Git – instrukcja dla studentów

mgr inż. Maciej Długosz mgr inż. Marek Kokot dr inż. Krzysztof Simiński

30 września 2017

Spis treści

1. Wstęp

Dobra organizacja pracy grupy programistów tworzących wspólny projekt wymaga wykorzystania wielu narzędzi. Do najbardziej podstawowych należą tzw. *systemy kontroli wersji*. Są to narzędzia przechowujące historię pracy nad projektem. Umożliwiają przeglądanie wszystkich zmian wprowadzonych do projektu od czasu jego rozpoczęcia.

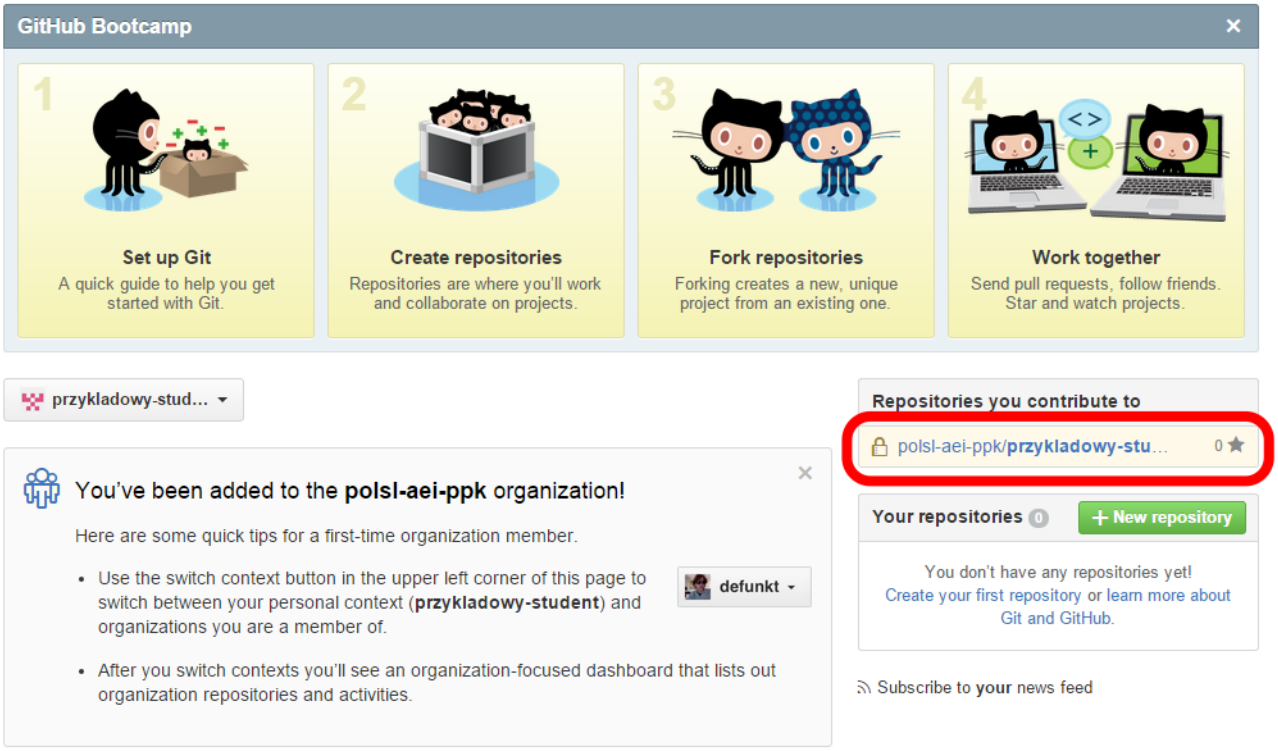
Podczas pracy na zajęciach będziemy wykorzystywać system kontroli wersji Git oraz platformę GitHub. Pierwszą czynnością jaką należy wykonać jest utworzenie konta na stronie `https://github.com`. Następnie należy przekazać prowadzącemu nazwę użytkownika utworzonego konta, swoje imię, nazwisko i numer sekcji laboratoryjnej. Na podstawie tych informacji prowadzący utworzy repozytorium. Ukończenie tej czynności następuje w momencie przyjęcia przez studenta zaproszenia, które zostaje mu wysłane pocztą elektroniczną. Do właściwego działania klientów Gita konieczne jest także potwierdzenie adresu mailowego.

2. GitHub

Na platformie GitHub każdy student otrzyma jedno *repozytorium*. W dużym uproszczeniu repozytorium można wyobrazić sobie jako folder dostosowany do przechowywania historii zmian plików oraz współdzielenia go przez wiele osób.

Repozytorium utworzone przez prowadzącego będzie wstępnie skonfigurowane. Aby dokonać pierwszego podglądu repozytorium należy:

1. zalogować się do utworzonego konta na stronie `https://github.com`,
2. kliknąć nazwę repozytorium (rys. 1).



Rysunek 1: Wybór repozytorium

Znajduje się tam struktura katalogów odpowiadająca organizacji zajęć, np.:

- Temat 01,
- Temat 02,
- ...
- projekt 1,
- projekt 2,
- Student.

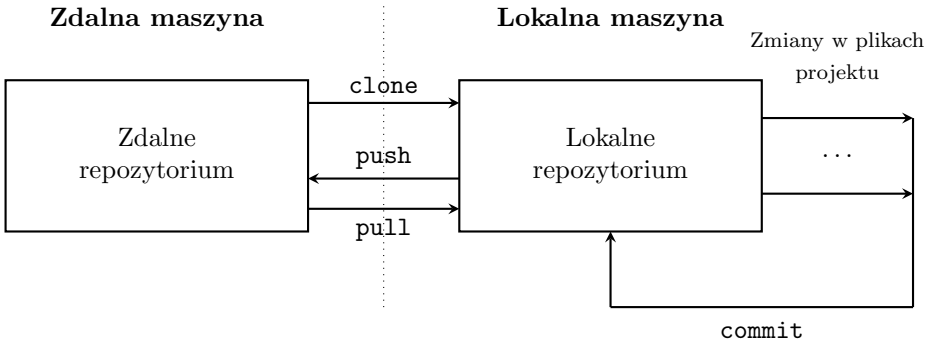
Katalog `Student` może być wykorzystany przez studenta jako brudnopis.

Mimo że wiele operacji jest dostępnych bezpośrednio przez stronę `https://github.com`, do pracy z systemem Git wykorzystuje się tak zwany *klient*. Jest to osobne narzędzie instalowane na komputerze umożliwiające pracę z repozytorium. Opis kilku klientów dostępnych dla systemu Windows znajduje się w dalszej części instrukcji.

3. System kontroli wersji Git – zasada działania

Ideą systemu kontroli wersji jest przechowywanie i synchronizacja wersji kodu źródłowego zapisanego w plikach tekstowych. Zasady działania różnych systemów kontroli, choć często pozornie podobne, mogą różnić się w istotnym stopniu. W celu prawidłowego wykorzystania tego narzędzia należy zwrócić uwagę na pewne niuasne. W tym rozdziale zostanie opisana zasada działania systemu Git oraz zostaną podkreślone różnice w stosunku do systemu Subversion (SVN).

Git jest systemem rozproszonym. Oznacza to, że repozytorium może (i powinno być) przechowywane w wielu kopiach umieszczanych na różnych maszynach. Rozwiązanie to zapewnia zdecydowanie większą niezawodność oraz bezpieczeństwo danych przechowywanych w repozytorium niż systemy scentralizowane (np. SVN), w których awaria serwera repozytorium może spowodować utratę (przynajmniej części) danych, a z pewnością uniemożliwia korzystanie z repozytorium do momentu przywrócenia prawidłowej pracy serwera.



Rysunek 2: Schemat przepływu danych w systemie Git

Pracę z rozproszonym systemem kontroli wersji często rozpoczynamy od wykonania kopii repozytorium udostępnianej przez inną maszynę na maszynę lokalną. Operację tą nazywamy klonowaniem (ang. `clone`). Zazwyczaj maszyna będąca źródłem kopii jest centralnym serwerem, którego zadaniem jest udostępnianie użytkownikom kopii repozytorium oraz scalanie zmian wprowadzanych w tych kopiach. Jednakże, jak zostało wcześniej wspomniane, praca z repozytorium Git może odbywać się także przy całkowitym braku bądź przy wystąpieniu awarii centralnego serwera. Kopię znajdującą się na serwerze nazywamy *repozytorium zdalnym*, a każdą kopię utworzoną na komputerze użytkownika – *repozytorium lokalnym*.

Po wykonaniu klonowania można przystąpić do pracy na plikach zawartych w lokalnym repozytorium. Po wykonaniu zadania (lub jego fragmentu) należy zapisać wprowadzone zmiany do repozytorium. Do tego celu służy operacja `commit`, która wymaga podania opisu wprowadzonych zmian oraz ich autora. Dane te są użyteczne przy przeglądaniu historii zmian plików. Ważne jest, aby zdawać sobie sprawę, że operacja ta zapisuje zmiany tylko w repozytorium lokalnym. W celu przeniesienia zmian z repozytorium lokalnego do zdalnego należy wykonać operację `push`[1].

Należy w tym miejscu podkreślić różnicę między operacją `clone` w Git, a `checkout` w SVN. Polecenie `clone` powoduje wykonanie kopii pełnego repozytorium wraz z historią operacji oraz gałęziami. Z kolei operacja `checkout` w systemie SVN powoduje pobranie jedynie aktualnej wersji plików zapisanych w repozytorium na komputer klienta. Różnica ta jest o tyle istotna, że operacja nazywana `checkout` w systemie Git także występuje, jednak posiada inną rolę. Powoduje ona przełączenie aktualnej wersji plików roboczych do wersji zapisanej w innej gałęzi. Zagadnienie gałęzi nie będzie szczegółowo omawiane.

W uproszczeniu schemat przepływu danych w wyniku wykonywania różnych operacji w systemie Git można przedstawić jak na rys. 2.

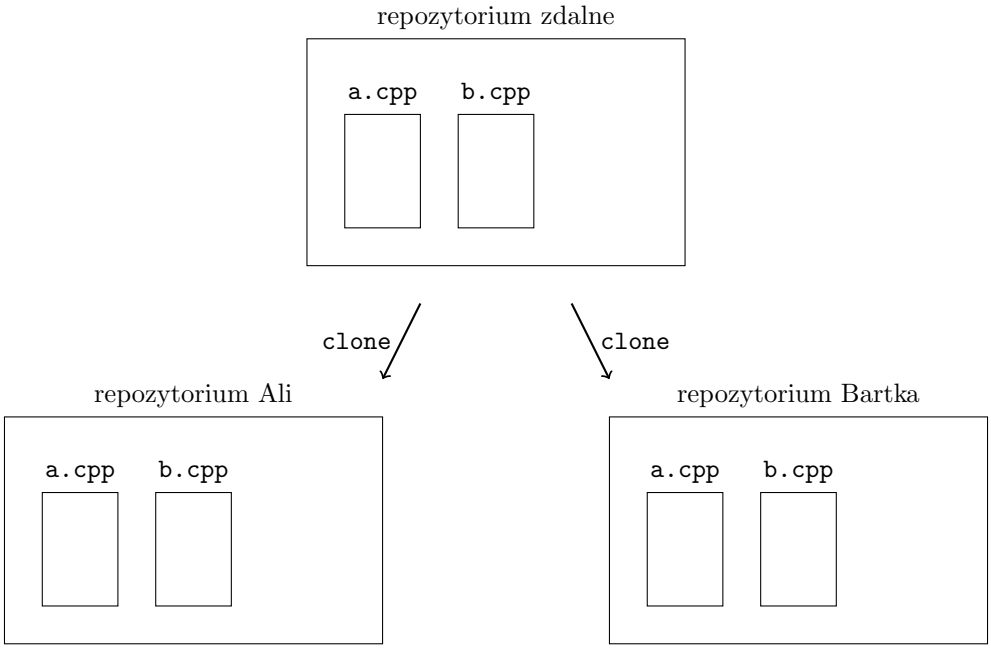
Podczas zajęć będziemy korzystali ze zdalnych repozytoriów dostępnych na serwerze GitHub. Podczas zajęć oraz pracy w domu studenci powinni wykorzystywać lokalne repozytoria uprzednio sklonowane z tego serwera. Należy pamiętać o regularnym wykonywaniu operacji `push` w celu zapisania zmian w zdalnym repozytorium.

3.1. Przykład

Typowy przebieg synchronizacji wersji plików źródłowym prześledźmy na przykładzie projektu, nad którym pracują Ala i Bartek. Najpierw oboje tworzą swoje lokalne repozytoria komendą `clone`. Utworzone zostaną lokalne kopie plików `a.cpp` i `b.cpp` (rys. 3). Komendę `clone` wykonuje się tylko raz – żeby utworzyć lokalne repozytorium.

Zarówno Ala jak i Bartek modyfikują lokalnie pliki. Ala zapisała zmiany pliku `a.cpp` do lokalnego repozytorium komendą `commit`. Bartek zaś wprowadził zmiany w pliku `a.cpp`, ale nie zapisał ich w repozytorium lokalnym. Zmiany Bartka nie są znane systemowi kontroli wersji (rys. 4). Ponadto Bartek utworzył nowy plik `c.cpp`, ale jeszcze nie poinformował systemu kontroli wersji o nowym pliku. Żeby poddać plik kontroli wersji, Bartek wykonuje komendę `add` (rys. 5). Teraz system kontroli wersji wie, że należy śledzić zmiany w pliku `c.cpp`. Jednak Bartek nie zatwierdził zmian w pliku `c.cpp` komendą `commit`. System Git wie, że plik `c.cpp` jest poddany kontroli wersji, ale zmiany w pliku roboczym nie zostały zatwierdzone w repozytorium lokalnym, są zatem nieznane systemowi kontroli wersji.

[1] W rzeczywistości wiele narzędzi wykorzystujących Gita zalicza operację `push` do ogólniejszej grupy – `sync`. Z taką właśnie opcją spotkamy się w dalszej części instrukcji.



Rysunek 3: Wykonując operację `clone` Ala i Bartek utworzyli swoje repozytoria, w których znajdują się kopie plików z repozytorium zdalnego.

Ala kończy pracę i zapisuje swoje zmiany w repozytorium zdalnym komendą `push`. Jej repozytorium lokalne i repozytorium zdalne są w pełni zsynchronizowane. Bartek na razie zapisał zmiany w swoim repozytorium lokalnym komendą `commit` (rys. 6). Bartek i Ala mają różne wersje plików, ale o tym nie wiedzą. Bartek chciałby przesłać swoje zmiany do repozytorium zdalnego komendą `push`. Wtedy dowiaduje się, że jest to niemożliwe, bo w repozytorium zdalnym są modyfikacje, których nie ma w swoim repozytorium lokalnym. Musi najpierw pobrać te zmiany komendą `pull`. W repozytorium Bartka jego wersja zatwierdzona komendą `commit` i wersja z repozytorium zdalnego zostają automatycznie scalone przez system kontroli wersji. Bartek w swoim repozytorium ma pliki z modyfikacjami swoimi i Ali (rys. 7). W czasie gdy Bartek uaktualnia swoje repozytorium, Ala wprowadziła zmiany w kopii roboczej pliku `a.cpp`, ale jeszcze ich nie zatwierdziła. Bartek ma najaktualniejszą wersję plików. Przesyła je do repozytorium zdalnego – zostaje także przesłany utworzony przez Bartka plik `c.cpp` (rys. 8). Ala chciałaby zapisać w repozytorium zmiany w kopii roboczej pliku `a.cpp`. W tym celu wykonuje trzy operacje w następującej kolejności:

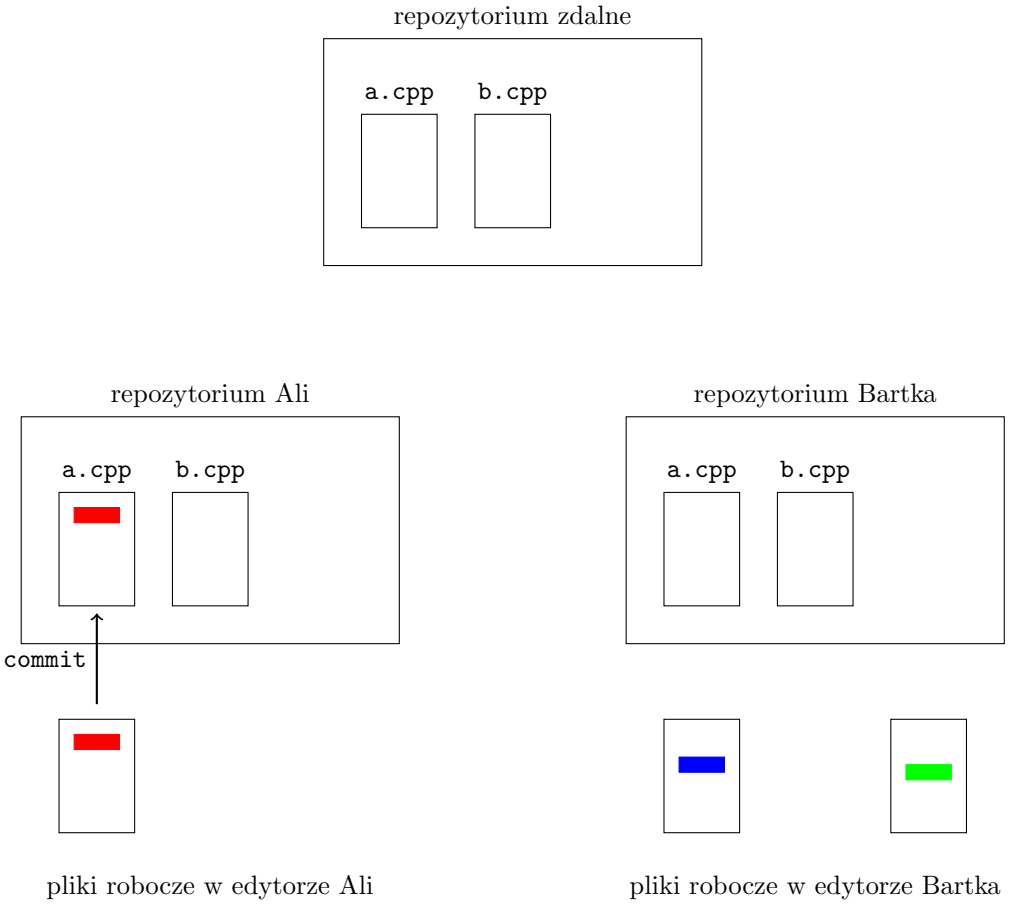
1. `commit` – najpierw zatwierdza zmiany w repozytorium lokalnym (rys. 9), jeżeli tego nie zrobi, zmiany w kopiach roboczych plików przepadną,

2. `pull` – pobiera najaktualniejszą wersję plików z repozytorium zdalnego i automatycznie scala ją ze swoimi zatwierdzonymi zmianami (rys. 10),

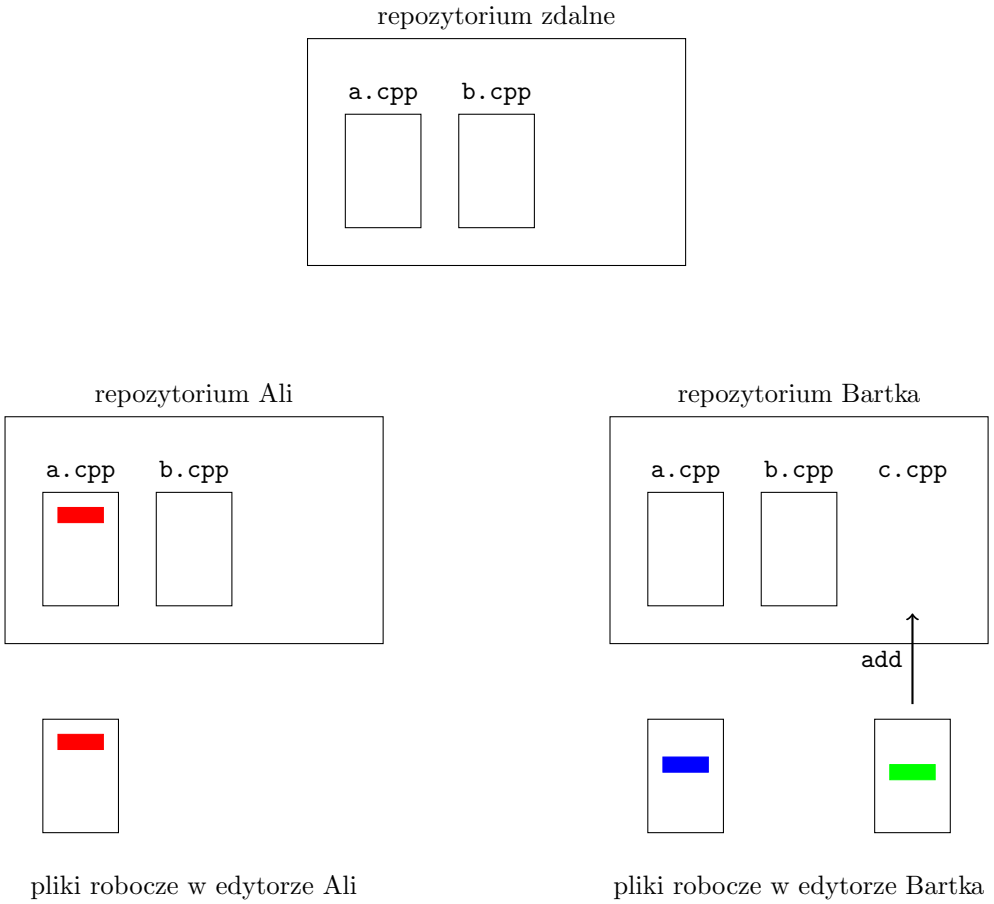
3. `push` – przesyła do repozytorium zdalnego scaloną wersję (rys. 11).

Bartek po przerwie wraca do projektu. Najpierw pobiera komendą `pull` zmiany, których dokonali współpracownicy. Teraz zarówno w repozytorium zdalnym, w repozytoriach lokalnych Ali i Bartka jest spójna, ta sama aktualna wersja plików (rys. 11).

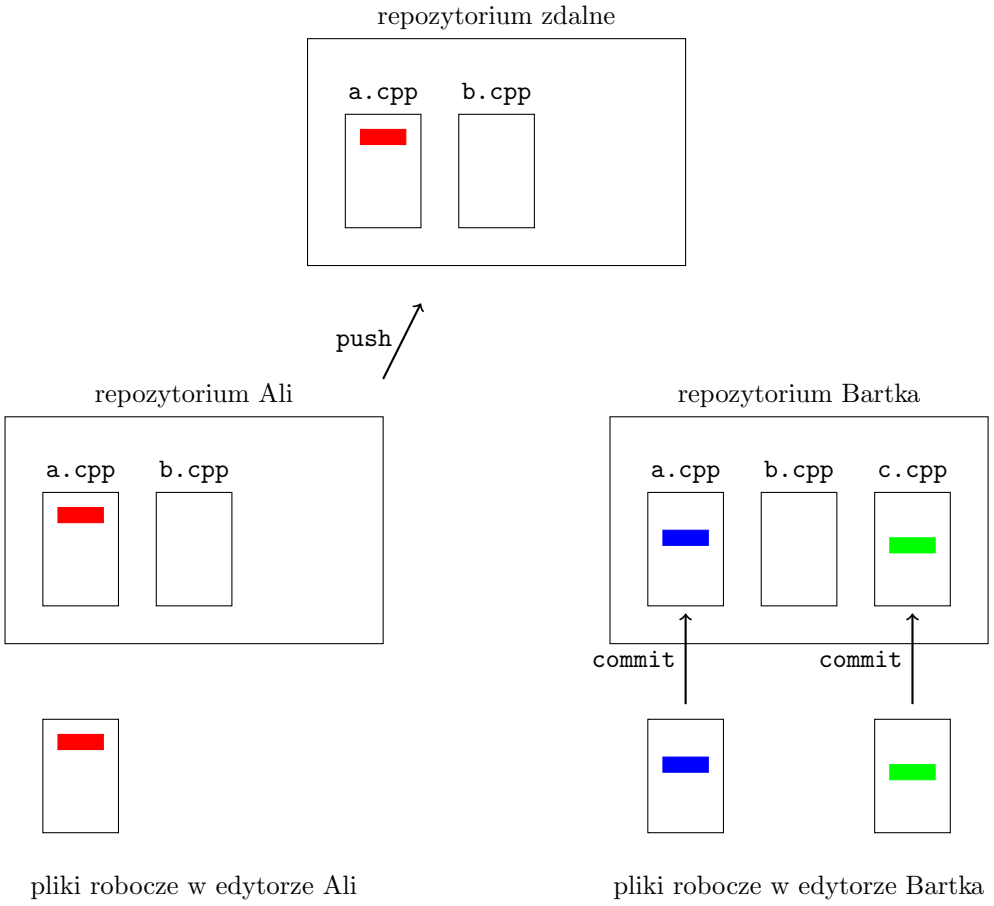
Zanim Ala i Bartek zaczęli pracę, w repozytorium były pliki `a.cpp` i `b.cpp`. Ala modyfikowała tylko plik `a.cpp`. Bartek modyfikował plik `a.cpp`, dodał i modyfikował plik `c.cpp`. Plik `b.cpp` nie był modyfikowany przez Alę ani Bartka. System kontroli wersji zapewnia, że wszyscy użytkownicy mają dostęp do najaktualniejszej (i wszystkich wcześniejszych zatwierdzonych) wersji wszystkich plików. Określaniem, które pliki trzeba zsynchronizować, synchronizacją i scalaniem różnych wersji tego samego pliku zajmuje się system kontroli wersji.



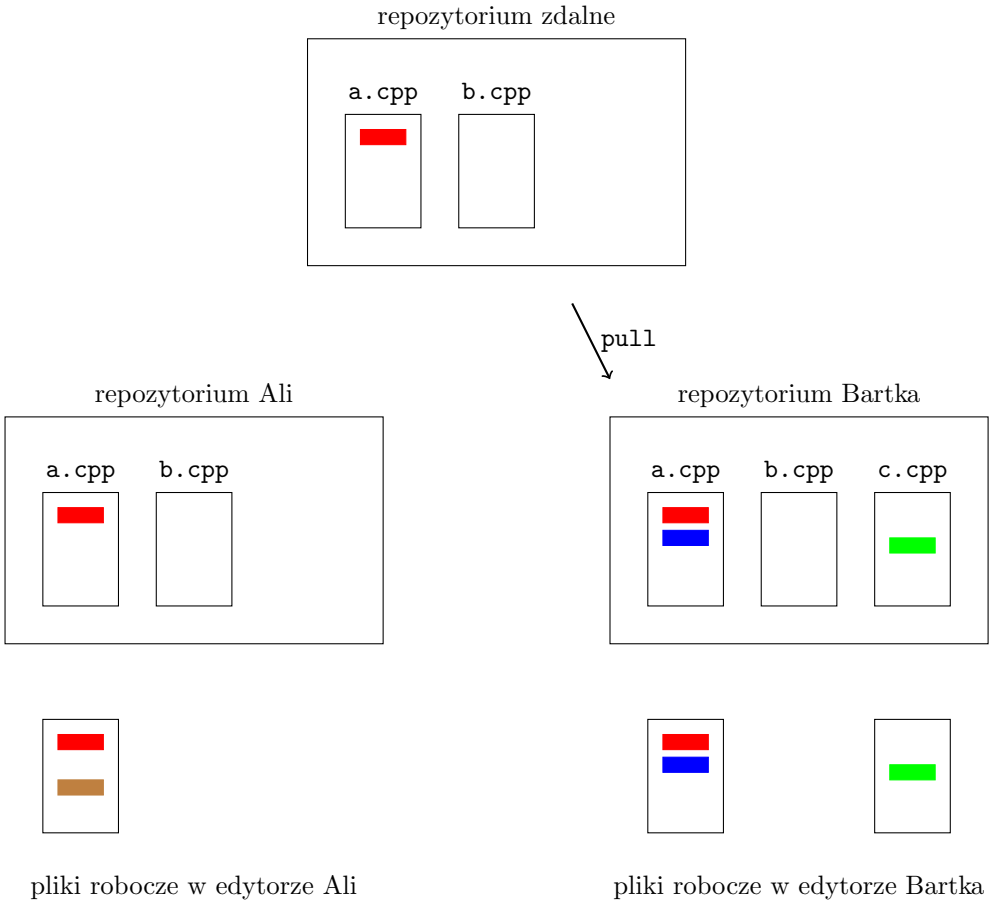
Rysunek 4: Ala zmodyfikowała plik `a.cpp` i zapisała zmiany w swoim repozytorium lokalnym komendą `commit`. Bartek zmodyfikował plik `a.cpp`, ale nie zapisał zmian w swoim repozytorium lokalnym. Bartek utworzył plik roboczy `c.cpp`, ale nie umieścił go pod kontrolą systemu kontroli wersji.



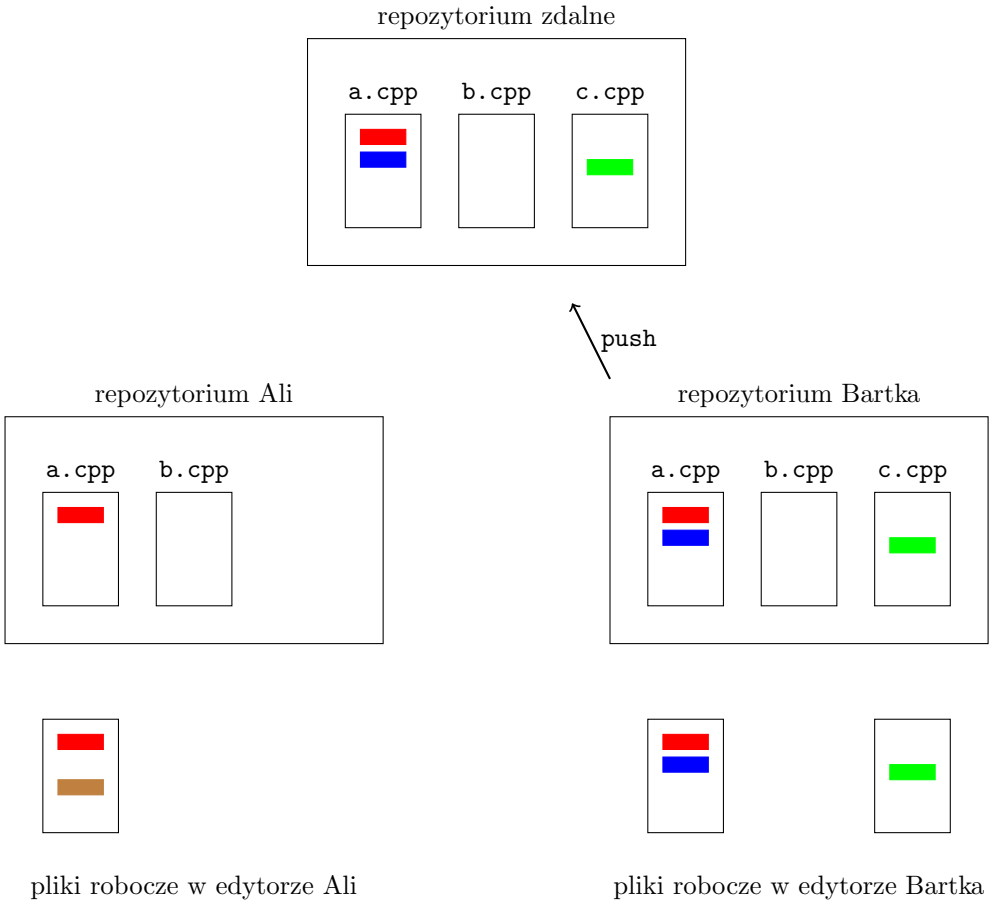
Rysunek 5: Bartek poinformował system kontroli wersji komendą `add`, że plik `c.cpp` należy poddać kontroli wersji. Jednak zmiany z pliku `c.cpp` nie zostały jeszcze zatwierdzone w repozytorium lokalnym.



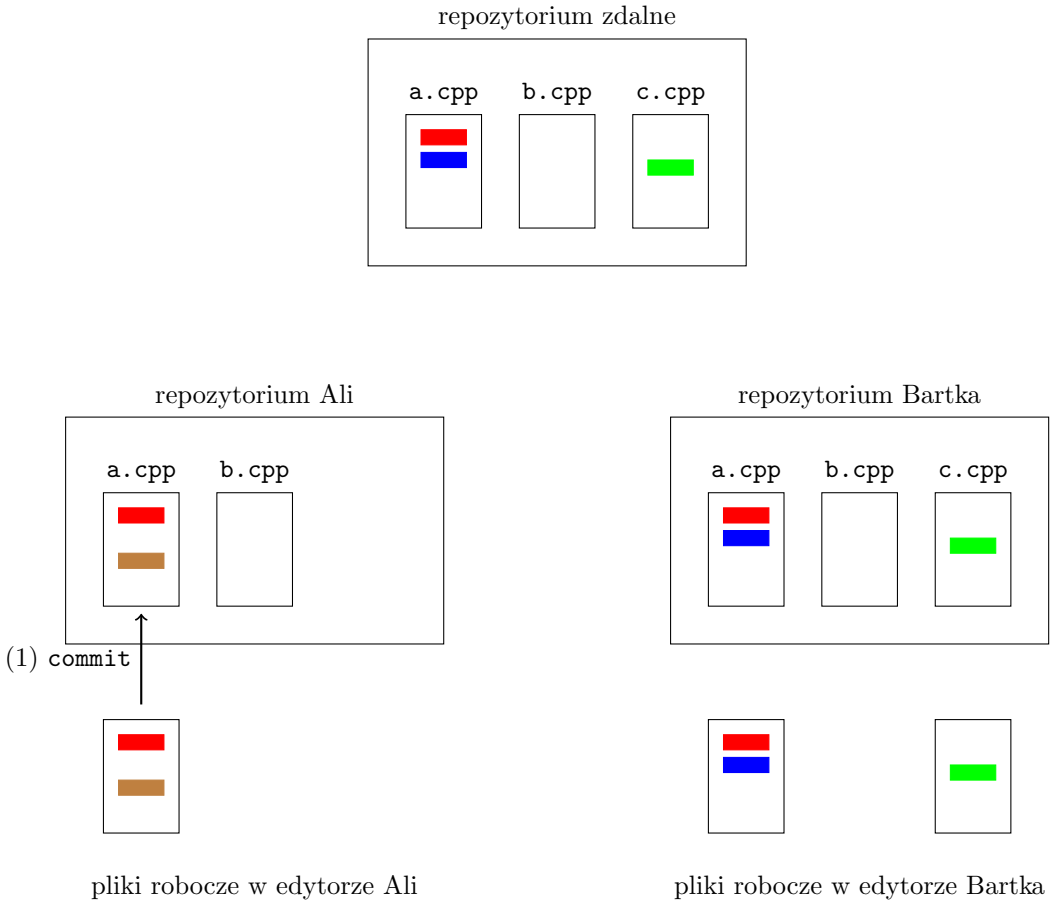
Rysunek 6: Ala zapisała zmiany w pliku `a.cpp` do repozytorium zdalnego komendą `push`. Bartek zapisał modyfikacje plików `a.cpp` i `c.cpp` do swojego repozytorium lokalnego. Ala i Bartek mają w swoich repozytoriach różne wersje pliku `a.cpp`.



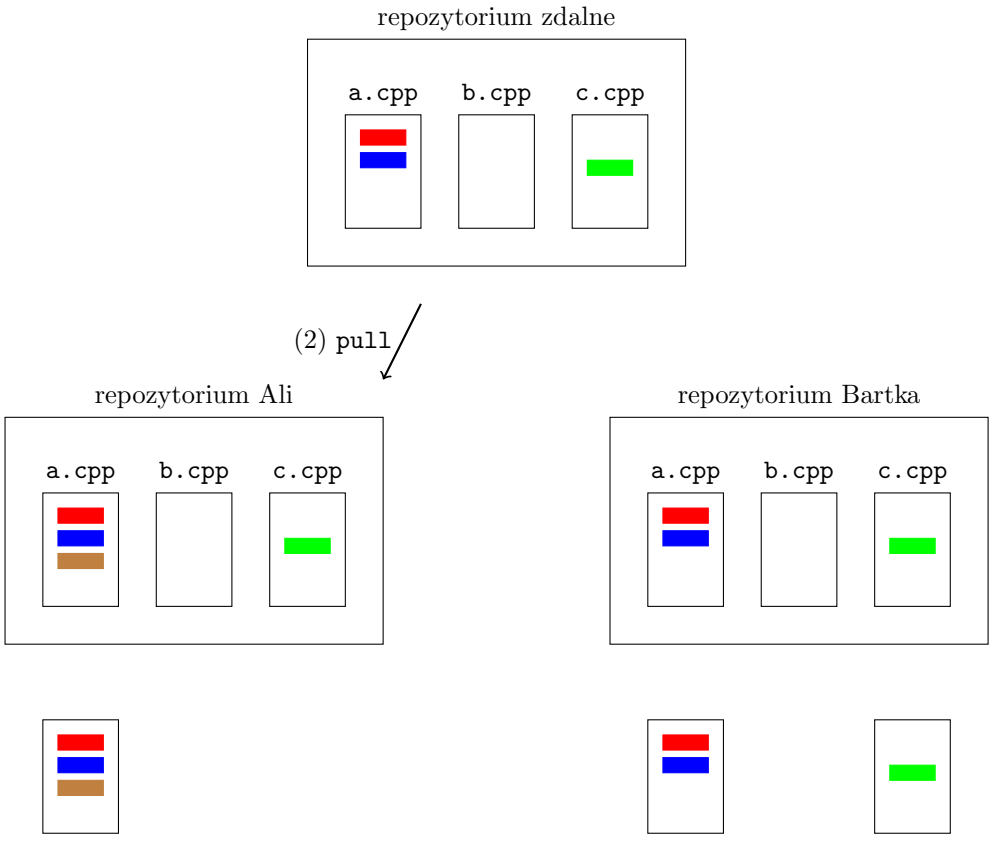
Rysunek 7: Bartek nie może przesłać plików ze swojego repozytorium do repozytorium zdalnego, ponieważ przesłanie ich spowodowałoby utratę modyfikacji Ali. Zatem Bartek najpierw pobiera wersję pliku `a.cpp` z repozytorium zdalnego komendą `pull`. Obie wersje zostają automatycznie scalone. Bartek ma już wersję Ali z repozytorium zdalnego scaloną ze swoimi zmianami. W tym samym czasie Ala zmodyfikowała plik `a.cpp`, ale nie zatwierdziła tej zmiany w swoim repozytorium.



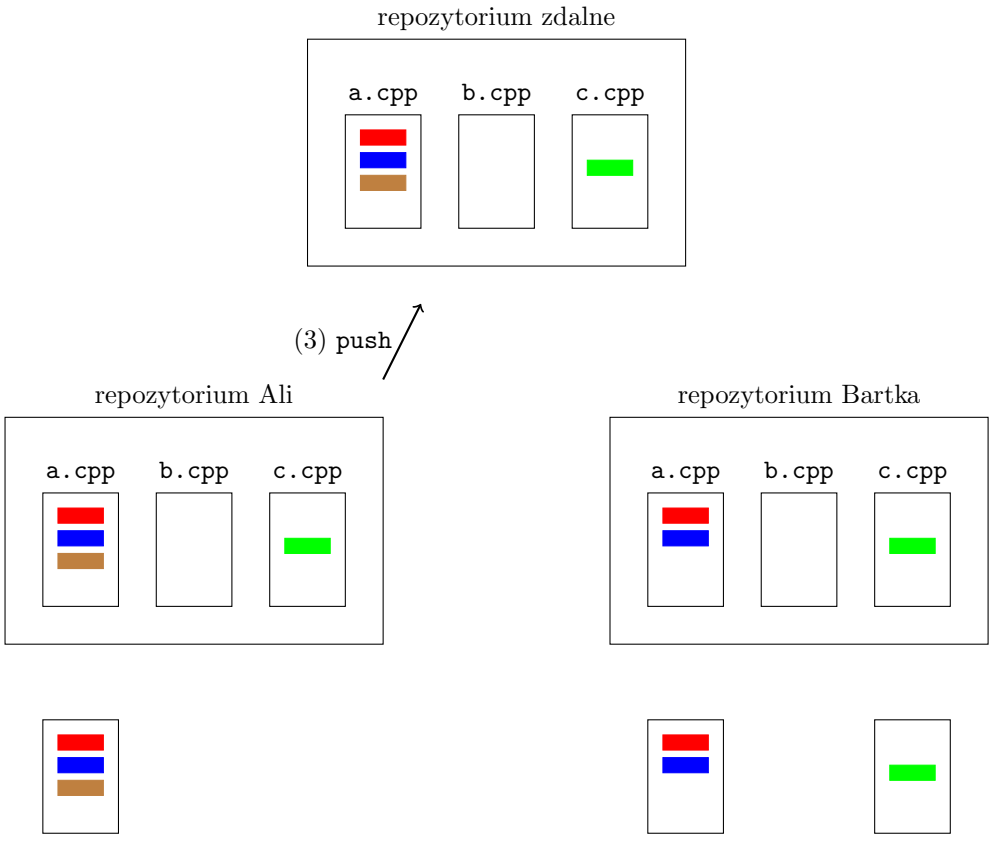
Rysunek 8: Ala zmodyfikowana swoją kopię pliku `a.cpp`, ale nie zatwierdziła zmian w swoim repozytorium. Bartek przesyła plik `c.cpp` i scaloną wersję pliku `a.cpp` do repozytorium zdalnego.



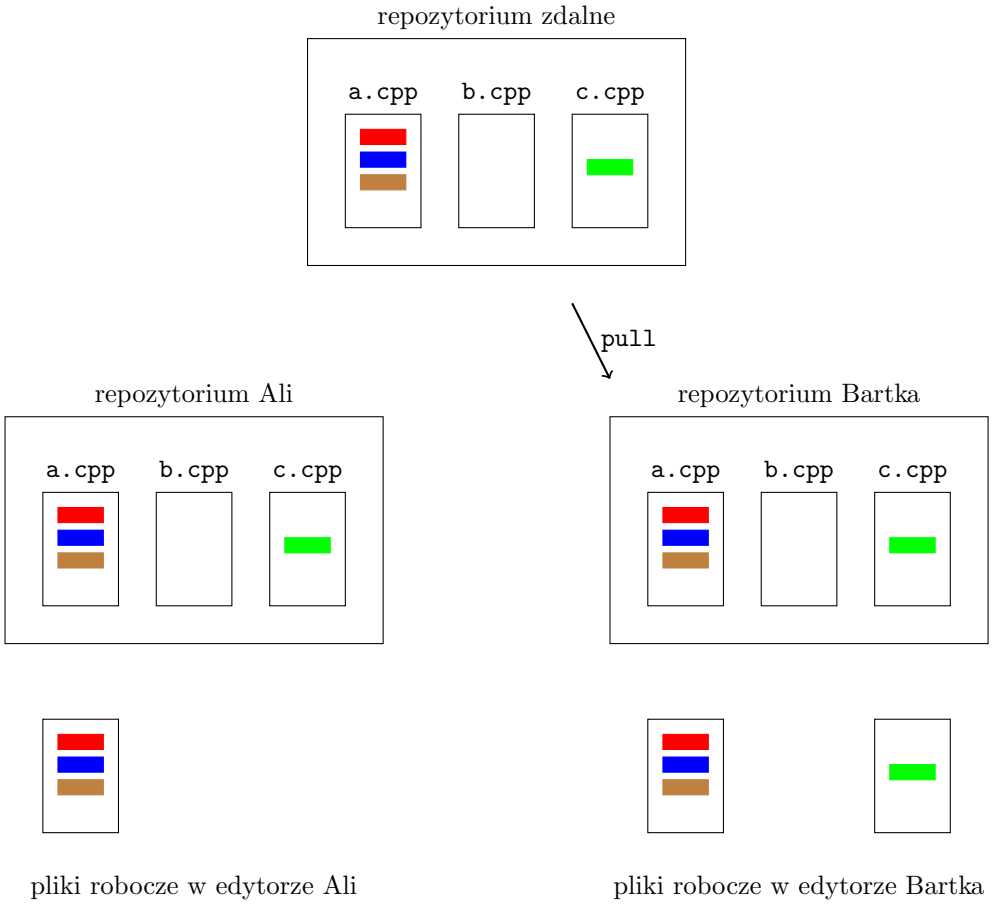
Rysunek 9: Ala, żeby przesłać swoją wersję pliku `a.cpp`, musi wykonać następujące operacje: (1) `commit`, żeby zatwierdzić zmiany w swoim repozytorium lokalnym, ...



Rysunek 10: ... (2) `pull`, żeby pobrać wersję Bartka i scalić ją ze swoimi zmianami, ...



Rysunek 11: ... i wreszcie (3) komendę `push`, żeby przesłać scaloną wersję do repozytorium zdalnego.



Rysunek 12: Bartek pobiera z repozytorium zdalnego zmienione pliki i scala je plikami w swoim repozytorium komendą `pull`. Ala i Bartek mają tę samą wersję wszystkich plików.

4. Obsługa klienta Git

Podstawowym sposobem korzystania z systemu Git jest wydawanie komend w linii poleceń. Ten sposób zadziała, jeżeli tylko w systemie został zainstalowany Git. Dodatkowo można korzystać z nakładek graficznych, które ułatwiają korzystanie z systemu Git. Nakładki te uruchamiają odpowiednie komendy Git. Jednak często ich funkcjonalność stanowi tylko podzbiór funkcjonalności systemu Git – nie wszystkie komendy Git są obsługiwane przez nakładkę.

4.1. Obsługa z linii poleceń

W podrozdziale tym zostaną przedstawione podstawowe polecenia obsługi systemu Git z linii poleceń. Bardziej szczegółowy opis zarówno systemu Git, jak i obsługi z poziomu linii poleceń można znaleźć pod adresem `http://git-scm.com/book/pl/v1/`.

Klonowanie repozytorium

Żeby sklonować repozytorium, czyli utworzyć lokalną kopię repozytorium, zależy użyć polecenia `clone`:

```
git clone adres_repozytorium
```

na przykład:

```
git clone https://github.com/polsl-aei-ppk/przykladowy-student-gr1-s1-repo
```

W wyniku tej operacji powstaje lokalne repozytorium.

Zapis zmian do lokalnego repozytorium

Aby podejrzeć aktualny stan repozytorium lokalnego, należy użyć plecenia:

```
git status
```

Komenda

```
git log
```

wyświetla informację o zatwierdzeniach zmian (`commit`) w repozytorium lokalnym (wyjście poprzez naciśnięcie klawisza `q`). Można wyświetlić także we wersji nieco bogatszej graficznie (na ile pozwala terminal) używając komendy z przełącznikiem

```
git log --graph
```

Nowo utworzony plik trzeba dodać do repozytorium. Inaczej, mimo że jest w katalogu objętym kontrolą wersji, nie będzie śledzony przez system kontroli wersji – będzie mieć status zasobu nieśledzonego (ang. *untracked*). Aby `plik` mógł być zapisany w repozytorium, należy wykonać polecenie:

```
git add plik
```

na przykład:

```
git add laboratorium/wprowadzenie/wprowadzenie/main.cpp
```

Jeśli przed poleceniem `commit` plik ten zostanie zmieniony, zmiany nie zostaną zapisane w repozytorium przy wykonaniu operacji `commit`. Aby mogło się tak stać, trzeba ponownie wykonać polecenie `git add`. W przeciwnym razie polecenie `commit` zapisze tylko wcześniejsze zmiany w lokalnym repozytorium. Aby zapisać zmiany w lokalnym repozytorium, należy wykonać polecenie:

```
git commit
```

Żeby uniknąć konieczności wykonywania komendy `add` przed każdym poleceniem `commit`, można użyć przełącznika `-a` (*add*) komendy `commit`, np.

```
git commit -a zmieniony.plik
```

Dodatkowo można za pomocą parametru `-m` dopisać wiadomość opisującą `commit`. Na przykład:

```
git commit -m 'ciekawa zmiana'
```

Niepodanie opisu operacji spowoduje uruchomienie edytora `nano` dla wpisania opisu. Można połączyć przełączniki `-a` (*add*) i `-m` (*message*), np.

```
git commit -am 'poprawka nawiazywania polaczania z~baza danych'
```

Przy próbie wykonania pierwszej komendy `commit` system Git będzie domagał się zapisania pewnych danych konfiguracyjnych. Trzeba wtedy podać swój adres mejlowy (taki jak zarejestrowany przy tworzeniu konta na serwerze Git) i login. Dzięki temu Git będzie mógł zidentyfikować, kto wykonuje polecenia.

```
git config user.name uzytkownik
git config user.email uzytkownik@abc.pl
```

Dane zostaną zapisane lokalnie w katalogu objętym kontrolą wersji w pliku `.git/config` w postaci wpisu (można te dane wpisać ręcznie w pliku):

```
[user]
email = uzytkownik@abc.pl
name = uzytkownik
```

Można te dane zapisać globalnie, wtedy należy skorzystać z przełącznika `--global`, np.

```
git config --global user.name uzytkownik
git config --global user.email uzytkownik@abc.pl
```

Na komputerach laboratoryjnych zalecamy korzystanie z wersji lokalnej – nie będzie wtedy problemów z korzystaniem z własnych repozytoriów przez różnych użytkowników tego samego komputera.

Zapis zmian do repozytorium zdalnego

Aby przesłać zmiany z lokalnego repozytorium na zdalne, należy użyć polecenia:

```
git push
```

Operacja `push` wymaga podania hasła.

Pobranie zmian z repozytorium zdalnego

Pobranie zmienionych plików z repozytorium globalnego do lokalnego wykonuje się poleceniem `pull`:

```
git pull
```

Operacja `pull` wymaga podania hasła. Jeżeli w zdalnym repozytorium są jakieś nowe zmiany, to zostaną one pobrane i scalone ze zmianami w repozytorium lokalnym.

Konflikty

Czasami system kontroli wersji nie jest w stanie automatycznie scalić zmian wprowadzonych przez autorów. Git zasygnalizuje, że jest konflikt i zaznaczy w pliku miejsca, gdzie nie powiodło się automatyczne scalanie w następujący sposób: Pierwsza wersja rozpoczyna się znacznikiem `<<<<<<<`, a kończy ciągiem `=======`. W tym miejscu zaczyna się wersja druga, która zakończona jest znacznikiem `>>>>>>>`.

```
automatycznie scalona tresc pliku

<<<<<<< HEAD:proba
Pierwsza wersja fragmentu pliku.
=======
Druga wersja fragmentu pliku.
>>>>>>> af8588a:proba

automatycznie scalona tresc pliku
```

Trzeba ręcznie wprowadzić poprawki, usunąć znaczniki konfliktu:

```
automatycznie scalona tresc pliku

Koncowa wersja fragmentu pliku.  // recznie rozwiazany konflikt

automatycznie scalona tresc pliku
```

Pozostaje już tylko zatwierdzenie zmian komendą `commit`:

```
git commit -am 'recznie usuniety konflikt'
```

Przenoszenie plików, zmiana nazw plików

Przenoszenie plików i zmiany nazwy wymagają poinformowania systemu kontroli wersji, inaczej system Git może nie wiedzieć, że została zmieniona nazwa pliku. Do tego służą komendy:

```
git rm plik-usun-mnie                  # usuniecie pliku
git mv stara-nazwa nowa-nazwa          # zmiana nazwy pliku
git mv plik ./nowa/lokalizacja/        # przeniesienie pliku w nowe miejsce
```

Używanie komend spoza systemu Git grozi utratą historii zmian plików.

Odtwarzanie poprzedniej wersji pliku

Może się zdarzyć, że w wyniku prac nad projektem autor popełni tak wiele błędów, że naprawienie ich jest dość trudne. Można wtedy odtworzyć wersję pliku zachowaną w systemie kontroli wersji. Przyjmijmy, że autor doprowadził swój projekt zawarty w pliku o nazwie `plik` do wersji nienadającej się użycia. Na szczęście system kontroli wersji pozwala na odtworzenie dowolnej historycznej wersji pliku. W tym celu należy odczytać numer zatwierdzenia (`commit`) – umożliwia to komenda:

```
git log plik
```

Zostaną wypisane zatwierdzenia pliku:

```
commit 0f57c94f5bdcc3ee93d1fb232efb6e60aa91dfae
Author: gonzo <gonzo@polsl.pl>
Date:   Sat Aug 16 15:48:06 2017 +0200

   dodanie 12. linii

commit fe6203ea4a0dcbe80bc2ca1f4e8bf6cb725047eb
Author: gonzo <gonzo@polsl.pl>
Date:   Thu Jul 7 23:35:19 2017 +0200

   linia dziewiata

commit 580757a6c01c0adcd01792a836bf9f420553415a
Author: gonzo <gonzo@polsl.pl>
Date:   Thu Jun 7 23:33:43 2017 +0200

   siodma linia
```

Wykonanie komendy

```
git checkout fe6203ea4a0dcbe80bc2ca1f4e8bf6cb725047eb plik
```

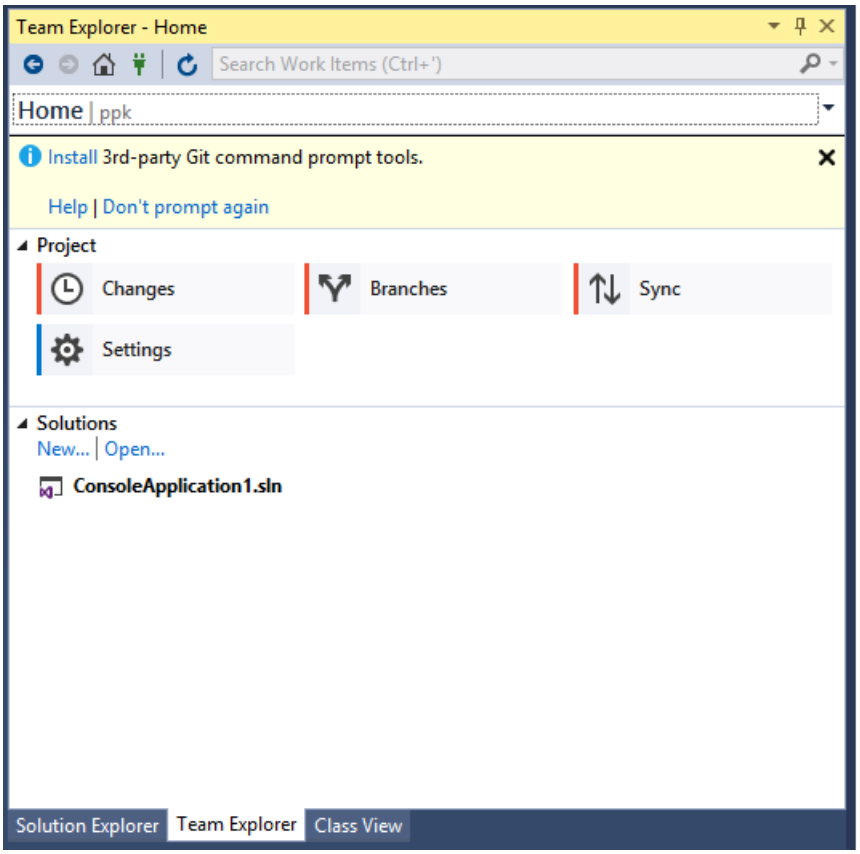
spowoduje odtworzenie `pliku` w wersji z 7 lipca 2017 r. Dlatego warto często zatwierdzać zmiany w plikach.

4.2. Nakładki graficzne

Istnieje wiele programów graficznych umożliwiających korzystanie z systemu Git. Wywołują one w tle odpowiednie komendy konsolowe systemu Git.

4.2.1 Instrukcja obsługi przez Visual Studio

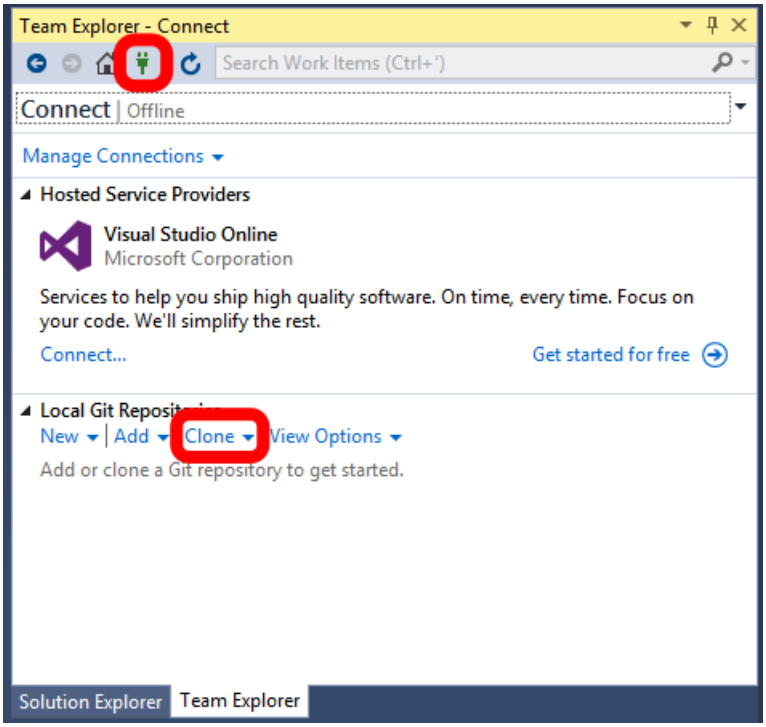
Po uruchomieniu środowiska Visual Studio należy upewnić się, że otwarte jest okno Team Explorer (rys. 13).



Rysunek 13: Team Explorer

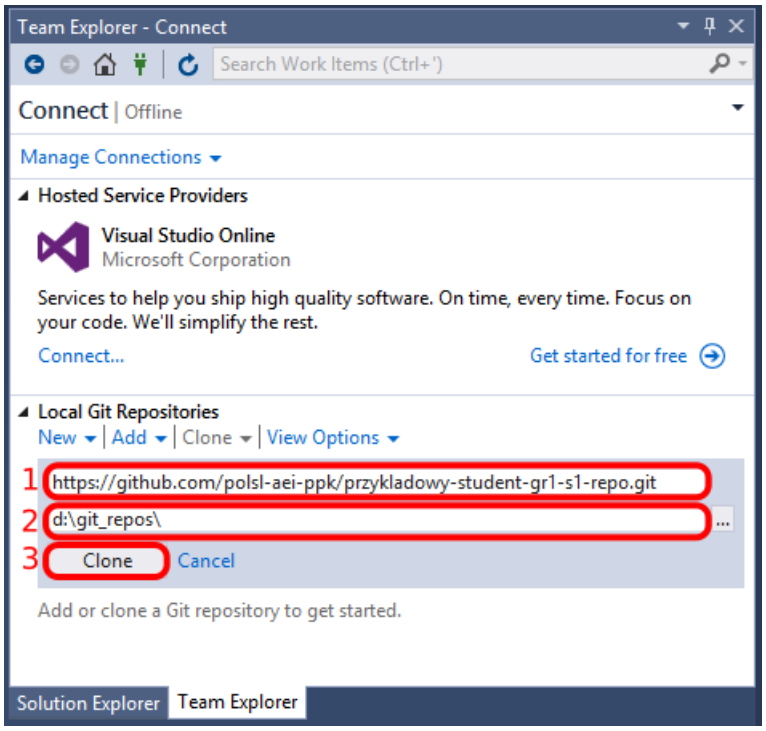
Jeżeli okno to nie jest otwarte, można otworzyć je z menu View → Team Explorer.

W celu wykonania klonowania repozytorium należy kliknąć ikonę *Manage Connections* (rys. 14) i z sekcji *Local Git Repositories* wybrać opcję *Clone*.



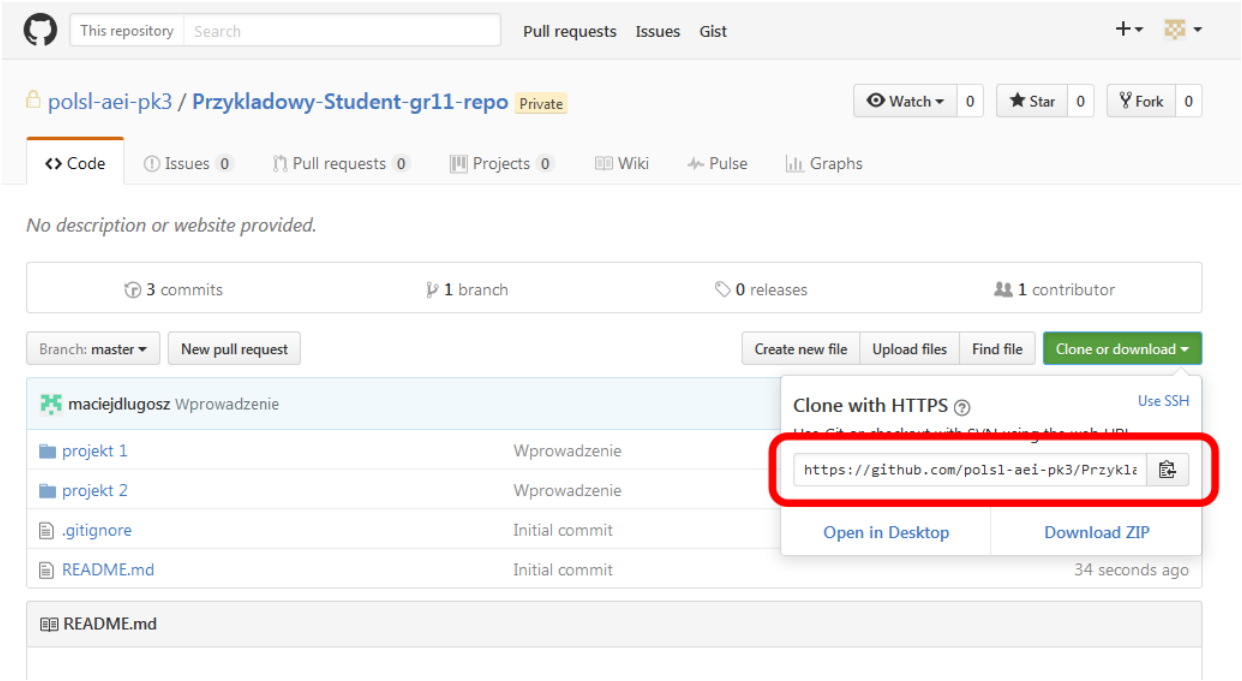
Rysunek 14: Połączenia

Następnie w wyświetlonych polach tekstowych należy podać kolejno adres zdalnego repozytorium (1) oraz ścieżkę do katalogu (2), w którym zostanie utworzone lokalne repozytorium, a następnie kliknąć `clone`(3) (rys. 15).



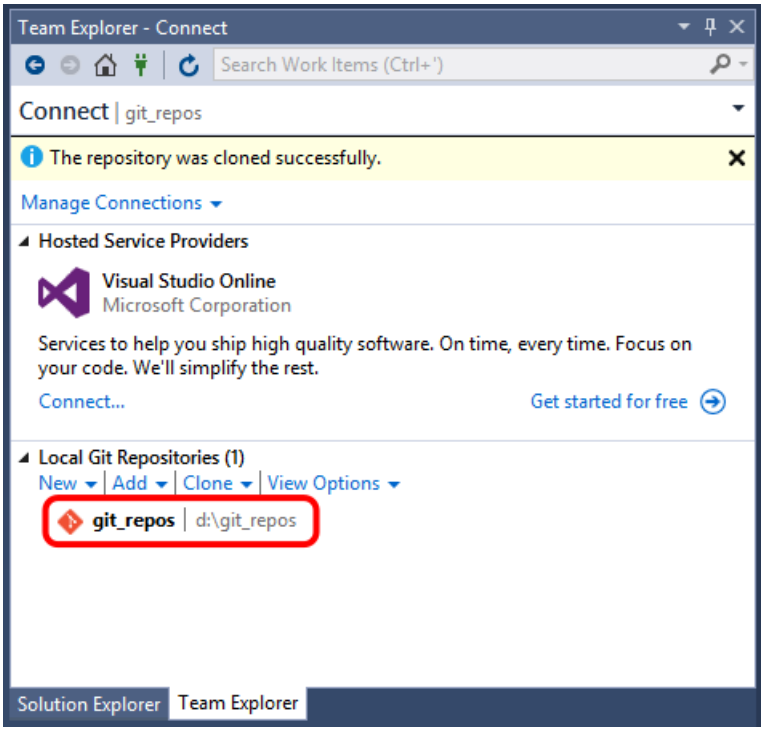
Rysunek 15: Klonowanie

Adres repozytorium można uzyskać podglądając repozytorium na stronie `https://github.com` (rys. 16).



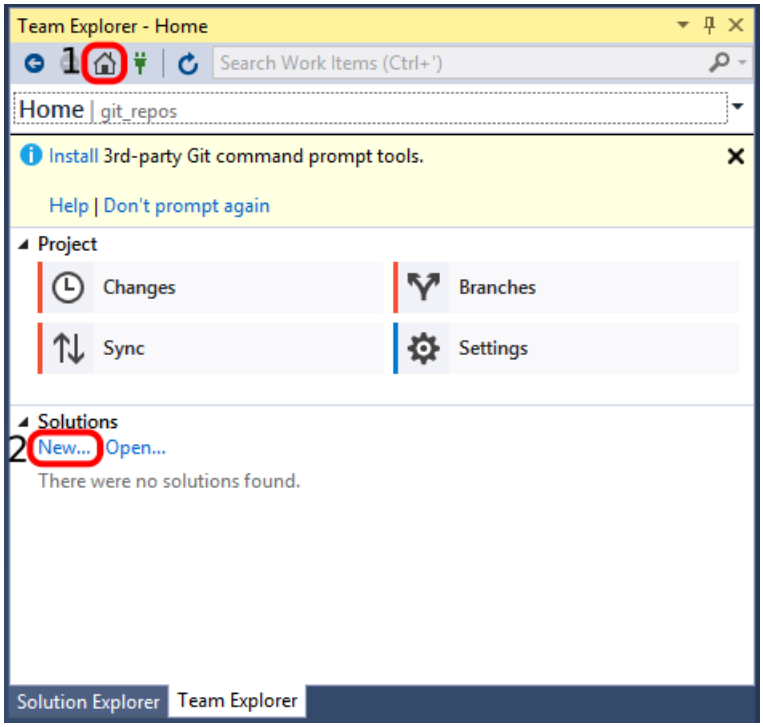
Rysunek 16: Adres repozytorium

Ścieżka do katalogu może być dowolna, chyba że prowadzący zarządzi inaczej. Po kliknięciu `clone` pojawi się okienko, w którym należy podać nazwę użytkownika oraz hasło konta GitHub. Po wykonaniu tych operacji repozytorium powinno być widoczne na liście lokalnych repozytoriów (rys. 17).



Rysunek 17: Lokalne repozytoria

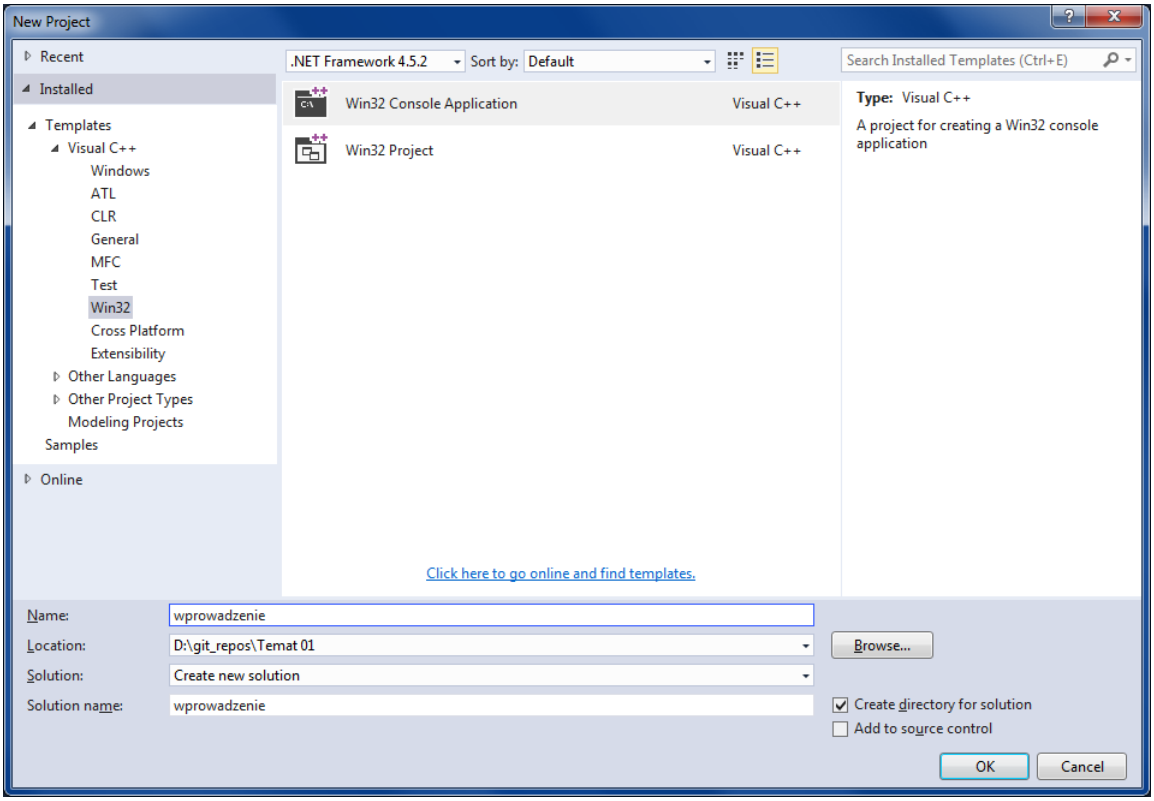
Tworzenie nowego projektu Visual Studio w repozytorium Aby utworzyć nowy projekt Visual Studio w istniejącym repozytorium należy w *Team Explorer* przejść do *Home*(1), a następnie z sekcji *Solutions* wybrać *New...* (2) (rys. 18).



Rysunek 18: Nowy projekt w repozytorium

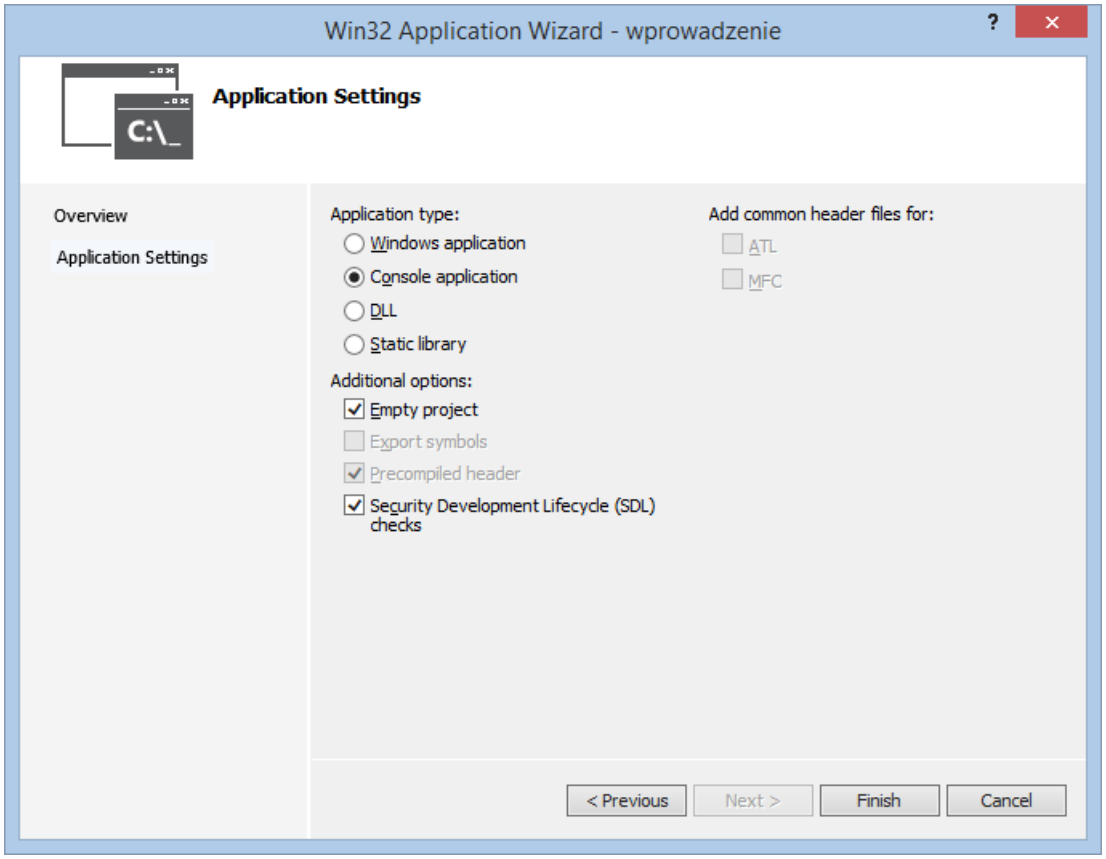
W oknie tworzenia nowego projektu należy:

1. jako typ projektu wybrać Visual C++ → Win32 Console Application,
2. wybrać nazwę projektu adekwatną do jego przeznaczenia albo podaną przez prowadzącego,
3. jako lokalizację należy podać ścieżkę do repozytorium lokalnego wraz z podkatalogiem, w którym projekt ma zostać uwtorzony. W przypadku pierwszych zajęć, zakładając że lokalne repozytorium znajduje się w lokalizacji `D:\git_repos`, powinna to być następująca ścieżka: `D:\git_repos\Temat 01\` (rys. 19),



Rysunek 19: Konfiguracja projektu

4. kliknąć *OK*,
5. kliknąć *Next*,
6. zaznaczyć opcję *Empty project* (rys. 20),



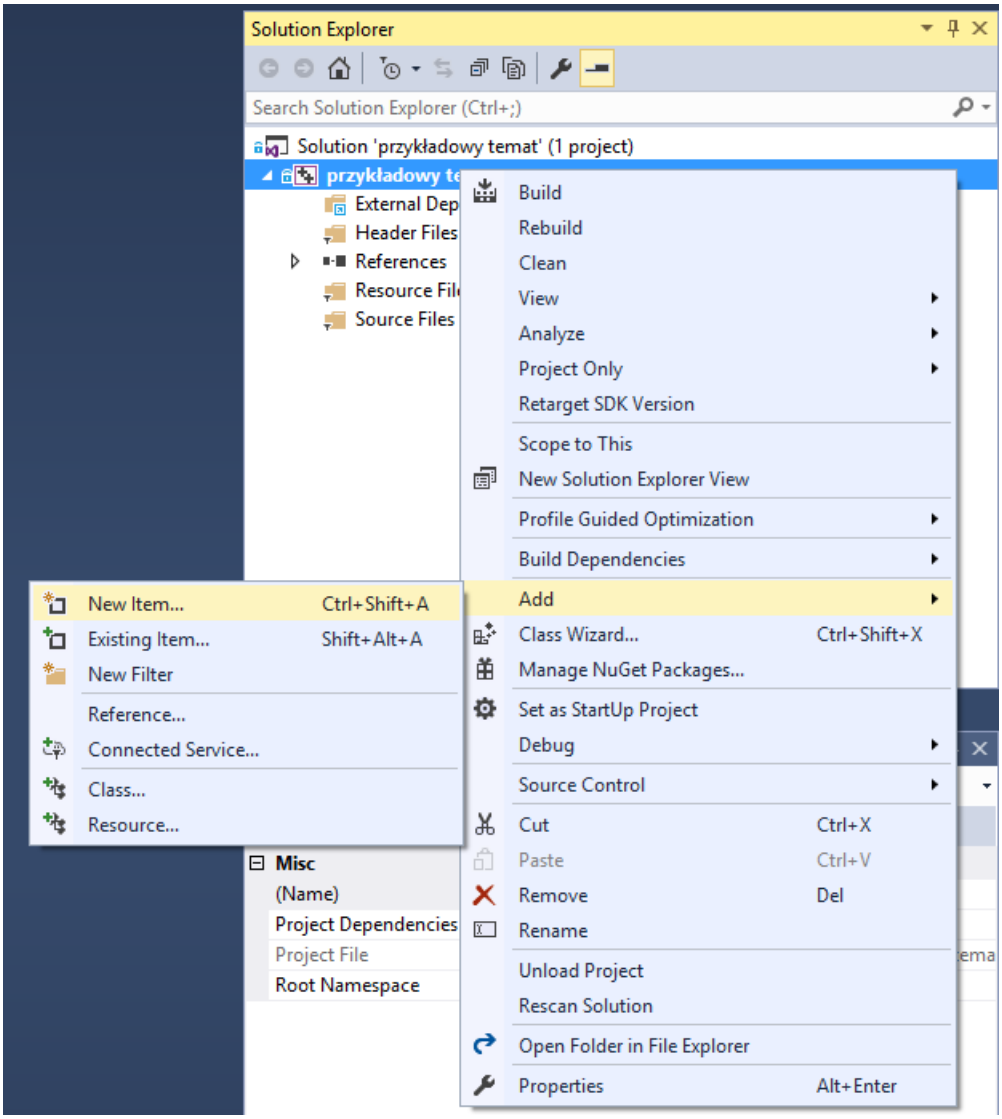
Rysunek 20: Konfiguracja projektu

7. kliknąć *Finish*.

Po wykonaniu powyższych kroków w oknie *Team Explorer* w sekcji *Solutions* powinien pojawić się utworzony projekt.

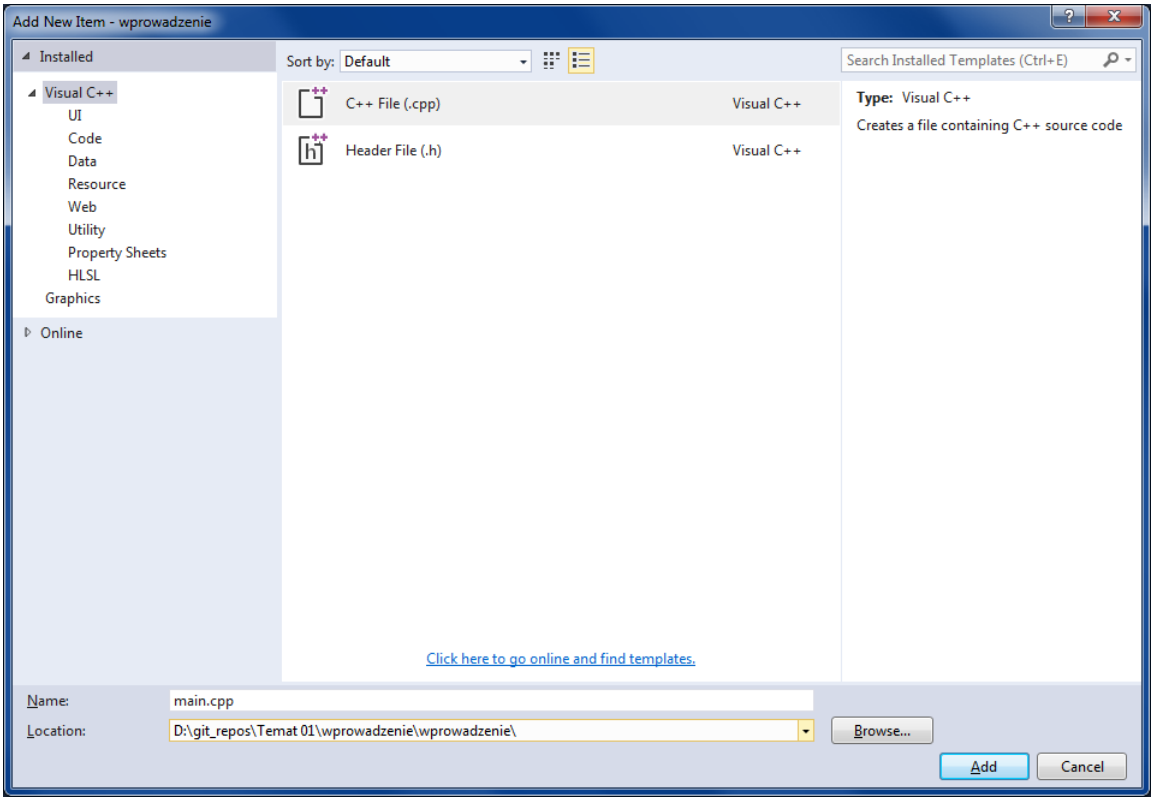
Otwieranie istniejącego projektu Visual Studio znajdującego się w repozytorium Projekty istniejące już w repozytorium są widoczne w oknie *Team Explorer* w zakładce *Home* w sekcji *Solutions*. Aktualnie otwarty projekt wyróżniony jest **pogrubieniem**. Aby otworzyć inny projekt należy kliknąć na niego dwukrotnie lewym przyciskiem myszy bądź kliknąć prawym i z menu wybrać *Open*

Dodawanie nowych plików do projektu Aby dodać nowy plik do projektu należy przejść do okna *Solution Explorer* (Menu View → Solution Explorer). Następnie kliknąć prawym przyciskiem myszy na nazwę projektu i z menu kontekstowego wybrac Add → New Item... (rys. 21).



Rysunek 21: Dodawanie nowego pliku

Możliwe jest dodanie różnych typów plików. Zaczniemy od dodania pliku z kodem źródłowym C++ (rys. 22).



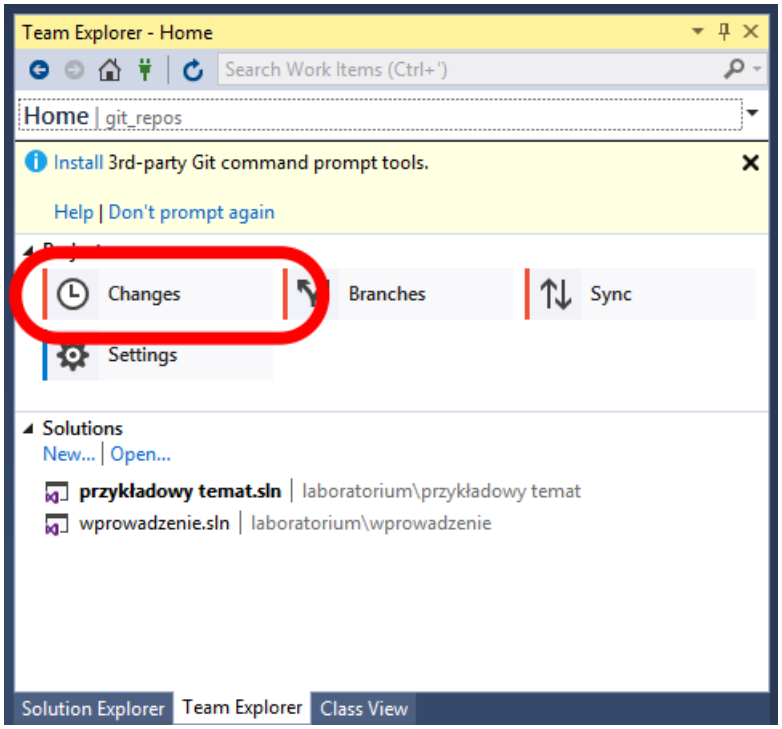
Rysunek 22: Dodawanie nowego pliku

Nazwa pliku powinna być zgodna z zaleceniami prowadzącego. W utworzonym pliku należy umieścić poniższy, przykładowy kod:

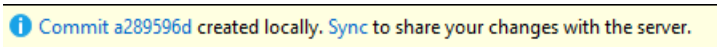
```
#include <iostream>

int main(int argc, char **argv)
{
    std::cout << "Hello World!";
    return 0;
}
```

Prowadzący może zarządzić wpisanie innego kodu.

Zapisywanie zmian w lokalnym repozytorium Przed zapisaniem zmian należy skonfigurować nazwę użytkownika. W tym celu należy przejść do *Team Explorer* → *Home* → *Settings*. Następnie należy wybrać *Global Settings*. Należy tutaj wprowadzić nazwę użytkownika i adres poczty elektronicznej autora zmian. W ogólnym przypadku podane dane mogą być właściwie dowolne, jednak na zajęciach należy podać nazwę zgodną z nazwą konta GitHub. Następnie należy kliknąć *Update*. Visual Studio automatycznie obserwuje zmiany w plikach oraz dodawane pliki źródłowe. W celu podejrzenia zmian należy przejść do *Team Explorer* → *Home* → *Changes* (rys. 23). W oknie, które się pojawi, można zapisać zmiany do lokalnego repozytorium. Lista zmian, które zostaną zapisane, widoczna jest w sekcji *Included Changes*. Aby zapisać zmiany należy wprowadzić ich opis a następnie kliknąć *Commit*. Jeśli nie wystąpił żaden błąd, powinien pojawić się komunikat podobny do przedstawionego na rys. 24. Kliknięcie *Sync* spowoduje przejście do okna synchronizacji repozytorium lokalnego i zdalnego.



Rysunek 23: Podgląd zmian

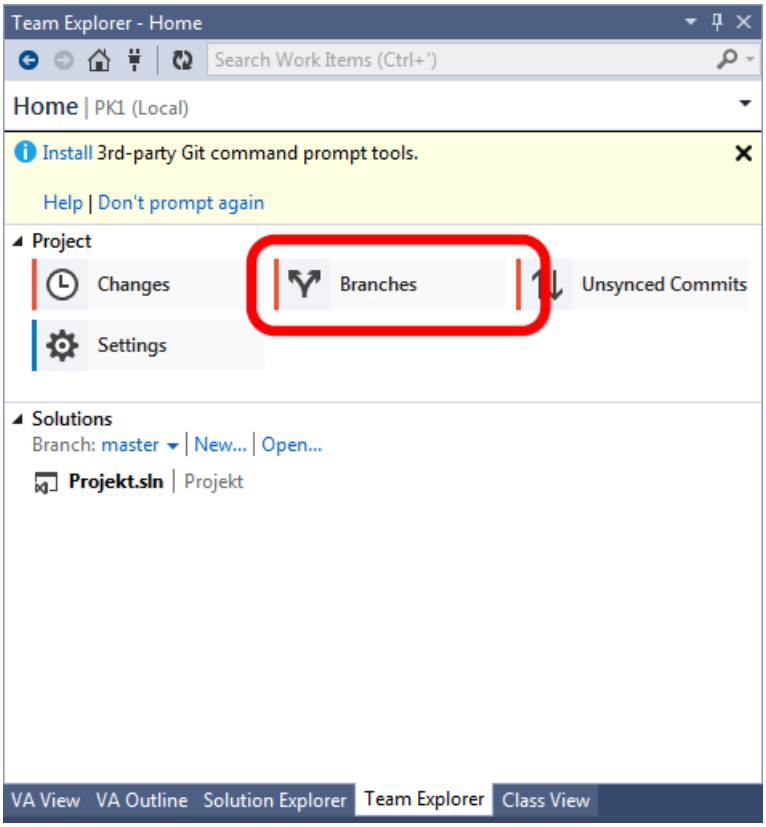


Rysunek 24: Commit – powodzenie

Synchronizacja lokalnego i zdalnego repozytorium Aby przenieść zmiany zapisane w lokalnym repozytorium do repozytorium zdalnego, należy przejść do *Team Explorer* → *Home* → *Sync*. W oknie, które się pojawi, będą widoczne różnice pomiędzy repozytorium lokalnym i zdalnym. W sekcji *Outgoing Commits* wylistowane są wszystkie operacje `commit` wykonane na lokalnym repozytorium od czasu ostatniego przesłania zmian do zdalnego repozytorium. Aby przesłać te zmiany do zdalnego repozytorium należy kliknąć *Push*. Po tej operacji może pojawić się okno z zapytaniem o nazwę użytkownika oraz hasło, należy wprowadzić tutaj dane konta GitHub. Jeżeli operacja zakończy się sukcesem, pojawi się komunikat: *Successfully pushed to origin/master*.

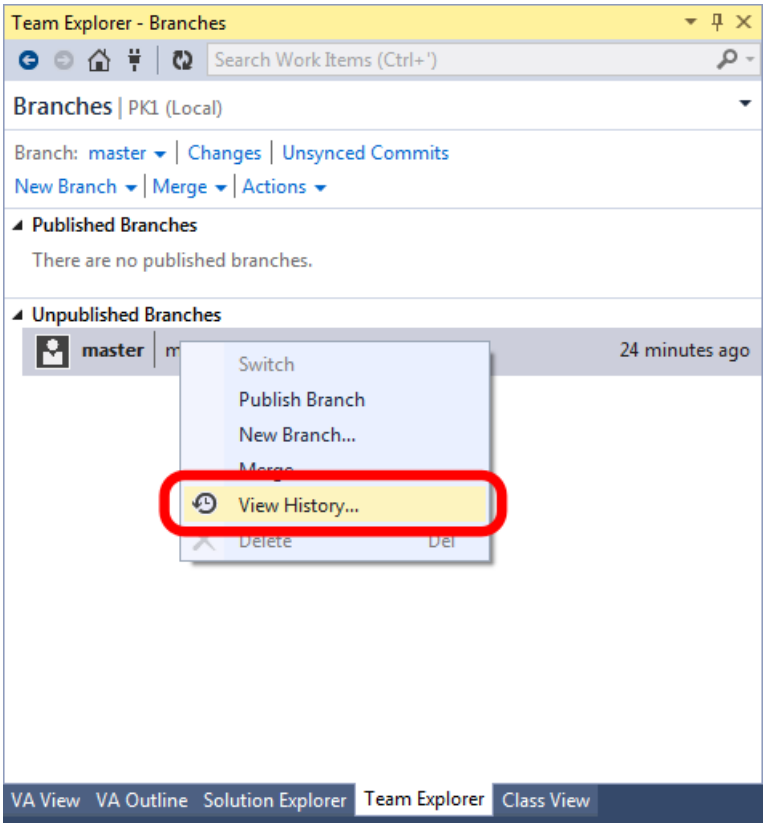
Przeglądanie historii zmian Do najważniejszych zalet systemów kontroli wersji jest przeglądanie historii zmian plików projektu. Systemy te pozwalają na porównanie różnych wersji plików umieszczonych w repozytorium w wyniku wykonania operacji `commit`.

W celu przejrzenia listy zmian w projekcie oraz w poszczególnych plikach należy w oknie Team Explorer wybrać opcję *Branches* (rys. 25).



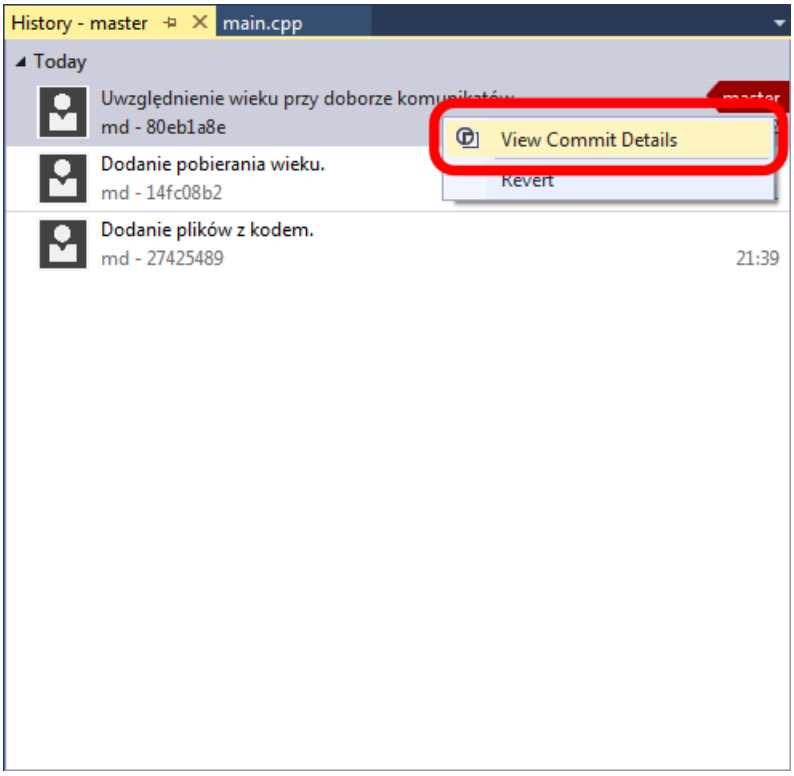
Rysunek 25: Wybór menu gałęzi

W otwartym oknie, w sekcji *Published Branches* lub *Unpublished Branches* (w zależności od tego, czy zmiany zostały już umieszczone w zdalnym repozytorium) znajduje się lista gałęzi projektu. Domyślną i jedyną gałęzią jest *master*. Z menu kontekstowego gałęzi należy wybrać opcję *View History* (rys. 26).



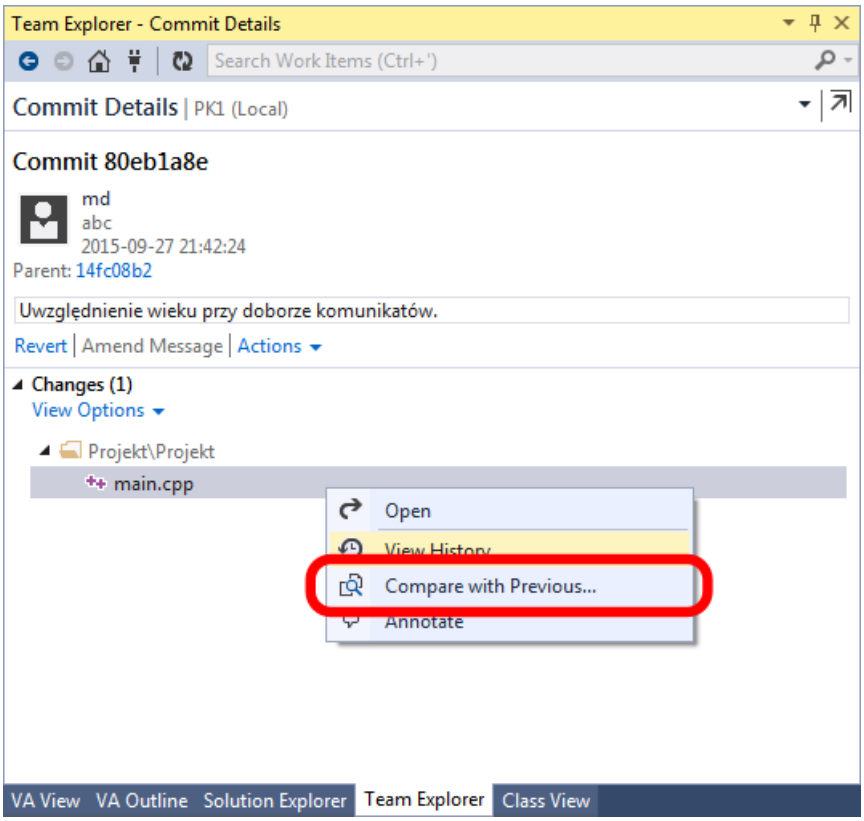
Rysunek 26: Menu kontekstowe gałęzi

Wyświetlone zostanie nowe okno zawierające listę przeprowadzonych zmian (*commitów*) w projekcie wraz z opisami oraz autorami zmian. W celu przejrzenia listy zmian obejmowanych przez dany `commit` należy z menu kontekstowego wybrać opcję *View Commit Details* (rys. 27).



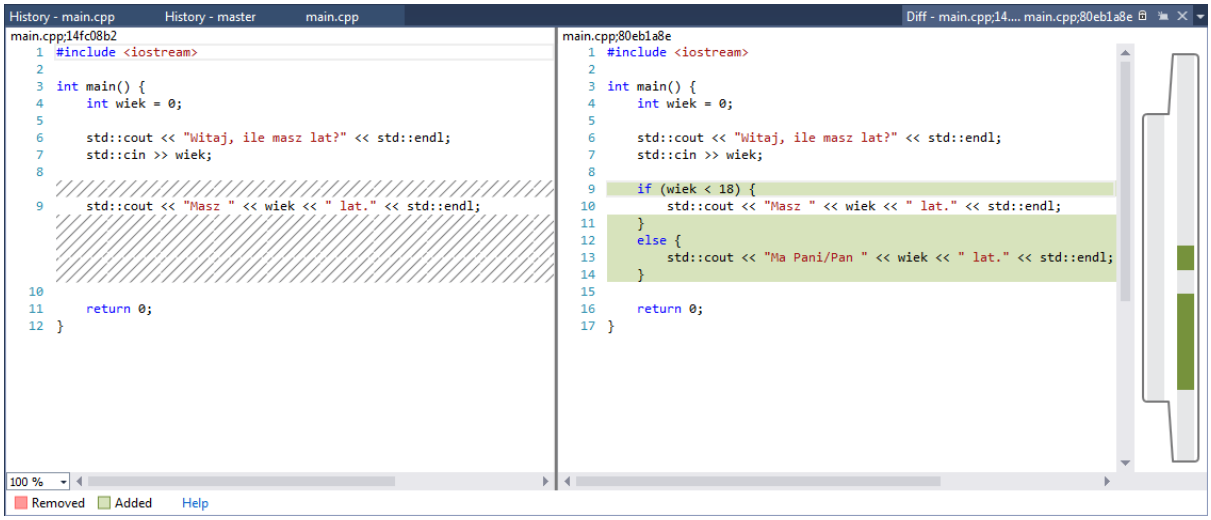
Rysunek 27: Menu kontekstowe *commitu*

W oknie Team Explorer zostanie wyświetlona lista wszystkich zmienionych plików. W celu przejrzenia zmian wprowadzonych w pliku, z jego menu należy wybrać opcję *Compare with Previous...* (rys. 28).



Rysunek 28: Opcja porównania plików

Zostanie wyświetlone porównanie dwóch wersji pliku (rys. 29).



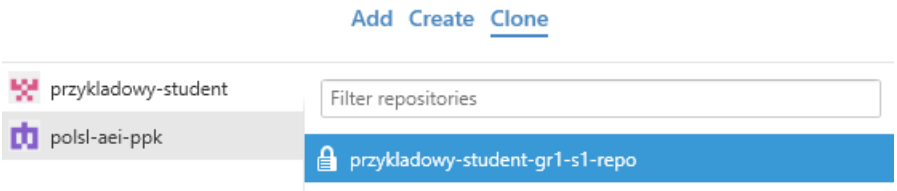
Rysunek 29: Porównanie plików

4.2.2 Obsługa klienta GitHub for Windows

Kolejnym klientem, ktory zostanie opisany jest GitHub for Windows. Jest od dostępny pod adresem `https://desktop.github.com/`.

Konfiguracja Po pierwszym uruchomieniu programu należy go skonfigurować postępując zgodnie z wyświetlanymi instrukcjami. Po tym program zaproponuje przeprowadzenie krótkiego wprowadzenia do jego używania. Wprowadzenie to nie będzie tu opisane, jednak warto się z nim zapoznać.

Klonowanie repozytorium Aby sklonować repozytorium należy kliknąć znak + w prawym górnym rogu, a następnie wybrać `clone`. Z menu należy wybrać odpowiednią organizację oraz repozytorium, nastepnie kliknąć przycisk `clone`. (rys. 30).

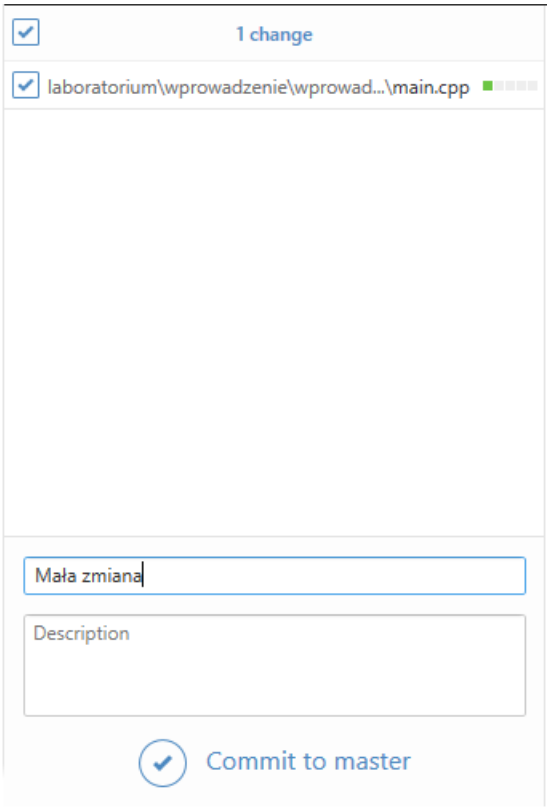


Rysunek 30: Klonowanie

Zapis zmian w lokalnym repozytorium GitHub for Windows automatycznie rejestruje zmiany wykonane na plikach. W przypadku zarejestrowania takiej zmiany w górnej części okna, w środkowej jego części, pojawia się stosowna informacja (rys. 31).

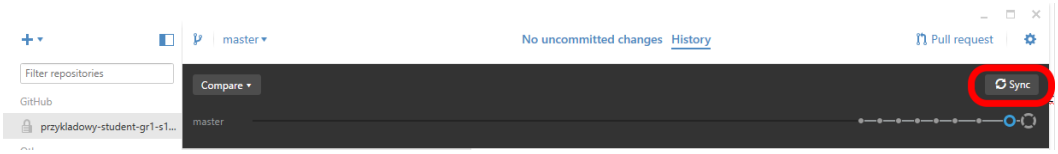
Rysunek 31: Zmiany

Po kliknięciu tekstu z tą informacją program umożliwi zapis zmian w lokalnym repozytorium. Po wprowadzeniu informacji na temat zmian można dokonać operacji `commit` (rys. 32).



Rysunek 32: Commit

Zapis zmian w zdalnym repozytorium Aby zapisać zmiany na zdalnym serwerze należy kliknąć *Sync* (rys. 33).



Rysunek 33: Przesłanie zmian do zdalnego repozytorium

4.2.3 Inne programy klienckie dla systemu Git

Dozwolone jest użycie innych niż opisane w niniejszej instrukcji programów klienckich dla systemu Git. Przykładowe programy alternatywne:

- TortoiseGIT: `https://tortoisegit.org/`
- Source Tree: `https://www.sourcetreeapp.com/`

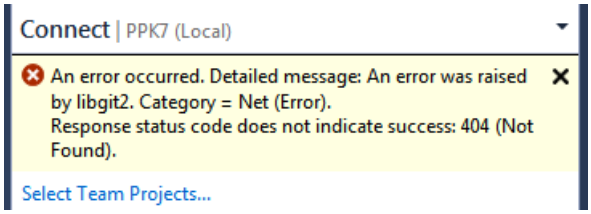
4.3. Typowe błędy i problemy

Do najczęstszych błędów i problemów pojawiających się podczas pracy z narzędziem Git na zajęciach należą:

- wykonywanie klonowania repozytorium w niewłaściwym miejscu (zwykle wewnątrz innego lokalnego repozytorium, często repozytorium innej osoby) – podczas klonowania należy zwrócić uwagę na ścieżkę lokalnego repozytorium (rys. 15);
- tworzenie projektu w niewłaściwym repozytorium (np. repozytorium innej osoby) – na początku zajęć należy wybrać swoje repozytorium lokalne z listy dostępnych (rys. 17) bądź wykonać klonowanie; podczas tworzenia projektu należy zwrócić uwagę na ścieżkę (rys. 19);

- klonowanie repozytorium (`clone`) w celu pobrania najnowszej wersji plików do istniejącego już repozytorium lokalnego – do tego służy komenda `pull`, komenda `clone` jest używana tylko raz dla utworzenia lokalnego repozytorium;
- niezachowywanie porządku w strukturze repozytorium – w momencie utworzenia przez prowadzącego zdalnego repozytorium zostaje zainicjowana także podstawowa struktura katalogów (np. katalogi *Temat 1*; *Projekt 1* i *Projekt 2*); struktury tej należy przestrzegać zwracając uwagę podczas tworzenia projektu (rys. 19) na podanie właściwej ścieżki z uwzględnieniem zarówno lokalizacji repozytorium, jak i odpowiedniego katalogu;
- nieregularne zamieszczanie zmian w repozytorium – dokładne zapamiętywanie historii pracy nad projektem należy do podstawowych celów wykorzystania systemu kontroli wersji; systematyczne zamieszczanie zmian z jednej strony ułatwia studentowi wgląd do wcześniejszych wersji oraz ich przywracanie, a z drugiej umożliwia prowadzącemu śledzenie postępu prac;
- stosowanie nic nie mówiących nazw projektów (np. *Project 1*) lub opisów zmian (np. *fsdfsfs* albo *Wgranie zmian*) – te elementy mają na celu ułatwienie pracy poprzez lepsze jej zorganizowanie, a nie wprowadzanie zamętu i zaśmiecanie repozytorium;
- przesyłanie do repozytorium plików binarnych (np. `*.zip`, `*.exe`, `*.pdf`) – plik taki jest traktowany jako całość, niemożliwe jest wskazanie zmienionych fragmentów, scalanie zmian itd.;
- zmiana nazw plików, katalogów, struktury katalogów komendami spoza systemu Git – zmian takich należy dokonywać komendami systemu kontroli wersji, inaczej może dojść do utraty powiązania pliku z jego historią i poprzednimi wersjami;
- przechowywanie kolejnych wersji w osobnych plikach lub katalogach – właśnie po to jest system kontroli wersji, żeby tego nie trzeba było robić.

Do problemów związanych z pracą z GITem należy zaliczyć także możliwość zgłoszenia przez klienta GITa błędu podczas klonowania jak na rys. 34. Jego przyczyną jest zazwyczaj zapisanie hasła innego użytkownika w systemowym magazynie poświadczeń. W celu rozwiązania problemu należy wejść do Panelu Sterowania, a następnie wybrać kolejno pozycje *Konta użytkowników* → *Zarządzaj poświadczeniami*, a następnie w sekcji *Poświadczenia rodzajowe* usunąć zapisane hasła logowania do serwisu GitHub.



Rysunek 34: Błąd klonowania