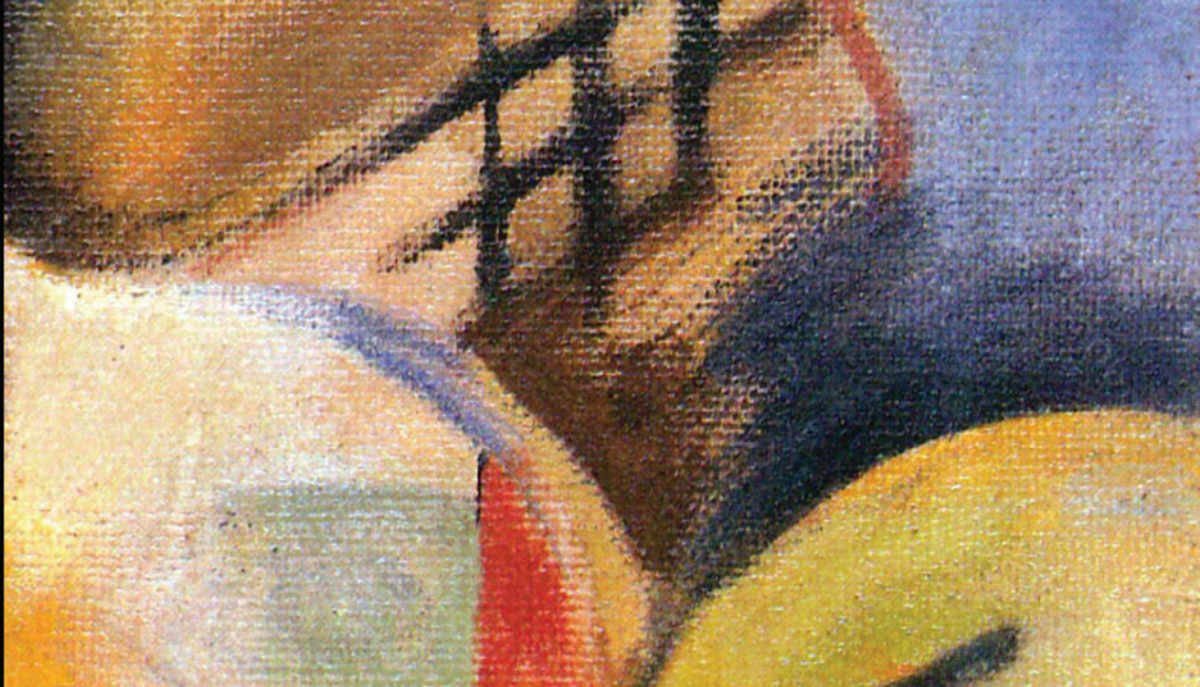




SYSTEMY REAKTYWNE

WZORCE PROJEKTOWE I ICH STOSOWANIE

DR ROLAND KUHN, BRIAN HANAFEE, JAMIE ALLEN



 MANNING

Helion 

Tytuł oryginału: Reactive Design Patterns

Tłumaczenie: Andrzej Watrak

ISBN: 978-83-283-3795-4

Original edition copyright © 2017 by Manning Publications Co.
All rights reserved.

Polish edition copyright © 2018 by HELION SA
All rights reserved.

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

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

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

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

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

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

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
<ftp://ftp.helion.pl/przyklady/sysrea.zip>

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści

<i>Przedmowa</i>	13
<i>Wstęp</i>	15
<i>Podziękowania</i>	17
<i>O książce</i>	19
<i>O autorach</i>	21

CZĘŚĆ I WPROWADZENIE 23

Rozdział 1. System reaktywny? 25

1.1.	Anatomia systemu reaktywnego	26
1.2.	Problem obciążenia	28
1.3.	Problem awarii	29
1.4.	Tworzenie responsywnego systemu	30
1.5.	Zapobieganie efektowi kuli błota	32
1.6.	Integrowanie komponentów niereaktywnych	33
1.7.	Podsumowanie	34

Rozdział 2. Przegląd Manifestu reaktywnego 35

2.1.	Reagowanie systemu na interakcje użytkowników	35
2.1.1.	Podjęcie tradycyjne	36
2.1.2.	Analiza czasu odpowiedzi współdzielonego zasobu	38
2.1.3.	Ograniczanie maksymalnego czasu odpowiedzi za pomocą kolejki	39
2.2.	Analiza paralelizmu	41
2.2.1.	Skrócenie czasu odpowiedzi z wykorzystaniem paralelizmu	41
2.2.2.	Usprawnianie paralelizmu za pomocą komponowanych futur	43
2.2.3.	Cena szeregowej iluzji	44
2.3.	Ograniczenia paralelizmu	46
2.3.1.	Prawo Amdahla	46
2.3.2.	Uniwersalne prawo skalowalności	47
2.4.	Obsługa awarii	48
2.4.1.	Rozczłonkowanie i gradzenie	50
2.4.2.	Bezpieczniki	51
2.4.3.	Nadzorowanie usług	53

- 2.5. Utrata wysokiej spójności danych 54
 - 2.5.1. *ACID 2.0* 56
 - 2.5.2. *Odbieranie zmian* 57
- 2.6. Wzorce projektowania reaktywnego 58
 - 2.6.1. *Zarządzanie złożonością oprogramowania* 59
 - 2.6.2. *Przystosowanie modeli programistycznych do rzeczywistości* 60
- 2.7. Podsumowanie 61

Rozdział 3. Narzędzia 63

- 3.1. Pierwsze rozwiązania reaktywne 63
- 3.2. Programowanie funkcyjne 65
 - 3.2.1. *Niezmiennosc* 66
 - 3.2.2. *Przejrzystosc referencyjna* 68
 - 3.2.3. *Efekty uboczne* 69
 - 3.2.4. *Funkcje pierwszej klasy* 70
- 3.3. Responsywnosc dla użytkowników 70
 - 3.3.1. *Ustalanie priorytetow cech wydajnosciowych* 71
- 3.4. Dostepne narzędzia reaktywne 72
 - 3.4.1. *Zielone wgtki* 72
 - 3.4.2. *Petle zdarzen* 73
 - 3.4.3. *Język CSP* 74
 - 3.4.4. *Futury i promesy* 76
 - 3.4.5. *Rozszerzenia reaktywne* 80
 - 3.4.6. *Model Aktor* 82
- 3.5. Podsumowanie 86

CZĘŚĆ II FILOZOFIA REAKTYWNOŚCI W PIGUŁCE 87

Rozdział 4. Przesyłanie komunikatów 89

- 4.1. Komunikaty 89
- 4.2. Pionowa skalowalność aplikacji 90
- 4.3. Sterowanie zdarzeniami i komunikatami 91
- 4.4. Synchroniczne i asynchroniczne przesyłanie komunikatów 93
- 4.5. Sterowanie przepływem danych 95
- 4.6. Gwarancja dostarczania komunikatów 97
- 4.7. Zdarzenia jako komunikaty 100
- 4.8. Synchroniczne przesyłanie komunikatów 101
- 4.9. Podsumowanie 102

Rozdział 5. Przezroczystosc lokalizacji 103

- 5.1. Czym jest przezroczystosc lokalizacji? 103
- 5.2. Bledne wyobrazenia o przezroczystosci wywołan 104
- 5.3. Ratunek w jawnym przesyłaniu komunikatów 105
- 5.4. Optymalizacja lokalnego przekazywania komunikatów 107
- 5.5. Utraty komunikatów 107
- 5.6. Pozioma skalowalność aplikacji 109

- 5.7. Przezroczystość lokalizacji upraszcza testy 110
- 5.8. Dynamiczne komponowanie systemu 111
- 5.9. Podsumowanie 112

Rozdział 6. Dziel i rządź 113

- 6.1. Hierarchiczna struktura problemu 114
 - 6.1.1. Tworzenie hierarchii modułów 114
- 6.2. Zależności i moduły pochodne 115
 - 6.2.1. Zapobieganie powstawaniu matryc 116
- 6.3. Budowanie własnej ogromnej korporacji 118
- 6.4. Zalety precyzyjnej specyfikacji i testów 119
- 6.5. Skalowalność pozioma i pionowa aplikacji 120
- 6.6. Podsumowanie 121

Rozdział 7. Strukturalna obsługa awarii 123

- 7.1. Własność oznacza zobowiązanie 123
- 7.2. Własność określa kontrolę cyklu życia modułu 125
- 7.3. Odporność na awarie na wszystkich poziomach 127
- 7.4. Podsumowanie 127

Rozdział 8. Rozdzielona spójność danych 129

- 8.1. Ratunek w niezależnych modułach 130
- 8.2. Grupowanie danych i transakcji 131
- 8.3. Modelowanie przepływów danych ponad granicami transakcyjnymi 131
- 8.4. Jednostka awaryjności = jednostka spójności 133
- 8.5. Segregacja odpowiedzialności 133
- 8.6. Utrzymywanie odizolowanych zakresów spójności 135
- 8.7. Podsumowanie 136

Rozdział 9. Niedeterminizm na życzenie 137

- 9.1. Programowanie logiczne i deklaratywny przepływ danych 137
- 9.2. Reaktywne programowanie funkcyjne 139
- 9.3. Współdzielenie niczego upraszcza równoległość operacji 140
- 9.4. Współdzielenie stanu i równoległość 141
- 9.5. Co zatem powinniśmy robić? 141
- 9.6. Podsumowanie 143

Rozdział 10. Przepływ danych 145

- 10.1. Wysyłanie danych 145
- 10.2. Modelowanie procesów w domenie 147
- 10.3. Określanie ograniczeń odporności na awarie 147
- 10.4. Szacowanie ilości komunikatów i skali wdrożenia 148
- 10.5. Planowanie sterowania przepływami 149
- 10.6. Podsumowanie 149

CZĘŚĆ III WZORCE 151**Rozdział 11. Testy aplikacji reaktywnych 153**

- 11.1. Jak testować aplikacje? 153
 - 11.1.1. Testy jednostkowe 154
 - 11.1.2. Testy komponentów 155
 - 11.1.3. Testy łańcuchowe 155
 - 11.1.4. Testy integracyjne 155
 - 11.1.5. Testy akceptacyjne 156
 - 11.1.6. Testy czarnej i białej skrzynki 156
- 11.2. Środowisko testowe 157
- 11.3. Testy asynchroniczne 158
 - 11.3.1. Blokujące odbiorniki komunikatów 159
 - 11.3.2. Sztuka doboru czasu oczekiwania 161
 - 11.3.3. Wykrywanie braku komunikatów 167
 - 11.3.4. Tworzenie synchronicznych kodów wykonawczych 168
 - 11.3.5. Asercje asynchroniczne 170
 - 11.3.6. Testy w pełni asynchroniczne 170
 - 11.3.7. Wykrywanie braku błędów asynchronicznych 173
- 11.4. Testowanie systemów niedeterministycznych 176
 - 11.4.1. Problem z planowaniem wykonywania testów 176
 - 11.4.2. Testowanie komponentów rozproszonych 176
 - 11.4.3. Aktorzy imitacji 177
 - 11.4.4. Komponenty rozproszone 179
- 11.5. Testowanie elastyczności systemu 179
- 11.6. Testowanie sprzężystości systemu 179
 - 11.6.1. Sprężystość aplikacji 180
 - 11.6.2. Sprężystość infrastruktury 183
- 11.7. Testowanie responsywności systemu 185
- 11.8. Podsumowanie 186

Rozdział 12. Wzorce uodporniania na awarie i odtwarzania systemu 187

- 12.1. Wzorec Prosty Komponent 187
 - 12.1.1. Opis problemu 188
 - 12.1.2. Stosowanie wzorca 188
 - 12.1.3. Weryfikacja wzorca 190
 - 12.1.4. Kiedy stosować wzorec? 191
- 12.2. Wzorec Jądro Błędu 191
 - 12.2.1. Opis problemu 191
 - 12.2.2. Stosowanie wzorca 192
 - 12.2.3. Weryfikacja wzorca 195
 - 12.2.4. Kiedy stosować wzorec? 196
- 12.3. Wzorec Pozwól Na Awarię 196
 - 12.3.1. Opis problemu 197
 - 12.3.2. Stosowanie wzorca 197
 - 12.3.3. Weryfikacja wzorca 198
 - 12.3.4. Zagadnienia implementacyjne 199

12.3.5.	<i>Konkluzja: wzorzec Bicie Serca</i>	200
12.3.6.	<i>Konkluzja: wzorzec Proaktywny Sygnał Awaryjny</i>	201
12.4.	<i>Wzorzec Bezpiecznik</i>	202
12.4.1.	<i>Opis problemu</i>	202
12.4.2.	<i>Stosowanie wzorca</i>	203
12.4.3.	<i>Weryfikacja wzorca</i>	206
12.4.4.	<i>Kiedy stosować wzorzec?</i>	207
12.5.	<i>Podsumowanie</i>	207
Rozdział 13. Wzorce replikacyjne 209		
13.1.	<i>Wzorzec Replikacja Aktywna-Pasywna</i>	209
13.1.1.	<i>Opis problemu</i>	210
13.1.2.	<i>Stosowanie wzorca</i>	211
13.1.3.	<i>Weryfikacja wzorca</i>	220
13.1.4.	<i>Kiedy stosować wzorzec?</i>	221
13.2.	<i>Wzorzec Replikacja Wielokrotna-Główna</i>	221
13.2.1.	<i>Replikacja oparta na konsensusie</i>	222
13.2.2.	<i>Replikacja z wykrywaniem i rozwiązywaniem konfliktów</i>	225
13.2.3.	<i>Bezkonfliktowe typy replikowanych danych</i>	226
13.3.	<i>Wzorzec Replikacja Aktywna-Aktywna</i>	233
13.3.1.	<i>Opis problemu</i>	234
13.3.2.	<i>Stosowanie wzorca</i>	235
13.3.3.	<i>Weryfikacja wzorca</i>	240
13.3.4.	<i>Odniesienie do wirtualnej synchroniczności</i>	241
13.4.	<i>Podsumowanie</i>	242
Rozdział 14. Wzorce zarządzania zasobami 245		
14.1.	<i>Wzorzec Enkapsulacja Zasobów</i>	245
14.1.1.	<i>Opis problemu</i>	246
14.1.2.	<i>Stosowanie wzorca</i>	246
14.1.3.	<i>Weryfikacja wzorca</i>	252
14.1.4.	<i>Kiedy stosować wzorzec?</i>	253
14.2.	<i>Wzorzec Wypożyczenie Zasobu</i>	253
14.2.1.	<i>Opis problemu</i>	254
14.2.2.	<i>Stosowanie wzorca</i>	254
14.2.3.	<i>Weryfikacja wzorca</i>	256
14.2.4.	<i>Kiedy stosować wzorzec?</i>	257
14.2.5.	<i>Zagadnienia implementacyjne</i>	257
14.2.6.	<i>Wariant: zastosowanie wzorca Wypożyczenie Zasobu do częściowego udostępniania zasobu</i>	258
14.3.	<i>Wzorzec Złożone Polecenie</i>	258
14.3.1.	<i>Opis problemu</i>	259
14.3.2.	<i>Stosowanie wzorca</i>	260
14.3.3.	<i>Weryfikacja wzorca</i>	267
14.3.4.	<i>Kiedy stosować wzorzec?</i>	267
14.4.	<i>Wzorzec Pula Zasobów</i>	268
14.4.1.	<i>Opis problemu</i>	268
14.4.2.	<i>Stosowanie wzorca</i>	269

- 14.4.3. Weryfikacja wzorca 271
- 14.4.4. Zagadnienia implementacyjne 272
- 14.5. Wzorec Zarządzane Blokowanie 272
 - 14.5.1. Opis problemu 273
 - 14.5.2. Stosowanie wzorca 273
 - 14.5.3. Weryfikacja wzorca 276
 - 14.5.4. Kiedy stosować wzorec? 277
- 14.6. Podsumowanie 277

Rozdział 15. Wzorce przepływów komunikatów 279

- 15.1. Wzorec Zapytanie-Odpowiedź 280
 - 15.1.1. Opis problemu 280
 - 15.1.2. Stosowanie wzorca 281
 - 15.1.3. Popularne implementacje wzorca 282
 - 15.1.4. Weryfikacja wzorca 287
 - 15.1.5. Kiedy stosować wzorec? 288
- 15.2. Wzorec Samowystarczalny Komunikat 288
 - 15.2.1. Opis problemu 289
 - 15.2.2. Stosowanie wzorca 289
 - 15.2.3. Weryfikacja wzorca 291
 - 15.2.4. Kiedy stosować wzorec? 292
- 15.3. Wzorec Zapytaj 292
 - 15.3.1. Opis problemu 293
 - 15.3.2. Stosowanie wzorca 293
 - 15.3.3. Weryfikacja wzorca 296
 - 15.3.4. Kiedy stosować wzorec? 297
- 15.4. Wzorec Przekaz Przepływ 298
 - 15.4.1. Opis problemu 298
 - 15.4.2. Stosowanie wzorca 298
 - 15.4.3. Weryfikacja wzorca 299
 - 15.4.4. Kiedy stosować wzorec? 299
- 15.5. Wzorec Agregator 300
 - 15.5.1. Opis problemu 300
 - 15.5.2. Stosowanie wzorca 300
 - 15.5.3. Weryfikacja wzorca 304
 - 15.5.4. Kiedy stosować wzorec? 304
- 15.6. Wzorec Saga 304
 - 15.6.1. Opis problemu 305
 - 15.6.2. Stosowanie wzorca 306
 - 15.6.3. Weryfikacja wzorca 307
 - 15.6.4. Kiedy stosować wzorec? 309
- 15.7. Wzorec Biznesowy Uścisk Dłoni (lub Niezawodna Dostawa) 309
 - 15.7.1. Opis problemu 310
 - 15.7.2. Stosowanie wzorca 310
 - 15.7.3. Weryfikacja wzorca 314
 - 15.7.4. Kiedy stosować wzorec? 315
- 15.8. Podsumowanie 315

Rozdział 16. Wzorce sterowania przepływem komunikatów 317

- 16.1. Wzorec Pobierz 317
 - 16.1.1. Opis problemu 318
 - 16.1.2. Stosowanie wzorca 318
 - 16.1.3. Weryfikacja wzorca 320
 - 16.1.4. Kiedy stosować wzorec? 321
- 16.2. Wzorec Zarządzana Kolejka 321
 - 16.2.1. Opis problemu 322
 - 16.2.2. Stosowanie wzorca 322
 - 16.2.3. Weryfikacja wzorca 323
 - 16.2.4. Kiedy stosować wzorec? 324
- 16.3. Wzorec Pomiń 324
 - 16.3.1. Opis problemu 324
 - 16.3.2. Stosowanie wzorca 325
 - 16.3.3. Weryfikacja wzorca 327
 - 16.3.4. Kiedy stosować wzorec? 329
- 16.4. Wzorec Dławik 330
 - 16.4.1. Opis problemu 330
 - 16.4.2. Stosowanie wzorca 330
 - 16.4.3. Weryfikacja wzorca 333
- 16.5. Podsumowanie 333

Rozdział 17. Wzorce zarządzania i zapisywania stanów 335

- 17.1. Wzorec Obiekt Domenowy 336
 - 17.1.1. Opis problemu 336
 - 17.1.2. Stosowanie wzorca 336
 - 17.1.3. Weryfikacja wzorca 339
- 17.2. Wzorec Odłamkowanie 340
 - 17.2.1. Opis problemu 340
 - 17.2.2. Stosowanie wzorca 340
 - 17.2.3. Weryfikacja wzorca 342
 - 17.2.4. Ważna uwaga 342
- 17.3. Wzorec Źródło Zdarzeń 343
 - 17.3.1. Opis problemu 343
 - 17.3.2. Stosowanie wzorca 343
 - 17.3.3. Weryfikacja wzorca 345
 - 17.3.4. Kiedy stosować wzorec? 345
- 17.4. Wzorec Strumień Zdarzeń 346
 - 17.4.1. Opis problemu 347
 - 17.4.2. Stosowanie wzorca 347
 - 17.4.3. Weryfikacja wzorca 349
 - 17.4.4. Kiedy stosować wzorec? 349
- 17.5. Podsumowanie 350

DODATKI***Dodatek A Tworzenie diagramów systemów reaktywnych*** 353***Dodatek B Zilustrowany przykład*** 355

- B.1. Partycje geograficzne 355
- B.2. Planowanie przepływu informacji 357
 - B.2.1. Krok 1.: odbieranie danych 358
 - B.2.2. Krok 2.: przysyłanie danych do odpowiedniego węzła 358
 - B.2.3. Krok 3.: relokacja i efektywne odczytywanie danych 359
 - B.2.4. Bilans 362
- B.3. Co będzie w przypadku awarii? 362
 - B.3.1. Awaria klienta 363
 - B.3.2. Awaria łącza sieciowego 364
 - B.3.3. Awaria węzła wejściowego dla danych 364
 - B.3.4. Awaria łącza sieciowego między punktem wejścia danych a kwadratem mapy 364
 - B.3.5. Awaria węzła z kwadratami mapy 365
 - B.3.6. Awaria komponentu kwadratu sumarycznego 366
 - B.3.7. Awaria łącza pomiędzy kwadratami mapy 366
 - B.3.8. Awaria węzła z widokami map 366
 - B.3.9. Podsumowanie obsługi awarii 366
- B.4. Czego nauczyliśmy się z tego przykładu? 367
- B.5. Co dalej? 368

Dodatek C Manifest reaktywny 369

- C.1. Główna treść 369
- C.2. Glosariusz 371
 - C.2.1. Asynchronizm 371
 - C.2.2. Ciśnienie wsteczne 371
 - C.2.3. Przetwarzanie wsadowe 371
 - C.2.4. Komponent 372
 - C.2.5. Delegowanie 372
 - C.2.6. Elastyczność 372
 - C.2.7. Awaria 373
 - C.2.8. Izolacja 373
 - C.2.9. Przezroczystość lokalizacji 374
 - C.2.10. Sterowanie komunikatami 374
 - C.2.11. Algorytm nieblokujący 375
 - C.2.12. Protokół 375
 - C.2.13. Replikacja 375
 - C.2.14. Zasób 375
 - C.2.15. Skalowalność 376
 - C.2.16. System 376
 - C.2.17. Użytkownik 376

Skorowidz 377

Przesyłanie komunikatów

4

Podstawowa idea przesyłania komunikatów jest oparta na zdarzeniach. Fakt, że w aplikacji zaistniała jakaś sytuacja (*zdarzenie*), jest wiązany z kontekstową informacją, np. co kto zrobił, kiedy i gdzie. Taki fakt jest sygnalizowany w postaci *komunikatu* przez nadawcę. Nadawca (*producent*) komunikatu informuje zainteresowane strony (*konsumentów*), wykorzystując uniwersalny mechanizm transportowy.

W tym rozdziale szczegółowo opisujemy następujące aspekty przesyłania komunikatów:

- różnice pomiędzy komunikatami a zdarzeniami,
- asynchroniczne i synchroniczne przesyłanie komunikatów,
- różnej jakości metody przesyłania komunikatów bez przeciążania ich odbiorcy.

Po przeczytaniu tego rozdziału będziesz również wiedział, jak za pomocą komunikatów uzyskać pionową skalowalność systemu. Na koniec opisujemy modelowe zależności pomiędzy zdarzeniami a komunikatami.

4.1. Komunikaty

Gdy w rzeczywistym świecie wysyłasz komuś list, oczekujesz, że jego treść nie zmieni się po drodze. Spodziewasz się również, że do adresata dotrze ten sam list, który wysłałeś, i że nie zmieni się w inny list. Zasada ta obowiązuje niezależnie od tego, czy list wysyłasz na drugi koniec świata i dotrze on do adresata za kilka tygodni, czy też na adres w tym samym mieście albo przekażesz adresatowi list osobiście. Ważna jest niezmiennosc treści listu.

W pierwszej części rozdziału 3. opisaliśmy język Erlang i pierwsze implementacje modelu Aktor. W modelu tym aktorzy komunikują się między sobą za pomocą *komunikatów*. Komunikaty zazwyczaj przesyłane są między procesami, ale mogą być przesyłanie

między komputerami lub między wątkami w ramach tego samego procesu. Twoim zadaniem jest serializacja komunikatu przed transmisją, chyba że jesteś pewien, iż komunikat nigdy nie opuści bieżącego procesu. Lepiej jednak nie przyjmować takich założeń. Za pomocą przesyłania komunikatów można często w prosty sposób poziomo skalować aplikacje, przenosząc odbiorcę komunikatów do innego procesu.

Wywołanie metody możesz traktować jako przesłanie dwóch komunikatów: jednego zawierającego parametry wejściowe metody, a drugiego zawierającego jej wynik działania. Przykład ten może wydawać się dość ekstremalny, ale języki programowania sprzed trzech dekad, np. Smalltalk-80, pokazały, że jest to przydatne podejście¹. Jeżeli metoda nie zwraca komunikatu, wtedy czasami nazywana jest procedurą lub mówi się o niej, że zwraca wartość void (pustka).

4.2. Pionowa skalowalność aplikacji

Wyobraź sobie ruchliwy urząd pocztowy z czasów, gdy listy nie były sortowane za pomocą maszyn ani komputerów. Robili to ludzie i aby proces przebiegał szybciej, angażowanych było kilku pracowników (patrz rysunek 4.1). Ta sama zasada obowiązuje w aplikacjach reaktywnych, w których kolejność przetwarzania zapytań nie jest ważna (więcej informacji na ten temat znajdziesz w rozdziale 14.).



Rysunek 4.1. Dwóch pracowników poczty sortujących równocześnie listy

¹ <https://pl.wikipedia.org/wiki/Smalltalk>

Wyobraź sobie kod wykonujący skomplikowane obliczenia (np. rozkład liczby na czynniki pierwsze, przeszukiwanie grafu, przekształcanie kodu XML itp.). Jeżeli taki kod jest metodą wywoływaną wyłącznie w sposób synchroniczny, wtedy zadaniem kodu nadrzędnego jest zapewnienie równoległości, aby można było wykorzystać kilka rdzeni procesora. Można to osiągnąć np. za pomocą `futur`, wywołując metodę w kilku wątkach z puli. Problem w takim podejściu polega jednak na tym, że tylko kod nadrzędny „wie”, czy metodę można wywoływać równoległe, nie powodując problemów typowych dla równoległości. Zazwyczaj problemy pojawiają się wtedy, gdy do obliczeń wykorzystywane są zewnętrzne struktury danych klasy lub obiekt bez poprawnie zaimplementowanej synchronizacji wewnętrznych operacji.

Przesyłanie komunikatów rozwiązuje powyższy problem, ponieważ nadawca i odbiorca komunikatu są osobnymi procesami. Przy użyciu komunikatów można rozproszyć obliczenia na kilka jednostek bez wiedzy nadawcy.

Piękno takiego podejścia polega na tym, że mechanizm przysyłania komunikatów nie jest uzależniony od sposobu ich przetwarzania przez odbiorcę. Aplikację można transparentnie skalować pionowo na wiele rdzeni procesora i ukrywać w ten sposób przed kodem nadrzędnym szczegóły implementacyjne i opcje konfiguracyjne procesu.

4.3. Sterowanie zdarzeniami i komunikatami

Istnieją dwa modele łączenia producentów danych z ich konsumentami: za pomocą zdarzeń lub z wykorzystaniem komunikatów. Systemy sterowane zdarzeniami umożliwiają wiązanie odpowiedzi z określonymi zdarzeniami. Za każdym razem, gdy pojawi się zdarzenie, system musi wykonać odpowiednią operację. Podstawą działania tego rodzaju systemów są zazwyczaj pętle zdarzeń. Gdy coś się w systemie wydarzy, np. użytkownik kliknie myszą, w kolejce umieszczane jest odpowiednie zdarzenie. W pętli pobierane są na bieżąco zdarzenia z kolejki i wywoływane skojarzone z nimi funkcje zwrotne. Każda funkcja jest zazwyczaj krótkim, asynchronicznym kodem, który może zgłaszać kolejne zdarzenia, również umieszczane następnie w kolejce i przetwarzane, gdy nadejdzie ich kolej. Ten model obsługi zdarzeń jest implementowany w jednowątkowych środowiskach, np. w Node.js i w interfejsach graficznych większości systemów operacyjnych.

Natomiast w systemach sterowanych komunikatami nadawcy wysyłają komunikaty do określonych odbiorców. Zamiast anonimowych funkcji zwrotnych wykorzystywani są aktywni odbiorcy konsumujący komunikaty wysyłane przez potencjalnie anonimowych producentów. W odróżnieniu od systemu sterowanego zdarzeniami, w którym producent zdarzenia ma adres, aby można mu było przypisać funkcję zwrotną, w systemie sterowanym komunikatami adres ma konsument, aby mógł przetwarzać odpowiednie komunikaty. Ani producent komunikatu, ani system nie zajmują się przetwarzaniem komunikatu, ponieważ jest to zadanie jego konsumenta. Przykładowo komponent systemu zgłaszający komunikaty logujące nie „martwi się”, czy będą one co 6 godzin lub co 24 godziny konsumowane przez sieć, bazę danych, czy system plików. Proces odbierający komunikat jest odpowiedzialny za jego właściwe przetworzenie.

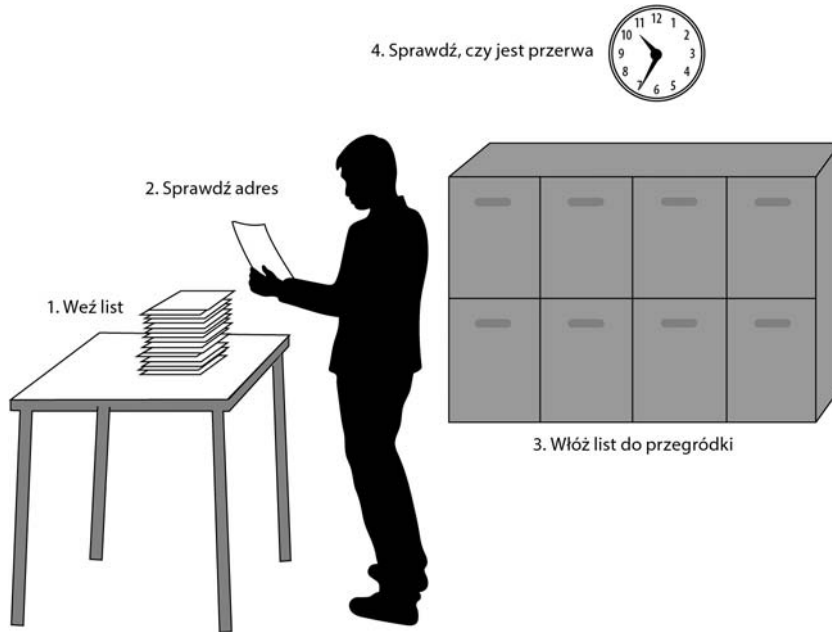
Przeniesienie odpowiedzialności za przetwarzanie komunikatów ma wiele zalet.

- Umożliwia sekwencyjne, stanowe przetwarzanie komunikatów przez poszczególnych konsumentów bez konieczności synchronizowania ich działań. Dzięki temu można lepiej wykorzystać możliwości komputera, ponieważ konsumenci agregują odbierane komunikaty i od razu je przetwarzają, a dzisiejsze komputery są zoptymalizowane pod kątem takich operacji.
- Dzięki sekwencyjnemu przetwarzaniu komunikatów konsument odsyła odpowiedź zależną od jego bieżącego stanu. Zatem odebrane wcześniej komunikaty mają wpływ na bieżące działanie konsumenta. W odróżnieniu od tego modelu, w systemie sterowanym zdarzeniami decyzja o rodzaju odpowiedzi podejmowana jest w momencie rejestrowania zdarzenia, a nie jego wystąpienia.
- Konsument może pominąć komunikat lub uprościć jego przetwarzanie, jeżeli system jest przeciążony. Uogólniając, można stwierdzić, że jawne kolejkowanie komunikatów pozwala konsumentowi kontrolować ich przepływ. Więcej na temat przepływu danych dowiesz się w podrozdziale 4.5.
- Ponadto takie podejście jest bliższe stylowi pracy człowieka, który również sekwencyjnie przetwarza zapytania odbierane od współpracowników.

Ostatni punkt może wydawać się dziwny, ale naszym zdaniem tego rodzaju porównania pomagają ilustrować działanie komponentów systemu. Wyobraź sobie stary urząd pocztowy, taki jak na rysunku 4.2, w którym pracownicy sortują listy, zdejmując je ze stery na stole i umieszczając w przegródkach. Pracownik bierze list, sprawdza adres i podejmuje decyzję, do której przegródki włożyć kopertę. Potem wraca do pliku listów i albo bierze następny, albo sprawdza, czy minęło już południe i czas na przerwę obiadową. Ta analogia pozwoli zrozumieć mechanizm sterowania komunikatami. Teraz wystarczy przenieść ją na kod. Zadanie to, bez powyższego prostego ćwiczenia umysłowego, byłoby znaczne trudniejsze.

Podobieństwa między przesyłaniem komunikatów a czynnościami człowieka dotyczą nie tylko procesów sekwencyjnych. Zamiast bezpośrednio odczytywać (i zapisywać) czyjeś myśli, ludzie przesyłają komunikaty: rozmawiają, piszą notatki, obserwują twarze itd. Ta sama zasada obowiązuje w projektowaniu oprogramowania. Tworzy się obiekty oddziałujące na siebie poprzez przesyłanie komunikatów. Nie są to typowe obiekty, jakie znasz z języków Java, C# czy C++, ponieważ komunikacja w tych językach odbywa się w sposób synchroniczny i odbiorca zapytania nie ma wpływu na moment jego przetworzenia. Odpowiada to sytuacji, w której szef dzwoni do pracownika i domaga się natychmiastowej odpowiedzi na swoje pytanie. Wiadomo, że taki przypadek jest raczej wyjątkiem, a nie regułą. Spodziewamy się, że pracownik odpowie: „Dobrze, oddzwonię, gdy będę znał odpowiedź” albo jeszcze lepiej — szef, zamiast nerwowo dzwonić, wyśle wiadomość e-mail, szczególnie gdy wie, że znalezienie odpowiedzi zajmie pracownikowi trochę czasu. Odpowiedź „oddzwonię do ciebie” odpowiada użyciu futury, natomiast wysłanie e-maila odpowiada jawnemu przesłaniu komunikatu.

Teraz, po uzasadnieniu, że przesyłanie komunikatów jest przydatną ideą, musimy omówić dwa fundamentalne problemy pojawiające się podczas przetwarzania komunikatów zarówno na poczcie, jak i w komputerze.



Rysunek 4.2. Pracownik na zapleczu poczty rozdziela stos listów na przegródki

- Czasami trzeba zagwarantować dostarczenie bardzo ważnej przesyłki.
- Jeżeli komunikaty są nadawane szybciej, niż można je dostarczać, trzeba je gdzieś gromadzić. W szczególnym przypadku system może ulec awarii albo listy mogą być gubione.

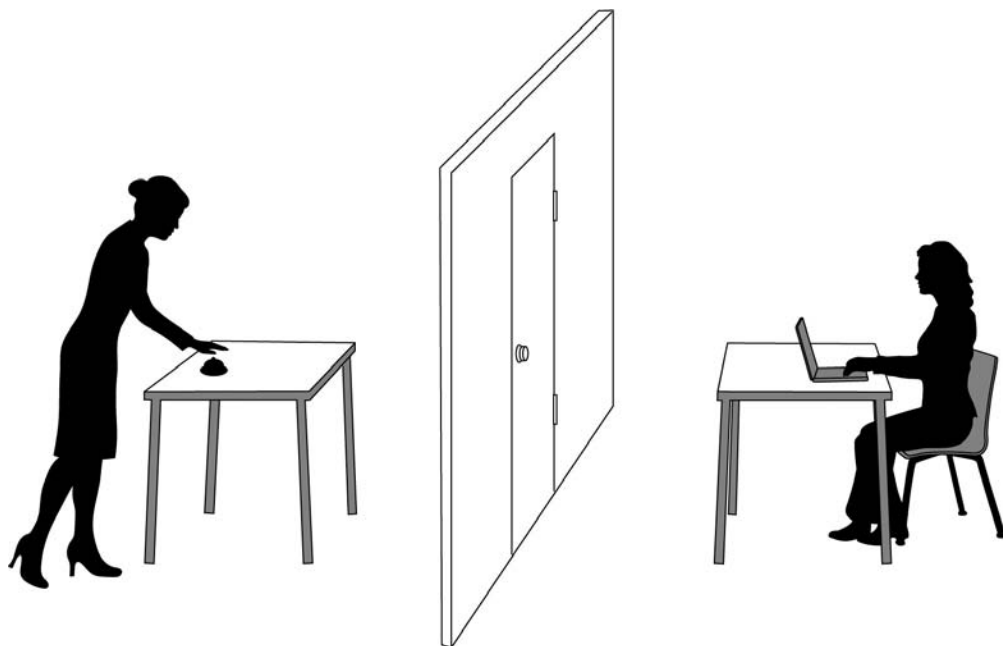
Gwarancję dostarczenia komunikatu omówimy później. Teraz opiszemy, jak system reaktywny steruje przepływem komunikatów, aby odbiorca mógł je sprawnie przetwarzać i nie był przeciążony.

4.4. Synchroniczne i asynchroniczne przesyłanie komunikatów

Komunikacja pomiędzy producentem a konsumentem komunikatu może być dwojakiego rodzaju:

- *synchroniczna*, w której obie strony w danej chwili muszą być gotowe do przesyłania komunikatu
- *asynchroniczna*, w której nadawca wysyła komunikaty niezależnie od tego, czy druga strona może je odbierać, czy nie.

Rysunek 4.3 ilustruje synchroniczne przesyłanie komunikatów na poczcie. Klientka Jola chce nadać list i musi poprosić pracownika poczty, Jacka, o pomoc. Na szczęście, przy stanowisku Jacka nie ma interesantów, ale samego Jacka nigdzie nie widać. Prawdopodobnie jest gdzieś na zapleczu i ekspediuje przesyłkę nadaną właśnie przez poprzedniego



Rysunek 4.3. Komunikat, że interesant potrzebuje pomocy, jest sygnalizowany pracownikowi za pomocą dzwonka

interesanta. Gdy Jacek pracuje, Jola musi na niego beczynnie czekać. Nie może pójść do pracy, do sklepu, ani zająć się czymś innym. Jeżeli będzie czekać zbyt długo, zrezygnuje i spróbuje wysłać list później, prawdopodobnie na innej poczcie.

W praktyce potrafimy sobie bezkonfliktowo radzić w sytuacjach, gdy odbiorca komunikatu jest nieosiągalny przez dłuższy czas. W informatyce przekroczenie czasu i podejmowanie odpowiednich działań jest traktowane jak sytuacja wyjątkowa.

Natomiast asynchroniczne przesyłanie komunikatów oznacza, że Jola nadaje list, wrzucając go do skrzynki. Od razu może zająć się pracą lub iść na randkę. Jakiś czas później pracownik poczty opróżni skrzynkę i umieści listy w odpowiednich przegródkach. Takie rozwiązanie jest o wiele lepsze dla obu stron. Jola nie musi czekać, a pracownik poczty będzie dostępny, a pracownik może sukcesywnie wykonywać swoje zadania i w ten sposób efektywniej wykorzystywać czas. Dlatego taki tryb pracy jest preferowany w każdej sytuacji.

Jeżeli odbiorców komunikatów jest wielu, zalety asynchronicznego wysyłania komunikatów są jeszcze wyraźniej widoczne. Byłoby wielką stratą czasu oczekiwanie, aż wszyscy odbiorcy będą jednocześnie gotowi do komunikacji, czy też synchroniczne wysyłanie komunikatu każdemu odbiorcy z osobna. W rzeczywistym świecie pierwszy przypadek odpowiada organizowaniu spotkania wszystkich konsumentów przez producenta i przesyłaniu im komunikatu w jednej chwili, a drugi przypadek — cierpliwemu oczekiwaniu na każdego konsumenta przy biurku. O wiele lepiej byłoby wysłać komunikat — kiedyś list, dzisiaj e-mail — asynchronicznie.

Podsumowując, kiedy mówimy o *przesyłaniu komunikatów*, zawsze będziemy mieć na myśli *asynchroniczną komunikację pomiędzy producentem a pewną liczbą konsumentów komunikatów*.

Komunikacja asynchroniczna oznacza, że odbiorca zawsze dowie się, że nadszedł nowy komunikat i przetworzy go, gdy tylko nadarzy się sposobność. Odbiorca można się dowiadywać o komunikacie na dwa sposoby: rejestrując funkcję zwrotną wykonującą określoną operację w chwili pojawienia się komunikatu lub sprawdzając skrzynkę (zwaną również *kolejką*) i podejmując za każdym razem decyzję, co zrobić z komunikatem.

Anegdota autora

Pakiet Akka wykorzystujący aktorów od samego początku miał być przystosowany do przesyłania komunikatów. Jednak przed wersją 2.0 mechanizm ten nie obejmował całego pakietu — był dostępny tylko na jego brzegu, tj. w interfejsie API dla użytkownika. Wewnątrz wykorzystywane były blokady synchronizacyjne, a kiedy nadzorowany aktor ulegał awarii, stosowana była asynchroniczna strategia wywołań. W efekcie, w takiej architekturze zdalny nadzór nad aktorami nie był możliwy. Wszystko, co dotyczyło interakcji między aktorami na odległość, było implementowane w dziwaczny i skomplikowany sposób. Użytkownicy pakietu zaczęli się orientować, że tworzenie aktorów odbywało się synchronicznie w wątku inicjującym. Dlatego odradzaliśmy im kodowanie czasochłonnych operacji w konstruktorach aktorów i zamiast tego wysyłanie komunikatów inicjujących.

Gdy lista tego rodzaju mankamentów zaczęła się nadmiernie wydłużać, zastanowiliśmy się i na nowo zaprojektowaliśmy całą wewnętrzną architekturę pakietu Akka, w całości opartą na asynchronicznym przesyłaniu komunikatów. Usunęliśmy wszystkie funkcjonalności, które nie były zgodne z tą zasadą, i zbudowaliśmy całkowicie nieblokujący wewnętrzny mechanizm. W rezultacie dopasowaliśmy do siebie wszystkie elementy układanki. Po usunięciu więzów łączących ściśle poszczególne ruchome części pakietu kosztem włożenia rozsądnej ilości pracy zaimplementowaliśmy nadzorowanie aktorów i lokalną przezroczystość oraz osiągnęliśmy wyjątkową skalowalność systemu. Jedynym jego mankamentem jest to, że niektóre funkcjonalności, np. eskalowanie informacji o awarii w górę hierarchii nadzorców, nie tolerują utraty komunikatów, przez co w zaimplementowanie zewnętrznej komunikacji trzeba włożyć nieco więcej wysiłku.

4.5. Sterowanie przepływem danych

Sterowanie przepływem danych jest to proces spowalniania prędkości strumienia komunikatów, aby nie przeciążyć ich odbiorcy. Informowanie nadawcy, aby zwolnił, nosi nazwę **ciśnienia wstecznego**.

Zwykle wywoływanie metod, takie jak np. w języku C, z natury obejmuje pewne sterowanie przepływem komunikatów — wykonywanie kodu nadawcy zapytania jest wstrzymywane do czasu zakończenia wykonywania kodu odbiorcy. Jeżeli o zasoby odbiorcy konkuruje wielu nadawców, ich kody są wykonywane szeregowo i są synchronizowane za pomocą różnego rodzaju blokad, np. semaforów, które dodatkowo blokują nadawcę, aż zostanie zakończone wykonywanie kodu poprzedniego nadawcy. Tego rodzaju niejawne ciśnienie wsteczne może się na pierwszy rzut oka wydawać użyteczne, ale okazuje się przeszkodą w miarę powiększania programu i wzrostu znaczenia jego cech niefunkcjonalnych. Programista, zamiast implementować algorytm, musi tracić czas na diagnozowanie słabych punktów programu.

Przesyłanie komunikatów oferuje większe możliwości sterowania przepływem danych, ponieważ opiera się na idei kolejki. Przykładowo program wykorzystujący związaną kolejkę, opisaną w rozdziale 2., może w zależności od stawianych mu wymagań różnie reagować, gdy kolejka będzie pełna, np. może pomijać najnowsze, najstarsze lub wszystkie komunikaty. Ten ostatni przypadek stosowany jest w systemach przetwarzających dane w czasie rzeczywistym, np. wyświetlających najnowsze dane. Aby system płynnie przetwarzał dane i niwelował ich nierównomierny napływ, można zastosować mały bufor, przy czym szybkie przechodzenie nowych komunikatów przez kolejkę jest ważniejsze niż przetwarzanie wszystkich komunikatów, nawet tych, które oczekują w kolejce dłuższy czas. Ponieważ konsument odczytuje komunikaty z kolejki, może podejmować różne decyzje, gdy okaże się, gdy ma zaległości w przetwarzaniu.

Innym rozwiązaniem jest umieszczanie odbieranych komunikatów o różnych priorytetach w osobnych kolejkach. Można również, jeśli trzeba, natychmiast odrzucać komunikaty i powiadamiać o tym ich nadawcę. Te i inne możliwości opiszemy dokładniej w rozdziale 15.

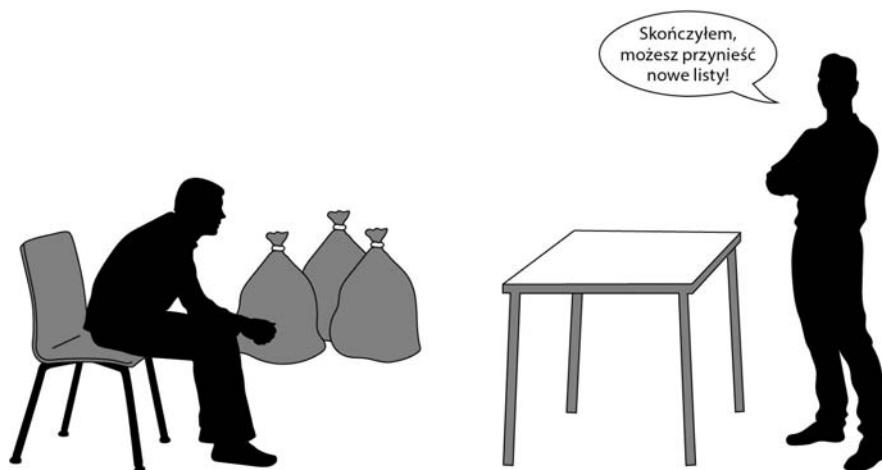
Rysunek 4.4 przedstawia dwa podstawowe schematy sterowania przepływem danych. Osoba z lewej strony przynosi worek listów dla drugiej osoby, która sortuje listy na stole. Schemat **negatywnego potwierdzenia** (ang. *negative acknowledgment* — NACK) polega na tym, że gdy na stole nie ma miejsca na nowe listy, osoba z prawej strony natychmiast odmawia ich przyjęcia. W schemacie **pozytywnego potwierdzenia** (ang. *positive acknowledgment* — ACK) osoba z lewej strony czeka, aż dostanie informację, że osoba z prawej skończyła sortować. Istnieje wiele odmian tego schematu, niektóre z nich zostały opisane w rozdziale 15., ale ich idea wiernie przekłada się na podobne do opisanych przykłady z życia wzięte. Ty również możesz puścić wodze fantazji i na podstawie codziennych obserwacji wymyślać nowe schematy.

Przesyłanie komunikatów i sterowanie przepływem danych jest możliwe w wielu językach programowania obiektowego, dzięki czemu można tworzyć nietypowe rozwiązania. Oczywiście, wiąże się z tym określone koszty. Należy przede wszystkim zastanowić się, jak ma przebiegać sterowanie przepływem, jak również czy w przypadku niewielkiego strumienia danych lepszym rozwiązaniem będzie zwykle wywoływanie metod. Aby zilustrować ten przypadek, wyobraźmy sobie system oparty na usługach, wyposażony w asynchroniczne interfejsy. Same usługi mogą być zaimplementowane w tradycyjny, synchroniczny sposób i też mogą przysyłać między sobą informacje za pomocą komunikatów. Podczas usprawniania jednej z usług może okazać się, że warto byłoby zastosować w niej sterowanie przepływem danych. Taką decyzję można podjąć w zależności od wymagań stawianych danej usłudze, jak również od zasad pracy zespołu programistów.

Teraz, gdy wiesz już, jak zapobiegać przeciążeniu systemu opartego na przysyłaniu komunikatów, możemy zająć się tematem gwarancji dostarczania wybranych ważnych komunikatów.



Sterowanie przepływem z negatywnym potwierdzeniem



Sterowanie przepływem z pozytywnym potwierdzeniem

Rysunek 4.4. Dwa podstawowe schematy sterowania przepływem danych

4.6. Gwarancja dostarczania komunikatów

Listy mogą się gubić, nawet gdy pracownicy poczty starają się, jak mogą. Prawdopodobieństwo takiego zdarzenia jest małe, ginie kilka listów w roku, ale zawsze jest to możliwe. Co się wtedy dzieje? Jeżeli list zawierał życzenia urodzinowe, wtedy o jego utracie nadawca i odbiorca dowiedzą się za jakiś czas, gdy się spotkają. Jednak może zawierać

fakturę, która nie zostanie zapłacona i nadawca wyśle przypomnienie. Ważne jest, że w rzeczywistym świecie mają miejsce interakcje między ludźmi pozwalające stwierdzać zagubienie listów. Czasami mają miejsce nieprzyjemne konsekwencje, ale świat toczy się dalej i z powodu straty jednego listu życie całego społeczeństwa nie zatrzymuje się w miejscu. W tym podrozdziale dowiesz się, że aplikacje reaktywne działają w podobny sposób. Zaczniemy jednak z innej strony: od systemów synchronicznych.

Umieszczając w kodzie wywołanie metody, programista ma całkowitą pewność, że zostanie ona wykonana. Nie ma możliwości pominięcia niektórych wierszy kodu podczas wykonywania programu. Jednak trzeba wziąć pod uwagę możliwość przerwania programu, awarii komputera, przepełnienia stosu lub pojawienia się krytycznego sygnału. Jak już wiesz, wywołanie metody można porównać do wysłania komunikatu do obiektu, zatem nawet w systemie synchronicznym trzeba uwzględnić fakt, że komunikat, czyli wywołanie metody, zgubi się. Wniosek stąd płynie taki, że nie można dać całkowitej gwarancji, iż system przetworzy zapytanie i odeśle odpowiedź.

Bardzo niewielu ludzi uważa, że jest to konstruktywny wniosek, ponieważ wszyscy akceptujemy w granicach rozsądku pewne ograniczenia i wyjątki od reguł. Kierujemy się zdrowym rozsądkiem, a ludzi, którzy tak nie robią, nazywamy pedantami. Przykładowo w interakcjach między ludźmi domyślnie zakładamy, że żadna z komunikujących się stron nie kłamie. Pomijając rzadkie wyjątki, staramy się zaradzić własnej wrodzonej omyłności i sprawdzamy, czy wysłane informacje dotarły do odbiorcy, oraz przypominamy kolegom o ważnych zadaniach do wykonania czy terminach. Jednak przenosząc ten proces na grunt informatyki, oczekujemy, że będzie on całkowicie niezawodny i wykona swoje zadania bezawaryjnie. Ludzie z natury nie lubią powtarzających się nieprawidłowości i oczekują, że komputery je wyeliminują. Smutna prawda jest jednak taka, że urządzenia, które budujemy, są od nas wprawdzie szybsze i dokładniejsze, ale nie są idealne. Dlatego musimy liczyć się z ich nieprzewidywanymi awariami.

Jak zwykle, codzienne życie dostarcza dobrych wzorów do naśladowania. Jeżeli jedna osoba prosi drugą o przysługę, musi wziąć pod uwagę, że nie otrzyma odpowiedzi. Druga osoba może być zajęta lub prośba — np. list lub wiadomość e-mail — zgubi się gdzieś po drodze. W takich sytuacjach pierwsza osoba będzie ponawiać prośbę do momentu, aż uzyska odpowiedź albo uzna, że dalsze ponawianie prośb nie ma sensu. Analogia w informatyce jest oczywista: komunikat wysłany przez jeden obiekt do innego obiektu może się zgubić lub odbiorca może nie być w stanie przetworzyć komunikatu, ponieważ jakiś niezbędny do tego celu zasób nie jest dostępny, np. dysk jest pełny lub baza danych nie odpowiada. W takiej sytuacji obiekt nadawczy może ponawiać próby aż do skutku albo zaprzestać wysyłania komunikatów.

W przypadku synchronicznego wywoływania metod zazwyczaj nie istnieje coś takiego jak reakcja na utratę komunikatu. Jeżeli metoda nie zostanie wykonana, zazwyczaj oznacza to jakąś katastrofę, np. niewłaściwe działanie programu. Kod wywołujący metodę nie ma żadnej możliwości jej ponownego wywołania i kontynuowania swojego działania. Natomiast przesyłanie komunikatów ma tę zaletę, że zapytanie może być zapamiętane i powtórzone, gdy problem braku odpowiedzi zostanie rozwiązany. Jeśli nawet całe centrum danych przestanie działać z powodu awarii zasilania, po jego przywróceniu programy

mogą wznowić swoje działanie i przetworzyć komunikaty zapisane wcześniej w nieulotnej pamięci. Podobnie jak w przypadku sterowania przepływem danych, trzeba w zależności od wymagań stawianych aplikacji wybrać odpowiednią granulację przesyłania komunikatów.

Mając na względzie powyższe sytuacje, oczywistym wydaje się projektowanie aplikacji o obniżonej gwarancji dostarczania komunikatów. Taka aplikacja będzie odporna na utraty komunikatów spowodowane czy to przerwami w transmisji sieciowej, niedostępnością innych usług, przeciążeniem systemu, błędami w kodzie, czy innymi czynnikami.

Implementacja środowiska o bardzo wysokiej gwarancji dostarczenia komunikatów jest kosztowna, ponieważ wymaga stosowania dodatkowych mechanizmów, np. do wielokrotnego wysyłania komunikatów, dopóki nie zostanie odebrana odpowiedź. Tego typu mechanizmy obniżają wydajność i skalowalność systemu nawet wtedy, gdy nie ma awarii. Jeszcze szybciej koszty rosną w rozproszonym środowisku, głównie z tego powodu, że czas przesyłania komunikatu przez sieć jest kilka rzędów wielkości dłuższy niż wewnątrz lokalnego systemu (np. pomiędzy dwoma rdzeniami tego samego procesora). Poprzez obniżenie gwarancji dostarczenia komunikatu można zwykły system zaimplementować szybciej i prościej, a wyższą cenę za lepszą gwarancję płacić tylko wtedy, gdy jest to naprawdę konieczne. Zwróć uwagę na analogię ceny systemu do opłat za list zwykły i polecony.

Istnieją trzy podstawowe typy gwarancji.

- *Najwyżej jedno dostarczenie*: każdy komunikat jest wysyłany tylko raz. Jeżeli się zgubi lub odbiorca nie będzie mógł go przetworzyć, następna próba nie zostanie podjęta. Dlatego żądana usługa może być wywołana tylko jeden raz albo wcale. Jest to najbardziej podstawowa gwarancja o najniższym koszcie uzyskania, ponieważ ani nadawca, ani odbiorca komunikatu nie musi kontrolować stanu komunikacji.
- *Co najmniej jedno dostarczenie*: zagwarantowanie, że komunikat zostanie przetworzony, wymaga rozbudowania powyższej formy gwarancji o dwa wymagania. Po pierwsze, odbiorca musi potwierdzać odbiór komunikatu, wysyłając inny komunikat. Po drugie, nadawca musi zapamiętać zapytanie na wypadek, gdyby nie odebrał potwierdzenia. Ponieważ brak potwierdzenia może skutkować ponownym wysłaniem komunikatu, odbiorca może go otrzymać więcej niż jeden raz. Jeżeli czasu jest wystarczająco dużo i komunikacja się powiedzie, wtedy odbiorca otrzyma komunikat przynajmniej jeden raz.
- *Dokładnie jedno dostarczenie*: jeżeli komunikat musi zostać przetworzony dokładnie jeden raz, wtedy do powyższej gwarancji trzeba dodać warunek, że odbiorca musi deduplikować otrzymywane komunikaty. W tym celu musi rejestrować komunikaty, które już przetworzył. Jest to najbardziej kosztowny wariant gwarancji, ponieważ zarówno nadawca, jak i odbiorca muszą śledzić stan komunikacji. Jeżeli dodatkowo komunikaty muszą być przetwarzane w kolejności ich wysyłania, wtedy ich prędkość przesyłania maleje, ponieważ dochodzi zwłoka wprowadzana przez potwierdzenia otrzymania każdego komunikatu (chyba że sterowanie przepływem danych jest dopasowane do wielkości bufora komunikatów po stronie odbiorcy).

Implementacje modelu Aktor zazwyczaj z definicji oferują pierwszy rodzaj gwarancji i pozwalają na dodanie dwóch kolejnych rodzajów, jeżeli będzie trzeba. Co ciekawe, lokalne wywołania metod również są objęte powyższą gwarancją, choć prawdopodobieństwo niepowodzenia wywołania metody jest znikome.

Dobór kontroli przepływu danych i rodzaju gwarancji dostarczania komunikatów jest bardzo ważny, ponieważ przesyłanie komunikatów ujawnia istniejące ograniczenia w komunikacji i ewentualne utrudnienia. W kolejnym podrozdziale zajmiemy się naturalną analogią pomiędzy komunikatami a rzeczywistymi zdarzeniami oraz utrzymaniem spójności komunikacji pomiędzy warstwami aplikacji.

4.7. Zdarzenia jako komunikaty

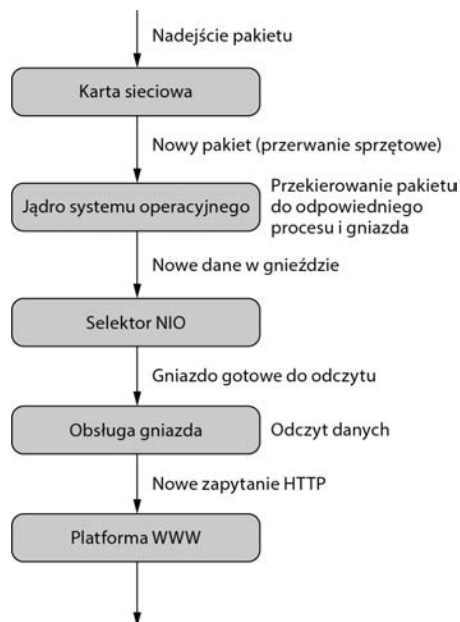
W rzeczywistych wysokowydajnych systemach najważniejszą kwestią jest utrzymanie w określonych granicach maksymalnego czasu reakcji na zewnętrzne zdarzenia, czyli uzyskanie ścisłego czasowego związku pomiędzy zdarzeniem a reakcją na nie. Natomiast macierze o dużej pojemności, np. rejestrujące komunikaty, muszą charakteryzować się wysoką przepustowością, jednak ich czasy odpowiedzi mogą być dłuższe. Czas potrzebny na zapisanie komunikatu nie jest tak istotny, jeżeli ma się pewność, że zostanie na pewno zapisany. Wymagania dotyczące reagowania systemu na zdarzenia mogą być bardzo różne, jednak interakcje pomiędzy człowiekiem a komputerem zawsze sprowadzają się do zgłaszania zdarzeń i odpowiadania na nie.

Komunikaty są naturalnymi reprezentantami zdarzeń, a przesyłanie komunikatów w naturalny sposób reprezentuje interakcje oparte na zdarzeniach. Propagację zdarzeń w systemie można traktować jako przesyłanie komunikatów przez łańcuch przetwarzających go jednostek. Reprezentacja zdarzeń w postaci komunikatów pozwala osiągnąć kompromis pomiędzy zwłoką w odpowiedzi a przepustowością systemu. Kompromis ten może być różny w zależności od przypadku, a nawet zmieniać się dynamicznie. Jest to przydatna cecha w sytuacji, gdy czasy odpowiedzi systemu muszą być krótkie, ale mogą się wydłużać w miarę wzrostu obciążenia.

Rysunek 4.5 przedstawia proces odbierania i przetwarzania pakietów sieciowych. Obowiązują tu różne wymagania dotyczące czasu realizacji poszczególnych etapów. Najpierw karta sieciowa (ang. *network interface controller* — NIC), wykorzystując przerwanie synchroniczne, informuje procesor o pojawieniu się danych. Ta operacja musi być wykonana tak szybko, jak to jest możliwe. Jądro systemu operacyjnego odbiera dane od karty i sprawdza, do jakiego gniazda i procesu należy je przekazać. Od tej chwili wymagania dotyczące szybkości przetwarzania są łagodniejsze, ponieważ dane są bezpiecznie przechowywane w pamięci komputera, choć oczywiście dobrze byłoby, gdyby proces odpowiedzialny za przetwarzanie tych danych i wysyłanie odpowiedzi był jak najszybciej informowany. Dostępność danych w gnieździe jest następnie sygnalizowana np. poprzez wybudzenie selektora, po czym aplikacja żąda od jądra systemu dostarczenia danych.

Zwróć uwagę na widoczną tu fundamentalną zasadę: dane odebrane z sieci są przenoszone w postaci serii zdarzeń do coraz wyższych warstw oprogramowania. Każdy

pomyślnie odebrany pakiet danych ostatecznie dociera do oprogramowania użytkowego w formie zawierającej te same informacje, co na początku (aczkolwiek ze względów wydajnościowych pakiety mogą być łączone). Na najniższym poziomie oprogramowania interakcja pomiędzy komputerami polega na przesyłaniu fizycznych komunikatów. Odebranie pakietu jest sygnalizowane zgłoszeniem zdarzenia. Dlatego w naturalny sposób operacje przesyłania danych przez sieć są na *wszystkich* poziomach oprogramowania modelowane w postaci strumienia zdarzeń przekształcanego na strumień komunikatów. Opisaliśmy ten przykład dlatego, ponieważ w ten sposób zaimplementowaliśmy nową warstwę operacji wejścia/wyjścia w pakiecie Akka. Jednak możliwości wiązania komunikatów przesyłanych pomiędzy różnymi warstwami systemu jest mnóstwo. Wszystkie niskopoziomowe operacje wprowadzania danych (naciśnięcia klawiszy, kliknięcia myszą, wysłanie klatki obrazu przez kamerę itp.) są oparte na zdarzeniach, które można wygodnie zamienić na komunikaty. W ten sposób przesyłanie komunikatów staje się najbardziej naturalną formą komunikacji pomiędzy niezależnymi obiektami dowolnych rodzajów.



Rysunek 4.5. Etapy przetwarzania zapytania WWW od chwili odebrania pakietu sieciowego do wywołania usługi

4.8. Synchroniczne przesyłanie komunikatów

Jawne przesyłanie komunikatów często stanowi wygodny sposób komunikowania się odizolowanych od siebie części aplikacji nawet wtedy, gdy asynchroniczna komunikacja nie jest konieczna. Jeżeli jednak asynchroniczność nie jest potrzebna, wtedy jej stosowanie przysparza niepotrzebnych kosztów w postaci większego obciążenia systemu wynikającego ze zlecania zadań do wykonania oraz opóźnień w ich wykonywaniu. Czasami lepszym rozwiązaniem jest synchroniczne przesyłanie komunikatów.

Wspominamy o takiej możliwości dlatego, ponieważ synchroniczne przesyłanie komunikatów często przydaje się podczas przetwarzania strumieni danych. Przykładowo złączenie kilku operacji transformacji danych powoduje, że dane są przetwarzane przez jeden rdzeń procesora i dzięki temu lepiej można wykorzystać jego pamięć podręczną. Synchroniczne przesyłanie komunikatów ma więc inny cel niż oddzielenie od siebie części reaktywnej aplikacji. W takiej sytuacji nasze rozważania na temat konieczności asynchroniczności nie mają odniesienia.

4.9. Podsumowanie

W tym rozdziale uzasadniliśmy szczegółowo konieczność przesyłania komunikatów, zwłaszcza jako alternatywę dla komunikacji synchronicznej. Przedstawiliśmy różnice pomiędzy źródłami zdarzeń w systemach sterowanych zdarzeniami a odbiorcami komunikatów w systemach sterowanych komunikatami. Opisaliśmy różne formy sterowania przepływem danych, stosowanego podczas przesyłania komunikatów. Poznałeś też różne odmiany gwarancji dostarczania komunikatów.

Ogólnie przedstawiliśmy związki pomiędzy zdarzeniami a komunikatami. Dowiedziałeś się, jak dzięki przesyłaniu komunikatów można uzyskać pionową skalowalność aplikacji. W kolejnym rozdziale pokażemy, jak przezroczystość lokalizacji pozwala dodatkowo osiągnąć poziomą skalowalność aplikacji.

Skorowidz

A

ACID 2.0, 56
Agregator, 300
aktor, 83, 85
aktorzy imitacyjni, 177
algorytm nieblokujący, 375
analiza
 czasu odpowiedzi, 38
 paralelizmu, 41
anatomia systemu reaktywnego, 26
API, 183
asercje asynchroniczne, 170
asynchroniczne
 asercje, 170
 łączenie wyników, 44
 przesyłanie komunikatów, 93
asynchroniczność naturalna, 82
asynchronizm, 371
atrapa, 178
awaria, 29, 48, 82, 123, 154, 187, 373
 klienta, 363
 komponentu kwadratu sumarycznego, 366
 łącza sieciowego, 364, 366
 węzła, 364–366

B

bezpiecznik, 51, 202, 206
biblioteka
 Akka, 84
 Erlang, 84
 Rx, 81
biblioteki aktorów, 84
Biznesowy Uścisk Dłoni, 309
blokujące odbiorniki komunikatów, 159

błędy, 154
 asynchroniczne, 173
 domenowe, 183

C

centrum danych, 185
ciśnienie wsteczne, 371
CRDT, 141
CSP, Communicating Sequential Processes, 74
cykl życia modułu, 125
czas
 oczekiwania, 161
 odpowiedzi, 38, 39, 41

D

deklaratywny przepływ danych, 137
delegowanie, 372
diagramy systemów reaktywnych, 353
Dławik, 330
dobór czasu oczekiwania, 161
dynamiczne komponowanie systemu, 111
dziel i rządź, 113

E

efekt kuli błota, 32
efekty uboczne, 69
elastyczność, 372
 systemu, 179
Enkapsulacja Zasobów, 245

F

front-end, 35
funkcje pierwszej klasy, 70
futury, 76

G

grodzienie, 50
grupowanie danych i transakcji, 131
gwarancja dostarczania komunikatów, 97

H

hierarchia modułów, 114, 117, 118
hierarchiczna struktura problemu, 114

I

imitacja, 177
infrastruktura, 183
instancje, 184
integrowanie komponentów niereaktywnych,
33
interakcje użytkowników, 35
interfejs API, 183
izolacja, 373

J

jawne przesyłanie komunikatów, 105
Jądro Błędu, 191
jednostka
awaryjności, 133
spójności, 133
język CSP, 74

K

klaster, 184
komponenty, 372
niereaktywne, 33
rozproszone, 176, 179
komponowane futury, 43
komponowanie systemu, 111
komunikaty, 89, 91
blokujące odbiorniki, 159
gwarancja dostarczania, 97
optymalizacja przekazywania, 107

przesyłanie asynchroniczne, 93
przesyłanie jawne, 105
przesyłanie synchroniczne, 93, 101
szacowanie ilości, 148
utruty, 107
wykrywanie, 167
wzorce przepływów, 279, 317
konsensus rozproszenia, 130
kontrola cyklu życia modułu, 125

L

lokalizacja, 83, 103
lokalne przekazywanie komunikatów, 107

Ł

łańcuch zapytań, 109

M

manifest reaktywny, 369
mapa Europy, 356
model Aktor, 82
modele programistyczne, 60
modelowanie
procesów w domenie, 147
przepływów danych, 131
moduł CQRS, 134
moduły
niezależne, 130
pochodne, 115

N

nadzorowanie usług, 53
nadzór, 82
narzędzia, 63
reaktywne, 72
niedeterminizm, 137
niezmiennosc, 66

O

obciążenie, 28
Obiekt Domenowy, 336
obsługa awarii, 48, 82, 123, 366
odbiorniki komunikatów, 159
odizolowane zakresy spójności, 135

Odłamkowanie, 340
 odporność na awarie, 127
 odtwarzanie systemu, 187
 ograniczanie maksymalnego czasu
 odpowiedzi, 39
 ograniczenia
 odporności na awarie, 147
 paralelizmu, 46
 optymalizacja lokalnego przekazywania
 komunikatów, 107

P

paralelizm, 41, 43
 ograniczenia, 46
 skrócenie czasu odpowiedzi, 41
 usprawnianie, 43
 partycje geograficzne, 355
 partycjonowanie sieci, 184
 pętle zdarzeń, 73
 pionowa skalowalność, 90
 planowanie
 przepływu informacji, 357
 sterowania przepływami, 149
 Pobierz, 317
 Pomiń, 324
 pozioma skalowalność aplikacji, 109
 Pozwól Na Awarię, 196
 prawo
 Amdahla, 46
 skalowalności, 47
 problem
 awarii, 29
 obciążenia, 28
 z planowaniem wykonywania testów, 176
 programowanie
 funkcyjne, 65, 139
 logiczne, 137
 promesy, 76
 Prostý Komponent, 187
 protokół, 375
 AMQP, 286
 HTTP, 282
 przejrzystość referencyjna, 68
 Przekaż Przepływ, 298
 przepływ
 danych, 95, 131, 145
 deklaratywny, 137
 informacji, 357
 komunikatów, 279, 317

przesyłanie
 kodu, 262
 komunikatów, 89
 przetwarzanie wsadowe, 371
 przezroczystość
 lokalizacji, 83, 103, 110, 374
 wywołań, 104
 Pula Zasobów, 268

R

reaktywne
 programowanie funkcyjne, 139
 rozszerzenia, 80
 rozwiązania, 63
 reaktywny manifest, 369
 replikacja, 209, 375
 Aktywna-Aktywna, 233
 Aktywna-Pasywna, 209
 Wielokrotna-Główna, 221
 responsywność
 dla użytkowników, 70
 systemu, 30, 185
 rozczłonkowywanie, 50
 rozdzielona spójność danych, 129
 rozszerzenia reaktywne, 80
 rozszerzenie niejawnych kolejek, 327
 równoległość operacji, 140

S

Saga, 304
 Samowystarczalny Komunikat, 288
 segregacja odpowiedzialności, 133
 serwer frontowy, 36
 serwery, 184
 skala wdrożenia, 148
 skalowalność aplikacji, 90, 376
 pionowa, 120
 pozioma, 109, 120
 skrócenie czasu odpowiedzi, 41
 specyfikacja, 119
 spójność danych, 54, 129, 135
 sprzężystość
 aplikacji, 180
 centrum danych, 185
 działania aplikacji, 180
 infrastruktury, 183
 instancji i serwerów, 184
 interfejsu api, 183

sprężystość
 klastra, 184
 sieci, 184
 systemu, 179
 sterowanie
 komunikatami, 91, 374
 przepływem danych, 95, 149
 przepływem komunikatów, 149, 317
 zdarzeniami, 91
 stosowanie
 aktorów, 85
 języka domenowego, 264
 strukturalna obsługa awarii, 123
 Strumień Zdarzeń, 346
 synchroniczne
 kody wykonawcze, 168
 przysyłanie komunikatów, 93, 101
 system, 376
 reaktywny, 25
 rozproszony, 129
 niedeterministyczny, 176
 szacowanie ilości komunikatów, 148

Ś

środowisko testowe, 157

T

testowanie
 aplikacji reaktywnych, 153
 czarnej i białej skrzynki, 156
 elastyczności systemu, 179
 komponentów, 155
 komponentów rozproszonych, 176
 regularności działania usługi, 162
 responsywności systemu, 185
 sprężystości systemu, 179
 sprężystości systemu produkcyjnego, 185
 systemów niedeterministycznych, 176
 umowy sła, 164, 172
 testy, 119
 akceptacyjne, 156
 asynchroniczne, 158, 170
 integracyjne, 155
 jednostkowe, 154
 łańcuchowe, 155

tworzenie
 diagramów systemów reaktywnych, 353
 hierarchii modułów, 114, 118
 responsywnego systemu, 30
 synchronicznych kodów wykonawczych, 168

U

uniwersalne prawo skalowalności, 47
 uodpornianie na awarie, 187
 usprawnianie
 paralelizmu, 43
 równoległych pomiarów, 166
 ustalanie priorytetów cech
 wydajnościowych, 71
 utraty komunikatów, 107
 użytkownik, 376

W

współdzielenie, 140
 stanu, 141
 wstrzykiwanie błędów domenowych, 183
 wykrywanie
 braku błędów asynchronicznych, 173
 braku komunikatów, 167
 Wypożyczenie Zasobu, 253
 wysyłanie danych, 145
 wzorce, 151
 projektowania reaktywnego, 58
 przepływów komunikatów, 279
 replikacyjne, 209
 sterowania przepływem komunikatów, 317
 zarządzania stanami, 335
 zarządzania zasobami, 245
 wzorzec
 Agregator, 300
 Bezpiecznik, 202
 Biznesowy Uścisk Dłoni, 309
 Dławik, 330
 Enkapsulacja Zasobów, 245
 Jądro Błędu, 191
 Obiekt Domenowy, 336
 Odłamkowanie, 340
 Pobierz, 317
 Pomiń, 324
 Pozwól Na Awarię, 196
 Prosty Komponent, 187
 Przekaz Przepływ, 298
 Pula Zasobów, 268

Replikacja Aktywna-Aktywna, 233
Replikacja Aktywna-Pasywna, 209
Replikacja Wielokrotna-Główna, 221
Saga, 304
Samowystarczalny Komunikat, 288
Selektywny Odbiór, 84
Strumień Zdarzeń, 346
testowy Odwrócona Cebula, 178
Wypożyczenie Zasobu, 253
Zapytaj, 292
Zapytanie-Odpowiedź, 280
Zarządzana Kolejka, 321
Zarządzane Blokowanie, 272
Złożone Polecenie, 258
Źródło Zdarzeń, 343

Z

zakresy spójności danych, 135
zależności, 115
zapisywanie stanami, 335
zapobieganie powstawaniu matryc, 116
Zapytaj, 292
Zapytanie-Odpowiedź, 280
Zarządzana Kolejka, 321
Zarządzane Blokowanie, 272
zarządzanie
 stanami, 335
 zasobami, 245
 złożonością oprogramowania, 59
zasób, 375
zdarzenia, 91
 jako komunikaty, 100
zielone wątki, 72
Złożone Polecenie, 258

Ź

źródła zdarzeń, 136, 343

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



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

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

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

GRUPA WYDAWNICZA



Helion SA

SYSTEMY REAKTYWNE

— odpowiedź na wyzwania naszych czasów!

Uznane duże aplikacje internetowe, takie jak portale społecznościowe, systemy bankowe czy handlowe, działają szybko i niezawodnie. Muszą radzić sobie nawet wtedy, gdy ich niektóre części ulegną awarii. Nie mogą zawieść, gdy będzie z nich korzystać większa liczba użytkowników niż zwykle. Naturalnie, muszą być odporne na różne zagrożenia, a oprócz tego skalowalne i łatwe w rozbudowie. Tym i wielu innym wymaganiom odpowiadają systemy responsywne — które bez względu na okoliczności sprawnie przetwarzają dane wprowadzane przez użytkowników. Pomyślne wdrożenie systemu reaktywnego wymaga jednak nieco innego spojrzenia na tworzenie oprogramowania.

Niniejsza książka jest wyczerpującym wprowadzeniem do implementacji systemów reaktywnych. Opisano tu filozofię programowania reaktywnego, zasady projektowania aplikacji, wzorce projektowe i ich zastosowanie. Szczegółowo wyjaśniono, jakie problemy można rozwiązywać w ten sposób, a podane przykłady opatrzone pełnymi kodami źródłowymi. Bazując na postulatach Manifestu reaktywnego, przedstawiono metodykę budowy architektury modułowej, zasady tworzenia komunikatów, które sterują tą architekturą, opisano też potrzebne narzędzia i sposób ich wykorzystania. Nie zabrakło informacji o dobrych praktykach programowania i testowaniu aplikacji.

Najważniejsze zagadnienia:

- Manifest reaktywny i jego postulaty
- hierarchia modułów i przepływy danych
- programowanie funkcyjne i reaktywne
- obsługa awarii
- wzorce projektowe i ich stosowanie

DR ROLAND KUHN jest ekspertem w dziedzinie rozproszonych systemów obliczeniowych. Obronił doktorat w instytucie CERN w Szwajcarii. Później pracował dla Niemieckiej Agencji Kosmicznej. Obecnie tworzy systemy reaktywne.

BRIAN HANAFEE jest głównym architektem systemów w banku Wells Fargo. Wcześniej tworzył nowe produkty dla Oracle i pisał oprogramowanie do systemów wizyjnych montowanych w hełmach pilotów samolotów wojskowych.

JAMIE ALLEN jest programistą i architektem oprogramowania. Pracuje jako dyrektor techniczny projektu platformy handlowej UCP w Starbucks. Od 2008 r. tworzy w języku Scala reaktywne aplikacje dla klientów na całym świecie.

Helion helion.pl 0 801 339900 0 601 339900	<i>Sprawdź nasze szkolenia!</i> SZKOLENIA  AKADEMIA IT & BUSINESS WWW.SZKOLENIA.HELION.PL	KOD KORZYŚCI Ściągnij po więcej!  ISBN 978-83-283-3795-4  9 788328 337954
INFORMATYKA W NAJLEPSZYM WYDANIU		Cena: 67,00 zł