Połączony skrypt

Aleksandra Snopek

15 stycznia 2026

Spis treści

Wprowadzenie **2**

1 Podstawy teoretyczne: System operacyjny **2**
1.1 Czym jest system operacyjny? . . . 2
1.2 Serce systemu, czyli jądro (kernel) . . . 2
1.3 Dwa światy: Linux vs. Windows . . . 2
1.3.1 Porównanie jąder systemowych . . . 2
1.3.2 Zalety i wady obu systemów . . . 3
1.3.3 Przykłady wykorzystania . . . 3
1.4 Świat Linuksa: Dystrybucje . . . 4

2 Podstawowe komendy w systemie Linux **4**
2.1 Powłoka systemowa: Bash . . . 4
2.2 Struktura poleceń: opcje i argumenty . . . 4
2.3 Polecenia informacyjne . . . 5
2.4 Nawigacja i listowanie plików . . . 6
2.5 Zarządzanie plikami i katalogami . . . 7
2.6 Przeglądanie i przetwarzanie plików . . . 8
2.6.1 Wyszukiwanie tekstu: polecenie `grep` . . . 9
2.6.2 Wyszukiwanie plików – Polecenie `find` . . . 10

3 Skrypty Bash – Wprowadzenie do automatyzacji **11**
3.1 Tworzenie i edycja skryptu: edytor `nano` . . . 11

4 System uprawnień plików **12**
4.1 Odczyt uprawnień: Anatomia `ls -l` . . . 12
4.2 Zmiana uprawnień: polecenie `chmod` . . . 13
4.3 Uruchamianie skryptu . . . 15

5 Monitorowanie i zarządzanie procesami **16**
5.1 Stany procesów 16
5.2 Interaktywny monitoring: polecenie `top` 16
5.3 Wyszukiwanie i zabijanie procesów 17
5.3.1 Wyszukiwanie procesów: `pgrep` 17
5.3.2 Zabijanie procesów: sygnały i `kill` 17

6 Instrukcje Warunkowe oraz Pętle **19**
6.1 Instrukcje warunkowe: `if` 19
6.2 Pętle: `for` i `while` 20
6.2.1 Pętla `for` – iteracja po liście elementów 20
6.2.2 Pętla `while` – wykonywanie do spełnienia warunku . . 21

7 Przekazywanie argumentów do skryptu **21**

8 Przetwarzanie Tekstu i Potoki **21**

9 Tablice Asocjacyjne **22**

10 Przetwarzanie Plików: `find` i `while read` **23**
10.1 Problem z "trudnymi"nazwami plików 23
10.2 Rozwiązanie: Połączenie `find` i `while read` 24

11 Sumy Kontrolne **24**

12 Laboratorium: Analiza i Modyfikacja Skryptów **26**
12.1 Skrypt 2: Wyszukiwanie Duplikatów (Wersja Podstawowa) . . 26

13 Wprowadzenie do Kontroli Wersji **29**
13.1 Problem: Chaos w plikach 29
13.2 Rozwiązanie: System Kontroli Wersji (VCS) 29
13.3 Git: Król Systemów Kontroli Wersji 29
13.4 Dlaczego Git to standard w branży? 29

14 Praca z Gitem: Podstawowe Koncepty i Komendy **30**
14.1 Repozytorium: Twój projekt z superpamięcią 30
14.2 Zapisywanie historii: cykl `add` i `commit` 30
14.3 Ignorowanie Plików: plik `.gitignore` 31
14.4 Gałęzie (Branches): Bezpieczne eksperymenty 31

15 Ekosystem Git: GitHub i Dobre Praktyki **32**
15.1 Git vs. GitHub: Narzędzie kontra Platforma 32
15.2 Praca ze Zdalnym Repozytorium (GitHub) 32
15.3 Anatomia wzorowego repozytorium na GitHubie 33
15.4 GUI dla Gita: GitHub Desktop 34
15.5 Uwierzytelnianie: Osobisty Token Dostępu (PAT) 34

Wprowadzenie

Niniejszy dokument stanowi wprowadzenie do podstaw systemu operacyjnego Linux oraz pracy w jego interfejsie wiersza poleceń. Interfejs ten, zwany powłoką, w większości systemów Linux jest realizowany przez program **Bash**. Celem jest przedstawienie fundamentalnych koncepcji teoretycznych, które odróżniają Linuksa od innych systemów, a następnie zapoznanie z podstawowymi poleceniami niezbędnymi do wydajnej pracy w środowisku tekstowym.

1 Podstawy teoretyczne: System operacyjny

1.1 Czym jest system operacyjny?

System Operacyjny (OS)

Jest to nadrzędne oprogramowanie, które zarządza wszystkimi zasobami komputera. Działa jako pośrednik między użytkownikiem a sprzętem (procesorem, pamięcią, dyskami). Umożliwia uruchamianie programów i wykonywanie zadań bez konieczności znajomości technicznych detali działania podzespołów. Najpopularniejsze systemy to Windows, Linux, macOS i Android.

1.2 Serce systemu, czyli jądro (kernel)

Jądro systemu (ang. kernel)

To centralna i najważniejsza część systemu operacyjnego. Odpowiada za fundamentalne zadania: zarządzanie procesami (uruchomionymi programami), alokację pamięci RAM oraz komunikację ze wszystkimi urządzeniami podłączonymi do komputera. Można je postrzegać jako "mózg"operacji systemowych.

1.3 Dwa światy: Linux vs. Windows

Chociaż oba systemy służą do zarządzania komputerem, ich filozofia i budowa znacząco się różnią.

1.3.1 Porównanie jąder systemowych

- **Linux** wykorzystuje jądro **monolityczne**. Oznacza to, że większość kluczowych funkcji jest zintegrowana w jednym, dużym programie. Jądro Linuksa jest otwarte (*open-source*), co oznacza, że każdy może prze-

glądać i modyfikować jego kod. Jest też modularne – sterowniki można dodawać i usuwać w trakcie pracy systemu.

- **Windows** wykorzystuje jądro **hybrydowe** (o nazwie *NT Kernel*). Łączy ono cechy jądra monolitycznego (dla wydajności) z mikrojądrem (dla stabilności). Kod jądra Windows jest zamknięty i stanowi własność firmy Microsoft.

1.3.2 Zalety i wady obu systemów

Windows

Zalety:

- Prostota obsługi i intuicyjność.
- Ogromna kompatybilność z oprogramowaniem i grami.

Wady:

- System jest płatny (licencja).
- Postrzegany jako bardziej podatny na wirusy.

Linux

Zalety:

- Jest darmowy i otwartoźródłowy.
- Wysoka stabilność i bezpieczeństwo.

Wady:

- Wyższy próg wejścia dla początkujących.
- Mniejsza liczba komercyjnych programów i gier.

1.3.3 Przykłady wykorzystania

Windows jest idealnym wyborem dla użytkowników domowych i biurowych, którzy potrzebują systemu działającego "od razu"do przeglądania internetu, pracy z dokumentami czy rozrywki.

Linux jest narzędziem dla osób, które potrzebują pełnej kontroli nad środowiskiem pracy – programistów, administratorów serwerów czy naukowców. Umożliwia precyzyjną konfigurację każdego elementu systemu.

1.4 Świat Linuksa: Dystrybucje

Linux nie jest jednym, konkretnym systemem, lecz jądrem, na bazie którego tworzone są tzw. **dystrybucje**. Są to kompletne systemy operacyjne z jądrem Linux oraz zestawem programów.

- **Debian**: Znany ze swojej stabilności, często używany na serwerach.
- **Ubuntu**: Bardzo popularny, uważany za przyjazny dla początkujących.
- **Linux Mint**: Ceniony za elegancki interfejs, ułatwiający przejście z Windowsa.
- **Hannah Montana Linux**: Przykład, że na bazie Linuksa można stworzyć niemal wszystko.

2 Podstawowe komendy w systemie Linux

2.1 Powłoka systemowa: Bash

Zanim przejdziemy do poleceń, warto zrozumieć, gdzie je wpisujemy. Terminal to program, który emuluje tekstowy interfejs, a wewnątrz niego działa **powłoka systemowa** (ang. *shell*).

Powłoka systemowa (shell)

To program-interpreter, który tłumaczy polecenia wpisywane przez użytkownika na język zrozumiały dla jądra systemu operacyjnego. Jest to podstawowe narzędzie do interakcji z systemem w trybie tekstowym. Najpopularniejszą powłoką w świecie Linuksa jest **Bash** (Bourne-Again SHell).

Bash oferuje wiele ułatwień, takich jak historia wpisywanych poleceń (dostępna strzałkami w górę/dół), autouzupełnianie (klawisz `Tab`) oraz możliwość tworzenia skryptów automatyzujących zadania.

2.2 Struktura poleceń: opcje i argumenty

Praca w terminalu polega na wpisywaniu poleceń według określonego schematu:

```
polecenie [opcje] [argumenty]
```

- **Polecenie** to nazwa programu, który chcemy uruchomić (np. ‘ls‘, ‘cp‘).

- **Opcje** (nazywane też flagami lub przełącznikami) to modyfikatory, które zmieniają domyślne zachowanie polecenia. Zwykle poprzedzone są myślnikiem.
- **Argumenty** to obiekty, na których polecenie ma operować (np. nazwy plików, ścieżki do katalogów).

Opcje występują w dwóch formach:

- **Krótka forma**: jeden myślnik i jedna litera, np. `-l`. Krótkie opcje można łączyć. Zamiast pisać `ls -l -h -a`, można to skrócić do `ls -lha`.
- **Długa forma**: dwa myślniki i pełna nazwa, np. `-list`. Są bardziej czytelne, ale nie można ich łączyć. Pełny odpowiednik `ls -lha` to `ls -list -human-readable -all`.

Instrukcja obsługi: polecenie `man`

Jeśli nie wiesz, jak działa polecenie lub jakich opcji można użyć, skorzystaj z wbudowanej instrukcji (manuala). Wpisz `man <nazwa_polecenia>`, np. `man ls`, aby wyświetlić jego pełną dokumentację. Z manuala wychodzi się, wciskając klawisz `q`.

2.3 Polecenia informacyjne

Poniżej znajdują się podstawowe polecenia służące do uzyskiwania informacji o systemie i użytkowniku.

Polecenie `echo`

Wyświetla tekst (argument) na standardowym wyjściu (ekranie).

```
# Wyświetla podany tekst na ekranie
$ echo 'Hello World'
```

Polecenie `whoami`

Wyświetla nazwę aktualnie zalogowanego użytkownika.

```
# Wyświetla nazwę zalogowanego użytkownika
$ whoami
```

Polecenie `groups`

Pokazuje nazwy grup, do których należy bieżący użytkownik.

```
# Pokazuje grupy, do których należy użytkownik
$ groups
```

Polecenie `date`

Wyświetla aktualną datę i godzinę. Jego format można kontrolować.

```
# Wyświetla bieżącą datę i godzinę
$ date

# Wyświetla datę w formacie ROK-MIESIĄC-DZIEŃ
$ date +"%Y-%m-%d"
```

Polecenie `uname`

Wyświetla informacje o systemie operacyjnym i jego jądrze.

```
# Wyświetla informacje o jądrze systemu (-a to --all)
$ uname -a
```

2.4 Nawigacja i listowanie plików

Te polecenia są kluczowe do poruszania się po systemie plików.

Polecenie `pwd`

Drukuje pełną ścieżkę do bieżącego katalogu roboczego (print working directory).

```
# Sprawdź, gdzie jesteś
$ pwd
```

Polecenie `cd`

Zmienia bieżący katalog (change directory).

```
# Zmień katalog na podany
$ cd /sciezka/do/katalogu

# Przejdź do katalogu domowego użytkownika
```

```
$ cd ~

# Wróć do poprzedniego katalogu
$ cd -
```

Polecenie `ls`

Wyświetla listę plików i katalogów w bieżącej lokalizacji.

```
# Wyświetl zawartość katalogu (list)
$ ls

# Opcje dla ‘ls‘:
# -l: format listy (szczegółowy)
# -h: rozmiary w czytelnej formie (human-readable)
# -a: pokaż wszystkie pliki, także ukryte (zaczynające się od .)
# -t: sortuj według czasu modyfikacji (najnowsze pierwsze)
$ ls -lhat
```

2.5 Zarządzanie plikami i katalogami

Polecenie `touch`

Tworzy nowy, pusty plik lub aktualizuje datę modyfikacji istniejącego pliku.

```
# Utwórz pusty plik
$ touch nowy_plik.txt
```

Polecenie `mkdir`

Tworzy nowy katalog (make directory).

```
# Utwórz nowy katalog
$ mkdir nowy_katalog

# Utwórz całą ścieżkę katalogów (parents)
$ mkdir -p A/B/C
```

Polecenie `cp`

Kopiuje pliki lub katalogi.

```
# Kopiuj plik (cp <źródło> <cel>)
# -v: tryb gadatliwy (verbose), pokazuje co robi
# -i: tryb interaktywny, pyta przed nadpisaniem
$ cp -iv plik.txt /tmp/kopia_pliku.txt

# Kopiuj cały katalog (opcja -r, recursive)
$ cp -r moj_folder/ /tmp/
```

Polecenie `mv`

Przenosi lub zmienia nazwę plików i katalogów (move).

```
# Zmień nazwę pliku (mv <stara> <nowa>)
$ mv plik.txt dokument.txt

# Przenieś plik do innego katalogu z trybem gadatliwym
$ mv -v dokument.txt /tmp/
```

Polecenie `rm`

Usuwa pliki lub katalogi (remove).

```
# Usuń plik
$ rm plik.txt

# Usuń katalog i całą jego zawartość
# -r: rekurencyjnie (dla katalogów)
# -f: wymuszenie (force), nie pyta o potwierdzenie
$ rm -rf stary_katalog/
```

UWAGA!

Polecenie `rm -rf` jest ekstremalnie niebezpieczne. Usuwa wszystko bezpowrotnie i bez pytania o potwierdzenie. Używaj go z najwyższą ostrożnością, zawsze upewniając się, w którym katalogu się znajdujesz (`pwd`).

2.6 Przeglądanie i przetwarzanie plików

Polecenie `cat`

Wyświetla zawartość pliku na standardowym wyjściu.

```
# Wyświetl całą zawartość pliku z numerami linii (-n)
$ cat -n skrypt.sh
```

Polecenie `head`

Wyświetla pierwsze linie pliku (domyślnie 10).

```
# Wyświetl pierwsze 5 linii pliku
$ head -n 5 /var/log/syslog
```

Polecenie `tail`

Wyświetla ostatnie linie pliku (domyślnie 10).

```
# Wyświetl ostatnie 3 linie pliku
$ tail -n 3 /var/log/syslog

# Śledź plik na żywo (idealne do logów, -f to follow)
$ tail -f /var/log/syslog
```

Polecenie `wget`

Pobiera pliki z internetu.

```
# Pobierz plik i zapisz pod inną nazwą (-O)
$ wget -O pan_tadeusz.txt "https://wolnelektury.pl/media/book/txt/pan-tadeusz.tx
```

Przekierowanie wyjścia >

Zapisuje wynik (standardowe wyjście) polecenia do pliku, nadpisując jego zawartość, jeśli plik istnieje.

```
# Zapisz wynik polecenia ‘ls -l‘ do pliku
$ ls -l > lista_plikow.txt
```

2.6.1 Wyszukiwanie tekstu: polecenie `grep`

Służy do wyszukiwania w tekście linii pasujących do zadanego wzorca.

```
# Znajdź linie zawierające "error" w pliku log.txt
$ grep "error" log.txt

# Ignoruj wielkość liter (-i)
$ grep -i "Error" log.txt

# Pokaż numery linii (-n)
$ grep -n "error" log.txt
```

```
# Policz, ile jest pasujących linii (-c)
$ grep -c "error" log.txt

# Pokaż linie, które NIE zawierają wzorca (-v, invert match)
$ grep -v "info" log.txt

# Szukaj rekurencyjnie we wszystkich plikach w katalogu (-r)
$ grep -r "TODO" /sciezka/do/projektu/
```

2.6.2 Wyszukiwanie plików – Polecenie `find`

W pracy z dużymi projektami lub na serwerach często musimy znaleźć pliki o określonej nazwie, typie czy rozmiarze. Polecenie `find` jest do tego idealnym narzędziem.

```
# Znajdź wszystkie pliki z rozszerzeniem .txt w bieżącym katalogu i podkatalogac
$ find . -name "*.txt"

# Znajdź wszystkie katalogi (-type d) o nazwie "Documents" w całym systemie (/)
$ find / -type d -name "Documents"

# Znajdź pliki większe niż 100MB w katalogu domowym (~)
$ find ~ -size +100M
```

3 Skrypty Bash – Wprowadzenie do automatyzacji

Skrypt powłoki (shell script)

Jest to plik tekstowy zawierający sekwencję poleceń, które są wykonywane przez powłokę linia po linii. Skrypty pozwalają zautomatyzować złożone lub powtarzalne zadania, takie jak przygotowywanie danych do analizy, uruchamianie symulacji numerycznych, tworzenie kopii zapasowych czy generowanie cyklicznych raportów.

Zamiast ręcznie wpisywać dziesięciu poleceń, aby pobrać dane, przetworzyć je i wygenerować wykres, możemy zapisać je w skrypcie i uruchomić za pomocą jednej komendy.

3.1 Tworzenie i edycja skryptu: edytor `nano`

Do tworzenia i edycji plików tekstowych w terminalu służą edytory tekstu. Jednym z najprostszych jest `nano`.

```
# Otwórz (lub utwórz) plik o nazwie ‘analiza.sh‘ w edytorze nano
$ nano analiza.sh
```

Po otwarciu edytora, na dole ekranu widoczne są najważniejsze skróty klawiszowe (znak ^ oznacza klawisz `Ctrl`).

- `^O` (Ctrl+O): Zapisz plik (*Write Out*).
- `^X` (Ctrl+X): Wyjdź z edytora (*Exit*).

Wpiszmy w edytorze `nano` treść naszego pierwszego skryptu:

```
#!/bin/bash
# Prosty skrypt analityczny
# 1. Tworzy katalog na wyniki
# 2. Zapisuje w nim log z datą rozpoczęcia
# 3. Symuluje długotrwałe obliczenia

echo "Rozpoczynam analizę..."
mkdir -p wyniki_analizy
date > wyniki_analizy/log.txt
echo "Przeprowadzam symulację..."
sleep 5 # Czeka 5 sekund
echo "Analiza zakończona." >> wyniki_analizy/log.txt
date >> wyniki_analizy/log.txt
```

4 System uprawnień plików

Zanim uruchomimy nasz skrypt, musimy nadać mu prawo do wykonania. W Linuksie każdy plik i katalog ma precyzyjnie określone uprawnienia, które decydują o tym, kto i w jaki sposób może z nim wchodzić w interakcję.

Podstawą systemu uprawnień są trzy fundamentalne prawa, reprezentowane przez litery:

- **r (read)** – Prawo do **odczytu**.
  - Dla pliku: pozwala na przeglądanie jego zawartości (np. poleceniem `cat`).
  - Dla katalogu: pozwala na wylistowanie jego zawartości (np. poleceniem `ls`).
- **w (write)** – Prawo do **zapisu**.
  - Dla pliku: pozwala na modyfikowanie jego zawartości (edycję, nadpisywanie).
  - Dla katalogu: pozwala na tworzenie, usuwanie i zmianę nazw plików wewnątrz tego katalogu.
- **x (execute)** – Prawo do **wykonania**.
  - Dla pliku: pozwala na uruchomienie go jako programu lub skryptu.
  - Dla katalogu: pozwala na "wejście"do niego (np. poleceniem `cd`).

4.1 Odczyt uprawnień: Anatomia `ls -l`

Wynik polecenia `ls -l` zawiera szczegółowe informacje, w tym 10-znakowy ciąg opisujący uprawnienia:

```
$ ls -l analiza.sh
-rw-r--r-- 1 student users 215 paź 26 10:30 analiza.sh
```

Analiza uprawnień: `-rw-r-r-`

Ten 10-znakowy ciąg dzielimy na cztery części:

Znak 1	Znaki 2-4	Znaki 5-7	Znaki 8-10
-	rw-	r-	r-
Typ pliku	Właściciel	Grupa	Inni

- **Typ pliku**: - oznacza zwykły plik, `d` to katalog (directory).
- **Właściciel (user)**: Uprawnienia dla właściciela pliku. Tutaj: odczyt (`r`) i zapis (`w`).
- **Grupa (group)**: Uprawnienia dla grupy. Tutaj: tylko odczyt (`r`).
- **Inni (others)**: Uprawnienia dla pozostałych. Tutaj: tylko odczyt (`r`).
- Myślnik - w miejscu uprawnienia oznacza jego brak.

4.2 Zmiana uprawnień: polecenie `chmod`

Polecenie `chmod` (change mode) pozwala modyfikować te uprawnienia.

```
# Zobaczmy obecne uprawnienia
$ ls -l analiza.sh
-rw-r--r-- 1 student users 215 paź 26 10:30 analiza.sh

# Nadajmy właścicielowi prawo do wykonania.
$ chmod rwx r-x r-x analiza.sh

# Sprawdźmy ponownie. Znak ‘x‘ został dodany, a nazwa pliku zmieniła kolor.
$ ls -l analiza.sh
-rwxr-xr-x 1 student users 215 paź 26 10:32 analiza.sh
```

Uprawnienia jako system ósemkowy

Każde z trzech uprawnień (`r`, `w`, `x`) można przedstawić jako bit w liczbie 3-bitowej. Jest to naturalna konsekwencja potęg dwójki:

- **r** (read) = $2^2 = \mathbf{4}$
- **w** (write) = $2^1 = \mathbf{2}$
- **x** (execute) = $2^0 = \mathbf{1}$

Sumując te wartości, otrzymujemy jedną cyfrę (od 0 do 7) dla każdej kategorii użytkowników (właściciel, grupa, inni). Przykładowo, `chmod 754 plik.txt` oznacza:

- **7** dla właściciela: 4+2+1 → `rwx`
- **5** dla grupy: 4+0+1 → `r-x`

- **4** dla innych: 4+0+0 → `r--`

```
# Zobaczmy obecne uprawnienia
$ ls -l analiza.sh
-rw-r--r-- 1 student users 215 paź 26 10:30 analiza.sh

# Nadajmy właścicielowi prawo do wykonania.
# Chcemy: rwx r-x r-x  ->  (4+2+1)(4+0+1)(4+0+1)  ->  755
$ chmod 755 analiza.sh

# Sprawdźmy ponownie. Znak ‘x‘ został dodany, a nazwa pliku zmieniła kolor.
$ ls -l analiza.sh
-rwxr-xr-x 1 student users 215 paź 26 10:32 analiza.sh
```

Pełna tabela systemu ósemkowego uprawnień (0–7)

Liczba (Octal)	Postać binarna (rwx)	Suma	Znaczenie uprawnień
0	`--`	0+0+0	Brak jakichkolwiek uprawnień. Plik jest całkowicie niedostępny.
1	`--x`	0+0+1	Tylko wykonanie. Umożliwia wejście do katalogu lub uruchomienie pliku binarnego, ale nie pozwala na odczyt jego zawartości.
2	`-w-`	0+2+0	Tylko zapis. Rzadko używane samodzielnie, ponieważ aby zapisać plik, zazwyczaj trzeba go najpierw odczytać.
3	`-wx`	0+2+1	Zapis i wykonanie. Pozwala na modyfikację i uruchomienie pliku.
4	`r--`	4+0+0	Tylko odczyt. Typowe uprawnienie dla plików z danymi, które nie powinny być modyfikowane przez wszystkich.
5	`r-x`	4+0+1	Odczyt i wykonanie. Standardowe uprawnienie dla programów i katalogów, do których potrzebny jest dostęp.
6	`rw-`	4+2+0	Odczyt i zapis. Typowe uprawnienie dla plików, nad którymi pracujemy (np. dokumenty, kod źródłowy).
7	`rwx`	4+2+1	Pełne uprawnienia. Zapewnia pełną kontrolę nad plikiem lub katalogiem.

4.3 Uruchamianie skryptu

Gdy plik ma już prawo do wykonania, możemy go uruchomić, podając jego ścieżkę.

```
# Kropka (.) to skrót oznaczający bieżący katalog
$ ./analiza.sh
# Wynik:
# Rozpoczynam analizę...
```

```
# Przeprowadzam symulację...
# (czekamy 5 sekund)
# Analiza zakończona.
```

5 Monitorowanie i zarządzanie procesami

Proces

To uruchomiona instancja programu. Każdy program, który działa w systemie (nawet sam terminal), ma co najmniej jeden proces. Każdy proces ma unikalny, numeryczny identyfikator (**PID** - Process ID) oraz określony stan.

5.1 Stany procesów

- **Running (R)**: Proces jest aktualnie wykonywany przez procesor lub czeka w kolejce na swoją turę.
- **Sleeping (S)**: Proces czeka na zdarzenie (np. dane z dysku, odpowiedź z sieci). Większość procesów przez większość czasu jest w tym stanie.
- **Stopped (T)**: Proces został zatrzymany (np. przez użytkownika) i może być wznowiony.
- **Zombie (Z)**: Proces zakończył działanie, ale jego wpis w tablicy procesów wciąż istnieje, ponieważ proces nadrzędny nie odczytał jego statusu zakończenia.

5.2 Interaktywny monitoring: polecenie `top`

`top` to fundamentalne narzędzie diagnostyczne. Poza wyświetlaniem listy procesów, pozwala na interaktywne zarządzanie widokiem.

Interaktywne polecenia w `top`

Będąc w `top`, wciśnij klawisz, aby zmienić zachowanie:

- `M` (duże M): Sortuj procesy według użycia pamięci (%MEM).
- `P` (duże P): Sortuj procesy według użycia CPU (%CPU) – domyślne.
- `k`: "Zabij"proces. `top` zapyta o PID procesu do zabicia.

- `h`: Wyświetl pomoc.
- `q`: Wyjdź.

Ciekawostka: `htop`

`htop` to nowocześniejsza, bardziej kolorowa i interaktywna wersja `top`. Oferuje m.in. łatwiejsze przewijanie i zabijanie procesów za pomocą klawiszy funkcyjnych.

5.3 Wyszukiwanie i zabijanie procesów

Ręczne szukanie PID w `top` jest nieefektywne. Lepiej użyć dedykowanych narzędzi.

5.3.1 Wyszukiwanie procesów: `pgrep`

Polecenie `pgrep` (process grep) wyszukuje procesy po nazwie i zwraca ich PID.

```
# Uruchommy w tle proces, który będzie długo działał
$ sleep 600 &
[1] 12345

# Znajdźmy PID procesu o nazwie ‘sleep‘
$ pgrep sleep
12345
```

5.3.2 Zabijanie procesów: sygnały i `kill`

Polecenie `kill` nie "zabija"procesu wprost. Wysyła do niego **sygnał**, czyli komunikat systemowy. Proces może na sygnał zareagować, np. grzecznie się zamykając.

Sygnał	Numer	Opis i zastosowanie
SIGTERM	15	**Terminate (zakończ)**. Domyślny, "uprzejmy"sygnał. Prosi proces o zakończenie pracy, dając mu szansę na zapisanie danych i posprzątanie. To pierwsza próba zamknięcia programu.
SIGKILL	9	**Kill (zabij)**. Sygnał ostateczny, "brutalny". Nie może być zignorowany. Jądro systemu natychmiast usuwa proces z pamięci. Używany, gdy proces się zawiesił i nie reaguje na SIGTERM.
SIGINT	2	**Interrupt (przerwij)**. Sygnał wysyłany po naciśnięciu Ctrl+C w terminalu.

```
# Znajdź PID procesu ‘sleep‘
$ pgrep sleep
12345

# Wyślij domyślny sygnał SIGTERM (prośba o zamknięcie)
$ kill 12345

# Jeśli proces nie reaguje, użyj SIGKILL
# kill -9 <PID> LUB kill -SIGKILL <PID>
$ kill -9 12345
```

Polecenie pkill

pkill łączy w sobie funkcjonalność pgrep i kill. Znajduje procesy po nazwie i od razu wysyła do nich sygnał. Jest bardzo wygodne, ale też bardziej ryzykowne – można przypadkowo zabić wiele procesów naraz.

```
pkill -9 nazwa_procesu
```

6 Instrukcje Warunkowe oraz Pętle

Prawdziwa moc skryptów polega na ich zdolności do **powtarzania** operacji i **podejmowania decyzji**. Te dwie koncepcje – pętle i instrukcje warunkowe – zamieniają prostą listę poleceń w inteligentny program.

6.1 Instrukcje warunkowe: `if`

Instrukcja `if` pozwala skryptowi na wykonanie określonego bloku kodu tylko wtedy, gdy pewien warunek jest prawdziwy. Działa to jak rozwidlenie dróg – skrypt wybiera ścieżkę w zależności od sytuacji.

Anatomia instrukcji `if`

Strukturę można rozbudowywać o kolejne warunki (`elif`) oraz blok domyślny (`else`), który wykona się, gdy żaden z poprzednich warunków nie będzie prawdziwy.

```
if [[ warunek_1 ]]; then
    # Ten kod wykona się, jeśli warunek_1 jest prawdziwy.
    # Bash przejdzie od razu do ‘fi‘.
elif [[ warunek_2 ]]; then
    # Ten kod wykona się, jeśli warunek_1 był fałszywy,
    # ale warunek_2 jest prawdziwy.
else
    # Ten kod wykona się, jeśli wszystkie powyższe warunki
    # okazały się fałszywe.
fi
```

Kluczowa składnia: W Bashu warunki najczęściej umieszcza się w podwójnych nawiasach kwadratowych `[[ ... ]]`. Jest to nowoczesna i bezpieczna forma.

Operatory testów – Twój zestaw narzędzi do sprawdzania warunków

Bash oferuje bogaty zestaw operatorów do budowania warunków. Oto najważniejsze z nich:

- **Testy plików (najczęstsze w skryptach):**
  - `[[ -e $sciezka ]]` – sprawdza, czy plik lub katalog **e**gzistuje (*exists*).

  - `[[ -f $sciezka ]]` – sprawdza, czy to zwykły **f**ile (plik).
  - `[[ -d $sciezka ]]` – sprawdza, czy to **d**irectory (katalog).

- **Testy napisów (zmiennych):**
  - `[[ -z "\$zmienna" ]]` – prawda, jeśli zmienna jest pusta (**z**ero length).
  - `[[ -n "\$zmienna" ]]` – prawda, jeśli zmienna **n**ie jest pusta (*non-empty*).
  - `[[ "\$a" == "\$b" ]]` – prawda, jeśli napisy są identyczne.
- **Testy liczb całkowitych:**
  - `[[ \$a -eq \$b ]]` – równe (*equal*).
  - `[[ $a -ne $b ]]` – nie równe (*not equal*).
  - `[[ $a -gt $b ]]` – większe (*greater than*).
  - `[[ $a -lt $b ]]` – mniejsze (*less than*).

  Operator negacji `!`:
  - `[[ ! -d "\$sciezka" ]]` – prawda, jeśli Sciezka nie jest katalogiem.

6.2 Pętle: `for` i `while`

Pętle służą do wielokrotnego wykonywania tego samego bloku kodu. Bez nich musielibyśmy ręcznie powielać polecenia dla każdego przetwarzanego elementu.

6.2.1 Pętla `for` – iteracja po liście elementów

Pętla `for` jest idealna, gdy mamy z góry znaną listę elementów (np. listę plików w katalogu) i chcemy wykonać jakąś operację dla każdego z nich.

Anatomia pętli `for`

```
for zmienna in element1 element2 element3; do
    # Ten blok kodu wykona się 3 razy.
    # W pierwszej iteracji $zmienna będzie miała wartość "element1",
    # w drugiej "element2", itd.
    echo "Przetwarzam: $zmienna"
done
```

Najczęstszym zastosowaniem jest iteracja po plikach: `for plik in sciezka/*`. Powłoka Bash automatycznie rozwija wzorzec `*` na listę wszystkich plików i katalogów w danej lokalizacji.

6.2.2 Pętla `while` – wykonywanie do spełnienia warunku

Pętla `while` działa inaczej: wykonuje blok kodu tak długo, jak długo jej warunek jest prawdziwy. Jest idealna w sytuacjach, gdy nie wiemy, ile będzie iteracji, np. podczas wczytywania danych z pliku linia po linii.

Anatomia pętli `while`

```
# Przykład: odliczanie od 5 do 1
licznik=5
while [[ $licznik -gt 0 ]]; do
    echo "Odliczanie: $licznik"
    # Ważne: musimy samodzielnie modyfikować warunek,
    # w przeciwnym razie pętla będzie nieskończona!
    licznik=$((licznik - 1))
done
echo "Start!"
```

Pętla `while` sprawdza warunek na początku każdej iteracji. Jeśli warunek od razu jest fałszywy, pętla nie wykona się ani razu.

7 Przekazywanie argumentów do skryptu

Skrypty mogą przyjmować argumenty z wiersza poleceń. Wewnątrz skryptu mamy dostęp do tych argumentów za pomocą specjalnych zmiennych:

- `$0` – nazwa samego skryptu.
- `$1`, `$2`, ... – pierwszy, drugi, itd. argument.
- `$#` – całkowita liczba przekazanych argumentów.

8 Przetwarzanie Tekstu i Potoki

Jedną z najpotężniejszych cech powłoki jest możliwość łączenia prostych narzędzi w złożone sekwencje za pomocą **potoków (pipes)**, reprezentowanych przez znak `|`.

Potok (pipe)

Potok przekierowuje standardowe wyjście jednego polecenia na standardowe wejście następnego. Działa to jak linia montażowa: każde narzędzie wykonuje swoją pracę, a wynik przekazuje dalej.
W skryptach laboratoryjnych używamy potoków do wyodrębnienia kategorii z pliku. Przeanalizujmy tę linię:

```
category=$(grep -i '^CATEGORY:' "$file" | cut -d':' -f2 | tr -d '[:space:]')
```

- `grep -i '^CATEGORY:' "$file"`
  Etap 1: Filtrowanie. Polecenie `grep` odnajduje w pliku interesującą nas linię.
  - `-i`: Ignoruj wielkość liter (*ignore case*).
  - `'^CATEGORY:'`: Wzorzec. Znak `^` to "kotwica", która oznacza "początek linii". Szukamy więc linii, które **zaczynają się** od `CATEGORY:`.
- `cut -d':' -f2`
  Etap 2: Wycinanie. Wynik z `grep` (cała linia) jest przekazywany do `cut`, które wycina z niego tylko potrzebny fragment.
  - `-d':'`: Ustawia separator (delimiter) na dwukropek. Dzieli linię na pola względem `:`.
  - `-f2`: Każe wybrać drugie pole (*field*), czyli wszystko po pierwszym dwukropku.
- `tr -d '[:space:]'`
  Etap 3: Czyszczenie. Fragment tekstu z `cut` jest przekazywany do `tr`, które usuwa z niego niepotrzebne białe znaki.
  - `-d`: Usuń znaki (*delete*).
  - `'[:space:]'`: Specjalna klasa znaków oznaczająca wszystkie białe znaki (spacje, tabulatory, itp.).

9 Tablice Asocjacyjne

Gdy potrzebujemy przechowywać zbiór powiązanych danych (np. mapowanie sum kontrolnych na ścieżki plików), idealnym rozwiązaniem są tablice asocjacyjne.

Tablica Asocjacyjna

To struktura danych przechowująca pary **klucz-wartość**, podobnie do słownika w Pythonie. Deklarujemy ją za pomocą `declare -A nazwa_mapy`.

Sprawdzanie istnienia klucza

Niezawodnym sposobem na sprawdzenie, czy klucz **istnieje** w tablicy, jest składnia: `[[ -n "${mapa[$klucz]+x}"]]`

Tablice asocjacyjne w praktyce: śledzenie duplikatów

Wyobraźmy sobie, jak skrypt do wyszukiwania duplikatów używa tablicy `checksum_map` do śledzenia napotkanych plików:

1. Skrypt analizuje plik `raport.txt`. Oblicza jego sumę kontrolną: `abc...`.
2. Sprawdza, czy klucz `abc...` istnieje w mapie. **Nie istnieje**.
3. Dodaje wpis do mapy: `checksum_map[abc...]` = `"/home/user/raport.txt"`.
4. Skrypt analizuje plik `dane.csv`. Oblicza jego sumę: `def...`.
5. Sprawdza, czy klucz `def...` istnieje w mapie. **Nie istnieje**.
6. Dodaje wpis: `checksum_map[def...]` = `"/home/user/dane.csv"`.
7. Skrypt analizuje plik `raport_kopia.txt`, który jest identyczny jak pierwszy.
8. Oblicza jego sumę kontrolną: `abc...`.
9. Sprawdza, czy klucz `abc...` istnieje w mapie. **Tak, istnieje!**
10. Skrypt wie, że plik `raport_kopia.txt` jest duplikatem, ponieważ jego "odcisk palca"jest już w kartotece. Raportuje znalezisko.

10 Przetwarzanie Plików: `find` i `while read`

10.1 Problem z "trudnymi"nazwami plików

Pętla `for plik in *` zawodzi, gdy nazwy plików zawierają spacje lub inne znaki specjalne.

10.2 Rozwiązanie: Połączenie `find` i `while read`

Standardowym i w 100% niezawodnym sposobem na przetwarzanie listy plików jest połączenie polecenia `find` z pętlą `while read`.

Wzorzec niezawodnego przetwarzania plików

```
while IFS= read -r -d '' nazwa_pliku; do
    # Tutaj bezpiecznie przetwarzamy plik, którego ścieżka jest w "$nazwa_pliku"
    echo "Przetwarzam: $nazwa_pliku"
done < <(find /sciezka/startowa -type f -print0)
```

- `find ... -print0`: Generuje strumień ścieżek oddzielonych znakiem **NULL**.
- `< <(...)`: **Podstawienie procesu** przekierowuje ten strumień na wejście pętli `while`.
- `IFS= read -r -d ''`: Każe poleceniu `read` czytać dane aż do napotkania znaku NULL (`-d ''`), bez interpretowania znaków specjalnych (`-r`) i bez dzielenia według spacji (`IFS=`).

11 Sumy Kontrolne

W skryptach do wyszukiwania duplikatów kluczową rolę odgrywa mechanizm sum kontrolnych. Pozwala on w sposób jednoznaczny i efektywny sprawdzić, czy dwa pliki są identyczne pod względem zawartości, ignorując ich nazwy czy daty modyfikacji.

Suma Kontrolna (Checksum)

Suma kontrolna to **cyfrowy odcisk palca** pliku. Jest to unikalny, krótki ciąg znaków o stałej długości, wygenerowany na podstawie całej zawartości pliku za pomocą algorytmu kryptograficznego.
Właściwości sumy kontrolnej są analogiczne do ludzkiego odcisku palca:

- **Unikalność:** Dwa różne pliki prawie na pewno będą miały zupełnie inne sumy kontrolne.
- **Determinizm:** Ten sam plik zawsze wygeneruje dokładnie tę samą sumę kontrolną.

- **Efekt lawinowy:** Nawet najmniejsza zmiana w pliku (np. zmiana jednej litery) powoduje, że nowa suma kontrolna jest **całkowicie inna**.
- **Jednokierunkowość:** Na podstawie samej sumy kontrolnej nie da się odtworzyć oryginalnego pliku.

`sha256sum` – Cyfrowy Skaner Odcisków Palców

`sha256sum` to standardowe polecenie w Linuksie, które implementuje algorytm **SHA-256** (Secure Hash Algorithm, 256-bit). Jest to "urządzenie", które pobiera cyfrowy odcisk palca. Wynikiem jego działania jest 256-bitowy hash, reprezentowany jako 64 znaki szesnastkowe.

Efekt lawinowy w praktyce

Zobacz, jak dramatycznie zmienia się suma kontrolna po dodaniu jednego znaku:

```
# Używamy ‘echo -n‘, aby uniknąć dodania znaku nowej linii na końcu
$ echo -n "Witaj swiecie" | sha256sum
2db652244951239987819875e3f3ac7c9615a1334237c1a8397a216439e6a037  -

$ echo -n "Witaj swiecie." | sha256sum
266228333f28d81084b423f7e5015e58983e20042472d423982e5d52e5b74108  -
```

12 Laboratorium: Analiza i Modyfikacja Skryptów

Celem laboratorium jest zrozumienie działania, a następnie samodzielne rozbudowanie trzech skryptów automatyzujących pracę z plikami.

Skrypt 1: Automatyczne Sortowanie Plików

Przeznaczenie: Skrypt automatycznie klasyfikuje pliki z katalogu `inbox` do podkatalogów w folderze `classified` na podstawie zawartości każdego pliku.

```
#!/usr/bin/env bash
INCOMING="inbox"; OUTDIR="classified"
mkdir -p "$OUTDIR"
for file in "$INCOMING"/*; do
    [ -e "$file" ] || continue
    category=$(grep -i '^CATEGORY:' "$file" | cut -d':' -f2 | tr -d '[:space:]')
    if [[ -z "$category" ]]; then
        category="unknown"
    fi
    mkdir -p "$OUTDIR/$category"
    cp "$file" "$OUTDIR/$category"/
    echo "Copied $(basename "$file") -> $OUTDIR/$category/"
done
echo "Classification complete."
```

Kluczowe koncepcje: Pętla `for`, potoki (‘|‘), polecenia ‘grep‘, ‘cut‘, ‘tr‘, instrukcja warunkowa `if`.

12.1 Skrypt 2: Wyszukiwanie Duplikatów (Wersja Podstawowa)

Przeznaczenie: Skrypt znajduje duplikaty plików w jednym, wskazanym katalogu. Porównuje pliki na podstawie ich zawartości za pomocą sum kontrolnych `sha256sum`.

```
#!/usr/bin/env bash
if [[ $# -ne 1 ]]; then echo "Usage: $0 <directory>"; exit 1; fi
DIR="$1"
if [[ ! -d "$DIR" ]]; then echo "Error: '$DIR' is not a directory."; exit 1; fi
```

```
declare -A checksum_map
for file in "$DIR"/*; do
    [[ -f "$file" ]] || continue
    checksum=$(sha256sum "$file" | cut -d ' ' -f1)
    if [[ -n "${checksum_map[$checksum]+x}" ]]; then
        echo "Duplicate found:"
        echo "  Original: ${checksum_map[$checksum]}"; echo "  Duplicate: $file"
    else
        checksum_map[$checksum]="$file"
    fi
done
```

Kluczowe koncepcje: Argumenty skryptu, walidacja danych, tablice asocjacyjne, sumy kontrolne.

Skrypt 3: Wyszukiwanie Duplikatów (Wersja Zaawansowana)

Przeznaczenie: Ulepszona wersja poprzedniego skryptu. Jest rekurencyjna i bezpieczna dla nietypowych nazw plików.

```
#!/usr/bin/env bash
if [[ $# -ne 1 || ! -d "$1" ]]; then echo "Usage: $0 <directory>"; exit 1; fi
DIR="$1"
declare -A checksum_map
while IFS= read -r -d '' file; do
    checksum=$(sha256sum "$file" | cut -d ' ' -f1)
    if [[ -n "${checksum_map[$checksum]+x}" ]]; then
        echo "Duplicate found:"
        echo "  Original: ${checksum_map[$checksum]}"; echo "  Duplicate: $file"
    else
        checksum_map[$checksum]="$file"
    fi
done < <(find "$DIR" -type f -print0)
```

Kluczowe koncepcje: Niezawodne przetwarzanie plików za pomocą `find -print0` i pętli `while read`.

Zadania do wykonania

1. **Modyfikacja Skryptu 1 (Sortowanie):**

- Zmień skrypt tak, aby **przenosił** pliki (`mv`) zamiast je kopiować (`cp`).
- Dodaj obsługę drugiego argumentu, który będzie określał katalog docelowy (zamiast na stałe wpisanego `"classified"`). Jeśli argument nie zostanie podany, skrypt powinien użyć wartości domyślnej.

2. **Modyfikacja Skryptu 3 (Duplikaty):**
   - Dodaj do skryptu nową funkcjonalność: po znalezieniu duplikatu, skrypt powinien zapytać użytkownika, czy chce go usunąć. Użyj polecenia `read -p "Pytanie"zmienna`, aby wczytać odpowiedź. Wewnątrz pętli 'while' dodaj instrukcję warunkową, która sprawdzi odpowiedź użytkownika (np. "t"lub "T").
   - (Zaawansowane) Rozbuduj skrypt tak, aby raportował nie tylko pierwszy duplikat, ale wszystkie. (Wskazówka: wartość w tablicy asocjacyjnej może być listą plików, a nie pojedynczym plikiem).

13 Wprowadzenie do Kontroli Wersji

13.1 Problem: Chaos w plikach

Prawdopodobnie każdy spotkał się z problemem utrzymywania wielu wersji tego samego pliku: `praca_v1.doc`, `praca_v2_poprawiona.doc`, `praca_FINALNA.doc`. Taki sposób pracy jest chaotyczny i prowadzi do błędów.

13.2 Rozwiązanie: System Kontroli Wersji (VCS)

System Kontroli Wersji (VCS)

To oprogramowanie, które śledzi i zarządza zmianami w plikach w czasie. Rejestruje każdą modyfikację w specjalnej bazie danych, co pozwala na przeglądanie historii, cofanie się do poprzednich wersji, eksperymentowanie bez ryzyka i, co najważniejsze, efektywną współpracę wielu osób.

13.3 Git: Król Systemów Kontroli Wersji

Git

Git to najpopularniejszy na świecie, darmowy i otwarty **rozproszony** system kontroli wersji. Słowo "rozproszony"oznacza, że każdy członek zespołu ma na swoim komputerze **pełną kopię całej historii projektu**.

13.4 Dlaczego Git to standard w branży?

Nauka Gita to nie tylko kwestia techniczna – to inwestycja w swoją przyszłość zawodową.

Git jako fundament nowoczesnej pracy zespołowej

W każdej firmie technologicznej praca nad kodem odbywa się w zespole. Git jest językiem, za pomocą którego ten zespół komunikuje się w sprawach technicznych. Jego znajomość pozwala na:

- **Bezpieczną współpracę:** Git pozwala dziesiątkom osób pracować nad tym samym projektem jednocześnie, bez ryzyka nadpisania czyjejś pracy.
- **Zarządzanie złożonością:** Nowoczesne oprogramowanie to miliony linii kodu. Git pozwala na organizację tej złożoności poprzez gałęzie.

- **Utrzymanie porządku i odpowiedzialności:** Każda zmiana w Gicie jest "podpisana"przez autora i opatrzona opisem, co tworzy przejrzystą historię projektu.

14 Praca z Gitem: Podstawowe Koncepty i Komendy

14.1 Repozytorium: Twój projekt z superpamięcią

Repozytorium (Repository, "repo")

To folder z Twoim projektem, który jest śledzony przez Gita. Zawiera on wszystkie pliki projektu oraz ukryty podkatalog o nazwie `.git`, w którym Git przechowuje całą historię zmian.

```
# Wejdź do katalogu ze swoim projektem
$ cd moj-projekt
# Zainicjuj puste repozytorium Gita
$ git init
```

14.2 Zapisywanie historii: cykl `add` i `commit`

Praca w Gicie to powtarzalny cykl: **1. Modyfikuj pliki** → **2. Dodaj do poczekalni** (`git add`) → **3. Zatwierdź** (`git commit`).

- `git status` – **Twoja najważniejsza komenda!** Pokazuje, które pliki zostały zmienione, które są w poczekalni, a które nie są śledzone.
- `git add <plik>` – dodaje zmiany z konkretnego pliku do "poczekalni"(Staging Area).
- `git commit -m "Opis zmian"` – tworzy nowy "punkt zapisu"(commit) z plików w poczekalni.
- `git log` – wyświetla historię wszystkich commitów.

```
# Zobaczmy status repozytorium
$ git status
# Stwórzmy nowy plik
$ echo "Pierwsza linia" > plik.txt
# Dodajmy plik do poczekalni, aby Git zaczął go śledzić
$ git add plik.txt
```

```
# Zatwierdźmy zmiany, tworząc pierwszy commit
$ git commit -m "Dodano plik.txt z pierwszą linią"
# Zobaczmy historię
$ git log
```

14.3 Ignorowanie Plików: plik `.gitignore`

Plik `.gitignore` to prosta lista wzorców, które Git ma ignorować. Jest kluczowy, aby do repozytorium nie trafiały pliki tymczasowe, dane wrażliwe (hasła!) czy pliki systemowe.

Dobra praktyka: Stwórz `.gitignore` na samym początku!

Nie musisz pisać tego pliku od zera. Serwisy takie jak `gitignore.io` pozwalają wygenerować gotowe szablony.

14.4 Gałęzie (Branches): Bezpieczne eksperymenty

Gałąź (Branch)

To **ruchoma etykieta** wskazująca na konkretny commit. Gałęzie pozwalają na tworzenie niezależnych, równoległych linii rozwoju. Zawsze twórz nową gałąź dla nowego zadania!

- `git branch <nazwa>` – tworzy nową gałąź.
- `git switch <nazwa>` – przełącza się na istniejącą gałąź.
- `git switch -c <nazwa>` – tworzy nową gałąź i od razu się na nią przełącza.
- `git merge <nazwa>` – łączy zmiany z gałęzi `<nazwa>` do tej, na której aktualnie jesteś.

```
# Stwórz nową gałąź i przełącz się na nią
$ git switch -c nowa-funkcja
# ... pracuj na plikach, rób commity ...
# Wróć na gałąź główną
$ git switch main
# Połącz zmiany z ‘nowa-funkcja‘ do ‘main‘
$ git merge nowa-funkcja
```

15 Ekosystem Git: GitHub i Dobre Praktyki

15.1 Git vs. GitHub: Narzędzie kontra Platforma

- **Git** to narzędzie działające w wierszu poleceń na Twoim komputerze. To "silnik".
- **GitHub** to platforma internetowa, która służy do **hostowania** repozytoriów Gita. To "chmura"dla Twojego kodu.

15.2 Praca ze Zdalnym Repozytorium (GitHub)

Twoje lokalne repozytorium może być połączone ze zdalną kopią, np. na GitHubie. Ta zdalna kopia, nazywana domyślnie `origin`, jest centralnym punktem dla współpracy.

Wysyłanie zmian na serwer: `git push`

Polecenie `git push` wysyła Twoje lokalne commity (z konkretnej gałęzi) do zdalnego repozytorium. To jak synchronizacja zapisów z chmurą.

```
# Wyślij zmiany z lokalnej gałęzi ‘main‘ do zdalnej ‘origin‘
$ git push origin main
```

Przy pierwszym wysłaniu nowej gałęzi, użyj `git push -u origin nazwa-galezi`. Opcja `-u` tworzy "powiązanie śledzące", dzięki czemu w przyszłości wystarczy wpisać samo `git push`.

Pobieranie zmian z serwera: `git pull`

Polecenie `git pull` robi odwrotną rzecz: pobiera najnowsze zmiany ze zdalnego repozytorium i łączy je z Twoją lokalną wersją. To kluczowa komenda w pracy zespołowej – **zawsze wykonuj ją przed rozpoczęciem pracy**, aby mieć pewność, że pracujesz na aktualnym kodzie.

```
# Pobierz i połącz najnowsze zmiany z gałęzi ‘main‘ na serwerze
$ git pull origin main
```

Nie panikuj! Wstęp do rozwiązywania konfliktów

Czasami, podczas `git pull` lub `git merge`, Git napotka **konflikt**. Dzieje się tak, gdy Ty i inna osoba zmodyfikowaliście **te same linie w tym**

samym pliku. Git nie wie, która wersja jest poprawna, więc zatrzymuje proces i prosi Ciebie o podjęcie decyzji.
W pliku objętym konfliktem zobaczysz specjalne znaczniki:

```
<<<<<<< HEAD
Twoja zmiana (to, co miałeś lokalnie)
=======
Zmiana, która nadeszła z serwera
>>>>>>> nazwa_commita_lub_galezi
```

Twoim zadaniem jest:

1. Otworzyć plik w edytorze.
2. Zdecydować, która wersja kodu ma zostać. Możesz wybrać swoją, cudzą, albo połączyć obie.
3. **Usunąć wszystkie znaczniki** dodane przez Gita («<, ===, »>).
4. Zapisać plik.
5. Dodać rozwiązany plik do poczekalni (`git add <plik>`) i zakończyć scalanie, tworząc nowy commit (`git commit`).

Konflikty są normalną częścią pracy w zespole. Najważniejsze to zrozumieć, dlaczego powstały, i spokojnie je rozwiązać.

15.3 Anatomia wzorowego repozytorium na GitHubie

Dobrze prowadzone repozytorium jest czytelne i łatwe w nawigacji.
Powinno zawierać:

- **Plik `README.md`:** To "strona główna"Twojego projektu z opisem i instrukcjami.
- **Plik `.gitignore`:** Dba o to, by w repozytorium nie znalazły się niepotrzebne pliki.
- **Przejrzysta historia commitów:** Każdy commit jest mały i ma jasny, zrozumiały opis.

15.4 GUI dla Gita: GitHub Desktop

GitHub Desktop

To oficjalna, darmowa aplikacja od GitHuba, która pozwala na wykonywanie większości operacji w Gicie za pomocą kliknięć, a nie komend. Jest świetna na początek.

15.5 Uwierzytelnianie: Osobisty Token Dostępu (PAT)

Zamiast hasła, do uwierzytelniania w terminalu używa się **Osobistego Tokenu Dostępu (PAT)**.

- **Jak go wygenerować?:** Zaloguj się na GitHubie, wejdź w `Settings` → `Developer settings` → `Personal access tokens` i kliknij `Generate new token`.
- **Ważne:** Po wygenerowaniu tokena **skopiuj go i zapisz w bezpiecznym miejscu**. Już nigdy więcej go nie zobaczysz!
- **Jak go użyć?:** Gdy terminal poprosi o hasło podczas `git push`, wklej skopiowany token.

Pełny cykl pracy: od zera do GitHuba

Scenariusz A: Pierwsze wysłanie projektu na GitHuba

1. Stwórz nowe, puste repozytorium na stronie GitHub.com.
2. W swoim lokalnym folderze projektu, zainicjuj repozytorium Gita:
   `git init`
3. Dodaj wszystkie pliki do poczekalni i stwórz pierwszy commit:
   ```
   git add .
   git commit -m "First commit"
   ```
4. Połącz swoje lokalne repozytorium ze zdalnym na GitHubie (URL skopiuj ze strony GitHuba):
   `git remote add origin link_do_twojego_repozytorium`
5. Upewnij się, że główna gałąź nazywa się `main` (dobra praktyka):
   `git branch -M main`

6. Wyślij swoje commity na serwer GitHuba. Opcja `-u` ustawia zdalną gałąź jako domyślną dla przyszłych poleceń `push`:
   `git push -u origin main`

Scenariusz B: Codzienny cykl pracy w zespole

1. **Zsynchronizuj się:** Zawsze zaczynaj od pobrania najnowszych zmian. (`git pull origin main`)
2. **Stwórz nową gałąź:** Utwórz i przełącz się na nową gałąź dla swojego zadania.
   (`git switch -c moja-nowa-funkcja`)
3. **Pracuj:** Wprowadzaj zmiany i regularnie rób małe, dobrze opisane commity.
   (`git add .` → `git commit -m "..."`)
4. **Wyślij gałąź na serwer:** Gdy Twoja praca jest gotowa do oceny.
   (`git push origin moja-nowa-funkcja`)
5. **Stwórz Pull Request:** Na stronie GitHub.com, aby poprosić o włączenie Twoich zmian.
6. **Scal (Merge):** Po akceptacji, Twoje zmiany stają się częścią głównej gałęzi projektu.