

Robert C. Martin

# ZWINNE WYTWARZANIE OPROGRAMOWANIA

Najlepsze zasady, wzorce i praktyki

Poznaj nowoczesne  
sposoby wytwarzania  
oprogramowania!

0110 0001  
1000 1000  
0100 1010  
0000 0000  
1011 1110  
1010 1001  
0000 1111

1110 1001  
0000 0000  
0100 0000

Helion 

Tytuł oryginału: Agile Software Development, Principles, Patterns, and Practices

Tłumaczenie: Radosław Meryk

Projekt okładki: Studio Gravite / Olsztyn; Obarek, Pokoński, Pazdrijowski, Zaprucki  
Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock Images LLC.

ISBN: 978-83-246-9682-6

Authorized translation from the English language edition, entitled:  
AGILE SOFTWARE DEVELOPMENT, PRINCIPLES, PATTERNS, AND PRACTICES;  
ISBN 0135974445; by Robert C. Martin; published by Pearson Education, Inc;  
publishing as Prentice Hall. Copyright © 2003 by Pearson Education, Inc.

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

Polish language edition published by HELION S.A., Copyright © 2015.

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.

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

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

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

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:

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

Printed in Poland.

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

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

---

# Spis treści

<b>Słowo wstępne</b>	<b>13</b>	
<b>Przedmowa</b>	<b>14</b>	
<b>O autorach</b>	<b>20</b>	
<hr/>		
<b>CZĘŚĆ I</b>	<b>ZWINNE WYTWARZANIE OPROGRAMOWANIA</b>	<b>21</b>
<hr/>		
<b>Rozdział 1</b>	<b>Praktyki agile</b>	<b>23</b>
	Agile Alliance	24
	Manifest Agile Alliance	24
	Zasady	27
	Wniosek	29
	Bibliografia	29
<b>Rozdział 2</b>	<b>Przegląd informacji o programowaniu ekstremalnym</b>	<b>31</b>
	Praktyki programowania ekstremalnego	31
	Klient jest członkiem zespołu	32
	Historyjki użytkowników	32
	Krótkie cykle	32
	Testy akceptacyjne	33
	Programowanie parami	33
	Programowanie sterowane testami	34
	Wspólna własność	34
	Ciągła integracja	34
	Równomierne tempo	35
	Otwarta przestrzeń robocza	35
	Gra w planowanie	35
	Prosty projekt	36
	Refaktoryzacja	37
	Metafora	37
	Wniosek	38
	Bibliografia	38
<b>Rozdział 3</b>	<b>Planowanie</b>	<b>39</b>
	Początkowa eksploracja	40
	Tworzenie prototypów, dzielenie i szybkość	40
	Planowanie wersji dystrybucyjnych	41
	Planowanie iteracji	41
	Planowanie zadań	41
	Półmetek	42
	Przebieg iteracji	42
	Wniosek	43
	Bibliografia	43

<b>Rozdział 4</b>	<b>Testowanie</b>	<b>45</b>
	Programowanie sterowane testami	45
	Przykład projektu w stylu „najpierw test”	46
	Izolacja testu	47
	Nieoczekiwane wyeliminowanie sprzężeń	48
	Testy akceptacyjne	49
	Przykład testów akceptacyjnych	50
	Architektura „przy okazji”	51
	Wniosek	51
	Bibliografia	52
<b>Rozdział 5</b>	<b>Refaktoryzacja</b>	<b>53</b>
	Generowanie liczb pierwszych — prosty przykład refaktoryzacji	54
	Ostateczny przegląd	59
	Wniosek	62
	Bibliografia	63
<b>Rozdział 6</b>	<b>Epizod programowania</b>	<b>65</b>
	Gra w kręgle	66
	Wniosek	98
<hr/>		
<b>CZĘŚĆ II</b>	<b>PROJEKT AGILE</b>	<b>101</b>
<hr/>		
	Symptomy złego projektu	101
	Zasady	101
	Zapachy a zasady	102
	Bibliografia	102
<b>Rozdział 7</b>	<b>Co to jest projekt agile?</b>	<b>103</b>
	Co złego dzieje się z oprogramowaniem?	103
	Zapachy projektu — woń psującego się oprogramowania	104
	Co stymuluje oprogramowanie do psucia się?	106
	Zespoły agile nie pozwalają psuć się oprogramowaniu	106
	Program Copy	106
	Przykład programu Copy wykonanego zgodnie z metodyką agile	109
	Skąd deweloperzy agile wiedzieli, co należy zrobić?	110
	Utrzymywanie projektu w jak najlepszej postaci	110
	Wniosek	111
	Bibliografia	111
<b>Rozdział 8</b>	<b>SRP — zasada pojedynczej odpowiedzialności</b>	<b>113</b>
	SRP — zasada pojedynczej odpowiedzialności	113
	Czym jest odpowiedzialność?	115
	Rozdzielanie sprzężonych odpowiedzialności	115
	Trwałość	116
	Wniosek	116
	Bibliografia	116
<b>Rozdział 9</b>	<b>OCP — zasada otwarte-zamknięte</b>	<b>117</b>
	OCP — zasada otwarte-zamknięte	117
	Opis	118
	Kluczem jest abstrakcja	118

Aplikacja Shape	119
Naruszenie zasady OCP	120
Zachowanie zgodności z zasadą OCP	121
Przyznaję się. Kłamałem	122
Przewidywanie i „naturalna” struktura	122
Umieszczanie „haczyków”	123
Stosowanie abstrakcji w celu uzyskania jawnego domknięcia	124
Zastosowanie podejścia „sterowania danymi” w celu uzyskania domknięcia	125
Wniosek	126
Bibliografia	126
<b>Rozdział 10 LSP — zasada podstawiania Liskov</b>	<b>127</b>
LSP — zasada podstawiania Liskov	127
Prosty przykład naruszenia zasady LSP	128
Kwadraty i prostokąty — bardziej subtelne naruszenie zasady LSP	129
Prawdziwy problem	131
Poprawność nie jest wrodzona	132
Relacja IS-A dotyczy zachowania	132
Projektowanie według kontraktu	132
Specyfikowanie kontraktów w testach jednostkowych	133
Realny przykład	133
Motywacja	133
Problem	135
Rozwiązanie niezgodne z zasadą LSP	136
Rozwiązanie zgodne z zasadą LSP	136
Wydzielanie zamiast dziedziczenia	137
Heurystyki i konwencje	139
Zdegenerowane funkcje w klasach pochodnych	140
Zgłaszanie wyjątków z klas pochodnych	140
Wniosek	140
Bibliografia	140
<b>Rozdział 11 DIP — zasada odwracania zależności</b>	<b>141</b>
DIP — zasada odwracania zależności	141
Podział na warstwy	142
Odwrócenie własności	142
Zależność od abstrakcji	143
Prosty przykład	144
Wyszukiwanie potrzebnych abstrakcji	145
Przykład programu Furnace	146
Polimorfizm dynamiczny i statyczny	147
Wniosek	148
Bibliografia	148
<b>Rozdział 12 ISP — zasada segregacji interfejsów</b>	<b>149</b>
Zaśmiecanie interfejsów	149
Odrębne klienty oznaczają odrębne interfejsy	150
Siła oddziaływania klientów na interfejsy	151
ISP — zasada segregacji interfejsów	151
Interfejsy klas a interfejsy obiektów	152
Separacja przez delegację	152
Separacja przez wielokrotne dziedziczenie	153

Przykład interfejsu użytkownika bankomatu	153
Poliady i monady	158
Wniosek	159
Bibliografia	159
<b>CZĘŚĆ III</b>	<b>STUDIUM PRZYPADKU: SYSTEM PŁACOWY</b>
	<b>161</b>
Szczątkowa specyfikacja systemu płacowego	162
Ćwiczenie	162
Przypadek użycia nr 1: dodawanie nowego pracownika	162
Przypadek użycia nr 2: usuwanie pracownika	163
Przypadek użycia nr 3: dostarczenie karty pracy	163
Przypadek użycia nr 4: dostarczenie raportu sprzedaży	163
Przypadek użycia nr 5: dostarczenie informacji o opłacie na rzecz związku zawodowego	164
Przypadek użycia nr 6: zmiana danych pracownika	164
Przypadek użycia nr 7: wygenerowanie listy płac na dzień	164
<b>Rozdział 13</b>	<b>Wzorce projektowe Polecenie i Aktywny obiekt</b>
	<b>165</b>
Proste polecenia	166
Transakcje	167
Fizyczny i czasowy podział kodu	168
Czasowy podział kodu	168
Metoda Undo	169
Aktywny obiekt	169
Wniosek	173
Bibliografia	173
<b>Rozdział 14</b>	<b>Metoda szablonowa i Strategia: dziedziczenie a delegacja</b>
	<b>175</b>
Metoda szablonowa	176
Nadużywanie wzorca	178
Sortowanie bąbelkowe	179
Strategia	181
Sortowanie jeszcze raz	183
Wniosek	185
Bibliografia	185
<b>Rozdział 15</b>	<b>Wzorce projektowe Fasada i Mediator</b>
	<b>187</b>
Fasada	187
Mediator	188
Wniosek	190
Bibliografia	190
<b>Rozdział 16</b>	<b>Wzorce projektowe Singleton i Monostate</b>
	<b>191</b>
Singleton	192
Korzyści ze stosowania wzorca Singleton	193
Koszty stosowania wzorca Singleton	193
Wzorzec projektowy Singleton w praktyce	193
Monostate	194
Korzyści ze stosowania wzorca Monostate	196
Koszty stosowania wzorca Monostate	196
Wzorzec projektowy Monostate w praktyce	196
Wniosek	200
Bibliografia	200



<b>Rozdział 17 Wzorzec projektowy Obiekt Null</b>	<b>201</b>
Wniosek	204
Bibliografia	204
<b>Rozdział 18 Studium przypadku: system płacowy. Pierwsza iteracja</b>	<b>205</b>
Wprowadzenie	205
Specyfikacja	206
Analiza według przypadków użycia	206
Dodawanie pracowników	207
Usuwanie pracowników	208
Dostarczenie karty pracy	209
Dostarczenie raportów sprzedaży	209
Dostarczenie informacji o opłacie na rzecz związku zawodowego	210
Zmiana danych pracownika	210
Wypłaty	212
Refleksja: czego się nauczyliśmy?	214
Wyszukiwanie potrzebnych abstrakcji	214
Abstrakcja harmonogramu	214
Sposoby wypłaty	215
Przynależność do związków zawodowych	216
Wniosek	216
Bibliografia	216
<b>Rozdział 19 Studium przypadku: system płacowy. Implementacja</b>	<b>217</b>
Dodawanie pracowników	218
Baza danych systemu płacowego	219
Zastosowanie wzorca Metoda szablonowa do dodawania pracowników	220
Usuwanie pracowników	223
Zmienne globalne	225
Karty pracy, raporty sprzedaży i składki	225
Zmiana danych pracowników	231
Zmiana klasyfikacji	235
Co ja paliłem?	240
Realizacja wypłat	244
Czy chcemy, aby deweloperzy podejmowali decyzje biznesowe?	246
Realizacja wypłat dla pracowników ze stałą pensją	246
Realizacja wypłat dla pracowników zatrudnionych w systemie godzinowym	248
Okresy rozliczeniowe: problem projektowy	251
Program główny	257
Baza danych	257
Podsumowanie projektu systemu płacowego	258
Historia	259
Zasoby	259
Bibliografia	259
<b>CZĘŚĆ IV PODZIAŁ SYSTEMU PŁACOWEGO NA PAKIETY</b>	<b>261</b>
<b>Rozdział 20 Zasady projektowania pakietów</b>	<b>263</b>
Projektowanie z wykorzystaniem pakietów?	263
Ziarnistość: zasady spójności pakietów	264
Zasada równoważności wielokrotnego wykorzystania kodu i dystrybucji (REP)	264
Zasada zbiorowego wielokrotnego użytku (CRP)	265

Zasada zbiorowego zamykania (CCP)	266
Podsumowanie tematyki spójności pakietów	266
Stabilność: zasady sprzęgania pakietów	267
Zasada acyklicznych zależności (ADP)	267
Cotygodniowe kompilacje	267
Eliminowanie cykli zależności	268
Skutki istnienia cykli w grafie zależności między pakietami	269
Przerywanie cykli	270
Odchylenia	270
Projekt góra-dół	271
Zasada stabilnych zależności (SDP)	272
Stabilność	272
Metryki stabilności	273
Nie wszystkie pakiety muszą być stabilne	274
Gdzie powinna się znaleźć implementacja projektu wysokiego poziomu?	276
Zasada stabilnych abstrakcji (SAP)	276
Mierzenie abstrakcji	276
Ciąg główny	277
Odległość od ciągu głównego	278
Wniosek	280
<b>Rozdział 21 Wzorzec projektowy Fabryka</b>	<b>281</b>
Cykl zależności	283
Fabryki wymienne	284
Wykorzystanie wzorca Fabryka do tworzenia zestawów testowych	284
Znaczenie korzystania z fabryk	286
Wniosek	287
Bibliografia	287
<b>Rozdział 22 Studium przypadku: system płacowy (część 2.)</b>	<b>289</b>
Struktura pakietów i notacja	290
Zastosowanie zasady zbiorowego domykania (CCP)	291
Zastosowanie zasady równoważności	
wielokrotnego wykorzystania kodu i dystrybucji (REP)	292
Sprzężenia i hermetyzacja	294
Metryki	296
Zastosowanie wskaźników do aplikacji płacowej	297
Fabryki obiektów	300
Fabryka obiektów dla pakietu TransactionImplementation	300
Inicjowanie fabryk	301
Przebudowa granic spójności	301
Ostateczna struktura pakietów	302
Wniosek	304
Bibliografia	304
<b>CZĘŚĆ V</b>	<b>STUDIUM PRZYPADKU: STACJA POGODOWA</b>
	<b>305</b>
<b>Rozdział 23 Wzorzec projektowy Kompozyt</b>	<b>307</b>
Przykład: polecenia kompozytowe	308
Wielokrotność czy brak wielokrotności	309



<b>Rozdział 24</b>	<b>Obserwator — ewolucja kodu do wzorca</b>	<b>311</b>
	Zegar cyfrowy	311
	Wniosek	326
	Wykorzystanie diagramów w tym rozdziale	327
	Wzorzec projektowy Obserwator	327
	Zarządzanie zasadami projektu obiektowego dla wzorca projektowego Obserwator	328
	Bibliografia	329
<b>Rozdział 25</b>	<b>Wzorce projektowe Serwer abstrakcyjny i Most</b>	<b>331</b>
	Wzorzec projektowy Serwer abstrakcyjny	332
	Kto jest właścicielem interfejsu?	333
	Wzorzec projektowy Adapter	333
	Wzorzec projektowy Adapter w formie klasy	334
	Problem modemu. Adaptery i zasada LSP	334
	Wzorzec projektowy Most	338
	Wniosek	339
	Bibliografia	340
<b>Rozdział 26</b>	<b>Wzorce projektowe Pełnomocnik i Schody do nieba</b>	
	— zarządzanie zewnętrznymi interfejsami API	<b>341</b>
	Wzorzec projektowy Pełnomocnik	342
	Implementacja wzorca projektowego Pełnomocnik w aplikacji koszyka na zakupy	345
	Podsumowanie wiadomości o wzorcu projektowym Pełnomocnik	356
	Obsługa baz danych, oprogramowania middleware oraz zewnętrznych interfejsów API	357
	Schody do nieba	359
	Przykład zastosowania wzorca Schody do nieba	360
	Wniosek	365
	Inne wzorce projektowe, które można wykorzystywać z bazami danych	365
	Wniosek	366
	Bibliografia	366
<b>Rozdział 27</b>	<b>Analiza przypadku: stacja pogodowa</b>	<b>367</b>
	Firma Chmura	367
	Oprogramowanie WMS-LC	369
	Wybór języka	369
	Projekt oprogramowania systemu Nimbus-LC	369
	Historia 24-godzinna i utrwalanie	382
	Implementacja algorytmów HiLo	384
	Wniosek	391
	Bibliografia	391
	Przegląd wymagań dla oprogramowania Nimbus-LC	391
	Wymagania użytkowe	391
	Historia 24-godzinna	392
	Konfiguracja użytkownika	392
	Wymagania administracyjne	392
	Przypadki użycia systemu Nimbus-LC	393
	Aktorzy	393
	Przypadki użycia	393
	Historia pomiarów	393
	Konfiguracja	393
	Administracja	393

Plan publikacji wersji dystrybucyjnych systemu Nimbus-LC	394
Wprowadzenie	394
Wydanie I	394
Zagrożenia	394
Produkty projektu	395
Wydanie II	395
Zaimplementowane przypadki użycia	395
Zagrożenia	395
Produkty projektu	395
Wydanie III	396
Zaimplementowane przypadki użycia	396
Zagrożenia	396
Produkty projektu	396
<hr/>	
<b>CZĘŚĆ VI   STUDIUM PRZYPADKU: ETS</b>	<b>397</b>
<hr/>	
<b>Rozdział 28 Wzorzec projektowy Wizytator</b>	<b>399</b>
Rodzina wzorców projektowych Wizytator	400
Wizytator	400
Wzorzec projektowy Wizytator działa jak macierz	403
Wzorzec projektowy Acykliczny wizytator	403
Wzorzec projektowy Wizytator działa jak macierz rzadka	407
Wykorzystanie wzorca projektowego Wizytator w generatorach raportów	407
Inne zastosowania wzorca projektowego Wizytator	412
Wzorzec projektowy Dekorator	413
Wiele dekoratorów	416
Wzorzec projektowy Obiekt rozszerzenia	418
Wniosek	426
Przypomnienie	426
Bibliografia	426
<b>Rozdział 29 Wzorzec projektowy Stan</b>	<b>427</b>
Przegląd informacji o automatach stanów skończonych	427
Techniki implementacji	429
Zagnieżdżone instrukcje Switch/Case	429
Interpretacja tabeli przejść	432
Wzorzec projektowy Stan	433
SMC — kompilator maszyny stanów	436
Kiedy należy korzystać z maszyn stanów?	439
Wysokopoziomowe strategie obsługi GUI	439
Kontrolery interakcji z GUI	440
Przetwarzanie rozproszone	441
Wniosek	441
Listingi	441
Implementacja klasy Turnstile.java z wykorzystaniem interpretacji tabeli przejść	441
Klasa Turnstile.java wygenerowana przez kompilator SMC oraz inne pliki pomocnicze	443
Bibliografia	447
<b>Rozdział 30 Framework ETS</b>	<b>449</b>
Wprowadzenie	449
Przegląd informacji o projekcie	449
Wczesny okres 1993 – 1994	451
Framework?	451

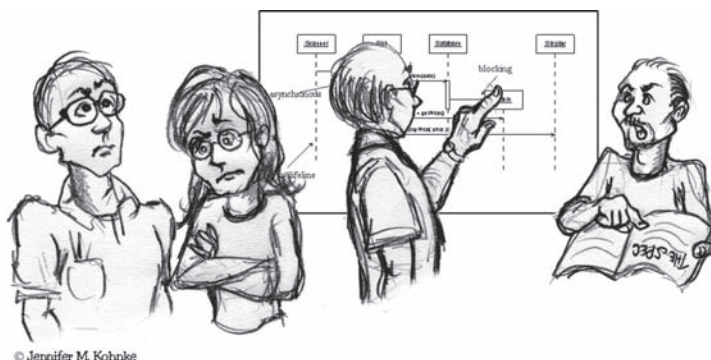
Framework	452
Zespół z roku 1994	452
Termin	452
Strategia	452
Wyniki	453
Projekt frameworka	454
Wspólne wymagania dla aplikacji oceniających	454
Projekt frameworka do wyznaczania ocen	456
Przypadek zastosowania wzorca Metoda szablonowa	459
Napisać pętlę raz	460
Wspólne wymagania dla aplikacji zdawania	463
Projekt frameworka do zdawania	463
Architektura menedżera zadań	469
Wniosek	472
Bibliografia	472
<b>Dodatek A Notacja UML. Część I: Przykład CGI</b>	<b>473</b>
System rejestrowania kursów: opis problemu	474
Aktorzy	475
Przypadki użycia	475
Model dziedziny	478
Architektura	482
Klasy abstrakcyjne i interfejsy na diagramach sekwencji	492
Podsumowanie	494
Bibliografia	494
<b>Dodatek B Notacja UML. Część II: STATMUX</b>	<b>495</b>
Definicja statystycznego multipleksera	495
Środowisko oprogramowania	496
Ograniczenia czasu rzeczywistego	496
Procedury obsługi przerwania wejścia	497
Procedury obsługi przerwania wyjścia	501
Protokoły komunikacji	502
Wniosek	512
Bibliografia	512
<b>Dodatek C Satyra na dwa przedsiębiorstwa</b>	<b>513</b>
Rufus! Inc. Project Kickoff	513
Rupert Industries Projekt Alpha	513
<b>Dodatek D Kod źródłowy jest projektem</b>	<b>525</b>
Czym jest projekt oprogramowania?	525
<b>Skorowidz</b>	<b>535</b>

---

# Lista wzorców projektowych

ACYKLICZNY WIZYTATOR	403
ADAPTER	333
AKTYWNY OBIEKT	169
DEKORATOR	413
FABRYKA	281
FASADA	187
KOMPOZYT	307
MEDIATOR	188
MENEDŻER ZADAŃ	469
METODA SZABLONOWA	176
MONOSTATE	194
MOST	338
OBIEKT NULL	201
OBIEKT ROZSZERZENIA	418
OBSERWATOR	327
PEŁNOMOCNIK	341
POLECENIE	165
SCHODY DO NIEBA	359
SERWER ABSTRAKCYJNY	332
SINGLETON	192
STAN	433
STRATEGIA	181
WIZYTATOR	400

# Co to jest projekt agile?



*Po zapoznaniu się z cyklem rozwoju oprogramowania doszedłem do wniosku, że jedyną dokumentacją oprogramowania, która rzeczywiście wydaje się spełniać kryteria projektu technicznego, są listingi z kodem źródłowym*

— Jack Reeves

W 1992 roku Jack Reeves napisał słynny artykuł w magazynie „C++ Journal” zatytułowany *What is Software Design?*<sup>2</sup>. W tym artykule Reeves twierdzi, że projekt systemu oprogramowania jest udokumentowany głównie za pomocą jego kodu źródłowego. Diagramy reprezentujące kod źródłowy są uzupełniające w stosunku do projektu i same w sobie nie są projektem. Jak się okazało, artykuł Jacka stał się zwiastunem rozwoju agile.

Na kolejnych stronach tej książki często będziemy mówili o projekcie. Nie należy zakładać, że oznacza to zestaw diagramów UML oddzielonych od kodu. Zbiór diagramów UML może reprezentować część projektu, ale to nie jest projekt. Projekt oprogramowania jest pojęciem abstrakcyjnym. Wiąże się zarówno z ogólnym kształtem i strukturą programu, jak również ze szczegółowym kształtem i strukturą każdego modułu, klasy i metody. Może być reprezentowany za pomocą wielu różnych mediów, ale jego ostateczną postacią jest kod źródłowy. Ostatecznie projektem jest kod źródłowy.

## Co złego dzieje się z oprogramowaniem?

Jeśli ktoś ma szczęście, to rozpoczyna projekt, mając jasny obraz tego, jaką postacią ma system przyjąć. Projekt systemu jest żywym obrazem w naszym umyśle. Jeśli ktoś ma jeszcze więcej szczęścia, to klarowność towarzyszy projektowi do pierwszej wersji dystrybucyjnej.

<sup>2</sup> [Reeves 92] To doskonały artykuł. Gorąco zachęcam do jego przeczytania. Dołączyłem go do tej książki jako dodatek D.

Wtedy coś zaczyna iść nie tak. Oprogramowanie zaczyna się psuć jak kawałek starego mięsa. W miarę upływu czasu zepsucie rozprzestrzenia się i rośnie. W kodzie gromadzą się brzydkie, ropiejące wrzody i czyraki, sprawiając, że jego utrzymanie staje się coraz trudniejsze.

Ostatecznie wysiłek, jaki trzeba włożyć, aby wprowadzić nawet najprostsze zmiany, staje się tak uciążliwy, że deweloperzy i menedżerowie zaczynają nalegać na zaprojektowanie systemu od początku.

Takie przedsięwzięcia rzadko kończą się sukcesem. Chociaż projektanci mają dobre intencje, to okazuje się, że strzelają do ruchomego celu. Stary system ewoluuje i zmienia się, a odzwierciedlenie tych zmian musi znaleźć miejsce w nowym projekcie. Brodawki i wrzody gromadzą się w nowym projekcie, zanim powstanie jego pierwsza wersja dystrybucyjna.

## Zapachy projektu — woń psującego się oprogramowania

Psujące się oprogramowanie można rozpoznać po dowolnym z następujących zapachów:

1. **Sztywność** — system jest trudny do zmiany, ponieważ każda zmiana wymusza wiele innych zmian w innych częściach systemu.
2. **Kruchość** — zmiany powodują psucie się systemu w miejscach, które na pozór nie mają związku z tą częścią, która została zmieniona.
3. **Brak mobilności** — trudno rozdzielić system na komponenty, które mogą być ponownie użyte w innych systemach.
4. **Lepkość** — stosowanie dobrych praktyk jest trudniejsze niż stosowanie złych praktyk.
5. **Zbędna złożoność** — projekt zawiera infrastrukturę, z której nie ma bezpośredniego pożytku.
6. **Zbędne powtórzenia** — projekt zawiera powtarzające się struktury, które mogłyby być ujednoczone w ramach pojedynczej abstrakcji.
7. **Nieczytelność** — system jest trudny do czytania i zrozumienia. Kod nie komunikuje dobrze swojej intencji.

**Sztywność.** Sztywność to cecha oprogramowania polegająca na trudnym wprowadzaniu nawet najprostszych zmian. Projekt jest sztywny, jeśli pojedyncza zmiana powoduje kaskadę kolejnych zmian w modułach zależnych. Im więcej modułów, które muszą być zmienione, tym bardziej sztywny projekt.

Większość programistów wcześniej czy później spotkała się z taką sytuacją. Ktoś prosi nas o wprowadzenie zmiany, która wydaje się być prosta. Zapoznajemy się z problemem i dokonujemy rozsądnego oszacowania nakładów wymaganej pracy. Ale później, podczas pracy nad wprowadzeniem zmiany, okazuje się, że istnieją reperkusje zmiany, których nie przewidzieliśmy. Musimy przebijać się przez ogromne porcje kodu i modyfikować znacznie więcej modułów, niż początkowo ocenialiśmy. Ostatecznie zmiany zajmują znacznie więcej czasu, niż szacowaliśmy. Na pytanie, dlaczego oszacowanie było takie niedokładne, powtarzamy tradycyjny lament programisty: „To było o wiele bardziej skomplikowane, niż sądziłem!”.

**Kruchość.** Kruchość jest tendencją oprogramowania do psucia się w wielu miejscach po wprowadzeniu pojedynczej zmiany. Często nowe problemy pojawiają się w obszarach, które nie mają koncepcyjnego związku z obszarem, który został zmieniony. Naprawienie tych problemów prowadzi do jeszcze większej liczby problemów, a zespół deweloperski zaczyna przypominać psa, który goni własny ogon.

W miarę wzrostu kruchości modułu prawdopodobieństwo, że wprowadzenie zmiany spowoduje nieoczekiwane problemy, jest bliskie pewności. Wydaje się to absurdalne, ale takie moduły wcale nie należą do rzadkości. Są to moduły, które stale wymagają naprawy. Nigdy nie znikają z listy błędów. Programiści wiedzą, że trzeba je zaprojektować od nowa (ale nikt nie chce się zmierzyć z zadaniem ich przebudowy). Im więcej poprawek w nich wprowadzamy, tym stają się *gorsze*.

**Brak mobilności.** Projekt jest niemobilny, gdy zawiera części, które mogłyby być użyteczne w innych systemach, ale wysiłek i ryzyko związane z oddzieleniem tych części od pierwotnego systemu są zbyt duże. Jest to sytuacja niefortunna, ale niestety bardzo częsta.



**Lepkość.** Lepkość może mieć dwie formy: lepkość oprogramowania oraz lepkość środowiska.

W obliczu konieczności wprowadzenia zmiany programiści zwykle wymyślają więcej niż jeden sposób na wprowadzenie tej zmiany. Niektóre ze sposobów zachowują projekt, inne nie (są to tzw. *hacki* — ang. *hacks*). Gdy sposoby zachowujące projekt są trudniejsze do wprowadzenia niż *hacki*, to lepkość projektu jest wysoka. Łatwo wprowadza się zmiany w niewłaściwy sposób, natomiast trudno zrobić to właściwie. Należy dążyć do zaprojektowania oprogramowania w taki sposób, aby wprowadzanie zmian, które zachowują projekt, było łatwe do wykonania.

Lepkość środowiska powstaje w sytuacji, gdy środowisko programistyczne jest powolne i niewydajne. Na przykład jeśli czasy kompilacji są bardzo długie, deweloperzy mają pokusę, aby dokonywać zmian, które nie wymuszają dużych rekompilacji, choć zmiany te nie zachowują projektu. Jeśli pobranie zaledwie kilku plików z systemu kontroli kodu źródłowego wymaga godzin, to deweloperzy będą się starać wprowadzać zmiany, które wymagają jak najmniej pobrań z systemu kontroli wersji, bez względu na to, czy projekt jest zachowany, czy nie.

W obu przypadkach lepki projekt to taki, w którym trudno utrzymać strukturę oprogramowania. Należy stworzyć systemy i środowiska projektowe, które ułatwiają zachowywanie projektu.

**Zbytняя złożoność.** Projekt zawiera niepotrzebną złożoność, gdy zawiera elementy, które w danej chwili nie są użyteczne. To często się dzieje, gdy deweloperzy przewidują zmiany w wymaganiach i umieszczają w oprogramowaniu mechanizmy mające na celu obsługę tych potencjalnych zmian. Początkowo może się wydawać, że to jest dobre. W końcu przygotowywanie się do przyszłych zmian powinno utrzymać elastyczność kodu i zapobiec kosztownym zmianom w późniejszym okresie.

Niestety, efekt często jest dokładnie odwrotny. Poprzez przygotowywanie się do zbyt wielu zmian projekt jest zaśmiecony konstrukcjami, które nigdy nie są używane. Niektóre spośród tych przygotowań mogą się opłacać, ale wiele innych nie. W międzyczasie projekt znosi ciężar tych niewykorzystanych elementów konstrukcyjnych. W ten sposób oprogramowanie staje się skomplikowane i trudne do zrozumienia.

**Niepotrzebne powtórzenia.** Operacje wtylnij i wklej mogą być przydatne do edycji tekstu, ale w przypadku edycji kodu ich stosowanie może mieć katastrofalne skutki. Bardzo często systemy oprogramowania są budowane na dziesiątkach lub setkach powtarzających się elementów kodu. Dochodzi do tego w następujący sposób:

Rafał musi napisać kod, który realizuje funkcjonalność A. Przegląda inne części kodu, w których podejrzewa występowanie podobnych operacji, i znajduje odpowiedni fragment kodu. Wycina ten kod i wkleja do swojego modułu, a następnie wprowadza odpowiednie modyfikacje.

Rafał nie wie, że kod, który wyciął za pomocą myszy, został tam wprowadzony przez Tadek, który wyciął go z modułu napisanego przez Lidę. Lidia była pierwszą osobą, która zrealizowała funkcjonalność A, ale zdała sobie sprawę, że ta funkcjonalność była bardzo podobna do funkcjonalności B. Znalazła implementację tej funkcjonalności, wycięła i wkleiła ją do swojego modułu, a następnie wprowadziła niezbędne modyfikacje.

Kiedy ten sam kod pojawia się w kółko w wielu miejscach, choć w nieco innej formie, deweloperom brakuje abstrakcji. Wyszukiwanie wszystkich powtórzeń i eliminowanie ich z wykorzystaniem odpowiedniej abstrakcji może nie znajdować się zbyt wysoko na ich liście priorytetów, ale wykonanie tej czynności znacząco przyczynia się do tworzenia systemu, który jest łatwiejszy do zrozumienia i utrzymania.

Gdy w systemie jest zbędny kod, zadanie modyfikowania systemu może stać się uciążliwe. Błędy znalezione w takiej powtarzającej się jednostce muszą być poprawiane we wszystkich powtórzeniach. Ponieważ jednak każde powtórzenie nieco różni się od pozostałych, poprawka nie zawsze jest taka sama.

**Nieczytelność.** Nieczytelność jest cechą modułu polegającą na trudności w jego zrozumieniu. Kod może być napisany w jasny i czytelny sposób lub może być napisany w sposób nieprzejrzysty i zawiły. Kod, który ewoluuje, z biegiem czasu wykazuje tendencję do stawania się coraz bardziej nieprzezroczystym. Utrzymanie nieprzezroczystości na minimalnym poziomie wymaga stałego wysiłku. Trzeba ciągle dbać o to, by kod był czytelny i ekspresywny.

Kiedy deweloperzy piszą moduł po raz pierwszy, kod może im się wydawać jasny. To dlatego, że wglębili się w kod i doskonale go rozumieją. Po jakimś czasie, gdy wracają do tego modułu, często zastanawiają się, jak mogli napisać coś tak okropnego. Aby temu zapobiec, deweloperzy powinni starać się stawiać w pozycji czytelników swojego kodu i podejmować wysiłki zmierzające do refaktoryzacji kodu w taki sposób, aby czytelnicy kodu mogli go zrozumieć. Kod powinien być także przeglądany przez inne osoby.

## Co stymuluje oprogramowanie do psucia się?

W środowiskach, które nie stosują metodyki agile, projekty pogarszają się z powodu zmian wymagań w sposób, którego pierwotny projekt nie przewidywał. Często zmiany te muszą być wykonane szybko i mogą być wykonywane przez deweloperów, którzy nie znają pierwotnej filozofii projektu. Tak więc choć zmiana w projekcie działa, to w jakiś sposób narusza oryginalny projekt. Kawałek po kawałku, w miarę wprowadzania kolejnych zmian, naruszenia te kumulują się i projekt zaczyna „pachnieć”.

Nie możemy jednak winić dryfowania wymagań za degradację projektu. Deweloperzy powinni doskonale zdawać sobie sprawę, że wymagania się zmieniają. Rzeczywiście, większość z nas zdaje sobie sprawę, że wymagania są najbardziej lotnymi elementami projektu. Jeżeli projekty nie udają się ze względu na ciągle deszcz zmieniających się wymagań, to winę za to ponoszą stosowane przez nas praktyki projektowe. Trzeba znaleźć jakiś sposób, aby projekty były odporne na zmiany, i zastosować praktyki, które zabezpieczają kod przed psuciem się.

## Zespoły agile nie pozwalają psuć się oprogramowaniu

Zespół agile kwitnie w obliczu zmian. Zespół niewiele inwestuje w początkowe prace, dlatego nie jest przywiązany do wyjściowego projektu. Zamiast tego utrzymuje konstrukcję systemu w sposób maksymalnie czysty i prosty i uzasadnia to za pomocą wielu testów jednostkowych i testów akceptacyjnych. Dzięki temu projekt jest elastyczny, a wprowadzanie w nim zmian jest łatwe. Zespół korzysta z tej elastyczności w celu ciągłego poprawiania projektu. Dzięki temu każda iteracja kończy się powstaniem systemu, którego projekt jest jak najlepszy w stosunku do wymagań w tej iteracji.

## Program Copy

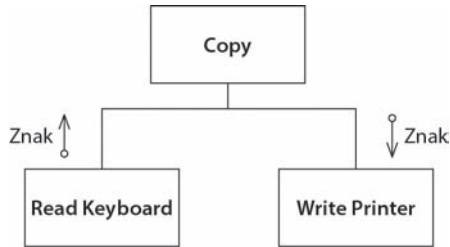
Aby zilustrować powyższe punkty, warto przyjrzeć się procesowi psucia się oprogramowania. Powiedzmy, że szef przyszedł do Ciebie w poniedziałek rano i poprosił o napisanie programu, który kopiuje znaki z klawiatury na drukarkę. Po wykonaniu w myślach kilku szybkich ćwiczeń umysłowych doszedłeś do wniosku, że taki program nie powinien zająć więcej niż dziesięć linii kodu. Projektowanie i kodowanie nie powinny zająć więcej niż godzinę. Razem ze spotkaniami grup interdyscyplinarnych, szkoleniami jakości, codziennymi spotkaniami grup badających postępy prac i trzema bieżącymi kryzysami w branży zakończenie tego programu powinno zająć około tygodnia — jeśli zostaniesz po godzinach. Zawsze jednak nasze szacunki mnożymy przez trzy.

„Trzy tygodnie” — takiej odpowiedzi udzielasz szefowi. Szef mruczy coś pod nosem i zgadza się, pozostawiając Cię z zadaniem.

**Wstępny projekt.** Masz teraz trochę czasu, zanim rozpocznie się spotkanie poświęcone przeglądowi procesu, więc decydujesz, że narysujesz projekt programu. Przy zastosowaniu projektu strukturalnego doszedłeś do diagramu struktury pokazanego na rysunku 7.1.

Aplikacja składa się z trzech modułów lub podprogramów. Moduł Copy wywołuje dwa pozostałe. Program kopiujący pobiera znaki z modułu Read Keyboard i kieruje je do modułu Write Printer.

Patrzysz na swój projekt i oceniasz, że jest dobry. Uśmiechasz się i wychodzisz z biura, by pójść na spotkanie. Przynajmniej będzie można tam trochę pospać.



Rysunek 7.1. Diagram struktury programu Copy

We wtorek przyszedłeś nieco wcześniej, dzięki czemu możesz zakończyć pracę nad programem Copy. Niestety, jeden z kryzysów w branży dał o sobie znać w nocy. W związku z tym musisz iść do laboratorium i pomóc w debugowaniu problemu. W przerwie na obiad, na którą w końcu wyszedłeś o 15.00, udało Ci się napisać kod programu Copy. Efekt pokazano na listingu 7.1.

#### Listing 7.1. Program Copy

```
void Copy()
{
    int c;
    while ((c=RdKbd()) != EOF)
        WrtPrt(c);
}
```

Ledwo udało Ci się zapisać edytowany kod, kiedy zdałeś sobie sprawę, że jesteś już spóźniony na spotkanie jakości. Wiesz, że to spotkanie jest ważne. Będzie poświęcone znaczeniu produkcji bezusterkowej. Zatem chowasz swoje krakersy i colę i udajesz się na spotkanie.

W środę znów przyszedłeś wcześniej. Tym razem masz nadzieję, że nic nie stanie Ci na przeszkodzie. Otwierasz kod źródłowy programu Copy i zaczynasz go kompilować. I oto program kompiluje się za pierwszym razem — bez błędów! Dobrze i to, bo Twój szef woła Cię na nieplanowane spotkanie na temat konieczności oszczędzania tonerów drukarek laserowych.

W czwartek, po czterech godzinach spędzonych na rozmowie przez telefon z serwisantem w Rocky Mount w Karolinie Północnej, udzieleniu mu zdalnej pomocy w debugowaniu i rejestrowaniu błędów w jednym z bardziej nieczytelnych elementów systemu, otrzymałeś podziękowania i zaczynasz testować program Copy. Działa! Za pierwszym razem! To dobrze, bo Twój nowy uczeń właśnie usunął główny katalog z kodem źródłowym z serwera. Musisz więc znaleźć taśmy z najnowszą kopią zapasową i odtworzyć go. Oczywiście ostatnia pełna kopia zapasowa została zrobiona trzy miesiące temu. Dlatego musisz odtworzyć jeszcze dziewięćdziesiąt cztery przyrostowe kopie zapasowe.

Piątek wydaje się zupełnie wolny od dodatkowych zajęć. To dobrze, ponieważ cały dzień zajmuje Ci pomyślnie załadowanie programu Copy do systemu kontroli kodu źródłowego.

Oczywiście program odnosi niezwykle sukces i jest wdrażany w całej firmie. Twoja reputacja jako programistycznego asa ponownie została potwierdzona. Możesz teraz wygrzać się w chwale swoich osiągnięć. Przy odrobinie szczęścia może faktycznie uda Ci się napisać w tym roku trzydzieści linii kodu!

**Wymagania. One się zmieniają!** Kilka miesięcy później szef przychodzi do Ciebie i mówi, że chciałby, aby program Copy mógł czytać z czytnika taśmy papierowej. Zgrzytasz zębami i przewracasz oczami. Zastanawiasz się, dlaczego ludzie zawsze zmieniają wymagania. Twój program nie był projektowany z myślą o obsłudze czytnika taśmy papierowej! Ostrzegasz swojego szefa, że wprowadzenie tego rodzaju zmian może naruszyć elegancję projektu. Niemniej jednak szef jest nieugięty. Mówi, że użytkownicy naprawdę potrzebują od czasu do czasu czytać znaki z czytnika taśmy papierowej.

Zatem wdychasz i planujesz wprowadzenie modyfikacji. Chcesz dodać do funkcji Copy argument typu Boolean. Jeśli będzie miał wartość true, będziesz czytać z czytnika taśmy papierowej; jeśli będzie miał wartość false, będziesz czytać z klawiatury, tak jak przedtem. Niestety, teraz już tak wiele innych

programów korzysta z programu Copy, że nie można zmieniać interfejsu. Zmiana interfejsu spowodowałaby, że ponowna kompilacja zajęłaby całe tygodnie. Sami inżynierowie testów zlincolnaliby Cię, nie wspominając nawet o siedmiu gościach z grupy zarządzania konfiguracją. Trzeba by również poświęcić wiele czasu na przeglądanie kodu każdego modułu, który wywołuje program Copy.

Nie. Zmiana interfejsu nie wchodzi w rachubę. Ale w takim razie w jaki sposób poinformować program Copy o tym, że powinien czytać z czytnika taśmy papierowej? Oczywiście użyjesz zmiennej globalnej. Użyjesz także najlepszej i najwartościowszej funkcji grupy języków C-podobnych — operatora ?: . Efekt pokazano na listingu 7.2.

### Listing 7.2. Pierwsza modyfikacja programu Copy

```
bool ptFlag = false;
// pamiętaj, żeby zresetować tę flagę
void Copy()
{
    int c;
    while ((c=(ptflag ? RdPt() : RdKbd())) != EOF)
        WrtPrt(c);
}
```

Programy wywołujące funkcję Copy, które chcą czytać z czytnika taśmy papierowej, muszą najpierw ustawić flagę ptFlag na true. Później mogą wywołać funkcję Copy, a ta będzie szczęśliwie czytać z czytnika taśmy papierowej. Kiedy funkcja Copy zwróci sterowanie, proces wywołujący musi zresetować flagę ptFlag. Gdyby tego nie zrobił, to następny wywołujący mógłby przez pomyłkę czytać z czytnika taśmy papierowej zamiast z klawiatury. Aby przypomnieć programistom o obowiązku zresetowania tej flagi, dodałeś odpowiedni komentarz.

Po raz kolejny publikujesz wersję dystrybucyjną. Odnosi nawet większy sukces niż wcześniej, a hordy chętnych programistów czekają na okazję, aby z niej skorzystać. Życie jest piękne.

**Dać im palec...** Kilka tygodni później Twój szef (który nadal jest Twoim szefem pomimo trzech reorganizacji w całej korporacji na przestrzeni wielu miesięcy) przychodzi do Ciebie i mówi, że klienci chcieliby, aby program Copy mógł pisać na dziurkarce taśmy papierowej.

Ach ci klienci! Zawsze rujną nasze projekty. *Pisanie oprogramowania byłoby dużo łatwiejsze, gdyby nie to, że istnieją klienci.*

Mówisz szefowi, że te niestanne zmiany mają głęboko negatywny wpływ na elegancję Twojego projektu. Ostrzegasz go, że jeśli zmiany w dalszym ciągu będą wprowadzane w tym strasznym tempie, to oprogramowanie do końca roku będzie niemożliwe do utrzymania. Twój szef kiwa głową ze zrozumieniem, a potem mówi, aby i tak dokonać zmian.

Ta zmiana w projekcie jest podobna do poprzedniej. Wystarczy kolejna zmienna globalna i kolejny operator ?: . Efekt wysiłków pokazano na listingu 7.3.

### Listing 7.3. Druga modyfikacja programu Copy

```
bool ptFlag = false;
bool punchFlag = false;
// pamiętaj, żeby zresetować te flagi
void Copy()
{
    int c;
    while ((c=(ptflag ? RdPt() : RdKbd())) != EOF)
        punchFlag ? WrtPunch(c) : WrtPrt(c);
}
```

Szczególnie dumny jesteś z faktu, że pamiętałeś o tym, aby zmienić komentarz. Mimo to martwisz się, że struktura programu zaczyna się przewracać. Każda dodatkowa zmiana urzędzenia wejściowego z pewnością zmusi Cię do całkowitej przebudowy pętli while. Być może nadszedł czas, aby odkurzyć swoje CV...

**Oczekiwanie zmian.** Ocenie czytelnika pozostawiam ustalenie, jak wiele z powyższego opisu było satyryczną przesadą. Sednem tej historii było pokazanie, że w obliczu zmian projekt programu może ulec szybkiej degradacji. Wyjściowy projekt programu Copy był prosty i elegancki. Wystarczyły jednak dwie zmiany, aby zaczął wykazywać oznaki **szttywności, kruchości, braku mobilności, złożoności, redundancji i nieczytelności**. Ten trend z pewnością będzie trwał, a program przekształci się w śmietnik.

Możemy usiąść i zrzucić winę za ten stan rzeczy na zmiany. Możemy narzekać, że program został dobrze zaprojektowany zgodnie z oryginalną specyfikacją, a późniejsze zmiany w specyfikacji spowodowały degradację projektu. Jednak takie podejście ignoruje jeden z najbardziej znanych faktów w produkcji oprogramowania: *wymagania zawsze się zmieniają!*

Należy pamiętać, że najbardziej ulotną rzeczą w większości projektów wytwarzania oprogramowania są wymagania. Wymagania przez cały czas są w stanie płynnym. To jest fakt, który my deweloperzy musimy zaakceptować! *Żyjemy w świecie zmieniających się wymagań, a naszym zadaniem jest zapewnienie, aby nasze oprogramowanie przetrwało te zmiany.* Jeżeli projekt naszego oprogramowania degradowuje się, ponieważ zmieniły się wymagania, to nie jesteśmy agile.

## Przykład programu Copy wykonanego zgodnie z metodyką agile

Produkcja agile programu Copy mogła rozpocząć się dokładnie tak jak wcześniej — od kodu z listingu 7.1<sup>3</sup>. Kiedy szef poprosił deweloperów agile, aby przystosowali program do czytania z czytnika taśmy papierowej, mogliby zareagować zmodyfikowaniem projektu w taki sposób, aby był odporny na tego rodzaju zmiany. Efekt mógłby wyglądać podobnie do kodu z listingu 7.4.

**Listing 7.4.** Wersja numer 2 programu Copy wykonanego zgodnie z metodyką agile

```
class Reader
{
    public:
        virtual int read() = 0;
};

class KeyboardReader : public Reader
{
    public:
        virtual int read() {return Rdkbd ();}
};

KeyboardReader GdefaultReader;

void Copy(Reader& reader = GdefaultReader)
{
    int c;
    while ((c=reader.read()) != EOF)
        WrtPrt(c);
}
```

Zamiast próbować łątać projekt, aby sprostać nowemu wymaganiu, zespół wykorzystał okazję do poprawienia projektu w taki sposób, aby w przyszłości był odporny na tego rodzaju zmiany. Od tej pory, gdy szef poprosi o obsługę nowego rodzaju urządzenia wejściowego, zespół będzie w stanie zareagować w sposób, który nie powoduje degradacji programu Copy.

Zespół zastosował zasadę otwarte-zamknięte (ang. *Open-Closed Principle* — OCP), o której można przeczytać w rozdziale 9. Według tej zasady moduły powinny być projektowane w taki sposób, by można je było rozszerzać bez modyfikowania. To jest dokładnie to, co zrobił zespół. Można dostarczyć dowolne nowe urządzenie wejściowe, o które szef poprosi, bez modyfikowania programu Copy.

<sup>3</sup> W rzeczywistości jest bardzo prawdopodobne, że stosowanie praktyki produkcji sterowanej zmusiłoby do stworzenia projektu na tyle elastycznego, że udałoby mu się przetrwać bez zmian żądania szefa. Jednak w tym przykładzie zignorujemy to.

Zwróćmy jednak uwagę, że zespół nie próbował przewidywać, jak program będzie się zmieniał, gdy projektował moduł po raz pierwszy. Zamiast tego program został napisany w najprostszy możliwy sposób. Dopiero gdy zmieniły się wymagania, zespół zmienił projekt modułu w taki sposób, aby był odporny na tego rodzaju zmiany.

Można by się spierać, że członkowie zespołu wykonali tylko połowę pracy. Gdy zabezpieczali się przed różnymi urządzeniami wejściowymi, mogli również zabezpieczyć się przed różnymi urządzeniami wyjściowymi. Jednak zespół w rzeczywistości nie miał pojęcia, czy urządzenia wyjściowe kiedykolwiek się zmienią. Wprowadzenie dodatkowej ochrony w tym momencie byłoby pracą, która nie służyłaby żadnemu bieżącemu celowi. Jest oczywiste, że jeśli będzie potrzebne takie zabezpieczenie, łatwo będzie je można dodać później. W związku z tym w rzeczywistości nie ma powodu, aby dodawać je teraz.

## Skąd deweloperzy agile wiedzieli, co należy zrobić?

Deweloperzy agile w przykładzie zamieszczonym powyżej stworzyli klasę abstrakcyjną, aby zabezpieczyć się przed zmianami urządzenia wejściowego. Skąd wiedzieli, jak należy to zrobić? Jest to związane z jednym z podstawowych założeń projektu obiektowego.

Początkowy projekt programu Copy jest nieelastyczny ze względu na *kierunek* jego zależności. Spójrzmy ponownie na rysunek 7.1. Zwróćmy uwagę, że moduł Copy zależy bezpośrednio od modułów Read Keyboard oraz Write Printer. W tej aplikacji Copy jest modułem wysokiego poziomu. Ten moduł określa strategię aplikacji. „Wie”, jak kopiować znaki. Niestety, został również uzależniony od niskopoziomowych szczegółów klawiatury i drukarki. Z tego powodu zmiana niskopoziomowych szczegółów będzie miała wpływ na wysokopoziomową strategię.

Kiedy ta nieelastyczność uwidoczniła się, deweloperzy agile wiedzieli, że zależność modułu Copy od urządzenia wejściowego będzie trzeba *odwrócić*<sup>4</sup> tak, aby moduł Copy przestał zależeć od urządzenia wejściowego. Wtedy zastosowano wzorzec Strategia<sup>5</sup> w celu utworzenia żądanej inwersji.

A zatem w skrócie — deweloperzy agile wiedzieli, co należy zrobić, ponieważ:

1. Wykryli problem, ponieważ postępowali zgodnie z praktykami agile.
2. Zdiagnozowali problem poprzez zastosowanie zasad projektowania.
3. Rozwiązali problem poprzez zastosowanie odpowiedniego wzorca projektowego.

Współdziałanie tych trzech aspektów wytwarzania oprogramowania *jest* aktem projektowania.

## Utrzymywanie projektu w jak najlepszej postaci

Deweloperzy agile starają się utrzymywać projekt w taki sposób, aby był jak najwłaściwszy i jak najczystszy. To zobowiązanie nie jest ani przypadkowe, ani niepewne. Deweloperzy agile nie „porządkują” projektu co kilka tygodni. Zamiast tego starają się utrzymywać oprogramowanie w maksymalnie czystej i ekspresywnej postaci w każdym dniu, o każdej godzinie i nawet w każdej minucie. Nigdy nie mówią „wrócimy do tego i naprawimy później”. Nigdy nie pozwalają na to, aby oprogramowanie zaczęło się psuć.

Postawa, jaką deweloperzy agile stosują w odniesieniu do projektu oprogramowania, jest taka sama jak postawa, którą stosują chirurdzy w odniesieniu do zachowania sterylności. To właśnie zachowanie procedur sterylności sprawia, że chirurgia jest *możliwa*. Bez niej ryzyko infekcji byłoby zbyt wysokie. Deweloperzy agile stosują takie samo podejście do swoich projektów. Ryzyko pozwolenia sobie na nawet najmniejsze objawy zepsucia jest zbyt wysokie, aby można je było tolerować.

Projekt musi być utrzymywany w czystości, a ponieważ kod źródłowy jest najważniejszym wyrazem projektu, on także musi pozostać czysty. Profesjonalizm podpowiada, że programiści nie mogą tolerować psucia się kodu.

<sup>4</sup> Zobacz zasadę odwracania zależności (*Dependency Inversion Principle* — DIP) w rozdziale 11.

<sup>5</sup> Wzorzec Strategia omówimy bardziej szczegółowo w rozdziale 14.



## Wniosek

A zatem co to jest projekt agile? Projekt agile jest procesem, a nie zdarzeniem. Polega na ciągłym stosowaniu zasad, wzorców i praktyk w celu poprawy struktury i czytelności oprogramowania. To zaangażowanie w ciągle utrzymywanie projektu systemu w maksymalnie prostej, czystej i ekspresywnej postaci.

W następnych rozdziałach będziemy omawiali zasady i wzorce projektowania oprogramowania. Podczas lektury należy pamiętać, że deweloper agile nie stosuje tych zasad i wzorców do wysokopoziomowego projektu „z góry”. Zamiast tego praktyki te są stosowane od iteracji do iteracji, jako próba utrzymania kodu i projektu, który on obejmuje, w jak najczystszej postaci.

## Bibliografia

1. Jack Reeves, *What Is Software Design?*, „C++ Journal”, wolumin 2, nr 2. 1992. Dostępny pod adresem <http://www.bleading-edge.com/Publications/C++Journal/Cpjour2.htm>.



---

# Skorowidz

## A

abstrakcja, 118, 124, 145, 214  
abstrakcja harmonogramu, 214  
abstrakcje stabilne, 276  
abstrakcyjna fabryka, 388  
abstrakcyjność, 297  
abstrakcyjny interfejs, 381  
acykliczne zależności, 267  
Acykliczny wizytator, 403, 404  
Adapter, 333, 337  
ADP, Acyclic Dependencies Principle, 267  
ADP, Adaptive Software Development, 39  
agile, 23  
Agile Alliance, 24  
agregacje, 486  
aktorzy, 475  
aktualizacja obiektu, 313  
Aktywny obiekt, 169  
algorytm termostatu, 146  
algorytmy HiLo, 384  
analiza, 206  
analiza przypadku  
  stacja pogodowa, 367  
API, 375  
API utrwalania, 382  
aplikacja  
  koszyk na zakupy, 345  
  multipleksera statystycznego, 495  
  obsługi kursów, 489  
  Payroll, 48  
  placowa, 295, 298  
  Shape, 119  
  stacji pogodowej, 382  
  zdawania, 463  
architektura, 482  
  menedżera zadań, 469, 471  
  programów CGI, 493  
  systemu pomiaru, 370  
  webowa, 482  
artefakt, 362  
asocjacje, 486  
atrybuty, 486  
automat stanów skończonych, 427, 428

## B

baza danych, 167, 219, 257  
błąd w strukturze transakcji, 207, 209  
błędy transakcji, 211  
BOM, Bill of Material, 418

brak mobilności, immobility, 101  
brzydkie zapachy kodu, 101  
budowanie systemów wielowątkowych, 172

## C

CCP, Common-Closure Principle, 266  
CGI, 473  
charakterystyki zależności, 487  
ciąg główny, 277  
ciągła integracja, 34  
ciśnienie atmosferyczne, 371  
cykle, 270  
cykle zależności, 268, 283

## D

DAG, Directed Acyclic Graph, 269  
DBC, design by contract, 132  
decyzje, 505  
decyzje biznesowe, 246  
Dekorator, 365, 413, 414  
delegacja, 175, 323  
delegowanie obserwatora, 323  
diagram  
  aktywności, 503, 505  
  czujnika temperatury, 375  
  klas, 212, 378  
  kolaboracji, 509  
  komponentów systemu, 484  
  obiektów, 507  
  pakietów, 269, 290, 382  
  przypadków użycia, 477  
  sekwencji, 371, 490, 492  
  stanów, 498  
diagramy  
  STD, 428  
  UML, 217, 230  
DIP, Dependency Inversion Principle, 141  
DM, Dependency Management, 263  
dodawanie  
  do modelu obiektowego, 342  
  do modelu relacyjnego, 343  
  pracowników, 218  
dokumenty planowania  
  oprogramowania, 369  
domknięcie, 124, 125  
dostęp do bazy danych, 345  
dynamiczna  
  struktura abstrakcji, 216  
  struktura programu głównego, 302

dynamiczny model  
  abstrakcji, 215  
  programu, 257  
  scenariusza, 245  
  transakcji, 221, 228, 233, 235, 239  
  wzorca, 345  
działanie wzorca Pełnomocnik, 344  
dziedziczenie, 129, 137, 175, 222, 321  
  wielokrotne, 153  
  wirtualne, 362

## E

elastyczność, 484  
eliminowanie  
  cykli zależności, 268  
  sprzężeń, 48, 372  
ETS, Educational Testing Service, 397, 449  
ewolucja kodu do wzorca, 311

## F

Fabryka, 161, 281, 287, 388  
fabryki  
  obiektów, 301  
  wymienne, 284, 285  
Fasada, Facade, 161, 187, 366  
fasada bazy danych, 366  
firma Chmura, 367  
framework  
  do zdawania, 463  
ETS  
  informacje o projekcie, 449  
  projekt, 454  
  strategia, 452  
  wyniki, 453  
  wyznaczania ocen, 456  
FSM, finite state machine, 427  
funkcja  
  DoEval, 458  
  GeneratePrimes, 55  
  testMove, 46  
funkcje zdegenerowane, 140

## G

generator  
  menu sesji, 490  
  raportów, 407

generowanie liczb pierwszych, 54  
 gra  
   w kręgle, 66–98  
   w planowanie, 35  
 graf  
   acykliczny, 268  
   zależności, 269  
 granice systemu, 478  
 grupowanie klientów, 158  
 GUI, 439

## H

hermetyzacja, 294  
 heurystyki, 139  
 hierarchia  
   cech testu, 455  
   klas, 135, 208  
   klas Modem, 400  
   pakietów, 292  
   typu Modem, 338  
   wizytatora, 403  
 historia 24-godzinna, 383, 392  
 historyjki użytkowników, 32

## I

imitowanie, spoofing, 285  
   fabryki, 286  
   klasy, 285  
 implementacja  
   algorytmów HiLo, 384  
   klasy Turnstile, 441  
   wzorca, 192, 195  
   wzorca Pełnomocnik, 345, 387  
   wzorca Polecenie, 224  
 inicjalizacja interfejsu, 157  
 inicjowanie fabryk, 302  
 instrukcje switch-case, 429, 432  
 interakcja z GUI, 440  
 interfejs, 149, 150  
   API, 341, 357, 375  
   bazy danych, 489  
   GUI, 439  
   klasy ModemImplementation, 115  
   Modem, 334  
   rozdzielony użytkownika, 155  
   StationToolkit, 376  
   TimedDoor, 152  
   TimerClient, 150  
   TimeSink, 320  
   Transaction, 217  
   TurnstileLockedState, 434  
   TurnstileState, 434  
   użytkownika, 153, 372  
   WeatherStation, 381  
   znacznikowy, marker interface, 418  
 interfejsy  
   klas, 152  
   obiektów, 152

interpretacja tabeli przejść, 432  
 ISP, Interface Segregation Principle, 149  
 iteracje przypadków użycia, 482  
 izolacja testu, 47

## J

jawne domknięcie, 124  
 język  
   PDL, 531  
   Smalltalk, 133

## K

kanoniczna postać modelu, 327  
 klasa  
   ActiveObjectEngine, 170  
   AddEmployeeTransaction, 221, 222  
   AddSalariedEmployee, 222, 223  
   AlarmClock, 374, 378  
   Application, 177  
   ApplicationRunner, 182  
   Assembly, 408, 422  
   BubbleSorter, 179–183  
   ChangeAffiliationTransaction, 241  
   ChangeClassificationTransaction, 237  
   ChangeEmployeeTransaction, 234  
   ChangeHourlyTransaction, 238  
   ChangeMemberTransaction, 242  
   ChangeNameTransaction, 235  
   ChangeUnaffiliatedTransaction, 243  
   Clock, 312  
   ClockDriverTest, 313, 317, 318  
   CompositeCommand, 309  
   CompositeShape, 307, 308  
   CSVPiecePartExtension, 424  
   DB, 347, 352, 355  
   DBTest, 352, 355  
   DedicatedHayesModem, 338  
   DelayedTyper, 172  
   DeleteEmployeeTransaction, 224  
   DoubleBubbleSorter, 181  
   Employee, 255  
   Evaluator, 457  
   FSMError, 446  
   ftocStrategy, 182  
   HayesModem, 415  
   HiLoDataImp, 384  
   HourlyClassification, 250  
   HTMLTemplate, 485  
   IntBubbleSorter, 180  
   IntSortHandle, 184  
   ItemData, 355  
   LinearObject, 138, 139  
   LoudDialModem, 414, 415, 418  
   NullEmployee, 202  
   MockTimeSink, 315, 317, 325  
   MockTimeSource, 314, 319–325  
   ModemDecorator, 417

ModemDecoratorTest, 416  
 MonthlySchedule, 247  
 ObservableClock, 321  
 ObserverTest, 324  
 OrderData, 351  
 OrderImp, 352  
 OrderProxy, 354  
 Part, 421  
 PartCountVisitor, 410  
 PaymentSchedule, 252  
 PayrollDatabase, 219  
 PersistentAssembly, 364  
 PersistentObject, 360, 362  
 PersistentProduct, 363  
 PiecePart, 409, 422  
 PrimeGenerator, 56–61  
 Product, 349  
 ProductData, 346  
 ProductImp, 349  
 productPersistenceTestCase, 360  
 ProductProxy, 349  
 ProxyTest, 348, 351  
 QuickBubbleSorter, 185  
 QuickEntryMediator, 188, 190  
 SalesReceiptTransaction, 296  
 Scheduler, 371–373  
 ServiceChargeTransaction, 230, 231  
 SessionMenuGenerator, 490  
 SleepCommand, 171  
 SMCTurnstileTest, 446  
 SortHandle, 184  
 StationToolkit, 376  
 Subject, 324  
 TemperatureHiLo, 383  
 TemperatureSensor, 377  
 TestBOMReport, 410  
 TestBOMXML, 419  
 TestGeneratePrimes, 55  
 TestModemVisitor, 402, 406  
 TestSleepCommand, 170  
 TimeCard, 226  
 TimeCardTransaction, 225, 227, 296  
 TimeSource, 316, 320, 321  
 TimeSourceImplementation, 322  
 Turnstile, 197–200, 441–443  
 TurnstileFSM, 437  
 UserDatabaseSource, 194  
 WeatherStation, 377–380  
 WeeklySchedule, 251  
 XMLAssemblyExtension, 424  
 XMLPiecePartException, 423  
 klasy  
   abstrakcyjne, 134, 492  
   aplikacji płacowej, 298, 304  
   testowe, 370  
 klient, 32, 150  
 kod źródłowy, 525  
 kompilator  
   maszyny stanów, 436  
   SMC, 438, 443, 468

komponenty systemu rejestracji, 483  
 kompozycje, 486  
 Kompozyt, Composite, 307  
 komunikacja, 502  
 konfiguracja pakietów, 274  
 konwencje, 139  
 konwersja stopni, 176  
 krótkie cykle, 32  
 kruchość, fragility, 101

## L

lepkość, viscosity, 101  
 liczności, 481  
 LSP, Liskov Substitution Principle, 127

## M

macierz  
 dodawania, 456, 458  
 funkcji, 403  
 rzadka, 407  
 manifest Agile Alliance, 24  
 maszyna  
 FSM, 428, 438  
 stanów, 436, 439, 469  
 rysowanie prostokąta, 440  
 Mediator, 188  
 menedżer zadań, 469, 471  
 metafora, 37  
 metoda  
 CalculateDeductions(), 253  
 metoda  
 configureForUnix, 400  
 Execute, 224, 247  
 Undo, 169  
 Metoda szablonowa, Template method,  
 119, 161, 175–181, 220, 459, 462  
 metodologia  
 Adaptive Software Development, 29  
 Crystal, 29  
 Feature Driven Development, 29  
 SCRUM, 29  
 metryki, 296  
 D, 279  
 stabilności, 273  
 zarządzania zależnościami, 280  
 mierzenie abstrakcji, 276  
 model  
 dodawania pracownika, 221  
 dostarczania karty, 226  
 dziedziny, 478  
 obiektowy, 342  
 procesora zdarzeń, 465  
 programu głównego, 257  
 transakcji  
 AddEmployeeTransaction, 218  
 DeleteEmployee, 223  
 SalesReceiptTransaction, 228

wzorca Pełnomocnik, 345  
 zależności klas, 144  
 modyfikacja  
 klasy, 296  
 klasy Scheduler, 373  
 monady, 158  
 Monostate, 194  
 Most, Bridge, 338  
 multiplexer, 495

## N

naruszanie  
 stabilności, 275  
 zasady DIP, 282  
 zasady LSP, 128, 129, 136  
 zasady OCP, 120, 130  
 zasady SRP, 116  
 zasady SDP, 275  
 narzędzia klas Scope, 390  
 niepotrzebne powielenia, 101  
 nieprawidłowy format transakcji, 210  
 nieprzezroczystość, opacity, 101  
 niestabilność, 297  
 nieznanym identyfikator, 208  
 notacja UML  
 agregacje, 486  
 aktorzy, 475  
 asocjacje, 486  
 atrybuty, 486  
 decyzje, 505  
 diagramy aktywności, 503, 505  
 diagramy kolaboracji, 510  
 diagramy obiektów, 507  
 diagramy stanów, 498  
 kompozycje, 486  
 liczności, 481  
 model dziedziny, 478  
 notacja lizaka, 508  
 obiekty aktywne, 508  
 operacje, 486  
 pakiety, 488  
 podsystemy, 488  
 przejścia pomiędzy stanami, 500  
 przejścia wewnętrzne, 499  
 przejścia złożone, 505  
 przykład CGI, 473  
 przypadki użycia, 475  
 schemat blokowy, 496  
 semantyka klas, 479  
 stany, 499  
 stany zagnieżdżone, 500  
 STATMUX, 495  
 wykresy sekwencji komunikatów, 512

## O

Obiekt Null, 201  
 Obiekt rozszerzenia, Extension Object,  
 365, 418

obiekty aktywne, 508  
 obliczanie wynagrodzenia, 213, 214  
 Obserwator, Observer, 311, 326, 327  
 obserwatory ciśnienia atmosferycznego,  
 373  
 obsługa  
 baz danych, 357  
 GUI, 439  
 kolejności, 124  
 przerwań wejścia, 497  
 przerwań wyjścia, 501  
 wyjątków, 178  
 zdarzenia, 467  
 zdarzenia standardowego, 466  
 ochrona prywatności klasy, 296  
 OCP, Open-Closed Principle, 117  
 odchylenia, 270  
 oddziaływania klientów, 151  
 odległość od ciągu głównego, 278, 297  
 znormalizowana, 298  
 odpowiedzialność, 114, 115  
 pojedyncza, 113  
 sprzężona, 115  
 odpytywanie obiektów, 159  
 odwracanie  
 własności, 142  
 zależności, 141, 147, 271, 281, 359  
 ograniczenia czasu rzeczywistego, 496  
 okresowe pomiary, 370  
 okresy rozliczeniowe, 251  
 opakowanie obiektów, 157  
 operacje, 486  
 oprogramowanie, 525, 532  
 oprogramowanie middleware, 357, 359  
 oprogramowanie Nimbus-LC, 391  
 oprogramowanie protokołów  
 komunikacyjnych, 506  
 oprogramowanie WMS-LC, 369  
 otwarta przestrzeń robocza, 35  
 ozdobniki, adornments, 294

## P

pakiet  
 Classifications, 295  
 TransactionImplementation, 301  
 pakiety, 263, 488  
 nieodpowiedzialne, irresponsible, 291  
 odpowiedzialne, responsible, 291  
 z metrykami, 300  
 Pełnomocnik, 342, 356, 387  
 pełnomocnik dla relacji, 350  
 pętle, 460  
 plan publikacji wersji dystrybucyjnych, 394  
 planowanie, 39  
 iteracji, 41  
 wersji dystrybucyjnych, 41  
 zadań, 41  
 podstawianie typów, 127  
 podsystemy, 488

podział  
 klas na pakiety, 379  
 kodu  
   czasowy, 168  
   fizyczny, 168  
   na warstwy, 142  
 polecenia, Command, 161  
   kompozytowe, 308  
   proste, 166  
   sterowane przez czujnik, 166  
 Polecenie, 165, 224  
 poliady, 158  
 polimorfizm statyczny, 147  
 poprawność modelu, 132  
 porządkowanie typów, 125  
 potwierdzenie pakietu, 511  
 praktyki agile, 23  
 problem  
   modemu, 334–339  
   projektowy, 251  
   dotyczący tworzenia, 376  
 procedura obsługi przerwań  
   wejścia, 497  
   wyjścia, 501  
 proces inicjalizacji, 509  
 procesor zdarzeń, 464, 469  
 program  
   Furnace, 146  
   główny, 257, 302  
   testowy, 343  
 programowanie, 65  
   ekstremalne, 29, 31  
   parami, 33  
   sterowane testami, 34, 45  
   zabezpieczanie przed zmianami, 123  
 projekt  
   agile, 101  
   Alpha, 513  
   dzielnice na pakiety, 261  
   frameworka, 454, 463  
   góra-dół, 271  
   Kickoff, 513  
   modelu zdarzeń, 465  
   oprogramowania, 525  
   według kontraktu, DBC, 132  
 projektowanie  
   pakietów, 263  
   według kontraktu, 132  
 prosty projekt, 36  
 protokoły komunikacyjne, 502, 506  
 przebieg iteracji, 42  
 przejścia  
   pomiędzy stanami, 500  
   złożone, 505  
 przerywanie  
   cykli, 270  
   zależności, 315  
 przetwarzanie  
   rozproszone, 441  
   zdarzenia, 466, 467

przypadek testowy, 226, 240, 246–254  
 przypadek użycia, 206, 475  
   dodawanie pracownika, 162, 207  
   dostarczenie informacji, 164, 210  
   dostarczenie karty pracy, 163, 209  
   dostarczenie raportu sprzedaży, 163, 209  
   systemu Nimbus-LC, 393  
   usuwanie pracownika, 163, 208  
   generowanie listy płac, 164, 212  
   zmiana danych pracownika, 164, 210  
 pusty obiekt, Null object, 161

## R

realizacja wypłat, 244, 246, 248  
 redundancja, 150  
 refaktoryzacja, 37, 53  
 rejestrowanie kursów, 474  
 relacja IS-A, 132  
 relacje, 476  
 relacyjny model danych, 342  
 rozdzielone  
   odpowiedzialności, 114  
   interfejs, 155  
 rozszerzanie klasy, 332  
 rozwiązanie problemu modemu, 335–339  
 równomierne tempo, 35  
 RTC, run-to-completion, 172

## S

scenariusze, 245  
 schemat  
   blokowy systemu, 496  
   granic systemu, 478  
 Schody do nieba, Stairway to heaven, 359  
 segregacja interfejsów, 149  
 semantyka klas UML, 479  
 separacja  
   przez delegację, 152  
   przez wielokrotne dziedziczenie, 153  
   utrwalania od strategii, 386  
 Serwer abstrakcyjny, 332  
 silnik przejść, 432  
 Singleton 161, 192, 193  
 skrypt depend.sh, 297  
 SMC, 436  
 sortowanie, 183  
 sortowanie bąbelkowe, 179  
 specyfikacja systemu płacowego, 162  
 specyfikowanie kontraktów, 133  
 spójność, cohesion, 113, 302  
   pakietów, 266  
   projektu, 130, 131  
   relacyjna, 297  
 sprżężenia, 47, 294, 374  
   przychodzące, 297  
   wychodzące, 297  
 sprężony podsystem utrwalania, 116

SRP, Single Responsibility Principle, 113  
 stabilne  
   abstrakcje, 276  
   zależności, 272  
 stabilność, 272  
 stacja pogodowa, 305, 367  
 Stan, 427, 433, 436  
 stany zagnieżdżone, 500  
 STATMUX, 495  
 statyczna struktura klasy, 219, 225  
 statyczny  
   model programu, 257, 492  
   model transakcji, 218, 228–232, 244  
   multiplexer, 495  
 stereotypy na listach, 498  
 stosowanie  
   klas-pełnomocników, 359  
   wzorca Monostate, 196  
   wzorca Singleton, 193  
 Strategia, Strategy, 119, 161, 175, 181,  
 386, 435, 462  
 strategia obsługi GUI, 439  
 struktura  
   czujnika, 374  
   danych planu piętra, 459  
   generatora raportów, 407  
   klas, 462  
   klasy BubbleSorter, 180  
   klasy TemperatureHiLo, 383  
   klasy TimeCardTransaction, 225  
   pakietów, 290, 380, 389  
   pakietów aplikacji płacowej, 303  
   programu głównego, 302  
   wzorca, 462  
   zależności, 273  
 studium przypadku  
   ETS, 397  
   stacja pogodowa, 305  
   system płacowy, 161, 205, 217  
 symptomy złego projektu, 101  
 system Nimbus-LC, 369  
 system płacowy, 161, 258, 289  
   implementacja, 217  
   pierwsza iteracja, 205  
   podział 8na pakiety, 261  
 system rejestrowania kursów, 474  
 system Statmux, 496  
 systemy wielowątkowe, 172  
 szablon HTML, 484, 485  
 sztywność, rigidity, 101

## T

tabela przejść, 432, 433, 441  
 TDD, test-driven development, 34  
 techniki implementacji, 429  
 test jednostkowy, 55, 133, 284, 430  
 testowanie, 45  
   akcji, 432  
   obiektu DigitalClock, 312  
 testy akceptacyjne, 33, 49, 51



- transakcja, 167
    - AddEmployee, 168
    - AddSalariedTransaction, 219
    - ChangeAffiliationTransaction, 240
    - ChangeClassificationTransaction, 235
    - ChangeCommissionedTransaction, 236
    - ChangeEmployeeTransaction, 232, 233
    - ChangeMethodTransaction, 239
    - PaydayTransaction, 244
    - SalariedTransaction, 236
    - SalesReceiptTransaction, 228
    - ServiceChargeTransaction, 229
    - TimeCardTransaction, 226
  - tworzenie
    - fabryki, 389
    - macierzy dla dodawania, 458
    - obiektów trwałych, 286
    - pełnomocników, 350
    - prototypów, 40
    - zestawów testowych, 284
- ## U, W
- uniwersalny regulator, 147
  - usuwanie pracowników, 223
  - utrwalanie, 382, 386
  - warstwa, 142
    - adaptera klasy, 134
    - middleware, 358
    - pośrednia, 358
  - wątek
    - czasowy, 504
    - odbierający, 504
    - wysyłający, 503
  - wersje dystrybucyjne systemu, 394
  - wewnętrzne przejścia, 499
  - wiele dekoratorów, 416
  - wielokrotnie
    - dziedziczenie, 321
    - wykorzystanie kodu, 264, 292
    - wykorzystywanie maszyny stanów, 468
  - wielokrotność, 309
  - wielokrotny zbiorowy użytek, 265
  - Wizytator, Visitor, 365, 399–401, 426
  - właściciel interfejsu, 333
  - wskaźnik
    - A, 297
    - Ca, 297
    - Ce, 297
    - D, 297
    - D', 298
    - H, 297
    - I, 297
  - wskaźniki
    - dla pakietów, 300, 305
    - globalne, 157
  - wspólna własność, 34
  - wybór
    - języka, 369
    - platformy oprogramowania, 482
  - wydziałanie, 137
  - wyjątek FSMError, 446
  - wyjątki, 140
  - wykorzystanie
    - diagramów, 327
    - wzorca Fabryka, 284
  - wykres
    - czasowy, 279
    - punktowy, 279
  - wykresy sekwencji komunikatów, 512
  - wymagania
    - administracyjne, 392
    - dla aplikacji oceniających, 454
    - dla aplikacji zdawania, 463
    - dla Nimbus-LC, 391
    - użytkowe, 391
  - wyszukiwanie abstrakcji, 145, 214
  - wyścig, 511
  - wyścig ACK, 512
  - wyświetlanie parametrów pogody, 370
  - wyznaczanie ocen, 456
  - wzorze projektowe, 161, 165
  - wzorzec
    - Acykliczny wizytator, 403, 404
    - Adapter, 333, 334, 337
    - Aktywny obiekt, 169
    - Dekorator, 365, 413, 414
    - Fabryka, 281, 287
    - Fasada, 187, 366
    - Kompozyt, 307
    - Mediator, 188
    - Metoda szablonowa, 119, 161, 175–181, 220, 459, 462
    - Monostate, 194
    - Most, 338
    - Obiekt Null, 201
    - Obiekt rozszerzenia, 365, 418, 419
    - Obserwator, 311, 326, 327
    - Pełnomocnik, 342, 356, 387
    - Polecenie, 165, 224
    - Schody do nieba, 359
    - Serwer abstrakcyjny, 332
    - Singleton, 192, 193
    - Stan, 427, 433, 436
    - Strategia, 119, 161, 175, 181, 386, 435, 462
    - Wizytator, 365, 399–401, 426
- ## Z
- zachowanie zgodności z OCP, 121, 123
  - zadanie
    - MeasureTask, 469
    - RTC, 172
  - zależności
    - acykliczne, 267
    - między pakietami, 269
    - modułów, 141
    - niewłaściwe, 275
    - między pakietami, 267
    - stabilne, 272
    - zależność od abstrakcji, 143
  - zarządzanie
    - kolejnością, 124
    - zależnościami, DM, 263
    - zasadami projektu, 328
  - zasada, 27, 101
    - ADP, acyklicznych zależności, 267
    - CCP, zbiorowego zamykania, 266, 291
    - CRP, zbiorowego wielokrotnego użytku, 265
    - DIP, odwracania zależności, 141, 281, 332
    - ISP, segregacji interfejsów, 149, 151
    - LSP, podstawiania Liskov, 127, 334
    - OCP, otwarte-zamknięte, 117, 328
    - REP, 264, 292
    - SAP, stabilnych abstrakcji, 276
    - SDP, stabilnych zależności, 272
    - SRP, pojedynczej odpowiedzialności, 113, 333
  - zasady
    - projektowania pakietów, 263
    - spójności pakietów, 264
    - sprzęgania pakietów, 267
  - zasięg zmiennej stanu, 431
  - zastosowanie
    - delegacji, 323
    - wskaźników, 298
    - wzorca Fabryka, 282
    - wzorca Kompozyt, 308
    - wzorca Metoda szablonowa, 178, 220, 459
    - wzorca Obserwator, 372
    - wzorca Schody do nieba, 360
    - wzorca Strategia, 181
    - wzorca Wizytator, 412
    - zasady CCP, 291
    - zasady REP, 292
  - zbiorowe domykanie, 266, 291
  - zbyttna złożoność, 101
  - zdarzenie
    - Erase, 464
    - Measure, 466
    - ScreenCursor, 464
    - Sketch, 465
    - specyficzne dla winiety, 467
    - thirdBadPassword, 439
  - zegar cyfrowy, 311
  - zestawienia materiałów, 418
  - zgodność z zasadą LSP, 137
  - zgodność z zasadą OCP, 121
  - ziarnistość, 264
  - złożoność, 150
  - zmiana
    - danych pracowników, 231
    - klasyfikacji, 235
  - zmiennie globalne, 225
  - zwinne wytwarzanie oprogramowania, 29



# 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**

# ZWINNE WYTWARZANIE OPROGRAMOWANIA

## Najlepsze zasady, wzorce i praktyki

Czasy kaskadowego tworzenia projektów odchodzą w niepamięć. Obecne tempo rozwoju aplikacji i rynku nie pozwala poświęcać miesięcy na analizę, tworzenie dokumentacji, projektowanie, a na końcu wytwarzanie, testowanie i wdrażanie. Produkt musi być dostępny błyskawicznie! Pozwala to na natychmiastowe zebranie opinii na jego temat, dostosowanie go do oczekiwań i szybkie reagowanie na wymagane zmiany. Takie założenia może spełnić tylko zespół wytwarzający oprogramowanie w zwinny sposób!

Ta książka została w całości poświęcona zwinnym praktykom. Sięgnij po nią i przekonaj się, w jaki sposób planować kolejne iteracje, tworzyć kod, a następnie go testować. W trakcie lektury poznasz praktyczne aspekty zwinnego tworzenia kodu — zobaczysz, jak istotne są zasady: pojedynczej odpowiedzialności, podstawienia Liskov czy odwracania zależności. Znajdziesz tu także zasady projektowania pakietów oraz przydatne wzorce projektowe, omówione na konkretnych przykładach. Książka ta jest obowiązkową lekturą dla wszystkich osób zaangażowanych w wytwarzanie oprogramowania i chcących udoskonalić swój proces.

### Dzięki tej książce:

- poznasz fundamenty zwinnego wytwarzania oprogramowania
- zaznajomisz się z najlepszymi praktykami
- dowiesz się, jak refaktoryzować kod
- wybierzesz właściwe wzorce i unikniesz typowych błędów
- poprawisz swój proces wytwarzania oprogramowania

**Zwinne programowanie = większa szansa na sukces projektu!**

**Helion**

27248 numer katalogowy

księgarnia internetowa



<http://helion.pl>

zamówienia telefoniczne



0 801 339900



0 601 339900

Sprawdź najnowsze promocje:

🔗 <http://helion.pl/promocje>

Książki najchętniej czytane:

🔗 <http://helion.pl/bestsellery>

Zamów informacje o nowościach:

🔗 <http://helion.pl/nowosci>

Helion SA  
ul. Kościuszki 1c, 44-100 Gliwice  
tel.: 32 230 98 63  
e-mail: [helion@helion.pl](mailto:helion@helion.pl)  
<http://helion.pl>

sięgnij po **WIĘCEJ**



KOD KORZYŚCI

ISBN 978-83-246-9682-6



9 788324 696826

Informatyka w najlepszym wydaniu

cena: 89,00 zł