy sposób pobrać na nasz dysk jakieś repozytorium umieszczone na serwerze (to może, ale nie musi, być GitHub).

Możemy też pobrać np. repozytorium naszego przedmiotu.

Co ważne: pobieramy **całe** repozytorium, wraz z jego historią tworzenia.

# GitHub

Przypomnijmy sobie: git jest **rozproszonym** systemem kontroli wersji.

Aby w pełni wykorzystać jego możliwości, musimy zacząć myśleć w kategoriach **rozproszonych**.

Najpopularniejszą metodą współpracy na repozytoriach gita jest portal GitHub. Zapewnia on:

- przestrzeń dyskową na przechowywanie repozytorium
- graficzny (i intuicyjny!) interfejs do obsługi repozytoriów
- płynną integrację z lokalnymi implementacjami gita

# Ćwiczenie 3

## Jak odtworzyć czynności z zadania 1. w serwisie GitHub?

## Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

*Required fields are marked with an asterisk (\*).*

Owner \*



Repository name \*

/ verbose-octo-disco

✓ verbose-octo-disco is available.

Great repository names are short and memorable. Need inspiration? How about **verbose-octo-disco** ?

Description (optional)



Public

Anyone on the internet can see this repository. You choose who can commit.



Private

You choose who can see and commit to this repository.



Repozytorium tworzymy "jednym kliknięciem"

## Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Możemy też zacząć od istniejącego repozytorium i "wstawić" je na GitHuba

Required fields are marked with an asterisk (\*).

Owner \*

 janorlinski ▾

Repository name \*

/ verbose-octo-disco

Repo musi mieć nazwę i właściciela.

✓ verbose-octo-disco is available.

Great repository names are short and memorable. Need inspiration? How about **verbose-octo-disco** ?

Description (optional)



Public

Anyone on the internet can see this repository. You choose who can commit.



Private

You choose who can see and commit to this repository.

**WAŻNE!** Ustawiamy widoczność naszego repozytorium (dostęp typu "read").  
Możliwość commitowania ustawiamy osobno (dostęp typu "write")

Na start mamy reklamę githubowego AI ;) i możliwość ustawienia współpracowników.



### Set up GitHub Copilot

Use GitHub's AI pair programmer to autocomplete suggestions as you code.

Get started with GitHub Copilot





### Add collaborators to this repository

Search for people using their GitHub username or email address.

Invite collaborators

Ale mamy też instrukcję, jak dodać nowe pliki oraz łączyć się z lokalnym środowiskiem pracy.

### Quick setup — if you've done this kind of thing before

 Set up in Desktop or HTTPS SSH  

Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).


### ...or create a new repository on the command line

```
echo "# verbose-octo-disco" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/janorlinski/verbose-octo-disco.git
git push -u origin main
```



### ...or push an existing repository from the command line

```
git remote add origin https://github.com/janorlinski/verbose-octo-disco.git
git branch -M main
git push -u origin main
```



janorlinski / verbose-octo-disco

<> Code

Issues

Pull requests

Actions

Projects

Wiki

Security

main

verbose-octo-disco /

Go to file

Add file

janorlinski

Create plik1.txt

8db12de · now

Name	Last commit message	Last commit date
<div></div> plik1.txt	Create plik1.txt	now

## Commit c1a4701

[Browse files](#)

 **janorlinski** authored now Verified

Update plik1.txt

 [main](#)

1 parent [8db12de](#) commit c1a4701 

 **1 file changed** +1 -0 lines changed

 Search within code



▼ plik1.txt 

+1  ...

... @@ -1,2 @@

1 1 raz dwa trzy

2 + cztery piec

**Comments** 0

 Lock conversation



Comment



Unsubscribe

You're receiving notifications because you're subscribed to this thread.

# Ćwiczenie 4

## Integracja środowiska lokalnego z serwerem (GitHub)

# Jak pobrać repozytorium

```
> git clone  
https://github.com/janorlinski/verbose-octo-disco  
  
> cd verbose-octo-disco  
  
> nano plik1.txt  
  
> git commit -a  
  
> git push
```

Komendą "clone" możemy w łatwy sposób pobrać na nasz dysk dowolne repozytorium umieszczone na GitHubie.

W ten sposób pliki w repozytorium lokalnym są "automatycznie" połączone z remote'em na serwerze.

Pierwsza nowa komenda, która odnosi się konkretnie do pracy z serwerami to "pull", czyli pobranie danych z serwera.

I mamy pierwsze schody...

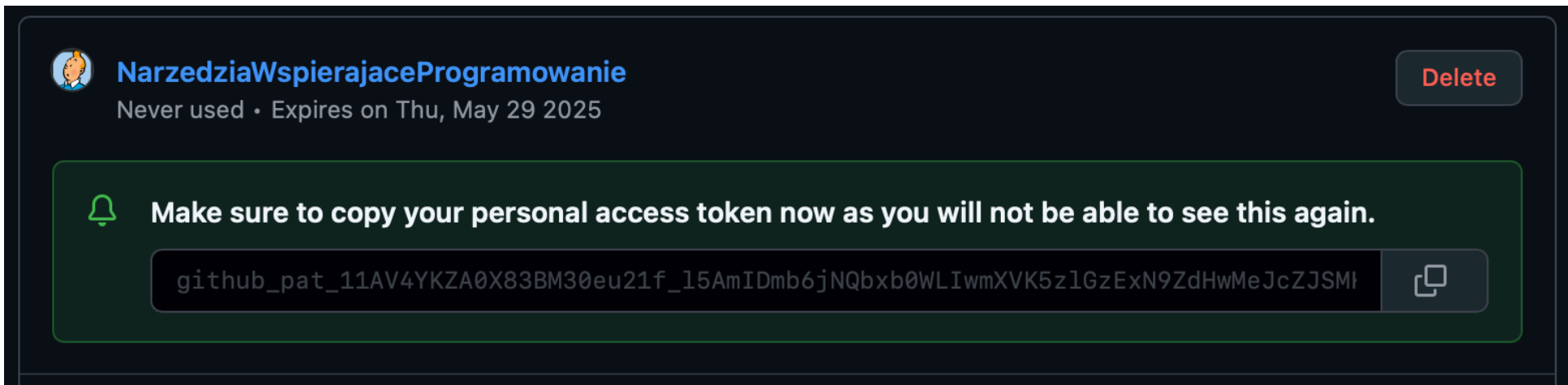


# Tokeny dostępu

1. Settings
2. Developer settings (na samym dole)
3. Personal access tokens
4. Fine-grain tokens
5. Generate new token

... ustawiamy wszystkie szczegóły

## 6. NAJWAŻNIEJSZY MOMENT:

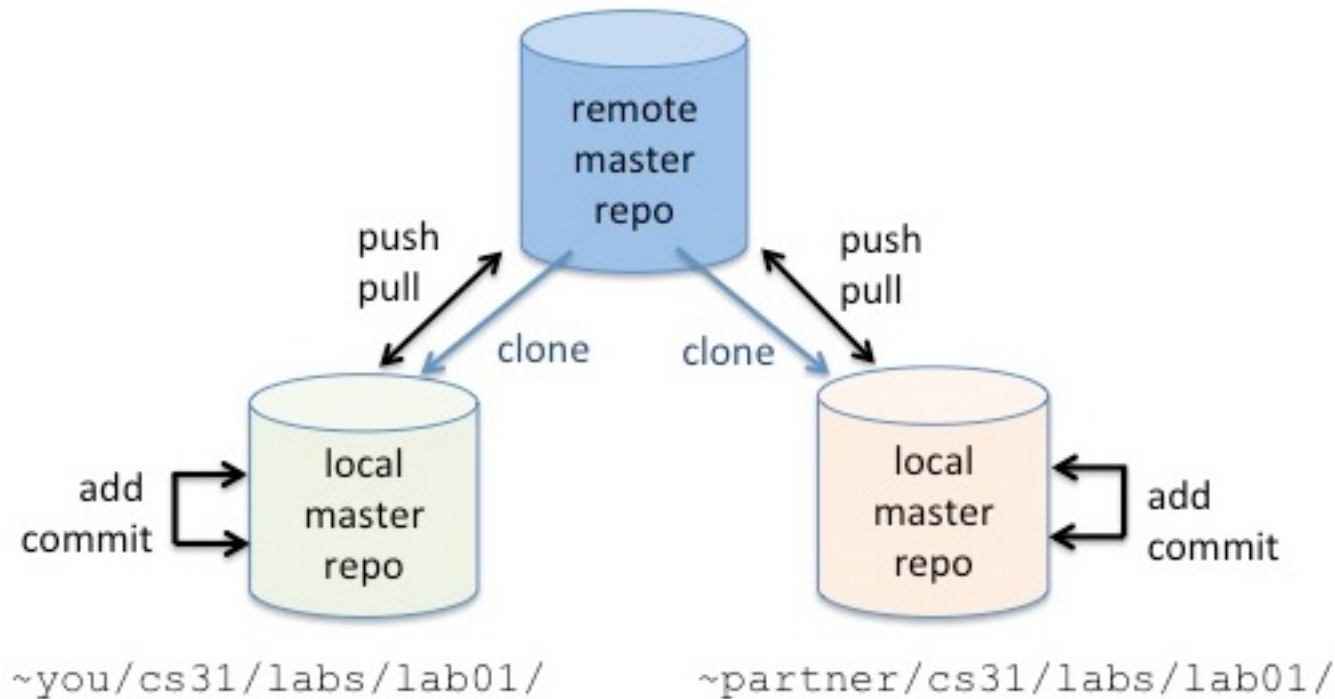


# Jak uaktualnić repozytorium

```
> git pull
```

Komendą "pull" pobieramy najnowszą wersję repozytorium na nasz lokalny dysk **oraz** uaktualniamy nasze lokalne repozytorium do najnowszej wersji.

Jeśli chcemy tylko pobrać dane, ale zostać w naszej wersji, używamy "fetch".



# Ćwiczenie 5

## Konflikty

# Konflikty

W systemach kontroli wersji "konflikt" powstaje, gdy równolegle wprowadzono różne zmiany w tym samym pliku.

W systemach scentralizowanych, gdzie przechowujemy tylko ostatnią wersję pliku, połączenie równoległych ścieżek kończy się utratą postępu.

Ale git przechowuje całą historię repozytoriów -- "zauważy" że repozytoria rozjechały się i nie pozwoli nam nadpisać zmian.

W jaki sposób możemy rozwiązać taki konflikt?

1. Poprzez pull + bezpośrednie ręczne naniesienie poprawek w kodzie
2. Poprzez push -force
3. Poprzez utworzenie nowej gałęzi w historii repozytorium

# Tworzenie nowej gałęzi lokalnie + zdalnie

```
> git branch BRANCH_NAME
```

```
> git checkout BRANCH_NAME
```

```
> git commit -a
```

```
> git push
```

```
> git push -u origin BRANCH_NAME
```

Komendą "git branch" tworzymy nową gałąź na naszym lokalnym repozytorium

Komendą "checkout" przenosimy się na nią

Możemy teraz wprowadzać zmiany na tej gałęzi, nie naruszając stanu gałęzi "main" ("master").

Ale "push" nie zadziała...  
Na zdalnym repozytorium nie istnieje gałąź "BRANCH\_NAME".

Musimy zmusić GitHuba do utworzenia nowej gałęzi o nazwie "BRANCH\_NAME".

# Konsekwencje push --force

**GIT PUSH ORIGIN MASTER --FORCE**



**PUSH REJECTED, REBASE OR  
MERGE**

**GIT PUSH --FORCE**



`git push  
--force`



`git push  
--force-with-lease  
--force-if-includes`



# Przed kolokwium

Zakres części kolokwium dotyczącej gita, jak również przykładowe zadania, zostanie udostępniony niebawem w osobnym dokumencie.