

O'REILLY®



Podstawy architektury oprogramowania dla inżynierów

Helion 

Mark Richards, Neal Ford

Tytuł oryginału: Fundamentals of Software Architecture: An Engineering Approach

Tłumaczenie: Leszek Sagalara

ISBN: 978-83-283-7027-2

© 2021 Helion SA

Authorized Polish translation of the English edition of Fundamentals of Software Architecture ISBN 9781492043454 © 2020 Mark Richards, Neal Ford

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

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

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

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

Autorzy 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. Autorzy oraz Helion SA 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/poarop>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

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

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

Spis treści

Przedmowa: obalanie aksjomatów	13
1. Wprowadzenie	17
Zdefiniowanie architektury oprogramowania	19
Oczekiwania wobec architekta	22
Podejmowanie decyzji architektonicznych	23
Ciągłe analizowanie architektury	23
Śledzenie najnowszych trendów	24
Zapewnienie zgodności z decyzjami	24
Bogate i zróżnicowane doświadczenie	25
Wiedza z zakresu biznesu	25
Umiejętności interpersonalne	26
Znajomość i umiejętność stosowania polityki firmy	26
Punkty przecięcia architektury z innymi elementami	27
Praktyki inżynierskie	28
Operacje (DevOps)	31
Proces	32
Dane	32
Prawa architektury oprogramowania	33

Część I. Podstawy

2. Myślenie architektoniczne	37
Architektura a projekt	38
Rozpiętość techniczna	39
Analiza kompromisów	43
Czynniki biznesowe	46
Zachowanie równowagi między architekturą a kodowaniem	47

3. Modułowość	49
Definicja	50
Pomiar modułowości	52
Spójność	52
Sprzężenie	55
Abstrakcyjność i niestabilność	56
Odległość od ciągu głównego	57
Splątanie	59
Unifikacja wskaźników sprzężenia i splątania	63
Od modułów do składników	64
4. Definiowanie parametrów architektury.....	65
(Niepełna) lista parametrów architektury	68
Operacyjne parametry architektury	68
Strukturalne parametry architektury	69
Przekrojowe parametry architektury	69
Kompromisy i najmniej niekorzystna architektura	73
5. Identyfikacja parametrów architektury	75
Określanie parametrów architektury na podstawie zagadnień dziedzinowych	75
Określanie parametrów architektury na podstawie wymagań	77
Studium przypadku: Krzemowe Kanapki	79
Parametry sprecyzowane	79
Parametry dorozumiane	83
6. Pomiar parametrów architektury i zarządzanie nimi	85
Pomiar parametrów architektury	85
Pomiary operacyjne	86
Pomiary strukturalne	87
Pomiary procesowe	89
Funkcje zarządzania i dopasowania	89
Zarządzanie parametrami architektury	89
Funkcje dopasowania	90
7. Zakres parametrów architektury	97
Sprzężenie i splątanie	97
Kwanty architektury i ziarnistość	98
Studium przypadku: „Po raz pierwszy, po raz drugi, sprzedane!”	100

8. Myślenie w oparciu o składniki	105
Zakres składnika	105
Rola architekta	106
Podział architektury	107
Studium przypadku: Krzemowe Kanapki — podział	110
Rola programisty	112
Proces identyfikacji składników	113
Identyfikacja składników początkowych	113
Przypisywanie wymagań do składników	113
Analiza ról i odpowiedzialności	114
Analiza parametrów architektury	114
Restrukturyzacja składników	114
Szczegółowość składników	114
Projektowanie składników	114
Odkrywanie składników	115
Studium przypadku: „Po raz pierwszy, po raz drugi, sprzedane!” — odkrywanie składników	117
Jeszcze raz o kwantach architektury: wybór między architekturą monolityczną a rozproszoną	120

Część II. Style architektoniczne

9. Podstawy	123
Podstawowe wzorce	123
Bryła błotna	123
Architektura unitarna	125
Klient-serwer	125
Architektury monolityczne a rozproszone	127
Mit 1. Sieć jest niezawodna	127
Mit 2. Opóźnienie jest zerowe	128
Mit 3. Przepustowość jest nieskończona	129
Mit 4. Sieć jest bezpieczna	130
Mit 5. Topologia nigdy się nie zmienia	131
Mit 6. Jest tylko jeden administrator	131
Mit 7. Koszt transportu jest zerowy	132
Mit 8. Sieć jest homogeniczna	133
Inne kwestie związane z rozproszeniem	133

10. Styl architektury warstwowej.....	135
Topologia	135
Warstwy izolacji	137
Dodawanie warstw	138
Inne kwestie	139
Dlaczego warto stosować ten styl architektoniczny?	140
Ocena parametrów architektury	141
11. Styl architektury potokowej	143
Topologia	143
Potoki	144
Filtry	144
Przykład	145
Ocena parametrów architektury	146
12. Styl architektury mikrojądra.....	149
Topologia	149
Podstawowy system	150
Dołączane składniki	151
Rejestr	155
Kontrakty	156
Przykłady i przypadki użycia	156
Ocena parametrów architektury	157
13. Styl architektury bazującej na usługach	161
Topologia	161
Warianty topologii	162
Projektowanie usług i ich szczegółowość	164
Podział bazy danych	166
Przykład architektury	168
Ocena parametrów architektury	169
Kiedy należy używać tego stylu architektonicznego?	172
14. Styl architektury sterowanej zdarzeniami	173
Topologia	174
Topologia brokera	174
Topologia mediatora	179
Komunikacja asynchroniczna	186
Obsługa błędów	188
Zapobieganie utracie danych	191
Rozgłaszanie	193
Żądanie-odpowiedź	193

Wybór między modelem opartym na żądaniach a modelem opartym na zdarzeniach	196
Architektury hybrydowe sterowane zdarzeniami	196
Ocena parametrów architektury	197
15. Styl architektury przestrzennej	201
Ogólna topologia	202
Jednostka przetwarzająca	203
Zwirtualizowane oprogramowanie pośredniczące	203
Pompy danych	208
Jednostki zapisu danych	210
Jednostki odczytu danych	211
Kolizje danych	212
Wdrożenia chmurowe a lokalne	215
Buforowanie replikowane a rozproszone	215
Model near-cache	218
Przykłady wdrożeń	219
System sprzedaży biletów na koncerty	220
System aukcji internetowych	220
Ocena parametrów architektury	221
16. Architektura zorientowana na usługi sterowana orkiestracją	223
Historia i filozofia	223
Topologia	224
Taksonomia	224
Usługi biznesowe	224
Usługi korporacyjne	225
Usługi aplikacji	225
Usługi infrastrukturalne	225
Silnik orkiestracji	225
Przepływ komunikatów	226
Wykorzystuj ponownie... i sprzęgaj	227
Ocena parametrów architektury	229
17. Architektura mikrousług	231
Historia	231
Topologia	232
Rozproszenie	233
Ograniczony kontekst	233
Poziom szczegółowości	234
Izolacja danych	234
Warstwa API	235

Wieloużywalność operacyjna	235
Interfejsy	238
Komunikacja	239
Choreografia i orkiestracja	241
Transakcje i sagi	243
Ocena parametrów architektury	247
Dodatkowe informacje	248
18. Wybór odpowiedniego stylu architektonicznego.....	249
Zmiana „mody” w architekturze	249
Kryteria decyzyjne	250
Studium przypadku architektury monolitycznej: Krzemowe Kanapki	253
Monolit modułowy	253
Mikrojądro	254
Studium przypadku architektury rozproszonej: „Po raz pierwszy, po raz drugi, sprzedane!”	255

Część III. Techniki i umiejętności miękkie

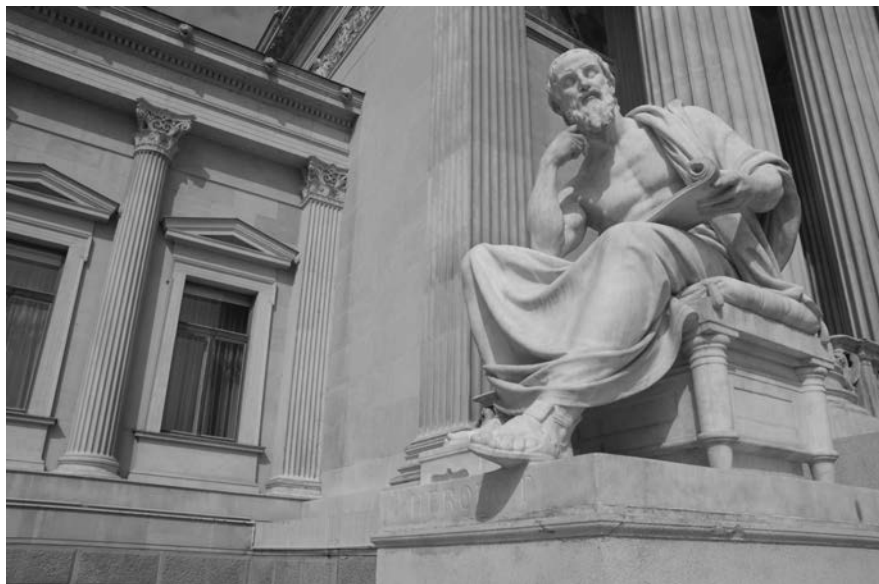
19. Decyzje architektoniczne.....	261
Antywzorze w decyzjach architektonicznych	261
Antyworzec Obrona Swojego Stanowiska	261
Antyworzec Dzień Świstaka	262
Antyworzec Architektura Sterowana Wiadomościami E-mail	263
Istotność architektoniczna	264
Rejestr decyzji architektonicznych	264
Podstawowa struktura	265
Przechowywanie dokumentów ADR	270
ADR jako dokumentacja	272
Wykorzystanie dokumentów ADR do standaryzacji	272
Przykład	273
20. Analiza ryzyka w architekturze	275
Macierz ryzyka	275
Ocena ryzyka	276
Risk storming	279
Identyfikacja	281
Konsensus	281
Ograniczanie	283
Analizy ryzyka historyjek w metodykach zwinnych	285

Przykłady risk stormingu	285
Dostępność	286
Elastyczność	288
Bezpieczeństwo	289
21. Tworzenie diagramów i prezentacja architektury.....	293
Diagramy	294
Narzędzia	294
Standardy tworzenia diagramów: UML, C4 i ArchiMate	296
Wskazówki dotyczące sporządzania diagramów	297
Prezentacja	298
Manipulowanie czasem	299
Diagramy przyrostowe	299
Karty informacyjne a prezentacje	300
Slajdy to połowa przekazu	302
Niewidoczność	302
22. Zwiększanie efektywności zespołów	303
Granice zespołów	303
Osobowości architektów	304
Maniak kontroli	304
Architekt fotelowy	306
Skuteczny architekt	307
Poziom kontroli	308
Znaki ostrzegawcze w zespole	312
Wykorzystanie list kontrolnych	315
Lista kontrolna gotowości kodu	317
Lista kontrolna testów jednostkowych i funkcjonalnych	318
Lista kontrolna wydania oprogramowania	318
Udzielanie wskazówek	319
Podsumowanie	321
23. Umiejętności negocjacyjne i zdolności przywódcze.....	323
Negocjacje i facylitacja	323
Negocjacje z interesariuszami biznesowymi	324
Negocjacje z innymi architektami	326
Negocjacje z programistami	327
Architekt oprogramowania jako lider	328
Cztery aspekty architektury	328
Bądź pragmatyczny, ale zarazem wizjonerski	330
Przewodzenie zespołom poprzez dawanie przykładu	331

Integracja z zespołem	335
Podsumowanie	338
24. Rozwijanie ścieżki kariery zawodowej	339
Zasada 20 minut	339
Opracowanie osobistego radaru	341
Radar technologiczny firmy ThoughtWorks	341
Wizualizacje open source	344
Korzystanie z mediów społecznościowych	345
Kilka rad na pożegnanie	346
A. Pytania sprawdzające	347

Myślenie architektoniczne

Architekt widzi rzeczy inaczej niż programista, tak samo jak meteorolog może postrzegać chmury inaczej niż artysta. Nazywamy to **myśleniem architektonicznym**. Niestety zbyt wielu architektów uważa, że myślenie architektoniczne to po prostu „myślenie o architekturze”, co zostało ukazane na rysunku 2.1.



Rysunek 2.1. Myślenie architektoniczne (iStockPhoto)

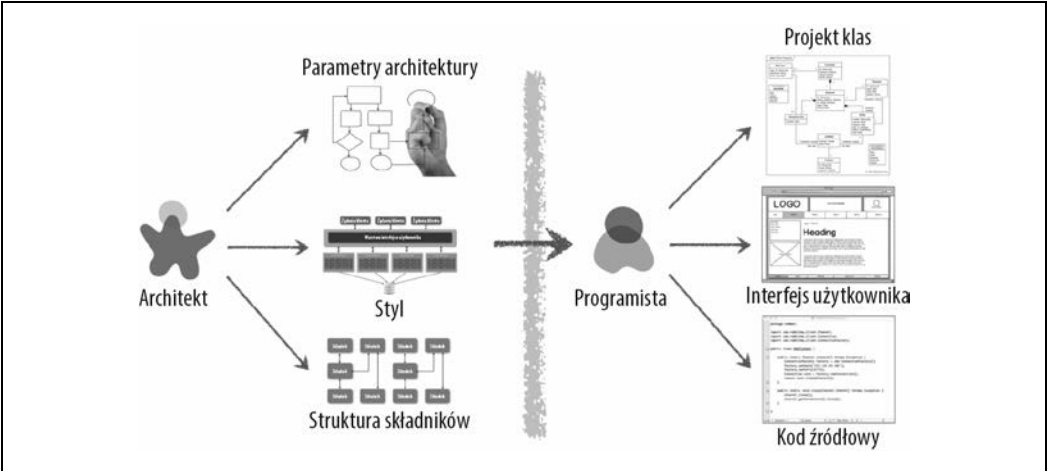
Myślenie architektoniczne to coś więcej. To patrzeć na rzeczy okiem architekta, czyli architektoniczny punkt widzenia. Istnieją cztery główne aspekty myślenia jak architekt. Po pierwsze, chodzi o zrozumienie różnicy między architekturą i projektowaniem oraz umiejętność współpracy z zespołami programistów, aby architektura zadziałała. Po drugie, chodzi o szeroki zakres wiedzy technicznej przy jednoczesnym zachowaniu pewnego poziomu technicznej głębi, pozwalającej architektowi dostrzec rozwiązania i możliwości, których inni nie dostrzegają. Po trzecie, chodzi o zrozumienie, analizę i kompromisy między różnymi rozwiązaniami i technologiami. I wreszcie po czwarte, chodzi o zrozumienie znaczenia czynników biznesowych i tego, jak przekładają się one na kwestie architektoniczne.

W tym rozdziale przeanalizujemy te cztery aspekty myślenia jak architekt i patrzenia na rzeczy okiem architekta.

Architektura a projekt

Różnica między architekturą a projektem jest często myląca. Gdzie kończy się architektura, a zaczyna projekt? Jakie obowiązki ma architekt, a jakie programista? Myślenie jak architekt to świadomość różnicy między architekturą a projektem i dostrzeganie, jak te dwie rzeczy ściśle się ze sobą łączą, tworząc rozwiązania problemów biznesowych i technicznych.

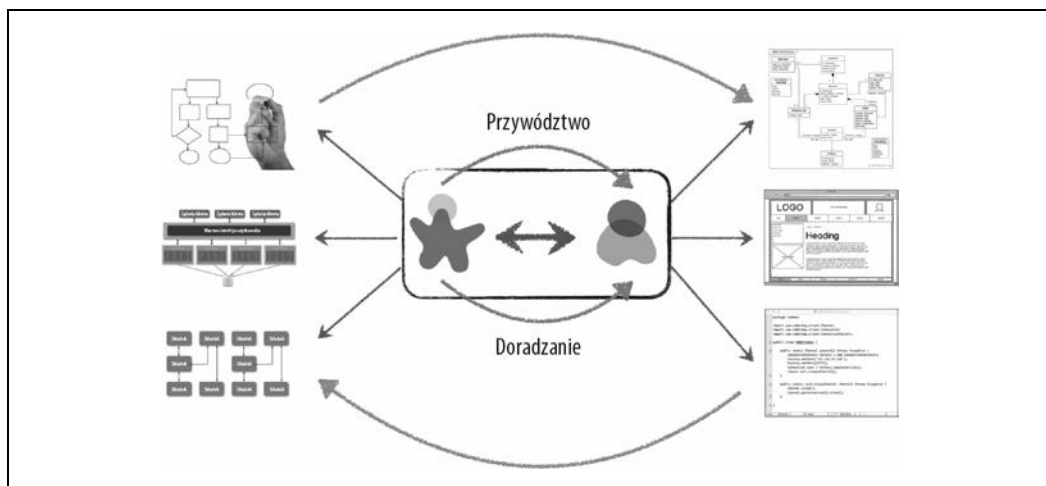
Spójrz na rysunek 2.2, który przedstawia porównanie tradycyjnych obowiązków architekta z obowiązkami programisty. Jak pokazano na diagramie, architekt jest odpowiedzialny za takie rzeczy jak analiza wymagań biznesowych w celu wyodrębnienia i zdefiniowania parametrów architektury („-ości”), wybór wzorców i stylów architektonicznych, które pasowałyby do danej problematyki oraz tworzenie składników (elementów konstrukcyjnych systemu). Artefakty stworzone na podstawie tych czynności są następnie przekazywane zespołowi programistów, który jest odpowiedzialny za tworzenie diagramów klas dla każdego składnika, tworzenie ekranów interfejsu użytkownika oraz opracowanie i testowanie kodu źródłowego.



Rysunek 2.2. Tradycyjne spojrzenie na architekturę i projekt

Istnieje kilka kwestii związanych z tradycyjnym modelem odpowiedzialności przedstawionym na rysunku 2.2. Ilustracja ta pokazuje dokładnie, dlaczego architektura rzadko się sprawdza. Wszystkie problemy związane z architekturą są spowodowane przez jednokierunkową strzałkę przechodzącą przez wirtualne i fizyczne bariery oddzielające architekta od programisty. Decyzje podejmowane przez architekta czasami nigdy nie trafiają do zespołów programistycznych, a podejmowane przez programistów decyzje, które zmieniają architekturę, rzadko do niego wracają. W tym modelu architekt jest oderwany od zespołów programistycznych i taka architektura rzadko zapewnia to, do czego była pierwotnie przeznaczona.

Aby architektura działała, należy przełamać zarówno fizyczne, jak i wirtualne bariery istniejące między architektami i programistami, tworząc w ten sposób silną, dwukierunkową relację między architektami i zespołami programistycznymi. Architekt i programista muszą być w tym samym wirtualnym zespole, co zostało ukazane na rysunku 2.3. Model ten nie tylko ułatwia silną, dwukierunkową komunikację między architekturą i rozwijaniem oprogramowania, ale także umożliwia architektowi doradzanie i udzielanie wsparcia programistom w zespole.

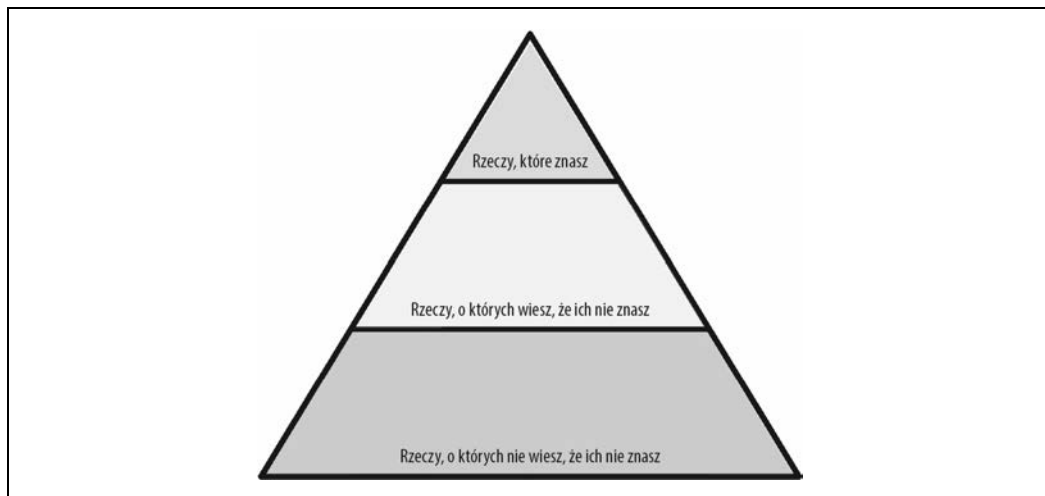


Rysunek 2.3. Tworzenie architektury poprzez współpracę

W odróżnieniu od staromodnych, kaskadowych podejść do statycznej i sztywnej architektury oprogramowania architektura dzisiejszych systemów zmienia się i ewoluuje w każdej iteracji czy fazie projektu. Ścisła współpraca między architektem i zespołem programistycznym ma kluczowe znaczenie dla sukcesu każdego projektu programistycznego. Gdzie więc kończy się architektura, a zaczyna projektowanie? Nie ma takiego punktu. Wspólnie są częścią kręgu życia w projekcie programistycznym i zawsze muszą być ze sobą zsynchronizowane, aby odnieść sukces.

Rozpiętość techniczna

Zakres szczegółowości technicznej różni się w przypadku programistów i architektów. W odróżnieniu od programisty, którego praca wymaga znacznej **głębi technicznej**, architekt oprogramowania musi dysponować znaczną **rozpiętością techniczną**, aby myśleć jak architekt i postrzegać rzeczy z punktu widzenia architektury. Ukazuje to piramida wiedzy przedstawiona na rysunku 2.4, która obejmuje całość wiedzy technicznej na świecie. Okazuje się, że rodzaj informacji, jakie powinny być wartościowe dla technologa, różni się w zależności od etapu jego kariery zawodowej.



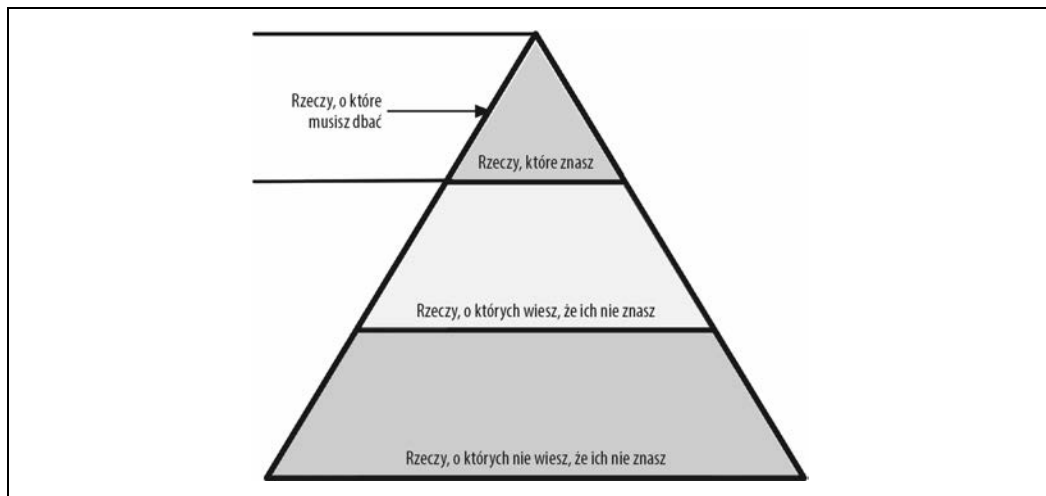
Rysunek 2.4. Piramida reprezentująca całą wiedzę

Jak widać na rysunku 2.4, każda osoba może podzielić całą swoją wiedzę na trzy części: *rzeczy, które znasz*, *rzeczy, o których wiesz, że ich nie znasz* i *rzeczy, o których nie wiesz, że ich nie znasz*.

Rzeczy, które znasz, to technologie, platformy, języki i narzędzia, które technolog wykorzystuje na co dzień podczas wykonywania swojej pracy, takie jak znajomość języka Java przez programistę piszącego w tym języku. *Rzeczy, o których wiesz, że ich nie znasz*, obejmują rzeczy, o których technolog niewiele wie lub o których słyszał, ale ma w nich niewielkie lub zerowe doświadczenie. Dobrym przykładem tego poziomu wiedzy jest język programowania Clojure. Większość technologów słyszała o Clojure i wie, że jest to język programowania oparty na Lispie, ale nie potrafi kodować w tym języku. *Rzeczy, o których nie wiesz, że ich nie znasz*, to największa część trójkąta wiedzy i obejmuje cały szereg technologii, narzędzi, platform i języków, które byłyby idealnym rozwiązaniem problemu, który technolog stara się rozwiązać, ale nie wie on nawet, że takie rzeczy istnieją.

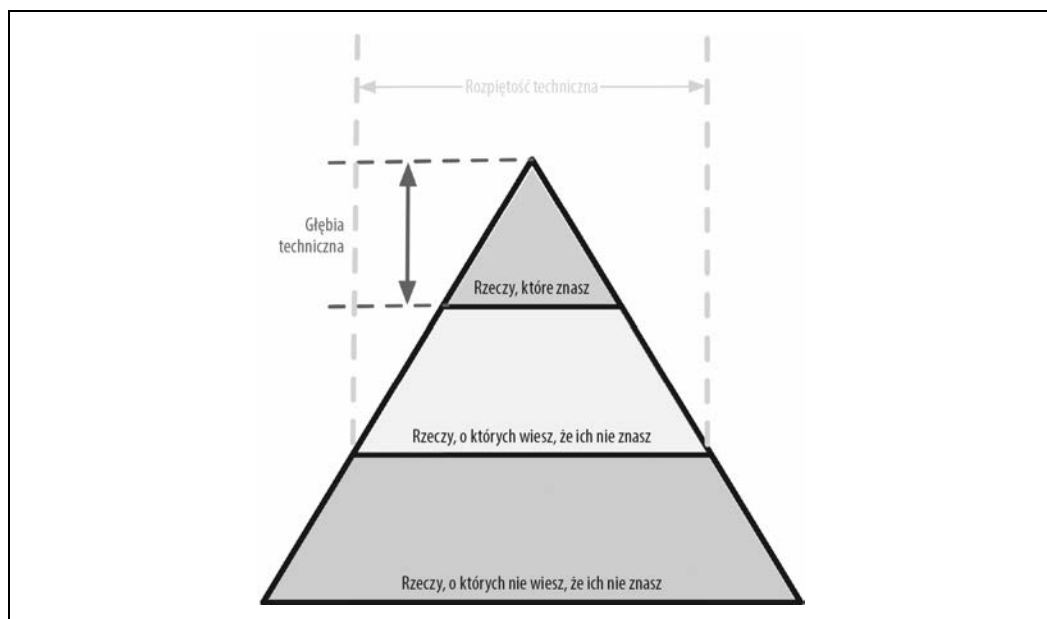
Początki kariery programisty koncentrują się na rozbudowie szczytu piramidy w celu zdobycia doświadczenia i wiedzy specjalistycznej. Jest to idealny punkt wyjścia, ponieważ programiści potrzebują więcej perspektywy, wiedzy praktycznej i doświadczenia zawodowego. Rozszerzenie górnej części przy okazji rozszerza środkową część; w miarę jak programiści stykają się z coraz większą liczbą technologii i związanych z nimi artefaktów, powiększa się ich zasób *rzeczy, o których wiedzą, że ich nie znają*.

Poszerzanie wierzchołka piramidy z rysunku 2.5 jest korzystne, ponieważ wiedza specjalistyczna jest ceniona. Jednak *rzeczy, które znasz*, to również *rzeczy, o które musisz dbać* — w świecie oprogramowania nic nie jest statyczne. Jeśli programista zostanie ekspertem w Ruby on Rails, jego wiedza specjalistyczna nie będzie trwała, gdy porzuci Ruby on Rails na rok lub dwa. Rzeczy na szczycie piramidy wymagają poświęcenia czasu na utrzymanie wiedzy specjalistycznej. Ostatecznie rozmiar wierzchołka piramidy danej osoby to jej **głębia techniczna**.



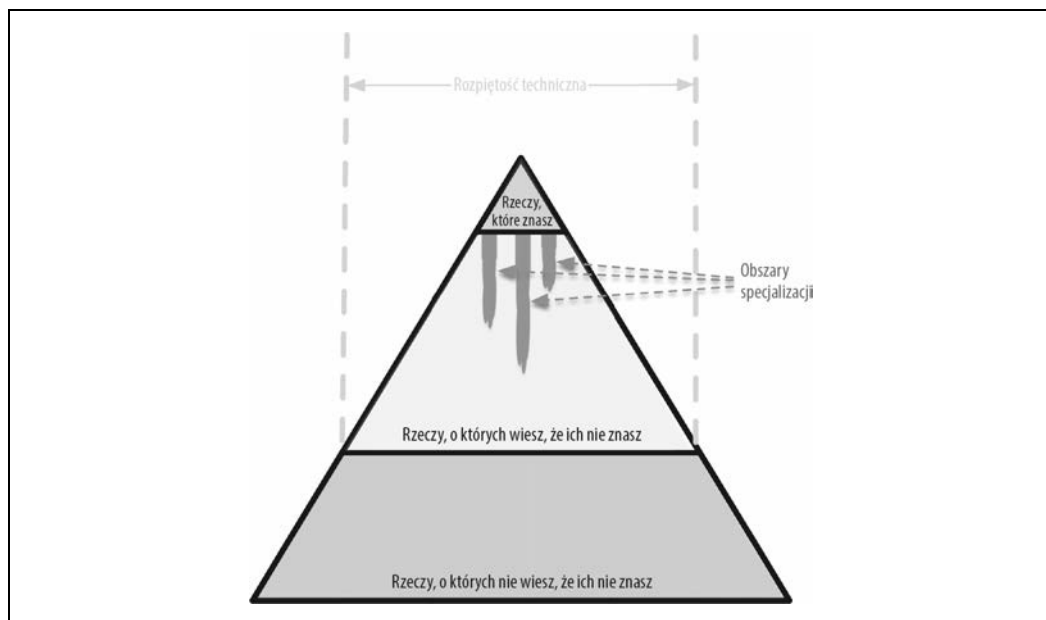
Rysunek 2.5. Programiści muszą dbać o wiedzę specjalistyczną, aby ją zachować

Jednak charakter wiedzy zmienia się, gdy programiści przeobrażają się w architektów. Duża część wartości architekta to *rozległa* znajomość technologii i sposobów jej wykorzystania do rozwiązywania konkretnych problemów. Dla przykładu, dla architekta korzystniejsza jest wiedza o tym, że istnieje pięć rozwiązań konkretnego problemu, niż dysponowanie specjalistyczną wiedzą tylko w przypadku jednego z nich. Dla architektów najważniejsze znaczenie mają dwie części: górna i środkowa; stopień, w jakim część środkowa przenika do części dolnej, stanowi **rozpiętość techniczną** architekta, co zostało ukazane na rysunku 2.6.



Rysunek 2.6. To, co ktoś zna, to głębina techniczna, a to, o czym ktoś wie, to rozpiętość techniczna

Dla architekta *rozpiętość* jest ważniejsza niż *głębina*. Ponieważ architekci muszą podejmować decyzje, które dopasowują możliwości do ograniczeń technicznych, ważne jest posiadanie przez nich rozległej wiedzy na temat szerokiej gamy rozwiązań. Dlatego też dla architekta mądrym sposobem działania jest poświęcenie części ciężko zdobytej wiedzy specjalistycznej i wykorzystanie tego czasu do poszerzenia swojego portfolio, co ukazuje rysunek 2.7. Jak widać na diagramie, niektóre obszary wiedzy specjalistycznej (prawdopodobnie te ze szczególnie przyjemnych obszarów technologii) zostaną zachowane, podczas gdy inne z pożytkiem zanikną.



Rysunek 2.7. Zwiększona rozpiętość i zmniejszona głębina dla roli architekta

Nasza piramida wiedzy pokazuje, jak fundamentalnie różna jest rola architekta w porównaniu z rolą programisty. Programiści poświęcają całą karierę na pogłębianie swojej wiedzy specjalistycznej, a przejście do roli architekta oznacza zmianę tej perspektywy, co dla wielu osób bywa trudne. To z kolei prowadzi do dwóch często spotykanych dysfunkcji: po pierwsze, architekt stara się zachować wiedzę specjalistyczną w wielu różnych obszarach, nie odnosząc przy tym sukcesów w żadnym z nich, mimo wielu wysiłków. Po drugie, objawia się to w postaci nieaktualnej wiedzy specjalistycznej — błędnego wrażenia, że przestarzałe informacje są wciąż aktualne. Widzimy to często w dużych firmach założonych przez programistów, którzy przeniesli się na stanowiska kierownicze, a mimo to nadal podejmują decyzje technologiczne na podstawie dawnych kryteriów (patrz ramka „Antywzorzec zamrożonego jaskiniowca”).

Architekci powinni skupić się na rozpiętości technicznej, tak aby mieli większy kołczan, z którego mogliby wyciągać strzały. Programiści wchodzący w rolę architekta mogą być zmuszeni do zmiany sposobu postrzegania zdobywania wiedzy. Każdy programista powinien w swojej karierze zastanowić się nad zrównoważeniem swojego portfolio wiedzy, uwzględniając głębść i rozpiętość.

Antywzorzec zamrożonego jaskiniowca

Często spotykany w naturze behawioralny **antywzorzec zamrożonego jaskiniowca** opisuje architekta, który zawsze wraca do swoich ulubionych, irracjonalnych obaw o każdą architekturę. Jedną z koleżanek Neala pracowała nad systemem charakteryzującym się scentralizowaną architekturą. Jednak za każdym razem, gdy dostarczano projekt architektom klienta, oni uporczywie pytali: „A co, jeśli stracimy Włochy?”. Kilka lat wcześniej pewien dziwaczny problem uniemożliwił centrali komunikowanie się z jej sklepami we Włoszech, powodując ogromne niedogodności. Choć prawdopodobieństwo pojawienia się tego problemu było bardzo małe, architekci mieli obsesję na punkcie tego konkretnego parametru architektury.

Na ogół ten antywzorzec przejawia się u architektów, którzy w przeszłości mieli nieprzyjemne doświadczenia na skutek złej decyzji lub nieoczekiwanego zdarzenia, co czyni ich szczególnie ostrożnymi na przyszłość. Chociaż ocena ryzyka jest ważna, powinna być również realistyczna. Zrozumienie różnicy między rzeczywistym a postrzeganym ryzykiem technicznym jest częścią procesu ciągłego uczenia się architektów. Myślenie jak architekt wymaga przewyciężenia tych pomysłów i doświadczeń „zamrożonego jaskiniowca”, dostrzeżenia innych rozwiązań i stawiania bardziej istotnych pytań.

Analiza kompromisów

Myślenie jak architekt polega na dostrzeganiu kompromisów w każdym rozwiązaniu, technicznym czy innym, i analizowaniu tych kompromisów w celu określenia, które rozwiązanie jest najlepsze. Cytując Marka (jednego z autorów tej książki):

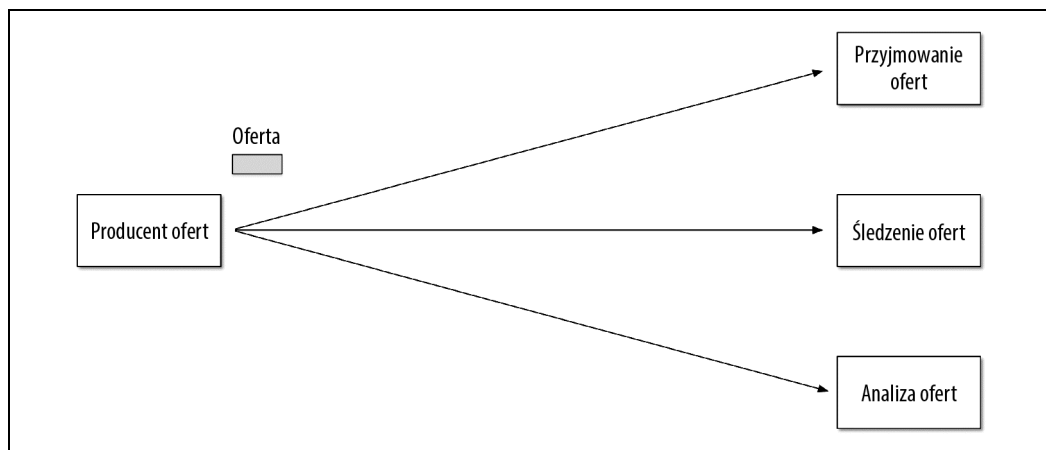
Architektura jest tym, czego nie można znaleźć w Google.

Wszystko w architekturze jest kompromisem, dlatego słynna odpowiedź na każde pytanie dotyczące architektury brzmi „to zależy”. Choć taka odpowiedź coraz bardziej denerwuje wiele osób, to niestety jest ona prawdziwa. Nie można znaleźć w Google odpowiedzi na pytanie, co będzie lepsze, REST czy komunikaty, albo czy mikrousługi to odpowiedni styl architektury, bo to zależy. Zależy od środowiska wdrożeniowego, czynników biznesowych, kultury organizacyjnej, budżetów, ram czasowych, umiejętności programistów i kilkudziesięciu innych czynników. Każde środowisko, sytuacja i problem są odmienne, dlatego też architektura jest tak trudna. Cytując Neala (kolejnego z autorów tej książki):

W architekturze nie ma dobrych i złych odpowiedzi — są tylko kompromisy.

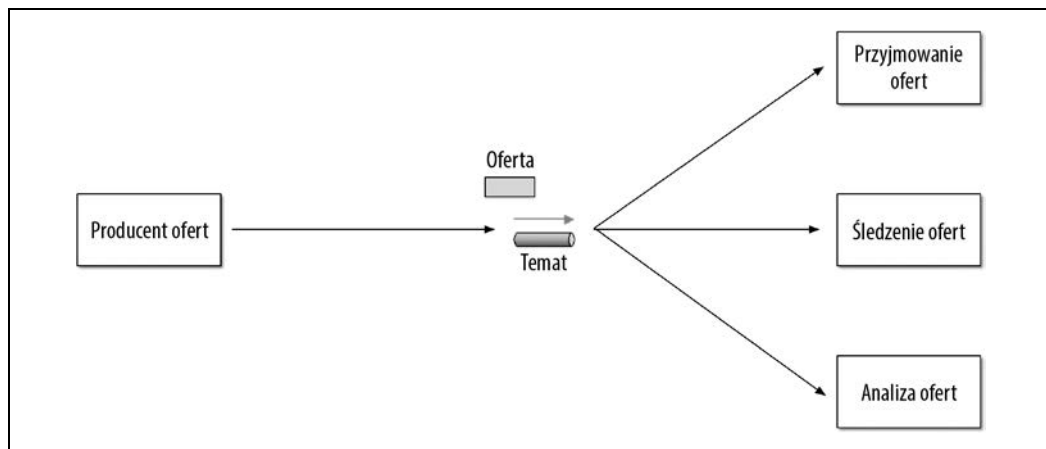
Rozważmy dla przykładu system aukcyjny, przedstawiony na rysunku 2.8, w którym ktoś składa ofertę kupna przedmiotu wystawionego na licytacji.

Usługa Producent ofert generuje ofertę od licytującego, a następnie wysyła kwotę tej oferty do usług Przyjmowanie ofert, Śledzenie ofert i Analiza ofert. Można to zrobić za pomocą kolejek w komunikacji typu punkt-punkt lub używając tematu w komunikacji typu publikacja-subskrypcja. Którego sposobu powinien użyć architekt? Nie znajdziesz odpowiedzi w Google. Myślenie architektoniczne wymaga od architekta przeanalizowania kompromisów związanych z każdą opcją i wybrania najlepszej w danej sytuacji.



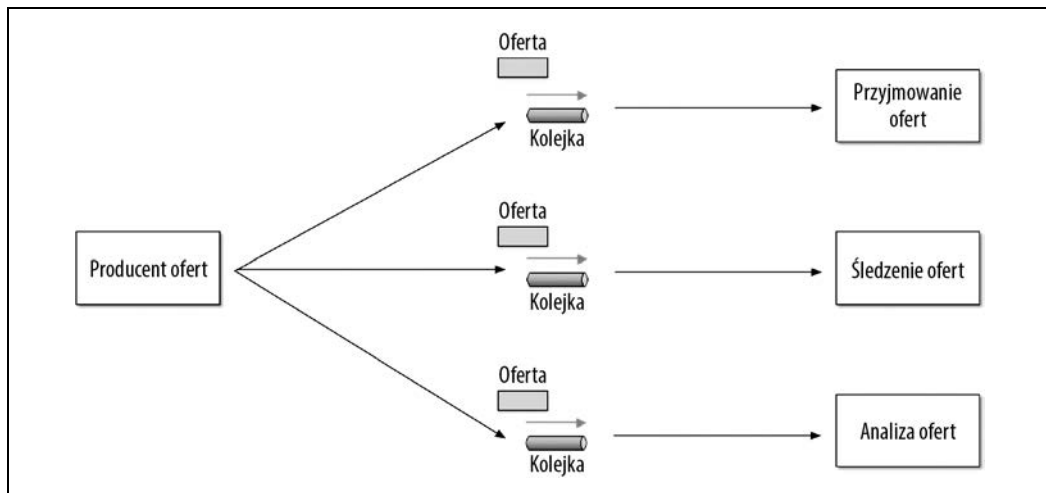
Rysunek 2.8. System aukcyjny jako przykład kompromisu — kolejki czy tematy?

Te dwa warianty komunikacji w systemie aukcyjnym zostały przedstawione na rysunkach 2.9 i 2.10, przy czym na rysunku 2.9 pokazano wykorzystanie tematu w modelu komunikacji typu publikacja-subskrypcja, a rysunek 2.10 przedstawia wykorzystanie kolejek w modelu komunikacji typu punkt-punkt.



Rysunek 2.9. Wykorzystanie tematu do komunikacji między usługami

Wyraźną zaletą (i pozornie oczywistym rozwiązaniem) problemu na rysunku 2.9 jest jego *architektoniczna rozszerzalność*. Producent ofert wymaga tylko jednego połączenia z tematem, w odróżnieniu od wariantu kolejkowego na rysunku 2.10, gdzie Producent ofert musi być połączony z trzema różnymi kolejkami. Gdyby do tego systemu została dodana nowa usługa o nazwie Historia ofert (ze względu na wymóg dostarczenia każdemu z licytujących historii wszystkich ofert, jakie złożyli w każdej aukcji), nie byłyby potrzebne żadne zmiany w istniejącym systemie. Nowa usługa Historia ofert może po prostu zasubskrybować temat, który zawiera już informacje o ofertach. W wariantcie kolejkowym, przedstawionym na rysunku 2.10, dla usługi Historia ofert wymagana byłaby jednak nowa



Rysunek 2.10. Wykorzystanie kolejek do komunikacji między usługami

kolejka, a usługa Producent ofert musiałaby zostać zmodyfikowana, aby wprowadzić dodatkowe połączenie do nowej kolejki. Chodzi o to, że korzystanie z kolejek wymaga znacznych zmian w systemie przy dodawaniu nowych funkcjonalności przetargowych, podczas gdy podejście tematyczne nie wymaga żadnych zmian w istniejącej infrastrukturze. Zauważ również, że w opcji z tematem usługa Producent ofert jest mniej sprzężona — nie wie, jak będą wykorzystywane informacje przetargowe ani przez jakie usługi. W wariantcie kolejkowym usługa Producent ofert dokładnie „wie”, w jaki sposób są wykorzystywane informacje przetargowe (i przez kogo), a więc jest bardziej sprzężona z systemem.

Przy tej analizie wydaje się oczywiste, że podejście tematyczne, wykorzystujące model komunikacji typu publikacja-subskrypcja, jest oczywistym i najlepszym wyborem. Jednak, cytując Richa Hickeya, twórcę języka programowania Clojure:

Programiści znają korzyści wszystkiego, ale nie znają kompromisów. Architekci muszą znać obie te rzeczy.

Myślenie architektoniczne to patrzeć na korzyści płynące z danego rozwiązania, ale także analizowanie związanych z nim negatywów czy kompromisów. Kontynuując przykład z systemem aukcyjnym — architekt oprogramowania przeanalizowałby negatywy rozwiązania tematycznego. Analizując różnice, należy przede wszystkim zwrócić uwagę na rysunek 2.9, gdzie za pomocą tematu *każdy* może uzyskać dostęp do danych ofertowych, co wprowadza ewentualny problem z dostępem do danych i ich bezpieczeństwem. W modelu kolejkowym, przedstawionym na rysunku 2.10, dane przesyłane do kolejki mogą być dostępne *tylko* dla konkretnego klienta otrzymującego ten komunikat. Gdyby jakaś nieuczciwa usługa nasłuchiwała w kolejce, oferty te nie zostałyby odebrane przez właściwą usługę i natychmiast zostałoby wysłane powiadomienie o utracie danych (a tym samym możliwym naruszeniu bezpieczeństwa). Innymi słowy, bardzo łatwo jest podsłuchiwać temat, ale nie kolejkę.

Oprócz kwestii bezpieczeństwa rozwiązanie tematyczne na rysunku 2.9 obsługuje tylko kontrakty homogeniczne. Wszystkie usługi otrzymujące dane przetargowe muszą zaakceptować ten sam kontrakt i zestaw danych. W wariantcie kolejkowym na rysunku 2.10 każdy klient może mieć swój własny kontrakt dostosowany do danych, których potrzebuje. Załóżmy dla przykładu, że nowa usługa

Historia ofert wymaga podania aktualnej ceny wywoławczej wraz z ofertą, ale żadna z pozostałych usług nie potrzebuje tych informacji. W tym przypadku kontrakt musiałby zostać zmodyfikowany, co miałoby wpływ na wszystkie inne usługi wykorzystujące te dane. W modelu kolejkowym byłby to osobny kanał, a zatem oddzielny kontrakt nie miałby wpływu na pozostałe usługi.

Kolejną wadą modelu tematycznego, przedstawionego na rysunku 2.9, jest to, że nie obsługuje on monitorowania liczby komunikatów w temacie, a tym samym nie daje możliwości autoskalowania. Jednak w opcji kolejkowej, przedstawionej na rysunku 2.10, każda kolejka może być monitorowana indywidualnie, a programowe równoważenie obciążenia można zastosować do każdego klienta ofert, dzięki czemu każdy z nich może być automatycznie skalowany niezależnie od siebie. Należy zauważyć, że ten kompromis jest związany z konkretną technologią, ponieważ protokół AMQP (ang. *Advanced Message Queuing Protocol*; <https://www.amqp.org/>) może obsługiwać programowe równoważenie obciążenia i monitorowanie z powodu oddzielenia wymiany (tego, co wysyła producent) od kolejki (tego, czego nasłuchuje klient).

Gdy weźmie się pod uwagę tę analizę kompromisów, które rozwiązanie jest teraz lepsze? A odpowiedź? To zależy! Tabela 2.1 zawiera podsumowanie tych kompromisów.

Tabela 2.1. Kompromisy związane z tematami

Zalety tematu	Wady tematu
Rozszerzalność architektoniczna	Kwestie dostępu do danych i ich bezpieczeństwo
Rozprzężenie usług	Brak kontraktów heterogenicznych
	Monitorowanie i skalowalność programowa

Chodzi o to, że **wszystko** w architekturze oprogramowania jest kompromisem: ma swoje wady i zalety. Myślenie jak architekt polega na przeanalizowaniu tych kompromisów, a następnie zadaniu pytania „co jest ważniejsze: rozszerzalność czy bezpieczeństwo?”. Wybór pomiędzy różnymi rozwiązaniami zawsze będzie zależeć od czynników biznesowych, otoczenia i wielu innych aspektów.

Czynniki biznesowe

Myślenie jak architekt polega na zrozumieniu czynników biznesowych, które są wymagane do skutecznego funkcjonowania systemu i przełożenia tych wymagań na parametry architektury (takie jak skalowalność, wydajność i dostępność). Jest to trudne zadanie, które wymaga od architekta posiadania pewnego poziomu wiedzy z dziedziny biznesu oraz zdrowych, opartych na współpracy relacji z kluczowymi interesariuszami biznesowymi. W książce poświęciliśmy temu zagadnieniu kilka rozdziałów. W rozdziale 4. definiujemy różne parametry architektury. W rozdziale 5. opisujemy sposoby identyfikowania i kwalifikowania parametrów architektury. Natomiast w rozdziale 6. opisujemy, jak mierzyć każdy z tych parametrów, aby zapewnić zaspokojenie potrzeb biznesowych systemu.

Zachowanie równowagi między architekturą a kodowaniem

Jednym z trudnych zadań stojących przed architektem jest znalezienie równowagi między samodzielnym pisanem kodu a planowaniem architektury oprogramowania. Głęboko wierzymy, że każdy architekt powinien kodować i być w stanie utrzymać pewien poziom głębi technicznej (patrz podrozdział „Rozpiętość techniczna” wcześniej w tym rozdziale). Choć może się to wydawać łatwym zadaniem, czasami jest to dość trudne do wykonania.

Pierwszą wskazówką w dążeniu do równowagi między samodzielnym pisanem kodu a byciem architektem oprogramowania jest unikanie tworzenia zatorów. Ma to miejsce wtedy, gdy architekt przejmuje kontrolę nad kodem należącym do ścieżki krytycznej projektu (zazwyczaj jest to kod tworzący podstawę struktury) i staje się zatorem dla zespołu. Dzieje się tak, ponieważ architekt nie jest pełnoetatowym programistą i dlatego musi balansować między rolą programisty (pisanie i testowanie kodu źródłowego) a rolą architekta (rysowanie diagramów, uczestniczenie w spotkaniach i... cóż, uczestniczenie w większej liczbie spotkań).

Jednym ze sposobów, w jaki skuteczny architekt oprogramowania może uniknąć tworzenia zatorów, jest przekazanie ścieżki krytycznej i kodu ramowego innym osobom w zespole programistycznym, a następnie skupienie się na kodowaniu jakiegoś elementu funkcjonalności biznesowej (usługi lub ekranu) od jednej do trzech iteracji podczas pracy nad projektem. W ten sposób powstają trzy pozytywne zjawiska. Po pierwsze, architekt zdobywa praktyczne doświadczenie w pisaniu kodu produkcyjnego, nie będąc jednocześnie zatorem dla zespołu. Po drugie, ścieżka krytyczna i kod ramowy są przekazywane zespołowi programistów (tam, gdzie powinny się znaleźć), dając im poczucie kontroli i możliwość lepszego zrozumienia trudniejszych części systemu. Po trzecie, i być może najważniejsze, architekt pisze ten sam kod źródłowy co programiści i dlatego jest w stanie lepiej się z nimi identyfikować, jeśli chodzi o trudności, jakie mogą napotkać w związku z procesami, procedurami i środowiskiem programistycznym.

Załóżmy jednak, że architekt nie jest w stanie tworzyć kodu wspólnie z zespołem programistycznym. Jak architekt oprogramowania może nadal być pomocny i utrzymywać pewien poziom głębi technicznej? Istnieją cztery podstawowe sposoby, w jaki architekt może być przydatny w pracy bez konieczności „praktykowania kodowania w domu” (choć zalecamy również praktykowanie kodowania w domu).

Pierwszym sposobem jest wykonanie częstego dowodzenia koncepcji (POC — ang. *proof-of-concept*). Praktyka ta nie tylko wymaga od architekta pisania kodu źródłowego, ale także pomaga w potwierdzeniu decyzji architektonicznej poprzez uwzględnienie szczegółów implementacji. Przykładowo, jeśli architekt utknął, próbując dokonać wyboru między dwoma rozwiązaniami buforowania, jedynym skutecznym sposobem na podjęcie decyzji jest opracowanie działającego przykładu dla każdego produktu buforowania i porównanie wyników. Pozwala to architektowi bezpośrednio poznać szczegóły implementacji oraz nakład pracy niezbędny do opracowania pełnego rozwiązania. Umożliwia to również architektowi lepsze porównanie parametrów architektury, takich jak skalowalność, wydajność czy ogólna odporność na błędy różnych rozwiązań buforowania.

Podczas pracy nad dowodami koncepcji radzimy, aby w miarę możliwości architekt pisał kod produkcyjny o najlepszej możliwej jakości. Zalecamy tę praktykę z dwóch powodów. Po pierwsze, dość często taki jednorazowy kod dowodu koncepcji trafia do repozytorium kodu źródłowego i staje się architekturą referencyjną lub przykładem do naśladowania dla innych. Ostatnią rzeczą, jakiej życzyłby sobie architekt, jest to, aby jego przykładowy, niedbały kod stał się odzwierciedleniem ich typowej pracy. Drugim powodem jest to, że pisząc kod dowodu koncepcji w jakości produkcyjnej, architekt nabiera praktyki pisania kodu o dobrej jakości i odpowiedniej strukturze zamiast ciągłego rozwijania złych praktyk kodowania.

Architekt może być przydatny także w taki sposób, że zajmuje się niektórymi kwestiami związanymi z długiem technicznym lub sprawami architektonicznymi, pozostawiając zespołowi opracowywanie funkcjonalności kluczowych dla użytkowników. Kwestie te mają zazwyczaj niski priorytet, więc jeśli architekt nie będzie mógł ukończyć prac związanych z długiem technicznym lub kwestiami architektonicznymi w ramach danej iteracji, nie będzie to oznaczać końca świata i na ogół nie będzie miało wpływu na powodzenie tej iteracji.

Kolejnym sposobem na utrzymanie umiejętności w zakresie praktycznego kodowania przy jednoczesnej pomocy zespołowi programistycznemu jest praca nad poprawkami błędów w ramach iteracji. Z pewnością nie jest to efektywna technika, ale umożliwia architektowi zidentyfikowanie miejsc w kodzie bazowym i ewentualnie w architekturze, w których występują problemy i słabe punkty.

Wykorzystanie automatyzacji poprzez opracowanie prostych narzędzi wiersza poleceń i analizatorów pomagających zespołowi programistów w wykonywaniu codziennych zadań to jeszcze jeden świetny sposób na utrzymanie praktycznych umiejętności kodowania przy jednoczesnym zwiększeniu efektywności zespołu. Szukaj powtarzających się zadań, które wykonują programiści, i zautomatyzuj proces. Będą Ci za to wdzięczni. Przykładem mogą być zautomatyzowane walidatory kodu źródłowego, pomagające zweryfikować specyficzne standardy kodowania, które nie są ujęte w innych testach analizujących kod, zautomatyzowane listy kontrolne oraz powtarzalne ręczne zadania związane z refaktoryzacją kodu.

Automatyzacja może mieć również formę analizy architektonicznej i funkcji dopasowania, aby zapewnić żywotność i zgodność architektury. Przykładowo, architekt może napisać kod Java w ArchUnit (<https://www.archunit.org/>) na platformie Java, aby zautomatyzować zachowanie zgodności z architekturą, lub napisać niestandardowe funkcje dopasowania (<https://evolutionaryarchitecture.com/>), aby zapewnić zachowanie zgodności z architekturą i jednocześnie zdobyć praktyczne doświadczenie. O tych technikach mówimy w rozdziale 6.

Ostatnią techniką umożliwiającą architektowi pracę nad kodem jest dokonywanie jego częstych przeglądów. Co prawda w tym przypadku architekt nie pisze kodu źródłowego, ale przynajmniej jest *zaangażowany* w jego tworzenie. Co więcej, dokonywanie przeglądów kodu przynosi dodatkowe korzyści w postaci możliwości zapewnienia zgodności z architekturą oraz poszukiwania możliwości doradzania i udzielania wsparcia członkom zespołu.

PROGRAM PARTNERSKI

— GRUPY HELION —



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

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

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

GRUPA
Helion 

W architekturze chodzi o ważne rzeczy (czymkolwiek to jest)

— Ralph Johnson

Rola architekta oprogramowania się zmienia. Dziś jest on odpowiedzialny za wiele spraw, zarówno technicznych, jak i tych wynikających ze specyfiki organizacji, której ma służyć aplikacja. Co więcej, jego rola nie kończy się na podjęciu decyzji projektowych na początku pracy. Nowoczesne style architektoniczne, takie jak mikroustugi, umożliwiają przyrostowe wprowadzanie zmian, co jednak wymusza ciągłe wypracowywanie kompromisów z innymi kwestiami. Ważne są kontekst, perspektywy i wciąż zmieniający się ekosystem dostępnych technologii.

Oto kompleksowy przewodnik po nowych aspektach architektury oprogramowania. Skorzysta z niego zarówno praktykujący architekt, chcący odświeżyć swoje podejście do tego zagadnienia, jak i programista aspirujący do roli architekta. W książce zaprezentowano szereg zagadnień, które mimo zmieniających się uwarunkowań pozostają podstawami, takich jak parametry architektury, wzorce architektoniczne, określanie składników, tworzenie diagramów, prezentowanie architektury, architektura ewolucyjna i wiele innych. Dokładnie wyjaśniono te zasady, które mogą być zastosowane do wszystkich zestawów rozwiązań technologicznych. Przedstawiono niezwykle ważną kwestię analizy kompromisów, która pozwala na obiektywną ocenę rozwiązań technologicznych. Duży nacisk położono na konieczność uwzględniania wszystkich innowacji ostatniej dekady.

Najciekawsze zagadnienia:

- wzorce architektoniczne
- etapy pracy przy projektowaniu nowoczesnej architektury
- umiejętności miękkie pomocne w pracy architekta
- nowe praktyki w projektowaniu architektury oprogramowania
- architektura oprogramowania jako dziedzina inżynierii

Mark Richards jest doświadczonym architektem oprogramowania. Zajmuje się projektowaniem i wdrażaniem mikroustug oraz innych systemów o architekturze rozproszonej. Założył tematyczny serwis dla programistów — *Developer to Architect*.

Neal Ford jest konsultantem w ThoughtWorks, międzynarodowej firmie konsultingowej z branży IT. Wcześniej był dyrektorem technicznym w spółce The DSW Group. Specjalizuje się w nauczaniu architektury oprogramowania.

  helion.pl  HELION SA ul. Kościuszki 1c 44-100 Gliwice tel.: 32 230 98 63 helion@helion.pl	<i>Sprawdź nasze szkolenia!</i>  AKADEMIA IT & BUSINESS HELIONSZKOLENIA.PL	KOD KORZYŚCI <i>Sięgnij po więcej!</i>   ISBN 978-83-283-7027-2  9 788328 370272
INFORMATYKA W NAJLEPSZYM WYDANIU		Cena: 79,00 zł