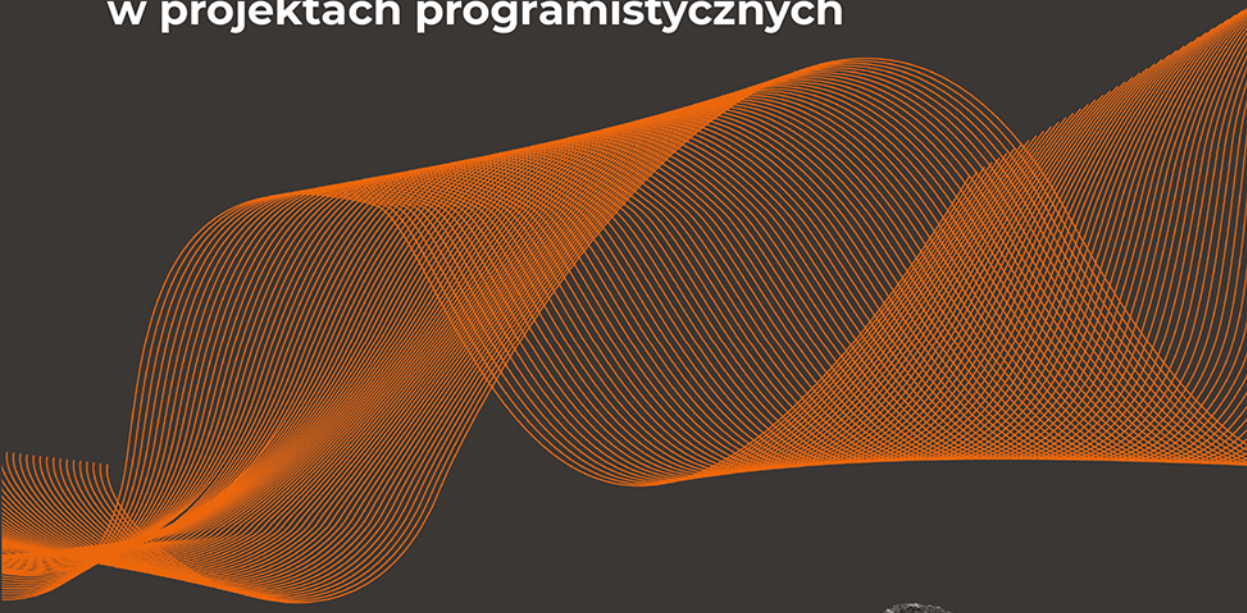
Git dla programistów

Efektywna kontrola wersji w projektach programistycznych



Jesse Liberty

Helion

Packt

Tytuł oryginału: Git for Programmers: Master Git for effective implementation of version control for your programming projects

Tłumaczenie: Piotr Luboch

ISBN: 978-83-283-8914-4

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/gitdla
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

4

ROZDZIAŁ

Scalanie, zgłoszenia i obsługa konfliktów scalania

W tym rozdziale zobaczysz, jak scalać gałęzie za pomocą różnych rodzajów scalania. Poznasz również sposoby na radzenie sobie z konfliktami scalania oraz narzędzia sprawiające, że zarządzanie tymi konfliktami będzie prostsze. Zdobędziesz wiedzę na temat zgłoszeń (*pull requests*) oraz zgłębisz różnicę między scalaniem przewijanym (*fast-forward merge*) a „prawdziwym" scalaniem.

W tym rozdziale dowiesz się:

- jak wysłać zmiany na serwer,
- jak zarządzać commitami w wierszu poleceń, Visual Studio i GitHub Desktop,
- jak scalić zmiany z główną gałęzią,
- czym jest zgłoszenie,
- czym są konflikty scalania i jak je rozwiązywać,
- czym jest scalanie przewijane,
- czym jest prawdziwe scalanie.

Zacznijmy od zarysowania pojęcia scalania.

Scalanie — zarys

Gdy znajdujesz się na gałęzi funkcji i ta nowa funkcja jest poprawnie zaimplementowana i przetestowana, to będziesz chciał scalić tę gałąź z gałęzią główną. W niektórych firmach będziesz mógł po prostu scalić zmiany, inne (większość?) będą wymagały od Ciebie utworzenia ***zgłoszenia*** (***PR***[1]). *PR* zasadniczo oznacza „sprawdź, proszę, mój kod i jeśli myślisz, że jest prawidłowy, scal go z gałęzią główną".

Gdy druga (albo i trzecia) para oczu spojrzy na Twój kod przed scaleniem, może to zaoszczędzić wielu rozterek w późniejszym czasie (aby dowiedzieć się więcej o naprawianiu błędów, sprawdź rozdział 12. „Naprawianie błędów").

Jeśli będziesz ostrożny (patrz poniżej), to najczęściej będziesz scalał bez problemów. Jednak od czasu do czasu natkniesz się na budzący postrach konflikt scalania (*merge conflict*). Poniżej zapoznasz się z kilkoma sposobami, jak radzić sobie w takich przypadkach.

Książki

Jak zapewne pamiętasz z poprzedniego rozdziału, folder *C:\GitHub\VisualStudio\ProGitFor*
↪Programmers jest domem dla aplikacji dotyczącej książek. Zmian w niej dokonywaliśmy przy użyciu Visual Studio. Oczywiście, nie musimy nią zarządzać w Visual Studio; możemy użyć dowolnego z naszych narzędzi. Mogę na przykład otworzyć terminal i przejść do katalogu zawierającego książkową aplikację:

Rysunek 4.1. Otwieranie terminala

Zauważ, że na rysunku 4.1 wyświetlana jest informacja o jednej zmianie do wysłania (jest to zasygnalizowane przez strzałkę wskazującą w górę, po której następuje 1). Musiałem zapomnieć to zrobić ostatnio, kiedy pracowałem z tym kodem. Jednak nie chcę wysłać tych zmian ot tak, bo kto wie, co się tam kryje? Istnieje kilka sposobów, aby się dowiedzieć.

Co będę wysyłać?

W wierszu poleceń możemy użyć polecenia **`git show`**:

[1] Od ang. *Pull Request — przyp. tłum.*

```
❯ git show
commit c3c60c3deebda09633518fa47e40a3b0ba4d0ac8 (HEAD -> Book)
Author: Jesse Liberty <JesseLiberty@non.se.com>
Date:   Mon Feb 8 09:08:08 2021 -0500

    Add properties

diff --git a/ProGitForProgrammers/ProGitForProgrammers/Book.cs b/ProGitForProgrammers/ProGitForProgrammers/Book.cs
new file mode 100644
index 0000000..43a0844
--- /dev/null
+++ b/ProGitForProgrammers/ProGitForProgrammers/Book.cs
@@ -0,0 +1,14 @@
+ using System;
+using System.Collections.Generic;
+using System.Text;
+
+namespace ProGitForProgrammers
+{
+    public class Book
+    {
+        public string Title {get; set;}
+        public List<string> Authors {get; set;}
+
+        public DateTime PublicationDate {get; set;}
+    }
+}
SESA560987@DESKTOP-D21661F C:\GitHub\VisualStudio\ProGitForProgrammers  Book ↑1
```

Rysunek 4.2. Kontrolowanie zmian do wysłania

Na rysunku 4.2 znajduje się wiele informacji. Po pierwsze widzimy autora i datę. Następnie znajduje się opis zmian (*Add properties — Dodaj właściwości*). Potem Git wykonuje `diff` (polecenie wyświetla różnice, zmiany) pomiędzy *Book.cs* i *Book.cs*, nazywając ten pierwszy *a* i drugi *b*. *Book.cs* oznaczony jako *a* to plik z poprzedniego commita, natomiast ten oznaczony jako *b* to zawartość z nowego commita.

Zapewne zauważyłeś linijkę o treści `/dev/null`. Oznacza, że plik jest porównywany z pustymi danymi, tak więc wszytko jest nowe.

W kolejnej linii (rysunek 4.3) widzimy informację o porównaniu `/dev/null` z plikiem *b* (nowym *Book.cs*):

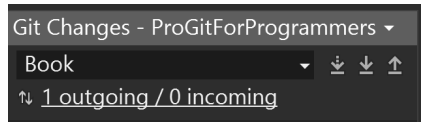
```
diff --git a/ProGitForProgrammers/ProGitForProgrammers/Bo
new file mode 100644
index 0000000..43a0844
--- /dev/null
+++ b/ProGitForProgrammers/ProGitForProgrammers/Book.cs
```

Rysunek 4.3. Porównanie z /dev/null

Na końcu wyświetlane są zmiany. To, co zostało usunięte, będzie oznaczone na czerwono, to, co zmodyfikowane, na zielono, a nowy kod na żółto (ten widok oraz kolory mogą się różnić w zależności od tego, jakiej powłoki używasz). Widzimy tu trzy instrukcje `using`, przestrzeń nazw oraz klasę `Book`. Wszystkie te elementy zostały dodane w ramach tego commita. Zanim wyślemy zmiany, zobaczmy, czego możemy się dowiedzieć w Visual Studio.

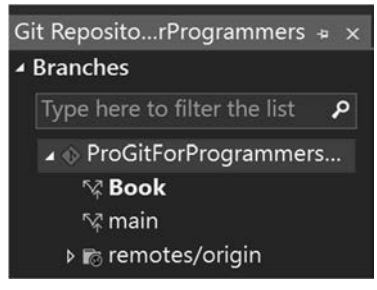
Visual Studio

Gdy otworzysz ten sam folder w Visual Studio i przejdziesz do okna Gita, to zgodnie z oczekiwaniami odkryjesz, że mamy jedną zatwierdzoną zmianę do wysłania (wychodzącą). Widać to na rysunku 4.4:



Rysunek 4.4. Visual Studio pokazujące jeden plik do wysłania

Zanim wyślemy zmiany, sprawdźmy, co zawierają. Kliknij *1 outgoing* (*1 wychodzący*), a otworzą się dwa okna, pokazane na rysunkach 4.5 i 4.6. Okno *Branches* (*Gałęzie*) pokazuje nam, na jakiej gałęzi się znajdujemy:



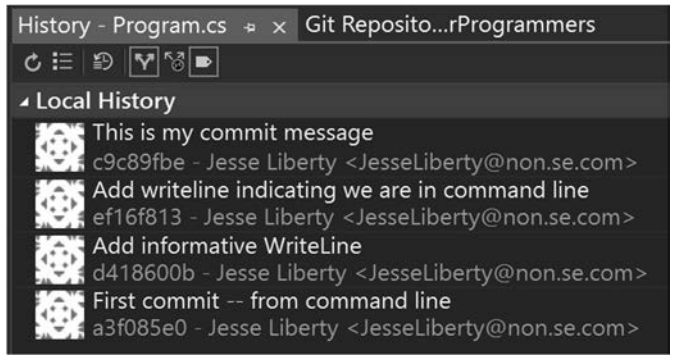
Rysunek 4.5. Visual Studio pokazujące zawartość lokalnego repozytorium

Środkowy panel ma kilka ciekawych informacji. Przedstawia lokalną (a nie źródłową) historię Twoich gałęzi:

Rysunek 4.6. Visual Studio pokazujące historię zmian

Widzimy, że *main* ma pięć commitów (od najnowszego do najstarszego) oraz że przed najnowszym commitem w *main* mamy wychodzące zmiany na gałęzi *Book*, oznaczone wiadomością *Add properties* (*Dodaj właściwości*). Jest to zgodne z tym, co widzieliśmy w wierszu poleceń.

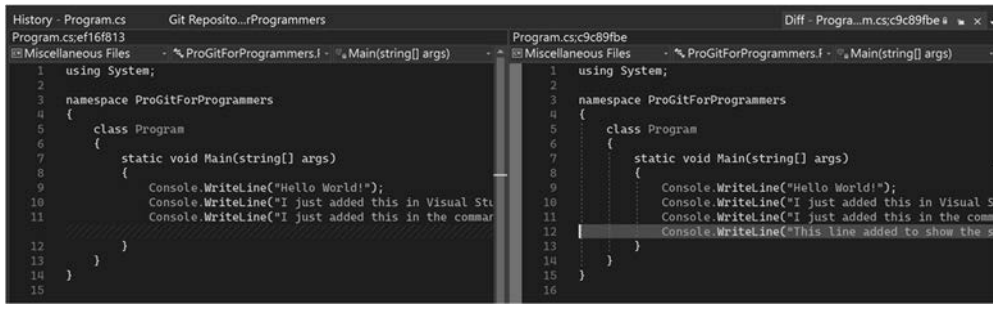
Możemy pójść dalej i wrócić do *Solution Explorer* (*Eksploratora rozwiązań*). Ponieważ więcej zmian dotyczy *Program.cs* niż *Book.cs*, kliknij prawym przyciskiem *Program.cs* i wybierz *Git*, a następnie *History* (*Wyświetl historię*). Otworzy to stronę *History* (*Historia*) dla *Program.cs*:



Rysunek 4.7. Visual Studio pokazujące historię Program.cs

> Jeśli Twój zarejestrowany użytkownik ma obrazek, będzie on wyświetlony po lewej stronie.

Na rysunku 4.7 widzimy 4 commity. Możemy je porównać, klikając je prawym przyciskiem myszy, na przykład pierwszy z nich, i wybierając *Compare with previous* (*Porównaj z poprzednim*). Otworzą się dwa okna. Po lewej zobaczysz starszy commit, a po prawej nowszy. Możemy zauważyć, że w nowym commicie jedna linijka została dodana. Visual Studio podświetla ją na zielono:

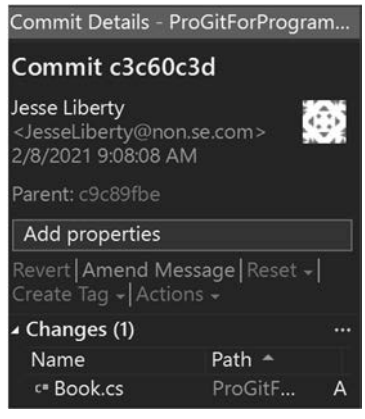


Rysunek 4.8. Porównywanie w sąsiednich oknach

Jak widzisz na rysunku 4.8, Visual Studio może przedstawić graficzną reprezentację tych samych informacji, które uzyskałeś w wierszu poleceń.

Szczegóły, szczególiki

Zamknijmy wszystkie okna dotyczące historii i wróćmy do listy wychodzących zmian i historii lokalnej. Poniżej *Outgoing* (*Wychodzące*) widzimy *Add properties* (*Dodaj właściwości*). Gdy klikniesz prawym przyciskiem tę linijkę, po prawej stronie otworzy się okno (patrz rysunek 4.9). Zobaczysz tam identyfikator commita (ID), jak również nazwę wprowadzającego zmiany, datę itd. Zobaczysz również opis zmian oraz listę plików, które zostały zmienione (w tym wypadku *Book.cs*):



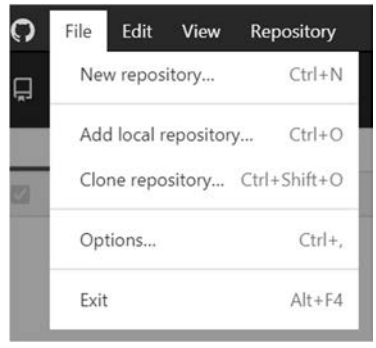
Rysunek 4.9. Visual Studio pokazujące, które pliki zmieniły się w commicie

Chcielibyśmy się dowiedzieć, co zmieniło się w *Book.cs*. W tym celu kliknij prawym przyciskiem *Book.cs* i wybierz *View History* (*Wyświetl historię*). W środkowym oknie zostanie wyświetlona jedna zmiana. Kliknij ją dwukrotnie, a zobaczysz klasę `Book`, która została dodana w tym commicie.

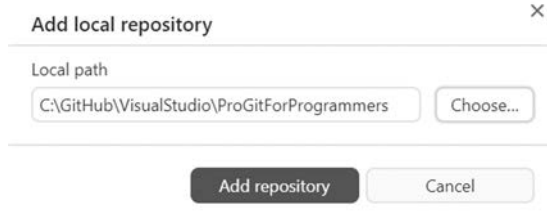
GitHub Desktop

Możemy otworzyć GitHub Desktop w tym samym katalogu. Kliknij *File* (plik) i wybierz *Add local repository…* (dodaj lokalne repozytorium…), tak jak pokazano na rysunku 4.10.

Kolejny krok polega na wskazaniu GitHub Desktop, gdzie znajduje się to repozytorium. Otworzy się okno dialogowe (rysunek 4.11), w którym będziesz mógł ręcznie wprowadzić ścieżkę albo kliknąć *Choose…* (wybierz…). Otworzy się wtedy eksplorator plików, gdzie będziesz mógł wybrać właściwy katalog. Kiedy już to zrobisz, kliknij *Add repository* (dodaj repozytorium).

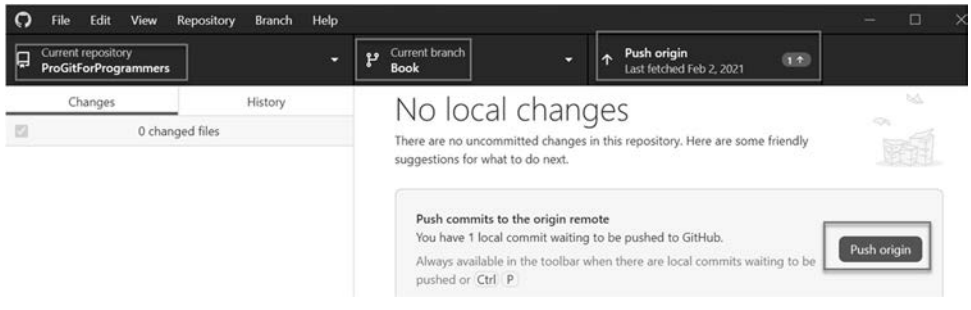


Rysunek 4.10. Otwieranie GitHub Desktop



Rysunek 4.11. Dodawanie lokalnego repozytorium

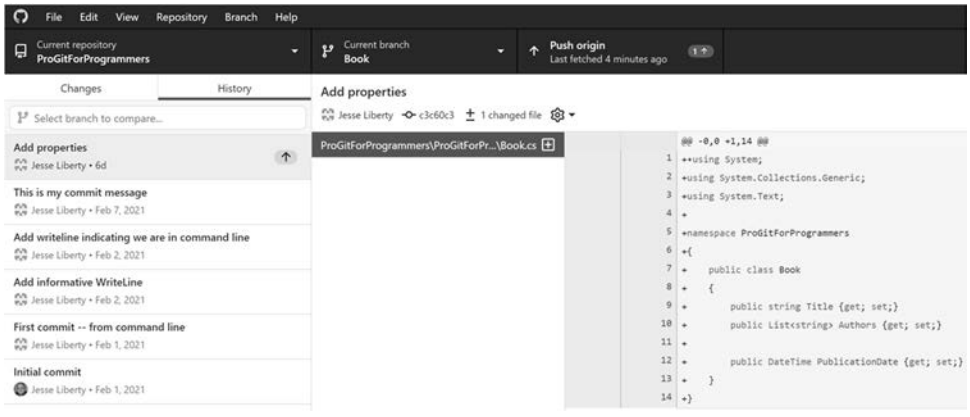
Zostaniesz przeniesiony na główną stronę (rysunek 4.12). Zauważ, że nadal znajdujesz się w repozytorium *GitForPorgrammers*, ale tym razem na gałęzi *Book*, oraz że masz jedną zmianę do wysłania. Znajduje się tam całkiem przydatny przycisk *Push origin* (wyślij do źródła) z wyjaśnieniem, że wciśnięcie tego przycisku spowoduje wysłanie zmian do źródła (serwera; Twojego repozytorium na GitHubie):



Rysunek 4.12. Pasek informacji w GitHub Desktop

Ponownie chcielibyśmy sprawdzić, co będziemy wysyłać. To żaden problem — po prostu kliknij *History* (historia), a zobaczysz historię zmian (rysunek 4.13) oraz zmiany dla dowolnego commita, który zaznaczysz:

Skoro już zobaczyliśmy, jak zarządzać commitami w wierszu poleceń, Visual Studio i GitHub Desktop, to pora na wysłanie zmian na serwer.



Rysunek 4.13. Historia w GitHub Desktop

Wyślijmy to w końcu

Wróćmy do wiersza poleceń (rysunek 4.14) i wyślijmy zmiany, które badaliśmy:

```
> git push
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 16 threads
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 601 bytes | 300.00 KiB/s, done.
Total 5 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To github.com:JesseLiberty/ProGitForProgrammers.git
   c9c89fb..c3c60c3  Book → Book
```

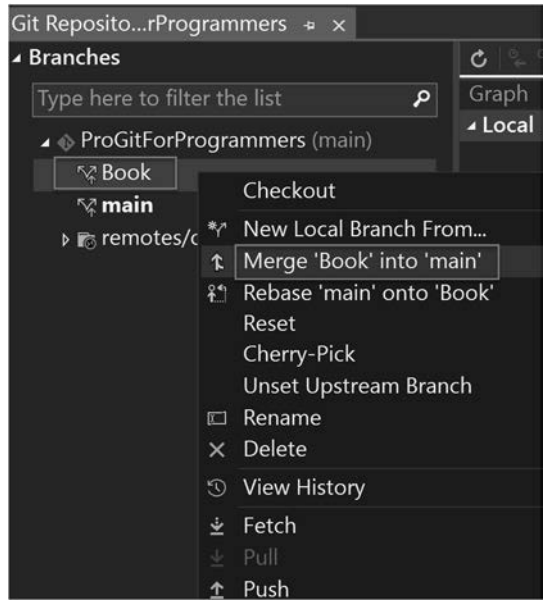
Rysunek 4.14. Wysyłanie w wierszu poleceń

Po przejściu do Visual Studio powinieneś zobaczyć *0 outgoing* (*0 wychodzących*). Dzieje się tak, ponieważ wysłałeś oczekujące zmiany. Podobnie będzie w GitHub Desktop. Przycisk *Push Origin* (wyślij do źródła) zamieni się na *Create Pull Request* (utwórz zgłoszenie) — prawdopodobnie kolejna rzecz w kolejce do zrobienia.

Skoro już uporaliśmy się ze wszystkim na gałęzi *Book*, pora (w końcu!) scalić ją z *main*.

Visual Studio

Naszym zadaniem jest scalenie *Book* z *main*. W tym celu otwórz Visual Studio, kliknij *Git*, a następnie *Manage Branches* (*Zarządzaj gałęziami*). Otworzy się okno *Branches* (*Gałęzie*). Kliknij prawym przyciskiem *main* i wybierz *Checkout* (*Wyewidencjonuj*). Możesz teraz scalić *Book* z *main*, co zobaczysz w menu kontekstowym, klikając prawym przyciskiem *Book*:



Rysunek 4.15. Scalanie w Visual Studio

Sztuczka polega na tym, żeby przejść na gałąź główną, kliknąć prawym przyciskiem *Book*, a opcja scalania stanie się dostępna, tak jak to widać na rysunku 4.15.

Konflikty scalania

Wróćmy teraz do wiersza poleceń i ściągnijmy (*pull*) zmiany, jako że nasza gałąź różni się od źródłowej (*origin*). Kiedy to zrobimy, otrzymamy informację o konflikcie scalania (*merge conflict*) w pliku *Program.cs* oraz o niepowodzeniu scalania (patrz rysunek 4.16). Git podpowie Ci, abyś rozwiązał te konflikty, a następnie zatwierdził zmiany. Pojawienie się konfliktów scalania podczas ściągania zmian nie jest typową sytuacją, ale jak widać, zdarza się. Rozwiążmy ten konflikt, a potem zajmiemy się bardziej typową sytuacją:

```
❯ git pull
Auto-merging ProGitForProgrammers/ProGitForProgrammers/Program.cs
CONFLICT (content): Merge conflict in ProGitForProgrammers/ProGitForProgrammers/Program.cs
Automatic merge failed; fix conflicts and then commit the result.
SESA560987@DESKTOP-D21661F  C:\GitHub\VisualStudio\ProGitForProgrammers  main ↑1 ↓4 +0 ~0 -0 !1 |
```

Rysunek 4.16. Konflikt scalania

Jest kilka sposobów na przeprowadzenie scalania, ale najprostszym jest użycie przeznaczonego do tego narzędzia (*merge tool*). Osobiście używam KDiff3 (*https://sourceforge.net/projects/kdiff3/*). Ponieważ używam go bardzo często, umieściłem go w moim pliku konfiguracyjnym:

```
git config --edit --global
```

```
[merge]
        tool = kdiff3
[mergetool]
        prompt = false
        keepBackup = false
        keepTemporaries = false
[mergetool "kdiff3"]
        path =  c:kdiff3\\kdiff3
```

Rysunek 4.17. Przeglądanie pliku konfiguracyjnego

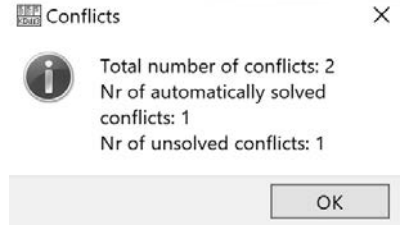
Plik konfiguracyjny przedstawiony na rysunku 4.17 konfiguruje KDiff3 jako narzędzie scalania i podpowiada Gitowi, gdzie się znajduje. Jedną z rzeczy, które lubię najbardziej w KDiff, jest to, że często jest w stanie rozwiązać problem za mnie.

Aby go uruchomić, wystarczy wpisać:

```
git mergetool
```

KDiff zostanie uruchomiony w celu rozwiązania konfliktu.

W naszym wypadku znalazł dwa problemy i udało mu się rozwiązać jeden z nich (rysunek 4.18):



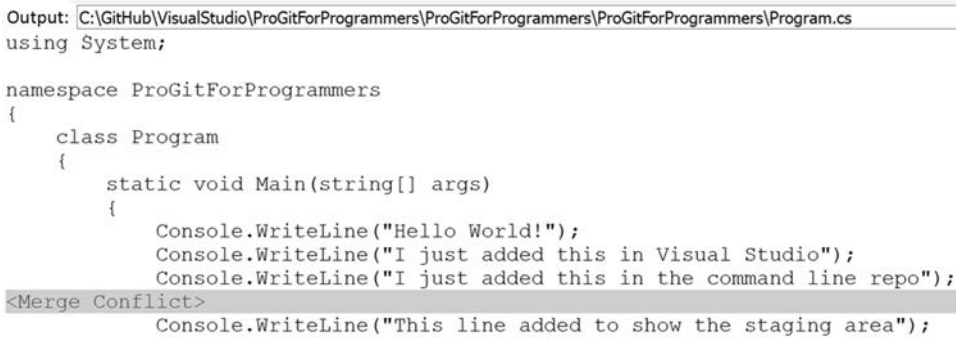
Rysunek 4.18. KDiff automatycznie rozwiązuje jeden konflikt

Następnie przenosi nas do okna podzielonego na kilka obszarów. Na rysunku 4.19 obszar u góry przedstawia konflikt:

Rysunek 4.19. Konflikt przedstawiony w KDiff

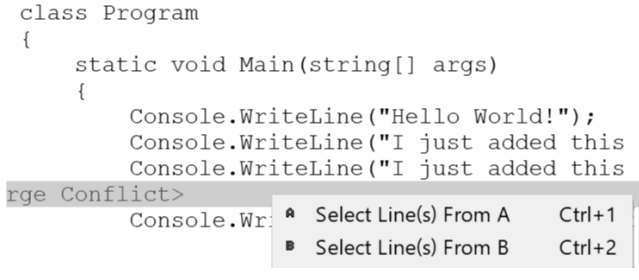
Po lewej stronie (*Local*) widzimy, że została dodana jedna linijka, natomiast po prawej (*Remote*) — dwie linijki. Najwidoczniej ktoś inny zmienił plik, który my edytowaliśmy, i teraz Git nie wie, co ma zrobić.

Poniżej na rysunku 4.20 znajduje się obszar przedstawiający pełny kontekst wraz z oznaczeniem miejsca, gdzie musisz dokonać wyboru, które linijki zatwierdzić:



Rysunek 4.20. KDiff przedstawia kontekst dla scalania

Kiedy klikniesz tę linijkę prawym przyciskiem (rysunek 4.21), będziesz mógł wybrać, czy chcesz zatwierdzić zmiany z okna z lewej (okno A), z prawej (okno B), czy może z obydwu (możesz wybrać kolejność, w jakiej zmiany z tych okien będą wprowadzone):

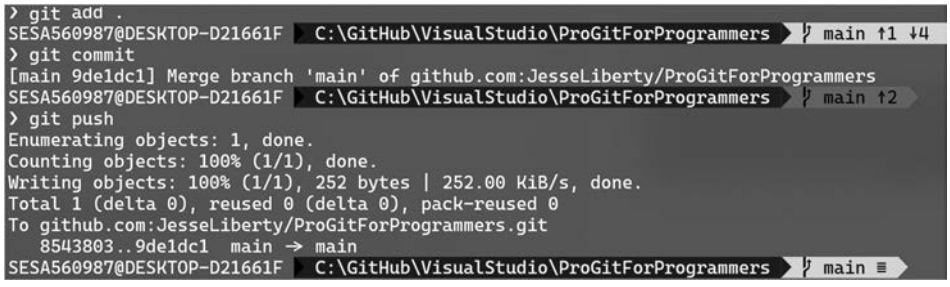


Rysunek 4.21. KDiff pyta, która wersja powinna być wybrana

Kiedy już skończysz, zapisz plik i zamknij KDiff. No i proszę, konflikt zniknął. Git pokaże Ci teraz zmiany, których dokonałeś i które powinny zostać scalone (rysunek 4.22):

Rysunek 4.22. Konsola potwierdzająca, że konflikty zostały rozwiązane

Teraz możesz dodać i zatwierdzić ten plik, a następnie wysłać go do źródła (rysunek 4.23):



Rysunek 4.23. Zatwierdzanie zmian i wysyłanie do źródła

Przekonaliśmy się, w jaki sposób KDiff i podobne programy mogą nam znacząco zmniejszyć ilość pracy potrzebnej do rozwiązywania konfliktów scalania.

Scalanie w wierszu poleceń

Konflikt scalania dużo częściej pojawia się, gdy scalasz zmiany lokalnie. Całkiem łatwo to zrobić. W wierszu poleceń przejdź na gałąź, z którą chcesz scalić zmiany (*main*), a następnie użyj komendy Gita `merge`:

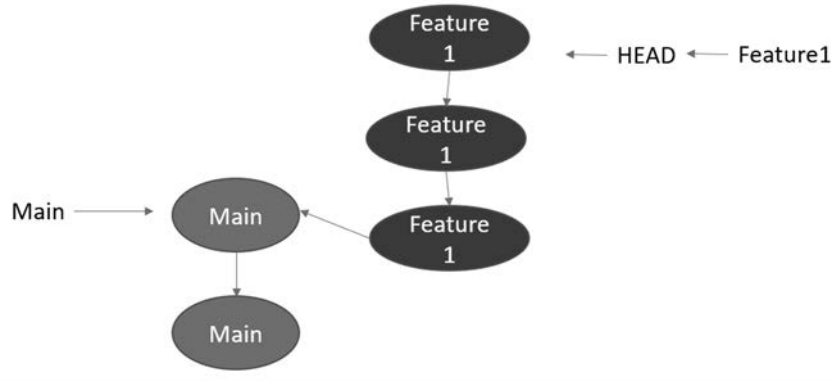
Rysunek 4.24. Polecenie merge

Na rysunku 4.24 widzimy, że Git użył strategii „rekursywnej” (*recursive*); przyspieszając w ten sposób scalanie.

Przewijanie

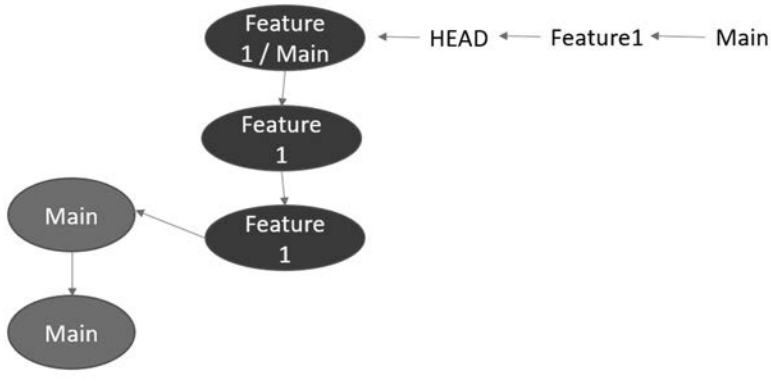
Bardzo często zdarza się, że Git używa scalania przewijanego[2] (*fast forward merge*). Przedstawię teraz, jak ono działa. Załóżmy, że mamy układ wejściowy, taki jak na rysunku 4.25:

[2] Określenie *przewijanie* nawiązuje do angielskiego zwrotu *fast forward*, oznaczającego szybkie przewinięcie utworu (piosenki, filmu) do przodu, przeskoczenie do jego kolejnej części — *przyp. tłum.*



Rysunek 4.25. Konflikt scalania

Chcemy następnie scalić *Feature1* z *Main*. Zauważ, że *Feature1* odgałęzia się z czubka *Main* (z ostatniego commita). W tym przypadku istnieje prosta ścieżka z pierwszego commita w *Main* do ostatniego w *Feature1*. Jak widać na rysunku 4.26, Gitowi nie pozostaje nic innego, jak przenieść wskaźnik *Main* na czubek *Feature1*, co da nam w rezultacie jedną gałąź (którą Git nazwie *Main*):

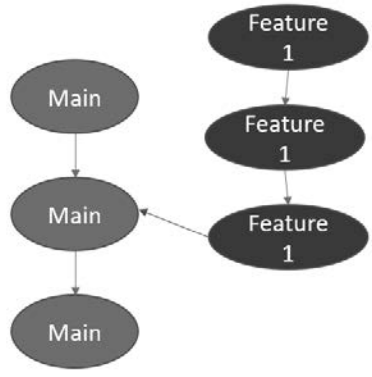


Rysunek 4.26. Przenoszenie wskaźnika

Wszystko, co należy tu zrobić, to przesunąć wskaźnik na ostatni commit. Dlatego nazywamy to *przewijaniem*.

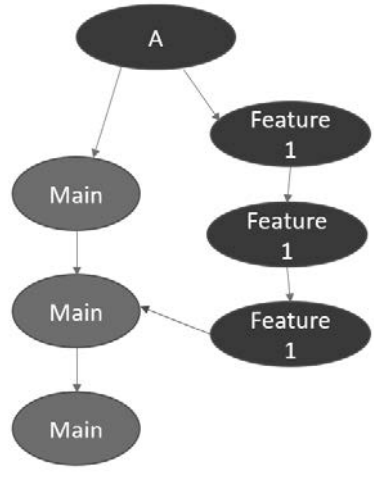
Prawdziwe scalanie

W poprzednim przykładzie *Feature1* odgałęził się z *Main* w miejscu, które wciąż jest czubkiem *Main*. Ale co się stanie, jeśli ktoś inny scali swoje zmiany z *Main*? Commit, od którego tworzyłeś nową gałąź, już nie będzie czubkiem gałęzi *Main*:



Rysunek 4.27. Feature1 nie odgałęzia się od czubka Main

W tym wypadku będziesz musiał zmienić bazę (zostanie to omówione w kolejnym rozdziale) albo wykonać „prawdziwe" scalanie:



Rysunek 4.28. Prawdziwe scalanie jest wymagane

Jak widać na rysunku 4.27, nie istnieje ścieżka z *Main* do czubka *Feature1*, która nie pomijałaby żadnych zmian (na przykład czubka *Main*), będziemy więc potrzebować dodatkowego commita, który zepnie razem obie gałęzie.

Zauważ, że to podejście wprowadza nowy commit (A), który istnieje tylko po to, aby scalić zmiany (patrz rysunek 4.28). Z biegiem czasu będziesz miał mnóstwo takich relatywnie nic niewnoszących commitów i będą one niejako zaśmiecać Twoją historię zmian. Rozwiązaniem tego problemu jest zmiana bazy, która zostanie przedstawiona w kolejnym rozdziale.

Gdy wykonujesz przewijanie lub prawdziwe scalanie, to nic nie zmieniasz; ty scalasz, a Git zajmuje się szczegółami.

Unikanie konfliktów

Unikanie konfliktów jest ogólnie dobrym podejściem, a w przypadku Gita nawet bardzo dobrym. Zamiast rozwiązywać wiele konfliktów naraz, lepiej, jeśli będziesz wyłapywał je na bieżąco (co będzie oznaczać, że będziesz musiał rozwiązywać jeden lub dwa w danej chwili). Jeśli pracujesz w zespole, niektórych konfliktów nie da się uniknąć. Istnieją jednak dwie sprawdzone zasady, które w znaczący sposób pomagają ograniczyć ilość pracy związanej z konfliktami:

- nie więcej niż jeden programista powinien modyfikować dany plik (o ile to możliwe),
- scalaj gałąź główną ze swoją gałęzią bardzo często.

Zasada #2 *nie* mówi, abyś scalał swoją gałąź funkcji z gałęzią główną, lecz na odwrót. Dzięki temu główny wątek pozostanie nietknięty, a Ty szybko wykryjesz potencjalne konflikty. Jeśli jakieś się pojawią, to błyskawicznie będziesz mógł je naprawić na swojej gałęzi i ruszyć dalej.

Podsumowanie

W tym rozdziale dowiedziałeś się:

- jak scalać gałęzie,
- jakie są różne typy scalania,
- jak scalać konflikty,
- w jaki sposób narzędzia, takie jak KDiff, mogą ułatwić życie,
- czym jest zgłoszenie,
- czym jest scalanie przewijane,
- czym jest prawdziwe scalanie.

Wyzwanie

Wcielisz się w role dwóch programistów pracujących nad tym samym projektem — narzędziem, które zawiera kalkulator oraz konwerter stopni Celsjusza na Fahrenheita. Jeśli będziesz miał drugiego programistę, który Ci w tym pomoże, tym lepiej.

Załóż nowe repozytorium i sklonuj je do dwóch różnych folderów. Niech jedna osoba doda do głównej gałęzi zalążek projektu *UtilityKnife*, zatwierdzi zmiany i je wyśle. Druga osoba powinna ściągnąć zmiany z gałęzi głównej.

OK, teraz kiedy obie osoby mają trochę kodu na gałęzi głównej, niech stworzą swoje własne gałęzie. Na jednej będziemy pracować nad kalkulatorem, a na drugiej nad konwerterem.

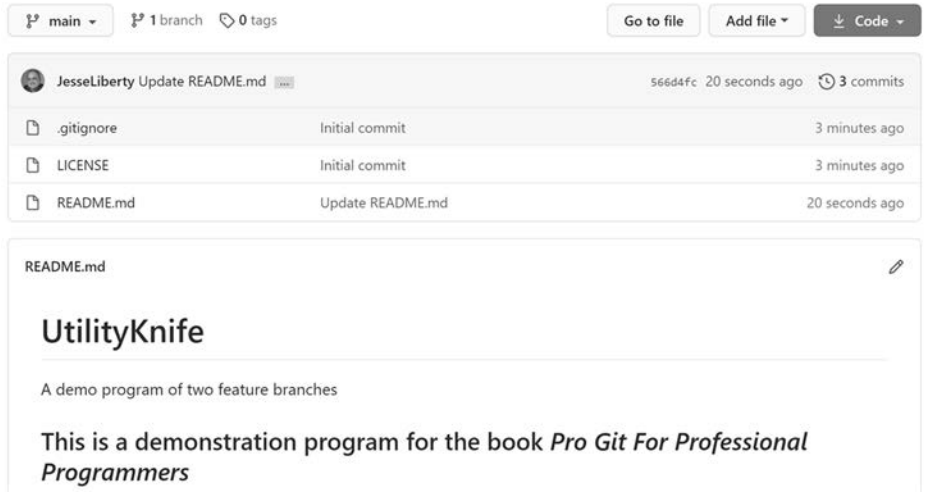
W trakcie pracy okaże się, że konwerter będzie chciał użyć niektórych metod kalkulatora. Staraj się unikać lub minimalizować konflikty, scalać często i rozwiązywać konflikty, gdy tylko się pojawią.

Rozwiązanie

Jak zwykle, nie ma jednego słusznego sposobu na rozwiązanie tego zadania. Oto jak ja nad nim pracowałem.

Zadanie #1: stwórz nowe repozytorium i sklonuj je do dwóch różnych folderów

Zauważ, że będziemy używać jednego repozytorium. Budujemy pojedynczą aplikację, ale przynajmniej na początku John będzie tworzył kalkulator, podczas gdy Sara zajmie się konwerterem temperatur. Cały program nazwiemy *UtilityKnife*. Aby rozpocząć, przejdźmy do *GitHub.com* i stwórzmy nasze nowe repozytorium, tak jak na rysunku 4.29:



Rysunek 4.29. Tworzenie nowego repozytorium

> Plik *README.md* jest napisany przy użyciu języka Markdown. Możesz się dowiedzieć więcej o jego składni w internecie, na przykład na stronie *https://www.markdownguide.org/cheat-sheet/*.

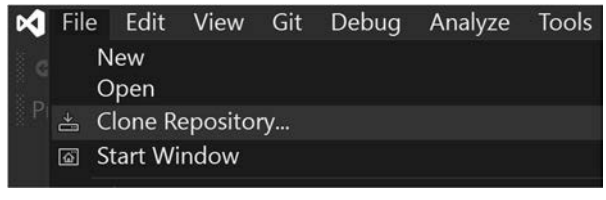
Następnie sklonujemy repozytorium do folderów (lub na dwa osobne komputery, jeśli jest dwóch lub więcej użytkowników). Utworzę katalog *John* i sklonuję do niego repozytorium.

```
SESA560987@DESKTOP-D21661F  C:\GitHub
❯ mkdir John


    Directory: C:\GitHub

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----          2/15/2021  8:12 AM                John

SESA560987@DESKTOP-D21661F  C:\GitHub
❯ cd John
SESA560987@DESKTOP-D21661F  C:\GitHub\John
❯ git clone git@github.com:JesseLiberty/UtilityKnife.git
Cloning into 'UtilityKnife'...
remote: Enumerating objects: 11, done.
remote: Counting objects: 100% (11/11), done.
remote: Compressing objects: 100% (11/11), done.
remote: Total 11 (delta 4), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (11/11), 5.20 KiB | 1.73 MiB/s, done.
Resolving deltas: 100% (4/4), done.
SESA560987@DESKTOP-D21661F  C:\GitHub\John
❯
```

Rysunek 4.30. Klonowanie przy użyciu wiersza poleceń

Jak widać na rysunku 4.30, John postanowił użyć wiersza poleceń. Sara natomiast lubi pracować w Visual Studio (rysunek 4.31).

Zacznij od kliknięcia *File* (*Plik*) i wybrania *Clone Repository…* (*Klonuj repozytorium*):



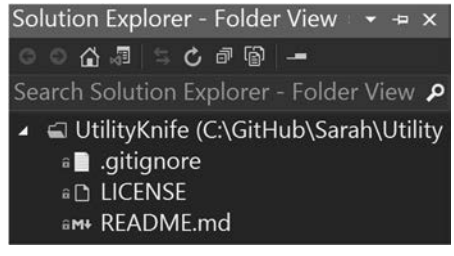
Rysunek 4.31. Otwieranie menu Gita

Jak widać na rysunku 4.32, zostanie otworzone okno dialogowe, w którym będziesz mógł wkleić link do repozytorium na GitHubie oraz ścieżkę do swojego nowego repozytorium.

Naciśnij przycisk *Clone* (*Klonuj*), a Visual Studio zajmie się przygotowaniem Twojego sklonowanego repozytorium.

W *Solution Explorer* (eksploratorze rozwiązań) możesz sprawdzić, że sklonowałeś repozytorium i że pobrałeś trzy pliki z GitHuba (rysunek 4.33).

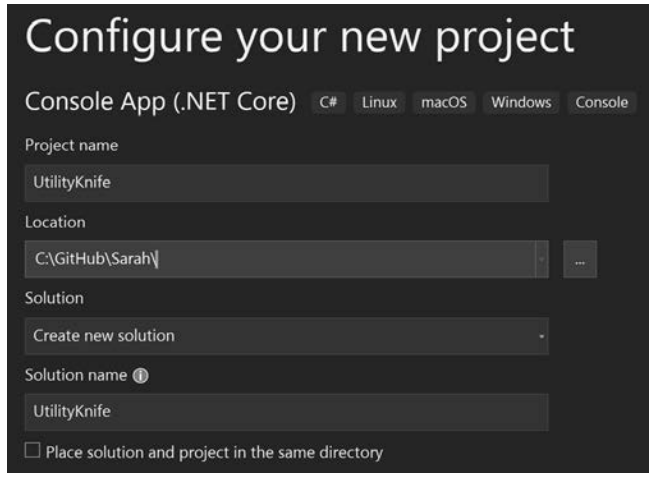
Rysunek 4.32. Klonowanie w Visual Studio



Rysunek 4.33. Eksplorator rozwiązań pokazuje rezultat klonowania

Zadanie #2: jedna osoba powinna dodać do głównej gałęzi zalążek UtilityKnife, zatwierdzić zmiany i je wysłać

Przyjmijmy, że Sara stworzy nowe rozwiązanie dla programu UtilityKnife w swoim katalogu (patrz rysunek 4.34):

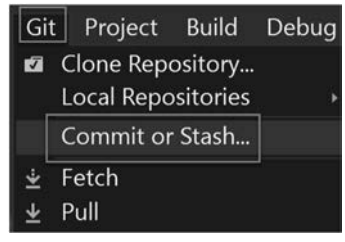


Rysunek 4.34. Tworzenie programu

Gdy projekt jest już stworzony, powinna dostosować *Program.cs*, tak aby był on szkieletem dla dalszych prac:

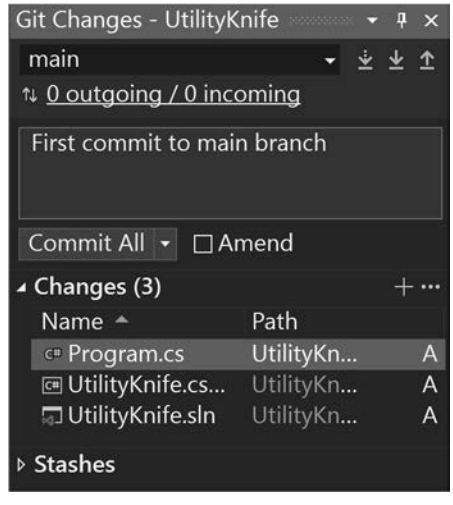
```
namespace UtilityKnife
{
  public static class Program
  {
        static void Main(string[] args)
        {
              //szkielet programu
        }
  }
}
```

Po dodaniu powyższego kodu Sara zatwierdzi zmiany, używając menu *Git*, które jest pokazane na rysunku 4.35:



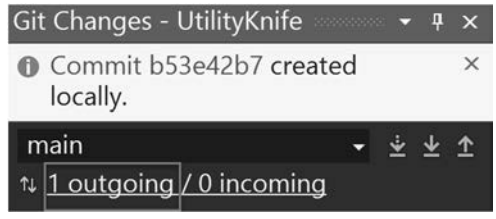
Rysunek 4.35. Menu Gita w Visual Studio

Jak widać na rysunku 4.36, zostanie otworzone okno zatwierdzania zmian, gdzie będziesz mógł wprowadzić opis zmian, a następnie kliknąć *Commit All* (*Zatwierdź wszystko*):



Rysunek 4.36. Okno zmian Gita w Visual Studio

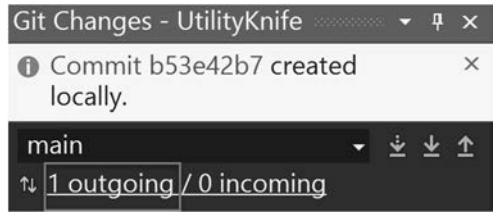
Gdy to zrobisz, z widoku znikną pliki oraz opis zmian. W ich miejsce pojawi się potwierdzenie utworzenia commita oraz informacja, że jeden commit jest gotowy do wysłania:



Rysunek 4.37. Potwierdzenie zatwierdzenia w Visual Studio

To właśnie zamierzaliśmy zrobić. Kliknij więc przycisk wysyłania (strzałka skierowana w górę) i wyślij zmiany do repozytorium na GitHubie.

Otrzymasz potwierdzenie (rysunek 4.38), że operacja się powiodła, oraz podpowiedź, że może zostać dla Ciebie utworzone zgłoszenie (*pull request*) — na ten moment możemy to pominąć:

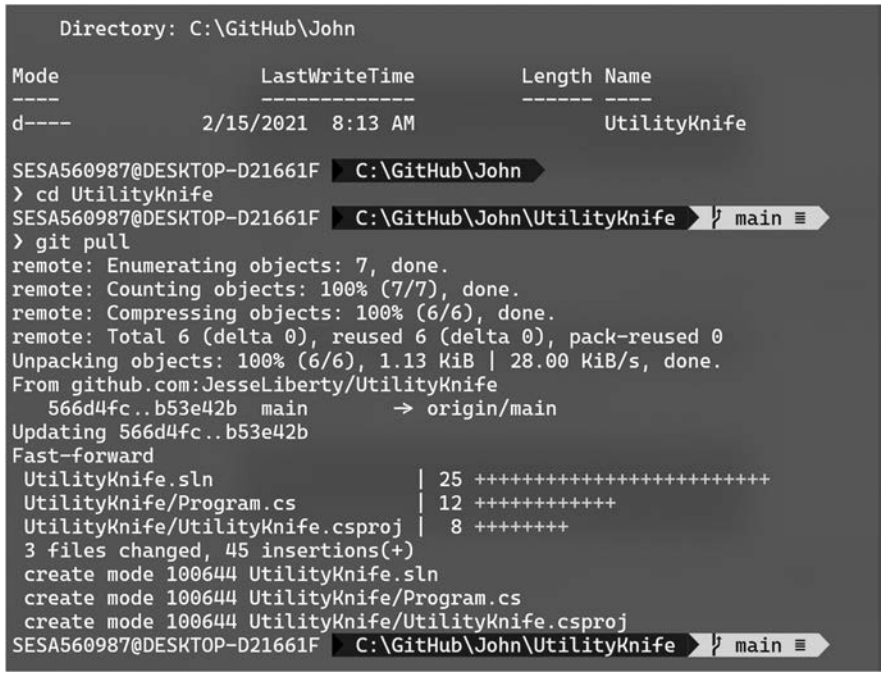


Rysunek 4.38. Po zatwierdzeniu zmian Visual Studio wskazuje, że jest jeden plik oczekujący na wysłanie

Sara zainicjowała główną gałąź i jest teraz gotowa, aby utworzyć gałąź nowej funkcji. Zanim się tym zajmiemy, niech John również ściągnie główną gałąź.

Na rysunku 4.39 widać dosyć złożony zrzut ekranu. Na początku widzimy, że wewnątrz *C:\Github\John* znajduje się folder *UtilityKnife*. Po przejściu do tego katalogu możemy wywołać `git pull`. W rezultacie otrzymamy pliki programu *UtilityKnife*.

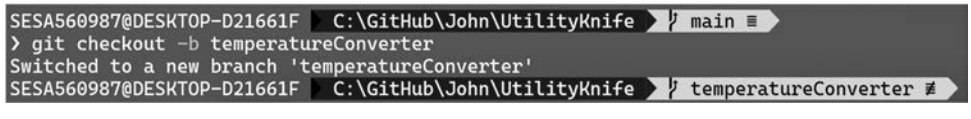
Teraz zarówno Sara, jak i John mają tę samą bazę programu UtilityKnife.



Rysunek 4.39. Pobieranie repozytorium ze źródła

Zadanie #3: każdy programista tworzy własną gałąź funkcji. Następnie każdy z nich umieszcza wstępne zmiany na swoich gałęziach i zatwierdza je często (częściej, niż robiłby to normalnie)

John, który używa wiersza poleceń, utworzy swoją gałąź za pomocą polecenia `checkout -b`, które jednocześnie tworzy nową gałąź i ustawia ją jako aktywną (rysunek 4.40):



Rysunek 4.40. Tworzenie nowej gałęzi w wierszu poleceń

Teraz jest gotowy, aby zacząć programować. Utwórzmy nowy folder, a następnie wewnątrz tego folderu umieśćmy szkielet naszej klasy i jej pierwszą metodę:

```
namespace UtilityKnife.Converters
{
  public class FahrenheitToCelesius
  {
```

```
            public double FahrenheitToCelesiusConverter(double FahrenheitTemp)
            {
                double _fahrenheithTemp = 0.0;
                double _celesius = 0.0;
                return _celesius;
            }
        }
    }
```

Zapiszmy i zatwierdźmy zmiany (rysunek 4.41):

```
❯ git status
On branch temperatureConverter
Untracked files:
  (use "git add <file>..." to include in what will be committed)

nothing added to commit but untracked files present (use "git add" to track)
SESA560987@DESKTOP-D21661F  C:\GitHub\John\UtilityKnife  temperatureConverter ≠ +1 ~0 -0 !
❯ git add .
SESA560987@DESKTOP-D21661F  C:\GitHub\John\UtilityKnife  temperatureConverter ≠ +1 ~0 -0 ~
❯ git commit
[temperatureConverter 738a95f] Creaete skeleton for FtoC converter
 1 file changed, 10 insertions(+)
 create mode 100644 UtilityKnife/Converters/FahrenheitToCelsius.cs
SESA560987@DESKTOP-D21661F  C:\GitHub\John\UtilityKnife  temperatureConverter ≠
```

Rysunek 4.41. Zatwierdzanie zmian w wierszu poleceń

Zacznijmy od sprawdzenia statusu. Pokazuje nam, że mamy jeden nieśledzony plik. Następnie dodajemy ten plik (pamiętaj, że `add .` oznacza dodanie wszystkich nieśledzonych oraz zmodyfikowanych plików do indeksu) i zatwierdzamy zmiany, dodając do nich opis. O nie, opis zmian ma w sobie literówkę! Możemy to naprawić, używając nowej komendy: `amend`. Ponieważ nie wysłaliśmy jeszcze zmian, wszystko, co musimy zrobić, to wpisać `--amend` i użyć `-m`, aby podać poprawioną wiadomość:

```
❯ git commit --amend -m "Create skeleton for FtoC converter"
[temperatureConverter 121012c] Create skeleton for FtoC converter
 Date: Mon Feb 15 15:29:40 2021 -0500
 1 file changed, 10 insertions(+)
 create mode 100644 UtilityKnife/Converters/FahrenheitToCelsius.cs
SESA560987@DESKTOP-D21661F  C:\GitHub\John\UtilityKnife  temperatureConverter ≠
```

Rysunek 4.42. Używanie flagi amend

Zwróć uwagę, że druga linijka na rysunku 4.42 odzwierciedla tę zmianę. Użyj `git log`, aby zobaczyć commity. Na rysunku 4.43 pokazano, że opis zmian jest poprawny:

```
❯ git log --oneline
121012c (HEAD -> temperatureConverter) Create skeleton for FtoC converter
```

Rysunek 4.43. Używanie logu w celu sprawdzenia zmian

John postanawia wysłać swoje zmiany z lokalnego repozytorium do źródłowego (czyli repozytorium GitHub). Kiedy spróbuje to zrobić, Git poinformuje go, że serwer nic nie wie o jego gałęzi (patrz rysunek 4.44), ale jednocześnie w ramach pomocy poda właściwą komendę:

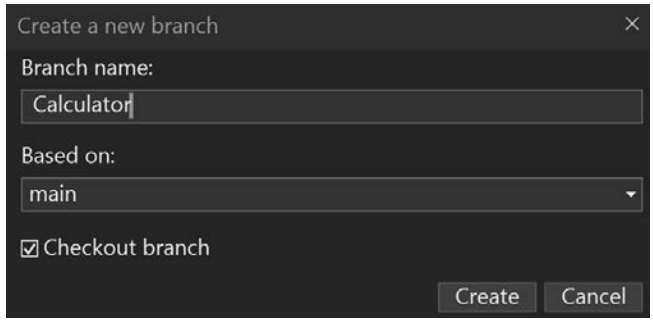
```
> git push
fatal: The current branch temperatureConverter has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin temperatureConverter

SESA560987@DESKTOP-D21661F  C:\GitHub\John\UtilityKnife  temperatureConverter ≠
> git push --set-upstream origin temperatureConverter
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 16 threads
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 584 bytes | 584.00 KiB/s, done.
Total 5 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote:
remote: Create a pull request for 'temperatureConverter' on GitHub by visiting:
remote:      https://github.com/JesseLiberty/UtilityKnife/pull/new/temperatureConverter
remote:
To github.com:JesseLiberty/UtilityKnife.git
 * [new branch]      temperatureConverter -> temperatureConverter
Branch 'temperatureConverter' set up to track remote branch 'temperatureConverter' from 'origin'.
SESA560987@DESKTOP-D21661F  C:\GitHub\John\UtilityKnife  temperatureConverter ≡
```

Rysunek 4.44. Nieudana próba wysłania. Git przychodzi na ratunek

W tym samym czasie Sara rozpoczęła pracę nad klasą `Calculator`.

Sara i kalkulator

Wewnątrz Visual Studio wybierz menu *Git*, a następnie *New Branch* (*Nowa gałąź*). Otworzy się okno dialogowe, takie jak na rysunku 4.45. Zakłada ono, że chcesz utworzyć rozgałęzienie od *main* (jeśli masz wiele gałęzi, możesz oczywiście odgałęzić się od dowolnej z nich):



Rysunek 4.45. Tworzenie nowej gałęzi

Sara jest teraz gotowa do działania. Cokolwiek teraz napisze, nie będzie to miało wpływu na kod Johna (ani na kod na gałęzi *main*). Możesz sprawdzić, że nie widzi ona nawet tego, co zrobił John. Obydwoje znajdują się na innych (i przez to odizolowanych) gałęziach funkcji.

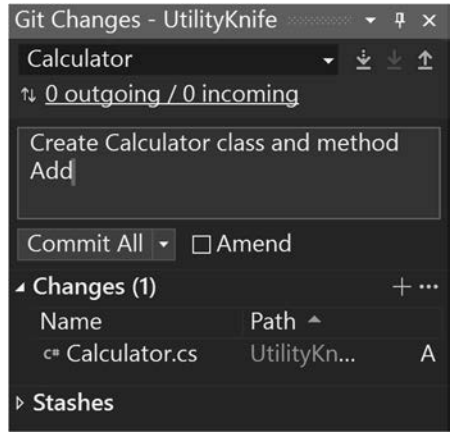
Sara będzie teraz mogła dodać szkielet klasy `Calculator` w osobnym folderze.

```
namespace UtilityKnife.Calculator
{
  public class Calculator
  {
```

```
            public static int Add (int x, int y)
            {
                    return x + y;
            }
    }
}
```

Sara zatwierdzi teraz zmiany, ale w przeciwieństwie do Johna nie umieści ich na serwerze. W ten sposób będą się one znajdowały tylko w jej lokalnym repozytorium.

Po wybraniu *Git/Commit or Stash* (*Git/Zatwierdź lub umieść w przechowalni*) Sara wprowadzi opis i kliknie *Commit All* (*Zatwierdź wszystko*):

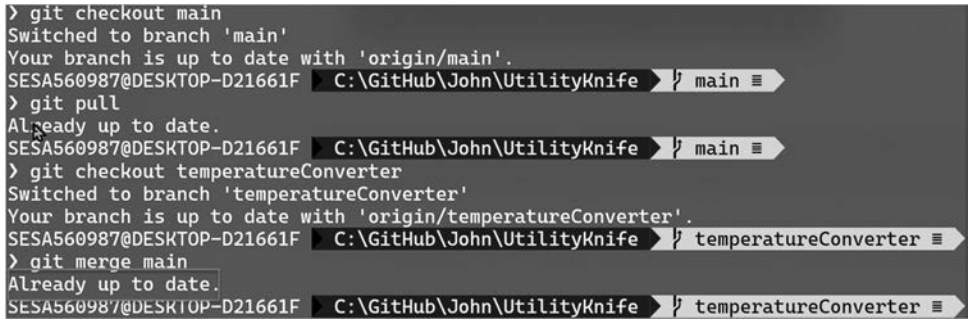


Rysunek 4.46. Zatwierdzanie wszystkich zmian

Jak zaznaczono wyżej, zmiany dodane na rysunku 4.46 trafią jedynie do jej lokalnego repozytorium.

Zadanie #4: scalaj główną gałąź z gałęzią funkcji często, aby mieć pewność, że jeśli pojawią się konflikty, to wyłapiesz je jak najwcześniej

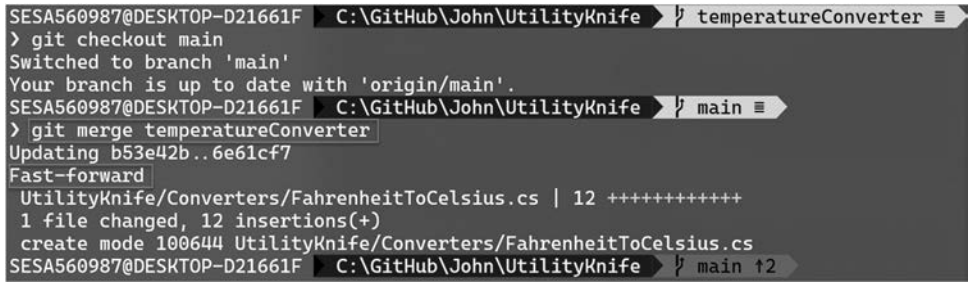
John chce scalić główną gałąź ze swoją, aby mieć pewność, że wcześnie wyłapie wszelkie błędy. Żeby to zrobić, musi przełączyć się na gałąź *main*, zaktualizować ją żądaniem `pull`, a następnie wrócić na swoją gałąź funkcji i wpisać `merge main`:



Rysunek 4.47. Main jest już aktualny

Jak widać na rysunku 4.47, wszystko poszło gładko. Główna gałąź nie zmieniła się od czasu, kiedy się od niej odgałęziliśmy, tak więc *temperatureConverter* jest w pełni aktualna.

Załóżmy teraz, że John chce scalić swoją gałąź z główną. Niezależnie od tego, czy jest to mądre, wystarczy, że scali gałęzie w odwrotnej kolejności niż przed chwilą, co zademonstrowano na rysunku 4.48:



Rysunek 4.48. Odwracanie kolejności scalania

Kluczowa jest tu linijka:

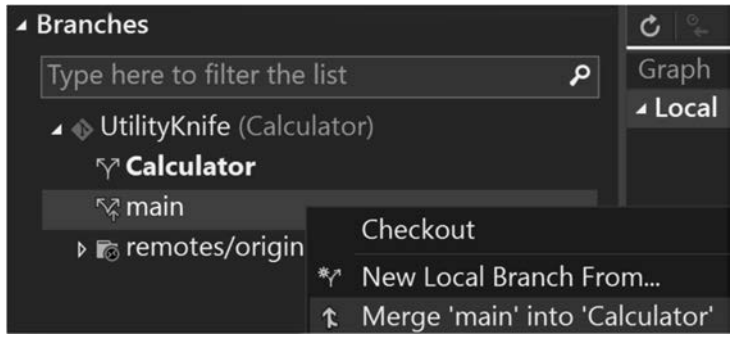
```
git merge temperatureConverter
```

Znajdujemy się na gałęzi głównej i to polecenie scala gałąź funkcji z gałęzią główną. Jak widzisz, Git jest w stanie wykonać scalanie przewijane, jak to zostało opisane w poprzednim rozdziale.

John może teraz śmiało kontynuować swoją pracę na istniejącej gałęzi funkcji lub utworzyć nową. Jeśli musiałby utworzyć zgłoszenie i czekać, aż jego PR zostanie zatwierdzony, zanim nastąpi scalenie, najrozsądniej byłoby stworzyć nową gałąź, odgałęziając się od *temperatureConverter*.

Sara miała chwilę przerwy, ale już wróciła do pracy. Będąc ostrożną, postanowiła najpierw scalić główną gałąź ze swoją, aby mieć pewność, że nie ma żadnych konfliktów. Pamiętaj, że John i Sara mogą ze sobą doskonale współpracować, ale nie będą się nawzajem informować za każdym razem, gdy któreś z nich zatwierdza zmiany lub scala gałęzie.

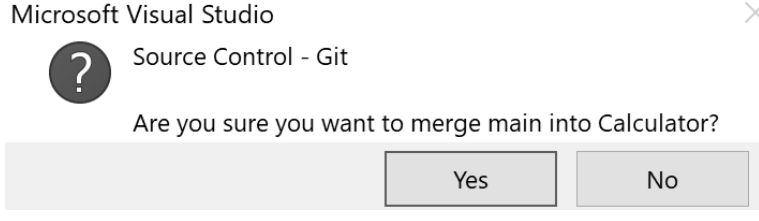
Aby zacząć, Sara przełącza się na gałąź główną i wykonuje `pull`, aby uzyskać najnowsze wersje plików z gałęzi głównej. Teraz przełącza się na *Calculator* i klika prawym przyciskiem *main*:



Rysunek 4.49. Scalanie gałęzi głównej z Calculator w Visual Studio

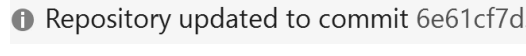
Jak widać na rysunku 4.49, wybierze *Merge 'main' into 'Calculator'* (*Scal gałąź 'main' z gałęzią 'Calculator'*). Ta operacja *nie* scali jej zmian z *main*, ale wybierze najnowszą wersję głównej gałęzi i scali ją z gałęzią funkcji.

Ponieważ Visual Studio jest bardzo ostrożne, zapyta, czy jesteś pewien, że chcesz scalić zmiany:



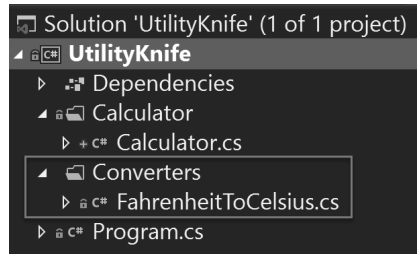
Rysunek 4.50. Visual Studio sprawdza, czy to, co zamierzasz zrobić, jest tym, co naprawdę chcesz zrobić

Naciśnięcie *Yes* (*Tak*) rozpocznie scalanie (rysunek 4.50). Jak zapewne pamiętasz, John wykonał pewną pracę i scalił swoją gałąź z gałęzią główną. Ponieważ nie było żadnych konfliktów, Visual Studio po prostu poinformuje Sarę, że scalanie się powiodło (rysunek 4.51):



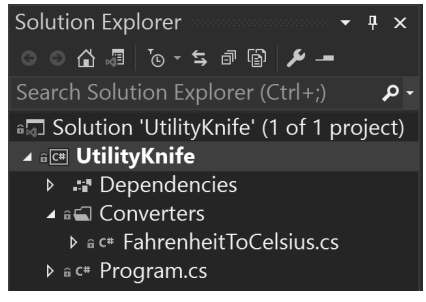
Rysunek 4.51. Visual Studio sygnalizuje powodzenie

Oczywiście scalenie gałęzi głównej z *Calculator* zmieni tę gałąź (co widać na rysunku 4.52), dodając do niej wszytko, co znajdowało się w *main*. Kluczowymi zmianami w *main* są te scalone przez Johna, co możemy zauważyć na gałęzi *Calculator*:



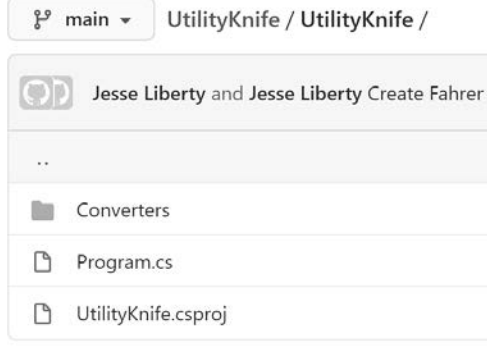
Rysunek 4.52. Przeglądanie rezultatów scalania w Visual Studio

Wspomnijmy, że Sara nie scaliła swojego kodu z *main*, tak więc John nie ma pojęcia o istnieniu klasy `Calculator`, jak również nie jest w stanie dostać się do jej kodu. Jeśli otworzymy Visual Studio na gałęzi Johna, zobaczymy folder *Converters*, ale nie będzie śladu po *Calculator* (rysunek 4.53):



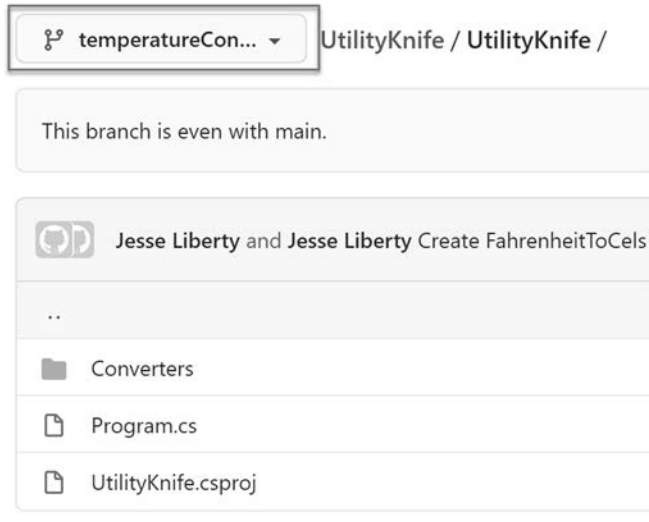
Rysunek 4.53. Gałąź Johna bez folderu Calculator

Zatrzymajmy się na moment i zastanówmy się, co właściwie dzieje się na GitHubie. Sara zatwierdziła swoje zmiany, ale ich nie wysłała, więc GitHub nic nie wie o jej gałęzi. John wysłał swoje zmiany i scalił je z gałęzią główną. Możemy oczekiwać, że na GitHubie będą się znajdować dwie gałęzie, główna i Johna; co więcej, na ten moment obie powinny być identyczne, a Sara nie powinna mieć gałęzi na GitHubie:



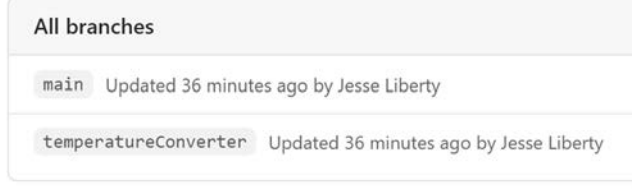
Rysunek 4.54. Gałęzie w repozytorium źródłowym

Jak widać na rysunku 4.54, na gałęzi *main* powinien się znajdować folder *Converters* (w związku ze scalaniem Johna), ale *Calculator* już nie (Sara jeszcze nie scaliła swoich zmian). Gałąź Johna (*temperatureConverter*) powinna być identyczna (rysunek 4.55):



Rysunek 4.55. Zmiana gałęzi w repozytorium źródłowym

Aby zamknąć ten temat, możemy poprosić GitHuba, aby wylistował nam wszystkie znane mu gałęzie (rysunek 4.56):

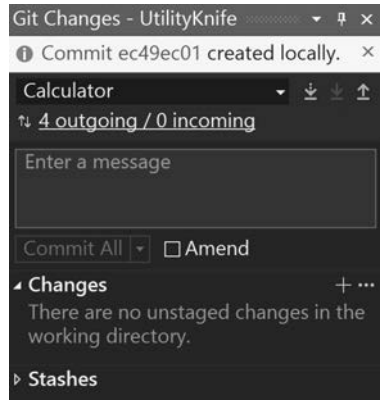


Rysunek 4.56. GitHub wyświetla listę wszystkich gałęzi

> Wszystkie gałęzie są oznaczone jako zaktualizowane przeze mnie, ponieważ Sara i John nie istnieją.

Zadanie #5: John buduje konwerter temperatur. Pozwólmy mu „pożyczyć" kod kalkulatora. Sprawdźmy, czy nie ma konfliktów scalania

W kolejnych czterech zatwierdzeniach zmian Sara rozszerza kalkulator o operacje odejmowania, mnożenia, dzielenia i dzielenia całkowitoliczbowego. Jak pokazano na rysunku 4.57, jeszcze nie wysłała swoich zmian:

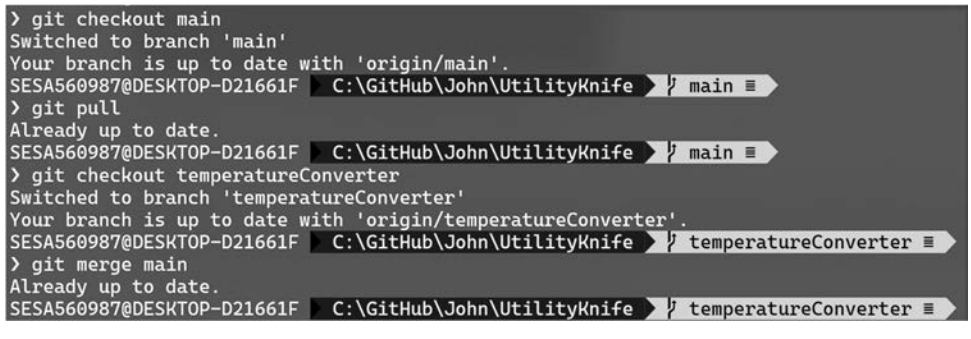


Rysunek 4.57. Rozszerzanie liczby funkcji kalkulatora

Wzór na zamianę stopni Fahrenheita na Celsjusza wygląda następująco:

$$(F–32)\cdot{}^{5}/_{9}$$

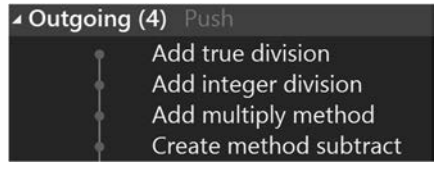
W ramach przypadku testowego John chce zamienić 212° Fahrenheita (punkt wrzenia wody) i oczekuje, że otrzyma z powrotem 100° Celsjusza. Aby to zrobić, mógłby skorzystać z wbudowanych operatorów odejmowania i dzielenia, jednak postanawia, że skorzysta z przygotowanego przez Sarę kalkulatora. W swoim pierwszym kroku, zademonstrowanym na rysunku 4.58, chce scalić gałąź główną ze swoją:



Rysunek 4.58. Scalanie main z gałęzią roboczą

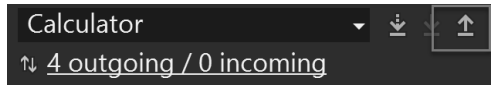
Gałąź *main* jest aktualna i nie ma żadnych różnic pomiędzy nią a *temperatureConverter*. John wciąż nie ma funkcji, których potrzebuje. To mu podpowiada, że funkcje kalkulatora, które są mu niezbędne, nie są jeszcze wysłane do GitHuba. Mógłby zadzwonić do Sary i poprosić ją, żeby je wysłała, tak aby mógł je ściągnąć. Mogłaby je również scalić z *main*, a wtedy John zaktualizowałby tę gałąź. Jednak Sara nie jest jeszcze gotowa, aby scalić swoje zmiany z *main*, więc zgodziła się wysłać swoją gałąź.

Jak widać na rysunku 4.59, Sara ma cztery wychodzące commity (czyli takie, które nie zostały wysłane do repozytorium źródłowego):



Rysunek 4.59. Visual Studio informujące o czterech niewysłanych commitach

Aby je wysłać, wystarczy, że kliknie skierowaną w górę strzałkę (rysunek 4.60):



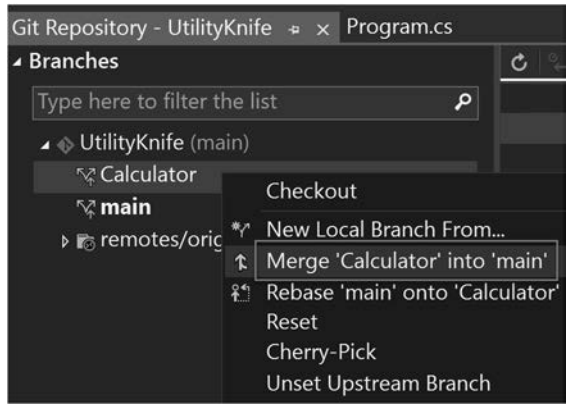
Rysunek 4.60. Przycisk wysyłania zmian w Visual Studio

John próbuje teraz pobrać te zmiany, ale ponownie zderza się ze ścianą (rysunek 4.61):

Rysunek 4.61. Gałąź Calculator nie jest widoczna

Jego lokalne repozytorium nigdy nie słyszało o gałęzi *Calculator*. Istnieje kilka sposobów na poradzenie sobie z tym, ale najprostszym jest poproszenie Sary, aby scaliła wyniki swojej pracy z *main* (rysunek 4.62).

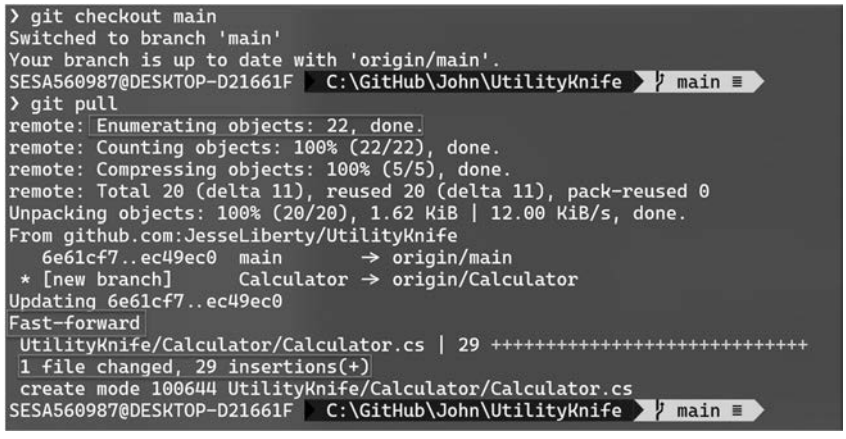
> Gdy Sara scali *Calculator* z *main*, zrobi to *lokalnie*. Wciąż będzie musiała wysłać te zmiany do źródła, jeśli mają one pomóc Johnowi. Może to zrobić w taki sposób, w jaki wysłałaby dowolne inne zmiany.



Rysunek 4.62. Scalanie Calculator z main

Po operacjach przedstawionych na rysunku 4.62 John jest gotowy, aby ściągnąć zmiany. Kiedy już to zrobi, zorientuje się, że Sara użyła liczb całkowitych, a on musi pracować na zmiennoprzecinkowych. Zmienia więc klasę `Calculator`, aby wykorzystywała typ `double`, i dodatkowo nadaje wszystkim metodom (oraz klasie) modyfikator `static` (jeśli nie jesteś zaznajomiony z C#, to nie martw się, co to znaczy; najważniejsze jest, że wprowadził zmiany):

```
namespace UtilityKnife.Calculator
{
  public static class Calculator
  {
        public static double Add(double x, double y)
        {
              return x + y;
        }

        public static double Subtract(double x, double y)
        {
              return x - y;
        }

        public static double Multiply(double x, double y)
        {
        return x * y;
        }

        public static int Division (int x, int y)
        {
              return x / y;
        }

        public static double Division (double x, double y)
        {
              return x / y;
        }
  }
}
```



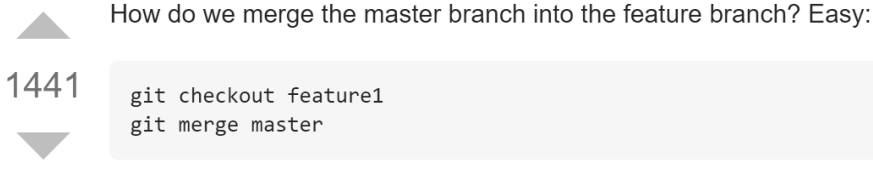
Rysunek 4.63. Ściąganie zmian dla gałęzi

Druga linijka na rysunku 4.63 wskazuje, że zaczynamy z gałęzią główną, która jest na bieżąco ze zmianami w *origin/master*. Jednak kiedy wykonamy `pull`, lokalny Git znajdzie obiekty, które może pobrać do gałęzi głównej. Wymienione są 22 obiekty. Czemu aż 22, skoro są tylko 4 commity? Niektóre z tych obiektów są używane wewnętrznie przez Gita.

Dalej widzimy informację o scalaniu przewijanym, a następnie, że zostało dodanych 29 linii, żadna nie była zmodyfikowana ani usunięta (jeśli policzysz znaki +, zobaczysz, że będzie ich 29). Na końcu mamy potwierdzenie, że jeden plik został zmieniony poprzez dodanie 29 nowych linii.

John ma już niemal wszystko gotowe. Jego lokalna kopia *main* ma teraz wszystkie niezbędne zmiany, ale są one nadal na niewłaściwej gałęzi. Rozwiązaniem w tej sytuacji jest scalenie *main* z *temperatureConverter*.

Ponieważ kolejność, w jakiej scala się ze sobą gałęzie, ma znaczenie, zawsze sprawdzam ją na Stack Overflow (rysunek 4.64):



Rysunek 4.64. Porada ze Stack Overflow

Są to dokładnie te kroki, które musi wykonać John:

```
> git checkout temperatureConverter
Switched to branch 'temperatureConverter'
Your branch is up to date with 'origin/temperatureConverter'.
SESA560987@DESKTOP-D21661F  C:\GitHub\John\UtilityKnife  temperatureConverter ≡
> git merge main
Updating 6e61cf7..ec49ec0
Fast-forward
 UtilityKnife/Calculator/Calculator.cs | 29 ++++++++++++++++++++++++++++
 1 file changed, 29 insertions(+)
 create mode 100644 UtilityKnife/Calculator/Calculator.cs
SESA560987@DESKTOP-D21661F  C:\GitHub\John\UtilityKnife  temperatureConverter ↑4
```

Rysunek 4.65. Scalanie

Ostatnie dwie linie na rysunku 4.65 wskazują, że klasa `Calculator` została dodana poprzez scalenie oraz że na *temperatureConverter* znajdują się dwa commity do wysłania.

Szybki rzut oka na historię pokaże nam, że *HEAD*, *origin/temperatureConverter*, *origin/main*, *origin/HEAD* i *origin/Calculator* wskazują na ten sam commit co *main*! Dzięki temu, jak widać na rysunku 4.66, gałąź Johna ma teraz dostęp do klasy `Calculator`:

```
> git log --oneline
ec49ec0 (HEAD -> temperatureConverter, origin/temperatureConverter, origin/main, origin/HEAD,
origin/Calculator, main) Add true division
```

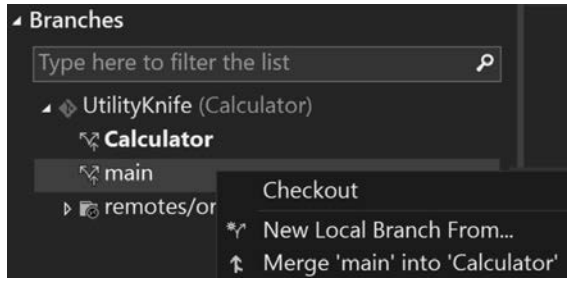
Rysunek 4.66. Dostęp do Calculator

Może teraz powrócić do swojego programu i użyć wcześniej zdefiniowanych metod statycznych:

```
namespace UtilityKnife.Converters
{
  public class FahrenheitToCelsius
  {
        public double FahrenheitToCelsiusConverter(double fahrenheitTemp)
        {
              double _celsius = 0.0;

              // (F – 32) * 5/9

              var step1 = Calculator.Calculator.Subtract(
                    fahrenheitTemp, 32);
              var step2 = Calculator.Calculator.Multiplication(
                    step1, 5.0);
              _celsius = Calculator.Calculator.Division(step2, 9.0);
              return _celsius;
        }
  }
}
```

Zgadzam się, że jest to okropnie brzydkie rozwiązanie, ale działa i jednocześnie demonstruje, że *temperatureConverter* Johna może używać kodu `Calculator`. Co więcej, John może edytować `Calculator`. Zobaczymy, co się wydarzy, gdy to wszystko zostanie scalone.

Gdy przeskakiwaliśmy w tę i z powrotem między Johnem i Sarą, wykonałem nieco pracy w folderze Sary. Bez obaw, musimy po prostu zatwierdzić te zmiany. O rety, zmiany zostały zrobione na *main*. Musimy posprzątać ten bałagan. Na początek na komputerze Sary scalmy *main* z *Calculator*:

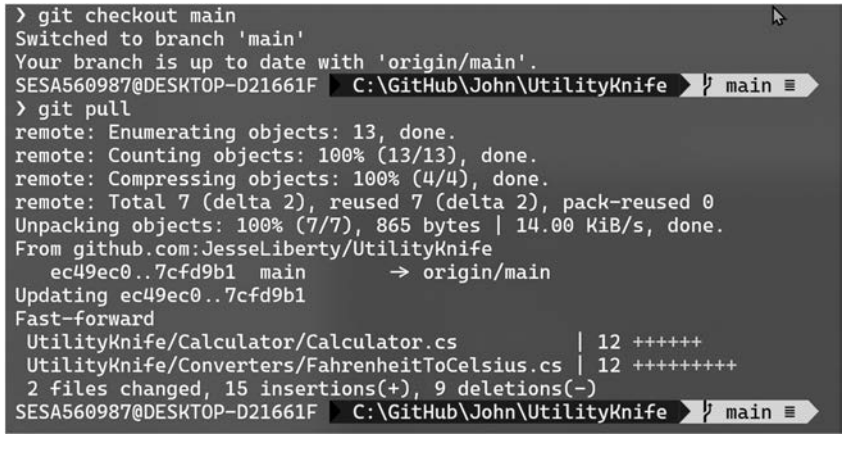


Rysunek 4.67. Scalanie main z Calculator w Visual Studio

Na rysunku 4.67 są przedstawione dokładnie takie same operacje co poprzednio, z tym że teraz aktywną gałęzią jest *Calculator*. Klikamy prawym przyciskiem *main*, aby użyć opcji *Merge 'main' into 'Calculator'* (*Scal gałąź 'main' z gałęzią 'Calculator'*). Teraz, aby mieć pewność, że wszystko jest tip-top, scal *Calculator* z gałęzią główną.

Na tę chwilę znajdujące się u Sary gałęzie *main* i *Calculator* są identyczne. Jednak John nadal nie ma tego, czego potrzebuje. Sara może więc teraz wysłać gałąź *main* do źródła zwykłym poleceniem `push`.

John może teraz pobrać *main*, który będzie zawierał potrzebne mu zmiany:



Rysunek 4.68. Pobieranie main ze zmianami

Fantastycznie, John ma niezbędne zmiany (rysunek 4.68), ale muszą się pojawić również na jego gałęzi. Jak pokazano na rysunku 4.69, to żaden problem. Wystarczy, że scalimy główną gałąź z *temperatureConverter*:

```
> git checkout temperatureConverter
Switched to branch 'temperatureConverter'
Your branch is up to date with 'origin/temperatureConverter'.
SESA560987@DESKTOP-D21661F  C:\GitHub\John\UtilityKnife  temperatureConverter ≡ [08:27]
> git merge main
Updating ec49ec0..7cfd9b1
Fast-forward
 UtilityKnife/Calculator/Calculator.cs          | 12 ++++++
 UtilityKnife/Converters/FahrenheitToCelsius.cs | 12 +++++++++
 2 files changed, 15 insertions(+), 9 deletions(-)
SESA560987@DESKTOP-D21661F  C:\GitHub\John\UtilityKnife  temperatureConverter ↑1[08:27]
```

Rysunek 4.69. Scalanie main z temperatureConverter

Spójrzmy na konwerter Fahrenheita przygotowany przez Johna i sprawdźmy, czy jest aktualny.

```
namespace UtilityKnife.Converters
{
  public class FahrenheitToCelsius
  {
                  public double FahrenheitToCelsiusConverter(double
fahrenheitTemp)
                  {
                           double _celsius = 0.0;

                           // (F – 32) * 5/9

                           var step1 = Calculator.Calculator.Subtract(
                           fahrenheitTemp, 32);
                           var step2 = Calculator.Calculator.Multiplication(
                           step1, 5.0);
                           _celsius = Calculator.Calculator.Division(step2,
9.0);
                           return _celsius;
                  }
  }
}
```

Możemy teraz przetestować jego działanie, podając metodzie wartość 212 i oczekując, że zwróci nam 100. Spójrzmy teraz na program, który posłuży do tego celu:

```
using System;
using UtilityKnife.Converters;

namespace UtilityKnife
{
  public static class Program
  {
         static void Main(string[] args)
         {
                  var converter = new FahrenheitToCelsius();
                  var celsius = converter.FahrenheitToCelsiusConverter(212.0);
                  Console.WriteLine($"Fahrenheit temp of 212 is {celsius}.");
         }
  }
}
```

Uruchommy nasz program, tak jak pokazano na rysunku 4.70:

Rysunek 4.70. Testowanie programu

Zakończyliśmy nasze wyzwanie i sprawnie zarządzaliśmy naszymi gałęziami. Co ważniejsze, nasz program działa!

Kluczowym elementem każdego projektu programistycznego jest kontrola wersji!

Kontrola wersji jest warunkiem sukcesu w każdym projekcie programistycznym. Im większy zespół i im bardziej złożony projekt, tym ważniejsze jest skuteczne zarządzanie wersjami. Do tych celów najczęściej używa się Gita. To popularne i dynamicznie rozwijane darmowe oprogramowanie. Zawiera wiele przydatnych narzędzi, pozwala też na wyrafinowaną konfigurację i dostosowanie do szczególnych potrzeb. Jeśli chcesz w pełni skorzystać z jego potencjału, musisz nabrać wprawy, pewności siebie i dobrze poznać poszczególne elementy Gita.

Oto opracowany z myślą o programistach praktyczny przewodnik, dzięki któremu szybko zaczniesz się posługiwać Gitem. Zawiera wszystkie informacje niezbędne do dogłębnego zrozumienia specyfiki tego narzędzia. Dowiesz się, w jaki sposób je zainstalować i skonfigurować, nauczysz się tworzyć i klonować repozytoria, a także zapoznasz się z narzędziami GUI Gita i zrozumiesz zasady pracy z gałęziami. Poznasz techniki rozwiązywania konfliktów scalania i korzystania z historii zmian. W książce omówiono polecenia potrzebne do zarządzania repozytorium, wyjaśniono też kwestie dotyczące bisekcji, polecenia blame i wielu innych narzędzi ułatwiających naprawianie błędów i rozwiązywanie typowych problemów.

Dzięki książce dowiesz się, jak:

- zacząć pracę z Gitem
- tworzyć repozytoria lokalne i zdalne
- używać gałęzi, zarządzać nimi i scalać je do gałęzi głównej
- rozwiązywać konflikty scalania
- mieć pełną kontrolę nad wszystkimi informacjami w repozytorium
- naprawiać błędy w Gicie

Jesse Liberty jest głównym inżynierem oprogramowania w firmie StoryBoardThat. Ma przeszło trzydziestoletnie doświadczenie w programowaniu; obecnie korzysta głównie z .NET 5/6, C# 9 i powiązanych z nimi technologii. Jest autorem ponad dwudziestu książek o programowaniu. Często występuje na międzynarodowych konferencjach, prowadzi także podcast — *Yet Another Podcast*.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-283-8914-4
9 788328 389144
Cena: 69,00 zł
Packt