

O'REILLY®

Helion 

Tworzenie architektury oprogramowania

Wspieranie zespołów
w podejmowaniu trafnych decyzji



Andrew Harmel-Law

Tytuł oryginału: Facilitating Software Architecture: Empowering Teams
to Make Architectural Decisions

Tłumaczenie: Piotr Pilch

ISBN: 978-83-289-2750-6

© 2025 Helion S.A.

Authorized Polish translation of the English edition of *Facilitating Software Architecture*
ISBN 9781098151867 © 2025 Andrew Harmel-Law.

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.

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

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

helion.pl/user/opinie/twarop

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.

ul. Kościuszki 1c, 44-100 Gliwice

tel. 32 230 98 63

e-mail: helion@helion.pl

WWW: helion.pl (księgarnia internetowa, katalog książek)

Printed in Poland.

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

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

Spis treści

Słowo wstępne	17
Przedmowa	19
1. Praktyki dotyczące scentralizowanej architektury w zdecentralizowanym świecie ...	29
Kluczem do sukcesu jest zarówno praktyka, jak i końcowy rezultat architektury oprogramowania	29
Jakie są praktyki w przypadku tradycyjnej architektury?	31
Architekci z wieży z kości słoniowej	32
Architekci praktyczni	33
Na czym polega problem z obydwojema tradycyjnymi podejściami?	34
Pięć rewolucji, które uwolniły moc oprogramowania	35
Wpływ pięciu rewolucji na praktykę architektury	39
Rozkwit decentralizacji	40
Zmierzch scentralizowanych praktyk architektury	45
Co musi zapewniać każda nowa praktyka architektury?	49
Żadne podejście nie ochroni przed siłami chaosu	50
Architektury powinny uwzględniać niepewność	50
Architektury powinny dopuszczać emergencję	52
Podsumowanie	55

CZĘŚĆ I. Podstawowe zasady **57**

2. Projektowanie architektury to podejmowanie decyzji	59
Decyzje są podstawą architektury oprogramowania	60
Co stanowi decyzję architektoniczną?	61
Struktura	61
Cechy wielofunkcyjne	61
Zależności	62
Interfejsy	63
Techniki budowania	63

Przykłady decyzji architektonicznych i niearchitektonicznych	65
Kto podejmuje te decyzje architektoniczne?	66
Istotne decyzje architektoniczne	67
Co sprawia, że decyzja architektoniczna jest „istotna”?	67
Czego nie należy brać pod uwagę przy ocenie znaczenia z punktu widzenia architektury?	70
Znaczenie architektury w kontekście wdrożonego oprogramowania	71
Przykłady istotnych decyzji architektonicznych	72
Podsumowanie	74
3. Podejmowanie decyzji na dużą skalę	75
Procesy decyzyjne	75
Etap 1: wymagana decyzja	76
Etap 2: proces podejmowania decyzji	76
Etap 3: wdrożenie decyzji	78
Gdzie znajduje się stan równowagi władzy w procesach decyzyjnych?	78
Tradycyjne podejścia architektoniczne niewystarczająco angażują zespoły w etap tworzenia opcji w procesie decyzyjnym	79
Procesy decyzyjne na dużą skalę	80
Standardowe podejścia przy podejmowaniu decyzji na dużą skalę	81
Procesy decyzyjne w zrewolucjonizowanym świecie	88
Podsumowanie	92
4. Proces doradztwa w kwestii architektury	93
Potrzeba szybszego i zdecentralizowanego procesu decyzyjnego	93
Proces doradztwa w kwestii architektury: jedna reguła, dwie grupy doradcze i jedna umowa	94
Proces doradztwa w kwestii architektury jest szybki	96
Proces doradztwa w kwestii architektury jest zdecentralizowany	97
Proces doradztwa w kwestii architektury prowadzi do umowy społecznej	99
Dwa przykłady praktycznego zastosowania procesu doradztwa w kwestii architektury	100
Historia 1: zespół projektowy decyduje się na użycie przełączników wersji	101
Historia 2: architekt postanawia rozwiązać problem z przepływem pracy	108
Kluczowa rola porad	115
Porady to sugerowane wskazówki wraz z uzasadnieniem	115
Wspomaganie radą decydentów i osób tworzących opcje	118
Odpowiedzialnym czyni Cię nie udzielanie porady, lecz podejmowanie decyzji	119
Znaczenie rozmów	120
Znaczenie zaufania	121
Podsumowanie	122

5. Wdrażanie procesu doradztwa w kwestii architektury	123
Pierwsze kroki	123
Jeśli już masz uprawnienia do podejmowania decyzji	123
Jeśli obecnie nie masz uprawnień do podejmowania decyzji	124
Niezależnie od punktu wyjścia zacznij od czegoś niewielkiego	126
Pokonywanie początkowych wyzwań	127
Wyjaśnij proces doradztwa dotyczącego architektury	
wszystkim zaangażowanym osobom	127
Szukaj porad u właściwych osób	128
Pytaj właściwe osoby i otrzymuj uzasadnienie	130
Ustal wprost, kto odpowiada za poszczególne decyzje	131
Obawy dotyczące wiarygodności wynikające z procesu doradztwa	
w kwestii architektury	133
Brak pewności co do podejmowania decyzji	133
Brak pewności siebie w szukaniu i udzielaniu porad	133
Brak pewności co do pełnego obrazu sytuacji	134
Podsumowanie	134
 6. Dokumenty ADR	 136
Wprowadzenie do dokumentów ADR	136
Struktura dokumentu ADR	138
Tytuł	139
Elementy meta	139
Decyzja	140
Kontekst	141
Opcje	142
Konsekwencje	143
Porady	145
Przygotowywanie dokumentu ADR w celu wsparcia podejmowania decyzji	146
Krok 1: utwórz pusty dokument ADR i ustaw jego metadane	147
Krok 2: utwórz kontekst	148
Krok 3: opracuj opcje i rozważ ich konsekwencje	151
Krok 4: zaproponuj wybraną opcję	156
Użycie dokumentów ADR w celu ułatwienia procesu doradztwa	157
Kiedy i jak szukać porad?	157
Przygotowywanie sekcji porad	159
Uzyskiwanie porad	160
Aktualizowanie dokumentu ADR w celu uwzględnienia wkładu innych osób	162
Podejmowanie decyzji i finalizowanie dokumentu ADR	167
Wybierz opcję decyzji	167
Utwórz sekcję swojej decyzji i zaktualizuj tytuł	169
Zmiana statusu dokumentu ADR na Zaakceptowano	170
Poinformowanie o decyzji	170

Cykl życia dokumentu ADR	170
Standardowe statusy: Wersja robocza, Proponowane, Przyjęto i Zastąpiono	171
Niestandardowe statusy dokumentu ADR	172
Zarządzanie dokumentami ADR	174
Fundamentalne aspekty zarządzania dokumentami ADR	174
Dokumenty ADR na stronach wiki i pliki tekstowe z formatowaniem w systemie kontroli wersji	175
Dokumenty ADR w systemach obsługi zgłoszeń	175
Nadzór nad dokumentami ADR	176
Podsumowanie	178

CZĘŚĆ II. Kształtowanie i rozwijanie kultury zdecentralizowanego zaufania **179**

7. Zastępowanie hierarchii zdecentralizowanym zaufaniem	181
Zmiana zakresu odpowiedzialności	182
Większość tradycyjnych praktyk nadzoru staje się przestarzała	182
Proces doradztwa jest zgodny z istniejącymi strukturami architektury korporacyjnej	183
Bezpośrednia odpowiedzialność za dokument ADR jako zabezpieczenie przed lekkomyślnością	184
Zespoły nie muszą podejmować decyzji	184
Rozszerzanie obszaru praktyki architektury	185
Tworzenie przestrzeni dla kultury uczenia się i zaufania	186
Dwa przykłady zaufania (lub jego braku) w praktyce	186
Nie można zakładać, że kultura zaufania i nauki pojawi się sama z siebie	187
Kulturę zaufania i nauki trzeba starannie kształtować	187
Zmieniająca się dynamika w firmach o różnej wielkości	188
Adaptowanie elementów wspierających w celu ochrony przestrzeni zaufania	191
Ochrona przestrzeni praktyki architektury	192
Brak standardowego zalecenia dotyczącego elementów wspierających	192
Dostosuj dowolny element wspierający	193
Przemyśl swoją motywację, zanim coś dodasz	193
Netflix: przykład sposobu myślenia zmierzającego do znalezienia przepływu	194
Strzeż się zwodniczych obietnic pewności i przewidywalności	195
Eksperyment w celu znalezienia przepływu w Twojej firmie	196
Podsumowanie	196
8. Forum porad architektonicznych	198
Wprowadzenie do forum porad architektonicznych	198
Prosty plan spotkania	199
Czym forum porad różni się od tradycyjnych spotkań dotyczących architektury?	200

Prowadzenie forum porad architektonicznych	201
Przed rozpoczęciem forum porad architektonicznych	201
Udostępnianie planu	201
Rozpoczęcie sesji forum porad architektonicznych	202
Prezentowanie dokumentu ADR	203
Udzielanie i przyjmowanie porad	203
Rejestrowanie porad	204
Dynamiczne elementy społeczne na forum porad	204
Interakcje związane z architekturą są z natury antagonistyczne i hierarchiczne	205
Argumentacja koalescencyjna: alternatywa podejścia konfrontacyjnego	206
Proces doradztwa architektonicznego i dokumenty ADR jako spójne podejście do podejmowania decyzji	207
Fora porad przyspieszają skuteczną dynamikę grup	209
Doświadczenie w przypadku osób udzielających porad	210
Doświadczenie w przypadku osób poszukujących porady	211
Doświadczenie w przypadku uczących się obserwatorów	212
Wspólne uczestnictwo w praktycznej kreatywności	213
Fora porad architektonicznych wspierają integrację koncepcyjną i spójność społeczną	214
Decentralizacja wykonywania przy jednoczesnej centralizacji koordynacji	214
Obszerna wiedza z danej dziedziny: kiedy centralizacja sprawdza się najlepiej?	216
Przejrzystość w kwestii spójności społecznej i zaufania	217
Rytuał wzmacniania rytmu	221
Rozpoczynanie procesu doradztwa z wykorzystaniem forum porad	223
Zakres działań	223
Użycie forum porad do rozpoczęcia procesu doradztwa	223
Kto zwołuje pierwsze forum porad?	224
Czym pierwsze forum porad będzie się różnić od kolejnych?	224
Podsumowanie	226
9. Możliwe do przetestowania wymagania CFR oraz strategia technologiczna	227
Znaczenie dopasowania organizacyjnego	227
Dopasowanie nie gwarantuje skuteczności	228
Wykrywanie niedostatecznego dopasowania organizacyjnego	231
Cztery sposoby zapewnienia dopasowania	233
Istnieje dopasowanie, gdy nic Cię nie zaskakuje	234
Minimalnie satysfakcjonujące porozumienie	237
Wymagania CFR	238
Strategia technologiczna	248
Dążenie do minimalnie satysfakcjonującego porozumienia w Twojej firmie	254
Podsumowanie	255

10. Zasady architektoniczne oparte na doświadczeniu grupy	256
Pozyskuj zasady architektoniczne od wszystkich zaangażowanych osób	256
Przykłady zasad architektonicznych	257
Cechy dobrych zasad architektonicznych	259
Cechy kiepskich zasad architektonicznych	260
Zasady mogą być ukrytymi wymaganiami CFR	261
Zasady architektoniczne uzupełniają proces doradztwa i dokumenty ADR	261
Określanie zasad architektonicznych podczas warsztatów	262
Przygotowanie danych wejściowych na warsztaty dotyczące zasad	263
Przygotowywanie warsztatów dotyczących zasad	267
Prowadzenie warsztatów dotyczących zasad	271
Prezentacja zasad gotowych do zastosowania	279
Zachowanie użyteczności zasad	282
Informacje zwrotne z decyzji powinny wpływać na zasady architektoniczne	282
Aktualizacja i utrzymywanie zasad	284
Informacje zwrotne z decyzji mogą wpływać na strategię techniczną	286
Podsumowanie	286
11. Zastosowanie radaru technologicznego	288
Rozpoznawanie trendów technologicznych i wytycznych dotyczących wychwytywania	288
Radary technologiczne nieustannie zbierają i udostępniają wytyczne	289
Sposób działania radaru technologicznego firmy Thoughtworks	290
Twój wewnętrzny radar technologiczny będzie miał inną strukturę	294
Miejsce Twojego radaru w procesie doradztwa	294
Tworzenie własnego radaru technologicznego	297
Gromadzenie punktów	298
Sortowanie i sprawdzanie poprawności punktów	301
Doprecyzowanie punktów	303
Pozycjonowanie punktów	305
Dokumentowanie punktów	306
Publikowanie radaru	307
Aktualizowanie punktów	309
Okresowe przeglądy	309
Doraźne aktualizacje oparte na wspólnych doświadczeniach	311
Gromadzenie poprzednich punktów i ich historii	312
Podsumowanie	313

12. Sztuka podejmowania decyzji	317
Znaczenie „miękkich” aspektów	317
Ustalanie ram kontekstu	317
Wykluczanie jest równie ważne jak dołączanie	318
Pytania usprawniające ocenę sytuacji podczas ustalania ram	320
Zaangażuj właściwe zainteresowane osoby	321
Analiza opcji i ich konsekwencji	322
Nie pozwól, aby ramy ograniczały Twoją kreatywność	322
Szukaj inspiracji	323
Doprecyzuj kontekst i opcje dzięki poradom	324
Rozwijaj swoje umiejętności metapoznawcze	326
Ćwiczenie 1: podziel się swoimi powodami	327
Ćwiczenie 2: reagujesz odruchowo czy świadomie?	327
Ćwiczenie 3: celowo szukaj porad, które są dla Ciebie wyzwaniem	328
Radzenie sobie z osobami, które nie współpracują	329
Wybór opcji decyzyjnej	331
Pokonywanie strachu	332
Przewycięzanie uprzedzeń	334
Podejmowanie decyzji	335
Moment podejmowania decyzji	335
Gdy inni podejmują decyzję	336
Podsumowanie	339
 13. Radzenie sobie ze zmiennością architektoniczną	 340
Wpływ zmienności na praktykę architektury	341
Zmienność utrudnia projektowanie systemów	341
Zmienność to fundament projektowania oprogramowania	344
Skuteczne radzenie sobie ze zmiennością architektoniczną	345
Radź sobie ze zmiennością przez szybkie podejmowanie i testowanie decyzji	346
Dokładne testowanie decyzji w ich kontekście funkcjonalnym	347
Użycie przemieszczających się prototypów do testowania najwcześniejszych decyzji w ich kontekście funkcjonalnym	348
Radzenie sobie ze zmiennością przez podejmowanie mniejszych decyzji	349
Wpływ decyzji architektonicznych na przyszły przepływ	358
Dobra infrastruktura techniczna umożliwia podejmowanie małych decyzji	358
Dobra infrastruktura socjotechniczna także umożliwia podejmowanie drobnych decyzji	359
Wykorzystanie procesu doradztwa do eliminacji sprzężenia decyzji pod kątem przepływu	359

Wykorzystanie zmienności do eliminowania ryzyka i ujawniania wartości	360
Lepiej szybko i źle niż wolno i poprawnie	361
Zwróć uwagę na wartości odstające	362
Najpierw podejmuj najbardziej wartościowe decyzje	363
Podsumowanie	364
14. Zmienność i wzajemne powiązania decyzji	365
Odnajdywanie ścieżki w zmienności	365
Wprowadzenie do techniki spike	366
Technika spike umożliwia radzenie sobie ze zmiennością architektoniczną w tańszy sposób	367
Gdzie w procesie decyzyjnym stosuje się technikę spike?	367
Zastosowanie techniki spike w odniesieniu do dokumentów ADR	368
Dokumenty ADR z użytym elementem techniki spike nadal przestrzegają procesu doradztwa	374
Sprawdzanie wzajemnych powiązań decyzji	374
Decyzje to seria	375
Hierarchia decyzji jest odwrócona	379
Decyzje są atomowe	388
Decyzje to dwustronne konwersacje	391
Powiązane decyzje są społecznie skomplikowane	394
Podsumowanie	395

CZĘŚĆ IV. Umieszczenie elementu społecznego w centrum swojej praktyki architektury **397**

15. Zmiany w zakresie władzy i odpowiedzialności	399
Zmiany władzy nigdy nie są proste	399
Przejście w przypadku osób zdobywających władzę	401
Czy każdy wierzy w to, że ma prawo decydować?	401
Czy wszyscy rozumieją zakres swojej odpowiedzialności?	403
Dlaczego ludzie nie korzystają z udostępnionych im uprawnień?	407
Znaczenie bezpieczeństwa	409
Zmiana dla osób, które muszą dzielić się władzą	422
Strach wśród tych, którzy oddali władzę	423
Zachowanie osób niezadowolonych z podziału władzy	426
Zmiany bywają niewygodne	428
Podsumowanie	429

16. O przywództwie	430
Błędne wyobrażenia o przywództwie	431
Błędne przekonanie numer 1: przywództwo jest wrodzone	431
Błędne przekonanie numer 2: przywództwo jest związane z hierarchią	432
Błędne przekonanie numer 3: przywództwo działa tylko w jednym kierunku	432
Błędne przekonanie numer 4: przywództwo to zarządzanie	433
Czym jest przywództwo?	434
14 zasad Deminga	434
Przywództwo służebne	436
Przywództwo adaptacyjne	437
Przywództwo partnerskie	440
Wyzwania związane z przejściem na model przywództwa partnerskiego	441
Potrzeba ciągłego przywództwa moralnego	448
Radzenie sobie z wyzwaniami moralnymi	449
Bądź wrażliwy na wszystkie obecne kultury, ale nie bądź im podporządkowany	451
Nie ma „stałych liderów”	451
Podsumowanie	452
17. Dopasowanie procesu doradztwa do Twojej firmy	453
Subkultura inżynierii oprogramowania	453
Powód 1: projektowanie oprogramowania nie podąża za standardowymi modelami myślowymi procesu twórczego	454
Powód 2: tempo zmian w działach informatycznych jest znacznie wyższe niż gdziekolwiek indziej	455
Powód 3: punkty styku między inżynierią oprogramowania i resztą firmy są nieliczne i wyrażne	455
Powód 4: podział kulturowy między działem inżynierii oprogramowania i resztą firmy jest już akceptowany	455
„Bańki” procesu doradztwa	456
„Bańki” są widoczne tylko dla tych, którzy ich szukają	458
„Bańki” organizują się same	459
„Bańki” są przepuszczalne	461
Zewnętrzne oczekiwania wobec osób wewnątrz „bańki”	462
Oczywiste oczekiwania	462
Mniej oczywiste oczekiwania	464
„Bańki” zapewniają interfejs niezależny od implementacji	468
Kontrakt interfejsu „bańki”	468
Uwidocznij wartościowe działania związane z translacją	469
Powiększające się „bańki”	471
Rozwój przyrostowy zespół po zespole	471
Zadbaj o ochronę podstawowych celów swojej praktyki architektury	472

Dzielenie „baniek”, gdy stają się zbyt duże	473
Nic nie może rosnać w nieskończoność	473
Podziel „bańkę”, gdy zagrożone są główne cele Twojej praktyki architektury	474
Sposób podziału „bańki”	474
Zachowanie spójności między „bańkami”	477
Chroń i wspieraj różnice między „bańkami”	478
Podsumowanie	478

Projektowanie architektury to podejmowanie decyzji

Na tym etapie trzeba zagłębić się w fundamentalny element, który stanowi podstawę każdego podejścia do architektury oprogramowania, czyli decyzje. W tym rozdziale jasno zdefiniujemy, czym są decyzje architektoniczne. Natomiast w rozdziale 3. zostanie omówiony proces podejmowania decyzji, zarówno indywidualnie, jak i w grupach.

Zanim zacznę, chciałbym, żebyś coś dla mnie zrobił. Zamknij oczy i wyobraź sobie, że podejmowana jest decyzja dotycząca architektury oprogramowania. Gdy otworzysz oczy, zastanów się nad następującymi kwestiami:

- Jakie obrazy przyszły Ci na myśl?
- Czy to Ty podejmowałeś decyzję, czy ktoś inny, a może kilka osób?
- Czy osoba podejmująca decyzję zajmowała wysokie stanowisko w organizacji i „narzucała” ją z góry? A może była niżej w hierarchii od Ciebie? Czy może znajdowała się na tym samym poziomie co Ty?
- W jakim wieku była ta osoba? Czy była starsza, młodsza czy w podobnym wieku? Czy w ogóle zwróciłeś na to uwagę?
- Jakie umiejętności techniczne te osoby posiadają? Czy są one takie same jak Twoje, a może większe lub mniejsze? Czy dotychczas one programowały? W jakim języku? Czy znasz ten język? Czy uruchamiały swój kod w środowisku produkcyjnym? Na jaką skalę?
- Czy te osoby mają takie samo wykształcenie co Ty? A co z ich perspektywą i doświadczeniem? Czy są one takie same jak Twoje? Jeśli nie, w jaki sposób się różnią?

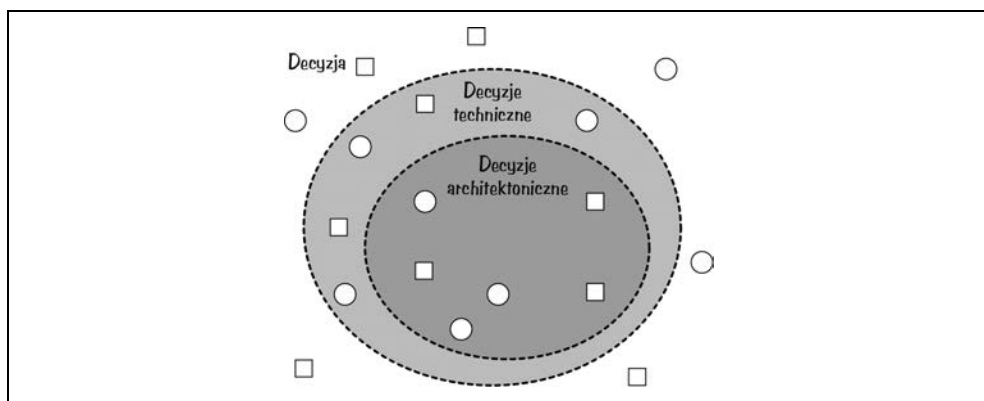
Miej w pamięci odpowiedzi na te pytania, czytając ten rozdział i odkrywając, co sprawia, że decyzja „architektoniczna” jest istotna. Po drodze zwrócę uwagę na powszechne założenia dotyczące decyzji, które mogą zaciemniać nasze myślenie. Możesz być zaskoczony tym, jak bardzo zmieni się Twoje obecne postrzeganie podejmowania decyzji dotyczącej architektury oprogramowania.

Decyzje są podstawą architektury oprogramowania

Niezależnie od wybranego podejścia praktyka architektury oprogramowania polega na podejmowaniu decyzji. Architektura oprogramowania jest pełna decyzji, tak wielu, że czasem możesz nawet nie zdawać sobie sprawy z ich istnienia. Podczas tworzenia oprogramowania musisz decydować o strukturze poszczególnych elementów, o tym, jak ma przebiegać proces ich tworzenia, itd. Musisz nawet zdecydować, czy Twój zespół powinien korzystać z zasobów zewnętrznych, takich jak biblioteka *open source*.

W tym kontekście architektury oprogramowania są sumą wszystkich decyzji architektonicznych podjętych w całym ich cyklu projektowym¹. Wraz z pojawieniem się architektur ewolucyjnych² decyzje architektoniczne podejmowane są nieustannie przez cały okres funkcjonowania systemu w sposób ciągły i nieprzewidywalny. To ogromna liczba decyzji nawarstwiających się jedna na drugiej, a każda z nich stanowi utrwalony zapis struktur władzy i pętli sprzężenia zwrotnego (lub ich braku)³.

W związku z tym kluczowe jest, aby wszyscy, zarówno osoby zajmujące się projektowaniem architektury, jak i programiści, dobrze rozumieli, czym *jest* decyzja w architekturze oprogramowania, a także *jak się ją podejmuje*. Takie zrozumienie sprzyja lepszemu zgraniu zespołów i pomaga wszystkim podejmować trafniejsze wybory. Na tej drodze jest kilka etapów, a pierwszym z nich jest jasne określenie, co sprawia, że dana decyzja jest architektoniczna, a nie tylko techniczna. Jak widać na rysunku 2.1, wszystkie decyzje architektoniczne są decyzjami technicznymi, lecz nie wszystkie decyzje techniczne to decyzje architektoniczne.



Rysunek 2.1. Wszystkie decyzje architektoniczne są decyzjami technicznymi, lecz nie wszystkie decyzje techniczne to decyzje architektoniczne

¹ Dla jasności: mam tutaj na myśli wyłącznie decyzje architektoniczne, a nie decyzje mające wpływ na architekturę. Nie oznacza to jednak, że te drugie nie wywierają istotnego wpływu.

² Zajrzyj do książki *Architektura ewolucyjna. Projektowanie oprogramowania i wsparcie zmian* (<https://helion.pl/książki/architektura-ewolucyjna-projektowanie-oprogramowania-i-wsparcie-zmian-wydanie-ii-neal-ford-rebecca-parsons-patrick-kua-pramod-sa,arche2.htm#format/d>) napisanej przez Neala Forda, Rebeccę Parsons, Patricka Kua i Pramoda Sadalage.

³ Temu tematowi poświęciłem wątek w serwisie Twitter: *Thoughts on the anthropology of software (power and freedom special edition)* (<https://x.com/al94781/status/1564288746228416513>).

Co stanowi decyzję architektoniczną?

W *Słowniku Cambridge* decyzję zdefiniowano jako „wybór dokonywany w odniesieniu do czegoś po rozważeniu kilku możliwości”⁴. Czym zatem jest decyzja architektoniczna⁵?

Szybkie wyszukiwanie w internecie ujawni *wiele* definicji architektury oprogramowania, a także prawie tyle samo definicji decyzji architektonicznej. Spośród dostępnych definicji ta zaproponowana przez Michaela Nygarda we wpisie *Dokumentowanie decyzji architektonicznych* (<https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions>) na blogu z 2011 r. okazała się dla mnie najbardziej przydatna w ramach praktyki architektury oprogramowania, głównie dlatego, że jest prosta do zrozumienia i łatwa do zapamiętania.

Autor stwierdza: „[...] decyzje istotne pod względem architektonicznym⁶: *te, które wpływają na strukturę, charakterystyki niefunkcjonalne, zależności, interfejsy lub techniki konstrukcyjne*” [wyróżnienie moje]. Definicja ta wyszczególnia pięć kryteriów identyfikujących decyzję architektoniczną. Przyjrzyjmy się bliżej każdemu z nich.

Struktura

Przed wszystkim decyzje strukturalne są architektoniczne. Wydaje się to oczywiste, ponieważ odnosi się do organizacji części systemu oprogramowania, ich funkcji oraz sposobu, w jaki ze sobą współdziałają (lub nie).

Im bardziej architektura jest zdecentralizowana, tym wyraźniej widoczne są jej „części”, głównie dlatego, że interakcje między nimi zwykle odbywają się za pośrednictwem sieci (np. za pomocą protokołu HTTP lub kolejki komunikatów). Jednakże aspekt „części” strukturalnych ma zastosowanie również w innych stylach architektonicznych. „Części” to zarówno moduły w „modularnym monolicie”, jak i zbiór mikrousług. W obu przypadkach nadal istnieją połączenia między nimi, choć są one mniej widoczne w ramach wywołań funkcji środowiska wykonawczego.

Cechy wielofunkcyjne

W trakcie pracy nad dowolnym oprogramowaniem zauważysz, że nie da się uchwycić wszystkich cech (czy wymagań), używając wyłącznie formatu funkcjonalnego. Istnieją cechy, które muszą zapewniać Twoje systemy, a które nie pasują do standardowego formatu historyjki użytkownika⁷

⁴ *Słownik Cambridge*, hasło *decision* (<https://dictionary.cambridge.org/dictionary/english/decision>), stan z 28 kwietnia 2023.

⁵ Inne decyzje mogą dotyczyć produktów, finansów, zatrudniania, struktury organizacyjnej i personelu itp. Choć wszystkie one są ważne, zajmuję się tutaj wyłącznie decyzjami architektonicznymi.

⁶ W dalszej części tego rozdziału powrócę do znaczenia istotnych decyzji architektonicznych. Na razie potraktuj definicję po prostu jako całość.

⁷ Najlepsze wprowadzenie do historyjek użytkownika można znaleźć w rozdziale 6. książki *Mapowanie historyjek użytkownika* (<https://helion.pl/ksiazki/mapowanie-historyjek-uzytkownika-przepis-na-produkt-idealny-jeff-patton,maphiv.htm#format/e>) Jeffa Pattona i Roya McCreya (Helion, 2016).

lub przypadku użycia⁸ z pojedynczą funkcją. Niektóre wymagania po prostu nie definiują konkretnego przypadku czegoś. Co np. z wymaganiami dotyczącymi wydajności, bezpieczeństwa, przepisów, skalowalności, możliwości obsługi, użyteczności, odporności i kosztów eksploatacji?

Mógłbym tak długo wymieniać. Tego typu wymagania, zamiast określać zachowania różnych rzeczy, pozwalają ocenić działanie całego systemu.

Cechy te są również powszechnie określane jako „wymagania niefunkcjonalne” lub „atrybuty jakości”. Oba terminy nie są jednak pozbawione problemów⁹. Nygard określa je mniej problematycznie jako „cechy niefunkcjonalne”, ale wciąż pojawia się tu słowo „nie”, które sugeruje brak czegoś. W książce będę używał terminu *cechy wielofunkcyjne* (ang. *cross-functional characteristics*), które są określane przez wymagania wielofunkcyjne CFR (ang. *Cross-Functional Requirements*). Uważam, że dokładniej oddaje to potrzeby, które zaspokoilem, mające zastosowanie *w całym systemie*.

Decyzja, która jest odpowiedzią na tego typu potrzeby, również ma charakter architektoniczny. Architektura systemu przeznaczonego dla dziesięciu jednoczesnych użytkowników będzie się znacząco różniła od architektury systemu obsługującego równocześnie dziesiątki tysięcy użytkowników.

Zależności

Lubię traktować *zależności* jako elementy prowadzące interakcję z systemem, lecz takie, *nad którymi nie mamy kontroli*. Ten brak kontroli może wynikać z faktu, że dany system jest usługą dostarczaną przez zewnętrznego dostawcę. Może to być również platforma, na której działa ten system (np. w postaci kontenera zapewnianego przez środowisko typu Drupal, środowiska uruchomieniowego .NET Common Language Runtime lub systemu operacyjnego). Albo nawet może to być interfejs API oferowany przez zespół z innego działu tej samej firmy. Wszystkie te przypadki stwarzają ten sam problem, czyli niewielkie albo żadne możliwości kontroli nad ich harmonogramem prac i priorytetem zadań. W związku z tym trzeba się za stanowić nad sposobem łączenia zależności i komunikowania się z nimi. Decyzje, które z tego wynikają, mają charakter architektoniczny.

⁸ Przed popularyzacją historyjek użytkownika przez programowanie ekstremalne powszechnie stosowano przypadki użycia. Po raz pierwszy zostały one zaproponowane przez Ivara Jacobsona na konferencji OOP-SLA 1987 i stały się kluczowym elementem notacji diagramów UML oraz procesu RUP (*Rational Unified Process*). Przypadki użycia to bardziej rozbudowane, oparte na dokumentacji sposób zbierania wymagań użytkowników względem oprogramowania.

⁹ Problem polega na tym, że niektóre z tych wymagań można uznać za funkcjonalne (np. niektóre wymagania związane z regulacjami). Sarah Taraporewalla, moja koleżanka z firmy Thoughtworks, ukuła anglojęzyczny termin *cross-functional* (https://sarahtaraporewalla.com/agile/design/decade_of_cross_functional_requirements_cfrs), aby nie tylko uniknąć tego problemu, ale także przeciwdziałać mylnemu przekonaniu ludzi co do tego, że przedrostek *non* („nie”) w słowie *nonfunctional* („niefunkcjonalny”) sugeruje, iż te wymagania są jakoś mniej istotne niż wymagania funkcjonalne. Obecnie oba terminy są stosowane w dużym stopniu wymiennie. FURPS+ to inny termin, popularny szczególnie wśród starszego pokolenia. To także nie jest idealne rozwiązanie (znak + to swego rodzaju unik), ale w tamtym czasie większość ludzi wiedziała, co on oznacza.

Interfejsy

Interfejsy to druga strona zależności, która jest równie ważna, choć w inny sposób. W przeciwieństwie do zależności *interfejs* pojawia się, gdy Twoje oprogramowanie jest wykorzystywane przez inny podmiot. Udostępniając interfejs, stajesz się tym, od kogo inni zależą w realizacji swoich celów. Choć masz pełną kontrolę nad swoim harmonogramem, musisz być tutaj ostrożny. Nawet drobne zmiany w Twoich interfejsach API mogą zakłócić integracje użytkowników, zmuszając ich do dokonania modyfikacji. W efekcie decyzje wpływające na interfejs, który Twój system udostępnia innym, mają charakter architektoniczny.

Relacje między systemami mogą być dwukierunkowe

Czasami Twój i inny system będą wzajemnie zależne. Twój system będzie korzystał z interfejsu systemu, który też używa interfejsu po stronie Twojego systemu. Jeśli brzmi to skomplikowanie i wygląda na koszmar pod względem zarządzania, masz rację, gdyż tak właśnie jest. Wielokierunkowe relacje mogą szybko stać się „piekłem”, ponieważ wszyscy muszą myśleć o swojej architekturze z obu perspektyw.

Techniki budowania

Wreszcie pojawiają się *techniki budowania*, co prawdopodobnie tłumaczy, dlaczego tak bardzo cenię definicję decyzji architektonicznych sformułowaną przez Nygarda. W książce *Release It!* (Pragmatic Bookshelf) był on jednym z pierwszych propagatorów projektowania architektury z myślą o „budowaniu i uruchamianiu”¹⁰, dlatego nie dziwi, że w swojej definicji uwzględnił techniki budowania.

W ciągu ostatniej dekady lub w dłuższym czasie od początków każdej z rewolucji w dziedzinie oprogramowania, które opisałem w rozdziale 1., coraz więcej wniosków zostało uwzględnionych zarówno w narzędziach, jak i technikach tworzenia i publikowania oprogramowania. W konsekwencji zmieniających się metod budowania zmieniły się również same architektury naszych systemów.

Jak to? Pozwól, że wyjaśnię.

Był rok 2005, gdy po raz pierwszy zobaczyłem wpływ rozwiniętych technik budowania. Dołączyłem wtedy do projektu opartego na platformie J2EE¹¹, w którym kod był podzielony

¹⁰ Powszechnie uznaje się, że koncepcję tę spopularyzował Werner Vogels, dyrektor ds. technicznych w firmie Amazon, w wywiadzie udzielonym w 2006 r. (<https://queue.acm.org/detail.cfm?id=1142065>) dla serwisu ACMQueue. Stwierdził on: „Tradycyjny model zakłada, że bierzesz swoje oprogramowanie, przerzucasz je przez mur oddzielający dział projektowy od działu operacyjnego, a następnie o nim zapominasz. W firmie Amazon tak nie jest. Jeśli budujesz oprogramowanie, korzystasz z niego. Podejście to zapewnia projektantom kontakt z codzienną obsługą operacyjną ich oprogramowania, a także z jego użytkownikami. Taka pętla informacji zwrotnej od klientów jest kluczowa dla poprawy jakości usługi”.

¹¹ Projekt był realizowany dwie przecznice od drogi, przy której miała miejsce eksplozja w Buncefield (https://en.wikipedia.org/wiki/Buncefield_fire#Explosion_and_fire), dlatego z różnych powodów jest dobrze zapamiętany.

na moduły w sposób, który dziś nazwano by architekturą „heksagonalną”, „cebuli” lub „portów i adapterów” (https://web.archive.org/web/20071028094131/http://alistair.cockburn.us/index.php/Hexagonal_architecture). Oznaczało to, że istniały wyodrębnione pakiety domenowe, które nie były zależne od standardowych bibliotek platformy Java EE. Dlaczego to zrobiliśmy? Z myślą o możliwości testowania. W tamtych czasach uruchomienie serwera J2EE lokalnie na maszynie projektowej w celu przeprowadzenia zautomatyzowanych testów mogło zająć nawet 25 minut. Większość „elementów inteligencji” znajdowała się w tych pakietach domenowych. Właśnie te elementy chcieliśmy testować najbardziej szczegółowo, a żeby to osiągnąć, potrzebowaliśmy szybko działających testów. Izolując te pakiety, mogliśmy uruchamiać testy kluczowych elementów bez konieczności aktywowania całej powiązanej infrastruktury. I to się sprawdziło. Zajmując się funkcjonalnością głównego komponentu, mogliśmy pracować, korzystając ze stylu projektowania Red Green Refactor opartego na testach z pętlą informacji zwrotnych trwającą 10 – 15 sekund. Szybko i bezpiecznie.

Od 2005 r. inne praktyki programowania ekstremalnego (<http://www.extremeprogramming.org>) stały się standardem (w szczególności ciągle wdrażanie¹²), realizując korzyści wynikające z „nowej fizyki”. Obecnie jesteśmy w stanie niezwykle szybko i często dostarczać kod do środowiska produkcyjnego i udostępniać go klientom. Zalety tego podejścia są dobrze znane. Wspomniałem o nich podczas omawiania pięciu rewolucji w rozdziale 1. Kluczowe jest to, że aby móc ciągle wdrażać oprogramowanie, trzeba stosować określone techniki budowania, ponadto niezbędne jest dokonanie określonych wyborów dotyczących architektury. Na przykład użycie takiego narzędzia jak LaunchDarkly (<https://launchdarkly.com>) do zarządzania przełącznikami wersji nigdy nie będzie opcją widoczną dla użytkowników (można uznać za porażkę sytuację, w której kiedykolwiek zauważą jej obecność), ale ma wpływ na Twój kod.

Stosowanie przełączników opcji, ciągłego wdrażania, a co za tym idzie: potoków i wielu innych rozwiązań, zalicza się do kategorii technik budowania. Związane z tym decyzje mają charakter architektoniczny, ponieważ wpływają na sposób, w jaki kształtujemy nasze oprogramowanie.

Czy powinienem przejmować się umiejscowieniem platform, języków, środowisk i bibliotek?

Choć można dyskutować, czy wybór platformy, języka, środowisk i bibliotek najlepiej pasuje do kryteriów struktury lub technik budowy, są one niewątpliwie istotne, nieważne gdzie je umieścimy w pięciu kryteriach decyzji architektonicznych Nygarda. W większości przypadków można argumentować za obiema opcjami. W niektórych sytuacjach mogą one nawet wchodzić w zakres cech wielofunkcyjnych. Niezależnie od tego, jak zdecydujemy się je sklasyfikować, wybory dotyczące platform, języków, środowisk i bibliotek z pewnością spełniają definicję decyzji architektonicznych.

¹² Aby uzyskać pełne informacje na ten temat, zajrzyj do książki *Continuous Deployment* (O'Reilly, 2024, https://helion.pl/ksiazki/continuous-deployment-valentina-servile,e_4098.htm) Valentiny Servile.

Przykłady decyzji architektonicznych i niearchitektonicznych

Jak wspomniano, decyzje architektoniczne muszą spełniać co najmniej jedno z pięciu kryteriów: powinny wpływać na strukturę, cechy wielofunkcyjne, zależności, interfejsy i/lub techniki budowania. Pora przećwiczyć dokonywanie takich ustaleń, analizując poniższe listy, które zawierają decyzje architektoniczne i niearchitektoniczne.

Decyzje architektoniczne

Struktura

Jak podzielić moduł i jakie będą nowe granice?

Które zespoły będą odpowiedzialne za poszczególne usługi (choć jest to decyzja dotycząca organizacji i personelu, powszechnie uznaje się obecnie, że ma ona bezpośredni wpływ na ostateczną strukturę)?

Cechy wielofunkcyjne

Format, jakiego wszyscy będą używać do zapisywania swoich dzienników, a także miejsce, do którego będą one wysyłane w celu gromadzenia.

Sposób skalowania poziomego (albo nie, gdyż dostępne są też inne formy skalowania) różnych aspektów swojego systemu.

Zależności

Jak dokonać integracji z nowym interfejsem API zewnętrznego dostawcy?

Interfejsy

Jak udostępniać funkcjonalność lub dane za pośrednictwem interfejsu API albo kolejki komunikatów (zarówno innemu zespołowi, z którym współpracujesz, jak i w przyszłości nieznanemu zewnętrznemu dostawcy)?

Techniki budowania

Jakiego środowiska użyć do testowania kontraktów?

Jak korzystać z przełączników wersji?

Zależności, prawdopodobna struktura oraz całkiem możliwe cechy wielofunkcyjne

Aktualizacja do nowej wersji głównej używanego środowiska¹³.

Zależności i najbardziej prawdopodobne cechy wielofunkcyjne

Wybór w celu użycia podstawowego typu wirtualnych zasobów obliczeniowych dostawcy usług chmurowych.

¹³ Zdaję sobie sprawę z tego, że mogłeś mieć zupełnie inne doświadczenia. Wynika to z nieprzestrzegania zasad wersjonowania semantycznego (<https://semver.org>). Na samej górze strony tej specyfikacji stwierdzono, że główne wersje wchodzą w grę wtedy, gdy środowisko wprowadza niezgodne zmiany w interfejsie API.

Decyzje niearchitektoniczne

Cechy, które nie są wielofunkcyjne

Jakie parametry strojenia przekazać procedurze odzyskiwania pamięci używanej maszyny wirtualnej? Można się spodziewać, że będą one zmieniane w trakcie użytkowania systemu. Chociaż można zaprojektować system tak, aby zapewnić dużą elastyczność w tym obszarze, zmiana tych ustawień nie jest decyzją architektoniczną.

Nie ma znaczenia to, czy skalowanie poziomu obejmuje 5, 7 czy 30 instancji części systemu (jest to po prostu efekt uboczny skalowania systemu).

Nie są to interfejsy

Który segment klientów wybierzesz spośród zestawu dostępnych opcji (jest to decyzja biznesowa/produktowa)?

Nie są to techniki budowania

Z jakiego środowiska projektowego IDE powinni skorzystać projektanci (nie powinno to mieć wpływu na architekturę systemu; jest to indywidualna decyzja projektanta)?

Podejmowanie decyzji dotyczącej tego, czy projektanci będą pracować w parach w przypadku danej historyjki użytkownika (jest to decyzja dotycząca praktyki zespołowej; zespół może jednak zdecydować się na częstsze działanie w parach w przypadku historyjek użytkownika o większym znaczeniu architektonicznym).

Jaką konwencję nazewnictwa chcesz zastosować względem folderów i plików (nie powinno to mieć wpływu na strukturę systemu)?

Nie są to zależności ani techniki budowania

Wybór biblioteki do testów jednostkowych, która zostanie użyta (sposób projektowania architektury systemu pod kątem testów jednostkowych prawdopodobnie nie zmieni się, jeśli zmienisz konkretną bibliotekę do tych testów, choć one same niewątpliwie ulegną zmianie).

Wybór narzędzia do testowania wydajności to także kwestia zewnętrzna względem systemu (choć można zaprojektować architekturę w sposób ułatwiający testowanie wydajności, sam wybór oprogramowania do generowania obciążenia nie jest decyzją architektoniczną).

Kto podejmuje te decyzje architektoniczne?

Przed przejściem dalej wróćmy do naszego ćwiczenia wizualizacyjnego. Kogo na początku rozdziału wyobraziłeś sobie jako osobę podejmującą decyzje architektoniczne? Prawdopodobnie w przypadku większości z nich dostrzegłeś architekta lub głównego inżyniera. Jednakże w odniesieniu do innych decyzji, takich jak wybór środowiska do testów kontraktów, tak naprawdę mogła to być osoba z zespołu projektowego (np. inżynier ds. zapewnienia jakości). Zwróć na to uwagę. Decyzje architektoniczne mogą być podejmowane przez każdego, a nie tylko przez osoby z tytułem „architekta” w nazwie stanowiska.

Pora zająć się doprecyzowaniem kryteriów decyzji architektonicznych, koncentrując się na tych *najbardziej istotnych*.

Istotne decyzje architektoniczne

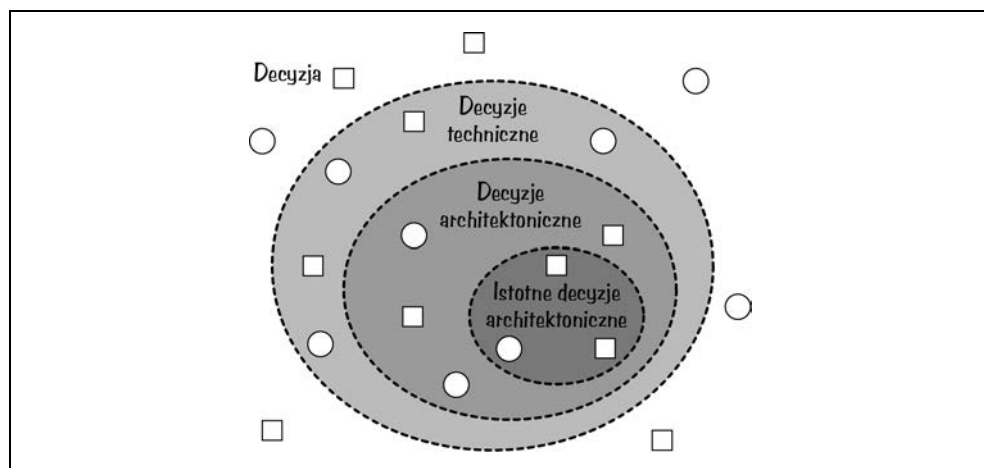
Podczas tworzenia, dostarczania, uruchamiania i rozwijania systemów oprogramowania decyzje architektoniczne podejmowane są nieustannie. Nie wszystkie jednak mają taki sam wpływ.

Choć decyzje podejmowane przez architektów *powinny być* istotne z punktu widzenia architektury, ważne jest zrozumienie, że projektanci również mogą podejmować znaczące decyzje architektoniczne. Oznacza to, że nie można oceniać tego, czy decyzja zasługuje na uwagę, wyłącznie na podstawie tego, kto ją podjął lub ile czasu zajęło jej podjęcie.

Doprecyzujemy pięć kryteriów podejmowania decyzji architektonicznych pozwalających uznać je za *istotne* z punktu widzenia architektury.

Co sprawia, że decyzja architektoniczna jest „istotna”?

Tak jak nie wszystkie decyzje techniczne są architektoniczne, tak nie wszystkie decyzje architektoniczne mają jednakowe znaczenie lub wpływ. Zilustrowano to na rysunku 2.2.



Rysunek 2.2. Nie wszystkie decyzje architektoniczne są „znaczące”

Możesz określić, czy decyzja architektoniczna jest istotna, jeśli w wyróżniający się sposób zawiera bardziej skonkretyzowane wersje wcześniejszych kryteriów, czyli strukturę, cechy wielofunkcyjne, zależności, interfejsy lub techniki budowania. Stosunkowo łatwo jest zidentyfikować znaczącą decyzję architektoniczną na podstawie trzech spośród tych kryteriów (zależności, interfejsy i techniki budowania), ponieważ skala ich wpływu jest przewidywalna. Zacznijmy więc od nich.

Twoja decyzja architektoniczna jest istotna, jeśli uwzględniysz:

- Zupełnie nową zależność lub (w przypadku istniejącej zależności) aktualizację do wersji głównej tego samego dostawcy albo całkowitą zmianę dostawcy.

- Nowy interfejs API w miejsce posiadanego interfejsu lub wprowadzenie w nim dowolnej zmiany, która wymagałaby aktualizacji do wersji głównej z myślą o konsumentach¹⁴.
- Nową technikę budowania w ramach środowiska produkcyjnego (np. wdrożenia kanarowe lub niebiesko-zielone) lub jako etap w potoku.

Trudniej jest zidentyfikować istotną decyzję architektoniczną na podstawie pozostałych dwóch kryteriów (struktura i cechy wielofunkcyjne), ponieważ skala ich wpływu jest trudniejsza do określenia ilościowego. Znaczące decyzje architektoniczne związane ze strukturą dotyczą zwykle albo *umiejscowienia funkcji* (gdzie znajduje się jakaś logika lub odpowiedzialność za coś), albo decyzji o rozpoczęciu (lub zaprzestaniu) stosowania nowego wzorca projektowego lub wzorca architektonicznego wyższego poziomu.

Wzorce projektowe

Mówiąc o wzorcu, mam na myśli powszechnie znane i wielokrotnie używane rozwiązania problemów często spotykanych w strukturze oprogramowania. Termin *wzorec projektowy* (ang. *design pattern*) odnosi się do koncepcji, która zyskała szeroką popularność w środowisku projektantów oprogramowania po wydaniu w 1994 r. książki *Wzorce projektowe* (Helion, <https://helion.pl/ksiazki/wzorce-projektowe-elementy-oprogramowania-objektowego-wielokrotnego-uzytku-erich-gamma-richard-helm-ralph-johnson-john-vli,wzoenv.htm#format/d>) Ericha Gammy, Richarda Helma, Ralpha Johnsona i Johna Vlissidesa. Kolejne prace rozwinęły ten temat, np. seria książek *Pattern-Oriented Software Architecture* (Wiley), książka *Architektura systemów zarządzania przedsiębiorstwem. Wzorce projektowe* (Helion, <https://helion.pl/ksiazki/architektura-systemow-zarzadzania-przedsiębiorstwem-wzorce-projektowe-martin-fowler,szabko.htm#format/d>) Martina Fowlera, książka *Domain-Driven Design. Zapanuj nad złożonym systemem informatycznym* (Helion, <https://helion.pl/ksiazki/domain-driven-design-zapanuj-nad-zlozonym-systemem-informatycznym-eric-evans,dddzap.htm#format/e>) napisana przez Erica Evansa, książka *Enterprise Integration Patterns* (Addison-Wesley) Gregora Hohpe’a oraz książka *Cloud Architecture Patterns* (O’Reilly) Billa Wildera. Wszystkie te książki wprowadziły nie tylko nowe wzorce projektowe, ale także wzorce architektoniczne. Gorąco polecam zapoznanie się z nimi, choć nie ma potrzeby uczenia się ich wszystkich na pamięć. Ja sam z pewnością tego nie zrobiłem.

Decyzja architektoniczna jest zatem istotna (w sensie strukturalnym), jeśli wiąże się z umieszczeniem kluczowej logiki lub odpowiedzialności w konkretnym miejscu albo z rozpoczęciem lub zaprzestaniem stosowania jednego lub więcej uznanych wzorców projektowych opisanych w książkach wymienionych w powyższej ramce.

Na koniec docieramy do najtrudniejszego do zdefiniowania spośród pięciu kryteriów. Mowa o decyzjach wpływających na cechy wielofunkcyjne. Wspomniałem wcześniej, że pojęcie *cech wielofunkcyjnych* obejmuje szeroki i zróżnicowany zbiór niejasnych koncepcji, lecz to nie dlatego trudno jest określić, co sprawia, że dana cecha wielofunkcyjna jest istotna. Trudność w ich

¹⁴ Wersjonowanie semantyczne (<https://semver.org>) pozwala we właściwy sposób ustalać, kiedy coś powinno być traktowane jako zmiana dotycząca głównej wersji.

sformułowaniu wynika z tego, że powiązane z nimi wymagania wielofunkcyjne CFR są kiepsko zdefiniowane. Gdy uda się doprecyzować te wymagania, łatwiej będzie zdefiniować cechy wielofunkcyjne i znacznie prostsze stanie się ustalenie ich znaczenia.

Jak zatem uzyskać przejrzystość definicji w przypadku wymagań CFR? Przewidywane wymagania należy sformułować dokładnie tak samo jak wszystkie inne wymagania, czyli przez wyraźne określenie ich zakresu, wyjaśnienie ich wartości i zdefiniowanie ich kryteriów akceptacji.

Być może nie jest to zaskakujące, ale dobrym sposobem na zrobienie tego jest wykorzystanie standardowej formy historyjki użytkownika. Zaczniemy od wartości:

PRZYKŁAD

„Odgrywając rolę [ROLA]..., oczekuję działania [DZIAŁANIE]..., aby zapewnić wartość [WARTOŚĆ]”

Jako następne występują kryteria akceptacji:

PRZYKŁAD

„Biorąc pod uwagę okoliczności [OKOLICZNOŚCI]..., gdy wystąpi zdarzenie [ZDARZENIE]..., uzyska się wynik [WYNIK]”

Gdy już poświęcisz czas na osiągnięcie tego poziomu szczegółowości, masz realną szansę na stwierdzenie, czy coś jest istotne (zamieszczając znacznie bardziej szczegółowe omówienie w rozdziale 8., powrócę w nim do tego sposobu uzyskiwania wymagań funkcjonalnych CFR, ale na razie tyle wystarczy).

Decyzja architektoniczna jest istotna, jeśli zmienia naszą zdolność do spełnienia co najmniej jednego wymagania CFR. Przyjmijmy następujące wymaganie CFR:

PRZYKŁAD

Jako klient
Chcę otrzymywać wyniki wyszukiwania w ciągu 500 milisekund
Aby móc szybko znaleźć to, czego szukam
Mając na uwadze to, że witryny dotyczą normalne poziomy obciążenia żądaniami wyszukiwania
Po wysłaniu przez klienta żądania
System zwraca w 99% przypadków poprawne wyniki wyszukiwania w czasie krótszym niż 500 milisekund

Jeśli podejmuję decyzję, która może zagrozić spełnieniu kryteriów akceptacji, jest ona istotna.

Podsumowując: decyzję architektoniczną można uznać za istotną, gdy spełnia ona co najmniej jedno z pięciu kryteriów decyzji architektonicznej.

Czego nie należy brać pod uwagę przy ocenie znaczenia z punktu widzenia architektury?

Pora spojrzeć na drugą stronę medalu: czego *nie* powinniśmy brać pod uwagę przy ocenie znaczenia z punktu widzenia architektury?

Porównajmy ze sobą dwa przykłady.

Po pierwsze, wyobraź sobie sytuację, w której projektant pracuje nad kodem i zauważa, że jedna z zależności jest nieaktualna, ale dotyczy to jedynie braku wersji zawierającej poprawkę błędu. Decyduje się ją zaktualizować, uruchamiając skrypt kompilacji. Nie zdaje sobie jednak sprawy, że ta operacja przypadkowo powoduje jednoczesne zaktualizowanie wielu innych zależności przechodnich. W rezultacie jedna z tych zależności zmienia swoją licencję oprogramowania *open source* z licencji MIT (którą dział bezpieczeństwa informacji akceptuje) na licencję GPL 3.0 (nie jest ona akceptowana ze względu na jej tzw. wirusową naturę)¹⁵.

Po drugie, wyobraź sobie architekta, który przeprowadza długi, szczegółowy i technicznie skomplikowany proces analizy decyzji o przeniesieniu kluczowej części systemu z relacyjnej bazy danych do bazy dokumentów (konkretnie z bazy MySQL do bazy MongoDB). W trakcie tego procesu angażuje wielu ekspertów, przeprowadza pewne testy i prowadzi szeroko zakrojone badania, a nawet organizuje pojedynek „wypieków”¹⁶ między dostawcami obu rozwiązań, aby sprawdzić, jak ich produkty radzą sobie przy realistycznych wolumenach danych i profilach obciążenia.

Jakie wnioski można wyciągnąć z tych dwóch przykładów? Pojawiają się trzy wnioski.

Pierwszy wniosek, do którego nawiązywałem przez cały czas, jest taki, że każdy może podejmować decyzje dotyczące czegoś o znaczeniu architektonicznym. Choć najczęściej robią to architekci, nie są oni jedynym źródłem decyzji. Projektanci także mogą je podejmować. Nie można określić, czy decyzja jest istotna, bazując jedynie na tym, kto był w nią zaangażowany. W związku z tym oba te przykłady mają znaczenie architektoniczne niezależnie od tego, kto brał w nich udział (motyw wskazujący, że każdy może podejmować decyzje, jest kluczowy dla tej książki i będzie pojawiał się wielokrotnie).

Drugi wniosek jest taki, że czas poświęcony na proces decyzyjny nie ma znaczenia. Długotrwały proces wyboru nowej bazy danych, która ma zastąpić istniejącą bazę MySQL, jest nieistotny. Bardzo możliwe, że w innych okolicznościach ta sama decyzja mogłaby zostać podjęta w ciągu kilku sekund przez dyrektora ds. technologii lub głównego architekta, którzy opieraliby się wyłącznie na swoich osobistych uprzedzeniach lub preferencjach. Podobnie natychmiastowość „decyzji” o użyciu zależności objętej licencją GPL nie byłaby ani mniej, ani bardziej

¹⁵ W tym przypadku unikanie licencji GPL 3.0, a nawet wszystkich „wirusowych” licencji *open source*, powinno być ujęte jako wymaganie CFR i w idealnej sytuacji testowane w ramach potoku.

¹⁶ „Wypieki” (ang. *bake-off*) to potoczne określenie konkursu pomiędzy firmami o zdobycie kontraktu. Termin ten stał się znany dzięki brytyjskiemu programowi telewizyjnemu *The Great British Bake-Off* (znanemu w Ameryce Północnej jako *The Great British Baking Show*, co bywa mylące).

znacząca, gdyby poddano ją kilkudniowej analizie. Oba te przykłady są istotne z punktu widzenia architektury niezależnie od tego, ile czasu zajęło podjęcie decyzji.

Trzeci wniosek mówi o tym, że decyzja nie musi być celowa, aby miała znaczący wpływ na architekturę. Przypomnijmy sobie ponownie przypadkowe dodanie zależności z licencją GPL. Zmiana ta nie była planowana. Łatwo mogłaby umknąć uwadze, ponieważ nic by nas o niej nie poinformowało. Kompilacja nadal by działała, a kod wykonywałby swoje zadanie.

Wszystkie te elementy są nieistotne w kontekście znaczenia decyzji. Oba przykłady są architektonicznie znaczące niezależnie od tego, czy osoby odpowiedzialne za decyzje były świadome ich wagi.

Znaczenie architektury w kontekście wdrożonego oprogramowania

Dysponujemy zatem kryteriami istotności, czyli kluczowymi decyzjami dotyczącymi struktury, cech wielofunkcyjnych, zależności, interfejsów i technik budowania. Wiadomo też dobrze, jakie czynniki są nieistotne: kto podejmuje decyzje, jak długo to trwa i czy proces był świadomy, czy nie. Ważne jest jednak, aby zwrócić uwagę na jeszcze jeden aspekt, który kryje się za tym wszystkim. W dzisiejszym zrewolucjonizowanym świecie decyzje są istotne tylko wtedy, gdy weźmiemy pod uwagę ich związek z wdrożonym oprogramowaniem: albo w odniesieniu do ścieżki prowadzącej system do środowiska produkcyjnego, albo jego cech, gdy już tam trafi.

Może się to wydać arbitralne, dlatego pozwól, że wyjaśnię. Przypomnij sobie przykład projektanta dodającego nową zależność objętą licencją GPL, która trafiła do środowiska produkcyjnego. Wyobraź sobie teraz, że zanim projektant zaktualizował zależności, dodał do potoku (odpowiedzialnego za kompilację, weryfikację poprawności i wdrażanie) skaner zależności. Skaner wykryłby naruszenie licencji i spowodowałby niepowodzenie kompilacji. Zależność przechodnia nigdy nie trafiłaby do środowiska produkcyjnego. Ta świadoma i istotna decyzja projektanta sprawiła, że „przypadkowa” decyzja nie była znacząca. Z kolei istotna z punktu widzenia architektury decyzja o dodaniu skanera i skonfigurowaniu go w celu wykrywania zależności przechodnich z licencją GPL była znacząca.

Co jednak będzie, jeśli jeszcze niczego nie dostarczono do środowiska produkcyjnego? W takiej sytuacji powinno się zwrócić uwagę na decyzje, które aktywnie utrudniają wprowadzenie Twojego oprogramowania do tego środowiska. Te decyzje są znaczące w sensie negatywnym.

Piszę o tym, ponieważ dopóki Twoje decyzje architektoniczne nie znajdują odzwierciedlenia we wdrożonym oprogramowaniu, nie są to jeszcze prawdziwe decyzje, a jedynie *aspiracje*, a kod to tylko zbiór schematów. W takich okolicznościach poważnie zalecam jak najszybsze wprowadzenie swojej architektury do środowiska produkcyjnego. Zawsze istnieje sposób, by to zrobić, poza *wyjątkowo trudnymi okolicznościami*, a 99,99% z nas nie staje przed takimi wyzwaniem (zachęcam do ciągłego zadawania sobie następującego pytania: „Jak można to wdrożyć już teraz?” zamiast pytania: „Dlaczego nie mogę tego wdrożyć?”). Jeśli potrzebujesz nowych lub innych decyzji, by to osiągnąć, tak właśnie postąp. Na tym etapie możesz zdać sobie sprawę, że niektóre z Twoich kluczowych decyzji architektonicznych faktycznie stoją na przeszkodzie. Jeżeli nie spróbujesz uzyskać dostępu do środowiska produkcyjnego, nie będziesz miał żadnych możliwości poznania tego negatywnego aspektu.

Czy przypominasz sobie sytuację, o której wcześniej wspomniałem: zespół projektowy zastosował techniki projektowania opartego na domenie (DDD — *Domain-Driven Design*) do określenia struktury kodu w taki sposób, że większość można było testować lokalnie bez uruchamiania serwera J2EE? Korzyść ta objęła również potoki. Gdyby podjęto inną decyzję, a testowanie zarówno lokalnie, jak i w potoku byłoby utrudnione, uznano by to za znaczące (znacząco złe, ale wciąż znaczące). Późniejsza decyzja projektantów o wykorzystaniu architektury heksagonalnej umożliwiającej lokalne testowanie, które było również możliwe do przeprowadzenia w potokach, okazała się kluczowa dla wdrożenia kodu w środowisku produkcyjnym.

Przed umieszczeniem kodu w tym środowisku oczekiwane korzyści z obecności serwera J2EE w środowisku uruchomieniowym pozostawały jedynie w sferze nadziei i nie były jeszcze zrealizowane. Jednocześnie ich negatywny wpływ na etapie kompilacji był łagodzony.

Dlatego uważam, że kryteria technik budowania są równie ważne jak wszystkie pozostałe kryteria znaczących decyzji architektonicznych. Techniki te zapewniają, że temu ważnemu aspektowi poświęca się tyle samo uwagi, co pozostałym czterem kryteriom. Służy to zrównoważeniu, przynajmniej częściowo, tendencji innych kryteriów do pomijania środowiska wykonawczego.

Decyzje dotyczące produktów i przydziału zadań zespołom

Aby utrzymać koncentrację w tym rozdziale, celowo pominąłem następujące dwa źródła wpływu na decyzje architektoniczne:

- Zarządzanie produktem/działalność biznesowa.
- Określenie, kto decyduje o przydziale zespołów do zadań.

Nie ulega wątpliwości, że decyzje dotyczące produktu mogą bezpośrednio wpływać na architekturę oprogramowania. To samo można powiedzieć o decyzjach związanych z personelem lub organizacją pracy zespołów.

Aby uniknąć nieporozumień, traktuj te decyzje wpływające na architekturę oprogramowania *tak, jakby były decyzjami architektonicznymi*. Podchodź do nich w taki sam sposób jak do wszystkich innych decyzji architektonicznych. Metody ich ułatwienia, które przedstawię, są w pełni możliwe do zastosowania. Menedżer produktu może podejmować decyzje mające wpływ na architekturę, ale w idealnym świecie będzie świadomy tego faktu i zaangażuje odpowiednie osoby, gdy to nastąpi.

Przykłady istotnych decyzji architektonicznych

Istotne decyzje architektoniczne mają duży wpływ, ponieważ spełniają co najmniej jedno z pięciu rozszerzonych kryteriów: decydują o strukturze, cechach wielofunkcyjnych, zależnościach, interfejsach i/lub technikach budowania. Znajdują one też odzwierciedlenie w kodzie.

Na zakończenie przyjrzyjmy się kilku dodatkowym przykładom decyzji. Tym razem wszystkie dotyczą architektury, ale tylko te z pierwszej listy są istotne, natomiast decyzje z drugiej listy nie mają takiego znaczenia. Warto zastanowić się nad decyzjami, w których ostatnio brałeś udział. Zadaj sobie pytanie: czy były to decyzje architektoniczne, czy jedynie techniczne? A jeśli architektoniczne, to czy były znaczące?

Istotne decyzje architektoniczne

Struktura

Projektanci w zespole decydują się na refaktoryzację kodu w celu wyodrębnienia mikro-frontendu.

Cechy wielofunkcyjne

Projektant infrastruktury dodaje bramę API przed usługami publicznymi. Nie jest włączony żaden mechanizm uwierzytelniania, ograniczanie liczby żądań ani inne opcje interwencji. Na razie jest to proste przekazywanie danych.

Zależności i prawdopodobne cechy wielofunkcyjne

Komitet ds. architektury podejmuje decyzję o zmianie podstawowego typu wirtualnych zasobów obliczeniowych dostawcy usług chmurowych, z których firma będzie korzystać.

Interfejsy

Architekt aktualizuje interfejs API REST, aby usunąć opcjonalny parametr, który mógł być wykorzystywany przez niektórych użytkowników.

Techniki budowania

Osoba zapewniająca jakość oprogramowania decyduje, jakie będzie używane środowisko do testowania kontraktów.

Decyzje nieistotne z punktu widzenia architektury

Nieistotna struktura

Projektant dokonuje refaktoryzacji klasy, aby wyodrębnić metodę prywatną.

Nieistotne cechy wielofunkcyjne

Projektant przeprowadza refaktoryzację kodu bez zmiany jego struktury, co eliminuje wąskie gardło i pozwala na podwojenie przepustowości (zauważ, że gdyby podjęto odwrotną decyzję, która w konsekwencji spowodowałaby brak możliwości spełnienia wymagania CFR, sytuacja zostałaby uznana za istotną).

Nieistotne zależności

Projektant porządkuje skrypt kompilacji, usuwając szereg zależności bez zmiany funkcjonalności systemu.

Dokonujesz aktualizacji do nowej wersji dodatkowej używanego już środowiska (jeśli faktycznie jest to zmiana dotycząca tej wersji, powinna być ona niezauważalna dla całego kodu, który z niej korzysta).

Nieistotne interfejsy

Architekt aktualizuje stronę *wiki*, usuwając opcjonalny parametr z interfejsu API po uprzednim sprawdzeniu, że żaden z jego użytkowników z niego nie korzysta.

Nieistotne techniki budowania

Projektant zajmujący się infrastrukturą wdraża portal dla projektantów, który umożliwia wszystkim zespołom udostępnianie swoich usług i interfejsów API.

Nie wszystkie decyzje mające wpływ na architekturę są na początku decyzjami „architektonicznymi”

Zaznajamiając się z przykładami decyzji w tym rozdziale, być może nie zgodziłeś się z moim sposobem ich kategoryzacji. Warto wiedzieć, że (1) w pełni uznaję wpływ decyzji nietechnicznych na architekturę oraz (2) zdecydowanie zgadzam się z tym, że niektóre decyzje techniczne, choć niekoniecznie o znaczeniu architektonicznym, powinny przejść przez pewnego rodzaju formalny (lub przynajmniej świadomy) proces decyzyjny.

Podsumowanie

Dlaczego poświęciłem cały rozdział na podkreślenie kwestii znaczenia architektury? Po pierwsze, skuteczne praktykowanie architektury w świecie po rewolucjach oznacza, że podejmuje się znacznie więcej decyzji. W obliczu tego wszystkiego ważne jest rozpoznanie, które decyzje są architektoniczne, a także które z nich są znaczące. Jest to absolutnie niezbędne do skoncentrowania się na kwestii czasu, nakładu pracy i zasobów.

Po drugie, ważne jest, aby docenić wpływ projektantów na architekturę działającego systemu, niezależnie czy są tego świadomi, czy nie. Wbrew wszelkim kryteriom istotności projektanci mogą (i faktycznie to robią) podejmować decyzje istotne z punktu widzenia architektury, co należy postrzegać w pozytywny sposób.

Jeśli nasze architektury składają się wyłącznie z decyzji, trzeba wiedzieć, na które z nich zwrócić szczególną uwagę. Podkreślałem również, że choć obecność decyzji we wdrożonym kodzie nie stanowi ścisłego kryterium znaczenia architektonicznego, decyzje utrudniające wczesne i częste wdrażanie są istotne w *negatywnym sensie*. Do momentu zainicjowania danej decyzji przypuszczamy jedynie, czy jest ona przydatna. Przestań zgadywać, zacznij wdrażać i wyciągaj wnioski z informacji zwrotnych.

Zanim przejdziesz dalej, przypomnij sobie ćwiczenie wizualizacyjne, które wykonałeś na początku rozdziału. Gdy zamknąłeś oczy i wyobraziłeś sobie, kto podejmuje decyzje architektoniczne (istotne lub nie) w Twojej firmie, czy ujrzałeś jedną osobę, czyli architekta? A może wyobraziłeś sobie grupę ludzi?

W rozdziale 3. zostanie rozwinięte to podejście i udzielona odpowiedź na następujące dwa pytania: „Jakie są standardowe metody podejmowania decyzji na dużą skalę?” oraz „Jakie są cechy »idealnego« zdecentralizowanego procesu decyzyjnego opartego na informacji zwrotnej?”.

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

Oto książka pełna praktycznej mądrości, trafiająca w sedno tego, czym jest architektura.

Grady Booch, IBM Fellow

Rola architekta oprogramowania się zmienia. W miarę jak systemy stają się coraz bardziej złożone, tradycyjny model działania architekta przestaje wystarczać. Zadań jest zbyt wiele, a ich zaniebdanie w końcu doprowadzi do punktu krytycznego. Kluczowe staje się współdziałanie architekta z zespołem projektowym — tylko wtedy możliwe jest tworzenie trwałej, elastycznej i efektywnej architektury.

Tę książkę doceni każdy, kto praktykuje architekturę w ramach i na rzecz zespołu. Dzięki niej zrozumiesz wady scentralizowanych praktyk architektury w rozproszonym świecie. Poznasz też kluczowe aspekty w zdecentralizowanego podejścia do architektury opartego na informacji zwrotnej i zasady jego wdrażania takiego podejścia. Znajdziesz tu omówienie czynników wpływających na efektywność decyzji architektonicznych, a także ich społeczny wymiar — dobra architektura bowiem to nie tylko struktura kodu, ale również jakość współpracy. Dzięki opisanym metodom rozwiniesz sposób myślenia, który pozwala każdemu w zespole praktykować architekturę i budować lepsze systemy.

Najlepsza architektura oprogramowania ewoluuje dzięki zaangażowaniu wszystkich.

Martin Fowler, główny badacz w Thoughtworks

Najważniejsze zagadnienia:

- jak się zmienia nowoczesny proces dostarczania oprogramowania
- metodologia łączenia architektury oprogramowania z jego rozwojem
- współzależność decyzji, architektury i informacji zwrotnej z działających systemów
- wprowadzanie praktyk maksymalizacji korzyści i minimalizacji ryzyka
- dostrajanie podejścia do architektury, umiejętności osób w zespole i kultury firmy

Andrew Harmel-Law jest głównym specjalistą technicznym w firmie Thoughtworks. Zajmuje się projektowaniem opartym na domenach, modernizacją na dużą skalę i transformacjami organizacyjnymi. Pasjonuje się oprogramowaniem *open source*, dzieli się swoją wiedzą w ramach szkoleń online, artykułów, publicznych prelekcji i mentoringu.

**helion.pl**

**HELION S.A.**
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!



ISBN 978-83-289-2750-6



Cena: 119,00 zł