

Helion



Włodzimierz Gajda

Rozproszony system
kontroli wersji

Spis treści

Podziękowania	9
Część I Repozytoria o liniowej historii	11
Rozdział 1. Wprowadzenie	13
Git	13
Jak przebiega praca nad projektem stosującym Git?	14
Hosting projektów Git	19
Czego się nauczysz z tego podręcznika?	20
Dokumentacja	20
Rozdział 2. Instalacja programu Git	23
Konsola Gita w systemie Windows	25
Ułatwienia uruchamiania konsoli w systemie Windows	26
Podstawowa konfiguracja klienta Git	27
Edytor	28
Rozdział 3. Tworzenie repozytoriów	29
Inicjalizacja nowego repozytorium	29
Klonowanie repozytoriów	30
Badanie historii projektu	33
Wizualizacja historii projektu	36
Rozdział 4. Obszar roboczy	39
Przywracanie stanu projektu, który zawiera nowe pliki	41
Rozdział 5. Tworzenie rewizji i przywracanie stanu plików	43
Tworzenie rewizji	43
Przywracanie stanu plików do wybranej rewizji	45
Przenoszenie repozytorium	48
Rezygnacja z repozytorium	49
Rozdział 6. Stany plików	51
Uproszczony model pracy: przestrzeń robocza i repozytorium	51
Indeksowanie	52
Diagram stanów	53

Obszar roboczy, indeks i repozytorium	56
Modyfikowanie stanu plików repozytorium	57
Stan repozytorium	61
Uproszczony model pracy raz jeszcze	62
Oznaczenia stanów pliku	68
Stany dwuliterowe (mieszane)	69
Repozytoria zwykłe i surowe	72
Składnia polecen Gita	73
Rozdział 7. Ignorowanie plików	75
Uzupełnienie diagramu stanów	78
Rozdział 8. Znaczniki	83
Znaczniki lekkie i oznaczone	83
Tworzenie znaczników opisanych	84
Tworzenie znaczników lekkich	84
Usuwanie znaczników	85
Sprawdzanie dostępnych znaczników	85
Szczegółowe dane znacznika	85
Użycie znaczników	86
Generowanie skompresowanych plików odpowiadających konkretnej wersji projektu	89
Rozdział 9. Identyfikowanie rewizji	91
Pełne skróty SHA-1	91
Skrócona postać SHA-1	92
Znaczniki	92
Nazwa symboliczna HEAD	93
Rewizja domyślna	93
Repozytoria o historii nielinowej	94
Dziennik reflog	100
Polecenia rev-parse oraz rev-list	101
Znaki specjalne wiersza polecen Windows	102
Rozdział 10. Skróty komend	107
Komendy ułatwiające zapisywanie stanu projektu	108
Komendy ułatwiające wykonywanie ćwiczeń	110
Rozdział 11. Modyfikowanie historii projektu	115
Usuwanie ostatnich rewizji	116
Modyfikowanie ostatniej rewizji	117
Łączenie rewizji	117
Usuwanie zmian wprowadzonych przez rewizję	120
Odzyskiwanie poszczególnych plików z dowolnej rewizji	125
Rozdział 12. Podsumowanie części I	127
Co powinieneś umieć po lekturze pierwszej części?	130
Lista poznanych polecień	130

Część II	Repozytoria z rozgałęzieniami	139
Rozdział 13.	Tworzenie i usuwanie gałęzi	141
	Gałęzie to wskaźniki rewizji!	141
	Gałź master	141
	Tworzenie gałęzi	143
	Dodawanie rewizji w bieżącej gałęzi	143
	Tworzenie gałęzi wskazujących dowolną rewizję	144
	Przełączanie gałęzi	145
	Tworzenie i przełączanie gałęzi	147
	Stan detached HEAD	148
	Relacja zawierania gałęzi	150
	Usuwanie gałęzi	153
	Zmiana nazwy gałęzi	155
	Gałęzie jako identyfikatory rewizji	156
	Uwagi o usuwaniu ostatnich rewizji	157
	Sprawdzanie różnic pomiędzy gałęziami	157
	Gałęzie i dziennik reflog	161
	Zgubione rewizje	163
Rozdział 14.	Łączenie gałęzi: operacja merge	167
	Przewijanie do przodu	168
	Przewijanie do przodu dla wielu gałęzi	169
	Łączenie gałęzi rozłącznych	170
	Łączenie kilku rozłącznych gałęzi	171
	Wycofywanie operacji git merge	173
Rozdział 15.	Łączenie gałęzi: operacja rebase	175
	Podobieństwa i różnice pomiędzy poleceniami merge i rebase	176
	Wycofywanie operacji git rebase	178
Rozdział 16.	Podsumowanie części II	181
	Co powinieneś umieć po lekturze drugiej części?	181
	Lista poznanych poleceń	182
Część III	Gałęzie zdalne	185
Rozdział 17.	Definiowanie powiązania między repozytorium lokalnym a zdalnym	187
	Klonowanie raz jeszcze	187
	Klonowanie repozytorium z dysku	191
	Definiowanie repozytoriów zdalnych	192
	Definiowanie powiązania między gałęzią lokalną a gałęzią śledzoną	193
	Listowanie gałęzi	194
Rozdział 18.	Podstawy synchronizacji repozytoriów	195
	Pobieranie gałęzi z repozytorium zdalnego do repozytorium lokalnego	195
	Uaktualnianie sklonowanych repozytoriów	197
	Repozytoria surowe	198
	Przesyłanie gałęzi do repozytorium zdalnego	199

Wysyłanie dowolnej gałęzi	206
Przełączanie na gałąź zdalną	208
Przesyłanie gałęzi ze zmianą nazwy	208
Usuwanie gałęzi zdalnych	209
Zabezpieczanie przed utratą rewizji	209
Polecenie backup	210
Przesyłanie gałęzi do repozytorium zwykłego	210
Rozdział 19. Praktyczne wykorzystanie Gita — scenariusz pierwszy	215
Inicjalizacja projektu	216
Dołączanie do projektu	216
Wprowadzanie zmian w projekcie	217
Wykorzystywanie kilku gałęzi	218
Rozdział 20. Łączenie oddzielnych repozytoriów	219
Graf niespójny	223
Rozdział 21. Podsumowanie części III	225
Co powinieneś umieć po lekturze trzeciej części?	226
Lista poznanych poleceń	226
Część IV Treść pliku	231
Rozdział 22. Konflikty	233
Konflikt tekstowy: wynik operacji git merge	233
Konflikt tekstowy: wynik operacji git rebase	236
Dublowanie konfliktów przez operacje merge i rebase	238
Konflikty binarne	238
Konflikt binarny: wynik operacji git merge	239
Konflikt binarny: wynik operacji git rebase	240
Przywracanie plików do postaci z łączonych gałęzi	242
Polecenia checkout i show	242
Rozdział 23. Badanie różnic	245
Szukanie zmienionych wyrazów	253
Szukanie zmienionych plików	254
Wyszukiwanie rewizji, w których podany plik został zmieniony	255
Rozdział 24. Pliki tekstowe i binarne	257
Odróżnianie plików binarnych od tekstowych	257
Atrybut diff — konflikty tekstowe i binarne	258
Konwersja znaków końca wiersza	259
Projekty wieloplatformowe	260
Ustalenie konwersji znaków końca wiersza dla konkretnych plików	261
Rozdział 25. Podsumowanie części IV	263
Co powinieneś umieć po lekturze czwartej części?	263
Lista poznanych poleceń	264

Część V Praca w sieci	267
Rozdział 26. Serwisy github.com i bitbucket.org	269
Rozdział 27. Klucze SSH	277
Instalacja oprogramowania SSH w systemie Windows	277
Konfiguracja klucza SSH na serwerze github.com	279
Konfiguracja klucza SSH na serwerze bitbucket.org	280
Repozytorium zdalne na serwerze SSH	280
Rozdział 28. Tworzenie i usuwanie repozytoriów w serwisach	
github.com i bitbucket.org	283
Inicjalizowanie nowego repozytorium: serwis github.com	283
Import istniejącego kodu: serwis github.com	286
Inicjalizowanie nowego repozytorium: serwis bitbucket.org	287
Rozdział 29. Praktyczne wykorzystanie Gita — scenariusz drugi	291
Scenariusz pierwszy realizowany w serwisach github.com i bitbucket.org	292
Rozdział 30. Praca grupowa w serwisach github.com oraz bitbucket.org	293
Praca oparta na żądaniach aktualizacji	294
Praca grupowa wykorzystująca żądania aktualizacji (bez gałęzi) w pigułce	301
Żądania aktualizacji i gałęzie	303
Opisy i dyskusje	313
Rozdział 31. Zintegrowany system śledzenia błędów	315
Rozdział 32. Podsumowanie części V	319
Repozytoria do ćwiczenia znajomości Gita	319
Dodatki	321
Dodatek A Literatura	321
Dodatek B Słownik terminów angielskich	323
Skorowidz	325

Podziękowania

Ta książka nie powstałyby, gdyby nie praca, wiedza i życzliwość wielu osób.

Bardzo serdecznie dziękuję:

- ◆ autorom oprogramowania Git;
- ◆ wszystkim uczestnikom projektu Git;
- ◆ właścicielom serwisu *github.com* za wyrażenie zgody na wykorzystanie ilustracji prezentujących interfejs serwisu;
- ◆ właścicielom serwisu *bitbucket.org* za wyrażenie zgody na wykorzystanie ilustracji prezentujących interfejs serwisu oraz za umożliwienie mi pracy w serwisie *bitbucket.org* na licencji akademickiej;
- ◆ Scottowi Chaconowi za udostępnienie wspaniałego podręcznika na temat Gita;
- ◆ twórcom i uczestnikom projektu Symfony 2 za pomoc, którą mi okazali podczas rozpoczętania mojej przygody na githubie;
- ◆ pracownikom Wydawnictwa Helion, szczególnie Pani Redaktor Eweliny Burskiej, za opiekę i profesjonalizm;
- ◆ studentom Katolickiego Uniwersytetu Lubelskiego im. Jana Pawła II, którzy w latach 2011 – 2012 uczestniczyli w prowadzonych przeze mnie zajęciach realizowanych z wykorzystaniem Gita;
- ◆ uczestnikom organizowanych przeze mnie szkoleń za opinie na temat materiału, który posłużył do opracowania niniejszego podręcznika;
- ◆ moim najbliższym za wsparcie i mobilizację.

Włodzimierz Gajda

Lublin, 31 sierpnia 2012 r.



Część I

Repozytoria

o liniowej historii

Rozdział 1.

Wprowadzenie

Współczesne oprogramowanie komputerowe jest w znacznej większości produkowane przez zespoły składające się z wielu osób. Szczególnym przykładem są rozwiązania open source, których kod jest dostępny publicznie. Każdy może do takiego projektu dołączyć i wprowadzać własne modyfikacje. Jeśli projekt składa się z dużej liczby plików i uczestniczy w nim wiele osób, to synchronizacja wersji kodu przy użyciu zwykłych operacji plikowych (np. kopiowania) staje się niewykonalna. Z tego powodu powstały narzędzia określane wspólnym terminem **systemów kontroli wersji**¹ (ang. *version control systems*).

Głównym zadaniem systemu kontroli wersji jest ułatwienie przeprowadzania operacji synchronizacji plików projektu przez wszystkich uczestników.



Terminy **system kontroli wersji** oraz **system kontroli rewizji** (ang. *revision control system*) są synonimami.

Git

Git to nowoczesny, rozproszony systemem kontroli wersji. Jego popularność w ciągu ostatnich kilku lat rośnie w ogromnym tempie. Git powstał w 2005 roku dla potrzeb zarządzania kodem źródłowym jądra systemu Linux². Do połowy roku 2012 liczba plików kodu źródłowego jądra Linuksa urosła do 39 000 plików, a liczba programistów, którzy wzięli udział w jego tworzeniu, do blisko 10 000³. W pierwszym tygodniu czerwca 2012 r. w projekcie tym wprowadzono 400 rewizji.

¹ Por. http://pl.wikipedia.org/wiki/System_kontroli_wersji, http://en.wikipedia.org/wiki/Revision_control.

² Por. [http://pl.wikipedia.org/wiki/Git_\(oprogramowanie\)](http://pl.wikipedia.org/wiki/Git_(oprogramowanie)), [http://en.wikipedia.org/wiki/Git_\(software\)](http://en.wikipedia.org/wiki/Git_(software)).

³ Podana statystyka dotyczy tylko okresu, w którym jądro Linuksa było tworzone z wykorzystaniem oprogramowania Git, czyli od kwietnia 2005 r.



Wskazówka

Do ustalenia liczby:

- ◆ plików kodu źródłowego jądra Linuksa,
- ◆ programistów uczestniczących w projekcie
- ◆ i rewizji wykonanych w okresie od 2012-06-01 do 2012-06-08

użyłem poleceń:

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git .
git shortlog -s -n | wc -l
find . -type f -print | grep -v '/\.git/' | wc -l
git log --pretty=oneline --since="2012-06-01" --until="2012-06-08" | wc -l
```

Na tej podstawie możemy stwierdzić, że Git sprawdził się jako narzędzie do zarządzania bardzo dużym projektem.

Głównymi zaletami Gita są jego:

- ◆ lokalność;
- ◆ kontrola spójności danych;
- ◆ optymalizacja pod kątem rozgałęziania;
- ◆ fakt, że jest to system rozproszony;
- ◆ wydajność.

Jak przebiega praca nad projektem stosującym Git?

Repozytoria

Git śledzi zmiany plików w obrębie konkretnego folderu. Nie ma znaczenia, czy folder ten zawiera kod źródłowy programu komputerowego, rękopis książki czy stronę WWW. Folder, którego zawartość jest kontrolowana przez Git, będziemy nazywali **repozytorium** (ang. *repository*). Repozytoria⁴ zawierają specjalny podfolder *.git*, w którym zapisywane są szczegółowe dane o śledzonych plikach.

Przyjmijmy, że mamy na dysku folder *C:\strona-www*, który zawiera stronę WWW. Struktura folderów oraz pliki projektu są przedstawione na rysunku 1.1.

⁴ Git umożliwia tworzenie dwóch rodzajów repozytoriów: zwykłych i surowych (ang. *bare*). Podany opis dotyczy repozytoriów zwykłych. Więcej informacji o repozytoriach zwykłych i surowych znajdziesz w rozdziale 6.

Rysunek 1.1.
Pliki przykładowego projektu



Jeśli folder *C:\strona-www* zaczniemy śledzić przy użyciu oprogramowania Git, program Git utworzy wówczas podfolder *C:\strona-www\.git*. Zawartość folderu *C:\strona-www* po zainicjalizowaniu nowego repozytorium będzie taka jak na rysunku 1.2.

Rysunek 1.2.
Program Git tworzy w folderze projektu specjalny podfolder *.git*



Zwróć uwagę, że powstaje tylko jeden folder *.git* dla całego projektu. W folderach *css/* oraz *js/* nie pojawiają się żadne dodatkowe pliki ani foldery.

Dla programu Git nie ma znaczenia struktura oraz liczba folderów i plików zawartych wewnątrz folderu *C:\strona-www*. Git będzie kontrolował wszystkie pliki i foldery zawarte wewnątrz folderu *C:\strona-www*. Nie ma także znaczenia sposób wprowadzania poprawek. Wszystkie pliki projektu mogą być modyfikowane dowolnymi edytoremami. Co więcej, w folderze *C:\strona-www* możemy wykonywać wszystkie operacje plikowe, takie jak kopiowanie, usuwanie czy zmiana nazwy w standardowy sposób (np. wykorzystując eksplorator Windows). Sposób wykonywania operacji plikowych jest nieistotny.



Folder zawierający repozytorium Gita możemy przemianować i przenieść w dowolne miejsce. Na przykład folder *strona-www* możemy przemianować na *abc* i przenieść do *C:\moje\nowe\abc*. Operacja zmiany nazwy oraz przeniesienie folderu projektu w inne miejsce nie mają wpływu na repozytorium.

Zatwierdzanie zmian

Do wprowadzania zmian w projekcie służy specjalna **operacja zatwierdzania** (ang. *commit*). Nie jest ona nigdy wykonywana przez Git automatycznie. Jeśli uznamy, że bieżący stan plików i folderów jest istotny, należy samodzielnie wykonać operację zatwierdzania.



Operację zatwierdzania możemy w pewnym uproszczeniu traktować jako zapisanie bieżącego stanu wszystkich plików i folderów projektu w danej chwili.

Wykonanie operacji zatwierdzania powoduje zapisanie **rewizji** (ang. *commit, revision*). Każda rewizja zawiera szczegółowe dane składające się m.in. z:

- ◆ identyfikatora rewizji,
- ◆ danych osoby wykonującej rewizję,
- ◆ daty i godziny wykonania rewizji,
- ◆ identyfikatorów poprzednich rewizji,
- ◆ informacji o zmodyfikowanych plikach i katalogach.



W oryginalnej dokumentacji zarówno **operacja zatwierdzania**, jak i **rewizja** są zazwyczaj określane jednym angielskim terminem: *commit*. W podręczniku będę stosował terminy **operacja zatwierdzania** oraz **rewizja**.

Przyjmijmy, że zatwierdzamy projekt widoczny na rysunku 1.2, w wyniku czego powstaje rewizja:

```
commit:
  id:      a
  autor:   gajdaw
  data:    2012-06-20 10:53
  parent:  brak
```

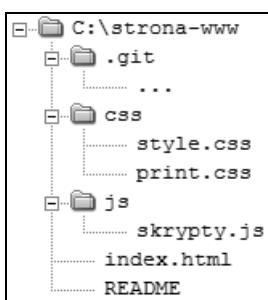


Szczegółowe dane każdej rewizji są zapisywane w folderze *.git*.

Następnie utwórzmy w projekcie pliki *css/print.css* oraz *README*. Stan projektu jest przedstawiony na rysunku 1.3. Teraz wykonajmy drugą operację zatwierdzania, w wyniku czego w folderze *.git* zostanie zapisana druga rewizja:

```
commit:
  id:      b
  autor:   gajdaw
  data:    2012-06-21 16:10
  parent:  a
```

Rysunek 1.3.
Stan projektu po utworzeniu plików *print.css* oraz *README*



Zwróć uwagę, że na rysunku 1.3 pojawiły się dwa nowe pliki. Dla operacji zatwierdzania liczba wprowadzonych zmian jest nieistotna. W pojedynczej rewizji możemy

zmodyfikować równie dobrze jeden, jak i 1000 plików. Ważne jest jedynie, byśmy wprowadzili jakąkolwiek modyfikację. Git nie pozwoli na utworzenie rewizji, gdy żaden plik ani zawartość żadnego folderu nie zostały zmodyfikowane.

Następnie utwórzmy strony:

```
strony/strona-1.html  
strony/strona-2.html  
strony/strona-3.html  
strony/strona-4.html  
strony/strona-5.html
```

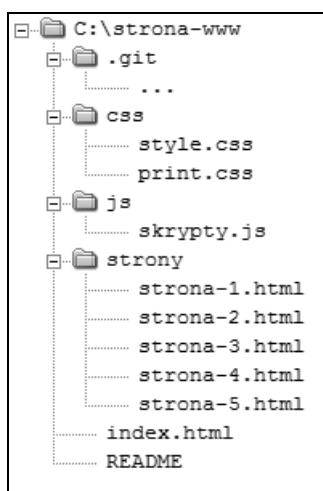
i wykonajmy kolejną operację zatwierdzania, która utworzy trzecią rewizję:

```
commit:  
  id:      c  
  autor:   gajdaw  
  data:    2012-06-25 09:00  
  parent:  b
```

Stan projektu jest przedstawiony na rysunku 1.4.

Rysunek 1.4.

Stan projektu po
utworzeniu plików
w folderze strony



W ten sposób w repozytorium pojawiły się trzy rewizje o identyfikatorach:

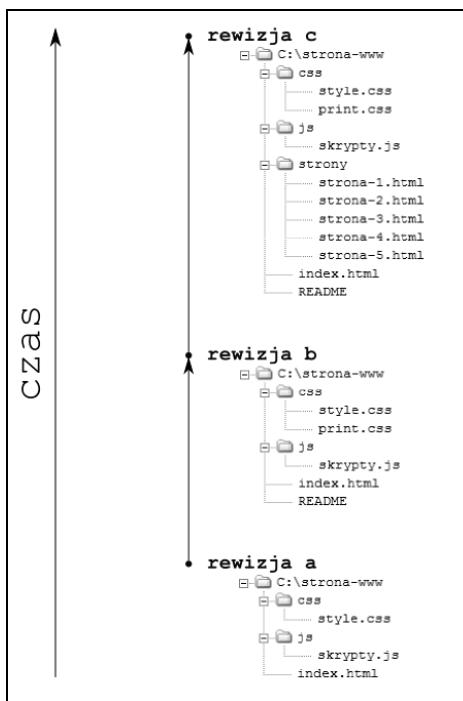
a
b
c

Rewizja a stanowi zapis stanu projektu z rysunku 1.2. Rewizja b zawiera stan projektu widoczny na rysunku 1.3. Rewizja c odpowiada natomiast stanowi, który jest widoczny na rysunku 1.4.

Oprogramowanie Git pozwala na przywrócenie stanu projektu odpowiadającego dowolnej rewizji. Możemy więc powiedzieć, że Git dodaje do systemu plików wymiar czasowy. Stan plików w dowolnej chwili rejestrujemy, wykonując operację zatwierdzania.

Historię projektu możemy przedstawić w postaci diagramu widocznego na rysunku 1.5. Oś czasu takiego diagramu będziemy na kolejnych diagramach pomijali, gdyż zawsze będzie ona skierowana do góry (tj. najnowsza rewizja będzie się zawsze znajdowała na samej górze, a najstarsza — na samym dole).

Rysunek 1.5.
Historia projektu przedstawionego na rysunkach 1.2, 1.3 oraz 1.4



Praca grupowa

Praca grupowa nad projektem polega na udostępnianiu własnych rewizji innym uczestnikom projektu. Odbywa się to w taki sposób, że na serwerze umieszczamy repozytorium, które jest współdzielone. Każdy uczestnik projektu tworzy lokalną kopię repozytorium na swoje potrzeby. We własnym lokalnym repozytorium możemy zatwierdzać zmiany (tj. tworzyć rewizje) bez żadnych ograniczeń. Po utworzeniu nowych rewizji wykonane zmiany przesyłamy na serwer. W zależności od konfiguracji operacja taka może wymagać zatwierdzenia przez administratora.

Po ewentualnym zatwierdzeniu przez administratora nasze rewizje stają się dostępne w głównym repozytorium na serwerze. Teraz każdy uczestnik projektu może pobrać wykonane przez nas zmiany.

Git umożliwia dokładne sprawdzenie różnic pomiędzy dwoma dowolnymi rewizjami. W ten sposób osoba zatwierdzająca rewizje na serwerze ma możliwość dokładnej analizy zmian wprowadzonych przez danego użytkownika.

W takim modelu pracy wykonujemy następujące operacje:

- ♦ pobranie aktualnego stanu repozytorium z serwera do własnego lokalnego repozytorium (polecenie `git pull`),
- ♦ przesłanie własnego lokalnego repozytorium na serwer (polecenie `git push`),
- ♦ zatwierdzenie wykonanych zmian w lokalnym repozytorium (polecenie `git commit`).

System Git prowadzi pełną kontrolę operacji `pull` oraz `push` i nigdy nie powoduje utraty danych. Jeśli przesyłanie danych z serwera lub na serwer nie jest możliwe (np. inny użytkownik zmodyfikował dokładnie ten sam fragment pewnego pliku, wprowadzając inne zmiany), operacja nie zostanie wówczas przeprowadzona, o czym zostaniemy dokładnie poinformowani. W takiej sytuacji musimy usunąć ewentualne konflikty oraz ponownie wykonać operacje `pull` i `push`.

Hosting projektów Git

Bardzo duży wpływ na popularyzację Gita ma serwis <http://github.com>. Jest to obecnie najpopularniejsze rozwiązanie hostingowe projektów open source. W połowie roku 2011 github.com wyprzedził takie platformy hostingowe jak Source Forge oraz Google Code⁵.

Github umożliwia darmowy hosting projektów open source w oparciu o system Git. Oprócz wygodnego interfejsu zapewniającego dostęp do szczegółowych informacji na temat projektu i wprowadzanych zmian, Github zapewnia kontrolę nad wprowadzonymi rewizjami⁶ oraz dostęp do zintegrowanego systemu śledzenia błędów⁷. Oto lista przykładowych projektów, które są prowadzone w serwisie Github:

- ♦ Git
<https://github.com/git/git>
- ♦ witryna <http://git-scm.com>
<https://github.com/github/gitscm-next>
- ♦ rękopis książki Scotta Chacona pt. *Pro Git*
<https://github.com/progit/progit>
- ♦ jądro Linuksa
<https://github.com/torvalds/linux>
- ♦ jQuery
<https://github.com/jquery/jquery>
- ♦ Node.js
<https://github.com/joyent/node>

⁵ Por. <https://github.com/blog/865-github-dominates-the-forges>.

⁶ Służą do tego tzw. żądania aktualizacji (ang. *pull requests*).

⁷ Por. <http://pl.wikipedia.org/wiki/Bugtracker>.

- ◆ Dojo
<https://github.com/dojo/dojo>
- ◆ HTML 5 boilerplate
<https://github.com/h5bp/html5-boilerplate>
- ◆ Symfony
<https://github.com/symfony/symfony>
- ◆ Zend Framework 2
<https://github.com/zendframework/zf2>
- ◆ Ruby on Rails
<https://github.com/rails/rails>
- ◆ Curl
<https://github.com/bagder/curl>

Drugim bardzo popularnym rozwiązaniem hostingowym dla projektów prowadzonych z wykorzystaniem Gita jest <http://bitbucket.org>. Jego funkcjonalność jest zbliżona do <http://github.com>. Serwis Bitbucket różni się od serwisu Github sposobem licencjonowania. Jest on darmowy nie tylko dla projektów o kodzie otwartym, ale także dla projektów prywatnych, czyli takich, których kod nie jest publicznie dostępny.

Czego się nauczysz z tego podręcznika?

Czytając podręcznik:

- ◆ poznasz podstawy pracy w systemie Git,
- ◆ nauczysz się posługiwać wierszem poleceń do wydawania najważniejszych komend Gita,
- ◆ nauczysz się operować gałęziami lokalnymi i zdalnymi,
- ◆ poznasz strukturę repozytoriów i dowiesz się, jak je synchronizować,
- ◆ poznasz serwisy Github oraz Bitbucket,
- ◆ nauczysz się prowadzić projekty grupowe z wykorzystaniem serwisów Github oraz Bitbucket.

Dokumentacja

Podstawowym źródłem informacji o oprogramowaniu Git jest dokumentacja zawarta w folderze:

C:\Program Files (x86)\Git\doc\git\html

Po wydaniu w wierszu poleceń komendy:

```
git help git
```

ujrzysz opis zawarty w pliku:

```
C:\Program Files (x86)\Git\doc\git\html\git.html
```

Kolejnymi dokumentami, które warto przeczytać, są:

```
doc\git\html\gittutorial.html  
doc\git\html\gittutorial-2.html  
doc\git\html\everyday.html  
doc\git\html\user-manual.html  
doc\git\html\gitglossary.html
```

Opis poszczególnych komend Gita uzyskasz, wydając komendy:

```
git add --help  
git branch --help  
git clone --help  
git config --help  
itd.
```

lub:

```
git help add  
git help branch  
git help clone  
git help config  
itd.
```

Dokumentacja Gita jest także dostępna w Internecie na stronie:

<http://www.kernel.org/pub/software/scm/git/docs/>



Dokumentacja Gita dzieli dostępne polecenia na:

- ◆ polecenia wysokopoziomowe (ang. *porcelain*),
- ◆ polecenia niskopoziomowe (ang. *plumbing*).

Książka omawia głównie polecenia wysokopoziomowe.

Dodatkowym źródłem obszernych informacji jest witryna internetowa:

<http://git-scm.com>

książka Scotta Chacona pt. *Pro Git*:

<http://git-scm.com/book/pl>

oraz dokumentacja serwisów *github.com* i *bitbucket.org*:

<https://help.github.com>

<https://confluence.atlassian.com/display/BITBUCKET/bitbucket+Documentation+Home>

Pełny spis literatury zawierający najważniejsze podręczniki opisujące Git jest zawarty w dodatku A.

Rozdział 2.

Instalacja programu Git

Oprogramowanie Git dla platform Windows, Linux, Mac oraz Solaris znajdziemy na stronie:

<http://git-scm.com/downloads>

Po zainstalowaniu klienta Git uruchom wiersz poleceń i w dowolnym folderze wydaj komendę:

```
C:\>git --version
```

W odpowiedzi ujrzyś komunikat informujący o wersji zainstalowanego oprogramowania:

```
git version 1.7.11.msysgit.0
```

W ten sposób upewnisiś się, że Git jest gotowy do pracy.

Podczas instalacji klienta Git dla systemu Windows ujrzyś pytania przedstawione na rysunkach 2.1 oraz 2.2.

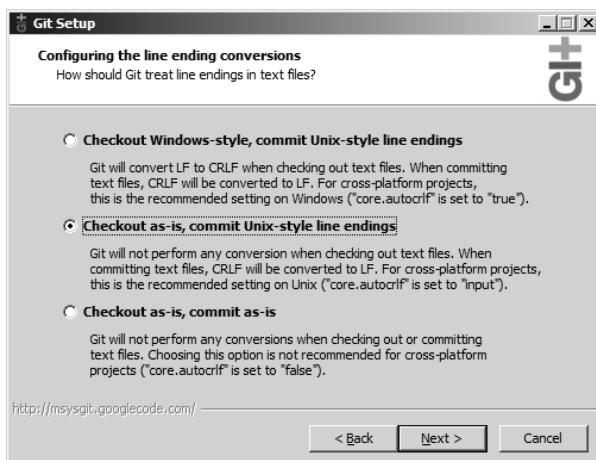
Rysunek 2.1.

Ustalanie wartości zmiennej środowiskowej PATH podczas instalacji klienta Git w systemie Windows



Rysunek 2.2.

Ustawianie sposobu konwersji znaków końca wiersza podczas instalacji klienta Git w systemie Windows



Opcje z rysunku 2.1 decydują o tym, w jaki sposób program instalacyjny zmodyfikuje ścieżki dostępu do programów. Opcja pierwsza powoduje, że ścieżki nie będą modyfikowane. Po zainstalowaniu programu wywołanie polecenia git nie będzie działało. Wymagana będzie ręczna modyfikacja ścieżek.

Druga opcja powoduje, że do ścieżek dostępu dodana zostanie ścieżka prowadząca do programu git. W wierszu poleceń będziemy mogli wydawać komendę git, lecz nie będziemy mogli wydawać komend ssh, wc, find itd.

Ostatnia opcja powoduje, że do ścieżek dostępu dodane zostaną ścieżki prowadzące do programu git oraz do narzędzi ssh, wc itd. Po zainstalowaniu programu będziemy mogli — bez żadnych dalszych modyfikacji — wywoływać polecenia: git, ssh, wc itd.

Wpływ opcji z rysunku 2.1 na ścieżki poszukiwań jest podsumowany w tabeli 2.1.

Tabela 2.1. Wpływ opcji z rysunku 2.1 na ścieżki poszukiwań

Opcja	Sposób modyfikacji ścieżek dostępu
<i>Use Git Bash Only</i>	Niemodyfikowane
<i>Run Git from the Windows Command Prompt</i>	Dodany folder: <i>C:\Program Files (x86)\Git\cmd</i>
<i>Run Git and included Unix tools from the Windows Command Prompt</i>	Dodane foldery: <i>C:\Program Files (x86)\Git\cmd</i> <i>C:\Program Files (x86)\Git\bin</i>

Opcje z rysunku 2.2 ustalają sposób konwersji znaków złamania wiersza. Pierwsza możliwość powoduje, że podczas zapisywania plików w repozytorium znaki złamania wiersza będą konwertowane z formatu CRLF na format LF, natomiast podczas odtwarzania plików — z formatu LF na format CRLF.

Druga opcja powoduje, że podczas zapisywania plików w repozytorium znaki końca wiersza będą konwertowane z CRLF na LF. Odtwarzanie plików nie będzie powodowało żadnych konwersji.

Trzecia opcja wyłącza wszelkie konwersje znaków końca wiersza.



Zagadnienia dotyczące konwersji znaków końca wiersza są szczegółowo omówione w rozdziale 24.

Wpływ opcji z rysunku 2.2 na konwersję znaków końca wiersza jest podsumowany w tabeli 2.2.

Tabela 2.2. Wpływ opcji z rysunku 2.2 na konwersję znaków końca wiersza

Opcja	Sposób modyfikacji znaków końca wiersza
<i>Checkout Windows-style, commit Unix-style</i>	Checkout (odtwarzanie plików): LF => CRLF Commit (zapisywanie plików): CRLF => LF
<i>Checkout as-is, commit Unix style</i>	Commit (zapisywanie plików): CRLF => LF
<i>Checkout as-is, commit as-is</i>	-



Najprostszym sposobem pracy w programie Git przy użyciu serwera *github.com* w systemie Windows jest instalacja dedykowanego klienta dla platformy Windows: <http://windows.github.com>. To rozwiązanie wyklucza jednak korzystanie z własnego serwera SSH oraz z serwera *bitbucket.org*. Z tego powodu omawiam w książce metody pracy w programie Git za pośrednictwem konsoli. Jest to najbardziej uniwersalne rozwiązanie, niewiążące nas z żadnym dostawcą usług hostingowych.

Konsola Gita w systemie Windows

Po zainstalowaniu klienta Git w systemie Windows pojawi się specjalna konsola Gita. Przypomina ona wiersz poleceń Windows. Wygląd konsoli Gita w systemie Windows jest przedstawiony na rysunku 2.3.

Rysunek 2.3.

Konsola Gita instalowana z klientem Git dla platformy Windows

The screenshot shows a terminal window titled "MINGW32~". The title bar also displays "Welcome to Git (version 1.7.11-preview20120620)". The main area of the window contains the following text:
Run 'git help git' to display the help index.
Run 'git help <command>' to display help for specific commands.
gajdaw@GAJDAW ~
\$

Konsola Gita jest oznaczona ikoną z rysunku 2.4.

Rysunek 2.4.
Ikona konsoli Gita



Do pracy z klientem Git możemy wykorzystywać zarówno standardowy wiersz poleceń dostępny w systemie, jak i konsolę Gita. Różnica polega na tym, że w konsoli Gita są dostępne linuksowe polecenia plikowe, m.in.:

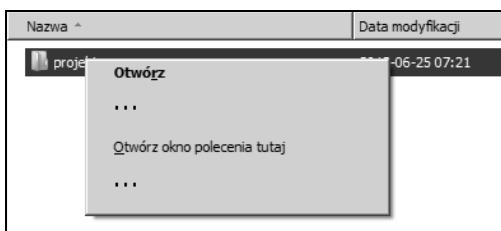
```
cat
find
grep
wc
```

Dostępność powyższych poleceń w konsoli Windows zależy od opcji wybranej w oknie dialogowym z rysunku 2.1.

Ułatwienia uruchamiania konsoli w systemie Windows

W systemie Windows 7 pojawiła się opcja menu kontekstowego pozwalająca na uruchomienie wiersza poleceń w dowolnym folderze. Opcja ta nazywa się *Otwórz okno polecenia tutaj*. Jest ona przedstawiona na rysunku 2.5.

Rysunek 2.5.
Opcja pozwalająca na uruchomienie wiersza poleceń w wybranym folderze



W podobny sposób możemy uruchamiać konsolę bash. W celu włączenia opcji menu kontekstowego *Uruchom bash* uruchom edytor rejestru Windows. W oknie dialogowym *Start/Uruchom* wpisz nazwę programu *regedit* i naciśnij *Enter*.

Najpierw dodaj w rejestrze klucz:

```
HKEY_CLASSES_ROOT\Directory\shell\PrzejscieDoBash
```

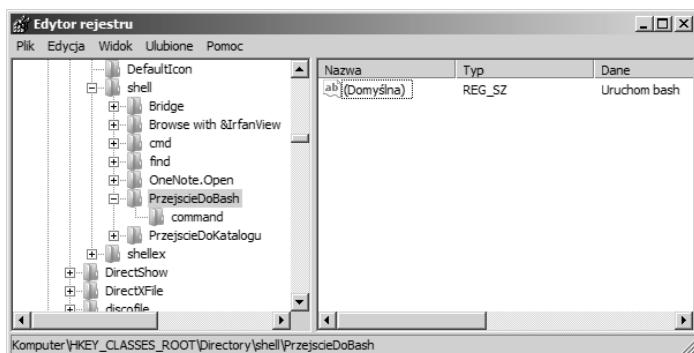
o wartości:

```
Uruchom bash
```

Klucz taki jest przedstawiony na rysunku 2.6.

Następnie dodaj klucz:

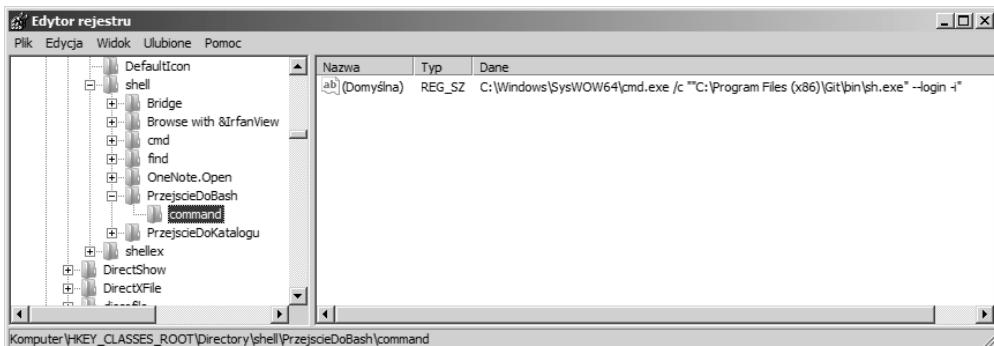
```
HKEY_CLASSES_ROOT\Directory\shell\PrzejscieDoBash\command
```

Rysunek 2.6.**Klucz****PrzejscieDoBash**

o wartości:

```
C:\Windows\SysWOW64\cmd.exe /c ""C:\Program Files (x86)\Git\bin\sh.exe" --login -i"
```

Klucz taki jest przedstawiony na rysunku 2.7.

**Rysunek 2.7. Klucz PrzejscieDoBash\command**

Po zmodyfikowaniu rejestru zrestartuj system.

Jeśli teraz klikniesz w eksploratorze prawym przyciskiem nazwę folderu, w menu kontekstowym ujrzyś opcję *Uruchom bash*.

Podstawowa konfiguracja klienta Git

Oprogramowanie Git umożliwia dostosowanie parametrów pracy programu przy użyciu opcji konfiguracyjnych. Wydaj polecenie:

```
git config -l
```

W ten sposób sprawdzisz, jakie obecnie obowiązują ustawienia.

Jak pamiętasz, wszystkie rewizje w repozytorium zawierają informacje o autorze. Z tego powodu pracę nad własnymi projektami należy poprzedzić ustaleniem danych autora. Służą do tego komendy:

```
git config --global user.name "Imie Nazwisko"  
git config --global user.email you@example.com
```

Jeśli Twoje imię lub nazwisko zawiera polskie znaki diakrytyczne, to powyższą komendę wydaj w konsoli bash.

W systemie Windows 7 komenda git config --global zapisuje podane zmienne konfiguracyjne w pliku `.gitconfig` w folderze użytkownika, np. `C:\Users\nazwakonta\.gitconfig`. Oczywiście zawartość tego pliku możesz także edytować dowolnym programem. Pamiętaj, że plik ten powinien być zakodowany w UTF-8. Oto fragment mojego pliku `.gitconfig`:

```
[user]  
name = Włodzimierz Gajda  
email = gajdaw@gajdaw.pl
```

Jeśli polecenie git config wydasz bez parametru `--global`, np.:

```
git config user.name "Imie Nazwisko"
```

podane ustawienia zostaną wówczas zapisane wyłącznie w repozytorium, w którym powyższa komenda zostanie wydana.

Edytor

Niektóre komendy programu Git uruchamiają edytor tekstowy. Domyślnym edytorem tekstowym jest vi. Osobom nieznającym systemu Linux program vi przysparza wiele trudności. Jeśli zamiast programu vi chcesz używać domyślnego edytora, który w Twoim systemie jest powiązany z rozszerzeniem `.txt` (np. Notatnika), zainstaluj program GitPad:

<https://github.com/github/GitPad>

W celu odinstalowania programu GitPad usuń z rejestru wpis:

HKEY_CURRENT_USER\Environment\EDITOR

i zrestartuj system.

Rozdział 3.

Tworzenie repozytoriów

Repozytorium, w skrócie repo, to projekt prowadzony w systemie Git. Nowe repozytoria możemy tworzyć na dwa sposoby:

- ◆ inicjalizując nowy projekt,
- ◆ klonując istniejące repozytorium.

Iinicjalizacja nowego repozytorium

Nowe repozytorium inicjalizujemy komendą:

```
git init
```

Komendę należy wydać w pustym folderze, który jest przeznaczony na projekt, lub w folderze zawierającym pliki projektu. Komenda `git init` nigdy nie powoduje utraty danych. Jeśli wydasz ją w folderze, który zawiera repozytorium Gita (tj. w którym istnieje folder `.git`), to istniejący projekt nie zostanie uszkodzony.

Oczywiście jeśli komendę `git init` wydasz w błędny folderze i zechcesz zrezygnować z prowadzenia utworzonego repozytorium, wystarczy usunąć folder `.git`.

Ćwiczenie 3.1

W folderze `cw-03-01/` utwórz nowe repozytorium Gita.

ROZWIĄZANIE

Utwórz folder `cw-03-01/`, uruchom w nim wiersz poleceń, po czym wydaj komendę:

```
git init
```

W odpowiedzi ujrzesz komunikat:

```
Initialized empty Git repository in C:/...
```

Wydanie komendy `git init` powoduje utworzenie w folderze `cw-03-01/` folderu `.git`. To jest jedyne jej działanie. Po wydaniu komendy sprawdź zawartość folderu `cw-03-01/`. Oczywiście nowo utworzone repozytorium jest puste.

Folder `.git` przechowuje komplet informacji o repozytorium. W folderze tym na razie nie wprowadzamy żadnych modyfikacji.



Uwaga

Polecenie:

```
git init
```

możesz wydać w folderze, który nie jest pusty. Pliki i foldery zawarte w folderze, w którym wydajesz komendę `git init`, nie mają żadnego wpływu na przebieg wykonania komendy. Utworzone repozytorium i tak będzie puste: żadne pliki ani foldery nie zostaną do niego automatycznie dodane.

Zwróć uwagę, że repozytorium zawarte w folderze `cw-03-01/` zostało wykonane lokalnie. Do pracy lokalnej nie potrzebujemy ani połączenia sieciowego, ani konta w żadnym serwisie internetowym. Nie musimy także instalować żadnego demona czy serwera Gita. Program konsolowy Gita zainstalowany w rozdziale 2. to wszystko, co jest potrzebne do prowadzenia lokalnych repozytoriów Gita.



Wskazówka

Czy warto wykorzystywać Git do projektów tworzonych przez jedną osobę?

Oczywiście głównym zadaniem oprogramowania Git jest organizacja pracy grupowej nad projektem. Jak pamiętasz, jedną z głównych zalet Gita jest łatwość tworzenia, przełączania i scalania gałęzi. Ta cecha powoduje, że wszystkie swoje prace, także projekty jednoosobowe, realizuję z wykorzystaniem Gita. Daje mi to możliwość rozgałęziania prac nad projektem. Ponadto korzystając z serwisów `github.com` oraz `bitbucket.org`, zyskuję kopię zapasową danych oraz wbudowany system śledzenia błędów.

Klonowanie repozytoriów

Drugą metodą tworzenia repozytoriów jest klonowanie. Komenda:

```
git clone [adres]
```

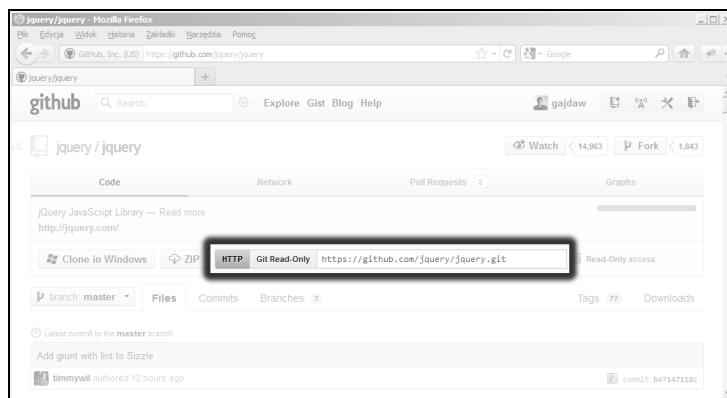
tworzy w bieżącym folderze kopię repozytorium o podanym adresie. W ten sposób możemy wykonać kopię dowolnego repozytorium dostępnego publicznie. Odwiedź stronę projektu jQuery:

<https://github.com/jquery/jquery>

W centralnej części strony ujrzyś publiczny adres repozytorium Gita. Adres ten jest zaznaczony na rysunku 3.1.

Rysunek 3.1.

Adres publicznego repozytorium projektu *jQuery* dostępnego w serwisie *Github*



Korzystając z przycisków *HTTP* oraz *Git Read-Only*, możesz zmienić stosowany protokół:

Adres HTTP: <https://github.com/jquery/jquery.git>

Adres Git Read-Only: <git://github.com/jquery/jquery.git>

W celu wykonania kopii repozytorium należy wydać komendę `git clone`. Parametrem komendy może być dowolny z powyższych adresów, czyli *HTTP* lub *Git Read-Only*.

Wynik działania obu poniższych komend będzie identyczny:

```
git clone https://github.com/jquery/jquery.git  
git clone git://github.com/jquery/jquery.git
```

Po wykonaniu powyższej operacji na dysku lokalnym zostanie utworzone repozytorium będące dokładną kopią oryginalnego repozytorium. Pomijając kwestię gałęzi, możemy powiedzieć, że oba repozytoria są identyczne. Każde z nich zawiera kompletną historię projektu.

W identyczny sposób klonujemy repozytoria z serwera *bitbucket.org*. Odwiedź adres:

<https://bitbucket.org/atlassian/aui-archive>

Znajdziesz tam informacje o projekcie Atlassian User Interface oraz polecenie, którego należy użyć do sklonowania projektu:

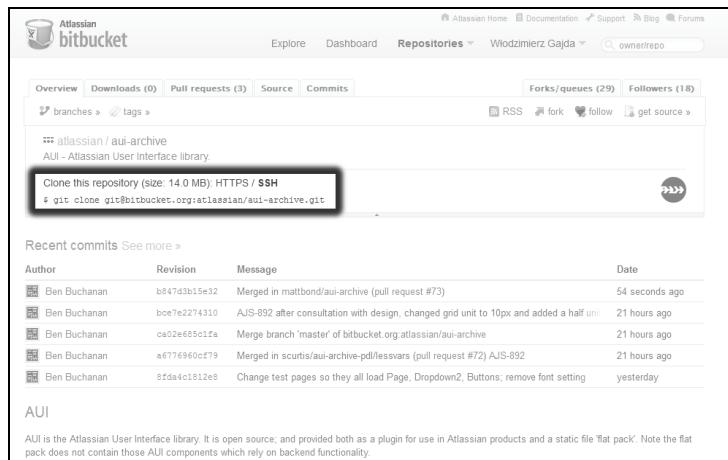
```
git clone git@bitbucket.org:atlassian/aui-archive.git
```

Polecenie to jest zaznaczone na rysunku 3.2.

Ćwiczenie 3.2

W folderze *cw-03-02/* utwórz kopię projektu *jQuery*. Sprawdź liczbę plików projektu, liczbę osób biorących udział w tworzeniu projektu oraz liczbę rewizji.

Rysunek 3.2.
Klonowanie projektu
dostępnego
na serwerze
bitbucket.org



ROZWIĄZANIE

Utwórz folder *cw-03-02/* i wydaj w nim komendę:

```
git clone https://github.com/jquery/jquery.git .
```

Spowoduje ona utworzenie kopii repozytorium jQuery w bieżącym folderze. Jeśli w poleceniu:

```
git clone https://github.com/jquery/jquery.git .
```

pominiesz końcową kropkę:

```
git clone https://github.com/jquery/jquery.git
```

w bieżącym folderze zostanie wówczas utworzony podfolder *jquery/*. Repozytorium będzie zawarte w podfolderze.

Uwaga

Operacja klonowania powoduje pobranie z serwera na dysk lokalny kompletnego repozytorium. W dużych projektach, takich jak jądro Linuksa czy Symfony, operacja ta będzie trwała nawet kilkadesiąt minut. Podczas pracy nad projektem nie jest to uciążliwe, gdyż klonowanie projektu wykonujemy tylko jeden raz, rozpoczynając pracę nad projektem.

W celu sprawdzenia uczestników projektu wydaj komendę:

```
git shortlog -s -n
```

Liczba uczestników projektu poznasz, wydając polecenie:

```
git shortlog -s -n | wc -l
```

W celu ustalenia liczby plików projektu wydaj komendę:

```
find . -type f -print | grep -v '\.git/' | wc -l
```

Do sprawdzenia liczby rewizji wprowadzonych w projekcie w pierwszym tygodniu czerwca użyj polecenia:

```
git log --pretty=oneline | wc -l
```

Na zakończenie sprawdź rozmiar folderu *cw-03-02/* oraz zawartego w nim folderu *.git*. Okaże się, że folder *cw-03-02/* zajmuje 15,3 MB, z czego 14,3 MB jest zawarte w folderze *.git!* Powód takiej dysproporcji jest taki, że w folderze *.git* zapisane są informacje o wszystkich modyfikacjach.

Oto statystyka projektu jQuery wykonana 27 czerwca 2012 roku:

- ◆ liczba uczestników: 156
- ◆ liczba plików: 131
- ◆ liczba rewizji: 4336
- ◆ rozmiar całego folderu *jQuery*: 15,3 MB
- ◆ rozmiar folderu *.git*: 14,3 MB

Ćwiczenie 3.3

Uzupełnij tabelkę 3.1. Adresy repozytoriów podanych projektów znajdziesz w rozdziale 1.

Tabela 3.1. Statystyka wybranych projektów

	jQuery	HTML 5 boilerplate	Symfony	jądro Linuksa	Ruby on Rails
Liczba uczestników	156				
Liczba plików	131				
Liczba rewizji	4336				
Rozmiar całego repozytorium (w MB)	15.3				
Rozmiar folderu <i>.git</i> (w MB)	14.3				

Badanie historii projektu

Do sprawdzania zmian wprowadzonych w repozytorium służy komenda:

```
git log
```

Wydana bez parametrów drukuje informacje o ostatnich modyfikacjach w projekcie. Do przewijania wyników komendy `git log` służą spacja oraz *Q*.

Przykładowy wydruk generowany polecienniem `git log` wygląda tak jak na listingu 3.1.

Listing 3.1. Przykładowy wydruk generowany poleciem `git log`

```

commit b47147118c8f291e7db34b377706b6f48ac45b3e
Author: timmywil <timmywillisn@gmail.com>
Date:   Sun Jun 24 15:33:04 2012 -0400

    Add grunt with lint to Sizzle

commit d4b5a1974d2b60989e9fe1158ce0b8c1f577c2d1
Author: Rick Waldron <waldron.rick@gmail.com>
Date:   Sat Jun 23 19:43:12 2012 -0400

    Make @VERSION replace regex global. Fixes #11960

```

Historia projektu składa się z **rewizji**. Jak pamiętasz, rewizja może dotyczyć jednego lub wielu plików. Do identyfikacji rewizji Git wykorzystuje funkcję skrótu SHA-1¹. Na listingu 3.1 widoczne są informacje o dwóch rewizjach. Pierwsza z nich jest oznacona skrótem rozpoczętym się od znaków b4714711. Z wydruku możemy odczytać, kto i kiedy wykonał rewizję.



Wskazówka

Skróty SHA-1 są wykorzystywane jako jednoznaczne identyfikatory rewizji. Wydając polecenia Gita, nie musisz jednak przepisywać kompletnych skrótów SHA-1. Wystarczy kilka początkowych znaków. Minimalna długość skróconego SHA-1 to cztery znaki. Oczywiście skrócone SHA-1 musi być jednoznaczne. Jeśli nie jest, to należy użyć kolejnych kilku znaków (aż do uzyskania unikatowego skrótu).

Polecenie `git log` pozwala filtrować dane oraz formatować sposób prezentacji. Parametr `--pretty` ustala domyślny format wydruku. Może on przyjmować wartości:

<code>oneline</code>	— ID rewizji, krótki opis
<code>short</code>	— format <code>oneline</code> + autor rewizji
<code>medium</code>	— format <code>short</code> + data rewizji
<code>full</code>	— format <code>medium</code> bez daty + osoba dołączająca rewizję do projektu
<code>fuller</code>	— format <code>full</code> + data wykonania rewizji, data dołączenia rewizji
<code>email</code>	— podstawowe informacje w formacie <code>email</code>
<code>raw</code>	— informacje dotyczące powiązań rewizji (<code>parent</code> , <code>tree</code>)
<code>format</code>	— formatowanie zdefiniowane przez użytkownika

Do filtrowania zmian służą m.in. parametry:

<code>-n</code>	— liczba interesujących nas rewizji np. <code>-7</code> oznacza ostatnie 7 rewizji
<code>--since="yyyy-mm-dd"</code>	— data początkowa
<code>--until="yyyy-mm-dd"</code>	— data końcowa
<code>--author=gajdaw</code>	— rewizje wykonane przez wybranego autora

¹ Por. <http://pl.wikipedia.org/wiki/SHA-1>.

Polecenie:

```
git log --pretty=oneline
```

wydrkuje listę ostatnich rewizji w formacie jednowierszowym. Wydruk taki będzie zawierał identyfikator rewizji (czyli skrót SHA-1) oraz krótki opis. Wydruk z listingu 3.1 w formacie oneline przyjmie postać:

```
b47147118c8f291e7db34b377706b6f48ac45b3e Add grunt with lint to Sizzle  
d4b5a1974d2b60989e9fe1158ce0b8c1f577c2d1 Make @VERSION replace regex global...
```

Dodatkowe parametry `--abbrev-commit` oraz `-abbrev=4` maksymalnie skróćą drukowane skróty SHA-1:

```
git log --abbrev-commit --abbrev=4 --pretty=oneline
```

Do wydrukowania informacji o trzech ostatnich rewizjach w pełnym formacie użyj polecenia:

```
git log -3
```

Komenda:

```
git log -5 --author=janek
```

wydrkuje natomiast rewizje wykonane przez podanego autora.

Ćwiczenie 3.4

W folderze `cw-03-04/` utwórz kopię projektu Symfony 2: <https://github.com/symfony/symfony>. Poleceniem:

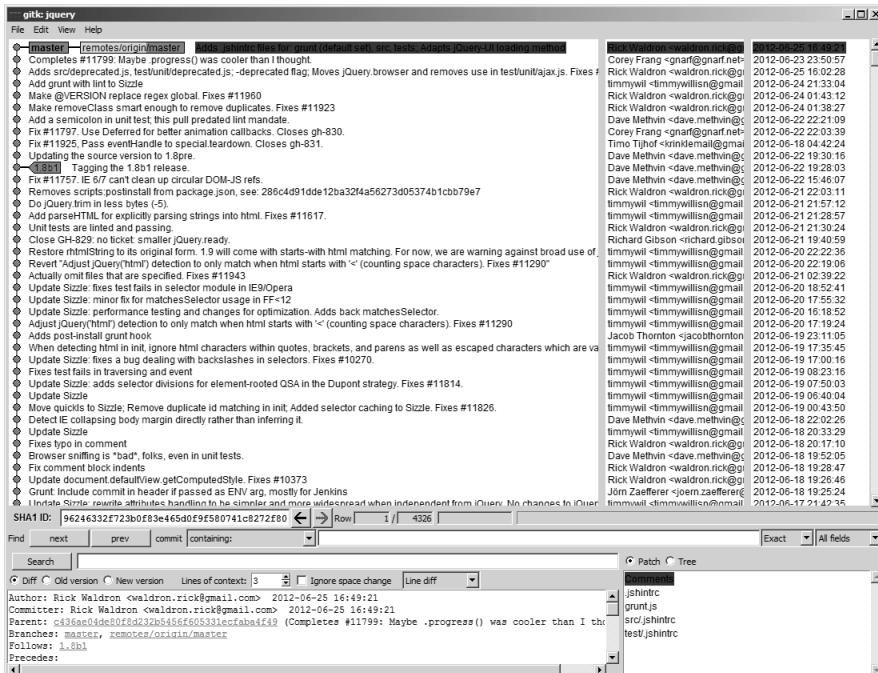
```
git log --pretty=oneline --author=stloyd | wc -l
```

ustal liczbę rewizji wykonanych przez użytkownika `stloyd`. W analogiczny sposób sprawdź informacje o rewizjach wykonanych przez pozostałych najaktywniejszych polskich programistów uczestniczących w rozwoju Symfony 2:

- ◆ *stloyd*
- ◆ *jakzal*
- ◆ *canni*
- ◆ *michal-pipa*
- ◆ *gajdaw*
- ◆ *l3l0*

Wizualizacja historii projektu

Do programu Git dołączona jest aplikacja Git GUI, która umożliwia wizualne przedstawienie historii projektu. Uruchom program Git GUI, otwórz repozytorium jQuery z ćwiczenia 3.2, po czym z menu głównego wybierz operację: *Repository/Visualise master's History*. Ujrzyś okno dialogowe z rysunku 3.3.



Rysunek 3.3. Historia projektu jQuery w aplikacji Git GUI



Aplikację z rysunku 3.3 możesz uruchomić w dowolnym repozytorium komendą:

gitk

Komenda:

git gui

uruchamia natomiast w danym repozytorium program Git GUI.

Pojedyncze kropki widoczne z lewej strony okna to rewizje. Wybierając konkretną rewizję, poznasz następujące szczegóły:

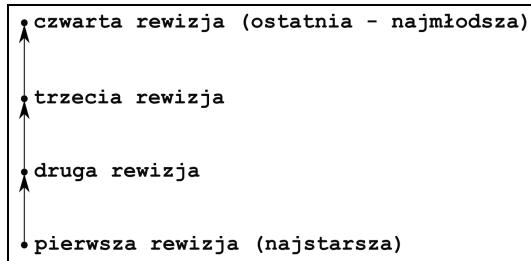
- ◆ Kto jest jej autorem?
- ◆ Kiedy została wykonana?
- ◆ Jaki jest jej skrót SHA-1?

- ♦ Które pliki zostały zmienione?
- ♦ Jakie zmiany wprowadzono w plikach?

Historię projektu możemy traktować jako ciąg następujących po sobie rewizji przedstawionych na rysunku 3.4.

Rysunek 3.4.

Historia projektu to ciąg następujących po sobie rewizji: od najstarszej do najmłodszej

**Wskazówka**

Istnieją dwa sposoby tworzenia repozytoriów:

- ♦ inicjalizacja nowego repozytorium poleceniem `git init`,
- ♦ sklonowanie istniejącego repozytorium poleceniem `git clone`.

Pod względem zawartości generowanego folderu repozytoria możemy podzielić na:

- ♦ repozytoria zwykłe tworzone poleceniami `git init`, `git clone`;
- ♦ repozytoria surowe tworzone poleceniami `git init --bare`, `git clone --bare`.

Repozytoria surowe będą wykorzystywane do synchronizacji projektów. Folder repozytorium surowego odpowiada zawartości folderu `.git/` repozytorium zwykłego. Bardziej szczegółowe informacje o repozytoriach surowych są zawarte w rozdziale 6.

Rozdział 4.

Obszar roboczy

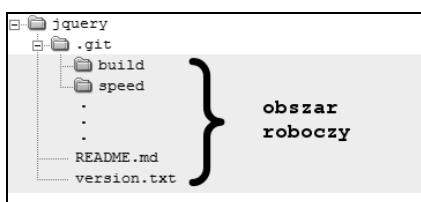
Wiemy, że w folderze zawierającym repozytorium Gita znajduje się specjalny folder o nazwie `.git`. Cała pozostała zawartość folderu projektu jest nazywana **obszarem roboczym** (ang. *working area, working directory*). Obszar roboczy repozytorium zawartego w folderze `C:\repozytorium` jest przedstawiony na rysunku 4.1.

Rysunek 4.1.
Obszar roboczy projektu z folderu C:\repozytorium



W projekcie jQuery obszar roboczy zawiera m.in. foldery `build` i `speed` oraz pliki `README.md` i `version.txt`. Obszar roboczy projektu jQuery z ćwiczenia 3.2 jest przedstawiony na rysunku 4.2.

Rysunek 4.2.
Obszar roboczy projektu jQuery



Stan plików projektu w obszarze roboczym możemy przywrócić do stanu odpowiadającemu ostatniej rewizji, wydając polecenie:

```
git reset --hard
```



Polecenia:

```
git reset --hard  
git reset --hard HEAD
```

są równoważne. Znaczenie parametru `HEAD` zostanie przedstawione w rozdziale 9.

Nawet jeśli usuniemy wszystkie pliki z obszaru roboczego, to po wydaniu powyższej komendy stan plików zostanie odtworzony na podstawie zawartości folderu `.git`. W podobny sposób możemy odtwarzać stan projektu odpowiadający dowolnej rewizji:

```
git reset --hard [SHA-1]
```

np.

```
git reset --hard aabbcc01020304...
```

Polecenie to przywraca pliki w obszarze roboczym do stanu z rewizji aabbcc01020304... oraz usuwa z historii projektu wszystkie późniejsze rewizje oraz zmiany plików w obszarze roboczym. Pamiętaj, że w wielu przypadkach zmiany te mogą okazać się nieodwracalne!



Uwaga

Wiele poleceń opisanych w podręczniku działa w sposób destrukcyjny i nieodwracalny. Przykładem może być polecenie:

```
git reset --hard ...
```

Wydane w nieodpowiednim momencie może spowodować nieodwracalną utratę danych. Z tego powodu podczas pierwszej lektury książki sugeruję, by użycie polecień ćwiczyć na plikach tymczasowych `lorem.txt`, `ipsum.txt`, `poniedzialek.txt`, `wtorek.txt` itd., a nie na najnowszym projekcie realizowanym komercyjnie.

Ćwiczenie 4.1

Sklonuj repozytorium HTML 5 boilerplate:

```
https://github.com/h5bp/html5-boilerplate
```

po czym usuń w nim zawartość obszaru roboczego (czyli skasuj wszystkie pliki i foldery oprócz folderu `.git`). Następnie odtwórz zawartość obszaru roboczego tak, by odpowiadała ostatniej rewizji.

ROZWIĄZANIE

W folderze `cw-04-01/` wydaj polecenie:

```
git clone git://github.com/h5bp/html5-boilerplate.git .
```

Następnie usuń wszystkie pliki i foldery¹ zawarte w folderze `cw-04-01/`. W celu przywrócenia stanu projektu wydaj polecenie:

```
git reset --hard
```

Wszystkie pliki w obszarze roboczym zostaną odtworzone zgodnie ze stanem z ostatniej rewizji.

¹ Oczywiście nie usuwaj folderu `.git`.



Wskazówka

Stan plików w obszarze roboczym możemy także przywrócić polecienniem:

```
git checkout -f  
git checkout -f [SHA-1]
```

Polecenie to nie powoduje usuwania rewizji z historii projektu. Będziemy się nim posługiwali w części drugiej, poświęconej gałęziom. Po wydaniu polecenia:

```
git checkout -f [SHA-1]
```

repozytorium znajduje się w stanie detached HEAD. Stan ten jest szczegółowo omówiony w rozdziale 13.

Przywracanie stanu projektu, który zawiera nowe pliki

Jeśli utworzysz w projekcie nowe pliki, żadne z poleceń:

```
git reset --hard  
git checkout -f
```

nie spowoduje wówczas ich usunięcia. Jeśli zmodyfikowałeś projekt w wielu folderach, dodając nowe pliki, i nie jesteś z tych zmian zadowolony, to:

- ♦ Usuń wszystkie pliki z obszaru roboczego.
- ♦ Wydaj jedną z komend:

```
git reset --hard  
git checkout -f
```

Po tej operacji polecenie:

```
git status
```

zwróci informacje, że projekt nie zawiera żadnych modyfikacji.



Wskazówka

Dużo lepszą metodą rezygnacji z wprowadzonych zmian jest usunięcie gałęzi. Taką technikę pracy poznamy w części drugiej, która jest poświęcona gałęziom.

Ćwiczenie 4.2

Sklonuj rękopis książki Scotta Chacona pt. *Pro Git*:

<https://github.com/progit/progit>

W repozytorium zmodyfikuj kilka plików. Następnie odtwórz zawartość obszaru roboczego tak, by odpowiadała ostatniej rewizji.

ROZWIĄZANIE

W folderze *cw-04-02/* wydaj polecenie:

```
git clone https://github.com/progit/progit .
```

Następnie zmodyfikuj w projekcie kilka wybranych plików, usuń kilka plików oraz utwórz kilka plików. Poleciem:

```
git status
```

sprawdź wprowadzone zmiany.

W celu przywrócenia stanu repozytorium usuń wszystkie pliki z obszaru roboczego, po czym wydaj jedną z komend:

```
git reset --hard  
git checkout -f
```

Poleciem:

```
git status
```

upewnij się, że obszar roboczy nie zawiera żadnych zmian.



Wskazówka

Komendy:

```
git reset --hard  
git checkout -f
```

nie wpływają na nowe pliki utworzone w obszarze roboczym. Jeśli więc podczas wykonywania ćwiczenia 4.2 pominiesz usuwanie wszystkich plików z obszaru roboczego, wydana na końcu ćwiczenia komenda:

```
git status
```

zwróci wówczas informacje o nowo utworzonych plikach.

Rozdział 5.

Tworzenie rewizji i przywracanie stanu plików

Tworzenie rewizji

Pracę w systemie Git możemy sprowadzić do dwóch prostych zadań:

- ◆ modyfikacji plików i folderów projektu,
- ◆ zapamiętywania stanu wszystkich plików projektu w postaci rewizji.

Operacje edycyjne przeprowadzamy za pomocą dowolnych narzędzi, m.in. edytorów tekstu, środowiska IDE, menedżerów plików itd.

Druga operacja, zapamiętywanie stanu plików projektu w postaci rewizji, sprowadza się do wydania polecień:

```
git add -A  
git commit -m "Krótki opis rewizji..."
```

Szczegółowy opis polecień git add oraz git commit jest zawarty w kolejnym rozdziale.



Polecenia

```
git add -A  
git commit -m "komunikat..."
```

tworzą rewizję stanowiącą zapis **bieżącego stanu wszystkich plików w folderze roboczym** (ang. *snapshot*). Są one równoważne poleceniom:

```
git add .  
git commit -a -m "komunikat..."
```

Ponadto parametr -A komendy git add jest równoważny parametrowi --all. Obie poniższe operacje są równoważne:

```
git add -A  
git add --all
```

W przypadku pominięcia parametru -A polecenie git add wymaga podania nazwy pliku lub folderu. Poniższa komenda jest niepoprawna:

```
git add
```



Wskazówka

Bieżący stan wszystkich plików w folderze roboczym jest określany angielskim terminem *snapshot*, który w niektórych polskich publikacjach jest tłumaczony jako migawka.

Ćwiczenie 5.1

Wykonaj repozytorium przedstawione na rysunku 1.5.

ROZWIĄZANIE

Pracę rozpoczęnij od utworzenia folderu C:\strona-www:

```
mkdir strona-www
```

Następnie w folderze C:\strona-www dowolną metodą utwórz foldery i pliki z rysunku 1.1:

```
C:\strona-www\css  
C:\strona-www\css\style.css  
C:\strona-www\css\js  
C:\strona-www\css\skrypty.js  
C:\strona-www\css\index.html
```

Wszystkie pliki w ćwiczeniu mogą być puste.

W kolejnym kroku zainicjalizuj nowe repozytorium:

```
git init
```

Stan folderu projektu jest teraz taki jak na rysunku 1.2: w folderze C:\strona-www pojawił się folder .git.

W celu zapisania stanu plików projektu w postaci rewizji wydaj polecenia:

```
git add -A  
git commit -m "Pierwsza rewizja - a"
```

W historii projektu pojawiła się pierwsza rewizja. Sprawdź to poleceniem:

```
git log
```

Następnie w folderze projektu (czyli w obszarze roboczym) utwórz pliki:

```
C:\strona-www\css\print.css  
C:\strona-www\README
```

po czym wykonaj drugą rewizję:

```
git add -A  
git commit -m "Druga rewizja - b"
```

Historia projektu zawiera dwie rewizje, co stwierdzisz poleceniem:

```
git log --pretty=oneline
```

Na zakończenie utwórz folder:

```
C:\strona-www\strony\
```

zawierający pięć plików:

```
C:\strona-www\strony\strona-1.html  
C:\strona-www\strony\strona-2.html  
C:\strona-www\strony\strona-3.html  
C:\strona-www\strony\strona-4.html  
C:\strona-www\strony\strona-5.html
```

i zatwierdź zmiany:

```
git add -A  
git commit -m "Trzecia rewizja - c"
```

Projekt zawiera teraz trzy rewizje. Polecenie:

```
git log --abbrev-commit --abbrev=4 --pretty=oneline
```

zwróci przykładowe wyniki:

```
9cd5 Trzecia rewizja - c  
683b Druga rewizja - b  
4bf7 Pierwsza rewizja - a
```

Przywracanie stanu plików do wybranej rewizji

Pliki obszaru roboczego możemy odtworzyć na podstawie dowolnej rewizji. Polecenie:

```
git checkout [SHA-1]
```

przywraca pliki z przestrzeni roboczej do stanu z podanej rewizji. Polecenie to powoduje zmianę stanu repozytorium Gita. Po wydaniu polecenia odłączymy się od głównej gałęzi projektu, o czym informuje komunikat:

```
You are in 'detached HEAD' state.
```



Szczegółowe informacje dotyczące odłączania gałęzi znajdziesz w rozdziale 13.

Bieżącą gałąź możemy sprawdzić poleceniem:

```
git branch
```

Jeśli stan plików przywróci się do wybranej rewizji:

```
git checkout [SHA-1]
```

to wynikiem polecenia git branch będzie komunikat:

```
* (no branch)  
  master
```

potwierdzający, że nie znajdujemy się w żadnej gałęzi. W celu powrotu do gałęzi głównej wydajemy komendę:

```
git checkout master
```

Pliki w przestrzeni roboczej zostaną przywrócone do stanu z ostatniej rewizji. Teraz polecenie:

```
git branch
```

zwróci komunikat:

```
* master
```

informujący, że znajdujemy się na głównej gałęzi master.

Ćwiczenie 5.2

Zakładając, że w repozytorium z ćwiczenia 5.1 polecenie:

```
git log --abbrev-commit --abbrev=4 --pretty=oneline
```

zwraca wyniki:

```
9cd5 Trzecia rewizja - c  
683b Druga rewizja - b  
4bf7 Pierwsza rewizja - a
```

przywrócić stan plików w obszarze roboczym do wszystkich rewizji w kolejności od pierwszej do ostatniej.

ROZWIĄZANIE

Poleceniem:

```
git branch
```

sprawdź, że domyślną gałęzią jest gałąź o nazwie master.

W celu przywracenia stanu plików z rewizji pierwszej wydaj komendę:

```
git checkout 4bf7
```

Sprawdź, że zawartość folderu projektu jest teraz identyczna jak na rysunku 1.2. Poleceniem:

```
git branch
```

przekonasz się, że nie jesteś już na gałęzi master. Poleceniem:

```
git checkout master
```

powrócisz na gałąź master. Pliki w folderze projektu będą teraz zgodne z rysunkiem 1.4.

Poleceniem:

```
git checkout 683b
```

przywróci stan plików w obszarze roboczym do drugiej rewizji. Sprawdź, że zawartość folderu projektu jest teraz identyczna jak na rysunku 1.3. Poleceniem:

```
git branch
```

przekonasz się, że nie jesteś już na gałęzi master. Poleceniem:

```
git checkout master
```

powrócisz na gałąź master. Pliki w folderze projektu będą teraz zgodne z rysunkiem 1.4.

Ćwiczenie 5.3

Sklonuj repozytorium jQuery i przywróć je do stanu z pierwszej rewizji wykonanej w 2012 roku.

Polecenie¹:

```
git log  
-5  
--abbrev-commit --abbrev=5  
--pretty=format:"%h %cd"  
--since="2012-01-01" --until="2012-01-15"
```

generuje wydruk zawierający informacje o rewizjach wykonanych od 1 do 15 stycznia 2012 r.:

```
c0da4 Thu Jan 12 20:30:45 2012 -0500  
d8289 Thu Jan 12 20:14:51 2012 -0500  
6c8dd Thu Jan 12 20:04:17 2012 -0500  
cc5e8 Thu Jan 12 19:57:04 2012 -0500  
d0711 Wed Jan 11 22:16:30 2012 -0500
```

Pierwszą rewizję z 2012 roku jest:

```
d0711 Wed Jan 11 22:16:30 2012 -0500
```

W celu przywrócenia plików do postaci z podanej rewizji wydajemy komendę:

```
git checkout d0711
```

¹ W celu zwiększenia czytelności polecenie to zostało zapisane w kilku wierszach. Wydając polecenie, należy je jednak zapisać w jednym wierszu.



Wskazówka

Do sformatowania wydruku polecenia git log w postaci:

<SKRÓCONE SHA-1> <DATA>

stosujemy parametr:

--pretty=format:"%h %cd"

Ciąg znaków ujęty w cudzysłów może zawierać specjalne znaczniki formatujące:

- ◆ %H — SHA-1;
- ◆ %h — skrócone SHA-1;
- ◆ %an — nazwa autora;
- ◆ %ae — adres e-mail autora.

Pełny wykaz znaczników formatujących poznasz, wydając komendę git log --help.

Ćwiczenie 5.4

Sklonuj repozytorium Symfony 2 i przywróć je do stanu z pierwszej rewizji wykonanej przez użytkownika 1310.

Polecenie:

```
git log  
--abbrev-commit --abbrev=4  
--pretty=format:"%h %cd %an"  
--author=1310
```

generuje wydruk:

```
83ff Sun Jun 10 14:54:03 2012 +0200 1310
```

Stan repozytorium przywracamy poleceniem:

```
git checkout 83ff
```

Przenoszenie repozytorium

Na folderze zawierającym repozytorium Gita możemy wykonywać operacje zmiany nazwy oraz przenoszenia.

Ćwiczenie 5.5

Zmień nazwę folderu *strona-www/* z ćwiczenia 5.1 na *webpage/*. Następnie przenieś folder *webpage/* z folderu głównego do dowolnego innego folderu na dysku. Poleceniami:

```
git status  
git log -pretty=oneline
```

upewnij się, że przeniesienie i zmiana nazwy folderu nie mają wpływu na repozytorium.

Rezygnacja z repozytorium

Usunięcie folderu `.git` zawartego w folderze projektu powoduje usunięcie całej historii projektu. Przed ewentualnym wykonaniem takiej operacji należy zawsze upewnić się, że pliki w folderze roboczym odpowiadają odpowiedniej rewizji.

Ćwiczenie 5.6

W folderze `strona-www/` z ćwiczenia 5.1 usuń folder `.git`. Polecenie:

```
git log
```

zwróci komunikat:

```
fatal: Not a git repository (or any of the parent directories): .git
```

informujący o tym, że bieżący folder nie zawiera repozytorium Gita. Oczywiście repozytorium możemy ponownie zainicjować:

```
git init
```

jednak cała poprzednia historia (czyli rewizje z rysunku 1.5) zostały bezpowrotnie utracone.



Wskazówka

Bardziej złożone projekty mogą składać się z wielu podprojektów, z których każdy jest prowadzony w programie Git. Przykładem może być projekt Symfony 2, który zawiera kilkanaście pakietów zależnych. Po zainstalowaniu wszystkich pakietów zależnych projekt będzie zajmował około 100 MB, z czego znaczna większość będzie zawarta w folderach `.git` podprojektów. W celu usunięcia wszystkich folderów `.git` zawartych w bieżącym folderze należy użyć polecenia:

```
find . -name .git -type d -exec rm -fr {} \;
```


Rozdział 6.

Stany plików

Uproszczony model pracy: przestrzeń robocza i repozytorium

W poprzednim rozdziale wykonaliśmy pierwsze rewizje. Pojedyncza rewizja była tworzona dwoma poleceniami:

```
git add -A  
git commit -m "komunikat..."
```

Polecenie git add -A powoduje, że rewizja obejmie wszystkie pliki. Dzięki użyciu parametru -A operacje usuwania plików mogę bez żadnych obaw wykonywać w standardowy sposób (np. w eksploratorze Windows), bez konieczności użycia dodatkowego polecenia git rm.

Bezpośrednio po wykonaniu operacji:

```
git add -A  
git commit -m "komunikat..."
```

zawartość każdego pliku jest zapisana w repozytorium i może być odzyskana (np. poleciением git checkout). Pliki takie nazwiemy **aktualnymi** (ang. *unmodified*).

Jeśli plik zmodyfikujemy, to jego zawartość w obszarze roboczym oraz w repozytorium będą się różnić. Pliki takie nazwiemy **zmodyfikowanymi** (ang. *modified*).

Jeśli w obszarze roboczym utworzymy nowy plik, plik ten nie będzie wówczas zawarty w repozytorium Gita (dokładniej: nie będzie zawarty w żadnej rewizji; kolejne rewizje również nie spowodują zapisania stanu pliku). Pliki takie nazwiemy **nieśledzonymi** (ang. *untracked*).

W ten sposób podzieliliśmy pliki na trzy grupy:

- ◆ pliki nieśledzone,
- ◆ pliki zmodyfikowane,
- ◆ pliki aktualne.

Dwa polecenia:

```
git add -A  
git commit -m "komunikat..."
```

zapisują wszystkie pliki w repozytorium, a więc zmieniają stan wszystkich plików nieśledzonych oraz zmodyfikowanych na aktualny.

Pracując w uproszczonym modelu, dzielimy projekt na dwa obszary: **przestrzeń roboczą** oraz **repozytorium**.

Plik nieśledzony to taki plik, który jest zawarty w przestrzeni roboczej, ale nie jest zawarty w repozytorium. Każdy nowo utworzony plik jest nieśledzony aż do momentu wydania komend `git add` oraz `git commit`.

Plik aktualny to taki plik, który jest zapisany w repozytorium i w przestrzeni roboczej i jego zawartość w obu lokalizacjach jest identyczna. Bezpośrednio po wydaniu komend `git add` oraz `git commit` wszystkie pliki są aktualne.

Plik zmodyfikowany to plik, którego treść w repozytorium różni się od treści w obszarze roboczym. Jeśli po wydaniu polecień `git add` oraz `git commit` zmienimy treść pliku w obszarze roboczym, to plik ten stanie się zmodyfikowany.



W folderze `.git` zapisana jest baza danych zawierająca wszystkie rewizje projektu. Baza ta jest określana terminem **repozytorium**. Pisząc: „plik `dane.txt` został zapisany w repozytorium”, mam na myśli fakt, że plik został zawarty w rewizji. Innymi słowy plik został zapisany w bazie danych zawartej wewnątrz folderu `.git`.

Termin **repozytorium** jest zatem używany w dwóch kontekstach. Określamy nim zarówno cały folder zawierający projekt, jak i bazę danych zawartą w folderze `.git`, w której Git zapisuje rewizje konkretnego projektu.

Indeksowanie

Wykonywanie rewizji przebiega dwuetapowo. Najpierw ustalamy, które pliki mają zostać uwzględnione w następnej rewizji, a następnie zatwierdzamy rewizję. Polecenie:

```
git add -A
```

ustala, że następna rewizja ma uwzględnić wszystkie pliki z obszaru roboczego. Komenda:

```
git commit -m "komunikat..."
```

tworzy natomiast rewizję, która obejmie wszystkie pliki.

Innymi słowy każdy plik zawarty w przestrzeni roboczej oraz repozytorium możemy scharakteryzować, ustalając, czy obejmie go następna operacja git commit.

Pliki, które zostaną objęte przez następną operację git commit, nazwiemy **zaindeksowanymi** (ang. *staged*).

Pliki zmodyfikowane, które nie zostały zaindeksowane, nazwiemy **niezaindeksowanymi** (ang. *unstaged*).

Oczywiście podczas pracy nad projektem nie tylko tworzymy pliki, ale także je usuwamy. W związku z tym indeksacja dotyczy także plików, które nie istnieją w obszarze roboczym, ale istnieją w bazie danych *.git* (czyli w repozytorium). Rozważmy przypadek pliku aktualnego (czyli zapisanego w jednej z rewizji), który usuwamy z obszaru roboczego. Po usunięciu z obszaru roboczego plik — pomimo tego, że nie istnieje już w obszarze roboczym — jest niezaindeksowany. W celu zatwierdzenia operacji usuwania pliku należy najpierw go zaindeksować, a następnie wykonać rewizję.

Diagram stanów

Jeśli utworzymy nowy plik *dane.txt*, to plik ten jest **nieśledzony**. W odniesieniu do pliku nieśledzonego nie ma sensu stosowanie dalszych przymiotników: niezaindeksowany, zmodyfikowany, niezmodyfikowany, gdyż plik nieśledzony nie może być zaindeksowany. Nigdzie nie jest też zapisana informacja o modyfikacjach pliku: nie wiemy, kto i kiedy plik zmienił. W kolejnym rozdziale zajmiemy się scharakteryzowaniem kolejnej cechy pliku nieśledzonego: wyróżnimy pliki ignorowane i nieignorowane. Na razie o pliku *dane.txt* mówimy krótko: nieśledzony.



W odniesieniu do plików **nieśledzonych** stosowanie przymiotników **zmodyfikowany** i **niezmodyfikowany** nie ma sensu. Pliki nieśledzone podzielimy w następnym rozdziale na **ignorowane** i **nieignorowane**.

Po wydaniu komendy:

```
git add dane.txt
```

plik jest **zaindeksowany**. Mówiąc dokładniej, jest on **śledzony**, **zmodyfikowany** i **zaindeksowany**. Przymiotniki „śledzony” oraz „zmodyfikowany” możemy jednak pojmować: każdy plik zaindeksowany jest śledzony i zmodyfikowany.



W odniesieniu do plików **zaindeksowanych** stosowanie przymiotników **zmodyfikowany** oraz **śledzony** nie ma sensu. Każdy plik zaindeksowany jest **śledzony** i **zmodyfikowany**.

Kolejna komenda:

```
git commit -m "komunikat..."
```

tworzy rewizję obejmującą wszystkie pliki zaindeksowane, zatem także plik *dane.txt*. Po wykonaniu rewizji pliki, które były **zaindeksowane**, stają się **aktualne**. Oczywiście każdy plik aktualny jest śledzony i niezmodyfikowany.



Wskazówka W odniesieniu do plików **aktualnych** stosowanie przymiotników **śledzony** i **niezmodyfikowany** nie ma sensu, gdyż każdy plik aktualny jest śledzony i niezmodyfikowany.

Jeśli zmienimy teraz treść pliku *dane.txt*, to plik ten pozostanie plikiem **śledzonym**, będzie on **zmodyfikowany** oraz **niezaindeksowany**. W takiej sytuacji będziemy stosowali jeden przymiotnik: **niezaindeksowany**. Będzie to jednoznaczne, gdyż plik taki zawsze jest plikiem śledzonym. Ponadto nie można zaindeksować pliku aktualnego. Połączenia przymiotników „**nieśledzony** i **zaindeksowany**” oraz „**aktualny** i **zaindeksowany**” nie mają zatem sensu.



Wskazówka Śledzone pliki, które zostały zmodyfikowane, ale jeszcze nie są zaindeksowane, będą mieli nazwę **niezaindeksowanymi**.

Jeśli ponownie wykonamy polecenia:

```
git add -A  
git commit -m "komunikat..."
```

to oczywiście wszystkie pliki projektu, w tym także plik *dane.txt*, staną się aktualne. Co się stanie, gdy z obszaru roboczego usuniemy aktualny plik *dane.txt*? Plik ten zniknie z obszaru roboczego, ale pozostanie w ostatniej rewizji. W takim przypadku powiemy też, że plik *dane.txt* jest niezaindeksowany — pomimo tego, że już nie istnieje na dysku. Polecenie:

```
git add -A
```

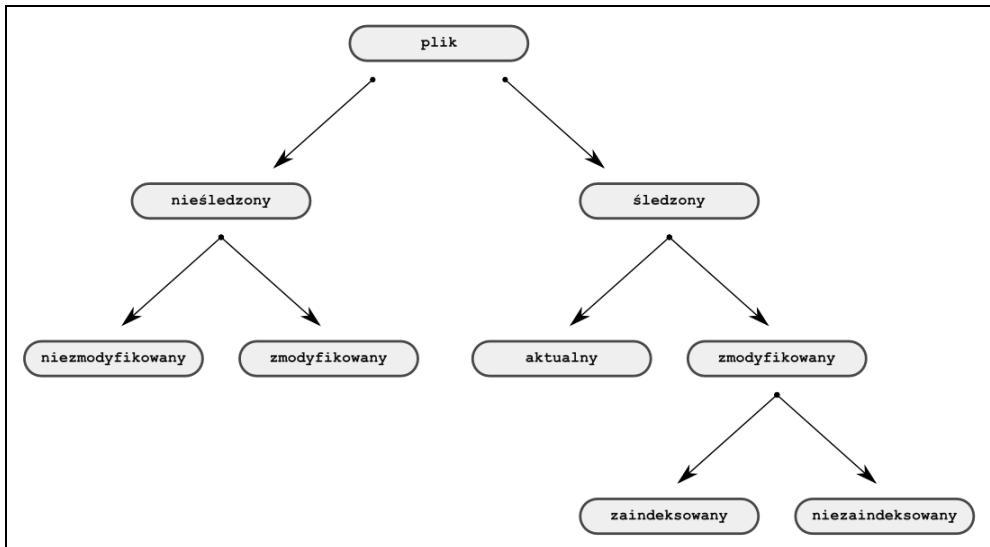
indeksuje wszystkie pliki: nowo utworzone, zmodyfikowane oraz usuwane.

Kompletna hierarchia wszystkich wymienionych stanów pliku jest przedstawiona na rysunku 6.1.

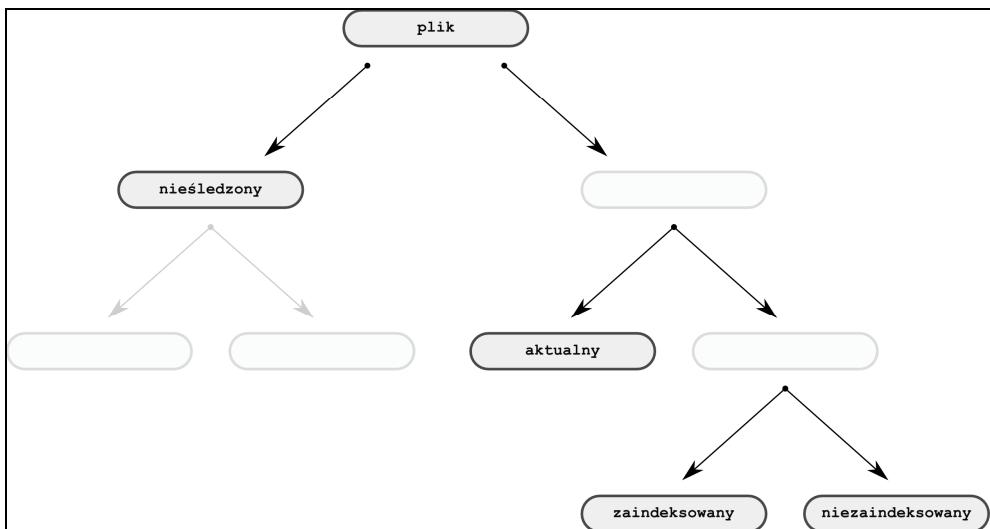
Spośród stanów widocznych na rysunku 6.1 w praktyce będziemy operowali czterema jednoznaczonymi określeniami:

- ◆ nieśledzony,
- ◆ aktualny,
- ◆ zaindeksowany,
- ◆ niezaindeksowany.

Uproszczony schemat stanów plików projektu jest przedstawiony na rysunku 6.2.



Rysunek 6.1. Wszystkie wymienione stany pliku projektu



Rysunek 6.2. Jednoznaczne stany plików projektu

Mając na uwadze operację usuwania plików, pamiętaj, że określenia:

- ♦ zaindeksowany
- ♦ niezaindeksowany

mogą dotyczyć plików, których nie ma w obszarze roboczym.

Obszar roboczy, indeks i repozytorium

Wiemy już, że **obszar roboczy** to zawartość folderu projektu z wykluczeniem specjalnego folderu `.git`. W folderze `.git` zapisana jest baza danych zawierająca rewizje wykonane w projekcie. Baza ta jest nazywana **repozytorium**. Oprócz tego w folderze `.git` zawarty jest specjalny plik nazywany **indeksem** (nazwa pliku to `.git/index`). Indeks to baza danych zawierająca informacje o tym, które pliki zostały zaindeksowane. Pliki zaindeksowane to pliki, które zostaną zapisane w bazie danych `.git` po wydaniu następnej komendy `git commit`.

Pracując w programie Git, operujemy więc trzema obszarami:

- ◆ obszarem roboczym (tj. folderem projektu z wykluczeniem podfolderu `.git`);
- ◆ repozytorium (tj. bazą danych rewizji zawartą w folderze `.git`);
- ◆ indeksem (tj. bazą danych plików zaindeksowanych — czyli przygotowanych do kolejnej rewizji — zawartą w pliku `.git/index`).

W kontekście tych trzech obszarów pliki z rysunku 6.2 (pliki nieśledzone, aktualne, zaindeksowane i niezaindeksowane) możemy scharakteryzować następująco:

Plik nieśledzony występuje w obszarze roboczym, ale nie występuje w repozytorium ani w indeksie.

Plik aktualny występuje w obszarze roboczym i w repozytorium. Wersja w obszarze roboczym i w repozytorium jest identyczna. Pliku takiego nie można zaindeksować.

Indeksacja może dotyczyć zarówno tworzenia, jak i usuwania plików. Dlatego mówimy, że **plik zaindeksowany**:

- ◆ występuje w obszarze roboczym, w repozytorium i w indeksie; wersja zapisana w obszarze roboczym jest różna od wersji zapisanej w repozytorium;
- ◆ lub też nie występuje w obszarze roboczym, a występuje w repozytorium i w indeksie;

natomiast **plik niezaindeksowany**:

- ◆ występuje w obszarze roboczym i w repozytorium, lecz nie występuje w indeksie; wersja zapisana w obszarze roboczym jest różna od wersji zapisanej w repozytorium;
- ◆ lub też nie występuje w obszarze roboczym i w indeksie, a występuje w repozytorium.



Wskazówka

Indeks, czyli baza danych zapisana w pliku `.git/index`, jest określany angielskimi terminami *index* oraz *staging area*. Termin *staging area* bywa tłumaczony jako **przechowalnia** lub **obszar tymczasowy**. Pliki zaindeksowane są określane w dokumentacji terminem *staged*, a niezaindeksowane — terminem *unstaged*. W dalszej części książki będę konsekwentnie stosował terminy: indeks, zaindeksowany oraz niezaindeksowany. Odpowiednikami angielskimi w różnych źródłach są terminy:

- ◆ indeks (tj. baza danych zapisana w pliku `.git/index`) — *index, staging area*;
- ◆ zaindeksowany (tj. występujący w bazie danych `.git/index`) — *staged*;
- ◆ niezaindeksowany (tj. niewystępujący w bazie danych `.git/index`) — *unstaged*.

Modyfikowanie stanu plików repozytorium

Tworzenie plików nieśledzonych

Pliki w obszarze roboczym mogą być tworzone dowolnymi metodami. Możemy wydawać polecenia konsoli, np.:

```
echo "Lorem ipsum" > dane.txt      (Windows, Linux)
touch dane.txt                      (Linux, Windows: konsola git bash)
copy oryginalny.txt skopiowany.txt (Windows)
cp oryginalny.txt skopiowany.txt   (Linux)
```

a także stosować metody kopiuj-wklej w oknie eksploratora oraz używać operacji *Zapisz* w dowolnym programie. Słowniem: sposób tworzenia nowego pliku w obszarze roboczym nie ma żadnego znaczenia. Dla Gita liczy się wyłącznie nazwa pliku i jego treść.

Każdy nowo utworzony plik jest zawsze nieśledzony.

Polecenie git add

Polecenie:

```
git add [nazwa-pliku]
```

powoduje zaindeksowanie pliku. Podany plik może być plikiem nieśledzonym lub niezaindeksowanym (czyli był już w jakiejś rewizji, lecz został zmieniony). Wydanie komendy `git add` w odniesieniu do pliku aktualnego nie powoduje żadnego skutku. Pliku aktualnego nie można zaindeksować.

Jeśli parametrem komendy `git add` jest nazwa folderu, to komenda powoduje zaindeksowanie wszystkich plików i folderów zawartych w danym folderze. Polecenie:

```
git add .
```

indeksuje zatem wszystkie pliki i foldery z obszaru roboczego.

W komendzie git add możemy wykorzystywać znaki * oraz ?:

```
git add *.txt
git add folder/*
git add nowy/fold*
git add *.t?t
```

Znak * zastępuje dowolny ciąg liter, a znak ? — dowolną pojedynczą literę.

Obie poniższe komendy są równoważne i powodują indeksację wszystkich plików nieśledzonych i niezaindeksowanych z obszaru roboczego:

```
git add .
git add *
```



Nie ma innej możliwości zaindeksowania nowego pliku niż przy użyciu komendy git add.

Parametr -A zapisywany alternatywnie jako --all powoduje indeksację wszystkich plików nowych, zmodyfikowanych oraz usuniętych.



Polecenie git add może przyjmować parametry --all oraz --update (w skrócie zapisywane jako -A oraz -u). Ich działanie jest następujące:

- ◆ git add . — obejmuje pliki z przestrzeni roboczej (obejmuje pliki nowe; nie obejmuje plików zmodyfikowanych ani usuniętych).
- ◆ git add --update . — obejmuje pliki z bazy danych (te, które były już śledzone, lecz zostały zmodyfikowane lub usunięte; nie obejmuje plików nowych).
- ◆ git add --all . — obejmuje wszystkie pliki.

Polecenie git commit

Pliki stają się aktualne po wykonaniu **operacji zatwierdzania**:

```
git commit
```

Operacja ta tworzy w historii projektu jedną nową rewizję, zawierającą wszystkie pliki, które w momencie wydawania polecenia były zaindeksowane. Po wydaniu polecenia indeks jest opróżniany. Należy pamiętać, że polecenie to obejmuje wyłącznie pliki zaindeksowane. Nie obejmuje ono plików nieśledzonych, aktualnych ani niezaindeksowanych.

Oczywiście baza danych *index* może zawierać dowolnie wiele plików z obszaru roboczego. Dlatego pojedyncza rewizja może równie dobrze dotyczyć jednego, jak i wielu plików.

Każda rewizja zawiera krótki opis. Po wydaniu polecenia:

```
git commit
```

uruchomiony zostanie edytor tekstu, w którym należy wprowadzić opis rewizji. Opis ten możemy także podać jako wartość parametru -m:

```
git commit -m "pierwsza rewizja..."
```

W takiej sytuacji edytor tekstowy nie jest uruchamiany.

**Wskazówka**

Domyślnym edytorem w programie Git jest **vi**. Jeśli pracujesz w systemie Windows i nie zainstalowałeś programu GitPad (opis instalacji znajdziesz w rozdziale 2.), to w przypadku pominięcia parametru **-m** wydanie komendy **git commit** spowoduje uruchomienie edytora **vi**. Jeśli go nie znasz, to możesz mieć kłopoty. **vi** to jeden z najbardziej kontrowersyjnych programów. Wiele osób uważa go za skrajnie dziwaczny, niewygodny i nieudany. Zwolennicy, a jest wśród nich wielu zawodowych programistów, uważają go za szczyt osiągnięć w dziedzinie edycji plików tekstowych. Do której grupy Ty należysz?

Polecenie **git commit** domyślnie nie uwzględnia plików niezaindeksowanych. Oznacza to, że jeśli plik, który był aktualny, zmodyfikujesz (po modyfikacji plik stanie się niezaindeksowany), po czym wydasz komendę **git commit**, to plik ten nie zostanie zapisany w utworzonej rewizji. Pozostanie on w stanie *niezaindeksowany*. Działanie polecenia **git commit** możesz zmodyfikować przy użyciu opcji **-a**. Komenda:

```
git commit -a
```

obejmie swoim działaniem wszystkie pliki zaindeksowane oraz niezaindeksowane (krótko: wszystkie pliki zmodyfikowane) — także pliki, które były aktualne, a zostały usunięte. Z tego powodu do zapisania bieżącego stanu wszystkich plików repozytorium należy użyć komend:

```
git add -A  
git commit -m "komunikat..."
```

lub:

```
git add .  
git commit -a -m "komunikat..."
```

**Wskazówka**

Polecenie:

```
git commit -a
```

nie obejmuje plików nieśledzonych. Z tego powodu rewizje wykonujemy dwoma poleceniami:

```
git add .  
git commit -a -m "komunikat..."
```

Polecenie **git add** indeksuje pliki nieśledzone.

Parametr **-a** polecenia **git commit** powoduje uwzględnienie plików śledzonych (tj. zmodyfikowanych oraz usuniętych).

Polecenie **git rm**

Do zaindeksowania usuwanego pliku służy polecenie:

```
git rm [nazwa-pliku]
```

Podany plik powinien być aktualny.

Domyślnie polecenie to wykonuje dwie operacje:

- ◆ Jeśli plik o podanej nazwie istnieje w obszarze roboczym, to komenda powoduje usunięcie pliku.
- ◆ Plik zostaje zaindeksowany (oczywiście w indeksie jest zawarta informacja mówiąca o tym, że plik ten ma zostać usunięty w kolejnej rewizji).

Plik podany jako parametr polecenia `git rm` nie musi istnieć. Możemy go usunąć standardowymi operacjami plikowymi, a następnie wydać komendę `git rm`.

Jeśli chcemy wyłącznie zaindeksować usuwanie pliku, pozostawiając plik w obszarze roboczym, należy użyć parametru `--cached`:

```
git rm --cached [nazwa-pliku]
```

Jeśli po wydaniu powyższej komendy wykonamy rewizję, to plik zostanie usunięty w repozytorium (dokładniej: w bazie danych zawartej w folderze `.git`), ale pozostałe w obszarze roboczym. Taką operację wykonuje się np. na plikach, w których wprowadzamy tajne informacje, np. hasła dostępu do bazy danych¹.

Co się stanie, gdy polecenie `git rm` wykonamy na pliku, który nie jest aktualny? Jeśli plik nie jest aktualny, to musi być zmodyfikowany, a zatem zawiera modyfikacje nie-występujące w żadnej rewizji. W takim przypadku usunięcie pliku z dysku spowoduje nieodwracalną utratę danych. Z tego powodu w przypadku plików zmodyfikowanych operacja `git rm` zakończy się niepowodzeniem. Jeśli jesteśmy pewni, że chcemy usunąć podany plik, należy użyć parametru `-f`. Polecenie:

```
git rm -f nazwa-pliku
```

powoduje usunięcie pliku zarówno z repozytorium, jak i z przestrzeni roboczej. Operacja taka zostanie wykonana zarówno na pliku aktualnym, jak i zmodyfikowanym.

Zmiana nazwy pliku

Do zmiany nazwy pliku służy polecenie:

```
git mv stara-nazwa nowa-nazwa
```

Podany plik o nazwie `stara-nazwa` musi istnieć w obszarze roboczym. Może to być plik aktualny bądź zaindeksowany. Polecenie powoduje zmianę nazwy pliku w obszarze roboczym oraz indeksację pliku.

Modyfikowanie plików przy użyciu standardowych operacji plikowych

Pliki w obszarze roboczym możemy usuwać i przemianowywać standardowymi operacjami plikowymi, bez konieczności używania komend `git rm` oraz `git mv`. Możemy

¹ Por. ćwiczenie 7.2.

oczywiście także modyfikować ich treść. Pliki zmodyfikowane, usunięte oraz przemianowane standardowymi operacjami plikowymi przechodzą ze stanu *aktualny* do stanu *niezaindeksowany*.

Z tego powodu kolejna operacja:

```
git commit
```

nie obejmie ich swoim działaniem. Jeśli chcesz, by pliki poddane wymienionym zmianom były automatycznie objęte rewizją, bez konieczności ręcznego wydawania polecień:

```
git add  
git rm  
git mv
```

użyj parametru -A polecenia git add:

```
git add -A
```

lub parametru -a polecenia git commit:

```
git commit -a
```

Parametry te spowodują, że wykonywana rewizja obejmuje swoim działaniem wszystkie śledzone pliki (zarówno zaindeksowane, jak i niezaindeksowane).

Stan repozytorium

Do sprawdzania stanu repozytorium służy komenda:

```
git status
```

Jeśli wydamy ją zaraz po komendzie git init wykonanej w pustym folderze, ujrzymy wówczas wydruk:

```
# On branch master  
#  
# Initial commit  
#  
nothing to commit (create/copy files and use "git add" to track)
```

Wydruk świadczy o tym, że obszar roboczy projektu jest pusty, a repozytorium nie zawiera żadnych rewizji.

Jeśli komendę git status wydamy po sklonowaniu projektu, wydruk będzie następujący:

```
# On branch master  
nothing to commit (working directory clean)
```

Wydruk informuje o tym, że wszystkie pliki projektu są aktualne.

Opcjonalny parametr `-s` powoduje, że polecenie `git status` drukuje wyłącznie informację o wykonanych zmianach, bez jakichkolwiek komentarzy. Na wydruku pominięte są pliki aktualne. Jeśli więc w projekcie nie wykonano żadnych zmian, polecenie:

```
git status -s
```

wydrukuje wówczas jeden pusty wiersz.

W tym stanie mamy gwarancję, że wprowadzenie jakichkolwiek zmian w obszarze roboczym (np. usunięcie plików) jest w pełni odwracalne.

Uproszczony model pracy raz jeszcze

Teraz działanie dwóch poleceń użytych w poprzednim rozdziale, czyli:

```
git add -A  
git commit -m "komunikat..."
```

powinno stać się jasne. Komendą:

```
git add -A
```

indeksujemy wszystkie pliki projektu. Oczywiście wielokrotne wydanie tej komendy:

```
git add -A  
git add -A  
git add -A
```

ma dokładnie ten sam skutek co jednokrotne.

Polecenie:

```
git commit -m "komunikat..."
```

tworzy rewizję, która obejmie wszystkie zaindeksowane pliki. Dzięki użyciu w poleceniu `git add` parametru `-A` pliki projektu mogą przemianowywać i usuwać standaryзовymi operacjami plikowymi bez konieczności wydawania niewygodnych komend `git rm` i `git mv`.

Po wykonaniu dwóch poleceń:

```
git add -A  
git commit -m "komunikat..."
```

wszystkie pliki projektu są aktualne². Polecenie:

```
git status -s
```

zwraca pusty wiersz. W takim stanie zostawiam każde swoje repozytorium, gdy kończę pracę na dany dzień. Niezmiernie rzadko wykonuję rewizje w inny sposób niż przez wydanie następujących po sobie poleceń:

```
git add -A  
git commit -m "komunikat..."
```

² Wyjątek stanowią jedynie pliki ignorowane, którymi zajmiemy się w kolejnym rozdziale.

Ćwiczenie 6.1

W folderze *cw-06-01/* utwórz nowe repozytorium. W tym celu wydaj komendę:

```
git init
```

Następnie w folderze *cw-06-01/* utwórz trzy pliki (ich zawartość może być dowolna — mogą to być pliki puste):

```
lorem.txt  
ipsum.txt  
dolor.txt
```

Teraz wydaj komendę:

```
git status
```

Wygenerowana odpowiedź:

```
# Untracked files:  
#   (use "git add <file>..." to include in what will be committed)  
#  
#       dolor.txt  
#       ipsum.txt  
#       lorem.txt  
nothing added to commit but untracked files present (use "git add" to track)
```

zawiera informację, że w folderach znajdują się trzy pliki, które są w stanie *nieśledzone* (ang. *untracked file*).

Polecenie:

```
git status -s
```

generuje wydruk:

```
?? dolor.txt  
?? ipsum.txt  
?? lorem.txt
```

Pliki nieśledzone w skróconej postaci są oznaczane dwoma znakami zapytania ??.

Teraz zmieniamy stan pliku *lorem.txt* z nieśledzonego na zaindeksowany:

```
git add lorem.txt
```

Odpowiedź:

```
# Changes to be committed:  
#   (use "git rm --cached <file>..." to unstage)  
#  
#       new file:  lorem.txt  
#  
# Untracked files:  
#   (use "git add <file>..." to include in what will be committed)  
#  
#       dolor.txt  
#       ipsum.txt
```

mówi o tym, że plik *lorem.txt* jest zaindeksowany (*changes to be committed* — zmiany, które zostaną zatwierdzone).

Postać skrócona informacji generowana poleceniem:

```
git status -s
```

jest następująca:

```
A_ lorem.txt
?? dolor.txt
?? ipsum.txt
```

Nowe pliki po zaindeksowaniu są oznaczane dwuznakowym kodem składającym się z litery A, po której następuje spacja. Dla zwiększenia czytelności na powyższym wydruku spacja została zapisana jako podkreślenie _.



Zwróć uwagę na podpowiedź:

```
# (use "git rm --cached <file>..." to unstage)
```

Git podpowiada, że jeśli chcemy plik usunąć z indeksu, należy wydać polecenie:

```
git rm --cached nazwa-pliku
```

Wykonajmy rewizję:

```
git commit -m "Pierwszy"
```

Spowoduje ona zapamiętanie w repozytorium stanu wyłącznie jednego pliku: *lorem.txt*. Plik *lorem.txt* stanie się po tej operacji aktualny. Komenda `git status -s` zwróci informacje o dwóch niezapisanych plikach, pomijając aktualny plik *lorem.txt*:

```
?? ipsum.txt
?? dolor.txt
```

Dodajmy pozostałe dwa pliki do repozytorium w postaci jednej rewizji:

```
git add -A
git commit -m "Drugi"
```

Teraz komenda `git status` zwróci informację, że wszystkie pliki są aktualne (ang. *nothing to commit* — nic do zatwierdzenia).

Sprawdźmy historię projektu:

```
git log
```

W repozytorium występują dwie rewizje.

Zmodyfikuj plik *lorem.txt*. Wprowadź w nim dowolny tekst, np. *Lorem ipsum*. Teraz polecenie `git status` zwróci informację:

```
# On branch master
# Changes not staged for commit:
# (use "git add <file>..." to update what will be committed)
# (use "git checkout -- <file>..." to discard changes in working directory)
```

```
# modified: lorem.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Plik *lorem.txt* jest teraz niezaindeksowany — nie zostanie więc uwzględniony w kolejnej operacji `git commit`. Skrócona postać polecenia `git status` wygeneruje wydruk:

```
_M lorem.txt
```

Pliki, które były aktualne, lecz zostały zmodyfikowane, stają się plikami niezaindeksowanymi. Ich stan w skróconej postaci jest określany jako `_M` (czyli spacja`M`).

Jeśli teraz wydasz komendę `git commit` (bez parametru `-a`):

```
git commit -m "Trzeci"
```

rewizja nie zostanie wykonana. Ujrzysz jedynie komunikat identyczny jak po wydaniu polecenia `git status`, informujący o tym, że w projekcie nie wprowadzono zmian. W celu zapisania w repozytorium zmodyfikowanego pliku *lorem.txt* należy wydać polecenie `git commit` z parametrem `-a`:

```
git commit -a -m "Trzeci"
```

Ten sam skutek osiągniemy, wyając dwie komendy:

```
git add lorem.txt
git commit -m "Trzeci"
```

lub dwie komendy:

```
git add .
git commit -m "Trzeci"
```

Komenda `git status -s` zwraca pusty wynik i w ten sposób upewnia nas, że wszystkie pliki są zapisane w repozytorium. Wydajmy teraz polecenie:

```
git rm lorem.txt
```

Spowoduje ono usunięcie pliku z obszaru roboczego oraz z repozytorium. Wynikiem komendy `git status` będzie informacja:

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   deleted:  lorem.txt
#
```

Komunikat:

```
deleted:    lorem.txt
```

informuje nas, że plik został usunięty z obszaru roboczego oraz że podczas kolejnej operacji zatwierdzania zostanie usunięty z repozytorium. Ta sama informacja w krócej postaci zwracanej poleceniem `git status -s` przyjmuje postać:

```
D_lorem.txt
```

Oznaczenie stanu pliku po operacji git rm to D_ (czyli Dspacja).

Po wykonaniu polecenia:

```
git commit -m "Czwarty"
```

stan repozytorium bez pliku *lorem.txt* zostanie zatwierdzony.

Na zakończenie, stosując standardowe operacje eksploratora, usuń plik *ipsum.txt*. Jeśli wydasz polecenie:

```
git status -s
```

to przekonasz się, że stan pliku jest oznaczony jako _D (tj. spacjaD):

```
_D lorem.txt
```

W takiej sytuacji operacja:

```
git commit -m "Czwarty"
```

(zwróć uwagę na brak parametru -a) nie powiedzie się. Do zatwierdzenia usuwania pliku standardowymi operacjami plikowymi konieczny jest parametr -a polecenia git commit, wydanie polecenia git rm lub wydanie polecenia git add z parametrem --all. W celu zatwierdzenia zmiany wydaj polecenie:

```
git commit -a -m "Piąty"
```

Ćwiczenie 6.2

W folderze *cw-06-02/* utwórz nowe repozytorium:

```
git init
```

Następnie w folderze *cw-06-02/* utwórz plik:

```
1.txt
```

Stan plików projektu zapamiętaj w repozytorium:

```
git add -A  
git commit -m "Pierwsza"
```

Następnie utwórz folder 2/, a w nim pliki:

```
2/a.txt  
2/b.txt  
2/c.txt
```

Folder 2/ i jego zawartość zapisz w repozytorium:

```
git add -A  
git commit -m "Druga"
```

W identyczny sposób utwórz pliki (możesz skopiować folder 2/ i zmienić jego nazwę):

```
3/a.txt  
3/b.txt  
3/c.txt
```

i wykonaj trzecią rewizję:

```
git add -A
git commit -m "Trzecia"
```

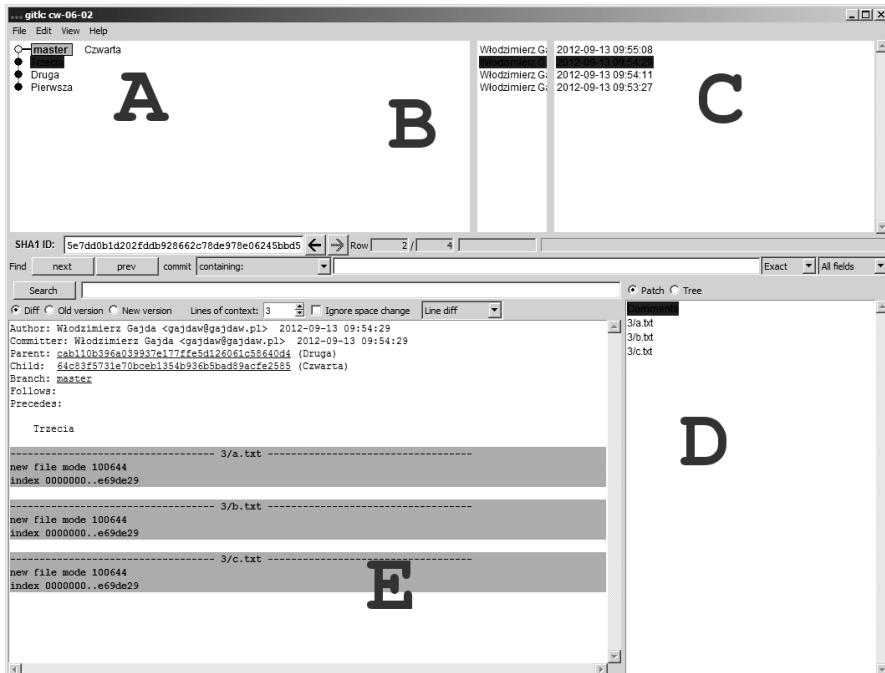
Teraz usuńmy wszystkie pliki *a.txt*, a wykonane zmiany zatwierdzmy:

```
git rm */a.txt
git commit -m "Czwarta"
```

Po wydaniu powyższych komend w repozytorium znajdują się cztery rewizje, o czym przekonasz się, wydając polecenie:

```
git log --pretty=oneline
```

Na zakończenie ćwiczenia uruchom program Git GUI i otwórz w nim repozytorium z ćwiczenia 6.2. Następnie uruchom opcję *Repository/Visualise master's History*. Ujrzyś okno dialogowe przedstawione na rysunku 6.3.



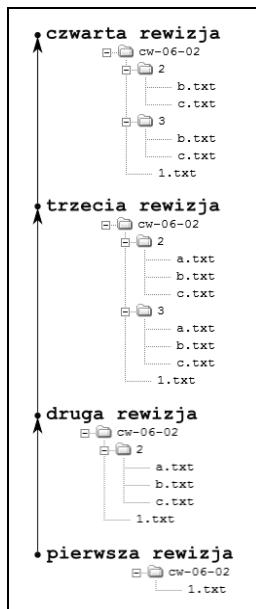
Rysunek 6.3. Wizualne przedstawienie historii projektu z ćwiczenia 6.2 w programie Git GUI

Na rysunku 6.3 literami oznaczono obszary:

- ♦ A — rewizje zawarte w repozytorium;
- ♦ B — autorzy rewizji;
- ♦ C — daty wykonania rewizji;
- ♦ D — lista plików zmodyfikowanych w wybranej rewizji;
- ♦ E — szczegółowe informacje o wykonanych zmianach.

Historia projektu z ćwiczenia 6.2 wraz ze stanem plików jest przedstawiona na rysunku 6.4.

Rysunek 6.4.
Historia projektu
z ćwiczenia 6.2
z uwzględnieniem
stanu repozytorium



Oznaczenia stanów pliku

Stany plików zwracane przez polecenie `git status -s` są oznaczane dwuznakowymi skrótami. Pierwsza litera skrótu dotyczy indeksu, a druga — obszaru roboczego.



Wskazówka

Pliki aktualne nie są oznaczane żadnym kodem. Są one pomijane przez polecenie `git status`.

Nowy plik jest oznaczony skrótem:

`?? dane.txt nowy plik`

Po wydaniu polecenia `git add dane.txt` będzie oznaczony jako `A_`:

`A_ dane.txt`

Po wykonaniu operacji `git commit` i zmodyfikowaniu pliku zostanie oznaczony jako `_M`:

`_M dane.txt`

Po ponownym wydaniu polecenia `git add` stan pliku zmieni się na `M_`:

`M_ dane.txt`

Po usunięciu pliku z obszaru roboczego przy użyciu polecień plikowych (dokładniej mówiąc, bez użycia komendy git rm) jego stan będzie oznaczony jako D:

D dane.txt

Po wydaniu polecenia git rm stan pliku zmieni się na D_:

D_ dane.txt

Po wydaniu polecenia git mv stan pliku będzie oznaczony jako R_:

R_ dane.txt -> info.txt

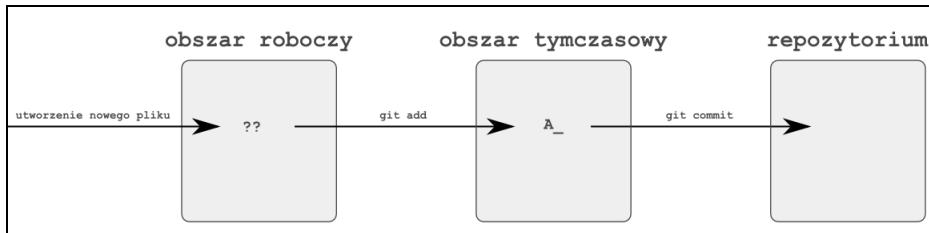
Jeśli przed wykonaniem rewizji zmienimy treść pliku, jego stan zostanie wówczas określony jako RM:

RM dane.txt -> info.txt

W oznaczaniu stanu pliku stosowane są m.in. litery:

- ◆ A — plik dodany (ang. *added*);
- ◆ D — plik usunięty (ang. *deleted*);
- ◆ R — plik o zmienionej nazwie (ang. *renamed*);
- ◆ M — zmodyfikowany (ang. *modified*)

Działanie wszystkich polecen Gita oraz operacji wykonywanych na plikach możemy precyzyjnie opisać, stosując powyższe dwuznakowe symbole. Rysunki 6.5, 6.6 oraz 6.7 ilustrują wpływ wybranych polecień na stan pliku.



Rysunek 6.5. Zmiany stanów nowego pliku

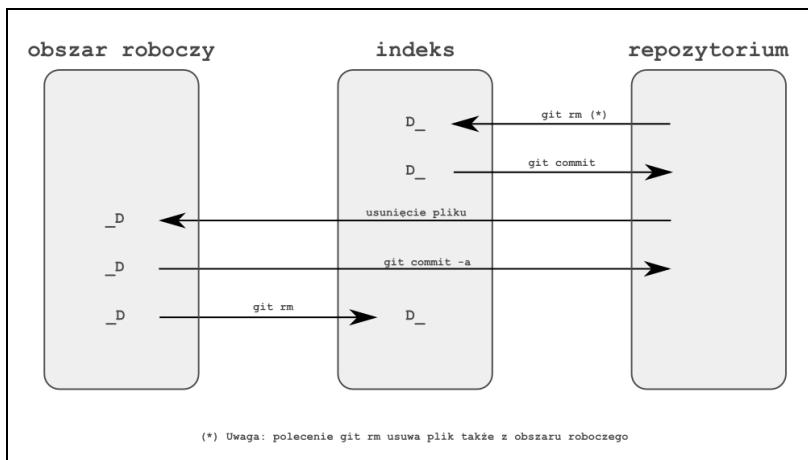
Stany dwuliterowe (mieszane)

Co ciekawe, stan pliku może być oznaczony dwoma literami. Jeśli na przykład utworzymy nowy plik i dodamy go do obszaru tymczasowego, po czym usuniemy:

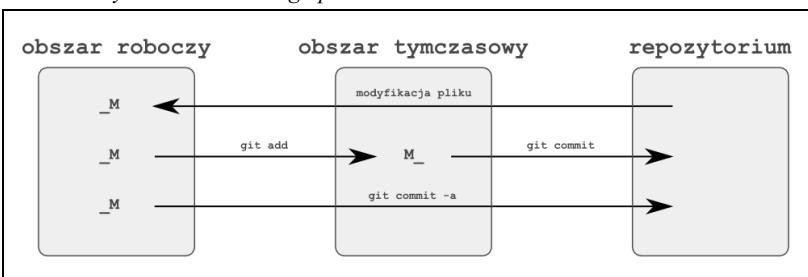
```

echo "Lorem ipsum " > dane.txt
git add dane.txt
git rm dane.txt

```



Rysunek 6.6. Zmiany stanów usuwanego pliku



Rysunek 6.7. Zmiany stanów zmodyfikowanego pliku

to plik będzie oznaczony stanem:

AD dane.txt

Podobnie modyfikacja pliku o stanie M_ zmieni stan pliku na MM.

W codziennej pracy nigdy nie stosuję stanów dwuliterowych.



Polecenie:

`echo "Lorem ipsum" > dane.txt`

tworzy w bieżącym folderze plik *dane.txt* zawierający tekst *Lorem ipsum*.

Ćwiczenie 6.3

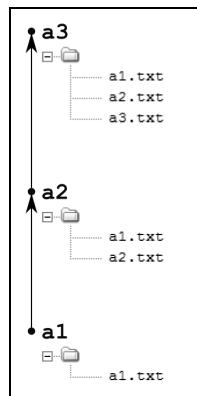
Utwórz repozytorium przedstawione na rysunku 6.8.

Wydaj polecenie:

`git init`

Rysunek 6.8.

Repozytorium
z ćwiczenia 6.3



po czym utwórz plik o nazwie *a1.txt*. W pliku wpisz tekst *a1*. Możesz to wykonać poleciением:

```
echo a1 > a1.txt
```

Następnie wykonaj rewizję oznaczoną komentarzem *a1*:

```
git add -A  
git commit -m a1
```

Zauważ, że jeśli komentarz rewizji nie zawiera spacji, to cudzysłów możesz pominąć. Repozytorium przyjmie stan przedstawiony na rysunku 6.9.

**Rysunek 6.9.** Repozytorium z ćwiczenia 6.3 po wykonaniu rewizji *a1*

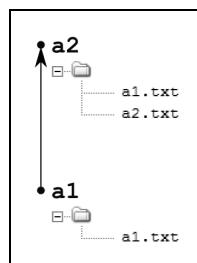
W celu utworzenia drugiej rewizji wydaj polecenia:

```
echo a2 > a2.txt  
git add -A  
git commit -m a2
```

Repozytorium przyjmie stan przedstawiony na rysunku 6.10.

Rysunek 6.10.

Repozytorium
z ćwiczenia 6.3
po wykonaniu
rewizji *a2*



Po wydaniu polecień:

```
echo a3 > a3.txt  
git add -A  
git commit -m a3
```

repozytorium przyjmie postać widoczną na rysunku 6.8.



Repozytoria takie jak w ćwiczeniu 6.3 są bardzo wygodne do nauki operowania gałęziami, gdyż w przypadku łączenia gałęzi nigdy nie powodują kolizji.

Repozytoria zwykłe i surowe

Repozytoria Gita tworzone poleceniami:

```
git init  
git clone
```

składają się z:

- ◆ obszaru roboczego,
- ◆ bazy danych rewizji zawartej w folderze *.git*,
- ◆ indeksu.

Do synchronizacji pracy repozytoriów w grupie programistów potrzebna jest wyłącznie baza danych rewizji. Z tego powodu Git umożliwia tworzenie repozytoriów pozbawionych obszaru roboczego oraz indeksu. Polecenia:

```
git init --bare  
git clone --bare
```

tworzą nowe repozytorium Gita zawierające wyłącznie bazę danych rewizji.

Repozytoria takie nazwiemy **surowymi** (ang. *bare*), natomiast repozytoria zawierające bazę danych rewizji, obszar roboczy i indeks — **repozytoriami zwykłymi**.

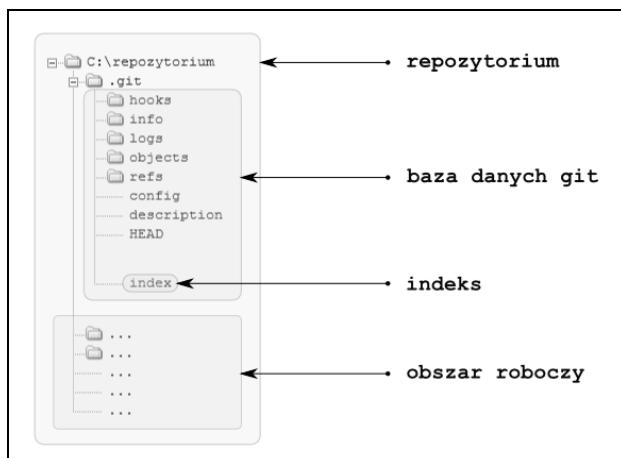
Repozytoria zwykłe są wykorzystywane do codziennej pracy. W nich wydajemy polecenia `git add` oraz `git commit`.

Repozytoria surowe są udostępniane w sieci. Uczestnicy projektu grupowego przesyłają do nich własne rewizje oraz pobierają z nich rewizje kolegów. Zadania te wykonujemy opisanymi w trzeciej części poleceniami `git push` oraz `git pull`.

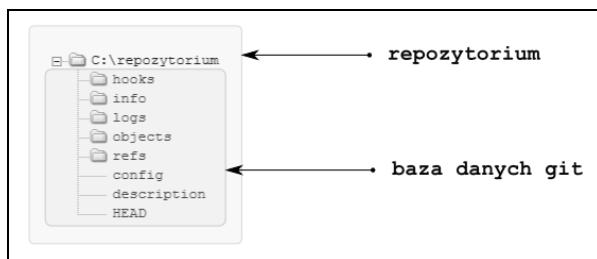
Zawartość folderu repozytorium surowego odpowiada zawartości folderu *.git* repozytorium zwykłego.

Strukturę repozytoriów zwykłego i surowego ilustrują rysunki 6.11 i 6.12.

Rysunek 6.11.
Struktura repozytorium zwykłego



Rysunek 6.12.
Struktura repozytorium surowego



Ćwiczenie 6.4

Utwórz repozytorium surowe i sprawdź zawartość folderu repozytorium.

Wydaj komendę:

```
git init --bare cw-06-04
```

po czym zajrzyj do folderu *cw-06-04/*.



Repozytoria surowe są w wielu źródłach umieszczane w folderach, których nazwa ma rozszerzenie *.git*, np.

```
git init --bare projekt.git
```

Składnia poleceń Gita

Polecenia Gita przyjmują wiele różnych parametrów, po których następują nazwy plików, np.:

```
git add --verbose nazwa-pliku
```

W niektórych przypadkach może to powodować dwuznaczność. W celu zaindeksowania pliku o nazwie `--verbose` należy użyć polecenia:

```
git add -- --verbose
```

Specjalny parametr:

`--`

oddziela opcje polecenia od nazw plików. Wszystkie ciągi występujące po znakach `--` są traktowane jako nazwy plików.

Ćwiczenie 6.5

Utwórz repozytorium, które w kolejnych rewizjach będzie zawierało pliki:

```
--all  
-A  
--update  
-u
```

Każdy plik zaindeksuj poleceniem `git add`, które nie stosuje kropki, gwiazdki ani parametru `--all`.

ROZWIĄZANIE

Po utworzeniu repozytorium:

```
git init
```

utwórz plik o nazwie `--all`:

```
echo "Lorem" > --all
```

W celu zaindeksowania pliku wydaj komendę:

```
git add -- --all
```

W podobny sposób utworzysz i zaindeksujesz pozostałe pliki:

```
echo "Ipsum" > -A  
git add -- -A
```

```
echo "Dolor" > --update  
git add -- --update
```

```
echo "Sit" > -u  
git add -- -u
```

Rozdział 7.

Ignorowanie plików

Czasami zdarzy się, że będziemy chcieli zrezygnować z umieszczania w repozytorium jakiegoś pliku, który znajduje się w przestrzeni roboczej. W praktyce z takim problemem spotykam się w następujących przypadkach:

- ◆ W projektach open source nie mogę umieszczać plików tworzonych przez środowiska deweloperskie PHPStorm i NetBeans.
- ◆ W publikowanych kodach aplikacji internetowych nie chcę umieszczać własnych plików konfiguracyjnych z hasłami do baz danych, serwerów FTP itd.
- ◆ W repozytoriach nie chcę umieszczać kodów binarnych powstających po uruchomieniu kompilatora.

Oczywiście w celu zignorowania nowych plików (tj. plików środowiska deweloperskiego) nie musimy wykonywać żadnej operacji. Git nigdy nie zaczyna śledzić plików, których ręcznie nigdy nie zaindeksowaliśmy. Oznacza to jednak, że do wykonywania rewizji nie moglibyśmy stosować uproszczonego modelu:

```
git add -A  
git commit -m "..."
```

Co więcej, polecenie:

```
git status -s
```

będzie generowało mylące wyniki. Najwygodniej jest pracować w taki sposób, by po wykonaniu rewizji polecenie `git status` zwracało informacje, że wszystkie pliki są aktualne.

Istnieją dwa sposoby wykluczania plików zawartych w obszarze roboczym z repozytorium. Pierwszy z nich polega na wyszczególnieniu wykluczonych plików w specjalnym pliku `.gitignore`. Drugim sposobem jest użycie pliku `.git/info/exclude`. Różnica polega na tym, że plik `.gitignore` będzie zawarty w repozytorium, a plik `exclude` — nie.



Wskazówka

Plik `exclude` tworzymy dla każdego repozytorium osobno. Plik ten nie jest zapisywany w repozytorium.

Plik `.gitignore` możemy utworzyć wewnątrz repozytorium lub globalnie dla wszystkich repozytoriów użytkownika. Plik `.gitignore` tworzony wewnątrz repozytorium jest zapisywany w repozytorium. Globalny plik `.gitignore` nie jest zapisywany w repozytorium.

Jeśli w pliku `.git/info/exclude` umieścimy wpisy widoczne na listingu 7.1, polecenia Gita będą wówczas ignorować wszystkie pliki zawarte w folderach¹ `.idea/`, `nbproject/` oraz `tmp/`.

Listing 7.1. Przykładowa zawartość pliku `.git/info/exclude`

```
/.idea
/nbproject
/tmp
```

Taka technika wykluczania jest wygodna wtedy, gdy musimy dostosować się do kryteriów współpracy panujących w projekcie. Dzięki temu, że informacje o wykluczaniu są zawarte w folderze `.git`, będą one niewidoczne dla pozostałych uczestników projektu. Nie musimy tego z nikim konsultować.

Drugie rozwiązanie jest wygodne, gdy przygotowujemy projekt, który należy — podczas uruchamiania — dostosować do własnych potrzeb. Założymy, że w projekcie występuje plik o nazwie `parameters.ini`, który zawiera ustawienia dostępu do bazy danych. Przykładowa zawartość pliku jest przedstawiona na listingu 7.2.

Listing 7.2. Przykładowy plik konfiguracyjny `config.ini`

```
[parameters]
database_host      =
database_name       =
database_user       =
database_password  =
```

W pliku tym należy wprowadzić dane dostępu do bazy danych. Nie możemy wprowadzić tych danych i zapisać pliku w kolejnej rewizji, gdyż po upublicznieniu projektu każdy będzie mieć hasło dostępu do naszej bazy danych. Nie możemy tego pliku po prostu usunąć, gdyż po pierwsze, projekt przestanie nam działać, a po drugie, skąd osoba pobierająca kod projektu ma wiedzieć, jaki plik konfiguracyjny, w którym folderze i o jakiej strukturze utworzyć?

Z tego powodu należy postąpić w następujący sposób: najpierw zmieniamy nazwę pliku konfiguracyjnego na `properties.ini.dist`. Rozszerzenie `.dist` oznacza, że plik ten należy dostosować do własnych potrzeb. Plik `properties.ini.dist` zapisujemy w repozytorium w standardowy sposób, np. poleceniami:

¹ Folder `.idea/` jest tworzony przez środowisko PHPStorm, a folder `nbproject/` — przez środowisko NetBeans.

```
git add -A  
git commit -m "..."
```

Następnie plik *properties.ini.dist* kopujemy. Skopiowany plik nazywamy *properties.ini*. W pliku tym umieszczamy dane dostępu do serwera SQL. Następnie w folderze głównym repozytorium tworzymy plik *.gitignore* i dodajemy w nim wpis widoczny na liście 7.3.

Listing 7.3. Przykładowa reguła w pliku .gitignore

```
/sciezka/do/pliku/properties.ini
```



Wskazówka

Reguły zawarte w plikach *.gitignore* oraz *exclude* wskazują pliki z folderu projektu. Początkowy znak / jest opcjonalny. Obie poniższe reguły mają identyczne działanie:

```
/jakis/folder/plik.txt  
jakis/folder/plik.txt
```

Po utworzeniu obu plików: *properties.ini* oraz *.gitignore* rewizje możemy wykonywać w uproszczony sposób:

```
git add -A  
git commit -m "..."
```

Polecenie git status -s zwróci pusty wynik. Wszystkie pliki w repozytorium są zatem aktualne. Repozytorium będzie zawierało plik *properties.ini.dist*, a nie będzie zawierało pliku *properties.ini*. W celu uruchomienia sklonowanego projektu należy dostosować do własnych potrzeb wszystkie pliki o rozszerzeniu *.dist*.

Oczywiście w plikach *.gitignore* oraz *.git/info/exclude* możemy posługiwać się metaznakami * oraz ?.

W plikach *.gitignore* umieszcza się zwykle reguły ignorujące pliki tymczasowe oraz pliki powstałe po komplikacji, np.:

```
*.[oa]  
*~  
*.exe
```

Zwróć uwagę, że zawartość pliku *.gitignore* będzie zapisywana w repozytorium (to jest zwykły plik w obszarze roboczym). Dzięki temu wszyscy uczestnicy projektu będą dysponowali podanymi regułami. Reguła wykluczająca plik *properties.ini* będzie potrzebna każdemu uczestnikowi, który zechce uruchomić projekt i chronić hasło dostępu do własnej bazy danych.

Jeśli identyczne reguły chcemy zastosować we wszystkich repozytoriach, należy użyć globalnego pliku *.gitignore*. Najpierw tworzymy (w dowolnym folderze na dysku) plik *.gitignore*, np.:

```
C:\git\konfiguracja\.gitignore
```

Następnie wydajemy polecenie:

```
git config --global core.excludesfile C:\git\konfiguracja\.gitignore
```

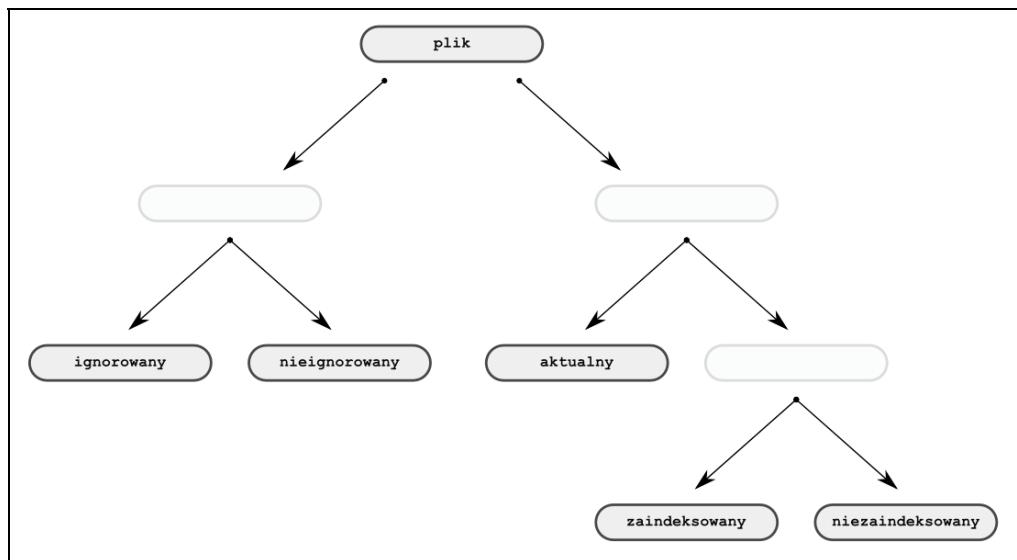
Polecenie to doda następującą regułę konfiguracyjną w globalnym pliku `.gitconfig`:

```
[core]  
excludesfile = C:\\git\\konfiguracja\\\\.gitignore
```

Od tej pory we wszystkich projektach ignorowane będą pliki wymienione w pliku `C:\git\konfiguracja\.gitignore`.

Uzupełnienie diagramu stanów

Kompletny diagram stanów plików w repozytorium Gita jest przedstawiony na rysunku 7.1.



Rysunek 7.1. Kompletny diagram stanów plików

Pliki ignorowane to pliki zawarte w przestrzeni roboczej, które są objęte przez reguły zawarte w pliku `.gitignore` lub `.git/info/exclude`. Pliki takie nie są śledzone i Git nie zgłasza nigdy komunikatów o zmianach ich treści.

Pliki nieignorowane to pliki, które występują w przestrzeni roboczej, lecz nie występują w repozytorium ani w indeksie. Pliki te nie pasują do reguł zawartych w plikach `.gitignore` ani `.git/info/exclude`. Polecenie `git status` uwzględnia pliki nieignorowane i oznacza je dwoma znakami zapytania ??.



Wskazówka

Jeśli projekt zawiera ignorowane pliki, to nie można usuwać zawartości obszaru roboczego. Jeśli usuniesz zawartość całego obszaru roboczego, to polecenie:
git reset --hard
nie przywróci stanu plików ignorowanych!

Ćwiczenie 7.1

W folderze *cw-07-01/* utwórz nowe repozytorium:

```
git init
```

Następnie utwórz pliki:

```
config/user.cfg  
config.ini
```

Teraz wydaj komendę:

```
git status -s
```

Wygenerowana odpowiedź będzie zawierała informację, że plik *config.ini* oraz folder *config/* są nieśledzone²:

```
?? config.ini  
?? config/
```

W pliku *.git/info/exclude* dodaj na końcu wiersz zawierający regułę:

```
/config
```

Komenda:

```
git status -s
```

zwróci teraz informację, że tylko jeden plik nie jest śledzony:

```
?? config.ini
```

Następnie w folderze projektu utwórz plik *.gitignore* i wydaj polecenie:

```
git status -s
```

Teraz dwa pliki nie są śledzone:

```
?? .gitignore  
?? config.ini
```

Po wprowadzeniu w pliku *.gitignore* reguły:

```
config.ini
```

² Dokładniej mówiąc, nieśledzone i nieignorowane.

polecenie:

```
git status -s
```

zwróci informację, że tylko plik *.gitignore* nie jest śledzony:

```
?? .gitignore
```

Zatwierdzmy wykonane zmiany:

```
git add -A  
git commit -m "..."
```

Teraz wszystkie śledzone pliki są aktualne. Zmodyfikuj zawartość plików:

```
config/user.cfg  
config.ini
```

Pomimo tego, że pliki zostały zmodyfikowane, polecenie:

```
git status -s
```

zwróci pusty wynik.



Wskazówka

Plik *.gitignore* zapisujemy w repozytorium. W ten sposób wszyscy uczestnicy projektu będą mogli wykorzystać zawarte w nim reguły.

Ćwiczenie 7.2

Wykonaj ćwiczenie demonstrujące użycie plików o rozszerzeniu *.dist*.

ROZWIĄZANIE

W folderze *cw-07-02/* utwórz nowe repozytorium:

```
git init
```

Następnie w folderze *cw-07-02/* utwórz plik *README*, po czym wykonaj pierwszą rewizję.

Załóżmy, że projekt ma zawierać plik *db.ini*, definiujący parametry połączenia z bazą danych. Utwórz nowy plik *db.ini.dist* zawierający fałszywe informacje dostępu do bazy danych, np.:

```
[parameters]  
database_host      = your.host.example.net  
database_name       = dbname  
database_user       = admin  
database_password   = sEcREtPaSSword
```

Wykonaj rewizję, która będzie zawierała plik *db.ini.dist*.

Następnie utwórz plik *.gitignore* o zawartości:

```
db.ini
```

po czym wykonaj rewizję. Repozytorium zawiera teraz pliki:

```
README  
db.ini.dist  
.gitignore
```

W celu uruchomienia projektu nowy użytkownik musi najpierw sklonować nasz projekt³:

```
git clone cw-07-02 .
```

a następnie zmienić nazwę pliku *db.ini.dist* na *db.ini*. W pliku *db.ini* wprowadzamy dane dostępu do własnej bazy danych, np.:

```
[parameters]  
database_host      = sql.gajdaw.pl  
database_name      = ksiazka_git  
database_user       = gajdaw  
database_password  = k_xdu3jd7?ki+d
```

Dzięki temu, że plik *db.ini* jest wymieniony w pliku *.gitignore*, dane dostępu do bazy danych nie będą nigdy dostępne w repozytorium.

Ćwiczenie 7.3

Zainstaluj środowisko NetBeans i wykorzystaj je do wprowadzania zmian w repozytorium Symfony 2.

ROZWIĄZANIE

Zainstaluj środowisko NetBeans. Następnie sklonuj projekt Symfony 2:

```
git clone https://github.com/symfony/symfony .
```

Uruchom środowisko NetBeans, po czym zainicjalizuj nowy projekt NetBeans, wskażając folder sklonowanego repozytorium Symfony 2. Po wyłączeniu środowiska NetBeans wydaj komendę:

```
git status
```

Okaże się, że środowisko NetBeans utworzyło w folderze projektu folder *nbproject/*. Folderu tego nie możesz dołączyć do projektu (autorzy projektu nie zaakceptują żadnych rewizji zawierających takie zmiany). W celu wykluczenia folderu *nbproject/* w pliku *.git/info/exclude* wprowadź regułę:

```
/nbproject
```

Teraz komenda:

```
git status
```

zwróci pusty wiersz.

³ Metody publikowania repozytorium tak, by było ono dostępne do sklonowania przez innych użytkowników, zostaną omówione w piątej części podręcznika.



Wskazówka

W ćwiczeniu 7.3 reguła powodująca ignorowanie folderu `/nbproject` nie może być zapisana w pliku `.gitignore`. Autorzy projektu nie zaakceptują takich zmian w pliku `.gitignore`, który jest przecież zapisywany w repozytorium. Możemy wykorzystać globalny plik `.gitignore` dotyczący wszystkich repozytoriów. Wadą takiego rozwiązania jest to, że obejmie ono także nasze prywatne repozytoria, a nie tylko dany projekt open source. W swoich projektach zawsze zapisuję pliki tworzone przez środowisko programistyczne. Z tego powodu najlepszym rozwiązaniem tego problemu wydaje mi się użycie pliku `exclude`.



Wskazówka

Kolekcję przydatnych szablonów plików `.gitignore` znajdziesz w repozytorium:

<https://github.com/github/gitignore>

Rozdział 8.

Znaczniki

Powszechnie przyjętym rozwiązaniem oznaczania wersji programów komputerowych jest nadawanie etykiet postaci 3.2 czy 1.7.2. Na co dzień możemy spotkać się ze zwrotami „jądro Linuksa w wersji 3.2” czy „jQuery w wersji 1.7.2”. Oprogramowanie Git umożliwia oznaczanie rewizji przy użyciu **znaczników** (ang. *tags*). Dzięki temu zyskujemy naturalny sposób wiążenia wersji programu z historią projektu prowadzonego w programie Git.

Znaczniki lekkie i oznaczone

W środowisku Git dostępne są dwa rodzaje znaczników: **znaczniki opisane** (ang. *annotated tags*) i **znaczniki lekkie** (ang. *lightweight tags*).

Znaczniki opisane zawierają:

- ◆ dane autora,
- ◆ datę utworzenia,
- ◆ komentarz,
- ◆ skrót SHA-1 rewizji, którą wskazują.

Znaczniki lekkie zawierają wyłącznie skrót SHA-1 rewizji.

Zalecanym rodzajem znaczników są znaczniki opisane.



Znaczniki opisane są obiektami zapisywanymi w bazie danych `.git` w postaci rewizji. Znaczniki lekkie są zapisywane w plikach tekstowych w folderze `.git/refs/tags`.

Tworzenie znaczników opisanych

Do tworzenia znaczników opisanych służy komenda:

```
git tag -a NAZWA -m KOMENTARZ
```

np.

```
git tag -a v1.2.3 -m "Wydanie ver. 1.2.3"
```

Po wydaniu powyższej komendy utworzony zostanie znacznik v1.2.3 wskazujący ostatnią rewizję w projekcie. Dodając na końcu polecenia skrót SHA-1:

```
git tag -a NAZWA -m KOMENTARZ SHA-1
```

np.

```
git tag -a v4.5.6 -m "Wydanie ver. 4.5.6" aabbccdd
```

możemy utworzyć znacznik opisany wskazujący dowolną rewizję w projekcie.

Oczywiście znacznik musi być unikatowy. Drugie z poleceń:

```
git tag -a v9.0 -m "..."  
git tag -a v9.0 -m "..."
```

zakończy się błędem. Komunikat poinformuje nas, że znacznik v9.0 już istnieje.

Tworzenie znaczników lekkich

Do tworzenia znaczników lekkich służy komenda `git tag` bez parametrów `-a` oraz `-m`:

```
git tag NAZWA
```

np.

```
git tag v7.7
```

Po wydaniu powyższej komendy utworzony zostanie lekki znacznik v7.7 wskazujący ostatnią rewizję w projekcie. Dodając na końcu polecenia skrót SHA-1:

```
git tag NAZWA SHA-1
```

np.

```
git tag v3.3 ffeedd11
```

możemy utworzyć znacznik lekki wskazujący dowolną rewizję w projekcie.

Znaczniki lekkie również muszą być unikatowe.

Usuwanie znaczników

Do usuwania znaczników służy komenda:

```
git tag -d NAZWA
```

Komendą tą usuwamy zarówno znaczniki lekkie, jak i oznaczone.

Sprawdzanie dostępnych znaczników

Polecenie:

```
git tag
```

wyświetla listę wszystkich znaczników, zarówno lekkich, jak i oznaczonych.

Do sprawdzenia wszystkich znaczników (lekkich oraz oznaczonych) wraz z datami utworzenia należy użyć komendy:

```
git log --tags --simplify-by-decoration --pretty="format:%ai %d"
```

Jeśli pracujesz w konsoli bash, to polecenie:

```
git log --tags --simplify-by-decoration --pretty="format:%ai %d" | sort
```

wyświetli listę znaczników posortowaną według dat.

Do wyszukania ostatniego (tj. dotyczącego najmłodszej możliwej rewizji) znacznika służy komenda:

```
git describe
```

Komenda ta wywołana bez parametru dotyczy wyłącznie znaczników opisanych. Jeśli dodamy parametr --tags:

```
git describe --tags
```

komenda będzie wówczas dotyczyć zarówno znaczników opisanych, jak i lekkich.

Szczegółowe dane znacznika

Szczegółowe dane o znaczniku możemy wyświetlić komendą `git show`¹:

```
git show -s v2.0.10
```

W przypadku znacznika opisanego wydruk będzie zawierał informacje zarówno o znaczniku, jak i o rewizji, np.:

¹ Parametr `-s` skraca postać wydruku generowanego przez polecenie `git show`.

```
tag v2.0.10
Tagger: Fabien Potencier <fabien.potencier@gmail.com>
Date: Mon Feb 6 10:49:20 2012 +0100
```

```
created tag 2.0.10
```

```
commit 7f97b60d50031e4f1beb9ala10c32dac816584cf
Author: Fabien Potencier <fabien.potencier@gmail.com>
Date: Mon Feb 6 10:49:11 2012 +0100
```

```
update CONTRIBUTORS for 2.0.10
```

Jeśli polecenia git show użyjemy w stosunku do znacznika lekkiego:

```
git show -s 1.4
```

ujrzymy informacje o rewizji:

```
commit d431519d61f55af8bd1714d00126c948d888f3a9
Author: jeresig <jeresig@gmail.com>
Date: Wed Jan 13 15:23:05 2010 -0500
```

```
We only care that some of the html return value is escaped...
```



Niektóre projekty open source (np. Symfony 2) stosują konwencję nazewniczą vX.Y.Z, np.:

```
v2.0.10
v2.0.11
```

W innych (np. jQuery) pomijana jest początkowa litera v:

```
1.4
1.4.1
1.4.2
```

Użycie znaczników

Znacznik może zostać użyty wszędzie tam, gdzie występuje skrót SHA-1, czyli na przykład w poleceniach:

```
git checkout SHA-1
git reset --hard SHA-1
```

Polecenie:

```
git checkout -f v1.2.3
```

przywróci pliki w obszarze roboczym do stanu z rewizji oznaczonej znacznikiem v1.2.3.

Komenda:

```
git reset --hard v4.5.6
```

przywróci stan obszaru roboczego oraz usunie z projektu wszystkie rewizje występujące po znaczniku v4.5.6.

Ćwiczenie 8.1

Sklonuj repozytorium:

```
https://github.com/jquery/jquery
```

po czym sprawdź dostępne znaczniki:

```
git tag
```

Następnie sprawdź dostępne znaczniki wraz z datami utworzenia:

```
git log --tags --simplify-by-decoration --pretty="format:%ai %d"
```

Pliki w obszarze roboczym przywróć do stanu odpowiadającego wersji 1.3. W tym celu wydaj komendę:

```
git checkout 1.3
```

Polecenie:

```
git describe
```

upewni Cię, że wszystkie znaczniki w projekcie jQuery są znacznikami lekkimi. Poleceniem:

```
git show -s 1.7
```

sprawdź szczegóły znacznika 1.7.

W celu przywrócenia plików obszaru roboczego do wersji 1.3 wydaj polecenie:

```
git checkout -f 1.3
```

Ćwiczenie 8.2

W projekcie Symfony:

```
https://github.com/symfony/symfony
```

komendą:

```
git describe
```

sprawdź najnowsze dostępne wydanie. Przyjmując, że jest to wydanie oznaczone znacznikiem v2.0.15, wydaj komendę:

```
git show v2.0.15
```

W ten sposób przekonasz się, jakie modyfikacje zostały wprowadzone w rewizji, do której odnosi się etykieta v2.0.15.

Ćwiczenie 8.3

Utwórz repozytorium, które będzie zawierało trzy rewizje. Pierwsza rewizja ma zawsze zawierać plik *1.txt*, druga — plik *2.txt*, a trzecia — plik *3.txt*. Kolejne rewizje oznacz znacznikami opisanymi: v0.0.1, v0.0.2 i v0.0.3.

ROZWIĄZANIE

Utwórz folder *cw-08-03/* i zainicjalizuj nowe repozytorium:

```
mkdir cw-08-03  
cd cw-08-03  
git init
```

Następnie utwórz plik *1.txt* i wykonaj rewizję:

```
git add -A  
git commit -m "Pierwsza rewizja"
```

W podobny sposób utwórz kolejne dwie rewizje. Polecenie:

```
git log --pretty=oneline --abbrev-commit
```

wygeneruje informacje o rewizjach:

```
8e436d9 Trzecia rewizja  
4be69af Druga rewizja  
916e627 Pierwsza rewizja
```

W celu oznaczenia pierwszej rewizji znacznikiem v0.0.1 wydaj komendę:

```
git tag -a v0.0.1 -m "Wydanie ver. 0.0.1" 916e627
```

W podobny sposób utwórz dwa kolejne znaczniki:

```
git tag -a v0.0.2 -m "Wydanie ver. 0.0.2" 4be69af  
git tag -a v0.0.3 -m "Wydanie ver. 0.0.3" 8e436d9
```

Polecenie:

```
git describe
```

wyświetli trzy dostępne znaczniki.

W celu przywrócenia stanu plików do wersji v0.0.1 wydaj komendę:

```
git checkout -f v0.0.1
```

Teraz obszar roboczy zawiera wyłącznie jeden plik *1.txt*. Repozytorium znajduje się w stanie detached HEAD. W celu przywrócenia stanu z ostatniej rewizji wydaj komendę:

```
git checkout master
```

Generowanie skompresowanych plików odpowiadających konkretnej wersji projektu

Polecenie `git archive` umożliwia wygenerowanie skompresowanego pliku zawierającego pliki projektu odpowiadające konkretnej rewizji. Dzięki temu, że do oznaczania rewizji możemy użyć znaczników, zyskujemy wygodną metodę pobierania pełnego projektu w konkretnej wersji.

Polecenie:

```
git archive --format=zip --output=NAZWA-PLIKU SHA-1
```

zapisuje plik `.zip` z zawartością repozytorium w stanie z podanej rewizji. Po wydaniu komendy:

```
git archive --format=zip --output=../projekt-0.1.2.zip v0.1.2
```

otrzymamy plik:

```
projekt-0.1.2.zip
```

zawierający całe repozytorium w stanie oznaczonym znacznikiem `v0.1.2`. Pominięcie parametru `--output`:

```
git archive --format=zip v0.1.2
```

spowoduje wysłanie treści archiwum zip na standardowy strumień wyjściowy stout. Polecenia:

```
git archive --format=zip v0.1.2 > ../project-0.1.2.zip
```

```
git archive --format=zip --output=../projekt-0.1.2.zip v0.1.2
```

są równoważne.



Czy zdarzyło Ci się kiedykolwiek zapisywać spakowany projekt w pliku, którego nazwa ma postać `projekt_x_y_z.zip` i zawiera konkretną wersję projektu? Na przykład:

`NotH_0_17.zip`

`NotH_0_29.zip`

`NotH_1_2.zip`

Jeśli pracujesz w programie Git, wystarczy utworzyć znaczniki wskazujące odpowiednią rewizję. W ten sposób w jednym miejscu będziesz mieć dostęp do wszystkich wersji! W celu uzyskania dostępu do konkretnej wersji wystarczy użyć polecień `git checkout` oraz `git archive`.

Rozdział 9.

Identyfikowanie rewizji

Rewizje w repozytorium możemy identyfikować, stosując:

- ◆ pełne skróty SHA-1,
- ◆ zminimalizowane skróty SHA-1,
- ◆ znaczniki,
- ◆ odwołanie symboliczne HEAD,
- ◆ odwołania do n -tego przodka (\sim),
- ◆ odwołania do n -tego rodzica (\wedge),
- ◆ nazwy gałęzi¹,
- ◆ dziennik odwołań reflog.



Pełny opis metod identyfikacji rewizji znajdziesz w dokumentacji, wydając polecenie:
`git revisions --help`

Pełne skróty SHA-1

Identyfikatorami rewizji są czterdziestoznakowe skróty SHA-1. Polecenie:

```
git log --pretty=oneline
```

generuje wydruk prezentujący w każdym wierszu jedną rewizję. Najpierw drukowany jest skrót SHA-1 rewizji, a następnie jej komunikat, np.:

```
99e6e9fa28eace803940deab3ba0c9b4474575c2 Czwarta rewizja  
6528eb2822651269c5cf33a5bb6b77695d966d5b Trzecia rewizja
```

¹ Gałęzie zostaną omówione w drugiej części podręcznika.

```
98c6e84c6e65f64f6485805ed9035baa8f2221c0 Druga rewizja  
f5a0b9364371db097c2ce87c1d429f6a4837559a Pierwsza rewizja
```

Oczywiście każda z powyższych rewizji jest jednoznacznie identyfikowana przez skrót SHA-1. W poleceniach, które zawierają jako parametr identyfikator rewizji, zawsze możemy użyć pełnego skrótu SHA-1. Na przykład polecenie:

```
git checkout -f 98c6e84c6e65f64f6485805ed9035baa8f2221c0
```

przywróci pliki z obszaru roboczego do rewizji oznaczonej podanym skrótem SHA-1.

Skrócona postać SHA-1

Skróty SHA-1 możemy zapisywać w skróconej postaci, podając tylko początkowe znaki. Minimalna długość skróconego SHA-1 to cztery znaki. Warunkiem użycia skróconego zapisu SHA-1 jest jego unikatowość. W przypadku gdy skrót nie jest unikatowy, należy użyć większej liczby znaków.

Polecenie:

```
git log --abbrev-commit --abbrev=4 --pretty=oneline
```

wygeneruje najkrótsze możliwe skróty SHA-1:

```
99e6 Czwarta rewizja  
6528 Trzecia rewizja  
98c6 Druga rewizja  
f5a0 Pierwsza rewizja
```

Wszystkie trzy poniższe polecenia są synonimami²:

```
git checkout -f 98c6e84c6e65f64f6485805ed9035baa8f2221c0  
git checkout -f 98c6e84  
git checkout -f 98c6
```

Znaczniki

Poznane w poprzednim rozdziale znaczniki również służą do identyfikowania rewizji. Jeśli rewizje:

```
f5a0 Pierwsza rewizja
```

oznaczymy znacznikiem opisanyem v0.1.2:

```
git tag -a v0.1.2 -m "Wydanie ver. 0.1.2"
```

wówczas polecenia:

² Zakładając, że w projekcie nie występuje żadna inna rewizja, której SHA-1 rozpoczyna się od znaków 98c6.

```
git checkout -f v0.1.2  
git checkout -f 98c6
```

są synonymami.

Nazwa symboliczna HEAD

Kolejną metodą identyfikowania rewizji jest użycie nazwy symbolicznej HEAD. Wskazuje ona ostatnią rewizję w bieżącej gałęzi³. Jeśli więc polecenie git log zwraca wydruk:

```
99e6 Czwarta rewizja  
6528 Trzecia rewizja  
98c6 Druga rewizja  
f5a0 Pierwsza rewizja
```

odwołanie HEAD dotyczy wówczas rewizji oznaczonej skrótem 99e6. W takim przypadku polecenia:

```
git checkout -f 99e6  
git checkout -f HEAD
```

są synonymami.



Wskazówka Identyfikator rewizji wskazywanej przez nazwę symboliczną HEAD jest zapisany w pliku `.git/HEAD`. Ponieważ w systemie Windows wielkość liter w nazwach plików nie jest rozróżniana, nazwę HEAD możemy zapisywać małymi literami. W systemach rozróżniających wielkość znaków w nazwach plików nazwę HEAD należy zapisywać dużymi literami.

Rewizja domyślna

Jeśli w poleceniu pominiemy identyfikator rewizji, użyta zostanie wartość HEAD. Jeśli polecenie git log zwraca wydruk:

```
99e6 Czwarta rewizja  
6528 Trzecia rewizja  
98c6 Druga rewizja  
f5a0 Pierwsza rewizja
```

wówczas komendy:

```
git checkout -f 99e6  
git checkout -f HEAD  
git checkout -f
```

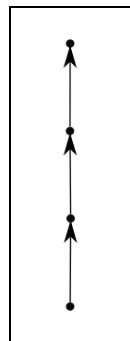
są synonymami.

³ Dodatkowe informacje o nazwie symbolicznej HEAD znajdziesz w rozdziale 13.

Repozytoria o historii nieliniowej⁴

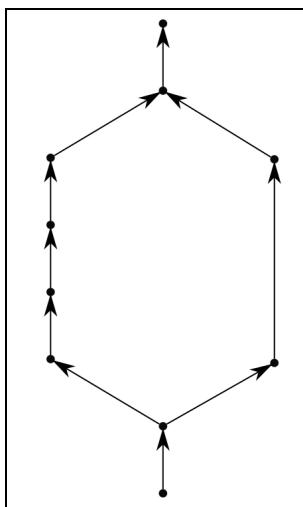
Tworzone przez nas repozytoria miały do tej pory zawsze strukturę liniową. Innymi słowy każda rewizja miała dokładnie jednego rodzica. Przykład repozytorium o takiej historii jest przedstawiony na rysunku 9.1.

Rysunek 9.1.
Repozytorium
o liniowej historii



Gdy przejdziemy do omawiania gałęzi, nauczymy się wówczas tworzyć repozytoria, których historia nie będzie liniowa. W takim repozytorium wystąpią rozgałęzienia. W ogólnym przypadku repozytorium Gita ma strukturę **acyklicznego grafu skierowanego**⁵. Historia projektu z rozgałęzieniami jest przedstawiona na rysunku 9.2.

Rysunek 9.2.
Historia projektu
zawierającego
rozgałęzienia



⁴ W tej części zajmujemy się wyłącznie repozytoriami o liniowej historii. Wołalem jednak w bieżącym rozdziale umieścić wszystkie możliwe metody identyfikowania rewizji. Rozdział ten możesz traktować jako kompletny przewodnik po metodach identyfikowania rewizji. Tworzeniem repozytorium o nieliniowej historii zajmiemy się w części drugiej podręcznika.

⁵ Por. <http://eagain.net/articles/git-for-computer-scientists/>, http://pl.wikipedia.org/wiki/Skierowany_graf_acykliczny.

W obu przypadkach dla każdej rewizji (tj. dla każdego węzła grafu) możemy mówić o rodzicach (ang. *parent revisions*, *parent nodes*).

Git pozwala na identyfikowanie rodziców przy użyciu specjalnej notacji:

X~n

X^n

W powyższym zapisie znak X oznacza identyfikator rewizji (np. skrót SHA-1), a n jest liczbą naturalną wskazującą, o którego rodzica chodzi. Znak ~ wskazuje przodka n-tej generacji, a znak ^ wskazuje n-tego rodzica.



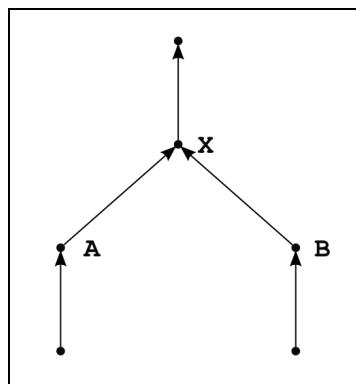
W kodzie źródłowym Linuksa występują rewizje, które mają trzydziestu rodziców!

Kolejność rodziców rewizji

Który z rodziców jest pierwszy, a który drugi? O tym decyduje metoda tworzenia rewizji będącej połączeniem. Przyjmijmy, że w historii projektu występują przedstawione na rysunku 9.3 rewizje A, B i X oraz że polecenie `git log` zwraca wydruk przedstawiony na listingu 9.1.

Rysunek 9.3.

Rewizja mająca kilku rodziców



Listing 9.1. Wynik działania polecenia git log dla repozytorium z rysunku 9.3

X Rewizja powstała przez połączenie rewizji A i B

B Drugi rodzic rewizji X

A Pierwszy rodzic rewizji X

W takim przypadku pierwszym rodzicem rewizji X jest rewizja A, a drugim rodzicem jest rewizja B.

Innymi słowy pierwszym rodzicem rewizji X jest najniższa rewizja na wydruku polecenia `git log`.

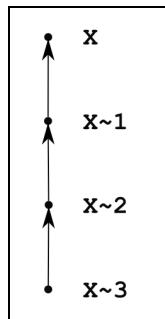


Pierwszym rodzicem rewizji jest rodzic, który na wydruku polecenia git log pojawia się najniżej.

Przodek n-tej generacji

Działanie odwołań do przodków n-tej generacji w projekcie, którego historia jest liniowa, jest przedstawione na rysunku 9.4.

Rysunek 9.4.
Odwołania do
przodków n-tej
generacji w projekcie
o liniowej historii



Liczba podawana po tyldzie określa liczbę kroków, o jaką należy się cofnąć w historii projektu. W ten sposób odwołanie $x\sim 1$ wskazuje rodzica rewizji x , odwołanie $x\sim 2$ — dziadka, odwołanie $x\sim 3$ — pradziadka itd.

Przed znakiem tyldy możemy użyć dowolnego identyfikatora. Może to być pełny kod SHA-1, skrót SHA-1, nazwa symboliczna HEAD lub znacznik:

```
99e6e9fa28eace803940deab3ba0c9b4474575c2~  
99e6~3  
HEAD~4  
v0.1.2~5
```

Liczba 1 podawana po tyldzie może być pominięta. Odwołania:

$x\sim 1$

oraz:

$x\sim$

są równoważne.

Tilda w odwołaniu może wystąpić wielokrotnie. Zapis:

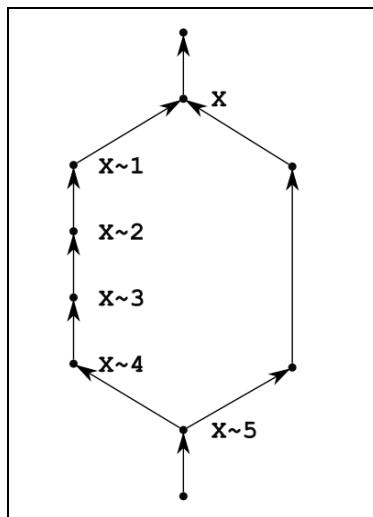
$x\sim\sim$

oznacza dziadka (tj. rodzica rodzica) rewizji x , a zatem jest równoważny zapisowi:

$x\sim 2$

W przypadku węzłów, które mają kilku rodziców, odwołanie zawierające tyldę dotyczy zawsze pierwszego rodzica. Działanie odwołań $x\sim 1$, $x\sim 2$, $x\sim 3$, $x\sim 4$ oraz $x\sim 5$ dla węzła, który ma dwóch rodziców, jest przedstawione na rysunku 9.5.

Rysunek 9.5.
Odwołania do
przodków n -tej
generacji rewizji X



n-ty rodzic

Do oznaczenia kolejnych rodziców danej rewizji służy znak \wedge . Odwołanie:

X^1

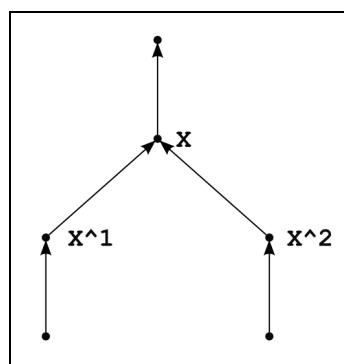
wskazuje pierwszego rodzica rewizji X , zaś odwołanie:

X^2

wskazuje drugiego rodzica.

Przykładowe odwołania do pierwszego i drugiego rodzica rewizji są zilustrowane na rysunku 9.6.

Rysunek 9.6.
Odwołania do
pierwszego i drugiego
rodzica rewizji



Podobnie jak w przypadku odwołań wykorzystujących tylde, tak i teraz możemy używać skrótów SHA-1, nazwy symbolicznej HEAD oraz znaczników. Poprawnymi odwołaniami są:

- 99e6^2 — drugi rodzic rewizji 99e6
- HEAD^2 — drugi rodzic bieżącej rewizji
- v1.2.3^2 — drugi rodzic rewizji oznaczonej znacznikiem v1.2.3

Jeśli odwołujemy się do pierwszego rodzica, liczba 1 może wówczas zostać pominięta. Odwołania:

$X^$

oraz:

X^1

są równoważne.

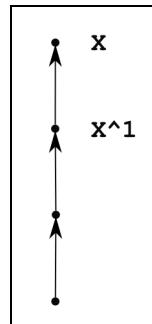
Podobnie jak tylde, tak i znak \wedge może wystąpić wielokrotnie. Zapis:

X^{2^2}

wskazuje drugiego rodzica rewizji X.

Oczywiście odwołanie do pierwszego rodzica pozostaje poprawne także dla węzłów, które mają tylko jednego rodzica. Odwołanie takie ilustruje rysunek 9.7.

Rysunek 9.7.
Odwołanie do
pierwszego rodzica
dla rewizji, która ma
tylko jednego rodzica,
jest poprawne



Wskaźówka

Wszystkie cztery odwołania:

X^1
 $X^$
 X_{-1}
 X_{-}

są równoważne i wskazują pierwszego rodzica rewizji X.

Łączenie odwołań zawierających znaki ~ oraz ^

Odwołania zawierające tylde i znak \wedge możemy łączyć. Identyfikator:

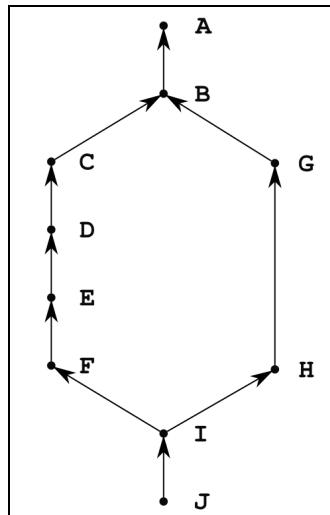
$X_{-1}^{1^2}$

wskazuje drugiego rodzica (${}^{(2)}$) rewizji nadzędnej (tj. pierwszego rodzica) rewizji X.

Przykłady odwołań zawierających znaki \sim i \wedge

Przyjmijmy, że projekt zawiera historię przedstawioną na rysunku 9.8, przy czym pierwszym rodzicem rewizji B jest rewizja C.

Rysunek 9.8.
Przykładowe
repozytorium
o nieliniowej historii



Odwołania:

A~1
A^1
A~
A^

są równoważne i wskazują rewizję B.

Podobnie odwołania:

D~1
D^1
D~
D^

wskazują rewizję E.

Rewizję D możemy wskazać odwołaniami przedstawionymi na listingu 9.2.

Listing 9.2. Odwołania wskazujące rewizję D

A~3	A~~~
B~2	B~~
C~1	C~

Zwrót uwagi, że do wszystkich rewizji z gałęzi poniżej węzła C dotrzymy, stosując znak \sim .

Do wskazania rewizji H niezbędne będzie użycie odwołania zawierającego ^, gdyż musimy wskazać drugiego rodzica rewizji B. Odwołania prowadzące do rewizji H są przedstawione na listingu 9.3.

Listing 9.3. Odwołania wskazujące rewizję H

```
A~^2~  
A^^2^  
B^2~  
B^2^  
G^  
G~
```

Dziennik reflog

Oprogramowanie Git prowadzi dziennik reflog, w którym zapisywane są identyfikatory bieżących rewizji. Każde polecenie, które zmienia bieżącą rewizję, powoduje dodanie do dziennika reflog kolejnego wpisu.

Jeśli w repozytorium utworzymy kolejno trzy rewizje: a, b, c:

```
echo a>a.txt  
git add -A  
git commit -m a  
  
echo b>b.txt  
git add -A  
git commit -m b  
  
echo c>c.txt  
git add -A  
git commit -m c
```

wówczas bieżącą rewizją będą kolejno: rewizja a, rewizja b, rewizja c, a w dzienniku reflog pojawią się wpisy widoczne na listingu 9.4.

Listing 9.4. Przykładowa zawartość dziennika reflog

```
55e7230 HEAD@{0}: commit: c  
cf3bde3 HEAD@{1}: commit: b  
af7d15f HEAD@{2}: commit: a
```

Do sprawdzenia zawartości dziennika reflog służy komenda:

```
git reflog
```

Wpisy dziennika reflog możemy również wykorzystać do identyfikowania rewizji. Będą one przydatne do wycofywania operacji łączenia gałęzi.

Pierwszy wpis z listingu 9.4:

```
55e7230 HEAD@{0}: commit: c
```

pozwala na odwołanie do rewizji oznaczonej skrótem SHA-1:

55e7230

przy użyciu identyfikatora:

HEAD@{0}

Zapis:

HEAD@{n}

oznacza n-ty od góry wpis w dzienniku reflog. Dziennik reflog pozwala na przeglądanie wpisów dotyczących konkretnych gałęzi. Na przykład polecenie:

git reflog abc

pokaże zawartość dziennika reflog dla gałęzi abc, a polecenie:

git reflog master

dla gałęzi master. Odwołania do wpisów dziennika dotyczących gałęzi mają postać:

nazwa-galezi@{n}

np.:

master@{5}
HEAD@{7}
abc@{1}

W nawiasach klamrowych możemy podać liczbę naturalną określającą, o który wpis chodzi, lub wskazanie punktu w czasie. Odwołanie:

master@{1.hour.ago}

wskazuje stan gałęzi master godzinę temu, a odwołanie:

abc@{2012-06-30.10:25:00 }

stan gałęzi abc w dniu 30 czerwca 2012 o godzinie 10:25. Poprawnymi wskazaniami są m.in.:

@{yyyy-mm-dd.hh:mm:ss}
@{n.minute.ago}
@{n.hour.ago}
@{n.day.ago}

Zawartość dziennika reflog możemy wyczyścić poleceniem:

git reflog expire --all --expire=now

Polecenia rev-parse oraz rev-list

Polecenie:

git rev-parse

umożliwia konwersję wszystkich typów identyfikatorów do skrótu SHA-1.

Polecenie:

```
git rev-list
```

wyświetla listę wszystkich rewizji, do których możemy dotrzeć, rozpoczynającwędrówkę od podanej rewizji.

Wydając komendę:

```
git rev-parse HEAD
```

poznamy skrót SHA-1 rewizji identyfikowanej nazwą symboliczną HEAD. W analogiczny sposób możemy wyznaczyć skróty SHA-1 innych odwołań, np.:

```
git rev-parse 99e6
git rev-parse HEAD~3
git rev-parse HEAD^2
```

Dodatkowy parametr --short:

```
git rev-parse --short=4 HEAD
```

umożliwia skrócenie wydruku do zadanej liczby znaków.

Ponieważ znaczniki opisane są obiekttami, które posiadają własne skróty SHA-1, polecenie:

```
git rev-parse v0.1.2
```

nie wyświetla skrótu SHA-1 rewizji, a skrót SHA-1 znacznika. W celu poznania skrótu SHA-1 rewizji, której dotyczy znacznik, należy użyć polecenia:

```
git rev-list v0.1.2 -1
```

Powyższe polecenie drukuje skrót pierwszej rewizji (parametr -1), która jest osiągalna za pomocą znacznika v0.1.2. W celu skrócenia generowanego wydruku stosujemy opcje --abbrev-commit oraz --abbrev:

```
git rev-list v0.1.2 -1 --abbrev-commit --abbrev=4
```

Znaki specjalne wiersza poleceń Windows

W wierszu poleceń Windows znaki >, <, | i ^ mają specjalne znaczenie i muszą być cytowane. Znaki >, < i | przekierowują strumienie, zaś znak ^ służy do cytowania⁶. Polecenie:

```
echo a>b
```

tworzy plik o nazwie b i zapisuje w nim znak a. W celu wydrukowania napisu a>b należy znak > zabezpieczyć znakiem ^. Komenda:

```
echo a^>b
```

⁶ W wielu językach (m.in. PHP, Python, C++, Java, SQL) do cytowania znaków specjalnych służy znak \.

wydrukuje napis a>b. W celu wydrukowania napisu a^b należy zatem wydać komendę:

```
echo a^^b
```

W przypadku plików wsadowych cytowanie należy wykonać dwukrotnie.

Utwórzmy plik wsadowy o nazwie *skrypt.cmd* i zapiszmy w nim polecenie, które wydrukuje pierwszy z podanych parametrów:

```
echo %1
```

Jeśli teraz zechcemy wywołać *skrypt.cmd* z parametrem zawierającym znaki specjalne, należy zabezpieczenie znaku ^ wykonać dwukrotnie. Po wydaniu polecenia:

```
skrypt.cmd a^^^^b
```

wydrukowany zostanie napis a^b.

Jeśli parametr przekazywany do polecenia wsadowego zawiera zatem znak ^, to w miejscu każdego ^ należy użyć czterech znaków ^.

Program Git uruchamiany w wierszu poleceń Windows powoduje uruchomienie skryptu wsadowego *C:\Program Files (x86)\Git\cmd\git.cmd*, dlatego parametry przekazywane do polecenia git należy zabezpieczać w opisany sposób. Polecenie:

```
git show HEAD^1
```

należy w wierszu polecień Windows zapisać jako:

```
git show HEAD^^^^1
```



Wskazówka

Paradoks ósmiu odwrotnych ukośników

Najzabawniejszym przypadkiem cytowania, z jakim się spotkałem, jest użycie wyrażenia regularnego w kodzie SQL zapisanym w ciągu znaków PHP. PHP, SQL i wyrażenia regularne stosują ten sam znak do cytowania, więc w celu wyszukania pojedynczego znaku \ w zmiennej należy użyć ósmiu znaków \:

```
$q = "
    SELECT
        *
    FROM
        tparadoks
    WHERE
        napis REGEXP '\\\\\\\\\\\\\\\\'
    ";
```

Ćwiczenie 9.1

Sklonuj repozytorium Symfony 2, a następnie wydaj polecenie:

```
git rev-parse v2.0.10
```

W ten sposób sprawdzisz, że znacznik v2.0.10 ma skrót SHA-1:

```
a449273706336b13d4265a4f2396d0d160a0a16d
```

W celu poznania skrótu rewizji wskazanej przez znacznik v2.0.10 wydaj komendę:

```
git rev-list v2.0.10 -1 --abbrev-commit --abbrev=4
```

Wygenerowany zostanie wydruk:

```
7f97b
```

Sprawdźmy, jak wygląda historia projektu w punkcie odpowiadającym rewizji 7f97b.
W tym celu wydajmy jedno z poleceń:

```
git log --pretty=oneline --abbrev-commit --abbrev=4 -10 --graph v2.0.10
git log --pretty=oneline --abbrev-commit --abbrev=4 -10 --graph 7f97b
```

Wygenerowany wydruk będzie miał postać przedstawioną na listingu 9.5.

Listing 9.5. Fragment historii projektu *Symfony 2* od rewizji v2.0.10

```
* 7f97b
* 011b0
* af46e
* d1347
* c9e874
|\
| * a7b4
| * 8e130
| * 045f9
| * a1b6d4
|/
* 9fa32
```



Parametr `--graph` polecenia `git log` generuje wydruk prezentujący strukturę grafu rewizji. Oczywiście graf rewizji możemy także analizować przedstawionym na rysunku 9.9 programem Git GUI.

Odwołanie:

```
v2.0.10
```

wskazuje rewizje:

```
7f97b
```

Odwołanie:

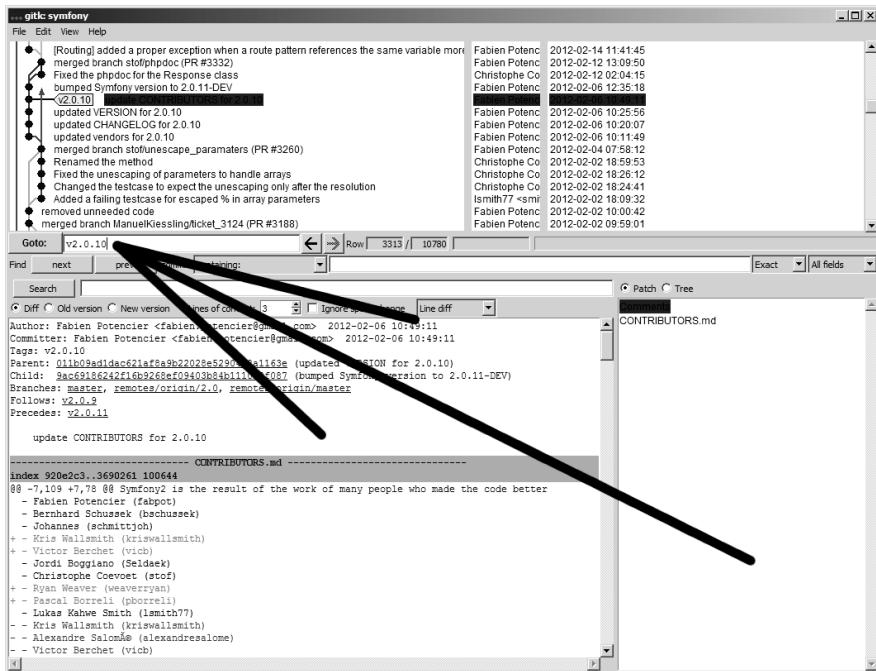
```
v2.0.10~3
```

wskazuje rewizje:

```
d1347
```

Przekonamy się o tym, wydając polecenie:

```
git rev-parse --short=4 v2.0.10~3
```



Rysunek 9.9. Analiza historii projektu Symfony 2 w rewizji oznaczonej znacznikiem v2.0.10

Rewizja c9e874 ma dwóch rodziców. Pierwszym rodzicem jest rewizja 9fa32. Dowodzi tego polecenie:

```
git rev-parse --short=4 c9e874^1
```

Powyższe polecenie należy w systemie Windows zapisać jako:

```
git rev-parse --short=4 c9e874^^^^1
```

Drugim rodzicem rewizji c9e874 jest rewizja a7b4. Sprawdzimy to poleceniami:

```
git rev-parse --short=4 c9e874^2
```

```
//wersja dla wiersza poleceń Windows
git rev-parse --short=4 c9e874^^^^2
```

Do rewizji alb6d4 możemy dotrzeć od rewizji v2.0.10 odwołaniem:

```
v2.0.10~4^2~3
```

Dowodzą tego polecenia:

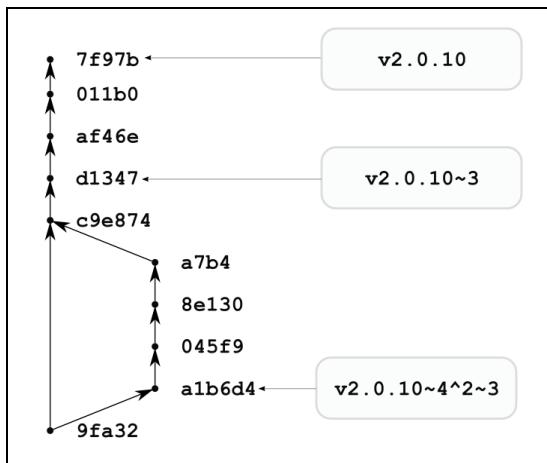
```
git rev-parse --short=4 v2.0.10~4^2~3
```

```
//wersja dla wiersza poleceń Windows
git rev-parse --short=4 v2.0.10~4^^^^2~3
```

Repozytorium z listingu 9.5 wraz z opisanymi powyżej odwołaniami jest przedstawione na rysunku 9.10.

Rysunek 9.10.

Przykładowe odwołania do rewizji projektu *Symfony 2*



Na zakończenie ćwiczenia poleceniem:

```
git log --min-parents=3
```

przekonaj się o tym, że projekt Symfony 2 nie zawiera ani jednej rewizji, która ma więcej niż dwóch rodziców.

Ćwiczenie 9.2

Sklonuj repozytorium zawierające system Linux, po czym poleceniem:

```
git log --pretty=oneline --min-parents=30 --abbrev-commit
```

odszukaj rewizję mającą trzydziestu rodziców. Wygenerowany wydruk będzie następujący:

```
fa623d1 Merge branches 'x86/apic', 'x86/cleanups', ...
```

Następnie poleceniem:

```
git log --pretty=oneline --abbrev-commit --abbrev=4 -40 --graph fa623d1
```

wygeneruj wykres prezentujący czterdzieści rewizji (parametr `-40`) poprzedzających w historii projektu rewizję `fa623d1`. Poleceniem:

```
git rev-parse --short=4 fa623d1^30
```

//wersja dla wiersza poleceń Windows

```
git rev-parse --short=4 fa623d1^^^^30
```

sprawdź, że trzydziestym rodzicem rewizji `fa623d1` jest rewizja widoczna po prawej stronie wykresu. Jej skrót SHA-1 rozpoczyna się od znaków `ecbf29`.

Ćwiczenie 9.3

Wykonaj repozytorium zawierające trzy rewizje: `x`, `y` i `z`. W każdej rewizji dodaj jeden nowy plik tekstowy. Po utworzeniu każdej rewizji sprawdź stan dziennika `reflog`.

Rozdział 10.

Skróty komend

Długie komendy, które często wydajemy, warto zdefiniować w postaci skrótów. Dzięki temu unikniemy wielokrotnego przepisywania parametrów.

Skróty komend zapisujemy w pliku konfiguracyjnym `.gitconfig`, w folderze domowym użytkownika¹, na końcu sekcji oznaczonej etykietą `[alias]`. Przykładowa zawartość pliku `.gitconfig` jest przedstawiona na listingu 10.1.

Listing 10.1. Przykładowe skróty komend zapisane w pliku `.gitconfig`

```
...
[alias]
    l   = log --pretty=oneline --abbrev-commit --abbrev=4 -25
    g   = log --pretty=oneline --abbrev-commit --abbrev=4 -25 -graph
    s   = status -sb
    b   = branch
...
```

Wiersz:

```
l = log --pretty=oneline --abbrev-commit --abbrev=4 -10
```

definiuje skrót, który wywołujemy:

```
git l
```

Do skrótu możemy przekazać dodatkowe parametry, np.:

```
git l -20 --graph
```

Parametr `-20` nadpisze wartość ustaloną na listingu 10.1. W ten sposób wywołując skróconą postać komendy, możemy modyfikować pojedyncze parametry zawarte w definicji skrótu.

¹ W systemie Windows plik ten znajdziemy w folderze `C:\Users\nazwakonta\.gitconfig`.



Szczegółowa dokumentacja skracania komend Gita jest dostępna na stronie:
<https://git.wiki.kernel.org/index.php/Aliases>

Komendy ułatwiające zapisywanie stanu projektu

We wszystkich wykonanych do tej pory ćwiczeniach stan plików repozytorium zapisywaliśmy, wydając dwie komendy:

```
git add -A  
git commit -m "..."
```

Komendy te możemy zdefiniować w postaci skrótu o nazwie save. Skrót taki jest przedstawiony na listingu 10.2.

Listing 10.2. Definicja skróconej komendy git save

```
save = !git add -A && git commit -m "\"$@\"\" && shift 1 && echo Saved!
```

Tak zdefiniowany skrót wywołujemy:

```
git save "Komentarz rewizji..."
```

Porównaj skrócone komendy:

```
1    = log --pretty=oneline --abbrev-commit --abbrev=4 -25  
save = !git add -A && git commit -m "\"$@\"\" && shift 1 && echo Saved!
```

Zwróć uwagę na rolę, jaką odgrywa wykrzyknik. Skrócona komenda rozpoczynająca się od znaku innego niż wykrzyknik:

```
1orem = ipsum
```

po wywołaniu:

```
git 1orem
```

będzie zamieniona na wywołanie:

```
git ipsum
```

Jeśli natomiast skrócona komenda rozpoczyna się od wykrzywnika:

```
dolor = !sit
```

wówczas wywołanie:

```
git dolor
```

spowoduje wykonanie komendy:

```
sit
```

Innymi słowy wykrzyknik należy stosować w przypadku komend, które są kompletnymi poleceniami bash.



Przedstawione na listingach 10.2 – 10.6 skróty komend Gita są napisane w języku bash. Zapis²:

```
lorem && ipsum && dolor
```

powoduje wykonanie trzech poleceń (pod warunkiem, że wszystkie polecenia zwracają logiczną prawdę, która w przypadku komend wsadowych ma wartość 0):

```
lorem  
ipsum  
dolor
```

W skryptach z listingów 10.2 – 10.6 stosowane są następujące zagadnienia składni bash:

◆ drukowanie napisów:

```
echo "Saved!"
```

◆ przekierowywanie strumienia wyjściowego do pliku:

```
echo a>a.txt
```

◆ pobieranie pierwszego parametru polecenia: \$1

◆ pobieranie wszystkich parametrów polecenia: \$@

◆ przesuwanie parametrów polecenia:

```
shift 1
```

◆ cytowanie cudzysłowa \" \"

◆ definicja i wywoływanie funkcji:

```
foo() { }; foo
```

◆ pętla for

```
for name in zakres; do done;
```

◆ pętla while

```
while ... ; do done;
```

◆ przypisywanie wartości zmiennej:

```
NAME=$1;
```

◆ obliczanie wartości wyrażenia arytmetycznego i przypisywanie do zmiennej:

```
i=${$i+1} :
```

◆ porównywanie dwóch liczb:

```
[ $i -le $2 ]
```

◆ generowanie wartości na podstawie dwóch zmiennych:

```
$NAME$i
```

◆ wywoływanie serii poleceń:

```
lorem && ipsum && dolor
```

² Por. dokumentacja Linuksa <http://tldp.org/LDP/abs/html/list-cons.html#LISTCONSREF>.

Komendy ułatwiające wykonywanie ćwiczeń

W wielu ćwiczeniach pojawia się konieczność utworzenia pojedynczej rewizji oznaczonej krótkim tekstem i zawierającej jeden plik. Rewizję taką możemy wykonać poleceniami:

```
echo a>a.txt
git add -A
git commit -m a
```

W celu ułatwienia wykonywania tego typu rewizji zdefiniujmy przedstawiony na listingu 10.3 skrót `simple-commit`.

Listing 10.3. Definicja³ skróconej komendy `git simple-commit`

```
simple-commit =
    !echo $1>$1.txt &&
    git add -A &&
    git commit -m "\"$1\""
    shift 1 &&
    echo Simple commit done!
```

Skrót `simple-commit` wywołujemy:

```
git simple-commit a
```

Do wykonania serii rewizji takich jak w ćwiczeniu 6.3 przydatna będzie skrócona komenda `simple-commits`, która jest widoczna na listingu 10.4.

Listing 10.4. Definicja skróconej komendy `git simple-commits`

```
simple-commits = "
!simpleCommits() {
    for name in \"\$@\"; do
        git simple-commit $name;
        done;
    };
    simpleCommits
"
```

Komendę `single-commits` z listingu 10.4 możemy w ćwiczeniu 6.3 wywołać:

```
git simple-commits a1 a2 a3
```

Powyższa komenda spowoduje utworzenie trzech rewizji:

```
a1
a2
a3
```

³ Skrót ten należy zapisać w jednym wierszu. Poniższy zapis został podzielony na osobne linijki w celu zwiększenia czytelności.

Polecenie simple-commits możemy wywołać, podając jako parametry dowolne litery lub napisy:

```
git simple-commits a b c  
git simple-commits lorem ipsum dolor  
git simple-commits p q r s t
```

Innym rozwiązaniem przydatnym podczas tworzenia serii rewizji jest automatyczna numeracja. Do tworzenia automatycznie numerowanych rewizji służy przedstawiona na listingu 10.5 komenda simple-loop.

Listing 10.5. Definicja skróconej komendy git simple-loop

```
simple-loop = "  
    !simpleLoop() {  
        NAME=$1;  
        i="1";  
        while [ $i -le $2 ]; do  
            git simple-commit $NAME$i;  
            i=$[$i+1];  
        done;  
    };  
    simpleLoop  
"
```

Komendę simple-loop z listingu 10.5 możemy w ćwiczeniu 6.3 wywołać:

```
git simple-commits a 3
```

Powyższa komenda spowoduje utworzenie trzech rewizji:

```
a1  
a2  
a3
```

Rewizje tworzone polecienniem simple-loop są numerowane od 1. Dowolną numerację rewizji osiągniemy przedstawionym na listingu 10.6 skrótem simple-loop2.

Listing 10.6. Definicja skróconej komendy git simple-loop2

```
simple-loop2 = "  
    !simpleLoop2() {  
        NAME=$1;  
        i=$2;  
        while [ $i -le $3 ]; do  
            git simple-commit $NAME$i;  
            i=$[$i+1];  
        done;  
    };  
    simpleLoop2  
"
```

Polecenie z listingu 10.6 wywołujemy następująco:

```
git simple-loop2 lorem 7 13
```

Spowoduje ono utworzenie rewizji:

```
torem-7
torem-8
...
torem-13
```



Wskazówka

Skrócone polecenia:

```
git simple-commit
git simple-commits
git simple-loop
git simple-loop2
```

służą wyłącznie do ćwiczenia umiejętności operowania komendami Gita. Będą one szczególnie przydatne w repozytoriach, których gałęzie zawierają wiele rewizji, jak na przykład w ćwiczeniach 14.9 oraz 15.5. Polecenia te są zupełnie nieprzydatne podczas pracy wykonywanej w rzeczywistych repozytoriach.

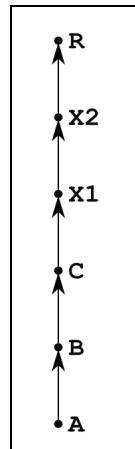
Ćwiczenie 10.1

Wykonaj repozytorium przedstawione na rysunku 10.1. W każdej rewizji umieść jeden plik, którego nazwa będzie zgodna z podpisem rewizji. Na przykład w rewizji X1 utwórz jeden nowy plik *X1.txt*, zawierający tekst *X1*. Użyj skróconych komend:

```
git simple-commits
git simple-loop
git simple-commit
```

Rysunek 10.1.

Repozytorium, które należy utworzyć w ćwiczeniu 10.1



Przed każdym poleceniem sprawdź zawartość dziennika reflog.

Następnie poleceniami:

```
git reset --hard
```

przywrócić stan repozytorium z rewizji X1. Po tej operacji sprawdź stan dziennika reflog.

Następnie przywróć stan repozytorium z rewizji R. Do odnalezienia rewizji R użyj dziennika reflog.

Na zakończenie wyczyść dziennik reflog.

ROZWIĄZANIE

W dowolnym folderze wydaj polecenie:

```
git init
```

W repozytorium nie ma rewizji, więc polecenie:

```
git reflog
```

zwraca informację o błędzie:

```
fatal: bad default revision 'HEAD'
```

Do utworzenia rewizji użyj polecień:

```
git simple-commits A B C  
git simple-loop X 2  
git simple-commit R
```

Po wydaniu powyższych trzech poleceń stan dziennika będzie taki jak na listingu 10.7.

Listing 10.7. Stan dziennika reflog po utworzeniu rewizji z rysunku 10.1

```
ce06c9b HEAD@{0}: commit: R  
6376f81 HEAD@{1}: commit: X2  
1c54452 HEAD@{2}: commit: X1  
3833c18 HEAD@{3}: commit: C  
95028fc HEAD@{4}: commit: B  
4abbc77 HEAD@{5}: commit (initial): A
```

Poleceniem:

```
git reset --hard HEAD~2
```

przywracamy stan repozytorium z rewizji X1. Dziennik reflog przyjmie postać taką jak na listingu 10.8.

Listing 10.8. Stan dziennika reflog po przywróceniu obszaru roboczego do rewizji X1

```
1c54452 HEAD@{0}: reset: moving to HEAD~2  
ce06c9b HEAD@{1}: commit: R  
6376f81 HEAD@{2}: commit: X2  
1c54452 HEAD@{3}: commit: X1  
3833c18 HEAD@{4}: commit: C  
95028fc HEAD@{5}: commit: B  
4abbc77 HEAD@{6}: commit (initial): A
```

W celu przywrócenia stanu repozytorium z rewizji R wykorzystujemy identyfikator HEAD@{1} widoczny na listingu 10.8. Polecenie:

```
git reset --hard HEAD@{1}
```

przywróci stan repozytorium do rewizji R. Po wydaniu polecenia:

```
git reflog expire --all --expire=now
```

dziennik reflog będzie pusty, o czym przekonasz się, wydając komendę:

```
git reflog
```

W takiej sytuacji przywrócenie stanu z rewizji R nie byłoby możliwe.



Wszystkie opisane w podręczniku skrócone komendy znajdziesz w repozytorium:

<https://github.com/git-podrecznik/alias>

Rozdział 11.

Modyfikowanie historii projektu

W niektórych przypadkach zachodzi potrzeba zmodyfikowania historii projektu. Przykładem takiej sytuacji jest konieczność połączenia kilku nowych rewizji w jedną rewizję zbiorczą. Wymaganie takie jest stawiane m.in. uczestnikom projektów open source. Uczestnictwo w projekcie open source przebiega według następującego schematu:

- ◆ Uczestnik dodaje do projektu rewizję zawierającą pewne zmiany.
- ◆ Na podstawie dyskusji z pozostałymi członkami projektu uczestnik wprowadza kolejne rewizje odpowiadające wnioskom z dyskusji.

W ten sposób w projekcie pojawia się seria rewizji dotyczących jednego zagadnienia. Przed ostatecznym dołączeniem wprowadzonych zmian uczestnik scala wszystkie swoje rewizje. Dzięki temu komplet zmian wprowadzonych przez uczestnika, wraz z poprawkami wynikającymi z dyskusji, występuje w jednej rewizji.

Innym przykładem jest konieczność usunięcia błędnej rewizji, która jest zawarta w historii projektu. W takim przypadku do projektu dodawana jest następna rewizja, która odwraca modyfikacje wprowadzone przez błędą rewizję.

Należy pamiętać, że modyfikowanie historii projektu to operacja bardzo ryzykowna, która — w przypadku błędnego wykonania — uniemożliwi współpracę grupową. Synchronizacja repozytorium pomiędzy uczestnikami projektu grupowego opiera się na tym, że współdzielona historia projektu nie podlega zmianom. Z tego powodu operacje modyfikacji historii projektu możemy podzielić na dwie grupy:

- ◆ operacje mogące być wykonywane wyłącznie w odniesieniu do rewizji, które nie są współdzielone;
- ◆ operacje mogące być wykonywane na wszystkich rewizjach.

Operacjami, które możemy wykonywać wyłącznie na prywatnych rewizjach¹, są:

- ◆ usuwanie kilku ostatnich rewizji (`git reset --hard`),
- ◆ modyfikowanie ostatniej rewizji (`git commit --amend`),
- ◆ łączenie wybranych rewizji (`git rebase`).

Operacją, którą możemy wykonywać na wszystkich rewizjach (zarówno prywatnych, jak i publicznych), jest:

- ◆ dodawanie rewizji odwracającej działanie innej rewizji (`git revert`).



Pamiętaj, że podstawową cechą publicznego repozytorium Gita jest niezmienność jego historii. Jedyną dopuszczalną operacją na publicznym repozytorium Gita jest dodawanie nowych rewizji. W historii projektu nie można modyfikować ani usuwać żadnych rewizji, gdyż zakłoci to synchronizację repozytorium pozostałym uczestnikom projektu. Z tego powodu jedyną metodą modyfikowania publicznych rewizji jest dodawanie rewizji odwracających działanie poprzednich rewizji (`git revert`).

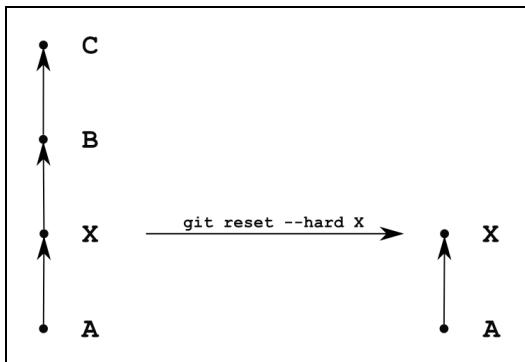
Usuwanie ostatnich rewizji

Polecenie:

`git reset --hard X`

usuwa z historii projektu wszystkie rewizje, które wystąpiły po rewizji X. Działanie komendy ilustruje rysunek 11.1.

Rysunek 11.1.
Usuwanie ostatnich
rewizji poleceniem
`git reset --hard X`



Jeśli historia projektu jest liniowa i nie zawiera rozgałęzień, rewizje B i C zostaną wówczas utracone². Jeśli historia projektu zawiera gałęzie, a rewizje B i C występują w kilku

¹ Terminem „rewizja prywatna” określам rewizję, która nie została udostępniona publicznie.

² W ćwiczeniu 10.1 omówiliśmy metodę odzyskiwania rewizji usuniętych poleceniem `git reset --hard` przy użyciu dziennika reflog.

gałęziach, polecenie git reset --hard usunie wówczas rewizje B i C wyłącznie z bieżącej gałęzi.



Uwaga

Operację usuwania ostatnich rewizji wykonujemy wyłącznie na prywatnych gałęziach. Operacji tej nigdy nie wykonujemy w celu usuwania publicznych rewizji.

Modyfikowanie ostatniej rewizji

Do poprawienia ostatniej rewizji służy polecenie git commit z parametrem --amend, np.:

```
git commit --amend -m "..."
```

Polecenie to zmienia skrót SHA-1 rewizji, dlatego należy je interpretować jako usunięcie ostatniej rewizji i utworzenie nowej rewizji, której modyfikacje obejmą zmiany z ostatniej rewizji oraz z indeksu. Ponieważ skrót SHA-1 rewizji ulega zmianie, rewizja ta będzie interpretowana podczas synchronizacji jako nowa rewizja.



Uwaga

Operację modyfikowania ostatniej rewizji wykonujemy tylko wtedy, gdy ostatnia rewizja nie została udostępniona innym uczestnikom projektu.

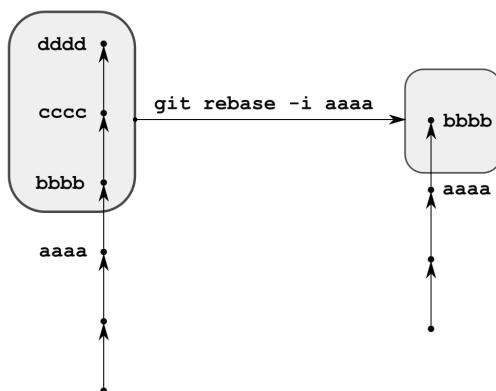
Łączenie rewizji

Do łączenia rewizji służy polecenie:

```
git rebase -i X
```

Mogimy je wykorzystać do połączenia wszystkich rewizji występujących w historii projektu po rewizji X. Działanie polecenia ilustruje rysunek 11.2.

Rysunek 11.2.
Operacja łączenia
rewizji





Wskazówka

Jeśli nazwa symboliczna HEAD wskazuje na rewizję dddd, to polecenia:

```
git rebase -i aaaa
git rebase -i HEAD~3
```

są równoważne. Polecenie:

```
git rebase -i HEAD~n
```

możemy interpretować jako łączenie ostatnich n rewizji.

Polecenie:

```
git rebase -i aaaa
```

możemy zaś interpretować jako łączenie wszystkich rewizji występujących po rewizji aaaa.

Jeśli w historii projektu występuje rewizja oznaczona skróconym SHA-1 aaaa, wówczas polecenie:

```
git rebase -i aaaa
```

możemy wykorzystać do modyfikacji historii projektu w sposób przedstawiony na rysunku 11.2. Wynikiem działania polecenia jest połączenie rewizji występujących po rewizji aaaa. Wszystkie zmiany plików w łączonych rewizjach zostaną zachowane.

W celu wykonania operacji z rysunku 11.2 należy najpierw wydać polecenie:

```
git rebase -i aaaa
```

Program Git uruchomi edytor tekstowy³ zawierający treść przedstawioną na listingu 11.1.

Listing 11.1. Zawartość edytora tekstuowego uruchamianego po wydaniu polecenia git rebase -i aaaa

```
pick bbbb
pick cccc
pick dddd

# Rebase aaaa..ddd onto aaaa
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
```

Zmodyfikuj zawartość edytora tak, by zawierał on treść:

³ Parametr -i włącza tzw. tryb interaktywny polecenia rebase.

```
pick bbbb
fixup cccc
fixup dddd
```

Pierwszą z rewizji (tę, która występuje na samej górze edytora) oznaczamy słowem `pick`. Kolejne rewizje oznaczamy słowami `fixup`. Tak zmodyfikowany plik tekstowy zapisujemy i wyłączamy edytor.

Po zamknięciu edytora Git wykona operację łączenia.

Jeśli po wydaniu polecenia `git rebase -i` zechcesz zrezygnować z wykonywania łączenia, usuń całą zawartość edytora i zapisz pusty plik.

Podpowiedź widoczna na listingu 11.1:

```
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
```

wyjaśnia znaczenie komend, które możemy umieścić w pliku tekstowym.

Słowo `pick` (możemy je zapisać skrótnie jako `p`) powoduje, że podana rewizja wystąpi w historii projektu po zakończeniu operacji.

Słowo `reword` (skrót `r`) ma działanie identyczne jak `pick`, ale pozwoli na edycję opisu danej rewizji (tj. tekstu przekazywanego jako parametr `-m` komendy `git commit`). Poleceniem tym możemy zmienić opisy wszystkich rewizji.

Słowo `edit` (skrót `e`) ma działanie takie jak `pick`, ale pozwala na dokonanie zmian w rewizji.

Słowo `squash` (skrót `s`) łączy podaną rewizję z poprzednią rewizją. Pozwala ono na modyfikację komentarza wynikowej rewizji.

Słowo `fixup` (skrót `f`) ma działanie takie jak `squash`, lecz nie pozwala na modyfikowanie komentarza.

Ostatnia komenda, `exec` (skrót `x`), pozwala na wykonanie dowolnych komend wiersza poleceń.

Jeśli użyjesz polecenia, które pozwala na modyfikację komentarza, np. `squash`, wówczas po zamknięciu edytora zawierającego tekst z listingu 11.1 uruchomiony zostanie edytor zawierający treść komentarza generowanej rewizji.



Operację łączenia rewizji wykonujemy tylko na rewizjach, które nie zostały udostępnione innym uczestnikom projektu.

Usuwanie zmian wprowadzonych przez rewizję

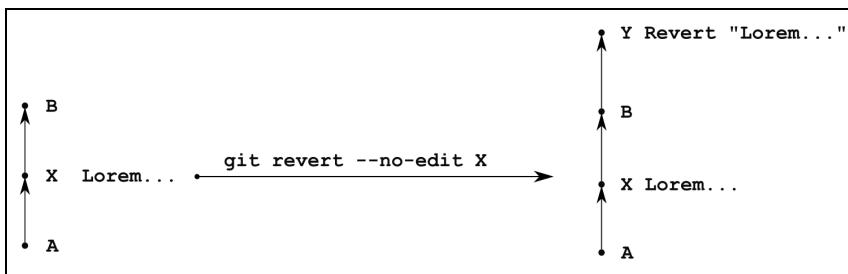
Jeśli chcesz usunąć modyfikacje wprowadzone przez rewizję, użyj polecenia:

```
git revert --no-edit X
```

Polecenie to tworzy w historii projektu kolejną rewizję, która wycofuje modyfikacje wprowadzone przez rewizję X. Parametr `--no-edit` nadaje domyślny komentarz tworzonej rewizji:

```
Revert "..."
```

Działanie komendy `git revert --no-edit X` ilustruje rysunek 11.3.



Rysunek 11.3. Działanie komendy `git revert --no-edit X`



Komenda `git revert` to jedyna z opisanych w tym rozdziale modyfikacji historii projektu, którą możemy wykonywać także na publicznie dostępnych rewizjach.

Ćwiczenie 11.1

W folderze *cw-11-01*/utwórz repozytorium o historii przedstawionej na rysunku 11.4.

Rysunek 11.4.

Repozytorium z ćwiczenia 11.1



W tym celu wydaj komendy przedstawione na listingu 11.2.

Listing 11.2. Komendy tworzące repozytorium z rysunku 11.4

```
mkdir cw-11-01  
cd cw-11-01  
git init  
  
git simple-commits Lorem Ipsum Dolor Sit Amet
```



Wskazówka

Omówione w poprzednim rozdziale polecenie `simple-commits` naprawdę ułatwia naukę Gita. Ręczne utworzenie repozytorium z rysunku 11.1 będzie wymagało wydania kilkunastu komend:

```
echo Lorem>Lorem.txt  
git add -A  
git commit -m Lorem  
  
echo Ipsum>Ipsum.txt  
git add -A  
git commit -m Ipsum  
  
echo Dolor>Dolor.txt  
git add -A  
git commit -m Dolor  
  
echo Sit>Sit.txt  
git add -A  
git commit -m Sit  
  
echo Amet>Amet.txt  
git add -A  
git commit -m Amet
```

Po wykonaniu komend z listingu 11.2 polecenie `git log4` zwróci informacje o pięciu rewizjach:

```
0543 Amet  
960a Sit  
af0c Dolor  
1e9c Ipsum  
1637 Lorem
```

Ćwiczenie 11.2

Wykonaj kopię repozytorium z ćwiczenia 11.1, po czym usuń wszystkie rewizje występujące po rewizji oznaczonej komunikatem *Ipsum*.

⁴ Oczywiście polecenie `git log` możesz wywołać przy użyciu skrótu `git l` zdefiniowanego w rozdziale 10.

ROZWIĄZANIE

Skopiuj folder *cw-11-01/*. Kopię nazwij *cw-11-02/*. Chcemy usunąć wszystkie rewizje występujące po rewizji:

```
1e9c Ipsum
```

W tym celu wydajemy komendę:

```
git reset --hard 1e9c
```

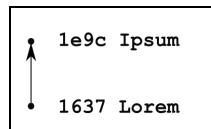
Po wykonaniu powyzszego polecenia historia projektu będzie taka jak na rysunku 11.5. Dowodzi tego komenda *git log*, która generuje wydruk:

```
1e9c Ipsum
```

```
1637 Lorem
```

Rysunek 11.5.

*Repozytorium
otrzymane
po wykonaniu
ćwiczenia 11.2*



Ćwiczenie 11.3

Wykonaj kopię repozytorium z ćwiczenia 11.1, po czym zmodyfikuj ostatnią rewizję. Dodaj do rewizji dwa pliki: *foo.txt* i *bar.txt* oraz zmień komunikat na *Foo bar*.

ROZWIĄZANIE

Skopiuj folder *cw-11-01/*. Kopię nazwij *cw-11-03/*.

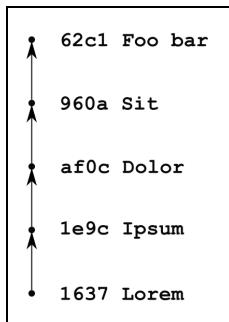
W folderze projektu utwórz pliki *foo.txt* i *bar.txt*, a następnie wydaj komendy:

```
git add -A
git commit --amend -m "Foo bar"
```

Otrzymasz repozytorium o historii przedstawionej na rysunku 11.6.

Rysunek 11.6.

*Repozytorium
otrzymane
po wykonaniu
ćwiczenia 11.3*



Wydruk generowany poleciennem git log będzie następujący:

```
62c1 Foo bar
960a Sit
af0c Dolor
1e9c Ipsum
1637 Lorem
```

Zwróć uwagę, że ostatnia z rewizji ma zmieniony opis (z *Amet* na *Foo bar*) oraz skrót SHA-1.

Ćwiczenie 11.4

Wykonaj kopię repozytorium z ćwiczenia 11.1. W skopiowanym repozytorium trzy ostatnie rewizje oznaczone komunikatami *Dolor*, *Sit* i *Amet* połącz w jedną rewizję oznaczoną komunikatem *Foo bar*.

ROZWIĄZANIE

Skopiuj folder *cw-11-01/*. Kopię nazwij *cw-11-04/*. Chcemy połączyć wszystkie rewizje występujące po rewizji:

```
1e9c Ipsum
```

W tym celu wydajemy komendę:

```
git rebase -i 1e9c
```

Parametr *-i* włącza tryb interaktywny, który spowoduje uruchomienie edytora tekstu-wego zawierającego treść:

```
pick af0c3b7 Dolor
pick 960af27 Sit
pick 0543e46 Amet
```

Zmieniamy treść zawartą w edytorze na:

```
pick af0c3b7 Dolor
squash 960af27 Sit
squash 0543e46 Amet
```

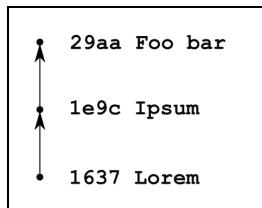
Po zapisaniu i zamknięciu edytora uruchomiony zostanie edytor zawierający komunikat opisujący tworzoną rewizję. W edytorze usuń całą zawartość i wprowadź tekst:

```
Foo bar
```

Po zapisaniu pliku i zamknięciu edytora wykonana zostanie operacja łączenia, w wyniku której trzy rewizje: *Dolor*, *Sit* i *Amet* zostaną połączone w jedną rewizję oznaczoną komunikatem *Foo bar*. Otrzymamy repozytorium przedstawione na rysunku 11.7.

Rysunek 11.7.

*Repozytorium
otrzymane
po wykonaniu
ćwiczenia 11.4*



Polecenie git log zwróci wynik:

```
29aa Foo bar
1e9c Ipsum
1637 Lorem
```

Jeśli wykonując ćwiczenie, zamiast komend squash użyjesz fixup, otrzymana rewizja będzie wówczas oznaczona komunikatem *Dolor*.

Ćwiczenie 11.5

Wykonaj kopię repozytorium z ćwiczenia 11.1. W skopiowanym repozytorium dodaj rewizję odwracającą działanie rewizji oznaczonej komunikatem *Dolor*.

ROZWIĄZANIE

Skopiuj folder *cw-11-01/*. Kopię nazwij *cw-11-05/*. Chcemy usunąć zmiany wprowadzone przez rewizję:

```
af0c Dolor
```

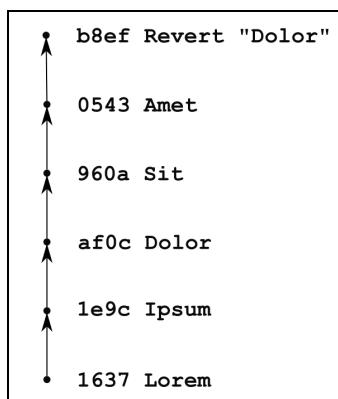
W tym celu wydajemy komendę:

```
git revert --no-edit af0c
```

Otrzymamy repozytorium przedstawione na rysunku 11.8.

Rysunek 11.8.

*Repozytorium
otrzymane
po wykonaniu
ćwiczenia 11.5*



Polecenie git 1 zwróci wynik:

```
b8ef Revert "Dolor"  
0543 Amet  
960a Sit  
af0c Dolor  
1e9c Ipsum  
1637 Lorem
```

Sprawdź zawartość obszaru roboczego. Okaże się, że nie występuje w nim plik *dolor.txt*. Plik ten utworzyliśmy w rewizji *Dolor*. Komenda git revert odwraca zmiany wprowadzone w danej rewizji, co powoduje m.in. usunięcie utworzonych plików.



Uwaga Git nie umożliwia tworzenia rewizji, które nie wprowadzają żadnych zmian. Jeśli żadne zmiany nie zostały zaindeksowane, operacja git commit zwróci wówczas komunikat:

```
nothing to commit (working directory clean)
```

Z tego powodu nie możemy wykonać operacji łączenia dwóch rewizji, w których jedna jest odwróceniem drugiej. Innymi słowy jeśli w projekcie występują rewizje:

```
X Revert "Abc"  
Y Abc
```

wówczas rewizje te nie mogą być scalone.

Odzyskiwanie poszczególnych plików z dowolnej rewizji

Rewizje stanowią trwały zapis stanu projektu, który zawsze możemy odzyskać. Wiemy już, że polecenie:

```
git checkout -f X
```

przywraca stan wszystkich plików w obszarze roboczym do postaci z rewizji X.

Operację przywracania pliku możemy również wykonywać na pojedynczych plikach i folderach. Polecenie:

```
git checkout -f X nazwa-pliku
```

przywraca plik *nazwa-pliku* w obszarze roboczym do stanu z rewizji X. Dzięki temu każdy z plików repozytorium możemy przywrócić do dowolnego stanu, który jest zapisany w którejkolwiek rewizji.

Po wykonaniu ćwiczenia 11.5 obszar roboczy nie zawiera pliku *dolor.txt*. Jeśli plik ten zechcemy odzyskać z rewizji:

```
af0c Dolor
```

należy wówczas wydać polecenie:

```
git checkout -f af0c dolor.txt
```


Rozdział 12.

Podsumowanie części I

Kluczowymi pojęciami, które należy opanować w początkowej fazie nauki Gita, są:

- ◆ repozytorium (tj. folder, którego zawartość jest kontrolowana oprogramowaniem Git),
- ◆ rewizja,
- ◆ obszar roboczy,
- ◆ indeks,
- ◆ baza danych repozytorium (określana w skrócie jako repozytorium),
- ◆ stan pliku.

Terminem repozytorium określamy zarówno cały folder, którego zawartość kontrolujemy Gitem, jak i samą bazę danych zawartą wewnątrz folderu `.git`. Ta dwuznaczność zazwyczaj nie prowadzi do niejasności.

Rewizja to atomowa operacja zapisująca stan plików projektu w repozytorium. Może ona dotyczyć dowolnej liczby plików i wprowadzanych w nich zmian. Każdy plik zawarty w repozytorium możemy przywrócić do stanu odpowiadającego dowolnej rewizji. W ten sposób repozytorium możemy traktować jako dodatkowy wymiar czasowy systemu plików. Git gwarantuje, że nie utracimy żadnych zmian wprowadzonych w repozytorium, o ile zmiany te zostały zapisane w którykolwiek rewizji.

Praca grupowa nad projektem jest ograniczona wyłącznie do operacji dodawania nowych rewizji. Innymi słowy rewizja, która pojawiła się w projekcie i została upublicziona (tj. inni uczestnicy projektu ją pobrali), nigdy nie zostanie zmieniona ani usunięta. Jedyną metodą usunięcia takiej rewizji jest wykonanie operacji `git revert`, która dodaje do projektu nową rewizję niwelującą działanie usuwanej rewizji. Historia projektu rozwija się wyłącznie przez dodawanie nowych rewizji.



Wskazówka

Git gwarantuje, że nie utracimy żadnych zmian wprowadzonych w repozytorium, o ile zmiany te zostały zapisane w którejkolwiek rewizji. Z tego powodu codzienną pracę nad każdym projektem zawsze kończę wykonaniem rewizji. Każdy projekt, nad którym pracuję, pozostawiam w stanie, w którym polecam:

```
git status -s
```

zwraca pusty wynik. Dodatkowo repozytorium przesyłam na serwer. W ten sposób wykonuję codzienny backup danych. Ponieważ Git działa różnicowo (tj. nowe rewizje nie zawierają danych zdublowanych z poprzednich rewizji) takie rozwiązanie jest bardzo wydajne i zajmuje znacznie mniej czasu niż kopiowanie danych na nośniki (np. CD, DVD, pendrive'a, FTP).

Zawartość folderu kontrolowanego przez Git jest podzielona na trzy fragmenty:

- ◆ obszar roboczy,
- ◆ indeks,
- ◆ baza danych repozytorium.

Obszar roboczy to folder główny repozytorium z wykluczeniem podfolderu `.git`. Obszar roboczy traktujemy jako tymczasową przechowalnię, w której pracujemy. Gdy pliki w obszarze roboczym przyjmą postać, którą chcemy zapamiętać w repozytorium, wtedy tworzymy rewizję. Pliki obszaru roboczego możemy przywrócić do stanu odpowiadającego dowolnej rewizji pojedynczym poleceniem.

Indeks to baza danych zawarta w folderze `.git`. Zawiera ona informacje o zmianach, które zostaną zapisane w następnej rewizji.

Baza danych repozytorium (nazywana w skrócie także repozytorium) zawiera natomiast informacje o wszystkich rewizjach.

Git nigdy nie tworzy rewizji automatycznie. Jeśli więc zmienimy jakieś pliki, to w celu zapamiętania stanu projektu należy samodzielnie wykonać rewizję.

Pliki projektu możemy podzielić na:

- ◆ ignorowane,
- ◆ nieignorowane,
- ◆ aktualne,
- ◆ zaindeksowane,
- ◆ niezaindeksowane.

Codzienna praca w znacznej większości ogranicza się do uproszczonego modelu, w którym pliki możemy podzielić na aktualne i pozostałe. Dwa polecenia:

```
git add -A  
git commit -m "..."
```

zapisują bieżący stan wszystkich plików obszaru roboczego w postaci nowej rewizji. Po wydaniu powyższych poleceń wszystkie pliki projektu są aktualne.

Pliki zawarte w obszarze roboczym możemy modyfikować dowolnymi metodami. Spół wprowadzania zmian w plikach nie ma żadnego znaczenia. Możemy stosować dowolne edytory oraz polecenia.

Folder, którego zawartość nadzorujemy programem Git, możemy przemianować i prześwietlić w dowolne miejsce na dysku. Repozytorium Gita nie jest wrażliwe na taką operację. W ten sposób postąpiliśmy między innymi w ćwiczeniach z rozdziału 11.

Po lekturze pierwszej części podręcznika powinieneś rozumieć dwie pierwsze zalety Gita, które wymieniliśmy w rozdziale 1., czyli:

- ♦ lokalność,
- ♦ kontrolę spójności danych.

Lokalność oznacza, że operacje wykonywane poleceniami Gita nie wymagają transferu danych poprzez sieć. Są to zwykłe operacje plikowe wykonywane na dysku lokalnym. Git zapisuje wszystkie informacje o repozytorium w różnych plikach w podfolderze `.git/`. Główne zadania wykonywane podczas codziennej pracy, czyli:

- ♦ tworzenie rewizji,
- ♦ tworzenie, usuwanie i łączenie gałęzi,
- ♦ analizowanie historii projektu,
- ♦ operowanie znacznikami,
- ♦ modyfikacja konfiguracji Gita,

są wykonywane na dysku lokalnym. Spośród poznanych w pierwszej części operacji tylko polecenie `git clone` wymagało komunikacji sieciowej. W części trzeciej poznamy m.in. polecenia:

```
git pull  
git push
```

służące do synchronizacji repozytoriów wszystkich uczestników projektu grupowego. Jeśli repozytoria członków zespołu będą miały dostęp do wspólnego dysku, który możemy przemapować, nadając mu nazwę `S:\`, wówczas także operacje:

```
git clone  
git pull  
git push
```

możemy traktować jako lokalne!

Kontrola spójności danych jest osiągnięta dzięki temu, że wszystkie zmiany zapisywane w rewizji są oznaczane skrótami SHA-1. Jeśli jakiś plik projektu został zmieniony, to Git to rozpozna. Modyfikacje zostaną zawarte w kolejnej rewizji. Nie ma możliwości zmiany pliku zawartego w repozytorium w taki sposób, by Git tego nie zauważał. Bez względu na to, jaką zmianę w pliku wykonamy (możemy dodać tylko jedną spację lub jeden pusty wiersz), modyfikacja będzie widoczna w ostatniej rewizji.

Co powinieneś umieć po lekturze pierwszej części?

Po wykonaniu ćwiczeń z pierwszej części podręcznika powinieneś umieć biegle tworzyć repozytoria o liniowej historii, które w poszczególnych rewizjach zawierają pojedyncze nowe pliki. Ten sposób pracy będzie stanowił punkt wyjściowy do poznawania zagadnień dotyczących gałęzi.

Listy poznanych poleceń

Wersja oprogramowania

```
git --version
```

Dokumentacja

Dokumentacja komend Gita:

```
git add --help  
git branch --help  
git config --help  
git init --help
```

lub:

```
git help add  
git help branch  
git help config  
git help init
```

Konfiguracja

Lista wszystkich opcji konfiguracyjnych:

```
git config -l
```

Ustalenie wartości wybranej opcji konfiguracyjnej:

```
git config --global user.name "Imię Nazwisko"  
git config --global user.email you@example.com
```

Globalny plik konfiguracyjny użytkownika:

```
C:\Users\nazwa-uzytkownika\.gitconfig
```

Plik konfiguracyjny dotyczący konkretnego repozytorium:

```
.git\config
```

Inicjalizacja repozytorium

Inicjalizacja nowego repozytorium zwykłego w bieżącym folderze:

```
git init
```

Inicjalizacja nowego repozytorium zwykłego w podanym folderze:

```
git init sciezka/do/folderu
```

Inicjalizacja nowego repozytorium surowego w bieżącym folderze:

```
git init --bare
```

Inicjalizacja nowego repozytorium surowego w podanym folderze:

```
git init --bare sciezka/do/folderu
```

Klonowanie repozytorium

Klonowanie repozytorium do folderu o nazwie identycznej jak repozytorium:

```
git clone adres-repozytorium
```

Klonowanie repozytorium do bieżącego folderu:

```
git clone adres-repozytorium .
```

Klonowanie repozytorium do dowolnego folderu:

```
git clone adres-repozytorium sciezka/do/folderu
```

Informacje o repozytorium

Lista uczestników projektu:

```
git shortlog -s -n
```

Liczba uczestników projektu:

```
git shortlog -s -n | wc -l
```

Liczba plików w repozytorium:

```
find . -type f -print | grep -v -E '/\.git/' | wc -l
```

Liczba rewizji zawartych w repozytorium:

```
git log --pretty=oneline | wc -l
```

Historia projektu

Sprawdzanie historii projektu:

```
git log
```

Skrócona historia projektu:

```
git log --pretty=oneline  
git log --oneline
```

Skrócona historia zawierająca zminimalizowane identyfikatory SHA-1:

```
git log --abbrev-commit --abbrev=4 --pretty=oneline
```

Historia zawierająca trzy ostatnie rewizje:

```
git log -3
```

Historia zawierająca rewizje podanego użytkownika:

```
git log --author=janek
```

Historia zawierająca rewizje z podanego okresu:

```
git log --since="2012-01-01" --until="2012-01-15"
```

Formatowanie wydruku historii według dowolnych reguł:

```
git log --pretty=format:"%h %cd"
```

Lista ostatnich 10 rewizji w skróconym formacie prezentowana graficznie:

```
git log --pretty=oneline --abbrev-commit --abbrev=4 -10 --graph
```

Interfejs graficzny

Uruchomienie programu Git GUI:

```
git gui
```

Uruchomienie programu do wizualnej prezentacji zawartości repozytorium:

```
gitk
```

Przywracanie obszaru roboczego do wybranej rewizji

Przywrócenie stanu plików w obszarze roboczym do postaci z ostatniej rewizji (polecenia nie usuwają nowych plików):

```
git reset --hard  
git checkout -f
```

Przywrócenie stanu plików w obszarze roboczym do postaci z podanej rewizji (rewizje późniejsze są usuwane z historii projektu):

```
git reset --hard [SHA-1]
```

Przywrócenie stanu plików w obszarze roboczym do postaci z podanej rewizji (po tej operacji repozytorium znajduje się w stanie detached head):

```
git checkout -f [SHA-1]
```

Stan repozytorium

Informacje o stanie repozytorium:

```
git status
```

Skrócona informacja o stanie repozytorium:

```
git status -s
```

Uproszczony model tworzenia rewizji

Tworzenie rewizji zawierającej bieżący stan wszystkich plików (pierwsza wersja):

```
git add -A  
git commit -m "komunikat..."
```

Tworzenie rewizji zawierającej bieżący stan wszystkich plików (druga wersja):

```
git add .  
git commit -a -m "komunikat..."
```

Gałęzie

Sprawdzanie listy wszystkich gałęzi oraz gałęzi bieżącej:

```
git branch
```

Przejście na wybraną gałąź:

```
git checkout [nazwa-gałezi]
```

np.

```
git checkout master  
git checkout dev
```

Usuwanie folderów .git

Usunięcie wszystkich folderów `.git` zawartych w folderze bieżącym i jego podfolderach:

```
find . -name .git -type d -exec rm -fr {} \:
```

Tworzenie rewizji (zatwierdzanie zmian)

Po wydaniu komendy:

```
git commit
```

uruchamiany jest edytor tekstu, w którym należy wprowadzić opis rewizji.

Opis rewizji możemy podać jako wartość parametru `-m`:

```
git commit -m "komunikat..."
```

Parametr `-a` powoduje automatyczną indeksację (przed wykonaniem rewizji) plików zmodyfikowanych oraz usuniętych:

```
git commit -a
```

Zmiana stanu pliku

Indeksacja pliku:

```
git add [nazwa-pliku]
```

Indeksacja wszystkich nowych plików:

```
git add .
```

Indeksacja wszystkich plików (nowych, zmodyfikowanych, usuniętych); równoważne wersje:

```
git add --all .
git add --all
git add -A .
git add -A
```

Indeksacja plików zmodyfikowanych i usuniętych:

```
git add --updated .
git add -u .
```

Usunięcie pliku:

```
git rm [nazwa-pliku]
```

Indeksacja usuwania pliku (plik pozostanie w obszarze roboczym):

```
git rm --cached [nazwa-pliku]
```

Usunięcie pliku, nawet jeśli nie jest on aktualny:

```
git rm -f nazwa-pliku
```

Zmiana nazwy pliku:

```
git mv stara-nazwa nowa-nazwa
```

Tworzenie plików

```
echo a > a.txt  
echo "Lorem..." > lorem.txt
```

Składnia poleceń Gita

Specjalny parametr `--` oddziela parametry od ścieżek:

```
git add -- --all
```

Powyższe polecenie indeksuje plik o nazwie `--all`.

Ignorowanie plików

Plik konfiguracyjny podlegający zapisowi w repozytorium:

```
.gitignore
```

Plik konfiguracyjny niepodlegający zapisywaniu w repozytorium:

```
.git/info/exclude
```

Globalny plik `.gitignore` (niezapisywany w repozytoriach):

```
sciezka\do\dowolnego\folderu\.gitignore
```

Ustalenie nazwy globalnego pliku `.gitconfig`:

```
git config --global core.excludesfile sciezka\do\dowolnego\folderu\.gitignore
```

Powyższa instrukcja tworzy wpis w globalnym pliku konfiguracyjnym:

```
[core]  
excludesfile = sciezka\\do\\dowolnego\\folderu\\.gitignore
```

Przykładowa zawartość pliku `.gitignore`:

```
#foldery  
.idea  
nbproject  
tmp  
  
#pliki  
*.oa  
*~  
*.exe
```

Znaczniki

Tworzenie znacznika opisanego dotyczącego ostatniej rewizji:

```
git tag -a NAZWA -m KOMENTARZ
```

np.

```
git tag -a v1.2.3 -m "Wydanie ver. 1.2.3"
```

Tworzenie znacznika opisanego dotyczącego dowolnej rewizji:

```
git tag -a NAZWA -m KOMENTARZ SHA-1
```

np.

```
git tag -a v4.5.6 -m "Wydanie ver. 4.5.6" aabbccdd
```

Tworzenie znacznika lekkiego dotyczącego ostatniej rewizji:

```
git tag NAZWA
```

np.

```
git tag v7.7
```

Tworzenie znacznika lekkiego dotyczącego dowolnej rewizji:

```
git tag NAZWA SHA-1
```

np.

```
git tag v3.3 ffeeddl1
```

Usuwanie znacznika (opisanego lub lekkiego):

```
git tag -d NAZWA
```

np.

```
git tag -d dev
```

Sprawdzanie dostępnych znaczników:

```
git tag
```

Sprawdzanie dostępnych znaczników wraz z datami utworzenia:

```
git log --tags --simplify-by-decoration --pretty="format:%ai %d"
```

Posortowana lista znaczników wraz z datami utworzenia:

```
git log --tags --simplify-by-decoration --pretty="format:%ai %d" | sort
```

Lista dostępnych znaczników opisanych:

```
git describe
```

Lista wszystkich dostępnych znaczników:

```
git describe --tags
```

Wyświetlenie szczegółowych danych znacznika:

```
git show -s v2.0.10
```

Zapisywanie rewizji oznaczonej znacznikiem v0.1.2 do skompresowanego archiwum:

```
git archive --format=zip --output=../projekt-0.1.2.zip v0.1.2  
git archive --format=zip v0.1.2 > ../project-0.1.2.zip
```

Dziennik reflog

Przeglądanie całego dziennika reflog:

```
git reflog
```

Przeglądanie dziennika reflog gałęzi:

```
git reflog nazwa-galezi
```

np.

```
git reflog dev
```

Czyszczenie dziennika reflog:

```
git reflog expire --all --expire=now
```

Identyfikatory rewizji

Ustalenie pełnego SHA-1 rewizji:

```
git rev-parse identyfikator
```

np.:

```
git rev-parse v0.1.2  
git rev-parse HEAD  
git rev-parse master~2  
git rev-parse abc@{3.day.ago}
```

Lista rewizji osiągalnych z podanej rewizji:

```
git rev-list id
```

np.

```
git rev-list HEAD
```

Polecenia pomocnicze ułatwiające wykonywanie ćwiczeń

Zapisywanie stanu wszystkich plików projektu w nowej rewizji:

```
git save "komunikat..."
```

Tworzenie rewizji oznaczonej komunikatem lorem i zawierającej jeden nowy plik *lorem.txt*:

```
git simple-commit lorem
```

Tworzenie kilku rewizji oznaczonych podanymi komunikatami (każda z nich zawiera jeden nowy plik o nazwie takiej jak opis rewizji):

```
git simple-commits a b c
```

Tworzenie ciągu rewizji numerowanych od 1:

```
git simple-loop a 5
```

Tworzenie ciągu numerowanych rewizji:

```
git simple-loop2 x 5 8
```

Sprawdzanie stanu repozytorium:

```
git l  
git g  
git s  
git b
```

Modyfikowanie historii projektu

Usuwanie ostatnich rewizji

```
git reset --hard X
```

Modyfikowanie ostatniej rewizji

```
git commit --amend -a -m "..."
```

Łączanie rewizji

```
git rebase -i X
```

Usuwanie zmian wprowadzonych przez rewizje

```
git revert --no-edit X
```

Odzyskiwanie pliku z wybranej rewizji

```
git checkout -f X nazwa-pliku
```

Część II

Repozytoria

z rozgałęzieniami

Rozdział 13.

Tworzenie i usuwanie gałęzi



Wskazówka

Rozważania zawarte w części II dotyczą pojedynczego repozytorium. Gałęzie zawarte w repozytorium, w którym pracujemy, określamy przyimkiem „lokalne”.

Gałęzie to wskaźniki rewizji!

Najważniejszą cechą wyróżniającą Git spośród innych programów do zarządzania wersjami jest optymalizacja pod kątem obsługi gałęzi. **Gałęzie Gita są wskaźnikami rewizji.** Dzięki temu tworzenie i usuwanie gałęzi sprowadza się do operowania skrótami SHA-1. Utworzenie gałęzi to utworzenie pliku tekstowego zawierającego skrót SHA-1.

Gałąź master

Domyślnie w nowym repozytorium utworzonym poleciением:

```
git init
```

po wykonaniu pierwszej rewizji tworzona jest jedna gałąź o nazwie master. Przekonamy się o tym, wydając polecenie:

```
git branch
```

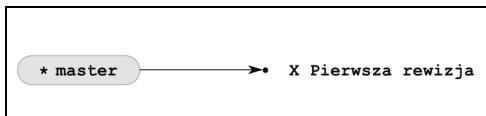
Wydruk będzie zawierał informacje:

```
* master
```

Gwiazdka informuje, że jest to gałąź, na której się obecnie znajdujemy. Bezpośrednio po wykonaniu pierwszej rewizji repozytorium będzie wyglądało tak jak na rysunku 13.1.

Rysunek 13.1.

Repozytorium po utworzeniu pierwszej rewizji



Gwiazdka widoczna na rysunkach wskazuje gałąź bieżącą.

Wskazówka

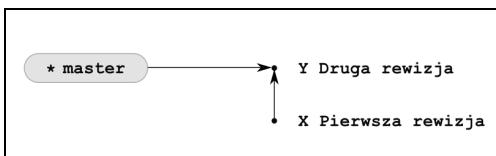
Dodajmy drugą rewizję:

```
git add -A  
git commit -m "Druga rewizja"
```

Wskaźnik master zostanie automatycznie przesunięty na drugą rewizję. Repozytorium przyjmie postać taką jak na rysunku 13.2.

Rysunek 13.2.

Repozytorium po wykonaniu drugiej rewizji



Polecenie:

```
git status -sb
```

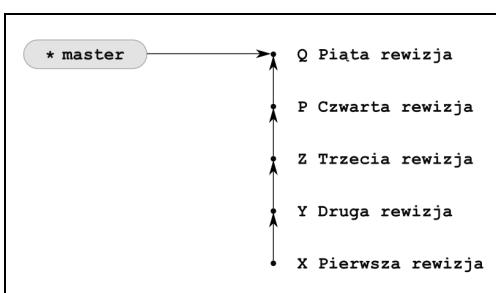
wyświetla skróconą informację o plikach repozytorium oraz nazwę gałęzi bieżącej.

Wskazówka

Za każdym razem, gdy tworzymy rewizję, wskaźnik bieżącej gałęzi jest przesuwany do ostatniej utworzonej rewizji. Po dodaniu trzech kolejnych rewizji repozytorium będzie wyglądało tak jak na rysunku 13.3.

Rysunek 13.3.

Repozytorium po wykonaniu pięciu rewizji



Wskaźnik master to skrót SHA-1 wskazywanej rewizji. Jest on zapisywany w pliku `.git/refs/heads/master`. Plik `.git/refs/heads/master` dla repozytorium z rysunku 13.1 będzie zawierał tekst X, dla repozytorium z rysunku 13.2 — tekst Y, a dla repozytorium z rysunku 13.3 — tekst Q.



Na rysunku 13.3 litery X, Y, Z, P oraz Q symbolizują skróty SHA-1 rewizji.

Wskazówka

Tworzenie gałęzi

Do tworzenia gałęzi służy polecenie:

```
git branch nazwa-gałęzi
```

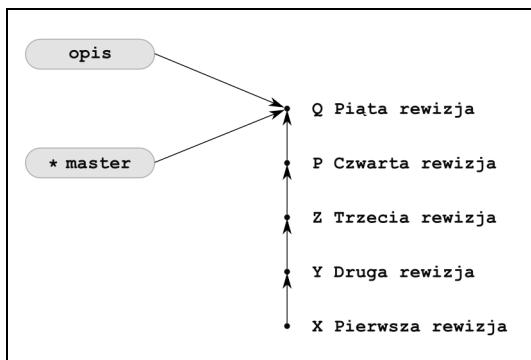
Jeśli w repozytorium z rysunku 13.3 wydamy komendę:

```
git branch opis
```

spowoduje ona utworzenie nowej gałęzi o nazwie opis. Repozytorium będzie teraz wyglądało tak jak na rysunku 13.4.

Rysunek 13.4.

Repozytorium
z rysunku 13.3
po utworzeniu
gałęzi opis



W folderze .git pojawi się plik .git/refs/heads/opis zawierający wartość SHA-1 rewizji piątej.

Sprawdźmy gałęzie repozytorium. Polecenie:

```
git branch
```

zwróci tym razem wydruk:

```
* master
  opis
```

Na podstawie tego wydruku wiemy, że w repozytorium znajdują się dwie gałęzie. Bieżącą gałęzią jest gałąź master. Informacja o bieżącej gałęzi jest zapisywana w pliku .git/HEAD. Jeśli bieżącą gałęzią jest gałąź master, to w pliku HEAD znajdziemy wpis:

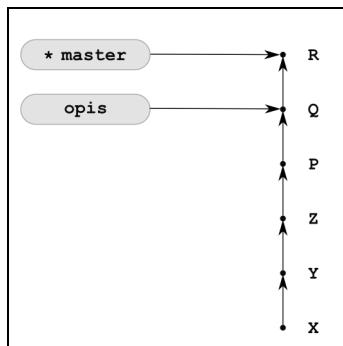
```
ref: refs/heads/master
```

Dodawanie rewizji w bieżącej gałęzi

Co się stanie, jeśli teraz utworzymy nową rewizję? Gałąź bieżąca, czyli master, zostanie przesunięta na dodaną rewizję, a gałąź opis pozostanie niezmieniona. Rysunek 13.5 ilustruje stan repozytorium po wykonaniu szóstej rewizji R.

Rysunek 13.5.

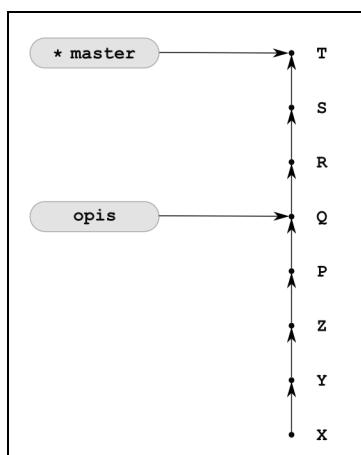
*Repozytorium
z rysunku 13.4
po wykonaniu rewizji
R w gałęzi master*



Dodajmy w gałęzi master kolejne dwie rewizje: S oraz T. W tym celu wystarczy wykonać rewizje, gdyż nowe rewizje są automatycznie dodawane do bieżącej gałęzi. Otrzymamy repozytorium widoczne na rysunku 13.6.

Rysunek 13.6.

*Repozytorium
z rysunku 13.5
po wykonaniu rewizji
S oraz T*



Tworzenie gałęzi wskazujących dowolną rewizję

Gałązie możemy tworzyć, wskazując dowolną rewizję. W celu utworzenia nowej gałęzi o nazwie `proba` wskazującej na rewizję Z należy wydać polecenie:

```
git branch nazwa-gałęzi SHA-1
```

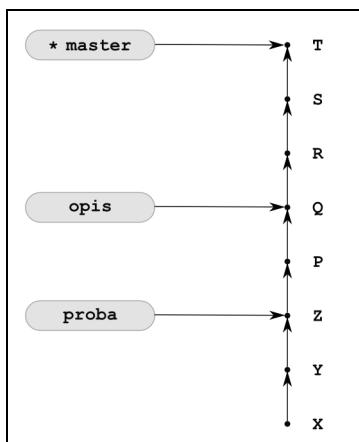
Po wydaniu komendy:

```
git branch proba Z
```

repozytorium będzie wyglądało tak jak na rysunku 13.7.

Rysunek 13.7.

Repozytorium
z rysunku 13.6
po wydaniu polecenia
git branch proba Z



Oczywiście w folderze *.git* pojawi się plik *.git/refs/heads/proba* zawierający wartość SHA-1 rewizji oznaczonej na rysunku symbolem *Z*. Gałąź bieżąca nie ulegnie zmianie, o czym przekonamy się, wydając polecenie:

```
git branch
```

Wydruk będzie następujący:

```
* master
  opis
  proba
```

Porządek gałęzi na powyższym wydruku jest alfabetyczny.

Przełączanie gałęzi

Do zmiany gałęzi bieżącej służy komenda:

```
git checkout nazwa-gałęzi
```

Po wydaniu tej komendy bieżącą gałęzią stanie się gałąź o podanej nazwie, a pliki w obszarze roboczym przyjmą postać z ostatniej rewizji w tej gałęzi.



Wskazówka W początkowym okresie nauki komendę przełączania gałęzi najlepiej wydawać wyłącznie wtedy, gdy wszystkie pliki są aktualne (tj. gdy polecenie *git status -s* zwraca pusty wynik).

Po wydaniu komendy:

```
git checkout proba
```

pliki obszaru roboczego będą odpowiadały stanowi z rewizji *Z*. Ponadto polecenie:

```
git branch
```

zwróci wydruk:

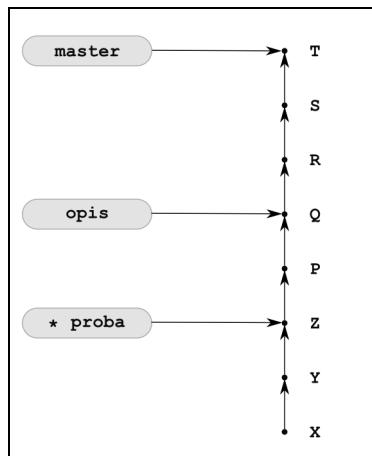
```
master
opis
* proba
```

Gwiazdka przy gałęzi proba informuje, że jest to gałąź bieżąca. Repozytorium będzie więc w stanie takim jak na rysunku 13.8. W pliku *.git/HEAD* znajdziemy wpis:

```
ref: refs/heads/proba
```

Rysunek 13.8.

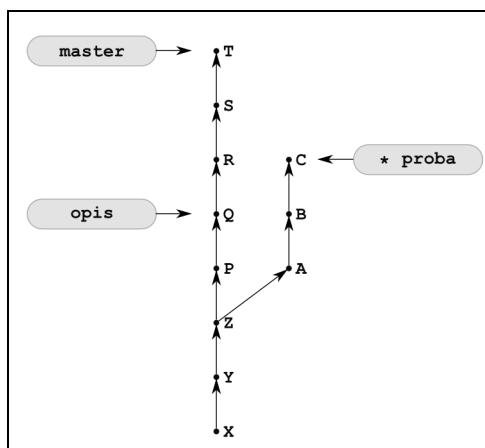
*Repozytorium
z rysunku 13.7
po wydaniu polecenia
git branch proba*



Jeśli teraz wykonamy rewizje A, B i C, to trafią one do gałęzi proba. Stan repozytorium po wykonaniu rewizji A, B i C jest przedstawiony na rysunku 13.9.

Rysunek 13.9.

*Repozytorium
z rysunku 13.8
po dodaniu rewizji
A, B i C*



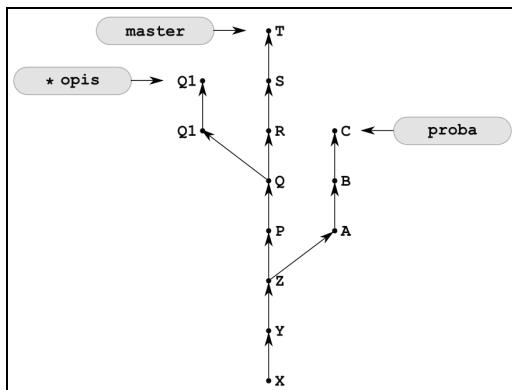
Przejdzmy na gałąź opis:

```
git checkout opis
```

i dodajmy rewizje Q1 i Q2. Repozytorium przyjmie postać przedstawioną na rysunku 13.10.

Rysunek 13.10.

Repozytorium z rysunku 13.9 po dodaniu w gałęzi opis rewizji Q1 i Q2



Polecenie:

Wskazówka

`git checkout nazwa-gałęzi`

służy do przełączania gałęzi w repozytoriach zwykłych. W repozytorium surowym gałęzie przełączamy komendą:

`git symbolic-ref HEAD refs/heads/nazwa-gałęzi`

Tworzenie i przełączanie gałęzi

Operację tworzenia i przełączania gałęzi możemy wykonać jednym poleceniem:

`git checkout -b nazwa-gałęzi`

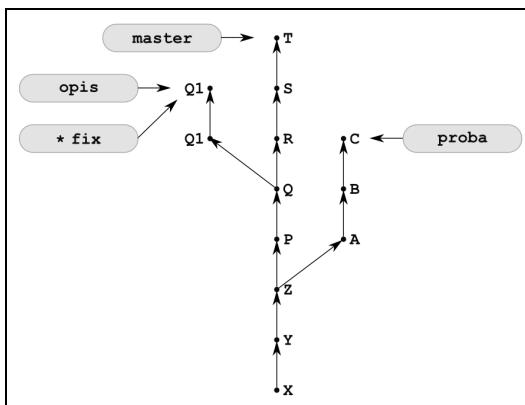
Jeśli w repozytorium w stanie z rysunku 13.10 wydamy komendę:

`git checkout -b fix`

otrzymamy wówczas repozytorium takie jak na rysunku 13.11. W repozytorium pojawi się gałąź `fix` i będzie to gałąź bieżąca.

Rysunek 13.11.

Repozytorium z rysunku 13.10 po wydaniu polecenia git checkout -b fix



Nowo utworzona gałąź może wskazywać dowolną rewizję. Służy do tego polecenie:

```
git checkout -b nazwa-gałęzi SHA-1
```

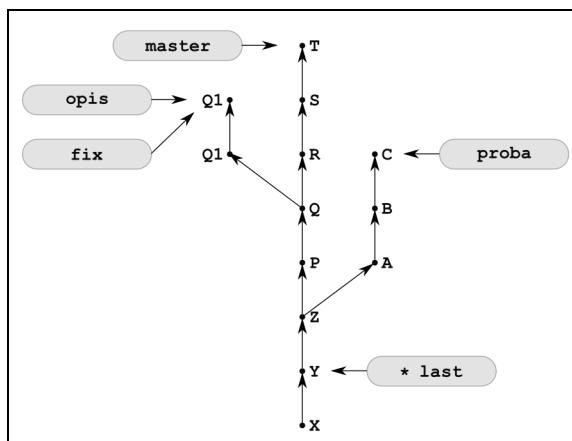
W celu utworzenia gałęzi last wskazującej rewizję Y należy wydać komendę:

```
git checkout -b last Y
```

Powyższa komenda przekształci repozytorium z rysunku 13.11 do postaci widocznej na rysunku 13.12.

Rysunek 13.12.

Repozytorium
z rysunku 13.11
po wydaniu polecenia
`git checkout -b last Y`



Stan detached HEAD

Jeśli pliki z obszaru roboczego przywrócić do stanu z konkretnej rewizji, posługując się jej skrótem SHA-1, tak jak to robiliśmy w rozdziale 5.:

```
git checkout SHA-1
```

wówczas repozytorium będzie znajdowało się w stanie określonym w dokumentacji terminem *detached HEAD*. W tym momencie żadna gałąź nie jest gałęzią bieżącą. W pliku `.git/HEAD` zapisany jest skrót SHA-1 konkretnej rewizji, a nie nazwa symboliczna gałęzi (np. ref: `refs/heads/master`).

Jeśli w repozytorium z rysunku 13.12 wydamy komendę:

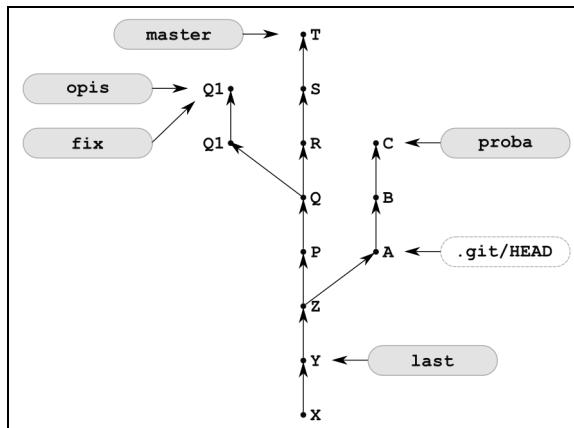
```
git checkout A
```

wówczas repozytorium przyjmie stan z rysunku 13.13¹. Pliki w obszarze roboczym będą odpowiadały rewizji A, a w pliku `.git/HEAD` zapisany będzie skrót SHA-1 rewizji A.

¹ Zwróć uwagę na brak gwiazdki na rysunku.

Rysunek 13.13.

Repozytorium
z rysunku 13.12
po wydaniu polecenia
git checkout A



Polecenie:

git branch

zwróci wynik:

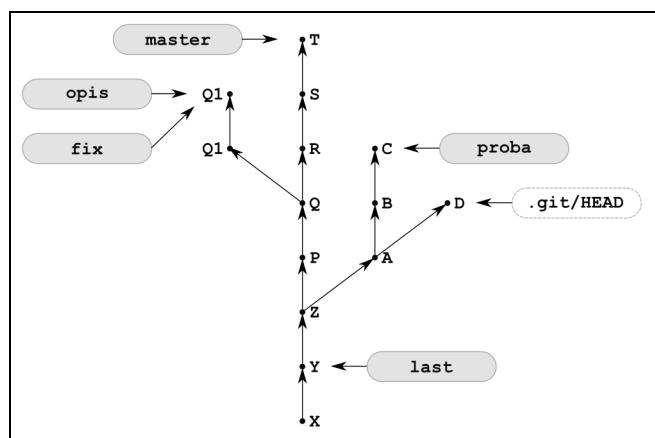
- * (no branch)
- fix
- last
- master
- opis
- proba

W takim stanie najlepiej nie wprowadzać żadnych modyfikacji w repozytorium (tj. nie tworzyć nowych rewizji), gdyż można je utracić.

Co się stanie, jeśli mimo ostrzeżenia wykonamy nową rewizję? W repozytorium pojawi się rewizja, do której nie można dotrzeć, stosując gałąź. Rewizja ta zostanie dodana jako dziecko rewizji wskazanej przez HEAD. Jeśli w repozytorium z rysunku 13.13 wykonamy nową rewizję D, postać repozytorium będzie taka jak na rysunku 13.14.

Rysunek 13.14.

Repozytorium
z rysunku 13.13
po wykonaniu
rewizji D



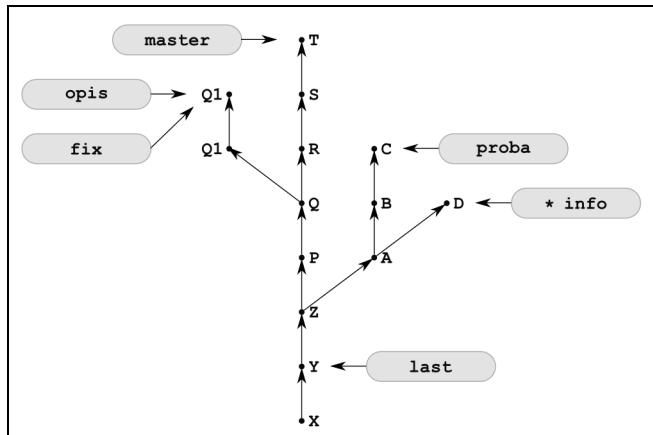
Dla nowej rewizji D możemy utworzyć gałąź. Po wydaniu polecenia:

```
git checkout -b info
```

powstanie gałąź o nazwie `info` wskazująca rewizję D. Będzie to gałąź bieżąca. Otrzymamy repozytorium takie jak na rysunku 13.15.

Rysunek 13.15.

Repozytorium
z rysunku 13.14
po utworzeniu
gałęzi info



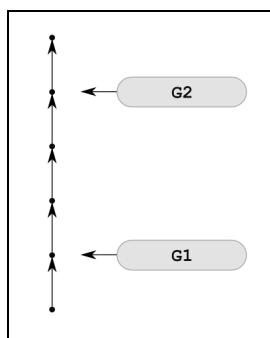
Do rewizji takich jak widoczna na rysunku 13.14 rewizja D możemy dotrzeć, stosując odwołania zapisane w dzienniku reflog.

Relacja zawierania gałęzi

Gałęzie repozytorium możemy scharakteryzować relacją zawierania. Mówimy, że gałąź G1 jest **zawarta w gałęzi** (ang. *merged*) G2 wtedy, gdy wszystkie rewizje zawarte w gałęzi G1 występują w gałęzi G2. Rysunek 13.17 przedstawia gałąź G1, która jest zawarta w gałęzi G2.

Rysunek 13.16.

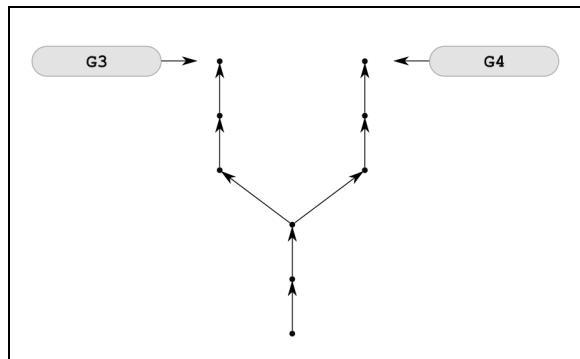
Gałąź G1 jest zawarta
w gałęzi G2



Rysunek 13.17 ilustruje dwie gałęzie: G3 i G4. Na rysunku tym gałąź G3 nie zawiera gałęzi G4 i gałąź G4 nie zawiera gałęzi G3. Gałęzie takie nazwiemy **rozłącznymi**.

Rysunek 13.17.

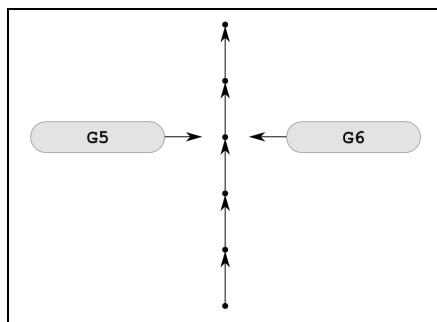
Rozłączne gałęzie G3 oraz G4: gałąź G3 nie jest zawarta w gałęzi G4, gałąź G4 nie jest zawarta w gałęzi G3



Relacja zawierania gałęzi zazwyczaj nie jest relacją zwrotną. Dla gałęzi z rysunku 13.17 gałąź G1 jest zawarta w gałęzi G2, lecz gałąź G2 nie jest zawarta w gałęzi G1. Jednym przypadkiem, w którym relacja ta jest zwrotna, jest sytuacja, gdy obie gałęzie wskazują tę samą rewizję. Przykład taki jest przedstawiony na rysunku 13.18.

Rysunek 13.18.

Gałąź G5 jest zawarta w gałęzi G6, gałąź G6 jest zawarta w gałęzi G5



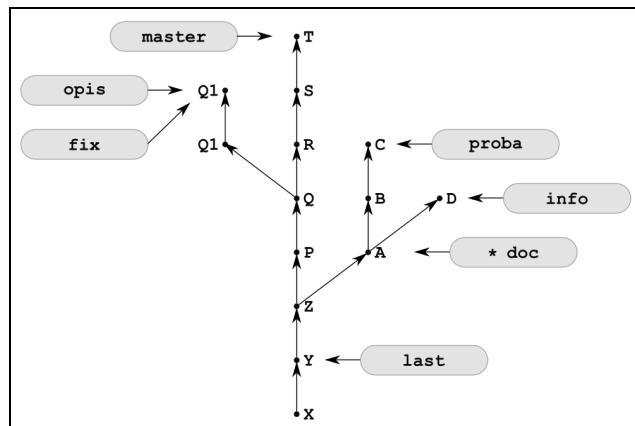
Zmodyfikujmy repozytorium z rysunku 13.15, dodając gałąź doc wskazującą rewizję A:

```
git checkout -b doc A
```

Otrzymamy repozytorium przedstawione na rysunku 13.19.

Rysunek 13.19.

Repozytorium z rysunku 13.15 po utworzeniu gałęzi doc wskazującej rewizję A



W repozytorium z rysunku 13.19:

- ◆ Gałąź master zawiera gałąź last.
- ◆ Gałąź opis zawiera gałęzie fix i last.
- ◆ Gałąź fix zawiera gałęzie opis i last.
- ◆ Gałąź proba zawiera gałęzie doc i last.
- ◆ Gałąź info zawiera gałęzie doc i last.
- ◆ Gałąź doc zawiera gałąź last.
- ◆ Gałąź last nie zawiera żadnej gałęzi.

O tym, które gałęzie są zawarte w gałęzi bieżącej, informują polecenia:

```
git branch --merged
git branch --no-merged
```

Pierwsze z nich wyświetla listę gałęzi, które są zawarte w gałęzi bieżącej, a drugie — listę gałęzi, które nie są zawarte w gałęzi bieżącej. Jeśli w poleceniach tych dodamy identyfikator rewizji:

```
git branch -merged SHA-1
git branch --no-merged SHA-1
```

wówczas poznamy listę gałęzi zawartych (`--merged`) lub niezawartych (`--not-merged`) w podanej rewizji.

W repozytorium z rysunku 13.19 bieżącą gałęzią jest doc. Polecenie²:

```
git branch --merged
```

zwróci wydruk:

```
last
```

Wynikiem polecenia:

```
git branch --no-merged
```

będzie:

```
fix
info
master
opis
proba
```

Polecenie:

```
git branch --merged B
```

będzie dotyczyło gałęzi osiągalnych z rewizji B. Wynikiem będzie lista:

```
doc
last
```

² Polecenie `git branch --merged` jest równoważne poleceniu `git branch --merged HEAD`.

Polecenie:

```
git branch --no-merged B
```

wygeneruje natomiast wydruk:

```
fix  
info  
master  
opis  
proba
```

Usuwanie gałęzi

Git rozróżnia dwa przypadki usuwania gałęzi:

- ♦ usuwanie gałęzi, które nie powoduje usuwania rewizji;
- ♦ usuwanie gałęzi, które pociąga za sobą usuwanie rewizji.

Przypadki te są rozróżniane na podstawie relacji zawierania usuwanej gałęzi względem gałęzi bieżącej lub śledzonej. Gałąź śledzona dotyczy repozytoriów, które są synchronizowane z innymi repozytoriami. Takim przypadkiem zajmiemy się, gdy będziemy omawiali publikowanie repozytorium na serwerze. Na razie ograniczmy się do przypadku repozytorium, które nie jest synchronizowane z żadnym innym repozytorium (tj. nie ma gałęzi śledzących gałęzie zdalne).

Usuwanie gałęzi zawartych

Polecenie:

```
git branch -d nazwa-gałezi
```

powoduje usunięcie gałęzi o podanej nazwie. Polecenie to działa poprawnie wyłącznie wtedy, gdy usuwana gałąź jest zawarta w gałęzi bieżącej.

Dla repozytorium z rysunku 13.19 polecenie:

```
git branch -d last
```

zakończy się sukcesem, gdyż gałąź last jest zawarta w gałęzi bieżącej doc. Otrzymamy repozytorium z rysunku 13.20.



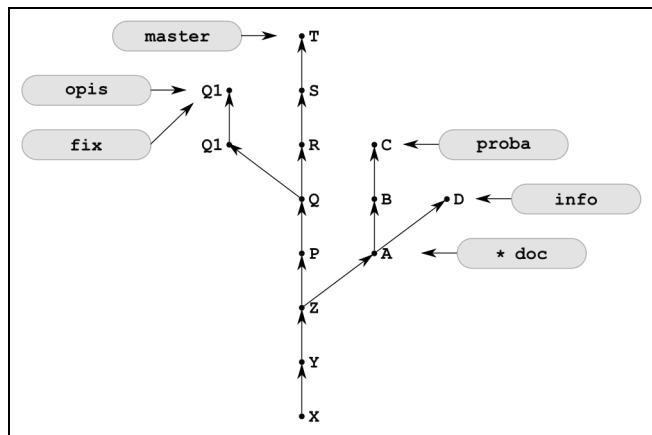
Gałęzi bieżącej nie można usunąć.

Gałąź master możemy usunąć tak jak każdą inną gałąź.

Ponieważ gałąź proba nie jest zawarta w gałęzi bieżącej doc, polecenie:

```
git branch -d proba
```

Rysunek 13.20.
Repozytorium
z rysunku 13.19
po usunięciu
gałęzi last



zakończy się błędem:

```
error: The branch 'proba' is not fully merged.
```

Usuwanie gałęzi rozłącznych

Polecenie:

```
git branch -D nazwa-gałęzi
```

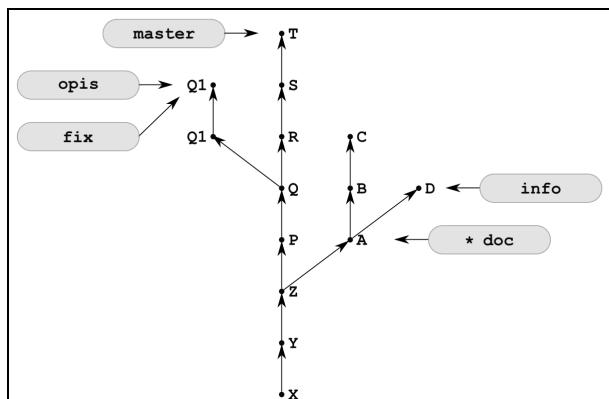
usuwa gałąź o podanej nazwie bez względu na to, czy jest ona zawarta w gałęzi bieżącej, czy nie.

Dla repozytorium z rysunku 13.20 polecenie:

```
git branch -D proba
```

da wynik przedstawiony na rysunku 13.21.

Rysunek 13.21.
Repozytorium
z rysunku 13.20
po wydaniu polecenia
git branch -D proba



Teraz rewizje B oraz C nie są zawarte w żadnej gałęzi. Możemy do nich dotrzeć wyłącznie z użyciem skrótów SHA-1 lub odwołań zawartych w dzienniku reflog.



Ostrzeżenie

Uwaga: rewizje B i C z rysunku 13.21 oraz rewizję D z rysunku 13.14 możemy bezpowrotnie utracić! Po okresie dwóch tygodni lub jednego miesiąca zostaną one automatycznie usunięte z repozytorium. Czas przechowywania rewizji zależy od wykonanych operacji oraz od ustawień konfiguracyjnych. Szczegółowy opis tych zagadnień jest zawarty w punkcie „Zgubione rewizje”.

Zmiana nazwy gałęzi

Do zmiany nazwy gałęzi służy polecenie:

```
git branch -m stara-nazwa nowa-nazwa
```

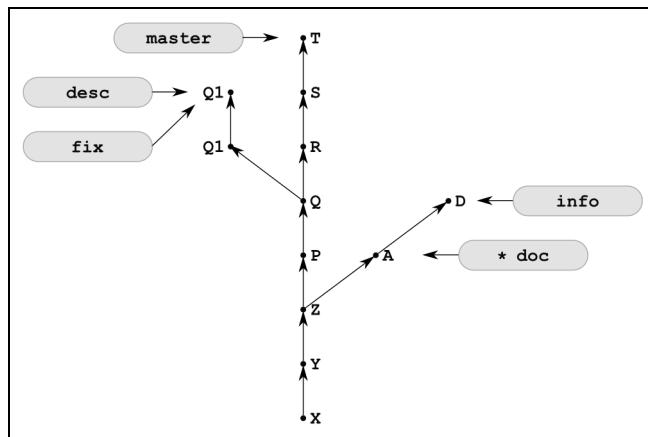
Jeśli w repozytorium z rysunku 13.21 wydamy komendę:

```
git branch -m opis desc
```

to otrzymamy repozytorium z rysunku 13.22.

Rysunek 13.22.

Repozytorium
z rysunku 13.21
po wydaniu
polecenia git
branch -m opis desc



Jeżeli w repozytorium istnieje już gałąź o nazwie *nowa-nazwa*, to polecenie git branch -m zakończy się błędem:

```
fatal: A branch named 'nowa-nazwa' already exists.
```

W takim przypadku możemy użyć polecenia:

```
git branch -M stara-nazwa nowa-nazwa
```

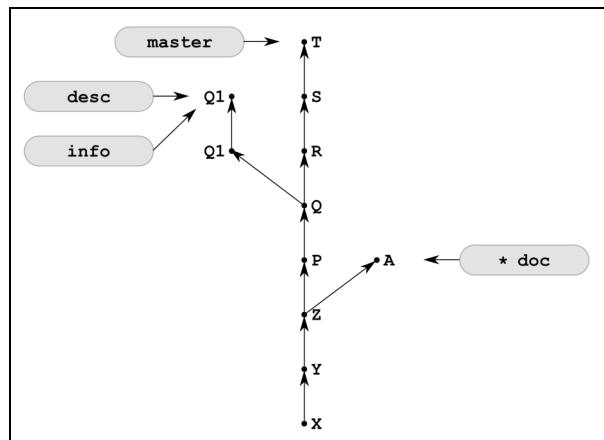
które mimo wszystko wymusza zmianę nazwy gałęzi. Polecenie to spowoduje usunięcie gałęzi *nowa-nazwa* oraz zmianę nazwy gałęzi *stara-nazwa* na *nowa-nazwa*.

Jeśli w repozytorium z rysunku 13.22 wydamy komendę:

```
git branch -M fix info
```

otrzymamy repozytorium przedstawione na rysunku 13.23. Rewizja D zostanie utracona.

Rysunek 13.23.
Repozytorium
z rysunku 13.22
po wydaniu
poleceń git
`branch -M fix info`



Gałęzie jako identyfikatory rewizji

Gałęzie są kolejną metodą identyfikowania rewizji. Od opisanych w rozdziale 8. znaczników odróżnia je to, że są one ruchome. Znacznik zawsze wskazuje jedną i tą samą rewizję, podczas gdy gałąź jest automatycznie przesuwana przy tworzeniu nowych rewizji.

W odniesieniu do gałęzi możemy stosować odwołania do przodków n -tej generacji oraz do n -tego rodzica. Dla repozytorium z rysunku 13.23 rewizję oznaczoną literą Y możemy wskazać na wiele sposobów:

```

HEAD~2
HEAD~~
HEAD^^
doc~2
doc~~
doc^^
master~6
master^^^^^
desc~5
desc^^^^
info~5
info^^^^
  
```

Uwaga Jeśli odwołanie `master^^^^^` zechcesz zapisać w wierszu poleceń Windows, będziesz musiał użyć dwudziestu czterech znaków ^!

Uwagi o usuwaniu ostatnich rewizji

Opisane w rozdziale 11. polecenie:

```
git reset --hard SHA-1
```

usuwa rewizje wyłącznie z bieżącej gałęzi. Dlatego jeśli w repozytorium z rysunku 13.23 wydamy komendy:

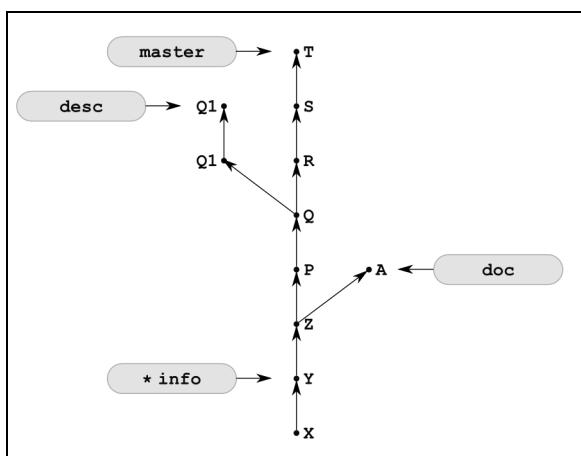
```
//przechodzimy do gałęzi info  
git checkout info
```

```
//usuwamy ostatnie pięć rewizji w gałęzi info  
git reset -hard HEAD~5
```

otrzymamy repozytorium z rysunku 13.24. Rewizje Q1, Q1, P, Q oraz Z pozostaną w gałęzi desc.

Rysunek 13.24.

Repozytorium z rysunku 13.23 po usunięciu pięciu ostatnich rewizji w gałęzi info



Sprawdzanie różnic pomiędzy gałęziami

Do sprawdzania różnic pomiędzy dwoma dowolnymi rewizjami służy polecenie git diff. Możemy je wykorzystać do porównania dwóch gałęzi. Polecenie:

```
git diff --name-status desc..doc
```

wyświetli listę plików, którymi różnią się gałęzie desc oraz doc.

Ćwiczenia

W każdym z poniższych ćwiczeń zastosuj następującą konwencję: rewizja oznaczona symbolem X ma mieć komunikat X i zawierać jeden nowy plik o nazwie *X.txt*. W pliku *X.txt* wprowadź tekst X. Konwencję taką stosowaliśmy już w ćwiczeniu 6.3. Tworzenie rewizji X ułatwi Ci omówione w rozdziale 10. polecenie:

```
git simple-commit X
```

Tworzenie wielu takich rewizji możesz uprościć, stosując pozostałe polecenia z rozdziału 10.:

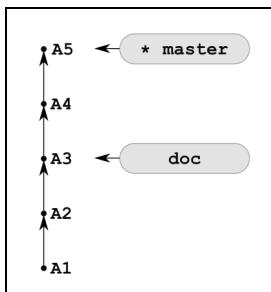
```
git simple-commits A B C
git simple-loop X 5
git simple-loop Y 8 15
```

Ćwiczenie 13.1

Wykonaj repozytorium przedstawione na rysunku 13.25.

Rysunek 13.25.

Repozytorium
z ćwiczenia 13.1



ROZWIĄZANIE

W folderze *cw-13-01/* utwórz nowe repozytorium. Następnie wydaj polecenie:

```
git simple-loop A 5
```

Następnie utwórz gałąź *doc* wskazującą na rewizję A3:

```
git branch doc master~2
```

Na zakończenie ćwiczenia poleceniami:

```
git checkout master
git checkout master~
git checkout master~2
git checkout master~3
git checkout master~4
git checkout doc
git checkout doc~
git checkout doc~2
```

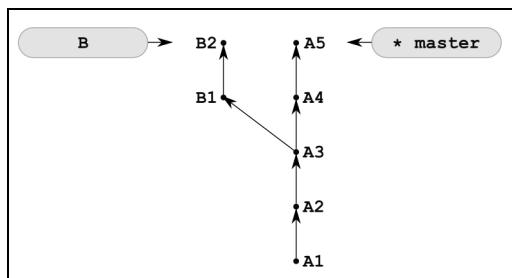
sprawdź, czy każda z rewizji zawiera odpowiednie pliki. Np. rewizja A4 powinna zawierać pliki: *A1.txt*, *A2.txt*, *A3.txt* oraz *A4.txt*.

Ćwiczenie 13.2

Wykonaj repozytorium przedstawione na rysunku 13.26.

Rysunek 13.26.

Repozytorium z ćwiczenia 13.2



ROZWIĄZANIE

W folderze *cw-13-02/* utwórz nowe repozytorium zawierające rewizje A1, A2, A3, A4 i A5.

Następnie utwórz gałąź B wskazującą na rewizję A3:

```
git checkout -b B HEAD~2
```

Teraz gałęzią bieżącą jest gałąź B. Wykonaj rewizje B1 i B2. Na zakończenie zmień gałąź na master:

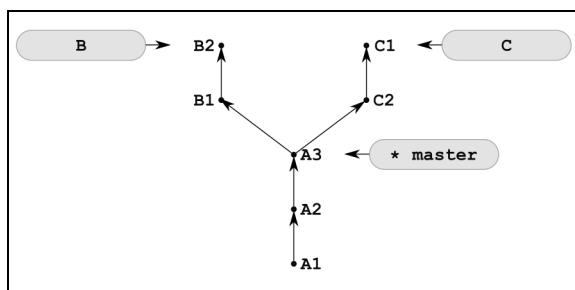
```
git checkout master
```

Ćwiczenie 13.3

Wykonaj repozytorium przedstawione na rysunku 13.27.

Rysunek 13.27.

Repozytorium z ćwiczenia 13.3



ROZWIĄZANIE

W folderze *cw-13-03/* utwórz nowe repozytorium i wykonaj rewizje A1, A2 i A3.

Następnie utwórz gałąź B:

```
git checkout -b B
```

Teraz gałęzią bieżącą jest gałąź B, która wskazuje na rewizję A3. Wykonaj rewizje B1 i B2.

Wróć na gałąź master:

```
git checkout master
```

i utwórz gałąź C:

```
git checkout -b C
```

Gałęzią bieżącą jest C. Wskazuje ona na rewizję A3. Wykonaj rewizje C1 i C2, po czym wróć na gałąź master:

```
git checkout master
```

Poleceniami:

```
git checkout B
git checkout B~
git checkout B~2
git checkout C
git checkout C~
git checkout C~2
git checkout master
git checkout master~
git checkout master~2
```

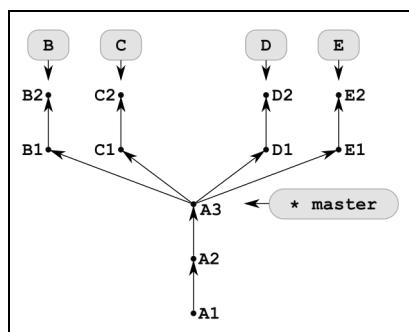
sprawdź, czy każda z rewizji zawiera odpowiednie pliki.

Ćwiczenie 13.4

Wykonaj repozytorium przedstawione na rysunku 13.28.

Rysunek 13.28.

Repozytorium
z ćwiczenia 13.4



ROZWIĄZANIE

W folderze *cw-13-04/* utwórz nowe repozytorium i wykonaj rewizje A1, A2 i A3.

Następnie utwórz gałąź B:

```
git checkout -b B
```

W utworzonej gałęzi wykonaj rewizje B1 i B2.

Wróć na gałąź master:

```
git checkout master
```

i utwórz gałąź C:

```
git checkout -b C
```

Wykonaj w niej rewizje C1 i C2.

Ponownie wróć na gałąź master, utwórz gałąź D i wykonaj w niej rewizje D1 i D2.

W identyczny sposób wykonaj gałąź E.

Na zakończenie wróć na gałąź master.

Ćwiczenie 13.5

Wykonaj repozytorium przedstawione na rysunku 13.20.

Gałęzie i dziennik reflog

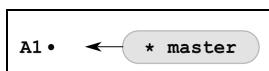
Podobnie jak tworzenie rewizji, tak i operacje zmiany gałęzi są zapisywane w dzienniku reflog.

W nowym repozytorium utwórzmy rewizję A1. Dla repozytorium z rysunku 13.29 polecenie git reflog zwróci wynik:

```
a1da48 HEAD@{0}: commit (initial): A1
```

Rysunek 13.29.

Repozytorium zawierające jedną rewizję

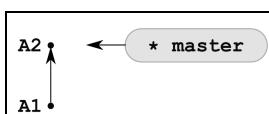


Dodajmy rewizję A2. Repozytorium przyjmie postać z rysunku 13.30. W dzienniku reflog pojawi się nowy wpis:

```
02a22c4 HEAD@{0}: commit: A2
```

Rysunek 13.30.

Repozytorium zawierające dwie rewizje



Po utworzeniu nowej gałęzi doc poleceniem:

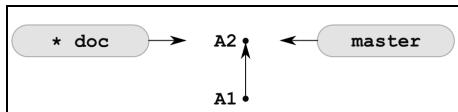
```
git checkout -b doc
```

repozytorium przyjmie postać z rysunku 13.31, a w dzienniku reflog dodany zostanie wiersz:

```
02a22c4 HEAD@{0}: checkout: moving from master to doc
```

Rysunek 13.31.

Repozytorium po przejściu na nową gałąź doc

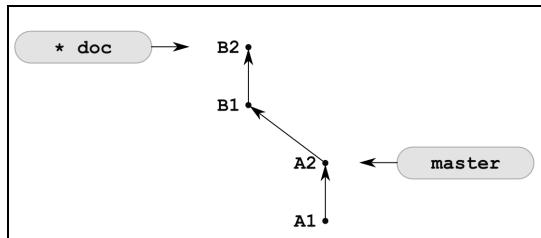


Dodajmy w gałęzi doc dwie rewizje: B1 oraz B2. Otrzymamy repozytorium z rysunku 13.32. W dzienniku reflog pojawią się dwa nowe wpisy:

```
25ead2c HEAD@{0}: commit: b2
ccad7e7 HEAD@{1}: commit: b1
```

Rysunek 13.32.

Repozytorium po dodaniu w gałęzi doc dwóch rewizji: B1 i B2



Zawartość dziennika reflog dla repozytorium z rysunku 13.32 jest przedstawiona na listingu 13.1.

Listing 13.1. Zawartość dziennika reflog dla repozytorium z rysunku 13.32

```
25ead2c HEAD@{1}: commit: b2
ccad7e7 HEAD@{2}: commit: b1
02a22c4 HEAD@{3}: checkout: moving from master to doc
02a22c4 HEAD@{4}: commit: a2
af1da48 HEAD@{5}: commit (initial): a1
```

Wpisy w dzienniku reflog pojawiają się za każdym razem, gdy zmieniamy bieżące położenie w repozytorium. Dodawanie nowych pozycji w dzienniku reflog powodują m.in. polecenia:

```
git commit
git checkout
```

Polecenie:

```
git commit
```

dodaje w dzienniku reflog wpis:

```
SHA-1 HEAD@{1}: commit : komunikat rewizji
```

Polecenie:

```
git checkout
```

tworzy wpis:

```
SHA-1 HEAD@{1}: checkout: moving from NAZWA to NAZWA
```

Git tworzy jeden dziennik główny oraz osobny dziennik dla każdej gałęzi. Wszystkie dzienniki reflog są tworzone w folderze `.git/logs/`. Polecenie:

```
git reflog
```

zwraca informacje z głównego dziennika. W celu wydrukowania zawartości dziennika reflog gałęzi doc należy wydać komendę:

```
git reflog show doc
```

Dziennik główny jest zapisywany w pliku `.git/logs/HEAD`.

Zgubione rewizje

W repozytorium z rysunku 13.32 przejdźmy do rewizji A1:

```
git checkout master~1
```

Znajdziemy się w stanie detached HEAD. Utwórzmy teraz rewizję C1:

```
echo c1>c1.txt
git add -A
git commit -m c1
```

Powróćmy na gałąź master:

```
git checkout master
```

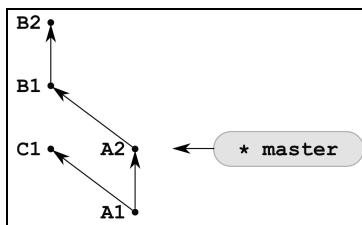
i usuńmy gałąź doc:

```
git branch -D doc
```

Otrzymamy repozytorium przedstawione na rysunku 13.33.

Rysunek 13.33.

Repozytorium
z rysunku 13.32
po dodaniu rewizji C1
i usunięciu gałęzi doc



Rewizje, które są nieosiągalne za pośrednictwem gałęzi, nazwiemy **zgubionymi** (ang. *dangling*). Na rysunku 13.33 zgubionymi rewizjami są: C1, B1 oraz B2. Rewizja C1 jest

zgubiona, gdyż została utworzona w stanie detached HEAD. Rewizje B1 oraz B2 są zgubione w wyniku usunięcia gałęzi doc.

Stan przedstawiony na rysunku jest bardzo ryzykowny, gdyż rewizje zgubione są automatycznie usuwane z repozytorium. Czas, po jakim nastąpi nieodwracalne usunięcie zgubionych rewizji, jest ustalany dwoma parametrami konfiguracyjnymi:

```
gc.pruneexpire  
gc.reflogexpireunreachable
```

Domyślna wartość tych parametrów wynosi:

gc.pruneexpire	2 tygodnie
gc.reflogexpireunreachable	30 dni

Poleceniami:

```
git config gc.pruneexpire now  
git config gc.reflogexpireunreachable now
```

możemy skrócić okres przechowywania zgubionych rewizji do 0.

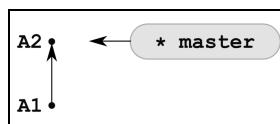
Jeśli teraz wydamy polecenie:

```
git prune
```

wówczas repozytorium przyjmie postać widoczną na rysunku 13.34. Rewizje C1, B1 oraz B2 zostały bezpowrotnie utracone.

Rysunek 13.34.

Repozytorium z rysunku 13.33 po wydaniu polecenia git prune



Polecenie git prune trwale usuwa z repozytorium wszystkie obiekty, które nie są dostępne przy użyciu nazw symbolicznych. Polecenie to jest wykonywane jako jeden z etapów pracy polecenia git gc, które optymalizuje strukturę repozytorium. Polecenie git gc jest automatycznie uruchamiane przez niektóre komendy Gita, dlatego operacja usuwania zgubionych rewizji może zostać wykonana bez intencjonalnego wydania komendy:

```
git prune
```

Uwaga
Jako ważny wniosek wynikający z rysunków 13.33 oraz 13.34 zapamiętaj, że rewizje niedostępne za pośrednictwem gałęzi mogą trwale zniknąć z repozytorium. Nigdy nie pracuj w stanie detached HEAD.

Zakładając, że repozytorium znajduje się w stanie z rysunku 13.33, w jaki sposób możemy odzyskać zgubione rewizje? Wystarczy ustalić skrót SHA-1 rewizji, którą chcemy odzyskać, i wydać komendę:

```
git checkout -b nowa-gałaz SHA-1
```

W ten sposób utworzymy nową gałąź wskazującą wybraną rewizję. Dzięki temu rewizja nie zostanie objęta operacjami:

```
git prune  
git gc
```

Skróty SHA-1 wszystkich rewizji, które jeszcze możemy odzyskać, znajdziemy w dziennikach reflog.

Ćwiczenie 13.6

Wykonaj repozytorium przedstawione na rysunku 13.22. Po każdej komendzie sprawdź zawartość dziennika reflog.

Ćwiczenie 13.7

Zgodnie z opisem podanym w punkcie „Zgubione rewizje” doprowadź repozytorium z ćwiczenia 13.6 do postaci z rysunku 13.33. Po każdej komendzie sprawdź zawartość dziennika reflog.

Na zakończenie poleceniami:

```
git config gc.pruneexpire now  
git config gc.reflogexpireunreachable now
```

oraz

```
git prune  
usuń zgubione rewizje.
```

Ćwiczenie 13.8

Zaprogramuj skróconą komendę git branches, która będzie tworzyła kilka gałęzi wskazujących bieżącą rewizję. Wywołanie:

```
git branches a b c  
ma powodować utworzenie gałęzi a, b i c.
```

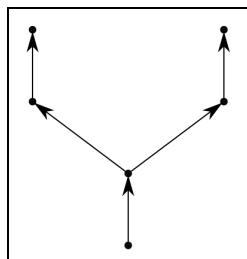

Rozdział 14.

Łączenie gałęzi: operacja merge

W wyniku tworzenia gałęzi i dodawania w nich rewizji powstaje repozytorium, które ma strukturę drzewa. Repozytorium takie jest przedstawione na rysunku 14.1.

Rysunek 14.1.

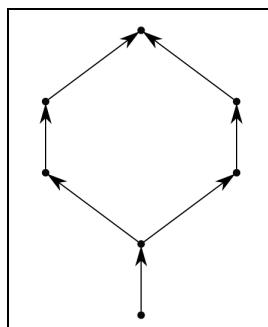
*Repozytorium
zawierające
rozłączne gałęzie*



Teraz zajmiemy się łączeniem gałęzi. Repozytorium z rysunku 14.1 będziemy przekształcali do postaci przedstawionej na rysunku 14.2. Proces łączenia gałęzi powoduje pojawienie się w repozytorium rewizji, która mają kilku rodziców. Odwołania ^ omówione w rozdziale 9. dotyczą węzłów powstających po połączeniu gałęzi.

Rysunek 14.2.

*Repozytorium
z rysunku 14.1
po połączeniu gałęzi*



Przebieg łączenia gałęzi omówimy, wyróżniając dwa przypadki:

- ◆ przewijanie do przodu (ang. *fast forward*),
- ◆ łączenie gałęzi rozłącznych.



Uwaga

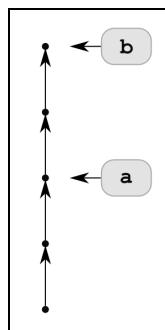
Podczas łączenia gałęzi mogą wystąpić konflikty. Ćwiczenia omówione w tym rozdziale wykorzystują repozytoria takie jak w ćwiczeniu 6.3. Ponieważ każda rewizja zawiera jeden nowy plik o unikalnej nazwie, kolizje się nie pojawiają. Kolizje i sposoby ich rozwiązywania omówimy w części IV.

Przewijanie do przodu

Łączenie określane terminem **przewijania do przodu** (ang. *fast forward*) dotyczy gałęzi, z których jedna jest zawarta w drugiej. Przyjmijmy, że repozytorium znajduje się w stanie przedstawionym na rysunku 14.3. Zawiera ono dwie gałęzie: a i b. Gałąź a jest zawarta w gałęzi b.

Rysunek 14.3.

Repozytorium,
w którym gałąź a
jest zawarta
w gałęzi b



W takim przypadku możemy mówić o **dolaczaniu zawartości gałęzi b do gałęzi a**. Operację taką rozpoczynamy od przejścia na gałąź a:

```
git checkout a
```

Łączenie wykonujemy komendą:

```
git merge b
```

Po wykonaniu powyższej komendy gałąź a wskazuje tę samą rewizję co gałąź b. Wydruk będzie zawierał komunikat:

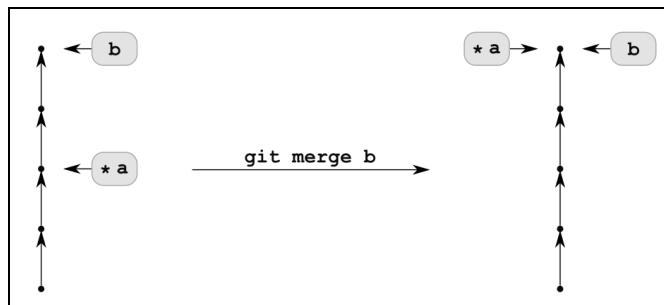
```
Updating ...
Fast-forward
...
```

informujący o tym, że łączenie zostało wykonane jako przewijanie do przodu.

Działanie polecenia `git merge b` dla repozytorium z rysunku 14.3 ilustruje rysunek 14.4.

Rysunek 14.4.

Przebieg operacji przewijania do przodu gałęzi a w repozytorium z rysunku 14.3



Jeśli operację łączenia spróbujemy wykonać w gałęzi b:

```
git checkout b
git merge a
```

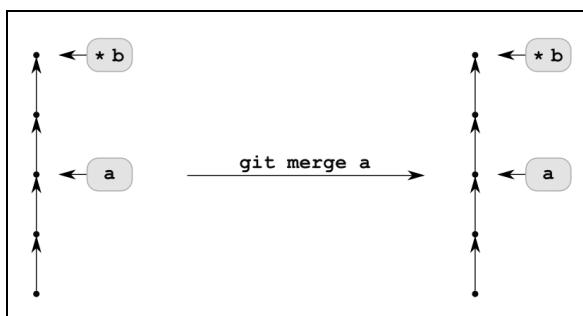
ujrzymy wówczas komunikat:

Already up-to-date.

Informuje on o tym, że gałąź b zawiera wszystkie rewizje zawarte w gałęzi a. Operacja ta nie zmieni więc repozytorium. Taki przebieg operacji łączenia ilustruje rysunek 14.5.

Rysunek 14.5.

Działanie polecenia `git merge` dla gałęzi b, która zawiera gałąź a, nie modyfikuje repozytorium



Przewijanie do przodu dla wielu gałęzi

Przewijanie do przodu możemy wykonać dla wielu gałęzi na raz. Jeśli w repozytorium z rysunku 14.6 w gałęzi a wydamy komendę:

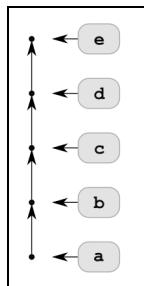
```
git merge b c d e
```

ujrzymy wówczas wydruk:

```
Fast-forwarding to: b
Fast-forwarding to: c
Fast-forwarding to: d
Fast-forwarding to: e
```

Rysunek 14.6.

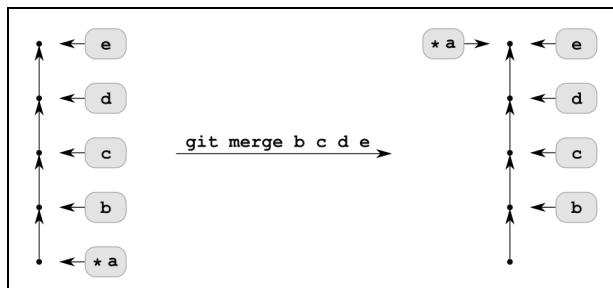
Repozytorium zawierające kilka zawierających się gałęzi



Przebieg operacji przewijania do przodu dla wielu gałęzi jest przedstawiony na rysunku 14.7.

Rysunek 14.7.

Przewijanie do przodu gałęzi a w repozytorium z rysunku 14.6

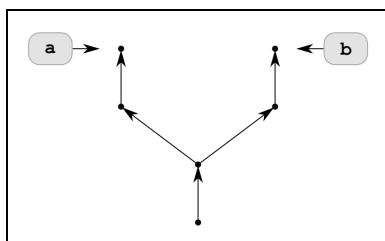


Łączanie gałęzi rozłącznych

Łączanie gałęzi rozłącznych dotyczy repozytorium o strukturze przedstawionej na rysunku 14.8.

Rysunek 14.8.

Repozytorium zawierające rozłączne gałęzie



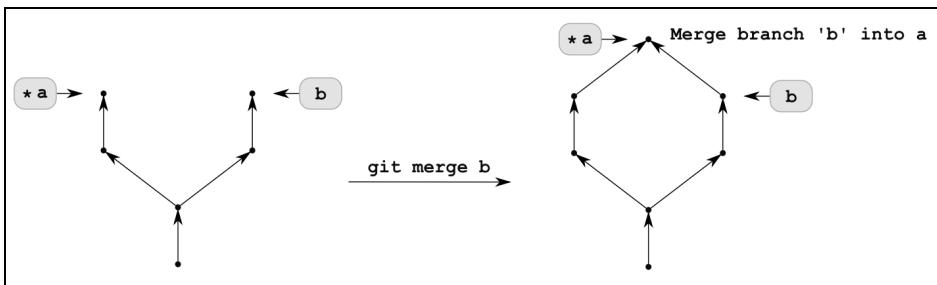
W celu dołączenia gałęzi b do gałęzi a przechodzimy na gałąź a:

```
git checkout a
```

po czym wydajemy komendę:

```
git merge b
```

Przebieg powyższej operacji ilustruje rysunek 14.9.



Rysunek 14.9. Przebieg operacji łączenia dwóch gałęzi rozłącznych

Zwróć uwagę, że po wykonaniu takiej operacji w repozytorium pojawi się nowa rewizja oznaczona komunikatem:

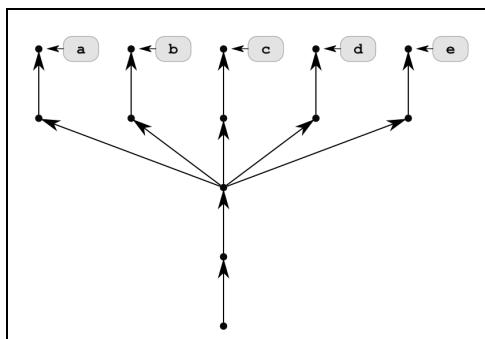
Merge branch 'b' into a

Łączenie kilku rozłącznych gałęzi

Repozytorium zawierające pięć rozłącznych gałęzi: a, b, c, d, e jest przedstawione na rysunku 14.10.

Rysunek 14.10.

Repozytorium
zawierające pięć
rozłącznych gałęzi:
a, b, c, d, e



W celu dołączenia gałęzi b, c, d oraz e do gałęzi a najpierw przechodzimy na gałąź a:

git checkout a

po czym wydajemy komendę:

git merge b c d e

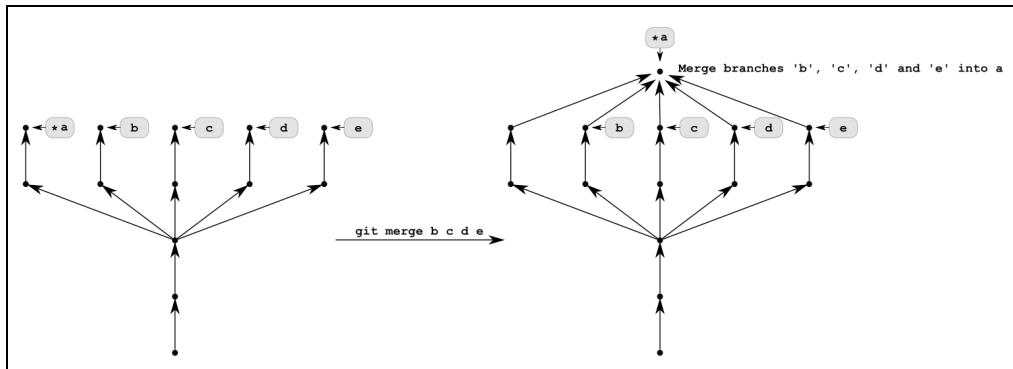
Komenda ta wygeneruje wydruk:

```
Trying simple merge with b
Trying simple merge with c
Trying simple merge with d
Trying simple merge with e
Merge made by octopus.
```

W repozytorium pojawi się nowa rewizja oznaczona komunikatem:

Merge branches 'b', 'c', 'd' and 'e' into a

Procedura łączenia wielu rozłącznych gałęzi jest przedstawiona na rysunku 14.11.



Rysunek 14.11. Przebieg operacji łączenia rozłącznych gałęzi a, b, c, d, e



Operacja łączenia większej liczby gałęzi jest niezmiernie rzadko spotykana w praktyce. Spośród projektów wymienionych w rozdziale 1. tylko jądro Linuksa oraz Git stosują łączenie większej liczby gałęzi niż dwie.

Ćwiczenie 14.1

Wykonaj repozytorium przedstawione na rysunku 14.3. Kolejne rewizje oznacz komunikatami A1, A2, A3 itd. i umieść w nich pliki *a1.txt*, *a2.txt*, *a3.txt* itd.

Ćwiczenie 14.2

W repozytorium z ćwiczenia 14.1 wykonaj operację łączenia przedstawioną na rysunku 14.4.

Ćwiczenie 14.3

Wykonaj repozytorium przedstawione na rysunku 14.6. Kolejne rewizje oznacz komunikatami A1, A2, A3 itd. i umieść w nich pliki *a1.txt*, *a2.txt*, *a3.txt* itd.

Ćwiczenie 14.4

W repozytorium z ćwiczenia 14.3 wykonaj operację łączenia przedstawioną na rysunku 14.7.

Ćwiczenie 14.5

Wykonaj repozytorium przedstawione na rysunku 14.8. Rewizje wspólne obu gałęzi oznacz jako x1, x2, x3 itd. Rewizje z gałęzi a oznacz jako a1, a2, a3 itd. Rewizje z gałęzi b oznacz jako b1, b2, b3 itd.

Ćwiczenie 14.6

W repozytorium z ćwiczenia 14.5 wykonaj operację łączenia przedstawioną na rysunku 14.9.

Ćwiczenie 14.7

Wykonaj repozytorium przedstawione na rysunku 14.10. Rewizje wspólne obu gałęzi oznacz jako x1, x2, x3 itd. Rewizje z gałęzi a oznacz jako a1, a2, a3 itd. Rewizje z gałęzi b oznacz jako b1, b2, b3 itd. W kolejnych gałęziach zastosuj identyczną konwencję.

Ćwiczenie 14.8

W repozytorium z ćwiczenia 14.7 wykonaj operację łączenia przedstawioną na rysunku 14.11.

Wycofywanie operacji git merge

W celu wycofania operacji łączenia gałęzi wykonanej poleceniem `git merge` wystarczy użyć polecień `git log` oraz `git reset`. Jeśli po wykonaniu operacji `git merge` polecenie `git log` zwraca wynik:

```
X Merge branch '...' into ...
Y ...
Z ...
```

(X, Y oraz Z są skrótami SHA-1 poszczególnych rewizji), wówczas polecenie:

```
git reset --hard HEAD~
```

spowoduje usunięcie rewizji łączenia gałęzi.

Ćwiczenie 14.9

Sprawdź wydajność operacji `git merge`, łącząc dwie gałęzie zawierające po 100 nowych rewizji. Na zakończenie wycofaj operację łączenia gałęzi.

ROZWIĄZANIE

Utwórz puste repozytorium:

```
git init
```

i wykonaj w nim pierwszą rewizję:

```
git simple-commit init
```

Następnie utwórz dwie gałęzie:

```
git checkout -b a  
git checkout -b b
```

po czym w obu gałęziach wykonaj po 100 rewizji:

```
git checkout a  
git simple-loop a 100  
  
git checkout b  
git simple-loop b 100
```

Teraz przejdź do gałęzi a i dołącz do niej gałąź b:

```
git checkout a  
git merge b
```

Operacja ta zajmie niecałą sekundę. Wydruk polecenia git log z opcją --graph będzie prezentował diagram ilustrujący łączenie gałęzi:

```
*   daa5 Merge branch 'b' into a  
|\  
| * d7a4 b100  
| * 73fb b99  
| * ef07 b98  
| * debf b97  
...  
...
```

W celu wycofania operacji łączenia gałęzi, będąc w gałęzi a, wydaj komendę:

```
git reset --hard HEAD~
```

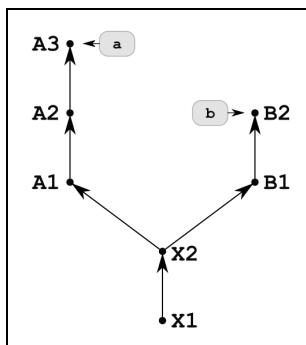
Rozdział 15.

Łączenie gałęzi: operacja rebase

Operacja łączenia gałęzi polecienniem `merge` generuje projekt zawierający historię z rozgałęzieniami. Innym wariantem łączenia jest operacja `rebase`. Po połączeniu gałęzi operacją `rebase` otrzymujemy projekt o liniowej historii.

Operację taką wykonamy dla repozytorium przedstawionego na rysunku 15.1.

Rysunek 15.1.
*Repozytorium
zawierające dwie
rozłączne gałęzie: a i b*



Jeśli po przejściu na gałąź a:

```
git checkout a
```

wydamy polecenie:

```
git rebase b
```

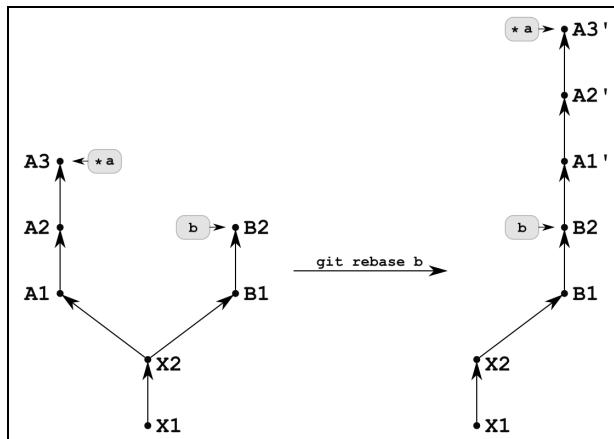
otrzymamy wówczas repozytorium przedstawione na rysunku 15.2. Polecenie `git rebase` wygeneruje wydruk:

```
First, rewinding head to replay your work on top of it...
```

```
Applying: a1
```

```
Applying: a2
```

Rysunek 15.2.
Operacja łączenia gałęzi *a* i *b* poleciem
git rebase

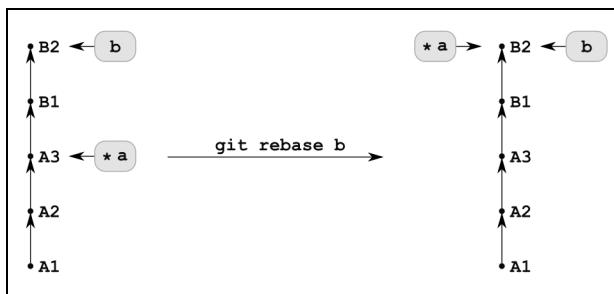


Przebieg operacji *git rebase* możemy interpretować jako zastosowanie zmian wykonywanych w rewizjach A1, A2 i A3 do repozytorium będącego w stanie z rewizji B2.

Symboli A1', A2' i A3' widoczne na rysunku 15.2 oznaczają rewizje, które zawierają te same modyfikacje (tj. takie same nowe pliki oraz takie same zmiany w istniejących plikach). Nie są to jednak rewizje identyczne jak A1, A2 i A3. Skróty SHA-1 rewizji A1 oraz A1' będą różne. Z punktu widzenia synchronizacji projektu będą to więc zupełnie nowe rewizje. Z tego powodu operacja *rebase* może być stosowana wyłącznie wtedy, gdy rewizje A1, A2 oraz A3 nie zostały udostępnione publicznie.

W przypadku gdy gałąź bieżąca jest zawarta w gałęzi dołączanej, operacja *rebase* daje ten sam efekt co operacja *merge*. Przebieg operacji *rebase* dla gałęzi zawartych ilustruje rysunek 15.3.

Rysunek 15.3.
Przebieg operacji
rebase dla gałęzi *a*
zawartej w gałęzi *b*



Podobieństwa i różnice pomiędzy poleceniami *merge* i *rebase*

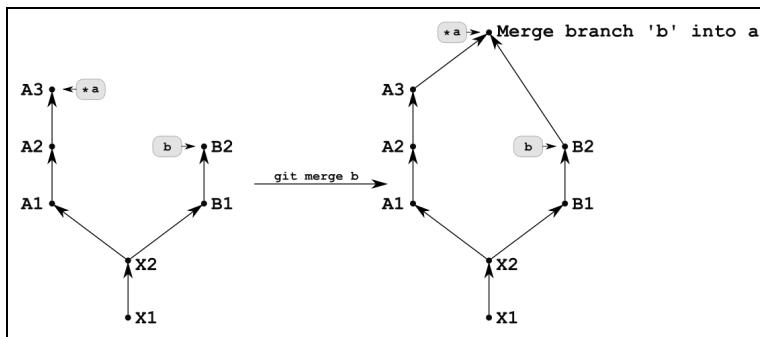
Łączenie gałęzi, bez względu na to, czy zostanie wykonane poleciem *git rebase*, czy poleciem *git merge*, da w rezultacie identyczne pliki. Otrzymane repozytorium

różnić się będzie wyłącznie historią, czyli strukturą powiązań poprzednich rewizji oraz skrótami SHA-1 rewizji.

W celu porównania operacji:

```
git rebase  
git merge
```

wykonajmy operację git merge dla repozytorium z rysunku 15.1. Przebieg operacji łączenia gałęzi a i b z rysunku 15.1 poleciem git merge jest przedstawiony na rysunku 15.4.

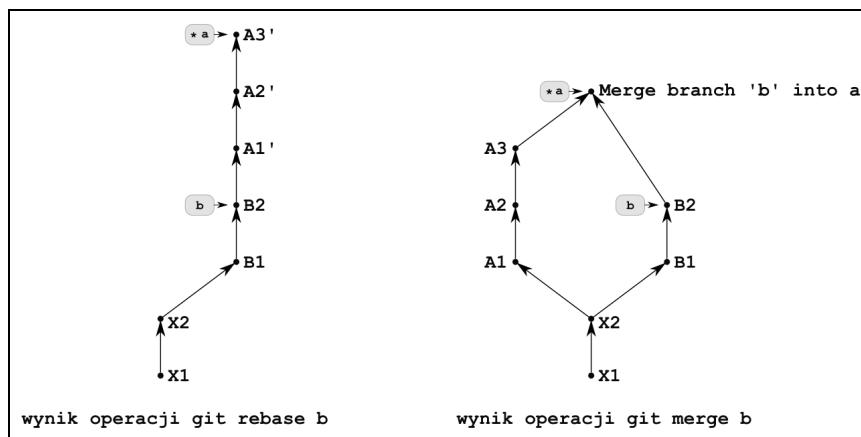


Rysunek 15.4. Łączenie gałęzi z rysunku 15.1 poleciem git merge

A zatem:

- ◆ rysunek 15.3 przedstawia przebieg operacji łączenia gałęzi a i b poleciem git rebase,
- ◆ a rysunek 15.4 przedstawia przebieg operacji łączenia gałęzi a i b poleciem git merge

dla repozytorium z rysunku 15.1. Otrzymamy repozytoria przedstawione na rysunku 15.5.



Rysunek 15.5. Porównanie wyników operacji git rebase oraz git merge

Stan plików w gałęziach oznaczonych symbolem a* w obu drzewach z rysunku 15.5 jest identyczny.

Operacja łączenia git merge dodaje do historii projektu jedną rewizję, która ma kilku przodków.

Ćwiczenie 15.1

Wykonaj repozytorium przedstawione na rysunku 15.1.

Ćwiczenie 15.2

W repozytorium z ćwiczenia 15.1 wykonaj przedstawioną na rysunku 15.2 operację łączenia git rebase.

Ćwiczenie 15.3

W repozytorium z ćwiczenia 15.1 wykonaj przedstawioną na rysunku 15.4 operację łączenia git merge.

Ćwiczenie 15.4

Porównaj stan plików w obszarze roboczym repozytoriów z ćwiczeń 15.2 oraz 15.3. W obu przypadkach po zakończeniu operacji łączenia obszar roboczy powinien zawierać pliki:

a1.txt, a2.txt, a3.txt
b1.txt, b2.txt
x1.txt, x2.txt

Wycofywanie operacji git rebase

Usuwanie operacji git rebase jest bardziej skomplikowane, gdyż operacja ta modyfikuje rewizje zawarte w projekcie. Nie możemy przywrócić stanu jednej z rewizji zwracanych jako wynik polecenia git log, gdyż rewizje z gałęzi bieżącej zostały zmodyfikowane.

W celu wycofania operacji łączenia gałęzi wykonanej poleceniem git rebase należy użyć dziennika reflog. Po wydaniu komendy:

```
git reflog
```

wyszukujemy rewizję, która wskazuje stan repozytorium przed operacją git rebase. Po odnalezieniu odpowiedniej rewizji wydajemy komendę:

```
git reset --hard HEAD@{n}
```



Uwaga

Jeśli operacja git rebase nie powiedzie się, repozytorium pozostaje w stanie detached head. Operację taką możemy anulować poleceniem:

```
git rebase --abort
```

Ćwiczenie 15.5

Sprawdź wydajność operacji git rebase, łącząc dwie gałęzie zawierające po 100 nowych rewizji. Na zakończenie wycofaj operację łączenia gałęzi.

ROZWIĄZANIE

Utwórz puste repozytorium:

```
git init
```

i wykonaj w nim pierwszą rewizję:

```
git simple-commit init
```

Następnie utwórz dwie gałęzie:

```
git checkout -b a  
git checkout -b b
```

po czym w obu gałęziach wykonaj po 100 rewizji:

```
git checkout a  
git simple-loop a 100
```

```
git checkout b  
git simple-loop b 100
```

Teraz przejdź do gałęzi b i dołącz do niej gałąz a:

```
git checkout b  
git rebase a
```

Otrzymamy repozytorium, którego historia będzie zawierała najpierw rewizje z gałęzi a, zaś później — rewizje z gałęzi b. Wykonanie operacji git rebase zajmuje około jednej sekundy dla każdej rewizji. Łącznie będzie to więc około 100 sekund — znacznie dłużej niż w przypadku operacji git merge.

Oczywiście wydruk polecenia git log z opcją --graph będzie prezentował diagram liniowy:

```
* 7del b100  
* 5263 b99  
* bb2c b98  
* 6f94 b97  
* e53c b96  
...
```

W celu wycofania operacji łączenia poleceniem:

```
git reflog
```

sprawdzamy dziennik reflog. Znajdziemy w nim po jednym wpisie dla każdej rewizji zawartej w gałęzi b:

```
7de1b4b HEAD@{0}: rebase finished: returning to refs/heads/b
7de1b4b HEAD@{1}: rebase: b100
5263df8 HEAD@{2}: rebase: b99
bb2ce72 HEAD@{3}: rebase: b98
...
e1df85d HEAD@{98}: rebase: b3
9d54682 HEAD@{99}: rebase: b2
fccc9e7 HEAD@{100}: rebase: b1
8e1398f HEAD@{101}: checkout: moving from b to 8e13...
3cb1c14 HEAD@{102}: commit: b100
...
```

Operacja git rebase rozpoczyna się od przejścia do gałęzi a (skrót 8e13 to SHA-1 ostatniej rewizji w gałęzi a):

```
8e1398f HEAD@{101}: checkout: moving from b to 8e13...
```

Stan bezpośrednio przed tym krokiem jest oznaczony jako:

```
3cb1c14 HEAD@{102}: commit: b100
```

zatem w celu wycofania operacji łączenia należy wydać komendę:

```
git reset --hard HEAD@{102}
```

Operacja wycofywanie łączenia jest natychmiastowa.

Rozdział 16.

Podsumowanie części II

Rozgałęzianie i łączenie gałęzi to dwie najmocniejsze strony Gita. Operacja rozgałęziania jest natychmiastowa: wymaga wyłącznie utworzenia nowego pliku tekstowego zawierającego wskaźnik do rewizji. Operacja łączenia jest bardziej pracochłonna. Ponieważ jednak czas trwania łączenia zależy bezpośrednio od liczby dodanych rewizji, w większości przypadków również nie stanowi to zbyt dużego obciążenia.

Pamiętaj, by wszystkie rewizje zawsze umieszczać w gałęziach. Nigdy nie pracuj w stanie detached HEAD. Rewizje umieszczone w gałęziach nigdy nie przepadają i zawsze będziesz mógł do nich wrócić.

Gałęzie są niezmiernie wygodnym rozwiązaniem pozwalającym na testowanie nowych pomysłów. Jeśli zechcesz wprowadzić w projekcie — bez żadnych konsekwencji — drastyczne zmiany, wykonaj to w następujący sposób:

- ◆ Utwórz nową gałąź.
- ◆ W nowej gałęzi wprowadź zmiany.

Zmiany wprowadzone w taki sposób nie kolidują z główną gałęzią projektu. Jeśli stwierdzisz, że zmiany są dobre, możesz je dołączyć do projektu jedną z operacji: git merge lub git rebase. Jeśli nie jesteś pewny, czy zmiany są dobre, czy złe — pozostaw nową gałąź w projekcie. Jeśli stwierdzisz, że zmiany są złe — usuń nową gałąź.

Jeśli się pomyliłeś i dołączysz do projektu zmiany, które później okażą się błędne, również nic nie ryzykujesz. Po prostu w przyszłości łączenie gałęzi wycofasz, stosując polecenie git revert.

Co powinieneś umieć po lekturze drugiej części?

Po wykonaniu ćwiczeń z drugiej części podręcznika powinieneś umieć biegle:

- ◆ listować,
- ◆ tworzyć,
- ◆ usuwać
- ◆ oraz łączyć

gałęzie zawarte w repozytorium. Powinieneś rozumieć różnicę pomiędzy łączeniem gałęzi operacjami:

```
git merge  
git rebase
```

oraz umieć wycofać operację łączenia.

Listy poznanych poleceń

Operowanie gałęziami

Sprawdzanie bieżącej gałęzi:

```
git branch  
git status -sb
```

Tworzenie gałęzi wskazującej na bieżącą rewizję:

```
git branch nazwa
```

Tworzenie gałęzi wskazującej na dowolną rewizję:

```
git branch nazwa SHA-1
```

Przełączanie gałęzi:

```
git checkout nazwa
```

Tworzenie i przełączanie gałęzi (nowa gałąź wskazuje bieżącą rewizję):

```
git checkout -b nazwa
```

Tworzenie i przełączanie gałęzi (nowa gałąź wskazuje dowolną rewizję):

```
git checkout -b nazwa SHA-1
```

Przełączanie gałęzi w repozytorium surowym:

```
git symbolic-ref HEAD refs/heads/nazwa-galezi
```

Lista gałęzi zawartych w bieżącej gałęzi:

```
git branch --merged
```

Lista gałęzi rozłącznych z bieżącą gałęzią:

```
git branch --no-merged
```

Lista gałęzi zawartych w dowolnej gałęzi:

```
git branch --merged SHA-1
```

Lista gałęzi rozłącznych z dowolną gałęzią:

```
git branch --no-merged SHA-1
```

Usuwanie gałęzi zawartych w dowolnej gałęzi występującej w repozytorium:

```
git branch -d nazwa-gałezi
```

Usuwanie gałęzi rozłącznych:

```
git branch -D nazwa-gałezi
```

Zmiana nazwy gałęzi:

```
git branch -m stara-nazwa nowa-nazwa
```

Zmiana nazwy gałęzi z wymuszeniem usunięcia istniejącej gałęzi o podanej nazwie:

```
git branch -M stara-nazwa nowa-nazwa
```

Sprawdzanie różnic pomiędzy gałeziami:

```
git diff --name-status galaz-a..galaz-b
```

Optymalizacja repozytorium

Parametry usuwania zgubionych rewizji:

```
git config gc.pruneexpire now  
git config gc.reflogexpireunreachable now
```

Usuwanie zgubionych rewizji:

```
git prune
```

Optymalizacja repozytorium

```
git gc
```

Łączenie gałęzi (merge)

Łączenie gałęzi:

```
git merge nazwa-gałezi
```

Łączenie wielu gałęzi:

```
git merge a b c
```

Usuwanie łączenia gałęzi wykonanego operacją git merge:

```
git reset --hard HEAD~
```

Łączenie gałęzi (rebase)

Łączenie gałęzi:

```
git rebase nazwa-gałęzi
```

Usuwanie łączenia gałęzi wykonanego operacją git rebase (w oparciu o dziennik reflog):

```
git reflog  
git reset --hard HEAD@{n}
```

Anulowanie nieudanej operacji git rebase:

```
git rebase --abort
```

Część III

Gałęzie zdalne

Rozdział 17.

Definiowanie powiązania między repozytorium lokalnym a zdalnym

Przejdźmy do definiowania powiązań pomiędzy repozytoriami. Repozytorium, w którym poleceniami `git add` oraz `git commit` będziemy wykonywali rewizje, nazwiemy **repozytorium lokalnym**. Repozytoria, które posłużą do synchronizacji rewizji, nazwiemy **repozytoriami zdalnymi**.

Repozytoria lokalne będą repozytoriami zwykłymi, a repozytoria zdalne — repozytoriami surowymi.

Klonowanie raz jeszcze

Poznana w rozdziale 3. operacja klonowania:

```
git clone adres [folder]
```

dotyczy dwóch repozytoriów. Na lokalnym dysku tworzymy repozytorium, które będzie kopią repozytorium zdalnego. Klonowanie jest więc najprostszym przykładem tworzenia powiązania pomiędzy dwoma repozytoriami.

Po wykonaniu operacji klonowania repozytorium utworzone na dysku nazwiemy **repozytorium lokalnym**, a repozytorium, którego adres pojawił się w poleceniu `git clone` — **repozytorium zdalnym**.



Wskazówka

Po wykonaniu operacji:

```
git clone git://github.com/symfony/symfony.git .
```

repozytorium:

```
git://github.com/symfony/symfony.git
```

jest repozytorium zdalnym. Repozytorium utworzone na dysku to **repozytorium lokalne**.

Podczas klonowania wykonywane są następujące czynności:

1. Proces rozpoczyna się od inicjalizacji nowego pustego repozytorium lokalnego.
2. W repozytorium lokalnym dodawany jest adres repozytorium zdalnego. Adres ten jest automatycznie oznaczany nazwą origin.
3. Z repozytorium zdalnego kopowane są rewizje ze zdalnej gałęzi master do lokalnej gałęzi master.
4. W repozytorium lokalnym w folderze `.git/refs/remotes/origin` tworzony jest plik `HEAD` zawierający nazwę symboliczną domyślnej gałęzi repozytorium zdalnego.
5. Następnie definiowane jest powiązanie lokalnej gałęzi master ze zdalną gałęzią master. Gałąź lokalna będzie **śledziła** (ang. *track*) gałąź zdalną.
6. Na zakończenie stan plików w obszarze roboczym repozytorium lokalnego jest przywracany do postaci z gałęzi master.

Wszystkie powyższe operacje możemy wykonać ręcznie.

Procedura ręcznego klonowania

1. Polecenie:

```
git init
```

tworzy nowe puste repozytorium git.

2. Polecenie:

```
git remote add origin adres
```

dodaje w konfiguracji adres repozytorium zdalnego.

3. Polecenie:

```
git fetch --no-tags origin master:refs/remotes/origin/master
```

kopiuje z repozytorium zdalnego do repozytorium lokalnego wszystkie rewizje zawarte w gałęzi master. Ponadto w repozytorium lokalnym w folderze `.git/refs/remotes/origin` tworzona jest nazwa symboliczna dla zdalnej gałęzi master. Parametr `--no-tags` powoduje, że znaczniki z repozytorium zdalnego nie będą kopiowane.

4. Polecenie:

```
git branch --set-upstream master origin/master
```

tworzy powiązanie pomiędzy lokalną gałęzią master a zdalną gałęzią master.

5. Na zakończenie polecenie:

```
git reset --hard HEAD
```

przywraca stan plików w obszarze roboczym.



Repozytorium wykonane opisana powyżej procedurą różni się od repozytorium klonowanego tylko tym, że w repozytorium klonowanym w pliku *refs/remotes/origin/HEAD* adres gałęzi zdalnej master jest zapisany w postaci symbolicznej:
ref: refs/remotes/origin/master

Ćwiczenie 17.1

Sklonuj repozytorium jQuery:

```
git://github.com/jquery/jquery.git
```

po czym sprawdź konfigurację repozytorium lokalnego.

Po wykonaniu polecenia:

```
git clone git://github.com/jquery/jquery.git .
```

w repozytorium lokalnym w pliku *.git/config* znajdziemy wpisy ustalające adres origin oraz powiązanie lokalnej gałęzi master ze zdalną gałęzią master:

```
[remote "origin"]
  fetch = +refs/heads/*:refs/remotes/origin/*
  url = git://github.com/jquery/jquery.git
[branch "master"]
  remote = origin
  merge = refs/heads/master
```

W pliku *.git/HEAD* znajdziemy odwołanie symboliczne:

```
ref: refs/heads/master
```

W pliku *.git/refs/heads/master* znajdziemy skrót SHA-1 rewizji, a w pliku *.git/refs/remotes/origin/HEAD* — odwołanie symboliczne:

```
ref: refs/remotes/origin/master
```

Ćwiczenie 17.2

Wykorzystując polecenia:

```
git init
git remote add origin git://github.com/jquery/jquery.git
git fetch --no-tags origin master:refs/remotes/origin/master
```

```
git branch --set-upstream master origin/master  
git reset --hard HEAD
```

sklonuj repozytorium jQuery:

```
git://github.com/jquery/jquery.git
```

Przed i po wykonaniu każdego kroku sprawdź zawartość następujących plików i folderów konfiguracyjnych:

```
.git/config  
.git/HEAD  
.git/refs/heads/master  
.git/refs/remotes/origin
```

ROZWIĄZANIE

Krok 1.

Utwórz nowy folder i wydaj w nim polecenie:

```
git init
```

W tym momencie plik *.git/config* nie zawiera żadnych informacji o repozytoriach zdalnych. W pliku *.git/HEAD* obecne jest odwołanie symboliczne:

```
ref: refs/heads/master
```

W folderze *.git/refs* nie występują plik *.git/refs/heads/master* i folder *.git/refs/remotes*.

Krok 2.

Po wydaniu polecenia:

```
git remote add origin git://github.com/jquery/jquery.git
```

w pliku *.git/config* znajdziemy wpis:

```
[remote "origin"]  
url = git://github.com/jquery/jquery.git  
fetch = +refs/heads/*:refs/remotes/origin/*
```

który ustala adres repozytorium określonego nazwą symboliczną *origin*.

Zawartość folderu *.git/refs* i pliku *.git/HEAD* nie uległa zmianie.

Krok 3.

Wydaj polecenie:

```
git fetch --no-tags origin master:refs/remotes/origin/master
```

Spowoduje ono pobranie z repozytorium zdalnego *origin* wszystkich rewizji zawartych w gałęzi *master*.

Ponadto w folderze *.git/refs/remotes/origin/master* utworzony zostanie plik zawierający skrót SHA-1 ostatniej rewizji w gałęzi *master* repozytorium zdalnego *origin*.

Krok 4.

Wydaj polecenie:

```
git branch --set-upstream master origin/master
```

W ten sposób zdefiniowane zostanie powiązanie pomiędzy lokalną gałęzią master a zdalną gałęzią master. Powiązanie to jest zapisywane w pliku `.git/config` w postaci wpisu:

```
[branch "master"]
remote = origin
merge = refs/heads/master
```

Krok 5.

Ostatnie z poleceń:

```
git reset --hard HEAD
```

przywraca stan plików obszaru roboczego do postaci z ostatniej rewizji zawartej w lokalnej gałęzi master.

Klonowanie repozytorium z dysku

Operację klonowania repozytorium możemy wykonać lokalnie, bez żadnej komunikacji sieciowej. Adresem repozytorium zdalnego może być ścieżka prowadząca do repozytorium. Polecenie:

```
git clone C:\my\repos\example .
```

klonuje repozytorium z folderu `C:\my\repos\example` do folderu bieżącego.

Ścieżkę prowadzącą do repozytorium zdalnego możemy także przekazać jako parametr polecenia `git remote`, np.:

```
git remote add origin C:\my\repos\example
```

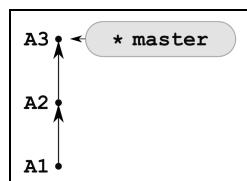
Dzięki takiemu rozwiążaniu ćwiczenia dotyczące synchronizacji repozytoriów będziemy mogli wykonywać w pełni lokalnie.

Ćwiczenie 17.3

W folderze `cw-17-03/` wykonaj repozytorium przedstawione na rysunku 17.1.

Rysunek 17.1.

Repozytorium
z ćwiczenia 17.3



Ćwiczenie 17.4

Repozytorium z ćwiczenia 17.3 sklonuj do folderu *cw-17-04/*.

ROZWIĄZANIE

Przyjmijmy, że foldery:

```
cw-17-03/  
cw-17-04/
```

znajdują się w tym samym folderze.

W wierszu poleceń przejdź do folderu *cw-17-04/* i wydaj komendę:

```
git clone ../cw-17-03 .
```

Alternatywnie klonowanie możesz wykonać, wydając polecenie:

```
git clone cw-17-03 cw-17-04
```

w folderze zawierającym foldery *cw-17-03/* oraz *cw-17-04/*.

Po tej operacji w folderze *cw-17-04/* znajdziemy kopię repozytorium z ćwiczenia 17.3. Ponadto w pliku *.git/config* znajdziemy wpisy:

```
[remote "origin"]  
fetch = +refs/heads/*:refs/remotes/origin/*  
url = C:/git/cw-17-04/..../cw-17-03  
[branch "master"]  
remote = origin  
merge = refs/heads/master
```

Definiowanie repozytoriów zdalnych

Do ustalenia adresu repozytorium zdalnego służy komenda:

```
git remote add nazwa adres
```

Parametrem *nazwa* określamy sposób odwoywania się do definiowanego repozytorium zdalnego, a parametr *adres* określa jego adres. Polecenie `git remote add` zapisuje informacje o repozytorium zdalnym w pliku *.git/config*. Przykładowy wpis przyjmuje postać przedstawioną na listingu 17.1.

Listing 17.1. Fragment pliku *.git/config* zawierający informacje o repozytorium zdalnym *nazwa*

```
[remote "nazwa"]  
url = adres  
fetch = +refs/heads/*:refs/remotes/nazwa/*
```

Jeśli repozytorium znajduje się na dysku w folderze `C:\repos\zdalne` i zechcemy mu nadać nazwę `zdalne`, polecenie `git remote` przyjmie wówczas postać:

```
git remote add zdalne C:\repos\zdalne
```

W kolejnych rozdziałach repozytoria zdalne będą pochodziły z serwerów `github.com` oraz `bitbucket.org`. Polecenie `git remote` przyjmie wówczas postać:

```
git remote add my git@github.com:gajdaw/symfony.git  
git remote add gajdaw git@bitbucket.org:gajdaw/symfony.git
```

Do wyświetlenia listy repozytoriów zdalnych służy polecenie:

```
git remote -v
```

Adres repozytorium zdalnego możemy usunąć poleceniem:

```
git remote rm nazwa
```

Definiowanie powiązania między gałęzią lokalną a gałęzią śledzoną

Dla każdej gałęzi zawartej w repozytorium lokalnym możemy ustalić odpowiadającą jej **gałąź śledzoną** (ang. *tracking branch*). Dzięki temu polecenia synchronizacji, np.:

```
git pull  
git push  
git fetch
```

mogą być wywoływane bez parametrów. W takiej sytuacji synchronizacja będzie dotyczyć bieżącej gałęzi oraz odpowiadającej jej gałęzi śledzonej.

Nazwy gałęzi śledzonych poznamy, wydając polecenie:

```
git config --list
```

Wydruk będzie zawierał informacje postaci:

```
branch.master.remote=origin  
branch.master.merge=refs/heads/master
```

Ogólnie rzecz biorąc, dla gałęzi lokalnej o nazwie `X` wpisy ustalające gałąz śledzoną będą następujące:

```
branch.X.remote=...  
branch.X.merge=...
```

Wskaźówka Nazwę gałęzi śledzonej odpowiadającej gałęzi `X` poznamy także, wydając komendy:

```
git config --get branch.X.remote  
git config --get branch.X.merge
```

Do ręcznego ustalenia gałęzi śledzonej możemy użyć polecenia:

```
git branch --set-upstream galaz-lokalna repozytorium-zdalne/galaz-zdalna
```

Polecenie:

```
git branch --set-upstream master origin/master
```

ustala, że gałęzią śledzoną dla gałęzi master będzie gałąź master w repozytorium origin.

Podobnie polecenie:

```
git branch --set-upstream lorem ipsum/dolor
```

ustala, że gałęzią śledzoną dla gałęzi lorem będzie gałąź dolor w repozytorium o nazwie ipsum.

Wydruk generowany poleceniem:

```
git config --list
```

przyjmie postać:

```
branch.lorem.remote=ipsum  
branch.lorem.merge=refs/heads/dolor
```



Wskazówka

Polecenie:

```
git branch --set-upstream master origin/master
```

jest równoważne dwóm poleceniom:

```
git config branch.master.remote origin  
git config branch.master.merge refs/heads/master
```

Listowanie gałęzi

Do sprawdzania listy gałęzi lokalnych służy poznane w części drugiej polecenie:

```
git branch
```

Gałęzie zdalne poznamy, wydając polecenie:

```
git branch -r
```

Komenda:

```
git branch -a
```

wyświetla listę wszystkich gałęzi.

Rozdział 18.

Podstawy synchronizacji repozytoriów

Repozytoria Git zawierają rewizje pogrupowane w gałęzie. Każda rewizja ma unikatowy skrót SHA-1 i na tej podstawie jest jednoznacznie identyfikowana. Synchronizacja repozytoriów polega na uzgodnieniu rewizji zawartych w poszczególnych gałęziach. Co ciekawe, proces uzgadniania zawartości gałęzi jest wykonywany w identyczny sposób jak łączenie gałęzi lokalnych. Synchronizacja to po prostu operacja łączenia gałęzi. To, że gałęzie są zapisane w innych repozytoriach, nie ma wpływu na przebieg operacji.

Pobieranie gałęzi z repozytorium zdalnego do repozytorium lokalnego

W celu pobrania rewizji z gałęzi zdalnej należy wydać polecenie:

```
git fetch nazwa-repozytorium-zdalnego nazwa-galezi-zdalnej
```

Na przykład polecenie:

```
git fetch origin master
```

pobiera z repozytorium zdalnego o nazwie origin zawartość gałęzi master. Jeśli pominiemy parametry polecenia:

```
git fetch
```

to pobrana zostanie zawartość gałęzi śledzonej ustalonej poleceniem:

```
git branch --set-upstream galaz-lokalna repozytorium-zdalne/galaz-zdalna
```

Rewizje pobrane poleceniem git fetch nie są automatycznie łączone z bieżącą gałęzią.

W celu dołączenia pobranych rewizji do bieżącej gałęzi wydajemy polecenie:

```
git merge repozytorium-zdalne/galaz-zdalna
```

np.

```
git merge origin/master
```

Do wykonania trzech operacji:

- ◆ pobrania rewizji z gałęzi zdalnej,
- ◆ dołączenia pobranych rewizji do bieżącej gałęzi
- ◆ oraz ustalenia stanu plików w obszarze roboczym

służy polecenie:

```
git pull nazwa-repozytorium-zdalnego nazwa-gałęzi-zdalnej
```

Pominięcie parametrów:

```
git pull
```

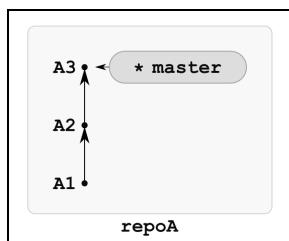
powoduje wykonanie operacji na gałęzi bieżącej i odpowiadającej jej gałęzi zdalnej.

Ćwiczenie 18.1

W folderze *cw-18-01/repoA/* wykonaj repozytorium przedstawione na rysunku 18.1.

Rysunek 18.1.

*Repozytorium repoA
z ćwiczenia 18.1*



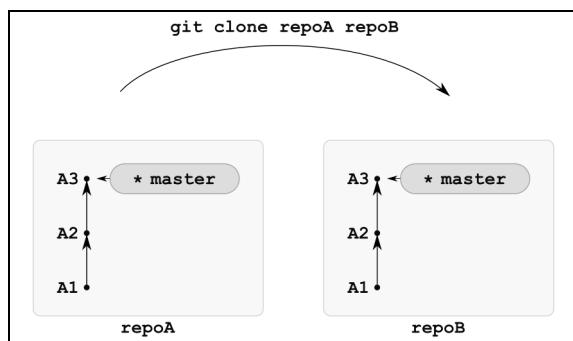
Utworzono repozytorium sklonuj do folderu *cw-18-01/repoB/*. W tym celu w folderze *cw-18-01/* wydaj komendę:

```
git clone repoA repoB
```

Otrzymasz dwa repozytoria przedstawione na rysunku 18.2.

Rysunek 18.2.

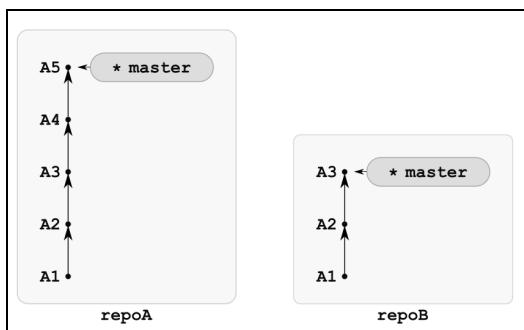
*Klonowanie
repozytorium repoA
do folderu repoB*



Następnie w repozytorium *repoA*/ dodaj rewizje A4 oraz A5. W tym celu przejdź do folderu *repoA/*, utwórz plik *a4.txt*, po czym wykonaj rewizję A4. W identyczny sposób wykonujemy rewizję A5. Repozytoria wyglądają teraz tak jak na rysunku 18.3.

Rysunek 18.3.

Stan ćwiczenia po dodaniu rewizji A4 i A5 w repozytorium *repoA*



Teraz w repozytorium *repoB*/ wykonujemy operację pobierania rewizji. Najpierw przechodzimy do repozytorium *repoB/*:

```
cd repoB
```

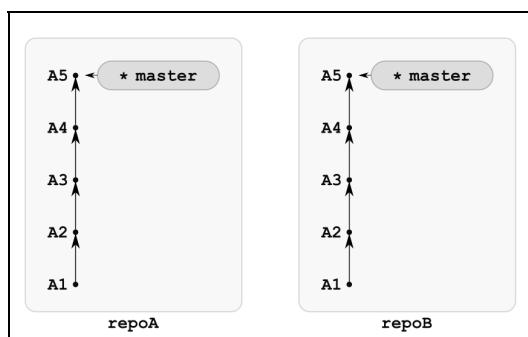
po czym wydajemy komendę:

```
git pull
```

Otrzymamy repozytoria przedstawione na rysunku 18.4.

Rysunek 18.4.

Repozytoria z rysunku 18.3 po wydaniu w repozytorium *repoB* komendy *git pull*



Operacja przedstawiona na rysunkach 18.3 oraz 18.4 odpowiada operacji przewijania do przodu, która jest zilustrowana na rysunku 14.4.

Uaktualnianie sklonowanych repozytoriów

Operacja z ćwiczenia 18.1 w praktyce występuje bardzo często. Jeśli sklonowane repozytorium, np. jQuery, zechcemy po kilku dniach uaktualnić, wówczas w gałęzi master należy wykonać operację:

```
git pull
```

Spowoduje ona pobranie z serwera brakujących rewizji.

Ćwiczenie 18.2

Repozytorium z ćwiczenia 3.2 zostało wykonane kilka dni temu. Przejdz do folderu repozytorium i wydaj komendę:

```
git log --pretty=oneline -10
```

Ujrzysz informacje o dziesięciu ostatnich rewizjach.

W celu uaktualnienia repozytorium wydaj polecenie:

```
git pull
```

Na zakończenie sprawdź w historii projektu informacje o pobranych rewizjach.

Ćwiczenie 18.3

Jeśli nie dysponujesz folderem z ćwiczenia 3.2, uaktualnianie repozytorium możesz pozwolić, wykorzystując operację usuwania ostatnich rewizji. Sklonuj repozytorium jQuery:

```
git clone git://github.com/jquery/jquery.git
```

a następnie usuń ostatnie 15 rewizji:

```
git reset --hard HEAD~15
```

Poleceniem `git log` sprawdź rewizje zawarte w repozytorium. Na zakończenie poleceniem:

```
git pull
```

uaktualnij stan repozytorium.

Repozytoria surowe

Repozytoria lokalne, w których pracujemy, dodając rewizje, składają się z trzech obszarów:

- ◆ bazy danych nazywanej repozytorium,
- ◆ indeksu,
- ◆ obszaru roboczego.

Szczegółowe omówienie tych zagadnień jest zawarte w rozdziale 6.

Zwróć uwagę, że indeks oraz obszar roboczy są potrzebne wyłącznie do tworzenia nowych rewizji na podstawie obszaru roboczego.

W repozytoriach zdalnych, które odgrywają wyłącznie rolę bazy danych rewizji, obszar roboczy oraz indeks są zbędne. Nigdy nie będziemy w nich wykonywali operacji git commit. Jedyną metodą dodawania rewizji do takiego repozytorium będzie wykonywanie operacji git push i git pull. Repozytorium pozbawione obszaru roboczego i indeksu będziemy nazywali **repozytorium surowym**.

Do utworzenia repozytorium surowego stosujemy parametr --bare:

```
git clone --bare adres folder  
git init --bare
```

Po wydaniu polecenia:

```
git init --bare
```

w folderze bieżącym powstaną pliki, które w przypadku zwykłych repozytoriów były umieszczane w folderze .git/.

Przesyłanie gałęzi do repozytorium zdalnego

Do przesyłania gałęzi z repozytorium lokalnego do repozytorium zdalnego służy polecenie:

```
git push nazwa-repozytorium-zdalnego nazwa-galezi-zdalnej
```

lub ogólnie:

```
git push nazwa-repozytorium-zdalnego nazwa-galezi-lokalnej:nazwa-galezi-zdalnej
```

Pominięcie parametrów:

```
git push
```

powoduje przesłanie bieżącej gałęzi do gałęzi śledzonej. Przy domyślnych ustawieniach konfiguracyjnych polecenie git push może być wykonane wyłącznie wtedy, gdy repozytorium zdalne jest surowe.

Jeśli wydając polecenie git push, dodasz parametr -u:

```
git push -u nazwa-repozytorium-zdalnego nazwa-galezi-zdalnej
```

wówczas gałąź o podanej nazwie zostanie zapamiętana jako gałąź śledzona gałęzi bieżącej.

Domyślnie polecenie git push umożliwia aktualnianie repozytorium zdalnego wyłącznie wtedy, gdy możliwe jest wykonanie operacji przewijania do przodu. W takim przypadku nigdy nie zachodzi ryzyko utraty danych. Jeśli operacja przewijania do

przodu nie może być wykonana, polecenie git push zakończy się błędem. Jeśli mimo ostrzeżeń chcemy uaktualnić gałąź śledzoną tak, by była identyczna z gałęzią bieżącą, należy użyć opcji -f:

```
git push -f
```

lub:

```
git push -f nazwa-repozytorium-zdalnego nazwa-galezi-zdalnej
```

np.:

```
git push -f origin master
```

Działanie powyższej komendy możemy interpretować jako usunięcie gałęzi śledzonej w zdalnym repozytorium i utworzenie nowej gałęzi zdalnej, która będzie identyczna jak gałąź bieżąca. Po wykonaniu tej operacji utracimy wszystkie rewizje, które nie są zawarte w gałęzi bieżącej.

Parametr --all powoduje przesłanie do repozytorium zdalnego wszystkich gałęzi z repozytorium lokalnego:

```
git push --all
```

Ćwiczenie 18.4

W folderze *cw-18-04/* utwórz trzy repozytoria o nazwach *repoA*, *magazyn* oraz *repoB*. Repozytorium o nazwie *magazyn* będzie służyło do synchronizacji zawartości repozytoriów *repoA* oraz *repoB*, a zatem powinno to być repozytorium surowe. W celu utworzenia repozytoriów w folderze *cw-18-04/* wydaj komendy:

```
git init repoA
git init --bare magazyn
git init repoB
```

Stan ćwiczenia jest przedstawiony na rysunku 18.5.

Rysunek 18.5.
Ćwiczenie 18.4 po utworzeniu trzech pustych repozytoriów



Następnie w repozytorium *repoA* dodaj trzy rewizje: A1, A2 i A3. Otrzymamy repozytorium przedstawione na rysunku 18.6.

Rewizje A1, A2 i A3 chcemy udostępnić użytkownikowi repozytorium *repoB*. W tym celu w repozytorium *repoA* wydajemy polecenia:

```
git remote add magazyn ../magazyn
git push -u magazyn master
```

Rysunek 18.6.

*Ćwiczenie 18.4
po dodaniu rewizji
w repozytorium repoA*

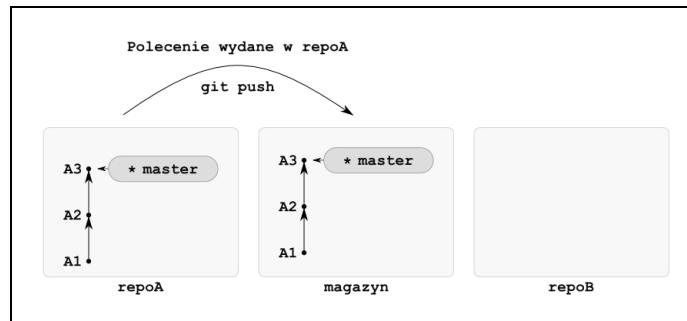


Po tej operacji repozytorium *magazyn* będzie zawierało rewizje A1, A2 i A3, zaś w konfiguracji repozytorium *repoA* dodana zostanie gałąź śledzona.

Stan repozytoriów ćwiczenia jest przedstawiony na rysunku 18.7.

Rysunek 18.7.

*Ćwiczenie 18.4
po wydaniu
w repozytorium
repoA polecenia
git push*



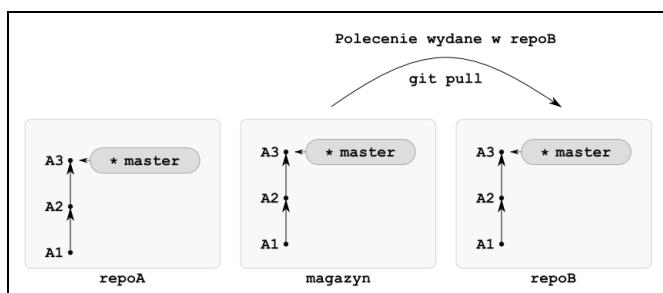
Teraz rewizje z magazynu pobierzymy do *repoB*. W tym celu w repozytorium *repoB* wydajemy polecenia:

```
git remote add magazyn ../magazyn
git pull magazyn master
```

Otrzymamy repozytoria przedstawione na rysunku 18.8.

Rysunek 18.8.

*Ćwiczenie 18.4
po wydaniu
w repozytorium
repoB polecenia
git pull*



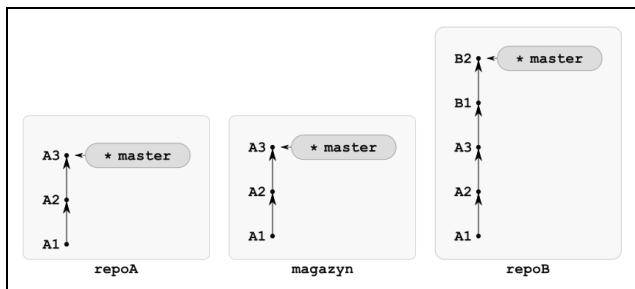
Kolejnym krokiem jest dodanie w repozytorium *repoB* rewizji B1 i B2. Repozytoria przyjmą postać widoczną na rysunku 18.9.

Rewizje B1 i B2 chcemy przesyłać do repozytorium *repoA*. W tym celu w repozytorium *repoB* wydajemy polecenie:

```
git push -u magazyn master
```

Rysunek 18.9.

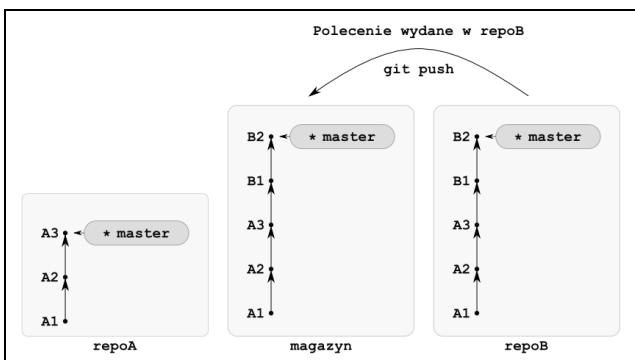
*Ćwiczenie 18.4
po dodaniu
w repozytorium
repoB rewizji B1 i B2*



Po wydaniu powyższej komendy repozytorium *magazyn* będzie zawierało rewizje B1 i B2, a w repozytorium *repoB* zostanie ustalona gałąź śledzona gałęzi *master*. Stan repozytoriów ćwiczenia jest przedstawiony na rysunku 18.10.

Rysunek 18.10.

*Ćwiczenie 18.4
po wydaniu
w repozytorium
repoB polecenia
git push*



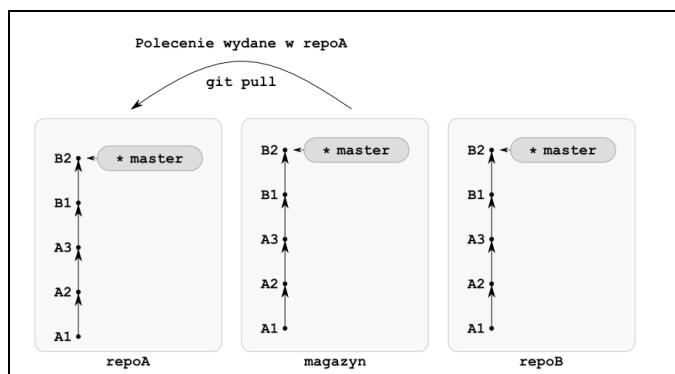
W celu pobrania do repozytorium *repoA* rewizji B1 i B2 należy w repozytorium *repoA* wydać komendę:

```
git pull
```

Tym razem nie musimy podawać żadnych parametrów polecenia `git pull`, gdyż parametr `-u` wydanego wcześniej (w repozytorium *repoA*) polecenia `git push` ustalił, że dla gałęzi *master* gałęzią śledzoną jest gałąź *magazyn/master*. Stan repozytoriów ćwiczenia jest przedstawiony na rysunku 18.11.

Rysunek 18.11.

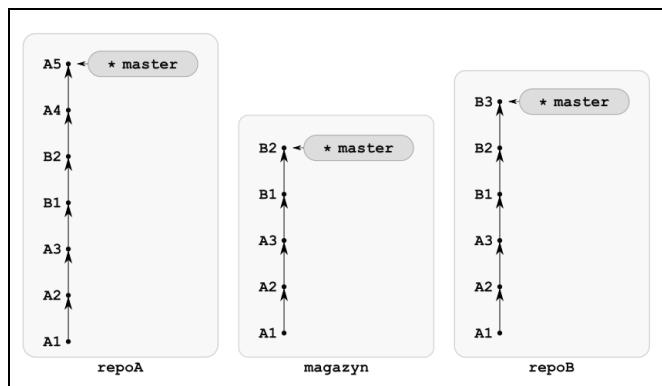
*Ćwiczenie 18.4
po wydaniu
w repozytorium
repoA polecenia
git pull*



Co się stanie, jeśli użytkownicy repozytoriów *repoA* oraz *repoB* spróbowają dodać zmiany w tym samym czasie? Sprawdźmy. Dodajmy w repozytorium *repoA* rewizje A4 i A5, a w repozytorium *repoB* rewizję B3. Repozytoria ćwiczenia powinny mieć postać przedstawioną na rysunku 18.12.

Rysunek 18.12.

Ćwiczenie 18.4
po dodaniu rewizji
A4, A5 i B3



Jeśli teraz w repozytorium *repoA* wydasz polecenie:

```
git status -sb
```

ujrzysz wówczas wydruk informujący o tym, że gałąź master repozytorium *repoA* zawiera o dwie rewizje więcej niż śledzona gałąź magazyn/master:

```
## master...magazyn/master [ahead 2]
```

W repozytorium *repoB* wynikiem polecenia:

```
git status -sb
```

będzie wydruk:

```
## master...magazyn/master [ahead 1]
```

Użytkownicy repozytoriów *repoA* oraz *repoB* chcą udostępnić swoje nowe rewizje. Oczywiście wydadzą w tym celu polecenia:

```
git push
```

W praktyce jedna z tych operacji zostanie wykonana jako pierwsza. Przyjmijmy, że jest to operacja wydana w repozytorium *repoB*. Operacja ta przebiegnie identycznie jak na rysunkach 18.7 oraz 18.10. Stan repozytoriów po wydaniu w repozytorium *repoB* polecenia git push jest przedstawiony na rysunku 18.13.

Teraz repozytorium *repoA* zawiera nowe rewizje A4 oraz A5, lecz nie zawiera wszystkich rewizji z repozytorium *magazyn*. W takiej sytuacji komenda:

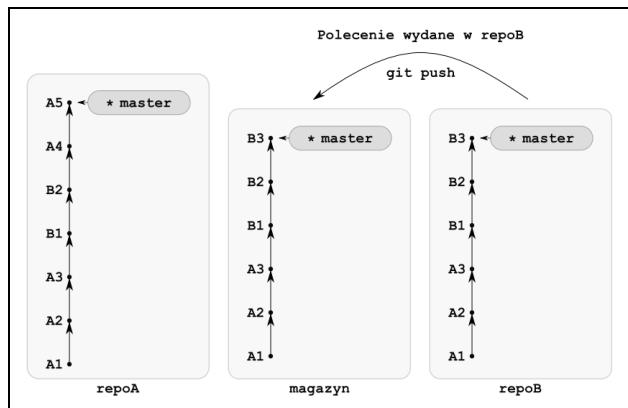
```
git push
```

wydana w repozytorium *repoA* zakończy się błędem:

```
To .../magazyn
! [rejected]           master -> master (non-fast-forward)
```

Rysunek 18.13.

Ćwiczenie 18.4 po przesłaniu do repozytorium magazyn rewizji B3



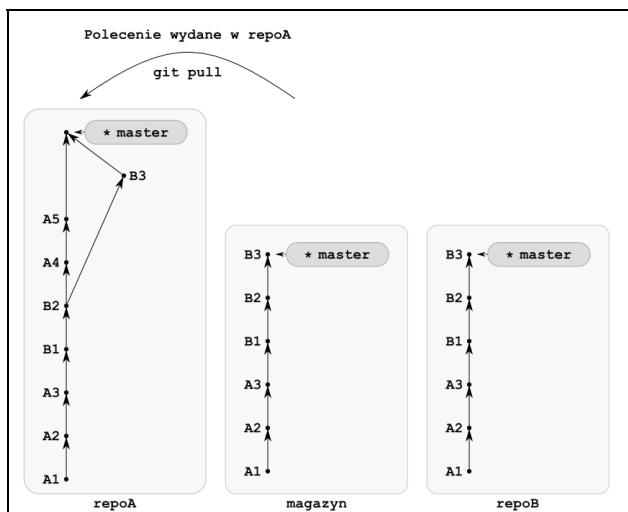
Problem ten rozwiążemy, pobierając najpierw z repozytorium *magazyn* do repozytorium *repoA* wszystkie rewizje. W repozytorium *repoA* wydajemy zatem najpierw komendę:

```
git pull
```

Spowoduje ona pobranie do *repoA* rewizji B3. Ponieważ jednak operacja ta nie była operacją przewijania do przodu, po złączeniu gałęzi repozytorium *repoA* nie będzie miało liniowej historii. Stan repozytoriów po wydaniu komendy *git pull* w repozytorium *repoA* jest przedstawiony na rysunku 18.14.

Rysunek 18.14.

Ćwiczenie 18.4 po pobraniu do repoA rewizji B3



W tym momencie rewizje A4 oraz A5 możemy przesłać do repozytorium *magazyn*. W tym celu należy w *repoA* wydać komendę:

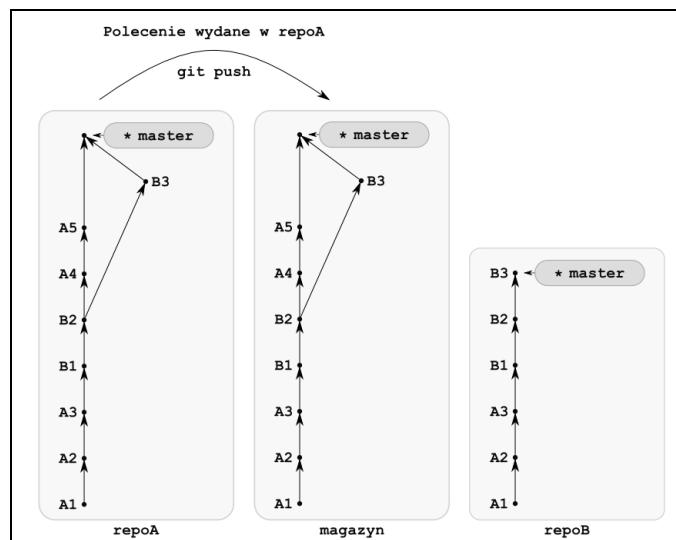
```
git push
```

Otrzymamy stan ćwiczenia przedstawiony na rysunku 18.15. Oczywiście stan repozytorium *magazyn* możemy pobrać do repozytorium *repoB* poleceniem:

```
git pull
```

Rysunek 18.15.

*Ćwiczenie 18.4
po przesłaniu rewizji
A4 i A5 do magazynu*



Podobnie jak w przypadku gałęzi lokalnych, tak i teraz łączenie możemy wykonać operacją `git rebase`. Dzięki temu otrzymamy liniową historię projektu.

Przywróćmy stan ćwiczenia do postaci widocznej na rysunku 18.14. W tym celu w repozytorium *repoA* tworzymy gałąź tymczasową o nazwie *kopia*:

```
git branch kopia
```

Gałąź ta będzie zawierała wszystkie rewizje z rysunku 18.15. Gałąź *master* w repozytorium *repoA* przywracamy do stanu repozytorium *magazyn* z rysunku 18.14. Zadanie to realizujemy komendą¹:

```
git reset --hard HEAD^2
```

Otrzymaną gałąź przesyłamy do repozytorium *magazyn*:

```
git push -f
```

Parametr `-f` powoduje, że stan gałęzi *master* w repozytorium *magazyn* zostanie uaktualniony do postaci gałęzi *master* w repozytorium *repoA* bez względu na to, że spowoduje to utratę rewizji.

Następnie w repozytorium *repoA* przechodzimy na gałąź *kopia*:

```
git checkout kopia
```

po czym zmieniamy nazwę gałęzi *kopia* na *master*:

```
git branch -M kopia master
```

Po tych operacjach ćwiczenie znajduje się w stanie przedstawionym na rysunku 18.14.

¹ Oczywiście w systemie Windows polecenie to zapisujemy z czterema znakami ^.



Zwróć uwagę, że przywracanie stanu repozytoriów wykonaliśmy, pracując wyłącznie w repozytorium *repoA*. Innymi słowy udział innych uczestników projektu był zbędny.

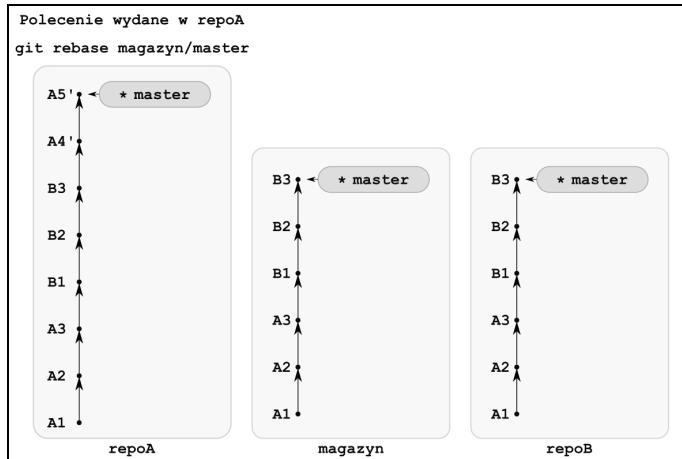
W celu wyprostowania historii repozytorium *repoA* z rysunku 18.14 należy w repozytorium *repoA* wydać polecenie:

```
git rebase magazyn/master
```

Repozytorium *repoA* przyjmie postać widoczną na rysunku 18.16.

Rysunek 18.16.

Stan ćwiczenia z rysunku 18.14 po wydaniu w repozytorium *repoA* komendy *git rebase magazyn/master*



Rewizje A4 oraz A5 zostały zmodyfikowane. Zmieniły się ich skróty SHA-1, dlatego są one na rysunku 18.16 oznaczone symbolami A4' i A5'. Nie stanowi to żadnego problemu, gdyż rewizje te nie były jeszcze udostępniane.

Oczywiście teraz wystarczy w repozytorium *repoA* wydać komendę:

```
git push
```

a w repozytorium *repoB* wydać komendę:

```
git pull
```

W wyniku wydania tych komend repozytoria *repoA*, *magazyn* oraz *repoB* przyjmą postać widoczną na rysunku 18.17.

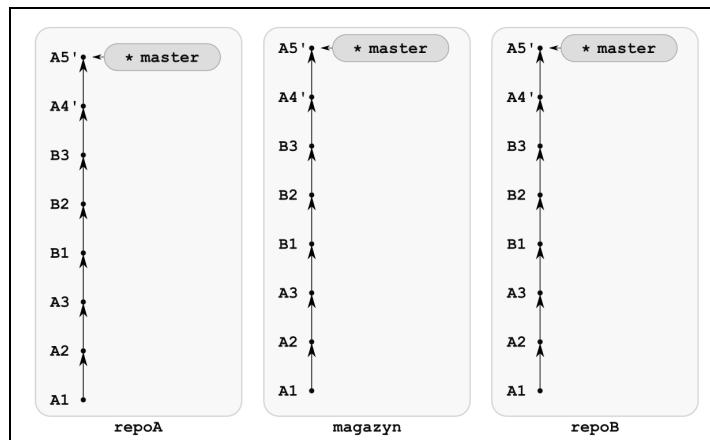
Wysyłanie dowolnej gałęzi

Polecenie:

```
git push repozytorium-zdalne nazwa-galezi
```

powoduje wysłanie do podanego repozytorium podanej gałęzi. Jeśli taka gałąź nie istniała, to zostanie utworzona.

Rysunek 18.17.
Stan końcowy
repozytoriów
w ćwiczeniu 18.4

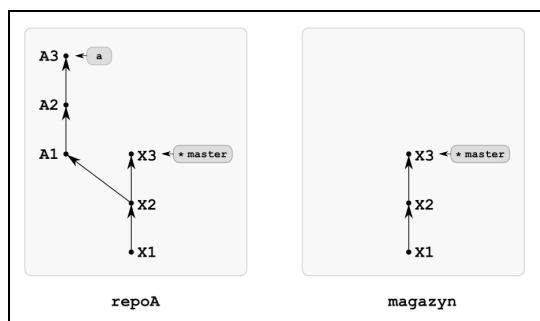


Jeśli dane są dwa repozytoria: *repoA* oraz *magazyn*, które znajdują się w stanie przedstawionym na rysunku 18.18², to w wyniku wydania polecen:

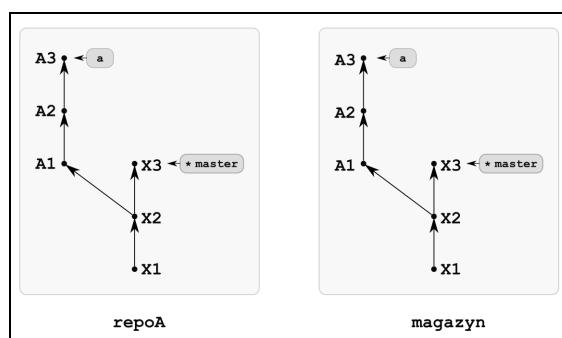
```
git add remote magazyn C:\sciezka\magazyn
git push magazyn a
```

stan repozytorium *magazyn* zostanie przekształcony do postaci z rysunku 18.19.

Rysunek 18.18.
Dwa repozytoria,
z których *magazyn*
jest klonem *repoA*
wykonanym przed
dodaniem w *repoA*
rewizji *A1*, *A2* i *A3*



Rysunek 18.19.
Stan repozytoriów
z rysunku 18.18 po
wydaniu w *repoA*
polecenia git push
magazyn a



² Zakładam, że repozytorium *magazyn* powstało jako klon repozytorium *repoA*.



Wskazówka

Gałąź bieżącą możemy wysłać do repozytorium zdalnego poleceniem:

```
git push repozytorium-zdalne HEAD
```

np.:

```
git push origin HEAD
```

Przełączanie na gałąź zdalną

Polecenie:

```
git checkout nazwa-gałęzi
```

zmienia bieżącą gałąź. Jeśli gałąź o podanej nazwie nie istnieje w repozytorium lokalnym, a istnieje w repozytorium zdalnym, wówczas polecenie spowoduje:

- ◆ utworzenie gałęzi lokalnej o podanej nazwie,
- ◆ przejście na nową gałąź lokalną,
- ◆ pobranie zawartości gałęzi zdalnej o podanej nazwie do gałęzi lokalnej,
- ◆ ustalenie, że dla gałęzi lokalnej gałęzią śledzoną jest gałąź zdalna.

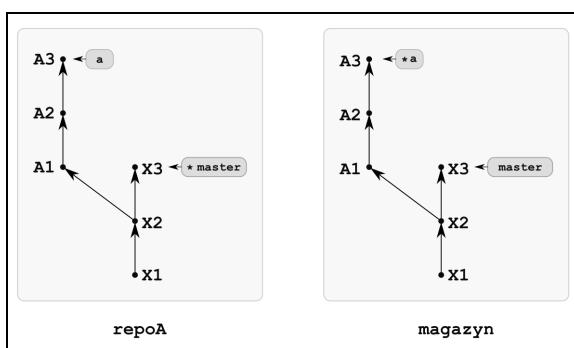
Dla repozytoriów z rysunku 18.18 polecenie:

```
git checkout a
```

wydane w repozytorium *magazyn* da efekt przedstawiony na rysunku 18.20.

Rysunek 18.20.

Stan repozytoriów z rysunku 18.18 po wydaniu w repozytorium *magazyn* polecenia *git checkout b*



Przesyłanie gałęzi ze zmianą nazwy

Polecenie:

```
git push origin nazwa-lokalna:nazwa-zdalna
```

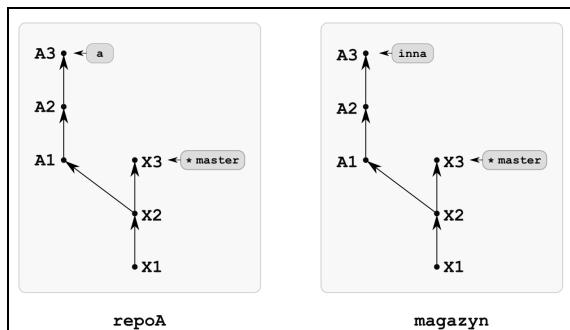
umożliwia przesyłanie gałęzi ze zmianą nazwy. Jeśli w repozytorium z rysunku 18.18 wydamy polecenie:

```
git push magazyn a:inna
```

otrzymamy repozytorium przedstawione na rysunku 18.21.

Rysunek 18.21.

Stan repozytoriów z rysunku 18.18 po wydaniu w repozytorium repoA polecenia git push magazyn a:inna



Usuwanie gałęzi zdalnych

Polecenie:

```
git push repository-zdalne :nazwa-galezi-zdalnej
```

usuwa gałąź zdalną o podanej nazwie.

Jeśli w repozytorium z rysunku 18.21 wydamy komendę:

```
git push magazyn :inna
```

otrzymamy repozytorium z rysunku 18.18.

Zabezpieczanie przed utratą rewizji

Repozytorium zdalne, które wykorzystujemy do synchronizacji rewizji, możemy zabezpieczyć przed utratą rewizji. Przy domyślnych ustawieniach konfiguracyjnych polecenie:

```
git push -f
```

spowoduje przesyłanie bieżącej gałęzi do gałęzi śledzonej, nawet jeśli taka operacja powoduje utratę rewizji. Po włączeniu opcji konfiguracyjnych:

```
git config receive.denyDeletes true
git config receive.denyNonFastForwards true
```

operacje git push -f, powodujące utratę rewizji, będą odrzucane.

Polecenie backup

W projektach jednoosobowych przydatne bywa polecenie `git backup`, które powoduje wykonanie poleceń:

```
git add --all .
git commit -m "..."
git push
```

W celu zdefiniowania takiego polecenia w globalnym pliku `.gitconfig` dodajemy definicję widoczną na listingu 18.1.

Listing 18.1. Definicja polecenia `git backup`³

```
backup =
  !git add --all . &&
  git commit -m "\"$@\\""
  shift 1 &&
  git push &&
  echo Saved and stored on server!
```

Dzięki temu bieżący stan projektu możemy zapisać w postaci rewizji i wysłać na serwer jednym poleceniem:

```
git backup "..."
```

Przesyłanie gałęzi do repozytorium zwykłego

Domyślna konfiguracja repozytoriów nie pozwala na wykonanie operacji:

```
git push
```

jeśli repozytorium zdalne jest repozytorium zwykłym. Zachowanie takie możemy zmienić, stosując opcję konfiguracyjną `receive.denyCurrentBranch`. Oprogramowanie zdarzenia `post-update` pozwoli dodatkowo przywrócić stan plików w obszarze roboczym repozytorium zdalnego.

W celu zezwolenia na wykonywanie operacji `git push` należy w repozytorium zdalnym, w pliku `.git/config`, wprowadzić opcję przedstawioną na listingu 18.2.

Listing 18.2. Opcja, która zezwala na wykonanie operacji `git push` dla repozytorium zwykłego

```
[receive]
denyCurrentBranch = ignore
```

³ Oczywiście polecenie należy zapisać w jednym wierszu.

Opcję z listingu 18.2 możemy ustalić poleceniem:

```
git config receive.denyCurrentBranch ignore
```

Jeśli teraz w repozytorium lokalnym wydamy polecenie:

```
git push adr-repo-zdalnego master
```

to do repozytorium zdalnego zostaną przesłane rewizje z gałęzi master. Powyższe polecenie zostanie poprawnie wykonane nawet wtedy, gdy repozytorium zdalne jest zwykłe.

Jeśli jednak zajrzymy do obszaru roboczego repozytorium zdalnego, to nie znajdziemy w nim najnowszych plików. Polecenie:

```
git push
```

przesyła rewizje, ale nie modyfikuje obszaru roboczego. Obecność rewizji w repozytorium zdalnym stwierdzimy, wydając komendę:

```
git log
```

Jeśli chcemy, by polecenie git push przesyłało rewizje i ustalało zawartość obszaru roboczego, należy oprogramować obsługę zdarzenia post-update. W tym celu należy w repozytorium zdalnym utworzyć plik *.git/hooks/post-update* o treści przedstawionej na listingu 18.3.

Listing 18.3. Procedura obsługi zdarzenia

```
#!/bin/sh  
  
exec git reset --hard
```

Skrypt z listingu 18.3 powoduje, że za każdym razem, gdy zawartość repozytorium zostanie zaktualizowana, wykonane w nim zostanie polecenie:

```
git reset --hard
```

Ostatnią modyfikacją, jaką należy wykonać, jest ustalenie w repozytorium zdalnym położenia obszaru roboczego. W tym celu w konfiguracji repozytorium dodajemy opcję widoczną na listingu 18.4.

Listing 18.4. Opcja ustalająca położenie obszaru roboczego

```
[core]  
worktree = ../
```

Ćwiczenie 18.5

Wykonaj dwa repozytoria zwykłe: *repoLokalne* i *repoDeploy*. Repozytoria te skonfiguruj w taki sposób, by polecenie:

```
git push repoDeploy master
```

wydane w repozytorium *repoLokalne* powodowało:

- ◆ przesłanie do repozytorium *repoDeploy* wszystkich rewizji z gałęzi master,
- ◆ zresetowanie zawartości folderu roboczego repozytorium *repoDeploy* do stanu odpowiadającego przesłanym rewizjom.

ROZWIĄZANIE

W dowolnym folderze utwórz dwa repozytoria:

```
git init repoLokalne  
git init repoDeploy
```

W repozytorium *repoLokalne* utwórz kilka rewizji:

```
git simple-loop a 5
```

Następnie w repozytorium *repoDeploy* utwórz plik *.git/hooks/post-update* o treści przedstawionej na listingu 18.3.

W kolejnym kroku poleceniami:

```
git config receive.denyCurrentBranch ignore  
git config core.worktree ../
```

aktualnij konfigurację repozytorium *repoDeploy*.

Jeśli teraz w repozytorium *repoLokalne* ustawisz adres repozytorium zdalnego:

```
git remote add repoDeploy ../repoDeploy
```

i wydasz komendę:

```
git push repoDeploy master
```

to w repozytorium *repoDeploy* zostaną dodane rewizje i zresetowany zostanie obszar roboczy. W obszarze roboczym repozytorium *repoDeploy* znajdziesz pliki *a1.txt*, *a2.txt*, ..., *a5.txt* z rewizji a1, a2, ..., a5, wykonanych w początkowej fazie ćwiczenia.

Ćwiczenie zakończ, tworząc w repozytorium *repoLokalne* kolejne rewizje:

```
//polecenie należy wydać w repozytorium repoLokalne  
git Simple-commits lorem ipsum dolor
```

Po wydaniu polecenia:

```
git push repoDeploy master
```

w obszarze roboczym repozytorium *repoDeploy* powinny znaleźć się pliki *lorem.txt*, *ipsum.txt* i *dolor.txt*.



Wskazówka

Rozwiązanie opisane w ćwiczeniu 18.5 może służyć do implementacji prostego systemu wdrażania (ang. *deployment*) aplikacji. Informacje na ten temat są dostępne w dokumentacji Gita (punkt pt. „Why won’t I see changes in the remote repo after »git push«?”):

<https://git.wiki.kernel.org/index.php/GitFaq>

oraz w artykułach:

<http://toroid.org/ams/git-website-howto>

<http://williamdurand.fr/2012/02/25/deploying-with-git/>

Rozdział 19.

Praktyczne wykorzystanie Gita — scenariusz pierwszy

Rozwiążanie opisane w ćwiczeniu 18.4 może być wdrożone w niewielkiej firmie. Podstawowym założeniem takiego rozwiązania jest wzajemne zaufanie pracowników. Innymi słowy przyjmujemy, że żaden z pracowników nie będzie świadomie dezorganizował pracy zespołu. Przed ewentualnymi błędami możemy się zabezpieczyć, ustalając opcje `receive.denyDeletes` oraz `receive.denyNonFastForwards` jako `true`.

Synchronizacja pracy grupy będzie wykonywana w repozytorium surowym o nazwie takiej jak projekt. Repozytorium takie nazwiemy głównym.

Każdy pracownik będzie dysponował własnym repozytorium, w którym będzie wykonywał zmiany (tj. będzie tworzył własne rewizje).

Wszystkie zmiany tworzymy w dedykowanych gałęziach we własnym repozytorium.

Sprawdzone zmiany scalamy z gałęzią `master` i przesyłamy do repozytorium głównego.

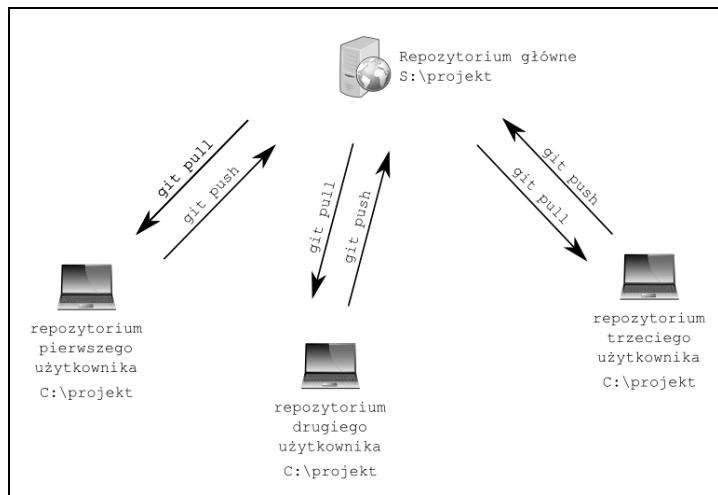
Operację przesyłania rewizji do repozytorium głównego poprzedzamy aktualnieniem własnej gałęzi `master`.

Nigdy nie modyfikujemy historii repozytorium głównego. Innymi słowy nie wykonujemy operacji `git push` z parametrem `-f`.

Opisany model pracy jest przedstawiony na rysunku 19.1.

Foldery oznaczone na rysunku 19.1 symbolami `C:\projekt` są prywatnymi folderami użytkowników. Folder `S:\projekt` jest folderem dostępnym na wspólnym dysku sieciowym. Folder `S:\projekt` należy udostępnić wszystkim uczestnikom projektu w trybie do odczytu i zapisu.

Rysunek 19.1.
Pierwszy scenariusz
wdrożenia
oprogramowania Git
w niewielkiej firmie



Inicjalizacja projektu

Projekt rozpoczyna dowolny z uczestników. Najpierw tworzy on własne prywatne repozytorium:

```
C:\>git init projekt
```

W repozytorium umieszczamy pierwszą rewizję (np. plik *README*).

Następnie tworzymy surowe repozytorium główne:

```
C:\>git clone --bare projekt S:\projekt
```

Użytkownik, który inicjalizował projekt, wydaje we własnym repozytorium komendy:

```
git add remote origin S:\magazyn
git branch --set-upstream master origin/master
```

dzięki czemu synchronizację gałęzi master będzie mógł wykonywać, tak jak pozostali uczestnicy projektu, poleceniami:

```
git push
git pull
```

Dołączanie do projektu

Osoba, która chce dołączyć do projektu, musi mieć dostęp do folderu zawierającego repozytorium główne. Na własnym komputerze klonujemy repozytorium główne:

```
git clone S:\projekt C:\jakis\folder\projekt
```

W ten sposób tworzymy własną kopię repozytorium głównego.

Ponieważ kopię wykonaliśmy poleceniem `git clone`, synchronizacja gałęzi master repozytoriów będzie wymagała jedynie komend:

```
git pull  
git push
```

Wprowadzanie zmian w projekcie

W celu wprowadzenia zmian w projekcie użytkownik przechodzi na własną lokalną gałąź master:

```
git checkout master
```

i aktualizuje ją:

```
git pull
```

Następnie tworzy kopię gałęzi master. Kopię nazywa dev:

```
git checkout -b dev
```

Teraz znajdujemy się w gałęzi dev. Tworzymy w niej własne rewizje. Gałąź doprowadzamy do stanu, gdy polecenie:

```
git status -s
```

zwraca pusty wynik.

W celu udostępnienia własnej pracy pozostałym uczestnikom projektu:

1. Przechodzimy do gałęzi master:

```
git checkout master
```

2. Uaktualniamy gałąź master:

```
git pull
```

3. Scalamy gałąź dev z gałęzią master:

```
git merge dev
```

4. Tworzymy liniową historię gałęzi master:

```
git rebase origin/master
```

5. Przesyłamy zmiany do repozytorium głównego

```
git push
```

Może się zdarzyć, że zanim wyślemy zmiany poleceniem `git push`, w repozytorium głównym pojawią się nowe rewizje. W takiej sytuacji uaktualniamy własną gałąź master poleceniami z punktów 2., 3. i 4.

Wykorzystywanie kilku gałęzi

Oczywiście prace możemy prowadzić w dowolnej liczbie gałęzi. W celu utworzenia gałęzi dev w repozytorium głównym dowolny z uczestników tworzy gałąź dev we własnym repozytorium, po czym wydaje komendę:

```
git push -u origin dev
```

W ten sposób gałąź dev pojawi się w repozytorium głównym.

Użytkownicy, którzy chcą pracować w gałęzi dev, wydają komendę:

```
git checkout dev
```

(zakładając, że nie mają własnej gałęzi dev; jeśli mają, to przed wydaniem powyższej komendy zmieniają nazwę własnej gałęzi dev).

Synchronizacja gałęzi dev odbywa się w identyczny sposób jak synchronizacja gałęzi master.



Ten tryb pracy możemy stosować, wykorzystując protokół SSH. Ułatwi to organizację grupy pracowników zdalnych. Instalacja repozytorium synchronizacyjnego przy użyciu SSH jest opisana w końcowej części rozdziału 27.

Rozdział 20.

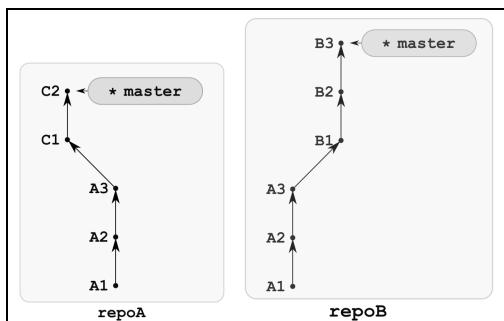
Łączenie oddzielnych repozytoriów

Opisana w rozdziale 18. procedura łączenia zawartości dwóch gałęzi zawartych w osobnych repozytoriach może dotyczyć zupełnie różnych gałęzi. Jeśli gałęzie nie będą zawierały żadnych wspólnych rewizji, to będziemy mieli do czynienia z łączeniem różnych repozytoriów.

Przyjmijmy, że dane są dwa repozytoria: *repoA* oraz *repoB*, które wywodzą się z jednego wspólnego repozytorium i zawierają własne nowe rewizje. Repozytoria takie są przedstawione na rysunku 20.1.

Rysunek 20.1.

Dwa repozytoria, które zawierają wspólne rewizje A1, A2 i A3



Jeśli w repozytorium *repoA* z rysunku 20.1 skonfigurujemy dostęp do repozytorium zdalnego *repoB*:

```
git remote add origin C:\jakis\folder\repoB
```

i poleceniem:

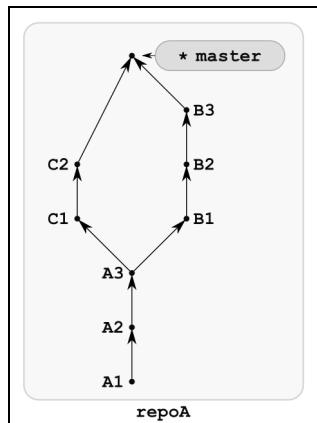
```
git pull origin master
```

pobierzemy rewizje, otrzymamy efekt taki jak na rysunku 20.2.

Historia repozytorium z rysunku 20.2 zawiera wspólny fragment historii A1, A2 i A3, po nim rozgałęzienie, nowe rewizje w oddzielnych gałęziach oraz rewizję łączącą.

Rysunek 20.2.

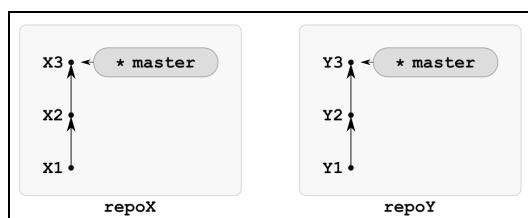
*Skutek wykonania
w repozytorium
repoA operacji
git pull origin master*



Spróbujmy taką samą operację wykonać na dwóch różnych repozytoriach *repoX* i *repoY*. Ponieważ repozytoria są różne, nie zawierają żadnych wspólnych rewizji. Każde z nich zawiera własne unikatowe rewizje. Repozytoria *repoX* oraz *repoY* są przedstawione na rysunku 20.3.

Rysunek 20.3.

*Dwa repozytoria
o zupełnie innej
historii*



Jeśli w repozytorium *repoX* dodamy adres repozytorium zdalnego:

```
git remote add origin C:\jakis\folder\repoY
```

po czym wydamy polecenie::

```
git pull origin master
```

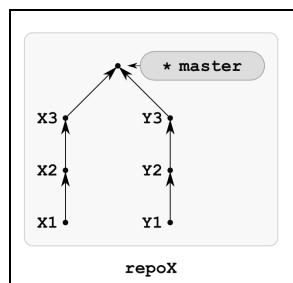
ujrzymy wówczas ostrzeżenie:

```
warning: no common commits
```

i otrzymamy repozytorium przedstawione na rysunku 20.4.

Rysunek 20.4.

*Repozytorium repoX
po wydaniu polecenia
git pull origin master*



W repozytorium z rysunku 20.4 występują dwie rewizje, które nie mają przodków. Są to rewizje X1 oraz Y1. Możemy to sprawdzić poleceniem:

```
git log --pretty=oneline --max-parents=0
```

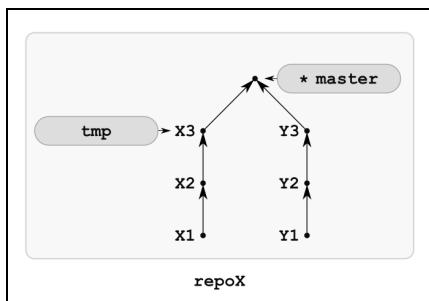
Jeśli chcemy, by historia repozytorium z rysunku 20.4 była liniowa, należy najpierw utworzyć gałąź tymczasową tmp wskazującą rewizję X3. Po wydaniu polecenia:

```
git branch tmp HEAD~1
```

repozytorium z rysunku 20.4 przyjmie postać przedstawioną na rysunku 20.5.

Rysunek 20.5.

Repozytorium z rysunku 20.4 po utworzeniu gałęzi tmp



Polecenie:

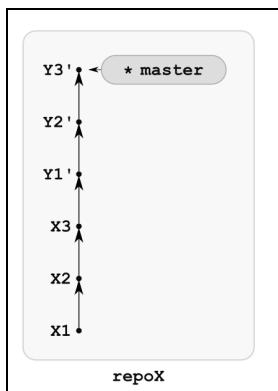
```
git rebase tmp
```

wydane w gałęzi master przeksztali repozytorium z rysunku 20.5 do postaci przedstawionej na rysunku 20.6. Po tej operacji gałąź tmp możemy usunąć:

```
git branch -d tmp
```

Rysunek 20.6.

Repozytorium z rysunku 20.5 po wydaniu polecenia git rebase tmp i usunięciu gałęzi tmp



Ćwiczenie 20.1

Sprawdź, które spośród projektów wymienionych w rozdziale 1. powstały przez połączenie kilku osobnych repozytoriów. Użyj polecenia:

```
git log --pretty=oneline --max-parents=0
```

Ćwiczenie 20.2

Utwórz dwa repozytoria *repoX* oraz *repoY* przedstawione na rysunku 20.3. Połącz ich historię, tworząc repozytorium o strukturze przedstawionej na rysunku 20.6.

Ćwiczenie 20.3

Utwórz repozytorium, które będzie zawierało całą historię projektów:

jQuery: <https://github.com/jquery/jquery>

Dojo: <https://github.com/dojotoolkit/dojotoolkit>

Pliki projektów umieść w osobnych folderach:

jQuery/
Dojo/

ROZWIĄZANIE

Sklonuj repozytorium jQuery:

```
git clone https://github.com/jquery/jquery
```

W folderze repozytorium utwórz folder *jQuery/*, przenieś do niego wszystkie pliki i foldery (z wyjątkiem folderu *.git/*) i wykonaj rewizje:

```
git add --all .
git commit -m "Pliki przeniesione do folderu jQuery/"
```

Sklonuj repozytorium Dojo:

```
git clone https://github.com/dojotoolkit/dojotoolkit
```

W folderze repozytorium utwórz folder *Dojo/*, przenieś do niego wszystkie pliki i foldery (z wyjątkiem folderu *.git/*) i wykonaj rewizje:

```
git add --all .
git commit -m "Pliki przeniesione do folderu Dojo/"
```

W repozytorium jQuery utwórz gałąź o nazwie *tmp*:

```
git branch tmp
```

Będzie ona wskazywała rewizje:

```
Pliki przeniesione do folderu jQuery/
```

Następnie ustal, by adres zdalny *lokalne-dojo* wskazywał repozytorium Dojo na dysku:

```
git remote add lokalne-dojo ../dojotoolkit
```

W repozytorium jQuery wydaj polecenie:

```
git pull lokalne-dojo master
```

Spowoduje ono dołączenie do repozytorium jQuery wszystkich rewizji z repozytorium lokalnego Dojo. Zawartość otrzymanego repozytorium sprawdź poleceniem:

```
git log --pretty=oneline --graph
```

W celu otrzymania repozytorium o liniowej historii wydaj komendę:

```
git rebase tmp
```

Wykonywanie tego polecenia zajmie około 3000 sekund. Repozytorium Dojo zawiera ponad 3000 rewizji, a przewijanie jednej rewizji zajmuje około sekundy.

Na zakończenie usuń gałąź tmp:

```
git branch -d tmp
```



Wskażówka

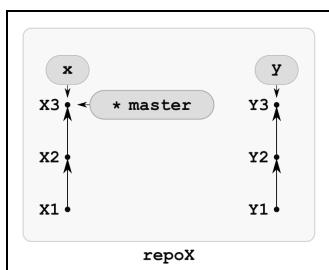
Oczywiście procedura łączenia dwóch oddzielnych repozytoriów jest w praktyce bardzo rzadko spotykana. Opisane ćwiczenia mają Ci uświadomić fakt, że łączenie gałęzi nie jest ograniczone. Połączyciemy każde dwie gałęzie występujące w dowolnych repozytoriach. Jeśli gałęzie się zawierają, będziemy mieli do czynienia z przewijaniem do przodu (rysunek 14.4). Jeśli gałęzie się nie zawierają, ale mają wspólnie początkowe rewizje, będziemy mieli do czynienia z sytuacją z rysunku 20.2. Jeśli natomiast gałęzie nie posiadają wspólnych rewizji, wystąpi przypadek przedstawiony na rysunku 20.5.

Graf niespójny

Stosując operacje łączenia oddzielnych repozytoriów i wycofywania operacji łączenia, możemy otrzymać niespójny graf rewizji. Przykład takiego repozytorium jest przedstawiony na rysunku 20.7. W jednym repozytorium będą zawarte dwa oddzielne repozytoria.

Rysunek 20.7.

Niespójny graf rewizji,
czyli repozytorium
zawierające dwa
oddzielne repozytoria



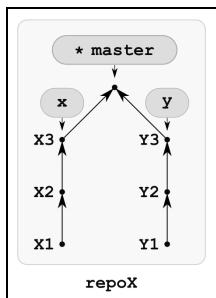
W celu otrzymania z repozytorium przedstawionego na rysunku 20.4 repozytorium widocznego na rysunku 20.7 należy najpierw utworzyć dwie gałęzie: x oraz y:

```
git branch x HEAD^1  
git branch y HEAD^2
```

Jeśli powyższe polecenia wydamy w repozytorium *repoX* z rysunku 20.4, otrzymamy repozytorium widoczne na rysunku 20.8.

Rysunek 20.8.

Repozytorium z rysunku 20.4 po utworzeniu gałęzi x oraz y



Jeśli w repozytorium z rysunku 20.8 wydamy komendę:

```
git reset --hard HEAD^
```

otrzymamy repozytorium z rysunku 20.7. Graf rewizji będzie zawierał dwie gałęzie niemające ani jednej rewizji wspólnej.

Ćwiczenie 20.4

Utwórz repozytorium przedstawione na rysunku 20.8.

Rozdział 21.

Podsumowanie części III

Część trzecia zawiera szczegółowe omówienie zagadnień dotyczących wiązania repozytoriów. W systemie Git powiązane repozytoria są równoprawne, żadne z nich nie odgrywa roli nadzędnej. Repozytorium, w którym wydajemy komendy, nazywamy **lokalnym**, a powiązane z nim repozytoria nazywamy **zdalnymi** (ang. *remote*).

Powiązanie dwóch repozytoriów sprowadza się do ustalenia adresów repozytoriów zdalnych. Jeśli dane są dwa osobne repozytoria repoA oraz repoB, wówczas powiązanie repozytorium repoA z repozytorium repoB będzie polegało na wydaniu komendy:

```
//komenda wydana w repozytorium repoA  
git remote add nazwa repoB
```

Po wydaniu powyższej komendy, pracując w repoA, powiemy, że jest to repozytorium lokalne, a repoB nazwiemy repozytorium zdalnym.

Oczywiście relacja ta może zostać odwrócona. Jeśli pracując w repoB, wydamy komendę:

```
//komenda wydana w repozytorium repoB  
git remote add nazwa repoA
```

wówczas repozytorium lokalnym będzie repoB, a zdalnym — repoA.

Oczywiście każde repozytorium możemy powiązać z wieloma różnymi repozytoriami zdalnymi, np.

```
git remote add nazwa1 adres1  
git remote add nazwa2 adres2  
git remote add nazwa3 adres3  
git remote add nazwa4 adres4  
...
```

W ten sposób powstaje rozproszony system powiązanych repozytoriów.

Synchronizacja zawartości repozytoriów polega na uzgodnieniu rewizji w poszczególnych gałęziach. Jeśli dwa repozytoria zawierają w danej gałęzi identyczne rewizje, gałęzie takie nazwiemy wówczas zsynchronizowanymi. Zwróci uwagę, że jeśli dwa

repozytoria są w pełni zsynchronizowane, to każde z nich zawiera pełną historię projektu i umożliwia odtworzenie dowolnego stanu plików. Rozproszony system powiązanych repozytoriów zmniejsza ryzyko utraty danych. Każde repozytorium możemy bowiem traktować jako kompletną kopię zapasową oryginalnego repozytorium.

Do synchronizacji gałęzi służą polecenia:

```
git pull  
git push
```

Ponieważ polecenie `git push` możemy wykonywać wyłącznie wtedy, gdy repozytorium zdalne jest tzw. repozytorium surowym, repozytoria wykorzystywane do synchronizacji będą repozytoriami surowymi.

Sposób synchronizacji repozytoriów oraz implementacja gałęzi jako wskaźników powoduje, że rozgałęzianie i scalanie gałęzi przebiega bardzo wydajnie. Tworzenie, usuwanie i łączenie gałęzi to operacje lokalne, niewymagające żadnej komunikacji sieciowej. Oczywiście synchronizacja gałęzi wymaga komunikacji sieciowej. Dzięki temu, że przekazywane są wyłącznie rewizje różniące gałąź lokalną i zdalną, także i te operacje są jednak wykonywane w sposób optymalny.

Co powinieneś umieć po lekturze trzeciej części?

Po przećwiczeniu materiału z trzeciej części powinieneś umieć:

- ◆ dodawać, listować i usuwać adresy zdalne (`git remote add`, `git remote -v`, `git remote rm`),
- ◆ synchronizować repozytorium lokalne z repozytorium zdalnym (`git pull`),
- ◆ przesyłać rewizje do repozytorium zdalnego (`git push`),
- ◆ definiować relację śledzenia pomiędzy gałęziami.

Lista poznanych poleceń

Ręczne klonowanie repozytorium

```
git init  
git remote add origin adres-klonowanego-repozytorium  
git fetch --no-tags origin master:refs/remotes/origin/master  
git branch --set-upstream master origin/master  
git reset --hard HEAD
```

Repozytoria zdalne

Definiowanie adresu repozytorium zdalnego

```
git remote add nazwa adres
```

Listowanie adresów repozytoriów zdalnych:

```
git remote -v
```

Usuwanie adresu repozytorium zdalnego:

```
git remote rm nazwa
```

Gałęzie śledzone

Sprawdzanie gałęzi śledzonych dla gałęzi X:

```
git config --get branch.X.remote  
git config --get branch.X.merge
```

Ustalanie gałęzi śledzonej:

```
git branch --set-upstream galaz-lokalna repozytorium-zdalne/galaz-zdalna
```

np.

```
git branch --set-upstream master origin/master
```

Równoważny sposób definiowania gałęzi śledzonej master (należy wydać oba polecenia):

```
git config branch.master.remote origin  
git config branch.master.merge refs/heads/master
```

Listowanie gałęzi

Gałęzie lokalne:

```
git branch
```

Gałęzie zdalne:

```
git branch -r
```

Wszystkie gałęzie:

```
git branch -a
```

Synchronizacja gałęzi

Pobieranie gałęzi zdalnej:

```
git fetch nazwa-repozytorium-zdalnego nazwa-gałezi-zdalnej
```

np.

```
git fetch origin master
```

Pobieranie gałęzi śledzonej dla gałęzi bieżącej:

```
git fetch
```

Pobieranie i łączenie gałęzi zdalnej:

```
git merge repozytorium-zdalne/galaż-zdalna
```

np.

```
git merge origin/master
```

Wykonywanie operacji git merge dla gałęzi śledzonej:

```
git merge
```

Pobieranie gałęzi zdalnej, łączenie z gałęzią bieżącą i przywracanie obszaru roboczego:

```
git pull nazwa-repozytorium-zdalnego nazwa-galezi-zdalnej
```

Wykonywanie operacji git pull dla gałęzi śledzonej:

```
git pull
```

Przesyłanie zawartości bieżącej gałęzi do repozytorium zdalnego:

```
git push nazwa-repozytorium-zdalnego nazwa-galezi-zdalnej
```

np.

```
git push my abc
```

Przesyłanie gałęzi ze zmianą nazwy:

```
git push nazwa-repozytorium-zdalnego nazwa-galezi-lokalnej:nazwa-galezi-zdalnej
```

np.

```
git push my lokalna-loreum:zdalna-ipsum
```

Przesyłanie gałęzi bieżącej do repozytorium zdalnego (wydane w gałęzi xyz prześle gałąź xyz):

```
git push nazwa-repozytorium-zdalnego HEAD
```

Przesyłanie gałęzi bieżącej do gałęzi śledzonej:

```
git push
```

Przesyłanie gałęzi bieżącej i wiązanie gałęzi zdalnej z lokalną:

```
git push -u nazwa-repozytorium-zdalnego nazwa-galezi-zdalnej
```

Przesyłanie gałęzi lokalnej z nadpisaniem historii:

```
git push -f
```

```
git push -f nazwa-repozytorium-zdalnego nazwa-galezi-zdalnej
```

```
git push -f nazwa-repozytorium-zdalnego nazwa-galezi-lokalnej:nazwa-galezi-zdalnej
```

Przełączanie na gałąź zdalną:

```
git checkout nazwa-gałęzi-zdalnej
```

Usuwanie gałęzi zdalnej:

```
git push repozytorium-zdalne :nazwa-gałęzi-zdalnej
```

Zabezpieczanie repozytorium

Zabezpieczanie przed utratą rewizji:

```
git config receive.denyDeletes true  
git config receive.denyNonFastForwards true
```

Przydatne skróty

Wykonywanie rewizji i przesyłanie gałęzi bieżącej na serwer:

```
git backup "..."
```


Część IV

Treść pliku

Rozdział 22.

Konflikty

Podczas pracy grupowej wcześniej czy później spotkamy się z sytuacją, w której ten sam plik jest poddany dwóm różnym modyfikacjom. Jeśli zmiany dotyczą tej samej linijki pliku, Git nie będzie mógł automatycznie złączyć obu wersji plików. Zdarzenie takie nazywamy **konfliktami**. Dochodzi do niego podczas łączenia gałęzi poleceniami:

```
git merge  
git rebase
```

Konflikt tekstowy: wynik operacji git merge

Aby zbadać zachowanie Gita w przypadku wystąpienia konfliktu, wykonajmy repozytorium, które będzie zawierało w pierwszej rewizji jeden plik o nazwie *tekst.txt* o treści:

```
łorem  
ipsum  
dolor
```

Następnie utwórzmy dwie gałęzie: a i b. W gałęzi a w pliku *tekst.txt* zamieńmy wyraz *ipsum* na ABC, a w gałęzi b zamieńmy wyraz *ipsum* na DEF. Zmiany wykonane w obu gałęziach zatwierdzamy w osobnych rewizjach. Otrzymamy repozytorium przedstawione na rysunku 22.1.

Zwróć uwagę, że podczas przygotowywania repozytorium z rysunku 22.1 nie otrzymamy żadnych ostrzeżeń. Tworząc rewizje, możemy modyfikować dowolne pliki. Git nie blokuje dostępu do żadnych plików.

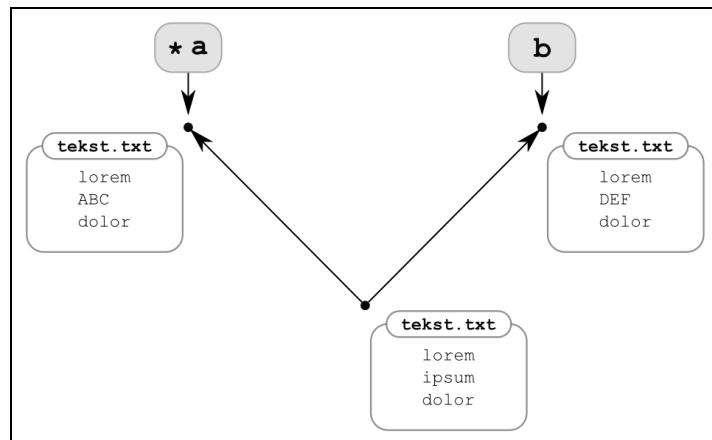
Konflikt pojawia się podczas łączenia gałęzi zawierających pokrywające się zmiany.

Jeśli w gałęzi a repozytorium z rysunku 22.1 wydamy polecenie:

```
git merge b
```

Rysunek 22.1.

W gałęzi a wiersz ipsum zmieniono na ABC, a w gałęzi b na DEF



otrzymamy komunikat informujący o tym, że automatyczne łączenie gałęzi zakończyło się konfliktem w pliku *tekst.txt*:

```

Auto-merging tekst.txt
CONFLICT (content): Merge conflict in tekst.txt
Automatic merge failed; fix conflicts and then commit the result.
  
```

Sprawdźmy stan repozytorium:

```
git status -s
```

Plik zawierający konflikt jest oznaczony symbolem UU:

```
UU tekst.txt
```

W pliku *tekst.txt* oprócz treści znajdziemy specjalne znaczniki:

```
<<<<<
=====
>>>>>
```

Treść pliku *tekst.txt* po wydaniu polecenia `git merge b` w repozytorium z rysunku 22.1 jest przedstawiona na listingu 22.1.

Listing 22.1. Konflikt występujący w pliku *tekst.txt* po wydaniu w repozytorium z rysunku 22.1 polecenia `git merge b`

```
lorem
<<<<< HEAD
ABC
=====
DEF
>>>>> b
dolor
```

Znaczniki <<<<<, ===== oraz >>>>> określają fragmenty, które spowodowały konflikt. Pomiędzy znacznikami:

```
<<<<< HEAD  
...(treść gałęzi bieżącej a)  
=====
```

zawarta jest treść gałęzi bieżącej¹. Pomiędzy znacznikami:

```
=====  
...(treść z gałęzi b)  
>>>>> b
```

zawarta jest treść gałęzi dołączanej, czyli b. W dowolnym edytorze tekstu należy usunąć wszystkie znaczniki. Przykładowo doprowadźmy plik z listingu 22.1 do postaci z listingu 22.2. Treść pliku możemy ustalić dowolnie: w przykładzie przedstawionym na listingu 22.2 wykorzystana została treść zarówno gałęzi a, jak i b. Równie dobrze w miejsce napisów:

```
ABC  
DEF
```

widocznych na listingu 22.2 możemy wpisać dowolne inne teksty.

Listing 22.2. *Plik z listingu 22.1 po rozwiążaniu konfliktu*

```
torem  
ABC  
DEF  
dolor
```

Gdy treść pliku *tekst.txt* jest taka jak na listingu 22.2, poleceniami:

```
git add --all .  
git commit
```

wykonujemy rewizję. Polecenie `git commit` wywołane bez parametru powoduje uruchomienie edytora tekstu. Tym razem domyślny opis rewizji zawiera oprócz standartowego komunikatu o łączeniu rewizji także informację o konfliktach:

```
Merge branch 'b' into a
```

```
Conflicts:  
  tekst.txt
```

Otrzymamy repozytorium przedstawione na rysunku 22.2.

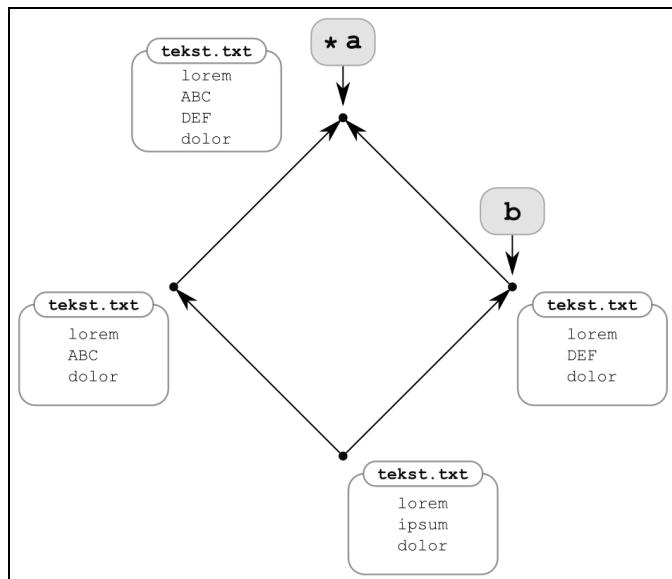
Ćwiczenie 22.1

Przygotuj repozytorium z rysunku 22.1, po czym poleceniem `git merge` połącz gałęzie a i b. Po usunięciu konfliktu powinieneś otrzymać repozytorium z rysunku 22.2.

¹ Tj. gałęzi, w której wydaliśmy polecenie `git merge`; w przykładzie była to gałąź a.

Rysunek 22.2.

Repozytorium z rysunku 22.1 po wydaniu komendy git merge b i rozwiązaniu konfliktu zgodnie z listingiem 22.2



Konflikt tekstowy: wynik operacji git rebase

Spróbujmy połączyć gałęzie a i b z rysunku 22.2 tak, by otrzymać repozytorium o nowej historii. W gałęzi a wydajmy komendę:

```
git rebase b
```

Otrzymamy komunikat o konflikcie:

```
...
Auto-merging tekst.txt
CONFLICT (content): Merge conflict in tekst.txt
...
```

Zawartość pliku *tekst.txt* będzie tym razem taka jak na listingu 22.3. Repozytorium znajdzie się w stanie detached HEAD.



Wskazówka Jeśli operacja git rebase zakończy się konfliktami, to repozytorium pozostanie w stanie detached HEAD.

Listing 22.3. Konflikt występujący w pliku *tekst.txt* po wydaniu w repozytorium z rysunku 22.1 polecenia *git rebase b*

```
lorem
<<<<< HEAD
DEF
```

```
=====
ABC
>>>>> a
Dolor
```

Znaczniki konfliktu są na listingach 22.1 oraz 22.3 identyczne. Zwróć uwagę, że na listingu 22.3 zmiany są przedstawione w innej kolejności, pomimo tego, że w obu przypadkach znajdowaliśmy się w gałęzi a.

Rozwiążmy konflikt, modyfikując treść pliku z listingu 22.3 do postaci widocznej na listingu 22.4.

Listing 22.4. Plik z listingu 22.3 po usunięciu konfliktu

```
lorem
DEF
ABC
dolor
```

W celu dokończenia operacji git rebase najpierw zaindeksuj pliki, w których rozwiązałeś konflikty:

```
git add --all .
```

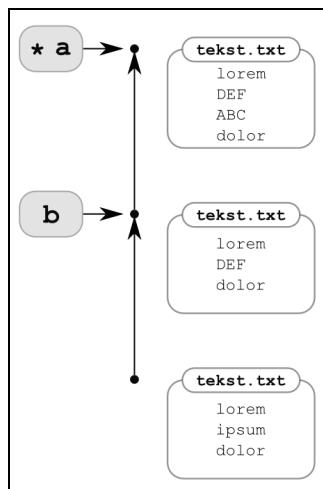
po czym wydaj komendę:

```
git rebase --continue
```

Otrzymasz repozytorium przedstawione na rysunku 22.3.

Rysunek 22.3.

Repozytorium z rysunku 22.1 po wydaniu komendy git rebase b i rozwiązaniu konfliktu zgodnie z listingiem 22.4



Repozytorium powróci ze stanu detached HEAD do gałęzi a.

Ślad po opuszczeniu gałęzi a znajdziemy, analizując wpisy w dzienniku reflog:

```
git reflog
```

Ćwiczenie 22.2

Przygotuj repozytorium z rysunku 22.1, po czym gałęzie a i b połącz polecieniem git rebase. Powinieneś otrzymać repozytorium z rysunku 22.3.

Dublowanie konfliktów przez operacje merge i rebase

Co się stanie, gdy po operacji:

```
git merge b
```

wykonamy operację:

```
git rebase b
```

Okazuje się, że konflikt — pomimo tego, że został rozwiązyany podczas operacji git merge — po wydaniu komendy git rebase pojawi się ponownie. Jeśli w repozytorium z rysunku 22.2 wydasz komendę:

```
git rebase b
```

to otrzymasz konflikt przedstawiony na listingu 22.3. Innymi słowy jeśli łączenie gałęzi będziesz wykonywać dwoma poleceniami:

```
git merge  
git rebase
```

to konflikty będziesz musiał rozwiązywać dwukrotnie.

Konflikty binarne

Oczywiście znaczniki kolizji:

```
<<<<<  
=====  
>>>>>
```

dotyczą wyłącznie plików tekstowych. W plikach binarnych znaczniki tego typu nigdy nie są dodawane. Co się dzieje, gdy kolizję spowoduje plik binarny występujący w obu gałęziach? Należy wtedy zdecydować, którą wersję chcemy zostawić. Do przywrócenia wersji pliku z gałęzi bieżącej służy polecenie:

```
git checkout --ours nazwa-pliku
```

Do przywrócenia wersji z gałęzi dołączanej służy polecenie:

```
git checkout --theirs nazwa-pliku
```

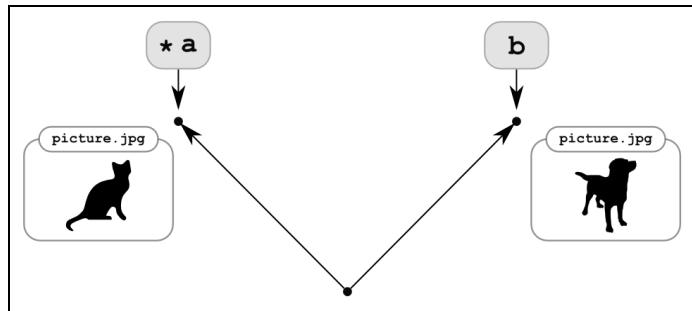
Domyślnie pozostawiana jest wersja z gałęzi bieżącej.

Konflikt binarny: wynik operacji git merge

Przygotujmy repozytorium, które będzie zawierało w osobnych gałęziach pliki *picture.jpg* różniące się treścią. Przykład takiego repozytorium jest przedstawiony na rysunku 22.4.

Rysunek 22.4.

Repozytorium
zawierające
w gałęziach a i b plik
picture.jpg, który
w wyniku łączenia
gałęzi a i b spowoduje
konflikt binarny



Jeśli w repozytorium z rysunku 22.4 wydamy polecenie:

```
git merge b
```

ujrzymy komunikat o konflikcie:

```
warning: Cannot merge binary files: picture.jpg (HEAD vs. b)
Auto-merging picture.jpg
CONFLICT (add/add): Merge conflict in picture.jpg
Automatic merge failed; fix conflicts and then commit the result.
```

Operacja łączenia zostanie przerwana. W obszarze roboczym plik *picture.jpg* będzie pochodził z gałęzi a (tj. będzie przedstawiał kota). Polecenie:

```
git checkout --theirs picture.jpg
```

przywraca w obszarze roboczym plik *picture.jpg* z gałęzi b (tj. rysunek przedstawiający psa).

Polecenie:

```
git checkout --ours picture.jpg
```

przywraca natomiast w obszarze roboczym plik *picture.jpg* z gałęzi a (rysunek przedstawiający kota).

Po wybraniu wersji pliku powodującego konflikt operację `git merge` kończymy poleceniami:

```
git add -a
git commit
```

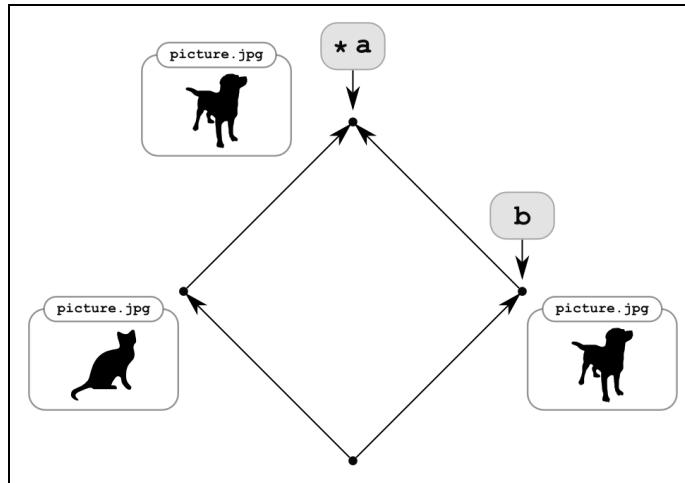
Jeśli w repozytorium z rysunku 22.4 wydamy kolejno polecenia z listingu 22.5, otrzymamy wówczas repozytorium z rysunku 22.5.

Listing 22.5. Polecenia, które repozytorium z rysunku 22.4 przekształci do postaci z rysunku 22.5

```
git merge b
git checkout --theirs picture.jpg
git add --all .
git commit
```

Rysunek 22.5.

Repozytorium
z rysunku 22.4 po
wydaniu polecen
z listingu 22.5



Ćwiczenie 22.3

Przygotuj repozytorium z rysunku 22.4, po czym poleceniami z listingu 22.5 połącz gałęzie a i b. Powinieneś otrzymać repozytorium z rysunku 22.5.

Konflikt binarny: wynik operacji git rebase

Jeśli w repozytorium z rysunku 22.4 wydamy polecenie:

```
git rebase b
```

ujrzymy wówczas komunikat o konflikcie:

```
First, rewinding head to replay your work on top of it...
Applying: a
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
warning: Cannot merge binary files: picture.jpg (HEAD vs. a)
```

Operacja łączenia zostanie przerwana. W obszarze roboczym plik *picture.jpg* będzie pochodził z gałęzi b (tj. będzie przedstawiał psa). Polecenie:

```
git checkout --theirs picture.jpg
```

przywraca w obszarze roboczym plik *picture.jpg* z gałęzi a (tj. rysunek przedstawiający kota).

Polecenie:

```
git checkout --ours picture.jpg
```

przywraca natomiast w obszarze roboczym plik *picture.jpg* z gałęzi b (rysunek przedstawiający psa).



Wskazówka

Polecenia:

```
git checkout --theirs picture.jpg  
git checkout --ours picture.jpg
```

działają inaczej w przypadku operacji git merge, a inaczej podczas operacji git rebase.

Po wybraniu wersji pliku powodującego konflikt operację git rebase kończymy poleceniami:

```
git add --all .  
git rebase --continue
```

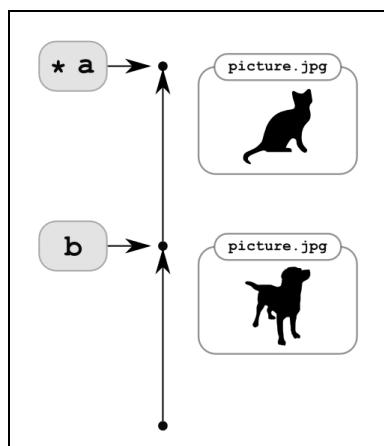
Jeśli w repozytorium z rysunku 22.4 wydamy kolejno polecenia z listingu 22.6, otrzymamy wówczas repozytorium z rysunku 22.6.

Listing 22.6. Polecenia, które repozytorium z rysunku 22.4 przekształca do postaci z rysunku 22.6

```
git rebase b  
git checkout --theirs picture.jpg  
git add --all .  
git rebase --continue
```

Rysunek 22.6.

Repozytorium z rysunku 22.4 po wydaniu polecień z listingu 22.6



Ćwiczenie 22.4

Przygotuj repozytorium z rysunku 22.4, po czym poleceniami z listingu 22.6 połącz gałęzie a i b. Powinieneś otrzymać repozytorium z rysunku 22.6.

Przywracanie plików do postaci z łączonych gałęzi

Polecenia:

```
git checkout --theirs *
git checkout --ours *
```

możesz wykorzystywać do automatycznego rozwiązywania zarówno konfliktów binarnych, jak i tekstowych. Jeśli po wykonaniu w gałęzi a operacji łączenia git merge b wszystkie pliki mają pochodzić z gałęzi a, użyj komendy:

```
//wszystkie pliki przywracamy do stanu z gałęzi a
git checkout --ours *
```

a jeśli z gałęzi b:

```
//wszystkie pliki przywracamy do stanu z gałęzi b
git checkout --theirs *
```

W przypadku łączenia wykonanego w gałęzi a operacją git rebase b znaczenie parametrów --ours i --theirs jest odwrotne. Komenda:

```
//wszystkie pliki przywracamy do stanu z gałęzi a
git checkout --theirs *
```

przywraca pliki do stanu z gałęzi a, a polecenie:

```
//wszystkie pliki przywracamy do stanu z gałęzi b
git checkout --ours *
```

do stanu z gałęzi b.

Polecenia checkout i show

Plik zawarty w każdej rewizji możemy odtworzyć poleceniami:

```
git checkout [SHA-1] nazwa-pliku
git show [SHA-1]:nazwa-pliku
```

Oczywiście w powyższych poleceniach możemy użyć dowolnych identyfikatorów rewizji, np. odwołań do przodków lub gałęzi. Poprawnymi poleceniami są:

```
git checkout HEAD~2 tekst.txt
git show b:tekst.txt
```

Polecenie `git checkout` przywraca stan pliku w obszarze roboczym, a komenda `git show` wyświetla zawartość pliku odpowiadającego podanej rewizji bez modyfikowania obszaru roboczego repozytorium. Dzięki temu możemy obejrzeć konfliktujące pliki bez konieczności modyfikowania obszaru roboczego.

Rozdział 23.

Badanie różnic

Do porównywania treści plików służy komenda:

```
git diff
```

Stosując ją, możemy porównać rewizje, indeks oraz obszar roboczy. Wywołana bez parametrów sprawdza różnice pomiędzy obszarem roboczym a plikami aktualnymi (tj. zapisanymi w rewizjach).

Jeśli w repozytorium z rysunku 22.1 wydamy polecenie:

```
git diff a b
```

otrzymamy wydruk widoczny na listingu 23.1.

Listing 23.1. Wynik działania polecenia `git diff a b` dla repozytorium z rysunku 22.1

```
diff --git a/tekst.txt b/tekst.txt
index 192c835..2255d8d 100644
--- a/tekst.txt
+++ b/tekst.txt
@@ -1,3 +1,3 @@
    lorem
-ABC
+DEF
 dolor
```

Polecenie `git diff` działa na bazie narzędzi GNU diffutils:

<http://www.gnu.org/software/diffutils/>

Pełna dokumentacja pakietu GNU diffutils jest dostępna na stronie:

<http://www.gnu.org/software/diffutils/manual/>

Informacje widoczne na listingu 23.1 są przedstawione w ogólnym formacie opisany w rozdziale 2.2.2 dokumentacji pakietu diffutils, pt. „Szczegółowy opis ujednoliconego formatu”.



Uwaga

Oznaczenia:

--- a/tekst.txt
+++ b/tekst.txt

nie są związane z nazwami gałęzi a i b. Polecenie git diff plik w wersji początkowej oznacza jako:

a/nazwa-pliku

a plik w wersji końcowej jako:

b/nazwa-pliku

Nazwy gałęzi, z których pochodzą pliki, nie mają znaczenia.

Ujednolicony format opisu zmian diffutils

W ujednoliconym formacie opisu zmiany są przedstawione w postaci:

```
--- a/tekst.txt
+++ b/tekst.txt
@@ -1,3 +1,3 @@
    lorem
-ABC
+DEF
 dolor
```

Początkowy fragment:

```
--- a/tekst.txt
+++ b/tekst.txt
```

mówią, że porównywane są dwie wersje pliku *tekst.txt*. Pierwsza wersja jest oznaczona jako:

--- a/tekst.txt

a druga jako:

+++ b/tekst.txt

Przedstawiana różnica wyjaśnia, w jaki sposób z pliku z wersji początkowej (tj. a/nazwa-pliku) otrzymać plik końcowy (tj. b/nazwa-pliku). Innymi słowy wydruk polecenia git diff opisuje transformację:

- ◆ z wersji początkowej oznaczonej ---
- ◆ do wersji końcowej oznaczonej +++.

Zapis:

@@ -1,3 +1,3 @@

zawiera dwie sekcje:

-1,3
+1,3

Sekcja poprzedzona znakiem - określa, jak na podstawie prezentowanego wydruku otrzymać stan początkowy, a sekcja poprzedzona znakiem + ustala, jak otrzymać stan końcowy.

Podane liczby a,b mówią, że:

- ♦ Pierwszym widocznym wierszem jest wiersz o numerze a.
- ♦ Widocznych jest b wierszy.

Zapis:

7,15

ustala, że:

- ♦ Pierwszy widoczny wiersz ma numer 7.
- ♦ Na wykazie widać 15 wierszy.

Poniżej sekcji:

@@ -1,3 +1,3 @@

prezentowane są fragmenty pliku z wersji początkowej i końcowej. Wiersze rozpoczynające się spacją są identyczne w obu wersjach. Wiersze rozpoczynające się znakiem + są dodane w wersji końcowej. Wiersze rozpoczynające się znakiem - są usunięte z wersji początkowej.

Z wydruku:

```
1orem  
-ABC  
+DEF  
dolor
```

wynika, że początkowa wersja pliku zawiera wiersze:

```
1orem  
ABC  
dolor
```

a końcowa:

```
1orem  
DEF  
dolor
```

Wiersz:

-ABC

został usunięty z wersji początkowej, a wiersz:

+DEF

dodany w wersji końcowej.

Liczba prezentowanych wierszy to tzw. **kontekst**, który domyślnie wynosi trzy wiersze. Zakres kontekstu możemy zmieniać parametrem `--unified` polecenia git diff. Polecenie:

```
git diff --unified=10
```

pokaże każdą zmianę, poprzedzając ją dziesięcioma niezmienionymi wierszami i dodając po zmianie dziesięć niezmienionych wierszy.

Przygotujmy repozytorium zawierające plik o nazwie *numerowane-wiersze.txt* o treści przedstawionej na listingu 23.2. Plik z listingu 23.2 zapiszmy w repozytorium, wykonując rewizję.

Listing 23.2. *Plik numerowane-wiersze.txt ułatwiający zrozumienie ujednoliconego formatu opisu zmian*

```
Wiersz 1...
Wiersz 2...
Wiersz 3...
Wiersz 4...
Wiersz 5...
Wiersz 6...
Wiersz 7...
Wiersz 8...
Wiersz 9...
Wiersz 10...
Wiersz 11...
Wiersz 12...
Wiersz 13...
Wiersz 14...
Wiersz 15...
Wiersz 16...
Wiersz 17...
Wiersz 18...
Wiersz 19...
Wiersz 20...
Wiersz 21...
Wiersz 22...
Wiersz 23...
Wiersz 24...
Wiersz 25...
Wiersz 26...
Wiersz 27...
Wiersz 28...
Wiersz 29...
Wiersz 30...
```

Następnie w pliku z listingu 23.2 usuńmy wiersz o numerze 13. Otrzymamy plik z listingu 23.3.

Listing 23.3. *Plik z listingu 23.2 po usunięciu wiersza 13*

```
Wiersz 1...
Wiersz 2...
Wiersz 3...
Wiersz 4...
Wiersz 5...
```

Wiersz 6...
Wiersz 7...
Wiersz 8...
Wiersz 9...
Wiersz 10...
Wiersz 11...
Wiersz 12...
Wiersz 14...
Wiersz 15...
Wiersz 16...
Wiersz 17...
Wiersz 18...
Wiersz 19...
Wiersz 20...
Wiersz 21...
Wiersz 22...
Wiersz 23...
Wiersz 24...
Wiersz 25...
Wiersz 26...
Wiersz 27...
Wiersz 28...
Wiersz 29...
Wiersz 30...

Po wykonaniu zmiany widocznej na listingu 23.3 wydajmy polecenie:

```
git diff
```

Spowoduje ono wyświetlenie zmian pomiędzy zawartością repozytorium a obszarem roboczym. Ujednolicony opis zmiany pomiędzy listingami 23.2 oraz 23.3 jest przedstawiony na listingu 23.4.

Listing 23.4. Zmiana pomiędzy listingami 23.2 i 23.3 po wydaniu polecenia git diff

```
--- a/numerowane-wiersze.txt
+++ b/numerowane-wiersze.txt
@@ -10,7 +10,6 @@
    Wiersz 10...
    Wiersz 11...
    Wiersz 12...
-   Wiersz 13...
    Wiersz 14...
    Wiersz 15...
    Wiersz 16...
```

Informacja **-10,7** określa zakres wierszy dla stanu początkowego. Prezentowane wiersze rozpoczynają się od wiersza o numerze 10 i jest ich 7.

Informacja **+10,6** określa zakres wierszy dla stanu końcowego. Prezentowane wiersze rozpoczynają się od wiersza o numerze 10 i jest ich 6.

Zmieńmy kontekst, nadając mu wartość 1:

```
git diff --unified=1
```

Wydruk przyjmie postać widoczną na listingu 23.5

Listing 23.5. Zmiana pomiędzy listingami 23.2 i 23.3 po wydaniu polecenia `git diff --unified=1`

```
@@ -12,3 +12,2 @@
 Wiersz 12...
-Wiersz 13...
 Wiersz 14...
```

Zwiększenie kontekstu do 10:

```
git diff --unified=10
```

da efekt przedstawiony na listingu 23.6.

Listing 23.6. Zmiana pomiędzy listingami 23.2 i 23.3 po wydaniu polecenia `git diff --unified=10`

```
@@ -3,21 +3,20 @@
 Wiersz 3...
 Wiersz 4...
 Wiersz 5...
 Wiersz 6...
 Wiersz 7...
 Wiersz 8...
 Wiersz 9...
 Wiersz 10...
 Wiersz 11...
 Wiersz 12...
-Wiersz 13...
 Wiersz 14...
 Wiersz 15...
 Wiersz 16...
 Wiersz 17...
 Wiersz 18...
 Wiersz 19...
 Wiersz 20...
 Wiersz 21...
 Wiersz 22...
 Wiersz 23...
```

Jak wynika z listingów 22.4, 22.5 oraz 22.6, kontekst to nic innego jak liczba wierszy zawartych na wydruku i otaczających wykonaną zmianę.

W pliku przedstawionym na listingu 23.3 dodajmy przed wierszem siódmym wiersz z tekstem *lorem*. Otrzymamy plik widoczny na listingu 23.7.

Listing 23.7. Plik z listingu 23.3 po dodaniu przed wierszem siódmym wiersza *lorem*

```
Wiersz 1...
Wiersz 2...
Wiersz 3...
Wiersz 4...
Wiersz 5...
Wiersz 6...
lorem
```

Wiersz 7...
Wiersz 8...
Wiersz 9...
Wiersz 10...
Wiersz 11...
Wiersz 12...
Wiersz 14...
Wiersz 15...
Wiersz 16...
Wiersz 17...
Wiersz 18...
Wiersz 19...
Wiersz 20...
Wiersz 21...
Wiersz 22...
Wiersz 23...
Wiersz 24...
Wiersz 25...
Wiersz 26...
Wiersz 27...
Wiersz 28...
Wiersz 29...
Wiersz 30...

Tym razem wykonaliśmy w pliku dwie zmiany. Polecenie:

```
git diff --unified=1
```

wyświetli dwa osobne bloki zmian przedstawione na listingu 23.8.

Listing 23.8. Zmiana pomiędzy listingami 23.2 i 23.7 po wydaniu polecenia `git diff --unified=1`

```
--- a/numerowane-wiersze.txt
+++ b/numerowane-wiersze.txt
@@ -6,2 +6,3 @@
    Wiersz 6...
+lorem
    Wiersz 7...
@@ -12,3 +13,2 @@
    Wiersz 12...
-Wiersz 13...
    Wiersz 14...
```

Dodawanie wiersza *lorem* jest opisane blokiem:

```
@@ -6,2 +6,3 @@
    Wiersz 6...
+lorem
    Wiersz 7...
```

a usuwanie wiersza 13 blokiem:

```
@@ -12,3 +13,2 @@
    Wiersz 12...
-Wiersz 13...
    Wiersz 14...
```

Ponieważ kontekst jest ograniczony do jednego wiersza poprzedzającego i następującego po zmianie, obie wykonane zmiany są prezentowane niezależnie.

Jeśli kontekst zwiększymy do 4:

```
git diff --unified=4
```

obie zmiany z listingu 23.7 będą prezentowane jednym blokiem widocznym na listingu 23.9.

Listing 23.9. Zmiana pomiędzy listingami 23.2 i 23.7 po wydaniu polecenia git diff --unified=4

```
--- a/numerowane-wiersze.txt
+++ b/numerowane-wiersze.txt
@@ -3,15 +3,15 @@
 Wiersz 2...
 Wiersz 3...
 Wiersz 4...
 Wiersz 5...
 Wiersz 6...
 +lorem
 Wiersz 7...
 Wiersz 8...
 Wiersz 9...
 Wiersz 10...
 Wiersz 11...
 Wiersz 12...
 -Wiersz 13...
 Wiersz 14...
 Wiersz 15...
 Wiersz 16...
 Wiersz 17...
```

Podsumowując, wyniki generowane przez polecenie `git diff` są prezentowane w postaci:

```
--- a/nazwa-pliku
+++ b/nazwa-pliku
@@ -a,b +c,d @@
 XXX
 +YYY
 -ZZZ
 XXX
```

Stan pliku przed zmianą był:

```
//wersja początkowa
XXX
ZZZ
XXX
```

a po zmianie:

```
//wersja końcowa
XXX
YYY
XXX
```

Zakres:

-a,b

określa, że wersja początkowa:

xxx

zzz

xxx

rozpoczyna się od wiersza o numerze a i zawiera b wierszy.

Zakres:

+c,d

określa, że wersja końcowa:

xxx

yyy

xxx

rozpoczyna się od wiersza c i zawiera d wierszy.

Ćwiczenie 23.1

Przygotuj repozytorium zawierające plik z listingu 23.2. Wykonaj modyfikacje z listingu 23.3, po czym wygeneruj wydruki z listingów 23.4, 23.5 oraz 23.6.

Ćwiczenie 23.2

Przygotuj repozytorium zawierające plik z listingu 23.2. Wykonaj modyfikacje z listingu 23.7, po czym wygeneruj wydruki z listingów 23.8 oraz 23.9.

Szukanie zmienionych wyrazów

Do wyszukiwania zmienionych wyrazów służy opcja --word-diff. Polecenie:

```
git diff --word-diff
```

wyświetli informacje o zmienionych wyrazach.

Przygotujmy plik *pytania.txt* o treści widocznej na listingu 23.10.

Listing 23.10. Plik pytania.txt

Kto? Gdzie? Kiedy?

Po co? Dlaczego? Za ile?

Z jakim skutkiem?

Zapiszmy plik w rewizji, po czym zmieńmy pytanie:

Dlaczego?

na pytanie:

W jakim celu?

Treść zmienionego pliku *pytania.txt* jest przedstawiona na listingu 23.11.

Listing 23.11. Zmieniony plik *pytania.txt*

Kto? Gdzie? Kiedy?
Po co? W jakim celu? Za ile?
Z jakim skutkiem?

Polecenie:

git diff --word-diff --unified=0

zwróci wynik widoczny na listingu 23.12.

Listing 23.12. Wynik polecenia *git diff* z parametrem *-word-diff* dla plików z listingów 23.10 i 23.11

```
--- a/pytania.txt
+++ b/pytania.txt
@@ -2 +2 @@
Po co? [-Dlaczego?-]{+W jakim celu?+} Za ile?
```

Zapis:

@@ -2 +2 @@

mówi o tym, że wydruk prezentuje wyłącznie jedną linijkę o numerze 2. W linijce tej usunięto wyraz:

[-Dlaczego?-]

i wstawiono tekst:

{+W jakim celu?+}

Ćwiczenie 23.3

Przygotuj repozytorium zawierające plik z listingu 23.10. Wykonaj modyfikacje z listingu 23.11, po czym wygeneruj wydruk z listingu 23.12.

Szukanie zmienionych plików

W celu wyświetlenia nazw zmienionych plików stosujemy parametr *--name-only*.
Polecenie:

git diff --name-only

wydrukuje nazwy plików, których treść została zmieniona. Komenda:

```
git diff --name-only master dev
```

ustali nazwy plików, którymi różnią się gałęzie master i dev. Dodatkowy parametr *jakis/folder*:

```
git diff --name-only master dev -- jakis/folder
```

pozwoli ograniczyć sprawdzane pliki do konkretnego folderu. Zwróć uwagę, że przed nazwą folderu *jakis/folder* należy użyć znacznika `--`, który kończy parametry poleceńia. W przeciwnym razie ciąg *jakis/folder* zostanie potraktowany jako identyfikator rewizji.

Wyszukiwanie rewizji, w których podany plik został zmieniony

Do wyszukiwania rewizji, w których podany plik został zmieniony, służy polecenie `git log`:

```
git log -- nazwa-pliku
```

Oczywiście podanie nazwy pliku możemy połączyć z parametrami formatującymi wydruk:

```
git log --abbrev-commit --abbrev=4 --pretty=oneline -- sprawdzanie-postaci-pliku.txt
```

Ćwiczenie 23.4

Sklonuj repozytorium jQuery, a następnie poleceniem:

```
git log --tags --simplify-by-decoration --pretty="format:%ai %d" | sort
```

sprawdź ostatnie dostępne znaczniki. Następnie poleceniem:

```
git diff --name-only 1.8.0 1.8rc1
```

sprawdź, jakie pliki zostały zmienione pomiędzy wersjami oznaczonymi znacznikami 1.8.0 oraz 1.8rc1.

Następnie poleceniem:

```
git log --oneline -- version.txt
```

sprawdź, w jakich rewizjach modyfikowany był plik *version.txt*.

Na zakończenie poleceniem:

```
git show 1.7:version.txt
```

wyswietl postać pliku *version.txt* z rewizji oznaczonej znacznikiem 1.7.

Rozdział 24.

Pliki tekstowe i binarne

Pliki tekstowe są traktowane przez Git w inny sposób niż pliki binarne. Różnice te dotyczą dwóch aspektów:

- ◆ metody rozwiązywania konfliktów
- ◆ oraz konwersji znaków końca wiersza.

W rozdziale 22. omówiliśmy dwie sytuacje konfliktowe. Pierwsza z nich, przedstawiona na rysunku 22.1, dotyczyła pliku *tekst.txt*. Drugi przypadek (widoczny na rysunku 22.4) dotyczył konfliktu pliku binarnego *picture.jpg*.

W przypadku plików tekstowych konflikty są rozwiązywane narzędziami GNU diffutils. Dzięki temu możemy poznać poszczególne wiersze pliku, które powodują konflikty. Wynikiem rozwiązywania konfliktu może być plik zawierający fragmenty obu wersji, które powodują konflikt.

Konflikty binarne są rozwiązywane znacznie prościej. Pliki nigdy nie są scalane ani edytowane. Rozwiążanie konfliktu polega na podjęciu decyzji, którą wersję pliku chcemy pozostawić.

Odróżnianie plików binarnych od tekstowych

W jaki sposób Git odróżnia pliki binarne od tekstowych? Przy domyślnych ustawieniach programu Git decyzja, czy plik jest tekstowy, czy binarny, jest podejmowana na podstawie zawartości pliku.

Zachowanie domyślne możemy zmienić, stosując tzw. atrybuty plików.



Opis atrybutów jest dostępny w dokumentacji:

`git attribute --help`

Atrybuty plików umieszczamy w pliku `.gitattributes` w folderze głównym repozytorium lub w pliku `.git/info/attributes`. Format pliku jest następujący:

nazwa-pliku *wartość-atrybutu-1* *wartość-atrybutu-2* ...

Nazwa pliku może zawierać znaki * oraz ?, a wartość atrybutu przyjmuje postać:

atrybut
-atrybut
atrybut=wartosc

Zapis:

nazwa-pliku *atrybut*

włącza podany atrybut. Instrukcja:

nazwa-pliku *-atrybut*

wyłącza atrybut, a:

nazwa-pliku *atrybut=wartosc*

ustala wartość atrybutu.

Atrybut diff — konflikty tekstowe i binarne

Atrybut `diff` ustala, czy plik ma być porównywany jako tekstowy, czy jako binarny. Wartość:

nazwa-pliku *diff*

powoduje, że plik będzie porównywany narzędziem `diff` jako plik tekstowy. Wartość:

nazwa-pliku *-diff*

ustala natomiast, że plik porównujemy jako binarny.

Ćwiczenie 24.1

W nowym repozytorium utwórz plik `.gitattributes` i dodaj w nim wpis:

**.txt* *-diff*

Utwórz plik `opis.txt`, wprowadź w nim dowolny tekst i wykonaj rewizję. Następnie zmodyfikuj treść pliku `opis.txt` i wydaj komendę:

`git diff`

Ujrzyś komunikat:

Binary files a/opis.txt and b/opis.txt differ

Plik tekstowy `opis.txt` jest zatem traktowany jako plik binarny.

Konwersja znaków końca wiersza

Systemy operacyjne Linux, Windows oraz Mac OS różnią się znakami końca wiersza stosowanymi w plikach tekstowych¹. Repozytoria Gita tworzymy w celu organizacji pracy grupowej, dlatego problem ten jest bardzo istotny. Jeśli w pracy grupowej uczestniczą programiści stosujący różne platformy, to musimy zwrócić uwagę na konwersje znaków końca wiersza.

Git umożliwia wykonanie automatycznej konwersji znaków końca wiersza podczas tworzenia rewizji (czyli zapisywania pliku w repozytorium) oraz podczas przywracania pliku w obszarze roboczym. O wykonywanej konwersji decydują opcje konfiguracyjne:

```
core.autocrlf  
core.eol
```

oraz atrybuty:

```
text  
eol
```

Globalne wyłączenie konwersji znaków końca wiersza

W celu wyłączenia konwersji znaków końca wiersza należy opcji:

```
core.autocrlf
```

nadać wartość:

```
false
```

Po wydaniu polecenia:

```
git config --global core.autocrlf false
```

w żadnym repozytorium nie będą wykonywane żadne konwersje znaków końca wiersza — ani podczas operacji commit, ani podczas operacji checkout. Stan ten odpowiada najniższej opcji z rysunku 2.2.

Globalna normalizacja znaków końca wiersza

Jeśli opcja konfiguracyjna:

```
core.autocrlf
```

przyjmuje wartość:

```
true
```

¹ Por. <http://pl.wikipedia.org/wiki/End-of-line>.

wówczas podczas umieszczania plików tekstowych w rewizjach (tj. podczas operacji `commit`) oraz odtwarzania plików w obszarze roboczym (tj. podczas operacji `checkout`) będzie wykonywana konwersja znaków końca wiersza. Takie ustawienia odpowiadają najwyższej opcji w oknie dialogowym z rysunku 2.2.

Pliki zapisywane w repozytorium otrzymają końca wiersza `LF` (tj. `\n`). Podczas odtwarzania treści pliku w obszarze roboczym (tj. podczas operacji `checkout`) pliki tekstowe otrzymają końca wiersza ustalone opcją konfiguracyjną:

```
core.eol
```

Opcja ta może przyjąć wartości:

```
lf  
crlf  
native
```

Wartości `lf` i `crlf` wymuszają użycie konkretnego końca wiersza, zaś wartość `native` powoduje użycie końców wiersza zależnych od danej platformy. Domyślne ustawienia:

```
git config --global core.autocrlf true  
git config --global core.eol native
```

powodują, że we wszystkich repozytoriach pliki tekstowe zapisane w rewizjach będą stosowały znak `\n` (LF). Pliki w obszarze roboczym będą stosowały znak końca wiersza platformy, na której pracujemy. Powyższe polecenia powodują dodanie w globalnym pliku konfiguracyjnym `.gitconfig` wpisów:

```
[core]  
autocrlf = true  
eol = native
```

Jeśli w podanych wyżej poleceniach pominiemy opcję `--global`:

```
git config core.autocrlf true  
git config core.eol native
```

wówczas podane parametry będą dotyczyć wyłącznie bieżącego repozytorium.



Każde repozytorium może zawierać inne ustawienia opcji `core.autocrlf` oraz `core.eol`.

Projekty wieloplatformowe

Jeśli w projektach uczestniczą programiści pracujący na wielu platformach, zalecane jest wówczas nadanie opcji `core.autocrlf` wartości `input`:

```
git config --global core.autocrlf input
```

Odpowiada to środkowej opcji z okna dialogowego widocznego na rysunku 2.2.

Przy takich ustawieniach operacja `commit` będzie powodowała konwersję znaków końca wiersza z `CRLF` do `LF`. Operacja `checkout` nie będzie powodowała żadnej konwersji znaków końca wiersza.

Ustalenie konwersji znaków końca wiersza dla konkretnych plików

Stosując atrybuty plików, możemy zdefiniować dowolną metodę konwersji znaków końca wiersza indywidualnie dla każdego pliku. Atrybut:

`text`

ustala, że plik jest tekstowy i ma być objęty automatyczną konwersją. Wyłączenie atrybutu:

`-text`

powoduje wyłączenie wszelkich konwersji końca znaków dla danego pliku. Jeśli więc w pliku `.gitattributes` umieścimy wpis:

```
*.yml    text  
*.xml   -text
```

wówczas pliki o rozszerzeniu `*.yml` będą poddawane automatycznej konwersji znaków końca wiersza, zaś w plikach `*.xml` żadne konwersje nie będą wykonywane.

Dodatkowy atrybut:

`eol`

pozwala wymusić stosowanie konkretnego znaku końca wiersza w obszarze roboczym. Jeśli w pliku `.gitattributes` dodamy wpisy:

```
*.txt      eol=crlf  
*.html     eol=lf
```

wówczas pliki `*.txt` będą stosowały znak końca wiersza `crlf`, a pliki `*.html` — znak `lf`.

Rozdział 25.

Podsumowanie części IV

Część czwarta zapoznała Cię z konfliktami. Dowiedziałeś się:

- ◆ w jakich okolicznościach występują konflikty;
- ◆ jak są raportowane konflikty tekstowe, a jak binarne;
- ◆ w jaki sposób można rozwiązywać konflikty.

Rozdział 23. zapoznał Cię z ujednoliconym formatem opisu różnic stosowanym przez pakiet diffutils. Dzięki temu będziesz mógł analizować wydruki generowane komendą git diff. Pamiętaj, że dodatkowe parametry komendy git diff pozwalają ustalić:

- ◆ zmodyfikowane wyrazy
- ◆ oraz nazwy zmodyfikowanych plików,

natomiast polecenie git log umożliwia wyszukiwanie rewizji, w których dany plik został poddany modyfikacjom.

Informacje zawarte w rozdziale 24. wyjaśniały, na jakiej podstawie Git klasyfikuje pliki jako tekstowe lub binarne oraz w jaki sposób możemy to zachowanie dostosować do własnych potrzeb.

Co powinieneś umieć po lekturze czwartej części?

Po lekturze czwartej części powinieneś umieć:

- ◆ stwierdzać wystąpienie konfliktu,
- ◆ analizować fragmenty powodujące konflikty,
- ◆ rozwiązywać konflikty tekstowe oraz binarne.

Pamiętaj, że jeśli operacja rozwiązywania konfliktu nie przebiegła zgodnie z Twoimi oczekiwaniemi, to zawsze możesz ją wycofać, przywracając repozytorium do stanu sprzed próby scalenia. Stan taki jest zapisany w dzienniku reflog.

Lista poznanych poleceń

Przerwana operacja git rebase

Kontynuacja operacji git rebase przerwanej przez konflikty:

```
git rebase --continue
```

Rezygnacja z dokończenia przerwanej operacji git rebase:

```
git rebase --abort
```

Przywracanie plików (po operacji git merge)

Przywracanie plików z gałęzi, w której wykonano operację git merge:

```
git checkout --ours nazwa-pliku
```

Przywracanie plików z gałęzi x dołączanej poleceniem git merge x:

```
git checkout --theirs nazwa-pliku
```

Przywracanie plików (po operacji git rebase)

Przywracanie plików z gałęzi, w której wykonano operację git rebase:

```
git checkout --theirs nazwa-pliku
```

Przywracanie plików z gałęzi x dołączanej poleceniem git rebase x:

```
git checkout --ours nazwa-pliku
```

Przywracanie plików

Przywracanie pliku w obszarze roboczym do postaci z podanej rewizji:

```
git checkout [SHA-1] nazwa-pliku
```

Wyświetlanie treści pliku z podanej rewizji (bez modyfikacji obszaru roboczego):

```
git show [SHA-1]:nazwa-pliku
```

Badanie różnic

Porównywanie obszaru roboczego i repozytorium:

```
git diff
```

Porównywanie wybranych gałęzi:

```
git diff --name-only master dev
```

Porównywanie zawartości folderu w dwóch gałęziach:

```
git diff --name-only master dev -- jakis/folder
```

Modyfikacja kontekstu:

```
git diff --unified=1
```

Badanie zmienionych wyrazów:

```
git diff --word-diff
```

Wyszukiwanie nazw zmienionych plików:

```
git diff --name-only
```

Wyszukiwanie rewizji, w których podany plik został zmieniony:

```
git log -- nazwa-pliku  
git log --abbrev-commit --abbrev=4 --pretty=oneline -- nazwa-pliku
```

Ustalanie metod konwersji znaków końca wiersza

Ustawienia z najwyższej opcji z rysunku 2.2:

```
git config --global core.autocrlf true  
git config --global core.eol native
```

Ustawienia z drugiej opcji z rysunku 2.2:

```
git config --global core.autocrlf input
```

Ustawienia z najniższej opcji z rysunku 2.2:

```
git config --global core.autocrlf false
```


Część V

Praca w sieci

Rozdział 26.

Serwisy **github.com** i **bitbucket.org**

Najwygodniejszą obecnie metodą współdzielenia repozytoriów Gita są serwisy internetowe:

- ◆ *github.com*
- ◆ *bitbucket.org*

Oprócz standardowych metod synchronizacji repozytoriów serwisy te:

- ◆ Umożliwiają kontrolę uprawnień użytkowników.
- ◆ Zawierają wbudowane systemy śledzenia błędów.
- ◆ Poprzez tzw. *żądania aktualizacji* zapewniają kontrolę akceptowanych rewizji.

Funkcjonalność obu serwisów: *github.com* oraz *bitbucket.org* jest zbliżona. Zasadnicza różnica dotyczy kwestii licencjonowania:

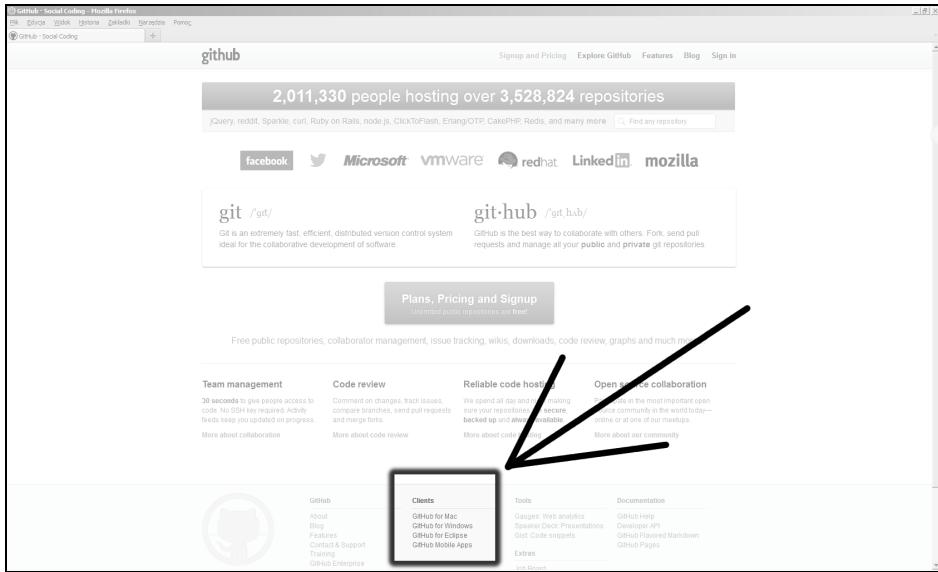
- ◆ Koszt użytkowania serwisu *github.com* jest zależny od liczby prywatnych repozytoriów; liczba współpracowników nie jest limitowana.
- ◆ Koszt użytkowania serwisu *bitbucket.org* jest zależny od liczby współpracowników; liczba prywatnych repozytoriów nie jest limitowana.

W obu serwisach możemy bezpłatnie prowadzić dowolną liczbę projektów dostępnych publicznie. Serwisy *github.com* i *bitbucket.org* nie nakładają żadnych ograniczeń na projekty publiczne. Nie jest limitowana ani liczba projektów, ani liczba współpracowników żadnego projektu.

Na korzyść serwisu *github.com* przemawia jego ogromna popularność w środowisku open source oraz dedykowane oprogramowanie ułatwiające pracę na wybranych platformach:

- ◆ <http://windows.github.com>
- ◆ <http://mac.github.com>
- ◆ <http://eclipse.github.com>

Informacje o najnowszych dedykowanych klientach Github znajdziesz w dolnej części strony głównej, w miejscu wskazanym na rysunku 26.1.



Rysunek 26.1. Informacje o dedykowanych klientach Github

Zaletami serwisu bitbucket.org są:

- ◆ możliwość tworzenia darmowych repozytoriów prywatnych;
- ◆ licencja akademicka, skierowana do studentów i pracowników uniwersytetów;
- ◆ możliwość tworzenia repozytoriów Gita oraz Mercuriala.

Wskazówka

Najpopularniejszymi rozproszonymi systemami kontroli wersji (ang. DVCS) są:

- ◆ Git,
- ◆ Mercurial,
- ◆ Baazar.

Obszerne zestawienie i porównanie systemów DVCS znajdziesz na stronie:

http://en.wikipedia.org/wiki/Comparison_of_revision_control_software

Tabela 26.1 prezentuje zestawienie najważniejszych cech serwisów github.com oraz bitbucket.org.

Tabela 26.1. Porównanie najważniejszych cech serwisów github.com oraz bitbucket.org

	github.com	bitbucket.org
Najpopularniejszy hosting projektów o otwartym kodzie	+	–
Możliwość tworzenia bezpłatnych repozytoriów prywatnych	–	+
Wbudowany system śledzenia błędów	+	+
Żądania aktualizacji	+	+
Obsługiwane VCS	Git	Git, Mercurial
Dedykowane oprogramowanie klienckie	Mac, Windows, Eclipse	–
Bezpłatna licencja akademicka	–	+
Zarządzanie dużą liczbą repozytoriów	Github umożliwia tworzenie organizacji: kont służących do grupowania repozytoriów	–
Zarządzanie uprawnieniami do repozytoriów	+	+

W celu rozpoczęcia pracy z serwisami należy się najpierw w każdym z nich zarejestrować. Rejestracja w obu serwisach jest bezpłatna.

W obu serwisach możemy tworzyć repozytoria publiczne oraz prywatne. Repozytoria publiczne są dostępne dla wszystkich w trybie do odczytu. Repozytoria prywatne są domyślnie niedostępne dla nikogo oprócz właściciela.

W każdym repozytorium możemy ustalić rozszerzone uprawnienia dostępu, które wybranym użytkownikom lub grupom pozwolą na dostęp w trybie do zapisu lub pełny dostęp administracyjny.

 **Wskazówka**

Szczegółowy opis interfejsu serwisów github.com oraz bitbucket.org został zawarty w książce za zgodą właścicieli. Serdecznie dziękuję właścicielom serwisów github.com oraz bitbucket.org za wyrażenie zgody na umieszczenie w książce ilustracji przedstawiających możliwości obu serwisów. Jednocześnie przepraszam wszystkich czytelników za ewentualne niezgodności prezentowanych ilustracji ze stanem faktycznym. Oba serwisy są bardzo intensywnie rozwijane i najprawdopodobniej w momencie ukazania się książki będą zawierały nowe cechy.

Serwis github.com

Interfejs serwisu github.com pozwala na tworzenie własnych repozytoriów oraz śledzenie repozytoriów innych użytkowników. Lista repozytoriów własnych jest dostępna na stronie głównej każdego użytkownika, a do repozytoriów śledzonych prowadzi odsyłacz *starred repos*. Rysunek 26.2 prezentuje listę repozytoriów własnych oraz hiperłącze do repozytoriów śledzonych.

Rysunek 26.2.
Repozytoria śledzone
oraz własne

The screenshot shows a GitHub user profile for 'gajdaw'. At the top right, there is a box containing the number '114' with an arrow pointing to it, and below it, '1' followed by 'starred repos'. The 'Repositories' section lists three repositories: 'symfony', 'symfony-standard', and 'symfony-docs'. Below the repositories, there is a 'Public Activity' feed with several recent events. A second arrow points to the 'Issues' section of the activity feed.



Repozytoria śledzone nie zajmują przestrzeni przyznanej użytkownikowi. Jeśli chcesz przeglądać wybrane repozytorium bez dodawania rewizji, użyj do tego operacji śledzenia.

W górnej części serwisu dostępny jest formularz do wyszukiwania. Wykorzystując go, wyszukaj repozytorium:

symfony/symfony

Przejdź na stronę wyszukanego repozytorium. Podstawowy interfejs aplikacji *github.com* służący do zarządzania wybranym repozytorium jest przedstawiony na rysunku 26.3.

Rysunek 26.3.
Podstawowy interfejs
aplikacji *github.com*

The screenshot shows the main interface for the 'symfony/symfony' repository. It includes a header with basic repository stats (Unwatched, Starred, Forked), a navigation bar with tabs for 'Code', 'Network', 'Pull Requests', 'Issues', and 'Graphs'. Below the header is a search bar and a clone button. The main content area displays a file tree for the 'master' branch, showing files like 'src', 'editorconfig', 'ignore', 'travis.yml', 'CHANGELOG-2.0.md', 'CONTRIBUTORS.md', 'LICENSE', 'README.md', 'UPGRADE-2.1.md', 'autoload.php.dist', 'composer.json', and 'phpunit.xml.dist'. Each file entry includes its last commit date and message. Arrows labeled A through D point to specific features: A points to the 'Issues' tab, B points to the 'Graphs' tab, C points to the 'Clone' button, and D points to the file list.

W obszarze roboczym aplikacji występuje:

- ♦ pasek informacji o wybranym repozytorium (strzałka A);
- ♦ menu główne wybranego repozytorium (strzałka B);
- ♦ menu kontekstowe, którego zawartość zależy od wybranej pozycji menu głównego (strzałka C).

W pasku informacji o wybranym repozytorium znajdziemy:

- ♦ nazwę repozytorium;
- ♦ ikonę informującą o tym, czy repozytorium jest publiczne, czy prywatne;
- ♦ przyciski *Watch/Unwatch* oraz *Star*, które umożliwiają śledzenie repozytorium;
- ♦ przycisk *Fork* umożliwiający tworzenie własnej kopii repozytorium.

Menu główne repozytorium (strzałka B) zawiera opcje:

- ♦ *Code*,
- ♦ *Network*,
- ♦ *Pull request*,
- ♦ *Issues*,
- ♦ *Graphs*.

Opcja *Code* pozwala na przeglądanie plików oraz rewizji zawartych w repozytorium.

Opcja *Network* umożliwia odnalezienie repozytoriów, które wywodzą się od wspólnego przodka.

Opcja *Pull request* służy do zarządzania żądaniami aktualizacji.

Opcja *Issues* prowadzi do zintegrowanego systemu śledzenia błędów.

Opcja *Graphs* udostępnia różnorodne statystyki (m.in. listę uczestników projektu — *Graphs/Contributors*).

Obszary oznaczone na rysunku 26.3 strzałkami C prezentują kontekstową zawartość dostępną na stronie opcji *Code*. Strzałka C wskazuje panel zawierający:

- ♦ przycisk *ZIP* umożliwiający pobranie spakowanego archiwum zip zawierającego pliki odpowiadające wybranej rewizji;
- ♦ adres repozytorium;
- ♦ przycisk *branch*, który służy do zmiany bieżącej gałęzi;
- ♦ opcje: *Files*, *Commits*, *Branches* oraz *Tags*, które umożliwiają przeglądanie plików, rewizji, gałęzi i znaczników.

Obszar oznaczony na rysunku 26.3 strzałką D prezentuje pliki i foldery zawarte w repozytorium. Pliki zawarte w obszarze D odpowiadają wybranej gałęzi (przycisk *branch*) oraz rewizji (opcja *Commit*).



W przypadku własnego repozytorium menu wskazane strzałką B będzie zawierało na końcu opcję *Admin*, umożliwiającą m.in. zarządzanie uprawnieniami do repozytorium oraz usunięcie repozytorium.

Ćwiczenie 26.1

Zarejestruj się w serwisie *github.com* i rozpoczęj śledzenie wszystkich repozytoriów wymienionych w rozdziale 1.

Ćwiczenie 26.2

Przeanalizuj zawartość repozytorium jQuery:

<https://github.com/jquery/jquery>

Sprawdź:

- ◆ gałęzie,
- ◆ znaczniki,
- ◆ listę uczestników projektu,
- ◆ rewizje uczestnika jeresig,
- ◆ listę rewizji zawartych w gałęzi master
- ◆ oraz stan plików odpowiadający wersji oznaczonej tagiem 1.2,

a następnie pobierz pakiet zip odpowiadający najnowszej rewizji.

Gałęzie i znaczniki poznasz, odwiedzając odsyłacze *Branches* i *Tags*.

Lista uczestników projektu jest zawarta na stronie *Graphs/Contributors*. Rewizje uczestnika jeresig poznasz, odwiedzając odsyłacz *1591 commits*¹ dostępny poniżej identyfikatora jeresig.

Rewizje zawarte w gałęzi master wyświetla opcja *Code/Commits*.

Stan plików odpowiadający znacznikowi 1.2 poznasz, odwiedzając odsyłacz *Code/Tags*. Na stronie znaczników wybierz hiperłącze *ce256312d5* dostępny poniżej odsyłacza *1.2.zip*.

W celu pobrania pliku zip odpowiadającego najnowszej rewizji wybierz przycisk ZIP dostępny w obszarze oznaczonym na rysunku 26.3 literą C.

¹ Liczba zawarta w nazwie odsyłacza będzie większa.

Ćwiczenie 26.3

Przeanalizuj zawartość repozytorium *symfony*:

<https://github.com/symfony/symfony>

Serwis bitbucket.org

Interfejs serwisu *bitbucket.org* dotyczący konkretnego repozytorium jest przedstawiony na rysunku 26.4.

Rysunek 26.4.
Podstawowy interfejs aplikacji *bitbucket.org*

The screenshot shows the Bitbucket application interface. At the top, there's a navigation bar with links like 'Explore', 'Dashboard', 'Repositories', and 'Włodzimierz Gajda'. Below the navigation is a search bar and a user profile icon. The main content area displays a repository named 'gajdaw / git-ksiazka'. It includes sections for 'Overview', 'Downloads (0)', 'Pull requests (0)', 'Source', 'Commits', 'Issues (0)', 'Admin', 'Forks/queues (0)', and 'Followers (1)'. There are also buttons for 'invite', 'RSS', 'fork', 'following', 'get source', 'compare', and 'create pull request'. A sidebar on the left lists 'branches' and 'tags'. The 'Recent commits' section shows five commits made by 'Włodzimierz Gajda' with their respective dates: '54 minutes ago', '20 hours ago', '2 days ago', '2 days ago', and '2 days ago'. Arrows labeled A, B, and C point to specific parts of the interface: arrow A points to the 'Recent commits' section; arrow B points to the 'Recent commits' section; arrow C points to the 'Recent commits' section.

Obszar oznaczony strzałką A zawiera menu główne z opcjami:

- ♦ *Overview* — przegląd informacji o repozytorium;
- ♦ *Downloads* — pobieranie spakowanego archiwum zawierającego pliki z repozytorium;
- ♦ *Pull requests* — żądania aktualizacji;
- ♦ *Source* — kod źródłowy zawarty w repozytorium;
- ♦ *Commits* — rewizje;
- ♦ *Issues* — zintegrowany system śledzenia błędów;
- ♦ *Admin* — administracja repozytorium (m.in. zarządzanie uprawnieniami oraz usuwanie repozytorium);

a także odsyłacze:

- ♦ *branches* — lista gałęzi repozytorium;
- ♦ *tags* — lista znaczników;
- ♦ *invite* — wysyłanie zaproszeń do współczestnictwa;
- ♦ *RSS* — kanał zawierający najnowsze rewizje;
- ♦ *fork* — tworzenie własnej kopii repozytorium;
- ♦ *follow/following* — odsyłacz włączający/wyłączający śledzenie repozytorium;

- ♦ *get source* — pobieranie spakowanego archiwum zawierającego pliki zawarte w repozytorium.

Jeśli w panelu wskazanym strzałką B widoczna jest informacja:

git clone ...

wówczas repozytorium jest prowadzone w systemie Git. Jeśli natomiast operacja klonowania jest przedstawiona jako:

hg clone ...

wówczas repozytorium jest prowadzone w systemie Mercurial.

Śledzenie repozytoriów umożliwiają przyciski *following/follow*, które są dostępne w obszarze wskazanym strzałką A.

Ćwiczenie 26.4

Zarejestruj się w serwisie *bitbucket.org* i rozpoczęź śledzenie repozytorium:

<https://bitbucket.org/gajdaw/wierszyki>

Ćwiczenie 26.5

Przeanalizuj zawartość repozytorium:

<https://bitbucket.org/gajdaw/wierszyki>

Sprawdź:

- ♦ gałęzie,
- ♦ znaczniki,
- ♦ listę rewizji zawartych w gałęzi master
- ♦ oraz stan plików gałęzi master,

a także pobierz pakiet zip odpowiadający najnowszej rewizji.

Gałęzie i znaczniki poznasz, odwiedzając odsyłacze *branches* oraz *tags*.

Rewizje zawarte w gałęzi master wyświetla opcja *Commits*.

Stan plików poznasz, odwiedzając odsyłacz *Source*.

W celu pobrania pliku zip odpowiadającego najnowszej rewizji wybierz przycisk *get source* dostępny w obszarze oznaczonym na rysunku 26.4 literą A.

Rozdział 27.

Klucze SSH

Oprogramowanie Git do transferu danych poprzez sieć może wykorzystywać protokoły:

- ◆ SSH,
- ◆ HTTPS,
- ◆ Git,
- ◆ file.

Serwisy *github.com* oraz *bitbucket.org* umożliwiają pracę wyłącznie za pomocą protokołu SSH¹, dlatego zanim przejdziemy do tworzenia własnych repozytoriów, konieczna jest konfiguracja protokołu SSH.

Najprostszym rozwiązaniem jest użycie dedykowanego klienta *github.com* (np. *http://windows.github.com*). Cały proces konfiguracji zostanie wtedy wykonany automatycznie. Wadą takiego rozwiązania jest to, że zainstalowany klient umożliwia współpracę wyłącznie z serwisem *github.com*. Ponieważ repozytoria prywatne przechowuję na serwerze *bitbucket.org*, nie korzystam z dedykowanych klientów *github.com*.

Opisana poniżej procedura konfiguracji SSH umożliwia korzystanie zarówno z serwisu *github.com*, jak i *bitbucket.org*.

Instalacja oprogramowania SSH w systemie Windows

Najnowsze wersje klienta Git dla systemu Windows zawierają w folderze:

C:\Program Files (x86)\Git\bin

¹ Do przeglądania zawartości repozytorium wystarczy protokół HTTP. Protokół SSH jest konieczny do wykonywania w repozytorium lokalnym poleceń git push, które przesyłają rewizje z dysku lokalnego na serwer.

oprogramowanie Open SSH. Jeśli podczas instalacji klienta Git w oknie dialogowym widocznym na rysunku 2.1 zaznaczysz opcję:

Run Git and included Unix tools from the Windows Command Prompt

wówczas ścieżki dostępu w wierszu poleceń zostaną zmodyfikowane i program SSH będzie dostępny w wierszu poleceń. W przeciwnym razie należy samodzielnie zmodyfikować ścieżki dostępu.



Wskazówka

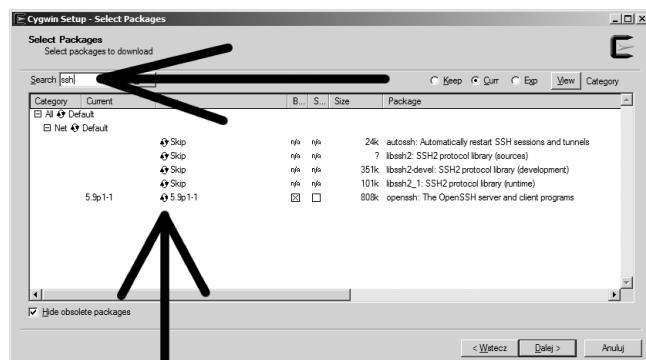
Jeśli w systemie Windows zachodzi konieczność ręcznej instalacji pakietu openssh lub innych narzędzi dostępnych dla systemów Linux, możemy wówczas użyć oprogramowania Cygwin (<http://www.cygwin.com>). Cygwin to kolekcja wielu narzędzi uniwersalnych przeniesionych do systemu Windows. Znajdziesz tam m.in. wspomniane narzędzia diffutils, git oraz Open SSH. Pobierz program instalacyjny *setup.exe* i uruchom go. Przy pierwszym uruchomieniu wykonaj operację:

Download without installing

W ten sposób pobierzesz wszystkie wybrane pakiety na dysk twardy.

Domyślna konfiguracja Cygwin zawiera tylko wybrane narzędzia; na przykład oprogramowanie Open SSH nie jest instalowane przy domyślnych ustawieniach. Z tego powodu po uruchomieniu programu instalacyjnego należy wyszukać i ręcznie włączyć pozycję Open SSH. Procedura wyszukiwania i włączania pakietu Open SSH jest zilustrowana na rysunku 27.1.

Rysunek 27.1.
Instalacja pakietu
Open SSH



Generowanie kluczy SSH

W celu wygenerowania klucza SSH uruchom wiersz poleceń i wydaj komendę:

```
ssh-keygen -t rsa -C "your_email@youremail.com"
```

Oczywiście tekst:

your_email@youremail.com

zastąp swoim adresem e-mail. Po wydaniu komendy ujrzyś trzy pytania:

```
Enter file in which to save the key (/home/nazwa-konta/.ssh/id_rsa):  
Enter passphrase (empty for no passphrase):  
Enter same passphrase again:
```

Na wszystkie z nich udziel domyślnej odpowiedzi, tj. naciśnij *Enter*. Zostaną wówczas utworzone pliki z kluczami²:

```
/home/nazwa-konta/.ssh/id_rsa  
/home/nazwa-konta/.ssh/id_rsa.pub
```

Pytanie:

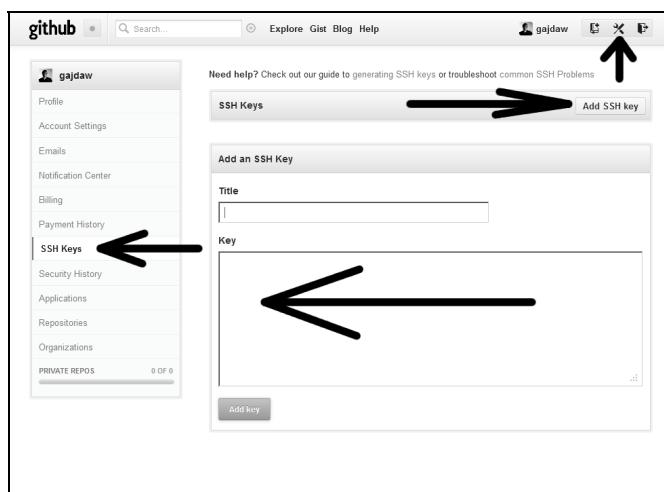
```
Enter passphrase (empty for no passphrase):
```

umożliwia nadanie dodatkowego hasła do klucza. Stosowanie tego hasła ma na celu zabezpieczenie dostępu do klucza SSH w przypadku, gdy plik z kluczem jest zapisany na komputerze, z którego korzystają inne osoby. Jeśli pracujesz na komputerze, do którego nikt inny nie ma dostępu, wówczas takie zabezpieczenie jest zbędne.

Konfiguracja klucza SSH na serwerze github.com

Aby skonfigurować dostęp za pomocą protokołu SSH do konta na serwerze github.com, należy zalogować się na własne konto i w ustawieniach konta dodać klucz SSH. Zawartość pliku *id_rsa.pub*, który został wygenerowany w poprzednim kroku, należy wkleić do okna przedstawionego na rysunku 27.2.

Rysunek 27.2.
Konfiguracja klucza SSH w serwisie github.com

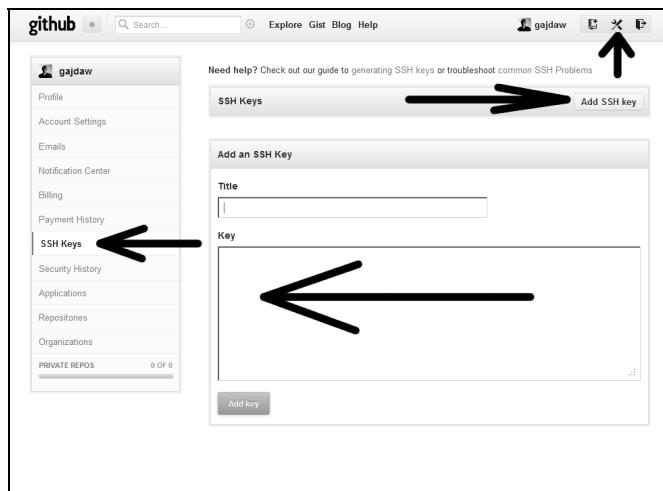


² W systemie Windows pliki te znajdziesz w folderze *C:\Użytkownicy\nazwa-konta*.

Konfiguracja klucza SSH na serwerze bitbucket.org

Konfiguracja klucza SSH w serwisie *bitbucket.org* przebiega identycznie. W oknie dialogowym przedstawionym na rysunku 27.3 należy wkleić zawartość pliku *id_rsa.pub*.

Rysunek 27.3.
Konfiguracja klucza
SSH w serwisie
bitbucket.org



Repozytorium zdalne na serwerze SSH

Repozytorium zdalne możemy także umieścić na własnym serwerze dostępnym za pomocą protokołu SSH. Praca będzie w takim przypadku przebiegała zgodnie ze scenariuszem pierwszym opisany w rozdziale 19.

Przyjmijmy, że dysponujemy kontem SSH, do którego dostęp zapewnia komenda:

```
ssh -p 123456 user@example.net
```

Najpierw tworzymy wówczas na serwerze nowe repozytorium surowe. W tym celu po zalogowaniu się na serwerze SSH wydajemy komendy:

```
mkdir repo.git
cd repo.git
git init --bare
```

Tak utworzone repozytorium będzie miało adres:

```
ssh://user@example.net:123456/~/repo.git/
```

Następnie tworzymy repozytorium lokalne. W dowolnym folderze wydajemy polecenie:

```
git clone ssh://user@example.net:123456/~/repo.git/ .
```

Tworzymy rewizje w lokalnym repozytorium:

```
git simple-commits a b c
```

Pierwszy raz wysyłając rewizje na serwer, używamy polecenia:

```
git push -u origin master
```

Do wykonywania kolejnych operacji wystarczy polecenie:

```
git push
```


Rozdział 28.

Tworzenie i usuwanie repozytoriów w serwisach **github.com** i **bitbucket.org**

Tworzenie repozytoriów w serwisach *github.com* oraz *bitbucket.org* odbywa się na jeden z dwóch sposobów. Pierwszą metodą jest zainicjowanie nowego pustego repozytorium niepowiązanego z żadnym innym projektem. Drugie rozwiązanie jest określone angielskim terminem *fork* i polega na klonowaniu istniejącego projektu. W tym rozdziale zajmiemy się tworzeniem nowych pustych repozytoriów.

Iinicjalizowanie nowego repozytorium: serwis **github.com**

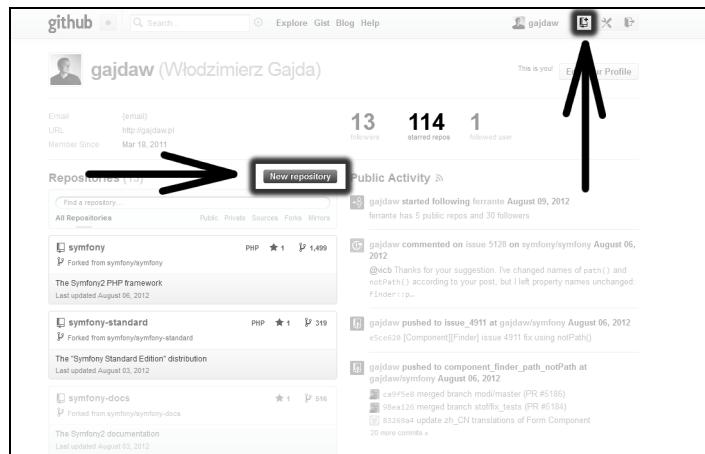
W serwisie *github.com* nowe repozytorium inicjalizujemy przyciskami *New repository* dostępnymi na stronie domowej użytkownika. Przyciski te zaznaczono na rysunku 28.1.

Po przejściu do strony tworzenia nowego projektu należy wprowadzić jego nazwę i ustalić, czy repozytorium ma być prywatne¹, czy publiczne. Dodatkowo możemy umieścić w repozytorium pierwszą rewizję zawierającą plik *README* oraz dodać plik *.gitignore*. Jeśli repozytorium zostanie zainicjalizowane plikiem *README*, będzie wówczas zawierało jedną rewizję, co umożliwi wykonywanie operacji klonowania.

¹ Tworzenie prywatnych repozytoriów będzie możliwe wyłącznie w przypadku korzystania z płatnego konta.

Rysunek 28.1.

Przyciski do tworzenia nowego repozytorium w serwisie *github.com*



Repozytoria puste, tj. niezawierające żadnych rewizji, nie mogą być klonowane.

Formularz do tworzenia nowego repozytorium w serwisie *github.com* jest przedstawiony na rysunku 28.2.

Rysunek 28.2.

Formularz do tworzenia nowego pustego repozytorium w serwisie *github.com*

The screenshot shows the 'Create repository' form. It asks for the owner (set to 'gajdaw') and the repository name ('test'). It includes fields for a description (optional), visibility (Public or Private), and a README initialization option. The 'Create repository' button is at the bottom.

Po utworzeniu repozytorium przejdziemy na stronę widoczną na rysunku 26.6. Menu główne będzie zawierało dodatkową opcję *Admin*. Opcja *Admin* umożliwia usuwanie repozytorium oraz zarządzanie uprawnieniami. Licencjonowanie kont *github.com* użależnia koszt konta od liczby prywatnych repozytoriów, nie nakładając limitu na liczbę osób uczestniczących w projekcie.

Obszar wskazany na rysunku 26.6 literą C zawiera publiczny adres repozytorium. Adres taki ma postać:

git@github.com:konto/repozytorium.git

np.:

git@github.com:gajdaw/test.git

Stosując powyższy adres, możemy sklonować repozytorium na dysk lokalny:

```
git clone git@github.com:gajdaw/test.git
```

Po wykonaniu operacji klonowania repozytorium lokalne na dysku będzie zawierało adres zdalny origin:

```
git@github.com:gajdaw/test.git
```

Gałąz lokalna master będzie śledziła gałąź master z repozytorium zdalnego. Rewizje, które wykonamy w repozytorium lokalnym, możemy przesyłać do repozytorium na serwerze *github.com* poleceniem:

```
git push
```

Ćwiczenie 28.1

Utwórz nowe repozytorium *cw-28-01* na swoim koncie na serwerze *github.com*. W repozytorium tym umieść trzy rewizje: a, b i c. Sprawdź stan repozytorium na serwerze, po czym usuń je.

ROZWIĄZANIE

Przyciskiem *New repository* utwórz nowe repozytorium o nazwie *cw-28-01*. Repozytorium to zainicjalizuj domyślnym plikiem *README*. Nowo utworzone repozytorium będzie miało adres:

```
git@github.com:konto/cw-28-01.git
```

Poleceniem:

```
git clone git@github.com:konto/cw-28-01.git
```

sklonuj repozytorium z serwera *github.com* na dysk lokalny.

W lokalnym repozytorium wykonaj trzy rewizje: a, b i c:

```
git simple-commits a b c
```

Rewizje z repozytorium lokalnego prześlij na serwer *github.com*:

```
git push
```

Powyższe polecenie zostanie poprawnie wykonane tylko wtedy, gdy zgodnie z opisem z rozdziału 27. skonfigurowałeś klucz SSH.

Na zakończenie ćwiczenia odwiedź stronę repozytorium:

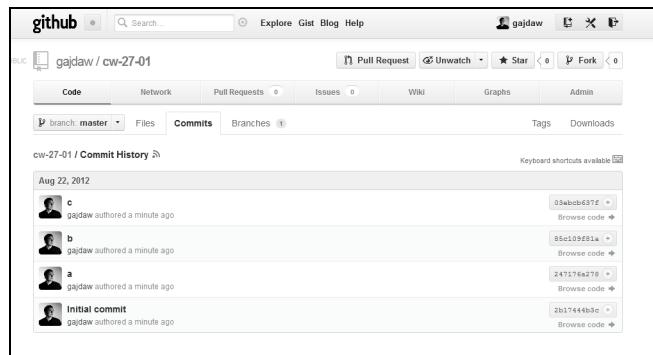
<https://github.com/konto/cw-28-01>

i wybierz opcję *Commits*. Ujrzyś rewizje przedstawione na rysunku 28.3.

Ćwiczenie zakończ, wybierając z menu głównego operację *Admin*. W dolnej części strony znajdziesz przycisk *Delete this repository*. Po naciśnięciu przycisku, wprowadzeniu nazwy oraz potwierdzeniu chęci wykonania operacji repozytorium zostanie usunięte.

Rysunek 28.3.

*Po wykonaniu operacji
git push rewizje a,
b i c będą zawarte
w repozytorium na
serwerze github.com*

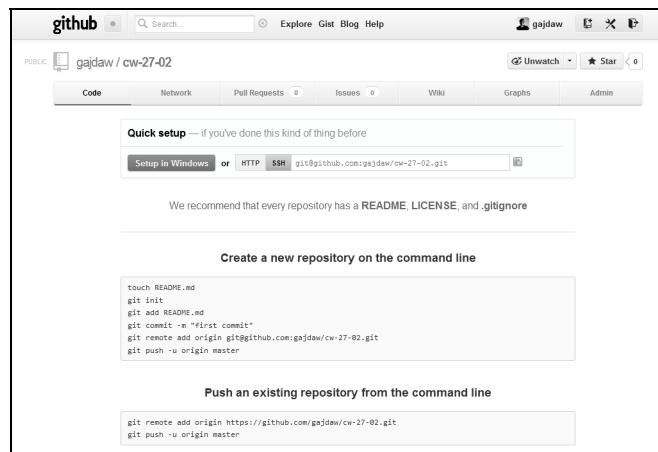


Import istniejącego kodu: serwis **github.com**

Procedura importowania kodu do nowego repozytorium jest nieco inna. Tworząc repozytorium, nie zaznaczamy opcji *Initialize this repository with a README*. W ten sposób otrzymamy puste repozytorium, które nie zawiera rewizji. Na stronie głównej repozytorium ujrzymy podpowiedzi widoczne na rysunku 28.4.

Rysunek 28.4.

*Strona nowego
pustego repozytorium
zawiera podpowiedzi*



W celu zimportowania kodu z dowolnego folderu na dysku lokalnym należy w folderze tym utworzyć nowe repozytorium zawierające wszystkie pliki, ustalić adres repozytorium zdalnego i przesłać rewizje na serwer. Służą do tego polecenia, które widać na rysunku 28.4:

```
git init
git add --all .
git commit -m "pierwsza rewizja"
git remote add origin git@github.com:konto/repozytorium.git
git push -u origin master
```

Jeśli na dysku lokalnym mamy przygotowane repozytorium, wystarczy wówczas wydać w nim dwie komendy:

```
git remote add origin git@github.com:konto/repozytorium.git  
git push -u origin master
```

Oczywiście tylko pierwsze polecenie `git push` wymaga parametru `-u`. Kolejne polecenia wydajemy w skróconej postaci:

```
git push
```

Ćwiczenie 28.2

Utwórz nowe repozytorium *cw-28-02* na swoim koncie na serwerze *github.com*. W repozytorium tym umieść trzy rewizje: *x*, *y* i *z*. Zadanie wykonaj w taki sposób, by repozytorium na serwerze nie zawierało żadnych innych rewizji.

ROZWIĄZANIE

Przyciskiem *New repository* utwórz nowe repozytorium o nazwie *cw-28-02*. W formularzu z rysunku 28.2 odznacz opcję inicjalizacji pliku *README*. Nowo utworzone repozytorium będzie miało adres:

```
git@github.com:konto/cw-28-02.git
```

W dowolnym folderze na dysku lokalnym wydaj polecenia:

```
git init  
git simple-commits x, y, z  
git remote add origin git@github.com:konto/repozytorium.git  
git push -u origin master
```

Inicjalizowanie nowego repozytorium: serwis bitbucket.org

W serwisie *bitbucket.org* nowe repozytorium inicjalizujemy w menu głównym opcją *Repositories/Create repository* lub przyciskiem wskazanym w dolnej części rysunku 28.5.

Rysunek 28.5.

Przyciski do tworzenia nowego repozytorium w serwisie *bitbucket.org*



Po przejściu do strony tworzenia nowego projektu ujrzymy formularz przedstawiony na rysunku 28.6.

Rysunek 28.6.
Formularz do tworzenia nowego pustego repozytorium w serwisie bitbucket.org

The screenshot shows the 'Create new repository' page on Bitbucket. At the top, there are two main buttons: 'Create new repository' (with a 'Start from scratch' link) and 'Import existing code' (with a 'Import your svn, hg or git repository online.' link). Below these are sections for 'Name (required)', 'Repository type' (radio buttons for Git and Mercurial), 'Project management' (checkboxes for Issue tracking and Wiki), and 'Language' (dropdown menu with 'Select language...'). There's also a 'Description' text area and a 'Website' URL input field. At the bottom is a large blue 'Create repository' button.

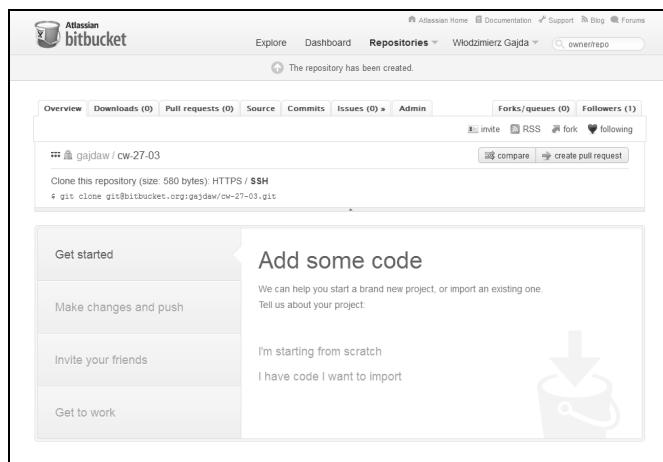
W formularzu należy wprowadzić nazwę repozytorium oraz ustalić opcje decydujące:

- ◆ czy repozytorium jest prywatne, czy publiczne;
- ◆ czy repozytorium jest prowadzone w systemie Git, czy Mercurial;
- ◆ czy w repozytorium ma być włączona obsługa śledzenia błędów (ang. *issue cracking*).

Po utworzeniu repozytorium przejdziemy na stronę prezentującą dwie podpowiedzi widoczne na rysunku 28.7:

- ◆ *I'm starting from scratch* (Rozpoczynam od nowa).
- ◆ *I have code I want to import* (Mam kod, który chciałbym zaimportować).

Rysunek 28.7.
Podpowiedzi ułatwiające rozpoczęwanie od nowa oraz importowanie kodu



W opcji *Rozpoczynam od nowa* znajdziemy podpowiedzi:

```
$ mkdir /path/to/your/project
$ cd /path/to/your/project
$ git init
$ git remote add origin ssh://git@bitbucket.org/konto/repozytorium.git
```

Podpowiedź wyświetlana po wybraniu opcji *Mam kod, który chciałbym zainportować* będzie natomiast:

```
$ cd /path/to/my/repo
$ git remote add origin ssh://git@bitbucket.org/konto/repozytorium.git
$ git push -u origin master # to push changes for the first time
```

Ćwiczenie 28.3

Utwórz nowe repozytorium *cw-28-03* na swoim koncie na serwerze *bitbucket.org*. W repozytorium tym umieść trzy rewizje: a, b i c. Sprawdź stan repozytorium na serwerze, po czym usuń je.

ROZWIĄZANIE

Wykorzystując opcję *Repositories/Create repository* utwórz nowe repozytorium o nazwie *cw-28-03*. Na dysku lokalnym w dowolnym folderze wydaj polecenia:

```
git init
git remote add origin ssh://git@bitbucket.org/konto/cw-28-03.git
git simple-commits a b c
git push -u origin master
```

Po tej operacji repozytorium na serwerze będzie wyglądało tak jak na rysunku 28.8.

Rysunek 28.8.
Repozytorium
z ćwiczenia 28.3
po przesłaniu na
serwer bitbucket.org

Author	Revision	Message	Date
Włodzimierz Gajda	ff6fc4ff3f67b	c	51 seconds ago
Włodzimierz Gajda	4bd7b093285bb	b	51 seconds ago
Włodzimierz Gajda	fe1b12b2b4ac8c	a	51 seconds ago

W celu usunięcia repozytorium wybierz opcję *Admin/Delete repository*.

Rozdział 29.

Praktyczne wykorzystanie Gita — scenariusz drugi

Oprogramowanie Git zwiększa komfort pracy także w projektach realizowanych jednoosobowo. Jeśli będziemy realizowali projekt jako prywatne repozytorium na serwerze github.com lub bitbucket.org, zyskamy wówczas:

- ◆ możliwość rozgałęziania pracy nad projektem,
- ◆ automatyczną kopię bezpieczeństwa,
- ◆ wbudowany system śledzenia błędów.

Ponadto jeśli zajdzie taka potrzeba, bez żadnej dodatkowej pracy będziemy mogli cały projekt udostępnić.

Nie bez znaczenia jest także fakt, że praca jest wykonywana z użyciem powszechnie stosowanych narzędzi. Pracując jednoosobowo, stosujemy ten sam zestaw narzędzi, który jest wykorzystywany w projektach grupowych. Gdy zajdzie konieczność (np. przy zmianie pracy), możemy bez konieczności doszkalania się przestawić się z pracy jednoosobowej w tryb pracy grupowej. Git wyznacza nowy standard w pracy grupowej i jest coraz częściej dołączany jako jedno z wymagań stawianych nowym pracownikom.

Dzięki znacznikom i poleceniu `git archive` w pojedynczym miejscu przechowujemy wszystkie wersje projektu. Archiwizacja kolejnych wersji projektu:

```
projekt-0.1.zip  
projekt-0.2.zip  
projekt-0.3.zip  
...
```

staje się zatem zbędna. Co więcej: dodatkowa archiwizacja projektu, np. na nośnikach CD-ROM lub DVD sprowadza się do zapisania na płycie pojedynczego folderu (tj. folderu projektu wraz z zawartym w nim folderem `.git`).

Repozytorium Gita jest jedynym miejscem wprowadzania poprawek w projekcie, a system śledzenia błędów jest jedynym miejscem do zarządzania błędami oraz udoskonaleniami.

Powyższe zalety powodują, że wszystkie realizowane prace (m.in. rękopisy książek, programy zajęć, witryny internetowe), nawet jeśli są to projekty jednoosobowe, realizuję z wykorzystaniem Gita.

Realizacja projektu jednoosobowego sprowadza się do wykonywania kroków opisanych w rozdziale 28. Przesłanie całego projektu na serwer w postaci kolejnej rewizji wymaga wydania jednego skróconego polecenia backup opisanego w rozdziale 18.:

```
git backup "..."
```

Komunikat podany jako parametr posłuży do oznaczenia rewizji.

Serwer *bitbucket.org* umożliwia bezpłatne tworzenie dowolnej liczby prywatnych repozytoriów, więc ten tryb pracy nie wiąże się z żadnymi dodatkowymi kosztami.



Opisany tryb pracy pierwszy raz próbowałem wdrożyć w 2007 roku, wykorzystując oprogramowanie SVN¹. Brak możliwości lokalnego wykonywania rewizji oraz nakład pracy związanej z konfiguracją każdego projektu znacznie komplikował jednak takie rozwiązanie. Lokalna praca w programie Git oraz darmowy hosting prywatnych repozytoriów na serwerze *bitbucket.org* rozwiązują wszelkie utrudnienia.

Osobnym aspektem wykorzystywania oprogramowania Git jest kontrola postępu prac. Jeśli pracownik jest zobowiązany do pracy w systemie Git z dodatkowym wymaganiem codziennego przesyłania rewizji na serwer, wówczas zleceniodawca, szczególnie w przypadku zadań długoterminowych, ma możliwość kontroli postępu prac.

Scenariusz pierwszy realizowany w serwisach **github.com i bitbucket.org**

Dzięki kontroli uprawnień scenariusz omówiony w rozdziale 19. możemy realizować, wykorzystując serwisy *github.com* oraz *bitbucket.org*. W tym celu wystarczy utworzyć nowe repozytorium prywatne oraz w panelu administracyjnym repozytorium dodać uprawnienia do zapisu uczestnikom projektu. Każdy uczestnik klonuje dane repozytorium na dysk lokalny. Modyfikacje wprowadzamy w repozytorium lokalnym, po czym wysyłamy je do repozytorium głównego. Schemat pracy będzie wyglądał dokładnie tak jak na rysunku 19.1.

¹ Por. <http://gajdaw.pl/varia/subversion-system-kontroli-wersji-tutorial/>.

Rozdział 30.

Praca grupowa w serwisach **github.com** oraz **bitbucket.org**

Praca grupowa w serwisach *github.com* oraz *bitbucket.org* może być zorganizowana na dwa sposoby.

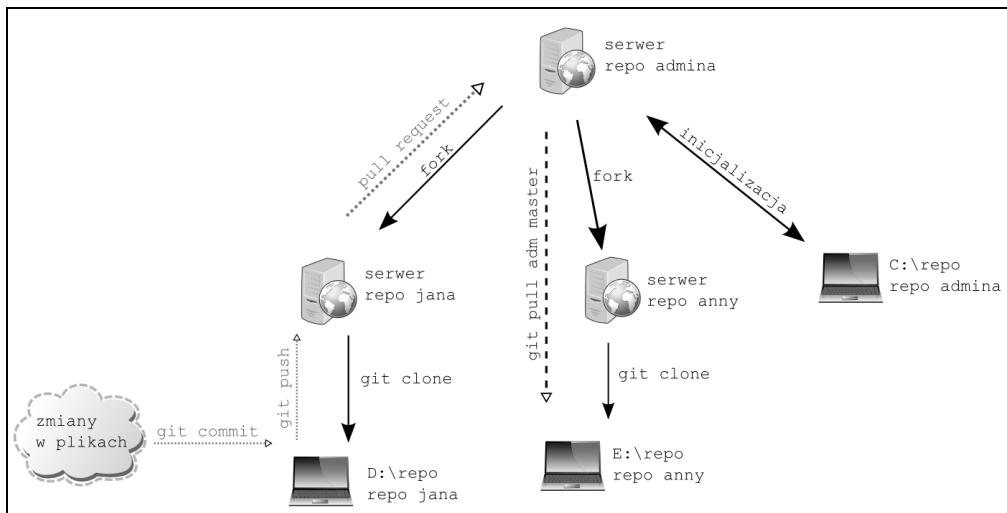
Pierwsze rozwiązanie polega na współdzieleniu repozytorium przez wielu użytkowników. Rozwiązanie takie jest opisane w punkcie „Scenariusz pierwszy realizowany w serwisach *github.com* i *bitbucket.org*” rozdziału 29. W panelu administracyjnym repozytorium ustalamy listę osób, które mają prawo zapisywania zmian w repozytorium. Każdy użytkownik klonuje repozytorium główne na dysk lokalny. Zmiany wykonujemy w repozytorium lokalnym, po czym wysyłamy je na serwer poleceniami *git push*.

Druga technika pracy wykorzystuje **żądanie aktualizacji** (ang. *pull request*). Każdy uczestnik projektu tworzy dwie kopie repozytorium głównego. Najpierw w interfejsie WWW wykonujemy operację **rozgałęziania** (ang. *fork*). W wyniku tej operacji otrzymujemy na serwerze własne repozytorium, które jest klonem repozytorium głównego. Repozytorium to klonujemy na dysk lokalny. Zmiany w repozytorium głównym wykonujemy dwuetapowo. Najpierw modyfikujemy repozytorium na dysku lokalnym. Wykonane zmiany przesyłamy na własne repozytorium odgałęzione. Następnie w repozytorium odgałęzionym wykonujemy operację żądania aktualizacji, która polega na zgłoszeniu w repozytorium głównym części scalenia naszych zmian. Administrator repozytorium głównego analizuje żądanie aktualizacji i jeśli stwierdza, że jest ono odpowiednie, akceptuje je.

Ten tryb pracy umożliwia administratorowi repozytorium głównego akceptację wybranych żądań aktualizacji, dlatego takie rozwiązanie jest bardzo powszechne w projektach o otwartym kodzie.

Praca oparta na żądaniach aktualizacji

Technika pracy oparta na żądaniach aktualizacji jest przedstawiona na rysunku 30.1.



Rysunek 30.1. Technika pracy oparta na żądaniach aktualizacji

W projekcie przedstawionym na rysunku 30.1 uczestniczą trzy osoby:

- ◆ admin,
- ◆ jan,
- ◆ anna.

Zwrót uwagi, że każdy uczestnik projektu pracuje w dwóch repozytoriach. Jedno z nich jest przechowywane na serwerze, a drugie — na dysku lokalnym danego użytkownika. Ponadto jedno z repozytoriów użytkownika admin odgrywa rolę repozytorium głównego, które jest wykorzystywane przez wszystkich do synchronizacji prac.

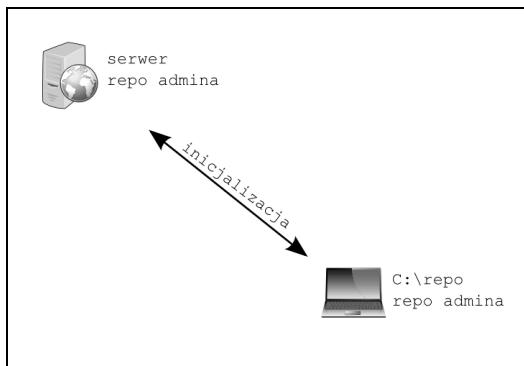
KROK 1.

Projekt rozpoczyna administrator techniką opisaną w rozdziale 28. Tworzy on własne repozytorium lokalne, umieszcza w nim dowolny kod, po czym publikuje je na serwerze. Ten etap pracy jest przedstawiony na rysunku 30.2.

Repozytorium administratora dostępne na serwerze będzie służyło wszystkim uczestnikom projektu do synchronizacji.

Rysunek 30.2.

Administrator tworzy własne repozytorium lokalne i publikuje je na serwerze

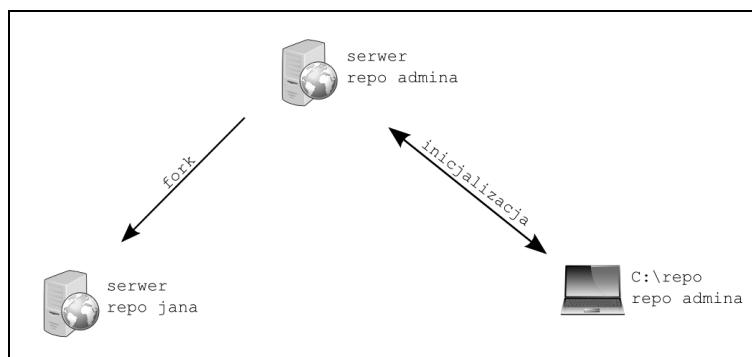
**KROK 2.**

Do projektu dołącza użytkownik jan. Najpierw administrator w panelu administracyjnym na serwerze nadaje użytkownikowi jan dostęp do repozytorium w trybie do odczytu (ang. *read access*).

Następnie użytkownik jan wykonuje w interfejsie WWW repozytorium administratora operację rozgałęziania (ang. *fork*). W ten sposób na serwerze powstaje repozytorium, którego właścicielem jest jan. Stan projektu jest widoczny na rysunku 30.3.

Rysunek 30.3.

Użytkownik Jan wykonał operację fork na repozytorium administratora

**KROK 3.**

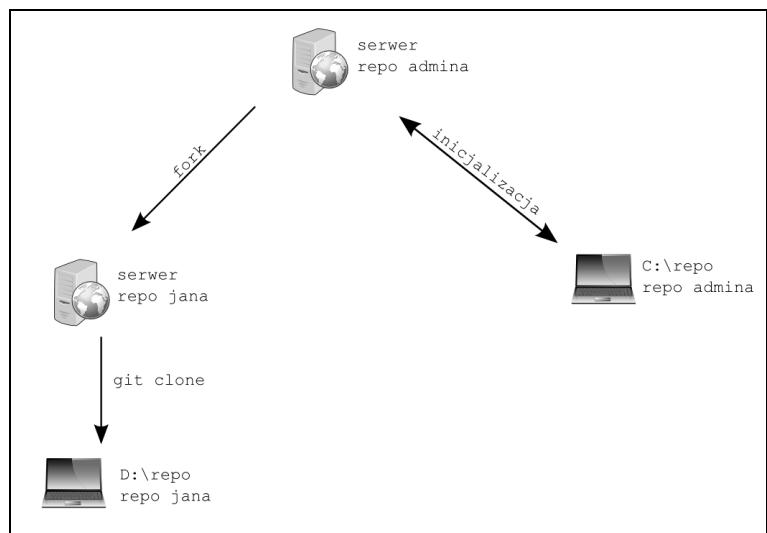
Następnie jan klonuje swoje repozytorium. W wierszu poleceń wykonuje komendę:

```
git clone ...
```

podając adres repozytorium oznaczonego na rysunku 30.3 jako serwer repo jana.

W ten sposób powstaje widoczne na rysunku 30.4 repozytorium oznaczone jako D:\repo repo jana.

Rysunek 30.4.
Użytkownik jan sklonował swoje repozytorium

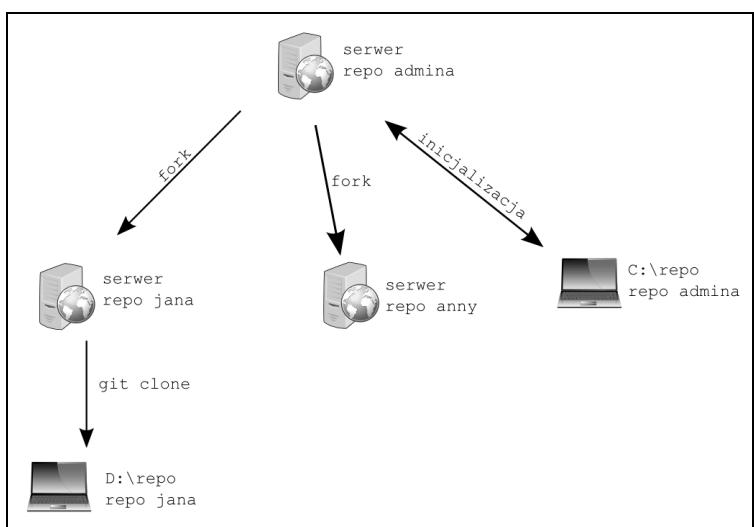


KROK 4.

Do projektu dołącza użytkownik anna. Administrator w panelu administracyjnym swojego repozytorium na serwerze nadaje użytkownikowi anna dostęp w trybie do odczytu. Użytkownik anna wykonuje operację rozgałęzienia projektu (ang. *fork*).

Na serwerze powstaje repozytorium, którego właścicielem jest anna. Stan projektu jest widoczny na rysunku 30.5.

Rysunek 30.5.
Anna wykonała rozgałęzienie repozytorium administratora



KROK 5.

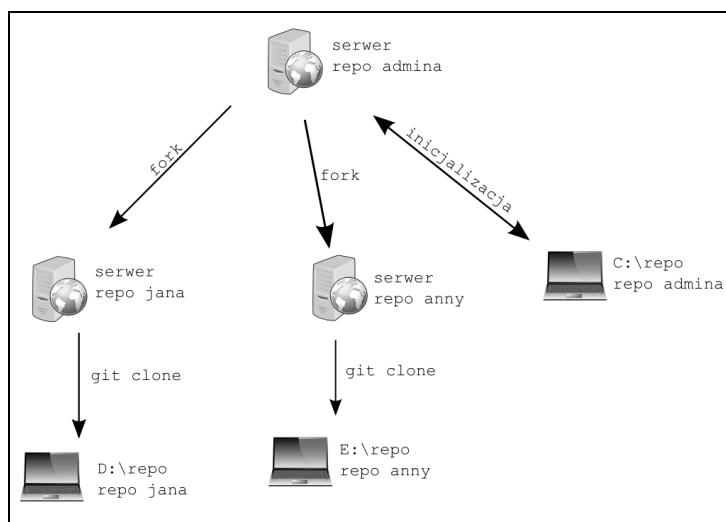
Następnie Anna klonuje swoje repozytorium z serwera na dysk lokalny. Wykonuje to w wierszu poleceń:

```
git clone ...
```

podając adres repozytorium oznaczonego na rysunku 30.5 jako serwer repo anny. Przebieg operacji klonowania jest zilustrowany na rysunku 30.6.

Rysunek 30.6.

Anna klonuje swoje repozytorium



W projekcie mamy teraz sześć repozytoriów:

- ♦ trzy repozytoria lokalne (administratora, Jana i Anny),
- ♦ trzy repozytoria na serwerze (administratora, Jana i Anny).

Każdy użytkownik ma pełne uprawnienia do własnych repozytoriów.

Wszystkie repozytoria są zsynchronizowane (zawierają dokładnie te same rewizje co repozytoria administratora).

Repozytorium zdalne administratora będzie głównym repozytorium projektu.

Do tego repozytorium wszyscy użytkownicy będą synchronizowali stan swojego repozytorium lokalnego.

KROK 6.

Przejdzmy do wprowadzania zmian w projekcie. Użytkownik Jan modyfikuje pliki projektu i wykonuje własną rewizję w repozytorium lokalnym:

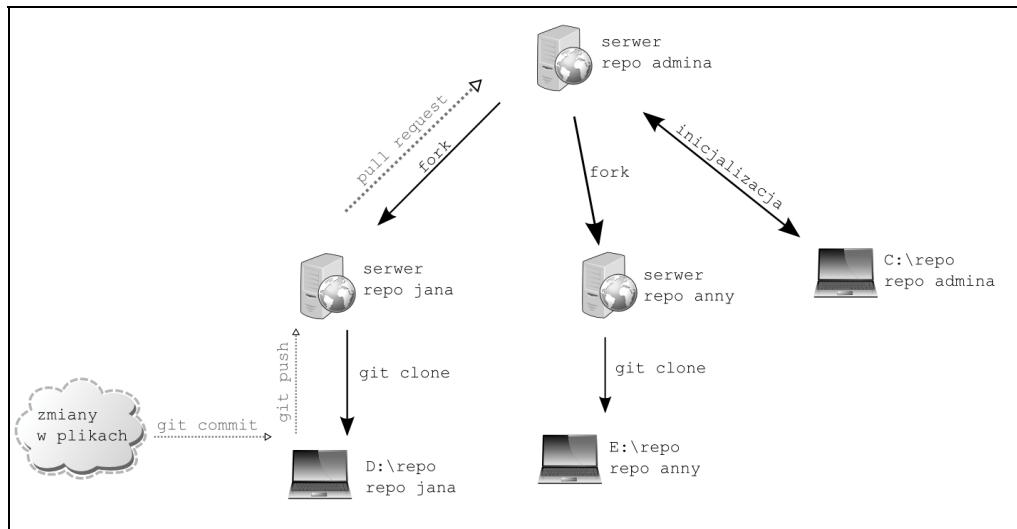
```
git commit ...
```

Następnie Jan wysyła rewizje z repozytorium do własnego repozytorium na serwerze:

```
git push
```

Trzecim etapem jest zgłoszenie żądania aktualizacji. W interfejsie WWW własnego repozytorium na serwerze Jan wykonuje operację *pull request*.

Przebieg opisanej operacji ilustruje rysunek 30.7.



Rysunek 30.7. Jan wykonuje rewizje, wysyła je do swojego repozytorium na serwerze i zgłasza żądanie aktualizacji

KROK 7.

Administrator w interfejsie WWW repozytorium głównego przegląda poprawki zawarte w żądaniu aktualizacji wysłanym przez Jana. Jeśli stwierdzi, że są dobre — zatwierdza je. Jeśli stwierdzi, że są złe — odrzuca.

Jeśli administrator zaakceptuje poprawki, w repozytorium głównym pojawią się wówczas rewizje wykonane przez Jana w kroku 6.

KROK 8.

Projekt Anny nie jest zsynchronizowany z repozytorium głównym. Anna na swoim dysku lokalnym nie ma rewizji wykonanych przez Jana w kroku 6.

W celu synchronizacji Anna dodaje w swoim lokalnym repozytorium adres zdalny repozytorium głównego (tj. repozytorium admina, które jest na serwerze):

```
git remote add adm ...
```

Następnie pobiera zawartość repozytorium z adresu adm do swojego repozytorium:

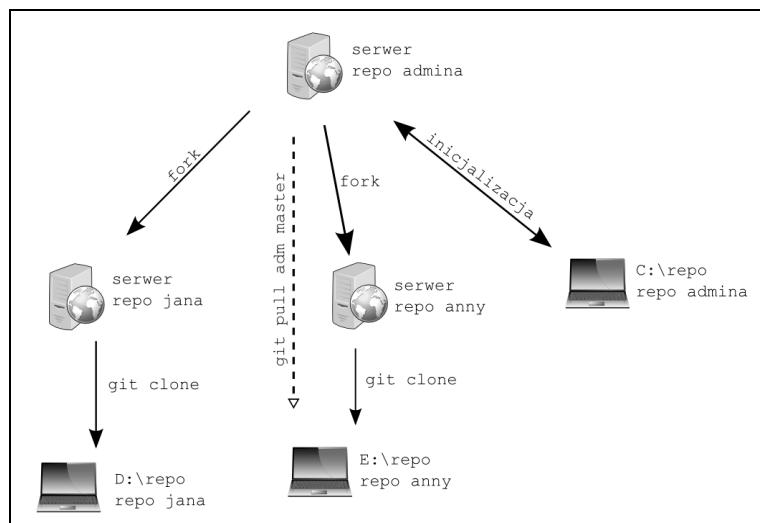
```
git pull adm master
```

W ten sposób repozytorium Anny jest zsynchronizowane z repozytorium głównym.

Przebieg operacji ilustruje rysunek 30.8.

Rysunek 30.8.

Anna synchronizuje swoje repozytorium z repozytorium głównym



Teraz repozytorium zdalne Anny (tj. repozytorium serwer repo anny) nie jest zsynchronizowane z resztą projektu. Synchronizacja repozytorium zdalnego Anny będzie wykonana, gdy Anna zacznie wprowadzać poprawki w projekcie. Jeśli Anna wykona operację:

```
git push
```

jej archiwum zdalne zostanie wówczas zsynchronizowane z jej archiwum lokalnym.

KROK 9.

Procedura uaktualniania plików projektu przez Annę jest przedstawiona na rysunku 30.9.

Przebiega ona identycznie jak procedura uaktualniania wykonana w kroku 6. przez Jana. Najpierw Anna wykonuje we własnym repozytorium lokalnym rewizje:

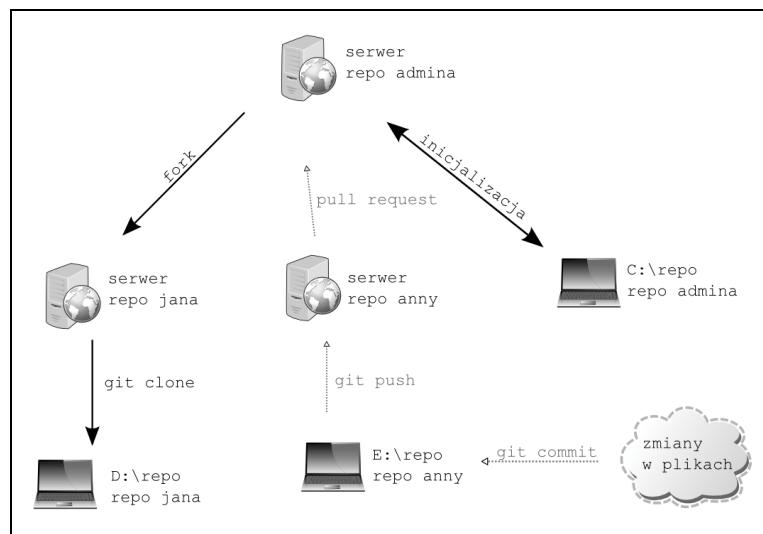
```
git commit ...
```

następnie przesyła rewizje do własnego repozytorium na serwerze:

```
git push
```

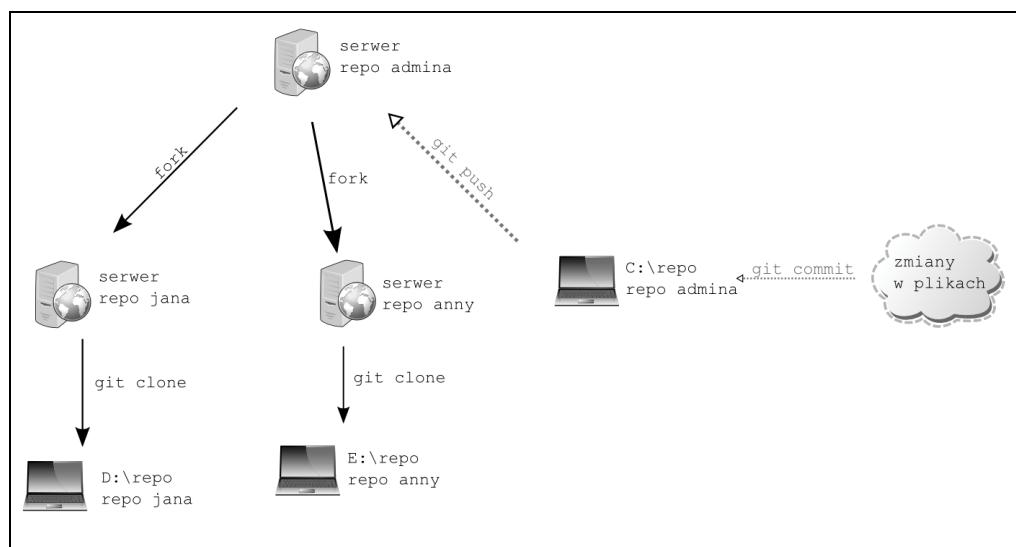
po czym zgłasza żądanie aktualizacji.

Rysunek 30.9.
Uaktualnianie plików projektu przez Annę



KROK 10.

Procedura uaktualniania plików projektu przez administratora przebiega z pominięciem żądań aktualizacji. Przebieg operacji ilustruje rysunek 30.10.



Rysunek 30.10. Uaktualnianie plików projektu przez administratora

Praca grupowa wykorzystująca żądania aktualizacji (bez gałęzi) w pigułce

Synchronizacja

Lokalny projekt synchronizujemy z projektem głównym, wykonując operacje:

```
git remote add glowny ADRES-GLOWNEGO-REPO  
git pull glowny master
```

Oczywiście komendę `git remote add` wykonujemy jeden raz. Za drugim i kolejnymi razami wydajemy tylko komendę:

```
git pull glowny master
```

Uaktualnienie lokalnego repozytorium (na dysku C) odbywa się z pominięciem własnego repozytorium wykonanego operacją *fork*.

Wprowadzanie poprawek w projekcie grupowym

Poprawki wprowadzamy jako rewizje w lokalnym repozytorium. Wykonane zmiany wysyłamy do własnego repozytorium powstałego po operacji *fork*:

```
git push
```

Następnie w interfejsie WWW własnego repozytorium zdalnego wysyłamy żądanie aktualizacji (ang. *pull request*) do repozytorium głównego.

Klonowanie repozytorium zdalnego

Na rysunku 30.1 repozytorium:

```
D:\repo (repo Jana)
```

powstało przez sklonowanie repozytorium:

```
Serwer: repo Jana
```

W repozytorium lokalnym Jana adres `origin` odnosi się wtedy do repozytorium serwer `repo Jana`. Polecienniem `git remote add` dodaliśmy w repozytorium lokalnym Jana drugi adres zdalny wskazujący repozytorium główne.

W ten sposób w repozytorium lokalnym Jana `D:\repo` mamy adresy:

- ◆ origin: wskazuje repozytorium serwer: repo Jana.
- ◆ adm: wskazuje repozytorium główne serwer repo admina.

Do aktualizacji będziemy wówczas stosowali komendy:

```
//przesłanie rewizji z repozytorium D:\repo  
//do repozytorium Serwer: repo jana  
git push  
  
//pobranie rewizji z repozytorium głównego  
//do repozytorium D:\repo  
git pull adm master
```

Możemy postąpić inaczej. Repozytorium lokalne możemy utworzyć przez sklonowanie repozytorium głównego. Następnie w sklonowanym repozytorium dodajemy adres zdalny o nazwie `my` wskazujący repozytorium zdalne Jana. W repozytorium lokalnym Jana wystąpią wówczas adresy:

- ◆ origin: wskazuje repozytorium główne serwer repo admina.
- ◆ `my`: wskazuje repozytorium Serwer: repo Jana.

W takim przypadku aktualizację będziemy przeprowadzali komendami:

```
//przesłanie rewizji z repozytorium D:\repo  
//do repozytorium Serwer: repo jana  
git push my master  
  
//pobranie rewizji z repozytorium głównego  
//do repozytorium D:\repo  
git pull
```

Takie rozwiązanie jest o tyle wygodniejsze, że wprowadza ujednolicony sposób nazwania repozytoriów zdalnych. W każdym projekcie adres `origin` oznacza repozytorium główne, a adres `my` — repozytorium prywatne użytkownika. Bez względu na to, w którym projekcie pracujesz, do pobrania najnowszych rewizji z repozytorium głównego użyjesz komendy:

```
git pull
```

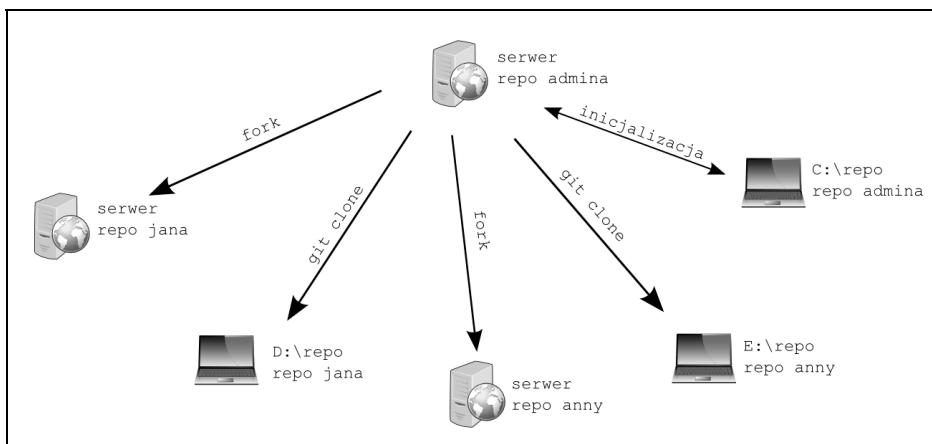
Do przesyłania własnych zmian do własnego repozytorium zdalnego posłuży natomiast komenda:

```
git push my master
```

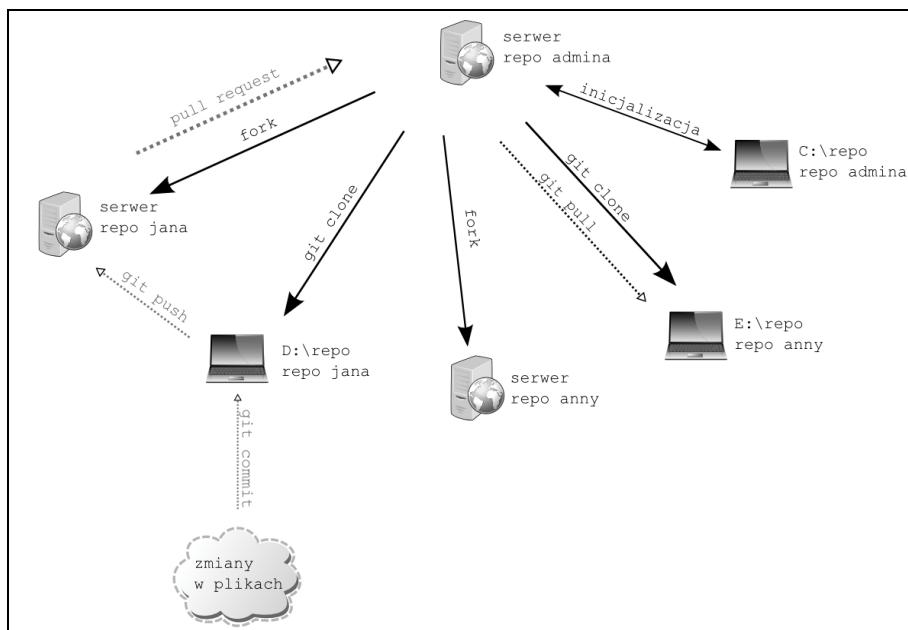
Drugi sposób inicjalizacji pracy grupowej jest przedstawiony na rysunku 30.11.

Procedura aktualizacji repozytoriów z rysunku 30.11 będzie przebiegała zgodnie ze schematem z rysunku 30.12.

Oczywiście schematy przedstawione na rysunkach 30.1 oraz 30.11 stwarzają dokładnie te same możliwości pracy nad projektem. Różnią się one bowiem wyłącznie wpisami ustalającymi adresy repozytoriów zdalnych oraz ustaleniami dotyczącymi śledzenia gałęzi. Schemat 30.1 możemy przekształcić w schemat 30.12 odpowiednimi poleceńami ustalającymi adresy zdalne i gałęzie śledzone.



Rysunek 30.11. Druga metoda inicjalizacji projektu grupowego



Rysunek 30.12. Procedura aktualizacji repozytoriów z rysunku 30.11

Żądania aktualizacji i gałęzie

Żądania aktualizacji dotyczą konkretnych gałęzi. Najwygodniejszą metodą pracy jest wykonywanie zmian w gałęziach i zgłoszanie żądań aktualizacji dotyczących modyfikowanej gałęzi. Odrzucenie żądania aktualizacji nie powoduje wówczas żadnych komplikacji.

Ćwiczenie 30.1

W projekcie:

<https://github.com/git-podrecznik/wiersze>

dodaj dowolny wiersz polskiego poety. Zastosuj schemat pracy z rysunku 30.1 oraz żądania aktualizacji wykorzystujące gałęzie. Ćwiczenie wykonaj, wykorzystując serwis *github.com*.

ROZWIĄZANIE

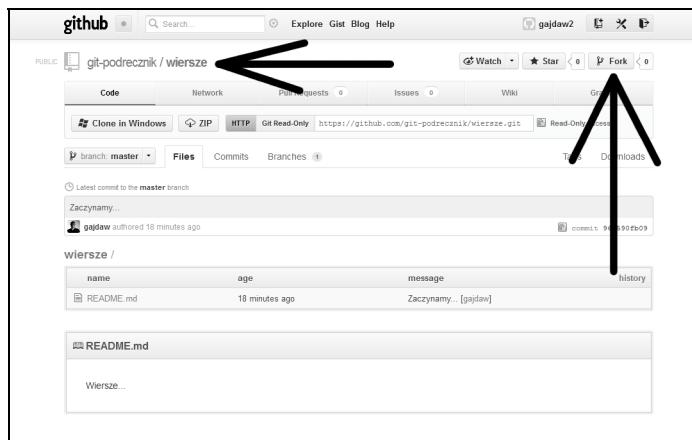
Zaloguj się na swoje konto w serwisie *github.com*, odwiedź stronę repozytorium:

<https://github.com/git-podrecznik/wiersze>

i naciśnij zaznaczony na rysunku 30.13 przycisk *fork*.

Rysunek 30.13.

Tworzenie własnej kopii repozytorium *wiersze*



Po wykonaniu operacji z rysunku 30.13 przejdziesz do strony widocznej na rysunku 30.14. W miejscu wskazanym strzałką ujrzysz adres własnego repozytorium, np.:

git@github.com:twoje-konto/wiersze.git

Następnie sklonuj repozytorium z rysunku 30.14 na dysk twardy. W tym celu w dowolnym folderze wydaj komendę:

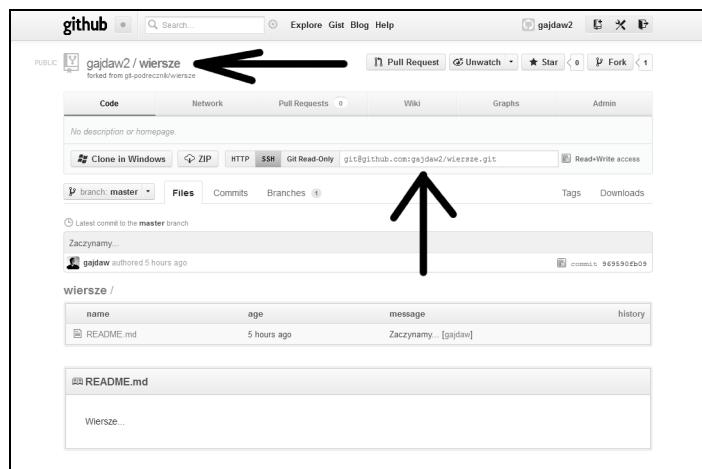
```
git clone git@github.com:nazwa-konta/wiersze.git
```

W repozytorium lokalnym utwórz nową gałąź o dowolnej nazwie¹, np. *c_k_n*:

```
git checkout -b c_k_n
```

¹ Nazwa gałęzi *c_k_n* to skrótowiec od nazwiska Cypriana Kamila Norwida. Oczywiście wykonując ćwiczenie, wprowadź wiersz, którego jeszcze nie ma w repozytorium. Jeśli będzie to wiersz innego poety, np. Adama Mickiewicza, to użyj innej nazwy gałęzi, np. *a_m*.

Rysunek 30.14.
Repozytorium otrzymane po wykonaniu operacji fork



Następnie w gałęzi `c_k_n` dodaj plik tekstowy z nowym wierszem. Wprowadzoną zmianę zatwierdź rewizją.

W celu wysłania gałęzi lokalnej `c_k_n` do swojego repozytorium *wiersze* wydaj komendę:

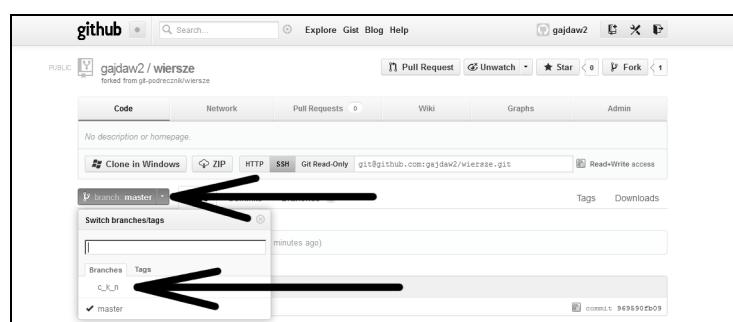
```
git push origin c_k_n
```

Po wykonaniu powyższego polecenia ujrzysz informację o tym, że na serwerze utworzona została nowa gałąź `c_k_n`:

```
* [new branch]      HEAD -> c_k_n
```

Przesłaną gałąź znajdziesz w interfejsie WWW, naciskając wskazany na rysunku 30.15 przycisk *branch: master*.

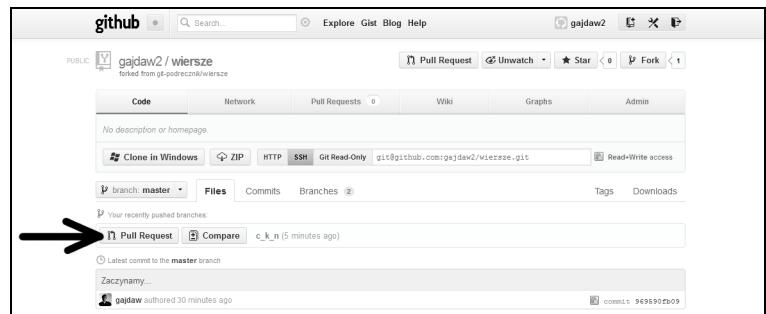
Rysunek 30.15.
Wyszukiwanie gałęzi `c_k_n` przesłanej na serwer [github.com](#)



Jeśli przełączysz gałąź na `c_k_n`, będziesz mógł sprawdzić, czy wiersz został poprawnie dodany do plików zawartych w repozytorium.

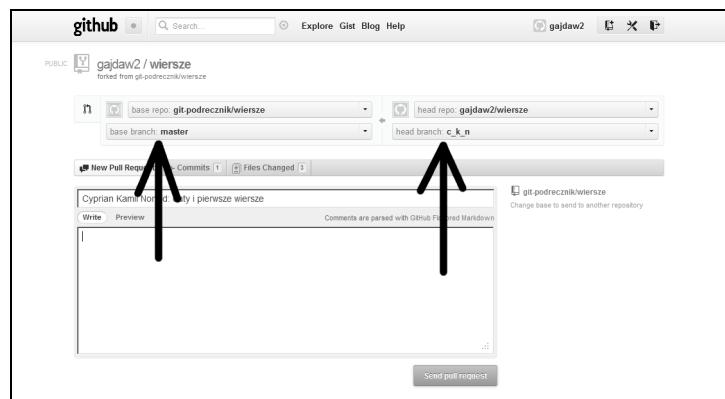
W celu przesłania wprowadzonych zmian do repozytorium głównego przejdź na stronę główną swojego repozytorium i odszukaj widoczny na rysunku 30.16 przycisk *Pull request*.

Rysunek 30.16.
Przycisk *Pull Request*,
który umożliwia
zgłoszenie żądania
aktualizacji



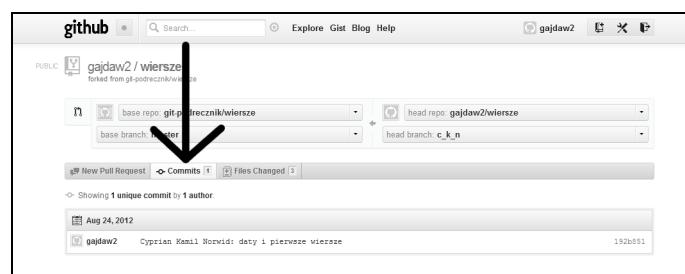
Po naciśnięciu przycisku z rysunku 30.16 przejdziesz do formularza przedstawionego na rysunku 30.17. W formularzu tym wybrać można gałąź, która będzie przesyłana (*head branch*), oraz gałąź repozytorium głównego, do której trafić mają zmiany (*base branch*). Domyślnie będą to gałąź przesłana poleceniem git push origin c_k_n oraz gałąź master.

Rysunek 30.17.
Formularz ustalający
parametry żądania
aktualizacji



Wskazana na rysunku 30.18 zakładka *Commits* pozwala sprawdzić rewizje zawarte w zgłoszeniu aktualizacji.

Rysunek 30.18.
Zakładka *Commits*
pokazuje rewizje
zawarte w zgłoszeniu
aktualizacji



Widoczna na rysunku 30.19 zakładka *Files Changed* pozwala natomiast szczegółowo sprawdzić, jakie zmiany zostały wprowadzone w plikach.

Rysunek 30.19.

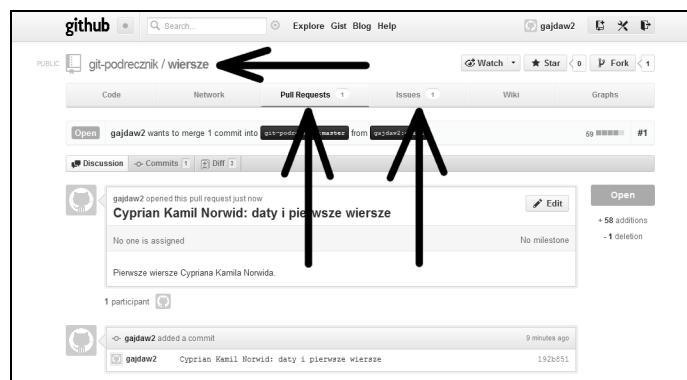
Zakładka *Files*
Changed pokazuje zmiany wprowadzone w plikach



W formularzu² z rysunku 30.17 wprowadź krótki opis wykonanych aktualizacji i na- ciśnij przycisk *Send pull request*. Zostaniesz przeniesiony do repozytorium głównego, tj. tego, w którym wykonałeś operację *fork*. Na stronie repozytorium głównego w zakładkach *Pull Requests* oraz *Issues* znajdziesz przesłane przez siebie żądanie aktualizacji. Odszukiwanie zgłoszonych żądań aktualizacji ilustruje rysunek 30.20.

Rysunek 30.20.

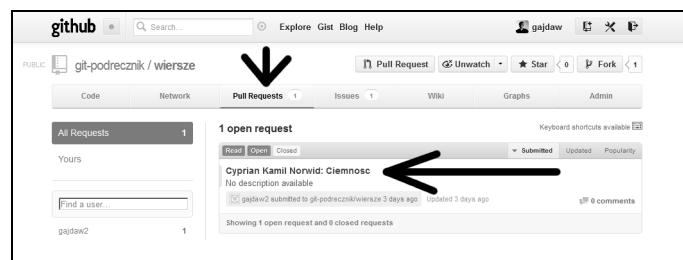
Przesłane żądanie aktualizacji pojawia się w interfejsie WWW repozytorium, w którym wykonałeś operację *fork*



Akceptację zgłoszenia aktualizacji wykonuje administrator repozytorium głównego. Zgodnie z rysunkiem 30.21 administrator wybiera opcję *Pull Requests*, a następnie przechodzi do strony *Pull Requests*.

Rysunek 30.21.

W celu zaakceptowania żądania aktualizacji administrator przechodzi do strony *Pull Requests*

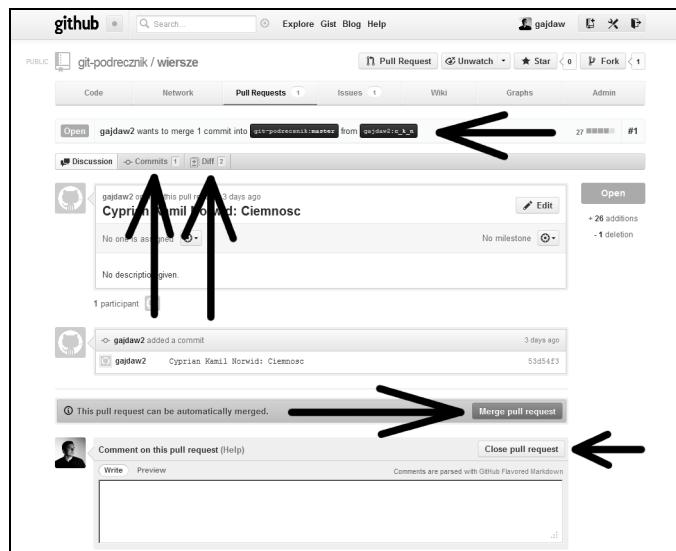


² Formularz ten jest dostępny w zakładce *Discussion*.

Strona ze szczegółowymi informacjami o żądaniu aktualizacji jest przedstawiona na rysunku 30.22. Możemy na niej sprawdzić, jakich gałęzi dotyczy, jakie zawiera rewizje oraz zmiany. Do akceptacji żądania służy przycisk *Merge pull request*, a do odrzucenia — *Close pull request*.

Rysunek 30.22.

Strona ze szczegółowymi informacjami o żądaniu aktualizacji



Po zaakceptowaniu żądania rewizje zostaną dołączone do odpowiedniej gałęzi repozytorium głównego. Stan repozytorium po akceptacji żądania z rysunku 30.22 jest widoczny na rysunku 30.23.

Rysunek 30.23.

Po zaakceptowaniu żądania aktualizacji rewizji pojawiają się w odpowiedniej gałęzi repozytorium głównego



Po wykonaniu opisanej operacji Twoje repozytorium lokalne nie będzie zsynchronizowane z repozytorium głównym. W gałęzi master Twojego repozytorium brakowało będzie dwóch ostatnich rewizji z rysunku 30.23. W celu aktualizacji repozytorium lokalnego wydaj komendy:

```
git checkout master
git remote add git-podrecznik https://github.com/git-podrecznik/wiersze.git
git pull git-podrecznik master
```

Po wydaniu powyższych poleceń gałąź master repozytorium lokalnego będzie zawierała te same rewizje co gałąź master repozytorium głównego. Gałąź master Twojego repozytorium zdalnego nie będzie jednak zawierała dwóch ostatnich rewizji. Przekonasz się o tym, wydając polecenie:

```
git status -sb  
## master...origin/master [ahead 2]
```

W celu uaktualnienia własnego repozytorium zdalnego wydaj komendę:

```
git push
```

Teraz gałęzie master trzech repozytoriów: lokalnego, Twojego repozytorium zdalnego oraz repozytorium głównego są zsynchronizowane. Gałęzie c_k_n w repozytorium lokalnym oraz w Twoim repozytorium na serwerze są zatem zbędne. Przekonasz się o tym, wyając polecenie:

```
//lista gałęzi lokalnych,  
//które są zawarte w gałęzi master  
git branch --merged
```

```
//lista gałęzi zdalnych,  
//które są zawarte w gałęzi master  
git branch -r --merged
```

W celu usunięcia lokalnej gałęzi c_k_n wydaj komendę:

```
git branch -d c_k_n
```

Gałąź zdalną c_k_n usuniesz poleciem:

```
git push origin :c_k_n
```

Ćwiczenie 30.4

W projekcie:

<https://bitbucket.org/gajdaw/pisarze>

dodaj informacje o dowolnym pisarzu. Zastosuj schemat pracy z rysunków 30.11 i 30.12 oraz zgłoszenia aktualizacji wykorzystujące gałęzie. Użyj własnego konta w serwisie bitbucket.org.

ROZWIĄZANIE

Zaloguj się na swoje konto w serwisie bitbucket.org, odwiedź stronę repozytorium:

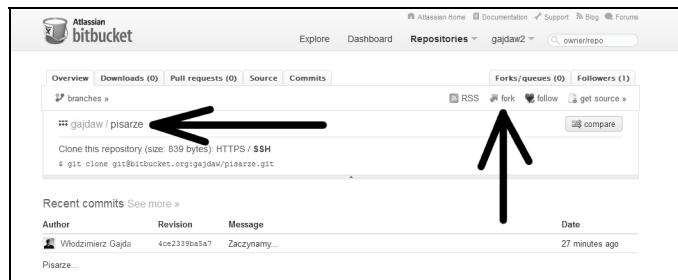
<https://bitbucket.org/gajdaw/pisarze>

i naciśnij zaznaczony na rysunku 30.24 przycisk *fork*. W ten sposób utworzysz własną kopię repozytorium *gajdaw/pisarze*.

Po wykonaniu operacji *fork* zostaniesz przekierowany na stronę główną utworzonego repozytorium. Jeśli pracujesz na koncie nazwa-konta, to utworzone repozytorium będzie miało adres:

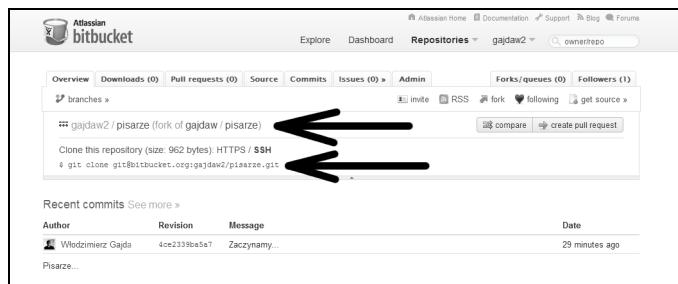
```
git@bitbucket.org:nazwa-konta/pisarze.git
```

Rysunek 30.24.
Operacja fork tworzy kopię aktualnie odwiedzanego repozytorium



Adres utworzonego repozytorium możesz zawsze sprawdzić w oknie właściwości repozytorium. Rysunek 30.25 prezentuje informacje o nowo utworzonym repozytorium.

Rysunek 30.25.
Informacje o nowo utworzonym repozytorium



Ponieważ ćwiczenie mamy wykonać zgodnie ze schematem z rysunku 30.11, w celu utworzenia repozytorium lokalnego klonujemy repozytorium główne. W wierszu poleceń wydajemy komendę:

```
git clone git@bitbucket.org:gajdaw/pisarze.git
```

Następnie w repozytorium lokalnym tworzymy nową gałąź³:

```
git checkout -b stanislaw_lem
```

i dodajemy w niej rewizje. Zawartość nowej gałęzi chcemy przesłać do własnego repozytorium na serwerze. Najpierw dodajemy adres własnego repozytorium zdalnego:

```
git remote add my git@bitbucket.org:nazwa-konta/pisarze.git
```

po czym wydajemy komendę:

```
git push my stanislaw_lem
```

Komunikat:

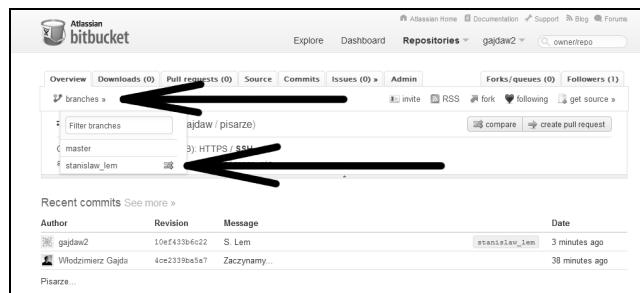
```
* [new branch]      stanislaw_lem -> stanislaw_lem
```

informuje, że w repozytorium zdalnym utworzona została nowa gałąź.

Fakt ten weryfikujemy w interfejsie WWW serwisu *bitbucket.org*. Procedura badania gałęzi w repozytorium na serwerze *bitbucket.org* jest przedstawiona na rysunku 30.26.

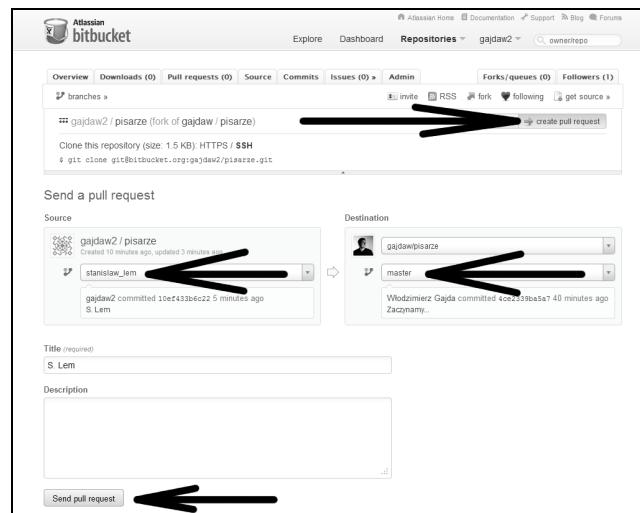
³ Oczywiście wykonując ćwiczenie, wprowadź dane dotyczące innego pisarza, np. Henryka Sienkiewicza. Gałąź nazwij imieniem i nazwiskiem pisarza, np. henryk_sienkiewicz.

Rysunek 30.26.
Badanie gałęzi
w repozytorium
na serwerze
[bitbucket.org](#)



Po przesyłaniu gałęzi do własnego repozytorium zdalnego przystępujemy do zgłoszenia aktualizacji. Wybieramy operację *create pull request*, ustalamy gałąź źródłową stanislaw_lem oraz gałąź docelową master, po czym naciskamy przycisk *Send pull request*. Szczegółowe dane na temat rewizji i modyfikowanych plików są w serwisie [bitbucket.org](#) wyświetlane poniżej przycisku *Send pull request*. Procedura zgłaszania żądania aktualizacji w serwisie [bitbucket.org](#) jest przedstawiona na rysunku 30.27.

Rysunek 30.27.
Zgłaszanie żądania
aktualizacji w serwisie
[bitbucket.org](#)



Gdy formularz z rysunku 30.27 zatwierdzimy przyciskiem *Send pull request*, przygotowane zgłoszenie będzie dostępne w repozytorium głównym w zakładce *Pull requests*. Lista wszystkich zgłoszeń aktualizacji repozytorium głównego jest widoczna na rysunku 30.28.

Rysunek 30.28.
Lista zgłoszeń
aktualizacji
w repozytorium
głównym



Administrator repozytorium głównego po wybraniu z listy widocznej na rysunku 30.28 konkretnej pozycji ujrzy szczegółowe dane zgłoszenia aktualizacji oraz przyciski umożliwiające akceptację, odrzucenie i edycję. Formularz do zatwierdzania zgłoszenia aktualizacji jest przedstawiony na rysunku 30.29.

Rysunek 30.29.

Szczegółowe informacje o zgłoszeniu aktualizacji

The screenshot shows a Bitbucket repository page for 'gajdaw / pisarze'. A pull request from 'gajdaw2 / pisarze' to 'gajdaw / pisarze' has been created. The pull request details show an incoming commit from 'gajdaw2' with revision '10ef433b6c22' and message 'S. Lem'. The file 'stanislaw-lem.txt' is being modified. At the top right, there are 'Accept and merge' and 'Reject' buttons. In the center, there are 'Source' and 'Destination' sections. Arrows point to the 'Accept and merge' button, the source and destination repository names, the author and revision of the incoming commit, and the file being modified.

Jeśli administrator zatwierdzi zgłoszenie przyciskiem *Accept*, rewizje zawarte w zgłoszeniu zostaną wówczas scalone z gałęzią master repozytorium głównego. Po tej operacji na stronie domowej repozytorium głównego ujrzymy dodane rewizje. Zwróć uwagę, że przedstawiona na rysunku 30.30 lista rewizji jest przedstawiona w sposób graficzny, przypominający wydruk polecenia `git log --graph`.

Rysunek 30.30.

Po zaakceptowaniu zgłoszenia rewizje są widoczne na stronie domowej repozytorium głównego

The screenshot shows the same Bitbucket repository page after the pull request has been accepted. The commit history now includes the merged commit from 'gajdaw2' with revision '10ef433b6c22' and message 'Merged in gajdaw/pisarze/stanislaw_lem (pull request #1)'. The commit from 'Włodzimierz Gajda' is still present. An arrow points to the merged commit.

Gdy zgłoszenie zostało zaakceptowane, należy zsynchronizować własne dwa repozytoria. W celu synchronizacji repozytorium lokalnego przechodzimy na gałąź master:

```
git checkout master
```

i wydajemy komendę:

```
git pull
```

Własne repozytorium zdalne (powstałe po operacji *fork*) synchronizujemy natomiast komendą:

```
git push my master
```

Na zakończenie sprawdzamy, które gałęzie są zbędne:

```
git branch -a --merged
```

po czym usuwamy te zbędne gałęzie:

```
git branch -d stanislaw_lem  
git push my :stanislaw_lem
```

Opisy i dyskusje

Oba serwisy, *github.com* oraz *bitbucket.org*, umożliwiają szczegółowe opisywanie zgłoszeń aktualizacji oraz prowadzenie dyskusji dotyczących proponowanych zmian. W serwisie *github.com* wszystkie wypowiedzi możemy formatować, stosując rozszerzoną wersję języka MarkDown (tzw. GFM — *Github Flavored Markdown*). Serwis *bitbucket.org* pozwala na stosowanie języka MarkDown wyłącznie w plikach *README*.

W języku MarkDown do formatowania treści nie stosujemy znaczników, a pojedyncze znaki. Zestawienie znaczników języka MarkDown jest przedstawione na listingu 30.1.

Listing 30.1. Zestawienie podstawowych metod formatowania tekstu w języku MarkDown

```
# Nagłówek <h1>  
## Nagłówek <h2>  
### Nagłówek <h3>  
  
*Pochylenie tekstu *  
_Pochylenie tekstu _  
**Wytluszczenie tekstu **  
__Wytluszczenie tekstu __  
  
Wypunktowanie:  
* Pozycja 1  
* Pozycja 2  
* Pozycja 2a  
* Pozycja 2b  
  
Numerowanie:  
1. Pozycja 1  
2. Pozycja 2  
3. Pozycja 3  
* Pozycja 3a  
* Pozycja 3b
```



Szczegółowy opis języka MarkDown jest dostępny na stronie:

<http://daringfireball.net/projects/markdown/syntax>

Dokumentację dialekту GFM (*Github Flavored Markdown*) znajdziesz na stronie:

<http://github.github.com/github-flavored-markdown/>

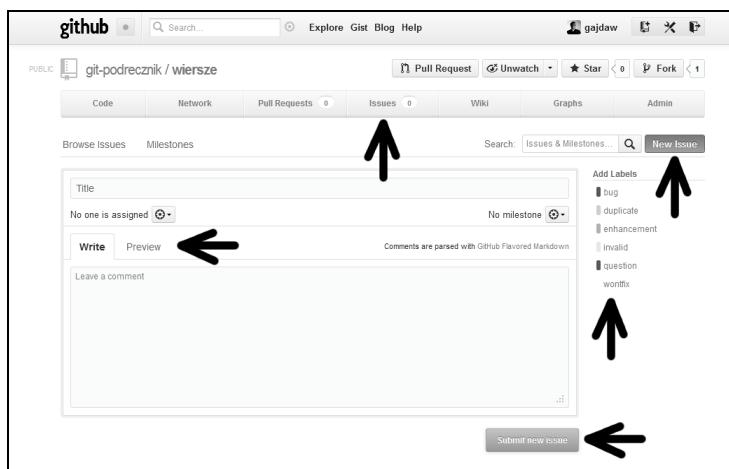
Rozdział 31.

Zintegrowany system śledzenia błędów

Ogromną zaletą serwisów *github.com* oraz *bitbucket.org* jest wbudowany system śledzenia błędów. Dzięki niemu w projekcie pojawia się centralny, zsynchronizowany rejestr zawierający wszystkie usterki i niedociągnięcia.

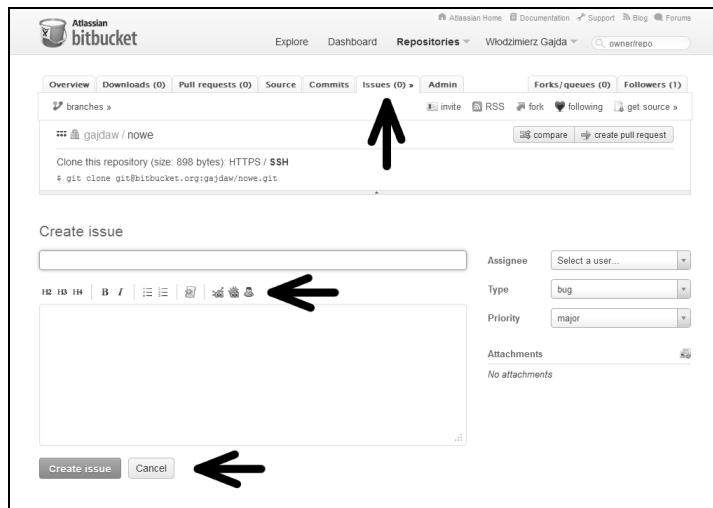
Wpisy w systemie śledzenia błędów w serwisie *github.com* dodajemy za pomocą formularza przedstawionego na rysunku 31.1.

Rysunek 31.1.
github.com
— formularz do tworzenia nowego wpisu w systemie śledzenia błędów



Do tworzenia wpisów w dzienniku błędów w serwisie *bitbucket.org* służy formularz przedstawiony na rysunku 31.2.

Rysunek 31.2.
bitbucket.org — formularz do tworzenia nowego wpisu w systemie śledzenia błędów



Ćwiczenie 31.1

Stosując system śledzenia poprawek w repozytorium:

<https://github.com/git-podrecznik/wiersze>

zaproponuj¹ nowego poetę lub nowy wiersz.

Ćwiczenie 31.2

Stosując system śledzenia poprawek w repozytorium:

<https://bitbucket.org/gajdaw/pisarze>

zaproponuj² nowego pisarza.

Ćwiczenie 31.3

Przejrzyj rejestr błędów projektu:

<https://github.com/git-podrecznik/wiersze>

Wprowadź zmiany³ zaproponowane w jednym ze zgłoszeń.

¹ Wykonanie ćwiczenia polega na utworzeniu nowego wpisu w rejestrze *Issues*.

² Wykonanie ćwiczenia polega na utworzeniu nowego wpisu w rejestrze *Issues*.

³ Wykonanie ćwiczenia polega na zgłoszeniu żądania aktualizacji dotyczącego konkretnego wpisu z rejestru błędów.

Ćwiczenie 31.4

Przejrzyj rejestr błędów projektu:

<https://bitbucket.org/gajdaw/pisarze>

Wprowadź zmiany⁴ zaproponowane w jednym ze zgłoszeń.

⁴ Wykonanie ćwiczenia polega na zgłoszeniu żądania aktualizacji dotyczącego konkretnego wpisu z rejestru błędów.

Rozdział 32.

Podsumowanie części V

Rozwiązywanie opisane w rozdziale 30. to chyba najprostsza i najwygodniejsza metoda wdrożenia systemu Git do realizacji małych i średnich projektów. Praca w oparciu o żądania aktualizacji daje gwarancję, że przypadkowe błędy jednego uczestnika nie wpłyną na innych użytkowników. Dodatkową zaletą jest zintegrowany system śledzenia błędów.

Oczywiście rozwiązanie oparte o żądania aktualizacji możemy łączyć ze scenariuszem pracy opisanym w rozdziale 19. Osoby o odpowiednich kompetencjach otrzymują konieczne zapewniające dostęp w trybie do zapisu. Dzięki temu mogą aktualizować zawartość repozytorium głównego z pominięciem systemu żądań aktualizacji.

Repozytoria do ćwiczenia znajomości Gita

Repozytoria wymienione w rozdziale 30., czyli:

<https://github.com/git-podrecznik/wiersze>

<https://bitbucket.org/gajdaw/pisarze>

służą wyłącznie do ćwiczenia umiejętności posługiwania się Gitem i serwisami *github.com* oraz *bitbucket.org*. W systemie tym bez żadnych ograniczeń oraz obaw można zgłaszać błędy, uaktualniać oraz żądać aktualizacji.

Dodatki

Dodatek A Literatura

Książki

- 1.** Scott Chacon, *Pro Git*, APress, 2009
http://git-scm.com/book/pl
http://www.apress.com/9781430218333
- 2.** Jon Loeliger, *Version Control With Git*, O'Reilly, 2009
http://shop.oreilly.com/product/9780596520137.do
- 3.** Eric Sink, *Version Control by Example*, Pyrenean Gold Press, 2011
http://www.ericsink.com/vcbe/html/index.html
- 4.** Travis Swicegood, *Pragmatic Version Control Using Git*, The Pragmatic Programmers, 2008
http://pragprog.com/book/tsgit/pragmatic-version-control-using-git
- 5.** Travis Swicegood, *Pragmatic Guide to Git*, The Pragmatic Programmers, 2010
http://pragprog.com/book/pg_git/pragmatic-guide-to-git

Materiały dostępne w Internecie

- 1.** Ben Lynn, *Git Magic*
<http://www-cs-students.stanford.edu/~blynn/gitmagic/>
- 2.** *The Git Community Book*
<http://alx.github.com/gitbook/>
- 3.** John Wiegley, *Git from the bottom up*, 2009
<http://ftp.newartisans.com/pub/git.from.bottom.up.pdf>
- 4.** Yan Pritzker, *Git Workflows*, 2011
<http://yanpritzker.com/git-book/>
- 5.** Tommi Virtanen, *Git for Computer Scientists*
<http://eagain.net/articles/git-for-computer-scientists/>
- 6.** William Durand, *My git branching model*
<http://williamdurand.fr/2012/01/17/my-git-branching-model/>
- 7.** Scott Chacon, *github flow*
<http://scottchacon.com/2011/08/31/github-flow.html>
- 8.** William Durand, *Deploying With Git*
<http://williamdurand.fr/2012/02/25/deploying-with-git/>
- 9.** Abhijit Menon-Sen, *Using Git to manage a web site*
<http://toroid.org/ams/git-website-howto>
- 10.** *Git FAQ*
<https://git.wiki.kernel.org/index.php/GitFaq>

Dodatek B

Słownik terminów angielskich

bare repository — repozytorium surowe (tj. pozbawione obszaru roboczego i indeksu)

bug cracking system — system śledzenia błędów

commit —

1. rzeczownik: rewizja

2. czasownik: zatwierdzać zmiany, wykonywać rewizje

detached HEAD — stan, w którym wskaźnik symboliczny HEAD nie wskazuje żadnej gałęzi, a konkretną rewizję; rewizje tworzone w tym stanie nie są dołączane do żadnej gałęzi

directed acyclic graph — graf acykliczny skierowany

disjoint branches — gałęzie rozłączne

fork — operacja rozgałęziania, która jest dostępna w serwisach *github.com* oraz *bitbucket.org*; powoduje ona sklonowanie repozytorium; użytkownik wykonujący operację *fork* zostaje właścicielem utworzonej kopii

index — indeks; baza danych zawierająca listę zmian, które zostaną uwzględnione przez kolejną rewizję

issue — błąd (pojedyncze zgłoszenie w systemie śledzenia błędów)

local repository — repozytorium lokalne (tj. takie, w którym wydajemy komendy Gita)

modified file — plik zmodyfikowany

pull request — żądanie aktualizacji

remote repository — repozytorium zdalne

repo — skrót od *repository*

repository — repozytorium

revision — rewizja

revision control system/software — system/oprogramowanie kontroli rewizji

SCM — *software configuration management* — zarządzanie kodem źródłowym oprogramowania

snapshot — bieżący stan plików (migawka)

stage area — indeks

staged file — plik zaindeksowany (tj. dodany do bazy danych *index*)

tracked file — plik śledzony

tracking branch — gałąź śledzona

unmodified file — plik aktualny

unstaged file — plik niezaindeksowany (tj. niewystępujący w bazie danych *index*)

untracked file — plik nieśledzony

version control system/software — system/oprogramowanie kontroli wersji

working directory — obszar roboczy

Skorowidz

B

- Bazaar, 270
baza danych
repozytorium, *Patrz:* repozytorium
rewizji, *Patrz:* repozytorium
Bitbucket, 20, 269, 270, 280, 283, 287, 292, 293, 313
interfejs, 275
branch, *Patrz:* gałąź

C

- commit, *Patrz:* rewizja, operacja zatwierdzania
Cygwin, 278
cytowanie, 102

D

- diagram stanów, 78
DVCS, *Patrz:* system kontroli wersji
dziennik reflog, 100, 137, 154, 161, 162, 178, 237

E

- edytor
tekstowy, 28, 58
vi, 28, 59

F

- fast forward, *Patrz:* przewijanie do przodu
file
added, *Patrz:* plik dodany
deleted, *Patrz:* plik usunięty

- modified, *Patrz:* plik zmodyfikowany
renamed, *Patrz:* plik o zmienionej nazwie
staged, *Patrz:* plik indeksowany
unmodified, *Patrz:* plik aktualny
unstaged, *Patrz:* plik niezaindeksowany
untracked, *Patrz:* plik nieśledzony
file modified, *Patrz:* plik zmodyfikowany
folder
domowy użytkownika, 107
roboczy, 43
fork, *Patrz:* rozgałęzianie

G

- gałąź, 141, 156, 161, 181, 303
bieżąca, 142, 148, 153, 193
lista, 194
lokalna, 194, 217, 285
łączenie, 167, 170, 171, 175, 176, 178, 181, 195, 233
master, 141, 153, 211, 215, 285
przelaczanie, 145, 147, 208
przesłanie ze zmianą nazwy, 209
przesyłanie, 210
rozłączność, 150
śledzona, 193, 194, 199, 209
tworzenie, 141, 143, 144, 147
tymczasowa, 221
usuwanie, 141, 153, 154, 157, 209
wysyłanie, 206
zawieranie, 150
zdalna, 194, 209
zmiana nazwy, 155
GFM, 313
Github, 19, 20, 269, 279, 283, 286, 292, 293, 313
interfejs, 271

Github Flavored Markdown, *Patrz:* GFM
 GitPad, 28
 Google Code, 19
 graf niespójny, 223

H

HEAD, 93, 96
 hosting, 19

I

importowanie kodu, 286
 indeks, 56, 72, 127, 128, 198, 245
 indeksowanie, 52

J

język
 bash, 109
 MarkDown, 313
 jQuery, 39

K

klient
 Git, 278
 github.com, 277
 klucz SSH, 278, 279, 280
 komenda, 130
 echo, 135
 find, 24, 131, 134
 git, 24, 130
 git add, 43, 51, 57, 75, 128, 130, 133, 134
 git archive, 89, 291
 git backup, 210
 git branch, 46, 130, 133, 143, 144, 153, 155,
 182, 189, 194, 226, 227
 git checkout, 45, 86, 125, 133, 145, 147, 148,
 164, 182, 208, 217, 229, 241, 242, 243, 260,
 264
 git checkout master, 46
 git clone, 30, 129, 131, 187, 191
 git commit, 43, 58, 75, 116, 117, 128, 133,
 134, 162, 199, 260
 git config, 130, 183, 193, 227, 259, 265
 git diff, 157, 183, 245, 248, 263, 265
 git fetch, 188, 193, 195, 226, 227
 git gc, 183
 git gui, 132
 git help add, 130
 git help branch, 130
 git help config, 130

git help init, 130
 git init, 29, 130, 131, 141, 188
 git log, 33, 91, 96, 104, 131, 132, 173, 255,
 263, 265
 git merge, 168, 170, 171, 173, 176, 178, 181,
 183, 195, 217, 228, 233, 238, 239
 git mv, 60
 git prune, 164, 183
 git pull, 72, 129, 193, 196, 198, 199, 228, 301
 git push, 72, 129, 193, 199, 210, 217, 228, 301
 git rebase, 116, 117, 119, 175, 176, 178, 181,
 184, 217, 236, 238, 240, 264
 git reflog, 100, 137, 184
 git remote, 227
 git remote add, 192, 222, 225, 227, 301
 git remote add origin, 188, 191, 219, 220, 226
 git remote rm, 227
 git reset, 39, 86, 116, 133, 157, 173, 178, 184,
 189, 226
 git revert, 116, 120, 127, 181
 git rev-list, 137
 git rev-parse, 101, 137
 git rm, 51, 59, 134
 git shortlog, 131
 git show, 85, 242, 243, 264
 git simple commits, 121
 git simple-commit, 110, 112
 git simple-loop, 111
 git status, 68, 75, 128, 133, 182
 git symbolic, 182
 git tag, 84
 gitk, 132
 skrót, 107
 ssh, 24
 wc, 24
 konflikt, 233, 236, 263
 binarny, 238, 242, 263
 dublowanie, 238
 tekstowy, 233, 236, 242, 263
 konsola, 25, 26
 bash, 26
 kontekst, 248
 kontrola
 akceptowanych rewizji, 269
 spójności danych, 129
 uprawnień użytkowników, 269
 kopia bezpieczeństwa, 30, 291

L

Linux, 278
 lokalność, 129

M

menu kontekstowe, 26
Mercurial, 270
migawka, *Patrz: snapshot*

N

nazwa symboliczna HEAD, 93, 96

O

obszar roboczy, 39, 56, 72, 127, 128, 132, 148, 198, 211, 245
openssh, 278
operacja zatwierdzania, 15, 16

P

pakiet openssh, 278
parent, *Patrz: rodzic*
plik
 aktualny, 51, 52, 54, 128
 binarny, 257
 dodany, 69
 git/HEAD, 93
 git/info/exclude, 75, 76, 135
 gitattributes, 258, 261
 gitignore, 75, 76, 135
 ignorowany, 53, 75, 78, 128
 konfiguracyjny, 76, 131
 konfiguracyjny gitconfig, 107
 nieignorowany, 53, 78, 128
 nieśledzony, 51, 52, 53, 57
 niezaindeksowany, 53, 54, 56, 59, 128
 o zmienionej nazwie, 69
 odpowiadające rewizji, 89
 przywracanie, 40
 stan, 68, 127
 stan dwuliterowy, 69
 tekstowy, 238, 257
 usunięty, *Patrz: plik usunięty*
 zaindeksowany, 53, 56, 128
zmiana nazwy, 60
zmieniony, 255
zmodyfikowany, 51, 52, 69
polecenie, *Patrz: komenda*
praca grupowa, 18, 30, 127, 233, 291, 293
program GitPad, 28
projekt
 historia, 18, 33, 36, 115, 116, 125, 127
 hosting, 19
 stan, 17, 108, 125

protokół

file, 277
Git, 277
HTTPS, 277
SSH, 277, 280

przestrzeń robocza, 51, 52
przewijanie do przodu, 168, 169, 199, 223
przodek, 96
pull request, *Patrz: żądanie aktualizacji*

R

repo, *Patrz: repozytorium*
repository, *Patrz: repozytorium*
 bare, *Patrz: repozytorium surowe*
repozytorium, 14, 18, 29, 30, 39, 48, 52, 56, 127, 128, 141
 główne, 216
 inicjalizowanie, 29
 klonowanie, 30, 187, 188, 191, 293, 301, 302
 lokalne, 187, 188, 195, 198, 199, 225, 302
 łączenie, 219, 223
 o historii nieliniowej, 94
 prywatne, 271, 291, 292
 publiczne, 271
 stan, 61
 surowe, 72, 198, 216, 226
 synchonizacja, 195, 225
 śledzone, 272
 tworzenie, 283, 287
 uaktualnianie, 197, 199
 współdzielenie, 269, 293
 zdalne, 187, 188, 192, 193, 195, 199, 209, 211, 219, 225, 280
 zwykłe, 72, 210
revision, *Patrz: rewizja*
revision control system, *Patrz: system kontroli wersji*
rewizja, 16, 18, 34, 45, 52, 127, 156, 245
 akceptowana, 269
 domyślna, 93
 graf niespójny, 223
 identyfikacja, 91, 92, 93, 100
 identyfikator, 34
 łączenie, 115, 117, 119
 tworzenie, 58, 142, 143
 usuwanie, 115, 116, 127
 zabezpieczanie przed utratą, 209
 zgubiona, 149, 163
rodzic, 95, 96, 97
rozgałęzianie, 293

S

serwer
 bitbucket.org, 25
 SSH, 25
 skrót SHA-1, 91, 92, 96, 129, 148, 154, 164, 195
 snapshot, 44
 Source Forge, 19
 stan detached HEAD, 148, 163, 164, 181, 236, 237
 stan projektu, *Patrz*: projekt stan
 strumień przekierowanie, 102
 system
 jądro, 13
 kontroli rewizji, *Patrz*: system kontroli wersji
 kontroli wersji, 13, 270
 śledzenia błędów, 30, *Patrz*: śledzenie błędów

Ś

ścieżka dostępu, 24
 śledzenie błędów, 269, 291, 315, 319

T

tag, *Patrz*: znacznik
 annotated, *Patrz*: znacznik opisany
 lightweight, *Patrz*: znacznik lekki

U

ujednolicony format opisu, 246

V

vi, 28, 59

W

wiersz poleceń, 26, 102
 working area, *Patrz*: obszar roboczy
 working directory, *Patrz*: obszar roboczy

Z

zmienna
 środowiskowa
 PATH, 23
 znacznik, 83, 92, 96, 156, 291
 dane, 85
 dostępność, 85, 136
 konfliktu, 234
 lekki, 83, 84
 opisany, 83, 84
 tworzenie, 136
 usuwanie, 85, 136
 znak
 !, 108
 ", 102, 103
 ^, 97, 98, 99, 102
 |, 102
 <>, 102
 >, 102
 końca wiersza, 259, 260, 261
 tylda, 96, 98, 99
 złamania wiersza, 24

Ż

żądanie aktualizacji, 269, 293, 294, 301, 303, 319