

O'REILLY®



Przewodnik

Jak stać się lepszym programistą

PODRECZNIK PRAWDZIWEGO PROGRAMISTY!

Tytuł oryginału: Becoming a Better Programmer

Tłumaczenie: Tomasz Walczak

ISBN: 978-83-283-0239-6

© 2015 Helion S.A.

Authorized Polish translation of the English edition of Becoming a Better Programmer, ISBN 9781491905531 © 2015 Pete Goodliffe.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

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.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

Wprowadzenie	11
1. Troska o jakość kodu	15
<i>O właściwym podejściu i nastawieniu do kodu</i>	
<hr/>	
I ty.piszesz(kod);	19
2. Utrzymywanie pozorów	21
<i>Prezentacja kodu — układ i nazwy</i>	
3. Pisz mniej kodu!	31
<i>Unikanie zbędnych wierszy kodu</i>	
4. Ulepszanie kodu przez jego usuwanie	43
<i>Identyfikowanie i usuwanie martwego kodu</i>	
5. Duch minionego kodu bazowego	51
<i>Jak się uczyć na podstawie własnego starszego kodu?</i>	
6. Odnajdowanie się w kodzie	59
<i>Jak się poruszać po nieznanym kodzie?</i>	
7. Po uszy w bagnie	69
<i>Jak radzić sobie z chaotycznym kodem?</i>	
8. Nie ignoruj tego błędu!	77
<i>Zalecane podejście do obsługi błędów</i>	
9. Oczekuj nieoczekiwanego	83
<i>O pisaniu niezawodnego kodu odpornego na wszelkie okoliczności</i>	
10. Polowanie na błędy	87
<i>Jak wyszukiwać i naprawiać błędy?</i>	

11. Czas na testy	99
<i>Testy wykonywane przez programistów: jednostkowe, integracyjne i systemowe</i>	
12. Radzenie sobie ze złożonością	113
<i>Jak dobrze projektować kod i uniknąć zbędnej złożoności?</i>	
13. Historia dwóch systemów	123
<i>Konsekwencje dobrych i złych projektów</i>	

II Praktyka czyni mistrza **137**

14. Tworzenie oprogramowania to...	139
<i>Czym jest to całe programowanie?</i>	
15. Zgodnie z regułami gry	149
<i>Ustalanie reguł, według których funkcjonuje zespół programistyczny</i>	
16. Zachowaj prostotę	153
<i>Twórz proste oprogramowanie</i>	
17. Użyj głowy	159
<i>Programiści mogą i powinni używać głowy — nie bądź głupi!</i>	
18. Nic nie jest niezmienne	163
<i>Żaden kod nie jest święty, a wszystko się zmienia</i>	
19. Ponowne wykorzystanie kodu	169
<i>Właściwe sposoby powtórnego wykorzystania kodu</i>	
20. Skuteczna kontrola wersji	175
<i>Poprawna kontrola wersji</i>	
21. Jak umieścić piłkę w bramce?	187
<i>Skuteczna współpraca z działem kontroli jakości</i>	
22. Ciekawy przypadek zamrożonego kodu	199
<i>Zamrażanie kodu — czym jest i dlaczego jest niezbędne?</i>	
23. Proszę o wydanie	207
<i>Udostępnianie oprogramowania</i>	

III Sprawy osobiste **215**

24. Żyj tak, aby kochać się uczyć	217
<i>Jak uczyć się skutecznie?</i>	
25. Programiści sterowani testami	227
<i>Programowanie jest jak prowadzenie samochodu — jak się tego nauczyć i zdać egzamin?</i>	

26. Zagustuj w wyzwaniach	233
<i>Stawianie sobie wyzwań, aby zachować motywację i rozwijać umiejętności</i>	
27. Unikaj stagnacji	239
<i>Co możesz zrobić, aby Twoje umiejętności programistyczne nie stały się przestarzałe?</i>	
28. Etyczny programista	245
<i>Kwestie etyczne w życiu programisty</i>	
29. Miłość do języków	253
<i>Poznaj wiele języków programowania i pokochaj te, których używasz</i>	
30. Postawa programisty	261
<i>Dbanie o zdrowie programisty: postawa ciała, zmęczenie oczu i morale</i>	
<hr/>	
IV Doprowadzanie rzeczy do końca	269
31. Pracuj mądrzej, a nie ciężiej	271
<i>Skuteczna praca — unikanie błędnych zadań i rozwiązywanie właściwych problemów</i>	
32. Kiedy kod jest gotowy?	281
<i>Zdefiniuj swoje zadania i ustal, kiedy uznasz je za wykonane</i>	
33. Tym razem na pewno będzie dobrze... ..	289
<i>Zdejmij klapki z oczu — znajdź najlepszy sposób rozwiązania problemu</i>	
<hr/>	
V Kwestie międzyludzkie	295
34. Presja społeczna	297
<i>Jak znaleźć się w zespole świetnych programistów i dobrze z nimi współpracować?</i>	
35. Liczy się myślenie	303
<i>Bycie odpowiedzialnym wpływa pozytywnie na Ciebie i Twoją pracę</i>	
36. Komunikuj się!	309
<i>Umiejętności komunikowania się dla programistów</i>	
37. Manifesty	319
<i>Manifesty w świecie oprogramowania — czym są i do czego służą?</i>	
38. Oda do kodu	325
<i>Historia z morałem o nieumiejętnym zarządzaniu oprogramowaniem</i>	
Epilog	329
Skorowidz	332

Pracuj mądrzej, a nie ciężiej

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, brnąć w ślepią 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 dokonaniem. 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 przeemyśl, które testy warto napisać. Szttywne 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 API¹. 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 zadziało, 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 zadania, 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 produktywnie 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

Znajdź 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 powinienes.
- 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 początku 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 wyszały 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 ciężiej (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 prze-róbek 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.

10 000 MAŁP

(LUB COŚ KOŁO TEGO)



10 000 MAŁP

(LUB COŚ KOŁO TEGO)

PROGRAMOWANIE STEROWANE ZAKOŁAMI

SZYBKOŚĆ WYPADANIA WŁOSÓW
JEST ODWROTNIE PROPORCJONALNA
DO CZASU DO TERMINU
ODDANIA PROJEKTU

DLATEGO IM BARDZIEJ JESTEŚ ŁYSY, TYM
DŁUŻSZE PROJEKTY POWINIENES WYBIERAĆ



A

analiza
 kodu, 64
 przyczyn błędów, 92
 statyczna, 62
archeologia oprogramowania,
 71, 90
architektura, 62
 oprogramowania, 130
 systemu Metropolis, 124
asercje, 90
atomowe zmiany, 180
atrapy, mock, 110
automatyzacja, 275
automatyzacja testów, 100

B

błędny układ kodu, 22
błędy, 55, 77, 83, 87–95, 194,
 276
błędy nieregularne, 94
brak obsługi błędów, 79
budowanie wydania, 209

C

cechy dobrego testu, 105
certyfikat, 230, 245
Chaotyczne Metropolis
 niepotrzebne powiązania, 126
 niespójność, 125
 niezrozumiałość, 124
 problemy poza kodem, 127
 problemy z kodem, 126
 zła architektura, 128

ciągła integracja, 104
ciekawe projekty, 234
cierpliwość, 257
czas projektowania, 266

Ć

ćwiczenie oczu, 267

D

debuger, 92
debugowanie, 88, 92
decyzje projektowe, 55
dekompozycja, 282
dług techniczny, 131, 167, 203
długość okresu zamrożenia, 203
dobra
 architektura, 134
 komunikacja, 314
dobre
 narzędzia, 276
 relacje w pracy, 195
 testy, 105
dobry kod, 111, 310
dokumentacja, 62, 65
dostęp do pamięci, 95
dostrzeganie problemów, 69
doświadczenie, 299
dubler, 110
dział kontroli jakości, 191, 196

E

efekty uboczne, 78
egzamin, 229
ekspert, 298

etyczne postawy, 247
etyka, 245, 246, 326

F

funkcja, 83

G

gotowe rozwiązanie, 283

I

idiomy, 27, 54
ignorowanie możliwych
 błędów, 77
informacje
 wielomodalne, 219
 zwrotne, 99
informowanie o postępach, 291
inicjowanie tworzenia
 wydania, 209
interakcje
 między zespołami, 189
 w sieci, 94

J

jakość
 kodu, 15, 62, 70, 111, 305
 projektu, 131
jednoznaczność, 27
język
 C++, 254, 311
 ciała, 313
 naturalny, 311

języki

- assemblerowe, 254
- deklaratywne, 254
- funkcyjne, 254
- kompilowane, 253
- logiczne, 254
- skryptowe, 253

K

kod, 309

- bazowy, 51
- do testowania kodu, 100
- interfejsu użytkownika, 45
- kontenerowy, 73
- podatny na błędy, 79
- pomocniczy, 45
- wielowątkowy, 94

kodeks etyczny, 250

komentarze, 36, 310

kompetencje, 220

komunikacja, 257, 276, 309, 312, 315

równoległa, 314

w kontekście zespołów, 314

konserwowanie

kodu, 45

testów, 108

kontekst, 26, 106

kontrola

jakości, 188, 191

wersji, 175

konwersacja, 312

kończenie

pracy programu, 84

zadania, 281

kopiowanie własności

intelektualnej, 247

koszty niepotrzebnego kodu, 45

kółka, 114

kwestie

międzyludzkie, 295

prawne, 246

L

licencja GPL, 246

liczba komentarzy, 310

linie, 116

ludzie, 118

Ł

łatwe cele, 63

łatwość

budowania kodu, 61

pobierania kodu

źródłowego, 60

M

manifest, 321

Craftsmanship Manifesto, 320

GNU, 320

The Agile Manifesto, 320

The Hacker Manifesto, 320

The Refactoring Manifesto, 320

manifesty uniwersalne, 319

mapa powiązań, 117

martwy kod, 36, 44, 45

menedżer, 249

mentor, 298

Miasto Projektu

praca z projektem, 133

projektowanie, 133

rozwijanie architektury, 130

spójność, 129

testy jednostkowe, 132

utrzymywanie jakości, 131

zasada YAGNI, 130

znajdowanie funkcji, 129

model

Dreyfusa, 221

liniowy, 187

wodospadu, 195

modele uczenia się, 221

modyfikowanie

działania kodu, 27, 72, 163

interfejsu API, 73

układu kodu, 27, 72

motywacja, 233

narzędzie

Cucumber, 109

debuger, 92

Electric Fence, 91

Fit, 109

Valgrind, 92

nastawienie, 330

nazwy, 26

nazwy testów, 107

niedbała logika, 32

niekonieczna złożoność, 114

niepotrzebne

funkcje, 44

powiązania, 126

niepotrzebny kod, 32, 43, 45

niespójność systemu, 125

nieświadoma

kompetencja, 220, 227

niekompetencja, 220

niezabezpieczony kod, 79

NNW, Narzędzie Niszczenia

Wzroku, 261

O

obiekty fikcyjne, dummy, 110

obsługa

błędów, 77, 83, 84

odgałęzień, 181

odgałęzienia, 181

odgałęzienie wydania, 201, 210

odpowiedzialność, 304, 306

odróżnianie decyzji

projektowych, 130

odstępny, 25

okres zamrożenia, 203

oprogramowanie, 139

optymalizacja, 156

organizacja

ACM, 245

BSI, 245

P

P2P, 177

pakowanie wydania, 211

pętle z informacjami

zwrotnymi, 100

pisanie kodu, 309

plik README, 65

- podział
 - na połowy, 90
 - zadania, 282
- ponowne wykorzystanie kodu, 169–171
- poprawki, 72
- portfel wiedzy, 222
- porządkowanie kodu, 64, 71
- postawa programisty, 261–266
- powiązanie, 125
- powstawanie błędów nieregularnych, 94
- powtórzenia, 33
- poziom
 - kompetencji, 220
 - wykorzystania pamięci, 95
 - złożoności, 114
- poznawanie kodu, 60
- praca, 271
- praca zespołowa, 315
- pracodawca, 249
- praktyka, 63, 137
- prawo Goodliffe’a, 21
- presja społeczna, 297
- prewencja, 88
- prezentacja kodu, 22
- priorytety, 273
- problemy, 69, 326
 - poza kodem, 127
 - z kodem, 126
- programowanie ekstremalne, 43
- od końca, 282
- sterowane testami, 99
- w parach, 305

proste projekty

- krótsze ścieżki, 155
- stabilność, 155
- użytkowanie, 154
- wielkość, 154
- zapobieganie błędnemu użyciu, 154

proste wiersze kodu, 155

prostota, 156

prototyp, 273

przechowywanie

- jak najmniej danych, 178
- wszystkich plików, 178
- wydań oprogramowania, 179

przedwczesna optymalizacja, 156

przekazywanie informacji, 23

przyczyny problemów, 326

przygotowanie wydania, 207, 209

przysięga Hipokodesa, 250

pułapki, 90

punkt

- RTM, 204
- zakończenia prac, 199

R

ramki z komentarzami, 23

raport o błędach, 194

RC, release candidate, 200

refaktoryzacja, 34, 165, 273

regresja oprogramowania, 192

reguła KISS, 153, 159

reguły, 149–151, 200

rejestrowanie zdarzeń, 90

relacje, 118

- w pracy, 195
- z językiem, 256

repozytorium, 177–179

repozytorium z kontrolą wersji, 182

rodzaje

- dublerów, 110
- testów, 101
- zamrażania, 201

rozdzielanie pracy zespołów, 190

rozmowa z klientami, 315

rozproszona tożsamość, 118

rozwijanie

- architektury, 130
- kodu, 165

rozwlekłość, 37

rozwój

- kompetencji, 221
- osobisty, 215

RTM, release to manufacture, 199

S

schematy układu kodu, 52

singletony, 95, 109

spójność, 25, 125, 129

sprawdzanie pamięci, 91

stagnacja, 239

standard pisania kodu, 26

strategia podziału na połowy, 90

strategie pośrednie, 93

struktura

- katalogów, 61
- kodu, 24
- organizacji, 190
- powiązań, 117
- repozytorium, 179
- testów, 108
- zespołu, 133

SUT, System Under Test, 101

syndrom rozbitych szyb, 93

system

- CVS, 177
- kontroli wersji, 175, 180, 183
- oprogramowania, 117, 123
 - Chaotyczne Metropolis, 124
 - Miasto Projektu, 128

systemy

- rozproszone, 177
- scentralizowane, 177

szukanie wskazówek, 60

szybki prototyp, spike, 273

Ś

śledzenie historii zmian, 176

świadoma

- kompetencja, 220, 228
- niekompetencja, 220

T

TDD, test-drive development, 99, 102

technika

- kopiuj-wklej, 169
- TDD, 102

techniki inżynierii, 88
test double, 110
testowanie jednostki, 109
testy, 61, 64, 99–111, 195
 cechy, 105
 konserwacja, 108
 nazwy, 107
 rodzaje, 101
 struktura, 108
 uruchamianie, 104
 wybór platformy, 109
testy
 integracyjne, 102
 jednostkowe, 91, 101, 132, 191, 273
 systemowe, 102
 zautomatyzowane, 132, 167
tworzenie
mapy powiązań, 117
tworzenie
 nazw, 26
 niepotrzebnego kodu, 44
 oprogramowania, 140–145
 portfela wiedzy, 222
 systemu oprogramowania, 117
 wersji, 193
 wydania oprogramowania, 208

U

uczenie się, 218, 223, 241
udostępnianie
 wersji, 191, 192
 wydania, 211
 oprogramowania, 199
układ kodu, 24, 52
ułatwianie wykrywania
 błędów, 79
umiejętności, 239
umiejętności programistyczne, 230

unikanie
 beźmyślności, 160
 błędów, 22
 zamrażania, 204
upraszczanie relacji, 118
uruchamianie testu, 104
ustalanie
 priorytetów, 273
 reguł, 150
usuwanie
 funkcji, 44
 kodu, 43
 martwego kodu, 45, 46
utrata danych, 176
utrzymywanie jakości, 131

W

wątki, 84
wersja oprogramowania, 191
 alfa, 200
 beta, 200
 RC, 200
wiadomości na temat zmian, 180
własność intelektualna, 247
wprowadzanie poprawek, 72, 164, 166
wskazówki, 60
wspólne wartości, 258
wydanie oprogramowania, 207, 208
 budowanie, 209
 inicjowanie, 209
 pakowanie, 211
 przygotowywanie, 209
 udostępnianie, 211
wyjątki, 78
wykorzystywanie gotowego kodu, 272
wykrywanie błędów, 79, 87, 89
wymagania, 62
wystarczalność, 157

wyszukiwanie martwego kodu, 46
wytyczne dotyczące stylu, 26
wyznaczanie standardów, 306
wyzwania, 234, 235

Z

zaangażowanie, 256
zadanie gotowe, 283
zależności projektu, 62
założenia, 156
zamrażanie kodu, 199–205
zapobieganie
 błędom, 276
 infekcjom, 92
zarządzanie długiem technicznym, 132
zasada, 149–151
 DRY, 34, 169
 KISS, 274
 YAGNI, 130, 171
zbiory reguł, 150
zespół, 187, 190, 248, 315
zła
 architektura, 128
 struktura, 79
złe
 nazwy, 22
 testy, 105
złożoność w
 oprogramowaniu, 113
zły
 kod, 74, 285
 projekt, 38
zmęczenie oczu, 266
zmiany przesłane do repozytorium, 180, 181
zmienne globalne, 95, 109
zmniejszanie złożoności kółek, 115
znajdowanie funkcji, 129
zwracane kody błędów, 77

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



- 1. ZAREJESTRUJ SIĘ**
- 2. PREZENTUJ KSIĄZKI**
- 3. ZBIERAJ PROWIZJĘ**

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA

 **Helion SA**

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? Trafieś 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>

sięgnij po WIĘCEJ



KOD KORZYŚCI

ISBN 978-83-283-0239-6



9 788328 130239 6