Inżynieria Internetu Rzeczy

Materiały Warsztatowe

Moduły i systemy Internetu Rzeczy

Warsztat 5: Oprogramowanie inżynierskie do pracy grupowej

Autorzy: dr inż. Konrad Markowski

Piotr Araszkiewicz

Spis treści

Część 1 – Wstęp

Niniejszy dokument stanowi pewnego rodzaju wstęp do metod zarządzania procesami w projektach inżynierskich. W treści tego podręcznika, autorzy starali się umieścić przydatne informacje, które umożliwią lepszą organizację pracy dla przedmiotów projektowych, ale również w przyszłej pracy zawodowej. Obecnie, właściwie w każdym obszarze branży IT przedstawione metody są wykorzystywane przez pracodawców korporacyjnych, ale również mniejsze przedsiębiorstwa, również start-up'y. Autorzy podręcznika mają nadzieję, że ten zwięzły zbiór treści będzie pomocy dla czytelników i przyczyni się do efektywnego przyswajania wiedzy podczas procesu nauczania na kierunku Inżynieria Internetu Rzeczy.

1.1. Motywacja – dlaczego warto korzystać z takich narzędzi

Proces realizacji projektu inżynierskiego, zwłaszcza takiego, który zakłada wytworzenie szeregu elementów, zarówno o charakterze sprzętowym, jak i programowym, zazwyczaj jest bardzo złożony. Typowo, w pracy inżynierskiej, bardzo ważnym elementem jest dotrzymywanie terminów realizacji zadań, składających się na cały projekt inżynierski. Wszelkiego rodzaju opóźnienia, w trakcie realizacji konkretnych działań z kolei mogą powodować opóźnienie realizacji całego projektu, bądź innych negatywnych zjawisk, np. powszechnie znane zjawisko **crunch'u** (wielomiesięcznych, codziennych nadgodzin zespołu projektowego), często występujące w przypadku produkcji gier komputerowych.

O ile nie sposób przewidzieć wielu problemów natury merytorycznej, które podczas realizacji projektu mogą wystąpić, o tyle sama kontrola realizacji poszczególnych zadań pozwala na szybką diagnozę problemu i wdrożenie odpowiednich działań, na jak najwcześniejszym etapie.

Oprócz samego aspektu związanego z kontrolą procesu projektowego, niezwykle ważne jest wykorzystanie narzędzi, które pozwala na odpowiednią synchronizację działań. Zarówno, podczas pracy na studiach, jak i pracy zawodowej, konieczne jest stosowanie narzędzi umożliwiających grupową implementację oprogramowania, jak również kontrolę wersji wytwarzanego oprogramowania. Głównym atutem systemów kontroli wersji jest możliwość współbieżnej pracy wielu osób nad tym samym projektem. Wiąże się z tym również możliwość jednoczesnego dodawania nowych funkcji(ang. features) oraz wprowadzania poprawek (hotfixes) do produkcyjnie działającego kodu. W praktyce, podczas rozwoju oprogramowania członkowie zespołu dokonują zmian, wpływających na cały projekt. W efekcie takich zmian, program może utracić swoją funkcjonalność, a nawet przestać działać. W takich przypadkach, bardzo wygodny jest powrót do **ostatniej działającej wersji** oraz identyfikacja zmian w kodzie, które spowodowały te niedogodności.

1.2. Komunikacja oraz zarządzanie pracą

W dobrym zespole inżynierskim, bardzo ważnym elementem jest aspekt związany z prawidłową komunikacją między jego członkami. Należy pamiętać o otwartości i wspólnym omawianiu problemów, gdy takie się pojawią podczas realizacji prac projektowych. Dobra komunikacja zespołowa jest oparta na organizacji regularnych

spotkań, w których omawiany jest postęp projektu i ewentualne zmiany w założeniach poszczególnych zadań projektowych. Ważny jest również wybór sposobu komunikacji, który będzie wygodny i odpowiedni dla wszystkich członków zespołu.

Ważne!

- Staraj się organizować regularne spotkania projektowe, a jeśli jest to konieczne, bądź bardziej wygodne – organizuj spotkania w formie online
- Mów otwarcie o problemach związanych z realizacją konkretnego zadania
- Informuj o opóźnieniach i zaznaczaj ten fakt w odpowiednim komunikacie/narzędziu
- Dokonaj, w porozumieniu z zespołem, wyboru odpowiedniego środowiska pracy i komunikacji

1.3. Zarządzanie zespołem – podział zadań

Kolejnym ważnym aspektem, z punktu dobrej organizacji pracy, jest również odpowiedni podział zadań pomiędzy konkretnymi członkami zespołu. Ważne jest, żeby pracę podzielić, w miarę możliwości, równomiernie, tak by nie dopuszczać do przeciążenia pracą jednej konkretnej osoby. W praktyce, często stosuje się grupowanie zadań w większe paczki, obejmujące pewien większy zakres. Niemniej jednak, nie należy unikać określania zadań szczegółowych, o ile jest to możliwe – tak jak w powyższym tekście zostało to opisane, pozwoli to na precyzyjną kontrolę postępów i, w miarę potrzeby, adekwatne działanie całego zespołu. Ważne również jest, by konkretny członek zespołu odpowiadał za realizację prac w ramach danego obszaru.

Ważne!

- Podziel projekt na obszary, a następnie obszary na konkretne aktywności
- Staraj się podzielić pracę równomiernie pomiędzy członków zespołu
- Określ osobę odpowiedzialną za realizację danego obszaru

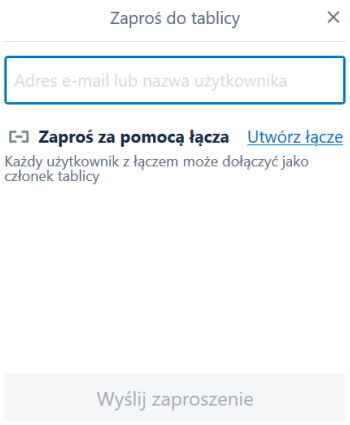
Część 2 – Narzędzia usprawniające zarządzanie zespołem i pracą w zespole.

W ramach tej części podręcznika warsztatowego, zostanie opisanych kilka narzędzi do organizacji pracy w zespole inżynierskim. Należy podkreślić, że nie zostały omówione wszystkie popularne narzędzia, które są stosowane, a tylko niektóre z nich. Niemniej jednak, opisana baza będzie dobrą podstawą do nauki dobrych praktyk w pracy grupowej nad projektem inżynierskim.

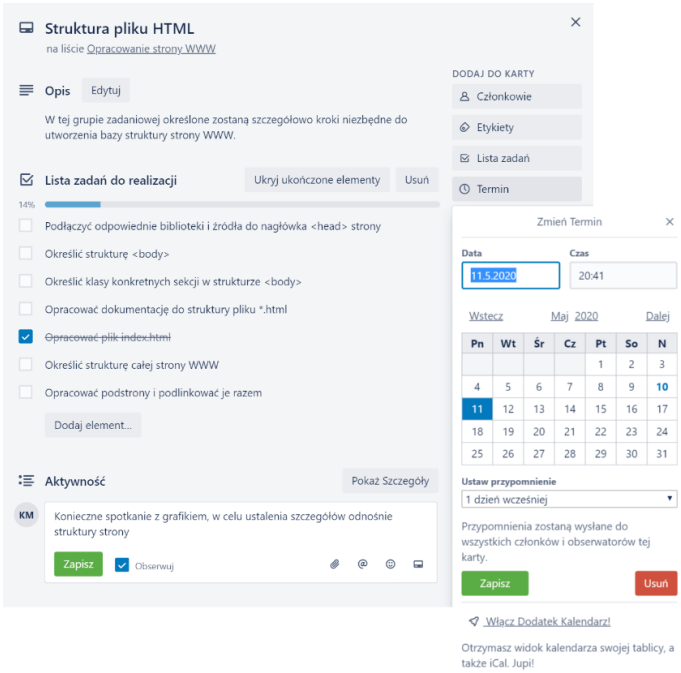
2.1. Trello

Trello jest popularnym narzędziem online do planowania oraz kontroli postępów w projekcie. Narzędzie to jest interesujące przede wszystkim, ze względu na fakt, że możliwe jest uruchomienie go z poziomu aplikacji na wszelkich typach urządzeń, jak również dostęp do aplikacji możliwy jest przez stronę WWW.

W Trello, pierwszym etapem rozpoczęcia pracy jest utworzenie dedykowanej tablicy dla projektu. Tablice mają charakter **prywatny** bądź **publiczny**. Charakter publiczny tablicy oznacza, że dostęp do takiej tablicy możliwy dla każdego, a dane dotyczące projektu, które w niej zostaną umieszczone, będą dostępne dla każdego użytkownika Internetu. Dlatego tez, zaleca się tworzenie tablic prywatnych. Dostęp do takiej tablicy można zrealizować przez udostępnienie członkom zespołu linku, bądź wysłanie zaproszenia na adres e-mail.



Rys.1 Zapraszanie członków zespołu do tablicy.



Rys. 2 Dodawanie listy zadań do realizacji.

Po wysłaniu zaproszenia do wszystkich członków zespołu, w następnym kroku, w celu utworzenia danego projektu, można utworzyć listę o konkretnej nazwie, w ramach której możliwe jest utworzenie kart zawierających szczegółowy podział na grupy zadań.

Każda karta może zostać szczegółowo opisana, a do karty dołączona lista zadań do realizacji. Ponadto, termin realizacji zadań dla całej karty może zostać określony i podłączony do odpowiedniej aplikacji, np. kalendarza w telefonie. Trello również może wysłać powiadomienie na adres e-mail członków zespołu, przypominające o zbliżającym się terminie realizacji grupy zadań. Narzędzie również umożliwia odkreślenie wykonanego zadania z listy zadań. Wygodne jest również możliwość dodawania komentarzy do karty, informującej resztę zespołu np. o konieczności podjęcia konsultacji.

2.2 Slack

Slack jest darmową aplikacją, pozwalającą na skuteczne zarządzanie komunikacją w zespole. Jest to aplikacja pozwalająca na efektywną komunikację, jak również organizację pracy. Opis narzędzia można znaleźć pod następującym linkiem: https://slack.com/intl/en-pl/resources/using-slack/slack-tutorials.

2.3 MS Teams

MS Teams jest zestawem narzędzi, bazujących na technologii chmury, ułatwiające zdalną pracę zespołową. Dla studentów Politechniki Warszawskiej, MS Teams jest usługą darmową. Wszelkie informacje na temat funkcjonalności, jak również szkolenia korzystania z tego narzędzia, można znaleźć na oficjalnych stronach Microsoftu, pod adresem: https://support.microsoft.com/pl-pl/teams.

Część 3 – Narzędzia do kontroli wersji oprogramowania

Jednym z podstawowych narzędzi pracy dla każdego programisty, są wszelakie systemy kontroli wersji. Narzędzia takie nie tylko pozwalają na zwiększenie efektywności pracy grupowej, ale również umożliwiają powrót do wcześniejszych wersji oprogramowania, czy plików. Aspekt ten jest niezwykle istotny, ponieważ pozwala na dokonywanie zmian, bez fundamentalnego ryzyka „zepsucia" całego, pisanego przez zespół programistów, oprogramowania, w przypadku wystąpienia błędu w kodzie. Ponadto, praca na systemie kontroli wersji umożliwia dosyć elastyczne dodawanie kolejnych funkcjonalności do budowanego oprogramowania. Umiejętność poprawnego korzystania z systemów kontroli wersji to jedno z najważniejszych, praktycznych umiejętności wymaganych przez pracodawców na obecnym rynku pracy w IT!

W praktyce, istnieje wiele różnych środowisk umożliwiających pracę grupową. Niemniej jednak, najbardziej popularnym obecnie środowiskiem, jest środowisko GIT. Poniższa część podręcznika opisywać będzie tylko najważniejsze aspekty związane z uruchomieniem środowiska i korzystaniem z niego. Należy pamiętać, że to bardzo rozbudowane narzędzie, a dokładne jego poznanie zwyczajnie wymaga praktyki.

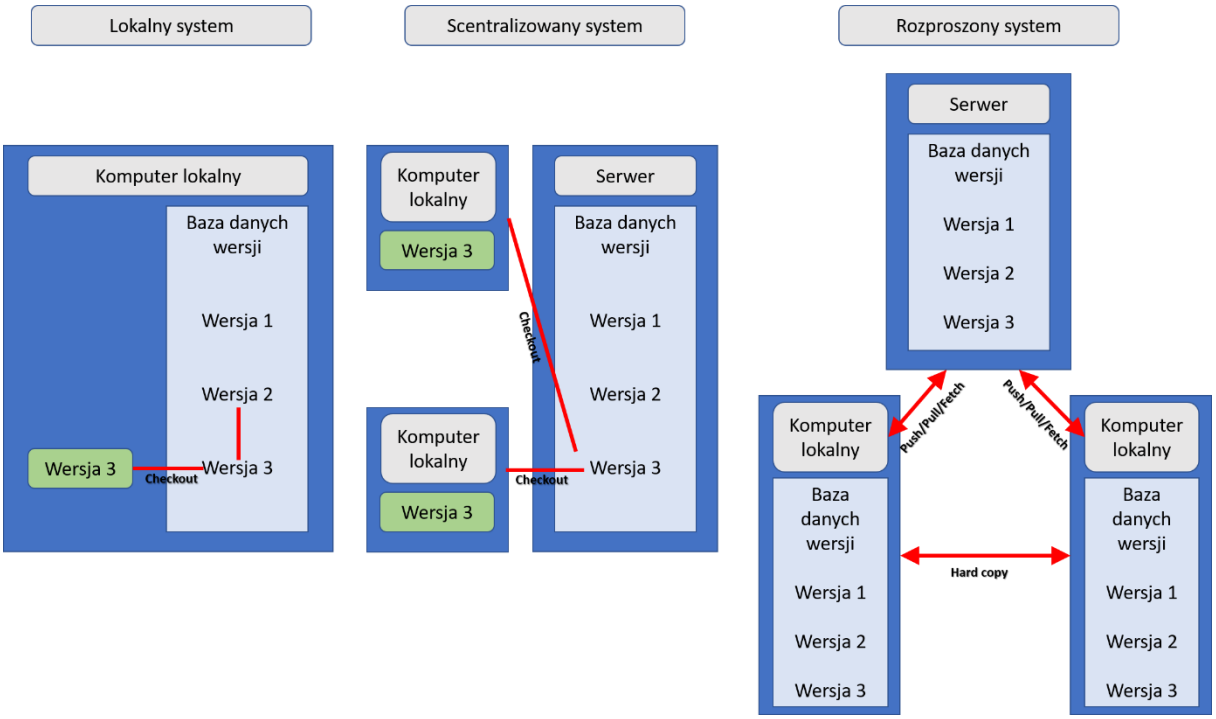
Z pewnością poniższy opis nie wyczerpie w całości aspektów związanych z GITem. Na szczęście, to bardzo popularne środowisko! Na większość pytań, które zrodzą się podczas korzystania z tego narzędzia znajdziesz odpowiedź w sieci, a większość aspektów została opisana w książce Pro Git, napisanej przez Scotta Chacona oraz Bena Strauba.

Książka jest w pełni darmowa, a opisane w niniejszym skróconym podręczniku aspekty GITa, właśnie powstały na bazie informacji zawartej w tej książce. Pełen opis (częściowo dostępny w języku Polskim) środowiska można znaleźć pod następującym linkiem: https://git-scm.com/book/pl/v2.

3.1. Typy systemów kontroli wersji

Tak jak wcześniej zostało to opisane, systemy kontroli wersji pozwalają na śledzenie wszelkich zmian dokonywanych na pliku, bądź wielu plikach – dzięki takiej funkcjonalności, możliwe jest przywołanie wcześniejszych wersji i powrót do „starego" kodu. Narzędzie to nie ogranicza się do kontroli wersji dla predefiniowanych języków programowania – może z powodzeniem służyć do kontroli rozwoju aplikacji, bądź skryptu tworzonego w dowolnym standardzie. Niemniej jednak główną istotną cechą pracy w systemach kontroli wersji jest fakt, że umożliwiają one bardzo sprawną współpracę przy tworzeniu aplikacji i kodów w bardzo dużych zespołach programistycznych.

W praktyce można rozróżnić trzy typy systemów kontroli wersji – lokalne, scentralizowane oraz rozproszone. Architektura takich systemów została zaprezentowana na rysunku 3.



Rys. 3 Architektura systemów kontroli wersji.

Najbardziej intuicyjną metodą kontroli wersji jest metoda bazująca na **lokalnym systemie kontroli**. W praktyce, by zastosować taki system kontroli plików, nie jest właściwie potrzebne jakiekolwiek środowisko pracy. Zwyczajnie, odpowiednie wersje w takim podejściu mogą być kopiowane do zadanych katalogów ręcznie, również z nazwami wersji. W przypadku realizowania samodzielnie prostych projektów, bądź opracowania skryptów do pojedynczych działań, takie podejście jak najbardziej może być stosowane. Niemniej jednak, im bardziej rozbudowany projekt, tym większe ryzyko wystąpienia błędu. By eliminować błędy wynikające z „ręcznego" kopiowania plików, opracowanych zostało

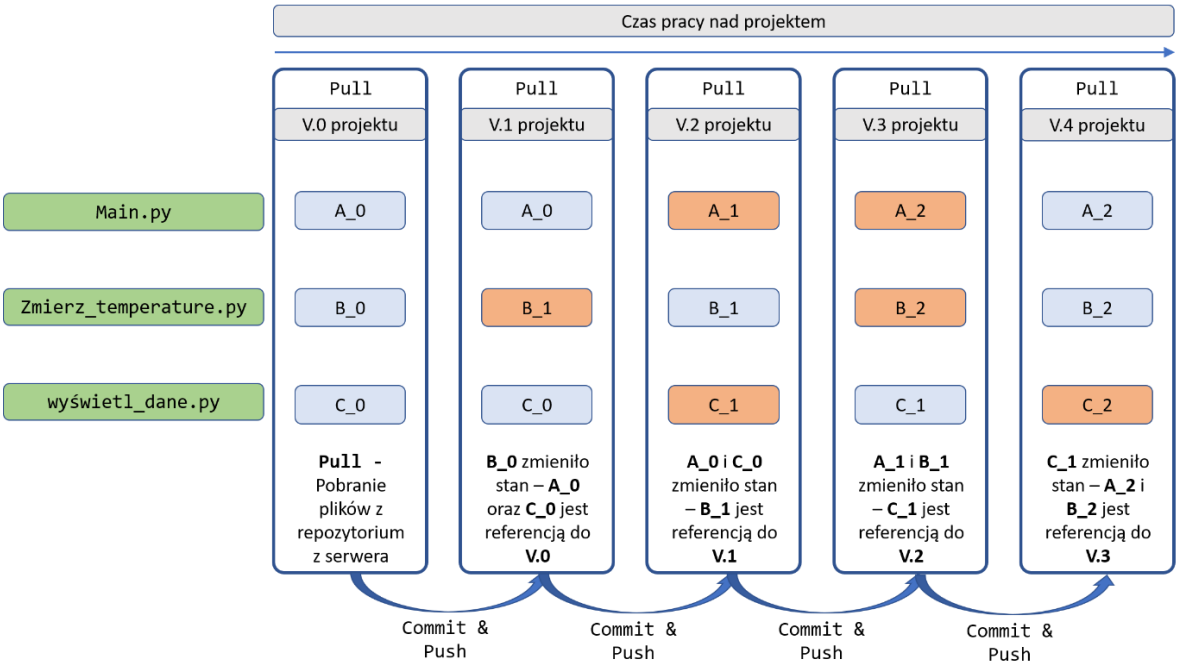
szereg narzędzi – najpopularniejszym z nich jest system RCS, nawet aktualnie dostępny w systemach operacyjnych Mac OS X (by system ten uruchomić należy zainstalować Narzędzia Programistyczne). RCS zapisuje w dedykowanym formacie plików dane różnicowe pomiędzy wersjami, umożliwiające przywołać poprzednie stany plików z dowolnego momentu.

O ile systemy lokalne sprawdzają się w przypadku pracy indywidualnej, o tyle dla pracy grupowej bywają bardzo nieefektywne. W tym celu opracowane zostały **scentralizowane systemy kontroli**, bazujące na wykorzystaniu serwera, w którym to umieszczone jest repozytorium (baza danych wersji) kodu programu. W takiej architekturze, programiści mają dostęp do **najnowszej wersji plików**, które mogą Takie podejście umożliwia bardzo łatwą kontrolę dostępu i rozwoju oprogramowania przez administratora repozytorium. Niemniej jednak, w przypadku wystąpienia uszkodzenia danych na serwerze, istnieje poważne ryzyko utraty całości wytworzonego oprogramowania. Ponadto, takie podejście wymaga łączenia się z serwerem, w którym baza repozytorium została umieszczona. Dlatego też, w przypadku utraty połączenia (np. fizyczne zerwanie łącza, czy problem z dostępem do Internetu u jednego z użytkowników), może okazać się, że praca nad rozwojem oprogramowania będzie musiała zostać wstrzymana.

By wyeliminować powyższe problemy związane z realizacją oprogramowania, zaproponowano architekturę systemu będącą **rozproszonym systemem kontroli**. Względem systemu scentralizowanego, w takiej architekturze, każdy użytkownik posiada na swoim własnym fizycznym nośniku kopię całego, umieszczonego na serwerze, repozytorium kodu. W efekcie, w momencie gdy jedna z jednostek, przechowująca kod programu, ulegnie awarii, repozytorium znajdujące się u jednego z klientów może zostać łatwo przekopiowane i przywrócone. Architektura taka może być traktowana jak połączenie własności repozytorium centralnego i lokalnego. Istnieje wiele systemów bazujących na architekturze rozproszonej, m.in. Git, Mercurial, czy Baazar. Obecnie, najbardziej popularnym rozproszonym systemem kontroli wersji, jest środowisko Git.

3.2. System kontroli wersji Git

Jak zostało to stwierdzone w poprzedniej części tego rozdziału **Git jest obecnie jednym z najpopularniejszych systemów kontroli wersji**. Jego popularność wynika bezpośrednio z efektywności tego systemu. W momencie zapisu stanu projektu (`commit`), system Git tworzy obraz tego, jak w aktualnej wersji wyglądają pliki. Co również istotne, zapisywane w tym systemie są dla każdej następnej wersji tylko te pliki, które są zmodyfikowane, a te które się nie zmieniły zapisywane są jako referencja do poprzedniej, zupełnie identycznej wersji. By w sposób bardziej jasny przedstawić charakter działania systemu, na rysunku 4 zaprezentowano schemat działania Gita, na przykładzie zmian wykonywanych na trzech plikach. Umówmy się, że pierwszy plik `main.py` jest podstawowym plikiem programu, plik `zmierz_temperature.py` zawiera funkcję, która dokonuje odczytu temperatury, natomiast `wyświetl_dane.py` realizuje funkcję wyświetlania odczytanych danych na wyświetlaczu. Do realizacji takiego projektu, przypisano trzy osoby – każda z osób poproszona jest o napisanie kodu, realizującej zadane funkcjonalności.



Rys. 4 Przechowywanie danych w systemie Git.

Na powyższym schemacie kolorem niebieskim oznaczono te wersje plików, które są referencją do pliku z poprzedniej wersji. Natomiast kolor pomarańczowy oznacza **nowy stan pliku** – przez nowy stan pliku zwyczajnie rozumie się zmianę w kodzie źródłowym, zapisanym w pliku (np. dodanie kolejnej zmiennej, czy opisanie zadanego procesu pomiarowego pętlą `for`). Należy podkreślić, że standardowo, Git dodaje nowe dane do bazy. To w efekcie powoduje, że bardzo trudno jest przeprowadzić taki proces, który uniemożliwi powrotu do poprzedniej wersji. W praktyce, na bazie powyższego przykładu, można bez problemu powrócić z V.3 projektu do np. V.1, czy V.0, będącego pierwotnym stadium.

W powyższym schemacie zaznaczono, że pomiędzy kolejnymi wersjami projektu, programista dokonujący zmian przeprowadza proces `Pull`, `Commit` oraz `Push`. W praktyce są to trzy najczęściej wykorzystywane komendy środowiska Git, jakimi posługuje się użytkownik. Komenda `Pull` umożliwia ściągnięcie repozytorium programu z zewnętrznego serwera, do którego użytkownik jest podłączony. Komenda `Commit` natomiast oznacza zapis danej wersji w lokalnym repozytorium użytkownika, zapisanym na dysku twardym. Natomiast komenda `Push` oznacza przesłanie wersji repozytorium użytkownika na serwer zewnętrzny

Kluczowe w kontekście pracy z Git jest zapamiętanie, że każdy plik, który jest opracowywany, może posiadać trzy stany: zatwierdzony, zmodyfikowany oraz śledzony. Stan **zatwierdzony** oznacza, że dane zostały bezpiecznie zachowane w lokalnej bazie danych. Stan **zmodyfikowany** oznacza, że plik został zmieniony, ale zmiany jeszcze nie zostały wprowadzone. Natomiast stan **śledzony** oznacza, że środowisko Git obserwuje zmiany dokonane w pliku i porównuje je z ostatnią zatwierdzoną wersją pliku. Zatwierdzenie zmian odbywa się właśnie przez użycie komendy `Commit`.

Oczywiście, przed rozpoczęciem pracy w środowisku Git, należy je zainstalować – plik instalacyjny środowiska do wszystkich obecnie najpopularniejszych środowisk systemów operacyjnych znajduje się pod adresem: https://git-scm.com/.

Samo środowisko natywnie obsługuje się z poziomu konsoli – w przypadku systemów operacyjnych z poziomu Windows PowerShell, natomiast dla środowiska Linux jest to `bash`. W praktyce, większość programistów oraz deweloperów kodu korzysta z nakładek graficznych, które ułatwiają wykorzystanie Gita. Niemniej jednak nakładki graficzne mogą wprowadzać pewne ograniczenia, zwłaszcza przy pracy z dużymi repozytoriami. Dlatego też, doświadczony deweloper kodu powinien posiadać wiedzę o możliwej obsłudze środowiska Git z poziomu konsoli.

Bardzo istotnym elementem korzystania z samego środowiska, jest uruchomienie repozytorium na zdalnym serwerze. Duże korporacje zazwyczaj posiadają własne zasoby serwerowe – niemniej jednak, na potrzeby rozwoju swojego oprogramowania można skorzystać z wielu dostępnych w sieci zasobów, takich jak **GitHub**, **GitLab**, czy **BitBucket**. Dla studentów Politechniki Warszawskiej wszystkie te repozytoria są całkowicie darmowe.

3.2.1 Podstawy środowiska Git na lokalnym komputerze

Instalacja i konfiguracja środowiska na lokalnym komputerze jest bardzo prosta – wystarczy jednie pobrać ze strony https://git-scm.com/ odpowiednią, dla swojego systemu operacyjnego, wersję środowiska. Po zainstalowaniu środowiska, należy je skonfigurować, przez podanie swojej sieciowej tożsamości oraz wybranie docelowego edytora kodu. Określenie tożsamości w środowisku Git jest bardzo prosta – w bash, bądź Windows PowerShell wystarczy wpisać:

```
$ git config --global user.name "Student_Studencki"
$ git config --global user.email s.studencki@pw.edu.pl
```

Spróbuj teraz wpisać polecenie:

```
$ git config --list
```

by sprawdzić, że faktycznie w ustawieniach konfiguracyjnych Gita widnieje Twoje imię i nazwisko oraz adres e-mail. Sprawdzenie wartości danej zmiennej w środowisku może odbyć się przez bezpośrednie zapytanie się o tylko tę zmienną. Spróbuj wpisać poniższą komendę w środowisku bash lub Windows PowerShell, by sprawdzić, czy do środowiska zostały wprowadzone Twoje poprawne dane:

```
$ git config user.name
```

Aspekt identyfikacji użytkownika jest bardzo istotny - w przypadku pracy z repozytorium zdalnym w grupie, umieszczonym na serwerze, pozwoli na identyfikację, jaka osoba wykonała procedurę `commit/push`, bądź umożliwi nadanie odpowiednich uprawnień do łączenia plików, najbardziej doświadczonej w zespole osobie.

Tak jak zostało to wcześniej wspomniane, pliki w środowisku Git posiadają trzy stanach, a w szczególności mogą przebywać w stanie śledzenia (sprawdzania, czy zostały dokonane jakieś zmiany w opracowywanym kodzie programu). Zazwyczaj, wywołując procedurę `commit` (które zazwyczaj wykonuje się relatywnie często), użytkownik środowiska Git oczekuje przeniesienia wszystkich opracowywanych plików do repozytorium. Niemniej jednak, zdarzają się sytuacje, gdy podczas testów oprogramowania wygenerowane zostaną dodatkowe dane wrażliwe, które pod żadnym pozorem nie mogą

znaleźć się w Internecie, takie jak dane osobowe, wyniki pomiarów oraz przede wszystkim **hasła i klucze do systemów** – błędne umieszczanie takich danych jest bardzo powszechne zwłaszcza wśród młodych programistów. W takim przypadku, w szczególności, gdy do obsługi Gita wykorzystywana jest konsola poleceń, bardzo uciążliwe jest za każdym określanie plików jakie mają być pominięte. Szczęśliwie, w środowisku Git możliwe jest utworzenie pliku `.gitignore` zawierającego listę nazw plików, bądź rozszerzeń plików, jakie mają być pomijane w procedurze `commit`. Poniżej znajduje się przykład konstrukcji pliku `.gitignore`, wraz z odpowiednią składnią[1]:

```
# Wyklucz wszystkie pliki z rozszerzeniem .a
*.a

# mimo wykluczenia plików z rozszerzeniem .a, commituj plik lib.a
!lib.a

# wyklucz folder TODO, będący w korzeniu katalogu, ale comituj już pliki w
# katalog/TODO
/TODO

# Wyklucz wszystkie pliki w katalogach build/
build/

# Wyklucz doc/notes.txt, lecz nie wykluczaj plików w katalogu np.
# doc/server/arch.txt
doc/*.txt

# wyklucz wszystkie pliki .txt w katalogach doc/
doc/**/*.txt
```

3.2.2 Rozgałęzianie projektu – Git Branches

Możliwość wprowadzenia rozgałęzienia w systemach kontroli wersji stanowi fundamentalny argument, dlaczego warto korzystać z tego typu środowiska. W trakcie rozwoju oprogramowania, często pojawia się koncepcja dodania nowej funkcjonalności do opracowywanej aplikacji, czy urządzenia – dla systemów Internetu Rzeczy może to być to funkcja zapewniająca statystyczną analizę danych z czujników, czy chociażby wyposażenie elektroniki w dodatkowy element, taki jak na przykład kamera. W takim przypadku, by nie ryzykować utraty postępów w głównym trzonie aplikacji, środowisko Git oferuje możliwość rozgałęziania projektu – w terminologii Gita takie rozgałęzienie określamy jako `branch`.

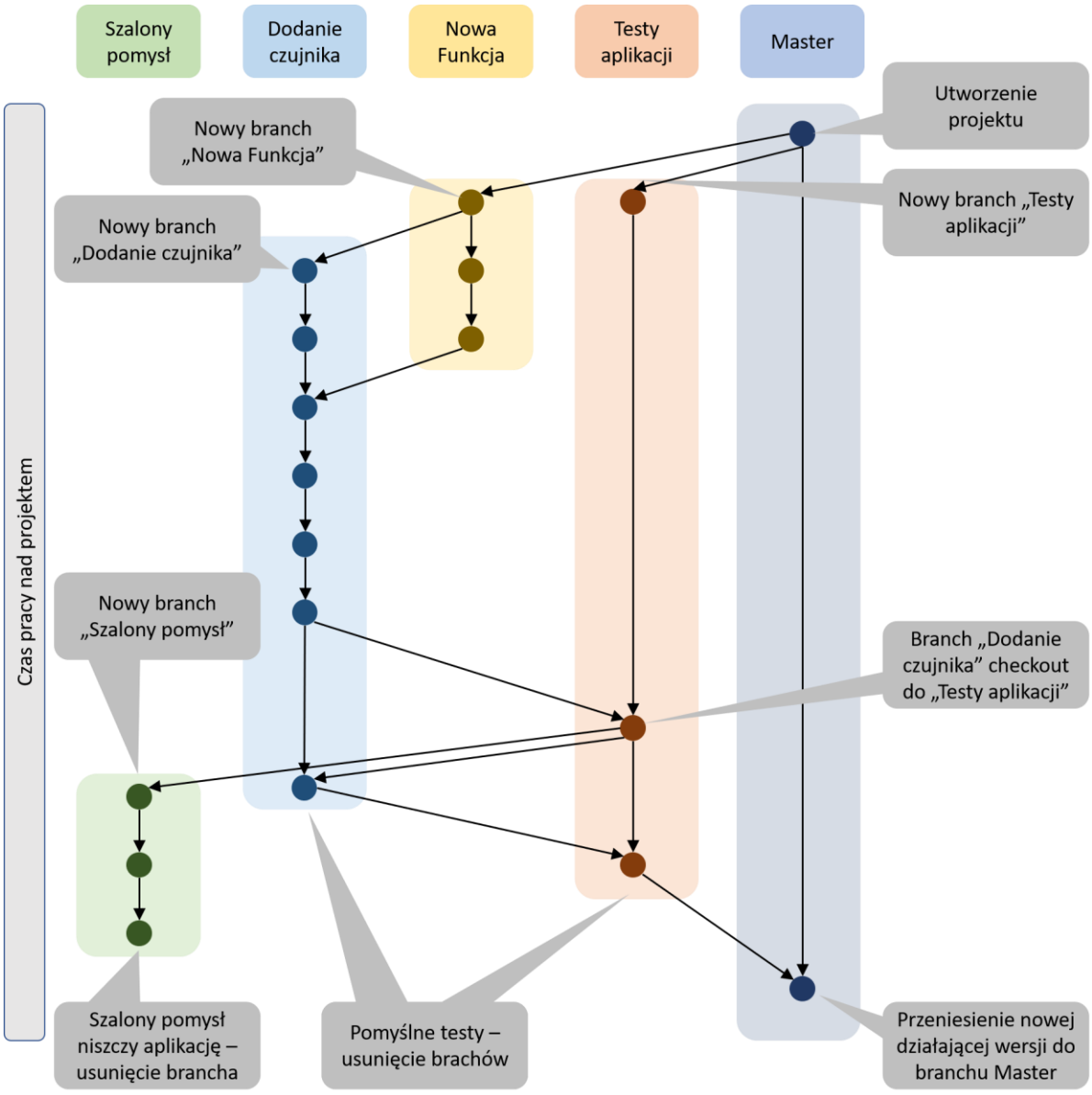
Czym zatem jest gałąź? Jest to wskaźnik na zestaw zmian, dokonanych przez programistów w opracowywanym kodzie. Ze względu na właśnie taką konstrukcje tego systemu jest on niezwykle efektywny, w porównaniu z konkurencyjnymi narzędziami. Pierwotnie, Git został opracowany na potrzeby rozwoju jądra systemu operacyjnego Linux – projekt ten jest niezwykle obszerny, dlatego też efektywność była kluczowym aspektem, który został wzięty pod uwagę przez architekta tego rozwiązania, Linusa Torvaldsa.

Pierwotnie, przy utworzeniu repozytorium w danym katalogu, przez komendę `git init` tworzona jest gałąź pierwotna, nazwana domyślnie `master`. Choć gałąź ta nie posiada

[1] Tekst pliku, znajdujący się za znakiem # jest traktowany przez środowisko Git jako komentarz autora i jest nieinterpretowany.

specjalnych cech, w środowisku programistycznym przyjmuje się, że jest to gałąź głównego, działającego programu, w którym umieszczana jest aktualna, najbardziej zaawansowana i przede wszystkim przetestowana oraz działająca wersja oprogramowania. Co również istotne, w korzeniu tej gałęzi znajdować się będzie (a właściwie powinien się znajdować) pierwotny i pierwszy `commit`, wykonany w danym repozytorium. Każde kolejne liście gałęzi `master`, bądź innych, dodatkowych, przez właśnie wykorzystanie wskaźnika odwoływać się będą to pierwszej wersji oprogramowania.

Dzięki takiemu „gałęziowemu" podejściu, środowisko Git umożliwia bardzo dokładne śledzenie zmian, dokładanie nowych funkcjonalności do rozwijanego oprogramowania, jak również utrzymanie porządku w zestawie wersji, w raz z możliwością powrotu do wersji wcześniej ukończonej, a przede wszystkim bardzo ułatwia pracę dużym zespołom programistycznym. Szczegółowa koncepcja gałęzi została przedstawiona na rysunku 5.



Rys. 5 Schemat pracy przy wykorzystaniu wielu gałęzi do wytworzenia przykładowego oprogramowania.

Na powyższym rysunku, każda kropka symbolizuje oddzielną procedurę `commit`. W praktyce, jeśli opracowujesz kod, nie musisz dokonywać procedury `commit` do gałęzi, z której pliki zostały pobrane – możesz wysłać pliki do dowolnej gałęzi a Git sam zinterpretuje pierwotne źródło plików. Co również istotne, częściowo napisany kod może być przesłany do innej gałęzi i z nią połączony (`merge`). Tak jak w przypadku gałęzi „Szalony Pomysł" można bez konsekwencji usuwać gałęzie, jeśli rozwijany na niej kod okaże się klapą.

3.2.3 Łączenie rozgałęzionego projektu w całość – procedura merge

Scalanie kodu jest jedną z najbardziej istotnych funkcjonalności wykorzystywania repozytorium GIT. Najpewniej, podczas pracy w projekcie, wiele osób może dokonać modyfikacji jednego i tego samego pliku. Najczęściej powoduje to swego rodzaju kolizje – w efekcie, przy braku czujności, może dojść do sytuacji, że połączenie dwóch plików w jeden będzie skutkować brakiem możliwości kompilowania oprogramowania. Dlatego też, zazwyczaj do scalania plików powinno delegować się jedną, odpowiedzialną za to osobę.

Przeanalizujmy proces scalania na przykładzie, w którym okazuje się, że w trakcie pracy programistycznej, następuje konieczność dodania jednej, kluczowej funkcjonalności, Nazwijmy taką poprawkę jako `featureX`. Utwórzmy zatem nową gałąź i nazwijmy ją właśnie `featureX`. By utworzyć gałąź posłużymy się komendą `git branch`, w następujący sposób:

```
$ git branch featureX
```

Teraz, by przełączyć się na nowo utworzoną gałąź, musimy posłużyć się poleceniem `git checkout featureX`. Niemniej jednak, nie jest konieczne wpisywanie poleceń krok po kroku. Cała wyżej opisana procedura może być zrealizowana przez wpisanie jednej komendy – wystarczy w poleceniu `checkout` dodać flagę `-b`. W takim przypadku GIT jednocześnie utworzy nową gałąź i automatycznie się do niej przełączy. W takim wypadku należy wpisać następujące polecenie:

```
$ git branch -b featureX
```

Teraz można zacząć spokojnie pracować, bez ryzyka, że działający kod w głównej gałęzi przestanie się kompilować.

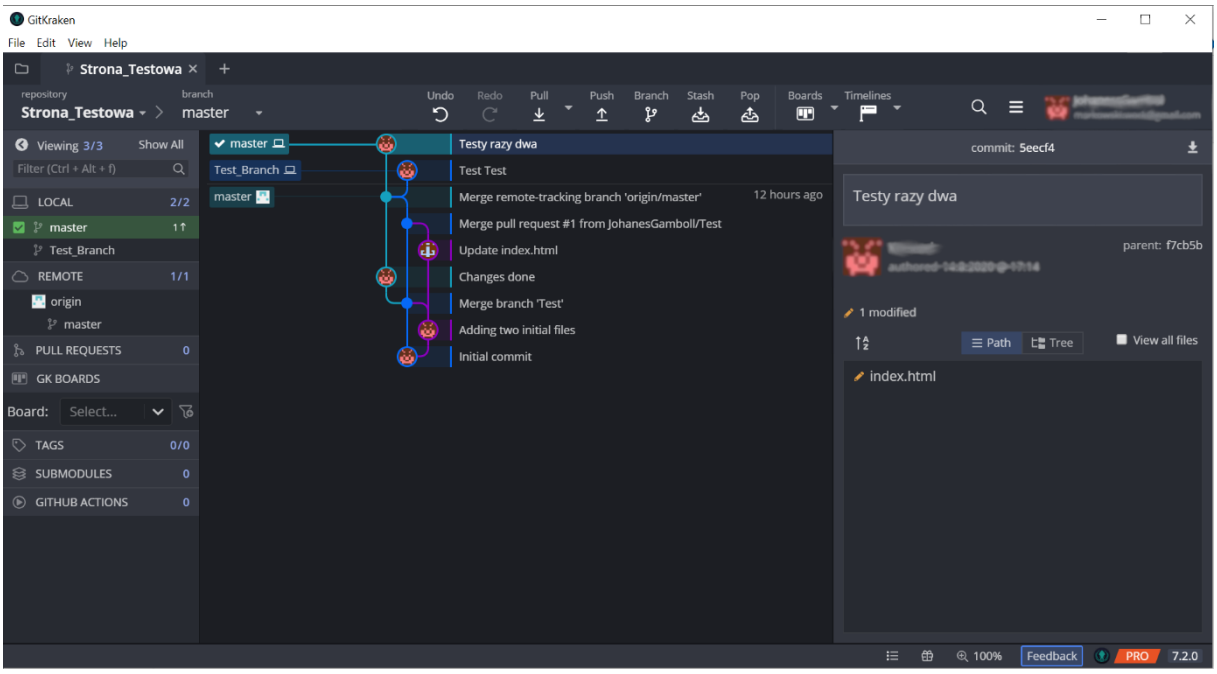
Rozważmy następujący scenariusz: po kilku commitach w gałęzi `featureX`, okazuje się, że należy wprowadzić szybką poprawkę do gałęzi master. W nomenklaturze programistycznej, często takie poprawki nazywa się `hotfix`. W tym wypadku należy wrócić do brancha master, ponieważ właśnie tę wersję kodu chcemy poprawić. Należy zauważyć, że zmiana gałęzi nie będzie możliwa dopóki zmiany w folderze nie są zacomitowane lub zestashowane dla gałęzi `featureX`. Po zmianie gałęzi należy utworzyć gałąź `hotfix`. Należy pamiętać, że Rozgałęzienie nastąpi w miejscu (commicie) na którym obecnie wskazujemy – po wcześniejszym przełączeniu się do gałęzi master, właśnie kopia tej wersji kodu zostanie umieszczona w gałęzi hotflix. Po wprowadzeniu poprawek można wykonać `merge`, w tym przpadku w trybie `fast-forward`. To znaczy, że od ostatniego rozgałęzienia, w gałęzi, do której się scala, nie ma zmian. Merge polega wtedy jedynie na przesunięciu wskaźnika brancha na ostatni commit. W tym momencie, `hotfix` jest już umieszczony w gałęzi `master`. Następnie Można wrócić do pracy nad featureX poprzez

checkout brancha `featureX`. Po skończonej pracy nad funkcjonalnościami w tej gałęzi znów należy wykonać `merge` do gałęzi `master`. Taki proces łączenia nazywa się `three-way merge`, ponieważ należy scalić zmiany z gałęzi `featureX`, ze zmianami które pojawiły się od rozgałęzienia z `featureX`. W tej sytuacji stworzony zostanie merge commit. W przypadku, kiedy zmiany będą w tym samych linijkach kodu, najpierw należy rozwiązać konflikty, tzn wybrać z której gałęzi linijka ma być dodana do merge commita. Po tej operacji w masterze są już zmiany z brancha `hotfix` oraz `featureX`.

3.3. Nakładki graficzne na Git

W sensie podstawowym, wszelkie operacje w środowisku Git dokonywane są z poziomu konsoli. Niemniej jednak, bardzo popularne wśród wszelkich programistów są nakładki graficzne. Formalnie, jest to zwyczajny interfejs, który komunikuje się z konsolą, ale bez konieczności ręcznego wpisywania komend w interfejsie `bash`, czy Windows PowerShell.

Jedną z najbardziej popularnych nakładek graficznych wśród programistów jest nakładka GitKraken. Zrzut ekranu tej aplikacji został przedstawiony na rysunku 6.



Rys. 6 Interfejs graficzny programu GitKraken, ułatwiającego korzystanie ze środowiska Git.

O ile nakładki graficzne pozwalają na zdecydowane uproszczenie pracy, często posiadają swego rodzaju ograniczenia, zwłaszcza, gdy konieczna jest praca z bardzo dużym repozytorium. W takich przypadkach zawsze niezawodnie sprawdza się skorzystanie z natywnego interfejsu Gita, czyli konsoli.

Dodatek – Krótki słowniczek najważniejszych pojęć w Git

`.gitignore`	Zawiera definicję plików, które będą ignorowane przez środowisko Git
`Fetch`	Sprawdzenie, czy na serwerze dokonano jakichś zmian, względem wersji lokalnej
`Pull`	Ściąganie wersji repozytorium z serwera
`Stage`	Ustawia zadany plik jako „śledzony” przez Gita – wymagane, żeby poprawnie dokonać zapisu zmian w repozytorium (commitu)
`Unstage`	Przestaje śledzić zadany plik
`Commit`	Zapisanie zmian w kodzie źródłowym na lokalnym komputerze użytkownika
`Push`	Wysłanie zapisanych zmian z lokalnego komputera użytkownika na serwer
`Branch`	Gałąź repozytorium – pozwala na rozwijanie nowej funkcjonalności, bez ryzyka zniszczenia i niemożności powrotu do kodu pierwotnego