Git - Working with changes

Spis treści

Klonowanie repozytorium

Wspomniałem wcześniej, że materiały te są przygotowywane w ten sposób, żeby w jak największym stopniu pokrywały się z praktyką. Powiedzieliśmy sobie również wcześniej o tym, jak stworzyć repozytorium lokalne i w jaki sposób można stworzyć repozytorium zdalne. W dalszych przykładach będziemy skupiać się na pracy na obu rodzajach repozytoriów, korzystając z obu jednocześnie. Odrzucimy wariant pracy tylko i wyłącznie z repozytorium lokalnym, gdyż w praktyce jest to mało prawdopodobne, że będziesz pracować z **Git**em tylko i wyłącznie na swoim komputerze, bez synchronizacji Twoich zmian z serwerem.

W przykładzie, gdzie stworzyliśmy repozytorium zdalne, zostało ono utworzone od razu na **GitHub**ie. W praktyce wyglądałoby to w podobny sposób. Developer, inicjując nowy projekt, zacząłby od stworzenia repozytorium na platformie, która jest używana w firmie (w jednej firmie może to być **GitLab**, a w innej **Bitbucket**). Po stworzeniu takiego repozytorium możemy je **sklonować** do siebie na komputer (o tym będzie zaraz).

Gdy dostaniesz już pracę i zostaniesz przydzielony/-a do jakiegoś projektu, to istnieje ogromne prawdopodobieństwo, że będziesz pracować nad projektem, który już istnieje i jest w fazie developmentu. Pierwszego dnia w firmie prawdopodobnie dostaniesz komputer, będziesz musiał/-a zainstalować na nim Javę i pojawia się pytanie, skąd wziąć kod źródłowy projektu, nad którym masz pracować? Kod źródłowy takich projektów jest przechowywany w repozytoriach.

To, co będziesz musiał/-a zrobić, żeby rozpocząć pracę z projektem, to będzie pobranie

kodu źródłowego, który już istnieje. Projekty takie często mają tysiące setki albo tysiące klas lub różnych plików. Kopiowanie tego manualnie nie wchodzi w grę. Całe szczęście **Git** przychodzi nam z pomocą i możemy wykorzystać **klonowanie**.

Żeby pobrać kod ze zdalnego repozytorium, należy takie repozytorium **sklonować**. Klonowanie będzie oznaczało stworzenie repozytorium lokalnego na podstawie informacji zawartych w repozytorium zdalnym. Repozytorium lokalne będzie zawierało te same informacje o historii zmian w projekcie, które są zawarte w repozytorium zdalnym. Czyli po skończeniu klonowania, na Twoim komputerze pojawią się wszystkie pliki potrzebne do pracy z projektem i cała historia ich zmian. Inaczej można to rozumieć w ten sposób, że od tego momentu, masz u siebie na komputerze kompletny backup całego projektu.

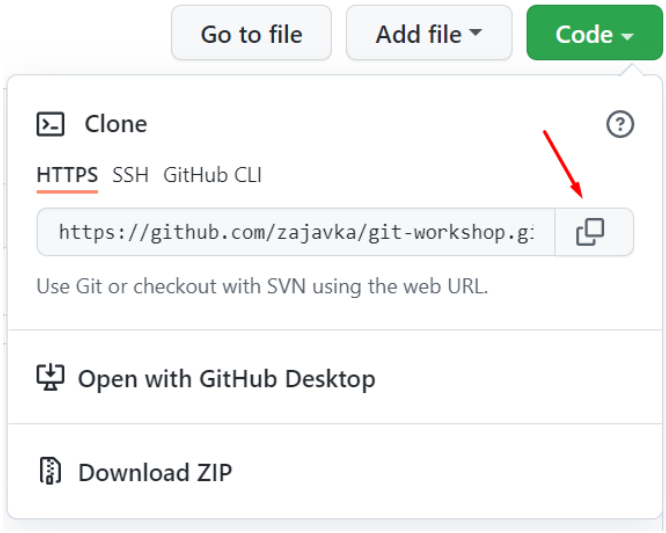
Możemy założyć, że stan repozytorium lokalnego będzie taki sam jak stan repozytorium zdalnego od razu po **sklonowaniu**. Jeżeli chwilę po **sklonowaniu**, ktoś zaktualizuje stan repozytorium zdalnego, to musimy pobrać już te zmiany manualnie. Nie zostaną one aktualizowane automatycznie. Czyli można powiedzieć, że repozytorium lokalne trzeba co jakiś czas synchronizować z repozytorium zdalnym.

Jak sklonować repozytorium?

Gdy wejdziesz na główną stronę swojego zdalnego repozytorium, czyli:

```
https://github.com/<twoja_nazwa_uzytkownika>/<twoja_nazwa_repozytorium>
```

To znajdziesz tam zielony przycisk z napisem "Code":



Obraz 1. GitHub clone repository

Gdy go wciśniesz, to mając zaznaczoną zakładkę **HTTPS**, wciśnij ikonkę zaznaczoną na obrazku wyżej. Dzięki temu zostanie skopiowany do schowka adres repozytorium.

Uruchom teraz terminal i przejdź w terminalu do lokalizacji na dysku na Twoim komputerze, gdzie chcesz umieścić ten projekt. Możesz wpisać w terminalu np.:

```
cd C:\Users\karol\Documents\Coding
```

Gdy jesteś już w terminalu w tej lokalizacji, wpisz następujące polecenie:

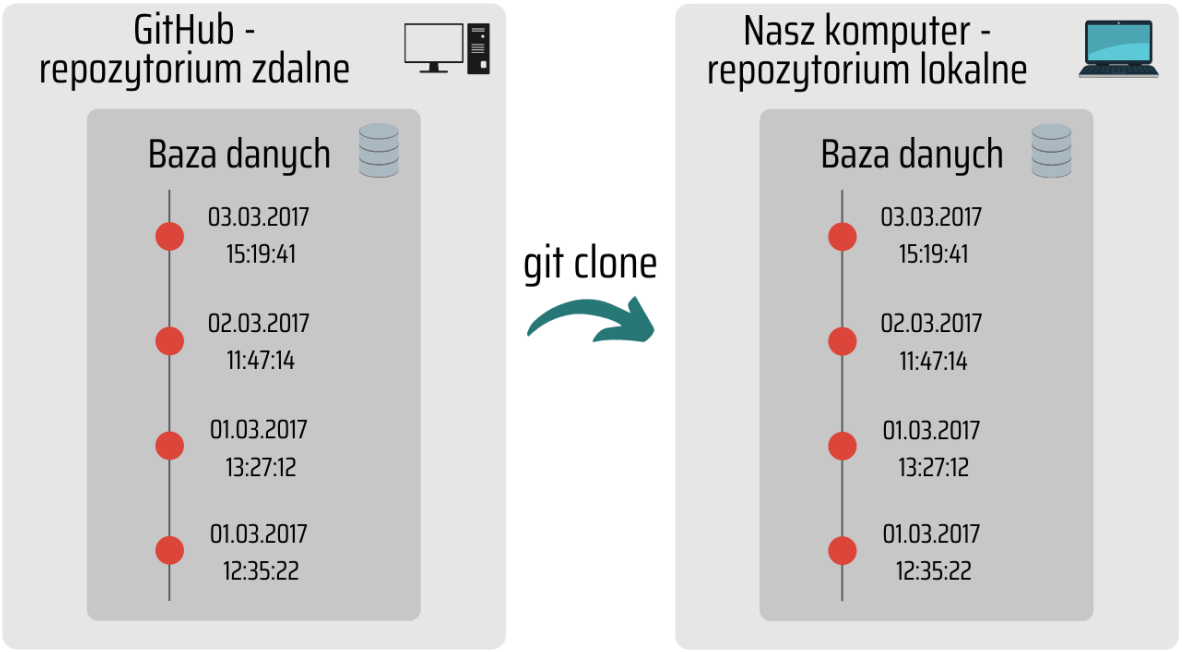
```
# git clone <tutaj_wklej_adres_skopiowany_do_schowka_z_github>
# czyli np.:
git clone https://github.com/zajavka/git-workshop.git
```

I zobaczysz wtedy na ekranie zapis podobny do poniższego:

```
C:\Users\karol\Documents\Coding>git clone https://github.com/zajavka/git-workshop.git
Cloning into 'git-workshop'...
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (4/4), done.
```

Obraz 2. GitHub cloned repository

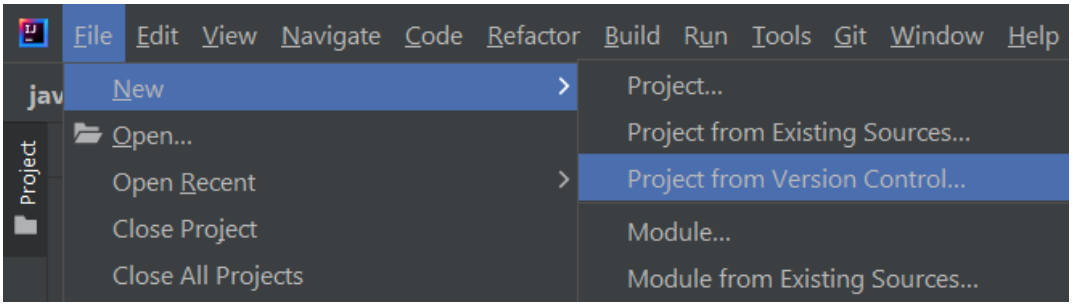
W tym momencie pojawi się u Ciebie na dysku repozytorium lokalne będące odzwierciedleniem stanu repozytorium zdalnego. Zauważ, że utworzony został również katalog o takiej nazwie jak nazwa Twojego zdalnego repozytorium. Grafika poniżej pomoże Ci zrozumieć operację, jaka została wykonana przed chwilą.



Obraz 3. Repository clone

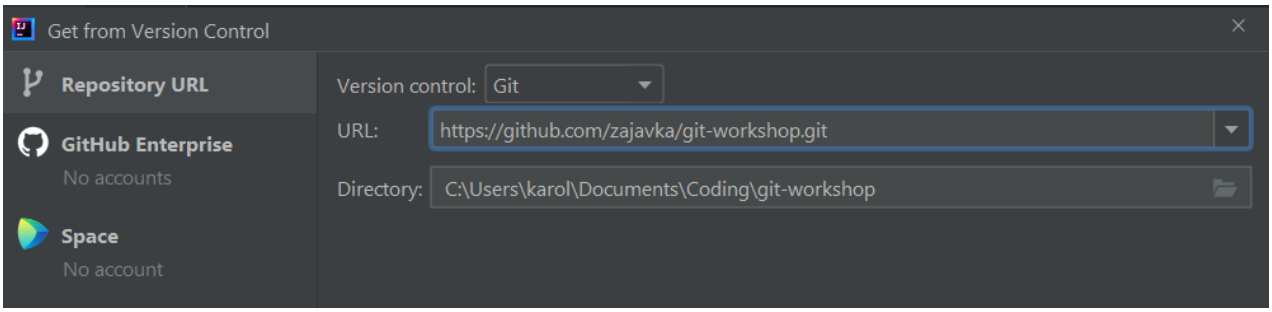
Intellij

Większość funkcjonalności, które będą pokazywane można zrealizować z poziomu Intellij. Przejdź do zakładki tak jak na poniższym obrazku:



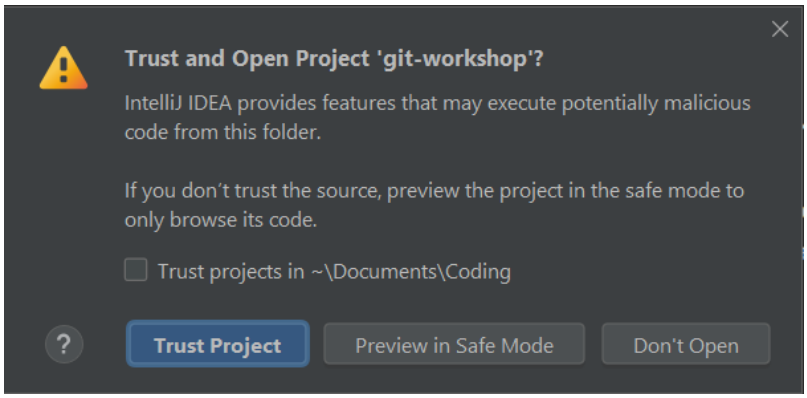
Obraz 4. Intellij project from Version Control

Następnie podaj adres repozytorium na **GitHub**:



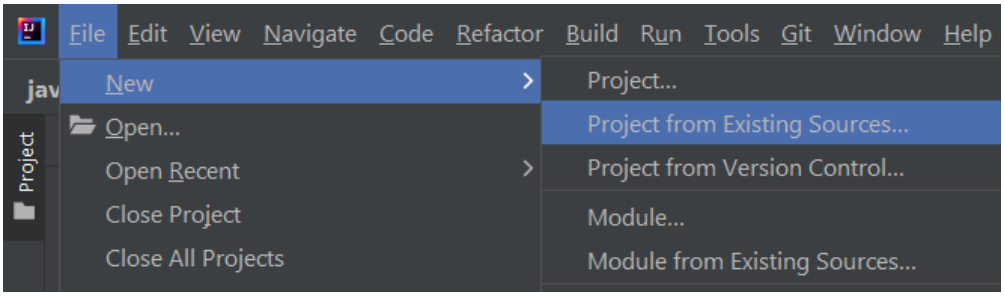
Obraz 5. Intellij project from Version Control

Intellij się o nas troszczy, dlatego zapyta, czy ufamy klonowanemu projektowi:



Obraz 6. Intellij project from Version Control

Korzystając z Intellij mamy projekt otwarty od razu w Intellij. Gdybyśmy wykonali `git clone` ręcznie, musielibyśmy dodać jeszcze ten projekt do Intellij:



Obraz 7. Intellij project from Existing Sources

Zwróć teraz uwagę, że po wykonaniu **clone**, masz dostępne u siebie na dysku dokładnie te same pliki, które zostały stworzone wraz ze stworzeniem repozytorium zdalnego. Chodzi o pliki `.gitignore` oraz `README.md`. Katalogu `.idea` nie ma w repozytorium zdalnym - to jest katalog, w którym Intellij trzyma

swoje ustawienia dla każdego projektu z osobna.

Możemy teraz rozpocząć pracę z takim projektem, pamiętając, że aby jakikolwiek zapis punktu w czasie dla zmian w projekcie został dodany do **Git** - musimy to zrobić sami. Nie dzieje się to automatycznie.

W tych materiałach będziemy wracać do Intellij i jego możliwości w integracji z **Git**em. Więcej na ten temat możesz przeczytać tutaj.

Zapisywanie zmian

Żeby zapisać zmiany w repozytorium musimy zrobić "zdjęcie" stanu repozytorium w danym momencie. Bardziej profesjonalnie używa się w tym przypadku stwierdzenia **snapshot** (*migawka*), które pojawiło się już wcześniej. W terminologii **Git** taki zrzut będziemy określali stwierdzeniem **commit**. Czyli zapisanie stanu w czasie będzie nazywane **commit**tem.

Natomiast zanim przejdziemy do omówienia samego **commit**a, musimy zapoznać się z krokami, które trzeba wykonać wcześniej.

Chcę zaznaczyć, że jak na razie skupimy się na pracy z repozytorium lokalnym. Gdy włączymy do pracy repozytorium zdalne - zostanie to wyraźnie zaznaczone.

Git status

Jeżeli jeszcze nie masz stworzonego repozytorium zdalnego i nie udało Ci się go sklonować, żeby odtworzyć to repozytorium lokalnie - to jest dobry moment, żeby to zrobić. Dalsze przykłady będą odnosiły się do stworzonego przeze mnie wcześniej repozytorium.

W stworzonym poprzednio repozytorium mieliśmy dodane 2 pliki: `.gitignore` oraz `README.md`. Możemy teraz dokonać w takim repozytorium następujące zmiany:

- możemy dodać nowe pliki do repozytorium,
- możemy zmodyfikować pliki, które już istnieją,
- możemy usunąć pliki, które są zapisane w repozytorium,
- możemy też te pliki przeczytać, ale nie zmieni to stanu repozytorium, więc to pominiemy. 😀

Każda ze wspomnianych czynności zostanie przez to repozytorium zauważona, gdyż będzie to oznaczało zmianę stanu całego repozytorium od ostatniego momentu, który jest naszym ostatnim "checkpointem" - takim punktem w czasie. Zanim natomiast dokonamy jakichkolwiek zmian, chciałbym pokazać Ci, jak można takie zmiany z perspektywy repozytorium zauważyć. Do tego służy komenda:

```
git status
```

Jeżeli wykonasz tę komendę, znajdując się w katalogu Twojego obecnego repozytorium lokalnego, na ekranie zostanie wydrukowana taka informacja:

```
On branch main ①
Your branch is up to date with 'origin/main'. ②

Untracked files: ③
  (use "git add <file>..." to include in what will be committed) ④
        .idea/ ⑤

nothing added to commit but untracked files present (use "git add" to track) ⑥
```

① Ta informacja mówi nam, na jakim **branch**u aktualnie się znajdujemy. O tym, czym jest **branch** powiemy sobie później.

② To zdanie informuje nas, że w repozytorium zdalnym nie ma żadnych zmian, których nie mamy zaktualizowanych w repozytorium lokalnym. O co chodzi z oznaczeniem *origin/main* też zostanie wyjaśnione później.

③ Ten fragment interesuje nas w tym momencie. Przypomnij sobie, że wcześniej, jak rozmawialiśmy o stanach plików, to pojawiło się takie stwierdzenie jak **untracked file**. Oznaczało to, że dany plik nie jest traktowany przez **Git** jako plik, który należy śledzić.

④ W tej linijce dostajemy podpowiedź, co należy zrobić, żeby wypisane poniżej na czerwono pliki zacząć śledzić.

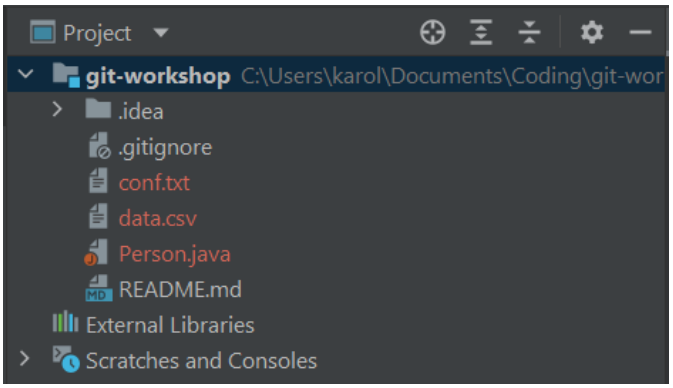
⑤ Tutaj mamy wypisane pliki lub katalogi, które nie są **trackowane**, czyli, których zmian **Git** nie śledzi.

⑥ Ta linijka mówi nam, że w tej chwili żaden plik nie zostanie dodany do **commita**. Jednocześnie powtarzane nam jest, że **Git** widzi pliki, których zmian nie rejestruje, czyli ich nie śledzi. Ponownie przypominane jest nam, co mamy zrobić, że zacząć śledzić zmiany danych plikach.

Nie używając Intellij, czyli korzystając ze zwykłego okienka do przeglądania plików, stwórz teraz w projekcie (w głównym katalogu) 3 dowolne pliki tekstowe, z dowolnymi rozszerzeniami, np.: `Person.java`, `conf.txt`, `data.csv` i dodaj w nich cokolwiek, treść może być dowolna. Następnie spróbuj ponownie uruchomić komendę `git status`. Na ekranie zostaną wydrukowane stworzone przez Ciebie pliki, zaznaczone kolorem czerwonym z informacją, że są one **untracked**. Czyli **Git** nadal nie uznaje tych plików za swoje - nie śledzi ich, ale je widzi.

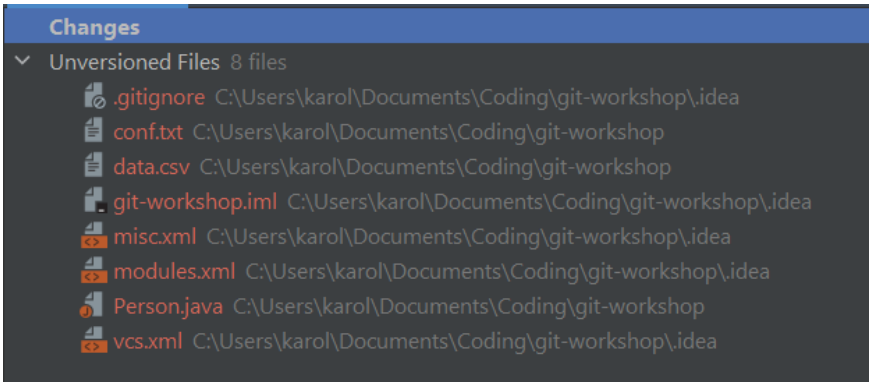
Intellij

A jak pokaże nam to Intellij? Na drzewie projektu będzie to wyglądało w ten sposób:



Obraz 8. Intellij i Git na drzewie projektu

Natomiast jak otworzysz zakładkę **Git**, która znajduje się w lewym dolnym rogu ekranu:



Obraz 9. Intellij i zmiany Git

W rozumieniu Intellij, stwierdzenie **Unversioned Files** oznacza, że **Git** widzi te pliki w stanie **untracked**.

Wiem, że nowego słownictwa znowu jest dużo. Do tego trzeba się po prostu przyzwyczaić.

Git add

Jeżeli chcemy dać **Git**owi znać, że ma zacząć śledzić te pliki, należy wykonać teraz komendy:

```
git add Person.java
git add conf.txt
git add data.csv
```

Jeżeli teraz wykonamy `git status`, na ekranie pokaże się taki wynik:

```
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed: ①
  (use "git restore --staged <file>..." to unstage) ②
        new file:   Person.java ③
        new file:   conf.txt
        new file:   data.csv

Untracked files: ④
  (use "git add <file>..." to include in what will be committed)
        .idea/
```

① W tej części zostały wydrukowane nazwy plików, które zostaną uwzględnione, gdy wykonamy **commit**. Pamiętasz stan **staged**? Te pliki są właśnie w tym stanie. Inaczej można powiedzieć, że przenosimy te pliki do obszaru **staging area**.

② Tutaj **Git** podpowiada nam, co możemy wpisać, żeby pliki w stanie **stage**, przestały być w tym stanie.

③ Tutaj mamy listę plików, które zostaną uwzględnione, jeżeli zrobilibyśmy teraz **commit**. Zwróć uwagę na dopisek *new file*, później nam się to przyda.

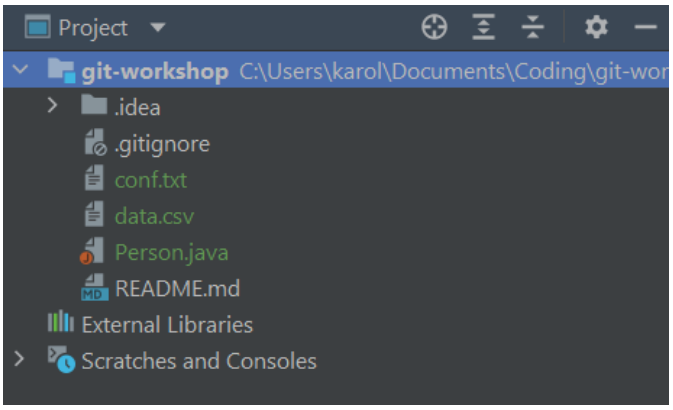
④ Na dole **Git** pokazuje nam pliki/katalogi, które nie zostaną uwzględnione, gdy zrobimy **commit**.

Oprócz wspomnianej komendy `git add <file>` możemy posłużyć się jeszcze innymi wariantami:

- `git add -A` - wszystkie nieśledzone pliki zaczynają być śledzone. Z tą komendą trzeba uważać, jeżeli nie wszystkie pliki, które istnieją w projekcie, mają trafić do repozytorium i jednocześnie pliki te nie są ignorowane (o ignorowaniu powiemy później).
- `git add .` - **Git** zaczyna śledzić wszystkie pliki i katalogi, które znajdują się w obecnym katalogu.

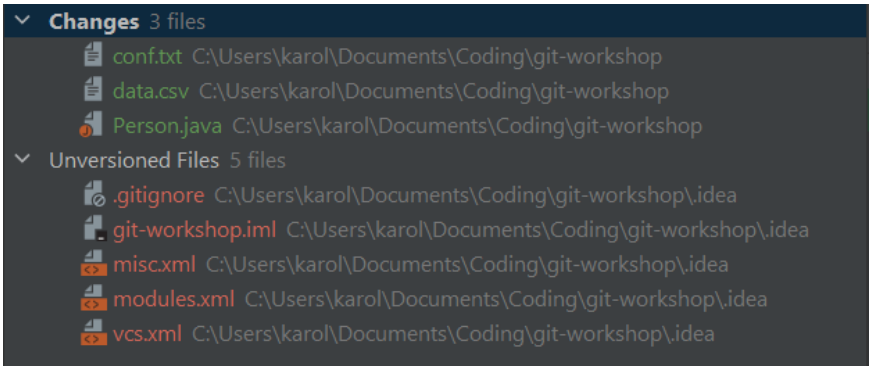
Intellij

Jak powyższą sytuację pokaże nam Intellij? Tak na drzewie projektu:



Obraz 10. Intellij i zmiany Git

A tak w okienku **Git**:



Obraz 11. Intellij i zmiany Git

Co się stanie, gdy zmodyfikujemy teraz zawartość któregoś z plików zaznaczonych na zielono?

Zmień treść dowolnego pliku, wykonaj ponownie komendę `git status` i zobacz, co zostanie wydrukowane na ekranie. Zwracam tylko uwagę - zmień tę treść gdzie indziej niż w Intellij, on robi trochę magii pod spodem. W tym przykładzie zmieniłem plik `Person.java`.

```
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed: ①
  (use "git restore --staged <file>..." to unstage)
        new file:   Person.java ②
        new file:   conf.txt
        new file:   data.csv
```

```
Changes not staged for commit: ③
  (use "git add <file>..." to update what will be committed) ④
  (use "git restore <file>..." to discard changes in working directory) ⑤
        modified:   Person.java ⑥

Untracked files: ⑦
  (use "git add <file>..." to include in what will be committed)
        .idea/
```

① Ten fragment widzieliśmy już wcześniej, tutaj mamy zaznaczone pliki, które wejdą w następny **commit**.

② Zwróć uwagę, że mamy tutaj wypisany plik `Person.java`.

③ Tutaj mamy wypisane **ZMIANY** (ciekawe, czy dopiero teraz zwróciłeś/-aś uwagę, że tu jest napisane **changes** 😀), które nie zostaną uwzględnione, gdy wykonamy **commit**. Czyli te zmiany są **unstaged**.

④ Jeżeli chcemy, żeby dodane zmiany były uwzględnione, gdy zrobimy **commit** - musimy wykonać komendę `git add <file>`.

⑤ Możemy również wprowadzone przed momentem zmiany odrzucić. Wykonaj tę komendę i zobacz, że faktycznie zostały odrzucone zmiany, które przed chwilą dodaliśmy.

⑥ Plik `Person.java` pojawia się ponownie na drugiej liście.

⑦ Nadal nie zrobiliśmy nic z katalogiem `.idea/` (i na razie nie mam zamiaru), dlatego dalej pozostaje on **untracked**.

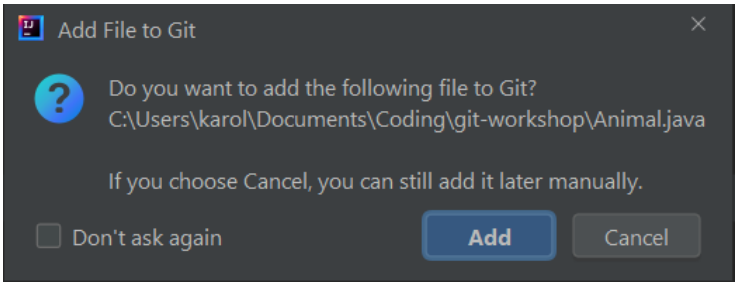
Changes

Skoro już tak wyraźnie zostało zaznaczone słowo **ZMIANY**, należy tutaj zwrócić uwagę na pewną kwestię. **Git** zapamiętuje stan plików, w którym znajdowały się one, gdy uruchamialiśmy komendę `git add`. Jeżeli wykonamy na danym pliku komendę `git add` po raz pierwszy - **Git** zapisze stan tego pliku po raz pierwszy i od tego momentu zacznie śledzić zmiany w tym pliku. Należy jednak pamiętać, że jeżeli zdecydujemy się na zapis **migawki**, czyli zrobimy **commit**, to zapisane zostaną ostatnie zmiany, które były dodane komendą `git add`. Jeżeli po wykonaniu komendy `git add`, wprowadzimy jakieś nowe zmiany w pliku, to **Git** zapisze sobie w migawce (w **commicie**) ostatnie zmiany, które zostały mu powiedziane komendą `git add`.

Czyli jeżeli jakiś plik został już dodany komendą `git add` i wprowadzimy w nim jakieś zmiany, to musimy wywołać `git add` ponownie. Inaczej, gdy będziemy wykonywali **commit** - zostanie zapisana ostatnia wersja pliku, która została "uwzględniona" komendą `git add`. Czyli pliki zaznaczone na zielono można rozumieć jak pliki w poczekalni do zostania zatwierdzonym (do zrobienia **commit**). Natomiast, w poczekalni znajdują się konkretne wersje tych plików. Czyli gdy plik znajdujący się w poczekalni zostanie zmodyfikowany (dodamy jakieś zmiany), należy dodać jego nową wersję do poczekalni ponownie. Inaczej zatwierdzona zostanie ostatnia wersja pliku, która znajdowała się w poczekalni. Możemy też te zmiany odrzucić komendą `git restore`. Z jednej strony daje nam to dużo możliwości, z drugiej natomiast trzeba uważać, co się robi. 😀

Czemu tak zwracałem uwagę wcześniej na to, żeby przykłady robić bez użycia Intellij?

Intellij dodaje do tego wszystkiego trochę swojej magii. Stwórz teraz nowy plik w projekcie, ale z poziomu Intellij. Dostaniesz wtedy poniższe pytanie. Możesz również zaznaczyć opcję, żeby nowe pliki zawsze były od razu dodawane.



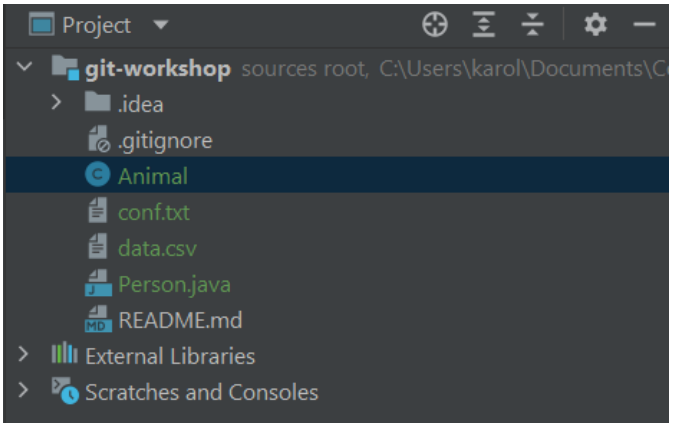
Obraz 12. Intellij i dodawanie nowego pliku

Jeżeli klikniesz "Cancel" i uruchomisz `git status`, to na ekranie zostanie wydrukowany taki wynik:

```
# reszta
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .idea/
        Animal.java
```

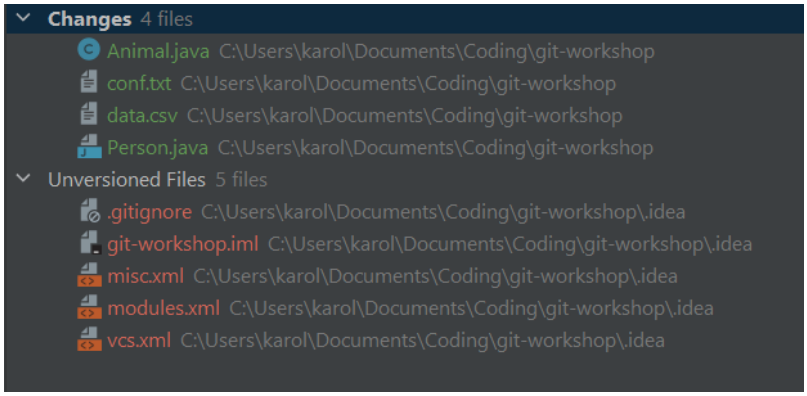
Jeżeli natomiast wybierzesz **Add**, to Intellij automatycznie za Ciebie wykona `git add`. Jeżeli zaczniesz w tym momencie modyfikować zawartość tego pliku z poziomu Intellij, to po wykonaniu `git status`, plik `Animal.java` pojawi się w dwóch sekcjach, analogicznie do pliku `Person.java` w poprzednich przykładach.

Jak plik `Animal.java` będzie teraz wyświetlany w Intellij? Tak na drzewie projektu:



Obraz 13. Intellij i dodawanie nowego pliku

A tak w zakładce **Git**:



Obraz 14. Intellij i dodawanie nowego pliku

Należy tutaj zwrócić uwagę na to, że Intellij zachowuje się troszeczkę inaczej niż sam **Git**. Jeżeli modyfikujemy plik z poziomu Intellij, to cały czas będzie on widoczny w sekcji **Changes**, nie pojawia się on z powrotem w **Unversioned Files**. Jeżeli natomiast wykonasz teraz `git status`, to zobaczysz, że najnowsze zmiany dodane w pliku `Animal.java` będą dodane jako **not staged**.

Git commit

No dobrze, to jak w końcu można dodać te zmiany do repozytorium? Jak zrobić migawkę? Należy wykonać komendę `git commit`.

Spróbuj teraz wywołać `git commit`. Otworzy Ci się domyślny edytor tekstowy, który jest ustawiony ustawieniach **Git** - w moim przypadku jest to **Notepad++**. Na ekranie pokaże Ci się taka wiadomość:

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
①
```

① Dalej pod spodem będzie to samo, co jest drukowane na ekranie przy wykorzystaniu komendy `git status`.

Git prosi o podanie **commit message**. W jakim celu? Wyobraź sobie, że repozytorium jest już całkiem spore, np. mamy zapisane około 5000 punktów w czasie. Jak można teraz zacząć szukać konkretnej zmiany? Najlepiej by było, gdyby ktoś na etapie dodawania tej zmiany do projektu zapewnił jej adekwatny opis. Opis na tyle sensowny, żeby czytając ten opis po roku, dało się zrozumieć, czego dana zmiana dotyczyła albo po co była, co w tej zmianie zostało naprawione, dodane itp. Po to właśnie jest **commit message**.

Dodajmy zatem wiadomość "Hi, it's my first commit in this repo. Git is nice", zapisujemy i zamknijmy edytor tekstu.

Najczęstsza konwencja dotycząca opisu zmian w **Git**, która jest stosowana w projektach, zakłada, żeby pisać te opisy w języku angielskim. Chyba że umówicie się w projekcie inaczej.

Na ekranie zostanie teraz wydrukowane coś takiego:

```
[main cf941b7] Hi, it's my first commit in this repo. Git is nice ①
 4 files changed, 5 insertions(+) ②
 create mode 100644 Animal.java ③
 create mode 100644 Person.java
 create mode 100644 conf.txt
 create mode 100644 data.csv
```

① Pierwsza część linijki - *main* - oznacza nazwę **branch**a, do tego wrócimy potem. Następnie mamy skróconą wersję **commit hash** - jest to unikalny identyfikator dodanej przez nas zmiany, nadaje go **Git**. Dalej widoczna jest wiadomość, którą wpisaliśmy w edytorze tekstowym.

② Tutaj **Git** nam mówi, że zmieniliśmy 4 pliki (nawet jeżeli dodaliśmy je po raz pierwszy), a sama zmiana dotyczyła pięciu dodanych linijek.

③ Kolejne linijki mówią nam, które pliki zostały uwzględnione przy robieniu **commit**. Początek tych linijek nas w tym momencie nie interesuje.

Czasami można się spotkać ze stwierdzeniem **revision** (*rewizja*). Stwierdzenia tego używa się do oznaczenia konkretnej wersji kodu źródłowego. Można powiedzieć, że **Git** implementuje rewizje przy wykorzystaniu commitów. Commity są wtedy identyfikowalne poprzez **hash** lub **short hash**. Dodam tutaj tylko, że algorytm wykorzystany do wygenerowania hasha to *SHA-1*.

Gratulacje! Na tym etapie masz już dodany swój pierwszy **commit** do repozytorium lokalnego. Zwróć uwagę, że nigdzie nie powiedziałem jeszcze, że nasze zmiany będą widoczne w repozytorium zdalnym. Możesz nawet wejść na stronę:

```
https://github.com/<twoja_nazwa_uzytkownika>/<twoja_nazwa_repozytorium>/commits/main
```

I zobaczyć, że **commit**, który dodaliśmy przed chwilą, nie jest tutaj widoczny. Jeszcze do tego wrócimy.

Modyfikacja plików committed

Spróbuj teraz zmodyfikować, któryś z plików, jaki jest już za**commit**owany w repozytorium. Następnie wykonaj komendę `git status` i zobacz, co zostanie wydrukowane na ekranie:

```
  (use "git push" to publish your local commits) ①

Changes not staged for commit: ②
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   Animal.java ③

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .idea/ ④

no changes added to commit (use "git add" and/or "git commit -a") ⑤
```

① Tego zapisu nie widzieliśmy wcześniej, **Git** podpowiada nam, co możemy zrobić, żeby nasze zmiany znalazły się w repozytorium zdalnym. Jeszcze do tego wrócimy.

② Tutaj mamy wypisane **ZMIANY**, które nie zostaną uwzględnione, gdy wykonamy **commit**. Czyli te zmiany są **unstaged**.

③ Tutaj mamy wypisane pliki **unstaged**. Zwróć uwagę, że wcześniej w tej linijce pojawiał się napis **new file**. Teraz mamy tutaj napisane **modified**. Jeżeli usuniemy jakiś plik i wykonamy `git status`, to obok takiego pliku będzie napisane **deleted**.

④ Tutaj mamy wypisane pliki lub katalogi, które nie są **trackowane**, czyli, których zmian **Git** nie śledzi.

⑤ Ta linijka mówi nam, że w tej chwili żaden plik nie zostanie dodany do **commit**a. Pojawia się tutaj natomiast nowa komenda `git commit -a`, o której przeczytasz poniżej.

Git commit -a

Do komendy `git commit` możemy dołożyć flagę `-a`. Ewentualnie flagę `--all`. Możemy traktować taki zapis jak dodanie kroku `git add` przed wykonaniem `git commit`. Różnica jest jednak taka, że dodając wspomniane flagi, pliki, które są w naszym working directory, są automatycznie przenoszone w stan **staged**. Nie wpływa to na pliki **untracked**.

Dokumentacja opisuje to w ten sposób:

> Tell the command to automatically stage files that have been modified and deleted, but new files you have not told Git about are not affected.

Czyli zapisując komendę w taki sposób:

```
git commit -a
# lub
git commit --all
```

Nie musimy się martwić, że zapomnimy dodać nowe zmiany do **stage** przy wykorzystaniu `git add`.

Git commit -m

Git pozwala nam na wykorzystanie kolejnej flagi `-m` lub `--message`, która daje możliwość dodania wiadomości do **commit**a w jednej linijce. Nie będzie się nam wtedy otwierał edytor tekstowy.

```
git commit -m "And here it is my next commit message"
```

Wspomniane flagi możemy ze sobą złączyć i napisać to w ten sposób:

```
# możemy tak
git commit -am "My next commit message"
# lub możemy tak
git commit -a -m "My next commit message"
# tak nie możemy, bo wiadomość jest parametrem flagi -m
# git commit -ma "My next commit message"
```

Git diff

Zanim zrobimy **commit**, dobrze by było zrobić przegląd naszych zmian i upewnić się, że wszystko co dodajemy do **commit**ta, powinno się tam faktycznie znaleźć. Komenda `git diff` służy do podglądu wprowadzanych zmian. Dzięki temu możemy uniknąć błędów - czyli dodania do repozytorium plików, albo zmian, których dodać nie chcemy. Patrząc przez analogię - `git status` pokazuje nam tylko, które pliki zostały zmienione, natomiast `git diff`, powie nam, jakie dokładnie to były zmiany.

Dodaj jakieś zmiany w kilku plikach i spójrz wynik wywołania komendy:

```
git diff
```

Rezultat:

```
diff --git a/Animal.java b/Animal.java
index 5fe5a24..576f1d3 100644
--- a/Animal.java
+++ b/Animal.java
@@ -1,6 +1,7 @@
 public class Animal {

     public static void main(String[] args) {
-        // sdjjksfjw3il23jh4j ①
+        // sjkf234jhjhrh ②
+        //hsdfhfk h23uf
     }
 }
```

cd.

```
diff --git a/Giraffe.java b/Giraffe.java
index 7a0eab4..6bad2b8 100644
--- a/Giraffe.java
+++ b/Giraffe.java
@@ -1,4 +1,4 @@
 public class Giraffe {
-
     // sdfjksdfhksdfsdf
+    // jkhfsf2hlhuh3
 }
diff --git a/conf.txt b/conf.txt
index 4c0281c..0e5394d 100644
--- a/conf.txt
+++ b/conf.txt
@@ -1 +1,2 @@
-some config ③
\ No newline at end of file
+some new config ③
+fk234jh34
\ No newline at end of file
```

① W ten sposób oznaczana jest linijka, która została usunięta.

② Tak oznaczana jest linijka, która została dodana

③ Tutaj mamy oznaczenie linijki, która została zmodyfikowana. **Git** traktuje to jako usunięcie i dodanie od nowa.

Zwróć uwagę, że wywołanie komendy `git diff` odnosi rezultat, jeżeli wspomniane pliki nie są oznaczone jako **staged**. Jeżeli na wszystkich zmodyfikowanych plikach wykonamy teraz komendę `git add`, a następnie wykonamy ponownie `git diff`, to otrzymamy pusty wynik.

Wynika z tego, że komenda `git diff` może być stosowana do plików, które są w naszym working directory, ale nie są **staged**. Jaka komenda ma być zatem stosowana do plików **staged**? Ta sama, ale z dopiskiem `--cached`:

```
git diff --cached
```

Przypomnę, że informacje drukowane na ekranie przez **Git** możemy przewijać wykorzystując **spację**. Kończymy drukowanie tych informacji klawiszem **q**.

Git log

Dodaliśmy już kilka **commit**tów, zatem pewnie zastanawiasz się - **Jak je teraz przeglądać?**, albo **Jak zobaczyć teraz historię zapisywanych zmian?** Do tego służy komenda `git log`. Jeżeli ją teraz wykonasz, to na ekranie wydrukowane zostaną takie informacje (albo podobne w zależności od Twoich działań):

```
commit 6665d10dd03b972a70a323744ed85901c41a1ded (HEAD -> main) ①
Author: <user_name> <email> ②
Date:   Mon Apr 18 13:58:46 2022 +0200 ③

    My next commit message ④

commit cf941b7f855ed1157dd21d2399e6ee84552a1db4 ⑤
Author: <user_name> <email>
Date:   Fri Apr 15 18:08:14 2022 +0200

    Hi, it's my first commit in this repo. Git is nice

commit cb785f56fdbb4355b7a308c320eef5b8b2f0b4c4 (origin/main, origin/HEAD) ⑥
Author: zajavka <103562129+zajavka@users.noreply.github.com>
Date:   Wed Apr 13 13:44:42 2022 +0200

    Initial commit
```

① Ten długi krzak nazywa się **hash**. Jest to unikalny identyfikator naszego **commit**a. Wcześniej wspomniałem o wersji skróconej, czyli np. **cf941b7**. Teraz widzimy ten identyfikator w pełnej okazałości. Jeżeli chodzi o stwierdzenia **HEAD** i **main** - do nich jeszcze wrócimy.

② Tutaj możemy podejrzeć autora razem z jego mailem.

③ Tutaj widzimy datę wprowadzenia danej zmiany - czyli datę **commit**ta.

④ Tutaj mamy wypisaną wiadomość, która została wprowadzona razem z **commit**tem.

⑤ **Hash** pojawia się ponownie, oznacza to kolejny **commit**, czyli kolejny wpis w naszym repozytorium.

⑥ To jest ostatni **commit**, który znajduje się w repozytorium zdalnym. Wrócimy jeszcze do tych oznaczeń.

Jeżeli wywołamy komendę `git log` bez żadnych argumentów, zmiany będą wyświetlane od najnowszej do najstarszej.

Git commit --amend

Amend jest możliwością modyfikacji **commit**a. Najczęściej służy on do modyfikacji treści wiadomości lub modyfikacji samej zawartości **commit**a. Aby wykonać **amend**, należy dodać w komendzie `git commit` flagę `--amend`, przykład:

```
git commit --amend -a -m "New commit message"
```

Flagi `-a` i `-m` widzieliśmy już wcześniej. Dodanie flagi `--amend` spowoduje, że wymusimy edycję poprzedniego **commit**a, zamiast dodawać nowy w historii.

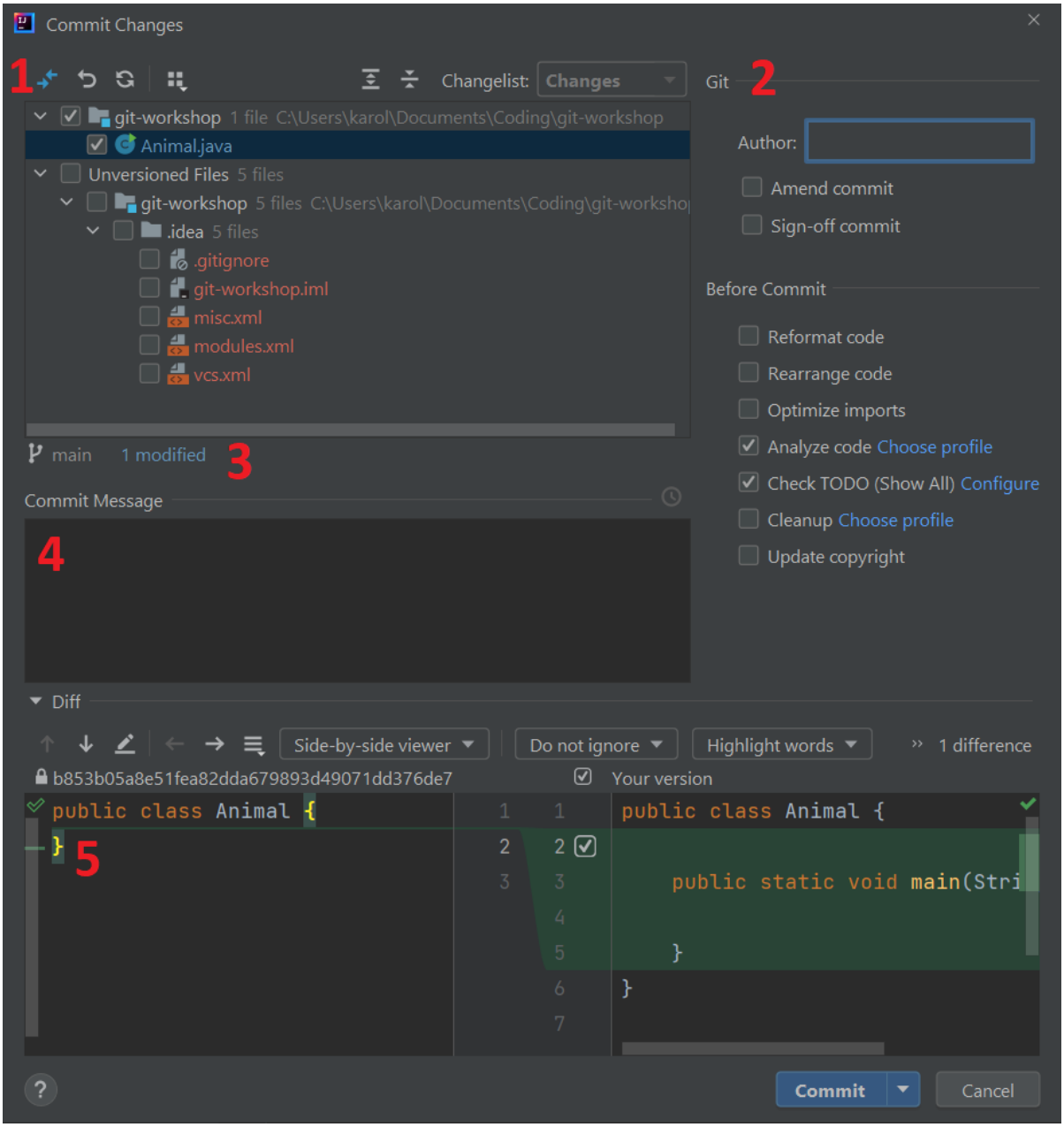
Powiedzmy, że stosowanie tej komendy jest okej, jeżeli nasze zmiany nie zostały jeszcze zsynchronizowane z repozytorium zdalnym. Jeżeli historia naszego repozytorium lokalnego jest już widoczna w repozytorium zdalnym, to stosowanie `--amend` może doprowadzić do dziwnych i nieprzediwywalnych błędów. Polecenie `--amend` zmienia **hash commit**a i wyprostowanie takiej sytuacji między repozytorium lokalnym, a zdalnym to wyższa szkoła jazdy.

Intellij

Commit

A jak wykonać **commit** z poziomu Intellij? W zakładce **Git** Należy wybrać pliki, które nas interesują, kliknąć prawym przyciskiem myszy i można wybrać opcję **Commit Files...**. Drugi sposób to kliknąć zieloną ikonkę "checked", w prawym górnym rogu ekranu. Trzecia możliwość to skrót **CTRL** + **K**.

Otworzy nam się teraz okienko wykonania **commit**ta, które daje bardzo dużo możliwości.



Obraz 15. Intellij i Git commit

Możemy wyróżnić w tym okienku kilka części:

1. W tej części możemy wybrać jakie pliki mają wejść w zakres **commit**ta.
2. Tutaj możemy wybrać dodatkowe ustawienia dotyczące **commit**ta, o których jeszcze powiemy. Możemy tutaj również zaznaczyć, co Intellij ma zrobić, zanim wykonany zostanie **commit**.
3. Tutaj mamy wypisaną informację dotyczącą tego, jakie zmiany wykonujemy i ile ich jest. Do tego mamy wypisany **branch**.
4. Tutaj podajemy **commit message**.
5. W tym miejscu możemy sprawdzić zmiany, które będziemy dodawać. Do tematyki sprawdzania dodawanych zmian jeszcze wrócimy. Po lewej stronie widzimy stan pliku na moment ostatniego **commit**ta, gdzie pokazywany **hash** to unikalny identyfikator ostatniego **commit**ta. Po prawej stronie mamy stan obecny (po zmianach).

Jeżeli wciśniemy teraz przycisk **Commit** - Intellij wykona **git commit**. Pamiętać należy, że Intellij

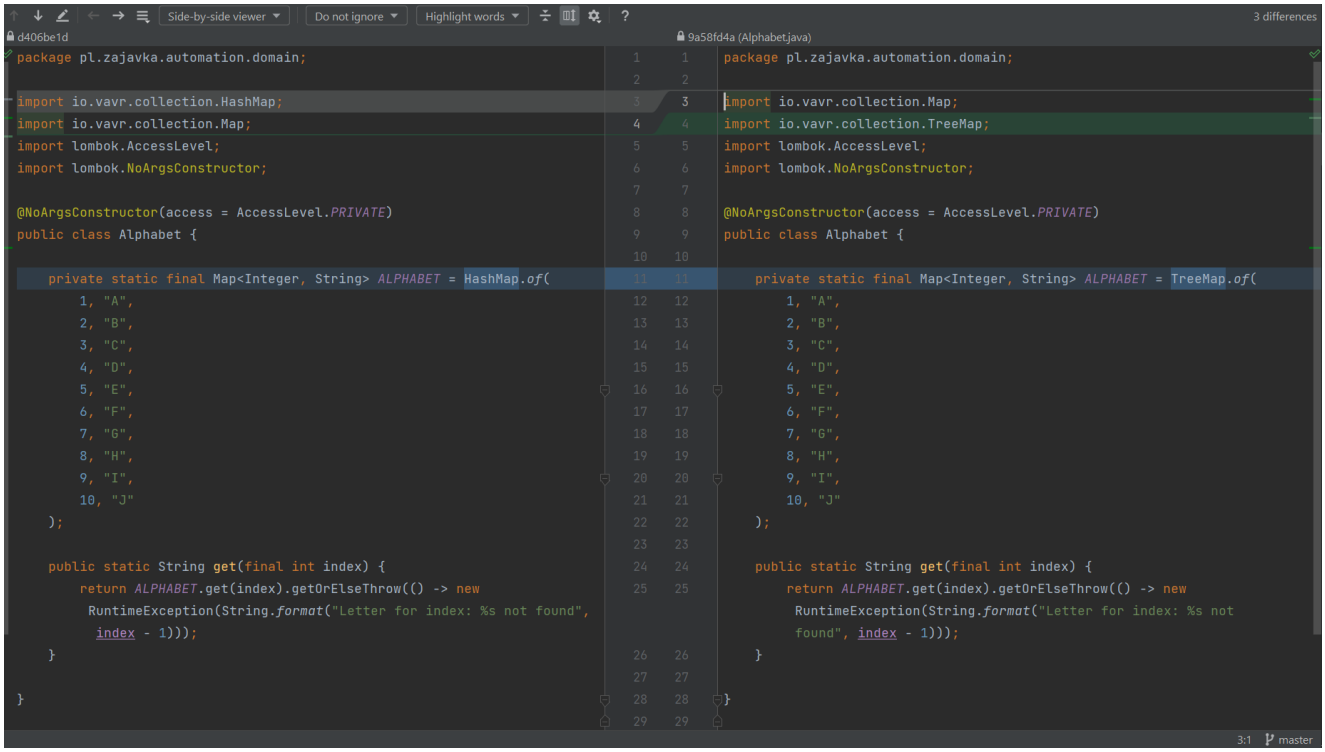
wykonuje pod spodem trochę bardziej złożoną komendę niż my sami. Wykonywane przez Intellij komendy możemy podejrzeć w zakładce **Git** > **Console**. Więcej możesz przeczytać tutaj.

Zwróć uwagę, że opisując **commit**owanie w Intellij, nigdzie nie wspomniałem o komendzie **git add**. Intellij robi to za nas, skracając nam jednocześnie trochę ścieżkę.

Diff

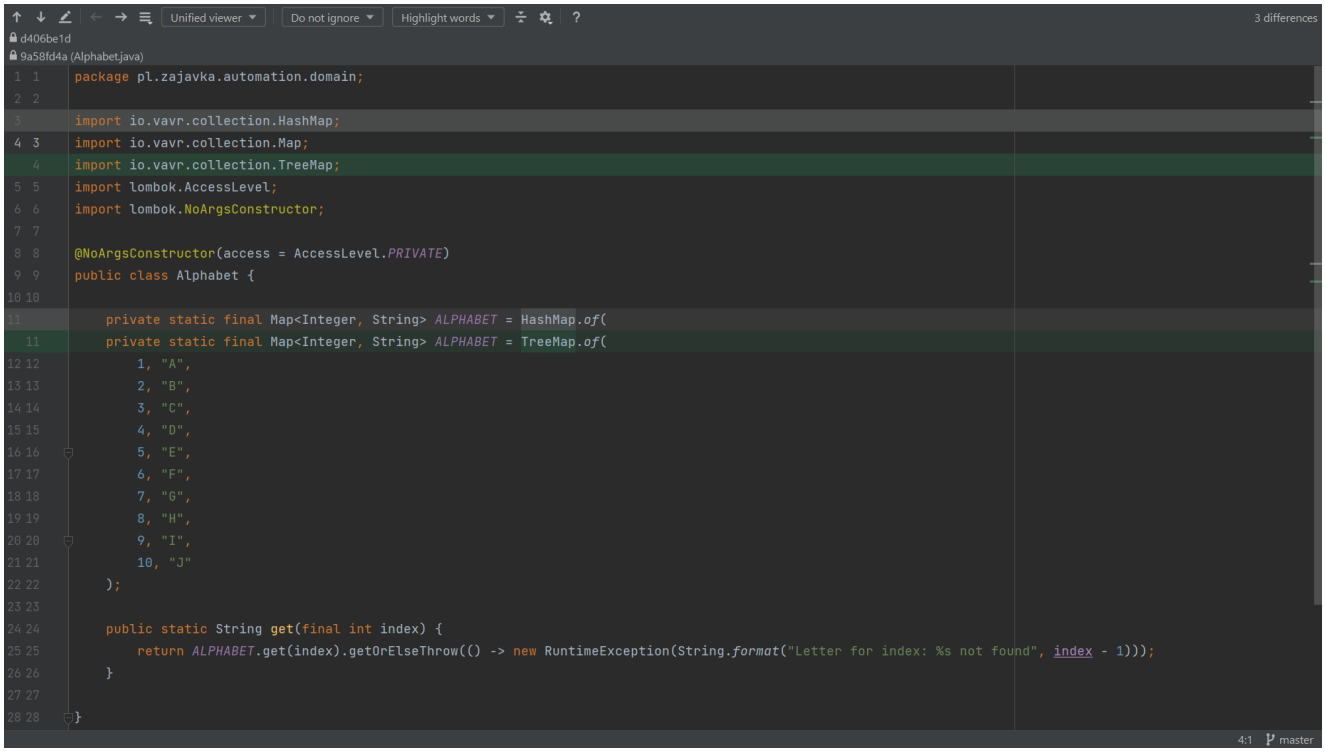
Intellij potrafi pokazać w znacznie bardziej przystępny sposób, jakie zmiany zostały wprowadzone w kolejnych plikach. Żeby to zobaczyć, przejdź do zakładki **Git** i wybierz dowolny plik - po prawej stronie pokażą się zmiany. Możesz również wcisnąć teraz skrót `Ctrl + D` - otworzy to nowe okno z porównaniem.

Intellij daje nam dwa możliwe widoki porównania plików. Pierwszy - **Side-By-Side**:



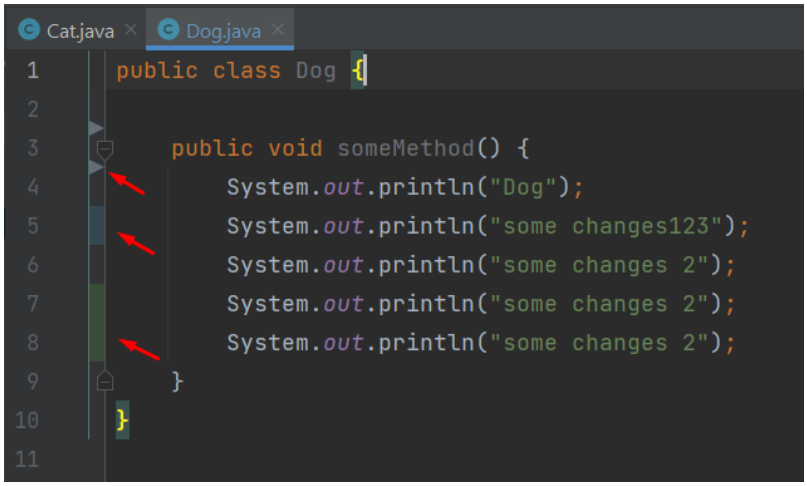
Obraz 16. Intellij Diff Side-By-Side

Drugi - **Unified** - w tym przypadku wszystkie zmiany są pokazane w jednym widoku, a nie dwóch oddzielnych jak poprzednio.



Obraz 17. Intellij Diff Unified

Gdy pracujemy w Intellij i dodajemy jakieś modyfikacje w pliku, są one również widoczne linijka po linijce przy numerach linijek:

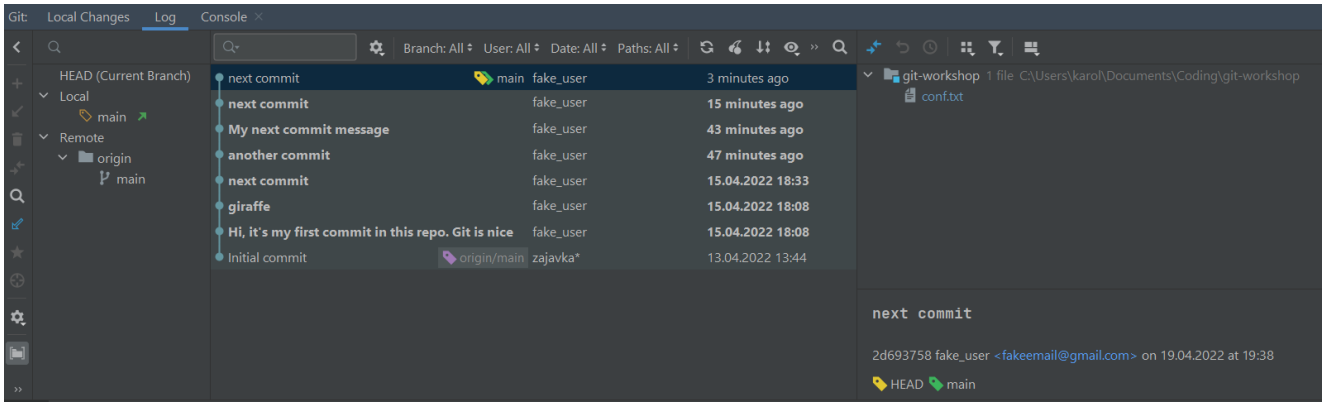


Obraz 18. Intellij Diff Unified

Kolor niebieski oznacza modyfikację treści, kolor zielony dodanie treści, a szary trójkąt oznacza usunięcie treści. Jeżeli klikniesz w któryś kolor, to Intellij pokaże Ci kolejne okno, które pozwala podjąć następne kroki z tymi zmianami, np. wycofać zmiany wprowadzone w konkretnych linijkach.

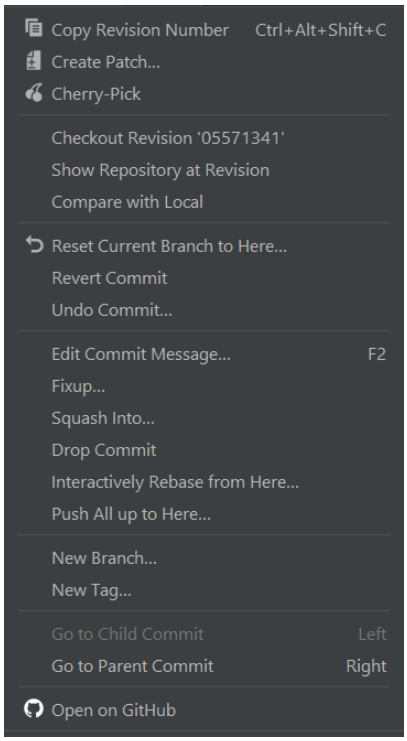
Log

Historia zmian w repozytorium może zostać również przedstawiona w Intellij w sposób bardziej przystępny niż w konsoli. Przejdź do zakładki **Git** > **Log**. Zobaczysz tutaj o wiele więcej informacji, niż było wydrukowanych po wykonaniu komendy `git log`. Poniżej grafika:



Obraz 19. Intellij i Git log

Ten widok jest o tyle istotny, że dużą ilość operacji, które będą omawiane w następnej kolejności można też wykonać z poziomu tego widoku. W tym celu należy kliknąć prawym przyciskiem myszy na konkretny commit. Zobaczymy wtedy takie okienko:



Obraz 20. Intellij i Git log

Z poziomu tego okienka możemy teraz wybrać polecenia analogiczne do komend, które będziemy omawiać w następnej kolejności.

Okienko to daje nam również możliwość wykorzystania skrótu `Ctrl + D`, wystarczy, że w historii zmian znajdziemy dowolny plik, na którym została dodana jakaś modyfikacja i wciśniemy `Ctrl + D`. Pojawi nam się wtedy okno pokazujące wprowadzone zmiany w pliku.

Więcej na temat tej zakładki możesz przeczytać tutaj.

Ignorowanie zmian

Chyba w każdym projekcie będziemy mieli pliki i katalogi, których nie chcemy zapisywać w **Git**, bo zwyczajnie nie ma to żadnego sensu. Najczęściej są to pliki, które są generowane przez różne narzędzia,

czyli przykładowo katalog **target** generowany przez **Maven**, lub katalog **build** albo **.gradle** generowany przez **Gradle**. Dodawanie katalogu **out**, w którym Intellij umieszcza skompilowane pliki `.class` również nie ma sensu, bo pliki te są wygenerowane na podstawie naszego kodu źródłowego. Z tego właśnie powodu nie dodawałem wcześniej do repozytorium całego katalogu `.idea/`, jest to katalog potrzebny z punktu widzenia Intellij może być on w pełni odtworzony od zera przez Intellij. Dodawanie tego typu plików będzie nam zaciemniało faktyczne zmiany dodawane w projekcie i może prowadzić do rozmaitych błędów. Dlatego pamiętaj, że do repozytorium dodajemy tylko faktyczny kod źródłowy.

Jak dotychczas, dodawanie katalogu `.idea/` było przeze mnie faktycznie pomijane, ale można to zrobić lepiej. Możemy stworzyć plik z konfiguracją określającą pliki lub katalogi, które mają być ignorowane przez repozytorium. Możemy takie pliki wtedy dowolnie modyfikować, ale zmiany w tych plikach nie będą widoczne z poziomu repozytorium. Aby określić konfigurację ignorowania, należy dodać do **Git** plik `.gitignore`. Może się to wydawać dziwne, ale ten plik jakby nie ma nazwy tylko samo rozszerzenie `.gitignore`. W treści takiego pliku możemy wpisać:

```
### IntelliJ IDEA ##①
.idea ②
*.iws ③
*.iml
*.ipr
out/
!**/src/main/**/out/ ④
!**/src/test/**/out/

### Gradle ###
.gradle
build/ ⑤
!gradle/wrapper/gradle-wrapper.jar
!**/src/main/**/build/
!**/src/test/**/build/
```

① Znaczek `#` oznacza komentarz.

② Ignoruj cały katalog `.idea` wraz z jego zawartością.

③ Ignoruj wszystkie pliki z rozszerzeniem `*.iws`.

④ Wykrzyknik oznacza dodanie wyjątku do zdefiniowanych wcześniej reguł. Linijka wyżej mówi, żeby ignorować wszystkie katalogi `out/`. Ta linijka dodaje wyjątek do ignorowania wszystkich katalogów `out/` i mówi, żeby nie ignorować katalogów `out/`, które znajdą się na ścieżce pasującej do podanego wzorca.

⑤ Ponownie ignorujemy cały katalog `build/`.

Co oznaczają podwójne gwiazdki?

```
**/src/main/**/out/
```

Podwójne gwiazdki można rozumieć jako cokolwiek. Czyli może się w ich miejscu pojawić dowolna nazwa katalogu.

Z plikiem `.gitignore` trzeba uważać, żeby dodawać go jak najwcześniej jest to możliwe. Jak już dodamy jakiś plik do repozytorium i jednak będziemy chcieli zacząć go ignorować to trzeba się już troszkę z tym

pobawić. Na ten moment nie będziemy tego poruszać.

Zwróć uwagę, że mamy dodany już plik `.gitignore` do repozytorium. Został on wygenerowany automatycznie na etapie tworzenia repozytorium zdalnego i wybraliśmy wtedy schemat tego pliku, mówiąc, że piszemy projekt w Java. Należy jednak zwrócić uwagę, że nie ma w tym pliku wszystkiego, chociażby musimy sami dopisać ignorowanie katalogu `.idea/`.

Plik `.gitignore` jest zwykłym plikiem tekstowym, który jest specjalnie traktowany przez **Git**. Oznacza to, że musimy ten plik normalnie za**commit**tować do repozytorium i normalnie podlega on wersjonowaniu tak jak inne pliki w **Git**. Więcej na temat `.gitignore` można przeczytać w dokumentacji.