www.bnsit.pl

# Jak całkowicie odmienić sposób programowania używając REFAKTORYZACJI

Mariusz Sieraczkiewicz


*Poznaj tajniki najlepszych programistów!*

*Aby zrozumieć, musisz doświadczyć.*

bns it

Mariusz Sieraczkiewicz

**Książka z serii Mistrz Programowania®**

# Jak całkowicie odmienić sposób programowania używając refaktoryzacji

Przedmowa Michał Bartyzel

Data: 09.03.2009

Wersja 151

Tytuł: Jak całkowicie odmienić sposób programowania używając refaktoryzacji
Autor: Mariusz Sieraczkiewicz



Pierwsze trzy rozdziały tej książki są udostępniane bezpłatnie. Pozostałe rozdziały dostępne są w płatnej wersji książki. Zabronione są jakiekolwiek zmiany w zawartości publikacji bez pisemnej zgody BNS IT s. c. Zabrania się jej odsprzedaży.

## Aby dowiedzieć się więcej o pełnej wersji, kliknij tutaj



BNS IT s. c.
ul. Wyszyńskiego 22 lok. 17
94 - 042 Łódź
www.bnsit.pl
biuro@bnsit.pl

*Mojej najwspanialszej żonie Kamili i córce Poli, którą już niedługo zobaczę*

## Mariusz Sieraczkiewicz

Trener, konsultant, menedżer projektów IT, coach. Założyciel zespołu programistów Equilibrium. Współinicjator JUGa Łódź. Autor artykułów o inżynierii oprogramowania. Współwłaściciel firmy szkoleniowej BNS IT.
Z językiem Java ma do czynienia od dziewięciu lat na stanowiskach programisty, projektanta, architekta, analityka biznesowego, kierownika projektów i kierownika zespołu.
Jego pasją jest programowanie i aspekty psychologiczne pracy programistów. Poszukuje wyznaczników efektywności programisty szczególnie w obszarze zarządzania czasem, inteligencji emocjonalnej, przekonań, pracy w zespole, motywacji i ich wpływu na codzienną pracę.

# Spis treści

# Przedmowa

Większość programistów wie, co to refaktoryzacja, zna zalety wynikające z jej stosowania, zna również konsekwencje zaniedbywania refaktoryzacji. Jednocześnie wielu programistów uważa, że refaktoryzacja to bardzo kosztowny proces, wymaga wysiłku i brak na nią czasu w szybko zmieniających się warunkach biznesowych.

W tej niesamowitej książce, Mariusz zaprezentuje Ci kilka bardzo prostych sposobów refaktoryzacji kodu. Dowiesz się, jak w łatwy sposób stosować refaktoryzację w twoim projekcie. Ostrzegam, że legną w gruzach twoje z trudem wypracowane przekonania. Być może zaczniesz uważać, że refaktoryzacja jest niezwykle prosta, zajmuje bardzo mało czasu i nie sprawia trudności. Ze zdziwieniem będziesz się zastanawiać, jak to się stało, że wcześniej nie używałeś tych technik. Powodzenia!

Michał Bartyzel

# Wstęp

Na płycie nagrobnej pewnego anglikańskiego biskupa (ok. 1100 r.) spoczywającego w krypcie Opactwa Westminsterskiego widnieją następujące słowa:

Kiedy byłem młody i wolny, a moja wyobraźnia nie znała granic, marzyłem o tym, by zmienić świat.

Gdy wzrosłem w latach i mądrości, zrozumiałem, że świata nie da się zmienić.

Więc przykróciłem nieco swe zamiary i postanowiłem zmienić jedynie swój kraj.

Lecz i on także pozostał niezmienny.

Gdy dożyłem swego zmierzchu, w ostatnim, rozpaczliwym zrywie zdecydowałem zmienić choć swoją rodzinę, istoty mi najbliższe.

Lecz niestety! Na nic się to zdało!

Teraz zaś, gdy spoczywam na łożu śmierci, nagle zdałem sobie sprawę, że jeślibym zmienił najpierw tylko samego siebie, to być może swym przykładem zmieniłbym i swą rodzinę. Przy jej inspiracji i wsparciu byłbym w stanie ulepszyć swój kraj, i kto wie, może zmieniłbym wówczas cały świat.

## Zacząć od siebie

Rzeczywistość projektowa jest złożona.

- Zazwyczaj brakuje czasu.

- Trzeba pracować na starym kodzie, który pozostawia wiele do życzenia.
- Koledzy z biurka obok nie kwapią się do tego, żeby zacząć lepiej pisać programy.
- Klienci nie widzą czego chcą.
- Szefowi zależy tylko i wyłącznie na najszybszym wykonywaniu zadań.
- Biblioteki są źle udokumentowane i zawierają błędy.
- Narzędzia nie działają do końca tak jak powinny.

Jest wiele przeszkód, które utrudniają tworzenie oprogramowania i niestety zawsze tak będzie. Chcielibyśmy, żeby **wszystko dookoła nas się zmieniło**, żebyśmy mogli w końcu zacząć programować w przyzwoity sposób. Czy nie przypomina to trochę sytuacji biskupa z Opactwa Westminsterskiego?

## Dla kogo jest ta książka

Jeśli z jakiegokolwiek powodu temat refaktoryzacji wydaje ci się interesujący - ta książka jest dla ciebie. Została napisana w taki sposób, aby każdy kto nie miał do tej pory styku z tym tematem, mógł się go nauczyć i rozpocząć stosowanie techniki refaktoryzacji. **Jest to książka dla osób początkujących w temacie refaktoryzacji, wystarczy tylko, że znasz obiektowy język programowania i możesz zaczynać**.

Książka którą masz w ręku oparta jest o język programowania Java, jednak wiele z przedstawionych technik można zastosować z powodzeniem w innych językach. Refaktoryzacja jest tematem uniwersalnym i niezależnym od języka programowania.

Uwaga!

Jeśli znasz dobrze inny język programowania i chciałbyś opracować wersję tej książki w tym języku napisz do nas na ebooks@bnsit.pl — staniesz się współautorem.

Dla osób, które znają już tę technikę może ona stanowić punkt odniesienia dla stosowanych przez siebie metod.

Jeśli zdecydowałeś się na dalsze czytanie tej książki, mam prośbę, abyś zastanowił się nad celem, intencją, która ma towarzyszyć jej przeczytaniu. Być może będziesz chciał zwiększyć swoją efektywność, być może będziesz chciał doprowadzić do tego, aby twój kod źródłowy stał się czytelny, a być może po prostu będziesz chciał spędzić miło czas. Zapisz sobie ten cel i niech ci towarzyszy podczas lektury.

**Moja intencja**

## Podziękowania

Stworzenie książki wymaga sporo wysiłku, wsparcia i pomocy innych.

Chciałbym podziękować mojej wspaniałej żonie Kamili za ogromne wsparcie w zrealizowaniu tego szaleńczego dzieła.

Chciałbym podziękować Michałowi Bartyzelowi. Długie dyskusje z nim doprowadziły do wielu wniosków zawartych w tej książce.

Chciałbym podziękować osobom, które krytycznym okiem spojrzały na tę książkę, przyczyniając się do jej rozwoju — Piotrowi Majowi, Jackowi Laskowskiego, Piotrowi Stróżykowi, Marcinowi Kasińskiemu, Grzegorzowi Smoragowi, Maciejowi Kubiszowi i Teresie Sas.

Chciałbym podziękować wszystkim osobom z zespołu Equilibrium, gdyż to dzięki nim, mogłem się tak wiele nauczyć i odkrywać tajniki pracy programistów.

# Rozdział 1

## Refaktoryzacja - o co tyle krzyku

Dawno, dawno temu, tworzyłem swoje pierwsze zaawansowane programy. Sprawiało mi to niesamowitą przyjemność, mogłem stworzyć coś z niczego, a tworzone przeze mnie aplikacje były użyteczne. Mimo tej ogromnej radości tworzenia oprogramowania, wewnętrznie pojawiał się pewien niedosyt. Tworzony kod z czasem stawał się coraz bardziej skomplikowany, podobne konstrukcje się powtarzały i miałem wrażenie, że tracę kontrolę nad tym, co robię.
Zazwyczaj wyglądało to tak, iż przez wiele godzin programowałem swoje dzieło, a następnie modliłem się, żeby zadziałało. Najczęściej nie działało. Metodą prób i błędów po długim czasie naprawiałem kolejne niedociągnięcia. Ciągle towarzyszyła mi myśl: „Obym nie musiał niczego później zmieniać w tym, co napisałem".

W tym czasie wydawało mi się, że właśnie taka jest specyfika pracy programisty. Programowanie to dziedzina inżynieryjna, ludzie tworzą jedne z najbardziej skomplikowanych tworów. „Tak musi być!" — myślałem, choć przeczucie mówiło mi, że jest inny sposób.

Od tego czasu wiele wody w rzekach upłynęło i poznałem wiele wspaniałych obiektowych technik tworzenia oprogramowania, iż teraz już wiem, że programowanie może być prostsze. A co najważniejsze mam pewność, że nad tworzonym oprogramowaniem można mieć pełną kontrolę, może być napisane czytelnie i może być przygotowane do dalszego rozwoju.

Jedną z technik, która przybliża do wypracowania dobrego stylu pracy, jest refaktoryzacja.

## Czym jest refaktoryzacja?

Czym zatem jest refaktoryzacja i w czym może pomóc? Jednym z największych wyzwań programisty związanych z tworzeniem kodu, jest jego czytelność i łatwość modyfikacji. Można powiedzieć, że obecnie nie jest sztuką umiejętność napisania rozwiązania w wybranym języku programowania. Ze względu na złożoność współczesnych systemów informatycznych, ważniejsze jest, aby łatwo było je przeanalizować, zrozumieć i zmodyfikować. Nade wszystko - czytelność.

**Definicja**

Refaktoryzacja to zmiana dokonana w wewnętrznej strukturze oprogramowania w celu jego łatwiejszego zrozumienia i modyfikacji bez zmiany obserwowalnego zachowania.

## Postulaty nowoczesnej inżynierii oprogramowania, czyli odpowiedź na pytanie „Dlaczego?"

Często zdarza mi się słyszeć stwierdzenie, że na refaktoryzację nie ma czasu lub że to kosztuje zbyt wiele. Dlaczego zmiana struktury rozwiązania ma takie znaczenie i czy warto poświęcać na tego typu działania bezcenny czas programisty?

### Postulaty dotyczące współczesnej inżynierii oprogramowania

Kiedy myślimy o tworzeniu oprogramowania, najczęściej nasuwa się na myśl sam etap implementowania kodu. Tymczasem jest to tylko jeden z elementów układanki nazywanej wytwarzaniem oprogramowania. Najważniejsze dwa postulaty związane z inżynierią oprogramowania brzmią następująco[1]:

- **kod oprogramowania jest częściej czytany niż pisany** — samo napisanie rozwiązania to tylko część wysiłku potrzebnego do tego, aby rozwiązanie mogło działać, i aby było rozwijane np. programista poprawiając błędy w swoim fragmencie, musi go wielokrotnie czytać; kod jest czytany podczas

[1] Kent Beck, Implementation Patterns, Addison Wesley Publishing Company, 2007

przeglądów (ang. review); kod jest poddawany testowaniu; jest analizowany w momencie potrzeby wprowadzenia zmiany;

- **w procesie wytwarzania oprogramowania często nie istnieje stwierdzenie „zrobione"** — oprogramowanie ciągle się rozwija, jeśli powstała wersja pierwsza, prawdopodobnie powstanie wersja druga, system wdrożony w przedsiębiorstwie, w banku czy innej organizacji będzie ciągle rozwijany, aby dostosować się do zmieniającej się rzeczywistości; szacuje się, że ok. 80−90% profesjonalnych programistów pracuje w projektach, które polegają na rozwoju istniejących rozwiązań; niewielu programistów ma możliwość brania udziału w projekcie tworzonym całkowicie od początku.

## Koszt wytwarzania oprogramowania

Koszt wytworzenia oprogramowania (od momentu rozpoczęcia prac implementacyjnych) to średnio w 30% koszt samego programowania, a w 70% koszt utrzymania rozwiązania[2]. Koszt utrzymania to między innymi koszt poprawy błędów, analizy rozwiązania, zmiany, przetestowania i wdrożenia.

Niezwykle istotne jest, aby zadbać o owe 70%. Obniżenie tego kosztu można osiągnąć dążąc do realizacji trzech wartości:

- **komunikatywność** — tworzenie rozwiązań w taki sposób, aby były czytelne i możliwie łatwe w analizie;
- **prostota** — dobre rozwiązania są zazwyczaj proste;
- **elastyczność** – stworzony kod powinien być przygotowany na zmiany, tam gdzie te *zmiany są realne i wynikają z wymagań*.

Dzięki refaktoryzacji możemy w usystematyzowany sposób dążyć do realizacji tych wartości.

## Entropia wzrasta

Istnieje w fizyce II zasada termodynamiki o wzroście entropii w układzie zamkniętym, którą możemy sparafrazować do zastosowania w inżynierii oprogramowania.

[2]Kent Beck, Implementation Patterns, Addison Wesley Publishing Company, 2007

**Ważne**

Z upływem czasu stopień nieuporządkowania w projekcie informatycznym wzrasta.

Cóż to oznacza? Im dłużej jest realizowany projekt, tym więcej pojawia się w nim rozwiązań, które nie są już używane lub realizują zadania w bardziej skomplikowany sposób niż by mogły. Pojawia się mnóstwo nazw w projekcie, które nie są już adekwatne lub wręcz są mylne. W projekcie istnieje mnóstwo obejść, które umożliwiają dopasowanie nowych rozwiązań do starych. Pojawiają się rozwiązania tymczasowe, które ostatecznie nie są nigdy poprawiane. Można by tak długo wymieniać. Entropia układu, jakim jest system informatyczny, z czasem wzrasta i potrzeba planowanego działania, aby temu przeciwdziałać.

Tak jak każdy dom wymaga sprzątania, tak samo sprzątania wymaga system informatyczny. Można oczywiście przez pewien czas nie robić porządków i akceptować pewien poziom bałaganu, ale z czasem poruszanie się po projekcie staje się coraz bardziej nieprzyjemne, a nawet niemożliwe. Byłem świadkiem wielu sytuacji, w których komercyjne systemy wymagały przepisania, właśnie dlatego, że nie stosowano w ogóle refaktoryzacji.

Dzięki refaktoryzacji możemy posprzątać i doprowadzić system do stanu używalności.

## Refaktoryzacja w locie

W tym momencie może pojawić się pytanie: *„W takim razie mam teraz zaprzestać prac i zrobić porządki w całym systemie?"* W dużej części przypadków może to być albo nierealne, albo nierozsądne. Jest niewiele projektów, które nagle można przerwać i rozpocząć dłuższy proces refaktoryzacji.

Najważniejsze jest to, aby refaktoryzacja stała się stałym elementem realizowanych projektów informatycznych, żeby stała się nawykiem. Jeśli programista posiądzie umiejętności dokonywania refaktoryzacji na wysokim poziomie, będzie potrafił dokonywać tzw. **refaktoryzacji w locie** — w momencie tworzenia rozwiązania będzie rozpoznawał miejsca, które należy skonstruować inaczej, aby

rozwiązanie w najbliższej przyszłości nie wymagało refaktoryzacji (czyli aby było zrefaktoryzowane w momencie tworzenia). Choć wydaje się to brzmieć nieco abstrakcyjnie, zapewniam, iż taki stan osiąga się po kilku tygodniach intensywnego stosowania refaktoryzacji.

Umiejętność refaktoryzacji w locie, jest o tyle ważna, że refaktoryzacja w naszym rozumieniu to **proces**, który musi trwać bezustannie, aby projekt płynnie się rozwijał.

## Refaktoryzacja ewolucyjna

Co zatem w sytuacji, gdy istnieje kod, który wyraźnie wymaga refaktoryzacji, a nie ma takiej możliwości, aby poświęcić całą energię na jego zmianę. Proponujemy podejście ewolucyjne. Proces ten nazywam *Czteroetapowym modelem refaktoryzacji ewolucyjnej*:

1. zidentyfikuj miejsca, które ulegają najczęstszym zmianom
2. określ rodzaj refaktoryzacji, który chciałbyś wprowadzić
3. jeśli nie masz testu danego fragmentu, stwórz go
4. zrefaktoryzuj lokalnie — dokonaj najmniejszej możliwej zmiany

Główną ideą tego modelu jest zmiana tych elementów systemu, które są najczęściej modyfikowane, gdyż to koszt modyfikacji jest w tym przypadku najwyższy. Szczególnie, jeśli stworzone rozwiązane nie posiada konstrukcji ułatwiających ich dalszy rozwój.

## Najważniejsza ze wszystkich zasad - wydzielanie odpowiedzialności

Proces refaktoryzacji jest najbardziej efektywny, jeśli będzie oparty o najważniejszą zasadę związaną z programowaniem obiektowym - wydzielaniem odpowiedzialności. **Odpowiedzialność** to pojęcie pierwotne, za pomocą którego można sformułować niemal wszystkie inne zasady stojące za technikami refaktoryzacji, za wzorcami implementacyjnymi czy projektowymi[3].

[3]Jedna z definicji brzmi następująco: Powód wprowadzenia zmian w klasie. Jednak jest to tak skondesowana definicja, iż w naszych rozważaniach jest bezużyteczna

**Ważne**

Odpowiedzialność to najważniejsze pojęcie związane z obiektowością.

Jeśli odpowiednio nauczysz posługiwać się odpowiedzialnością w praktyce, każda dziedzina inżynierii oprogramowania stanie przed tobą otworem.

Osobiście traktuję to pojęcie bardzo szeroko. Możemy je lapidarnie zdefiniować jako to, „czym dany element się zajmuje".

**Definicja**

Odpowiedzialność określa to, czym dany element się zajmuje

Odpowiedzialność dotyczy każdego aspektu tworzonego systemu:

- klasy - za co odpowiada klasa, jakie metody i atrybuty z tego wynikają,
- metody - za co odpowiada metoda i jaki wynik zwraca,
- zmiennej, pola klasy, stałej - za co odpowiada zmienna, jakie informacje przechowuje,
- pakietu - jakiego typu elementy powinny się w nim znaleźć,
- podsystem, moduł - z jakim procesem biznesowym jest związany.

W przypadku klas z odpowiedzialności jasno musi wynikać, co z obiektem danej klasy można zrobić, a czego nie. Odpowiedzialność jest ściśle powiązana z nazwą.

Jeśli nazwa klasy nie oddaje tego, za co klasa jest odpowiedzialna, to mamy dwie możliwości: albo nazwa klasy jest nieprawidłowa, albo klasa realizuje coś innego niż powinna.

Jeśli nazwa metody nie odzwierciedla tego, co robi dana metoda i jaki wynik zwraca, to albo nazwa metody jest nieprawidłowa, albo odpowiedzialność metody wymknęła się spod kontroli.

Analogiczne rozumowanie można by przenieść na pozostałe elementy systemu — zmienne, pola klasy, stałe, pakiety i moduły.

# Antywzorce w kodzie

Jak zatem rozpoznawać miejsca w systemie, które należy zrefaktoryzować? Po pierwsze intuicja. Być może nie jest to przekonujące dla ścisłych umysłów programistów, ale zapewniam cię, że jeśli masz odczucie, że rozwiązanie, które stworzyłeś jest nienajlepsze, albo że trudno ci się nad nim pracuje, albo nie chciałbyś do niego zaglądać za dwa miesiące, to najprawdopodobniej nadaje się do refaktoryzacji.

Z drugiej strony programowanie to dziedzina techniki, zatem przyda się kilka reguł, które pomogą zlokalizować fragmenty kodu, które należy zrefaktoryzować. Tego typu reguły i wskazówki są nazywane **zapachami kodu** (ang. code smells).

Poniżej znajdują się zapachy kodu, które w ten czy inny sposób wynikają z nienajlepszej realizacji zasady odpowiedzialności.

## Don't repeat yourself (DRY) - likwiduj powtórzenia

Powtórzenia to wróg numer jeden programisty. Wynikają najczęściej ze stosowania metody kopiuj-wklej lub powielania podobnych rozwiązań w różnych miejscach. Jeśli znajdujesz w swoim kodzie fragmenty, które niewiele się różnią od siebie — prawdopodobnie warto je zrefaktoryzować np. poprzez wyodrębnienie nowej metody.

## Długie metody i duże klasy

Zbyt długie metody (kilkadziesiąt lub co gorsza kilkaset wierszy), to zazwyczaj zły znak — sygnał, że odpowiedzialność metody nie jest dobrze zdefiniowana. Analogicznie występowanie dużych klas z dużą ilością metod lub dużą ilością pól, to również sygnał alarmowy ostrzegający przed nieodpowiednio wydzieloną odpowiedzialnością klasy.

### Zmiana lub dodanie funkcjonalności powoduje konieczność wielu zmian w systemie

Kiedy wprowadzasz zmiany do systemu np. dodajesz nową funkcjonalność albo rozszerzasz istniejącą i musisz dokonywać zmian w wielu miejscach, warto się przyjrzeć tym miejscom. Prawdopodobnie można za pomocą wzorców projektowych wyodrębnić i odizolować to, co jest zmienne.

### Metoda danej klasy wykonuje zbyt wiele operacji na innych klasach

Jeśli metoda danej klasy pracuje przede wszystkim na obiekcie innej klasy, prawdopodobnie powinna znajdować się w tej drugiej klasie.

### Analogiczne instrukcje warunkowe w różnych miejscach w projekcie

Jeśli w kilku miejscach w systemie, przebieg operacji jest zależny od typu obiektu (seria instrukcji `if` lub konstrukcja `switch`), zamiast instrukcji warunkowych warto użyć polimorfizmu.

### Nie twórz bytów ponad miarę

Programowanie obiektowe wiąże się z dwoma zagrożeniami. Początkujący programiści obiektowi, obawiają się nieco klas i wychodzą z założenia, iż powinno być ich niewiele, co często prowadzi do zbyt długich metod lub zbyt dużych klas. Kiedy programista nabierze już sprawności w posługiwaniu się obiektowością, nierzadko wpada w drugą skrajność - tworzy nowe klasy przy każdej nadarzającej się okazji. Jeśli nowe klasy niewiele różnią się od innych, zlikwiduj je.

### Pola tymczasowe

W klasie istnieją atrybuty, które potrzebne są tylko w pewnych okolicznościach, np. do wykonania jednej konkretnej metody. Przechowywanie takiej jednorazowej informacji nie powinno leżeć w odpowiedzialności klasy, więc nie powinno być również polem w klasie.

## Klasa przechowująca tylko i wyłącznie dane

Szczególnie w Javie jest to sytuacja, która pojawia się dość często. Tworzone są klasy, które mają tylko i wyłącznie pola oraz metody dostępowe (ang. setters, getters), bez względu na to, czy będą potrzebne czy nie. Konwencja Java Beans wymaga od programisty stworzenia metod dostępowych do pól w klasie, do tego stopnia, iż w wielu narzędziach istnieje mechanizm, który generuje je automatycznie. W ten sposób powstają klasy, które poprzez metody dostępowe udostępniają stan obiektu na zewnątrz i przechowują wyłącznie dane. Klasa taka nie jest odpowiednio hermetyzowana oraz mamy do czynienia z tzw. anemicznymi obiektami modelu. Powstają w systemie dwie niezależne warstwy klas:

- klasy modelu - przechowujące dane, na przykład `User`,
- klasy usługowe - wykonujące operacje na tych danych, na przykład `UserManager`.

Następuje nadmierne mnożenie klas, zaburzenie odpowiedzialności obiektów oraz wiele obiektów jest ze sobą trwale powiązanych.

## Komentarze

Komentarze są złe. Tak powiedziałby konserwatywny zwolennik metodyki Extreme Programming. Komentarzy nikt nie lubi pisać, a jak już napisze, to później nikt ich nie aktualizuje. Nie wiadomo co jest gorsze - brak komentarza czy nieaktualny komentarz. Najlepszy jest samodokumentujący się kod.

Poza pierwszym zdaniem, zdecydowanie zgadzam się z tym stanowiskiem. Komentarze nie są z natury złe, ale są bardzo trudnym narzędziem i rzadko znajduje się odpowiednia motywacja, żeby z niego dobrze korzystać. Poza tym **w dużych projektach utrzymywanie dokumentacji implementacyjnej jest bardzo drogim przedsięwzięciem**. Jeśli to tylko możliwe, należy tworzyć kod, który sam się dokumentuje. Do którego nie potrzeba słownego wyjaśnienia, aby go zrozumieć. Jeśli przeczytasz w całości następny rozdział, to zobaczysz na żywym przykładzie, jak to robić.

**Ważne**

Jeśli masz ochotę napisać komentarz tekstowy, najpierw zastanów się czy wprowadzając prostą refaktoryzację (taką jak zmiana nazwy zmiennej, pola, klasy, wyodrębnienie metody) nie uzyskasz podobnego, łatwiejszego w utrzymaniu efektu.

## Testowanie

Na koniec dość istotna uwaga dotycząca refaktoryzacji. Technika ta polega na zmianie struktury kodu, stąd istnieje duże ryzyko, iż popełnimy błąd. Wtedy refaktoryzacja może przynieść więcej szkody niż pożytku. Dlatego nieodłącznym towarzyszem tej techniki są testy jednostkowe. Żeby bezpiecznie refaktoryzować, powinieneś mieć testy, które weryfikują poprawność zmienionego rozwiązania.

Dla osób które w projekcie nie posiadają wystarczającej liczby testów, jest dobra wiadomość — wiele współczesnych środowisk programistycznych automatyzuje proste refaktoryzacje, dzięki czemu możemy wykonywać je bezpiecznie, nawet jeśli nie posiadamy testów.

W tej książce aspekt testowania został pominięty. Jest to jedna z pozycji serii Mistrz Programowania®.

# Rozdział 2

# Kod, który czyta się jak książkę - techniki, które całkowicie odmienią twoje życie programisty

Głównym celem serii Mistrz Programowania® jest wskazanie najważniejszych 20% umiejętności, które pozwolą zrealizować 80% codziennych zadań. Ten rozdział dotyczy czterech najważniejszych technik związanych z refaktoryzacją. Jeśli chcesz poświęcić możliwie najmniej czasu po to, by poznać najbardziej praktyczne elementy refaktoryzacji, to powinieneś przeczytać ten rozdział. Zawiera on przykłady, które pokazują, jak kilka prostych, wdrożonych w życie nawyków, może mieć ogromne konsekwencje w codziennej pracy.

Pierwszym krokiem, który przybliży nas do celu opanowania umiejętności refaktoryzacji, będzie sztuka tworzenia kodu, który czyta się jak książkę, kodu, który się sam dokumentuje.

## Co to znaczy - czytać jak książkę

Jest pewne powiedzenie, który brzmi mniej więcej następująco:

**Ważne**

Nie jest sztuką stworzenie działającego kodu, który będzie zrozumiały dla komputera. Sztuką jest stworzenie działającego kodu, który będzie zrozumiały dla człowieka.

Mając do czynienia z programowaniem zapewne często spotykasz się z niezrozumiałym kodem. Często masz do czynienia z metodami złożonymi z kilkudziesięciu lub nawet kilkuset wierszy, z ogromną ilością zmiennych i zagnieżdżonymi instrukcjami sterującymi. Zrozumienie takiego zapisu jest niezwykle trudnym zadaniem, a modyfikacja jeszcze trudniejszym. Jest to antyprzykład „kodu, który czyta się jak książkę".

Jeśli masz do czynienia z kodem, którego metody są niezbyt rozbudowane, na podstawie zastosowanych nazw jesteś w stanie szybko zorientować, jak stworzone rozwiązanie realizuje swoje zadanie. Jeśli sposób, w jakim napisane są metody, jest przejrzystym zapisem użytego algorytmu, to można powiedzieć, że "kod czyta się jak książkę". Powyższy opis z pewnością nie wyczerpuje tego stwierdzenia. Najlepszy będzie konkretny przykład.

## Przykład

Jestem zwolennikiem praktycznego działania, dlatego ideę refaktoryzacji oraz ideę "kodu, który czyta się jak książkę" pokażę na przykładowym projekcie.

Zajmiemy się prostym systemem, którego zadaniem będzie tłumaczenie pojedynczych wyrazów z użyciem internetowych słowników języka angielskiego. Za pomocą stworzonego programu użytkownik będzie mógł znaleźć odpowiednik poszukiwanego słowa w innym języku i wykorzystania do własnych celów.

Na początku system będzie miał następujące funkcjonalności:

- użytkownik za pomocą menu wybiera opcje dostępne w systemie,

- użytkownik wpisuje wyraz w języku angielskim, system z użyciem wybranego internetowego słownika znajdzie jego tłumaczenia i wyświetli na ekranie,
- użytkownik może zapamiętać w systemie jedno ze znalezionych tłumaczeń,
- użytkownik może wyświetlić zapamiętane słowa,
- użytkownik może wyświetlić słowa znalezione w ostatnim zapytaniu.

System dla uproszczenia będzie posługiwał się interfejsem opartym o konsolę. Osoby mniej zorientowane zachęcam do zapoznania się z tematem strumieni w języku Java, w szczególności z klasą `java.util.Scanner`, która będzie głównym narzędziem pobierania danych.

**Uwaga!**

Kody zamieszczone poniżej dostępne są w wersji komercyjnej książki oraz są dołączane do warsztatów multimedialnych. Szczegóły można znaleźć na stronie http://www.bnsit.pl

W celu zorganizowania interfejsu użytkownika użyjemy narzędzi dostępnych w standardowej bibliotece języka Java. Pracując nad kodem będę stosować technikę *Małych kroków* tak, aby kolejne elementy aplikacji powstawały etapami - niewielkimi fragmentami.

**Ważne**

Technika **Małych kroków** jest zasadniczym sposobem pracy skutecznych programistów - dzieląc zadanie na mniejsze części łatwiej jest opanować zmiany zachodzące w kodzie i prościej jest odnaleźć ewentualne błędy.

Poniżej znajduje się kod, który będzie bazą dla konsolowego interfejsu użytkownika.

```
package pl.bnsit.webdictionary;

import java.util.Scanner;
```

```
public class ConsoleSample {


    public static void main(String[] args) {

        boolean ok = true;
        Scanner s = new Scanner( System.in );

        System.out.println( "Welcome to Web Dictionary System." );

        while (ok) {
            System.out.print("dictionary> ");
            String c = s.nextLine();

            if ( c.equals( "hello" ) ) {
                System.out.println(
                        "Welcome to Web Dictionary System." );

            } else if ( c.startsWith( "search" ) ) {
                String f = c.split( " " )[ 1 ];
                // TODO wyszukaj

            } else if ( c.startsWith( "save" ) ) {
                // TODO zapisz

            } else if ( c.equals( "showFound" ) ) {
                // TODO znalezione

            } else if ( c.equals( "showSaved" ) ) {
                // TODO zachowane
            } else if ( c.equals( "exit" ) ) {
                ok = false;
            }
        }

        s.close();
    }
}
```

**Ważne**

Jeśli nie znasz szczegółów dotyczących użytkowania strumieni w języku Java, znajdziesz je na stronie:
http://java.sun.com/docs/books/tutorial/essential/io/

Naszym głównym zadaniem jest odczytanie zawartości strony internetowej,

zawierającej dostępne tłumaczenia wybranego słowa z języka angielskiego. Proponuję w formie ćwiczenia wybrać dowolny słownik online i przygotować oparte na nim rozwiązanie. W tej książce użyjemy strony http://www.dict.pl.

**Ważne**

Pamiętajmy, że witryny ze słownikami internetowymi działają komercyjnie i są posiadaczami praw autorskich związanych ze słownikiem lub posiadają odpowiednie prawa użytkowania. W tej książce używamy ich tylko do celów zobrazowania technik refaktoryzacji. Nie należy wykorzystywać tych witryn do własnych celów (np. do działalności komercyjnej) bez upewnienia się, że jest to zgodne z prawem.

Poniżej znajduje się kod, który umożliwia odczytanie zawartości strony internetowej o zadanym adresie i wyświetlenie go na konsoli.

```
package pl.bnsit.webdictionary;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.MalformedURLException;
import java.net.URL;

public class WebDictionary {

    public static void main(String[] args) {

        BufferedReader r = null;

        try {
            r = new BufferedReader(new InputStreamReader(
                    new URL( "http://www.dict.pl/dict?word="
                            + "boy" + "&words=&lang=PL" )
                            .openStream()));

            String t = null;
            while ( (t = r.readLine()) != null ) {
                System.out.println( t );

            }

        } catch (MalformedURLException ex) {
            ex.printStackTrace();
```

```
        } catch (IOException ex) {
            ex.printStackTrace();

        } finally {
            try {
                if (r != null ) {
                    r.close();
                }
            } catch (IOException ex) {
                ex.printStackTrace();
            }
        }
    }

}
```

W celu pobrania zawartości strony używamy obiektu klasy `java.net.URL`. Za jego pomocą otrzymujemy dostęp do strumienia danych związanych z pobieraną stroną `URL.openStream()`. Jeśli nie używałeś wcześniej tych klas, proponuję poświęcić chwilę, aby z nimi poeksperymentować.

Jeśli przyjrzymy się bliżej efektowi działania powyższej metody, zauważymy, że polskie wyrazy nie są wyświetlane prawidłowo. Dzieje się tak, gdyż strona, z której korzystamy, jest zapisana z użyciem kodowania UTF−8. Tę kwestię możemy rozwiązać używając specjalnego konstruktora klasy `String`:

```
    String encodedString = new String( line.getBytes(), "UTF8" );
```

Trzecim elementem, który będzie potrzebny do zrealizowania zadania, jest mechanizm wyrażeń regularnych. Z jego pomocą odnajdziemy te wiersze kodu HTML strony internetowej, które zawierają tłumaczenia poszukiwanego wyrazu. Aby wykonać to zadanie, najpierw należy przeanalizować strukturę kodu HTML, aby doszukać się schematów charakterystycznych dla danych, które nas interesują.

**Ważne**

Poniżej podano przykłady, które są aktualne w momencie pisania tej książki. Za jakiś czas generowany kod HTML może wyglądać inaczej. Należy przede wszystkim zrozumieć zasadę przedstawioną poniżej tak, aby samodzielnie móc dostosować podany przykład do własnych potrzeb.

Zajmijmy się zatem analizą struktury strony słownika www.dict.pl. Możemy zauważyć, że pojedynczy wiersz zawierający słowo z polskim odpowiednikiem wygląda następująco:

```
... class="resWordCol"><a href="dict?word=chĹ,opiec&lang=PL">chĹ,opiec {m}</
    a></td>
```

Zaś pojedynczy wiersz z angielskim odpowiednikiem wygląda następująco:

```
... class="resWordCol"><a href="dict?word=boy&lang=PL"><font style="
    background: #E3ECFB; color: black">boy</font></a></td>
```

Ponadto najpierw następuje wiersz ze słowem polskim, później wiersz dla nas nieistotny, a następnie wiersz ze słowem angielskim.

```
... class="resWordCol"><a href="dict?word=chĹ,opiec&lang=PL">chĹ,opiec</a></
    td>
... <a href="http://www.google.com/custom?q=boy& ...
... class="resWordCol"><a href="dict?word=boy&lang=PL"><font style="
    background: #E3ECFB; color: black">boy</font></a></td>
```

Schemat, który możemy wyodrębnić w powyższym przykładzie wygląda następująco. Szukane wyrazy znajdują się w znaczniku `<a href`. Wyróżniającym elementem adresu są między innymi ciągi `dict?word` oraz `lang`. Sam wyraz znajduje się za sekwencją `dict?word=` lub `dict?words=`. Przykładowe wyrażenie regularne, które jest w stanie wyłuskać poszukiwane słowo, może wyglądać następująco:

```
Pattern pattern
        = Pattern.compile( ".*<a href=\"dict\\?words?=(.*)&lang.*" );
```

## Bardzo krótkie wprowadzenie do wyrażeń regularnych

Wyrażenia regularne służą do dopasowywania ciągów znakowych, które spełniają pewien schemat. Używając ich można stwierdzić, czy dany napis zawiera np. ciąg `<a href`. Może oczywiście zawierać inne znaki. Aby takie wyszukiwanie było możliwe, wyrażenia regularne zawierają specjalne znaki, które zastępują podciągi napisów. Na przykład:

- `.*` oznacza dowolną ilość dowolnych znaków
- `.` oznacza jeden dowolny znak
- `.?` oznacza zero lub jedno wystąpienie dowolnego znaku
- `\?` oznacza dosłownie wystąpienie znaku zapytania
- `\d*` oznacza dowolną ilość cyfr następujących po sobie

Zatem jeśli chcemy stwierdzić, czy w badanym ciągu znajduje się podciąg `<a href`, musimy skonstruować wyrażenie regularne postaci `.*<a href.*`, ponieważ oprócz szukanego podciągu mogą wystąpić również inne znaki.

Jeśli chcemy stwierdzić czy dany ciąg znakowy zawiera podciąg postaci `<a href="dict?word=` lub `<a href="dict?words=`, to wyrażenie regularne powinno wyglądać następująco:

```
.*<a href="dict\?words?=.*
```

Skonstruowane wyrażenie regularne należy ostatecznie zapisać w języku Java. Wymaga to trochę wysiłku, gdyż wyrażenia regularne zawierają znaki, które w zmiennych typu String są niedozwolone i należy je poprzedzić znakiem maskującym. Takimi znakami są m. in. " oraz \. Dlatego ostatecznie powyższe wyrażenie regularne będzie miało następującą postać:

```
String expression = ".*<a href=\"dict\\?words?=.*";
```

Największa moc wyrażeń regularnych objawia się wtedy, kiedy nie tylko chcemy stwierdzić, czy dany ciąg spełnia wyrażenie, ale wyłuskać coś z niego. W naszym przykładzie interesuje nas to, co się znajduje za znakiem = a przed ciągiem `&lang`, gdyż to

jest szukane przez nas słowo. Musimy zatem stworzyć tzw. grupę używając nawiasów zwykłych w celu oznaczenia znaków, które chcemy wyłuskać. Nasze wyrażenie będzie wyglądało następująco:

```
.*<a href="dict\?words?=(.*)&lang.*
```

W języku Java zapis będzie następujący:

```
String expression = ".*<a href=\"dict\\?words?=(.*)&lang.*";
```

**Ważne**

Jeśli chciałbyś dowiedzieć się więcej na temat wyrażeń regularnych, zajrzyj na stronę
http://java.sun.com/docs/books/tutorial/essential/regex/index.html

## Analiza struktury strony

Łącząc analizę wyrażeń regularnych z przykładem odczytu danych ze strony interentowej, uzyskujemy następujący kod:

```
package pl.bnsit.webdictionary;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class WebDictionary {

    public static void main(String[] args) {

        BufferedReader r = null;

        try {
            r = new BufferedReader(new InputStreamReader(
                    new URL( "http://www.dict.pl/dict?word="
                            + "boy" + "&words=&lang=PL" )
                            .openStream()));
```

```
            boolean polish = true;
            String t = null;
            Pattern pat = Pattern.compile(
                    ".*<a href=\"dict\\?words?=(.*)&lang.*" );
            while ( (t = r.readLine()) != null ) {

                Matcher matcher = pat.matcher( t );
                if (matcher.find()) {
                    String t2 = matcher.group(
                            matcher.groupCount() );
                    if ( polish ) {
                        System.out.print( t2 + " => " );
                        polish = false;
                    } else {
                        System.out.println( t2 );
                        polish = true;
                    }
                }
            }
        } catch (MalformedURLException ex) {
            ex.printStackTrace();

        } catch (IOException ex) {
            ex.printStackTrace();

        } finally {
            try {
                if (r != null ) {
                    r.close();
                }
            } catch (IOException ex) {
                ex.printStackTrace();
            }
        }
    }

}
```

W wymaganiach do naszego systemu mamy jeszcze zaplanowaną opcję zapamiętywania wybranych tłumaczeń. W ostatecznej klasie, która będzie realizować zadania postawione przed systemem, dodajmy dwie pomocnicze listy — jedna do przechowywania zapamiętanych słów, a druga do przechowywania wyników ostatniego zapytania. Ponadto dodajmy do naszego systemu klasę pomocniczą `DictionaryWord`, która będzie przechowywać dane odnośnie pojedynczego znalezionego tłumaczenia.

Ostateczną wersję klasy `WebDictionary` wraz z menu oraz klasy `DictionaryWord`, znajdziesz poniżej.

```
package pl.bnsit.webdictionary;

import java.util.Date;

public class DictionaryWord {

    private String polishWord;

    private String englishWord;

    private Date date;

    public DictionaryWord() {}

    public DictionaryWord(
                String polishWord, String englishWord, Date date) {
        this.polishWord = polishWord;
        this.englishWord = englishWord;
        this.date = date;
    }

    @Override
    public String toString() {
        return polishWord + " => " + englishWord;
    }

    public String getPolishWord() {
        return polishWord;
    }

    public void setPolishWord(String polishWord) {
        this.polishWord = polishWord;
    }

    public String getEnglishWord() {
        return englishWord;
    }

    public void setEnglishWord(String englishWord) {
        this.englishWord = englishWord;
    }

    public Date getDate() {
        return date;
    }

    public void setDate(Date date) {
        this.date = date;
    }
```

```
}
```

```
package pl.bnsit.webdictionary;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import java.util.Scanner;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class WebDictionary {

    public static void main(String[] args) {

        List<DictionaryWord> list1 = new ArrayList<DictionaryWord>();
        List<DictionaryWord> list2 = new ArrayList<DictionaryWord>();


        boolean ok = true;
        Scanner s = new Scanner(System.in);

        System.out.println("Welcome to Web Dictionary System.");

        while (ok) {
            System.out.print("dictionary> ");
            String c = s.nextLine();

            if (c.equals("hello")) {
                System.out.println(
                        "Welcome to Web Dictionary System.");

            } else if ( c.startsWith( "search" ) ) {

                list2.clear();

                BufferedReader r = null;
                String plW = null;
                String enW = null;
                int i = 1;

                try {
                    r = new BufferedReader(new InputStreamReader(
                            new URL( "http://www.dict.pl/dict?word="
```

```
                                + c.split(" ")[1] + "&words=&lang=PL" )
                                .openStream()));

                boolean polish = true;
                String t = null;
                Pattern pat = Pattern.compile(
                        ".*<a href=\"dict\\?words?=(.*)&lang.*" );
                while ( (t = r.readLine()) != null ) {

                    Matcher matcher = pat.matcher( t );
                    if (matcher.find()) {
                        String t2 = matcher.group(
                                matcher.groupCount() );
                        if ( polish ) {
                            System.out.print( i + ") " + t2
                                    + " => " );
                            plW = new String( t2.getBytes(), "UTF8" );
                            polish = false;
                        } else {
                            System.out.println( t2 );
                            polish = true;

                            enW = new String( t2.getBytes(), "UTF8" );
                            list2.add(
                                    new DictionaryWord(plW, enW,
                                            new Date()));
                            i++;
                        }
                    }
                }
            } catch (MalformedURLException ex) {
                ex.printStackTrace();
            } catch (IOException ex) {
                ex.printStackTrace();
            } finally {
                try {
                    if (r != null ) {
                        r.close();
                    }
                } catch (IOException ex) {
                    ex.printStackTrace();
                }
            }
        } else if ( c.startsWith( "save" ) ) {
            int num = new Integer( c.split(" ")[1] );
            list1.add( list2.get( num - 1 ) );
        } else if ( c.equals( "showFound" ) ) {
            int i = 1;
            for (DictionaryWord el : list2) {
                System.out.println( i++ + ") " + el );
```

```
                }
            } else if ( c.equals( "showSaved" ) ) {
                for (DictionaryWord el : list1) {
                    System.out.println( el );
                }
            } else if ( c.equals( "exit" ) ) {
                ok = false;
            }
        }
        s.close();
    }
}
```

Analizując powyższy kod, warto zauważyć kilka rzeczy. Program nie jest prawie wcale napisany obiektowo, gdyż cały zawiera się w metodzie `main` (wyjątkiem jest tu wprowadzona klasa `DictionaryWord`).

Ponadto przytoczony kod klasy `WebDictionary` i metody `main` to przykład antywzorca *Zbyt długa metoda*. **Odpowiedzialność metody `main` jest w tym momencie bardzo szeroka** - zajmuje się obsługą interfejsu użytkownika, obsługą zewnętrznego portalu słownikowego, przechowywaniem i przetwarzaniem danych tymczasowych. To wszystko w jednej metodzie. Taką metodę bardzo trudno się czyta. Trzeba ją całą przeanalizować, żeby zrozumieć, jakie zadania realizuje.

## Podstawowa zasada - wydzielanie odpowiedzialności

W celu refaktoryzacji metody `main`, zastanówmy się, z jakich głównych części się składa. Jak wyglądałby słowny zapis algorytmu? Być może w taki sposób:

1. inicjuj zmienne
2. przetwarzaj menu:
   (a) pobierz komendę od użytkownika;
   (b) jeśli wybrana komenda to *hello*, wypisz tekst powitalny;
   (c) jeśli wybrana komenda rozpoczyna się od słowa *search*, to pobierz dane ze strony, wypisz i zapamiętaj;
   (d) jeśli wybrana komenda to *showFound*, to wyświetl wyniki ostatniego wyszukiwania;
   (e) jeśli wybrana komenda to *showSaved*, to wyświetl zapamiętane słowa;

(f) jeśli wybrana komenda to *exit*, to zakończ program.

Dlaczego nie dążyć do tego, aby kod, który tworzymy, odpowiadał powyższemu zapisowi, wydzielając części algorytmu do osobnych metod.

**Ważne**

Być może znasz odpowiedź na pytanie, jak zjeść słonia. Dzieląc go na mniejsze kawałki ...

**Ważne**

Zbyt długie metody dziel na mniejsze, tak aby odzwierciedlały kroki algorytmu oraz stanowiły jego dokumentację poprzez odpowiednio dobrane nazwy.

## Refaktoryzacja: Wydzielenie metody

*Wydzielenie metody* to jedna z najważniejszych refaktoryzacji. Polega na przeniesieniu pewnego zestawu instrukcji do osobnej metody. Poprzez odpowiednią nazwę możemy sprawić, że powstanie samodokumentujący się kod.

Kiedy wydzielać nowe metody? Jedna z zasad mówi, że metoda powinna się mieścić na jednym ekranie. Obecnie ekrany mają różne wymiary i rozdzielczości, zatem jest to miara niejednoznaczna. Sugerowałbym zatem długość metod ograniczyć do mniej więcej 40 wierszy, gdyż średnio taki rozmiar jesteśmy w stanie jednorazowo objąć swoim wzrokiem i przeanalizować.

W klasie `WebDictionary` i metodzie `main` wydzielmy metody odpowiadające za:

- wyświetlanie komunikatu powitalnego (`printInfo()`),
- przetwarzanie menu (`processMenu()`),
- wyszukiwanie danych w słowniku internetowym (`searchWord()`),
- zapisanie wybranego słowa (`saveWord()`),
- wyświetlanie wyników ostatniego wyszukiwania (`showAll()`) i zapamiętanych słów (`showSaved()`).

Ponieważ pojawią się dodatkowe metody, które współdzielą dane, toteż listy przechowujące wyrazy uczynimy polami klasy.

Kod po refaktoryzacji wygląda następująco:

```
package pl.bnsit.webdictionary;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import java.util.Scanner;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class WebDictionary {

    private static List<DictionaryWord> list1
                            = new ArrayList<DictionaryWord>();
    private static List<DictionaryWord> list2
                            = new ArrayList<DictionaryWord>();

    public static void main(String[] args) {
        processMenu();
    }

    private static void processMenu() {
        boolean ok = true;
        Scanner s = new Scanner(System.in);

        printInfo();

        while (ok) {
            System.out.print("dictionary> ");
            String c = s.nextLine();

            if (c.equals("hello")) {
                printInfo();

            } else if ( c.startsWith( "search" ) ) {
                searchWord(c);

            } else if ( c.startsWith( "save" ) ) {
                saveWord(c);

            } else if ( c.equals( "showFound" ) ) {
```

```
                showFound();

            } else if ( c.equals( "showSaved" ) ) {
                showSaved();

            } else if ( c.equals( "exit" ) ) {
                ok = false;
            }
        }
        s.close();
    }

    private static void showSaved() {
        for (DictionaryWord el : list1) {
            System.out.println( el );
        }
    }

    private static void showFound() {
        int i = 1;
        for (DictionaryWord el : list2) {
            System.out.println( i++ + ") " + el );
        }
    }

    private static void saveWord(String c) {
        int num = new Integer( c.split(" ")[1] );
        list1.add( list2.get( num - 1 ) );
    }

    private static void searchWord(String c) {
        list2.clear();

        BufferedReader r = null;
        String plW = null;
        String enW = null;
        int i = 1;

        try {
            r = new BufferedReader(new InputStreamReader(
                    new URL( "http://www.dict.pl/dict?word="
                    + c.split(" ")[1] + "&words=&lang=PL" )
                    .openStream()));

            boolean polish = true;
            String t = null;
            Pattern pat = Pattern.compile(
                    ".*<a href=\"dict\\?words?=(.*)&lang.*" );
            while ( (t = r.readLine()) != null ) {
```

```
                Matcher matcher = pat.matcher( t );
                if (matcher.find()) {
                    String t2 = matcher.group(
                            matcher.groupCount() );
                    if ( polish ) {
                        System.out.print( i + ") " + t2
                                + " => " );
                        plW = new String( t2.getBytes(), "UTF8" );
                        polish = false;
                    } else {
                        System.out.println( t2 );
                        polish = true;

                        enW = new String( t2.getBytes(), "UTF8" );
                        list2.add(
                                new DictionaryWord(plW, enW,
                                        new Date()));
                        i++;
                    }
                }
            }
        } catch (MalformedURLException ex) {
            ex.printStackTrace();

        } catch (IOException ex) {
            ex.printStackTrace();

        } finally {
            try {
                if (r != null ) {
                    r.close();
                }
            } catch (IOException ex) {
                ex.printStackTrace();
            }
        }
    }

    private static void printInfo() {
        System.out.println(
                "Welcome to Web Dictionary System.");
    }
}
```

Widzimy kilka pozytywnych efektów tego działania:

1. kod nie jest już monolityczny

2. powstało kilka mniejszych metod, które są łatwiejsze w zrozumieniu

3. nazwy nowych metody dokumentują ich odpowiedzialność
4. osoba analizująca kod może tylko i wyłącznie na podstawie analizy metody `processMenu()` zrozumieć główną myśl algorytmu
5. jeśli czytelnik kodu chce poznać szczegóły danego kroku algorytmu, może przeanalizować konkretną metodę
6. kod zaczyna się czytać jak książkę

...ciągle pozostaje kilka wad. Wszystkie metody są statyczne, co w konsekwencji oznacza, iż program jest nieobiektowy. Elementy statyczne są związane z klasami, mają zatem charakter globalny względem obiektów. Rozwiązanie, które stworzyliśmy wpisuje się w konwencję programowania proceduralnego.

Zróbmy mały krok, który uczyni nasz program bardziej obiektowym i zbuduje podwaliny dla dalszego rozwoju systemu - wydzielone metody oraz atrybuty uczyńmy niestatycznymi, zaś w metodzie `main` stworzymy obiekt klasy `WebDictionary`. Efekt tej zmiany możesz obejrzeć na listingu zamieszczonym pod koniec następnego podrozdziału.

**Ważne**

Unikaj elementów statycznych. Są nieobiektowe, gdyż mają charakter globalny.
W szczególności w odniesieniu do metod statycznych warto pamiętać, że w ich przypadku nie działa mechanizm polimorfizmu.

## Refaktoryzacja: Zmiana nazwy klasy, metody, atrybutu lub zmiennej

Przed nami kolejny ważny krok w stronę czytelnego kodu - zmiana nazw metod, atrybutów i zmiennych. Jak już wcześniej wspominałem, nazwy odzwierciedlają odpowiedzialność elementu, dlatego warto poświęcić im część swojej energii. Jest to jeden z najważniejszych nośników informacji w tworzonym kodzie.

Spójrzmy na przykładowy kod źródłowy. Jak odróżnimy atrybuty `list1` i `list2`? Na pierwszy rzut oka nie ma niczego, co by podpowiedziało, jaka intencja przyświecała autorowi. W konsekwencji oznacza to, że będziemy musieli przeanalizować kod i

zapamiętać różnicę między jedną listą a drugą. Dodatkowy wysiłek. Jeśli takich przypadków w tworzonym kodzie mamy kilka . . . kilkanaście . . . kilkaset - łatwo można sobie wyobrazić, jak trudno będzie zrozumieć i przeanalizować taki kod.

Miłośnicy komentarzy mogą mieć ochotę zmodyfikować nieco kod na przykład w taki sposób:

```
// saved words
private List<DictionaryWord> list1
                        = new ArrayList<DictionaryWord>();
// last search words
private List<DictionaryWord> list2
                        = new ArrayList<DictionaryWord>();
```

Jednak taki komentarz niewiele zmienia:

- komentarz występuje w miejscu deklaracji, zatem nie widać go najczęściej w miejscu, gdzie dany element jest używany;
- jeśli zmieni się nieco odpowiedzialność, łatwo zapomnieć o zmianie komentarza;
- komentarze są dla ludzi - nie są analizowane ani interpretowane przez kompilator, nie istnieją narzędzia, które pozwalają automatycznie nimi zarządzać (np. zmieniać powiązane komentarze występujące w wielu miejscach systemu).

A czy nie prościej byłoby zmienić nazwy zmiennych w następujący sposób?

```
private List<DictionaryWord> savedWords
                        = new ArrayList<DictionaryWord>();
private List<DictionaryWord> lastSearchWords
                        = new ArrayList<DictionaryWord>();
```

Zauważmy, że uzyskujemy ten sam efekt, który się pojawił w przypadku komentarza, lecz tym razem to *nazwa pola jest komentarzem*. Komentarz ten pojawia się nie tylko w miejscu deklaracji, ale również tam, gdzie pole jest używane:

```
private void showSaved() {
    for (DictionaryWord el : savedWords) {
        System.out.println( el );
    }
}

private void showFound() {
    int i = 1;
    for (DictionaryWord el : lastSearchWords) {
```

```
            System.out.println( i++ + ") " + el );
        }
    }

    private void saveWord(String c) {
        int num = new Integer( c.split(" ")[1] );
        savedWords.add( lastSearchWords.get( num - 1 ) );
    }
```

W momencie, kiedy stwierdzimy, że odpowiedzialność nie została precyzyjnie wyrażona poprzez nazwę, możemy zmienić nazwę, zaś automatyczne narzędzia dokonają zmian w każdym miejscu w projekcie, gdzie taka nazwa występuje[1].

W przytoczonym przykładzie pojawia się wiele przypadków, w których nazwy zmiennych niewiele mówią, są skrótowe, nie odzwierciedlają znaczenia i utrudniają zrozumienie kodu. Nazwa `ok` jest lakoniczna, może oznaczać niemalże wszystko, podobnie z nazwami `c`, `s`, `el`, `plW`, `enW`, `t`, `r` .... Co się dzieje kiedy nagle w kodzie dostrzegamy wiersz

```
        s.close();
```

Albo musimy pamiętać co jest przechowywane pod nazwą `s`, albo musimy odnaleźć miejsce deklaracji pola, zmiennej lub parametru. W obu przypadkach jest to dodatkowy wysiłek. Jeśli takich zmiennych w niewielkim fragmencie kodu jest kilkanaście, każdy przypadek kosztuje dodatkową energię, którą musimy na niego wydatkować.

Czy zastanawiałeś się, dlaczego po kilku minutach czytania kodu, stajesz się bardzo zmęczony?

Chciałbym podkreślić, że nazwy klas, pól, metod, zmiennych są jak awers monety, której rewersem jest odpowiedzialność. Nazwy muszą być jednoznaczne, możliwie krótkie, ale na tyle długie, aby oddać sens i odpowiedzialność nazywanego elementu. Często mam wrażenie, że programiści boją się nadawać nieco dłuższe nazwy swoim klasom lub zmiennym, ale cóż za zysk z napisania `String w = "slowo";` zamiast `String word = "slowo";`, `list1` zamiast `savedWords`, `list2` zamiast `lastSearchWords` itd. Wiele współczesnych środowisk programistycznych podpowiada nazwy zmiennych, klas i pól, zatem i tak najczęściej nie wpisujemy ich ręcznie. Krótko mówiąc, na krótkich nazwach nie oszczędzamy.

[1] Obecnie wiele środowisk programistycznych (np. Eclipse, NetBeans, Visual Studio) wspiera proces refaktoryzacji i dzięki czemu wielu zmia można dokonywać używając automatów, które zadbają o wprowadzane zmiany.

Poniżej znajduje się analizowany przykład `WebDictionary`, tym razem już ze zmodyfikowanymi nazwami zmiennych:

```
package pl.bnsit.webdictionary;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import java.util.Scanner;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class WebDictionary {

    private List<DictionaryWord> savedWords
                            = new ArrayList<DictionaryWord>();
    private List<DictionaryWord> lastSearchWords
                            = new ArrayList<DictionaryWord>();



    public static void main(String[] args) {
        WebDictionary webDictionary = new WebDictionary();
        webDictionary.start();
    }

    public void start() {
        processMenu();
    }

    private void processMenu() {
        boolean running = true;
        Scanner scanner = new Scanner(System.in);

        printInfo();

        while (running) {
            System.out.print("dictionary> ");
            String command = scanner.nextLine();

            if (command.equals("hello")) {
                printInfo();

            } else if ( command.startsWith( "search" ) ) {
```

```
                searchWord(command);

            } else if ( command.startsWith( "save" ) ) {
                saveWord(command);

            } else if ( command.equals( "showFound" ) ) {
                showFound();

            } else if ( command.equals( "showSaved" ) ) {
                showSaved();

            } else if ( command.equals( "exit" ) ) {
                running = false;
            }
        }
        scanner.close();
    }

    private void showSaved() {
        for (DictionaryWord dictionaryElement : savedWords) {
            System.out.println( dictionaryElement );
        }
    }

    private void showFound() {
        int counter = 1;
        for (DictionaryWord dictionaryElement : lastSearchWords) {
            System.out.println( counter++ + ") " + dictionaryElement );
        }
    }

    private void saveWord(String command) {
        int wordNumber = new Integer( command.split(" ")[1] );
        savedWords.add( lastSearchWords.get( wordNumber - 1 ) );
    }

    private void searchWord(String command) {
        lastSearchWords.clear();

        BufferedReader bufferedReader = null;
        String polishWord = null;
        String englishWord = null;
        int counter = 1;

        try {
            bufferedReader = new BufferedReader(new InputStreamReader(
                    new URL( "http://www.dict.pl/dict?word="
                    + command.split(" ")[1] + "&words=&lang=PL" )
                    .openStream()));
```

```
            boolean polish = true;
            String line = null;
            Pattern pat = Pattern.compile(
                    ".*<a href=\"dict\\?words?=(.*)&lang.*" );
            while ( (line = bufferedReader.readLine()) != null ) {

                Matcher matcher = pat.matcher( line );
                if (matcher.find()) {
                    String foundWord = matcher.group(
                            matcher.groupCount() );
                    if ( polish ) {
                        System.out.print( counter + ") " + foundWord
                                + " => " );
                        polishWord
                            = new String( foundWord.getBytes(), "UTF8" );
                        polish = false;
                    } else {
                        System.out.println( foundWord );
                        polish = true;

                        englishWord
                            = new String( foundWord.getBytes(), "UTF8" );

                        lastSearchWords.add(
                                new DictionaryWord(polishWord,
                                        englishWord, new Date()));
                        counter++;
                    }
                }
            }
        } catch (MalformedURLException ex) {
            ex.printStackTrace();

        } catch (IOException ex) {
            ex.printStackTrace();

        } finally {
            try {
                if (bufferedReader != null ) {
                    bufferedReader.close();
                }
            } catch (IOException ex) {
                ex.printStackTrace();
            }
        }
    }

    private void printInfo() {
        System.out.println(
                "Welcome to Web Dictionary System.");
```

```
    }
}
```

## Refaktoryzacja: Wydzielenie zmiennej tymczasowej

Kod naszego systemu tłumaczącego poddany refaktoryzacji zawiera coraz więcej elementów, które powodują, że metody są coraz bliższe rozumowania człowieka. Stosując kilka prostych zabiegów kod staje się bardziej samodokumentujący. Na co jeszcze możemy zwrócić uwagę? Przyjrzyjmy się następującemu fragmentowi:

```
    bufferedReader = new BufferedReader(new InputStreamReader(
            new URL( "http://www.dict.pl/dict?word="
            + command.split(" ")[1] + "&words=&lang=PL" )
            .openStream()));
```

Dzieje się tu wiele rzeczy jednocześnie. Powstają trzy nowe obiekty, następuje konkatenacja napisów. Jeden z członów konkatenacji to złożone wyrażenie `command.split("")[1]` wyłuskujące wyraz do wyszukiwania. Aby zrozumieć ten fragment trzeba poświęcić sporo czasu (kilka lub kilkanaście sekund). Dokonajmy prostych zmian w kilku krokach, stosując tę samą zasadę, która towarzyszyła refaktoryzacji *Wydzielenie zmiennej* - podzielmy konstrukcje na mniejsze składowe. Zacznijmy od ciągu napisowego:

```
            String urlString = "http://www.dict.pl/dict?word="
                + command.split(" ")[1] + "&words=&lang=PL";

            bufferedReader = new BufferedReader(new InputStreamReader(
                    new URL( urlString ).openStream()));
```

Atmosfera wokół tworzenia obiektu klasy `BufferedReader` już nie jest tak zagęszczona. Nie jest skondensowana w jednym wierszu - łatwiej ją zrozumieć, tym bardziej, że wprowadziliśmy niejawnie komentarz w formie nazwy zmiennej `urlString`. Cały czas jednak konstrukcja `command.split("")[1]` jest bardzo tajemnicza. Być może domyślasz się o co chodziło autorowi, ale można tę itencję wyrazić dużo jaśniej.

```
            String wordToFind = command.split(" ")[1];

            String urlString = "http://www.dict.pl/dict?word="
                + wordToFind + "&words=&lang=PL";

            bufferedReader = new BufferedReader(new InputStreamReader(
```

```
                    new URL( urlString ).openStream()));
```

Ciekawe czy jesteś w stanie zrozumieć ten fragment kodu, w sytuacji, gdy nie znasz dobrze efektu działania metody `split`. Oczywiście nic nie stoi na przeszkodzie, żeby dociekliwie zbadać, jak dokładnie przebiega wykonanie konstrukcji `command.split(" ")[1]`. Można by się jednak pokusić o wprowadzenie do kodu dodatkowej informacji:

```
            String[] commandParts = command.split(" ");
            String wordToFind = commandParts[1];
```

Teraz sprawa jest jeszcze prostsza, gdyż zmienna `commandParts` informuje o tym, co jest w tym przypadku efektem działania metody `String.split`.

**Ważne**

Tam gdzie może to uprościć analizę kodu, wydzielaj zmienne, które podzielą złożone konstrukcje na mniejsze i udokumentują program.

Wydzielając zmienne należy kierować się zdrowym rozsądkiem. Nie chciałbym, abyś Czytelniku odniósł wrażenie, że każda składowa złożonego wyrażenia powinna zostać wydzielona. Często nie ma to większego sensu. Na przykład we fragmencie

```
System.out.print( counter + ") " + foundWord + " => " );
```

wprowadzenie pomocniczych zmiennych nic by nie wniosło do kodu. Wręcz mogłoby zmniejszyć jego czytelność. Należy wydzielać zmienne tam, gdzie złożoność wyrażenia utrudnia jego analizę. Jest to szczególnie widoczne podczas poszukiwania przyczyn wyjątków lub błędów. Jeśli w wierszu o postaci:

```
    bufferedReader = new BufferedReader(new InputStreamReader(
            new URL( "http://www.dict.pl/dict?word="
            + command.split(" ")[1] + "&words=&lang=PL" )
            .openStream()));
```

pojawi się wyjątek, nie będzie oczywistą sprawą, gdzie leży jego przyczyna. Jeśli zaś ten wiersz rozbijemy na kilka części i wydzielimy zmienne, to w momencie pojawienia się wyjątku lub błędu, będzie oczywiste, które wyrażenie spowodowało jego wystąpienie.

**Ważne**

Wydzielanie zmiennych ze złożonych wyrażeń może ułatwić odnajdywanie przyczyn wystąpienia wyjątków lub błędów.

## Nazywanie warunków

Szczególną odmianą refaktoryzacji *Wydzielenie zmiennej tymczasowej* lub *Wydzielenie metody* jest *Nazywanie warunków*. Przyjrzymy się wierszowi:

```
String line = null;
while ( (line = bufferedReader.readLine()) != null ) {
```

Dzieje się tu wiele rzeczy jednocześnie - jest przypisanie wartości, jest sprawdzenie warunku. To trudny wiersz do analizy. W tym przypadku nie udało mi się znaleźć jednego prostego przejścia, dlatego proponuję następujące zmiany:

```
// ...
        String line = bufferedReader.readLine();
        while ( hasNextLine( line ) ) {

            // ...

            line = bufferedReader.readLine();
        }

// ...

private boolean hasNextLine(String line) {
    return ( line != null );
}
```

Pojawiła się pomocnicza metoda `hasNextLine` oraz odczyt kodu HTML został przesunięty z warunku pętli do osobnych wierszy. Pojawia się w zamian pewna nadmiarowość (dwukrotnie w kodzie występuje odczyt `line = bufferedReader.readLine();`).

Chciałbym zwrócić uwagę w tym przykładzie na metodę `hasNextLine`, gdyż ona jest tu głównym bohaterem. Często w kodach źródłowych pojawiają się skomplikowane konstrukcje definiujące warunki wykonywania przepływu. Są one trudne w interpretacji i rzadko wyrażają bezpośrednio intencję autora. Taki warunek lepiej zapisać w postaci nowej zmiennej pomocniczej (refaktoryzacja *Wydzielenie zmiennej tymczasowej*) lub nowej metody pomocniczej (refaktoryzacja *Wydzielenie metody*). Dzięki

temu posunięciu dodatkowo dokumentujemy warunek i jest on prostszy w interpretacji. Zapis `hasNextLine` zawiera informację o tym, jaka była intencja autora, czego z pewnością w sposób bezpośredni nie zawiera konstrukcja `(line = bufferedReader.readLine()) != null)`.

Innym przykładem (który nie występuje w realizowanym w tej książce projekcie) może być sprawdzenie, czy ciąg znakowy zawiera przynajmniej jeden znak niepusty.

```
String text = readFromFile( filename );

if ( text != null && text.trim().equals("") == false ) {
    System.out.println( text );
}
```

Oczywiście każdy wprawiony programista języka Java rozszyfruje ten zapis po pewnym czasie, jednak potrzebna będzie na to dodatkowa energia.

```
String text = readFromFile( filename );

boolean textNotEmpty
    = (text != null && text.trim().equals("") == false);
if ( textNotEmpty ) {
    System.out.println( text );
}
```

Warunek `if ( textNotEmpty )` jest napisany niemalże ludzkim językiem. Programista może oczywiście sprawdzić, co kryje się za nazwą `textNotEmpty`, ale tylko wtedy, kiedy jest mu to potrzebne. Nie musi analizować warunku, aby zrozumieć dany fragment kodu.

**Ważne**

Używaj nazwanych warunków, aby w prosty i bezpośredni sposób wyrazić intencję związaną z danym fragmentem kodu.

Często się zdarza, że pod konkretną postacią warunku może stać intencja, którą warto zapisać. W jednym z szkieletów aplikacyjnych do tworzenia aplikacji sieciowych w języku Java, używa się często warunku w postaci:

```
if ( person.getId() != null ) {
    // ...
} else {
    // ...
}
```

Sprawa niby prosta i oczywista. Z tym, że za zapisem `person.getId() != null` stoi pewna intencja. Warunek ten służy do sprawdzenia, czy w danym momencie następuje edycja istniejącego obiektu `person.getId() != null` czy też tworzenie nowego obiektu `person.getId() == null`. Można by powyższy kod zapisać w następująco:

```
boolean editMode = ( person.getId() != null );
if ( editMode ) {
    // ...
} else {
    // ...
}
```

W ten sposób używając refaktoryzacji *Wydzielenie zmiennej tymczasowej* rozjaśniliśmy intencję stojącą za pozornie dość oczywistym warunkiem. Refaktoryzacja służy przede wszystkim temu, aby osoba, która będzie czytała dany fragment kodu za kilka tygodni lub miesięcy, w ciągu możliwie krótkiego czasu mogła zorientować się, co jest intencją danego fragmentu kodu oraz które miejsce należy zmodyfikować, aby wprowadzić zmiany.

Dokonując zmian w systemie, dotarliśmy do bardziej uporządkowanego rozwiązania. Cały czas aplikacja nie ma zbyt wielu znamion kodu obiektowego. I bardzo dobrze! Oznacza to, że przedstawione do tej pory refaktoryzacje mogą być stosowane z powodzeniem w językach proceduralnych.

Głównym mankamentem rozwiązania, nad którym pracujemy, jest to, że metoda `searchWord` cały czas pozostaje dość złożona, trudna w zrozumieniu i nieprzygotowana na rozbudowywanie . Jednak kolejne zmiany zostawimy do następnego rozdziału.

## Złote reguły refaktoryzacji

Powtórzmy na koniec podstawowe elementy, które w prosty sposób mogą spowodować, iż kod będzie bardziej samodokumentujący się i że będzie się go czytało jak książkę.

1. przeanalizuj odpowiedzialność swoich klas, pól w klasie, metod i zmiennych; analiza ta jest bazą do wprowadzanych zmian;
2. jeśli odpowiedzialność jest zbyt duża podziel te elementy na składowe (poprzez *Wydzielenie metody* lub *Wydzielenie zmiennej*);
3. jeśli odpowiedzialność jest zbyt mała (co dotyczy głównie klas i czasami metod), połącz je;

4. upewnij się, że nazwy definiują jednoznacznie odpowiedzialność klasy, metody, zmiennej lub pola; jeśli nie - zmień je najszybciej jak to możliwe; nazwy to główne nośniki informacji w twoim kodzie;

5. warunki wydzielaj do osobnych zmiennych lub metod, aby bezpośrednio i jednoznacznie wyrazić intencję, która za nimi stoi;

6. ciesz się kodem, który jest czytelny.

## Kod trudny w testowaniu prawdopodobnie nadaje się do refaktoryzacji

Jak już wspominałem, testowanie jest bardzo ważnym uzupełnieniem procesu refaktoryzacji - jest ono niezbędne, aby mieć pewność, iż wprowadzane zmiany rzeczywiście nie zmieniają obserwowalnego zachowania.

Jest jeszcze jeden powód, który sprawia, że testowanie jest takie ważne.

**Ważne**

Jeśli chcesz przetestować fragment swojego kodu i wydaje ci się to trudne a nawet niemożliwe, prawdopodobnie kod wymaga refaktoryzacji.

Często miałem okazję brać udział w sytuacji, która przebiegała następująco. Razem z innym programistą analizuję jego kod oraz testy do kodu. Zauważam, że pewien fragment w ogóle nie jest przetestowany.
- Dlaczego nie ma testu do tej metody?
Konsternacja na twarzy mojego rozmówcy.
- Bo wiesz . . . tego się nie da przetestować . . .
Zazwyczaj patrzę wtedy z niedowierzaniem i zaczynamy analizować rozwiązanie. Po pewnym czasie wspólnie już stwierdzamy, że tego kodu się nie da przetestować. Jest po prostu nienajlepiej napisany i wymaga refaktoryzacji. Na przykład trudno przetestować długą metodę, gdyż zazwyczaj zawiera kilkanaście skomplikowanych scenariuszy, do których należy przygotować często złożone dane wejściowe i spodziewane wyjściowe. Podzielenie metody na mniejsze części powoduje, że możemy testować mniejsze fragmenty, co jest zazwyczaj dużo prostsze.

Dlatego uważam, że każdy programista powinien znać techniki testowania, nawet

jeśli sam testowaniem się nie zajmuje. Spojrzenie na kod z punktu widzenia testera pozwala odkryć jego słabe punkty.

**Uwaga!**

W tym miejscu kończy się udostępniana bezpłatnie część książki. Jeśli temat cię zaciekawił, jeśli chciałbyś poznać go głębiej zapraszamy na naszą stronę http://www.bnsit.pl, aby dowiedzieć się więcej. Możesz nabyć pełną, licencjonowaną wersję tej książki w formie elektronicznej, jak i papierowej oraz warsztaty, które stanowią multimedialną formę przekazania doświadczenia związanego z refaktoryzacją.

**Uwaga!**

Jeśli chciałbyś współtworzyć tę książkę np. przygotowując wersję w innym języku programowania, albo masz pomysł na nową książkę, napisz na adres ebooks@bnsit.pl