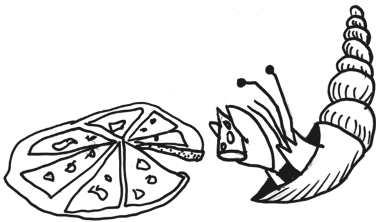
Teoria grafów a systemy kontroli wersji, czyli jak zarządzać wielką firmą informatyczną

Jakub RADOSZEWSKI

Nie jest aż tak trudno założyć firmę informatyczną. Potrzebny jest do tego dobry pomysł na produkt, którego ludzie będą chcieli używać (na przykład urządzenie elektroniczne, program komputerowy lub portal internetowy), oraz kilku programistów-zapaleńców, którzy będą skłonni poświęcić kilka miesięcy życia na stworzenie wstępnej, działającej wersji produktu. W ten właśnie sposób powstała większość znanych obecnie firm z branży technologii informacyjnych, takich jak Microsoft (początkowo związany z interpreterem języka BASIC), Apple (pierwszym produktem był komputer do samodzielnego złożenia), Google (wyszukiwarka) czy Facebook (portal społecznościowy). Później, kiedy nasz produkt zyskuje większą popularność, do jego obsługi i ulepszania potrzeba coraz więcej ludzi. I tak, obecnie, każda z pierwszych trzech wymienionych korporacji liczy już dziesiątki tysięcy pracowników (sam Microsoft ma ponad 90 tys. pracowników), a stosunkowo młody i nieduży Facebook zatrudnia „zaledwie” 3 tys. osób.



Programiści nie potrzebują zbyt wiele do szczęścia. Wielkie firmy zapewniają im mieszkania na wynajem, transport do miejsca pracy (busiki, obowiązkowo z dostępem do bezprzewodowego Internetu), pełen wikt i opierunek, sport oraz rozrywki – i to wszystko w miejscu pracy. Pieniądze na to wszystko zapewniają najpierw firmy sponsorujące (patrz też artykuł Jacka Migdała w numerze), a potem przychody z reklam (patrz też artykuł Karola Kuracha). Pod tym względem zarządzanie firmą informatyczną nie stanowi większego problemu. Inaczej sprawa wygląda, jeśli chodzi o zarządzanie pracą programistów.

Wyobraźmy sobie firmę X, zatrudniającą liczny zespół programistów, który pracuje nad edytorem tekstu o nazwie *Text*. Nietrudno wyobrazić sobie, jakie problemy może rodzić współpraca między programistami firmy. Przede wszystkim, programiści ci muszą w jakiś sposób współdzielić kod źródłowy produktu, nad którym pracują. Gdyby np. wszyscy oni zaczęli przysyłać sobie nawzajem poprawki pocztą elektroniczną, bardzo szybko zapanowałby w tym kompletny chaos. To jednak nie wszystko. Mogłoby się okazać, że dwóch programistów w tym samym czasie postanowiło zmienić ten sam fragment kodu, tyle że każdy z nich zrobił to zupełnie inaczej. Albo gdyby jeden programista niechcący popełnił błąd w programie i, rozpropagowawszy go wszystkim współpracownikom, zablokował ich dalszą pracę... Jeśli dołożyć do tego usterki sprzętowe, które mogłyby spowodować nieodwracalną utratę części kodu projektu, widać wyraźnie, że potrzeba tu jakiejś dobrej metody zarządzania kodem projektu.

Szukanym rozwiązaniem są tytułowe systemy kontroli wersji. Rozważmy, przykładowo, system o nazwie Subversion (SVN). Mamy w nim centralny serwer, zwany repozytorium, na którym znajduje się cały kod projektu. Każdy programista ma na swoim komputerze kopię całego kodu albo większego wycinka, nad którym w danym momencie pracuje. Cykl pracy programisty jest teraz bardzo prosty: pobiera z repozytorium najnowszą wersję kodu (używając polecenia `svn update`), następnie wykonuje swoje zmiany – edytuje pliki, a także dodaje, usuwa i przenosi je wedle uznania (`svn add`, `svn delete`, `svn move`...), na końcu zaś wysyła pakiet swoich zmian z powrotem do repozytorium (`svn commit`). Ponieważ na serwerze utrzymujemy historię wszystkich zmian, więc całą sytuację możemy przedstawić jako bardzo prosty graf – pojedynczą ścieżkę (rys. 1).

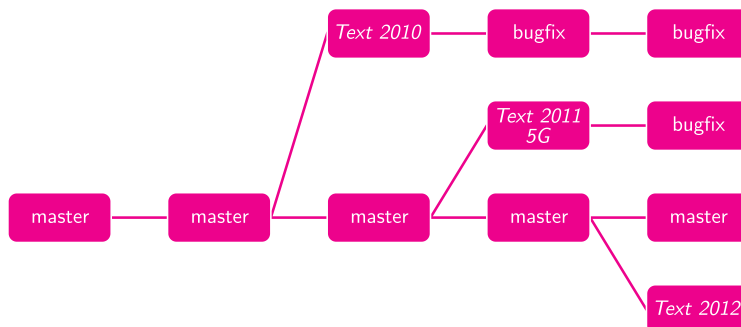
Rys. 1

W ten sposób uporaliśmy się już z problemem współdzielenia kodu i odporności na awarie – wystarczy trzymać kopię zapasową na repozytorium. Łatwo możemy poradzić sobie z błędami programistów. Faktycznie, jeśli zmiany wprowadzone przez jednego programistę powodują, że projekt przestaje poprawnie działać, możemy po prostu wycofać te zmiany i powrócić do wcześniejszej wersji repozytorium. W tym miejscu warto dodać, że każda kolejna wersja repozytorium pamięta jedynie zmiany w stosunku do poprzedniej wersji, gdyż w przeciwnym razie repozytorium szybko urosłoby do ogromnych rozmiarów.

Pozostała jeszcze tylko kwestia konfliktujących zmian. Mogą to być, jak już wspominaliśmy, niezgodne zmiany w ramach jednego pliku, ale zdarzają się również

bardziej złożone sytuacje: np. w tym samym czasie jeden programista wprowadził jakieś zmiany do pliku, a drugi przeniósł fragment tego pliku gdzieś indziej albo w ogóle skasował ten plik. W takiej sytuacji szybszy z konfliktujących programistów ma szczęście i udaje mu się wprowadzić zmiany, natomiast drugiemu z nich komenda `svn commit` nie powiedzie się i będzie on musiał samemu rozwiązać powstały konflikt. Dokładniej, system SVN wskaże miejsca w kodzie, w których wykrył konflikty, i pozwoli programiście ręcznie poprawić stosowne pliki i wysłać ujednolicone zmiany (`svn resolve`).

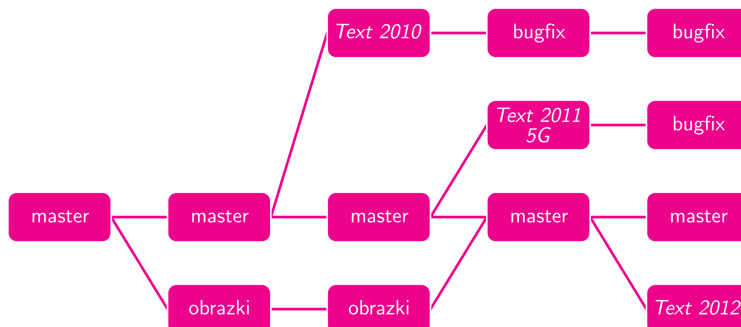
Widzimy, że SVN dobrze radzi sobie z podstawowymi wymaganiami dotyczącymi pracy nad projektem informatycznym. Zdarzają się jednak bardziej złożone scenariusze. Przypomnijmy, że firma X pracuje nad edytorem *Text*. Załóżmy, że firma wypuściła na rynek stabilną wersję produktu *Text 2010*. Od tego momentu firma równolegle wprowadza poprawki zgłaszane przez użytkowników do wersji *Text 2010* i szykuje aplikację *Text 2011 5G* na komórki. W końcu firma wypuszcza na rynek *Text 2011 5G*. Odtąd firma równocześnie wprowadza poprawki do dwóch wcześniejszych wersji produktu i zaczyna już pracować nad edytorem *Text 2012*. . .



Rys. 2

W tym scenariuszu w historii zmian repozytorium zaczynają pojawiać się odgałęzienia, w których kolejne wersje naszego produktu zaczynają żyć własnym życiem, tj. zaczynają podlegać niezależnym zmianom (rys. 2). Podobnie sytuacja wyglądałaby, gdyby dostępna była wersja on-line edytora *Text*, do której co jakiś czas dodawalibyśmy nowe funkcjonalności, choć, oczywiście, nie chcielibyśmy każdej pojedynczej zmiany w repozytorium od razu umieszczać w wersji produktu aktualnie dostępnej dla użytkowników.

Historia zmian może być jeszcze bardziej zagmatwana. Załóżmy, że w bieżącym repozytorium jest prosta wersja modułu do wstawiania obrazków w edytorze *Text*. Grupa pracowników firmy pracuje nad ulepszonym modułem do obrazków, dopuszczającym także obrazki trójwymiarowe. Programiści ci przewidują, że ich rewolucyjny moduł w pewnym momencie w pełni zastąpi obecny. Jednak dopóki ich moduł nie jest jeszcze gotowy – tj. nie działa stabilnie – nie chcieliby udostępniać swoich zmian wszystkim zespołom, żeby ich usterki nie przeszkadzały innym w pracy. Co więcej, projekt ma charakter eksperymentalny – jeśli nowy moduł z jakichś powodów nie powstanie, firma po prostu pozostanie przy starym.



Rys. 3

Tym razem mamy więc nie tylko rozgałęzienie, w którym pojawia się tzw. gałąź eksperymentalna, lecz także możliwość powrotu tej gałęzi do głównej linii rozwoju projektu (`svn merge`). Historia zmian może mieć teraz postać acyklicznego grafu skierowanego (tzw. DAG), patrz rysunek 3.

Można wskazać jeszcze inne ciekawe przypadki użycia systemów kontroli wersji. Oto mój ulubiony: załóżmy, że w firmie X co noc sprawdza się, czy kod edytora *Text* kompiluje się poprawnie na różne systemy operacyjne i środowiska uruchomieniowe (ang. *nightly build*). Robi się tak dlatego, że pełna kompilacja trwa na tyle długo, iż nie można jej wykonywać po każdej pojedynczej zmianie kodu. Poza samą kompilacją można także sprawdzać poprawność działania programu na tzw. testach jednostkowych (ang. *unit test*). Pewnej nocy testy nie przebiegły pomyślnie. W jaki sposób można łatwo stwierdzić, która konkretnie zmiana spowodowała ten błąd? Wiemy, że poprzedniej nocy wszystko działało poprawnie, ale w ciągu jednego dnia mogły pojawić się setki nowych zmian w kodzie, no i która z nich jest tą wadliwą?

Już dawno temu wymyślono efektywne rozwiązanie algorytmiczne tego problemu – wyszukiwanie binarne (`svn bisect`). Powiedzmy, że stan repozytorium po zmianie numer N jest poprawny, ale po zmianie numer $N + 100$ jest błędny. Sprawdźmy, jak wygląda repozytorium po zmianie numer $N + 50$. Jeśli jest poprawne, to wiemy, że błąd znajduje się wśród zmian $N + 51, \ldots, N + 100$, a w przeciwnym razie – wśród zmian $N + 1, \ldots, N + 50$. W obu przypadkach została nam już tylko połowa możliwości do sprawdzenia, więc kontynuujemy to postępowanie i po logarytmicznej liczbie kroków na pewno zidentyfikujemy problem.

Niestety, w pewnych sytuacjach system SVN okazuje się niewystarczający. Przykładowo, pojedynczemu programiście zdarza się pracować równolegle nad kilkoma zmianami, np. nad dwoma drobnymi usprawnieniami w module obrazków w edytorze *Text*. Innym problemem jest konieczność stałej łączności z serwerem przy wykonywaniu praktycznie każdej operacji związanej z repozytorium. Istnieją inne systemy kontroli wersji, w których podane problemy nie występują. Jednym z nich jest system Git, w którym każdy programista może tworzyć odgałęzienia na swoim komputerze lokalnym bez konieczności nawiązania połączenia sieciowego z centralnym repozytorium. Co więcej, po zakończeniu prac zmiany te mogą być dokładane pojedynczo do głównej linii rozwoju projektu bez tworzenia w niej rozgałęzień (zapobiega to nadmiernemu bałaganowi w grafie zmian repozytorium).

W systemie Git ciekawe są nie tylko funkcje wspierające w pełni rozproszony system pracy, lecz także sama historia jego powstania. Początkowym przeznaczeniem Gita było przechowywanie kodu jądra systemu operacyjnego Linux, a powstał on wtedy, gdy wcześniej używany do tego system kontroli wersji o nazwie BitKeeper przestał być darmowy dla projektów o otwartym kodzie źródłowym. Prace nad Gitem rozpoczęły się 3 kwietnia 2005 roku, a już dwa i pół miesiąca później, 16 czerwca, system był używany do przechowywania kodu jądra Linuksa. Warto odnotować, że zgodnie z wolą twórców, do dziś Git stanowi tzw. wolne oprogramowanie, co generalnie oznacza, że każdy może go za darmo używać i modyfikować praktycznie bez ograniczeń. System Git, jakkolwiek bardzo funkcjonalny, jest koncepcyjnie nieco trudniejszy do ogarnięcia od „mniej rozproszonych” systemów takich jak SVN.

W tym artykule staraliśmy się pokazać, dlaczego każda większa firma z branży komputerowej używa systemu kontroli wersji. Na koniec warto jednak zapytać, czy takie systemy mogą się przydać „zwykłemu śmiertelnikowi”, który nie jest akurat dyrektorem ani pracownikiem takiej korporacji. Otóż tak! Systemy kontroli wersji świetnie spisują się w sytuacji, gdy kilka osób wspólnie pracuje nad jednym projektem, na przykład pracą naukową, redakcją książki czy niewielkim programem komputerowym. Ale nie tylko! Nawet jedna osoba może z powodzeniem stosować systemy takie jak SVN do przechowywania ważnych danych na komputerze. W sieci dostępnych jest wiele darmowych i prostych w obsłudze serwisów udostępniających serwery na repozytoria SVN. Dobrą motywacją jest przypomnieć sobie, kiedy ostatnio usunęło się lub przypadkowo nadpisało jakiś bardzo ważny plik...

Reklama w Internecie, czyli o aukcjach Vickreya

Karol KURACH[*]

[*]doktorant, Wydział Matematyki, Informatyki i Mechaniki, Uniwersytet Warszawski

Czy zastanawiałeś się kiedykolwiek, drogi Czytelniku, na czym zarabiają największe firmy informatyczne w Dolinie Krzemowej? Google nie pobiera opłat za wyszukiwanie informacji, konto na Facebooku jest darmowe, Yahoo nie żąda pieniędzy za e-mail... Z punktu widzenia użytkownika praktycznie wszystko wydaje się bezpłatne. Dlaczego zatem powyższe firmy wyceniane są na miliardy dolarów? Kto pokrywa koszty utrzymania serwerów, zużycia prądu czy pensji pracowników?

Kluczem do odpowiedzi na postawione pytania są reklamy internetowe. Zyski z nich stanowią główne źródło dochodów każdej z powyższych firm (np. w przypadku Google w roku 2010 było to 96% całkowitego dochodu, a w przypadku Yahoo około 90%). Jest to bardzo dynamicznie rozwijająca się branża – wydatki przeznaczane na ten typ reklamy rosną co roku o miliardy dolarów. W samych tylko Stanach Zjednoczonych było to ponad 30 mld dolarów w roku 2010, z szacowanymi 50 mld dolarów na rok 2015. W artykule postaram się przybliżyć najważniejsze pojęcia i mechanizmy związane z tym zagadnieniem.

Co mają wspólnego reklamy z aukcjami? Wpiszmy w wyszukiwarce internetowej jakieś zapytanie. Otrzymamy listę wyników i zapewne również pewien zbiór odnośników wyróżnionych jako *sponsorowane*. Te odnośniki to przykład tzw. *marketingu bezpośredniego* (ang. *direct marketing*). Są one nastawione na natychmiastową akcję ze strony użytkownika – kliknięcie i kupienie czegoś. To, jakie reklamy zostaną wyświetlone, zależy od treści zapytania. Takie podejście ma za zadanie zmaksymalizować skuteczność reklamy oraz zmniejszyć ilość spamu docierającego do odbiorców.

Wyświetlane reklamy zależą również od innych czynników, jak np. adres IP, kraj, z którego pochodzi zapytanie, pora dnia itp. Dla uproszczenia pominiemy te czynniki w naszych rozważaniach.