O'REILLY®

CSS

Refaktoryzacja kodu

Helion

Steve Lindstrom

Tytuł oryginału: CSS Refactoring: Tune Your Style Sheets for Performance

Tłumaczenie: Lech Lachowski

ISBN: 978-83-283-2098-7

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

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

Printed in Poland.

Spis treści

ROZDZIAŁ 1.

Refaktoryzacja a architektura

Oto punkt wyjścia dla naszej podróży po refaktoryzacji CSS. W tym rozdziale dowiemy się, czym jest refaktoryzacja i w jaki sposób odnosi się ona do architektury oprogramowania. Zastanowimy się także nad znaczeniem refaktoryzacji i omówimy niektóre z powodów, dla których kod może jej potrzebować. Przeanalizujemy również dwa przykłady refaktoryzacji, aby wyjaśnić te koncepcje.

Czym jest refaktoryzacja?

Refaktoryzacja to proces przepisywania kodu w celu uproszczenia go i zwiększenia możliwości jego ponownego wykorzystania bez zmiany jego zachowania. Jest to niezbędna umiejętność podczas pisania kodu, ponieważ w pewnym momencie tak czy inaczej trzeba będzie to zrobić. Być może nawet przeprowadzałeś już kiedyś refaktoryzację kodu, nie zdając sobie z tego sprawy! Ponieważ refaktoryzacja nie zmienia zachowania kodu, można się zastanawiać, dlaczego w ogóle warto ją robić. Zanim będzie można odpowiedzieć na to pytanie, ważne jest jednak zrozumienie, czym jest architektura oprogramowania.

Co to jest architektura oprogramowania?

Tak jak żywa istota, system oprogramowania zazwyczaj składa się z wielu części, które specjalizują się w robieniu jednej konkretnej rzeczy. Po połączeniu te fragmenty współpracują ze sobą, aby stworzyć system oprogramowania. Termin **architektura oprogramowania** jest używany do opisania, w jaki sposób pasują do siebie te wszystkie fragmenty projektu oprogramowania.

Każdy fragment oprogramowania, od prostej strony po system kontroli statku kosmicznego, ma jakąś architekturę, bez względu na to, czy jest to zamierzone, czy nie. Jednak najlepsze architektury są zazwyczaj zaplanowane na długo przed rozpoczęciem procesu kodowania. W kolejnych punktach opisane zostały niektóre z najważniejszych cech dobrej architektury.

Dobre architektury są przewidywalne

Przewidywalność oznacza, że można przyjąć dokładne założenia dotyczące sposobu działania oprogramowania i jego struktury. Ta cecha jest wskaźnikiem właściwego planowania i pomaga zaoszczędzić czas poświęcony na rozwój oprogramowania, ponieważ nie będzie żadnych wątpliwości w następujących kwestiach:

- jakie są odpowiedzialności komponentów;
- gdzie można znaleźć konkretny fragment kodu;
- gdzie należy umieścić nowy fragment kodu.

Ponieważ w przewidywalnej architekturze można przyjmować dokładne założenia, programiści, którzy nie są zaznajomieni z kodem, powinni być w stanie szybciej go zrozumieć.

Dobre architektury promują ponowne wykorzystanie kodu

Możliwość ponownego wykorzystania kodu to cecha, która pozwala użyć tego samego kodu w różnych miejscach bez jego duplikowania. Ponowne wykorzystywanie kodu jest pożądane, ponieważ skraca czas rozwoju oprogramowania, gdyż nie musimy przepisywać już istniejących fragmentów kodu. Ponadto im mniejsza liczba fragmentów kodu służących do rozwiązania konkretnego problemu, tym mniej czasu trzeba będzie poświęcić na utrzymywanie wszystkich tych implementacji. Jeśli na przykład odkryjemy błąd we fragmencie kodu, który jest wielokrotnie wykorzystywany w całym projekcie, będziemy wiedzieć, że ten błąd będzie obecny wszędzie tam, gdzie stosowany jest ten kod. Jednak poprawiając go w jednym miejscu, poprawimy go automatycznie we wszystkich miejscach, w których ten fragment kodu jest używany.

Dobre architektury są rozszerzalne

Rozszerzalność jest zasadą dobrej architektury, ponieważ pozwala z łatwością dodawać do systemu nowe funkcjonalności. Większość oprogramowania nie jest budowana od początku do końca w ciągu jednego dnia, więc bardzo ważne jest, aby można było budować system stopniowo bez konieczności wprowadzania istotnych zmian strukturalnych. Jeśli projekt często wymaga wprowadzania znaczących zmian w architekturze, staje się o wiele trudniejszy do opracowania.

Dobre architektury są łatwe w utrzymywaniu

Podobnie jak rozszerzalność, również **utrzymywalność** jest bardzo ważną cechą architektury, ponieważ pozwala łatwo modyfikować istniejące funkcjonalności. Z biegiem czasu wymagania mogą się zmieniać i będziemy zmuszeni zmodyfikować kod. Utrzymywalność oprogramowania oznacza, że będzie można zmodyfikować jeden fragment kodu bez konieczności wprowadzania drastycznych zmian w wielu innych miejscach kodu.

Architektura oprogramowania a refaktoryzacja

Mówiąc w skrócie, refaktoryzacja ma pomóc w utrzymywaniu i promowaniu dobrej architektury oprogramowania. Nie jest ona niczym więcej niż zestawem technik, które można wykorzystać do reorganizacji kodu w bardziej znaczącą strukturę z zamiarem uczynienia go bardziej przewidywalnym, rozszerzalnym, utrzymywalnym i zapewniającym wyższy stopień ponownego użycia. Gdy architektura oprogramowania będzie mieć wymienione cechy, będzie znacznie bardziej niezawodna dla docelowych użytkowników i o wiele przyjemniej będzie się nad nią pracować programistom.

Braki, które prowadzą do refaktoryzacji

Dlaczego kod nie jest pisany prawidłowo od razu, aby później nie trzeba było go refaktoryzować? Pomimo dołożenia najlepszych starań, aby zaprojektować i napisać możliwie najwyższej jakości kod, z biegiem czasu zawsze zmieni się *coś*, co będzie wymagać refaktoryzacji. Rzućmy okiem na kilka przyczyn.

Zmieniające się wymagania

Systemy oprogramowania ewoluują w wyniku zmieniających się wymagań. Gdy oprogramowanie zostało napisane w celu zaspokojenia jednego zestawu wymagań, zwykle nie uwzględniono możliwości zaspokojenia innego zestawu wymagań, który nie został jeszcze napisany (i w ogóle nie był planowany). Dlatego gdy zmieniają się wymagania, musi również zmieniać się kod, a jeśli istnieją ograniczenia czasowe, to w wyniku chodzenia na skróty może obniżyć się jakość kodu.

Źle zaprojektowana architektura

Nawet jeśli jesteś świadomy cech dobrej architektury, to nie zawsze możliwe jest poświęcenie dużej ilości czasu na zaplanowanie wszystkiego. A jeśli od samego początku nie masz jasnego wyobrażenia o tym, jak wszystko powinno ze sobą współdziałać, być może po drodze trzeba będzie zrobić jakąś refaktoryzację. Powszechne jest również dość szybkie budowanie nowych funkcji (co może doprowadzić do chodzenia na skróty), aby sprawdzić, czy zyskają one uznanie użytkowników, a gdy tak się stanie, to późniejsze czyszczenie kodu (lub w przeciwnym razie jego usuwanie).

Niedoszacowanie poziomu trudności

Oszacowanie, jak długo może potrwać proces rozwoju oprogramowania, jest trudne, a niestety te szacunki są często wykorzystywane do tworzenia harmonogramów. Gdy harmonogram danego projektu jest niedoszacowany, wywołuje presję na programistów, aby „po prostu to zrobić", co prowadzi do szybkiego pisania kodu bez zbytniego zastanowienia. Jeśli dzieje się to dość często, nawet najlepszy kod może zamienić się w wielki talerz „kodu spaghetti", który jest trudny do zrozumienia i niesforny w utrzymywaniu.

Ignorowanie najlepszych praktyk

Bycie na bieżąco ze wszystkimi najlepszymi praktykami może być trudne, zwłaszcza jeśli Twoja praca obejmuje wiele technologii i (lub) zarządzanie ludźmi. Jeśli pracujesz w zespole i przeoczysz jakąś dobrą praktykę, jest nadzieja, że uświadomi Ci to któryś z kolegów. Jeżeli nie wykorzysta się okazji do zastosowania jakiejś najlepszej praktyki, to w pewnym momencie w przyszłości konieczne może być zrewidowanie kodu i przeprowadzenie pewnej refaktoryzacji.

Trudności z nadążaniem za najlepszymi praktykami

Technologia zmienia się bardzo szybko, a w rezultacie technika, która była kiedyś uważana za najlepszą praktykę może okazać się nieaktualna. Jeśli na przykład przed rokiem 2011 chcieliśmy na stronie internetowej wyświetlić kontener, który ma zaokrąglone rogi, potrzebowaliśmy obrazu każdego rogu, musieliśmy osadzić te obrazy w kodzie HTML, a następnie wypozycjonować je za pomocą CSS, aby upewnić się, że wszystko jest prawidłowo wyrównane. Dziś ta technika jest przestarzała, ponieważ nowoczesne przeglądarki mogą wyświetlać zaokrąglone rogi za pomocą właściwości `border-radius` CSS. Jeśli nie będziesz stale aktualizować kodu pod kątem nowych najlepszych praktyk, z biegiem czasu jego techniczne zapóźnienie będzie narastać i okaże się, że jest on w znacznie gorszym stanie, niż byłby w przeciwnym razie.

Kiedy należy refaktoryzować kod?

Refaktoryzacja kodu jest znacznie łatwiejsza, gdy jest robiona w kontekście. Zwykle najlepiej jest przeprowadzać ją, kiedy usuwamy błąd lub budujemy nową funkcję, która wykorzystuje istniejący kod. Konsekwentna refaktoryzacja kodu podczas pracy nad mniejszymi zadaniami zmniejsza prawdopodobieństwo popsucia czegokolwiek, a ponadto skorzystają na tym również ci, którzy będą modyfikować ten sam kod po jego zrefaktoryzowaniu. Z biegiem czasu konsekwentna refaktoryzacja prowadzi do uzyskania lepszej jakości kodu, pod warunkiem że wprowadzane zmiany będą dostosowane do cech dobrej architektury.

Czasem jednak napotykamy fragment kodu, który ma wiele zależności, i możemy stanąć w obliczu decyzji, czy należy refaktoryzować ten kod, czy nie. Refaktoryzacja kodu, który ma wiele zależności, może być jak wyciąganie luźnej nitki z koszuli: im bardziej ją ciągniemy, tym bardziej się ona pruje. Analogicznie, im bardziej modyfikujemy fragment kodu posiadający wiele zależności, tym więcej zależności będziemy musieli w końcu zaktualizować. W takich sytuacjach, jeśli mamy napięty termin, korzystne może być najpierw skończenie pracy, a potem przeznaczenie trochę czasu na to, aby wrócić do pewnych kwestii i przeprowadzić refaktoryzację. Jeśli jednak po drodze okaże się, że są jakieś mniejsze rzeczy, które można zrefaktoryzować bez negatywnego wpływu na harmonogram, warto rozważyć zrefaktoryzowanie ich od razu.

Kiedy NIE należy refaktoryzować kodu?

Wiedza na temat tego, kiedy *nie* należy refaktoryzować kodu, jest chyba nawet ważniejsza od wiedzy na temat tego, kiedy refaktoryzować trzeba. Refaktoryzacja może mieć złą reputację, ponieważ programiści często przepisują kod tylko po to, żeby go przepisać. Może ktoś inny napisał kod, a osoba wykonująca niepotrzebną refaktoryzację cierpi na syndrom „to nie zostało napisane tutaj", mając poczucie, że kod jest gorszy, ponieważ sama go nie napisała. Albo być może kiedyś ktoś zdecyduje, że po prostu nie podoba mu się, w jaki sposób wcześniej napisał dany kod (może w nazwach klas użył podkreślników zamiast łączników i teraz chce zrobić odwrotnie), więc rozpętuje burzę wprowadzania zmian, aby zaspokoić swoje żądze. W wielu przypadkach można to uznać za pozorne działania, które sprawiają, że ludzie czują się produktywni, nawet gdy nie są. W rozdziale 5. poprzez sporządzenie zestawu standardów kodowania omówimy, jak zaplanować to, w jaki sposób powinien być napisany kod. W tym momencie najbardziej klarowna będzie wskazówka, że refaktoryzacja jest zalecana tylko wtedy, gdy poprawia architekturę lub powoduje dostosowanie się do standardów kodowania.

Czy mogę refaktoryzować swój kod?

Jeśli pracujesz nad własnym projektem, to odpowiedź donośnie brzmi „tak!". Ale jeśli pracujesz w organizacji, w której niekoniecznie jesteś osobą decyzyjną, odpowiedź może nie być tak oczywista. W idealnym świecie każda organizacja rozumiałaby znaczenie refaktoryzacji, ale często rzeczywistość jest inna. Jeśli Twoim kolegom w pracy brakuje wiedzy technicznej na temat refaktoryzacji, można spróbować ich edukować. Słyszałem, że książka *CSS. Refaktoryzacja kodu* jest miłym prezentem!

Rozsądne osoby, które są odpowiedzialne za dostarczanie oprogramowania z wysokiej jakości kodem, prawdopodobnie rozumieją tę kwestię, ale inni mogą twierdzić, że:

- poświęcanie czasu na przepisywanie kodu bez odnotowywania żadnych zmian jest stratą czasu i pieniędzy;
- jeśli coś nie jest popsute, nie trzeba tego naprawiać;
- trzeba było od razu napisać kod poprawnie.

Jeśli spotkasz się z którymkolwiek z tych argumentów, ale będziesz się czuć na tyle pewnie, aby przeprowadzić refaktoryzację, polecam, byś ją zrobił, pod warunkiem że zmieścisz się w harmonogramie i będziesz ostrożny, żeby niczego nie popsuć. Jeśli kiedykolwiek słyszałeś takie argumenty, mogę się założyć, że wypowiadająca je osoba nigdy nie uczestniczyła w przeglądaniu kodu, więc wprowadzone przez Ciebie zmiany prawdopodobnie i tak nie zostaną zauważone. Jeżeli jednak refaktoryzujesz kod tylko po to, żeby go zrefaktoryzować, być może powinieneś poczekać, aż stanie się bardziej oczywiste, że zmiany są konieczne. Przedwczesna optymalizacja często może być tak samo zła jak zapóźnienie techniczne.

Przykłady refaktoryzacji

Ponieważ masz już ogólne pojęcie o zaletach refaktoryzacji i wiesz, kiedy jej przeprowadzanie jest (a kiedy nie jest) dobrym pomysłem, możemy zacząć rozmawiać o tym, jak się do niej zabrać.

Chociaż ta książka poświęcona jest refaktoryzacji CSS, dużo łatwiej jest na początku przeanalizować tę koncepcję na podstawie kodu, który oblicza wartości dyskretne, zamiast skorzystać z kodu zmieniającego wygląd elementów HTML. Tak więc nasz pierwszy przykład przedstawia refaktoryzację podstawowego kodu JavaScript, który oblicza całkowitą wartość zamówienia e-commerce. W drugim przykładzie zrefaktoryzujemy fragment kodu CSS.

Przykłady kodów

Ponieważ trudno może być zrozumieć, co dzieje się w długich fragmentach kodu, które obejmują wiele stron i plików, w przykładach w tej książce wykorzystane zostały mniejsze fragmenty kodu. Cały kod JavaScript z naszego pierwszego przykładu można umieścić w pliku HTML, aby łatwiej było go uruchomić.

W przypadku bardziej skomplikowanych przykładów kod CSS, wykorzystywany do określenia ogólnego wyglądu i stylu elementów w przykładach, zostanie załączony w osobnym pliku CSS.

Używane w tej książce style, które są wplatane pomiędzy znacznikami `<style>` i `</style>`, będą bezpośrednio związane z omawianym przykładem i zostaną wykorzystane do zilustrowania koncepcji ziarnistości.

Wszystkie przykłady kodu są dostępne na serwerze wydawnictwa Helion, pod adresem *ftp://ftp.helion.pl/przyklady/cssref.zip*.

Refaktoryzacja. Przykład 1. Obliczanie całkowitej wartości zamówienia e-commerce

Listing 1.1 zawiera kod JavaScript, który oblicza całkowitą wartość zamówienia e-commerce, jeśli dostarczymy następujące dane:

- cenę każdej pozycji z zamówienia;
- liczbę sztuk każdej pozycji z zamówienia;
- koszt wysyłki każdej pozycji z zamówienia;
- informacje dotyczące wysyłki do danego klienta;
- opcjonalny kod rabatowy, który może obniżyć wartość zamówienia.

Listing 1.1. Obliczanie całkowitej wartości zamówienia e-commerce

```
/**
 * Oblicza całkowitą wartość zamówienia po uwzględnieniu kosztów dostawy, rabatów i podatków.
 *
 * @param {Object} customer - kolekcja informacji na temat osoby, która złożyła zamówienie.
 *
 * @param {Array.<Object>} lineItems - kolekcja zakupionych produktów
 * i ich liczebności oraz koszty wysyłki jednej pozycji.
 *
 * @param {string} discountCode - opcjonalny kod rabatu, który może
 * spowodować uwzględnienie rabatu przed dodaniem kosztów wysyłki i podatków.
 */
var getOrderTotal = function (customer, lineItems, discountCode) {
  var discountTotal = 0;
  var lineItemTotal = 0;
  var shippingTotal = 0;
  var taxTotal = 0;

  for (var i = 0; i < lineItems.length; i++) {
      var lineItem = lineItems[i];
      lineItemTotal += lineItem.price * lineItem.quantity;
      shippingTotal += lineItem.shippingPrice * lineItem.quantity;
  }

  if (discountCode === '20PERCENT') {
      discountTotal = lineItemTotal * 0.2;
  }

  if (customer.shiptoState === 'CA') {
      taxTotal = (lineItemTotal - discountTotal) * 0.08;
  }
```

```
  var total = (
      lineItemTotal -
      discountTotal +
      shippingTotal +
      taxTotal
  );
  return total;
};
```

Wywołanie funkcji `getOrderTotal` z wykorzystaniem danych z listingu 1.2 spowoduje wyświetlenie informacji *Całkowita wartość zamówienia: 266 zł*. Listing 1.3 wyjaśnia, dlaczego otrzymamy taki wynik.

Listing 1.2. Uruchamianie funkcji getOrderTotal z testowymi danymi wejściowymi

```
var lineItem1 = {
  price: 50,
  quantity: 1,
  shippingPrice: 10
};

var lineItem2 = {
  price: 100,
  quantity: 2,
  shippingPrice: 20
};
var lineItems = [lineItem1, lineItem2];

var customer = {
  shiptoState: 'CA'
};

var discountCode = '20PERCENT';

var total = getOrderTotal(customer, lineItems, discountCode);

document.writeln('Całkowita wartość zamówienia:' + total + 'zł');
```

Listing 1.3. Wyjaśnienie, dlaczego getOrderTotal spowoduje wyświetlenie wyniku Całkowita wartość zamówienia: 266 zł

```
discountTotal = 0
lineItemTotal = 0
shippingTotal = 0
taxTotal = 0

# Pierwsza iteracja PĘTLI FOR:
lineItemTotal = 0 + (50 * 1) = 50
shippingTotal = 0 + (10 * 1) = 10
```

```
# Druga iteracja PĘTLI FOR:
lineItemTotal = 50 + (100 * 2) = 250
shippingTotal = 10 + (20 * 2) = 50

# discountTotal jest obliczane, ponieważ discountCode równa się "20PERCENT":
discountTotal = 250 * 0.2 = 50

# taxTotal jest obliczane, ponieważ customer.shiptoState równa się "CA":
taxTotal = (250 - 50) * 0.08 = 16

total = 250 - 50 + 50 + 16 = 266
```

Testy jednostkowe

Po sprawdzeniu wszystkich obliczeń operacje matematyczne okazują się prawidłowe i wszystko wydaje się działać zgodnie z oczekiwaniami. Aby upewnić się, że wszystko będzie dalej funkcjonować prawidłowo, możemy teraz napisać test jednostkowy. Mówiąc wprost, **test jednostkowy** to fragment kodu, który wykonuje inny fragment kodu, żebyśmy mogli upewnić się, że wszystko działa zgodnie z oczekiwaniami. Testy jednostkowe powinny być pisane w celu przetestowania pojedynczych fragmentów funkcjonalności, aby zawęzić przyczynę wszelkich problemów, które mogą się pojawić. Ponadto zestaw testów jednostkowych napisanych dla całego projektu powinien zostać uruchomiony przed opublikowaniem nowego kodu, aby błędy, które zostały wprowadzone do systemu, mogły zostać wykryte i usunięte, zanim będzie za późno.

Dane wejściowe z listingu 1.2 mogą zostać wykorzystane do napisania testu jednostkowego, pokazanego w listingu 1.4, który to test zakłada, że funkcja zwraca oczekiwaną wartość (266). Po zakończeniu testu wyświetlona zostanie informacja o liczbie udanych i nieudanych prób oraz dodatkowo lista nieudanych prób.

Listing 1.4. Test jednostkowy dla funkcji getOrderTotal

```
var successfulTestCount = 0;
var unsuccessfulTestCount = 0;
var unsuccessfulTestSummaries = [];

/**
 * Przyjmuje asercję, że obliczenia w getOrderTotal() są prawidłowe.
 */
var testGetOrderTotal = function () {
```

```
    // ustawienie oczekiwań

    var expectedTotal = 266;

    // ustawienie danych testowych

    var lineItem1 = {
        price: 50,
        quantity: 1,
        shippingPrice: 10
    };

    var lineItem2 = {
        price: 100,
        quantity: 2,
        shippingPrice: 20
    };

    var lineItems = [lineItem1, lineItem2];

    var customer = {
        shiptoState: 'CA'
    };

    var discountCode = '20PERCENT';

    var total = getOrderTotal(customer, lineItems, discountCode);

    // sprawdzanie wyników pod kątem oczekiwań

    if (total === expectedTotal) {
      successfulTestCount++;
    } else {
      unsuccessfulTestCount++;
      unsuccessfulTestSummaries.push(
          'testGetOrderTotal: oczekiwane ' + expectedTotal + ';
          ↪otrzymane ' + total
      );
    }
};

// uruchomienie testów

testGetOrderTotal();
document.writeln('Liczba udanych prób: ' + successfulTestCount + '<br/>');
document.writeln('Liczba nieudanych prób: ' + unsuccessfulTestCount + '<br/>');

if (unsuccessfulTestCount) {
    document.writeln('<ul>');
```

```
        for(var i = 0; i < unsuccessfulTestSummaries.length; i++) {
            document.writeln('<li>' + unsuccessfulTestSummaries[i] + '</li>');
        }
        document.writeln('</ul>');
    }
```

W wyniku wykonania funkcji `testGetOrderTotal` test spełnił asercję, tak jak widać na rysunku 1.1.

Rysunek 1.1. Pomyślne wyniki testu jednostkowego

Jeśli jednak w przyszłości z jakiegoś powodu wprowadzony zostanie błąd i stosowany w obliczeniach mnożnik `discountTotal` zmieni się z 0.2 na –0.2, sytuacja będzie wyglądała inaczej i otrzymamy wynik przedstawiony na rysunku 1.2.

Testy jednostkowe są skutecznym sposobem, aby upewnić się, że z biegiem czasu system nadal będzie pracował zgodnie z oczekiwaniami. Testy te mogą być szczególnie przydatne podczas przepisywania kodu, ponieważ asercja będzie już udokumentowana i da większe poczucie pewności, że zachowanie kodu nie uległo zmianie.

Ponieważ rozumiemy już kod używany do obliczenia całkowitej wartości zamówienia e-commerce i mamy towarzyszący mu test jednostkowy, zobaczmy, w jaki sposób refaktoryzacja może coś poprawić.

Rysunek 1.2. Negatywne wyniki testu jednostkowego

Refaktoryzacja funkcji getOrderTotal

Przyglądając się z bliska `getOrderTotal`, zauważymy, że w tej jednej funkcji wykonywanych jest wiele obliczeń:

- całkowity rabat jest odejmowany od ostatecznej ceny;
- obliczana jest całkowita wartość wszystkich pozycji;
- obliczany jest całkowity koszt wysyłki;
- obliczany jest całkowity podatek;
- obliczana jest całkowita wartość zamówienia.

Jeśli do jednego z tych pięciu obliczeń przypadkowo wprowadzony zostanie błąd, test jednostkowy (`testGetOrderTotal`) będzie wskazywać, że coś poszło nie tak, ale nie będzie oczywiste, co *konkretnie* poszło źle. Jest to główny powód, dla którego testy jednostkowe powinny być pisane do testowania pojedynczych fragmentów funkcjonalności.

Aby zwiększyć ziarnistość kodu, każde z wyżej wymienionych obliczeń powinno zostać wyodrębnione do osobnej funkcji, której nazwa opisuje, co ona robi, tak jak w listingu 1.5.

Listing 1.5. Wyodrębnianie fragmentów kodu do nowych funkcji

```
/**
 * Oblicza całkowitą wartość wszystkich pozycji zamówienia.
 *
 * @param {Array.<Object>} lineItems - kolekcja zakupionych produktów
 * i ich liczebności oraz koszty wysyłki jednej pozycji.
 *
 * @returns {number} Całkowita wartość wszystkich zamówionych pozycji.
 */
var getLineItemTotal = function (lineItems) {
    var lineItemTotal = 0;

    for (var i = 0; i < lineItems.length; i++) {
        var lineItem = lineItems[i];
        lineItemTotal += lineItem.price * lineItem.quantity;
    }

    return lineItemTotal;
};

/**
 * Oblicza całkowite koszty wysyłki wszystkich zamówionych pozycji.
 *
 * @param {Array.<Object>} lineItems - kolekcja zakupionych produktów
 * i ich liczebności oraz koszt wysyłki jednej pozycji.
 *
 * @returns {number} - całkowity koszt wysyłki wszystkich zamówionych produktów.
 */
var getShippingTotal = function (lineItems) {
    var shippingTotal = 0;

    for (var i = 0; i < lineItems.length; i++) {
        var lineItem = lineItems[i];
        shippingTotal += lineItem.shippingPrice * lineItem.quantity;
    }

    return shippingTotal;
};

/**
 * Oblicza całkowity rabat, który ma być odjęty od całkowitej wartości zamówienia.
 *
 * @param {number} lineItemTotal - całkowita wartość wszystkich zamówionych produktów.
 *
 * @param {string} discountCode - opcjonalny kod rabatu, który może spowodować
 * uwzględnienie rabatu przed dodaniem kosztów wysyłki i podatków.
```

```
 *
 * @returns {number} - całkowity rabat, który ma być odjęty od całkowitej wartości zamówienia.
 */
var getDiscountTotal = function (lineItemTotal, discountCode) {
    var discountTotal = 0;

    if (discountCode === '20PERCENT') {
        discountTotal = lineItemTotal * 0.2;
    }

    return discountTotal;
};

/**
 * Oblicza całkowity podatek, który ma być dodany do zamówienia.
 *
 * @param {number} lineItemTotal - całkowita wartość wszystkich zamówionych produktów.
 *
 * @param {Object} customer - kolekcja informacji na temat osoby, która złożyła zamówienie.
 *
 * @returns {number} - całkowita wartość podatku dodana do zamówienia.
 */
var getTaxTotal = function () {
    var taxTotal = 0;

    if (customer.shiptoState === 'CA') {
        taxTotal = lineItemTotal * 0.08;
    }

    return taxTotal;
};
```

Każdej nowej funkcji powinien również towarzyszyć test jednostkowy, taki jak ten pokazany w listingu 1.6.

Listing 1.6. Testy jednostkowe dla wyodrębnionych funkcji napisanych w języku JavaScript

```
/**
 * Przyjmuje asercję, że getLineItemTotal działa zgodnie z oczekiwaniami.
 */
var testGetLineItemTotal = function () {
    var lineItem1 = {
        price: 50,
        quantity: 1
    };
```

```
    var lineItem2 = {
        price: 100,
        quantity: 2
    };

    var lineItemTotal = getLineItemTotal([lineItem1, lineItem2]);
    var expectedTotal = 250;

    if (lineItemTotal === expectedTotal) {
      successfulTestCount++;
    } else {
      unsuccessfulTestCount++;
      unsuccessfulTestSummaries.push(
          'testGetLineItemTotal: oczekiwane ' + expectedTotal + ';
          ↪otrzymane ' + lineItemTotal
      );
    }
};

/**
 * Przyjmuje asercję, że getShippingTotal działa zgodnie z oczekiwaniami.
 */
var testGetShippingTotal = function () {
    var lineItem1 = {
        quantity: 1,
        shippingPrice: 10
    };

    var lineItem2 = {
        quantity: 2,
        shippingPrice: 20
    };

    var shippingTotal = getShippingTotal([lineItem1, lineItem2]);
    var expectedTotal = 250;

    if (shippingTotal === expectedTotal) {
      successfulTestCount++;
    } else {
      unsuccessfulTestCount++;
      unsuccessfulTestSummaries.push(
          'testGetShippingTotal: oczekiwane ' + expectedTotal + ';
          ↪otrzymane ' + shippingTotal
      );
    }
};

/**
 * Upewnia się, że GetDiscountTotal działa zgodnie z oczekiwaniami,
```

```
 * gdy zastosowany zostanie prawidłowy kod rabatu.
 */
var testGetDiscountTotalWithValidDiscountCode = function () {
    var discountTotal = getDiscountTotal(100, '20PERCENT');
    var expectedTotal = 20;

    if (discountTotal === expectedTotal) {
      successfulTestCount++;
    } else {
      unsuccessfulTestCount++;
      unsuccessfulTestSummaries.push(
          'testGetDiscountTotalWithValidDiscountCode: oczekiwane ' +
          ↪expectedTotal + '; otrzymane ' + discountTotal
      );
    }
};

/**
 * Upewnia się, że GetDiscountTotal działa zgodnie z oczekiwaniami,
 * gdy zastosowany zostanie nieprawidłowy kod rabatu.
 */
var testGetDiscountTotalWithInvalidDiscountCode = function () {
    var discountTotal = get_discount_total(100, '90PERCENT');
    var expectedTotal = 0;

    if (discountTotal === expectedTotal) {
      successfulTestCount++;
    } else {
      unsuccessfulTestCount++;
      unsuccessfulTestSummaries.push(
          'testGetDiscountTotalWithInvalidDiscountCode: oczekiwane ' +
          ↪expectedTotal + '; otrzymane ' + discountTotal
      );
    }
};

/**
 * Upewnia się, że GetTaxTotal działa zgodnie z oczekiwaniami, gdy klient mieszka w Kalifornii.
 */
var testGetTaxTotalForCaliforniaResident = function () {
    var customer = {
      shiptoState: 'CA'
    };

    var taxTotal = getTaxTotal(100, customer);
    var expectedTotal = 8;

    if (taxTotal === expectedTotal) {
      successfulTestCount++;
```

```
    } else {
      unsuccessfulTestCount++;
      unsuccessfulTestSummaries.push(
          'testGetTaxTotalForCaliforniaResident: oczekiwane ' +
          ↪expectedTotal + '; otrzymane ' + taxTotal
      );
    }
};

/**
 * Upewnia się, że GetTaxTotal działa zgodnie z oczekiwaniami, gdy klient nie mieszka w Kalifornii.
 */
var testGetTaxTotalForNonCaliforniaResident = function () {
    var customer = {
        shiptoState: 'MA'
    };

    var taxTotal = getTaxTotal(100, customer);
    var expectedTotal = 0;

    if (taxTotal === expectedTotal) {
      successfulTestCount++;
    } else {
      unsuccessfulTestCount++;
      unsuccessfulTestSummaries.push(
          'testGetTaxTotalForNonCaliforniaResident: oczekiwane ' +
          ↪expectedTotal + '; otrzymane ' + taxTotal
      );
    }
};
```

Na koniec funkcję `getOrderTotal` należy zmodyfikować w taki sposób, aby korzystała z tych nowych funkcji, tak jak widać w listingu 1.7.

Listing 1.7. Modyfikowanie getOrderTotal w celu wykorzystania wyodrębnionych funkcji

```
/**
 * Oblicza całkowitą wartość zamówienia po uwzględnieniu kosztów wysyłki, rabatów i podatków.
 *
 * @param {Object} customer - kolekcja informacji na temat osoby, która złożyła zamówienie.
 *
 * @param {Array.<Object>} lineItems - kolekcja zakupionych produktów
 * i ich liczebności oraz koszt wysyłki jednej pozycji.
 *
 * @param {string} discountCode - opcjonalny kod rabatu, który może spowodować
 * uwzględnienie rabatu przed dodaniem kosztów wysyłki i podatków.
 */
```

```
var getOrderTotal = function (customer, lineItems, discountCode) {
    var lineItemTotal = getLineItemTotal(lineItems);
    var shippingTotal = getShippingTotal(lineItems);
    var discountTotal = getDiscountTotal(lineItemTotal, discountCode);
    var taxTotal = getTaxTotal(lineTtemTotal, customer);

    return lineItemTotal - discountTotal + shippingTotal + taxTotal;
};
```

Po przeanalizowaniu powyższego kodu można poczynić następujące obserwacje:

- teraz jest więcej funkcji niż przedtem;
- jest więcej testów jednostkowych niż wcześniej;
- każda funkcja robi jedną konkretną rzecz;
- każdej funkcji towarzyszy test jednostkowy;
- funkcje mogą być wykorzystywane razem, aby wykonywać bardziej złożone obliczenia.

Ogólnie rzecz biorąc, ten kod jest teraz w znacznie lepszej kondycji. Wyodrębnione zostały poszczególne obliczenia wykorzystywane w funkcji `getOrderTotal`, a każdemu z nich towarzyszy test jednostkowy. Oznacza to, że znacznie łatwiej będzie wskazać, który dokładnie fragment funkcjonalności zostanie uszkodzony, gdy do kodu wkradnie się błąd. Dodatkowo, jeśli całkowity koszt podatku lub wysyłki trzeba będzie obliczyć w innym fragmencie kodu, będzie można użyć istniejącej funkcjonalności, która ma już testy jednostkowe.

Refaktoryzacja. Przykład 2. Prosty przykład refaktoryzacji kodu CSS

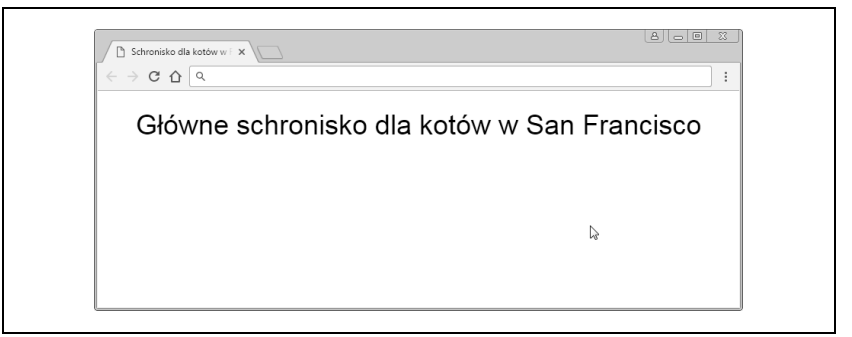
Listing 1.8 to kod wyświetlający nagłówek strony internetowej.

Listing 1.8. Kod HTML dla nagłówka strony internetowej

```
<!doctype html>
<html>
  <head>
    <title>Schronisko dla kotów u Fergusona</title>
    <link rel="stylesheet" type="text/css" href="css/style.css" />
  </head>
  <body>
    <main>
      <h1 style="font-family: Helvetica, Arial, sans-serif;font-size: 36px;
        font-weight: 400;text-align: center;">
        Główne schronisko dla kotów w San Francisco
```

```
      </h1>
    </main>
  </body>
</html>
```

Po otwarciu pliku *index.html* w przeglądarce zobaczysz to, co zostało przedstawione na rysunku 1.3.



Rysunek 1.3. Zrzut ekranu dla nagłówka strony internetowej

W naszym pierwszym przykładzie przed przeprowadzeniem refaktoryzacji napisaliśmy test jednostkowy dla kodu, aby upewnić się, że jego zachowanie nie uległo zmianie. Podczas refaktoryzowania kodu CSS również należy mieć pewność, że modyfikacje niczego nie zmienią, ale niestety nie jest to takie proste, ponieważ testowane jest coś wizualnego, a nie coś, co produkuje wartości dyskretne. Rozdział 5. omawia użyteczne techniki zachowywania jakości wizualnej. Na razie jednak wystarczy, że zrobiliśmy po prostu zrzut ekranu przed refaktoryzacją, aby zapewnić odniesienie wizualne.

Refaktoryzacja nagłówka strony internetowej

Gdy spojrzymy na kod z listingu 1.8, jasne stanie się, że istnieje pole do poprawy, ponieważ style nagłówka (oznaczonego znacznikiem `<h1>`) są osadzone w atrybucie `style`. Gdy style są osadzone w kodzie HTML za pomocą atrybutu `style` elementu lub pomiędzy znacznikami `<style></style>`, nazywamy je **stylami lokalnymi** (ang. *inline styles*).

Podobnie jak pierwotna funkcja z listingu 1.1, która wykonywała wiele obliczeń, style lokalne nie są raczej wielokrotnego użytku. Gdy style są ustawiane przy użyciu atrybutu `style`, mogą być zastosowane tylko do tego konkretnego elementu.

Kiedy style są osadzone pomiędzy znacznikami `<style></style>`, mogą być zastosowane tylko do danej strony.

Ponieważ większość witryn internetowych ma wiele stron, a każda z nich może mieć nagłówek, style te powinny zostać wyodrębnione z kodu HTML do osobnego pliku CSS (w tym przypadku *style.css*), który może zostać dołączony do wielu stron i zbuforowany przez przeglądarkę. Zawartość pliku *style.css* przedstawiono w listingu 1.9, a listing 1.10 pokazuje HTML z wyodrębnionym lokalnym kodem CSS.

Listing 1.9. Kod CSS nagłówka wyodrębniony do pliku style.css

```
h1 {
  font-family: Helvetica, Arial, sans-serif;
  font-size: 36px;
  font-weight: 400;
  text-align: center;
}
```

Listing 1.10. HTML z wyodrębnionym lokalnym kodem CSS

```
<!doctype html>
<html>
  <head>
    <title>Schronisko dla Kotów u Fergusona</title>
    <link rel="stylesheet" type="text/css" href="css/style.css" />
  </head>
  <body>
    <main>
      <h1>Główne schronisko dla kotów w San Francisco</h1>
    </main>
  </body>
</html>
```

Szybkie odświeżenie przeglądarki pokazuje, że nic się nie zmieniło, i po raz kolejny można poczynić pewne obserwacje:

- wyodrębnianie lokalnego CSS promuje ponowne wykorzystanie;
- rozdzielenie funkcjonalności (stylów i struktury) sprawia, że kod jest czytelniejszy;
- testy regresyjne mogą być wykonywane ręcznie w przeglądarce internetowej lub poprzez porównanie zrefaktoryzowanego interfejsu ze zrzutem ekranu.

Wyodrębnianie stylów do osobnego pliku promuje ponowne wykorzystanie kodu, ponieważ te style mogą być używane w wielu stronach. Gdy CSS znajduje się

w innym pliku niż HTML, zarówno HTML, jak i CSS są łatwiejsze do odczytania, ponieważ HTML nie zawiera wyjątkowo długich linii definicji stylów, a CSS jest pogrupowany w logicznych blokach. Ponadto testowanie zmian może być wykonywane poprzez ręczne przeładowywanie strony w przeglądarce, więc zmiany można porównywać ze zrzutem ekranu, który został zrobiony przed refaktoryzacją.

Chociaż ten przykład był bardzo prosty, wiele takich drobnych zmian może z upływem czasu przynieść spore korzyści.

Podsumowanie rozdziału

Przebrnęliśmy przez pierwszy rozdział i wiemy już, czym jest refaktoryzacja i w jaki sposób odnosi się ona do architektury oprogramowania. Dowiedzieliśmy się także, dlaczego refaktoryzacja jest istotna i kiedy powinna być przeprowadzana. Na koniec przeanalizowaliśmy dwa przykłady refaktoryzacji i poznaliśmy testy jednostkowe. W następnej kolejności dowiemy się, czym jest kaskada, będąca bez wątpienia najważniejszą koncepcją, jaką trzeba zrozumieć, jeśli chodzi o pisanie CSS.

Skorowidz

R

S

T

U

W

Z

O'REILLY®

Refaktoryzacja — kodowanie w najlepszym stylu

Tworzenie nowoczesnych stron internetowych wymaga opanowania trzech kluczowych technologii: HTML, JavaScriptu i CSS. Ta ostatnia jest zaskakująco potężnym językiem, który ułatwia nadanie stronie atrakcyjnego wyglądu, a równocześnie pozwala zapewnić jej responsywność. Niezależnie od tego kod CSS należy tworzyć tak, aby był odpowiednio zorganizowany, czytelny i łatwy w utrzymaniu. Pomocna w osiągnięciu tego celu jest refaktoryzacja — technika polegająca na przeglądaniu kodu w celu usunięcia zbędnych elementów i niespójności.

Trzymasz w ręku praktyczny przewodnik dla projektantów stron internetowych. Przedstawiono tu szereg istotnych zagadnień, takich jak architektura kodu CSS, sposób jego zorganizowania, a przede wszystkim cele i strategie refaktoryzacji kodu. Książka ta będzie przydatna również dla tych, którzy dopiero zaczynają naukę CSS, jednak chcą od razu tworzyć kod czytelny, spójny, łatwy w utrzymaniu. Dzięki lekturze zyskasz wiedzę pozwalającą na bezproblemowe tworzenie, testowanie i wielokrotne wykorzystywanie skryptów CSS.

Steve Lindstrom — pierwszą stronę internetową napisał w 1999 roku, jako uczeń szkoły średniej. Od tego czasu tworzy strony WWW i inne aplikacje — ma w tym zakresie ogromne doświadczenie. Często programuje dla branży obronnej, projektuje też aplikacje ułatwiające zarządzanie podróżami. Od pewnego czasu tworzy oprogramowanie dla branży handlu elektronicznego.

W tej książce znajdziesz między innymi:

- wyjaśnienie pojęcia refaktoryzacji i opis korzyści z jej stosowania
- odniesienie refaktoryzacji do architektury oprogramowania
- omówienie takich aspektów CSS jak kaskada, szczegółowość selektorów i model pola
- wyjaśnienie standardów kodowania i bibliotek wzorców
- organizowanie i testowanie kodu CSS
- strategie refaktoryzacji CSS

Helion

księgarnia internetowa

http://helion.pl

zamówienia telefoniczne

0 801 339900

0 601 339900

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

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

ISBN 978-83-283-2098-7

9 788328 320987

Informatyka w najlepszym wydaniu

cena: 34,90 zł