

Sprawdzone rozwiązania  
Twoich problemów!

  
ADDISON-WESLEY

# WZORCE IMPLEMENTACYJNE

KENT BECK



Tytuł oryginału: Implementation Patterns

Tłumaczenie: Piotr Rajca

Projekt okładki: Maciej Pasek

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

ISBN: 978-83-246-8644-5

Authorized translation from the English language edition, entitled: IMPLEMENTATION PATTERNS; ISBN 0321413091; by Kent Beck; published by Pearson Education, Inc, publishing as Addison Wesley. Copyright © 2008 by Pearson Education.

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 © 2014.

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.

Wydawnictwo HELION dołożyło wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie bierze jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Wydawnictwo HELION nie ponosi 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/wzoimp>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

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

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

# Spis treści

<b>Wstęp</b> .....	<b>11</b>
Podziękowania .....	12
<b>Rozdział 1. Wprowadzenie</b> .....	<b>13</b>
Przewodnik .....	15
A teraz... ..	16
<b>Rozdział 2. Wzorce</b> .....	<b>17</b>
<b>Rozdział 3. Teoria programowania</b> .....	<b>21</b>
Wartości .....	22
Komunikatywność .....	22
Prostota .....	23
Elastyczność .....	24
Zasady .....	25
Lokalne konsekwencje .....	26
Minimalizacja powtórzeń .....	26
Połączenie logiki i danych .....	27
Symetria .....	27
Przekaz deklaracyjny .....	28
Tempo zmian .....	29
Wnioski .....	30
<b>Rozdział 4. Motywacja</b> .....	<b>31</b>
<b>Rozdział 5. Klasy</b> .....	<b>33</b>
Klasa .....	34
Prosta nazwa klasy bazowej .....	35
Kwalifikowana nazwa klasy pochodnej .....	36
Interfejs abstrakcyjny .....	37
Interfejs .....	38
Klasa abstrakcyjna .....	39

Interfejs wersjonowany .....	40
Obiekt wartościowy .....	41
Specjalizacja .....	43
Klasa pochodna .....	44
Implementator .....	46
Klasa wewnętrzna .....	47
Zachowanie zależne od instancji .....	48
Konstrukcja warunkowa .....	48
Delegacja .....	50
Selektor dołączany .....	52
Anonimowa klasa wewnętrzna .....	53
Klasa biblioteczna .....	53
Wniosek .....	54
<b>Rozdział 6. Stan .....</b>	<b>55</b>
Stan .....	56
Dostęp .....	57
Dostęp bezpośredni .....	58
Dostęp pośredni .....	59
Wspólny stan .....	60
Stan zmienny .....	60
Stan zewnętrzny .....	62
Zmienna .....	62
Zmienna lokalna .....	63
Pole .....	65
Parametr .....	66
Parametr zbierający .....	67
Parametr opcjonalny .....	68
Zmienna lista argumentów .....	68
Obiekt parametrów .....	69
Stałe .....	70
Nazwa sugerująca znaczenie .....	71
Zadeklarowany typ .....	72
Inicjalizacja .....	73
Inicjalizacja wczesna .....	73
Inicjalizacja leniwa .....	74
Wniosek .....	75
<b>Rozdział 7. Zachowanie .....</b>	<b>77</b>
Przepływ sterowania .....	78
Przepływ główny .....	78
Komunikat .....	79
Komunikat wybierający .....	80
Dwukrotne przydzielanie .....	80
Komunikat dekomponujący (sekwencjonujący) .....	81
Komunikat odwracający .....	82
Komunikat zapraszający .....	83

Komunikat wyjaśniający .....	83
Przepływ wyjątkowy .....	84
Klauzula strażnika .....	84
Wyjątek .....	86
Wyjątki sprawdzane .....	87
Propagacja wyjątków .....	87
Wniosek .....	88
<b>Rozdział 8. Metody .....</b>	<b>89</b>
Metoda złożona .....	92
Nazwa określająca przeznaczenie .....	93
Widoczność metody .....	94
Obiekt metody .....	96
Metoda przesłonięta .....	98
Metoda przeciążona .....	98
Typ wynikowy metody .....	99
Komentarz do metody .....	100
Metoda pomocnicza .....	100
Metoda komunikatu informacyjnego .....	101
Konwersja .....	102
Metoda konwertująca .....	102
Konstruktor konwertujący .....	103
Utworzenie .....	103
Kompletny konstruktor .....	104
Metoda wytwórcza .....	105
Fabryka wewnętrzna .....	106
Metoda dostępu do kolekcji .....	106
Metoda określająca wartości logiczne .....	108
Metoda zapytania .....	108
Metoda równości .....	109
Metoda pobierająca .....	110
Metoda ustawiająca .....	111
Bezpieczna kopia .....	112
Wniosek .....	113
<b>Rozdział 9. Kolekcje .....</b>	<b>115</b>
Metafory .....	116
Zagadnienia .....	117
Interfejsy .....	119
Tablice (klasa Array) .....	120
Interfejs Iterable .....	120
Interfejs Collection — kolekcje .....	121
Interfejs List — listy .....	121
Interfejs Set — zbiory .....	121
Interfejs SortedSet — zbiory posortowane .....	122
Interfejs Map — mapy .....	123
Implementacje .....	123

Implementacje interfejsu Collection .....	124
Implementacje interfejsu List .....	125
Implementacje interfejsu Set .....	125
Implementacje interfejsu Map .....	126
Klasa Collections .....	128
Wyszukiwanie .....	128
Sortowanie .....	128
Kolekcje niezienne .....	129
Kolekcje jednoelementowe .....	129
Kolekcje puste .....	129
Rozszerzanie kolekcji .....	130
Wniosek .....	131
<b>Rozdział 10. Rozwijanie platform .....</b>	<b>133</b>
Modyfikowanie platform bez zmian w aplikacjach .....	133
Niezgodne aktualizacje .....	134
Zachęcanie do wprowadzania zgodnych zmian .....	136
Klasa biblioteczna .....	137
Obiekty .....	137
Wnioski .....	146
<b>Dodatek A. Pomiary wydajności .....</b>	<b>149</b>
Przykład .....	150
API .....	150
Implementacja .....	151
Klasa MethodTimer .....	152
Eliminacja narzutów czasowych .....	154
Testy .....	154
Porównywanie kolekcji .....	155
Porównywanie kolekcji ArrayList i LinkedList .....	157
Porównywanie zbiorów .....	158
Porównywanie map .....	159
Wnioski .....	160
<b>Bibliografia .....</b>	<b>163</b>
Ogólne zagadnienia programistyczne .....	163
Filozofia .....	165
Java .....	166
<b>Spis szablonów .....</b>	<b>167</b>
<b>Skorowidz .....</b>	<b>169</b>

## Rozdział 7

---

# Zachowanie

John von Neumann wprowadził jedną z najważniejszych metafor programowania komputerów — sekwencję instrukcji wykonywanych kolejno jedna po drugiej. To właśnie dzięki niej mogły powstać języki programowania, w tym także Java. Ten rozdział jest poświęcony sposobom przekazywania informacji o zachowaniu programów. Zostały w nim opisane następujące wzorce:

- Przepływ sterowania (ang. *Control Flow*) — wyraża obliczenia jako sekwencję czynności.
- Przepływ główny (ang. *Main Flow*) — precyzyjnie wyraża główny przepływ sterowania.
- Komunikat (ang. *Message*) — wyraża przepływ sterowania w formie wysyłania komunikatów.
- Komunikat wybierający (ang. *Choosing Message*) — zmienia implementujące komunikaty w celu wyrażenia wyboru.
- Podwójne przydzielanie (ang. *Double Dispatch*) — zmienia elementy implementujące komunikat względem dwóch osi, wyrażając w ten sposób wybór kaskadowy.
- Komunikat dekomponujący (ang. *Decomposing Message*) — dzieli złożone obliczenia na spójne fragmenty.
- Komunikat odwracający (ang. *Reversing Message*) — zapewnia symetrię przepływu sterowania poprzez wysyłanie sekwencji komunikatów do tego samego odbiorcy.
- Komunikat zapraszający (ang. *Inviting Message*) — zachęca do wprowadzania przyszłych zmian poprzez wysyłanie komunikatu, który może być implementowany na różne sposoby.
- Komunikat wyjaśniający (ang. *Explaining Message*) — wysyła komunikat wyjaśniający przeznaczenie pewnego fragmentu logiki.

- Przepływ wyjątkowy (ang. *Exceptional Flow*) — jak najbardziej precyzyjnie wyraża wyjątkowy przepływ sterowania, bez zakłócania głównego przepływu sterowania.
- Klauzula strażnika (ang. *Guard Clause*) — wyraża lokalny przepływ wyjątkowy poprzez wcześniejsze zakończenie wywołania.
- Wyjątek (ang. *Exception*) — wyraża nielocalne przepływy wyjątkowe, używając w tym celu wyjątków.
- Wyjątek sprawdzany (ang. *Checked Exception*) — wymusza przechwytywanie wyjątków poprzez ich jawne deklarowanie.
- Propagacja wyjątków (ang. *Exception Propagation*) — przekazuje wyjątki, przekształcając je w razie konieczności tak, by dostępne w nich informacje były odpowiednie dla kodu, który będzie je obsługiwać.

## Przepływ sterowania

Dlaczego przepływ sterowania pojawia się we wszystkich programach? Istnieją języki programowania, takie jak Prolog, w których nie istnieje jawne pojęcie przepływu sterowania. W programach pisanych w takich językach wszystkie elementy logiki są wymieszane i czekają na zaistnienie odpowiednich warunków, by mogły się uaktywnić.

Java należy do rodziny języków programowania, w których sekwencja kontroli jest podstawową zasadą organizacji kodu. Sąsiadujące ze sobą instrukcje są wykonywane kolejno, jedna po drugiej. Warunki sprawiają, że dany kod będzie wykonywany tylko w określonych okolicznościach. Pętle pozwalają na cykliczne wykonywanie fragmentu kodu. Wysyłanie komunikatów powoduje aktywację jednej z kilku podprocedur. Wyjątki umożliwiają przeskakiwanie do kodu położonego w górze stosu.

Wszystkie te mechanizmy sumują się, tworząc wspólnie bogate medium służące do wyrażania sposobu działania obliczeń. Jako autorzy (programiści) decydujemy, czy przepływ sterowania, który sobie wyobrażamy, zostanie wyrażony jako jeden przepływ główny, wiele przepływów dodatkowych, z których każdy ma równie duże znaczenie, czy też jako pewna kombinacja obu powyższych rozwiązań. Grupujemy poszczególne elementy przepływu sterowania, tak by przeciętny odbiorca mógł je od razu zrozumieć na dosyć abstrakcyjnym poziomie, dodając jednocześnie więcej szczegółów dla tych, którzy będą chcieli je zrozumieć dokładniej. Grupowanie to czasami przybiera postać procedur umieszczanych w klasach, a czasami przekazywania sterowania do innych obiektów.

## Przepływ główny

Programiści zazwyczaj wiedzą, jak ma wyglądać główny przepływ sterowania w tworzonych programach. Przetwarzanie rozpoczyna się i kończy w precyzyjnie określonych miejscach. W jego trakcie mogą być podejmowane decyzje lub występować wyjątki,



niemniej jednak obliczenia mają ściśle wytyczoną ścieżkę, po której przebiega ich realizacja. Jest ona jasno wyrażana przy użyciu wybranego języka programowania.

W przypadku niektórych programów, zwłaszcza tych, które mają działać niezawodnie w niesprzyjających okolicznościach, przepływ główny nie jest widoczny. Jednak takie programy spotyka się raczej sporadycznie. Wykorzystanie dużych możliwości wyrazu używanego języka programowania w celu informowania o rzadko wykonywanych i modyfikowanych faktach dotyczących naszego programu sprawia, że mniej zauważalne stają się jego ważniejsze fragmenty: te, które będą często analizowane, rozumiane i zmieniane. Nie chodzi o to, że takie wyjątkowe warunki nie mają znaczenia, lecz o to, że dużo cenniejsze jest skoncentrowanie uwagi na precyzyjnym i jasnym wyrażeniu głównego przepływu obliczeń.

Dlatego też ten główny przepływ programu należy wyrażać przejrzysto, a wyjątków oraz klauzul strażników używać jedynie w sytuacjach niezwykłych oraz przy błędach.

## Komunikat

Jednym z podstawowych sposobów wyrażania logiki w języku Java są komunikaty. W proceduralnych językach programowania mechanizmem służącym do ukrywania informacji są wywołania procedur:

```
compute() {
    input();
    process();
    output();
}
```

Powyższa procedura przekazuje następujące informacje: „W celu zrozumienia tych obliczeń wystarczy wiedzieć, że składają się one z trzech kroków; aktualnie wszelkie pozostałe szczegóły nie mają znaczenia”. Jednym z najpiękniejszych aspektów programowania obiektowego jest to, że ta sama procedura wyraża także znacznie więcej. Dla każdej metody istnieje potencjalnie cały zbiór obliczeń o podobnej strukturze, różniących się jedynie szczegółami. A dodatkową korzyścią jest to, że tworząc niezmienny fragment obliczeń, nie trzeba zwracać uwagi na wszystkie szczegóły ewentualnych przyszłych odmian.

Wykorzystanie komunikatów jako głównego mechanizmu przepływu sterowania sprawia, że zmiany są podstawową cechą programów. Każdy komunikat jest potencjalnym miejscem, w którym można coś zmienić bez konieczności modyfikowania kodu będącego źródłem tego komunikatu. Procedury opierające się na idei komunikatów nie stwierdzają: „Tam coś się dzieje, ale szczegóły nie są na razie istotne”; zamiast tego komunikują: „W tym miejscu naszej historii z danymi wejściowymi dzieje się coś interesującego. Szczegóły tego, co się dzieje, mogą się jednak zmienić”. Rozważne korzystanie z tej elastyczności, jasne i bezpośrednie wyrażanie logiki zawsze, gdy to jest możliwe, i odpowiednie opóźnianie przedstawiania szczegółów to bardzo ważne umiejętności, przydatne, gdy zależy nam na pisaniu programów, które efektywnie wyrażają nasze intencje.

## Komunikat wybierający

Czasami wysyłam komunikat służący do wyboru implementacji, podobny do sposobu, w jaki w językach proceduralnych działa instrukcja wyboru switch. Na przykład aby wyświetlić jakiś element graficzny na jeden z kilku dostępnych sposobów i zaznaczyć, że wybór może zostać dokonany w trakcie działania programu, można skorzystać z komunikatu polimorficznego.

```
public void displayShape(Shape subject, Brush brush) {
    brush.display(subject);
}
```

Komunikat `display()` wybiera implementację na podstawie faktycznego typu, który w trakcie działania programu przyjmie parametr `brush`. Dzięki temu później będzie można zaimplementować różne typy parametrów `brush`: `ScreenBrush`, `PostscriptBrush` itd.

Swobodne korzystanie z komunikatów wybierających prowadzi do powstawania kodu, w którym występuje niewiele jawnych warunków. Stosowanie tych komunikatów jest zaproszeniem do późniejszego wprowadzania rozszerzeń. Każdy jawnie podany warunek jest punktem, w którym, w razie chęci zmiany działania całego programu, konieczne będzie wprowadzenie jawnych modyfikacji.

Analiza kodu, który w dużym stopniu korzysta z komunikatów wybierających, wymaga nabycia doświadczenia i umiejętności. Jedną z wad stosowania takich komunikatów jest to, że zrozumienie konkretnej ścieżki działania programu może wymagać przeanalizowania kilku klas. Twórcy kodu mogą jednak upraszczać życie innym, nadając metodom nazwy informujące o ich przeznaczeniu. Dodatkowo należy także zwracać uwagę na to, kiedy stosowanie komunikatów wybierających będzie przesadą. Jeśli w wykonywanych obliczeniach nie może się nic zmieniać, to nie należy tworzyć metody tylko po to, by zapewnić możliwość wprowadzenia ewentualnej zmiany.

## Dwukrotne przydzielanie

Komunikaty wybierające dobrze nadają się do wyrażania jednego wymiaru zmienności. W przykładzie podanym w poprzednim podrozdziale wymiarem tym był rodzaj medium, na którym był rysowany kształt. Jeśli jednak konieczne jest wyrażenie dwóch niezależnych wymiarów zmienności, to można to osiągnąć, tworząc kaskadę składającą się z dwóch komunikatów wybierających.

Założmy przykładowo, że należy w jakiś sposób wyrazić, iż postscriptowy owal jest obliczany inaczej niż prostokąt wyświetlany na ekranie. W pierwszej kolejności należałoby określić, gdzie mają się znaleźć obliczenia. Wydaje się, że podstawowe obliczenia należą do klasy `Brush`, dlatego też komunikat wybierający w pierwszej kolejności zostanie wysłany do klasy `Shape`, a dopiero potem do klasy `Brush`:

```
displayShape(Shape subject, Brush brush) {
    subject.displayWith(brush);
}
```

Wówczas każdy typ `Shape` ma możliwość zaimplementowania metody `displayWith()` w odmienny sposób. Jednak zamiast wykonywać jakieś szczegółowe operacje, dodają one jedynie do komunikatu swój typ i przekazują działanie do klasy `Brush`:

```
Oval.displayWith(Brush brush) {
    brush.displayOval(this);
}
Rectangle.displayWith(Brush brush) {
    brush.displayRectangle(this);
}
```

Dzięki temu do poszczególnych typów pochodnych klasy `Brush` przekazywane są informacje niezbędne do prawidłowego działania:

```
PostscriptBrush.displayRectangle(Rectangle subject) {
    writer.print(subject.left() + " "+...+ rect);
}
```

Takie rozwiązanie wprowadza pewne powtórzenia, którym towarzyszy utrata elastyczności. Nazwy typów, do których są przekazywane pierwsze komunikaty wybierające, zostają bowiem podane w kodzie metod obsługujących drugi komunikat wybierający. W powyższym przykładzie oznacza to, że aby dodać nowy kształt — klasę pochodną `Shape` — konieczne będzie dodanie odpowiednich metod do wszystkich klas pochodnych klasy `Brush`. Jeśli wiadomo, że prawdopodobieństwo zmienienia się jednego wymiaru jest większe niż drugiego, to ten pierwszy wymiar powinien się stać odbiorcą drugiego komunikatu.

Naukowiec, który czasami się we mnie odzywa, chciałby uogólnić to rozwiązanie i stworzyć wzorzec przydzielania trzykrotnego, czterokrotnego itd. Niemniej jednak udało mi się jedynie wypróbować rozwiązanie przydzielania trzykrotnego, lecz nawet ono nie zdało egzaminu na dłuższą metę. Zawsze znajdowałem bardziej przejrzyste i rozumiałem sposoby wyrażania logiki wielowymiarowej.

## Komunikat dekomponujący (sekwencjonujący)

Kiedy pisany program wykorzystuje złożony algorytm, na który składa się wiele kroków, to czasami można pogrupować kroki powiązane ze sobą i wykonywać je, wysyłając jeden komunikat. Taki komunikat jest tworzony w celu zapewnienia możliwości wprowadzania specjalizacji lub jakichkolwiek bardziej wyszukanych rozwiązań. Stanowi on odpowiednik staromodnej dekompozycji funkcjonalnej. Komunikat jest w tym przypadku stosowany tylko po to, by wywołać podsekwencję kroków umieszczoną w procedurze.

Komunikat dekomponujący musi nosić odpowiednią, opisową nazwę. Większość osób analizujących kod powinna uzyskać wszystkie informacje konieczne do określenia przeznaczenia procedury wyłącznie na podstawie jej nazwy. Konieczność przeanalizowania całego kodu wywoływanego przez ten komunikat powinna dotyczyć jedynie osób, które chcą poznać szczegóły implementacyjne.

Problemy z doбором nazwy komunikatu dekomponującego mogą sugerować, że użycie tego wzorca nie jest właściwym rozwiązaniem. Innym sygnałem może być długa lista parametrów. Widząc takie sygnały, zastępuję komunikat dekomponujący wywoływanymi w nim metodami i staram się zastosować inny wzorzec (taki jak obiekt metody), który będzie mi w stanie pomóc właściwie wyrazić strukturę programu.

## Komunikat odwracający

Symetria wpływa na poprawę czytelności kodu. Przeanalizujemy poniższą metodę:

```
void compute() {
    input();
    helper.process(this);
    output();
}
```

Choć składa się ona z wywołań trzech różnych metod, brakuje w niej symetrii. Jej czytelność można by poprawić, wprowadzając metodę pomocniczą, której użycie zapewniłoby symetrię. Analizując zmodyfikowaną wersję metody `compute()`, nie trzeba już śledzić, do kogo jest wysyłany komunikat — wszystkie komunikaty są bowiem kierowane do bieżącego obiektu (`this`).

```
void process(Helper helper) {
    helper.process(this);
}
void compute() {
    input();
    process(helper);
    output();
}
```

W takim przypadku osoba analizująca kod może zrozumieć strukturę metody `compute()`, przeglądając kod jednej klasy.

Czasami zdarza się, że dużego znaczenia nabiera sama metoda wywoływana przez komunikat odwracający. Może się także zdarzyć, że przesadne stosowanie komunikatów odwracających skutecznie ukryje potrzebę przeniesienia funkcjonalności. Gdyby w programie pojawił się następujący kod:

```
void input(Helper helper) {
    helper.input(this);
}
void output(Helper helper) {
    helper.output(this);
}
```

to najprawdopodobniej można by poprawić całą strukturę kodu, przenosząc metodę `compute()` do klasy `Helper`:

```
compute() {
    new Helper(this).compute();
}
```

```

Helper.compute() {
    input();
    process();
    output();
}

```

Osobiście czasami czuję się trochę głupio, tworząc metody „tylko” po to, by zaspokoić „estetyczne” dążenie do uzyskania symetrii. Jednak ta estetyka ma znacznie głębsze podłoże. Angażuje mózg w większym stopniu niż liniowy proces myślowy. Kiedy już wykształcimy w sobie dążenie do estetyki kodu, to wrażenia estetyczne, które kod będzie nam zapewniał, okażą się cenną informacją o jego jakości. Te odczucia wydostające się spod powierzchni symbolicznych myśli mogą być równie cenne, jak jawnie nazwane i uzasadnione wzorce.

## Komunikat zapraszający

Czasami, pisząc kod, można oczekiwać, że inni programiści będą chcieli zmodyfikować fragment obliczeń przez wprowadzenie klasy pochodnej. Możliwość wprowadzenia takich późniejszych usprawnień da się wyrazić poprzez wysłanie komunikatu o odpowiednio dobranej nazwie. Taki komunikat zaprasza innych programistów do usprawnienia obliczeń wedle własnych potrzeb.

Jeśli dostępna jest jakaś domyślna implementacja logiki, to należy użyć jej jako implementacji komunikatu. Jeśli jednak takiej domyślnej implementacji nie ma, to chcąc jawnie przekazać zaproszenie, wystarczy zadeklarować metodę jako abstrakcyjną.

## Komunikat wyjaśniający

W programowaniu zawsze duże znaczenie miało rozróżnienie intencji i implementacji. To właśnie dzięki niemu można najpierw zrozumieć wykonywane obliczenia, a później, w razie konieczności, poznać ich szczegóły. Rozróżnienie to można utworzyć, korzystając z komunikatów, wystarczy zacząć od wysłania komunikatu, którego nazwa określa rozwiązywany problem, a w obsługującym go kodzie wysłać kolejny komunikat o nazwie odpowiadającej sposobowi rozwiązania problemu.

Pierwszy raz spotkałem się z takim rozwiązaniem w języku Smalltalk. Konkretnie rzecz biorąc, moją uwagę zwróciła następująca metoda:

```

highlight(Rectangle area) {
    reverse(area);
}

```

Zastanawiałem się: „Dlaczego takie rozwiązanie jest przydatne? Dlaczego nie wywołać metody `reverse()` bezpośrednio, tylko przy użyciu dodatkowej metody `highlight()`?”. Jednak po przemyśleniu zagadnienia zrozumiałem, że choć metoda `highlight()` nie ma żadnego znaczenia z punktu widzenia wykonywanych obliczeń, to jednak służy do wyrażenia intencji. W kodzie wywołującym tę metodę można było zwrócić uwagę na rozwiązywany problem, którym w tym przypadku było wyróżnienie fragmentu ekranu.

Można zastanowić się nad wykorzystaniem komunikatu wyjaśniającego w sytuacjach, gdy odczuwamy potrzebę skomentowania jakiegoś wiersza kodu. Kiedy widzę następujący wiersz kodu:

```
flags|= LOADED_BIT; // ustawienie wartości bitu zajętości
```

wolałbym użyć następującego wywołania:

```
setLoadedFlag();
```

Choć implementacja metody `setLoadedFlag()` jest trywialnie prosta, została ona jednak utworzona w celu wyrażenia intencji.

```
void setLoadedFlag() {
    flags|= LOADED_BIT;
}
```

Zdarza się, że metody pomocnicze wywoływane przez komunikaty wyjaśniające stają się cennymi miejscami dalszego rozwijania kodu. Miło jest móc skorzystać z możliwości, kiedy się ona nadarzy. Niemniej jednak podstawowym celem, w którym stosuję komunikaty wyjaśniające, jest bardziej precyzyjne wyrażenie moich intencji.

## Przepływ wyjątkowy

Oprócz przepływu głównego programy mają także co najmniej jeden przepływ wyjątkowy. Stanowią one ścieżki obliczeń, których wyrażanie nie jest tak ważne, gdyż są rzadziej wykonywane, rzadziej zmieniane, a czasami także mniej istotne od przepływu głównego z koncepcyjnego punktu widzenia. Przepływ główny należy wyrażać w sposób przejrzysty, natomiast ścieżki wyjątkowe — w możliwie jak najbardziej przejrzysty, lecz jednocześnie taki, by nie zaciemniał przekazu przepływu głównego. Klauzule strażników oraz wyjątki są dwoma sposobami wyrażania przepływu wyjątkowego.

Analiza programów jest znacznie łatwiejsza, jeśli tworzące je instrukcje są wykonywane sekwencyjnie, jedna po drugiej. W takich przypadkach, aby zrozumieć intencję programu, osoby przeglądające kod mogą skorzystać z wygodnych i dobrze znanych umiejętności czytania prozy. Czasami jednak może się zdarzyć, że program będzie miał kilka ścieżek realizacji. Wyrażanie ich wszystkich w taki sam sposób może doprowadzić do dużego chaosu i sytuacji, gdy flagi ustawiane w jednym miejscu kod są używane w innych, a wartości wynikowe mogą mieć specjalne znaczenia. Wówczas znalezienie odpowiedzi na pytanie: „Które instrukcje są wykonywane?” staje się zadaniem z pogranicza archeologii i logiki. Należy zatem wybrać przepływ główny i przejrzyste go wyrazić. A do wyrażenia wszelkich innych ścieżek w programie użyć wyjątków.

## Klauzula strażnika

Choć programy mają pewien przepływ główny, to jednak, w niektórych sytuacjach, może zaistnieć konieczność wykonywania ich innymi ścieżkami. Klauzula strażnika pozwala wyrazić prostą i logiczną sytuację wyjątkową, posiadającą wyłącznie lokalne konsekwencje. Porównajmy dwa przedstawione poniżej fragmenty kodu:

```
void initialize() {
    if (!isInitialized()) {
        ...
    }
}
```

oraz:

```
void initialize() {
    if (isInitialized())
        return;
    ...
}
```

W pierwszym z nich, kiedy zaczynam analizować pierwszą klauzulę instrukcji warunkowej, zwracam uwagę, by później poszukać klauzuli `else`. Zupełnie jakbym w myślach odłożył warunek na stos. Jest to czynnik, który rozprasza uwagę. W drugim przykładzie pierwsze dwa wiersze metody informują mnie jedynie o fakcie — odbiorca nie został zainicjowany.

Konstrukcja `if-then-else` wyraża alternatywne przepływy sterowania, z których oba są równie ważne. Klauzula strażnika nadaje się natomiast do wyrażenia innej sytuacji — sytuacji, w której jeden przepływ sterowania jest ważniejszy od drugiego. W przedstawionym wyżej przykładzie ze sprawdzaniem inicjalizacji ważniejszym przepływem sterowania jest ten określający, co się stanie, kiedy obiekt będzie zainicjowany. Oprócz niego w kodzie można zwrócić uwagę tylko na jeden prosty fakt, że niezależnie od tego, ile razy zażądamy zainicjowania obiektu, kod wykonujący tę inicjalizację zostanie wykonany tylko jeden raz.

Niegdyś istniało pewne przykazanie programistyczne, nakazujące, by każdy podprogram miał tylko jeden punkt wejścia i jeden punkt wyjścia. Sformułowano go, by zapobiec zamieszaniu, które mogłoby wystąpić, gdyby realizacja podprogramu mogła się zaczynać w wielu miejscach jego kodu i w wielu miejscach sterowanie mogłoby go opuszczać. Takie rozwiązanie było sensowne w języku FORTRAN oraz w programach pisanych w języku assemblera, które w dużym stopniu korzystały z danych globalnych i w których już samo wskazanie wykonywanych instrukcji było trudnym zadaniem. Natomiast w języku Java, w którym tworzone metody są niewielkie i korzystają przeważnie z danych lokalnych, takie podejście jest niepotrzebnie konserwatywne. Niemniej jeśli ten swoisty programistyczny folklor będzie rygorystycznie przestrzegany, uniemożliwi stosowanie wzorca klauzuli strażnika.

Klauzule strażników są szczególnie użyteczne w sytuacjach, gdy kontrolowanych jest wiele warunków:

```
void compute() {
    Server server= getServer();
    if (server != null) {
        Client client= server.getClient();
        if (client != null) {
            Request current= client.getRequest();
            if (current != null)
```

```

        processRequest(current);
    }
}

```

Zagnieżdżone warunki reprezentują jakieś problemy. W razie wykorzystania klauzul strażników ten sam kod wskazuje wymagania wstępne, niezbędne do obsługi żądania, a przy tym nie wymaga stosowania żadnych złożonych struktur sterujących:

```

void compute() {
    Server server= getServer();
    if (server == null)
        return;
    Client client= server.getClient();
    if (client == null)
        return;
    Request current= client.getRequest();
    if (current == null)
        return;
    processRequest(current);
}

```

Pewnym wariantem wzorca klauzuli strażnika jest instrukcja `continue` umieszczona w pętlach. Jej użycie stanowi komunikat: „Nie przejmuj się tym elementem i zajmij się następnym”.

```

while (line = reader.readLine()) {
    if (line.startsWith('#') || line.isEmpty())
        continue;
    // Normalne przetwarzanie
}

```

Także w tym przypadku chodzi o wskazanie (jedyne lokalnej) różnicy między przetwarzaniem normalnym i wyjątkowym.

## Wyjątek

Wyjątki są przydatne do wyrażania przeskoków w przepływie programu, przekraczających wiele poziomów wywołań. Jeśli w momencie wystąpienia problemu, takiego jak zapełnienie nośnika danych lub utrata połączenia sieciowego, na stosie będzie się znajdowało wiele poziomów wywołań, to najprawdopodobniej problem ten będzie można sensownie obsłużyć jedynie na znacznie niższym poziomie stosu. Zgłoszenie wyjątku w momencie wykrycia problemu oraz przechwycenie go w miejscu, gdzie problem ten można obsłużyć, jest znacznie lepszym rozwiązaniem niż umieszczanie w kodzie jawnych testów sprawdzających zaistnienie wszelkich możliwych warunków wyjątkowych, z których i tak w danym miejscu żadnego nie można obsłużyć.

Jednak stosowanie wyjątków jest kosztowne. Z punktu widzenia projektu programu są one problematyczne. Fakt, że wywoływana metoda zgłasza wyjątek, ma bowiem wpływ zarówno na projekt, jak i na implementację wszystkich metod od momentu zgłoszenia wyjątku aż do dotarcia do metody, w której zostanie on przechwycony. Wyjątki



utrudniają także śledzenie przepływu sterowania, gdyż sąsiadujące ze sobą instrukcje mogą znajdować się w innych metodach, obiektach lub pakietach. Kod, który zamiast z konstrukcji warunkowych i komunikatów korzysta z wyjątków, jest diabelnie trudny do zrozumienia, gdyż bezustannie trzeba starać się odgadnąć, co, oprócz standardowej struktury sterowania, może się w nim jeszcze zdarzyć. Krótko mówiąc, zawsze gdy to tylko możliwe, przepływ sterowania należy wyrażać przy użyciu sekwencji, komunikatów, iteracji oraz konstrukcji warunkowych (w takiej kolejności). Wyjątków można używać, jeśli zrezygnowanie z nich utrudni proste wyrażenie przepływu głównego.

## Wyjątki sprawdzane

Jednym z niebezpieczeństw stosowania wyjątków jest możliwość wystąpienia sytuacji, gdy zgłoszony wyjątek nigdy nie zostanie przechwycony. Takie sytuacje mają tylko jeden koniec — przerwanie działania programu. Można jednak przypuszczać, że twórca programu chciałby mieć kontrolę nad tym, kiedy program zostanie nieoczekiwanie przerwany, móc wyświetlić wówczas informacje niezbędne do zdiagnozowania problemu i poinformować użytkownika o tym, co się stało.

Takie nieprzechwycone wyjątki są jeszcze poważniejszym problemem, kiedy kto inny pisze kod, który je zgłasza, a kto inny kod, który je obsługuje. Wówczas każdy chybiony przekaz prowadzi do nagłego i nieuprzedzonego zakończenia działania programu.

Aby nie dopuścić do takich sytuacji, w języku Java wprowadzono wyjątki sprawdzane. Są one jawnie deklarowane przez programistę oraz sprawdzane przez kompilator. Kod, w którym może się pojawić taki wyjątek, musi go przechwycić lub przekazać dalej.

Stosowanie wyjątków sprawdzanych wiąże się ze znaczącymi kosztami. Pierwszym z nich jest koszt samej deklaracji. Użycie wyjątku bez trudu może powodować nawet dwukrotne wydłużenie deklaracji metody i dodanie kolejnego elementu, który należy przeczytać, zrozumieć i uwzględnić na wszystkich poziomach pomiędzy momentem zgłoszenia i obsługi. Poza tym wyjątki sprawdzane znacząco utrudniają modyfikowanie kodu. Refaktoryzacja kodu korzystającego z takich wyjątków jest znacznie trudniejsza i bardziej żmudna niż kodu, w którym nie są one stosowane, i to bez względu na to, że nowoczesne narzędzia programistyczne ułatwiają takie zmiany.

## Propagacja wyjątków

Wyjątki pojawiają się na różnych poziomach abstrakcji. Przechwytywanie i raportowanie o wyjątkach niskiego poziomu może być kłopotliwe dla osób, które się ich nie spodziewały. Kiedy serwer WWW wyświetla stronę błędu z obrazem stosu rozpoczynającym się od `NullPointerException`, to nie wiem, co mam zrobić z tą informacją. Wolałbym zobaczyć komunikat taki jak: „Programista nie przewidział zaistniałego scenariusza”. Nie miałbym także nic przeciwko temu, żeby na stronie zostały wyświetlone wskazówki dotyczące dodatkowych informacji, które można by przesłać do programisty,

aby ułatwić mu zdiagnozowanie problemu; natomiast wyświetlanie niewyjaśnionych w żaden sposób technicznych szczegółów nie jest ani użyteczne, ani pomocne.

Wyjątki niskiego poziomu często zawierają cenne informacje, przydatne do diagnozowania zaistniałych problemów. Takie wyjątki można umieszczać wewnątrz wyjątków wyższych poziomów; dzięki temu podczas prezentowania informacji o wyjątku (na przykład w dzienniku) znajdzie się w nich wszystko, co potrzebne do odnalezienia i rozwiązania problemu.

---

## Wniosek

W programach obiektowych sterowanie jest przekazywane pomiędzy metodami. W następnym rozdziale zostało opisane wykorzystanie metod w celu wyrażenia koncepcji występujących w obliczeniach.

# Skorowidz

## A

- Abstract Class, 34
- Abstract Interface, 33
- abstrakcja, 138, 140
- Access, 55
- akcesory, 59
- akcje
  - wykonywane na większą odległość, 112
- aktualizacja
  - danych, 112
    - metody ustawiające, 112
    - komentarzy, 100
    - powiększenie bazy użytkowników, 136
- algorytmy
  - zaimplementowane w obiektach, 111
- Anonymous Inner Class, 34
- API, 150
  - klasy biblioteczne, 137
  - problemy, 137

## B

- bezpieczna kopia, 112
  - metody ustawiające, 113
- bibliografia, 163
- biblioteka JUnit, 24
- biblioteki stanu, 55
- boolean, 145
- Boolean Setting Method, 91

## C

- całkowity czas wywołania metody, 153
- Checked Exception, 78
- Choosing Message, 77

- Class, 33
- Collecting Parameter, 56
- Collection, 72
- Collection Accessor Method, 91
- Common State, 55
- Comparator, 122
- Complete Constructor, 91
- Composed Method, 90
- Conditional, 34
- Constant, 56
- Control Flow, 77
- Conversion, 91
- Conversion Constructor, 91
- Conversion Method, 91
- Creation, 91
- czas istnienia zmiennych, 63

## D

- dane
  - a logika, 34, 44
  - pobieranie, 58
  - powiązane
    - dostęp pośredni, 59
  - przeniesienie, 112
  - stałe
    - w różnych miejscach kodu, 70
  - szybkość zmian, 29
  - typu
    - ArrayList, 73
    - Collection, 73
    - HashSet, 73
  - zapisywanie, 58
  - zmienność, 34
  - kolekcje, 115

Debug Print Method, 91  
 Declared Type, 56  
 Decomposing Message, 77  
 decyzje
 

- programistyczne, 17
- projektowe, 37

 dekompozycja funkcjonalna, 81  
 delegacje, 50
 

- konstrukcje warunkowe, 49
- odmiana, 51
- przechowywanie, 51

 Delegation, 34  
 Direct Access, 55  
 domyślna wartość parametru, 146  
 dostęp, 57
 

- bezpośredni, 58
  - reguły używania, 59
  - w kodzie klientów, 59
  - w ramach klasy, 59
- czytelność, 58
- do elementów projektowych, 38
- do kolekcji
  - obiektu, 106
  - ogólny, 107
- do pola, 91
- do przechowywanej wartości, 57
- do stanu, 59
  - obiektu, 110
- ograniczenie zbioru typów, 63
- pośredni, 59

 dostosowywanie wysiłku do celu, 149  
 Double Dispatch, 77  
 dwukrotne przydzielanie, 80
 

- powtórzenia, 81

 dziedziczenie, 44
 

- klasy wewnętrznej, 47
- ograniczenia, 44, 46
- po klasach kolekcji, 130
- równoległe hierarchie klas, 44
- wyważone stosowanie, 35

 dzielenie logiki, 90

## E

Eager initialization, 56  
 Eclipse, 135  
 efektywność programowania obiektowego, 34  
 elastyczność, 21, 24, 38
 

- a złożoność, 25

dostęp, 58
 

- do zmiennej, 58
- pośredni, 59

 dwukrotne przydzielanie, 81  
 konstruktor bezargumentowy, 104  
 metoda
 

- dostępu do kolekcji, 106
- strumień sformatowany, 99

 obiekty
 

- w kolekcjach, 115
- wytwórcze, 144

 platformy, 140  
 sekwencja metod ustawiających, 104  
 udostępnianie informacji, 73  
 ujawnianie metod, 94  
 widoczność metod, 95  
 wielokrotne dziedziczenie, 38  
 wykorzystanie delegacji, 50  
 zapewnienie przekazu, 73  
 zastosowanie stałych, 154  
 Equality Method, 91  
 Exception, 78  
 Exception Propagation, 78  
 Exceptional Flow, 78  
 Explaining Message, 77  
 Extrinsic State, 55

## F

fabryka
 

- statyczna, 143
- wewnętrzna, 106

 Factory Method, 91  
 Field, 56  
 flaga, 44
 

- bordered, 61
- logiczna, 65

 funkcja
 

- gromadzenie w bibliotece, 53
- typ wynikowy, 99

## G

Getting Method, 91  
 grupowanie
 

- elementów przepływu sterowania, 78
- powiązanych ze sobą obiektów, 115
- złożony algorytm, 81

 Guard Clause, 78

## H

Helper Method, 90  
 hierarchia  
   bez powtórzeń, 45  
   równoległa, 45  
   uproszczona, 52  
 horyzont zdarzeń, 104

## I

implementacja  
 @RunWith, 140  
 gniazda, 46  
 interfejsu  
   Collection, 124  
   List, 125  
   Map, 126  
   Set, 125  
 kolekcji, 123  
   niezmiennych, 129  
   nowych klas, 130  
 kompletnego konstruktora, 105  
 komunikat wyboru, 80  
 konwersji, 102  
 metoda ustawiająca, 111  
 platformy, 138  
   pomiarowej, 152  
   wielokrotna protokołu, 46  
 Implementor, 34  
 Indirect Access, 55  
 inicjalizacja, 73  
   deklaracja zmiennej, 73  
   leniwa, 74  
     fabryki wewnętrzne, 106  
     metody pobierające, 111  
     środowiska, 74  
   pół obiektu, 74  
   wczesna, 73  
 Initialization, 56  
 Inner Class, 34  
 Instance-Specific Behavior, 34  
 instancje  
   interfejsu  
     Command, 40  
     ReversibleCommand, 40  
   określanie sposobu działania, 48  
   tej samej klasy, 48  
   tworzenie zachowań, 51

instrukcja warunkowa  
   for, 120  
   if/else, 48  
   switch, 48  
 instrukcje  
   continue, 86  
   wykonywane sekwencyjne, 84  
 intencje  
   a implementacja, 83  
 Intention-Revealing Name, 90  
 Interface, 33  
 interfejs, 37  
   a hierarchie klas, 40  
   a klasy abstrakcyjne, 39  
   a nazwy metod, 94  
   abstrakcje platformy, 140  
     metody, 145  
   abstrakcyjny, 39  
   aktualizacja platformy, 140  
   anonimowe klasy wewnętrzne, 53  
   Collection, 119, 121  
     implementacje, 124  
   dodanie operacji, 40  
   elastyczność, 37  
   instanceof, 41  
   Iterable, 119, 120  
   jako klasa bez implementacji, 38  
   jako obiekt zaimplementowany, 39  
   jako typ wynikowy metod, 99  
 klasy  
   Array, 119, 120  
   Object, 101  
 kolekcji, 119  
 koszty stosowania, 37  
 List, 119, 121  
   implementacje, 125  
 Map, 119, 123  
   implementacje, 126  
   testowanie, 159  
 metoda  
   ustawiająca, 111  
   wykorzystująca stałe, 70  
 modyfikowanie  
   implementacji, 38  
   struktury, 41  
 nieprzewidywalność oprogramowania, 37  
 obiektu źródłowego i docelowego, 102  
 obliczeń alternatywny, 90  
 określanie nazw, 38

## interfejs

- operacje publiczne, 38
- proceduralny, 42
- Set, 119, 121
  - implementacje, 125
  - porównanie implementacji, 158
- SortedSet, 119, 122, 125, 158
- ustawienie wartości pola, 111
- wersjonowany, 39, 140
- zalety, 140
- zarządcy układu, 140
- zmienność komunikatów, 70
- zmienny, 41

Internal Factory, 91

Inviting Message, 77

iterator, 94, 107

- mapy, 123

## J

## Java

- mechanizmy tworzenia interfejsów, 38
- możliwość zmian interfejsu, 39
- obiekt parametrów, 69
- organizacja kodu, 78
- typy podstawowe, 41
- wartości parametrów, 68

jawność kodu, 80

## język

- obiektyowy, 57
- proceduralny
  - mechanizm ukrywania informacji, 79
- Smalltalk, 83

JUnit, 133

- wykonywanie testów, 152

## K

## klarowność

- dostęp bezpośredni, 58

## klasa

- abstrakcyjna
  - a interfejsy, 39
  - dodawanie nowych operacji, 39
  - ograniczenia, 39
  - Socket, 46
- Account, 147
- ArrayList, 123, 124, 125, 143
- bazowa, 35

abstrakcja platformy, 141

abstrakcyjna, 39

anonimowa, 53

dostęp klientów, 141

możliwość tworzenia instancji, 40

prosta nazwa, 35

rozdzielenie logiki, 45

testu, 156

biblioteczna, 53, 137

zastąpienie obiektami, 54

Bordered, 61

Brush, 80

Cartesian, 103

Collection, 54, 118

CollectionFactory, 144

Collections, 137

funkcjonalności, 128

jednoelementowe kolekcje, 129

niezmienne kolekcje, 129

puste kolekcje, 129

sortowanie, 128

wyszukiwanie, 128

Comparator, 139

ComplexCalculator, 97

Customer, 94

deklaracja, 34

dobieranie nazwy, 35

Enumerator, 135

Figure, 35

File, 103

Handle, 36

HashMap, 126

HashSet, 123, 124, 125, 159

idea, 33

interfejsy, 38

jako element projektowy, 35

JUnitCore, 139

komunikacja, 33

Library, 54, 130

LinkedHashMap, 126

LinkedHashSet, 125

LinkedList, 125

List, 54

ListTest, 52

Map, 118

MethodsTimer, 150

MethodTimer

wydajność, 152

Nothing, 151

- organizowanie w hierarchie, 35
- pochodna, 35, 44
  - awaria, 98
  - konstrukcje warunkowe, 49
  - kwalifikowana nazwa, 36
  - różne testy, 52
  - usprawnienia, 106
- Point, 103
- Polar, 103
- potomna
  - wywołanie własnego kodu, 98
- RectangleTool, 51
- Set, 118
- SetVsArrayList, 156
- Shape, 80
- String, 103
- Train, 44
- Transaction, 147
- TreeMap, 126
- TreeSet, 125, 138, 158
- używana w jednym miejscu, 53
- Vector, 135
- Vehicle, 44
- w pakiecie, 135
- wewnętrzna, 47
  - anonimowa, 53
  - kopie obiektów, 47
  - niezależna, 48
- Widget, 145
- klauzula strażnika, 84
- użyteczność, 85
- wymagania wstępne obsługi żądania, 86
- klucze, 123
- kod
  - a dane, 27
  - a zapewnienie wydajności, 118
  - abstrakcyjność, 105
  - aktualizacja, 135
    - automatyczna, 135
  - analiza
    - metody konwersji, 103
    - metod przeciężonych, 99
  - anonimowej klasy wewnętrznej, 53
  - czytelność, 82, 93
  - deklaracje zmiennych, 73, 117
  - deklaratywny, 28
  - deklarowanie pól, 65
  - dostosowanie stylu do kontekstu, 160
  - fragmenty niezalecane, 135
  - informacje specjalnego przeznaczenia, 62
  - inicjalizacja leniwa, 75
  - instrukcje warunkowe, 48
  - intencje twórcy, 22
  - jakość, 12
  - kolekcje, 117
  - komentarze, 100
  - komunikaty, 79
    - wybierające, 80
  - komunikatywność, 22
  - konstrukcje warunkowe, 46
  - lokalne konsekwencje zmian, 26
  - mechanizmy optymalizacji, 149
  - metody
    - określające wartości logiczne, 108
    - pomocniczej, 101
    - wytwórcze, 105
  - nadawanie struktury, 93
  - obiekty
    - metody, 97
    - powtórzenia wierszy, 101
    - wytwórcze, 144
  - powtarzanie, 26
    - eliminacja, 28
  - prostota, 24
  - przejrzystość, 32
  - przeniesienie logiki, 109
  - rozwój platformy, 133
  - różne kombinacje dostępu, 63
  - skopiowany, 45
  - stan, 57
  - symetria, 27, 82
  - testy warunków wyjątkowych, 86
  - usprawnienie obliczeń, 83
  - w klasie bazowej, 45
  - wciążenia, 92
  - wrażliwość, 112
  - współużytkowanie
    - delegacje, 51
  - wyjątki sprawdzane, 87
  - zabezpieczenie, 113
  - zgodne zmiany, 136
- kolejka wiadomości
  - lista, 121
- kolekcje, 63, 115
  - aktualizowanie platform, 135
  - ArrayList, 157
  - dodawanie elementów, 116

- kolekcje
  - elementy
    - unikalność, 118
    - uporządkowanie, 122
    - usunięcie powtórzeń, 122
    - usuwanie, 120
    - znaczenie kolejności, 117
  - implementacje, 123
  - inicjalizacja klasy
    - testy, 156
  - interfejsy, 119
    - i klasy, 124
  - jako obiekty, 116
  - jako zbiór, 121
  - jako zbiór obiektów, 117
  - jednoelementowe, 129
  - kluczy, 123
  - LinkedList, 157
  - metafory, 116
  - modyfikowanie zawartości, 120
  - niezmienne, 129
  - odwołanie do obiektów, 116
  - pojęcia ortogonalne, 117
  - poła reprezentujące, 116
  - porównywanie, 157
  - posiadające tożsamość, 116
  - przekazywanie, 68
  - puste, 129
  - rozszerzanie, 116, 130
  - równoległe architektury, 135
  - sortowanie, 128
  - typu Collection, 155
  - typu Map
    - elementy, 123
  - typu Set
    - duplikaty elementów, 121
    - przechowywanie elementów, 121
  - w kolekcji, 107
  - wartości, 123
  - wielkość, 117
  - wydajność, 118
    - pomiary, 149
  - wyrażanie intencji, 115
  - wyszukiwanie, 128
  - zabezpieczenie, 107, 120
  - zagadnienia, 117
- komentarz
  - do metody, 100
  - dokumentujący, 100
- komparator, 122, 126
  - byFirstName, 139
- komputery mainframe, 24
- komunikat, 46, 79
  - a konstrukcje warunkowe, 49
  - dekomponujący, 81
  - display(), 80
  - logika, 49
  - metoda, 101
  - odwracający, 82
  - polimorficzny, 46, 80
  - sekwencjonujący, 81
  - wybierający, 80
    - kaskada, 80
  - wyjaśniający, 83
    - komentarz kodu, 84
  - zapraszający, 83
- komunikatywność, 21, 22
  - a elastyczność, 25
  - a prostota, 24
  - określanie nazw klas, 36
  - podstawa ekonomiczna, 23
  - stan zmienny, 61
- komunikowanie intencji
  - kolekcje, 115
  - za pośrednictwem kodu, 13
- konfiguracja
  - platformy, 138
  - zastosowanie stałych, 154
- konstrukcja
  - if-then-else, 85
- konstrukcje warunkowe, 48
- konstruktor
  - czteroargumentowy, 105
  - File(String name), 103
  - główny, 105
  - klasy
    - MethodTimer, 152
    - ServerSocket, 68
    - wewnętrznej, 47
  - kompletny, 104
  - konwertujący, 103
  - przygotowanie obiektów, 104
  - StringReader(String contents), 103
  - tworzenie obiektów platformy, 143
  - URL(String spec), 103
  - zamiennik
    - metoda statyczna, 96
    - zastosowanie metod pomocniczych, 101



konwersja, 102  
 a zmiana klienta, 103  
 cel, 103  
 implementacja, 102  
 między obiektami podobnych typów, 102  
 obiektu źródłowego na docelowy, 102

koszty  
 abstrakcji, 40  
 aktualizacji  
 niezgodnych, 135  
 redukcja, 134  
 testów, 100  
 analizy i zrozumienia kodu, 32  
 kolekcja w kolekcji, 107  
 komentarzy, 100  
 modyfikowania programów  
 obiekty, 104  
 napisania, 31  
 określania zmiennych, 74  
 oprogramowania, 31  
 selektora dołączanego, 53  
 tworzenia  
 oprogramowania, 23  
 platform, 134  
 utrzymania, 31  
 widoczność metod, 95  
 wydajności, 118  
 wyjątki sprawdzane, 87  
 wywoływania obiektów, 95  
 zachowania zgodności, 134  
 zachowań zależnych od instancji, 48  
 zmiany kodu, 133

## L

Lazy initialization, 56  
 Library Class, 34  
 liczba  
 iteracji, 153  
 potrzebnych konwesji, 102  
 licznik czasu, 150  
 dokładność, 151  
 narzut, 154  
 zewnętrzny interfejs, 150  
 List, 72  
 listy, 121  
 literate programming, 22  
 Local Variable, 56

logika  
 a metoda pobierająca, 111  
 anonimowe klasy wewnętrzne, 53  
 grupowanie w klasy, 34  
 i dane, 27  
 zgrupowanie, 29  
 komunikaty polimorficzne, 115  
 konstrukcje warunkowe, 115  
 metody pomocniczej, 101  
 obiektu, 48  
 obsługi kliknięcia, 51  
 operująca  
 na różnych danych, 44  
 na tych samych danych, 44  
 parametry obiektowe, 70  
 podział na metody, 89  
 przeniesienie, 112  
 w jednej klasie, 48  
 w metodach statycznych, 53  
 w różnych instancjach, 50  
 warunkowa  
 zastąpienie komunikatami, 49  
 wyrażanie  
 podobieństw i różnic, 74  
 przez komunikaty, 79  
 zmienna, 46  
 zmienność, 34

lokalne konsekwencje, 26

## M

Main Flow, 77  
 mapy, 123  
 porównywanie, 159  
 stan zmienny, 60  
 mechanizm  
 odzwierciedlania, 52, 153  
 klasy wewnętrzne, 47  
 optymalizacji, 149  
 przekazywania zmiennej liczby  
 argumentów, 69  
 przepływu sterowania  
 komunikaty, 79  
 Message, 77  
 metafora  
 nazywanie klas, 35  
 Method Comment, 90  
 Method Object, 90  
 Method Return Type, 90

- Method Visibility, 90
- metoda, 89
  - abstrakcyjna, 83
    - dodawanie do klasy bazowej, 141
  - addAll(), 121
  - akcesorów, 59
  - argumenty opcjonalne, 69
  - calculate(), 97
  - chroniona, 95, 100
  - clear(), 130
  - Collections.binarySearch(list, element), 128
  - Collections.factory(), 144
  - Collections.singleton(), 129
  - complexCalculation(), 97
  - compute(), 82
  - contains(), 117, 124
  - createArrayList(), 144
  - czas wykonania, 153
  - display(), 48
  - displayWith(), 81
  - długość, 92
  - do pomiaru czasu, 156
  - dostępna w pakiecie, 95
  - dostępu do kolekcji, 106, 113
  - equal(), 121
  - equals(), 91, 109
  - fabryki statycznej, 154
  - find(), 94
  - get(int), 157
  - getDeclaredMethods(), 152
  - getX(), 103
  - getY(), 103
  - hashCode(), 91, 109
  - hasNext(), 94
  - highlight(), 83
  - inactive(), 145
  - indexOf(), 128
  - inicjalizacja wartości pola, 74
  - instancji obiektu wytwórczego, 106
  - invisible(), 145
  - iterator(), 120
  - komentarze, 100
  - komunikat odwracający, 82
  - komunikatu informacyjnego, 101
  - konwertująca, 102
  - koszty aktualizacji, 136
  - mouseDown(), 51
  - nazwa określająca przeznaczenie, 93
  - next(), 94, 95
  - objektu źródłowego, 102
  - ogólna struktura kodu, 92
  - określająca wartości logiczne, 108
  - overheadTimer(), 154
  - paragraph.centered(), 111
  - platformy, 138
  - pobierająca, 110, 141
    - parametry, 68
    - publiczna, 111
      - udostępnianie klientom, 145
      - wewnętrzna, 111
  - podobne listy parametrów, 54
  - pomiarowa, 150, 151
  - pomocnicza, 100
    - komunikaty wyjaśniające, 84
    - zmienna lokalna, 64
  - prywatna, 96, 100
  - przeciążona, 98
    - cel, 99
  - przekazywanie
    - łańcucha znaków String, 98
    - parametrów, 68
  - przesłanie, 102
  - przesłonięta, 98
  - publiczna, 95
    - zwracająca wartość pola, 111
  - remove(), 107, 120, 124
  - removeAll(), 121
  - report(), 150, 151
  - retainAll(), 121
  - reverse(), 83
  - reverse(list), 128
  - równości, 109
  - run(), 153
  - run(Class ...), 139
  - run(Classes ...), 146
  - runTest(), 52
  - search(), 150
  - setLoadedFlag(), 84
  - setVisibility(), 145
  - setVisibility(boolean), 145
  - setVisibility(State), 145
  - sfinalizowana, 96
  - shuffle(list), 128
  - size(), 117
  - sort(list), 54, 129
  - sort(list, comparator), 129
  - statyczna, 96
    - przekształcenie w metody instancji, 54

String.asFile(), 103  
 suit(), 28  
 toArray(), 130  
 toString(), 91, 101  
 typ wyników, 99  
 typu Collection, 99, 121  
 udostępnianie, 96  
 ustawiająca, 106, 108, 111  
   publiczna, 111  
   udostępnianie klientom, 145  
   wewnętrzna, 112  
 visible(), 145  
 w klasie bazowej, 98  
 wytwórcza, 103, 105  
   tworzenie obiektów, 144  
 wywołująca inne metody, 94  
 zabezpieczenie, 96  
 zapytania, 108  
 złożona, 92, 113  
   konsekwencje stosowania, 100  
 zwracająca iterator, 107  
 minikomputery, 24  
 minimalizacja powtórzeń, 26  
 modyfikator  
   abstract, 39  
   final, 65, 96  
   package, 62  
   private, 62  
   protected, 62  
   public, 62  
   static, 48  
 modyfikowanie programów  
   typy wyników, 99  
 motywacja, 31

## N

narzędzia refaktoryzujące, 100  
 narzut, 153  
   czasowy  
     eliminacja, 154  
   dynamiczne wywoływanie metod, 154  
 nazwy  
   interfejsów, 39  
   klas, 35  
     funkcje, 36  
     pochodnych, 36  
     wielopoziomowe hierarchie klas, 36  
 komunikatów dekomponujących, 81

metod, 80, 89, 93  
   przedrostek, 110, 111  
 obliczeń, 99  
 stałych, 70  
 sugerujące znaczenie, 71  
 zmiennych, 63, 71  
   użyteczność, 14  
 niezgodne aktualizacje, 134  
 społeczność klientów, 136  
 struktura, 136  
 numer seryjny, 109

## O

obiekt  
 Account, 43  
 Comparator, 138  
 dane specjalnego przeznaczenia, 123  
 dołączany, 51  
 GraphicEditor, 51  
 GraphicsContext, 68  
 HashMap, 159  
 Iterable, 120  
 Iterator, 120  
 jako łańcuch znaków, 101  
 JUnitCore, 139  
 konstrukcje warunkowe, 48  
 konstruktory, 104  
 konwersja, 102, 103  
   na wiele obiektów docelowych, 103  
 metafora kolekcji, 116  
 MethodTimer, 151, 152  
 metody, 93, 96  
   statyczna, 91  
 niezmienny, 43, 113  
 numer seryjny, 109  
 o zmiennym stanie, 41  
 odpowiedzialność za dane, 112  
 parametr wywołania wielu metod, 65  
 parametrów, 69  
   zastępowanie jawną listą, 70  
 platformy, 137  
 podejmowanie decyzji, 108  
 pole, 65  
 pomocniczy, 44  
   operacja prywatna, 106  
   uproszczenie projektu, 61  
 Rectangle, 105  
 RectangleTool, 51

- obiekt
    - reprezentujący kolekcje, 115
    - sprawdzenie równości, 109
    - stan
      - programu, 55
      - wspólny, 60
      - zapewnienie dostępu, 110
      - zewnętrzny, 62
      - zmienny, 43
    - TestResult, 68, 146
    - Transaction, 42
    - tworzenie
      - przez klientów, 142
      - sposób alternatywny, 105
    - typu Comparator, 126
    - umieszczenie w pamięci podręcznej, 105
    - unieważnienie stanu wewnętrznego, 107
    - utworzenie, 91
    - uzyskiwanie informacji, 101
    - w metodzie wytwórczej, 110
    - w pamięci podręcznej, 111
    - wartościowy
      - argumenty przeciwko, 43
    - wartość skrótu, 109
    - wymagania wstępne, 104
    - wytwórczy, 106, 144
    - zabezpieczenie, 96
    - zachowanie i stan, 55
    - zależność od stanu innego obiektu, 108
    - zapisanie w polu, 65
    - zmiana w klasie pochodnej, 106
  - odczytywanie
    - wartości logicznych, 91
  - odmienność
    - delegacje, 50
    - logika warunkowa, 50
    - tworzenie obiektu, 46
    - wprowadzanie nowej, 45
    - zbiór, 44
    - złożona, 44
  - odwoływanie
    - do elementów kolekcji, 118
  - ograniczenia
    - klas bazowych, 141
    - stylu stosowania platform, 139
    - widoczność metod, 94
    - wydajności
      - niewielkie metody, 92
  - określanie równości obiektów, 109
  - operacja
    - contains(), 160
    - containsKey(), 160
    - put(), 160
  - oprogramowanie
    - elastyczność, 37
    - koszty, 31
      - strategia redukcji, 31
      - wytworzenia, 31
    - nieprzewidywalność, 37
    - utrzymanie, 31
  - Overloaded Method, 90
  - Overriden Method, 90
- ## P
- pakiety, 135
    - wewnętrzne, 142
  - Parameter, 56
  - Parameter Object, 56
  - parametr, 66
    - brush, 80
    - obiektowy, 69
    - opcjonalny, 68
    - powiązania, 66
    - powtórzenie, 67
    - skojarzenie z obiektem, 66
    - TestResult, 146
    - zastąpienie wskaźnikiem, 67
    - zbierający, 67
  - platformy
    - abstrakcja, 140
    - aktualizacja, 134
      - idealne aktualizacje, 133
      - kodu, 135
      - warianty zgodności, 136
      - zgodne zmiany, 136
    - bez możliwości tworzenia obiektów, 143
    - deklarowanie pól, 134
    - ekonomiczne aspekty tworzenia, 134
    - funkcjonalność, 145
      - a rozwój, 137
    - implementacja, 139
      - skalowalność, 139
    - konfiguracja, 138
    - koszty, 146
    - łatwość użycia i modyfikacji, 145
    - ograniczenie możliwości zastosowania, 134
    - określanie pojęć, 147

- pomiarowe
      - sposoby wykorzystania, 154
    - reprezentacja w formie obiektów, 137
    - rozwód, 146
    - równowaga, 137
    - style stosowania, 138
    - tworzenie obiektów, 138, 144
      - fabryki statyczne, 143
      - konstruktory, 143
      - obiekt wytwórczy, 144
    - wersja przejściowa, 135
    - widoczność
      - pakiety, 142
    - zadania twórców, 147
    - zapewnienie wielkości, 147
  - Pluggable Selector, 34
  - podwyrażenia
    - eliminacja, 101
  - podział na metody, 90
  - pole, 65
    - borderColor, 61
    - borderWidth, 61
    - dane pomocnicze, 65
    - deklaracja, 65
    - flaga, 65
    - kolekcji, 130
    - komponenty, 66
    - publiczne, 58
    - stan, 65
    - strategia, 65
  - polimorfizm, 61
  - powielanie, 156
  - powtórzenia kodu, 26
  - poziomy abstrakcji, 92
  - prefiks, 71
  - private, 63
  - problemy
    - kopiowanie metod klasy bazowej, 98
    - rozszerzanie klas kolekcji, 130
    - wielokrotnego użycia, 89
    - wyjątki niskiego poziomu, 88
  - procedura
    - podsekwencja kroków, 81
  - programowanie
    - dane a logika, 44
    - dostęp bezpośredni, 58
    - elementy, 22
    - imperatywne, 28
    - objektowe, 33
      - cel wprowadzenia, 58
      - piśmienne, 22
      - problemy, 17
      - przechowywanie danych, 59
      - teoria, 21
      - wartości, 21
      - współbieżne
        - źródła, 15
      - zasady, 21
  - programy
    - korzyści zwiększenia złożoności, 160
    - modyfikowanie, 104
    - odmienność, 43
    - ścieżki realizacji, 48
    - wyrażanie podobieństw i różnic, 43
  - projektowanie oprogramowania
    - czynnik warunkujący, 31
    - możliwość zmiany strategii, 46
  - propagacja wyjątków, 87
  - prostota, 21, 23
  - przechwycenie wyjątku, 86, 87
    - niskiego poziomu, 87
  - przeciążanie, 99
  - przejrzystość
    - dostęp pośredni, 59
    - głównego przepływu programu, 79
    - obliczeń
      - metoda pomocnicza, 100
      - wspólny stan, 60
  - przekaz deklaracyjny, 28
  - przekazywanie
    - parametrów, 99
    - przez referencję
      - odpowiednik, 116
  - przepływ
    - główny, 78
      - wykonywanie innymi ścieżkami, 84
    - przeskoki, 86
    - sterowania, 78
      - alternatywny, 85
      - poprawne wyrażanie, 87
      - ważność, 85
      - wyjątki, 87
    - wyjatkowy, 84
  - przesłanie metod, 35, 98
  - przeszukiwanie hierarchii, 153
- ## Q
- Qualified Subclass Name, 33
  - Query Method, 91

## R

refaktoryzacja  
tworzenie obiektu metody, 97

Reversing Message, 77

Role-Suggesting Name, 56

rozwijanie platform, 133

równoległe

architektury, 135

hierarchie klas, 26

rzutowane w dół, 40

## S

Safe Copy, 91

scalanie wyników, 67

sekwencja

instrukcji, 77

kontroli, 78

Setting Method, 91

Simple Superclass Name, 33

słowo kluczowe

abstract, 39, 141

extends, 46

final, 142

implements, 46

socket, 46

Specialization, 34

specjalizacja

dobór wielkości metod, 93

komunikat dekomponujący, 81

spis szablonów, 167

stałe, 70

ONE\_SECOND, 154

znaczenie, 70

stan, 41, 55, 56

efektywne zarządzanie, 57

elementy podobne, 57

odwołania

dostęp bezpośredni, 59

podział na fragmenty, 57

pole, 66

przekazywanie informacji między

obiektami, 66

wspólny, 60

zewnętrzny, 62

kopiowanie, 62

mapy, 123

zmienny, 60

mapy, 123

przechowywanie, 60

wada, 61

State, 55

static, 63, 96

strategia

implementacji, 31

w nazwach metod, 93

przyrostowego wprowadzania zmian, 135

redukcji kosztów ogólnych, 32

redukcji złożoności aplikacji, 134

zapewniania wydajności, 118

zwiększenia złożoności platformy, 134

struktura drzewiasta

linearyzacja, 67

powiązania między węzłami, 66

strumień OutputStream, 99

styl programowania, 22

funkcyjny, 41

styl stosowania, 138

Subclass, 34

symetria

czasowa, 29

koodu, 83

pojęciowa, 27

pól, 29

tempo zmian, 29

w kodzie, 27

sytuacja statyczna, 41

## Ś

ścieżka realizacji programu, 48

prawdopodobieństwo poprawności, 48

ścieżki wyjątkowe, 84

## T

tablice, 117, 119, 120

zamiana na kolekcję, 120

techniki kompresji, 35

tekstowe reprezentacje obiektu, 102

tempo zmian, 29

teoria programowania, 21

testowanie równości obiektów, 109

testy

@After, 139

@Before, 139

@RunWith, 140  
 @Test, 139  
 abstrakcyjna implementacja metody, 156  
 anonimowych klas wewnętrznych, 53  
 czas dodania i usunięcia elementu, 157  
 czas dostępu do elementów kolekcji, 158  
 czas przeglądania zawartości kolekcji, 156  
 czas przeszukiwania listy, 150  
 czas wykonania metody, 152, 153  
 czas wykonania operacji, 150  
 czas zmodyfikowania kolekcji, 156  
 implementacji interfejsu Map, 159  
 kolekcji ArrayList oraz LinkedList, 157  
 koszt aktualizacji, 100  
 modyfikacja zawartości zbiorów, 158  
 operacji, 150  
 pomiarowe, 155  
 porównywanie kolekcji, 155  
 selektor dołączany, 52  
 sprawdzenie występowania elementu  
   zbioru, 158  
 umieszczanie w jednej klasie, 53  
 zautomatyzowane, 100  
 zbiorów, 158  
 tworzenie  
   API, 137  
   biblioteki widżetów, 145  
   obiektów, 138  
     platformy, 138  
     wnioski, 144  
     wybór stylu, 142  
   platform  
     prostota a możliwość rozwoju, 146  
 typ wyliczeniowy States, 145  
 typ wynikowy metody, 99  
   generalizacja, 99  
   konwertującej, 103  
   void, 99

## U

ujawnianie metod, 94  
 unikalność elementów, 118  
 unikanie powtórzeń, 26  
 uporządkowanie elementów, 117  
 utożsamianie nazw, 91, 112, 116  
 utworzenie, 103

## V

Value Object, 34  
 Variable, 55  
 Variable State, 55  
 Versioned Interface, 33

## W

wartości, 22  
   mieszające, 155  
   oprogramowania, 23  
   pobierania danych, 160  
   skrót, 109  
 węzeł rodzica, 66  
 widoczność  
   metody, 94  
   ograniczenie, 96  
   system pakietów Javy, 142  
 widżet  
   stan  
     wspólny, 61  
     zmienny, 61  
 współbieżność  
   a stan, 57  
 współużytkowanie implementacji, 44  
 wyciekanie informacji projektowych, 145  
 wydajność, 118  
   dostęp, 58  
   inicjalizacja, 73  
   kolekcji, 118  
     ArrayList, 124, 125  
     HashSet, 124, 125  
     implementacja, 123  
     LinkedHashSet, 125  
     LinkedList, 125  
     operacje sortowania, 129  
     porównanie, 125, 126  
     reprezentacja danych, 155  
     TreeSet, 126  
 map, 159  
 metody  
   indexOf(), 128  
   niewielkie, 92  
 pomiary, 149  
   implementacja, 151  
   w konstruktorze, 153  
 skalowanie ilości danych, 151

- wydajność
  - tablice, 120
  - wyszukiwanie binarne, 128
  - zbiorów, 158
- wydziałanie odmienności, 50
- wyjątek, 86
  - ClassCastException, 109
  - IllegalArgumentException, 109
  - sprawdzany, 87
  - UnsupportedOperationException, 130
  - wady stosowania, 86
- wymiar zmienności
  - komunikaty wybierające, 80
- wyniki wywołań wielu metod, 67
- wyrażanie intencji
  - komunikat wyjaśniający, 83
  - parametry obiektowe, 69
  - proceduralne, 46
  - przy użyciu obiektu, 46
- wyrażenie
  - new ArrayList(), 143
  - new ServerSocket(), 138
- wyszukiwanie binarne, 128
- wywołanie
  - obliczeń, 57
  - procedur, 79
  - super.metoda(), 98
- wzorce, 17, 22
  - anonimowa klasa wewnętrzna, 34, 53
    - ograniczenia, 53
  - bezpieczna kopia, 91, 112
  - cechy programowania, 17
  - delegacja, 34, 50
  - dostęp, 55, 57
    - bezpośredni, 55, 58
    - pośredni, 55, 59
  - dwukrotne przydzielanie, 80
  - fabryka wewnętrzna, 91, 106
  - implementacyjne, 11
    - a koszty oprogramowania, 31
    - koncentracja na korzyściach, 32
    - lista zasad, 26
    - zalety stosowania, 32
    - zaspokajanie potrzeb, 32
    - związane z tworzeniem, 104
  - implementator, 34, 46
  - inicjalizacja, 56, 73
    - leniwa, 56, 74
    - wczesna, 56, 73
  - interfejs, 33, 38
    - abstrakcyjny, 33, 37
    - wersjonowany, 33, 40
  - klasa, 33, 34
    - abstrakcyjna, 34, 39
    - biblioteczna, 34, 53
    - Collections, 128
    - pochodna, 34, 44
    - wewnętrzna, 34, 47
  - klauzula strażnika, 78, 84
  - kolekcje, 115
    - implementacje, 123
    - interfejsy, 119
    - metafory, 116
    - rozszerzanie, 130
    - zagadnienia, 117
  - komentarz do metody, 90, 100
  - kompletny konstruktor, 91
  - komunikat, 77, 79
    - dekomponujący, 77, 81
    - odwracający, 77, 82
    - wyberający, 77, 80
    - wyjaśniający, 77, 83
    - zapraszający, 77, 83
  - konstrukcja warunkowa, 34, 48
  - konstruktor
    - kompletny, 104
    - konwertujący, 91, 103
  - konwersja, 91, 102
  - kwalifikowana nazwa klasy pochodnej, 33, 36
  - lista założeń, 17
  - metoda
    - dostępu do kolekcji, 91, 106
    - komunikatu informacyjnego, 91, 101
    - konwertująca, 91, 102
    - określająca wartości logiczne, 91, 108
    - pobierająca, 91, 110
    - pomocnicza, 90, 100
    - przeciążona, 90, 98
    - przesłonięta, 90, 98
    - równości, 91, 109
    - ustawiająca, 91
    - wytwórcza, 91, 105
    - zapytania, 91, 108
    - złożona, 90, 92
  - modyfikowanie platform bez zmian
    - w aplikacjach, 133



nazwa  
     określająca przeznaczenie, 90, 93  
     sugerująca znaczenie, 56, 71  
 niezgodne aktualizacje, 134  
 obiekt  
     metody, 90, 96  
     parametrów, 56, 69  
     wartościowy, 34, 41  
 parametr, 56, 66  
     opcjonalny, 68  
     teleskopowy, 68  
     zbierający, 56, 67  
 podwójne przydzielanie, 77  
 pole, 56, 65  
 projektowe, 14  
     związki między klasami, 33  
 propagacja wyjątków, 78, 87  
 prosta nazwa klasy bazowej, 33, 35  
 przepływ  
     główny, 77, 78  
     sterowania, 77, 78  
     wyjątkowy, 78, 84  
 selektor dołączany, 34, 52  
 specjalizacja, 34, 43  
 sposoby prezentacji, 14, 18  
 sposoby wyrażania podobieństw i różnic, 44  
 stałe, 56, 70  
 stan, 55, 56  
     wspólny, 55, 60  
     zewnętrzny, 55, 62  
     zmienny, 55, 60  
 typ wynikowy metody, 90, 99  
 utworzenie, 91, 103  
 widoczność metody, 90, 94  
 współpraca, 18  
 wyjątek, 78, 86  
     sprawdzany, 78, 87  
 zachęcanie do wprowadzania zgodnych  
     zmian, 136  
 zachowanie zależne od instancji, 34, 48  
 zadeklarowany typ, 56, 72  
 zalety stosowania, 18  
 zmienna, 55, 62  
     lista argumentów, 68  
     lokalna, 56, 63  
 zwiększone koszty, 25

## Z

zachowanie zależne od instancji  
     anonimowe klasy wewnętrzne, 53  
     selektor dołączany, 52  
 zadeklarowany typ, 72  
 zakres pól  
     określanie, 62  
 zależności między klasami  
     konwersja, 102  
 zasady, 25  
     lokalne konsekwencje, 26, 118  
     minimalizacja powtórzeń, 26  
     połączenie logiki i danych, 27  
     programowania, 22  
     zalety stosowania, 21  
     przekaz deklaracyjny, 28  
     symetria, 27  
     użyteczność danych dla kodu, 62  
     tempo zmian, 29  
     zalety poznawania, 25  
     zrozumienie, 26  
 zautomatyzowane testy, 100  
 zbiory, 121  
     obiektów jako kolekcja, 117  
     pojemność  
         testowanie, 159  
     porównywanie, 158  
     posortowane, 122  
 zdarzenia SWT, 143  
 zgłoszenie wyjątku, 86  
 zgodne zmiany, 136  
 zgodność  
     domyślna wartość parametru, 146  
     tworzenie platform, 134  
     w przód, 136  
     wstecz, 134, 136  
 zgrupowanie obliczeń, 47  
 złożoność, 23  
     aplikacji  
         redukcja, 134  
         równoległe architektury, 135  
 zmienna, 62  
     count, 72  
     deklarowanie, 63, 74  
         typu, 72  
     each, 64, 72  
     element, 64

## zmienna

- gromadzenie, 63
- inicjalizacja, 74
- intencje, 63
- licznik, 64
- lokalna, 62, 63
  - deklarowanie, 63
  - przechowywanie elementu, 64
  - przeznaczenie, 63
- members, 72
- nazwa opisowa, 64
- nazywanie, 71
- nieprywatna
  - powiązanie między klasami, 66
- odczytywanie nazw, 72
- odwołująca się do kolekcji, 116
- poła, 62
- przeznaczenie, 71
- przypisywanie stanu, 73
- result, 63, 72

results, 63

statyczna, 62

typy używane IDE, 71

w obiekcie

- czas istnienia, 61

wielokrotne wykorzystanie, 64

wyjaśnienie, 64

zadeklarowana jako Iterable, 120

zakres, 62, 71

zapisywanie wartości, 58

zmienna liczba elementów, 115

zmienna lista argumentów, 68

zmienność

- kolekcje, 115

- w procesie tworzenia obiektów, 144

znacznik

- czasu, 64

**Ż**

źródła, 163

# PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



- 1. ZAREJESTRUJ SIĘ**
- 2. PREZENTUJ KSIĄZKI**
- 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**



# WZORCE IMPLEMENTACYJNE

Przy nauce programowania warto uczyć się na cudzych błędach. Programiści tworzący aplikację codziennie nатыkają się na przeróżne problemy oraz zagadnienia do rozwiązania. Rzadko jednak zdarza się, żeby były one wyjątkowe i niespotykane wcześniej. Jeżeli masz problem, możesz być prawie pewien, że ktoś też już go miał – i w dodatku rozwiązał. Właśnie tak powstały wzorce, które w jasny sposób opisują rozwiązania typowych problemów.

W tej książce znajdziesz 77 wzorców, które pozwolą Ci uniknąć wielu pułapek oraz rozwiązać najczęściej spotykane problemy. W trakcie lektury dowiesz się, jak przechowywać stan oraz gdzie umieścić logikę Twojej aplikacji. Ponadto poznasz najefektywniejsze sposoby sterowania przebiegiem programu oraz wybierzesz rodzaj kolekcji odpowiedni do Twoich potrzeb. Nauczysz się dobrać właściwe nazwy dla zmiennych i metod oraz przekonasz się, że można sprawnie opanować wysyp wyjątków. Książka ta jest obowiązkową lekturą każdego programisty. Dzięki niej Twoje życie stanie się prostsze, a Twoje oprogramowanie będzie bardziej przejrzyste!

## Poznaj sprawdzone wzorce, które ulepszą Twoje oprogramowanie!

Dzięki tej książce:

- ▼ poznasz 77 przydatnych wzorców
- ▼ wybierzesz odpowiedni typ kolekcji w zależności od problemu
- ▼ zaprojektujesz przejrzystą hierarchię klas
- ▼ zbudujesz lepszą i bardziej niezawodną aplikację

**helion.pl**  
księgarnia  
internetowa

Nr katalogowy: 17048



Księgarnia internetowa:  
<http://helion.pl>



Zamówienia telefoniczne:  
**0 801 339900**



**0 601 339900**



**Helion**

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 **WIECEJ**



KOD KORZYŚCI

ISBN 978-83-246-8644-5



Cena 39,90 zł

Informatyka w najlepszym wydaniu