

Podręcznik profesjonalnego programisty!

# Mistrz czystego kodu

Kodeks postępowania profesjonalnych programistów

Tytuł oryginału: The Clean Coder: A Code of Conduct for Professional Programmers

Łąmaczenie: Wojciech Moch

ISBN: 978-83-246-7537-1

Authorized translation from the English language edition, entitled: THE CLEAN CODER: A CODE OF CONDUCT FOR PROFESSIONAL PROGRAMMERS; ISBN 0137081073;  
by Robert C. Martin; published by Pearson Education, Inc, publishing as Prentice Hall.  
Copyright © 2011 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 © 2013.

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 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 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](mailto: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/mckcod\\_ebook](http://helion.pl/user/opinie/mckcod_ebook)  
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- [Poleć książkę na Facebook.com](#)
- [Kup w wersji papierowej](#)
- [Oceń książkę](#)

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

## Pochwały dla Mistrza czystego kodu

Wujek Bob Martin w swojej nowej książce zdecydowanie podnosi poprzeczkę. Opisuje swoje oczekiwania wobec zawodowych programistów w zakresie interakcji z kadrą zarządzającą, zarządzania czasem, radzenia sobie z presją, współpracy w zespole oraz doboru narzędzi. Martin zaznacza, że każdy programista chcący uznawać się za profesjonalistę powinien znać techniki wykraczające poza TDD oraz ATDD, a dodatkowo powinien ściśle śledzić ich rozwój, tak żeby podsycać swoją pasję do programowania.

— Markus Gärtner  
Senior Software Developer  
it-agile GmbH  
[www.it-agile.de](http://www.it-agile.de)  
[www.shino.de](http://www.shino.de)

Niektóre książki techniczne nie tylko uczą, ale też inspirują. Inne zachwycają i bawią. Tylko rzadko zdarza się, żeby książka techniczna miała te wszystkie cechy. Książki Roberta Martina zawsze na mnie działały, a *Mistrz czystego kodu* nie jest tu wyjątkiem. Czytaj, ucz się, poznawaj lekcje zawarte w książce, a z pewnością uznasz się za zawodowego programistę.

— George Bullock  
Senior Program Manager  
Microsoft Corp.

Jeżeli do uzyskania tytułu nauk komputerowych konieczne byłoby przeczytanie pewnej lektury, to na pewno byłaby to ta książka. W rzeczywistym świecie zły kod nie znika po zakończeniu semestru, nie dostaje się piątki za maraton kodowania na dzień przed oddaniem projektu, a najgorsze jest to, że trzeba kontaktować się z innymi ludźmi. Oznacza to, że guru programowania niekoniecznie można nazwać zawodowcami. *Mistrz czystego kodu* opisuje podróż w kierunku profesjonalizmu... a robi to w wyjątkowo zabawny sposób.

— Jeff Overby  
Uniwersytet stanu Illinois w Urbana-Champaign

*Mistrz czystego kodu* jest czymś więcej niż tylko zbiorem zasad i wskazówek. Zawiera mądrość popartą praktyką oraz wiedzą, którą normalnie trzeba zdobywać przez lata prób i błędów, pracując jako uczeń mistrza. Jeżeli nazywasz siebie zawodowym programistą, to ta książka jest dla Ciebie niezbędna.

— R.L. Bogetti  
Lead System Designer  
Baxter Healthcare  
[www.RLBogetti.com](http://www.RLBogetti.com)



W latach od 1986 do 2000 ściśle współpracowałem z Jimem Newkirkiem, kolegą z firmy Teradyne. Mieliśmy wspólną pasję do programowania i tworzenia czystego kodu. Razem spędzaliśmy całe wieczory i noce, a nawet weekendy, bawiąc się różnymi stylami programowania oraz technikami projektowymi. Bez ustanku zastanawialiśmy się nad różnymi pomysłami na biznes. W końcu założyliśmy firmę Object Mentor Inc. Podczas wspólnych prac nauczyłem się od Jima wielu rzeczy. Jednak jego najważniejszą cechą była *etyka pracy*. To coś, nad czym staram się cały czas pracować. Jim jest zawodowcem, a ja jestem dumny, że mogłem razem z nim pracować i nazywać go moim przyjacielem.



---

# **SPIS TREŚCI**

---

<b>Słowo wstępne</b>	<b>11</b>
<b>Wprowadzenie</b>	<b>17</b>
<b>Podziękowania</b>	<b>21</b>
<b>O autorze</b>	<b>25</b>
<b>Na okładce</b>	<b>27</b>
<b>Obowiązkowy wstęp</b>	<b>29</b>
<b>Rozdział 1 Profesjonalizm</b>	<b>35</b>
Uważaj, czego sobie życzysz	36
Przejmowanie odpowiedzialności	36
Po pierwsze nie szkodzić	38
Etyka pracy	43
Bibliografia	48
<b>Rozdział 2 Kiedy mówić „nie”</b>	<b>49</b>
Przeciwstawne role	51
Wysokie stawki	54
Gracz zespołowy	55
Koszta przytakiwania	60
Kod niemożliwy	65

<b>Rozdział 3</b>	<b>Kiedy mówić „tak”</b>	<b>69</b>
	Język zobowiązań	71
	Naucz się, jak mówić „tak”	75
	Wnioski	78
<b>Rozdział 4</b>	<b>Kodowanie</b>	<b>79</b>
	Przygotowanie	80
	Strefa	83
	Blokada twórcza	85
	Debugowanie	87
	Wyznaczanie sobie rytmu	90
	Spóźnienia	91
	Pomoc	93
	Bibliografia	95
<b>Rozdział 5</b>	<b>TDD</b>	<b>97</b>
	Sąd na sali	98
	Trzy prawa TDD	99
	Czym TDD nie jest	103
	Bibliografia	103
<b>Rozdział 6</b>	<b>Ćwiczenia</b>	<b>105</b>
	Kilka ćwiczeń w tle	106
	Dojo kodowania	109
	Zwiększenie doświadczenia	112
	Wnioski	113
	Bibliografia	113
<b>Rozdział 7</b>	<b>Testy akceptacyjne</b>	<b>115</b>
	Komunikowanie wymagań	115
	Testy akceptacyjne	120
	Wnioski	129
<b>Rozdział 8</b>	<b>Strategie testowania</b>	<b>131</b>
	Kontrola jakości nie powinna nic znaleźć	132
	Piramida automatyzacji testów	133
	Wnioski	136
	Bibliografia	136

---

<b>Rozdział 9</b>	<b>Zarządzanie czasem</b>	<b>137</b>
	Spotkania	138
	Manna skupienia	142
	Paczkowanie czasu i pomidory	144
	Uniki	145
	Ślepe uliczki	146
	Marsze, bagna i bałagan	146
	Wnioski	147
<b>Rozdział 10</b>	<b>Szacowanie</b>	<b>149</b>
	Czym jest szacowanie?	151
	PERT	154
	Szacowanie zadań	157
	Prawo wielkich liczb	159
	Wnioski	160
	Bibliografia	160
<b>Rozdział 11</b>	<b>Presja</b>	<b>161</b>
	Unikanie presji	163
	Jak radzić sobie z presją	165
	Wnioski	166
<b>Rozdział 12</b>	<b>Współpraca</b>	<b>167</b>
	Programiści kontra ludzie	169
	Mózdzki	173
	Wnioski	174
<b>Rozdział 13</b>	<b>Zespoły i projekty</b>	<b>175</b>
	Można to zmiksować?	176
	Wnioski	178
	Bibliografia	179
<b>Rozdział 14</b>	<b>Nauczanie, terminowanie i mistrzostwo</b>	<b>181</b>
	Stopnie niepowodzenia	181
	Nauczanie	182
	Terminowanie	187
	Rzemiosło	190
	Wnioski	191

---

<b>Dodatek A</b>	<b>Narzędzia</b>	<b>193</b>
	Narzędzia	195
	Kontrola kodu źródłowego	195
	IDE i edytor	199
	Śledzenie problemów	201
	Ciągła komplikacja	202
	Narzędzia do testów jednostkowych	202
	Narzędzia do testów komponentów	203
	Narzędzia do testów integracyjnych	204
	UML/MDA	205
	Wnioski	207
<b>Skorowidz</b>		<b>209</b>

---

# SŁOWO WSTĘPNE

---

Skoro ta książka wzbudziła w Tobie zainteresowanie, to zakładam, że mam do czynienia z zawodowcem. To dobrze, bo i ja się za takiego uważam. A skoro zdobyłem już Twoje zainteresowanie, to pozwól, że powiem, dlaczego ja zająłem się tą książką.

Wszystko zaczęło się nie tak dawno temu, w całkiem nieodległym miejscu. Kurtyna w góre, światła, kamery i Charley...

Kilka lat temu pracowałem w średniej wielkości korporacji sprzedającej produkty podlegające ściszej regulacji. Z pewnością znasz ten typ: siedzieliśmy w przedziałach, w trzypiętrowym budynku. Dyrektorzy i tym podobni mieli prywatne gabinety, a zebranie wszystkich osób potrzebnych na jednym spotkaniu trwało mniej więcej tydzień.

Działaliśmy na rynku z bardzo dużą konkurencją, a wtedy rząd otworzył drzwi dla całkowicie nowego produktu.

Nagle stanęła przed nami cała rzesza nowych, potencjalnych klientów. Wystarczyło tylko przekonać ich do zakupu naszego produktu. Oznaczało to, że musielibyśmy złożyć we właściwym czasie i w odpowiednim urzędzie pewien dokument, następnie w określonym terminie przejść pozytywnie audyt i szybko wprowadzić produkt na rynek.

Od czasu do czasu kadra zarządzająca przypominała nam o tym, jak ważne są poszczególne daty. Jeden poślizg i urzędnicy na rok wykluczą nas z rynku, a jeżeli klienci nie będą mogli skorzystać z naszego produktu już od samego początku, to pójdą do konkurencji, przez co całkowicie wypadniemy z obiegu.

Powstało to specyficzne środowisko, w którym część ludzi narzeka, a inni powtarzają tylko, że „pod ciśnieniem powstają diamenty”.

Byłem technicznym menedżerem projektu, awansowanym z działu rozwojowego. Moim zadaniem było uruchomienie strony projektu w dniu jego oficjalnego udostępnienia, tak żeby potencjalni klienci mogli pobrać niezbędne informacje, a przede wszystkim formularze zamówień. Moim współpracownikiem w ramach tego projektu był menedżer biznesowy, którego tutaj będę nazywał Joe. Jego zadaniem była praca po drugiej stronie, czyli obsługa sprzedaży, marketingu oraz wymagań niezwiązanych z technikaliami. Był też jednym z tych powtarzających frazę o „diamentach powstających pod ciśnieniem”.

Jeżeli zdarzyło Ci się pracować w amerykańskich korporacjach, to zapewne wiesz, że to całe wskazywanie palcem, poszukiwanie winnych i niechęć do pracy są całkowicie naturalne. W naszej firmie Joe i ja znaleźliśmy całkiem ciekawe rozwiązanie tego problemu.

Naszym zadaniem było wykonanie roboty i czuliśmy się z tym zupełnie jak Batman i Robin. Codziennie spotykałem się z zespołem technicznym, za każdym razem poprawialiśmy plan prac, wypatrywaliśmy potencjalnych problemów i usuwaliśmy z naszej drogi wszystkie powstające przeszkody. Jeżeli ktoś potrzebował określonego oprogramowania, to je zdobywaliśmy. Jeżeli ktoś „z chęcią” zająłby się konfiguracją zapory internetowej, ale „o rany, właśnie mam przerwę obiadową”, to kupowaliśmy i dostarczaliśmy cały obiad. Jeżeli ktoś chciał popracować nad problemem z konfiguracją, ale miał aktualnie inne priorytety, to Joe i ja zaczynaliśmy rozmawiać z jego przełożonym.

A potem z menedżerem.

A potem z dyrektorem.

Po prostu wszystko załatwialiśmy.

Może pewną przesadą jest twierdzenie, że przewracaliśmy krzesła, krzyczeliśmy i wrzeszczaliśmy, ale dla osiągnięcia celu wykorzystaliśmy wszystkie znane nam metody, przy okazji wymyślając kilka nowych. Wszystko, co robiliśmy, robiliśmy zgodnie z zasadami etyki, z czego jestem dumny do dzisiaj.

Uważyłem się za członka zespołu. Nie byłem ponad to, żeby samodzielnie napisać wymagane zapytanie SQL albo wykonać inne prace, byle tylko przygotować kod na czas. Wtedy podobnie myślałem i o Joe — jako o członku zespołu, a nie kimś ponad nami.

Ostatecznie przekonałem się, że Joe nie podzielał tej opinii. To był dla mnie bardzo smutny dzień.

Był piątek, godzina 13.00, a tworzona przez nas strona była gotowa do uruchomienia w następny poniedziałek.

Byliśmy gotowi. **GOTOWI**. Każdy system był sprawdzony i przygotowany. Zebrałem cały zespół w ramach ostatniego spotkania scrumu, na którym gratulowaliśmy sobie doskonałej

pracy. I nie chodziło tylko o zespół „techniczny”. Towarzyszyły nam też osoby z działu marketingu, a nawet właściciele produktu.

Wszyscy byliśmy dumni. To był naprawdę wspaniały moment.

Wtedy na zebranie wpadł Joe.

Powiedział coś w rodzaju: „Złe wieści. Dział prawny nie przygotował jeszcze formularzy zamówień, czyli nie możemy jeszcze uruchomić strony”.

Nie był to wielki problem. Przez cały czas trwania projektu wstrzymywały nas najróżniejsze przeciwności, dzięki czemu całkiem dobrze odnajdowaliśmy się w rolach Batmana i Robina. Byłem gotowy, dlatego odpowiedziałem: „Dobra, partnerze. Zrobimy to jeszcze raz. Dział prawny jest na trzecim piętrze, o ile pamiętam”.

I wtedy się zaczęło.

Zamiast się ze mną zgodzić, Joe zapytał: „O czym ty w ogóle mówisz, Matt?”.

Odpowiedziałem: „No wiesz, ten nasz zwyczajowy taniec. W końcu chodzi o cztery pliki PDF. Do tego one są gotowe. Dział prawny musi tylko je zatwierdzić. Powisamy w ich boksach, położącmy się na nich i w końcu to załatwią”.

Joe nie zgodził się z moją oceną i po prostu stwierdził: „Uruchomimy stronę w ciągu przyszłego tygodnia. To przecież żaden kłopot”.

Z pewnością możesz sobie wyobrazić ciąg dalszy tej rozmowy. Szło mniej więcej tak:

Matt: „A niby dlaczego? Przecież mogą to zrobić w parę godzin”.

Joe: „Oj, to może potrwać trochę dłużej”.

Matt: „Przecież mają cały weekend. To masa czasu. Do roboty!”.

Joe: „Matt, to przecież zawodowcy. Nie możemy przecież wisieć nad nimi i żądać, żeby poświęcali swój prywatny czas dla naszego małego projektu”.

Matt: (chwila ciszy) „Joe... a co niby robiliśmy z zespołem inżynierów przez kilka ostatnich miesięcy?”.

Joe: „No tak, ale tutaj mówimy o zawodowcach”.

Cisza.

Głęboki oddech.

Co. Joe. Właśnie. Powiedział?

W tym czasie uważałem, że zespół techniczny składał się z zawodowców w najlepszym znaczeniu tego słowa.

Teraz gdy o tym myślę, wcale nie jestem tego pewien.

Przyjrzyjmy się jeszcze raz technice Batmana i Robina, przyjmując tym razem inną perspektywę. Sądziłem, że w ten sposób motywujemy zespół do jeszcze lepszej pracy, ale teraz myślę, że Joe bawił się z nami i zakładał, że zespół techniczny jest jego przeciwnikiem. Wystarczy pomyśleć: Dlaczego konieczne były ciągłe bieganie, przewracanie krzeseł i inne działania?

Czy nie mogliśmy po prostu zapytać zespołu, kiedy będą gotowi, otrzymać solidnej odpowiedzi, w którą moglibyśmy uwierzyć, i się na niej nie sparzyć?

Oczywiście w przypadku zawodowców moglibyśmy... a jednocześnie wcale nie. Joe nie dowierzał naszym zapewnieniom i dlatego uznał, że konieczne będzie mikrozarządzanie całym zespołem technicznym. Równocześnie z tych samych powodów był w stanie zaufać działowi prawnemu i wcale nie chciał stosować wobec nich mikrozarządzania.

O co tu chodzi?

W jakiś sposób dział prawny zademonstrował swój profesjonalizm, czego nie zdołał zrobić zespół techniczny.

Innej grupie udało się przekonać Joeego, że niepotrzebny jest jej nadzorca, że nie bawią się w takie gierki i że należy ich traktować jak godnych szacunku współpracowników.

Nie sądzę, żeby miały tu znaczenie te wszystkie certyfikaty wiszące na ścianach ani kilka dodatkowych lat studiów. Choć to ostatnie mogło pomóc w uzyskaniu doświadczenia i umiejętności właściwego zachowania się w takich sytuacjach.

Od tego dnia, a było to już lata temu, zastanawiałem się, jak będą się musiały zmienić zawody techniczne, żeby w końcu ich przedstawicieli uznano za zawodowców.

I rzeczywiście wpadło mi kilka pomysłów. Trochę blogowałem, sporo czytałem, zdołałem poprawić własną sytuację w pracy, a nawet pomóc kilku innym osobom. Nie znam jednak żadnej książki, która szkicowałaby czytelnikowi określony plan i jawnie prezentowała poszczególne elementy.

W końcu któregoś dnia, całkowicie niespodziewanie, otrzymałem ofertę recenzowania wczesnego szkicu książki. Dokładnie tej, którą właśnie trzymasz w ręku.

To z tej książki dowieš się, jak prezentować się i współpracować jak na zawodowca przystało. Bez stosowania tanich sztuczek, bez wykorzystywania haków, ale właśnie tak, jak należy to robić.

W niektórych przypadkach takie przykłady są absolutnie dosłowne.

Część przykładów zawiera odpowiedzi, kontry, wyjaśnienia, a nawet porady na wypadek, gdyby ta druga osoba próbowała Cię „po prostu ignorować”.

Och, popatrz na to, Joe wraca na scenę, choć tym razem coś się zmienia: znowu jesteśmy w wielkiej firmie, Joe i ja, i znów pracujemy nad wielkim projektem konwersji strony WWW.

Jednak tym razem wyobraź sobie nieco inną sytuację.

Zamiast unikać zobowiązań, zespół techniczny faktycznie je podejmuje. Zamiast przemilczać szacunki albo pozwalać na przygotowywanie planów przez innych (a potem narzekać na te plany), zespół techniczny sam się organizuje i definiuje rzeczywiste zobowiązania.

Wyobraź sobie, że taki zespół rzeczywiście ze sobą współpracuje. Jeżeli infrastruktura „wytrzymuje” programistów, to wystarczy jeden telefon, żeby administratorzy wykonali niezbędne prace.

Gdy Joe zamierza przyspieszyć prace nad błędem 14321, to okazuje się, że nie ma takiej potrzeby, ponieważ od razu widzi, że administrator bazy danych nad nim pracuje, a nie surfuje w sieci. Podobnie wszystkie szacunki, jakie otrzymał od zespołu, nie są ze sobą sprzeczne i nie tworzą wrażenia, że cały projekt otrzymał priorytet gdzieś między obiadem a sprawdzaniem poczty. Na wszelkie próby manipulowania przygotowanym planem zespół nie odpowiada „spróbujemy”, ale „to są nasze zobowiązania, jeżeli chcesz tworzyć własne, to rób je we własnym zakresie”.

Sądzę, że po pewnym czasie Joe zacząłby uważać zespół techniczny za, zaskoczenie, zawodowców. I miałby całkowitą rację.

Co trzeba zrobić, żeby przekształcić się ze zwykłego technika w zawodowca? Wszystko znajdziesz w tej książce.

Witaj na kolejnym etapie swojej kariery. Wydaje mi się, że będzie Ci się podobać.

— Matthew Heusser  
Software Process Naturalist



---

# Wprowadzenie

---



28 stycznia 1986 roku, o godzinie 11.39, po zaledwie 73,124 sekundy od startu, na wysokości 48 000 stóp prom kosmiczny Challenger został rozerwany na strzępy przez uszkodzenie prawego silnika wspomagającego. Zginęło siedmioro dzielnych astronautów, w tym nauczycielka Christa McAuliffe. Do dzisiaj prześladuje mnie wyraz twarzy jej matki obserwującej śmierć swojej córki 15 kilometrów nad sobą.

Challenger rozpadł się, ponieważ gorące gazy wylotowe z uszkodzonego silnika wspomagającego wydostały się spomiędzy segmentów jego obudowy, trafiając na główny zbiornik paliwa. Podstawa głównego zbiornika z ciekłym wodorem została rozerwana, paliwo się zapaliło, przez co zbiornik przesunął się w kierunku zbiornika z ciekłym tlenem. W tym samym momencie silnik wspomagający zerwał się z tylnego wspornika i obrócił wokół tylnego. Jego czubek przebił zbiornik z ciekłym tlenem. Te wszystkie

nieskoordynowane siły spowodowały, że lecący z prędkością 1,5 macha prom zaczął obracać się w poprzek strumienia powietrza. Siły aerodynamiczne szybko rozerwały całość na kawałeczki.

Pomiędzy koncentrycznymi segmentami silnika znajdowały się dwie okrągłe uszczelki z syntetycznej gumy. Gdy segmenty były łączone ze sobą, uszczelki zostały ściśnięte, tworząc uszczelnienie, którego gazy wylotowe nie powinny być w stanie przebić.

Niestety, w noc poprzedzającą start temperatura na wyrzutni spadła do  $-8^{\circ}\text{C}$  ( $17^{\circ}\text{F}$ ), czyli o  $12^{\circ}\text{C}$  ( $23^{\circ}\text{F}$ ) poniżej minimalnej temperatury dla tych uszczelk. Było to też o całych  $18^{\circ}\text{C}$  ( $33^{\circ}\text{F}$ ) mniej niż przy którymkolwiek wcześniejszym startie. W efekcie uszczelki zeszytniwały i nie były w stanie odpowiednio blokować gorących gazów. W momencie odpalenia silnika wspomagającego nastąpił nagły wzrost ciśnienia wynikający z szybkiego gromadzenia się gazów wylotowych. Segmenty silnika zostały rozcięgięte, co zmniejszyło stopień ściśnięcia uszczelk, które były na tyle zeszytniowe, że nie zdołały się odpowiednio rozprężyć i wypełnić powstały szczelin. W efekcie część uciekających gorących gazów odparowała uszczelki na długości 70% łuku.

Inżynierowie z firmy Morton-Thiokol, którzy projektowali silniki wspomagające, wiedzieli o problemach z uszczelkami i przez siedem poprzednich lat informowali o nich kadrę zarządzającą firmą oraz NASA. Rzeczywiście uszczelki używane w poprzednich startach zostały uszkodzone na kilka różnych sposobów, choć żadne z tych uszkodzeń nie mogło doprowadzić do katastrofy. Podczas startu w najniższej temperaturze zostały jednak uszkodzone najmocniej. Inżynierowie przygotowali rozwiązanie tego problemu, ale jego wprowadzenie zostało mocno opóźnione.

Inżynierowie podejrzewali, że uszczelki sztywniały pod wpływem zimna. Wiedzieli też, że temperatury panujące podczas startu Challengera były niższe niż podczas któregokolwiek wcześniejszego startu — znacznie poniżej temperatury dopuszczalnej. W skrócie: inżynierowie wiedzieli, że ryzyko jest zbyt wielkie i wykorzystując tę wiedzę, zaczęli odpowiednio działać. Pisali notatki i zapalali światła alarmowe. Nastawiali na menedżerów Thiokolu i NASA, żeby opóźnić start. Na jedenastogodzinnym spotkaniu tuż przed startem zaprezentowali wszystkim posiadane dane. Wściekali się, przymilali, komu trzeba, i protestowali. Ostatecznie jednak menedżerowie ich zignorowali.

Część inżynierów nie chciała nawet oglądać przekazów ze startu wahadłowca, ponieważ obawiali się, że wybuchnie jeszcze na platformie. Jednak Challenger zgrabnie wzniósł się w niebo, przez co napięcie zaczęło opadać. Na kilka chwil przed zniszczeniem pojazdu, w momencie gdy przekraczał on prędkość 1 macha, jeden z inżynierów stwierdził, że „tym razem się udało”.

Mimo wszystkich protestów, notatek i uwag zgłaszanych przez inżynierów menedżerowie uznali, że to oni mają rację. Sądzili, że inżynierowie najzwyczajniej przesadzają, dlatego

nie ufali ich danym oraz wnioskom. Uruchomili silniki Challengera, ponieważ sami byli pod wielką presją finansową i polityczną. Mieli tylko *nadzieję*, że wszystko się uda.

Wszyscy ci menedżerowie nie tylko byli głupi, ale stali się przestępczami. Tego mroźnego poranka zginęło siedmioro dzielnych kobiet i mężczyzn, a nadzieje całych pokoleń marzących o podrózach w kosmos zostały zawiedzione dlatego, że kilku menedżerów przedłożyło własne obawy, nadzieję i przeczucia nad słowa zatrudnianych przez siebie ekspertów. Podjęli decyzję, której nie mieli prawa podejmować. Przypisali sobie autorytet ludzi, którzy *wiedzieli*, co robić: inżynierów.

A co z inżynierami? Z pewnością zrobili to wszystko, co powinni byli zrobić. Poinformowali przełożonych i ciężko walczyli, prezentując swoje zalecenia. Przeszli przez wszystkie właściwe kanały i powołali się na odpowiednie protokoły. Zrobili zatem, co mogli w *ramach* istniejącego systemu, ale menedżerowie i tak postawili na swoim. Wygląda na to, że inżynierowie mogą mieć czyste sumienie.

Czasami jednak zastanawiam się, czy któryś z nich nie może spać w nocy, prześladowany przez obraz matki Christy McAuliffe, rozważając, dlaczego nie zadzwonił do Dana Rothera.

## O tej książce

To książka o profesjonalizmie przy tworzeniu oprogramowania. Zawiera wiele pragmatycznych wskazówek i próbuje w ten sposób odpowiedzieć na takie pytania, jak:

- Kim jest zawodowy programista?
- Jak zachowuje się profesjonalista?
- Jak zawodowiec radzi sobie z konfliktami, krótkimi terminami i nierozsądnymi menedżerami?
- Kiedy i jak profesjonalista powinien powiedzieć „nie”?
- Jak zawodowiec radzi sobie z naciskami?

W tych pragmatycznych poradach znajdziesz też rodzącą się postawę. Jest to postawa pełna godności, honoru, poszanowania oraz dumy. Jest w niej też gotowość do zaakceptowania tego straszliwego zobowiązania związanego z byciem rzemieślnikiem i inżynierem. Częścią tego zobowiązania jest konieczność wykonywania czystej i dobrej pracy, a także utrzymania dobrej komunikacji i dokonywania prawidłowych szacunków. Dotyczy ono też zarządzania swoim czasem i mierzenia się ze skomplikowanymi decyzjami typu ryzyko – nagroda.

Do tego zobowiązania należy też jeszcze jedna, niezwykle przerażająca rzecz. Jako inżynier dysponujesz dogłębną wiedzą o swoich systemach i projektach, która nie jest dostępna dla menedżerów. Z tą wiedzą związane jest zobowiązanie do *działania*.

## Bibliografia

[McConnell87]: Malcolm McConnell, *Challenger. A Major Malfunction*, New York, Simon & Schuster, 1987.

[Wiki-Challenger]: „Katastrofa promu Challenger”, [http://pl.wikipedia.org/wiki/Katastrofa\\_promu\\_Challenger](http://pl.wikipedia.org/wiki/Katastrofa_promu_Challenger).

---

# PODZIĘKOWANIA

---

Moja kariera jest całą serią epizodów współpracy i planowania. Sam miałem wiele różnych marzeń i dążeń, ale zawsze udawało mi się znaleźć kogoś, kto je podzielał. Pod tym względem czuję się trochę jak Sith. „Zawsze dwóch ich jest”.

Pierwsza współpraca, którą mógłbym uznać za profesjonalną, miała miejsce, gdy miałem 13 lat, a moim współpracownikiem był John Marchese. Razem planowaliśmy zbudowanie własnego komputera. Ja miałem mózg, a on mięśnie. Pokazywałem mu, gdzie ma przylutować przewód, a on go przylutowywał. Pokazywalem, gdzie ma zamocować przekaźnik, a on go mocował. To była przednia zabawa, na której spędziliśmy setki godzin. I rzeczywiście zbudowaliśmy kilka całkiem imponująco wyglądających rzeczy wyposażonych w przyciski, przekaźniki, światelka, a nawet teletekst! Oczywiście żadna z tych rzeczy nie wykonywała nic konkretnego, ale były niezwykle imponujące i poświęciliśmy im bardzo dużo czasu. Dziękuję Ci za to, John!

W pierwszym roku szkoły średniej na lekcjach niemieckiego spotkałem Tima Conrada. Tim był naprawdę *mądry*. Gdy przystępowaliśmy do budowania komputera, to on był mózgiem, a ja mięsniami. Nauczył mnie elektroniki i zapoznał z komputerem PDP-8. Razem zbudowaliśmy działający elektroniczny kalkulator 18-bitowy, wykorzystując do tego podstawowe elementy. Potrafił on dodawać, odejmować, mnożyć i dzielić. Zajęło nam to wszystkie weekendy jednego roku oraz całe wakacje letnie i bożonarodzeniowe. Pracowaliśmy nad nim bez ustanku i na koniec kalkulator działał doskonale. Dziękuję Ci za to, Tim!

Razem z Timem nauczyłem się programować komputery. W roku 1968 nie było to łatwe, ale jakoś się udało. Zdobyliśmy książki o asemblerze komputera PDP-8, o językach Fortran, Cobol oraz PL/1 i po prostu je pochłanialiśmy. Pisaliśmy programy, których nie dało się

nawet uruchomić, ponieważ nie mieliśmy dostępu do komputera. Ale pisaliśmy je mimo to z czystej przyjemności.

W drugiej klasie liceum nasza szkoła zaczęła prowadzić zajęcia z nauk komputerowych. Do modemu telefonicznego o prędkości 110 bodów podłączono teletekst ASR-33. Szkoła miała własne konto w systemie podziału czasu Univac 1108 prowadzonym przez Institute of Technology w stanie Illinois. Tim i ja bardzo szybko staliśmy się faktycznymi operatorami tego sprzętu. Nikt poza nami nie miał prawa się do niego zbliżać.

Połączenie modemowe wykonywało się przez podniesienie słuchawki i ręczne wybranie numeru. Po usłyszeniu pisków modemu z drugiej strony należało naciąć przycisk „orig” na teleteksie, żeby nasz modem zaczął pisać w drugą stronę. Teraz wystarczyło odwiesić słuchawkę i połączenie było ustanowione.

Tarcza telefonu była zabezpieczona kluczem, do którego dostęp mieli wyłącznie nauczyciele. Nie miało to jednak znaczenia, ponieważ wykombinowaliśmy, że można wybrać numer (dowolny), wystukując go za pomocą wyłącznika pod słuchawką. Byłem perkusistą, więc miałem całkiem niezłe wyczucie czasu i refleks. Nawet przy zablokowanej tarczy telefonu byłem w stanie wybrać numer w mniej niż 10 sekund.

W laboratorium dostępne były dwa telegrafy. Tylko jeden z nich był maszyną podłączoną do sieci telefonicznej, ale oba były używane przez uczniów do pisania programów. Pisali oni programy, których kod był wybijany na taśmie papierowej. Każe naciśnięcie klawisza powodowało wybicie nowych dziurek w taśmie. Programy były pisane w niezwykle rozbudowanym, interpretowanym języku IITran. Przygotowane taśmy z programami należało zostawić w koszyku umieszczonym obok telegrafów.

Po szkole Tim i ja wybieraliśmy numer komputera (oczywiście wystukując go), ładowaliśmy zawartość taśm do systemu wykonawczego IITran i rozłączaliśmy się. Przy prędkości 10 znaków na sekundę cała procedura zajmowała trochę czasu. Po mniej więcej godzinie ponownie łączymy się z komputerem, aby wydrukować wyniki. Znow z prędkością 10 znaków na sekundę. System nie rozdzielał listingów poszczególnych uczniów na pojedyncze strony. Drukował po prostu stronę za stroną, dlatego musielibyśmy pociąć je nożyczkami, dołączyć odpowiednie taśmy z programami i odłożyć do koszyka z wynikami.

Tim i ja byliśmy w tym mistrzami. Nawet nauczyciele nas nie nadzorowali, gdy byliśmy w laboratorium. Wiedzieli, że wykonujemy ich pracę, choć nigdy nas o to nie prosili ani nawet nie zasugerowali takiej możliwości. Nigdy też nie dostaliśmy klucza do telefonu. Po prostu my się tym zajmowaliśmy, a oni wychodzili, pozostawiając nam bardzo dużo swobody. Dziękuję zatem moim nauczycielom matematyki: panu McDermitowi, panu Fogelowi i panu Robienowi.

Potem, po zakończeniu prac domowych, mogliśmy zająć się zabawą. Pisaliśmy program za programem, a robiły one najróżniejsze szalone i dziwaczne rzeczy. Napisaliśmy program

rysujący okręgi i parbole za pomocą znaków ASCII drukowanych przez telegraf. Pisaliśmy generatory losowych słów. Do ostatniej cyfry wyliczaliśmy silnię z 50. Poświęcaliśmy całe godziny na wymyślanie programów, pisanie ich i uruchamianie.

Dwa lata później Tim, nasz kolega Richard Lloyd i ja zostaliśmy zatrudnieni jako programiści w firmie ASC Tabulating w Lake Bluff w stanie Illinois. W tym czasie Tim i ja mieliśmy po 18 lat. Zdecydowaliśmy, że liceum to strata czasu i powinniśmy od razu zacząć nasze kariery. To właśnie w tej firmie spotkaliśmy Billa Hohriego, Franka Rydera, Wielkiego Jima Carlina oraz Jona Millera. To dzięki nim żółtodzioby mogły się dowiedzieć, co oznacza profesjonalne tworzenie oprogramowania. Doświadczenie to nie było ani całkowicie pozytywne, ani całkowicie negatywne, ale z całą pewnością było pouczające. Dziękuję zatem wszystkim wymienionym oraz Richardowi, który był katalizatorem i siłą napędową całego tego procesu.

W wieku 20 lat, po złożeniu wypowiedzenia i uspokojeniu się, przez pewien czas pracowałem jako serwisant kosiarki do trawy w firmie mojego szwagra. W tym fachu byłem tak fatalny, że szwagier był zmuszony mnie wyrzucić z pracy. I za to Ci dziękuję, Wes!

Jakiś rok później rozpoczęłem pracę w firmie Outboard Marine Corporation. W tym czasie miałem już żonę i dziecko w drodze. Stamtąd też zostałem zwolniony. Dziękuję Wam, John, Ralph i Tom!

Następnie zacząłem pracę w firmie Teradyne, gdzie spotkałem Russa Ashdowna, Kena Findera, Boba Copithorne'a, Chucka Studee oraz CK Srithrana (teraz Krisa Iyera). Ken był moim szefem, a Chuck i CK kolegami. Od nich wszystkich bardzo dużo się nauczyłem. Dziękuję Wam, chłopaki!

Później pojawił się Mike Carew. W Teradyne razem stworzyliśmy dynamiczny duet. Razem napisaliśmy kilka systemów. Jeżeli ktoś chciał coś przygotować i zrobić to naprawdę szybko, to Bob i Mike doskonale się do tego nadawali. Świetnie się razem bawiliśmy. Dziękuję Ci, Mike.

Jerry Fitzpatrick również pracował w Teradyne. Poznaliśmy się podczas wspólnych sesji Dungeons & Dragons i bardzo szybko zaczęliśmy współpracować. Napisaliśmy oprogramowanie dla Commodore 64 wspomagające graczy w D&D. W Teradyne rozpoczęliśmy też prace nad nowym projektem, który otrzymał nazwę „elektronicznego recepcjonisty”. Pracowaliśmy razem przez kilka lat, a Jerry został i pozostał moim bliskim przyjacielem. Dziękuję Ci za to, Jerry!

Pracując dla firmy Teradyne, spędzałem cały rok w Anglii, gdzie spotkałem Mike'a Kergozou. Razem snuliśmy wiele różnych planów, choć większość z nich była związana z motocyklami i pubami. Mike był jednak oddanym programistą poświęcającym wiele uwagi kwestiom jakości i dyscypliny (choć sam pewnie by się nie zgodził z tym twierdzeniem). Dziękuję Ci, Mike!

Gdy w 1987 roku wróciłem z Anglii, zacząłem swoje zwykłe planowanie wspólnie z Jimem Newkirkiem. Obaj odeszliśmy z Teradyne (w odstępie miesiąca) i zatrudniliśmy się w start-upie o nazwie Clear Communications. Następnych kilka lat spędziliśmy razem, harując w nadziei na miliony, które jakoś się nie pojawiły. Nigdy jednak nie zaprzestaliśmy planowania. Dziękuję Ci, Jim!

Na koniec wspólnie założyliśmy firmę Object Mentor. Jim jest najbardziej bezpośrednią, zdyscyplinowaną i skupioną osobą, z jaką kiedykolwiek zdarzyło mi się pracować. Nauczył mnie tak wielu rzeczy, że nawet trudno byłoby mi je wymienić. W zamian zadeykowałem mu tę książkę.

Współpracowałem i snułem plany z wieloma innymi osobami, a jeszcze więcej miało wpływ na moje zawodowe życie: Lowell Lindstrom, Dave Thomas, Michael Feathers, Bob Koss, Brett Schuchert, Dean Wampler, Pascal Roy, Jeff Langr, James Grenning, Brian Button, Alan Francis, Mike Hill, Eric Meade, Ron Jeffries, Kent Beck, Martin Fowler, Grady Booch i wielu, wielu innych. Wszystkim Wam z całego serca dziękuję.

Oczywiście moim najważniejszym współpracownikiem była moja ukochana żona Ann Marie. Ożeniłem się z nią w wieku 20 lat, całe trzy dni po jej 18 urodzinach. Przez 38 lat była moim niezłomnym kompanem, moim sternikiem i żaglem, miłością mojego życia. Mam nadzieję przeżyć z nią kolejne cztery dekady.

Aktualnie moimi współpracownikami i partnerami w snuci planów są moje dzieci. Ścisłe współpracuję z moją najstarszą córką Angelą — moją dzielną opiekunką i nieustraszoną pomocnicą. Nigdy nie pozwala mi zboczyć z wybranej ścieżki ani zapomnieć o terminach lub zobowiązaniach. Obmyślam plany biznesowe z moim synem Micahem, założycielem firmy 8thlight.com. Muszę powiedzieć, że ma znacznie lepszą głowę do interesów, niż ja miałem kiedykolwiek. Nasz ostatni projekt — cleancoders.com — jest naprawdę bardzo ekscytujący.

Mój młodszy syn Justin właśnie zaczął pracować razem z Micahem w 8th Light. Z kolei młodsza córka Gina jest chemikiem i pracuje dla Honeywell. Z tymi dwojgiem dopiero zaczęliśmy poważne planowanie.

To właśnie dzieci są w stanie nauczyć Cię więcej niż ktokolwiek inny w Twoim życiu. Dziękuję Wam za to, dzieciaki!

---

# O AUTORZE

---



**Robert C. Martin („Wujek Bob”)** był programistą od 1970 roku. Jest założycielem i prezesem Object Mentor, międzynarodowej firmy skupiającej doświadczonych inżynierów oprogramowania oraz menedżerów specjalizujących się we wspomaganiu firm podczas prac nad najróżniejszymi projektami. Firma Object Mentor oferuje korporacjom na całym świecie konsultacje z usprawniania procesów i obiektowego projektowania oprogramowania, szkolenia oraz usługi z zakresu rozwoju umiejętności oprogramowania.

Martin opublikował dziesiątki artykułów w najróżniejszych specjalistycznych magazynach i jest częstym prelegentem na międzynarodowych konferencjach i targach.

Jest autorem oraz redaktorem wielu książek, w tym:

- *Designing Object Oriented C++ Applications Using the Booch Method*
- *Patterns Languages of Program Design 3*
- *More C++ Gems*

- *Extreme Programming in Practice*
- *Agile Software Development: Principles, Patterns, and Practices*
- *UML for Java Programmers*
- *Czysty kod. Podręcznik dobrego programisty*, (Helion 2010)

Będąc liderem przemysłu tworzącego oprogramowanie, Martin przez trzy lata pracował jako główny redaktor magazynu „C++ Report” oraz przewodniczący organizacji Agile Alliance.

Robert jest też założycielem firmy Uncle Bob Consulting LLC, a wspólnie ze swoim synem Micahem — współzałożycielem firmy The Clean Coders LLC.

---

# NA OKŁADCE

---



Niesamowity obraz umieszczony na okładce (przypomina oko Saurona) to M1 lub mgławica kraba. M1 znajduje się w gwiazdozbiorze Byka, mniej więcej jeden stopień na prawo od Zeta Tauri, czyli gwiazdy znajdującej się na czubku lewego rogu byka. Mgławica kraba jest pozostałością po supernowej, która wybuchła 4 lipca 1054 roku. Mimo odległości 6500 lat świetlnych eksplozja ta pojawiła się przed oczami chińskich obserwatorów jako nowa gwiazda o jasności zbliżonej do jasności Jowisza. Podobno była widoczna nawet *w dzień!* Przez następnych sześć miesięcy powoli bladła, aż zniknęła z nieboskłonu.

Obrazek umieszczony na okładce jest złożeniem obrazów światła widzialnego oraz promieni X. Obraz w świetle widzialnym został zarejestrowany przez teleskop Hubble'a i umieszczony w zewnętrznej winiecie. Wnętrze obrazu, przypominające trochę niebieską tarczę łuczniczą, zostało zarejestrowane przez teleskop rentgenowski Chandra.

Prezentowany obraz przedstawia szybko rozprzestrzeniającą się chmurę pyłów i gazów, poprzetykaną cięższymi elementami pozostałymi po eksplozji supernowej. Ta chmura ma teraz 11 lat świetlnych średnicy, wagę równą 4,5 masy naszego Słońca i rozszerza się z przerząającą prędkością 1500 kilometrów na sekundę. Powiedzieć, że energia kinetyczna tej starej eksplozji jest imponująca, to lekkie niedopowiedzenie.

W samym środku znajduje się mała niebieska kropka. Właśnie w tym miejscu mieści się *pulsar*. To jego narodziny spowodowały eksplozję starej gwiazdy. Materiał jądra ginącej gwiazdy o masie niemalże dorównującej masie naszego Słońca implodował, tworząc kulę neutronów o średnicy mniej więcej 30 kilometrów. Energia kinetyczna tej implozji, połączona z niezwykłym ostrzałem neutrin powstały podczas tworzenia tych wszystkich neutronów, rozsadziła powłoki gwiazdy i rozerwała ją na strzępy.

Powstały pulsar obraca się z prędkością mniej więcej 30 obrotów na sekundę, jednocześnie błyskając, co możemy obserwować przez nasze teleskopy. Właśnie to pulsujące światło sprawiło, że takie gwiazdy są nazywane pulsarami, co jest skrótem od Pulsating Star, czyli pulsująca gwiazda.

---

# OBOWIĄZKOWY WSTĘP

---

(*Nie pomijaj tego, będzie później potrzebne*).



Zakładam, że masz w ręce tę książkę, ponieważ jesteś programistą i zaintrygowała Cię wzmianka o profesjonalizmie. Tak powinno być. Profesjonalizm jest czymś, czego nasz zawód bardzo potrzebuje.

Ja również jestem programistą. Byłem nim przez ostatnie 42<sup>1</sup> lata i w tym czasie — *naprawdę nie kłamię* — widziałem już wszystko. Zostałem zwolniony z pracy, byłem wychwalany, byłem szefem zespołu, menedżerem, szeregowym programistą, a nawet prezesem firmy. Współpracowałem ze znakomitymi programistami, ale i z nieudacznikami. Pracowałem

---

<sup>1</sup> Bez paniki.

na najnowocześniejszych, osadzonych systemach sprzętowo-programowych, ale też na budżetowych systemach korporacyjnych. Programowałem w COBOL-u, FORTRAN-ie, BAL-u, PDP-8, PDP-11, C, C++, Javie, Ruby, Smalltalku i masie innych języków i systemów operacyjnych. Współpracowałem z niegodnymi zaufania złodziejami wypłat, ale także z doskonałymi zawodowcami. Ale tematem niniejszej książki jest ta ostatnia klasyfikacja.

Na stronach tej książki będę się starał zdefiniować znaczenie terminu „profesjonalny programista”. Będę opisywać nastawienie, dyscypliny i działania, które uważam za niezbędne dla osiągnięcia profesjonalizmu.

Skąd wiem, że właśnie o te cechy chodzi? Ponieważ uzyskałem tę wiedzę w czasie lat swojej pracy. Gdy dostałem pierwszą pracę jako programista, zdecydowanie nie można było mnie określić mianem profesjonalisty.

Był to rok 1969, a ja miałem 17 lat. Mój ojciec namówił lokalną firmę o nazwie ASC, żeby zatrudniła mnie jako tymczasowego programistę na pół etatu. (Owszem, mój ojciec umiał robić takie rzeczy. Raz nawet widziałem, jak wyszedł prosto przed maskę przekraczającego dozwoloną prędkość samochodu, krzycząc w jego kierunku „Stój”! Samochód się zatrzymał. Nikt nie umiał odmówić mojemu ojcu). Firma zapędziła mnie do pracy w pomieszczeniu, w którym trzymane były wszystkie podręczniki do komputerów IBM. Kazali mi dodać do nich całe lata aktualizacji. To właśnie wtedy pierwszy raz zobaczyłem tekst „Ta strona jest celowo pusta”.

Po paru dniach aktualizowania podręczników mój przełożony poprosił mnie o napisanie prostego programu w języku Easycoder<sup>2</sup>. Byłem zaszczystyony taką prośbą. Jeszcze nigdy wcześniej nie pisałem programów dla prawdziwego komputera. Zdążyłem już jednak przeczytać książkę o języku Autocoder i wiedziałem mniej więcej, jak należy zacząć.

Program miał tylko odczytywać rekordy z taśmy i zamieniać ich identyfikatory na inne. Nowe identyfikatory zaczynały się od 1 i każdy następny rekord miał otrzymywać wartość większą o 1 od poprzedniego. Oczywiście rekordy z nowymi identyfikatorami miały być zapisywane na taśmie.

Mój przełożony pokazał mi półkę, na której leżało wiele zestawów czerwonych i niebieskich kart perforowanych. Wyobraź sobie, że kupujesz 50 talii kart do gry, 25 czerwonych i 25 niebieskich. Następnie układasz talie jedna na drugiej. Dokładnie tak wyglądały te karty perforowane. Ułożone były w czerwone i niebieskie paski, a każdy z pasków składał się z jakichś 200 kart i zawierał kod źródłowy biblioteki procedur używanych przez programistów. Programiści zwykle brali górną talię kart, uważając, żeby chwycić tylko czerwone albo tylko niebieskie karty. Potem kładli ją na końcu talii ze swoim programem.

---

<sup>2</sup> Easycoder był językiem asemblera komputerów Honeywell H200, bardzo podobnym do języka Autocoder dla komputerów IBM 1401.

Swój program napisałem na formularzach do programowania. Były to wielkie prostokątne arkusze podzielone na 25 wierszy po 80 kolumn. Każdy wiersz reprezentował jedną kartę. Program pisało się na nich, stosując wielkie litery i używając ołówka. W ostatnich 6 kolumnach ołówkiem zapisywany był numer sekwencji. Zazwyczaj numery były powiększane o 10, tak żeby możliwe było późniejsze wstawienie nowych kart.

Formularz był następnie przekazywany do wybijaczy kart. W firmie pracowało kilkudziesiąt kobiet, które pobierały takie formularze z korytką i „wpisywały” je do maszyn wybijających karty. Maszyny te były podobne do maszyn do pisania, jednak znaki nie były drukowane na papierze, ale wybijane na kartach.

Nazajutrz w firmowej poczcie dostałem swój program w postaci kart perforowanych. Moja mała talia kart była owinięta w formularz do programowania i gumkę. Przejrzałem wszystkie karty, szukając na nich błędów powstałych przy przepisywaniu, ale ich nie znalazłem. Wziąłem zatem talię z biblioteką procedur, połączylem ją z talią mojego programu i zabrałem je do operatorów komputerów.

Komputery znajdowały się za zamkniętymi drzwiami w pomieszczeniu klimatyzowanym o podniesionej podłodze (musiała pomieścić te wszystkie kable). Zapukałem do drzwi i jeden z operatorów bez słowa wziął moją talię i położył ją w kolejnym korytku. Gdy przyjdzie na to czas, zajmą się uruchomieniem mojego programu.

Następnego dnia znowu dostałem swoją talię. Była owinięta w listę wyników pracy programu, którą przytrzymywała gumka. (W tamtych czasach używaliśmy *wielu* gumek!)

Otwarłem listę wyników i zobaczyłem, że mojego programu nie udało się skompilować. Wypisane komunikaty o błędach były dla mnie słabo zrozumiałe, dlatego poszedłem do przełożonego. Przejrzał całą listę, mamrocząc coś pod nosem, dopisał szybko kilka słów, złapał moją talię kart i kazał iść za nim.

Zabrał mnie do pokoju z wybijkami kart i usiadł przy jednej z wolnych maszyn. Po kolei poprawiał karty zawierające błędy i dodał jedną nową lub dwie. Opisywał mi, co właściwie robi, ale wszystko działało się zdecydowanie za szybko.

Nową talię zabrał do pomieszczenia komputerów i zapukał do drzwi. Jednemu z operatorów powiedział kilka magicznych słów i wszedł do pomieszczenia za nim, pokazując mi, że i ja mam wejść. Operator przygotował napędy taśmowe i załadował talię kart, a my po prostu patrzyliśmy. Taśmy zaczęły się kręcić, drukarka zaczęła trzaskać i już było po wszystkim. Program zadziałał.

Następnego dnia mój przełożony podziękował mi za pomoc i wypowiedział pracę. Najwyraźniej firma nie miała czasu na nianczenie siedemnastolatka.

Ale moje związki z ASC jeszcze się nie skończyły. Kilka miesięcy później dostałem pełny etat na drugiej zmianie jako operator drukarek. Drukowały one pisma reklamowe na podstawie

danych zapisywanych na taśmie. Moim zadaniem było ładowanie papieru do drukarek, ładowanie taśm do napędów, usuwanie zaciętych kartek i pilnowanie, żeby maszyny działały.

Był rok 1970. Nie miałem szans na studiowanie, ale ta perspektywa nie była też dla mnie szczególnie kusząca. Nadal szalała wojna w Wietnamie, a na kampusach panował chaos. Nadal pochłaniałem książki o COBOL-u, FORTRAN-ie, PL/1, PDP-8 i asemblerze IBM 360. Miałem plan, żeby pominąć szkołę i jak najszybciej dostać pracę jako programista.

Dwanaście miesięcy później udało mi się osiągnąć ten cel. Dostałem awans na programistę w firmie ASC. Ja oraz dwóch moich dobrych przyjaciół, Richard i Tim (również mieli po 19 lat), pracowaliśmy w zespole z trzema innymi programistami, tworząc działający w czasie rzeczywistym system księgowy dla Teamsters Union. Do dyspozycji mieliśmy maszynę Varian 620i, czyli po prostu minikomputer o architekturze zbliżonej do PDP-8, z tą różnicą, że miał 2 rejestrów i był 16-bitowy. Używanym językiem był assembler.

W tym systemie pisaliśmy każdy wiersz kodu. Dosłownie *każdy*. Napisaliśmy system operacyjny, obsługę przerwań, sterowniki wejścia i wyjścia, *system plików* na potrzeby dysku, przełącznik nakładek, a nawet konsolidator. Nie wspominając nawet o kodzie samej aplikacji. Pisaliśmy to wszystko przez 8 miesięcy, pracując po 70 lub 80 godzin na tydzień, by dotrzymać wyznaczonego terminu. Moja pensja wynosiła wtedy 7200 dolarów rocznie.

Dostarczyliśmy system na czas, a zaraz potem złożyliśmy wypowiedzenie.

Odeszliśmy nagle i z pełną premedytacją. Po tak gigantycznej pracy, gdy na czas dostarczyliśmy świetnie działający system, firma dała nam podwyżkę o całe 2%. Czuliśmy się oszukani i wykorzystani. Niektórzy z nas dostali pracę w innych firmach i po prostu odeszli.

Ja jednak przyjąłem nie do końca szczęśliwe rozwiązanie. Razem z kolegą wpadliśmy do biura naszego szefa i całkiem głośno rzuciliśmy pracę. Przez jeden dzień było to naprawdę satysfakcjonujące.

Nastecnego dnia dotarło do mnie, że nie mam pracy. Miałem 19 lat, byłem bezrobotny i nie miałem skończonych studiów. Starałem się o kilka stanowisk programisty, ale rozmowy kwalifikacyjne nie poszły najlepiej. Zacząłem więc pracować w firmie naprawiającej kosiarki należącej do mojego szwagra, w której przetrwałem cztery miesiące. Niestety, do naprawiania kosiarek zupełnie się nie nadawałem. Ostatecznie szwagier musiał mnie pożegnać, co napędziło mi wielkiego stracha.

Co noc siedziałem do 3 nad ranem przed starym czarno-białym telewizorem moich rodziców, jedząc pizzę i oglądając stare filmy o potworach. W łóżku leżałem potem do 1 po południu, ponieważ nie chciałem mierzyć się z przerażającą rzeczywistością. Na miejscowym uniwersytecie skorzystałem z kursu analizy matematycznej, którego nie udało mi się zaliczyć. Pograżałem się.

Matka wzięła mnie pewnego dnia na stronę i powiedziała mi, że moje życie to katastrofa, że byłem idiotą, rzucając pracę i nie mając nic w zamian, rzucając ją tak emocjonalnie i wspólnie z kolegą. Powiedziała mi też, że nie wolno rzucać pracy, jeśli nie ma się już nowego kontraktu, że takie rzeczy robi się po cichu, spokojnie i samodzielnie. Powiedziała mi, że mam zadzwonić do swojego byłego szefa i błagać go o ponowne przyjęcie. Na koniec stwierdziła: „Musisz nauczyć się pokory”.

Dziewiętnastoletni chłopcy raczej nie mają ochoty na naukę pokory, a ja nie byłem żadnym wyjątkiem, jednak okoliczności mocno nadwątpiły moją dumę. Ostatecznie musiałem zadzwonić do byłego szefa i się przed nim upokorzyć. Ale zadziałało. Bardzo chętnie zatrudnił mnie za 6800 dolarów rocznie, a ja byłem szczęśliwy, że mam pracę.

W tej pracy spędziłem kolejnych 18 miesięcy, pilnując się i próbując być jak najbardziej wartościowym pracownikiem. Zostałem za to nagrodzony awansami, podwyżkami i regularną wypłatą. Życie nabralo kolorów. Gdy ponownie opuściłem tę firmę, odbyło się to na dobrych warunkach, a co ważniejsze, miałem już wtedy propozycję lepszej pracy.

Możesz sądzić, że czegoś się w tym czasie nauczyłem, że stałem się profesjonalistą. Byłem od tego daleki. To była dopiero pierwsza z czekających mnie lekcji. W kolejnych latach zostałem zwolniony z pracy za niedbanie o dotrzymywanie ważnych terminów, a z innej prawie mnie wyrzucono za nieumyślne przekazanie klientowi tajnych informacji.

Zajmowałem się prowadzeniem projektu skazanego na porażkę; pozwoliłem na jego zagładę, ponieważ nie poprosiłem nikogo o pomoc, której tak bardzo potrzebowałem. Agresywnie broniłem swoich decyzji technicznych, mimo że w konfrontacji z potrzebami klienta nie miały one sensu. Przyjąłem do pracy osobę zupełnie do niej nieprzygotowaną, czym wpędziłem pracodawcę w kłopoty prawne. A co gorsza, moja niezdarność jako przełożony spowodowała, że dwie osoby zostały zwolnione z pracy.

Tę książkę można zatem traktować jak zbiór moich własnych błędów, katalog moich występków i zbiór wskazówek pozwalających Ci uniknąć wpadnięcia w te same pułapki.

## **OBOWIĄZKOWY WSTĘP**

---

---

# 1 PROFESJONALIZM

---



*Śmiej się, Curtin! Ktoś spłatał nam figla! Bóg, los, natura — sam wybierz.  
Jednak ktokolwiek to był, miał niezłe poczucie humoru!*

— Howard, Skarb Sierra Madre

Dobrze rozumiem, że chcesz być zawodowym programistą? Chcesz unieść wysoko głowę i z komunikować całemu światu: „Jestem zawodowcem!”. Chcesz, żeby ludzie patrzyli na Ciebie z szacunkiem i traktowali Cię z poważaniem. Chcesz, żeby matki wskazywały Cię palcem, mówiąc swoim dzieciom, że mają starać się upodobnić do Ciebie. Tego właśnie chcesz. Mam rację?

## **Uważaj, czego sobie życysz**

Profesjonalizm to bardzo przeładowane pojęcie. Z całą pewnością jest powodem do dumy, ale też znakiem odpowiedzialności. Te dwa elementy są nierozerlaczne. Nie da się czuć dumy z czegoś, za co nie jest się odpowiedzialnym.

Znacznie łatwiej jest nie być profesjonalistą. Nie trzeba wtedy brać odpowiedzialności za swoją pracę — tę można pozostawić pracodawcom. Jeżeli nieprofesjonalista popełnia błąd, to pracodawca musi po nim posprzątać. Jeżeli jednak błąd popełnia profesjonalista, to *sam* stara się go skorygować.

Co by się stało, gdyby z powodu Twojego niedopatrzenia do systemu wkradł się błąd, który kosztowałby firmę 10 000 dolarów? Nieprofesjonalista wzruszyłby ramionami, stwierdzając „takie rzeczy się zdarzają”, i przystąpiłby do pracy nad kolejnym modułem. Profesjonalista wypisałby dla firmy czek na kwotę 10 000 dolarów<sup>1</sup>!

Gdy chodzi o Twoje własne pieniądze, całość wygląda nieco inaczej. Ale tak właśnie czuje się profesjonalista. Co więcej, to uczucie jest esencją profesjonalizmu. Po prostu w profesjonalizmie chodzi o przyjęcie odpowiedzialności.

## **Przejmowanie odpowiedzialności**

Mam nadzieję, że przeczytałeś wprowadzenie. Jeżeli nie, to lepiej zrób to w tej chwili. To właśnie wprowadzenie tworzy kontekst dla całej zawartości książki.

Wszystko, co wiem na temat odpowiedzialności, poznałem jako konsekwencję braku odpowiedzialności.

W 1979 roku pracowałem dla firmy o nazwie Teradyne. Byłem tam inżynierem odpowiedzialnym za oprogramowanie kontrolujące system mini- i mikrokomputerowy mierzący jakość linii telefonicznych. Centralny minikomputer był połączony za pomocą dedykowanych linii telefonicznych o szybkości 300 bodów z dziesiątkami satelitarnych mikrokomputerów sterujących oprogramowaniem pomiarowym. Całość kodu została napisana w asemblerze.

---

<sup>1</sup> Można mieć nadzieję, że wyznaje dobre zasady radzenia sobie z błędami i omyłkami.

Naszymi klientami byli menedżerowie serwisu największych firm telekomunikacyjnych. Każdy z nich był odpowiedzialny za przynajmniej 100 000 linii telefonicznych. Mój system pomagał im znaleźć i naprawić usterkę linii telefonicznej, jeszcze zanim zauważą ją klienci. Takie działanie zmniejszało liczbę skarg składanych przez klientów, która była kontrolowana przez instytucje publiczne ustalające stawki pobierane przez firmy telekomunikacyjne. W skrócie: te systemy były niezmierne ważne.

Co noc systemy uruchamiały specjalną „procedurę nocną”, w ramach której centralny minikomputer nakazywał poszczególnym satelickim mikrokomputerom, żeby te przetestowały wszystkie linie telefoniczne znajdujące się pod ich kontrolą. Każdego ranka centralny komputer zbierał listę uszkodzonych linii wraz ze szczegółami na temat każdego uszkodzenia. Menedżerowie serwisowi korzystali z tych raportów w celu przygotowania planu prac dla serwisantów, tak żeby mogli oni usunąć usterki, zanim zauważą je klienci.

Pewnego dnia wysłałem do kilku klientów nową wersję oprogramowania. „Wysłałem” jest tutaj określeniem jak najbardziej na miejscu. Zapisałem oprogramowanie na taśmach i wysłałem je pocztą do klientów. Klienci po otrzymaniu taśm załadowali je do systemów i ponownie uruchomili komputery.

Nowa wersja usuwała kilka drobnych defektów i dodawała nową funkcję, której klienci domagali się już od pewnego czasu. Poinformowaliśmy, że funkcję tę dostarczymy przed określona datą, a terminu udało się dotrzymać tylko dlatego, że zdążyłem jeszcze wysłać taśmy przesyłką ekspresową, dostarczaną następnego dnia.

Dwa dni później zadzwonił do mnie nasz menedżer serwisu Tom. Powiedział mi, że część klientów skarży się z powodu nieukończenia „procedury nocnej” i braku raportów o uszkodzeniach. Robilem wszystko, co w mojej mocy, żeby wysłać oprogramowanie na czas, ale przy okazji pominąłem testy procedury nocnej. Przeprowadziłem wiele testów pozostałych funkcji systemu, ale testowanie tej procedury ciągnęło się godzinami, a ja musiałem przygotować wysyłkę. Żadna z wprowadzonych poprawek nie dotyczyła procedury nocnej, więc nie widziałem też takiej potrzeby.

Utrata raportu nocnego była *poważnym problemem*. Oznaczała, że serwisanci mieli mniej pracy, ale za to później byli przeciążeni. Oznaczała, że klienci zauważali usterkę i składali reklamację. Utrata danych z raportu nocnego była wystarczająca, żeby menedżerowie serwisowi dzwoniли do Toma z awanturą.

Uruchomiłem nasz system laboratoryjny, załadowałem nowe oprogramowanie i uruchomiłem procedurę. Działała przez kilka godzin, po czym została przerwana. Cała procedura nocna się załamała. Gdybym uruchomił ten test jeszcze przed wysłaniem taśm, to klienci nie traciliby danych, a menedżerowie serwisu nie przypiekaliby Toma żywym ogniem.

Zadzwoniłem to Toma, mówiąc mu, że udało mi się odtworzyć problem. Odpowiedział, że do tej pory większość klientów dzwoniła już do niego, by zgłosić tę samą usterkę. Zapytał też, kiedy uda mi się to naprawić. Powiedziałem, że nie wiem, ale cały czas nad tym pracuję. Jako tymczasowe rozwiązanie zaproponowałem, żeby klienci wrócili do poprzedniej wersji oprogramowania. Był na mnie zły, bo oznaczało to podwójny kłopot dla klientów. Po pierwsze utracili dane, a po drugie nie mogli skorzystać z nowej funkcji, na którą tak czekali.

Znalezienie błędu było naprawdę trudne, a testowanie ciągnęło się całymi godzinami. Pierwsza poprawka nie zadziałała, podobnie jak druga. Znalezienie przyczyny wymagało przynajmniej kilku nieudanych prób, przez co całość trwała parę dni. W tym czasie Tom dzwonił do mnie co kilka godzin, pytając, czy już poprawilem błąd. Przy okazji przekazywał mi pomyje, jakimi częstowali go dzwoniący menedżerowie serwisu, i przypominał, jakim wstydem było dla niego informowanie klientów o konieczności użycia starych taśm.

W końcu udało mi się znaleźć usterkę, wysłać nowe taśmy i wszystko wróciło do normy. Tom nie był moim szefem, dlatego po pewnym czasie uspokoił się i cały ten epizod puściliśmy w niepamięć. Z kolei mój szef przyszedł do mnie i powiedział: „Założę się, że więcej takiego numeru nie wykręcisz”. Nie mogłem się nie zgodzić.

Zastanawiając się nad tym incydentem, doszędłem do wniosku, że wysłanie taśm bez przeprowadzenia pełnych testów było nieodpowiedzialne. Jedynym powodem pominięcia procedury testowej była chęć wysłania taśm na czas. Chciałem w ten sposób zachować twarz. Nie zastanawiałem się nad konsekwencjami dla klientów i pracodawcy. Myślałem wyłącznie o swojej reputacji. Powinienem był jednak już wcześniej zadzwonić do Toma i powiedzieć mu, że testy nie są jeszcze zakończone i nie jestem gotowy, żeby wysłać oprogramowanie na czas. To byłoby trudne, a Tom na pewno by się zdenerwował. Ale żaden z klientów nie straciłby swoich danych i żaden menedżer serwisu nie musiałby do nas dzwonić.

## Po pierwsze nie szkodzić

W jaki sposób bierzemy na siebie odpowiedzialność? Istnieje w tym zakresie kilka zasad. Korzystanie z przysięgi Hipokratesa może wydawać się aroganckie, ale czy mamy jakieś lepsze źródło? Poza tym, czy nie jest sensowne, że pierwszym celem i podstawową odpowiedzialnością profesjonalisty powinno być wykorzystanie swoich umiejętności do czynienia dobra?

Jakich szkód może narobić programista? Z punktu widzenia czysto programowego może uszkodzić zarówno funkcje, jak i strukturę oprogramowania. Spróbujemy teraz zastanowić się, jak można tego uniknąć.

## Nie szkodzić funkcji

Oczywiście chcemy, żeby nasze oprogramowanie działało. Z pewnością większość z nas jest dzisiaj programistami, ponieważ swego czasu udało się nam coś uruchomić i chcemy ponownie poczuć się tak samo jak wtedy. Nie jesteśmy jednak jedynymi osobami, które chcą, żeby oprogramowanie działało. Chcą tego również nasi klienci i pracodawcy. Co więcej, płacą nam za to, żebyśmy przygotowali oprogramowanie, które będzie działało tak, jak oni to sobie wyobrażają.

Funkcję naszego oprogramowania uszkadzamy, wprowadzając do niego błędy. Oznacza to, że chcąc być profesjonalistami, nie możemy robić błędów.

Już słyszę, jak wiele głosów odzywa się w stylu: „Momencik! Przecież to nie ma sensu. Oprogramowanie jest zbyt złożone, żeby udało się je wytworzyć bez żadnych błędów”.

Zgadzam się całkowicie. Oprogramowanie *jest* zbyt skomplikowane, żeby można je było tworzyć bez błędów. Niestety, nie zwalnia nas to z odpowiedzialności. Ludzkie ciało jest zbyt złożone, żeby można było zrozumieć każdy jego szczegół, a mimo to lekarze przysięgają, że nie będą szkodzić. Skoro oni *nie* uciekają przed odpowiedzialnością, to czy my możemy to robić?

„Chcesz powiedzieć, że mamy być doskonali”? Czyżbym słyszał protesty?

Nie, twierdzę jedynie, że musimy być odpowiedzialni za swoje niedoskonałości. Fakt, że błędy na pewno pojawią się w Twoim kodzie, nie oznacza, że nie masz brać za nie odpowiedzialności. Fakt, że zadanie tworzenia doskonałego oprogramowania jest praktycznie nie do wykonania, nie oznacza, że nie jesteśmy odpowiedzialni za swoje niedoskonałości.

Cechą wyróżniającą zawodowca jest to, że bierze odpowiedzialność za błędy, mimo iż ich pojawienie się jest niemal całkowicie pewne. Oznacza to, że aspirując do miana profesjonalisty, przede wszystkim musisz nauczyć się przepraszać. Przeprosiny są niezbędne, ale niestety nie są wystarczające. Nie możesz po prostu cały czas robić nowych błędów. Dojrzewając w swoim zawodzie, musisz starać się, żeby liczba generowanych błędów cały czas spadała asymptotycznie w kierunku zera. Nigdy nie uda Ci się osiągnąć zera, ale Twoim zadaniem jest zbliżyć się do niego tak bardzo, jak się da.

## Kontrola jakości nie powinna nic znaleźć

Oznacza to, że gdy przygotowujesz nową wersję oprogramowania, spodziewasz się, że działa kontrola jakości nie znajdzie w niej żadnych problemów. Niezwykle nieprofesjonalne jest przesyłanie do kontroli kodu, o którym wiesz, że zawiera błędy. A o jakim kodzie wiesz, że zawiera błędy? Chodzi o dowolny kod, co do którego nie masz całkowitej pewności.

Znam ludzi, którzy wykorzystują dział kontroli jakości jako wykrywacz błędów. Wysyłają do niego kod, którego nie sprawdzili dość dokładnie. Mają nadzieję, że kontrola jakości znajdzie w kodzie błędy i prześle informację o nich do programistów. Co więcej, niektóre firmy nagradzają pracowników kontroli jakości na podstawie liczby znalezionych błędów. Im więcej błędów, tym wyższa nagroda.

Pomińmy już nawet to, że jest to niezwykle drogie zachowanie, które szkodzi zarówno firmie, jak i oprogramowaniu. Przymknijmy oko na to, że takie zachowanie niszczy plany i podminowuje zaufanie firmy do zespołu programistów. Spuśćmy zasłonę milczenia na fakt, że takie działanie jest przykładem lenistwa i nieodpowiedzialności. Przekazywanie do działu kontroli jakości kodu, co do którego nie mamy pewności, czy działa, jest najzwyczajniej w świecie nieprofesjonalne. W ten sposób naruszana jest zasada „nie szkodzić”.

Czy dział kontroli jakości znajdzie błędy? Zapewne tak, dlatego przygotuj już przeprosiny, a potem zastanów się, jak tym błędom udało się umknąć Twojej uwadze, i zrób wszystko, żeby w przyszłości się to nie powtórzyło.

Za każdym razem, gdy kontrola jakości albo co gorsza *użytkownik* znajdą w oprogramowaniu błąd, powinniśmy być zaskoczeni, zniesmaczeni i zdeterminowani, żeby się to nigdy więcej nie powtórzyło.

### **Musisz wiedzieć, że to działa**

Jak możesz mieć pewność, że Twój kod działa? To całkiem proste. Przetestuj go. A potem jeszcze raz. Przetestuj dogłębnie i na wylot. Testuj go na wszelkie możliwe sposoby.

Możesz się zastanawiać, czy taki ogrom testów nie będzie zajmować zbyt wiele czasu. W końcu musisz zrealizować plany i dotrzymać terminów. Jeżeli cały czas poświęcisz na testowanie, to nie zdołasz napisać właściwego kodu. Słuszna uwaga! Oznacza to tylko, że musisz zautomatyzować swoje testy. Napisz testy jednostkowe, które będzie można uruchomić w sekundę, i uruchamiaj je tak często, jak to będzie możliwe.

„Jaka część kodu powinna być testowana za pomocą tych zautomatyzowanych testów jednostkowych?” Czy naprawdę muszę odpowiadać na to pytanie? Oczywiście, że całość!

Czy sugeruję tu stu procentowe pokrycie kodu? Nie, wcale tego nie sugeruję. Ja się tego domagam. Każdy wiersz kodu, który wyszedł spod Twoich palców, powinien zostać przetestowany. Kropka.

Czy to nie jest aby nierealistyczne? Oczywiście, że nie. Piszesz kod tylko dlatego, że oczekujesz jego wykonania. A skoro oczekujesz, że zostanie wykonany, to musisz mieć pewność, że działa właściwie. Jedyną metodą, żeby to osiągnąć, jest jego przetestowanie.

Jestem głównym programistą i współtwórcą otwartoźródłowego projektu o nazwie FitNesse. Aktualnie projekt ten składa się z ponad 60 000 wierszy kodu. Spośród tych 60 aż 26 składa

się na ponad 2000 testów jednostkowych. Według raportów Emma pokrycie generowane przez te testy wynosi około 90%.

Dlaczego nie uzyskałem większego pokrycia kodu? Wynika to z tego, że Emma nie widzi wszystkich wierszy wykonywanego kodu. Sądzę zatem, że rzeczywiste pokrycie jest o wiele większe. Czy wynosi 100%? Nie, wartość 100% jest tutaj asymptotą.

Ale czy część kodu nie jest trudna do przetestowania? Owszem, ale tylko wtedy, gdy został on *zaprojektowany* tak, żeby trudno było go testować. Rozwiązaniem jest zawsze przygotowanie takiego projektu, który będzie *ulatwiał* testowanie kodu. A najlepszym sposobem na uzyskanie takiego projektu jest pisanie testów najpierw, jeszcze przed powstaniem kodu, który ma je przechodzić.

Taki sposób postępowania zyskał nazwę TDD (*Test Driven Development* — programowanie sterowane testami), o czym opowiem więcej w jednym z późniejszych rozdziałów.

## Zautomatyzowana kontrola jakości

Cały proces kontroli jakości dla projektu FitNesse polega na uruchomieniu testów jednostkowych i akceptacyjnych. Jeżeli te testy zostaną zaliczone, to mogę dostarczyć kolejną wersję. Oznacza to, że moja procedura kontroli jakości zajmuje mi jakieś trzy minuty i mogę ją uruchomić w dowolnym momencie.

Oczywiście w razie wystąpienia jakiegoś błędu w FitNesse raczej nikt nie umrze ani nikt nie straci milionów dolarów. Projekt ma jednak bazę wielu tysięcy użytkowników i jednocześnie *bardzo* krótką listę błędów.

Z całą pewnością istnieją systemy tak ważne, że krótki zautomatyzowany test nie jest wystarczający, aby sprawdzić jego gotowość do pracy. Ponadto każdy programista potrzebuje względnie szybkiego i skutecznego mechanizmu informującego go o tym, czy napisany kod nie wpływa negatywnie na pozostałe części systemu. Oznacza to, że zautomatyzowane testy mogą powiedzieć nam, czy system *ma szanse* przejść kontrolę jakości.

## Nie szkodzić strukturze

Prawdziwy zawodowiec wie, że realizowanie funkcji kosztem struktury to postępowanie głupca. To właśnie struktura kodu pozwala mu zachować elastyczność. Jeżeli narusysz tę strukturę, zaszkodzisz przyszłości projektu.

Podstawowym założeniem każdego projektu oprogramowania jest umożliwianie łatwej zmiany programu. Jeżeli naruszamy to założenie, tworząc nieelastyczne struktury, to zaczniemy podcinać model ekonomiczny, na którym bazuje cała nasza gałąź przemysłu.

W skrócie: *Musisz zachować możliwość wprowadzania zmian bez generowania gigantycznych kosztów.*

Niestety, zbyt wiele projektów potrafi utknąć w błotnistym dole źle zaprojektowanych struktur. Zadania, które wcześniej trwały kilka dni, zaczynają zajmować tygodnie, a potem nawet miesiące. Kadra zarządzająca w desperackiej próbie odzyskania poprzedniej prędkości prac zatrudnia więcej programistów. Ale ci nowi programiści pogłębiają tylko istniejące bagno, zwiększając liczbę nieprawidłowości w strukturach oprogramowania i spiętrzając przeszkodey.

Wiele napisano już na temat zasad i wzorców projektowych oprogramowania, które pozwalają na tworzenie elastycznych i łatwych w utrzymaniu struktur<sup>2</sup>. Profesjonalni twórcy oprogramowania zawsze pamiętają o tych rzeczach i starają się tworzyć zgodne z nimi oprogramowanie. Istnieje jednak pewna sztuczka, o której wie znacznie mniej programistów: *Jeżeli chcesz, żeby oprogramowanie było elastyczne, to musisz je pożynać!*

Jedyną metodą udowodnienia, że oprogramowanie jest elastyczne, jest wprowadzenie w nim zmian. Jeżeli stwierdzisz, że ich wprowadzenie nie jest tak proste, jak się początkowo wydawało, to znaczy, że konieczna jest zmiana projektu, tak aby zmiany były łatwiejsze.

Kiedy wprowadza się takie proste zmiany? *Cały czas!* Za każdym razem, gdy patrzysz na moduł, poprawiasz w nim szczegółowe usprawniające całą strukturę. Za każdym razem, gdy czytasz kod, dopasowujesz jego strukturę.

Ta filozofia czasami nazywana jest *bezlitosną refaktoryzacją*. Ja nazywam ją „regułą skauta”: dany moduł zawsze pozostaw czystszy, niż go zastałeś. Przeglądając kod, zawsze wyświadcz mu jakąś przysługę.

To zupełnie inny sposób myślenia o oprogramowaniu, być może obcy większości ludzi. Zwykle uważa się, że wprowadzanie ciągłych zmian do działającego już oprogramowania jest *niebezpieczne*. Tak wcale nie jest! Niebezpieczne jest pozwalanie na ustatkowanie się kodu. Jeżeli nie jest on cały czas zginany, to gdy zajdzie *potrzeba wprowadzenia zmian*, okaże się, że będzie on na nie niepodatny.

Dlaczego większość programistów obawia się wprowadzania zmian do swojego kodu? Po prostu boją się, że mogą coś zepsuć! A dlaczego boją się takiej ewentualności? Ponieważ nie korzystają z testów.

Wszystko sprowadza się do testów. Jeżeli masz zautomatyzowany zestaw testów pokrywający prawie 100% kodu i zestaw ten może zostać szybko uruchomiony w dowolnym momencie, to *po prostu nie boisz się wprowadzania zmian do kodu*. Jak możesz udowodnić, że nie boisz się wprowadzania zmian do kodu? Cały czas go zmieniając.

---

<sup>2</sup> [PPP2002].

Profesjonalni programiści są tak pewni swojego kodu i testów, że bez żadnych oporów prowadzą do kodu różne oportunistyczne zmiany. Ot tak zmieniają nazwę klasy. Jeżeli podczas czytania kodu modułu zauważą przydługą metodę, to przy okazji podzielią ją na mniejsze kawałki. Zamienią instrukcję wyboru w rozwiązywanie polimorficzne albo zwiną hierarchię dziedziczenia w strukturę typu łańcuch dowodzenia. W skrócie: traktują oprogramowanie tak jak rzeźbiarz traktuje glinę — cały czas je przekształcają i formują.

## Etyka pracy

Twoja kariera to *Twoja odpowiedzialność*. Zapewnienie Twojej atrakcyjności na rynku pracy nie jest zadaniem Twojego pracodawcy. Jego zadaniem nie jest też szkolenie Cię, wysyłanie na konferencje ani kupowanie Ci książek. To wszystko to *Twoja odpowiedzialność*. Biada programiście, który zawierzy swoją karierę pracodawcy.

Niektórzy pracodawcy chętnie kupują książki dla pracowników, wysyłają ich na kursy i konferencje. To bardzo miło, że robią Ci takie przysługi. Nigdy jednak nie wpadaj w pułapkę myślenia, że jest to zadanie pracodawcy. Jeżeli pracodawca tego nie robi, to musisz szukać sposobu, żeby robić to samodzielnie.

Podobnie nie jest zadaniem pracodawcy organizowanie Ci czasu niezbędnego na naukę. Niektórzy pracodawcy taki czas wygospodarowują, a część pracowników może się nawet tego domagać. I tu ponownie trzeba zaznaczyć, że takie działanie może być tylko miłym gestem pracodawcy, który musimy odpowiednio docenić. Nie można jednak od pracodawcy oczekwać takich prezentów.

Swojemu pracodawcy winni jesteśmy określony czas oraz nakład pracy. Na potrzeby tej dyskusji przyjmijmy standardowe w USA 40 godzin na tydzień. Tych 40 godzin powinniśmy spędzić, rozwiązyując problemy *pracodawcy*, a nie *swoje*.

Należy jednak planować pracę w zakresie 60 godzin w tygodniu, pierwszych 40 poświęcając swojemu pracodawcy, a pozostałe 20 zachowując dla siebie. Podczas tych dodatkowych 20 godzin możesz czytać, ćwiczyć, uczyć się lub w inny sposób rozwijać swoją karierę.

Zapewne zaczynasz teraz myśleć: „A co z moją rodziną? Co z moim życiem prywatnym? Czy mam je poświęcić na rzecz pracodawcy?”.

Nie mówię jednak o *całym* Twoim czasie wolnym, a jedynie o dodatkowych 20 godzinach tygodniowo. To zaledwie trzy godziny dziennie. Jeżeli wykorzystasz przerwę obiadową na czytanie, w drodze do i z pracy posłuchasz podcastów, a 90 minut poświęcisz każdego dnia na naukę nowego języka, to nic więcej nie trzeba.

To prosta matematyka. W tygodniu mamy 168 godzin. Pracodawcy poświęcasz 40, a na rozwój kariery kolejnych 20. Pozostaje 108 godzin. Sen zajmuje 56, a zatem zostają nam 52 godziny na inne aktywności.

Być może nie chcesz takiego poświęcenia. Nie ma sprawy, ale wtedy nie myśl o sobie jako o profesjonalistie. Zawodowcy poświęcają *czas*, żeby zadbać o swoją profesję.

Myślisz zapewne, że praca powinna pozostać w pracy i nie należy przynosić jej do domu. Całkowicie się zgadzam! Podczas tych dodatkowych 20 godzin nie należy pracować dla swojego pracodawcy. Ten czas poświęć na pracę nad własną karierą.

Czasami te dwa cele są ze sobą zgodne. Czasami praca dla pracodawcy jest bardzo korzystna dla Twojej kariery. W takiej sytuacji całkiem rozsądne będzie poświęcenie części z tych 20 godzin pracodawcy. Pamiętaj jednak, że te 20 godzin ma służyć przede wszystkim *Tobie*. Dzięki nim masz podnieść swoją wartość jako zawodowca.

Możesz teraz pomyśleć, że jest to prosta recepta na wypalenie zawodowe. Wręcz przeciwnie! To recepta na *uniknięcie wypalenia*. Zakładam, że wybór zawodu programisty wynikał z Twojej pasji związanej z tworzeniem oprogramowania i dążysz do tego, by zostać profesjonalistą napędzanym pasją. Podczas tych 20 godzin należy robić wszystko, żeby tę pasję *podsycać*. Tych 20 godzin ma sprawiać Ci *przyjemność*!

## Znaj swoje otoczenie

Wiesz, czym jest wykres Nassiego-Schneidermana? Jeżeli nie wiesz, to dlaczego? Znasz różnicę między maszynami stanu Mealy'ego i Moora? Każdy powinien ją znać. Jesteś w stanie napisać procedurę quicksort bez szukania informacji? Wiesz, co oznacza pojęcie „analiza transformacji”? Jesteś w stanie wykonać funkcjonalną dekompozycję za pomocą diagramu przepływu danych? Co oznacza pojęcie „wędrowne dane” (ang. *tramp data*)? Obiło Ci się o uszy pojęcie *conascence*? A co z tablicami Parnasa?

Przez ostatnie 50 lat istnienia naszej dziedziny nauki powstało wiele ciekawych idei, technik, narzędzi i terminologii. Ile z nich udało Ci się poznać? Jeżeli chcesz być profesjonalistą, to musisz znać ich całkiem sporo i cały czas powiększać swoją wiedzę.

Dlaczego musimy to wszystko znać? Czy nasza dziedzina nie posuwa się naprzód w takim tempie, że te stare idee tracą na istotności? Pierwsza część tego pytania tylko z pozoru jest oczywista. Faktycznie nasza dziedzina rozwija się w szalonym tempie, ale co ciekawe, rozwój ten pod wieloma względami jest jedynie powierzchowny. To prawda, że nie czekamy już 24 godziny na komplikację kodu. To prawda, że piszemy systemy, których rozmiary dochodzą do gigabajtów. To prawda, że pracujemy w sieci rozciągającej się na cały świat, która pozwala nam na natychmiastowy dostęp do informacji. Mimo to nadal piszemy te same instrukcje *if* i *while*, z których korzystaliśmy już 50 lat temu. Wiele się zmieniło, ale wiele pozostało bez zmian.

Druga część pytania nie jest już tak oczywista. Zaledwie niewielka część idei powstałych w ciągu tych 50 lat straciła na ważności. To prawda, że niektóre z nich zostały zepchnięte na boczny tor. Metoda tworzenia oprogramowania w systemie wodospadu nie ma już tak

---

dobrej prasy. Ale nie oznacza to, że nie powinniśmy wiedzieć, na czym ona polega i jakie są jej dobre i złe strony.

Mimo wszystko większość idei wypracowanych w ciągu tych 50 lat dziś nadal jest tak samo wartościowa. Część z nich nawet zyskała na wartości.

Pamiętaj o przekleństwie Santayany: „Kto nie pamięta o przeszłości, skazany jest na to, żeby ją powtarzać”.

Oto *minimalna* lista rzeczy, które powinien wiedzieć każdy zawodowy twórca oprogramowania:

- Wzorce projektowe. Musisz być w stanie opisać wszystkie 24 wzorce opisane w książce GOF i mieć doświadczenie w pracy z wieloma wzorcami opisywanymi w książkach POSA.
- Zasady projektowania. Musisz znać zasady SOLID i dobrze poznać poszczególne terminy.
- Metody. Musisz znać metodologie XP, Scrum, Lean, Kanban, wodospadu, analizy strukturalnej i projektowania strukturalnego.
- Dziedziny. Musisz korzystać z technik TDD, projektowania obiektowego, programowania strukturalnego, ciągłej integracji i programowania w parach.
- Artefakty. Musisz wiedzieć, jak korzystać z UML, DFD, wykresów struktur, sieci Petriego, diagramów i tabel przejść stanów, diagramów przepływu oraz tabel decyzji.

## Ciągła nauka

To niezwykłe tempo zmian w naszej gałęzi przemysłu sprawia, że twórcy oprogramowania muszą ciągle przyswajać ogromne ilości wiedzy, tylko po to, żeby nadążyć za tym rozwojem. Biada architektowi, który zaprzestanie tworzenia kodu. Bardzo szybko okaże się, że nie przystaje do rzeczywistości. Biada programistom, którzy przestali uczyć się nowych języków. Będą mogli tylko popatrzeć, jak oddala się od nich czołówka. Biada projektantom, którzy przestaną uczyć się nowych technik i rozwiązań. Koledzy szybko ich prześcigną.

Chcesz korzystać z usług doktora, który przestał czytać o najnowszych postępach w medycynie? Zatrudnisz doradcę podatkowego, który nie śledzi najnowszych zmian w prawie? Dlaczego zatem mielibyśmy zatrudniać programistów, którzy nie śledzą nowości?

Czytaj książki, artykuły, blogi i tweety. Jeźdź na konferencje. Wstępuj do grup użytkowników. Weź udział w spotkaniach grup naukowych i czytelniczych. Ucz się rzeczy, które nie są dla Ciebie najłatwiejsze. Jeżeli jesteś programistą .NET, to naucz się Javy. Jeżeli normalnie programujesz w Javie, to poznaj język Ruby. Jeżeli programujesz w C, naucz się Lispu. A jeżeli naprawdę chcesz zmusić mózg do pracy, to zacznij poznawać Prolog lub Forth.

## Ćwiczenia

Zawodowcy nieustannie ćwiczą. Prawdziwi zawodowcy ciężko pracują nad ciągłym rozwojem swoich umiejętności. Nie wystarczy tylko wykonywać swojej codziennej pracy i nazywać to ćwiczeniem. W czasie pracy chodzi o wydajność, a nie ćwiczenia. O ćwiczeniu można mówić wtedy, gdy wykorzystujesz swoje umiejętności *poza* normalną pracą tylko po to, żeby te je utrwalic i rozwinać.

Co zatem dla programisty oznacza ćwiczenie? Na pierwszy rzut oka ta koncepcja jest trochę absurdalna. Ale wystarczy przez chwilę pomyśleć. Zastanów się, jak muzycy doskonala swoje rzemiosło. Wcale nie występując, tylko ćwicząc. A jak to robią? Między innymi mają specjalne ćwiczenia, które regularnie wykonują. Skale, etiudy i sekwencje. Powtarzają je w nieskończoność, ćwicząc swoje palce i umysły, próbując osiągnąć doskonałość w tym, co robią.

Co zatem mogą robić programiści w ramach ćwiczeń? W tej książce cały rozdział poświęciłem różnym rodzajom ćwiczeń, dlatego nie będę się tutaj rozpisywał. Jedną z technik, z których sam często korzystam, jest powtarzanie prostych ćwiczeń, takich jak Gra w kręgle (ang. *Bowling game*) lub Czynnik pierwszy (ang. *Prime factor*). Takie ćwiczenia nazywam *kata*. Istnieje spory zbiór takich kata do wyboru.

Kata najczęściej zdefiniowane jest jako prosty problem programistyczny, który należy rozwiązać, taki jak napisanie funkcji obliczającej czynniki pierwsze danej liczby całkowitej. Celem wykonywania kata nie jest wymyślenie sposobu rozwiązania problemu, ponieważ każdy zna już jego rozwiązanie. Celem jest tutaj wyćwiczenie swoich palców i umysłu.

Każdego dnia robię jedno lub dwa kata, często w ramach przygotowań do właściwej pracy. Mogę wykonać je w Javie, w Ruby albo w Clojure lub dowolnym innym języku, którego zasady w danym momencie chcę sobie przypomnieć. Wykorzystuję kata do poprawienia określonej umiejętności, takiej jak przypomnienie palcom różnych skrótów klawiszowych, albo wykonania określonych rodzajów refaktoryzacji.

O kata możesz myśleć jak o dziesięciominutowych rozgrzewkach o poranku i dziesięciominutowym rozciąganiu po południu.

## Współpraca

Kolejną dobrą metodą nauki jest współpraca z innymi ludźmi. Zawodowi programiści szczególnie starają się wspólnie programować, ćwiczyć, projektować i planować. W ten sposób wiele się nawzajem od siebie uczą, a poza tym wykonują zadania w znacznie krótszym czasie i z mniejszą liczbą błędów.

Nie oznacza to, że musisz 100% swojego czasu spędzać, pracując wspólnie z innymi. Równie ważny jest czas wykorzystywany samodzielnie. Jak bardzo bym lubił programowanie

---

w parze z innym kolegą, to brak możliwości samodzielnego popracowania od czasu do czasu sprawia, że źle się czuję.

## Nauczanie

Najlepszą metodą uczenia siebie jest nauczanie innych. Nic tak szybko nie wprowadzi do głowy nowych faktów i wartości jak przekazywanie ich ludziom, za których jesteśmy odpowiedzialni. Oznacza to, że zalety nauczania zdecydowanie leżą po stronie nauczającego.

Według tego samego schematu można powiedzieć, że nie ma lepszego sposobu na wprowadzenie nowej osoby do organizacji, niż usiąść razem z nią i pokazać jej wszystkie szczegóły. Zawodowcy przejmują osobistą odpowiedzialność za nauczanie młodych. Nie pozwalają młodzikom na nienadzorowane zabawy.

## Znaj swój fach

Zadaniem każdego zawodowego twórcy oprogramowania jest dokładne poznanie dziedziny i rozwiązań, które ma zaprogramować. Jeżeli tworzysz system księgowy, to należałoby znać się na księgowości. Jeżeli piszesz aplikację do zamawiania wycieczek, to dobrze byłoby znać się trochę na sprzedaży wycieczek. Nie musisz być ekspertem w tych dziedzinach, ale odpowiedni poziom wiedzy na ich temat na pewno jest wskazany.

Uruchamiając projekt w nowej dziedzinie, przeczytaj przynajmniej jedną poświęconą jej książkę. Porozmawiaj z klientem i użytkownikami o podstawach danej dziedziny. Trochę czasu spędź z ekspertami, próbując poznać ich zasady i wartości.

Przystąpienie do tworzenia kodu, gdy opieramy się jedynie na specyfikacji, bez próby zrozumienia, dlaczego ta specyfikacja pasuje do danego rodzaju działalności, jest najgorszym przykładem nieprofesjonalnego działania. Musisz dowiedzieć się na tyle dużo o tej działalności, żeby móc rozpoznawać i wskazywać błędy w specyfikacji.

## Identyfikuj się ze swoim pracodawcą lub klientem

Problemy Twojego pracodawcy są *Twoimi* problemami. Musisz poznać te problemy i starać się wspomagać ich rozwiązywanie. Rozwijając dany system, musisz spojrzeć na niego oczami pracodawcy i upewnić się, że rozwijane przez Ciebie funkcje na pewno będą zaspokajały potrzeby pracodawcy.

Programistom bardzo łatwo jest identyfikować się ze sobą nawzajem. Często przyjmują zatem względem pracodawcy postawę *my kontra oni*. Profesjonalisci starają się tego za wszelką cenę unikać.

## **Pokora**

Programowanie jest aktem tworzenia. Pisząc kod, tworzymy coś z niczego. Odważnie wymuszamy porządek w chaosie. Pewnie i szczegółowo nakazujemy maszynie określone zachowania, które zdefiniowane inaczej, mogłyby spowodować nieobliczalne straty. Oznacza to, że programowanie jest aktem gigantycznej arogancji.

Profesjonalisci wiedzą, że są aroganccy, i dlatego nie obnoszą się z fałszywą skromnością. Zawodowiec zna swoją pracę i czuje dumę, widząc swoje dzieła. Profesjonalista jest pewien swoich umiejętności i na tej podstawie podejmuje odważne, choć kalkulowane ryzyko. Zawodowiec nie będzie nieśmiały.

Mimo tego profesjonalista wie, że przyjdzie taki czas, kiedy się pomyli, jego ocena ryzyka okaże się błędna, a umiejętności niewystarczające. Spojrzy wtedy w lustro i zobaczy śmiejącego się do niego aroganckiego głupca.

Oznacza to, że gdy zawodowiec stanie się obiektem kpin, sam będzie się z nich śmiać. Nigdy nie będzie wykpiwał innych, ale przyjmie kpiny, na które zasłuży, a wyśmieje te niezasłużone. Nie będzie poniżać nikogo za popełnione błędy, ponieważ wie, że sam może wkrótce się pomylić.

Profesjonalista zna rozmiary swojej arogancji i wie, że los w końcu też ją zauważa i odpowiednio potraktuje. A gdy się to stanie, najlepsze, co będzie można zrobić, to skorzystać z rady Howarda: śmiać się.

## **Bibliografia**

[PPP2002]: Robert C. Martin, *Agile Software Development. Principles, Patterns, and Practices*, Upper Saddle River, NJ: Prentice Hall, 2002.

---

# KIEDY MÓWIĆ „NIE”<sup>2</sup>

---



*Zrób albo nie rób. Nie ma próbowania.*

— Yoda

Na początku lat 70. razem z dwoma moimi dziewiętnastoletnimi przyjaciółmi pracowaliśmy w firmie Teamsters z Chicago nad systemem księgowym działającym w czasie rzeczywistym, przeznaczonym dla firmy o nazwie ASC. Jeżeli komuś przypomina się w związku z nim nazwisko Jimmy'ego Hoffy, to dobrze. W roku 1971 lepiej było nie zadzierać z Teamsters.

Nasz system miał zostać uruchomiony określonego dnia. Z tą datą związana była cała góra pieniędzy. Nasz zespół pracował 60, 70, a nawet 80 godzin tygodniowo, byle tylko dotrzymać terminu.

Na tydzień przed terminem w końcu udało się nam złożyć system w całość. Musieliszy się jednak uporać z długą listą błędów i problemów, nad którą nadal wytrwale pracowaliśmy. Nie mieliśmy czasu na jedzenie, spanie, nie mówiąc już o myśleniu.

Frank, menedżer z ASC, był emerytowanym pułkownikiem Air Force. Był jednym z tych głośnych i bardzo bezpośrednich kierowników. Nie znosił sprzeciwu i każdy jego przejaw dusił brutalnie już w samym zarodku. Nasza trójka dziewiętnastolatków nie była w stanie nawet nawiązać z nim kontaktu wzrokowego.

Frank powiedział, że system ma być gotowy do wyznaczonego dnia. Nie było żadnej dyskusji. Wielki dzień nadjeźdie i system będzie gotowy. Kropka. Żadnych dyskusji. Bez odbioru.

Mój szef Bill był całkiem miłym człowiekiem. Z Frankiem pracował już od ładnych kilku lat i wiedział, na co można sobie z nim pozwolić, a co było wykluczone. Powiedział nam, że system zostanie uruchomiony wyznaczonego dnia, niezależnie od okoliczności.

A zatem system został uruchomiony. To była prawdziwa katastrofa.

Mieliszy tuzin półdupleksowych terminali o szybkości 300 bodów, które łączyły siedzibę firmy Teamsters w Chicago z naszym komputerem znajdującym się 50 kilometrów na północ od miasta. Każdy z tych terminali blokował się mniej więcej co 30 minut. Wiedzieliśmy już wcześniej o tym problemie, ale nigdy nie symulowaliśmy ruchu, który zaczęli generować pracownicy klienta.

Żeby było jeszcze gorzej, podczas drukowania arkuszy za pomocą teletekstów ASR-35, podłączonych do systemu liniami telefonicznymi o szybkości 110 bodów, zdarzały się blokady w samym środku arkusza.

Rozwiązaniem przy każdej takiej blokadzie było ponowne uruchomienie systemu. Oznaczało to, że każda osoba, której terminal jeszcze działał, musiała dokończyć pracę i się zatrzymać. Gdy już nikt nie pracował, mogli do nas zadzwonić z prośbą o ponowne uruchomienie. Osoba, której zdarzyła się blokada, musiała zaczynać swoją pracę od początku. A zdarzało się to kilka razy na godzinę.

Po kilku godzinach takiej zabawy menedżer biura Teamsters nakazał nam zamknąć system i nie uruchamiać go ponownie, dopóki nie będzie działał poprawnie. Już stracili pół dnia pracy i byli zmuszeni ponownie wprowadzać te same dane do starego systemu.

W całym budynku słyszać było krzyki i przekleństwa rzucane przez Franka. Trwało to całą wieczność. Później przyszedł do nas Bill w towarzystwie analityka systemowego Jalila z pytaniem, na kiedy możemy doprowadzić system do stanu stabilnego. Powiedziałem, że za cztery tygodnie.

Na ich twarzach pojawiły się przerażenie i determinacja. „To niemożliwe” — powiedzieli — „musi działać już w piątek”.

Odpowiedziałem: „Przecież w zeszłym tygodniu dopiero co udało się nam jako tak po składać system. Musimy pozbyć się wszystkich problemów i błędów, a na to potrzebujemy czterech tygodni”.

Niestety, Bill i Jalil byli niewzruszeni. „Nie, musi działać już w piątek. Możecie przynajmniej spróbować?”

I wtedy szef naszego zespołu powiedział: „No dobra, spróbujemy”.

Piątek był całkiem niezły wyborem, ponieważ weekendowe obciążenie systemu było o wiele mniejsze. Przed poniedziałkiem mogliśmy znaleźć i usunąć jeszcze więcej błędów. Mimo wszystko cały ten domek z kart zawalił się raz jeszcze. Problemy z zawieszaniem się terminali pojawiały się jeszcze raz lub dwa razy każdego dnia. Oprócz tego system trąpiły też inne problemy. Jednak powoli, w ciągu kilku tygodni doprowadziliśmy go do stanu, w którym skargi nie były już aż tak częste, a normalność wydawała się całkiem osiągalna.

Wtedy, tak jak wspomniałem już we wstępnie, wszyscy złożyliśmy wypowiedzenie. Firma miała teraz naprawdę poważny problem. Musiała zatrudnić nowych programistów, którzy mieli próbować zająć się ogromnym strumieniem problemów zgłaszanych przez klienta.

Kogo można było winić za tę sytuację? Z pewnością sposób bycia Franka stanowił jakiś problem. Jego ciągłe onieśmielanie rozmówcy sprawiało, że ciężko było mu usłyszeć prawdę. Z pewnością Bill i Jalil powinni byli bardziej naciskać Franka. Z pewnością szef zespołu programistów nie powinien był zgadzać się na piątkowy termin. Ostatecznie ja sam powiniensem był powiedzieć „nie”, zamiast ustawać się za szefem zespołu.

Profesjonalisci mówią prawdę niezależnie od sytuacji. Profesjonalisci mają odwagę powiedzieć „nie” swoim przełożonym.

Ale jak powiedzieć „nie” swojemu szefowi? W końcu to Twój szef! Czy nie należałooby robić tego, co szef nakazuje?

Nie. Nie, jeżeli uważasz się za profesjonalistę.

Niewolnik nie ma prawa powiedzieć „nie”. Robotnicy mogą mieć opory przed mówieniem „nie”. Ale od profesjonalistów *oczekuje się*, że będą mówić „nie”. Co więcej, dobrzy menedżerowie szukają ludzi, którzy mają charakter, żeby się sprzeciwiać. Tylko w ten sposób można naprawdę wiele osiągnąć.

## Przeciwstawne role

Jeden z recenzentów naprawdę nienawidził tego rozdziału. Powiedział, że z jego powodu prawie odłożył książkę na półkę. Tworzył już zespoły, w których nie było wrogich stosunków. Takie zespoły pracowały w harmonii, bez żadnych konfrontacji.

Cieszę się z sytuacji tego recenzenta, ale zastanawiam się, czy jego zespoły naprawdę były tak harmonijne, jak twierdzi. Jeżeli takie były, to zastanawiam się, czy były tak wydajne, jak mogłyby być. Z mojego doświadczenia wynika, że najtrudniejsze decyzje najlepiej podejmuje się w trakcie konfrontacji przeciwnych ról.

Menedżerowie są ludźmi, którym powierzono określone zadanie, i większość z nich doskonale wie, jak to zadanie wykonać. Część ich pracy polega na jak najagresywniejszej obronie wyznaczonych im celów.

Podobnie programiści — też są ludźmi, którzy również mają do wykonania pewne zadanie i większość z nich wie, jak się z niego dobrze wywiązać. Jeżeli są oni profesjonalistami, to będą starali się bronić swoich celów tak agresywnie, jak tylko będą mogli.

Gdy Twój kierownik mówi Ci, że strona logowania ma być gotowa na jutro, to stara się osiągać swoje własne cele. Wykonuje po prostu swoją pracę. Jeżeli doskonale wiesz, że przygotowanie strony logowania na jutro jest po prostu niemożliwe, to mówiąc: „OK, spróbuję”, zdecydowanie nie wykonujesz swojej pracy. Tę pracę wykonasz właściwie, tylko twierdząc, że to jest niemożliwe.

Czy jednak nie masz obowiązku wykonywania poleceń przełożonego? Wcale nie. Twój menedżer liczy właśnie na to, że będziesz bronić swoich celów tak agresywnie, jak on broni swoich. Tylko w ten sposób będziecie w stanie wypracować *rozwiązańe najlepsze z możliwych*.

Takim rozwiązańem jest cel, pod którym Ty i Twój kierownik możecie się podpisać. Cała sztuka polega na odnalezieniu takiego celu, a to zwykle wymaga negocjowania.

Czasami takie negocjacje mogą być całkiem przyjemne.

Mike: „Paula, ta strona logowania musi być gotowa na jutro”.

Paula: „O rany! Tak szybko? Ale OK, spróbuje”.

Mike: „Świetnie, wielkie dzięki!”.

To była całkiem miła rozmowa. Całkowity brak konfrontacji, a obie strony rozstały się z uśmiechem. Wspaniale.

Jednak obie strony zachowały się bardzo nieprofesjonalnie. Paula wie doskonale, że przygotowanie strony logowania zajmie jej trochę więcej niż jeden dzień, a zatem kłamie. Być może nawet nie myśli o tym w kategorii kłamstwa i naprawdę *ma zamiar spróbować*, bo ma choć cień nadziei, że uda jej się dotrzymać terminu. Jak by na to nie patrzeć, to nadal jest kłamstwo.

Jednakże Mike uznał słowo „spróbuje” za „tak”. To wyjątkowo głupie założenie. Powinien był wiedzieć, że Paula próbuje tylko uniknąć konfrontacji, dlatego miał pociągnąć tę sprawę

dalej, mówiąc: „Dobrze, widzę, że się wahasz. Na pewno uda ci się przygotować stronę do jutra?”.

A oto inna miła rozmowa.

Mike: „Paula, ta strona logowania musi być gotowa na jutro”.

Paula: „Przykro mi, Mike, ale to zajmie trochę więcej czasu”.

Mike: „W takim razie na kiedy będzie gotowa?”.

Paula: „Co powiesz na jakieś dwa tygodnie?”.

Mike: (wpisuje coś do notatnika) „OK, dzięki”.

Choć była całkiem miła, to jednocześnie była straszliwie niewłaściwa i potwornie nieprofesjonalna. Obie strony nawet nie starały się znaleźć lepszego rozwiązania. Zamiast pytać, czy dwa tygodnie będą w porządku, Paula powinna była odpowiedzieć bardziej asertywnie: „Oj, to zajmie jakieś dwa tygodnie”.

Z kolei Mike po prostu przyjął do wiadomości termin wyznaczony przez Paulę, tak jakby jego własne cele nie miały znaczenia. Można się zastanawiać, czy czasem nie poinformuje swojego szefa, że demo dla klienta zostanie opóźnione z powodu Pauli. Tego rodzaju pasywno-agresywne zachowanie jest całkowicie niemoralne.

W obu przedstawionych przypadkach żadna ze stron nie próbowała zdefiniować wspólnego celu. Nikt nie próbował poszukać najlepszego rozwiązania. Przyjrzyjmy się zatem tej rozmowie.

Mike: „Paula, ta strona logowania musi być gotowa na jutro”.

Paula: „Przykro mi, Mike, ale to robota na dwa tygodnie”.

Mike: „Dwa tygodnie? Architekci szacowali to na trzy dni, a minęło już pięć!”.

Paula: „To znaczy, że architekci się pomylii. Swoje szacunki przygotowali, jeszcze zanim marketing zakończył definiowanie wymagań systemowych. Roboty jest tyle, że zajmie mi to jeszcze przynajmniej dziesięć dni. Nie widziałeś moich szacunków w wiki?”.

Mike: (spogląda surowo i aż trzęsie się ze złości) „Paula! Nie mogę tego zaakceptować. Klienci chcą jutro zobaczyć demo i muszę im pokazać działającą stronę logowania”.

Paula: „Która część strony musi do jutra działać?”.

Mike: „Potrzebuję strony logowania. Muszę się jakoś zalogować do systemu”.

Paula: „OK, mogę dać ci szablon strony logowania, który pozwoli ci się zalogować.

Ta część już działa. Pamiętaj tylko, że nie sprawdza nazwy użytkownika ani hasła i nie prześle na maila zapomnianego hasła. Nie pojawi się wielki firmowy banner w stylu Time Square, a przycisk pomocy nie będzie szczególnie pomocny.

Nie zapiszą się ciasteczka zapamiętujące użytkownika na przyszłość, a przede wszystkim ten szablon w żaden sposób nie ograniczy ci dostępu do systemu. No, ale będziesz mógł się zalogować. Wystarczy?”.

Mike: „Na pewno będę mógł się zalogować?”.

Paula: „Owszem, będziesz mógł”.

Mike: „Paula, to świetnie! Ratujesz mi życie” (odchodzi, podskakując i krzycząc „yes!”)!

Udało im się znaleźć najlepsze rozwiązanie. Zrobili to, mówiąc najpierw „nie”, a potem starając się zdefiniować rozwiązanie zadowalające dla obu stron. Zachowali się jak prawdziwi zawodowcy. Rozmowa nie była całkiem przyjemna, pojawiło się w niej kilka zgrzytów, ale tego należało oczekiwac od dwóch osób asertywnie broniących celów, które nie są ze sobą zgodne.

## A co z „dlaczego”?

Możesz myśleć, że Paula powinna była wyjaśnić, *dlaczego* przygotowanie strony logowania zajmie dużo więcej czasu. Z mojego doświadczenia wynika jednak, że *powód* ma dużo mniejsze znaczenie od *faktu*. A faktem jest, że prace nad stroną logowania potrwały dwa tygodnie. Dlaczego potrwa to tak długo, jest już tylko mało znaczącym szczegółem.

Mimo to, znając powody, Mike mógłby zrozumieć, a przez to i zaakceptować sam fakt. Zgadza się. I w sytuacjach, w których Mike ma odpowiednią wiedzę techniczną oraz chęć zrozumienia, takie wyjaśnienia mogłyby być przydatne. Jednak Mike mógłby się też nie zgodzić z takimi wnioskami. Mógłby dojść do wniosku, że Paula wszystko robi źle. Mógłby stwierdzić, że wcale nie potrzeba jej tych wszystkich testów, kontroli, a poza tym krok 12. można śmiało pominąć. Prezentowanie zbyt wielu informacji może być zaproszeniem do mikrozarządzania.

## Wysokie stawki

Najmocniej należy sprzeciwiać się wtedy, gdy stawka jest najwyższa. Im wyższa stawka, tym większą siłę zyskuje Twoje „nie”.

To w ogóle nie powinno wymagać wyjaśnień. Jeżeli koszt pomyłki byłby tak duży, że zaleałoby od tego przetrwanie firmy, Twoim bezwzględnym obowiązkiem jest przekazanie menedżerom jak najdokładniejszych informacji. A to często oznacza sprzeciwianie się ich żądaniom.

Don (dyrektor działu rozwoju): „Zgodnie z naszymi aktualnymi szacunkami projekt Złota Kura powinien zostać ukończony za dwanaście tygodni z dokładnością do plus minus pięciu tygodni”.

Charles (prezes): (siedzi przez piętnaście sekund z coraz bardziej czerwoną twarzą)

„Chcesz przez to powiedzieć, że będziemy gotowi dopiero za siedemnaście tygodni?".

Don: „Tak, to możliwe".

Charles: (wstaje, sekundę później wstaje też Don) „Do jasnej cholery! Całość miała być gotowa trzy tygodnie temu! Klienci dzwonią do mnie codziennie i wypytyują, gdzie podziwia się ich system. *Nie* mam zamiaru informować ich, że będą musieli poczekać jeszcze cztery miesiące. Musisz się bardziej postarać"!

Don: „Charles, mówiłem ci już *trzy miesiące temu*, że po tych zwolnieniach będziemy potrzebować czterech dodatkowych miesięcy. Wyrzuciłeś ponad 20 procent mojego zespołu! Czy już wtedy informowałeś klientów, że projekt się opóźni?".

Charles: „Dobrze wiesz, że tego nie zrobiłem. Don, nie możemy sobie pozwolić na utratę tego zamówienia (milknie i blednie). Przecież bez niego siedzimy w głębokim bagnie. Chyba dobrze o tym wiesz. A teraz jeszcze to opóźnienie. Obawiam się... co ja mam powiedzieć zarządowi? (Powoli siada w swoim fotelu, próbując zebrać się w sobie). Naprawdę musisz się bardziej postarać".

Don: „Tutaj nic nie mogę zrobić. Przerabialiśmy to już. Klient nie chce ograniczyć projektu ani nie chce zaakceptować częściowych wydań. Chcą mieć jedną wielką instalację i zapomnieć o wszystkim. W tych warunkach nie da się zrobić nic szybciej. Po prostu się *nie da*".

Charles: „Cholera. Gdyby to twoja posada była zagrożona, pewnie nie byłbyś tak twardy".

Don: „Przykro mi, ale zwolnienie mnie w żaden sposób nie poprawi szacunków".

Charles: „Dobra, kończymy. Wracaj do swojego zespołu i pchaj ten projekt do przodu. Ja muszę wykonać kilka mało przyjemnych telefonów".

Oczywiście Charles powinien był poinformować klienta o nowych terminach już trzy miesiące temu, gdy tylko się o nich dowiedział. Przynajmniej teraz podjął właściwą decyzję i złapał za telefon (zarząd też musi poinformować). Jeżeli jednak Don nie obstawałby przy swoim, to te telefony z pewnością nie zostałyby wykonane.

## Gracz zespołowy

Wszyscy wielokrotnie słyszeliśmy, jak ważne jest, żeby być „graczem zespołowym”. Gracz zespołowy gra na swojej pozycji tak dobrze, jak tylko potrafi, i pomaga kolegom z zespołu, gdy tego potrzebują. Gracz zespołowy często wymienia się informacjami, współpracuje z całym zespołem, a swoje obowiązki wykonuje tak dobrze, jak tylko potrafi.

Graczem zespołowym nie jest jednak ktoś, kto na wszystko się zgadza. Przyjrzymy się kolejnej rozmowie.

Paula: „Mike! Mam dla ciebie kilka szacunków. Zespół zdecydował, że będziemy w stanie dostarczyć wersję demonstracyjną za mniej więcej osiem tygodni. Plus minus tydzień”.

Mike: „Chyba żartujesz. Według oficjalnego planu demo ma być gotowe za sześć tygodni”.

Paula: „I nie zapytaliście nas o zdanie? Daj spokój, przecież nie możesz tak wymuszać terminów”.

Mike: „Przykro mi, już się stało”.

Paula: (wzdycha) „OK, zapytam w zespole, czy możemy coś zrobić, żeby cokolwiek dostarczyć w ciągu sześciu tygodni. Ale na cały system nie ma najmniejszych szans. Pewnych funkcji będzie brakowało, a i dane będą niekompletne”.

Mike: „Paula, klienci chcą zobaczyć w pełni funkcjonalne demo”.

Paula: „Zapomnij. Na to nie ma najmniejszych szans”.

Mike: „Cholera! Dobra, zobacz, co się da zrobić, i do jutra mnie poinformuj”.

Paula: „Tyle mogę ci obiecać”.

Mike: „Nie ma żadnej możliwości, żeby skrócić prace? Może da się pracować jakoś sprytniej i szybciej?”.

Paula: „Wszyscy działamy już całkiem sprytnie. Doskonale wiemy, jak się za to zabrać, i z szacunków wynika, że całość będzie gotowa za osiem lub dziewięć tygodni. Ale nie za sześć”.

Mike: „Może jakieś nadgodziny?”.

Paula: „To tylko wszystko spowolni. Pamiętasz, jaki powstał bałagan, gdy ostatnio zarządzono pracę w nadgodzinach?”.

Mike: „No... tak, ale tym razem wcale tak nie musi być”.

Paula: „Nie ludź się. Będzie tak jak ostatnio. Możesz mi zaufać. Demo będzie gotowe za osiem lub dziewięć tygodni. Nie za sześć”.

Mike: „Dobra, prześlij mi szczegółowy plan, ale zastanów się jeszcze, jak załatwić to w sześć tygodni. Wiem, że coś się wasm uda wymyślić”.

Paula: „Mike! Nie! Nie uda się! Mogę ci dostarczyć plan na sześć tygodni, ale w demo będzie brakowało pewnych funkcji i danych. Tylko taką masz możliwość”.

Mike: „OK, OK. I tak wiem, że jak się postaracie, to będziecie w stanie zdziałać cuda” (odchodzi, kręcząc głową).

Później, w trakcie zebrania dyrektorów w sprawie strategii.

Don: „Mike, jak wiesz, klient zawita do nas za sześć tygodni i będzie chciał obejrzeć demo swojego systemu. Mówili już, że chcieliby, aby wszystko do tego czasu działało”.

Mike: „Nie ma sprawy. Będziemy gotowi! Mój zespół urabia się po łokcie, żeby ze wszystkim zdążyć. Zapewne będzie trzeba zrobić trochę nadgodzin i zastosować parę sprytnych rozwiązań, ale wierzę, że nam się uda”.

Don: „Miło słyszeć, że mamy w firmie grupę wspaniałych graczy zespołowych”.

Kto w tym scenariuszu był *prawdziwym* graczem zespołowym? Paula występowała w imieniu swojego zespołu i najlepiej, jak potrafiła, prezentowała, co można, a czego nie można oczekiwac. Agresywnie broniła swoich pozycji pomimo pochlebstw i podstępów Mike'a. Mike z kolei grał w zespole jednoosobowym. Działał tylko w swoim interesie. Zdecydowanie nie należał do zespołu Pauli, ponieważ podjął za nią zobowiązanie, które ona jasno opisała jako nie do wykonania. Nie gra też w zespole Dona (choć z pewnością by się z tym nie zgodził), ponieważ okłamał go bez zmrużenia oka.

Dlaczego zatem Mike robi takie rzeczy? Chce, żeby Don uznał go za gracza zespołowego. Jednocześnie wierzy, że będzie w stanie w ten lub inny sposób przymusić Paulę, by spróbowała zrobić wszystko w sześć tygodni. Mike wcale nie jest zły. Jest pewien własnych umiejętności, za pomocą których narzuca ludziom swoją wolę.

## Próbowanie

Najgorszą odpowiedzią, jakiej Paula mogłaby udzielić w wyniku manipulacji Mike'a, byłoby: „OK, spróbujemy”. Nie chciałbym tu brzmieć jak Yoda, ale w tym przypadku ma on całkowitą rację. *Nie ma próbowania*.

Możliwe, że ta idea Ci się nie podoba. Być może sądzisz, że *próbowanie* jest czymś pozytywnym. W końcu czy Kolumb odkryłby Amerykę, gdyby nie próbował?

Niestety, słowo *próbować* ma wiele definicji. Definicja, z którą mam najwięcej problemów, jest taka: „Dołożyć starań”. Jakich dodatkowych starań można wymagać od Pauli, żeby dostarczyła system na czas? Jeżeli dałoby się zwiększyć starania w tym zakresie, to znaczyłoby, że zespół Pauli i ona sama nie wykorzystywali wszystkich swoich możliwości. Musieli ukrywać pewne rezerwy na wszelki wypadek<sup>1</sup>.

Obietnica spróbowania jest jednocześnie przyznaniem się, że nie wykorzystujesz wszystkich swoich możliwości; że masz jeszcze w zanadru coś, czego można użyć. Obietnica spróbowania jest też przyznaniem, że cel jest osiągalny, o ile zostaną zastosowane te dodatkowe środki. Co więcej, jest też zobowiązaniem się do wykorzystania tych środków, aby osiągnąć cel. Oznacza to, że obiecując spróbować, zobowiązujesz się do osiągnięcia wyznaczonych celów. Na obiecującego spada zatem cały ciężar. Jeżeli wasze „próbowanie” nie przyniesie pożądanych skutków, to znaczy, że zawiedliście.

<sup>1</sup> Tak jak Kurak Leghorn: „Na takie właśnie przypadki zawsze numeruję swoje pióra”.

Masz jakiś dodatkowy rezeruar energii na czarną godzinę? Jeżeli wykorzystasz te rezerwy, to uda się zrealizować plan? Czy może obiecując spróbować, najzwyczajniej w świecie podpisujesz wyrok na siebie i zespół?

Obiecując spróbować, obieczysz też zmienić dotychczasowe plany. W końcu te stare były niewystarczające. Obiecując spróbować, przyznajesz, że istnieje jakiś nowy plan. Jak zatem on wygląda? Jakie zmiany wprowadzisz w swoim zachowaniu? Czym będą różniły się Twoje działania, ponieważ teraz zaczynasz „próbować”?

Jeżeli nie masz żadnego nowego planu, jeżeli nie wprowadzisz zmiany w swoim działaniu, jeżeli będziesz pracować dokładnie tak jak do tej pory, zanim zaczęło się „próbowanie”, to na czym to próbowanie ma niby polegać?

Jeżeli jednak nie masz żadnych rezerw energetycznych, jeżeli nie masz nowego planu, jeżeli nie masz zamiaru zmieniać swoich zachowań i jeżeli masz całkowitą pewność co do swojego pierwotnego planu, to obietnica „spróbowania” będzie absolutnie nieszczera. Można będzie nazwać ją *kłamstwem*. Jedynym powodem, dla którego składasz taką obietnicę, jest próba zachowania dobrej opinii i uniknięcia konfrontacji.

Postawa Pauli jest tu zdecydowanie lepsza. Cały czas przypominała Mike'owi, że szacunek zespołu nie jest całkowicie dokładny. Mówiła ciągle o „ośmiu lub dziewięciu tygodniach”. Wyraziła tę niepewność i nigdy się z niej nie wycofała. Nigdy nie zasugerowała, że istnieją jakieś rezerwy, nowy plan albo jakakolwiek zmiana zachowań, które mogłyby zmniejszyć stopień tej niepewności.

Trzy tygodnie później...

Mike: „Paula, termin dema już za trzy tygodnie, a klienci chcą zobaczyć działającą funkcję przesyłania plików”.

Paula: „Hola! Tego nie było na uzgodnionej liście funkcji”.

Mike: „No wiem, ale klient się tego domaga”.

Paula: „Dobra, to znaczy jednak, że w demie nie będą działały centralne logowanie albo kopia zapasowa”.

Mike: „Wykluczone! Te funkcje również chcą zobaczyć”.

Paula: „Czyli chcą zobaczyć działające wszystkie funkcje? To mi chcesz powiedzieć? Przecież już dawno temu mówiłem ci, że nie ma na to najmniejszych szans”.

Mike: „Przykro mi, Paula, ale klient jest nieugięty. Chcą zobaczyć wszystkie funkcje”.

Paula: „Ale nie zobaczą. Nie ma takiej możliwości”.

Mike: „Daj spokój, nie możecie chociaż spróbować?”.

Paula: „Mike, próbować to ja mogę lewitacji. Mogę próbować zmienić ołów w złoto. Mogę próbować przepływać żabką Atlantyk. Jak myślisz, uda mi się?”.

Mike: „Gadasz od rzeczy. Przecież nie żądam *niemożliwego*”.

Paula: „Owszem, żądasz”.

(Mike uśmiecha się, kręci głową i odchodzi).

Mike: „Mimo wszystko wierzę w ciebie. Wiem, że mnie nie zawiedziesz”.

Paula: (mówiąc do pleców Mike'a) „Mike, w jakim świecie ty żyjesz? To nie zapowiada *niczego dobrego*”.

(Mike tylko macha ręką, nawet się nie odwracając).

## Pasywna agresja

Paula musi podjąć bardzo ciekawą decyzję. Podejrzewa ona, że Mike nie przekazuje Donowi informacji o jej szacunkach. Może zatem pozwolić Mike'owi iść dalej tą drogą, aż do przepaści. Może sprawdzić, czy w dokumentacji znajdują się wszystkie przygotowane przez nią dokumenty, tak żeby w momencie rozpoczęcia się awantury mogła udowodnić, *co i kiedy* mówiła Mike'owi. Mówiąc krótko, pozwoli Mike'owi przygotować własną szubienicę.

Mожет быть запобiec катастрофе и порозмавлять безпосередньо з Доном. Очевидно, то до єї ризикового поступку, але на цьому рівні погано відповідає стратегії засобами. Якщо потяг товарного поїзда просто підійде до вас, а тільки ви знаєте, що він зможе побачити, то ви можете або відкрито відповісти: „Увага! Потяг! Підійти до тільки!”.

Dwa dni później...

Paula: „Mike, przekazałeś już Donowi moje szacunki? Mam nadzieję, że poinformował już klienta, iż w demie nie będzie działała funkcja przesyłania plików”.

Mike: „Przecież powiedziałaś, że to dla mnie załatwisz”.

Paula: „Nie, Mike, nic takiego nie powiedziałam. Mówiłam, że to niemożliwe. Tutaj masz kopię wiadomości, którą wysłałam ci po naszej rozmowie”.

Mike: „No tak, ale zrozumiałem, że mimo wszystko będziesz próbować”.

Paula: „Już o tym rozmawialiśmy. Pamiętasz? Ołów, złoto...”

Mike: (wzdycha) „Paula, no musisz to załatwić. Po prostu musisz. Proszę, porusz niebo i ziemię, ale ta funkcja musi działać na czas. Jesteś mi to winna”.

Paula: „Mike, mylisz się. Nic nie jestem ci winna. Jedyne, co muszę zrobić, to przekazać tę informację Donowi, skoro ty nie jesteś w stanie”.

Mike: „To byłoby przekroczeniem twoich kompetencji. Nie ośmielisz się”.

Paula: „Nie chcę tego, ale nie dajesz mi innego wyboru”.

Mike: „Och, Paula...”

Paula: „Mike, pogódź się z tym, że pewne funkcje *nie będą* gotowe na dzień uruchomienia dema. Przyjmij to w końcu do wiadomości. Przestań namawiać mnie do ciężej pracy. I zapomnij o tym, że w jakiś sposób uda się mi wyciągnąć królika z kapelusza. Nie masz wyboru. Musisz o tym powiedzieć Donowi, i to najlepiej *dzisiaj*”.

Mike: (robi wielkie oczy) „Dzisiaj?”.

Paula: „Tak, Mike. Dzisiaj, ponieważ na jutro planuję spotkanie z tobą i Donem w celu omówienia funkcji, które mają działać w demie. Jeżeli to spotkanie nie odbędzie się jutro, to będę musiała sama udać się do Dona. A tutaj masz kopię wiadomości wyjaśniającej tę sytuację”.

Mike: „Zabezpieczasz sobie tyły?”.

Paula: „Mike, próbuję zabezpieczyć tyły nam *oboju*. Wyobrażasz sobie, jakie mielibyśmy kłopoty, gdyby przyszedł tu klient i chciał obejrzeć demo z niedokończonymi funkcjami?”.

Co stanie się z Paulą i Mikiem? Pozwolę Ci wymyślić możliwe rozwiązania. Ważne jest, że Paula zachowała się bardzo profesjonalnie. Zawsze we właściwy sposób i we właściwym momencie odmawiała Mike’owi. Powiedziała „nie”, gdy była namawiana do zmiany swoich szacunków. Odmawiała mimo różnych manipulacji, pochlebstw i prośb. I co najważniejsze, konsekwentnie zaprzeczała złudzeniom i pasywnym działaniom Mike’a. Paula była prawdziwym graczem zespołowym. Mike potrzebował pomocy, a ona zrobiła wszystko, co w jej mocy, żeby mu pomóc.

## Koszta przytakiwania

Zwykle chcemy po prostu powiedzieć „tak”. I rzeczywiście zdrowo działające zespoły szukają jakiekolwiek możliwości, żeby powiedzieć „tak”. Menedżerowie i programiści współpracujący w ramach takich zespołów negocują ze sobą tak długo, aż ustalą plan działań, na który obie strony mogą się zgodzić.

Jak jednak widzieliśmy, czasami jedyną metodą na uzyskanie *zdrowego „tak”* jest brak oporów przed mówieniem „nie”.

Zastanówmy się nad poniższą opowieścią, którą John Blanco opublikował na swoim blogu<sup>2</sup>. Prezentuję ją tutaj za jego zgodą. Czytając tekst, spróbuj zastanowić się, kiedy i jak powinien był powiedzieć „nie”.

---

<sup>2</sup> <http://raptureinvenice.com/?p=63>.

## Czy dobry kod jest nieosiągalny?

Po osiągnięciu wieku lat nastu decydujesz, że chcesz zostać programistą. W szkole średniej uczysz się pisania programów, wykorzystując zasady programowania obiektowego. Po dostaniu się na studia wykorzystujesz te wyuczone zasady we wszystkich nowych dziedzinach, takich jak sztuczna inteligencja lub grafika trójwymiarowa.

I gdy w końcu dostajesz się do środowiska zawodowców, rozpoczynasz niekończącą się próbę tworzenia wysokiej jakości, łatwego w obsłudze, po prostu „perfekcyjnego” kodu, który przetrwa próbę czasu.

Wysokiej jakości. He, he! Niezły dowcip.

Uważam się za szczęściarza i *uwielbiam* wzorce projektowe. Lubię czytać teorie opisujące perfekcyjny kod. Nie mam żadnych problemów z prowadzeniem ciągnących się godzinami dyskusji, dlaczego wybór mojego partnera dotyczący hierarchii dziedziczenia jest niewłaściwy — w końcu konstrukcja MA jest w wielu przypadkach znacznie lepsza od JEST. Ale ostatnio coś nie daje mi spokoju i cały czas się zastanawiam...

Czy podczas tworzenia nowoczesnego oprogramowania niemożliwe jest uzyskanie dobrego kodu?

## Typowa propozycja projektu

Jako kontraktowy programista pracujący na pełny etat (a czasem też na pół etatu) spędżam całe dnie (oraz noce) na tworzeniu aplikacji mobilnych dla różnych klientów. I przez całe lata takiej pracy nauczyłem się, że wymagania związane z pracą dla klienta uniemożliwiają mi przygotowanie aplikacji o jakości, jaką chciałbym osiągnąć.

Zanim zacznę, pozwólcie, że wyjaśnię: nie wynika to z tego, że nigdy się odpowiednio nie starałem. Uwielbiam temat czystego kodu. Nie znam nikogo, kto równie mocno jak ja stara się uzyskać doskonały projekt oprogramowania. Niestety, samo wykonanie nie jest już tak idealne, choć wcale nie z powodów, które mogą was teraz przyjść na myśl.

Opowiem wam historię.

Pod koniec zeszłego roku całkiem znana firma ogłosiła przetarg na przygotowanie dla niej aplikacji. To ogromna sieć handlowa, której w tej historii nadam fikcyjną nazwę Gorilla Mart. Twierdzili, że chcą pojawić się w końcu na iPhonech, i dlatego chcieliby, żeby przygotować dla nich aplikację jeszcze przed czarnym piątkiem<sup>3</sup>. Jaki w tym haczyk? Jest pierwszy listopada. To znaczy, że na przygotowanie aplikacji pozostały niecałe cztery tygodnie. I dodatkowo Apple potrzebuje dwóch tygodni na zatwierdzenie zgłoszonego programu (wspomnienie starych dobrych dni). Czyli... zaraz, cała aplikacja ma zostać napisana w.... DWA TYGODNIE??!!

Tak. Mamy dwa tygodnie na napisanie aplikacji. Pech chciał, że wygraliśmy przetarg. (W tym biznesie ważna jest wielkość klienta). I teraz musimy się wywiązać ze zobowiązania.

„Będzie dobrze” — mówi jeden z dyrektorów Gorilla Martu. — „Aplikacja ma być prosta.

Musi wyświetlać tylko kilka produktów z naszego katalogu i pozwalać na wyszukiwanie sklepów. Taką funkcję mamy już na naszej stronie. Dostarczmy też potrzebną grafikę.

Prawdopodobnie możecie wszystko, jak to się mówi... zakodować na twardo!”

<sup>3</sup> Czarny piątek — potoczna nazwa dnia przypadającego po Święcie Dziękczyznienia.

W USA jest to dla sprzedawców często najbardziej dochodowy dzień w ciągu całego roku.

Wtrąca się dyrektor numer 2: „I jeszcze parę kuponów, które klient będzie mógł pokazać przy kasie. Sama aplikacja będzie do jednorazowego użytku. Za chwilę przejdziemy do fazy drugiej i przygotujemy coś większego od podstaw”.

I wtedy się zaczyna. Mimo lat ciągłego przypominania sobie, że każda funkcja wymyślona przez klienta będzie o wiele bardziej skomplikowana w realizacji, niż wynika z opisu, i tak się zgadzasz. Naprawdę wierzysz, że da się to zrobić w dwa tygodnie. Tak! Zrobimy to! Tym razem będzie inaczej! To tylko parę obrazków i wywołanie usługi, żeby pobrać pozycję sklepu. XML! Bułka z masłem. Zrobimy! Ale się nakręciłem! Do roboty!

A wystarcza tylko jeden dzień, żeby rzeczywistość sprowadziła cię z powrotem na ziemię.

Ja: „Możesz powiedzieć mi, jak wywołać tę usługę sieciową, żeby dostać pozycję sklepu?”.

Klient: „Jaką usługę sieciową?”.

Ja: „....”.

To się naprawdę wydarzyło. Ich usługa wyznaczania pozycji sklepu, dostępna w prawym górnym rogu ich strony internetowej, nie była usługą sieciową. Była generowana przez kod napisany w Javie. Zapomnij o jakimś API. Niet. A żeby było gorzej, kod jest serwowany przez strategicznego partnera Gorilla Martu.

Witamy w strasznym świecie „stron trzecich”.

Z punktu widzenia klienta „strona trzecia” jest podobna do Angeliny Jolie. Mimo obietnic, że będziesz mógł poprowadzić z nią światłą konwersację przy wystawnej kolacji, z cieniem nadzieję na coś ciekawszego później... przykro mi, nic z tego. Pozostaje ci fantazjowanie w czasie, gdy sam musisz zająć się sprawami.

W moim przypadku jedyne, co udało mi się wydusić z Gorilla Martu, to była aktualna lista sklepów w postaci pliku Excela. Musiałem od zera pisać kod wyznaczający pozycje sklepów.

A po południu tego samego dnia założyli nam podwójnego nelsona. Chcieli, żeby produkty i dane do kuponów były dostępne online, tak żeby mogli je co tydzień zmieniać. Tyle widzieliśmy z kodowania na twardo! Dwa tygodnie na napisanie aplikacji dla iPhone'a zmieniły się w dwa tygodnie na napisanie aplikacji dla iPhone'a, aplikacji serwera w PHP i wykonanie integracji. Że co? Chcą, żebymy jeszcze przejął na siebie zadania QA?

Żeby zrekompensować tak krótki czas, musimy zatem szybciej tworzyć kod. Zapomnij o fabryce abstrakcyjnej. Zamiast kompozycji użyj wielkiej, ohydnej pętli. Nie ma czasu!

Dobry kod jest w tych warunkach nieosiągalny.

## **Dwa tygodnie do celu**

Mówię wam, te dwa tygodnie były naprawdę paskudne. Dodatkowo dwa dni można było skreślić z powodu całodniowych spotkań dotyczących mojego następnego projektu. (To jeszcze bardziej podkreśla, jak krótki mieliśmy termin realizacji). Ostatecznie miałem zaledwie osiem dni na wszystkie prace. W pierwszym tygodniu pracowałem 74 godziny, a w drugim... O Boże! Nawet nie pamiętam, po prostu wypaliło mi te synapsy, ale to chyba akurat jest w porządku.

Tych osiem dni spędziłem na tworzeniu kodu w przyspieszonym tempie. Wykorzystałem wszystkie dostępne mi narzędzia, pozwalające szybciej zakończyć prace: kopuj i wklej (czyli ponowne wykorzystanie kodu), magiczne liczby (uniakalem pracy związanego z definiowaniem stałych i potem ciągłym ich wpisywaniem) i absolutnie żadnych testów jednostkowych! (Komu są w takim momencie potrzebne czerwone kropki? To tylko demotywuje!)

Powstały kod był naprawdę paskudny i nigdy nie miałem czasu, żeby go poprawić. Jeżeli jednak wziąć pod uwagę czas wykonania, to wyszedł całkiem sprawnie, a poza tym i tak miał zostać zaraz wyrzucony. Coś wam to przypomina? Tylko czekajcie, będzie jeszcze lepiej.

Kiedy wprowadzałem w aplikacji ostatnie poprawki (te dotyczyły wyłącznie kodu serwerowego), zacząłem przeglądać całość powstałego kodu i pomyślałem sobie, że może jednak było warto.

Mimo wszystko aplikacja była gotowa, a mnie udało się przeżyć!

„Cześć, właśnie zatrudniliśmy Boba. Jest bardzo zajęty i nie może sam do ciebie zadzwonić, ale twierdzi, że powinniśmy żądać od użytkowników podania adresu e-mail w zamian za kupon. Nie widział jeszcze aplikacji, ale uważa, że to byłby świetny pomysł. Chcemy też mieć system raportów, żeby te adresy pobierać z serwera. Taki ładny i niezbyt kosztowny. (Zaraz, ten ostatni tekst był z Monty Pythona). A skoro mówimy o kuponach, to powinny się przedawniać po zdefiniowanej przez nas liczbie dni. I jeszcze...”

Wróćmy jeszcze do podstaw. Co wiemy na temat tego, jak powinien wyglądać dobry kod? Dobry kod powinien łatwo się rozbudowywać i być łatwy w konserwacji. Powinien aż prosić się o wprowadzanie zmian. Powinno się go czytać jak prozę. Cóż, to niestety nie był dobry kod.

Inna rzecz. Jeżeli chcesz być lepszym programistą, wciąż musisz o tym pamiętać: klienci zawsze będą przesuwać termin. Zawsze będą chcieli mieć więcej funkcji. Zawsze będą chcieli wprowadzać zmiany, ale już w późnej fazie prac. A oto formuła opisująca to, czego można się spodziewać:

(liczba dyrektorów)<sup>2</sup>  
 + 2 \* liczba nowych dyrektorów  
 + liczba dzieci Boba  
 = DNI DODANE W OSTATNIEJ CHWILI

Trzeba jednak przyznać, że dyrektorzy to całkiem porządni ludzie. Tak sądzą. Muszą dbać o swoje rodziny (zakładając, że szatan pozwolił im je założyć). Chcą, żeby aplikacja okazała się sukcesem (czas na awans). Problem polega na tym, że wszyscy oni chcą być bezpośrednio odpowiedzialni za ten sukces. Gdy już wszystko zostanie powiedziane i zrobione, chcą wskazać określoną funkcję lub decyzję projektową, którą mogliby nazwać swoją.

A wracając do opowieści, dodaliśmy do projektu jeszcze dwa dni i wprowadziliśmy funkcję pobierania adresów e-mail. Zaraz potem padłem z wyczerpania.

## Klient nigdy nie przejmuje się tak jak Ty

Klienci, mimo tego, jak się zachowują, mimo pozornego pośpiechu, tak naprawdę nie przejmują się, czy aplikacja powstanie na czas. Tego popołudnia, kiedy ponownie zakończyłem prace nad aplikacją, wysłałem e-mail z końcową wersją programu do wszystkich udziałowców, dyrektorów

(grrr!), menedżerów itd. „GOTOWE! OTO WERSJA 1.0! CHWALCIE ME IMIĘ!” Kliknąłem przycisk *Wyślij* i z uśmiechem opadłem na krzesło. Zacząłem fantazjować, jak to cała firma poniesie mnie na rękach w kierunku 42 Ulicy, gdzie zostanę koronowany na „Najlepszego Programistę Wszech Czasów”. A przynajmniej moją podobiznę umieszczą na wszystkich swoich reklamach.

Zadziwiające, że nie podzielali mojej wizji. Tak naprawdę to nie wiem, o czym myśleli, bo nie dostałem żadnej odpowiedzi. Absolutnie żadnej. Okazało się, że wszyscy w Gorilla Marcie byli tak zabiegani, że zajmowali się już następnym projektem.

Myślisz, że kłamię? No to jedziemy dalej. Wysłałem podanie do sklepu Apple bez podawania opisu aplikacji. Prosiłem Gorilla Mart o przygotowanie takiego, ale się do mnie nie odezwały, a ja nie mogłem już dłużej czekać. (Czytaliście akapity powyżej?) Prosiłem raz jeszcze. I jeszcze raz. Wspierało mnie w tym nasze własne kierownictwo. Dwa razy dostałem odpowiedź o treści: „Czego właściwie potrzebujesz?”. **MUSZĘ MIEĆ OPIS APLIKACJI!**

Tydzień później Apple zaczęło testować aplikację. Zwykle jest to czas wielkiej radości, ale tym razem czułem tylko śmiertelny strach. Tak jak oczekiwaliem, później tego dnia aplikacja została odrzucona. Był to najsmutniejszy i najgorszy z możliwych powodów odrzucenia aplikacji, jaki mogę sobie wyobrazić: „Aplikacja nie ma opisu”. Działa doskonale, ale nie ma opisu. I z tego powodu Gorilla Mart nie miał aplikacji gotowej na czarny piątek. Byłem troszeczkę wściekły.

Opuściłem swoją rodzinę, pracując w dwutygodniowym supersprincie, ale w Gorilla Marcie nikt nie miał czasu na napisanie opisu, mimo tygodnia nagabywań. Przesłali nam go godzinę po odrzuceniu aplikacji. Najwyraźniej to był sygnał, żeby wziąć się do pracy.

Jeżeli wtedy byłem wściekły, to półtora tygodnia później zagotowałem się. Wyobrażacie sobie, że nadal nie dostarczyli nam rzeczywistych danych? Produkty i kupony na serwerze były fałszywkami. Wymyślone. Kod kuponu to było 1234567890. No wiecie, dmuchawce, latawce.

I tego doniosłego dnia o poranku sprawdziłem portal, a na nim DOSTĘPNA BYŁA TA APLIKACJA! Z wszystkimi fałszywymi danymi itp.! Zawyłem w najwyższym przerażeniu, zacząłem dzwonić, do kogo tylko się dało, i krzyczeć: „**POTRZEBUJĘ DANYCH!**”. Kobieta po drugiej stronie linii spytała, czy potrzeba mi policji, czy strażaków, więc rozłączyłem się z numerem 911. Następnie jednak zadzwoniłem do Gorilla Martu i ponownie krzyczałem: „**POTRZEBUJĘ DANYCH!**”. Nigdy nie zapomnę ich odpowiedzi:

„O, cześć, John! Mamy teraz nowego wiceprezesa i zdecydowaliśmy się nie publikować tej aplikacji. Jeżeli mógłbyś, to wycofaj ją z AppStore”.

Ostatecznie okazało się, że przynajmniej 11 osób zarejestrowało swoje adresy e-mail w bazie danych, co znaczyło, że jakieś 11 osób mogło wejść do Gorilla Martu z fałszywym kuponem na iPhone. Oj, mogło się to źle skończyć.

Po tym wszystkim mogłem stwierdzić, że klient w jednym nie kłamał. Kod był do wyrzucenia. Problem polegał na tym, że nigdy nie został wykorzystany.

## **Wyniki? Pośpiech w realizacji, powolna publikacja**

Lekcja wynikająca z tej opowieści jest taka, że udziałowcy, niezależnie od tego, czy w postaci zewnętrznego klienta, czy też wewnętrznej kadry zarządzającej, wymyślili, jak można zmusić

programistę do szybkiego tworzenia kodu. Skutecznego? Nie. Szybkiego. Oj, tak! A całość działa następująco:

- **Powiedz programiście, że aplikacja będzie prosta.** W ten sposób wpuszczasz zespół w fałszywe nastawienie. Poza tym programiści zaczną też wcześniej pracować, a wtedy...
- **Dodawaj funkcje, twierdząc, że to zespół nie przewidział ich niezbędności.** W tym przypadku treści zakodowane na twardo wymagałyby ciągłego aktualizowania aplikacji, przy każdej ich zmianie. Jak mogłem tego nie zauważyc? Zauważylem, ale wcześniej dostałem fałszywą obietnicę. Na tym to polega. Albo klient zatrudni „kogoś nowego”, kto zauważy, że w aplikacji brakuje czegoś oczywistego. Czy jak pewnego dnia klient powie, że zatrudnili właśnie Steve'a Jobsa, to do aplikacji trzeba będzie dodać trochę alchemii? A potem...
- **Przesuwaj termin. I jeszcze raz, i jeszcze.** Programiści pracują najszybciej i najskuteczniej (ale też są najbardziej podatni na robienie błędów, ale kto się tym przejmuję?), gdy do terminu oddania projektu zostaje już tylko kilka dni. No to po co mówić im, że można przesunąć termin, skoro są aż tak produktywni? Wykorzystaj to! I w ten sposób dodaje się jeszcze parę dni, a potem tydzień, a ty cały czas pracujesz w 20-godzinnych zmianach, żeby zdążyć na czas. To jak z osłem i marchewką, tylko że osioł jest znacznie lepiej traktowany.

To rozwiązanie doskonałe. Czy można ich winić za to, że wierzą w jego skuteczność? Oni nie widzą tego ohydnie paskudnego kodu. I tak dzieje się za każdym razem, pomimo tego, jakie są wyniki tych praktyk.

W ekonomii globalnej, w której korporacje sprawują władzę nad wszechmocnym dolarem, a podniesienie ceny akcji wymaga zwolnienia części pracowników, zmuszenia pozostałych do nadgodzin i zlecenia prac za granicę, taka metoda minimalizowania kosztów pracy programistów sprawia, że dobry kod jest niepotrzebny. Jeżeli nie będziemy ostrożni, to zostaniemy poproszeni/zmuszeni/wmanewrowani w pisanie kodu o dwa razy większej objętości w czasie o połowę krótszym.

## Kod niemożliwy

W przytoczonej opowieści John pyta: „Czy dobry kod jest niemożliwy do osiągnięcia?”. Ale tak naprawdę pytanie brzmi: „Czy profesjonalizm jest niemożliwy do osiągnięcia?”. W końcu w tej opowieści o wszelkich dysfunkcjach ucierpiał nie tylko kod. W tej całej przygodzie stracili też rodzina Johna, jego pracodawca, klient oraz użytkownicy. Tutaj straty ponieśli wszyscy<sup>4</sup>. A ponieśli je z powodu braku profesjonalizmu.

Kto zatem zachowywał się nieprofesjonalnie? John jasno stwierdza, że winę ponoszą dyrektorzy Gorilla Martu. W całej jego opowieści całkiem jasno przedstawił ich niewłaściwe zachowania. Ale czy to zachowanie rzeczywiście było złe? Osobiście sądę inaczej.

Zarząd Gorilla Martu chciał mieć możliwość udostępnienia aplikacji dla iPhone na czarny piątek. Zdecydowali się zapłacić za taką aplikację. Znaleźli kogoś, kto podjął się tego zadania. Dlaczego więc zrzucać winę na nich?

---

<sup>4</sup> Być może wyjątkiem jest bezpośredni pracodawca Johna, ale myślę, że i on poniósł tu stratę.

Owszem, wystąpiło kilka niewypałów w komunikacji. Najwyraźniej kadra zarządzająca nie miała pojęcia, czym jest usługa sieciowa. Poza tym pojawiły się standardowe problemy wynikające z faktu, że jedna część wielkiej korporacji nie wie, co robi inna część. Ale tego wszystkiego można było oczekiwąć. Nawet John przyznaje to, mówiąc: „Mimo lat ciągłego przypominania sobie, że każda funkcja wymyślona przez klienta będzie o wiele bardziej skomplikowana w realizacji, niż wynika z opisu...”.

Skoro zatem winna całej tej sytuacji nie jest firma Gorilla Mart, to kto jest winny?

Być może winą należy obarczyć pracodawcę Johna? Co prawda, John nie powiedział tego jasno, ale ze stwierdzenia: „W tym biznesie ważna jest wielkość klienta” można wiele wyczytać. Czy zatem pracodawca Johna złożył firmie Gorilla Mart obietnicę, której nie mógł dotrzymać? Czy naciskali na Johna (pośrednio lub bezpośrednio), żeby tej obietnicy jednak dotrzymać? John nic o tym nie mówi, więc zostają nam tylko domysły.

Biorąc to wszystko pod uwagę, jaką w tym wszystkim odpowiedzialność miał John? Uważam, że w tym przypadku winę za całą tę sytuację ponosi właśnie on. To on zaakceptował pierwotny dwutygodniowy termin, choć doskonale wiedział, że projekty zwykle są znacznie bardziej złożone, niż początkowo się wydaje. To on zgodził się na żądanie dopisania jeszcze serwera PHP. To on zaakceptował funkcje rejestracji adresów e-mail oraz terminu ważności kuponów. To John pracował po 20 godzin dziennie i 90 godzin tygodniowo. To on opuścił na ten czas swoją rodzinę i porzucił normalne życie, tylko po to, żeby zdążyć na czas.

Dlaczego zatem podjął się tego zadania? Mówi nam o tym bez żadnego skrępowania: „Kliknąłm przycisk *Wyślij* i z uśmiechem opadłem na krzesło. Zacząłem fantazjować, jak to cała firma poniesie mnie na rękach w kierunku 42 Ulicy, gdzie zostanę koronowany na »Najlepszego Programistę Wszech Czasów«”. John chciał po prostu zostać bohaterem. Zauważył swoją szansę na wielką nagrodę i postanowił z niej skorzystać. Pochylił się, żeby podnieść tę złotą monetę.

Profesjonalisci często stają się bohaterami, ale nie dlatego, że bardzo się o to starają. Stają się bohaterami, ponieważ dobrze wywiązują się ze swoich obowiązków, wykonując zadania na czas i w ramach budżetu. Próbuje zdobyć sławę zbawiciela, John nie zachowywał się profesjonalnie.

John nie powinien był się zgadzać na pierwotny termin dwóch tygodni. A jeżeli już by się na to zgodził, to powinien był protestować, gdy tylko dowiedział się, że nie ma żadnej usługi sieciowej. Powinien nie zgadzać się na żądania dodania funkcji rejestracji adresów e-mail i przedawniania kuponów. Powinien był blokować wszystko to, co wymagałoby tak straszliwych nadgodzin i innych poświęceń.

Przede wszystkim jednak John powinien był zaprzeczyć swojej własnej decyzji, że jedyną metodą zrealizowania projektu na czas jest wytworzenie bałaganierskiego kodu. Zauważ, co John powiedział o dobrym kodzie i testach jednostkowych: „Żeby zrekompensować

tak krótki czas, musimy zatem szybciej tworzyć kod. Zapomnij o fabryce abstrakcyjnej. Zamiast kompozycji użyj wielkiej, ohydnej pętli. Nie ma czasu!”.

I jeszcze to:

„Tych osiem dni spędziłem na tworzeniu kodu w przyspieszonym tempie. Wykorzystałem wszystkie dostępne mi narzędzia, pozwalające szybciej zakończyć prace: kopij i wklej (czyli ponowne wykorzystanie kodu), magiczne liczby (uniakalem pracy związanej z definiowaniem stałych i potem ciągłym ich wpisywaniem) i absolutnie żadnych testów jednostkowych! (Komu są w takim momencie potrzebne czerwone kropki? To tylko demotywuje!)”.

Zatwierdzenie tych wszystkich decyzji było prawdziwym powodem całej katastrofy. John założył, że jedynym sposobem na sukces jest w tym przypadku nieprofesjonalne zachowanie, dlatego zasłużył sobie na odpowiednią nagrodę.

To wszystko może brzmieć bardzo okrutnie, ale wcale nie o to mi chodzi. W poprzednim rozdziale opowiadałem już, że podobne błędy popełniałem również w swojej karierze. Pokusa bohaterskiego „rozwiązania problemu” jest po prostu ogromna. Trzeba sobie jednak uświadomić, że celowe porzucanie naszej profesjonalnej dyscypliny *nie* jest właściwą metodą rozwiązywania problemów. Porzucenie tej dyscypliny jest pierwszym krokiem do padnięcia w nowe kłopoty.

W ten sposób mogę ostatecznie odpowiedzieć na główne pytania postawione przez Johna.

„Czy dobry kod jest nieosiągalny? Czy profesjonalizm jest niemożliwy?”

Odpowiedź: Uważam, że *nie*.



---

# KIEDY MÓWIĆ „TAK”<sup>3</sup>

---



Czy wiesz, że to ja wynalazłem pocztę głosową? To prawda! Właściwie to patent na pocztę głosową otrzymały trzy osoby: Ken Finder, Jerry Fitzpatrick i ja. Było to na początku lat 80., w czasie gdy pracowaliśmy dla firmy o nazwie Teradyne. Nasz prezes przydzielił nam zadanie wymyślenia zupełnie nowego produktu, a my wymyśliliśmy „elektronicznego recepcjonistę”, w skrócie ER.

Dzisiaj chyba każdy już wie, czym jest ER. To jedna z tych straszliwych maszyn odbierających telefony w firmach i zadających te wszystkie głupie pytania, na które musisz odpowiadać, naciskając klawisze telefonu (*For English, press 1*).

Nasz recepcjonista odbierał telefon i prosił o wprowadzenie nazwiska osoby, z którą chcesz rozmawiać. Prosił też o podanie swojego nazwiska i przystępował do realizacji połączenia. Najpierw jednak zapowiadał w firmie dane połączenie i pytał, czy ma ono zostać odebrane. Jeżeli otrzymał potwierdzenie, to posłusznie łączył i się wycofywał.

Naszemu recepcjonistie można było powiedzieć, gdzie się jest. Można mu było podać kilka numerów, z którymi ma próbować się łączyć. Dzięki temu nawet jeżeli było się w innym biurze, to recepcjonista i tak mógł cię znaleźć. Podobnie jeżeli było się w domu albo w innym mieście. Jeżeli jednak mimo tych wszystkich zabiegów ER nie mógł cię dopaść, to pozwalał na pozostawienie wiadomości. To właśnie tutaj pojawiła się poczta głosowa.

Co ciekawe, firma Teradyne nie była w stanie wymyślić sposobu na skuteczną sprzedaż ER. Projekt przekroczył budżet i został włączony do czegoś, co umieliśmy dobrze sprzedawać — CDS, czyli *Craft Dispatch System*, który służył do przydzielania zadań serwisantom linii telefonicznych. Poza tym Teradyne wycofała wniosek patentowy, nawet nas o tym nie informując (!). Aktualny właściciel patentu złożył wniosek trzy miesiące po nas (!!)<sup>1</sup>.

Długo po tym, jak system ER został włączony do CDS, ale też na długo, zanim dowiedziałem się o wycofaniu wniosku patentowego, czekałem na naszego prezesa, siedząc na drzewie. Przed budynkiem firmy rósł ogromny stary dąb, na który się wspiąłem, i czekałem na przyjazd jaguara. Prezesa w końcu spotkałem przed drzwiami firmy i zapytałem, czy ma dla mnie kilka minut. Zaprosił mnie do biura.

Powiedziałem mu, że naprawdę musimy ponownie uruchomić projekt ER, ponieważ jestem pewien, że przyniesie on nam spore pieniądze. Zaskoczył mnie, mówiąc: „Dobrze, Bob. Przygotuj jakiś plan i pokaż mi, jak na tym zarobić. Jeżeli zdołasz mnie przekonać, to ponownie uruchomię projekt”.

Tego się zupełnie nie spodziewałem. Zakładałem, że usłyszę coś w rodzaju: „Masz rację, Bob. Zaraz zreanimuję cały projekt i postaram się wykombinować, jak na tym zarobić”. Ale nie. Cały ten ciężar zrzucił na moje barki. Na dodatek miałem do tego wszystkiego dość ambiwalentny stosunek. W końcu byłem tylko programistą, a nie marketingowcem. Chciałem pracować nad projektem ER, a nie przejmować odpowiedzialność za zyski lub straty. Nie chciałem jednak okazywać swoich uczuć. Podziękowałem mu zatem i wyszedłem z biura, mówiąc:

„Dzięki, Russ. Chyba... się tego podejmę”.

W tym momencie pozwól, że zapoznam Cię z Royem Osherovem, który powie Ci, jak żałosna była moja odpowiedź.

---

<sup>1</sup> Nie chodzi o to, że ten patent miał dla mnie znaczenie finansowe. Zgodnie z umową o pracę sprzedalem go firmie Teradyne za jednego dolara (ale go nie dostałem).

# Język zobowiązania

Autor: Roy Oshero.

Powiedz. Zamierzaj. Zrób.

Oto trzy elementy niezbędne do podjęcia zobowiązania:

1. *Mówisz, że to zrobisz.*
2. *Zamierzasz to zrobić.*
3. *Faktycznie to robisz.*

Jednak często spotykamy ludzi (to oczywiście nie my), którzy nigdy nie są w stanie przejść przez wszystkie trzy etapy.

- **Zapytaj gościa od IT**, dlaczego sieć działa tak wolno, a odpowie: „No tak, zdecydowanie potrzeba nam nowych routerów”. I już wiesz, że w tej materii nic się nigdy nie zmieni.
- **Zapytaj członka swojego zespołu**, czy przeprowadzi ręczne testy, zanim wprowadzi kod do repozytorium, a odpowie: „No pewnie. Mam nadzieję, że skończę to jeszcze dzisiaj”. I jakoś tak czujesz, że jutro trzeba będzie spytać, czy przed wrzuceniem kodu do repozytorium wykonał jakiekolwiek testy.
- **Twój szef** wchodzi do pokoju i mamrocze: „Musimy działać szybciej”. A Ty wiesz, że tak naprawdę chodzi mu o to, że to TY masz działać szybciej. On nie ma zamiaru się z tego powodu przemęczać.

Jest niewiele osób, które mówiąc o czymś, zamierzają to zrobić, a potem rzeczywiście zabierają się do roboty. Są też tacy ludzie, którzy mówiąc o czymś, rzeczywiście zamierzają się tym zająć, ale w końcu i tak nic nie robią. Ale najwięcej jest osób obiecujących różne rzeczy, które tak naprawdę nawet nie zamierzają się nimi zająć. Zdarzyło Ci się słyszeć: „Rany, naprawdę muszę zacząć się odchudzać” i wiedzieć, że ta osoba nigdy nic z tym nie zrobi? Takie rzeczy dzieją się cały czas.

Dlaczego ciągle towarzyszy nam to dziwne uczucie, że ludzie raczej nie są zaangażowani w realizację swoich planów?

Co gorsza, często może nas zawodzić nasza własna intuicja. Czasami *chcemy uwierzyć*, że ktoś naprawdę zamierza zrobić to, o czym opowiada, mimo że tak naprawdę tylko opowiada. *Chcemy wierzyć* programiście, który zapędzony w kozi róg, twierdzi, że zadanie zaplanowane na dwa tygodnie skończy w ciągu jednego tygodnia. Mimo wszystko nie powinniśmy dawać takim zapewnieniom wiary.

Zamiast ufać instynktom, powinniśmy stosować pewne sztuczki językowe, aby sprawdzić, czy rozmówca rzeczywiście zamierza zrobić to, o czym opowiada. A zmieniając sposób

mówienia, możemy zająć się 1. i 2. punktem z powyższej listy. Kiedy mówimy, że się czegoś podejmiemy, i rzeczywiście zamierzamy to zrobić.

## **Rozpoznawanie braku zaangażowania**

Należy zwracać uwagę na język używany podczas *podejmowania się* jakiegoś zadania, ponieważ może on zdradzać zakończenie tej historii. Szczególnie chodzi tu o szukanie określonych słów, które są wypowiadane. Jeżeli nie jesteśmy w stanie znaleźć tych małych magicznych słówek, to jest całkiem możliwe, że nie angażujemy się w to, o czym mówimy, albo nie wierzymy, że realizacja jest możliwa.

Oto kilka przykładów słów i wyrażeń, na które trzeba zwracać uwagę, ponieważ mogą być zwiastunami braku zaangażowania:

- **Trzeba by / Musiałby.** „Trzeba by się tym zająć”. „Musiałbym zrzucić parę kilo”. „Ktoś musiałby się tym zająć”.
- **Mam nadzieję / Dobrze by było.** „Mam nadzieję zrobić to na jutro”. „Mam nadzieję, że się jeszcze spotkamy”. „Dobrze by było znaleźć na to czas”. „Dobrze by było mieć szybszy komputer”.
- **Powinniśmy** (bez ciągu dalszego zawierającego „ja”). „Powinniśmy się od czasu do czasu spotykać”. „Powinniśmy to skończyć”.

Wystarczy zacząć szukać tych słów, a okaże się, że znajdujesz je w niemal wszystkich wysłuchanych wypowiedziach, a nawet w tym, co mówisz innym.

Zauważysz, że bardzo się staramy, aby nie przyjąć na siebie odpowiedzialności za cokolwiek.

A to *nie jest* w porządku, jeżeli Ty lub ktokolwiek inny jesteście uzależnieni w swojej pracy od takich obietnic. Pierwszy krok został już zrobiony. Umiesz już wykrywać brak zobowiązania w otoczeniu i w sobie.

Wiemy już, jak brzmi brak zaangażowania. Jak jednak rozpoznać prawdziwe zaangażowanie?

## **Jak brzmi zaangażowanie?**

Elementem wspólnym wszystkich wyrażeń przedstawionych powyżej jest założenie, że te sprawy nie leżą w „mojej” gestii albo odpowiedzialność za nie nie dotyczy mnie osobiście. W każdym z tych przypadków ludzie zachowują się tak, jakby byli *ofiarami* danej sytuacji, a nie jej panami.

Prawda jest jednak taka, że Ty *osobiście ZAWSZE* masz coś pod *swoim* panowaniem, a zatem zawsze będzie coś, do czego możesz się całkowicie zobowiązać.

Składnikami pozwalającymi na rozpoznanie prawdziwego zaangażowania są zdania brzmiące mniej więcej tak: Ja (...) do (...), na przykład: „Ja to zrobię do wtorku”.

Co jest takiego ważnego w tym zdaniu? *To, że podajesz w nim информацию o czymś, co TY zrobisz w określonym przez Ciebie czasie.* Nie mówisz tu o nikim innym, tylko o sobie. Mówisz tu o konkretnym *działaniu*, którego się *podejmujesz*. Nie *chyba* to zrobisz albo *może się tym zajmiesz*. Ty to *na pewno* zrobisz.

Nie istnieje żadna metoda wykręcenia się z takiego zobowiązania. Mówisz, że to zrobisz, i teraz możliwy jest jeden z dwóch wyników — albo to zrobisz, albo nie. Jeżeli nie osiągniesz zamierzonego celu, to ludzie mogą pociągnąć Cię do odpowiedzialności. *Nie będziesz się z tego powodu dobrze czuć.* Poczujesz się *niezręcznie*, mówiąc komuś, że nie udało Ci się tego zrobić (jeżeli ten ktoś słyszał wcześniej Twoją obietnicę).

Przerażające.

Przymajesz na siebie pełną odpowiedzialność za coś i robisz to przed pewną grupą ludzi, a przynajmniej przed jedną osobą. To nie to samo co gadanie do siebie przed lustrem albo przed ekranem komputera. Tym razem to Ty stoisz przed inną osobą i twierdzisz, że wykonasz to zadanie. To się nazywa początek zobowiązania. Stawiasz się w sytuacji, która zmusza Cię do podjęcia działania.

Zmieniasz stosowany przez siebie język na język zobowiązań, co pomaga w przejściu kolejnych dwóch etapów: zamierzeń i realizacji.

Istnieje jednak wiele powodów, dla których możesz tak naprawdę nie zamierzać nic robić albo całkiem pominąć realizację, ale i na to można znaleźć rozwiązanie.

### **Nie udało się, ponieważ osoba X nie wywiązała się ze swoich zobowiązań**

Zobowiązać można się jedynie do tych rzeczy, które ma się pod *całkowitą kontrolą*. Na przykład jeżeli Twoim celem jest zakończenie prac nad modułem, które uzależnione są od współpracy z innym zespołem, to nie możesz zobowiązać się do ukończenia modułu i integracji z tym innym zespołem. Możesz jednak zobowiązać się do wykonania określonych działań, które przybliżą Cię do tego celu. Możesz:

- Posiedzieć godzinę z Grzegorzem z zespołu infrastruktury, żeby poznać wszystkie zależności.
- Przygotować interfejs odzwierciedlający zależności Twojego modułu od infrastruktury innego zespołu.
- Spotykać się przynajmniej trzy razy w tygodniu z nadzorcą komplikacji, aby się upewnić, że Twoje zmiany dobrze komponują się w firmowym systemie komplikacji.
- Przygotować swoją osobistą komplikację, która będzie uruchamiała testy integracyjne tworzonego modułu.

Widzisz różnicę?

Jeżeli rezultat jest uzależniony od kogoś innego, to lepiej zobowiązywać się do konkretnych działań, które przybliżą Cię do ostatecznego celu.

### **Nie udało się, ponieważ wcale nie było pewne, że to się da zrobić**

Jeżeli czegoś nie da się zrobić, to mimo wszystko możesz zobowiązać się do wykonania działań przybliżających Cię do celu. Jednym z tych działań może być nawet sprawdzenie, czy dana operacja jest w ogóle wykonalna.

Zamiast zobowiązywać się do poprawienia wszystkich 25 błędów przed oficjalnym wydaniem oprogramowania (co może nie być możliwe), możesz zobowiązać się do poniższych działań, które z pewnością przybliżą Cię do ostatecznego celu:

- Przejrzeć wszystkie 25 błędów i spróbować je odtworzyć.
- Posiedzieć z testerami, którzy znaleźli te błędy, żeby zobaczyć, jak udało im się je wywołać.
- Poświęcić w tym tygodniu cały dostępny czas na próby naprawienia poszczególnych błędów.

### **Nie dało się tego zrobić, bo czasami po prostu nie nadążam**

To też się zdarza. Czasami dzieją się rzeczy nieoczekiwane; takie jest życie. Ale mimo to chcemy sprostać oczekiwaniom. W takim wypadku należy zmieniać te oczekiwania *tak wcześnie, jak tylko jest to możliwe*.

Jeżeli nie jesteś w stanie dotrzymać zobowiązania, to najważniejsze jest jak najwcześniej podniesienie czerwonej flagi.

Im wcześniej wskażesz problem udziałowcom, tym więcej czasu będzie miał zespół na to, żeby zatrzymać się i ponownie ocenić wszystkie podejmowane działania, a następnie zdecydować, czy cokolwiek można zrobić lub zmienić (na przykład priorytety). W ten sposób możesz jeszcze dotrzymać swojego zobowiązania albo je zmienić, dopasowując do nowych warunków.

Oto kilka przykładów:

- Jeżeli zaplanujesz południowe spotkanie z kolegą w kawiarni w centrum miasta, ale utkniesz w korku, to możesz przypuszczać, że nie uda Ci się dotrzeć do kawiarni na czas. Możesz zatem zadzwonić do kolegi odpowiednio wcześniej i poinformować go o zaistniałej sytuacji. Być może uda się Wam znaleźć bliższe miejsce na spotkanie albo po prostu je przesuniecie.
- Jeżeli zobowiąziesz się do poprawienia błędu, który początkowo wyglądał całkiem niegroźnie, ale potem okazuje się, że jest znacznie bardziej złożony i kłopotliwy, to możesz

od razu podnieść czerwoną flagę. Zespół może wtedy podjąć decyzję o działaniach koniecznych do wykonania tego zobowiązania (działanie w parze, burza mózgów lub inne) albo po prostu zmienić priorytety i przenieść Cię do prac nad innym błędem.

W tym wszystkim ważne jest to, że jeżeli nie powiesz nikomu o zaistniałym problemie, to nie dasz nikomu szansy na udzielenie Ci pomocy niezbędnej do wywiązania się ze zobowiązania.

## **Podsumowanie**

Wytwarzanie języka zobowiązań może wydawać się przerażające, ale pomaga to rozwiązywać wiele problemów z komunikacją, z którymi programiści stykają się każdego dnia — niewłaściwymi szacunkami, niedopasowanymi terminami i nieporozumieniami w bezpośredniej komunikacji. Dzięki temu językowi będą Cię traktować jak poważnego zawodowca, który zawsze dotrzymuje danego słowa. A to chyba najlepsza reputacja, jaką można mieć.

~~~

## **Naucz się, jak mówić „tak”**

Poprosiłem Roya o pozwolenie na opublikowanie tego artykułu, ponieważ udało mu się we mnie coś poruszyć. Cały czas nauczałem o tym, w jaki sposób należy odmawiać. Ale również ważne jest to, żeby nauczyć się mówić „tak”.

## **Druga strona „próbowania”**

Wyobraźmy sobie, że Peter jest odpowiedzialny za pewne modyfikacje w systemie ocen. Oszacował on, że niezbędne poprawki zajmą mu pięć do sześciu dni. Oprócz tego twierdzi, że na napisanie dokumentacji wszystkich tych zmian będzie potrzebować kilku godzin. W poniedziałek rano jego menedżer Marge pyta o stan prac.

Marge: „Peter, uda ci się skończyć poprawki w systemie ocen do piątku?”.

Peter: „To się chyba da zrobić”.

Marge: „Same poprawki kodu czy już z dokumentacją?”.

Peter: „Mogę spróbować zdążyć z dokumentacją”.

Być może Marge nie słyszy niezdecydowania w potwierdzeniach Petera, ale on na pewno nie definiuje tu twardych zobowiązań. Marge oczekuje na swoje pytania potwierdzenia lub zaprzeczenia, podczas gdy wypowiedzi Petera są raczej niejasne.

Zauważ nadużycie słowa „próbować”. W poprzednim rozdziale podałem definicję tego słowa oznaczającą „dodatkowy nakład pracy”. W tym przypadku Peter stosuje definicję „może, a może nie”.

Zdecydowanie lepiej byłoby, gdyby Peter odpowiadał tak:

Marge: „Peter, uda ci się skończyć poprawki w systemie ocen do piątku?”.

Peter: „Prawdopodobnie, ale może się to przeciągnąć do poniedziałku”.

Marge: „Same poprawki kodu czy już z dokumentacją?”.

Peter: „Na dokumentację potrzebuję dodatkowych kilku godzin, dlatego jest szansa, że będzie gotowa w poniedziałek, ale nie wykluczam, że prace przeciągną się do wtorku”.

W tym przypadku wypowiedzi Petera są bardziej szczerze. Opisuje przed Marge swoją niepewność. Dzięki temu Marge może tę niepewność uwzględnić w swoich planach. Choć wcale nie musi.

## **Dyscyplina zobowiązań**

Marge: „Peter, potrzebuję tutaj jasnej odpowiedzi. Czy do piątku uda ci się wprowadzić wszystkie zmiany do systemu ocen i napisać do nich dokumentację?”.

Nic nie stoi na przeszkodzie, żeby Marge swoje pytanie sformułowała w ten sposób. Musi pilnować planu, dlatego potrzebuje jasnej odpowiedzi na temat zakończenia prac w piątek. Jak powinien odpowiedzieć Peter?

Peter: „W takiej sytuacji muszę odpowiedzieć: nie. Jestem pewien, że zmiany i dokumentację będę miał gotowe najpóźniej we wtorek”.

Marge: „Czyli mogę zapisać wtorek jako koniec prac?”.

Peter: „Owszem, na wtorek wszystko będzie gotowe”.

Co jednak, jeżeli Marge będzie potrzebowała wprowadzenia zmian i dokumentacji już w piątek?

Marge: „Peter, ten wtorek to dla mnie prawdziwy problem. Willy z dokumentacji technicznej będzie wolny już w poniedziałek. Zaplanował już pięć dni na przygotowanie instrukcji dla użytkownika. Jeżeli w poniedziałek nie będę miała gotowej dokumentacji systemu ocen, to nie uda mu się na czas napisać instrukcji. Czy możesz najpierw napisać dokumentację?”.

Peter: „Nie bardzo. Najpierw muszę wprowadzić modyfikacje w systemie, ponieważ dokumentacja jest generowana na podstawie wyników naszych testów”.

---

Marge: „Nie ma jakiegoś sposobu, żeby udało ci się skończyć modyfikacje i dokumentację przed poniedziałkiem rano?”.

Teraz to Peter musi podjąć decyzję. Istnieje pewna szansa, że zdąży wprowadzić zmiany do systemu ocen już w piątek, a przed pójściem do domu na weekend może nawet uda mu się przygotować dokumentację. *Móglby* też popracować kilka godzin w sobotę, jeżeli całość zajęłaby więcej czasu, niż zakładał. Co zatem powiedzieć Marge?

Peter: „Noooo, jest szansa, że zdążę wszystko przygotować na poniedziałek rano, jeżeli zrobiłbym kilka nadgodzin w sobotę”.

Czy to rozwiązuje problem Marge? Nie, to tylko zwiększa prawdopodobieństwo sukcesu, i to właśnie powinien powiedzieć jej Peter.

Marge: „Czy mogę zatem liczyć na poniedziałek rano?”.

Peter: „Możliwe, choć nie mam całkowitej pewności”.

Dla Marge to może być za mało.

Marge: „Peter, naprawdę potrzebuję tu konkretów. Czy istnieje jakikolwiek sposób na to, żebyś ze wszystkim zdążył na poniedziałek rano?”.

Petera może kusić, żeby w tym miejscu porzucić swoją dyscyplinę. Być może udałoby się mu przygotować wszystko szybciej, jeżeli pominąłby pisanie testów. Być może udałoby się mu wykonać zadania wcześniej, jeżeli zapomniałby o refaktoryzacji kodu. Być może wszystko poszłoby szybciej, jeżeli nie przeprowadziłby wszystkich testów regresyjnych.

To właśnie w tym miejscu zawodowcy kreślą grubą kreskę. Po pierwsze Peter myli się we wszystkich tych przypuszczeniach. *Nie zrobi* nic szybciej, jeżeli pominie pisanie testów. Pominiecie refaktoryzacji kodu *nie przyspieszy* pracy. *Nie skróci* czasu pracy, jeżeli odpuści sobie pełne testy regresyjne. Lata doświadczeń nauczyły go tego, że porzucenie normalnej dyscypliny tylko nas spowalnia.

Po drugie jako profesjonalista jest zobowiązany do zachowania pewnych standardów. Jego kod musi być przetestowany, czyli musi mieć napisane testy. Jego kod musi być czysty. A na dodatek Peter musi mieć pewność, że nie zepsuł niczego w istniejącym systemie.

Peter jako zawodowiec już wcześniej zobowiązał się do zachowania tych standardów. Wszystkie pozostałe zobowiązania muszą się zatem do tych standardów dopasowywać. Dlatego właśnie cała ta linia rozumowania musi zostać w tym miejscu przerwana.

Peter: „Przykro mi, ale nie ma sposobu na to, żeby z całą pewnością mógł zakończyć w terminie krótszym niż do wtorku. Przykro mi, że to psuje twój plan, ale taka jest rzeczywistość. Nic na to nie poradzę”.

Marge: „Cholera. A tak chciałam wcześniej skreślić ten punkt z listy. Jesteś pewny?”.

Peter: „Jestem absolutnie pewny, że całość może się przeciągnąć do wtorkowego popołudnia”.

Marge: „OK, chyba muszę pogadać z Willym, czy nie przestawi swojego planu”.

W tym przypadku Marge zaakceptowała odpowiedź Petera i zaczęła szukać innych rozwiązań. Ale co będzie, jeżeli Marge nie ma już żadnego asa w rękawie? Co zrobić, jeżeli Peter jest jej ostatnią nadzieję?

Marge: „Peter, proszę. Wiem, że to spora prośba, ale naprawdę te prace muszą być zakończone przed poniedziałkiem rano. Mam nóż na gardle. Na pewno nie da się nic poradzić?”.

Teraz Peter zaczyna myśleć o zrobieniu całej górki nadgodzin i poświęceniu na to zadanie sporej części weekendu. Musi jednak zdawać sobie sprawę z własnej wytrzymałości i rezerw energii. Łatwo można *powiedzieć*, że wszystko zostanie zrobione w ciągu weekendu, ale realizacja tych prac jest znacznie trudniejsza i wymaga dodatkowych nakładów, aby zachowana była odpowiednia jakość pracy.

Profesjonalisci znają granice swoich możliwości. Wiedzą, ile nadgodzin mogą skutecznie przepracować, i zdają sobie sprawę, jakie będą ich koszta.

W tym przypadku Peter czuje się na tyle pewnie, że kilka dodatkowych godzin w ciągu tygodnia i jeszcze parę w trakcie weekendu wystarczy do zakończenia prac.

Peter: „Dobra, powiem ci coś. Zadzwonię do domu i zapytam, czy zniosą trochę moich nadgodzin. Jeżeli nie będzie protestów, to zrobię te poprawki do poniedziałku rano. Nawet w poniedziałek przyjdę wcześniej, by się upewnić, że Willy nie będzie miał problemów. Ale potem wrócę do domu i nie pojawię się aż do środy. Zgoda?”.

Bardzo rozsądna propozycja. Peter wie, że będzie w stanie wprowadzić wszystkie zmiany i napisać ich dokumentację, jeżeli zrobi kilka nadgodzin. Wie też, że po tym wszystkim przez parę dni nie będzie się do niczego nadawał.

## **Wnioski**

Zawodowcy wcale nie muszą się zgadzać na wszystko, co się im proponuje. Powinni jednak bardzo się starać, szukając wszelkich możliwości znalezienia zadowalającego wszystkich rozwiązania. Jeżeli jednak profesjonalista mówi „tak”, to używa przy tym języka zobowiązania, tak żeby nie było żadnych wątpliwości co do zakresu danej obietnicy.

---

# 4 KODOWANIE

---



W poprzedniej książce<sup>1</sup> napisałem całkiem sporo o strukturze i naturze *czystego kodu*. W tym rozdziale będę omawiać sam *akt* tworzenia kodu oraz kontekst otaczający ten akt.

Gdy miałem 18 lat, całkiem nieźle szło mi pisanie na klawiaturze, ale nadal musiałem patrzeć na klawisze. Nie byłem w stanie pisać bezwzrokowo. Któregoś wieczoru spędziłem kilka godzin nad klawiaturą komputera IBM 029, starając się nie patrzeć na swoje palce w czasie, gdy wpisywałem program, który zapisałem na kilku formularzach. Po zakończeniu wprowadzania sprawdziłem każdą kartę z osobna i wyrzuciłem te, które zawierały błędy.

---

<sup>1</sup> [Martin09].

Początkowo zdarzało mi się całkiem sporo błędów, ale pod wieczór mogłem zapisywać karty niemal perfekcyjnie. W ciągu długiej nocy przekonałem się, że pisanie bezwzrokowe wymaga przede wszystkim pewności siebie. Moje palce wiedziały, gdzie są poszczególne klawisze, i musiałem tylko zyskać pewność, że nie popełniam błędów. Jedną z rzeczy, które mi bardzo przy tym pomogły, było to, że dokładnie czułem, kiedy popełniam błąd. Pod wieczór od razu wiedziałem, kiedy popełniłem błąd, i mogłem zaraz wyrzucić kartę bez patrzenia na nią.

Umiejętność wyczuwania własnych błędów jest niezwykle ważna. Nie tylko podczas pisania na klawiaturze, ale w każdym elemencie. Ten dodatkowy zmysł oznacza, że bardzo szybko zamkasz pętlę sprzężenia zwrotnego i jeszcze szybciej jesteś w stanie uczyć się na swoich błędach. Od czasu tego dnia spędzonego nad klawiaturą dwudziestki dziewiątki wiele się nauczyłem i pogłębiłem swoją wiedzę. Za każdym razem przekonywałem się, że kluczem do sukcesu są pewność siebie i umiejętność wyczuwania błędów.

W tym rozdziale opiszę mój osobisty zbiór zasad i reguł dotyczących tworzenia kodu. Te reguły i zasady nie odnoszą się do samego kodu, ale raczej do mojego zachowania, nastawienia i humoru podczas jego pisania. Opisuję mój mentalny, moralny i emocjonalny kontekst pisania kodu. To właśnie one są podstawą mojej pewności siebie i umiejętności wyczuwania błędów.

Z pewnością nie zgodzisz się ze wszystkim, co tutaj napiszę, to w końcu sprawy bardzo osobiste. Co więcej, całkiem prawdopodobne jest, że gwałtownie zaoponujesz przeciwko niektórym moim zasadom. To zupełnie normalne. Nie mają być one prawdami absolutnymi dla nikogo poza mną. Są tylko moją własną metodą profesjonalnego tworzenia kodu.

Być może analizując moje środowisko tworzenia kodu, będziesz w stanie lepiej zrozumieć całą zawartość tej książki.

## **Przygotowanie**

Tworzenie kodu jest bardzo wyczerpującą czynnością i ogromnym wyzwaniem intelektualnym. Wymaga ono osiągnięcia pewnego poziomu koncentracji i skupienia, niezbędnego tylko w kilku innych dziedzinach. Wynika to z tego, że tworzenie kodu wymaga od programisty żonglowania wieloma sprzecznymi zasadami.

- 1.** Twój kod musi działać. Musisz poznać problem, który masz rozwiązać, i zdefiniować sposób jego rozwiązania. Musisz mieć pewność, że napisany przez Ciebie kod będzie wiernym odwzorowaniem tego rozwiązania. Musisz zadbać o każdy szczegół tego rozwiązania, pozostając w zgodzie z językiem programowania, platformą, używaną architekturą oraz kruczkami systemu.
- 2.** Kod musi rozwiązywać problem zdefiniowany przez klienta. Często jest tak, że wymagania określone przez klienta wcale nie rozwiązują jego problemów.

Twoim zadaniem jest wychwycenie tych rozbieżności i takie prowadzenie negocjacji z klientem, żeby zostały zaspokojone jego rzeczywiste potrzeby.

3. Twój kod musi dopasować się do istniejącego systemu. Nie powinien zwiększać jego sztywności, delikatności ani nieprzejrzystości. Wszystkie zależności muszą pozostać pod kontrolą. W skrócie: Twój kod musi być zgodny z podstawowymi zasadami inżynierii<sup>2</sup>.
4. Twój kod musi być czytelny dla pozostałych programistów. I nie chodzi tu tylko o pisanie odpowiednich komentarzy, a raczej o takie ukształtowanie kodu, żeby sam ujawniał Twoje zamiary. To jest najtrudniejsze. Wydaje się, że dla programisty właśnie to może być najtrudniejsze do opanowania.

Žonglowanie tymi wszystkimi zasadami może być naprawdę trudne. Już samo fizjologiczne utrzymywanie niezbędnego skupienia przez dłuższy czas jest niełatwne. Dodaj do tego cały zbiór problemów i elementów rozpraszających wynikających z pracy w zespole, w większej organizacji, nie wspominając nawet o typowych rozterkach codziennego życia. Chodzi o to, że na każdym rogu czai się coś, co może zmniejszyć Twoje skupienie.

Jeżeli nie możesz się odpowiednio skoncentrować i skupić, to z pewnością napisany przez Ciebie kod będzie nieprawidłowy. Będzie zawierał błędy. Będzie miał nieodpowiednią strukturę. Będzie nieprzejrzysty i zagmatwany. Nie będzie rozwiązywał problemów klienta. Oznacza to, że będzie musiał zostać przebudowany i napisany na nowo. Praca bez skupienia tworzy tylko śmieci.

Jeżeli jesteś zmęczony lub brakuje Ci skupienia, to *nie twórz kodu*. Ostatecznie staniesz przed koniecznością ponownego wykonania tej samej pracy. Lepiej będzie od razu wyeliminować elementy rozpraszające i uspokoić swój umysł.

## Kod z godziny 3 nad ranem

Najgorszy kod w życiu napisałem o godzinie 3 nad ranem. Było to w roku 1988, kiedy to pracowałem dla młodej firmy telekomunikacyjnej o nazwie Clear Communications. Wszyscy w firmie pracowaliśmy przez długie godziny, żeby wytworzyć „kapitał” (ang. *sweat equity*). Oczywiście wszyscy marzyliśmy o bogactwie.

Pewnego bardzo późnego wieczoru (a może raczej bardzo wczesnego ranka) w ramach rozwiązywania problemu czasowego nakazałem swojemu kodowi wysyłać do samego siebie komunikat poprzez system rozprowadzania zdarzeń (nazywaliśmy to „wysyłaniem poczty”). To było bardzo *niewłaściwe* rozwiązanie, ale o 3 nad ranem wyglądało bardzo obiecująco. Co więcej, po 18 godzinach tworzenia kodu (nie wspominając o 60 – 70 godzinach pracy w tygodniu) na nic więcej nie było stać.

---

<sup>2</sup> [PPP2002].

Pamiętam, że byłem dumny z tego, że tak długo pracowałem. Pamiętam, że czułem swoje *poświęcenie*. Pamiętam też, że sądziłem, iż zawodowcy zawsze pracują do 3 nad ranem. Oj, jak bardzo się myliłem!

Tamten kod kopnął nas jeszcze wielokrotnie. Zdefiniowałem wadliwą strukturę, z której korzystali wszyscy, ale jednocześnie zmuszeni byli tworzyć ciągle obejścia. Wytworzyła ona najdziwniejsze problemy czasowe i niezwykłe pętle sprzężenia zwrotnego. Wpadaliśmy w nieskończone pętle wiadomości, gdy jeden komunikat powodował wysłanie innego i jeszcze kolejnego, *ad infinitum*. Nigdy nie mieliśmy czasu na napisanie na nowo fragmentów powodujących te problemy (a przynajmniej tak się nam wydawało), ale zawsze znajdowaliśmy czas na utworzenie kolejnej łatki i poprawki obchodzących wykrywane problemy. Całość ciągle rosła, opakowując ten kod napisany o 3 nad ranem w coraz większy bagaż efektów ubocznych. Później stało się to podstawą żartów w zespole. Za każdym razem, gdy byłem zmęczony lub sfrustrowany, mówili do siebie: „Patrzcie! Bob zaraz wyśle do siebie maila!”.

Morał tej historii jest taki: nie pisz kodu, jeżeli jesteś zmęczony. Poświęcenie i profesjonalizm są bardziej związane z odpowiednią dyscypliną, a nie z godzinami pracy. Upewnij się, że Twój styl życia, zdrowie i dawka snu są wystarczające, żeby pracy poświęcić osiem *dobrych* godzin.

## **Kod ze zmartwieniami**

Czy kiedykolwiek zdarzyło Ci się ostro pokłócić ze wspólnałożonkiem lub przyjacielem, a zaraz potem przystąpić do pisania kodu? Pamiętasz zapewne, że gdzieś z tyłu Twojej głowy trwał mały proces, który starał się ponownie analizować przebieg kłótni. Stres generowany przez ten mały proces można czasami poczuć w klatce piersiowej lub w żołądku. Sprawia to, że czujesz niepokój jak po wypiciu zbyt wielu kaw lub coli. To bardzo rozprasza.

Jeżeli martwię się kłótnią ze swoją żoną albo problemami z klientem albo chorym dzieckiem, to nie jestem w stanie się odpowiednio skupić. Moja koncentracja cały czas odpływa. Okazuje się, że mimo oczu skierowanych na ekran, a palców leżących na klawiaturze nie jestem w stanie nic zrobić. Katatonia. Paraliż. Jestem tysiąc kilometrów stąd, analizując palący mnie problem, a nie zastanawiając się nad tworzonym kodem.

Czasami zmuszam się do *myślzenia* na temat kodu. W ten sposób jestem w stanie dopisać wiersz lub dwa. Mogę zmusić się do napisania jednego testu lub dwóch, ale nie jestem w stanie dłużej koncentrować się na pracy. Ostatecznie zawsze popadam w to niezwykłe odrętwienie, w którym nie widzę nic mimo otwartych oczu i przetwarzam moje najróżniejsze troski.

Nauczyłem się już, że nie jest to dobry czas na tworzenie kodu. Każdy kod, który w tym czasie napiszę, będę mógł wyrzucić do kosza. Dlatego właśnie zamiast pisać, staram się rozwiązywać problemy powodujące zmartwienia.

Oczywiście istnieje wiele trosk, których nie da się usunąć z godziny na godzinę. Co więcej, nasz pracodawca raczej nie będzie chętnie tolerował naszej bezczynności, w czasie gdy

---

będziemy starać się rozwiązywać prywatne problemy. Sztuczka polega zatem na tym, żeby nauczyć się zamkniąć ten działający w tle proces, a przynajmniej zmniejszać jego priorytet, tak aby nie stawał się on nieustającym elementem rozpraszającym.

Sam robię to, odpowiednio dzieląc swój czas. Zamiast zmuszać się do pisania kodu z ciągle atakującymi mnie myślami, poświęcam im pewien wybrany wycinek czasu — może nawet godzinę — starając się pracować nad powodem mojego niepokoju. Jeżeli moje dziecko jest chore, to dzwonię do domu, aby dowiedzieć się, czy wszystko jest w porządku. Jeżeli pokłóciłem się z żoną, to dzwonię do niej i staram się obgadać powód tej kłótni. Jeżeli mam problemy finansowe, to myślę nad tym, jak sobie z nimi poradzić. Wiem, że najprawdopodobniej nie uda mi się rozwiązać problemu w czasie tej godziny, ale jest bardzo możliwe, że zmniejszę w ten sposób swoje napięcie i uciszę działający w tle proces.

Najlepiej byłoby, gdyby czas poświęcony na walkę z osobistymi problemami był Twoim czasem prywatnym. Spędzenie w ten sposób godziny w pracy byłoby niewłaściwe. Zawodowi programiści wykorzystują swój prywatny czas tak, żeby czas spędzony w biurze był jak najbardziej produktywny. Oznacza to, że jeszcze w domu należy poświęcić trochę czasu na zmniejszenie wewnętrznych niepokojów, tak aby nie przynosić ich ze sobą do biura.

Jednak jeżeli jesteś już w pracy, a wewnętrzne niepokoje nie pozwalają pracować z pełną skutecznością, to może lepiej poświęcić godzinę na ich uspokojenie, niż zmuszać się do pisania kodu, który później i tak będzie trzeba napisać od nowa. Albo co gorsza dalej z nim żyć.

## Strefa

Wiele już napisano o tym stanie hiperproduktywności nazywanym „przepływem” (ang. *flow*). Niektórzy programiści nazywają ten stan „strefą” (ang. *zone*). Niezależnie od nazwy każdy z nas z pewnością się już z tym stanem zetknął. To stan świadomości z dużym skupieniem i widzeniem tunelowym, w który mogą wejść programiści tworzący kod. Wówczas czują się naprawdę produktywni. To właśnie wtedy czują się całkowicie *nieomylni*. W efekcie starają się jak najdłużej pozostawać w tym stanie, a miarą ich wartości staje się spędzony w nim czas.

Oto mała rada od kogoś, kto dokładnie przerobił już ten temat: *unikaj strefy*. W tym stanie świadomości tak naprawdę nie jest się hiperproduktywnym, a już na pewno nie jest się nieomylnym. Tak naprawdę jest to tylko pewien stan medytacyjny, w którym niektóre czynniki racjonalne zostają umniejszone w celu uzyskania większej prędkości pracy.

Muszę tu wyrazić się jasno. Będąc w strefie, *na pewno* napiszesz więcej kodu. Jeżeli stosujesz praktyki TDD, to na pewno szybciej przejdiesz pętlę czerwone – zielone – refaktoryzacja. I dodatkowo *poczujesz* lekką euforię. Problem polega na tym, że będąc w strefie, stracisz szerszy pogląd na sprawy, przez co możesz podejmować decyzje, które później będzie trzeba zmieniać. Kod napisany w strefie na pewno powstaje szybciej, ale później trzeba go częściej poprawiać.

Aktualnie, gdy czuję, że sam zaczynam wpadać w strefę, na kilka minut odchodzę od komputera. Oczyszczam umysł, odpisując na kilka e-maili i czytając kilka wiadomości na Tweeterze. Jeżeli jest tuż przed południem, to idę na obiad. Jeżeli pracuję w zespole, to idę poszukać sobie partnera.

Jedną z zalet programowania w parach jest to, że dla pary praktycznie niemożliwe jest wejście w strefę. Jest to stan niekomunikatywny, podczas gdy programowanie w parach wymaga ciągłej i intensywnej komunikacji. Co ciekawe, często słyszę, że jedną z wad programowania w parach jest to, że nie pozwala ono na wejście w strefę. O rany! Przecież nikt *nie* powinien dążyć do tego, żeby znaleźć się w strefie.

To jednak *nie do końca* prawda. Czasami strefa jest miejscem, w którym chciałoby się znaleźć. Ale zdarza się to tylko podczas ćwiczeń. O tym opowiem jednak w innym rozdziale.

## Muzyka

W latach 70. w firmie Teradyne miałem prywatne biuro. Byłem wtedy administratorem systemu na naszym PDP 11/60 i jako jeden z niewielu programistów dostałem swój własny terminal. Terminalem tym był VT100 podłączony do PDP-11 za pomocą 25 metrów kabla RS-232 działającego z prędkością 9600 bodów, który przypiąłem do sufitu swojego biura i w ten sposób poprowadziłem do pokoju komputerowego.

W biurze miałem zestaw stereo. Składał się ze starego gramofonu, wzmacniacza i sporych głośników. Miałem też dużą kolekcję płyt winylowych, w tym Led Zeppelin, Pink Floyd i... chyba już wiesz, o co chodzi.

Często korzystałem z tego zestawu w czasie pisania kodu. Sądziłem, że muzyka ułatwiała mi koncentrację. Niestety, byłem w błędzie.

Pewnego dnia musiałem wrócić do modułu, który pisałem, słuchając sekwencji otwierającej *The Wall*. W komentarzach do kodu zobaczyłem kawałki tekstu tego utworu, a także wzmianki o bombowcach nurkujących i płaczących dzieciach.

To wtedy zrozumiałem, że czytelnik mojego kodu dowie się z niego więcej o moich gustach muzycznych niż o problemie, który ten kod ma rozwiązywać.

Okazało się, że słuchając muzyki, wcale nie piszę lepszego kodu. Muzyka wcale nie pomaga mi się skoncentrować. Co więcej, słuchanie muzyki zdaje się pochłaniać całkiem sporą część zasobów mojego umysłu, których potrzebuję do tworzenia czystego i dobrze zaprojektowanego kodu.

Być może w Twoim przypadku działa to inaczej. Być może Tobie muzyka *ułatwia* pisanie kodu. Znam w końcu wiele osób, które piszą kod ze słuchawkami na głowie. Jestem w stanie zaakceptować fakt, że muzyka ułatwia im pracę, ale mam dziwne podejrzenia, że tak naprawdę pomaga im ona wejść do strefy.

## Przerwy

Wyobraź sobie, że właśnie tworzysz kod na swojej stacji roboczej. Jak odpowiadasz, gdy ktoś zadaje Ci pytanie? Odpowiadasz niemiło? Patrzysz gniewnie? Czy język Twojego ciała informuje rozmówcę, że ma się szybko oddalić, bo masz inne zajęcie? W skrócie: czy zachowujesz się niegrzecznie? A może przerwywasz swoją pracę i chętnie pomagasz komuś, kto właśnie ma jakiś problem? Czy traktujesz intruza tak jak w Twoim wyobrażeniu on powinien potraktować Ciebie, gdy przyjdzieš z jakimś problemem?

Takie niegrzeczne zachowania często wiążą się ze strefą. Mogą wynikać z rozgoryczenia powodowanego wyciągnięciem ze strefy albo przerwaniem prób wejścia do niej. W obu przypadkach niegrzeczne zachowania są powiązane ze strefą.

Czasami jednak powodem wcale nie jest strefa, ale prosty fakt, że próbujesz zrozumieć coś złożonego, co wymaga sporej koncentracji. W takim przypadku masz do wyboru kilka rozwiązań.

Dobrą metodą radzenia sobie z takimi przerwami może być praca w parach. Twój partner może nadal pamiętać kontekst Waszych prac, podczas gdy Ty odbierasz telefon albo odpowiadasz na pytania innego kolegi. Po powrocie partner szybko pomoże Ci odtworzyć stan umysłu sprzed tej przerwy.

Doskonałą pomocą może być też technika TDD. Jeżeli masz niedziałający test, to z pewnością przechowuje on cały kontekst zadania, nad którym pracujesz. Po przerwie możesz łatwo wrócić do pracy i sprawić, żeby test zaczął działać.

Oczywiście zawsze będą zdarzały się przerwy, które będą Cię rozpraszały i powodowały straty czasu. Gdy coś takiego się zdarzy, pamiętaj, że następnym razem to Ty możesz potrzebować pomocy i będziesz komuś przerywać pracę. Oznacza to, że do profesjonalnego zachowania należy uprzejmość i chęć udzielania pomocy.

## Blokada twórcza

Czasami kod po prostu nie chce powstać. Zdarzyło mi się to kilka razy i wiele razy widziałem taki stan u innych. Siedzi się wtedy przed komputerem i zupełnie nic się nie dzieje.

Często znajdujesz wtedy jakieś inne zajęcie. Czytasz e-maile albo wiadomości z Tweetera. Przeglądasz różne książki, dokumenty lub kalendarze. Zwołujesz zebranie albo zajmujesz się rozmową ze współpracownikami. Robisz wtedy *cokolwiek*, byle tylko nie siedzieć przed komputerem i nie patrzeć, jak kod uparcie nie chce się urodzić.

Co powoduje taką blokadę? O wielu czynnikach wspominałem już wcześniej. Dla mnie jedną z najważniejszych rzeczy jest sen. Jeżeli się dobrze nie wyśnię, to najzwyczajniej

w świecie nie jestem w stanie pisać kodu. Innymi podobnie działającymi czynnikami mogą być strach, zmartwienia lub depresja.

Najdziwniejsze jest to, że istnieje dla tego problemu bardzo proste rozwiązanie, które sprawdza się niemal za każdym razem. Jest naprawdę proste, a może dać Ci rozpoczętystarczający do napisania całkiem sporych ilości kodu.

Rozwiązanie jest takie: znajdź sobie partnera do programowania w parze.

To niesamowite, jak dobrze sprawdza się ta metoda. Gdy tylko usiądziesz przy kimś innym, wszystkie problemy powodujące blokadę same znikają. Podczas pracy z inną osobą następuje zmiana *fizjologiczna*. Nie mam pojęcia, na czym ona polega, ale za każdym razem czuję ją doskonale. W moim mózgu lub w moim ciele następuje zmiana chemiczna, która pozwala mi się przebić przez blokadę i wrócić do normalnej pracy.

Nie jest to niestety rozwiązanie doskonałe. Czasami na zmianę trzeba czekać godzinę lub dwie, po których następuje tak poważne zmęczenie, że muszę opuścić swojego partnera i znaleźć jakiś kącik, aby odpocząć. Czasami nawet siedząc z kimś przy komputerze, nie jestem w stanie zrobić nic poza przytakiwaniem. Najczęściej jednak moją typową reakcją w takiej sytuacji jest odzyskanie swojego typowego rytmu.

## Kreatywne wejście

Istnieje jeszcze kilka innych rzeczy, które robię, żeby zapobiegać takim blokadom. Już dawno temu przekonałem się, że kreatywne wyjście zależy od kreatywnego wejścia.

Sporo czytam, i to na wiele różnych tematów. Czytam teksty poświęcone oprogramowaniu, polityce, biologii, astronomii, fizyce, chemii, matematyce i wielu innym zagadnieniom. Co więcej, uważam, że literatura science fiction najlepiej pobudza we mnie kreatywne wyjście.

W Twoim przypadku elementem pobudzającym może być coś całkiem innego. Być może dobra powieść detektywistyczna, poezja albo nawet romans. Jak sądzę, chodzi tu o to, że kreatywność tworzy kreatywność. Może też chodzić o swego rodzaju ucieczkę, eskapizm. Godziny, które spędzam z dala od moich normalnych problemów, stymulując się i wchłaniając różne kreatywne idee, wywołują niemal nieodpartą potrzebę samodzielnego tworzenia.

Nie działają na mnie jednak wszystkie rodzaje kreatywnego wejścia. Na przykład zupełnie nie pomaga mi oglądanie telewizji. Wypad do kina działa lepiej, ale tylko troszkę lepiej. Słuchanie muzyki nie pomaga mi w tworzeniu kodu, ale bardzo ułatwia tworzenie prezentacji, przemówień i filmów. Jednak ze wszystkich rodzajów kreatywnego wejścia nic nie działa na mnie lepiej niż stara dobra space opera.

## Debugowanie

Jedna z najgorszych sesji debugowania w mojej karierze przytrafiła mi się w 1972 roku. Terminale podłączone do systemu księgowego firmy Teamsters zawieszały się raz lub dwa razy na dzień. Nie dało się jednak w żaden sposób wymusić tego stanu. Błąd nie był związany z konkretnym rodzajem terminala ani z określonymi aplikacjami. Nie miało też znaczenia, co użytkownik robił tuż przed zawieszeniem. W jednej chwili terminal działał doskonale, a w następnej nie reagował już na nic.

Zdiagnozowanie tego problemu zajęło nam całe tygodnie, w czasie których firma Teamsters stawała się coraz bardziej nerwowa. Za każdym razem osoba, której terminal się zawiesił, musiała skoordynować się z innymi użytkownikami, tak żeby zakończyli oni pracę, a następnie zadzwonić do nas z prośbą o restart. To był prawdziwy koszmar.

Pierwsze dwa tygodnie spędziliśmy, zbierając dane z wywiadów prowadzonych z osobami, którym przytrafiła się blokada. Pytaliśmy, co robiły w czasie wystąpienia blokady i wcześniej. Pytaliśmy, czy zauważały cokolwiek na swoich zawieszonych terminalach. Wszystkie te pytania zadawaliśmy przez telefon, ponieważ wszystkie terminaly były umiejscowione w centrum Chicago, a my pracowaliśmy 50 kilometrów na północ od miasta, w środku pola kukurydzy.

Nie mieliśmy żadnych protokołów, żadnych liczników, żadnych debuggerów. Jedynym dostępem, jaki mieliśmy do systemu, były światełka i przełączniki na przednim panelu. Mogliśmy zatrzymać komputer i przeglądać jego pamięć po jednym słowie. Nie mogliśmy tego jednak robić dłużej niż pięć minut, ponieważ firma Teamsters musiała korzystać ze swojego systemu.

Kilka dni poświęciliśmy na napisanie prostego inspektora działającego w czasie rzeczywistym, którym mogliśmy sterować z teletekstu ASR-33 służącego nam za konsolę. W ten sposób mogliśmy zaglądać do pamięci komputera w czasie pracy systemu. Dodaliśmy komunikaty protokołu, które w krytycznych momentach były wypisywane na teletekście. Przygotowaliśmy liczniki liczające zdarzenia i zapamiętujące historię stanów, którą mogliśmy sprawdzać za pomocą inspektora. Oczywiście to wszystko zostało napisane od zera w asemblerze i było testowane wieczorami, gdy system nie był używany.

Terminale były sterowane przerwaniami, natomiast znaki przesyłane do nich były umieszczane w cyklicznych buforach. Za każdym razem, gdy port szeregowy kończył wysyłanie znaku, uruchamiane było przerwanie, w którym do wysłania był przygotowywany kolejny znak z cyklicznego bufora.

Ostatecznie okazało się, że terminaly zawieszały się wtedy, gdy rozsynchronizowały się trzy zmienne zarządzające stanem cyklicznego bufora. Nie mieliśmy pojęcia, dlaczego tak się działało, ale to była już jakaś wskazówka. Gdzieś w 5 000 wierszy kodu superwizora znajdował się błąd powodujący nieprawidłowe działanie tych wskaźników.

Ta wiedza pozwalała nam też ręcznie odblokować terminale! Mogliśmy wpisać do tych trzech zmiennych domyślne wartości za pomocą inspektora, a terminal zaczynały znowu działać. W końcu napisaliśmy mały programik, który kontrolował wartość tych liczników i w razie stwierdzenia nieprawidłowości przywracał im wartości domyślne. Początkowo program ten był wywoływany przez naciśnięcie specjalnego przełącznika na panelu komputera za każdym razem, gdy ktoś z firmy Teamsters zgłaszał zawieszenie terminala. Później uruchamialiśmy go automatycznie co sekundę.

Po mniej więcej miesiącu z punktu widzenia firmy Teamsters sprawa zawieszających się terminali została zamknięta. Czasami jakiś terminal zatrzymywał się na mniej więcej pół sekundy, ale przy prędkości 30 znaków na sekundę nikt tego nawet nie zauważał.

Ale dlaczego te liczniki się rozsynchonizowywały? Miałem wtedy 19 lat i byłem zdeterminowany, żeby znaleźć przyczynę.

Kod superwizora został napisany przez Richarda, który teraz był już na studiach. Nikt z pozostałych członków zespołu nie znał tego kodu zbyt dobrze, ponieważ Richard był w tym względzie dość zaborczy. Ten kod był *jego*, a my mieliśmy się od niego trzymać z daleka. Teraz jednak Richarda nie było już w firmie, dlatego wyciągnąłem gruby wydruk programu i zacząłem przeglądać go strona po stronie.

Cykliczne kolejki w tym systemie były po prostu strukturami typu FIFO, czyli najwyklejszymi kolejkami. Programy umieszczały znaki na jednym końcu kolejki, aż została ona zapełniona. Przerwania zdejmowały z kolei znaki z drugiego końca kolejki, za każdym razem gdy drukarka była gotowa na ich przyjęcie. Jeżeli kolejka była pusta, to drukarka się zatrzymywała. Drobny błąd sprawiał, że aplikacje były przekonane o tym, iż kolejka jest pełna, natomiast przerwania stwierdzały, że kolejka jest pusta.

Przerwania były wykonywane w innym „wątku” niż pozostały kod programu. Oznaczało to, że liczniki i inne zmienne aktualizowane zarówno przez przerwania, jak i pozostały kod musiały być odpowiednio zabezpieczone przed jednoczesnym zapisem. W naszym przypadku oznaczało to wyłączanie przerwań w pobliżu kodu manipulującego tymi zmiennymi. W tym momencie wiedziałem już, że muszę znaleźć w kodzie miejsce, które zmienia zawartość feralnych zmiennych, nie wyłączając uprzednio przerwań.

Dzisiaj mamy do dyspozycji całą paletę różnych narzędzi pozwalających wyszukać te miejsca w kodzie, które interesują się określona zmienną. W ciągu kilku sekund można wyświetlić wszystkie podejrzane wiersze kodu. Znalezienie wycinka, w którym zapomniano o wyłączeniu przerwań, zajęłoby zaledwie minutę. Jednak w roku 1972 nie istniały takie narzędzia, a do dyspozycji miałem wyłącznie swoje oczy.

Dokładnie przeglądałem kod zapisany na każdej stronie, poszukując feralnych zmiennych. Niestety, były one używane właściwie wszędzie. Niemal na każdej stronie były w ten lub inny sposób modyfikowane. W wielu przypadkach przerwanie nie były wyłączone, ponieważ

zmienne były tylko odczytywane, a przez to cała operacja była niegroźna. Problem polegał na tym, że w tym szczególnym assemblerze nie dało się stwierdzić, czy referencja zmiennej była niegroźnym odczytem, bez dokładniejszego sprawdzenia kodu. Po każdym odczycie zmiennej mogły nastąpić jej aktualizacja i zapisanie. Jeżeli zdarzyło się to w czasie, gdy przerwania były włączone, to zawartość zmiennej mogła zostać uszkodzona.

Dokładne sprawdzenie kodu zajęło mi kilka dni, ale w końcu znalazłem przyczynę. W samym środku kodu ukryło się jedno miejsce, w którym aktualizowana była jedna z trzech zmiennych bez uprzedniego wyłączenia przerwań.

Dokonałem wtedy pewnych obliczeń. Czas trwania podatności to mniej więcej dwie mikrosekundy. W firmie działało jakieś 12 terminali pobierających dane z prędkością 30 znaków na sekundę, czyli jedno przerwanie pojawiało się mniej więcej co 3 milisekundy. Biorąc pod uwagę wielkość superwizora, częstotliwość zegara procesora, można było się spodziewać, że terminal zostanie zablokowany jeden raz lub dwa razy dziennie. Bingo!

Usunąłem ten błąd, ale nigdy nie miałem odwagi wyłączyć automatu sprawdzającego i korygującego liczniki. Do dzisiaj nie mam pewności, czy w programie nie ukrywał się jeszcze inny błąd.

## Czas debugowania

Z nieznanych mi powodów część programistów nie uznaje czasu poświęconego na debugowanie za czas tworzenia kodu. Twierdzą, że czas debugowania jest po prostu wymogiem natury, czyli czymś, co trzeba zrobić. Jednak z punktu widzenia firmy czas debugowania jest tak samo kosztowny jak czas tworzenia kodu, dlatego wszystko, co zrobimy, żeby uniknąć debugowania albo przynajmniej skrócić jego czas, będzie pozytywnym działaniem.

Dzisiaj spędzam na debugowaniu znacznie mniej czasu niż jeszcze 10 lat temu. Oczywiście nie mierzyłem różnicy, ale sądzę, że jest to mniej więcej jeden rząd wielkości. Tę radykalną redukcję czasu debugowania uzyskałem przez przyjęcie praktyki TDD (*Test Driven Development*), o której będę mówił w innym rozdziale.

Niezależnie od tego, czy stosujemy praktykę TDD, czy inne rozwiązania o podobnej skuteczności<sup>3</sup>, na każdym zawodowym programiste ciąży zadanie dążenia do jak najskuteczniejszego redukowania czasu spędzanego na debugowaniu. Zeroowy czas jest tutaj celem asymptotycznym, ale mimo to zawsze powinien być naszym celem.

Chirurdzy nie lubią ponownie otwierać swoich pacjentów, żeby poprawić coś, co zrobili źle. Prawnicy nie lubią powtarzać pozów tylko dlatego, że przy poprzednim palnęli głupotę.

---

<sup>3</sup> Nie znam innej techniki o skuteczności zbliżonej do TDD, ale może wiesz coś, czego ja nie wiem.

Chirurg lub prawnik, który zbyt często popełnia takie błędy, nigdy nie będzie uznany za profesjonalistę. Podobnie programista robiący zbyt wiele błędów działa nieprofesjonalnie.

## **Wyznaczanie sobie rytmu**

Tworzenie oprogramowania to bardziej maraton niż sprint. Tego wyścigu nie da się wygrać, biegając od samego początku tak szybko, jak się da. Kluczem do zwycięstwa jest zachowywanie niezbędnych zasobów i wyznaczanie sobie odpowiedniego rytmu. Maratończyk dba o siebie zarówno przed wyścigiem, jak i *podczas* niego. Zawodowi programiści w podobny sposób zachowują swoją energię i kreatywność.

## **Wiedzieć, kiedy odejść**

Nie możesz wrócić do domu, dopóki nie rozwiążesz tego problemu? Oczywiście, że możesz, a naprawdę tak należałoby zrobić. Kreatywność i inteligencja to bardzo ulotne stany umysłu. Jeżeli uginasz się od zmęczenia, to na pewno się ich pozbywasz. Jeżeli wtedy zmuszasz swój przemęczony mózg do pracy w późnych godzinach nocnych, próbując rozwiązać jakiś problem, to tylko zwiększasz swoje zmęczenie, a tym samym zmniejszasz szanse na to, że prysznic lub samochód pomogą Ci znaleźć rozwiązanie.

Jeżeli utkniesz w pracy i zmęczenie nie pozwoli Ci dalej pracować, to spróbuj się na chwilę odłączyć. Daj swojej podświadomości szansę na podjęcie próby rozwiązania problemu. Jeżeli odpowiednio zadbasz o swoje zasoby, to uzyskasz więcej w krótszym czasie przy znacznie mniejszym wysiłku. Odpowiednio wyznaczaj rytm sobie i swojemu zespołowi. Naucz się stosować wzorce kreatywności oraz błyskotliwości i staraj się je wykorzystać do swoich celów, a nie walczyć przeciwko nim.

## **Jazda do domu**

Miejscem, w którym udało mi się rozwiązać zadziwiająco wiele problemów, jest mój samochód wiozący mnie z pracy do domu. Kierowanie autem wymaga zaangażowania wielu niekreatywnych zasobów umysłowych. Wykonywaniu tego zadania musisz poświęcić swoje oczy, ręce oraz część umysłu. Oznacza to, że musisz odłączyć się od problemów z pracy. Takie właśnie *odłączenie* pozwala Twojemu umysłowi na poszukiwanie rozwiązań w inny, znacznie bardziej kreatywny sposób.

## **Prysznic**

Równie wielką liczbę problemów udało mi się rozwiązać pod prysznicem. Być może ten strumień wody wcześnie rano pobudza mnie na tyle, że mogę ponownie przeanalizować wszystkie rozwiązania, na jakie wpadł mój umysł w czasie, gdy spałem.

Pracując nad jakimś problemem, czasami wchodzisz w niego tak głęboko, że nie jesteś w stanie dostrzec wszystkich możliwych opcji. Pomijasz eleganckie rozwiązania, ponieważ kreatywna część Twojego umysłu jest tłumiona przez intensywność skupienia na zadaniu. Czasami najlepszym sposobem na rozwiązywanie problemu jest powrót do domu, zjedzenie kolacji, obejrzenie czegoś w telewizji, pójście do łóżka i skorzystanie z prysznica po przebudzeniu się następnego dnia.

## Spóźnienia

Spóźnienia będą Ci się przytrafiać. To zdarza się najlepszym spośród nas. Zdarza się tym najbardziej oddanym pracy. Czasami po prostu nasze szacunki nie są dość dokładne i wtedy się spóźniamy.

Właściwą metodą radzenia sobie z opóźnieniami jest wczesne ich wykrywanie i otwarte informowanie o nich. Nie ma nic gorszego niż utrzymywanie do samego końca, że zdążyłeś na czas, i sprawienie zawodu w ostatnim momencie. Tak się *nie* robi. Zamiast tego *regularnie* mierz swoje postępy i porównuj je z celem, a następnie podawaj trzy<sup>4</sup> wynikające z tego porównania daty: najlepszy przypadek, normalny przypadek i najgorszy przypadek. *Do tych szacunków nie dodawaj swoich nadziei!* Trzy wyliczone daty zaprezentuj całemu zespołowi i wszystkim zainteresowanym, a następnie codziennie je aktualizuj.

## Nadzieja

Co zrobić, jeżeli z tych szacunków wynika, że *możesz* przestrzelić termin? Dla przykładu założmy, że za 10 dni są targi i do tego czasu musimy mieć gotowy produkt. Założmy też, że Twoje trzy szacowane daty ukończenia funkcji, nad którą pracujesz, to 8/12/20.

*Nie opieraj się na nadziei, że zdołasz zrobić wszystko w 10 dni!* Nadzieja jest zabójcą projektów. Nadzieja niszczy plany i rujnuje reputację. Nadzieja wpędzi Cię w poważne tarapaty. Jeżeli targi zaczynają się za 10 dni, a Twoja szacunkowa, normalna data ukończenia to 12 dni, to znaczy, że *nie zdążyłeś* na czas. Upewnij się, że cały zespół i udziałowcy znają szczegóły sytuacji, i nie odpuszczaj, dopóki nie powstanie plan awaryjny. Nie dawaj innym fałszywej nadziei.

## Pośpiech

Co zrobić, jeżeli przełożony usadza Cię i prosi o próbę dotrzymania terminu? Co zrobić, jeżeli kierownik nastaje, żeby „zrobić, co trzeba”? *Nie koryguj swoich szacunków!* Twoje pierwotne szacunki będą o wiele dokładniejsze niż wszelkie zmiany, jakie wprowadzisz do nich podczas takiej konfrontacji z szefem. Powiedz szefowi, że wszelkie możliwości zostały

---

<sup>4</sup> Więcej na ten temat opowiem w rozdziale „Szacowanie”.

już wzięte pod uwagę (bo tak się stało) i jedyną metodą poprawienia planu jest redukcja zakresu projektu. *Nie ulegaj pokusom przyspieszenia prac.*

Biada biednemu programiście, który ugnie się pod presją i zgodzi się *spróbować* dotrzymać terminu. Taki programista zacznie wykorzystywać skróty i pracować w nadgodzinach w próżnej nadziei na cud. To jest najprostsza recepta na katastrofę, ponieważ daje to zespołowi i udziałowcom fałszywą nadzieję na sukces. Przez to nikt może nie zauważać problemu, co opóźnia podjęcie niezbędnych, choć trudnych decyzji.

Pośpiech nie jest żadnym rozwiązaniem. Nie jesteś w stanie szybciej pisać kodu. Nie zmusisz się do szybszego rozwiązywania problemów. Jeżeli zaczniesz podejmować takie próby, to tylko spowolnisz swoje działania, tworząc przy okazji bałagan, który zacznie spowalniać pozostałych członków zespołu.

Musisz zatem udzielić szefowi i zespołowi uczciwej odpowiedzi, która pozbawi ich fałszywej nadziei.

## **Nadgodziny**

Zdarza się, że szef mówi Ci: „A jeżeli popracujesz dwie dodatkowe godziny dziennie? A jeżeli przyjdziesz do pracy w sobotę? Przecież musi być jakiś sposób, żeby wygospodarować dodatkowe godziny i przygotować tę funkcję na czas”.

Nadgodziny mogą być dobrym rozwiązaniem, a czasami są nawet nieodzowne. Czasami można dotrzymać niemożliwego terminu, pracując po 10 godzin dziennie, a nawet w jedną sobotę lub dwie. To jednak bardzo ryzykowne założenie. Nie jesteś w stanie wykonać 20% więcej pracy, poświęcając na to 20% więcej czasu. Co więcej, nadgodziny *na pewno* stracą jakikolwiek sens, jeżeli taki tryb pracy potrwa dłużej niż dwa lub trzy tygodnie.

Oznacza to, że nie należy zgadzać się na nadgodziny, chyba że (1) możesz sobie na to pozwolić, (2) jest to rozwiązanie krótkoterminowe, maksymalnie na dwa tygodnie i (3) Twój szef ma już plan awaryjny na wypadek, gdyby praca w nadgodzinach nie wystarczyła.

Ten ostatni warunek jest tutaj najważniejszy. Jeżeli Twój szef nie jest w stanie stwierdzić, co zrobi, jeżeli nadgodziny nie przyniosą oczekiwanych efektów, to nie zgadzaj się na dodatkowe godziny pracy.

## **Fałszywa dostawa**

Ze wszystkich nieprofesjonalnych zachowań, na jakie może sobie pozwolić programista, chyba najgorsze jest twierdzenie, że prace są zakończone, choć ma się pełną świadomość tego, że tak nie jest. Czasami jest to najzwyczalsze kłamstwo, co samo w sobie jest złe. Jednak znacznie bardziej podstępnym przypadkiem jest próba stworzenia nowej definicji

---

„gotowego”. Próbujemy się wtedy przekonać, że jesteśmy wystarczająco gotowi, i od razu przechodzimy do następnego zadania. Uznajemy, że praca, która została jeszcze do wykonania, może zostać zrobiona później, kiedy będzie na to czas.

Takie zachowanie jest bardzo zaraźliwe. Jeżeli robi tak jeden programista, to pozostały na pewno to zauważą i zaczną postępować podobnie. Jeden z nich jeszcze bardziej rozciągnie definicję tego, co „gotowe”, a reszta się do niej szybko przystosuje. Miałem nieprzyjemność oglądać już ekstremalne przypadki takiego działania. Jeden z moich klientów definiował „gotowe” jako „zapisane w systemie”. Kod nawet nie musiał się komplikować. Jeżeli nic nie musi działać, to bardzo łatwo powiedzieć „gotowe”!

Gdy zespół wpada w tę pułapkę, menedżerowie słyszą, że wszystko w projekcie jest w porządku. Raporty o stanie wskazują, że wszyscy pracują zgodnie z planem. Przypomina to piknik ślepców na torach kolejowych. Nikt nie zauważa pociągu towarowego wiożącego niedokończoną pracę, aż w końcu jest za późno.

## Definicja „gotowego”

Problemu fałszywych dostaw można uniknąć, tworząc całkowicie niezależną definicję „gotowego”. Najlepszą metodą jej uzyskania jest współpraca analityków i testerów w celu przygotowania zautomatyzowanych testów akceptacyjnych<sup>5</sup>, które muszą zostać zaliczone, aby dana funkcja mogła zostać uznana za gotową. Takie testy powinny zostać zapisane w wyspecjalizowanym języku, takim jak FitNesse, Selenium, RobotFX, Cucumber lub podobnym. Poszczególne testy muszą być zrozumiałe dla udziałowców projektu oraz dla biznesmenów, a przede wszystkim muszą być uruchamiane odpowiednio często.

## Pomoc

Programowanie to ciężka praca. Im ktoś jest młodszy, tym trudniej mu w to uwierzyć. W końcu to tylko zbiór instrukcji if i while. Jednak zbierając doświadczenie, zaczynasz dostrzegać, że najważniejszy jest sposób, w jaki łączone są ze sobą te wszystkie instrukcje. Nie możesz ich umieścić po prostu w sekwencji i mieć nadzieję, że taka konstrukcja zadziała. Konieczne jest ostrożne dzielenie całego systemu na mniejsze, zrozumiałe części, które mają ze sobą jak najmniej wspólnego. I to właśnie jest najtrudniejsze.

Programowanie jest tak trudne, że jedna osoba nie jest w stanie zrobić tego dobrze. Niezależnie od tego, jak wielki masz talent, to na pewno skorzystasz na przemyśleniach i pomysłach innych programistów.

---

<sup>5</sup> Zajrzyj do rozdziału 7., „Testy akceptacyjne”.

## **Pomaganie innym**

Z tego właśnie powodu każdy programista powinien w miarę możliwości pomagać innym. Izolowanie się w swoim boksie lub biurze i odmawianie odpowiedzi na pytania zadawane przez innych jest naruszeniem etyki zawodowca. Twoja praca nie jest tak ważna, że nie możesz poświęcić chwili na to, by pomóc kolegom. Co więcej, każdy profesjonalista podejmuje honorowe zobowiązanie do udzielania pomocy wszędzie tam, gdzie jest ona niezbędna.

Nie oznacza to, że nie potrzeba Ci czasu spędzonego w samotności. Oczywiście, że tego potrzebujesz. Ale musisz o tym otwarcie i spokojnie informować. Na przykład możesz przekazać zespołowi, że pomiędzy godziną 10 rano a południem nie należy Ci przeszkadzać, ale już w godzinach od 13 do 15 Twoje drzwi są otwarte i każdy może przyjść z pytaniami.

Musisz mieć pełną świadomość tego, w jakim stanie są członkowie Twojego zespołu. Jeżeli zauważasz, że ktoś ma problemy, to zaoferuj mu swoją pomoc. Możesz się naprawdę zdziwić, jak wielki skutek może odnieść Twój pomoc. Nie chodzi o to, że ta inna osoba jest mniej inteligentna od Ciebie. Po prostu spojrzenie z innej perspektywy może mieć ogromne znaczenie przy rozwiązywaniu problemów.

Próbuając komuś pomóc, najlepiej jest usiąść razem z nim i razem zacząć tworzyć kod. Zaplanuj na ten cel jakąś godzinę, a może i więcej. Być może potrzeba będzie mniej czasu, ale też nie chodzi o popędzanie kogokolwiek. Uznaj dane zadanie za własne i przyłoż się do niego solidnie. Całkiem możliwe, że na końcu nauczysz się więcej, niż dasz od siebie.

## **Przyjmowanie pomocy**

Jeżeli ktoś proponuje Ci swoją pomoc, okaż mu wdzięczność. Przyjmij ją i staraj się ją jak najlepiej wykorzystać. *Nie chroń swojego terytorium*. Nie rezygnuj z oferowanej pomocy, nawet jeżeli masz nóż na gardle. Poświęć na ten cel jakieś pół godziny. Jeżeli przez ten czas okaże się, że kolega nie jest w stanie Ci pomóc, to pięknie mu podziękuj i zakończ sesję. Pamiętaj, że tak jak honor zobowiązuje Cię do udzielania pomocy, tak i zobowiązuje Cię do jej przyjmowania.

Naucz się też *prosić* o pomoc. Jeżeli utkniesz, dopadnie Cię zamroczenie albo po prostu nie będziesz w stanie ogarnąć problemu, to poproś kogoś o pomoc. Jeżeli siedzisz w pokoju z całym zespołem, możesz po prostu wstać i powiedzieć: „Potrzebuję pomocy”. W innych okolicznościach wykorzystaj Tweetera, e-mail albo telefon stojący na biurku i poproś kogoś o pomoc. Powtarzam raz jeszcze, że jest to kwestia etyki zawodowej. Zdecydowanie nieprofesjonalne jest stanie w miejscu, podczas gdy pomoc jest tak łatwo dostępna.

W tym momencie może Ci się wydawać, że zaraz chóralnie zaśpiewam *Kumbaya*, a w tle różowe króliczki będą wskakiwać na grzbiety jednorożców i wspólnie poszybijemy ponad tęczą nadziei i zmian. No, nie do końca. Okazuje się, że programiści *bywają* aroganckimi,

pochloniętymi sobą introwertykami. Tego zawodu nie wybiera się dlatego, że tak bardzo lubi się ludzi. Większość z nas została programistami, ponieważ wolimy koncentrować się na sterylnych szczegółach, jednocześnie przerzucając różne pomysły, i w ogóle udowadniać, że mamy mózgi wielkości małej planety, a jednocześnie unikać skomplikowanej interakcji z innymi ludźmi.

To oczywiście stereotyp i wielka generalizacja, z wieloma różnymi wyjątkami. Ale rzeczywistość jest taka, że programiści raczej nie są najlepszymi współpracownikami<sup>6</sup>. A mimo to współpraca jest bardzo istotnym czynnikiem skutecznego programowania. Oznacza to, że choć dla wielu z nas współpraca nie jest instynktowna, musimy wyrobić sobie dyscyplinę, która będzie nas do tego zmuszała.

## Mentor

W dalszej części książki poświęcenie temu tematowi cały rozdział. Na razie powiem tylko tyle, że szkolenie mniejszości doświadczonych programistów jest zadaniem tych o większym doświadczeniu. I nie liczą się tutaj kursy programowania. Nie liczą się książki. Nic nie jest w stanie szybciej wynieść młodego programisty na wyżyny wydajności niż własna motywacja i pomoc starszych kolegów. I znów poświęcenie czasu, żeby wziąć młodych pod swoje skrzydła i odpowiednio ich szkolić, należy do etyki zawodowej doświadczonych programistów. Z tego wynika też, że zadaniem młodszych programistów jest poszukiwanie możliwości skorzystania na doświadczeniu starszych kolegów.

## Bibliografia

- [Martin09]: Robert C. Martin, *Czysty kod. Podręcznik dobrego programisty*, Gliwice, Helion, 2009.
- [PPP2002]: Robert C. Martin, *Agile Software Development. Principles, Patterns, and Practices*, Upper Saddle River, NJ: Prentice Hall, 2002.

---

<sup>6</sup> To znacznie częściej sprawdza się w przypadku mężczyzn niż kobiet. Prowadziłem kiedyś bardzo interesującą rozmowę z @desi (Desi McAdam, założycielką witryny DevChix) na temat tego, co motywuje kobiety programistki. Powiedziałem, że gdy udaje mi się w końcu uruchomić prawidłowo działający program, czuję się, jakbym powałił wielką bestię. Ona odpowiedziała, że dla niej i wielu innych kobiet, z którymi rozmawiała, akt tworzenia kodu jest aktem wychowawczej kreacji.



---

# T<sub>5</sub> D

---



Minęło już ponad 10 lat od czasu, gdy po raz pierwszy świat usłyszał o TDD (*Test Driven Development* — rozwój oprogramowania sterowany testami). Praktyka ta była częścią fali programowania ekstremalnego (XP — *Extreme Programming*), ale później została zastosowana też w takich metodach jak Scrum i właściwie wszystkich innych metodach zwinnych. Nawet zespoły niestosujące metodologii zwinnych używają dzisiaj TDD.

Kiedy w 1998 roku pierwszy raz usłyszałem o „programowaniu najpierw testów”, byłem raczej sceptycznie nastawiony. Kto by nie był? *Najpierw* pisać testy? Któz wpadł na tak szalony pomysł?

W tym czasie miałem już 30-letnie doświadczenie w tworzeniu oprogramowania, dlatego widziałem już różne pomysły powstające na naszej grzędce. Wiedziałem też, że nie należy tak po prostu odrzucać jakichkolwiek pomysłów, a szczególnie tych, o których mówi Kent Beck.

A zatem w 1999 roku pojechałem do Medford w stanie Oregon, żeby spotkać się z Kentem i od niego nauczyć się nowej metodologii. To, czego doświadczyłem, wywołało u mnie szok!

Kent siedział ze mną w swoim biurze i zaczął pisać prosty programik w Javie. Ja przystąpiłbym bezpośrednio do pracy nad tą drobnostką, ale Kent mnie powstrzymał i kazał przejść krok po kroku przez cały proces. Najpierw napisał małą część testu jednostkowego, który z ledwością można było uznać za kod. Potem napisał tyle kodu programu, żeby ten test udało się skompilować. Znowu dopisał trochę testów i uzupełniał je kodem programu.

Taki cykl pracy zupełnie nie pasował do moich doświadczeń. Byłem przyzwyczajony do pisania kodu przez prawie godzinę, zanim w ogóle spróbowałem go skompilować i uruchomić. Ale Kent uruchamiał swój kod niemalże co trzydzieści sekund. Byłem niezmiernie zdziwiony.

Co więcej, od razu rozpoznałem czas trwania tego cyklu! Takiego samego cyklu pracy używałem w czasie swoich dziecięcych<sup>1</sup> zabaw z programowaniem gier w językach interpretowanych, takich jak Basic lub Logo. W tych językach nie istniała konieczność komplikacji, dlatego wystarczyło dopisać wiersz kodu i uruchomić go. Taki cykl powtarzał się bardzo szybko, a dzięki temu pisanie programów w tych językach też było *niezmiernie szybkie*.

Jednak w przypadku *prawdziwego* programowania taki cykl pracy był absurdalny. W *prawdziwym* programowaniu trzeba poświęcić sporo czasu na napisanie kodu, a jeszcze więcej na jego komplikację. A potem całą góre czasu przeznaczyć na debugowanie. W końcu *byłem programistą C++!* A w tym języku czas komplikacji i konsolidacji mierzony jest w minutach, a czasem i w godzinach. Takie 30-sekundowe cykle były nie do pomyślenia.

A mimo to Kent pracował sobie nad programem w Javie, stosując takie właśnie 30-sekundowe cykle, i nic nie wskazywało na to, że zaraz zacznie zwalniać. Siedząc tak oszołomiony w biurze Kenta, uświadomiłem sobie, że przy zachowaniu takiej dyscypliny mógłbym tworzyć kod w prawdziwych językach z szybkością, jaką znałem z Logo! To było porażające!

## Sąd na sali

Od tego czasu dowiedziałem się, że technika TDD jest czymś więcej niż tylko prostą sztuczką pozwalającą skrócić cykl pracy nad oprogramowaniem. Wiąże się z nią cały zbiór zalet, o których napiszę w poniższych akapitach.

---

<sup>1</sup> Z mojego punktu widzenia dzieckiem jest jeszcze każdy, kto nie ukończył 35. roku życia. Gdy miałem 20 lat, sporo czasu spędzałem na pisaniu małych, prostych gier w językach interpretowanych. Pisałem w nich gry wojenne, przygodowe, wyścigi konne, węzopodobne, hazardowe i inne.

Najpierw muszę jednak powiedzieć, że:

- Sąd jest na sali!
- Kontrowersje się skończyły.
- GOTO jest złe.
- A TDD naprawdę działa.

Owszem, w tym czasie napisano już wiele kontrowersyjnych blogów oraz artykułów poświęconych TDD i nadal powstają nowe. Początkowo przeważały całkiem poważne próby krytykowania i zrozumienia tej techniki. Dzisiaj można już przeczytać tylko puste tyradły. Najważniejsze jest, że TDD działa, i powinniśmy przyjąć to do wiadomości.

Wiem, to brzmi strasznie jednostronnie, ale sądzę, że tak jak chirurdzy nie powinni być zmuszani do udowadniania zalet mycia rąk, tak programiści nie powinni być zmuszani do obrony TDD.

Czy możesz myśleć o sobie jako o profesjonalście, a jednocześnie nie mieć *pewności*, że Twój kod naprawdę działa? A jak możesz mieć pewność, że Twój kod działa, jeżeli nie przetestujesz go za każdym razem, gdy wprowadzisz do niego zmianę? Jak możesz testować swój kod przy każdej zmianie, jeżeli nie masz gotowych zautomatyzowanych testów o bardzo wysokim pokryciu kodu? Jak możesz uzyskać takie zautomatyzowane testy jednostkowe z wysokim pokryciem, jeżeli nie stosujesz TDD?

To ostatnie zdanie wymaga pewnego uszczegółowienia. Czym właściwie jest TDD?

## Trzy prawa TDD

1. Nie wolno napisać Ci nawet wiersza produktywnego kodu, jeżeli wcześniej nie napiszesz pasującego testu jednostkowego, który nie zostanie zaliczony.
2. Nie wolno Ci napisać więcej testu jednostkowego, niż jest konieczne, żeby test nie został zaliczony. Nieudana komplikacja też powoduje, że test nie jest zaliczany.
3. Nie wolno Ci napisać więcej produktywnego kodu, niż potrzeba, żeby aktualnie oblany test został zaliczony.

Te trzy proste prawa zamkijają Cię w cyklu, który trwa być może 30 sekund. Zaczynasz od napisania małej części testu jednostkowego. Ale już po kilku sekundach musisz wpisać nazwę jakiejś klasy lub jej funkcji, która jeszcze nie została napisana, przez co testu nie da się zaliczyć z powodu błędu komplikacji. Oznacza to, że musisz dopisać kod produktywny, tak żeby test można było skompilować. Nie możesz jednak napisać nic ponadto, a zatem wracasz do pisania kodu w teście jednostkowym.

I tak w koło Macieju. Dopisanie kawałczka kodu testowego. Dopisanie kawałczka kodu produktywnego. Dwa strumienie kodu rosnące jednocześnie w postaci uzupełniających się składników. Test dopasowuje się do kodu produktywnego, jak antyciało dopasowuje się do antygenu.

## Litania zalet

### Pewność

Jeżeli zaczniesz w normalnej pracy stosować TDD, to każdego dnia będziesz tworzyć dziesiątki testów, co tydzień wytwarzysz ich setki, a w rok zbiorą się tysiące. A wszystkie te testy będą zawsze aktualne i będziesz uruchamiać je przy każdej zmianie, jaką wprowadzisz do kodu.

Jestem głównym autorem i konserwatorem projektu FitNesse<sup>2</sup>, czyli napisanego w Javie narzędzia do testów akceptacyjnych. W czasie tworzenia tej książki projekt składał się z 64 000 wierszy kodu, z czego 28 000 zawierało trochę ponad 2200 testów jednostkowych. Testy te pokrywają ponad 90% kodu produktywnego<sup>3</sup>, a uruchomienie ich wszystkich zajmuje jakieś 90 sekund.

Za każdym razem, gdy wprowadzam jakąkolwiek zmianę do FitNesse, po prostu uruchamiam testy jednostkowe. Jeżeli zostaną zaliczone, mam niemal całkowitą pewność, że wprowadzona właśnie zmiana niczego nie popsuła. Co właściwie znaczy „niemal całkowitą pewność”? Na tyle dużą, żeby udostępniać poprawkę!

Proces kontroli jakości projektu FitNesse składa się z jednego polecenia: `ant release`. Polecenie to kompliluje cały projekt od zera, uruchamia wszystkie testy jednostkowe i akceptacyjne. Jeżeli wszystkie testy zostaną zaliczone, to znaczy, że mogę udostępniać wersję.

### Współczynnik wprowadzania błędów

Oczywiście, że FitNesse nie jest szczególnie ważną aplikacją. Jeżeli pojawi się w niej błąd, to nikt nie zginie ani nie straci milionów dolarów. Dzięki temu mogę wydawać kolejne wersje na podstawie jedynie zaliczonych testów. Jednakże FitNesse ma już tysiące użytkowników, a pomimo dopisania w ostatnim roku 20 000 wierszy nowego kodu na mojej liście błędów znajduje się tylko 17 pozycji (wiele z nich to tylko kosmetyka). Wiem zatem, że mój współczynnik wprowadzania błędów jest bardzo niski.

---

<sup>2</sup> <http://fitnesse.org>.

<sup>3</sup> 90% to minimum. Faktyczna wartość jest zdecydowanie wyższa. Niestety, dokładną wartość pokrycia bardzo trudno jest wyliczyć, ponieważ obliczające ją narzędzia nie widzą kodu uruchamianego w zewnętrznych procesach albo w blokach `catch`.

To nie jest efekt jednostkowy. Istnieje już kilka raportów<sup>4</sup> i badań<sup>5</sup> opisujących ogromną redukcję liczby błędów. Każda firma od IBM do Microsoftu, od Sabre do Symanteca doświadczyła dwukrotnego, pięciokrotnego, a nawet dziesięciokrotnego zmniejszenia liczby błędów. Takich wartości nie może ignorować żaden profesjonalista.

## Odwaga

Dlaczego nie poprawiasz złego kodu w momencie, w którym go dostrzeżesz? Twoją pierwszą reakcją po zobaczeniu kodu zabałaganionej funkcji jest myśl: „Ten bałagan trzeba koniecznie uprątać”. Zaraz po niej pojawia się jednak: „Nawet tego nie dotknę”. Dlaczego? Ponieważ wiesz, że jeżeli tego dotkniesz, to prawdopodobnie coś zepsujesz. A jeżeli coś zepsujesz, to na Ciebie spadnie odpowiedzialność za to.

Co by się jednak stało, gdyby dano Ci pewność, że Twoje poprawki niczego nie zepsują? Czy wpłynęłoby to na Twoje postępowanie? Co by było, gdyby po kliknięciu jednego przycisku i odczekaniu 90 sekund można było uzyskać pewność, że Twoje zmiany nic nie zepsuły, a jedynie *poprawiły jakość kodu*?

To jedna z najważniejszych zalet TDD. Jeżeli dysponujesz zestawem testów, którym możesz zaufać, to nie masz już obaw przed wprowadzaniem zmian. Jeżeli widzisz zły kod, to po prostu go poprawiasz. Kod upodabnia się do plasteliny, którą możesz dowolnie kształtować, tworząc proste i przyjemne struktury.

Gdy tylko programista przestaje się bać wprowadzania zmian, zaczyna czyścić istniejący kod. Z kolei czysty kod jest znacznie łatwiejszy do zrozumienia, łatwiejszy do poprawienia, a i jego rozbudowa jest prostsza. Dzięki temu, że kod staje się prostszy, zmniejsza się szansa powstania błędów. A całość kodu ciągle jest *poprawiana*, zupełnie inaczej niż w przypadku typowego dla naszego zawodu powolnego niszczenia.

Czy zawodowy programista może pozwolić sobie na dalsze niszczenie swojego kodu?

## Dokumentacja

Zdarzyło Ci się już korzystać z zewnętrznych bibliotek? Często do takich bibliotek dołączana jest ładnie sformatowana instrukcja napisana przez wyznaczonych do tego ludzi. Zazwyczaj taka instrukcja składa się z błyszczących fotografii uzupełnionych kółkami i strzałkami oraz podpisem wyjaśniającym, jak należy skonfigurować i zainstalować daną bibliotekę, manipulować nią czy też po prostu jej używać. A na końcu instrukcji, w których z dodatków znajduje się kilka małych przykładów paskudnego kodu.

---

<sup>4</sup> [http://www.objectmentor.com/omSolutions/agile\\_customers.html](http://www.objectmentor.com/omSolutions/agile_customers.html).

<sup>5</sup> [Maximilien], [George2003], [Janzen2005], [Nagappan2008].

Od czego zaczynasz czytanie takiej instrukcji? Jeżeli jesteś programistą, to zapewne od przykładów. Przechodzisz od razu do kodu, ponieważ wiesz, że kod powie Ci prawdę. Wszystkie te błyszczące fotografie, kółeczka i strzałki z pewnością wyglądają bardzo ładnie, ale jeżeli chcesz się dowiedzieć, jak tego używać, to patrzysz na kod.

Jeżeli będziesz postępować zgodnie z trzema przedstawionymi tu regułami, to każdy z napisanych przez Ciebie testów jednostkowych będzie opisywał sposób użycia systemu. Postępując zgodnie z tymi regułami, napiszesz testy jednostkowe opisujące metodę tworzenia każdego obiektu używanego w systemie, a nawet każdy sposób na utworzenie takiego obiektu. Powstaną testy jednostkowe opisujące wszystkie sensowne wywołania każdej funkcji systemu. Każda czynność, którą trzeba wykonać w systemie, zostanie dokładnie opisana za pomocą testów jednostkowych.

Testy jednostkowe są dokumentami. Opisują one najniższy poziom projektu danego systemu. Są niedwuznaczne, dokładne i napisane w języku zrozumiałym dla czytającego, a jednocześnie są na tyle formalne, że można je uruchomić. To najlepszy z możliwych rodzaj niskopoziomowej dokumentacji. Który zawodowiec nie chciałby tworzyć tak wspaniałej dokumentacji?

## **Projekt**

Jeżeli postępując zgodnie z tymi trzema prawami, najpierw napiszesz testy, staniesz przed pewnym dylematem. Często dokładnie wiesz, jaki kod chcesz napisać, ale zasada testu jednostkowego nakazuje Ci najpierw napisać test, który nie zostanie zaliczony! To znaczy, że musisz przetestować kod, który dopiero zaczniesz pisać.

Problem z testowaniem takiego kodu polega na tym, że najpierw musisz go wydzielić. Często trudno testuje się funkcję, jeżeli ta funkcja wywołuje inne funkcje. Aby napisać taki test, musisz wymyślić sposób na oddzielenie danej funkcji od pozostałych. Oznacza to, że konieczność napisania najpierw testu zmusza Cię do wymyślenia *dobrego projektu*.

Jeżeli nie napiszesz najpierw testu, to nic nie powstrzyma Cię przed połączeniem wielu funkcji w niepoddającą się testowaniu masę. Jeżeli test napiszesz później, to prawdopodobnie uda Ci się przetestować wejścia i wyjścia tej masy, ale przetestowanie poszczególnych funkcji może być bardzo utrudnione.

Oznacza to, że postępując zgodnie z trzema prawami i pisząc najpierw testy, wytwarzasz siłę, która zmusza Cię do tworzenia lepiej odizolowanego projektu. A który profesjonalista nie skorzystałby z narzędzi pozwalających mu uzyskać lepszy projekt oprogramowania?

Powiesz teraz: „Przecież testy mogę napisać później”. Niestety, nie możesz. Naprawdę nie. Oczywiście możesz później napisać *jakieś* testy, a jeżeli się bardzo postarasz, to możesz nawet osiągnąć wysokie pokrycie kodu. Ale testy pisane po powstaniu produktywnego kodu są tylko rodzajem *obrony*. Testy pisane przed powstaniem kodu są bardzo *ofensywne*.

Testy powstające po fakcie są pisane przez osobę, która poznała dokładnie testowany kod i doskonale wie, jak został rozwiązany dany problem. Nie ma takiej możliwości, żeby te testy były tak wnikliwe jak testy pisane przed faktem.

## Profesjonalne rozwiązanie

Wniosek z tych wszystkich rozważań jest taki, że TDD to bardzo profesjonalne rozwiązanie. Jest to dziedzina rozwijająca pewność siebie, odwagę, zmniejszającą liczbę błędów, poprawiająca dokumentację oraz projekt. Patrząc na to z tej perspektywy, nieużywanie tej metody można uznać za *nieprofesjonalne*.

## Czym TDD nie jest

Mimo wszystkich swoich zalet TDD nie jest żadną religią ani magicznym zaklęciem. Postępowanie zgodnie z trzema regułami nie zagwarantuje Ci żadnej z opisanych zalet. Nadal możesz tworzyć brzydkie kod, nawet jeżeli najpierw napiszesz do niego testy. Co więcej, możesz pisać paskudne testy.

Trzeba też pamiętać, że czasami stosowanie się do trzech reguł jest niepraktyczne i nieodpowiednie. Takie sytuacje są rzadkie, ale jednak istnieją. Żaden zawodowy programista nie powinien nigdy stosować danej metodologii, jeżeli ta przysparza mu kłopotów, a nie ułatwia pracę.

## Bibliografia

[Maximilien]: E. Michael Maximilien, Laurie Williams, „Assessing Test-Driven Development at IBM”, [http://collaboration.csc.ncsu.edu/laurie/Papers/MAXIMILIEN\\_WILLIAMS.PDF](http://collaboration.csc.ncsu.edu/laurie/Papers/MAXIMILIEN_WILLIAMS.PDF).

[George2003]: B. George i L. Williams, „An Initial Investigation of Test-Driven Development in Industry”, <http://collaboration.csc.ncsu.edu/laurie/Papers/TDDpaperv8.pdf>.

[Janzen2005]: D. Janzen i H. Saedian, *Test-driven development concepts, taxonomy, and future direction*, „IEEE Computer”, t. 38: 2005, nr 9, s. 43 – 50.

[Nagappan2008]: Nachiappan Nagappan, E. Michael Maximilien, Thirumalesh Bhat i Laurie Williams, „Realizing quality improvement through test driven development: results and experiences of four industrial teams”, Springer Science + Business Media, 2008: [http://research.microsoft.com/en-us/groups/ese/nagappan\\_tdd.pdf](http://research.microsoft.com/en-us/groups/ese/nagappan_tdd.pdf).



---

# ĆWICZENIA

---



Każdy zawodowiec rozwija się w swoim zawodzie, stosując ćwiczenia poprawiające różne umiejętności. Muzycy ciągle powtarzają gamy. Gracze futbolu amerykańskiego biegają przez opony. Lekarze ćwiczą szwy i metody chirurgiczne. Prawnicy ćwiczą się w argumentowaniu. Żołnierze powtarzają różne misje. Gdy chodzi o wydajność, każdy profesjonalista ćwiczy. W tym rozdziale skoncentrujemy się na różnych sposobach pozwalających programistom ćwiczyć się w swoim zawodzie.

## Kilka ćwiczeń w tle

Ćwiczenia nie są nowym pomysłem w dziedzinie programowania, ale aż do nowego tysiąclecia takie działania nie były uważały za ćwiczenia. Chyba pierwszy przykład formalnego programu do ćwiczeń został wydrukowany na 6. stronie [K&R-C].

```
main()
{
    printf("hello, world\n");
}
```

Kto z nas nie napisał tego programu w tej lub innej formie? Używamy go jako sposób na sprawdzenie nowego środowiska lub nowego języka. Pisanie i uruchamianie tego programu jest dowodem, że jesteśmy w stanie napisać i uruchomić *dowolny* program.

Gdy byłem znacznie młodszy, jednym z pierwszych programów, jakie pisałem na nowym komputerze, był SQUIN, który obliczał kwadraty liczb całkowitych. Napisałem go w asemblerze, BASIC-u, FORTRAN-ie, COBOL-u i setkach innych języków. Był to dla mnie sposób, by dowieść, że mogę zmusić komputer do zrobienia tego, co mu zadam.

Na początku lat 80. w sklepach zaczęły pojawiać się komputery osobiste. Gdy tylko dostałem w swoje ręce jeden z nich, czy był to VIC-20, Commodore-64, czy też TRS-80, zawsze pisałem na nim mały program, który wypisywał na ekranie niekończący się ciąg znaków ukośnika (\) i lewego ukośnika (/). Wzory generowane przez taki program były całkiem przyjemne, a jednocześnie wyglądały na bardziej skomplikowane niż sam program, który je wytworzył.

Mimo że takie programy zdecydowanie były ćwiczeniami, programiści ogólnie nie stosowali ćwiczeń. Po prostu nigdy nikomu nie przyszło to na myśl. Byliśmy zbyt zajęci pisaniem kodu, żeby w ogóle pomyśleć o ćwiczeniu naszych umiejętności. Poza tym jaki miałby być tego cel? Przez całe lata programowania nie potrzebowałem umiejętności szybkiego reagowania ani szczególnie sprawnych palców. Do końca lat 70. nie używaliśmy edytorów ekranowych. Większość czasu spędzaliśmy w oczekiwaniu na zakończenie pracy kompilatora albo na debugowaniu straszliwie długich ciągów kodu. Nie wymyślimy jeszcze króciutkich cykli TDD, dlatego niepotrzebne było nam to szczegółowe dopasowanie do środowiska, które można uzyskać tylko za pomocą ćwiczeń.

## Dwadzieścia dwa zera

Od początków programowania trochę się jednak zmieniło. A niektóre rzeczy zmieniły się naprawdę mocno. Z kolei pewne rzeczy nie zmieniły się prawie wcale.

Jedną z pierwszych maszyn, na które pisałem programy, było PDP-8/I. Maszyna ta miała cykl o czasie 1,5 mikrosekundy. Do dyspozycji miałem 4096 12-bitowych słów pamięci operacyjnej. Cały komputer miał wielkość lodówki i pożerał całkiem sporo energii elektrycznej.

Miałem też dysk, na którym można było zapisać 32 K 12-bitowych słów. Z komputerem porozumiewałem się za pośrednictwem teletekstu o prędkości 10 znaków na sekundę. Wtedy sądziliśmy, że mamy *potężną* maszynę, za pomocą której tworzymy prawdziwe cuda.

Ostatnio kupiłem nowego laptopa MacBook Pro. Ma dwurdzeniowy procesor z zegarem 2,8 GHz, 8 GB pamięci RAM, dysk SSD o pojemności 512 GB oraz 17-calowy ekran o rozdzielczości 1920×1200 pikseli. Mogę nosić go ze sobą w pleczaku, a pracować, kładąc go na kolanach. Pobiera niecałe 85 watów.

Mój laptop jest osiem tysięcy razy szybszy, ma dwa miliony razy więcej pamięci, sześć milionów razy więcej miejsca na dysku i pobiera zaledwie 1% energii, zajmując tylko 1% przestrzeni starego PDP-8/I i kosztując zaledwie jedną dwudziestą piątą jego ceny. Policzymy zatem:

$$8000 \times 2\,000\,000 \times 16\,000\,000 \times 100 \times 100 \times 25 = 6,4 \times 10^{22}$$

To *gigantyczna* wartość. Mówimy tu o 22 rzędach wielkości! Mniej więcej tyle angstromów jest między Ziemią a Alfa Centauri. Tyle elektronów ukrywa się w srebrnym dolarze. Tyle wynosi masa Ziemi w jednostkach Michaela Moore'a. Ta liczba jest naprawdę wielka. A przy tym spokojnie leży sobie na moich kolanach. Na Twoich pewnie też.

I co robię z tą mocą powiększoną o dziesięć do dwudziestej drugiej? Mniej więcej to samo, co robiłem z PDP-8/I. Piszę instrukcję `if`, pętle `while` i *instrukcje przypisania*.

Oczywiście mam znacznie lepsze narzędzia do pisania tych instrukcji. Korzystam z dużo bardziej rozbudowanych języków, ale przez cały ten czas nie zmieniła się natura pisanych przeze mnie instrukcji. Programista z lat 60. bez większego problemu rozpoznałby znaczenie kodu z roku 2010. Plastelina, którą się bawimy, nie zmieniła się mocno w ciągu ostatnich dziesięcioleci.

## Czas wykonania

Bardzo zmienił się jednak *sposób*, w jaki pracujemy. W latach 60. musielibyśmy czekać dzień lub dwa, żeby zobaczyć w końcu efekty naszej komplikacji. Pod koniec lat 70. komplikacja programu o długości 50 000 wierszy trwała mniej więcej 45 minut. Nawet w latach 90. długie czasy komplikacji nie były niczym niezwykłym.

Dzisiaj programiści nie muszą czekać na zakończenie komplikacji<sup>1</sup>. Dzisiaj programiści mają do dyspozycji tak gigantyczną moc, że czerwono-zielony cykl refaktoryzacji mogą realizować w ciągu sekund.

---

<sup>1</sup> Fakt, że część programistów nadal czeka na jej zakończenie, jest prawdziwą tragedią i oznaką bezmyślności. Dzisiaj czasy komplikacji powinny być mierzone w sekundach, a nie minutach, a już na pewno nie w godzinach.

Na przykład pracuję nad projektem o nazwie FitNesse, który składa się z 64 000 wierszy kodu Javy. Pełna komplikacja projektu, połączona z wykonaniem wszystkich testów jednostkowych i integracyjnych, zajmuje mniej niż 4 minuty. Jeżeli wszystkie testy zostaną zaliczone, to wiem, że mogę wydać kolejną wersję. *Cały proces kontroli jakości, od kodu źródłowego do udostępnienia, zajmuje niecałe 4 minuty.* Czas samej komplikacji jest trudny do zmierzenia. Testy częstotliwe trwają kilka sekund. Oznacza to, że cały cykl komplikacji i testów mogę wykonywać dziesięć razy na minutę!

Nie zawsze warto działać aż tak szybko. Często znacznie lepszym rozwiązaniem jest zwolnić i *pomyśleć*<sup>2</sup>. Ale zdarzają się sytuacje, w których jak najszybsza realizacja kolejnych cykli jest działaniem *niewykle produktywnym*.

Jakiekolwiek szybkie działanie wymaga odpowiedniej praktyki. Szybkie działanie w pętli kodowania i testowania wymaga bardzo szybkiego podejmowania decyzji. A szybkie podejmowanie decyzji wymaga umiejętności rozpoznawania wielu sytuacji i problemów oraz *dopasowywania* do nich właściwych rozwiązań.

Zastanówmy się nad walką dwóch doświadczonych wojskowników. Każdy z nich musi rozpoznać zamiary przeciwnika i w ciągu milisekund odpowiednio na nie zareagować. W trakcie walki nie masz możliwości zatrzymania czasu, przeanalizowania pozycji i zastanowienia się nad właściwą reakcją. Walcząc, musisz po prostu *reagować*. Co więcej, reagowaniem zajmuje się Twoje *ciało*, podczas gdy umysł pracuje nad bardziej rozbudowaną strategią.

Podczas wykonywania kilku cykli tworzenia kodu i testowania na minutę to Twoje *ciało* wie, które klawisze ma naciskać. Pierwotna część Twojego umysłu rozpoznaje daną sytuację i reaguje w ciągu milisekund, stosując pasujące rozwiązanie, podczas gdy umysł może swobodnie koncentrować się na problemach wyższego poziomu.

Zarówno w przypadku wojskowników, jak i programistów szybkość jest uzależniona od *praktyki*. A w obu przypadkach uzyskuje się ją w podobny sposób. Wybieramy cały repertuar par problem – rozwiązanie i powtarzamy je tak długo, aż poznamy je na pamięć.

Zastanówmy się nad gitarzystą klasy Carlosa Santany. Muzyka powstająca w jego głowie po prostu przepływa przez jego palce. Nie zastanawia się on nad ułożeniem palców ani nad zastosowaniem właściwej techniki. Jego umysł może swobodnie planować rozbudowane melodie, podczas gdy jego ciało przekształca te plany w niskopoziomowe ruchy palców.

Jednak uzyskanie takiej łatwości grania wymaga *ćwiczeń*. Muzycy ćwiczą gamy i etiudy, powtarzając je do znudzenia, aż mogą grać je na ślepco.

---

<sup>2</sup> Ta technika jest nazywana przez Richa Hickeyego HDD, czyli *Hammock-Driven Development* (rozwój hamakowy).

## Dojo kodowania

Od 2001 roku prowadzę zajęcia demonstracyjne z TDD, które nazywam *Grę w kręgle* (ang. *The Bowling Game*)<sup>3</sup>. To małe i przyjemne ćwiczenie zajmuje około 30 minut. Tworzy najpierw konflikt w projekcie, potem buduje napięcie, a kończy się niespodzianką. Na podstawie tego przykładu napisałem cały rozdział książki [PPP2002].

Przez lata zademonstrowałem to kata setki, a może i tysiące razy. Sprawiło to, że jestem w tym *bardzo* dobry! Mogę je wykonywać na śpiąco. Zminimalizowałem liczbę naciśnięć klawiszy, dopasowałem nazwy zmiennych i zmieniłem strukturę algorytmu tak, że w końcu stała się doskonała. To było moje pierwsze kata, choć wtedy tego jeszcze nie wiedziałem.

W 2005 roku wziąłem udział w konferencji XP2005 w Sheffield, w Anglii. Skorzystałem z sesji o nazwie *Dojo kodowania* (ang. *Coding Dojo*) prowadzonej przez Laurenta Bossavita i Emmanuela Gaillota. Kazali wszystkim otworzyć swoje laptopy i tworzyć kod razem z nimi, podczas gdy oni wykorzystywali TDD, żeby przygotować *Grę życia Conwaya*. To, co robili, nazywali „kata”, a pierwszeństwo idei<sup>4</sup> przyznawali „Pragmatycznemu” Dave’owi Thomasowi<sup>5</sup>.

Od tego czasu wielu programistów przyjęło tę pochodzącą ze sztuk walki metaforę sesji ćwiczeń. Wygląda na to, że „dojo kodowania”<sup>6</sup> zostało bardzo dobrze przyjęte. Czasami grupa programistów spotyka się i razem ćwiczy, podobnie jak robią to wojownicy. Zdarza się też, że programiści ćwiczą samodzielnie, tak jak zdarza się to wojownikom.

Mniej więcej rok temu uczyłem grupę programistów z Omaha. W czasie obiadu poprosili mnie o wzięcie udziału w ich dojo kodowania. Obserwowałem, jak dwudziestu programistów otworzyło swoje laptopy i klawisz po klawiszu powtarzali ruchy prowadzącego, który wykonywał kata *Gra w kręgle*.

W ramach dojo wykonywanych jest kilka rodzajów działań. Oto niektóre z nich:

## Kata

W sztukach walki kata to szczegółowy zestaw ruchów, który symuluje jedną stronę w walce. Celem jest próba osiągnięcia perfekcji, do której można się zbliżyć, ale nigdy nie można jej osiągnąć. Wojownik stara się nauczyć swoje ciało, by doskonale wykonywało każdy ruch,

<sup>3</sup> Stała się ona bardzo popularnym kata, a wyszukiwanie tego hasła w Google zwraca wiele źródeł. Oryginał można znaleźć tutaj: <http://butunclebob.com/Articles/UncleBob.TheBowlingGameKata>.

<sup>4</sup> <http://codekata.pragprog.com>.

<sup>5</sup> Określenie „Pragmatyczny” ma go odróżniać od „Wielkiego” Dave’a Thomasa z OTI.

<sup>6</sup> <http://codingdojo.org/>.

a następnie połączyć poszczególne ruchy w płynny taniec. Dobrze wykonane kata naprawdę przyjemnie się ogląda.

Mimo całego ich uroku celem nauki kata nie jest prezentowanie ich na scenie. Ich zadaniem jest takie wyćwiczenie umysłu i ciała, żeby reagowały one na szczególne sytuacje w walce. Celem jest sprawienie, że takie doskonałe ruchy staną się automatyczne i instynktowne, tak by w razie potrzeby były zawsze do dyspozycji.

W programowaniu kata jest dokładnym zestawem chronologicznie ułożonych naciśnięć klawiszy oraz ruchów myszą symulujących rozwiązanie określonego problemu programistycznego. Nie chodzi o samo wymyślenie rozwiązania problemu, ponieważ ono jest już znane, ale o wyćwiczenie związań z nim ruchów oraz decyzji.

Tutaj również celem jest dążenie do perfekcji. Powtarzasz ćwiczenie raz za razem, aby wyćwiczyć mózg i palce w wykonywaniu odpowiednich ruchów i reagowaniu. Podczas takich ćwiczeń zauważysz drobne poprawy każdego wykonywanego ruchu albo i całego rozwiązania.

Ćwiczenie całego zbioru różnych kata jest dobrą metodą na nauczenie się skrótów klawiszowych oraz idiomów nawigacyjnych. Jest też dobrą metodą na naukę takich metodologii jak TDD lub CI. Najważniejsze jest jednak to, że jest to świetna metoda na wprasowanie rozwiązań typowych problemów w swoją podświadomość, tak żeby od razu wiedzieć, jak je rozwiązywać w momencie, gdy się na nie natknieniemy.

Podobnie jak wojownik programisty powinien znać kilka różnych kata i ćwiczyć je regularnie, tak żeby nie wyleciały mu z pamięci. Wiele kata zostało zapisanych na stronie <http://katas.softwarecraftsmanship.org>. Inne można znaleźć pod adresem <http://codekata.pragprog.com>. Do moich ulubionych należą:

- *Gra w kręgle (The Bowling Game)*: <http://butunclebob.com/ArticleS.UncleBob.TheBowlingGameKata>.
- *Czynniki pierwsze (Prime Factors)*: <http://butunclebob.com/ArticleS.UncleBob.ThePrimeFactorsKata>.
- *Zawijanie słów (Word Wrap)*: <http://thecleanncoder.blogspot.com/2010/10/craftsman-62-darkpath.html>.

Prawdziwym wyzwaniem jest nauczenie się kata tak dobrze, że można zmienić je w muzykę. To dopiero jest *trudne*<sup>7</sup>.

---

<sup>7</sup> <http://katas.softwarecraftsmanship.org/?p=71>.

## Wasa

W czasie gdy ćwiczyłem jujitsu, wiele czasu spędziliśmy w parach, ćwicząc nasze *wasa*. Wasa można przymierzyć do dwuosobowego kata. Trzeba dokładnie zapamiętać poszczególne ruchy, a następnie je odtworzyć. Jeden z partnerów odgrywa rolę agresora, a drugi próbuje się bronić. Wszystkie ruchy są powtarzane w nieskończoność, a potem ćwiczący zamieniają się rolami.

Programiści mogą ćwiczyć w podobny sposób, wykorzystując do tego grę *ping-pong*<sup>8</sup>. Dwóch partnerów wybiera w niej kata lub prosty problem. Jeden z nich pisze test jednostkowy, a drugi musi sprawić, że zostanie on zaliczony. Potem zamieniają się rolami.

Jeżeli partnerzy wybiorą standardowe kata, to wynik ćwiczenia jest znany, a programiści mogą ćwiczyć i krytykować wzajemne techniki posługiwania się klawiaturą lub myszą, a także poziom zapamiętania danego kata. Jeżeli jednak wybiorą nowy problem do rozwiązania, to gra staje się trochę bardziej interesująca. Programista piszący test ma ogromną władzę nad kształtem rozwiązania. Ma też wielkie możliwości tworzenia ograniczeń. Na przykład jeżeli programista zdecyduje się zaimplementować algorytm sortowania, to piszący test może łatwo nałożyć ograniczenia odnośnie do prędkości i zapotrzebowania na pamięć, co będzie wyzwaniem dla partnera. To sprawia, że cała gra może stać się poważną rywalizacją... i niezłą zabawą.

## Randori

*Randori* jest walką w stylu wolnym. W naszym dojo jujitsu przygotowywaliśmy kilka rodzajów starcia, a następnie je odgrywaliśmy. Czasami jedna osoba miała się bronić, podczas gdy pozostali starali się ją po kolei atakować. Niekiedy dwie osoby miały atakować jednego obrońcę (zwykle był nim sensei, który niemal zawsze wygrywał). Kiedy indziej stawaliśmy dwóch na dwóch itd.

Symulowana walka nie najlepiej przekłada się na programowanie, ale istnieje pewna gra nazywana randori, którą wykorzystywałem podczas wielu dojo programowania. Przypomina ona wasa z dwójką partnerów starających się rozwiązać określony problem. Tym razem jednak w zabawie bierze udział więcej osób, a w regułach zostaje wprowadzona drobna zmiana. Zawartość ekranu jest prezentowana na ścianie. Jedna osoba pisze test i siada na swoje miejsce. Następna osoba sprawia, że test zostaje zaliczony, i pisze kolejny test. Można to powtarzać w kolejności dookoła stołu, ale poszczególne osoby mogą się po prostu zgłaszać, jeżeli czują taką potrzebę. W obu przypadkach wykonywanie takiego ćwiczenia może sprawiać sporo radości.

<sup>8</sup> <http://c2.com/cgi/wiki?PairProgrammingPingPongPattern>.

Zadziwiające jest, jak wiele można się z takich sesji nauczyć. Możesz w ten sposób poznać metody rozwiązywania problemów stosowane przez inne osoby. Takie informacje pomagają poszerzyć własne horyzonty i poprawić umiejętności.

## **Zwiększenie doświadczenia**

Profesjonalni programiści zwykle cierpią z powodu braku zróżnicowania stawianych przed nimi problemów. Pracodawcy zazwyczaj wymuszają stosowanie jednego języka, platformy oraz dziedziny, w których programista musi się poruszać. Przy braku czynnika poszerzającego horyzonty może to doprowadzić do niezdrowego zawężenia żywiołu i umysłu. Nierzadko zdarza się, że tak ograniczeni programiści są całkowicie nieprzygotowani na zmiany, jakie okresowo przetaczają się przez naszą branżę.

### **Otwarte źródła**

Jednym ze sposobów wyrwania się z zastoju jest to, co często robią lekarze i prawnicy: wykonanie pewnych prac *pro bono*, na przykład przez współpracę przy jakimś otwartoźródłowym projekcie. W sieci istnieją setki takich projektów i nie ma chyba lepszego sposobu na poszerzenie repertuaru swoich umiejętności niż praca nad czymś, na czym zależy komuś innemu.

Jeżeli zatem programujesz normalnie w Javie, to dołącz do projektu pisanej w Ruby. Jeżeli w pracy wiele piszesz w C++, to znajdź sobie projekt w Pythonie i postaraj się go rozwijać.

### **Etyka pracy**

Profesjonalni programiści ćwiczenia wykonują w swoim czasie wolnym. Utrwalanie i rozwijanie Twoich umiejętności nie jest zadaniem Twojego pracodawcy. Nie on ma się zajmować uatrakcyjnianiem Twojego CV. Pacjenci nie płacą lekarzom za ćwiczenie zakładania szwów. Fani footbolu nie płacą (zwykle) zawodnikom za patrzenie, jak biegają przez opony. Chodzący chętnie na koncerty nie płacą muzykom za ich ćwiczenia i granie gam. A pracodawcy programistów nie powinni płacić za czas poświęcony przez nich na ćwiczenia.

Skoro czas ćwiczeń należy całkowicie do Ciebie, nie musisz korzystać z języków i platform stosowanych przez Twojego pracodawcę. Wybierz sobie dowolny język i rozwijaj swój talent poligloty. Jeżeli w pracy poruszasz się w środowisku .NET, to w czasie przerwy obiadowej albo w domu poćwicz trochę programowanie w Javie albo Ruby.

## Wnioski

W ten lub inny sposób *każdy* zawodowiec musi ćwiczyć. Robi to, ponieważ chce wykonywać swoją pracę najlepiej, jak to możliwe. Co więcej, na ćwiczenia poświęca swój prywatny czas, gdyż utrzymywanie umiejętności na wysokim poziomie uznaże za obowiązek swój, a nie pracodawcy. Ćwiczenia wykonuje się wtedy, gdy Ci za to *nie* płacą. Wykonuje się je po to, żeby uzyskać *dobrą zapłatę* za swoją pracę.

## Bibliografia

[K&R-C]: Brian W. Kernighan i Dennis M. Ritchie, *The C Programming Language*, Upper Saddle River, NJ: Prentice Hall, 1975.

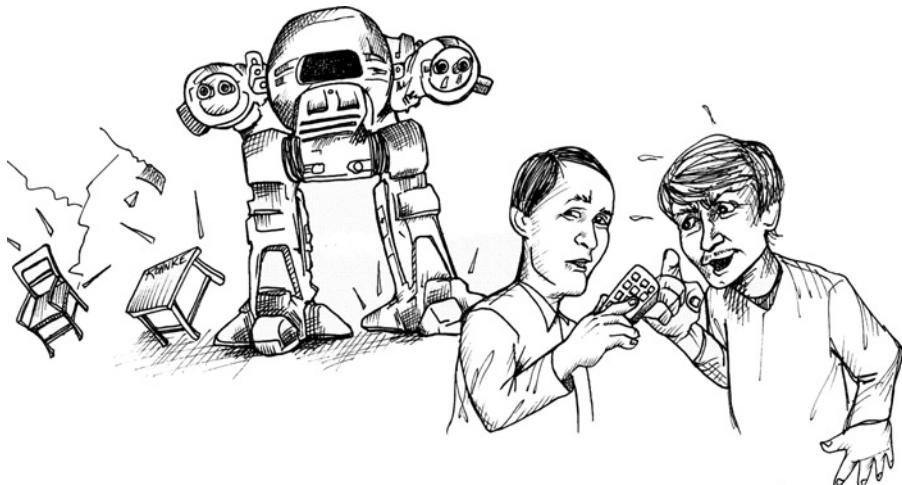
[PPP2002]: Robert C. Martin, *Agile Software Development. Principles, Patterns, and Practices*, Upper Saddle River, NJ: Prentice Hall, 2003.



---

# 7 TESTY AKCEPTACYJNE

---



Rolę profesjonalnego programisty można podzielić na część komunikacyjną i część programistyczną. Pamiętaj, że programistów również dotyczy zasada: śmieci na wejściu, śmieci na wyjściu. Z tego właśnie powodu profesjonalni programiści starają się upewnić, że ich komunikacja z pozostałymi członkami zespołu i całą firmą jest dokładna i niedwuznaczna.

## Komunikowanie wymagań

Jednym z najczęstszych problemów w komunikacji między programistami i menedżerami jest zdefiniowanie wymagań. Biznesmeni opowiadają o tym, co — jak im się wydaje — jest im potrzebne. Następnie programiści tworzą coś, co — jak im się wydaje — opisał

biznesmen. Przynajmniej tak powinno to działać. W rzeczywistości komunikacja dotycząca wymagań jest niezmiernie skomplikowana, a cały proces narażony na błędy.

W 1979 roku, kiedy pracowałem dla firmy Teradyne, odwiedził mnie Tom, menedżer instalacji i serwisu. Poprosił mnie, żebym pokazał mu, jak użyć edytora tekstu ED-402 do przygotowania prostego systemu zgłaszania problemów.

ED-402 był własnościowym edytorem napisanym dla komputera M365, który był klonem PDP-8 używanym w Teradyne. Jako narzędzie do edycji tekstu był naprawdę bardzo rozbudowany. Miał nawet wbudowany język skryptowy, którego używaliśmy do przygotowywania różnego rodzaju aplikacji działających na tekstach.

Tom nie był programistą, ale potrzebna mu aplikacja była tak prosta, że poprosił mnie o nauczenie go szybko tej sztuki, tak aby mógł samodzielnie napisać program. W swojej naiwności uznałem, że to możliwe. W końcu ten język skryptowy był tylko troszkę bardziej zaawansowanym wariantem języka makr stosowanego do edycji poleceń, wyposażonym w podstawowe konstrukcje decyzyjne i zapętlające.

Usiedliśmy zatem razem i zapytałem, co planowana przez niego aplikacja miałyby robić. Zaczął od początkowego ekranu wprowadzania. Pokazałem mu zatem, jak utworzyć plik tekstowy zawierający instrukcje skryptu i jak wpisywać symboliczną reprezentację poleceń edycyjnych składających się na skrypt. Gdy jednak spojrzałem w jego oczy, zobaczyłem tylko pustkę. Moje wyjaśnienia nie miały dla niego żadnego sensu.

Pierwszy raz się z czymś takim spotkałem. Dla mnie było to oczywiste, że polecenia edytora są zapisywane za pomocą symboli. Na przykład aby zapisać polecenie control-B (umieszczało ono kurSOR na początku bieżącego wiersza), wystarczy wpisać w pliku skryptu znaki ^B. Dla Toma nie miało to żadnego sensu. Nie był w stanie przestawić się z bezpośredniego edytowania pliku na edytowanie pliku, który będzie edytował plik.

Tom nie był głupi. Szybko zorientował się, że zadanie jest znacznie trudniejsze, niż mu się początkowo wydawało, a nie chciał inwestować czasu i energii na uczenie się czegoś tak niesamowicie skomplikowanego jak używanie edytora do sterowania edytorem.

I tak krok po kroku zacząłem sam tworzyć całą aplikację, podczas gdy on siedział obok i mi się przyglądał. W ciągu pierwszych dwudziestu minut stało się jasne, że jego zainteresowanie przeniosło się z myślenia, jak to zrobić samodzielnie, na upewnianie się, że *ja* robię to, czego *on* chce.

Zajęło nam to cały dzień. On opisywał kolejne funkcje, a ja je implementowałem pod jego okiem. Cykl takiej pracy trwał jakieś pięć minut, dlatego nie było sensu, żeby zajmował się on czymkolwiek innym. Prosił mnie o zrobienie funkcji X, a pięć minut później funkcja X działała.

Często musiał rysować swoją wizję na kartce papieru. Niektóre z jego pomysłów trudno byłoby zrealizować za pomocą ED-402, dlatego proponowałem inne rozwiązania. Ostatecznie zgadzaliśmy się na rozwiązanie pośrednie, a potem starałem się je zastosować.

W końcu próbowałem to, co zostało zaimplementowane, a on zmieniał zdanie. Mówił coś w rodzaju: „No tak, ale to jednak nie działa tak, jak sobie wyobrażałem. Może spróbujemy czegoś innego”.

Godzinę po godzinie próbowałem, przedstawiałem i nadawałem aplikacji ostateczny kształt. Próbowałem jednej rzeczy, potem innej, a potem jeszcze innej. Stawało się dla mnie coraz wyraźniejsze, że to *on* był tutaj rzeźbiarzem, a ja tylko trzymanym przez niego narzędziem.

W końcu udało się nam przygotować aplikację, o której myślał Tom, choć on nadal nie miał pojęcia o tym, jak mógłby sam napisać kolejny program. Z kolei ja dostałem porządną lekcję na temat tego, jak klienci dowiadują się, czego im naprawdę potrzeba. Nauczyłem się, że ich wizja poszczególnych funkcji często nie jest w stanie przetrwać rzeczywistego kontaktu z komputerem.

## Przedwczesna dokładność

Zarówno programiści, jak i biznesmeni często wpadają w pułapkę przedwczesnej dokładności. Biznesmeni chcą od samego początku, jeszcze przed autoryzacją projektu, wiedzieć, co właściwie dostaną. Programiści z kolei chcą od samego początku, jeszcze przed podaniem jakichkolwiek szacunków, wiedzieć, co mają dostarczyć. Obie strony chcieliby mieć dane o nieosiągalnej dokładności, ale często są w stanie poświęcić fortunę, żeby mimo wszystko je otrzymać.

## Zasada niepewności

Problem polega na tym, że pewne rzeczy wyglądają inaczej na papierze, a inaczej w działającym systemie. Gdy biznesmen widzi to, co opisał wcześniej, w gotowym systemie, często zdaje sobie sprawę, że nie o to mu chodziło. Gdy w końcu zobaczy swoje wymagania w akcji, stwierdza, że ma lepszy pomysł, który najczęściej nie ma nic wspólnego z tym, co właśnie widzi.

W grę wchodzi tu pewnego rodzaju efekt obserwatora albo zasada niepewności. Kiedy demonstrujesz funkcję biznesmenom, dajesz im więcej informacji, niż mieli wcześniej, co wpływa na ich postrzeganie systemu jako całości.

Ostatecznie okazuje się, że im dokładniej zdefiniowane będą wymagania systemu, tym mniejsze będą miały one znaczenie w czasie implementowania.

## **Niepokojące szacunki**

Programiści również mogą wpaść w pułapkę zbyt dużej dokładności. Wiedzą, że muszą oszacować system, i często sądzą, że to wymaga dokładnych informacji. Nieprawda.

Po pierwsze nawet z doskonale dokładnymi informacjami Twoje szacunki będą miały sporą niedokładność. Po drugie zasada niepewności te wstępnie dokładne dane przerabia na sieczkę. Wymagania odnośnie do systemu na pewno się zmieniają, przez co początkowa precyzja staje się nieistotna.

Profesjonalni programiści wiedzą, że szacunki mogą i powinny być tworzone na podstawie mało dokładnych wymagań, a jednocześnie muszą pamiętać, że *szacunki są tylko szacunkami*. Na diagramach przedstawiających takie szacunki dodają jeszcze paski prezentujące możliwy błąd, tak żeby uwidoczyć menedżerom tę niepewność. (Zajrzyj do rozdziału 10., „Szacowanie”).

## **Późna wieloznaczność**

Rozwiązaniem problemu przedwczesnej dokładności jest jak najdłuższe opóźnianie pojawienia się tej precyzji. Profesjonalni programiści nie zastanawiają się nad danym wymogiem do czasu, aż przystępują do jego implementowania. To może jednak powodować inny rodzaj kłopotów: późną wieloznaczność.

Udziałowcy projektu często się ze sobą nie zgadzają. W takiej sytuacji znacznie łatwiej jest im w jakiś sposób *obejść* ten brak porozumienia, niż faktycznie rozwiązać problem. Wyszukiwane są zatem sposoby takiego wyrażenia wymagania, z którym każdy może się zgodzić, choć to wcale nie rozwiązuje problemu. Jakiś czas temu słyszałem, jak Tom DeMarco powiedział: „Wieloznaczność w dokumencie wymagań systemu jest odzwierciedleniem konfliktu między udziałowcami”<sup>1</sup>.

Oczywiście do wytworzenia wieloznacznego wymagania nie trzeba żadnej kłótni ani braku zgody. Czasami udziałowcy zakładają, że czytelnicy dokumentu domyślą się, co oni mieli na myśli.

We własnym kontekście mogą doskonale znać swoje wymagania, ale dla programisty, który później czyta ten tekst, może on mieć całkowicie odmienne znaczenie. Ten rodzaj kontekstowej wieloznaczności może pojawić się również wtedy, gdy klienci i programiści rozmawiają bezpośrednio ze sobą.

Sam (udziałowiec): „Dobrze, natomiast te pliki trzeba będzie zapisywać w kopii zapasowej”.

---

<sup>1</sup> XP Immersion, 3 maja 2000, <http://c2.com/cgi/wiki?TomsTalkAtXpImmersionThree>.

Paula: „W porządku, jak często tworzyć taką kopię?”.

Sam: „Codziennie”.

Paula: „Dobrze. A gdzie mamy ją zapisywać?”.

Sam: „Co masz na myśli?”.

Paula: „Chcesz, żeby kopię zapisywać w jakimś konkretnym podkatalogu?”.

Sam: „To całkiem niezły pomysł”.

Paula: „Jak nazwać ten podkatalog?”.

Sam: „Może po prostu *backup*?”.

Paula: „Pewnie, nie ma problemu. A zatem codziennie będziemy zapisywać kopię zapasową w podkatalogu. A kiedy?”.

Sam: „Codziennie”.

Paula: „Nie, mam na myśli, o której godzinie mamy zapisywać te pliki?”.

Sam: „Och, to nie ma znaczenia”.

Paula: „W południe?”.

Sam: „No nie! Nie w czasie pracy. Lepiej będzie o północy”.

Paula: „A zatem o północy”.

Sam: „Świetnie, dzięki”.

Paula: „Zawsze do usług”.

Później Paula opowiada o tym zadaniu Peterowi, swojemu koledze z zespołu.

Paula: „Musimy też codziennie o północy kopować pliki protokołu do podkatalogu o nazwie *backup*”.

Peter: „Dobra, jak nazwać plik takiej kopii?”.

Paula: „Myślę, że *log.backup* powinno pasować”.

Peter: „Się zrobi”.

W innym biurze Sam rozmawia przez telefon z klientem.

Sam: „Oczywiście, pliki protokołów będą zachowywane”.

Carl: „Świetnie. To bardzo ważne, żebyśmy nigdy nie stracili protokołów. Musimy do nich wracać nawet po miesiącach albo i latach przy okazji jakichś wyłączeń, awarii albo konfliktów”.

Sam: „Bez obaw, właśnie rozmawiałem z Paulą. Pliki będą zapisywane codziennie o północy w katalogu o nazwie *backup*”.

Carl: „Brzmi nieźle”.

Zakładam, że widzisz już niejednoznaczność. Klient oczekuje, że zachowywane będą wszystkie pliki protokołów, ale Paula sądzi, że chodzi o zachowanie protokołów z ostatniego dnia. Gdy klient będzie szukał plików z kilku ostatnich miesięcy, znajdzie tylko wczorajsze.

W tym przypadku to Paula i Sam dopuścili się niedopatrzenia. Zadaniem profesjonalnych programistów (i menedżerów) jest usunięcie z wymagań systemu wszelkich nieścisłości.

To naprawdę *trudne* zadanie, a znam tylko jeden sposób na jego wykonanie.

## Testy akceptacyjne

Pojęcie *testów akceptacyjnych* jest używane zdecydowanie zbyt szeroko. Niektórzy uważają, że są to testy przeprowadzane przez użytkowników, zanim zaakceptują dane wydanie systemu. Inni sądzą, że chodzi o testy wykonywane przez dział kontroli jakości. W tym rozdziale zdefiniujemy testy akceptacyjne jako testy napisane przez współpracujących ze sobą udziałowców i programistów *w celu zdefiniowania, kiedy dane wymaganie zostaje spełnione*.

### Definicja gotowego

Jedną z największych niejednoznaczności, z jakimi stykają się zawodowi programiści, jest niejednoznaczne określenie, kiedy coś jest gotowe. Gdy programista mówi, że wykonał dane zadanie, to co to właściwie oznacza? Czy zadanie zostało wykonane tak, że programista może przekazać daną funkcję dalej i mieć całkowitą pewność, że działa ona prawidłowo? Czy może oznacza to, że wszystko jest gotowe dla działu kontroli jakości? A może tylko napisał kod funkcji i raz ją uruchomił, ale nie przetestował jej dokładnie?

Pracowałem już z zespołami, które miały bardzo różne definicje słowa „gotowe” i „zakończone”. Jeden z takich zespołów stosował nawet dwa pojęcia: „gotowe” i „gotowe-gotowe”.

Profesjonalni programiści mają tylko jedną definicję tego słowa: gotowe znaczy *gotowe*. Oznacza ono, że kod został napisany, wszystkie testy zostały zaliczone, a funkcja została zaakceptowana przez dział kontroli jakości i udziałowców projektu.

Jak jednak można osiągnąć aż tak wysoki poziom „gotowości” i nadal szybko rozwijać system w każdej iteracji? Trzeba przygotować zestaw automatycznych testów, po których zaliczeniu stwierdzane jest, że kod spełnia wszystkie kryteria! Testy akceptacyjne rozwijanej funkcji zostają zaliczone i wszystko jest *gotowe*.

Profesjonalni programiści rozwijają definicję wymagań systemu tak daleko, że na ich podstawie opracowują testy akceptacyjne. Pracują przy tym z udziałowcami projektu i działem kontroli jakości, aby upewnić się, że przygotowane testy akceptacyjne w pełni opisują wymagania.

- Sam: „Dobrze, natomiast te pliki trzeba będzie zapisywać w kopii zapasowej”.
- Paula: „W porządku, jak często tworzyć taką kopię?”.
- Sam: „Codziennie”.
- Paula: „Dobrze. A gdzie mamy ją zapisywać?”.
- Sam: „Co masz na myśli?”.
- Paula: „Chcesz, żeby kopię zapisywać w jakimś konkretnym podkatalogu?”.
- Sam: „To całkiem niezły pomysł”.
- Paula: „Jak nazwać ten podkatalog?”.
- Sam: „Może po prostu *backup*?”.
- Tom (tester): „Nie, *backup* to zbyt ogólna nazwa. Co właściwie ma znaleźć się w tym katalogu?”.
- Sam: „Kopie zapasowe”.
- Tom: „Kopie czego?”.
- Sam: „Plików protokołów”.
- Paula: „Ale mamy tylko jeden plik protokołu”.
- Sam: „Nie, jest ich kilka. Po jednym na każdy dzień”.
- Tom: „To znaczy, że jest jeden *aktywny* plik protokołu i wiele kopii zapasowych?”.
- Sam: „To chyba oczywiste”.
- Paula: „O! A myślałam, że chodzi tylko o tymczasową kopię zapasową”.
- Sam: „Nie, nie! Klient chce przechowywać te pliki w nieskończoność”.
- Paula: „A to coś nowego. Dobrze, że teraz się o tym dowiedziałam”.
- Tom: „A zatem nazwa podkatalogu powinna mówić nam, co się w nim znajduje”.
- Sam: „Znajdę się w nim wszystkie pliki nieaktywnych protokołów”.
- Tom: „Czyli można go nazwać *old\_inactive\_logs*”.
- Sam: „Brzmi dobrze”.
- Tom: „A kiedy ten katalog będzie tworzony?”.
- Sam: „Słucham?”.
- Paula: „Możemy tworzyć ten katalog przy uruchamianiu systemu, ale tylko w przypadku, gdy jeszcze nie istnieje”.
- Tom: „No to mamy pierwszy test. Muszę uruchomić system i sprawdzić, czy utworzony został katalog *old\_inactive\_logs*. Potem mogę do tego katalogu dodać jakiś plik. Następnie muszę zamknąć system, uruchomić go ponownie i sprawdzić, czy katalog i umieszczony w nim plik jeszcze istnieją”.

Paula: „Ten test zajmie ci strasznie dużo czasu. Aktualnie czas uruchomienia systemu to 20 sekund i ciągle się wydłuża. Poza tym nie chcę kompilować całego systemu na potrzeby uruchomienia testu akceptacyjnego”.

Tom: „Co zatem proponujesz?”.

Paula: „Utworzmy klasę SystemStarter. Program główny załaduje taki starter z dołączoną listą obiektów StartupCommand zbudowanych zgodnie ze wzorcem **polecenia**. Następnie program nakaże obiekowi SystemStarter uruchomić wszystkie obiekty z listy. Jedno z tych poleceń utworzy katalog, jeżeli nie będzie jeszcze istniał”.

Tom: „OK. Czyli wszystko, co muszę zrobić, to przetestować ten konkretny obiekt wywiedziony z klasy StartupCommand. Mogę napisać prosty test w FitNesse”.

Tom podchodzi do tablicy.

„Pierwsza część będzie wyglądała mniej więcej tak”:

zakładając, że mamy polecenie LogFileDirectoryStartupCommand  
zakładając, że katalog old\_inactive\_logs nie istnieje  
gdy polecenie zostanie uruchomione  
to katalog old\_inactive\_logs powinien istnieć  
i być pusty

„Druga część testu musi wyglądać tak”:

zakładając, że mamy polecenie LogFileDirectoryStartupCommand  
zakładając, że katalog old\_inactive\_logs istnieje  
i zawiera plik o nazwie x  
gdy polecenie zostanie uruchomione  
to katalog old\_interactive\_logs powinien nadal istnieć  
i powinien nadal zawierać plik o nazwie x

Paula: „To chyba powinno załatwić sprawę”.

Sam: „Rany! To wszystko jest naprawdę konieczne?”.

Paula: „Sam, które z tych zdań jest na tyle nieistotne, że można je pominąć?”.

Sam: „Chodzi mi o to, że to wygląda na strasznie dużo roboty, takie wymyślanie i zapisywanie tych wszystkich testów”.

Tom: „To jest sporo pracy, ale nie jest jej więcej niż przy ręcznym zapisywaniu planu testów. A jeszcze więcej pracy wymaga ręczne powtarzanie tego samego testu”.

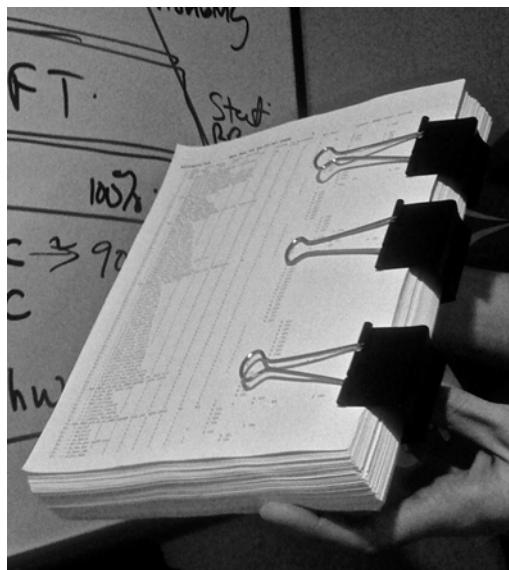
## Komunikacja

Zadaniem testów akceptacyjnych jest odpowiednia komunikacja, wprowadzenie jasności i precyzji. Zgadzając się na to, programiści, udziałowcy i testerzy ustalają, jakie jest planowane zachowanie systemu. Za uzyskanie takiej jasności odpowiedzialne są wszystkie strony projektu. Profesjonalni programiści uznają za część swojej pracy współpracę z udziałowcami i testerami w celu ustalenia, że wszyscy wiedzą, co ma zostać przygotowane.

## Automatyzacja

Testy akceptacyjne zawsze powinny być automatyzowane. W cyklu życia oprogramowania miejsce na testy manualne znajduje się gdzie indziej. *Ten* rodzaj testów nigdy nie powinien być wykonywany ręcznie. Powód jest bardzo prosty: koszta.

Spójrz na rysunek 7.1. Widoczne na nim ręce należą do menedżera kontroli jakości pracującego w wielkiej firmie internetowej. Trzyma w nich dokument będący *spisem treści* jego planu testów *manualnych*. Zarządza on całą armią testerów w kilku zagranicznych centrach, którzy wykonują cały ten plan raz na sześć tygodni. Za każdym razem kosztuje to ponad milion dolarów. Trzyma ten dokument w taki sposób, ponieważ właśnie wraca ze spotkania ze swoim menedżerem, który powiedział mu, że musi ograniczyć budżet o połowę. Kieruje zatem do mnie pytanie: „Która połowę tych testów mam pomijać?“.



Rysunek 7.1. Plan testów manualnych

Nazwanie tego katastrofą byłoby mocnym niedopowiedzeniem. Koszta prowadzenia planu testów manualnych są tak wielkie, że zdecydowali się je poświęcić i przystali na stan, w którym *nie będą wiedzieli, czy połowa ich produktów będzie działać*.

Profesjonalni programiści nie pozwalają na powstanie takiej sytuacji. Koszt zautomatyzowania testów akceptacyjnych jest tak niewielki w porównaniu z kosztami wykonywania testów manualnych, że nie jest ekonomicznie opłacalne tworzenie skryptów, które miałyby być wykonywane przez ludzi. Profesjonalni programiści biorą odpowiedzialność za swoją część pracy, upewniając się, że testy akceptacyjne są wykonywane automatycznie.

Istnieje wiele otwartoźródłowych i komercyjnych narzędzi pozwalających na zautomatyzowanie testów akceptacyjnych. FitNesse, Cucumber, cuke4duke, Robot Framework i Selenium to zaledwie kilka z nich. Wszystkie te narzędzia pozwalają na zdefiniowanie zautomatyzowanych testów w formie czytelnej dla osób niezwiązań z programowaniem. Co więcej, umożliwiającej takim osobom pisanie własnych testów.

## Dodatkowa praca

Uwaga Sama dotycząca nakładu pracy jest całkiem zrozumiała. Pisanie tego rodzaju testów akceptacyjnych *naprawdę* wygląda na ogrom pracy. Jednak z rysunku 7.1 wynika, że tak naprawdę nie jest to praca dodatkowa. Pisanie tych testów jest pracą związaną z tworzeniem specyfikacji systemu. Tworzenie specyfikacji o takim poziomie dokładności jest jedynym sposobem pozwalającym nam, programistom, stwierdzić, co właściwie znaczy „gotowe”. Tak dokładna specyfikacja pozwala udziałowcom projektu upewnić się, że system, za który płacą, naprawdę robi to, co powinien. Poza tym napisanie tak dokładnej specyfikacji jest jedynym sposobem na to, żeby testy można było zautomatyzować. Nie można zatem patrzeć na to jak na dodatkową pracę, ale raczej trzeba widzieć w tym metodę pozwalającą na ogromną oszczędność czasu i pieniędzy. Takie testy zapobiegają implementowaniu niewłaściwego systemu i *powiedzą* Ci dokładnie, kiedy system będzie gotowy.

## Kto i kiedy pisze testy akceptacyjne?

W idealnym świecie udziałowcy i kontrola jakości współpracują ze sobą, pisząc takie testy, a programiści kontrolują je jeszcze dla uzyskania pełnej spójności. W rzeczywistym świecie udziałowcy tylko rzadko mają czas i ochotę, żeby wystarczająco dokładnie poznać system. W związku z tym zwykle przekazują to zadanie analitykom biznesowym, kontroli jakości albo nawet programistom. Jeżeli okazuje się, że to programista musi napisać testy akceptacyjne, to trzeba zadbać o to, by programista piszący testy nie zajmował się jednocześnie implementowaniem testowanej funkcji.

Analitycy biznesowi zwykle tworzą „*optimistyczną*” wersję testów, ponieważ takie testy opisują funkcje mające dużą wartość dla firmy. Dział kontroli jakości pisze zwykle testy „*pesymistyczne*”, obejmujące wszelkie warunki krańcowe. Wynika to z faktu, że zadaniem kontroli jakości jest wymyślanie, co może pójść źle.

Zgodnie z zasadą „*późnej dokładności*” testy akceptacyjne powinny być pisane tak późno, jak to tylko możliwe, najczęściej na kilka dni przed zimplementowaniem danej funkcji. W projektach zwinnych są pisane *po wybraniu* funkcji dla następnej iteracji lub sprintu.

Pierwszych kilka testów akceptacyjnych powinno być gotowych już pierwszego dnia iteracji. Kolejne powinny być pisane w następnych dniach, aż do połowy iteracji, kiedy to wszystkie testy akceptacyjne muszą być przygotowane. Jeżeli testy te nie będą przygotowane do połowy

iteracji, to część programistów będzie musiała wspomóc ich tworzenie. Jeżeli będzie się to zdarzało często, to należy dodać do zespołu więcej analityków biznesowych i testerów.

## Rola programisty

Prace nad implementowaniem funkcji zaczyna się, gdy gotowe będą jej testy akceptacyjne. Programiści uruchamiają testy akceptacyjne tworzonej funkcji, aby się dowiedzieć, w jaki sposób nie zostaną one zaliczone. Następnie pracują nad połączeniem testu akceptacyjnego z systemem, a potem starają się zaliczyć ten test, implementując opisywaną przez niego funkcję.

Paula: „Peter, pomożesz mi z tym?”.

Peter: „No pewnie. W czym problem?”.

Paula: „Mam tutaj test akceptacyjny, którego jak widzisz, nie udało się zaliczyć”.

```
zakładając, że mamy polecenie LogFileDirectoryStartupCommand  
zakładając, że katalog old_inactive_logs nie istnieje  
gdy polecenie zostanie uruchomione  
to katalog old_inactive_logs powinien istnieć  
i powinien być pusty
```

Peter: „No faktycznie, czerwony. Ale nikt nie dopisał do niego scenariuszy. Na szybko napiszę pierwszy”.

```
|scenario| given the command _|cmd|  
|create command|@cmd|
```

Paula: „Operacja createCommand jest już gotowa?”.

Peter: „Oczywiście. Jest w klasie CommandUtilitiesFixture. Napisałem ją w zeszłym tygodniu”.

Paula: „OK, spróbuj uruchomić test”.

Peter: (uruchamia test) „Dobrze, pierwszy wiersz już zielony. Możemy przejść do kolejnego”.

Nie przejmuj się za bardzo scenariuszami i dodatkami. To tylko uzupełnienia, które trzeba dopisać, żeby połączyć test z testowanym systemem.

Wystarczy powiedzieć, że wszystkie narzędzia udostępniają jakąś metodę dopasowywania wzorców do rozpoznawania i analizowania zapisanego testu, a następnie wywoływanie funkcji, które dostarczają dane do testowanego systemu. Nakład pracy jest tutaj naprawdę niewielki, a poszczególnych scenariuszy i dodatków można używać ponownie w innych testach.

Najważniejsze w tym jest to, że przyłączenie testu akceptacyjnego do systemu i zaliczenie poszczególnych testów jest zadaniem programisty.

## **Negocjowanie testów i pasywna agresja**

Autorzy testów są tylko ludźmi i dlatego popełniają błędy. Czasami testy zapisane w określony sposób tracą swój sens w momencie, gdy zacznie się je implementować. Niektóre mogą być nadmiernie skomplikowane. Inne z kolei dziwacznie sformułowane. Mogą zawierać niedorzecze założenia. Albo być z gruntu nieprawidłowe. Takie testy mogą doprowadzać do frustracji programistów, którzy mają je zaliczyć.

Zadaniem każdego zawodowego programisty jest negocjowanie z autorem w celu uzyskania lepszego testu. Z kolei *nigdy* nie wolno przyjmować postawy pasywno-agresywnej i stwierdzać: „Skoro to zostało zapisane w teście, dokładnie to zaimplementuję”.

Pamiętaj, że zadaniem zawodowca jest wspomaganie zespołu w dążeniu do przygotowania jak najlepszego oprogramowania. Oznacza to, że każdy powinien wyszukiwać błędy oraz niedopatrzenia i wspólnie pracować nad ich korygowaniem.

Paula: „Tom, ale to chyba się nie zgadza”.

upewnij się, że operacja wysyłania zakończy się w ciągu 2 sekund

Tom: „Jak dla mnie jest OK. W wymaganiach zapisano, że użytkownicy nie powinni czekać dłużej niż dwie sekundy. Gdzie widzisz problem?”.

Paula: „Problem polega na tym, że takiej gwarancji możemy udzielić tylko w sensie statystycznym”.

Tom: „To brzmi trochę podstępnie. W wymaganiach są zapisane dwie sekundy”.

Paula: „Zgadza się. I możemy tego dotrzymać w 99,5% przypadków”.

Tom: „Ale wymaganie nie zostało zdefiniowane w ten sposób”.

Paula: „Ale tak wygląda rzeczywistość. Nie ma sposobu na to, żeby zagwarantować zakończenie wysyłania w dwie sekundy”.

Tom: „Sam się wścieknie”.

Paula: „No właśnie nie. Już z nim rozmawiałam o tym i nie ma nic przeciwko, pod warunkiem że *normalne* odczucie użytkownika nie przekroczy dwóch sekund”.

Tom: „No dobrze. W takim razie jak mam zapisać ten test? Nie mogę przecież wpisać, że operacja wysyłania *zwykle* kończy się w ciągu dwóch sekund”.

Paula: „Zapisz to statystycznie”.

Tom: „Czyli mam tysiąc razy uruchomić operację wysyłania i sprawdzić, że co najwyżej pięć prób trwało dłużej niż dwie sekundy? To przecież absurd”.

Paula: „Nie, to zajęłoby prawie godzinę. Jest na to lepsza metoda. Co powiesz na to?”.

wykonaj 15 operacji wysyłania i zsumuj ich czasy  
upewnij się, że ocena Z dla 2 sekund wynosi przynajmniej 2,57

Tom: „Co to jest ocena Z?”.

Paula: „To trochę statystyki. Może to będzie lepsze”.

wykonaj 15 operacji wysyłania i zsumuj ich czasy  
upewnij się, że w 99,5% przypadków czas wykonania jest krótszy  
niż 2 sekundy

Tom: „To jest zdecydowanie czytelniejsze, ale czy mogę zaufać obliczeniom  
wykonywanym w tle?”.

Paula: „W wynikach testu będę podawać wszystkie obliczenia pośrednie, tak żeby  
można było wszystko skontrolować w razie wątpliwości”.

Tom: „OK, to mi się podoba”.

## Testy akceptacyjne i testy jednostkowe

Testy akceptacyjne nie są testami *jednostkowymi*. Testy jednostkowe są pisane *przez* programistów *dla* programistów. Są formalnymi dokumentami projektowymi opisującymi najniższą strukturę i zachowanie kodu. Odbiorcami tych testów są inni programiści, nikt inny.

Testy akceptacyjne są pisane *przez* nieprogramistów *dla* nieprogramistów (nawet jeżeli ostatecznie to programista musi je zapisać). Są formalnymi dokumentami opisującymi wymagania, definiującymi, jak system powinien się zachowywać z punktu widzenia użytkownika. Odbiorcami tych testów są zarówno programiści, *jak i* pozostały uczestnicy projektu.

Kuszące może się wydawać wyeliminowanie „dodatkowej pracy” przez założenie, że te dwa rodzaje testów są nadmiarowe. Chociaż testy jednostkowe i akceptacyjne często sprawdzają te same rzeczy, to jednak nie są względem siebie nadmiarowe.

Po pierwsze, mimo że testują te same rzeczy, to jednak robią to za pomocą innych mechanizmów i ścieżek. Testy jednostkowe dobierają się do wnętrzości systemu i wywołują poszczególne metody w wybranych klasach. Testy akceptacyjne korzystają z systemu na znacznie wyższym poziomie: na poziomie API, a nawet interfejsu użytkownika. Oznacza to, że ścieżka wykonania w tych testach jest odmienna.

Jednak najważniejszym powodem, dla którego tych testów nie można traktować jako nadmiarowych, jest to, że ich podstawowym zadaniem *nie jest testowanie*. Fakt, że są one testami, to tylko przypadek. Testy jednostkowe i akceptacyjne są przede wszystkim dokumentami, a dopiero potem testami. Ich podstawowym zadaniem jest formalne udokumentowanie projektu, struktury i zachowania systemu. To, że przy okazji sprawdzają, czy te parametry są zgodne z oczekiwaniami, jest bardzo użyteczne, ale to specyfikacja jest ich prawdziwą naturą.

## Interfejsy użytkownika i inne komplikacje

Wstępne zaprojektowanie interfejsu użytkownika jest bardzo trudne. Oczywiście jest to możliwe, ale tylko rzadko takie próby kończą się dobrze. Powodem takiego stanu rzeczy jest fakt, że estetyka jest subiektywna, a zatem może się zmieniać. Ludzie chcą się *bawić* interfejsami użytkownika. Chcą je dopieszczać i zmieniać. Chcą próbować różnych czcionek, kolorów, układów strony i przepływów. Oznacza to, że GUI ciągle się zmienia.

Sprawia to, że pisanie testów akceptacyjnych dla interfejsów użytkownika jest kłopotliwe. Sztuczka polega na takim zaprojektowaniu GUI, żeby można było go traktować jak API, a nie jak zbiór przycisków, suwaków, tabel i menu. Może to brzmieć dziwacznie, ale na tym polega dobry projekt.

Istnieje zasada projektowania nazywana zasadą pojedynczej odpowiedzialności (SRP — *Single Responsibility Principle*). Zgodnie z nią należy oddzielić od siebie rzeczy zmieniające się z różnych powodów, a grupować te, które zmieniają się z tego samego powodu. Interfejsy użytkownika nie są tu wyjątkiem.

Układ, format i przepływ w interfejsie użytkownika zmieniają się z powodu estetyki i wydajności pracy, ale znajdujące się poniżej możliwości GUI pozostają niezmienne mimo tych wszystkich zmian. Z tego powodu w trakcie pisania testów akceptacyjnych dla interfejsu użytkownika trzeba wykorzystywać te niezmieniające się zbyt często podstawowe abstrakcyjne elementy.

Na przykład na stronie może znajdować się wiele przycisków. Zamiast tworzyć testy klikające poszczególne przyciski na podstawie ich pozycji na stronie, lepiej jest kliknąć je, wykorzystując ich nazwy. A jeszcze lepiej byłoby, gdyby każdy z tych przycisków miał swój unikalny identyfikator, którego mogłyby używać testy. Znacznie łatwiej pisze się test wybierający przycisk o identyfikatorze `ok_button` niż wybierający przycisk w trzecim rzędzie i czwartej kolumnie siatki.

## Testowanie przez właściwy interfejs

Najlepszym rozwiązaniem jest jednak tworzenie testów wywołujących funkcje testowanego systemu poprzez właściwe API, a nie przez operacje na GUI. Testowane API musi być jednak tym samym, z którego korzysta GUI. To nic nowego. Ekspertci od projektowania już od dziesięcioleci nakazują nam oddzielać interfejs użytkownika od reguł biznesowych.

Testowanie poprzez interfejs użytkownika jest zawsze problematyczne, chyba że testowany jest *sam* interfejs. Powodem jest to, że GUI może się zmieniać, przez co takie testy są bardzo nietrwałe. Jeżeli każda zmiana w interfejsie użytkownika psuje tysiąc testów, to trzeba te testy wyrzucić albo przestać wprowadzać zmiany do GUI. Żadna z tych opcji nie jest dobra. Dlatego testy reguł biznesowych powinny korzystać z API znajdującego się tuż pod interfejsem użytkownika.

Niektóre testy akceptacyjne opisują zachowanie samego GUI. Takie testy *muszą* korzystać z interfejsu użytkownika. Trzeba jednak pamiętać, że nie sprawdzają one reguł biznesowych, a zatem nie wymagają podłączenia ich do GUI. Z tego powodu podczas testowania GUI dobrze jest odłączyć interfejs użytkownika od reguł biznesowych i zastąpić te ostatnie atrapami.

Testy interfejsu użytkownika należy ograniczyć do minimum. Takie testy są nietrwałe, ponieważ GUI szybko się zmienia. Im więcej testów GUI przygotujesz, tym bardziej prawdopodobne jest, że z czasem zostaną one porzucone.

## Ciągła integracja

Upewnij się, że wszystkie Twoje testy jednostkowe i akceptacyjne są uruchamiane kilka razy dziennie w ramach *systemu ciągłej integracji* (ang. *continuous integration system*). System ten powinien być uruchamiany przez system kontroli wersji kodu źródłowego. Za każdym razem, gdy ktoś wprowadza zmiany do modułu, system CI powinien uruchamiać komplikację, a następnie wykonywać wszystkie testy w systemie. Wyniki takiego przebiegu powinny być wysyłane do wszystkich członków zespołu.

## Zatrzymać prasę

Bardzo ważne jest, żeby testy w systemie CI były uruchamiane bardzo często. Wszystkie te testy powinny zostać zaliczone. Jeżeli to się nie uda, to cały zespół powinien wstrzymać prace i skupić się na tym, żeby poprawić feralny test. Niepełną komplikację i niezaliczone testy należy traktować jak sytuację alarmową i zdarzenie „zatrzymujące prasę”.

Współpracowałem już z zespołami, które nie traktowały poważnie nieudanych komplikacji systemu. Wszyscy byli „zbyt zajęci”, żeby poprawiać niezaliczone testy, dlatego spychali je na bok, obiecując, że poprawią je później. W jednym przypadku zespół usunął z procesu komplikacji niedziałające testy, ponieważ ich czerwone statusy były w tym momencie nie na rękę. Później, po przekazaniu systemu klientowi, przypomnieli sobie, że zapomnieli włączyć te testy do procesu. Dowiedzieli się o tym, ponieważ wściekły klient dzwonił i ciągle zgłaszał kolejne błędy.

## Wnioski

Porozumiewanie się w sprawie szczegółów jest bardzo trudne. Specjalnego wyrazu nabiera to w przypadku, gdy programiści i udziałowcy projektu muszą porozumieć się w sprawie szczegółów tworzonej aplikacji. Zbyt łatwo każda ze stron może machnąć ręką i założyć, że druga strona ją zrozumie. Często zdarza się też tak, że obie strony zgadzają się na porozumienie, ale rozchodzą się, ponieważ mają zupełnie różne wyobrażenia.

Jedynym znanym mi sposobem na skuteczne wyeliminowanie błędów komunikacji między programistami i udziałowcami jest pisanie zautomatyzowanych testów akceptacyjnych. Ich wykonywanie jest działaniem czysto formalnym. Są całkowicie jednoznaczne i nie mogą rozsynchonizować się z aplikacją. Są zatem doskonałym dokumentem opisującym wymagania.

---

# STRATEGIE TESTOWANIA

---



Profesjonalni programiści testują swój kod. Jednak testowanie nie jest tylko kwestią napisania kilku testów jednostkowych albo testów akceptacyjnych. Pisanie tych testów z całą pewnością pomaga, ale nie jest wystarczające. Każdy profesjonalny zespół programistów musi mieć dobrą strategię testowania.

W 1989 roku pracowałem w firmie Rational nad pierwszym wydaniem systemu Rose. Mniej więcej co miesiąc szef działu kontroli jakości ogłaszał dzień „polowania na błędy”. Każdy członek zespołu od programistów do menedżerów, sekretarki i administratorów baz danych siedział i próbował wywołać jakiś błąd w Rose. Za wykrycie określonych rodzajów błędów

były przyznawane różne nagrody. Osoba, która wykryła błąd powodującyagle zamknięcie systemu, wygrywała obiad dla dwojga. Z kolei osoba, która znalazła najwięcej błędów, wygrywała weekend w Monterrey.

## **Kontrola jakości nie powinna nic znaleźć**

Mówiłem już o tym wcześniej, ale powiem jeszcze raz. Mimo że Twoja firma ma osobny zespół kontroli jakości, którego zadaniem jest testowanie oprogramowania, to celem zespołu programistów powinno być takie przygotowanie tego oprogramowania, żeby kontrola jakości nie znalazła nic.

Oczywiście mało prawdopodobne jest, żeby ten cel udawało się osiągnąć za każdym razem. W końcu skoro mamy grupę inteligentnych ludzi, którzy są zdeterminowani, żeby znaleźć wszelkie niedociągnięcia i braki w naszym produkcie, to z pewnością uda im się coś znaleźć. Mimo to za każdym razem, gdy kontrola jakości znajdzie błąd w systemie, zespół programistów powinien reagować przerażeniem. Powinni zapytać samych siebie, jak to było możliwe, i przedsięwziąć kroki uniemożliwiające powstanie takich błędów w przyszłości.

## **Kontrola jakości jest częścią zespołu**

Z poprzedniego podrozdziału może wynikać, że działy rozwojowy i kontroli jakości powinny ze sobą konkurować, a między nimi powinny panować wrogie stosunki. Nie o to jednak chodzi. Oba działy powinny raczej współdziałać, żeby zapewnić wysoką jakość systemu. Najlepszą rolą dla pracowników działu kontroli jakości jest działanie definiujące i uszczegóławiające.

### **Kontrola jakości i definicje**

Zadaniem działu kontroli jakości powinna być współpraca z całą firmą w celu przygotowania zautomatyzowanych testów akceptacyjnych, które staną się prawdziwą specyfikacją systemu i dokumentem opisującym wymagania wobec niego. W każdej kolejnej iteracji dział powinien gromadzić wymagania od klienta i przekładać je na testy opisujące programistom pożądane zachowanie systemu (rozdział 7., „Testy akceptacyjne”). Ogólnie analitycy biznesowi tworzą testy optymistyczne, natomiast dział kontroli jakości tworzy pesymistyczne testy obejmujące wszystkie warunki krańcowe.

### **Kontrola jakości i uszczegółowienia**

Innym zadaniem działu kontroli jakości jest wykorzystanie testów badawczych<sup>1</sup> do zdefiniowania prawdziwego zachowania działającego systemu i zgłoszenia go programistom

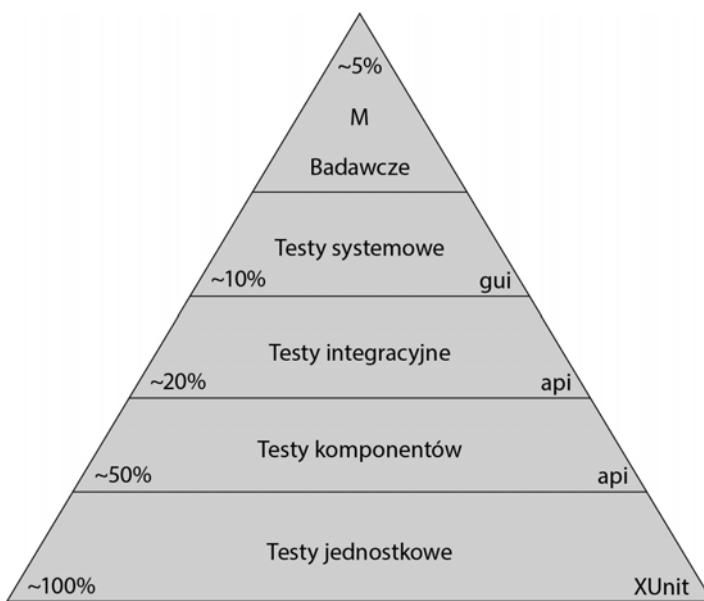
---

<sup>1</sup> [http://www.satisfice.com/articles/what\\_is\\_et.shtml](http://www.satisfice.com/articles/what_is_et.shtml).

i analitykom biznesowym. W tym przypadku dział kontroli jakości *nie* interpretuje wymagań wobec systemu, ale raczej dokumentuje jego rzeczywiste zachowanie.

## Piramida automatyzacji testów

Profesjonalni programiści do tworzenia testów jednostkowych stosują technikę TDD. Zespoły profesjonalnych programistów wykorzystują testy akceptacyjne do definiowania zachowań swoich systemów oraz do ciągłej integracji (rozdział 7.) zapobiegającej błędom regresyjnym. Ale te testy to tylko część całości. Oczywiście bardzo dobrze jest mieć pełen zestaw testów jednostkowych i akceptacyjnych, ale oprócz tego potrzebne są jeszcze testy wyższego poziomu, dające nam pewność, że dział kontroli jakości nie znajdzie już nic. Na rysunku 8.1 przedstawiłem piramidę automatyzacji testów<sup>2</sup>, czyli graficzną reprezentację rodzajów testów wymaganych w profesjonalnych zespołach tworzących oprogramowanie.



Rysunek 8.1. Piramida automatyzacji testów

### Testy jednostkowe

Na samym dole piramidy są testy jednostkowe. Testy te są pisane przez programistów dla programistów, w języku programowania, w którym tworzony jest system. Zadaniem tych testów jest zdefiniowanie systemu na jego najniższym poziomie. Programiści piszą je przed napisaniem kodu produktywnego i traktują jak sposób na zdefiniowanie tego, co mają

<sup>2</sup> [COHN09], s. 311 – 312.

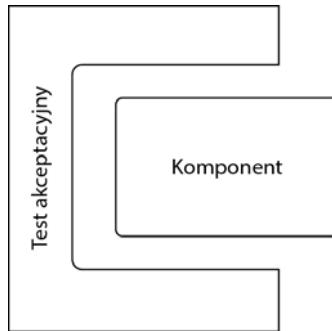
zamiar napisać. Testy te są wykonywane w ramach ciągłej integracji i zapewniają tym samym, że zamiary programisty zostały zrealizowane.

Teoretycznie testy jednostkowe powinny zapewniać niemal stu procentowe pokrycie kodu. W rzeczywistości pokrycie to powinno obejmować ponad 90% kodu. I powinno to być *prawdziwe* pokrycie, a nie oszukane testy, które tylko wykonują kod, ale nie sprawdzają jego wyników.

## Testy komponentów

Tutaj znajdują się niektóre z testów akceptacyjnych, o których wspomniałem w poprzednim rozdziale. Pisane są one tak, żeby testować poszczególne komponenty systemu. Komponenty te obejmują także reguły biznesowe, dlatego ich testy można traktować jak testy akceptacyjne tych reguł.

Jak można zobaczyć na rysunku 8.2, test komponentu zawiąza się naokoło tego, co testuje. Przekazuje do komponentu dane wejściowe i odbiera od niego dane wyjściowe. Sprawdza następnie, czy wyjście pasuje do wejścia. Wszelkie pozostałe komponenty systemu są w tym momencie odłączone z wykorzystaniem odpowiednich technik podmiany i imitowania komponentów.



---

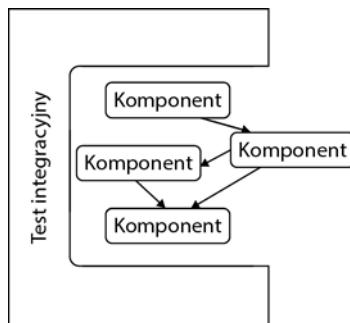
**Rysunek 8.2.** Akceptacyjny test komponentu

Testy komponentów są pisane przez dział kontroli jakości i analityków biznesowych przy współudziale działu rozwojowego. Powstają w środowiskach do tworzenia testów komponentów, takich jak FitNesse, JBehave albo Cucumber. (Elementy GUI są testowane za pomocą przeznaczonych do tego środowisk, takich jak Selenium albo Watir). Celem jest, żeby każdy był w stanie odczytać i zinterpretować takie testy, a może nawet je tworzyć.

Testy komponentów powinny pokrywać mniej więcej połowę systemu. Tworzone są zwykle w celu sprawdzenia optymistycznych wariantów i oczywistych przypadków alternatywnych i krańcowych. Ogromna większość przypadków pesymistycznych pokrywana jest już w testach jednostkowych, a przez to tracą one znaczenie w testach komponentów.

## Testy integracyjne

Te testy mają znaczenie tylko w większych systemach składających się z wielu komponentów. Jak pokazałem na rysunku 8.3, łączą kilka komponentów w grupę i testują ich wzajemną komunikację. Pozostałe komponenty systemu pozostają odłączone z wykorzystaniem typowych technik podmiany i imitowania.



Rysunek 8.3. Test integracyjny

Testy integracyjne można nazwać testami *choreografii*. Nie mają testować reguł biznesowych, ale raczej to, czy poszczególne komponenty właściwie ze sobą współpracują. Są to testy *hydrauliczne*, sprawdzające, czy poszczególne komponenty są ze sobą odpowiednio połączone i mogą się ze sobą komunikować.

Testy integracyjne zazwyczaj pisane są przez architektów systemu albo przez jego głównych projektantów. Zapewniają, że struktura architektoniczna systemu pozostaje nienaruszona. To właśnie na tym poziomie można przeprowadzać testy wydajności i przepustowości.

Testy integracyjne zwykle pisane są w tym samym języku i środowisku, w którym powstają testy komponentów. Nie są jednak wykonywane w ramach ciągłej integracji, ponieważ często czasy ich wykonania są bardzo długie. Zamiast tego są uruchamiane cyklicznie (co noc albo raz na tydzień), zależnie od tego, czy ich autorzy uznają to za konieczne.

## Testy systemowe

Testy systemowe są testami zautomatyzowanymi uruchamianymi na całym, połączonym systemie. Są ostatecznymi testami integracyjnymi. Nie testują bezpośrednio reguł biznesowych, ale sprawdzają, czy cały system został prawidłowo połączony, a poszczególne jego części działają zgodnie z założeniami. Wśród tych testów powinny znaleźć się też testy przepustowości i wydajności.

Ten rodzaj testów jest pisany przez architektów systemu. Zwykle są tworzone w tym samym języku i środowisku, w którym powstawały testy integracyjne interfejsu użytkownika. Ze względu na czas trwania wykonywane są raczej rzadko, ale im częściej się to dzieje, tym lepiej.

Testy systemowe pokrywają może 10% systemu. Wynika to z faktu, że ich zadaniem nie jest zapewnienie właściwego zachowania systemu, ale jego właściwej *konstrukcji*. Pożądane zachowania bazowego kodu i komponentów zostały już dokładnie przetestowane przez testy z niższych poziomów piramidy.

## **Manualne testy badawcze**

To w tym momencie ludzie kładą ręce na klawiatury, a oczy kierują na ekrany. Ten rodzaj testów nie jest zautomatyzowany, *nie ma w nim żadnych skryptów*. Ich celem jest badanie reakcji systemu na nieoczekiwane zachowania użytkownika i potwierdzanie zachowań oczekiwanych. Na koniec potrzebne są ludzki umysł i kreatywność pozwalające na badanie i poznawanie systemu. Przygotowywanie jakiegokolwiek planu tego rodzaju testów jest sprzeczne z ich celami.

W niektórych zespołach są specjalne osoby wykonujące te zadania. Inne zespoły po prostu ogłaszały jeden dzień lub dwa „polowania na błędy”, w czasie których jak najwięcej osób, w tym i menedżerowie, sekretarki, programiści, testerzy i twórcy dokumentacji, bawi się systemem i stara na wszelkie sposoby sprowadzić go na kolana.

Tutaj celem nie jest pokrycie kodu. Nie będziemy sprawdzać poprawności każdej reguły biznesowej i każdej możliwej ścieżki wykonania. Chodzi bardziej o sprawdzenie, czy system zachowuje się poprawnie przy współpracy z człowiekiem, i kreatywne wyszukanie jak największej liczby „osobliwości”.

## **Wnioski**

TDD jest bardzo potężnym narzędziem, a testy akceptacyjne są cennym środkiem wyrażania i wymuszania wymagań wobec systemu. Są one jednak tylko częścią ogólnej strategii testowania. Chcąc jak najbardziej zbliżyć się do celu, w którym „kontrola jakości nie znajdzie nic”, zespoły programistów muszą współpracować z działem kontroli jakości i wspólnie przygotowywać hierarchię testów jednostkowych, testów komponentów, testów integracyjnych, testów systemu i testów badawczych. Testy te powinny być uruchamiane tak często, jak to możliwe, aby zapewnić jak największą ilość danych i utrzymać nieprzerwaną czystość systemu.

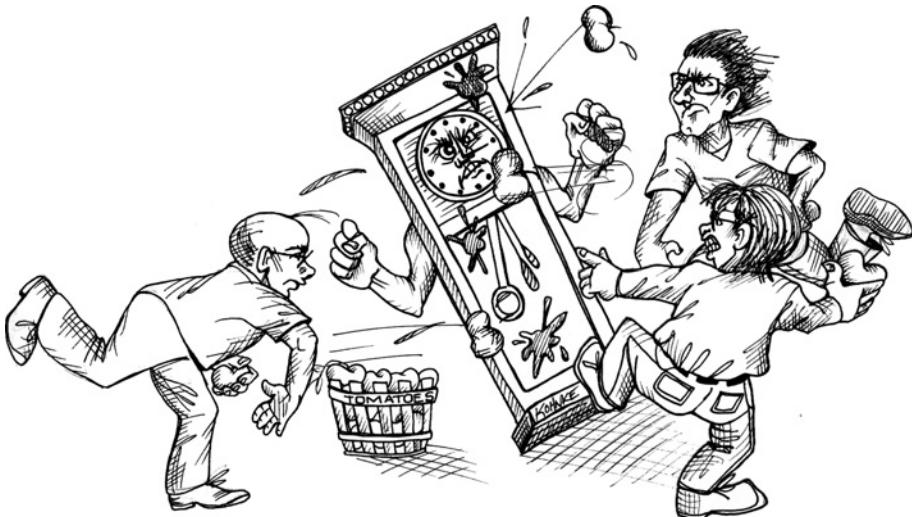
## **Bibliografia**

[COHN09]: Mike Cohn, *Succeeding with Agile. Software Development Using Scrum*, Boston, MA: Addison-Wesley, 2009.

---

# ZARZĄDZANIE CZASEM

---



Osiem godzin to zadziwiająco krótki czas. To tylko 480 minut albo 28 800 sekund. Jako profesjonalista oczekujesz zapewne, że jak najskuteczniej wykorzystasz tę niewielką pulę sekund. Jaką strategię możesz zastosować, żeby na pewno nie marnować cennego czasu? Jak można skutecznie zarządzać swoim czasem?

W 1986 roku mieszkałem w Little Sandhurst, w hrabstwie Surrey w Anglii. Zarządałem wtedy 15-osobowym działem programistów firmy Teradyne w Bracknell. Moje chaotyczne dni wypełniały rozmowy telefoniczne, improwizowane spotkania i najróżniejsze przerwy. Chcąc wykonać jakąkolwiek pracę, przyjąłem kilka dość drastycznych zasad zarządzania czasem.

- Każdego ranka budziłem się o 5 rano i jechałem rowerem do biura w Bracknell, gdzie docierałem o 6 rano. To dawało mi 2,5 godziny ciszy, zanim zaczynał się codzienny chaos.
- Po dotarciu do pracy rysowałem na tablicy plan dnia. Dzieliłem czas na 15-minutowe części i wpisywałem do nich działania, którymi będę się zajmował w danym wycinku czasu.
- Całkowicie wypełniałem pierwsze 3 godziny planu. Od godziny 9 zaczynałem pomijać w każdej godzinie po jednym 15-minutowym bloku. W ten sposób mogłem łatwo wepchnąć nieprzewidziane przerwy w takie otwarte przestrzenie, a potem pracować dalej.
- Czas po obiedzie pozostawiałem niezaplanowany, ponieważ wiedziałem, że do tego czasu rozpęta się prawdziwe piekło i przez pozostałą część dnia będę musiał działać w trybie reaktywnym. W czasie tych rzadkich popołudniowych chwil, kiedy chaos trochę przycichał, pracowałem po prostu nad najważniejszymi rzeczami.

Ten plan nie zawsze się sprawdzał. Nie zawsze udawało mi się wstać o 5 rano, a czasami chaos pochłaniał moje staranne strategie i zajmował mnie przez cały dzień. Ale przez większość dni byłem w stanie utrzymać się na powierzchni wody.

## **Spotkania**

Koszt spotkania to mniej więcej 200 dolarów za godzinę, za uczestnika. W tę cenę wliczam zarówno pensję i dodatki, jak i koszta pomieszczenia itp. Następnym razem biorąc udział w spotkaniu, policz jego koszta. Na pewno będą zadziwiające.

O spotkaniach można powiedzieć na pewno, że:

1. Spotkania są niezbędne.
2. Spotkania są strasznym marnotrawstwem czasu.

Czasami oba punkty równie dobrze opisują jedno spotkanie. Część uczestników może uznać je za wartościowe, podczas gdy pozostali uznają je za niepotrzebne.

Profesjonalisci są świadomi wysokich kosztów spotkań. Wiedzą też, że ich własny czas jest cenny — w końcu muszą napisać kod i zdążyć zrobić to w terminie. Z tego powodu aktywnie oponują przed udziałem w spotkaniach, z których nie da się czerpać bezpośrednich i znaczących zysków.

## **Odmawianie**

Nie musisz brać udziału w każdym spotkaniu, na które dostaniesz zaproszenie. Co więcej, udział w zbyt wielkiej liczbie spotkań można uznać za działanie nieprofesjonalne. Musisz mądrze korzystać ze swojego czasu, a zatem ostrożnie wybieraj spotkania i grzecznie odmawiaj uczestniczenia w tych, które uznasz za nieprzydatne.

---

Osoba zapraszająca Cię na spotkanie nie zajmuje się zarządzaniem Twoim czasem. Tylko *Ty* możesz to robić. A zatem po otrzymaniu zaproszenia na spotkanie nie przyjmuj go, chyba że Twoja obecność na nim jest absolutnie niezbędna i przydatna dla prac, które aktualnie wykonujesz.

Czasami spotkania będą dotyczyć czegoś, co Cię interesuje, ale nie jest Ci całkowicie niezbędne. Wtedy musisz samodzielnie zdecydować, czy możesz pozwolić sobie na poświęcenie tego czasu. Zachowaj jednak ostrożność — takich spotkań może być dość dużo, żeby wypełnić całe Twoje dni.

Czasami spotkania będą dotyczyć czegoś, w czym możesz pomóc, ale niemającego żadnego znaczenia dla aktualnie wykonywanych przez Ciebie prac. Musisz wtedy zdecydować, czy strata poniesiona przez Twój projekt warta jest zysków w innym projekcie. To może brzmieć bardzo cynicznie, ale Twoją podstawową odpowiedzialnością jest Twój własny projekt. Mimo to czasem dobrze jest, gdy jeden zespół wspomaga inny, dlatego warto omówić swój udział w takim spotkaniu z własnym zespołem i menedżerem.

Czasami Twojej obecności na spotkaniu żądać będzie ktoś z kierownictwa, na przykład starszy projektant albo menedżer innego projektu. Wówczas musisz zdecydować, czy żądanie tych osób jest ważniejsze od Twojego własnego planu prac. Wtedy również pomocne może okazać się zapytanie własnego zespołu i menedżera.

Jednym z najważniejszych zadań Twojego kierownika jest *ograniczanie* liczby Twoich spotkań. Dobry menedżer będzie bardzo chętnie bronił Twojej decyzji o odmowie udziału w spotkaniu, ponieważ powinien on na równi z Tobą przejmować się wykorzystaniem Twojego czasu.

## **Wychodzenie ze spotkań**

Spotkania nie zawsze idą zgodnie z planem. Czasami bierzesz udział w spotkaniu, które odbyłyby się bez Ciebie, gdyby tylko wcześniej dostarczono Ci odpowiednie informacje. Niektóre dodawane są nowe tematy albo ktoś zaczyna zdominowywać rozmowę swoim „konikiem”. Przez lata wypracowałem sobie prostą zasadę: gdy spotkanie staje się nudne, to po prostu wychodzę.

Przypominam, że Twoim zadaniem jest odpowiednie zarządzanie swoim czasem. Jeżeli okazuje się, że bierzesz udział w spotkaniu, które zupełnie nie przyczynia się do dobrego wykorzystania Twojego czasu, znajdź sposób, żeby grzecznie się z niego wycofać.

Oczywiście nie należy wybiegać z pokoju konferencyjnego, krzycząc: „Ale nudy!”. To byłoby niegrzeczne. Możesz jednak we właściwym momencie zapytać, czy Twoja obecność jest konieczna. Należy wtedy wyjaśnić, że nie masz aż tyle czasu do dyspozycji, i poprosić o przyspieszenie dyskusji albo zmianę planu spotkania.

Trzeba sobie uświadomić, że dalszy udział w takim spotkaniu będzie dla Ciebie stratą czasu i nie jesteś w stanie wnieść nic istotnego do dyskusji. Twoim zadaniem jest mądre gospodarowanie czasem i pieniędzmi pracodawcy, dlatego wybranie właściwego momentu i wyjście ze spotkania jest zachowaniem godnym profesjonalisty.

## **Przygotuj plan i cel spotkania**

Powodem, dla którego jesteśmy gotowi ponosić koszta spotkań, jest to, że czasami *konieczne* jest zebranie w jednym pokoju wszystkich uczestników, aby osiągnąć określony cel. Chcąc rozsądnie wykorzystać czas uczestników spotkania, powinniśmy przygotować jego plan z rozpisanym czasem dyskusji na każdy z tematów i określić cel.

Jeżeli poproszą Cię o udział w spotkaniu, upewnij się, że znasz wszystkie dyskutowane na nim tematy, ilość czasu przeznaczoną na każdy z nich i cele, jakie chce się osiągnąć. Jeżeli nie uzyskasz na te pytania jasnych odpowiedzi, to grzecznie odmów udziału w spotkaniu.

Jeżeli po przystąpieniu do spotkania stwierdzisz, że pierwotny plan został całkiem zmieniony albo porzucony, poproś o zarzucenie nowego tematu spotkania i postępowanie zgodnie z pierwotnym planem. Jeżeli tak się nie stanie, o ile to możliwe, postaraj się grzecznie wycofać.

## **Spotkania na stojąco**

Takie spotkania należą do kanonu metodologii zwinnych. Nazwa wywodzi się z tego, że od uczestników spotkania oczekuje się stania przez cały jego czas. Po kolei każdy uczestnik odpowiada na trzy pytania:

1. Co dzisiaj zrobiłem?
2. Co mam zamiar zrobić jutro?
3. Co mi przeszkadza?

To wszystko. Odpowiedź na każde z pytań powinna zajmować *nie więcej niż* 20 sekund, dlatego na każdego uczestnika spotkania przypadać będzie najwyżej minuta. Nawet w grupie dziesięciu osób spotkanie powinno się skończyć przed upływem 10 minut.

## **Spotkania planujące iteracje**

W kanonie metodologii zwinnych są to spotkania, które najtrudniej zorganizować prawidłowo. Jeżeli ich prowadzenie będzie złe, to zajmą o wiele za dużo czasu. Właściwe ich prowadzenie wymaga umiejętności, które na pewno warto sobie przyswoić.

W ramach spotkania planującego iteracje powinno się wybrać pozycje backlogu, które zostaną zrealizowane w kolejnej iteracji projektu. Dla kandydujących pozycji powinny być

już gotowe szacunki pracochności. Podobnie powinna być gotowa ocena ich biznesowej wartości. W naprawdę dobrych organizacjach napisane już są (a przynajmniej naszkicowane) testy komponentów i akceptacyjne.

Spotkanie powinno szybko posuwać się naprzód, a każdy kandydat wybrany z backlogu powinien być tylko pokrótkę omówiony, a następnie zatwierdzony lub odrzucony. Nad poszczególnymi pozycjami nie należy dyskutować dłużej niż 5 do 10 minut. Jeżeli konieczna jest dłuższa dyskusja, to należy ją przenieść na inny termin i wybrać do niej tylko część zespołu.

Moja ogólna zasada mówi, że spotkanie planujące nie powinno trwać dłużej niż 5% ogólnego czasu iteracji. Oznacza to, że w przypadku tygodniowej iteracji (40 godzin) spotkanie powinno zakończyć się po dwóch godzinach.

## **Retrospektywa iteracji i demonstracja produktu**

Te spotkania mają miejsce na zakończenie każdej iteracji. Członkowie zespołu rozmawiają o tym, co poszło prawidłowo, a co źle. Udziałowcy mogą obejrzeć demonstrację nowych, działających funkcji. Z tymi spotkaniami można bardzo mocno przesadzić, przez co pochłaniają ogromne ilości czasu. Zaplanuj je zatem na 45 minut przed końcem pracy ostatniego dnia iteracji. Przeznacz nie więcej niż 20 minut na retrospektywę i 25 minut na demonstrację funkcji. Pamiętaj, że minął zaledwie tydzień lub dwa, dlatego nie powinno być zbyt wielu tematów do omawiania.

## **Kłótnie i nieporozumienia**

Kent Beck powiedział mi kiedyś coś bardzo ważnego: „Każda kłótnia, której nie da się załagodzić w 5 minut, nie zostanie załagodzona w ramach dalszej kłótni”. Powodem takiego przeciągania sprawy jest to, że żadna ze stron nie może znaleźć konkretnego dowodu potwierdzającego jej tezy. Takie kłótnie są najzwyczajniej religijne, w przeciwieństwie do kłótni bazujących na faktach.

Techniczne rozbieżności potrafią wybuchnąć aż po samą stratosferę. Każda ze stron ma pewne umocowania swoich poglądów, ale rzadko dysponuje jakimkolwiek danymi. Bez danych w każdej kłótni, w której nie uda się zawrzeć porozumienia w ciągu kilku minut (pomiędzy 5 a 30), nie ma najmniejszych szans na porozumienie. Jedynym rozwiązaniem jest dostarczenie niezbędnych danych.

Są ludzie, którzy próbują wygrać taką kłótnię za pomocą siły charakteru. Mogą wtedy krzyczeć, działać natarczywie albo protekcyjnie. To nie ma żadnego znaczenia. Siła woli nie usuwa niezgody na długo. Tylko dane mogą tego dokonać.

Niektórzy mogą zachowywać się pasywno-agresywnie. Zgadzają się na zakończenie sporu, a potem sabotują wyniki, odmawiając swojego zaangażowania w rozwiązanie. Tacy ludzie

mówią sobie: „Przecież tego chcieli, a zatem dostaną dokładnie to, o co prosili”. To chyba najgorszy z możliwych przykład nieprofesjonalnego zachowania. Nigdy nie należy tak postępować. Jeżeli się na coś zgadzasz, to musisz się w to zaangażować.

Jak zdobyć dane niezbędne do załagodzenia sporu? Czasami można wykonać kilka eksperymentów albo symulacji. Niektóre jednak alternatywą będzie rzucenie monetą i wybranie jednego z dwóch możliwych rozwiązań.

Jeżeli wybrane rozwiązanie zadziała, to znaczy, że było właściwe. Jeżeli wpadniesz w tarapaty, to zawsze możesz się wycofać i spróbować drugiego rozwiązania. Dobrze jest też ustalić czas i zestaw kryteriów ułatwiających stwierdzenie, kiedy należy porzucić wybrane rozwiązanie.

Wystrzegaj się spotkań, które są jedynie sposobem na upublicznienie sporu i zebranie poparcia dla jednej lub drugiej strony. I unikaj tych spotkań, na których obecna jest tylko jedna ze stron sporu.

Jeżeli spór musi być załagodzony, to poproś obie strony o zaprezentowanie zespołowi swoich argumentów w ciągu maksymalnie 5 minut. Następnie cały zespół powinien zagłosować. Tego rodzaju spotkanie nie potrwa nawet 15 minut.

## **Manna skupienia**

Wybacz, jeżeli ten, podrozdziały będzie miał posmak metafizyki New Age albo Dungeons & Dragons. Po prostu w ten sposób wyobrażam sobie całe to zagadnienie.

Programowanie jest ćwiczeniem intelektualnym, które wymaga długich okresów koncentracji i skupienia. Skupienie jest niestety bardzo rzadkim dobrem, podobnie jak manna<sup>1</sup>. Po wyczerpaniu swojej manny skupienia musisz ją uzupełnić, wykonując przez przynajmniej godzinę czynności niewymagające koncentracji.

Nie wiem dokładnie, czym jest ta manna skupienia, ale mam wrażenie, że jest to wręcz fizyczna substancja (albo może jej brak), która wpływa na naszą uwagę i czujność. Czymkolwiek jest, na pewno czujesz, gdy jest z Tobą, czujesz również, gdy jej brakuje. Profesjonalni programiści nauczyli się tak zarządzać swoim czasem, żeby jak najlepiej wykorzystać swoją mannę skupienia. Piszą kod, gdy masz sporo manny skupienia, a mniej produktywne czynności wykonuj wtedy, gdy jej brakuje.

---

<sup>1</sup> Manna jest zasobem często wykorzystywany w grach fantasy i przygodowych, takich jak Dungeons & Dragons. Każdy gracz ma pewien zasób manny, czyli magicznej substancji, która wyczerpuje się w czasie rzucania czarów. Im potężniejsze jest zaklęcie, tym więcej manny zużywa. Niestety, manna odradza się bardzo powoli, w pewnym dziennym rytmie. Oznacza to, że łatwo można wyczerpać swoje zasoby w zaledwie kilku sesjach rzucania zaklęć.

---

Manna skupienia jest też zasobem, którego powoli ubywa. Jeżeli nie korzystasz z niej wtedy, gdy jest dostępna, to najprawdopodobniej ją stracisz. To między innymi dlatego spotkania mogą mieć tak niszczące działanie. Jeżeli całą swoją manną skupienia roztrwoniś na spotkaniach, to nie zostanie Ci nic w czasie, gdy zechcesz pisać kod.

Zmartwienia i rozproszona uwaga również zużywają Twoją manną skupienia. Wczorajsza kłótnia z żoną lub mężem, zadrapanie, które udało Ci się zrobić dziś rano na zderzaku, albo niezapłacony rachunek — to wszystko bardzo szybko wysysa z Ciebie manną skupienia.

## **Sen**

Nie jestem w stanie wyrazić tego dość stanowczo. Najwięcej manny skupienia mam po dobrze przespanej nocy. Siedem godzin snu często dostarcza mi manny skupienia na pełne osiem godzin pracy. Profesjonalni programiści odpowiednio planują swój sen tak, żeby najwięcej manny skupienia mieć w czasie, gdy rano przychodzą do pracy.

## **Kofeina**

Bez wątpienia niektórzy z nas umieją lepiej wykorzystać swoje zasoby manny skupienia, wlewając w siebie niewielkie ilości kawy. Ale tu trzeba postępować ostrożnie. Kofeina sprawia, że nasze skupienie staje się „roztrzęsione”, a większe jej ilości mogą skierować je na bardzo dziwne tory. Naprawdę solidna dawka kofeiny może sprawić, że stracisz cały dzień, koncentrując się na zupełnie niepotrzebnych rzeczach.

Konsumpcja i tolerancja kofeiny to sprawa bardzo osobista. Sam lubię wypić kubek mocnej kawy rano i dietetyczną kolę do obiadu. Czasami podwajam sobie dawkę, ale bardzo rzadko przekraczam tę granicę.

## **Ładowanie akumulatorów**

Mannę skupienia można częściowo odzyskać, robiąc coś niewymagającego koncentracji. Długi spacer, rozmowa z przyjacielem albo spoglądanie przez okno mogą podnieść poziom Twojej manny skupienia.

Niektórzy próbują medytować. Inni pozwalają sobie na krótką drzemkę. Jeszcze inni słuchają jakiegoś podcastu albo czytają gazetę.

Wiem też, że po wyczerpaniu manny nie można zmuszać się do dalszego skupienia. Nadal możesz pisać kod, ale niemal na pewno konieczne będzie poprawianie go następnego dnia. No chyba że pogodzisz się z istnieniem tego paskudztwa przez kolejne tygodnie i miesiące. Lepiej jednak poświęcić 30 albo i 60 minut na coś niewymagającego koncentracji.

## **Skupienie mięśni**

Coś niesamowitego jest w wykonywaniu ćwiczeń fizycznych, takich jak sztuki walki, tai-chi albo joga. Mimo że ćwiczenia te wymagają od Ciebie dużego skupienia, to jednak jest to inny rodzaj skupienia niż to niezbędne podczas tworzenia kodu. Nie jest to skupienie intelektu, ale mięśni. I w jakiś przedziwny sposób skupienie mięśni pozwala na odzyskanie możliwości intelektu. To jednak coś więcej niż proste naładowanie akumulatorów. Przekonałem się już, że regularne ćwiczenia fizyczne zwiększą moje możliwości skupienia umysłowego.

Moją ulubioną formą skupienia fizycznego jest jazda na rowerze. Jadę przez godzinę lub dwie, pokonując w tym czasie jakieś 30 lub 40 kilometrów. Jeżdżę zwykle po ścieżce biegającej wzdłuż rzeki Plaines, dlatego nie muszę się przejmować samochodami.

W czasie jazdy słucham podcastów na temat astronomii lub polityki. Czasami wybieram po prostu swoją ulubioną muzykę, a innym razem zdejmuję słuchawki i wsłuchuję się w odgłosy natury.

Są też ludzie, którzy lubią prace ręczne. Na przykład bawią się stolarką, budowaniem modeli albo pielęgnacją ogrodu. Niezależnie od rodzaju takiej aktywności jest coś w pracach wykorzystujących głównie mięśnie, że powiększą one możliwości pracy umysłowej.

## **Wejście kontra wyjście**

Inną rzeczą, która moim zdaniem bardzo wpływa na skupienie, jest dopasowanie mojego wyjścia do właściwego wejścia. Pisanie oprogramowania jest zajęciem *kreatywnym*. Uważam, że najbardziej kreatywny jestem wtedy, gdy doświadczam kreatywności innych ludzi. Czytam zatem sporo literatury science fiction. Kreatywność tych autorów w jakiś sposób stymuluje moje własne kreatywne ścieżki tworzące oprogramowanie.

## **Paczkowanie czasu i pomidory**

Jedną z bardzo skutecznych technik zarządzania czasem i skupieniem jest doskonale znana technika pomidora<sup>2</sup> (ang. *pomodoro technique*). Sam pomysł jest bardzo prosty. Nastawiasz standardowy czasomierz kuchenny (tradycyjne mają kształt pomidora) na 25 minut. W czasie gdy czasomierz odlicza, nie pozwalasz, żeby *cokolwiek* przeszkadzało Ci w tym, co robisz. Jeżeli zadzwoni telefon, odbierz i zapytaj, czy możesz oddzwonić w ciągu 25 minut. Jeżeli ktoś będzie chciał zadać Ci pytanie, grzecznie poproś, żeby zadał je za 25 minut. Niezależnie od tego, co Ci przeszkadzi, po prostu przesuń to do momentu, aż czasomierz zadzwoni. W końcu mało która sprawa jest aż tak pilna, że nie może poczekać 25 minut.

---

<sup>2</sup> <http://www.pomodorotechnique.com/>.

---

Gdy czasomierz zadzwoni, natychmiast przerywasz to, co aktualnie robisz. Zajmujesz się wszystkimi sprawami, które zostały przesunięte w trakcie ostatniego pomidora. Następnie robisz sobie 5-minutową przerwę. Potem znowu nastawiasz czasomierz i zaczynasz kolejnego pomidora. Po czterech pomidorach robisz sobie dłuższą przerwę — mniej więcej 30-minutową.

Na temat tej techniki napisano już całkiem sporo i zachęcam Cię do poczytania o niej. Już jednak podany wyżej opis powinien dać Ci jej ogólny zarys.

Stosując tę technikę, dzielisz swój czas na czas pomidora i czas bez pomidora. Czas pomidora jest czasem produktywnym. To właśnie podczas każdego pomidora wykonujesz rzetelnie swoją pracę. Poza pomidorami są tylko spotkania, przerwy i inne zajęcia, które nie przyczyniają się do realizacji Twoich zadań.

Ile pomidorów możesz przerobić jednego dnia? Dobrego dnia może Ci się udać zrobić nawet 12 lub 14 pomidorów. Jeżeli zdarzy się gorszy dzień, to być może zdołasz zrobić zaledwie dwa lub trzy. Jeżeli zaczniesz liczyć pomidory, to szybko dowiesz się, jaką część dnia spędzasz na produktywnych zajęciach, a ile czasu poświęcasz na „inne rzeczy”.

Niektórym ta technika spodobała się tak bardzo, że szacują pracochłonność zadań w pomidorach, a swoją wydajność mierzą w pomidorach na tydzień. To jednak tylko wisienka na torcie. Prawdziwą zaletą techniki pomidora jest to 25-minutowe okno produktywnie spędzonego czasu, którego agresywnie bronisz przed jakimkolwiek wtrąceniami.

## Uniki

Czasami najzwyczajniej w świecie nie masz serca do pracy. Być może wynika to z tego, że zadanie, które na Ciebie czeka, jest tak straszne, nieprzyjemne albo po prostu nudne. Może sądzisz, że zadanie to popchnie Cię w kierunku niechcianej konfrontacji albo zapędzi w kozi róg. A może po prostu nie masz ochoty na akurat tę pracę.

## Odwroćenie priorytetów

Niezależnie od powodu zawsze znajdziesz jakiś sposób na unikanie faktycznej pracy. Przekonujesz się, że coś jest o wiele ważniejsze, i tym właśnie się zajmujesz. Takie działanie jest nazywane *odwróceniem priorytetów*. Podnosisz priorytet jednego zadania tylko po to, żeby opóźnić wykonanie zadania o faktycznie wyższym priorytecie. Odwrócenie priorytetów jest kłamstwem, którym sami siebie mamimy. Nie możemy znieść tego, co trzeba zrobić, dlatego przekonujemy się, że inne zadanie jest ważniejsze. Wiemy, że tak nie jest, ale i tak wmwawiamy sobie co innego.

Choć tak naprawdę wcale się nie okłamujemy. Tak naprawdę przygotowujemy sobie tylko kłamstwo na wypadek, gdy ktoś nas zapyta, co i dlaczego będziemy dzisiaj robić. Tworzymy sobie w ten sposób ochronę przed oceną innych.

Jak można się domyślać, nie jest to zachowanie profesjonalne. Profesjonalisci oceniają priorytet poszczególnych zadań, nie uwzględniając przy tym osobistych lęków i pragnień, a następnie wykonują te zadania w odpowiedniej kolejności.

## **Ślepe uliczki**

Ślepe uliczki są częścią życia każdego twórcy oprogramowania. Czasami podejmuję się decyzję, która kieruje nas na technologiczną ścieżkę prowadzącą donikąd. Im bardziej jesteśmy związaniani ze swoją decyzją, tym głębiej wchodzimy w tę dzicz. Jeżeli na szali waży się Twoja reputacja profesjonalisty, to z raz obranej ścieżki nie jezdzisz już nigdy.

Roztropność i doświadczenie pozwolą Ci unikać niektórych ślepych uliczek, ale nigdy nie uda Ci się przed nimi uciec. Dlatego właśnie potrzebna jest Ci umiejętność szybkiego rozpoznawania takich sytuacji i odwaga, żeby się z nich wycofać. Czasami jest to nazywane *regułą dziury*: gdy się w jakiejś znajdziesz, przestań kopać.

Profesjonalisci starają się nie przywiązywać do idei tak mocno, że później nie są w stanie jej porzucić i podążyć inną ścieżką. Starają się mieć umysł otwarty na inne pomysły, by po zabrnięciu w ślepą uliczkę mieć jakieś inne rozwiązania.

## **Marsze, bagna i bałagan**

Od ślepych uliczek gorszy jest jednak bałagan. Każdy bałagan Cię spowalnia, choć nigdy całkiem Cię nie zatrzymuje. Bałagan utrudnia posuwanie się naprzód, ale przy odpowiedniej determinacji postępy nadal są możliwe. Bałagan jest gorszy od ślepej uliczki, ponieważ zawsze widzisz w nim drogę prowadzącą naprzód, która dodatkowo sprawia wrażenie krótszej niż droga powrotna (choć to złudne wrażenie).

Widziałem już, jak bałagan w oprogramowaniu niszczył produkty i całe firmy. Widziałem, jak produktywność zespołów w kilka miesięcy zmieniała się z szybkiego jazzu w marsz żałobny. Nic poza bałaganem nie ma tak długofalowego i negatywnego wpływu na produktywność zespołu programistów. Naprawdę nic.

Problem polega na tym, że tworzenia bałaganu nie da się uniknąć, podobnie jak nie da się unikać ślepych uliczek. Doświadczenie i ostrożność pomagają trochę je omijać, ale w końcu i tak podejmiesz decyzję, w wyniku której powstanie jakiś bałagan.

Rozwój takiego bałaganu jest dobrze zakamuflowany. Tworzysz rozwiązanie jakiegoś prostego problemu, starając się, żeby powstający kod był prosty i czysty. Gdy zakres i złożoność tego

problemu rosną, starasz się rozbudowywać istniejący kod i nadal jak najlepiej utrzymywać jego czystość. W pewnym momencie uświadamiasz sobie, że podjęta już na samym początku decyzja była błędna i dlatego kod nie skaluje się dobrze wraz z napływaniem kolejnych wymagań.

To jest właśnie krytyczny moment! Możesz zawrócić i poprawić projekt, ale możesz też brnąć dalej. Zawrócenie w tym momencie wydaje się kosztowne, ponieważ zmusza Cię do przebudowy istniejącego kodu, ale już *nigdy później* zawrócenie nie będzie tak łatwe jak teraz. Jeżeli zdecydujesz się brnąć dalej, to zmienisz system w bagno, z którego możesz się już nigdy nie wydostać.

Profesjonalisci obawiają się bałaganu dużo bardziej niż ślepych uliczek. Zawsze szukają miejsc, w których bałagan zaczyna narastać, i nie szczędzą środków na jak najwcześniejszą ucieczkę od niego.

Dalsze posuwanie się w takim bagnie, jeżeli *wiesz*, że to jest bagno, to najgorszy z możliwych przykład odwrócenia priorytetów. Prac naprzód, oszukujesz siebie, swój zespół, swoją firmę i swoich klientów. Mówisz im, że wszystko będzie w porządku, podczas gdy wspólnie zmierzacie ku zagładzie.

## Wnioski

Profesjonalni programiści starają się jak najlepiej zarządzać swoim czasem i możliwościami koncentracji. Znają pokusy wynikające z odwrócenia priorytetów, a ich zwalczanie staje się dla nich sprawą honoru. Nie zamkują się na inne rozwiązania, ale starają się mieć otwarty umysł. Nie przywiązuje się za bardzo do jednego rozwiązania, tak żeby nie mieć oporów przed jego porzuceniem. Cały czas wypatrują też rosnącego bałaganu i starają się jak najszybciej posprzątać wypatrzone miejsca. Nie ma smutniejszego widoku niż zespół programistów bezproduktywnie walczących z coraz gębszym bagnem.



---

# 10 SZACOWANIE

---



Szacowanie jest jednym z najprostszym i jednocześnie najbardziej przerażającym działań, z którymi zmierzyć się musi profesjonalny programista. Uzależnione są przecież od niego wielkie pieniądze, a także nasza reputacja. Powoduje ono wiele naszych strachów i porażek. Szacowanie jest główną kością niezgody panującej między menedżerami a programistami. Jest też źródłem wszelkiego braku zaufania wpływającego na ten związek.

W 1978 roku byłem głównym programistą 32-kilobajtowego programu dla procesora Z-80, pisaneego w asemblerze. Program ten był zapisywany na 32 układach EEPROM o pojemności 1 KB. Wszystkie 32 układy scalone były następnie umieszczane na trzech płytach drukowanych, z których każda mogła pomieścić do 12 takich układów.

Obsługiwaliśmy wtedy setki urządzeń zainstalowanych w centralach telefonicznych w całych Stanach Zjednoczonych. Gdy tylko poprawialiśmy jakiś błąd albo dodawaliśmy nową funkcję, musieliśmy wysyłać serwisantów do każdego urządzenia, żeby wymienili wszystkie 32 układy!

To był prawdziwy koszmar. Zarówno układy, jak i same płytki były dość delikatne. Nóżki scalaków wyginały się i odłamywały. Ciągle wyginanie płyt drukowanych mogło spowodować uszkodzenia lutów. Ryzyko powstania uszkodzeń i błędów było ogromne. A koszt takich działań dla naszej firmy był o wiele za wysoki.

Mój szef Ken Finder zapytał mnie wtedy, czy nie da się czegoś w tym wszystkim poprawić. Chciał, żebyśmy wymyśliły jakiś sposób na wprowadzanie zmian do jednego układu bez konieczności jednoczesnej wymiany wszystkich pozostałych. Jeżeli czytasz moje książki albo słuchasz moich wykładów, to wiesz, jak bardzo zachęcam do stosowania niezależnego rozmieszczenia komponentów. To wtedy dostałem pierwszą lekcję.

Problemem naszego oprogramowania było to, że powstawało jako jeden wielki plik wykonywalny. Jeżeli do programu został dodany nowy wiersz, to wszystkie adresy z kolejnych wierszy musiały zostać zmienione. Niestety, każdy układ mieścił przestrzeń adresową o wielkości zaledwie 1 K, dlatego przy każdej zmianie programu zmieniała się zawartość niemal wszystkich układów.

Rozwiążanie tego problemu było całkiem proste. Każdy układ musiał zostać oddzielony od pozostałych. Każdy z nich musiał stać się niezależnym elementem komplikacji, który można by zapisywać niezależnie od pozostałych.

Zmierzyłem zatem wielkości wszystkich funkcji naszej aplikacji i napisałem prosty program dopasowujący je jak układankę do poszczególnych układów, pozostawiając w każdym z nich mniej więcej 100 bajtów wolnego miejsca na rozbudowę. Na początku każdego układu umieściłem tablicę wskaźników do wszystkich funkcji zapisanych w tym układzie. W czasie rozruchu urządzenia wskaźniki te były przenoszone do pamięci RAM. Cały kod w systemie został zmieniony tak, żeby funkcje były wywoływane wyłącznie przez te wskaźniki rezydujące w pamięci, ale nigdy bezpośrednio.

Dokładnie tak. Układy stały się obiektami z tablicami metod wirtualnych. Wszystkie funkcje były uruchamiane polimorficznie. I w ten sposób poznałem kilka zasad projektowania obiektowego, jeszcze zanim dowiedziałem się, czym jest obiekt.

Zyski z tej zmiany były ogromne. Mogliśmy wysyłać do serwisantów tylko pojedyncze układy, ale co ważniejsze, możliwe stało się wprowadzanie poprawek w działających systemach przez przeniesienie funkcji do pamięci RAM i zmianę wektora wywołania. Dzięki temu debugowanie i wprowadzanie poprawek u klienta stało się o wiele łatwiejsze.

Odchodzę jednak od tematu. Gdy Ken prosił mnie o rozwiązanie tego problemu, wspomniał coś o wskaźnikach do funkcji. Jakiś dzień lub dwa zajęło mi sformalizowanie tego pomysłu,

a potem przedstawiłem mu szczegółowy plan. Zapytał, jak długo potrwa wprowadzanie tych zmian, a ja odpowiedziałem, że zajmie mi to jakiś miesiąc.

Prace trwały całe *trzy* miesiące.

W całym swoim życiu pijany byłem dwa razy, a tylko raz spiłem się *całkowicie*. Było to w czasie przyjęcia bożonarodzeniowego firmy Teradyne w 1978 roku. Miałem wtedy 26 lat.

Przyjęcie odbywało się w biurze firmy, które składało się głównie z otwartych przestrzeni laboratoriów. Wszyscy przyjechali do firmy wcześniej, a potem przyszła potężna burza śnieżna, która uniemożliwiła przyjazd zamówionego zespołu i firmy kateringowej. Na szczęście mieliśmy sporo alkoholu.

Z tej nocy pamiętam bardzo mało. A to, co *zapamiętałem*, wolałbym raczej zapomnieć. Przedstawię Ci jednak pewien ważny moment.

Siedziałem na podłodze ze skrzyżowanymi nogami obok Kena (mojego szefa, który w tym czasie miał 29 lat i *nie* był pijany), jęcząc o tym, jak długo zajmuje mi praca nad wektoryzacją wywołań. Alkohol uwolnił moje skrywane dotychczas obawy i niepewność co do moich szacunków. Nie *sądzę*, że położyłem głowę na jego kolanach, ale takie szczegóły nie zachowały się w mojej pamięci.

Pamiętam jednak, że spytałem go, czy jest na mnie wściekły i czy sądzi, że cała ta operacja zajmuje za wiele czasu. Mimo że z całej tej nocy pamiętam niewiele, to jednak jego odpowiedź zapisała mi się w pamięci na stałe. Powiedział: „Tak, uważam, że zajmuje ci to strasznie dużo czasu, ale widzę też, że ciężko nad tym pracujesz i robisz ciągłe postępy. To jest coś, co jest nam bardzo potrzebne. I nie, nie jestem wściekły”.

## **Czym jest szacowanie?**

Problem polega na tym, że nasze szacunki traktujemy zupełnie odmiennie. Kadra menedżerska patrzy na nie jak na zobowiązania, natomiast programiści traktują je raczej jak przypuszczenia. A różnica między tymi znaczeniami jest ogromna.

## **Zobowiązanie**

Zobowiązanie jest czymś, czego musisz dotrzymać. Jeżeli zobowiązujesz się wykonać coś do wyznaczonego dnia, to po prostu *musisz* to zrobić do tego czasu. Jeżeli oznacza to, że musisz pracować 12 godzin dziennie, w weekendy i zrezygnować z rodzinnego urlopu, to tak musi być. Podejmując zobowiązanie, musisz go dotrzymać.

Profesjonalisci nie podejmują zobowiązań, chyba że dokładnie *wiedzą*, że mogą ich dotrzymać. To naprawdę jest takie proste. Jeżeli proszą Cię o zobowiązanie się do czegoś, a Ty nie masz *pewności*, czy będziesz w stanie to zrobić, to najlepiej będzie odmówić.

Jeżeli proszą Cię o zobowiązanie się do czegoś, a Ty wiesz, że to *jest możliwe*, ale będzie wymagało długich nadgodzin, weekendu i rezygnacji z urlopu, to tylko do Ciebie należy decyzja. Ale lepiej przygotuj się na te wszystkie poświęcenia.

Zobowiązania wymagają pewności. Inni ludzie, przyjmując Twoje zobowiązanie, zaczną na jego podstawie tworzyć własne plany. Koszt niedotrzymania zobowiązania będzie ogromny zarówno dla nich, jak i dla Twojej reputacji. Niedotrzymanie zobowiązania jest rodzajem nieuczciwości, tylko trochę mniej nieprzyjemnym od samego kłamstwa.

## Szacunek

Szacunek jest tak naprawdę tylko przypuszczeniem. Nie ma nic wspólnego ze zobowiązaniem. Nie wiąże się z obietnicami. Nietrafienie z szacunkiem nie jest żadną ujmą na honorze. Powodem, dla którego tworzymy szacunki, jest to, że *nie wiemy*, jak długo trwać będzie dane zadanie.

Niestety, większość programistów zupełnie nie umie szacować. Nie chodzi o przyswojenie sobie jakieś tajnej umiejętności szacowania — nic takiego nie istnieje. Powodem naszych błędnych szacunków jest to, że nie znamy natury szacowania.

Szacunek nie jest liczbą, ale *rozkładem*. Pomyśl proszę:

Mike: „Kiedy według twoich szacunków zakończysz to zadanie?”.

Peter: „Za trzy dni”.

Czy Peter naprawdę zrobi wszystko w ciągu trzech dni? Możliwe, ale jak bardzo prawdopodobne? Odpowiedź brzmi: nie mamy zielonego pojęcia. Co Peter miał na myśli, a czego dowiedział się Mike? Czy Mike będzie zaskoczony za trzy dni, jeżeli Peter nie będzie jeszcze gotowy? Niby dlaczego miałby być? Przecież Peter do niczego się nie zobowiązywał. Nie powiedział mu jednak też, jak bardzo prawdopodobne jest zakończenie prac w trzy dni w porównaniu z czterema albo pięcioma dniami.

Co się stanie, gdy Mike zapyta Petera o prawdopodobieństwo sprawdzenia się jego trzydniowego szacunku?

Mike: „Jakie są szanse na to, że skończysz w ciągu trzech dni?”.

Peter: „Całkiem spore”.

Mike: „Możesz jakoś je określić?”.

Peter: „Pięćdziesiąt, sześćdziesiąt procent”.

Mike: „Czyli istnieje niemała szansa, że zajmie to nawet cztery dni”.

Peter: „No tak, to może potrwać nawet pięć do sześciu dni, choć w to akurat wątpię”.

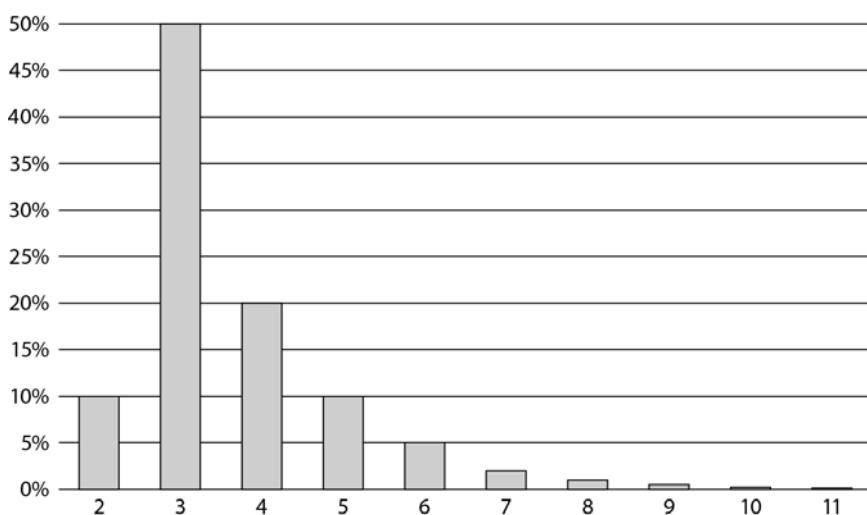
Mike: „Jak bardzo wątpisz?”.

Peter: „No nie wiem. Jestem na 95% pewien, że zrobię wszystko przed upływem sześciu dni”.

Mike: „Czyli może się to przeciągnąć do siedmiu dni?”.

Peter: „Tylko jeżeli wszystko naraz pójdzie źle. Rany, jeżeli wszystko pójdzie źle, to może mi zajść nawet dziesięć lub jedenaście dni. Ale taki natłok nieszczęścia jest za bardzo możliwy”.

Teraz zaczynamy zbliżać się do prawdy. Szacunek Petera jest *rozkładem prawdopodobieństwa*. Peter widzi prawdopodobieństwo ukończenia zadania w sposób przedstawiony na rysunku 10.1.



Rysunek 10.1. Rozkład prawdopodobieństwa

Teraz już wiemy, dlaczego Peter podał swój pierwotny trzydniowy szacunek. Na wykresie trzeci dzień ma najwyższy słupek. A zatem dla Petera jest to najbardziej prawdopodobny czas trwania zadania. Mike widzi to jednak nieco inaczej. Patrzy na prawą stronę wykresu i martwi się tym, że Peterowi prace mogą zajść nawet jedenaście dni.

Czy Mike powinien się tym martwić? Oczywiście! Murphy<sup>1</sup> zawsze znajdzie jakiś sposób na Petera, a zatem coś na pewno pójdzie nie tak.

## Ukryte zobowiązania

Teraz Mike ma problem. Nie ma pewności, ile czasu zajmie Peterowi wykonanie zadania. Żeby zmniejszyć tę niepewność, może poprosić Petera o jakieś zobowiązanie. Ale jest to coś, czego Peter nie może mu dać.

<sup>1</sup> Prawo Murphy'ego mówi, że jeżeli coś może pójść źle, to na pewno pójdzie źle.

Mike: „Peter, możesz mi podać konkretną datę zakończenia prac?”.

Peter: „Niestety nie. Jak już mówiłem, prawdopodobnie całość zakończę w trzy, może cztery dni”.

Mike: „Możemy zatem powiedzieć, że cztery?”.

Peter: „Nie, bo całość może potrwać pięć lub sześć dni”.

Jak na razie wszyscy zachowują się fair. Mike prosił o zobowiązanie, a Peter ostrożnie odmówił. W związku z tym Mike próbuje innego podejścia:

Mike: „No dobra, możesz jednak *spróbować* zrobić to w nie więcej niż sześć dni?”.

Prośba Mike'a brzmi dość niewinnie, a on zapewne nie ma też złych intencji. Ale tak naprawdę, to czego Mike żąda od Petera? Jak ma wyglądać to „próbowanie”?

Mówiłem już o tym w rozdziale 2. Słowo *próbować* ma wiele znaczeń. Jeżeli Peter zgodzi się „*spróbować*”, to zobowiąże się do zakończenia prac w sześć dni. Nie można tego inaczej interpretować. Zgoda na próbowanie jest zgodą na zakończenie prac.

Jak inaczej można to interpretować? Co właściwie Peter ma zamiar zrobić w ramach „*próbowania*”? Czy będzie pracował więcej niż osiem godzin? Taka jest jasna implikacja. Czy będzie musiał pracować w weekendy? To również się narzuca. Czy będzie musiał zrezygnować z urlopu? Owszem, to też się pod tym kryje. Te wszystkie rzeczy są częścią „*próbowania*”. Jeżeli Peter nie zrobi tych rzeczy, to Mike będzie mógł oskarżyć go o niedostateczne starania.

Zawodowcy jasno odróżniają szacunki od zobowiązań. Nie zobowiązują się, chyba że na pewno wiedzą, iż osiągną sukces. Starają się też nie brać na siebie *ukrytych* zobowiązań. Jak najwyraźniej starają się przedstawić rozkład prawdopodobieństwa swoich szacunków, aby menedżerowie mogli odpowiednio przygotować plany.

## **PERT**

W 1957 roku została utworzona technika PERT (*Program Evaluation and Review Technique* — technika oceny i rozwoju programów) mająca wspomagać projekt polarnej łodzi podwodnej dla U.S. Navy. Jednym z elementów tej techniki jest sposób obliczania szacunków. Przygotowano wtedy bardzo prostą, a jednocześnie skuteczną metodę przekształcania szacunków w rozkłady prawdopodobieństwa prezentowane menedżerom.

Szacując pracochnośc zadania, podajesz trzy wartości. Nazywane jest to *analizą trzech zmiennych*:

- **O:** Szacunek optymistyczny — ta wartość powinna być *bardzo optymistyczna*. Zakończenie zadania w tym czasie powinno być możliwe jedynie wówczas, gdy absolutnie

wszystko udałoby się bez problemów. Cała procedura będzie miała sens tylko wtedy, gdy prawdopodobieństwo prawidłowości tego szacunku będzie wynosiło około 1%<sup>2</sup>. W przypadku Petera, zgodnie z rysunkiem 10.1, powinien to być jeden dzień.

- **N:** Szacunek normalny — to szacunek z największym prawdopodobieństwem. Jeżeli narysowalibyśmy wykres, to miałby on najwyższy słupek. Według rysunku 10.1 byłoby to trzy dni.
- **P:** Szacunek pesymistyczny — tutaj również powinna być to *bardzo* pesymistyczna ocena. Powinna obejmować wszystko z wyjątkiem huraganów, wojny nuklearnej, zabłąkanej czarnej dziury i tym podobnych katastrof. Całość zadziała, jeżeli prawdopodobieństwo tej oceny będzie mniejsze niż 1%. Zgodnie z wykresem Petera byłoby to 12 dni.

Mamy już te trzy szacunki, więc możemy obliczyć rozkład prawdopodobieństwa, wykorzystując poniższe wzory:

$$\eta = \frac{O + 4N + P}{6}$$

Wartość  $\mu$  jest tutaj oczekiwany czasem trwania zadania. W przypadku Petera będzie to  $(1 + 12 + 12)/6$ , czyli jakieś 4,2 dnia. W przypadku większości zadań będzie to wartość nieco pesymistyczna, ponieważ prawa strona rozkładu prawdopodobieństwa zawsze jest bardziej rozciągnięta od lewej<sup>3</sup>.

$$\sigma = \frac{P - O}{6}$$

Wartość  $\sigma$  jest odchyleniem standardowym<sup>4</sup> rozkładu prawdopodobieństwa dla danego zadania. Jest miarą niepewności danego zadania. Jeżeli wartość ta jest duża, to duża jest też niepewność. W przypadku Petera wynosi ona  $(12 - 1)/6$ , czyli jakieś 1,8 dnia.

Otrzymując od Petera szacunek 4,2/1,8 Mike wie, że zadanie najpewniej zostanie wykonane w ciągu 5 dni, ale może też zająć 6, a nawet 9 dni.

Mike nie zajmuje się jednak tylko tym jednym zadaniem. Zarządza projektem składającym się z wielu różnych zadań. Peter otrzymał trzy z nich, które muszą być wykonywane w określonej kolejności. Peter przygotował dla tych zadań szacunki przedstawione w tabeli 10.1.

<sup>2</sup> Dokładna wartość w przypadku rozkładu normalnego to 1:769 albo 0,13% albo 3 sigma. Prawdopodobieństwo jeden do tysiąca jest już całkiem bezpieczne.

<sup>3</sup> PERT zakłada, że przypomina to rozkład beta. To całkiem dobre założenie, ponieważ minimalny czas trwania zadania często jest bardziej pewny niż czas maksymalny. [McConnell2006], rysunek 1.3.

<sup>4</sup> Jeżeli nie wiesz, czym jest odchylenie standardowe, to znajdź dobry podręcznik do rachunku prawdopodobieństwa i statystyki. To pojęcie jest całkiem łatwe do ogarnięcia, a bywa niezwykle przydatne.

**Tabela 10.1.** Zadania Petera

| Zadanie | Optymistyczne | Normalne | Pesymistyczne | $\mu$ | $\sigma$ |
|---------|---------------|----------|---------------|-------|----------|
| Alfa    | 1             | 3        | 12            | 4,2   | 1,8      |
| Beta    | 1             | 1,5      | 14            | 3,5   | 2,2      |
| Gamma   | 3             | 6,25     | 11            | 6,5   | 1,3      |

Nie sądzisz, że zadanie „beta” jest trochę dziwne? Wygląda na to, że Peter ma sporą pewność co do czasu ukończenia, ale może zdarzyć się coś, co bardzo utrudni mu pracę. Jak Mike powinien to zinterpretować? Jaki czas ma on zaplanować na ukończenie wszystkich trzech zadań przez Petera?

Okazuje się, że za pomocą kilku prostych obliczeń Mike może połączyć wszystkie zadania Petera i wyznaczyć rozkład prawdopodobieństwa całego zestawu zadań. Obliczenia te są naprawdę nieskomplikowane:

$$\mu_{\text{sekwencji}} = \sum \eta_{\text{zadania}}$$

W przypadku dowolnej sekwencji zadań oczekiwany czas jej trwania jest prostą sumą oczekiwanych czasów trwania wszystkich zadań z tej sekwencji. A zatem skoro Peter miał do wykonania trzy zadania o szacunkach 4,2/1,8; 3,5/2,2 i 6,5/1,3, to prawdopodobnie uda mu się zakończyć je wszystkie w 14 dni:  $4,2 + 3,5 + 6,5$ .

$$\sigma_{\text{sekwencji}} = \sqrt{\sum \sigma_{\text{zadania}}^2}$$

Standardowe odchylenie całej sekwencji jest pierwiastkiem kwadratowym sumy kwadratów standardowych odchyleń poszczególnych zadań. Oznacza to, że standardowe odchylenie wszystkich trzech zadań Petera wynosi mniej więcej 3.

$$\begin{aligned} (1,8^2 + 2,2^2 + 1,3^2)^{1/2} &= \\ (3,24 + 2,48 + 1,69)^{1/2} &= \\ 9,77^{1/2} &= \sim 3,13 \end{aligned}$$

Dzięki temu Mike wie, że zadania Petera zajmą najpewniej 14 dni, ale całkiem możliwe jest też, że będzie on na nie potrzebował 17 dni ( $1\sigma$ ), a istnieje też prawdopodobieństwo 20 dni ( $2\sigma$ ). Wszystkie zadania mogą zająć też jeszcze więcej czasu, ale to akurat jest bardzo mało prawdopodobne.

Przyjrzyj się jeszcze raz tabeli z wszystkimi szacunkami. Czujesz chęć zakończenia wszystkich trzech zadań w pięć dni? W końcu optymistyczne szacunki to 1, 1 i 3 dni. Nawet szacunki normalne sumują się do zaledwie 10 dni. W jaki sposób całość urosła do 14 dni z możliwością dalszego wzrostu do 17 lub 20? Otóż wskaźnik niepewności poszczególnych zadań sprawia, że cały plan nabiera *realizmu*.

Jeżeli masz trochę więcej niż kilka lat doświadczeń w programowaniu, to z pewnością zdarzyło Ci się widzieć projekty, które początkowo szacowano bardzo optymistycznie, a potem zajmowały trzy do pięciu razy więcej czasu. Przedstawiona tu prosta technika PERT jest jedną z metod pozwalających na unikanie zbyt optymistycznych oczekiwania. Profesjonalni programiści starają się definiować bardzo rozsądne oczekiwania mimo nacisków, żeby spróbować działać szybko.

## Szacowanie zadań

Mike i Peter popełnili straszliwy błąd. Mike pytał Petera, jak długo potrwa jego zadania. Peter podał mu szczerze szacunki, ale zapomnieli o opiniach pozostałych członków zespołu. Być może będą mieli inne pomysły.

Najważniejszym z dostępnych Ci narzędzi do szacowania zadań są otaczający Cię ludzie. Oni mogą zobaczyć coś, co Ty przeoczyłeś. To oni mogą pomóc Ci przygotować dokładniejsze szacunki.

### Wideband Delphi

W latach 70. XX wieku Barry Boehm zaprezentował technikę szacowania, którą nazwał „wideband delphi”<sup>5</sup>. Później przez lata powstało wiele jej odmian. Niektóre są bardziej formalne, a inne zupełnie nieformalne, ale wszystkie mają pewną cechę wspólną: konsensus.

Sama strategia jest prosta. Zespół się zbiera, omawia zadanie, szacuje je i powtarza całą dyskusję i szacowanie tak długo, aż zostanie osiągnięte porozumienie.

Pierwotna metoda opisana przez Boehma wymagała przynajmniej kilku spotkań i dokumentów, co jak na mój gust wiązało się ze zbyt wielkim ceremoniałem i przygotowaniami. Wolę jednak stosować rozwiązania z niewielką nadmiarowością nakładu pracy, takie jak poniższe.

### Latające palce

Wszyscy siedzą wokół stołu i omawiają po kolej i wszystkie zadania. Rozmawiają o tym, co jest związane z każdym zadaniem, co może je skomplikować i jak można je zaimplementować. Uczestnicy spotkania umieszczają swoje ręce pod stołem i podnoszą od zera do pięciu palców w zależności od własnej oceny czasochłonności zadania. Prowadzący spotkanie liczy 1, 2, 3 i wszyscy naraz pokazują swoje ręce.

---

<sup>5</sup> [Boehm81].

Jeżeli wszyscy się na to zgodzą, to przechodzą do następnego zadania, a jeżeli nie, to dyskutują dalej, by ustalić przyczynę swojej niezgody. Całość powtarza się tak długo, aż zostanie uzyskany konsensus.

Zgoda nie musi być całkowita. Wystarczy, że poszczególne szacunki nie będą się od siebie bardzo różniły. Na przykład mieszkańców trójkę i czwórkę również można uznać za porozumienie. Jeżeli jednak wszyscy podnoszą cztery palce, a jedna osoba pokazuje tylko jeden palec, to jest o czym rozmawiać.

Skalę szacunków ustala się zaraz na początku spotkania. Może to być liczba dni przeznaczonych na realizację zadania, ale można też ustalić ciekawszą skalę typu: „liczba palców razy 3” albo „liczba palców do kwadratu”.

Ważne jest to, że wszyscy muszą jednocześnie pokazać swoje palce. Nie chcemy, żeby poszczególne osoby zmieniały szacunki na podstawie ocen pozostałych.

### **Planujący poker**

W 2002 roku James Grenning napisał wspaniały artykuł<sup>6</sup> opisujący „planującego poker” (ang. *planning poker*). Ten wariant techniki wideband delphi stał się tak popularny, że kilka różnych firm wykorzystało go do przygotowania materiałów marketingowych w postaci talii kart do planującego pokeru<sup>7</sup>. Istnieje nawet strona internetowa o nazwie planningpoker.com, za pomocą której można przeprowadzić sesję planowania z rozproszonym zespołem.

Idea jest bardzo prosta. Każdy członek zespołu dostaje zestaw kart z różnymi numerami. Liczby od 0 do 5 sprawdzają się całkiem dobrze, a jednocześnie cały system jest wtedy równoznaczny z *latającymi palcami*.

Wybierane jest zadanie i zaczyna się dyskusja. W pewnym momencie prowadzący prosi wszystkich o wybranie karty. Członkowie zespołu wybierają kartę odpowiadającą ich ocenie i trzymają ją tak, żeby nikt nie widział wybranej liczby. Następnie prowadzący nakazuje wszystkim pokazać swoje karty.

Reszta działa dokładnie tak samo jak latające palce. Jeżeli wszyscy się zgadzają, to szacunek jest przyjmowany, w przeciwnym wypadku karty wracają do zestawów, a gracze do dyskusji.

Wybieranie właściwych kart do zestawu stało się już całkiem osobną dziedziną „nauki”. Niektórzy poszli już tak daleko, że używają kart odpowiadających ciągowi Fibonacciego. Inni dodali jeszcze karty ze znakami nieskończoności i zapytania. Osobiście uważam, że karty o oznaczeniach 0, 1, 3, 5, 10 powinny wystarczyć.

---

<sup>6</sup> [Grenning2002].

<sup>7</sup> <http://store.mountaingoatsoftware.com/products/planning-poker-cards>.

## Szacowanie pokrewieństwa

Szczególnie ciekawą wariację techniki wideband delphi zaprezentował mi kilka lat temu Lowell Lindstrom. Rozwiążanie to dobrze sprawdziło się kilka razy z różnymi klientami i zespołami.

Wszystkie zadania są zapisane na kartach bez żadnych szacunków dotyczących pracochłonności. Zespół szacujący stoi wokół stołu albo przy ścianie, na której karty są poprzyczepiane losowo. Członkowie zespołu nie rozmawiają ze sobą, ale po prostu sortują karty względem siebie. Dłuższe zadania wędrują na prawo, a krótsze na lewo.

Każdy członek zespołu może w dowolnym momencie przesunąć dowolną kartę, nawet jeśli została ona już przesunięta przez kogoś innego. Każda karta przesunięta więcej niż 1 raz jest odkładana na bok do omówienia.

Ostatecznie ciche sortowanie kart się kończy i można rozpocząć dyskusję. Omawiane są różnice zdań co do kolejności ułożenia kart. Szybkie sesje projektowania albo ręcznie rysowane diagramy mogą ułatwić uzyskanie konsensusu.

Następnym krokiem jest rysowanie linii pomiędzy kartami, które reprezentują grupy rozmiarów liczone w dniach, tygodniach albo punktach. Tradycyjnie stosuje się pięć grup zgodnych z ciągiem Fibonacciego (1, 2, 3, 5, 8).

## Szacunki trójwartościowe

Przedstawione tu techniki sprawdzają się przy wybieraniu normalnych szacunków poszczególnych zadań. Jak jednak wspominałem wcześniej, najczęściej chcemy uzyskać trzy wartości szacunków pozwalające na określenie rozkładu prawdopodobieństwa. Optymistyczny i pesymistyczny szacunek każdego zadania można szybko określić za pomocą każdego z przedstawionych wariantów techniki wideband delphi. Na przykład stosując technikę planującego pokera, wystarczy poprosić o podniesienie kart z szacunkiem pesymistycznym, a potem optymistycznym.

# Prawo wielkich liczb

Szacunki zawsze są obarczone błędem. To właśnie dlatego nazywa się je szacunkami. Jedną z metod radzenia sobie z takimi błędami jest wykorzystanie *prawa wielkich liczb*<sup>8</sup>. Z tego prawa wynika, że jeżeli jedno wielkie zadanie podzielimy na kilka mniejszych i oszacujemy je niezależnie od siebie, to suma tych szacunków będzie dokładniejsza od szacunku dotyczącego tego wielkiego zadania. Powodem zwiększenia dokładności jest to, że błędy w mniejszych zadaniach mają tendencję do znoszenia się wzajemnie.

---

<sup>8</sup> [http://pl.wikipedia.org/wiki/Prawo\\_wielkich\\_liczb](http://pl.wikipedia.org/wiki/Prawo_wielkich_liczb).

Oczywiście to optymistyczne podejście. Błędy w szacunkach zwykle są związane z niedoszacowaniem, a rzadziej z przeszacowaniem, dlatego wzajemne znoszenie się błędów nigdy nie jest doskonałe. Mimo to podzielenie wielkiego zadania na kilka mniejszych i szacowanie ich niezależnie od siebie nadal sprawdza się doskonale. Część błędów *faktycznie* się wzajemnie zniesie, a samo podzielenie zadania sprawia, że lepiej poznajemy jego szczegóły i odkrywamy ewentualne pułapki.

## Wnioski

Profesjonalni programiści wiedzą, jak przedstawić swoim przełożonym szacunki, które ci mogą wykorzystać do planowania. Nie składają obietnic, których nie mogą dotrzymać, i nie biorą na siebie zobowiązań, jeżeli nie mają pewności, czy będą w stanie ich dotrzymać.

Gdy profesjonalista bierze na siebie zobowiązanie, to podaje *mocne* dane i się z niego wywiązuje. W większości przypadków profesjonalisci nie biorą takich zobowiązań, ale podają probabilistyczne szacunki opisujące oczekiwany czas ukończenia zadania i możliwą wariancję.

Profesjonalni programiści współpracują z innymi członkami swojego zespołu i wspólnie starają się określić szacunki zadań, które później przedstawią menedżerom.

Techniki opisane w tym rozdziale są *przykładami* różnych metod wykorzystywanych przez programistów do tworzenia szacunków. Nie są to jedyne takie techniki i niekoniecznie są najlepsze z możliwych. Są to techniki, które sprawdziły się w mojej pracy.

## Bibliografia

[McConnell2006]: Steve McConnell, *Software Estimation: Demystifying the Black Art*, Redmond, WA: Microsoft Press, 2006.

[Boehm81]: Barry W. Boehm, *Software Engineering Economics*, Upper Saddle River, NJ: Prentice Hall, 1981.

[Grenning2002]: James Grenning, *Planning Poker or How to Avoid Analysis Paralysis while Release Planning*, kwiecień 2002, <http://renaissancesoftware.net/papers/14-papers/44-planing-poker.html>.

---

# 11 PRESJA

---



Wyobraź sobie, że doświadczasz fenomenu przebywania poza ciałem, obserwujesz siebie na stole operacyjnym w czasie, gdy chirurg wykonuje operację na Twoim otwartym sercu. Ten chirurg stara się uratować Twoje życie, ale czasu jest mało, dlatego musi zdążyć przed pewnym ostatecznym terminem. I ten termin *naprawdę* jest ostateczny.

Jak wyobrażasz sobie zachowanie tego lekarza? Chcesz, żeby zachował spokój i rozsądek? Chcesz, żeby wydawał swojemu zespołowi jasne i precyzyjne polecenia? Chcesz, żeby postępował zgodnie z wyuczonymi procedurami i według najlepszych zasad swojego zawodu?

Czy może raczej wolisz, żeby pocił się i przeklinał? Chcesz, żeby rozbijał i rozrzucał swoje narzędzia? Chcesz, żeby obwiniał zarząd o nierealistyczne oczekiwania i ciągle narzekał na brak czasu? Chcesz, żeby zachowywał się jak profesjonalista, czy jak typowy programista?

Profesjonalny programista pod presją jest spokojny i zdecydowany. Gdy presja wzrasta, odwołuje się do swojego wyszkolenia i swojej dyscypliny, wiedząc, że to najlepsza metoda na dotrzymanie terminów i zobowiązań.

W 1988 roku pracowałem w firmie Clear Communications. Był to typowy start-up, który nigdy tak naprawdę nie wystartował. Przebrnęliśmy przez pierwsze kłopoty finansowe, a zaraz potem pojawiły się kolejne, i jeszcze jedne.

Początkowa wizja produktu wyglądała bardzo obiecująco, ale nigdy nie udało się nam odpowiednio zdefiniować jego architektury. Najpierw produkt miał składać się zarówno z oprogramowania, jak i ze sprzętu. Potem przekształcił się w samo oprogramowanie. Platforma dla tego oprogramowania zmieniła się z komputerów PC na Sparc. Potencjalni klienci zmienili się z bogatych firm w odbiorcę masowego. Ostatecznie nawet cel naszego produktu zmienił się, ponieważ firma szukała czegokolwiek, co wygenerowałoby zyski. W ciągu niemal czterech lat, które spędziłem w tej firmie, nie zobaczyła ona chyba nawet grosza przychodu.

Nie muszę zatem nadmieniać, że pracujący w niej programiści znajdowali się pod ciągłą presją. Przed terminalem w biurze spędziliśmy wiele bardzo długich nocy, a nawet jeszcze dłuższych weekendów. Pisaliśmy w języku C funkcje o *długości 3000 wierszy*. Pojawiały się awantury z wykrzykiwanymi obiegami. Często knuto intrigi i planowano podstęp. Pięści przebiły ściany, długopisy rzucone w gniewie odbijały się od tablic, na ścianach pojawiały się wydrapane karykatury nielubianych kolegów. Ten strumień stresu i gniew nigdy się nie kończyły.

Terminy były wyznaczane przez określone wydarzenia. Poszczególne funkcje miały być gotowe na targi albo prezentacje dla klientów. Na następną prezentację musieliśmy mieć gotowe wszystko, o co poprosił klient, nieważne, jak głupie to były żądania. Zawsze brakowało nam czasu i zawsze mieliśmy jakieś zaległości. Tworzone plany przyprawiały o zawrót głowy.

Jeżeli pracowałeś 80 godzin w tygodniu, mogłeś zasłużyć na miano bohatera. Takim samym bohaterem można było zostać po przygotowaniu jakiegoś paskudnego kodu na prezentację dla klienta. Jeżeli ktoś się dostatecznie starał, mógł dostać awans. Jeżeli ktoś się nie starał, wylatywał na bruk. To był prawdziwy start-up i chodziło tylko o „wypracowane wartości”. I w 1988 roku, mimo 20 lat doświadczenia, wsiadłem do tego wózka.

Byłem jednym z menedżerów informujących pracujących dla mnie programistów, że muszą pracować szybciej i więcej. Byłem jednym z tych pracujących po 80 godzin wyrobników, tworzących liczące 3000 wierszy funkcje w C o 2 nad ranem, podczas gdy moje dzieci spały

---

w domu bez opieki ojca. Byłem jednym z krzyczących i rzucających długopisami wściekłych ludzi. Wyrzucałem ludzi z pracy, jeżeli nie chcieli się dopasować. To był koszmar. Ja sam byłem koszmarem.

Potem nadszedł dzień, w którym moja żona zmusiła mnie do spojrzenia w lustro. Nie spodobało mi się to, co w nim zobaczyłem. Powiedziała mi, że przebywanie ze mną nie jest przyjemne, i musiałem się z nią zgodzić. Ale wcale mi się to nie podobało, dlatego wściekły wybiegłem z domu, zacząłem chodzić po okolicy bez żadnego celu. Łaziłem tak jakieś pół godziny, gotując się ze złości. I wtedy zaczęło padać.

I coś w mojej głowie się przełączyło. Zacząłem się śmiać. Śmiałem się ze swojej własnej głupoty. Śmiałem się ze swojego stresu, z człowieka, którego zobaczyłem w lustrze, a który uprzykrzał życie sobie i swoim bliskim w imię... czego?

Tego dnia wszystko się zmieniło. Przystałem robić tak szalone nadgodziny i zakończyłem życie w wielkim stresie. Przystałem rzucać długopisami i pisać gigantyczne funkcje w C. Stwierdziłem, że będę cieszył się swoją karierą, wykonując swoją pracę dobrze, a nie głupio.

Odszedłem z tej pracy tak profesjonalnie, jak tylko mogłem, i zostałem konsultantem. Od tego dnia nigdy nikogo nie nazwałem już „szefem”.

## Unikanie presji

Najlepszym sposobem na uzyskanie spokoju mimo presji jest unikanie sytuacji powodujących presję. Takie uniki zapewne nie zlikwidują presji całkowicie, ale pozwolą na zminimalizowanie i skrócenie tych nieprzyjemnych okresów.

## Zobowiązania

Jak już mówiłem w rozdziale 10., ważne jest, żeby nie zobowiązywać się do dotrzymywania terminów, których nie jesteśmy całkowicie pewni. Menedżerowie zawsze nastają na tego typu zobowiązania, ponieważ eliminują one ryzyko. Naszym zadaniem jest kwantyfikacja tego ryzyka i przedstawienie go menedżerom, tak żeby mogli nim odpowiednio zarządzać. Przyjmowanie nierealistycznych zobowiązań niweczy ten cel, a zatem nie służy ani nam, ani naszym przełożonym.

Czasami zobowiązania są przenoszone na nas. Okazuje się, że menedżerowie potrafią złożyć klientowi obietnice, nawet się z nami nie konsultując. W takiej sytuacji honor nakazuje nam pomóc menedżerom wywiązać się ze zobowiązania. Trzeba jednak pamiętać, że honor wcale *nie* nakazuje nam go zaakceptować.

To bardzo ważna różnica. Profesjonalisci zawsze wspomagają menedżerów w poszukiwaniu sposobu osiągnięcia celu. Ale jednocześnie niekoniecznie przyjmują na siebie zobowiązania

złożone w ich imieniu. Może się przecież okazać, że nie ma żadnego sposobu na dotrzymanie złożonej obietnicy, a wtedy osoba składająca taką obietnicę musi przyjąć na siebie odpowiedzialność za nią.

Łatwo powiedzieć. Jeśli firma się chwieje, a Twoje wypłaty są opóźniane z powodu niedotrzymanych zobowiązań, trudno nie ulegać presji. Jeżeli jednak zachowujesz się profesjonalnie, to przynajmniej możesz zacząć szukać nowej pracy z podniesioną głową.

## **Utrzymanie czystości**

Sposobem na szybki rozwój projektu i dotrzymywanie wszystkich terminów jest odpowiednie utrzymanie czystości. Profesjonalisci nie poddają się pokusie tworzenia bałaganiarskiego kodu tylko po to, żeby robić to szybciej. Profesjonalisci wiedzą doskonale, że wyrażenie „szybko i paskudnie” to oksymoron. Paskudny kod zawsze oznacza spowolnienie!

Możemy unikać presji, utrzymując nasz system, kod i projekt w jak najlepszym stanie. Nie oznacza to, że mamy spędzać godziny na polerowaniu kodu. To znaczy tylko tyle, że nie powinniśmy tolerować bałaganu. Wiemy, że bałagan nas spowalnia, przez co nie dotrzymujemy terminów i łamiemy dane słowo. A zatem staramy się, żeby wyniki naszej pracy były tak schludne, jak tylko się da.

## **Dyscyplina kryzysowa**

Obserwując swoje zachowania w czasie kryzysu, możesz stwierdzić, w co naprawdę wierzysz. Jeżeli podczas kryzysu przestrzegasz swojej dyscypliny, to znaczy, że naprawdę wierzysz w jej skuteczność. Jeżeli jednak w czasie kryzysu zmieniasz sposób postępowania, to znaczy, że tak naprawdę nie uznajesz swoich normalnych zachowań za skuteczne.

Jeżeli podczas normalnej pracy stosujesz metodologię TDD, ale porzucasz ją w czasie kryzysu, to nie uznajesz jej za naprawdę użyteczną. Jeżeli w spokojnych czasach utrzymujesz w swoim kodzie porządek, ale gdy tylko pojawi się kryzys, dopuszczasz powstanie bałaganu, to znaczy, że nie wierzysz w to, że bałagan może Cię spowolnić. Jeżeli w kryzysie programujesz w parach, ale normalnie tego nie robisz, to uznajesz, że programowanie w parach jest skuteczniejsze od programowania solo.

Wybierz rozwiązania, które uznasz za skuteczne w czasie kryzysu, *a potem stosuj je przez cały czas*. Wykorzystywanie tych rozwiązań w normalnej pracy jest najlepszym sposobem na unikanie kryzysu.

Nie zmieniaj też swojego zachowania w czasie, gdy pojawi się presja. Jeżeli Twoje rozwiązania są najlepszym sposobem pracy, to należy ich przestrzegać również podczas najgłębszego kryzysu.

## Jak radzić sobie z presją

Unikanie i łagodzenie presji z całą pewnością są dobrymi metodami, ale czasami presja dopada nas mimo naszych najlepszych intencji i zabiegów. Czasami projekt trwa dłużej, niż ktokolwiek mógł przypuszczać. Czasami pierwotny projekt okazuje się błędny i musi zostać przebudowany. Czasami tracimy wartościowego członka zespołu albo klienta. Czasami podejmujemy zobowiązania, których nie jesteśmy w stanie dotrzymać. Co robić w takiej sytuacji?

### Nie panikuj

Opanuj stres. Bezsenne noce nie pomogą Ci w szybszej pracy. Podobnie nie pomoże Ci siedzenie i zamartwianie się. A najgorsze, co możesz zrobić, to zacząć się spieszyć! Za wszelką cenę opieraj się tej pokusie. Pośpiech wpędzi Cię tylko głębiej w bagno.

Zdecydowanie lepiej będzie zwolnić. Jeszcze raz przemyśleć problem. Zaplanować kurs na najlepsze z możliwych rozwiązań, a następnie podążać w tym kierunku w stałym i rozsądny tempie.

### Informuj

Poinformuj swój zespół i przełożonego o kłopotach. Przedstaw im swój plan wybrnięcia z nich i poproś o uwagi i wskazówki. Unikaj niespodzianek. Poza niespodziankami chyba nic nie wprawia ludzi w większy gniew i nie prowokuje do mniej racjonalnych zachowań. Niespodzianki dziesięciokrotnie zwiększą presję.

### Polegaj na swoich metodach

Gdy robi się ciężko, *polegaj na swoich metodach pracy*. Powodem, dla którego stosujesz te metody, jest to, że pomagają Ci one w czasach dużej presji. To właśnie wtedy należy szczególnie uważnie trzymać się sprawdzonych rozwiązań. To nie jest odpowiedni czas, żeby je kwestionować i porzucać.

Zamiast panicznie poszukiwać czegokolwiek, co mogłoby pomóc szybciej wykonać pracę, jeszcze dokładniej zaczni ją postępować zgodnie z przyjętymi metodami. Jeżeli stosujesz TDD, to pisz jeszcze więcej testów jednostkowych niż zwykle. Jeżeli bezwzględnie refaktorujesz kod, to jeszcze mocniej się do tego przyłoż. Jeżeli piszesz niewielkie funkcje, to pisz jeszcze mniejsze. Jedyną metodą na wybrnięcie z tego kociołka jest poleganie na tym, co już wiesz, że działa — na swoich sprawdzonych metodach pracy.

## **Sprawadź pomoc**

Programuj w parach! Gdy grunt zaczyna się palić, znajdź kogoś, kto zechce programować z Tobą w parze. Zadanie ukończysz szybciej, z mniejszą liczbą błędów. Twój partner ułatwi Ci trzymanie się stałych metod pracy i powstrzyma Twoją panikę. Twój partner zauważy szczegóły, które umkną Tobie, będzie miał ciekawe pomysły, a kiedy się zdekoncentrujesz, będzie mógł przejąć pałeczkę.

Natomiast gdy ktoś inny ma kłopoty, zaproponuj mu programowanie w parach. Pomóż mu wydostać się z dółka.

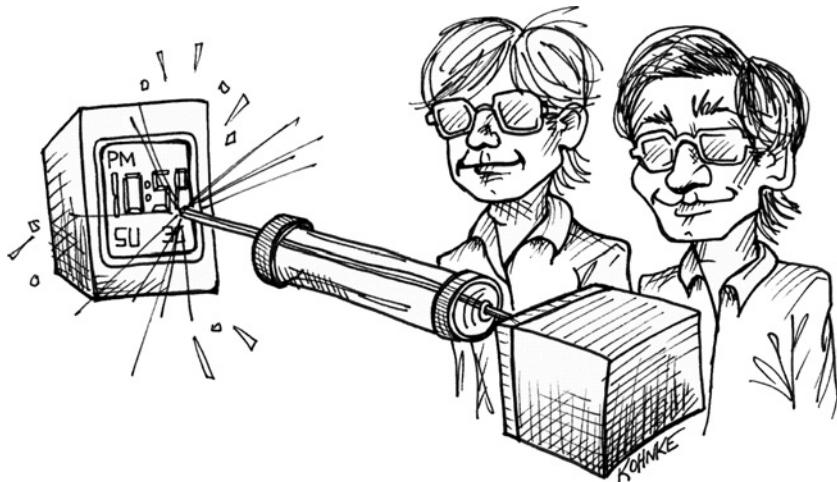
## **Wnioski**

Podstawową sztuką radzenia sobie z presją jest unikanie jej, gdy tylko to możliwe. Gdy jednak nas dopadnie, trzeba jakoś ją przetrwać. Presji można unikać przez odpowiednie dobieranie zobowiązań, przestrzeganie uznanych metod pracy i utrzymywanie czystego kodu. Presję można przetrwać, zachowując spokój, informując o istniejącej sytuacji, przestrzegając metod pracy i prosiąc o pomoc.

---

# 12 WSPÓŁPRACA

---



Większość oprogramowania jest tworzona przez zespoły. Zespoły działają najskuteczniej, gdy ich członkowie profesjonalnie ze sobą współpracują. Nieprofesjonalne jest bycie w zespole odrudkiem i samotnikiem.

W 1974 roku miałem 22 lata. Moje małżeństwo z cudowną Ann Marie trwało zaledwie sześć miesięcy. Dopiero za rok miało się urodzić moje pierwsze dziecko: Angela. W tym czasie pracowałem w oddziale firmy Teradyne znany jako Chicago Laser Systems.

Razem ze mną pracował mój kolega ze szkoły Tim Conrad. W ramach naszej współpracy przygotowaliśmy kilka małych cudów. W jego piwnicy budowaliśmy razem komputery. W mojej piwnicy zbudowaliśmy drabinę Jakuba. Nauczyliśmy się programować PDP-8 i łączyć układy scalone i tranzystory tak, żeby powstał kalkulator.

Byliśmy programistami pracującymi nad systemem wykorzystującym laser do bardzo dokładnego dopasowania parametrów elementów elektronicznych, takich jak oporniki i kondensatory. Na przykład przygotowaliśmy kryształ do pierwszego cyfrowego zegarka — Motoroli Pulsar.

Programowaliśmy wtedy na komputerze M365, który był klonem PDP-8 używanym w firmie Teradyne. Pisaliśmy programy w assemblerze, a wszystkie pliki przechowywaliśmy na taśmach magnetycznych. Oczywiście mogliśmy edytować programy na ekranie, ale było to dość uciążliwe, dlatego większość naszego kodu czytaliśmy i wstępnie poprawialiśmy na wydrukach.

Nie mieliśmy żadnych funkcji pozwalających na przeszukiwanie istniejącego kodu. Nie było sposobu na znalezienie wszystkich miejsc, w których była wywoływana dana funkcja albo używana określona stała. Jak możesz sobie wyobrazić, nie ułatwiało nam to pracy.

Pewnego dnia Tim i ja postanowiliśmy, że napiszemy sobie generator referencji. Program powinien odczytywać dane z taśmy i wypisywać wszystkie symbole z podaniem pliku i wiersza, w których dany symbol był używany.

Pierwotny program napisaliśmy całkiem szybko. Nie był szczególnie skomplikowany. Po prostu odczytywał dane z taśmy, parsował składnię assemblera, tworzył tabelę symboli i do poszczególnych pozycji dopisywał referencje. Działało to świetnie, ale niestety okropnie wolno. Ponad godzinę zajmowała programowi analiza naszego głównego programu operacyjnego (*Master Operating Program — MOP*).

Całość działała tak wolno dlatego, że rosnącą tabelę symboli przechowywaliśmy w jednym buforze pamięci. Po znalezieniu każdej nowej referencji dopisywaliśmy ją do bufora, przesuwając pozostałe dane w dół o kilka bajtów.

Ani Tim, ani ja nie byliśmy ekspertami od struktur danych i algorytmów. Nigdy nie słyszeliśmy o tablicach mieszczących i przeszukiwaniu binarnym. Nie mieliśmy pojęcia, jak można usprawnić algorytm. Wiedzieliśmy tylko, że to, co zrobiliśmy, było za wolne.

Próbowaliśmy zatem jednej rzeczy za drugą. Próbowaliśmy umieszczać referencje w listach. Próbowaliśmy pozostawiać w tablicy przerwy i przesuwać dane w buforze, gdy te przerwy zostały zapełnione. Próbowaliśmy tworzyć listy takich przerw. Wypróbowaliśmy dziesiątki szalonych pomysłów.

Staliśmy przed tablicą w biurze, rysowaliśmy diagramy naszych struktur danych i robiliśmy obliczenia, próbując przewidzieć wydajność. Codziennie przychodziliśmy do biura z jakimś nowym pomysłem. Współpracowaliśmy ze sobą jak szaleni.

Część z naszych prób faktycznie podniosła wydajność. Część raczej ją zmniejszyła. To doprowadzało nas do szewskiej pasji. Wtedy pierwszy raz dowiedziałem się, jak trudno optymalizuje się oprogramowanie i jak nieintuicyjny jest to proces.

Ostatecznie udało nam się osiągnąć czas poniżej 15 minut, co było czasem zbliżonym do tego, ile zajmowało proste przeczytanie taśmy z kodem. Byliśmy zatem całkiem zadowoleni.

## Programiści kontra ludzie

Nie zostaliśmy programistami, ponieważ tak bardzo lubimy pracować z innymi ludźmi. Z reguły uważaemy, że relacje interpersonalne są zagmatwane i nieprzewidywalne. Lubimy czyste i przewidywalne zachowania maszyn, które programujemy. Najszczęśliwi jesteśmy wtedy, gdy możemy zamknąć się sami w pokoju i skoncentrować nad jakimś bardzo interesującym problemem.

Tak, to ogromna generalizacja, od której istnieje wiele wyjątków. Jest wielu programistów, którzy dobrze sobie radzą, współpracując z innymi ludźmi, i nawet świetnie się przy tym bawią. Jednak średnia naszej grupy pasuje raczej do przedstawionego na początku obrazu. My, programiści, lubimy delikatną deprywację sensoryczną i zanurzenie się w stanie *skupienia*.

## Programiści kontra pracodawcy

W latach siedemdziesiątych i osiemdziesiątych, gdy pracowałem jako programista w firmie Teradyne, stwierdziłem, że jestem *naprawdę* dobry w debugowaniu. Uwielbiałem takie wyzwania i rzucałem się na problemy pełen wigoru i entuzjazmu. Przede mną nie ukrył się żaden błąd!

Gdy udało mi się wyeliminować kolejny błąd, czułem się jak zwycięzca, jakbym zabił Jabberwocka! Szedłem wtedy do mojego szefa Kena Findera, w ręce trzymając Vorpala, i z pasją opowiadałem, jaki *interesujący* błąd znalazłem. Pewnego dnia Ken krzyknął na mnie: „Błędy nie są interesujące. Je trzeba po prostu usunąć!”.

Tego dnia czegoś się nauczyłem. Dobrze jest z pasją wykonywać swoje zadania, ale dobrze jest też mieć na oku cele ludzi, którzy Ci płacą.

Podstawowym zadaniem profesjonalnego programisty jest spełnianie wymagań przedstawionych przez pracodawcę. Oznacza to współpracę z menedżerami, analitykami biznesowymi, testerami i pozostałymi członkami zespołu, by *dogłębnie poznać* cele firmy i projektu. Nie oznacza to, że musisz zostać firmowym kujonem. Ważne jest jednak, żeby dobrze wiedzieć, dlaczego piszesz kod, który właśnie piszesz, i w jaki sposób zatrudniająca Cię firma na tym kodzie skorzysta.

Najgorszą rzeczą, jaką może zrobić profesjonalny programista, jest zagrzebanie się w grobowcu technologii, podczas gdy naokoło niego cała firma wali się i płonie. Twoim *zadaniem* jest utrzymanie firmy na powierzchni!

Oznacza to, że profesjonalny programista poświęca czas na poznanie swojej firmy i branży, w której ona działa. Rozmawia z klientami o używanym przez nich oprogramowaniu. Rozmawia z ludźmi z działu sprzedaży i marketingu o trapiących ich problemach. Rozmawia z ich menedżerami, chcąc poznać krótko- i długoterminowe cele zespołu.

Mówiąc krótko, profesjonalni programiści interesują się statkiem, którym przyszło im płynąć.

Jedyny raz, kiedy wyrzucono mnie z pracy związanej z programowaniem, miał miejsce w 1976 roku. W tym czasie pracowałem w firmie Outboard Marine Corp. Pomagałem w niej pisać system automatyzacji fabryki wykorzystujący komputery IBM System/7 do monitorowania dziesiątek maszyn do odlewu aluminium, które pracowały na hali.

Pod względem technicznym była to bardzo wymagająca, ale też wdzięczna praca. Architektura komputerów System/7 była fascynująca, a i sam system automatyzacji fabryki był bardzo ciekawy.

Mieliśmy też bardzo dobry zespół. Jego kierownik John był zarówno kompetentny, jak i zmotywowany, natomiast moi dwaj koledzy programiści — mili i pomocni. Pracowaliśmy w laboratorium specjalnie przygotowanym na potrzeby tego projektu. Nasz biznesowy partner bardzo zaangażował się z nami w prace laboratoryjne. Nasz menedżer Ralph był kompetentny, skoncentrowany i rzeczywiście miał władzę.

Wszystko powinno było działać doskonale. To ja byłem problemem. Bardzo entuzjastycznie podchodziłem do całego projektu i do używanej w nim technologii. Niestety, w podeszłym wieku 24 lat nie byłem w stanie zmusić się do zainteresowania samą firmą i jej wewnętrzną polityczną strukturą.

Pierwszy błąd popełniłem już pierwszego dnia. Pojawiłem się w pracy bez krawata. Miałem krawat w czasie rozmowy kwalifikacyjnej i widziałem wtedy, że wszyscy nosili krawaty, ale jakoś nie wyciągnąłem odpowiednich wniosków. Pierwszego dnia Ralph podszedł do mnie i po prostu powiedział: „W tej firmie nosimy krawaty”.

Nie mogę nawet wyrazić, jak bardzo tego nie znośalem. Męczyło mnie to straszliwie. Codziennie nosiłem krawat i nienawidziłem tego. Ale dlaczego? Przecież wiedziałem, w co się pakuję. Znałem konwencje panujące w tej firmie. Dlaczego miałbym się denerwować? Tylko dlatego, że byłem samolubnym, narcystycznym małym palantem.

Po prostu nie mogłem dotrzeć do pracy na czas i sądziłem, że nie ma to znaczenia. W końcu zrobiłem „dobrą robotę”. To prawda, naprawdę doskonale wywiązywałem się z zadania polegającego na pisaniu programów. Mogę powiedzieć, że byłem najlepszym programistą w zespole. Umiałem pisać kod szybciej i lepiej niż pozostali koledzy. Byłem w stanie szybciej rozwiązywać różne problemy. *Wiedziałem, że jestem dla firmy wartościowy, i dlatego nie liczyły się dla mnie terminy i czas.*

Decyzja o zwolnieniu mnie zapadła pewnego dnia, gdy nie zjawiłem się na czas podczas prezentacji kolejnego kamienia milowego. Najwyraźniej John powiedział nam wszystkim, że w przyszły poniedziałek chce zobaczyć prezentację działających funkcji systemu. Jestem pewien, że o tym wiedziałem, ale takie terminy nie były dla mnie szczególnie istotne.

System nadal był aktywnie rozwijany i nie został wprowadzony do produkcji. Nie było powodu, żeby działał, skoro nikogo nie było w laboratorium. Byłem chyba ostatnią osobą wychodzącą w piątek do domu i najwyraźniej pozostawiłem system w stanie nienadającym się do użycia. Fakt, że w poniedziałek będzie tak ważny, po prostu wyleciał mi z głowy.

W poniedziałek spóźniłem się godzinę i zobaczyłem, jak wszyscy stoją zebrani nad niedziałającym systemem. John zapytał mnie: „Bob, dlaczego system dzisiaj nie działa?”. Odpowiedziałem: „Nie mam pojęcia”. Usiadłem i zacząłem szukać przyczyny. Nadal nic mi nie świtało na temat poniedziałkowej prezentacji, ale z mowy ciała wszystkich zebranych wnioskowałem, że coś jest mocno nie w porządku. Wtedy John podszedł do mnie i szepnął mi do ucha: „A co by było, gdyby Stenberg postanowił nas odwiedzić?”. A potem odszedł zniesmaczony.

Stenberg był wiceprezesem firmy odpowiedzialnym za automatyzację. Dzisiaj cieszyłby się tytułem CIO. Zadane pytanie nie miało dla mnie żadnego sensu. „No i co?” — pomyślałem. — „Przecież system nie jest jeszcze w produkcji, co za różnica?».

Później tego dnia dostałem pierwszy list z ostrzeżeniem. Informował mnie, że muszę zmienić swoje nastawienie albo „konieczne będzie szybkie wypowiedzenie”. Byłem naprawdę przerażony!

Trochę czasu zajęło mi przeanalizowanie swojego zachowania i uświadomienie sobie, co właściwie robiłem źle. Rozmawiałem na ten temat z Johnem i Ralphem. Byłem zdeterminowany, żeby zmienić siebie i swój stosunek do pracy.

I udało mi się! Przestałem się spóźniać. Zacząłem zwracać uwagę na wewnętrzną politykę firmy. Zacząłem rozumieć, dlaczego John tak bardzo obawiał się Stenberga. W końcu zobaczyłem, w jak trudnej sytuacji go postawiłem, unieruchamiając cały system przed feralnym poniedziałkiem.

Niestety, to wszystko było za mało i za późno. Kości zostały już rzucone. Miesiąc później dostałem drugi list z ostrzeżeniem za trywialny błąd, jaki mi się przytrafił. Już wtedy powiniensem był zauważyc, że te listy są tylko formalnością, a decyzja o zwolnieniu mnie została już podjęta. Ale byłem zdeterminowany, żeby ratować sytuację, dlatego zacząłem pracować jeszcze ciężej.

Spotkanie, na którym zostałem zwolniony, miało miejsce kilka tygodni później.

Poszedłem do domu do mojej 22-letniej, ciężarnej żony i musiałem jej powiedzieć, że straciłem pracę. Nie jest to doświadczenie, które chciałbym kiedykolwiek powtórzyć.

## **Programiści kontra programiści**

Programiści często mają problemy ze ścisłą współpracą z innymi programistami. Prowadzi to do naprawdę poważnych problemów.

### **Własność kodu**

Jednym z najgorszych symptomów dysfunkcyjnego zespołu jest sytuacja, w której każdy programista tworzy mur otaczający *jego* kod i odmawia pozostałym praw do zmian. Bywałem w miejscach, w których programiści nie pozwalali innym nawet *zobaczyć* swojego kodu. To prosta recepta na katastrofę.

Pewnego razu pracowałem jako konsultant dla firmy tworzącej drukarki najwyższej klasy. Takie maszyny składają się z wielu różnych elementów, takich jak podajniki, drukarki, sztablarki, zszywarki, obcinarki itp. Firma inaczej wartościowała każdy z tych elementów. Podajniki były ważniejsze od sztablarek, ale nic nie było ważniejsze od samych drukarek.

Każdy programista pracował nad *swoim* urządzeniem. Jedna osoba pisała kod podajnika, inna zajmowała się kodem przeznaczonym dla sztablarki. Każdy z nich zazdrośnie strzegł swoich technologii i uniemożliwiał wprowadzanie zmian do swojego kodu. Polityczne znaczenie, jakie mieli poszczególni programiści, było związane bezpośrednio z wartością przypisywaną przez firmę poszczególnym urządzeniom. Programista pracujący nad samą drukarką był nie do ruszenia.

To była prawdziwa katastrofa dla całej technologii drukowania. Jako konsultant byłem w stanie stwierdzić, że cała masa kodu była duplikowana, a interfejsy łączące poszczególne moduły były dokumentnie wypaczone. Mimo tego żaden z moich argumentów nie był w stanie przekonać programistów (i samej firmy) do zmiany sposobu pracy. W końcu ich wypłaty były powiązane z wagą obsługiwanych przez nich urządzeń.

### **Własność kolektywna**

Zdecydowanie lepiej jest, gdy zostaną zburzone mury własności kodu, a sam kod stanie się własnością całego zespołu. Wolę współpracować z zespołami, w których każdy programista może przejrzeć dowolny moduł i wprowadzić do niego takie zmiany, jakie uzna za stosowne. Chcę, żeby kod był własnością *zespołu*, a nie jego członków.

Profesjonalni programiści nie zabraniają innym pracy nad swoim kodem. Nie budują naokoło niego murów. Starają się raczej współpracować ze sobą nad wspólnym rozwijaniem systemu. Uczę się nawzajem, pracując wspólnie nad różnymi częściami systemu.

### **Praca w parach**

Wielu programistów nie lubi pracy w parach. Osobiście uważam, że to dziwne, ponieważ większość programistów *pracuje* w parach w sytuacjach awaryjnych. Dlaczego?

---

Najwyraźniej jest to najefektywniejszy sposób rozwiązywania problemów. Sprawdza się tutaj stare powiedzenie: co dwie głowy, to nie jedna. Jeżeli jednak praca w parach jest najsłuszniejszą przy rozwiązywaniu problemów w sytuacjach awaryjnych, to dlaczego nie jest najsłuszniejszym sposobem rozwiązywania problemów w ogóle?

Nie mam zamiaru cytować tutaj różnych badań, choć istnieją takie, które warto byłoby zacytować. Nie mam zamiaru opowiadać żadnych anegdot, choć istnieją takie, które warto byłoby opowiedzieć. Nie będę nawet opowiadał o tym, jak bardzo warto jest programować w parach. Powiem tylko tyle, że *profesjonalisci pracują w parach*. Dlaczego? Ponieważ w przypadku przynajmniej części problemów jest to najsłuszniejszy sposób ich rozwiązywania. Ale to niejedyny powód.

Profesjonalisci pracują w parach również dlatego, że jest to najlepszy sposób dzielenia się wiedzą. Profesjonalisci nie tworzą zamkniętych ogrodów wiedzy, ale poznają różne części systemu, pracując w parach. Wiedzą doskonale, że choć każdy członek zespołu ma w nim swoją pozycję, to jednak każdy powinien być w stanie natychmiast zająć pozycję innego członka.

Profesjonalisci pracują w parach, ponieważ jest to najlepszy sposób na dokonanie inspekcji kodu. Żaden system nie powinien zawierać kodu, który nie został przejrzany przez innych programistów. Istnieje wiele sposobów na przeprowadzenie inspekcji kodu, a większość z nich jest przerzążająco nieefektywna. Najefektywniejszą i najsłuszniejszą metodą inspekcji kodu jest współpraca przy jego pisaniu.

## Móźdżki

Pewnego ranka 2000 roku, w samym środku szaleństwa dot comów, jechałem pociągiem w Chicago. Wyszedłem z wagonu na peron, gdzie zaatakował mnie wielki billboard wiszący nad drzwiami wyjściowymi. Reklamował on doskonale znaną firmę software'ową, która zatrudniała programistów. Przeczytałem na nim „Przyjdź i zetrzyj swój mózg z najlepszymi”.

Od razu uderzył mnie poziom głupoty tak sformułowanego hasła. Biedni, niemający zielonego pojęcia ludzie z działu reklamy próbowali przemawiać do populacji technicznych, inteligentnych i mądrych programistów. To przecież jest rodzaj ludzi nie najlepiej znoszących jakiekolwiek przejawy głupoty. Reklamodawcy próbowali przywołać obraz wiedzy współdzielonej z innymi inteligentnymi osobami. Niestety, odwołali się do tej części mózgu — mózgów, która zajmuje się kontrolowaniem mięśni i nie ma nic wspólnego z inteligencją. A zatem ludzie, do których kierowana była ta reklama, śmiały się z powodu tak idiotycznego błędu.

W billboardzie zaintrygowało mnie jednak coś innego. Sprawił on, że zacząłem wyobrażać sobie grupę ludzi trących swoje mózgi. Jak wiadomo, mózg znajduje się w tylnej części mózgu, a zatem chcąc pocierać mózgi, trzeba odwrócić się do siebie plecami.

Wyobraziłem sobie zespół programistów pracujących w boksach, siedzących w kątach plecami do siebie, patrzących na ekrany, ze słuchawkami na uszach. *W ten sposób* właśnie ściera się mózdkie. Ale nie tak wygląda zespół.

Profesjonalisci pracują *razem*. Nie można ze sobą pracować, siedząc w kątach i ze słuchawkami na uszach. Chciałbym, żebyście siedzieli wokół stołu *twarzą do siebie*. Chcę, żebyście mogli pod słuchać pełne frustracji mamroтанie kolegi. Chcę widzieć spontaniczną komunikację, zarówno verbalną, jak i niewerbalną. Chcę, żebyście komunikowali się jako jedna całość.

Możesz sądzić, że pracujesz lepiej, siedząc samotnie. To może być prawda, ale nie musi to znaczyć, że *zespół* pracuje lepiej w czasie, gdy Ty pracujesz samotnie. Poza tym raczej mało prawdopodobne jest, że samotnie *będziesz* w stanie rzeczywiście pracować lepiej.

W pewnych okolicznościach samotna praca jest najwłaściwszym rozwiązaniem. W pewnych okolicznościach po prostu musisz dłużej i intensywnie rozmyślać nad pewnym problemem. Niekiedy zadanie jest tak trywialne, że praca z inną osobą oznaczałaby tylko stratę jej czasu. Ale ogólnie znacznie lepiej jest ściśle współpracować z innymi i łączyć się w pary przez większą część czasu.

## Wnioski

Być może programowaniem nie zainteresowaliśmy się po to, żeby pracować z innymi ludźmi. Niestety, mamy pecha. Programowanie oznacza *nieustanną współpracę z innymi*. Musimy współpracować ze sobą, a także z całą naszą firmą.

Wiem, wiem. Czy nie byłoby lepiej, gdyby po prostu zamknęli nas w pokoju z sześcioma wielkimi ekranami, szybkim łączem, baterią superszybkich procesorów, nielimitowaną pamięcią RAM i miejscem na dysku oraz nieskończonym zapasem dietetycznej koli i pikantnych chipsów? Przykro mi, ale tego się nie da zrobić. Jeżeli naprawdę chcemy całe dnie spędzać na programowaniu, to musimy nauczyć się rozmawiać z — ludźmi<sup>1</sup>.

---

<sup>1</sup> Odniesienie do ostatnich słów z filmu *Zielona pożywka*.

---

# 13 ZESPOŁY I PROJEKTY

---



Co zrobić, jeżeli musisz pracować nad wieloma małymi projektami? Jak przydzielać te projekty programistom? A co w przypadku, gdy musisz się zająć tylko jednym gigantycznym projektem?

## Można to zmiksować?

Przez lata pracowałem jako konsultant w wielu bankach i firmach ubezpieczeniowych. Jedyną rzeczą wspólną dla wszystkich tych firm jest dziwaczny sposób dzielenia projektów.

W banku projekt jest często względnie niewielką pracą, wymagającą udziału jednego programisty lub dwóch przez kilka tygodni. Do takiego projektu zwykle przydzielany jest menedżer, który jednocześnie zajmuje się innymi projektami. Dorzuca się do tego analityka biznesowego, który definiuje wymagania w kilku innych projektach. Poza tym do zespołu dołączanych jest kilku programistów pracujących również w innych projektach. Na koniec dodaje się testera lub dwóch, którzy również pracują nad kilkoma projektami.

Już widzisz, o co chodzi? Projekt jest zbyt mały, żeby pracujący nad nim ludzie mogli poświęcić się mu w pełnym wymiarze. Każdy pracuje nad projektem w 50 albo nawet 25%.

A teraz ważna zasada: nie istnieje coś takiego jak pół osoby.

Bezsensem jest nakazywanie programiście przeznaczenia połowy jego czasu na projekt A, a pozostałą część na projekt B, szczególnie wtedy, gdy oba te projekty mają różnych menedżerów, analityków biznesowych, programistów i testerów. W jakim świecie można takiego potworka nazywać zespołem? To nie zespół, tylko efekt pracy blendera.

## Zespół zespolony

Na sformowanie się zespołu potrzeba czasu. Członkowie zespołu zaczynają tworzyć między sobą związki. Uczę się współpracy, poznają swoje silne i słabe strony. W ten sposób zespół zaczyna się *spajać*.

W zintegrowanych zespołach jest coś prawdziwie magicznego. Takie zespoły mogą działać cuda. Ich członkowie przewidują wzajemnie swoje działania, wspierają się i wymagają od siebie tego, co najlepsze. Takie zespoły naprawdę sprawiają, że coś się dzieje.

Zespolony zespół składa się zwykle z mniej więcej tuzina osób. Może być ich nawet dwadzieścia albo mogą być jedynie trzy, ale zazwyczaj liczba członków zespołu bliska jest dwunastu. Zespół powinien składać się z programistów, testerów i analityków. I powinien mieć menedżera projektu.

Stosunek liczby programistów do testerów i analityków może się bardzo różnić, ale dobrą proporcją jest 2:1. Oznacza to, że dobrze zespolony zespół może się składać z siedmiu programistów, dwóch testerów, dwóch analityków i jednego menedżera projektu.

Analityk definiuje wymagania i na ich podstawie pisze zautomatyzowane testy akceptacyjne. Testerzy również zajmują się tworzeniem zautomatyzowanych testów akceptacyjnych.

Różnicą między nimi jest inna perspektywa. Wszyscy zapisują wymagania, ale analityk koncentruje się na wartościach biznesowych, natomiast tester na poprawności. Analitycy tworzą optymistyczne ścieżki w testach, a testerzy zastanawiają się nad tym, co może pójść źle, i piszą testy obejmujące błędy i przypadki brzegowe.

Menedżer projektu śledzi postępy zespołu i upewnia się, że zespół zna wszystkie priorytety i harmonogramy.

Jeden z członków zespołu może przejąć rolę trenera lub mistrza, którego odpowiedzialnością jest ochrona procesów i metod stosowanych przez zespół. Taka osoba ma być sumieniem zespołu, jeżeli ten chciałby porzucić istniejące procesy w wyniku zwiększającej się presji.

## Fermentacja

Taki zespół potrzebuje czasu na zapoznanie się z wewnętrznymi różnicami, pogodzenie się z nimi i rzeczywiste spojenie. Ten proces może potrwać sześć miesięcy, a nawet cały rok. Ale gdy się zakończy, dzieją się cuda. Zespolony zespół zaczyna wspólnie planować, rozwiązywać problemy, wspólnie stawia czoła przeciwnościom i *praca posuwa się naprzód*.

Gdy to się w końcu zdarzy, przestępstwem byłoby rozbijać ten układ tylko dlatego, że kończy się projekt. Znacznie lepiej jest utrzymać niezmieniony skład zespołu i poszukać mu odpowiedniego projektu.

## Co było pierwsze: zespół czy projekt?

Banki i firmy ubezpieczeniowe próbowały tworzyć zespoły na potrzeby projektów. To jednak jest nierozsądne podejście. Zespołu nie da się szybko zintegrować. Poszczególne osoby uczestniczą w projekcie przez krótki czas, poświęcając mu tylko część swojej uwagi, i dlatego nigdy nie uczą się, jak ze sobą współpracować.

Profesjonalne organizacje rozwojowe przypisują projekty istniejącym już zintegrowanym zespołom, ale nie formują zespołów na potrzeby projektów. Zespolony zespół może przyjąć jednocześnie kilka różnych projektów i podzielić prace zgodnie z własnym uznaniem i umiejętnościami. Zespolony zespół na pewno sobie z takimi projektami poradzi.

## Ale jak tego dokonać?

Każdy zespół ma swoje tempo<sup>1</sup>, które jest po prostu ilością pracy, jaką może on wykonać w określonym czasie. Niektóre zespoły mierzą swoje tempo w *punktach* na tydzień, a punkty są jednostkami złożoności. Poszczególne funkcje aktualnie rozwijanego projektu

<sup>1</sup> [PPP2002], s. 20–22; [COHN2006], tu szukaj w indeksie wielu świetnych odniesień do tempa prac.

są rozbijane na części i każdej z nich jest przypisywany szacunek wyrażony w punktach. Następnie zespół mierzy, ile punktów uda się przerobić w ciągu tygodnia.

Tempo jest miarą statystyczną. Zespół może w jednym tygodniu zrobić 38 punktów, w kolejnym 42, a w następnym 25. Z czasem uzyska się pewną wartość średnią.

Menedżerowie mogą wyznaczać priorytety w każdym projekcie przydzielanym zespołowi. Na przykład jeżeli średnie tempo zespołu wynosi 50, a pracuje on nad trzema projektami, to menedżer może poprosić zespół o podzielenie nakładu pracy na 15, 15 i 20.

Zaletą tego rozwiązania, oprócz tego, że nad projektami pracuje zespołowy zespół, jest to, że w przypadkach awaryjnych można powiedzieć: „Projekt B jest w tarapatach. Przez następne trzy tygodnie poświęcacie mu 100% swojej uwagi”.

Taka zmiana priorytetów jest praktycznie niemożliwa w przypadku zespołów, które wyszły z blendra. Natomiast zespoły zintegrowane, które pracują nad dwoma lub trzema projektami jednocześnie, mogą tego dokonać bez żadnego kłopotu.

## **Dylemat właściciela projektu**

Jednym z częstych argumentów przeciwko proponowanej przeze mnie metodzie jest to, że właściciel projektu traci część swojej władzy i bezpieczeństwa. Właściciele projektów, którzy mają zespół przypisany do danego projektu, mogą liczyć na swój zespół. Wiedzą o tym, ponieważ formowanie i rozbijanie zespołu jest operacją bardzo kosztowną, dlatego firma nie odbierze im zespołu z błahych powodów.

Jednak jeżeli projekt zostanie przekazany zgranemu zespołowi, a zespół ten będzie się zajmował kilkoma projektami jednocześnie, to firma nie będzie miała oporów przed zmianą priorytetów. To może sprawiać, że właściciel projektu nie będzie pewny swojej przyszłości. Zasoby, z których korzysta, mogą zostać mu nagle odebrane.

Osobiście wolę jednak tę drugą sytuację. Nie powinno się wiązać firmie rąk z powodu sztucznej trudności formowania i rozwiązywania zespołów. Jeżeli firma zdecyduje, że jeden projekt jest ważniejszy od drugiego, to powinna mieć możliwość szybkiego transferu zasobów. Zadaniem właściciela projektu jest odpowiednie reprezentowanie go.

## **Wnioski**

Zespoły tworzy się trudniej niż projekty. Z tego powodu lepiej jest tworzyć trwałe zespoły, które razem pracują nad kolejnymi projektami i mogą obsługiwać w danym czasie kilka projektów. Celem formowania zespołu jest przekazanie mu wystarczającej ilości czasu na zintegrowanie się. Później taki zespół nie powinien być rozbijany, ponieważ jest on gotowy na realizację wielu projektów.

## **Bibliografia**

[PPP2002]: Robert C. Martin, *Agile Software Development. Principles, Patterns, and Practices*, Upper Saddle River, NJ: Prentice Hall, 2003.

[COHN2006]: Mike Cohn, *Agile Estimating and Planning*, Upper Saddle River, NJ: Prentice Hall, 2006.



---

# **NAUCZANIE, TERMINOWANIE I MISTRZOSTWO**

---



Niezmiennie byłem rozczarowany poziomem absolwentów studiów informatycznych. Nie chodzi o to, że absolwenci nie są utalentowani i inteligentni, ale o to, że nie nauczono ich, o co właściwie chodzi w programowaniu.

## **Stopnie niepowodzenia**

Prowadziłem kiedyś rozmowę kwalifikacyjną z młodą kobietą studującą informatykę na wielkim uniwersytecie. Starała się o letnie stanowisko stażysty. Poprosiłem ją o napisanie dla mnie prostego kodu, a ona odpowiedziała: „Ale ja tak naprawdę nie piszę kodu”.

Proszę przeczytać poprzedni akapit ponownie, a potem od razu przejść do następnego.

Zapytałem ją, które zajęcia z programowania wybrała, skoro chce uzyskać tytuł magistra informatyki. Powiedziała, że w swoim planie nie ma żadnych tego rodzaju zajęć.

*Może zechcesz zacząć czytać ten rozdział od początku, aby się upewnić, że nie znajdujesz się w jakimś alternatywnym wszechświecie albo w innym koszmarze.*

W tym momencie można zadać sobie pytanie: jak student studiów magisterskich z informatyki może unikać zajęć z programowania? Wtedy zadałem sobie dokładnie takie pytanie. Do dzisiaj nie znam na nie odpowiedzi.

Oczywiście to najpoważniejsze z całej serii roczarowań, jakie spotkały mnie w czasie rozmów z absolwentami uniwersytetów. Oczywiście nie wszyscy absolwenci są tak roczarowujący, daleki jestem od tego stwierdzenia! Zauważałem jednak, że osoby niesprawiające roczarowań mają pewną wspólną cechę: niemal wszystkie *nauczły się programować* jeszcze przed rozpoczęciem nauki na uniwersytecie i uczyły się tego nadal mimo opuszczenia murów uczelni.

Proszę mnie źle nie zrozumieć. Uważam, że na uniwersytecie można uzyskać doskonałe wykształcenie. Uważam jednak też, że można prześlizgnąć się przez ten system, uzyskując dyplom i nic ponad to.

Jest jeszcze jeden problem. Nawet najlepsze uniwersytety nie przygotowują swoich absolwentów na to, co spotka ich w naszej branży. Nie chcę tu nikogo oskarżać, ponieważ jest to stan obejmujący niemal wszystkie kierunki studiów. To, czego nauczysz się w szkole i czego doświadczysz w pracy, to dwie zupełnie różne rzeczy.

## **Nauczanie**

Jak można się nauczyć programować? Opowiem Ci historyjkę o byciu nauczonym.

### **Digi-Comp I, mój pierwszy komputer**

W 1964 roku na moje dwunaste urodziny mama dała mi plastikowy komputer. Nazywał się on Digi-Comp I<sup>1</sup>. Miał trzy plastikowe przełączniki i sześć plastikowych bramek I. Można było połączyć wyjścia przełączników z wejściami bramek, a nawet połączyć wyjścia bramek z wejściami przełączników. W skrócie: pozwalał on tworzyć trójbitową, nieskończoną maszynę stanów.

---

<sup>1</sup> Jest wiele stron internetowych udostępniających symulatory tego małego stymulującego komputera.

W zestawie był też podręcznik prezentujący kilka programów, które można było uruchamiać. Maszynę programowało się, wsuwając małe rurki (kawałki rurek do napojów) na niewielkie bolce wystające z przełączników. Podręcznik informował, gdzie dokładnie należy umieścić poszczególne rurki. Nie wspominał jednak o tym, co te rurki właściwie *robiły*. Było to dla mnie strasznie frustrujące!

Patrzyłem na maszynę całymi godzinami, próbując dowiedzieć się, jak działa na poziomie podstawowym. Nie byłem jednak w stanie wykombinować, jak zmusić ją do robienia tego, czego bym od niej chciał. Na ostatniej stronie podręcznika znalazła się informacja, że za dolara prześlą mi podręcznik z informacją, jak programować maszynę<sup>2</sup>.

Wysłałem zatem dolara i czekałem z niecierpliwością godną dwunastolatka. W dniu, gdy w końcu został dostarczony, po prostu go pochłonąłem. Była to prosta rozprawka na temat algebry Boole'a opisująca podstawowe formy równań, prawa łączności i rozdzielności oraz twierdzenia De Morgana. Podręcznik uczył, jak wyrazić dany problem w postaci sekwencji równań Boole'a. Mówił też o tym, jak zredukować te równania tak, żeby pasowały do sześciu bramek *I*.

Napisałem zatem swój pierwszy program. Nadal pamiętam, jak go nazwałem: Komputerowa brama Pana Pattersona. Napisałem równania, zredukowałem je i odwzorowałem na rurkach i bolcach maszyny. *I to działało!*

Napisanie tych trzech słów sprawiło, że przeszedł mnie dreszcz. Taki sam, jaki przeszedł tego dwunastolatka prawie pół wieku temu. Połknąłem haczyk. Moje życie już nigdy nie będzie takie jak dawniej.

Pamiętasz moment, gdy Twój pierwszy program w końcu zadziałał? Czy to zmieniło Twoje życie, umieszczając je na kursie, z którego nie było już odwrotu?

Sam niczego tutaj nie rozgryzłem. Zostałem tego *nauczony*. Jacyś bardzo mili i bardzo zdolni ludzie (wobec których mam ogromny dług wdzięczności) poświęcili czas na napisanie rozprawy o algebrze Boole'a, która była zrozumiała dla dwunastolatka. Połączyli matematyczną teorię z pragmatyką prostego, plastikowego komputera i pomogli mi zmusić ten komputer do wykonywania zaplanowanych przeze mnie rzeczy.

Właśnie wziąłem do ręki kopię tego niezwykłego podręcznika. Trzymam go w specjalnym foliowym woreczku, a mimo to czas odcisał na nim swoje piętno. Kartki są już pożółkłe i stały się kruche. Niemniej jednak siła zapisanych w nim słów nadal jest ogromna. Elegancki opis algebry Boole'a zajmuje trzy krótkie strony. Omówienie poszczególnych równań użytych w pierwotnych programach nadal jest bardzo zajmujące. To była prawdziwie mistrzowska praca. Praca, która zmieniła życie przynajmniej jednego młodego mężczyzny. Obawiam się jednak, że nigdy nie poznam nazwisk jej autorów.

---

<sup>2</sup> Nadal mam ten podręcznik. W mojej bibliotece zajmuje honorowe miejsce.

## ECP-18 w szkole średniej

W wieku lat piętnastu trafiłem właśnie do szkoły średniej. Bardzo lubiłem wtedy przesiadywać w dziale matematyki. (A to zaskoczenie!) Pewnego dnia wtoczyli do niego maszynę wielkości piły stołowej. Był to ECP-18 — komputer edukacyjny zbudowany specjalnie dla szkół średnich. W naszej szkole miała się odbywać dwutygodniowa prezentacja.

Stałem sobie z boku w czasie, gdy nauczyciele rozmawiali z technikami. Ta maszyna ma 15-bitowe słowo (*co to jest to słowo?*) i 1024-słowową pamięć bębnową. (Wiedziałem, co to jest pamięć bębnowa, ale znałem tylko ogólną koncepcję).

Gdy w końcu włączyli maszynę, wydała z siebie wysoki dźwięk przypominający start odrzutowca. Sądziłem, że właśnie rozpędza się bęben. Gdy już się rozpędził, całość działała względnie cicho.

Ta maszyna była *wspaniała*. Było to właściwie zwyczajne biurko, na którym znajdował się niesamowity panel sterowania, przypominający mostek pancernika. Na panelu świeciły się całe rzędy lampek, a na naciśnięcie czekało wiele różnych klawiszy. Siedzenie przy tym biurku było jak zasiadanie w fotelu kapitana Kirka.

Przyglądając się technikom obsługującym maszynę, zauważylem, że po naciśnięciu klawisza zapalał się on, a ponowne naciśnięcie powodowało gaszenie światełka. Zauważylem też, że naciskali pewne specjalne klawisze, które były opisane *deposit* i *run*.

Klawisze w każdym wierszu zostały podzielone na pięć grup po trzy. Mój Digi-Comp również był trzybitowy, więc mogłem odczytywać wyrażone dwójkowo cyfry ósemkowe. Zrozumienie, że chodzi tu o pięć cyfr ósemkowych, nie zajęło mi dużo czasu.

Słyszałem, że technicy mamroczą coś do siebie, wciskając klawisze. Gdy wpisywali liczby 1, 5, 2, 0, 4 do bufora pamięci, mówili do siebie: „Zapisz w 204”. Wciskali klawisze 1, 0, 2, 1, 3 i mamrotali: „Ładuj 213 do akumulatora”. Jeden z rzędów klawiszy był nazywany *akumulatorem*!

Po dziesięciu minutach dla mojego piętnastoletniego umysłu stało się jasne, że liczba 15 oznaczała *zapisz*, a liczba 10 — *załaduj*, że to właśnie do akumulatora można było zapisywać albo ładować wartości, a pozostałe liczby oznaczały jedno z 1024 słów na bębnie. (A więc to *tym* jest słowo!)

Bit za bitem (tutaj całkiem dosłownie) mój chętny umysł zauważał coraz więcej kodów instrukcji i pojęć. W momencie, gdy technicy sobie poszli, znałem już podstawy działania maszyny.

Tego popołudnia, w czasie na samodzielnej naukę, zakradłem się do laboratorium matematycznego i zacząłem bawić się komputerem. Już dużo wcześniej nauczyłem się,

---

że lepiej prosić o wybaczanie niż o pozwolenie! Napisałem mały program, który mnożył wartość akumulatora razy dwa i dodawał do niego jeden. Wpisałem do akumulatora liczbę 5, uruchomiłem program i zobaczyłem w akumulatorze wartość 13<sub>8</sub>! Zadziałało!

Wpisałem kilka innych prostych programów tego rodzaju i wszystkie działały zgodnie z planem. Byłem panem wszechświata!

Kilka dni później uświadomiłem sobie, jak głupi byłem i ile miałem szczęścia. Znalazłem w laboratorium instrukcję leżącą na stole. Podawała ona wszystkie polecenia i kody operacyjne, w tym i takie, których nie poznałem, podglądając techników. Byłem szczęśliwy, że prawidłowo zinterpretowałem te polecenia, które już znałem, ale pozostałe wprawiły mnie w ekscytację. Jedną z nowych instrukcji było HTML. Jak się okazało, kodem tej instrukcji zatrzymania były same zera. I całkiem przypadkiem na końcu każdego mojego programu umieszczałem słowo składające się z samych zer, tak żeby móc wpisać je do akumulatora i tym samym go wyczyścić. Pomyśl na instrukcję zatrzymującą w ogóle nie przyszedł mi do głowy. Sądziłem, że program po prostu zatrzyma się, gdy się skończy.

Pamiętam, że pewnego dnia siedziałem w laboratorium i obserwowałem, jak jeden z nauczycieli próbował uruchomić jakiś program. Chciał wprowadzić dwie liczby dziesiętne za pomocą podłączonego dalekopisu i wypisać ich sumę. Każdy, kto próbował napisać taki program w języku maszynowym minikomputera, wie, że nie jest to całkiem trywialne. Trzeba odczytać znaki, zmienić je w cyfry, te z kolei przekształcić do postaci binarnej, dodać je, przekształcić w postać dziesiętną i ostatecznie zapisać jako znaki. I uwierz mi, całość jest jeszcze gorsza, gdy wprowadzasz taki program w postaci binarnej za pomocą panelu komputera!

Patrzyłem, jak wprowadzał do swojego programu instrukcję zatrzymania i uruchamiał go, a ten działał aż do wyznaczonego instrukcją miejsca. (Rany! To dopiero pomysł!) Ten prymitywny punkt wstrzymania pozwalał mu sprawdzić zawartość rejestrów i skontrolować to, czego program dokonał do tej pory. Pamiętam, że mamrotałem do siebie: „Wow, ależ to szybkie!”. Cóż, mam dla niego ciekawe wiadomości!

Nie mam pojęcia, jakiego algorytmu używał. Ten rodzaj programowania to była dla mnie nadal czarna magia. Nauczyciel nigdy się do mnie nie odezwał w czasie, gdy patrzyłem mu przez ramię. Co więcej, *nikt* nie rozmawiał ze mną na temat komputera. Sądzę, że uznawali mnie za latający po laboratorium niczym śmieszny drobny szczegół, który należy ignorować. Wystarczy powiedzieć, że ani uczni, ani nauczyciele nie rozwinęli szczególnie swoich zdolności towarzyskich.

W końcu udało mu się uruchomić program. Obserwowanie go było niesamowite. Powoli wprowadzał dwie liczby, ponieważ mimo zachwytów komputer wcale *nie* był szybki (pomyśl tylko o odczytywaniu kolejnych słów z obracającego się bębna, w 1967 roku). Gdy nacisnął klawisz *Return* po wprowadzeniu drugiej liczby, komputer zaczynał szaleńczo błyskać,

a po chwili wypisywał już wynik obliczeń. Zajmowało mu to mniej więcej po jednej cyfrze na sekundę. Wypisywał wszystkie cyfry z wyjątkiem ostatniej, ponownie szaleńczo błykał przez pięć sekund i wtedy pojawiała się ostatnia cyfra, a program się zatrzymywał.

Skąd brała się ta przerwa przed ostatnią cyfrą? Nigdy się tego nie dowiedziałem. Ale uświadomiło mi to, że sposób rozwiązyania problemu może mieć ogromny wpływ na użytkownika. Mimo że program generował prawidłową odpowiedź, wiedziałem, że *nadal* coś w nim nie działa dobrze.

Tak wyglądało nauczanie. Z pewnością nie był to wymarzony jego rodzaj. Byłoby miło, gdyby jeden z nauczycieli wziął mnie pod swoje skrzydła i zaczął ze mną pracować. Ale nie miało to znaczenia, ponieważ mogłem ich *obserwować* i uczyć się w straszliwym tempie.

## **Nauczanie niekonwencjonalne**

Zaprezentowałem te dwie historie, ponieważ opisują one dwa całkowicie różne rodzaje nauczania, z których żaden nie miał wiele wspólnego z tym, co kojarzy się ze słowem „nauczanie”. W pierwszym przypadku uczyłem się od autorów doskonale przygotowanego podręcznika. W drugim — obserwując osoby, które bardzo starały się mnie ignorować. W obu przypadkach zdobyta wiedza była głęboka i niezwykle ważna.

Oczywiście miałem też innych nauczycieli. Choćby miły sąsiad, który pracował w firmie Teletype i przyniósł mi do domu pudełko z 30 przełącznikami telefonicznymi, którymi mogłem się bawić. Powiem tylko: dajcie chłopakowi kilka przełączników i transformator z kolejki elektrycznej, a będzie próbował podbić świat!

Był też inny miły sąsiad — radiooperator, który pokazał mi, jak korzystać z multimetru (bardzo szybko go jednak zepsałem). Był też właściciel sklepu z materiałami biurowymi, który pozwalał mi przychodzić i „bawić się” jego drogim programowalnym kalkulatorem. Było też biuro firmy DEC, które pozwalało mi „bawić się” ich komputerami PDP-8 i PDP-10.

Był też wielki Jim Carlin, programista języka BAL, który ocalił mnie przed wyrzuceniem z pierwszej pracy związanej z programowaniem — uczył mnie debugować program w COBOL-u, który wykracał poza moje ówczesne możliwości. Nauczył mnie czytać zrzuty jądra i formatować kod sprytnie wstawianymi pustymi wierszami, rzędami gwiazdek i komentarzami. To on pierwszy popchnął mnie w kierunku dobrego rzemiosła. Żałuję, że nie mogłem mu pomóc rok później, gdy spadł na niego gniew naszego szefa.

Ale tak naprawdę to już wszystko. Na początku lat 70. nie było zbyt wielu starszych programistów. Wszędzie, gdzie pracowałem, to *ja byłem* tym starszym. Nie było nikogo, kto powiedziałby mi, czym naprawdę jest profesjonalne programowanie. Nie było wzorca, który nauczyłby mnie, jak należy się zachowywać i co należy cenić. Tego wszystkiego musiałem nauczyć się samodzielnie i zdecydowanie nie było to łatwe.

## Mocne ciosy

Jak wspominałem już wcześniej, w 1976 roku zostałem zwolniony z pracy przy automatyzacji fabryki. Mimo że pod względem technicznym byłem bardzo kompetentny, to jednak nie nauczyłem się jeszcze zwracać uwagi na firmę i jej cele. Niewiele znaczyły dla mnie wyznaczone terminy. Zapomniałem o wielkim poniedziałkowym pokazie, w piątek zostawiłem system w nieużywalnym stanie, a w poniedziałek spóźniłem się i wszyscy się na mnie wściekali.

Mój szef wysłał mi list z ostrzeżeniem mówiącym, że mam się natychmiast zmienić albo zostanę zwolniony. To był dla mnie poważny wstrząs. Przemyślałem swoje życie oraz karierę i zacząłem wprowadzać w zachowaniu daleko idące zmiany. Niestety, to wszystko było za mało i za późno. Wcześniej nabräałem już rozpedu w niewłaściwym kierunku i teraz szczegóły, które wcześniej nie miałyby znaczenia, urosły do ogromnych rozmiarów. Chociaż starałem się naprawdę mocno, ostatecznie i tak zostałem odeskortowany poza budynek.

Nie muszę chyba wspominać, że przyniesienie takich wiadomości ciężarnej żonie i dwuletniej córce nie jest niczym przyjemnym. Pozbierałem się jednak i do następnej pracy zabrałem kolejną ważną nauczkę. Pamiętałem o niej i o innych przez następnych 15 lat, dlatego stały się one podstawą mojej aktualnej kariery.

Ostatecznie przeżyłem i dalej się rozwijałem. Istnieją jednak znacznie lepsze metody. Lepiej dla mnie byłoby, gdybym miał prawdziwego mentora, który nauczyłby mnie odróżniać dobro od zła. Kogoś, kogo mógłbym obserwować podczas prac nad drobnymi zadaniami, kto sprawdzałby moją wczesną pracę i kierował nią. Kogoś, kto byłby dla mnie wzorem i uczył mnie właściwych reakcji i wartości. Sensei. Mistrza. Mentora.

## Terminowanie

Co robią lekarze? Sądzisz, że szpitale zatrudniają absolwentów szkół medycznych i już pierwszego dnia pracy kierują ich na sale operacyjne? Oczywiście, że nie.

W zawodzie lekarza wypracowano metody intensywnej nauki, związane z ustanowionym rytuałem i zanurzone w tradycji. Świat medyczny kontroluje uniwersytety i upewnia się, że ich absolwenci mają jak najlepsze wykształcenie. Na to wykształcenie niemal *po równo* składają się zajęcia w murach uczelni i działania kliniczne w szpitalach pod okiem profesjonalistów.

Po ukończeniu szkoły, ale jeszcze przed przyznaniem licencji nowi lekarze muszą spędzić cały rok na kontrolowanej praktyce i ćwiczeniach nazywanych stażem. W istocie jest to bardzo intensywne szkolenie przez pracę. Każdy stażysta jest otoczony nauczycielami i wzorcami do naśladowania.

Po zakończeniu stażu każda ze specjalizacji medycznych wymaga kolejnych trzech do pięciu lat nadzorowanych praktyk i ćwiczeń, które tym razem nazywane są rezydenturą. Rezydenci nabierają pewności siebie, przejmując coraz większą odpowiedzialność, choć nadal są obserwowani i nadzorowani przez starszych lekarzy.

Wiele specjalności wymaga jeszcze kolejnych lat współpracy, w czasie których uczniowie kontynuują specjalizowaną naukę i nadzorowane ćwiczenia.

Dopiero wtedy są gotowi, żeby przystąpić do egzaminów i otrzymać certyfikat.

Taki opis zawodu lekarza jest troszeczkę wyidealizowany i być może nie całkiem dokładny. Istotne jest jednak to, że gdy stawka jest wysoka, nie wysyła się absolwentów do działania, oczekując doskonałych rezultatów. Dlaczego zatem dokładnie tak postępujemy przy tworzeniu oprogramowania?

To prawda, że błędy w oprogramowaniu powodują względnie niewielką liczbę zgonów. Ale *powodują* poważne straty finansowe. Firmy tracą ogromne kwoty z powodu niedostatecznego wyszkolenia swoich programistów.

Z jakiegoś powodu branża oprogramowania wymyśliła sobie, że programiści stają się programistami zaraz po ukończeniu szkoły i od razu umieją programować. Naprawdę firmy bardzo często zatrudniają dzieci kończące szkoły, formując z nich „zespoły” i każą im tworzyć najważniejsze systemy. To przecież szaleństwo!

Nie robią tak malarze. Nie robią tak hydraulicy. Nie robią tak elektrycy. Nawet kucharze w fast foodach tak nie robią. Wydaje mi się, że firmy zatrudniające absolwentów informatyki powinny inwestować w ich wykształcenie nieco więcej niż McDonald inwestuje w swoich kelnerów.

Nie oszukujmy się, że to nie ma znaczenia. Ma, i to ogromne. Nasza cywilizacja jest napędzana oprogramowaniem. To oprogramowanie porusza i manipuluje informacją wpływającą na nasze życie. Oprogramowanie steruje silnikami naszych samochodów, ich skrzyniami biegów i hamulcami. Zarządza saldami naszych kont bankowych, wysyła nam rachunki i przyjmuje płatności. Oprogramowanie pierze nasze ubrania i informuje nas o aktualnej godzinie. To oprogramowanie wyświetla obrazy w telewizji, wysyła nam SMS-y, pozwala na rozmowy telefoniczne i bawi nas, gdy czujemy się znudzeni. Oprogramowanie jest wszędzie.

Skoro wpuszczamy programistów do każdego zakątku naszego życia, od tych najdrobniejszych po najistotniejsze, to sądzę, że pewien okres szkolenia i nadzorowanych praktyk byłby całkiem wskazany.

## Terminowanie programisty

Jak zatem *należałoby* w zawodzie programisty wprowadzać młodych absolwentów w szeregi profesjonalistów? Jakie kroki należy podejmować? Jakie wyzwania przed nimi stawiać? Zaczniemy rozważania od tyłu.

### Mistrzowie

Mistrzowie są programistami, którzy prowadzili już niejeden poważny projekt programistyczny. Zwykle mają już ponad 10 lat doświadczenia i pracowali nad różnymi rodzajami systemów, w różnych językach i systemach operacyjnych. Wiedzą, jak prowadzić i koordynować wiele zespołów, są sprawnymi projektantami i architektami, bez żadnego trudu są w stanie napisać dowolny kod. Oferowano im już stanowiska związane z zarządzaniem, ale odmówili albo po ich zaakceptowaniu szybko się wycofali, albo połączycyli nowe zadania z pierwotnymi pracami technicznymi. Nadal utrzymują swoje umiejętności techniczne, czytając, ucząc się, ćwicząc, pracując i *nauczając*. To właśnie takim mistrzom firmy będą przypisywać odpowiedzialność za techniczną stronę projektu. Pomyśl o „Scottym”.

### Czeladnicy

To programiści o odpowiednim wyszkoleniu, kompetencjach i chęciach. W tym okresie swojej kariery uczą się dobrze współpracować z zespołem i stają się liderami. Doskonale orientują się w aktualnych technologiach, ale zwykle brak im doświadczenia z różnorodnymi systemami. Zwykle znają jeden język, jeden system, jedną platformę, ale ciągle się uczą. Poziom doświadczenia może być bardzo różny, ale najczęściej jest to jakieś pięć lat. Na jednym końcu tej średniej mamy rodzących się mistrzów, na drugiej — niedawnych praktykantów.

Czeladnicy są nadzorowani przez mistrzów albo starszych czeladników. Młodym czeladnikom rzadko tylko pozwala się na jakąkolwiek autonomię. Ich praca zawsze jest ściśle nadzorowana. Ich kod jest przeglądany. Wraz z zyskiwanym doświadczeniem zwiększa się też ich autonomia. Nadzór przestaje być tak bezpośredni, a jego gorset jest powoli luzowany. Ostatecznie ogranicza się do krótkich recenzji.

### Praktykanci i stażyści

Absolwenci zaczynają swoją karierę jako praktykanci. Praktykanci nie mają żadnej autonomii i są ściśle nadzorowani przez czeladników. Początkowo nietrzymają żadnych zadań, a jedynie wspomagają czeladników w pracy. To powinien być dla nich czas bardzo intensywnego programowania w parach. To właśnie wtedy uczą się różnych metodologii. To właśnie wtedy tworzone są w nich podstawowe wartości.

Czeladnicy są nauczycielami. Upewniają się, że praktykanci znają zasady projektowania, wzorce projektowe, metodologie i rytuały. Czeladnicy uczą ich TDD, refaktoryzacji, szacowania itd. Podsuwają praktykantom różne książki i artykuły do czytania, nakazując wykonywanie ćwiczeń i sprawdzają ich postępy.

Praktyka powinna trwać około roku. Po tym czasie, jeżeli czeladnicy zechcą przyjąć praktykantów w swoje szeregi, powinni przekazać mistrzom swoje rekomendacje. Mistrzowie powinni sprawdzić praktykantów zarówno w bezpośredniej rozmowie, jak i przeglądając ich osiągnięcia. Jeżeli mistrzowie się zgodzą, to praktykant staje się czeladnikiem.

## Rzeczywistość

Tutaj również wszystko zostało wyidealizowane i pozostaje w sferze hipotez. Jeżeli jednak pozmieniać kilka nazw i przymknąć oko na pewne określenia, to cały opis nie będzie bardzo odległy od tego, czego byśmy *oczekiwali*. Absolwenci są nadzorowani przez młodych liderów zespołów, którzy z kolei są nadzorowani przez prowadzących projekty itd. Problem polega na tym, że w większości przypadków takie nadzorowanie nie ma *nic wspólnego z techniką!* W większości firm nie istnieje coś takiego jak nadzór techniczny. Programiści dostają podwyżki i ewentualne awanse, ponieważ... tak się postępuje z programistami.

Różnica między tym, co naprawdę robimy, a moim wyidealizowanym programem terminowania polega na koncentracji na nauczaniu, ćwiczeniach, nadzorze i recenzjach.

Różnicę stanowi pogląd, że wartości stanowiące o profesjonalizmie i techniczna doskonałość muszą być uczone, wpajane, pielęgnowane, pieszczone i hodowane. W naszym aktualnym podejściu brakuje nakazu dla starszych, żeby uczyli młodych.

## Rzemiosło

Teraz możemy już zdefiniować to słowo: *rzemiosło*. Czym ono właściwie jest? Chcąc je zrozumieć, musimy przyjrzeć się słowi *rzemieślnik*. Słowo to przyprowadzi na myśl kwalifikacje i jakość. I jeszcze doświadczenie, i kompetencję. Rzemieślnik jest kimś, kto pracuje szybko, ale bez pośpiechu, kto podaje rozsądne szacunki i dotrzymuje zobowiązań. Rzemieślnik wie, kiedy należy odmówić, ale z całych sił stara się tego unikać. Rzemieślnik jest profesjonalistą.

Rzemiosło jest *stanem umysłu* właściwym dla rzemieślników. Rzemiosło jest memem zawierającym w sobie wartości, metody, techniki, nastawienie i odpowiedzi.

Jak jednak rzemieślnicy przyjmują do siebie ten mem? Jak osiągają ten stan umysłu?

Mem rzemiosła jest przekazywany z jednej osoby na drugą. Starsi uczą go młodszych. Jest dzielony przez współpracujące osoby. Można go obserwować i ponownie się go uczyć,

tak jak starsi obserwują młodszych. Rzemiosło jest zaraźliwe, to swego rodzaju wirus, który można złapać, obserwując innych i pozwalając memowi zagnieździć się w sobie.

## Przekonywanie

Nie można nikogo przekonać do bycia rzemieślnikiem. Nie można naklonić nikogo do przyjęcia memu rzemiosła. Tutaj nie działają żadne argumenty, żadne dane nie mają znaczenia, a studia przypadków są pustymi historiami. Przyjęcie tego memu nie jest decyzją racjonalną, ale emocjonalną. To bardzo *ludzkie*.

Jak zatem sprawić, żeby ludzie zaakceptowali mem rzemiosła? Pamiętaj, że ten mem jest zaraźliwy, ale tylko wtedy, gdy jest obserwowany. Trzeba zatem sprawić, żeby ktoś *chciał* go obserwować. Musisz stać się dla innych wzorcem. Najpierw to Ty musisz stać się rzemieślnikiem i sprawić, żeby Twoje rzemiosło było widoczne. Potem pozwól memowi wykonać resztę pracy.

## Wnioski

W szkołach można nauczyć się teorii programowania komputerów. Szkoła nie jest jednak w stanie nauczać dyscypliny, praktyki i umiejętności niezbędnych rzemieślnikom. Te rzeczy nabywa się przez lata wypełnione osobistą kuratelą i uczeniem się. Musimy sobie uświadomić, że wprowadzenie następnych pokoleń programistów w zawodową dorosłość będzie naszym zadaniem, a nie zadaniem uniwersytetów. Nadszedł dla nas czas przyjęcia programu praktyk, stażów i długoterminowego prowadzenia młodych.



---

# A NARZĘDZIA

---



W 1978 roku pracowałem w firmie Teradyne nad systemem testującym telefony, o którym już wcześniej opowiadałem. System składał się z mniej więcej 80 000 wierszy kodu w asemblerze komputera M365. Cały kod źródłowy był przechowywany na taśmach.

Same taśmy były podobne do kasetek do magnetofonów 8-ścieżkowych, tak popularnych w latach 70. Taśma była niekończącą się pętlą, a napęd mógł przewijać ją tylko w jednym kierunku. W kasetkach były dostępne taśmy o długości 3, 7, 15 i 30 metrów. Im dłuższa była taśma, tym dłużej trwało jej „przewijanie”, ponieważ napęd musiał ją przesuwać naprzód aż do napotkania „punktu lądowania”. W przypadku taśm 30-metrowych procedura

poszukiwania takiego punktu trwała 5 minut, dlatego zawsze dobieraliśmy odpowiednio taśmę do aktualnego zadania<sup>1</sup>.

Każda taśma była logicznie dzielona na pliki. Na jednej taśmie można było zapisać tyle plików, ile się na nią zmieściło. Chcąc odszukać konkretny plik na taśmie, trzeba było ją załadować i przewijać po jednym pliku do momentu znalezienia tego poszukiwanego. Listę zawartości katalogu z kodem źródłowym mieliśmy zapisaną na scianie, dzięki temu wiedzieliśmy, ile plików ominąć, żeby dostać się do potrzebnego.

Na półce w laboratorium mieliśmy też główną kopię kodu źródłowego na taśmie 30-metrowej. Miała ona etykietę **Master**. Chcąc edytować jakiś plik, ładowaliśmy źródłową taśmę główną do jednego napędu, a czystą taśmę trzymetrową wkładaliśmy do drugiego. Na taśmie źródłowej pomijaliśmy poszczególne pliki, aż znaleźliśmy aktualnie potrzebny. Następnie znaleziony plik kopiowaliśmy na czystą taśmę. Potem obie taśmy były „przewijane”, a taśma główna wędrowała z powrotem na półkę.

Na specjalnej tablicy w laboratorium znajdowała się lista katalogu taśmy **Master**. Po wykonaniu kopii pliku, który miał być edytowany, umieszczałyśmy kolorową pineskę przy jego nazwie. Tak wyglądało wymeldowanie plików!

Pliki edytowaliśmy na ekranie. Nasz edytor ED-402 był naprawdę dobry i trochę przypominał słynnego vi. Odczytywaliśmy z taśmy „stronę”, edytowaliśmy jej zawartość, następnie ją zapisywaliśmy i przechodziliśmy do następnej. Na stronę składało się zwykle 50 wierszy kodu. Nie dało się obejrzeć dalszych stron ani skontrolować stron poprzednio edytowanych, dlatego zawsze stosowaliśmy listingi.

Co więcej, w istniejących listingach zaznaczaliśmy zmiany, które chcieliśmy wprowadzić, a potem edytowaliśmy pliki zgodnie z wcześniej przygotowanymi oznaczeniami. *Nikt* nie pisał ani nie modyfikował kodu na terminalu! To byłoby samobójstwo.

Po wprowadzeniu zmian we wszystkich wymagających tego plikach łączyliśmy je z główną taśmą, tworząc tym samym taśmę roboczą. Wykorzystywaliśmy ją do przeprowadzenia komplikacji i testów.

Po zakończeniu testowania i upewnieniu się, że wprowadzone zmiany działają, sprawdzaliśmy tablicę w laboratorium. Jeżeli nie pojawiły się na niej żadne nowe pineski, to po prostu

---

<sup>1</sup> Taśmy przewijały się tylko w jednym kierunku. Jeżeli przy odczytcie zdarzył się błąd, napęd nie miał już możliwości cofnięcia się i odczytania danych jeszcze raz. Trzeba było przerwać procedurę, przewinąć taśmę do punktu ładowania i rozpocząć wszystko od nowa. Zdarzało się to dwa lub trzy razy dziennie. Powszechnie były też błędy zapisu, a napęd nie miał żadnych możliwości ich wykrywania. Dlatego właśnie zawsze zapisywaliśmy taśmy parami, a po zakończeniu zapisu każdą z nich kontrolowaliśmy. Jeżeli jedna z taśm zawierała błędy, to natychmiast robiliśmy kopię. Jeżeli obie taśmy były uszkodzone, co nie zdarzało się często, całą operację zaczynaliśmy od początku. Tak właśnie wyglądało życie w latach 70.

nadawaliśmy taśmie roboczej etykietę Master, a z tablicy zdejmowaliśmy pineski. Jeżeli jednak w międzyczasie pojawiły się nowe pineski, to usuwaliśmy własne i przekazywaliśmy taśmę roboczą osobie, której pineski nadal znajdowały się na tablicy. To ona musiała połączyć wszystkie zmiany.

Było nas trzech i każdy używał pinesek innego koloru, więc łatwo było sprawdzić, kto jeszcze pracował nad kodem. A dzięki temu, że wszyscy pracowaliśmy w tym samym laboratorium i ciągle ze sobą rozmawialiśmy, status z tablicy mogliśmy łatwo zapamiętać. Okazywało się zatem, że tablica była niepotrzebna i dlatego często jej nie używaliśmy.

## Narzędzia

Dzisiaj programiści mogą wybierać z wielkiego zasobnika różnych narzędzi. Większość z nich nie jest warta zainteresowania, ale jest kilka takich, które każdy programista powinien doskonale znać. W tym rozdziale przedstawię aktualną zawartość mojego narzędziownika. Nie sprawdzałem dokładnie wszystkich dostępnych narzędzi, dlatego nie można uznać tego opisu za wyczerpujący. Po prostu tych narzędzi używam.

## Kontrola kodu źródłowego

Jeżeli chodzi o kontrolę kodu źródłowego, to narzędzia o otwartych źródłach zwykle są najlepszym wyborem. Dlaczego? Z tego prostego powodu, że są pisane przez programistów dla programistów. Narzędzia otwartoźródłowe tworzą programiści dla samych siebie zawsze wtedy, gdy potrzebują czegoś, co naprawdę działa.

Na rynku dostępnych jest kilka dosyć drogich, komercyjnych systemów kontroli wersji klasy „enterprise”. Uważam, że takie narzędzia nie są sprzedawane samym programistom, ale raczej menedżerom, dyrektorom i innym „grupom narzędziowym”. Lista funkcji takich systemów jest imponująca, ale niestety zazwyczaj nie mają one tego, co jest najbardziej potrzebne programistom. A najważniejsza jest *szybkość*.

### System kontroli wersji klasy „enterprise”

Być może Twoja firma zainwestowała małą fortunę w system kontroli wersji klasy „enterprise”. W takiej sytuacji składam swoje kondolencje. Zapewne nie byłoby politycznie poprawne chodzenie teraz po firmie i wołanie: „Wujek Bob twierdzi, że tego nie można używać”. Na szczęście istnieje pewne proste rozwiązywanie.

Możesz wprowadzać kod źródłowy do firmowego systemu kontroli wersji pod koniec każdej iteracji (na przykład raz na tydzień), a w międzyczasie lokalnie korzystać z jednego z systemów otwartoźródłowych. W ten sposób każdy będzie szczęśliwy, żadne firmowe zasady nie zostaną naruszone, a Twoja produktywność się zwiększy.

## Blokowanie pesymistyczne i optymistyczne

Blokowanie pesymistyczne było całkiem dobrą metodą gdzieś w latach 80. W końcu najprostszy sposób kontrolowania jednoczesnych aktualizacji to ich poszeregowanie. A zatem skoro *ja* edytuję plik, to *ty* trzymaj się od niego z daleka. Co ciekawe, system z kolorowymi pineskami, którego używałem w latach 70., był takim właśnie rodzajem pesymistycznego blokowania. Jeżeli przy nazwie pliku była pineska, to nie wolno było go edytować.

Oczywiście blokowanie pesymistyczne rodziło inne problemy. Jeżeli zablokowałem plik, a potem poszedłem na urlop, to ktoś, kto chciałby go też edytować, nie mógł wykonać swojej pracy. Co więcej, nawet jeżeli blokuję plik tylko przez dzień lub dwa, to i tak mogę opóźnić pracę innych, którzy też chcieli coś do tego pliku dopisać.

Od tamtej pory bardzo rozwinęły się narzędzia pozwalające na łączenie jednocześnie edytowanych plików źródłowych. Jeżeli się nad tym zastanowić, to jest to naprawdę niesamowite. Narzędzia te analizują dwa różne pliki, porównują je z plikiem wyjściowym i stosują różne strategie łączenia pozwalające na właściwe połączenie ze sobą dwóch osobnych zmian. I efekt jest naprawdę świetny.

A zatem era blokowania pesymistycznego się zakończyła. Nie musimy już blokować plików, które chcemy edytować. W ogóle nie musimy pamiętać o wyciąganiu z repozytorium poszczególnych plików. Po prostu pobieramy cały system i edytujemy wymagające tego pliki.

Gdy jesteśmy już gotowi wprowadzić nasze zmiany do repozytorium, wystarczy wykonać operację „aktualizacji”. W ten sposób dowiadujemy się, czy ktoś inny nie wprowadził przed nami zmian, które są automatycznie łączone z naszymi poprawkami, wyłapywane są też wszelkie konflikty. Mimo że te ostatnie musimy rozwiązać sami, to jednak narzędzia nadal nas w tym wspomagają. Na koniec wystarczy już tylko umieścić w repozytorium połączony kod.

W dalszej części dodatku będę miał wiele do powiedzenia na temat roli, jaką w tym procesie odgrywają zautomatyzowane testy i ciągła integracja. Na razie ograniczę się do stwierdzenia, że do repozytorium *nigdy* nie wysyłamy kodu, który nie przeszedł wszystkich testów. *Absolutnie nigdy!*

## CVS i SVN

Najstarszym znanym systemem kontroli wersji jest CVS. W swoich czasach był doskonałym rozwiązaniem, ale nie sprawdza się już w dzisiejszych projektach. Mimo że świetnie radzi sobie z pojedynczymi plikami i katalogami, to jednak operacje jak zmiana nazw plików albo usuwanie katalogów go przerastają. A poza tym... może lepiej pominąć to milczeniem.

Z kolei Subversion działa naprawdę przyjemnie. Umożliwia pobranie całego systemu w jednej tylko operacji. Możliwe jest łatwe aktualizowanie, łączenie i wstawianie. Dopóki nie zaczniesz bawić się w tworzenie rozgałęzień, system SVN jest bardzo prosty w obsłudze.

## Rozgałęzianie

Do 2008 roku stosowałem jedynie najprostsze formy tworzenia rozgałęzień. Jeżeli programista utworzył nowe rozgałęzienie, to musiało ono zostać włączone do głównej linii jeszcze przed zakończeniem iteracji. Naprawdę tak bardzo nie lubiłem rozgałęziania, że w projektach, w których uczestniczyłem, praktycznie ich nie używano.

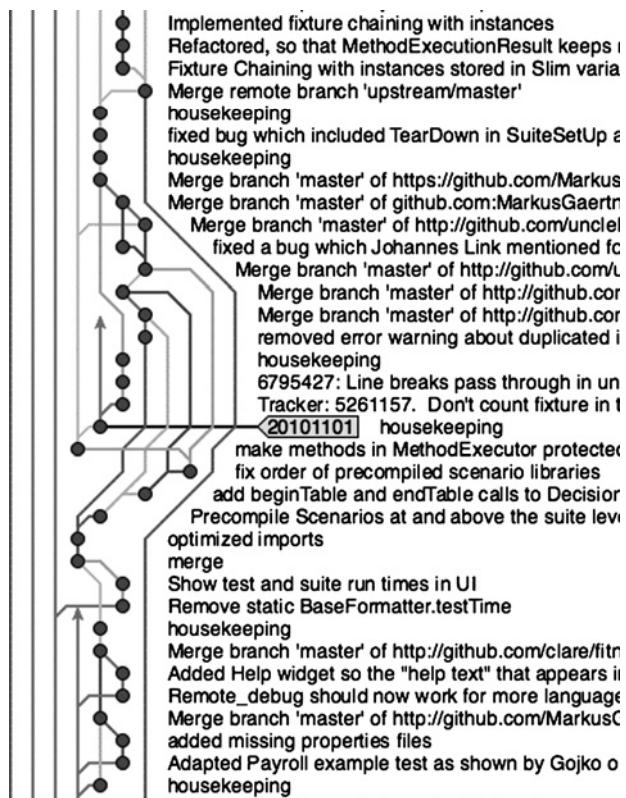
Nadal sądzę, że stosując SVN, należy trzymać się z daleka od rozgałęzień. Na szczęście istnieje kilka nowych narzędzi, które diametralnie zmieniają zasady gry. Są nimi *rozproszone* systemy kontroli wersji. Moim ulubionym jest Git i o nim chciałbym pokróćce opowiedzieć.

### Git

Systemu Git zacząłem używać w 2008 roku i od razu zmienił on wszystko w moim sposobie wykorzystywania mechanizmów kontroli wersji. Wyjaśnienie, dlaczego to narzędzie powoduje tak ogólną zmianę, wykracza poza ramy tej książki. Jednak porównanie rysunku A.1 z rysunkiem A.2 powinno zastąpić wiele wyjaśnień, które pozwolą sobie zatem pominąć.

- More bug fixes
- Docs now say that Java 1.5 is required.
- Bug fix
- Many usability and behavioral improvements.
- Clean up
- Added PAGE\_NAME and PAGE\_PATH to pre-defined variables.
- Added \*\* to Ipath widget.
- link to the fixture gallery
- fixture gallery release 2.0 (2008-06-09) copied into the trunk wiki at
- Firefox compatibility for invisible collapsible sections; removed .ce
- Updated documentation suite for all changes since last release.
- Enhancement to handle nulls in saved and recalled symbols. Adds
- Added a "Prune" Properties attribute to exclude a page and its chil
- Fixed type-o
- Added check for existing child page on rename.
- Added "Rename" link to Symbolic Links property section; renamed
- Adjusted page properties on recently added pages such that they c
- Enhanced Symbolic Links to allow all relative and absolute path for
- Cleaned up renamPageReponder a bit more.
- Cleaned Up PathParser names a bit. Pop -> RemoveNameFromE
- Cleaned up RenamePageReponder a bit. Fixed TestContentsHel
- updated usage message
- Fixed a bug wherein variables defined in a parent's preformatted bl
- Added explicit responder "getPage" to render a page in case query
- Tweaks to TOC help text.
- New property: Help text; TOCWidget has rollover balloon with new
- Redundant to the JUnit tests and elemental acceptance tests.
- Removed the last of the [acd] tags.
- Icontents -f option enhancement to show suite filters in TOC list; fix
- TOC enhancements for properties (-p and PROPERTY\_TOC and F
- 1) Render the tags on non-WikiWord links;
- Added http:// prefix to google.com for firewall transparency.
- Isolate query action from additional query arguments. For example
- Accommodate query strings like "?suite&suiteFilter=X"; prior logic \
- Cleaned up AliasLinkWidget a bit.

Rysunek A.1. FitNesse w Subversion



Rysunek A.2. FitNesse w systemie Git

Na rysunku A.1 przedstawiono kilka tygodni prac nad projektem FitNesse w czasie, gdy pracowałem jeszcze z systemem SVN. Dokładnie widać tu efekty mojej zasady niestosowania żadnych rozgałęzień. Po prostu ich nie tworzyliśmy. Zamiast tego bardzo często dokonywaliśmy aktualizacji głównej linii kodu, łączymy ją i wprowadzaliśmy do niej zmiany.

Na rysunku A.2 przedstawiono kilka tygodni prac nad tym samym projektem, ale już w systemie Git. Jak widać, przez cały czas tworzymy i łączymy różne rozgałęzienia. Nie wynika to z faktu, że nie stosuję się ściśle do zasady braku rozgałęzień. Po prostu okazało się, że jest to całkiem oczywisty i wygodny sposób pracy. Poszczególni programiści mogą tworzyć bardzo krótkie rozgałęzienia, a potem bardzo prosto łączyć je ze sobą.

Zwrócić uwagę, że nie da się tu zauważać głównej linii kodu. Wynika to z faktu, że *takiej nie ma*. Korzystając z systemu Git, nie masz czegoś takiego jak centralne repozytorium ani główna linia kodu. Każdy programista przechowuje na swoim komputerze własną kopię *całej* historii projektu. Operacje pobrania i wstawienia kodu są wykonywane na tej właśnie lokalnej kopi, a potem łączy się je z kopiami pozostałych programistów.

Prawdą jest, że utrzymuję specjalne, „złote” repozytorium, do którego przesyłam poszczególne wydania i pośrednie wersje projektu. Jednak nazwanie tego repozytorium centralnym nie byłoby zgodne z prawdą. Jest to tylko bardzo wygodna migawka całej historii projektu, którą każdy programista przechowuje lokalnie.

Jeżeli tego nie rozumiesz, to się nie przejmuj. Na początek Git naprawdę potrafi zabić klinę. Trzeba się po prostu przyzwyczaić do nowego sposobu pracy. Ale mogę powiedzieć to: Git i podobne do niego narzędzia są przyszłością systemów kontroli kodu źródłowego.

## IDE i edytor

Programiści większość swojego czasu poświęcają na czytanie i edytowanie kodu. Narzędzia, których używamy do tego celu, przez dziesięciolecia bardzo się zmieniły. Niektóre z nich stały się niezwykle wszechstronne, ale inne praktycznie nie zmieniły się od lat 70.

### vi

Można by sądzić, że dni wykorzystywania vi jako edytora do rozwoju oprogramowania powinny już dawno minąć. Dzisiaj mamy narzędzia przewyższające możliwościami vi i inne, podobne do niego proste edytory tekstu. Prawda jest jednak taka, że vi znowu zaczyna zyskiwać na popularności dzięki swojej prostocie, łatwej obsłudze, szybkości i elastyczności. Być może nie jest on aż tak rozbudowany jak Emacs albo Eclipse, ale nadal jest bardzo szybkim i wszechstronnym edytorem.

Po tym wstępnie mogę się przyznać, że nie jestem już zaawansowanym użytkownikiem vi. Był taki czas, kiedy byłem znany jako „bóg” vi, ale to już zamierchła przeszłość. Nadal jednak od czasu do czasu używam vi, jeżeli muszę szybko wprowadzić jakąś zmianę do pliku tekstowego. Ostatnio użyłem go nawet, gdy musiałem szybko poprawić coś w pliku źródłowym Javy w zdalnym środowisku. Jednak ogólna ilość kodu, jaką w ostatnim dziesięcioleciu napisałem za pomocą vi, jest przerażająco niewielka.

### Emacs

Emacs to nadal jeden z najbardziej rozbudowanych edytorów, jakie znam. Najprawdopodobniej nie zmieni się to jeszcze przez co najmniej dekadę, co gwarantuje wewnętrzny model Lispu. Nic nie może się z nim równać w kategorii edytora ogólnego przeznaczenia. Uważam jednak, że Emacs nie może konkurować z dominującymi aktualnie IDE tworzonymi pod konkretne zadania. Pisanie kodu *nie* jest niestety zadaniem dla edytora o ogólnym przeznaczeniu.

W latach 90. byłem prawdziwym wyznawcą Emacsa. Nawet nie myślałem o używaniu czegokolwiek innego. Edytory typu wskaż-i-kliknij z tamtego czasu były śmiesznymi

zabawkami, których żaden programista nie mógł traktować poważnie. Jednak po roku 2000 poznałem IDE o nazwie IntelliJ, którego używam do dzisiaj, i nigdy nie tesknię za starymi czasami.

## Eclipse/IntelliJ

Jestem użytkownikiem środowiska IntelliJ. Po prostu je kocham. Używam go do pisania programów w Javie, Ruby, Clojure, Scali, JavaScripcie i wielu innych językach. To narzędzie zostało napisane przez programistów, którzy dokładnie wiedzą, czego potrzebuje programista tworzący kod. Przez te wszystkie lata tylko rzadko mnie rozczarowywali, a prawie zawsze spełniali moje oczekiwania.

Jeżeli chodzi o możliwości, to Eclipse jest bardzo podobny do IntelliJ. W kontekście edytowania kodu Javy oba IDE pod każdym możliwym względem przewyższają Emacsą. Oczywiście istnieją też inne podobne IDE, ale nie będę o nich wspominał, ponieważ nie mam żadnego doświadczenia w pracy z nimi.

Funkcje, dzięki którym te IDE tak bardzo wyprzedzają Emacsą, dotyczą przede wszystkim ogromnej różnorodności metod manipulacji kodem. Na przykład w IntelliJ za pomocą jednego polecenia można wydobyć superklasę z wybranej klasy. Wśród innych ważnych funkcji można wymienić zmianę nazw zmiennych, wydobywanie metod i przekształcanie dziedziczenia w kompozycje.

Dzięki takim narzędziom edytowanie kodu nie ogranicza się do wierszy i znaków, ale pozwala też na wykonywanie złożonych manipulacji. Zamiast myśleć o kolejnych kilku znakach i wierszach, które musisz wprowadzić, możesz myśleć o kilku przekształceniach, jakie będą jeszcze konieczne. W skrócie: taki model programowania jest zdecydowanie bardziej produktywny.

Oczywiście takie możliwości mają swoją cenę. Poznanie tych narzędzi wymaga czasu, a i skonfigurowanie całego projektu to dodatkowa praca. Tych narzędzi *nie* można nazwać lekkimi. Do działania potrzebują sporo zasobów komputera.

## TextMate

TextMate jest rozbudowanym i jednocześnie lekkim edytorem. Oczywiście nie można w nim robić tych wspaniałych manipulacji na kodzie, na które pozwalają IntelliJ i Eclipse. Nie jest też wyposażony w potężne mechanizmy Lispu i biblioteki Emacsą. Nie ma też szybkości i płynności działania vi. Jednakże prawie nie trzeba się uczyć jego obsługi, która jest zadziwiająco intuicyjna.

Edytora TextMate używam od czasu do czasu, szczególnie do rzadkiego już pisania programów w C++. Do większych projektów w tym języku użyłbym Emacsą, ale do krótkich zadań, jakie na mnie spadają, TextMate w zupełności mi wystarcza.

## Śledzenie problemów

Aktualnie używam systemu Pivotal Tracker, który w codziennym użytkowaniu jest bardzo elegancki i prosty. Doskonale wpasowuje się w rozwiązania iteracyjne i zwinne. Umożliwia szybką komunikację między programistami a udziałowcami projektu. Mogę powiedzieć, że jestem z niego bardzo zadowolony.

W przypadku bardzo małych projektów czasami korzystam z systemu Lighthouse. Jest bardzo szybki i prosty zarówno w konfiguracji, jak i w użytkowaniu, ale pod względem możliwości nie może się równać z Pivotal Trackerem.

Dawniej używaliśmy też po prostu wiki, które doskonale sprawdzają się w projektach wewnętrznych. Umożliwiają przygotowanie dowolnego schematu postępowania. Nie zmuszają nas do stosowania określonych procesów albo sztywnych struktur. Można je łatwo poznać i jeszcze łatwiej się ich używa.

Czasami najlepszym systemem śledzenia problemów jest zestaw kartek powieszonych na tablicy. Taką tablicę dzieli się na trzy części: „Do zrobienia”, „W toku” i „Gotowe”. Programiści muszą tylko przenosić karty z jednej do drugiej kolumny. Taki system śledzenia problemów jest prawdopodobnie najpowszechniejszy używany w zespołach stosujących metodologie zwinne.

Klientom zalecam na początek pracę z ręcznymi systemami, takimi jak kartki na tablicy, i dopiero później dokupienie konkretnego narzędzia. Gdy nauczą się wykorzystywać taki ręczny system, zdobędą też doświadczenie niezbędne do wybrania właściwego rozwiązania. A nieraz jest też tak, że najlepszym rozwiązaniem jest pozostań przy systemie ręcznym.

## Licznik problemów

Zespoły programistów z pewnością muszą mieć listy problemów, nad którymi trzeba pracować. Do takich problemów zaliczają się zarówno nowe zadania i funkcje, jak i zwyczajne błędy. W przypadku zespołów o rozsądnej wielkości (od 5 do 12 programistów) na takiej liście mogą znajdować się dziesiątki lub setki pozycji. *Nie tysiące*.

Jeżeli masz tysiące błędów, to coś na pewno działa źle. Jeżeli masz tysiące nowych funkcji lub zadań, to również coś jest nie tak. Ogólnie rzecz biorąc, lista problemów powinna być względnie krótka, tak żeby można było ją opanować za pomocą lekkich narzędzi, jak wiki, Lighthouse lub Tracker.

Dostępne są też narzędzia komercyjne, które sprawiają całkiem dobre wrażenie. Widziałem używających ich klientów, ale nigdy nie miałem okazji pracować z nimi osobiście. Nie jestem przeciwnikiem takich narzędzi, o ile liczba problemów będzie pozostawać krótką i przejrzystą.

Gdy narzędzia do śledzenia problemów są zmuszone do obsługiwanego tysięcy zgłoszeń, to słowo „śledzenie” traci tu znaczenie. Stają się tylko „śmietnikami problemów” (i często bardzo podobnie śmierdzą).

## Ciągła kompilacja

Ostatnio jako mechanizmu ciągłej kompilacji używałem systemu Jenkins. Jest naprawdę niewielki, prosty i prawie nie wymaga nauki. Wystarczy go pobrać, uruchomić, szybko i łatwo skonfigurować, i gotowe. Sama przyjemność.

Moja filozofia dotycząca ciągłej kompilacji jest bardzo prosta. Podłącz system do systemu kontroli wersji kodu źródłowego. Gdy tylko ktoś wprowadzi zmiany do kodu, kompilacja powinna zostać uruchomiona automatycznie, a jej wynik przesłany do zespołu.

Zadaniem zespołu jest ciągłe utrzymywanie bezproblemowej kompilacji systemu. Każda nieudana kompilacja powinna być sygnałem krytycznym, a zespół powinien starać się jak najszybciej rozwiązać ten problem. Pod żadnym pozorem nie można pozwolić, żeby taka sytuacja trwała dzień lub dłużej.

W przypadku projektu FitNesse nakazuję każdemu programiście uruchomienie skryptu ciągłej kompilacji jeszcze przed wprowadzeniem zmian do repozytorium. Cała kompilacja trwa poniżej 5 minut, a zatem nie jest to bardzo uciążliwe. Jeżeli pojawią się jakieś problemy, to programista musi je rozwiązać, zanim przekaże kod do repozytorium. Dzięki temu automatyczna kompilacja prawie nigdy nie zgłasza błędów. Najczęstszym powodem takich błędów okazują się problemy związane ze środowiskiem, ponieważ środowisko automatycznej kompilacji różni się od środowiska stosowanego przez poszczególnych programistów.

## Narzędzia do testów jednostkowych

Każdy język ma swoje własne narzędzia do testów jednostkowych. Moimi ulubionymi są JUnit dla Javy, RSPEC dla Ruby, NUnit dla .NET, Midje dla Clojure i CppUTest dla C i C++.

Bez względu na to, które narzędzie wybierzesz, wszystkie mają pewien zestaw wspólnych funkcji.

1. Powinny pozwalać szybko i łatwo uruchamiać testy. Niezależnie od tego, czy odbywa się to przez wtyczkę do IDE, czy proste narzędzia wiersza poleceń, programista musi mieć możliwość uruchomienia testów w dowolnym momencie. Praca potrzebna do ich uruchomienia musi być minimalna.

Na przykład testy w CppUTest uruchamiam, wpisując w TextMate polecenie `command-M`. Skonfigurowałem je tak, żeby uruchamiało plik `makefile`, który automatycznie przeprowadza testy i wypisuje raport, gdy wszystkie testy zostaną

zaliczone. IntelliJ obsługuje zarówno JUnit, jak i RSPEC, dlatego wystarczy naciśnięcie tylko jednego przycisku. Do testów NUnit wykorzystuję wtyczkę Resharper, która również udostępnia przycisk uruchamiania testów.

2. Narzędzie powinno jasno przedstawiać informację, czy testy zostały zaliczone, czy też nie. Nie ma znaczenia, czy będzie to graficzny zielony pasek, czy konsolowy komunikat „Testy zaliczone”. Ważne jest to, że musisz być w stanie szybko i jednoznacznie stwierdzić, czy testy udało się zaliczyć. Jeżeli musisz w tym celu czytać wielowierszowy raport albo — co gorsza — porównywać zawartość dwóch plików, to narzędzie nie wykonuje swojego zadania.
3. Narzędzie powinno jasno prezentować postęp w wykonywaniu testów. Nie ma znaczenia, czy będzie to prosty graficzny miernik, czy ciąg kropek. Ważne jest, aby jasno było widać, że testy nadal są wykonywane, a nie zatrzymały się lub zostały przerwane.
4. Narzędzie powinno utrudniać komunikację między poszczególnymi testami. JUnit robi to, tworząc nowy egzemplarz klasy testowej przed uruchomieniem każdej metody testowej. W ten sposób nie pozwala testom na wykorzystywanie zmiennych obiektów do komunikacji. Inne narzędzia wykonują testy w losowej kolejności, tak że nie można zakładać uruchomienia jednego testu przed drugim. Bez względu na zastosowany mechanizm narzędzie powinno wspomagać tworzenie testów całkowicie niezależnych od siebie. Testy uzależnione od siebie są pułapką, w którą nikt nie chce wpaść.
5. Narzędzia powinny ułatwiać pisanie testów. JUnit na przykład udostępnia wygodne API do tworzenia założeń. Wykorzystuje też refleksję i atrybuty Javy, aby odróżnić funkcje testujące od normalnych. Pozwala to dobrym IDE na automatyczne wyszukiwanie testów i eliminowanie tym samym problemów wynikających z łączenia ze sobą poszczególnych zestawów testów oraz tworzenia ich list.

## Narzędzia do testów komponentów

Te narzędzia są przeznaczone do testowania komponentów na poziomie ich API. Ich zadaniem jest sprawdzanie, czy zachowanie komponentu jest opisane w języku zrozumiałym dla menedżerów i działu kontroli jakości. Co więcej, w idealnym przypadku to właśnie analitycy biznesowi i testerzy mogą *pisać specyfikację*, wykorzystując wspomniane narzędzia.

### Definicja gotowego

Za pomocą narzędzi do testów komponentów można bardzo łatwo określić, co znaczy *gotowe*. Jeżeli analitycy biznesowi i testerzy współpracują ze sobą, tworząc specyfikację opisującą zachowanie komponentu, i specyfikację tę można uruchamiać jako zestaw testów, to *gotowe* może mieć tylko jedno znaczenie: wszystkie testy zostały zaliczone.

## FitNesse

Moim ulubionym narzędziem do testów komponentów jest FitNesse. Napisałem sporą część tego systemu i jestem jego głównym programistą. Mogę powiedzieć, że to moje dziecko.

FitNesse jest systemem wykorzystującym wiki, który umożliwia analitykom biznesowym i testerom pisanie testów w prostej, tabelarycznej formie. Tabele te są podobne do tablic Parnasa zarówno w formie, jak i przeznaczeniu. Poszczególne testy można szybko łączyć w zestawy, a same zestawy można bardzo łatwo uruchamiać.

FitNesse jest tworzony w Javie, ale może testować systemy pisane w dowolnym języku, ponieważ komunikuje się z podstawowym systemem testowym, który może powstać w innym języku. Na liście obsługiwanych języków znajdują się Java, C#/.NET, C, C++, Python, Ruby, PHP, Delphi i inne.

U podstaw FitNesse znajdują się dwa systemy testowe: Fit i Slim. Fit został napisany przez Warda Cunninghama i stał się inspiracją do powstania FitNesse i podobnych narzędzi. Slim to znacznie prostszy i przenośny system testowy, z którego chętniej korzystają użytkownicy FitNesse.

## Inne narzędzia

Znam też kilka innych narzędzi, które można zakwalifikować do kategorii testów komponentów.

- RobotFX jest narzędziem przygotowanym przez inżynierów firmy Nokia. Stosuje tabelaryczny format podobny do używanego w FitNesse, ale za podstawę nie służy mu wiki. Wykorzystuje zwyczajne pliki przygotowywane za pomocą Excela lub podobnych aplikacji. Samo narzędzie zostało napisane w Pythonie, ale może służyć do testowania dowolnego innego języka za pośrednictwem odpowiednich mostków.
- Green Pepper to narzędzie komercyjne, o wielu podobieństwach do FitNesse. Bazuje na popularnym wiki Confluence.
- Cucumber to proste narzędzie tekstowe powstające w Ruby, ale umożliwiające testowanie różnych platform. Językiem stosowanym w Cucumberze jest popularny styl Mając/Gdy/To (ang. *Given/When/Then*).
- JBehave to narzędzie podobne do Cucumbera, jako że jest jego logicznym rodzicem. System został napisany w Javie.

## Narzędzia do testów integracyjnych

Narzędzia do testów komponentów mogą być też używane do wielu testów integracyjnych, ale słabo nadają się do wykonywania testów sterowanych poprzez interfejs użytkownika.

Szczerze mówiąc, nie chcemy przeprowadzać wielu testów sterowanych interfejsem użytkownika, ponieważ takie interfejsy cały czas się zmieniają. Ich ulotność sprawia, że wykorzystujące je testy są bardzo wrażliwe.

Istnieje jednak pewien rodzaj testów, który *musi* używać UI. Najważniejsze są w tej grupie *same* testy interfejsu użytkownika. Oprócz tego niektóre testy muszą angażować złożony w całość system, a to oznacza, że muszą też uwzględniać UI.

Narzędzia, których najczęściej używam podczas testów interfejsów użytkownika, to Selenium i Watir.

## UML/MDA

Na początku lat 90. miałem wielką nadzieję, że narzędzia typu CASE całkowicie zmienią sposób, w jaki rozwijane jest oprogramowanie. Patrząc w przyszłość, sądziłem, że dzisiaj każdy będzie tworzył oprogramowanie za pomocą diagramów na wysokim poziomie abstrakcji, a kod składający się z tekstu będzie należeć do przeszłości.

Straszliwie się pomyliłem. Nie chodzi nawet o to, że moje marzenie się nie spełniło, ale że każda próba jego realizacji skończyła się katastrofalnym fiaskiem. Oczywiście istnieją narzędzia i systemy będące demonstracją możliwości tej technologii, ale niestety nie są one faktycznym spełnieniem marzenia, a na dodatek wygląda na to, że nikt nie ma zamiaru ich używać.

Moje marzenie zakładało, że programiści porzucą szczegółowość kodu tekstowego i zaczną tworzyć systemy za pomocą wysokopoziomowych języków zbudowanych z diagramów. Co więcej, mogłyby się okazać, że wcale nie potrzebujemy programistów. Architekci mogliby tworzyć całe systemy za pomocą diagramów UML. Następnie zastępujące programistów potężne, choć zimne mechanizmy przekształcałyby te diagramy w wykonywalny kod. Tak właśnie wyglądało wielkie marzenie o architekturze sterowanej modelami (MDA — *Model Driven Architecture*).

Niestety, w tym wspaniałym marzeniu była mała skaza. MDA zakłada, że to kod jest problemem. Ale kod problemem *nie jest*. Nigdy nim nie był. Prawdziwym problemem są *szczegółы*.

## Szczegóły

Programiści zarządzają szczegółami. Dokładnie tak! Definiujemy zachowanie systemu w najdrobniejszym detalu. Wykorzystujemy przy tym języki tekstowe (kod), ponieważ taka ich forma jest niezwykle wygodna (pomyśl na przykład o języku polskim).

Jaki rodzaj szczegółów musimy obsługiwać?

Znasz różnicę między znakami `\n` i `\r`? Pierwszy z nich (`\n`) oznacza wysunięcie wiersza. Drugi (`\r`) to powrót karetki. Ale czym jest karetka?

Na przełomie lat 60. i 70. najpowszechniejszym urządzeniem wyjściowym komputerów był teletekst. Najczęściej używany był model ASR-33<sup>2</sup>.

Urządzenie do miało głowicę drukującą zdolną do wydrukowania dziesięciu znaków na sekundę. Składała się ona z małego cylindra, na którym wygrawerowane były poszczególne znaki. Był on przesuwany i obracany tak, żeby w stronę papieru skierowany był właściwy znak. Pomiędzy cylindrem a papierem była umieszczona wstążka z atramentem, który był nakładany na papier w kształcie wybranego znaku.

Cała głowica drukująca poruszała się na karetce. Po wydrukowaniu każdego znaku karetka była przesuwana o jedną pozycję w prawo, przenosząc przy okazji głowicę. Gdy ostatecznie docierała do końca 72-znakowego wiersza, trzeba było nakazać jej powrót do początku, wysyłając znak powrotu karetki (`\r = 0x0D`). Bez tego głowica wypisywałaby kolejne znaki w 72. kolumnie, tworząc paskudny, czarny prostokąt.

Oczywiście nie było to wystarczające. Powrót karetki na początek wiersza nie powodował podniesienia papieru o jeden wiersz. Jeżeli po powrocie karetki nie został wysłany znak wysunięcia wiersza (`\n = 0x0A`), to nowe znaki były wypisywane w już raz zapisanym wierszu.

Oznacza to, że w przypadku teletekstu ASR-33 sekwencją znaków kończącą wiersz było `"\r\n"`. Trzeba było na to zwracać uwagę, ponieważ czas powrotu karetki mógł być dłuższy niż 100 ms. Jeżeli wysłałoby się sekwencję `"\n\r"`, to kolejny znak mógłby zostać wydrukowany jeszcze w czasie, gdy karetka wracała na początek wiersza, co utworzyłoby rozmażany znak gdzieś na środku strony. Na wszelki wypadek do sekwencji kończącej wiersz często dodawaliśmy jeszcze jeden znak wymazywania<sup>3</sup> (`0xFF`) lub dwa.

W latach 70. teleteksty zaczęły wychodzić z użycia, a systemy operacyjne takie jak UNIX skróciły sekwencję znaków końca wiersza do prostego `"\n"`. Niestety, inne systemy operacyjne, takie jak DOS, nadal stosowały starą konwencję `"\r\n"`.

Kiedy ostatnio podczas pracy z plikami tekstowymi zdarzyło Ci się zastosować „niewłaściwą” konwencję? Z problemem tym stykam się przynajmniej raz do roku. Dwa takie same pliki tekstowe nie są identyczne, nie generują tej samej sumy kontrolnej, ponieważ mają inne zakończenia wierszy. Edytory tekstu nie zauważają poprawnie wierszy albo tworzą podwojone

---

<sup>2</sup> [http://en.wikipedia.org/wiki/ASR-33\\_Teletype](http://en.wikipedia.org/wiki/ASR-33_Teletype).

<sup>3</sup> Znaki wymazywania bardzo przydawały się podczas edytowania taśm papierowych. Zgodnie z konwencją znaki te były ignorowane, ale ich kod `0xFF` oznaczał, że na taśmie pozostały wybite wszystkie dziurki. Oznaczało to, że każdy znak dało się zmienić w znak wymazywania przez wybicie dodatkowych dziurek. Dzięki temu w przypadku popełnienia błędu w czasie wpisywania programu można było cofnąć się do tego znaku, nacisnąć klawisz *wymaż* (ang. *rubout*) i pisać dalej.

odstępów między nimi, gdyż zakończenia wierszy są „nieprawidłowe”. Programy, które nie dopuszczają pustych wierszy, odmawiają działania, ponieważ interpretują znaki `\r\n` jako dwa wiersze. Niektóre programy rozpoznają sekwencję `\r\n\r\n`, ale nie znają już `\n\r`. Można tak długo opowiadać.

*Dlatego właśnie wspomniałem o szczegółach.* Spróbuj zapisać straszliwą logikę algorytmu sortowania za pomocą UML!

## Bez nadziei, bez zmian

Nadzieją związaną z przejściem na MDA było to, że wielką część szczegółów będzie można wyeliminować przez zastąpienie kodu diagramami. Ta nadzieja została jednak porzucona. Okazuje się, że w kodzie nie ma zbyt wielu szczegółów, które można by wyeliminować za pomocą obrazków. Co więcej, obrazki wprowadzają swoją własną grupę szczegółów. Mają swoją gramatykę, składnię, reguły i ograniczenia. A zatem nie ma żadnej różnicy.

Nadzieją związaną z MDA było to, że diagramy staną się wyższym poziomem abstrakcji kodu, tak jak Java jest na wyższym poziomie w stosunku do asemblera. I ponownie ta nadzieja okazała się źle ulokowana. Różnica w poziomie abstrakcji jest w najlepszym razie minimalna.

Załóżmy na koniec, że pewnego dnia komuś uda się wymyślić naprawdę przydatny język diagramowy. Rysowaniem tych diagramów nie będą się zajmować architekci, ale programiści, ponieważ staną się one po prostu nową formą kodu, którą programiści będą rysować, a nie piszą. Ostatecznie wszystko sprowadza się do szczegółów, a to właśnie programiści zajmują się szczegółami.

## Wnioski

Od czasu gdy zacząłem zajmować się oprogramowaniem, narzędzia programistyczne zyskały wiele możliwości i stały się bardziej różnorodne. Osobiście używam tylko niewielkiego podzbioru tej całej menażerii. Do kontroli wersji kodu źródłowego używam Gita, Trackera do zarządzania problemami, Jenkinsa do ciągłej komplikacji, IntelliJ jest moim IDE, XUnit służy mi do testów, a FitNesse do testowania komponentów.

Moim komputerem jest MacBook Pro, 2,8 GHz Intel Core i7, z 17-calowym matowym ekranem, 8 GB pamięci RAM, 512-gigabajtowym dyskiem SSD i dwoma dodatkowymi ekranami.



---

# SKOROWIDZ

---

## A

algebra Boole'a, 183  
algorytm sortowania, 111  
anityk biznesowy, 124, 132, 169, 176, 177, 204  
analiza  
    strukturalna, 45  
    trzech zmiennych, 154  
API, 127, 128  
    testowanie, *Patrz: test API*  
aplikacja mobilna, 61  
architekt, 45, 135, 205, 207  
architektura sterowana modelami, *Patrz: MDA*  
artefakt, 45

## B

backlog, 140, 141  
Beck Kent, 98, 141  
biblioteka zewnętrzna, 101  
biznesmen, 116, 117  
Blanco John, 60  
blokowanie pesymistyczne, 196  
błędy, 39, 80, 131, 201  
    regresyjne, 133  
Boehm Barry, 157  
Boole'a algebra, 183  
Bossavit Laurent, 109  
Bowling game, *Patrz: Gra w kręgle*

## C

Carlin Jim, 186  
CASE, 205  
CDS, 70  
cele, 52  
CI, *Patrz: system ciągłej integracji*  
ciąg Fibonacciego, 158, 159  
Coding Dojo, *Patrz: dojo kodowania*  
Conrad Tim, 167  
continuous integration system, *Patrz: system ciągłej integracji*  
CppUTest, 202  
Craft Dispatch System, *Patrz: CDS*  
Cucumber, 93, 124, 134, 204  
cuke4duke, 124  
Cunningham Ward, 204  
CVS, 196  
Czynnik pierwszy, 46

## Ć

ćwiczenia, 46, 105, 108, 113, 187, 190

## D

De Morgana twierdzenie, 183  
debugowanie, 87, 98  
    czas, 89  
DeMarco Tom, 118

DFD, 45  
diagram  
przejść stanów, 45  
przepływu, 45  
przepływu danych, 44  
UML, 205

Digi-Comp, 182  
dojo kodowania, 109, 110  
dokument wymagań systemu, 118  
dokumentacja, 101, 127  
niskopoziomowa, 102

**E**

Eclipse, *Patrz:* edytor Eclipse  
ECP-18, 184  
edytor, 116  
Eclipse, 199, 200  
Emacs, 199  
IntelliJ, 200  
TextMate, 200, 202  
vi, 199  
efekt obserwatora, 117  
elektroniczny recepcjonista, 69  
Emacs, *Patrz:* edytor Emacs  
Emma, 41  
Extreme Programming, *Patrz:* XP

**F**

Fibonacciego ciąg, 158, 159  
Finder Ken, 69  
FitNesse, 40, 41, 93, 100, 108, 124, 134, 198, 204  
Fitzpatrick Jerry, 69  
flow, *Patrz:* przepływ  
funkcja, 102

**G**

Git, 197, 198  
gra ping-pong, *Patrz:* ping-pong  
Gra w kręgle, 46, 109  
gracz zespołowy, 55, 57, 59  
Green Pepper, 204  
Grenning James, 158  
GUI, *Patrz:* interfejs użytkownika

**H**  
hierarchia dziedziczenia, 43  
hiperproduktywność, *Patrz:* przepływ  
Hipokrates, 38  
Hoffa Jimmy, 49

**I**

IDE, *Patrz:* edytor  
idiom nawigacyjny, 110  
instrukcja wyboru, 43  
IntelliJ, *Patrz:* edytor IntelliJ  
interfejs użytkownika, 127, 128, 205  
iteracja, 140, 195  
retrospekytywa, 141

**J**

JBehave, 134, 204  
Jenkins, 202  
język  
makr, 116  
skryptowy, 116  
zobowiązań, 71, 72, 74, 75  
JUnit, 202, 203

**K**

karetka, 206  
kariera, 43, 44  
kata, 46, 109  
kod  
edytowanie, 199  
gotowy, 92, 120, 203  
inspekcja, 173  
pokrycie testami, 99  
struktura, 41, 79  
tworzenie, 79, 80, 82, 85, 86, 87, 90, 91, 92, 93, 98,  
146, 164  
czas, 89  
własność, 172  
kompilacja, 98, 107  
ciąгла, 202  
nieudana, 99  
kontrola jakości, 39, 41, 132

**L**

latające palce, 157, 158  
 Lighthouse, 201  
 Lindstrom Lowell, 159

**Ł**

łańcuch dowodzenia, 43

**M**

manna skupienia, 142, 143  
 maszyna stanu, 44  
 MDA, 205, 207  
 Mealy George, 44  
 medytacja, 83  
 menedżer, 115, 118, 169, 176, 177, 195  
 mentor, 95, 187, 189  
 metodologia, 189  
     analizy strukturalnej, 45  
     Kanban, 45  
     Lean, 45  
     projektowania strukturalnego, 45  
     Scrum, 45, 97  
     wodospadu, 45  
     XP, 45  
     zwinna, 97, 140, 201  
 Midje, 202  
 mikrozarządzanie, 54  
 mistrz, *Patrz:* mentor  
 Model Driven Architecture, *Patrz:* MDA  
 Moore Gordon, 44  
 Moore Michael, 107  
 Murphy'ego prawo, 153  
 muzyka, 84, 108, 110

**N**

narzędzia, 195, 196, 197, 201, 202  
 CASE, 205  
 edytowanie kodu, *Patrz:* edytor  
 o otwartych źródłach, 195  
 test  
     integracyjny, 204  
     jednostkowy, 202  
     komponentów, 203, 204  
 Nassi Ike, 44  
 NUnit, 202

**O**

odpowiedzialność, 36, 39, 43, 47, 72, 164  
     zasada, *Patrz:* SRP  
 odwrócenie priorytetów, 145  
 oprogramowanie, 188  
     optymalizacja, 168  
 Osherove Roy, 70, 71

**P**

PERT, 157  
 Petriego sieć, 45  
 ping-pong, 111  
 Pivotal Tracker, 201  
 planning poker, *Patrz:* planujący poker  
 planujący poker, 158  
 poczta głosowa, 69, 70  
 polimorfia, 43  
 pomodoro technique, *Patrz:* technika pomidora  
 powrót karetki, 206  
 prawdopodobieństwo, 153  
     rozkład, 153, 154, 155  
 prawo  
     Murphy'ego, 153  
     wielkich liczb, 159  
 Prime factor, *Patrz:* Czynnik pierwszy  
 problem  
     późnej wieloznaczności, 118  
     przedwczesnej dokładności, 118  
 profesjonalizm, 36  
 program, *Patrz:* projekt  
 Program Evaluation and Review Technique,  
     *Patrz:* PERT  
 programista, 45, 52, 60, 61, 65, 101, 115, 118, 120,  
     125, 127, 133, 149, 169, 176, 190, 195, 205, 207  
 czeladnik, 189, 190  
 gracz zespołowy, *Patrz:* gracz zespołowy  
 mistrz, *Patrz:* mentor  
 pod presją, 162, 163, 165  
 praktykant, 189, 190  
 rzemieślnik, 190, 191  
 szkolenie, 189  
 współpraca, 46, 160, 167, 169, 172, 173, 176, 177  
 zespół, *Patrz:* programowanie w zespole  
 programowanie, 48, 80, 86, 89, 90, 93, 98, 142, 144, 146  
 blokada, 85, 86, 196  
 ekstremalne, *Patrz:* XP

- programowanie  
    lista problemów, 201  
    łączenie edytowanych plików, 196  
    metodą ciągley integracji, 45  
    narzędzia, *Patrz*: narzędzia  
        nauczanie, 182, 186, 187  
    obiektowe, 61  
    rozgałęzienie, *Patrz*: rozgałęzienie  
    sterowane testami, *Patrz*: TDD  
    strukturalne, 45  
szacowanie, 118, 149, 151, 152, 153, 154, 157, 158,  
    159, 160, 190  
śledzenie problemów, 201  
tempo, 177  
w parach, 45, 47, 84, 85, 164, 166, 172, 173, 189  
w systemie wodospadu, 44  
w zespole, 167, 172, 174, 176, 177
- projekt  
    elastyczność, 42  
    funkcja, 41  
    iteracja, *Patrz*: iteracja  
    otwartoźródłowy, 40  
    struktura, 41, 45  
    właściciel, 178  
    zmiany, 41
- projektant, 45  
    gracz zespołowy, *Patrz*: gracz zespołowy
- projektowanie, 65  
    fakty, 54  
    koszty, 54  
    obiektowe, 45, 150  
    strukturalne, 45  
    zasady, 45  
        SOLID, 45
- przekleństwo Santayany, 45  
przepływ, 83, 84  
przerwy, 85  
przysięga Hipokratesa, 38
- R**
- randori, 111  
raport Emma, 41  
refaktoryzacja, 42, 46, 83, 107, 165, 190  
reguła  
    biznesowa, 128, 129  
    dziury, 146  
    skauta, 42  
rezydentura, 188
- Robot Framework, 124  
RobotFX, 93, 204  
rozgałęzienie, 197  
RSPEC, 202  
rzemiosło, 190, 191
- S**
- Santana Carlos, 108  
Santayany przekleństwo, 45  
scenariusz, 125  
Schneiderman Ben, 44  
Selenium, 93, 124, 134, 205  
sieć Petriego, 45  
Single Responsibility Principle, *Patrz*: SRP  
skrót klawiszowy, 110  
spotkanie  
    cel, 140  
    demonstracja produktu, 141  
    koszt, 138  
    na stojąco, 140  
    plan, 140  
    planujące iteracje, 140, 141  
    retrospekytywa iteracji, 141  
    selekция, 138  
    wychodzenie, 139
- SRP, 128
- stażysta, 187
- strefa, 83, 84, 85  
strona trzecia, 62  
struktury wykres, 45  
SVN, 196, 197, 198
- system  
    ciągley integracji, 110, 129  
    konstrukcja, 136  
    kontroli wersji, 129, 195, 202  
        CVS, *Patrz*: CVS  
        Git, *Patrz*: Git  
        klasy „enterprise”, 195  
        rozproszone, 197  
        Subversion, *Patrz*: SVN  
        SVN, *Patrz*: SVN
- przydzielający zadania, *Patrz*: CDS  
testowanie, *Patrz*: test systemu
- szacunek  
    normalny, 155  
    optimistyczny, 154  
    pesymistyczny, 155
- szkolenia, 43

**T**

tabela  
 decyzji, 45  
 przejść stanów, 45  
 tablica Parnasa, 44, 204  
 TDD, 41, 45, 83, 85, 89, 97, 98, 99, 100, 101, 103, 109, 110, 133, 136, 164, 165, 190  
 Teamsters, 49  
 technika  
 oceny i rozwoju programów, *Patrz*: PERT  
 PERT, *Patrz*: PERT  
 pomidora, 144  
 test, 42, 131, 133  
 akceptacyjny, 41, 93, 100, 120, 125, 127, 129, 133, 134, 141  
 automatyzacja, 123, 124, 130, 176  
 błędy, 126  
 GUI, *Patrz*: test akceptacyjny interfejsu użytkownika  
 interfejsu użytkownika, 128, 129  
 komunikacja, 122  
 tworzenie, 124  
 API, 128  
 automatyzacja, 133  
 badawczy, 132  
 choreografii, 135  
 hydrauliczny, 135  
 integracyjny, 135  
 narzędzia, *Patrz*: narzędzia test integracyjny  
 interfejsu użytkownika, 128, 129, 205  
 jednostkowy, 40, 41, 63, 99, 102, 127, 129, 133, 165  
 narzędzia, *Patrz*: narzędzia test jednostkowy  
 komponentów, 134, 141  
 narzędzia, *Patrz*: narzędzia test komponentów  
 manualny, 123, 136  
 optymistyczny, 132  
 pesymistyczny, 132  
 systemowy, 135  
 zestaw zautomatyzowany, 42  
 Test Driven Development, *Patrz*: TDD  
 tester, 169, 176, 177, 204  
 testowanie, 131, 133  
 TextMate, *Patrz*: edytor TextMate  
 Thomas Dave, 109  
 twierdzenie De Morgana, 183

**U**

UML, 45, 205

**W**

warunki krańcowe, 132, 134  
 wasa, 111  
 Watir, 134, 205  
 wideband delphi, 157, 158, 159  
 właściciel projektu, 178  
 wykres  
 Nassiego-Schneidermana, 44  
 struktury, 45  
 wypalenie zawodowe, 44  
 wzorzec projektowy, 45, 61, 190

**X**

XP, 97

**Z**

zaangażowanie, 72  
 zarządzanie czasem, 137, 144, 145, 147  
 zasada  
 niepewności, 117  
 pojedynczej odpowiedzialności, *Patrz*: SRP  
 zespół, *Patrz*: programowanie w zespole  
 zobowiązanie, 71, 72, 73, 74, 75, 151, 154, 160, 163  
 ukryte, 153  
 zone, *Patrz*: strefa

# PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ 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>