

Wersjonowanie i backup kodu

Systemy kontroli wersji kodu źródłowego

Systemy kontroli wersji (ang. *version control systems*, VCS) to oprogramowanie pozwalające na gromadzenie i przeglądanie kolejnych wersji wytwarzanego kodu źródłowego. Przez wiele lat najpopularniejszym systemem kontroli wersji był CVS, a potem zgodny z nim SVN (*Subversion*), lecz teraz bezapelacyjnie króluje stworzony przez Linusa Torvaldsa system *Git* (rysunek C.1). W odróżnieniu od wcześniejszych systemów, *Git* jest rozproszony – projekty przechowywane są nie tylko na centralnym serwerze, ale również na poszczególnych stanowiskach pracy. *Git* umożliwia natomiast ich synchronizację i scalanie. Symptomatyczne jest to, że w środowiskach programistycznych firmy Microsoft, w których przez wiele lat forsowany był system kontroli wersji i pracy zespołowej o nazwie *Team Foundation Server*, nigdy nie był jednak zbyt lubiany, obecnie wszystkie wbudowane w Visual Studio mechanizmy wersjonowania, a więc do kupionego przez Microsoft serwisu *GitHub*, do *Azure Repos* (część *Azure DevOps Services*, czyli chmury ze zbiorem usług wspierających programistów), a także do dowolnie skonfigurowanego serwera, opierają się na użyciu *Git*.



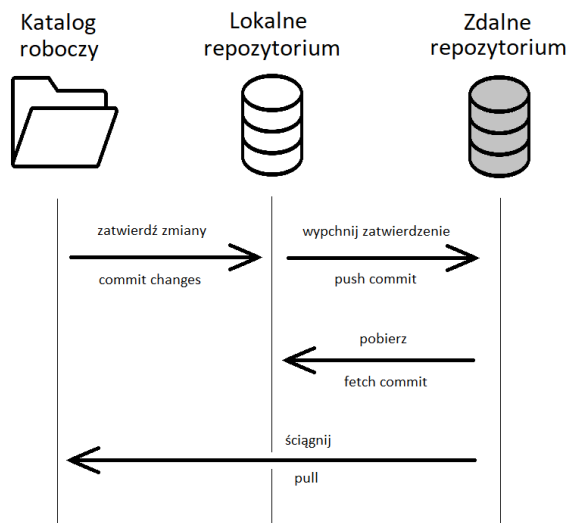
Rysunek C1. Logo systemu kontroli wersji Git

Git otaczany jest kultem i podobnie, jak np. TeX ma rzeszę swoich wyznawców i apologetów, którzy potrafią spierać się o to, czy lepszy jest *merge*, czy *rebase*. Korzystając z wbudowanego klienta *Git* w środowisku Visual Studio, możemy okazać się dla nich ludźmi niepoważnymi, którzy rezygnują z całej mocy tego narzędzia na rzecz wygody.

Użytkownicy *Git* używają specyficznej terminologii na określenie poszczególnych czynności, która w początkowej fazie może utrudniać zrozumienie tego, co naprawdę system robi. Warto ją zatem poznać od razu na początku, aby móc bez problemu komunikować się z systemem i z innymi jego użytkownikami. Przy okazji poznamy też podstawowe założenia tego systemu. Miejsca przechowywania kolejnych wersji projektu nazywają się **repozytoriami**. Są dwa repozytoria – lokalne i zdalne. Ponieważ „repozytorium” to długie słowo, często jest skracane do „repo”, choć brzmi to trochę pretensjonalnie. W efekcie pliki przechowywane są w trzech miejscach: 1) w katalogu projektu (tzw. katalogu roboczym, ang. *working directory*), gdzie edytujemy je za pomocą edytora Visual Studio, 2) w lokalnym repozytorium, które ulokowane jest w podkatalogu *.git* katalogu projektu (używamy Visual Studio 2019) oraz 3) w zdalnym repozytorium przechowywanym na serwerze np. oferowanym przez jedną z usług, jak *GitHub*, *GitLab*, *Bitbucket*, *Azure DevOps* lub wiele innych. Wszelkie zmiany plików projektu są śledzone (jak zobaczymy, zmiany plików są sygnalizowane dodatkowymi ikonkami w *Eksploratorze rozwiązań*) - innymi słowy prowadzony jest indeks zmian. Zmiany takie co pewien czas należy zaakceptować, co powoduje zebranie ich w jedno tzw. zatwierdzenie¹, które jednak częściej nazywane jest z angielskiego commitem. Słowo *commit* używane jest też w jako czasownik i oznacza przesłanie zmienionego

¹ „Zatwierdzenie” i inne polskie tłumaczenia to nazwy używane w Visual Studio 2019.

projektu do lokalnego repozytorium. W polskiej terminologii Visual Studio proces ten nazywany jest zatwierdzaniem zmian. Commity (zatwierdzenia) można następnie przesłać do zdalnego repozytorium – to po polsku nazywa się **wypychaniem** zatwierdzeń, a po angielsku *commit push*. Przed wypchnięciem do zdalnego repozytorium warto sprawdzić, czy nie ma w nim nowszych wersji projektu. Ten proces po angielsku nazywa się *fetch*, a po polsku pobierz. Oznacza *de facto* porównywanie zawartości lokalnego i zdalnego repozytorium. Komplet czynności – *fetch* i *push* – to synchronizacja repozytoriów. Każdy członek zespołu może oczywiście uzyskać najnowszą wersję projektu ze zdalnego repozytorium do swojego katalogu roboczego, co się nazywa po angielsku *pull*, a po polsku ściąganiem. Warto zwrócić uwagę, że dwa synonimiczne w potocznym języku określenia – ściąganie i pobieranie – tu są nazwami różnych, choć podobnych czynności. Ostatnim terminem jest klonowanie projektu – to jest pobieranie całego projektu ze zdalnego repozytorium, ze wszystkimi jego gałęziami i wersjami, do komputera, gdzie wcześniej żadnej wersji tego projektu nie było. Zdaję sobie sprawę, że to bardzo dużo nowych pojęć i na początku trudno się w nich zorientować. Mam nadzieję, że trochę uporządkuje je rysunek C.1 przedstawiający schemat podstawowych procesów *Git*.



Rysunek C.1. Procesy *Git*. Na rysunku nie uwzględniono indeksu rejestrującego pliki zmienione w katalogu roboczym

Zwróćmy uwagę na trzy zasadnicze zalety korzystania z systemu kontroli wersji. Podstawową zaletą jest przede wszystkim regularne tworzenie kopii zapasowej kodu, i to zarówno na stanowisku programisty (lokalnie), jak i na zdalnym serwerze. Drugą zaletą jest wersjonowanie. Dzięki niemu mamy kopię nie tylko ostatniej wersji projektu, ale również wszystkich poprzednich, co pozwala szybko wrócić do wcześniejszej wersji (całkowicie lub tylko w wybranych plikach). Wersjonowanie wiąże się z trzecią podstawową zaletą – każda zmiana w zdalnym repozytorium jest podpisana przez któregoś członka zespołu. To umożliwia zarządzanie projektem, którego rozwój można śledzić bez potrzeby chodzenia od stanowiska do stanowiska, ale także zarządzanie realizującym projekt zespołem, które oparte jest na wiarygodnych danych o jego efektywności.




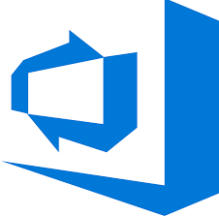
Serwisy *Git*

Git to aplikacja konsolowa, której można użyć do postawienie własnego serwera kontroli wersji. Jest jednak kilka publicznie dostępnych usług sieciowych, które umożliwiają utworzenie zdalnego repozytorium bez potrzeby samodzielnej konfiguracji systemu *Git*. Wszystkie one udostępniają również dostępny przez przeglądarkę interfejs, który pozwala na przeglądanie zawartości repozytoriów oraz wiele dodatkowych funkcjonalności, jak na przykład ustalanie uprawnień dostępu do poszczególnych gałęzi.

Najbardziej popularne usługi tego typu to *GitHub*, *GitLab*, *Bitbucket* i *Azure DevOps Services*, czyli dawne *Visual Studio Team Foundation Services* (tabela C.1). Właściciel *Bitbucket*, firma *Atlassian*, oferuje dodatkowo popularny klient *Git* z graficznym interfejsem użytkownika o nazwie *SourceTree*, który współpracuje nie tylko z usługą *Bitbucket*, ale również z pozostałymi usługami *Git*. *SourceTree* to program, który jest często używany, obok klienta wbudowanego w nowe wersje *Visual Studio*. Często „zwykłe” czynności, jak akceptowanie i wypychanie commitów wykonywane są w *Visual Studio*, a do bardziej złożonych lub do rozwiązywania problemów sięga się po *SourceTree*.

Poniżej użyję usługi *GitLab*, choć równie dobrze mógłby to być *Bitbucket* lub *GitHub*. Wszystkie one pozwalają obecnie na przechowywanie prywatnych projektów za darmo. Co ważne, po skonfigurowaniu połączenia ze zdalnym repozytorium, wybór konkretnej usługi nie ma większego znaczenia. Ich obsługa, dzięki standardowi *Git*, jest w Visual Studio ujednolicona.

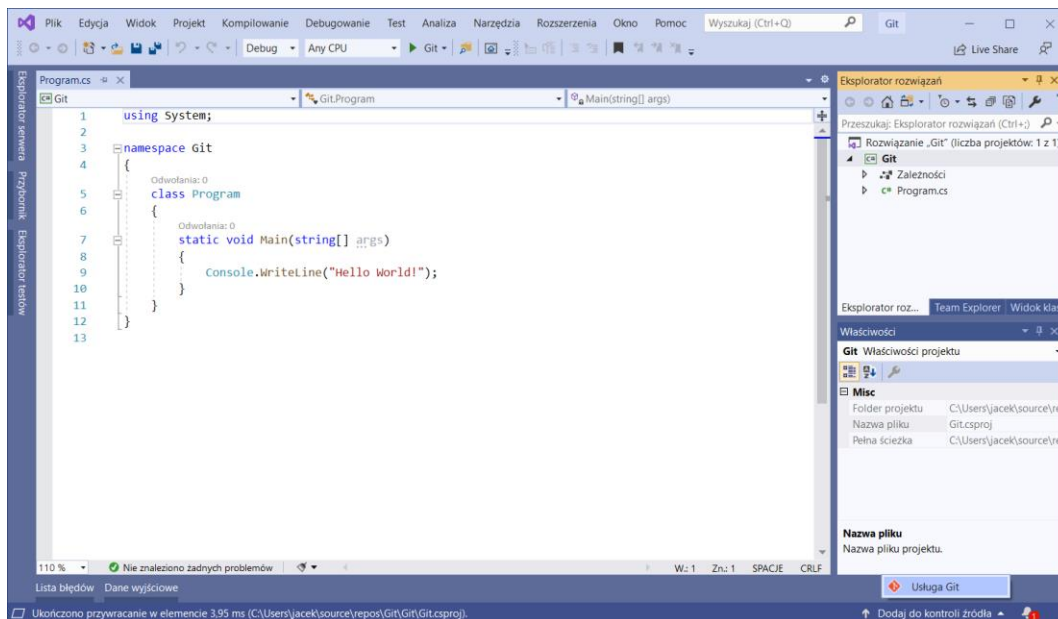
Tabela C.1. Porównanie darmowych usług oferowanych przez popularne serwisy Git

| Nazwa usługi | GitHub | GitLab (Community Edition) | Bitbucket | Azure DevOps Services (Azure Repos) |
|--|--|---|--|---|
| Nazwa firmy | Microsoft (7,5 mld \$) | GitLab Inc. | Atlassian | Microsoft |
| Logo |  |  |  |  |
| Dostępny od | 2008 | 2011 | 2008 | 2012 |
| Dostępność | Publiczne i prywatne projekty są darmowe (te ostatnie od niedawna, po przejęciu przez Microsoft) | Prywatne są darmowe. Możliwe jest nadawanie uprawnień użytkownikom. | Darmowy do 5 użytkowników (są też płatne plany) | Prywatne bezpłatne repozytoria |
| Popularność (stan z 2019) | 32 mln osób | 100 tys. osób | 5 mln osób | brak informacji |
| Maksymalna liczba członków zespołu w wersji darmowej prywatnego repozytorium | nieograniczona (od marca 2020 w ramach planu GitHub Free) | nieograniczona | 5 | 5, chyba że jest to projekt OpenSource |
| Maksymalny rozmiar darmowego | 500MB w jednym repozytorium (nieograniczona liczba repozytoriów) | 1GB | 1GB | |
| Charakterystyka | Tradycyjne miejsce przechowywania publicznych repozytoriów oprogramowania OpenSource. Tablica Kanban dla projektu. Płatne są dodatkowe funkcjonalności. Udostępnia API. | Dla projektów OpenSource, ale również dla prywatnych projektów. Możliwe jest pobranie oprogramowania i postawienie własnej usługi GitLab. Mniejsza społeczność niż GitHub. Udostępnia API. | Udostępnia REST API do budowy aplikacji korzystających z repozytorium. | |

Tworzenie projektu

Stworzymy nowy projekt aplikacji dla platformy .NET Core. Aplikacja nie będzie miała żadnej sensownej funkcjonalności, więc nazwijmy ją po prostu *Git*. Po utworzeniu projektu, na pasku stanu w prawym dolnym rogu okna Visual Studio (pod oknem *Właściwości*) widoczny jest przycisk, a raczej rozwijana lista z etykietą *Dodaj do kontroli źródła*². Po jej rozwinięciu pojawia się tylko jedna pozycja *Usługa Git* (rysunek C.2). Kliknijmy ją.

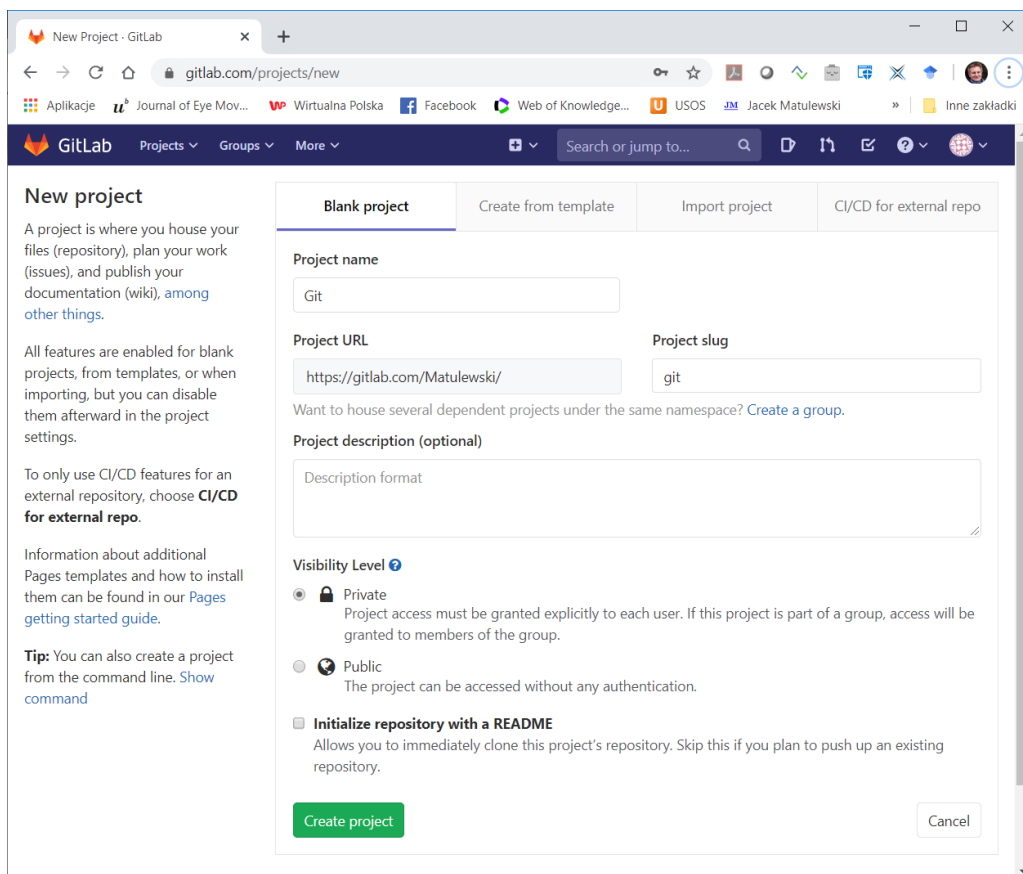
² Powinno być chyba „Dodaj do kontroli kodu źródłowego”.



Rysunek C.2. Poleceń dodawania kontroli wersji Git nie trzeba szukać w menu, jest dostępna na pasku stanu

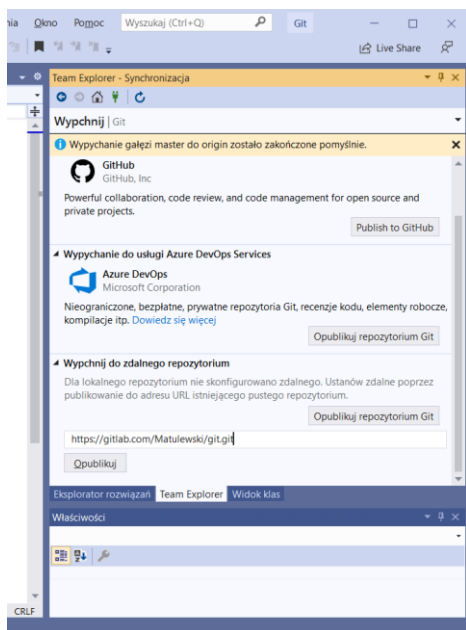
To spowoduje dwie rzeczy. Po pierwsze do katalogu projektu dodany zostanie podkatalog `.git` z rozbudowaną strukturą podkatalogów. To lokalne repozytorium projektu, w którym będą trzymane wszystkie wersje projektu. Po drugie w oknie Visual Studio widoczne stanie się podokno *Team Explorer*, w którym widoczna będzie strona *Synchronizacja*. Na niej zaprezentowane są trzy możliwości: publikacji projektu w *GitHub*, umieszczenie go w usłudze *Azure DevOps Services* albo w innym zdalnym repozytorium *Git*. Ponieważ postanowiłem wykorzystać serwis *GitLab*, wybierzemy tę trzecią możliwość.

Pozostawmy na chwilę Visual Studio i uruchommy przeglądarkę, w której wczytajmy stronę *gitlab.com*. Załóżmy konto w serwisie *GitLab* (przycisk *Register*), a po zalogowaniu kliknijmy dobrze widoczny zielony przycisk z etykietą *New project* (jest na niemal każdej podstronie). Zostaniemy przeniesieni na stronę *New project* (rysunek C.3), na której należy podać nazwę projektu (ja nazwałem go także *Git*), ewentualny opis i zaznaczyć, czy chcemy żeby projekt był prywatny, czy publiczny. Tworzenie projektu z podanymi ustawieniami kończymy klikając przycisk *Create Project*. Po utworzeniu projektu zostaniemy przeniesieni do strony, na której opisane są podstawowe instrukcje *Git*. Skopiujemy z nich adres URL zdalnego repozytorium. W moim przypadku jest to <https://gitlab.com/Matulewski/git.git>.



Rysunek C.3. Tworzenie pustego projektu w usłudze GitLab

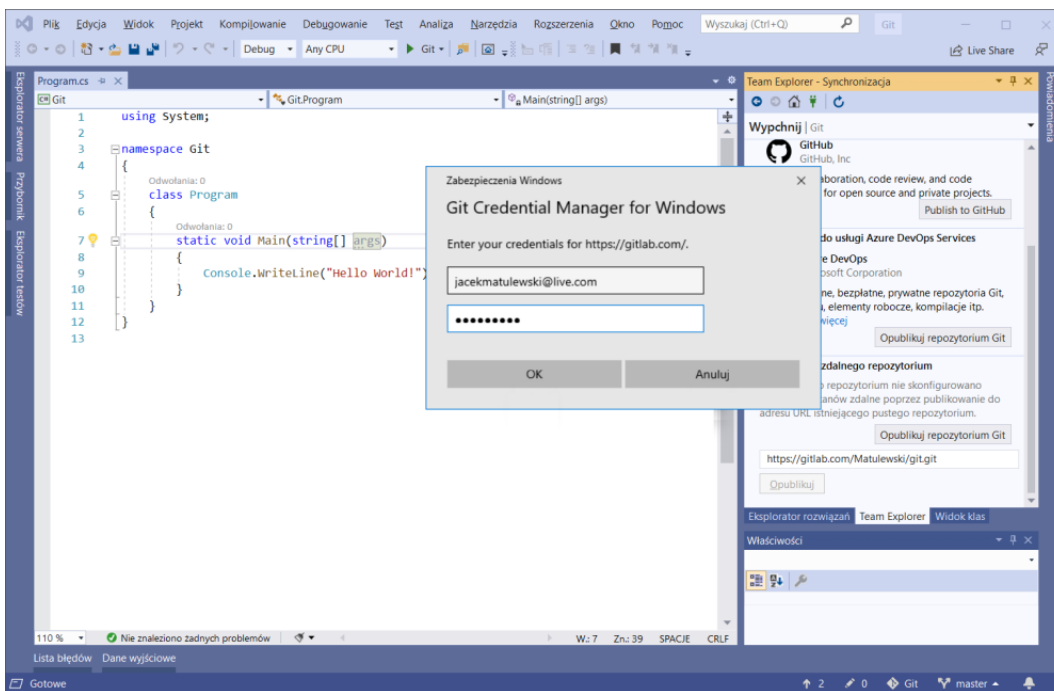
Teraz możemy wrócić do Visual Studio, aby skonfigurować połączenie z utworzonym przed chwilą zdalnym repozytorium. W podoknie *Team Explorer*, na widocznej w nim stronie *Synchronizacja* kliknijmy przycisk *Opublikuj repozytorium Git* (w części *Wypchnij do zdalnego repozytorium*). Pojawi się pole tekstowe, do którego należy wkleić uzyskany z *GitLab* adresu URL (rysunek C.4). Zatwierdźmy połączenie klikając przycisk *Opublikuj*.



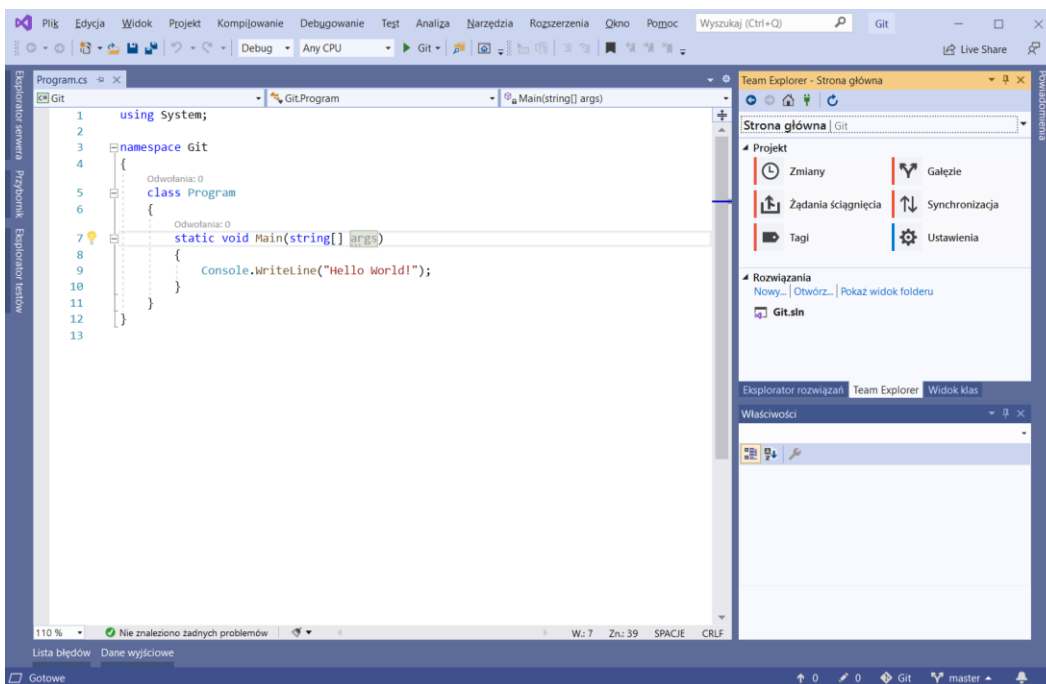
Rysunek C.4. Wstępna konfiguracja zdalnego repozytorium ogranicza się do podania jego adresu URL

Przy pierwszym połączeniu zostaniemy poproszeni o podanie loginu (adresu e-mail) i hasła do konta w usłudze *GitLab* (rysunek C.5). Potem ustawienia te zostaną zapamiętane i nie musimy ich ponownie podawać. Po

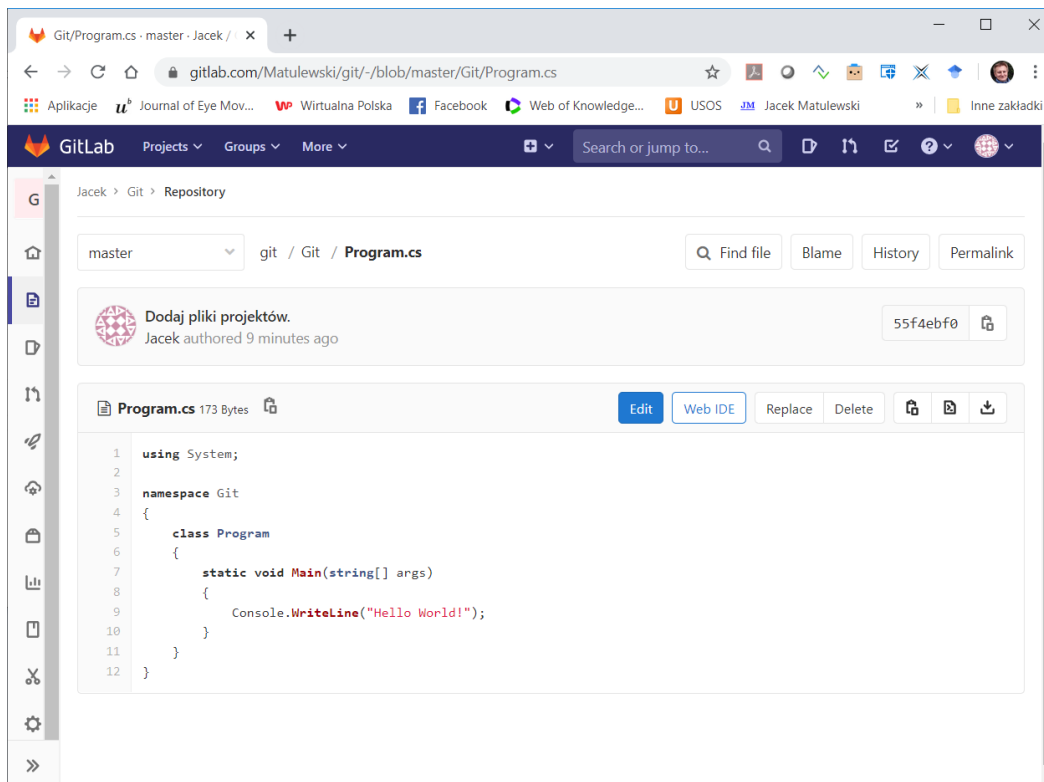
połączeniu do zdalnego repozytorium, projekt zostanie od razu do niego wypchnięty, co możemy sprawdzić na stronie *GitLab* (rysunek C.7), a my zostaniemy przeniesieni do strony głównej podokna *Team Explorer* (rysunek C.6).



Rysunek C.5. Autoryzacja w usłudze GitLab



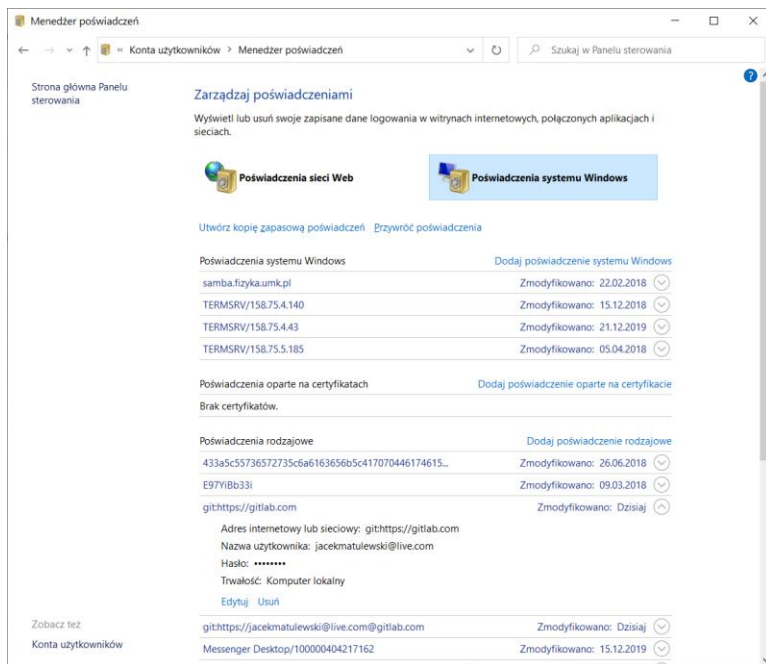
Rysunek C.6. W oknie Team Explorer widoczna jest strona główna. My będziemy jednak częściej korzystać z ikon widocznych na pasku stanu



Rysunek C.7. Po połączeniu ze zdalnym repozytorium w usłudze GitLab, natychmiast przesyłane są do niej aktualne pliki projektu

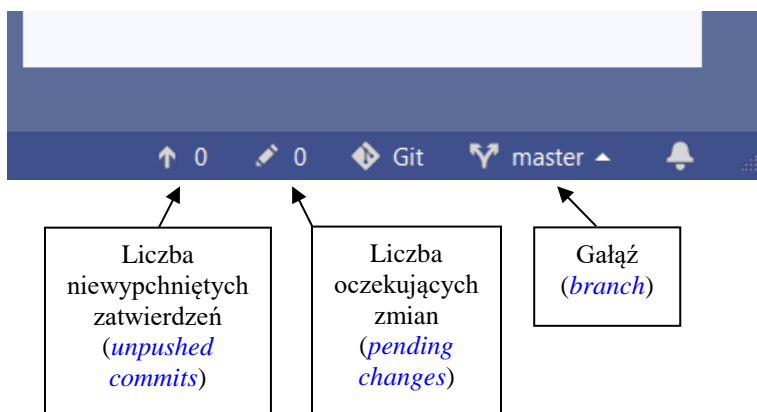
Jednak, jeżeli błędnie wpiszę hasło, pojawi się problem. Poświadczenia nie są bowiem przechowywane w Visual Studio i nie można ich z Visual Studio zmienić, zatem kolejne próby łączenia z serwerem będą bezowocne – nie pojawi się okno z prośbą o login i hasło (rysunek C.5). Aby się pojawiło, należy wpierw usunąć zapisane poświadczenia, co jest możliwe po przejściu do *Menadżera poświadczeń* dostępnego w *Panelu sterowania*³. W menadżerze należy odnaleźć pozycję z protokołem git i stroną *gitlab.com*. W moim przypadku było to *git:http://gitlab.com* oraz *git:http://jacekmatulewski@live.com@gitlab.com*. Obydwa poświadczenia usunąłem, dzięki czemu przy próbie ponownego połączenia ze zdalnym repozytorium w Visual Studio miałem możliwość ponownego wpisania hasła.

³ Także w Windows 10. W marcu 2020 menadżer ten nie został jeszcze przeniesiony do nowych ustawień.



Rysunek C.8. Hasła do usługi GitLab trzymane są w systemie Windows, a nie w ustawieniach Visual Studio

Na stronie głównej widocznej teraz w podoknie *Team Explorer* dostępne są podstawowe czynności, jakie możemy wykonać z repozytorium. Dzięki nim nie ma konieczności korzystania z poleceń linii komend systemu *Git*. My jednak rzadko będziemy zaczynać od tej strony. Wygodniejsze jest korzystanie z czterech ikon widocznych na pasku stanu Visual Studio (rysunek C.9). Przejrzyjmy je od prawej strony. Pomijamy dzwonek, który z wersjonowaniem kodu nie ma nic wspólnego. Następną jest ikona z etykietą *master*. „Master” to nazwa gałęzi głównej (na razie jedynej), w której aktualnie pracujemy. Gałęziami zajmiemy się już za chwilę. Następną ikonę możemy zignorować – pozwala na zmianę lokalnego repozytorium *Git*, czyli w praktyce zmianę projektu. Nie będziemy w tym tutorialu korzystać z tej możliwości. Kolejna ikona z ołówkiem i liczbą zero, to liczba zmian wprowadzonych w projekcie (liczba zmienionych plików) od ostatniego umieszczenia projektu w lokalnym repozytorium. Taka zebrana i nazwana zmiana projektu po zaakceptowaniu stanie się *commit* (w polskim tłumaczeniu - zatwierdzeniem). Liczba *commit*ów, które zostały już przesłane do lokalnego repozytorium (tego w podkatalogu *.git* projektu), ale jeszcze nie wysłane do zdalnego repozytorium (na stronie *GitLab*) widoczna jest przy ostatniej ikonice (ze strzałką w górę). To ikona niewypchniętych zatwierdzeń.



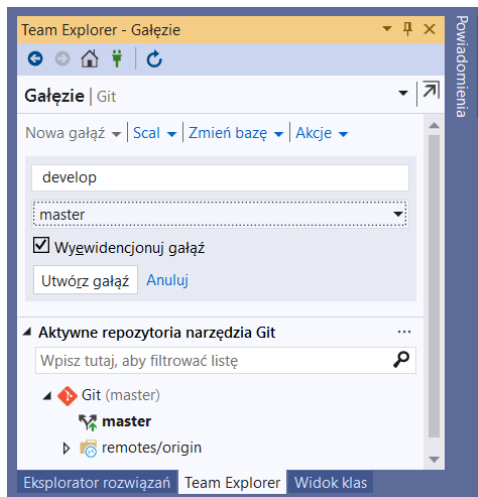
Rysunek C.9. Ikony kontroli Git na pasku stanu Visual Studio

Tworzenie nowej gałęzi

Zanim wprowadzimy w projekcie jakieś zmiany, stwórzmy nową gałąź o nazwie *develop*. Gałąź *master* powinna zawsze zawierać projekt, który jest stabilny, gotowy do kompilacji i prezentacji np. szefowi lub klientowi. Zmiany powinno się wprowadzać na osobnej gałęzi, często nazywanej właśnie *develop* i po zakończeniu cyklu

implementacji nowej funkcjonalności rozwijanego przez nas oprogramowania scalane z wątkiem głównym przez osobę uprawnioną do zatwierdzania zmian w gałęzi *master*.

Aby dodać nową gałąź, kliknijmy ikonę gałęzi na pasku stanu (z etykietą *master*) i wybierzmy polecenie *Nowa gałąź...*. W podoknie *Team Explorer* pojawi się wówczas strona o nazwie *Gałęzie* (rysunek C.10), w której możemy wprowadzić nazwę nowej gałęzi, czyli np. *develop*. Pod wprowadzoną nazwą nowej gałęzi należy w rozwijanej liście pozostawić wybraną gałąź *master* – to gałąź, od której nowa gałąź będzie odgałęziona. To oznacza, że po rozgałęzieniu w gałęzi *develop* znajdzie się kod projektu w stanie, w jakim jest w gałęzi *master*. Utworzenie gałęzi potwierdzamy, klikając przycisk *Utwórz gałąź*. Po jej utworzeniu, staje się ona automatycznie bieżącą gałęzią, co poznamy po zmianie nazwy w ikoncie gałęzi na pasku stanu. Zamiast *master*, zobaczymy tam teraz *develop*.



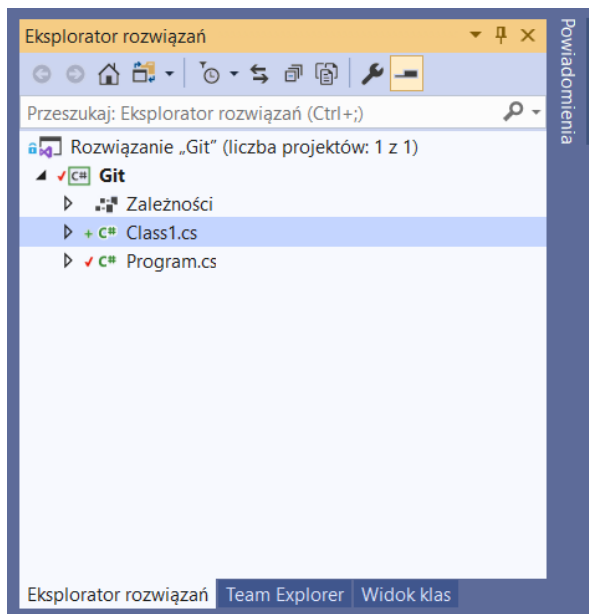
Rysunek C.10. Strona pozwalająca na zarządzanie gałęziami w repozytorium

Wprowadzanie zmian w projekcie

Pomimo podłączenia projektu do kontroli wersji, edycja kodu w żaden sposób się nie zmienia. Wszystkie modyfikacje oznaczone są jednak w podoknie *Eksplorator rozwiązań*, do którego warto teraz przejść. Dla przykładu do metody *Main* z pliku *Program.cs* dodajmy polecenie

```
Console.WriteLine("Git");
```

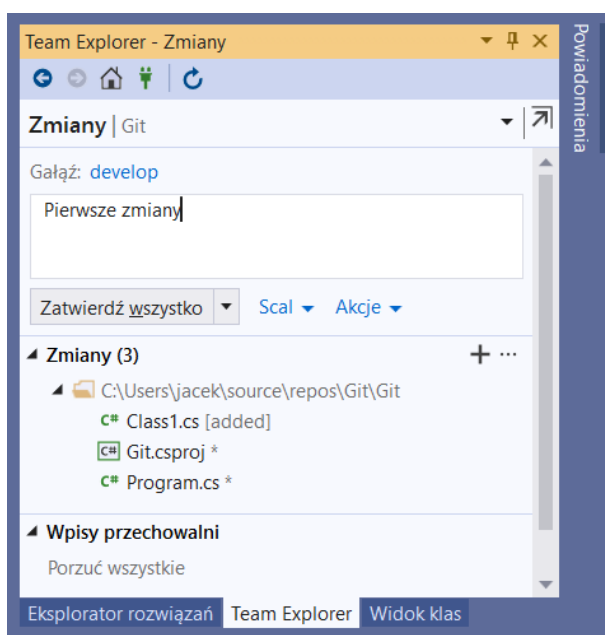
W *Eksploratorze rozwiązań* koło pliku *Program.cs* natychmiast pojawi się czerwony „ptaszek” (rysunek C.11). Jednocześnie liczba zmian widoczna na pasku stanu zwiększy się do 1. Nie poprzestańmy na tym i do projektu dodajmy nowy plik klasy. Możemy pozostawić jego domyślną nazwę tj. *Class1.cs*. Zwróćmy uwagę, że w *Eksploratorze rozwiązań* przy tym pliku pojawi się ikona + oznaczająca nowy plik projektu. Ale nie tylko, przy nazwie projektu (odpowiada jej plik projektu z rozszerzeniem *.csproj*) również pojawi się czerwony znaczek, sygnalizujący że ten plik też został zmieniony. Mamy więc już trzy zmiany (rysunek C.11), co widać na pasku stanu przy ikoncie ołówka.



Rysunek C.11. W Eksploratorze rozwiązań widoczne są symbole oznaczające zmiany wprowadzone w plikach projektu

Zatwierdzanie zmian

Kliknijmy ikonę zmian na pasku stanu – ikonę ołówka z liczbą 3. W efekcie w oknie *Team Explorer* zobaczymy stronę *Zmiany*. Na niej widoczna jest lista zmienionych plików. W naszym przypadku powinny to być pliki: *Class1.cs*, *Git.csproj* i *Program.cs* (rysunek C.12). W górnej części okna widoczne jest pole tekstowe, w którym należy obowiązkowo opisać, a przynajmniej zatytułować zatwierdzane zmiany. Ja tam wpisałem „Pierwsze zmiany”, ale gdybym był kierownikiem projektu, zirytowałbym się na taki nic nie mówiący opis. Opis powinien jasno zdawać sprawę z tego, co kryje się w zmianach danego commitu, żeby łatwo było zidentyfikować taki commit na długiej liście, która powstanie w miarę rozwoju projektu. Następnie kliknijmy przycisk *Zatwierdź wszystko*. Jeżeli wcześniej nie zapisaliśmy zmian w plikach na dysku, to teraz zostaniemy o to poproszeni. Po zatwierdzeniu zmian licznik przy ikonie ołówka na pasku stanu powinien zostać wyzerowany, a w zamian pojawił się jedynek w liczbie niewypchniętych commitów (zatwierdzeń). To oznacza, że zmiany znalazły się w lokalnym repozytorium *Git*, ale jeszcze nie zostały wypchnięte na serwer do zdalnego repozytorium.

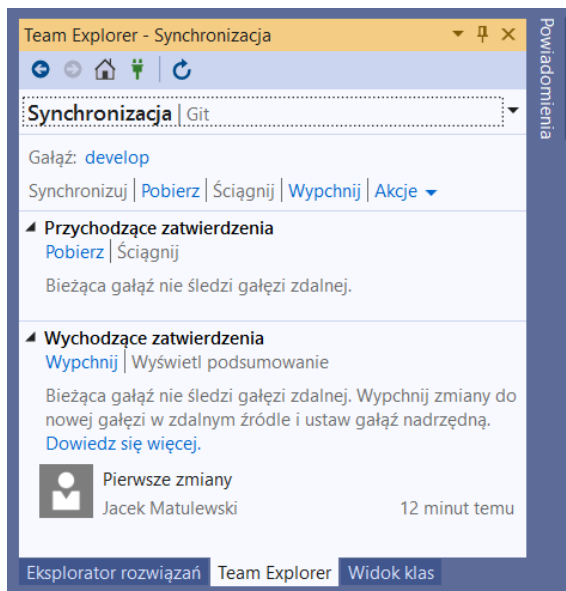


Rysunek C.12. Opis zatwierdzonych zmian powinien zdawać sprawę z wprowadzonych zmian w kodzie lub naprawionych błędów

Wypychanie do repozytorium

Jak wspomniałem, projekt widoczny na stronie *GitLab* nie został jeszcze zmieniony. Zmiany zostały jedynie umieszczone w podkatalogu `.git` projektu, czyli w lokalnym repozytorium. Takich zmian nie powinniśmy jednak nadmiernie długo trzymać lokalnie. To jak długo możemy trzymać takie zmiany lokalnie, zależy od tego czy nad implementowaną funkcjonalnością pracujemy sami, czy z innymi członkami zespołu oraz czy mamy własną gałąź, w której wprowadzamy zmiany, czy jest ona zmieniana także przez inne osoby. Jednak jeżeli celem korzystania z *Git* jest nie tylko wersjonowanie, ale również tworzenie kopii zapasowej kodu, przynajmniej raz dziennie, najlepiej pod koniec dnia, zmiany powinny być wypchnięte do zdalnego repozytorium, tak żeby kolejny dzień zacząć z dwoma zerami na pasku stanu, a kierownik projektu mógł obserwować jego postęp.

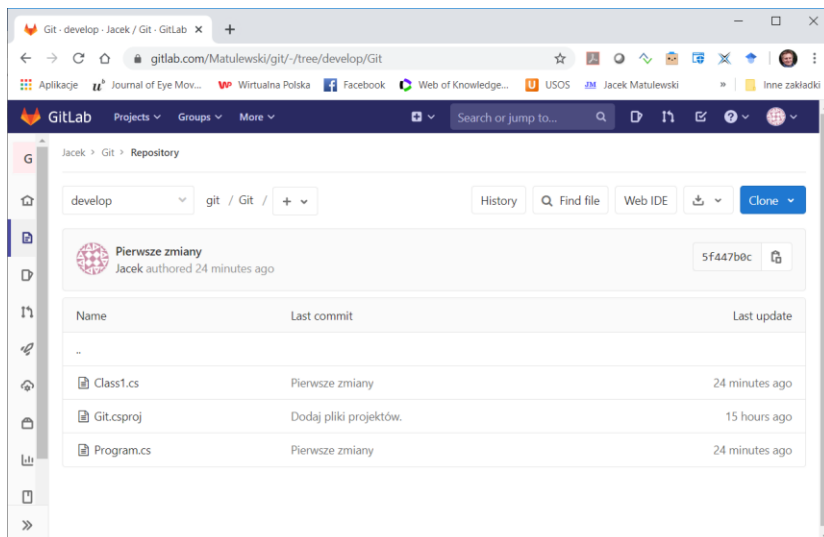
Kliknijmy ikonę niewypchniętych zatwierdzeń (commitów), przy której widoczna jest teraz liczba 1. W efekcie w podoknie *Team Explorer* pojawi się strona *Synchronizacja* (rysunek C.13). W tym oknie w przyszłości będziemy także pobierać zmiany ze zdalnego repozytorium, ale na razie chcemy je na serwer wypchnąć. Należy wobec tego kliknąć *Wypchnij*, co jeżeli nie będzie problemów z połączeniem powinno po chwili zaowocować pojawieniem się komunikatu „Wypychanie gałęzi develop do origin zostało zakończone pomyślnie.”. *Origin* to domyślna nazwa dla zdalnego repozytorium.



Rysunek C.13. Strona synchronizacji lokalnego i zdalnego repozytorium

Aby sprawdzić, czy rzeczywiście tak się stało, możemy przełączyć się do okna przeglądarki z wczytaną stroną *GitLab*. W górnej części strony zmienimy gałąź z *master* na *develop*. Wówczas powinniśmy zobaczyć wypchnięte przed chwilą zmiany w kodzie, co rozpoznamy po obecności pliku *Class1.cs* (rysunek C.14).

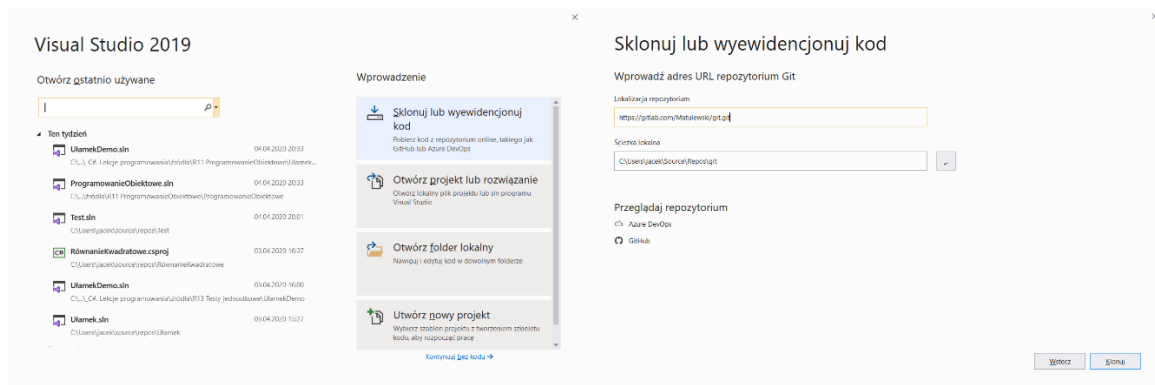
I na tym zasadniczo polega praca z repozytoriami *Git*: wprowadzane w edytorze zmiany należy zatwierdzać, a następnie utworzony w ten sposób commit, wypchnąć do zdalnego repozytorium. Dzięki temu tworzymy historię zmian, a jednocześnie uzyskujemy kopię zapasową kodu. To jednak nie wszystkie korzyści. Nie dotknęliśmy jeszcze przecież zagadnienia pracy zespołowej.



Rysunek C.14. Zmiany w gałęzi develop widoczne na stronie GitLab

Klonowanie projektu

Załóżmy teraz, że do projektu dołącza nowy członek zespołu. Jego pierwszą czynnością powinno być pobranie całego projektu ze zdalnego repozytorium. Uruchamia wobec tego na swoim komputerze środowisko Visual Studio 2019 i na stronie powitalnej (rysunek C.15, lewy) wybiera pozycję *Sklonuj lub wyewidencjonuj kod*. Zobaczysz dzięki temu okno o takim samym tytule (rysunek C.15, prawy), w którym jest pole tekstowe *Lokalizacja repozytorium*. Tu należy wpisać adres URL, który uzyskaliśmy z usługi *GitLab* tworząc w niej projekt. W moim przypadku jest to <https://gitlab.com/Matulewski/git.git>.



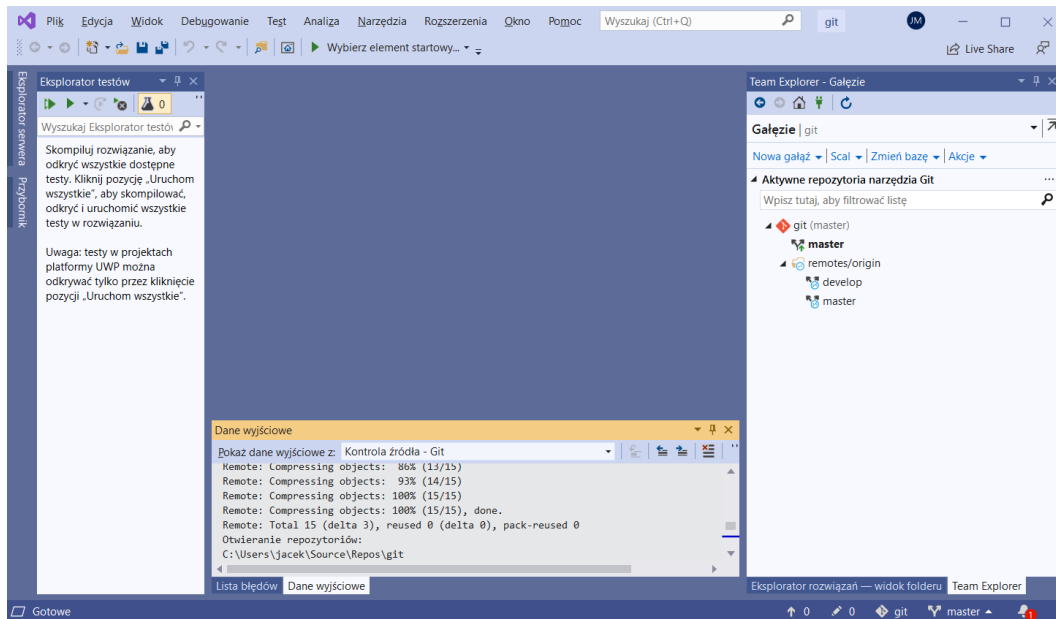
Rysunek C.15. Pobieranie projektu ze zdalnego repozytorium.

Kliknięcie przycisku *Klonuj* spowoduje otwarcie głównego okna Visual Studio. Następnie poproszeni zostaniemy o autoryzację w serwisie *GitLab* (por. rysunek C.5). Do tego należy użyć konta z serwisu *GitLab*, które ma uprawnienia do klonowanego projektu (takie uprawnienie należy nadać na stronie GitLab, zapraszając współpracownika do projektu). Ja jednak użyłem jeszcze raz swojego własnego konta, bo tak naprawdę celem klonowania nie jest w moim przypadku praca zespołowa, a możliwość pracy nad projektem z wielu stanowisk. Taki scenariusz też jest możliwy i częsty. Nie zmienia jednak zasadniczej zalety – możliwość równoczesnej modyfikacji jednego projektu z wielu stanowisk.

Obawy może budzić, że po wczytaniu projektu nie zobaczymy dodanego do niego pliku *Class1.cs*. To dlatego, że domyślnie pokazywana jest gałąź główna (ta o nazwie *master*). Wystarczy jednak rozwinąć listę gałęzi na pasku stanu (ikona gałęzi z etykietą *master*) i wybrać polecenie *Zarządzaj gałęziami*. Dzięki temu w podoknie *Team Explorer* zobaczymy stronę *Gałęzie* (rysunek C.16), na której możemy dwa razy kliknąć gałąź *develop*. Zmianę gałęzi potwierdzi zmiana etykiety ikony gałęzi widoczna na pasku stanu⁴. Natomiast w *Eksploratorze*

⁴ Kolejne przełączania między gałęziami będzie można już wykonać z samej ikony gałęzi na pasku stanu – po jej kliknięciu, na szczycie rozwijanej listy będą widoczne obie nazwy gałęzi.

rozwiązań zobaczymy plik *Class1.cs*. Warto podkreślić, że klonowanie dotyczy całego projektu, z wszystkimi gałęziami w repozytorium, natomiast synchronizacja (pobieranie i wypychanie zatwierdzonych zmian) dotyczy tylko bieżącej gałęzi.



Rysunek C.16. Wybór gałęzi po sklonowaniu projektu ze zdalnego repozytorium

Rozwiązywanie konfliktów (scalanie)

Nadal pozostajemy na drugim stanowisku pracy. Wprowadźmy jakąś zmianę w kodzie źródłowym projektu. Proponuję w klasie *Class1* z pliku *Class1.cs* zdefiniować pole i konstruktor (listing C.1). Plik *Class1.cs* w *Eksploratorze rozwiązań* zostanie natychmiast oznaczony czerwonym znacznikiem. Następnie zatwierdźmy te zmiany i wypchnijmy je do zdalnego repozytorium. To oznacza, że musimy kliknąć ikonę zmian (z ołówkiem i liczbą 1), co spowoduje pojawienie się w podoknie *Team Explorer* strony *Zmiany*. Równocześnie pojawi się okno dialogowe z pytaniem o to, jak podpisać zmiany z tego komputera – ja używam tego samego konta, ale w etykiecie dodałem informacje, że zmiany wysyłane są z laptopa (rysunek C.17). Musimy oczywiście podać także nazwę zatwierdzonych zmian – u mnie jest to „Definicja pola i konstruktora w *Class1*” – i kliknąć przycisk *Zatwierdź wszystko*. Następnie kliknijmy ikonę niewypchniętych zatwierdzeń na pasku stanu, co spowoduje pokazanie strony *Synchronizacja* w podoknie *Team Explorer*. W niej kliknijmy przycisk *Synchronizuj*. To tego przycisku powinniśmy od tej pory używać, aby mieć pewność, że zmiany wypchnięte z komputera do zdalnego repozytorium nie doprowadzą do tego, że projektu z bieżącej gałęzi nie będzie można skompilować. Synchronizacja oznacza, że przed wysłaniem commitu do zdalnego repozytorium, sprawdzone zostanie wpiertw to, czy nie ma w nim zmian przesłanych przez innego członka zespołu lub z innego stanowiska (opisany we wstępie proces *fetch*). Jeżeli takie zmiany są, zostaną pobrane na lokalny komputer. Jeżeli te zmiany dotyczą tych samych plików, co zmiany wprowadzone z lokalnego komputera, będziemy mieli szansę je przejrzeć i scalić kod. My jednak wprowadzamy zmiany do przed chwilą sklonowanego projektu, więc synchronizacja powinna się powieść bez zgłaszania żadnych konfliktów i konieczności scalania kodu. Powinien pojawić się komunikat potwierdzający jej zakończenie.

Listing C.1. Zmiany wprowadzone w pliku *Class1.cs* z drugiego komputera

```
using System;

using System.Collections.Generic;

using System.Text;

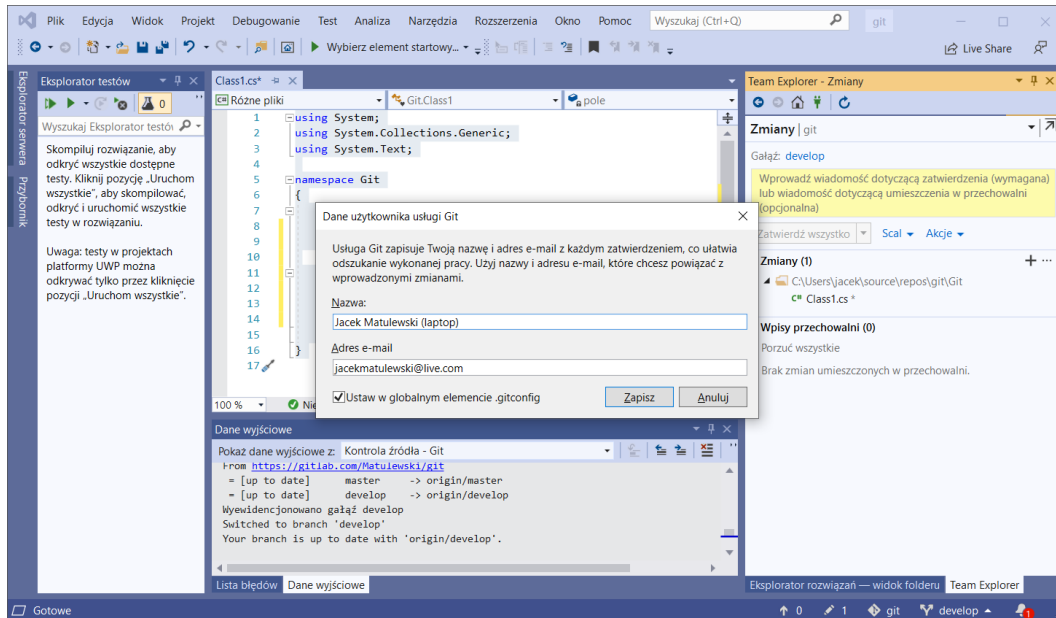
namespace Git
{
    class Class1
    {
```

```

private int pole;

public Class1(int argument)
{
    pole = argument;
}
}

```



Rysunek C.17. Ustalanie podpisu jakim zostaną opatrzone zatwierdzone zmiany u nowego członka zespołu lub na nowym stanowisku

Wróćmy teraz do pierwszego komputera, który nie widzi jeszcze zmian przesłanych z drugiego stanowiska do zdalnego repozytorium. Nie pobieramy ich na razie z serwera, choć generalnie dobrze jest zaczynać pracę właśnie od synchronizacji z serwerem. Unikniemy w ten sposób konieczności późniejszego skomplikowanego scalania. Załóżmy jednak, że dwaj członkowie zespołu pracują przy dwóch komputerach nad tym samym projektem i edytują ten sam plik. Może się zdarzyć, że ich zmiany będą dotyczyły tych samych fragmentów kodu. Dla przykładu wprowadźmy do pliku *Class1.cs* zmiany widoczne na listingu C.2.

Listing C.2. Niezależne zmiany wprowadzone w pliku *Class1.cs*

```

using System;

using System.Collections.Generic;

using System.Text;

namespace Git
{
    class Class1
    {
        private int pole = 1;

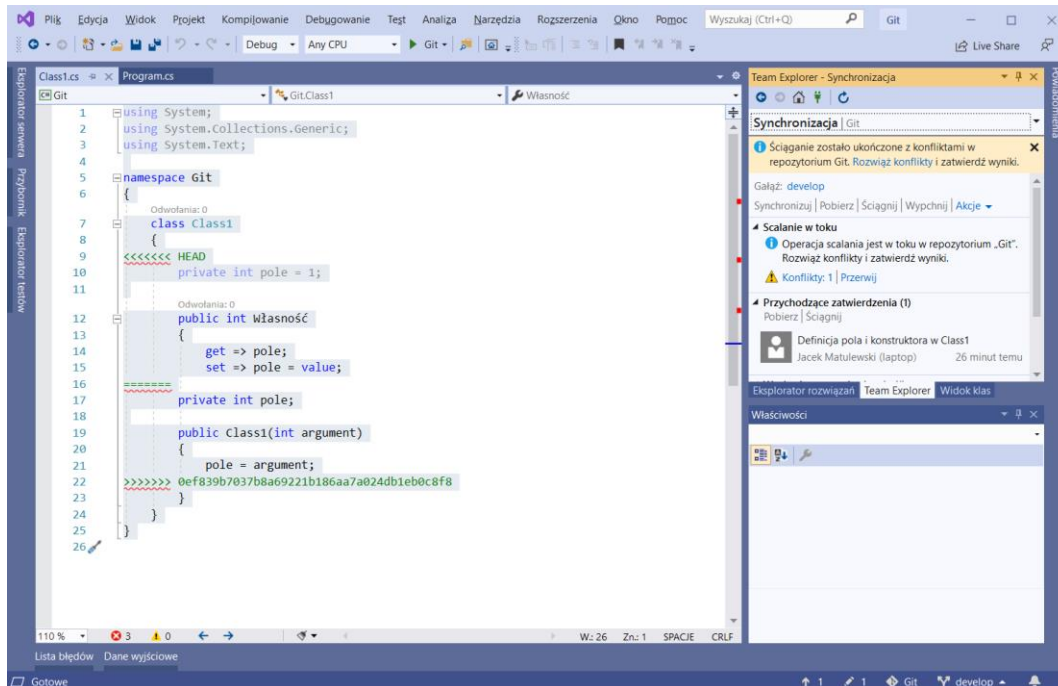
        public int Własność
        {
            get => pole;
            set => pole = value;
        }
    }
}

```

```
}  
}
```

Tą zmianę spróbujmy teraz przesłać na serwer. Wpierw musimy ją zatwierdzić, nadając jej nazwę „Zdefiniowane pole i własność w Class1”, a następnie spróbujemy wypchnąć. Pamiętajmy, aby w oknie *Synchronizacja* nie używać funkcji *Wypchnij*, a *Synchronizuj*. To ważne, żeby przed przesłaniem na serwer, sprawdzić wprowadzone w międzyczasie zmiany w zdalnym repozytorium.

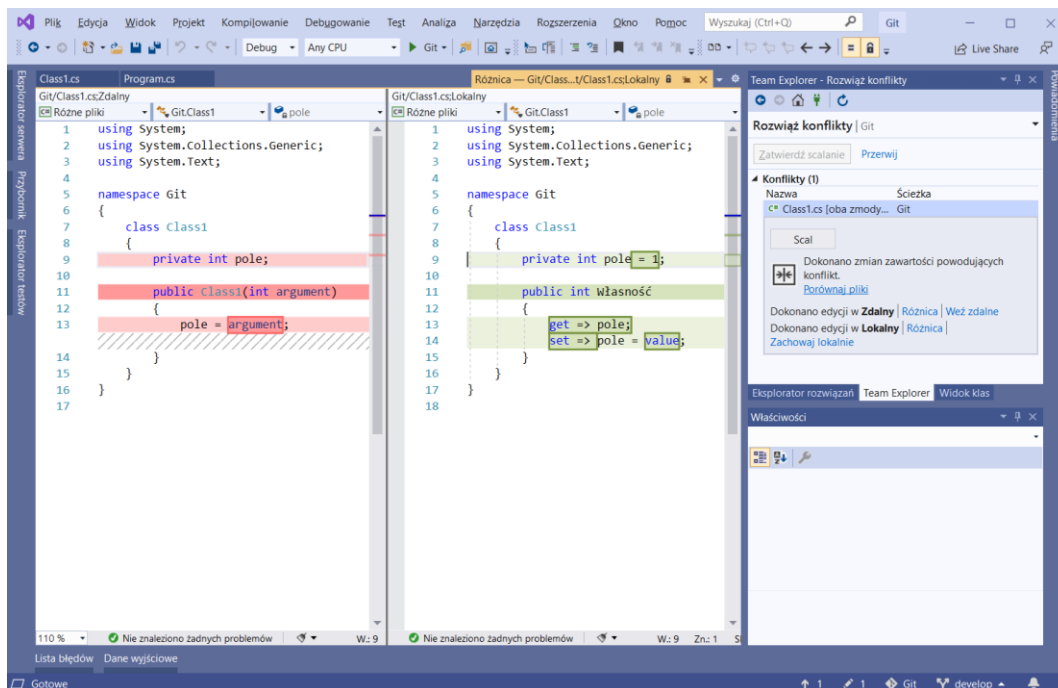
Efekt będzie taki, że w oknie *Team Explorer* zobaczymy komunikat „Ściąganie zostało ukończone z konfliktami w repozytorium Git. Rozwiąż konflikty i zatwierdź wyniki”. Jednocześnie w oknie edytora zobaczymy zawartość pliku *Class1.cs* z naniesionymi zmianami z obu komputerów (rysunek C.18). To oznacza, że synchronizacja zakończyła się na etapie pobierania projektu z serwera. Kod z bieżącego komputera nie został jeszcze wysłany. Najpierw należy rozwiązać konflikty w kodzie.



Rysunek C.18. Próba synchronizacji zakończyła się zgłoszeniem konfliktów

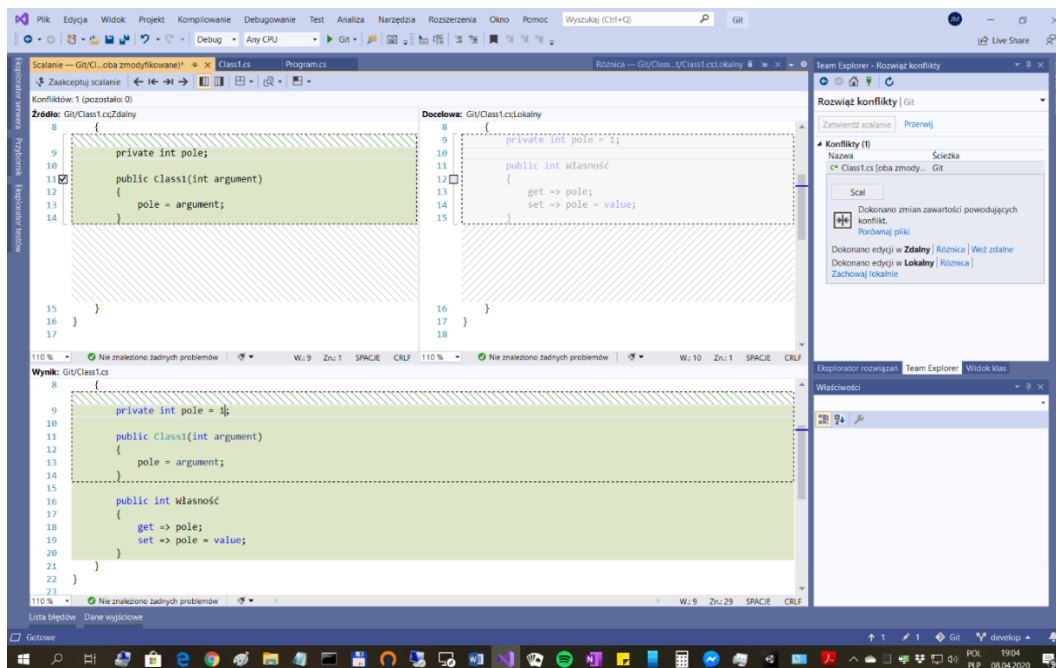
Zanim zaczniemy scalanie, przejdźmy do strony głównej w podoknie *Team Explorer* (ikona z domkiem), a na niej kliknijmy pozycję *Ustawienia*, a następnie *Ustawienia Globalne*. Następnie na dole strony *Ustawienia Git* wybierz Visual Studio jako narzędzie do porównywania i do scalania kodu.

Wróćmy do okna *Synchronizacja* i kliknijmy *Rozwiąż konflikt*. Pojawi się okno *Rozwiąż konflikt*, w którym klikamy *Porównaj pliki*. Zobaczymy wówczas ciekawszy widok, w którym z lewej strony jest kod pobrany z serwera, a z prawej nasz kod (rysunek C.19). W tym widoku możemy wygodnie porównać obie wersje kodu, ale nie możemy go edytować.



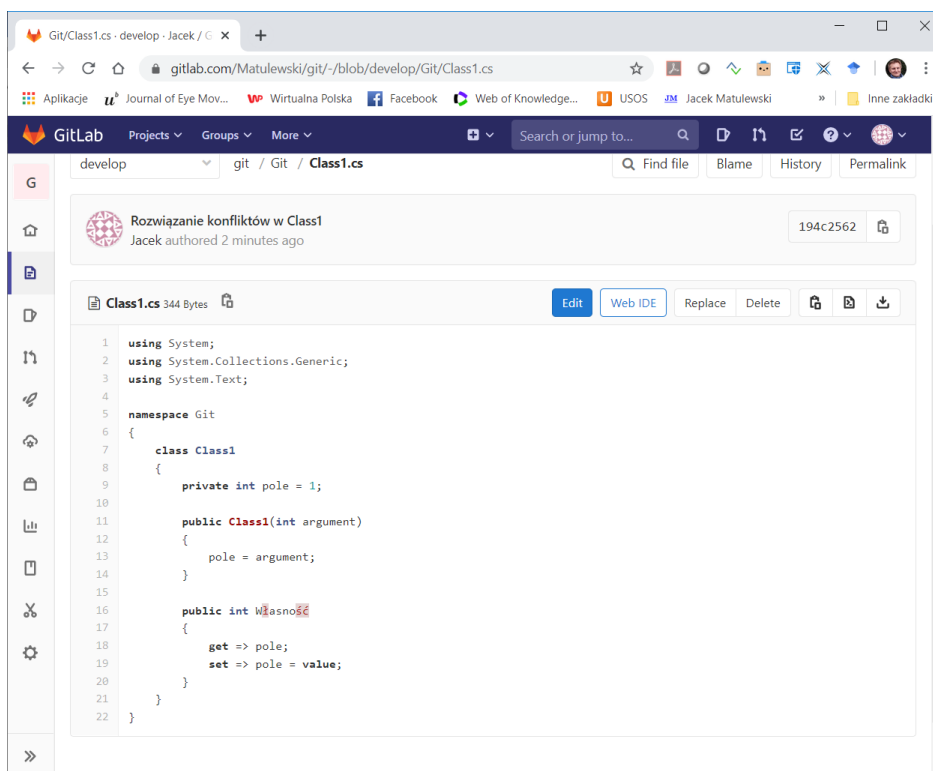
Rysunek C.19. Narzędzie porównywania dwóch wersji kodu w Visual Studio

Aby to było możliwe, kliknijmy *Scal* w podoknie *Team Explorer* (strona *Rozwiąż konflikty*). To przełączy nas do kolejnego widoku, w którym widoczne są trzy panele (rysunek C.20). Podobnie jak w widoku porównywania, z lewej u góry jest widoczny kod z serwera, a z prawej u góry nasz kod. Jednak teraz przy każdym zmienionym fragmencie jest pole opcji (*checkbox*), pozwalające na wybór jednej z wersji. Zaznaczona wersja pojawi się w dolnym panelu, w którym jest scalony kod, który możemy także edytować. Możemy zaznaczać poszczególne „skonfliktowane” fragmenty klikając pola opcji w lewym lub prawym panelu u góry i w ten sposób wybierając te, które mają znaleźć się w scalonym kodzie na dole. Niestety w naszym przypadku zmiany dotyczą jednego fragmentu, więc możemy wybrać albo wersję z polem i konstruktorem, albo wersję z polem i własnością. Nie ma jednak problemu, żeby własność skopiować z prawego panelu i dodać ją samodzielnie do scalonego kodu. Efekt może być taki, jaki widoczny jest w dolnym panelu na rysunku C.20. Scalanie kończymy klikając *Zaakceptuj scalanie* na pasku narzędzi w lewym górnym rogu zakładki *Scalanie*. Możemy wówczas zamknąć tą zakładkę. Zaakceptowany scalony kod dostępny jest teraz w plikach projektu (oczywiście w tej gałęzi, w której aktualnie jesteśmy tj. w naszym przypadku w gałęzi *develop*). Po scaleniu konieczne trzeba sprawdzić, czy projekt można skompilować.



Rysunek C.20. Narzędzie scalania kodu w Visual Studio

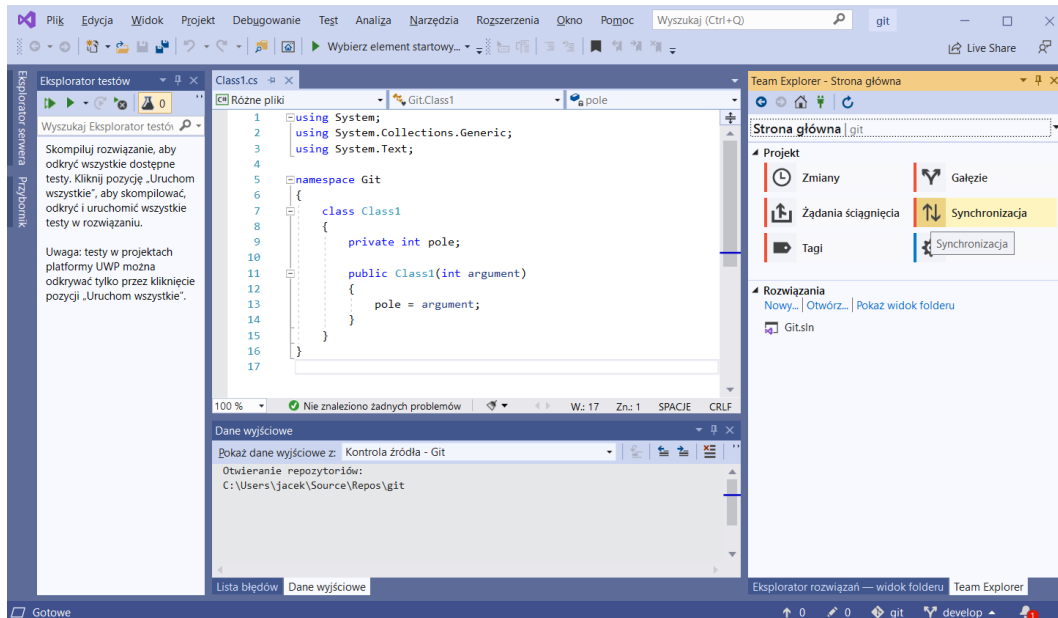
Chcemy jednak dokończyć synchronizację. Przejdźmy w tym celu ponownie do zatwierdzania zmian (ikona z ołówkiem na pasku stanu). Commit nazwijmy „Rozwiązanie konfliktów w Class1” i kliknijmy „Zatwierdź przygotowane”. W efekcie będziemy mieli dwa niewypchnięte zatwierdzenia (liczba dwa przy strzałce w górę na pasku stanu). Kliknijmy ją i przejdźmy do strony *Synchronizacja*, na której kliknijmy *Synchronizuj*. Teraz synchronizacja powinna przebiec bez problemu, co możemy sprawdzić na stronie *GitLab*. Powinien tam być scalony kod (rysunek C.21, przy okazji zobaczymy, że podgląd kodu w serwisie *GitLab* nie lubi polskich znaków).



Rysunek C.21. Scalony kod widoczny w podglądzie zdalnego repozytorium na stronie GitLab

Warto wrócić teraz do drugiego komputera i pobrać zmiany. Możemy to zrobić przechodząc w oknie *Team Explorer* do strony *Synchronizacja* (rysunek C.22). Pamiętajmy, żeby teraz i zawsze używać do pobierania

polecenia *Synchronizuj*. W ten sposób stan projektu na obu komputerach zostanie zsynchronizowany, co dzieje się za pośrednictwem serwera, który przechowuje poszczególne commity wraz z informacją o tym, kto je przesłał. Jak wspominałem we wstępie, to pozwala nie tylko na wersjonowanie kodu i tworzenie jego kopii bezpieczeństwa, ale również na kontrolę pracy zespołu i rozwoju projektu. Przypominam, że każda gałąź synchronizowana jest niezależnie, a więc synchronizacja gałęzi *develop* nie powoduje synchronizacji gałęzi *master*, choć ta ostatnia tego w tym momencie nie wymaga.

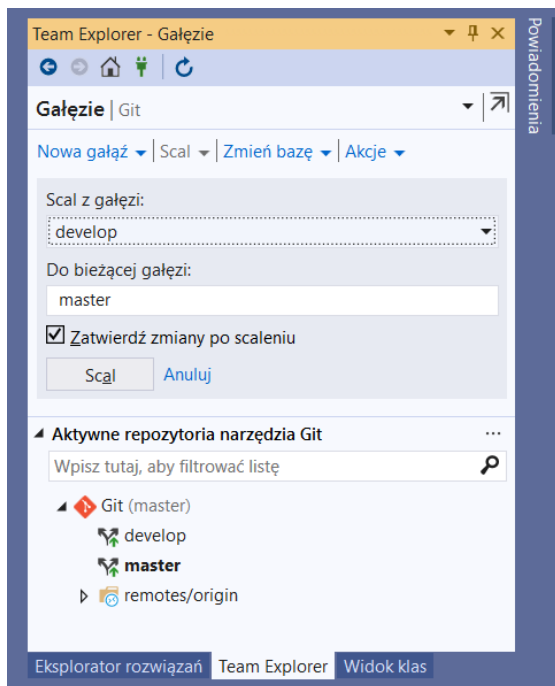


Rysunek C.22. Potrzebna jest synchronizacja projektu na drugim komputerze

Scalanie gałęzi

W miarę rozwoju projektu, gromadzi się wiele zmian wypchniętych do zdalnego repozytorium, ale przechowywanych także w lokalnym repozytorium. Jeżeli w efekcie wprowadzania tych zmian uda się zaimplementować nową funkcjonalność i została ona już przetestowana, możemy myśleć o przeniesieniu nowego kodu z gałęzi *develop* do gałęzi *master*, która powinna być miejscem przechowywania stabilnej wersji projektu. Oficjalnie zakończymy w ten sposób cykl (iterację) rozwoju oprogramowania i dodawania do niego nowej funkcjonalności.

Aby scalić gałęzie, kliknijmy ikonę gałęzi na pasku stanu Visual Studio (tą z napisem *develop*) i przełączmy się na gałąź *master*. Następnie jeszcze raz użyjmy tej ikony i wybierzmy polecenie *Zarządzaj gałęziami*. W efekcie w oknie *Team Explorer* pojawi się strona *Gałęzie*, w której klikamy *Scal*. Pojawi się wówczas strona z rozwijaną listą *Scal z gałęzi*, w której wskazujemy gałąź *develop* i polem *Do bieżącej gałęzi*, w której pozostawiamy *master* (rysunek C.23). Zadbajmy, aby zaznaczone było pole *Zatwierdź zmiany po scaleniu* i kliknijmy przycisk *Scal*. Nie powinno być konfliktów, ponieważ gałąź *master* od początku pozostawała niezmienną. Jeżeli byłyby konflikty, do scalania używalibyśmy tych samych narzędzi, których wcześniej używaliśmy przy scalaniu commitów z różnych stanowisk.



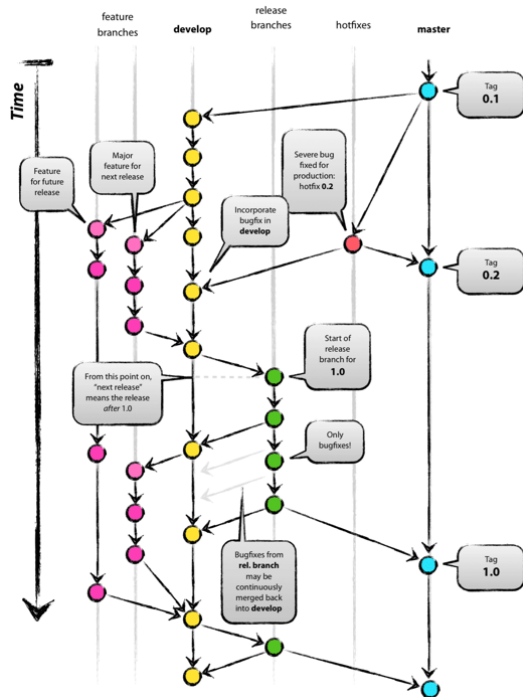
Rysunek C.23. Scalanie gałęzi na stronie Gałęzie widocznej w podoknie Team Explorer

Przejdźmy teraz do *Eksploratora rozwiązań* – jest w nim widoczny plik *Class1.cs*. To oznacza, że kod z gałęzi *develop* rzeczywiście został przeniesiony do *master*. Zwróćmy też uwagę, że na pasku są 4 niewypchnięte zatwierdzenia – scalenie gałęzi odbyło się lokalnie, gałąź *master* w repozytorium zdalnym nie została jeszcze zsynchronizowana. Warto to teraz zrobić, przechodząc w podoknie *Team Explorer* do strony głównej (ikona domku), a następnie do strony *Synchronizacja* (lub można tu wejść od razu klikając ikonę niewypchniętych zatwierdzeń na pasku stanu). Po synchronizacji, nowy stan gałęzi *master* można pobrać na drugim komputerze (pamiętajmy, żeby zawsze używać do tego polecenia *Synchronizuj*).

Scalenie gałęzi, nie powoduje że gałąź *develop* zniknie. Jeżeli dalszy rozwój będziemy prowadzili w gałęzi *master*, czego nie powinniśmy robić, lub stworzymy nową gałąź do rozwijania kolejnej funkcjonalności, gałąź *develop* przechowywać będzie stan projektu sprzed scalenia. Można jednak gałęzi *develop* używać dalej, a potem ponownie scalić z nią gałąź *master*, żeby przenieść do głównej gałęzi nową zaimplementowaną funkcjonalność. Tak, czy inaczej w repozytoriach pozostanie commit z wersją tej gałęzi w momencie scalenia.

W większych zespołach, szczególnie jeżeli prowadzą „eksperymenty” z różnymi technologiami, może powstać wiele gałęzi. Osobne gałęzie mogą służyć jako bezpieczne „piaskownice” dla testowania pomysłów, bądź być miejscem pracy mniej doświadczonych członków zespołu (scalanie ich gałęzi przez bardziej doświadczonego członka zespołu będzie świetną formą nadzoru nad ich pracą). Wiele gałęzi powstanie również, gdy zespół stosuje metodologię *Git flow*⁵, w której oprócz gałęzi *master* i *develop* tworzy się osobne gałęzie dla poszczególnych implementowanych funkcjonalności (*features*) i wydań oprogramowania (*releases*). Osobną gałąź mają także poprawki, jakie trzeba na szybko wprowadzać do kodu w gałęzi *master* (*hotfixes*).

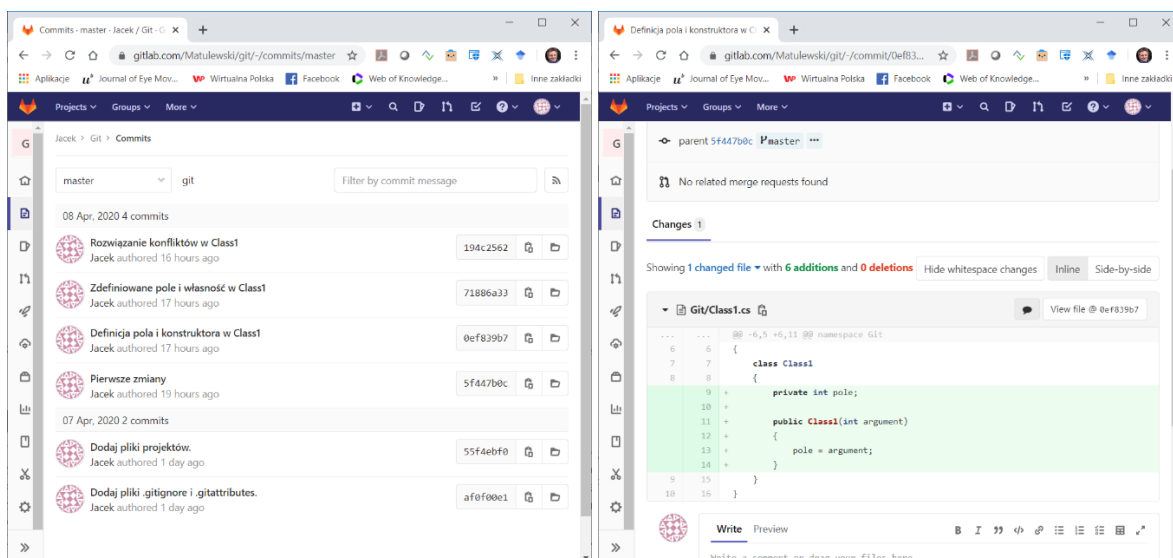
⁵ Zob. artykuł autora Git flow pod adresem <https://nvie.com/posts/a-successful-git-branching-model/>



Rysunek C.24. Schemat gałęzi w git flow (źródło: <https://nvie.com/posts/a-successful-git-branching-model/>)

Przywracanie wcześniejszej wersji projektu

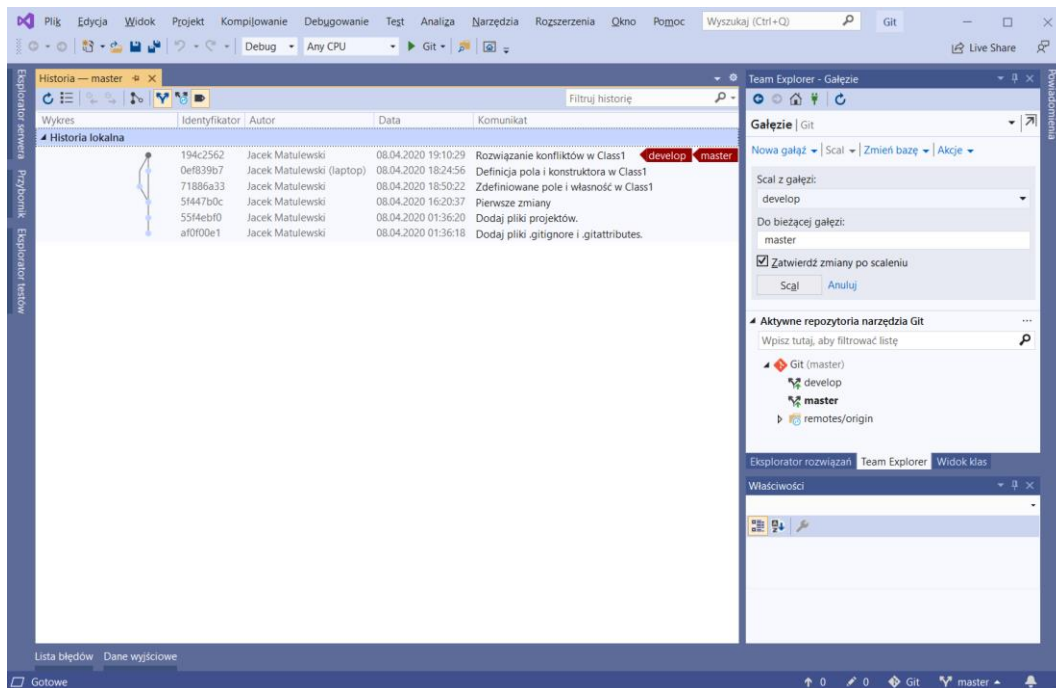
Jeżeli okazuje się, że zmiany wprowadzane w kodzie nie prowadzą do pożądanego efektu i zespół postanowił się z nich wyciągnąć, możemy chcieć przywrócić jedną ze starszych wersji gałęzi. Aby przejrzeć przesłane commity w zdalnym repozytorium w serwisie *GitLab*, należy wejść w przeglądarce na stronę projektu. W górnej części strony widoczna jest (od prawej) wielkość projektu, liczba tagów, o których nie wspominałem, liczba gałęzi i liczba commitów. Kliknięcie tej ostatniej pozycji (powinniśmy mieć 6 zatwierdzeń), spowoduje wyświetlenie ich listy (rysunek C.25). Domyślnie uporządkowana jest ona datami tak, że najnowsza jest na samym szycie listy. Każdą wersję projektu można podejrzeć – zobaczymy wówczas zmiany w kodzie źródłowym przesłanych w commicie plików wyróżnione zielonym kolorem.



Rysunek C.25. Lista commitów w zdalnym repozytorium i podgląd jednego z nich

Z kolei historię wersji w lokalnym repozytorium projektu możemy zobaczyć w Visual Studio klikając ikonę gałęzi w pasku stanu i wybierając polecenie *Wyświetl historię...* Zobaczymy wówczas listę commitów (rysunek C.26). W szczególności widać w formie graficznej fakt rozgałęzienia kodu spowodowanego przez niezależne

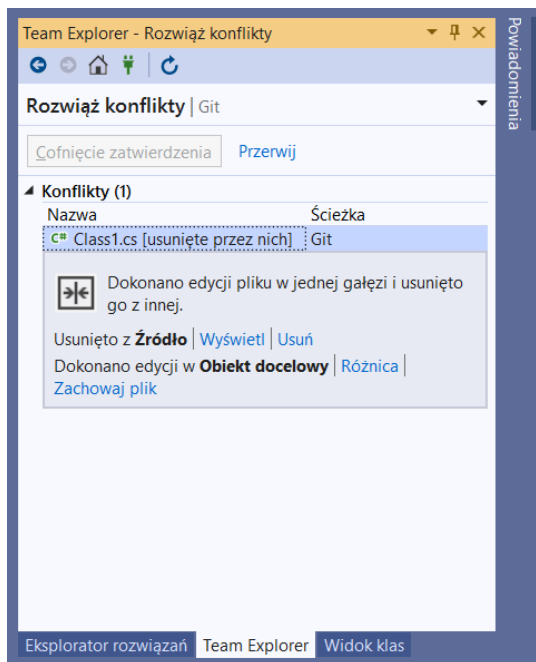
wprowadzanie zmian. W naszym przypadku dotyczy to zmian w klasie `Class1` (pozycja *Zdefiniowane pole i własność w Class1*), które później zostały scalone przy synchronizacji (pozycja *Rozwiązanie konfliktów w Class1*). Ten graficzny schemat nie przedstawia historii gałęzi.



Rysunek C.26. Przeglądanie historii wersji w lokalnym repozytorium

Zwróćmy też uwagę na dwa pierwsze commity, które zostały przesłane do lokalnego i zdalnego repozytorium automatycznie w momencie dodawania projektu do kontroli kodu źródłowego. Pierwszy dodaje dwa pliki, które za chwilę opiszę, a drugi – zasadniczy kod źródłowy projektu z momentu dodania obsługi *Git*.

Aby przywrócić którąś z wersji w lokalnym repozytorium, wystarczy kliknąć ją prawym klawiszem myszy i z menu kontekstowego wybrać pozycję *Przywróć*. Zanim tę możliwość przetestujemy, przejdźmy najpierw do gałęzi *develop* (pozwoli na to ikona gałęzi w pasku stanu). Następnie korzystając z tej samej ikony, wyświetlimy historię jej wersji i rozwinimy menu kontekstowe na pozycji *Pierwsze zmiany*, a z niej wybierzmy *Przywróć*. Pojawi się okno dialogowe z prośbą o potwierdzenie. Jeżeli potwierdzimy, w podoknie *Team Explorer* pojawi się okno *Rozwiąż konflikty*, w którym widoczna jest informacja, że przywrócenie wskazanej wersji spowoduje usunięcie z projektu pliku *Class1.cs* (rysunek C.27). W tym oknie jest też dostępne polecenie *Przerwij*, które przerywa przywracanie bez wprowadzenia zmian. Jeżeli jednak chcemy kontynuować przywracanie, powinniśmy zdecydować, czy rzeczywiście usunąć ten plik, czy go jednak zachować. Jeżeli różnice między bieżącą wersją projektu, a wersją przywracaną dotyczą większej liczby plików, taką decyzję trzeba podjąć w stosunku do każdego z nich. My kliknijmy *Usuń*. Po tym możemy kliknąć aktywny już teraz przycisk *Cofnij zatwierdzenie* (przypominam, że zatwierdzenie to polska nazwa commitu – pasowałaby tu raczej nazwa *Przywróć wcześniejsze zatwierdzenie*). Zmiana taka nie oznacza jednak wymazania historii wersji w danej gałęzi i usunięcia nowszych wersji niż ta przywracana. Wręcz przeciwnie, oznacza dodanie kolejnej zmiany do tej historii – zmiany, którą musimy teraz nazwać, zatwierdzić i możemy wypchnąć do zdalnego repozytorium. Jeżeli otwarta jest nadal zakładka z historią wersji dla gałęzi *develop*, aby zobaczyć ten nowy commit musimy kliknąć ikonę *Odśwież* na jej pasku narzędzi.



Rysunek C.27. Zatwierdzenie przywracania

O czym nie musimy wiedzieć korzystając z Git w Visual Studio

W katalogu projektu znajdziemy dwa wspomniane pliki, które zostały dodane do repozytorium w pierwszym automatycznie robionym zatwierdzeniu: `.gitignore` i `.gitattributes`. Pierwszy zawiera listę plików, a raczej mask plików wskazujących na ich rozszerzenia, które nie są przesyłane na serwer, ani z niego pobierane. Domyślnie są to wszystkie pliki tymczasowe tworzone przez Visual Studio w trakcie pracy nad projektem, a więc np. `*.suo`, a także pliki powstające podczas kompilacji, które nie są konieczne do uruchomienia programu np. pliki `.pdb`. Na szczęście nie musimy tego pliku tworzyć samodzielnie, bo powstaje i jest dostosowany do konkretnego szablonu projektu Visual Studio w momencie podłączania do repozytorium. Drugi plik to `.gitattributes`, który konfiguruje działanie *Git*. On też jest tworzony automatycznie i jest dostosowany do bieżącego projektu, więc zazwyczaj nie musimy nawet do niego zaglądać. W tym pliku określone są działania dla różnych rodzajów plików (np. czy traktować je jako tekstowe, czy jako binarne).

* * *

To oczywiście daleko nie wszystko, co można powiedzieć na temat *Git*. Nie wspomniałem na przykład w ogóle o nadawaniu uprawnień do poszczególnych procesów *Git*, w szczególności do scalania, czy o metodyce CI/CD (*continuous integration/continuous delivery*), w której repozytoria odgrywają zasadniczą rolę. Ale przedstawiona została ta część praktycznych informacji, która pozwoli na sprawną pracę z *Git* w typowych sytuacjach.