
Helion
Zrównoważony
kod
Dobre praktyki
i heurystyki
dla inżynierów
oprogramowania
Mark Seemann
Przedmowa Robert C. Martin

Tytuł oryginału: Code That Fits in Your Head: Heuristics for Software Engineering (Robert C. Martin Series)

Tłumaczenie: Joanna Zatorska

ISBN: 978-83-283-9226-7

Page 31, author photo: © Linea Vega Seemann Jacobsen
Page 45, Queen Alexandrine's Bridge, Denmark: Ulla Seemann
Page 68, baseball and bat: buriy/123RF
Page 73, illustration of human brain: maglyvi/Shutterstock
Page 73, illustration of laptop computer: grmarc/Shutterstock
Page 195, Figure 8.2: © Microsoft 2021
Page 196, Figure 8.3, scissors: Hurst Photo/Shutterstock
Page 196, Figure 8.3, hand saw: Andrei Kuzmik/Shutterstock
Page 196, Figure 8.3, utility knife: Yogamreet/Shutterstock
Page 196, Figure 8.3, Phillips-head screwdriver: bozmp/Shutterstock
Page 196, Figure 8.3, Swiss military knife: Billion Photos/Shutterstock
Page 197, Figure 8.4: Roman Babakin/Shutterstock
Page 209, Figure 8.5: © Microsoft 2021
Page 282, Figure 12.1: ajt/Shutterstock
Page 304, Figure 13.2, bursting star: Arcady/Shutterstock
Page 314, Figure 13.5: Verdandi/123RF
Page 324, Figure 14.1: Tatyana Pronina/Shutterstock
Page 338, Figure 15.2: kornilov007/Shutterstock
Page 339, hammer: bozmp/Shutterstock
Pages 355, Figure 15.3: Figure based on a screen shot from codescene.io
Pages 356, Figure 15.4: Figure based on a screen shot from codescene.io

Authorized translation from the English language edition, entitled Code That Fits in Your Head: Heuristics for Software Engineering, 1st Edition by Mark Seemann, published by Pearson Education, Inc, publishing as Addison Wesley Professional, Copyright © 2022 Pearson Education, Inc.

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 Pearson Education, Inc.

Polish language edition published by Helion S.A., Copyright © 2022.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek 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 wydawca 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 wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

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

Pliki z przykładami omawianymi w książce można znaleźć pod adresem: *https://ftp.helion.pl/przyklady/kodkto.zip*

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

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

1 Sztuka czy nauka?

Jesteś naukowcem lub artystą? Inżynierem lub rzemieślnikiem? A może ogrodnikiem lub szefem kuchni? Poetą lub architektem?

Jesteś programistą lub inżynierem oprogramowania? Kim właściwie jesteś?

Moja odpowiedź na te pytania brzmi: *nikim z wymienionych*.

Chociaż uważam, że jestem programistą, tak naprawdę mam cechy przedstawicieli wszystkich wymienionych zawodów. A zarazem nie jestem żadnym z nich.

Takie pytania są ważne. Branża tworzenia oprogramowania liczy sobie około 70 lat i ciągle nie wiadomo, czym właściwie jest. Ciągle nie udaje się rozwiązać problemu z ustaleniem właściwego sposobu *myślenia* o niej. Stąd wcześniejsze pytania. Czy tworzenie oprogramowania można porównać do budowy domu? A może do pisania wiersza?

Przez dziesięciolecia wypróbowaliśmy wiele metafor, ale żadna z nich się nie przyjęła. Tworzenie oprogramowania przypomina budowę domu, ale nią nie jest. Tworzenie oprogramowania przypomina pielęgnację ogrodu, ale nią nie jest. Ostatecznie żadna metafora nie spełniła pokładanych w niej nadziei.

Kup książkę Poleć książkę

Wierzę jednak, że sposób myślenia o tworzeniu oprogramowania wpływa na nasz sposób pracy.

Jeśli uważasz, że tworzenie oprogramowania przypomina budowę domu, będziesz popełniał błędy.

1.1. BUDOWANIE DOMU

Przez dziesięciolecia tworzenie oprogramowania porównywano do budowy domu. Oto jak ujął tę koncepcję Kent Beck:

> *Niestety projektowanie oprogramowania zostało zacementowane metaforami określającymi projektowanie obiektów fizycznych. [5]*

Jest to jedna z najbardziej wszechobecnych, złudnych i utrudniających pracę metafor opisujących tworzenie oprogramowania.

1.1.1. PROBLEM ZWIĄZANY Z PROJEKTAMI

Jeśli uważasz, że tworzenie oprogramowania można porównać do budowania domu, błędnie traktujesz je jak *projekt*. Projekt ma początek i koniec. Gdy dobiegnie końca, praca jest wykonana.

Koniec dotyczy tylko oprogramowania, które nie cieszy się powodzeniem. Oprogramowanie, które odniesie sukces, przetrwa. Jeśli masz szczęście pracować nad popularnym oprogramowaniem, po opublikowaniu jednej wersji zaczynasz pracę nad kolejną. To może trwać wiele lat. Niektóre programy, które cieszą się powodzeniem, są rozwijane przez dziesięciolecia[1].

Do zbudowanego domu mogą się wprowadzić ludzie. Trzeba go remontować, ale kosztuje to ułamek ceny budowy. Z pewnością istnieje oprogramowanie

[1] Tę książkę piszę z wykorzystaniem programu LaTeX — opublikowanego po raz pierwszy w 1984 roku!

Kup książkę Poleć książkę

o podobnych cechach. Szczególnie w sektorze przedsiębiorstw po zbudowaniu[2] wewnętrznej aplikacji biznesowej uznaje się ją za skończoną, a użytkownicy muszą z niej korzystać. Tego typu oprogramowanie przechodzi w fazę utrzymania od razu po zakończeniu projektu.

Jednak większość programów nie pasuje do tego schematu. Oprogramowanie, które konkuruje z innym, nigdy nie jest gotowe. Jeśli skorzystamy ponownie z metafory budowy domu, możemy sobie wyobrazić, że tworzenie oprogramowania obejmuje serię projektów. Możesz zaplanować wydanie kolejnej wersji produktu za dziewięć miesięcy i, ku swojemu przerażeniu, odkryć, że konkurencja publikuje ulepszenia co trzy miesiące.

Zatem pracujesz ciężko, aby skrócić czas trwania „projektów". Gdy wreszcie możesz publikować kolejne wersje co trzy miesiące, okaże się, że konkurent publikuje je co miesiąc. Rozumiesz już, do czego to może doprowadzić?

Prowadzi to do wdrożenia systemu ciągłego dostarczania [49]. Albo z niego skorzystasz, albo wypadniesz z gry. Książka *Przyspieszenie* [29], opierając się na badaniach, przekonuje, że możliwość natychmiastowej publikacji nowej wersji jest kluczowym kryterium w ustaleniu, czy zespoły mają wysoką czy niską wydajność.

Jeśli masz taką możliwość, pojęcie projektu w tworzeniu oprogramowania traci sens.

1.1.2. PROBLEM ETAPÓW

Kolejnym błędem, zwykle wynikającym ze stosowania metafory budowania domu, jest wiara, że tworzenie oprogramowania powinno się odbywać w osobnych *etapach*. Podczas budowania domu architekt najpierw opracowuje plany. Następnie trzeba zadbać o logistykę, dostarczyć materiały budowlane, a następnie można przystąpić do budowy.

[2] Staram się nie używać tego czasownika w odniesieniu do tworzenia oprogramowania, ale w tym kontekście dobrze oddaje on sens mojego wywodu.

Kup książkę Poleć książkę

Jeśli tworzenie oprogramowania postrzegamy w ten sposób, wyznaczamy *architekta oprogramowania*, który powinien opracować plan. Dopiero gdy plan jest gotowy, można przystąpić do prac programistycznych. Z tej perspektywy w fazie planowania wykonuje się pracę umysłową. Zgodnie z metaforą faza programowania przypomina fazę konstrukcji domu. Programiści są postrzegani jak łatwo wymienni pracownicy[3], tylko nieco przewyższający umiejętnościami gloryfikowane maszynistki.

Jest to bardzo mylne podejście. Jack Reeves napisał w 1992 r. [87], że etap *konstrukcji* w tworzeniu oprogramowania występuje podczas kompilowania kodu źródłowego. W przeciwieństwie do budowy domu właściwie nie wymaga żadnego wysiłku. Całą pracę wykonuje się w fazie *projektowania*, czy też, zgodnie z trafną uwagą Kevlina Henneya:

> *Czynność opisywania programu w jednoznaczny sposób oraz czynność programowania są tożsame. [42]*

Podczas tworzenia oprogramowania nie da się wyróżnić fazy konstrukcji. Nie sugeruje to, że planowanie jest bezużyteczne, ale oznacza, że metafora budowy domu jest co najmniej bezużyteczna.

1.1.3. ZALEŻNOŚCI

Podczas budowy domu uwarunkowania fizyczne nakładają pewne ograniczenia. Najpierw trzeba wykonać fundamenty, następnie wznieść ściany, a dopiero potem można położyć dach. Innymi słowy, dach zależy od ścian, które zależą od fundamentów.

Ta metafora sprawia, że ludzie są przekonani o konieczności zarządzania zależnościami. Spotkałem się z menedżerami projektów, którzy podczas planowania projektów opracowywali złożone wykresy Gantta.

Pracowałem z wieloma zespołami i większość z nich zaczynała nowe projekty programistyczne od opracowania schematu relacyjnej bazy danych. Baza

[3] Nie mam nic przeciwko budowlańcom; mój ukochany ojciec był murarzem.

Kup książkę Poleć książkę

danych jest podstawą większości usług internetowych, a zespoły wydawały się nieświadome możliwości rozwijania interfejsu użytkownika przed utworzeniem bazy danych.

Niektóre zespoły nigdy nie zdołają stworzyć działającego oprogramowania. Po zaprojektowaniu bazy danych uznają, że potrzebują towarzyszącej jej *platformy*. Zatem przystępują do opracowania narzędzia do mapowania obiektowo-relacyjnego, które można przyrównać do wojny wietnamskiej w świecie informatyki [70].

Metafora budowy domu jest szkodliwa, ponieważ sprawia, że o tworzeniu oprogramowania myślimy w określony sposób. Omijają nas przez to różne możliwości, ponieważ nasza perspektywa rozmija się z rzeczywistością. W rzeczywistości, tworząc oprogramowanie, *można*, mówiąc metaforycznie, zacząć od dachu. Potwierdzę to przykładem w dalszej części książki.

1.2. PIELĘGNACJA OGRODU

Metafora budowy domu jest błędna, ale może inna sprawdzi się lepiej. Metafora pielęgnacji ogrodu zaczęła zdobywać popularność w drugiej dekadzie XXI w. Nieprzypadkowo Nat Pryce i Steve Freeman zatytułowali swoją doskonałą książkę *Growing Object-Oriented Software, Guided by Tests* [36].

Zgodnie z tym podejściem do tworzenia oprogramowania program jest żywą istotą, której trzeba doglądać, którą trzeba pielęgnować i przycinać. Jest to kolejna przekonująca metafora. Czy kiedykolwiek czułeś, że kod źródłowy żyje własnym życiem?

Takie postrzeganie tworzenia oprogramowania może być oświecające, a przynajmniej wymusza zmianę perspektywy i może zburzyć wiarę w to, że tworzenie oprogramowania przypomina budowanie domu.

Postrzeganie oprogramowania, zgodnie z metaforą ogrodu, jako żywego organizmu kładzie nacisk na przycinanie. Ogród pozostawiony bez opieki zdziczeje. Aby ogród przyniósł korzyści, ogrodnik musi usuwać chwasty

Kup książkę Poleć książkę

i zapewniać podporę pielęgnowanym roślinom. Przekładając te czynności na tworzenie oprogramowania, należy się skupić na działaniach *zapobiegających* degradacji kodu, takich jak refaktoryzacja i usuwanie nieużywanego kodu.

Według mnie ta metafora nie jest równie problematyczna jak metafora budowania domu, ale nadal uważam, że pomija pewne aspekty.

1.2.1. DZIĘKI CZEMU OGRÓD ROŚNIE?

Podoba mi się, że metafora ogrodu podkreśla działania zwalczające nieporządek. Podobnie jak ogrodnik musi przycinać rośliny w ogrodzie i pielić chwasty, programista musi refaktoryzować kod źródłowy i redukować w nim dług techniczny.

Z drugiej strony metafora ogrodu pomija pochodzenie kodu. W ogrodzie rośliny rosną samodzielnie. Potrzebują tylko składników odżywczych, wody i światła. Natomiast oprogramowanie nie rośnie automatycznie. Nie wystarczy po prostu postawić komputera w ciemnym pokoju, przynieść czipsy i napoje i czekać na powstanie programu. Zabraknie bowiem istotnych składników: programistów.

Ktoś musi napisać kod. Jest to proces, którego próżno szukać w metaforze ogrodu. Skąd wiadomo, co napisać, a czego nie? Skąd wiadomo, *jaka* powinna być struktura fragmentu kodu?

Aby ulepszyć branżę tworzenia oprogramowania, trzeba odpowiedzieć również na te pytania.

1.3. W KIERUNKU INŻYNIERII

Tworzenie oprogramowania opisuje się też innymi metaforami. Pisałem już o *długu technicznym*, który przywodzi na myśl punkt widzenia księgowego. Pisałem też o procesie *pisania* kodu, który można porównać do pracy nad różnymi tekstami. Niektóre metafory są całkowicie chybione, a żadna z nich nie jest w pełni poprawna.

Kup książkę Poleć książkę

Nie bez powodu skupiłem się na metaforze budowy domu. Przede wszystkim jest ona bardzo rozpowszechniona. Ponadto wydaje się tak myląca, że nie da się jej obronić.

1.3.1. OPROGRAMOWANIE JAKO RZEMIOSŁO

Już wiele lat temu doszedłem do wniosku, że metafora budowy domu jest szkodliwa. Gdy porzucisz pewien sposób widzenia, zwykle szukasz innego. Ja znalazłem go w *rzemiośle programistycznym*.

Postrzeganie tworzenia oprogramowania jako rzemiosła jest przekonujące, ponieważ oznacza ono pracę *wykwalifikowaną*. Chociaż *można* podjąć edukację informatyczną, nie jest ona konieczna. Ja takiego wykształcenia nie mam[4].

Umiejętności potrzebne w pracy inżyniera ds. oprogramowania zależą od określonej sytuacji. Trzeba poznać strukturę określonego kodu źródłowego i nauczyć się korzystania z określonej platformy. Trzeba stracić kilka dni na poprawienie błędu występującego w środowisku produkcyjnym itp.

Im dłużej pracujesz w ten sposób, tym więcej masz umiejętności. Jeśli będziesz pracować w tej samej firmie przez kilka lat, nad tym samym kodem źródłowym, możesz się stać wyspecjalizowanym autorytetem, ale czy pomoże Ci to w znalezieniu innej pracy?

Szybciej będziesz się uczyć, jeśli będziesz pracował nad kodem różnych programów. Spróbuj popracować nad oprogramowaniem backendu, a potem nad oprogramowaniem frontendu. Być może warto popracować nad grą lub nad uczeniem maszynowym. Dzięki temu poznasz szeroki zakres problemów i zdobędziesz większe doświadczenie.

To postępowanie jest łudząco podobne do dawnej europejskiej tradycji *czeladnictwa*. Rzemieślnik, na przykład stolarz lub dekarz, podróżował

[4] Jeśli jesteś ciekawy, to przyznam się, że mam wykształcenie wyższe. Jestem ekonomistą, jednak z dyplomu skorzystałem tylko wtedy, gdy ubiegałem się o pracę w duńskim Ministerstwie Gospodarki.

Kup książkę Poleć książkę

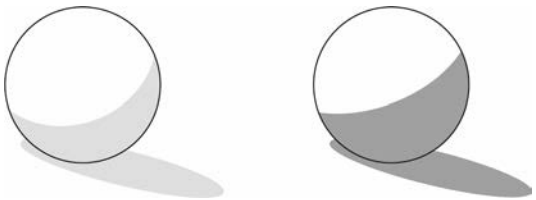
po Europie: pracował przez pewien czas w jednym miejscu, by następnie przenieść się gdzie indziej. Dzięki temu poznawał różne rozwiązania problemów i zwiększał umiejętności z zakresu swojego rzemiosła.

Takie postrzeganie programistów jest kuszące. Nawet książka *Pragmatyczny programista* ma podtytuł *Od czeladnika do mistrza* [50].

Jeśli ta perspektywa jest poprawna, powinniśmy odpowiednio ukształtować swoją branżę. Powinniśmy zatrudniać uczniów, którzy będą pracować pod okiem mistrzów. Moglibyśmy nawet organizować cechy rzemieślnicze.

Oczywiście *o ile* jest poprawna.

Rzemiosło programistyczne jest kolejną metaforą. Uważam, że jest oświecająca, ale jeśli oświetlisz obiekt jasnym światłem, pojawią się cienie. Im jaśniejsze światło, tym ciemniejszy cień, co widać na rysunku 1.1.



Rysunek 1.1. Im jaśniejsze światło pada na obiekt, tym ciemniejszy wydaje się cień

Nadal czegoś tu brakuje.

1.3.2. HEURYSTYKI

Lata spędzone na praktykowaniu rzemiosła programistycznego były dla mnie swego rodzaju okresem całkowitego rozczarowania. Uważałem, że umiejętność jest tylko nagromadzonym doświadczeniem. Wydawało mi się, że w tworzeniu programowania nie da się zastosować żadnej metodyki. Wszystko zależy od okoliczności i nie ma ani dobrego, ani złego sposobu postępowania.

Uważałem, że programowanie jest tak naprawdę sztuką.

Kup książkę Poleć książkę

Odpowiadało mi to. Zawsze lubiłem sztukę. W młodości chciałem zostać artystą[5].

To podejście jest problematyczne, ponieważ się nie skaluje. Aby „wychować" nowych programistów, trzeba ich przyjąć jako swoich uczniów i uczyć, dopóki nie zostaną czeladnikami. Następnie czeka ich jeszcze kilka lat praktyki, aż osiągną mistrzostwo.

Postrzeganie programowania jako sztuki lub rzemiosła niesie ze sobą jeszcze inny problem, ponieważ nie pasuje do rzeczywistości. Około 2010 r. zacząłem sobie uświadamiać [106], że podczas programowania przestrzegam pewnych heurystyk — praktycznych zasad i wytycznych, których można się nauczyć.

Początkowo się tym nie przejmowałem. Jednak po kilku latach zauważyłem, że często uczę innych programistów. Wtedy zwykle znajdowałem powody do pisania kodu w określony sposób.

Zacząłem sobie uświadamiać, że prawdopodobnie mój nihilizm był błędny. Możliwe, że wytyczne mogą być kluczem do postrzegania programowania jako dyscypliny inżynieryjnej.

1.3.3. WCZEŚNIEJSZE POGLĄDY NA INŻYNIERIĘ OPROGRAMOWANIA

Pojęcie inżynierii oprogramowania zaczęto stosować pod koniec lat 60. XX w.[6] Dotyczyło ono ówczesnego *kryzysu oprogramowania*, który zapanował, gdy się okazało, że programowanie jest *trudne*.

Ówcześni programiści w rzeczywistości dobrze wiedzieli, co robią. Działało wtedy wielu prominentnych przedstawicieli naszej branży: Edsger Dijkstra, Tony Hoare, Donald Knuth, Alan Kay. Gdybyśmy zapytali ludzi w tamtych

[5] Najpierw chciałem zostać autorem komiksów tworzącym dzieła w europejskim stylu. Później, gdy byłem nastolatkiem, zacząłem grać na gitarze i zapragnąłem zostać gwiazdą rocka. Okazało się, że chociaż lubiłem rysować i grać, nie byłem wystarczająco utalentowany.

[6] Ten termin może być starszy. Nie jestem tego pewien, a ponieważ nie było mnie wtedy na świecie, nie mogę sobie tego przypomnieć. Jednak termin inżynieria oprogramowania spopularyzowano na dwóch konferencjach NATO, które odbyły się w latach 1968 i 1969 [4].

Kup książkę Poleć książkę

czasach, czy uważają, że programowanie będzie dyscypliną inżynierską w latach 20. XXI w., prawdopodobnie odpowiedzieliby twierdząco.

Być może zauważyłeś, że omawiam pogląd na inżynierię oprogramowania raczej jako dążenie do osiągnięcia celu, a nie jako fakt ze świata tworzenia oprogramowania. Prawdopodobnie na świecie istnieją enklawy prawdziwej inżynierii oprogramowania[7], ale z doświadczenia wiem, że tworzenie oprogramowania najczęściej odbywa się inaczej.

Nie jestem osamotniony w poczuciu, że inżynieria oprogramowania nadal jest przyszłym celem. Pięknie ujął to Adam Barr:

> *Jeśli myślisz jak ja, marzysz o dniu, gdy inżynieria oprogramowania będzie przedmiotem przemyślanych, metodycznych studiów, a wytyczne dla programistów będą wynikiem eksperymentów i nie będą się opierać na doświadczeniu indywidualnych osób. [4]*

Wyjaśnia, że inżynieria oprogramowania świetnie się rozwijała, ale w pewnym momencie coś zakłóciło jej rozwój. Według Barra było to nadejście komputerów osobistych. Dzięki nim pojawiły się generacje programistów, którzy sami nauczyli się programować. Ponieważ mogli korzystać z komputerów w samotności, w dużym stopniu ignorowali istniejącą już wiedzę.

Wydaje się, że ta sytuacja utrzymuje się do dziś. Alan Kay nazywa korzystanie z komputerów *kulturą pop*:

> *Jednak kultura pop lekceważy historię. Kultura pop skupia się na tożsamości i poczuciu uczestnictwa. Nie ma nic wspólnego ze współpracą, przeszłością czy przyszłością — jest to życie teraźniejszością. Wydaje mi się, że tak samo postępuje większość ludzi piszących kod za pieniądze. Nie mają pojęcia, skąd [pochodzi ich kultura]. [52]*

Być może zmarnowaliśmy 50 lat, kiedy to inżynieria oprogramowania rozwinęła się tak nieznacznie, ale uważam, że poczyniliśmy postęp na innych polach.

[7] NASA wydaje się dobrym przykładem jednej z nich.

Kup książkę Poleć książkę

1.3.4. KU INŻYNIERII OPROGRAMOWANIA

Czym zajmuje się inżynier? Inżynierowie projektują i nadzorują konstrukcję rzeczy, od wielkich struktur, takich jak mosty, tunele, wieżowce i elektrownie, po małe obiekty, takie jak mikroprocesory[8]. Pomagają w produkcji obiektów fizycznych.

Most Królowej Aleksandry, zwany potocznie Mønbroen. Ukończony w 1943 r., łączy Zelandię z mniejszą wyspą Møn w Danii

Programiści tego nie robią. Oprogramowanie jest niematerialne. Jack Reeves stwierdził [87], że ponieważ nie produkują żadnych fizycznych obiektów, konstrukcja w rzeczywistości nic nie kosztuje. Tworzenie oprogramowania jest przede wszystkim projektowaniem. Wpisywanie kodu w edytorze odpowiada rysowaniu planów przez inżynierów, a nie konstruowaniu rzeczy przez pracowników fizycznych.

„Prawdziwi" inżynierowie postępują według metodyk, co zwykle prowadzi do powstania udanych wyników. To samo chcą osiągnąć programiści,

[8] Miałem kiedyś przyjaciela, który był z wykształcenia inżynierem chemikiem. Po ukończeniu uniwersytetu rozpoczął pracę jako piwowar w firmie Carlsberg. Inżynierowie również warzą piwo.

Kup książkę Poleć książkę

ale muszą naśladować tylko te czynności, które są uzasadnione w ich pracy. W pracy nad powstaniem obiektu fizycznego jego konstruowanie jest kosztowne. Nie można podjąć próby zbudowania mostu w formie eksperymentów, a na koniec stwierdzić, że nie jest dobry, zburzyć go i zacząć od nowa. Ponieważ rzeczywiste konstrukcje są kosztowne, inżynierowie przeprowadzają wiele obliczeń i symulacji. Obliczenie siły nośnej mostu wymaga mniej czasu i materiałów niż jego zbudowanie.

Istnieje cała dyscyplina inżynieryjna dotycząca logistyki. Ludzie drobiazgowo planują, ponieważ jest to bezpieczniejszy i znacznie tańszy sposób na budowanie obiektów fizycznych.

Tego aspektu inżynierii *nie* trzeba kopiować.

Inspirację możemy natomiast czerpać z wielu innych metodyk inżynieryjnych. Inżynierowie wykonują również kreatywną pracę, ale zwykle odbywa się ona w ramach platformy. Po pewnych czynnościach należy wykonać inne. Dokonują inspekcji i zatwierdzają pracę innych. Posługują się listami kontrolnymi [40].

Ty też możesz to robić.

O tym właśnie jest ta książka. Jest ona przewodnikiem po heurystykach, które według mnie są przydatne. Obawiam się, że są one bliższe zjawisku, które Adam Barr nazywa *grząskim gruntem osobistego doświadczenia*, niż zbiorowi reguł, który ma poparcie w nauce.

Wierzę, że jest to odzwierciedlenie bieżącego stanu naszej branży. Każdy, kto uważa, że istnieją twarde dowody naukowe na wszystko, powinien przeczytać książkę *The Leprechauns of Software Engineering* [13].

1.4. WNIOSEK

Jeśli przeanalizujesz historię tworzenia oprogramowania, prawdopodobnie uznasz, że dokonał się olbrzymi postęp. A jednak w dużej mierze dotyczy on sprzętu, a nie oprogramowania. Mimo to przez ostatnie 50 lat byliśmy świadkami ogromnego postępu w rozwijaniu oprogramowania.

Kup książkę Poleć książkę

Obecnie mamy do dyspozycji znacznie bardziej zaawansowane języki programowania niż 50 lat temu, dostęp do internetu (włącznie z internetową pomocą w postaci serwisu Stack Overflow), programowanie zorientowane obiektowo i funkcyjne, platformy do testowania automatycznego, Git, zintegrowane środowiska programistyczne itd.

Z drugiej strony nadal zmagamy się z *kryzysem oprogramowania*, chociaż można się spierać, czy można nazwać kryzysem sytuację trwającą ponad połowę stulecia.

Niezależnie od poważnych wysiłków branża tworzenia oprogramowania nadal nie przypomina dyscypliny inżynieryjnej. Między inżynierią a programowaniem istnieją fundamentalne różnice. Dopóki ich nie zrozumiemy, nie będziemy czynić postępów.

Dobrą stroną tej sytuacji jest możliwość stosowania wielu praktyk skopiowanych od inżynierów. Można przyjąć ich sposób myślenia oraz wiele różnych procesów.

William Gibson, autor literatury science fiction, napisał:

Przyszłość już tu jest — tylko nie jest równomiernie rozłożona[9].

Zgodnie z książką *Przyspieszenie* niektóre organizacje wykorzystują obecnie zaawansowane techniki, a inne pozostają w tyle [29]. Przyszłość rzeczywiście jest nierównomiernie rozłożona. Dobra wiadomość jest taka, że zaawansowane pomysły można wykorzystać za darmo. Ty możesz zdecydować o ich użyciu.

W rozdziale 2. poznasz wybrane konkretne czynności, które możesz wykonać.

[9] Jest to jeden z cytatów, których pochodzenie jest niejasne. Najprawdopodobniej tę ideę oraz jej ogólne sformułowanie zawdzięczamy Gibsonowi, ale nie wiadomo, kiedy dokładnie ją wygłosił [76].

Kup książkę Poleć książkę

Kup książkę Poleć książkę


PROGRAM
PARTNERSKI
GRUPY HELION
1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ
Zmień swoją stronę WWW w działający bankomat!
Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl
GRUPA
Helion

Celem nie jest pisanie kodu. Celem jest funkcjonalne oprogramowanie!

Wielu zdolnych programistów uczestniczy w nieefektywnych, źle prowadzonych projektach. Inni muszą utrzymywać kod, który właściwie nigdy nie powinien ujrzeć światła dziennego. Źródeł tego rodzaju trudności jest wiele: programowanie samo w sobie jest niełatwym zagadnieniem, a tworzenie funkcjonalnych aplikacji najczęściej wymaga współdziałania wielu osób. Sprawia to, że kod budujący aplikację szybko zwiększa swoją złożoność, staje się niezrozumiały i bardzo trudny w utrzymaniu. Na szczęście te problemy zostały dostrzeżone i obecnie mamy wiele sposobów ich rozwiązywania.

Ta książka jest przeznaczona dla doświadczonych programistów, którzy chcą zdobyć lub pogłębić wiedzę o metodologiach i dobrych praktykach tworzenia złożonego oprogramowania. Stanowi interesująco napisany zbiór heurystyk i technik ułożonych w kolejności stosowania podczas rozwijania projektu programistycznego. Omówiono tu między innymi listy kontrolne, TDD, rozdzielenie poleceń i zapytań, Git, złożoność cyklomatyczną, przezroczystość referencyjną, wycinki pionowe, refaktoryzację zastanego kodu czy programowanie typu outside-in. Pokazano również, jak utrzymać właściwe tempo pracy podczas dodawania funkcjonalności, jak rozwiązywać problemy optymalizacji, enkapsulacji, a także projektowania API i testów jednostkowych. Poszczególne zagadnienia zostały zilustrowane kodem przykładowego projektu napisanego w C#, zrozumiałego dla każdego, kto zna dowolny język zorientowany obiektowo.

Dzięki książce zrozumiesz, jak:

- **wybierać sprawdzone procesy**
- **tworzyć listy kontrolne ułatwiające polepszenie wyników**
- **unikać „paraliżu analitycznego"**
- **przeciwdziałać degradacji kodu i niepotrzebnej złożoności**
- **stosować lepsze techniki modyfikacji kodu i rozwiązywania problemów**
- **skuteczniej godzić wymogi wydajności i bezpieczeństwa**

Mark Seemann marzył o karierze gwiazdy rocka i próbował zostać ekonomistą, jego prawdziwym powołaniem jednak okazało się programowanie aplikacji internetowych i biznesowych. Jest certyfikowanym programistą Rockstara, autorem nagradzanych książek o programowaniu i prelegentem na prestiżowych konferencjach. Mieszka w Kopenhadze z żoną i dwójką dzieci.

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 978-83-283-9226-7

Cena: 99,00 zł

Pearson
Addison-Wesley