20 LAT OBECNOŚCI NA RYNKU I SETKI TYSIĘCY SPRZEDANYCH EGZEMPLARZY NA CAŁYM ŚWIECIE!

WYDANIE JUBILEUSZOWE

LEGENDARNY OSOBOMIESIĄC

OPOWIEŚCI O INŻYNIERII OPROGRAMOWANIA

WYDANIE II

FREDERICK P. BROOKS JR

Helion

Tytuł oryginału: The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition (2nd Edition)

Tłumaczenie: Wojciech Moch

ISBN: 978-83-283-5079-3

Authorized translation from the English language edition, entitled:
THE MYTHICAL MAN-MONTHS, ANNIVERSARY EDITION, 2nd Edition;
ISBN 0201835959; by Frederick Phillips Brooks; published by Pearson Education, Inc, publishing as Addison Wesley.

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

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

Printed in Poland.

Spis treści

2.
Legendarny osobomiesiąc

Dobra kuchnia wymaga czasu. Jeżeli zechcesz poczekać,
pozwoli to lepiej Cię obsłużyć i zadowolić.

MENU RESTAURACJI ANTOINE W NOWYM ORLEANIE

Więcej projektów upadło z powodu braku czasu w kalendarzu niż z wszystkich innych powodów razem wziętych. Dlaczego właśnie ten powód jest tak częsty?

Po pierwsze, nasze techniki szacowania są wyjątkowo słabej jakości. Przede wszystkim, wyrażają one ciche (i niczym nieuzasadnione) założenie, że wszystko uda się zrealizować bez problemów.

Po drugie, nasze techniki szacowania chętnie mieszają pojęcia pracy i postępu, ukrywając w ten sposób niesłuszne założenie, że pracownik i miesiąc są pojęciami wymiennymi.

Po trzecie, ponieważ nie jesteśmy pewni poprawności naszych szacunków, kadrze zarządzającej często brakuje tego wspaniałego uporu szefa restauracji Antoine.

Po czwarte, postępy projektu są zazwyczaj słabo monitorowane. Techniki, które dobrze się sprawdziły w innych dziedzinach inżynierskich i dlatego są w nich stosowane rutynowo, w przypadku inżynierii oprogramowania uznawane są za radykalne innowacje.

Po piąte, w przypadku zauważenia opóźnień w harmonogramie całkowicie naturalną (i tradycyjną) reakcją jest dodanie do zespołu nowych programistów. Powoduje to tylko ciągłe pogarszanie się sytuacji i przypomina próby gaszenia ognia benzyną. Większy ogień wymaga użycia większej ilości benzyny, a powtarzający się w ten sposób cykl prowadzi prosto do katastrofy.

Monitorowanie postępów będzie tematem jednego z rozdziałów, teraz przyjrzyjmy się innym aspektom tego problemu.

OPTYMIZM

Wszyscy programiści są optymistami. Być może nowoczesna forma magii przyciąga do siebie osoby, które wierzą w szczęśliwe zakończenia i Wróżki Zębuszki. Być może setki powodów do frustracji odstrasza wszystkich tych, którzy całkowicie koncentrują się na osiągnięciu celu. Być może wynika to z tego, że komputery są jeszcze całkiem młode, programiści są jeszcze młodsi, a młodzież zawsze promienieje optymizmem. Niezależnie od tego, jak działa ten mechanizm selekcji, w jego wyniku pojawiają się takie stwierdzenia jak: „Tym razem na pewno zadziała” albo „Właśnie znalazłem ostatni błąd”.

A zatem pierwsze fałszywe założenie stanowiące podstawę harmonogramów tworzonych systemów programistycznych brzmi: *Wszystko pójdzie jak należy*, czyli *każde zadanie zajmie tylko tyle czasu ile „powinno" zająć*.

Taka powszechność optymizmu wśród programistów zasługuje na coś więcej niż tylko pobieżną analizę. Dorothy Sayers w swojej doskonałej książce *The Mind of the Maker* dzieli działania kreatywne na trzy etapy: pomysłu, implementacji i interakcji. Oznacza to, że książka, komputer lub program powstaje najpierw jako konstrukcja idealna, stworzona poza czasem i przestrzenią, ale w umyśle jej autora kompletna i doskonała. Następnie jest realizowana w czasie i przestrzeni za pomocą pióra, atramentu i papieru albo przewodów, krzemu i żelaza. Taka kreacja zostaje ukończona, gdy ktoś przeczyta książkę, skorzysta z komputera albo uruchomi program, wchodząc tym samym w interakcję z umysłem twórcy.

Taki opis, którego Sayers używa nie tylko do objaśnienia ludzkiej kreatywności, ale też chrześcijańskiej doktryny Trójcy Świętej, przyda się w naszym aktualnym zadaniu. W przypadku ludzkich twórców różnych rzeczy niedoskonałości i niespójności naszych idei stają się widoczne podczas ich implementowania. Oznacza to, że pisanie, eksperymentowanie, „ćwiczenie" są bardzo ważnymi elementami dla teoretyka.

W wielu działaniach kreatywnych medium ich realizacji jest trudne do opanowania. Drewno się łamie, farby brudzą, a układy elektryczne się przepalają. Takie fizyczne ograniczenia stosowanego medium sprawiają, że nie wszystkie idee można za ich pomocą wyrazić, a na dodatek tworzą nieoczekiwane problemy podczas implementowania.

Sama implementacja też wymaga czasu i pracy, co wynika zarówno z wykorzystania fizycznego medium, jak i z niedopasowania samych idei. Zazwyczaj winę za większość problemów z implementacją zrzucamy na fizyczne medium. Po prostu medium nie jest „nasze" w ten sposób, w jaki „nasze" są idee, a duma zawsze wpływa na nasze oceny.

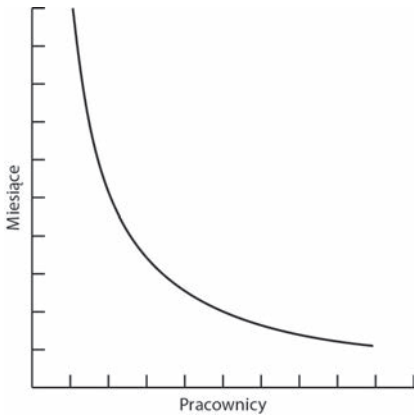
Programowanie komputerów to jednak praca z niezwykle plastycznym medium. Programiści tworzą wyłącznie za pomocą własnych myśli, stosując same idee oraz ich niezwykle elastyczne reprezentacje. Z powodu tak wielkiej plastyczności używanego medium oczekujemy niemalże całkowitego braku kłopotów przy implementowaniu. To właśnie z tego wynika nasz nieodłączny optymizm. Niestety nasze idee nie są idealne, a zatem powstają błędy podczas implementowania. To z kolei oznacza, że nasz optymizm jest nieuzasadniony.

W przypadku pojedynczego zadania założenie, że uda się je zrealizować bez problemów, ma probabilistyczny wpływ na cały harmonogram. Rzeczywiście wszystko może pójść zgodnie z planem, ponieważ prawdopodobieństwo opóźnienia w danym zadaniu ma pewien rozkład, w którym można wyznaczyć prawdopodobieństwo wystąpienia „braku opóźnienia". Niestety duże projekty programistyczne składają się z wielkiej liczby zadań, z których część musi być wykonywana sekwencyjnie. W takich warunkach prawdopodobieństwo, że żadne z nich nie będzie miało opóźnień, jest bardzo niskie.

OSOBOMIESIĄC

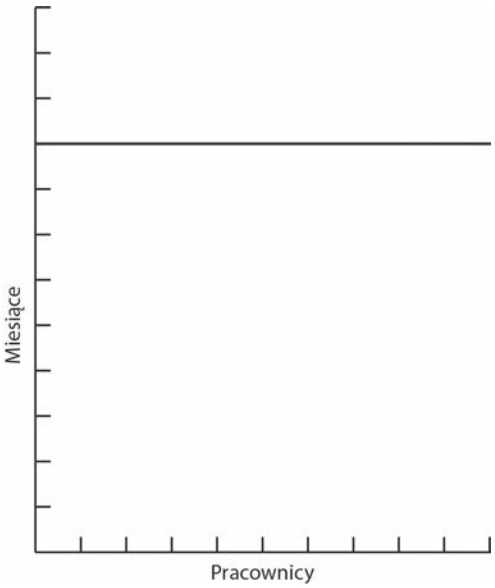
Kolejnym groźnym sposobem myślenia jest stosowanie samej jednostki wykonywanej pracy podczas szacowania oraz przygotowywania harmonogramu: osobomiesiąca. Rzeczywiście koszt tworzenia oprogramowania można wyrazić jako iloczyn liczby pracowników i liczby przepracowanych miesięcy. Nie dotyczy to jednak postępów. *Oznacza to, że osobomiesiąc stosowany jako jednostka miary wielkości poszczególnych zadań jest bardzo niebezpiecznym i podstępnym mitem*. Sugeruje, że pracownicy i miesiące są wartościami wymiennymi.

Pracownicy i miesiące stają się wartościami wymiennymi jedynie, gdy dane zadanie można podzielić pomiędzy wielu pracowników, *nie wymagając od nich żadnej komunikacji* (rysunek 2.1). Takie podejście sprawdza się przy młóceniu pszenicy lub zbieraniu bawełny, ale w przypadku programowania systemów zupełnie nie ma zastosowania.



Rysunek 2.1. Czas a liczba pracowników — zadanie doskonale podzielne

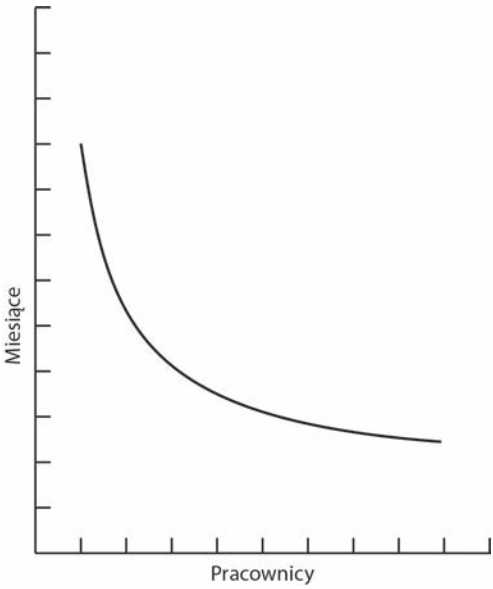
Jeżeli zadania nie da się podzielić, na przykład z powodu ograniczeń wynikających z sekwencji prac, to przypisanie do niego dodatkowych pracowników nie będzie miało wpływu na harmonogram (rysunek 2.2). Urodzenie dziecka zajmuje zawsze dziewięć miesięcy, niezależnie od tego, ile kobiet zostanie przydzielonych do tego zadania. Wiele zadań programistycznych ma taki właśnie charakter, co wynika z sekwencyjnej natury debugowania.



Rysunek 2.2. Czas a liczba pracowników — zadanie niepodzielne

W przypadku zadań, które można podzielić, ale wymagają one komunikacji pomiędzy poszczególnymi zadaniami cząstkowymi, do ilości pracy koniecznej do wykonania trzeba doliczyć jeszcze pracę związaną z samą komunikacją. Oznacza to, że najlepszy efekt, jaki można uzyskać, będzie zawsze gorszy od prostej zamiany pracowników na miesiące (rysunek 2.3).

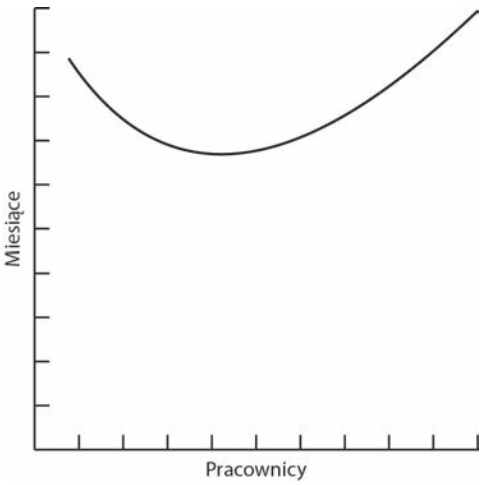
Dodatkowe obciążenia związane z komunikacją można podzielić na dwie kategorie: naukę oraz wymianę informacji. Każdy z pracowników musi nauczyć się korzystać ze stosowanej technologii, poznać cele całego przedsięwzięcia, ogólną strategię oraz plan pracy. Takiej nauki nie da się podzielić na mniejsze części, a zatem ta kategoria obciążenia powoduje wzrost nakładu pracy rosnący liniowo wraz ze wzrostem liczby pracowników[I].



Rysunek 2.3. Czas a liczba pracowników — podzielne zadanie wymagające komunikacji

Druga kategoria, czyli wymiana informacji, jest znacznie gorsza. Jeżeli każda część zadania musi być koordynowana z pozostałymi częściami, to wymagany nakład pracy rośnie zgodnie ze wzorem: $n(n-1)/2$. Trzech pracowników wymaga trzykrotnie intensywniejszej komunikacji dwustronnej niż dwóch. Czterech wymaga już sześciokrotnie więcej komunikacji dwustronnej. Co więcej, jeżeli konieczne są konferencje między trzema, czterema lub większą liczbą pracowników w celu wspólnego rozwiązywania problemów, to sprawy komplikują się jeszcze bardziej. Dodatkowe nakłady związane z komunikacją mogą całkowicie zanegować zyski wynikające z podziału zadania, przez co znajdziemy się w sytuacji przedstawionej na rysunku 2.4.

Tworzenie systemów programowych to jednoznacznie zadanie systemowe i jako takie wymaga obsłużenia wielu powiązań. Oznacza to, że nakłady na komunikację są bardzo wysokie i potrafią szybko zdominować czas wykonania poszczególnych zadań, nawet po ich podzieleniu. W takiej sytuacji dodanie nowych pracowników zamiast skracać, będzie wydłużać harmonogram prac.



Rysunek 2.4. Czas a liczba pracowników — zadanie ze złożonymi zależnościami wzajemnymi

TESTY SYSTEMOWE

Debugowanie komponentów i testy systemowe to części harmonogramu, które są najbardziej ograniczane przez konieczność utrzymania sekwencji prac. Co więcej, czas wymagany do ich wykonania uzależniony jest od liczby i skomplikowania wykrytych błędów. Teoretycznie takich błędów nie powinno być wcale. Z powodu naszego optymizmu zazwyczaj oczekujemy, że ogólnie błędów będzie mniej, niż jest ich w rzeczywistości. W związku z tym testowanie jest najgorzej zaplanowanym etapem tworzenia oprogramowania.

Przez wiele lat, z sukcesem, stosowałem poniższą ogólną regułę planowania harmonogramu prac dla zadania programowego:

1/3 to planowanie
1/6 to tworzenie kodu
1/4 to testy komponentów i wczesne testy systemowe
1/4 to testy systemowe, gdy gotowe są już wszystkie komponenty

W kilku punktach różni się to od konwencjonalnego przygotowywania harmonogramu:

- Część przeznaczona na planowanie jest większa niż normalnie stosowana. Mimo to i tak czasu z ledwością wystarcza na przygotowanie szczegółowej i porządnej specyfikacji.

Jest też całkowicie niewystarczająca, jeżeli chcielibyśmy przeprowadzić badania i ćwiczenia w nowych technikach.

- *Połowa* harmonogramu jest związana z debugowaniem gotowego kodu i jest to wartość zdecydowanie większa niż normalnie stosowana.
- Część, która jest najłatwiejsza do oszacowania, czyli tworzenie kodu, otrzymała zaledwie jedną szóstą całości czasu.

Kontrolując konwencjonalnie planowane projekty, zauważyłem, że zaledwie kilka z nich przeznaczało połowę harmonogramu na testowanie, ale niemal we wszystkich, w rzeczywistości, połowę czasu zajmowało testowanie. W wielu z tych projektów prace postępowały zgodnie z harmonogramem do momentu rozpoczęcia testów systemowych[II].

Błąd polegający na nieprzydzieleniu odpowiedniej ilości czasu na testy, w szczególności na testy systemowe, ma katastrofalne skutki. Ponieważ takie opóźnienie pojawia się pod koniec całego harmonogramu, niemal do samego końca nikt nie ma pojęcia, że projekt ma jakiekolwiek kłopoty. Złe wieści przekazywane tak późno i bez ostrzeżenia są niezwykle deprymujące zarówno dla klientów, jak i dla kadry zarządzającej.

Co więcej, opóźnienia pojawiające się na tym etapie zazwyczaj mają poważne reperkusje zarówno finansowe, jak i psychologiczne. Projekt ma pełną obsadę, a zatem koszt każdego dnia prac jest maksymalny. Co ważniejsze, tworzone oprogramowanie powinno stanowić wsparcie dla innych działów firmy (dostawy komputerów, prace w nowych oddziałach i inne) i takie pochodne koszty opóźnienia mogą być bardzo wysokie, ponieważ inni spodziewają się teraz otrzymać gotowe oprogramowanie.

I rzeczywiście, takie pochodne koszty opóźnienia mogą szybko przewyższyć wszystkie pozostałe. Z tego powodu bardzo ważne jest, żeby w pierwotnym harmonogramie przeznaczyć odpowiednio dużo czasu na testy systemowe.

SZACOWANIE TCHÓRZLIWE

Zauważ, że w przypadku programisty, jak w przypadku kucharza, naciski ze strony przełożonego mogą wpływać na planowany czas wykonania zadania, ale nie będą miały wpływu na czas faktycznego ukończenia. Przygotowywanie omletu, który zgodnie z planem ma być gotowy w dwie minuty,

może postępować zgodnie z harmonogramem. Jeżeli jednak po dwóch minutach nie będzie gotowy, to klient ma dwie możliwości: albo poczekać, albo zjeść surowe jajka. Klienci czekający na swoje oprogramowanie mają podobny wybór.

Kucharz może jeszcze podnieść temperaturę. W efekcie zazwyczaj powstaje omlet całkowicie niejadalny — przypalony z dołu, a surowy z góry.

Nie sądzę, by menedżerowie oprogramowania mieli mniej odwagi i niezłomności od kucharzy, podobnie jak inni menedżerowie prowadzący projekty inżynierskie. Niestety źle przygotowane harmonogramy, które dopasowywane są do daty wyznaczonej przez zleceniodawcę, w naszej branży pojawiają się znacznie częściej niż w innych działach inżynierskich. Niezwykle trudno jest przygotować poważną obronę swojego planu, ryzykując przy tym utratę pracy, szczególnie gdy taki plan nie został zbudowany z wykorzystaniem żadnej solidnej metody, do dyspozycji mieliśmy tylko niewiele danych, a poparty jest jedynie ogólnymi odczuciami menedżerów.

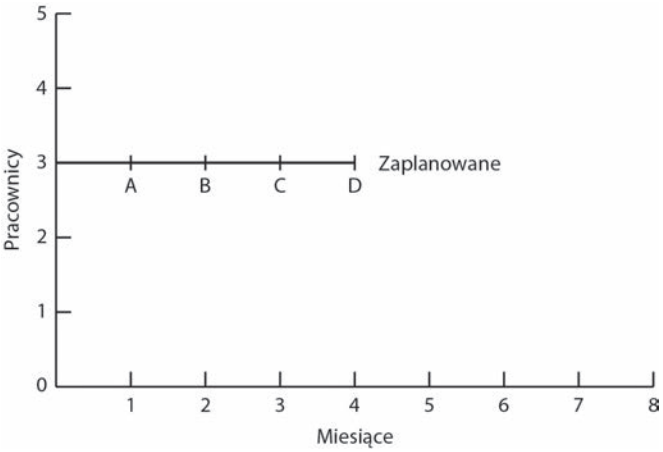
Oczywiście potrzeba nam tutaj dwóch rozwiązań. Musimy przygotować i opublikować wskaźniki produktywności, wskaźniki wykrywalności błędów, reguły szacowania i tym podobne. Cała nasza branża może tylko zyskać, jeżeli takie informacje staną się publicznie dostępne.

Do czasu, aż szacowanie będzie mogło być realizowane na tak solidnej podstawie, menedżerowie muszą usztywnić swoje karki i bronić przygotowanych przez siebie szacunków, mając pewność, że ich niepoparte niczym odczucia i tak są znacznie lepsze od szacowania życzeniowego.

POWTARZALNE KATASTROFY HARMONOGRAMÓW

Co należy zrobić, jeżeli w ważnym projekcie programistycznym nie da się dotrzymać harmonogramu? Oczywiście zatrudnić więcej ludzi! Jak widać na rysunkach od 2.1 do 2.4, takie podstępowanie nie zawsze jest prawdziwą pomocą.

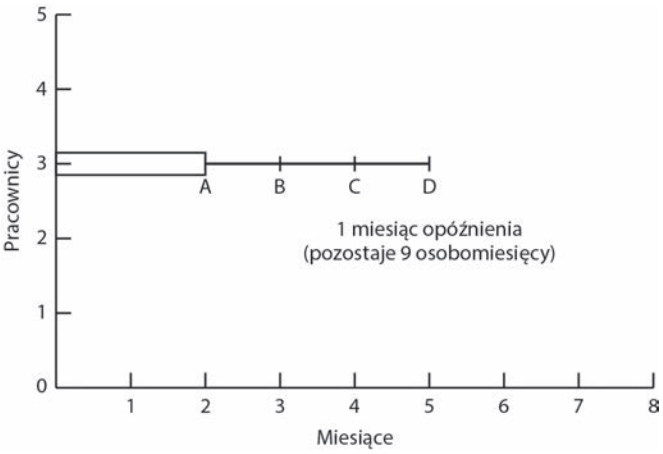
Przyjrzyjmy się pewnemu przykładowi[III]. Załóżmy, że do zadania szacowanego na dwanaście osobomiesięcy przydzielono trzech pracowników na cztery miesiące. Zdefiniowano też cztery mierzalne kamienie milowe oznaczone A, B, C i D, które zostały zaplanowane na koniec każdego miesiąca (rysunek 2.5).



Rysunek 2.5.

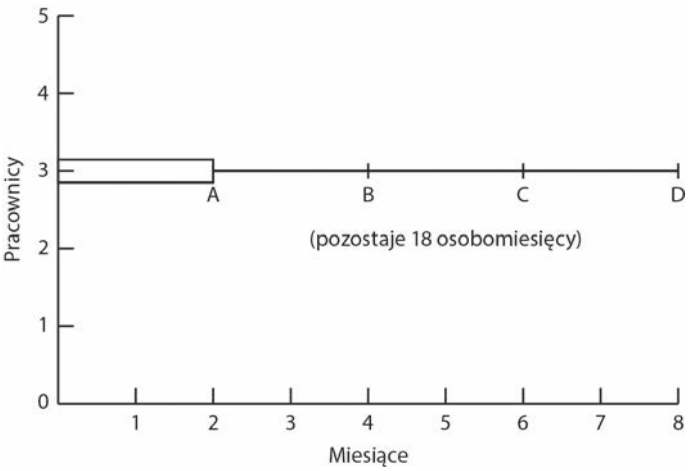
Załóżmy teraz, że pierwszy z kamieni milowych udało się osiągnąć dopiero po upływie dwóch miesięcy (rysunek 2.6). Jakie decyzje może podjąć w takiej sytuacji menedżer?

1. Założyć, że zadanie musi zostać wykonane w zaplanowanym czasie. Zakłada się zatem, że tylko pierwsza część całego zadania została źle oszacowana, czyli rysunek 2.6 całkiem dobrze przedstawia aktualną sytuację. Wówczas pozostaje jeszcze dziewięć osobomiesięcy prac i tylko dwa miesiące na ich wykonanie, czyli po cztery i pół osoby na każdy z miesięcy. Oznacza to, że do trzyosobowego zespołu trzeba dodać jeszcze dwóch pracowników.



Rysunek 2.6.

2. Założyć, że zadanie musi zostać wykonane w zaplanowanym czasie. Zakłada się też, że całość planu została oszacowana za nisko, a zatem aktualną sytuację lepiej opisuje rysunek 2.7. W tej sytuacji pozostaje do wykonania jeszcze osiemnaście osobomiesięcy prac, czyli potrzeba nam dziewięciu pracowników. Oznacza to, że do trzyosobowego zespołu trzeba dodać jeszcze sześciu pracowników.

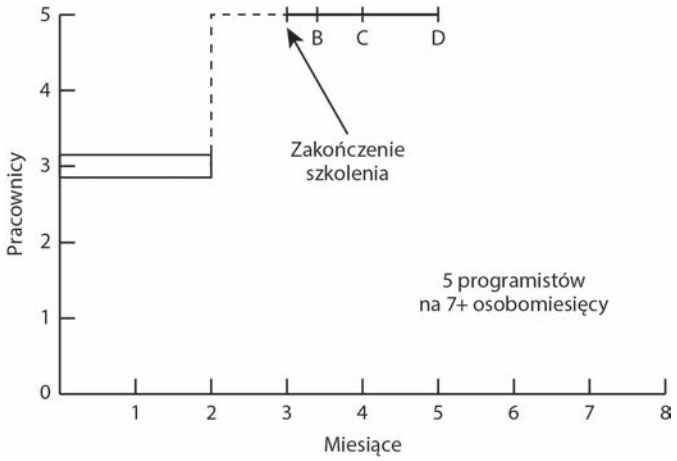


Rysunek 2.7.

3. Zmienić cały harmonogram. Bardzo podoba mi się rada P. Fagga — doświadczonego inżyniera, który mówił: „Nie akceptuj małych poślizgów". Oznacza to, że w nowym harmonogramie trzeba zaplanować dość czasu, aby mieć pewność, że prace zostaną przeprowadzone rzetelnie i dokładnie, tak żeby nie trzeba było dokonywać kolejnych korekt.
4. Zmniejszyć wielkość zadania. W praktyce zadania i tak są redukowane, gdy tylko zauważone zostaną opóźnienia. Jeżeli pochodne koszty opóźnień są bardzo wysokie, to jest to jedyne akceptowalne rozwiązanie. Menedżer ma zatem do wyboru formalne zmniejszenie zadania, korektę harmonogramu albo ciche obserwowanie, jak zadanie zostaje ograniczone przez pospieszne projektowanie i niepełne testy.

W pierwszych dwóch przypadkach upieranie się, że niezmienione zadanie może zostać zrealizowane w cztery miesiące, to prosta droga do katastrofy. Przyjrzyjmy się efektom regeneracyjnym, na przykład dla pierwszego

rozwiązania (rysunek 2.8). Dwóch nowych pracowników, niezależnie od tego, jak będą kompetentni i jak szybko zostaną pozyskani dla projektu, będzie wymagało wdrożenia w tematykę zadania przez jednego z doświadczonych programistów. Jeżeli zajmie to jeden miesiąc, to znaczy, że *trzy osobomiesiące zostaną zużyte na prace niezwiązane z pierwotnym harmonogramem*. Co więcej, zadanie, które początkowo podzielono na trzy części, teraz musi zostać podzielone na pięć. A to oznacza, że część wykonanych już prac zostanie stracona, natomiast wydłużona musi zostać faza testów systemowych. A zatem pod koniec trzeciego miesiąca do przepracowania zostanie jeszcze siedem osobomiesięcy, podczas gdy dostępnych będziemy mieli jeden miesiąc i pięciu przeszkolonych pracowników. Z rysunku 2.8 wynika, że projekt będzie miał takie samo opóźnienie, jakie miałby bez dodawania do zespołu dodatkowych pracowników (rysunek 2.6).



Rysunek 2.8.

Aby mieć nadzieję na zakończenie prac w cztery miesiące, biorąc pod uwagę sam czas szkolenia i nie uwzględniając konieczności ponownego podziału zadań oraz dodatkowych testów systemowych, pod koniec drugiego miesiąca powinniśmy dodać do zespołu czterech, a nie dwóch pracowników. Chcąc też pokryć nakłady związane z ponownym podziałem i dodatkowymi testami, należałoby rozważyć dodanie do zespołu kolejnych osób. W takiej sytuacji powstanie jednak zespół składający się przynajmniej z siedmiu osób, a nie tylko z trzech. W związku z tym organizacja zespołu oraz podział zadań stanowią zupełnie inny rodzaj problemu.

Proszę zauważyć, że pod koniec trzeciego miesiąca sprawy wyglądają bardzo źle. Pierwszy etap prac nie został ukończony mimo wzmożonych wysiłków menedżerskich. Bardzo mocna jest pokusa powtórzenia całego cyklu przez dodanie do zespołu kolejnych pracowników. To czyste szaleństwo.

To wszystko jest prawdą przy założeniu, że tylko pierwszy etap prac został źle zaplanowany. Jeżeli po jego ukończeniu przyjmiemy rozsądne założenie, że całość harmonogramu była zbyt optymistyczna, tak jak na rysunku 2.7, to potrzebnych będzie sześciu dodatkowych pracowników tylko po to, aby zrealizować pierwotne zadanie. Obliczenie czasu potrzebnego na szkolenie, ponowny podział prac i testy systemowe pozostawiam jako ćwiczenie dla czytelnika. Bez wątpienia taka samonapędzająca się katastrofa nie pozostanie bez wpływu na jakość produktu, który zostanie dostarczony później, niż to by nastąpiło w przypadku korekty harmonogramu i pozostaniu przy pierwotnym, trzyosobowym zespole.

Nadmiernie upraszczając, możemy tu zdefiniować prawo Brooka:

Dodawanie pracowników do opóźnionego projektu tylko zwiększa opóźnienie.

W ten sposób chcę zdemitologizować osobomiesiąc. Liczba miesięcy trwania projektu zależy od wielu ograniczeń sekwencyjnych. Maksymalna liczba pracowników uzależniona jest od liczby niezależnych od siebie zadań podrzędnych. Na podstawie tych dwóch wskaźników można przygotować harmonogram wymagający mniejszej liczby pracowników, ale większej liczby miesięcy. (Jedynym ryzykiem jest przedawnienie się produktu). Nie da się jednak uzyskać realizowalnego harmonogramu wymagającego większej liczby osób, a mniejszej liczby miesięcy. Więcej projektów programistycznych legło w gruzach z powodu braku czasu niż z powodu wszystkich innych przyczyn razem wziętych.

Skorowidz

N

O

P

R

S

Ś

T

U

W

Z

NIE MA DRUGIEJ TAK WPŁYWOWEJ I PONADCZASOWEJ KSIĄŻKI DLA TWÓRCÓW OPROGRAMOWANIA!

Zarządzanie procesem tworzenia oprogramowania bywa doświadczeniem bardzo pouczającym, a jednocześnie niezwykle frustrującym. Z jednej strony takie projekty są podobne do innych dużych przedsięwzięć, z drugiej — wymagają od kierownictwa sporo specjalistycznej wiedzy i specyficznego podejścia do zagadnień programistycznych. Oczywiście, wiedza na ten temat stale rośnie, pojawiają się też nowe koncepcje kierowania dużymi projektami. Jeśli brakuje Ci literatury, która potraktowałaby to zagadnienie kompleksowo, katalogowałaby poszczególne propozycje i opisywałaby je w przystępny i przydatny sposób — sięgnij po ten tytuł!

Książka *Legendarny osobomiesiąc* zyskała już miano kultowej; jest niezmiennie aktualna i wciąż inspiruje programistów na całym świecie. Składa się z kilkunastu esejów, które zawierają informacje i inspiracje bezcenne dla każdego menedżera i programisty. Przy dużych projektach konieczne jest zachowanie ich spójności koncepcyjnej, co w przypadku dużych zadań stanowi warunek dość trudny do spełnienia, dlatego wiele obiecujących przedsięwzięć zakończyło się porażką. Trzeba też zdawać sobie sprawę, że złożone zadanie oznacza dla zespołu dobre i złe chwile. Autor w niezwykle interesujący i praktyczny sposób pokazuje, jak czerpać siły z chwil radości i skutecznie radzić sobie z problemami, aby zakończyć z sukcesem nawet najbardziej złożony projekt programistyczny.

W TEJ KSIĄŻCE ZNALAZŁY SIĘ KONCEPCJE OBEJMUJĄCE MIĘDZY INNYMI:

— podejmowanie decyzji
— zadania architekta
— skalowanie i zachowanie spójności projektu
— sporządzanie dokumentacji i specyfikacji
— zarządzanie komunikacją w wielkich projektach

A TAKŻE: retrospektywna analiza tych i innych koncepcji sprzed 20 lat

Dr Frederick P. Brooks jr — wykładowca na Uniwersytecie Karoliny Północnej w Chapel Hill. Był architektem komputerów IBM Stretch i Harvest. Jest świetnie znany jako kierownik zespołu rozwijającego komputery IBM System/360 — za tę pracę w 1985 roku dostał National Medal of Technology. Z kolei w 1999 roku otrzymał Nagrodę Turinga. Na uniwersytecie w Chapel Hill założył wydział nauk komputerowych, którym kierował do 1984 roku. Był też aktywnym członkiem National Science Board i Defence Science Board. W ostatnich latach zajmował się badaniami oraz nauczaniem architektury komputerów, grafiki molekularnej i środowisk wirtualnych.

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 978-83-283-5079-3
9 788328 350793
INFORMATYKA W NAJLEPSZYM WYDANIU
Cena: 67,00 zł