



Architektura oprogramowania w praktyce

WYDANIE IV

Len Bass

Paul Clements

Rick Kazman



Helion 

Tytuł oryginału: Software Architecture in Practice, 4th Edition

Tłumaczenie: Tomasz Walczak

ISBN: 978-83-283-9052-2

Authorized translation from the English language edition, entitled Software Architecture in Practice, 1st Edition by Len Bass; Paul Clements; Rick Kazman, published by Pearson Education, Inc, publishing as Addison Wesley Professional, Copyright © 2022 Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc. Polish language edition published by Helion S.A., Copyright © 2022.

Figure 1.1: GraphicsRF.com/Shutterstock

Figure 15.2: Shutterstock Vector/Shutterstock

Figure 17.1: Oleksiy Mark/Shutterstock

Figure 17.2, cloud icon: luckyguy/123RF

Figures 17.2, 17.4, and 17.5 computer icons: Dacian G/Shutterstock

CMM. CMMI, Capability Maturity Model. Capability Maturity Modeling. Carnegie Mellon, CERT, and CERT Coordination Center are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

ATAM; Architecture Tradeoff Analysis Method; CMM Integration; COTS Usage-Risk Evaluation; CURE; EPIC; Evolutionary Process for Integrating COTS Based Systems; Framework for Software Product Line Practice; IDEAL; Interim Profile; OAR; OCTAVE; Operationally Critical Threat, Asset, and Vulnerability Evaluation; Options Analysis for Reengineering; Personal Software Process; PLTP; Product Line Technical Probe; PSP; SCAMPI; SCAMPI Lead Appraiser; SCAMPI Lead Assessor; SCE; SEI; SEPG; Team Software Process; and TSP are service marks of Carnegie Mellon University.

Special permission to reproduce portions of works copyright by Carnegie Mellon University, as listed on page 509, is granted by the Software Engineering Institute.

Polish edition copyright © 2022 by Helion S.A.

All rights reserved.

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

<https://helion.pl/user/opinie/aropw4>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.

ul. Kościuszki 1c, 44-100 Gliwice

tel. 32 231 22 19, 32 230 98 63

e-mail: helion@helion.pl

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

Printed in Poland.

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

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

Spis treści

Przedmowa	11
Podziękowania	14
CZĘŚĆ I. WPROWADZENIE	15
ROZDZIAŁ 1. Czym jest architektura oprogramowania?	17
1.1. Czym jest, a czym nie jest architektura oprogramowania	18
1.2. Struktury i perspektywy architektury	22
1.3. Co sprawia, że architektura jest „dobra”?	36
1.4. Podsumowanie	39
1.5. Literatura	40
1.6. Pytania do dyskusji	41
ROZDZIAŁ 2. Dlaczego architektura oprogramowania jest istotna?	43
2.1. Utrudnianie albo ułatwianie uzyskania głównych atrybutów jakościowych systemu	44
2.2. Wnioskowanie na temat zmian i zarządzanie nimi	45
2.3. Prognozowanie atrybutów jakościowych systemu	46
2.4. Komunikacja między interesariuszami	47
2.5. Wczesne decyzje projektowe	50
2.6. Ograniczenia dotyczące implementacji	51
2.7. Wpływ na strukturę organizacji	52
2.8. Umożliwianie stopniowego rozwijania systemu	53
2.9. Szacowanie kosztów i harmonogramu prac	53
2.10. Uniwersalny model wielokrotnego użytku	54
2.11. Architektura umożliwia dodawanie niezależnie rozwijanych elementów	55
2.12. Ograniczanie listy możliwości w projekcie	56
2.13. Podstawa do szkoleń	57

2.14. Podsumowanie	57
2.15. Literatura	58
2.16. Pytania do dyskusji	58

CZĘŚĆ II. ATRYBUTY JAKOŚCIOWE 61

ROZDZIAŁ 3. Czym są atrybuty jakościowe 63

3.1. Funkcjonalność	64
3.2. Rozważania o atrybutach jakościowych	66
3.3. Tworzenie specyfikacji wymagań dotyczących atrybutów jakościowych: scenariusze związane z atrybutami jakościowymi	67
3.4. Osiąganie atrybutów jakościowych za pomocą wzorców i taktyk architektonicznych	71
3.5. Projektowanie z użyciem taktyk	72
3.6. Analizowanie decyzji projektowych wpływających na atrybuty jakościowe: kwestionariusze oparte na taktykach	74
3.7. Podsumowanie	75
3.8. Literatura	76
3.9. Pytania do dyskusji	76

ROZDZIAŁ 4. Dostępność 78

4.1. Ogólny scenariusz dotyczący dostępności	81
4.2. Taktyki zapewniania dostępności	82
4.3. Oparty na taktykach kwestionariusz dostępności	92
4.4. Wzorce dostępności	95
4.5. Literatura	98
4.6. Pytania do dyskusji	100

ROZDZIAŁ 5. Łatwość wdrażania 101

5.1. Ciągłe wdrażanie	102
5.2. Łatwość wdrażania	106
5.3. Ogólny scenariusz dotyczący łatwości wdrażania	107
5.4. Taktyki zapewniania łatwości wdrażania	109
5.5. Oparty na taktykach kwestionariusz łatwości wdrażania	112
5.6. Wzorce łatwości wdrażania	113
5.7. Literatura	120
5.8. Pytania do dyskusji	120

ROZDZIAŁ 6.	Efektywność energetyczna	121
	6.1. Ogólny scenariusz dotyczący efektywności energetycznej	123
	6.2. Taktyki zapewniania efektywności energetycznej	125
	6.3. Oparty na taktykach kwestionariusz efektywności energetycznej	128
	6.4. Wzorce	130
	6.5. Literatura	132
	6.6. Pytania do dyskusji	133
ROZDZIAŁ 7.	Integrowalność	134
	7.1. Ocena integrowalności architektury	135
	7.2. Ogólny scenariusz dotyczący integrowalności	137
	7.3. Taktyki zapewniania integrowalności	139
	7.4. Oparty na taktykach kwestionariusz integrowalności	146
	7.5. Wzorce	147
	7.6. Literatura	150
	7.7. Pytania do dyskusji	151
ROZDZIAŁ 8.	Modyfikowalność	152
	8.1. Ogólny scenariusz dotyczący modyfikowalności	156
	8.2. Taktyki zapewniania modyfikowalności	157
	8.3. Oparty na taktykach kwestionariusz modyfikowalności	161
	8.4. Wzorce	163
	8.5. Literatura	168
	8.6. Pytania do dyskusji	169
ROZDZIAŁ 9.	Wydajność	171
	9.1. Ogólny scenariusz dotyczący wydajności	173
	9.2. Taktyki zapewniania wydajności	176
	9.3. Oparty na taktykach kwestionariusz wydajności	186
	9.4. Wzorce dotyczące wydajności	187
	9.5. Literatura	191
	9.6. Pytania do dyskusji	192
ROZDZIAŁ 10.	Bezpieczeństwo	194
	10.1. Ogólny scenariusz dotyczący bezpieczeństwa	197
	10.2. Taktyki zapewniania bezpieczeństwa	199
	10.3. Oparty na taktykach kwestionariusz na temat bezpieczeństwa	205

10.4.	Wzorce dotyczące bezpieczeństwa	208
10.5.	Literatura	211
10.6.	Pytania do dyskusji	212
ROZDZIAŁ 11.	Zabezpieczenia	215
11.1.	Ogólny scenariusz dotyczący zabezpieczeń	217
11.2.	Taktyki zabezpieczania systemu	219
11.3.	Oparty na taktykach kwestionariusz na temat zabezpieczeń	224
11.4.	Wzorce zabezpieczania systemu	226
11.5.	Literatura	228
11.6.	Pytania do dyskusji	229
ROZDZIAŁ 12.	Testowalność	230
12.1.	Ogólny scenariusz dotyczący testowalności	233
12.2.	Taktyki zapewniania testowalności	235
12.3.	Oparty na taktykach kwestionariusz zapewniania testowalności	241
12.4.	Wzorce dotyczące testowalności	241
12.5.	Literatura	244
12.6.	Pytania do dyskusji	244
ROZDZIAŁ 13.	Użyteczność	246
13.1.	Ogólny scenariusz dotyczący użyteczności	247
13.2.	Taktyki zapewniania użyteczności	248
13.3.	Oparty na taktykach kwestionariusz zapewniania użyteczności	251
13.4.	Wzorce użyteczności	253
13.5.	Literatura	256
13.6.	Pytania do dyskusji	256
ROZDZIAŁ 14.	Praca nad innymi atrybutami jakościowymi	257
14.1.	Inne rodzaje atrybutów jakościowych	257
14.2.	Stosowanie (lub nie) standardowych list atrybutów jakościowych	260
14.3.	Jak radzić sobie z „x-alnością”? Dodawanie nowego AJ do zestawu	264
14.4.	Literatura	266
14.5.	Pytania do dyskusji	267

CZĘŚĆ III. ROZWIĄZANIA ARCHITEKTONICZNE	269
ROZDZIAŁ 15. Interfejsy oprogramowania	271
15.1. Zagadnienia związane z interfejsami	272
15.2. Projektowanie interfejsu	275
15.3. Dokumentowanie interfejsu	285
15.4. Podsumowanie	286
15.5. Literatura	287
15.6. Pytania do dyskusji	287
ROZDZIAŁ 16. Wirtualizacja	289
16.1. Współużytkowane zasoby	290
16.2. Maszyny wirtualne	291
16.3. Obrazy maszyn wirtualnych	295
16.4. Kontenery	296
16.5. Kontenery i maszyny wirtualne	299
16.6. Przenośność kontenerów	300
16.7. Pody	300
16.8. Architektura bezserwerowa	301
16.9. Podsumowanie	303
16.10. Literatura	303
16.11. Pytania do dyskusji	304
ROZDZIAŁ 17. Chmura i przetwarzanie rozproszone	305
17.1. Podstawowe informacje o chmurze	306
17.2. Awarie w chmurze	309
17.3. Używanie wielu instancji do zwiększania wydajności i dostępności	312
17.4. Podsumowanie	322
17.5. Literatura	323
17.6. Pytania do dyskusji	324
ROZDZIAŁ 18. Systemy mobilne	325
18.1. Energia	326
18.2. Łączność sieciowa	328
18.3. Czujniki i aktuatory	330
18.4. Zasoby	332
18.5. Cykl życia	335

18.6. Podsumowanie	339
18.7. Literatura	340
18.8. Pytania do dyskusji	340

CZĘŚĆ IV. PRAKTYKI ZAPEWNIANIA SKALOWALNOŚCI ARCHITEKTURY 343

ROZDZIAŁ 19. Wymagania o znaczeniu architektonicznym	345
19.1. Zbieranie wymagań o znaczeniu architektonicznym na podstawie dokumentacji wymagań	346
19.2. Zbieranie wymagań o znaczeniu architektonicznym na podstawie rozmów z interesariuszami	348
19.3. Zbieranie wymagań o znaczeniu architektonicznym na podstawie celów biznesowych	351
19.4. Zapisywanie wymagań o znaczeniu architektonicznym w drzewie użyteczności	354
19.5. Zmiany się zdarzają	357
19.6. Podsumowanie	357
19.7. Literatura	358
19.8. Pytania do dyskusji	359
ROZDZIAŁ 20. Projektowanie architektury	360
20.1. Projektowanie oparte na atrybutach	360
20.2. Kroki w podejściu ADD	362
20.3. Jeszcze o podejściu ADD – krok 4., wybór koncepcji projektowych, aby spełnić ustalone czynniki	367
20.4. Jeszcze o podejściu ADD – krok 5., tworzenie struktur	371
20.5. Jeszcze o podejściu ADD – krok 6., tworzenie wstępnej dokumentacji w trakcie projektowania	375
20.6. Jeszcze o podejściu ADD – krok 7., przeprowadzanie analiz obecnego projektu i ocena realizacji celów iteracji oraz celu projektu	377
20.7. Podsumowanie	379
20.8. Literatura	380
20.9. Pytania do dyskusji	381
ROZDZIAŁ 21. Ewaluacja architektury	382
21.1. Ewaluacja jako aktywność zmniejszająca ryzyko	383
21.2. Główne aktywności związane z ewaluacją	383
21.3. Kto może dokonywać ewaluacji?	385

21.4.	Czynniki kontekstowe	386
21.5.	Metoda analizy kompromisów architektonicznych (ATAM)	387
21.6.	Metoda LAE	401
21.7.	Podsumowanie	404
21.8.	Literatura	404
21.9.	Pytania do dyskusji	405
ROZDZIAŁ 22.	Dokumentowanie architektury	406
22.1.	Zastosowania i odbiorcy dokumentacji architektury	407
22.2.	Notacje	408
22.3.	Perspektywy	409
22.4.	Łączenie perspektyw	418
22.5.	Dokumentowanie zachowań	420
22.6.	Poza perspektywy	424
22.7.	Dokumentowanie uzasadnienia	427
22.8.	Interesariusze architektury	428
22.9.	Uwagi praktyczne	432
22.10.	Podsumowanie	435
22.11.	Literatura	436
22.12.	Pytania do dyskusji	437
ROZDZIAŁ 23.	Zarządzanie długiem architektonicznym	438
23.1.	Ustalanie, czy występuje problem z długiem architektonicznym	439
23.2.	Wykrywanie hotspotów	442
23.3.	Przykład	445
23.4.	Automatyzacja	447
23.5.	Podsumowanie	448
23.6.	Literatura	448
23.7.	Pytania do dyskusji	449
CZĘŚĆ V.	ARCHITEKTURA I ORGANIZACJA	451
ROZDZIAŁ 24.	Rola architektów w projektach	453
24.1.	Architekt a menedżer projektu	453
24.2.	Przyrostowo rozwijana architektura i jej interesariusze	455
24.3.	Architektura i podejście zwinne	456
24.4.	Architektura a rozproszony rozwój oprogramowania	461

24.5. Podsumowanie	464
24.6. Literatura	464
24.7. Pytania do dyskusji	465
ROZDZIAŁ 25. Kompetencje architektoniczne	466
25.1. Kompetencje jednostek: obowiązki, umiejętności i wiedza architektów	467
25.2. Kompetencje organizacji rozwijającej architekturę oprogramowania	473
25.3. Jak stać się lepszym architektem?	475
25.4. Podsumowanie	476
25.5. Literatura	476
25.6. Pytania do dyskusji	477
CZĘŚĆ VI. WNIOSKI	481
ROZDZIAŁ 26. Rzut oka w przyszłość: przetwarzanie kwantowe	481
26.1. Jeden kubit	482
26.2. Teleportacja kwantowa	484
26.3. Przetwarzanie kwantowe i szyfrowanie	485
26.4. Inne algorytmy	486
26.5. Potencjalne zastosowania	488
26.6. Uwagi końcowe	488
26.7. Literatura	489
O autorach	491
Literatura	493
Skorowidz	511

2

Dlaczego architektura oprogramowania jest istotna?

Ach, budować, budować!
To najszlachetniejsza ze wszystkich sztuk.
— Henry Wadsworth Longfellow

Skoro architektura jest odpowiedzią, jak brzmi pytanie?

W tym rozdziale koncentrujemy się na tym, dlaczego architektura jest istotna z technicznego punktu widzenia. Przyjrzymy się 13 najważniejszym powodom. Możesz je wykorzystać do uzasadnienia prac nad nową architekturą lub do analizy i modyfikowania istniejącej architektury systemu.

1. Architektura może albo utrudniać, albo ułatwiać uzyskanie głównych atrybutów jakościowych systemu.
2. Decyzje podjęte w trakcie tworzenia architektury umożliwiają wnioskowanie na temat zmian i zarządzanie nimi w trakcie ewoluowania systemu.
3. Analiza architektury umożliwia wczesne oszacowanie atrybutów jakościowych systemu.
4. Udokumentowana architektura ułatwia komunikację między interesariuszami.
5. Architektura jest nośnikiem najwcześniejszych, a tym samym najbardziej fundamentalnych i najtrudniejszych do zmiany decyzji projektowych.
6. Architektura określa zestaw ograniczeń dotyczących późniejszej implementacji.
7. Architektura wyznacza strukturę organizacji lub na odwrót.
8. Architektura może być podstawą do stopniowego rozwijania oprogramowania.
9. Architektura jest najważniejszym artefaktem, który umożliwia architektowi i menedżerowi projektu wnioskowanie na temat kosztów i harmonogramu prac.
10. Architekturę można stworzyć w formie uniwersalnego modelu wielokrotnego użytku, który stanowi istotę linii produktowej.

11. Programowanie oparte na architekturze wymaga skupienia się na łączeniu komponentów, a nie tylko na ich tworzeniu.
12. Przez ograniczenie alternatywnych projektów architektura kierkuje kreatywność programistów, ponieważ zmniejsza złożoność projektu i systemu.
13. Architektura może być podstawą do szkolenia nowych członków zespołu.

Nawet jeśli już nam wierzysz, że architektura jest ważna, i nie musisz 13 razy sobie tego powtarzać, potraktuj tych 13 punktów (wyznaczających strukturę tego rozdziału) jako 13 przydatnych sposobów na wykorzystanie architektury w projekcie lub na uzasadnienie nakładów na opracowanie architektury.

2.1. Utrudnianie albo ułatwianie uzyskania głównych atrybutów jakościowych systemu

Możliwość osiągnięcia pożądaných (lub wymaganych) atrybutów jakościowych w systemie w dużym stopniu zależy od jego architektury. Jeśli masz zapamiętać z tej książki jedną rzecz, niech będzie to właśnie to.

Ta zależność jest tak ważna, że poświęciliśmy całą część II tej książki na to, aby szczegółowo ją wyjaśnić. Na razie jako punkt wyjścia zapamiętaj następujące przykłady:

- Jeśli system musi mieć wysoką wydajność, trzeba zwrócić uwagę na zarządzanie związanym z czasem zachowaniem elementów, korzystaniem ze współużytkowanych zasobów oraz częstością i ilością komunikacji między elementami.
- Jeżeli istotna jest modyfikowalność, należy zwrócić uwagę na przydział zadań do elementów i ograniczenie interakcji (powiązań) między elementami, aby większość zmian w systemie wpływała tylko na niewielką liczbę elementów. W idealnym scenariuszu każda zmiana powinna dotyczyć tylko jednego elementu.
- Jeśli system musi być dobrze zabezpieczony, trzeba zarządzać komunikacją między elementami i chronić ją. Należy też kontrolować, które elementy mają dostęp do poszczególnych informacji. Możliwe, że trzeba będzie wprowadzić do architektury wyspecjalizowane elementy (na przykład mechanizmy autoryzacji), aby zbudować solidną granicę chroniącą przed atakami.
- Jeżeli system ma być zabezpieczony i bezpieczny, trzeba zaprojektować zabezpieczenia i mechanizmy odzyskiwania stanu.
- Jeśli uważasz, że powodzenie systemu zależy od skalowalności wydajności, musisz przeanalizować wykorzystanie zasobów i umożliwić wprowadzanie

zastępników o wyższych możliwościach. Należy też unikać zapisywania nieprzekraczalnych założeń lub limitów dotyczących zasobów.

- Jeżeli projekty wymagają stopniowego dodawania podzbiorów systemu, musisz zarządzać tym, jak komponenty używają siebie nawzajem.
- Jeśli chcesz, aby elementy systemu można było zastosować także w innych systemach, musisz ograniczyć powiązania między elementami, by po wyodrębnieniu jakiegoś elementu nie miał on zbyt wielu zależności od obecnego środowiska i pozostał użyteczny.

Strategie dotyczące tych i innych atrybutów jakościowych w dużej mierze są związane z architekturą. Jednak sama architektura nie gwarantuje funkcji ani atrybutów jakościowych wymaganych od systemu. Niskiej jakości projekt z dalszych etapów prac lub złe decyzje projektowe zawsze mogą zniweczyć poprawny projekt architektoniczny. Czasem mówimy (*najczęściej dla żartu*), że „co architektura dała, implementacja może odebrać”. Jakość systemu zależy od decyzji podejmowanych na wszystkich etapach cyklu życia oprogramowania: od projektowania architektury po pisanie kodu i testy. Dlatego jakość nie wynika wyłącznie z projektu architektury, ale dbałość o nią zaczyna się właśnie od tego punktu.

2.2. Wnioskowanie na temat zmian i zarządzanie nimi

Jest to następstwo poprzedniego punktu.

Modyfikowalność, czyli łatwość wprowadzania zmian w systemie, jest atrybutem jakościowym, a tym samym dotyczą go uwagi z poprzedniego podrozdziału. Ten atrybut jest jednak tak ważny, że zasłużył na własne miejsce na „liście trzynastu”. Ludzie zajmujący się rozwojem oprogramowania zaczynają rozumieć, że mniej więcej 80% łącznych kosztów związanych z typowym systemem oprogramowania jest generowanych *po* jego wdrożeniu. Większość systemów, nad jakimi pracujemy, znajduje się właśnie na tym etapie. Wielu programistów i projektantów oprogramowania *nigdy* nie zajmuje się tworzeniem nowych systemów. Te osoby muszą uwzględniać ograniczenia związane z istniejącą architekturą i obecnym kodem. Prawie wszystkie systemy oprogramowania zmieniają się w ciągu życia. Modyfikacje są konieczne, aby uwzględnić nowe funkcje, dostosować system do nowych środowisk, poprawić błędy itd. Jednak wprowadzanie zmian często jest trudne.

W każdej architekturze niezależnie od jej postaci można podzielić zmiany na trzy kategorie: lokalne, nielocalne i architektoniczne.

- Zmiany lokalne można wprowadzić przez zmodyfikowanie jednego elementu, na przykład dodanie reguły biznesowej do modułu wyznaczania cen.

- Zmiana nielokalna wymaga modyfikacji kilku elementów, ale z zachowaniem pierwotnego podejścia architektonicznego. Dzieje się tak na przykład, gdy trzeba utworzyć nową regułę biznesową w module wyznaczania cen, a następnie dodać do bazy danych nowe pola niezbędne dla tej reguły i wyświetlić wyniki zastosowania tej reguły w interfejsie użytkownika.
- Zmiana architektoniczna wpływa na sposób interakcji między elementami i często wymaga modyfikacji w wielu miejscach systemu, na przykład przejścia z modelu jednowątkowego na wielowątkowy.

Oczywiście najbardziej pożądane są zmiany lokalne, dlatego *skuteczna* architektura powinna sprawiać, że większość zmian jest lokalna, a tym samym łatwa do wprowadzenia. Zmiany nielocalne są niepożądane, ale mają tę zaletę, że przeważnie można je wprowadzać stopniowo w uporządkowany sposób. Można na przykład najpierw utworzyć nową regułę wyznaczania cen, a następnie wprowadzić modyfikacje potrzebne do zastosowania tej reguły.

Decydowanie o tym, kiedy zmiany są niezbędne, ustalanie, jakie ścieżki ich wprowadzania są narażone na najmniejsze ryzyko, ocena skutków proponowanych modyfikacji i wyznaczanie sekwencji oraz priorytetów zmian wymaga dobrego zrozumienia relacji, wydajności i działania elementów systemu oprogramowania. Wszystkie te zadania pojawiają się w opisie stanowiska architekta. Wnioskowanie na temat architektury i jej analizowanie może zapewnić wiedzę niezbędną do podejmowania decyzji o przewidywanych zmianach. Jeśli pominiemy ten etap i nie zadbasz o utrzymanie koncepcyjnej integralności architektury, prawie na pewno doprowadzisz do narastania *długu architektonicznego*. Ten temat omawiamy w rozdziale 23.

2.3. Prognozowanie atrybutów jakościowych systemu

Ten punkt wynika z dwóch wcześniejszych: architektura nie tylko wpływa na atrybuty jakościowe systemu, ale też robi to w przewidywalny sposób.

Może się to wydawać oczywiste, ale nie dla każdego takie jest. Wtedy projektowanie architektury polega na podejmowaniu serii w dużym stopniu losowych decyzji projektowych, po czym system jest budowany i następuje sprawdzenie atrybutów jakościowych w nadziei na powodzenie. Ups, system nie jest wystarczająco szybki lub okazał się zatrważająco podatny na ataki? Zabieramy się do pracy.

Na szczęście *możliwe* jest prognozowanie atrybutów jakościowych systemu wyłącznie na podstawie oceny jego architektury. Jeśli wiesz, że określone decyzje architektoniczne prowadzą do uzyskania danych atrybutów jakościowych, możesz podjąć te decyzje i słusznie oczekiwać nagrody w postaci tych atrybutów. Po zakończeniu prac,

w trakcie oceny architektury, możesz stwierdzić, czy podjęte zostały odpowiednie decyzje, i z wysokim prawdopodobieństwem przewidzieć, czy architektura będzie miała określone cechy.

Ten i poprzedni punkt razem oznaczają, że architektura w dużym stopniu determinuje atrybuty jakościowe systemu i, co jeszcze lepsze, wiadomo, w jaki sposób to robi, dzięki czemu wiadomo też, jak z jej użyciem osiągnąć pożądaný efekt.

Nawet jeśli nie stosujesz analitycznego modelowania atrybutów jakościowych (czasem takie modelowanie jest konieczne, aby zagwarantować, że architektura zapewni określone korzyści), zasada oceny decyzji na podstawie ich wpływu na atrybuty jakościowe jest nieoceniona choćby dlatego, że pozwala wcześniej wykryć potencjalne problemy.

2.4. Komunikacja między interesariuszami

W rozdziale 1. napisaliśmy, że architektura jest abstrakcją. Wspomnieliśmy też, że jest przydatna, ponieważ reprezentuje uproszczony model całego systemu, który — w odróżnieniu od nieskończonej listy szczegółów dotyczących systemu — możesz utrzymywać w umyśle. To samo dotyczy innych członków zespołu. Architektura reprezentuje wspólną abstrakcję systemu, którą większość interesariuszy (a nawet każdy z nich) może posługiwać się jako podstawą do wzajemnego zrozumienia, w negocjacjach, do znajdowania konsensusu i do komunikacji. Architektura, a przynajmniej jej części, jest wystarczająco abstrakcyjna, aby większość osób bez wiedzy technicznej potrafiła zrozumieć ją na potrzebnym im poziomie, zwłaszcza z pewną pomocą ze strony architekta. Jednocześnie tę abstrakcję można rozbudować do postaci odpowiednio szczegółowych specyfikacji technicznych pomocnych w trakcie implementowania, integrowania, testowania i wdrażania systemu.

Każdy interesariusz związany z systemem oprogramowania (klient, użytkownik, menedżer projektu, koder, tester itd.) jest zainteresowany innymi cechami systemu wynikającymi z architektury. Oto przykład:

- dla użytkownika ważne jest, aby system był szybki, niezawodny i dostępny wtedy, gdy jest potrzebny;
- dla klienta (który płaci za system) liczy się, aby architekturę można było zaimplementować zgodnie z harmonogramem i budżetem;
- dla menedżera istotne jest (obok kosztów i harmonogramu), aby architektura umożliwiała zespołom w dużym stopniu niezależną pracę oraz interakcje w zdyscyplinowany i kontrolowany sposób;
- architekt zajmuje się strategiami pozwalającymi osiągnąć wszystkie wymienione cele.

Architektura zapewnia wspólny język, w którym różne kwestie można wyrazić, negocjować i rozwiązywać na poziomie dostępnym intelektualnie nawet w przypadku dużych, złożonych systemów. Bez takiego języka trudno jest wystarczająco dobrze zrozumieć duże systemy, aby móc na wczesnych etapach prac podejmować decyzje wpływające zarówno na jakość, jak i przydatność rozwiązania. Analiza architektoniczna, jak pokazujemy w rozdziale 21., zarówno zależy od tego poziomu komunikacji, jak i ją ułatwia.

W rozdziale 22., poświęconym dokumentacji architektury, szczegółowo omawiamy interesariuszy i ich potrzeby.

„Co się stanie, jeśli wcisnę ten przycisk?” – architektura jako narzędzie do komunikacji między interesariuszami

Przeгляд projektu ciągnął się w nieskończoność. Finansowane przez rząd prace były opóźnione, a koszty przekroczyły planowany budżet. Projekt był na tyle duży, że te zaniedbania przyciągnęły uwagę amerykańskiego Kongresu. Obecnie rząd nadrabiał wcześniejsze zaległości, planując niekończącą się sesję przeglądową z udziałem wszystkich stron. Kontraktor został niedawno wykupiony, co też nie pomagało w pracach. Było popołudnie drugiego dnia przeglądu i zgodnie z planem miała zostać zaprezentowana architektura oprogramowania. Młody architekt, uczeń głównego architekta systemu, odważnie wyjaśniał, że architektura oprogramowania tego rozbudowanego systemu pozwoli spełnić bardzo ambitne wymogi dotyczące pracy w czasie rzeczywistym w środowisku rozproszonym przy wysokim stopniu niezawodności. Architekt miał dobrą prezentację i solidną architekturę, poprawną i sensowną. Jednak słuchacze, około 30 przedstawicieli rządu pełniących różne funkcje w zarządzie i przy nadzorze tego problematycznego projektu, byli zmęczeni. Niektórzy z nich zastanawiali się nawet nad zajęciem się handlem nieruchomościami, aby tylko uniknąć następnego z długiej serii przeglądów „zrobmy to wreszcie tak, jak należy”.

Pojawił się slajd, na którym za pomocą półformalnej notacji z polami i liniami pokazane były główne elementy oprogramowania w perspektywie czasu wykonania. Wszystkie nazwy były akronimami, niezrozumiałymi bez wyjaśnień, które młody architekt zaprezentował. Linie przedstawiały przepływ danych, przepływ komunikatów i synchronizację procesów. Architekt wyjaśnił, że dla elementów zastosowano nadmiarowość. „W przypadku awarii” – zaczęła, pokazując jedną z linii wskaźnikiem laserowym – „uruchamiany jest mechanizm wznawiania pracy zgodnie z tą ścieżką...”.

„A co się stanie, jeśli wciśnięty zostanie przycisk wyboru trybu?” — przerwał jeden ze słuchaczy. Był to przedstawiciel rządu, reprezentujący użytkowników systemu.

„Przepraszam bardzo, nie rozumiem pytania” — odpowiedział architekt.

„Przycisk wyboru trybu” — powtórzył słuchacz. „Co się stanie, jeśli go wciśniemy?”.

„Hmm, to uruchomi zdarzenie w sterowniku urządzenia, w tym miejscu” — rozpoczął architekt, pokazując element wskaźnikiem. „System wczyta zawartość rejestru i zinterpretuje kod zdarzenia. Jeśli dotyczy ono wyboru trybu, to, no cóż, przesłany zostanie sygnał do tablicy, która z kolei przekaże go obiektom subskrybującym to zdarzenie...”.

„Nie, nie. Chodzi mi o to, co system *zrobi*” — przerwał autor pytania. „Czy zresetuje ekrany? I co się stanie, jeśli stanie się to w trakcie rekonfigurowania systemu?”.

Architekt wyglądał na nieco zaskoczonego i wyłączył wskaźnik. Nie było to pytanie z zakresu architektury, ale ponieważ prezentujący był architektem i dlatego dobrze znał wymogi dotyczące systemu, znał odpowiedź. „Jeśli wiersz poleceń znajduje się w trybie konfiguracji, ekrany zostaną zresetowane” — powiedział. „W przeciwnym razie w konsoli sterowania pojawi się komunikat o błędzie, a sygnał zostanie zignorowany”. Architekt ponownie włączył wskaźnik. „Wróćmy teraz do mechanizmu wznawiania pracy, o którym mówiłem...”.

„Hmm, tak się zastanawiam” — powiedział przedstawiciel użytkowników. „Ponieważ z wykresu wynika, że konsola przesyła sygnały do modułu lokalizowania celu”.

„A co *powinno* się dziać?” — spytał inny słuchacz autora pierwotnego pytania. „Czy naprawdę chciałbyś, żeby użytkownik otrzymywał dane o trybie w trakcie rekonfiguracji?”. Przez następnych 45 minut architekt obserwował, jak słuchacze zabierali jego czas, dyskutując na temat właściwego działania systemu w różnych rzadko występujących stanach. Były to bardzo ważne rozmowy, które jednak należało przeprowadzić na etapie formułowania wymogów, co z niewiadomych powodów nie nastąpiło.

Ta dyskusja nie dotyczyła architektury, jednak została wywołana przez architekturę i jej graficzną reprezentację. Naturalne jest myślenie o architekturze jak o podstawie do komunikacji między niektórymi interesariuszami, nie tylko architektami i programistami. Na przykład menedżerowie mogą na podstawie architektury tworzyć zespoły i przydzielać im zasoby. Ale użytkownicy? Przecież architektura jest dla nich niewidoczna. Dlaczego mieliby traktować ją jak narzędzie do zrozumienia systemu?

Tak się jednak dzieje. W omawianym przykładzie autor pytania przez dwa dni oglądał perspektywy ilustrujące funkcje, operacje, interfejsy użytkownika i testy. Jednak dopiero pierwszy slajd przedstawiający architekturę sprawił, że słuchacz, choć zmęczony i czekający tylko na powrót do domu, zdał sobie sprawę,

że czegoś nie rozumiał. Uczestnictwo w wielu przeglądach architektury przekonało mnie, że spojrzenie na system z nowego punktu widzenia pobudza umysł i rodzi nowe pytania. Dla użytkowników to architektura jest często tym nowym sposobem, a zadawane pytania dotyczą działania systemu. W trakcie pamiętnego dla mnie ćwiczenia z oceny architektury, które miało miejsce kilka lat temu, przedstawiciele użytkowników byli znacznie bardziej zainteresowani tym, co system będzie robić, niż tym, jak to robi. To zupełnie naturalne. Wcześniej kontaktowali się z producentem tylko za pośrednictwem marketingowców. Architekt był pierwszym prawdziwym ekspertem od systemu, z którym mogli porozmawiać, i nie wahali się wykorzystać tej okazji.

Naturalnie staranna i dokładna specyfikacja wymagań ułatwiłaby pracę, ale z różnych powodów nie zawsze jest tworzona lub dostępna. Gdy jej brak, specyfikacja architektury często zachęca do zadawania pytań i zwiększa jasność sytuacji. Prawdopodobnie rozsądniejsze jest zaakceptowanie tego faktu niż przeciwdziałanie mu.

Czasem przegląd architektury ujawnia też nieuzasadnione wymagania, których przydatność można ponownie ocenić. Przegląd tego rodzaju, w trakcie którego podkreślana jest synergia między wymogami a architekturą, umożliwiłby młodemu architektowi z naszej historii wyjście z impasu, ponieważ możliwe byłoby poświęcenie całej sesji na omówienie informacji tego typu. Ponadto przedstawiciel użytkowników uniknąłby skrzepowania wynikającego z zadania pytania w najwyraźniej niewłaściwym momencie.

— PCC

2.5. Wczesne decyzje projektowe

Architektura oprogramowania jest odzwierciedleniem najwcześniejszych decyzji projektowych dotyczących systemu. Te wczesne wytyczne mają olbrzymi wpływ na późniejszy rozwój systemu, jego wdrażanie i konserwację. Jest to także najwcześniejszy moment, w którym można przeanalizować istotne decyzje projektowe dotyczące systemu.

Każdy projekt w każdej dziedzinie można postrzegać jak sekwencję decyzji. W procesie malowania obrazu artysta jeszcze przed rozpoczęciem prac decyduje się na materiał płótna i narzędzia do malowania: farbę olejną, akwarele, kredki. W momencie rozpoczynania malowania podejmuje inne decyzje: gdzie narysować pierwszą linię, jakiej grubości, o jakim kształcie? Wszystkie te wczesne decyzje projektowe mają istotny wpływ na ostateczny wygląd obrazu. Każda z tych decyzji ogranicza też listę późniejszych wyborów. Każda pojedyncza decyzja może się wydawać mało istotna,

jednak zwłaszcza wczesne wybory mają nieproporcjonalnie duże znaczenie, ponieważ wpływają na tak wiele dalszych kroków i ograniczają ich zakres.

Tak samo jest z projektowaniem architektury. Ten proces też można traktować jako zestaw decyzji. Zmiana wczesnych decyzji wywołuje efekt domina, ponieważ wymaga modyfikacji także innych wyborów. To prawda, czasem konieczna jest refaktoryzacja lub zmiana projektu architektury, jednak nie należy pochopnie podejmować takich decyzji, gdyż efekt domina może łatwo przekształcić się w efekt lawiny.

Czego dotyczą wczesne decyzje projektowe ujęte w architekturze oprogramowania? Rozważ następujące pytania:

- Czy system będzie działał w jednym procesorze, czy będzie rozproszony między wiele procesorów?
- Czy oprogramowanie będzie warstwowe? Jeśli tak, to ile będzie warstw? Co będzie robić każda z nich?
- Czy komponenty będą komunikować się synchronicznie, czy asynchronicznie? Będą komunikować się ze sobą przez przekazywanie sterowania, przekazywanie danych czy z użyciem obu tych technik?
- Czy informacje przepływające przez system będą szyfrowane?
- Jaki system operacyjny będzie używany?
- Jaki protokół komunikacji wybierzesz?

Wyobraź sobie, jakim koszmarem byłaby zmiana dowolnej z tych lub wielu innych podobnych decyzji. Tego rodzaju wybory wpływają na kształt struktur architektury i interakcji między nimi.

2.6. Ograniczenia dotyczące implementacji

Jeśli chcesz, aby implementacja była zgodna z architekturą, trzeba dostosować ją do decyzji projektowych reprezentowanych w tej architekturze. Konieczne jest utworzenie zestawu elementów przedstawionych w architekturze, elementy muszą komunikować się między sobą w sposób opisany w architekturze, a każdy element musi wykonywać swoje obowiązki zgodnie z architekturą. Każdy opis z architektury jest ograniczeniem dla autora implementacji.

Autorzy elementów muszą najpierw dobrze poznać specyfikacje poszczególnych elementów, jednak czasem nie znają kompromisów narzucanych przez architekturę. Architektura (lub architekt) ograniczają autorów implementacji w taki sposób, aby przestrzegać tych kompromisów. Oto klasyczny przykład: architekt przydziela budżet wydajności do fragmentów oprogramowania związanych z większą funkcją. Jeśli każda

jednostka oprogramowania zmieści się w budżecie, cała transakcja będzie spełniać wymogi dotyczące wydajności. Autorzy implementacji poszczególnych fragmentów mogą nie znać całego budżetu; wiedzą tylko, ile wynosi ich budżet.

Z kolei architekci nie muszą być ekspertami od wszystkich aspektów projektowania algorytmów lub szczegółów języka programowania, choć oczywiście powinni mieć wystarczającą wiedzę, aby nie zaprojektować czegoś, co trudno będzie zbudować. Architekci odpowiadają za podejmowanie, analizowanie i zapewnianie przestrzegania decyzji oraz kompromisów architektonicznych.

2.7. Wpływ na strukturę organizacji

Architektura opisuje strukturę rozwijanego systemu, a ponadto ta struktura ma odzwierciedlenie w strukturze prac (a czasem nawet w strukturze całej organizacji). Standardowa metoda podziału prac w dużym projekcie polega na przydzielaniu różnym grupom porcji systemu do zbudowania. W architekturze ta struktura podziału prac ma postać struktury przydziału prac opisanej w rozdziale 1. Ponieważ architektura reprezentuje najbardziej ogólny podział systemu, zwykle jest używana jako punkt wyjścia do tworzenia struktury podziału prac. Z kolei struktura podziału prac wpływa na jednostki planowania, harmonogram i budżet, kanały komunikacji między zespołami, zarządzanie konfiguracją i układ systemu plików, plany i procedury integracji oraz testów, a nawet szczegóły takie jak struktura intranetu używanego w projekcie lub usadowanie osób na firmowym pikniku. Zespoły komunikują się między sobą na podstawie specyfikacji interfejsów rozwijanych elementów. Także konserwacja uruchomionego systemu odzwierciedla strukturę oprogramowania. Zespoły są tworzone w taki sposób, aby zajmować się konserwacją określonych elementów architektury: bazy danych, reguł biznesowych, interfejsu użytkownika, sterowników urządzeń itd.

Efektym ubocznym utworzenia struktury podziału prac jest „zamrożenie” niektórych aspektów architektury oprogramowania. Grupa odpowiedzialna za jeden z podsystemów może być przeciwna rozdzieleniu jej zadań między inne grupy. Jeśli te zadania zostały formalnie opisane w kontrakcie, zmiana obowiązków może być kosztowna lub nawet prowadzić do sporów prawnych.

Dlatego po uzgodnieniu architektury wprowadzanie w niej istotnych zmian staje się bardzo kosztowne (z przyczyn biznesowych i związanych z zarządzaniem). Jest to jeden z wielu argumentów za tym, aby architekturę dużego systemu przeanalizować przed podjęciem ostatecznych decyzji.

2.8. Umożliwianie stopniowego rozwijania systemu

Po zdefiniowaniu architektura może być podstawą do stopniowego rozwijania systemu. Pierwszym etapem może być szkieletowy system, w którym dostępna jest przynajmniej część infrastruktury (odpowiedzialna za to, jak elementy są inicjowane, jak się komunikują, współużytkują dane, uzyskują dostęp do zasobów, informują o błędach, rejestrują operacje itd.), ale wiele funkcji aplikacji z systemu jeszcze nie działa.

Tworzenie infrastruktury i rozwijanie funkcji aplikacji mogą się odbywać jednocześnie. Zaprojektuj i zbuduj niewielką część infrastruktury potrzebną do obsługi niewielkiej porcji funkcji, a następnie powtarzaj te kroki do momentu ukończenia systemu.

Wiele systemów jest budowanych w formie szkieletowej, którą można rozszerzać za pomocą wtyczek, pakietów lub rozszerzeń. Przykładami są: język R, edytor Visual Studio Code i większość przeglądarek internetowych. Rozszerzenia zapewniają dodatkowe funkcje obok tych dostępnych w wersji szkieletowej. To podejście ułatwia proces rozwoju, ponieważ gwarantuje, że system można uruchomić już na wczesnych etapach cyklu życia produktu. Wartość systemu rośnie wraz z dodawaniem rozszerzeń lub zastępowaniem wczesnych wersji określonych części oprogramowania ich bardziej kompletnymi odpowiednikami. Zdarza się, że części są wersjami o niskiej wartości lub prototypami ostatecznych funkcji. W innych sytuacjach tworzone są *zastępniki*, które konsumują i produkują dane z określoną szybkością, wykonując niewiele innych operacji. Pozwala to między innymi wykryć problemy z wydajnością (i inne) na wczesnych etapach cyklu życia produktu.

To podejście stało się znane na początku XXI wieku dzięki pomysłom Alistaira Cockburna i jego „chodzącego szkieletu”. Ostatnio zaczęły je stosować osoby posługujące się strategią MVP (ang. *minimum viable product*; jest to podstawowa wersja produktu o minimalnej funkcjonalności) do ograniczania ryzyka.

Jedną z korzyści płynących ze stopniowego rozwoju jest ograniczenie ryzyka w projekcie. Jeśli architektura dotyczy rodziny powiązanych systemów, infrastrukturę można wykorzystać w całej tej rodzinie, co zmniejsza koszty prac nad poszczególnymi systemami.

2.9. Szacowanie kosztów i harmonogramu prac

Szacunki kosztów i harmonogramu prac są ważnym narzędziem dla menedżera projektu. Pomagają pozyskać niezbędne zasoby, a także monitorować postęp prac nad projektem. Jednym z zadań architekta jest pomoc menedżerowi projektu w oszacowaniu

kosztów i harmonogramu na wczesnych etapach cyklu życia projektu. Szacunki generowane metodą „od ogółu do szczegółu” są przydatne do wyznaczania celów i planowania budżetu, jednak szacunki kosztów otrzymane metodą „od szczegółu do ogółu” na podstawie zrozumienia fragmentów systemu są zwykle bardziej precyzyjne niż te oparte na ogólnej wiedzy o systemie.

Wcześniej wspomnieliśmy, że struktura organizacyjna i struktura podziału prac w projekcie są prawie zawsze oparte na architekturze. Każdy zespół i każda osoba odpowiedzialna za jednostkę pracy potrafi dokładniej niż menedżer projektu przedstawić szacunki dotyczące swojego fragmentu i czuje większą odpowiedzialność za to, aby rzeczywiste wyniki się z nimi pokrywały. Jednak najlepsze szacunki kosztów i harmonogramu prac są wynikiem połączenia podejścia top-down (stosowanego przez architekta i menedżera projektu) z podejściem bottom-up (stosowanym przez programistów). Dyskusje i negocjacje prowadzone w takim procesie skutkują znacznie precyzyjniejszymi szacunkami niż stosowanie któregośkolwiek z obu podejść z osobna.

Korzystne jest, jeśli wymogi dotyczące systemu zostały już sprawdzone i zatwierdzone. Im większa jest wiedza na temat zakresu prac, tym dokładniejsze będą szacunki kosztów i harmonogramu.

W rozdziale 24. omawiamy wykorzystanie architektury do zarządzania projektem.

2.10. Uniwersalny model wielokrotnego użytku

Im wcześniej w cyklu życia fragmenty oprogramowania są ponownie używane, tym większe może to przynieść korzyści. Choć ponowne użycie kodu jest korzystne, ponowne użycie architektury pozwala znacznie ułatwić rozwój systemów, którym stawiane są podobne wymagania. Gdy decyzje architektoniczne można ponownie zastosować w wielu systemach, wszystkie wcześniej opisane konsekwencje wczesnego dokonywania wyborów będą dotyczyć także tych systemów.

Linia lub rodzina produktów to zestaw systemów budowanych za pomocą tego samego zbioru współużytkowanych zasobów: komponentów oprogramowania, dokumentów z wymogami, przypadków testowych itd. Najważniejszym z tych zasobów jest architektura zaprojektowana tak, aby zaspokajać potrzeby całej rodziny produktów. Architekci linii produktów wybierają architekturę (lub rodzinę ściśle powiązanych architektur), która będzie przydatna dla wszystkich planowanych produktów. Taka architektura określa, co jest stałe dla wszystkich produktów z rodziny, a co może się zmieniać.

Linie produktów są wartościowym sposobem rozwijania wielu systemów, który w praktyce daje olbrzymie korzyści w czasie do wprowadzenia produktu na rynek,

kosztach, produktywności i jakości produktów. Architektura jest istotą tego podejścia. Podobnie jak inne inwestycje kapitałowe, tak i architektury linii produktów stają się współużytkowanymi zasobami organizacji.

2.11. Architektura umożliwia dodawanie niezależnie rozwijanych elementów

We wcześniejszych modelach rozwoju oprogramowania najważniejszą aktywnością było *programowanie*, a postęp był mierzony w wierszach kodu. W rozwoju opartym na architekturze często koncentrujemy się na *łączeniu elementów*, które nieraz są tworzone osobno, a nawet niezależnie od siebie. Takie łączenie jest możliwe, ponieważ architektura określa elementy, które można włączyć do systemu. Architektura ogranicza możliwe zastępniki (lub dodatki) na podstawie ich interakcji ze środowiskiem, przejmowania i zwalniania kontroli, konsumowanych i produkowanych danych, dostępu do danych i protokołów używanych do komunikacji i współużytkowania zasobów. Więcej na ten temat piszemy w rozdziale 15.

Gotowe komercyjne komponenty, oprogramowanie otwartoźródłowe, publicznie dostępne aplikacje i usługi sieciowe to przykłady niezależnie rozwijanych elementów. Złożoność i powszechność procesu integrowania wielu niezależnie zbudowanych jednostek w systemie doprowadziły do powstania całej grupy narzędzi programowych, takich jak Apache Ant, Apache Maven, MSBuild i Jenkins.

W przypadku oprogramowania korzyści mogą być następujące:

- skrócenie czasu do wprowadzenia produktu na rynek (łatwiej powinno być zastosować czyjeś gotowe rozwiązanie, niż zbudować własne),
- wyższa niezawodność (w powszechnie używanym oprogramowaniu większość błędów powinna już być wyeliminowana),
- obniżenie kosztów (dostawca oprogramowania może przerzucić koszty prac rozwojowych na wielu klientów),
- elastyczność (jeśli element, który chcesz kupić, nie jest wysoce wyspecjalizowany, często jest dostępny z kilku źródeł, co poprawia Twoją pozycję negocjacyjną).

Otwarty system charakteryzuje się tym, że definiuje zestaw standardów dla elementów oprogramowania. Te standardy określają, jak elementy się zachowują, jak wchodzi z sobą w interakcje, jak współużytkują dane itd. Otwarty system ma umożliwić wielu dostawcom tworzenie potrzebnych elementów, a nawet zachęcać ich do tego. Pozwala to uniknąć uzależnienia się od producenta, kiedy tylko jeden producent

potrafi udostępnić dany element, przez co może dyktować wyższe ceny. Otwarte systemy są oparte na architekturze określającej elementy i interakcje między nimi.

2.12. Ograniczanie listy możliwości w projekcie

Po zebraniu różnych przydatnych rozwiązań architektonicznych staje się jasne, że choć elementy oprogramowania można łączyć ze sobą na niemal nieskończenie wiele sposobów, korzystne może być dobrowolne ograniczenie się do stosunkowo niewielkiej liczby elementów i interakcji między nimi. Dzięki temu minimalizujemy złożoność procesu projektowania rozwijanego systemu.

Inżynier oprogramowania nie jest *artystą*, dla którego najważniejsze są kreatywność i swoboda. W inżynierii ważna jest dyscyplina, która po części bierze się z *ograniczania* listy możliwości do sprawdzonych rozwiązań. Przykładowe sprawdzone rozwiązania projektowe to taktyki i wzorce, które szczegółowo omawiamy w części II tej książki. Ponowne wykorzystanie gotowych elementów to następne podejście pozwalające ograniczyć listę możliwości w projekcie.

Ograniczenie listy możliwości w projekcie do sprawdzonych rozwiązań może przynieść następujące korzyści:

- wyższy poziom ponownego wykorzystania elementów,
- bardziej standardowe i prostsze projekty, które są łatwiejsze do zrozumienia i przekazania, a także dają bardziej przewidywalne skutki,
- łatwiejsze analizy z wyższym poziomem pewności co do ich poprawności,
- krótszy czas dokonywania wyborów,
- lepsze współdziałanie elementów.

Bezprecedensowe projekty są ryzykowne. Sprawdzone projekty są, no cóż, sprawdzone. Nie chcemy przez to powiedzieć, że projekt oprogramowania nigdy nie może być innowacyjny ani nie powinien obejmować nowych i ekscytujących rozwiązań. Jest to do przyjęcia. Jednak rozwiązań nie należy wymyślać tylko po to, aby były nowatorskie. Nowinek należy szukać, jeśli istniejące rozwiązania nie wystarczają do rozwiązania danego problemu.

Cechy oprogramowania wynikają z wyboru taktyk lub wzorców architektonicznych. Taktyki i wzorce pożądane w określonym problemie powinny zwiększać jakość wynikowego rozwiązania projektowego, na przykład ułatwiać wybór między sprzecznymi ograniczeniami projektowymi, zwiększać wiedzę na temat słabo zrozumianych kontekstów projektu i pomagać ujawnić niespójne aspekty wymagań. Taktyki i wzorce architektoniczne omawiamy w części II.

2.13. Podstawa do szkoleń

Architektura, włącznie z opisem interakcji między elementami w celu wykonywania oczekiwanych zadań, może stanowić pierwsze wprowadzenie do systemu dla nowych uczestników prac nad projektem. To podkreśla nasze słowa na temat tego, że ważnym zastosowaniem architektury oprogramowania jest ułatwianie komunikacji między różnymi interesariuszami i zachęcanie do niej. Architektura stanowi wspólny punkt odniesienia dla wszystkich zainteresowanych.

Perspektywy modułów doskonale nadają się do prezentowania struktury projektu: kto czym się zajmuje, które zespoły są przydzielone do poszczególnych części systemu itd. Perspektywy komponenty-złącza są świetnym wyborem do wyjaśniania, jak system powinien działać i wykonywać swoje zadania. Perspektywy alokacji informują nowych członków projektu o tym, jakie miejsce przydzielone im elementy zajmują w środowisku rozwojowym lub wdrożeniowym projektu.

2.14. Podsumowanie

Architektura oprogramowania jest ważna z wielu powodów technicznych i nietechnicznych. Nasza 13-punktowa lista obejmuje następujące korzyści:

1. Architektura może albo utrudniać, albo ułatwiać uzyskanie głównych atrybutów jakościowych systemu.
2. Decyzje podjęte w trakcie tworzenia architektury umożliwiają wnioskowanie na temat zmian i zarządzanie nimi w trakcie ewoluowania systemu.
3. Analiza architektury umożliwia wczesne oszacowanie atrybutów jakościowych systemu.
4. Udokumentowana architektura ułatwia komunikację między interesariuszami.
5. Architektura jest nośnikiem najwcześniejszych, a tym samym najbardziej fundamentalnych i najtrudniejszych do zmiany decyzji projektowych.
6. Architektura określa zestaw ograniczeń dotyczących późniejszej implementacji.
7. Architektura wyznacza strukturę organizacji lub na odwrót.
8. Architektura może być podstawą do stopniowego rozwijania oprogramowania.
9. Architektura jest najważniejszym artefaktem, który umożliwia architektowi i menedżerowi projektu wnioskowanie na temat kosztów i harmonogramu prac.
10. Architekturę można stworzyć w formie uniwersalnego modelu wielokrotnego użytku, który stanowi istotę linii produktowej.

11. Programowanie oparte na architekturze wymaga skupienia się na łączeniu komponentów, a nie tylko na ich tworzeniu.
12. Przez ograniczenie alternatywnych projektów architektura kierkuje kreatywność programistów, ponieważ zmniejsza złożoność projektu i systemu.
13. Architektura może być podstawą do szkolenia nowych członków zespołu.

2.15. Literatura

W książce *The Software Architect Elevator: Redefining the Architect's Role in the Digital Enterprise* Gregora Hohpego znajdziesz opis wyjątkowych możliwości, jakie architekci mają w zakresie interakcji z osobami zajmującymi różne stanowiska w firmie i poza nią oraz ułatwiania komunikacji między interesariuszami [Hohpe 20].

Praprzodkiem prac na temat architektury i organizacji jest [Conway 68]. Zgodnie z prawem Conwaya „organizacje, które projektują systemy [...] są ograniczone do tworzenia projektów będących kopiami struktur komunikacji w tej organizacji”.

Koncepcja „chodzącego szkieletu” została opisana przez Cockburna w książce *Agile Software Development: The Cooperative Game* [Cockburn 06].

Dobrym przykładem standardu dla architektur otwartych systemów jest AUTOSAR. Został on opracowany na potrzeby branży motoryzacyjnej (<https://www.autosar.org/>).

Kompletne omówienie tworzenia linii produktów informatycznych znajdziesz w [Clements 16]. Inżynieria linii produktów na podstawie cech to nowe, oparte na automatyzacji podejście do budowania linii produktów, które rozszerza inżynierię oprogramowania do poziomu systemów. Dobre podsumowanie tego zagadnienia znajdziesz w [INCOSE 19].

2.16. Pytania do dyskusji

1. Jeśli masz zapamiętać tylko jedną rzecz z tej książki, jest nią... Dodatkowe punkty przyznaj sobie za niepodglądanie.
2. Dla każdego z 13 opisanych w tym rozdziale powodów, dla których architektura jest ważna, przyjmij postawę przekorną: opisz okoliczności, w jakich architektura nie jest konieczna, aby uzyskać pożądane wyniki. Uzasadnij swoje zdanie. Postaraj się wymyślić inne okoliczności dla każdego z 13 powodów.
3. W tym rozdziale argumentujemy, że architektura zapewnia szereg konkretnych korzyści. Jak można zmierzyć w określonym projekcie korzyści związane z każdym z 13 punktów?

4. Załóżmy, że chcesz wprowadzić w swojej firmie podejście oparte na architekturze. Zarząd jest otwarty na propozycję, ale chce wiedzieć, jaki będzie zwrot z inwestycji. Jak na to odpowiesz?
5. Uporządkuj według ważności 13 powodów z tego rozdziału. Zastosuj kryteria, które są dla Ciebie istotne. Odpowiedź uzasadnij. Jeśli możesz wybrać tylko dwa lub trzy powody, aby zarekomendować zastosowanie architektury w projekcie, które z nich podasz i dlaczego?

Skorowidz

A

- abstrakcja, 19
- adaptowanie, 143
 - dostosowywanie interfejsu, 144
 - konfigurowanie działania, 144
 - wykrywanie, 143
- ADD, Attribute-Driven Design, 360
- aktory, 271, 279, 283
- aktualizacja
 - oprogramowania, 88
- aktualizacje bez przerywania pracy, 88
- aktuator, 330
- alokacja zasobów, 127
- analiza
 - decyzji projektowych, 74
 - kompromisów architektonicznych, 387
- antywzorce, 442, 445
- architekt, 37, 453
 - monitorowanie źródła zasilania, 326
 - obowiązki, 468–470
 - projektowanie komunikacji, 329
 - rejestrwanie zdarzeń, 338
 - rekomendacje, 456
 - rozwijanie kompetencji, 475
 - testy urządzenia mobilnego, 335
 - umiejętności, 471, 472
 - wdrażanie aktualizacji, 337
 - wiedza, 472, 473
 - wybór sprzętu, 335
 - zadania, 454
 - zadania związane z czujnikami, 331
 - zarządzanie zasobami, 333
- architektura, 17
 - bezszyfrowana, 301, 303
 - korporacyjna, 21
 - prognozowanie, 46
 - systemu, 21
- atrakcyjność rynkowa, 258
- atrzybuty jakościowe, AJ, 63
 - analizowanie decyzji projektowych, 74
 - architektury, 258
 - atrakcyjność rynkowa, 258
 - bezpieczeństwo, 194
 - dostępność, 78
 - efektywność energetyczna, 121
 - funkcjonalność, 64
 - integralność koncepcyjna, 258
 - integrowalność, 134
 - łatwość tworzenia systemu, 258
 - łatwość wdrażania, 101
 - łączenie podejść projektowych, 265
 - modelowanie, 264
 - modyfikowalność, 152
 - prognozowanie, 46
 - scenariusze, 66–70
 - specyfikacja wymagań, 67
 - standardowe listy, 260
 - systemu, 44, 259
 - taktyki architektoniczne, 71
 - testowalność, 230
 - tworzenie scenariuszy, 264
 - użyteczność, 246
 - wydajność, 171
 - wzorce, 71
 - zabezpieczenia, 215
- automatyzacja analizy architektonicznej, 447
- autoskaler, 321
- autoskalowanie, 319
 - kontenerów, 322
 - maszyn wirtualnych, 320

B

BDUF, Big Design Up Front, 457
 bezpieczeństwo, 194
 kwestionariusz, 205
 poziomy, 210
 scenariusz, 197
 taktyki, 199
 wzorze, 208
 brama, 277
 komunikatów, 308
 zarządzania chmurą, 308, 309

C

cechy perspektywy
 alokacji, 416
 komponenty-złącza, 415
 modułów, 412
 cele biznesowe, 351
 chmura
 awarie, 309
 dostępność, 312
 limity czasu, 310
 wydajność, 312
 chmury
 hybrydowe, 306
 obliczeniowe, 305
 prywatne, 306
 publiczne, 306
 ciągle
 dostarczanie, 102
 wdrażanie, 102
 cykl życia systemów
 mobilnych, 335
 czas
 blokady, 177
 przetwarzania, 177
 referencyjny, 317
 czujnik, 330

D

decyzje projektowe, 50
 DevOps, 105
 diagram
 aktywności, 424
 kontekstu, 426
 maszyny stanowej, 425
 sekwencji, 422
 długi
 architektoniczny, 438
 ciągłe
 monitorowanie, 447
 identyfikowanie, 439, 442
 ocena ilościowa, 446
 techniczny, 442
 długi ogon, 311
 dodatkowe żądania, 312
 nadmiarowe żądania, 312
 długość cyklu, 104
 dokumentacja
 architektury, 406
 diagramy kontekstu, 426
 dokumentowanie dynamicznie zmieniających się architektur, 434
 interfejsu, 285
 uzasadnienia, 426, 427
 wstępne, 375, 376
 wymagań, 346
 wzorców, 426
 zachowań, 420
 lista akronimów, 427
 łączenie perspektyw, 418
 możliwość śledzenia, 435

 notacje, 408
 odbiorcy, 407
 odzworowania między perspektywami, 424
 online, 433
 perspektywy, 409
 słowniczek, 427
 zastosowania, 407
 dostępność, 78, 215
 kwestionariusz, 92, 94
 scenariusz, 81
 taktyki zapewniania, 82
 wymagania, 80
 wzorze, 95
 zasobów, 178
 drzewo użyteczności, 355
 DSM, design structure matrix, 439

E

efektywność
 energetyczna, 121
 kwestionariusz, 128
 scenariusz, 123
 taktyki, 125
 wzorze, 130
 energia
 monitorowanie źródła zasilania, 326
 ograniczanie zużycia, 327
 ewaluacja architektury, 382
 czynniki kontekstowe, 386
 główne aktywności, 383
 metoda
 ATAM, 387–391
 LAE, 401
 ewaluatorzy, 384, 385

F

FIFO, first-in, first-out, 183
 format
 JSON, 282
 XML, 282
 funkcjonalność, 64

G

głosowanie, 85
 gwarantowany poziom
 świadczonych usług, 80

H

harmonogram prac, 53
 hermetyzacja, 139
 hipernadzorca, 293, 309
 hostowany, 292
 natywny, 292
 hotspoty, 442
 identyfikowanie, 446

I

IaaS, infrastructure-
 -as-a-service, 306
 identyfikowanie
 długu
 architektonicznego,
 439
 hotspotów, 442, 446
 ignorowanie
 nieprawidłowej pracy, 88
 implementacja
 ograniczenia, 51
 integralność, 215
 konceptyjna, 258
 integrowalność, 134
 kwestionariusz, 146
 ocena, 135
 scenariusz, 137
 taktyki, 139
 wzorce, 147

interesariusze, 47
 architektury, 428
 interfejsy, 271
 dokumentacja, 285
 elementy, 271, 286
 ewolucja, 274
 grupa interfejsów, 273
 modyfikowanie, 274
 rozszerzanie, 275
 wersjonowanie, 275
 wycofywanie, 275
 obsługa błędów, 283
 projektowanie, 275
 style interakcji, 278
 zakres, 277
 zasady projektowe, 276
 zasoby, 271, 273

J

jakość
 potoku wdrażania,
 104
 produktu
 efektywność
 działania, 260
 kompatybilność, 260
 łatwość konserwacji,
 262
 niezawodność, 262
 przeñośność, 262
 przydatność
 funkcjonalna, 260
 użyteczność, 262
 zabezpieczenia, 262
 JSON, JavaScript Object
 Notation, 282

K

kanban, 379
 klasyfikacja
 dynamiczna, 126
 statyczna, 126
 klika, 443

kohezja, 158, 159
 kolejka FIFO, 183
 kolejkowanie, 265
 komunikacja między
 interesariuszami, 47
 kontenery, 296–300, 303
 przeñośność, 300
 kontrolowana degradacja,
 88
 kontrolowanie
 i obserwowanie stanu
 systemu
 abstrakcyjne źródła
 danych, 237
 izolowane środowisko,
 238
 rejestrowanie
 i odtwarzanie, 237
 specjalne interfejsy,
 236
 wykonywalne asercje,
 238
 zapisywanie stanu,
 237
 koordynacja, 145
 orkiestracja, 145
 zarządzanie zasobami,
 145
 koordynowanie
 czasu, 317
 danych, 318
 koszty
 szacowanie, 53
 kubit, 482
 kwestionariusz, 74
 dostępności, 92, 93, 94
 efektywności
 energetycznej, 128
 integrowalności, 146
 łatwości wdrażania, 112
 modyfikowalności,
 161
 na temat zabezpieczeń,
 205, 224
 wydajności, 186

kwestionariusz
zapewniania
testowalności, 241
zapewniania
użyteczności, 251
kwestionariusze oparte
na taktykach, 403

L

LAE, Lightweight
Architecture
Evaluation, 401
limit czasu, 86, 310
lista możliwości, 56
listy AJ, 260

Ł

łatwość
różnicowania, 155
tworzenia systemu,
258
wdrażania, 101, 106
kwestionariusz, 112
scenariusz, 107
taktyki, 109
wzorce, 113
łączenie
elementów, 55
perspektyw, 418
struktur, 33
łączność sieciowa
projektowanie, 329

M

macierz
odwracanie, 487
struktury projektu,
DSM, 439, 441
maszyny wirtualne, 289,
291, 299
kontenery, 296
obrazy, 295
menedżer projektu, 453

metoda
ATAM, 387
dane wyjściowe, 388
fazy metody, 390
interesariusze
architektury, 388
kroki faz ewaluacji,
391
osoby decyzyjne,
387
zespół ewaluacyjny,
387
LAE, 401
plan sesji, 402
metodyka DevOps, 105
mikrojądło, 164
mikrouslugi, 373
model
danych, 28, 31
kaskadowy, BDUF,
457
kolejkowania, 265
predyktywny, 91
systemu, 251
testów, 231
użytkownika, 251
wielokrotnego użytku,
54
zadania, 251
modelowanie atrybutu
jakościowego, 264
moduł typu sensor hub,
330
modyfikowalność, 45, 152
kwestionariusz, 161
scenariusz, 156
taktyki, 157
wzorce, 163
monitor, 84
monitorowanie
warunków, 85
zasobów, 125
możliwość śledzenia, 105
MTBF, mean time
between failures, 79

MTTR, mean time
to repair, 79

N

narzędzia do
modelowania, 433
niezależność lokalizacji,
155
NTP, Network Time
Protocol, 317

O

obsługa wyjątków, 87
odgradzanie parametrów,
86
odległość
czasowa, 137
dotycząca zasobów,
137
semantyki danych, 136
semantyki działania,
136
składniowa, 136
odraczanie wiązania, 160
odwracanie macierzy, 487
odzyskiwanie stanu, 204
po atakach, 223
audyt, 223
niezaprzeczalność,
223
po usterkach, 87
ograniczenie
czasu przetwarzania,
180
kosztów
obliczeniowych, 180
reakcji na zdarzenia,
179
szkód
bariery, 204
ograniczenie
skutków, 204
redundancja, 203

- zależności
 - hermetyzacja, 139
 - ograniczanie ścieżek komunikacji, 141
 - stosowanie się do standardów, 142
 - tworzenie abstrakcji wspólnych usług, 142
 - używanie
 - pośrednika, 141
 - złożoności, 239
 - ograniczanie
 - niedeterminizmu, 240
 - strukturalnej, 239
 - zużycia, 127
 - operacje na kubitach, 483
 - oprogramowanie
 - otwartoźródłowe, 55
 - organizacje
 - kompetencje
 - architektoniczne, 474
 - rozwijające
 - architekturę, 473
- P**
- pamięć QRAM, 486
 - perspektywy, 37, 409
 - alokacji, 416
 - architektury, 22
 - perspektywy
 - atrybutów
 - jakościowych, 417
 - komponenty-złącza, 413
 - łączenie, 418
 - modułów, 411, 412
 - piaskownica, 238
 - ping/echo, 84
 - podstawowe sprawdzanie
 - poprawności, 85
 - pody, 300
 - podział rozwoju
 - oprogramowania, 259
 - polecenia HTTP, 280
 - ponawianie prób, 88
 - poprawki
 - funkcji, 88
 - klas, 88
 - potok wdrażania, 102
 - pomiar jakości, 104
 - zarządzanie, 110
 - poufność, 215
 - powiązanie, 158
 - powtarzalność, 105
 - proces projektowania
 - architektury, 361
 - prognozowanie
 - atrybutów
 - jakościowych systemu, 46
 - programowanie
 - rozproszone, 461
 - zwinne, 456
 - zasady, 459–460
 - projekt, 20
 - Simian Army, 231
 - projektowanie
 - architektury
 - iteracja 0, 458
 - model emergentny, 457
 - model kaskadowy, 457
 - interfejsu, 275
 - łączności sieciowej, 329
 - oparte na atrybutach,
 - ADD, 360
 - analiza projektu, 366, 377
 - diagram
 - kontekstowy systemu, 362
 - kroki i artefakty, 363
 - obraz procesu, 361
 - ocena celu projektu, 377
 - ocena realizacji
 - celów iteracji, 377
 - określenie celów
 - iteracji, 364
 - perspektywa, 366
 - przegląd celów
 - iteracji, 366
 - przegląd danych
 - wejściowych, 362
 - przydział zadań, 365
 - realizacja celu
 - projektu, 366
 - tworzenie
 - perspektyw, 366
 - tworzenie struktur, 371
 - wstępna
 - dokumentacja, 375
 - wybór elementów
 - systemu, 364
 - wybór koncepcji
 - projektowych, 365, 367
 - zapisanie decyzji, 366
 - zdefiniowanie
 - interfejsów, 365
 - Protocol Buffers, 282
 - protokół NTP, 317
 - prototypy, 369, 370
 - prywatność, 216
 - przeciwdziałanie atakom, 221
 - autoryzacja jednostek, 221
 - identyfikowanie
 - jednostek, 221
 - ograniczanie dostępu, 221
 - ograniczanie
 - narażenia na atak, 222

przeciwdziałanie atakom,
 rozdzielanie
 jednostek, 222
 szyfrowanie danych,
 222
 uwierzytelnianie
 jednostek, 221
 walidacja danych
 wejściowych, 222
 zmiana ustawień
 związanych
 z uwierzytelnianiem,
 222
 przełączanie funkcji, 111
 przenośność, 155
 przetwarzanie
 kwantowe, 481, 485
 rozproszone, 305, 313
 przetwornik, 330
 przydział zadań, 373
 przypadki użycia, 421
 przywracanie, 89

Q

QRAM, quantum
 random access memory,
 486

R

reakcja na ataki, 223
 informowanie
 jednostek, 223
 odebranie dostępu,
 223
 ograniczanie
 możliwości
 logowania, 223
 redundancja
 analityczna, 86
 funkcjonalna, 85
 redundantny komponent
 rezerwowowy, 87
 refaktoryzacja, 72

reguły dotyczące
 struktury, 38
 rekonfiguracja, 89
 replikacja, 85
 reprezentacja danych,
 280
 REST, Representational
 State Transfer, 279
 ograniczenia, 279
 restart z eskalacją, 89
 resynchronizacja stanu,
 89
 rozmowy
 z interesariuszami, 349
 rozwijanie przyrostowe,
 455
 równoważniki
 obciążenia, 313, 314
 rywalizacja o zasoby, 177

S

samodiagnostyka, 87
 scenariusz dotyczący
 atrybutów
 jakościowych, 67
 artefakt, 68
 bodziec, 67
 miara reakcji, 68
 reakcja, 68
 środowisko, 68
 źródło bodźca, 68
 bezpieczeństwa, 197
 dostępności, 81
 efektywności
 energetycznej, 123
 integrowalności, 137
 łatwości wdrażania,
 107
 modyfikowalności,
 156
 testowalności, 233
 użyteczności, 247
 wydajności, 173–175
 zabezpieczeń, 217
 skalowalność, 154
 skalowanie
 udostępniania, 110
 skrypty wdrażania, 111
 SLA, service level
 agreement, 80
 specyfikacja wymagań, 67
 splątanie, 484
 stałe przekazywanie, 90
 stan, 316
 stopniowa
 implementacja, 38
 stopniowe rozwijanie
 systemu, 53
 stos narzędzi LAMP, 298
 strategię szeregowania,
 182
 struktura
 alokacji, 26, 31, 39,
 371
 architektury, 22, 33,
 34
 dekompozycji, 26, 27
 dekompozycji
 modułu, 455
 implementacji, 32
 klas, 28, 30
 komponenty-złącza,
 23, 35, 29, 39, 371
 modułu, 24, 26, 39,
 371, 455
 oprogramowania, 18
 organizacji, 52
 przesyłanych danych,
 280
 przydziału prac, 32
 usług, 30
 używania, 26
 warstwowa, 27, 29
 wdrażania, 31, 32, 455
 style
 interakcji
 model REST, 279
 zdalne wywołania
 procedur, 279

reprezentacji danych
 JSON, 282
 Protocol Buffers, 282
 XML, 281
 supertaktyki, 73
 sygnały kontrolne, 84
 system
 mobilny, 325
 aktuatory, 330
 cykl życia, 335
 czujniki, 330
 łączność sieciowa, 328
 zarządzanie energią, 326
 zasoby, 332
 oprogramowania, 20
 rozproszony
 autoskalowanie, 319
 koordynowanie czasu, 317
 koordynowanie danych, 318
 zarządzanie stanem, 316
 szacowanie kosztów, 53
 szeregowanie
 statyczne, 184
 zasobów, 128
 ze stałymi
 priorytetami, 183
 ze zmiennymi
 priorytetami, 184
 szkolenia, 57
 szyfrowanie, 485

Ś

środowisko
 integracyjne, 103
 produkcyjne, 103
 rozwojowe, 102
 testowe, 103

T

tablica kanban, 379
 taktyki architektoniczne, 71, 72, 76
 zabezpieczenia
 systemu, 219
 odzyskiwanie stanu po atakach, 223
 przeciwdziałanie atakom, 221
 reakcja na ataki, 223
 wykrywanie ataków, 219
 zapewniania
 bezpieczeństwa, 199
 odzyskiwanie stanu, 204
 ograniczanie szkód, 203
 unikanie niebezpiecznego stanu, 200
 wykrywanie niebezpiecznego stanu, 201
 zapewniania
 dostępności, 82
 odzyskiwanie stanu po usterkach, 87
 wykrywanie usterek, 84
 zapobieganie usterkom, 90
 zapewniania
 efektywności energetycznej, 125
 alokacja zasobów, 127
 monitorowanie zasobów, 125
 zmniejszanie zapotrzebowania na zasoby, 128

zapewniania
 integrowalności, 139
 adaptowanie, 143
 koordynacja, 145
 ograniczanie zależności, 139
 wykrywanie dynamiczne, 150
 zapewniania łatwości
 wdrażania, 109
 zarządzanie potokami wdrażania, 110
 zarządzanie wdrożonym systemem, 111
 zapewniania
 modyfikowalności, 157
 odraczanie wiązania, 160
 zmniejszanie powiązania, 160
 zwiększanie kohezji, 159
 zapewniania
 testowalności, 235
 kontrolowanie i obserwowanie stanu systemu, 236
 ograniczanie złożoności, 239
 zapewniania
 użyteczności, 248
 wspomaganie działań systemu, 250
 wspomaganie działań użytkownika, 249
 zapewniania
 wydajności, 176, 183
 na trasie, 185

taktyki architektoniczne,
zarządzanie
zapotrzebowaniem
na zasoby, 178
zarządzanie
zasobami, 181
zarządzania reakcją,
73

teleportacja kwantowa,
484

testowalność, 106, 230
kwestionariusz, 241
scenariusz, 233
taktyki, 235
techniki zastępowania
komponentu, 239
wzorce, 242

testy
A/B, 119
kanarkowe, 118

transakcje, 90

tworzenie
instancji elementów,
372
perspektywy, 366
prototypów, 369
struktur, 371
wstępnej
dokumentacji, 375
typizacja parametrów, 86

U

umieszczanie zależności
w pakiecie, 111

UML
diagram aktywności,
422
diagram maszyn
stanowych, 423
diagram sekwencji,
421
elementy modułów,
25
relacje, 25

umowa SLA, 80

unikanie
niebezpiecznego stanu,
200
model predyktywny,
201
zastępowanie, 200

usługi
bezstanowe, 316
stanowe, 316

użyteczność, 246
kwestionariusz, 251
scenariusz, 247
taktyki, 248
wzorce, 253

W

warsztat atrybutów
jakościowych, 349

wdrożenie, 101
kwestionariusz, 112
łatwość, 106
pomiar jakości, 104
potok, 102, 110
scenariusz, 108
taktyki, 109
wzorce, 113

wdrażanie
ciągłe, 102
kontrolowalne, 107
szczegółowe, 107
wydajne, 107

węzeł z podami, 301

wielkość modułu, 158

wirtualizacja, 104

wspomaganie działań
systemu, 250
utrzymywanie
modelu systemu, 251
modelu
użytkownika, 251
modelu zadania, 251

wspomaganie działań
użytkownika, 249

agregowanie, 250
anulowanie, 249
wstrzymywanie
i wznowianie, 250
wycofywanie, 250

współbieżność, 173

współużytkowanie
procesora, 290
przestrzeni dyskowej,
291
systemu
operacyjnego, 298
zasobów, 289, 290

wycofanie, 87, 110
z eksploatacji, 90

wydajność, 171
kwestionariusz, 186
scenariusz, 173–175
taktyki, 176
wzorce, 187

wyjątki systemowe, 86

wykorzystanie zasobów,
177

wykrywanie, 127
ataków, 219
anomalia, 220
nieдоступność
usługi, 220
weryfikowanie
integralności
komunikatów, 220
włamania, 219

dynamiczne, 150

niebezpiecznego
stanu
monitorowanie
warunków, 202
podstawowe
sprawdzanie
poprawności, 202
porównania, 202
znaczniki czasu, 202

usterek, 84

wyjątków, 86

- wymagania
 - funkcjonalne, 65
 - o znaczeniu
 - architektonicznym
 - cele biznesowe, 351
 - dokumentacja, 346
 - drzewo
 - użyteczności, 354
 - rozmowy
 - z interesariuszami, 348
 - wzorce architektoniczne, 36, 71
 - bezpieczeństwa, 208
 - monitor-aktuator, 208
 - podział systemu
 - ze względu na bezpieczeństwo, 209
 - redundantne
 - czujniki, 208
 - dostępności, 95, 98
 - aktywna pasywna, 95
 - aktywna
 - redundancja, 95
 - bezpiecznik, 97
 - komponent
 - rezerwowo, 96
 - omijanie błędów, 98
 - pary procesów, 98
 - potrójna
 - redundancja
 - modularna, 96
 - efektywności
 - energetycznej
 - monitor zużycia energii, 132
 - synteza danych
 - zamykanie
 - niezwykłych zadań, 131
 - integrowalności, 147
 - architektura oparta
 - na usługach, 149
 - mediator, 148
 - most, 148
 - nakładka, 147
 - łatwości wdrażania
 - aktualizacja po jednym węźle, 115
 - architektura
 - mikrousługowa, 113
 - niebieskie-zielone, 115
 - strukturyzowania
 - usług, 113
 - testowalności, 242
 - filtry z przechwytywaniem, 244
 - strategia, 243
 - wstrzykiwanie
 - zależności, 242
 - TMR, 97
 - użyteczności, 253
 - model-widok-kontroler, 253
 - obserwator, 254
 - pamiętka, 255
 - wydajności, 187
 - map-reduce, 190
 - ograniczanie
 - przepustowości, 189
 - równoważnik
 - obciążenia, 188
 - siatka usług, 187
 - zabezpieczenia
 - systemu, 226
 - system zapobiegania włamaniom, 227
 - walidator
 - z przechwytywaniem, 226
 - zapewniania
 - modyfikowalności, 163
 - klient-serwer, 163
 - publikuj-
 - subskrybuj, 166
 - warstwy, 165
 - wtyczka, 164
 - zastępowania usług
 - całkowitego, 114
 - częściowego, 117
- ## X
- XML, eXtensible Markup Language, 281
- ## Z
- zabezpieczenia, 215
 - dostępność, 215
 - integralność, 215
 - kwestionariusz, 224
 - poufność, 215
 - prywatność, 216
 - scenariusz, 217
 - taktyki, 219
 - wzorce, 226
 - zależność
 - ewolucyjna, 439
 - od innych obliczeń, 178
 - zapobieganie
 - usterkom, 90
 - wyjątkom, 91
 - zarządzanie
 - częstością
 - próbkowania, 73, 179
 - długim
 - architektonicznym, 438
 - interakcjami między usługami, 111
 - pojawianiem się zdarzeń, 178
 - potokami wdrażania, 110

- zarządzanie
 - reakcją na bodziec, 73
 - stanem, 316
 - wdrożonym systemem, 111
 - zapotrzebowaniem
 - na zasoby, 178
 - ograniczanie reakcji
 - na zdarzenia, 179
 - ograniczenie czasu przetwarzania, 180
 - ograniczenie kosztów obliczeniowych, 180
 - określanie priorytetów zdarzeń, 179
- zwiększanie efektywności, 181
- zasobami, 181
- zmianami, 45
- żądaniami wykonania zadań, 178
- zasoby, 178
- interfejsów
 - operacje, 274
 - semantyka, 273
 - składnia, 273
 - właściwości, 274
 - zdarzenia, 274
- sprzętowe, 177
- systemu mobilnego
 - bezpieczeństwo, 333
- ograniczenia termiczne, 333
- problemy środowiskowe, 333
- zarządzanie, 181
- zmniejszanie powiązania, 160
- ograniczanie zależności, 160
- zapotrzebowania na zasoby, 128
- znaczniki czasu, 84
- zwiększanie kohezji, 159
- podział modułu, 159
- przenoszenie zadań, 159
- kompetencji, 91

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 

Wydawać by się mogło, że gruntowna wiedza o architekturze oprogramowania jest zbędna: istnieją infrastruktury chmurowe, mikrousługi, wzorce projektowe, po które wystarczy tylko sięgnąć. Niestety, to nie jest prawda. Konieczność ciągłego dostosowywania i rozbudowywania oprogramowania, a także rosnąca złożoność systemów sprawiają, że wiedza architekta wciąż jest niezbędna. Jeśli architekt nie wykona swojej pracy dobrze, gotowy system będzie trudny do zrozumienia, modyfikowania i debugowania. Stanie się ciężarem dla firmy.

Ta książka to kompleksowy przewodnik po tworzeniu architektury nowoczesnego oprogramowania. Pomoże Ci zrozumieć, czym jest architektura oprogramowania i jak nią zarządzać w zdyscyplinowany i skuteczny sposób. Poznasz zasady przekształcania architektury w celu rozwiązywania nowych problemów i tworzenia architektury wielokrotnego użytku. Otrzymasz praktyczne wskazówki, zapoznasz się też z metodami stosowanymi przez ekspertów i ze sprawdzonymi modelami. Nauczysz się radzić sobie z coraz większymi wymaganiami i coraz wyższym poziomem abstrakcji dynamicznie zmieniających się systemów. Znajdziesz tu również informacje na temat optymalizowania za pomocą architektury najważniejszych atrybutów jakościowych systemów.

Najciekawsze zagadnienia:

- wpływ architektury na środowisko techniczne, cykle życia projektów i profile biznesowe
- optymalizacja jakości za pomocą architektury
- architektura dla rozwiązań mobilnych, chmury, uczenia maszynowego i przetwarzania kwantowego
- tworzenie architektury uwzględniającej wydajność energetyczną i bezpieczeństwo
- zarządzanie długiem architektonicznym
- właściwa architektura a rozwój organizacji

LEN BASS jest wykładowcą i autorem uznanych książek. Od ponad 50 lat zajmuje się rozwojem oprogramowania. Obecnie wykłada podejście DevOps na uczelni Carnegie Mellon.

DR PAUL CLEMENTS zajmuje się inżynierią linii produktów systemowych. Pracował nad wbudowanymi systemami czasu rzeczywistego w U.S. Naval Research Laboratory w Waszyngtonie. Autor blisko 100 prac z dziedziny inżynierii oprogramowania.

RICK KAZMAN jest badaczem. Zajmuje się architekturą oprogramowania i ekonomią inżynierii oprogramowania. Współtworzył cenione metody i narzędzia służące do analizowania architektury. Jest przewodniczącym IEEE TAC i członkiem ICSE Steering Committee.

” Upewnij się, że znasz budowę wszystkiego, co chcesz przedstawić

Leonardo da Vinci

	KOD KORZYŚCI Sięgnij po więcej! ▶	
 helion.pl	ISBN 978-83-283-9052-2	
 HELION SA ul. Kościuszki 1c 44-100 Gliwice tel.: 32 250 98 63 helion@helion.pl	 9 788328 390522	
Cena: 99,00 zł		