O'REILLY®

Przewodnik

Jak stać się lepszym programistą

PODRĘCZNIK PRAWDZIWEGO PROGRAMISTY!

Helion

Pete Goodliffe

Tytuł oryginału: Becoming a Better Programmer

Tłumaczenie: Tomasz Walczak

ISBN: 978-83-283-0239-6

Wydawnictwo HELION
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/jakpro
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 31.

Pracuj mądrzej, a nie ciężej

Bitwy wygrywa się zabijaniem i manewrowaniem. Im lepszy generał, tym większą wagę przykłada do manewrowania, a mniejszą do zabijania.

— Winston Churchill

Pozwól, że opowiem pewną historię. Jest ona w pełni prawdziwa. Kolega pracował nad kodem interfejsu użytkownika i musiał wyświetlić atrakcyjne, zaokrąglone strzałki. Miał trudności z programowym wygenerowaniem ich za pomocą podstawowych mechanizmów rysowania, dlatego zasugerowałem, aby po prostu nałożył grafikę na wyświetlany ekran. Implementacja takiego rozwiązania jest znacznie łatwiejsza.

Tak więc kolega zabrał się do pracy. Uruchomił Photoshopa i zaczął majstrować, i poprawiać. Potem jeszcze trochę pokombinował. Photoshop to Rolls-Royce programów graficznych, ale nie da się w nim szybko i łatwo narysować atrakcyjnej zaokrąglonej strzałki. Doświadczony grafik prawdopodobnie przygotowałby taką strzałkę w dwie minuty. Jednak mój znajomy po prawie godzinie rysowania, wycinania, łączenia i przesuwania nadal nie miał dobrej zaokrąglonej strzałki.

Powiedział mi o tym sfrustrowany, gdy szedł zrobić sobie szklankę herbaty.

Gdy wrócił z herbatą, na jego komputerze czekała nowiutka, błyszcząca grafika zaokrąglonej strzałki gotowa do użycia.

— Jak udało ci się zrobić to tak szybko? — zapytał.

— Użyłem właściwego narzędzia — odpowiedziałem, robiąc unik przed latającą szklanką z herbatą.

Photoshop *powinien* być odpowiednim narzędziem. To w tym programie wykonuje się większość zadań z zakresu projektowania grafiki. Wiedziałem jednak, że Open Office udostępnia wygodne narzędzie do generowania konfigurowalnych zaokrąglonych strzałek. Utworzyłem taką strzałkę w 10 sekund i przesłałem koledze jej zrzut. Nie było to eleganckie rozwiązanie, ale zadziałało.

Jaki z tego morał?

Zawsze istnieje ryzyko, że zanadto przywiążesz się do jednego narzędzia lub sposobu rozwiązywania problemów. Niezwykle łatwo jest zmarnować godziny pracy, brnąc w ślepą uliczkę, podczas gdy istnieje prostsza, bardziej bezpośrednia droga do celu.

Jak można się przed tym ustrzec?

Wybieraj pola bitwy

Aby być produktywnym programistą, musisz nauczyć się pracować *mądrzej*, a nie *ciężej*. Jedną z najważniejszych cech doświadczonych programistów jest nie tyle wiedza techniczna, co umiejętność rozwiązywania problemów i wybierania pól bitew.

Dobrzy programiści szybko wykonują zadania. Przy tym *nie* odstawiają fuszerki, co zdarza się strzelającym z biodra kowbojom świata programowania. Dobry programista pracuje mądrze. Nie musi wynikać to z tego, że jest bardziej inteligentny — po prostu wie, jak *dobrze* rozwiązywać problemy. Posiada bogate doświadczenie, z którego może czerpać, aby wybrać właściwe podejście. Dostrzega też niestandardowe rozwiązania — potrafi zastosować nietypową technikę, która pozwala wykonać zadanie mniejszym nakładem pracy. Wie, jak znaleźć drogę wokół przeszkód. Potrafi wykorzystać wiedzę do zdecydowania, jak najlepiej zainwestować swój wysiłek.

Strategie walki

Oto wybrane proste techniki, które pomogą Ci mądrzej pracować.

Rozsądnie wykorzystuj gotowy kod

Nie pisz długich fragmentów kodu samodzielnie, jeśli możesz wykorzystać istniejącą bibliotekę lub zastosować kod z innego źródła.

Nawet jeśli musisz zapłacić za bibliotekę, często i tak jest to bardziej ekonomiczne rozwiązanie niż samodzielne pisanie kodu. I testowanie go. I debugowanie.

> **KI ➢** Korzystaj z istniejącego kodu, zamiast pisać go od podstaw. Poświęć swój czas na ważniejsze rzeczy.

Nie poddawaj się niechęci do kodu, który *nie został wymyślony u nas*. Większość osób uważa, że potrafi opracować znacznie lepsze rozwiązanie lub utworzyć lepszą wersję danej aplikacji. Czy *naprawdę* tak jest? Nawet jeśli kod nie jest zaprojektowany dokładnie tak, jak byś tego chciał, użyj go. Nie musisz pisać go od nowa, jeśli działa. Jeżeli chcesz zintegrować go z systemem, napisz fasadę łączącą system z gotowym kodem.

Przekaż problem komuś innemu

Nie próbuj samodzielnie wymyślać rozwiązania zadania, jeśli ktoś inny wie już, jak to zrobić. Możliwe, że lubisz chwalić się dokonaniami. Możliwe, że chcesz nauczyć się czegoś nowego. Jeśli jednak ktoś może Ci pomóc lub znacznie szybciej wykonać pracę, często lepiej przekazać zadanie tej osobie.

Rób tylko to, co konieczne

Zastanów się nad tymi bluźnierczymi pytaniami: Czy *musisz* refaktoryzować kod? Czy *musisz* przeprowadzać testy jednostkowe?

Jestem gorącym zwolennikiem obu tych technik, jednak czasem ich stosowanie nie jest właściwe lub okazuje się złą inwestycją czasu. Tak, to prawda, że refaktoryzacja i testy jednostkowe dają dużo korzyści, dlatego nigdy nie należy z nich bezzasadnie rezygnować. Jeśli jednak przygotowujesz prosty prototyp lub badasz możliwy projekt funkcjonalny obejmujący kod, który i tak usuniesz, lepiej jest odłożyć wymienione wcześniej uświęcone praktyki na później.

Jeśli (co godne pochwały) poświęcasz czas na tworzenie testów jednostkowych, dobrze przemyśl, *które* testy warto napisać. Sztywne podejście „testujemy każdą metodę" jest nierozsądne. Często okazuje się, że pokrycie kodu testami jest wyższe, niż oczekiwano. Nie musisz na przykład testować każdego gettera i settera w interfejsie API[1]. Skoncentruj się w testach na przypadkach użycia, a nie na metodach. Ponadto zwróć uwagę zwłaszcza na te miejsca, w których spodziewasz się problemów.

Wybieraj pola bitew dla testów.

Zbuduj szybki prototyp

Jeśli masz do wyboru kilka możliwości projektowych i nie wiesz, na którą z nich się zdecydować, nie marnuj wielu godzin na zastanawianie się, które podejście będzie najlepsze. Szybki prototyp (ang. *spike*; jest to prototyp przeznaczony do wyrzucenia) pozwoli Ci błyskawicznie uzyskać przydatne odpowiedzi.

Aby to podejście dobrze zadziałało, określ, ile czasu chcesz poświęcić na prototyp (możesz zastosować technikę Pomodoro — *http://pomodorotechnique.com/*). Zakończ pracę, gdy ten czas upłynie. Jeśli chcesz dokładnie zastosować technikę Pomodoro, kup sobie trudny do zignorowania czasomierz, który zmusi Cię do zaprzestania pracy.

Korzystaj z narzędzi, które pomogą Ci szybko prześledzić ukończoną pracę (na przykład skutecznego systemu kontroli wersji).

Ustalaj priorytety

Przypisuj priorytety zadaniom. Zaczynaj od najważniejszych rzeczy.

> **KI ➢** Skoncentruj się najpierw na najważniejszych sprawach. Co jest najpilniejsze lub najbardziej wartościowe?

Potraktuj to bardzo poważnie. Nie zajmuj się nieistotnymi drobiazgami — niezwykle łatwo tracić na nie czas. Zwłaszcza gdy jedno proste zadanie zależy od wykonania innej prostej pracy. Ta wymaga wykonania innego łatwego zadanie, które z kolei… Po dwóch godzinach wrócisz na chwilę do rzeczywistości i będziesz się zastanawiał, dlaczego właśnie rekonfigurujesz serwer poczty elektronicznej, skoro chciałeś zmodyfikować metodę w klasie kontenerowej. W żargonie programistów taką sytuację nazywa się czasem *goleniem jaka* (*http://catb.org/jargon/html/Y/yak-shaving.html*).

Zauważ, że wiele drobnych zadań, którymi się zajmujesz, nie ma większego znaczenia. Chodzi na przykład o sprawdzanie e-maili, wypełnianie papierów, rozmowy telefoniczne — ogólnie

[1] Inny problem dotyczy tego, czy w ogóle *powinieneś* udostępniać gettery i settery w interfejsie API.

sprawy administracyjne. Zamiast wykonywać te czynności w trakcie dnia, kiedy przeszkadzają w ważnych zadaniach i rozpraszają, zajmij się nimi wszystkimi w jednej określonej porze dnia (lub w kilku blokach).

Możesz odkryć, że przydatne jest zapisywanie takich czynności na małych listach zadań do wykonania i w określonym czasie kończenie ich tak szybko, jak to możliwe. Warto wtedy odhaczać je na liście — poczucie dobrze spełnionego obowiązku jest często motywujące.

Co jest naprawdę potrzebne?

Gdy otrzymasz nowe zadanie, ustal, co jest *naprawdę* potrzebne. Czego klient rzeczywiście od Ciebie oczekuje?

Nie twórz pełnej wodotrysków wersji oprogramowania przypominającej Rolls-Royce'a, jeśli nie jest to konieczne. Nawet jeśli klient tego zażądał, najpierw sprawdź, co naprawdę jest potrzebne. W tym celu musisz znać kontekst, w jakim ma działać kod.

Nie chodzi tu o lenistwo. Występuje ryzyko napisania za wcześnie zbyt dużej ilości kodu. Zgodnie z *zasadą Pareto*[2] 80% oczekiwanych korzyści można uzyskać za pomocą 20% planowanej implementacji. Czy naprawdę musisz pisać resztę tego kodu? Możliwe, że uda Ci się lepiej spożytkować swój czas.

Jedna rzecz na raz

Rób jedną rzecz na raz. Trudno jest skoncentrować się na więcej niż jednym zadaniu (dotyczy to zwłaszcza mężczyzn, ponieważ ich mózgi gorzej sobie radzą z wielozadaniowością). Jeśli spróbujesz jednocześnie zajmować się dwoma zadaniami, żadnego z nich nie wykonasz poprawnie. Najpierw zrób jedno, a potem zabierz się za drugie. Wtedy szybciej ukończysz oba zadania.

Pisz krótki (i prosty) kod

Dbaj o to, aby kod i projekty były tak niewielkie i proste, jak to możliwe. W przeciwnym razie utworzysz dużo kodu, na którego konserwowanie będziesz musiał w przyszłości poświęcać czas i wysiłek.

> **KI ➢** Pamiętaj o zasadzie KISS.

Będziesz musiał modyfikować kod. Nigdy nie da się dokładnie przewidzieć przyszłych wymagań. Prognozowanie przyszłości do bardzo nieścisła nauka. Łatwiej i mądrzej jest od początku tworzyć podatny na modyfikacje kod, niż dodawać obsługę każdej funkcji, która może być w przyszłości przydatna.

Niewielki, precyzyjnie wykonujący swoje zadania kod jest znacznie łatwiejszy do zmiany niż rozbudowane projekty.

[2] W wielu sytuacjach 80% efektów uzyskuje się w wyniku 20% działań. Więcej informacji na ten temat znajdziesz na stronie: *http://pl.wikipedia.org/wiki/Zasada_Pareto*.

Nie odkładaj rozwiązań i nie kolekcjonuj problemów

Niektórych trudnych zadań (na przykład integracji kodu) *nie* należy odkładać na później tylko dlatego, że są skomplikowane. Wiele osób popełnia błąd i odracza zadania, aby uniknąć nieprzyjemności. To wygląda jak wybieranie pola bitwy, prawda?

W rzeczywistości lepiej zacząć wcześniej i zmierzyć się z nieprzyjemnościami, gdy jeszcze są niewielkie. Łatwiej jest wcześniej zintegrować niewielkie fragmenty kodu, a następnie często dołączać kolejne zmiany, niż przez rok pracować nad trzema rozbudowanymi funkcjami, a na zakończenie próbować je połączyć ze sobą.

To samo dotyczy testów jednostkowych. Pisz testy od razu, razem z kodem (lub nawet wcześniej). Dużo trudniejsze i mniej produktywne jest czekanie z pisaniem testów do momentu, gdy kod zacznie działać.

Jeśli coś boli, rób to częściej.

Automatyzacja

Pamiętaj starą wskazówkę: *Jeśli musisz coś zrobić więcej niż raz, napisz skrypt, który będzie robił to za Ciebie*.

> **KI ➢** Jeśli często wykonujesz jakieś zadania, spraw, aby komputer robił je za Ciebie. Zautomatyzuj proces za pomocą skryptu.

Automatyzacja często wykonywanych i żmudnych zadań pozwoli Ci uniknąć wielu godzin pracy. Pomyśl też o prostych zadaniach, które jednak trzeba wielokrotnie powtarzać. Możliwe, że szybciej będzie napisać narzędzie i raz je uruchomić, niż ręcznie wykonywać powtarzalną operację.

Automatyzacja ma też dodatkową zaletę — pomaga w bardziej inteligentnej pracy innym. Jeśli zdołasz przygotować proces budowania uruchamiany za pomocą *jednego* polecenia — zamiast serii 15 skomplikowanych instrukcji i kliknięć przycisków — cały zespół będzie mógł sprawniej budować oprogramowanie i będzie można szybciej wdrożyć nowicjuszy w pracę.

Dlatego doświadczeni programiści wybierają możliwe do zautomatyzowania narzędzia, nawet jeśli nie zamierzają w danym momencie automatyzować żadnych zadań. Preferuj procesy pracy, które prowadzą do generowania zwykłego tekstu lub plików pośrednich o prostej strukturze (na przykład w formacie JSON lub XML). Wybieraj narzędzia mające interfejs uruchamiany z poziomu wiersza poleceń oprócz (lub zamiast) mało elastycznego graficznego interfejsu użytkownika.

Czasem trudno stwierdzić, czy warto pisać skrypt do wykonywania danego zadania. Oczywiście, jeśli prawdopodobnie będziesz wykonywał dane zadanie wielokrotnie, warto rozważyć przygotowanie skryptu. Jeżeli nie jest on wyjątkowo trudny do napisania, prawdopodobnie nie zmarnujesz czasu, tworząc go.

Zapobieganie błędom

Znajduj błędy jak najszybciej, aby nie marnować dużo czasu na pracę nad niewłaściwymi rzeczami.

Aby osiągnąć ten cel:

- Pokazuj projekt klientom już na wczesnych etapach pracy i rób to często. Pozwoli to szybko wykryć, że budujesz nie to, co powinieneś.
- Omawiaj projekt kodu z innymi. Dzięki temu szybciej się dowiesz, czy nie da się lepiej ustrukturyzować rozwiązania. Nie wkładaj wysiłku w pracę nad złym kodem, jeśli możesz tego uniknąć.
- Często recenzuj małe, zrozumiałe fragmenty kodu zamiast dużych, skomplikowanych bloków.
- Od początkuj przeprowadzaj testy jednostkowe. Zadbaj o to, aby testy jednostkowe były uruchamiane często, co pozwoli wykryć błędy, zanim staną się problemem.

Komunikacja

Naucz się lepiej komunikować i zadawać właściwe pytania, aby jednoznacznie zrozumieć przekaz. Nieporozumienie może sprawić, że będziesz musiał potem modyfikować kod lub czekać na odpowiedzi na pytania, które za późno zadałeś. Komunikacja jest bardzo ważna, dlatego poświęcony jest jej cały rozdział „Komunikuj się".

Naucz się prowadzić produktywne spotkania, aby demony kryjące się po kątach sali narad nie wyssały z Ciebie całej energii.

Unikaj wypalenia

Nie pracuj po nocach, ponieważ może to prowadzić do przemęczenia i nierealistycznych oczekiwań ze strony przełożonych. Jasno komunikuj, że wykraczasz poza swoje obowiązki, aby inni nie oczekiwali od Ciebie zbyt często takich poświęceń.

W zdrowych projektach nie trzeba często zostawać w nadgodzinach.

Dobre narzędzia

Zawsze szukaj nowych narzędzi, które usprawnią Twój proces pracy.

Nie uzależniaj się jednak od wyszukiwania nowego oprogramowania. Nowe narzędzia często mają ostre krawędzie, o które łatwo się pokaleczyć. Preferuj sprawdzone rozwiązania, z których korzysta wiele osób. Dostępna przez wyszukiwarkę Google wspólna wiedza na temat takich narzędzi jest bezcenna.

Wnioski

Wybieraj pola bitew (tak, tak). *Pracuj mądrzej, a nie ciężej* (już to słyszeliśmy).

Są to wyświechtane powiedzenia, ale zawierające prawdę.

Oczywiście nie oznacza to, że *masz unikać ciężkiej pracy*. Chyba że chcesz, aby Cię zwolnili. Ale to nie jest zbyt mądre.

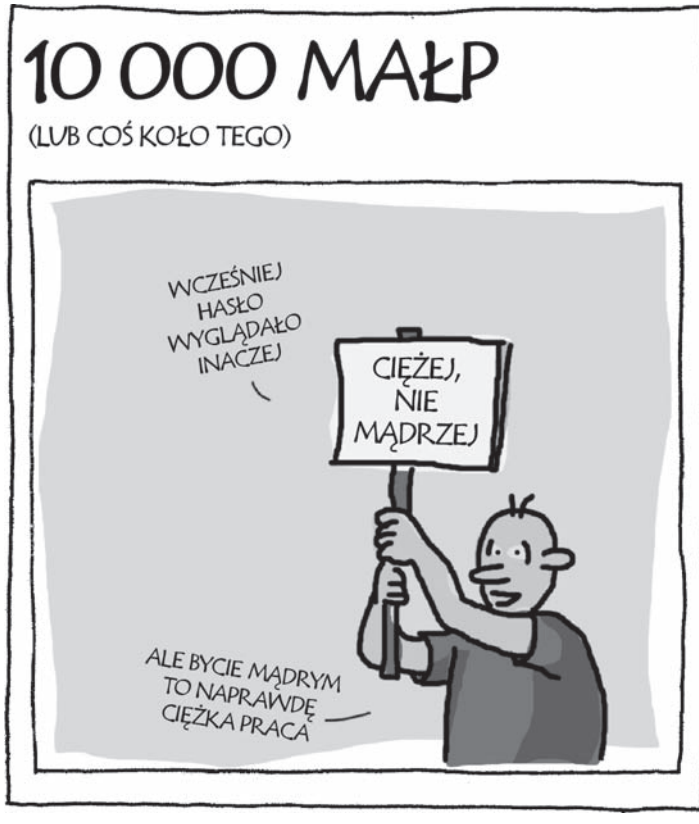
Pytania

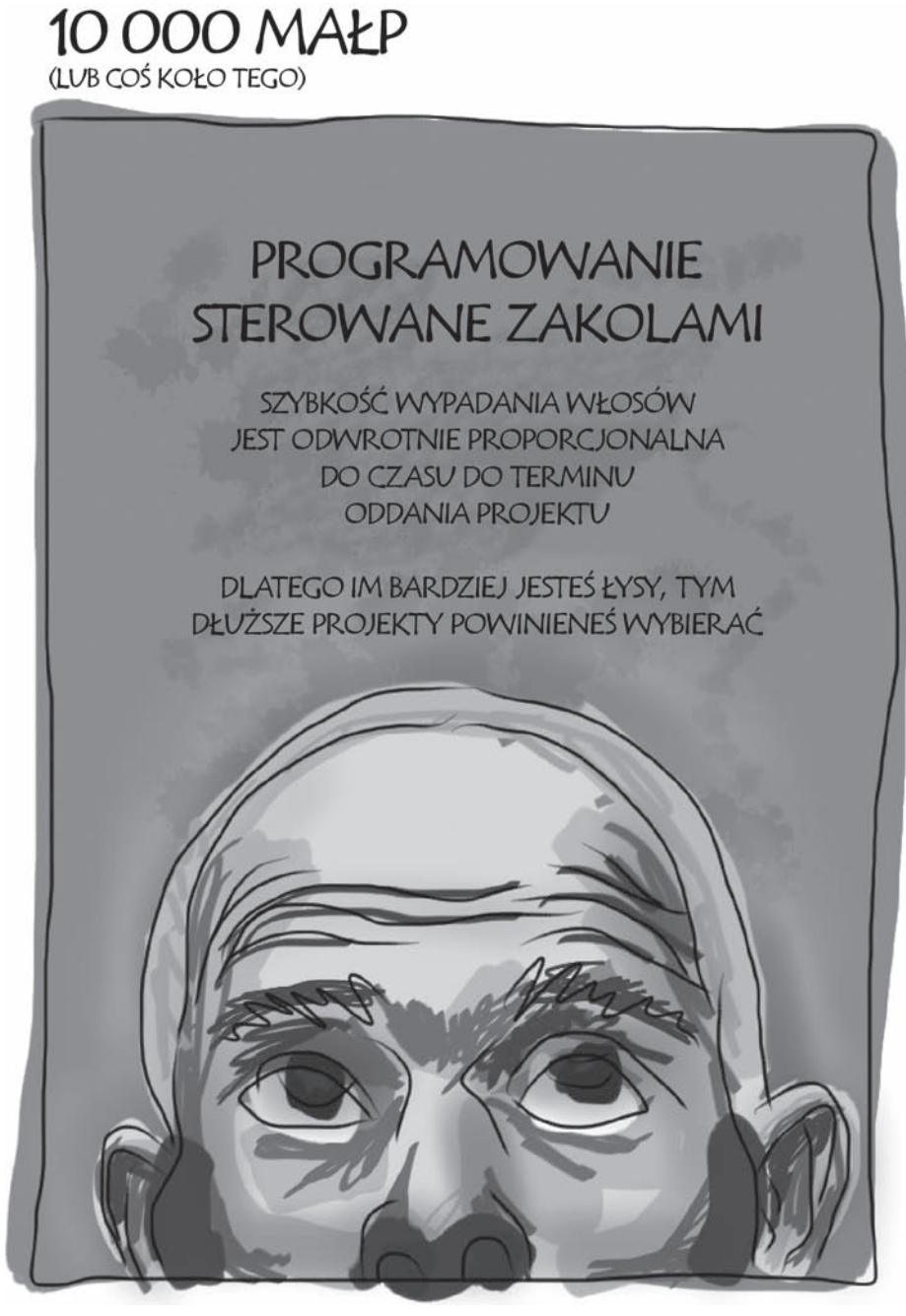
1. Jak określić, na ile dokładnie należy przetestować napisany kod? Korzystasz w tym zakresie z własnego doświadczenia czy z wytycznych? Pomyśl o kodzie, który napisałeś w ostatnich miesiącach. Czy na pewno został właściwie przetestowany?
2. Jak dobrze radzisz sobie z określaniem priorytetów prac? Czy możesz coś poprawić w tym obszarze?
3. Jak dbasz o to, aby jak najszybciej wykrywać usterki? Ilu popełnionych błędów lub przeróbek mogłeś uniknąć?
4. Czy cierpisz na syndrom *nie wynalezione u nas*? Czy z definicji uważasz kod innych programistów za kiepski? Czy możesz poprawić swoje nastawienie? Czy potrafisz pogodzić się z łączeniem kodu innych osób z własnym?
5. Jeśli pracujesz w środowisku, w którym liczbę przepracowanych godzin ceni się bardziej niż jakość kodu, jak możesz mądrze pracować, a jednocześnie nie wyjść na lenia?

Zobacz także

- „Tym razem na pewno będzie dobrze…". Tu znajdziesz pouczającą historię — łatwo jest pracować mniej *mądrze*, niż potrafisz.
- „Ponowne wykorzystanie kodu". Zastosuj inteligentne podejście do ponownego wykorzystania kodu. Unikaj chaosu związanego z powtórzeniami, ale nie pisz więcej kodu, niż jest to konieczne.
- „Kiedy kod jest gotowy?". Nie pisz więcej kodu, niż jest to niezbędne. Naucz się określać, kiedy rozwiązanie jest gotowe.

> **Spróbuj tego...**
>
> Wskaż trzy rzeczy, które pomogą Ci stać się *bardziej produktywnym* programistą. Postaraj się znaleźć dwie nowe techniki, które możesz zacząć stosować, i jeden szkodliwy nawyk. Zacznij korzystać z tych technik już od jutra. Niech ktoś sprawdza, czy je stosujesz.





Skorowidz

K

L

Ł

M

N

O

P

R

S

Ś

T

U

W

Z

O'REILLY®

Jak stać się lepszym programistą Przewodnik

Zawód programisty jak żaden inny wymaga ciągłego rozwoju, nauki i doskonalenia. Każdy dzień to nowe wyzwania, techniki programistyczne oraz kolejne linie wysokiej jakości kodu. Zastanawiasz się, co jeszcze możesz zrobić, aby stać się lepszym programistą? Chcesz zaimponować ciekawymi koncepcjami oraz wprowadzić znaczące ulepszenia w codziennej pracy? Trafiłeś na książkę, która sprawi, że Twój kod nabierze blasku!

Na kolejnych stronach tej książki przeczytasz o tym, jak powinien być sformatowany kod źródłowy, jakie stosować nazewnictwo oraz jak usuwanie zbędnego kodu wpływa na jego jakość. Nauczysz się odnajdywać błędy w kodzie i pisać dobre testy. Dowiesz się także, jak w pełni korzystać z możliwości systemu kontroli wersji. Książka ta jest lekturą obowiązkową dla każdego programisty ceniącego sobie ciągły rozwój, dążenie do doskonałości oraz tworzenie świetnego kodu!

Sprawdź:

- jak formatować kod
- jak budować testy
- jak sprawnie nawigować po kodzie źródłowym
- jak tworzyć dobry kod

Lektura obowiązkowa dla każdego programisty!

Peter Goodliffe — muzyk i programista. Autor artykułów i książek poświęconych programowaniu. Dysponuje szeroką wiedzą oraz równie dużym doświadczeniem programistycznym w zakresie implementacji systemu operacyjnego, tworzenia aplikacji desktopowych i aplikacji mobilnych, a także systemów wbudowanych.

Helion

31821 numer katalogowy

księgarnia internetowa

http://helion.pl

zamówienia telefoniczne

0 801 339900

0 601 339900

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

ISBN 978-83-283-0239-6

Informatyka w najlepszym wydaniu

cena 57,00 zł