



Refaktoryzacja

Ulepszanie struktury istniejącego kodu

PIERWSZY PODRĘCZNIK TAK GRUNTOWNIE I PRZEJRZYŚCIE
WYJASNIAJĄCY 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 wprowadzenia błędów?
- Jak skutecznie wykorzystywać przekształcenia refaktoryzacyjne?

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

» 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

» Kontakt

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 32 230 98 63
e-mail: helion@helion.pl
© Helion 1991–2011

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 przejrzysto 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!

Spis treści

Słowo wstępne	9
Przedmowa	11
Czym jest refaktoryzacja?	12
Co zawiera ta książka?	12
Kto powinien przeczytać tę książkę?	13
Podstawowe prace wykonane przez innych	14
Podziękowania	14
1. Refaktoryzacja: pierwszy przykład	17
Punkt wyjścia	17
Pierwszy krok refaktoryzacji	22
Dekompozycja i redystrybucja metody statement	22
Zastąpienie warunkowej logiki wyznaczania ceny polimorfizmem	39
Podsumowanie	48
2. Zasady refaktoryzacji	49
Definicja refaktoryzacji	49
Po co refaktoryzować?	50
Kiedy refaktoryzować?	52
Co mam powiedzieć kierownikowi?	54
Problemy z refaktoryzacją	56
Refaktoryzacja a projektowanie	59
Refaktoryzacja a wydajność	61
Skąd się wzięła refaktoryzacja?	62
3. Brzydkie zapaszki w kodzie	65
Zduplikowany kod	66
Długa metoda	66
Duża klasa	67
Długa lista parametrów	68
Rozbieżne zmiany	68
Fala uderzeniowa	69
Zazdrosne metody	69
Stada danych	69

Opętanie prymitywami	70
Instrukcje switch	70
Równoległe hierarchie dziedziczenia	71
Leniwa klasa	71
Spekulacyjne uogólnienia	71
Pole tymczasowe	72
Łańcuchy komunikatów	72
Pośrednik	73
Niestosowna bliskość	73
Alternatywne klasy z różnymi interfejsami	73
Niekompletna klasa biblioteczna	73
Klasa opakująca dane	74
Odmowa przyjęcia spadku	74
Uwagi	75
4. Testy	77
Zalety samotestującego się kodu	77
Testy jednostkowe JUnit	79
Więcej testów	84
5. Katalog przekształceń refaktoryzacyjnych	89
Format opisu przekształceń	89
Odnajdywanie odwołań	90
Dojrzałość przekształceń	91
6. Konstrukcja metod	93
Ekstrakcja Metody	94
Wchłonięcie Metody	100
Wchłonięcie Zmiennej Tymczasowej	102
Zastąpienie Zmiennej Tymczasowej Zapytaniem	103
Wprowadzenie Zmiennej Objaśniającej	107
Podział Zmiennej Tymczasowej	111
Eliminacja Przypisywania Wartości Parametrom	114
Zastąpienie Metody Obiektem	118
Zastąpienie Algorytmu	121
7. Przenoszenie składowych pomiędzy obiektami	123
Przeniesienie Metody	124
Przeniesienie Pola	128
Ekstrakcja Klasy	131
Wchłonięcie Klasy	135
Ukrycie Delegata	138
Usunięcie Pośrednika	141
Wprowadzenie Obcej Metody	143
Wprowadzenie Rozszerzenia Lokalnego	145
8. Organizacja danych	151
Samoenkapsulacja Pola	153
Zastąpienie Typu Prostego Obiektem	156
Zamiana Wartości na Referencję	159
Zamiana Referencji na Wartość	163
Zastąpienie Tablicy Obiektem	166

Duplikacja Obserwowanych Danych	169
Zamiana Asocjacji Jednokierunkowej na Dwukierunkową	176
Zamiana Asocjacji Dwukierunkowej na Jednokierunkową	179
Zastąpienie Magicznej Liczby Stałą Symboliczną	183
Enkapsulacja Pola	184
Enkapsulacja Kolekcji	185
Zastąpienie Rekordu Klasą z Danymi	193
Zastąpienie Kodu Typu Klasą	194
Zastąpienie Kodu Typu Podklasami	199
Zastąpienie Kodu Typu Wzorcem Stan lub Strategia	202
Zastąpienie Podklasy Polami	207
9. Upraszczenie wyrażeń warunkowych	211
Dekompozycja Instrukcji Warunkowej	212
Scalenie Instrukcji Warunkowej	214
Scalenie Zduplikowanych Fragmentów Instrukcji Warunkowej	217
Usunięcie Flagi Kontrolnej	219
Zastąpienie Zagnieżdżonej Instrukcji Warunkowej Instrukcją Wyjścia	224
Zastąpienie Instrukcji Warunkowej Polimorfizmem	229
Wprowadzenie Obiektu Pustego	233
Wprowadzenie Asercji	240
10. Upraszczenie wywołań metod	243
Zmiana Nazwy Metody	245
Dodanie Parametru	247
Usunięcie Parametru	249
Rozdzielenie Zapytania i Modyfikacji	251
Parametryzacja Metody	255
Zastąpienie Parametru Metodami o Różnych Nazwach	257
Przekazanie Całego Obiektu	260
Zastąpienie Parametru Metodą	263
Wprowadzenie Obiektu Parametrycznego	266
Usunięcie Metody Ustawiającej Wartość	270
Ukrycie Metody	273
Zastąpienie Konstruktora Metodą Wytwórczą	274
Enkapsulacja Rzutowania w Dół Hierarchii	278
Zastąpienie Kodu Błędu Wyjątkiem	280
Zastąpienie Wyjątku Testem	285
11. Praca z hierarchią dziedziczenia	289
Przesunięcie Pola w Górę Hierarchii	290
Przesunięcie Metody w Górę Hierarchii	291
Przesunięcie Ciała Konstruktora w Górę Hierarchii	294
Przesunięcie Metody w Dół Hierarchii	297
Przesunięcie Pola w Dół Hierarchii	298
Ekstrakcja Podklasy	299
Ekstrakcja Nadklasy	304
Ekstrakcja Interfejsu	308
Zwinięcie Hierarchii	311
Utworzenie Metody Szablonowej	312
Zastąpienie Dziedziczenia Delegacją	319
Zastąpienie Delegacji Dziedziczeniem	322

12. Duże przekształcenia	325
Rozplątanie Hierarchii Dziedziczenia	327
Przekształcenie Projektu Proceduralnego na Obiekty	332
Oddzielenie Dziedziny od Prezentacji	334
Ekstrakcja Hierarchii	338
13. Refaktoryzacja i reużywalność a rzeczywistość	343
Zejście na ziemię	344
Dlaczego programiści nie chcą refaktoryzować <i>własnych</i> programów?	345
Zejście na ziemię (raz jeszcze)	354
Materiały na temat refaktoryzacji	355
Sugestie na temat wielokrotnego wykorzystania kodu i transferu technologii	355
Na koniec	356
Bibliografia	356
14. Narzędzia refaktoryzacyjne	359
Refaktoryzacja z wykorzystaniem narzędzia	359
Techniczne kryteria narzędzia refaktoryzacyjnego	360
Praktyczne kryteria narzędzia refaktoryzacyjnego	362
Podsumowanie	363
15. Wykorzystanie zdobytej wiedzy	365
Bibliografia	369
Lista refaktoryzacji	373
Brzydkie zapaszki i ich przekształcenia	375
Skorowidz	377

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.

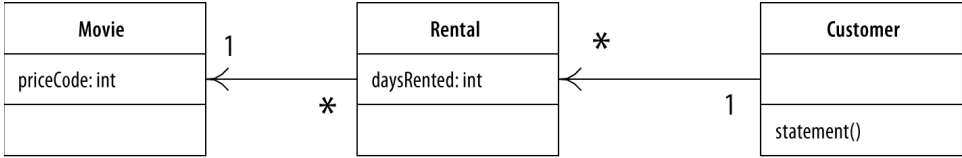
Z wyborem pierwszego przykładu wiąże się jednak pewien problem. Jeśli przedstawię rozbudowany program, opis jego samego i refaktoryzacji będzie zbyt skomplikowany dla Czytelnika. Okazuje się, że nawet umiarkowanie skomplikowany przykład wymaga ponad setki stron! Natomiast gdy wybiorę program na tyle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

Pszczególne 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 dostępne:

```

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;
    };
}

```

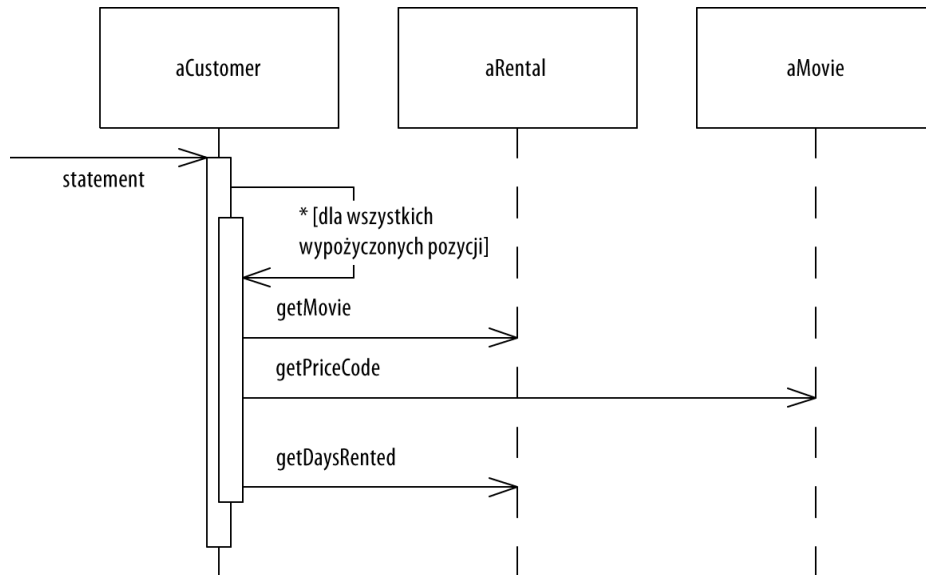
Dodatkowo klasa Customer posiada metodę generującą podsumowanie. Rysunek 1.2 przedstawia diagram interakcji tej metody. Kod metody został przedstawiony poniżej.

```

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 jednak 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.

Założmy 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 zmieniamy 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 popsuć, 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 rozszerzenie. 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 mogą 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() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Lista pozycji wypożyczonych przez klienta " + getName() + "\n";
    while (rentals.hasMoreElements()) {
        double thisAmount = 0;
        Rental each = (Rental) rentals.nextElement();

        thisAmount = amountFor(each);
        frequentRenterPoints++;
        if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
            each.getDaysRented() > 1) frequentRenterPoints++;

        result += "\t" + each.getMovie().getTitle() + "\t" +
            String.valueOf(thisAmount) + "\n";
        totalAmount += thisAmount;
    }
    result += " Należność wynosi " + String.valueOf(totalAmount) + "\n";
    result += " Klient otrzymał " + String.valueOf(frequentRenterPoints) + " punktów
    ↳ stałego klienta ";
    return result;
}

private int amountFor(Rental each) {
    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) {
    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 немало 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 — *przyj. 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` („pożyczka 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. Dobry 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 wynioskowe 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
```

```

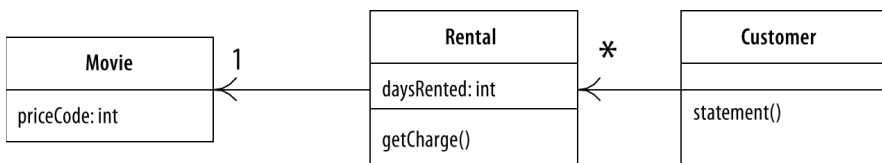
Enumeration rentals = _rentals.elements(); //wypożyczone pozycje
String result = "Lista pozycji wypożyczonych przez klienta " + getName() + "\n";
while (rentals.hasMoreElements()) {
    double thisAmount = 0;
    Rental each = (Rental) rentals.nextElement();

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

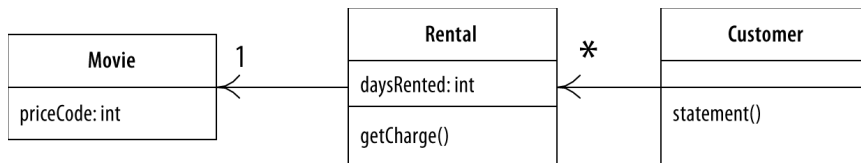
```
class Customer...
public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Lista pozycji wypożyczonych przez klienta " + getName() + "\n";
    while (rentals.hasMoreElements()) {
        Rental each = (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();
    }

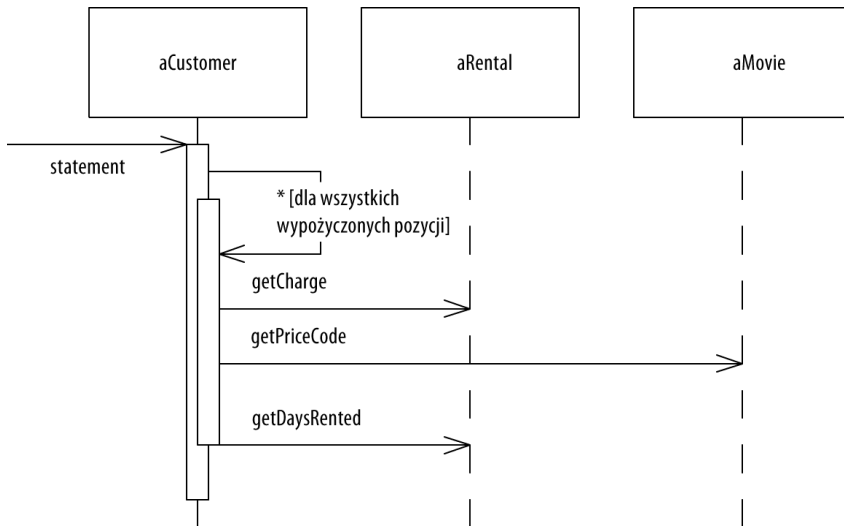
    //linie końcowe
    result += "Należność wynosi " + String.valueOf(totalAmount) + "\n";
    result += "Klient otrzymał " + String.valueOf(frequentRenterPoints) + " punktów
    ↪ stałego klienta";
    return result;
}

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

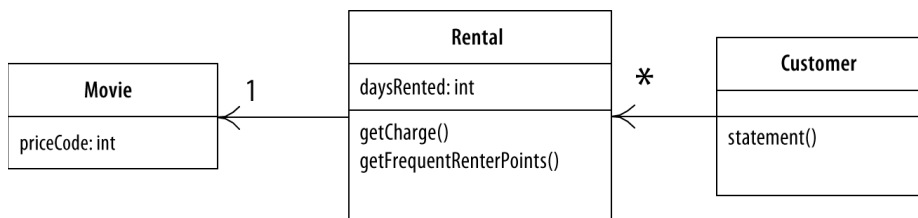
Podsumuję przeprowadzone zmiany za pomocą diagramów UML (rysunki 1.4 – 1.7). Diagramy po lewej przedstawiają stan „przed”, po prawej — stan „po”.



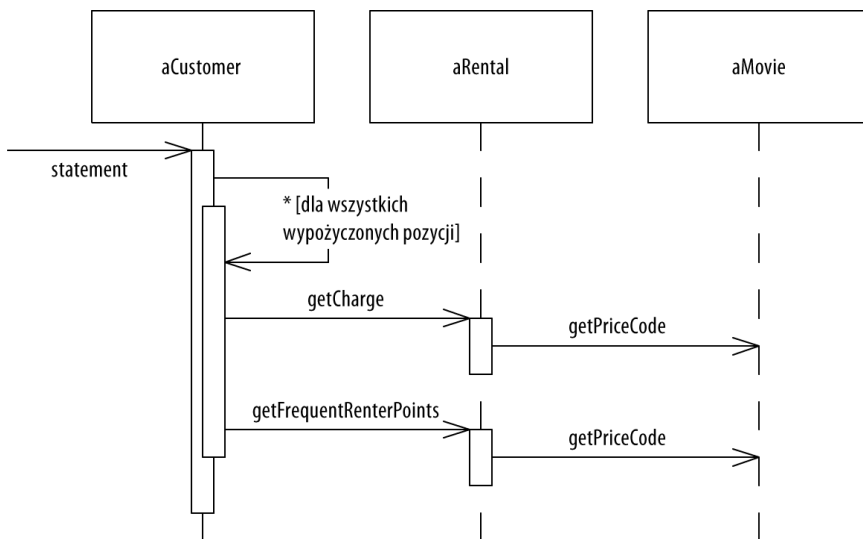
Rysunek 1.4. Diagram klas przed ekstrakcją i przeniesieniem obliczania punktów stałego klienta



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() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    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() {
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    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 stalego 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";
    }
}
}

```

```

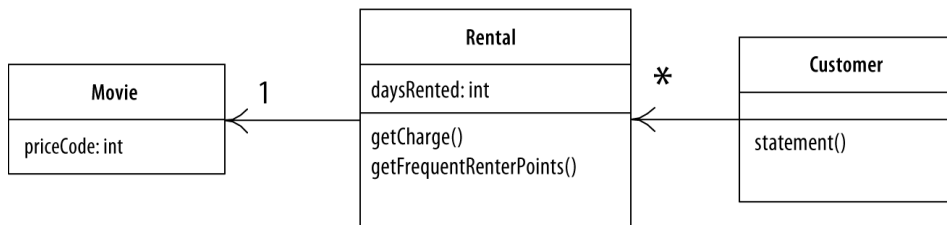
    }

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

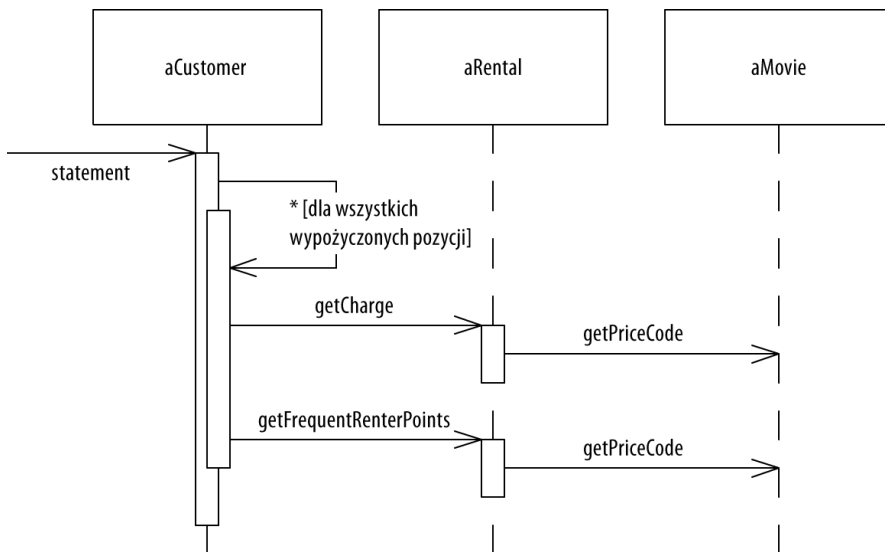
private int getTotalFrequentRenterPoints(){ //podaj wynikową liczbę punktów stałego klienta
int result = 0;
Enumeration rentals = _rentals.elements();
while (rentals.hasMoreElements()) {
    Rental each = (Rental) rentals.nextElement();
    result += each.getFrequentRenterPoints();
}
return result;
}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

```

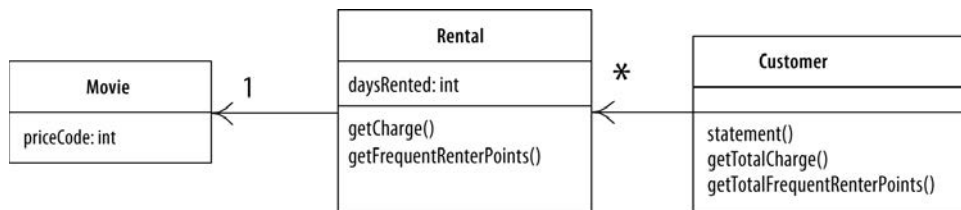
Rysunki 1.8 – 1.11 przedstawiają wprowadzane zmiany na diagramach klas i sekwencji.



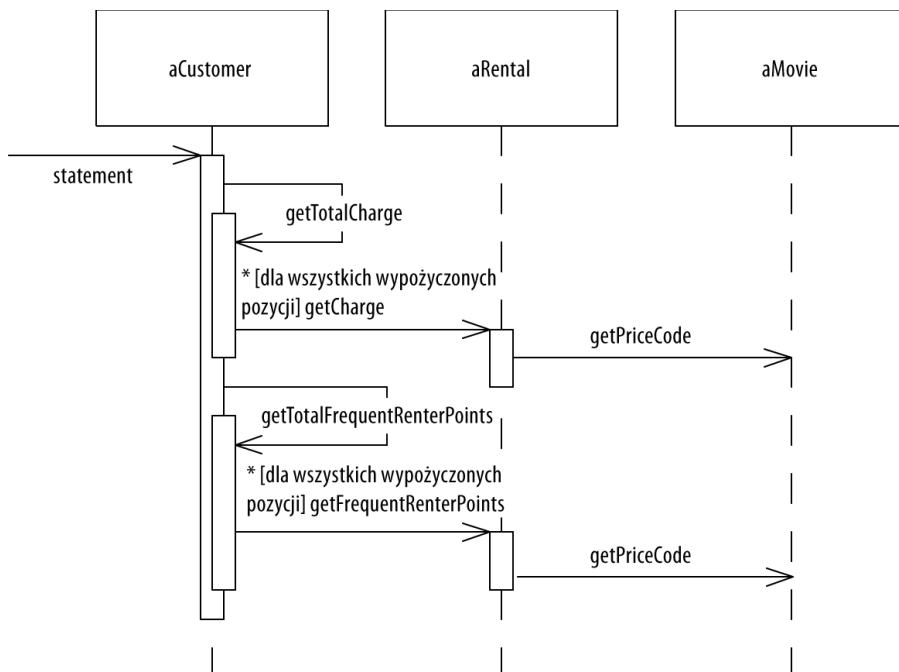
Rysunek 1.8. Diagram klas przed ekstrakcją obliczania punktów stałego klienta



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 oczywiście 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 zdecydowałoby 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 musimy 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;
}
```

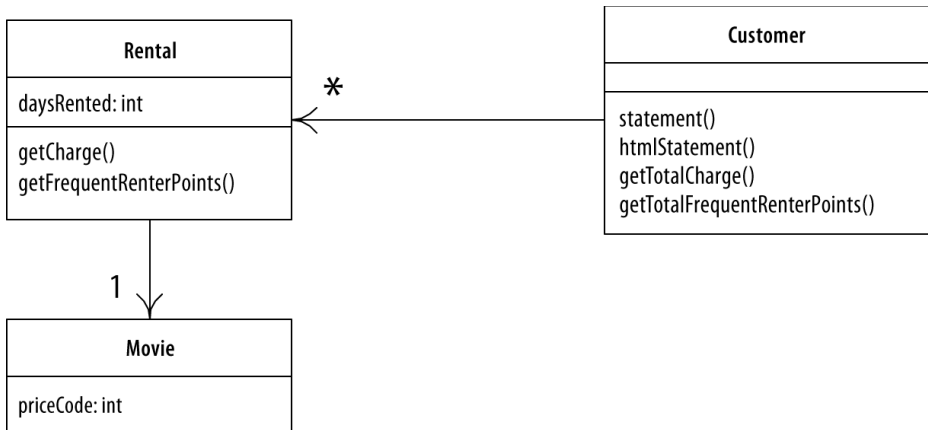
By kod zadziałał, jako parametr muszę przekazać czas wypożyczenia, czyli daną pochodzącą z klasy Rental. Metoda getCharge korzysta z dwóch informacji: czasu wypożyczenia oraz typu filmu. Dlaczego wołę przekazać długość wypożyczenia filmowi, a nie typ filmu klasie reprezentującej wypożyczoną pozycję? 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 wołę 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;
}
```



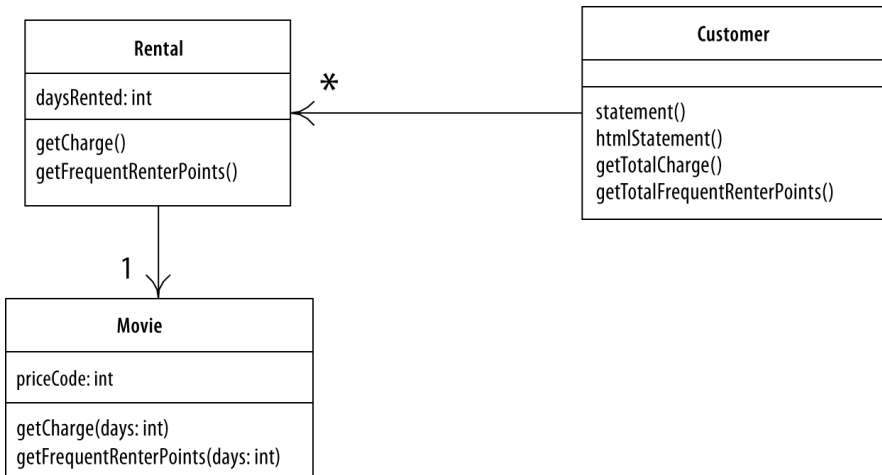
Rysunek 1.12. Diagram klas przed przeniesieniem metod do klasy Movie

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

class Movie...
```

```

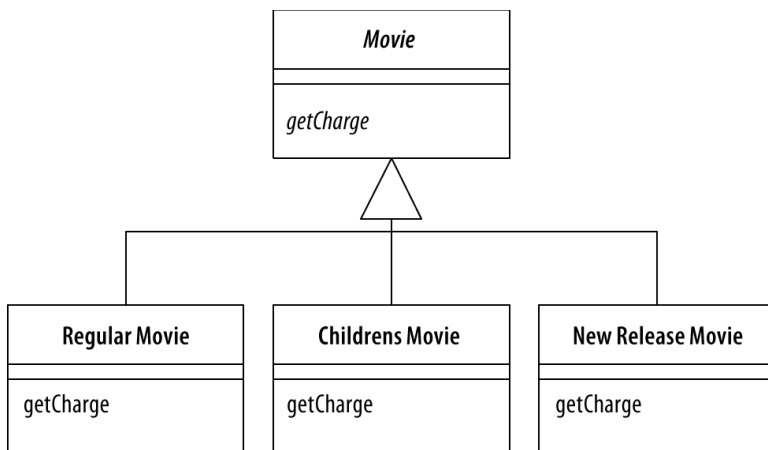
int getFrequentRenterPoints(int daysRented) { //podaj punkty stałego klienta
    if ((getPriceCode() == Movie.NEW_RELEASE) && daysRented > 1)
        return 2;
    else
        return 1;
}
    
```



Rysunek 1.13. Diagram klas po przeniesieniu metod do klasy Movie

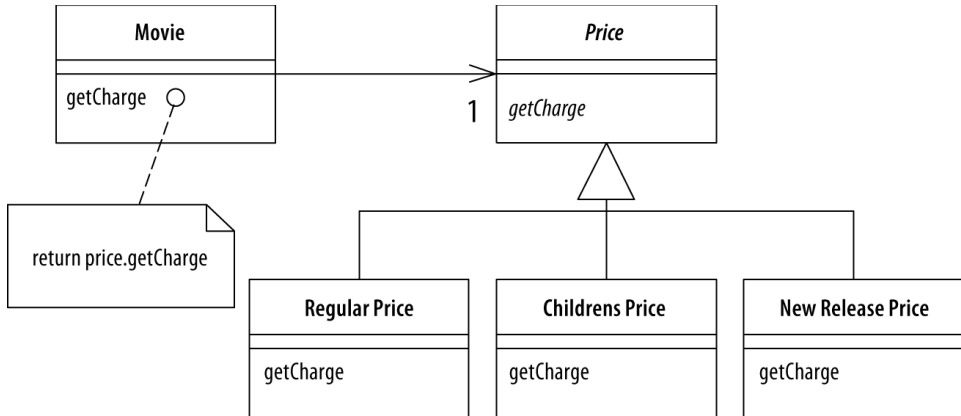
Wreszcie... Dziedziczenie

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 dostępne 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 dostępne 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 dostępne 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. Zaczęę 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;
  }

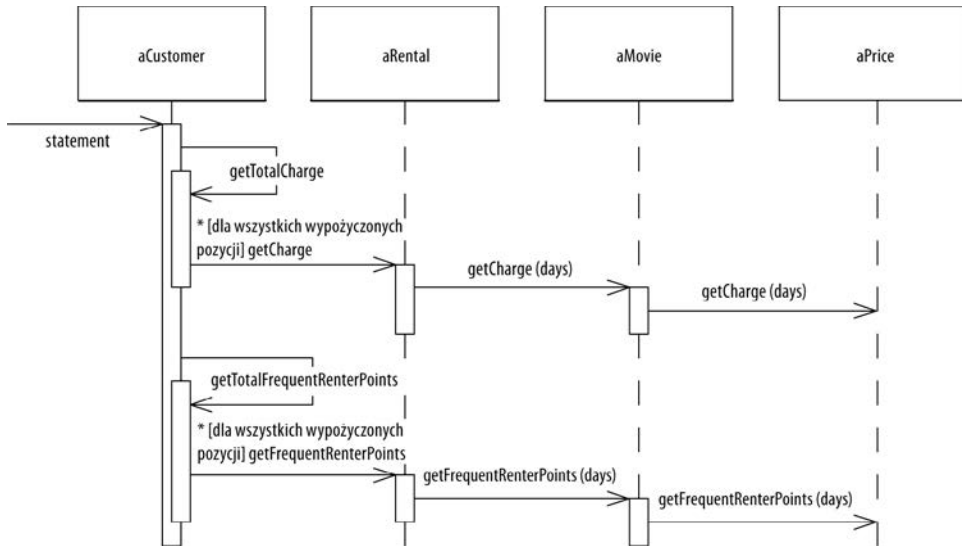
Class Price...
  int getFrequentRenterPoints(int daysRented){ //podaj punkty stałego klienta
    return 1;
  }

```

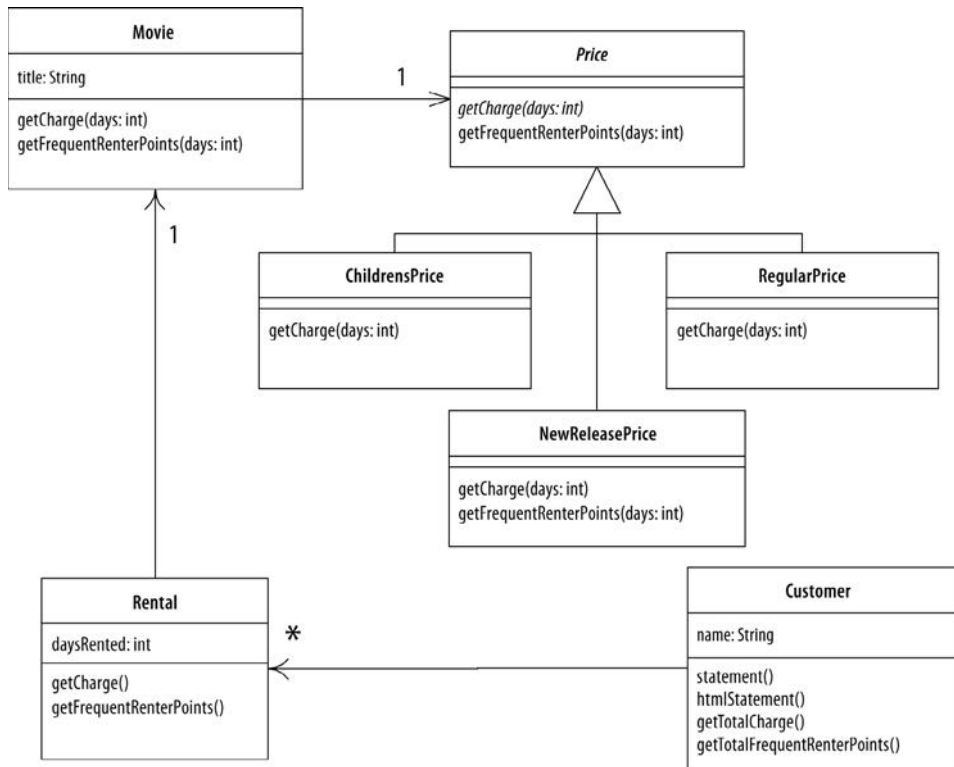
Zastosowanie wzorca Stan okazało się nie lada wyzwaniem. Czy było warto? Zyskałem to, że w razie zmiany zasad naliczania należności lub po wprowadzeniu nowych typów cen koszt wprowadzenia zmian w kodzie będzie o wiele niższy. Zastosowanie wzorca Stan nie wpływa na pozostałe części aplikacji. Oczywiście 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 wzorec 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

A

algorytm
 alternatywny, 122
 zastąpienie, 121
alternatywne klasy, 73
analizator programów, 352
asercja, 240
asocjacja dwukierunkowa, 179
Assert, 242

B

baza danych nieobiektowa, 57
bezpieczeństwo, 351
błąd
 komunikat dokładniejszy, 82
 szukanie przyczyny, 77
 wykrywanie, 52
 zgłoszenie, 84
Brant John, 14
break, 219

C

C++, 348
case, 70
Class.forName, 275, 276
code review, 53
continue, 219
czyszczenie kodu, 50
czytelność kodu, 26, 50, 51, 61

D

dane
 dziedziczenie, 74
 migracja, 57
 stado, 69
Date, 116
dekompozycja, 22
 metody, 118
Dekompozycja Instrukcji Warunkowej, 212
 instrukcja, 212
 motywacja, 212
 przykład, 212
delegacja, 138, 319, 322, 327
delegat, 138, 141
diagram
 interakcji, 20, 47
 klas, 18, 29, 32, 33, 36, 40, 41, 48
 sekwencji, 33, 36
długa metoda, 66, 118
Dodanie Parametru, 247
 instrukcja, 247
 motywacja, 247
dokładniejszy komunikat o błędzie, 82
drzewo parsowania, 361, 362
Duplikacja Obserwowanych Danych, 169
 instrukcja, 170
 motywacja, 169
 przykład, 170
duża klasa, 67
dziedziczenie, 41, 71, 319, 322, 327
 danych, 74
 metody, 74
dziedzina, 334

E

Ekstrakcja
 Hierarchii, 338
 instrukcja, 339
 motywacja, 338
 przykład, 339
 Interfejsu, 308
 instrukcja, 309
 motywacja, 308
 przykład, 309
 Klasy, 131
 instrukcja, 131
 motywacja, 131
 przykład, 132
 Metody, 22, 31, 94, 95, 109, 360
 Nadklasy, 304
 instrukcja, 304
 motywacja, 304
 przykład, 305
 Podklasy, 299
 instrukcja, 299
 motywacja, 299
 przykład, 300
 ekstremalne programowanie, 62
 elastyczne rozwiązanie, 60
 element danych, 156
 Eliminacja Przypisywania Wartości
 Parametrom, 114
 instrukcja, 115, 119
 motywacja, 114, 118
 przykład, 115, 119
 Employee, 205, 231, 309
 EmployeeType, 231
 enkapsulacja, 73
 Enkapsulacja, 184
 Kolekcji, 185
 instrukcja, 185
 motywacja, 185
 przykład, 186, 190
 Pola, 184
 instrukcja, 184
 motywacja, 184
 Rzutowania w Dół Hierarchii, 278
 instrukcja, 278
 motywacja, 278
 przykład, 279
 Tablic
 przykład, 191

Enumeration, 98
 equals, 148, 164

F

fala uderzeniowa, 69, 71
 flaga kontrolna, 219, 221
 for, 111
 funkcjonalność, 21

G

gorące punkty wydajności, 62
 grupa parametrów, 266
 guard clauses, 224
 GUI, 171

H

hermetyzacja, 184
 hierarchia, 74
 dziedziczenia, 71, 231, 327
 klas, 338
 htmlStatement, 38

I

IBM VisualAge, 91
 if-then-else, 212, 224
 indirection, 55
 inspekcja kodu, 53
 instrukcja
 warunkowa, 212, 214, 224, 229, 338
 wyjścia, 224
 interfejs, 308
 EventListener, 175
 Observer/Observable, 175
 opublikowany, 57
 zmiana, 57
 IOException, 87
 isNull, 237
 iteracja, 104

J

jednostkowy test, 84
 Johnson Ralph, 14
 JUnit, 79, 84

K

katalog przekształceń refaktoryzacyjnych, 89
 Kent Beck, 14
 klasa, 131, 135, 194, 299
 alternatywna, 73
 biblioteczna, 74
 duża, 67
 kliencka, 143
 kryterium usuwania, 71
 leniwa, 71
 opakowująca, 74, 145, 147
 rozdzielanie, 73
 samotestująca, 77
 scalenie, 311
 serwerowa, 143, 145
 klient, 138, 141, 247
 kod
 bezpieczne przekształcanie, 22
 błędu, 280
 czyszczenie, 50
 czytelny, 26, 50, 51, 61
 inspekcja, 53
 przyspieszenie wytwarzania, 52
 samotestujący, 77
 wartościowy, 26
 wyodrębnianie do metody, 67
 zarządzanie, 22
 zduplikowany, 66
 zrozumiały, 118
 kolekcja, 185
 komentarz, 67, 75
 komunikat,
 łańcuch, 72
 o błędzie, 82
 konstruktor, 120, 147, 274, 294
 chroniony, 207
 krótka metoda, 94, 100

L

lista parametrów, skracanie, 263
 logika
 dziedzinowa, 334
 warunkowa, 39, 211

Ł

łańcuch komunikatów, 72

M

metoda, 118, 207, 247, 263
 długa, 66
 docelowa, 125
 dziedziczenie, 74
 krótka, 94, 100
 nazwa, 245
 prywatna, 273
 skracanie, 66
 statement, 20, 21
 szablonowa, 312, 313, 318
 testowanie, 28
 wielkość, 94
 wytwórcza, 274
 zazdrosna, 69
 zbyt długa, 93
 źródłowa, 125
 migracja danych, 57
 model, 334
 Model-Widok-Kontroler, 334

N

nadklasa, 207, 290, 291, 294, 297, 304
 nadmiarowa zmienna, 30
 narzędzia
 integracja, 363
 refaktoryzacyjne, 352
 kryteria praktyczne, 362
 wymagania, 360
 nazwa metody, 245
 nieobiektowa baza danych, 57
 niezmiennosc obiektu wartości, 163
 nowa funkcjonalność, 21, 53
 null, 233, 236, 237

O

obiekt, 263, 266
 metody, 72
 pusty, 233
 referencyjny, 158, 159, 163
 reprezentujący dane, 158
 stan, 251
 wartości, 159, 163
 niezmiennosc, 163

Oddzielenie Dziedziny od Prezentacji, 334
 instrukcja, 335
 motywacja, 334
 przykład, 335
 odwołań odnajdowanie, 28, 90
 okienko GUI, 171
 Opdyke Bill, 14
 optymalizacja wydajności, 50
 Order, 157
 overdraftCharge, 126

P

parametr, 114, 247, 249, 263, 266
 długa lista, 68
 Parametryzacja Metody, 255, 257
 instrukcja, 255
 motywacja, 255
 przykład, 255
 parsowanie, 361
 pętla, 104, 111
 pisanie testów, 82
 podklasa, 145, 146, 207, 290, 291, 294, 297,
 298, 299
 Podział Zmiennej Tymczasowej, 111
 instrukcja, 111
 motywacja, 111
 przykład, 112
 pole, 128, 298
 nadklasy, 207
 prywatne, 184
 publiczne, 184
 polimorfizm, 39, 70, 199, 229
 wywołanie, 42
 pośrednik, 55, 73
 prezentacja, 334
 programowanie ekstremalne, 62
 projektowanie z góry, 59
 prosta delegacja, 138
 proste okienko GUI, 171
 Przekazanie Całego Obiektu, 260
 instrukcja, 261
 motywacja, 260
 przykład, 261
 przekazywanie parametrów przez wartość, 116
 Przekształcenie Projektu Proceduralnego
 na Obiekty, 332
 instrukcja, 333
 motywacja, 332
 przykład, 333
 przekształcenie refaktoryzacyjne, 49, 89
 Przeniesienie
 Metody, 124
 instrukcja, 124
 motywacja, 124
 przykład, 125
 Pola, 128
 instrukcja, 128
 motywacja, 128
 przykład, 129, 130
 Przesunięcie
 Ciała Konstruktora w Górę Hierarchii,
 294
 instrukcja, 294
 motywacja, 294
 przykład, 295
 Metody w Dół Hierarchii, 297
 instrukcja, 297
 motywacja, 297
 Metody w Górę Hierarchii, 291
 instrukcja, 291
 motywacja, 291
 przykład, 292
 Pola w Dół Hierarchii, 298
 instrukcja, 298
 motywacja, 298
 Pola w Górę Hierarchii, 290
 instrukcja, 290
 motywacja, 290
 przyspieszenie wytwarzania kodu, 52
 punkt wyjścia, 226

R

refactoring, 49
 Refactoring Browser, 14, 25, 359, 362
 refaktoryzacja, 12, 13, 22, 25, 49, 59, 62,
 325, 343, 345
 bazy danych, 57
 bezpieczna, 351
 cel, 50, 366
 cofanie zmian, 363
 definicja, 49
 narzędzia, 352, 359
 podstawowa technika, 91
 potrzeba, 54
 problemy, 56, 348
 przekształcenie, 49
 skracanie czasu, 351
 szybkość, 362

warunek konieczny, 77
 zakończenie, 365
 zyski, 349
 reference object, 158
 return, 219
 reużycie, 345
 Roberts Don, 14
 rozbieżne zmiany, 68
 rozdzielanie klas, 73
 Rozdzielenie Zapytania i Modyfikacji, 251
 instrukcja, 251
 motywacja, 251
 przykład, 252
 współbieżność, 254
 Rozplątanie Hierarchii Dziedziczenia, 327
 instrukcja, 328
 motywacja, 327
 przykład, 328
 rozwiązanie elastyczne, 60
 równoległe hierarchie dziedziczenia, 71
 rzutowanie, 278

S

Samoenkapsulacja Pola, 153
 instrukcja, 153
 motywacja, 153
 przykład, 154
 samotestująca klasa, 77
 samotestujący kod, 77
 Scalenie Instrukcji Warunkowej, 214
 instrukcja, 214
 motywacja, 214
 przykład alternatywa (OR, ||), 215
 przykład koniunkcja (AND, &&), 215
 scalenie klas, 311
 Scalenie Zduplikowanych Fragmentów
 Instrukcji Warunkowej, 217
 instrukcja, 217
 motywacja, 217
 przykład, 218
 send, 218
 setCourses, 191
 składowe, 299, 304
 skracanie
 listy parametrów, 263
 metody, 66
 słuchacze zdarzeń, 175
 Smalltalk, 25
 spekulacyjne uogólnienia, 71

stada danych, 69
 stała, 183
 stan, 202
 strategia, 202
 styl prezentacji, 330
 switch, 42, 70, 231, 275

Ś

ścieżka wykonania, 224

T

tablica, 166
 TelephoneNumber, 136
 test
 automatyczny, 78
 funkcjonalny, 84
 jednostkowy, 79, 84
 metody, 28
 pisanie, 82
 TestCase, 81

U

Ukrycie
 Delegata, 138
 instrukcja, 139
 motywacja, 138
 przykład, 139
 Metody, 273
 instrukcja, 273
 motywacja, 273
 uogólnienia spekulacyjne, 71
 Usunięcie
 Flagi Kontrolnej, 219
 instrukcja, 219
 motywacja, 219
 przykład, 220, 221
 Metody Ustawiającej Wartość, 270
 instrukcja, 270
 motywacja, 270
 przykład, 270
 Parametru, 249
 instrukcja, 249
 motywacja, 249
 Pośrednika, 141
 instrukcja, 141
 motywacja, 141
 przykład, 142

Metody Szablonowej, 312
 instrukcja, 313
 motywacja, 312
 przykład, 313

V

value, 318
 objects, 158

W

Ward Cunningham, 14
 wartościowy kod, 26
 wartość, 251
 warunki graniczne, 85
 Wchłonięcie
 Klasy, 135
 instrukcja, 135
 motywacja, 135
 przykład, 135
 Metody, 100
 instrukcja, 100
 motywacja, 100
 Zmiennej Tymczasowej, 102
 instrukcja, 102
 motywacja, 102
 wektor, 191
 widok, 334
 wprowadzanie zmian, 25
 Wprowadzenie
 Asercji, 240
 instrukcja, 240
 motywacja, 240
 przykład, 241
 Obcej Metody, 143
 instrukcja, 143
 motywacja, 143
 przykład, 144
 Obiektu Parametrycznego, 266
 instrukcja, 266
 motywacja, 266
 przykład, 267
 Obiektu Pustego, 233
 instrukcja, 234
 motywacja, 233
 przykład, 235, 238
 przypadki specjalne, 239

Rozszerzenia Lokalnego, 145
 instrukcja, 146
 motywacja, 145
 przykład, 146, 147
 Zmiennej Objaśniającej, 107
 instrukcja, 107
 motywacja, 107
 przykład, 108
 wskaźniki wsteczne, 176
 współbieżność, 254
 wydajność, 37, 61
 gorące punkty, 62
 optymalizacja, 50
 wyjątek, 87, 280
 wykrywanie błędów, 52
 wywołanie polimorficzne, 42
 wzrost produktywności, 78

Z

Zamiana

Asocjacji Dwukierunkowej
 na Jednokierunkową, 179
 instrukcja, 179
 motywacja, 179
 przykład, 180
 Asocjacji Jednokierunkowej
 na Dwukierunkową, 176
 instrukcja, 176
 motywacja, 176
 przykład, 177
 Referencji na Wartość, 163
 instrukcja, 163
 motywacja, 163
 przykład, 164
 Wartości na Referencję, 159
 instrukcja, 159
 motywacja, 159
 przykład, 160
 zarządzanie kodem, 22
 Zastąpienie
 Algorytmu, 121
 instrukcja, 122
 motywacja, 121
 Delegacji Dziedziczeniem, 322
 instrukcja, 322
 motywacja, 322
 przykład, 323

- Dziedziczenia Delegacją, 319
 - instrukcja, 319
 - motywacja, 319
 - przykład, 320
- Instrukcji Warunkowej Polimorfizmem, 229
 - instrukcja, 230
 - motywacja, 229
 - przykład, 230
- Kodu Błędu Wyjątkiem, 280
 - instrukcja, 280
 - motywacja, 280
 - przykład, 281
 - przykład, wyjątek kontrolowany, 282
 - przykład, wyjątek niekontrolowany, 282
- Kodu Typu Klasą, 194
 - instrukcja, 194
 - motywacja, 194
 - przykład, 195
- Kodu Typu Podklasami, 199
 - instrukcja, 200
 - motywacja, 199
 - przykład, 200
- Kodu Typu Wzorcem Stan lub Strategia, 202
 - instrukcja, 202
 - motywacja, 202
 - przykład, 203
- Konstruktora Metodą Wytwórczą, 274
 - instrukcja, 274
 - motywacja, 274
 - przykład, 274, 275, 277
- Magicznej Liczby Stałą Symboliczną, 183
 - instrukcja, 183
 - motywacja, 183
- Metody Obiektem, 118
- Parametru Metodami
 - o Różnych Nazwach, 257
 - instrukcja, 258
 - motywacja, 257
 - przykład, 258
- Parametru Metodą, 263
 - instrukcja, 263
 - motywacja, 263
 - przykład, 264
- Podklasy Polami, 207
 - instrukcja, 207
 - motywacja, 207
 - przykład, 208
- Rekordu Klasą z Danymi, 193
 - instrukcja, 193
 - motywacja, 193
- Tablicy Obiektem, 166
 - instrukcja, 166
 - motywacja, 166
 - przykład, 167
- Typu Prostego Obiektem, 156
 - instrukcja, 156
 - motywacja, 156
 - przykład, 157
- Wyjątku Testem, 285
 - instrukcja, 285
 - motywacja, 285
 - przykład, 285
- Zagnieżdżonej Instrukcji Warunkowej Instrukcją Wyjścia, 224
 - instrukcja, 225
 - motywacja, 224
 - przykład, 225
 - przykład, odwrócenie warunków, 226
- Zmiennej Tymczasowej Zapytaniem, 103
 - instrukcja, 104
 - motywacja, 103
 - przykład, 104
- zazdrosne metody, 69
- zduplikowany kod, 66
- zgłoszenie błędu, 84
- Zmiana Nazwy Metody, 245
 - instrukcja, 245
 - motywacja, 245
 - przykład, 246
- zmiany
 - rozbieżne, 68
 - wprowadzanie, 25
- zmienna, 219
 - lokalna, 118
 - nadmiarowa, 30
 - tymczasowa, 34, 103, 111, 114
 - typu wyliczeniowego, 98
- Zwinięcie Hierarchii, 311
 - instrukcja, 311
 - motywacja, 311

REFAKTORYZACJA

Ulepszanie struktury istniejącego kodu

KANON INFORMATYKI

Jak ryzykowne jest grzebanie w kodzie – wszyscy doskonale wiemy. Im głębiej sięgasz, tym więcej pojawiają się nowych problemów i jeszcze więcej rzeczy wymaga zmian. A nieustannie „poprawianie” działającego kodu może w końcu doprowadzić do powstania trudno wykrywalnych, krytycznych błędów. Jednak co zrobić, jeśli „oddziedziczymy” nieefektywny, trudny w utrzymaniu i roszczeniu program? Jak poprawić jego strukturalną spójność i wydajność? Wypracowane 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 zbytym 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 omówienia 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 znajdują 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

helion.pl
księgarnia internetowa

Nr katalogowy: 6795

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

Zamówienia telefoniczne:
0 801 339900
0 601 339900



Helion

Sprawdź najnowsze promocje
① <http://helion.pl/promocje>
Książki najchętniej czytane
② <http://helion.pl/bestseller>
Zamów informacje o nowościach
③ <http://helion.pl/nowosci>

Helion SA
ul. Koszalski 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: helion@helion.pl
<http://helion.pl>

sięgnij po **WIĘCEJ**



KOD KORZYŚCI

ISBN 978-83-246-3243-5



Cena 59,00 zł

Informatyka w najlepszym wydaniu