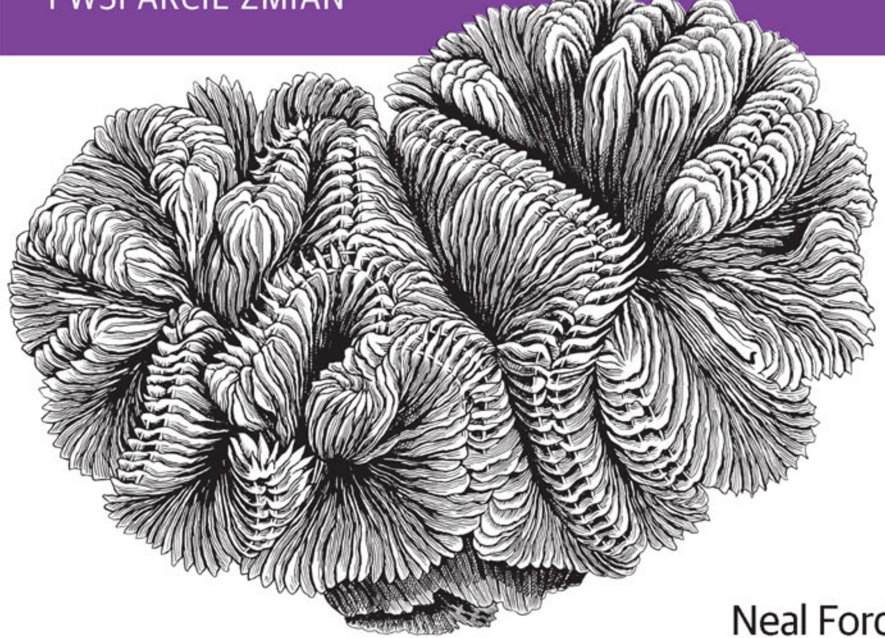
O'REILLY®

Architektura ewolucyjna

PROJEKTOWANIE OPROGRAMOWANIA I WSPARCIE ZMIAN



Neal Ford
Rebecca Parsons
Patrick Kua

Tytuł oryginału: Building Evolutionary Architectures: Support Constant Change

Tłumaczenie: Krzysztof Sawka

ISBN: 978-83-283-4724-3

© 2018 Helion S.A.

Authorized Polish translation of the English edition of Building Evolutionary Architectures
ISBN 9781491986363 © 2017 Neal Ford, Rebecca Parsons, and Patrick Kua

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

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

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w 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 HELION SA 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.

HELION SA
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/archew
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

Spis treści

Wstęp 9

Wprowadzenie 11

1. Architektura oprogramowania 15

Architektura ewolucyjna 17

W jaki sposób możemy planować długoterminowo, skoro wszystko wokół zmienia się przez cały czas? 17

W jaki sposób możemy po stworzeniu architektury zabezpieczyć ją przed stopniową degradacją? 21

Zmiana przyrostowa 22

Zmiana kierowana 23

Wielowymiarowość architektury 24

Prawo Conwaya 27

Dlaczego „ewolucyjna”? 31

Podsumowanie 31

2. Funkcje dopasowania 33

Czym jest funkcja dopasowania? 35

Kategorie 38

Atomowe/holistyczne 38

Wywoływane/ciągłe 39

Statyczne/dynamiczne 39

Kup ksi k Pole ksi k

Zautomatyzowane/ręczne 40
Czasowe 41
Zamierzone/wyłaniające się 41
Wyspecjalizowane 41
Wczesne rozpoznawanie funkcji dopasowania 42
Przegląd funkcji dopasowania 44

3. Projektowanie zmian przyrostowych 47
Elementy budulcowe 51
Testowalność 53
Potoki wdrażania 54
Kombinacje poszczególnych kategorii funkcji dopasowania 59
Analiza przypadku: Restrukturyzowanie architektury za pomocą 60 wdrożeń dziennie 61
Sprzeczne cele 64
Analiza przypadku: Dodawanie funkcji dopasowania do usługi fakturowania w firmie Nie Najgorsze Patenty 65
Projektowanie zorientowane na hipotezy i dane 68
Analiza przypadku: Co przenosić? 70

4. Sprzężenie architektury 73
Modułowość 73
Kwanty architektury i ziarnistość 74
Ewoluowalność stylów architektury 78
Bryła błotna 79
Monolity 80
Architektury sterowane zdarzeniami 89
Architektury zorientowane na usługi 95
Architektury „bezserwerowe” 110
Kontrolowanie rozmiaru kwantu 112
Analiza przypadku: Zabezpieczanie przed cyklicznymi zależnościami pomiędzy składnikami 113

Kup ksi k Pole ksi k

5. Dane ewolucyjne 117
Projektowanie ewolucyjnej bazy danych 117
Ewoluowanie schematów 118
Integracja współdzielonych baz danych 120
Nieprawidłowe sprzęganie danych 125
Zatwierdzanie dwufazowe transakcji 125
Wiek i jakość danych 128
Analiza przypadku: Ewolucja trasowania w firmie Nie Najgorsze Patenty 130

6. Tworzenie ewoluowalnych architektur 133
Mechanika 133
1. Identyfikacja wymiarów podlegających ewolucji 134
2. Definiowanie funkcji dopasowania dla każdego wymiaru 134
3. Stosowanie potoku wdrażania do automatyzacji funkcji dopasowania 134
Nowe projekty 135
Modernizowanie istniejących architektur 136
Prawidłowe sprzęganie i spójność 136
Praktyki inżynieryjne 137
Funkcje dopasowania 137
Skutki stosowania modelu COTS 138
Migrowanie architektur 140
Etapy migracji 141
Ewoluowanie oddziaływań pomiędzy modułami 144
Wskazówki dotyczące tworzenia architektur ewolucyjnych 148
Usuń niepotrzebną zmienność 148
Zagwarantuj odwracalność decyzji 150
Przedkładaj ewoluowalność nad przewidywalność 152
Twórz warstwy przeciwdegradacyjne 153
Analiza przypadku: Szablony usług 156
Tworzenie architektur ofiarniczych 157
Minimalizuj wpływ zmian zewnętrznych 159

Kup ksi k Pole ksi k

Aktualizowanie bibliotek i szkieletów 161
Preferuj dostarczanie ciągłe do migawek 162
Wersjonuj usługi wewnętrznie 164
Analiza przypadku: Ewoluowanie systemu oceniania w firmie Nie Najgorsze Patenty 165

7. Pułapki i antywzorce architektury ewolucyjnej 169
Architektura techniczna 169
Antywzorzec: Monopolista 169
Pułapka: Nieszczelne abstrakcje 171
Antywzorzec: Pułapka ostatnich 10% 174
Antywzorzec: Nadużywanie wielokrotnego wykorzystywania kodu 175
Analiza przypadku: Wielоużywalność w firmie Nie Najgorsze Patenty 178
Pułapka: Projektowanie zorientowane na CV 179
Zmiany przyrostowe 180
Antywzór: Nieprawidłowe zarządzanie 180
Analiza przypadku: Zarządzanie wyważone w firmie Nie Najgorsze Patenty 183
Pułapka: Brak szybkości wydawania 183
Kwestie biznesowe 185
Pułapka: Dostosowywanie produktu 186
Antywzorzec: Raportowanie 187
Pułapka: Horyzonty planowania 189

8. Stosowanie architektury ewolucyjnej w praktyce 191
Czynniki organizacyjne 191
Zespoły przekrojowe 191
Zorganizowane wokół umiejętności biznesowych 193
Produkt ponad projekt 194
Radzenie sobie ze zmianami zewnętrznymi 196
Związki pomiędzy członkami zespołu 198

Kup książkę Poleć książkę

Parameetry sprzęgania zespołów 199

Kup książkę Poleć książkę

ROZDZIAŁ 1.

Architektura oprogramowania

Programiści od dawna starają się ukuć zwięzłą, treściwą definicję architektury oprogramowania, ponieważ jej zakres jest bardzo duży i zmienny. Ralph Johnson doskonale stwierdził, że architektura oprogramowania to „ważne rzeczy (czymkolwiek one są)". Zadaniem architekta jest zrozumienie tych istotnych elementów (czymkolwiek one są) i znalezienie równowagi pomiędzy nimi.

Początkowym etapem pracy architekta jest zrozumienie wymagań biznesowych lub domenowych dla proponowanego rozwiązania. Mimo że wymagania te służą jako motywacja mobilizująca do korzystania z oprogramowania w celu rozwiązania problemu, ostatecznie stanowią tylko jeden z czynników, które architekci powinni brać pod uwagę podczas wyznaczania architektury. Muszą oni uwzględniać również mnóstwo innych czynników, zarówno jawnych (na przykład uzgodnienia wydajności na poziomie usługi), jak i niejawnych z natury (na przykład firma bierze udział w fuzji lub przejęciu innego przedsiębiorstwa). Wynika z tego, że przygotowywanie architektury oprogramowania przejawia się w zdolności architektów do analizowania wymagań biznesowych/domenowych wraz z innymi ważnymi czynnikami służącymi do znalezienia rozwiązania równoważącego w optymalny sposób wszystkie te kwestie. Zakres architektury oprogramowania jest zależny od kombinacji wszystkich tych czynników architektonicznych, co zostało zaprezentowane na rysunku 1.1.

Jak widać na rysunku 1.1, wymagania biznesowe i domenowe występują obok innych zagadnień dotyczących architektury (zdefiniowanych przez architektów). Zalicza się do nich szeroka gama zewnętrznych czynników wpływających na proces decyzyjny określający cel i sposób budowania systemu informatycznego. Przykładowa lista tych czynników została zaprezentowana w tabeli 1.1.

Kup ksi k Pole ksi k

Rysunek 1.1. Pełen zakres architektury obejmuje wymagania i „-ości”

Tabela 1.1. Częściowa lista „-ości”

osiągalność	odpowiedzialność	dokładność	przystosowalność	administrowalność
przystępność	zwinność	audytowalność	autonomiczność	dostępność
kompatybilność	komponowalność	konfigurowalność	poprawność	wiarygodność
nastrajalność	debugowalność	degradowalność	determinowalność	demonstrowalność
niezawodność	wdrażalność	odkrywalność	podzielność	trwałość
efektywność	sprawność	użyteczność	rozszerzalność	przejrzystość awarii
odporność na błędy	wierność	elastyczność	możliwość przeprowadzania inspekcji	instalowalność
integralność	interoperacyjność	poznawalność	utrzymywalność	zarządzalność
mobilność	modyfikowalność	modułowość	operacyjność	ortogonalność
przenośność	precyzja	przewidywalność	zdolności procesu	zdolność wytwórcza
udowadnialność	odzyskiwalność	trafność	rzetelność	powtarzalność
reproduktywność	odporność	responsywność	reużywalność	wytrzymałość
bezpieczeństwo	skalowalność	jednorodność	samodzielność	praktyczność
zabezpieczalność	prostota	stabilność	zgodność ze standardami	przeżywalność
podtrzymywalność	dostosowywalność	testowalność	terminowość	identyfikowalność

Podczas tworzenia oprogramowania architekci muszą wyznaczyć najważniejsze z tych „-ości”. Jednak wiele z tych czynników stoi ze sobą w sprzeczności.

Kup ksi k Pole ksi k

Przykładowo, jednoczesne uzyskanie dużej wydajności i maksymalnej skalowalności może być bardzo trudne, ponieważ każdy z tych czynników wymaga ostrożnego dobrania balansu architektury, operacji i wielu innych czynników. W konsekwencji niezbędna analiza projektu architektury i nieuniknione starcie konkurujących ze sobą czynników wymagają równowagi, jednak zalety i wady każdej podjętej decyzji prowadzą do *kompromisów*, tak bardzo znienawidzonych przez architektów. W ciągu kilku ostatnich lat przyrostowy rozwój głównych praktyk inżynieryjnych w procesie tworzenia oprogramowania położył podwaliny pod nowy sposób myślenia o zmianach architektury w czasie, a także o sposobach chronienia istotnych własności architektury w procesie tej ewolucji. Niniejsza książka ma za zadanie powiązanie tych aspektów z nowymi sposobami myślenia o *architekturze* i *czasie*.

Chcemy dodać nową „-ość" do architektury oprogramowania — **ewoluowalność**.

Architektura ewolucyjna

Pomimo naszych starań wraz z upływem czasu modyfikowanie oprogramowania staje się coraz trudniejsze. Elementy składowe systemów informatycznych z wielu różnych powodów nie poddają się łatwym modyfikacjom i z czasem stają się coraz delikatniejsze oraz trudniejsze w obróbce. Zmiany w projektach informatycznych wynikają głównie z przewartościowania funkcjonalności i (lub) zakresu. Istnieje jednak jeszcze jedna forma zmiany, wykraczająca poza kontrolę architektów i planistów. Architekci lubią móc planować strategicznie przyszłość, ale proces ten jest znacznie utrudniony przez ciągle zmieniający się ekosystem inżynierii oprogramowania. Skoro nie możemy unikać zmian, musimy spożytkować je na swoją korzyść.

W jaki sposób możemy planować długoterminowo, skoro wszystko wokół zmienia się przez cały czas?

W świecie biologicznym środowisko ulega ciągłym zmianom z przyczyn naturalnych oraz wywołanych przez człowieka. Przykładowo, na początku lat 30. ubiegłego wieku rolnicy na kontynencie australijskim mieli problem z chrząszczem z gatunku *Dermolepida albohirtum*, który zmniejszał plony trzciny cukrowej. W odpowiedzi ówczesny urząd Bureau of Sugar Experiment Stations sprowadził w czerwcu 1935 roku drapieżnika, ropuchę agę, pierwotnie występującego

Kup ksi k Pole ksi k

jedynie w Ameryce Południowej i Środkowej[1]. Młode osobniki zostały wypuszczone w rejonie północnego Queensland w lipcu i sierpniu 1935 roku. Dzięki nieobecności naturalnych wrogów i toksycznej skórze gatunek ten rozprzestrzenił się na znacznym obszarze Australii; szacuje się, że obecnie występuje tam około 200 milionów osobników. Morał: wprowadzanie zmian w bardzo dynamicznym (eko)systemie może prowadzić do nieprzewidywalnych skutków.

Ekosystem inżynierii oprogramowania składa się ze wszystkich narzędzi, struktur, bibliotek i najlepszych rozwiązań — zgromadzonych technologii w tej dziedzinie. W ekosystemie tym występuje równowaga — podobnie jak w układach biologicznych — która jest zrozumiała dla programistów i w ramach której potrafią oni tworzyć produkty. Równowaga ta jest jednak dynamiczna — cały czas pojawiają się nowe elementy i zaburzają ją aż do pojawienia się nowego stanu równowagi. Wyobraź sobie osobę jadącą na rowerze jednokołowym i wiozącą paczki: jest to układ **dynamiczny**, ponieważ osoba ta musi balansować ciałem, żeby nie spaść z monocyklu, a także znajdujący się w **równowadze**, ponieważ osoba ta jest w stanie utrzymywać się na pojeździe. W ekosystemie inżynierii oprogramowania każda nowa innowacja/rozwiązanie może zaburzyć stan bieżący, co wymusza ustalenie nowej równowagi. Mówiąc w przenośni, dokładamy monocykliście kolejne pudełka, czym zmuszamy go do wkładania większego wysiłku w utrzymanie równowagi.

Architekci pod wieloma względami przypominają naszego nieszczęsnego monocyklistę, gdyż bez przerwy muszą balansować i dostosowywać się do zmieniających się warunków. Praktyki inżynieryjne stanowiące część techniki ciągłego dostarczania stanowią przykład takiego zaburzania równowagi: wcielenie uprzednio odizolowanych funkcji, takich jak operacje, do cyklu inżynierii oprogramowania pozwoliło ujrzeć znaczenie **zmiany** z zupełnie nowej perspektywy. Architekci pracujący w korporacjach nie mogą już polegać na statycznych, pięcioletnich planach, ponieważ w tym okresie cały wszechświat inżynierii oprogramowania będzie ewoluować, przez co każda długoterminowa decyzja może okazać się w końcu nieistotna.

---

[1] G.M. Clarke, S. Gross, M. Matthews, P.C. Catling, B. Baker, C.L. Hewitt, D. Crowther, S.R. Saddler, *Environmental Pest Species in Australia*, „Australia: State of the Environment, Second Technical Paper Series (Biodiversity)” 2000, Department of the Environment and Heritage, Canberra.

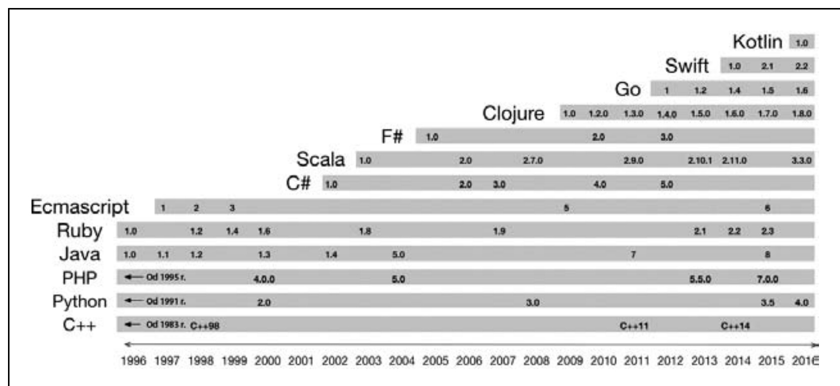
Kup ksi k Pole ksi k

Destrukcyjne zmiany są trudne do przewidzenia nawet dla doświadczonych praktyków. Wzrost zainteresowania narzędziami pozwalającymi na korzystanie z kontenerów, takimi jak Docker (*https://www.docker.com/*), stanowi przykład nieprzewidywalnych przemian w branży. Możemy jednak prześledzić wzrost popularności konteneryzacji jako szereg niewielkich, przyrostowych etapów. Dawno, dawno temu systemy operacyjne, aplikacje serwerowe i inne elementy infrastruktury były tworami komercyjnymi, wymagającymi licencji oraz olbrzymich wydatków. Wiele zaprojektowanych w owym czasie architektur było skoncentrowanych na wydajnym korzystaniu ze współdzielonych zasobów. Dystrybucje linuksowe stopniowo stawały się coraz przyjaźniejsze dla firm, co zredukowało koszty *monetarne* systemów operacyjnych do zera. Następnie pojawiły się rozwiązania z zakresu metody DevOps, na przykład zautomatyzowana aprowizacja maszyn poprzez takie narzędzia jak Puppet (*https://puppet.com/*) czy Chef (*https://www.chef.io/*) sprawiła, że dystrybucje Linuksa stały się *operacyjnie* darmowe. Gdy ekosystem stał się darmowy i został spopularyzowany, kwestią czasu była konsolidacja wokół najczęściej używanych, przenośnych formatów, padło więc na narzędzie Docker. Konteneryzacja nie byłaby jednak możliwa bez wcześniejszych etapów ewolucyjnych prowadzących do tego rozwiązania.

Używane przez nas platformy programistyczne ilustrują taką ciągłą ewolucję. Nowsze wersje języka programowania oferują lepsze interfejsy programowania aplikacji (API), poprawiające elastyczność lub stosowalność wobec nowych typów problemów; z kolei nowsze języki programowania wprowadzają odmienny paradygmat i inny zestaw konstruktów. Przykładowo, język Java został wprowadzony jako zamiennik języka C++ w celu ułatwienia pisania kodu sieciowego i rozwiązania problemów z zarządzaniem pamięcią. Jeśli spojrzymy 20 lat wstecz, zauważymy, że w przypadku wielu języków ich interfejsy API są do teraz rozwijane, a nowsze języki programowania regularnie są dostosowywane do rozwiązywania nowych problemów. Ewolucja języków programowania została zaprezentowana na rysunku 1.2.

Bez względu na aspekt inżynierii oprogramowania — platformę programistyczną, język programowania, środowisko operacyjne, trwałe technologie itd. — oczekujemy ciągłych zmian. Chociaż nie potrafimy przewidzieć momentu pojawienia się zmian w krajobrazie technicznym lub domenowym ani czy będą one trwałe, to wiemy, że są nieuniknione. W rezultacie powinniśmy tworzyć architekturę systemów z myślą o tych zmianach.

Kup ksi k Pole ksi k



Rysunek 1.2. Ewolucja popularnych języków programowania

Jeżeli ekosystem ulega ciągłym zmianom w nieprzewidziany sposób i nie można założyć, kiedy będą się pojawiać, to jaką mamy *alternatywę* dla ustalonych planów? Architekci korporacyjni i pozostali programiści muszą nauczyć się sztuki adaptacji. Jeden z powodów sporządzania planów długoterminowych był natury finansowej; wprowadzanie zmian w oprogramowaniu było kosztowne. Jednak współczesne rozwiązania inżynieryjne obalają to założenie, gdyż obecnie wprowadzanie zmian jest tańsze z powodu automatyzacji części procesów oraz pojawienia się nowych technik, takich jak DevOps.

Wraz z upływem lat wielu bystrych programistów uświadomiło sobie, że niektóre części stworzonych przez nich systemów były trudniejsze do zmodyfikowania od innych. Dlatego **architektura oprogramowania** jest zdefiniowana jako „elementy, które trudno zmieniać w późniejszym terminie". Taka wygodna definicja oddzieliła składniki, które można modyfikować bez większego wysiłku, od wymagających naprawdę trudnych zmian. Niestety definicja ta zaprowadziła w ślepy zaułek: założenie, że zmiany będą trudne, staje się samospełniającą się przepowiednią.

Kilka lat temu pewni innowacyjni architekci oprogramowania spojrzeli na problem „elementów, które trudno zmieniać w późniejszym terminie", w całkiem odmienny sposób: a może wbudować zmienność *w* architekturę? Innymi słowy, jeśli *swoboda zmian* stanie się fundamentalną zasadą architektury, to same zmiany przestaną być trudne. Wprowadzenie ewoluowalności do architektury oznacza pojawienie się zupełnie nowego zestawu zachowań, przez co zostaje ponownie zaburzona dynamiczna równowaga.

Kup ksi k Pole ksi k

Nawet jeśli ekosystem nie ulega zmianom, to co ze stopniową erozją parametrów architektury? Architekci projektują architekturę, ale później wystawiają ją na działanie zagmatwanego środowisko *implementowania* elementów w ramach tej architektury. W jaki sposób architekci chronią istotne, zdefiniowane przez siebie składniki?

W jaki sposób możemy po stworzeniu architektury zabezpieczyć ją przed stopniową degradacją?

Niefortunny rozpad, często zwany **gniciem bitów** (ang. *bit rot*), występuje w wielu organizacjach. Architekci dobierają określone wzorce architektoniczne w celu uwzględniania wymogów biznesowych i różnych „-ości", ale nieraz parametry te przypadkowo zanikają w miarę upływu czasu. Przykładowo, jeśli architekt zaprojektował wielowarstwową architekturę, zawierającą na szczycie warstwę prezentacji, na spodzie warstwę trwałości i kilka warstw pośrednich, programiści pracujący nad systemem raportowania często proszą o bezpośredni dostęp do warstwy trwałości z warstwy prezentacji, z pominięciem warstw pośrednich, gdyż uzyskują w ten sposób lepszą wydajność. Architekci wstawiają warstwy po to, aby odizolować zmiany. Programiści pomijają te warstwy, przez co zwiększają sprzężenie i niweczą cel ich obecności.

W jaki sposób architekci, po zdefiniowaniu istotnych parametrów architektury, mogą *chronić* je przed erozją? Dodanie **ewoluowalności** jako parametru architektury ma na celu ochronę pozostałych jej cech w miarę ewolucji systemu. Jeśli na przykład architekt zaprojektował architekturę pod względem skalowalności, to z pewnością nie chce, aby ten parametr zaniknął wraz z rozwojem układu. Zatem **ewoluowalność** jest metaparametrem, otoczką architektury chroniącą wszystkie pozostałe jej parametry.

W niniejszej książce udowadniamy, że skutkiem ubocznym architektury ewolucyjnej są mechanizmy chroniące ważne parametry tejże architektury. Przyjrzymy się koncepcjom **architektury ciągłej** (ang. *continual architecture*): tworzeniu architektury pozbawionej stanu końcowego i zaprojektowanej w celu ewoluowania wraz ze zmieniającym się ekosystemem inżynierii oprogramowania, a także zawierającej wbudowane mechanizmy ochraniające ważne parametry architektury. Nie próbujemy definiować całokształtu architektury oprogramowania; istnieje już wiele innych definicji (*https://martinfowler.com/ieeeSoftware/whoNeedsArchitect.pdf*). Zamiast tego koncentrujemy się na rozszerzeniu istniejących definicji i dodaniu *czasu* oraz *zmiany* jako głównych elementów architektury.

Kup ksi k Pole ksi k

Poniżej przedstawiamy naszą definicję architektury ewolucyjnej:

> Architektura ewolucyjna wspiera kierowane, przyrostowe zmiany w wielu różnych wymiarach.

Zmiana przyrostowa

Zmiana przyrostowa (ang. *incremental change*) opisuje dwa aspekty architektury oprogramowania: sposób przyrostowego tworzenia oprogramowania i jego wdrażania przez zespoły.

W trakcie tworzenia oprogramowania architektura umożliwiająca niewielkie, przyrostowe zmiany ewoluuje w prostszy sposób, ponieważ programiści mają wpływ na mniejszy zakres zmian. W przypadku wdrażania zmiana przyrostowa dotyczy stopnia modułowości i rozprzężenia funkcji biznesowych, a także ich odwzorowania na architekturę. Czas na przykład.

Załóżmy, że firma Nie Najgorsze Patenty, międzynarodowy sprzedawca widżetów, ma stronę katalogową bazującą na architekturze mikrousług i współczesnych technologiach inżynieryjnych. Jedną z cech tej strony jest możliwość oceniania różnych widżetów za pomocą systemu gwiazdek. Pozostałe usługi firmy Nie Najgorsze Patenty również wymagają oceniania (obsługa klienta, czas dostarczenia produktu itd.), zatem wszystkie korzystają z systemu „gwiazdkowego". Pewnego dnia zespół odpowiedzialny za ten system wypuszcza nową wersję umożliwiającą przyznawanie połówek gwiazdek — niewielkie, ale istotne udoskonalenie. Pozostałe usługi nie wymagają korzystania z nowej wersji systemu oceniania, ale w celu zwiększenia wygody i tak stopniowo na nią przechodzą. W zakres technik DevOps stosowanych w przedsiębiorstwie Nie Najgorsze Patenty wchodzi monitorowanie architektury nie tylko usług, lecz również połączeń pomiędzy nimi. Gdy grupa operacyjna zauważa, że nikt nie łączył się z określoną usługą w danym zakresie czasu, usługa ta zostaje automatycznie usunięta z ekosystemu.

Jest to przykład zmiany przyrostowej na poziomie architektury: pierwotna usługa może działać wraz z jej nową wersją, dopóki wymagają jej pozostałe usługi. Zespoły mogą wprowadzać nowe zachowanie w stosownej chwili (lub w razie potrzeby), a stara wersja automatycznie zostaje przeniesiona do śmietnika.

Aby zmiany przyrostowe były skuteczne, wymagana jest koordynacja różnych rozwiązań w zakresie ciągłego dostarczania. Nie musimy w każdym przypadku

Kup ksi k Pole ksi k

wykorzystywać wszystkich dostępnych rozwiązań, ale często są one spotykane razem. Sposób wprowadzania zmian przyrostowych opisujemy w rozdziale 3.

Zmiana kierowana

Po doborze istotnych parametrów architekci chcą *kierować* zmianami architektury, by ochronić jej atrybuty. W tym celu zapożyczymy z dziedziny obliczeń ewolucyjnych pojęcie **funkcji dopasowania** (ang. *fitness function*). Funkcją dopasowania nazywamy funkcję celu służącą do określania, czy potencjalne rozwiązanie projektowe będzie realizować założone zadania. W obliczeniach ewolucyjnych funkcja ta pozwala ocenić, czy dany algorytm uległ udoskonaleniu po upływie jakiegoś czasu. Inaczej mówiąc, funkcja dopasowania sprawdza, w jakim stopniu każdy wygenerowany wariant algorytmu jest „dopasowany" do definicji „dopasowanego" algorytmu wyznaczonej przez projektanta.

Architektura ewolucyjna stawia przed nami podobne wyzwanie — wraz z rozwojem architektury potrzebujemy mechanizmów określających wpływ zmian na istotne parametry architektury oraz zapobiegających degradacji tych parametrów w czasie. Opisywany tu odpowiednik funkcji dopasowania dotyczy różnorodnych mechanizmów wprowadzanych przez nas po to, aby zagwarantować, że architektura nie ulegnie zmianom w niepożądany sposób — należą do nich m.in. metryki, testy i inne narzędzia weryfikacyjne. Gdy architekt wyznacza parametry architektury, które mają być chronione w trakcie ewolucji systemu, definiuje przynajmniej jedną funkcję dopasowania służącą do zabezpieczenia tych cech.

W ujęciu historycznym część architektury była często postrzegana jako aktywność zarządcza i dopiero od niedawna architekci akceptują wprowadzanie zmian z poziomu architektury. Architektoniczne funkcje dopasowania pozwalają na podejmowanie decyzji w kontekście potrzeb organizacji i funkcji biznesowych przy jednoczesnym wprowadzaniu jawnych i testowalnych podstaw decyzyjnych. Architektura ewolucyjna nie jest nieograniczoną i nieodpowiedzialną techniką inżynierii oprogramowania, lecz metodą równoważącą potrzebę szybkich zmian z rygorem dotyczącym systemów i parametrów architektury. Funkcje dopasowania napędzają proces decyzyjny, nadając kierunek architekturze z jednoczesnym umożliwianiem zmian wymaganych do wspierania zmieniających się środowisk biznesowego i technologicznego.

Wykorzystujemy **funkcje dopasowania** do tworzenia wskazówek opisujących ewolucję architektur; omówimy je szczegółowo w rozdziale 2.

Kup ksi k

Pole ksi k

Wielowymiarowość architektury

> Nie istnieją systemy odizolowane. Świat jest ciągły. Granice wokół systemu są wyznaczane w kontekście celu dyskusji.
>
> — Donella H. Meadows

Począwszy od starożytnych Greków fizyka była stopniowo rozwijana w celu analizowania Wszechświata jako zbioru o skończonej liczbie punktów, czego zwieńczeniem jest **mechanika klasyczna** (*https://pl.wikipedia.org/wiki/Mechanika_klasyczna*). Jednak pojawienie się precyzyjniejszych instrumentów badawczych i obserwacja bardziej złożonych zjawisk na początku XX wieku skierowały naszą uwagę na kwestię względności. Badacze zrozumieli, że to, co uprzednio uznawali za odizolowane zjawiska, w istocie stanowi sieć wzajemnych zależności. Począwszy od lat 90. ubiegłego wieku światli architekci zaczęli stopniowo dostrzegać wielowymiarowość architektury. Technika ciągłego dostarczania rozwinęła tę perspektywę również na kwestie operacyjne. Jednakże architekci oprogramowania często koncentrują się głównie na architekturze **technicznej**, która stanowi zaledwie jedną z płaszczyzn projektu informatycznego. Jeśli architekci chcą tworzyć ewoluującą architekturę, muszą brać pod uwagę wszystkie elementy systemu, na które będą wpływać zmiany. Podobnie jak wiemy dzięki dokonaniom fizyków, że wszystko jest względne, tak architekci rozumieją, że projekt informatyczny składa się z wielu wymiarów.

Aby architekci byli w stanie tworzyć ewoluowalne systemy informatyczne, muszą wykraczać poza architekturę techniczną. Przykładowo, jeśli w projekcie jest wykorzystywana relacyjna baza danych, to jej struktura i relacje pomiędzy poszczególnymi encjami będą także ewoluować. Nie chcemy również budować systemu tak, aby w trakcie ewolucji pojawiały się kolejne dziury w zabezpieczeniach. Są to przykłady *wymiarów* architektury — jej składowych dopasowanych do siebie, często w ortogonalny sposób. Niektóre wymiary wpisują się w tak zwane **zagadnienia architektoniczne** (ang. *architectural concerns*; lista „-ości" wymienionych w tabeli 1.1), w rzeczywistości jednak pojęcie *wymiarów* jest szersze, gdyż obejmuje elementy tradycyjnie wykraczające poza zakres architektury technicznej. Każdy projekt zawiera wymiary, które architekt musi brać pod uwagę podczas analizowania procesu ewolucji. Poniżej prezentujemy niektóre powszechnie występujące wymiary wpływające na ewoluowalność współczesnych architektur oprogramowania:

Kup ksi k Pole ksi k

Techniczny

Elementy implementowane w ramach architektury: struktury, biblioteki i implementowane języki.

Danych

Schematy bazodanowe, układy tabel, planowanie optymalizacji itd. Wymiarem tym zajmuje się najczęściej administrator bazodanowy.

Zabezpieczeń

Definiuje politykę bezpieczeństwa, wytyczne oraz narzędzia pomagające odkrywać braki zabezpieczeń.

Operacyjny/systemowy

Dotyczy odwzorowania architektury na istniejącą fizyczną lub wirtualną infrastrukturę: serwery, klastry, switche, zasoby w chmurze itd.

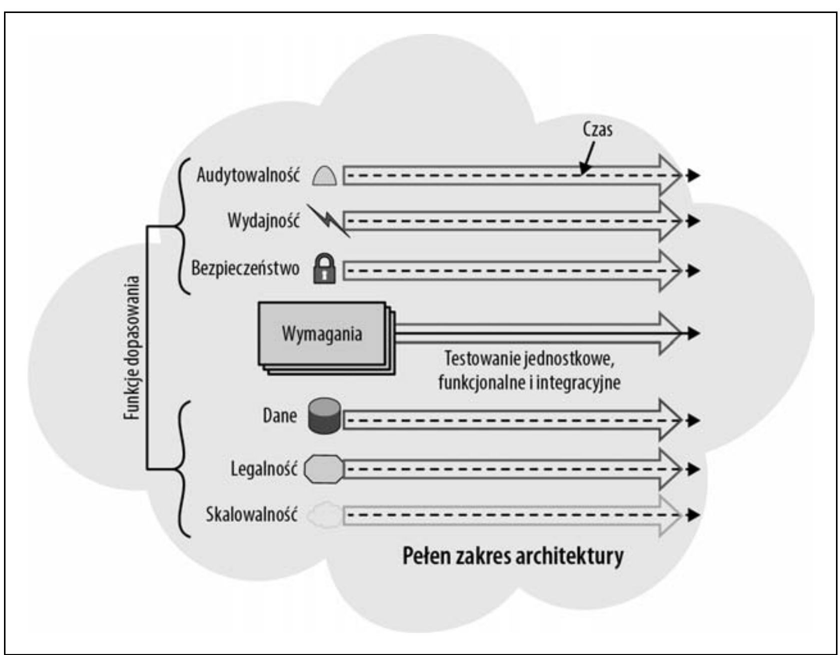
Każdy z tych aspektów kształtuje określony *wymiar* architektury — celowe rozdzielenie składowych danej perspektywy. W naszym ujęciu wymiary architektoniczne zawierają tradycyjne parametry architektury („-ości”), a także każdą inną rolę stanowiącą część inżynierii oprogramowania. Tworzą one perspektywę architektury, którą chcemy zachować w miarę ewolucji naszego problemu i zmian otaczającego nas świata.

Istnieje wiele technik służących do konceptualnego rozkładania architektury na składowe wymiary. Przykładowo, *model perspektyw 4 + 1 Kruchtena* (*http://www.project-media.pl/4plus1.php*), wprowadzony jako część definicji architektury oprogramowania w normach IEEE, koncentrujący się na rozdzielaniu różnych perspektyw od ról, wyznacza cztery perspektywy ekosystemu: **logiczną**, **procesową**, **implementacyjną** i **fizyczną**. W słynnej książce *Software Systems Architecture* (*https://www.viewpoints-and-perspectives.info/*) autorzy wyznaczają katalog perspektyw architektury oprogramowania określający większy zakres ról. Analogicznie model *C4* (*http://www.codingthearchitecture.com/*) Simona Browna rozdziela aspekty architektury w celu ułatwienia organizacji konceptualnej. Dla odmiany my nie próbujemy stworzyć nowej systematyki wymiarów, lecz wyznaczyć obecne w istniejących projektach. Patrząc pragmatycznie, architekt, bez względu na kategorię istotnego zagadnienia, musi nadal chronić dany wymiar. Różne projekty są powiązane z odmiennymi problemami, w wyniku czego mamy do czynienia z różnorodnymi zbiorami wymiarów. Każda z wymienionych technik dostarcza przydatnych spostrzeżeń, zwłaszcza w przypadku nowych projektów, lecz już istniejące projekty muszą sobie radzić z zastaną rzeczywistością.

Kup ksi k Pole ksi k

W kontekście wymiarów architektury istnieje mechanizm, za pomocą którego architekci mogą analizować ewoluowalność różnych architektur poprzez ocenę reakcji ważnego parametru na zmianę. Wraz ze wzrostem zależności systemów od ścierających się zagadnień (skalowalności, bezpieczeństwa, dystrybucji, transakcji itd.) architekci muszą rozszerzać obserwowane wymiary na projekty. Aby stworzyć ewoluowalny system, architekci muszą brać pod uwagę ewolucję tego systemu we wszystkich istotnych wymiarach.

Pełen zakres architektury projektu składa się z wymagań oprogramowania i pozostałych wymiarów. Możemy wykorzystać funkcje dopasowania do ochrony tych parametrów wraz z ewoluowaniem architektury i ekosystemu w czasie, co zostało ukazane na rysunku 1.3.



Rysunek 1.3. Każda architektura składająca się z wymiaru wymagań i pozostałych płaszczyzn jest chroniona przez funkcje dopasowania

Widzimy na rysunku 1.3, że architekci wyznaczyli **audytowalność**, **dane**, **bezpieczeństwo**, **wydajność**, **legalność** i **skalowalność** jako dodatkowe parametry architektury ważne dla danej aplikacji. Jako że wymagania biznesowe ewoluują

Kup ksi k Pole ksi k

w miarę upływu czasu, każdy z tych parametrów zawiera funkcję dopasowania chroniącą jego integralność.

Podkreślamy znaczenie holistycznego ujęcia architektury, mamy jednak świadomość, że znaczna część ewolucji architektury wiąże się z jej wzorcami technicznymi oraz zagadnieniami pokrewnymi, takimi jak sprzężenie czy spójność. W rozdziale 4. analizujemy wpływ sprzężenia architektury technicznej na ewolucyjność, natomiast rozdział 5. poświęcamy tematyce konsekwencji sprzęgania danych.

Sprzęganie dotyczy nie tylko elementów strukturalnych projektu informatycznego. Ostatnio wiele firm informatycznych odkryło wpływ struktury zespołu na tak zaskakujące elementy, jak architektura. Omówimy wszystkie aspekty sprzęgania w oprogramowaniu, jednak wpływ zespołu występuje tak często, że nie możemy go zignorować.

Prawo Conwaya

W kwietniu 1968 roku Melvin Conway opublikował w miesięczniku „Harvard Business Review” artykuł *How do Committees Invent?* (*http://www.melconway.com/research/committees.html*). Autor stwierdził w nim, że struktury społeczne, zwłaszcza kanały komunikacyjne pomiędzy ludźmi, nieuchronnie znajdują odzwierciedlenie w ostatecznym projekcie produktu.

Zgodnie z opisem Conwaya na bardzo wczesnych etapach projektowania tworzona jest ogólna definicja systemu w celu ustalenia sposobu rozbicia obszarów odpowiedzialności na różne wzorce. Sposób, w jaki dana grupa przeanalizuje problem, wpływa na wybory podejmowane na dalszych etapach. Zdefiniował on tak zwane **prawo Conwaya**:

> Organizacje zajmujące się projektowaniem systemów… są ograniczane do tworzenia projektów stanowiących odzwierciedlenie struktur komunikacyjnych tych organizacji.
>
> — Melvin Conway

Conway zwraca uwagę, że gdy technolodzy rozbijają problemy na mniejsze składowe przekazywane innym, to pojawiają się kłopoty z koordynacją. W wielu organizacjach formalne struktury organizacyjne lub sztywna hierarchia są w stanie sprostać tym wyzwaniom koordynacyjnym, często jednak prowadzą do

Kup ksi k Pole ksi k

niezbyt elastycznych rozwiązań. Na przykład w wielowarstwowej architekturze, w której zespół jest rozdzielony na podstawie funkcji technicznej (interfejsu użytkownika, logiki biznesowej itd.), rozwiązanie problemów dotyczących wielu warstw zwiększa koszty koordynacji. Osoby rozpoczynające karierę w młodych firmach, a następnie przechodzące do olbrzymich, międzynarodowych korporacji prawdopodobnie zauważyły kontrast pomiędzy lekką, przyjazną kulturą tych pierwszych a sztywnymi strukturami komunikacyjnymi tych drugich. Dobrym przykładem prawa Conwaya jest próba zmiany kontraktu pomiędzy dwiema usługami, co może okazać się bardzo trudne, jeśli skuteczna modyfikacja usługi obsługiwanej przez jeden zespół wymagałaby koordynacji i wysiłku ze strony drugiego zespołu.

W swoim artykule Conway umiejętnie zwraca uwagę architektów oprogramowania nie tylko na architekturę i projekt oprogramowania, lecz również na delegowanie, wyznaczanie i koordynowanie pracy pomiędzy zespołami.

W wielu organizacjach zespoły są podzielone zgodnie z pełnionymi funkcjami. Do często spotykanych przykładów należą:

Programiści części interfejsowej
: Zespół wyspecjalizowany w określonej technologii interfejsu użytkownika (UI), na przykład technologii HTML, mobilnej czy komputerów stacjonarnych.

Programiści serwerowi
: Zespół mający wyjątkowe umiejętności tworzenia usług serwerowych, czasami warstw interfejsu API.

Programiści bazodanowi
: Zespół potrafiący tworzyć usługi magazynowe i logiczne.

W organizacjach, w których występują silosy funkcjonalne, pion kierowniczy tak rozdziela zespoły, aby usatysfakcjonować dział kadr bez uwzględniania wydajności inżynieryjnej. Każdy zespół może być skuteczny w ramach swojej części projektu (na przykład w tworzeniu grafiki, dodawaniu interfejsu albo usługi serwerowej czy też projektowaniu nowego mechanizmu magazynowania), ale w celu wypuszczenia nowej funkcji biznesowej wszystkie trzy zespoły muszą być zaangażowane w proces jej budowania. Zespoły zazwyczaj są dostosowywane pod względem skuteczności do bieżących zadań, a nie bardziej abstrakcyjnych, strategicznych celów firmy, zwłaszcza jeżeli wisi nad nimi presja harmonogramu.

Kup ksi k Pole ksi k

Zamiast dostarczać całkowicie ukończoną funkcję, zespoły często koncentrują się na dostarczaniu jej elementów składowych, które mogą, ale nie muszą współdziałać z pozostałymi składnikami.

Przy takim rozdziale organizacyjnym tworzenie funkcji uzależnionej od wszystkich trzech zespołów trwa dłużej, ponieważ każdy zespół poświęca na nią odmienną ilość czasu. Rozważmy, jako przykład, scenariusz powszechnej zmiany biznesowej w postaci aktualizacji strony katalogowej. W ramach tej zmiany następuje modyfikacja interfejsu użytkownika, reguł biznesowych i schematów bazodanowych. Jeżeli każdy zespół pracuje we własnym silosie, muszą one koordynować ze sobą harmonogramy i przeznaczać więcej czasu na implementację funkcji. Jest to znakomity przykład ukazujący wpływ struktury zespołu na architekturę i zdolność rozwoju.

Conway zauważył, że *za każdym razem, gdy następuje delegowanie pracy i zawężenie czyjegoś zakresu obowiązków, zostaje jednocześnie zawężona klasa alternatyw projektowych, które można skutecznie wykorzystywać*. Możemy to ująć inaczej: trudno zmienić jakiś element, jeżeli przynależy on do kogoś innego. Architekci oprogramowania powinni zwracać uwagę na sposób podziału pracy i jej delegowania w celu powiązania celów architektury ze strukturą zespołu.

W wielu firmach tworzących architektury takie jak mikrousługi zespoły są budowane w granicach tych usług, a nie jako oddzielne silosy zajmujące się architekturą techniczną. W periodyku „ThoughtWorks Technology Radar” (*https://www.thoughtworks.com/radar*) nazywamy to zjawisko **odwróconym manewrem Conwaya** (ang. *Inverse Conway Maneuver*; *https://www.thoughtworks.com/de/radar/techniques/inverse-conway-maneuver*). Tego typu sposób organizowania zespołów jest idealny, ponieważ struktura zespołu będzie miała wpływ na olbrzymią liczbę wymiarów inżynierii oprogramowania, a także powinna odzwierciedlać rozmiar i zakres problemu. Przykładowo, w trakcie tworzenia architektury mikrousług firmy zazwyczaj w taki sposób dostosowują strukturę zespołów, aby przypominała architekturę — rozbijają silosy funkcjonalne i dołączają członków zajmujących się każdym aspektem biznesowym i technicznym architektury.

Buduj zespoły tak, aby przypominały docelową architekturę, a łatwiej będzie ją uzyskać.

Kup książkę Poleć książkę

Firma Nie Najgorsze Patenty i jej odwrócony manewr Conwaya

W niniejszej książce posługujemy się przykładową firmą Nie Najgorsze Patenty, Niemal Największy Dostawca Widżetów — dużym, internetowym sprzedawcą widżetów (różnych drobnych, bliżej nieokreślonych przedmiotów). Przedsiębiorstwo to stopniowo aktualizuje znaczną część swojej infrastruktury informatycznej. Wykorzystuje kilka starszych systemów, które chce zatrzymać jeszcze na pewien czas, a także stosuje pewne nowe systemy strategiczne wymagające bardziej iteracyjnego podejścia. W kolejnych rozdziałach przyjrzymy się wielu problemom trapiącym firmę Nie Najgorsze Patenty i wprowadzanym przez nią rozwiązaniom służącym spełnianiu potrzeb.

Pierwsza obserwacja, jakiej dokonali architekci tego przedsiębiorstwa, dotyczyła zespołów inżynierii oprogramowania. Stare, monolityczne aplikacje wykorzystywały architekturę wielowarstwową, w której rozdzieleniu uległy warstwy prezentacji, logiki biznesowej, trwałości i operacji. Znajduje to odzwierciedlenie w strukturze zespołu: osoby zajmujące się interfejsem użytkownika przebywają w jednym pomieszczeniu, programiści i administratorzy bazodanowi mają własny silos, a operacje zostały przekazane zewnętrznemu zleceniobiorcy.

Gdy programiści zaczęli opracowywać nowe elementy architektury — architekturę mikrousług zawierającą szczegółowe usługi — koszty koordynacji poszybowały w górę. Usługi były tworzone wokół domen (na przykład *procesu zakupowego*), a nie architektury technicznej, przez co wprowadzenie zmiany w pojedynczej domenie wymagało nieprawdopodobnej wręcz koordynacji pomiędzy poszczególnymi silosami.

Zamiast tego firma Nie Najgorsze Patenty zastosowała odwrócony manewr Conwaya i stworzyła zespoły przekrojowe dopasowane do zakresu usługi: każdy zespół zajmujący się usługą składa się z jej właściciela, kilku programistów, analityka biznesowego, administratora bazodanowego (DBA), kontrolera jakości (QA) oraz specjalisty ds. operacji.

Wpływ zespołu ukazywany jest w wielu miejscach książki, prezentujemy także przykładowe konsekwencje danej struktury zespołu.

Kup ksi k Pole ksi k

Dlaczego „ewolucyjna”?

Jednym z częściej spotykanych pytań dotyczących architektury ewolucyjnej jest pochodzenie jej nazwy: dlaczego jest ona **ewolucyjna**, a nie jakaś inna? Można wymienić znacznie więcej określeń, jak choćby **przyrostowa**, **ciągła**, **zwinna**, **reaktywna** czy **wschodząca**. Ale żadna z tych nazw nie oddaje w pełni istoty omawianej architektury. Podana przez nas definicja architektury ewolucyjnej zawiera dwa kluczowe parametry: przyrostowa i kierowana.

Pojęcia **ciągła**, **zwinna** i **wschodząca** ukazują koncepcję zmiany w czasie, co jest oczywiście kluczowym aspektem architektury ewolucyjnej, ale żadne z nich nie definiuje *sposobu, w jaki* te zmiany następują, ani jaki może być jej pożądany stan końcowy. Każdy z tych terminów sugeruje obecność zmiennego środowiska, ale żaden nie wskazuje, jak powinna wyglądać taka architektura. W naszej definicji za tę część odpowiada wyraz **kierowana** — odzwierciedla on architekturę, jaką chcemy uzyskać — nasz cel końcowy.

Wybraliśmy wyraz **ewolucyjna**, a nie **przystosowawcza**, ponieważ interesują nas architektury przechodzące przez fundamentalne zmiany ewolucyjne, a nie takie, które są modyfikowane i dostosowywane do stopniowo coraz mniej zrozumiałej, przypadkowej złożoności. **Przystosowywanie** sugeruje poszukiwanie sposobu zmuszania czegoś do pracy bez względu na jakość lub rozmiar rozwiązania. Aby tworzyć prawdziwie ewoluujące architektury, architekci muszą wprowadzać rzeczywiste zmiany, a nie prowizoryczne rozwiązania. Wróćmy na chwilę do naszej metafory biologicznej: **ewolucyjność** dotyczy procesu tworzenia układu dostosowanego do pełnienia określonej funkcji oraz zdolnego do przetrwania w zmiennym środowisku, w którym przyszło mu istnieć. Systemy mogą być pojedynczymi przystosowaniami, ale jako architekci powinniśmy troszczyć się o całokształt ewoluującego układu.

Podsumowanie

Architektura ewolucyjna składa się z trzech głównych aspektów: zmiany przyrostowej, funkcji dopasowania i właściwego sprzęgania. W dalszej części książki zajmiemy się osobnym omówieniem każdego z tych czynników, a następnie połączymy je i przekonamy się, co jest potrzebne do stworzenia i utrzymania architektury wspierającej ciągłe zmiany.

Kup ksi k Pole ksi k

Skorowidz

A

adaptacja, 216
administratorzy bazodanowi, 126
aktualizacje
  bibliotek ciągnione, 162
  spekulatywne, 163
  szkieletów popychane, 161, 162
antywzorce architektury ewolucyjnej, 169
antywzorzec
  Monopolista, 169
  Nieprawidłowe zarządzanie, 180
  Pułapka ostatnich 10%, 174
  Raportowanie, 187
  wielokrotne wykorzystywanie kodu, 175
  zakłócania abstrakcji, 153
architektura
  bazująca na usługach, 107
  bez zasobów współdzielonych, 48
  bezserwerowa, 110
  ciągła, 21
  domenowa, 217
  ewolucyjna, 17, 31
    antywzorce, 169
    czynniki organizacyjne, 191
    pułapki, 169
    tworzenie, 221
  ewoluowalna, 133
  korporacyjna, 209
  mikrojądra, 87
  mikrousług, 102
  ofiarnicza, 157, 218
  oprogramowania, 15, 20
  sterowana magistralą ESB, 95
  sterowana zdarzeniami, 89
  techniczna, 24, 169
  warstwowa, 82
  wielowymiarowość, 24
  zorientowana na usługi, 95
atomowe funkcje, 38
audytowalność, 26
automatyzacja, 40
  funkcji dopasowania, 134
  metod DevOps, 194

B

bazy danych
  ewolucyjne, 117
  jakość danych, 128
  punkt integracji, 123
  refaktoryzacja, 122
  sprzęganie danych, 125
  wiek danych, 128
  współdzielone, 120

bezpieczeństwo, 26
biblioteka, 73, 77, 161
brokery, 90
bryła błotna, 79
budżet, 203

C

ciągłe dostarczanie oprogramowania, 54, 162, 214
ciągnienie manualne, 214
cykl zależności pakietów, 113
czas
 cyklu, 184, 214
 realizacji projektu, 184

D

dane, 26
 ewolucyjne, 117
DDD, Domain Driven Design, 101
dług techniczny, 154
dostosowywanie produktu, 186
duplikowanie ponad sprzęganie, 177
dylemat innowatora, 213
dynamiczne funkcje, 40
dyrektor finansowy, 203

E

efekt drugiego systemu, 159
elementy budulcowe, 51
etapy migracji, 141
ewolucja, 216
 trasowania, 130
ewolucyjna baza danych, 117
ewoluowalność, 17, 21, 152, 212, 217
 architektury, 133
 stylów architektury, 78
ewoluowanie
 oddziaływań pomiędzy modułami, 144
 schematów, 118
 systemu, 165

F

framework, 161
funkcje
 atomowe, 38
 dopasowania, 23, 33, 35, 59, 137
  automatyzacja, 134
  dla każdego wymiaru, 134
  istotne, 43
  kategorie, 38
  kluczowe, 43
  korporacyjne, 205
  nieistotne, 43
  ogólnosystemowe, 34
  tymczasowe, 205
  wczesne rozpoznawanie, 42
  wykorzystujące sztuczną inteligencję, 211
 dynamiczne, 40
 holistyczne, 38
 statyczne, 39
 wyspecjalizowane, 41
 wywoływane, 39
 zautomatyzowane, 40

G

gnicie bitów, 21
grupowanie logiczne, 73

H

holistyczne funkcje, 38

I

infrastruktura, 208, 210
 niezmienialna, 149
 śnieżynkowa, 149
integracja, 104
 baz danych, 120
 ciągła, 55

Kup książkę Poleć książkę

inżynieryjna sieć asekuracyjna, 197
izolacja, 83
izolowanie parametrów architektury, 215

J

jakość, 59
język wszechobecny, 78
języki programowania, 20
judo doradcze, 219

K

kanoniczność, 177
kategorie funkcji dopasowania, 38, 59
kolejki wiadomości, 90
koncepcja DDD, 101
konsolidacja, 58
konstrukcja usługi, 165
kontener Docker, 55
korporacyjne funkcje dopasowania, 205
koszt, 204
kultura eksperymentowania, 199, 201
kwant architektury, 74, 112

L

legalność, 26

M

magistrala ESB, 95
mechanika klasyczna, 24
mediatory, 93
metody
    BDUF, 152
    DevOps, 194
    MDD, 39
metryka biznesowa, 214
migawki, snapshots, 162
migrowanie architektur, 140
mikrojądra, 86
mikrousługi, 99
model
    bryły błotnej, 217
    ciągniony, 161
    COTS, 138
modernizowanie architektur, 136
modułowość, 73
monolity, 80
    modułowe, 84
    nieustrukturyzowane, 80
monolityczny listing, 78

N

narzędzie
    JDepend, 114
    Starling, 154

O

oddziaływania fundamentalne, 76
odkładanie decyzji w czasie, 153
odkrywanie usług, 147
odwracalność decyzji, 150
odwrócony manewr Conwaya, 29, 134
ogólnosystemowa funkcja dopasowania, 34
ograniczony kontekst, 75
operacja rozdysponowania, 57

P

parametry
    architektury, 217
    sprzęgania zespołów, 199
perspektywy ekosystemu, 25
planowanie długoterminowe, 17
podział odpowiedzialności, 83
port wsteczny, 41
potoki wdrażania, 54–56, 134
praktyki inżynieryjne, 137
prawidłowe sprzęganie, 80–83, 86, 89, 94
prawo Conwaya, 27

Kup ksi k
Pole ksi k

problem nieznanych niewiadomych, 41
proces tworzenia zorientowanego na hipotezy, 69
procesory zdarzeń, 90
produkt, 194
projekt San Francisco, 176
projektowanie
  ewolucyjnej bazy danych, 117
  zmian przyrostowych, 47
  zorientowane
    na CV, 179
    na dane, 68
    na dziedzinę, 75
    na hipotezy, 68
przełączniki funkcji, 152
przewidywalność, 152, 212
przydatność, 217
pułapka
  Brak szybkości wydawania, 183
  Dostosowywanie produktu, 186
  Horyzonty planowania, 189
  Nieszczelne abstrakcje, 171
  ostatnich 10%, 174
  Projektowanie zorientowane na CV, 179
punkt integracji, 123

R

raportowanie, 187
refaktoryzacja, 138
  baz danych, 122
restrukturyzacja, 138
  architektury, 61
rozdysponowanie, 58
rozprzęganie wymuszone, 182
ryzyko, 221

S

skala wybiórcza, 215
skalowalność, 26
składniki, 73
spójność, 136
  funkcjonalna, 74
sprzeczne cele, 64
sprzęganie, 106, 136, 182
  architektury, 73
  danych, 125
  zespołów, 199
statyczne funkcje, 39
stosowanie architektury ewolucyjnej, 191
system śnieżynkowy, 151
szablony usług, 104, 106, 156
szkielety, frameworks, 161

Ś

środowisko produkcyjne, 59

T

teoria ciągłego dostarczania, 54
test celowo zawodzący po aktualizacji, 41
testowalność, 53
testowanie, 208
  generatywne, 212
testy BDD, 39
transakcje
  zatwierdzanie dwufazowe, 125
trasowanie, 130
tworzenie
  architektur ewolucyjnych, 133, 221
  architektur ofiarniczych, 157
  korporacyjnych funkcji dopasowania, 205
tymczasowe funkcje dopasowania, 205

U

układ dynamiczny, 18
umiejętności biznesowe, 193
usługi, 74, 95, 107
  aplikacji, 97
  fakturowania, 65

Kup ksi k Pole ksi k

infrastrukturalne, 97
korporacyjne, 96
usuwanie zmienności, 148

W

warstwy przeciwdegradacyjne, 153
wczesne rozpoznawanie funkcji, 42
wdrożenie, 49
wersjonowanie usług, 164
wielokrotne wykorzystywanie kodu, 175
wielоużywalność, 178
wielowymiarowość architektury, 24
wydajność, 26
wyszukiwanie zasobów, 194
wzorzec rozciągania/kurczenia, 121

Z

zagadnienia architektoniczne, 24
zależności
między składnikami, 113
statyczne, 163
zarządzanie wyważone, 182, 183
zasada
odkładania decyzji w czasie, 153
trzech, 201
zatwierdzanie dwufazowe transakcji, 125
zdarzenia, 89
międzyprocesowe, 90
inicjujące, 90
zespoły przekrojowe, 191
zespół, 199
ziarnistość, 74
usług, 107
złożoność cyklomatyczna, 36
zmiana kierowana, 23
z funkcjami dopasowania, 80–85, 89, 94
przyrostowa, 22, 47, 80–85, 89, 94, 180, 220
zmienność, 148
związki pomiędzy członkami zespołu, 198

Kup ksi k Pole ksi k

PROGRAM PARTNERSKI

GRUPY HELION

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl

O'REILLY®

Architektura korporacyjna: wysoka sztuka projektowania!

Dzięki tej książce architekci oprogramowania zapoznają się z nowym podejściem do powiązań architektury z czasem. Przekonają się też, że tworzenie architektury ewolucyjnej sprowadza się do trzech głównych zagadnień: funkcji dopasowania, zmian przyrostowych i prawidłowego sprzęgania. W tej książce uwzględniono analizę każdego z tych aspektów, ukazano też mechanizm budowania architektury wspierającej ciągłe zmiany. Szczegółowo wyjaśniono zasady wprowadzania pętli informacji zwrotnej pozwalających całemu zespołowi na spójne rozwijanie systemu w zgodzie z zasadą ciągłego dostarczania. Pokazano metody monitorowania stanu architektury. Sporo uwagi poświęcono problemom danych długowiecznych — jest to często pomijane zagadnienie.

W tej książce znajdziesz między innymi:

- funkcje dopasowania w architekturze
- zmiany przyrostowe wprowadzane za pomocą projektowania i operacji
- sprzężenie architektury i wprowadzanie zmian bez utraty stabilności systemu
- dane ewolucyjne oraz zmiany wymogów i architektury w miarę upływu czasu
- budowanie architektur ewolucyjnych
- praktyczne wdrażanie architektury ewolucyjnej w korporacji

Neal Ford — jest uznanym autorytetem w dziedzinie zwinnych technik inżynieryjnych i architektury oprogramowania. Napisał wiele artykułów i książek, setki razy zabierał głos na różnych konferencjach na całym świecie.

Dr Rebecca Parsons — od dziesięcioleci zajmuje się inżynierią oprogramowania, w tym wielkoskalowymi rozproszonymi aplikacjami obiektowymi, integracją systemów, optymalizacją oprogramowania, teorią obliczeń, uczenia maszynowego i biologii obliczeniowej.

Patrick Kua — słynie z umiejętności równoważenia technologii, ludzi i procesu w celu zwiększenia efektywności zespołu. Na wielu konferencjach wygłasza referaty na temat architektury i tworzenia silnej kultury inżynieryjnej.