

John Ferguson Smart



BDD W DZIAŁANIU

STEROWANIE ZACHOWANIEM W ROZWOJU APLIKACJI

Helion 

Tytuł oryginału: BDD in Action: Behavior-Driven Development for the whole software lifecycle

Tłumaczenie: Radosław Meryk

Projekt okładki: Studio Gravite / Olsztyn

Obarek, Pokoński, Pazdrijowski, Zaprucki

ISBN: 978-83-283-1747-5

Original edition copyright © 2015 by Manning Publications Co.

All rights reserved.

Polish edition copyright © 2016 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 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 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.

Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock Images LLC.

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)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:

<ftp://ftp.helion.pl/przyklady/bdddzi.zip>

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

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

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

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

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

Spis treści

<i>Słowo wstępne</i>	11
<i>Przedmowa</i>	15
<i>Podziękowania</i>	17
<i>O tej książce</i>	19

CZĘŚĆ I. PIERWSZE KROKI 23

<i>Rozdział 1. Budowanie oprogramowania, które sprawia różnicę</i>	25
1.1. BDD z wysokości 15 kilometrów	27
1.2. Jakie problemy próbujemy rozwiązywać?	29
1.2.1. Właściwe budowanie oprogramowania	30
1.2.2. Budowanie właściwego oprogramowania	32
1.2.3. Ograniczenia wiedzy — radzenie sobie z informacją niepewną	32
1.3. Wprowadzenie do programowania sterowanego zachowaniami	34
1.3.1. BDD pierwotnie zaprojektowano jako ulepszoną wersję TDD	35
1.3.2. Techniki BDD również sprawdzają się jako narzędzia analizy wymagań	37
1.3.3. Zasady i praktyki BDD	38
1.4. Korzyści z BDD	52
1.4.1. Mniejsze marnotrawstwo	52
1.4.2. Niższe koszty	52
1.4.3. Łatwiejsze i bezpieczniejsze zmiany	53
1.4.4. Szybsze publikacje	53
1.5. Wady i potencjalne problemy związane ze stosowaniem praktyk BDD	53
1.5.1. Stosowanie praktyk BDD wymaga dużego zaangażowania i współpracy	53
1.5.2. Praktyki BDD sprawdzają się najlepiej w kontekście metodologii Agile lub innej metodologii iteracyjnej	53
1.5.3. Praktyki BDD nie sprawdzają się dobrze w projektach typu silos	54
1.5.4. Źle napisane testy mogą prowadzić do wyższych kosztów utrzymania	54
1.6. Podsumowanie	54
<i>Rozdział 2. BDD z lotu ptaka</i>	57
2.1. Wprowadzenie w tematykę aplikacji rozkładu jazdy pociągów	58
2.2. Określenie korzyści ze stosowania proponowanej aplikacji	60
2.3. Analiza wymagań — odkrywanie i zrozumienie funkcji	60
2.3.1. Opisywanie funkcji	60
2.3.2. Podział cech funkcjonalnych na historyjki	62
2.3.3. Ilustrowanie historyjek przykładami	63

- 2.4. Implementacja — budowanie i dostarczanie cech funkcjonalnych 64
 - 2.4.1. *Od przykładów do kryteriów akceptacji* 64
 - 2.4.2. *Konfigurowanie narzędzi Maven i Git* 65
 - 2.4.3. *Specyfikacje wykonywalne — automatyzacja kryteriów akceptacji* 67
 - 2.4.4. *Automatyczne testy — implementacja kryteriów akceptacji* 72
 - 2.4.5. *Testy jako dynamiczna dokumentacja* 81
- 2.5. Utrzymanie 81
- 2.6. Podsumowanie 85

CZĘŚĆ II. CZEGO CHCĘ? DEFINIOWANIE WYMAGAŃ Z WYKORZYSTANIEM BDD87

Rozdział 3. Zrozumieć cele biznesowe. Wstrzykiwanie cech funkcjonalnych i związane z tym techniki 89

- 3.1. Poznajemy firmę Flying High 91
- 3.2. Wstrzykiwanie funkcji 92
 - 3.2.1. *Wyszukiwanie wartości* 93
 - 3.2.2. *Wstrzykiwanie cech funkcjonalnych* 93
 - 3.2.3. *Wskazanie przykładów* 94
 - 3.2.4. *Podsumowanie* 94
- 3.3. Co chcesz osiągnąć? Zacznij od wizji 96
 - 3.3.1. *Formuła wizji* 97
 - 3.3.2. *Korzystanie z szablonów formuły wizji* 98
- 3.4. W jaki sposób firma skorzysta na projekcie? Identyfikowanie celów biznesowych 99
 - 3.4.1. *Pisanie dobrych celów biznesowych* 100
 - 3.4.2. *Pokaż mi pieniądze — cele biznesowe a przychody* 101
 - 3.4.3. *Ściąganie ze „stosu dłaczego” — uściślanie celów biznesowych* 103
- 3.5. Mapowanie wpływu — podejście wizualne 106
- 3.6. Kto na tym skorzysta? Identyfikowanie interesariuszy i ich potrzeb 110
- 3.7. Co trzeba zbudować? Identyfikowanie zdolności 112
- 3.8. Jakie cechy funkcjonalne zapewnią największy wskaźnik ROI?
Model dopasowania do celów 114
 - 3.8.1. *Cechy wyróżniające* 116
 - 3.8.2. *Cechy równoważne* 116
 - 3.8.3. *Cechy partnerskie* 117
 - 3.8.4. *Cechy o minimalnym wpływie* 117
- 3.9. Podsumowanie 117

Rozdział 4. Definiowanie i ilustrowanie cech funkcjonalnych 119

- 4.1. Co to jest cecha funkcjonalna? 120
 - 4.1.1. *Cechy funkcjonalne dostarczają zdolności* 122
 - 4.1.2. *Cechy funkcjonalne można podzielić na łatwiejsze do zarządzania fragmenty* 126
 - 4.1.3. *Cecha funkcjonalna może być opisana za pomocą jednej lub kilku historyjek użytkowników* 128
 - 4.1.4. *Cecha funkcjonalna nie jest historyjką użytkownika* 131

- 4.1.5. *Eposy to naprawdę duże historyjki użytkownika* 132
- 4.1.6. *Nie wszystko pasuje do hierarchii* 133
- 4.2. Ilustrowanie cech funkcjonalnych przykładami 134
- 4.3. Realne opcje — podejmij zobowiązania dopiero wtedy, kiedy musisz 140
 - 4.3.1. *Opcje mają wartość* 141
 - 4.3.2. *Opcje wygasają* 143
 - 4.3.3. *Nigdy nie zobowiązuje się zbyt wcześnie, jeśli nie wiesz dlaczego* 143
- 4.4. Celowe odkrywanie 144
- 4.5. Od przykładów do działającego oprogramowania — szerszy obraz 145
- 4.6. Podsumowanie 146

Rozdział 5. Od przykładów do wykonywalnych specyfikacji 149

- 5.1. Przekształcanie konkretnych przykładów na wykonywalne scenariusze 151
- 5.2. Pisanie wykonywalnych scenariuszy 154
 - 5.2.1. *Plik cech funkcjonalnych zawiera tytuł i opis* 154
 - 5.2.2. *Opisywanie scenariuszy* 156
 - 5.2.3. *Struktura „Zakładając... Gdy... Wtedy”* 157
 - 5.2.4. *Uzupełniające słowa kluczowe* 158
 - 5.2.5. *Komentarze* 159
- 5.3. Wykorzystanie tabel w scenariuszach 160
 - 5.3.1. *Używanie tabel w pojedynczych krokach* 160
 - 5.3.2. *Wykorzystanie tabel przykładów* 161
- 5.4. Scenariusze ekspresywne — wzorce i antywzorce 165
 - 5.4.1. *Pisanie ekspresywnych kroków Zakładając* 165
 - 5.4.2. *Pisanie ekspresywnych kroków Gdy* 166
 - 5.4.3. *Pisanie ekspresywnych kroków Wtedy* 167
 - 5.4.4. *Podawanie tła i kontekstu* 168
 - 5.4.5. *Unikanie zależności między scenariuszami* 170
- 5.5. Organizowanie scenariuszy przy użyciu plików cech funkcjonalnych i tagów 171
 - 5.5.1. *Scenariusze zapisuje się w plikach opisu cech funkcjonalnych* 172
 - 5.5.2. *Plik opisu cechy funkcjonalnej może zawierać jeden lub więcej scenariuszy* 172
 - 5.5.3. *Organizowanie plików opisu cech funkcjonalnych* 173
 - 5.5.4. *Opisywanie scenariuszy za pomocą tagów* 174
- 5.6. Podsumowanie 176

Rozdział 6. Automatyzacja scenariuszy 179

- 6.1. Wprowadzenie do automatyzowania scenariuszy 182
 - 6.1.1. *Definicje kroków interpretują tekst scenariuszy* 182
 - 6.1.2. *Zachowaj prostotę metod definicji kroków* 184
- 6.2. Implementacja definicji kroków — zasady ogólne 186
 - 6.2.1. *Instalowanie narzędzi BDD* 186
 - 6.2.2. *Implementacja definicji kroków* 187
 - 6.2.3. *Przekazywanie parametrów do implementacji kroków* 187
 - 6.2.4. *Utrzymywanie stanu pomiędzy krokami* 188
 - 6.2.5. *Wykorzystywanie danych tabelarycznych z definicji kroków* 190
 - 6.2.6. *Implementacja scenariuszy bazujących na przykładach* 191
 - 6.2.7. *Wyniki scenariusza* 192

- 6.3. Bardziej efektywna implementacja BDD z wykorzystaniem narzędzia Thucydides 193
- 6.4. Automatyzowanie scenariuszy w Javie z wykorzystaniem JBehave 194
 - 6.4.1. Instalowanie i konfigurowanie narzędzia JBehave 194
 - 6.4.2. Definicje kroków JBehave 195
 - 6.4.3. Współdzielenie danych pomiędzy krokami 197
 - 6.4.4. Przekazywanie tabel do kroków 198
 - 6.4.5. Definicje kroków dla tabel przykładów 199
 - 6.4.6. Warianty wzorców 200
 - 6.4.7. Niepowodzenia i błędy w wynikach scenariuszy 200
- 6.5. Automatyzowanie scenariuszy w Javie przy użyciu narzędzia Cucumber-JVM 202
 - 6.5.1. Konfiguracja i struktura projektu Cucumber-JVM 202
 - 6.5.2. Definicje kroków Cucumber-JVM 203
 - 6.5.3. Warianty wzorców 204
 - 6.5.4. Przekazywanie tabel do definicji kroków 205
 - 6.5.5. Definicje kroków dla tabel przykładów 206
 - 6.5.6. Współdzielenie danych pomiędzy krokami 206
 - 6.5.7. Kroki oczekujące i wyniki kroków 207
- 6.6. Automatyzowanie scenariuszy w Pythonie z wykorzystaniem Behave 207
 - 6.6.1. Instalacja systemu Behave 208
 - 6.6.2. Struktura projektu Behave 208
 - 6.6.3. Definicje kroków Behave 209
 - 6.6.4. Łączenie kroków 209
 - 6.6.5. Definicje kroków z wykorzystaniem osadzonych tabel 210
 - 6.6.6. Definicje kroków dla tabel przykładów 210
 - 6.6.7. Uruchamianie scenariuszy w Behave 210
- 6.7. Automatyzowanie scenariuszy w .NET z wykorzystaniem SpecFlow 211
 - 6.7.1. Konfigurowanie SpecFlow 211
 - 6.7.2. Dodawanie plików opisu cech funkcjonalnych 211
 - 6.7.3. Uruchamianie scenariuszy 213
 - 6.7.4. Definicje kroków w SpecFlow 213
 - 6.7.5. Współdzielenie danych pomiędzy krokami 214
 - 6.7.6. Definicje kroków z wykorzystaniem tabel przykładów 215
- 6.8. Automatyzowanie scenariuszy w JavaScript z wykorzystaniem systemu Cucumber-JS 216
 - 6.8.1. Konfigurowanie systemu Cucumber-JS 216
 - 6.8.2. Pisanie plików opisu funkcji w Cucumber-JS 217
 - 6.8.3. Implementowanie kroków 218
 - 6.8.4. Uruchamianie scenariuszy 219
- 6.9. Podsumowanie 220

CZĘŚĆ III. JAK TO ZBUDOWAĆ? KODOWANIE ZGODNE Z BDD219

Rozdział 7. Od wykonywalnych specyfikacji do solidnych automatycznych testów akceptacyjnych 223

- 7.1. Pisanie niezawodnych testów akceptacji 225
- 7.2. Automatyzowanie procesu konfiguracji testu 228

- 7.2.1. *Inicjowanie bazy danych przed każdym testem* 228
- 7.2.2. *Inicjowanie bazy danych na początku zestawu testów* 229
- 7.2.3. *Korzystanie z haków inicjalizacji* 229
- 7.2.4. *Konfigurowanie danych specyficznych dla scenariusza* 233
- 7.2.5. *Użycie person i znanych encji* 235
- 7.3. *Oddzielenie warstwy „co” od warstwy „jak”* 237
 - 7.3.1. *Warstwa reguł biznesowych opisuje oczekiwane rezultaty* 238
 - 7.3.2. *Warstwa przepływu pracy opisuje działania użytkownika* 239
 - 7.3.3. *Warstwa techniczna realizuje interakcje z systemem* 241
 - 7.3.4. *Ile warstw?* 242
- 7.4. *Podsumowanie* 243

Rozdział 8. Automatyzacja kryteriów akceptacji dla warstwy interfejsu użytkownika 245

- 8.1. *Kiedy i jak należy testować interfejs użytkownika?* 247
 - 8.1.1. *Zagrożenie zbyt wieloma testami webowymi* 247
 - 8.1.2. *Testy webowe z przeglądarkami w trybie headless* 248
 - 8.1.3. *Ile testów webowych naprawdę potrzebujemy?* 250
- 8.2. *Automatyzowanie webowych kryteriów akceptacji z wykorzystaniem programu Selenium WebDriver* 251
 - 8.2.1. *Pierwsze kroki z WebDriver w Javie* 252
 - 8.2.2. *Identyfikacja elementów strony WWW* 255
 - 8.2.3. *Interakcje z elementami stron WWW* 263
 - 8.2.4. *Praca ze stronami asynchronicznymi i testowanie aplikacji AJAX* 265
 - 8.2.5. *Pisanie aplikacji webowych „przyjaznych dla testów”* 267
- 8.3. *Korzystanie z obiektów stron w celu poprawy czytelności testów* 267
 - 8.3.1. *Wprowadzenie do wzorca Obiekty stron* 268
 - 8.3.2. *Pisanie dobrze zaprojektowanych obiektów stron* 273
 - 8.3.3. *Korzystanie z bibliotek rozszerzających bibliotekę WebDriver* 279
- 8.4. *Podsumowanie* 281

Rozdział 9. Automatyzacja kryteriów akceptacji dla wymagań niekorzystających z UI 283

- 9.1. *Równowaga pomiędzy testami akceptacyjnymi z wykorzystaniem UI i bez UI* 285
- 9.2. *Kiedy używać testów akceptacji bez pośrednictwa UI* 286
- 9.3. *Typy automatycznych testów akceptacji niekorzystających z UI* 290
 - 9.3.1. *Testowanie z wykorzystaniem warstwy kontrolera* 291
 - 9.3.2. *Bezpośrednie testowanie logiki biznesowej* 295
 - 9.3.3. *Testowanie warstwy usług* 299
- 9.4. *Definiowanie i testowanie wymagań niefunkcyjnych* 304
- 9.5. *Odkrywanie projektu* 306
- 9.6. *Podsumowanie* 308

Rozdział 10. BDD a testy jednostkowe 309

- 10.1. BDD, TDD a testy jednostkowe 310
 - 10.1.1. *BDD dotyczy pisania specyfikacji, a nie testów, na wszystkich poziomach* 312
 - 10.1.2. *BDD bazuje na ugruntowanych praktykach TDD* 313
 - 10.1.3. *Narzędzia BDD do testów jednostkowych* 313
- 10.2. Od kryteriów akceptacji do zaimplementowanych cech funkcjonalnych 313
 - 10.2.1. *BDD sprzyja stosowaniu podejścia „z zewnątrz do wewnątrz”* 314
 - 10.2.2. *Zaczynamy od wysokopoziomowego kryterium akceptacji* 316
 - 10.2.3. *Automatyzacja scenariuszy kryteriów akceptacji* 317
 - 10.2.4. *Implementacja definicji kroków* 317
 - 10.2.5. *Zrozumienie modelu domeny* 318
 - 10.2.6. *Pisanie kodu, który chcielibyśmy mieć* 319
 - 10.2.7. *Wykorzystanie kodu definicji kroku do wyspecyfikowania i zaimplementowania kodu aplikacji* 319
 - 10.2.8. *W jaki sposób stosowanie praktyk BDD pomogło?* 324
- 10.3. Analiza niskopoziomowych wymagań, odkrywanie projektu i implementacja bardziej złożonych funkcjonalności 325
 - 10.3.1. *Wykorzystanie kodu definicji kroku do analizy niskopoziomowego projektu* 326
 - 10.3.2. *Praca z tabelami przykładów* 328
 - 10.3.3. *Odkrywanie nowych klas i usług w miarę implementowania kodu produkcyjnego* 330
 - 10.3.4. *Natychmiastowa implementacja prostych klas lub metod* 331
 - 10.3.5. *Wykorzystanie minimalnej implementacji* 331
 - 10.3.6. *Wykorzystanie namiastek i makiet w celu odroczenia implementacji bardziej złożonego kodu* 332
 - 10.3.7. *Rozwijanie niskopoziomowych specyfikacji technicznych* 333
- 10.4. Narzędzia, dzięki którym testy jednostkowe BDD stają się łatwiejsze 336
 - 10.4.1. *Stosowanie praktyk BDD z tradycyjnymi narzędziami testów jednostkowych* 336
 - 10.4.2. *Pisanie specyfikacji, a nie testów — rodzina RSpec* 339
 - 10.4.3. *Pisanie bardziej ekspresywnych specyfikacji z wykorzystaniem narzędzi Spock albo Spec2* 343
- 10.5. Używanie wykonywalnych specyfikacji jako dynamicznej dokumentacji 345
 - 10.5.1. *Używanie płynnego kodowania w celu poprawy czytelności* 346
 - 10.5.2. *Płynne asercje w języku JavaScript* 347
 - 10.5.3. *Płynne asercje w językach statycznych* 347
- 10.6. Podsumowanie 349

CZĘŚĆ IV. ZAAWANSOWANE ASPEKTY BDD349**Rozdział 11. Dynamiczna dokumentacja — raportowanie a zarządzanie projektem 353**

- 11.1. Dynamiczna dokumentacja — widok wysokopoziomowy 354
- 11.2. Czy osiągnęliśmy cel? Raporty dotyczące gotowości i pokrycia cech funkcjonalnych 356

- 11.2.1. *Gotowość cech funkcjonalnych — jakie cechy są gotowe do dostarczenia* 356
- 11.2.2. *Pokrycie cechy funkcjonalnej — jakie wymagania zostały spełnione* 357
- 11.3. Integracja z cyfrowym rejestrem projektu 360
- 11.4. Organizowanie dynamicznej dokumentacji 362
 - 11.4.1. *Organizowanie dynamicznej dokumentacji według wysokopoziomowych wymagań* 363
 - 11.4.2. *Organizowanie dynamicznej dokumentacji z wykorzystaniem tagów* 364
 - 11.4.3. *Dynamiczna dokumentacja do tworzenia raportów na temat publikacji oprogramowania* 364
- 11.5. Dostarczanie dokumentacji w luźniejszej formie 366
- 11.6. Techniczna dynamiczna dokumentacja 368
 - 11.6.1. *Testy jednostkowe jako dynamiczna dokumentacja* 369
 - 11.6.2. *Dynamiczna dokumentacja dla starszych aplikacji* 371
- 11.7. Podsumowanie 372

Rozdział 12. BDD w procesie budowania 373

- 12.1. Wykonywalne specyfikacje powinny być częścią automatycznego procesu budowy 374
 - 12.1.1. *Każda specyfikacja powinna być samowystarczalna* 375
 - 12.1.2. *Wykonywalne specyfikacje powinny być przechowywane w systemie kontroli wersji* 376
 - 12.1.3. *Powinna istnieć możliwość uruchomienia wykonywalnych specyfikacji z wiersza polecenia* 377
- 12.2. Ciągła integracja przyspiesza cykl sprzężenia zwrotnego 378
- 12.3. Ciągłe dostawy — każda kompilacja jest potencjalną wersją do opublikowania 380
- 12.4. Strategia ciągłej integracji w celu wdrażania dynamicznej dokumentacji 383
 - 12.4.1. *Publikowanie dynamicznej dokumentacji na serwerze kompilacji* 384
 - 12.4.2. *Publikowanie dynamicznej dokumentacji na dedykowanym serwerze WWW* 385
- 12.5. Szybsze zautomatyzowane kryteria akceptacji 385
 - 12.5.1. *Uruchamianie równoległych testów akceptacyjnych w obrębie zautomatyzowanego procesu budowy* 386
 - 12.5.2. *Uruchamianie równoległych testów na wielu maszynach* 388
 - 12.5.3. *Uruchamianie równoległych testów webowych z wykorzystaniem Selenium Grid* 391
- 12.6. Podsumowanie 395
- 12.7. Ostatnie słowa 396

Skorowidz 399

BDD z lotu ptaka



W tym rozdziale:

- Wyczerpujący przegląd praktyk BDD w akcji.
- Odkrywanie cech funkcjonalnych i opisywanie ich za pomocą historyjek i przykładów.
- Wykorzystanie wykonywalnych specyfikacji do szczegółowego opisywania cech.
- Wykorzystanie niskopoziomowych praktyk BDD do implementacji cech funkcjonalnych.
- Wykorzystanie wyników testów BDD jako dynamicznej dokumentacji.
- Wykorzystanie dynamicznej dokumentacji w celu wsparcia bieżącego utrzymania aplikacji.

W tym rozdziale przeanalizujemy konkretny przykład wykorzystania praktyk BDD w rzeczywistym projekcie. Jak dowiedzieliśmy się z poprzedniego rozdziału, stosowanie praktyk BDD wiąże się z zaangażowaniem zespołu rozwojowego w czasie trwania całego projektu w rozmowy z klientem, a także z wykorzystaniem przykładów do budowania bardziej konkretnego i jednoznacznego zrozumienia realnych potrzeb firmy. Specyfikacje są budowane w formie wykonywalnej. Można je wykorzystać do określenia wymagań dotyczących oprogramowania, kierowania ich implementacją oraz weryfikowania poprawności dostarczanego produktu. Można również stosować te techniki podczas bardziej wysokopoziomowej analizy wymagań. Ich wykorzystanie pozwala skupić się na tych możliwościach i funkcjach aplikacji, które przyniosą rzeczywistą wartość dla biznesu.

Kluczowym elementem tej praktyki jest określanie scenariuszy lub konkretnych przykładów sposobu działania określonej funkcji lub zdefiniowanej historyjki użytkownika. Scenariusze te pomagają zweryfikować poprawność rozumienia problemu i rozszerzyć wiedzę na jego temat. Są one również doskonałym narzędziem komunikacji. Stanowią fundament kryteriów akceptacji, które można później zintegrować z procesem kompilacji w formie automatycznych testów akceptacyjnych. W połączeniu z automatycznymi testami akceptacyjnymi te przykłady sterują procesem rozwoju. Pomagają projektantom przygotować projekt skutecznego i funkcjonalnego interfejsu użytkownika oraz wspierają deweloperów w odkrywaniu podstawowych zachowań, które trzeba zaimplementować, aby dostarczyć wymagane cechy funkcjonalne.

W pozostałej części tego rozdziału przyjrzymy się praktycznemu przykładowi zastosowania tego procesu. W opisie uwzględnimy aspekty całego cyklu rozwoju — od analizy biznesowej aż do implementacji, testowania i utrzymania kodu.

2.1. Wprowadzenie w tematykę aplikacji rozkładu jazdy pociągów

Aby zilustrować przykładem tematykę omawianą w niniejszym rozdziale, założmy, że pracujesz dla dużej rządowej agencji transportu publicznego. Poproszono Cię, abyś pokierował niewielkim zespołem mającym za zadanie stworzenie usługi, która dostarczy danych na temat rozkładu jazdy pociągów oraz w czasie rzeczywistym zapewni dane o opóźnieniach, pracach nad trakcją itp. Usługa ta ma być dostępna dla różnych aplikacji mobilnych używanych przez osoby dojeżdżające do pracy. Sieć kolejową, którą wykorzystamy w przykładzie, pokazano na rysunku 2.1.

W dziale właśnie wdrożono praktyki Agile i BDD, zaczynasz więc od rozmowy z kluczowymi interesariuszami, aby upewnić się, że Ty i Twój zespół macie czytelny obraz celów biznesowych sterujących projektem. To pomoże zespołowi dostarczyć lepszą, bardziej ukierunkowaną aplikację.

Gdy cele biznesowe zostaną zrozumiane i wyartykułowane, trzeba będzie pracować razem z analitykami biznesowymi i przedstawicielami biznesowymi w celu podjęcia decyzji dotyczącej cech funkcjonalnych oprogramowania, za pomocą których będzie można osiągnąć te cele. Cechy te są wymaganiami wysokiego poziomu w postaci: „zapewnienie podróżnym optymalnej trasy pomiędzy stacjami” lub „powiadomianie podróżnych o opóźnieniach pociągu”.

Prawdopodobnie nie uda się zrealizować tak obszernych cech funkcjonalnych w jednym kawałku, dlatego trzeba je rozbić na mniejsze jednostki, znane wśród zespołów praktykujących metodykę Agile jako *historyjki*. Historyjki mogą obejmować takie działania, jak „znajdź optymalną trasę między stacjami na tej samej linii” lub „znajdź optymalną trasę między stacjami na różnych liniach”.

Przystępując do implementacji historyjki, spotkałeś się z analitykiem biznesowym, programistą i testerem, aby opisać tę historyjkę za pomocą konkretnych przykładów. Wiele z tych przykładów zostało wcześniej omówionych z przedstawicielami biznesu. Przykłady te staną się **kryteriami akceptacji** dla historyjki. Są one wyrażone w formalnym stylu BDD, który później można zautomatyzować:

2.2. Określenie korzyści ze stosowania proponowanej aplikacji

Jednym z kluczowych celów stosowania praktyk BDD jest zadbanie o to, aby wszyscy uczestnicy projektu dokładnie rozumieli, co projekt stara się zrealizować, a także znali jego podstawowe cele biznesowe. To samo w sobie sprowadza się do zapewnienia zrealizowania tych celów przez aplikację.

Można to osiągnąć w wyniku współpracy z użytkownikami i innymi interesariuszami w celu zdefiniowania lub wyjaśnienia wysokopoziomowych celów biznesowych aplikacji. Cele te powinny dostarczać zwięzłej wizji tego, co potrzebujemy stworzyć. Cele biznesowe dotyczą dostarczania wartości, powszechnie stosuje się więc wyrażanie ich w kategoriach zwiększonych lub bezpiecznych dochodów lub obniżenia kosztów.

W tym przypadku celem aplikacji, którą budujemy, jest dostarczenie podróznym rozkładów pociągów i aktualizacji w czasie rzeczywistym. Podstawowy cel biznesowy tej aplikacji można wyrazić w następujący sposób:

Zwiększenie przychodów ze sprzedaży biletów dzięki ułatwieniu podróży pociągiem
 ↳ i zwiększenie wydajności takiego sposobu podróżowania

Zrozumienie i zdefiniowanie celów aplikacji znacznie ułatwia ustalenie względnej wartości planowanej cechy funkcjonalnej. Na przykład cecha funkcjonalna, która powiadamia podróznym o spóźnieniu ich pociągu, przyczynia się do ogólnego celu, ponieważ daje podróznym możliwość odpowiedniej zmiany planów. Z drugiej strony, cecha, która pozwala podróznym oceniać poszczególne stacje kolejowe, może być uznana za niezbyt wartościową.

2.3. Analiza wymagań – odkrywanie i zrozumienie funkcji

Kiedy uświadomimy sobie wysokopoziomowe cele aplikacji, możemy zacząć współpracę z interesariuszami, aby określić dokładnie, czego potrzebują, żeby osiągnąć te cele. Zazwyczaj polega to na zdefiniowaniu zestawu cech funkcjonalnych, które należy zaimplementować w aplikacji w celu dostarczenia wartości, których poszukujemy.

Dla potrzeb niniejszego rozdziału założmy, że uzgodniliśmy z zainteresowanymi stronami następujące kluczowe cechy funkcjonalne:

- Zaproponowanie podróznym optymalnej trasy.
- Zapewnienie podróznym w czasie rzeczywistym informacji na temat opóźnień w formie zestawienia.
- Umożliwienie podróznym nagrywania swoich ulubionych podróży.
- Powiadamianie podróznym o opóźnieniach ich pociągu.

Przyjrzyjmy się, jak można opisać niektóre z tych cech.

2.3.1. Opisywanie funkcji

Po uzyskaniu ogólnego pojęcia o cechach funkcjonalnych, które chcemy dostarczyć, należy opisać je bardziej szczegółowo. Istnieje wiele sposobów opisywania wymagań.

Zespoły stosujące metodologię Agile zazwyczaj piszą krótki zarys wymagania w formacie na tyle zwięzłym, aby zmieścił się na pojedynczej karcie katalogowej¹. Zespoły korzystające z praktyk BDD często używają następującego formatu:

Jakie cele biznesowe staramy się osiągnąć? ①

W celu <osiągnięcia celu biznesowego lub dostarczenia wartości biznesowej> ←

Jako <interesariusz> ← **② Kto tego potrzebuje?**

Chcę <czegoś> ← **③ Co trzeba zrobić, aby umożliwić osiągnięcie tego celu?**

Kolejność w tym przypadku ma znaczenie. Podczas planowania cechy funkcjonalnej i historii głównym celem powinno być dostarczenie wartości biznesowej. Należy rozpocząć od określenia wartości biznesowej, jaką staramy się dostarczyć ①, następnie podajemy, kto potrzebuje funkcji, którą proponujemy ②, i na koniec wymieniamy cechę funkcjonalną, która będzie wspomagać osiągnięcie tego celu ③.

Taki sposób opisywania pomaga uzyskać pewność, że każda cecha funkcjonalna aktywnie przyczynia się do osiągnięcia celu biznesowego, a to zmniejsza ryzyko wystąpienia niekontrolowanego rozrastania się zakresu projektu (ang. *scope creep*). Taki opis spełnia również rolę wygodnego przypomnienia powodów, dla których implementujemy tę cechę. Na przykład można powiedzieć coś takiego:

W celu bardziej efektywnego planowania podróży ← **Jaką wartość biznesową staramy się dostarczyć?**

Jako podróżny ← **Kto jest zainteresowany taką cechą?**

Chcę znać optymalną trasę pomiędzy dwoma stacjami ← **Co cecha funkcjonalna będzie robić?**

To nie jest jedyna możliwość. Wiele zespołów używa szablonów popularnych we wcześniejszych podejściach Agile:

① Kto skorzysta z tej cechy; kto jej chce?

Jako <interesariusz> ←

② Co cecha funkcjonalna robi?

Chcę <czegoś> ←

③ Jakie wartości biznesowe interesariusze uzyskają przez tę cechę funkcjonalną?

Tak, abym <mógł osiągnąć pewien cel biznesowy> ←

Ta odmiana opisu ma pomóc deweloperom zrozumieć kontekst wymagania w kategoriach tego, kto będzie używał cechy funkcjonalnej i czego od niej oczekuje. Interesariusz ① odnosi się do osoby używającej cechy funkcjonalnej ② lub osoby zainteresowanej wynikiem jej działania. Cel biznesowy ③ identyfikuje powód, dla którego ta cecha jest potrzebna, i wartość, jaką ma ona zapewnić. Odpowiednikiem opisu cechy funkcjonalnej podanego wcześniej może być następujący opis:

Jako podróżny
Chcę znać najlepszy sposób podróżowania pomiędzy dwoma stacjami
Tak, abym mógł szybko dotrzeć do celu podróży

Oba te formaty są wygodnymi konwencjami, ale nie istnieje obowiązek wybrania jednego lub drugiego formatu, pod warunkiem że pamiętamy o jasnym wyrażeniu korzyści

¹ Te karty katalogowe mogą być później użyte do zaplanowania i wizualizacji postępów.

² Taki format został pierwotnie zaproponowany przez Chrisa Mattsa w kontekście wstrzykiwania funkcjonalności — zagadnienia, któremu przyjrzymy się w następnym rozdziale.

biznesowych. Na przykład, niektórzy doświadczeni praktycy chętnie korzystają z notacji „W celu... Jako... Chcę” do opisanía takich cech wysokopoziomowych, w których kładziemy nacisk na wartości biznesowe, jakie system powinien dostarczyć, natomiast do określenia bardziej szczegółowych historyjek użytkowników w ramach cechy funkcjonalnej, gdy historie dotyczą wyraźnie dostarczania wartości dla poszczególnych użytkowników w ramach tej cechy, stosując konwencję „Jako... Chcę... Tak, aby”.

2.3.2. Podział cech funkcjonalnych na historyjki

Cecha funkcjonalna czasami jest sformułowana wystarczająco szczegółowo, aby można było przystąpić do jej realizacji od razu, ale często trzeba ją podzielić na mniejsze fragmenty. W projektach Agile większe cechy funkcjonalne często są dzielone na historyjki użytkownika. Każda historyjka dotyczy odrębnego aspektu problemu i jest wystarczająco zwięzła, aby można ją było dostarczyć w pojedynczej iteracji.

Na przykład funkcja „zapropozowanie podróznym optymalnej trasy” może być zbyt duża, aby stworzyć ją za jednym razem (z punktu widzenia deweloperów znalezienie trasy obejmujące łączenie kilku pociągów to skomplikowana operacja). Ponadto, być może przed zbudowaniem całej cechy funkcjonalnej chcielibyśmy uzyskać jakieś opinie na temat projektu interfejsu użytkownika. Tę cechę można podzielić na mniejsze historyjki, na przykład:

- Znajdź optymalną trasę pomiędzy stacjami na tej samej linii.
- Dowiedz się, o której godzinie odjeżdżają następne pociągi do stacji docelowej.
- Znajdź optymalną trasę pomiędzy stacjami na różnych liniach.

Można opisać te historyjki trochę bardziej szczegółowo, stosując ten sam format, którego używaliśmy w odniesieniu do funkcji:

Historyjka: Znajdź optymalną trasę pomiędzy stacjami na tej samej linii.

W celu dotarcia do miejsca docelowego na czas

Jako podróżny

Chcę się dowiedzieć, do jakiego pociągu powinienem wsiąść

Historyjka: Dowiedz się, o której godzinie odjeżdżają następne pociągi do stacji docelowej

W celu bardziej efektywnego planowania podróży

Jako podróżny

Chcę się dowiedzieć, jakie następne pociągi odjeżdżają do mojej stacji docelowej

Historyjka: Znajdź optymalną trasę pomiędzy stacjami na różnych liniach

W celu dotarcia do miejsca docelowego na czas

Jako podróżny

Chcę się dowiedzieć, do jakiego pociągu powinienem wsiąść

Oraz uzyskać szczegółowe informacje na temat potrzebnych połączeń

Zwróćmy uwagę, że pokazana lista historyjek w żadnym razie nie jest sztywnym zbiorem specyfikacji, pod którym powinni się podpisać użytkownicy i deweloperzy. Definiowanie historyjek jest dynamicznym, iteracyjnym procesem mającym na celu ułatwienie komunikacji i zapewnienie wspólnego zrozumienia przestrzeni problemu. Podczas implementowania poszczególnych historyjek możemy uzyskać opinie od interesariuszy. Na podstawie tych opinii można uściślić inne historyjki, usuwać niektóre z nich bądź

dodać nowe, które w inny sposób mogą przyczynić się do osiągnięcia celów biznesowych. Odkrywanie funkcji i tworzenie historyjek jest ciągłym procesem poznawania przestrzeni problemu.

2.3.3. Ilustrowanie historyjek przykładami

Po zidentyfikowaniu pewnego zbioru wartościowych cech funkcjonalnych i historyjek możemy przystąpić do analizowania ich w sposób bardziej szczegółowy. Bardzo skutecznym sposobem, aby to zrobić, jest poproszenie użytkowników i innych interesariuszy o podanie konkretnych przykładów.

Kiedy użytkownik pyta nas o cechę funkcjonalną, często od razu zaczynamy budować koncepcyjny model problemu, który należy rozwiązać. Jeśli będziemy postępować w ten sposób, nasze zrozumienie problemu może być łatwo zakłócone przez niejawne i niewypowiedziane założenia. To może doprowadzić do stworzenia niedokładnego modelu mentalnego, a następnie do błędnej implementacji. Poproszenie interesariuszy o konkretne przykłady tego, co mają na myśli, to świetny sposób sprawdzenia i potwierdzenia właściwego zrozumienia problemu.

Na przykład pomiędzy Tobą a Jerzym, ekspertem w dziedzinie sieci kolejowej, mogła odbyć się następująca rozmowa³:

Ty: Czy możesz podać mi przykład pasażera podróżującego pomiędzy dwoma stacjami?

Jerzy: Pewnie. Na przykład ze stacji Parramatta do Town Hall.

Ty: Jak mogłaby wyglądać taka trasa?

Jerzy: Pasażer musiałby skorzystać z linii Western. To bardzo często używana linia. Na godzinę kursuje na niej od 8 do 16 pociągów, w zależności od pory dnia. Po prostu trzeba zaproponować następny planowy odjazd na tej linii.

Ty: Czy możesz podać mi przykład podróży, w której pasażer ma do wyboru więcej niż jedną linię?

Jerzy: Tak, pasażer podróżujący ze stacji Epping do Central może wybrać linię Epping lub Northern. Czas podróży waha się od około 27 minut do około 43 minut, a pociągi na tych liniach zwykle przyjeżdżają co kilka minut, więc musimy przekazać pasażerom wystarczająco dużo informacji na temat godzin odjazdów i przyjazdów pociągów jeżdżących na obu tych liniach.

Nawet w tym prostym przykładzie widać, że istnieją pewne subtelności. Propozycja trasy nie zawsze sprowadza się do prostej informacji na temat godziny odjazdu następnego pociągu. Trzeba przekazać pasażerowi szczegółowe informacje na temat godzin odjazdów i przyjazdów wszystkich zaplanowanych następnych pociągów.

³ Aby śledzić przytaczane przykłady, można odnieść się do mapy zaprezentowanej na rysunku 2.1.

2.4. Implementacja – budowanie i dostarczanie cech funkcjonalnych

Po zidentyfikowaniu cech, których aplikacja potrzebuje, należy je zbudować. W tym podrozdziale przyjrzymy się podstawowemu cyklowi życia BDD.

Dowiemy się, w jaki sposób na podstawie przykładów skoncentrowanych wokół potrzeb biznesowych, które omawialiśmy w poprzednim podrozdziale, można stworzyć wykonywalne specyfikacje. Pokażemy też, jak można zautomatyzować te specyfikacje oraz w jaki sposób prowadzi to do odkrywania kodu, który trzeba napisać. Pokażemy też, że te specyfikacje wykonywalne mogą być doskonałym narzędziem do tworzenia raportów i tworzenia dynamicznej dokumentacji.

2.4.1. Od przykładów do kryteriów akceptacji

Przykłady (takie jak podróże, które opisał Jerzy) mogą być wykorzystane jako podstawa kryteriów akceptacji. W skrócie kryteria akceptacji są efektem, który zadowoli zainteresowane strony (i przedstawiciele kontroli jakości) i pozwoli im stwierdzić, że aplikacja robi to, co powinna robić.

Rozmowy takie jak ta z Jerzym (w punkcie 2.3.3) to świetny sposób budowania zrozumienia przestrzeni problemu. Jeśli jednak użyjemy nieco bardziej uporządkowanego stylu, możemy uzyskać o wiele więcej. W BDD do wyrażania przykładów często stosuje się zapis w takiej oto formie⁴:

```
Zakładając <kontekst>
Gdy <coś się wydarzy>
Wtedy <oczekujemy jakiegoś efektu>
```

Ten format pozwala myśleć w kategoriach sposobu interakcji użytkowników z systemem oraz oczekiwanych wyników. Jak dowiemy się w następnym podrozdziale, format ten jest również łatwy do konwersji na postać automatycznych testów akceptacyjnych za pomocą takich narzędzi, jak Cucumber i JBehave. Ale ze względu na ewentualną późniejszą możliwość zautomatyzowania tych testów, ich format jest trochę mniej elastyczny. Jak się przekonamy, słowa Zakładając (ang. Given), Gdy (ang. When) i Wtedy (ang. Then) mają dla tych narzędzi szczególnie znaczenie, więc najlepiej traktować je jako specjalne słowa kluczowe.

Używając tej notacji, możemy wyrazić przytoczone wcześniej wymaganie w następujący sposób:

```
Zakładając pociągi linii Western odjeżdżają ze stacji Parramatta o 7:58, 8:02, 8:08, 8:11
Gdy chcę podróżować z Parramatta do Town Hall o 8:00
Wtedy powinienem uzyskać informację, że należy wsiąść w pociąg o 8:02
```

Jaki jest kontekst lub tło tego przykładu? ❶

Jaką akcję opisujemy?

Jaki jest oczekiwany wynik?

⁴ Składnia zaprezentowana w tym przykładzie często jest określana jako format Gherkin, ale nie jest to precyzyjne — Gherkin to składnia używana w aplikacji Cucumber i powiązanych z nią narzędziach, natomiast w tych przykładach wykorzystano JBehave. Dokładniej zagadnienie to zostanie opisane w rozdziale 5.

Gdy rozmawiałeś z Jerzym, powiedział Ci, że pociągi kursują na linii w dwóch kierunkach, zatem sekcja ❶ jest niekompletna — trzeba także podać stację początkową i kierunek. Drugi wariant jest taki, że kierunek można wywnioskować na podstawie stacji docelowej. Powyższy scenariusz można uściślić w następujący sposób:

Zakładając pociągi linii Western z Emu Plains odjeżdżają ze stacji Parramatta do Town Hall o 7:58, 8:00, 8:02, 8:11
 Gdy chcę podróżować z Parramatta do Town Hall o 8:00
 Wtedy powinienem uzyskać informację, że należy wsiąść w pociąg o 8:02

Zawiera zarówno nazwę linii, jak i kierunek

Jednak podczas omawiania tego przykładu zdajemy sobie sprawę, że mamy tylko dwie minuty na zakup biletu i przejście na odpowiedni peron. Naprawdę trzeba przekazać informację o kilku następujących pociągach:

Zakładając pociągi linii Western z Emu Plains odjeżdżają ze stacji Parramatta do Town Hall o 7:58, 8:00, 8:02, 8:11, 8:14, 8:21
 Gdy chcę podróżować z Parramatta do Town Hall o 8:00
 Wtedy powinienem uzyskać informację o pociągach o: 8:02, 8:11, 8:14

Teraz oczekujemy trzech wyników

Podano sześć godzin odjazdów pociągów, tak aby przykład stał się bardziej reprezentatywny

Zanim rozwiemy ten przykład lub przejdziemy do bardziej złożonych wymagań, spróbujemy pokazać, jak można przekształcić te kryteria akceptacji na wykonywalne specyfikacje za pomocą narzędzi JBehave, Maven i Git.

2.4.2. Konfigurowanie narzędzi Maven i Git

Istnieje wiele specjalistycznych narzędzi BDD, które można wykorzystać w celu automatyzacji kryteriów akceptacji. Do popularnych programów służących do tego celu należą narzędzia JBehave, Cucumber, SpecFlow i Behat. Chociaż nie jest to niezbędne, to za pomocą tych narzędzi łatwiej wyrazić zautomatyzowane testy w strukturalnej formie podobnej do notacji „Zakładając... Gdy... Wtedy...”, zaprezentowanej w poprzednim podrozdziale. Stosowanie tej notacji ułatwia właścicielom produktu i testerom zrozumienie i zidentyfikowanie zautomatyzowanych kryteriów akceptacji. To z kolei pomaga zwiększyć zaufanie do zautomatyzowanych testów i w ogóle do podejścia automatycznego testowania akceptacyjnego.

W dalszej części tej książki będę prezentował przykłady, posługując się kilkoma różnymi narzędziami BDD. W tym rozdziale będę używał przykładów napisanych z wykorzystaniem JBehave i języka Java⁵, a projekt zostanie zbudowany i uruchomiony za pomocą narzędzia Maven⁶. Raporty z testów będą generowane z wykorzystaniem Thucydides⁷ — biblioteki *open source*, która ułatwia organizowanie i tworzenie raportów z wyników testów BDD.

⁵ Jeśli czytelnik nie programuje w Javie, nie ma powodu do obaw. Przykłady kodu zostały napisane w taki sposób, aby były czytelne dla każdego, kto ma podstawową wiedzę na temat programowania. Narzędzia BDD dla środowisk .NET, Ruby i Python zostaną omówione w rozdziale 5. i dalszych.

⁶ Maven (<http://maven.apache.org/>) jest powszechnie używanym narzędziem do budowania aplikacji w Javie.

⁷ Więcej informacji na temat biblioteki Thucydides można znaleźć w witrynie internetowej tej biblioteki (<http://thucydides.info>).

Kod źródłowy dla tego rozdziału jest dostępny w repozytorium GitHub⁸ oraz w witrynie internetowej wydawnictwa Helion. Aby móc śledzić przykład, należy skonfigurować środowisko programistyczne z zainstalowanym następującym oprogramowaniem:

- Java JDK (przykładowy kod tworzono w środowisku Java 1.7.0, ale powinien bezproblemowo działać w środowisku JDK 1.6.0).
- Maven 3.0.x.
- Git.

Serwis GitHub umożliwia dostęp do repozytorium na różne sposoby. Jeśli zainstalowaliśmy aplikację Git i mamy skonfigurowane konto w witrynie GitHub z dostępem SSH⁹, możemy utworzyć klon repozytorium z przykładowym kodem w następujący sposób:

```
$ git clone git@github.com:bdd-in-action/chapter-2.git
```

Jeśli nie ustawiliśmy i nie skonfigurowaliśmy kluczy SSH dla serwisu GitHub, możemy również użyć następującego polecenia (Git poprosi o podanie nazwy użytkownika i hasła):

```
$ git clone https://github.com/bdd-in-action/chapter-2.git10
```

Po sklonowaniu projektu warto uruchomić polecenie `mvn verify` w katalogu projektu, aby aplikacja Maven mogła pobrać zależności potrzebne do jej działania oraz do uruchomienia projektu. Operację tę trzeba przeprowadzić tylko raz, ale może ona trochę potrwać. Uruchom następujące polecenia:

```
$ cd chapter-2
$ cd train-timetables
$ mvn verify
```

Jeśli chcesz kodować każdy krok samodzielnie, przejdź do katalogu `train-timetables` i przełącz się na gałąź `start`:

```
$ git checkout start
```

Gdy to zrobisz, uzyskasz prosty szkielet projektu z prawidłowo skonfigurowanym skryp-tem kompilacji programu Maven (plik `pom.xml`) oraz strukturę katalogów, z której możesz skorzystać. Jeśli w dowolnym momencie chcesz przyjrzeć się przykładowemu rozwiązaniu, uruchom następujące polecenie:

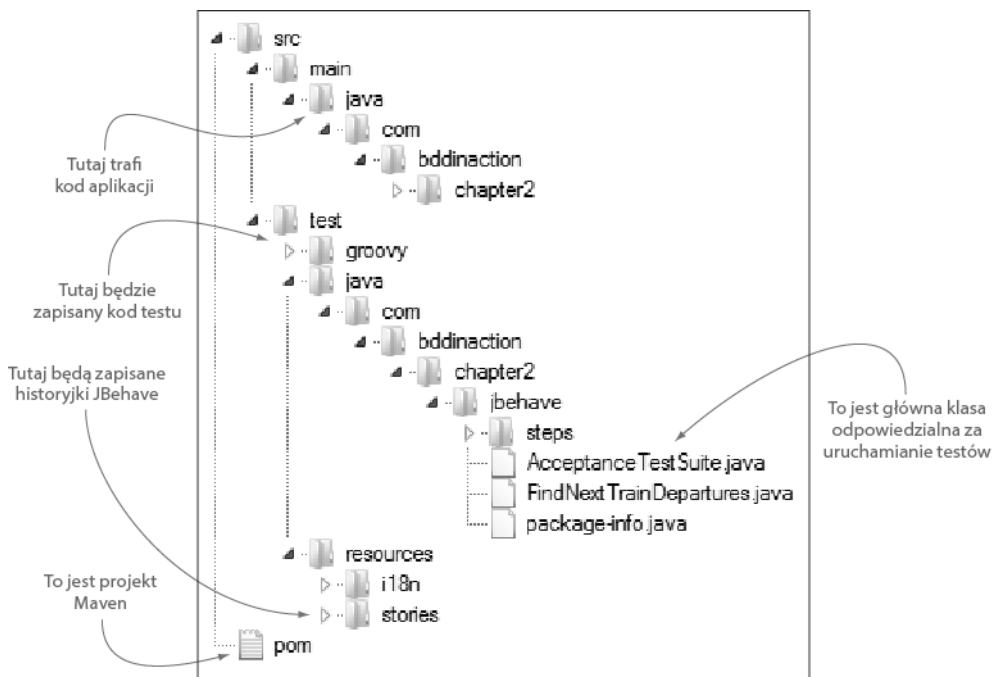
```
$ git checkout master
```

⁸ Kod źródłowy przykładów z tego rozdziału można pobrać pod adresem <https://github.com/bdd-in-action/chapter-2>.

⁹ Dobry przewodnik dotyczący instalacji narzędzia Git dla różnych systemów operacyjnych można znaleźć w pomocy serwisu GitHub (<https://help.github.com/articles/set-up-git>).

¹⁰ Podana ścieżka dostępu do repozytorium GitHub oraz wszystkie komendy Git dotyczą przykładowych kodów z oryginalnej wersji książki. Polską wersję należy pobrać z serwisu FTP wydawnictwa Helion pod adresem <ftp://ftp.helion.pl/przyklady/bdddzi.zip>. Dalszy opis przykładów dotyczy polskiej wersji językowej — *przyj. tłum.*

Początkową strukturę projektu pokazano na rysunku 2.2. Jest ona zgodna ze standardowymi konwencjami programu Maven. Kod aplikacji będzie zapisany w katalogu `src/main/java`, natomiast kod testu w katalogu `src/test/java`. Historyjki JBehave trafią do katalogu `src/test/resources/stories`. `AcceptanceTestSuite` jest prostą klasą uruchamiania testów bazującą na frameworku JUnit, która uruchomi historyjki JBehave wewnątrz katalogu `src/test/resources` i w katalogach podrzędnych.



Rysunek 2.2. Początkowa struktura projektu

2.4.3. Specyfikacje wykonywalne — automatyzacja kryteriów akceptacji

Wyrażanie wymagań w formie strukturalnych przykładów daje wiele korzyści. Przykłady stanowią doskonały punkt wyjścia do rozmów o potrzebach biznesowych i oczekiwaniach. W porównaniu z bardziej abstrakcyjnymi specyfikacjami pozwalają lepiej wyeliminować nieporozumienia i nieprawidłowe założenia. Inną ważną korzyścią ze stosowania tej metody jest to, że pozwala ona na łatwiejsze zautomatyzowanie wymagań w formie testów akceptacyjnych.

Po skonfigurowaniu środowiska programistycznego nadszedł czas, aby zautomatyzować przykład, który omówiliśmy w poprzednich podrozdziałach. JBehave, podobnie jak wiele narzędzi BDD, wykorzystuje specjalny język do reprezentowania specyfikacji wykonywalnych w strukturalnym, ale nadal bardzo czytelnym formacie. W JBehave scenariusz można wyrazić w sposób zaprezentowany na listingu 2.1.

Listing 2.1. Kryteria akceptacji wyrażone w JBehave

Dowiedz się, o której godzinie odjeżdżają następne pociągi do stacji docelowej

Narracja:

W celu bardziej efektywnego planowania podróży

Jako podróżny

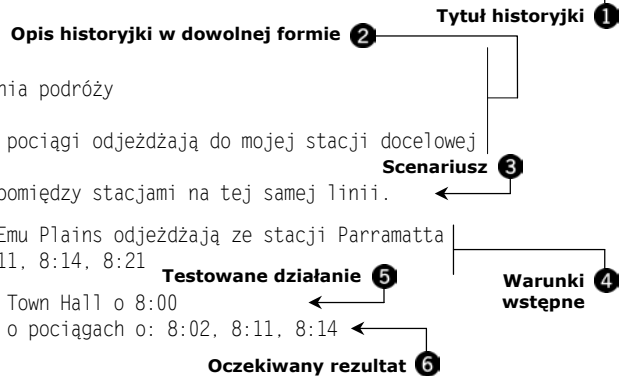
Chcę się dowiedzieć, jakie następne pociągi odjeżdżają do mojej stacji docelowej

Scenariusz: Znajdź optymalną trasę pomiędzy stacjami na tej samej linii.

Zakładając pociągi linii Western z Emu Plains odjeżdżają ze stacji Parramatta do Town Hall o 7:58, 8:00, 8:02, 8:11, 8:14, 8:21

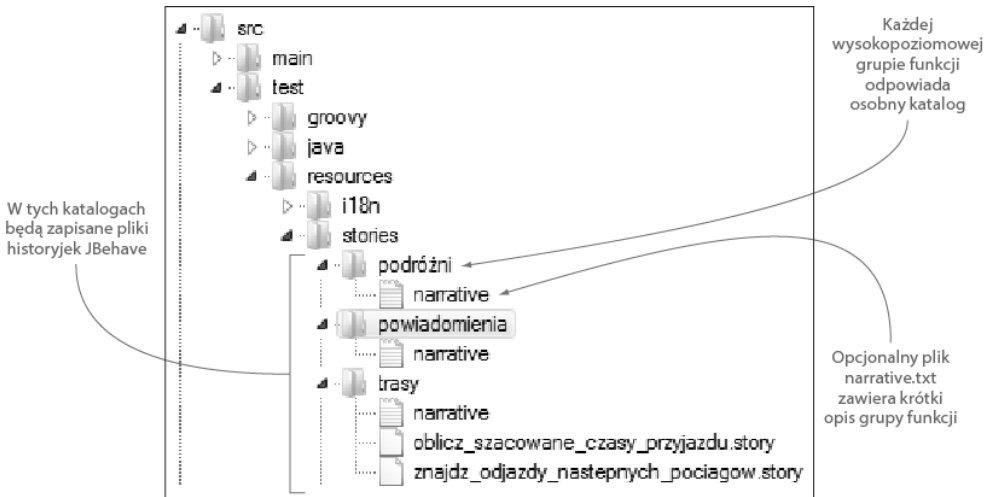
Gdy chcę podróżować z Parramatta do Town Hall o 8:00

Wtedy powinienem uzyskać informację o pociągach o: 8:02, 8:11, 8:14



Powyższy opis zawiera niewiele więcej w porównaniu ze strukturalną wersją przykładu, który omawialiśmy wcześniej. Zaczynamy od opisu historyjki **1** i **2**. Słowo kluczowe Scenariusz (ang. Scenario) **3** oznacza początek każdego nowego scenariusza. Słowa kluczowe Zakładając (ang. Given) **4**, Gdy (ang. When) **5** i Wtedy (ang. Then) **6** wprowadzają poszczególne części każdego scenariusza.

W JBehave scenariusze są pogrupowane według historyjek i zapisane w pliku z rozszerzeniem `.story`¹¹. Jak można zobaczyć na rysunku 2.3, plik zawierający definicję tej historyjki nosi nazwę `znajdz_odjazdy_nastepnych_pociagow.story`.



Rysunek 2.3. Pliki historyjek JBehave są zorganizowane w postaci katalogów

¹¹ Pod tym względem JBehave nieco różni się od systemu Cucumber i narzędzi bazujących na systemie Cucumber, które wykorzystują pliki `feature`. Do tej różnicy oraz tego, co z niej wynika, powrócimy w rozdziale 5.

Wszystkie pliki *.story* można umieścić bezpośrednio w katalogu *stories*. To jednak staje się niewygodne w przypadku, gdy mamy dużą liczbę plików historyjek. Zamiast tego najlepiej pogrupować historyjki w wysokopoziomowe grupy funkcjonalne. Na przykład w miarę rozwoju tego projektu może powstać następująca struktura katalogów:

- *trasy* (obliczenia tras i informacje o rozkładzie jazdy);
- *podróżni* (indywidualne dane dotyczące podróży dla pasażerów);
- *powiadomienia* (powiadomienia o opóźnieniach dla pasażerów).

W celu generowania dokumentacji w każdym z tych katalogów można również umieścić plik tekstowy o nazwie *narrative.txt*¹². Zawiera on nazwę grupy funkcji oraz krótki opis tego, co ona obejmuje. Na przykład plik *narrative.txt* dla katalogu *trasy* może wyglądać tak, jak na poniższym listingu.

Listing 2.2. Plik *narrative.txt* opisuje wysokopoziomową funkcjonalność

```
Trasy i rozkłady jazdy
Informacje w czasie rzeczywistym o rozkładach jazdy i trasach
                                ← Krótki tytuł
                                ←
                                Krótki, dowolny opis tej wysokopoziomowej funkcjonalności
```

Mamy teraz specyfikację wykonywalną. Choć nie ma jeszcze kodu obsługującego ten scenariusz, narzędzie JBehave nadal może ją uruchomić. Aby to sprawdzić, wystarczy przejść do katalogu *train-timetables* i uruchomić poniższe polecenie:

```
$ mvn verify
```

Spowoduje to wygenerowanie zbioru raportów w katalogu *target/site/thucydides*¹³. Jeśli otworzysz plik *index.html* w tym katalogu i klikniesz jedyny test w tabeli *Test* u dołu ekranu, powinieneś zobaczyć ekran podobny do pokazanego na rysunku 2.4.

W tym momencie stworzony scenariusz przestał być prostym dokumentem tekstowym. Teraz to jest *specyfikacja wykonywalna*. Może być uruchomiona w ramach procesu automatycznej kompilacji w celu stwierdzenia, czy określona funkcja została zaimplementowana, czy nie. Gdy takie testy są wykonywane po raz pierwszy, są oznaczane flagą *PENDING*, co w kategoriach BDD oznacza, że test został zautomatyzowany, ale kod, który implementuje testowaną funkcję, nie został jeszcze napisany. Jeśli funkcje zostaną zaimplementowane, a ich testy akceptacyjne zakończą się sukcesem, są oznaczone słowem *PASSED*, co oznacza, że zakończyliśmy pracę w tym obszarze.

Dynamiczna dokumentacja jest jednak czymś więcej niż tylko raportem z testów. Powinna ona również informować o stanie wszystkich wyspecyfikowanych wymagań, nawet tych, dla których jeszcze nie napisano żadnych testów. Daje to znacznie pełniejszy obraz projektu i produktu. Przykład raportu o tym poziomie szczegółowości możemy

¹²Jeśli plik *narrative.txt* istnieje, jest również wykorzystywany przez program Thucydides w celu generowania dynamicznej dokumentacji, którą zaprezentujemy w dalszej części tej książki.

¹³Maven najpierw ściąga biblioteki, których potrzebuje — ta operacja może zająć trochę czasu, ale jest wykonywana tylko raz.



Rysunek 2.4. Historyjka JBehave wewnątrz raportów z testów akceptacyjnych

zobaczyć, klikając zakładkę *Requirements* w raporcie, który przed chwilą wygenerowaliśmy (patrz rysunek 2.5). Na rysunku powinna się wyświetlić lista wszystkich wysokopoziomowych wymagań wraz z informacją o tym, ile prac zrealizowano dla każdego z nich (na tym etapie będzie to całkowity brak prac).

Zanim omówimy, w jaki sposób można zautomatyzować podobne scenariusze w Javie, przyjrzyjmy się innemu wariantowi. Bardzo ważną funkcją tworzonej aplikacji jest zdolność informowania podróżnych o godzinie, o której dotrą do stacji docelowej, jeśli wyjadą ze stacji początkowej w określonym czasie. Jerzy podał kilka przykładów, które można wykorzystać w celu stworzenia scenariusza opisującego to wymaganie. Zaprezentowano je na listingu 2.3.

Listing 2.3. Szacowanie godziny przyjazdu

Informacja dla podróżnych o czasie przybycia do stacji docelowej ← **Krótki tytuł**

Narracja:

W celu bardziej efektywnego planowania podróży

Jako podróżny

Chcę wiedzieć, o której godzinie dotrę do stacji docelowej

Tytuł scenariusza

← **Tutaj zaczynają się szczegóły scenariusza**

Scenariusz: Szacowanie czasu przyjazdu

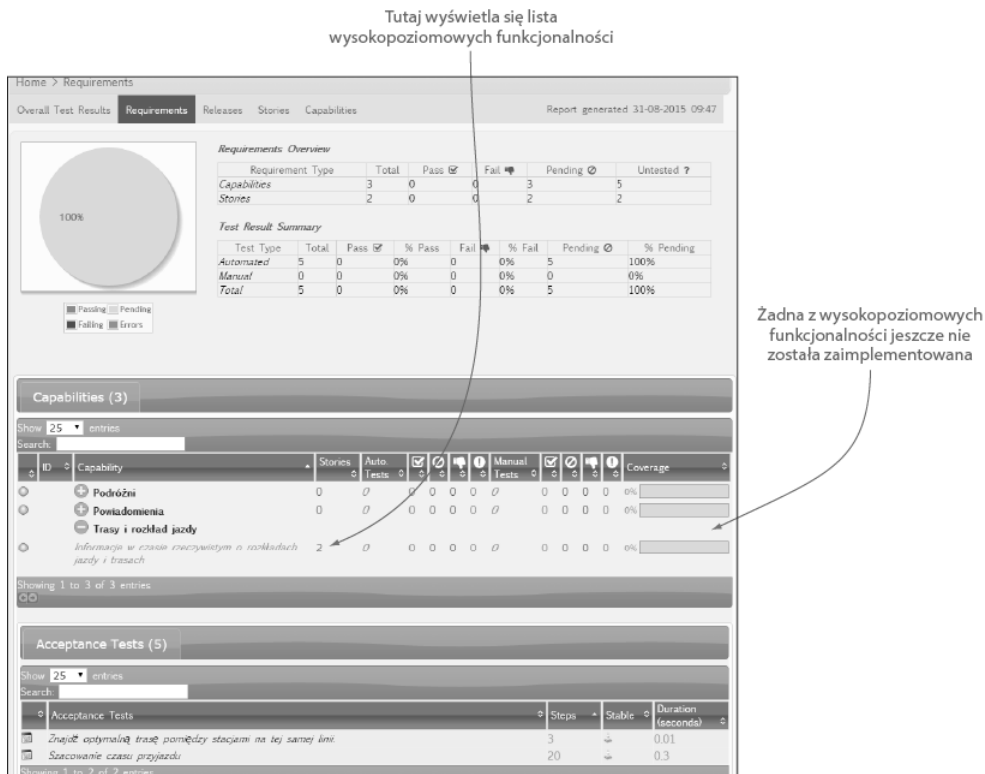
Zakładając chcę się dostać z <stacja-początkowa> do <stacja-docelowa>

I następny pociąg odjeżdża o <czas-wyjazdu> na linii <linia>

Gdy zapytam o godzinę przyjazdu

Wtedy powinienem uzyskać następujący szacowany czas przyjazdu: <czas-pryjazdu>

W scenariuszu wykorzystano dane z poniższej tabeli



Rysunek 2.5. Dynamiczna dokumentacja powinna również poinformować o tym, jakie wymagania zostały wyspecyfikowane, nawet wtedy, gdy jeszcze nie istnieją dla nich żadne testy

Przykłady:

Nagłówki w tabeli identyfikują wartości w testowych danych ①

stacja-początkowa	stacja-docelowa	czas-wyjazdu	linia	czas-przyjazdu
Parramatta	Town Hall	8:02	Western	8:29
Epping	Central	8:03	Northern	8:48
Epping	Central	8:07	Newcastle	8:37
Epping	Central	8:13	Epping	8:51

② **Każdy wiersz reprezentuje odrębny zestaw danych testowych**

Tutaj główną nowością jest wykorzystanie tekstowej tabeli do zaprezentowania danych testowych. Nagłówki tabeli ① identyfikują wartości testowych danych. Każdy wiersz w tabeli ② reprezentuje odrębny zestaw danych testowych do wykorzystania w tym scenariuszu.

Tę historyjkę można umieścić w pliku *oblicz_szacowane_czasz_przyjazdu.story* obok pliku z poprzednią historyjką. Spowoduje to dodanie historyjki do zbioru historyjek przetwarzanych przez JBehave. W ten sposób po uruchomieniu testów stanie się ona częścią dynamicznej dokumentacji (patrz rysunek 2.6).

Język używany w obu tych scenariuszach jest bardzo zbliżony do języka używanego przez użytkownika. Ponieważ te scenariusze pojawiają się w raportach z testów, to dzięki temu, że do ich stworzenia użyto znanego języka, będą one łatwiejsze do analizy przez testerów, użytkowników końcowych oraz innych użytkowników, którzy nie są deweloperami.

W tym scenariuszu użyto zmiennych z przykładowej tabeli

Szacowanie czasu przyjazdu 0,19s

Story: Oblicz Szacowane Czasy Przyjazdu

Narracja:
 W celu bardziej efektywnego planowania podróży
 Jako podróżny
 Chcę wiedzieć, o której godzinie dotrę do stacji docelowej
 Scenariusz: Szacowanie czasu przyjazdu
 Zakładając chcę się dostać z do
 I następny pociąg odjeżdża o na linii
 Gdy zapytam o godzinę przyjazdu
 Wtedy powinienem uzyskać następujący szacowany czas przyjazdu.
 Przykłady:
 stacja-początkowa | stacja-docelowa | czas-wyjazdu | linia | czas-przyjazdu
 Parramatta | Town Hall | 8.02 | Western | 8.29
 Epping | Central | 8.03 | Northern | 8.48
 Epping | Central | 8.07 | Newcastle | 8.37
 Epping | Central | 8.13 | Epping | 8.51

Trasy (capability) Oblicz szacowane czasy przyjazdu (story)

Scenario:

Zakładając, chcę się dostać z <stacja-początkowa> do <stacja-docelowa>
 I następny pociąg odjeżdża o <czas-wyjazdu> na linii <linia>
 Gdy zapytam o godzinę przyjazdu
 Wtedy powinienem uzyskać następujący szacowany czas przyjazdu:<czas-przyjazdu>

Examples:

Show 25 entries

Search:

Stacja-Początkowa	Stacja-Docelowa	Czas-Wyjazdu	Linia	Czas-Przyjazdu
Epping	Central	8.03	Northern	8.48
Epping	Central	8.07	Newcastle	8.37
Epping	Central	8.13	Epping	8.51
Parramatta	Town Hall	8.02	Western	8.29

Showing 1 to 4 of 4 entries

Steps	Outcome	Duration
[1] [stacja-początkowa=Parramatta, stacja-docelowa=Town Hall, czas-wyjazdu=8.02, linia=Western, czas-przyjazdu=8.29]	PENDING	0,04s
[2] [stacja-początkowa=Epping, stacja-docelowa=Central, czas-wyjazdu=8.03, linia=Northern, czas-przyjazdu=8.48]	PENDING	0,03s
[3] [stacja-początkowa=Epping, stacja-docelowa=Central, czas-wyjazdu=8.07, linia=Newcastle, czas-przyjazdu=8.37]	PENDING	0,02s
[4] [stacja-początkowa=Epping, stacja-docelowa=Central, czas-wyjazdu=8.13, linia=Epping, czas-przyjazdu=8.51]	PENDING	0,02s

Każdy wiersz tej tabeli jest oddzielnym testem

Wyniki testów dla każdego wiersza

Rysunek 2.6. Dynamiczna dokumentacja dla scenariusza zdefiniowanego za pomocą tabeli

2.4.4. Automatyczne testy – implementacja kryteriów akceptacji

Teraz, gdy już zdefiniowaliśmy i zautomatyzowaliśmy niektóre kryteria akceptacji, możemy przystąpić do prawdziwej pracy. Oczywiście, logika potrzebna do sprawdzenia kryteriów akceptacji nie napisze się sama: trzeba dodać trochę kodu, aby testy rzeczywiście wykonały swoją pracę.

Zacznijmy od pierwszego scenariusza z listingu 2.1:

Scenariusz: Znajdź optymalną trasę pomiędzy stacjami na tej samej linii.

Zakładając pociągi linii Western z Emu Plains odjeżdżają ze stacji Parramatta do Town Hall

↳ o 7:58, 8:00, 8:02, 8:11, 8:14, 8:21

Gdy chcę podróżować z Parramatta do Town Hall o 8:00

Wtedy powinienem uzyskać informację o pociągach o: 8:02, 8:11, 8:14

Kontynuując przykład z wykorzystaniem JBehave i Javy, możemy zaimplementować pusty automatyczny test dla tego scenariusza w klasie o nazwie `OptimalItinerarySteps`.

↳ java, tak jak pokazano na listingu 2.4.

Listing 2.4. Brakująca implementacja scenariusza JBehave

```
package com.bddinaction.chapter2.jbehave.steps;
```

```
import org.jbehave.core.annotations.Given;
import org.jbehave.core.annotations.Pending;
import org.jbehave.core.annotations.Then;
import org.jbehave.core.annotations.When;
import java.util.List;
import org.joda.time.LocalDateTime;
```

```
public class OptimalItinerarySteps {
    @Given("pociągi linii $line z $lineStart odjeżdżają ze stacji $departure do $destination o $departureTimes")
    @Pending
    public void givenArrivingTrains(String line,
                                    String lineStart,
                                    String departure,
                                    String destination,
                                    List<LocalTime> departureTimes) {

    }

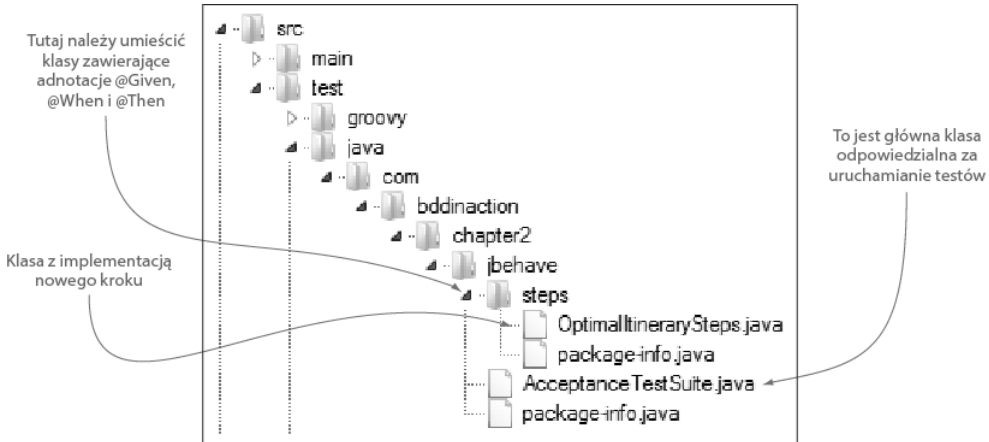
    @When("chcę podróżować z $departure do $destination o $time")
    @Pending
    public void whenIWantToTravel(String departure,
                                    String destination,
                                    LocalDateTime time) {

    }

    @Then("powinienem uzyskać informację o pociągach o: $viableTrainTimes")
    @Pending
    public void shouldFindTheseTrains(List<LocalTime> viableTrainTimes) {

    }
}
```

Klasę tę należy umieścić w pakiecie `steps` wewnątrz pakietu `jbehave` w katalogu `src/test/java` (patrz rysunek 2.7). Testy JBehave mogą być zaimplementowane w Javie albo w innych językach platformy JVM, takich jak Groovy lub Scala. Podczas uruchamiania scenariusza JBehave wykorzysta teksty z adnotacji `@Given` 1, `@When` 3 i `@Then` 4, aby określić metodę, która będzie wywołana w każdym kroku. Jak widać na listingu, można również przekazać elementy z tekstu scenariusza do metod testowych w postaci parametrów.



Rysunek 2.7. Struktura pakietu zawierająca nową klasę z implementacją kroku

Jeśli pozostawimy implementację testu w takiej postaci, jaką pokazano na listingu, to dzięki adnotacji `@Pending` uzyskamy pewność, że ten test wygeneruje dokładnie takie same wyniki, jak pokazano na rysunku 2.4. Ale już wkrótce dodamy implementację każdej metody, tak by przekształcić ten kod w działający w pełni test.

Te kroki spełniają rolę punktu wyjścia dla kodu produkcyjnego: mówią, co będzie trzeba zbudować, aby dostarczyć opisywaną funkcję. Zespoły praktykujące BDD zwykle zaczynają od określenia wyniku, którego potrzebują, i działają wstecz. Spróbujmy więc zobaczyć, jak ten sposób działania będzie wyglądał w tym przykładzie.

Zacniemy od kroku `@Then`, który wyraża oczekiwany wynik. W istocie chcemy, aby usługa zwracała listę proponowanych godzin odjazdów pociągów, które pasują do oczekiwanego pory. Oto jeden ze sposobów wyrażenia tego warunku:

```
@Then("powiniennem uzyskać informację o pociągach o: $expectedTrainTimes")
public void shouldBeInformedAbout(List<LocalTime> expectedTrainTimes) {
    assertThat(proposedTrainTimes).isEqualTo(expectedTrainTimes);
}
```

JBehave dopasuje pierwszą linijkę scenariusza, wyodrębni z niej listę oczekiwanych godzin (oznaczonych w adnotacji `@Then` zmienną `$expectedTrainTimes`) i przekaże je jako parametr do metody. Zwróćmy uwagę, że w tym przykładzie usunęliśmy również adnotację `@Pending`. Dzięki temu JBehave będzie wiedział, że powinien uruchomić ten krok. Ale ten test jeszcze się nie skompiluje — nadal trzeba zdecydować, skąd będzie odjeżdżał proponowany pociąg.

BDD pomaga odkryć techniczny projekt potrzebny w celu dostarczenia cech funkcjonalnych spełniających cele biznesowe. Załóżmy, że zdecydowaliśmy się zaimplementować tę aplikację za pomocą kilku różnych usług sieciowych. Na przykład możemy potrzebować usługi, która dostarcza informacji na temat godzin odjazdów do witryny WWW lub aplikacji mobilnej. Ta usługa jeszcze nie istnieje, więc trzeba odkryć, co powinna robić. Jednym z najbardziej skutecznych sposobów, aby to zrobić, jest po prostu napisanie kodu, który chcielibyśmy mieć. To pozwala na projektowanie czystego, czytelnego i samodokumentującego się interfejsu API.

Na przykład, aby znaleźć czasy odjazdu następnych pociągów z danej stacji, można wykorzystać jedną, bardzo prostą implementację:

```
proposedTrainTimes = itineraryService.findNextDepartures(departure,
                                                         destination,
                                                         startTime);
```

Jeśli ten kod zintegrujemy z krokiem @When, otrzymamy coś takiego:

```
List<LocalTime> proposedTrainTimes;
```

```
@When("chcę podróżować z $departure do $destination o $startTime")
public void whenIWantToTravel(String departure,
                              String destination,
                              LocalTime startTime) {
    ItineraryService itineraryService = new ItineraryService();
    proposedTrainTimes = itineraryService.findNextDepartures(departure,
                                                             destination,
                                                             startTime);
}
```

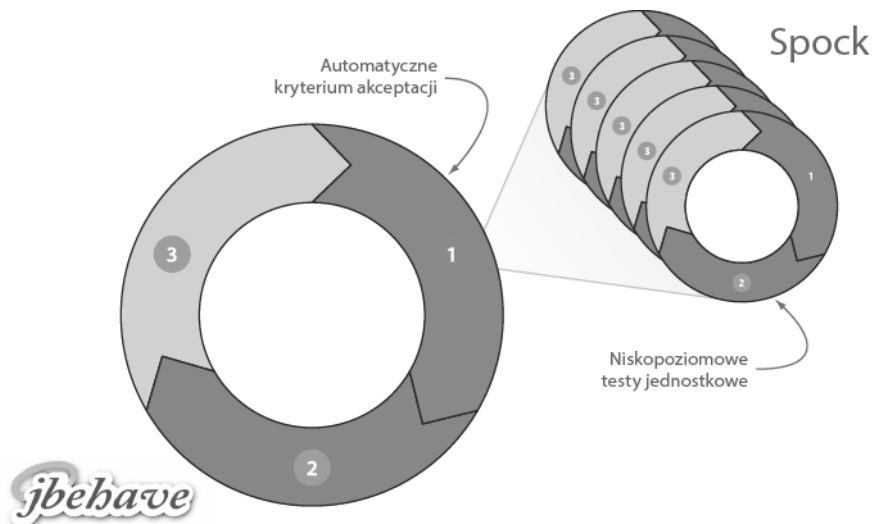
Aby to kryterium akceptacji mogło przejść, należy zaimplementować metodę `findNextDepartures()`. Jednak zanim to zrobimy, musimy przełączyć się z testowania akceptacyjnego na testy jednostkowe. Jak można zauważyć, testy akceptacyjne służą do zademonstrowania wysokopoziomowych zachowań aplikacji, natomiast testy jednostkowe są używane do budowania komponentów, które będą używane do zaimplementowania tego zachowania. Testy akceptacyjne zazwyczaj używają pełnego lub prawie pełnego stosu aplikacji, natomiast testy jednostkowe koncentrują się na poszczególnych komponentach w izolacji. Testy jednostkowe ułatwiają skoncentrowanie się na prawidłowym zaimplementowaniu konkretnej klasy oraz określeniu innych usług lub komponentów, których ona potrzebuje. Testy jednostkowe ułatwiają również wykrywanie i izolowanie błędów lub regresji. W celu spełnienia kryterium akceptacji zwykle pisze się wiele krótkich testów jednostkowych (patrz rysunek 2.8).

W celu napisania tych testów jednostkowych można by użyć konwencjonalnego frameworka, na przykład JUnit. Ponieważ jednak ta książka jest o BDD, skorzystamy z bardziej specyficznego dla BDD narzędzia testów jednostkowych o nazwie Spock. Spock jest prostą w użyciu i ekspresywną biblioteką testowania w stylu BDD dla aplikacji w językach Java i Groovy. Z tego narzędzia będziemy intensywnie korzystać w dalszej części tej książki.

Groovy jest językiem dynamicznym zbudowanym na bazie Javy, który zawiera wiele własności z takich języków, jak Ruby i Python. To bardzo ekspresywny język, który świetnie nadaje się do sprawdzania wyników — pozwala to robić w bardzo czytelny i zwięzły sposób.

Zacniemy od testowania kryterium akceptacji opisanego w scenariuszu JBehave na listingu 2.1, ale na poziomie testu jednostkowego. W Spock stworzymy klasę Groovy o nazwie `WhenCalculatingArrivalTimes.groovy` i umieścimy ją w odpowiednim pakiecie w katalogu `src/test/groovy`¹⁴. Klasa ta może mieć następującą postać:

¹⁴W przykładowym rozwiązaniu klasę tę umieściliśmy w pakiecie `com.bddinaction.chapter2.services`.



Rysunek 2.8. Aby spełnić kryterium akceptacji, zazwyczaj trzeba napisać wiele niskopoziomowych testów jednostkowych w stylu TDD. W przykładzie zaprezentowanym w tym rozdziale wykorzystano framework JBehave do implementacji kryterium akceptacji oraz framework Spock dla testów jednostkowych

Pisanie testów jednostkowych z wykorzystaniem Spock

Za pomocą narzędzia Spock testy jednostkowe pisze się w formie „specyfikacji”, przy użyciu bardzo czytelnej struktury „Given... When... Then”, podobnej do tej, która jest stosowana w scenariuszach JBehave. Na przykład, gdybyśmy chcieli za pomocą frameworka Spock przetestować klasę Calculator, moglibyśmy użyć kodu w następującej postaci:

```
def "powinna wyliczać sumę dwóch liczb"() {
    given: "dwie liczby"
        int a = 1
        int b = 2
    when: "dodajemy liczby do siebie"
        def calculator = new Calculator()
        int sum = calculator.add(a,b)
    then: "wynik powinien być sumą dwóch liczb"
        sum == 3
}
```

Definiowanie specyfikacji
 Klauzula given
 Klauzula when
 Klauzula then
 Odpowiednik instrukcji assert sum == 3

Wielu programistów Javy pisze testy jednostkowe za pomocą frameworka Spock, ponieważ pozwala on pisać zwięzłe i opisowe testy za pomocą kodu mniej szablonowego od tego, który byłby potrzebny w przypadku zastosowania języka Java.

```
package com.bddinaction.chapter2.services
```

```
import org.joda.time.LocalDateTime
import spock.lang.Specification
```

```
class WhenCalculatingArrivalTimes extends Specification {
    ItineraryService itineraryService;
    def "powinna obliczyć prawidłową godzinę przyjazdu"() {
```

Testy Spock
 muszą
 rozszerzać
 klasę
 Specification

Sprawdzone
 wymaganie

```

given:
  itineraryService = new ItineraryService():
when:
  def proposedTrainTimes =
    itineraryService.findNextDepartures("Parramatta",
                                        "Town Hall",
                                        at('8:00'));
then:
  proposedTrainTimes == [at('8:02'), at('8:11'), at('8:14' )]
}
def at(String time) {
  def hour = Integer.valueOf(time.split(':')[0])
  def minute = Integer.valueOf(time.split(':')[1])
  new LocalTime(hour.toInteger(), minute.toInteger())
}
}

```

1 „Zakładając mam usługę informacji o trasach”
2 „Gdy szukam następných odjazdów”
3 To powinienem uzyskać propozycję następujących godzin
Funkcja narzędziowa ułatwiająca zainicjowanie wartości godzin

Ogólnie rzecz biorąc, ta specyfikacja realizuje to samo, co test JBehave. Tworzy nową usługę tras **1**, szuka następných odjazdów na trasie Parramatta i Town Hall, począwszy od 8:00 **2**, a następnie sprawdza, czy odpowiada to oczekiwanym godzinom **3**. Kiedy test przejdzie, możemy mieć pewność, że klasa `ItineraryService` realizuje oczekiwane operacje.

Oczywiście, jeśli uruchomimy tę specyfikację, to zakończy się ona niepowodzeniem, ponieważ na razie nie napisaliśmy jeszcze żadnego kodu źródłowego aplikacji. Teraz nadszedł czas na napisanie tego kodu. Zobaczmy, jak mogłaby wyglądać metoda `find`

```

↪NextDepartures():
public List<LocalTime> findNextDepartures(String departure,
                                         String destination,
                                         LocalTime startTime) {
  List<Line> lines = timetableService.findLinesThrough(departure,
                                                       destination);
  List<LocalTime> allArrivalTimes = getArrivalTimesOnLines(lines,
                                                          departure);
  List<LocalTime> candidateArrivalTimes
    = findArrivalTimesAfter(startTime, allArrivalTimes);
  return atMost(3, candidateArrivalTimes);
}
private List<LocalTime> atMost(int max, List<LocalTime> times) {
  return Lists.partition(times, max).get(0);
}

```

1 Jakie linie kursują pomiędzy tymi dwoma stacjami?
2 O jakich godzinach pociągi na tych liniach wyjeżdżają ze stacji początkowej?
3 Jakie pociągi przyjeżdżają tu po podanej godzinie startu?
4 Interesuje mnie informacja tylko o pierwszych trzech z nich
Metoda narzędziowa pozwalająca zwrócić do trzech godzin przyjazdów.

Zadaniem usługi tras jest obliczenie informacji na temat odjazdów i szczegółów podróży na podstawie aktualnych rozkładów jazdy. Do tego celu potrzebne są dane dotyczące rozkładu jazdy, ale jeszcze nie napisaliśmy żadnego kodu, który mógłby obliczać te informacje: rozkłady jazdy są skomplikowane i stanowią odrębną dziedzinę problemu. Zgodnie z dobrymi praktykami projektowymi powinniśmy użyć odrębnej usługi, która dostarczy danych dotyczących rozkładu jazdy do usługi tras. W kodzie zamieszczonym powyżej zadeklarowano obiekt `timetableService` **1**, który jest odpowiedzialny za tę operację.


Pierwszą rzeczą, jaką ten kod robi, jest znalezienie linii kolejowej, która pozwoli pasażerowi na podróż ze stacji początkowej do stacji docelowej. Jest to domena usługi rozkładu jazdy, zatem prosimy ją o dostarczenie wykazu linii od jednej stacji do drugiej **1**. W tym kontekście *linia* reprezentuje ścieżkę, którą poruszają się pociągi z jednej stacji do drugiej, w określonym kierunku. Pamiętajmy, że usługa rozkładu jazdy nie ma jeszcze żadnych metod, zatem właśnie odkryliśmy coś, co powinna robić usługa rozkładu jazdy.

Po uzyskaniu listy linii możemy znaleźć dla każdej z nich godziny przyjazdu pociągów na stację początkową. Istnieje kilka sposobów, aby to zrobić. Ważne jest jednak to, że zadanie to oddelegujemy do innej metody **2** i zajmiemy się główną logiką biznesową.

Ostatnią rzeczą, jaką ta metoda musi robić, jest znalezienie godzin przyjazdów po żądanej godzinie początkowej **3** i zwrócenie kolejnych trzech godzin odjazdu **4**.

Kompletny kod źródłowy tego rozwiązania można znaleźć w serwisie FTP wydawnictwa Helion¹⁵; skoncentrujemy się jednak na metodzie `getArrivalTimesOnLines()` zamieszczonej poniżej:

```
private List<LocalTime> getArrivalTimesOnLines(List<Line> lines,
                                               String station) {
    TreeSet<LocalTime> allArrivalTimes = Sets.newTreeSet();
    for(Line line : lines) {
        allArrivalTimes.addAll(
            timetableService.findArrivalTimes(line, station));
    }
    return new ArrayList<LocalTime>(allArrivalTimes);
}
```



Ta metoda jest interesująca, ponieważ wskazuje na inną operację, którą powinna realizować usługa rozkładu jazdy. Logika nie jest skomplikowana — stworzenie prostej kolekcji godzin przyjazdu dla linii przechodzących przez daną stację **1**. Linia jest prostym obiektem dziedzicznym z nazwą, stacją wyjazdu i listą stacji należących do tej linii¹⁶.

Aby jednak metoda `getArrivalTimesOnLines()` zadziałała, musimy znać planowane godziny przyjazdu pociągów z danej linii na stację wyjazdu. Informację tę powinniśmy uzyskać z usługi rozkładu jazdy.

Oznacza to, że potrzebujemy obiektu `TimetableService`, który realizuje następujące działania:

- Wyszukuje linie przebiegające przez dowolne dwie stacje.
- Znajduje godziny przyjazdu na określonej stacji pociągów ze wskazanej linii.

Możemy sformalizować to, czego potrzebujemy, w kroku `Given` specyfikacji Spock, tworząc usługę, która „udaje” usługę rozkładu jazdy — zachowującą się dokładnie tak, jak tego chcemy. Musimy tylko zapewnić, aby usługa tras prawidłowo przetwarzała dane dostarczone z usługi rozkładu jazdy. Można to zrobić za pomocą poniższego kodu:

¹⁵ Kod źródłowy przykładów z tego rozdziału można pobrać pod adresem <ftp://ftp.helion.pl/przyklady/bddzi.zip>.

¹⁶ Dla uproszczenia w przykładowym kodzie zamieszczono pełną implementację klasy `Line`.


```

TimetableService timetableService = Mock(TimetableService)
def "powinna obliczyć prawidłową godzinę przyjazdu"() {
    given:
        def westernLine =
            Line.named("Western").departingFrom("Emu Plains")
            timetableService.findLinesThrough("Parramatta",
                "Town Hall") >> [westernLine]
            timetableService.findArrivalTimes(westernLine, "Parramatta") >>
                [at(7,58), at(8,00), at(8,02), at(8,11), at(8,14), at(8,21)]

        when:
            def proposedTrainTimes =
                itineraryService.findNextDepartures("Parramatta",
                    "Town Hall",
                    at(8,00));

        then:
            proposedTrainTimes == [at(8,02), at(8,11), at(8,14)]
}

```

Utworzenie makiety usługi rozkładu jazdy. |

Gdy zapytamy, które linie przechodzą przez stację Parramatta i Town Hall, makieta usługi rozkładu jazdy zwraca tę linię kolejową. ❶

Definiujemy linię kolejową

Makieta usługi rozkładu jazdy zwraca również prawidłowe godziny przyjazdu na stację Parramatta. ❷

Ten kod konfiguruje makietę (ang. *mock*) usługi rozkładu jazdy w celu zamodelowania działań, jakie powinna realizować prawdziwa usługa rozkładu jazdy. Wiemy, że powinna ona poinformować nas, jakie linie przebiegają przez dowolne dwie stacje ❶ i o której godzinie pociągi z danej linii przyjeżdżają do wskazanej stacji ❷. Symbol >> dla frameworka Spock jest skrótem stwierdzenia: „Gdy wywołam tę metodę z tymi parametrami, zwróć takie wartości”.

Aby powyższy kod się skompilował, potrzebujemy klasy `TimetableService`. W Javie zwykle zdefiniowalibyśmy interfejs. Pozwoliłoby to na odłożenie właściwej implementacji klasy `TimetableService` na później — po zaimplementowaniu klasy `ItineraryService`. Metody, których potrzebuje klasa `TimetableService`, zdefiniowaliśmy w ❶ i ❷, zatem ten interfejs może wyglądać następująco:

```

package com.bddinaction.chapter2.services;

import com.bddinaction.chapter2.model.Line;
import org.joda.time.LocalTime;

import java.util.List;

public interface TimetableService {
    List<LocalTime> findArrivalTimes(Line line, String targetStation);
    List<Line> findLinesThrough(String departure, String destination);
}

```

Ostatnim elementem układanki jest metoda `findArrivalTimesAfter()`, która zwraca listę godzin odjazdów po określonej godzinie, tak jak pokazano w poniższej przykładowej implementacji:

```

private List<LocalTime> findArrivalTimesAfter(LocalTime startTime,
                                             List<LocalTime> times) {
    List<LocalTime> viableArrivalTimes = Lists.newArrayList();
}

```

```

for(LocalTime arrivalTime : times) {
    if (arrivalTime.isAfter(startTime)) {
        viableArrivalTimes.add(arrivalTime);
    }
}
return viableArrivalTimes;
}

```

Jeśli teraz uruchomimy ten test za pomocą polecenia `mvn verify`, powinien on przejść, co pokazuje nam, że teraz mamy działającą usługę tras. Jednak na tym praca jeszcze się nie kończy. Działanie usługi tras bazuje na założeniu, że usługa rozkładu jazdy prawidłowo wykonuje swoje zadanie i używa „atrapy” usługi rozkładu jazdy (znanej pod nazwą namiastki — ang. *stub* — lub makiety — ang. *mock*), aby uniknąć konieczności wykorzystania rzeczywistej implementacji. Jest to bardzo skuteczny sposób zbudowania usługi tras, ponieważ pozwala skoncentrować się wyłącznie na logice biznesowej tej usługi. Ale wracając do spełnienia sformułowanego kryterium akceptacji, należy również zaimplementować usługę rozkładu jazdy.

Zachowanie zdefiniowane dla makiety usługi rozkładu jazdy dostarcza bardzo precyzyjnych niskopoziomowych wymagań co do zachowania tej usługi. Definicja ta będzie punktem wyjścia do implementacji tej usługi. Następnie napiszemy nowy test Spock, o nazwie `WhenFindingLinesThroughStations.groovy` (ponownie w pakiecie `com.bddinaction.chapter2.services`), który opiera się na tych wymaganiach i bardziej szczegółowo opisuje, co usługa rozkładu jazdy powinna robić:

```

package com.bddinaction.chapter2.services

import com.bddinaction.chapter2.model.Line
import spock.lang.Specification

class WhenFindingLinesThroughStations extends Specification {

    def timetableService = new TimetableService()

    def "powinna znaleźć prawidłowe linie pomiędzy dwoma stacjami"() {
        when:
            def lines = timetableService.findLinesThrough(departure,
                                                         destination) | Testowane
                                                         | działanie.

        then:
            def expectedLine = Line.named(lineName).
                                   departingFrom(lineDeparture) | Usługa rozkładu
                                                         | jazdy powinna
                                                         | zwrócić te linie
            lines == [expectedLine]

        where:
            departure | destination | lineName | lineDeparture |
            "Parramatta" | "Town Hall" | "Western" | "Emu Plains" |
            "Town Hall" | "Parramatta" | "Western" | "North Richmond" |
            "Strathfield" | "Epping" | "Epping" | "City" |
    }
}

```

Ten test bazuje na przykładzie kryterium akceptacji i służy do zbadania wymagań. Ale testy jednostkowe powinny być bardziej dokładne niż akceptacyjne. W tym przypadku skorzystaliśmy z tabeli ❶, podobnej do tabeli z listingu 2.3, wykorzystywanej przez

framework JBehave. To sprawia, że łatwo dodać bardziej wyczerpujący zbiór przykładów, a o to nam chodzi na tym poziomie szczegółowości.

Po zaimplementowaniu metody `findLinesThrough()` możemy przejść do implementowania kolejnej potrzebnej funkcji: wyszukiwania godzin przyjazdu na wskazaną stację. Do tego celu można zastosować podobne podejście — zacząć od napisania nowej specyfikacji Spock.

ĆWICZENIE 2.1. Napisz testy jednostkowe dla funkcji „znajdź godziny przyjazdu na podaną stację dla danej linii” i zaimplementuj odpowiedni kod.

Istnieje wiele możliwych sposobów implementacji tej usługi. W tym miejscu nie będziemy zagłębiać się w szczegóły. Po drodze jednak możemy odkryć inne usługi lub komponenty, które będą nam potrzebne. Te komponenty można zastąpić makieta, a następnie zaimplementować. Kiedy już nie będzie klas-atrap do zaimplementowania, kryterium akceptacji uruchomi się poprawnie, co oznacza zakończenie implementacji funkcji.

2.4.5. Testy jako dynamiczna dokumentacja

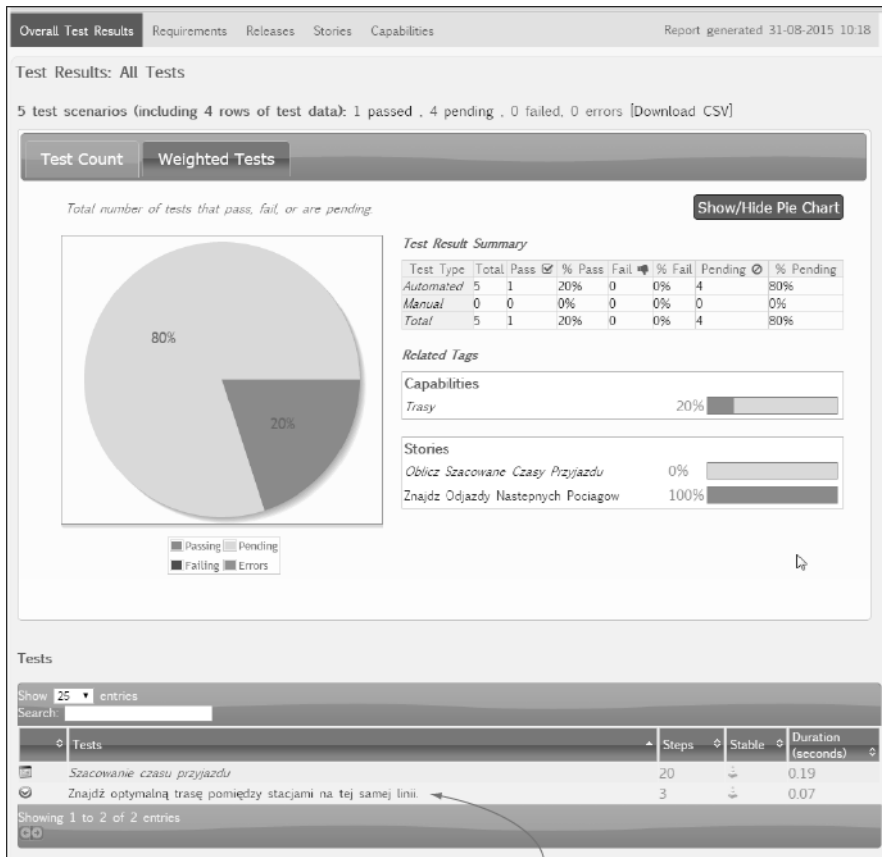
Po zaimplementowaniu funkcji powinno być możliwe uruchomienie testów. Obok testów zaległych (ang. *pending*) powinny się pojawić spełnione kryteria akceptacji (patrz rysunek 2.9). W przypadku stosowania takich praktyk jak BDD ten wynik to coś więcej niż po prostu wskaźnik, że aplikacja spełnia wymagania biznesowe. Przechodzący test akceptacji jest również konkretną miarą postępów. Zaimplementowany test albo przechodzi, albo kończy się niepowodzeniem. Najlepiej, jeśli wszystkie kryteria akceptacji dla funkcji są zautomatyzowane i pomyślnie przechodzą. Wtedy można powiedzieć, że ta funkcja jest skończona i gotowa do produkcji.

Stan testów daje więcej niż tylko ocenę jakości aplikacji — jest on wyraźnym wskaźnikiem tego, w jakim miejscu w procesie rozwoju się znajdujemy. Proporcja pomiędzy przechodzącymi testami a całkowitą liczbą zdefiniowanych kryteriów akceptacji daje dobry obraz tego, ile pracy zrealizowano do tej pory, a ile pozostało do zrobienia. Ponadto, dzięki śledzeniu liczby zakończonych automatycznych testów akceptacji w porównaniu z liczbą zaległych testów można uzyskać obraz postępów projektu w czasie.

Pisanie testów w takim narracyjnym stylu przynosi także inne korzyści. Każdy automatyczny test akceptacji staje się udokumentowanym, działającym przykładem tego, jak można wykorzystać system do spełniania szczegółowych wymagań biznesowych. A w przypadku, gdy raporty z testów są w postaci stron WWW, działające przykłady będą nawet zilustrowane zrzutami ekranu.

2.5. Utrzymanie

W wielu organizacjach deweloperzy, którzy pracowali w początkowym projekcie, nie zajmują się utrzymaniem aplikacji, gdy ta zostanie przekazana do produkcji. Zamiast tego zadania utrzymania aplikacji są przekazywane do zespołu pielęgnacyjnego (ang. *Business as Usual* — BAU — dosł. biznes jak zwykle). W tego rodzaju środowisku specyfikacje



Test teraz przechodzi

Rysunek 2.9. W raportach z testów powinna się teraz pojawić informacja o tym, że test przechodzi

Rola testerów. Automatyczne testowanie akceptacyjne a kontrola jakości

Automatyczne wdrażanie aplikacji do produkcji, gdy przechodzą automatyczne testy akceptacji, wymaga dużo dyscypliny i ogromnego zaufania do jakości i kompletności zautomatyzowanych testów. Jest to godny cel i wielu organizacjom udało się go osiągnąć, ale w większości przypadków to nie jest takie proste.

W typowych środowiskach korporacyjnych testerzy zazwyczaj będą przeprowadzali co najmniej kilka testów ręcznych przed wdrożeniem aplikacji do produkcji. Ale jeśli wyniki testów automatycznych są oczywiste i widoczne, mogą one zaoszczędzić zespołowi QA wielu dni lub tygodni, które normalnie byłyby przeznaczone na testy regresji lub podstawowe mechaniczne testy. Dzięki temu członkowie zespołu mogą skupić się na bardziej interesujących zadaniach związanych z testowaniem. To z kolei może znacznie przyspieszyć cykl publikacji.

wykonywalne i dynamiczna dokumentacja są świetnym sposobem na usprawnienie procesu przekazywania aplikacji do produkcji, ponieważ dostarczają działających przykładów cech funkcjonalnych aplikacji wraz z ilustracjami kodu, który obsługuje te cechy.

Ponadto, specyfikacje wykonywalne znacznie ułatwiają zespołom zajmującym się utrzymaniem wdrażanie zmian lub poprawek. Zobaczmy na prostym przykładzie, jak to działa. Załóżmy, że użytkownicy poprosili o cechę funkcjonalną wyświetlającą informacje o pociągach, które mają przybyć w ciągu najbliższych 30 minut, a nie tylko następnych 15, jak obecnie.

Oto scenariusz związany z tym wymaganiem:

Dowiedz się, o której godzinie odjeżdżają następne pociągi do stacji docelowej

Narracja:

W celu bardziej efektywnego planowania podróży

Jako podróżny

Chcę się dowiedzieć, jakie następne pociągi odjeżdżają do mojej stacji docelowej

Scenariusz: Znajdź optymalną trasę pomiędzy stacjami na tej samej linii.

Zakładając pociągi linii Western z Emu Plains odjeżdżają ze stacji Parramatta do Town Hall

o 7:58, 8:00, 8:02, 8:11, 8:14, 8:21

Gdy chcę podróżować z Parramatta do Town Hall o 8:00

Wtedy powinienem uzyskać informację o pociągach o: 8:02, 8:11, 8:14

Ten scenariusz wyraża nasz bieżący sposób rozumienia wymagania: aplikacja obecnie zachowuje się w ten sposób i mamy automatyczne kryterium akceptacji oraz testy jednostkowe, które to udowadniają.

Jednak nowe żądanie użytkownika wszystko zmieniło. Scenariusz teraz powinien wyglądać następująco:

Potrzebujemy więcej przykładowych godzin odjazdu.

Scenariusz: Znajdź optymalną trasę pomiędzy stacjami na tej samej linii.

Zakładając pociągi linii Western z Emu Plains odjeżdżają ze stacji Parramatta do Town Hall

o 7:58, 8:00, 8:02, 8:11, 8:14, 8:21, 8:31, 8:36

Gdy chcę podróżować z Parramatta do Town Hall o 8:00

Wtedy powinienem uzyskać informację o pociągach o: 8:02, 8:11, 8:14, 8:21

Teraz chcemy wiedzieć o wszystkich pociągach odjeżdżających w ciągu 30 minut.

Po uruchomieniu tego nowego scenariusza okaże się, że zakończył się on niepowodzeniem (patrz rysunek 2.10). Wszystko w porządku! To pokazuje, że aplikacja nie robi tego, czego żądają wymagania. Mamy teraz punkt wyjścia do implementacji tej modyfikacji.

Możemy użyć testów jednostkowych do wyizolowania kodu, który powinien zostać zmieniony. Należy zaktualizować specyfikację Spock „powinna obliczyć prawidłową godzinę przyjazdu” tak, aby uwzględniła nowe kryterium akceptacji:

```
def "powinna obliczyć prawidłową godzinę przyjazdu"() {
  given:
    def westernLineFromEmuPlains =
      Line.named("Western").departingFrom("Emu Plains")

    timetableService.findLinesThrough("Parramatta","Town Hall") >>
      [westernLineFromEmuPlains]
```

Overall Test Results Requirements Releases Stories Capabilities Report generated 31-08-2015 10:34

🔍 Znajdź optymalną trasę pomiędzy stacjami na tej samej linii. 0,1s

Story: Znajdź Odjazdy Następnych Pociągów

Narracja:
 W celu bardziej efektywnego planowania podróży
 Jako podróżny
 Chcę się dowiedzieć, jakie następane pociągi odjeżdżają do mojej stacji docelowej
 Scenariusz:
 Znajdź optymalną trasę pomiędzy stacjami na tej samej linii.
 Zakładając pociągi linii Western z Emu Plains odjeżdżają ze stacji Parramatta do Town Hall o 7.58, 8.00, 8.02, 8.11, 8.14, 8.21
 Gdy chcę podróżować z Parramatta do Town Hall o 8.00
 Wtedy powinienem uzyskać informację o pociągach o: 8.02, 8.11, 8.14

Znajdź odjazdy następnych pociągów (story) Trasy (capability)

Steps	Outcome	Duration
👍 Zakładając pociągi linii [Western] z [Emu Plains] odjeżdżają ze stacji [Parramatta] do [Town Hall] o {7.58, 8.00, 8.02, 8.11, 8.14, 8.21}	SUCCESS	0,04s
👍 Gdy chcę podróżować z [Parramatta] do [Town Hall] o {8:00}	SUCCESS	0,01s
👎 Wtedy powinienem uzyskać informację o pociągach o: {8:02, 8:11, 8:14}	FAILURE	0,03s

Aplikacja już nie spełnia wymagań opisanych w żądaniu zmiany

Rysunek 2.10. Nieprzechodzące kryterium akceptacji ilustruje różnicę pomiędzy tym, jakie są wymagania, a tym, co aplikacja obecnie robi

```

timetableService
    .findArrivalTimes(westernLineFromEmuPlains, "Parramatta") >>
        [at(7,58), at(8,00), at(8,02), at(8,11), at(8,14),
         at(8,21), at(8,31), at(8,36)]

when:

def proposedTrainTimes = itineraryService.
    findNextDepartures("Parramatta", "Town Hall", at(8,00)):

then:
    proposedTrainTimes == [at(8,02), at(8,11),
                           at(8,14), at(8,21)]
}

```

Usługa-atrapa zwraca teraz więcej kursów.

Teraz oczekujemy od usługi tras zwrócenia czterech godzin.

To z kolei pomaga wyodrębnić kod, który trzeba zmienić wewnątrz klasy `ItineraryService`. Po wykonaniu tych działań poprawne zaktualizowanie kodu powinno przyjsić nam znacznie łatwiej.

Wprowadzenie większych zmian będzie oczywiście wiązało się z większą pracą, ale dla modyfikacji dowolnych rozmiarów obowiązuje taka sama zasada. Jeśli żądanie zmiany dotyczy modyfikacji istniejących cech funkcjonalnych, trzeba zaktualizować automatyczne kryterium akceptacji tak, by uwzględniało nowe wymaganie. Jeśli zmiana jest poprawką błędu, który nie został wykryty przez kryterium akceptacji w obecnej postaci, to najpierw trzeba napisać nowe automatyczne kryterium akceptacji, aby powtórzyć błąd, następnie naprawić błąd i na koniec użyć kryterium akceptacji, aby wykazać, że błąd został usunięty. A jeśli zmiana jest wystarczająco duża do tego, aby istniejące kryterium akceptacji stało się zbędne, można usunąć stare kryterium akceptacji i napisać nowe.

2.6. Podsumowanie

W tym rozdziale zapoznaliśmy się z ogólnym cyklem życia projektu BDD. W szczególności dowiedzieliśmy się, że:

- Zrozumienie podstawowych celów biznesowych projektu pozwala odkryć funkcje i historyjki, które przyczyniają się do osiągnięcia tych celów biznesowych.
- Funkcje opisują operacje, które pomogą użytkownikom i interesariuszom osiągnąć swoje cele.
- Funkcje można podzielić na historyjki, które są łatwiejsze do stworzenia i dostarczenia za jednym razem.
- Skutecznym sposobem opisywania i omawiania funkcji są konkretne przykłady.
- Przykłady, wyrażone w półstrukturalnej notacji „Zakładając... Gdy... Wtedy”, można zautomatyzować, tworząc automatyczne kryteria akceptacji.
- Kryteria akceptacji sterują niskopoziomowymi zadaniami implementacji. Pomagają zaprojektować i napisać tylko taki kod, którego naprawdę potrzebujemy.
- Strukturę BDD w stylu „Zakładając... Gdy... Wtedy” można również stosować w testach jednostkowych.
- Automatyczne kryteria akceptacji dokumentują również dostarczone funkcje w postaci dynamicznej dokumentacji.
- Zautomatyzowane kryteria akceptacji i testy jednostkowe w stylu BDD znacznie ułatwiają utrzymanie aplikacji.

W kolejnych rozdziałach omówimy znacznie bardziej szczegółowo każdy z tematów zasygnalizowanych w niniejszym rozdziale. Pokażemy również, jak można zastosować omówione podejścia w praktyce, stosując różne narzędzia i technologie.

Skorowidz

A

- adnotacja
 - @FindBy, 271
 - @Pending, 74
- aktor, 110
- analityk biznesowy, 28
- analiza wymagań, 60
- aplikacje
 - AJAX, 265
 - webowe, 246
- architektura
 - MVC, 291
 - zorientowana na usługi, SOA, 299
- ATDD, Acceptance-Test-Driven Development, 38
- automatyczna dokumentacja, 51
- automatyczne
 - kryteria akceptacji, 120
 - testowanie akceptacyjne, 82
 - testy, 28, 72
 - testy aplikacji webowych, 246
- automatyczny proces budowy, 374
- automatyzowanie
 - procesu konfiguracji testu, 228
 - scenariuszy, 179, 182
 - kryteriów akceptacji, 67, 245, 283, 317
 - w .NET, 211
 - w JavaScript, 216
 - w Javie, 202
 - w Pythonie, 207
- webowych kryteriów akceptacji, 251

B

- BAU, Business as Usual, 81
- BDD, Behavior-Driven Development, 23

biblioteka

- Geb, 279
- Jasmine, 341
- Selenium WebDriver, 246
- Spock, 47
- Thucydides, 279
- WatiN, 279
- Watir, 279
- WebDriver, 252, 279

biblioteki

- automatyzacji, 180
- obsługi kroków, 240

bramy jakości, 380

budowanie, 373

- niewłaściwego oprogramowania, 29, 33
- właściwego oprogramowania, 32

C

- cecha funkcjonalna, feature, 39, 62, 95, 114, 119–123, 134

- fragmenty, 126

- historyjki użytkowników, 128, 131

- Różnica dla rynku, 116

- Ważność dla misji, 116

- zdolności, 122

cechy

- o minimalnym wpływie, 117

- partnerskie, 117

- równoważne, 116

- wyróżniające, 116

- cele biznesowe, 60, 89, 95, 102

- celowe odkrywanie, 120, 144

- ciągła integracja, CI, 181, 378

ciągłe

- dostawy, 181, 380

- wdrażanie, 181

cykl sprzężenia zwrotnego, 378
czytelność testów, 267

D

dane tabelaryczne, 190
DDD, Domain-Driven Design, 37
definicje kroków, 182, 199, 206, 210, 317
 Behave, 209
 Cucumber-JVM, 203
 JBehave, 195
definiowanie
 cech funkcjonalnych, 119
 wymagań, 87
 wymagań niefunkcjonalnych, 304
dokumentacja, 49
 automatyczna, 51
 dynamiczna, 49–51, 71, 81, 345, 354, 364, 371
domena, 318
dostarczanie
 cech funkcjonalnych, 64
 dokumentacji, 366
działające oprogramowanie, 145

E

edytowanie pliku, 212
efektywna implementacja BDD, 193
elementy strony WWW, 255
eposy, epic, 120, 132

F

feature injection, 91
firma Flying High, 91
formuła wizji, 97
framework
 JBehave, 81, 194
 JUnit, 75
 Spock, 76
frameworki testów jednostkowych, 339, 341
funkcje, 60
funkcjonalność wysokopoziomowa, 69

G

Gherkin, 37
gotowość cech funkcjonalnych, 356
grupowanie cech funkcjonalnych, 363

H

haki, hooks, 175
haki inicjalizacji, 225, 229
 w Behave, 232
 w cucumber-jvm, 231
 w JBehave, 229
 w Specflow, 232
hasła, 139
hierarchia, 133
historyjka JBehave, 70
historyjki użytkownika, user stories, 120, 132

I

identyfikowanie
 celów biznesowych, 99
 elementów strony WWW, 255, 258, 261
 interesariuszy, 110
 zdolności, 112
ilustrowanie
 cech funkcjonalnych, 134
 historyjek, 63
impact mapping, 91
implementacja, 64
 definicji kroków, 186, 317
 kodu produkcyjnego, 330
 kroków, 74, 187, 218
 kryteriów akceptacji, 72
 minimalna, 331
 natychmiastowa, 331
 scenariusza, 73
 złożonych funkcjonalności, 325
 interfejsu WebDriver, 255
inicjowanie bazy danych, 228
instalowanie
 systemu Behave, 208
 narzędzi BDD, 186
 narzędzia JBehave, 194
interakcje
 z elementami stron WWW, 263
 z systemem, 241
interesariusz, 33, 61, 110, 153

interfejs

- użytkownika, UI, 246
- WebDriver, 255

K

klasa

- Groovy, 75
- TimetableService, 79

klasy fixture, 367

kod z zewnątrz do wewnątrz, 314, 315

komentarze, 159

konfigurowanie

- danych specyficznych, 233
- narzędzia JBehave, 194
- projektu Cucumber-JVM, 202
- SpecFlow, 211
- systemu Cucumber-JS, 216
- testu, 228

kontekst, 168, 338

kontrola

- jakości, 82
- zmian, 103

korzystanie z haków, 229

korzyści, 52

koszty, 52

- utrzymania, 54

krok

- @Then, 74
- Gdy, 166, 197
- Wtedy, 167
- Zakładając, 165, 197

kroki

- Behave, 209
- Cucumber-JVM, 203
- JBehave, 195
- oczekujące, 207
- SpecFlow, 213
- tła, 169

kryteria akceptacji, 37, 58, 64, 72, 84, 228, 385, 388

L

logika biznesowa, 295

Ł

łącznie, 257

łączenie kroków, 209

M

makieta, mock, 80, 332

mapowanie wpływu, impact mapping, 91, 106

marnotrawstwo, 52

metodologia

- Agile, 53
- Unified Process, 108

metodologie iteracyjne, 53

mistrz młyna, scrum master, 361

model

- domeny, 318
- dopasowania do celów, 91, 114
- dopasowania do celu, 115

N

namiastka, stub, 80, 332

narzędzia

- analizy wymagań, 37
- BDD, 186
- testów jednostkowych, 75, 313, 336

narzędzie

- Behat, 65
- Cucumber, 64, 65
- Cucumber-JVM, 202
- Git, 65
- Jasmine, 343
- JBehave, 46, 59, 65, 194
- Maven, 65
- NSpec, 342
- RSpec, 339
- SpecFlow, 65
- Spock, 75, 76
- Thucydides, 193

niewiadome, 34

niewłaściwe budowanie oprogramowania, 29, 33

niskopoziomowa specyfikacja, 324

- wykonywalna, 47
- techniczna, 333

notacja <...>, 44

O

obiekt TimetableService, 78

obiekty stron, 242, 269, 273, 276, 279

oddzielanie warstw, 237

- odkrywanie projektu, 306
- ograniczenia wiedzy, 32
- opcja, 140
- opcje realne, 142
- opisywanie
 - cech funkcjonalnych, 94, 211
 - funkcji, 60
 - scenariuszy, 156
- oprogramowanie sterowane testami, 311
- organizowanie
 - dynamicznej dokumentacji, 362, 363, 364
 - scenariuszy, 171
- osoby użytkownika, 390

P

- parametry tabelaryczne, 198
- persony, personas, 225, 235
- pisanie
 - dobrych celów biznesowych, 100
 - ekspresywnych kroków, 165–167
 - plików opisu, 217
 - testów akceptacji, 225
 - wykonywalnych scenariuszy, 154
- plik opisu cechy funkcjonalnej, 43, 154, 172
- plynne
 - asercje, 347
 - kodowanie, 346
 - selektory, 281
- początkowa struktura projektu, 67
- podział
 - cech funkcjonalnych, 62
 - odpowiedzialności, 306
- pokrycie cech funkcjonalnych, 356, 357
- poła
 - tekstowe, 263
 - wyboru, 264
- potok budowy, 381
- prace konserwacyjne, 50
- praktyki BDD, 38
- problemy, 53
- proces
 - budowania, 373
 - rozwoju oprogramowania, 28
- program
 - Frequent Flyer, 107, 152, 300
 - Selenium WebDriver, 251
- programista, 28

- programowanie
 - dziedziczne, DDD, 37
 - sterowane testami, TDD, 31, 36
 - sterowane testami akceptacyjnymi, ATDD, 38
 - sterowane zachowaniami, BDD, 23
- projekt
 - Behave, 208
 - Cucumber-JVM, 202
 - typu silos, 54
- przekazywanie
 - parametrów, 187
 - tabel, 205
 - tabel do kroków, 198
- przepływ pracy, 239
- przyciski opcji, 264
- przypadki użycia, use cases, 120
- pseudostukturalny format, 150
- publikacje, 53
- publikowanie dynamicznej dokumentacji, 384, 385

R

- raportowanie, 49, 353
- realne opcje, 140
- refaktoryzacja, 322
- rejestr, backlog, 93
- rezultaty oczekiwane, 238
- rodzaje rozmów, 147
- rozkład jazdy pociągów, 58
- rozmowy, 147, 290
- rozszerzenie SpecFlow, 211
- rozwijane listy, 264

S

- scenariusz JBehave, 73
- scenariusze
 - automatyzacja, 179
 - automatyzowanie, 182, 194, 207, 211, 216
 - bazujące na przykładach, 191
 - błędy, 200
 - dane specyficzne, 233
 - krok, 201
 - niepowodzenia, 200
 - oczekujące, 192
 - opisywanie, 156, 174
 - organizowanie, 171

- tagi, 174
- unikanie zależności, 170
- uruchamianie, 213
- w opracowaniu, 192
- wykorzystanie tabel, 160
- scenariusze
 - ekspresywne, 165
 - kryteriów akceptacji, 317
 - wykonywalne, 151
- SDS, System Design Specification, 48
- selektory CSS, 258
- Selenium Grid, 392
- serwer Selenium Grid, 391
- serwis GitHub, 66
- sieć kolejowa, 59
- siła hasła, 139
- słowa kluczowe, 43, 64, 158
- SOA, Service Oriented Architecture, 299
- specyfikacja
 - cech funkcjonalnych, 40
 - niskopoziomowa, 47
 - przez przykład, 38
 - samowystarczalna, 375
 - wykonywalna, 45, 67
- Spock, 47
- stan pomiędzy krokami, 188
- stosowanie praktyk BDD, 53
- strategia ciągłej integracji, 378, 383
- strona WWW, 255
- struktura
 - „Zakładając... Gdy... Wtedy”, 157
 - pakietu, 74
 - projektu Behave, 208
 - projektu Cucumber-JVM, 202
- system
 - Behave, 208
 - Concordion, 367
 - Cucumber-JS, 216
 - ECSS, 25
 - kontroli wersji, 376
 - Selenium Grid, 394
- sablon formuły wizji, 98
- szacowanie godziny przyjazdu, 70

T

- tabele, 160
 - osadzone, 210
 - przykładów, 206

- tablice zadań, task boards, 360
- tag, 171
- TDD, Test-Driven Development, 31, 35, 76, 311
- techniczna dynamiczna dokumentacja, 368
- tematy, themes, 120
- tester, 28, 180
- testowanie
 - aplikacji AJAX, 265
 - getterów i setterów, 322
 - interfejsu użytkownika, 247
 - logiki biznesowej, 295
 - usług sieciowych RESTFUL, 300
 - warstwy usług, 299
 - wymagań niefunkcjonalnych, 304
- testy
 - akceptacyjne, 82, 290
 - akceptacyjne użytkownika, 374
 - automatyczne, 72, 224
 - jako dynamiczna dokumentacja, 81
 - jednostkowe, 48, 76, 309, 336, 369
 - w JUnit, 329
 - w Spock, 80, 329
 - nieuwzględniające UI, 250
 - NUnit, 342
 - równoległe, 388
 - webowe, 247, 248
- tło, 168
- tradycyjny proces rozwoju, 28
- tryb headless, 248
- tworzenie raportów, 364
- typy automatycznych testów akceptacji, 290

U

- UAT, 374
- UI, user interface, 246, 285
- uruchamianie scenariuszy, 213, 219
 - w Behave, 210
- uściślanie celów biznesowych, 103
- uwzględnianie niepewności, 41
- użytkownik, 33
- używanie
 - plików cech funkcjonalnych, 171
 - płynnego kodowania, 346
 - testów akceptacji, 286

W

wady, 53
 warianty wzorców, 200, 204
 warstwa
 przepływu pracy, 239
 reguł biznesowych, 238
 techniczna, 241
 usług, 299
 warstwy
 abstrakcji, 237
 interfejsu użytkownika, 245
 wartości biznesowe, 39
 wartość opcji, 141
 webowe kryteria akceptacji, 251
 wiersz polecenia, 377
 wizja projektu, 95–97
 właściwe budowanie oprogramowania, 30
 wprowadzenie do BDD, 34
 wskaźnik ROI, 114
 współdzielenie danych, 197, 206, 214
 wstrzykiwanie
 cech funkcjonalnych, 89–96
 funkcji, 91, 92
 wygasanie opcji, 143
 wykonywalne
 scenariusze, 150, 151
 specyfikacje, 42, 149
 wykorzystanie
 danych tabelarycznych, 190
 jawnego oczekiwania, 266
 kodu definicji kroku, 319
 niejawnego oczekiwania, 266
 Selenium Grid, 391
 tabel, 161
 tagów, 364
 UI, 285
 wymagania, requirements, 87, 120
 niefunkcjonalne, 304
 niekorzystające z UI, 283
 niskopoziomowe, 325
 wysokopoziomowe, 110

wyniki
 kroków, 207
 programowania BDD, 39
 scenariusza, 192
 wyrażenia XPath, 261
 wysokopoziomowe
 cechy funkcjonalne, 69, 108
 kryterium akceptacji, 316
 wymagania, 110, 312
 wysokopoziomowy opis cechy, 94
 wyszukiwanie
 wartości, 93
 zagnieżdżone, 263
 wzorzec, 200, 204
 architektury MVC, 291
 Obiekty stron, 268

Z

zadania, tasks, 120
 zakres projektu, 61
 zarządzanie
 bazujące na celach, 104
 niepewnością, 120
 projektem, 353
 zasady BDD, 38
 zautomatyzowane
 kryteria akceptacji, 385, 389
 testy akceptacyjne, 385
 zautomatyzowany proces budowy, 386
 zdolność, capability, 112, 120, 122
 zespół pielęgnacyjny, BAU, 81
 zestaw, suite, 375
 testów, test suite, 224
 wysokopoziomowych wymagań, 110
 znane encje, 235
 zobowiązania, 143
 zrefaktoryzowanie implementacji, 36

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**

Rozwój technik BDD jest odpowiedzią na poważny problem, z którym muszą się zmierzyć zespoły rozwijające oprogramowanie. Tym problemem jest skuteczne komunikowanie i zrozumienie się nawzajem. Jeśli jesteś kierownikiem projektu, musisz jakoś skłonić programistę do pisania testów, namówić testera do ich zaakceptowania i przekonać inwestora, że coś, co nie jest kodem produkcyjnym, może mieć swoją wartość. Okazuje się, że kluczem do sukcesu jest doprowadzenie do sytuacji, w której każdy rozumie, do czego ma służyć aplikacja, jak się ma zachować i jakie są jej kluczowe funkcje. Świetnym narzędziem ułatwiającym taką pracę jest technika BDD – obszerny zbiór najlepszych praktyk i narzędzi wspomagających analizę wymagań i automatyzację testów.

Książka, którą trzymasz w dłoni, stanowi przegląd praktyk BDD na wszystkich poziomach procesu rozwoju oprogramowania. Znajdziesz w niej informacje na temat odkrywania i określania wysokopoziomowych wymagań, implementacji funkcji aplikacji oraz pisania automatycznych testów akceptacyjnych i jednostkowych. Jest ona niezastąpionym przewodnikiem dla analityków biznesowych, deweloperów, testerów, a przede wszystkim liderów i menedżerów projektów.

Dzięki tej książce poznasz:

- teorię i praktykę BDD
- zasady stosowania BDD w pracy zespołowej
- testy akceptacyjne, integracyjne i jednostkowe BDD
- praktyczne przykłady w Javie, .NET, JavaScriptcie i innych językach
- sposoby tworzenia raportów i dynamicznej dokumentacji BDD

John Ferguson Smart – światowej klasy specjalista w dziedzinie BDD, automatycznego testowania i optymalizacji rozwoju oprogramowania w całym cyklu życia, umiejętnie łączący wiedzę programisty i zalety coacha.

Już dziś przedstaw swojemu zespołowi rewolucyjne techniki BDD!

40878	numer katalogowy	Sprawdź najnowsze promocje: ● http://helion.pl/promocje Książki najchętniej czytane: ● http://helion.pl/bestsellery Zamów informacje o nowościach: ● http://helion.pl/nowosci	
	http://helion.pl	 KOD KORZYŚCI	
	zamówienia telefoniczne		
	0 801 339900	Helion SA ul. Kościuszki 1c, 44-100 Gliwice tel.: 32 230 98 63 e-mail: helion@helion.pl http://helion.pl	
	0 601 339900	ISBN 978-83-283-1747-5 9 788328 317475	
Informatyka w najlepszym wydaniu		cena: 77,00 zł	