Sprawdzone rozwiązania
Twoich problemów!

ADDISON-WESLEY

Wzorce implementacyjne

KENT BECK

Helion

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 by 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 jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

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

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
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

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

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.

Kup książkę Poleć książkę

- 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 nielokalne 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,

Kup książkę Poleć książkę

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 wyrażaniu głównego przepływu obliczeń.

Dlatego też ten główny przepływ programu należy wyrażać przejrzyście, 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.

Kup książkę Poleć książkę

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.

Załóżmy 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);
}
```

Kup książkę Poleć książkę

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 zrozumiałem 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.

Kup książkę Poleć książkę

Problemy z doborem 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. Przeanalizujmy 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();
}
```

Kup książkę Poleć książkę

```
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 wywoływać 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.

Kup książkę Poleć książkę

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 przejrzyście 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:

Kup książkę Poleć książkę

```
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 asemblera, 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)
```

Kup książkę Poleć książkę

```
            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` umieszczana 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 (jedynie 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

Kup książkę Poleć książkę

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 nieuprzejmego 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,

Kup książkę Poleć książkę

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.

Kup książkę Poleć książkę

Skorowidz

Kup książkę Poleć książkę

E

F

G

Kup książkę Poleć książkę

H

I

Kup książkę Poleć książkę

J

K

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

L

M

Kup książkę Poleć książkę

Kup książkę Poleć książkę

N

O

Kup książkę Poleć książkę

P

Kup książkę Poleć książkę

Q

Kup książkę Poleć książkę

R

S

Ś

T

Kup książkę Poleć książkę

U

V

W

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Z

Kup książkę Poleć książkę

Ź

Kup książkę Poleć książkę

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

Wzorce implementacyjne

Przy nauce programowania warto uczyć się na cudzych błędach. Programiści tworzący aplikację codziennie natykają 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 dobierać 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!

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**

Poznaj sprawdzone wzorce, które ulepszą Twoje oprogramowanie!

sięgnij po WIĘCEJ

KOD KORZYŚCI

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
http://helion.pl

ISBN 978-83-246-8644-5

Cena 39,90 zł

Informatyka w najlepszym wydaniu