

Helion 



KOMPLETNY PRZEWODNIK

CIĄGŁE
DOSTARCZANIE
OPROGRAMOWANIA

EBERHARD WOLFF

Tytuł oryginału: A Practical Guide to Continuous Delivery

Tłumaczenie: Tomasz Walczak

ISBN: 978-83-283-3784-8

Authorized translation from the English language edition, entitled: A PRACTICAL GUIDE TO CONTINUOUS DELIVERY; ISBN 0134691474; by Eberhard Wolff; published by Pearson Education, Inc, publishing as Addison-Wesley Professional. Copyright © 2017 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 SA. Copyright © 2018.

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

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

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

Wydawnictwo HELION

ul. Kościuszki 1c, 44-100 GLIWICE

tel. 32 231 22 19, 32 230 98 63

e-mail: helion@helion.pl

WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/cidokp>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

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

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

Spis treści

Podziękowania	13
O autorze	14
Wprowadzenie	15
Część I. Podstawy	21
Rozdział 1. Ciągłe dostarczanie — co i jak	23
1.1. Wprowadzenie — czym jest ciągłe dostarczanie?	23
1.2. Dlaczego udostępnianie oprogramowania jest tak skomplikowane?	23
1.2.1. Ciągła integracja daje nadzieję	24
1.2.2. Powolne i ryzykowne procesy	24
1.2.3. Szybka praca jest możliwa	24
1.3. Wartość ciągłego dostarczania	24
1.3.1. Regularność	25
1.3.2. Możliwość śledzenia i sprawdzalność zmian	25
1.3.3. Regresja	26
1.4. Korzyści płynące z ciągłego dostarczania	26
1.4.1. Ciągłe dostarczanie w celu przyspieszenia udostępniania	26
1.4.2. Przykład	27
1.4.3. Implementowanie funkcji i udostępnianie jej w środowisku produkcyjnym	27
1.4.4. Przejście do następnej funkcji	27
1.4.5. Ciągłe dostarczanie zapewnia przewagę konkurencyjną	27
1.4.6. Scenariusz bez ciągłego dostarczania	28
1.4.7. Ciągłe dostarczanie i Lean Startup	28
1.4.8. Wpływ na proces rozwoju produktu	28
1.4.9. Minimalizowanie ryzyka za pomocą ciągłego dostarczania	29
1.4.10. Szybsze informacje zwrotne i podejście Lean	32

1.5. Procesy generowania i struktura potoku ciągłego dostarczania	32
1.5.1. Przykład	34
1.6. Wnioski	35
Rozdział 2. Zapewnianie infrastruktury	37
2.1. Wprowadzenie	37
2.1.1. <i>Automatyzowanie infrastruktury — przykład</i>	38
2.2. Skrypty instalacyjne	38
2.2.1. <i>Problemy związane ze standardowymi skryptami instalacyjnymi</i>	38
2.3. Chef	41
2.3.1. <i>Chef a Puppet</i>	41
2.3.2. <i>Inne możliwości</i>	42
2.3.3. <i>Podstawy techniczne</i>	43
2.3.4. <i>Chef Solo</i>	48
2.3.5. <i>Chef Solo — wnioski</i>	49
2.3.6. <i>Knife i Chef Server</i>	49
2.3.7. <i>Chef Server — wnioski</i>	52
2.4. Vagrant	53
2.4.1. <i>Przykład zastosowania Chefa i Vagranta</i>	54
2.4.2. <i>Vagrant — wnioski</i>	56
2.5. Docker	56
2.5.1. <i>Rozwiązanie oparte na Dockerze</i>	56
2.5.2. <i>Tworzenie kontenerów Dockera</i>	58
2.5.3. <i>Uruchamianie przykładowej aplikacji za pomocą Dockera</i>	60
2.5.4. <i>Docker i Vagrant</i>	62
2.5.5. <i>Docker Machine</i>	63
2.5.6. <i>Tworzenie złożonych konfiguracji za pomocą Dockera</i>	65
2.5.7. <i>Docker Compose</i>	67
2.6. Niemodyfikowalny serwer	69
2.6.1. <i>Wady idempotencji</i>	69
2.6.2. <i>Serwer niemodyfikowalny i Docker</i>	69
2.7. Infrastruktura jako kod	70
2.7.1. <i>Testowanie infrastruktury w postaci kodu</i>	71
2.8. Platforma jako usługa	72
2.9. Obsługa danych i baz	73
2.9.1. <i>Zarządzanie schematami</i>	74
2.9.2. <i>Dane testowe i główne</i>	75
2.10. Wnioski	76

Część II. Potok ciągłego dostarczania	77
Rozdział 3. Automatyzacja procesu budowania i ciągła integracja	79
3.1. Wprowadzenie	79
3.1.1. <i>Automatyzacja procesu budowania — przykład</i>	79
3.2. Automatyzowanie procesu budowania i narzędzia do budowania	80
3.2.1. <i>Narzędzia do budowania w świecie Javy</i>	80
3.2.2. <i>Ant</i>	81
3.2.3. <i>Maven</i>	82
3.2.4. <i>Gradle</i>	86
3.2.5. <i>Dodatkowe narzędzia do budowania</i>	88
3.2.6. <i>Wybór odpowiedniego narzędzia</i>	89
3.3. Testy jednostkowe	90
3.3.1. <i>Pisanie dobrych testów jednostkowych</i>	91
3.3.2. <i>Programowanie sterowane testami</i>	93
3.3.3. <i>Ruchy Clean Code i Software Craftsmanship</i>	94
3.4. Ciągła integracja	94
3.4.1. <i>Jenkins</i>	95
3.4.2. <i>Infrastruktura do ciągłej integracji</i>	100
3.4.3. <i>Wnioski</i>	101
3.5. Pomiar jakości kodu	103
3.5.1. <i>SonarQube</i>	104
3.6. Zarządzanie artefaktami	106
3.6.1. <i>Integracja z procesem budowania</i>	108
3.6.2. <i>Zaawansowane funkcje repozytoriów</i>	110
3.7. Wnioski	111
Rozdział 4. Testy akceptacyjne	113
4.1. Wprowadzenie	113
4.1.1. <i>Testy akceptacyjne — przykład</i>	113
4.2. Piramida testów	114
4.3. Czym są testy akceptacyjne?	116
4.3.1. <i>Zautomatyzowane testy akceptacyjne</i>	117
4.3.2. <i>Więcej niż wzrost wydajności</i>	117
4.3.3. <i>Testy ręczne</i>	118
4.3.4. <i>A co z klientem?</i>	118
4.3.5. <i>Testy akceptacyjne a testy jednostkowe</i>	118
4.3.6. <i>Środowiska testowe</i>	119
4.4. Testy akceptacyjne oparte na interfejsie GUI	120
4.4.1. <i>Problemy z testami z użyciem interfejsu GUI</i>	120
4.4.2. <i>Stosowanie abstrakcji</i>	120
4.4.3. <i>Automatyzacja z użyciem narzędzia Selenium</i>	121
4.4.4. <i>Interfejs API WebDriver</i>	121

4.4.5. Testy bez użycia przeglądarki — HtmlUnit	121
4.4.6. Interfejs API WebDriver dla Selenium	121
4.4.7. Środowisko IDE dla Selenium	121
4.4.8. Problemy ze zautomatyzowanymi testami interfejsu GUI	123
4.4.9. Wykonywanie testów z użyciem interfejsu GUI	123
4.4.10. Eksportowanie testów jako kodu	123
4.4.11. Ręczne modyfikowanie przypadków testowych	123
4.4.12. Dane testowe	124
4.4.13. Obiekty reprezentujące strony	124
4.5. Inne narzędzia do przeprowadzania testów z użyciem interfejsu GUI	125
4.5.1. PhantomJS	125
4.5.2. Windmill	125
4.6. Tekstowe testy akceptacyjne	126
4.6.1. Podejście BDD	126
4.6.2. Różne adaptery	128
4.7. Inne platformy	129
4.8. Strategie przeprowadzania testów akceptacyjnych	130
4.8.1. Właściwe narzędzie	131
4.8.2. Błyskawiczne informacje zwrotne	131
4.8.3. Pokrycie testami	131
4.9. Wnioski	132
Rozdział 5. Testy wydajności	133
5.1. Wprowadzenie	133
5.1.1. Testy wydajności — przykład	133
5.2. Testy wydajności — jak je przeprowadzać?	133
5.2.1. Cele testów wydajności	134
5.2.2. Środowiska i ilość danych	134
5.2.3. Testy szybkości tylko po zakończeniu implementowania kodu?	134
5.2.4. Testy wydajności = zarządzanie ryzykiem	135
5.2.5. Symulowanie działań użytkowników	135
5.2.6. Dokumentowanie wymogów związanych z szybkością	135
5.2.7. Sprzęt potrzebny w testach wydajności	135
5.2.8. Chmura i wirtualizacja	136
5.2.9. Minimalizowanie ryzyka dzięki ciągłemu testowaniu	136
5.2.10. Testy wydajności — sensowne czy nie?	137
5.3. Implementowanie testów wydajności	137
5.4. Testy wydajności z użyciem narzędzia Gatling	139
5.4.1. Wersja demonstracyjna a praktyka	142
5.5. Narzędzia używane zamiast Gatlinga	143
5.5.1. Grinder	143
5.5.2. Apache JMeter	144

5.5.3. Tsung	144
5.5.4. Rozwiązania komercyjne	144
5.6. Wnioski	145
Rozdział 6. Testy eksploracyjne	147
6.1. Wprowadzenie	147
6.1.1. Testy eksploracyjne — przykład	147
6.2. Po co stosować testy eksploracyjne?	147
6.2.1. Czasem testy ręczne i tak są lepsze	148
6.2.2. Testy z udziałem klientów	148
6.2.3. Testy ręczne wymagań нефunkcjonalnych	148
6.3. Jak zabrać się za testy eksploracyjne?	149
6.3.1. Zadanie określa testy	149
6.3.2. Zautomatyzowane środowisko	149
6.3.3. Przypadki pokazowe jako punkt wyjścia	149
6.3.4. Przykład — aplikacja z obszaru handlu elektronicznego	149
6.3.5. Testy wersji beta	150
6.3.6. Testy oparte na sesji	150
6.4. Wnioski	152
Rozdział 7. Wdrażanie — udostępnianie w środowisku produkcyjnym	153
7.1. Wprowadzenie	153
7.1.1. Wdrażanie — przykład	154
7.2. Udostępnianie i wycofywanie	154
7.2.1. Korzyści	154
7.2.2. Wady	154
7.3. Zastępowanie nową wersją	155
7.3.1. Korzyści	155
7.3.2. Wady	155
7.4. Wdrażanie w modelu niebieskie-zielone	156
7.4.1. Korzyści	156
7.4.2. Wady	156
7.5. Udostępnianie „kanarkowe”	157
7.5.1. Korzyści	158
7.5.2. Wady	158
7.6. Ciągłe wdrażanie	158
7.6.1. Korzyści	159
7.6.2. Wady	160
7.7. Wirtualizacja	160
7.7.1. Hosty fizyczne	161
7.8. Poza aplikacje sieciowe	161
7.9. Wnioski	162

Rozdział 8. Eksploatacja	163
8.1. Wprowadzenie	163
8.1.1. <i>Eksploatacja — przykład</i>	163
8.2. Trudności z eksploatacją oprogramowania	164
8.3. Pliki dziennika	165
8.3.1. <i>Co należy rejestrować?</i>	165
8.3.2. <i>Narzędzia do przetwarzania plików dziennika</i>	167
8.3.3. <i>Rejestrowanie danych w przykładowej aplikacji</i>	168
8.4. Analizowanie dzienników przykładowej aplikacji	169
8.4.1. <i>Analizowanie danych za pomocą Kibany</i>	171
8.4.2. <i>ELK — skalowalność</i>	172
8.5. Inne technologie obsługi dzienników	175
8.6. Zaawansowane techniki rejestrowania dzienników	176
8.6.1. <i>Anonimizacja</i>	176
8.6.2. <i>Szybkość działania</i>	176
8.6.3. <i>Czas</i>	176
8.6.4. <i>Operacyjna baza danych</i>	177
8.7. Monitorowanie	177
8.8. Pomiary z użyciem narzędzia Graphite	177
8.9. Pomiary w przykładowej aplikacji	179
8.9.1. <i>Struktura przykładu</i>	179
8.10. Inne rozwiązania z obszaru monitorowania	181
8.11. Inne wyzwania związane z eksploatacją aplikacji	182
8.11.1. <i>Skrypty</i>	182
8.11.2. <i>Aplikacje w centrum danych klienta</i>	183
8.12. Wnioski	183

Część III. Zarządzanie, kwestie organizacyjne i architektura w obszarze ciągłego dostarczania

Rozdział 9. Wprowadzanie ciągłego dostarczania w organizacji	187
9.1. Wprowadzenie	187
9.2. Ciągłe dostarczanie od początku projektu	187
9.3. Odwzorowywanie strumienia wartości	188
9.3.1. <i>Odwzorowywanie strumienia wartości pozwala opisać sekwencję zdarzeń</i>	188
9.3.2. <i>Optymalizacje</i>	189
9.4. Dodatkowe sposoby optymalizacji	190
9.4.1. <i>Inwestycje wysokiej jakości</i>	190
9.4.2. <i>Koszty</i>	190
9.4.3. <i>Korzyści</i>	191
9.4.4. <i>Nie dodawaj kodu, gdy proces budowania kończy się niepowodzeniem!</i>	191

9.4.5. Zatrzymywanie taśmy	191
9.4.6. Pięć pytań „dlaczego”	192
9.4.7. DevOps	192
9.5. Wnioski	193
Rozdział 10. Ciągłe dostarczanie i DevOps	195
10.1. Wprowadzenie	195
10.2. Czym jest model DevOps?	195
10.2.1. Problemy	195
10.2.2. Perspektywa klienta	196
10.2.3. Pionierska firma — Amazon	196
10.2.4. DevOps	197
10.3. Ciągłe dostarczanie i DevOps	198
10.3.1. DevOps — więcej niż ciągłe dostarczanie	198
10.3.2. Pełna odpowiedzialność i samoorganizowanie się	199
10.3.3. Decyzje z obszaru technologii	199
10.3.4. Mniej scentralizowanej kontroli	200
10.3.5. „Pluralizm” w obszarze technologii	200
10.3.6. Wymiana między zespołami	200
10.3.7. Architektura	201
10.4. Ciągłe dostarczanie bez modelu DevOps?	202
10.4.1. Końcowa część potoku ciągłego dostarczania	202
10.5. Wnioski	203
Rozdział 11. Ciągłe dostarczanie, DevOps i architektura oprogramowania	205
11.1. Wprowadzenie	205
11.2. Architektura oprogramowania	205
11.2.1. Po co tworzyć architekturę oprogramowania?	206
11.3. Optymalizowanie architektury pod kątem ciągłego dostarczania	207
11.3.1. Mniejsze jednostki wdrażania	208
11.4. Interfejsy	209
11.4.1. Prawo Postela lub zasada odporności	210
11.4.2. Projektowanie pod kątem niepowodzeń	210
11.4.3. Stan	211
11.5. Bazy danych	211
11.5.1. Zapewnianie stabilności baz danych	211
11.5.2. Baza danych = komponent	212
11.5.3. Widoki i procedury składowane	212
11.5.4. Baza danych dla komponentu	213
11.5.5. Bazy typu NoSQL	213

11.6. Mikrouslugi	213
11.6.1. Mikrouslugi i ciągle dostarczanie	214
11.6.2. Wprowadzanie ciągłego dostarczania z użyciem mikrouslug	214
11.6.3. Mikrouslugi wymagają ciągłego dostarczania	214
11.6.4. Struktura organizacyjna	215
11.7. Radzenie sobie z nowymi funkcjami	215
11.7.1. Odgałęzienia kodu funkcji	215
11.7.2. Przełączniki funkcji	216
11.7.3. Korzyści	216
11.7.4. Przykłady zastosowań przełączników funkcji	217
11.7.5. Wady	217
11.8. Wnioski	218
Rozdział 12. Wnioski — jakie korzyści wynikają z ciągłego dostarczania?	219
Skorowidz	221

Rozdział 1

Ciągłe dostarczanie — co i jak

1.1. Wprowadzenie — czym jest ciągłe dostarczanie?

Niełatwo jest odpowiedzieć na to pytanie. Autorzy nazwy nie przedstawiają konkretnej definicji¹. Martin Fowler w swoim omówieniu² ciągłego dostarczania koncentruje się na tym, że oprogramowanie można w dowolnym momencie udostępnić w środowisku produkcyjnym. Wymaga to automatyzacji procesów potrzebnych do instalacji oprogramowania i informacji zwrotnych dotyczących jakości oprogramowania. W Wikipedii³ ciągłe dostarczanie zdefiniowane jest jako optymalizacja i automatyzacja procesu udostępniania oprogramowania.

W podsumowaniu: głównym celem ciągłego dostarczania jest analiza i optymalizacja procesu prowadzącego do udostępnienia oprogramowania. Proces ten zostaje płynnie zintegrowanym z rozwojem produktu.

1.2. Dlaczego udostępnianie oprogramowania jest tak skomplikowane?

Udostępnianie oprogramowania to wyzwanie. Prawdopodobnie każdemu działowi informatycznemu zdarzyło się pracować w weekend, aby udostępnić oprogramowanie w środowisku produkcyjnym. Takie sytuacje często kończą się tym, że oprogramowanie jest jakoś umieszczane w środowisku produkcyjnym, ponieważ od określonego momentu przywrócenie starszej wersji okazuje się jeszcze bardziej ryzykowne i trudniejsze niż podążanie naprzód. Jednak po zainstalowaniu wersji często następuje długa faza, kiedy oprogramowanie trzeba ustabilizować.

¹ Jez Humble, David Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, Addison-Wesley, 2010, ISBN 978-0-32160-191-9.

² Zob. <https://martinfowler.com/bliki/ContinuousDelivery.html>.

³ Zob. https://en.wikipedia.org/wiki/Continuous_delivery.

1.2.1. Ciągła integracja daje nadzieję

Obecnie to właśnie udostępnianie oprogramowania w środowisku produkcyjnym stanowi wyzwanie. Nie tak dawno temu problemy zaczynały się znacznie wcześniej. Poszczególne zespoły pracowały nad modułami niezależnie od siebie, dlatego przed udostępnieniem kodu różne komponenty trzeba było zintegrować. Gdy moduły były łączone po raz pierwszy, system często nawet się nie kompilował. Nieraz zintegrowanie wszystkich zmian i doprowadzenie do udanej kompilacji zajmowało dni, a nawet tygodnie. Dopiero wtedy można było zacząć wdrażanie. Obecnie te problemy w większości zostały rozwiązane. Wszystkie zespoły pracują nad wspólną wersją kodu, która jest automatycznie stale integrowana, kompilowana i testowana. To podejście nazywane jest ciągłą integracją (ang. *continuous integration*). Infrastrukturę potrzebną do ciągłej integracji opisano w rozdziale 3. „Automatyzacja procesu budowania i ciągła integracja”. Wylimitowanie dawnych problemów związanych z tym etapem rodzi nadzieję na to, że pojawią się rozwiązania trudności z innymi etapów przedprodukcyjnych.

1.2.2. Powolne i ryzykowne procesy

Procesy z dalszych etapów są często złożone i skomplikowane. Ponadto ręczne wykonywanie zadań sprawia, że procesy są bardzo żmudne i narażone na błędy. Jest tak w przypadku udostępniania kodu w środowisku produkcyjnym, ale też we wcześniejszych fazach, np. testów. Błędy często pojawiają się zwłaszcza w ręcznych procesach, które są wykonywane tylko kilka razy w roku, co dodatkowo pogarsza sytuację. To oczywiście zwiększa ryzyko niepowodzenia całej procedury udostępniania.

Z powodu wysokiego ryzyka błędów i złożoności wersje oprogramowania są rzadko udostępniane w środowisku produkcyjnym. Wynikający z tego brak praktyki dodatkowo wydłuża procesy. Trudno jest też je zoptymalizować.

1.2.3. Szybka praca jest możliwa

Zawsze możliwe jest szybkie udostępnienie wersji w środowisku produkcyjnym w sytuacji awaryjnej — na przykład gdy trzeba pilnie naprawić błąd. Jednak w takiej sytuacji pomijane są wszystkie testy, a tym samym wszystkie zabezpieczenia będące integralną częścią standardowego procesu. Jest to oczywiście ryzykowne. To, że standardowo takie testy są przeprowadzane, ma solidne uzasadnienie.

Dlatego normalna droga udostępniania kodu w środowisku produkcyjnym jest powolna i ryzykowna. W sytuacjach awaryjnych można ją przyspieszyć kosztem jeszcze większego wzrostu ryzyka.

1.3. Wartość ciągłego dostarczania

Celem jest optymalizacja udostępniania wersji w środowisku produkcyjnym z uwzględnieniem technik ciągłej integracji i motywacji związanej z tym podejściem.

Podstawowa zasada ciągłej integracji brzmi: „Jeśli coś boli, rób to częściej i uświadom sobie ten ból”. Choć wygląda to na masochizm, w rzeczywistości jest sposobem rozwiązywania problemów. Zamiast unikać problemów z wersjami, udostępniając ich jak najmniej, procesy należy stosować tak często i wcześnie, jak jest to możliwe, by jak najszybciej je zoptymalizować

(z uwzględnieniem tempa pracy i niezawodności). Dlatego ciągle dostarczanie zmusza firmę do zmian i przyjęcia nowego modelu pracy.

Ostatecznie podejście to nie jest zaskakujące. Każda firma informatyczna potrafi błyskawicznie udostępnić poprawki w środowisku produkcyjnym. W takim scenariuszu często przeprowadza się tylko część standardowych testów i sprawdzianów bezpieczeństwa. Jest to możliwe, ponieważ zmiany są niewielkie, dlatego oznaczają małe ryzyko. Widoczne staje się tu inne podejście minimalizowania ryzyka — zamiast próbować zabezpieczać się przed błędami za pomocą złożonych procesów i rzadkiego udostępniania, można częściej udostępnić w środowisku produkcyjnym niewielkie zmiany. To podejście jest identyczne ze strategią ciągłego dostarczania. W ciągłym dostarczaniu nawet drobne zmiany oprogramowania wprowadzone przez poszczególnych programistów i zespół są trwale integrowane. Różni się to od niezależnej pracy zespołów i programistów przez wiele dni i tygodni, po których dopiero na końcu integrowane są wszystkie zakumulowane modyfikacje. Ta ostatnia strategia często prowadzi do poważnych problemów. W niektórych sytuacjach są one tak duże, że oprogramowania w ogóle nie da się skompilować.

Jednak ciągle dostarczanie to coś więcej niż „szybkie udostępnianie drobnych zmian”. Podejście to jest oparte na innych wartościach, z których wynikają konkretne rozwiązania techniczne.

1.3.1. Regularność

Regularność oznacza, że procesy są wykonywane częściej. Wszystkie procesy potrzebne do udostępnienia oprogramowania w środowisku produkcyjnym należy wykonywać regularnie, a nie tylko w momencie udostępniania wersji. Na przykład konieczne jest budowanie środowisk testowych i przedprodukcyjnych (ang. *staging*). Środowiska testowe można wykorzystać do wykonywania testów akceptacyjnych i technicznych. Środowiska przedprodukcyjne mogą posłużyć do testowania i oceny funkcji z nowej wersji przez docelowego klienta. Proces generowania środowisk można przekształcić w standardowe zadanie wykonywane nie tylko w sytuacji, gdy trzeba utworzyć środowisko produkcyjne. Aby generować wiele środowisk bez dużego wysiłku, procesy trzeba zautomatyzować. Regularność prowadzi do automatyzacji. Podobne reguły dotyczą testów. Nie ma sensu odkładać niezbędnych testów do momentu przed udostępnieniem wersji. Zamiast tego należy je przeprowadzać regularnie. Także to podejście wymusza automatyzację w celu ograniczenia niezbędnego wysiłku. Regularność prowadzi też do wzrostu niezawodności. Często stosowane procesy można niezawodnie powtarzać i wykonywać.

1.3.2. Możliwość śledzenia i sprawdzalność zmian

Wszystkie zmiany w oprogramowaniu, które mają zostać wprowadzone w środowisku produkcyjnym i infrastrukturze potrzebnej dla danej wersji kodu, muszą być możliwe do prześledzenia. Potrzebna jest możliwość odtworzenia stanu oprogramowania i infrastruktury. To prowadzi do systemu kontroli wersji uwzględniającego nie tylko oprogramowanie, ale też potrzebne środowiska. W idealnych warunkach możliwe jest wygenerowanie każdego stanu oprogramowania wraz z potrzebnym do jego pracy środowiskiem i odpowiednią konfiguracją. Zapewnia to możliwość prześledzenia wszystkich zmian w oprogramowaniu i środowisku oraz umożliwia łatwe uzyskanie systemu do analizy błędów. Ponadto zmiany można dokumentować lub audytować.

Jednym z rozwiązań jest zapewnienie dostępu do środowisk produkcyjnego i przedprodukcyjnego tylko wybranym osobom. Ma to zapobiegać powstawaniu „szybkich poprawek”, które

nie są udokumentowane i których śledzenie nie jest możliwe. Ponadto kwestie bezpieczeństwa i zabezpieczanie danych to argumenty na rzecz blokowania dostępu do środowiska produkcyjnego.

Gdy stosujesz ciągłe dostarczanie, interwencje w środowisko są możliwe tylko po zmianie skryptu instalacyjnego. Modyfikacje w skryptach są możliwe do śledzenia dzięki systemowi kontroli wersji. Twórcy skryptów nie mają dostępu do danych produkcyjnych, dlatego nie występuje problem z bezpieczeństwem danych.

1.3.3. Regresja

Aby zminimalizować ryzyko związane z udostępnianiem oprogramowania w środowisku produkcyjnym, oprogramowanie trzeba przetestować. Oczywiście poprzez użycie testów trzeba się upewnić, że nowe funkcje działają poprawnie. Wiele pracy wymaga unikanie regresji, czyli błędów powstałych w trakcie modyfikowania już przetestowanych fragmentów oprogramowania. Modyfikacja wymaga więc ponownego przeprowadzenia wszystkich testów, ponieważ zmiany w jednym miejscu systemu mogą spowodować błędy w innych. Dlatego konieczne są zautomatyzowane testy. W przeciwnym razie wykonywanie ich wymaga zbyt wiele wysiłku. Jeśli mimo testów w środowisku produkcyjnym wystąpi błąd, można go wykryć dzięki monitorowaniu. W idealnych warunkach możliwe jest proste zainstalowanie wolnej od błędów starszej wersji systemu produkcyjnego (wycofanie zmian, ang. *rollback*) lub szybkie wprowadzenie poprawki w środowisku produkcyjnym (zastąpienie nowszą wersją, ang. *roll forward*). Chodzi o to, by uzyskać system wczesnego ostrzegania, który dokonuje pomiarów na różnych etapach projektu (np. w czasie testów i w środowisku produkcyjnym), aby wykrywać regresję i umożliwiać jej wyeliminowanie.

1.4. Korzyści płynące z ciągłego dostarczania

Ciągłe dostarczanie zapewnia wiele korzyści. W zależności od scenariusza różne korzyści mogą mieć inną wagę i wpływać na sposób stosowania ciągłego dostarczania.

1.4.1. Ciągłe dostarczanie w celu przyspieszenia udostępniania

Ciągłe dostarczanie skraca czas potrzebny do udostępnienia zmian w środowisku produkcyjnym. Zapewnia to istotne korzyści biznesowe, ponieważ znacznie łatwiej jest reagować na zmiany na rynku.

Jednak zalety wykraczają poza szybsze udostępnianie. Nowe podejścia takie jak Lean Startup⁴ zalecają strategię, która pozwala odnieść jeszcze większe korzyści ze wzrostu szybkości prac. W modelu Lean Startup celem jest udostępnianie produktów na rynku i ocenianie ich szans, przy czym inwestowany wysiłek ma być jak najmniejszy. Podobnie jak w eksperymentach naukowych z góry definiowane jest, jak oceniany będzie sukces produktu. Następnie eksperyment jest przeprowadzany, po czym następuje ocena sukcesu (lub porażki).

⁴ Eric Ries, *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*, Crown Business, 2010, ISBN 978-0-67092-160-7; wyd. polskie: *Metoda Lean Startup. Wykorzystaj innowacyjne narzędzia i stwórz firmę, która zdoła wygrać rynek*, Onepress, 2012.

1.4.2. Przykład

Przyjrzyj się teraz konkretnemu przykładowi. W sklepie internetowym utworzona ma zostać nowa funkcja — dostarczanie zamówień w określonym dniu. W ramach pierwszego eksperymentu można zacząć reklamować nową funkcję. Liczba kliknięć odsyłacza z reklamy może posłużyć do oceny sukcesu tego eksperymentu. Na tym etapie oprogramowanie nie zostało jeszcze zbudowane — funkcja nie jest jeszcze zaimplementowana. Jeśli eksperyment nie daje obiecujących wyników, funkcja najwyraźniej nie daje korzyści i można — nie marnując dużo energii — priorytetowo potraktować inne mechanizmy.

1.4.3. Implementowanie funkcji i udostępnianie jej w środowisku produkcyjnym

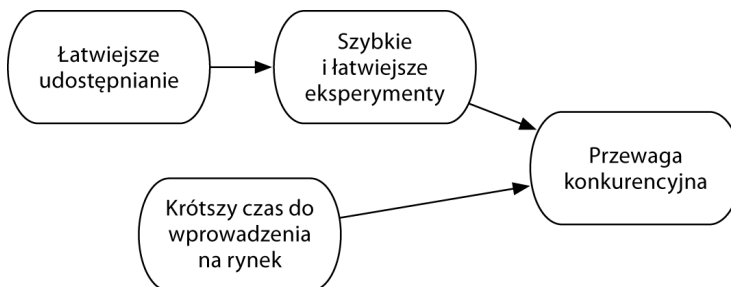
Jeśli eksperyment zakończył się powodzeniem, funkcja jest implementowana i udostępniana w środowisku produkcyjnym. Nawet ten krok można przeprowadzić jako eksperyment. Wskaźniki mogą pomóc w ocenie powodzenia funkcji. Możesz na przykład mierzyć liczbę zamówień o ustalonej dacie dostarczenia.

1.4.4. Przejście do następnej funkcji

Analizy wskaźników pokazują, że liczba zamówień jest wystarczająco duża. Co ciekawe, większość z nich nie jest kierowana do nabywcy, ale do osób trzecich. Z dodatkowych pomiarów wynika, że zamawiane towary to najwyraźniej prezenty urodzinowe. Na podstawie tych informacji można rozbudować funkcję, np. o kalendarz z urodzinami i polecane prezenty. Wymaga to oczywiście zaprojektowania dodatkowych funkcji, zaimplementowania ich, udostępnienia w środowisku produkcyjnym i oceny ich sukcesu. Inna możliwość to ocena potencjalnego sukcesu rynkowego takich funkcji bez ich implementowania. Można posłużyć się do tego reklamami, wiadomami z klientami, ankietami i innymi technikami.

1.4.5. Ciągłe dostarczanie zapewnia przewagę konkurencyjną

Ciągłe dostarczanie umożliwia szybsze wprowadzanie niezbędnych zmian w oprogramowaniu w środowisku produkcyjnym. Pozwala to firmie na szybsze testowanie różnych pomysłów i rozwijanie modelu biznesowego. To zapewnia przewagę konkurencyjną. Ponieważ można sprawdzić więcej pomysłów, łatwiej jest ustalić te właściwe — i to nie na podstawie subiektywnych szacunków prawdopodobieństwa sukcesu rynkowego, ale według obiektywnie zmierzonych danych (rysunek 1.1).



Rysunek 1.1. Argumenty na rzecz stosowania ciągłego dostarczania w startupie

1.4.6. Scenariusz bez ciągłego dostarczania

Bez ciągłego dostarczania funkcję dostaw w określonym dniu trzeba zaplanować i umieścić w środowisku produkcyjnym w czasie udostępniania następnej wersji — prawdopodobnie za kilka miesięcy. Do tego czasu dział marketingu zapewne nie odważy się reklamować funkcji, ponieważ długi czas do jej udostępnienia sprawia, że takie reklamy są bezcelowe. Gdyby funkcja nie okazała się sukcesem, jej zaimplementowanie wymagałoby poniesienia wysokich kosztów bez zapewniania korzyści. W tym tradycyjnym podejściu ocena powodzenia nowej funkcji też oczywiście jest możliwa, jednak informacje są uzyskiwane znacznie wolniej. Efekty dalszych prac, na przykład funkcje ułatwiające zakupy prezentów urodzinowych, trafiają na rynek dużo później, ponieważ wymagają ponownego umieszczenia oprogramowania w środowisku produkcyjnym i przejścia drugi raz przez czasochłonny proces udostępniania. Ponadto wątpliwe jest, czy w tym podejściu powodzenie funkcji da się przeanalizować wystarczająco szczegółowo, aby odkryć potencjał związany z dodatkowymi mechanizmami wspomagającymi zakupy prezentów urodzinowych.

1.4.7. Ciągłe dostarczanie i Lean Startup

Dzięki ciągłemu dostarczaniu cykle optymalizacji są znacznie krótsze, ponieważ każdą funkcję można umieścić w środowisku produkcyjnym w niemal dowolnym czasie, co pozwala na stosowanie podejść takich jak Lean Startup. Wpływa to na pracę działów biznesowych, które muszą szybciej wymyślać nowe funkcje i nie muszą koncentrować się na długoterminowych planach, ponieważ mogą błyskawicznie reagować na wyniki obecnie przeprowadzanych eksperymentów. Jest to łatwe przede wszystkim w startupach, jednak omawiane podejście można też zastosować także w tradycyjnych firmach. Podejście Lean Startup ma, niestety, mylącą nazwę. Dotyczy ono wprowadzania produktów na rynek za pomocą serii eksperymentów, co jest oczywiście możliwe nie tylko w startupach, ale i w zwykłych firmach. To podejście można też stosować do produktów, które są sprzedawane w tradycyjny sposób (np. na nośnikach takich jak płyty CD), wymagają złożonych procedur instalacji lub są częścią innego produktu (np. maszyny). Wtedy instalację oprogramowania trzeba uprościć, a najlepiej zautomatyzować. Oprócz określenia grupy klientów trzeba ustalić, kto chciałby przetestować nowe wersje oprogramowania, i przekazać informacje na ich temat. Osoby takie to klasyczni testerzy wersji beta i zaawansowani użytkownicy.

1.4.8. Wpływ na proces rozwoju produktu

Ciągłe dostarczanie wpływa na proces rozwoju oprogramowania. Gdy firma chce udostępniać w środowisku produkcyjnym pojedyncze funkcje, proces musi to umożliwiać. W niektórych procesach iteracje trwają tydzień lub kilka tygodni. Po zakończeniu każdej iteracji w środowisku produkcyjnym udostępniana jest nowa wersja z kilkoma nowymi funkcjami. Nie jest to idealne rozwiązanie dla ciągłego dostarczania, ponieważ funkcje nie mogą pojedynczo przepływać przez potok. Taki model utrudnia też stosowanie podejścia Lean Startup. Gdy kilka funkcji jest udostępnianych jednocześnie, nie jest oczywiste, która zmiana wpływa na uzyskane wyniki. Załóżmy, że opcja dostawy w określonym dniu jest udostępniana równoległe ze zmianą kosztów dostawy. Nie da się wtedy stwierdzić, która z tych zmian wywarła większy wpływ na wzrost liczby sprzedanych produktów.

Dlatego procesy takie jak Scrum, programowanie ekstremalne (ang. *extreme programming* — XP) i oczywiście model kaskadowy są przeszkodą, ponieważ zawsze oznaczają udostępnianie w środowisku produkcyjnym kilku funkcji. Natomiast Kanban⁵ polega na etapowym wprowadzaniu jednej funkcji do środowiska produkcyjnego i jest w pełni zgodny z ciągłym dostarczaniem. Inne procesy można oczywiście dostosować, aby umożliwić dostarczanie pojedynczych funkcji. Oznacza to jednak modyfikację procesów, przez co nie są one stosowane we wzorcowy sposób. Inna możliwość to dezaktywacja na początku dodatkowych funkcji. Pozwala to udostępnić razem zbiór funkcji z jednej wersji w środowisku produkcyjnym, a przy tym mierzyć skutki wprowadzenia każdej z nich z osobna.

Ciągłe dostarczanie oznacza też, że w zespołach pracują osoby pełniące różne role. Oprócz ról związanych z rozwojem produktu i eksploatacją mogą to być także role biznesowe dotyczące na przykład marketingu. Dzięki zmniejszeniu w ten sposób przeszkód w organizacji można jeszcze szybciej projektować eksperymenty na podstawie informacji zwrotnych dotyczących aspektów biznesowych.

Wypróbuj i poeksperymentuj

- Poszukaj informacji na temat podejść Lean Startup i Kanban. Skąd pochodzi Kanban?

Wybierz znany Ci projekt lub znajomą funkcję:

- Jak wyglądałaby minimalna wersja produktu? Powinna ona dać wyobrażenie o szansach rynkowych planowanego kompletnego produktu.
- Czy można ocenić produkt bez gotowego oprogramowania? Czy możliwe jest na przykład zareklamowanie go? Czy możliwe są wywiady z potencjalnymi użytkownikami?
- Jak można zmierzyć sukces funkcji? Czy ma ona wpływ na sprzedaż, liczbę kliknięć lub inne mierzalne aspekty?
- Ile czasu dział marketingu i sprzedaży mają na zaplanowanie produktu lub funkcji? Na ile jest to zgodne z podejściem Lean Startup?

1.4.9. Minimalizowanie ryzyka za pomocą ciągłego dostarczania

Stosowanie ciągłego dostarczania w sposób opisany w poprzednim punkcie jest związane z określonym modelem biznesowym. Jednak tradycyjne firmy często wymagają długoterminowych planów. Wtedy nie można stosować podejścia takiego jak Lean Startup. Ponadto w wielu firmach czas do wprowadzenia produktu na rynek jest stosunkowo mało istotny. Nie we wszystkich branżach ten aspekt ma kluczowe znaczenie. Może się to oczywiście zmienić, gdy firmy zaczną mierzyć się z konkurencją, która potrafi wejść na rynek z wykorzystaniem podejścia Lean Startup.

W wielu scenariuszach czas do wejścia na rynek nie jest motywacją do zastosowania ciągłego dostarczania. Jednak techniki z tego podejścia i tak mogą okazać się przydatne, ponieważ zapewnia ono wiele dodatkowych korzyści.

⁵ David J. Anderson, *Kanban: Successful Evolutionary Change for Your Technology Business*, Blue Hole Press, 2010, ISBN 978-0-98452-140-1.

- Ręczne procesy udostępniania wymagają wiele wysiłku. Nierzadko zdarza się, że cały dział informatyczny musi przez cały weekend pracować nad udostępnieniem kodu. Ponadto po zakończeniu tego procesu często trzeba wykonać wiele prac dodatkowych.
- Ponadto ryzyko jest wysokie. Udostępnianie oprogramowania wymaga wielu ręcznych modyfikacji, co może łatwo prowadzić do pomyłek. Jeśli błędy nie zostaną szybko wykryte i poprawione, mogą mieć daleko idące konsekwencje dla firmy.

Koszty tego ponoszą pracownicy działów informatycznych — programiści i administratorzy systemów, którzy pracują w weekendy i nocami, by udostępnić kod w środowisku produkcyjnym i poprawić błędy. Oprócz pracy w nadgodzinach są też poddani dużemu stresowi z powodu wysokiego ryzyka, którego nie należy lekceważyć. Na przykład firma Knight Capital straciła 440 milionów dolarów z powodu nieudanego udostępniania oprogramowania⁶. Wskutek tego firma stała się niewypłacalna. Takie scenariusze prowadzą do wielu pytań⁷ — zwłaszcza o to, dlaczego problem wystąpił, dlaczego szybko go nie wykryto i jak można zapobiec podobnym zdarzeniom w innych środowiskach.

Rozwiązaniem takich problemów może być ciągłe dostarczanie. Podstawowymi aspektami tego podejścia są wyższa niezawodność i jakość procesu udostępniania. Dzięki temu programiści i administratorzy systemu mogą dosłownie spać spokojnie. Korzyści te wynikają z różnych czynników.

- Dzięki wyższemu poziomowi automatyzacji procesów udostępniania ich efekty łatwo jest powtórzyć. Dlatego gdy oprogramowanie zostanie wdrożone i przetestowane w środowisku testowym lub przedprodukcyjnym, dokładnie ten sam efekt można uzyskać w środowisku produkcyjnym, ponieważ jest ono identyczne z pozostałymi. Pozwala to wyeliminować wiele źródeł błędów, co prowadzi do zmniejszenia ryzyka.
- Ponadto testowanie oprogramowania staje się znacznie łatwiejsze, ponieważ testy są w dużym stopniu zautomatyzowane. To dodatkowo poprawia jakość, ponieważ testy można wykonywać częściej.
- Gdy częstotliwość wdrażania rośnie, ryzyko spada, ponieważ przy każdym wdrożeniu liczba zmian wprowadzanych w środowisku produkcyjnym jest mniejsza. Mniej zmian oznacza mniejsze ryzyko pojawienia się błędów.

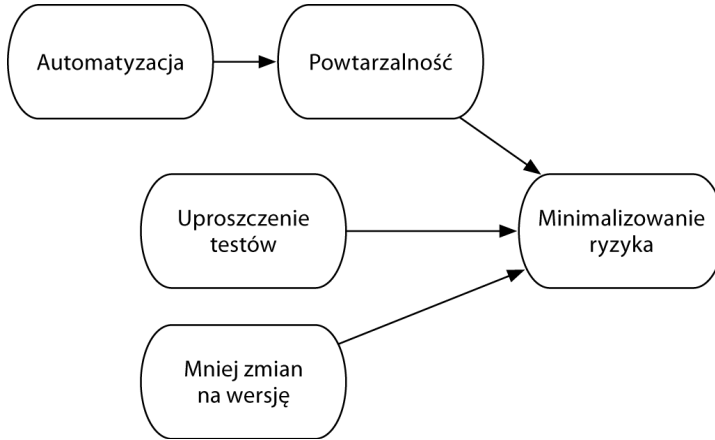
Sytuacja jest w pewien sposób paradoksalna — tradycyjne działy informatyczne starają się udostępnić wersje w środowisku produkcyjnym jak najrzadziej, ponieważ proces ten jest wysoce ryzykowny. W trakcie każdego udostępniania mogą pojawić się usterki o potencjalnie katastrofalnych skutkach. Mniej wersji powinno więc skutkować ograniczeniem problemów.

Jednak ciągłe dostarczanie zachęca do częstego udostępniania. W tym przypadku w każdej wersji udostępniana jest mniejsza liczba zmian, co zmniejsza prawdopodobieństwo wystąpienia błędów. Wymogiem wstępnym w tej strategii są zautomatyzowane i niezawodne procesy. W przeciwnym razie często udostępnianie prowadzi do przeciążenia informatyków wykonujących ręczne procesy, a także zwiększa ryzyko, ponieważ takie procesy są w wyższym stopniu narażone na błędy. Zamiast dążyć do rzadkiego udostępniania, należy zautomatyzować procesy, by ograniczyć ryzyko. Dodatkową korzyścią jest tu oczywiście to, że przy częstym udostępnianiu poszczególne wersje obejmują mniej zmian, dzięki czemu ryzyko wystąpienia błędów też jest mniejsze.

⁶ Zob. <https://www.sec.gov/litigation/admin/2013/34-70694.pdf>.

⁷ Zob. <http://www.kitchensoap.com/2013/10/29/counterfactuals-knight-capital/>.

Tak więc motywacja do stosowania ciągłego dostarczania (rysunek 1.2) jest wyraźnie odmienna od tej związanej z podejściem Lean Startup. Ważna jest niezawodność i wyższa jakość techniczna procesu udostępniania, a nie czas do wejścia na rynek. Ponadto korzyści odczuwają działy informatyczne, a nie tylko jednostki biznesowe.



Rysunek 1.2. Przyczyny stosowania ciągłego dostarczania w firmie

Ponieważ korzyści są odmienne, można pogodzić się z innymi kompromisami. Na przykład warto zainwestować w potok ciągłego dostarczania nawet wtedy, jeśli nie obejmuje on całej drogi do środowiska produkcyjnego (czyli gdy środowisko produkcyjne i tak trzeba budować ręcznie). W końcu środowisko produkcyjne wystarczy zbudować raz dla każdej wersji, natomiast dla różnych testów potrzebnych jest wiele środowisk. Jeśli jednak główną motywacją do wprowadzenia ciągłego dostarczania jest czas do wejścia na rynek, niezbędne jest, by potok obejmował też środowisko produkcyjne.

Wypróbuj i poeksperymentuj

Zastanów się nad projektem, nad którym pracujesz:

- Gdzie w trakcie instalacji zwykle występują problemy?
- Czy można je rozwiązać za pomocą automatyzacji?
- Gdzie obecne podejścia można uprościć, aby umożliwić automatyzację i optymalizację? Oceń potrzebne wysiłki i oczekiwane korzyści.
- Jak obecnie budowane są systemy produkcyjne i testowe? Czy odpowiada za to ten sam zespół? Czy możliwe jest zastosowanie automatyzacji do obu obszarów, czy tylko do jednego z nich?
- Dla których systemów automatyzacja byłaby przydatna? Jak często budowane są te systemy?

1.4.10. Szybsze informacje zwrotne i podejście Lean

Gdy programista modyfikuje kod, otrzymuje informacje zwrotne dzięki własnym testom, testom integracyjnym, testom wydajności i ostatecznie ze środowiska produkcyjnego. Jeśli zmiany są wprowadzane w środowisku produkcyjnym tylko raz na kwartał, to od zmodyfikowania kodu do uzyskania informacji ze środowiska produkcyjnego może minąć kilka miesięcy. To samo dotyczy testów akceptacyjnych i wydajności. Jeśli błąd zostanie wykryty po takim czasie, programista będzie musiał przypomnieć sobie, co implementował miesiące temu i w czym może tkwić problem.

Ciągłe dostarczanie sprawia, że cykl otrzymywania informacji zwrotnych jest znacznie krótszy. Za każdym razem, gdy kod przepływa przez potok, programista i cały jego zespół otrzymują informacje zwrotne. Po każdej zmianie można uruchamiać zautomatyzowane testy akceptacyjne i przepustowości. To pozwala programiście i zespołowi rozwoju produktu szybciej wykrywać i poprawiać błędy. Czas do uzyskania informacji zwrotnych można dodatkowo skrócić, stosując szybkie testy (na przykład testy jednostkowe) i najpierw przeprowadzając testy ogólne, a dopiero później szczegółowe. To od początku gwarantuje, że wszystkie funkcje działają przynajmniej w łatwych scenariuszach (tak zwanych „ścieżka optymistycznych”). Pozwala to łatwiej i szybciej wykrywać proste usterki. Ponadto testy, o których z doświadczenia wiadomo, że częściej kończą się niepowodzeniem, należy przeprowadzać na początku.

Ciągłe dostarczanie jest zgodne z podejściem Lean. W tym podejściu wszystko, za co klient nie płaci, jest uznawane za marnotrawstwo. Każda zmiana w kodzie jest marnotrawstwem, chyba że zostanie wprowadzona w środowisku produkcyjnym, ponieważ tylko wtedy klient będzie gotowy zapłacić za modyfikacje. Ponadto w ciągłym dostarczaniu stosowane są krótsze cykle w celu szybkiego uzyskania informacji zwrotnych. Jest to następna koncepcja z podejścia Lean.

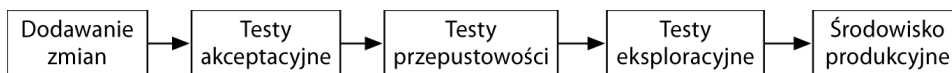
Wypróbuj i poeksperymentuj

Przyjrzyj się projektowi, nad którym obecnie pracujesz.

- Ile czasu mija od zmodyfikowania kodu do:
 - uzyskania informacji zwrotnych z serwera ciągłej integracji?
 - uzyskania informacji zwrotnych z testów akceptacyjnych?
 - uzyskania informacji zwrotnych z testów wydajności lub przepustowości?
 - udostępnienia zmian w środowisku produkcyjnym?

1.5. Procesy generowania i struktura potoku ciągłego dostarczania

Ciągłe dostarczanie to rozwinięcie ciągłej integracji o dodatkowe etapy. Ich przegląd przedstawia rysunek 1.3.



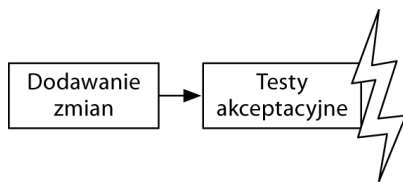
Rysunek 1.3. Etapy potoku ciągłego dostarczania

W tym podrozdziale przedstawiona jest struktura środowiska ciągłego dostarczania. Jest ono zgodne z opisem Humble'a i współpracowników (zob. przypis 1) i obejmuje następujące etapy:

- Faza dodawania zmian (ang. *commit*) obejmuje czynności obsługiwane zwykle przez infrastrukturę ciągłej integracji, takie jak proces budowania, testy jednostkowe i statyczna analiza kodu. Szczegółowe omówienie tej części potoku zawiera rozdział 3.
- Następnym krokiem są testy akceptacyjne, będące tematem rozdziału 4. „Testy akceptacyjne”. Ścisłe rzecz biorąc, omawiane są w nim testy automatyczne — albo interakcje z graficznym interfejsem użytkownika są automatyzowane w celu przetestowania systemu, albo wymogi są opisywane w języku naturalnym w sposób umożliwiający wykorzystanie ich jako testów automatycznych. Od tego etapu (jeśli nie wcześniej) konieczne jest generowanie środowisk, w których aplikacje mogą działać. Dlatego w rozdziale 2. „Zapewnianie infrastruktury” opisane jest generowanie takich środowisk.
- Testy przepustowości (rozdział 5. „Testy wydajności”) gwarantują, że oprogramowanie poradzi sobie z oczekiwanym obciążeniem. W tym celu należy posłużyć się testami automatycznymi, które jednoznacznie określają, czy oprogramowanie jest wystarczająco szybkie. Ważna jest tu nie tylko szybkość, ale też skalowalność. Dlatego test można przeprowadzić także w środowisku, które nie jest identyczne ze środowiskiem produkcyjnym. Jednak musi ono zapewniać wyniki zgodne z działaniem oprogramowania w środowisku produkcyjnym. W zależności od konkretnej sytuacji i wymogów niezwiązanych z pracą oprogramowania (a na przykład związanych z bezpieczeństwem) czasem możliwe jest zautomatyzowanie takich testów.
- W trakcie testów eksploracyjnych (rozdział 6. „Testy eksploracyjne”) aplikacja jest sprawdzana, ale nie według ścisłego planu. Zamiast tego eksperci testują aplikację, koncentrując się na nowych funkcjach i nieoczekiwanym ich działaniu. Dlatego nawet ciągle dostarczanie nie wymaga automatyzacji wszystkich testów. Dzięki dużej liczbie automatycznych testów można znaleźć czas na testy eksploracyjne, ponieważ pracownicy nie muszą już ręcznie przeprowadzać rutynowych testów.
- Wdrażanie kodu w środowisku produkcyjnym (rozdział 7. „Wdrażanie — udostępnianie w środowisku produkcyjnym”) polega jedynie na zainstalowaniu aplikacji w innym środowisku, dlatego jest stosunkowo mało ryzykowne. Istnieją różne podejścia pozwalające dodatkowo zminimalizować zagrożenia związane z umieszczeniem kodu w środowisku produkcyjnym.
- W trakcie eksploatacji aplikacji też pojawiają się wyzwania — zwłaszcza w obszarze monitorowania i obserwacji plików dziennika. Te wyzwania są opisane w rozdziale 8. „Eksploatacja”.

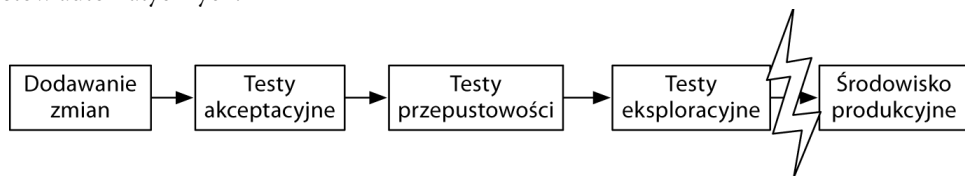
Wersje są przenoszone między poszczególnymi etapami. Możliwe jest, że wersja z powodzeniem przejdzie etap testów akceptacyjnych, ale okaże się za mało wydajna w fazie testów przepustowości. Wtedy nie należy przenosić wersji do następnych etapów, takich jak testy eksploracyjne lub środowisko produkcyjne. W tym modelu trzeba udowodnić, że oprogramowanie spełnia coraz wyższe wymogi. Dopiero po tym można udostępnić je w środowisku produkcyjnym.

Żałujemy, że w logice oprogramowania znajduje się błąd. Najpóźniej wykryty zostanie on w trakcie testów akceptacyjnych, ponieważ sprawdzają one, czy aplikacja jest poprawnie zaimplementowana. W konsekwencji potok zostaje zablokowany (rysunek 1.4). Na tym etapie dodatkowe testy nie są już potrzebne.



Rysunek 1.4. Potok ciągłego dostarczania zostaje zablokowany na etapie testów akceptacyjnych

Programiści później naprawiają błąd i budują oprogramowanie od nowa. Tym razem przechodzi ono także testy akceptacyjne. Jednak w nowej funkcji nadal występuje błąd, który nie zostaje wykryty w automatycznych testach akceptacyjnych. Można go znaleźć tylko w trakcie testów eksploracyjnych. Dlatego tym razem potok zostaje zablokowany na poziomie testów eksploracyjnych i oprogramowanie znów nie jest udostępniane w środowisku produkcyjnym (rysunek 1.5). Dzięki temu testerzy nie marnują czasu na oprogramowanie, jeśli nie spełnia ono wymogów z zakresu obsługi obciążenia lub zawiera błędy, które można wykryć za pomocą testów automatycznych.



Rysunek 1.5. Potok ciągłego dostarczania zablokowany na etapie testów eksploracyjnych

Teoretycznie możliwe jest równoległe przetwarzanie w potoku kilku wersji kodu. Oczywiście zachodzi warunek, aby potok na to pozwalał. Jeśli testy są wykonywane w niezmiennym środowisku, równoległe przetwarzanie wersji jest niewykonalne, ponieważ gdy środowisko jest zajęte jednym testem, nie może jednocześnie sprawdzać drugiej wersji.

Jednak w ciągłym dostarczaniu równoległe przetwarzanie wersji zdarza się bardzo rzadko. Należy utworzyć dokładnie jeden stan wersji, która jest następnie przenoszona w potoku. Co najwyżej modyfikacje oprogramowania można wykonywać z taką szybkością, by nowa wersja była umieszczana w potoku przed opuszczeniem go przez poprzednią wersję. Możliwe są wyjątki związane z pilnymi poprawkami, jednak jednym z celów ciągłego dostarczania jest traktowanie wszystkich wersji tak samo.

1.5.1. Przykład

W książce używana jest przykładowa aplikacja z procesem rejestracji klienta zainspirowanym przykładem opisującym Wielki Sklep Internetowy (zob. podrozdział P.2). Logika aplikacji jest celowo bardzo prosta. Rejestrowane są: imię, nazwisko i adres e-mail klienta. Aplikacja sprawdza poprawność rejestrowanych danych. Adres e-mail musi mieć poprawną składnię, a jeden adres może posłużyć do zarejestrowania tylko jednej osoby. Ponadto zarejestrowane osoby można wyszukiwać na podstawie adresów e-mail i usuwać.

Ponieważ ta aplikacja nie jest złożona, można ją stosunkowo łatwo zrozumieć. Dzięki temu czytelnicy mogą skoncentrować się na różnych aspektach ciągłego dostarczania przedstawianych za pomocą przykładowej aplikacji.

Aplikacja jest zaimplementowana z użyciem Javy i platformy Spring Boot. Dzięki temu można uruchamiać aplikację (w tym interfejs sieciowy) bez instalowania serwera WWW lub serwera aplikacji. Ułatwia to testy, ponieważ nie trzeba instalować infrastruktury. Jednak w razie potrzeby aplikację można też uruchamiać na serwerze aplikacji lub serwerze WWW takim jak Apache Tomcat. Dane są przechowywane w bazie HSQLDB. Jest to przechowywana w pamięci baza działająca w procesie Javy. Także to rozwiązanie zmniejsza techniczną złożoność aplikacji.

Przykładowy kod źródłowy możesz pobrać ze strony <http://github.com/ewolff/user-registration-V2>. Ważna uwaga: przykładowy kod obejmuje usługi, które działają z uprawnieniami administratora i są dostępne z poziomu internetu. W środowisku produkcyjnym jest to nieakceptowalne z powodu problemów z bezpieczeństwem. Jednak ten przykładowy kod jest przeznaczony tylko do eksperymentów. W tym kontekście wygodna struktura przykładów jest bardzo przydatna.

1.6. Wnioski

Udostępnianie oprogramowania w środowisku produkcyjnym to powolny i ryzykowny proces. Zoptymalizowanie go pozwala zwiększyć skuteczność i wydajność rozwoju oprogramowania. Dlatego ciągle dostarczanie może być jednym z najlepszych sposobów na usprawnienie projektów budowania oprogramowania.

Ciągłe dostarczanie ma skutkować powstawaniem stabilnych i powtarzalnych procesów dostarczania oprogramowania (w zakresie integrowania wszystkich zmian podobne efekty zapewnia ciągła integracja). Ciągłe dostarczanie jest świetnym sposobem na skrócenie czasu do wprowadzenia produktu na rynek, a dodatkowo ma do zaoferowania o wiele więcej. Podejście to minimalizuje ryzyko w projektach rozwoju oprogramowania, ponieważ gwarantuje, że aplikację można wdrożyć i uruchomić w środowisku produkcyjnym. Tak więc korzyści można odnieść w każdym projekcie — nawet w mało konkurencyjnej branży, gdzie czas do wejścia na rynek nie ma dużego znaczenia.

Skorowidz

A

Active Record Migrations, 74
adapter, 128
adnotacja @Given, 127
Amazon Cloud, 64, 72, 73
Amazon Elastic Beanstalk, 73
Ansible, 42, 43
Ant, 80, 81
Apache JMeter, 144
Apache Mesos, 66
artefakt, 80, 84, 89
 licencja, 110
 repozytorium, 80, 82, 84, 85, 96,
 106, 107, 108, 109, 110
 Artifactory, 110
 integracja, 108
 Nexus, 110
 testowanie, 91
 usuwanie, 102
 zarządzanie, 106, 110
Artifactory, 110
atrapa, 92
Azure App Service, 73

B

baza danych, 73, 211
 aktualizowanie, 74
 dla komponentu, 213
 jako komponent, 212, *Patrz też:*
 oprogramowanie architektura
 komponent
Microsoft SQL Server, 73
migracja, 74
MongoDB, 73
MySQL, 73
NoSQL, 72, 75, 213
Oracle, 73
PostgreSQL, 73

schemat danych, 74
SQL, 72
stabilność, 211, 212, 213
synchronizacja, 157
wielkość, 75
zgodność z komponentami, 211,
 212, 213
 zmiany w strukturze, 211
BDD, 126, 130
Behavior-Driven Development,
 Patrz: BDD
Blazemeter, 144
błąd, 26
Buildpack, 72
Buildr, 88

C

Chef, 41, 42, 43, 53, 58, 72, 73
 dostawca, 44
 DSL, 44
 katalog, 45
 receptura, 44, 45, 47
 księga, 46, 47, 49, 50
 reguła, 44
 rola, 47, 49, 50
 węzeł, 48
 zasób, 44
Chef Enterprise, 50
Chef Server, 43, 49, 50, 52
Chef Solo, 43, 48
Chef Zero, 43
ChefSpec, 71
ciągłe dostarczanie, 15, 25, 159, 205
 automatyzacja, 79, 80, 83, 85, 86,
 87, 89
 cel, 23, 84, 215
 etap, 33, 34
 informacje zwrotne, 131
 Lean Startup, *Patrz:* Lean Startup

minimalizowanie ryzyka, 29
potok, 31, 34, 79, 160
 struktura, 32
 przewaga konkurencyjna, 27
zalety, 17, 26, 27, 28, 32

Clean Code, 94
Clojure, 143
Cloud Foundry, 72, 73
CloudStack, 161
Code Retreat, 94
Coding Dojo, 94
concordion, 131
continuous delivery, *Patrz:* ciągłe
 dostarczanie
continuous integration, *Patrz:* kod
 ciągła integracja
CoreOS, 66
Cucumber, 129

D

dane
 ilość, 75
 testowe, 75, 124
 generator, 134
DDD, 126
DevOps, 205
Docker, 40, 56, 57, 60, 72, 73, 161,
 213
 kontener, 56, 57, 59, 62
 kapsułka, 66
 komunikacja, 58, 65
 środowisko uruchomieniowe,
 64
 tworzenie, 58
 udostępniający usługę, 67
obraz, 59, 60, 65
 w klastrze, 66
Docker Compose, 67

Docker Machine, 63, 64, 66
 Docker Registry, 65
 Dokku, 73
 Domain-Driven Design,
Patrz: DDD
 dziennik, 41, 151

E

Enterprise Service Bus, *Patrz:*
 magistrala usług korporacyjna
 ESX, 161
 Eucalyptus, 161
 extreme programming, *Patrz:*
 programowanie ekstremalne

F

Flynn, 73
 Flyway, 74
 Framework for Integrated Tests,
Patrz: platforma Fit
 funkcja
 aktywowanie, 216
 odgałęzienie kodu, 215, 216,
 217
 łącznik, 159, 216, 217

G

Gatling, 139, 143
 Gatling Recorder, 139, 143
 Google App Engine, 72
 Gradle, 81, 86, 89, 109
 wersja, 87
 Wrapper, 87
 wtyczka, 104
 zadanie, 86
 graphical user interface, *Patrz:* GUI
 Grinder, 143
 Grunt, 88
 GUI, 114, 120
 warstwa abstrakcji, 120, 124

H

Heroku, 72
 historia, 126
 HtmlUnit, 121
 Hudson, 94

I

idempotencja, 69
 infrastruktura jako kod, 70, 71
 interfejs, 209
 API WebDriver, 121
 REST, 64, 110
 testowanie, 124

użytkownika graficznego,
Patrz: GUI
 wersjonowanie, 209, 210, 212

J

Jail, 57
 Jasmine, 129
 Java Virtual Machine, *Patrz:* JVM
 JBehave, 127
 jednostka
 robocza, 43
 wdrażania, 207, 208
 język
 C#, 121
 Clojure, 72, 88
 dla systemów rozproszonych,
 144
 DSL, 86, 139
 EMACSLisp, 72
 Erlang, 144
 Go, 72, 73
 Groovy, 86
 Haskell, 73
 Java, 72, 73, 80, 82, 88, 121, 123,
 130
 JavaScript, 88, 121, 125, 129
 Jython, 143
 MySQL, 72
 naturalny, 127, 128, 129
 Node.js, 72, 73
 Perl, 73
 PHP, 72, 73
 Python, 72, 73, 121, 123
 rodziny .NET, 73
 Ruby, 42, 44, 72, 73, 88, 121,
 123, 129
 Scala, 72, 88, 139
 uniwersalny, 126
 JGiven, 130
 JMeter, 144
 JUnit, 123
 JVM, 139

K

Kanban, 29
 kapsułka, 66
 kata, 94
 Knife, 50, 52, 53
 kod
 analiza statyczna, 80, 104
 ciągła integracja, 24, 94, 95, 100,
 101
 jakość, 103, 104
 pokrycie testami, *Patrz:* test
 pokrycie kodu
 problemy z bezpieczeństwem, 150
 źródłowy, 80, 81

komunikat ZeroMQ, 43
 kontener Linuxa, *Patrz:* LXC
 Kubernetes, 66
 KVM, 161

L

Lean Startup, 28, 29
 Leiningen, 88
 Linux
 jądro, 56, 57
 kontener, *Patrz:* LXC
 Linux Container, *Patrz:* LXC
 Liquibase, 74
 LoadRunner, 144
 LoadStorm, 144
 LXC, 56, 57, 62

M

magistrala usług korporacyjna, 209
 manifest zwinnego wytwarzania
 oprogramowania, 15
 master server, *Patrz:* serwer
 nadrzędny
 maszyna wirtualna, 53, 55, 56, 160
 Javy, *Patrz:* JVM
 zastępnik, 56
 Maven, 81, 82, 89, 106, 109, 143
 snapshot, *Patrz:* snapshot
 wtyczka, 83, 85, 104
 Mesos, 66
 Mesosphere, 66
 Message Oriented Middleware,
Patrz: warstwa MOM
 metadane, 108
 metoda
 register, 92
 verify, 92
 mikrousluga, 213, 214
 minion, *Patrz:* jednostka robocza
 model
 kaskadowy, 29
 niebieskie-zielone, 156, 157, 161
 POM, *Patrz:* POM

N

narzędzie
 concordion, 131
 Docker, *Patrz:* Docker
 etcd, 66
 Go, 95
 Iva, 109
 OpenJDK, *Patrz:* OpenJDK
 sbt, 88
 Tomcat, *Patrz:* Tomcat
 Vagrant, *Patrz:* Vagrant

Nexus, 110
NUnit, 123

O

OpenJDK, 39
OpenNebula, 160
OpenShift, 73
OpenStack, 160
oprogramowanie, *Patrz też:* kod
aktualizowanie, 40, 41
architektura, 205, 206
implementacja, 207
komponent, 205, 206, 207,
209, 210, 211, 212
optymalizowanie, 207
dodawanie, 62
dostarczanie ciągle,
Patrz: ciągle dostarczanie
instalowane przez klienta, 161
instalowanie, 39, 40, 41
automatyzacja, 38
idempotentne, 41, 44,
Patrz też: idempotencja
ręczne, 38
jakość, *Patrz:* kod jakości
konfigurowanie ręczne, 38
proces budowania, 80
narzędzia, 80, 81, 82, 86, 87,
88, 89
przepustowość, 134
szybkość, 133, 134, 135
udostępnianie, 23
kanarkowe, 157, 158, 159,
161, 162
nowa wersja, 155, 156, 157,
158, 159
w środowisku produkcyjnym,
24, 154
wycofanie zmian, 154, 155,
159
wdrażanie, 153, 159, 161
przełącznik funkcji,
Patrz: funkcja przełącznik
wersja
demonstracyjna, 142
pośrednia, 84, 85
wydajność, 133, 134
wytwarzanie zwinne manifest,
Patrz: manifest zwinnego
wytwarzania
oprogramowania

P

PaaS, 72
Packer, 53, 55, 58

pakiet
Buildpack, 72
Debiana, 110
Heroku, 72
performance, *Patrz:*
oprogramowanie szybkość
Performance Tester, 144
PhantomJS, 125
piramida testów, 114, 115
Platform as a Service, *Patrz:* PaaS
platforma
Cucumber, 129
Fit, 131
Jasmine, 129
JBehave, 127
JUnit, 92, 118
Mockito, 92
RSpec, 129
TestNG, 92
platforma jako usługa, *Patrz:* PaaS
plik
default.rb, 45, 46
Dockerfile, 59, 60, 62
EAR, 80
JSON, 42, 47
knife.rb, 50
konfiguracyjny, 41
settings.xml, 104
Vagrantfile, 54
WAR, 80
XML, 45
YAML, 67
pod, *Patrz:* V
polecenie
docker logs, 61
docker ps, 61
docker pull, 61
docker push, 61
dockermachine.exe env, 64
mvn package, 83
mvn test, 83
vagrant destroy, 55
vagrant halt, 55
vagrant provision, 63
vagrant ssh, 55, 63
vagrant up, 55, 63, 102
POM, 83, 85
prawo Postela, 210
program apt-get, 39
programowanie
BDD, *Patrz:* BDD
ekstremalne, 29
sterowane testami, *Patrz:* TDD
Project Object Model, *Patrz:* POM
protokół
FTP, 144
HTTP, 143, 144, 208
Jabber/XMPP, 144
JMS, 144

LDAP, 144
MySQL, 144
PostgreSQL, 144
REST, 207, 208
SMTP, 144
SOAP, 144, 207
WebDAV, 144
przeglądarka, 121
żądanie, 139
przypadek
pokazowy, 149
testowy, 123
Puppet, 41, 42, 43, 52, 58, 72, 73

R

Rake, 88
Rational, 144
refaktoryzacja, 89, 93
regresja błędów, 26
RSpec, 123, 129

S

Salt, 43
SaltStack, 43
Scrum, 29
Selenium, 121, 122, 123, 125
Selenium Grid, 121
Serverspec, 71
serwer
aktualizacji, 161
Apache HTTP, 73
Apache Tomcat, 73
Bamboo, 95, 105
ciągłej integracji, 94, 95, 100,
101, 104
feniks, 56
instalowanie, 47
JBoss, 73
Jenkins, 94, 95, 100, 105, 108
wtyczka, 96, 97, 98, 100
nadrzędny, 43
Nexus, 108
niemodyfikowalny, 69
TeamCity, 95
Tomcat, 73, *Patrz:* Tomcat
tworzenie, 69
Vert.x, 73
skalowalność, 33
skrypt
Gradle'a, 86
HTML, 124
instalacyjny, 38, 41, 42, 69
Linux, 39
problemy, 39
wady, 39, 40
powłoki, 63

snapshot, 84, 85
 Software Craftsmanship, 94
 SonarQube, 104, 106
 Spirent Blitz, 144
 system
 konfigurowanie, 38
 kontrolni wersji, 25, 48

Ś

środowisko
 dostęp, 25
 generowanie, 25
 IDE, 121
 produkcyjne, 25, 30, 38, 70, 119
 przedprodukcyjne, 25
 testowe, 25, 38, 70, 119, 136
 ustandaryzowane, 72
 zatrzymywanie, 55

T

TDD, 93, 94, 126, 127
 test, 30, 32, 71
 adapter, 128
 akceptacyjny, 25, 33, 80, 113,
 114, 116, 118, 119, 121, 126,
 131
 automatyzacja, 117
 narzędzia, 130, 131
 ręczny, 118
 tekstowy, 126, 128
 bezpieczeństwa, 150
 białoskrzynkowy, 119
 czarnoskrzynkowy, 119
 eksploracyjny, 33, 132, 147
 cel, 147
 środowisko, 149
 zalety, 148
 eksportowanie jako kod, 123
 interesariusz, 118
 interfejsu, 124
 jednostkowy, 80, 90, 91, 92, 103,
 114, 118, 119
 tworzenie, 91
 zalety, 90

koszt, 147
 obciążeniowy, 80, 144
 oparty na sesji, 150, 151
 penetracyjny, 150
 piramida, *Patrz:* piramida
 testów
 pokrycie kodu, 90, 93, 131
 miara, 103
 pomijanie, 24
 protokół, 131
 przechowywanie, 81
 przepustowości, 33
 regresyjny, 148
 ręczny, 115, 117, 147
 akceptacyjny, 118
 eksploracyjny, 118
 wymagań niefunkcjonalnych,
 148
 zalety, 148
 strukturyzowanie, 116
 techniczny, 25
 użyteczności, 148
 wersji beta, 150
 wydajności, 116, 133, 135
 automatyzacja, 133
 implementowanie, 137, 139
 narzędzia, 139, 143, 144
 sprzęt, 135
 w chmurze, 136
 wiarygodność, 137, 138
 wirtualizacja, 136
 zachowania użytkowników,
 135
 wydajność, *Patrz też:*
 oprogramowanie wydajność
 z udziałem klientów, 148
 z użyciem interfejsu
 API, 114, 115, 119
 GUI, 114, 115, 119, 120, 123,
 130
 z użyciem przeglądarki, 125
 zdalnej, 121
 Test Kitchen, 71
 Test::Unit, 123
 test-driven development, *Patrz:* TDD

TestNG, 123
 throughput, *Patrz:* oprogramowanie
 przepustowość
 Tomcat, 39, 55
 Tsung, 144

U

unittest, 123

V

Vagrant, 40, 53, 54, 56, 62, 102, 106
 instalowanie, 55
 rozszerzanie, 53
 veewee, 53
 Virtual Box, 53
 VMware, 160
 VMware Fusion, 53
 VMware vSphere, 64
 VMware Workstation, 53

W

warstwa
 abstrakcji GUI, 120, 124
 MOM, 209
 Windmill, 125
 wirtualizacja, 53, 136, 160
 hipernadzorca, 161
 narzędzia, 160, 161
 wydajność, 56
 współbieżność, 139

X

Xen, 161
 XP, *Patrz:* programowanie
 ekstremalne

Z

zależność, 42, 80, 84
 zasada odporności, 210
 Zone, 57

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>

CIĄGŁE DOSTARCZANIE OPROGRAMOWANIA KLUCZEM DO OPTYMALIZACJI PRACY Z KODEM

Pojęcie ciągłego dostarczania wywodzi się z metodyk zwinnego wytwarzania oprogramowania, a polega na sprawnej analizie i optymalizacji procesu prowadzącego do udostępnienia oprogramowania oraz na zintegrowaniu go z rozwojem produktu. Dzięki usprawnieniu tych prac proces udostępniania oprogramowania automatyzuje się, staje się bardziej powtarzalny i o wiele mniej ryzykowny, a to spotyka się z aprobatą klientów.

Niniejsza książka jest wartościowym przewodnikiem dla zespołów projektowych. Docenią ją zwłaszcza programiści i menedżerowie pracujący według zasad DevOps. Przedstawiono tu podstawowe procesy, wymagania, korzyści i konsekwencje techniczne. Pokazano, w jaki sposób należy implementować potoki i nimi zarządzać. Dzięki lekturze tej książki płynne integrowanie ciągłego dostarczania z architekturą oprogramowania i pracą działów informatycznych stanie się o wiele łatwiejsze. Opisano tu również przykładowe projekty, które stanowią punkt wyjścia do samodzielnych eksperymentów, a nawet do tworzenia własnych potoków ciągłego dostarczania.

EBERHARD WOLFF od przeszło 15 lat zajmuje się architekturą oprogramowania i doradztwem w obszarze styku biznesu i technologii. Wygłaszał prelekcje i przemówienia na międzynarodowych konferencjach, był członkiem komisji programowych wielu sympozjów. Napisał ponad 100 artykułów i książek. Koncentruje się na nowoczesnych architekturach, często obejmujących chmurę, ciągłe dostarczanie, DevOps, mikrousługi i bazy typu NoSQL.

Najważniejsze zagadnienia przedstawione w książce:

- ciągłe dostarczanie: co to jest i jakie problemy rozwiązuje
- automatyzacja tworzenia oprogramowania
- testy: akceptacyjne, wydajności i eksploracyjne
- wdrażanie metodyki ciągłego dostarczania w organizacji
- wpływ ciągłego dostarczania na architekturę aplikacji

Helion 	<i>Sprawdź nasze szkolenia!</i>	KOD KORZYŚCI Sięgnij po więcej! 	
 helion.pl	 SZKOLENIA AKADEMIA IT & BUSINESS	ISBN 978-83-283-3784-8	
 0 801 339900		 9 788328 337848	
 0 601 339900	WWW.SZKOLENIA.HELION.PL	Cena: 49,00 zł	
INFORMATYKA W NAJLEPSZYM WYDANIU			