WYDANIE II

Refaktoryzacja

Ulepszanie struktury istniejącego kodu

Martin Fowler

we współpracy
z Kentem Beckiem

Helion

Tytuł oryginału: Refactoring: Improving the Design of Existing Code (2nd Edition)
(Addison-Wesley Signature Series (Fowler))

Tłumaczenie: Andrzej Watrak, z wykorzystaniem fragmentów książki „Refaktoryzacja. Ulepszanie struktury istniejącego kodu" w tłumaczeniu Justyny Walkowskiej.

ISBN: 978-83-283-5563-7

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/refak2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

Słowo wstępne do pierwszego wydania **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 15

1. Refaktoryzacja: pierwszy przykład **17**
Punkt wyjścia 17
Uwagi na temat przykładowego programu 19
Pierwszy krok refaktoryzacji 20
Dekompozycja funkcji statement 21
Aktualny stan: mnóstwo zagnieżdżonych funkcji 37
Rozdzielenie faz obliczeń i formatowania 39
Aktualny stan: podział na dwa pliki (i fazy) 46
Uporządkowanie obliczeń według typów przedstawień 48
Aktualny stan: tworzenie danych za pomocą polimorficznego kalkulatora 55
Podsumowanie 57

2. Zasady refaktoryzacji **59**
Definicja refaktoryzacji 59
Dwa kapelusze 60
Po co refaktoryzować? 60
Kiedy refaktoryzować? 63
Problemy z refaktoryzacją 68
Refaktoryzacja, architektura i yagni 73
Refaktoryzacja i szerszy proces tworzenia oprogramowania 74
Refaktoryzacja a wydajność 75
Skąd się wzięła refaktoryzacja? 77
Refaktoryzacja automatyczna 78
Dalsze kroki 80

3. Brzydkie zapaszki w kodzie 81
Tajemnicza nazwa 82
Zduplikowany kod 82
Długa funkcja 82
Długa lista parametrów 83
Dane globalne 84
Dane mutowalne 84
Rozbieżne zmiany 85
Fala uderzeniowa 85
Zazdrosne funkcjonalności 86
Stada danych 86
Opętanie typami prostymi 87
Powtarzane instrukcje warunkowe 87
Pętle 88
Leniwa klasa 88
Spekulacyjne uogólnienia 88
Pole tymczasowe 89
Łańcuchy komunikatów 89
Pośrednik 89
Niestosowna bliskość 90
Duża klasa 90
Alternatywne klasy z różnymi interfejsami 91
Klasa danych 91
Odmowa przyjęcia spadku 91
Uwagi 92

4. Testy 93
Zalety samotestującego się kodu 93
Prosty kod do przetestowania 95
Pierwszy test 97
Dodanie następnego testu 100
Modyfikacja danych początkowych 102
Sprawdzanie warunków granicznych 102
Dalsze kroki 105

5. Katalog przekształceń refaktoryzacyjnych 107
Format opisu przekształceń 107
Wybór przekształceń 108

6. Pierwszy pakiet przekształceń 109
Ekstrakcja Funkcji 110
Wchłonięcie Funkcji 118
Ekstrakcja Zmiennej 122
Wchłonięcie Zmiennej 126
Zmiana Deklaracji Funkcji 127
Enkapsulacja Zmiennej 134
Zmiana Nazwy Zmiennej 139
Wprowadzenie Obiektu Parametrycznego 142
Zebranie Funkcji w Klasę 146
Zebranie Funkcji w Transformatę 151
Podział na Fazy 156

7. Enkapsulacja 161
Enkapsulacja Rekordu 162
Enkapsulacja Kolekcji 170
Zastąpienie Typu Prostego Obiektem 174
Zastąpienie Zmiennej Tymczasowej Zapytaniem 178
Ekstrakcja Klasy 182
Wchłonięcie Klasy 186
Ukrycie Delegata 189
Usunięcie Pośrednika 192
Zastąpienie Algorytmu 195

8. Przenoszenie funkcjonalności 197
Przeniesienie Funkcji 198
Przeniesienie Pola 206
Przeniesienie Instrukcji do Funkcji 211
Przeniesienie Instrukcji do Kodu Wywołującego 215
Zastąpienie Wchłoniętego Kodu Wywołaniem Funkcji 220
Przesunięcie Instrukcji 221
Podział Pętli 225
Zastąpienie Pętli Potokiem 229
Usunięcie Martwego Kodu 234

9. Porządkowanie danych 235
Podział Zmiennej 236
Zmiana Nazwy Pola 240
Zastąpienie Wyliczanej Zmiennej Zapytaniem 244
Zamiana Referencji na Wartość 248
Zamiana Wartości na Referencję 252

10. Upraszczanie wyrażeń warunkowych 255
Dekompozycja Instrukcji Warunkowej 256
Scalenie Instrukcji Warunkowej 259
Zastąpienie Zagnieżdżonej Instrukcji Warunkowej Instrukcją Wyjścia 262
Zastąpienie Instrukcji Warunkowej Polimorfizmem 267
Wprowadzenie Przypadku Specjalnego 283
Wprowadzenie Asercji 296

11. Refaktoryzacja interfejsu API 299
Rozdzielenie Zapytania i Modyfikacji 300
Parametryzacja Funkcji 303
Usunięcie Parametru-Flagi 307
Przekazanie Całego Obiektu 312
Zastąpienie Parametru Zapytaniem 317
Zastąpienie Zapytania Parametrem 320
Usunięcie Funkcji Ustawiającej Wartość 324
Zastąpienie Konstruktora Funkcją Wytwórczą 327
Zastąpienie Funkcji Poleceniem 330
Zastąpienie Polecenia Funkcją 336

12. Praca z hierarchią klas 341
Przesunięcie Metody w Górę Hierarchii 342
Przesunięcie Pola w Górę Hierarchii 345
Przesunięcie Ciała Konstruktora w Górę Hierarchii 347
Przesunięcie Metody w Dół Hierarchii 351
Przesunięcie Pola w Dół Hierarchii 352
Zastąpienie Kodu Typu Podklasami 353
Usunięcie Podklasy 360
Ekstrakcja Nadklasy 366
Zwinięcie Hierarchii 371
Zastąpienie Podklasy Delegatem 372
Zastąpienie Nadklasy Delegatem 390

Bibliografia 395

Skorowidz 398

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 otwiera 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 dla Ciebie zbyt skomplikowany. 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 klasycznie w 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

W pierwszym wydaniu książki pierwszy program wyświetlał rachunek za wypożyczenie filmów wideo. Dzisiaj wielu z czytelników mogłoby zapytać: „Co to jest wypożyczalnia filmów?". Zamiast odpowiadać na to pytanie, postanowiłem przekształcić przykład w coś tradycyjnego, a jednocześnie bardziej nowoczesnego.

Wyobraź sobie firmę zatrudniającą aktorów teatralnych do grania w różnych przedstawieniach. Klienci zamawiają przedstawienia, a firma wystawia rachunki uzależnione od przedstawienia i liczebności publiki. W tej chwili do wyboru są dwa rodzaje przedstawień: tragedie i komedie.

Firma nie tylko wystawia rachunki, ale przyznaje też „punkty promocyjne", które można wymieniać na rabaty za przyszłe przedstawienia. Jest to coś w rodzaju programu lojalnościowego.

Aktorzy zapisują dane przedstawień w zwykłym pliku JSON, wyglądającym mniej więcej tak:

Plik *plays.json*

```
{
  "hamlet": {"name": "Hamlet", "type": "tragedia"},
  "as-like": {"name": "Jak wam się podoba", "type": "komedia"},
  "othello": {"name": "Otello", "type": "tragedia"}
}
```

Dane do rachunku też są zapisane w pliku JSON:

Plik *invoices.json*

```
{
  "customer": "BigCo",
  "performances": [
    {
      "playID": "hamlet",
      "audience": 55
    },
    {
      "playID": "as-like",
      "audience": 35
    },
    {
      "playID": "othello",
      "audience": 40
    }
  ]
}
```

Kod wyświetlający rachunek jest prostą funkcją:

```
function statement(invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Rachunek dla ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("pl-PL", {
    style: "currency",
    currency: "PLN",
    minimumFractionDigits: 2
  }).format;

  for (let perf of invoice.performances) {
    const play = plays[perf.playID];
    let thisAmount = 0;

    switch (play.type) {
      case "tragedia":
        thisAmount = 40000;
        if (perf.audience > 30) {
```

```
          thisAmount += 1000 * (perf.audience - 30);
        }
        break;
      case "komedia":
        thisAmount = 30000;
        if (perf.audience > 20) {
          thisAmount += 10000 + 500 * (perf.audience - 20);
        }
        thisAmount += 300 * perf.audience;
        break;
      default:
        throw new Error(`Nieznany typ przedstawienia: ${play.type}`);
    }

    // Przyznanie punktów promocyjnych.
    volumeCredits += Math.max(perf.audience - 30, 0);
    // Przyznanie dodatkowego punktu promocyjnego za każdych 5 widzów komedii.
    if ("komedia" === play.type)
      volumeCredits += Math.floor(perf.audience / 5);

    // Utworzenie wiersza rachunku.
    result += ` ${play.name}: ${format(thisAmount/100)} (liczba miejsc:
    ↪${perf.audience})\n`;
    totalAmount += thisAmount;
  }
  result += `Należność: ${format(totalAmount/100)}\n`;
  result += `Punkty promocyjne: ${volumeCredits}\n`;
  return result;
}
```

Powyższy kod po przetworzeniu przykładowych plików wyświetla następujący wynik:

```
Rachunek dla SuperFirma
 Hamlet: 650,00 zł (liczba miejsc: 55)
 Jak wam się podoba?: 580,00 zł (liczba miejsc: 35)
 Otello: 500,00 zł (liczba miejsc: 40)
Należność: 1 730,00 zł
Punkty promocyjne: 47
```

Uwagi na temat przykładowego programu

Co sądzisz o powyższym programie? Moim zdaniem jest napisany przyzwoicie. Tak krótki program nie musi mieć głębokiej struktury, aby był zrozumiały. Pamiętaj jednak, że starałem się, żeby przykłady były krótkie. Wyobraź sobie większy program składający się z setek wierszy. Wtedy jedna ciągła funkcja byłaby nieczytelna.

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 ta kwestia 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.

Dlatego gdybym musiał wprowadzić zmiany w programie składającym się z setek wierszy, najpierw nadałbym mu strukturę zbioru funkcji i innych elementów, dzięki której łatwiej byłoby mi zrozumieć, co program robi. Jeżeli kod nie ma takiej struktury, wtedy zazwyczaj wolę najpierw mu ją nadać, a dopiero potem wprowadzać zmiany.

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.

W tym przypadku jest kilka zmian, które można wprowadzić. Przede wszystkim należałoby dodać instrukcje umożliwiające wyświetlanie informacji w formacie HTML. Zastanówmy się nad konsekwencjami takiej zmiany. Trzeba byłoby wszystkie wiersze, w których do zmiennej `result` dołączane są ciągi znaków, opatrzyć instrukcjami warunkowymi. Cała funkcja stałaby się wtedy o wiele bardziej skomplikowana. W takich sytuacjach większość programistów kopiuje fragment programu do osobnej funkcji, która zwraca kod HTML. Operacja ta wprawdzie nie jest uciążliwa, ale może skutkować pojawieniem się innych problemów w przyszłości. Wszelkie zmiany w algorytmie wyliczania należności trzeba byłoby wtedy spójnie implementować w dwóch miejscach. Gdyby chodziło o program, którego algorytm na pewno pozostanie niezmienny, takie kopiowanie i wklejanie kodu byłoby do przyjęcia. Ale jeżeli program ma być użyteczny w dłuższej perspektywie, to duplikowanie jego fragmentów kodu jest porażką.

Oprócz tego pojawia się perspektywa innej zmiany. Aktorzy mogą planować wprowadzenie nowych rodzajów przedstawień, np. historycznych, dokumentalnych, historyczno-dokumentalnych, dramatyczno-historycznych, komediowo-dramatyczno-historycznych, poetyckich i różnych innych. Jednak nie zdecydowali jeszcze, co będą grać i kiedy. Te zmiany wpłyną zarówno na sposób wyliczania ceny przedstawień, jak również na liczby przyznawanych punktów promocyjnych. Jako doświadczony programista jestem pewien, że jakikolwiek schemat zostałby przyjęty, będzie on zmieniony w ciągu sześciu miesięcy. Ponadto potrzeby nowych funkcjonalności nie są zgłaszane pojedynczo, tylko całymi gromadami.

W takiej sytuacji zmiany wynikające z wprowadzenia nowych przedstawień i zasad ich wyceny należałoby również wprowadzać w funkcji `statement`. Kopiując kod z funkcji `statement` do `htmlStatement`, należałoby pilnować, aby zamiany były spójne. Niestety, wraz ze wzrostem złożoności tych zasad coraz trudniej będzie zarówno określać miejsca, w których należy wprowadzać zmiany, jak i unikać pomyłek.

Chciałem tutaj pokazać, jak planowane zmiany wymuszają potrzebę refaktoryzacji. Jeżeli kod działa poprawnie i nigdy nie będzie zmieniany, całkowicie uzasadnione jest pozostawienie go w spokoju. Kod warto oczywiście usprawniać, ale jeżeli programista go nie rozumie, to nie jest to duży problem. Jednak gdy pojawi się potrzeba poznania, jak funkcjonuje program, i okaże się to trudnym zadaniem, wtedy trzeba będzie coś z tym kodem zrobić.

Pierwszy krok refaktoryzacji

Przystępując do refaktoryzacji, zawsze zaczynam od tego samego kroku. Tworzę solidny zestaw testów do 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 to rzecz ludzka, dlatego rozsądnie jest wesprzeć się testami.

Jako że funkcja `statement` zwraca łańcuch znaków, tworzę kilka obiektów reprezentujących przedstawienia różnych rodzajów i generuję łańcuchy z rachunkami. Następnie porównuję wynikowe łańcuchy znaków z ręcznie wprowadzonymi łańcuchami referencyjnymi. Używając odpowiedniej platformy, przygotowuję testy, które w swoim środowisku programistycznym uruchamiam kilkoma kliknięciami. Ich wykonanie trwa tylko parę sekund, ale zobaczysz, że uruchamiam je naprawdę często.

Ważnym elementem testów jest sposób raportowania wyników. Są one albo koloru zielonego oznaczającego, że wszystkie uzyskane łańcuchy są zgodne z referencyjnymi, albo czerwonego. W tym drugim przypadku pojawia się lista wierszy, które się różnią. Ważne jest, aby testy były samosprawdzalne. Jeżeli takie nie są, wtedy trzeba poświęcać wiele czasu na ręczne sprawdzanie uzyskanych wyników z referencjami, co bardzo spowalnia pracę. Nowoczesne platformy testowe oferują wszelkie funkcjonalności potrzebne do tworzenia i uruchamiania samosprawdzalnych testów.

Podczas refaktoryzowania kodu polegam na testach. Traktuję je jak detektory błędów chroniące mnie przed własnymi pomyłkami. Gdy piszę coś dwukrotnie, tj. raz w kodzie i drugi raz w teście, wtedy aby oszukać detektor musiałbym popełnić błąd w obu miejscach. Podwójna kontrola pracy zmniejsza prawdopodobieństwo zrobienia czegoś źle. Choć pisanie testów wymaga poświęcenia dodatkowego czasu, pozwala go sporo zaoszczędzić podczas debugowania kodu.

Testy mają tak istotne znaczenie dla procesu refaktoryzacji, że zdecydowałem się poświęcić im cały rozdział 4. (Testy).

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

Dekompozycja funkcji statement

Przed refaktoryzacją tak długiej funkcji jak nasza, staram się w głowie określić miejsca, w których można podzielić kod na osobne zadania. Pierwszą rzeczą, która wpadła mi w oko, jest instrukcja `switch` znajdująca się w środku funkcji.

```
function statement(invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Rachunek dla ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("pl-PL", {
    style: "currency",
    currency: "PLN",
    minimumFractionDigits: 2
  }).format;

  for (let perf of invoice.performances) {
    const play = plays[perf.playID];
    let thisAmount = 0;

    switch (play.type) {
      case "tragedia":
        thisAmount = 40000;
        if (perf.audience > 30) {
          thisAmount += 1000 * (perf.audience - 30);
        }
```

```
        break;
      case "komedia":
        thisAmount = 30000;
        if (perf.audience > 20) {
          thisAmount += 10000 + 500 * (perf.audience - 20);
        }
        thisAmount += 300 * perf.audience;
        break;
      default:
        throw new Error(`Nieznany typ przedstawienia: ${play.type}`);
    }

    // Przyznanie punktów promocyjnych.
    volumeCredits += Math.max(perf.audience - 30, 0);
    // Przyznanie dodatkowego punktu promocyjnego za każdych 5 widzów komedii.
    if ("komedia" === play.type)
      volumeCredits += Math.floor(perf.audience / 5);

    // Utworzenie wiersza rachunku.
    result += ` ${play.name}: ${format(thisAmount/100)} (liczba miejsc: ${perf.audience})\n`;
    totalAmount += thisAmount;
  }
  result += `Należność: ${format(totalAmount/100)}\n`;
  result += `Punkty promocyjne: ${volumeCredits}\n`;
  return result;
}
```

Patrząc na ten fragment, dochodzę do wniosku, że oblicza on cenę jednego przedstawienia. Wynika to z analizy kodu, ale jak powiedział Ward Cunningham: wniosek ten istnieje tylko w mojej głowie, która jest wyjątkowo nietrwałym nośnikiem informacji. Dlatego muszę ten wniosek utrwalić, przenosząc go z głowy do kodu. Dzięki temu gdy wrócę kiedyś do tego programu, on sam mi powie, co robi, i nie będę musiał powtarzać wszystkiego od nowa.

Jednym ze sposobów zapisania wniosku w programie jest przekształcenie fragmentu kodu w funkcję o nazwie opisującej jej działanie, np. `amountFor(aPerformance)` („opłata za przedstawienie"). Gdy muszę wykonać operację taką jak ta, stosuję procedurę, która minimalizuje możliwość popełnienia błędu. Tę procedurę zanotowałem sobie pod nazwą Ekstrakcja Funkcji (s. x).

Najpierw szukam we fragmencie kodu zmiennych, które po wyekstrahowaniu funkcji nie będą już potrzebne. W tym przypadku są trzy takie zmienne: `perf`, `play` i `thisAmount`. Pierwsze dwie będą wykorzystywane w wyodrębnionym kodzie, ale ich wartości nie będą zmieniane, więc mogę je przekształcić w parametry funkcji. Z modyfikowanymi zmiennymi jest więcej zachodu. Tutaj jest tylko jedna, więc jej wartość będzie zwracana jako wynik. Dodatkowo instrukcję inicjującą można przenieść do wyodrębnionego kodu. W efekcie powstanie następująca funkcja:

Funkcja `statement`...

```
function amountFor(perf, play) {
  let thisAmount = 0;
  switch (play.type) {
    case "tragedia":
      thisAmount = 40000;
      if (perf.audience > 30) {
        thisAmount += 1000 * (perf.audience - 30);
      }
      break;
```

```
    case "komedia":
      thisAmount = 30000;
      if (perf.audience > 20) {
        thisAmount += 10000 + 500 * (perf.audience - 20);
      }
      thisAmount += 300 * perf.audience;
      break;
    default:
      throw new Error(`Nieznany typ przedstawienia: ${play.type}`);
  }
  return thisAmount;
}
```

Nagłówki zapisane pochyłą czcionką, takie jak *Funkcja* `jakaśNazwa`... oznaczają, że znajdujący się niżej fragment kodu jest umieszczony wewnątrz funkcji, pliku lub klasy o podanej nazwie. Zazwyczaj oprócz prezentowanego kodu w danym zakresie istnieje jeszcze inny kod, którego na razie nie opisuję.

Teraz funkcja `statement` wywołuje funkcję `thisAmount`.

Najwyższy poziom...

```
function statement(invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Rachunek dla ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("pl-PL", {
    style: "currency",
    currency: "PLN",
    minimumFractionDigits: 2
  }).format;
  for (let perf of invoice.performances) {
    const play = plays[perf.playID];
    let thisAmount = amountFor(perf, play);

    // Przyznanie punktów promocyjnych.
    volumeCredits += Math.max(perf.audience - 30, 0);
    // Przyznanie dodatkowego punktu promocyjnego za każdych 5 widzów komedii.
    if ("komedia" === play.type)
      volumeCredits += Math.floor(perf.audience / 5);

    // Utworzenie wiersza rachunku.
    result += ` ${play.name}: ${format(thisAmount/100)} (liczba miejsc: ${perf.audience})\n`;
    totalAmount += thisAmount;
  }
  result += `Należność: ${format(totalAmount/100)}\n`;
  result += `Punkty promocyjne: ${volumeCredits}\n`;
  return result;
```

Po wprowadzeniu zmian takich jak powyższe natychmiast kompiluję kod i testuję go, aby sprawdzić, czy czegoś nie zepsułem. Testowanie po refaktoryzacji to prosty, ale cenny nawyk. Łatwo jest popełnić błąd, przynajmniej w moim przypadku. Dzięki testowaniu każdej zmiany w razie pomyłki mam tylko niewielki fragment kodu do sprawdzenia, łatwiej jest mi znaleźć błąd i go poprawić. To jest esencja procesu refaktoryzacji: wprowadzanie małych zmian i testowanie kodu po

każdej zmianie. Gdyby zmian było dużo, wtedy w przypadku pomyłki musiałbym przeprowadzać żmudną, zajmującą dużo czasu diagnostykę. Małe zmiany z szybkimi rezultatami mają kluczowe znaczenie i pozwalają uniknąć takiego zamieszania.

> Słowo „kompilacja" w tym kontekście oznacza wykonanie wszystkich operacji niezbędnych do uruchomienia kodu JavaScript. Ponieważ kod w tym języku uruchamia się wprost, kompilacja nic nie oznacza. Czasami jednak trzeba np. przenieść kod do odpowiedniego katalogu lub użyć kompilatora Babel.

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

Ponieważ używam języka JavaScript, mogę funkcję `amountFor` zagnieździć w funkcji `statement`. Jest to wygodne rozwiązanie — dzięki niemu nie trzeba w parametrach nowej funkcji umieszczać danych wykorzystywanych tylko przez funkcję nadrzędną. W tym konkretnym przykładzie nie ma to znaczenia, ale często dzięki temu jest o jeden problem mniej do rozwiązania.

W tym przypadku test się powiódł, więc moim kolejnym krokiem było zatwierdzenie zmian w lokalnym systemie kontroli wersji. Używam systemów *git* i *mercurial* oferujących funkcjonalność prywatnych zatwierdzeń. Zmiany zatwierdzam po każdej udanej refaktoryzacji, dzięki czemu mogę szybko przywrócić kod do stanu używalności, gdybym narobił w nim bałaganu. Co jakiś czas zbieram kilka mniejszych prywatnych zatwierdzeń w jedno duże i przesyłam je do współdzielonego repozytorium.

Ekstrakcja Funkcji jest często automatyzowaną techniką refaktoryzacyjną. W środowisku Java odruchowo naciskam sekwencję klawiszy uruchamiającą to przekształcenie. Tutaj jednak używam języka JavaScript, dla którego nie ma uniwersalnego narzędzia przeznaczonego do tego celu. Dlatego wszystko trzeba robić ręcznie. Nie jest to trudne, ale należy uważać na zmienne lokalne.

Po wyekstrahowaniu funkcji przyjrzałem się efektom i sprawdziłem, czy są jakieś proste i szybkie rzeczy, które poprawiłyby czytelność nowej funkcji. Pierwsze, co zrobiłem, to zmieniłem nazwy niektórych zmiennych na bardziej zrozumiałe, np. `thisAmount` na `result`.

Funkcja `statement`...

```
function amountFor(perf, play) {
  let result = 0;
  switch (play.type) {
    case "tragedia":
      result = 40000;
      if (perf.audience > 30) {
        result += 1000 * (perf.audience - 30);
      }
      break;
    case "komedia":
      result = 30000;
      if (perf.audience > 20) {
        result += 10000 + 500 * (perf.audience - 20);
      }
      result += 300 * perf.audience;
      break;
    default:
      throw new Error(`Nieznany typ przedstawienia: ${play.type}`);
  }
  return result;
}
```

Mam zwyczaj nadawania nazwy `result` zmiennej zawierającej zwracany wynik. Dzięki temu wiem, jaką rolę odgrywa ta zmienna. Po zmianie jak zawsze kompiluję kod, testuję go i zatwierdzam, po czym to samo, co poprzednio, robię z pierwszym parametrem funkcji.

Funkcja `statement`...

```
function amountFor(aPerformance, play) {
  let result = 0;
  switch (play.type) {
    case "tragedia":
      result = 40000;
      if (aPerformance.audience > 30) {
        result += 1000 * (aPerformance.audience - 30);
      }
      break;
    case "komedia":
      result = 30000;
      if (aPerformance.audience > 20) {
        result += 10000 + 500 * (aPerformance.audience - 20);
      }
      result += 300 * aPerformance.audience;
      break;
    default:
      throw new Error(`Nieznany typ przedstawienia: ${play.type}`);
  }
  return result;
}
```

Powtórzę, to jest mój styl kodowania. W dynamicznie typowanym języku, takim jak JavaScript, warto umieszczać w kodzie informacje o typach danych. Dlatego domyślna nazwa parametru funkcji zawiera nazwę jego typu. Ponadto jeżeli nazwa nie zawiera informacji o roli parametru, umieszczam na jego początku angielski przedimek nieokreślony (*a*). Tej konwencji nauczył mnie Kent Beck. Stosuję ją, ponieważ uważam, że jest przydatna.

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

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 się wspomóc narzędziami, które w statycznie typowanych językach pokazują wszystkie wystąpienia niezmienionych nazw. Dodatkowo za pomocą zautomatyzowanych narzędzi refaktoryzujących modyfikowanie nazw nawet szeroko stosowanych funkcji jest trywialnie proste.

Kolejna zmiana nazwy, nad którą trzeba się zastanowić, dotyczy parametru `play`. Mam jednak inny koncept.

Usunięcie zmiennej play

Zastanawiając się nad parametrem funkcji `amountFor`, szukam źródła jego wartości. W tym parametrze dane umieszcza użyta w pętli zmienna przyjmująca w każdej iteracji inne wartości. Jednak wartość zmiennej `play` jest określana za pomocą obiektu opisującego przedstawienie, więc nie trzeba

w ogóle jej umieszczać w parametrze funkcji. Tę wartość można zdefiniować wewnątrz funkcji `amountFor`. Dzieląc długą funkcję na części, lubię usuwać zmienne takie jak `play`, ponieważ stanowią one tymczasowe, lokalne elementy, które komplikują ekstrakcję. Refaktoryzacja, którą teraz zastosuję, nosi nazwę Zastąpienie Zmiennej Tymczasowej Zapytaniem (s. x).

Zacznę od wyodrębnienia kodu znajdującego się po prawej stronie instrukcji przypisania i przekształcę go w funkcję.

Funkcja `statement`...

```
function playFor(aPerformance) {
  return plays[aPerformance.playID];
}
```

Najwyższy poziom...

```
function statement(invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Rachunek dla ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("pl-PL", {
    style: "currency",
    currency: "PLN",
    minimumFractionDigits: 2
  }).format;
  for (let perf of invoice.performances) {
    const play = playFor(perf);
    let thisAmount = amountFor(perf, play);

    // Przyznanie punktów promocyjnych.
    volumeCredits += Math.max(perf.audience - 30, 0);
    // Przyznanie dodatkowego punktu promocyjnego za każdych 5 widzów komedii.
    if ("komedia" === play.type)
      volumeCredits += Math.floor(perf.audience / 5);

    // Utworzenie wiersza rachunku.
    result += ` ${play.name}: ${format(thisAmount/100)} (liczba miejsc: ${perf.audience})\n`;
    totalAmount += thisAmount;
  }
  result += `Należność: ${format(totalAmount/100)}\n`;
  result += `Punkty promocyjne: ${volumeCredits}\n`;
  return result;
```

Wykonuję cykl kompilacja – test – zatwierdzenie, a następnie przeprowadzam refaktoryzację Wchłonięcie Zmiennej (s. x).

Najwyższy poziom...

```
function statement(invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Rachunek dla ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("pl-PL", {
    style: "currency",
```

```
    currency: "PLN",
    minimumFractionDigits: 2
  }).format;
  for (let perf of invoice.performances) {
    const play = playFor(perf);
    let thisAmount = amountFor(perf, playFor(perf));

    // Przyznanie punktów promocyjnych.
    volumeCredits += Math.max(perf.audience - 30, 0);
    // Przyznanie dodatkowego punktu promocyjnego za każdych 5 widzów komedii.
    if ("komedia" === playFor(perf).type)
      volumeCredits += Math.floor(perf.audience / 5);

    // Utworzenie wiersza rachunku.
    result += ` ${playFor(perf).name}: ${format(thisAmount/100)} (liczba miejsc:
    ↪${perf.audience})\n`;
    totalAmount += thisAmount;
  }
  result += `Należność: ${format(totalAmount/100)}\n`;
  result += `Punkty promocyjne: ${volumeCredits}\n`;
  return result;
```

Powtarzam cykl kompilacja – test – zatwierdzenie, a następnie stosuję refaktoryzację Zmiana Deklaracji Funkcji (s. x), aby usunąć parametr `play` z funkcji `amountFor`. Robię to w dwóch krokach. Najpierw wewnątrz funkcji `amountFor` umieszczam wywołanie nowej funkcji.

Funkcja `statement`...

```
function amountFor(aPerformance, play) {
  let result = 0;
  switch (playFor(aPerformance).type) {
    case "tragedia":
      result = 40000;
      if (aPerformance.audience > 30) {
        result += 1000 * (aPerformance.audience - 30);
      }
      break;
    case "komedia":
      result = 30000;
      if (aPerformance.audience > 20) {
        result += 10000 + 500 * (aPerformance.audience - 20);
      }
      result += 300 * aPerformance.audience;
      break;
    default:
      throw new Error(`Nieznany typ przedstawienia: ${playFor(aPerformance).type}`);
  }
  return result;
}
```

Wykonuję cykl kompilacja – test – zatwierdzenie, a następnie usuwam parametr funkcji.

Najwyższy poziom…

```
function statement(invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Rachunek dla ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("pl-PL", {
    style: "currency",
    currency: "PLN",
    minimumFractionDigits: 2
  }).format;
  for (let perf of invoice.performances) {
    let thisAmount = amountFor(perf, playFor(perf));

    // Przyznanie punktów promocyjnych.
    volumeCredits += Math.max(perf.audience - 30, 0);
    // Przyznanie dodatkowego punktu promocyjnego za każdych 5 widzów komedii.
    if ("komedia" === playFor(perf).type)
      volumeCredits += Math.floor(perf.audience / 5);

    // Utworzenie wiersza rachunku.
    result += ` ${playFor(perf).name}: ${format(thisAmount/100)} (liczba miejsc:
    ↪${perf.audience})\n`;
    totalAmount += thisAmount;
  }
  result += `Należność: ${format(totalAmount/100)}\n`;
  result += `Punkty promocyjne: ${volumeCredits}\n`;
  return result;
```

Funkcja `statement`…

```
function amountFor(aPerformance, play) {
  let result = 0;
  switch (playFor(aPerformance).type) {
    case "tragedia":
      result = 40000;
      if (aPerformance.audience > 30) {
        result += 1000 * (aPerformance.audience - 30);
      }
      break;
    case "komedia":
      result = 30000;
      if (aPerformance.audience > 20) {
        result += 10000 + 500 * (aPerformance.audience - 20);
      }
      result += 300 * aPerformance.audience;
      break;
    default:
      throw new Error(`Nieznany typ przedstawienia: ${playFor(aPerformance).type}`);
  }
  return result;
}
```

Ponownie wykonuję cykl kompilacja – test – zatwierdzenie.

Opisana refaktoryzacja może zaniepokoić niektórych programistów. W pierwotnej wersji kodu typ przedstawienia był wyszukiwany tylko jeden raz w każdej iteracji pętli, a teraz dzieje się tak aż trzy razy. O wpływie refaktoryzacji na wydajność kodu opowiem później. Na razie uznałem po prostu, że wprowadzone zmiany raczej nie mają znaczenia dla wydajności. Gdyby jednak miały, to o wiele łatwiej jest usprawniać dobrze zrefaktoryzowany kod.

Główną korzyścią płynącą z usunięcia lokalnych zmiennych jest uproszczenie procesu ekstrakcji, ponieważ lokalne zakresy widoczności są mniejsze. Przed wyodrębnieniem jakiejkolwiek funkcji pozbywam się lokalnych zmiennych.

Teraz, po uporządkowaniu parametrów funkcji `amountFor`, sprawdzam miejsca, w których jest ona wywoływana. Funkcja ta jest wykorzystywana do nadawania wartości tymczasowej zmiennej, która nie jest modyfikowana. Mogę więc zastosować refaktoryzację Wchłonięcie Zmiennej (s. x).

Najwyższy poziom…

```
function statement(invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Rachunek dla ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("pl-PL", {
    style: "currency",
    currency: "PLN",
    minimumFractionDigits: 2
  }).format;
  for (let perf of invoice.performances) {
    let thisAmount = amountFor(perf);

    // Przyznanie punktów promocyjnych.
    volumeCredits += Math.max(perf.audience - 30, 0);
    // Przyznanie dodatkowego punktu promocyjnego za każdych 5 widzów komedii.
    if ("komedia" === playFor(perf).type)
      volumeCredits += Math.floor(perf.audience / 5);

    // Utworzenie wiersza rachunku.
    result += ` ${playFor(perf).name}: ${format(amountFor(perf)/100)} (liczba miejsc:
    ↪${perf.audience})\n`;
    totalAmount += amountFor(perf);
  }
  result += `Należność: ${format(totalAmount/100)}\n`;
  result += `Punkty promocyjne: ${volumeCredits}\n`;
  return result;
```

Ekstrakcja punktów promocyjnych

Poniżej przedstawiona jest aktualna postać funkcji `statement`.

Najwyższy poziom…

```
function statement(invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Rachunek dla ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("pl-PL", {
    style: "currency",
```

```
    currency: "PLN",
    minimumFractionDigits: 2
  }).format;
  for (let perf of invoice.performances) {
    let thisAmount = amountFor(perf);

    // Przyznanie punktów promocyjnych.
    volumeCredits += Math.max(perf.audience - 30, 0);
    // Przyznanie dodatkowego punktu promocyjnego za każdych 5 widzów komedii.
    if ("komedia" === playFor(perf).type)
      volumeCredits += Math.floor(perf.audience / 5);

    // Utworzenie wiersza rachunku.
    result += ` ${playFor(perf).name}: ${format(amountFor(perf)/100)} (liczba miejsc:
    ↪${perf.audience})\n`;
    totalAmount += amountFor(perf);
  }
  result += `Należność: ${format(totalAmount/100)}\n`;
  result += `Punkty promocyjne: ${volumeCredits}\n`;
  return result;
```

Teraz, dzięki usunięciu zmiennej `play`, łatwiej jest wyekstrahować kod wyliczający punkty promocyjne.

W lokalnym zakresie wciąż są jednak używane dwie inne zmienne. Zmienną `perf` można łatwo umieścić w parametrze funkcji, natomiast nieco trudniej jest w przypadku zmiennej `volumeCredits`, ponieważ sumuje ona wartości w każdej iteracji pętli. Dlatego najlepszym rozwiązaniem będzie zainicjowanie jej odpowiednika wewnątrz wyodrębnionej funkcji, która będzie zwracała wartość nowej zmiennej.

Funkcja `statement`...

```
function volumeCreditsFor(perf) {
  let volumeCredits = 0;
  volumeCredits += Math.max(perf.audience - 30, 0);
  if ("komedia" === playFor(perf).type)
    volumeCredits +=   Math.floor(perf.audience / 5);
  return volumeCredits;
}
```

Najwyższy poziom...

```
function statement(invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Rachunek dla ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("pl-PL", {
    style: "currency",
    currency: "PLN",
    minimumFractionDigits: 2
  }).format;
  for (let perf of invoice.performances) {
    volumeCredits += volumeCreditsFor(perf);
```

```
    // Utworzenie wiersza rachunku.
    result += ` ${playFor(perf).name}: ${format(amountFor(perf)/100)} (liczba miejsc:
    ↪${perf.audience})\n`;
    totalAmount += amountFor(perf);
  }
  result += `Należność: ${format(totalAmount/100)}\n`;
  result += `Punkty promocyjne: ${volumeCredits}\n`;
  return result;
```

Usunąłem też niepotrzebny (a w tym przypadku nawet mylący) komentarz.

Wykonuję cykl kompilacja – test – zatwierdzenie, a następnie modyfikuję nazwy zmiennych stosowanych wewnątrz nowej funkcji.

Funkcja `statement`…

```
function volumeCreditsFor(aPerformance) {
  let result = 0;
  result += Math.max(aPerformance.audience - 30, 0);
  if ("komedia" === playFor(aPerformance).type)
    result +=   Math.floor(aPerformance.audience / 5);
  return result;
}
```

Pozornie zmiany wprowadziłem w jednym kroku, ale w rzeczywistości, tak jak poprzednio, dokonałem ich pojedynczo, wykonując za każdym razem cykl kompilacja – test – zatwierdzenie.

Usunięcie zmiennej format

Popatrzmy ponownie na główną funkcję `statement`.

Najwyższy poziom…

```
function statement(invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Rachunek dla ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("pl-PL", {
    style: "currency",
    currency: "PLN",
    minimumFractionDigits: 2
  }).format;
  for (let perf of invoice.performances) {
    volumeCredits += volumeCreditsFor(perf);

    // Utworzenie wiersza rachunku.
    result += ` ${playFor(perf).name}: ${format(amountFor(perf)/100)} (liczba miejsc:
    ↪${perf.audience})\n`;
    totalAmount += amountFor(perf);
  }
  result += `Należność: ${format(totalAmount/100)}\n`;
  result += `Punkty promocyjne: ${volumeCredits}\n`;
  return result;
```

Jak już wcześniej wspomniałem, zmienne lokalne mogą przysparzać problemów. Przydatne są tylko wewnątrz danej funkcji, dlatego prowokują do pisania długich, skomplikowanych bloków. Stąd moim następnym krokiem jest usunięcie kilku z nich. Najprościej będzie ze zmienną `format`. W tym przypadku wynik funkcji jest przypisywany zmiennej tymczasowej, którą zastąpię zadeklarowaną funkcją.

Funkcja `statement`…

```
function format(aNumber) {
  return new Intl.NumberFormat("pl-PL", {
    style: "currency",
    currency: "PLN",
    minimumFractionDigits: 2
  }).format;
}
```

Najwyższy poziom…

```
function statement(invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Rachunek dla ${invoice.customer}\n`;
  for (let perf of invoice.performances) {
    volumeCredits += volumeCreditsFor(perf);

    // Utworzenie wiersza rachunku.
    result += ` ${playFor(perf).name}: ${format(amountFor(perf)/100)} (liczba miejsc:
    ↪${perf.audience})\n`;
    totalAmount += amountFor(perf);
  }
  result += `Należność: ${format(totalAmount/100)}\n`;
  result += `Punkty promocyjne: ${volumeCredits}\n`;
  return result;
```

Wymiana zmiennej funkcyjnej na zadeklarowaną funkcję jest refaktoryzacją, jednak nie nadałem jej osobnej nazwy i nie umieściłem w katalogu. Istnieje wiele refaktoryzacji, które moim zdaniem nie są tak ważne, aby je w ten sposób wyróżniać. Powyższą operację wykonuje się łatwo i rzadko, więc nie uważam, że jest warta zachodu.

Nie podoba mi się nazwa `format`, ponieważ nie zawiera informacji, co właściwie funkcja robi. Z kolei nazwa `formatAsPLN` jest nieco za długa, tym bardziej że funkcja jest stosowana w szablonie ciągu znaków wewnątrz niewielkiego zakresu. Uważam jednak, że formatowanie walutowe jest tu warte podkreślenia, dlatego wybrałem odpowiednią nazwę i zastosowałem refaktoryzację Zmiana Deklaracji Funkcji (s. x).

Najwyższy poziom…

```
function statement(invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Rachunek dla ${invoice.customer}\n`;
  for (let perf of invoice.performances) {
    volumeCredits += volumeCreditsFor(perf);

    // Utworzenie wiersza rachunku.
```

```
    result += ` ${playFor(perf).name}: ${pln(amountFor(perf))} (liczba miejsc:
    ↪${perf.audience})\n`;
    totalAmount += amountFor(perf);
  }
  result += `Należność: ${pln(totalAmount)}\n`;
  result += `Punkty promocyjne: ${volumeCredits}\n`;
  return result;
```

Funkcja `statement`...

```
function pln(aNumber) {
  return new Intl.NumberFormat("pl-PL", {
    style: "currency",
    currency: "PLN",
    minimumFractionDigits: 2
  }).format(aNumber/100);
}
```

Dobieranie nazw jest obowiązkową, ale niełatwą operacją. Dzielenie dużej funkcji na mniejsze ma sens tylko wtedy, gdy stosuje się odpowiednie nazwy. Jeżeli są one dobre, nie trzeba czytać kodów funkcji, aby się dowiedzieć, co robią. Trudno jest jednak dobrać od razu odpowiednią nazwę, dlatego wybrałem najlepszą, która mi przyszła w tym momencie do głowy. Jeżeli później trzeba będzie ją zmienić, zrobię to na pewno. Często zdarza mi się, że najlepszą nazwę mogę dobrać dopiero po ponownym przejrzeniu całego kodu.

Przy okazji zmiany nazwy przeniosłem do funkcji powtarzające się działanie dzielenia przez 100. Zapisywanie wartości kwot w postaci całkowitych liczb groszy jest często stosowaną praktyką. Nie trzeba wtedy używać wartości ułamkowych, a jednocześnie można wykonywać operacje arytmetyczne. Zawsze gdy trzeba wyświetlić kwotę z dokładnością do groszy, trzeba stosować typ zmiennoprzecinkowy. Troszczy się o to funkcja formatująca.

Usunięcie zmiennej sumującej punkty promocyjne

Kolejną zmienną, którą się zajmę, jest `volumeCredits`. Ten przypadek jest nieco trudniejszy, ponieważ wartość zmiennej jest modyfikowana w kolejnych iteracjach pętli. Dlatego moim pierwszym ruchem będzie zastosowanie refaktoryzacji Podział Pętli (s. x), aby oddzielić operację sumowania od zmiennej `volumeCredits`.

Najwyższy poziom…

```
function statement(invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Rachunek dla ${invoice.customer}\n`;

  for (let perf of invoice.performances) {

    // Utworzenie wiersza rachunku.
    result += ` ${playFor(perf).name}: ${pln(amountFor(perf))} (liczba miejsc:
    ↪${perf.audience})\n`;
    totalAmount += amountFor(perf);
  }
```

```
  for (let perf of invoice.performances) {
    volumeCredits += volumeCreditsFor(perf);
  }

  result += `Należność: ${pln(totalAmount)}\n`;
  result += `Punkty promocyjne: ${volumeCredits}\n`;
  return result;
```

Teraz mogę zastosować refaktoryzację Przesunięcie Instrukcji (s. x) i przenieść deklarację zmiennej bliżej pętli.

Najwyższy poziom…

```
function statement(invoice, plays) {
  let totalAmount = 0;
  let result = `Rachunek dla ${invoice.customer}\n`;
  for (let perf of invoice.performances) {

    // Utworzenie wiersza rachunku.
    result += ` ${playFor(perf).name}: ${pln(amountFor(perf))} (liczba miejsc:
    ↪${perf.audience})\n`;
    totalAmount += amountFor(perf);
  }
  let volumeCredits = 0;
  for (let perf of invoice.performances) {
    volumeCredits += volumeCreditsFor(perf);
  }
  result += `Należność: ${pln(totalAmount)}\n`;
  result += `Punkty promocyjne: ${volumeCredits}\n`;
  return result;
```

Dzięki zebraniu razem instrukcji modyfikujących zmienną `volumeCredits` łatwiej jest wykonać refaktoryzację Zastąpienie Zmiennej Tymczasowej Zapytaniem (s. x). Jak poprzednio, pierwszym krokiem jest zastosowanie przekształcenia Ekstrakcja Funkcji (s. x) dla całego fragmentu kodu wyliczającego wartość zmiennej.

Funkcja `statement`…

```
function totalVolumeCredits() {
  let volumeCredits = 0;
  for (let perf of invoice.performances) {
    volumeCredits += volumeCreditsFor(perf);
  }
  return volumeCredits;
}
```

Najwyższy poziom…

```
function statement(invoice, plays) {
  let totalAmount = 0;
  let result = `Rachunek dla ${invoice.customer}\n`;
  for (let perf of invoice.performances) {
```

```
    // Utworzenie wiersza rachunku.
    result += ` ${playFor(perf).name}: ${pln(amountFor(perf))} (liczba miejsc:
    ↪${perf.audience})\n`;
    totalAmount += amountFor(perf);
  }
  let volumeCredits = totalVolumeCredits();
  result += `Należność: ${pln(totalAmount)}\n`;
  result += `Punkty promocyjne: ${volumeCredits}\n`;
  return result;
```

Po wyodrębnieniu funkcji mogę zastosować refaktoryzację Wchłonięcie Zmiennej (s. x):

Najwyższy poziom…

```
function statement(invoice, plays) {
  let totalAmount = 0;
  let result = `Rachunek dla ${invoice.customer}\n`;
  for (let perf of invoice.performances) {

    // Utworzenie wiersza rachunku.
    result += ` ${playFor(perf).name}: ${pln(amountFor(perf))} (liczba miejsc:
    ↪${perf.audience})\n`;
    totalAmount += amountFor(perf);
  }
  result += `Należność: ${pln(totalAmount)}\n`;
  result += `Punkty promocyjne: ${totalVolumeCredits()}\n`;
  return result;
```

Chciałbym się na chwilę zatrzymać i opisać, co tu zrobiłem. Przede wszystkim zdaję sobie sprawę, że czytelnicy znów mogą być zaniepokojeni wydajnością uzyskanego kodu. Wielu programistów nieufnie podchodzi do wielokrotnego stosowania pętli. Zazwyczaj jednak uruchamianie pętli takich jak powyższa ma znikomy wpływ na wydajność kodu. Gdyby zmierzyć czas jego działania przed refaktoryzacją i po niej, prawdopodobnie nie dałoby się stwierdzić istotniej różnicy. I tak jest w większości przypadków. Wielu programistów, nawet doświadczonych, nie potrafi właściwie oceniać faktycznej wydajności kodu. Nasze intuicyjne szacunki są nietrafione, ponieważ używamy inteligentnych kompilatorów, zaawansowanych technik buforowania danych i innych mechanizmów. Wydajność całego oprogramowania zazwyczaj jest zdeterminowana sprawnością kilku fragmentów kodu i zmiany wprowadzane w innych miejscach nie robią istotnej różnicy.

Jednak „zazwyczaj” to nie to samo co „zawsze”. Czasami refaktoryzacja ma znaczący wpływ na wydajność. Ale nawet wtedy jej nie przerywam, ponieważ dobrze zrefaktoryzowany kod łatwiej jest dostrajać. Jeżeli w wyniku refaktoryzacji wydajność wyraźnie się pogorszy, będę musiał poświęcić później nieco czasu na jej poprawę. Czasami może to oznaczać konieczność wycofania się z wykonanych operacji, ale w większości przypadków dzięki refaktoryzacji daje się znaleźć jeszcze skuteczniejszy sposób poprawy wydajności. W efekcie uzyskuje się kod, który jest zarówno czytelniejszy, jak i szybszy.

Moja generalna opinia o związku refaktoryzacji i wydajności brzmi: w większości przypadków nie trzeba się tym przejmować. Jeżeli w wyniku refaktoryzacji wydajność kodu spadnie, najpierw należy dokończyć przekształcenia, a dopiero potem dostrajać sprawność programu.

Drugą kwestią, na którą chciałbym zwrócić uwagę, są niewielkie kroki, jakie wykonywałem, usuwając zmienną `volumeCredits`. Były to cztery wymienione niżej operacje. Po każdej z nich kompilowałem, testowałem i zatwierdzałem kod w lokalnym repozytorium.

- Podział Pętli (s. x) — oddzielająca instrukcje sumujące wartości.
- Przesunięcie Instrukcji (s. x) — przenosząca instrukcję inicjującą zmienną bliżej kodu sumującego.
- Ekstrakcja Funkcji (s. x) — tworząca funkcję wyliczającą sumaryczną wartość.
- Wchłonięcie Zmiennej (s. x) — całkowicie usuwająca zmienną.

Przyznam się, że nie zawsze wykonuję takie małe kroki jak powyższe. Ale zawsze gdy napotykam trudniejszy problem, w pierwszym odruchu skracam kroki. Robię tak przede wszystkim dlatego, że jeżeli test po refaktoryzacji nie powiedzie się, a ja nie będę w stanie szybko znaleźć i usunąć przyczyny, będę mógł natychmiast wrócić do ostatnio zatwierdzonego, poprawnego kodu i powtórzyć wszystko w jeszcze mniejszych krokach. Jest to skuteczny sposób, ponieważ często zatwierdzam kod, a małe kroki zdecydowanie przyspieszają pracę, szczególnie gdy kod jest trudny.

Opisaną wyżej sekwencję wykonałem jeszcze raz w celu usunięcia zmiennej `totalAmount`. Zacząłem od podzielenia pętli (potem był cykl kompilacja – test – zatwierdzenie), następnie przesunąłem instrukcję inicjującą zmienną (znowu kompilacja – test – zatwierdzenie) i na koniec wyekstrahowałem funkcję. Tu pojawiła się mała niedoskonałość: najlepszą nazwą dla funkcji byłaby `totalAmount`, ale została już zajęta przez zmienną i w tym miejscu nie można jej było użyć. Dlatego podczas ekstrakcji nadałem funkcji pierwszą nazwę, która mi przyszła do głowy („kompot jabłkowy") i wykonałem cykl kompilacja – test – zatwierdzenie).

Funkcja `statement`...

```
function appleSauce() {
  let totalAmount = 0;
  for (let perf of invoice.performances) {
    totalAmount += amountFor(perf);
  }
  return totalAmount;
}
```

Najwyższy poziom...

```
function statement(invoice, plays) {
  let result = `Rachunek dla ${invoice.customer}\n`;
  for (let perf of invoice.performances) {
    result += ` ${playFor(perf).name}: ${pln(amountFor(perf))} (liczba miejsc:
    ↪${perf.audience})\n`;
  }
  let totalAmount = appleSauce();

  result += `Należność: ${pln(totalAmount)}\n`;
  result += `Punkty promocyjne: ${totalVolumeCredits()}\n`;
  return result;
```

Teraz wchłaniam zmienną (kompilacja – test – zatwierdzenie) i nadaję funkcji bardziej sensowną nazwę (kompilacja – test – zatwierdzenie).

Najwyższy poziom...

```
function statement(invoice, plays) {
  let result = `Rachunek dla ${invoice.customer}\n`;
  for (let perf of invoice.performances) {
```

```
    result += ` ${playFor(perf).name}: ${pln(amountFor(perf))} (liczba miejsc:
    ↪${perf.audience})\n`;
  }
  result += `Należność: ${pln(totalAmount())}\n`;
  result += `Punkty promocyjne: ${totalVolumeCredits()}\n`;
  return result;
```

Funkcja `statement`...

```
function totalAmount() {
  let totalAmount = 0;
  for (let perf of invoice.performances) {
    totalAmount += amountFor(perf);
  }
  return totalAmount;
}
```

Na koniec zgodnie z przyjętą konwencją modyfikuję nazwy zmiennych wewnątrz wyodrębnionych funkcji:

Funkcja `statement`...

```
function totalAmount() {
  let result = 0;
  for (let perf of invoice.performances) {
    result += amountFor(perf);
  }
  return result;
}
function totalVolumeCredits() {
  let result = 0;
  for (let perf of invoice.performances) {
    result += volumeCreditsFor(perf);
  }
  return result;
}
```

Aktualny stan: mnóstwo zagnieżdżonych funkcji

Teraz jest dobry moment, aby zrobić przerwę i sprawdzić, w jakim stanie jest kod:

```
function statement(invoice, plays) {
  let result = `Rachunek dla ${invoice.customer}\n`;
  for (let perf of invoice.performances) {
    result += ` ${playFor(perf).name}: ${pln(amountFor(perf))} (liczba miejsc:
    ↪${perf.audience})\n`;
  }
  result += `Należność: ${pln(totalAmount())}\n`;
  result += `Punkty promocyjne: ${totalVolumeCredits()}\n`;
  return result;
```

```
  function totalAmount() {
    let result = 0;
    for (let perf of invoice.performances) {
      result += amountFor(perf);
    }
    return result;
  }

  function totalVolumeCredits() {
    let result = 0;
    for (let perf of invoice.performances) {
      result += volumeCreditsFor(perf);
    }
    return result;
  }

  function pln(aNumber) {
    return new Intl.NumberFormat("pl-PL", {
      style: "currency",
      currency: "PLN",
      minimumFractionDigits: 2
    }).format(aNumber/100);
  }

  function volumeCreditsFor(aPerformance) {
    let result = 0;
    result += Math.max(aPerformance.audience - 30, 0);
    if ("komedia" === playFor(aPerformance).type)
      result +=   Math.floor(aPerformance.audience / 5);
    return result;
  }

  function playFor(aPerformance) {
    return plays[aPerformance.playID];
  }

  function amountFor(aPerformance) {
    let result = 0;
    switch (playFor(aPerformance).type) {
      case "tragedia":
        result = 40000;
        if (aPerformance.audience > 30) {
          result += 1000 * (aPerformance.audience - 30);
        }
        break;
      case "komedia":
        result = 30000;
        if (aPerformance.audience > 20) {
          result += 10000 + 500 * (aPerformance.audience - 20);
        }
        result += 300 * aPerformance.audience;
        break;
      default:
        throw new Error(`Nieznany typ przedstawienia: ${playFor(aPerformance).type}`);
    }
    return result;
  }
}
```

Teraz struktura kodu jest znacznie lepsza. Główna funkcja `statement` składa się z zaledwie siedmiu wierszy, a jedyną wykonywaną przez nią operacją jest wyświetlanie rachunku. Wszystkie obliczenia zostały przeniesione do kilku funkcji dodatkowych. Dzięki temu łatwiej jest zrozumieć poszczególne obliczenia, jak również ogólny proces tworzenia rachunku.

Rozdzielenie faz obliczeń i formatowania

Do tej pory refaktoryzacja dotyczyła modyfikowania struktury funkcji, aby łatwiej było ją zrozumieć i wyodrębnić z niej logiczne części. Tak się często robi na początku refaktoryzacji. Dzielenie skomplikowanego kodu na mniejsze funkcje, podobnie jak nadawanie im właściwych nazw, jest ważną operacją. Teraz mogę się skupić na wprowadzeniu poważniejszych zmian w funkcjonalności kodu, a konkretnie na utworzeniu odmiany funkcji `statement` zwracającej wynik w formacie HTML. Obecnie jest to znacznie prostsze zadanie. Kiedy cały kod wykonujący obliczenia jest wydzielony, wystarczy, że napiszę wersję HTML siedmiu wierszy głównego kodu. Problem polega jednak na tym, że wydzielone funkcje są zagnieżdżone w funkcji `statement` i chociaż są należycie uporządkowane, nie chcę ich kopiować i wklejać do nowej funkcji. Chciałbym, aby były wykorzystywane przez obie wersje funkcji `statement`, zarówno tekstową, jak i HTML-ową.

Można to osiągnąć na kilka sposobów. Moim ulubionym jest refaktoryzacja Podział Faz (s. x). Celem jest podzielenie algorytmu na dwie części: jedną wykonującą obliczenia wymagane przez funkcję `statement`, i drugą zamieniającą tekst na kod HTML. W pierwszej części będą tworzone pośrednie dane, przekazywane następnie drugiej części.

Refaktoryzację Podział Faz (s. x) zacznę od Ekstrakcji Funkcji (s. x), aby wyodrębnić kod dla drugiej fazy. W tym przypadku są to instrukcje wyświetlające informacje, czyli w zasadzie cały kod funkcji `statement`, który wraz z zagnieżdżonymi funkcjami umieszczam w osobnej funkcji najwyższego poziomu o nazwie `renderPlainText` („przygotuj zwykły tekst”).

```
function statement (invoice, plays) {
  return renderPlainText(invoice, plays);
}

function renderPlainText(invoice, plays) {
  let result = `Rachunek dla ${invoice.customer}\n`;
  for (let perf of invoice.performances) {
    result += ` ${playFor(perf).name}: ${pln(amountFor(perf))} (liczba miejsc:
    ↪${perf.audience})\n`;
  }
  result += `Należność: ${pln(totalAmount())}\n`;
  result += `Punkty promocyjne: ${totalVolumeCredits()}\n`;
  return result;

  function totalAmount() {...}
  function totalVolumeCredits() {...}
  function usd(aNumber) {...}
  function volumeCreditsFor(aPerformance) {...}
  function playFor(aPerformance) {...}
  function amountFor(aPerformance) {...}
```

Jak zwykle wykonuję cykl kompilacja – test – zatwierdzenie, a następnie tworzę obiekt, który będzie odgrywał rolę pośredniej struktury danych pomiędzy dwiema fazami. Obiekt ten umieszczam w parametrze funkcji `renderPlainText` (i znów cykl kompilacja – test – zatwierdzenie).

```
function statement (invoice, plays) {
  const statementData = {};
  return renderPlainText(statementData, invoice, plays);
}

function renderPlainText(data, invoice, plays) {
  let result = `Rachunek dla ${invoice.customer}\n`;
  for (let perf of invoice.performances) {
    result += ` ${playFor(perf).name}: ${pln(amountFor(perf))} (liczba miejsc:
    ↪${perf.audience})\n`;
  }
  result += `Należność: ${pln(totalAmount())}\n`;
  result += `Punkty promocyjne: ${totalVolumeCredits()}\n`;
  return result;

  function totalAmount() {...}
  function totalVolumeCredits() {...}
  function usd(aNumber) {...}
  function volumeCreditsFor(aPerformance) {...}
  function playFor(aPerformance) {...}
  function amountFor(aPerformance) {...}
```

Teraz sprawdzam pozostałe parametry funkcji `renderPlainText`. Zamierzam przenieść zawarte w nich dane do pośredniej struktury, abym potem mógł cały kod wykonujący obliczenia przesunąć do funkcji `statement`, a funkcja `renderPlainText` mogła przetwarzać wyłącznie dane przekazane jej za pomocą parametru `data`.

Moim pierwszym ruchem jest przeniesienie danych klienta do pośredniego obiektu (i kompilacja – test – zatwierdzenie).

```
function statement (invoice, plays) {
  const statementData = {};
  statementData.customer = invoice.customer;
  return renderPlainText(statementData, invoice, plays);
}

function renderPlainText(data, invoice, plays) {
  let result = `Rachunek dla ${data.customer}\n`;
  for (let perf of invoice.performances) {
    result += ` ${playFor(perf).name}: ${pln(amountFor(perf))} (liczba miejsc:
    ↪${perf.audience})\n`;
  }
  result += `Należność: ${pln(totalAmount())}\n`;
  result += `Punkty promocyjne: ${totalVolumeCredits()}\n`;
  return result;
```

Podobnie przenoszę informacje o przedstawieniach, dzięki czemu mogę usunąć parametr `invoice` funkcji `renderPlainText` (kompilacja – test – zatwierdzenie).

Najwyższy poziom…

```
function statement (invoice, plays) {
  const statementData = {};
  statementData.customer = invoice.customer;
  statementData.performances = invoice.performances;
  return renderPlainText(statementData, ~~invoice,~~ plays);
}

function renderPlainText(data, plays) {
  let result = `Rachunek dla ${data.customer}\n`;
  for (let perf of data.performances) {
    result += ` ${playFor(perf).name}: ${pln(amountFor(perf))} (liczba miejsc:
    ↪${perf.audience})\n`;
  }
  result += `Należność: ${pln(totalAmount())}\n`;
  result += `Punkty promocyjne: ${totalVolumeCredits()}\n`;
  return result;
```

Funkcja `renderPlainText`*…*

```
function totalAmount() {
  let result = 0;
  for (let perf of data.performances) {
    result += amountFor(perf);
  }
  return result;
}

function totalVolumeCredits() {
  let result = 0;
  for (let perf of data.performances) {
    result += volumeCreditsFor(perf);
  }
  return result;
}
```

Chciałbym, aby tytuł przedstawienia również znalazł się w pośrednich danych. W tym celu muszę wzbogacić rekord opisujący przedstawienie, dodając odpowiednie informacje (kompilacja – test – zatwierdzenie).

```
function statement (invoice, plays) {
  const statementData = {};
  statementData.customer = invoice.customer;
  statementData.performances =
    invoice.performances.map(enrichPerformance);
  return renderPlainText(statementData, plays);

  function enrichPerformance(aPerformance) {
    const result = Object.assign({}, aPerformance);
    return result;
  }
```

W tej chwili jedynie tworzę kopię obiektu z danymi o przedstawieniu, ale za chwilę dodam do niego nowe informacje. Kopiuję obiekt, ponieważ nie chcę modyfikować danych przekazywanych w parametrze funkcji. Wolę, aby parametr był w miarę możliwości niezmienny (niemutowalny). Mutowalność powoduje, że kod szybko zaczyna się psuć.

Zapis `result = Object.assign({}, aPerformance)` na pewno wygląda dziwnie dla kogoś mniej obeznanego z JavaScriptem. Instrukcja ta wykonuje tzw. płaską kopię obiektu. Wolałbym, aby ta czynność była zawarta w osobnej funkcji. Ale jest to jedna z operacji, która została „zaszyta" w języku JavaScript, i tworzenie dla niej osobnej funkcji wyglądałoby dziwnie dla programistów używających JavaScriptu.

Teraz, gdy mam informacje o przedstawieniu, muszę je dodać do obiektu. W tym celu stosuję refaktoryzację Przeniesienie Funkcji (s. x) w funkcjach `playFor` i `statement` (kompilacja – test – zatwierdzenie).

Funkcja `statement`...

```
function enrichPerformance(aPerformance) {
  const result = Object.assign({}, aPerformance);
  result.play = playFor(result);
  return result;
}

function playFor(aPerformance) {
  return plays[aPerformance.playID];
}
```

Następnie w funkcji `renderPlainText` zastępuję obiektem wszystkie wywołania funkcji `playFor` (kompilacja – test – zatwierdzenie).

Funkcja `renderPlainText`...

```
  let result = `Rachunek dla ${data.customer}\n`;
  for (let perf of data.performances) {
    result += ` ${perf.play.name}: ${pln(amountFor(perf))} (liczba miejsc:
    ↪${perf.audience})\n`;
  }
  result += `Należność: ${pln(totalAmount())}\n`;
  result += `Punkty promocyjne: ${totalVolumeCredits()}\n`;
  return result;

function volumeCreditsFor(aPerformance) {
  let result = 0;
  result += Math.max(aPerformance.audience - 30, 0);
  if ("komedia" === aPerformance.play.type)
    result += Math.floor(aPerformance.audience / 5);
  return result;
}

function amountFor(aPerformance) {
  let result = 0;
  switch (aPerformance.play.type) {
  case "tragedia":
    result = 40000;
    if (aPerformance.audience > 30) {
      result += 1000 * (aPerformance.audience - 30);
```

```
    }
    break;
  case "komedia":
    result = 30000;
    if (aPerformance.audience > 20) {
      result += 10000 + 500 * (aPerformance.audience - 20);
    }
    result += 300 * aPerformance.audience;
    break;
  default:
    throw new
      Error(`Nieznany typ przedstawienia: ${aPerformance.play.type}`);
  }
  return result;
}
```

W podobny sposób przenoszę funkcję `amountFor` (kompilacja – test – zatwierdzenie).

Funkcja `statement`...

```
function enrichPerformance(aPerformance) {
  const result = Object.assign({}, aPerformance);
  result.play = playFor(result);
  result.amount = amountFor(result);
  return result;
}

function amountFor(aPerformance) {...}
```

Funkcja `renderPlainText`...

```
  let result = `Rachunek dla ${data.customer}\n`;
  for (let perf of data.performances) {
    result += ` ${perf.play.name}: ${pln(perf.amount)} (liczba miejsc: ${perf.audience})\n`;
  }
  result += `Należność: ${pln(totalAmount())}\n`;
  result += `Punkty promocyjne: ${totalVolumeCredits()}\n`;
  return result;

function totalAmount() {
  let result = 0;
  for (let perf of data.performances) {
    result += perf.amount;
  }
  return result;
}
```

Następnie przenoszę kod wyliczający punkty promocyjne (kompilacja – test – zatwierdzenie).

Funkcja `statement`...

```
function enrichPerformance(aPerformance) {
  const result = Object.assign({}, aPerformance);
  result.play = playFor(result);
```

```
  result.amount = amountFor(result);
  result.volumeCredits = volumeCreditsFor(result);
  return result;
}

function volumeCreditsFor(aPerformance) {...}
```

Funkcja `renderPlainText...`

```
function totalVolumeCredits() {
  let result = 0;
  for (let perf of data.performances) {
    result += perf.volumeCredits;
  }
  return result;
}
```

Na koniec przenoszę dwie instrukcje wyliczające sumy.

Funkcja `statement...`

```
const statementData = {};
statementData.customer = invoice.customer;
statementData.performances = invoice.performances.map(enrichPerformance);
statementData.totalAmount = totalAmount(statementData);
statementData.totalVolumeCredits = totalVolumeCredits(statementData);
return renderPlainText(statementData, plays);

function totalAmount(data) {...}
function totalVolumeCredits(data) {...}
```

Funkcja `renderPlainText...`

```
let result = `Rachunek dla ${data.customer}\n`;
for (let perf of data.performances) {
  result += ` ${perf.play.name}: ${pln(perf.amount)} (liczba miejsc: ${perf.audience})\n`;
}
result += `Należność: ${pln(data.totalAmount)}\n`;
result += `Punkty promocyjne: ${data.totalVolumeCredits}\n`;
return result;
```

Choć zmienną `statementData` mogłem wykorzystać bezpośrednio w powyższych funkcjach wyliczających sumy, wolałem przekazać ją jawnie za pomocą parametrów.

Po przeniesieniu funkcji i wykonaniu cyklu kompilacja – test – zatwierdzenie nie mogłem sobie odmówić kilku szybkich refaktoryzacji Zastąpienie Pętli Funkcją Tablicową (s. x).

Funkcja `renderPlainText...`

```
function totalAmount(data) {
  return data.performances
    .reduce((total, p) => total + p.amount, 0);
}
```

```
function totalVolumeCredits(data) {
  return data.performances
    .reduce((total, p) => total + p.volumeCredits, 0);
}
```

Teraz wyodrębniam cały kod pierwszej fazy do osobnej funkcji (kompilacja – test – zatwierdzenie).

Najwyższy poziom…

```
function statement (invoice, plays) {
  return renderPlainText(createStatementData(invoice, plays));
}

function createStatementData(invoice, plays) {
  const statementData = {};
  statementData.customer = invoice.customer;
  statementData.performances =
    invoice.performances.map(enrichPerformance);
  statementData.totalAmount = totalAmount(statementData);
  statementData.totalVolumeCredits = totalVolumeCredits(statementData);
  return statementData;
```

Ponieważ kod tej fazy jest teraz wyraźnie oddzielony od reszty, przenoszę go do osobnego pliku (zmieniając jednocześnie nazwę zwracanego wyniku zgodnie z przyjętą konwencją).

Plik statement.js…

```
import createStatementData from './createStatementData.js';
```

Plik createStatementData.js…

```
export default function createStatementData(invoice, plays) {
  const result = {};
  result.customer = invoice.customer;
  result.performances = invoice.performances.map(enrichPerformance);
  result.totalAmount = totalAmount(result);
  result.totalVolumeCredits = totalVolumeCredits(result);
  return result;

  function enrichPerformance(aPerformance) {...}
  function playFor(aPerformance) {...}
  function amountFor(aPerformance) {...}
  function volumeCreditsFor(aPerformance) {...}
  function totalAmount(data) {...}
  function totalVolumeCredits(data) {...}
```

Jeszcze jeden, ostatni cykl kompilacja – test – zatwierdzenie i można łatwo napisać wersję funkcji zwracającej wynik w formacie HTML.

Plik statement.js…

```
function htmlStatement (invoice, plays) {
  return renderHtml(createStatementData(invoice, plays));
}
```

```
function renderHtml(data) {
  let result = `<h1>Rachunek dla ${data.customer}</h1>\n`;
  result += "<table>\n";
  result += "<tr><th>Przedstawienie</th><th>Liczba miejsc </th><th>Cena</th></tr>";
  for (let perf of data.performances) {
    result += `<tr><td>${perf.play.name}</td><td>${perf.audience}</td>`;
    result += `<td>${pln(perf.amount)}</td></tr>\n`;
  }
  result += "</table>\n";
  result += `<p>Należność: <em>${usd(data.totalAmount)}</em></p>\n`;
  result += `<p>Punkty promocyjne: <em>${data.totalVolumeCredits}</em> </p>\n`;
  return result;
}

function usd(aNumber) {...}
```

(Funkcję `pln` przeniosłem na najwyższy poziom, aby można jej było użyć w funkcji `renderHtml`).

Aktualny stan: podział na dwa pliki (i fazy)

Teraz jest dobry moment na podsumowanie dotychczasowych działań i sprawdzenie, w jakim stanie jest kod. Są dwa pliki.

Plik statement.js

```
import createStatementData from './createStatementData.js';
function statement(invoice, plays) {
  return renderPlainText(createStatementData(invoice, plays));
}

function renderPlainText(data, plays) {
  let result = `Rachunek dla ${data.customer}\n`;
  for (let perf of data.performances) {
    result += ` ${perf.play.name}: ${pln(perf.amount)} (liczba miejsc: ${perf.audience})\n`;
  }
  result += `Należność: ${pln(data.totalAmount)}\n`;
  result += `Punkty promocyjne: ${data.totalVolumeCredits}\n`;
  return result;
}

function htmlStatement(invoice, plays) {
  return renderHtml(createStatementData(invoice, plays));
}

function renderHtml(data) {
  let result = `<h1>Rachunek dla ${data.customer}</h1>\n`;
  result += "<table>\n";
  result += "<tr><th>Przedstawienie</th><th>Liczba miejsc </th><th>Cena</th></tr>";
  for (let perf of data.performances) {
    result += `<tr><td>${perf.play.name}</td><td>${perf.audience}</td>`;
    result += `<td>${pln(perf.amount)}</td></tr>\n`;
  }
  result += "</table>\n";
```

```
  result += `<p>Należność: <em>${pln(data.totalAmount)}</em></p>\n`;
  result += `<p>Punkty promocyjne: <em>${data.totalVolumeCredits}</em> </p>\n`;
  return result;
}

function pln(aNumber) {
    return new Intl.NumberFormat("pl-PL", {
      style: "currency",
      currency: "PLN",
      minimumFractionDigits: 2 }).format(aNumber/100);
}
```

Plik createStatementData.js

```
export default function createStatementData(invoice, plays) {
  const result = {};
  result.customer = invoice.customer;
  result.performances = invoice.performances.map(enrichPerformance);
  result.totalAmount = totalAmount(result);
  result.totalVolumeCredits = totalVolumeCredits(result);
  return result;

  function enrichPerformance(aPerformance) {
    const result = Object.assign({}, aPerformance);
    result.play = playFor(result);
    result.amount = amountFor(result);
    result.volumeCredits = volumeCreditsFor(result);
    return result;
  }

  function playFor(aPerformance) {
    return plays[aPerformance.playID]
  }

  function amountFor(aPerformance) {
    let result = 0;
    switch (aPerformance.play.type) {
      case "tragedia":
        result = 40000;
        if (aPerformance.audience > 30) {
          result += 1000 * (aPerformance.audience - 30);
        }
        break;
      case "komedia":
        result = 30000;
        if (aPerformance.audience > 20) {
          result += 10000 + 500 * (aPerformance.audience - 20);
        }
        result += 300 * aPerformance.audience;
        break;
      default:
      throw new
        Error(`Nieznany typ przedstawienia: ${aPerformance.play.type}`);
    }
    return result;
  }
```

```
  function volumeCreditsFor(aPerformance) {
    let result = 0;
    result += Math.max(aPerformance.audience - 30, 0);
    if ("komedia" === aPerformance.play.type)
      result += Math.floor(aPerformance.audience / 5);
    return result;
  }

  function totalAmount(data) {
    return data.performances
      .reduce((total, p) => total + p.amount, 0);
  }

  function totalVolumeCredits(data) {
    return data.performances
      .reduce((total, p) => total + p.volumeCredits, 0);
  }
}
```

Kod jest teraz dłuższy, ponieważ składa się z 70 wierszy (nie licząc funkcji `htmlStatement`), a na początku było ich 44. Zmiana wynika głównie z przeniesienia operacji do osobnych funkcji. Jeżeli poza liczbą wierszy kod niczym się nie różni od pierwotnej wersji, wtedy jego wydłużenie jest złym efektem. Jednak rzadko kiedy nie ma różnic. Dodatkowe wiersze dzielą algorytm programu na osobne części: wykonującą obliczenia i przygotowującą rachunek. Dzięki tej modularności łatwiej jest zrozumieć oddzielne części kodu i zależności między nimi. Zwięzłość jest istotą dowcipu, ale przejrzystość jest istotą rozwojowego programowania. Modularność pozwoliła mi dodać funkcję zwracającą wynik HTML bez powielania obliczeń.

Uwaga Podczas programowania kieruj się biwakową zasadą: pozostawiaj miejsce po sobie w lepszym stanie, niż zastałeś.

Mogłem wprowadzić więcej zmian upraszczających kod przygotowujący rachunek, ale zajmę się nimi później. Zawsze staram się znajdować kompromis między refaktoryzowaniem a wprowadzaniem nowych funkcjonalności. Większość programistów nie przywiązuje dużej wagi do refaktoryzacji i to też jest swego rodzaju kompromis. Poza tym przestrzegam biwakowej zasady: zawsze pozostawiaj po sobie miejsce w lepszym stanie, niż zastałeś. Nigdy nie będzie idealnie, ale na pewno będzie lepiej.

Uporządkowanie obliczeń według typów przedstawień

Teraz skupię się na kolejnej zmianie funkcjonalności: obsłudze nowych rodzajów przedstawień. Dla każdego z nich cena i punkty promocyjne będą wyliczane inaczej. W tej chwili zmiany mogę wprowadzać, modyfikując instrukcje warunkowe w funkcji wykonującej obliczenia. W funkcji `amountFor` główną rolę przy wyborze obliczeń odgrywa typ przedstawienia. Jednak kod oparty na instrukcjach warunkowych, taki ja wyżej opisany, będzie się pogarszał w miarę wprowadzania nowych modyfikacji, chyba że wykorzysta się elementy strukturalne oferowane przez język programowania.

Jest wiele sposobów nadawania struktury kodowi, ale w tym przypadku naturalnym wyborem będzie polimorfizm typu. Jest to podstawa programowania obiektowego. Klasyczne funkcjonalności

programowania obiektowego w języku JavaScript były przez długi czas kwestionowane, aż do chwili pojawienia się standardu ECMAScript 2015 cechującego się spójną składnią i strukturą. W przypadkach takich jak nasz warto stosować ten język.

Mój plan jest taki, aby ustanowić hierarchiczną strukturę, w której osobne podklasy będą wykonywały obliczenia właściwe dla komedii i tragedii. W głównym kodzie będzie wywoływana polimorficzna funkcja wykonująca różne obliczenia ceny w zależności od typu przedstawienia. W podobny sposób będą wyliczane punkty promocyjne. Aby to osiągnąć, zastosuję kilka refaktoryzacji. Będzie to przede wszystkim Zastąpienie Instrukcji Warunkowej Polimorfizmem (s. x). Dzięki niej fragment kodu warunkowego zamienię na kod polimorficzny. Zanim jednak zastosuję tę refaktoryzację, potrzebna mi będzie hierarchiczna struktura klas. Muszę utworzyć klasę zawierającą funkcje wyliczające cenę i punkty.

Zacznę od przejrzenia kodu wykonującego obliczenia. (Jedną z miłych konsekwencji przeprowadzonych wcześniej refaktoryzacji jest możliwość pominięcia kodu formatującego wynik, ponieważ struktura danych wyjściowych nie zostanie zmieniona. Mogę się dodatkowo zabezpieczyć, kodując testy sprawdzające pośrednią strukturę danych).

Plik createStatementData.js…

```
export default function createStatementData(invoice, plays) {
  const result = {};
  result.customer = invoice.customer;
  result.performances = invoice.performances.map(enrichPerformance);
  result.totalAmount = totalAmount(result);
  result.totalVolumeCredits = totalVolumeCredits(result);
  return result;

  function enrichPerformance(aPerformance) {
    const result = Object.assign({}, aPerformance);
    result.play = playFor(result);
    result.amount = amountFor(result);
    result.volumeCredits = volumeCreditsFor(result);
    return result;
  }

  function playFor(aPerformance) {
    return plays[aPerformance.playID]
  }

  function amountFor(aPerformance) {
    let result = 0;
    switch (aPerformance.play.type) {
      case "tragedia":
        result = 40000;
        if (aPerformance.audience > 30) {
          result += 1000 * (aPerformance.audience - 30);
        }
        break;
      case "komedia":
        result = 30000;
        if (aPerformance.audience > 20) {
          result += 10000 + 500 * (aPerformance.audience - 20);
        }
        result += 300 * aPerformance.audience;
        break;
      default:
```

```
      throw new
        Error(`Nieznany typ przedstawienia: ${aPerformance.play.type}`);
    }
    return result;
  }

  function volumeCreditsFor(aPerformance) {
    let result = 0;
    result += Math.max(aPerformance.audience - 30, 0);
    if ("komedia" === aPerformance.play.type)
      result += Math.floor(aPerformance.audience / 5);
    return result;
  }

  function totalAmount(data) {
    return data.performances
      .reduce((total, p) => total + p.amount, 0);
  }

  function totalVolumeCredits(data) {
    return data.performances
      .reduce((total, p) => total + p.volumeCredits, 0);
  }
}
```

Utworzenie kalkulatora przedstawień

Kluczowa jest funkcja `enrichPerformance`, ponieważ zapisuje ona dane każdego przedstawienia w pośredniej strukturze. Obecnie `enrichPerformance` wywołuje funkcje, które w obliczeniach ceny i punktów wykorzystują instrukcje warunkowe. Moim celem jest wywoływanie tych funkcji za pomocą głównej klasy. Ponieważ ta klasa będzie zawierała funkcje wyliczające dane przedstawień, nazwę ją `PerformanceCalculator` („kalkulator przedstawień”).

Funkcja `createStatementData`...

```
function enrichPerformance(aPerformance) {
  const calculator = new PerformanceCalculator(aPerformance);
  const result = Object.assign({}, aPerformance);
  result.play = playFor(result);
  result.amount = amountFor(result);
  result.volumeCredits = volumeCreditsFor(result);
  return result;
}
```

Najwyższy poziom…

```
class PerformanceCalculator {
  constructor(aPerformance) {
    this.performance = aPerformance;
  }
}
```

Na razie nowy obiekt nie robi nic. Zamierzam przenieść do niego kod. Zacznę od najprostszej rzeczy, jaką jest przeniesienie rekordu danych o przedstawieniu. Tak naprawdę nie muszę tego robić, ponieważ kod rekordu nie będzie podlegał polimorfizmowi. Jednak dzięki temu przetwarzanie danych będzie się odbywało w jednym miejscu, a kod będzie bardziej spójny i czytelny.

Aby osiągnąć zamierzony cel, zastosuję refaktoryzację Zmiana Deklaracji Funkcji (s. x), co pozwoli mi przekazywać dane o przedstawieniu do kalkulatora.

Funkcja `createStatementData`...

```
function enrichPerformance(aPerformance) {
  const calculator =
    new PerformanceCalculator(aPerformance, playFor(aPerformance));
  const result = Object.assign({}, aPerformance);
  result.play = calculator.play;
  result.amount = amountFor(result);
  result.volumeCredits = volumeCreditsFor(result);
  return result;
}
```

Klasa `PerformanceCalculator`...

```
class PerformanceCalculator {
  constructor(aPerformance, aPlay) {
    this.performance = aPerformance;
    this.play = aPlay;
  }
}
```

(Nie będę już więcej pisał *kompilacja – test – zatwierdzenie*, ponieważ przypuszczam, że jesteś tym znużony. Jednak ten cykl będę wykonywał przy każdej okazji. Czasami sam jestem tym zmęczony, przez co do kodu zakradają się błędy. Dostaję wtedy nauczkę i wracam na utarty szlak).

Przeniesienie funkcji do kalkulatora

Kolejny fragment kodu, który przeniosę, dotyczy wyliczania ceny przedstawienia. Poprzestawiałem już trochę zagnieżdżone funkcje, ale tym razem zmiana będzie głębsza, dlatego zastosuję refaktoryzację Przeniesienie Funkcji (s. x). Pierwszą czynnością będzie skopiowanie kodu do nowego kontekstu, tj. klasy kalkulatora. Następnie dopasuję odpowiednio ten kod, zmienię parametr `aPerformance` na właściwość `this.performance`, a funkcję `playFor(aPerformance)` na `this.play`.

Klasa `PerformanceCalculator`...

```
get amount() {
  let result = 0;
  switch (this.play.type) {
    case "tragedia":
      result = 40000;
      if (this.performance.audience > 30) {
        result += 1000 * (this.performance.audience - 30);
      }
      break;
    case "komedia":
```

```
      result = 30000;
      if (this.performance.audience > 20) {
        result += 10000 + 500 * (this.performance.audience - 20);
      }
      result += 300 * this.performance.audience;
      break;
    default:
      throw new Error(`Nieznany typ przedstawienia: ${this.play.type}`);
  }
  return result;
}
```

Mogę teraz skompilować kod, aby wykryć błędy składniowe. Kompilacja w moim środowisku zachodzi w chwili uruchomienia kodu, ponieważ korzystam z kompilatora Babel. Pozwala to sprawdzić, czy nowy kod nie zawiera błędów składniowych, ale nic poza tym. Jednak mimo to jest to przydatna operacja.

Teraz oryginalną funkcję zmieniam tak, aby wywoływała nową funkcję.

Funkcja `createStatementData`...

```
function amountFor(aPerformance) {
  return new PerformanceCalculator(aPerformance,
    playFor(aPerformance)).amount;
}
```

Następnie stosuję refaktoryzację Wchłonięcie Funkcji (s. x), aby nowa funkcja była wywoływana bezpośrednio.

Funkcja createStatementData...

```
function enrichPerformance(aPerformance) {
  const calculator = new PerformanceCalculator(aPerformance,
    playFor(aPerformance));
  const result = Object.assign({}, aPerformance);
  result.play = calculator.play;
  result.amount = calculator.amount;
  result.volumeCredits = volumeCreditsFor(result);
  return result;
}
```

Powyższy proces powtarzam, aby przenieść kod wyliczający punkty promocyjne.

Funkcja `createStatementData`...

```
function enrichPerformance(aPerformance) {
  const calculator = new PerformanceCalculator(aPerformance,
    playFor(aPerformance));
  const result = Object.assign({}, aPerformance);
  result.play = calculator.play;
  result.amount = calculator.amount;
  result.volumeCredits = calculator.volumeCredits;
  return result;
}
```

Klasa `PerformanceCalculator`...

```
get volumeCredits() {
  let result = 0;
  result += Math.max(this.performance.audience - 30, 0);
  if ("komedia" === this.play.type)
    result += Math.floor(this.performance.audience / 5);
  return result;
}
```

Utworzenie kalkulatora polimorficznego

Teraz gdy kod klasy jest gotowy, czas zaimplementować polimorfizm. Pierwszym krokiem będzie refaktoryzacja Zastąpienie Kodu Typu Podklasami (s. x), aby zamiast kodu typu pojawiły się podklasy. W tym celu utworzę podklasy kalkulatora i wykorzystam odpowiednią podklasę w funkcji `createPerformanceData`. Aby zastosować właściwą podklasę, muszę zastąpić wywołanie konstruktora funkcją, ponieważ w JavaScripcie konstruktor nie może zwracać podklasy. Dlatego zastosuję refaktoryzację Zastąpienie Konstruktora Funkcją Wytwórczą (s. x).

Funkcja `createStatementData`...

```
function enrichPerformance(aPerformance) {
  const calculator = createPerformanceCalculator(aPerformance,
    playFor(aPerformance));
  const result = Object.assign({}, aPerformance);
  result.play = calculator.play;
  result.amount = calculator.amount;
  result.volumeCredits = calculator.volumeCredits;
  return result;
}
```

Najwyższy poziom...

```
function createPerformanceCalculator(aPerformance, aPlay) {
  return new PerformanceCalculator(aPerformance, aPlay);
}
```

Teraz mogę utworzyć podklasy, które będzie zwracać nowa funkcja.

Najwyższy poziom...

```
function createPerformanceCalculator(aPerformance, aPlay) {
  switch(aPlay.type) {
    case "tragedia": return new TragedyCalculator(aPerformance, aPlay);
    case "komedia" : return new ComedyCalculator(aPerformance, aPlay);
  default:
    throw new Error(`Nieznany typ przedstawienia: ${aPlay.type}`);
  }
}
class TragedyCalculator extends PerformanceCalculator {
}
class ComedyCalculator extends PerformanceCalculator {
}
```

W ten sposób utworzyłem polimorficzną strukturę i mogę zastosować refaktoryzację Zastąpienie Instrukcji Warunkowej Polimorfizmem (s. x). Zacznę od wyliczenia ceny za tragedię.

Klasa `TragedyCalculator`...

```
get amount() {
  let result = 40000;
  if (this.performance.audience > 30) {
    result += 1000 * (this.performance.audience - 30);
  }
  return result;
}
```

Powyższa metoda w podklasie wystarczy, aby pozbyć się instrukcji warunkowej w nadklasie. Jeżeli jesteś tak obsesyjnie pedantyczny jak ja, możesz zrobić coś takiego:

Klasa `PerformanceCalculator`...

```
get amount() {
  let result = 0;
  switch (this.play.type) {
    case "tragedia":
      throw 'zła rzecz';
    case "komedia":
      result = 30000;
      if (this.performance.audience > 20) {
        result += 10000 + 500 * (this.performance.audience - 20);
      }
      result += 300 * this.performance.audience;
      break;
    default:
      throw new Error(`Nieznany typ przedstawienia: ${this.play.type}`);
  }
  return result;
}
```

Mogłem usunąć sekcję `case "tragedia"`, aby błąd był zgłaszany w sekcji `default`. Wolę jednak, aby błędy były zgłaszane jawnie. Poza tym kod w takiej postaci będzie istniał tylko przez chwilę (dlatego użyłem ciągu znaków, a nie obiektu błędu).

Teraz mogę się zająć przypadkiem komedii.

Klasa `ComedyCalculator`...

```
get amount() {
  let result = 30000;
  if (this.performance.audience > 20) {
    result += 10000 + 500 * (this.performance.audience - 20);
  }
  result += 300 * this.performance.audience;
  return result;
}
```

Mogę już usunąć metodę `amount` z nadklasy, ponieważ nigdzie nie jest wywoływana. Lepiej jednak będzie zostawić po niej jakiś ślad.

Klasa `PerformanceCalculator`...

```
get amount() {
  throw new Error('To robi podklasa.');
}
```

Kolejną instrukcją warunkową przeznaczoną do wymiany jest kod wyliczający punkty promocyjne. Z dyskusji o przyszłych przedstawieniach wiem, że w większości przypadków spodziewana jest publiczność większa niż 30 osób. Wyjątki będą dotyczyć tylko niektórych kategorii przedstawień. Dlatego uzasadnione jest pozostawienie domyślnego, najczęstszego przypadku w nadklasie, który w razie potrzeby będzie nadpisywany za pomocą podklas. Dlatego przypadek komedii umieszczam w podklasie:

Klasa `PerformanceCalculator`...

```
get volumeCredits() {
  return Math.max(this.performance.audience - 30, 0);
}
```

Klasa `ComedyCalculator`...

```
get volumeCredits() {
  return super.volumeCredits + Math.floor(this.performance.audience / 5);
}
```

Aktualny stan: tworzenie danych za pomocą polimorficznego kalkulatora

Pora podsumować, co polimorficzny kalkulator wniósł do kodu.

Plik createStatementData.js

```
export default function createStatementData(invoice, plays) {
  const result = {};
  result.customer = invoice.customer;
  result.performances = invoice.performances.map(enrichPerformance);
  result.totalAmount = totalAmount(result);
  result.totalVolumeCredits = totalVolumeCredits(result);
  return result;

  function enrichPerformance(aPerformance) {
    const calculator = createPerformanceCalculator(aPerformance,
      playFor(aPerformance));
    const result = Object.assign({}, aPerformance);
    result.play = calculator.play;
    result.amount = calculator.amount;
```

```
    result.volumeCredits = calculator.volumeCredits;
    return result;
  }

  function playFor(aPerformance) {
    return plays[aPerformance.playID]
  }

  function totalAmount(data) {
    return data.performances
      .reduce((total, p) => total + p.amount, 0);
  }
  function totalVolumeCredits(data) {
    return data.performances
      .reduce((total, p) => total + p.volumeCredits, 0);
  }
}

function createPerformanceCalculator(aPerformance, aPlay) {
  switch(aPlay.type) {
    case "tragedia": return new TragedyCalculator(aPerformance, aPlay);
    case "komedia" : return new ComedyCalculator(aPerformance, aPlay);
    default:
      throw new Error(`Nieznany typ przedstawienia: ${aPlay.type}`);
  }
}

class PerformanceCalculator {
  constructor(aPerformance, aPlay) {
    this.performance = aPerformance;
    this.play = aPlay;
  }
  get amount() {
    throw new Error('To robi podklasa.');
  }
  get volumeCredits() {
    return Math.max(this.performance.audience - 30, 0);
  }
}

class TragedyCalculator extends PerformanceCalculator {
  get amount() {
    let result = 40000;
    if (this.performance.audience > 30) {
      result += 1000 * (this.performance.audience - 30);
    }
    return result;
  }
}

class ComedyCalculator extends PerformanceCalculator {
  get amount() {
    let result = 30000;
    if (this.performance.audience > 20) {
      result += 10000 + 500 * (this.performance.audience - 20);
    }
```

```
    result += 300 * this.performance.audience;
    return result;
  }
  get volumeCredits() {
    return super.volumeCredits +
      Math.floor(this.performance.audience / 5);
  }
}
```

Jak poprzednio, po wprowadzeniu nowej struktury kod się wydłużył. Korzyść jest jednak taka, że wyliczenia dotyczące różnych rodzajów przedstawień zostały zgrupowane razem. Dzięki temu na pewno łatwiej będzie wprowadzać większość zmian w tym kodzie. Dodanie obsługi nowego przedstawienia będzie wymagało napisania nowej podklasy i zastosowania jej w funkcji `create ↪PerformanceCalculator`.

Opisany przykład daje pewne wyobrażenie o tym, kiedy przydają się podklasy takie jak powyższe. W tym przypadku instrukcje warunkowe użyte w dwóch funkcjach (`amountFor` i `volumeCreditsFor`) zastąpiłem jednym konstruktorem `createPerformanceCalculator`. Im więcej będzie funkcji uzależnionych od tego samego rodzaju polimorfizmu, tym bardziej korzystne okaże się opisane podejście.

Rozwiązaniem alternatywnym do przedstawionego jest utworzenie funkcji `createPerformance ↪Data` zwracającej kalkulator. Dzięki temu nie trzeba byłoby zapisywać danych w pośredniej strukturze. Jedną z ciekawych cech systemu klas w JavaScripcie jest możliwość używania getterów do zwykłego odwoływania się do danych. Decyzja, czy ma być zwracana instancja klasy, czy wyliczane samodzielne dane wyjściowe, zależy od tego, jak struktura danych będzie wykorzystywana w nadrzędnym kodzie. W tym przypadku wolałem pokazać, jak użyć pośredniej struktury danych do ukrycia decyzji o zastosowaniu kalkulatora polimorficznego.

Podsumowanie

Przedstawiony przykład jest bardzo prosty. Mam nadzieję, że jest także wystarczająco obrazowy, by dać pojęcie na temat istoty refaktoryzacji. Zastosowałem kilka przekształceń: Ekstrakcję Funkcji (s. x), Wchłonięcie Zmiennej (s. x), Przeniesienie Funkcji (s. x) oraz Zastąpienie Instrukcji Warunkowej Polimorfizmem (s. x).

Cały epizod refaktoryzacyjny składał się z trzech etapów: rozłożenia oryginalnej funkcji na zbiór zagnieżdżonych funkcji, zastosowanie refaktoryzacji Podział Fazy (s. x) w celu rozdzielenia kodów wykonującego obliczenia i wyświetlającego rachunek oraz zaimplementowania polimorficznego kalkulatora wykonującego obliczenia. Każdy etap wzmacniał strukturę kodu, dzięki której wykonywane przez niego operacje stawały się coraz bardziej zrozumiałe.

Jak to się często zdarza podczas refaktoryzacji, pierwsze przekształcenia wykonywane są po to, aby się dowiedzieć, o co chodzi w kodzie. Typowy cykl jest następujący: przeczytanie kodu, wyobrażenie sobie jego struktury, zastosowanie refaktoryzacji do przeniesienia struktury z głowy do kodu. Powstaje wtedy dodatnie sprzężenie zwrotne: im bardziej czytelny jest kod, tym łatwiej go zrozumieć i nadać mu lepszą strukturę. Jest jeszcze kilka ulepszeń, które można wprowadzić, ale wydaje mi się, że pomyślnie przeszedłem próbę uczynienia kodu znacznie lepszym, niż gdy go zastałem.

Uwaga Naprawdę dobry kod to taki, który można łatwo zmieniać.

Piszę tutaj o dobrym kodzie. Programiści uwielbiają dyskutować, co to znaczy dobry kod. Wiem, że niektórzy mogą mieć zastrzeżenia do mojej skłonności tworzenia małych, odpowiednio nazywanych funkcji. Jeżeli uznamy, że jest to kwestia estetyki, gdzie nic nie jest dobre ani złe, to znaczy, że nie mamy żadnego pomysłu i kierujemy się wyłącznie własnym gustem. Uważam jednak, że można wznieść się ponad gust i uznać, że prawdziwym wyróżnikiem dobrego kodu jest możliwość łatwego wprowadzania w nim zmian. Gdy ktoś będzie chciał coś zmienić w kodzie, powinien być w stanie znaleźć odpowiednie miejsce i łatwo dokonać modyfikacji bez popełniania błędów. Zdrowy kod maksymalizuje naszą wydajność, pozwala szybciej i taniej implementować nowe funkcjonalności oczekiwane przez użytkowników. Aby kod był zdrowy, zwracaj baczną uwagę na to, co się dzieje wewnątrz zespołu programistów, a następnie refaktoryzuj kod, by przybliżyć go do ideału.

Jednak najważniejszą rzeczą, którą powinieneś zapamiętać z tego rozdziału, jest rytm refaktoryzacji. Zawsze gdy pokazuję programistom, jak refaktoryzuję kod, są zaskoczeni, że wykonuję tak małe kroki i że kod zawsze kompiluje się poprawnie, przechodzi pomyślnie testy i działa prawidłowo. Tak samo ja byłem zaskoczony, gdy Kent Beck pokazał mi dwie dekady temu w hotelu w Detroit, jak się robi refaktoryzację. Kluczową kwestią jest nabranie przekonania, że cel można osiągnąć szybciej, gdy posuwa się małymi krokami, które się kumulują w istotne zmiany, przy czym kod cały czas pozostaje sprawny. Pamiętaj o tym. Całą reszta jest milczeniem.

Skorowidz

A

algorytm, 195
API, 299
architektura, 73
asercja, 296

B

bazy danych, 72
błędy, 61

C

czytelność kodu, 61

D

dane
    globalne, 84
    mutowalne, 84
dekompozycja funkcji statement, 21
delegat, 189, 372, 390
delegowanie, 380
dziedziczenie, 357

E

ekstrakcja
    funkcji, 110
    klasy, 182
    punktów promocyjnych, 29
    zmiennej, 122
enkapsulacja, 161
    kolekcji, 170
    rekordu, 162
    zmiennej, 134

F

fala uderzeniowa, 85
fazy, 46, 156
formatowanie, 39
funkcja statement, 21
funkcje
    wytwórcze, 327
    zagnieżdżone, 37
funkcjonalności, 64, 86, 197

H

hierarchia klas, 341

I

instrukcja, 212, 221
    migracyjna, 129
instrukcje warunkowe, 87, 223, 256, 259
interfejs, 90

K

kalkulator, 51
    polimorficzny, 53, 55
    przedstawień, 50

klasa, 90, 146, 182, 186
  danych, 91
kod zduplikowany, 82
konstruktor, 327

L

leniwa klasa, 88
literał obiektowy, 289
lista parametrów, 83

Ł

łańcuchy komunikatów, 89

M

Martwy Kod, 234
modyfikacja danych początkowych, 102

N

nadklasy, 366, 390
nazwa
  funkcji, 129
  pola, 240
  zmiennej, 139

O

obiekt, 174, 273
  parametryczny, 142
odgałęzienia kodu, 70

P

pakiet przekształceń, 109
parametr, 130, 320
parametryzacja funkcji, 303
pętla, 88, 225
podklasa, 353, 360, 372
podział na pliki, 46
pole tymczasowe, 89
polecenie, 330, 336
polimorfizm, 267, 273
pośrednik, 89, 192
potok, 229
problemy z refaktoryzacją, 68
programowanie, 62
przekazywanie obiektu, 312
przekształcenia refaktoryzacyjne, 107
przeniesienie
  funkcji, 198, 203
  funkcjonalności, 197
  instrukcji, 211, 215
  pola, 206, 209
przesunięcie
  ciała konstruktora, 347
  instrukcji, 221
  metody, 342, 351
  pola, 345, 352
punkty promocyjne, 29, 33

R

refaktoryzacja, 12, 17, 59, 61, 73, 77
  automatyczna, 78
  długoterminowa, 66
  interfejsu API, 299
  odśmiecająca, 65
  okazjonalna, 65
  planowana, 65
  podczas inspekcji kodu, 66
  poznawcza, 64
referencja, 248, 252

T

testy, 71, 93, 97
transformata, 151, 292
tworzenie oprogramowania, 74
typy proste, 87

U

usunięcie
  Parametru-Flagi, 307
  podklasy, 360
  zmiennej, 33
    format, 31
    play, 25

W

warunki graniczne, 102
wchłonięcie
  funkcji, 118
  klasy, 186
  zmiennej, 126

własność kodu, 69
wybór przekształceń, 108
wydajność, 75
wyrażenia warunkowe, 255
wywołanie funkcji, 220

Y

yagni, 73

Z

zapytania, 178, 244, 317, 320
zasada trzech, 63
zasady refaktoryzacji, 59
zmiana
  deklaracji funkcji, 127
  nazwy
    funkcji, 130
    pola, 240
    zmiennej, 139
zmienna
  format, 31
  play, 25
zmienne
  lokalne, 112, 114, 115
  tymczasowe, 178

REFAKTORYZACJA: CZYTELNY KOD, WYDAJNE DZIAŁANIE I BRAK BŁĘDÓW!

Refaktoryzacja ma na celu ulepszanie istniejącego kodu. Poprawia jego strukturę, czytelność i wydajność. Prowadzona poprawnie, cechuje się zdyscyplinowaniem metody, systematycznością i konsekwencją w działaniu, a także ciągłym minimalizowaniem ryzyka wprowadzenia błędów. Niemniej refaktoryzacja jest ryzykowna. Polega na wprowadzaniu zmian do działającego kodu, co może doprowadzić do powstania trudno wykrywalnych błędów. Ryzyko wzrasta, jeśli refaktoryzacja jest przeprowadzana w sposób nieprzemyślany. Okazuje się, że transformacja kodu, która ma doprowadzić do jego ulepszenia, to spore wyzwanie.

Ta książka jest zaktualizowanym wydaniem praktycznego przewodnika po refaktoryzacji. Choć jest przeznaczona dla profesjonalnego programisty, znalazło się tu zrozumiałe wprowadzenie do tego zagadnienia z opisem celów, technik i możliwości refaktoryzacji. Wspomniano także o problemach związanych z refaktoryzacją. Natomiast zasadniczą część książki stanowi znakomicie uzupełniony i wzbogacony katalog przekształceń refaktoryzacyjnych. Do zilustrowania poszczególnych technik refaktoryzacji autorzy wybrali język JavaScript, jednak kod został przedstawiony w taki sposób, aby prezentowane koncepcje bez trudu zrozumiał każdy programista.

W tej książce między innymi:

- solidne wprowadzenie do refaktoryzacji
- przekształcenia refaktoryzacyjne: zasady, sposoby, testy
- enkapsulacja w refaktoryzacji
- upraszczanie wyrażeń oraz porządkowanie danych, zmiennych i pól
- refaktoryzacja klas i API

Martin Fowler — zaczął programować we wczesnych latach 80. zeszłego stulecia. Jest autorem kilku książek, prelegentem i konsultantem. W projektowaniu systemów oprogramowania — co jest jego wielką pasją — stara się łączyć największe zalety różnych koncepcji.

Kent Beck — jest wybitnym autorytetem w dziedzinie programowania, twórcą metodologii programowania ekstremalnego i jednym z sygnatariuszy *Manifestu agile*.

Pearson
Addison-Wesley

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

ISBN 978-83-283-5563-7

Cena: 79,00 zł