KANON INFORMATYKI

Refaktoryzacja

Ulepszanie struktury istniejącego kodu

PIERWSZY PODRĘCZNIK TAK GRUNTOWNIE I PRZEJRZYŚCIE WYJAŚNIAJĄCY NAJLEPSZE PRAKTYKI ORAZ TECHNIKI REFAKTORYZACJI!

- Jak identyfikować błędy i problemy z istniejącym kodem?
- Jak poprawiać spójność, czytelność i wydajność kodu?
- Jak przekształcać kod bez ryzyka wprowadzania błędów?
- Jak skutecznie wykorzystywać przekształcenia refaktoryzacyjne?

Martin Fowler, Kent Beck,
John Brant, William Opdyke, Don Roberts

ADDISON-WESLEY

Helion

helion.pl

Idź do

- Spis treści
- Przykładowy rozdział
- Skorowidz

Katalog książek

- Katalog online
- Zamów drukowany katalog

Twój koszyk

- Dodaj do koszyka

Cennik i informacje

- Zamów informacje o nowościach
- Zamów cennik

Czytelnia

- Fragmenty książek online

Refaktoryzacja. Ulepszanie struktury istniejącego kodu

Autorzy: Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts
Tłumaczenie: Justyna Walkowska
ISBN: 978-83-246-3243-5
Tytuł oryginału: Refactoring: Improving the Design of Existing Code
Format: 172×245, stron: 384

Pierwszy podręcznik tak gruntownie i przejrzyście wyjaśniający najlepsze praktyki oraz techniki refaktoryzacji!

- Jak identyfikować błędy i problemy z istniejącym kodem?
- Jak poprawiać spójność, czytelność i wydajność kodu?
- Jak przekształcać kod bez ryzyka wprowadzania błędów?
- Jak skutecznie wykorzystywać przekształcenia refaktoryzacyjne?

Jak ryzykowne jest grzebanie w kodzie – wszyscy doskonale wiemy. Im głębiej sięgasz… tym więcej pojawia się nowych problemów i jeszcze więcej rzeczy wymaga zmian. A nieustanne „poprawianie" działającego kodu może w końcu doprowadzić do powstania trudno wykrywalnych, krytycznych błędów. Jednak co zrobić, jeśli "odziedziczymy" nieefektywny, trudny w utrzymaniu i rozszerzaniu program? Jak poprawić jego strukturalną spójność i wydajność? Wypracowywane latami przez najlepszych ekspertów techniki refaktoryzacji, czyli ulepszania projektu istniejącego kodu, są dziś sprawdzonymi rozwiązaniami, zapewniającymi jego trwałą czytelność i możliwość efektywnego rozwoju. Opracowane głównie na potrzeby frameworków, są obecnie narzędziem wykorzystywanym dla całego procesu produkcji oprogramowania. Jednak dla wielu programistów proces refaktoryzacji pozostaje wiedzą tajemną, bo jak dotąd żaden podręcznik nie przedstawił używanych przy tym technik w praktycznej, łatwej do wykorzystania formie. A przecież przeprowadzona błędnie lub w zbytnim pośpiechu refaktoryzacja zamiast ulepszenia kodu może kosztować nas dodatkowe dni lub całe tygodnie stresującej pracy nad programem.

Oto podręcznik, w którym słynny mentor i programistyczny guru Martin Fowler wraz z kilkoma innymi znanymi programistami podejmują się pierwszego tak gruntownego i przejrzystego objaśnienia technik związanych ze skutecznym procesem refaktoryzacji. Książka ta przedstawia zasady i najlepsze praktyki refaktoryzacyjne oraz zawiera wskazówki na temat tego, kiedy i jak zacząć ingerować w kod. Znajdziesz tu wyczerpujący katalog siedemdziesięciu przekształceń refaktoryzacyjnych. Każdemu z nich towarzyszą wskazówki dotyczące możliwości wykorzystania, instrukcja opisująca kolejne kroki oraz przykład. Ten podręcznik pokaże Ci zatem, jak przekształcać kod w sposób kontrolowany i efektywny, jak refaktoryzować go bez wprowadzania błędów, konsekwentnie ulepszając jego strukturę, oraz jak skutecznie go testować. Choć przedstawione w książce przykłady zostały napisane w języku Java, idee te znajdą zastosowanie w każdym innym języku obiektowym. Ponadto w opisach części przekształceń dodano uwagi związane z ich stosowaniem w innych językach.

Poznaj sprawdzone techniki ulepszania istniejącego kodu!

Kontakt

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 32 230 98 63
e-mail: helion@helion.pl

© Helion 1991–2011

Spis treści

1

Refaktoryzacja: pierwszy przykład

Od czego zacząć książkę na temat refaktoryzacji? Tradycyjne podejście nakazuje wyjść od historii i ogólnych zasad. Gdy ktoś w ten sposób zaczyna wystąpienie na konferencji, robię się senny. Mój umysł powoli odpływa. W tle uruchamia mi się wątek o niskim priorytecie, który co jakiś czas sprawdza, czy mówca wreszcie przeszedł do pierwszego przykładu. Przykłady natychmiast stawiają mnie na nogi, gdyż to właśnie one pozwalają mi zrozumieć, o co w ogóle chodzi. Przy przedstawianiu zasad łatwo o uogólnienia, co utrudnia zrozumienie, jak faktycznie stosować omawiane rozwiązanie. Przykład pozwala w pełniejszy sposób zrozumieć temat.

W związku z powyższym postanowiłem zacząć tę książkę od przykładu przekształcenia refaktoryzacyjnego. W trakcie jego omawiania wyjaśnię, jaki jest cel procesu refaktoryzacji i na czym on polega. Dopiero wtedy przejdę do tradycyjnego wstępu, w którym przedstawię podstawowe zasady i teorię.

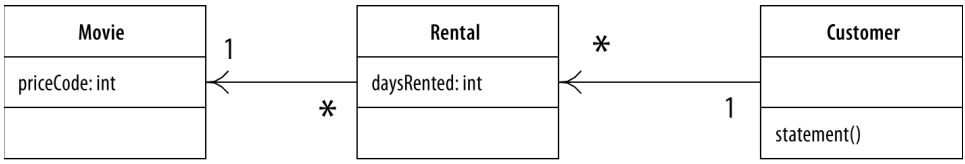
Z wyborem pierwszego przykładu wiąże się jednak pewien problem. Jeśli przedstawię rozbudowany program, opis jego samego i refaktoryzacji będzie zbyt skomplikowany dla Czytelnika. Okazuje się, że nawet umiarkowanie skomplikowany przykład wymaga ponad setki stron! Natomiast gdy wybiorę program na tyle niewielki, by zmieścić opis na paru stronach, trudno będzie uargumentować zasadność przekształceń.

Znalazłem się zatem w klasycznym, beznadziejnym położeniu osoby, która chce przedstawić techniki przydatne w przypadku dużych, „przemysłowych" programów. Ostatecznie zdecydowałem się na dość krótki przykład. Refaktoryzacja programu tej wielkości mija się z celem. Jeśli jednak pokazany tu kod jest częścią większego systemu, wysiłek włożony w refaktoryzację szybko się zwróci. Spróbuj więc wyobrazić sobie kod z przykładu w kontekście o wiele większego systemu.

Punkt wyjścia

Przykładowy program jest bardzo prosty. Jego zadaniem jest obliczenie i wyświetlenie podsumowania należności klienta wypożyczalni wideo. Program pobiera dane na temat wybranych filmów i czasu, na który zostały wypożyczone. Program sam rozpoznaje typ filmu. Istnieją trzy typy filmów: standardowe, dla dzieci oraz nowości. Poza wyliczaniem opłaty system wyznacza także punkty stałego klienta, których liczba zależy od tego, czy film należy do kategorii nowości.

Poszczególne elementy są reprezentowane przez różne klasy, widoczne na rysunku 1.1.



Rysunek 1.1. *Diagram klas przedstawiający wyjściowe klasy. Uwzględnia on tylko najważniejsze elementy. Zastosowana została notacja UML [Fowler, UML]*

Poniżej przedstawiam także kod każdej z klas.

Klasa Movie (film)

`Movie` to prosta klasa pełniąca rolę kontenera danych.

```
public class Movie {       //film

  public static final int  CHILDRENS = 2;    //dla dzieci
  public static final int  REGULAR = 0;      //standardowy
  public static final int  NEW_RELEASE = 1; //nowość

  private String _title;   //tytuł
  private int _priceCode;  //kod ceny

  public Movie(String title, int priceCode) {
      _title = title;
      _priceCode = priceCode;
  }

  public int getPriceCode() {
      return _priceCode;
  }

  public void setPriceCode(int arg) {
    _priceCode = arg;
  }

  public String getTitle (){
      return _title;
  };
}
```

Klasa Rental (pozycja wypożyczona)

Klasa `Rental` reprezentuje zdarzenie wypożyczenia filmu przez klienta.

```
class Rental {             //wypożyczenie — pozycja wypożyczona
  private Movie _movie;    //film
  private int _daysRented; //liczba dni
```

```
  public Rental(Movie movie, int daysRented) {
    _movie = movie;
    _daysRented = daysRented;
  }
  public int getDaysRented() {
    return _daysRented;
  }
  public Movie getMovie() {
    return _movie;
  }
}
```

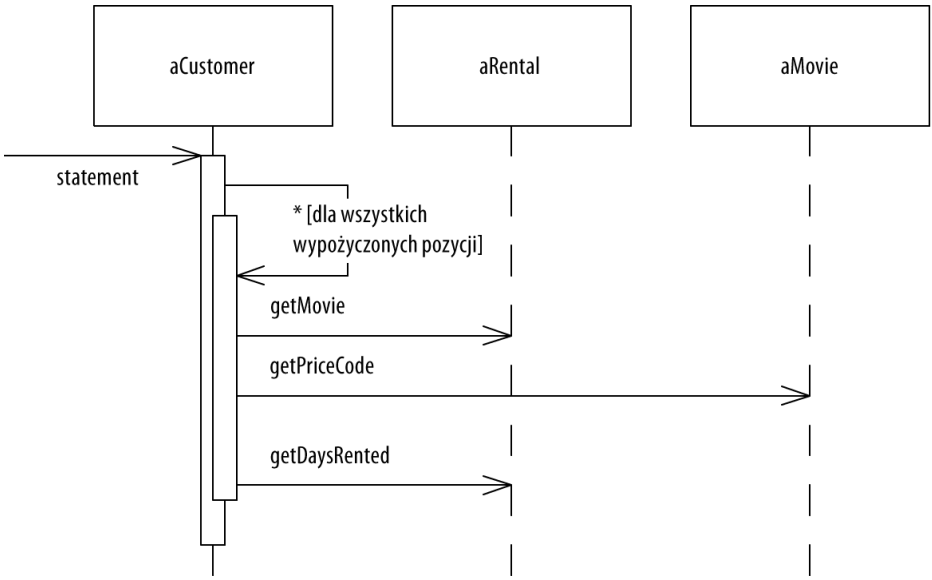
Klasa Customer (klient)

Klasa `Customer` reprezentuje klienta wypożyczalni. Tak jak wcześniejsze klasy, zawiera dane i metody dostępowe:

```
class Customer {                          //klient
  private String _name;                   //nazwisko
  private Vector _rentals = new Vector(); //lista wypożyczonych pozycji

  public Customer (String name){
      _name = name;
  };

  public void addRental(Rental arg) {
      _rentals.addElement(arg);
  }
  public String getName (){
      return _name;
  };
```

Dodatkowo klasa `Customer` posiada metodę generującą podsumowanie. Rysunek 1.2 przedstawia diagram interakcji tej metody. Kod metody został przedstawiony poniżej.

```
public String statement() {                       //podsumowanie
  double totalAmount = 0;                         //należność całkowita
  int frequentRenterPoints = 0;                   //punkty stałego klienta
  Enumeration rentals = _rentals.elements();      //wypożyczone pozycje
  String result = "Lista pozycji wypożyczonych przez klienta " + getName() + "\n";
  while (rentals.hasMoreElements()) {
    double thisAmount = 0;                        //należność za bieżącą pozycję
    Rental each = (Rental) rentals.nextElement(); //bieżąca pozycja

    //określenie opłaty za każdą z pozycji
    switch (each.getMovie().getPriceCode()) {
      case Movie.REGULAR:
        thisAmount += 2;
        if (each.getDaysRented() > 2)             //pobranie liczby dni
          thisAmount += (each.getDaysRented() - 2) * 1.5;
        break;
      case Movie.NEW_RELEASE:
```



Rysunek 1.2. *Diagram interakcji metody statement, generującej podsumowanie dla danego klienta*

```
        thisAmount += each.getDaysRented() * 3;
        break;
      case Movie.CHILDRENS:
        thisAmount += 1.5;
        if (each.getDaysRented() > 3)
          thisAmount += (each.getDaysRented() - 3) * 1.5;
        break;

    }

    //dodanie punktów stałego klienta
    frequentRenterPoints ++;
    //dodanie punktów za wypożyczenie nowości na więcej niż 1 dzień
    if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE)
      && each.getDaysRented() > 1) frequentRenterPoints ++;

    //zebranie danych na temat bieżącej pozycji
    result += "\t" + each.getMovie().getTitle()+ "\t" +
      String.valueOf(thisAmount) + "\n";
    totalAmount += thisAmount;

  }
  //linie końcowe
  result += "Należność wynosi " + String.valueOf(totalAmount) + "\n";
  result += "Klient otrzymał " + String.valueOf(frequentRenterPoints) + " punktów
  ↪stałego klienta";
  return result;

}
```

Uwagi na temat przykładowego programu

Co sądzisz o powyższym programie? Ja powiedziałbym, że jest niezbyt dobrze zaprojektowany i z pewnością nie jest obiektowy. W przypadku tak krótkiego programu nie ma to większego znaczenia. Krótkie programy mogą być brzydkie. Jeśli jednak ten fragment kodu jest reprezentatywną próbką większego systemu — mamy problem. Metoda `statement` w klasie `Customer` jest zbyt długa i wykonuje zbyt wiele operacji, z których znaczna część powinna być realizowana przez inne klasy.

Pomimo to program działa. Czy czepiam się jedynie estetyki? Owszem, tak właśnie jest, jednak tylko do chwili, w której będziemy chcieli wprowadzić zmiany w kodzie. Kompilator nie wybrzydza, wszystko mu jedno, czy kod jest elegancki, czy nie. Zmiany w programie wprowadzają wszak ludzie, a dla nich kwestia ta ma znaczenie. Źle zaprojektowany system trudno zmieniać i rozwijać, przede wszystkim dlatego, że niełatwo zorientować się, w którym miejscu należy wprowadzić zmiany. Jeśli trudno rozpoznać, który fragment wymaga zmian, znacznie wzrasta prawdopodobieństwo pomyłki i popełnienia błędów przez programistę.

Załóżmy teraz, że mamy wprowadzić zmianę na prośbę użytkowników systemu. Podsumowanie ma być generowane w języku HTML, by można było prezentować je na stronie internetowej. Jaki jest zasięg takiej zmiany? Okazuje się, że właściwie nie ma możliwości wykorzystania metody `statement`. Konieczne jest zdefiniowanie nowej metody, duplikującej spore fragmenty kodu. Z drugiej strony nie jest to takie uciążliwe. Możemy przecież skopiować metodę i zmienić odpowiednie fragmenty.

A co, jeśli zmienią się zasady naliczania opłat? Konieczne będzie wprowadzenie zmian zarówno w metodzie `statement`, jak i w `htmlStatement`. Programowanie metodą „kopiuj-wklej" okazuje się problematyczne właśnie w sytuacji, gdy musimy coś zmienić w wynikowym kodzie. Jeśli piszesz program, który prawdopodobnie nie będzie się zmieniał, możesz z powodzeniem stosować tę metodę. Jeśli jednak kod ma być długowieczny i może ulegać zmianom, radzę wystrzegać się tego rozwiązania.

Wyobraź sobie teraz kolejną zmianę. Użytkownicy chcą wprowadzić nowy sposób klasyfikacji filmów, ale jeszcze nie podjęli decyzji co do możliwych typów. Rozważają kilka możliwości, z których każda będzie miała inny wpływ na sposób naliczania opłat oraz punktów stałego klienta. Doświadczony programista rozumie, że niezależnie od „ostatecznego" wyboru użytkownika pewne jest tylko to, że w ciągu paru miesięcy decyzja znowu ulegnie zmianie.

Zmiany związane z zasadami klasyfikacji i naliczania opłat trzeba będzie wprowadzić w metodzie `statement`. Po skopiowaniu jej kodu do `htmlStatement` musimy zadbać o to, by analogiczne poprawki wprowadzić w obu miejscach. W miarę komplikowania się ustalanych przez użytkowników zasad coraz trudniej będzie wprowadzać zmiany bez ryzyka popełnienia błędu.

Możesz próbować zmieniać program tak, by zasięg zmian był jak najmniejszy. Zgodnie ze starym porzekadłem inżynierów: „nie naprawiaj tego, co się nie popsuło". Program może się nie popsuł, ale na pewno nie jest zdrowy. Utrudnia Twoje życie, ponieważ wprowadzanie żądanych przez użytkowników zmian staje się coraz bardziej złożone i męczące. To właśnie pole do popisu dla refaktoryzacji.

Uwaga Jeśli musisz wprowadzić do programu nową funkcjonalność, a struktura kodu nie pozwala na wygodne jej dodanie, zacznij od przeprowadzenia refaktoryzacji, która ułatwi rozszerzanie. Dopiero wtedy dodaj nową funkcjonalność.

Pierwszy krok refaktoryzacji

Przystępując do refaktoryzacji, zawsze zaczynam od tego samego kroku. Tworzę solidny zestaw testów dla analizowanej sekcji kodu. Testy mają kluczowe znaczenie, gdyż nawet przy ostrożnym wprowadzaniu uznanych i bezpiecznych przekształceń mogę się pomylić i doprowadzić do powstania błędów. Błądzić jest rzeczą ludzką, dlatego rozsądnie jest wesprzeć się testami.

Jako że metoda `statement` zwraca łańcuch znaków, tworzę paru klientów, każdemu z nich przypisuję kilka wypożyczonych pozycji różnego typu, po czym generuję treść poszczególnych podsumowań. Następnie porównuję wynikowe łańcuchy znaków z ręcznie wprowadzonymi łańcuchami referencyjnymi. Testy są uruchamiane jednym poleceniem z linii komend. Ich wykonanie trwa tylko parę sekund, ale zobaczysz, że uruchamiam je naprawdę często.

Ważnym elementem testów jest sposób raportowania wyników. Jeśli wszystko jest w porządku (łańcuchy znaków są identyczne), wystarczy informacja „OK". Jeśli jednak któryś z wyników zwróconych przez `statement` odbiega od idealnego, wyświetlana jest lista linii, które nie przeszły testów. Testy powinny być samosprawdzalne — porównywanie zwróconych wyników z wartościami oczekiwanymi jest stratą czasu programisty.

Podczas refaktoryzacji polegam na testach. Podczas przekształcania kodu zamierzam zaufać im w kwestii tego, czy wprowadziłem jakiś błąd. W związku z tym stworzenie dobrych, kompletnych testów jest kluczowe. Warto poświęcić czas na ich napisanie, ponieważ zapewniają one bezpieczeństwo podczas wprowadzania zmian w kodzie. Testy mają tak istotne znaczenie dla procesu refaktoryzacji, że zdecydowałem się poświęcić im cały rozdział 4.

Uwaga Przed przystąpieniem do refaktoryzacji upewnij się, że masz solidny pakiet testów. Testy muszą być samosprawdzalne.

Dekompozycja i redystrybucja metody statement

Elementem, który od początku nie daje mi spokoju, jest zdecydowanie zbyt długa metoda `statement`. Moim celem będzie jej dekompozycja (rozbicie) na mniejsze części. Mniejsze fragmenty z reguły ułatwiają zarządzanie kodem. Łatwiej nad nimi pracować i przenosić je w inne miejsca.

Pierwsza faza przekształceń refaktoryzacyjnych w tym rozdziale ma na celu rozbicie długiej metody na mniejsze fragmenty oraz przeniesienie ich do bardziej odpowiednich klas. Chcę sprawić, by możliwe było napisanie metody generującej podsumowanie w formacie HTML przy o wiele mniejszej duplikacji kodu.

Zacznę od wyszukania spójnego logicznie fragmentu kodu i użycia Ekstrakcji Metody (s. 94). Oczywistym kandydatem jest instrukcja `switch`. Przeniesienie tego fragmentu do własnej metody może okazać się korzystne.

Podczas ekstrakcji metody, podobnie jak w przypadku każdego innego przekształcenia refaktoryzacyjnego, powinienem mieć świadomość, co mogę zepsuć. Zła decyzja może doprowadzić do powstania błędów. Zanim przystąpię do przekształcania kodu, muszę się zastanowić, jak zrobić to bezpiecznie. Ponieważ wykonywałem to przekształcenie wielokrotnie, wypisałem odpowiednie kroki w katalogu przekształceń.

Po pierwsze, muszę zwrócić uwagę na wszelkie zmienne lokalne, czyli zmienne zdefiniowane wewnątrz metody oraz jej parametry. Badany fragment zawiera dwie takie zmienne: `each` oraz `thisAmount`, przy czym `each` nie jest modyfikowane przez dany fragment kodu, w przeciwieństwie do `thisAmount`. Zmienne niemodyfikowane mogą zostać przekazane nowej metodzie w postaci parametrów. W przypadku zmiennych modyfikowanych należy zachować większą

ostrożność. Jeśli jest tylko jedna taka zmienna, to metoda może ją zwracać. Zmienna `thisAmount` jest inicjalizowana wartością 0 przy każdym przebiegu pętli i nie jest zmieniana przed wykonaniem kodu `switch`. Oznacza to, że mogę po prostu przypisać do niej wynik zwracany przez nową metodę.

Kolejne dwie strony przedstawiają kod przed refaktoryzacją i po niej. Kod „przed" poniżej, kod „po" — na następnej stronie. Kod wydobyty (wyekstrahowany) z oryginalnej metody oraz wszelkie zmiany, które uznałem za potencjalnie nieoczywiste, zostały pogrubione. W dalszej części rozdziału będę trzymał się tej konwencji prezentacji przekształceń.

```
public String statement() {                    //podsumowanie
  double totalAmount = 0;                      //należność całkowita
  int frequentRenterPoints = 0;                //punkty stałego klienta
  Enumeration rentals = _rentals.elements(); //wypożyczone pozycje
  String result = "Lista pozycji wypożyczonych przez klienta " + getName() + "\n";
  while (rentals.hasMoreElements()) {
    double thisAmount = 0;                          //należność za bieżącą pozycję
    Rental each = (Rental) rentals.nextElement(); //bieżąca pozycja

    //określenie opłaty za każdą z pozycji
    switch (each.getMovie().getPriceCode()) {
      case Movie.REGULAR:
        thisAmount += 2;
        if (each.getDaysRented() > 2) //pobranie liczby dni
          thisAmount += (each.getDaysRented() - 2) * 1.5;
        break;
      case Movie.NEW_RELEASE:
        thisAmount += each.getDaysRented() * 3;
        break;
      case Movie.CHILDRENS:
        thisAmount += 1.5;
        if (each.getDaysRented() > 3)
          thisAmount += (each.getDaysRented() - 3) * 1.5;
        break;

    }

    //dodanie punktów stałego klienta
    frequentRenterPoints ++;
    //dodanie punktów za wypożyczenie nowości na więcej niż 1 dzień
    if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE)
      && each.getDaysRented() > 1) frequentRenterPoints ++;

    //zebranie danych na temat bieżącej pozycji
    result += "\t" + each.getMovie().getTitle()+ "\t" + String.valueOf(thisAmount) + "\n";
    totalAmount += thisAmount;

  }
  //linie końcowe
  result += "Należność wynosi " + String.valueOf(totalAmount) + "\n";
  result += "Klient otrzymał " + String.valueOf(frequentRenterPoints) + " punktów
stałego klienta";
  return result;

}
```

```
public String statement() {                    //podsumowanie
  double totalAmount = 0;                      //należność całkowita
  int frequentRenterPoints = 0;                //punkty stałego klienta
  Enumeration rentals = _rentals.elements(); //wypożyczone pozycje
  String result = "Lista pozycji wypożyczonych przez klienta " + getName() + "\n";
  while (rentals.hasMoreElements()) {
    double thisAmount = 0;
    Rental each = (Rental) rentals.nextElement();

    thisAmount = amountFor(each); //należność za bieżącą pozycję
    //dodanie punktów stałego klienta
    frequentRenterPoints ++;
    //dodanie punktów za wypożyczenie nowości na więcej niż 1 dzień
    if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
      each.getDaysRented() > 1) frequentRenterPoints ++;

    //zebranie danych na temat bieżącej pozycji
    result += "\t" + each.getMovie().getTitle()+ "\t" +
      String.valueOf(thisAmount) + "\n";
    totalAmount += thisAmount;
  }
  //linie końcowe
  result +=  " Należność wynosi " + String.valueOf(totalAmount) + "\n";
  result += " Klient otrzymał " + String.valueOf(frequentRenterPoints) + " punktów
  ↪stałego klienta ";
  return result;
}
private int amountFor(Rental each) { //należność za wypożyczoną pozycję
  int thisAmount = 0;
  switch (each.getMovie().getPriceCode()) {
    case Movie.REGULAR:
      thisAmount += 2;
      if (each.getDaysRented() > 2)
        thisAmount += (each.getDaysRented() - 2) * 1.5;
      break;
    case Movie.NEW_RELEASE:
      thisAmount += each.getDaysRented() * 3;
      break;
    case Movie.CHILDRENS:
      thisAmount += 1.5;
      if (each.getDaysRented() > 3)
        thisAmount += (each.getDaysRented() - 3) * 1.5;
      break;
    }
    return thisAmount;
}
```

Zawsze po wprowadzeniu tego typu zmiany kompiluję kod i uruchamiam testy. Tym razem nie wszystko poszło zgodnie z planem — ku mojemu zdumieniu testy zakończyły się niepowodzeniem. Zrozumienie, co się stało, zajęło mi kilka sekund. Okazało się, że zagapiwszy się, zadeklarowałem typ zwracany przez `amountFor` jako `int`, podczas gdy powinien to być typ `double`:

```
private double amountFor(Rental each) { //należność za wypożyczoną pozycję
  double thisAmount = 0;
  switch (each.getMovie().getPriceCode()) {
```

```
    case Movie.REGULAR:
      thisAmount += 2;
      if (each.getDaysRented() > 2)
        thisAmount += (each.getDaysRented() - 2) * 1.5;
      break;
    case Movie.NEW_RELEASE:
      thisAmount += each.getDaysRented() * 3;
      break;
    case Movie.CHILDRENS:
      thisAmount += 1.5;
      if (each.getDaysRented() > 3)
        thisAmount += (each.getDaysRented() - 3) * 1.5;
      break;
  }
   return thisAmount;
}
```

Tego typu błędy zdarzają mi się dość często. Ich wyśledzenie może zająć niemało czasu. W tym wypadku Java przekształci `double` na `int` bez zgłaszania problemów, zaokrąglając wynik [Java Spec]. Szczęśliwie tym razem udało się szybko odnaleźć przyczynę błędu, głównie dlatego, że wprowadzona zmiana była niewielka, a testy okazały się kompletne. Przypadkiem pokazałem sedno procesu refaktoryzacji. Jeśli kolejne zmiany są niewielkie, błędy można dość łatwo znaleźć. Nawet w przypadku nieuwagi takiej jak moja nie marnuje się ogromnej ilości czasu na poszukiwanie błędów.

Uwaga Proces refaktoryzacji polega na wprowadzaniu niewielkich zmian w kolejnych krokach. Dzięki temu, gdy się pomylisz, łatwo odnajdziesz błąd.

Ponieważ pracuję w Javie, musiałem przyjrzeć się kwestii zmiennych lokalnych. Jednak dla niektórych języków istnieją narzędzia, które automatycznie przeprowadzają tego typu przekształcenia. Na przykład w języku Smalltalk można skorzystać z narzędzia Refactoring Browser, które pozwala zaznaczyć odpowiedni fragment kodu, wybrać z menu opcję „Ekstrakcja Metody", wpisać nazwę nowej metody — i już. Co więcej, narzędzie to nie popełnia błędów podobnych do mojego. Nie mogę się doczekać wersji dla Javy![1]

Skoro podzieliłem wyjściową metodę na mniejsze fragmenty, mogę pracować osobno nad każdym z nich. Nie podobają mi się nazwy niektórych zmiennych w `amountFor`. To dobry moment na ich zmianę.

Oto oryginalny kod:

```
private double amountFor(Rental each) { //należność za wypożyczoną pozycję
  double thisAmount = 0;
  switch (each.getMovie().getPriceCode()) {
    case Movie.REGULAR:
      thisAmount += 2;
      if (each.getDaysRented() > 2)
        thisAmount += (each.getDaysRented() - 2) * 1.5;
      break;
    case Movie.NEW_RELEASE:
      thisAmount += each.getDaysRented() * 3;
```

[1] „Ekstrakcję Metody", a także kilka innych przekształceń refaktoryzacyjnych, można przeprowadzić na przykład w środowisku programistycznym Eclipse — *przyp. tłum.*

```
      break;
    case Movie.CHILDRENS:
      thisAmount += 1.5;
      if (each.getDaysRented() > 3)
        thisAmount += (each.getDaysRented() - 3) * 1.5;
      break;
  }
  return thisAmount;
}
```

Oto kod po wprowadzeniu zmian. Parametr `each` („każdy”) otrzymał nazwę `aRental` („pozycja wypożyczona”), a zmienna `thisAmount` („bieżąca należność”) zmieniła nazwę na `result` („wynik”):

```
private double amountFor(Rental aRental) { //należność za wypożyczoną pozycję
  double result = 0;
  switch (aRental.getMovie().getPriceCode()) {
    case Movie.REGULAR:
      result += 2;
      if (aRental.getDaysRented() > 2)
        result += (aRental.getDaysRented() - 2) * 1.5;
      break;
    case Movie.NEW_RELEASE:
      result += aRental.getDaysRented() * 3;
      break;
    case Movie.CHILDRENS:
      result += 1.5;
      if (aRental.getDaysRented() > 3)
        result += (aRental.getDaysRented() - 3) * 1.5;
      break;
  }
   return result;
}
```

Po wprowadzeniu zmian kompiluję program i uruchamiam testy, by upewnić się, że nic nie popsułem.

Czy warto zmieniać nazwy zmiennych? Jak najbardziej. Dobrze napisany kod powinien być czytelny, a znaczące nazwy zmiennych są kluczem do zrozumiałego kodu. Nigdy nie wahaj się zmieniać nazw w celu poprawienia czytelności. Nie jest to trudne, a na dodatek można wspomóc się narzędziami. Pamiętaj:

Uwaga Nie jest sztuką napisanie kodu zrozumiałego dla komputera. Dobrzy programiści tworzą kod zrozumiały dla ludzi.

Kod, który komunikuje swój cel, jest bardzo wartościowy. Często przeprowadzam refaktoryzację podczas czytania kodu. Dzięki temu, w miarę jak zaczynam rozumieć działanie kodu, wprowadzam wywnioskowane informacje do jego treści, przez co mam pewność, że nie zapomnę, czego się nauczyłem.

Przeniesienie obliczeń związanych z wyznaczaniem należności

Patrząc na kod metody `amountFor`, zauważam, że korzysta ona z informacji zawartych w klasie `Rental`, natomiast nie potrzebuje żadnych danych z klasy `Customer`.

```
class Customer...
  private double amountFor(Rental aRental) { //należność za wypożyczoną pozycję
    double result = 0;
    switch (aRental.getMovie().getPriceCode()) {
      case Movie.REGULAR:
        result += 2;
        if (aRental.getDaysRented() > 2)
          result += (aRental.getDaysRented() - 2) * 1.5;
        break;
      case Movie.NEW_RELEASE:
        result += aRental.getDaysRented() * 3;
          break;
        case Movie.CHILDRENS:
        result += 1.5;
        if (aRental.getDaysRented() > 3)
          result += (aRental.getDaysRented() - 3) * 1.5;
        break;
  }
  return result;
}
```

W oparciu o te przesłanki podejrzewam, że metoda została zdefiniowana w złym obiekcie. W większości wypadków metoda powinna należeć do obiektu, z którego danych korzysta. Dlatego przeniosę ją do klasy `Rental`. Stosuję tu przekształcenie Przeniesienie Metody (s. 124). Metoda zostanie skopiowana do docelowej klasy, dopasowana do nowego otoczenia i skompilowana.

```
class Rental...
  double getCharge() { //podaj należność
    double result = 0;
    switch (getMovie().getPriceCode()) {
      case Movie.REGULAR:
        result += 2;
        if (getDaysRented() > 2)
          result += (getDaysRented() - 2) * 1.5;
        break;
      case Movie.NEW_RELEASE:
        result += getDaysRented() * 3;
          break;
      case Movie.CHILDRENS:
        result += 1.5;
        if (getDaysRented() > 3)
          result += (getDaysRented() - 3) * 1.5;
        break;
    }
    return result;
  }
```

W tym wypadku dopasowanie do nowego otoczenia oznacza usunięcie parametru. Dodatkowo po przeniesieniu zmieniłem nazwę.

Chcę teraz przetestować działanie nowej metody. W tym celu zmieniam ciało metody `amountFor` w klasie `Customer` tak, by delegowało wykonanie do nowej metody:

```
class Customer...
  private double amountFor(Rental aRental) { //należność za wypożyczoną pozycję
    return aRental.getCharge();
  }
```

Następnie kompiluję kod i uruchamiam testy, by upewnić się, że nic nie zepsułem.

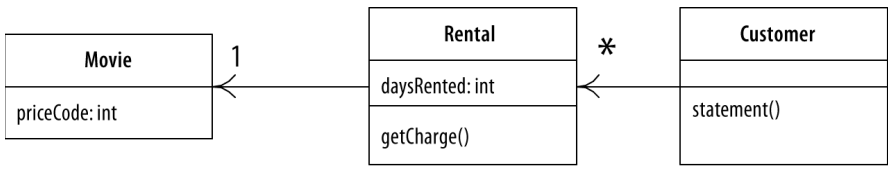
Następny krok to odnalezienie wszystkich odwołań do starej metody i zastąpienie ich wywołaniami nowej. Kod przed zmianami:

```
Class Customer...
  public String statement() {                    //podsumowanie
    double totalAmount = 0;                      //należność całkowita
    int frequentRenterPoints = 0;                //punkty stałego klienta
    Enumeration rentals = _rentals.elements(); //wypożyczone pozycje
    String result = "Lista pozycji wypożyczonych przez klienta " + getName() + "\n";
    while (rentals.hasMoreElements()) {
      double thisAmount = 0;
      Rental each = (Rental) rentals.nextElement();

      thisAmount = amountFor(each); //należność za bieżącą pozycję

      //dodanie punktów stałego klienta
      frequentRenterPoints ++;
      //dodanie punktów za wypożyczenie nowości na więcej niż 1 dzień
      if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
        each.getDaysRented() > 1) frequentRenterPoints ++;

      //zebranie danych na temat bieżącej pozycji
      result += "\t" + each.getMovie().getTitle()+ "\t" + String.valueOf(thisAmount) + "\n";
      totalAmount += thisAmount;

    }
     //linie końcowe
    result += "Należność wynosi " + String.valueOf(totalAmount) + "\n";
    result += "Klient otrzymał " + String.valueOf(frequentRenterPoints) + " punktów
    ↪stałego klienta";
    return result;

  }
```

Wykonanie tego kroku jest bardzo proste, gdyż metoda dopiero co została utworzona i jest używana tylko w jednym miejscu. W ogólnym przypadku należy przeszukać kod wszystkich klas. Kod po wprowadzeniu zmiany:

```
Class Customer...
  public String statement() {                    //podsumowanie
    double totalAmount = 0;                      //należność całkowita
    int frequentRenterPoints = 0;                //punkty stałego klienta
```

```
    Enumeration rentals = _rentals.elements(); //wypożyczone pozycje
    String result = "Lista pozycji wypożyczonych przez klienta " + getName() + "\n";
    while (rentals.hasMoreElements()) {
      double thisAmount = 0;
      Rental each = (Rental) rentals.nextElement();

      thisAmount = each.getCharge; //należność za bieżącą pozycję

      //dodanie punktów stałego klienta
      frequentRenterPoints ++;
      //dodanie punktów za wypożyczenie nowości na więcej niż 1 dzień
      if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
        each.getDaysRented() > 1) frequentRenterPoints ++;

      //zebranie danych na temat bieżącej pozycji
      result += "\t" + each.getMovie().getTitle()+ "\t" + String.valueOf(thisAmount) + "\n";
      totalAmount += thisAmount;

    }
    //linie końcowe
    result += "Należność wynosi " + String.valueOf(totalAmount) + "\n";
    result += "Klient otrzymał " + String.valueOf(frequentRenterPoints) + " punktów
    ↪stałego klienta";
    return result;

  }
```



Rysunek 1.3. *Diagram klas po przeniesieniu metody getCharge*

Kolejnym krokiem po wprowadzeniu opisanych zmian (rysunek 1.3) jest usunięcie starej metody. Kompilator poinformuje mnie, jeśli zapomnę o jakimś jej wywołaniu. Oczywiście przeprowadzę testy, by sprawdzić, czy wszystko się udało.

Zdarza się, że nie usuwam całkowicie starej metody, tylko pozostawiam ją w oryginalnej klasie w postaci delegującej wykonanie. Takie rozwiązanie jest uzasadnione, gdy metoda jest publiczna i nie chcę zmieniać interfejsu klasy.

W kodzie `Rental.getCharge` widzę potencjał na więcej zmian, jednak na razie wróćmy do `Customer.statement`.

```
public String statement() {                    //podsumowanie
  double totalAmount = 0;                      //należność całkowita
  int frequentRenterPoints = 0;                //punkty stałego klienta
  Enumeration rentals = _rentals.elements(); //wypożyczone pozycje
  String result = "Lista pozycji wypożyczonych przez klienta " + getName() + "\n";
  while (rentals.hasMoreElements()) {
    double thisAmount = 0;
    Rental each = (Rental) rentals.nextElement();

    thisAmount = each.getCharge; //należność za bieżącą pozycję
```

```
    //dodanie punktów stałego klienta
    frequentRenterPoints ++;
    //dodanie punktów za wypożyczenie nowości na więcej niż 1 dzień
    if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
      each.getDaysRented() > 1) frequentRenterPoints ++;

    //zebranie danych na temat bieżącej pozycji
    result += "\t" + each.getMovie().getTitle()+ "\t" + String.valueOf(thisAmount) + "\n";
    totalAmount += thisAmount;

  }
  //linie końcowe
  result += "Należność wynosi " + String.valueOf(totalAmount) + "\n";
  result += "Klient otrzymał " + String.valueOf(frequentRenterPoints) + " punktów
  ↪stałego klienta";
  return result;

}
```

Rzuca mi się w oczy, że zmienna `thisAmount` jest nadmiarowa. Otrzymuje ona wartość `each.charge`, która nie jest już później zmieniana. W związku z tym wyeliminuję ją, korzystając z przekształcenia Zastąpienie Zmiennej Tymczasowej Zapytaniem (s. 103).

```
public String statement() {                    //podsumowanie
  double totalAmount = 0;                      //należność całkowita
  int frequentRenterPoints = 0;                //punkty stałego klienta
  Enumeration rentals = _rentals.elements(); //wypożyczone pozycje
  String result = "Lista pozycji wypożyczonych przez klienta " + getName() + "\n";
  while (rentals.hasMoreElements()) {
    Rental each = (Rental) rentals.nextElement();

    //dodanie punktów stałego klienta
    frequentRenterPoints ++;
    //dodanie punktów za wypożyczenie nowości na więcej niż 1 dzień
    if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
      each.getDaysRented() > 1) frequentRenterPoints ++;

    //zebranie danych na temat bieżącej pozycji
    result += "\t" + each.getMovie().getTitle()+ "\t" + String.valueOf
     (each.getChange()) + "\n";
    totalAmount += each.getChange();

  }
  //linie końcowe
  result += "Należność wynosi " + String.valueOf(totalAmount) + "\n";
  result += "Klient otrzymał " + String.valueOf(frequentRenterPoints) + " punktów
  ↪stałego klienta";
  return result;

}
```

Po wprowadzeniu zmiany kompiluję kod i uruchamiam testy, by sprawdzić, czy niczego nie popsułem.

O ile to możliwe, staram się usuwać tego typu zmienne tymczasowe. Często prowadzą one do sytuacji, w której tam i z powrotem przekazywanych jest wiele parametrów, które w praktyce wcale nie są wymagane. Łatwo pogubić się w ich przeznaczeniu, a szczególnie uciążliwe stają się właśnie wewnątrz długich metod. Ceną za ich usunięcie może być pogorszenie wydajności — w powyższym kodzie należność jest wyznaczana dwukrotnie. Jednak ten element łatwo zoptymalizować wewnątrz klasy `Rental`, a optymalizacja zawsze jest łatwiejsza w przypadku eleganckiego, przemyślanego kodu. Wrócę do tego zagadnienia później, w części „Refaktoryzacja a wydajność" na stronie 61.

Ekstrakcja punktów stałego klienta

Zajmiemy się teraz punktami stałego klienta. Zasady ich naliczania różnią się w zależności od rodzaju filmu, są jednak mniej skomplikowane niż zasady wyznaczania należności. Rozsądne wydaje się przydzielenie tego zadania klasie `Rental`. Zaczniemy od Ekstrakcji Metody (s. 94). Pogrubiony fragment kodu zostanie wydobyty i przeniesiony do nowej metody.

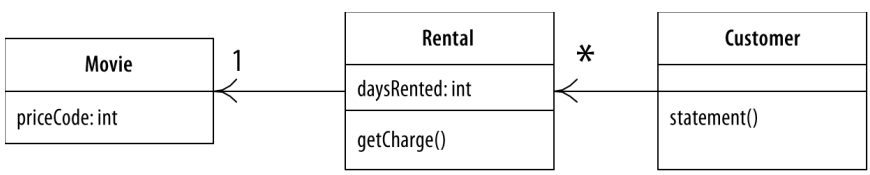
```
public String statement() {                    //podsumowanie
  double totalAmount = 0;                      //należność całkowita
  int frequentRenterPoints = 0;                //punkty stałego klienta
  Enumeration rentals = _rentals.elements(); //wypożyczone pozycje
  String result = "Lista pozycji wypożyczonych przez klienta " + getName() + "\n";
  while (rentals.hasMoreElements()) {
    Rental each = (Rental) rentals.nextElement();

    //dodanie punktów stałego klienta
    frequentRenterPoints ++;
    //dodanie punktów za wypożyczenie nowości na więcej niż 1 dzień
    if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
      each.getDaysRented() > 1) frequentRenterPoints ++;

    //zebranie danych na temat bieżącej pozycji
    result += "\t" + each.getMovie().getTitle()+ "\t" + String.valueOf(each.getChange()) +
    ➥"\n";
    totalAmount += each.getChange();

  }
  //linie końcowe
  result += "Należność wynosi " + String.valueOf(totalAmount) + "\n";
  result += "Klient otrzymał " + String.valueOf(frequentRenterPoints) + " punktów
  ➥stałego klienta";
  return result;

}
```

Podobnie jak w podczas ekstrakcji fragmentu odpowiedzialnego za wyznaczanie należności, musimy uważnie przyjrzeć się zmiennym lokalnym. Wykorzystywana jest zmienna `each`, do której będziemy mieli dostęp, gdyż reprezentuje ona obiekt klasy `Rental`, a w tej właśnie klasie definiujemy nową metodę. Kolejna zmienna lokalna to `frequentRenterPoints`. W tym wypadku ma ona określoną wartość przed wykonaniem kodu, nad którym pracujemy. Jej wartość nie jest jednak odczytywana w ciele metody, zatem nie musimy przekazywać jej jako parametru, o ile zastosujemy przypisanie zwiększające, a nie ustawiające wartość zmiennej.

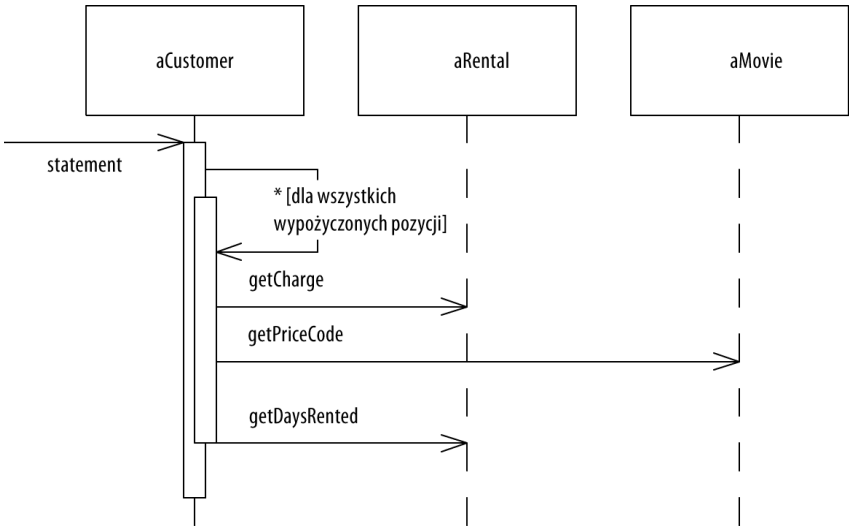
Przeprowadziłem ekstrakcję, skompilowałem program, uruchomiłem testy. Opisaną powyżej ekstrakcję można przeprowadzić w dwóch krokach. Krok pierwszy to utworzenie nowej metody wewnątrz klasy `Customer`, a drugi, poprzedzony testami, to przeniesienie jej do docelowej klasy `Rental`. Na koniec oczywiście należy wszystko ponownie przetestować.

```
class Customer...
  public String statement() {                    //podsumowanie
    double totalAmount = 0;                      //należność całkowita
    int frequentRenterPoints = 0;                //punkty stałego klienta
    Enumeration rentals = _rentals.elements(); //wypożyczone pozycje
    String result = "Lista pozycji wypożyczonych przez klienta " + getName() + "\n";
    while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();
      frequentRenterPoints += each.getFrequentRenterPoints();

       //zebranie danych na temat bieżącej pozycji
      result += "\t" + each.getMovie().getTitle()+ "\t" + String.valueOf(each.getCharge()) +
      ↪"\n";
      totalAmount += each.getCharge();
    }

    //linie końcowe
    result += "Należność wynosi " + String.valueOf(totalAmount) + "\n";
    result += "Klient otrzymał " + String.valueOf(frequentRenterPoints) + " punktów
    ↪stałego klienta";
    return result;
  }

class Rental...
  int getFrequentRenterPoints() { //podaj punkty stałego klienta
    if ((getMovie().getPriceCode() == Movie.NEW_RELEASE) && getDaysRented() > 1)
      return 2;
    else
      return 1;
  }
```

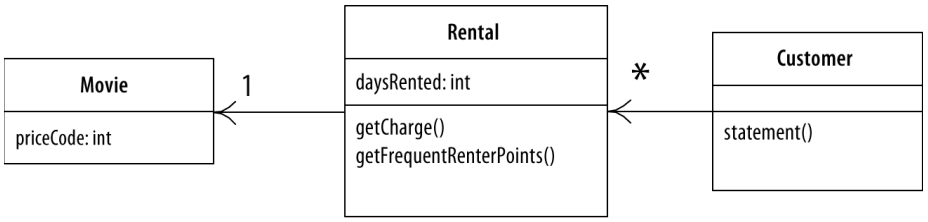
Podsumuję przeprowadzone zmiany za pomocą diagramów UML (rysunki 1.4 – 1.7). Diagramy po lewej przedstawiają stan „przed", po prawej — stan „po".



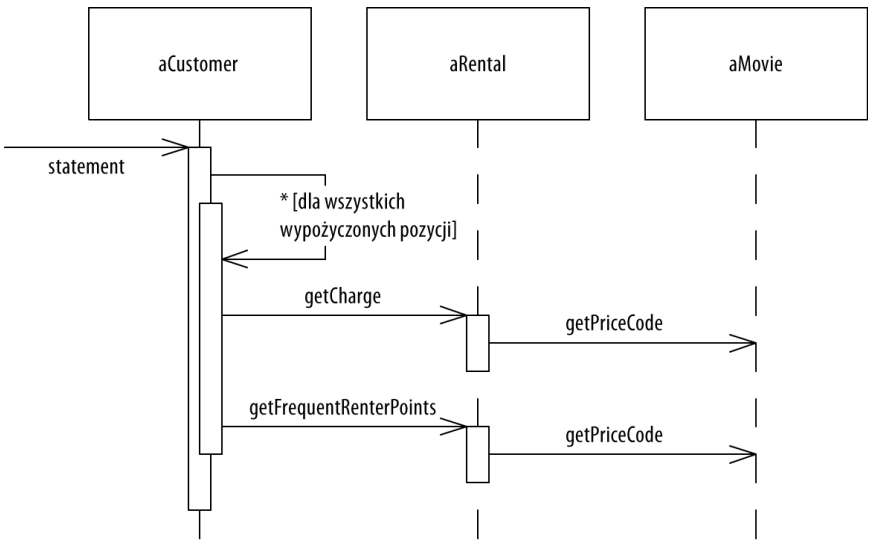
Rysunek 1.4. *Diagram klas przed ekstrakcją i przeniesieniem obliczania punktów stałego klienta*



Rysunek 1.5. *Diagram sekwencji przed ekstrakcją i przeniesieniem obliczania punktów stałego klienta*



Rysunek 1.6. *Diagram klas po ekstrakcji i przeniesieniu obliczania punktów stałego klienta*



Rysunek 1.7. *Diagram sekwencji po ekstrakcji i przeniesieniu obliczania punktów stałego klienta*

Usunięcie zmiennych tymczasowych

Już wcześniej sygnalizowałem, że zmienne tymczasowe mogą stanowić problem. Mają one sens tylko wewnątrz pewnej procedury, przez co zachęcają do tworzenia długich, złożonych procedur. W badanym fragmencie kodu mamy dwie takie zmienne służące do obliczenia całkowitej należności za pozycje wypożyczone przez danego klienta. Są one wykorzystywane zarówno w tekstowej wersji metody `statement`, jak i w wersji generującej kod HTML. Zastosuję przekształcenie Zastąpienie Zmiennej Tymczasowej Zapytaniem (s. 103). Zmienne `totalAmount` oraz `frequentRentalPoints` zostaną zastąpione wywołaniami metod, które są dostępne z wnętrza wszystkich metod klasy. Dzięki nim kod staje się bardziej przejrzysty, a metody krótsze:

```
class Customer...
  public String statement() {                    //podsumowanie
    double totalAmount = 0;                      //należność całkowita
    int frequentRenterPoints = 0;                //punkty stałego klienta
    Enumeration rentals = _rentals.elements(); //wypożyczone pozycje
    String result = "Lista pozycji wypożyczonych przez klienta " + getName() + "\n";
    while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();
      frequentRenterPoints += each.getFrequentRenterPoints();

       //zebranie danych na temat bieżącej pozycji
      result += "\t" + each.getMovie().getTitle()+ "\t" + String.valueOf(each.getChange()) +
      ↪"\n";
      totalAmount += each.getChange();
    }

    //linie końcowe
    result += "Należność wynosi " + String.valueOf(totalAmount) + "\n";
    result += "Klient otrzymał " + String.valueOf(frequentRenterPoints) + " punktów
    ↪stałego klienta";
    return result;
  }
```

Zacznę od zastąpienia `totalAmount` wywołaniem metody `getTotalCharge` klasy `Customer`:

```
class Customer...

  public String statement() {                    //podsumowanie
    int frequentRenterPoints = 0;                //punkty stałego klienta
    Enumeration rentals = _rentals.elements(); //wypożyczone pozycje
    String result = "Lista pozycji wypożyczonych przez klienta " + getName() + "\n";
    while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();
      frequentRenterPoints += each.getFrequentRenterPoints();

      //zebranie danych na temat bieżącej pozycji
      result += "\t" + each.getMovie().getTitle()+ "\t" + String.valueOf (each.getChange())
      ↪+ "\n";
    }

    //linie końcowe
    result += "Należność wynosi " + String.valueOf(getTotalCharge()) + "\n";
```

```
    result += "Klient otrzymał " + String.valueOf(frequentRenterPoints) + " punktów
    ↪stałego klienta";
    return result;
  }

  private double getTotalCharge() { //podaj całkowitą należność
    double result = 0;
    Enumeration rentals = _rentals.elements();
    while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();
      result += each.getCharge();
    }
    return result;
  }
```

Nie jest to najprostszy przykład zastosowania przekształcenia Zastąpienie Zmiennej Tymczasowej Zapytaniem (s. 103). Wartość zmiennej `totalAmount` była modyfikowana wewnątrz pętli, musiałem więc przekopiować pętlę do wnętrza nowej metody `getTotalCharge`.

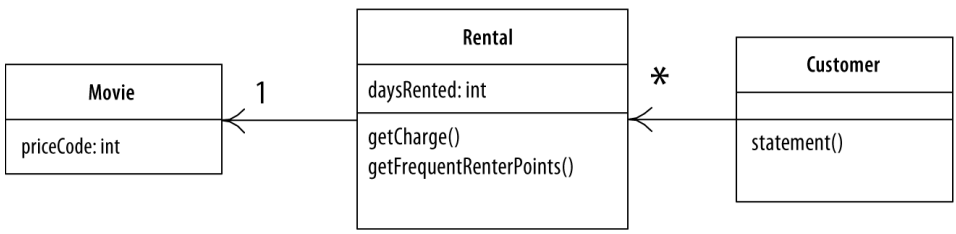
Po skompilowaniu i przetestowaniu przekształconego kodu wykonuję analogiczne operacje na zmiennej `frequentRenterPoints`:

```
class Customer...
  public String statement() {                  //podsumowanie
    int frequentRenterPoints = 0;              //punkty stałego klienta
    Enumeration rentals = _rentals.elements(); //wypożyczone pozycje
    String result = "Lista pozycji wypożyczonych przez klienta " + getName() + "\n";
    while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();
      frequentRenterPoints += each.getFrequentRenterPoints();

      //zebranie danych na temat bieżącej pozycji
      result += "\t" + each.getMovie().getTitle()+ "\t" + String.valueOf(each.getChange())
      ↪+ "\n";
    }

    //linie końcowe
    result += "Należność wynosi " + String.valueOf(getTotalCharge()) + "\n";
    result += "Klient otrzymał " + String.valueOf(frequentRenterPoints) + " punktów
    ↪stałego klienta";
    return result;
  }
```

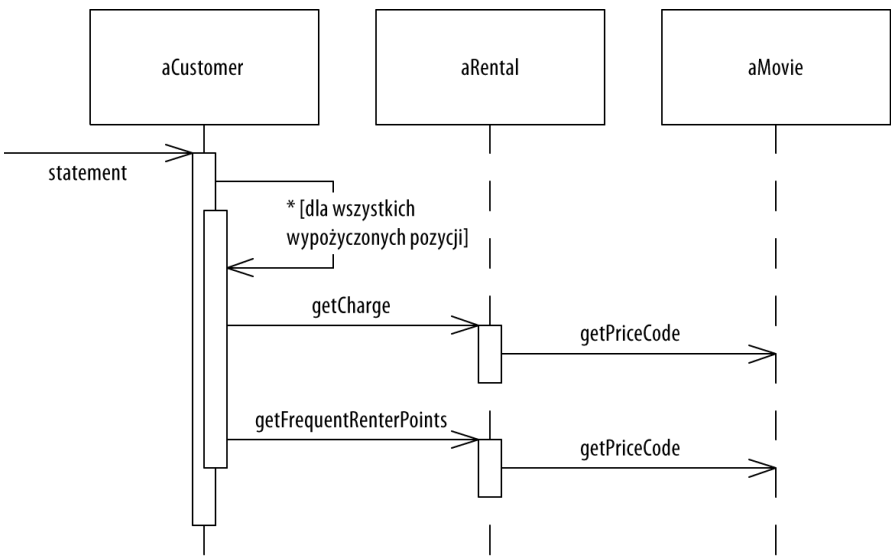
```
class Customer...
  public String statement() {                  //podsumowanie
    Enumeration rentals = _rentals.elements(); //wypożyczone pozycje
    String result = "Lista pozycji wypożyczonych przez klienta " + getName() + "\n";
    while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();

      //zebranie danych na temat bieżącej pozycji
      result += "\t" + each.getMovie().getTitle()+ "\t" + String.valueOf
      (each.getChange()) + "\n";
```

```
    }

    //linie końcowe
    result += "Należność wynosi " + String.valueOf(getTotalCharge()) + "\n";
    result += "Klient otrzymał " + String.valueOf(getTotalFrequentRenterPoints()) + "
    ↪punktów stałego klienta";
    return result;
  }

  private int getTotalFrequentRenterPoints(){ //podaj wynikową liczbę punktów stałego klienta
    int result = 0;
    Enumeration rentals = _rentals.elements();
    while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();
      result += each.getFrequentRenterPoints();
    }
    return result;
  }
```

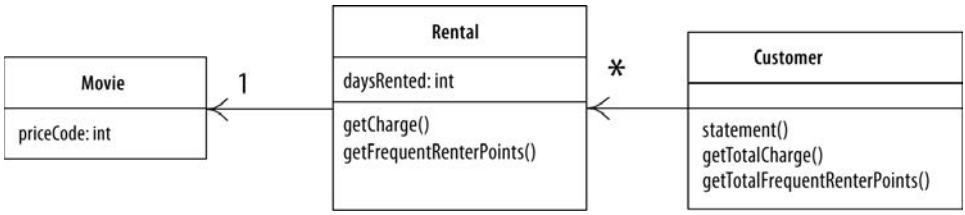
Rysunki 1.8 – 1.11 przedstawiają wprowadzane zmiany na diagramach klas i sekwencji.



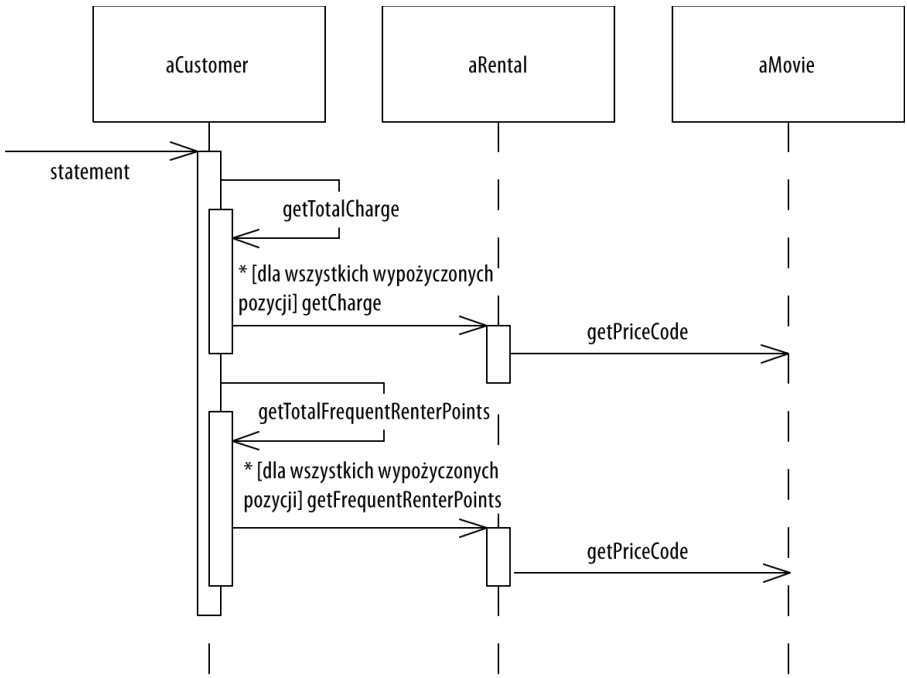
Rysunek 1.8. *Diagram klas przed ekstrakcją obliczania punktów stałego klienta*



Rysunek 1.9. *Diagram sekwencji przed ekstrakcją obliczania punktów stałego klienta*



Rysunek 1.10. *Diagram klas po ekstrakcji obliczania punktów stałego klienta*



Rysunek 1.11. *Diagram sekwencji po ekstrakcji obliczania punktów stałego klienta*

W tym miejscu warto na chwilę przystanąć i zastanowić się nad ostatnim przekształceniem refaktoryzacyjnym. Większość przekształceń zmniejsza długość kodu, tymczasem to spowodowało jego przyrost. Wynika to z faktu, że Java wymaga zastosowania wielu instrukcji podczas tworzenia pętli sumującej wartości. Prosta pętla sumująca, w której każdą instrukcję umieszczamy w osobnej linii, wymaga sześciu linii pomocniczych. To oczywiste dla programisty Javy — niemniej jednak liczba linii wzrasta.

Kolejny potencjalny problem to wydajność. Pierwotny kod wykonywał pętlę `while` tylko raz, nowy robi to aż trzy razy. Gdyby wykonanie pętli zajmowało dużo czasu, mogłaby ona obniżyć wydajność. Wielu programistów z tego właśnie powodu nie zdecydowałaby się na to przekształcenie. Warto jednak zwrócić uwagę na słowa „gdyby" oraz „mogłaby". Przed przeprowadzeniem analizy czasu wykonania nie jestem w stanie odpowiedzieć na pytanie, ile czasu trwa wykonanie pętli ani czy wpływa ona na wydajność systemu. Na etapie refaktoryzacji nie warto się tym martwić. Zajmiemy się tym podczas optymalizacji, jednak zanim do tego dojdzie, zmienimy kod w taki sposób, że optymalizacja stanie się o wiele łatwiejsza (więcej na ten temat na stronie 61).

Nowo utworzone metody zwracające wyniki obliczeń są dostępne dla wszystkich metod klasy `Customer`. Gdyby okazały się potrzebne także w innych klasach, można dodać je do interfejsu `Customer`. Gdyby metody te nie istniały, inne klasy musiałyby same pobierać dane związane z wypożyczonymi pozycjami i same definiować odpowiednie pętle. W złożonym systemie taka sytuacja spowodowałaby konieczność napisania i utrzymania o wiele większej ilości kodu.

W przypadku metody `htmlStatement` różnicę widać gołym okiem. Na chwilę zapomnę o refaktoryzacji i zajmę się dodawaniem nowych funkcjonalności. W tej chwili metoda `htmlStatement` może mieć następującą postać:

```
public String htmlStatement() {
  Enumeration rentals = _rentals.elements();
  String result = "<H1> Lista pozycji wypożyczonych przez klienta <EM>" + getName() +
  ↪"</EM></ H1><P>\n";
  while (rentals.hasMoreElements()) {
    Rental each = (Rental) rentals.nextElement();
    //zebranie danych na temat bieżącej pozycji
    result += each.getMovie().getTitle()+ ": " + String.valueOf(each.getCharge()) +
    ↪"<BR>\n";
  }
  //linie końcowe
  result +=  "<P>Należność wynosi <EM>" + String.valueOf(getTotalCharge()) +
  ↪"</EM><P>\n";
  result += " Klient otrzymał <EM>" + String.valueOf(getTotalFrequentRenterPoints()) +
  ↪"</EM> punktów stałego klienta<P>";
  return result;
}
```

Dzięki ekstrakcji obliczeń metoda `htmlStatement` może wykorzystać istniejący kod, oryginalnie znajdujący się wewnątrz metody `statement`. Rezygnacja z podejścia „kopiuj-wklej” powoduje, że w razie zmiany reguł muszę poprawić tylko jedno miejsce w kodzie. Jeśli pojawi się potrzeba generowania jeszcze innego rodzaju podsumowania, napisanie nowej metody nie zajmie dużo czasu. Refaktoryzacja nie była czasochłonna. Najwięcej wysiłku kosztowało mnie zrozumienie, co robi dany fragment kodu — stosując metodę „kopiuj-wklej”, i tak musiałbym wykonać tę analizę.

Część kodu `htmlStatement` została przekopiowana z oryginalnej wersji metody, na przykład kod związany z wykonaniem pętli. Można to poprawić, stosując dalsze przekształcenia refaktoryzacyjne. Mógłbym zdecydować się na ekstrakcję metod związanych z nagłówkiem, liniami końcowymi i informacją o samej należności. Odpowiedni przykład można znaleźć w części poświęconej przekształceniu Utworzenie Metody Szablonowej (s. 312). Wróćmy jednak do potrzeb użytkowników, którzy chcą zmienić zasady klasyfikacji filmów. Nadal nie znają ostatecznej listy kategorii, wiadomo jednak, że dla poszczególnych typów różne będą zasady naliczania należności oraz punktów stałego klienta. W tej chwili wprowadzanie takich zmian nie jest zbyt wygodne — musiałbym zmieniać kod warunkowy zależny od typów filmów. Spróbujmy więc ulepszyć ten fragment.

Zastąpienie warunkowej logiki wyznaczania ceny polimorfizmem

Podstawowy problem to instrukcja `switch`. Tworzenie takiej instrukcji w oparciu o atrybut innego obiektu nie jest dobrym rozwiązaniem. Jeśli już musisz jej używać — zrób to na własnych, a nie cudzych danych.

```
class Rental...
  double getCharge() { //podaj należność
    double result = 0;
    switch (getMovie().getPriceCode()) {
      case Movie.REGULAR:
        result += 2;
        if (getDaysRented() > 2)
          result += (getDaysRented() - 2) * 1.5;
        break;
      case Movie.NEW_RELEASE:
        result += getDaysRented() * 3;
        break;
      case Movie.CHILDRENS:
        result += 1.5;
        if (getDaysRented() > 3)
          result += (getDaysRented() - 3) * 1.5;
        break;
    }
    return result;
  }
```

Wynika z tego, że metoda `getCharge` powinna zostać przeniesiona do klasy `Movie`:

```
class Movie...
  double getCharge(int daysRented) { //podaj należność
    double result = 0;
    switch (getPriceCode()) {
      case Movie.REGULAR:
        result += 2;
        if (daysRented > 2)
          result += (daysRented - 2) * 1.5;
        break;
      case Movie.NEW_RELEASE:
        result += daysRented * 3;
        break;
      case Movie.CHILDRENS:
        result += 1.5;
        if (daysRented > 3)
          result += (daysRented - 3) * 1.5;
        break;
    }
    return result;
  }
```

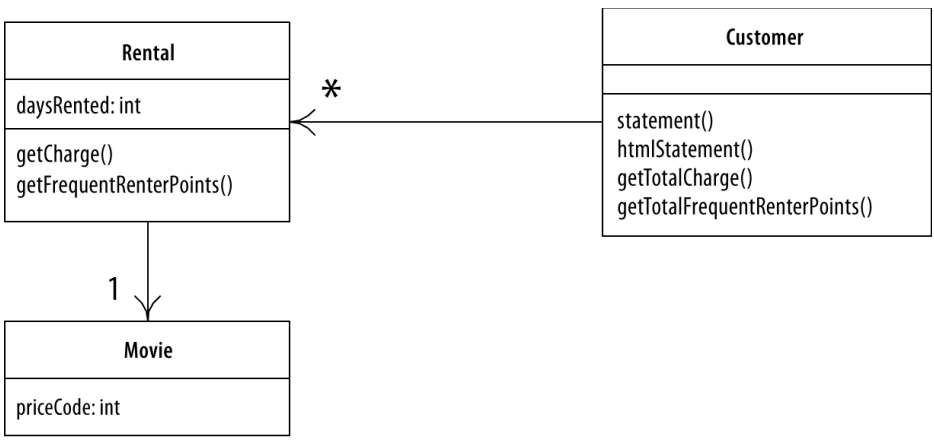
By kod zadziałał, jako parametr muszę przekazać czas wypożyczenia, czyli daną pochodzącą z klasy `Rental`. Metoda `getCharge` korzysta z dwóch informacji: czasu wypożyczenia oraz typu filmu. Dlaczego wolę przekazać długość wypożyczenia filmowi, a nie typ filmu klasie reprezentującej wypożyczoną pozycję? Otóż dlatego, że zmiany, których się spodziewam, obejmują dodawanie nowych typów filmów. Z zasady informacja na temat typów częściej ulega zmianom. Jeśli zmienią się typy, chcę, by zasięg zmian był możliwie niewielki. Dlatego wolę obliczać należność w klasie `Movie`.

Skompilowałem metodę w klasie `Movie` i zmieniłem kod metody `getCharge` w klasie `Rental` tak, by korzystała ona z nowo utworzonej metody (rysunki 1.12 i 1.13):

```
class Rental...
  double getCharge() { //podaj należność
    return _movie.getCharge(_daysRented);
  }
```

Po przeniesieniu metody `getChange` wykonam analogiczne przekształcenie fragmentu kodu wyliczającego punkty stałego klienta. W ten sposób wszystkie wyliczenia zależne od zmieniających się typów trafiają do klasy, która zna typ:

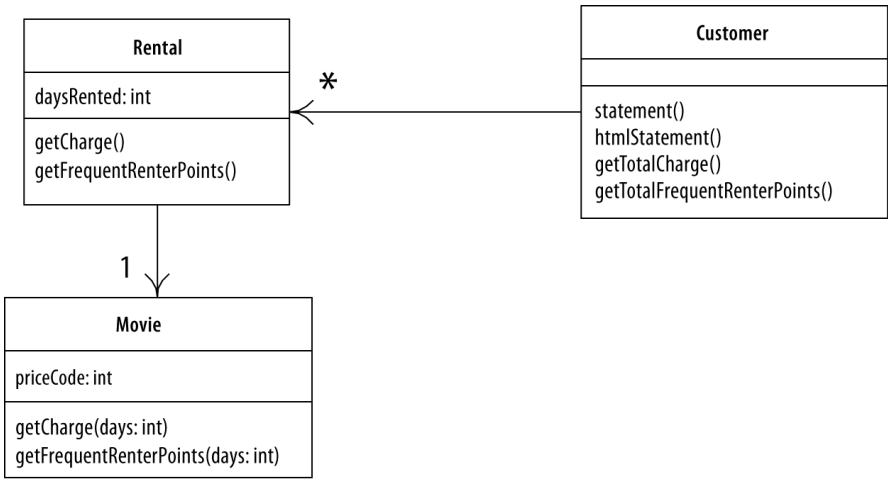
```
class Rental...
  int getFrequentRenterPoints() { //podaj punkty stałego klienta
    if ((getMovie().getPriceCode() == Movie.NEW_RELEASE) && getDaysRented() > 1)
      return 2;
    else
      return 1;
  }
```



Rysunek 1.12. *Diagram klas przed przeniesieniem metod do klasy Movie*

```
Class Rental...
  int getFrequentRenterPoints() { //podaj punkty stałego klienta
    return _movie.getFrequentRenterPoints(_daysRented);
  }

class Movie...
```

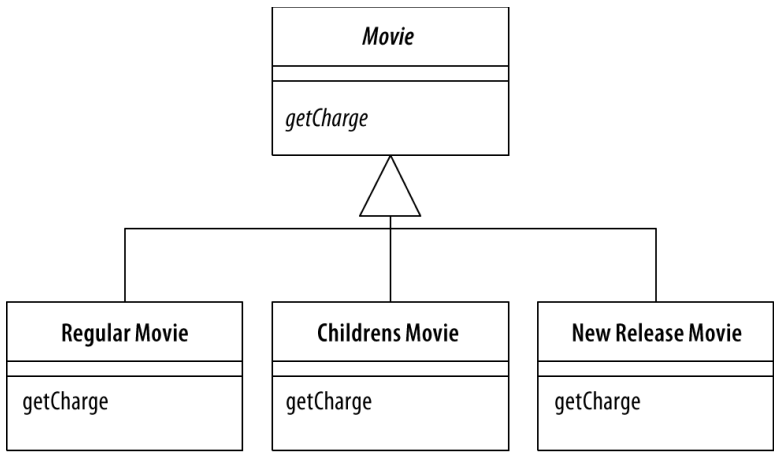
```
  int getFrequentRenterPoints(int daysRented) { //podaj punkty stałego klienta
    if ((getPriceCode() == Movie.NEW_RELEASE) && daysRented > 1)
      return 2;
    else
      return 1;
  }
```



Rysunek 1.13. *Diagram klas po przeniesieniu metod do klasy Movie*

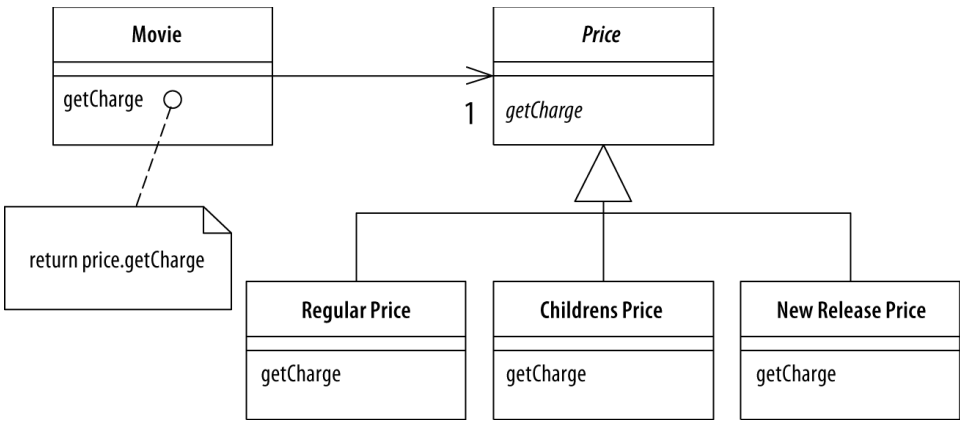
Wreszcie… Dziedziczenie

Mamy kilka typów filmów, które w różny sposób odpowiadają na to samo pytanie. Wydaje się, że to zadanie w sam raz dla podklas. Możemy wprowadzić trzy podklasy klasy `Movie`, z których każda inaczej wylicza należność (rysunek 1.14).



Rysunek 1.14. *Dziedziczenie z klasy Movie (podklasy odpowiadają filmowi standardowemu, filmowi dla dzieci oraz nowości)*

Ta decyzja projektowa pozwala mi zamienić instrukcję `switch` na wywołanie polimorficzne. Jest tylko jeden problem — to rozwiązanie nie zadziała. Film może zostać zaklasyfikowany do innego typu podczas swojego cyklu życia, natomiast typ obiektu jest stały. Pomocą służy wzorzec projektowy Stan [Banda Czterech]. Hierarchia klas po jego zastosowaniu została przedstawiona na rysunku 1.15.



Rysunek 1.15. *Zastosowanie wzorca projektowego Stan dla klasy Movie (RegularPrice, ChildrensPrice i NewReleasePrice oznaczają odpowiednio cenę za film standardowy, za film dla dzieci i za nowość)*

Wprowadzenie poziomu pośredniczącego pozwala na dziedziczenie z obiektu reprezentującego cenę. Wówczas cena wypożyczenia filmu może zmienić się w czasie cyklu życia obiektu `Movie`.

Jeśli znasz wzorce Bandy Czterech, być może zastanawiasz się, czy faktycznie zastosowałem tu wzorzec Stan, czy może jednak jest to Strategia. Co jest prawdą: czy klasa `Price` reprezentuje algorytm obliczania ceny (w takim wypadku powinna nazywać się raczej `Pricer` — „wyceniacz" lub `PricingStrategy` — „strategia cenowa"), czy stan danego filmu (na przykład informację o tym, że „Star Trek X" to nowa pozycja)? Na tym etapie wybór wzorca projektowego (i nazwy) odzwierciedla preferowany sposób myślenia o reprezentowanej dziedzinie. W tej chwili myślę o klasie `Price` jako o stanie danego filmu. Jeśli na późniejszym etapie uznam, że moją intencję lepiej wyrazi Strategia — przeprowadzę refaktoryzację sprowadzającą się do zmiany nazw.

Do zastosowania wzorca Stan potrzebuję trzech przekształceń. Najpierw użyję przekształcenia Zastąpienie Kodu Typu Wzorcem Stan lub Strategia (s. 202). W następnym kroku przeprowadzę Przeniesienie Metody (s. 124), w wyniku którego instrukcja `switch` trafi do klasy `Price`. Na końcu zastosuję Zastąpienie Instrukcji Warunkowej Polimorfizmem (s. 229) w celu wyeliminowania instrukcji `switch`.

Zgodnie z obietnicą zaczynam od Zastąpienia Kodu Typu Wzorcem Stan lub Strategia (s. 202). Pierwszy krok to Samoenkapsulacja Pola (s. 153). Chodzi o to, żeby wszystkie odwołania do typu odbywały się za pośrednictwem metod dostępowych `get...` i `set...`, a nie bezpośrednio. Jako że większość kodu pochodzi z innych klas, metody dostępowe już teraz są używane. Wyjątkiem jest konstruktor, w którym ma miejsce odwołanie bezpośrednie:

```
class Movie...
  public Movie(String name, int priceCode) {
    _name = name;
    _priceCode = priceCode;
   }
```

Zastąpię odwołanie bezpośrednie wywołaniem metody dostępowej:

```
class Movie
  public Movie(String name, int priceCode) {
    _name = name;
    setPriceCode(priceCode);
  }
```

Kompiluję kod i przeprowadzam testy, by upewnić się, że nic nie popsułem. Następnie dodaję nowe klasy. Fragment zależny od typu filmu umieszczam w obiekcie `Price`. Wykorzystuję do tego metodę abstrakcyjną w `Price` i konkretne metody w podklasach:

```
abstract class Price {                  //cena
  abstract int getPriceCode();
}
class ChildrensPrice extends Price { //cena za film dla dzieci
  int getPriceCode() {
    return Movie.CHILDRENS;
  }
}
class NewReleasePrice extends Price { //cena za nowość
  int getPriceCode() {
    return Movie.NEW_RELEASE;
  }
}
class RegularPrice extends Price {      //cena za standardową pozycję
  int getPriceCode() {
    return Movie.REGULAR;
  }
}
```

Na tym etapie kompiluję kod nowych klas.

Następnie zmieniam metody dostępowe związane z ceną w klasie `Movie`. Obecnie wyglądają one tak:

```
public int getPriceCode() {
    return _priceCode;
}
public setPriceCode (int arg) {
  _priceCode = arg;
}
 private int _priceCode;
```

Pole reprezentujące cenę będzie teraz typu `Price`. Metody dostępowe muszą zostać dopasowane do tej zmiany:

```
class Movie...
  public int getPriceCode() {
    return _price.getPriceCode();
  }
  public void setPriceCode(int arg) {
```

```
    switch (arg) {
      case REGULAR:
        _price = new RegularPrice();
        break;
      case CHILDRENS:
        _price = new ChildrensPrice();
        break;
      case NEW_RELEASE:
        _price = new NewReleasePrice();
        break;
      default:
        throw new IllegalArgumentException("Incorrect Price Code");
    }
  }
  private Price _price;
```

Ponownie przeprowadzam kompilację i testy. Złożone metody korzystające z tego kodu nie mają bladego pojęcia o tym, że świat uległ zmianie.

W kolejnym kroku stosuję Przeniesienie Metody (s. 124) dla `getCharge`:

```
class Movie...
  double getCharge(int daysRented) { //podaj należność
    double result = 0;
    switch (getPriceCode()) {
      case Movie.REGULAR:
        result += 2;
        if (daysRented > 2)
          result += (daysRented - 2) * 1.5;
        break;
     case Movie.NEW_RELEASE:
        result += daysRented * 3;
        break;
     case Movie.CHILDRENS:
        result += 1.5;
        if (daysRented > 3)
          result += (daysRented - 3) * 1.5;
        break;
     }
     return result;
  }
```

Jest to bardzo prosta operacja:

```
class Movie...
  double getCharge(int daysRented) { //podaj należność
    return _price.getCharge(daysRented);
  }

class Price...
  double getCharge(int daysRented) { //podaj należność
    double result = 0;
    switch (getPriceCode()) {
```

```
      case Movie.REGULAR:
        result += 2;
        if (daysRented > 2)
          result += (daysRented - 2) * 1.5;
          break;
      case Movie.NEW_RELEASE:
        result += daysRented * 3;
        break;
      case Movie.CHILDRENS:
        result += 1.5;
        if (daysRented > 3)
          result += (daysRented - 3) * 1.5;
        break;
    }
    return result;
  }
```

Po przeniesieniu metody stosuję na niej Zastąpienie Instrukcji Warunkowej Polimorfizmem (s. 229). Stan przed:

```
class Price...
  double getCharge(int daysRented) { //podaj należność
    double result = 0;
    switch (getPriceCode()) {
      case Movie.REGULAR:
        result += 2;
        if (daysRented > 2)
          result += (daysRented - 2) * 1.5;
        break;
      case Movie.NEW_RELEASE:
        result += daysRented * 3;
        break;
      case Movie.CHILDRENS:
        result += 1.5;
        if (daysRented > 3)
          result += (daysRented - 3) * 1.5;
        break;
    }
    return result;
}
```

Logiki warunkowej będę się pozbywał po kolei, po jednym odgałęzieniu na raz. Zacznę od ceny wypożyczenia standardowego filmu:

```
class RegularPrice...
  double getCharge(int daysRented){ //podaj należność
    double result = 2;
    if (daysRented > 2)
      result += (daysRented - 2) * 1.5;
    return result;
  }
```

Przesłaniam w ten sposób odpowiedni fragment instrukcji `switch` w nadklasie, której jednak na razie nie będę zmieniał. Kompiluję i testuję zmieniony kod, po czym zabieram się za kolejne rozgałęzienie. Czasem, aby upewnić się, że wykonywany jest kod podklasy, specjalnie wprowadzam w nim pewien błąd, który powinien zostać wykryty przez testy (nie żebym był paranoikiem).

```
class ChildrensPrice
  double getCharge(int daysRented){ //podaj należność
    double result = 1.5;
    if (daysRented > 3)
      result += (daysRented - 3) * 1.5;
    return result;
   }

class NewReleasePrice...
  double getCharge(int daysRented){ //podaj należność
    return daysRented * 3;
  }
```

Po obsłużeniu wszystkich rozgałęzień zamieniam metodę `Price.getCharge` na abstrakcyjną:

```
class Price...
  abstract double getCharge(int daysRented); //podaj należność
```

W analogiczny sposób postępuję z metodą `getFrequentRenterPoints`:

```
class Rental...
  int getFrequentRenterPoints(int daysRented) { //podaj punkty stałego klienta
    if ((getPriceCode() == Movie.NEW_RELEASE) && daysRented > 1)
      return 2;
    else
      return 1;
  }
```

Najpierw przenoszę metodę do klasy `Price`:

```
Class Movie...
  int getFrequentRenterPoints(int daysRented) { //podaj punkty stałego klienta
    return _price.getFrequentRenterPoints(daysRented);
  }
Class Price...
  int getFrequentRenterPoints(int daysRented) { //podaj punkty stałego klienta
    if ((getPriceCode() == Movie.NEW_RELEASE) && daysRented > 1)
      return 2;
    else
      return 1;
  }
```

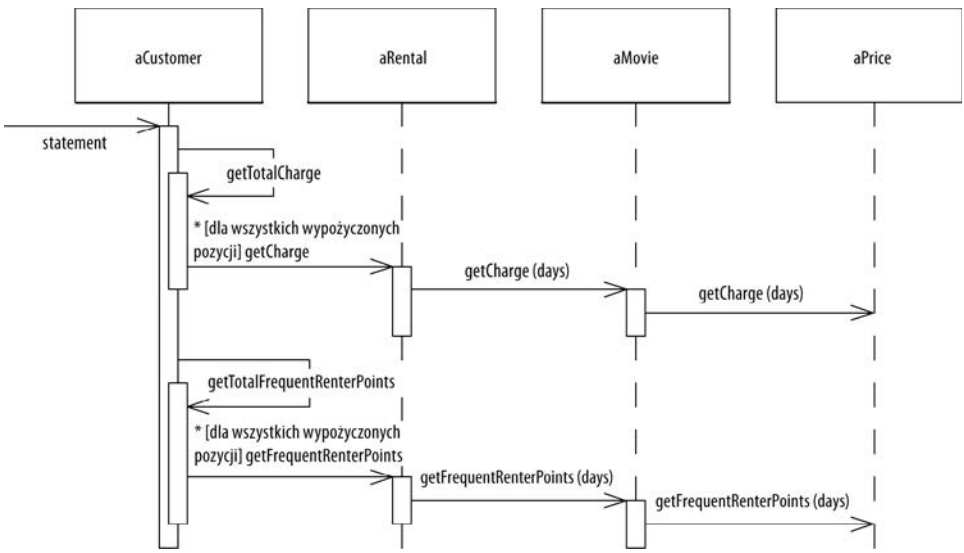
W tym wypadku nie zamieniam jednak metody w nadklasie w metodę abstrakcyjną. Zamiast tego tworzę przesłaniającą ją metodę w klasie reprezentującej nowość:

```
Class NewReleasePrice
  int getFrequentRenterPoints(int daysRented) { //podaj punkty stałego klienta
    return (daysRented > 1) ? 2: 1;
  }

Class Price...
  int getFrequentRenterPoints(int daysRented){ //podaj punkty stałego klienta
    return 1;
  }
```

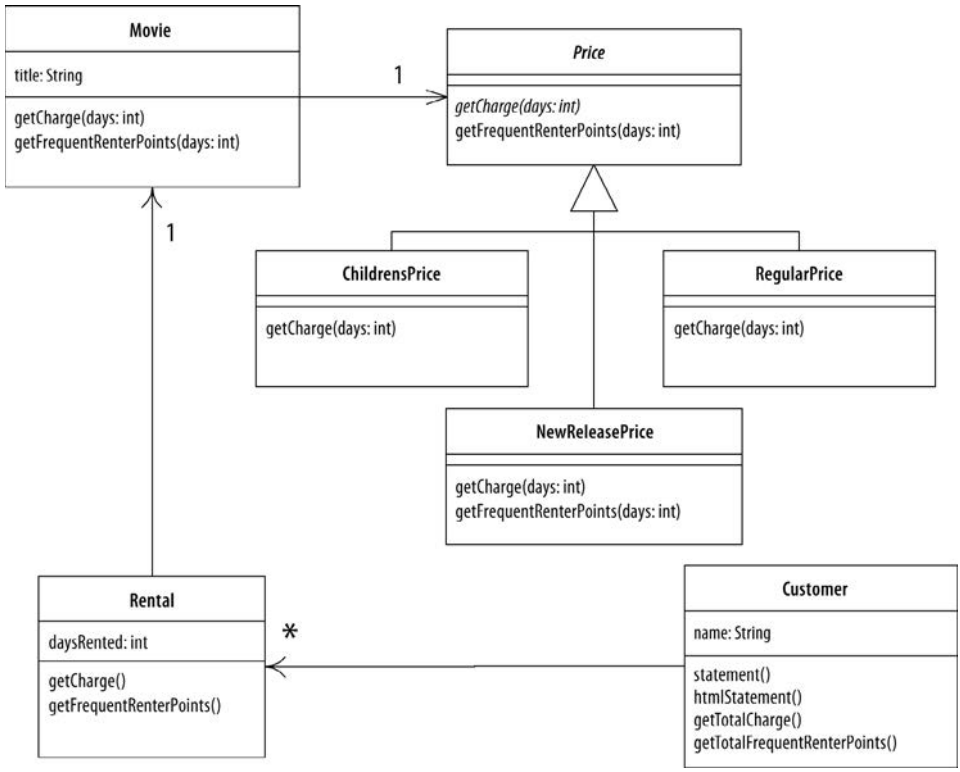
Zastosowanie wzorca Stan okazało się nie lada wyzwaniem. Czy było warto? Zyskałem to, że w razie zmiany zasad naliczania należności lub po wprowadzeniu nowych typów cen koszt wprowadzenia zmian w kodzie będzie o wiele niższy. Zastosowanie wzorca Stan nie wpływa na pozostałe części aplikacji. Oczywiście w tak małym programie trudno mówić o wielkich zyskach. Inaczej przedstawiałaby się sytuacja w rozbudowanej aplikacji z dziesiątkami metod zależnych od ceny.

Zmiany wprowadzałem w małych krokach. Wydaje się to czasochłonne, zwróć jednak uwagę, że ani razu nie musiałem uruchamiać debuggera, w związku z czym proces przebiegał bardzo płynnie. O wiele więcej czasu zajęło mi napisanie tekstu towarzyszącego przekształceniom niż wprowadzenie zmian w omawianym kodzie.

Zakończyłem właśnie drugie większe przekształcenie refaktoryzacyjne. Dzięki niemu zmiana klasyfikacji filmów oraz reguł naliczania należności i punktów stałego klienta będzie o wiele łatwiejsza. Działanie kodu przekształconego tak, by realizował wzorzec Stan, przedstawiają rysunki 1.16 i 1.17.



Rysunek 1.16. *Diagram interakcji po zastosowaniu wzorca Stan*



Rysunek 1.17. *Diagram klas po zastosowaniu wzorca Stan*

Podsumowanie

Przedstawiony przykład jest bardzo prosty. Mam nadzieję, że jest także wystarczająco obrazowy, by dać pojęcie na temat istoty refaktoryzacji. Zastosowałem kilka przekształceń: Ekstrakcję Metody (s. 94), Przeniesienie Metody (s. 124) oraz Zastąpienie Instrukcji Warunkowej Polimorfizmem (s. 229). Wszystkie one powodują powstanie kodu, który ma lepszą strukturę i jest łatwiejszy w utrzymaniu. Taki kod znacznie odbiega od paradygmatu proceduralnego — przyzwyczajenie się do niego może zabrać trochę czasu, jednak później trudno już wyobrazić sobie drogę powrotną.

Najważniejsza lekcja płynąca z omówionego przykładu to stały rytm refaktoryzacji: testy, mała zmiana, testy, mała zmiana, testy… To dzięki niemu refaktoryzacja przebiega sprawnie i bezpiecznie.

Jeśli nadal ze mną jesteś, zakładam, że rozumiesz już, czym jest refaktoryzacja. Możemy zatem przejść do zasad i teorii. Obiecuję — nie będzie ich zbyt dużo!

Skorowidz

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

V

W

Z

REFAKTORYZACJA

Ulepszanie struktury istniejącego kodu

KANON INFORMATYKI

Jak ryzykowne jest grzebanie w kodzie – wszyscy doskonale wiemy. Im głębiej sięgasz... tym więcej pojawia się nowych problemów i jeszcze więcej rzeczy wymaga zmian. A nieustanne „poprawianie" działającego kodu może w końcu doprowadzić do powstania trudno wykrywalnych, krytycznych błędów. Jednak co zrobić, jeśli „odziedziczymy" nieefektywny, trudny w utrzymaniu i rozszerzaniu program? Jak poprawić jego strukturalną spójność i wydajność? Wypracowywane latami przez najlepszych ekspertów techniki refaktoryzacji, czyli ulepszania projektu istniejącego kodu, są dziś sprawdzonymi rozwiązaniami, zapewniającymi jego trwałą czytelność i możliwość efektywnego rozwoju. Opracowane głównie na potrzeby frameworków, są obecnie narzędziem wykorzystywanym dla całego procesu produkcji oprogramowania. Jednak dla wielu programistów proces refaktoryzacji pozostaje wiedzą tajemną, bo jak dotąd żaden podręcznik nie przedstawił używanych przy tym technik w praktycznej, łatwej do wykorzystania formie. A przecież przeprowadzona błędnie lub w zbytnim pośpiechu refaktoryzacja zamiast ulepszenia kodu może kosztować nas dodatkowe dni lub całe tygodnie stresującej pracy nad programem.

Oto podręcznik, w którym słynny mentor i programistyczny guru Martin Fowler wraz z kilkoma innymi znanymi programistami podejmują się pierwszego tak gruntownego i przejrzystego objaśnienia technik związanych ze skutecznym procesem refaktoryzacji. Książka ta przedstawia zasady i najlepsze praktyki refaktoryzacyjne oraz zawiera wskazówki na temat tego, kiedy i jak zacząć ingerować w kod. Znajdziesz tu wyczerpujący katalog siedemdziesięciu przekształceń refaktoryzacyjnych. Każdemu z nich towarzyszą wskazówki dotyczące możliwości wykorzystania, instrukcja opisująca kolejne kroki oraz przykład. Ten podręcznik pokaże Ci zatem, jak przekształcać kod w sposób kontrolowany i efektywny, jak refaktoryzować go bez wprowadzania błędów, konsekwentnie ulepszając jego strukturę, oraz jak skutecznie go testować. Choć przedstawione w książce przykłady zostały napisane w języku Java, idee te znajdą zastosowanie w każdym innym języku obiektowym. Ponadto w opisach części przekształceń dodano uwagi związane z ich stosowaniem w innych językach.

Poznaj sprawdzone techniki ulepszania istniejącego kodu!

Martin Fowler to niezależny konsultant, od ponad dziesięciu lat stosujący obiekty do rozwiązywania ważnych problemów biznesowych. Wśród jego klientów można wymienić Chryslera, Citibank, brytyjską Narodową Służbę Zdrowia, Andersen Consulting i Netscape Communications. Ponadto Fowler regularnie wypowiada się na temat obiektów, języka UML oraz wzorców projektowych. Jest autorem nagradzanych książek, na przykład takich, jak „Analysis Patterns", „UML w kropelce", „Architektura systemów zarządzania przedsiębiorstwem. Wzorce projektowe".

W książce tej znajdziesz między innymi opis takich zagadnień, jak:

- zasady refaktoryzacji
- identyfikowanie błędów i problemów z kodem
- testowanie
- katalog przekształceń refaktoryzacyjnych
- konstrukcja metod
- przenoszenie składowych pomiędzy obiektami
- organizacja danych
- upraszczanie wyrażeń warunkowych i wywołań metod
- praca z hierarchią dziedziczenia
- duże przekształcenia
- refaktoryzacja i reużywalność
- narzędzia refaktoryzacyjne

ISBN 978-83-246-3243-5

helion.pl
księgarnia internetowa

Nr katalogowy: 6795

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

Informatyka w najlepszym wydaniu